Nonlinear Model Predictive Control for Combined Cycle Power Plants

Anna Johnsson

LUND UNIVERSITY

MASTERS THESIS in Automatic Control 2013

Supervisor: Stéphane Velut, Modelon
Killian Link, Siemens

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

ISSN 0280-5316
ISRN LUTFD2/TFRT--5926--SE

Acknowledgements

A big *danke schön* to the team at Siemens AG Energy Sector in Erlangen Germany. It has been a pleasure working with you! And a special thank you to Karin Dietl and Stephanie Gallardo Yances, for all valuable discussions and for your help and support.

I would also like to thank my supervisor at Modelon, Stéphane Velut, for his help and support.

Abstract

This master thesis project serves to investigate the possibilities of Nonlinear Model Predictive Control (NMPC) using the example of enthalpy control of the BENSON HRSG (heat recovery steam generator) of a combined cycle power plant (CCPP).

The general idea of NMPC is to solve an optimization problem, to find the next control action, and this optimization problem is based on a model of the system. The models used in the controller implementation are Modelica-based, and the system is described by algebraic differential equations (DAEs).

The controller was implemented in the Python interface of JModelica.org (Modelica-based modeling tool, supporting the Modelica extension Optimica for optimization), together with an extended Kalman filter (EKF) for state estimation.

The control algorithm was only evaluated for a setup where the controller model is very similar to the model representing the real process; both models are simplified representations of the real process.

Keywords:

Optimization, Nonlinear Model Predictive Control, Extended Kalman filter, Modelica, Optimica, JModelica.org

Contents

Acronyms

CCPP	Combined Cycle Power Plant
DAE	Differential Algebraic Equation
EKF	Extended Kalman Filter
FMI	Functional Mock-up Interface
FMU	Functional Mock-up Unit
JMI	JModelica Model Interface
JMU	JModelica Model Unit
NLP	Nonlinear Program
NMPC	Nonlinear Model Predictive Control
MHE	Moving Horizon Estimation
MPC	Model Predicitve Control
ODE	Ordinary Differential Equation

Overview of Chapters

- Chapter 1 contains the aim of the thesis.
- Chapter 2 describes the process and the control objective.
- Chapter 3 contains the basic theoretical concepts behind the implemented control algorithm.
- Chapter 4 and 5 describes the model objects and their functionallity.
- Chapter 6 describes the implementation.
- Chapter 7 contains results for some test cases.
- Chapter 8 contains some concluding comments about the implementation and the results and also ideas and suggestions for future steps to improve the control algorithm.

Chapter 1

Introduction

This master thesis project was carried out in cooperation with Siemens AG Energy Sector in Erlangen, Germany and Modelon AB in Lund, Sweden, within the ITEA2 project MODRIO. The work was founded by the German Ministry BMBF and by the Swedish government agency VINNOVA.

1.1 Aim of thesis

The aim of this master thesis project is to implement a temperature controller for the steam leaving the superheater section of a heat recovery steam generator (HRSG), a part of a combined cycle power plant (CCPP). The controller is to be implemented in the Python interface of JModelica.org.

This report serves as a starting point for the investigation of the possibilities to, in the future, replace the current plant controller with a more general controller which is more accurate and easier to evaluate, investigate and modify. The control algorithm that is currently in use is not available, but it is safe to say that it is quite complex, since the process itself is quite complex. For example, the controller has to consider behaviors at medium transitions, which are highly nonlinear, and long delays. The process is well-known, it is possible to model well and nonlinear model predictive control (NMPC) is therefore investigated as potential control strategy in this project.

NMPC is an advanced control strategy that can handle constraints and nonlinear multi-input-multi-output systems. A model representation of the system is central for the control algorihm. The model is used to predict the behavior of the process, and the predictions are then used to find the next suitable control action, by solving an optimization problem.

The platform JModelica.org is suitable for the implementation because of its modeling, optimization and simulation possibilities, which are all central concepts for the algorithm.

The NMPC-algorithm is combined with an extended Kalman filter (EKF) for state estimation. In spite of its simplicity, the EKF works sufficiently well (at least as a starting point) and is easily implemented in JModelica.org.

1.2 Siemens AG Energy sector

The Siemens Energy Sector is one of the the world's leading supplier of products, services and solutions for power generation. Siemens provides a great insight and knowledge of the process related to this project, which is highly relevant when developing and working with the corresponding models [10].

1.3 Modelon AB

Modelon specializes in providing solutions, services and technology for the research and development of dynamic systems. They offer unique know-how in physical modeling, simulation and optimization, and model-based control design. Modelon also maintains and develops JModelica.org (the software used for implementation and optimization in this project) [11].

1.4 MODRIO

The European reasearch project MODRIO extends state-of-the-art modeling and simulation environments based on open standards to increase energy and transportation systems safety, dependability and performance. The MODRIO project runs from September 2012 to November 2015 [12].

1.5 Tools

1.5.1 The Modelica language

Modelica is a unified object-oriented language for systems modeling. It is suitable for modeling of large, complex and dynamic systems, and it is also suitable for multi-domain modeling. Another advantage with the Modelica language is the possibility of extension, a necessary feature for the purpose of optimization. JModelica.org supports the extension Optimica, which enables high-level formulation of optimization problems based on Modelica models. For more information about Modelica, Optimica and the Modelica Association, see [13].

1.5.2 JModelica.org

JModelica.org is an extensible Modelica-based open source platform for optimization, simulation and analysis of complex dynamic systems. It is a very neat tool for the purposes of this project. It is possible to use for modeling as well as implementation of the control algorithm, using the Python interface [9].

It will be more clear exactly how the features of JModelica.org are used in the implementation section, see Chapter 6. For more information about JModelica.org, see [9].

1.5.3 Dymola

Dymola is a commercial modeling and simulation Modelica-based environment For this project, Dymola is used for development of the Modelica models. The models are developed by Siemens AG, and adjusted during the course of the project to better fit the problem at hand. For more information about Dymola, see [16].

1.5.4 Dynaplant

Dynaplant is an in-house tool of Siemens which is used for detailed modeling of the process. This tool is not Modelica-based.

Chapter 2

The CCPP process

2.1 General concepts of a combined cycle power plant

The combined cycle power plant (CCPP) considered in this report is a combination of a gas turbine and a steam turbine, with a water/steam cycle. The waste heat generated by the gas turbine is used to produce the steam in the heat recovery steam generator (HRSG), to run the steam turbine. This approach results in a higher efficiency than for a one-cycle plant (i.e without the water/steam cycle). The plant is briefly described in the following sections [1].

A simple flow diagram of a plant is displayed in Figure 2.1 were the key components are marked. The part of interest for this project is the HRSG.

2.1.1 The heat recovery steam generator

The HRSG considered in this project is a BENSON HRSG, which is constructed under Siemens license. The main components of a HRSG are: the economizer, the evaporator and the superheater. The economizer works as a preheater of the inlet water. The evaporator converts the water to steam and the superheater raises the temperature of the steam beyond the saturation point.

The BENSON HRSG is a once-through HRSG, this approach does not require a steam drum for phase separation (steam/water), which is the case for the standard implementation. One can describe the once-through implementation as a single tube where water enters, travels through a chain of economizers, evaporators and superheaters, to finally leave as steam at the other end. The transition from water to steam occurs in the evaporator, but it is a floating point transition and may vary. See Figure 2.2 [1].

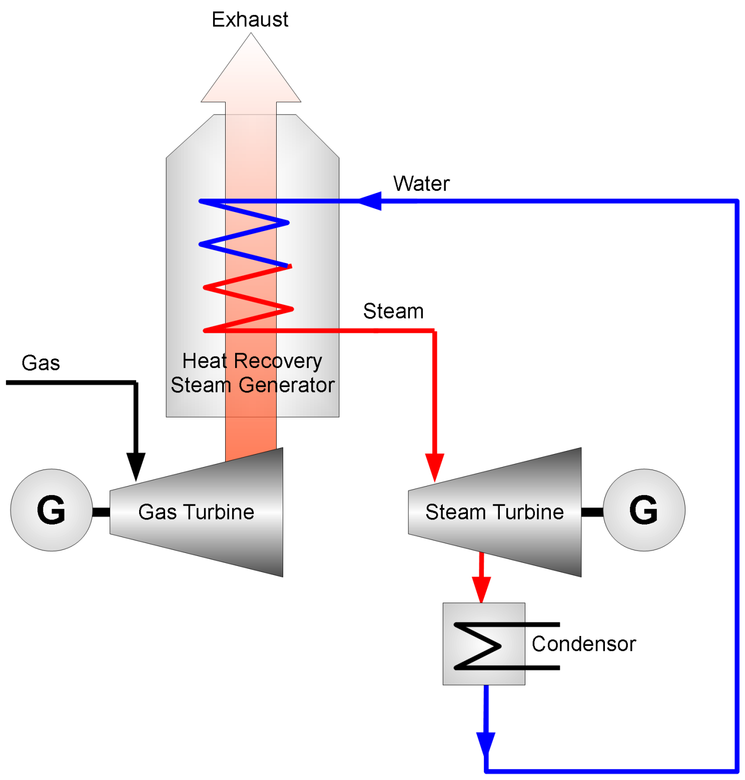

Figure 2.1: Overview of a combined cycle power plant.

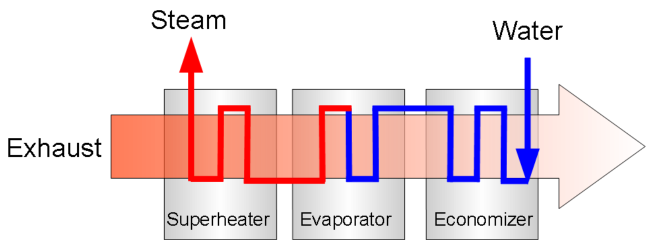

Figure 2.2: The HRSG.

2.2 Control objective

The main objective of the controller is to reach a specified enthalpy setpoint at the outlet of the superheater section. The setpoint is represented by a temperature, since the pressure is known, and is therefore referred to as a temperature controller. The setpoint is selected to ensure an adequate degree of superheating at the outlet.

There is also an additional objective for the controller: to avoid that the evaporations starts in the economizer, i.e that there is subcooling (water) at the outlet of the economizer-section. The subcooling is ensured by keeping the temperature at the evaporator inlet below a pressure dependent maximum value.

This is important in order to ensure the stablibity of the BENSON evaporator. If there is a steam/water mix at the inlet of the first evaporator it may result in an inhomogeneous steam/water mix in the evaporator piping. This can lead to temperatures that are too high for the evaporator components to handle. There is a distribution-component added to evaporators where a steam/water mix is expected at the inlet, but there is no such component for the first evaporator, i.e, the control objective is a result of the process design.

To summarize, there are three objectives:

1. Keep the temperature at the superheater outlet at desired setpoint.
2. Have an adequate degree of superheating at the outlet of the evaporator section.
3. Avoid subcooling by keeping the temperature at the evaporator outlet below a specified maximum value.

There are two control variables available to achieve this; the mass flow at the economizer inlet and bypassing the economizer section. The main control signal is the water mass flow. The bypass controller is mainly used to help ensure the subcooling. The valve should only be opened if the maximum temperature is violated.

2.3 Modelling scope

The process model used by the controller is a simplified representation of a part of the plant, and does not consider all plant dynamics. In order to be able to consider the process as a open-loop process, some boundary conditions are included, representing for example pressures. Some of the boundary conditions are time dependent, and need to be updated, and some are represented as constants.

The complete HRSG is more complicated than described in Section 2.1, and also includes three steam turbines with water/steam cycles corresponding to three different pressure levels. The BENSON HRSG only represents the high pressure cycle; there is also an intermediate pressure cycle and a low pressure cycle. The model developed for the purpose of this project mainly focuses on the high pressure cycle and the related components.

The effects of the low pressure cycle have no significant impact on the controller task and are therefore ignored. The effects of the intermediate pressure cycle are modeled as heat sinks, in order to consider the heat loss from the flue gas to the intermediate cycle.

The economizers in the high pressure cycle are described through an analytical approach, because it will reduce the number of equations, i.e less states are needed to represent the behavior. There does not exist an analytical description of the evaporators, and they are therefore described with discrete elements. This is necessary in order to monitor the floating phase transition point.

