Master's thesis

NTNU
Norwegian University of Science and Technology
Faculty of Natural Sciences
Department of Chemical Engineering

Abubakar Bampoye

An Comparative Study:

Model-Based vs. Noise Estimation Approach for State Estimation

Master's thesis in Chemical Engineering & Biotechnology
Supervisor: Johannes Jäschke
Co-supervisor: Halvor Aarnes Krog
August 2023

NTNU
Norwegian University of
Science and Technology

Abubakar Bampoye

An Comparative Study:

Model-Based vs. Noise Estimation Approach for State Estimation

Master's thesis in Chemical Engineering & Biotechnology
Supervisor: Johannes Jäschke
Co-supervisor: Halvor Aarnes Krog
August 2023

Norwegian University of Science and Technology
Faculty of Natural Sciences
Department of Chemical Engineering

NTNU
Norwegian University of
Science and Technology

Contents

List of Algorithms

List of Figures

List of Python modules

List of Tables

Abstracts

This thesis delves into an analysis conducted within the context of a bioprocess model. It involves a comparative investigation of two distinct estimation techniques, namely those proposed Krog & Jäschke[21] and Berry & Saue[8]. These techniques are categorized as model-based and data-driven approaches, respectively. Their primary objective is to eliminate the need for manual tuning of the covariance matrices **Q** and **R**, which play a crucial role as essential components of the Kalman filter, and crucial for controlling filter divergence. By enhancing the applicability of the filters, these estimation techniques contribute to improved performance and efficacy in state estimation.

Based on the results gathered, it is evident that the estimation technique presented by Krog & Jäschke[21], which utilizes the scaled unscented transformation to estimate the covariance matrix **Q**, yielded the most favorable outcomes, as determined by the Root Mean Squared Error (RMSE).

Sammendrag

Denne oppgaven tar for seg en analyse av en bioprosessmodell, som omfatter en sammenlignende undersøkelse av to forskjellige estimeringsteknikker. Selve estimeringsteknikkene er foreslått av Krog & Jäschke[21] og Berry & Saue[8]. Disse teknikkene er klassifisert som henholdsvis modellbasert og datadrevet. Hovedmålet deres er å eliminere behovet for manuell tilpasning av kovariansmatrisene **Q** og **R**, som spiller en avgjørende rolle som essensielle komponenter i Kalman-filteret, noe som er avgjørende for å kontrollere filterdivergens. Ved å forbedre anvendeligheten av filtrene bidrar disse estimeringsteknikkene til bedret ytelse og effektivitet i tilstands-estimering.

Basert på de innhentede resultatene er det klart at estimeringsteknikken presentert av Krog & Jäschke[21], som bruker den skalerte uscented transformasjonen for å estimere kovariansmatrisen **Q**, ga de mest gunstige resultatene, bestemt ved hjelp av Root Mean Squared Error (RMSE).

Acknowledgements

I would like to express my sincere gratitude to my co-supervisor, Halvor Aarnes Krog, for his invaluable support, encouragement, and genuine interest in my project. His diligent efforts in double and triple-checking the equations and Python modules, as well as his generous dedication of time, with weekly meetings throughout the entire semester, was a critical inspiration and motivation for me.

I would like to extend my heartfelt appreciation to my friends, whose companionship and camaraderie have transformed my five-year at the university into a wonderful journey, encompassing both social and academic aspects. Moreover, I wish to express my deep gratitude to my beloved family, especially my elder sister, Assia Bigirimana, whose unwavering kindness and support have been a constant pillar throughout my studies.

Preface

This thesis is submitted to fulfill the requirements for the degree in Chemical Engineering and Biotechnology at the Norwegian University of Science and Technology (NTNU). The thesis represents a continuation of a specialization project conducted during the autumn of 2022.

With the exception of explicitly indicated instances, all figures and illustrations presented in this thesis have been exclusively generated by the author. The simulation was executed using the Python programming language, and the utilization of the built-in functions and libraries was used to reduce the need for extensive manual programming.

List of abbreviations

EKF	Extended Kalman filter
GenUKF	Generalized unscented Kalman filter
KF	Kalman filter
PDF	Probability density functions
RMSE	Root mean square error
RV	Random variable
Scaled UKF	Scaled unscented Kalman filter
UKF	Unscented Kalman filter
UT	Unscented transformation
WSLR	Weighted statistical linear regression

1 Introduction

1.1 Motivation and problem statement

Over the past decade, model-based predictive control has become increasingly popular in chemical processing. These models require an estimate of all the dependent variables of the model, referred to as state variables, $\mathbf{x}(t)$, that describe the current state of the process[2, 3, 17, 22, 23, 26]. By combining the available output measurements, $\mathbf{y}(t)$, the process model, and the inputs, $\mathbf{u}(t)$, of the process to estimate the optimal state variables, $\hat{\mathbf{x}}$, we can obtain a more complete picture of the system's behaviour, and use this information to make more informed control decisions. Thus estimation is essential for gaining a deeper understanding of the inner workings of the process, which in turn facilitates the design of effective multivariable and nonlinear control strategies[28]. Overall, accurate estimation of the state variables is a critical component of modern control strategies that can lead to improved performance, greater efficiency, enhanced process stability, and condition monitoring.

State estimators need dynamic models of the system so that we can make predictions about the future[34]. Traditionally, the mathematical models used in predictive control techniques are developed from physics, chemistry, and biology (first principles), empirical models (data regression), or a mix between these two (hybrid models). However, a model will never exactly match the plant (plant-model mismatch). The mismatch is either a structural or parametric deviation from the process. The uncertainty in the model can be interpreted as the process noise, and uncertainty in the measurement is interpreted as measurement noise. State estimation assumes knowledge of the mean and covariance of the noise at each time step. This is an unrealistic assumption, and the specification of the noise statistics is therefore often considered as tuning parameters of the state estimation which greatly affects the performance of the state estimation. Often, the noise statistics are manually tuned.

This is laborsome and often results in sub-optimal performance of the filter. There exist, however, structural methods to tune the state estimator. In this work, we compare two methods, a model-based and data-driven.

The model-based is a method that accounts for uncertainty in the parameters and relies on the current state $\mathbf{x}_k$ and control variable $\mathbf{u}_k$. This yields time-varying estimates of $\mathbf{w}_k$, which is the process noise. On the other hand, the data-driven method utilizes measurements $\mathbf{y}_k$. It typically assumes that the noise statistics are stationary, meaning they do not change over time. We expect the data-driven method to perform well on continuous processes. However, it is possible that the data-driven method may not perform as effectively on batch processes, as it usually assumes stationary $\mathbf{w}_k$. Therefore, we intend to conduct a test with these two methods on a fed-batch bioreactor.

In order to achieve precise predictions, these models need to be initiated from the true initial condition, which is inherently unattainable[1]. When dealing with large-scale systems, obtaining a comprehensive experimental measurement of the system's complete state at a specific point in time is practically infeasible. Even in the hypothetical scenario where such measurements could be obtained, the measurements are always contaminated by noise, reducing the fidelity of our estimations. Lastly, the mathematical model that fully characterizes the underlying processes and dynamics of the system is either undisclosed or inherently challenging to handle. As a consequence, approximation and simplified models are adopted instead.

To mitigate these challenges, or at least their effects. State estimators can be used to estimate unmeasured states and the impact of disturbances. The goal of state estimation is to make optimal use of available information from the process model and measurements, in order to estimate the unmeasured states of the dynamic system. This information can then be utilised to monitor and control the process effectively[5].

1.2 Literature review of noise estimation methods

With the progress in measurement systems and the decreasing costs of sensors, recursive stochastic state estimation techniques, such as the Kalman filter, have become more applicable and very popular in systems engineering, robotics, navigation, and control[1]. They have traditionally been used for state estimation in many chemical processes, such as chemical reaction systems, polymerisation processes, and bioreactors[11, 29, 35]. The linear Kalman filter is used to estimate states based on linear dynamical systems, while the extended Kalman filter (EKF) is used to estimate nonlinear dynamical systems (in recent years, UKF). These state estimators will be explained in greater detail in Section 3. The standard equations provided originally by Kalman[15] are provably optimal for systems where the dynamics and observations are linear with Gaussian noise. If the covariance matrices $\mathbf{Q}_k$ and $\mathbf{R}_k$ referred to as process noise and measurement noise covariances, respectively, are known on each time step k, the Kalman filter gives the maximum likelihood estimate of the current state. If inexact $\mathbf{Q}_k$ and $\mathbf{R}_k$ are used in the Kalman filter, the filter will be suboptimal and may give reasonable state estimates[4]. Normally one assumes that $\mathbf{Q}$ is constant (i.e. $\mathbf{Q}_k = \mathbf{Q}_{k-1} = \cdots = \mathbf{Q}$). In the context of nonlinear processes, there exist estimators of higher precision, such as particle filters, moving horizon techniques, and modified versions of Kalman filters specifically designed for nonlinear systems[4, 6, 36].

In order to use Kalman filters, it's necessary to specify the level of accuracy for both the model and the measurements. This is done through the process noise and measurement noise covariances, $\mathbf{Q}$ and $\mathbf{R}$, respectively. The performance of the filter heavily relies on accurately specifying the levels of noise, as incorrect specifications can cause the filter to diverge. The measurement covariance matrix, $\mathbf{R}$, can be derived directly from the properties of the measurement instrument's accuracy, while the specification of $\mathbf{Q}$ is often achieved through a trial-and-error method. For a system with n states and additive process noise, the specification of $\mathbf{Q}$ requires $n(n+1)/2$ elements, considering it is a symmetric matrix. However, for time-varying processes such as batch and semi-batch,

or for nonlinear systems, a constant matrix **Q** may not be sufficient to provide accurate filter performance.

As previously mentioned, there are unavoidable differences between process models and the actual process due to the attempt to fit a simple model into a complex process. For instance parameters that exhibit slowly time-varying behavior, such as heat transfer coefficients. While they remain constant within a short time window, they may change when, for instance, fouling occurs on a heat exchanger[27]. This can lead to inaccurate state estimates being calculated. Additionally, disturbances that occur in the process but are not measured can also affect the estimates. One possible solution to this problem is to use a parameter-adaptive Kalman filter[32]. This type of filter includes non-stationary stochastic states in the estimation process, in addition to the original system states. While this approach does not solve the problem of determining the correct levels of process and measurement noise, it can help reduce errors in the state estimates.

Different methods have been discussed by several authors for approximating the correct selection of **Q** and **R**. Model-based methods, as described by Valappil & Georgakis and Krog & Jäschke[21, 34], as well as data-driven methods by Dunik, Berry & Saue[7, 8], are among them. In this thesis, we continue this discussion by exploring the approaches presented by Krog & Jäschke, as well as Berry & Saue. Both of these approaches involve online calculations and utilize current states from the filter. To apply these techniques, it is essential to have a well-defined process model and a quantification of the model's uncertainty through a parameter covariance matrix obtained during model development. The two proposed methods calculate time-varying values of the process noise covariance matrix, **Q**, online, which are then incorporated into the filter.

1.3 Outline

The following chapters will delve into the theoretical background, followed by an overview of Kalman filters. Subsequently, we will provide a description of the estimation of noise covariances. Moving on, we present our case study involving a bioprocess that has been investigated by Andrea Tuveri, Fernando Perez-Garcia, Pedro A. Lira-Parada, Lars Imsland, & Nadav Bar[33]. The system is described by four state variables, which represent the volume, V, biomass, X, sugar consumption, S, and carbon dioxide, CO_2. We then proceed with the obtained results, followed by a discussion and conclusion to round off the study.

2 Theoretical background

This chapter serves as the foundational framework for the subsequent implementations presented in this thesis. By providing this background information, readers will gain a comprehensive understanding of the equations and algorithms discussed in the subsequent chapters.

2.1 System description of linear systems

The presented expression embodies a general continuous state-space representation of a linear system:

$$\dot{\mathbf{x}}(t) = \mathbf{F}\mathbf{x}(t) + \mathbf{G}\mathbf{u}(t) + \mathbf{w}(t) \tag{2.1}$$

$$\mathbf{y}(t) = \mathbf{H}\dot{\mathbf{x}}(t) + \mathbf{v}(t) \tag{2.2}$$

where $\mathbf{x} \in \mathbb{R}^n$ is the state vector, $\mathbf{u} \in \mathbb{R}^p$ is the control (input) vector, $\mathbf{y} \in \mathbb{R}^q$ is the measurement (output) vector. The matrices $\mathbf{F}$, $\mathbf{G}$ and $\mathbf{H}$ are appropriately dimensioned matrices called the system (transition) matrix, control (input) matrix, and measurement (output) matrix, respectively[30]. We assume that the process and measurement noise are uncorrelated zero-mean Gaussian random process, i.e:

$$\mathbb{E}[\mathbf{w}(t)] = \mathbb{E}[\mathbf{v}(t)] = \mathbf{0} \tag{2.3}$$

$$\mathbb{E}[\mathbf{w}(t)\mathbf{w}^T(\tau)] = \mathbf{Q}(t)\delta(\mathbf{w}(t-\tau)) \tag{2.4}$$

$$\mathbb{E}[\mathbf{v}(t)\mathbf{v}^T(\tau)] = \mathbf{R}(t)\delta(\mathbf{v}(t-\tau)) \tag{2.5}$$

$$\mathbb{E}[\mathbf{w}(\tau)\mathbf{v}^T(t)] = \mathbf{0} \tag{2.6}$$

with probability density functions $p_{\mathbf{w}}(\mathbf{w})$ and $p_{\mathbf{v}}(\mathbf{v})$, and with unknown covariance matrices $\mathbf{Q} \in \mathbb{R}^{n\times n}$, $\mathbf{R} \in \mathbb{R}^{q\times q}$ respectively. $\mathbb{E}$, is the expectation operator, and δ is the Dirac delta-function. Further, we address a general discrete time of the linear system given in Equations (2.7 - 2.8) with a constant sampling time denoted by k given as follows:

$$\mathbf{x}_k = \mathbf{F}_{k-1}\mathbf{x}_{k-1} + \mathbf{G}_{k-1}\mathbf{u}_{k-1} + \mathbf{w}_{k-1} \tag{2.7}$$

$$\mathbf{y}_k = \mathbf{H}_k\mathbf{x}_k + \mathbf{v}_{k-1} \tag{2.8}$$

2.2 System description of nonlinear systems

In this thesis, we consider a general continuous nonlinear system given by the state space formulation that generates measurements given as follows:

$$\dot{\mathbf{x}}(t) = \mathbf{f}(\mathbf{x}(t), \mathbf{u}(t), \boldsymbol{\theta}) \tag{2.9}$$

$$\mathbf{y}(t) = \mathbf{h}(\dot{\mathbf{x}}(t)) \tag{2.10}$$

where $\mathbf{x} \in \mathbb{R}^n$, $\mathbf{u} \in \mathbb{R}^p$, $\boldsymbol{\theta} \in \mathbb{R}^{n_\theta}$, and $\mathbf{y} \in \mathbb{R}^q$, denotes the system states, control inputs, parameters of the state, process (system) noise, measurement (output), respectively, and $\mathbf{f}(\cdot) : \mathbb{R}^n \times \mathbb{R}^p \times \mathbb{R}^{n_\theta} \mapsto \mathbb{R}^n$, $\mathbf{h}(\cdot) : \mathbb{R}^n \mapsto \mathbb{R}^q$ are known mappings.

Further, we shall address the general discrete time of the nonlinear system with constant sampling time denoted by k given as follows:

$$\mathbf{x}_k = \mathbf{f}(\mathbf{x}_{k-1}, \mathbf{u}_{k-1}, \boldsymbol{\theta}, t_k) \tag{2.11}$$

$$\mathbf{y}_k = \mathbf{h}(\mathbf{x}_k, t_k) \tag{2.12}$$

where $\mathbf{f}(\cdot) : \mathbb{R}^n \times \mathbb{R}^p \times \mathbb{R}^{n_\theta} \mapsto \mathbb{R}^n$, $\mathbf{h}(\cdot) : \mathbb{R}^n \mapsto \mathbb{R}^q$ are the discretized version of Equation (2.9) and (2.10).

2.3 Additive noise

Multiple conceptual frameworks exist for incorporating noise into nonlinear state models. One prevalent approach involves modelling noise as an additive term, which can be mathematically expressed as follows:

$$\mathbf{x}_k = \mathbf{f}(\mathbf{x}_{k-1}, \mathbf{u}_{k-1}, \boldsymbol{\theta}, t_k) + \mathbf{w}_{k-1} \tag{2.13}$$

$$\mathbf{y}_k = \mathbf{h}(\mathbf{x}_k, t_k) + \mathbf{v}_{k-1} \tag{2.14}$$

as mentioned in Section 2.1, $\mathbf{w} \in \mathbb{R}^n$, and $\mathbf{v} \in \mathbb{R}^q$, are assumed to be uncorrelated zero-mean Gaussian noise with unknown covariance matrices $\mathbf{Q} \in \mathbb{R}^{n \times n}$, $\mathbf{R} \in \mathbb{R}^{q \times q}$, respectively.

2.4 Nonlinear noise

The system description with additive noise described in Section 2.3 may be reasonable in many applications. But in others, it may not be the best fit and it has been criticized by Kolås & Foss[20]. They argue that noise should be actively modelled, thus another way of modelling the noise is to model it as an extension of the process given as follows:

$$\mathbf{x}_k = \mathbf{f}(\mathbf{x}_{k-1}, \mathbf{u}_{k-1}, \boldsymbol{\theta}, \mathbf{w}_{k-1}, t_k) \tag{2.15}$$

$$\mathbf{y}_k = \mathbf{h}(\mathbf{x}_k, \mathbf{v}_{k-1}, t_k) \tag{2.16}$$

2.5 Random variation in plant parameters

The estimation of parameters in many processes results in an error in the parameter, which causes a systematic process-model mismatch due to unmodeled effects or model inaccuracies[34]. One way of reducing this error is to assume that the parameters in the plant vary with a given noise $\boldsymbol{w}_{\boldsymbol{\theta}}$ at each sample interval from a multivariate normal random distribution. Thus:

$$\boldsymbol{w}_{\boldsymbol{\theta}} \sim \mathcal{N}(\bar{\boldsymbol{\theta}}, \mathbf{P}_{\boldsymbol{\theta}}) \tag{2.17}$$

where $\boldsymbol{\theta}$ represents the values associated with the plant parameters, $\bar{\boldsymbol{\theta}}$ denotes the average value of the plant parameters, $\bar{\boldsymbol{\theta}} = \mathbb{E}[\boldsymbol{\theta}]$, and $\mathbf{P}_{\boldsymbol{\theta}}$ denotes the covariance matrix of the parameters. The varying plant parameter is assumed to deviate from the mean value by a fixed amount:

$$\boldsymbol{\theta} = \bar{\boldsymbol{\theta}} + \boldsymbol{w}_{\boldsymbol{\theta}} \tag{2.18}$$

where $\bar{\boldsymbol{\theta}}$ is used by the nonlinear model, and $\boldsymbol{\theta}$ is used by the plant.

3 Kalman filters

The Kalman filters maintains their status as one of the most widely used estimation algorithms[15]. The primary reason may be that other types of approximations that have been developed are either computationally unmanageable or require special assumptions about the form of the process and observation models that cannot be satisfied in practice[14]. The Kalman filter (KF) only utilizes the mean and covariance in its update rule, which offers several benefits like the linearity of the mean and covariance, and its successful compromise between computational complexity and representational flexibility.

3.1 The linear discrete-time Kalman filter

The linear discrete KF is used to estimate states based on linear dynamical systems in state space as described in Section 2.1. The process model defined in Equation (2.7) describes the evolution of the state from time $k-1$ to time k. For the sake of clarity, we will restate the Equation:

$$\mathbf{x}_k = \mathbf{F}_{k-1}\mathbf{x}_{k-1} + \mathbf{G}_{k-1}\mathbf{u}_{k-1} + \mathbf{w}_{k-1} \tag{3.1}$$

The process model is paired with the measurement model defined in Equation (2.8) that describes the relationship between the state and the measurement at the current step k as:

$$\mathbf{y}_k = \mathbf{H}_k\mathbf{x}_k + \mathbf{v}_{k-1} \tag{3.2}$$

The KF for the system provide an estimate of $\mathbf{x}_k$, and is formally characterized as a fusion of the a priori and a posteriori estimates, which is also called the time update[30].

It is given as follows:

$$\hat{\mathbf{x}}_k^- = \mathbf{F}_{k-1}\hat{\mathbf{x}}_{k-1}^+ + \mathbf{G}_{k-1}\mathbf{u}_{k-1} + \mathbf{w}_{k-1} \tag{3.3}$$

$$\mathbf{P}_k^- = \mathbf{F}_{k-1}\mathbf{P}_{k-1}^+\mathbf{F}_{k-1}^T + \mathbf{Q}_{k-1} \tag{3.4}$$

here the terms $\hat{\mathbf{x}}_k^-$ and $\mathbf{P}_k^-$, represent the state estimate and the covariance of the state estimation error, respectively. The superscript "-", associated with both variables, indicates that these values correspond to the state prior to the measurement update, hence referred to as the a priori estimate.

The equations for the measurement update of the state estimate, denoted by $\hat{\mathbf{x}}_k^+$ and $\mathbf{P}_k^+$, which computes the posteriori state estimates proceeds as follows:

$$\mathbf{K}_k = \mathbf{P}_k^+\mathbf{H}_k^T\mathbf{R}_k^{-1} \tag{3.5}$$

$$\hat{\mathbf{x}}_k^+ = \hat{\mathbf{x}}_k^- + \mathbf{K}_k\left(\mathbf{y}_k - \mathbf{H}_k\hat{\mathbf{x}}_k^-\right) \tag{3.6}$$

$$\mathbf{P}_k^+ = \left(\mathbf{I} - \mathbf{K}_k\mathbf{H}_k\right)\mathbf{P}_k^- \tag{3.7}$$

where the terms $\hat{\mathbf{x}}_k^+$ and $\mathbf{P}_k^+$ have a superscript "+" indicates the values subsequent to the measurement update (known as a posteriori estimate), and the $\mathbf{K}_k$, is the so-called Kalman gain.

