Moving Horizon Estimation for JModelica.org

Tor Larsson

LUND UNIVERSITY

Department of Automatic Control

MSc Thesis
ISRN LUTFD2/TFRT--5982--SE
ISSN 0280-5316

Department of Automatic Control
Lund University
Box 118
SE-221 00 LUND
Sweden

© 2015 by Tor Larsson. All rights reserved.
Printed in Sweden by Tryckeriet i E-huset
Lund 2015

Abstract

In this thesis a Moving Horizon Estimator (MHE) has been implemented for the JModelica.org platform. JModelica.org is an open-source software platform for simulation and optimization of systems described in the modeling language Modelica. MHE is an optimization-based strategy for state estimation where, at each time step, a finite horizon optimization problem is solved to generate an estimate of the current state values. The goal has been to implement an MHE that works with many already existing Modelica models and that has an intuitive user interface. The performance of the implemented MHE is evaluated using both linear and nonlinear systems in a series of simulation examples. The results indicate that the MHE performs well.

Acknowledgements

I want to thank all the people at Modelon AB that helped me during this time of struggle. I want to thank Håkan Runvik for helping me with the model he developed and Elias Palmqvist for helping me get familiarized with JModelica.org. Most of all I want to thank my supervisor Toivo Henningsson, without his continuous support and feedback I doubt much of this thesis would have been possible at all.

From the department of Automatic Control at LTH I want to thank my supervisor Björn Olofsson for his insightful advice regarding Moving Horizon Estimation and report writing. Without him this report would have taken much more time to complete.

Contents

1. Introduction **9**
1.1 State of the Art 10
1.2 Aim of the Thesis and Contributions 11
1.3 Outline 11

2. State Estimation **12**
2.1 The Kalman Filter 12
2.2 Extended Kalman Filter 14
2.3 Full Information Estimation 17
2.4 Moving Horizon Estimation 19
2.4.1 Choice of Arrival Cost Approximation 19

3. Tools **21**
3.1 Modelica 21
3.2 Python 21
3.3 JModelica.org 22

4. Using JModelica.org **23**
4.1 Modelica Models 23
4.2 `FMUModel` 23
4.3 Optimica 26
4.4 `OptimizationProblem` 26

5. Implementation **30**
5.1 Scope of the Implementation 30
5.2 Implementation Structure 32
5.3 `MHEOptions` Class 33
5.4 `MHE` Class 34
5.4.1 Initialization 34
5.4.2 Public Methods 35
5.4.3 Cost Function Approximation 38
5.4.4 Creating the Cost Function 39
5.5 `EKFArrivalCost` Class 40
5.5.1 Initialization 40

5.5.2 Public Methods 41
5.6 Example 43
5.6.1 Model 43
5.6.2 Script 44
6. Results **49**
6.1 Kalman Filter Comparison 49
6.2 Effect of Constraints 51
6.2.1 Constrained LTI 51
6.2.2 Constrained CSTR 53
6.3 Plant Model 53
6.4 Horizon Length 56
6.4.1 CSTR 56
6.4.2 Plant 59
6.5 Parameter Mismatch 61
7. Conclusions and Future Work **65**
Bibliography **67**

1

Introduction

For many physical systems it is impossible to get measurements of all the required states. These systems therefore require the use of state estimators to provide an value for the unmeasurable variables. Measurements taken in an environment under the influence of high noise levels can also benefit from the use of a state estimator. For linear, unconstrained systems with white Gaussian noise disturbances with known distributions, the Kalman filter functions as the optimal estimator. However, in most practical applications the systems being dealt with are not linear and can not directly make use of the Kalman filter. For the sake of being able to use state estimators in real applications, different methods for nonlinear systems have been developed [Soroush, 1998].

A natural extension to the Kalman filter that makes it applicable to nonlinear systems is the extended Kalman filter (EKF), which uses repeated local linearization of the nonlinear system around the current estimate [Gelb, 1974]. Having the system linearized around the current point means that the Kalman filter can then be directly applied. EKF has been used successfully in several different applications and can be considered the standard method for state estimation of nonlinear systems, even though very little can be proven about its stability [Liu, 2013]. The use of linearization means that the nonlinear properties of the system are ignored, which has been shown to result in the algorithm failing in specific cases [Wilson et al., 1998]. EKF also does not have the ability to take into consideration constraints on the system states. This has been shown to, in some cases, result in poor performance of the EKF algorithm [Rao et al., 2003].

Moving Horizon Estimation (MHE) uses an optimization-based approach to perform state estimation. To get estimates of the states, a fixed-horizon optimization problem is solved at each time step. MHE is in many respects similar to Model Predictive Control (MPC) and MHE started gaining interest as MPC rose in popularity. The solution of the optimization problem requires more computational power than that of the methods MHE is trying to replace, mainly EKF. However, the continuing improvement of both hardware and the efficiency of the algorithms used in computing has made it possible to use MHE in more and more applications [Rao et al., 2003].

One important advantage of MHE is that, unlike EKF, linearization is not used, meaning that potentially important nonlinear dynamics are not lost. The main advantage of MHE is that since it uses optimization to perform state estimation it has the ability to explicitly handle constraints on states and noise sources. Thus, physical properties of the system, such as concentration and pressure being non-negative, and known properties of the disturbances can be utilized to make the model adhere to the behavior of the process [Rao and Rawlings, 2002].

Modelica [*Modelica and the Modelica Association*] is a non-proprietary, object-oriented language used for modeling dynamic systems. Models of components or whole systems are described using differential, algebraic and discrete equations. These components can then be reused and connected giving the user a convenient way of expressing complex, large systems. Modelica works in real-time, meaning that MHE can not be directly applied to Modelica models.

JModelica.org [*JModelica.org*] is an open-source, Modelica based platform for performing simulation and optimization of Modelica models. The platform extends the Modelica language with the Optimica extension, which provides the platform with the ability to perform optimization based on Modelica models. JModelica.org is maintained by the Swedish company Modelon AB [*Modelon AB*] in cooperation with academia.

1.1 State of the Art

Recent development in the MHE field seems to consist of two main parts: finding new, effective ways to solve the optimization problem and finding new ways to summarize the information before the finite horizon over which the optimization is performed.

To make the optimization more effective, [Zavala et al., 2008] presents a new way of performing the optimization based on background optimization and nonlinear programming (NLP) sensitivity concepts. This new structure allows a large part of the computations to take place in between samples, allowing for quick estimation when the new measurement data become available.

Several different ways of approximating the data outside of the finite horizon window have been suggested. The classic way of doing it is by using the same covariance matrix update formula found in the EKF [Rao and Rawlings, 2002]. After that, other alternatives have been suggested. In [Qu and Hahn, 2009], a way that uses the unscented Kalman filter (UKF) to approximate the old data is suggested. This approach avoids performing linearization and produces improved results for some systems compared to that of using the EKF covariance update. It is also possible to use particle filters for the estimation of the old data, as suggested in [Ungarala, 2009].

When it comes to available software for performing MHE on nonlinear systems, there is not an overabundance of choices. There is, however, an implementation

written in Octave whose details are described in [Tenny and Rawlings, 2002]. That implementation is a part of a larger toolbox for MPC called MPC tools [*Rawlings Group*].

1.2 Aim of the Thesis and Contributions

The goal of this thesis is to implement a nonlinear MHE for the JModelica.org platform. The implementation is created as a basis for a possible future MHE toolbox, meaning that the implementation is made with the thought of it being easily expandable, if more features are desired later on. The goal is therefore not to implement several ways of approximating the data outside of the optimization window or to implement an abundance of features. The focus is instead on implementing the core functionality of nonlinear MHE.

Another goal is that the implementation should work with as many as possible of the already existing Modelica models, and that the interface to the user should demand as little work as possible from the users side, that is, having few inputs that are on an intuitive form.

Implementing this in the JModelica.org platform means that another open-source software for MHE is made available. Since it is a part of the JModelica.org platform it will be compatible with models written in the popular modeling language Modelica.

1.3 Outline

In Chapters 2, 3 and 4 the background of the thesis is presented. Chapter 2 presents the different methods of state estimation that relates to this thesis. Chapter 3 goes on by introducing the tools used in the thesis. How the most critical components of these tools are used are then explained in Chapter 4. This background is then used to explain the structure of the implementation and the different design choices in Chapter 5. Chapter 6 introduces the simulation examples used to evaluate the performance of the estimator and presents the results of said examples. The report is then rounded off in Chapter 7 by presenting the conclusions that can be drawn and how the MHE platform can be further developed in the future.

2

State Estimation

This chapter provides some background about the field of state estimation relevant to this thesis. The Kalman filter and its nonlinear extension the EKF are presented, as they represent the traditional way of performing state estimation. Two optimization-based estimators, the full information estimator and the MHE, are presented. Different ways of approximating the data outside of the optimization window of the MHE are presented.

2.1 The Kalman Filter

Since 1960 when Kalman wrote his influential article [Kalman, 1960] on a recursive solution to the linear data filtering problem, much research has been done on the Kalman filter. The filter itself is made up of a set of recursive equations and for a linear, unconstrained system this set of equations provides an optimal estimate [Brown, 1983].

Given a linear, time discrete system on the form

$$x_{k+1} = A_k x_k + B_k u_k + G_k w_k \tag{2.1}$$

and observations given by

$$y_k = C_k x_k + v_k \tag{2.2}$$

where x_k is the state vector, y_k the measurement vector, the A_k matrix specifies the dynamics of the linear system, the B_k matrix specifies how control inputs enter the system, the C_k matrix gives how the measurements depends on the states, the G_k matrix describes how the process noise enters the system, w_k the process noise vector and v_k the measurement noise vector. w_k and v_k are assumed to be uncorrelated, zero mean, Gaussian variables with a covariance specified by the matrices Q_k and R_k respectively. It is also assumed that an *a priori* estimate of the state vector $\hat{x}_0^-$ is available. Here the "hat" indicates an estimated variable and the "superminus" notation indicates that it is the *a priori* estimate. The error covariance matrix for the *a priori* estimate at the first time step P_0 is also assumed to be known. Given the *a*

priori state estimate defined as $\hat{x}_k^-$ and the *a posteriori* state estimate defined as $\hat{x}_k$, the *a priori* error and the *a posteriori* error are defined as

$$e_k^- \equiv x_k - \hat{x}_k^-$$

and

$$e_k \equiv x_k - \hat{x}_k$$

and their respective covariance matrices are defined as

$$P_k^- = E[e_k^- e_k^{-T}] \tag{2.3}$$

and

$$P_k = E[e_k e_k^T] \tag{2.4}$$

The *a posteriori* state estimate computed from $\hat{x}^-$ and y is the *measurement update*

$$\hat{x}_k = \hat{x}_k^- + K_k(y_k - C_k\hat{x}_k^-) \tag{2.5}$$

where K_k is called the *Kalman gain* or the *blending factor* [Bishop and Welch, 2001]. One usually refers to the difference $(y_k - C_k\hat{x}_k^-)$ as the measurement *residual* or *innovation* and is a measure of the difference between the expected state values and the measured ones. A small measurement residual therefore corresponds to having measurements which correspond well to the predicted values. The predicted values are generated from the model using the *time update*, i.e.,

$$\hat{x}_{k+1}^- = A_k\hat{x}_k + B_k u_k \tag{2.6}$$

where the process noise is removed since it is assumed to have a mean equal to zero and the best estimation of it is therefore to have it removed.

Using the definition of the *a priori* error we have

$$\begin{aligned} e_{k+1}^- &= x_{k+1} - \hat{x}_{k+1}^- \\ &= A_k x_k + G_k w_k - A_k\hat{x}_k \\ &= A_k e_k + G_k w_k \end{aligned} \tag{2.8}$$

Using this in (2.3), an expression for projecting the *a priori* error covariance matrix forwards in time (time update) can be constructed

$$\begin{aligned} P_{k+1}^- &= E[e_{k+1}^- e_{k+1}^{-T}] \\ &= E[(A_k e_k + G_k w_k)(A_k e_k + G_k w_k)^T] \\ &= A_k P_k A_k^T + G_k Q_k G_k^T \end{aligned} \tag{2.10}$$

The *Kalman gain* defined as

$$K_k = P_k^- C_k^T (C_k P_k^- C_k^T + R_k)^{-1} \tag{2.11}$$

is chosen so that it minimizes the *a posteriori* error covariance matrix. For details see [Bishop and Welch, 2001] and the sources contained therein.

From the expression it is evident that a P_k of small magnitude results in a Kalman gain of small magnitude, which in turn results in the residual being weighted less heavily. This can be seen as trusting the measurement y_k less and less as the magnitude of P_k grows smaller and putting more trust in the predicted value $C_k \hat{x}_k^-$. On the other hand a measurement covariance matrix R_k with small magnitude results in the magnitude of the Kalman Gain being larger and therefore more faith is put in the measurement compared to the prediction.

The update expression for the *a posteriori* error covariance matrix is given by

$$P_k = P_k^- - P_k^- C_k^T (C_k P_k^- C_k^T + R_k)^{-1} C_k P_k^- \tag{2.12}$$

For more detail on the derivation of the expression, see for example [Brown, 1983].

Using these equations, the filter is constructed. The filter operation consists of two major phases, a prediction phase and a correction phase. In the prediction phase, the *a priori* state estimate and the error covariance are projected forwards in time and in the correction phase the measurements are used to improve the state estimate and the error covariance matrix. The estimator is initialized using the known P_0^- and $\hat{x}_0^-$. In Figure 2.1 an overview of the estimator is presented.

The major tuning parameters of the filter are the noise covariance matrices Q_k and R_k. Since they are usually unknown they need to be estimated somehow before using the Kalman filter.

2.2 Extended Kalman Filter

The Kalman filter is the optimal estimator for linear, unconstrained systems. Most systems of interest are, however, not linear. This means that the Kalman filter can not be applied in its unmodified state. One way to make the filter applicable to nonlinear systems is to linearize the system around the current work point, and in doing so it is possible to apply the Kalman filter on the obtained linear system. At the cost of performing a linearization at every time point the Kalman filter can thus be extended to be used on nonlinear systems. The method of linearizing a system and applying the Kalman filter update to perform estimation is commonly referred to as the Extended Kalman filter or EKF.

For a long time the EKF was the defacto standard estimator for nonlinear systems, even though very little can be proven concerning the stability properties of the filter. This being the case the filter has still been successfully used in many different applications [Soroush, 1998].

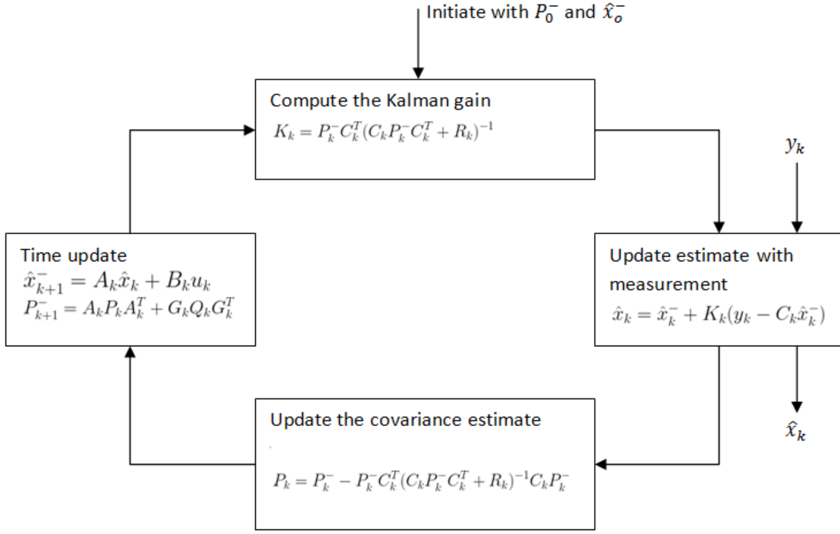


Figure 2.1 *High-level diagram of the Kalman filter loop*

The equations that make up an EKF are mostly the same as in the regular Kalman filter. However, some of the equations need to modified to account for the different system representation and the fact that linearization is performed. The discrete, nonlinear system can be represented as

$$\begin{aligned} x_{k+1} &= f(x_k, u_k, w_k) \\ y_k &= h(x_k, v_k) \end{aligned} \tag{2.13}$$

where $f(\cdot)$ and $h(\cdot)$ can be nonlinear functions. $f(\cdot)$ specifies the dynamics and $h(\cdot)$ describes the measurement equation.