The superheater of the high pressure cycle is modeled as a heat sink (steady state model, i.e no delay for changes)

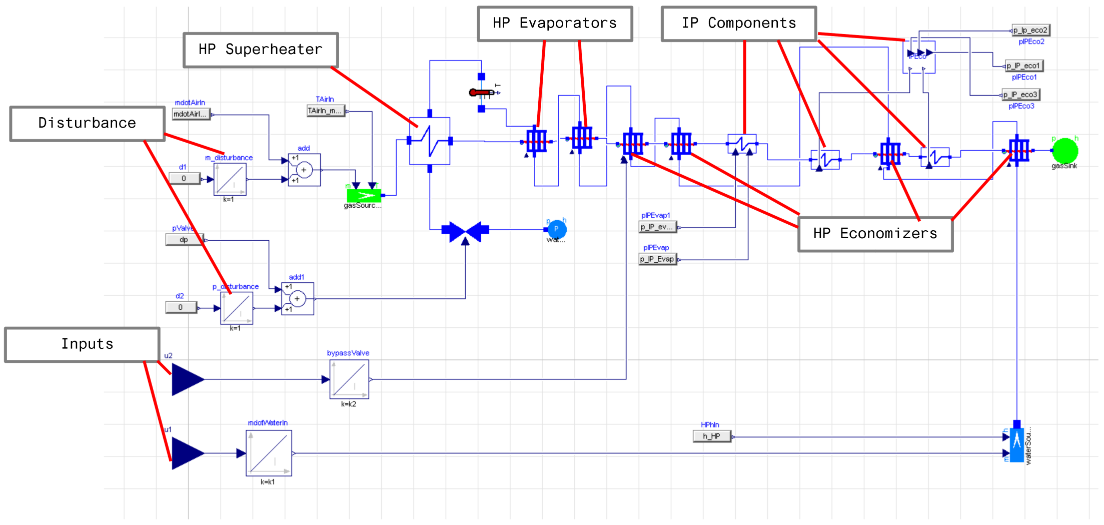

Figure 2.3: The model: Screen shot of the model representation in Dymola

Figure 2.3 shows how the model is represented in Dymola and displays the different parts. Also visible in the figure are the two inputs and two expected disturbances.

Chapter 3

Theory

This chapter contains some of the basics for understanding the concepts behind the implemented controller algorithm, it covers: NMPC, the EKF and dynamic optimization. The section about dynamic optimization focuses only on the concepts behind the approach that is relevant for the implementation: direct collocation and interior point optimization.

3.1 Nonlinear model predictive control

Model predictive control (MPC) is based on predicting the plant behavior using a model of the process, and to then determine the next control action to apply to the process by solving an optimization problem. Nonlinear model predictive control (NMPC) is an extension of MPC for nonlinear processes, but the basics are the same.

The optimization problem generally includes the control objective in the cost function (the desired behavior) and uses the system equations as constraints, in which the system behavior and the physical constraints are included. The process investigated in this project is nonlinear, hence NMPC. A good model of the system that is to be controlled is key in order to apply NMPC on a real process, and the control action computed is only meaningful if the model is good enough.

The basic algorithm described below is in generally not sufficient. The model used by the controller is not a perfect match to the real system, and it is therefore necessary to include some sort of state estimation in the algorithm, in order to cope with modelling errors, disturbances and unmeasurable states.

Two good references for MPC are [3] and [2], which also covers NMPC.

3.1.1 Basic algorithm

NMPC is an iterative process:

1. A optimization problem is solved to find the optimal control sequence. It is solved over a finite time horizon, called the prediction horizon H_p. The objective is to minimize a predefined cost function (i.e satisfy the control objective) without violating any of the constraints, with the current state values as starting point.
2. The first element of this optimal control sequence is then applied to the process.
3. The current states are updated according to measurements of the process, and then the procedure is repeated from step 1.

The optimization problem is open-loop and the feedback property is introduced by regularly updating the starting point of the optimization problem with measurements of the process states.

3.1.2 The optimization problem

The discrete time optimization problem is generally formulated as (the subscript k is used to indicate the time step):

$$\min_{u} \; V_{H_p}(x_0, u)$$

$$\begin{aligned} \text{Subject to:} \quad & x_{k+1} = f(x_k, u_k) \\ & x, x_0 \in \mathbb{X} \\ & u \in \mathbb{U} \end{aligned}$$

$f(.,.)$	The model of the process, assumed to be twice continuously differentiable.
x_0	The intitial state values.
x	The state vector.
u	The input vector.
$\mathbb{X}$	The state constraints, a closed set, i.e, all boundaries are included in the set.
$\mathbb{U}$	The input constraints, a compact set, that ensures that u_k does not "produce" an infeasible x_{k+1}.
$V_{H_p}(.,.)$	The cost function for the finite time interval H_p.

The feedback property is included through x_0, which is the latest plant state measurements.

The cost function is defined as:

$$V_{H_p}(x_0, u) = \sum_{i=0}^{H_p-1} l(x_i, u_i) + V_f(x_{H_p})$$

The cost $V_{H_p}(.)$ is divided in two parts; the stage cost $l(.,.)$ and the terminal cost $V_f(.)$. The stage cost includes the overall control objective and the control input. The input is included in order to ensure (or at least to help ensuring) uniqueness of the solution, although, the contribution may be very small [2].

The terminal cost is especially central in the discussion concerning stability assurance, see Section 3.1.4.

The prediction horizon (H_p) determines how far ahead in time the optimizer should observe the behavior of the system for different possible inputs. It is common to also use a control horizon (H_c) or blocking factors in order to limit the number of possible control actions for the optimizer to consider, in practice they specify for what intervals of H_p the input sequence must be kept constant. Also note that the algorithm is applied on a sampled system and that the time step has an influence, it determines how often the measurements are updated.

There is a trade off between the size of the problem and the "level of optimality" of the result (assuming that a perfect model of the real process is used by the controller). A longer H_p and larger set of possible inputs result in a larger problem, which will require a larger computational effort but will (if solved) result in a "better" control action. These parameters should be adjusted in order to fit the problem at hand and it is important to consider the process dynamics when doing so.

3.1.3 Properties of NMPC

NMPC is very useful in the sense that it is not necessary to determine the explicit control law (e.g $u = Kx$, for some gain K), since the control action that is to be applied is computed numerically at every iteration. The controller can therefore consider constraints and handle nonlinear processes (as long as the optimization problem is converging).

There is an important issue to consider when one is working with nonlinear models, the optimal control problem is generally non-convex, i.e there is no global solution to the optimization problem. However, there are usually solutions to the problem which are sub-optimal, but the optimizer has to be provided with a good initial guess to find these.

In NMPC, it is natural to use the previously computed control sequence as initial guess for the next iteration. Although, the first initital guess has to be constructed somehow, and with care. The strategy used in the project implementation is described in Section 6.4.1.

3.1.4 Stability - a stabilizing controller

Stability of the controller is generally ensured through Lyapunov theory, by defining a terminal cost function as a Lyapunov function. However, there are also discussions about how one can assume stability of the controller if the prediction horizon is long enough. This may seem a bit vague, but it is useful while implementing the algorithm, since it can be quite tricky to define a terminal cost function in the case of a nonlinear system with constraints. Nevertheless, according to [2] there is nothing to lose in introducing a terminal cost to the nonlinear, constrained, case. The complexity remains at the same level, but the convergence of the optimization problem is most likely to improve.

The terminal cost is generally implemented in such a way that the final state values are in a desired set. This to make sure that the optimal control action, for the given horizon, for example does not move the process operating point to a state where the process is uncontrollable.

3.1.5 Lyapunov theory

This section presents the basics of Lyapunov theory. See [2] for further details.

For a system $x_{k+1} = f(x_k, u)$, a Lyapunov function (the terminal cost function) must satisfy:

$$\begin{aligned} V(x) &\geq \alpha_1(|x|) \\ V(x) &\leq \alpha_2(|x|) \\ V(f(x,u)) &\leq V(x) - \alpha_3(|x|) \end{aligned}$$

$$x \in \mathbb{X}$$

$V(x)$	Lyapunov function.
$f(.)$	The system.
x	The state vector.
u	The optimal input.
$\mathbb{X}$	The state constraints, positive invariant.
$\alpha_1(.) - \alpha_2(.)$	$\mathcal{K}_\infty$ functions (continuous, strictly increasing zero at zero, and unbounded).
$\alpha_3(.)$	Positive definite function.

The controller is stable if the cost function is sufficiently decreasing at every time step. It may be so that there does not exist a global Lyapunov function, [2] discusses how to treat this as well. There is no general rule of thumb of how to pick the terminal cost, the variety of problems is too wide and ad hoc solutions are required for most problems.

3.1.6 State estimation

State estimation is used to reconstruct the states of a system, using the process measurements and a model of the plant. Since the model used is a simplified representation of the real plant, and because in reality there are unknown disturbances effecting the measurements, we know for a fact that the model does not exactly correspond to the real plant. It is therefore necessary to introduce a state observer to the algorithm, and instead of using the pure measurements x use the estimated $\hat{x}$ as the initial state values for the optimization problem.

When talking about state estimation for nonlinear systems two approaches are often mentioned; the extended Kalman filter (EKF, used in this project) and moving horizon estimation (MHE), and the suitable approach may differ depending on the application. The EKF is easier to implement and requires less computational effort, but the MHE generally outperforms the EKF with greater robustness. MHE also considers the constraints, which is not the case of EKF [5].

In order to use the estimates from the EKF, it is necessary to make sure that the states are feasible, i.e, that the estimated states does not violate the constraints before sending them to the NMPC-controller. This is discussed in more detail in Section 3.2.

MHE is an iterative approach which involves solving a constrained optimization problem much like the one solved in the optimization step of the NMPC (also with possible issues related to the nonlinearity of the system). For a linear system, without constraints, MHE corresponds to the Kalman filter. For more information about MHE see [5].

3.1.7 Observablilty for nonlinear systems

Observability is necessary for state estimation, however currently, there does not exist such a test that is easy to implement for nonlinear systems. One approach, mentioned in [5], suggests to check the condition number of the time varying Gramian of a linear time-varying approximation of the nonlinear system. If the Gramian is ill-conditioned, it generally inidicates poor observablilty of the system.

The EKF, implemented as state estimator in this project, includes a linearization of the system at every time step. It is easy to compute the observablilty matrix O with the linearized A and C, and it is possible to perform a test similar to the mentioned above, just to get a indication of the observability.

Observability test

A linear system:

$$\begin{aligned} \dot{x} &= Ax + Bu \\ y &= Cx + Du \\ x &\in \mathbb{R}^n \end{aligned}$$

is observable if and only if the observability matrix O satisfies:

$$\text{rank}(O) = n$$

$$O = \begin{bmatrix} C \\ CA \\ CA^2 \\ \vdots \\ CA^{n-1} \end{bmatrix}$$

This simple test was evaluated in the implementation. A state was introduced to estimate a constant disturbance for a time varying parameter. The simple test indicated that this state was unobservable, and after further investigation it turned out that changes and simplification of the model had made this parameter insignificant for the system behaviour. This does not prove that the simple test is sufficient, but at least that it can be used for debugging purposes.

3.2 The extended Kalman filter

The Kalman filter originates from probability theory and it is well established that the Kalman filter is the optimal state estimator for a linear system effected by white noise [6]. The Kalman filter minimizes the estimation error by considering past data of the system. This can be described in a recursive way which is convenient for implementation purposes, transforming the estimation to an estimation update.

The estimate update consists of two main steps; the prediction and the correction. In the prediction step, the next set of states is calculated from the system representation and also, the covariance matrix of the estimation error at the prediction step is updated (remember that the filter stems from probability theory).

In the correction step, the Kalman gain is updated. It is then used to correct the state estimation made in the prediction step, by also considering the latest plant measurements. The covariance matrix for the estimation error at the correction step is also updated here.

The EKF is an extension of the Kalman filter for nonlinear process models and the approach is basically the same as in the linear case, with an additional linearization to get approximations of the A_k and C_k matrices (using standard notation for linear systems), which are used by the filter. The linearization is done at each time step k. The EKF does not consider constraints, and this has to be compensated for in an additional step (the feasibility correction). The setup of the EKF is as displayed in Figure 3.1.

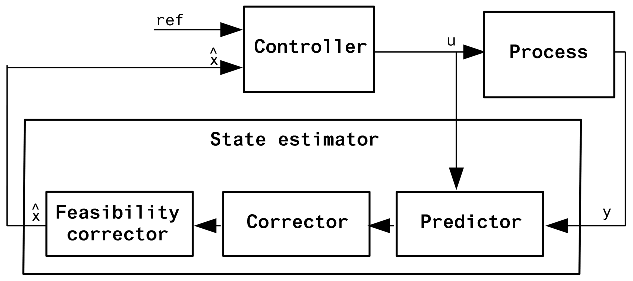

Figure 3.1: Setup for the state estimation, $\hat{x}$ represent the state estimate.

Since the EKF simply is an extended version of the Kalman filter, first the Kalman filter for a linear system is described, and then finally the EKF. See [6] and [5] for more information about the extended Kalman filter.

3.2.1 The Kalman filter - Linear system

For a linear system (discrete time, the subscript k is used to indicate the time step)

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

x	The state vector.
u	The input vector.
y	The measurement vector.
w	The process noise.
v	The measurement noise.