While covariance matrices are designed to capture the statistical characteristics of disturbances, the actual statistical properties of disturbances often remain unknown or exhibit non-Gaussian behaviour in practical scenarios[18]. To achieve the desired system performance, $\mathbf{Q}$ and $\mathbf{R}$ are commonly used as adjustable parameters. Additionally, the performance is dependent on the initial guesses $\hat{\mathbf{x}}_\mathbf{0}^+$ and $\mathbf{P}_\mathbf{0}^+$. Schneider & Georgakis[28], have provided a thorough description of this concept, emphasizing the importance of using common sense when determining these values. For instance, assuming a batch process starts with a complete absence of reactants and solely

products at the initial state is generally ill-advised. Conversely, it is typically more appropriate to make the opposite assumption.

It should be acknowledged that the derivation of the linear discrete-time KF is derived upon the assumption of linearity in both the process and measurement models, and the process and measurement noise are additive Gaussian. Hence, a Kalman filter provides an optimal estimate only if the assumption is satisfied.

3.2 Nonlinear Kalman filtering

Various approaches exist for estimating means and covariances in the context of nonlinear functions. One such approach involves linearizing the means and covariances, as in the case of the extended Kalman filter (EKF). The EKF linearizes the model by evaluating the gradient at the current estimate. However, when the assumptions of local linearity are violated, this approach can yield highly unstable filters, since linearized transformations are only reliable if the error propagation being well approximated by a linear function[14], and if this condition is not met, the linearized approximation can be poor. At best, this undermines the performance of the filter, and at worst it makes the estimation diverge.

Furthermore, linearization is only applicable when the Jacobian matrix exists, which is not always the case. In some systems, discontinuities occur as a response to changes in parameters. To overcome this limitation, the unscented transformation (UT) was developed as a method of propagating mean and covariance through nonlinear transformations[14]. The UT offers significant implementation and accuracy benefits compared to linearization methods, as it eliminates the need for gradient computations while maintaining a comparable computational cost to that of the EKF.

In the context of this thesis, emphasis is placed on the estimation of means and covariances using the unscented transformation, which will be discussed in detail in the subsequent chapters.

3.2.1 The unscented transformation

The unscented transformation (UT), proposed by Julier and Uhlmann[13], is based on two insights. First, "*it is easy to perform a nonlinear transformation on a single point rather than an entire probability density function (PDF)*"[30]. Second, "*it's not too hard to find a set of individual points in state space whose sample PDF approximates the true PDF of a state vector.*" The primary objective of this development was to remove the limitations associated with linearized transformation of means and covariances, by estimating a random variable through a nonlinear transformation. This process involves the computation of a collection of sampling points, referred to as *sigma points*. Through this framework, we can effectively capture the statistical moments of the standard Gaussian distribution, and then use the generated sigma points to obtain an estimate of the nonlinear system. Given that nearly all practical systems in the control of chemical process plants involve nonlinearities in some form or another, utilising the UT can result in significant benefits and precision, compared to the approach based on linearization.

As an example, consider the model prediction equation $\hat{\mathbf{x}}_k^- = \mathbf{f}(\mathbf{x}_{k-1}^+)$, where the mean and covariance matrix of the input variables are denoted as $\hat{\mathbf{x}}_{k-1}^+$, and $\mathbf{P}_{\mathbf{x},k-1}^+$, respectively. In order to calculate the output variables $\hat{\mathbf{x}}_k^-$ and $\mathbf{P}_{\mathbf{x},k}^-$ from time step $k-1$ to k, we form a matrix of sigma points $\hat{\mathbf{x}}_{k-1} \in \mathbb{R}^{2n}$ of $2n$ sigma vectors $\hat{\mathbf{x}}_{k-1}^{(i)} \in \mathbb{R}^n$ which satisfies this selection scheme:

$$\begin{aligned}
\hat{\mathbf{x}}_{k-1}^{(0)} &= \hat{\mathbf{x}}_0^+ \\
\hat{\mathbf{x}}_{k-1}^{(i)} &= \hat{\mathbf{x}}_{k-1}^+ + \left(\sqrt{n\mathbf{P}_{\mathbf{x},k-1}^+}\right)_i \qquad i = 1, ..., n \\
\hat{\mathbf{x}}_{k-1}^{(i)} &= \hat{\mathbf{x}}_{k-1}^+ - \left(\sqrt{n\mathbf{P}_{\mathbf{x},k-1}^+}\right)_i \qquad i = n+1, ..., 2n
\end{aligned} \tag{3.8}$$

where $\left(\sqrt{n\mathbf{P}_{\mathbf{x},k-1}^+}\right)_i$ is the i-th column (or row) of the matrix square root. One approach to calculate it is to utilize the numerically efficient *Cholesky factorization* method[25].

The weighting coefficients for the mean and covariance, W_i^m, W_i^c, respectively, are

defined as follows:

$$W_i^m = W_i^c = \frac{1}{2n} \quad i = 1, \ldots, 2n \tag{3.9}$$

After we generate the sigma points, we send each point through the known nonlinear function, thereby obtaining the transformed sigma points given as follows:

$$\hat{\mathbf{x}}_k^{(i)} = \mathbf{f}(\hat{\mathbf{x}}_{k-1}^{(i)}) \tag{3.10}$$

We can then evaluate the sample mean and the covariance of the transformed sigma points:

$$\hat{\mathbf{x}}_k^- = \sum_{i=0}^{2n} W_i^m \hat{\mathbf{x}}_k^{(i)} \tag{3.11}$$

$$\mathbf{P}_{\mathbf{x},k}^- = \sum_{i=0}^{2n} W_i^c \left(\hat{\mathbf{x}}_k^{(i)} - \hat{\mathbf{x}}_k^-\right)\left(\hat{\mathbf{x}}_k^{(i)} - \hat{\mathbf{x}}_k^-\right)^T \tag{3.12}$$

The full procedure of the UT is presented in Algorithm 1.

Algorithm 1: The scheme for computing the unscented transformation.

for $i \in \{1,...,2n+1\}$ **do**

Calculate the sigma points and the corresponding weights:

$$\begin{aligned}
\hat{\mathbf{x}}_{k-1}^{(0)} &= \hat{\mathbf{x}}_0^+ \\
\hat{\mathbf{x}}_{k-1}^{(i)} &= \hat{\mathbf{x}}_{k-1}^+ + \left(\sqrt{nP_{\mathbf{x},k-1}^+}\right)_i \qquad i = 1, ..., n \\
\hat{\mathbf{x}}_{k-1}^{(i)} &= \hat{\mathbf{x}}_{k-1}^+ - \left(\sqrt{nP_{\mathbf{x},k-1}^+}\right)_i \qquad i = n+1, ..., 2n \\
W_i^m &= W_i^c = \frac{1}{2n} \qquad i = 1, \ldots, 2n
\end{aligned}$$

Time update:

$$\begin{aligned}
\hat{\mathbf{x}}_k^{(i)} &= \mathbf{f}(\hat{\mathbf{x}}_{k-1}^{(i)}) \\
\hat{\mathbf{x}}_k^- &= \sum_{i=0}^{2n} W_i^m \hat{\mathbf{x}}_k^{(i)} \\
\mathbf{P}_{\mathbf{x},k}^- &= \sum_{i=0}^{2n} W_i^c \left(\hat{\mathbf{x}}_k^{(i)} - \hat{\mathbf{x}}_k^-\right)\left(\hat{\mathbf{x}}_k^{(i)} - \hat{\mathbf{x}}_k^-\right)^T
\end{aligned}$$

One disadvantage of the UT arises as the dimension of the state space n increases. In such cases, the radius of the sphere that bounds all the sigma points also increases. Although the UT accurately captures the mean and covariance of the prior distribution, it does so at the cost of possibly sampling non-local effects, which is observed by looking at $\sqrt{n\mathbf{P}^{+}_{\mathbf{x},k-1}}$ in Equation (3.8).

Various approaches exist for generating sigma points used in the UT, and in the subsequent sections, we will explore two versions, namely, "*scaled sigma points*" and "*generalized sigma points*"[8, 12]. These versions aim to mitigate some of the drawbacks associated with the UT, thereby enhancing its performance.

3.2.1.1 Scaled sigma points

The scaled sigma points assume a symmetric distribution, and the underlying motivation behind their creation is to scale these points towards or away from the mean of the prior distribution using a suitable scaling parameter $\lambda = \alpha^2(n + \kappa) - n$. This approach ensures that the calculated covariance remains positive definite. The distance of the $i-$th sigma point from $\hat{\mathbf{x}}^{+}_{k-1}$, $\left|\hat{\mathbf{x}}^{(i)}_{k-1} - \hat{\mathbf{x}}^{+}_{k-1}\right|$, is proportional to $\sqrt{(n+\lambda)}$. When $\lambda = 0$, the distance is proportional to $\sqrt{n}$, when $\lambda > 0$, the points are scaled further from $\hat{\mathbf{x}}^{+}_{k-1}$. Conversely, when $\lambda < 0$ the points are scaled toward $\hat{\mathbf{x}}^{+}_{k-1}$.

The procedure for selecting scaled sigma points results in a matrix of sigma points $\hat{\mathbf{x}}_{k-1} \in \mathbb{R}^{n\times(2n+1)}$ consisting of $2n + 1$ sigma vectors $\hat{\mathbf{x}}^{(i)}_{k-1} \in \mathbb{R}^{n}$. Each sigma vector is associated with corresponding weights for the mean and covariance, W^{m}_{i}, W^{c}_{i}, respectively:

$$
\begin{aligned}
&\hat{\mathbf{x}}^{(0)}_{k-1} = \hat{\mathbf{x}}^{+}_{0} && && W^{m}_{0} = \frac{\lambda}{n+\lambda} && i = 0\\
&\hat{\mathbf{x}}^{(i)}_{k-1} = \hat{\mathbf{x}}^{+}_{k-1} + \left(\sqrt{(n+\lambda)\mathbf{P}^{+}_{\mathbf{x},k-1}}\right)_{i} && i = 1,...,n && W^{c}_{0} = \frac{\lambda}{n+\lambda} + \left(1 - \alpha^2 + \beta\right) && i = 0\\
&\hat{\mathbf{x}}^{(i)}_{k-1} = \hat{\mathbf{x}}^{+}_{k-1} - \left(\sqrt{(n+\lambda)\mathbf{P}^{+}_{\mathbf{x},k-1}}\right)_{i} && i = n+1,...,2n && W^{c}_{i} = W^{m}_{i} = \frac{1}{2\,(n+\lambda)} && i = 1,...,2n
\end{aligned}
\tag{3.13}
$$

where α is a scaling parameter to minimize possible higher order effects, and takes the values between ($0 < \alpha < 1$) typically 10^{-3}, β is used to affect the weighting of the zeroth sigma-point for the calculation of the covariance, and κ is a scaling factor. Choosing $\kappa > 0$ ensures the covariance matrix is positive semidefinite. $\left(\sqrt{(n+\lambda)\mathbf{P}^{+}_{\mathbf{x},k-1}}\right)_i$ is the i-th column (or row) of the matrix square root.

3.2.1.2 Generalized sigma points

The scaled sigma points in Section 3.2.1.1 relies on the assumption of a symmetrical distribution of the variable $\mathbf{x}$. However, this assumption may not hold true in all scenarios. In cases where we encounter a random variable that is not symmetrical, it is recommended to employ the generalized sigma points instead. The objective of the generalized sigma points is to capture the diagonal components of the skewness and kurtosis tensors of most probability distributions with high accuracy[8].

The method employs $2n + 1$ sigma points to capture the mean and covariance matrix, but also utilises $n^2 + 2n + 1$ additional free parameters to accurately match the diagonal components of the skewness and kurtosis tensors. We define $\check{\mathbf{S}} \in \mathbb{R}^n$ as the skewness tensor, and $\check{\mathbf{K}} \in \mathbb{R}^n$ as the kurtosis tensor, given as follows:

$$\check{\mathbf{S}} = [\mathbf{S}_{111}, \mathbf{S}_{222}, \ldots, \mathbf{S}_{nnn}]^T \tag{3.14}$$

$$\check{\mathbf{K}} = [\mathbf{K}_{1111}, \mathbf{K}_{2222}, \ldots, \mathbf{K}_{nnnn}]^T \tag{3.15}$$

Then we assign weights to the sigma points in accordance with the first three moments of $\mathbf{x}$, and subsequently enforce constraints on these weights to ensure they match the fourth moment of $\mathbf{x}$, given as follows:

$$\sum_{i=0}^{2n} W_i = 1 \tag{3.16}$$

$$-W^{'} \odot \mathbf{u} + W^{''} \odot \mathbf{v} = 0 \tag{3.17}$$

$$W^{'} \odot \mathbf{u}^{\odot 2} + W^{''} \odot \mathbf{v}^{\odot 2} = 1 \tag{3.18}$$

$$-W^{'} \odot \mathbf{u}^{\odot 3} + W^{''} \odot \mathbf{v}^{\odot 3} = \sqrt{\mathbf{P}^{+}_{\mathbf{x},k-1}}^{\odot -3} \check{\mathbf{S}} \tag{3.19}$$

Note that the element-wise product $\odot$ (Hadamard product) and the element-wise division $\oslash$ (Hadamard division) are used, and are defined in A.1. The weights must satisfy:

$$W^{'} \odot \mathbf{u}^{\odot 4} + W^{''} \odot \mathbf{v}^{\odot 4} = \sqrt{\mathbf{P}^{+}_{\mathbf{x},k-1}}^{\odot -4} \check{\mathbf{K}} \tag{3.20}$$

By rearranging and manipulating Equations (3.16 - 3.19), and substituting it into Equation (3.20) we end up with design parameters $\mathbf{u}$, and $\mathbf{v}$ given as:

$$\mathbf{u} = \frac{1}{2}\left(\sqrt{\mathbf{P}^{+}_{\mathbf{x},k-1}}^{\odot -3} \check{\mathbf{S}} + \sqrt{4\sqrt{\mathbf{P}^{+}_{\mathbf{x},k-1}}^{\odot -4} \check{\mathbf{K}} - 3\left(\sqrt{\mathbf{P}^{+}_{\mathbf{x},k-1}}^{\odot -3} \check{\mathbf{S}}\right)^{\odot 2}}\right) \tag{3.21}$$

$$\mathbf{v} = \mathbf{u} + \sqrt{\mathbf{P}^{+}_{\mathbf{x},k-1}}^{\odot -3} \check{\mathbf{S}} \tag{3.22}$$

In the case of normal distributions, $\check{\mathbf{K}}$, can be derived as a function of its covariance matrix, as described by Isserlis' theorem defined in A.1. Conversely, when dealing with distributions of a different nature, an alternative methodology must be employed to compute $\check{\mathbf{K}}$.

The steps for computing the generalized sigma points is shown in Algorithm 2, and the Python module in Listing 2 shows how they are computed.

Algorithm 2: The steps for computing the generalized sigma points.

1. Choose the free parameter vector u > 0 by Equation (3.21)

2. Calculate the free parameter vector

$$\mathbf{v} = \mathbf{u} + \sqrt{\mathbf{P}_{\mathbf{x},k-1}^{+}}^{\odot -3} \breve{\mathbf{S}}$$

for $i \in \{1,...,n\}$ **do**

3. Calculate the sigma points:

$$\begin{aligned} \hat{\mathbf{x}}_{k-1}^{(0)} &= \hat{\mathbf{x}}_{k-1}^{+} && W_0 \\ \hat{\mathbf{x}}_{k-1}^{(i)} &= \hat{\mathbf{x}}_{k-1}^{+} - \mathbf{u}_i \left(\sqrt{\mathbf{P}_{\mathbf{x},k-1}^{+}} \right)_i && W_i^{'} \\ \hat{\mathbf{x}}_{k-1}^{(i)} &= \hat{\mathbf{x}}_{k-1}^{+} + \mathbf{v}_i \left(\sqrt{\mathbf{P}_{\mathbf{x},k-1}^{+}} \right)_i && W_i^{''} \end{aligned}$$

4. Calculate the weights:

$$\begin{aligned} W^{''} &= 1 \oslash \mathbf{v} \oslash (\mathbf{u} + \mathbf{v}) \\ W^{'} &= W^{''} \odot \mathbf{v} \oslash \mathbf{u} \\ W_0 &= 1 - \sum_{i=0}^{2n} = W_i \end{aligned}$$

where $W = \left[W_0, W^{'T}, W^{''T} \right]^T$.

Moreover, the utilization of generalized sigma points grants us the opportunity to enforce constraints on the generated sigma points, which proves advantageous in certain scenarios. To illustrate, consider a scenario where the state vector represents the concentration of chemical components. Under the circumstances of employing an unconstrained sigma-point algorithm, it is plausible that generated sigma points may correspond to negative concentrations for these chemical components. As negative

concentrations are physically impossible and lack validity within a process model, it becomes essential to impose constraints on the sigma points. Which is obtained by restricting the sigma points to be between a lower bound $\mathbf{a} \in \mathbb{R}^n$ and upper bound $\mathbf{b} \in \mathbb{R}^n$:

$$\mathbf{a} < \hat{\mathbf{x}}_{k-1}^{(i)} < \mathbf{b}$$

In order to impose restrictions on the sigma points, it becomes necessary to introduce a slack parameter θ, where $\theta \in (0, \ldots, 1)$, is a constant determined by the user. By incorporating θ, we proceed to redefine the independent parameters $\mathbf{u}_i$ and $\mathbf{v}_i$ for $i \in \{1, \ldots, n\}$ as follows:

$$\mathbf{u}_i = \theta \left[min \left\{ \left| \left(\hat{\mathbf{x}}_{k-1}^{+} - \mathbf{a} \right) \oslash \left(\sqrt{\mathbf{P}_{\mathbf{x},k-1}^{+}} \right)_i \right| \right\} \right] \qquad \text{if} \quad \hat{\mathbf{x}}_{k-1}^{(i)} < \mathbf{a} \tag{3.23}$$

$$\mathbf{v}_i = \theta \left[min \left\{ \left| \left(\mathbf{b} - \hat{\mathbf{x}}_{k}^{+} \right) \oslash \left(\sqrt{\mathbf{P}_{\mathbf{x},k-1}^{+}} \right)_i \right| \right\} \right] \qquad \text{if} \quad \hat{\mathbf{x}}_{k-1}^{(i+n)} > \mathbf{b} \tag{3.24}$$

The procedure of constraining the sigma points is given in the Algorithm 3, and the corresponding Python module is given in Listing 2.

Algorithm 3: The procedure for computing the constrained generalized sigma points.

Implement Algorithm 2

if $\hat{\mathbf{x}}_{k-1}^{(i)} < \mathbf{a}$ *for* $i \in \{1, \ldots, 2n\}$ **then**

  if $i \leq n$ **then**

    $\mathbf{u}_i = \theta \left[min \left\{ \left| \left(\hat{\mathbf{x}}_{k-1}^{+} - \mathbf{a} \right) \oslash \left(\sqrt{\mathbf{P}_{\mathbf{x},k-1}^{+}} \right)_i \right| \right\} \right]$

  if $i > n$ **then**

    $\mathbf{v}_{i-n} = \theta \left[min \left\{ \left| \left(\mathbf{a} - \hat{\mathbf{x}}_{k-1}^{+} \right) \oslash \left(\sqrt{\mathbf{P}_{\mathbf{x},k-1}^{+}} \right)_{i-n} \right| \right\} \right]$

Repeat step 3 and 4 from Algorithm 2 if $\mathbf{v}$ is not redefined,

otherwise only repeat step 4.

if $\hat{\mathbf{x}}_{k-1}^{(i+n)} > \mathbf{b}$ *for* $i \in \{1, \ldots, 2n\}$ **then**

  if $i \leq n$ **then**

    $\mathbf{u}_i = \theta \left[min \left\{ \left| \left(\hat{\mathbf{x}}_{k-1}^{+} - \mathbf{b} \right) \oslash \left(\sqrt{\mathbf{P}_{\mathbf{x},k-1}^{+}} \right)_i \right| \right\} \right]$

  if $i > n$ **then**

    $\mathbf{v}_{i-n} = \theta \left[min \left\{ \left| \left(\mathbf{b} - \hat{\mathbf{x}}_{k-1}^{+} \right) \oslash \left(\sqrt{\mathbf{P}_{\mathbf{x},k-1}^{+}} \right)_{i-n} \right| \right\} \right]$

Repeat step 3, 4 and 5 from Algorithm 2 if $\mathbf{v}$ is not redefined,

otherwise only repeat step 4 and 5.

However, it is worth noting that the UT relying on unconstrained sigma points offer a higher accuracy in terms of the Taylor series.