Since the system is now on a different form and contains nonlinear functions the time update is now given by

$$\hat{x}_{k+1}^- = f(\hat{x}_k, u_k, 0) \tag{2.14}$$

where again the best approximation of the process noise w_k is at its mean which is assumed to be equal to zero.

The measurement update is changed to

$$\hat{x}_k = \hat{x}_k^- + K_k(y_k - h(\hat{x}_k^-, 0)) \tag{2.15}$$

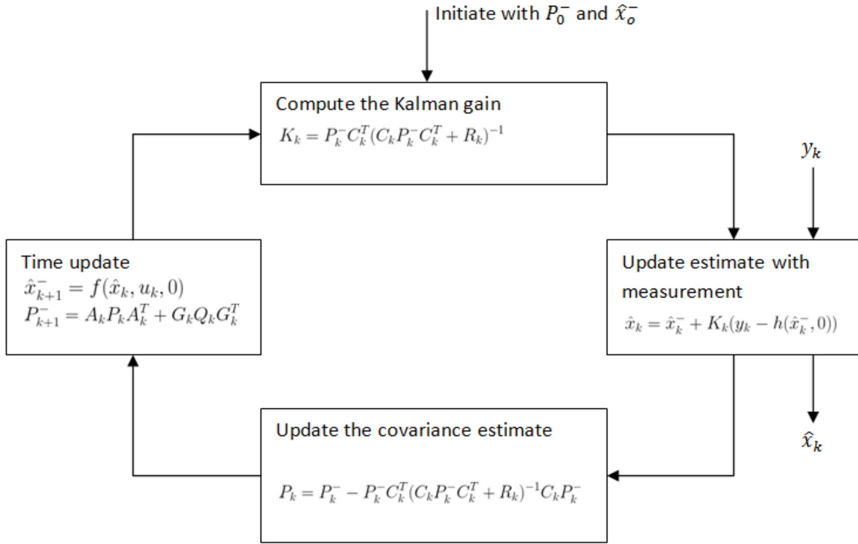


Figure 2.2 *High-level diagram of the extended Kalman filter loop*

to account for the nonlinear measurement dynamics. The measurement noise v_k is set to its mean, which is assumed to be equal to zero, as this is the best approximation.

The other equations remain unchanged but the matrices contained within have been exchanged for the matrices of the linearized system. The linearized system is given by

$$x_{k+1} = f(\hat{x}_k, u_k, 0) + A_k(x_k - \hat{x}_k) + G_k w_k \tag{2.16a}$$
$$y_k = h(\hat{x}_k^-, 0) + C_k(x_k - \hat{x}_k^-) + v_k \tag{2.16b}$$

where the matrices are given by

$$A_k := \left.\frac{\partial f(x_k, u_k, 0)}{\partial x}\right|_{x=\hat{x}_k}, \quad C_k := \left.\frac{\partial h(x, 0)}{\partial x}\right|_{x=\hat{x}_k^-}, \quad G_k := \left.\frac{\partial f(\hat{x}_k, w)}{\partial w}\right|_{w=0} \tag{2.17}$$

The new expressions for the time update and the measurement update of the states results in an algorithm summarized in Figure 2.2. For more details on the extended Kalman filter, see for instance [Bishop and Welch, 2001] and the sources given therein.

2.3 Full Information Estimation

One of the major drawbacks of both the Kalman filter and the EKF is their inability to explicitly handle constraints. Full information estimation (FIE) [Rawlings and Ji, 2012] is an optimization-based approach to perform state estimation. At every time step an optimization problem is solved to generate an estimate which is optimal with respect to a cost function. All the measurement data are used in the optimization, hence the name.

The FIE formulation can be derived using a probability theory interpretation of the state estimation problem. Assuming the same system as in the EKF case, i.e., (2.13) and also assuming that the states and disturbances are subject to the constraints

$$x_k \in \mathbb{X}_k, \ \ w_k \in \mathbb{W}_k, \ \ v_k \in \mathbb{V}_k \tag{2.18}$$

assuming for $k \geq 0$ that $\mathbb{X}_k \subseteq \mathbb{R}^n$, $\mathbb{W}_k \subseteq \mathbb{R}^m$ and $\mathbb{V}_k \subseteq \mathbb{R}^p$ are closed sets with $0 \in \mathbb{W}_k$ $0 \in \mathbb{V}_k$. n, m and p are integers specifying the different dimensions of the system. If the system is also considered a discrete-time Markov process, i.e., the w_k vectors are independent, the goal can be expressed as finding the most probable state trajectory given the measurements received from the system. If expressed using a maximum *a posteriori* Bayesian (MAP) estimate [Sorenson, 1980], it results in the problem formulation

$$\{\hat{x}_{0|T-1}, \hat{x}_{1|T-1}, .., \hat{x}_{T|T-1}\} = \arg \max_{x_0, x_1, \ldots, x_T} p(x_0, x_1, \ldots, x_T | y_0, \ldots, y_{T-1}) \tag{2.19}$$

where $\hat{x}_{k|T-1}$ is the estimate of x_k given measurements up to the time $T-1$.

Using Bayes rule and the Markov property, it is possible to rewrite this expression on the form

$$p(x_0, x_1, \ldots, x_T | y_0, \ldots, y_{T-1}) \propto p_{x_0}(x_0) \prod_{k=0}^{T-1} p_{v_k}(y_k - h(x_k)) p(x_{k+1}|x_k) \tag{2.20}$$

with

$$p(x_{k+1}|x_k) = p_{w_k}(x_{k+1} - f(x_k, u_k)) \tag{2.21}$$

which holds if w_k is assumed to be normally distributed, mutually independent and the system in (2.13) can be rewritten as

$$x_{k+1} = f(x_k, u_k) + G_k w_k \tag{2.22}$$

For more details on the derivation above, see [Rao, 2000].

Combining this with the MAP estimate formulation we get the optimization problem

$$
\begin{aligned}
&\arg\max_{x_0,\ldots,x_T} p(x_0,\ldots,x_T|y_0,\ldots,y_{T-1}) = \\
&\arg\max_{x_0,\ldots,x_T} p_{x_0}(x_0)\prod_{k=0}^{T-1} p_{v_k}(y_k-h(x_k))p_{w_k}(x_{k+1}-f(x_k,u_k)) = \\
&\arg\max_{x_0,\ldots,x_T} \log p_{x_0}(x_0)+\sum_{k=0}^{T-1}\log p_{v_k}(y_k-h(x_k))+\log p_{w_k}(x_{k+1}-f(x_k,u_k)) = \\
&\arg\min_{x_0,\ldots,x_T} \sum_{k=0}^{T-1} \|y_k-h(x_k)\|^2_{R_k^{-1}}+\|x_{k+1}-f(x_k,u_k)\|^2_{Q_k^{-1}}+\|x_0-\bar{x}_0\|^2_{P_o^{-1}} = \\
&\arg\min_{x_0,\ldots,x_T} \sum_{k=0}^{T-1} \|v_k\|^2_{R_k^{-1}}+\|w_k\|^2_{Q_k^{-1}}+\|x_0-\bar{x}_0\|^2_{P_o^{-1}}
\end{aligned}
\tag{2.23}
$$

where $\bar{x}_0$ is the *a priori* estimate of the initial value of the state variables. The sequence $x_0,\ldots,x_T$ can be generated using x_0 and the sequence of process noise from time $k=0$ to $T-1$, $\{w_k\}_{k=0}^{T-1}$. Knowing this, we can rewrite the problem formulation in (2.23) as follows to make the notations more convenient

$$
\min_{x_0,\{w\}_{k=0}^{T-1}} \Phi_T(x_0,\{w_k\}) \tag{2.24}
$$

with

$$
\Phi_T(x_0,\{w_k\}) = \sum_{k=0}^{T-1} \|v_k\|^2_{R_k^{-1}}+\|w_k\|^2_{Q_k^{-1}}+\|x_0-\bar{x}_0\|^2_{P_0^{-1}} \tag{2.25}
$$

The matrices R_k, Q_k and P_0 are the covariance matrices of v_k, w_k and x_0 respectively. They can also be seen as weighting matrices of the optimization problem. If viewed as design variables rather than covariance matrices, the magnitude of the matrices can be seen as the amount of confidence one has in the respective quantities. The magnitude of Q_k can be seen as the level of confidence held in the accuracy of the model, R_k the confidence in the accuracy of the sensors and P_0 the confidence held in the knowledge of the initial state of the system, where a large value represent a low level of confidence [Rao and Rawlings, 2002].

The full information formulation has many desirable properties. Robust global asymptotic stability (RGAS) can be shown for the estimator given certain restrictions on the states and disturbances, for details see [Rawlings and Ji, 2012]. The full information estimator is, however, not suited for any kind of online implementation. This is because the size of the problem will not converge as T increases. Since this is the case, it is better to view the FIE formulation as something that describes the properties you want your optimization based estimator to have rather than to view it as an actual estimation scheme.

2.4 Moving Horizon Estimation

To transform the full information estimator into something that is applicable in a real situation, the problem size has to be reduced. MHE does this by only considering the measurements going N steps back and approximating the older information. Consider

$$\Phi_T(x_0, \{w_k\}_{k=0}^{T-1}) = \sum_{k=T-N}^{T-1} \|v_k\|_{R^{-1}}^2 + \|w_k\|_{Q^{-1}}^2 + \sum_{k=0}^{T-N-1} \|v_k\|_{R^{-1}}^2 + \|w_k\|_{Q^{-1}}^2 + \|x_0 - \bar{x}_0\|_{P_0^{-1}}^2 \quad (2.26)$$

where the only difference from (2.25) is that the sum has been split into two intervals, $0 \leq k \leq T-N-1$ and $T-N \leq k \leq T$. Since the system was modeled as a Markov process the first sum in 2.26 only depends on data from within the window stretching back N steps [Rao and Rawlings, 2002]. It can therefore be rewritten as

$$\min_{x_{T-N}, \{w_k\}_{k=T-N}^{T-1}} \sum_{k=T-N}^{T-1} \|v_k\|_{R^{-1}}^2 + \|w_k\|_{Q^{-1}}^2 + Z_{T-N}(x_{T-N}) \quad (2.27)$$

with the *arrival cost*

$$Z_{T-N}(x_{T-N}) = \min_{x_0, \{w_k\}_{k=0}^{T-N-1}} \Phi_{T-N}(x_0, \{w_k\}_{k=0}^{T-N-1}) \quad (2.28)$$

Solving (2.27) gives the optimal pair $\hat{x}_{T-N}$ and $\{w_{k|T-1}\}_{T-N}^{T-1}$ which can be used to generate the sequence of optimal state estimates $\{\hat{x}_{k|T-1}\}_{k=T-N}^{T}$.

The arrival cost $Z_{T-N}(x_{T-N})$ is used to summarizes the influence of the data outside of the window. To transform the MHE formulation into a problem of bounded size as T grows, the arrival cost must be rewritten as either an algebraic expression or as some kind of approximation $\hat{Z}_{T-N}(x_{T-N})$ that can be written as an algebraic expression. It is possible in the case of an unconstrained, linear system with quadratic objectives to write the arrival cost as such an expression without approximating. This is done by setting the horizon length N to one and using the same update of P that was used in the Kalman filter. In fact, doing this results in the MHE becoming the standard Kalman filter [Rao et al., 2003].

2.4.1 Choice of Arrival Cost Approximation

Different choices for the approximation of the arrival cost have been proposed. Since the choice of approximation influences the stability and performance of the algorithm [Rao, 2000] the choice of arrival cost is an important design parameter when constructing an MHE.

2.4.1.1 Zero Prior Weighting The simplest solution would be to set $\hat{Z}_\tau(x_\tau) = 0$. This means that all prior information is discarded and that the states are estimated using only the information contained in the estimation window. The main drawbacks of this is that the system is required to be observable to be able to guarantee convergence and that the horizon N has to be long enough in order to achieve performance comparable to that of the full information estimator [Rawlings and Mayne, 2009].

2.4.1.2 EKF Approximation In [Rao and Rawlings, 2002] the arrival cost approximation is done by constructing a first order Taylor polynomial of the model around the estimated trajectory. This results in using an equivalent of the covariance update scheme used in EKF. The covariance update can be written as a single equation by inserting (2.10) into (2.12). This results in

$$P_{k+1} = G_k Q_k G_k^T + A_k P_k A_k^T - A_k P_k C_k^T (R_k + C_k P_k C_k^T)^{-1} C_k P_k A_k^T \tag{2.29}$$

where the matrices A_k, C_k and G_k are the result of the same linearization performed for the EKF, i.e., (2.17).

This results in an arrival cost approximation on the form

$$\hat{Z}_T = \|\bar{x}_T - \hat{x}_T\|^2_{P_T^{-1}} \tag{2.30}$$

where $\bar{x}_T$ is the estimate of the states at time T given measurements up until time $T-1$.

Stability for this scheme is, however, not guaranteed. For stability to be granted one must satisfy some technical conditions stated in [Rao et al., 2003]. It is, however, possible to introduce a *forgetting factor* $\beta \in [0,1]$. The forgetting factor makes sure that the prior data are not weighted too heavily by scaling the arrival scaling the arrival cost approximation, i.e., $\beta \hat{Z}_T$. Choosing the value of β to make sure this is achieved involves, at every time step, first performing the optimization using zero prior weighting. The interested reader is directed to [Rao et al., 2003] for details.

3
Tools

This chapter briefly introduces the different tools used in the thesis to familiarize the reader with them. How the most critical tools can be used are further explained in chapter 4.

3.1 Modelica

Modelica is a high level, object-oriented, multi-domain modeling language that enables component-based modeling of dynamic systems. The language is maintained and developed by the non-profit association The Modelica Association [*Modelica and the Modelica Association*], which also develops the free Modelica Standard Library. The library contains a large amount of generic model components that describe the behavior of components used in many different applications.

A Modelica model can contain algorithmic statements similar to those found in other programming languages but the main part of a Modelica model is the equations used to describe the dynamics of the model. The equations do not have to be written on any specific form or order as they are manipulated symbolically by the compiler. This means that the dynamics of a model written in Modelica can be written in a very convenient way that reduces some of the burden from the user and places it instead on the compiler, thus reducing the risk of human error.

3.2 Python

Python is an open-source, high level, general-purpose programming language. The language was introduced in 1991 and is today used in a wide area of different applications. The language includes a large standard library and a lot of different third-party packages are freely available and add new types of functionality. Most relevant to this thesis are the packages that enable scientific computing, namely NumPy [*NumPy*] and SciPy [*SciPy*], and a package called matplotlib [*matplotlib*] that provides plotting capabilities designed to closely resemble those found in the numerical computing software MATLAB [*MATLAB*].

3.3 JModelica.org

JModelica.org is an open-source, Modelica-based platform for performing simulation, analysis and optimization of large, complex dynamic systems. Python is used to create a user-friendly way of interfacing the different parts of JModelica.org. JModelica.org resulted from research done at the Department of Automatic Control at Lund University and is today maintained by the Swedish company Modelon AB in collaboration with academia. The platform was introduced in [Åkesson et al., 2010].

Where most other Modelica compilers mainly aim to perform simulation, the focus of JModelica.org is instead on optimization. The platform incorporates the Modelica extension Optimica [Åkesson, 2008] which allows for optimization problems on Modelica models to be stated in a convenient fashion within the Modelica framework. Using the included state-of-the-art numerical solvers the optimization problems can then be solved. In this thesis, a CasADi-based [*CasADi*] tool chain is used to solve the optimization problem. CasADi is a symbolic framework used to perform automatic differentiation and numeric optimization.