The covariance matrices of the noise are defined as (assuming zero mean white noise):

Process noise covariance Q_k (uncorrelated in time):

$$\mathbb{E}\{w_k w_l^T\} = Q_k \delta_{k,l}$$

$$\delta_{k,l} = 1 \text{ if } k = l, \text{ 0 otherwise.}$$

Measurement noise covariance R_k (uncorrelated in time):

$$\mathbb{E}\{v_k v_l^T\} = R_k \delta_{k,l}$$

Measurement and process noise are assumed to be uncorrelated:

$$\mathbb{E}\{w_k v_l^T\} = 0$$

The Kalman filter requires some extensive notation declaration:

x_0	The initial guess for the state estimates.
P_0	The initial guess for the covariance of the estimation error.
$P_{k\|k}$	The covariance matrix of the estimation error at the correction step.
$P_{k+1\|k}$	The covariance matrix of the estimation error at the prediction step.
K_k	The Kalman gain.
$\hat{x}_{k\|k}$	The corrected state estimates.
$\hat{x}_{k+1\|k}$	The predicted state estimates.

For the setup described, the Kalman filter is defined as:

$$\begin{array}{lll}
\text{Initialization:} & \hat{x}_{0|0} & := x_0 \\
 & P_{0|0} & := P_0 \\
 & & \\
\text{Prediction step:} & \hat{x}_{k+1|k} & = A_k \hat{x}_{k|k} + B u_k \\
 & P_{k+1|k} & = A_k P_{k|k} A_k^T + Q_k \\
 & & \\
\text{Correction step:} & K_{k+1} & = P_{k+1|k} C_{k+1}^T (C_{k+1} P_{k+1|k} C_{k+1}^T + R_{k+1})^{-1} \\
 & \hat{x}_{k+1|k+1} & = \hat{x}_{k+1|k} + K_{k+1}(y_{k+1} - C_{k+1}\hat{x}_{k+1|k}) \\
 & P_{k+1|k+1} & = P_{k+1|k} - K_{k+1} C_{k+1} P_{k+1|k}
\end{array}$$

Figure 3.2: The Kalman filter for a linear system.

3.2.2 The extended Kalman filter - Nonlinear system

For the nonlinear case (discrete time, the subscript k is used to indicate the time step)

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

The system may (assuming that f_k and h_k are smooth) be approximated as:

$$\begin{aligned} f_k(x_k, u_k) &\approx f_k(\hat{x}_{k|k}, u_k) + A_k(\underbrace{x_k - \hat{x}_{k|k}}_{error}) + w_k \\ h_k(x_k) &\approx h_k(\hat{x}_{k|k-1}) + C_k(\underbrace{x_k - \hat{x}_{k|k-1}}_{error}) + v_k \end{aligned}$$

where A_k and C_k are determined from linearization [6]:

$$A_k = \frac{\partial}{\partial x} f_k(x,u)\bigg|_{x=\hat{x}_{k|k}, u=u_k} \qquad C_k = \frac{\partial}{\partial x} h_k(x)\bigg|_{x=\hat{x}_{k|k-1}}$$

With the Kalman filter in mind, the extended Kalman filter is defined as:

$$
\begin{array}{lll}
\text{Initialization:} & \hat{x}_{0|-1} & := x_0 \\
 & P_{0|-1} & := P_0 \\
 & & \\
\text{Prediction step:} & A_k & = \frac{\partial}{\partial x} f_k(x,u)\big|_{x=\hat{x}_{k|k},u=u_k} \\
 & \hat{x}_{k+1|k} & = f(\hat{x}_{k|k}, u_{k+1}) \\
 & P_{k+1|k} & = A_k P_{k|k} A_k^T + Q_k \\
 & & \\
\text{Correction step:} & C_{k+1} & = \frac{\partial}{\partial x} h_k(x)\big|_{x=\hat{x}_{k+1|k}} \\
 & K_{k+1} & = P_{k+1|k} C_{k+1}^T (C_{k+1} P_{k+1|k} C_{k+1}^T + R_{k+1})^{-1} \\
 & \hat{x}_{k+1|k+1} & = \hat{x}_{k+1|k} + K_{k+1}(y_{k+1} - h_{k+1}(\hat{x}_{k+1|k})) \\
 & P_{k+1|k+1} & = P_{k+1|k} - K_{k+1} C_{k+1} P_{k+1|k}
\end{array}
$$

Figure 3.3: The extended Kalman filter.

where:

x_0	The initial guess for the state estimates.
P_0	The initial guess for the covariance of the estimation error.
$P_{k\|k}$	An approximation of the covariance matrix of the estimation error at the correction step.
$P_{k+1\|k}$	An approximation of the covariance matrix of the estimation error at the prediction step.
K_k	The Kalman gain.
R_k	The covariance matrix of the measurement noise.
Q_k	The covariance matrix of the process noise.
A_k	The A-matrix from the linearized system representation.
C_k	The C-matrix from the linearized system representation.
$f_k(.)$	The f_k from the system representation.
$h_k(.)$	The h_k from the system representation.
$\hat{x}_{k\|k}$	The corrected state estimates.
$\hat{x}_{k+1\|k}$	The predicted state estimates.
y_k	The process measurements.
u_k	The applied control action.

3.2.3 Adding disturbance states

Disturbance states d are added in order to compensate for modeling errors and unknown constant disturbances; uncertainties that may result in offsets. They are introduced by augmenting the system, and their values are also estimated by the EKF. These are unmeasurable states of the real plant.

$$\begin{bmatrix} x_{k+1} \\ d_{k+1} \end{bmatrix} = \begin{bmatrix} f(x_k, u_k) \\ d_k \end{bmatrix}$$

By defining the disturbance states like this, integral action is introduced. The general idea is to introduce a disturbance state for every measurement. This might, however, result in a lot of introduced states for large models, which result in a greater task for the estimator. Although some reduction might be possible, a first step is to investigate the impact a constant disturbance has on the control objective at each (state) measurement.

3.2.4 Handling infeasible state estimates

The main drawback with the EKF is its incapability to consider the system constraints, so what if these are violated, what if the estimator estimates a valve to be opened more than 100 %, or something else that is not feasible?

The infeasibility is also a problem for the optimizer, which is an interior point method. If the constraints already are violated when starting the optimization, no solution will be found. Some sort of ad hoc solution is necessary in order to handle this. In this case the process is not that sensitive to disturbances, and so, if a constraint is violated as a result of the correction step, then the value of the prediction step will be used instead, since the prediction considers the constraints.

An other possibility for handling the infeasibility could for example be to "move" the state estimates to the closest feasible set in the least square sense.

3.3 Dynamic optimization

Finding a solution to the optimization problem is not trivial, especially not for large, complex models, as indicated in the stability section. The strategy used in this project is the default strategy used for optimization in JModelica.org. The strategy involves direct collocation to transform the problem in to a nonlinear program (NLP), which is then solved by an interior point optimization algorithm (IPOPT). This approach requires that the model is described by a set of differential algebraic equations (DAEs) and that it does not contain any discontinuities. JModelica.org supports other approaches for dynamic optimization as well, see the JModelica.org User Guide [9] for more information.

3.3.1 Direct collocation

In direct collocation algorithms, an infinite-dimensional problem is approximated as a finite-dimensional nonlinear program (NLP). The key parameters for this approximation are:

H_p	The prediction horizon of the optimization problem
n_e	The number of finite elements
n_{cp}	The number of collocation points

The time horizon H_p is first divided into n_e elements, and within each element each time dependent variable is approximated as a polynomial of degree n_{cp}, called a collocation polynomial. The collocation polynomials are formed by using Lagrange interpolation polynomials, using the collocation points as interpolation points.

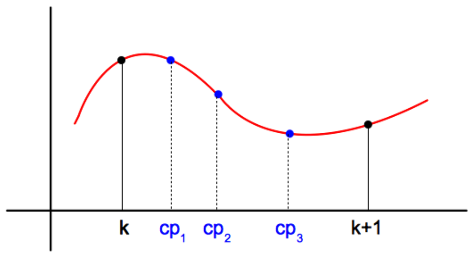

Figure 3.4: Collocation, with `n_cp` = 3.

There are different ways of choosing the location of the collocation points. The default for the optimization method in JModelica.org is Radau collocation, which always places a collocation point at the end of each element, to ensure that the states are continuous at the element boundaries, and the rest are placed to maximize the accuracy.

The size of the NLP is defined by the choice of n_e and n_{cp}, in relation to the model equations of course. For more details concerning the collocation step see [7].

3.3.2 Interior point optimization with IPOPT

The details of IPOPT are obtained from [4]. The details are quite tedious, but it is necessary to understand the basics behind the solver and to be able to analyze the optimization results, especially if the optimization is not converging.

Problem formulation

The IPOPT algorithm transforms the NLP internally, and replaces all inequality constraint with equality constrains by introducing additional slack variables. The optimization variable boundaries are also included in the constraints, yielding the representation:

$$\min_{x} \; f(x) \tag{3.1}$$

$$\text{Subject to:} \quad c(x) = 0 \tag{3.2}$$

$$x \leq 0 \tag{3.3}$$

$f(x)$	The objective function, the cost function.
x	The optimization variables.
$c(x)$	The set of equality constraints.

As an interior point method, IPOPT considers the auxiliary barrier problem formulation:

$$\min_{x} \Phi_{\mu}(x) = f(x) - \mu \underbrace{\sum_{i=1}^{n} \ln(x_i)}_{\textit{barrier term}} \tag{3.4}$$

$$\text{Subject to:} \quad c(x) = 0 \tag{3.5}$$

$\Phi_{\mu}(x)$	The barrier function.
μ	The barrier parameter, $\mu > 0$.
n	The number of variables.

The barrier function $\phi_\mu(x)$ is introduced to prevent the line search from leaving the feasible region, and it also replaces the bound: $x \leq 0$, since $\phi_\mu(x)$ goes to infinity as x_i approaches zero. The optimal solution of this problem will therefore be an interior point, and hence the name.

The influence of the barrier term depends on the size of μ. As μ approaches zero, the barrier problem approaches the original problem. The general strategy is to start solving the barrier problem with a moderate value of μ, say 0.1, and then continue solving the same problem for a smaller μ, i.e, for a higher level of accuracy, until a optimal solution according to the user specified tolerance is found.

The optimality conditions (the Karush-Kuhn-Tucker (KKT) conditions) for the barrier problem are defined as:

$$\nabla f(x) + \nabla c(x)y - z = 0 \tag{3.6}$$

$$c(x) = 0 \tag{3.7}$$

$$XZe - \mu e = 0 \tag{3.8}$$

$$x, z \geq 0 \tag{3.9}$$

$\nabla f(x)$	The gradient of the objective function.
$\nabla c(x)$	The gradient of the constraints.
y	The Lagrange multipliers for the equality constraints.
z	The Lagrange multipliers for the bound constraints.
Z	diag(z)
X	diag(x)
e	$[1, ..., 1]^T$

The Lagrange mulipliers are such that they represent the conditions for a stationary point, i.e, if x is a minima, then there exists a y and a z such that Equations 3.6-3.9 are fulfilled.

Solving the barrier problem

A Newton-type algortihm is used for finding the solution to the barrier problem. This is done for a fixed μ and when eventually a optimal solution is found, μ is increased and the problem is solved once again, this is repeated until a satisfying solution is found.

Solving for a fixed μ:

The new iterates are found by applying the Newton-algorithm to the system represented by Equations 3.6-3.8, this to generate a converging sequence of iterates that always strictly satisfy Equation 3.9.

Note! Not only minimizers satisfy these conditions, saddle points and maximizers may do as well. IPOPT has a strategy for considering this fact, Hessian regulation. Hessian regulation includes the introduction of the perturbation δ_x (see Equation 3.10), which encourages the algorithm to avoid those solutions.

Given an iterate (x_k, y_k, z_k) and assuming $x_k, z_k \geq 0$, the Newton step (Δx_k, Δy_k,Δz_k) is computed from:

$$\begin{bmatrix} W_k + X_k^{-1} Z_k + \delta_x I & \Delta c(x_k) \\ \Delta c(x_k)^T & 0 \end{bmatrix} \begin{bmatrix} \Delta x_k \\ \Delta y_k \end{bmatrix} = - \begin{bmatrix} \nabla \Phi(x_k) + \nabla c(x_k) y_k \\ c(x_k) \end{bmatrix} \tag{3.10}$$

$$\Delta z_k = \mu X_k^{-1} e - z_k X_k^{-1} Z_k \Delta x_k \tag{3.11}$$

W_k	is the Hessian of the Lagrangian function, defined below.
$\delta_x I$	the perturbation of W_k.

The Hessian (W_k) of the Lagrangian function: $\nabla f(x) + \nabla c(x) y - z$, see Equation 3.6, is:

$$W_k = \nabla^2 f(x_k) + \sum_{j=1}^{m} y_k^{(j)} \nabla^2 c^{(j)}(x_k) \tag{3.12}$$

m	represents the number of equality constraints for the barrier problem.

$W_k > 0$ indicates a local minimum, if 3.6 also is fulfilled. $\delta_x I$ is used to perturb W_k, to guarantee that the direction of the Newton step is descending, i.e, a nonzero value of δ_x indicates that the iterates seem to be in a region where the original problem is not strictly convex.