3.3 The unscented Kalman filter

The unscented Kalman filter (UKF) is a recursive nonlinear Kalman filter that utilizes the UT described in Section 3.2.1 to propagate means and covariances through the nonlinear process model and measurement equations. In the case of a linear system, the Kalman filter presented in Section 3.1 allows for precise updates of the mean and covariance. However, for nonlinear systems, the mean and covariance can be approximately updated using the UT. As a result, we replace the equations from Section 3.1 with UT to derive the UKF algorithm. Initially, the UT is applied to the model prediction equation, $\mathbf{f}(\hat{\mathbf{x}}_{k-1}^{+}) + \mathbf{w}_k$, to obtain the priori estimate, $\hat{\mathbf{x}}_k^{-}$ and $\hat{\mathbf{P}}_{\mathbf{x},k}^{-}$, by assuming additive noise, we get an additive term of $\mathbf{Q}_k$ as the covariance matrix. Subsequently, the measurement equation is used to transform the sigma points into $\hat{\mathbf{y}}_k^{(i)}$ given as follows:

$$\hat{\mathbf{y}}_k^{(i)} = \mathbf{h}(\hat{\mathbf{x}}_{k-1}^{(i)}) + \mathbf{v}_{k-1} \tag{3.25}$$

Then combine the $\hat{\mathbf{y}}_k^{(i)}$ vectors to obtain the predicted measurement at time step k, in this manner:

$$\hat{\mathbf{y}}_k = \sum_{i=0}^{2n} W_i^m \hat{\mathbf{y}}_k^{(i)} \tag{3.26}$$

Afterwards the covariance of the predicted measurement can be computed:

$$\mathbf{P}_{\mathbf{y},k} = \sum_{i=0}^{2n} W_i^c \left(\hat{\mathbf{y}}_k^{(i)} - \hat{\mathbf{y}}_k\right) \left(\hat{\mathbf{y}}_k^{(i)} - \hat{\mathbf{y}}_k\right)^T + \mathbf{R}_{k-1} \tag{3.27}$$

And the cross covariance between $\hat{\mathbf{x}}_k^-$ and $\hat{\mathbf{y}}_k$ is given as:

$$\mathbf{P}_{\mathbf{xy},k} = \sum_{i=0}^{2N} W_k^c \left(\hat{\mathbf{x}}_k^{(i)} - \hat{\mathbf{x}}_k^-\right)\left(\hat{\mathbf{y}}_k^{(i)} - \hat{\mathbf{y}}_k\right)^T \tag{3.28}$$

Finally the measurement update of the estimate can be computed using the normal Kalman filter equations:

$$\mathbf{K}_k = \mathbf{P}_{\mathbf{xy},k}\mathbf{P}_{\mathbf{xy},k}^{-\mathbf{1}} \tag{3.29}$$

$$\hat{\mathbf{x}}_k^+ = \hat{\mathbf{x}}_k^- + \mathbf{K}_k(\mathbf{y}_k - \hat{\mathbf{y}}_k) \tag{3.30}$$

$$\mathbf{P}_{\mathbf{x},k}^+ = \mathbf{P}_k^- - \mathbf{K}_k\mathbf{P}_{\mathbf{y},k}\mathbf{K}_k^T \tag{3.31}$$

The complete UKF algorithm that updates the mean $\hat{\mathbf{x}}_k^+$, and the covariance $\mathbf{P}_{\mathbf{x},k}^+$ is presented in Algorithm 4, and the implementation of the Python modules for the Scaled UT and GenUT filters is given in Listing 5 and 6, respectively.

Algorithm 4: The full procedure for computing the UKF.

1. Initialize the system with:

$$\hat{\mathbf{x}}_0^+ = \mathbb{E}[\mathbf{x}_0]$$
$$\mathbf{P}_0^+ = \mathbb{E}[(\mathbf{x}_0 - \hat{\mathbf{x}}_0)(\mathbf{x}_0 - \hat{\mathbf{x}}_0)^T]$$

for $k \in \{1,...,\infty\}$ **do**

2. Calculate the sigma points, $\hat{\mathbf{x}}_{k-1}^{(i)}$, and the corresponding, weights W_i, from either Equation (3.13) or Algorithm 2.

3. Time update:

$$\hat{\mathbf{x}}_k^{(i)} = \mathbf{f}(\hat{\mathbf{x}}_{k-1}^{(i)}) + \mathbf{w}_{k-1}$$
$$\hat{\mathbf{x}}_k^- = \sum_{i=0}^{2n} W_i^m \hat{\mathbf{x}}_k^{(i)}$$
$$\mathbf{P}_{\mathbf{x},k}^- = \sum_{i=0}^{2n} W_i^c \left(\hat{\mathbf{x}}_k^{(i)} - \hat{\mathbf{x}}_k^-\right)\left(\hat{\mathbf{x}}_k^{(i)} - \hat{\mathbf{x}}_k^-\right)^T + \mathbf{Q}_{k-1}$$

4. Measurement update:

$$\hat{\mathbf{y}}_k^{(i)} = \mathbf{h}(\hat{\mathbf{x}}_{k-1}^{(i)}) + \mathbf{v}_{k-1}$$
$$\hat{\mathbf{y}}_k = \sum_{i=0}^{2n} W_i^m \hat{\mathbf{y}}_k^{(i)}$$
$$\mathbf{P}_{\mathbf{y},k} = \sum_{i=0}^{2n} W_i^c \left(\hat{\mathbf{y}}_k^{(i)} - \hat{\mathbf{y}}_k\right)\left(\hat{\mathbf{y}}_k^{(i)} - \hat{\mathbf{y}}_k\right)^T + \mathbf{R}_{k-1}$$
$$\mathbf{P}_{\mathbf{xy},k} = \sum_{i=0}^{2N} W_k^c \left(\hat{\mathbf{x}}_k^{(i)} - \hat{\mathbf{x}}_k^-\right)\left(\hat{\mathbf{y}}_k^{(i)} - \hat{\mathbf{y}}_k\right)^T$$
$$\mathbf{K}_k = \mathbf{P}_{\mathbf{xy},k}\mathbf{P}_{\mathbf{xy},k}^{-1}$$
$$\hat{\mathbf{x}}_k^+ = \hat{\mathbf{x}}_k^- + \mathbf{K}_k(\mathbf{y}_k - \hat{\mathbf{y}}_k)$$
$$\mathbf{P}_{\mathbf{x},k}^+ = \mathbf{P}_k^- - \mathbf{K}_k\mathbf{P}_{\mathbf{y},k}\mathbf{K}_k^T$$

4 Estimation of the noise statistics

One essential component of a KF is its covariance matrices, $\mathbf{Q}$ and $\mathbf{R}$, which serve the purpose of regulating filter divergence and mitigating model errors. Nonetheless, in practical scenarios, acquiring precise information regarding unknown noise statistics becomes challenging. This limitation can lead to substantial errors in state estimation and consequently result in divergence. To address this detrimental aspect, several estimation methods have been developed, which can be classified into four distinct groups: correlation methods[23], the maximum-likelihood methods[17], the covariance matching methods[24], and the Bayesian methods[22]. In this section, we introduce the estimation approach by the works of Krog & Jäschke[21], as well as Berry & Sauer[8].

4.1 Model-based estimation of noise statistics

As discussed in Section 2.4, noise is generally not additive. The method discussed in this section assumes parametric uncertainty in the process model, and it translates this uncertainty into additive process noise. We can then use the process noise to estimate the noise statistic of $\mathbf{Q}$ in a more generalized form. One advantage of this approach compared to other approaches, such as a fixed hand-tuned $\mathbf{Q}$, or for instance by a linearization approximation by Fotopoulos[9] is the ability to capture statistical variations of the real trajectory.

4.1.1 Modelling noise as parametric uncertainty

Let us consider Equation (2.13) describe the actual plant in the nonlinear state space form given as follows:

$$\mathbf{x}_k^{true} = \mathbf{f}(\mathbf{x}_{k-1}^{true}, \mathbf{u}_{k-1}, \boldsymbol{\theta}, t_k) \tag{4.1}$$

where $\boldsymbol{\theta}$ is a random variable. Hence, the nonlinear noise in Equation (4.1) is considered to be captured by the uncertain parameter distribution. An approximation of the plant nominal value is obtained using the nonlinear model with the nominal value of the parameters given as:

$$\mathbf{x}_k^{nom} = \mathbf{f}(\mathbf{x}_{k-1}^{nom}, \mathbf{u}_{k-1}, \bar{\boldsymbol{\theta}}, t_k) \tag{4.2}$$

where $\bar{\boldsymbol{\theta}}$ is the mean value of the parameter distribution. By adding an additive noise term $\tilde{\mathbf{w}}_k \sim (\bar{\mathbf{w}}_k, \mathbf{Q}_k)$ in Equation (4.2), we can capture the real trajectory of the plant written as:

$$\mathbf{x}_k^{nom} = \mathbf{f}(\mathbf{x}_{k-1}^{nom}, \mathbf{u}_{k-1}, \bar{\boldsymbol{\theta}}, t_k) + \tilde{\mathbf{w}}_k \tag{4.3}$$

Under the circumstance that we are able to specify a $\tilde{\mathbf{w}}_k$ the prediction of the model will be similar to the real plant. Since we desire that Equation (4.1) and Equation (4.2) to be the same, we can calculate the value of $\tilde{\mathbf{w}}_k$ such as:

$$\tilde{\mathbf{w}}_k = \mathbf{f}(\mathbf{x}_{k-1}^{true}, \mathbf{u}_{k-1}, \boldsymbol{\theta}, t_k) - \mathbf{f}(\mathbf{x}_{k-1}^{nom}, \mathbf{u}_{k-1}, \bar{\boldsymbol{\theta}}) \tag{4.4}$$

$$\Rightarrow \tilde{\mathbf{w}}_k = \mathbf{f}(\mathbf{x}_{k-1}^{true}, \mathbf{u}_{k-1}, \boldsymbol{\theta}, t_k) - \mathbf{x}_k^{nom} \tag{4.5}$$

Here, $\mathbf{x}_{k-1}^{true}$ and $\mathbf{x}_{k-1}^{nom}$ represent RV, and under the assumption that they are fairly close to the approximated mean, $\mathbf{x}_k^{true}$ can be approximated as $\hat{\mathbf{x}}_{k-1}^{+}$, and the same apply for $\mathbf{x}_{k-1}^{nom}$. The only random variable left is therefore the parameter $\boldsymbol{\theta}$. Valappil and Georgakis[34] have two methods of exploiting the information about the uncertainty of $\boldsymbol{\theta}$ to estimate $\tilde{\mathbf{w}}_k$. The first approach propagates the covariance matrix $\mathbf{P}_{\boldsymbol{\theta}}$ through the system given in Equation (4.5). Which is a linearization method because it utilises a Taylor expansion on the right-hand side of Equation (4.5). Where the nonlinear dependence of $\tilde{\mathbf{w}}_k$ and $\boldsymbol{\theta}$ can be linearized around the nominal parameter values and states. The second approach uses a Monte Carlo simulation. The primary drawback of the first approach lies in its limited accuracy, although it offers the advantage of

low computational cost. Conversely, the latter approach achieves higher accuracy, but unfortunately at the expense of significantly increased computational requirements.

To strike a balance between accuracy and computational efficiency, we draw inspiration from the method proposed by Krog & Jäschke[21], and employ the UT described in Chapter 3.2.1.1 to get the proper sigma points $\boldsymbol{\chi}_{\boldsymbol{\theta}}$ and corresponding weights $W_{\boldsymbol{\theta}}$ for the computation of $\tilde{\mathbf{w}}_k$. It is also possible to use the version described in Chapter 3.2.1.2 to get similar results.

4.1.2 Scaled unscented transformation to estimate the noise statistics

Let us consider that the mean $\bar{\boldsymbol{\theta}}$, and covariance $\mathbf{P}_{\boldsymbol{\theta}}$ of our parameter distribution are available. By employing Equation (3.13), we can compute the $2n_{\boldsymbol{\theta}} + 1$ sigma points $\boldsymbol{\chi}_{\boldsymbol{\theta}} \in \mathbb{R}^{n_{\boldsymbol{\theta}} \times (2n_{\boldsymbol{\theta}}+1)}$, along with their corresponding weights represented $W_{\boldsymbol{\theta}} \in \mathbb{R}^{n_{\boldsymbol{\theta}}}$. The $(i+1)$-th column in $\boldsymbol{\chi}_{\boldsymbol{\theta}}$ is denoted as the as the $i-$th sigma point $\boldsymbol{\chi}_{\boldsymbol{\theta}}^{(i)}$. Subsequently, the mean and covariance of Equation (4.5) can be approximated using the following equations:

$$\tilde{\mathbf{w}}_k = \mathbf{f}(\hat{\mathbf{x}}_{k-1}^{+}, \mathbf{u}_{k-1}, \boldsymbol{\chi}_{\boldsymbol{\theta}}^{(i)}, t_k) - \mathbf{x}_k^{nom} \tag{4.6}$$

$$\bar{\mathbf{w}}_k = \sum_{i=0}^{2n_{\boldsymbol{\theta}}} W_{\boldsymbol{\theta}}^{(i)} \tilde{\mathbf{w}}_k^{(i)} \tag{4.7}$$

$$\mathbf{Q}_k = \sum_{i=0}^{2n_{\boldsymbol{\theta}}} W_{\boldsymbol{\theta}}^{(i)} \left(\tilde{\mathbf{w}}_k^{(i)} - \bar{\mathbf{w}}_k\right) \left(\tilde{\mathbf{w}}_k^{(i)} - \bar{\mathbf{w}}_k\right)^T \tag{4.8}$$

The algorithm outlining the procedure for estimating the noise statistics is presented in Algorithm 5 and is implemented in the Python module given in Listing 4.

It is worth noting that the initial step in Algorithm 5 can be changed to Algorithm 2 as an alternative to Equation (3.13) for the computation of the sigma points and the corresponding weights. Furthermore, it should be considered that the estimation of the

Algorithm 5: The scaled unscented transformation procedure for estimating the noise statistics.

1. Performe Equation (3.13) to get: χ_θ, W_θ

2. Propagate the sigma points and estimate the noise statistics by:

for $k \in \{1,...,\infty\}$ **do**

$$\mathbf{x}_k^{nom} = \mathbf{f}(\hat{\mathbf{x}}_{k-1}^+, \mathbf{u}_{k-1}, \bar{\boldsymbol{\theta}}, t_k)$$
$$\tilde{\mathbf{w}}_k^{(i)} = \mathbf{f}(\hat{\mathbf{x}}_{k-1}^+, \mathbf{u}_{k-1}, \boldsymbol{\chi}_{\boldsymbol{\theta}}^{(i)}, t_k) - \mathbf{x}_k^{nom}$$
$$\bar{\mathbf{w}}_k = \sum_{i=0}^{2n_\theta} W_{\boldsymbol{\theta}}^{(i)} \tilde{\mathbf{w}}_k^{(i)}$$
$$\mathbf{Q}_k = \sum_{i=0}^{2n_\theta} W_{\boldsymbol{\theta}}^{i} \left(\tilde{\mathbf{w}}_k^{(i)} - \bar{\mathbf{w}}_k\right) \left(\tilde{\mathbf{w}}_k^{(i)} - \bar{\mathbf{w}}_k\right)^T$$

measurement statistics, namely $\bar{\mathbf{v}}_k$, and $\mathbf{R}_k$ can also be computed utilizing the identical procedure outlined in Algorithm 5.

4.2 Data-driven estimation of noise statistics

The adaptive estimation of noise statistics proposed by Berry & Sauer[8] represents an extension of Mehra's method for reconstructing the error covariance matrices **Q** and **R** through the inclusion of auxiliary equations in the KF framework. This approach is based on Mehra's innovation correlation method, but it undergoes modifications to enable its application in the non-linear domain. The rationale behind these adaptations lies in the fact that Mehra's method relied on the assumption of stationarity within the KF's innovation sequence, which is no longer valid for non-linear models. Instead, Berry & Sauer use the innovations at each filter step, along with locally linearized dynamics and observations, to recover independent estimates of the matrices **Q** and **R**. These estimates are then integrated sequentially using a moving average to update the matrices $\mathbf{Q}_k^{filt}$ and $\mathbf{R}_k^{filt}$, used by the filter at time step k. By treating each innovation separately, using the local linearization relevant to the current state, we are able to recover the full matrices **Q** and **R** when the model error is Gaussian white noise.

4.2.1 Weighted statistical linear regression (WSLR)

In order to apply the adaptive scheme proposed by Berry & Sauer, it becomes necessary to linearize our model. One approach to achieve this linearization is through the utilization of weighted statistical linear regression (WSLR), which is described by Gelb[10]. Notably, UT itself can be viewed as an implicit WSLR.

Considering the nonlinear equation $\mathbf{f}(\mathbf{x})$ and the measurement equation $\mathbf{h}(\mathbf{x})$ outlined in Equations (2.15) and (2.16), respectively, if we assume that both equations have undergone mean and covariance propagation as presented in Equations (3.10 - 3.12) and (3.25 - 3.27) respectively, and our objective is to determine the linear regression that satisfies this condition:

$$\mathbf{y} = \mathbf{A}\mathbf{x} + \mathbf{b} \tag{4.9}$$

That minimizes the weighted sum of squared errors:

$$\{\mathbf{A}, \mathbf{b}\} = \operatorname{argmin} \sum_{i=1}^{N} W_i^m \boldsymbol{\varepsilon}_i^T \boldsymbol{\varepsilon}_i^T \tag{4.10}$$

Where the point-wise linearization error $\boldsymbol{\varepsilon}_i$ is defined as:

$$\boldsymbol{\varepsilon}_i = \hat{\mathbf{y}}_{k-1}^{(i)} - (\mathbf{A}\hat{\mathbf{x}}_{k-1}^{(i)} + \mathbf{b}) \tag{4.11}$$

Which can be interpreted as a finite sample bases approximation of the true expected statistical linearization given by:

$$\mathbf{J} = \mathbb{E}[\boldsymbol{\varepsilon}^T W \boldsymbol{\varepsilon}] \approx \sum_{i=1}^{N} W_i^m \boldsymbol{\varepsilon}_i^T \boldsymbol{\varepsilon}_i \tag{4.12}$$

It can be shown that the solution to the statistical linear regression is given as follows:

$$\mathbf{A} = \mathbf{P}_{\mathbf{xy},k} \mathbf{P}_{\mathbf{x}}^{-1} \tag{4.13}$$

$$\mathbf{b} = \hat{\mathbf{y}}_k - \mathbf{A}\hat{\mathbf{x}}_k^- \tag{4.14}$$

where $\mathbf{P}_{\mathbf{xy},k}$ and $\mathbf{P}_{\mathbf{x}}^{-1}$ can be computed by the UT. By applying Equation (4.9) to both the nonlinear process model and measurement equation, we obtain the following equations:

$$\mathbf{x}_k = \mathbf{f}(\mathbf{x}_{k-1}) \approx \mathbf{F}_{k-1}\mathbf{x}_{k-1} + \mathbf{b}_{\mathbf{f},k-1} \tag{4.15}$$

$$\mathbf{y}_k = \mathbf{h}(\mathbf{x}_k) \approx \mathbf{H}_k\mathbf{x}_k + \mathbf{b}_{\mathbf{h},k} \tag{4.16}$$

Comparing (4.15) and (4.16) with Equation (4.9), the solution of the statistical linear regression given in Equation (4.13) can be utilized similarly to compute the matrices $\mathbf{F}_{k-1}$ and $\mathbf{H}_k$ as the $\mathbf{A}$ matrix.

4.2.2 Adaptive unscented Kalman filter

In our specific scenario, the number of observations m was lower than the number of elements n in the state. Consequently, $\mathbf{H}_k$ and $\mathbf{H}_k\mathbf{F}_{k-1}$ was not invertible. To address this issue, Bélenger[3] proposed a method to estimate $\mathbf{Q}_{k-1}$ by parameterizing $\mathbf{Q}_k^e = \sum \mathbf{q}_p\mathbf{Q}_p$ as a linear combination of fixed matrices $\mathbf{Q}_p$. To enforce this constraint, we initially set:

$$\mathbf{C}_{k-1} = \boldsymbol{\varepsilon}_k\boldsymbol{\varepsilon}_{k-1}^T + \mathbf{H}_k\mathbf{F}_{k-1}\mathbf{K}_{k-1}\boldsymbol{\varepsilon}_{k-1}\boldsymbol{\varepsilon}_{k-1}^T - \mathbf{H}_k\mathbf{F}_{k-1}\mathbf{F}_{k-2}\mathbf{P}_{\mathbf{x},k-2}^{+}\mathbf{F}_{k-2}^T\mathbf{H}_{k-1}^T \tag{4.17}$$

Our objective is to solve the equation $\mathbf{H}_k\mathbf{F}_{k-1}\mathbf{Q}_{k-1}^e\mathbf{H}_{k-1}^T = \mathbf{C}_{k-1}$, which involves finding the vector $\mathbf{q}$ with values $\mathbf{q}_p$ that minimizes the Frobenius norm, given by:

$$\left\| \left(\mathbf{C}_{k-1} - \sum_p \mathbf{q}_p\mathbf{H}_k\mathbf{F}_{k-1}\mathbf{Q}_p\mathbf{H}_{k-1}^T \right) \right\|_F \tag{4.18}$$

To solve Equation (4.18), we simplify the problem by vectorizing all the matrices involved. Let $\mathrm{vec}(\mathbf{C}_{k-1})$ represent the vector formed by concatenating the columns of

$\mathbf{C}_{k-1}$. By solving the least-squares solution of the equation:

$$\mathbf{A}_{k-1}\mathbf{q} = \sum_p \mathbf{q}_p \text{vec}(\mathbf{H}_k \mathbf{F}_{k-1} \mathbf{Q}_p \mathbf{H}_{k-1}^T) \approx \text{vec}(\mathbf{C}_k) \tag{4.19}$$

where the p-th column of $\mathbf{A}_k$ is $\text{vec}(\mathbf{H}_k)\mathbf{F}_{k-1}\mathbf{Q}_p\mathbf{H}_k^T$, we can find the least-squares solution $\mathbf{q} = \mathbf{A}^{\dagger}\text{vec}(\mathbf{C}_k)$ and construct the estimated matrix $\mathbf{Q}_{k-1}^e$. In our applications, we utilize a diagonal parameterization using n matrices ($\mathbf{Q}_p = \mathbf{E}_{pp}$), where $\mathbf{E}_{ij}$ represents the elementary matrix with its only non-zero entry being 1 at the ij position. There are, however, other possible choices of $\mathbf{Q}_p$, and these choices should be considered as tuning parameters of the method. After constructing the estimated matrix $\mathbf{Q}_{k-1}^e$ we are able to update $\mathbf{Q}_k^{filt}$ at each step with an exponentially weighted moving average:

$$\mathbf{Q}_k^{filt} = \mathbf{Q}_{k-1}^{filt} + \delta(\mathbf{Q}_{k-1}^e - \mathbf{Q}_{k-1}^{filt}) \tag{4.20}$$

The parameter δ should be selected to be sufficiently small in order to achieve a smoothing effect on the moving average. The algorithm's process is illustrated in Algorithm 6, and the implementation of the algorithm can be found in Listing 4.