4

Using JModelica.org

Since many different features of the JModelica.org platform are being used in this thesis, it is necessary to explain some of the key features more in depth. In this section, some key terminology and features are therefore explained to make the later chapter on the implementation easier to follow.

4.1 Modelica Models

Modelling a system for use in JModelica.org is done using Modelica. The problem is, at its core, defined by the differential algebraic equations (DAE) that describes the dynamics of the system. Different properties about the variables such as their initial value or maximum and minimum value can also be specified. These properties are referred to as attributes. Listing 1 presents a simple example of a linear time-invariant (LTI) system modeled in Modelica.

4.2 `FMUModel`

Having built a model in Modelica, JModelica.org offers capabilities to compile the model into a *Functional Mock-up Unit* (FMU). An FMU is a compressed file that contains a description of the model. The description used follows a convention called the *Functional Mock-up Interface* (FMI) [*The FMI standard*], detailing how equations, variables, and different properties are represented in the file. The FMI standard is used so that different software can access the same models, enabling easier exchange of models.

The FMU-compiler in JModelica.org can be given a large number of different options. In this thesis only two non-default option for the compiler are utilized. The options are described in Table 4.1. See the JModelica.org user guide [*JModelica.org*] for more details on the compilation process and the other options.

The compiler option `inline_functions` is used to indicate to the compiler how much inlining of functions is to be performed. Inlining functions means to

```
model LTI
    parameter Real A = -1;
    parameter Real B = 1;
    parameter Real C = 1;

    Real x(start = 4, fixed = true);

    Real y;

    input Real u;


    input Real w;
equation
    der(x) = A*x + B*u + w;
    y = C*x;
end LTI;
```

Listing 1: Example of a very simple LTI system defined in Modelica. The `fixed` attribute is a Boolean that determines whether the value of the attribute `start` should be considered an initial value (`fixed = true`) or simply an initial guess for the solver (`fixed = false`).

Table 4.1 *Non-default compiler options used in this thesis*

`state_initial_equations`	Boolean that, if set to `True`, indicates that all initial equations are removed and replaced with an initial equation for each of the states. A parameter for each of the initial values is created. Default: `False`
`inline_functions`	Option indicating if function inlining should be performed. Default: `trivial`

replace a function call with a copy of the function being called. This is done to improve the speed and is done at the cost of space. If too much inlining is performed it can, however, have a negative impact on the speed [Chen et al., 1993]. The available levels for the option are `none`, `trivial` and `all`.

The equations of a model in Modelica are given as DAEs. The FMI standard only support ordinary differential equations (ODE). The DAEs are therefore converted to ODEs during the compilation process.

From a compiled FMU, a `FMUModel` object can be created in JModelica.org. This type of object is used to perform simulation of the model. In this thesis, the `FMUModel` objects are used to produce artificial measurement data which in turn is used when evaluating the performance of the estimator. It would be possible to use data from some real process, but using artificial data means that it is easier to analyze the performance and robustness of the estimator since the source of the data is well defined.

The FMUModel object has a number of different methods for checking the different properties of the model as well as methods for getting and setting values of the parameters. These can be used, for instance, to set the values of the newly created parameters for the initial values of the states, given that state_initial_equations was set to True. The main feature of the FMUModel object is, however, the simulate function which simulates the model. The function is called using a function call on the form

```
res = model.simulate(start_time = 0.,final_time = 10.,
                                        input = 0, options = {})
```

where model is the FMUModel object, start_time and final_time define the time interval of the simulation, input defines the value of the model input signals and options specifies the options. The call above specifies that the inputs are to be set to zero; for details on how more complicated inputs are specified, see the JModelica.org user guide [*JModelica.org*]. The returned res is a result object which contains the trajectories of all the different variables in the model. It is important to note that it is not possible to have the variables adhere to constraints. Since the simulations are used to create artificial measurement data it is important to make sure that the data produced do not violate any imposed constraints later on in the optimization part of the estimation.

The options in the example above are given as an empty dictionary object resulting in the defaults options being used. The non-default options used are given in Table 4.2.

Table 4.2 *Non-default options for the simulation of an FMUModel used in this thesis.*

ncp	Number of communication points used. If chosen as 0 the solver will return the internal steps taken. Default: 0

Changing the number of communication points is done to make sure that the simulation results contain data for the desired time points. Setting the number of communication points to one less than the number of data points wished to be generated ensures that data are generated for the correct time points.

The simulation itself is performed using Assimulo [*Assimulo*]. Assimulo is a simulation tool used to solve ODEs. The tool itself is not a solver, but instead serves as a high-level interface that allows for the use of many different solvers written in different programming languages.

4.3 Optimica

Optimica allows optimization problems to be defined within the Modelica framework. This is done by introducing a number of new features to Modelica. Chief among these is the new kind of class called `optimization`. The class functions similarly to the usual `model` class, in that it is possible to define a model by specifying the DAEs that control the dynamics. But in addition to this cost functions and constraints are introduced, as well as a number of new attributes. Listing 2 presents a simple example of an optimization class.

The cost function in Optimica consists of two different parts: the `objective` and the `objectiveIntegrand` part and together they make up the cost function of the problem. The expression in the `objectiveIntegrand` is integrated over time and is therefore used to account for the cost of time-varying quantities such as the value of states or inputs. The `objective` part is used to house time-invariant quantities such as parameters or the value of a time-varying variable at a certain point in time. Expressed next to the cost function are the two new attributes `startTime` and `finalTime`, which define the time interval over which the optimization is performed.

Constraints for the optimization are expressed in the `constraint` section. The constraints are separated into two different categories, path constraints and point constraints. Point constraints are constraints at single points in time such as

$$\begin{aligned} &s(z(t_0)) \leq 0 \quad \text{or} \\ &s(z(t_0)) = 0 \end{aligned} \tag{4.1}$$

where $s(\cdot)$ is some arbitrary function, z the constrained variables and t_0 the time at which the constraint applies. Path constraints on the other hand are constraints imposed for every point in time such as

$$\begin{aligned} &s(z) \leq 0 \quad \text{or} \\ &s(z) = 0 \end{aligned} \tag{4.2}$$

The possibility to access the value of a parameter at a certain time point is introduced in Optimica. This can be utilized both in point constraints as well as in the `objective` part of the cost function, which can contain the value of variables at a certain time point.

The new `free` attribute is used to specify which variables are free in the optimization, all non-parameter variables are by default set as free variables. To provide the solver with an initial guess for a variable the `initialGuess` attribute is used.

4.4 `OptimizationProblem`

To perform optimization using CasADi within the JModelica.org platform, an optimization problem formulation in Optimica is used to create an `Optimization-`

```
optimization OptimalControlLTI(objectiveIntegrand =
                                (x1 - x1_ref)^2 + (x2 - x2_ref)^2,
                                startTime = 0, finalTime = 10)
extends LTI(u1(free = true, initialGuess = 0),
            u2(free = true, initialGuess = 0));

    parameter Real x1_ref = 3;
    parameter Real x2_ref = 5;

constraint
    u1 >= -1;
    u1 <= 1;
    x1(finalTime) = 3;
end OptimalControlLTI;
```

Listing 2: Example of a simple optimal control problem defined in Optimica. The LTI system in Listing 1 is extended, so that the optimization problem includes all the variables, parameters and DAEs of that system. Some of the inputs from the LTI system are set as free variables and given initial guesses. New parameters defining reference values for the states are added and path and point constraints are added. The cost function punishes deviations of the states from their reference values.

`Problem` object in the Python environment. The `OptimizationProblem` object does not currently have a corresponding file format but is instead transferred directly from the compiler. The process is similar to that of the compiling and loading process of the `FMUModel`, but is performed in one step. The same compiler options as for the FMU are available, i.e., those in Table 4.1. It is also possible for the compiler to take a Modelica model and transfer it to an `OptimizationProblem` object. This option is specified in the function call. In this case, the `OptimizationProblem` is created with the optimization parts empty, i.e., no constraints and a cost function simply set to zero.

When an `OptimizationProblem` object is created, the DAEs are handled symbolically. If two variables are found to be equal, one of the variables will be replaced with the other to reduce the number of variables in the model. The replaced variable will be set as the *alias* of the variable used in the model.

It is possible to manipulate the `OptimizationProblem` object in a number of ways. There are functions for modifying the model by, for instance, adding new parameters, variables, DAEs, and initial equations. The various attributes for the variables and parameters can be changed. The formulation for the optimization can also be altered. It is possible to change the cost function, redefine the constraints and set the time interval for the optimization. Also the attributes associated with the optimization, free variables, and initial guesses can be changed. Hence, the `OptimizationProblem` object is very flexible to work with. It is entirely possible to start with an empty problem and build up the desired problem formulation using the interface to the `OptimizationProblem` object.

Optimization is performed using the `optimize` function. When called, CasADi uses automatic differentiation to symbolically calculate the first and second order derivatives of the DAEs and then a solver, either IPOPT [Wächter and Biegler, 2006] or WORHP, is used to solve the resulting nonlinear programming (NLP) problem.

Direct collocation is used to transfer the problem from an infinite dimensional one to a NLP. The time horizon, the time interval of the optimization, is first separated into elements. Each variable is then approximated as a polynomial within each element. This polynomial is referred to as the *collocation polynomial*, and the order of the polynomial is one below the number of points used as interpolation points. These points are called *collocation points*. It is possible to place the collocation points in a number of ways. The default for the `OptimizationProblem` is called *Radau collocation*, where one point is placed on the boundary of the element to ensure that stability property of the continuous system is kept for the discrete system. The other collocation points are placed for maximum accuracy. The number of collocation points chosen is equivalent to the stage order the of implicit Runge-Kutta method used to perform the discretization. For instance if set to one, implicit Euler is used as the discretization method. For more details see the JModelica.org user guide [*JModelica.org*].

The function call is on the form

```
res = op.optimize(options={})
```

where `op` is the `OptimizationProblem` object and `res` the options object. When the results are given as an empty dictionary, as in the example above, the default options are used. In Table 4.3 the non-default options used in this thesis are presented.

Table 4.3 *Non-default options for the optimization of an* `OptimizationProblem` *object used in this thesis.*

`n_e`	The number of finite elements. Default: 50
`n_cp`	The number of collocation points in each element. Default: 3
`blocking_factors`	Used to enforce piece-wise constant inputs. Values can only change at element boundaries. Specified as a list of over how many elements each input is constant. Default: None
`external_data`	Used to specify data used to eliminate, constrain or penalize certain time varying variables. Default: None

The blocking factors can be specified by, for instance, a list on the form

```
blocking_factors = [2, 2, 1]
```

where each integer specifies the number of elements the corresponding input signal is constant over. The example list above would result in an input signal on the form

$$u = (u_1, u_1, u_2, u_2, u_3)$$

It is important to note that the sum of the items in the list used to specify the blocking factors must equal the number of elements `n_e`. If no blocking factors are specified, then the collocation polynomials are used to describe the input signal wave form.

In this thesis, blocking factors are only used to make the inputs discrete, since in many real applications it is much more natural to assume that the inputs are discrete rather than continuous. It is not possible to use discrete input signals in the simulations performed using an `FMUModel`. This means that if measurement data created from an `FMUModel` object is used for the MHE, there will be a mismatch between the two. Simulation using an `OptimizationProblem` object is not a supported feature, but it is possible to optimize using no free variables and a zero cost function. This results in a "simulation" where it is possible to specify blocking factors and thus having discrete inputs in the simulation as well. Using this method also allows the measurement data to be generated using the same discretization that is later used in the estimator.

The `external_data` option can be used to eliminate input signals from the optimization, that is, to provide values for them so that they are not considered as optimization variables. Thus the inputs used for injecting process noise into the system are not eliminated since they should be free to be estimated.

5

Implementation

This section explains the structure and functionality of the implemented Moving Horizon Estimator. First the types of systems considered and the limitations put on said systems are explained. Then the general structure of the implementation in terms of the structure of classes and modules is explained. Each class is then in turn explained in terms of how they are initialized and their public methods. This chapter then ends by illustrating how the implementation can be utilized using an example.

5.1 Scope of the Implementation

The type of system considered in this implementation can be represented on the form

$$F(\dot{x}, x, y, t) = 0$$

where $F(\cdot)$ is a nonlinear function, x the state vector, y the algebraic variables and t the time variable. This DAE-formulation is natural since this is the form on which equations are specified in Modelica. Using index reduction the system can be converted to the form used in Chapter 2, that is

$$\begin{aligned} \dot{x} &= f(x, u, w) \\ y &= h(x, v) \end{aligned}$$

where $f(\cdot)$ and $h(\cdot)$ are nonlinear functions, x the state vector, y the measurement vector, u the input vector and w, v are the process noise and measurement noise vectors. The systems considered can also have constraints on the form

$$\begin{aligned} s(x, w) &\leq 0 \\ s(x, w) &= 0 \end{aligned}$$

where $s(\cdot)$ is some arbitrary function. This formulation is in continuous time, while an MHE works in discrete time and with measurements available at discrete time points. The continuous time formulation of the system is necessary since Modelica works in continuous time, and therefore all systems described as Modelica models are continuous time systems. The optimization tool chain in JModelica.org discretizes the system using direct collocation, but neither the generated functions that represent the difference equations nor the quadrature scheme used by the discretization are currently exposed. Not having access to the quadrature scheme creates problems when approximating the cost function unless a specific discretiztion is chosen. The decision to use backward Euler was made since it makes the cost function approximation manageable to do by hand. In Sections 5.4.3 and 5.4.4 how the cost function is approximated is further explained.

After being discretized using backward Euler, the system can be expressed as

$$\begin{aligned} x_{k+1} &= f_d(x_k, u_{k+1}, w_{k+1}) \\ y_k &= h_d(x_k, v_k) \end{aligned}$$

where $f_d(\cdot)$ and $h_d(\cdot)$ are nonlinear functions of the discretized system, x_k the state vector, y_k the measurement vector, u_k the input vector, w_k and v_k are the process noise and measurement noise vectors respectively and k is the time index. The constraints are given by

$$\begin{aligned} s(x_k, w_k) &\leq 0 \\ s(x_k, w_k) &= 0 \end{aligned}$$

To discretize the measurement noise covariance matrix, it is simply divided by the sample time h, that is $R_d = R_c/h$, where the d subscript indicates discrete and the c subscript indicates continuous time. For the process noise covariance matrix it is more complicated. If the process noise is simply added to the process and is not affected by the dynamics of the system it can be discretized as $Q_d = hQ_c$. If the noise is affected by the system dynamics, the distribution of the noise can be warped, but if the sampling time is small compared to that of the system's *time constants*, the speed of the system dynamics, the same discretization is a good approximation. The implementation therefore uses this discretization for the process noise covariance matrix.

The implementation uses measurements up to the current sample to generate the state estimate for the next sample. This makes the implementation a one-step ahead estimator, since it is possible to perform the estimation for the next sample when the measurements of the current sample becomes available. Having this structure means that large systems, where the computational time is not negligible compared to the sample time, can be considered as the calculations can be done between samples.

As stated in Section 4.3, there exist two different types of constraints in Optimica, path constraints and point constraints. In this implementation, only path con-

straints will be allowed. This decision was made since there might be ambiguities as to which time points the user wishes the point constraints to be active. Constraints are added by extending a model using the `optimization` class and adding constraints to the `constraint` section. Constraints can also be manually added to a created `OptimizationProblem` object.

The implementation limits the types of allowed Modelica models to the ones where there exists at least one input for the process noise, e.g., an input on the form `input Real w`, or that it is possible to consider at least one of the inputs of the system as a superposition of the intended signal and process noise, that is, inputs on the form

$$u = u_0 + w \tag{5.1}$$

where u_0 is the intended input signal, w the process noise and u is the input that enters the system. For inputs of this type the distribution of the process noise can be warped by the system, meaning that the discretization of the covariance matrix can be less accurate.