When (Δx_k, Δy_k,Δz_k) has been computed, the algorithm continues with the computation of the maximum step size as the largest $\alpha_k^{x,max}, \alpha_k^{z,max} \in (0, 1]$, satisfying:

$$x_k + \alpha_k^{x,max} \Delta x_k \geq (1 - \tau) x_k$$
$$z_k + \alpha_k^{z,max} \Delta z_k \geq (1 - \tau) z_k$$
$$\tau = \min\{0.99, \mu\}$$

This ensures that the new iterate will again strictly satisfy Equation 3.9, i.e strictly smaller than the past iterate.

The next step for the solver is a line search with trial step sizes, the trial step size is formulated as:

$$\alpha_{k,l}^{x} = 2^{-l}\alpha_{k}^{x,max}, l = 0, 1, ... \tag{3.13}$$

and l is increased until a step size with "sufficient progress" towards solving the barrier problem is found. The new iterates are then defined as:

$$\begin{aligned} x_{k+1} &= x_k + \alpha_{k,l}^{x}\Delta x_k \\ y_{k+1} &= y_k + \alpha_{k,l}^{x}\Delta y_k \\ z_{k+1} &= z_k + \alpha_{k,l}^{z,max}\Delta z_k \end{aligned}$$

The concept of "sufficient" progress for new iterates is in the context of either a "sufficient" decrease in either $\Phi_\mu(.)$ or in the norm of the constraint violation $||c(x)||_1$. Cycling, i.e getting stuck in a circular search wihout converging to a solution, is also considered in this step, by considering past iterates.

It may happen that there is no step size found that satisfies these objectives. In that case, the algorithm switches to a different mode, the feasibility restoration phase. In this phase, the objective function Φ_μ is ignored temporarily and the solver focuses on finding a point that satisfies the constraints, close to the point where the restoration phase was started. If this phase is successful, the solver returns to the original algorithm, otherwise, the problem is locally infeasible, and solver will return "no solution found" to the user.

Generally it is a bad sign if the solver switches to the restoration algorithm, especially if it happens among the first couple of iterations.

Output of the IPOPT-algorithm

The solver-progress for each iteration is available as output, and the content of each column is explained in Table 3.1. More information is available in [4].

col	**Header**	**Description**	**Desired behavior**
1	`iter`	iteration counter k	few iterations not in restoration phase
2	`objective`	current value of objective function	convergence (constant)
3	`inf_pr`	current primal infeasibility (max-norm), how well the model equations are fulfilled	convergence to tolerance
4	`inf_du`	current dual infeasibility (max-norm), how "optimal" the solution is	convergence to tolerance
5	`lg(mu)`	$\log_1 0$ of current barrier parameter, `mu` $= \mu$	$\mu \to 0$ $\lg(\mu) \to -11$
6	`||d||`	max-norm of the primal search direction,	$\to 0$
7	`lg(rg)`	log10 of Hessian perturbation δ_x	output: `--` (–: no perturbation)
8	`alpha_du`	dual step size $\alpha_k^{z,max}$	1, or increasing
9	`alpha_pr`	primal step size $\alpha_k^{z,max}$	1, or increasing
10	`ls`	number of backtracking steps $l + 1$, see Equation 3.13	1

Table 3.1: Output of the IPOPT-algorithm.

Some tips for interpreting the output
If the optimization is not converging check if the `inf_pr` is large. If `inf_pr` is, then there might be inconsistencies in the model, if there are any constraints that are difficult/impossible to satisfy.

If `inf_du` is large then it might be a good idea to change the optimization setting, for example to modify the cost function, or to increase the number of elements.

Chapter 4

Models

The model objects central for the implementation of the controller algorithm are Modelica-based. There are several modeling and simulation tools that are Modelica-based, JModelica.org and Dymola (which are used in this project) are just two examples. The base models in the project are developed and created in Dymola and the optimization models are created with JModelica.org.

4.1 Create a model

The models represent a system described by differential algebraic equations (DAEs), which for example can stem from mass and energy balances (as for models in this project). The JModelica.org User Guide [9] is a great reference for how to define a Modelica-model, and it also describes some of the relevant attributes and options in more detail, such as `start`, `fixed`, `min`, `max` and `enable_variable_scaling`.

The `start`-attribute is used to define the initial value of a variable and the `fixed`-attribute indicates if the initial value should be treated as an initial guess (if `fixed = false`) or as a fixed start value.

The attributes `min` and `max` corresponds to the possiblitiy of defining min/max values for model variables, which translates to constraints for the optimization problem, if the model is extended for that purpose. It is usually beneficial to set min/max values for all variables, since it limits the range for the optimizer to consider.

It is also possible to manually define scaling factors for all model variables, through the attribute `nominal`. This is important if the values of the model variables differ with several orders of magnitude, which is the case for e.g, thermodynamic models. Large value differences may affect the performance of numerical algorithms, and might even cause them to fail. To consider the nominal values, the model must be compiled with the option `enable_variable_scaling`.

Also worth noting is that the Modelica language supports inheritance; it is simple to extend models and to include instances of other models. Figure 4.1 displays a code example for a very simple model object. There are also tools that support a graphical interface for model develpoment, for example Dymola. This feature is quite nice when the models get large and complex, remember Figure 2.3.

```
model exampleFMU

    parameter Real a = 0.5;
    Real x(start = 1, max = 2, min = 0, nominal = 1, fixed = true);

    input u(start = 0, max = 1, min = -1);

    equation
        der(x) = a*x + u;

end exampleFMU;
```

Figure 4.1: Simple example

The Modelica-based models (without the Optimica extension) are of the class `model` and they are defined in mo-files. The optimization models are of the class `optimization` and they are defined in mop-files, see Section 4.2

4.2 Models for optimization

The models used for optimization are created through JModelica.org, which supports the Optimica extension through the JModelica Model Interface (JMI). A model incorporating this is referred to as an JMU (JModelica Model Unit). There are several features added through this extension, which are described in the JModelica.org User Guide [9]. Some concepts are, however, mentioned here because they are central in order to understand the implementation; the class `optimization` and elements related to the optimization problem introduced in Section 3.1.2: the cost function, constraints and the prediction horizon.

The cost function is represented by the attribute `objective` and may be defined as in Figure 4.2. It is possible to define `cost` directly among the equations in `exampleFMU`, the objective would then instead be connected to that equation.

```
optimization exampleJMU(objective = cost(finalTime),
                                   startTime = 0, finalTime = 1)

    extend exampleFMU;
    Real cost(start = 0, fixed = true);

    equation
        der(cost) = x^2 + u^2;

    constraint
        x+u <= 10;

end exampleJMU;
```

Figure 4.2: Simple example of a optimization model

The prediction horizon of the NMPC-algorithm is set by setting the attribute `finalTime`, assuming `startTime` is set to zero. It is not possible to this change parameter after compilation, but the user may change the time interval in the mop-file.

As mentioned before, constraints may be introduced through defining min/max values for model variables, this is still valid. In addition to this, the Optimica extension supports an additional section for defining constrains. These constraints may be functions of variables as well, however, the min/max contraints have to be constants.

4.3 Model exchange and the FMI-standard

A nice feature of the Modelica-based tools is that they support a standard for model exchange between different modeling and simulation environments, referred to as the FMI-standard (Functional Mock-up Interface). The FMI-standard specifies how the models should be represented and stored, e.g information about the variables, names, identifiers, types and start attributes. A model incorporating the FMI-standard is called an FMU (Functional Mock-up Unit). The FMI-standard enables the possibility to, for example, compile and export a model object in Dymola, and the model object may then be loaded to and used in JModelica.org [18].

Although, not only Modelica-based tools have this feature. In addition to the Modelica-based models used in the project, there is a model object developed through a Siemens in-house tool called Dynaplant. Dynaplant is not Modelica-based, but it supports FMU-export through the FMI-standard. Models may be compiled and exported from Dynaplant, and these model objects are then compatible with JModelica.org.

4.4 FMU vs. JMU

The FMU objects are mainly used for simulation and the JMU objects are used for optimization and linearization.

The model objects consist of a set of states, algebraic variables and input variables, representing the system. The same set of system equation may, however, result in different state selection for the two types. It is possible to define which states that are supposed to be selected by the compiler with the attribute `stateSelect.always`, [14]. Although, it is a good idea to verify that the desired states are selected, from experience it seems to not always be the case.

The state names are generated automatically, and differ in two cases, for derivative-states and for matrix generated states. For example: a JMU state name `'evaporator.der(Tfluid)'` corresponds to `'der(evaporator.Tfluid)'` in an FMU and a JMU state name `'Tfluid[1,1]'` corresponds to a `'Tfluid[1, 1]'` in an FMU. Note the space.

The requirements for initialization (related to simulation, optimization and linearization, see Chapter 5) are also somewhat different for the two types.

- For FMUs, it is enough to specify the state values, and the remaining equations are updated accordingly.
- For JMUs on the other hand, one have to define the values for all variables.

There are pros and cons with the two approaches. It is easier to consistently update the working point of an FMU, it is enough to set the state values, and set `initalize` (a simulation option, see Section 5.1) to `false`. Then all equations will be updated accordingly during the starting step of the next simulation. However, note, if `initalize` is set to `true`, the start values for each state will be used instead and note also that it is not possible to redefine the start values of the states for an FMU model object. If the new set of state values are too inconsistent, the simulation will fail.

For JMUs, it is possible redefine the start values for the states, but if the state values are changed, the system of equations might be inconsistent because all equations are not updated accordingly. It is therefore preferred to update JMU models using simulation or optimization results, containing consistent values for all variables.

For the optimization, recall that one column of the output of IPOPT displays how well the equations are fulfilled (see Section 3.3.2). If the equations are too inconsistent, no solution will be found. In the case of linearization, the result is completely dependent of the working point for all variables, i.e, the result will be completely different (bad) if the model is not initialized properly.

Chapter 5

Useful functions of JModelica.org

The JModelica-functions used for simulation, optimization and linearization are central for the implementaion, and are therefore given some extra attention in this section. For more information see the JModelica.org User Guide [9].

5.1 Simulation - `simulate`

The function used for simulating the models is called `simulate`.

Using the Assimulo package, JModelica.org supports simulation of both FMUs and JMUs. However, the performance is generally better for FMUs and it is the recommended approach for simulation of dynamic systems. For simulation of FMUs, the model is first converted to a system of ordinary differential equations (ODEs), and then simulated. This approach provides better performance and a more robust initialization mechanism, and is hence used in the implementation. For simulation of JMUs, the model is simulated as a DAE system. For more details see [9].

Assimulo is a Cython/Python based simulation package for solving explicit ODEs and implicit DAEs [17]. The reference also contains information about available algorithms and solvers, and the corresponding options for these. The default algorithm for simulation of FMUs in JModelica.org is used in this project, which is `'AssimuloFMIAlg'` with the solver `'CVode'`.

In order to simulate the process one needs to specify for what time interval, and for what input sequence. If the simulation is done for a constant input, it is enough to set the current value of the input variable to a desired constant, if not constant, one may define an input object. See the JModelica.org User Guide [9] for details about how to create an input object.

It is also possible to set a variety of options, both algorithm options and solver options. Only a few of them are used in the project implementation, for more options and more information about the options, see [17] and [9]. If no options are specified, then the default options are used.

`ncp`	The number of communication points. If ncp is zero, the solver will return the internal steps taken.
`initialize`	If set to True, an algorithm for initializing the differential equation is invoked, otherwise the differential equation is assumed to have consistent initial conditions.
`CVode_options:rtol`	Relative Tolerance. Positive float.

Table 5.1: Simulation settings

This is the syntax for simulation of an FMU:

```
res = model.simulate(start_time = 0, final_time = 1.0, input = (),
                             algorithm = 'AssimuloFMIAlg', options={})
```

`start_time`	Defines the start time of the simulation interval
`final_time`	Defines the end time of the simulation interval
`input`	Defines the input.
`algorithm`	Defines the algorithm used for simulation
`options`	Defines the settings for the simulation
`res`	Represents the simulation result

5.1.1 The simulation result

The simulation result contains trajectories for all variables, i.e both states and algebraic variables.

One important aspect to note for the implementation of NMPC: simulations are used to produce predictions of process states, these simulations do not consider the system constraints. The applied control action should consider this, but the constraints may still be violated, especially in the pre-simulation step of the algorithm. This needs to be handled, see the approach used in the implementation in Section 6.4.1.

This is not a problem in the control loop, since the control action is designed in such a way that the constraints are not violated.

5.2 Optimization - `optimize`

JModelica.org supports optimization of JMU models through the function `optimize`.

There are different algorithms available through the function, which are suitable for different kinds of problems, e.g solving control problems (see the user JModelica.org User Guide [9] for more information). The default method is used in this project; **Dynamic optimization of DAEs using direct collocation with JMUs**. This approach requires that the model is a DAE and that it does not contain any discontinuities, which is the case for the model in this project.

There are two main steps of the optimization method:

1. The optimization problem described by the JMU is first approximated as a set of difference equations using direct collocation, i.e, the infinite-dimensional optimization problem is transformed into a finite-dimensional nonlinear programming problem (NLP).
2. The approximated representation of the problem is then solved using the IPOPT-algorithm, an interior point optimization algorithm.