Algorithm 6: The steps for computing the adaptive UKF.

1. **Performe Algorithm 4 to get:** $\mathbf{H}_k$, $\mathbf{F}_{k-1}$
2. **Compute:**

$$\begin{aligned}
\mathbf{C}_{k-1} &= \boldsymbol{\varepsilon}_k \boldsymbol{\varepsilon}_{k-1}^T + \mathbf{H}_k \mathbf{F}_{k-1} \mathbf{K}_{k-1} \boldsymbol{\varepsilon}_{k-1} \boldsymbol{\varepsilon}_{k-1}^T - \mathbf{H}_k \mathbf{F}_{k-1} \mathbf{F}_{k-2} \mathbf{P}_{\mathbf{x},k-2}^{+} \mathbf{F}_{k-2}^T \mathbf{H}_{k-1}^T \\
\mathbf{A}_{k-1}\mathbf{q} &= \sum_p \mathbf{q}_p \text{vec}(\mathbf{H}_k \mathbf{F}_{k-1} \mathbf{Q}_p \mathbf{H}_{k-1}^T) \\
\mathbf{Q}_k^{filt} &= \mathbf{Q}_{k-1}^{filt} + \delta(\mathbf{Q}_{k-1}^e - \mathbf{Q}_{k-1}^{filt})
\end{aligned}$$

In line with our hypothesis stated in the introduction, we anticipate that the adaptive unscented Kalman filter will exhibit favorable performance in continuous processes.

However, its performance may be less satisfactory in batch processes, because it typically assumes stationary $\mathbf{w}_k$. Conversely, the scaled UT procedure has the potential to excel in batch cases, as it takes into account uncertainty in parameters, and the noise is non-stationary. To investigate these two methods, we plan to conduct tests on a fed-batch bioreactor case, which will be elaborated on the subsequent chapters.

5 Case study

In order to gain a deeper understanding of the diverse noise estimation techniques discussed in the preceding sections, we will examine a Monod-growth model. This model serves to estimate biomass formation, sugar consumption, and carbon dioxide formation in a fed-batch bacterial cultivation process. The investigation of this bioprocess has been carried out by Tuveri et.al[33]. Their study involves a comparative evaluation of the performance of UKF in contrast to the EKF within this specific context.

5.1 Motivation

Implementing accurate and reliable state estimators for bioprocesses are important, since it is often difficult or expensive to take measurements of some states. In the context of yeast production, optimizing the biomass yield becomes crucial, with potential increases from approximately 20% to around 50% achievable by maintaining glucose concentrations below a certain threshold[19]. This optimization is essential due to the metabolic shift of yeast from oxidative to oxidative-reductive pathways, leading to the generation of byproducts such as ethanol and acetate when substrate concentrations exceed the critical level[31]. Consequently, the implementation of automated processes through the integration of feedback controllers becomes important, allowing for the avoidance of manual and time-consuming operations.

Ideally, incorporating all relevant process variables into the feedback control strategy would ensure the desired performance of the fermentation process. However, the lack of online sensors poses a significant obstacle to the successful implementation of feedback controllers. While high-frequency measurements are crucial for monitoring and control applications, obtaining frequent measurements for all states is not always feasible. Therefore, sensor fusion presents a viable approach to overcome this

challenge. However, effective sensor fusion necessitates the fusion of measurements and a state model, highlighting the need for accurate noise estimation to improve the model. This enhancement enables the extrapolation of information regarding unmeasured states, facilitating a more comprehensive understanding and control of the fermentation process.

5.2 System model

The bioprocess system is a Monod-growth model, which accounts for growth on a single sugar substrate. It incorporates factors such as linear cell death and considers the effects of dilution during feeding. The system is characterized by a set of non-linear equations that govern its dynamics, and a corresponding measurement function that relates to the process observations. The formulation of this system is presented below:

$$\dot{\boldsymbol{x}} = \mathbf{f}(\mathbf{x}, \mathbf{u}, \boldsymbol{\theta}, t) = \begin{bmatrix} \dot{V} \\ \dot{X} \\ \dot{S} \\ \dot{CO_2} \end{bmatrix} = \begin{bmatrix} F_{in} \\ \frac{F_{in}}{V} X + \mu_{max} \frac{S}{K_S + S} X - k_d X \\ \frac{F_{in}}{V} (S_{in} - S) - \mu_{max} \frac{S}{K_S + S} \frac{X}{Y_{XS}} \\ \frac{1}{V_r - V} \left(\mu_{max} \frac{S}{K_S + S} \frac{X}{Y_{XCO_2}} V + (F_{in} - q_{air}) CO_2 \right) \end{bmatrix} \tag{5.1}$$

$$\mathbf{y}(\mathbf{x}) = \begin{bmatrix} V \\ X \\ CO_2 \end{bmatrix} \tag{5.2}$$

Where V, X, S, and CO_2 are the volume, biomass formation, sugar consumption, and carbon dioxide, respectively. F_{in} refers to the flow rate in the reactor and serves as an input to the system. S_{in} denotes the substrate concentration in the input stream, μ_{max}, K_S, k_d, Y_{XS}, and Y_{XCO_2} are time-invariant parameters detailed in Table 5.1 These parameters have been obtained from the work of Tuveri et.al[33], and remain consistent

throughout the analysis.

Table 5.1: The model parameters of the bioprocess system given in Equation (5.1), with estimated mean values from Tuveri et.al[33]. The units and standard deviations are also presented.

Parameters	*Description*	*Mean*	*Units*	$Std(\sigma)$
μ_{max}	Maximum growth rate	$1.9 \cdot 10^{-1}$	h^{-1}	$3.25 \cdot 10^{-6}$
K_S	Monod growth constant	$7.00 \cdot 10^{-3}$	gL^{-1}	$3.92 \cdot 10^{-6}$
k_d	Death rate constant	$6.00 \cdot 10^{-3}$	h^{-1}	$4.49 \cdot 10^{-6}$
Y_{XS}	S from X yield	$4.2 \cdot 10^{-1}$	gg^{-1}	$3.58 \cdot 10^{-6}$
Y_{XCO_2}	CO_2 from X yield	$5.43 \cdot 10^{-1}$	gg^{-1}	$2.22 \cdot 10^{-6}$
V_r	-	4	L	-
S_{in}	Substrate concentration in the input	100	$\frac{g}{l}$	-
q_{air}	The air inflow	$5.43 \cdot 10^{-1}$	$\frac{NL}{h}$	-

6 Results & Discussion

In our analysis of the bioprocess model, a measurement was acquired at a rate of $1min^{-1}$, and the simulation interval spans were from $0 - 15h$. The integration of the process model was carried out using a fourth-order Runge-Kutta method with adaptive step size. The system and state estimators was initialized with the following initial conditions:

$$\mathbf{x_0} = [1.5, \quad 1.2, \quad 20, \quad 10^{-9}]^T$$
$$\mathbf{P_0^+} = diag([10^{-1}, \quad 5 \cdot 10^{-1}, \quad 1, \quad 10^{-2}])$$
$$\hat{\mathbf{x}}_\mathbf{0}^+ = \mathbf{x_0} + \mathcal{N}(\mathbf{x_0}, \mathbf{P_0^+})$$

It was assumed that the measurement noise $\mathbf{v} \sim \mathcal{N}(0, \mathbf{R})$, was known from the error statistics of the measurement device and is additive with the given measurement covariance matrix $\mathbf{R}$:

$$\mathbf{R} = diag([10^{-2}, \quad 1, \quad 10^{-3}])$$

The initial process covariance matrix for both estimation methods was set to:

$$\mathbf{Q} = diag([10^{-6}, \quad 10^{-6}, \quad 10^{-6}, \quad 10^{-6}])$$

6.1 The performance between the model-based and data-driven estimation method

The aim of this analysis was to evaluate the performance of the data-driven and model-based method as described in Section 4, to estimate the covariance matrix $\mathbf{Q}$ in

the context of the model-based method. We tried to capture the inherent characteristics of the bioprocess system by incorporating random variations into the plant parameters as described in Section 2.5, by sampling the value of the parameter at each point in time given as:

$$\begin{aligned}
\bar{\boldsymbol{\theta}} &= [\mu_{max}, K_s, K_d, Y_{XS}, Y_{XCO_2}, S_{in}, q_{air}] \\
\mathbf{P}_{\boldsymbol{\theta}} &= diag(\bar{\boldsymbol{\theta}}^{\,2}) \\
\boldsymbol{w}_{\boldsymbol{\theta}} &\sim \mathcal{N}(\bar{\boldsymbol{\theta}}, \mathbf{P}_{\boldsymbol{\theta}}) \\
\boldsymbol{\theta} &= \bar{\boldsymbol{\theta}} + \boldsymbol{w}_{\boldsymbol{\theta}}
\end{aligned}$$

where $\bar{\boldsymbol{\theta}}$ are the model parameter given in Table 5.1, and is used by the nonlinear model, and $\boldsymbol{\theta}_{plant}$ is used by the plant. The uncertainty in the parameter could then be translated to create an additive process noise to estimate the noise statistic as the procedure described in Section 4.1.1.

We have employed the scaled unscented Kalman filter for both noise methods in the analysis, as the estimation of the noise matrix $\mathbf{Q}$ is the primary objective. This allows for a more comparable evaluation of the methods in the prediction of $\mathbf{Q}$. The value of the tuning parameters were selected according to Section 3.2.1.1, the parameter α was selected to be $\alpha = 10^{-3}$, ranging between 0 and 1, in order to minimize potential higher-order effects. The value of β is set to 2, as we assumed that the noise follows a Gaussian distribution, in accordance with Kandepu et al.[16]. This choice of β influences the weighting of the zeroth sigma-point during the calculation of the covariance. Furthermore, κ was set to 1 to ensure the covariance matrix remains positive semidefinite.

In the case of the data-driven method, additive noise was assumed to enter through $\mathbf{w} \sim \mathcal{N}(0, \mathbf{Q})$, and we adopted a diagonal parameterization using n matrices ($\mathbf{Q}_p = \mathbf{E}_{pp}$), where $\mathbf{E}_{ij}$ was an elementary matrix with its only non-zero entry being 1 at the

ij position. The value of δ was chosen as $1/50$, and was determined through a trial and error approach to identify an appropriate small value in order to achieve a smoothing effect on the moving average.

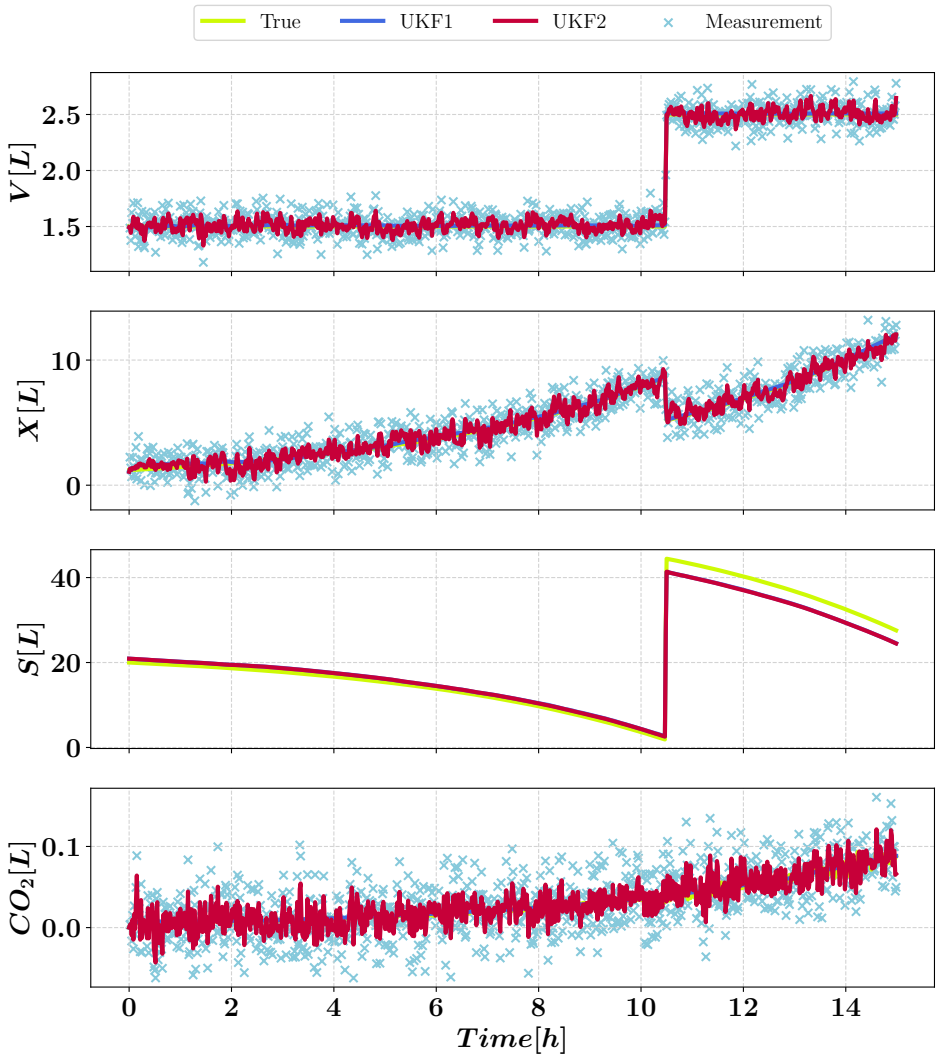


Figure 6.1: The plot illustrates the state estimation performance obtained from different approaches to estimate the covariance matrix **Q** for the bioprocess system. The plant is represented by the green line, while the model-based estimation of **Q** using the scaled unscented transformation is depicted by the blue line (UKF1). Additionally, the red line corresponds to the data-driven estimation of **Q** using an adaptive scaled unscented Kalman filter (UKF2), and the measurements are marked as x.

Figure 6.1 illustrates the behaviour of the plant through the green line, while the blue line (UKF1) represents the state estimation using **Q** matrix obtained from the model-based method. Additionally, the red line indicates the states estimated using the **Q** from the data-driven method. The measured values are denoted by the x markers.

Visually, the performance of both methods appeared quite similar in Figure 6.1. Both methods demonstrated a high level of accuracy in estimating the behaviour of the sugar concentration, which consistently decreased from the initial state $(20g/L)$, while the volume remained constant at $1.5L$, and both methods effectively captured this trend. However, when the sugar concentration fell below $2g/L$, a $0.5L$ step change in the volume of the sugar solution was introduced into the feed stream, F_{in}. At that point, both methods deviated from the actual plant behaviour.

This deviation was more prominently observed in the error plot, Figure 6.2, which represented the discrepancy between the actual plant state, $\mathbf{x}_{k,plant}$ and the estimated state, $\hat{\mathbf{x}}_k$. The deviation of the sugar concentration became more pronounced at around 10.5h and remained high thereafter.

To further quantify the errors, we also computed the root-mean-square error.

$$RMSE = \sqrt{\frac{1}{N}\sum_{k=1}^{N}\left\|\left(\hat{\mathbf{x}}_k - \mathbf{x}_{k,plant}\right)\right\|^2} \tag{6.1}$$

where $\mathbf{x}_{k,plant}$ is the ground truth, and the values are given in Table 6.1.

Table 6.1: The Root Mean Square Error (RMSE) value from the state estimation using the model-based method (UKF1) and the data-driven (UKF2) method gathered from 1, and the mean of 100 simulations are presented in their respective columns.

State variables	$\text{RMSE}_{\text{UKF1 (1)}}$	$\text{RMSE}_{\text{UKF2 (1)}}$	$\text{RMSE}_{\text{UKF1 (100)}}$	$\text{RMSE}_{\text{UKF2 (100)}}$
V	0.0090	0.0533	0.0087	0.0538
X	0.0120	0.5350	0.0925	0.5261
S	1.8401	1.8532	1.1400	8.9797
CO_2	0.0025	0.0145	0.0025	0.0170

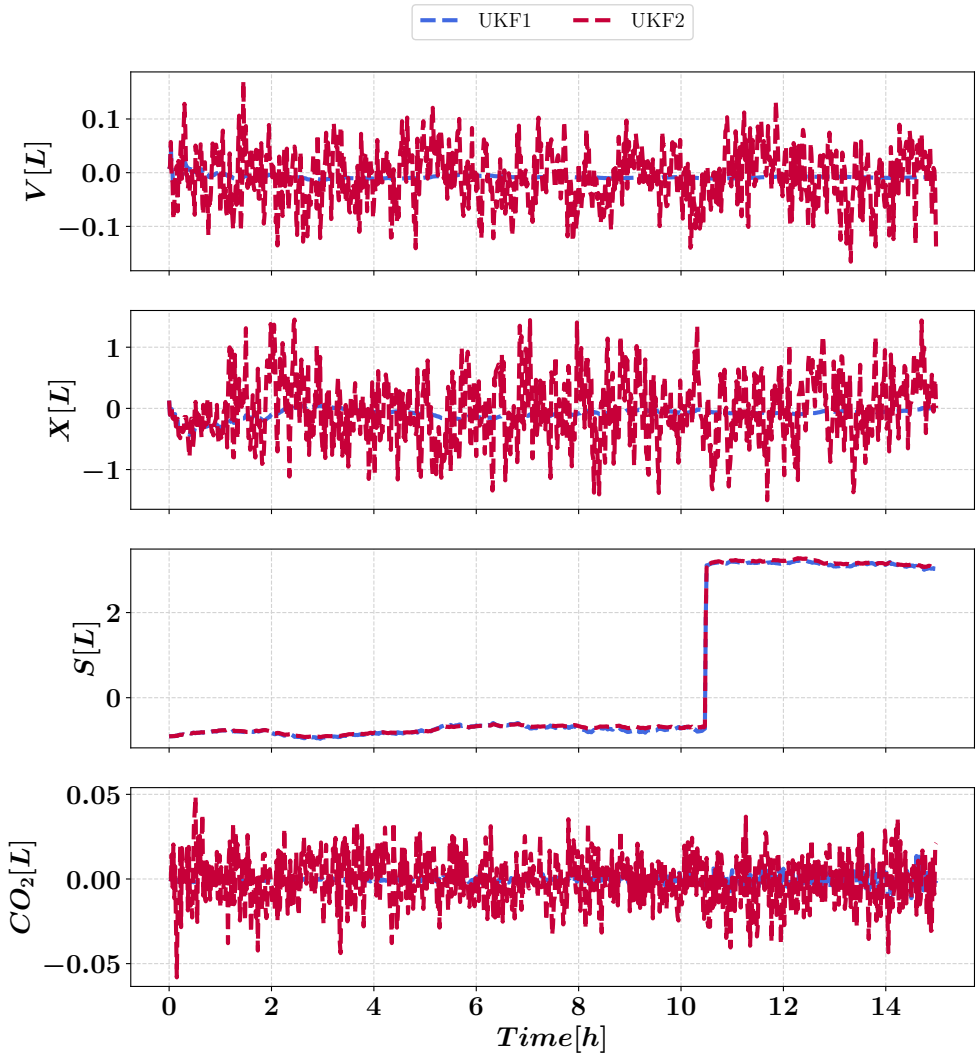


Figure 6.2: The error plot illustrates the deviation between $\mathbf{x}_{k,plant} - \hat{\mathbf{x}}_k$, the blue dashed line represents the deviation of the model-based estimation, while the red dashed line corresponds to the deviation of the data-driven estimation.

Based on the RMSE values presented in Table 6.1, it is evident that the model-based method described in Section 4.1.1, achieved the most accurate state estimates. This finding aligns with our hypothesis, as stated at the end of Section 4.2. By analyzing Figure 6.3, it is apparent that the data-driven method exhibited more noise in the diagonal elements compared to the model-based method. One possible reason for this difference is because of the tuning of δ, which might be improved through Bayesian

optimization. Additionally, the value of $\mathbf{Q}_p$ was tuned through parameterization. Alternative approaches to tuning $\mathbf{Q}_p$, may lead to even better performance outcomes.

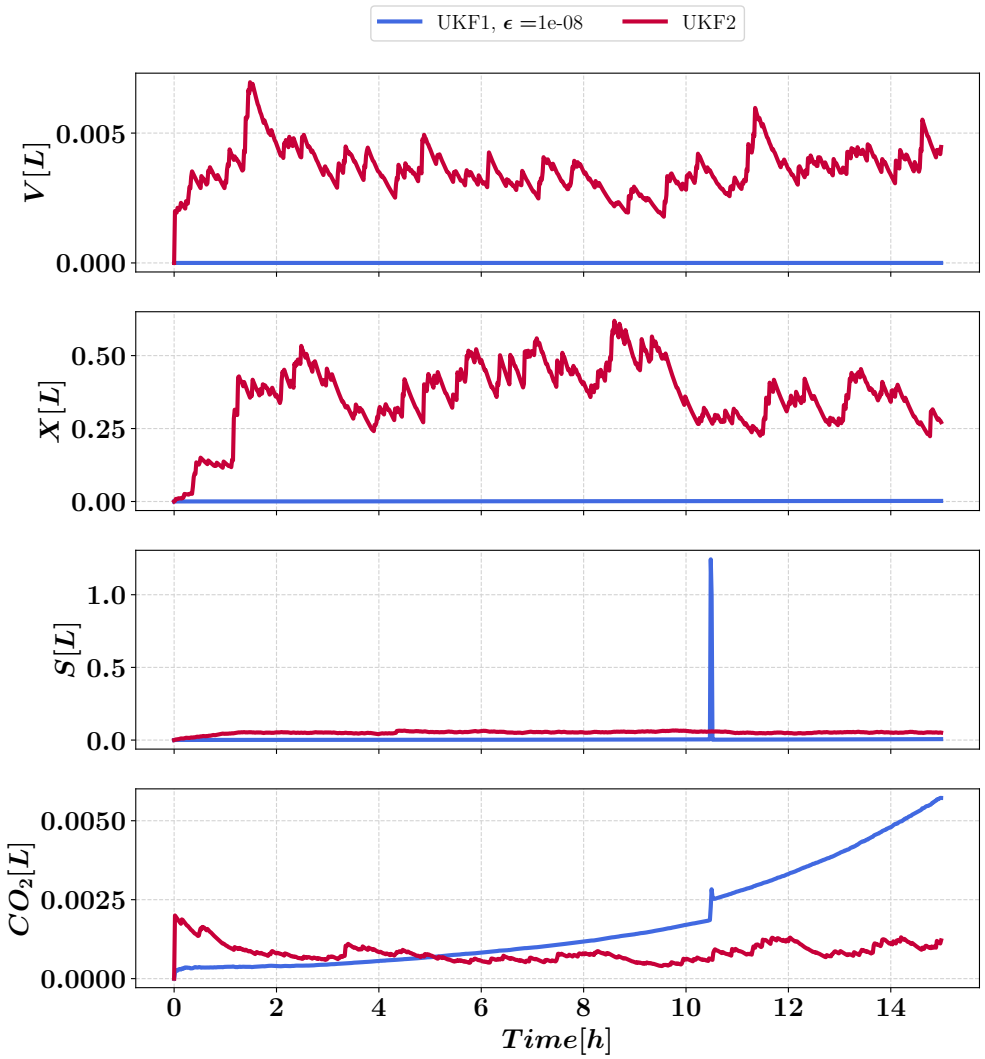


Figure 6.3: The diagonal elements of the covariance matrix $\mathbf{Q}$ obtained from the estimation of the model-based method is represented by the blue line (UKF1), and the estimation derived from the data-driven method is illustrated by the red line (UKF2).