5.2 Implementation Structure

To avoid making the structure of the implementation overly complicated, an effort was made to keep the number of classes and modules relatively small, while at the same time trying to keep some things separate to make maintaining the MHE platform manageable. The main part consists of a single class, the `MHE` class. It is only with this class and the corresponding options class `MHEOptions` that the user interacts. Contained within the `MHE` class, there is an object that performs linearization and calculates the error covariance matrix associated with the EKF covariance update used to approximate the arrival cost. This object belongs to a separate class `EKFArrivalCost`. The decision to keep the classes separated was made so that the method for performing the arrival cost approximation can be changed if other methods are implemented.

There are several methods in the `MHE` class that can be directly utilized by the user, but the most important is the `step` method that performs the estimation for one time step. The `step` method and the methods exposed to the user will be explained in detail in Sections 5.4.2 and 5.5.

The MHE package also includes two modules, `check_mhe_inputs` and `mhe_initial_values`. In `check_mhe_inputs`, functionality for checking if the inputs are correct, and raising exceptions for faulty inputs is collected. It also replaces any aliases the user may have given in the input. It can be difficult for the user to keep track of which variables that will be treated as aliases, especially when working with a large model, and therefore the module handles this. The user does not come in contact with this module, it is used internally in other parts of the implementation.

The mhe_initial_values module is an optional module and is not necessary for the MHE class. The module contains a method for generating the initial values of the state derivatives and the algebraic variables of the system. These are needed for the first linearization performed by the EKFArrivalCost object. The choice of having the module separate from the class was made so that the user has a choice regarding how the initial values are generated. If, for example, artificial measurement data are used it is possible to extract the initial values from the simulation, and thus having no need for the module at all. The module was created to provide the user with a consistent method of generating the initial values of the state derivatives and the algebraic variables, should no other convenient method exist.

5.3 MHEOptions Class

The MHEOptions class was created to gather objects used in the MHE class and in other parts of the MHE platform, such as the generation of initial values for the derivatives and algebraic variables. The usage of options objects is common in JModelica.org. The options objects for optimization, simulation and compilation all extend from the same base class as MHEOptions does. The base class, in turn, is an extension of the dictionary class. The objects contained in the MHEOptions object are explained in Table 5.1.

Table 5.1 *Options of the MHEOptions object.*

input_names	A list of all the inputs used as control signals.
process_noise_cov	A list that specifies the covariance of the process noise in continuous time that affects the system.
measurement_cov	A list that specifies the covariance of the measurement noise in continuous time that affects the system.
P_0_cov	A list that specifies the covariance of the initial guess of the state vector. That is it specifies the error covariance matrix at time step 0.
IPOPT_options	A dictionary that specifies the options for the NLP-solver IPOPT.

All the covariance lists in the MHEOptions object contain tuples, where the first item of the tuple specifies the names of the variables having a covariance matrix specified by the second item of the tuple. The first item can therefore be a single string, if the variable is uncorrelated to all other variables, or a list of names if there exists covariance between variables. There can of course be many such lists. The second item can therefore vary in size from a float, in the case of one variable, to a

Numpy array of suitable size for several variables. The goal of this kind of structure is to makes it possible to express sparse matrices, which is often the case for covariance matrices, easy. At the same time, this structure allows the user to easily modify the structure to add covariance between variables. It also removes the need for ordering the variables, which would have to be done if the covariance matrices were expressed as matrix-like objects. With the lists, it is clear which intensity belongs to which variable, reducing the risk of implementation errors.

The `input_names` option is there so that the user can define which of the model's input variables that will be used as control signals for the system. If an input variable is found both in `input_names` and `process_noise_cov`, the input signal is treated as an input with noise. If an input variable is only found in `process_noise_cov`, the input variable is regarded as a process noise input. If found in neither of the lists, an input is considered to be a zero input, i.e., it is zero at all points in time.

The `IPOPT_options` specify the options used by IPOPT to solve the NLP. See the IPOPT documentation [*IPOPT*] for more details.

5.4 `MHE` Class

This section explains how the `MHE` class is initialized and what the public methods do. It is also explained how the cost function is approximated and how the approximation is created.

5.4.1 Initialization

An `MHE` object can be created with a call on the form

```
MHE_object = MHE(op, sample_time, horizon,
                 x_0_guess, dx_0, c_0, MHE_options)
```

where `op` is an `OptimizationProblem` object, `sample_time` the sample time as a float, `horizon` the length of the horizon in samples, `x_0_guess` the guess of the initial value of the state vector, `dx_0` the initial values of the state derivatives, `c_0` the initial values of the algebraic variables and `MHE_options` a `MHEOptions` object.

The inputs `dx_0` and `c_0` are needed for the first linearization step. After the first step, the values of the derivatives and the algebraic variables are provided by the optimization result object.

When an `MHE` object is first created, the following steps are done

1. The user-provided inputs to the object are checked for errors and inconsistencies with the model. Any alias variables are replaced.
2. The `EKFArrivalCost` object is created.

3. The cost function and the required new parameters and variables are added to the `OptimizationProblem` object.

5.4.2 Public Methods

This section contains descriptions of the public methods of the `MHE` object, i.e., the methods that are exposed to the user.

5.4.2.1 `step`

```
step(self, u, y):
"""
Estimates the state vector at the next sample using the
input and measurement vector at the current sample and
returns it.

Parameters::
    u --
        The control signal for the current sample, given as
        a list of tuples on the form (name, value) where 'name'
        is the name of the control signal and 'value' its
        value.

    y --
        The measurement for the current sample, given as a
        list of tuples on the form (name, value) where 'name'
        is the name of the measured variable and 'value' its
        value.

Returns::
    x_est_dict --
        A dictionary with the state names defined by the user
        as keys and the state estimates at the next sample as
        values.
"""
```

Figure 5.1 presents an overview of the `step` method. The method starts out by making sure that the dictionaries provided by the user contain the proper variable names using functionality from the `check_mhe_inputs` module. If the next sample is strictly larger than the horizon, i.e., the data point at time zero is outside of the window, the `EKFArrivalCost` object is called to calculate the new value of the error covariance matrix. This is done by calling the `get_next_P` method with the point of the sample just outside of the window. The public methods of the `EKFArrivalCost` class are presented in Section 5.5.2.

After the error covariance matrix has been updated old data is removed and the different options for the optimization are defined. The number of elements, `n_e`, is set so that there is exactly one element between every pair of neighbouring sample points. To make the discritization backward Euler, the number of collocation points,

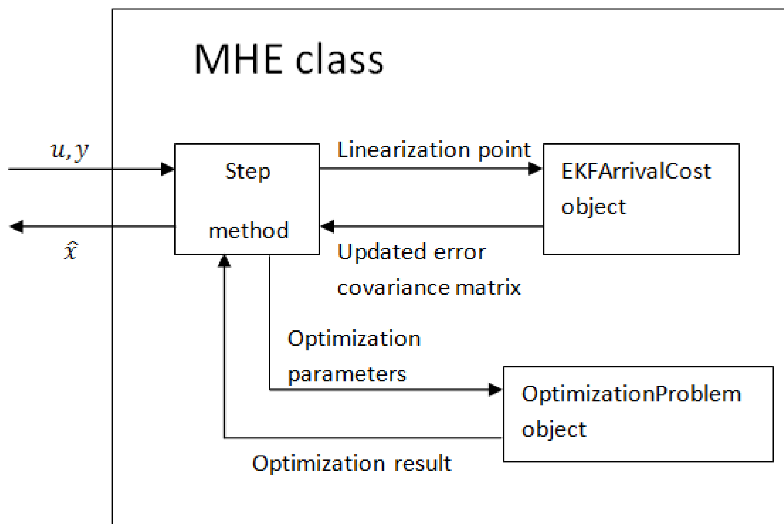


Figure 5.1 *Overview of the* `step` *method. The user provides the method the input and measurement array for the current time step and the state estimate for the next time step is returned.*

`n_cp`, is set to one. To create piece-wise constant inputs the `blocking_factors` options is used, with a list containing ones summing up to the number of elements.

The `external_data` option is used to provide the values of the control signals and the measurements to the optimization. To make the optimization result object contain a value for the state vector at the next sample, the optimization time interval must stretch to the time for that sample. There must therefore be values for the inputs and measurements up to that time. These are not available. The measurements are handled by adding a new input variable that is used to mask the value of the measurement in the last sample. This is further explained in Section 5.4.4. It is not possible to simply mask the control signal since the use of an explicit discretization means that the current state values are generated using the control signal at the current sample. Thus, some kind of approximation needs to be made. In the implementation the choice was made to use the value of the current sample. For a signal that does not vary quickly or drastically, this is the best and most natural assumption, and without more information about the controller structure there is no way to make a more informed guess.

5.4.2.2 `set_process_noise_covariance_matrix`

```
set_process_noise_covariance_matrix(self, process_noise_cov):
"""
Sets the process noise covariance matrix according to a
new covariance list describing the process noise covariance
matrix in continuous time.

Also makes sure that the process noise covariance matrix is
changed in the arrival cost object.

Parameters::
    process_noise_cov --
        A list describing the covariance structure of the
        process noise. Items are on the form
        (names, covariance_matrix).
"""
```

Calling this function will change the value of the process noise covariance matrix. A call is then made to a corresponding function in the `EKFArrivalCost` object that changes the representation of the matrix. This method was implemented so that the user can change the process noise covariance matrix should the characteristics of the noise entering the system change.

5.4.2.3 *set_measurement_noise_covariance_matrix*

```
set_measurement_noise_covariance_matrix(self, measurement_cov):
"""
Sets the measurement covariance matrix according to a new
covariance list describing the measurement noise covariance
matrix in continuous time.

Also makes sure that the measurement covariance matrix is
changed in the arrival cost object.

Parameters::
    measurement_cov --
        A list describing the covariance structure of the
        measurement noise. Items are on the form
        (names, covariance_matrix).
"""
```

This method does the same thing as the `set_process_noise_covariance_-matrix` method, but for the covariance matrix of the measurement noise.

5.4.2.4 *set_beta*

```
set_beta(self, value):
"""
Sets the value of the beta parameter that scales the arrival
cost part of the cost function. Defined for values between
0 and 1.
```

```
Parameters::
value --
    New value of the beta parameter. Between 0 and 1.
"""
```

This method gives the user a way to set the beta parameter discussed in Section 2.4.1.2. No method for calculating the value of the parameter was implemented, as the condition for guaranteeing stability was considered overly restrictive and would involve solving an additional optimization problem. The method was implemented to give the advanced user the ability to set the forgetting factor should it be required for some specific system.

5.4.2.5 `set_dirty`

```
set_dirty(self):
"""
Sets the dirty flag to True. This indicates that the
functions used in the linearization need to be recalculated.
This needs to be done if one parameter or more in the DAEs
have been changed.
"""
```

If the user changes one of the parameters in the `OptimizationProblem` object that affect the DAEs after the initialization this function needs to be called. When called it changes a boolean flag to `True`. When this flag is changed the next time the `step` method is called it, in turn, calls the `recalculate_jacobian_functions` method, Section 5.5.2.2, in the `EKFArrivalCost` object. That method recalculates the functions used to perform the linearization.

5.4.3 Cost Function Approximation

The sum in (2.27) depends on time-varying variables and is therefore approximated as an integral in the `objectiveIntegrand` part of the cost function in the Optimica formulation. This results in the sum being approximated by

$$J_1 = \int_{t_{k-N}}^{t_k} \|v\|^2_{R^{-1}} + \|w\|^2_{Q^{-1}} \, \mathrm{d}t.$$

where t_k is the time at the *k:th* time index. When evaluating the integral, the values at each collocation point in each element are added. If the number of collocation points is set to one, w and v are approximated as piece-wise constant functions. If, in addition to this, the number of elements is set to the number of time points in the interval minus one, so that the end of each element matches a the time point, the evaluation of the integral is equivalent to the sum

$$J_2 = h \sum_{k=T-N+1}^{T-1} \|v_k\|^2_{R^{-1}} + \|w_k\|^2_{Q^{-1}}$$

where h is the sample time.

Since using implicit Euler means that the value at the end point of the collocation element is used to approximate the function value, the function value at the first time point is not included. This is compensated for by adding the first time point to the `objective` part of the cost function instead.

The usage of the implicit Euler discretization has its drawbacks. Since the order of the method is low there will be a relatively large approximation error compared to higher-order methods. However, using implicit Euler is necessary to avoid making the approximation too difficult.

5.4.4 Creating the Cost Function

To create the cost function a number of new parameters, inputs and variables need to be added to the `OptimizationProblem` object. To avoid confusion between added variables and variables belonging to the model, all added variables are given the prefix "_MHE_". This naming convention is also used to identify and look up the variables in a number of different internal methods, meaning that it is important that the model contains no variables starting with the prefix.

For every input that is defined as a superposition of noise and a control signal, a new input variable is created for the original signal u_0 and a new variable is created for the noise. A new equation is added to specify the superposition $u = u_0 + w$ and the new input is used to feed the control signal to the system and the old input signal is set as a free optimization variable. For the inputs used exclusively for process noise, the input is set as a free variable in the optimization.

To specify the measurement noise structure, a new variable for the measurement noise and a new input for the measurements is created for every measured variable. The structure is specified by creating an equation on the form $v = y_{meas} - y$, where y is the measured variable, v the measurement noise and y_{meas} the input signal used to supply the measurements.

Parameters for all the necessary elements in the covariance matrices are added. Here the sparseness of the matrices is utilized so that parameters are only created for the elements that were specified in the covariance list. For the error covariance matrix, however, parameters are created for all the elements since the matrix will change and it can not be guaranteed that the sparsity of the matrix will remain. Parameters are also created for the initial guess of the state vector and the sample time.

When the `set_measurement_noise_covariance_matrix` or the `set_-process_noise_covariance_matrix` methods are called the parameters that were created during the initialization phase are given the value of the new inverse matrix. For elements of the matrix that were not described by the covariance list used during the initialization, new parameters are created and given their value. The cost function is then generated once again and replaces the old one. Using this strategy, parameters are introduced the first time they are needed. There is no way

to remove parameters from the `OptimizationProblem` object, meaning that there can be unused parameters in the `OptimizationProblem` object, but since they are not included in the cost function they do not contribute to the problem size.

To make the optimization result object contain the value of the state vector at the next sample, the optimization must run until the time of that sample. This means that the `objectiveIntegrand` will include values for the measurements at the next sample. Since the measurements at the next sample are not available, their contribution to the cost function must be removed. This is done by adding a new input variable that is used to mask the contribution of the last measurement. Since backward Euler is used and there is only one collocation point used per element the signal can simply be chosen as ones for every time point except for the last one where it is zero. A parameter corresponding to the beta parameter discussed in Section 2.4.1.2 is also added and its value is set to one.

After creating all the necessary variables, inputs and parameters the cost function is created. For all the process noise and measurement noise variables, corresponding timed variables, i.e., the variable values for a certain point in time, are created for the start time of the optimization interval, and added to the `objective` part. The mask signal is multiplied with the $v^T R^{-1} v$ part of the `objectiveIntegrand` and thus removes the influence of measurement at the last sample. The arrival cost part of the cost function is multiplied with the beta parameter so that it is possible to scale it in order to guarantee stability.

5.5 `EKFArrivalCost` Class

This section explains how an `EKFArrivalCost` object is initialized and explains the public methods of the class.

5.5.1 Initialization

When the `EKFArrivalCost` object is first created, the DAEs are extracted from the `OptimizationProblem` object and are made into functions that can be evaluated. The functions created are the derivatives of the DAEs with respect to the state derivatives, the states, the input signals and the algebraic variables of the system. An `EKFArrivalCost` is created with a call of the form

```
EKF_obj = EKFArrivalCost(op, sample_time, state_names,
                         alg_var_names, process_noise_names,
                         undefined_input_names,
                         measured_var_names, MHE_opts)
```

where op is the `OptimizationProblem` object, `sample_time` the sample time, `state_names` a list of the state names, `alg_var_names` the names of all the al-

gebraic variables, process_noise_names a list of all the variables affected by process noise, undefined_input_names a list of all the inputs not defined by the user and therefore considered as zero valued inputs, measured_var_names a list of the names of the measured variables and MHE_opts an MHEOptions object.