See Section 3.3 for more information.

This is the syntax for optimizing a JMU:

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

`options`	Defines the settings for the optimization
`res`	Represents the optimization result

The optimization problem depends on state start values of the model, and proper initialization is therefore also necessary, in addition to setting the `options`.

`startTime`	Set to 0, corresponds to the start time of the optimization.
`finalTime`	Prediction horizon (H_p). Set in the mop-file, when defining the optimization model. It cannot be changed after compilation.
`n_e`	Number of finite elements, set through the optimization option.
`n_cp`	Number of collocation points, set through the optimization option.
`blocking_factors`	The blocking factors, limiting the possible control actions. Set through the optimization option.
`init_traj`	The initial guess, set through the optimization option.

Table 5.2: Optimization settings

5.2.1 Blocking factors

As mentioned in Section 3.1, one may use blocking factors in order to limit the number of possible control sequences for the optimization to consider.

For `optimize`, they are specified via the option `blocking_factors`. The blocking factors are represented by an array, declearing for how many elements the control signal should be set constant. For example:

```
blocking_factors = [1, 2, 4];
```

correspond to a control sequence:

$$u = [u_1, \quad u_2, u_2, \quad u_3, u_3, u_3, u_3].$$

The blocking factors should correspond to the number of elements, for this example: 7. If the blocking factors are under specified, then the remaining elements are added to the last set.

If no blocking factors are specified, then the collocation polynomials are used to represent the controller. See the JModelica.org User Guide [9] for more information.

5.2.2 Initial trajectory

Most of the time, at least for nontrivial problems, it is necessary to specify an initial trajectory in order for the optimization to converge. A good guess is especially important if there does not exist a global optimal solution to the problem. Both simulation and optimization results may be used as initial guesses (using the same model), more about the strategy used in the implementation can be found in Section 6.

The initial guess should be feasible because of the IPOPT specifications.

5.2.3 The optimization result

The optimization result also contains a trajectory for every variable, that corresponds to the trajectory obtained by using the optimal control sequence as input.

5.3 Linearization - `linearize_ode`

If the model is represented as:

$$F(\frac{dx}{dt}, x, u, w, t) = 0 \tag{5.1}$$

$\frac{dx}{dt}$	Current value of derivatives.
x	Current state values.
u	Current input value.
w	Current value of algebraic variables.
t	Current time instant.

then the linearized representation is given as:

$$\frac{dx}{dt} = Ax + Bu + g \tag{5.2}$$

$$w = Hx + Mu + q \tag{5.3}$$

A, B, H, M	represents the system matrices.
g, q	represents constant offsets.

Through JModelica.org, it is possible to linearize the system of equations. This is necessary as a part of the extended Kalman filter (EKF).

The linearization step is based on the function `linearize_ode`, where the DAE is first converted to a ODE, and then linearized. Notice that the conversion into ODE form works only if the linear DAE has index 1. The outputs of this function are the matrices defining the linear representation, in addition to the point around which the linearization is done ($\frac{dx}{dt}$, x, u, w, t). The order of the states, inputs and algebraic variables are also specified by their names in three separate lists.

These lists are used in order to reorder the linearization results in the implementation, to fit an user-specified state list.

This is the syntax for linearization of a JMU:

From the library:

```
import pyjmi.linearization as lin


A, B, g, H, M, q, state_names, input_names,
      algebraic_names, dx0, x0, u0, w0, t0 = lin.linearize_ode(model);
```

The results are affected by the scaling. If scaling is enabled, then all variable values are divided by their nominal value (scaled to their nominal value). For example, if a state x_1 has the nominal value 10, then if the current value of x_1 is 5, the `x0`-variable corresponding to x_1 will have the value 0.5.

Note that the effect of the scaling only is present in the variables, not in the matrices. The scaling is "moved" to the matrices in the implementation for numerical reasons.

Chapter 6

Implementation

As mentioned, the algorithm is implemented in the Python interface of JModelica.org. This section covers the implementation setup:

- The software setup
- The model objects
- The algorithm setup
- The script
- Some implementation notes and issues

6.1 Software setup

Here is the software used listed:

`Modelica`	Modelica Language Specification 3.3
`JModelica.org`	JModelica.org-SDK-1.8.1, trunk revison 4878. Python packages are included in the installation, see [9].
`IPOPT solver`	MA27 This is not the default solver. See [9] for how to change solvers.
`Dymola`	Dymola 2013 FD01 (32-bit)
`Dynaplant`	2.11.2 revision 1129

6.2 Models

Two base-models were created for this project; one detailed representation of the process, developed in Dynaplant and one simplified representation of the process, developed in Modelica with Dymola, both by the team at Siemens AG Energy Sector. The controller is based on the latter, and the controller model was continuously modified in close cooperation with the algorithm development.

The implementation of the control algorithm involves four model objects, displayed in Figure 6.1:

- An FMU used for real plant simulations (`real_plant`)

and three model objects for the controller

- An FMU used for simulation (`sim_model`)
- A JMU used for optimization (`optl_model`)
- A JMU used for linearization of the system (`lin_model`).

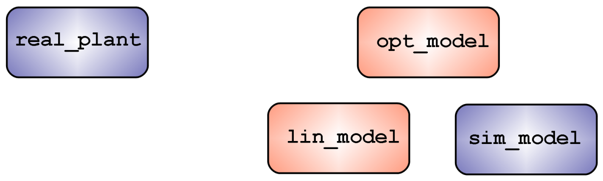

Figure 6.1: The model object representation. FMU = blue, JMU = red

The FMUs are compiled and exported with Dynaplant or Dymola, depending on how they are developed, and then loaded into JModelica.org in the script. The JMUs are compiled in the script with JModelica.org (with the option `'enable_variable_scaling'` set to `True`).

Models developed in Dymola may be compiled with JModelica.org as well, the results are generally the same. However, there have been some occations with problems related to the compliation of Dymola developed models with JModelica.org, both compilation problems and simulation problems. The compilation is therefore done with Dymola, to avoid these possible errors and additional debugging.

Note: The Dynaplant model was never properly evaluated as real plant. There are still issues to consider, mentioned in the section for future work, see Section 8.

6.2.1 The model states and start values

The control algorithm requires some tedious definitions related to the states of both models,

For the controller model:

`x_process`	The dynamic states describing the process.
`x_input`	The input states, should not be estimated.
`x_disturbance`	The disturbance states.

It is also necessary to define the variable names of the parameters that correspond to the initial values of the states. It is important that the order is correct.

`x_process_start`	The start variable names for `x_process`.
`x_input_start`	The start variable names for `x_input`.
`x_disturbance_start`	The start variable names for `x_disturbance`.

The start value variables are only used in the optimization.

It is quite trivial to obtain all state names of a model object;

For an FMU:

```
refs = model_fmu.get_state_value_references();
names_fmu = [model_fmu.get_variable_by_valueref(ref) for ref in refs];
```

For a JMU:

```
names_jmu = model_jmu.get_x_variable_names();
```

The names of the start value variables require more work. There is a compilation option that automatically generate these, but it is unfortunately not applicable for this model, the system turns out to be over-specified. It is instead necessary to investigate the mo-file, and manually define them.

It is sufficient to define the state names of the real plant model states that correspond to the process states of the controller model. The order is crucial here too, it defines which state that corresponds to which. However, note that this may be very difficult if the models are different.

`x_real`	corresponding to `x_process`.

6.2.2 The input states

As mentioned in Section 2.2, there are two inputs available for the process, the feed water mass flow rate and the bypass of the economizer section. An integrator is added to each input; this enables smooth inputs to the process as well as the possibility to include the input derivatives in the cost function, to instead consider the change of the inputs. This limits the possible control actions, and therefore limits the scope for the optimizer.

These states are referred to as `x_input` in the section above (`u1` and `u2` in Figure 2.3). The integrator states also simplifies the interaction with the models, it is not necessary to specify a suitable value for, e.g, the feed water mass flow. Suitable initial values for the inputs are instead considered on the modeling side.

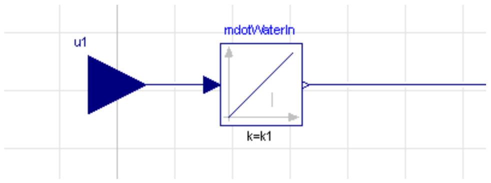

Figure 6.2: A input state in Dymola.

Since the same inputs are available in both the controller model and the plant-model, it is enough to specify the input names in one list: `input_vars`.

The input states are not estimated by the EKF, because they are considered to be known.

6.2.3 Disturbance states

The disturbance states represent states of the controller model, that are not measurable in the real process and are used to compensate modeling errors.

The impact of constant disturbances has been evaluated for several different setups, by adding disturbances, simulating and looking at the results. The controller model seem to be quite insensitive to these changes. At the end, first one disturbance state was added to the model, to compensate at the mass flow of the flue gas (see Test 1-5 in the result Section 7), and later an additional disturbance state for the pressure was introduced (Test 6).

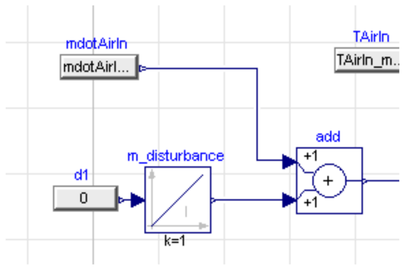

Figure 6.3: A disturbance state in Dymola.

6.2.4 The cost function

The cost function is defined in the controller model (in the .mo-file), it is not a process state, and it is not necessary to define its state names in order to run the application. However, it may be of interest to look at the different parts of the cost function in order to see that everything looks all right.

The cost function used in the implementation is defined as:

$$\texttt{cost} = \min \int_0^{H_p} \alpha_1 u_1^2 + \alpha_2 u_2^2 + \alpha_3 \underbrace{\left(T_{FluidOut,HPSH} - T_{ref}\right)^2}_{\texttt{setpoint}}$$
$$+ \alpha_4 \underbrace{\max\left(0, 1 - \frac{T_{Superheating}}{T_{max,Superheating}}\right)\left(m_{FeedWater} - 1\right)^2}_{\texttt{superheating}}$$
$$+ \alpha_5 \underbrace{\frac{T_{FluidOut,HPEco}}{T_{max}}\left(T_{FluidOut,HPEco} - T_{max}\right)^2}_{\texttt{subcooling}}$$

$$T_{max} = T_{sat} - 5K$$

$$T_{min,Superheating} \leq T_{Superheating} \leq T_{max,Superheating}$$

$\alpha_1 - \alpha_5$	The weights.
u_1, u_2	The control actions.
$T_{FluidOut,HPSH}$	The temperature of the superheater outlet.
T_{ref}	The desired temperature for $T_{FluidOut,HPSH}$.
$T_{Superheating}$	The degree of superheating at the evaporator outlet.
$T_{max,Superheating}$	The maximum value of $T_{Superheating}$.
$T_{min,Superheating}$	The minimum value of $T_{Superheating}$.
$m_{FeedWater}$	The feed water mass flow.
$T_{FluidOut,HPEco}$	The temperature at the economizer outlet.
T_{max}	The maximum temperature allowed at the economizer outlet.
T_{sat}	The saturation temperature, pressure dependent.

or in a more simple representation:

```
equation
...
der(cost) = a1*u1^2  + a2*u2^2  + a3*setpoint + a4*subcooling + a5*superheating
...
```

The cost function is designed with the controller objectives in mind, represented by `setpoint`, `subcooling` and `superheating`, which are all quadratic functions to keep the cost function convex. The control actions (`u1`, `u2`) are also included for convergence reasons, see Section 3.1.2. The weights (`a1-a5`) were evaluated by running the algorithm and analyzing the results, considering convergence of the problem, controller performance, and speed.

The same procedure was used in order to find proper optimization settings: `n_e`, `n_cp`, `blocking_factors` and `finalTime`. The blocking factors are constructed to fit the NMPC-loop, constant for every time step.

No terminal cost or terminal constraints were added to the cost function in the implementation, although this was suggested in Section 3.1.4. It was however also mentioned in the same section that a long enough prediction horizon often is enough. The problems that occurred with convergence in the project were generally related to issues, other than suboptimality, and therefore was $V_f = 0$ considered to be good enough.

6.2.5 Parameter update

It is possible to include parameter updates of the controller model, i.e, update the boundary conditions mentioned in Section 2.3.

Tables representing real plant measurements for a load change are included in the Modelica models that are used as `real_plant`. It is necessary to specify which parameters of the controller model that should be updated according to the tables, and it is also necessary to specify which parameters of the plant model that correspond to the tables.

The controller model is updated according to these measurements at every time step, and considers the current operating point but know nothing of future measurements, i.e future measurements are not a part of the prediction step.

6.3 Algorithm setup

There are five main blocks explaining the setup of the control problem:

- A controller block.
- A process block.

and three blocks representing the state estimation;

- A prediction block.
- A correction block.
- A feasibility correction block.