By further analyzing Figure 6.3, it becomes evident that the model-based method, having access to perfect information about the parameter distribution, successfully predicted the spike in the sugar concentration around 10.5h. In scenarios where perfect information about the parameter distribution is not available, one potential

approach to achieve favorable results could involve tuning the parameter κ.

7 Concluding Remarks

The performance of the model-based method and the data-driven method, as discussed by the works of Krog & Jäschke[21], and Berry & Sauer[8], demonstrates promising results as estimation techniques for the covariance matrix $\mathbf{Q}$, thereby reducing the need for manual tuning of $\mathbf{Q}$. Among the explored approaches, the estimation technique given by Krog & Jäschke[21], which utilizes the scaled unscented transformation to estimate the covariance matrix $\mathbf{Q}$, yielded the overall best results.

However, it is worth noting that the model-based method still holds potential for improvement. By selecting better parameters for δ, and $\mathbf{Q}_p$, it is possible to achieve better results. Nevertheless, in cases of the model-based method, if the model uncertainties biased the prediction of states significantly, the methods could result in noisy estimates. Due to the fact that in such case the values of $\mathbf{Q}$ will become very large, and the filter will rely heavily on the measurements, capturing the noise present in the measurements. Therefore, there exists a limit to the process-model mismatch beyond which these techniques may struggle to deliver accurate state estimates.

References

[1] Shady E. Ahmed, Suraj Pawar and Omer San. 'PyDA: A Hands-On Introduction to Dynamical Data Assimilation with Python'. In: *Fluids* (2020).

[2] K.J. Åström and P. Eykhoff. 'System identification—A survey'. In: *Automatica* 7.2 (1971), pp. 123–162. ISSN: 0005-1098. DOI: https://doi.org/10.1016/0005-1098(71)90059-8. URL: https://www.sciencedirect.com/science/article/pii/0005109871900598.

[3] Pierre R. Bélanger. 'Estimation of noise covariance matrices for a linear time-varying stochastic process'. In: *Automatica* 10.3 (1974), pp. 267–275. ISSN: 0005-1098. DOI: https://doi.org/10.1016/0005-1098(74)90037-5. URL: https://www.sciencedirect.com/science/article/pii/0005109874900375.

[4] Tyrus Berry and Timothy Sauer. 'Adaptive ensemble Kalman filtering of non-linear systems'. In: *Tellus A: Dynamic Meteorology and Oceanography* 65.1 (2013), p. 20331. DOI: 10.3402/tellusa.v65i0.20331. eprint: https://doi.org/10.3402/tellusa.v65i0.20331. URL: https://doi.org/10.3402/tellusa.v65i0.20331.

[5] Francesco Destro et al. 'A hybrid framework for process monitoring: Enhancing data-driven methodologies with state and parameter estimation'. In: *Journal of Process Control* 92 (Aug. 2020), pp. 333–351. ISSN: 0959-1524. DOI: 10.1016/J.JPROCONT.2020.06.002.

[6] A. Doucet, Simon J. Godsill and Christophe Andrieu. 'On sequential simulation-based methods for Bayesian filtering'. In: *Statistics and Computing* (1998).

[7] Jindrich Dunik et al. 'Noise covariance matrices in state-space models: A survey and comparison of estimation methods—Part I'. In: *International Journal of Adaptive Control and Signal Processing* 31 (May 2017). DOI: 10.1002/acs.2783.

[8] Donald Ebeigbe et al. 'A Generalized Unscented Transformation for Probability Distributions'. In: *ArXiv* (2021).

[9] Jake Fotopoulos, Christos Georgakis and Harvey G. Stenger. ‘Use of tendency models and their uncertainty in the design of state estimators for batch reactors1This contribution is dedicated to the remembrance of Professor Jacques Villermaux.1’. In: *Chemical Engineering and Processing: Process Intensification* 37.6 (1998), pp. 545–558. ISSN: 0255-2701. DOI: https://doi.org/10.1016/S0255-2701(98)00061-0. URL: https://www.sciencedirect.com/science/article/pii/S0255270198000610.

[10] Arthur Gelb et al. *Applied optimal estimation*. MIT press, 1974.

[11] Ravindra D. Gudi, Sirish L. Shah and Murray R. Gray. ‘Adaptive multirate state and parameter estimation strategies with application to a bioreactor’. In: *Aiche Journal* 41 (1995), pp. 2451–2464.

[12] Simon J. Julier. ‘The scaled unscented transformation’. In: *Proceedings of the 2002 American Control Conference (IEEE Cat. No.CH37301)* 6 (2002), 4555–4559 vol.6.

[13] Simon J. Julier and Jeffrey K. Uhlmann. ‘A General Method for Approximating Nonlinear Transformations of Probability Distributions’. In: 1996.

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

[15] Rudolf E. Kálmán. ‘A new approach to linear filtering and prediction problems" transaction of the asme journal of basic’. In: 1960.

[16] Rambabu Kandepu, Bjarne Foss and Lars Imsland. ‘Applying the unscented Kalman filter for nonlinear state estimation’. In: *Journal of Process Control* 18.7 (2008), pp. 753–768. ISSN: 0959-1524. DOI: https://doi.org/10.1016/j.jprocont.2007.11.004. URL: https://www.sciencedirect.com/science/article/pii/S0959152407001655.

[17] Rangasami L. Kashyap. ‘Maximum likelihood identification of stochastic linear systems’. In: *IEEE Transactions on Automatic Control* 15 (1970), pp. 25–34.

[18] Youngjoo Kim and Hyochoong Bang. 'Introduction to Kalman Filter and Its Applications'. In: *Introduction and Implementations of the Kalman Filter*. Ed. by Felix Govaers. Rijeka: IntechOpen, 2018. Chap. 2. DOI: 10.5772/intechopen.80600. URL: https://doi.org/10.5772/intechopen.80600.

[19] C. Klockow et al. 'ARE MONOD MODELS ENOUGH FOR BIOREACTOR CONTROL? PART I – EXPERIMENTAL RESULTS'. In: *IFAC Proceedings Volumes* 40.4 (2007). 10th IFAC Symposium on Computer Applications in Biotechnology, pp. 331–336. ISSN: 1474-6670. DOI: https://doi.org/10.3182/20070604-3-MX-2914.00057. URL: https://www.sciencedirect.com/science/article/pii/S1474667016328993.

[20] S. Kolås, B.A. Foss and T.S. Schei. 'Noise modeling concepts in nonlinear state estimation'. In: *Journal of Process Control* 19.7 (2009), pp. 1111–1125. ISSN: 0959-1524. DOI: https://doi.org/10.1016/j.jprocont.2009.03.002. URL: https://www.sciencedirect.com/science/article/pii/S0959152409000419.

[21] Halvor Aarnes Krog and Johannes Jäschke. 'Systematic Estimation of Noise Statistics for Nonlinear Kalman Filters'. In: *IFAC-PapersOnLine* 55.7 (2022). 13th IFAC Symposium on Dynamics and Control of Process Systems, including Biosystems DYCOPS 2022, pp. 19–24. ISSN: 2405-8963. DOI: https://doi.org/10.1016/j.ifacol.2022.07.416. URL: https://www.sciencedirect.com/science/article/pii/S2405896322008175.

[22] Demetrios G. Lainiotis. 'Optimal adaptive estimation: Structure and parameter adaption'. In: *IEEE Transactions on Automatic Control* 16 (1971), pp. 160–170.

[23] Raman K. Mehra. 'On the identification of variances and adaptive Kalman filtering'. In: *IEEE Transactions on Automatic Control* 15 (1970), pp. 175–184.

[24] Kenneth A. Myers and Byron D. Tapley. 'Adaptive sequential estimation with unknown noise statistics'. In: *IEEE Transactions on Automatic Control* 21 (1976), pp. 520–523.

[25] William H. Press et al. ‘Solution of Linear Algebraic Equations’. In: *Numerical Recipes in C The Art of Scientific Computing*. Press Syndicate of the University of Cambridge The Pitt Building, 2002. ISBN: 0 521 43108 5.

[26] Simo Särkkä and Aapo Nummenmaa. ‘Recursive Noise Adaptive Kalman Filtering by Variational Bayesian Approximations’. In: *IEEE Transactions on Automatic Control* 54 (2009), pp. 596–600.

[27] Tor Schei. ‘On-line estimation for process control and optimization applications’. In: *Journal of Process Control* 18 (Oct. 2008), pp. 821–828. DOI: 10.1016/j.jprocont.2008.06.014.

[28] René Schneider and Christos T. Georgakis. ‘How To NOT Make the Extended Kalman Filter Fail’. In: *Industrial & Engineering Chemistry Research* 52 (2013), pp. 3354–3362.

[29] H. Schuler. ‘Estimation of States in a Polymerization Reactor’. In: *IFAC Proceedings Volumes* 13.4 (1980). 4th IFAC Conference on Instrumentation and Automation in the Paper, Rubber, Plastics and Polymerisation Industries, Ghent, Belgium, 3-5 June 1980, pp. 369–376. ISSN: 1474-6670. DOI: https://doi.org/10.1016/S1474-6670(17)64951-6. URL: https://www.sciencedirect.com/science/article/pii/S1474667017649516.

[30] Dan Simon. *Matematisk modellering*. John Wiley & Sons, Inc, 2016. ISBN: 10 0-47 1-70858-5.

[31] Bernhard Sonnleitner and Othmar Käppeli. ‘Growth of Saccharomyces cerevisiae is controlled by its limited respiratory capacity: Formulation and verification of a hypothesis’. In: *Biotechnology and Bioengineering* 28 (1986).

[32] Peter Terwiesch, Mukul Agarwal and David W.T. Rippin. ‘Batch unit optimization with imperfect modelling: a survey’. In: *Journal of Process Control* 4.4 (1994), pp. 238–258. ISSN: 0959-1524. DOI: https://doi.org/10.1016/0959-1524(94)80045-6. URL: https://www.sciencedirect.com/science/article/pii/0959152494800456.

[33] Andrea Tuveri et al. ‘Sensor fusion based on Extended and Unscented Kalman Filter for bioprocess monitoring’. In: *Journal of Process Control* 106 (2021), pp. 195–207. ISSN: 0959-1524. DOI: https://doi.org/10.1016/j.jprocont.2021.09.005. URL: https://www.sciencedirect.com/science/article/pii/S0959152421001542.

[34] Jaleel Valappil and Christos Georgakis. ‘Systematic estimation of state noise statistics for extended Kalman filters’. In: *AIChE Journal* 46.2 (2000), pp. 292–308. DOI: https://doi.org/10.1002/aic.690460209. eprint: https://aiche.onlinelibrary.wiley.com/doi/pdf/10.1002/aic.690460209. URL: https://aiche.onlinelibrary.wiley.com/doi/abs/10.1002/aic.690460209.

[35] P. de Vallière and Dominique Bonvin. ‘Application of estimation techniques to batch reactors—III. Modelling refinements which improve the quality of state and parameter estimation’. In: *Computers & Chemical Engineering* 14 (1990), pp. 799–808.

[36] Anna Voelker, Konstantinos Kouramas and Efstratios N. Pistikopoulos. ‘Moving horizon estimation: Error dynamics and bounding error sets for robust control’. In: *Automatica* 49.4 (2013), pp. 943–948. ISSN: 0005-1098. DOI: https://doi.org/10.1016/j.automatica.2013.01.008. URL: https://www.sciencedirect.com/science/article/pii/S0005109813000095.

A Definitions & Theorems

Theorem A.1 *(Isserlis' Theorem) Let* $\mathbf{x} = (x_1, x_2, \ldots, x_n)$ *be a zero-mean multivariate normal random vector (i.e. for every "i",* $\mathbb{E}[x_i] = 0$ *and* $\mathbb{E}[x_i^2] = 1$*), if n is even then:*

$$\mathbb{E}[x_1, x_2, \ldots, x_n] = \sum_{p \in P_n^2} \prod_{\{i,j\ \in p\}} \mathbb{E}[x_i x_j] = \sum_{p \in P_n^2} \prod_{\{i,j\ \in p\}} \boldsymbol{Cov}(x_i x_j)$$

and if n is odd then:

$$\mathbb{E}[x_1, x_2, \ldots, x_n] = 0$$

Definition A.1 *Let* $\mathbf{x} = (x_1, x_2, \ldots, x_n)$ *be a zero-mean multivariate normal random vector, and k be some positive integer. We define the element-wise product (Hadamard product)* $\odot$*, such that:*

$$\mathbf{x}^{\odot k} = \underbrace{\mathbf{x} \odot \mathbf{x} \odot \cdots \odot \mathbf{x}}_{k \text{ times}} \tag{A.1}$$

$$\mathbf{x}^{\odot -k} = \left(\underbrace{\mathbf{x} \odot \mathbf{x} \odot \cdots \odot \mathbf{x}}_{k \text{ times}} \right)^{-1} \tag{A.2}$$

Definition A.2 *Let* $\mathbf{A}$ *and* $\mathbf{B}$ *be two matrices of the same dimension* $m \times n$*, the Hadamard product is a matrix of the same dimension as the operands, with elements*

given as follows:

$$\mathbf{A} \odot \mathbf{B} = (\mathbf{A} \odot \mathbf{B})_{i,j} = (\mathbf{A})_{i,j}(\mathbf{B})_{i,j} \tag{A.3}$$

Definition A.3 *Let* **A**, **B** *and* **C** *be three matrices of the same dimension* $m \times n$*, the Hadamard division is given such that:*

$$\mathbf{C} = \mathbf{A} \oslash \mathbf{B} \tag{A.4}$$

$$\mathbf{C}_{i,j} = \frac{\mathbf{A}_{i,j}}{\mathbf{B})_{i,j}} \tag{A.5}$$

B Code Listing

```
# @author: Abubakar Bampoye
# Importing requierd modules
#------------------------------------------------------------------
import numpy as np
import scipy.linalg
#------------------------------------------------------------------

def weights(x_posteriori):
    """
    This function calculates the weights for the unsented Kalaman
    filter.
    ------------------------------------------------------------------


    Parameters:
    ------------------------------------------------------------------
    x_posteriori : The current best guess for the mean of x.


    Return:
    ------------------------------------------------------------------
    An ndarray of weighting coefficients.
    """

    N_dim          = np.shape(x_posteriori)[0]  # The dimension of
   the state x
    alpha          = 1e-3 # positive scaling parameter which can be
   made arbitrarily small (0 < Îś < 1), controls the size of the
   sigma-point distribution
    beta           = 2.   # optimal choice for a Gaussian prior
    kappa          = np.max([3 - N_dim, 0]) # scaling parameter
    lambda_        = (alpha**2)*(N_dim + kappa) - N_dim
    N_sigma_points = 2 * N_dim + 1  # The nummber of sigmapoints
```

```
    Wm              = np.zeros(N_sigma_points)

    for i in range(1, N_sigma_points):
        Wm[i] = 1/(2*(N_dim + lambda_))

    Wm[0]           = lambda_/(N_dim + lambda_)
    Wc              = Wm.copy()  # Wc: The weighting coefficients for
    each sigma point for the covariance
    Wc[0]           = lambda_/(N_dim + lambda_) + (1 - alpha**2 +
    beta)  # Wc: The weighting coefficients for each sigma point for
    the covariance

    return Wm, Wc


def sigma_points(x_posteriori, P_posteriori):
    """
    Calculating sigma points for an uncented Kalman filter,
    where x_posteriori is the mean, and P_posteriori is the
    covariance of the filter. Kappa could be an arbitrary constant.
    ------------------------------------------------------------


    Parameters:
    ------------------------------------------------------------
    x_posteriori : The current best guess for the mean of x.

    P_posteriori : The current best guess for the covariance of x.


    Return:
    ------------------------------------------------------------
    An ndarray of sigma points.
    """
    dim_x = x_posteriori.shape[0]
```

```
    assert (P_posteriori.shape[0] == dim_x) and (P_posteriori.shape
    [1] == dim_x), f"The shape of P_posteriori = {P_posteriori.shape
    } is wrong, it must be ({dim_x, dim_x})"
    N_dim                = np.shape(x_posteriori)[0]  # The
    dimension of the state x
    alpha                = 1e-3 # positive scaling parameter which
    can be made arbitrarily small (0 < Îś < 1), controls the size of
     the sigma-point distribution
    kappa                = np.max([3 - N_dim, 0]) # scaling
    parameter
    lambda_              = alpha**2*(N_dim + kappa) - N_dim
    N_sigma_pointa       = 2 * N_dim + 1  # The nummber of
    sigmapoints
    X                    = np.zeros((N_dim, N_sigma_pointa))  #
    Sigmapoints
    sqrt_factor          = np.sqrt(N_dim + lambda_)
    x_posteriori_tilde = sqrt_factor*scipy.linalg.cholesky(
    P_posteriori, lower=True)
    X[:, 0]              = x_posteriori

    for i in range(N_dim):
        X[:, i+1]       = x_posteriori + x_posteriori_tilde[:, i]
        X[:, N_dim+i+1] = x_posteriori - x_posteriori_tilde[:, i]

    return X
```

Code 1: Functions for computing the scaled sigma points.

```
# @author: Abubakar Bampoye
# Importing requierd modules
#-----------------------------------------------------------------
import numpy as np
import scipy.linalg
#-----------------------------------------------------------------


def constrained_generalized_sigmas(x_bar, P, S_skew=None, P_kurt=
```

```
    None,
                                        lb=None, ub=None, theta=None):
    """
    This function generates generalized sigma points as described by
    Ebeige.

    ------------------------------------------------------------------



    Parameters:

    ------------------------------------------------------------------

    x_bar : The current best guess for the mean of x.

    P : The current best guess for the covariance of x.

    S_skew : A vector of diagonal components of the skewness tensor.

    P_kurt: A vector of diagonal components of the kurtiosis tensor.

    lb : A vector of lower bound of the state.

    ub : A vector of upper bound of the state.


    Return:

    ------------------------------------------------------------------

    A tuple of ndarray consisting of sigma points, and the
    corresponding
    weights of each sigma points.
    """
```

```
    dim_x = x_bar.shape[0]
    assert (P.shape[0] == dim_x) and (P.shape[1] == dim_x), f'''The shape
    of P_k_posteriori = {P.shape} is wrong, it must be ({dim_x, dim_x})'''

    # Assumes that the distribtions are symmetrical
    if S_skew == None:
        S_skew = np.zeros((1, x_bar.shape[0]))


    if P_kurt == None:
        # Using IsserlisâĂŹ Theorem to compute the P_kurt
        P_kurt = np.diag(P@(np.trace(P)*np.eye(x_bar.shape[0])) + 2*P)


    N      = np.shape(x_bar)[0]  # Acquire the number of states

    try:
        P_sqrt = scipy.linalg.cholesky(P, lower=True) # Evaluate the matrix square root

    except np.linalg.LinAlgError as e:
        P = (P + P.T)/2
        P_sqrt = scipy.linalg.cholesky(P, lower=True) # Evaluate the matrix square root

    std_P  = np.diag(P_sqrt)
    u      = 0.5*((-std_P**(-3))*S_skew + np.sqrt(4*((std_P**(-4))*P_kurt) -
             3*((std_P**(-3))*S_skew)**2)).flatten()  # Choosing the free parameter vector

    assert np.all(u>0),"Choose another paramter vector (u>0)."  # checking if the vector u > 0
```

```
66
    v = (u + (std_P**(-3))*S_skew).flatten()  # Calculating the
    parameters v
68


70  # Calculating the 2n + 1 sigma points
    N_sigma_pointa = 2 * N + 1  # The nummber of sigmapoints
72  X              = np.zeros((N, N_sigma_pointa))  # Sigmapoints
    X[:, 0]        = x_bar
74
    for i in range(1,N):
76      X[:, i+1]   = x_bar - u[i]*P_sqrt[:,i]
        X[:, i+N+1] = x_bar + v[i]*P_sqrt[:,i]
78
    # Calculating the weights
80  weight2 = 1/v/(u + v)
    weight1 = weight2*v/u
82  w0      = np.array([1 - (np.sum(weight1) + np.sum(weight2))])
    weights = np.append(w0, np.hstack((weight1, weight2)))
84


86  # Creating the constraints
    # For the bounds, we note that lb < x < ub
88  # Handle if the slack parameter is not given
    if theta == None:
90      theta = 0.9 # Default value of user defined slack parameter

92  # Handle if the lower bound is not given
    if lb == None:
94      lb=-np.PINF

96  # Handle if the upper bound is not given
    if ub == None:
98      ub=np.PINF

100 assert np.all(np.greater_equal(x_bar, lb)) and np.all(np.
```

```
    greater_equal(
        ub, x_bar)), '''Unable fo enforce constraints: one or more
    of the
    mean does not satisfy lb < mean < ub.'''


    for i in range(1, 2*N):
        if np.all(X[:,i]) < lb:
            if i <= N:
                u[i]   = theta*(np.min((x_bar - lb)/P_sqrt[:,i]))
            if i > N:
                v[i-N] = theta*(np.min((lb - x_bar)/P_sqrt[:,i-N]))

    v = (u + (std_P**(-3))*S_skew).flatten() # Calculating the
    parameters v

    for i in range(1,N):
        X[:, i+1]   = x_bar - u[i]*P_sqrt[:,i]
        X[:, i+N+1] = x_bar + v[i]*P_sqrt[:,i]

    for i in range(1,2*N):
        if np.all(X[:,i]) > ub:
            if i <= N:
                u[i]   = theta*(np.min((x_bar - ub)/P_sqrt[:,i]))
            if i > N:
                v[i-N] = theta*(np.min((ub - x_bar)/P_sqrt[:,i-N]))

    v = (u + (std_P**(-3))*S_skew).flatten()  # Calculating the
    parameters v

    for i in range(1,N):
        X[:, i+1]   = x_bar - u[i]*P_sqrt[:,i]
        X[:, i+N+1] = x_bar + v[i]*P_sqrt[:,i]

    weight2 = 1/v/(u + v)
    weight1 = weight2*v/u
```

```
    w0      = np.array([1 - (np.sum(weight1) + np.sum(weight2))])
    weights = np.append(w0, np.hstack((weight1, weight2)))

    return X, weights
```