The EKFArrivalCost object is used to update the error covariance matrix and therefore does a lot of matrix-based calculations. Since this is the case, it is important that the order of the variables is well defined. This is the reason why the inputs to the object largely consist of lists of names, since the variables in the MHE object need to have the same order. None of this is exposed to the user since the EKFArrivalCost object is only used, internally, by the MHE class.

5.5.2 Public Methods

This section contains descriptions of the public methods of the EKFArrivalCost object, i.e., the methods exposed to the MHE object.

5.5.2.1 get_next_P

```
get_next_P(self, t, x, dx, u, c):
"""
Calculates the error covariance matrix at the next time step.
This is achieved by evaluating the Jacobian function at the
work point, solving linear systems to get the system matrices
of the linearised system on the form:

xdot = Ax + Bu + Gw
y = Cx + v

and then discretizing the system using backward Euler to get
the system matrices of the system on the form

x_k+1 = Ax_k + Bu_k+1 + Gw_k+1
y_k = Cx_k + v_k

The error covariance is then updated using these matrices.

Parameters::
    t --
        The time of the work point. Given as a float.

    x --
        A list of tuples on the form (varName, value) of the
        state variables.

    dx --
        A list of tuples on the form (varName, value) of the
        derivatives of the state variables.

    u --
        A list of tuples on the form (varName, value) of the
```

```
            control signals.

        c --
            A list of tuples on the form (varName, value) of the
            algebraic variables.

    Returns::
        P --
            The updated error covariance matrix. 2D numpy array
    """
```

This method is invoked each time the `step` method of the `MHE` object is called, if the sample at index zero is outside of the optimization window. When called, the Jacobian functions that were created during initialization are evaluated at the point sent to the method. A linear system is solved to get the system matrices of the linearized, continuous system. This needs to be done since the model can contain algebraic variables that are not part of the linearized, continuous system representation. The system is then discretized, using backward Euler, to provide the matrices used in (2.29). The equation is then used to update the error covariance matrix and the result is returned.

5.5.2.2 *recalculate_jacobian_functions*

```
recalculate_jacobian_functions(self):
"""
Recalculates the Jacobian functions after a change of the model
parameter values.
"""
```

Called when the `step` method is invoked, before the call to the `get_next_P` method, but only if the dirty flag has been set to `True`. Recreates the Jacobian functions created during the initialization with the new parameter values.

5.5.2.3 *update_process_noise_covariance_matrix*

```
update_process_noise_covariance_matrix(self, process_noise_cov):
"""
Updates the process noise covariance matrix according to a
new covariance list describing the process noise covariance
matrix in continuous time.

Parameters::
    process_noise_cov --
        A list describing the covariance structure of the
        process noise. Items are on the form
        (names, covariance_matrix).
"""
```

The method recalculates the covariance matrix of the process noise when it is updated in the `MHE` object.

5.5.2.4 *update_measurement_noise_covariance_matrix*

```
update_measurement_noise_covariance_matrix(self, measurement_cov):
"""
Updates the measurement noise covariance matrix according
to a new covariance list describing the measurement noise
covariance matrix in continuous time.

Parameters::
    measurement_cov --
        A list describing the covariance structure of the
        measurement noise. Items are on the form
        (names, covariance_matrix).
"""
```

Same as the `update_process_noise_covariance_matrix` method, but for the covariance matrix of the measurement noise.

5.6 Example

This section illustrates how the implemented MHE-framework can be used. The section begins by explaining the model used, which is a continuously-stirred tank reactor (CSTR), introduced in [Hicks and Ray, 1971]. An example script that performs state estimation on the process is then explained to show how the implementation might be used.

5.6.1 Model

The model describes a tank where an exothermic reaction takes place. It has two states, the temperature T and the concentration c. There is a constant inflow to the tank with a fixed temperature and concentration, and a constant outflow from the tank. The speed of the reaction increases with the temperature, meaning that the process can easily spiral out of control. The controllable input of the system is the temperature of the cooling fluid T_c. The dynamics of the system can be described using the differential equations

$$
\begin{aligned}
\dot{c}(t) &= F_0 \frac{c_0 - c(t)}{V} - k_0 c e^{\frac{-EdivR}{T(t)}} \\
\dot{T}(t) &= F_0 \frac{T_0 - T(t)}{V} - \frac{dHk_0 c e^{\frac{-EdivR}{T(t)}}}{\rho C_p} + 2U \frac{T_c - T(t)}{r\rho C_p}
\end{aligned}
\tag{5.2}
$$

where F_0, c_0 and T_0 are the flow rate, concentration and temperature of the inflow. The other parameters are geometric constants describing the tank and thermodynamic constants defining the properties of the reaction.

The system can be described using the Modelica model:

```
model CSTR "A CSTR"
  parameter Modelica.SIunits.VolumeFlowRate F0=100/1000/60 "Inflow";
  parameter Modelica.SIunits.Concentration c0=1000 "Concentration of inflow";
  Modelica.Blocks.Interfaces.RealInput Tc "Cooling temperature";
  parameter Modelica.SIunits.VolumeFlowRate F=100/1000/60 "Outflow";
  parameter Modelica.SIunits.Temp_K T0 = 350;
  parameter Modelica.SIunits.Length r = 0.219;
  parameter Real k0 = 7.2e10/60;
  parameter Real EdivR = 8750;
  parameter Real U = 915.6;
  parameter Real rho = 1000;
  parameter Real Cp = 0.239*1000;
  parameter Real dH = -5e4;
  parameter Modelica.SIunits.Volume V = 100 "Reactor Volume";
  parameter Modelica.SIunits.Concentration c_init = 1000;
  parameter Modelica.SIunits.Temp_K T_init = 350;
  Real c(start=c_init,fixed=true,nominal=c0);
  Real T(start=T_init,fixed=true,nominal=T0);
equation
  der(c) = F0*(c0-c)/V-k0*c*exp(-EdivR/T);
  der(T) = F0*(T0-T)/V-dH/(rho*Cp)*k0*c*exp(-EdivR/T)+2*U/(r*rho*Cp)*(Tc-T);
end CSTR;
```

This model is one of many example models found in the JModelica.org platform [*JModelica.org*].

The two differential equations in (5.2) are defined in the `equation` section of the model. The rest is simply the declaration of the required variables and parameters.

5.6.2 Script

The script presented here will perform state estimation on a CSTR with a horizon length of 5 over a period of 10 seconds, using the sampling time 0.1 seconds. These properties are defined by

```
#Time properties
sim_time  = 10.0
nbr_of_points = 101
horizon = 5
#Sample time
sample_time = sim_time/(nbr_of_points - 1)
#Vector containing the time points
time = N.linspace(0., sim_time, nbr_of_points)
```

The total number of samples used is defined, where the sample at time index zero is included. The number of samples is not something the estimator needs to know in advance, but is defined for the generation of measurement data. A vector containing all the time points is also created. After this the `MHEOptions` object is created and the options are specified.

```
##Set up the MHEOptions-object
MHE_opts = MHEOptions()
#Process noise and input specifications
MHE_opts['process_noise_cov'] = [('Tc', 10.)]
MHE_opts['input_names'] = ['Tc']
#Measurement properties
MHE_opts['measurement_cov'] = [(['c', 'T'], N.array([[1., 0.0], [0.0, 0.1]]))]
#Error covariance matrix
MHE_opts['P0_cov'] = [('c', 10.), ('T', 5.)]
```

The `MHEOptions` object specifies that T_c will be affected by noise with a variance of 10 and that it is used to input a control signal. The covariance lists for the measurement noise and the initial guess show how the covariance matrix can be specified in different ways. The preferable way in this case is that of the initial guess since it takes advantage of the sparse structure of the matrix.

This is followed by creating the `OptimizationProblem` object.

```
#Transfer the optimization problem
op = transfer_optimization_problem(model_string,
                                   package_string,
                                   accept_model = True,
                                   compiler_options=
                                   {"state_initial_equations":True})
```

It is created using `transfer_optimization_problem`. The first two inputs are used to locate the Modelica model, and the third one says that it is an object of the `model` class and not the `optimization` class. The option `state_initial_equations` must be set to true so that all initial equations are replaced by initial equations for the states and parameters for the initial values are created. The naming convention of the created parameters is also utilized to locate them.

After this the preparations are made for generating artificial measurement. This consists of defining the control signal and the initial state values. No controller is used in this example since it simply illustrates the usage of the `MHE` class, thus the control signal is chosen beforehand. The control signal and initial values are defined as

```
#Define a control signal dictionary
sin_signal = N.array([N.sin(2.*N.pi*0.5*time)])
u = dict([('Tc', 300. + 50*sin_signal[0,:])])

#Define the initial state values for the artificial measurement data
x_0 = dict([('c', 1000.), ('T',325.)])
```

Omitted from this example is the actual generation of the artificial measurement data since it is somewhat technical and is not a necessary part of the implementation. The data are created by optimizing with an empty cost function over the entire time interval using one element between every pair of neighboring time points and

one collocation point per element. Process and measurement noise described by the options object are added and the measurements and the state values are extracted. The measurements are saved in `y_meas` and the states in `x`.

The initial values of the state derivatives and the algebraic variables are needed for the first linearization. These are calculated using the `mhe_initial_values` module, imported as `initv`. To generate the initial values the *a priori* estimate and the control signal at time index zero are required. The initial control signal, the *a priori* estimate and the initial values of the state derivatives and algebraic variables are generated by

```
#Define the guess of the initial state values
x_0_guess = dict([('c', 990.), ('T', 330.)])

#Create a dictionary for the first value of all the control signals
u_0 = {}
for (name, data) in u.items():
    u_0[name] = data[0]


#Get the initial values of state derivatives and algebraic variables
(dx_0, c_0) = initv.optimize_for_initial_values(op, x_0_guess, u_0, MHE_opts)
```

The input `u_0` is a dictionary defining the control signal at time index zero. The dictionary has the names of the control signals as its keys and the values of the initial control signals as values. The initial values of the state derivatives and algebraic variables are stored in `dx_0` and `c_0` respectively.

The `MHE` object can now be created with

```
#Create the MHE-object
MHE_object = MHE(op, sample_time, horizon, x_0_guess, dx_0, c_0, MHE_opts)
```

After creating a dictionary for storing the results a loop is used to call the `step` method at every time step. The method is provided with the control signal and measurements of the current sample and returns the state estimate of the next sample.

```
#Create a dictionary for storing the estimates
x_est = {'c':[x_0_guess['c']], 'T':[x_0_guess['T']]}

#Lists of the state and input names for convenience
state_names = ['c', 'T']
input_names = ['Tc']
#Loop over all the indices in the interval
for t in range(0, (nbr_of_points - 1)):
    #Create the list of measurements at the current sample
    y_t = []
    for name in state_names:
        y_t.append((name, y_meas[name][t]))
    #Create the list of control signals at the current sample
    u_t = []
    for name in input_names:
```

```
        u_t.append((name, u[name][t]))

    #Perform the estimation
    x_est_t = MHE_object.step(u_t, y_t)

    #Save the estimates
    for name in state_names:
        x_est[name].append(x_est_t[name])
```

Using matplotlib the results can be visualized with

```
#Plot the results
plt.close('MHE')
plt.figure('MHE')
plt.subplot(3, 1, 1)
plt.plot(time, x_est['c'])
plt.plot(time, x['c'], ls = '--', color = 'k')
plt.plot(time, y_meas['c'])
plt.legend(('Concentration estimate',
            'Simulated concentration',
            'Measured concentration'))
plt.grid()
plt.ylabel('Concentration')

plt.subplot(3, 1, 2)
plt.plot(time, x_est['T'])
plt.plot(time, x['T'], ls = '--', color = 'k')
plt.plot(time, y_meas['T'])
plt.legend(('Temperature estimate',
            'Simulated temperature',
            'Measured temperature'))
plt.grid()
plt.ylabel('Temperature')

plt.subplot(3, 1, 3)
plt.step(time, u['Tc'])
plt.grid()
plt.ylabel('Cooling temperature')
plt.xlabel('time')
plt.show()
```

which results in the plots in Figure 5.2.

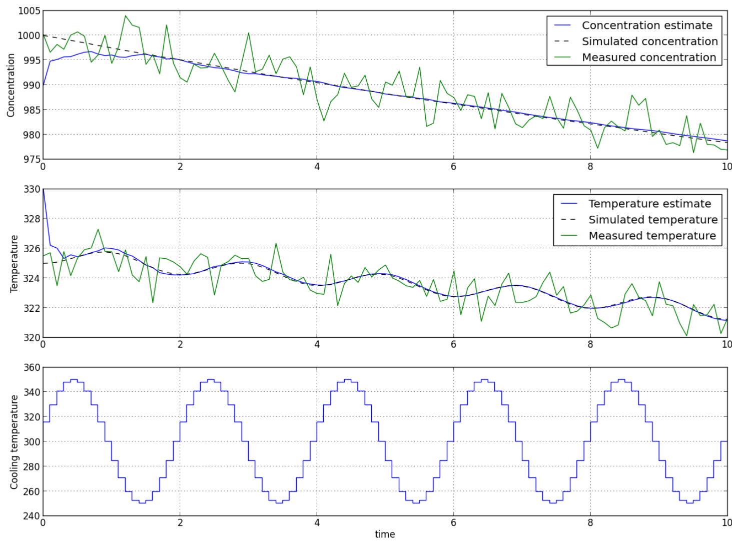


Figure 5.2 *The simulated, estimated and measured trajectories of the states for the CSTR example. Third plot shows the used control signal.*

6

Results

In this chapter, a number of simulation examples are used to evaluate the performance and show the capabilities of the implemented MHE. Four different examples are used. The first example compares the performance of the MHE to that of the Kalman filter on an LTI system. The second example aims to investigate the potential improvement achievable with constraints. A linear system and the nonlinear CSTR presented in Section 5.6.1 are used to illustrate this. The third simulation example uses the CSTR, which is a small system, and a large model of a simplified thermal power plant, to test the effects of different horizon lengths. Finally, the plant model is used to evaluate how the performance is affected by having a mismatch of the parameters used in the model used to generate the measurement data and in the model used in the MHE.

All the measurement data used for the tests were generated using `OptimizationProblem` objects, except for the plant model where the size of the model made it impossible to have enough elements to create data for longer time intervals. For the plant model, simulation using FMUs was used. Using FMUs means that piece-wise constant inputs could not be used, resulting in a mismatch between the model used to generate data and the one used by the MHE.

6.1 Kalman Filter Comparison

As stated in Section 2.4, using the EKF covariance update as the arrival cost approximation and having a horizon of one sample results in the Kalman filter and the MHE becoming the same, for LTI systems. To evaluate if this property is preserved for this implementation, the performance of the implemented MHE is compared to that of a Kalman filter. To make the comparison valid, backward Euler is used as the discretization scheme for the model used by the Kalman filter. A one-step ahead Kalman filter is used, meaning that the *a priori* estimate is used.

Since an implicit discretization is used, the Kalman filter should use the control signal at the next sample to generate the estimate at the next sample. Since this is

unreasonable, the same approximation as in the MHE is used, i.e., the control signal at the current sample is used.

The model used is the simple LTI system. This system can be expressed in state space form as

$$\begin{aligned}\dot{x} &= \begin{bmatrix} -1 & -0.5 \\ -0.5 & -1 \end{bmatrix} x + \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} u + \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} w \\ y &= \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} x + v\end{aligned}$$

where x is the state vector, y the measurement vector, u the control vector and w and v are the process and measurement noise vectors, respectively. When discretized using backward Euler with a sampling period of 0.1 s it results in the system representation

$$\begin{aligned}x_{k+1} &= \begin{bmatrix} 0.953 & -0.023 \\ -0.023 & 0.953 \end{bmatrix} x_k + \begin{bmatrix} 0.048 & -0.001 \\ -0.001 & 0.048 \end{bmatrix} u_{k+1} + \begin{bmatrix} 0.048 & -0.001 \\ -0.001 & 0.048 \end{bmatrix} w_{k+1} \\ y_k &= \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} x_k + v_k\end{aligned}$$

where x_k is the state vector, y_k the measurement vector, u_k the control vector, w_k and v_k are the process and measurement noise vectors, respectively, and k is the time index. The process noise covariance matrix and the measurement noise covariance matrix in continuous time are given by

$$Q_c = \begin{bmatrix} 10 & 0 \\ 0 & 10 \end{bmatrix}, \qquad R_c = \begin{bmatrix} 0.01 & 0 \\ 0 & 0.01 \end{bmatrix}$$

respectively. These are discretized as

$$Q_d = hQ_c = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}, \qquad R_d = R_c/h = \begin{bmatrix} 0.1 & 0 \\ 0 & 0.1 \end{bmatrix}$$

where h is the sample time.