How the four models are employed in relation to these blocks is displayed in Figure 6.4, the FMUs and JMUs are represented in blue and orange , respectively (darker vs. lighter if in black and white).

6.3.1 The controller block

The controller block determines the next control action by using `opt_model` to solve an optimization problem.

$$\texttt{opt\_model(}ref\texttt{, }\hat{x}_{k|k}\texttt{, sim\_res(t\_k))} \rightarrow u$$

ref	The controller objective.
$\hat{x}_{k\|k}$	The current state estimate. Used for updating the model object.
`sim_res(t_k)`	The values of all variables at t_k, a simulation result. Used for updating the model object.
u	The control action, the result of the optimization.

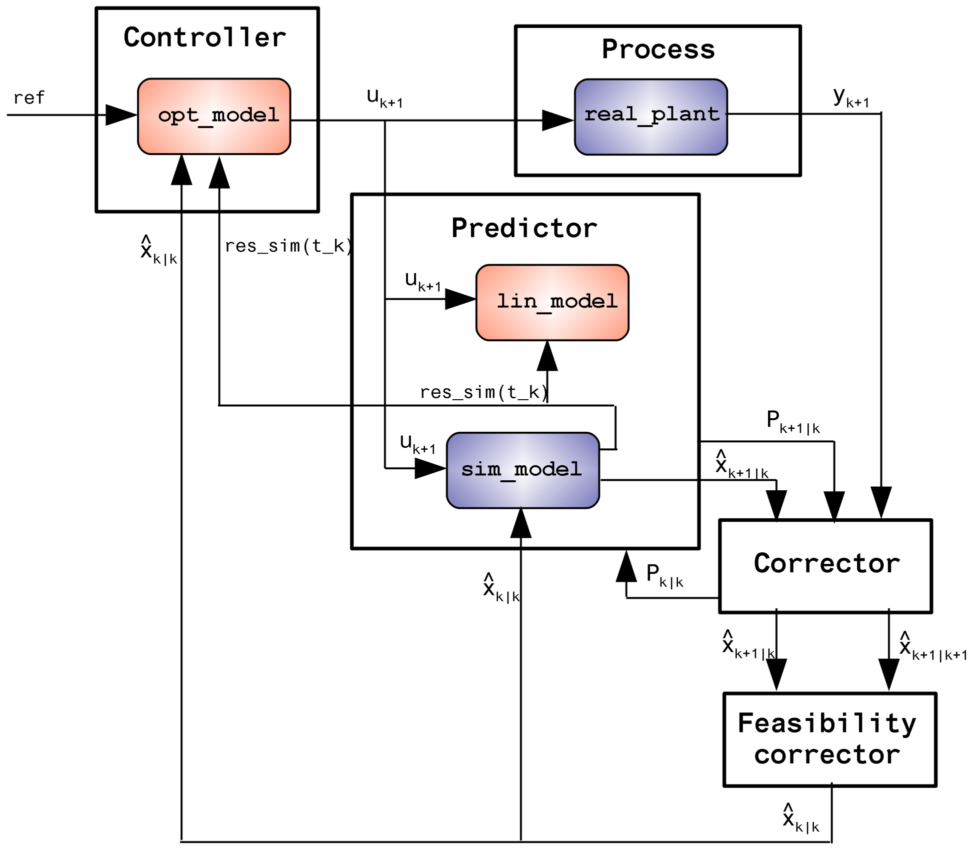

Figure 6.4: Implementation overview

6.3.2 The process block

The process block represents the real plant, where `real_plant` is used to get the plant measurements.

$$\texttt{real\_plant}(u) \rightarrow y$$

u	The input for the plant simulation.
y	The plant measurement, a result of the simulation.

6.3.3 The state estimation

The prediction block represents the EKF prediction step. It computes $\hat{x}_{k+1|k}$ and $P_{k+1|k}$. $P_{k+1|k}$ requires the result from the linearization step.

`sim_model` is used to predict the states and to obtain initialization data for the linearization.

$$\texttt{sim\_model}(u,\ \hat{x}_{k|k}) \rightarrow \texttt{sim\_res(t\_k)},\ \hat{x}_{k+1|k}$$

u	The input for the plant simulation
$\hat{x}_{k|k}$	The current state estimate. Used for updating the model object.
`sim_res(t_k)`	The values of all variables at t_k, a simulation result.
$\hat{x}_{k+1|k}$	The state prediction, a result of the simulation.

`lin_model` is used for the linearization step:

$$\texttt{lin\_model}(u,\ \texttt{sim\_res(t\_k)}) \rightarrow A_k,\ C_k$$

u	The current input.
`sim_res(t_k)`	The values of all variables at t_k, a simulation result. Used for updating the model object.
A_k	A-matrix from the linearization, at $\hat{x}_{k|k}$.
C_k	C-matrix from the linearization, at $\hat{x}_{k+1|k}$.

Note! Normally it would be necessary to reinitialize the model (with `sim_res(t_k)` from a simulation at $\hat{x}_{k+1|k}$) in order to determine C_k. But since C_k will be constant for this implementation, it will not be necessary.

The correction block represents the EKF correction step, and computes $P_{k+1|k+1}$, K_{k+1} and $\hat{x}_{k+1|k+1}$.

The feasibility correction block represents the step which handles infeasible estimates, see Section 6.4.4.

6.4 The script

6.4.1 Initializing

It is important that all model objects are initialized properly, they should be working at the same point in order to be equivalent. It is also preferred that the controller model is in somewhat steady state initially, this simplifies the first optimization since the first initial guess may be constant.

In order to satisfy this, some steps have to be considered on the model side, before running the algorithm. The initial mass flow, together with the initial constants corresponding to plant measurements (representing pressures and so on), should correspond to a desired steady state behavior where the nominal values are good, or at least good enough. This is quite tedious work, but necessary for the current implementation.

Get a starting point - pre simulation

Since it is not possible to set the initial state values of an FMU in the script, the model is simulated for a relatively long time, in order to get the system into a good starting point for a given constant inputs (i.e with the input derivatives (`u1`, `u2`) set to zero). This is referred to as a pre simulation. See Section 5.1 for details about the simulation function call.

```
t_start = 0;             # start of the simulation
t_stop = t_pre_sim;      # end of the simulation

options_sim = {'ncp' = int(t_pre_sim),
             'initialize' = True,
             'CVode_options':{'rtol':1e-7}}

res = sim_model.simulate(t_start, t_stop, options)

options_sim['initialize'] = False
```

Figure 6.5: Simulation of a FMU: the pre simulation.

From now on, the start time is set to the end of the pre simulation, the `initialize`-option is set to `False`.

The result is used to initialize `opt_model` and `lin_model`. It is necessary to initialize all variables of a JMU, not only the states.

```
lin_model.initialize_from_data(res, t_pre_sim)
```

It is necessary to set the start value parameters of `opt_model` in addition to this, in order to change the starting point of the optimization.

The first optimization

`sim_model` is then simulated for an additional period (for the duration of the prediction horizon, with initialize set to false) to construct an initial guess, which is necessary for the first optimization. See Section 5.2 for details about the optimization function call.

```
res_sim = sim_model(t_pre_sim, t_pre_sim+prediction_horizon,
                options = options_sim)

options_opt ={'n_e':n_e,
              'n_cp': n_cp,
              'blocking_factors': N.ones(n_e)/n_e,
              'IPOPT_options':{'max_iter':300,'tol':1e-7},
              'init_traj' = res_sim.result_data} // Setting initial trajectory

res_opt = opt_model.optimize(options = options_opt);

// Updating the initial trajectory:
//    res_opt is also including trajectories for the inputs
options_opt['init_traj'] = res_opt.result_data;
```

Figure 6.6: Optimization of a JMU: the first optimization

Initialization of the EKF

The EKF also requires some initialization. The matrices involved in the filter are adjusted to the nominal values, i.e they are supposed to be well dimensioned, for numerical reasons.

The scaling factors are also defined before the control loop starts. They may be obtained from JMU models:

```
x_model_full = x_dynamic + x_input + x_disturbance
x_scaling_factors = opt_model.variable_scaling_factors[
                          [opt_model.get_value_reference(var) for var in x_model_full]]
```

$P_{0\|0}$	is represented by a diagonal matrix. The diagonal elements are represented by ones, and the elements corresponding to the disturbance states are represented by a larger values. One should set large values for the state values that cannot be trusted.
Q	The rule of thumb is to set the value to the one over the the square root of the standard deviation, in general 1 for this process.
R	The rule of thumb is to set the value to the one over the the square root of the standard deviation, in general 1 for this process.
$x_{0\|0}$	Represented by the current model state values

Table 6.1: Initialization of the EKF

6.4.2 The control loop

The time step h_s corresponds to the duration of the first element of the computed control sequence, i.e $h_s = H_p/n_e$. Remember how the blocking factors are defined (`blocking_factors = N.ones(n_e)/n_e`)

The control loop structure is according to the scheme in Figure 6.4. At t_k:

1. Solve the optimization at $x_{k|k}$. Same options as for the first optimization.
2. Extract the next control action and set it for `real_plant`, `sim_model` and `lin_model`.
3. Update the initial trajectory for next iteration for `opt_model` with the optimization result. The result is shifted one time step in order to provide a better guess, the next optimization problem, solved in the next iteration, will be similar to the previous one, and so will the solution.
4. y_{k+1}:
   Simulate `real_plant` and collect measurements of real plant.
   `res_real = real_plant.simulate(t_k, t_k+h, options = options_sim)`
5. Parameter update:
   Performed before the simulation step of `sim_model` in order to properly update all equations of `lin_model` and `opt_model`.

Prediction step:

6. $\hat{x}_{k+1|k}$:
   Simulate `sim_model`, extract state measurements
   `res_sim = real_plant.simulate(t_k, t_k+h, options = options_sim)`

7. Use the simulation result at `t_k` (`res_sim(t_k)`) to update (re-initialize) `lin_model` and `opt_model`.
   `lin_model.intitialize_from_data(res_sim, t_k)`

8. Linearize `lin_model` to get A_k^{lin} and C^{lin}. C_k^{lin} will be constant and only depend on what states that are measurable, for example if all states were measurable then: $C_k^{lin} = I$. Hence, the linearization step is only done for the prediction step. The linearization step is quite tedious and is further described in Section 6.4.3.

9. Check observability by checking the rank of the observability matrix, computed for the linearized system. The test is described in Section 3.1.7.

10. Discretize A_k^{lin} and C^{lin}, because the EKF use the discretized matrices.
    C^{lin} is equivalent to its discrete representation by construction, $C = C^{lin}$.
    A_k^{lin} is discretized through: `A_k = linalg.exp(A_lin, hs)`

11. Update $P_{k+1|k}$: $P_{k+1|k} = A_k P_{k|k} A_k^T + Q$

Correction step:

12. Update K_{k+1}: $K_{k+1} = P_{k+1|k} C^T (C P_{k+1|k} C^T + R)^{-1}$

13. Update $\hat{x}_{k+1|k+1}$: $\hat{x}_{k+1|k+1} = \hat{x}_{k|k+1} + K_{k+1}(y_{k+1} - C x_{k+1|k})$

14. The states that should not be estimated are set to their corresponding measurement, i.e the input states.

15. Update $P_{k+1|k+1}$: $P_{k+1|k+1} = P_{k+1|k} - K_{k+1} C P_{k+1|k}$

Feasibility check:

16. Check feasibility, see implementation notes in Section 6.4.4

Update:

17. Finally, the `opt_model` and `sim_model` are updated according to the estimated states.

6.4.3 The linearization step

The function that handles the linearization step in the implementation is called `ekf_linerization`. The function call requires a variety of inputs and returns the discrete versions of the linearized A_k and C_k

1. `linearize_ode` is called for a linearization result. Recall Section 5.3.
2. The result is then arranged according to `x_model_full`.

   State names that are not represented in `x_model_full` are removed from the result, e.g the cost-states. The remaining matrices and vectors are then ordered according to the order of `x_model_full`.
3. The C-matrix is constructed as an identity matrix, for which the rows corresponding to disturbance states are removed.
4. The scaling is moved from the state vector to the A-matrix
5. Check observability. (Since A_k^{lin} and C_k^{lin} are available here.)
6. The C-matrix is equivalent to its discrete representation by construction.
7. The discretization of A-matrix is defined as $A = e^{A^{lin}hs}$ and is computed as:
   `A_k = linalg.exp(A_lin, hs)`

6.4.4 The feasibility check

The ad hoc solution for handling infeasible estimates used for the implementation is the one mentioned in Section 3.2.4. The model seems to be quite insensitive to constant state disturbances, and therefore infeasible estimates are simply replaced with the corresponding value from the prediction $\hat{x}_{k+1|k}$. $\hat{x}_{k+1|k}$ will be feasible. However, note that this problem never occurred when evaluating the algorithm.

6.5 Memory issues

There have been several issues when finalizing the implementation, but the most notable one of them is a memory issue. After a few iteration, depending on the setup and the size of the problem, the optimizer returns "Optimization failed. Not enough memory". There are some measures you can take to improve the situation.