Code 2: A function that compute the constrained generalized sigma points.

```
# @author: Abubakar Bampoye
# Importing the required modules
#----------------------------------------------------------------------
import matplotlib.pyplot as plt
import numpy as np
import scaled_filter
import utils_bioreactor_additive_noise as utils_br
import generalized_filter as gen_filter
plt.rc('text', usetex=True)
plt.rcParams['text.latex.preamble'] = r"\usepackage{amsmath}"
plt.rcParams['text.latex.preamble'] = r"\boldmath"
font = {'family': 'normal', 'weight':'bold', 'size':20}
plt.rc('font', **font)
#----------------------------------------------------------------------


#----------------------------------------------------------------------
# Defining the parameters (Data) from the literature
#----------------------------------------------------------------------
mu_max      = 1.94e-1  # Maximum growth rate [h^-1]
K_s         = 7.00e-3  # Monod growth constant [gL^-1]
k_d         = 6.00e-3  # Death rate constant [h^-1]
Y_XS        = 4.2e-1   # S from X yield [gg^-1]
Y_XCO_2     = 5.43e-1  # CO_2 from X yield [gg^-1]
V_r         = 4        # [L]
S_in        = 100      # [g/l]
q_air       = 120      # [NL/h]
theta_mean  = np.array([mu_max, K_s, k_d, Y_XS, Y_XCO_2, V_r, S_in,
    q_air])
P_theta     = np.diag((theta_mean*.05)**2)
```

```
30 p_dim       = theta_mean.shape[0]


32


#-------------------------------------------------------------------
34 # Defining the sampling time (integration time span)
#-------------------------------------------------------------------
36 dt          = 1     # [1/min]
t_0           = 0     # [min]
38 t_f         = 15*60 # [min]
t             = np.arange(t_0, t_f, dt)/60 # [h]
40 dim_t       = len(t)
dt_sampling = t[1] - t[0]
42 t_span      = np.array([t[0], t[1]])


44


#-------------------------------------------------------------------
46 # Defining the initial conditions and necessary information in
# order to estimate the state
48 #-------------------------------------------------------------------
# The initial conditions from the literature
50 V0, X0, S0, CO_20     = 1.5, 1.2, 20., 10e-10
# Initial guess of the state
52 x0                    = np.array([V0, X0, S0, CO_20])
# The dimention of the state
54 dim_x                 = x0.shape[0]
# Allocating memory for the true state
56 x_true                = np.zeros((dim_x, dim_t))
# Initializing the first element of the true state
58 x_true[:, 0]          = x0
# Allocating memory for the posteriori
60 x_posteriori          = np.zeros((dim_x, dim_t))
# Allocating memory for the posteriori of the generelized
62 # sigmapoints
x_gen_posteriori      = np.zeros((dim_x, dim_t))
64 # Inital covariance of the process
P0                    = np.diag(np.square([.1, .5, 1., 1e-2]))
```

```
# Inital covariance of the posteriori
P_posteriori          = P0
# Inital covariance of the generalized posteriori
P_gen_posteriori      = P0
# Initial state of the filter
x0_filter             = np.random.multivariate_normal(x0, P0)
x0_filter             = np.array([xi if xi > 0 else 1e-10 for xi in
    x0_filter])
# Initializing the first element of the posteriori state
x_posteriori[:,0]     = x0_filter
# Initializing the first element of the generalzed
# posteriori state
x_gen_posteriori[:,0] = x0_filter
# Allocating memory for the measurements
y                     = np.zeros((3, dim_t))
#remove first measurement
y[:, 0]               = np.nan
# Creating the process noise covariance matrix
Q                     = np.diag([1e-6]*4)
# Creating the measurement noise covariance matrix
R                     = np.diag([1e-2, 1, 1e-3])
#initial control variable
u                     = np.array([.0])


#-----------------------------------------------------------------
# Integrating and estimating the states
#-----------------------------------------------------------------
for k in range(1, dim_t):
    # Creating the noise vectors
    vk = np.random.multivariate_normal([0]*3, R)
    wk = np.random.multivariate_normal([0]*4, Q)

    x_true[:, k] = utils_br.f(t_span, x_true[:, k-1], u, theta_mean)
    + wk
    y[:, k]    = utils_br.h(x_true[:, k]) + vk
```

```
100
        #ScaledUKF prediction step
102     Q_ukf = Q.copy()/dt_sampling
        f_ukf = lambda x: utils_br.f(t_span, x, u, theta_mean) + wk
104
        x_priori, P_priori, _ = scaled_filter.predict(x_posteriori[:,k
        -1], P_posteriori, f_ukf, Q_ukf)
106
        #ScaledUKF update step
108     x_posteriori [:,k], P_posteriori, _, _, _, _, _ = scaled_filter.
        update(x_priori, P_priori, utils_br.h, y[:,k], R)


110
        # UKF 2
112     f_ukf2 = lambda x: utils_br.f(t_span, x, u, theta_mean) + wk
        #GenUKF prediction step
114     x_gen_priori, P_gen_priori = gen_filter.predict(x_gen_posteriori
        [:,k-1], P_gen_posteriori, f_ukf2, Q_ukf)

116     #GenUKF update step
        x_gen_posteriori [:,k], P_gen_posteriori, _ = gen_filter.update(
        x_gen_priori, P_gen_priori, utils_br.h, y[:,k], R)
118
        # Creating the control variable
120     if x_true[2, k-1] > 2:
            u = np.array([.0])
122     else:
            u = np.array([0.5*60])
124


126 #-----------------------------------------------------------------
    # Displaying the results
128 #-----------------------------------------------------------------
    # Exctracting the states
130 V_true, X_true, S_true, CO_2_true             = x_true[0], x_true
        [1], x_true[2], x_true[3]
```

```
V_ukf, X_ukf, S_ukf, CO_2_ukf                = x_posteriori [0,:],
    x_posteriori [1,:], x_posteriori [2,:], x_posteriori [3,:]
V_gen_ukf, X_gen_ukf, S_gen_ukf, CO_2_gen_ukf = x_gen_posteriori
    [0,:], x_gen_posteriori [1,:], x_gen_posteriori [2,:],
    x_gen_posteriori [3,:]
fig, (ax1, ax2, ax3, ax4) = plt.subplots(4, 1, figsize=(16, 8),
    sharex=True)

ax1.plot(t, V_true ,color="#CEFA05", label="True", linewidth=3,
    linestyle="-")
ax1.plot(t, V_ukf, color="#4169e1", label="ScaledUKF", linewidth=3)
ax1.plot(t, V_gen_ukf, color="#C70039", label="GenUKF", linewidth=3)
ax1.scatter(t, y[0], c="#86c9db", marker='x', label="Measurement")
ax1.set_ylabel("$V [L]$")
ax1.grid(color='lightgray',linestyle='--')

ax2.plot(t, X_true, color="#CEFA05", label="True", linewidth=3,
    linestyle="-")
ax2.plot(t, X_ukf, color="#4169e1", label="ScaledUKF", linewidth=3)
ax2.plot(t, X_gen_ukf, color="#C70039", label="GenUKF", linewidth=3)
ax2.scatter(t, y[1], c="#86c9db", marker='x', label="Measurement")
ax2.set_ylabel("$X [L]$")
ax2.grid(color='lightgray',linestyle='--')

ax3.plot(t, S_true, color="#CEFA05", label="True", linewidth=3,
    linestyle="-")
ax3.plot(t, S_ukf, color="#4169e1", label="ScaledUKF", linewidth=3)
ax3.plot(t, S_gen_ukf, color="#C70039", label="GenUKF", linewidth=3)
ax3.set_ylabel("$S [L]$")
ax3.grid(color='lightgray',linestyle='--')

ax4.plot(t, CO_2_true, color="#CEFA05", label="True", linewidth=3,
    linestyle="-")
ax4.plot(t, CO_2_ukf, color="#4169e1", label="ScaledUKF", linewidth
    =3)
ax4.plot(t, CO_2_gen_ukf, color="#C70039", label = "GenUKF",
```

```
    linewidth=3)
ax4.scatter(t, y[2], c="#86c9db", marker='x', label="Measurement") # aquamarine
ax4.set_xlabel("$Time [h]$")
ax4.set_ylabel("$CO_{2} [L]$")
ax4.grid(color='lightgray',linestyle='--')
ax1.legend(loc="center", bbox_to_anchor=(0.5, 1.25), ncol=4, fontsize=15)
plt.savefig('State_estiamtion_noise_additive.pdf')
plt.show()

#-----------------------------------------------------------------------
# Calculating the errors from the estimation
#-----------------------------------------------------------------------
V_error_ukf        = utils_br.error(V_true, V_ukf)
V_error_gen_ukf    = utils_br.error(V_true, V_gen_ukf)
V_rms_ukf          = utils_br.rms(V_true, V_ukf)
V_rms_gen_ukf      = utils_br.rms(V_true, V_gen_ukf)

X_error_ukf        = utils_br.error(X_true, X_ukf)
X_error_gen_ukf    = utils_br.error(X_true, X_gen_ukf)
X_rms_ukf          = utils_br.rms(X_true, X_ukf)
X_rms_gen_ukf      = utils_br.rms(X_true, X_gen_ukf)

S_error_ukf        = utils_br.error(S_true, S_ukf)
S_error_gen_ukf    = utils_br.error(S_true, S_gen_ukf)
S_rms_ukf          = utils_br.rms(S_true, S_ukf)
S_rms_gen_ukf      = utils_br.rms(S_true, S_gen_ukf)

CO_2_error_ukf     = utils_br.error(CO_2_true, CO_2_ukf)
CO_2_error_gen_ukf = utils_br.error(CO_2_true, CO_2_gen_ukf)
CO_2_rms_ukf       = utils_br.rms(CO_2_true, CO_2_ukf)
CO_2_rms_gen_ukf   = utils_br.rms(CO_2_true, CO_2_gen_ukf)


#-----------------------------------------------------------------------
```

```
# Printing MAE, MSE and RMSE
#-----------------------------------------------------------------------
print("-"*50)
print(f"V_rms_ukf: {V_rms_ukf}")
print(f"V_rms_gen_ukf:{V_rms_gen_ukf}")
print(f"X_rms_ukf: {X_rms_ukf}")
print(f"X_rms_gen_ukf:{X_rms_gen_ukf}")
print(f"S_rms_ukf: {S_rms_ukf}")
print(f"S_rms_gen_ukf:{S_rms_gen_ukf}")
print(f"CO_2_rms_ukf: {CO_2_rms_ukf}")
print(f"CO_2_rms_gen_ukf:{CO_2_rms_gen_ukf}")
print("-"*50)


#-----------------------------------------------------------------------
# Displaying the error
#-----------------------------------------------------------------------
fig, (ax1, ax2, ax3, ax4) = plt.subplots(4, 1, figsize=(16, 8),
    sharex=True)
ax1.plot(t, V_error_ukf, color="#4169e1", label="Error ScaledUKF",
    linewidth=3, linestyle="--")
ax1.plot(t, V_error_gen_ukf, color="#C70039", label="Error GenUKF",
    linewidth=3, linestyle="--")
ax1.grid(color='lightgray',linestyle='--')
ax1.set_ylabel("$V [L]$")

ax2.plot(t, X_error_ukf, color="#4169e1", label="Error ScaledUKF",
    linewidth=3, linestyle="--")
ax2.plot(t, X_error_gen_ukf, color="#C70039", label="Error GenUKF",
    linewidth=3, linestyle="--")
ax2.grid(color='lightgray',linestyle='--')
ax2.set_ylabel("$X [L]$")

ax3.plot(t, S_error_ukf, color="#4169e1", label="Error ScaledUKF",
    linewidth=3, linestyle="--")
ax3.plot(t, S_error_gen_ukf, color="#C70039",label="Error GenUKF",
```

```
    linewidth=3, linestyle="--")
ax3.grid(color='lightgray',linestyle='--')
ax3.set_ylabel("$S [L]$")

ax4.plot(t, CO_2_error_ukf, color="#4169e1", label="Error ScaledUKF"
    , linewidth=3, linestyle="--")
ax4.plot(t, CO_2_error_gen_ukf, color="#C70039", label="Error GenUKF
    ", linewidth=3,linestyle="--" )
ax4.grid(color='lightgray',linestyle='--')
ax4.set_ylabel("$CO_{2} [L]$")
ax1.legend(loc="center", bbox_to_anchor=(0.5, 1.25), ncol=4,
    fontsize=15)
plt.savefig('Error_noise_addititve.pdf')
plt.show()
```