Using a square wave for both the control signals and an initial error covariance matrix

$$P_0^- = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}$$

over a time period of 5 s gives the results presented in Figure 6.1. From the plots, it is clear that the Kalman filter and the MHE give the same results. The effect of the control signal approximation is clear in the points where the approximation is least valid, i.e., in the points where u_k changes drastically. In these points, most obvious for the first state around the point at 1 s, the estimators are using a control signal which is drastically incorrect and thus produces an erroneous estimate.

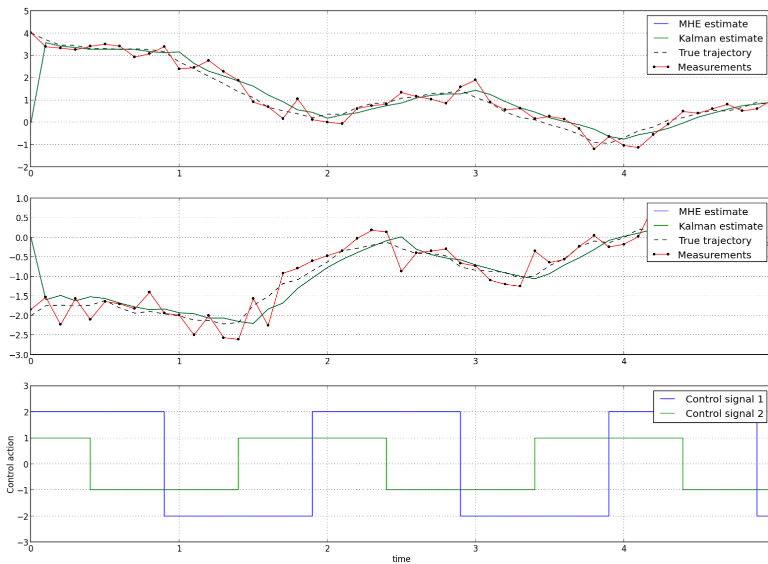


Figure 6.1 *Comparison of the estimates generated by the MHE and the Kalman Filter for the LTI system. The green and the blue lines overlap for both of the states.*

6.2 Effect of Constraints

This section investigates how the addition of constraints to the optimization problem affects the performance of the MHE estimator. Two different systems are used, an LTI system used in [Rao and Rawlings, 2002] and the CSTR presented in Section 5.6.1.

6.2.1 Constrained LTI

The aim of this section is to perform a test very similar to the one found in [Rao and Rawlings, 2002], where the effect constraints on an LTI system is investigated. Therefore the same constraint and model are used with similar initial values and the same covariance matrices. The noise sequence will be different and for this reason there is no straight comparison to be made, but more general conclusions can be drawn. The performance of the MHE is compared to that of the Kalman filter in accordance with the evaluation presented in [Rao and Rawlings, 2002].

The assumption that is made is that the characteristics of the process noise w_k is known. The process noise sequence w_k is the absolute value of the independent, zero-mean, normally distributed sequence z_k. To capture this knowledge, the constraint $w_k \geq 0$ is added.

The system used is described by the transfer function $G(s) = (-3s+1)/(s^2+3s+1)$. The system can, for instance, be represented in state space form as

$$\dot{x} = \begin{bmatrix} 0.097 & 0.984 \\ -0.984 & -3.005 \end{bmatrix} x + \begin{bmatrix} 0.001 \\ 1 \end{bmatrix} w$$
$$y = \begin{bmatrix} 1 & -3 \end{bmatrix} x + v$$

which, when discretized using backward Euler and the sample period 0.3 s, yields the system

$$x_{k+1} = \begin{bmatrix} 0.984 & 0.153 \\ -0.153 & 0.502 \end{bmatrix} x_k + \begin{bmatrix} 0.046 \\ 0.151 \end{bmatrix} w_{k+1}$$
$$y_k = \begin{bmatrix} 1 & -3 \end{bmatrix} x_k + v_k$$

Using the covariance matrices in [Rao and Rawlings, 2002], i.e.,

$$Q_d = 1, \quad R_d = 0.1, \quad P_0^- = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}$$

as well as the same initial *a priori* state estimate $\hat{x}_0^- = [0,0]$ and the horizon length 10, yields the results presented in Figure 6.2. The plot of the estimated process noise uses the estimate at the current sample since the fact that estimator uses prediction means that the estimated process noise for the next sample is always zero.

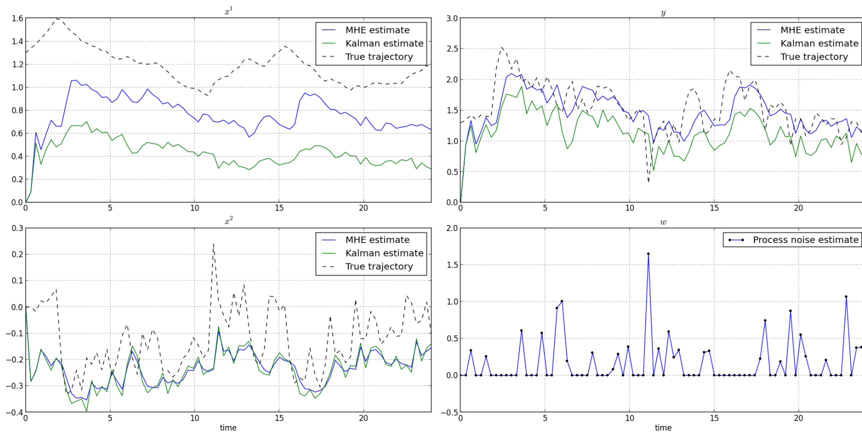


Figure 6.2 *Comparison of the state and measurement estimates generated by the MHE and the Kalman Filter, and the process noise estimates at the current sample for the MHE.*

The results generated here are similar in structure to those presented in [Rao and Rawlings, 2002]. The estimate of the first state is vastly improved while the

estimate of the second state is more or less unchanged. There is still a large bias, since the mean of the process noise is not zero. The estimate of the measurement y is slightly different in character from that in [Rao and Rawlings, 2002]. After some investigation it was discovered that this is likely due to the fact that in this thesis the *a priori* estimate of the Kalman filter is used. The plot of the estimated process noise at the current sample shows that the constraint seems to be active for a majority of the samples.

6.2.2 Constrained CSTR

In this example the CSTR system is used to investigate how the use of constraints founded in knowledge about the physical properties of the system can be used to improve the results. The fact that a concentration can not be negative can be captured using the constraint $c \geq 0$. If the system is then run close to this boundary in an environment with significant measurement noise, the benefit of constraints can easily be illustrated.

Consider the situation where the CSTR has a concentration of zero and the reactant is being introduced at an increased flow rate. The concentration will quickly deviate from the constraint, but significant measurement noise can make an estimator not utilizing constraints produce infeasible estimates. In Figure 6.3, this case is presented with a constant cooling temperature of 370 K, a horizon of 10 samples and a sample period of 0.2 s and the covariance matrices

$$Q_c = 20, \quad R_c = \begin{bmatrix} 10 & 0 \\ 0 & 1 \end{bmatrix}, \quad P_0^- = \begin{bmatrix} 5 & 0 \\ 0 & 2 \end{bmatrix}$$

The unconstrained MHE produces a number of infeasible estimates when working close to the constraint, while the constrained MHE avoids this completely. Generating infeasible estimates can be a major problem when combining the estimator with a controller that employs constraints, such as MPC. Being able to guarantee that constraints are not violated is therefore a desirable property of an estimator. The estimate of the temperature appears virtually identical for both of the estimators.

6.3 Plant Model

To evaluate the performance of the estimator on a larger system, a simplified model of a thermal power plant was chosen. The model is a part of the larger model presented in [Runvik, 2014]. The code implementing the model is very long, well over 2000 lines, meaning that the model is hard to interpret without using a graphical overview. Figure 6.4 shows a graphical representation of the model viewed using the Modelica interpreter Dymola [*Dymola*]. Three major components, the evaporator, the separator, and the wall of the separator, make up the model. Using these components several times with different parameter values and a few other components, such as turbines and pumps, an approximate model of an entire thermal

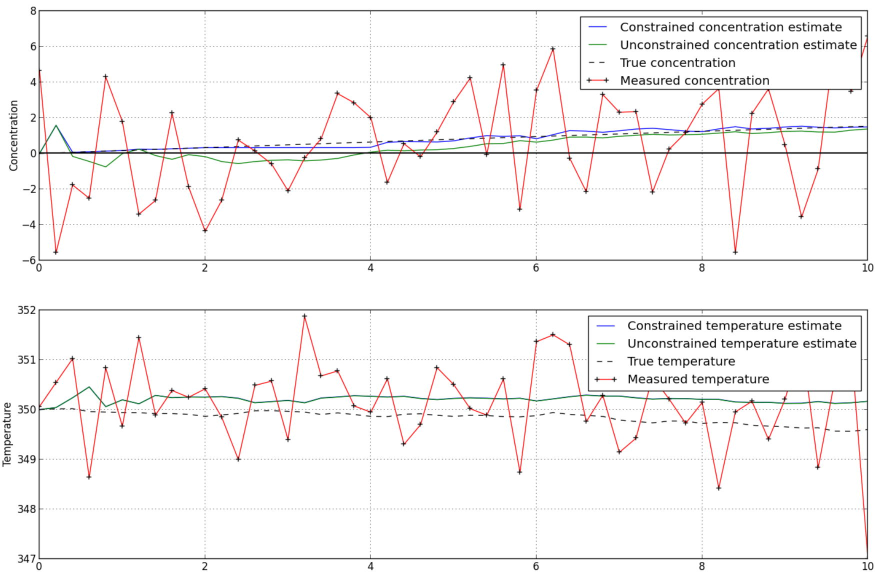


Figure 6.3 *Comparison of an MHE utilizing constraints and a corresponding MHE not utilizing constraints. The constraint is indicated by the bold black line.*

power plant can be constructed. This model gives a good example of how a large part of modeling in Modelica is done, using a graphical interface to connect different components, rather than writing all the code by hand. This way components can be connected to easily create large systems.

The system has six states, two inputs, 36 algebraic variables and 36 equations. The states and the inputs of the system are presented in Table 6.1.

Table 6.1 *The states and inputs of the plant model. States belonging to the evaporator component are denoted by the subscript e, the ones belonging to the separator s and the ones belonging to the separator wall sw.*

ρ_s	Water density in the separator.
u_s	Specific internal energy of the water in the separator.
T_{sw}^2	Temperature of the second node in the separator wall.
ρ_e	Water density in the evaporator.
u_e	Specific internal energy of the water in the evaporator.
T_e^{wall}	The temperature of the wall in the evaporator.
gasFlow	The flow rate of the gas input.
steamFlow	The flow rate of the steam input.

For this model, and many larger models in general, it is necessary to tell the

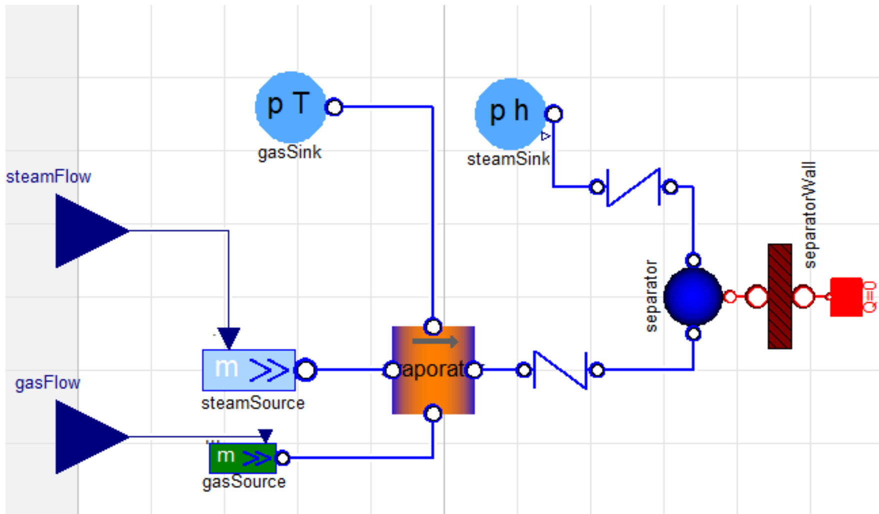


Figure 6.4 *Graphical overview of the plant model.*

compiler not to use function inlining. This is done using the `inline_functions` option in Table 4.1. If this is not done for this model, the performance of the solver will decrease gradually to the point where it can not find a solution.

The model is very sensitive to the initial values of the states. To find values which made it possible to perform optimization at all, the system was simulated for a long time until steady state were reached. The values of the states in steady state were then used as the initial values for the starting point of the iterations in the optimization. Providing the solver with initial guesses for many of the variables improved the stability and performance of the optimization algorithm, but the maximum number of elements that could be used before the solver could not find a solution was still only approximately two.

A part of this sensitivity has its root in the fact that some of the equations contain `if` statements that introduce a discontinuity. In this case the sign of a number of variables was used to determine the flow direction in some components and this introduced the discontinuity. Using the steady state initial values, the system was always strictly on one side of the if statement, but when searching for a solution the solver can still search on both sides of the discontinuities, which greatly affects the performance of the solver. Introducing constraints on the variables in the if statements that keep the solver away from the discontinuities greatly improved the performance of the solver and allowed the use of up to seven elements. For this model, constraints are utilized not to model a physical or practical limit of the system but for solver stability.

6.4 Horizon Length

This section investigates the effect of the horizon length on the estimate. Two systems are used, the CSTR and the plant model. Performance will be evaluated using the mean square error (MSE), that is the mean of the squared error of every sample.

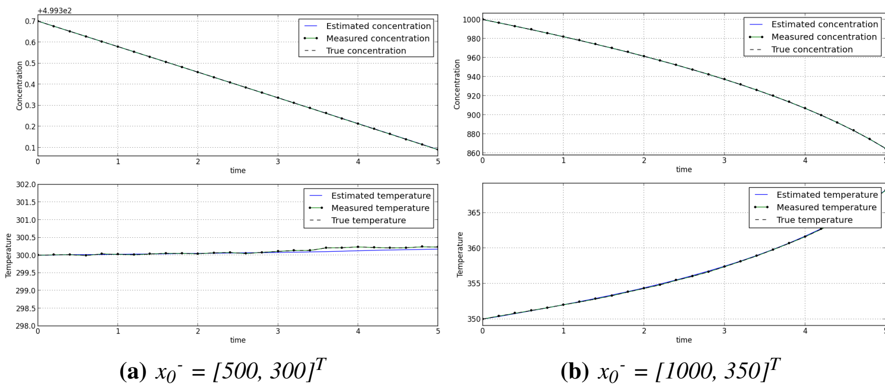
6.4.1 CSTR

This section uses the CSTR model to illustrate the effect different horizon lengths has on the estimation for different cases. A sample period of 0.2 s and a total of 25 samples are used for all the cases, along with the covariance matrices

$$Q_c = 50, \quad R_c = \begin{bmatrix} 0.2 & 0 \\ 0 & 0.02 \end{bmatrix}, \quad P_0^- = \begin{bmatrix} 10 & 0 \\ 0 & 5 \end{bmatrix}$$

where the subscript c indicates continuous time.