Some lines for garbage collection are added to improve the situation:

```
jmu_name=compile_jmu(.., separate_process=True, jvm_args='-Xmx1g')

...

# Collect garbage:
collected = gc.collect()
```

The error message can also be somewhat deceiving, because the issue may also lie in the problem formulation. An ill-posed problem may result in a similar error message.

Chapter 7

Results

The results have been obtained for some different setups. Different models were used to pose as real plant, in order to first verify that the algorithm was working as expected, and then later to evaluate its performance.

In Test 1-5 Modelica models were used as real plant, very similar to the model used by the controller.

Test 1	Verifying the algorithm.
Test 2	Longer run time.
Test 3	Including a load change.
Test 4	Introducing a modeling error.
Test 5	Modeling error and load change.

There are four plots related to each test and all results are **scaled** with reference to their nominal value and time:

1. An EKF plot, displaying the state trajectories for the controller-model and the plant-model for a few selected states: the enthalpy at first evaporator and the introduced disturbance state(s).
2. A controller plot, displaying the control objectives.
3. A controller plot, displaying the control actions applied.
4. A measurement plot, displaying the data used to include load changes.

Test 6 evaluates the effect of introducing a disturbance state in relation to the pressure. The controller model used was modified in order to make this work. The real plant model (a Modelica model) was also modified to fit the new setup.

The Dynaplant model was not evaluated, it was not manageable within the time frame.

7.1 Test 1

```
Setup:
--------------------------------
    Modelica model as real plant
    n_e = 15
    n_cp = 3
    Short loop duration
    One disturbance state
    No load change
--------------------------------
```

For the purpose of verifying the algorithm, the same base model is used both in the controller model and in the model representing the real plant. This especially simplifies the procedure of matching the states between the controller model and the real plant model. The modeling errors for this setup are basically nonexistent, since the two representations practically are the same.

Tables with real measurements for the flue gas mass flow, the flue gas temperature and the pressure are used to mimic a load change of the plant. However, for the time interval used in this test; the measurements do not vary too much and the operating point can be considered to be constant. These tables are not included in the controller model.

A disturbance state is added in relation to the flue gas mass flow, in order to compensate for constant disturbances and modeling errors. For the current setting, there is no constant disturbance (or modeling errors), but the initial guess for the constant disturbance is set to a non-zero value, a 3.6% increase. This value should be estimated to 0 by the EKF. In addition to this there are initial errors for most states, these should also be compensated for by the filter. The initial error arise because the constant disturbance is achieve during the initialization, and because of the slight difference due to the tables.

7.1.1 Test 1: Comments

The EKF appear to be working to satisfaction. The states are estimated fairly quickly, and the EKF estimates the disturbance to 0, see Figure 7.1.

The controller is also performing at a satisfying level, see Figure 7.2. The initial control action for the feed water mass flow is a result of the initial error in the state representation, but as the state representation is corrected, so is the control action, see Figure 7.3.

The offset remaining for the temperature setpoint control could maybe be removed by changing the weights in cost function, the performance is, however, considered to be sufficiently good. Another general approach for getting rid of constant offsets is to add additional disturbance states. This is considered in Test 6.

There is an issue with the memory which makes it impossible to run the algorithm for a longer time for the current settings, this is discussed in the implementation section, 6.5. A possible work-around to avoid this problem is to reduce the size of the optimization problem by for example reducing `n_e`.

Generally, one wants to see that the number of iterations necessary for the IPOPT decreases as the loop progresses, since the initial guess is improving, and since the plant is in somewhat steady state. This is the case for this test.

Test 1: The progress of the state estimation

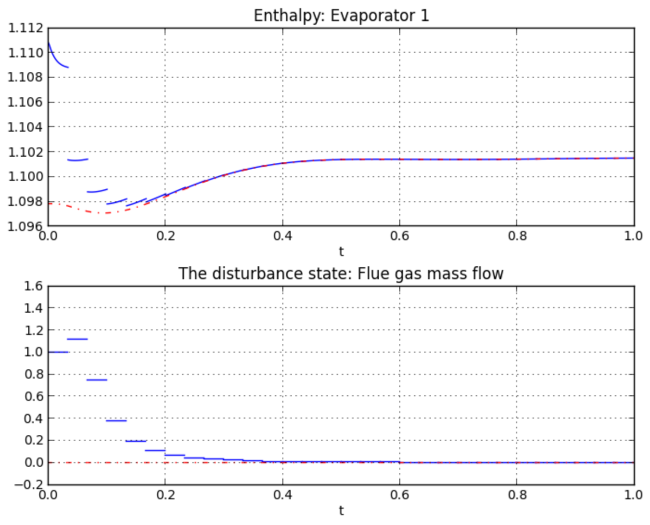

Figure 7.1: The progress of the state estimation, scaled according to nominal value and time. The dashed line represents the real plant behavior. The **top plot** represents one of the process states, the enthalpy at the inlet of first evaporator. The **bottom plot** represents the introduced disturbance state, with an initial error.

Test 1: The controller progress

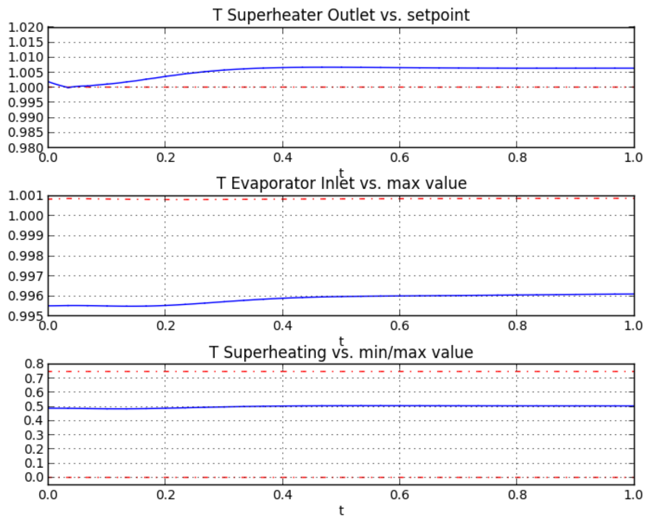

Figure 7.2: The control objective, scaled according to nominal value and time. The **top plot** displays the temperature setpoint control, the dashed line represents the setpoint. The **middle plot** displays the subcooling control, the dashed line represents the maximum value for the temperature at the economizer outlet (pressure dependent). The **bottom plot** displays the superheating control, the dashed lines represents the minimum/maximum degree of superheating.

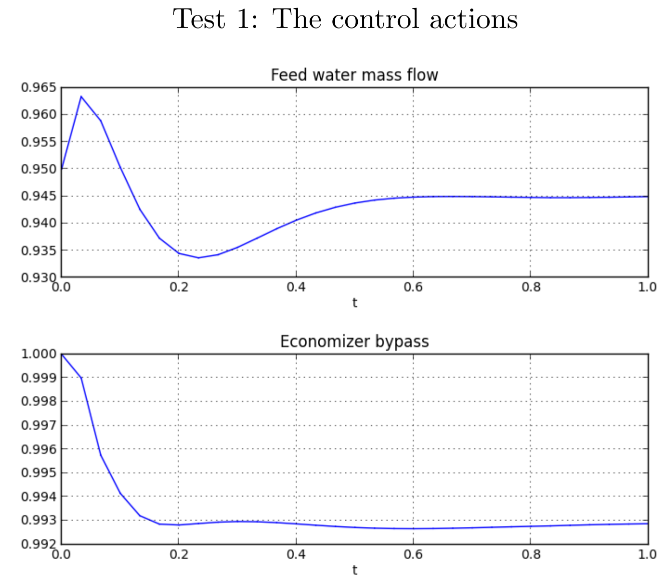

Figure 7.3: The control actions, scaled according to nominal value and time. The **top plot** displays the feed water mass flow. The **bottom plot** displays the opening of the bypass valve, 1 corresponds to a closed valve.

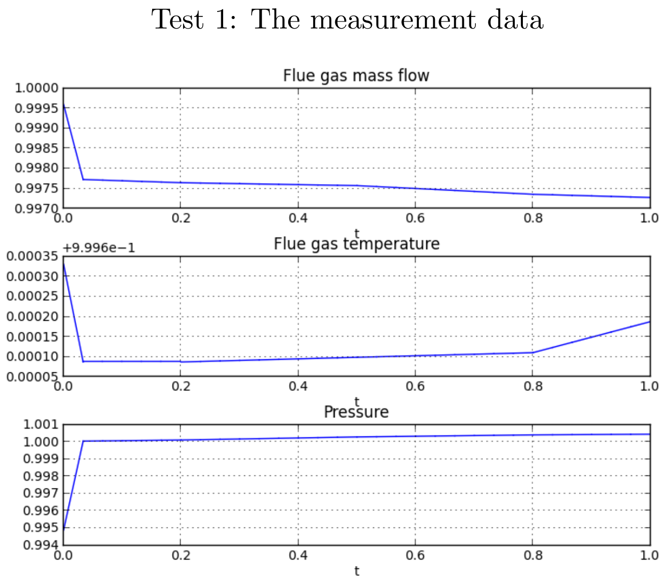

Figure 7.4: The measurement data, scaled according to nominal value and time. The **top plot** displays the flue gas mass flow measurements. The **middle plot** displays the flue gas temperature measurements. The **bottom plot** displays the pressure measurements.

7.2 Test 2

```
Setup:
--------------------------------
    Modelica model as real plant
--> n_e = 10
    n_cp = 3
--> Long loop duration
    One disturbance state
    No load change
--------------------------------
```

For this test the number of `n_e` was reduced in order to run the algorithm for a longer time without running out of memory, as discussed in Test 1. This test also evaluates if the controller seems to still be performing at the same satisfying level. A reduced `n_e` is equivalent to longer sampling time and a more coarse approximation of the system (related to the optimization problem).

However, longer run time is necessary in order to analyze the results for, e.g, load changes, but for this first test with long run time, there is no significant load change in the measurement data.

The disturbance setup for Test 1 remains.

Test 2: Comments

The test was successful. The memory problem was avoided and the results for the controller and the EKF remained at the same satisfying level as in Test 1. By reducing `n_e` the loop duration could be extended to 280%. Further testing will therefore be done for `n_e = 10`.

There is a small load change in the data, ar $t = 0.5$, see Figure 7.8. This clearly has an effect on the control task, and the controller takes action, but the offset for the temperature setpoint control is still remaining.

Test 2: The progress of the state estimation

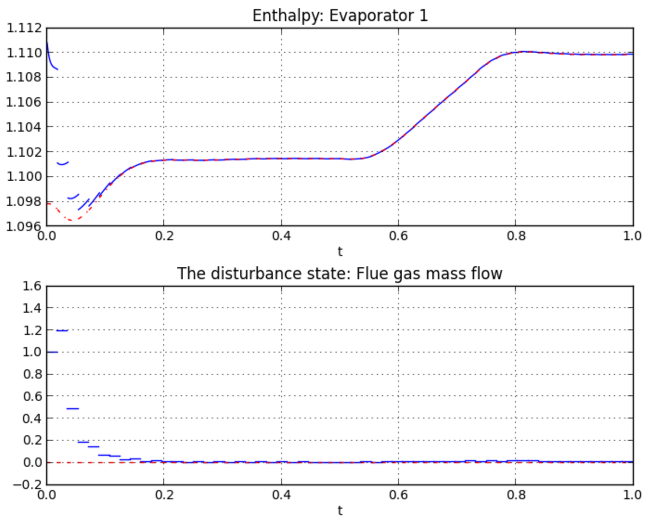

Figure 7.5: The progress of the state estimation, scaled according to nominal value and time. The dashed line represents the real plant behavior. The **top plot** represents one of the process states, the enthalpy at the inlet of first evaporator. The **bottom plot** represents the introduced disturbance state, with an initial error.

Test 2: The controller progress

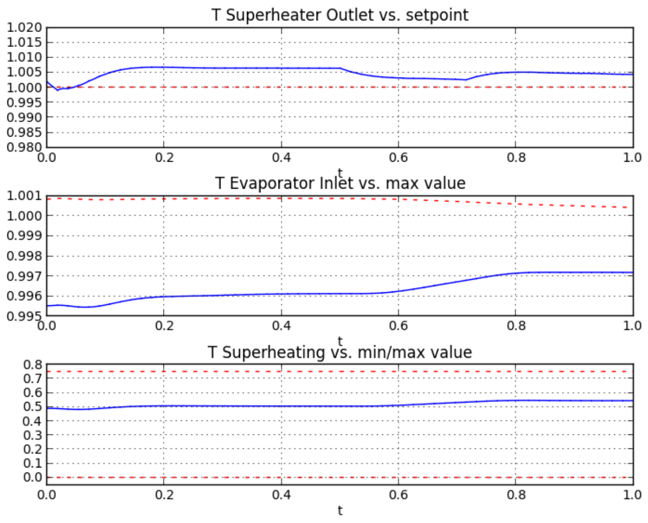

Figure 7.6: The control objective, scaled according to nominal value and time. The **top plot** displays the temperature setpoint control, the dashed line represents the setpoint. The **middle plot** displays the sub-cooling control, the dashed line represents the maximum value for the temperature at the economizer outlet (pressure dependent). The **bottom plot** displays the superheating control, the dashed lines represents the minimum/maximum degree of superheating.