Code 3: Implementation of the main module for the bioprocess system with additive noise.

```
# @author: Abubakar Bampoye
# Importing the required modules
#----------------------------------------------------------------
import scipy.optimize
import matplotlib.pyplot as plt
import numpy as np
import scaled_filter
import utils_bioreactor_parameter_noise as utils_br
plt.rc('text', usetex=True)
plt.rcParams['text.latex.preamble'] = r"\usepackage{amsmath}"
plt.rcParams['text.latex.preamble'] = r"\boldmath"
font = {'family': 'normal', 'weight':'bold', 'size':20}
plt.rc('font', **font)
#----------------------------------------------------------------



#----------------------------------------------------------------
# Defining the parameters (Data)
#----------------------------------------------------------------
mu_max     = 1.94e-1  # Maximum growth rate [h^-1]
```

```
K_s         = 7.00e-3  # Monod growth constant [gL^-1]
k_d         = 6.00e-3  # Death rate constant [h^-1]
Y_XS        = 4.2e-1   # S from X yield [gg^-1]
Y_XCO_2     = 5.43e-1  # CO_2 from X yield [gg^-1]
S_in        = 100      # [g/l]
q_air       = 120      # [NL/h]
theta_mean = np.array([mu_max, K_s, k_d, Y_XS, Y_XCO_2, S_in, q_air
    ])
P_theta     = np.diag((theta_mean*.05)**2)
p_dim       = theta_mean.shape[0]


#-----------------------------------------------------------------
# Defining the sampling time (integration time span)
#-----------------------------------------------------------------
dt           = 1 # [1/min]
t_0          = 0 # [min]
t_f          = 15*60 # [min]
t            = np.arange(t_0, t_f, dt)/60 # [h]
dim_t        = len(t)
dt_sampling = t[1] - t[0]

#-----------------------------------------------------------------
# Defining the initial conditions and necessary information in
# order to estimate the state
#-----------------------------------------------------------------
# The initial conditions from the literature
V0, X0, S0, CO_20   = 1.5, 1.2, 20., 10e-10
# Initial guess of the state
x0                  = np.array([V0, X0, S0, CO_20])
# The dimention of the state
dim_x               = x0.shape[0]
# Allocating memory for the true state
x_true              = np.zeros((dim_x, dim_t))
# Initializing the first element of the true state
x_true[:, 0]        = x0
```

```
# Allocating memory for the posteriori
x_posteriori        = np.zeros((dim_x, dim_t))
# Allocating memory for the posteriori of the generelized
# sigmapoints
x_posteriori2       = np.zeros((dim_x, dim_t))
# Inital covariance of the process
P0                  = np.diag(np.square([.1, .5, 1., 1e-2])) #
# Inital covariance of the posteriori
P_posteriori        = P0
# Inital covariance of the generalized posteriori
P_posteriori2       = P0
# Initial state of the filter
x0_filter           = np.random.multivariate_normal(x0, P0)
x0_filter           = np.array([xi if xi > 0 else 1e-10 for xi in
    x0_filter])
# Initializing the first element of the posteriori state
x_posteriori[:,0]   = x0_filter
# Initializing the first element of the generalzed
# posteriori state
x_posteriori2[:,0]  = x0_filter
# Allocating memory for the measurements
y                   = np.zeros((3, dim_t))
dim_y               = y.shape[0]
#remove first measurement
y[:, 0]             = np.nan

# Creating the process noise covariance matrix
Q                   = np.diag([1e-6]*4)
# Creating the measurement noise covariance matrix
R                   = np.diag([1e-2, 1, 1e-3])
# Integration time span
t_span              = np.array([t[0], t[1]])

# Noise method 1: sigma pints of the parameters and weights
X_theta                 = scaled_filter.sus.sigma_points(theta_mean,
    P_theta)
```

```
W_theta_m, W_theta_c    = scaled_filter.sus.weights(theta_mean)
dim_X_theta, dim_sigmas = np.shape(X_theta)
w_bar_hist              = np.zeros((dim_t, dim_x))
Q_est_hist              = np.zeros((dim_t, dim_x))
eps =1e-8

# Noise method 2
Qp            = [np.diag([1. if p == xi else 0. for p in range(dim_x
    )]) for xi in range(dim_x)]
epsilon_k     = np.zeros((dim_y, dim_t))
K             = np.zeros((dim_x, dim_y, dim_t))
H_hist_2      = np.zeros((dim_y, dim_x, dim_t))
F_hist_2      = np.zeros((dim_x, dim_x, dim_t))
x2_prior      = np.zeros((dim_x, dim_t))
P2_prior      = np.zeros((dim_x, dim_x,dim_t))
y_pred        = np.zeros((dim_y, dim_t))
Py_pred       = np.zeros((dim_y, dim_y,dim_t))
C             = np.zeros((dim_y, dim_y, dim_t))
Pa            = np.zeros((dim_x, dim_x, dim_t)) #posterior
    covariance matrices
Qe            = np.zeros((dim_x, dim_x, dim_t))
Q_est_history = np.zeros((dim_x, dim_x, dim_t))
delta         = 1/50 #tuning parameter
u             = np.array([.0]) #initial control variable


#-----------------------------------------------------------------
# Integrating and estimating the states
#-----------------------------------------------------------------
for k in range(1, dim_t):

    # Creating the noise vectors
    vk                  = np.random.multivariate_normal([0]*3, R)
    wk_theta            = np.random.multivariate_normal(np.zeros(
    p_dim), P_theta, size=1).flatten()
    p_plant             = theta_mean + wk_theta
```

```
    x_true[:, k]        = utils_br.f(t_span, x_true[:, k-1], u,
p_plant)
124 y[:, k]             = utils_br.h(x_true[:, k]) + vk

126

    #1) estimate noise
128 W_bar               = utils_br.w_tilde(t_span, x_posteriori[:,k
-1], u, X_theta, theta_mean)
    wk_bar_est, Qk_est = scaled_filter.process_noise(W_theta_m,
W_theta_c, W_bar)
130

    #save estimates for plotting
132 w_bar_hist[k,:]    = wk_bar_est
    Q_est_hist[k,:]    = np.sqrt(np.diag(Qk_est)) #standard deviation
134 Qk_est             = Qk_est + eps*np.eye(dim_x)

136 # UKF1 prediction step
    f_UKF1 = lambda x: utils_br.f(t_span, x, u, theta_mean) +
wk_bar_est
138

    x_priori, P_priori, F_k = scaled_filter.predict(x_posteriori[:,
k-1], P_posteriori, f_UKF1, Qk_est)
140

    #UKF1 update step
142 x_posteriori[:,k], P_posteriori, sig_y, _, _, _, _ =
scaled_filter.update(x_priori, P_priori, utils_br.h, y[:, k], R)


144

    #UKF2
146 #noise estimation
    Qk_est2 = Q_est_history[:,:,k-1]
148 wk_bar_est2 = 0

150 #UKF2 prediction step
    f_UKF2 = lambda x: utils_br.f(t_span, x, u, theta_mean) +
wk_bar_est2
```

```
152
    x_priori2, P_priori2, F_hist_2[:,:,k-1] = scaled_filter.predict(
    x_posteriori2[:, k-1], P_posteriori2, f_UKF2, Qk_est2)
154
    #UKF2 update step
156 x_posteriori2[:,k], P_posteriori2, sig_y2, y_pred[:,k], H_hist_2
    [:,:,k], Py_pred[:,:,k], K[:,:,k] = scaled_filter.update(
    x_priori2, P_priori2, utils_br.h, y[:,k], R)

158 epsilon_k[:,k]   = y[:,k] - y_pred[:,k]
    x2_prior[:,k]    = x_priori2.copy()
160 P2_prior[:,:,k]  = P_priori2.copy()
    Pa[:,:,k]        = P_posteriori2.copy()
162
    C[:,:,k-1] = (np.outer(epsilon_k[:,k], epsilon_k[:,k-1])
164              + H_hist_2[:,:,k]@F_hist_2[:,:,k-1]@K[:,:,k-1]@np.
    outer(epsilon_k[:,k-1], epsilon_k[:,k-1])
                 - H_hist_2[:,:,k]@F_hist_2[:,:,k-1]@F_hist_2[:,:,k
    -2]@Pa[:,:,k-2]@F_hist_2[:,:,k-2].T@H_hist_2[:,:,k-1].T)
166
    A = []
168 for p in range(len(Qp)):
        Akp = H_hist_2[:,:,k]@F_hist_2[:,:,k-1]@Qp[p]@H_hist_2[:,:,k
    -1].T
170     A.append(np.ravel(Akp, order='f').reshape(-1,1))

172 A = np.hstack(A)

174 vec_C = np.ravel(C[:,:,k-1], order='f')
    sol   = scipy.optimize.lsq_linear(A, vec_C, bounds=(1e-10,np.inf
    ))
176 q     = sol.x
    Qe[:,:,k-1] = sum(qj*Qpj for qj, Qpj in zip(q,Qp))
178
    delta = 1/50
180 Q_est_history[:,:,k] = Q_est_history[:,:,k-1] + delta*(Qe[:,:,k
```

```
    -1] - Q_est_history[:,:,k-1])

    # Creating the control variable
    if x_true[2, k-1] > 2:
        u = np.array([.0])
    else:
        u = np.array([0.5*60])


#-----------------------------------------------------------------
# Displaying the results
#-----------------------------------------------------------------
# Exctracting the states
V_true, X_true, S_true, CO_2_true     = x_true[0], x_true[1], x_true
    [2], x_true[3]
V_UKF1, X_UKF1, S_UKF1, CO_2_UKF1     = x_posteriori[0,:],
    x_posteriori[1,:], x_posteriori[2,:], x_posteriori[3,:]
V_UKF2, X_UKF2, S_UKF2, CO_2_UKF2 = x_posteriori2[0,:],
    x_posteriori2[1,:], x_posteriori2[2,:], x_posteriori2[3,:]
fig, (ax1, ax2, ax3, ax4)             = plt.subplots(dim_x, 1,
    figsize=(16, 8), sharex=True)

ax1.plot(t, V_true ,color="#CEFA05", label="True", linewidth=3,
    linestyle="-")
ax1.plot(t, V_UKF1, color="#4169e1", label="UKF1", linewidth=3)
ax1.plot(t, V_UKF2, color="#C70039", label="UKF2", linewidth=3)
ax1.scatter(t, y[0], c="#86c9db", marker='x', label="Measurement")
ax1.set_ylabel("$V [L]$")
ax1.grid(color='lightgray',linestyle='--')

ax2.plot(t, X_true, color="#CEFA05", label="True", linewidth=3,
    linestyle="-")
ax2.plot(t, X_UKF1, color="#4169e1", label="UKF1", linewidth=3)
ax2.plot(t, X_UKF2, color="#C70039", label="UKF2", linewidth=3)
ax2.scatter(t, y[1], c="#86c9db", marker='x', label="Measurement")
ax2.set_ylabel("$X [L]$")
```

```
ax2.grid(color='lightgray',linestyle='--')

ax3.plot(t, S_true, color="#CEFA05", label="True", linewidth=3,
    linestyle="-")
ax3.plot(t, S_UKF1, color="#4169e1", label="UKF1", linewidth=3)
ax3.plot(t, S_UKF2, color="#C70039", label="UKF2", linewidth=3)
ax3.set_ylabel("$S [L]$")
ax3.grid(color='lightgray',linestyle='--')

ax4.plot(t, CO_2_true, color="#CEFA05", label="True", linewidth=3,
    linestyle="-")
ax4.plot(t, CO_2_UKF1, color="#4169e1", label="UKF1", linewidth=3)
ax4.plot(t, CO_2_UKF2, color="#C70039", label = "GenUKF", linewidth
    =3)
ax4.scatter(t, y[2], c="#86c9db", marker='x', label="Measurement") #
     aquamarine
ax4.set_xlabel("$Time [h]$")
ax4.set_ylabel("$CO_{2} [L]$")
ax4.grid(color='lightgray',linestyle='--')
ax1.legend(loc="center", bbox_to_anchor=(0.5, 1.25), ncol=4,
    fontsize=15)
plt.savefig('State_estiamtion_paramater_noise.pdf')
plt.show()

#-------------------------------------------------------------------
# Displaying the noise matrices UKF1 & UKF2
#-------------------------------------------------------------------
Q_V_UKF1, Q_X_UKF1, Q_S_UKF1, Q_CO_2_UKF1   = Q_est_hist[:,0],
    Q_est_hist[:,1], Q_est_hist[:,2], Q_est_hist[:,3]
Q_V_UKF2, Q_X_UKF2, Q_S_UKF2, Q_CO_2_UKF2   = Q_est_history[0,0,:],
    Q_est_history[1,1,:], Q_est_history[2,2,:], Q_est_history[3,3,:]
fig_q, (ax_q1,ax_q2, ax_q3, ax_q4 )         = plt.subplots(dim_x, 1,
     figsize=(16, 8), sharex=True)

ax_q1.plot(t, Q_V_UKF1 ,color="#4169e1", label="UKF1, $\epsilon = $"
     + f"{eps}", linewidth=3, linestyle="-")
```

```
ax_q1.plot(t, Q_V_UKF2 ,color="#C70039", label="UKF2", linewidth=3,
    linestyle="-")
ax_q1.set_ylabel("$V [L]$")
ax_q1.grid(color='lightgray',linestyle='--')

ax_q2.plot(t, Q_X_UKF1, color="#4169e1", label="UKF1, $\epsilon = $"
    + f"{eps}", linewidth=3,linestyle="-")
ax_q2.plot(t, Q_X_UKF2, color="#C70039", label="UKF2", linewidth=3,
    linestyle="-")
ax_q2.set_ylabel("$X [L]$")
ax_q2.grid(color='lightgray',linestyle='--')

ax_q3.plot(t, Q_S_UKF1, color="#4169e1", label="UKF1, $\epsilon = $"
    + f"{eps}", linewidth=3, linestyle="-")
ax_q3.plot(t, Q_S_UKF2, color="#C70039", label="UKF2", linewidth=3,
    linestyle="-")
ax_q3.set_ylabel("$S [L]$")
ax_q3.grid(color='lightgray',linestyle='--')

ax_q4.plot(t, Q_CO_2_UKF1, color="#4169e1", label="UKF1, $\epsilon =
    $" + f"{eps}", linewidth=3, linestyle="-")
ax_q4.plot(t, Q_CO_2_UKF2, color="#C70039", label="UKF2", linewidth
    =3, linestyle="-")
ax_q4.set_xlabel("$Time [h]$")
ax_q4.set_ylabel("$CO_{2} [L]$")
ax_q4.grid(color='lightgray',linestyle='--')
ax_q1.legend(loc="center", bbox_to_anchor=(0.5, 1.25), ncol=4,
    fontsize=15)
plt.savefig('Noise_Q.pdf')
plt.show()


#-------------------------------------------------------------------
# Calculating the errors from the estimation
#-------------------------------------------------------------------
V_error_UKF1     = utils_br.error(V_true, V_UKF1)
```

```
V_error_UKF2     = utils_br.error(V_true, V_UKF2)
V_RMSE_UKF1      = utils_br.RMSE(V_true, V_UKF1)
V_RMSE_UKF2      = utils_br.RMSE(V_true, V_UKF2)

X_error_UKF1     = utils_br.error(X_true, X_UKF1)
X_error_UKF2     = utils_br.error(X_true, X_UKF2)
X_RMSE_UKF1      = utils_br.RMSE(X_true, X_UKF1)
X_RMSE_UKF2      = utils_br.RMSE(X_true, X_UKF2)

S_error_UKF1     = utils_br.error(S_true, S_UKF1)
S_error_UKF2     = utils_br.error(S_true, S_UKF2)
S_RMSE_UKF1      = utils_br.RMSE(S_true, S_UKF1)
S_RMSE_UKF2      = utils_br.RMSE(S_true, S_UKF2)

CO_2_error_UKF1  = utils_br.error(CO_2_true, CO_2_UKF1)
CO_2_error_UKF2  = utils_br.error(CO_2_true, CO_2_UKF2)
CO_2_RMSE_UKF1   = utils_br.RMSE(CO_2_true, CO_2_UKF1)
CO_2_RMSE_UKF2   = utils_br.RMSE(CO_2_true, CO_2_UKF2)


#-----------------------------------------------------------------
# Printing RMSEE
#-----------------------------------------------------------------
print("-"*50)
print(f"V_RMSE_UKF1: {V_RMSE_UKF1}")
print(f"V_RMSE_UKF2:{V_RMSE_UKF2}")
print(f"X_RMSE_UKF1: {X_RMSE_UKF1}")
print(f"X_RMSE_UKF2:{X_RMSE_UKF2}")
print(f"S_RMSE_UKF1: {S_RMSE_UKF1}")
print(f"S_RMSE_UKF2:{S_RMSE_UKF2}")
print(f"CO_2_RMSE_UKF1: {CO_2_RMSE_UKF1}")
print(f"CO_2_RMSE_UKF2:{CO_2_RMSE_UKF2}")
print("-"*50)


#-----------------------------------------------------------------
```

```
# Displaying the error
#-----------------------------------------------------------------------
fig, (ax1, ax2, ax3, ax4) = plt.subplots(4, 1, figsize=(16, 8),
    sharex=True)
ax1.plot(t, V_error_UKF1, color="#4169e1", label="UKF1", linewidth
    =3, linestyle="--")
ax1.plot(t, V_error_UKF2, color="#C70039", label="UKF2", linewidth
    =3, linestyle="--")
ax1.grid(color='lightgray',linestyle='--')
ax1.set_ylabel("$V [L]$")

ax2.plot(t, X_error_UKF1, color="#4169e1", label="UKF1", linewidth
    =3, linestyle="--")
ax2.plot(t, X_error_UKF2, color="#C70039", label="UKF2", linewidth
    =3, linestyle="--")
ax2.grid(color='lightgray',linestyle='--')
ax2.set_ylabel("$X [L]$")

ax3.plot(t, S_error_UKF1, color="#4169e1", label="UKF1", linewidth
    =3, linestyle="--")
ax3.plot(t, S_error_UKF2, color="#C70039",label="UKF2", linewidth=3,
     linestyle="--")
ax3.grid(color='lightgray',linestyle='--')
ax3.set_ylabel("$S [L]$")

ax4.plot(t, CO_2_error_UKF1, color="#4169e1", label="UKF1",
    linewidth=3, linestyle="--")
ax4.plot(t, CO_2_error_UKF2, color="#C70039", label="UKF2",
    linewidth=3,linestyle="--" )
ax4.grid(color='lightgray',linestyle='--')
ax4.set_ylabel("$CO_{2}[L]$")
ax4.set_xlabel("$Time [h]$")
ax1.legend(loc="center", bbox_to_anchor=(0.5, 1.25), ncol=4,
    fontsize=15)
plt.savefig('Error_paramater_noise.pdf')
plt.show()
```

Code 4: Implementation of the main module for the bioprocess system with nonlinear noise.

```
# @author: Abubakar Bampoye
# Importing requierd modules
#-----------------------------------------------------------------
import numpy as np
import scipy.linalg
import scaled_unscented_sigma_points as sus
#-----------------------------------------------------------------



#-----------------------------------------------------------------
def process_noise(W_theta, W_theta_c, w_tilde):
    """
    This function calculates the process noise.
    -----------------------------------------------------------------

    Parameters:
    -----------------------------------------------------------------
    W_theta : The weighting coefficients for each sigma point of
    the mean.

    w_tilde : The transformation of the sigma points.


    Returns:
    -----------------------------------------------------------------
    A the process noise.
    """
    dim_X_theta, dim_sigmas       = np.shape(w_tilde)
    assert W_theta.shape[0]       == dim_sigmas, f"Dimensions of
   W_theta = {dim_sigmas} is wrong."
    assert w_tilde.shape[1]       == W_theta.shape[0], f"Wrong shape
    in w_tilde {w_tilde.shape[1]} and W_theta = {W_theta.shape[0]}
```

```
    can't performe matrix multiplication."
32      W_bar                           = w_tilde @ W_theta
        assert w_tilde.shape[0]         == W_bar.reshape(-1, 1).shape[0],
    f"Wrong shape in W_bar = {W_bar.reshape(-1, 1).shape[0]} can't
    subtract with w_tilde = {w_tilde.shape[0]} can't perform matrix
    multiplication"
34      B                               = (w_tilde - W_bar.reshape(-1, 1))
        A                               = W_theta_c*B
36      Q_k                             = A@B.T # Evaluating the sample
    covariance of the transformed sigma points
        assert Q_k.shape == (dim_X_theta, dim_X_theta), f"The shape of
    Q_k = {Q_k.shape} is wrong"
38
        return W_bar, Q_k
40


42  def uncented_transformation(Wm, Wc, Y):
        """
44      This function performs the uncented transformation.
        ----------------------------------------------------------------
46


48      Parameters:
        ----------------------------------------------------------------
50      Wm : The weighting coefficients for each sigma point for the
        mean.
52
        Wc : The weighting coefficients for each sigma point for the
54      covariance.

56      Y : The transformation of the sigma points.


58
        Returns:
60      ----------------------------------------------------------------
        A tuple of ndarray consisting of an estimation of the mean and
```

```
    covariance.
    """

    dim_Y, dim_sigmas          = Y.shape
    assert Wm.shape[0]         == dim_sigmas and Wc.shape[0] ==
dim_sigmas, f"Dimensions of Wc = {Wc.shape} and Wm = {Wm.shape}
are wrong."
    assert Y.shape[1]          == Wm.shape[0], f"Wrong shape in Y {Y.
shape[1]} and Wm = {Wm.shape[0]} can't performe matrix
multiplication."
    y_u                        = Y @ Wm
    assert dim_Y               == y_u.reshape(-1, 1).shape[0], f"The
shape of Y = {Y.shape} is wrong."
    y_tilde                    = (Y - y_u.reshape(-1, 1))
    y_tilde_W                  = Wc * y_tilde # Evaluating the sample
    mean of the transformed sigma points
    assert y_tilde_W.shape[1] == y_tilde.T.shape[0], f"Wrong shape
in y_tiled = {y_tilde_W[1]} and {y_tilde.T[0]} can't perform
matrix multiplication."
    P_u                = y_tilde_W @ y_tilde.T # Evaluating the
sample covariance of the transformed sigma points

    return y_u, P_u


def predict(x_posteriori, P_posteriori, f, Q):
    """
    This function calculates the mean and covariance of the uncented
    approximation.
    ----------------------------------------------------------------

    Parameters:
    ----------------------------------------------------------------
    x_posteriori : The current best guess for the mean of x.

    P_posteriori : The current best guess for the covariance of x.
```

```
    f : The nonlinear system equations.

    Q : The process noise covariance matrix.

    Returns:
    ----------------------------------------------------------------
    A tuple of ndarray consisting of an estimation of the mean and
    covariance.
    """
    assert (Q.shape    == P_posteriori.shape), f"Shape of Q = {Q.
shape} is wrong, the shape must be the same as P_posteriori = {
P_posteriori.shape}"
    Wm, Wc             = sus.weights(x_posteriori)
    x_posteriori_out   = sus.sigma_points(x_posteriori, P_posteriori
)
    dim_x, dim_sigmas  = np.shape(x_posteriori_out)  # The dimension
     of the state x
    assert Wm.shape[0] == dim_sigmas and Wc.shape[0] == dim_sigmas,
f"Dimensions of Wc = {Wc.shape} and Wm = {Wm.shape} are wrong."
    assert dim_x       == P_posteriori.shape[0], f"The shape of
x_posteriori_out = {x_posteriori_out.shape} is wrong"
    x_hat              = np.zeros((dim_x, dim_sigmas))

    for i in range(dim_sigmas):
        x_hat[:,i] = f(x_posteriori_out[:,i])

    x_priori, P_priori = uncented_transformation(Wm, Wc, x_hat)
    P_posteriori_prior     = cross_cvariance(x_posteriori_out, x_hat
, x_posteriori, x_priori, Wc)
    F_k                    = P_posteriori_prior.T@scipy.linalg.inv(
P_posteriori)
    P_priori               = P_priori + Q

    return x_priori, P_priori, F_k

```

```
118 def cross_cvariance(x_hat, y_hat, x_priori, y_pred, Wc):
        """
120     This function estimate the cross covariance between x and y
        ----------------------------------------------------------------
122
124     Parameters:
        ----------------------------------------------------------------
126     kappa : A design parameter which is a scaling factor, it can be
        used to reduce higher order errors.
128
        x_hat : The transformation of the sigmapoints through the
130     nonlinear system equations.

132     y_hat : The transformation of the sigmapoints through the
        nonlinear measurment equation.
134
        x_priori : The estimation of the priori state at time k.
136
        y_pred : The predicted measurement at time k.
138
        Wc : The weighting coefficients for each sigma point for the
140     covariance.

142
        Returns:
144     ----------------------------------------------------------------
        Returns an ndarray with the cross covariance between x_k and
146     y_pred.
        """
148
        #Checking if the input is right
150     dim_x, dim_sigmas = x_hat.shape
        dim_y             = y_hat.shape[0]
152     assert y_hat.shape[1] == dim_sigmas, f'''Different number of
```

```
    sigma-points
    for x_hat and y_hat. Expected that the last dimensions are the
    same,
154     but the shapes are dim(x_hat) = {x_hat.shape} and
    dim(y_hat) = {y_hat.shape}'''
156
    #Checking the input size
158     assert x_priori.shape[0] == dim_x, "x-dimensions are wrong"
    assert y_pred.shape[0]   == dim_y, "y-dimensions are wrong"
160     assert Wc.shape[0]       == dim_sigmas, "Dimensions of Wc are
    wrong"
    assert ((x_priori.ndim == 1) and (y_pred.ndim == 1) and (Wc.ndim
    == 1)), "These input arrays should be 1D numpy arrays"
162     assert ((x_hat.ndim == 2) and (y_hat.ndim == 2)), '''These
    arrays should
    be 2D numpy arrays'''
164
    P_xy = np.zeros((dim_x, dim_y))  # Allocating space for the
    cross covariance
166
    for i in range(dim_sigmas):
168         P_xy += Wc[i]*(x_hat[:, i] - x_priori).reshape(-1,1) @ (
    y_hat[:, i] - y_pred).reshape(-1, 1).T

170     assert ((dim_x, dim_y) == P_xy.shape), f"shape of P_xy = {P_xy.
    shape} is wrong"

172     return P_xy


174
def update(x_priori, P_priori, h, y, R):
176     """
    This function performs the measurement update of the state
    estimate.
178     -----------------------------------------------------------------
```

```
    Parameters:
    ----------
    x_priori : The estimation of the priori state.

    P_priori : The estimation of the priori covariance.

    h : The measurement equation.

    y : The predicted measurement.

    R : The measurement noise covariance matrix.


    Returns:
    ----------
    Returns a tuple of ndarray with the updated prediction
    measurement and predicted covariance.
    """

    Wm, Wc              = sus.weights(x_priori)
    x_hat               = sus.sigma_points(x_priori, P_priori)
    dim_x, dim_sigmas   = x_hat.shape
    assert Wm.shape[0] == dim_sigmas and Wc.shape[0] == dim_sigmas,
    f"Dimensions of Wm = {Wm.shape} and Wc = {Wc.shape} are wrong."
    y_hat0              = h(x_hat[:, 0])
    dim_y               = y_hat0.shape[0]
    y_hat               = np.zeros((dim_y, dim_sigmas))
    y_hat[:, 0]         = y_hat0

    for i in range(1, dim_sigmas):
        y_hat[:, i] = h(x_hat[:, i])

    y_pred, P_y_pred      = uncented_transformation(Wm, Wc, y_hat)
    assert P_y_pred.shape == R.shape, f"The shape of P_y_pred = {
    P_y_pred.shape} is wrong"
    P_xy                  = cross_cvariance(x_hat, y_hat, x_priori,
```

```
    y_pred, Wc)
    H_k                     = P_xy.T@scipy.linalg.inv(P_priori)
    P_y_pred2               = H_k@P_priori@H_k.T
    assert np.allclose(P_y_pred2, P_y_pred), f"{P_y_pred2 - P_y_pred
    }"
    P_y_pred                = P_y_pred + R
    K                       = scipy.linalg.solve(P_y_pred.T, P_xy.T,
    assume_a="pos")
    K                       = K.T
    x_posteriori            = x_priori + (K @ (y - y_pred))
    P_posteriori            = P_priori - (K @ P_y_pred @ K.T)

    return x_posteriori, P_posteriori, y_hat, y_pred, H_k, P_y_pred,
    K
```