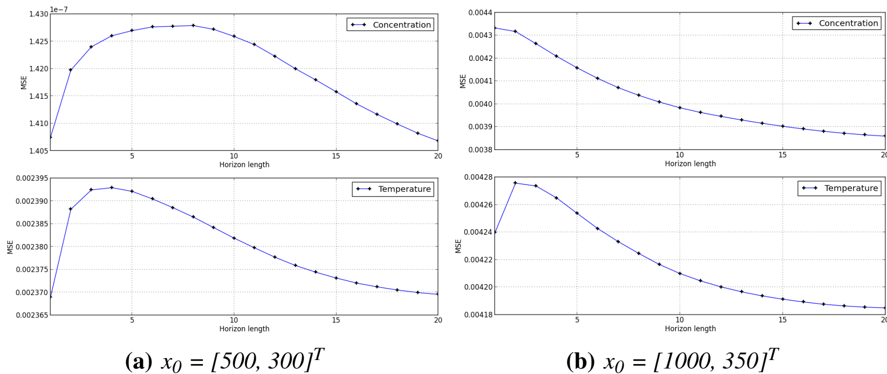
To illustrate the effect the horizon has in the most simple case for the CSTR system, a constant control signal of 300 K is used with no measurement noise added to the generated measurement data. Figure 6.5 presents the result for two different sets of initial values, $x_0^- = [500, 300]^T$ and $x_0^- = [1000, 350]^T$. The two different initial values were chosen since they result in very different system behaviors, making it possible to investigate different aspects of the effect created by the horizon length. Plots for different horizon lengths are not shown because there was no noticeable difference. Figure 6.6 presents the MSE as a function of horizon length for both the initial values.



(a) x_0^- = [500, 300]^T **(b)** x_0^- = [1000, 350]^T

Figure 6.5 *The estimates of the CSTR for different initial values, having no measurement noise and a constant control signal. A horizon length of 10 is shown for both plots.*

In Figure 6.6(a) the structure is surprising. That the shortest horizon yields the best results for this system is something that could not be explained with certainty.



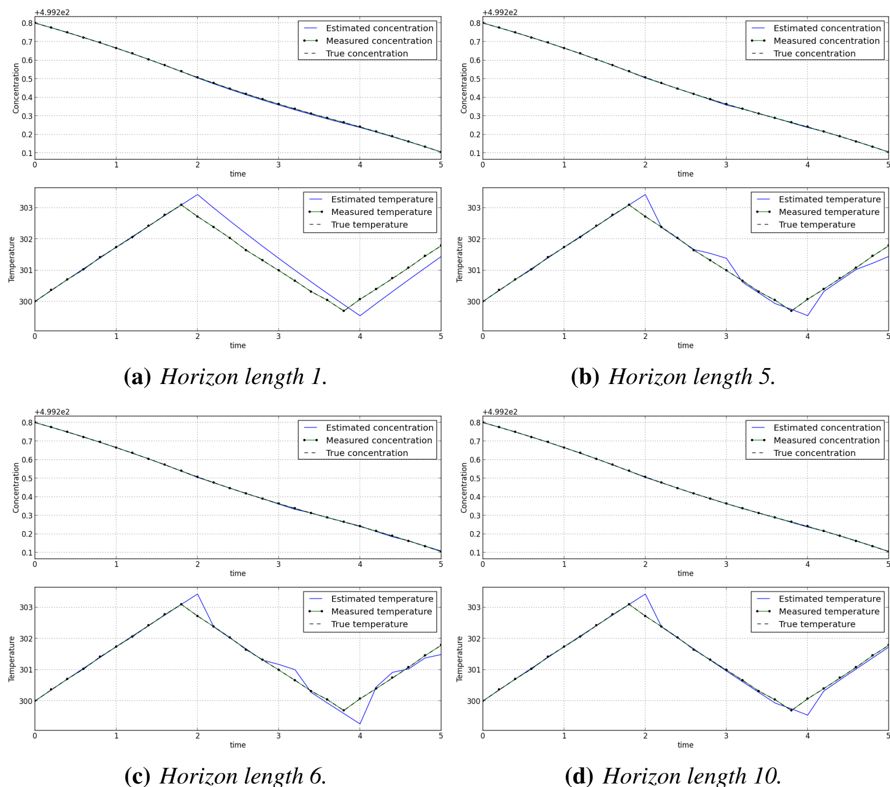
(a) x_0 = [500, 300]^T **(b)** x_0 = [1000, 350]^T

Figure 6.6 *The MSE as a function of the horizon length for the CSTR with no measurement noise and a constant control signal for the two initial values.*

Though the difference is very small. The MSE should decrease with longer horizons and its derivative should approach zero for long horizons, which is the case for the longer horizons in the plot. One possible explanation is that the cost function approximation introduces an error that is then overcome by the benefit of having a longer horizon as the horizon length increases. The structure in Figure 6.6(b) is closer to that which is expected. There is, however, still a lower MSE for horizon length one for both the states, and for horizon length two for the temperature. It appears that the structure in Figure 6.6(a) becomes less significant when estimating larger changes.

To investigate how an increased horizon length helps reduce the effect of the control signal approximation, a square wave control signal with frequency 0.25 Hz, minimum 250 K and maximum 350 K is used, again with no measurement noise to reduce the dependency of the test on random quantities. Figure 6.7 shows the results for four different horizon lengths. Here the length of the horizon reduces the effect the control signal approximation has on the system. The poor estimate resulting from the control signal approximation results in a periodic poor estimate. This happens with a periodicity of a horizon length samples and is caused by the fact that the poor estimate is used in the arrival cost term a horizon samples later. The poor estimate is also used in the linearization of the system and therefore, in turn, causes the error covariance estimate to be poor. A similar phenomenon is briefly discussed in [Tenny and Rawlings, 2002], where a poor initial guess of the error covariance matrix gives the same kind of periodic behavior. By coincidence this periodic behavior improves the estimate for the horizon lengths five and ten, where the period of the square wave and the poor estimates happen to match and subtract from each other resulting in the effect of the control signal approximation being reduced.

Figure 6.8 presents the MSE as a function of the horizon length for the case

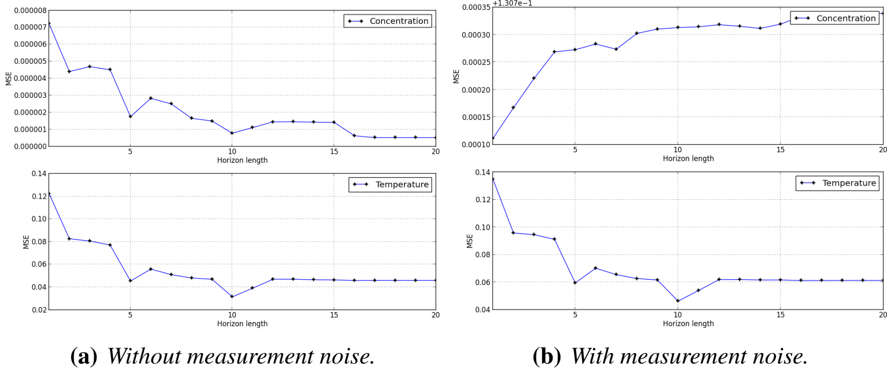


(a) *Horizon length 1.* **(b)** *Horizon length 5.*

(c) *Horizon length 6.* **(d)** *Horizon length 10.*

Figure 6.7 *The estimates of the CSTR for different lengths of the horizon, having no measurement noise and a square wave control signal.*

when a square wave control signal is used. Both the case without measurement noise and with measurement noise is included to illustrate the difference between the two. Figure 6.8(a) presents the MSE as a function of the horizon length for the test. Here the MSE has local minima at horizon lengths two, five and ten where the control signal approximation error is accidentally counteracted by the periodicity of the poor estimates. Figure 6.8(b) presents the case with measurement noise. The characteristic for the temperature is much the same as for the case where no measurement noise was added, but for the concentration the addition of measurement noise has made a change. The MSE now grows larger with the length of the horizon. It should be noted that the difference in scale between the plots is large. The MSE of the concentration is only slightly increased, while the MSE of the temperature is many times decreased as the horizon length grows.



(a) *Without measurement noise.* **(b)** *With measurement noise.*

Figure 6.8 *The mean square error as a function of the horizon length for the CSTR with and without measurement noise using a square wave control signal. The figure with measurement noise shows the average MSE for 30 different noise sequences. Different scales are used in the left and right figures.*

6.4.2 Plant

For the larger plant model, the length of the horizon can be a maximum of seven samples before the solver fails. The length of the horizon for the plant model is therefore evaluated up to this limit. The covariance matrices used are

$$
\begin{aligned}
Q_c &= \mathrm{diag}(1,\ 1), \\
R_c &= \mathrm{diag}(100,\ 5,\ 0.5,\ 100,\ 0.5,\ 5), \\
P_0^- &= \mathrm{diag}(15000,\ 1,\ 0.1,\ 15000,\ 0.1,\ 1)
\end{aligned}
$$

with

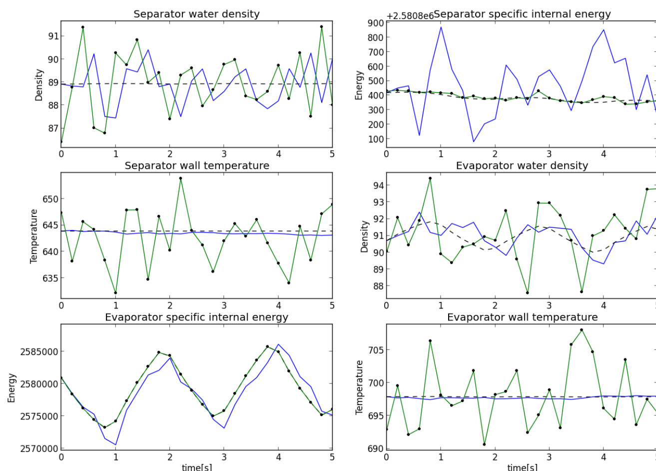
$$
u = \begin{bmatrix} gasFlow & steamFlow \end{bmatrix}^T, \quad x = y = \begin{bmatrix} u_e & T_e^{wall} & \rho_e & u_s & \rho_s & T_{sw}^2 \end{bmatrix}^T
$$

which corresponds to the covariance lists

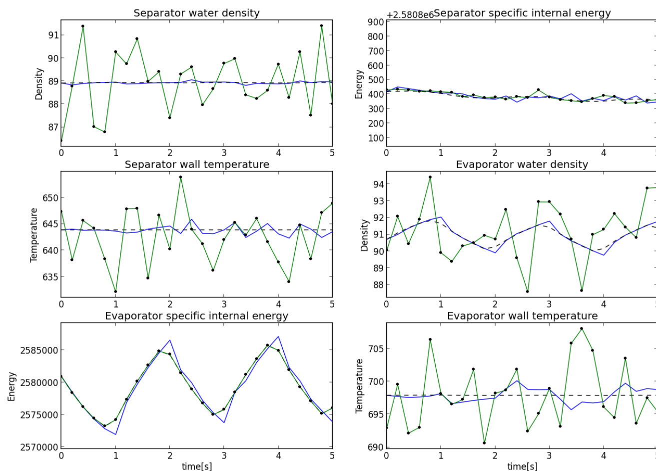
```
process_noise_cov = [('gasFlow', 1.), ('steamFlow', 1.)]
P0_cov = [('evaporator.u', 15000.), ('evaporator.T_wall', 1),
          ('evaporator.rho', 0.1), ('separator.u', 15000.),
          ('separator.rho', 0.1), ('separatorWall.T__2', 1.)]
measurement_cov =[('evaporator.u', 100.), ('evaporator.T_wall', 5.),
                  ('evaporator.rho', 0.5), ('separator.u', 100.),
                  ('separator.rho', 0.5), ('separatorWall.T__2', 5.)]
```

A total of 25 samples are used with a sampling period of 0.2 s.

Figure 6.9 presents the estimated states when using different horizon lengths. A square wave control signal with the frequency 0.5 Hz, the minimum 90 and the maximum 110 is used for both of the inputs.



(a) *Horizon length 1.*



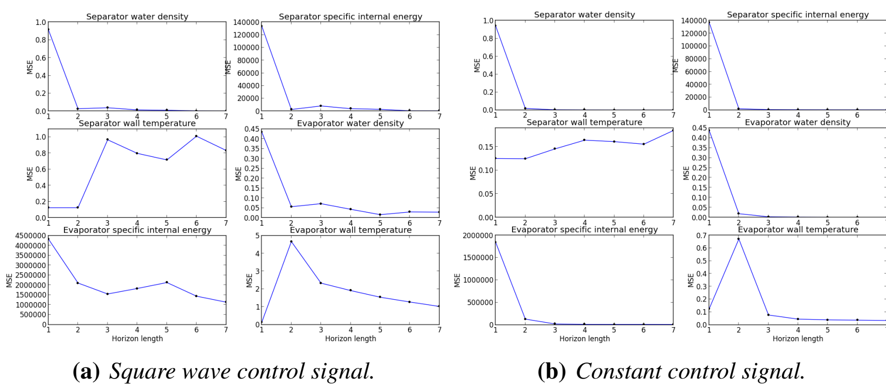
(b) *Horizon length 7.*

Figure 6.9 *The estimates of the plant model for different lengths of the horizon, using a square wave control signal. The green lines are the measurements, the blue lines the estimates and the dashed black lines the true trajectories.*

By inspecting the plots it is evident that the quality of the estimates for four of the six states is improved for longer horizons. For the temperature of the evaporator wall and the separator wall this is not the case. The reason behind this is not easy to trace. The model is large and the states are intricately connected making conclusions hard to draw. Adding to this the fact that the model is very sensitive and that many

different factors cause issues for the solver, makes the analysis even more difficult. It is fully possible that the improved estimates of the other states are causing this behavior, but is is also possible that the solver finds a non-global minimum instead of the global minimum.

Figure 6.10 shows the MSE as a function of the horizon length both for a square wave control signal and a constant control input at 110 for both signals. Figure 6.10(a) shows many of the phenomena that could be observed in Figure 6.9, that is, the estimate is generally improving for longer horizons for all of the states except for the temperature of the evaporator and separator walls. For the evaporator wall temperature, the curve shows the expected declining trend after the length of two, but for the separator wall temperature there is not a clear trend. It might be possible to determine if there is a trend if it was possible to use longer horizons.



(a) *Square wave control signal.* **(b)** *Constant control signal.*

Figure 6.10 *The average mean square error for ten different noise sequences as a function of the horizon length for the plant model using two different control signals.*

Figure 6.10(b) presents the case where two constant inputs were used. Here too the same structure can be seen where just the estimates of the temperature states are not strictly improving when using a longer horizon. It is worth to note that in contrast to the CSTR system, the case where constant inputs are used gives, at least for most of the states, improved estimates for longer horizons.

6.5 Parameter Mismatch

This section will investigate how estimator performance is affected when the model parameter values used to generate the measurement data are different from those used in the estimator. This is therefore an investigation of the estimators sensitivity to model uncertainties. The plant model will be used for this. All the independent parameters, that is, parameters whose value does not depend on other parameters,

of the model will have their value changed to different percentages of their original value. The results are evaluated using the MSE for the different mismatches.

Figure 6.11 presents the estimates for four different cases of the parameter values being mismatched. The same set up as in Section 6.4.2 is used, meaning that the number of samples, sample period, square wave input signals and covariance lists are the same. In this test, however, the horizon length is fixed and set to six.

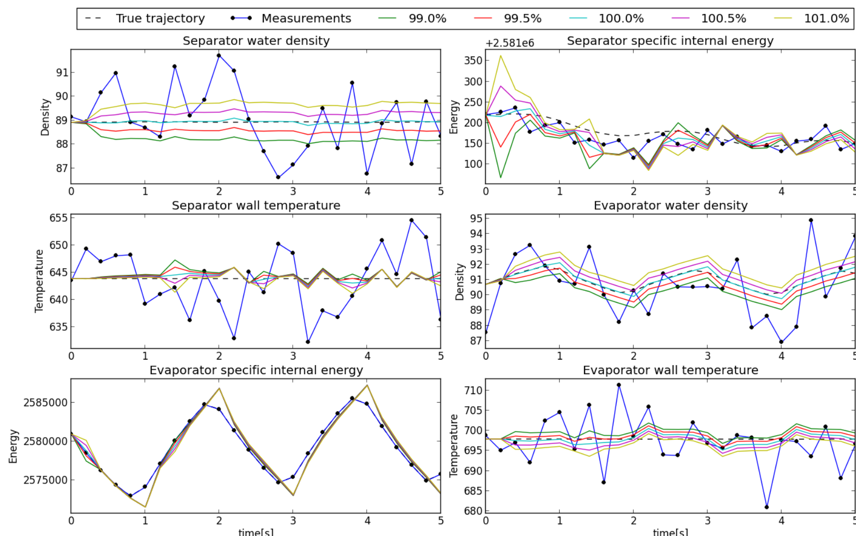


Figure 6.11 *The estimates of the plant model for different mismatches of the parameter values. The blue lines are the measurements, the dashed black lines the true trajectories and the other colors are the estimates for different mismatches. For the specific internal energy of the evaporator all the estimates overlap and the measurements and the true trajectory also overlap.*

For some of the states, mainly the water densities and the evaporator wall temperature, there seems to be a linear relation between the parameter mismatch and the resulting error, since the curves seem to move a fixed amount between each plot. The large spike in the plots for the specific internal energy of the separator also seems to grow close to linearly with the mismatch. If any of these are actually linear behaviors is of course hard to judge from the plots, but what is clear is that the parameter mismatch has a large effect on the performance of the estimator.

To reduce the effect of having an incorrect model it is possible to change the weight matrices, that is, the covariance matrices, of the cost function. By increasing the magnitude of the process noise covariance matrix, or by decreasing the magnitude of the measurement noise covariance matrix, less faith is put in the model and more faith is put in the measurements, possibly improving the results when work-

ing with an incorrect model. Figure 6.12 presents the average MSE for ten different noise sequences as a function of the percentage of the original parameter values for three different choices of covariance matrices. The first case uses the original covariance matrices, the second case uses a process noise covariance matrix where the intensities have been increased by a factor 100 and in the third case the intensities in the measurement covariance matrix have been reduced by a factor 100 in addition to the same change of the process noise covariance matrix as in the second case.

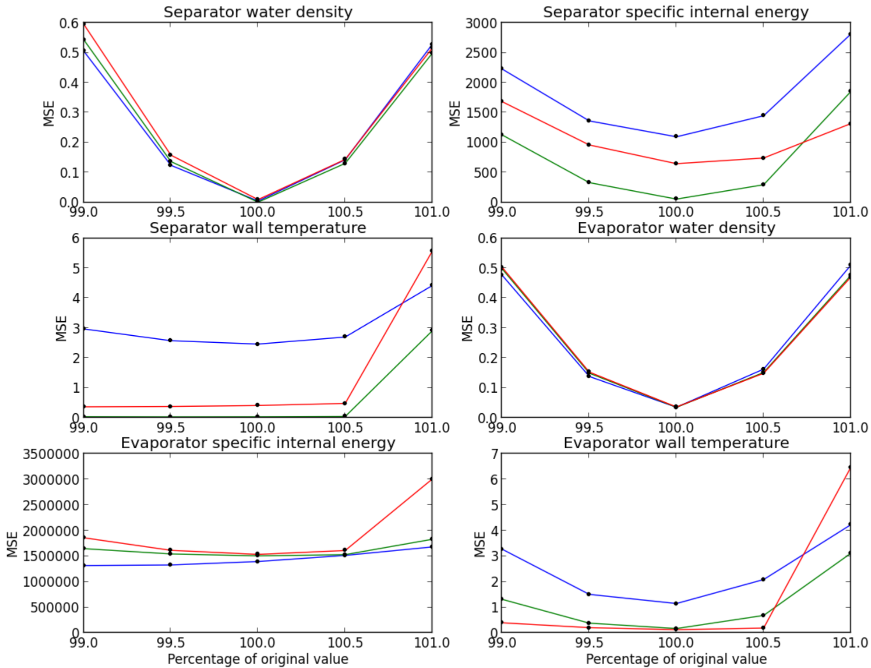


Figure 6.12 *The average MSE of ten different noise sequences as a function of the percentage of the original parameter values for the plant model using three different sets of weight matrices. The blue lines are for the original weights, the green lines for when the process noise covariance matrix has been multiplied by a factor* 100 *and the red lines for when both the process and measurement noise covariance matrices have been multiplied by* 100 *and* 0.01 *respectively.*

For a number of the states, the green and red curves show a visible flattening of the curves meaning that for these states the change of covariance matrices has improved the performance of the estimator when used on incorrect models. More surprising is that for some states the change of covariance matrices yield an overall lower MSE. This might be explained either by the fact that the dynamics of the system is very complicated meaning that an improvement of one states estimate

may be linked to the worsening of another. Another possible explanation is that since the process noise enters the system the same way as the control signal, thus having its distribution affected by the system. This in turn can mean that the best covariance matrices to use in the cost function are no longer those used to generate the noise. For the blue curves the states for which there was a linear behavior in Figure 6.11, the corresponding curves resemble parabolas. This serves to strengthen the hypothesis that for these state there is some kind of linear dependence between the mismatch and the estimates.

7

Conclusions and Future Work

A nonlinear MHE was successfully implemented in the JModelica.org platform. An effort was made to keep the user interface as simple as possible and to make the inputs few and on an intuitive form. The implementation works with many already existing Modelica models and requires little effort on the user's part. The examples in Chapter 6 show that the implementation performs well for several different systems. One of the major benefits of using MHE is the ability to utilize constraints, which is illustrated in the Chapter 6. The importance of the horizon length was also shown. While there was some behavior which could not be explained, it was clear that a longer horizon benefits the estimator. For this implementation in particular the horizon length is of importance since it helps decrease the effect of the control signal approximation.

Since the goal was to implement the core functionality of a possible future MHE platform in JModelica.org, there are many possible improvements that can be made.

Speed was not a key goal in this implementation, but is a key factor when considering the types of real systems the implementation could be used on. In the later stages of the thesis *warm start* capabilities for optimization were introduced in JModelica.org. Warm start refers to exploiting the fact that the discretization is the same for all the optimizations in the MHE if the problem size is constant. The results from the previous optimization is also used as an initial guess for the next optimization, helping convergence. For many models the discretization process can take up a substantial part of the total solution time. Since the problem size of the MHE is not constant for the first horizon length samples, the warm start of JModelica.org could not be directly applied to the MHE implementation. This in combination with the lack of time forced the decision to not use warm start in the implementation. To include warm start in the implementation, either optimization without warm start would have to be used until the problem size becomes constant or warm start is used but unavailable measurements are masked.

The approximation of the control signal is one of the major error sources in the implementation. To remove the periodic effect an incorrect estimate causes the smoothing update introduces in [Tenny and Rawlings, 2002] can be implemented. By using an estimate that utilizes more of the measurement data, the periodic effect can be removed. To lessen the effect of the control signal approximation itself, a method that allows the user to provide a guess for the next control signal can be implemented. When using MPC, a more accurate guess for the next control signal is readily available meaning that the effect of the approximation could be lessened.

Implementing the ability to use higher order discretizations than backward Euler would make it possible to reduce the discretization error in many cases. The difficulty lies in how to approximate the cost function and how to set the masking signal.

The implementation performs prediction. In some cases it is more natural to produce the estimate for the current sample using measurements up to the current sample, i.e., filtering. It could therefore be beneficial to add filtering functionality to the MHE platform.

Since the arrival cost approximation is separate from the `MHE` class, it is possible to implement other types of arrival cost approximations than the EKF covariance matrix update. This would give the user the ability to chose the approximation that works best for the system they are using.

Bibliography

Åkesson, J. (2008). "Optimica—an extension of Modelica supporting dynamic optimization". In: *Proc. 6th International Modelica Conference 2008*.

Åkesson, J., K.-E. Årzén, M. Gäfvert, T. Bergdahl, and H. Tummescheit (2010). "Modeling and optimization with Optimica and JModelica.org—languages and tools for solving large-scale dynamic optimization problems". *Computers & Chemical Engineering* **34**:11, pp. 1737–1749.

Assimulo. http://www.jmodelica.org/assimulo. Accessed: 2015-02-03.

Bishop, G. and G. Welch (2001). "An introduction to the Kalman filter". *Proc. of SIGGRAPH, Course* **8**:27599-23175, p. 41.

Brown, R. (1983). *Introduction to random signal analysis and Kalman filtering*. Vol. 8. Wiley New York.

CasADi. https://github.com/casadi/casadi/wiki. Accessed: 2015-02-03.

Chen, W., P. Chang, T. Conte, and W.-M. Hwu (1993). "The effect of code expanding optimizations on instruction cache design". *Computers, IEEE Transactions on* **42**:9, pp. 1045–1057. ISSN: 0018-9340. DOI: 10.1109/12.241594.

Dymola. http://www.3ds.com/products-services/catia/products/dymola. Accessed: 2015-05-07.

Gelb, A. (1974). *Applied Optimal Estimation*. Ed. by T. A. S. C. The Technical Staff. MIT Press, One Rogers Street, Cambridge MA 02142-1209, USA.

Hicks, G. and W. Ray (1971). "Approximation methods for optimal control synthesis". *The Canadian Journal of Chemical Engineering* **49**:4, pp. 522–528.

IPOPT. https://projects.coin-or.org/Ipopt. Accessed: 2015-05-07.

JModelica.org. http://www.jmodelica.org/. Accessed: 2015-02-03.

Kalman, R. (1960). "A new approach to linear filtering and prediction problems". *Journal of Fluids Engineering* **82**:1, pp. 35–45.

Liu, J. (2013). "Moving horizon state estimation for nonlinear systems with bounded uncertainties". *Chemical Engineering Science* **93**:0, pp. 376 –386. ISSN: 0009-2509. DOI: http://dx.doi.org/10.1016/j.ces.2013.02.030. URL: http://www.sciencedirect.com/science/article/pii/S0009250913001310.

MATLAB. www.mathworks.se/products/matlab/. Accessed: 2015-02-03.

matplotlib. http://matplotlib.org/. Accessed: 2015-02-03.

Modelica and the Modelica Association. https://www.modelica.org/. Accessed: 2015-02-03.

Modelon AB. http://www.modelon.com/. Accessed: 2015-02-03.

NumPy. http://www.numpy.org/. Accessed: 2015-02-03.

Qu, C. and J. Hahn (2009). "Computation of arrival cost for moving horizon estimation via unscented Kalman filtering". *Journal of Process Control* **19**:2, pp. 358 –363. ISSN: 0959-1524. DOI: http://dx.doi.org/10.1016/j.jprocont.2008.04.005. URL: http://www.sciencedirect.com/science/article/pii/S0959152408000711.

Rao, C. (2000). *Moving horizon strategies for the constrained monitoring and control of nonlinear discrete-time systems*. PhD thesis. University of Wisconsin-Madison.

Rao, C. and J. Rawlings (2002). "Constrained process monitoring: moving-horizon approach". *AIChE Journal* **48**, pp. 97–109.

Rao, C., J. Rawlings, and D. Mayne (2003). "Constrained state estimation for nonlinear discrete-time systems: stability and moving horizon approximations". *IEEE Transactions on Automatic Control* **48**:2, pp. 246–258. ISSN: 0018-9286. DOI: 10.1109/TAC.2002.808470.

Rawlings, J. and L. Ji (2012). "Optimization-based state estimation: current status and some new results." *Journal of Process Control* **22**:8, pp. 1439–1444. ISSN: 09591524. URL: http://ludwig.lub.lu.se/login?url=http://search.ebscohost.com.ludwig.lub.lu.se/login.aspx?direct=true&db=edselc&AN=edselc.2-52.0-84865684259&site=eds-live&scope=site.

Rawlings, J. and D. Mayne (2009). *Model Predictive Control: Theory and Design*. Ed. by C. Rawlings. Nob Hill Publishing, 18 Rough Lee Court, Madison, WI 53705.

Runvik, H. (2014). *Modelling and start-up optimization of a coal-fired power plant*. Master's Thesis ISRN LUTFD2/TFRT--5950--SE. Department of Automatic Control, Lund University, Sweden.

SciPy. http://www.scipy.org/. Accessed: 2015-02-03.

Sorenson, H. (1980). *Parameter estimation: Principles and problems*. Control and systems theory. M. Dekker. ISBN: 9780824769871.

Soroush, M. (1998). "State and parameter estimations and their applications in process control". *Computers & Chemical Engineering* **23**, pp. 229–245.

Tenny, M. and J. Rawlings (2002). "Efficient moving horizon estimation and nonlinear model predictive control". In: *American Control Conference, 2002. Proceedings of the 2002*. Vol. 6, 4475–4480 vol.6. DOI: 10.1109/ACC.2002.1025355.

The FMI standard. https://www.fmi-standard.org/. Accessed: 2015-02-03.

Ungarala, S. (2009). "Computing arrival cost parameters in moving horizon estimation using sampling based filters". *Journal of Process Control* **19**:9, pp. 1576 –1588. ISSN: 0959-1524. DOI: http://dx.doi.org/10.1016/j.jprocont.2009.08.002. URL: http://www.sciencedirect.com/science/article/pii/S0959152409001383.

Wächter, A. and L. Biegler (2006). "On the implementation of an interior-point filter line-search algorithm for large-scale nonlinear programming". English. *Mathematical Programming* **106**:1, pp. 25–57. ISSN: 0025-5610. DOI: 10.1007/s10107-004-0559-y. URL: http://dx.doi.org/10.1007/s10107-004-0559-y.

Wilson, D., M. Agarwal, and D. Rippin (1998). "Experiences implementing the extended Kalman filter on an industrial batch reactor". *Computers & Chemical Engineering* **22**:11, pp. 1653 –1672. ISSN: 0098-1354. DOI: http://dx.doi.org/10.1016/S0098-1354(98)00226-9. URL: http://www.sciencedirect.com/science/article/pii/S0098135498002269.

Wisconsin-Madison, U. of. *Rawlings Group*. http://jbrwww.che.wisc.edu/index.html. Accessed: 2015-02-03.

Zavala, V., C. Laird, and L. Biegler (2008). "A fast moving horizon estimation algorithm based on nonlinear programming sensitivity". *Journal of Process Control* **18**:9. Selected Papers From Two Joint Conferences: 8th International Symposium on Dynamics and Control of Process Systems and the 10th Conference Applications in Biotechnology 8th International Symposium on Dynamics and Control of Process Systems and the 10th Conference Applications in Biotechnology, pp. 876 –884. ISSN: 0959-1524. DOI: http://dx.doi.org/10.1016/j.jprocont.2008.06.003. URL: http://www.sciencedirect.com/science/article/pii/S0959152408001091.

Lund University
Department of Automatic Control
Box 118
SE-221 00 Lund Sweden

Document name
MASTER´S THESIS

Date of issue
June 2015

Document Number
ISRN LUTFD2/TFRT--5982--SE

Author(s)
Tor Larsson

Supervisor
Toivo Henningsson, Modelon AB
Björn Olofsson, Dept. of Automatic Control, Lund University, Sweden
Karl-Erik Årzén, Dept. of Automatic Control, Lund University, Sweden (examiner)

Sponsoring organization

Title and subtitle

Moving Horizon Estimation for JModelica.org

Abstract

In this thesis a Moving Horizon Estimator (MHE) has been implemented for the JModelica.org platform. JModelica.org is an open-source software platform for simulation and optimization of systems described in the modeling language Modelica. MHE is an optimization-based strategy for state estimation where, at each time step, a finite horizon optimization problem is solved to generate an estimate of the current state values. The goal has been to implement an MHE that works with many already existing Modelica models and that has an intuitive user interface. The performance of the implemented MHE is evaluated using both linear and nonlinear systems in a series of simulation examples. The results indicate that the MHE performs well.

Keywords

Classification system and/or index terms (if any)

Supplementary bibliographical information

ISSN and key title
0280-5316

ISBN

Language
English

Number of pages
1-69

Recipient's notes

Security classification

http://www.control.lth.se/publications/