Code 5: Functions that implements the scaled unscented filter.

```
# @author: Abubakar Bampoye
# Importing requierd modules
#-----------------------------------------------------------------
import numpy as np
import scipy.linalg
import constrained_generalized_sigma_points as cgsp
#-----------------------------------------------------------------


def uncented_transformation(W, Y):
    """
    This function performs the uncented transformation.
    ----------------------------------------------------------------


    Parameters:
    ----------------------------------------------------------------
    W : The weighting coefficients for each sigma point for the
    mean and covariance.
```

```
    Y : The transformation of the sigma points.


    Returns:
    ----------------------------------------------------------------
    A tuple of ndarray consisting of an estimation of the mean and
    covariance.
    """

    dim_Y, dim_sigmas         = Y.shape
    assert W.shape[0]         == dim_sigmas, f"Dimensions of W = {W.
    shape} is wrong."
    assert Y.shape[1]         == W.shape[0], f"Wrong shape in Y {Y.
    shape[1]} and Wm = {W.shape[0]} can't performe matrix
    multiplication."
    y_u                       = Y @ W
    assert dim_Y              == y_u.reshape(-1, 1).shape[0], f"The
    shape of Y = {Y.shape} is wrong"
    y_tilde                   = (Y - y_u.reshape(-1, 1))
    y_tilde_W                 = W * y_tilde # Evaluating the sample
    mean of the transformed sigma points
    assert y_tilde_W.shape[1] == y_tilde.T.shape[0], f"Wrong shape
    in y_tiled = {y_tilde_W} and {y_tilde.T} can't perform matrix
    multiplication."
    P_u                       = y_tilde_W @ y_tilde.T # Evaluating
    the sample covariance of the transformed sigma points

    return y_u, P_u



def predict(x_posteriori, P_posteriori, f, Q):
    """
    This function calculates the mean and covariance of the uncented
    approximation.
    ----------------------------------------------------------------
```

```
49
51      Parameters:
        -----------------------------------------------------------------
53      x_posteriori : The current best guess for the mean of x.
55      P_posteriori : The current best guess for the covariance of x.
57      f : The nonlinear system equations.
59      Q : The process noise covariance matrix.
61
        Returns:
63      -----------------------------------------------------------------
        A tuple of ndarray consisting of an estimation of the mean and
65      covariance.
        """
67
        assert (Q.shape         == P_posteriori.shape), f"Shape of Q = {
    Q.shape} is wrong."
69      x_posteriori_out, W   = cgsp.constrained_generalized_sigmas(
    x_posteriori, P_posteriori)
        dim_x, dim_sigma        = np.shape(x_posteriori_out)  # The
    dimension of the state x
71      assert W.shape[0]       == dim_sigma, f"Dimensions of W = {W.
    shape} is wrong."
        assert dim_x            == P_posteriori.shape[0], f"The shape of
     x_posteriori_out = {x_posteriori_out.shape} is wrong"
73      x_hat                 = np.zeros((dim_x, dim_sigma))
75
        for i in range(dim_sigma):
77          x_hat[:,i] = f(x_posteriori_out[:,i])
79      x_priori, P_k_priori = uncented_transformation(W, x_hat)
```

```
    P_k_priori              = P_k_priori + Q
81
    return x_priori, P_k_priori
83


85  def cross_cvariance(x_hat, y_hat, x_priori, y_pred, W):
    """
87  This function estimates the cross covariance between x and y.
    ----------------------------------------------------------------
89


91  Parameters:
    ----------------------------------------------------------------
93  kappa : A design parameter which is a scaling factor, it can be
    used to reduce higher order errors.
95
    x_hat : The transformation of the sigmapoints through the
97  nonlinear system equations.

99  y_hat : The transformation of the sigmapoints through the
    nonlinear measurment equation.
101
    x_priori : The estimation of the priori state at time k.
103
    y_pred : The predicted measurement at time k.
105
    W : The weighting coefficients for each sigma point for the
107 covariance.


109
    Returns:
111 ----------------------------------------------------------------
    Returns an ndarray with the cross covariance between x_k and
113 y_pred.
    """
115
```

```
    #Checking if the input is right
117 dim_x, dim_sigmas = x_hat.shape
    dim_y = y_hat.shape[0]
119 assert y_hat.shape[1] == dim_sigmas, f'''Different number of
    sigma-points
    for x_hat and y_hat. Expected that the last dimensions are the
    same, but
121 the shapes are dim(x_hat) = {x_hat.shape} and dim(y_hat) =
    {y_hat.shape}.'''
123
    #Checking the input size
125 assert x_priori.shape[0] == dim_x, "x-dimensions are wrong."
    assert y_pred.shape[0]   == dim_y, "y-dimensions are wrong."
127 assert W.shape[0]          == dim_sigmas, "Dimensions of Wc are
    wrong."
    assert ((x_priori.ndim == 1) and (y_pred.ndim == 1) and (W.ndim
    == 1)),'''
129 These input arrays should be 1D numpy arrays.'''
    assert ((x_hat.ndim == 2) and (y_hat.ndim == 2)), '''These
    arrays should
131 be 2D numpy arrays.'''

133 P_xy = np.zeros((dim_x, dim_y))  # Allocating space for the
    cross covariance

135 for i in range(dim_sigmas):
        P_xy += W[i]*(x_hat[:, i] - x_priori).reshape(-1,1) @ (y_hat
    [:, i]
137                                                              -
     y_pred).reshape(-1, 1).T

139 assert ((dim_x, dim_y) == P_xy.shape), f"Shape of P_xy = {P_xy.
    shape.shape} is wrong."

141 return P_xy
```

```
143
def update(x_priori, P_k_priori, h, y, R):
145     """
        This function performs the measurement update of the state
147     estimate.
        ----------------------------------------------------------------
149

151     Parameters:
        ----------------------------------------------------------------
153     x_priori : The estimation of the priori state.

155     P_k_priori : The estimation of the priori covariance.

157     h : The measurement equation.

159     y : The predicted measurement.

161     R : The measurement noise covariance matrix.

163
        Returns:
165     ----------------------------------------------------------------
        Returns an ndarray with with the updated prediction measurement
167     and predicted covariance.
        """
169
        x_hat, W          = cgsp.constrained_generalized_sigmas(x_priori
    , P_k_priori)
171     dim_x, dim_sigmas = x_hat.shape
        assert W.shape[0] == dim_sigmas, f"Dimensions of W = {W.shape}
    is wrong."
173     y_hat0            = h(x_hat[:, 0])
        dim_y             = y_hat0.shape[0]
175     y_hat             = np.zeros((dim_y, dim_sigmas))
        y_hat[:, 0]       = y_hat0
```

```
177
    for i in range(1, dim_sigmas):
179         y_hat[:, i] = h(x_hat[:, i])

181     y_pred, P_y_pred      = uncented_transformation(W, y_hat)
        assert P_y_pred.shape == R.shape, f"The shape of P_k_pred = {
    P_y_pred.shape} is wrong"
183     P_y_pred              = P_y_pred + R
        P_xy                  = cross_cvariance(x_hat, y_hat, x_priori,
    y_pred, W)
185     K                     = scipy.linalg.solve(P_y_pred.T, P_xy.T,
    assume_a="pos")
        K                     = K.T
187     x_posteriori          = x_priori + K @ (y - y_pred)
        P_posteriori          = P_k_priori - (K @ P_y_pred @ K.T)
189
    return x_posteriori, P_posteriori, y_pred
```

Code 6: Functions that implements the generalized unscented filter.

```
# @author: Abubakar Bampoye
2 # Importing the required modules
#-----------------------------------------------------------------
4 import numpy as np
import scipy.integrate
6 #-----------------------------------------------------------------


8

def bioreactor(t, x, u_array, p):
10     """
       This function models the system using Monod-like kinetics for
12     growth on a singular sugar, with linear cell death and
    considering
       dilution when the feeding is added.
14

    ----------------------------------------------------------------
```

```
16
    Parameters:
18
    ----------------------------------------------------------------

    x : The state vector
20  t : Sampling time
    uk : Input signal
22  p : The parameters of the system

24
    Return:
26
    ----------------------------------------------------------------

    A numpy array of the integrated value of the states
28  """

30  # Extracting the state variables
    V    = x[0]
32  X    = x[1]
    S    = x[2]
34  CO_2 = x[3]

36  # Extracting the parameters
    mu_max     = p[0]  # Maximum growth rate [h^-1]
38  K_s        = p[1]  # Monod growth constant [gL^-1]
    k_d        = p[2]  # Death rate constant [h^-1]
40  Y_XS       = p[3]  # S from X yield [gg^-1]
    Y_XCO_2    = p[4]  # CO_2 from X yield [gg^-1]
42  V_r        = p[5]  # [L]
    S_in       = p[6]  # [g/l]
44  q_air      = p[7]  # [NL/h]

46  # The control signal
```

```
    uk = u_array[0]

    # Defining the ode-system
    dVdt    = uk
    dXdt    = - (uk/(V))*X + mu_max*(S/(K_s + S))*X - k_d*X
    dSdt    = (uk/(V)) * (S_in - S) - mu_max * (S/(K_s + S))*(X/(
Y_XS))
    dCO_2dt = (1/(V_r - V))*(mu_max*(S/(K_s + S))*(X/(Y_XCO_2))*V +
(uk - q_air)*CO_2)
    xdot    = np.array([dVdt, dXdt, dSdt, dCO_2dt])
    return xdot


def f(t, x, u, p):
    """
    This function computes the sytem equations.

    ---------------------------------------------------------------------



    Parameters:

    ---------------------------------------------------------------------

    t : sampling time
    x : the state vector
    uk : Input signal
    p : The parameters of the system
    wk : Process noise vectors


    Return:

    ---------------------------------------------------------------------
```

```
    A list of the integrated value of the states
    """
    sol = scipy.integrate.solve_ivp(bioreactor, t, x, args=[u, p])
    return sol.y[:,-1]


def h(x):
    """
    This function computes the measurement equation.

    ----------------------------------------------------------------------



    Parameters:

    ----------------------------------------------------------------------

    x : the state vector.


    Return:

    ----------------------------------------------------------------------

    The calculated measurements.
    """
    H_matreise = np.array([[1, 0, 0, 0],
                           [0, 1, 0, 0],
                           [0, 0, 0, 1]])
    y_k        = H_matreise@x
    return y_k


def error(y_observation, y_prediction):
    """
```

```
    This function calculates the error in a set of predictions.
106
    ----------------------------------------------------------------------



108
    Parameters:
110
    ----------------------------------------------------------------------

    y_observation : The true observation.
112 y_prediction : The predicted.

114
    Return:
116
    ----------------------------------------------------------------------

    A numpy array with the calculated error,
118 """
    return y_observation - y_prediction
120

122 def mae(y_observation, y_prediction):
    """
124 This function measures the average over the test sample of the
    absolute differences between prediction and actual obeservation.
126
    ----------------------------------------------------------------------



128
    Parameters:
130
    ----------------------------------------------------------------------
```

```
    y_observation : The true observation.
    y_prediction : The predicted.

    Return:
    ------------------------------------------------------------------

    The mean absolute errors.
    """
    #Chekcing the shape of the input vectors
    assert np.shape(y_observation[0]) == np.shape(y_prediction[0]),
    "The shape of the input vectors is wrong"
    return np.mean(np.absolute(error(y_observation, y_prediction)))


def mse(y_observation, y_prediction):
    """
    This function measures the average over the squared test sample
    between prediction and actual obeservation.
    ------------------------------------------------------------------



    Parameters:
    ------------------------------------------------------------------

    y_observation : The true observation.
    y_prediction : The predicted.

    Return:
```

```
160
    ----------------------------------------------------------------

    The mean squared errors.
    """
162
    #Chekcing the shape of the input vectors
    assert np.shape(y_observation[0]) == np.shape(y_prediction[0]),
    "The shape of the input vectors is wrong"
166
    return np.mean(np.square(error(y_observation, y_prediction)))
168

170 def rms(y_observation, y_prediction):
    """
172 This function measures root squared average over the squared
    test
    sample between prediction and actual obeservation.
174
    ----------------------------------------------------------------


176
    Parameters:
178
    ----------------------------------------------------------------

    y_observation : The true observation.
180 y_prediction : The predicted.

182
    Return:
184
    ----------------------------------------------------------------

    The root mean squared error.
```

```
186
    TODO: det minste tallet er den beste prediksjonen
188 """

190 #Chekcing the shape of the input vectors
    assert np.shape(y_observation[0]) == np.shape(y_prediction[0]),
    "The shape of the input vectors is wrong"
192
    return np.sqrt(mse(y_observation, y_prediction))
```

Code 7: Utility module with function for the bioreactor system with additive noise.

```
1  # @author: Abubakar Bampoye
   # Importing the required modules
3  #-----------------------------------------------------------------
   import numpy as np
5  import scipy.integrate
   #-----------------------------------------------------------------
7


9  def bioreactor(t, x, u_array, p):
       """
11     This function models the system using Monod-like kinetics for
       growth
       on a singular sugar, with linear cell death and considering
       dilution
13     when the feeding is added.

       -----------------------------------------------------------------

15


17     Parameters:

       -----------------------------------------------------------------

19     x : the state vector.
```

```
    t : sampling time.
    uk : Input signal.
    p : parameters of the system.


    Return:

    ----------------------------------------------------------------------

    A list of the integrated value of the states.
    """

    # Extracting the state variables
    V    = x[0]
    X    = x[1]
    S    = x[2]
    CO_2 = x[3]

    # Extracting the parameters
    mu_max     = p[0]  # Maximum growth rate [h^-1]
    K_s        = p[1]  # Monod growth constant [gL^-1]
    k_d        = p[2]  # Death rate constant [h^-1]
    Y_XS       = p[3]  # S from X yield [gg^-1]
    Y_XCO_2    = p[4]  # CO_2 from X yield [gg^-1]
    S_in       = p[5]  # [g/l]
    q_air      = p[6]  # [NL/h]

    V_r        = 4        # [L] HAK: Volumet til reaktoren er en
    konstant parameter

    # The control signal
    uk = u_array[0]

    # Defining the ode-system
    dVdt    = uk
    dXdt    = - (uk/(V))*X + mu_max*(S/(K_s + S))*X - k_d*X
```

```
    dSdt    = (uk/(V)) * (S_in - S) - mu_max * (S/(K_s + S))*(X/(
    Y_XS))
    dCO_2dt = (1/(V_r - V))*(mu_max*(S/(K_s + S))*(X/(Y_XCO_2))*V +
    (uk - q_air)*CO_2)
    xdot    = np.array([dVdt, dXdt, dSdt, dCO_2dt])
    return xdot


def f(t,x,u,p):
    """
    This function computes the sytem equations.

    ----------------------------------------------------------------



    Parameters:

    ----------------------------------------------------------------

    t : sampling time
    x : the state vector
    uk : Input signal
    p : The parameters of the system
    wk : Process noise vectors


    Return:

    ----------------------------------------------------------------

    A list of the integrated value of the states
    """

    sol = scipy.integrate.solve_ivp(bioreactor, t, x, args=[u, p])
    return sol.y[:,-1]
```

```
81
83  def h(x):
        """
85      This function computes the measurement equation.
        ----------------------------------------------------------------
87
89      Parameters:
        ----------------------------------------------------------------
91      x : the state vector.
93
        Return:
95      ----------------------------------------------------------------
        The calculated measurements.
97      """
99      H_matreise = np.array([[1, 0, 0, 0],
                               [0, 1, 0, 0],
101                            [0, 0, 0, 1]])
        y_k        = H_matreise@x
103     return y_k
105
    def w_tilde(t_span, x_posteriori, uk, X_theta, theta_bar):
107     """
        This function calculates the mean of the noise.
109
        ----------------------------------------------------------------
```

```
111     Parameters:
        ----------------------------------------------------------------
113     t_span : The sampling time.
115     x_posteriori : the state vector.
117     uk : The input signal (manipulated variable)
119     X_theta : The sigma points of the parameters
121     theta_bar : The deterministic parameters for the state update
123     Return:
        ----------------------------------------------------------------
125     The mean of the noise
        """
127
        x_nom                   = f(t_span, x_posteriori, uk, theta_bar)
129     dim_X_theta, dim_sigmas = np.shape(X_theta)
        dim_theta               = np.shape(theta_bar)[0]
131     x_dim                   = np.shape(x_nom)[0]
        W_tilde                 = np.zeros((x_dim, dim_sigmas))
133
        #Checking if the shape is right
135     assert dim_sigmas == (2*dim_X_theta + 1), f"The shape of X_theta
        = {np.shape(X_theta)} is wrong"
        assert dim_theta  == dim_X_theta, f"The shape of theta_bar = {np
        .shape(theta_bar)[0]} is wrong"
137     assert x_dim      == np.shape(x_posteriori)[0], f"The shape of
        x_nom = {np.shape(x_dim)[0]} is wrong"
```

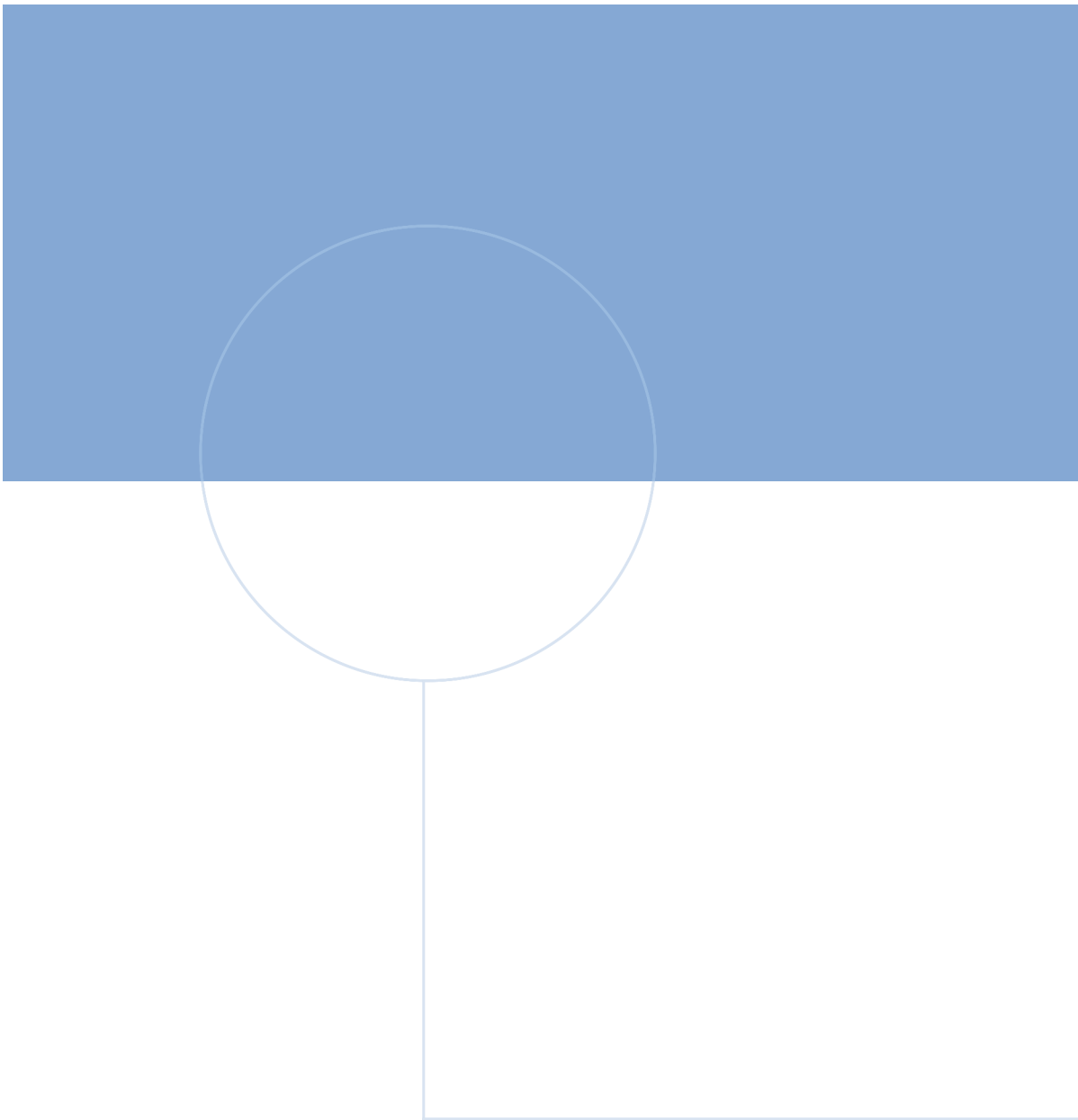
```
139     for i in range(dim_sigmas):
            W_tilde[:, i] = f(t_span, x_posteriori, uk, X_theta[:,i]) -
        x_nom
141
        assert ((x_dim, dim_sigmas) == W_tilde.shape), f"The shape of
        W_tilde = {W_tilde.shape} is wrong"
143
        return W_tilde
145
147 def error(y_observation, y_prediction):
        """
149     This function calculates the error in a set of predictions.
        ----------------------------------------------------------------
151
153     Parameters:
        ----------------------------------------------------------------
155     y_observation : The true observation.
        y_prediction : The predicted.
157
159     Return:
        ----------------------------------------------------------------
161     A numpy array with the calculated error,
        """
163
        #Chekcing the shape of the input vectors
165     assert np.shape(y_observation[0]) == np.shape(y_prediction[0]),
```

```
    "The shape of the input vectors is wrong"

    return y_observation - y_prediction


def mae(y_observation, y_prediction):
    """
    This function measures the average over the test sample of the
    absolute differences between prediction and actual obeservation.

    ----------------------------------------------------------------



    Parameters:

    ----------------------------------------------------------------

    y_observation : The true observation.
    y_prediction : The predicted.

    Return:

    ----------------------------------------------------------------

    The mean absolute errors.
    """

    #Chekcing the shape of the input vectors
    assert np.shape(y_observation[0]) == np.shape(y_prediction[0]),
    "The shape of the input vectors is wrong"

    return np.mean(np.absolute(error(y_observation, y_prediction)))


```

```
def mse(y_observation, y_prediction):
    """
    This function measures the average over the squared test sample
    between prediction and actual obeservation.

    ----------------------------------------------------------------



    Parameters:

    ----------------------------------------------------------------

    y_observation : The true observation.
    y_prediction : The predicted.

    Return:

    ----------------------------------------------------------------

    The mean squared errors.
    """

    #Chekcing the shape of the input vectors
    assert np.shape(y_observation[0]) == np.shape(y_prediction[0]),
    "The shape of the input vectors is wrong"

    return np.mean(np.square(error(y_observation, y_prediction)))


def rms(y_observation, y_prediction):
    """
    This function measures root squared average over the squared
    test
    sample between prediction and actual obeservation.
```

```
    ---------------------------------------------------------------------

223

    Parameters:

    ---------------------------------------------------------------------

    y_observation : The true observation.
    y_prediction : The predicted.

229

    Return:

    ---------------------------------------------------------------------

    The root mean squared error.
    """

235

    #Chekcing the shape of the input vectors
    assert np.shape(y_observation[0]) == np.shape(y_prediction[0]),
"The shape of the input vectors is wrong"

    return np.sqrt(mse(y_observation, y_prediction))
```

Code 8: Utility module with function for the bioreactor system with nonlinear noise.



NTNU

Norwegian University of
Science and Technology