Test 2: The control actions

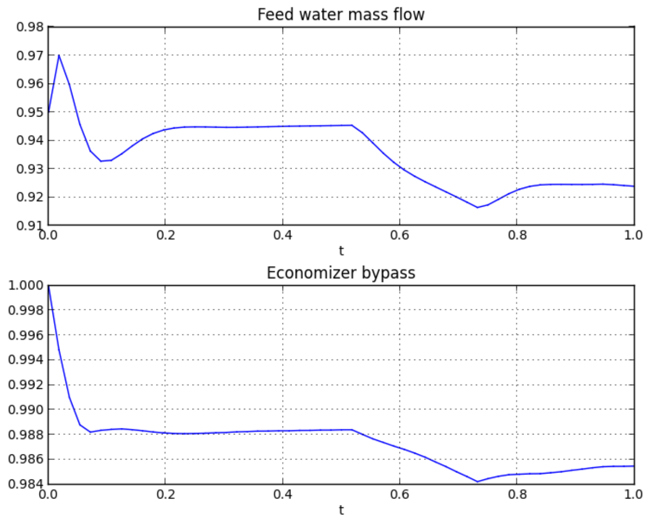

Figure 7.7: The control actions, scaled according to nominal value and time. The **top plot** displays the feed water mass flow. The **bottom plot** displays the opening of the bypass valve, 1 corresponds to a closed valve.

Test 2: The measurement data

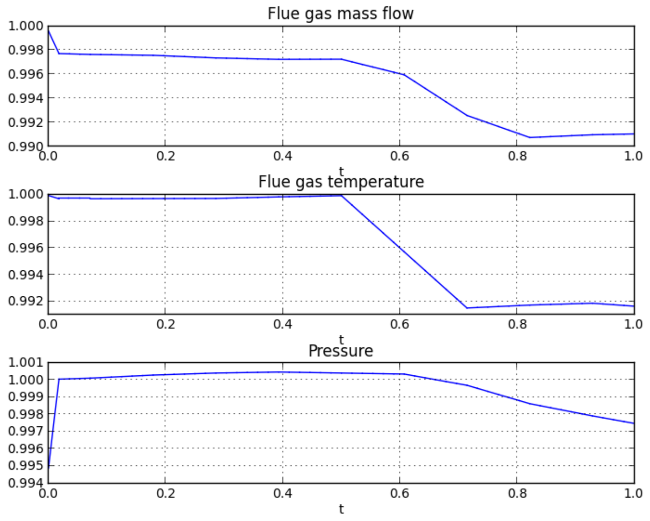

Figure 7.8: The measurement data, scaled according to nominal value and time. The **top plot** displays the flue gas mass flow measurements. The **middle plot** displays the flue gas temperature measurements. The **bottom plot** displays the pressure measurements.

7.3 Test 3

```
Setup:
--------------------------------
    Modelica model as real plant
    n_e = 10
    n_cp = 3
    Long loop duration
    One disturbance state
--> Load change
--------------------------------
```

This test was evaluated for a new set of measurements, including a load change. The settings were otherwise the same as in Test 2.

7.3.1 Test 3: Comments

The performance is still satisfying, but the results are strongly affected by the load change.

The system is not in steady state at the end of the test, so it is hard to evaluate whether or not the temperature setpoint controller recovers from the from the disturbance, it would be interesting to observe the behavior for an even longer time, to see if the offset that appeared in previous tests returns once the system is in steady state.

Something worth noting is that jumps in the ”start” measurement data seems to be hard to handle and they frequently occur in the first step of the measurement update.

Keep in mind that the model object used as real plant is very similar to the controller model. The important thing to note is that the controller can handle load changes, at least for this very simple setup. This is the first step towards using an other model as real plant, the next step is to introduce a modeling error.

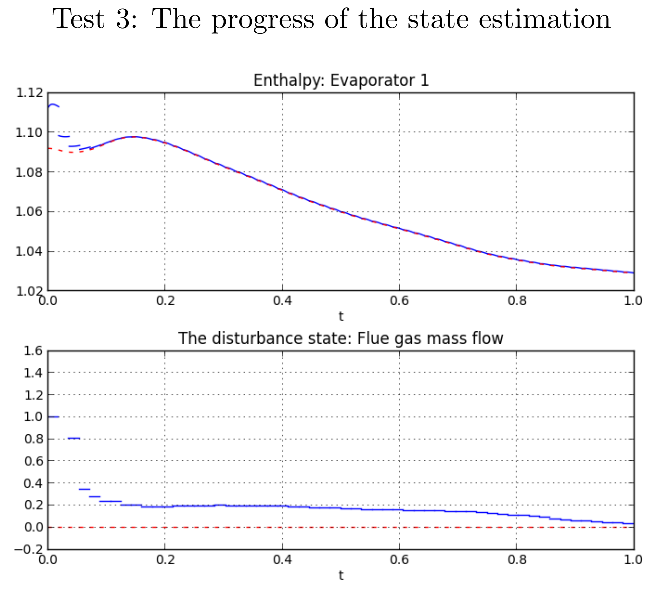

Figure 7.9: The progress of the state estimation, scaled according to nominal value and time. The dashed line represents the real plant behavior. The **top plot** represents one of the process states, the enthalpy at the inlet of first evaporator. The **bottom plot** represents the introduced disturbance state, with an initial error.

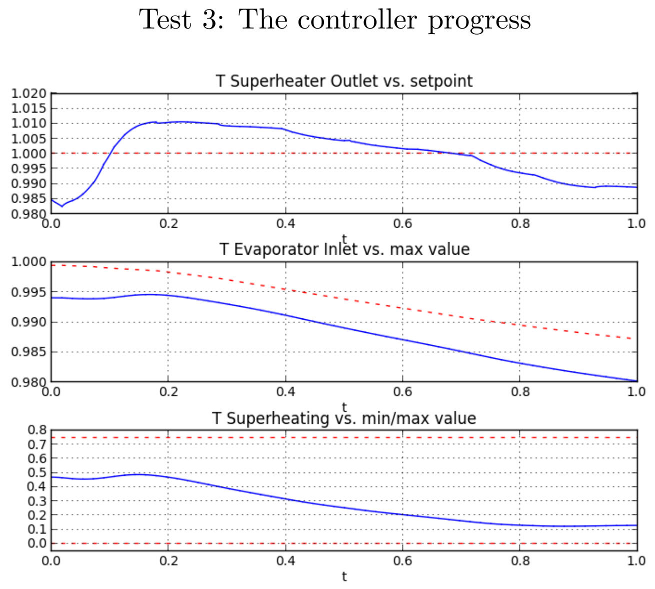

Figure 7.10: The control objective, scaled according to nominal value and time. The **top plot** displays the temperature setpoint control, the dashed line represents the setpoint. The **middle plot** displays the subcooling control, the dashed line represents the maximum value for the temperature at the economizer outlet (pressure dependent). The **bottom plot** displays the superheating control, the dashed lines represents the minimum/maximum degree of superheating.

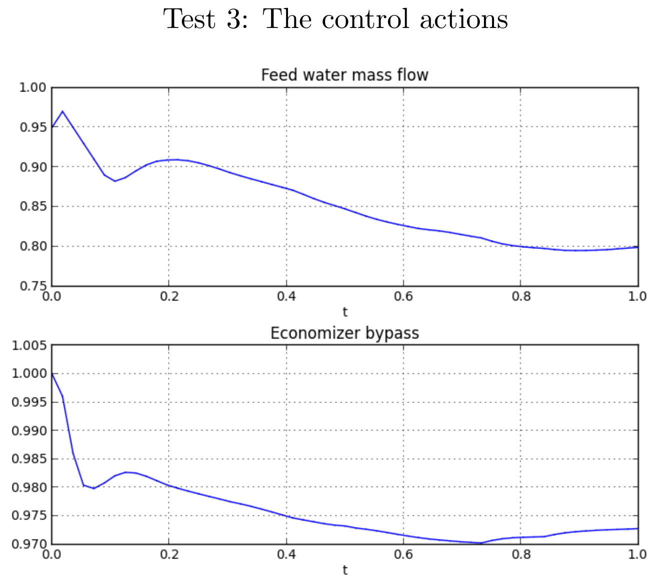

Figure 7.11: The control actions, scaled according to nominal value and time. The **top plot** display the feed water mass flow. The **bottom plot** display the opening of the bypass valve, 1 corresponds to a closed valve.

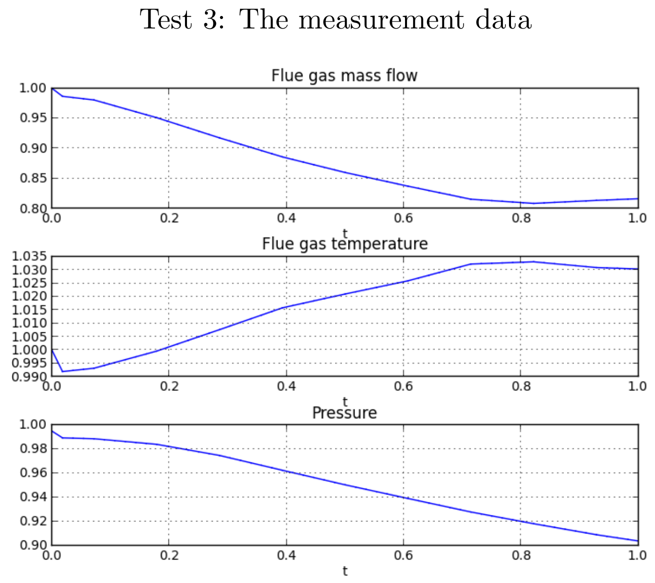

Figure 7.12: The measurement data: scaled according to nominal value and time. The **top plot** displays the flue gas mass flow measurements. The **middle plot** displays the flue gas temperature measurements. The **bottom plot** displays the pressure measurements.

7.4 Test 4

```
Setup:
-------------------------------------------------------------
--> Modelica model as real plant, including a modeling error
    n_e = 10
    n_cp = 3
    Long loop duration
    One disturbance state
--> No load change
-------------------------------------------------------------
```

In reality, there will be large differences between the controller model and the real plant, this is natural. In order to start the testing to evaluate if the controller is ready for real plant control a probable modeling error was introduced to the Modelica model used as real plant.

A probable modeling error could be related to the heat transfer coefficients of the evaporators. A parameter related to this was therefore changed in the real plant representation, it was increased with 9%. This is convenient way of adding a simple modeling error because the state representation remains the same.

The other settings are as in Test 2; a reduced number of elements, a disturbance state related to the flue gas mass flow and no significant load change in the data set.

Test 4: Comments

The EKF has to work harder in this test case, and it is harder to represent the real plant with the controller model. That is why the "jumping" in Figure 7.14 occurs. The controller model predicts the behavior of the states, and then the corrector step changes the value to match the real plant: this results in the jump.

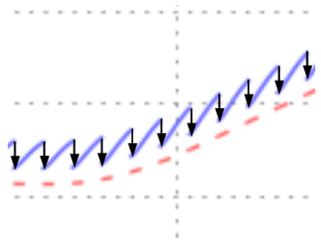

Figure 7.13: The correction of the EKF.

Also worth noting is that the disturbance state no longer estimates the constant disturbance to 0, and this is probably due to the introduced modeling error. The disturbance state tries to compensate for this as well.

The controller performance is quite satisfying and the results are very similar to the results of Test 2. This shows that the controller can handle this modeling error, at least for this set of measurement data.

Test 4: The progress of the state estimation

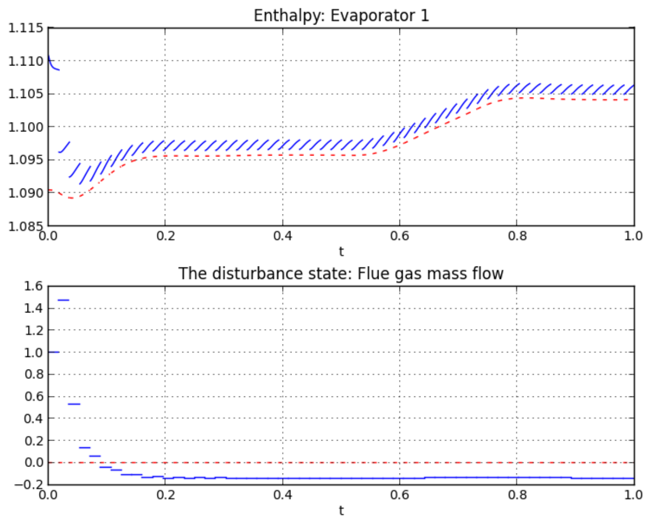

Figure 7.14: The progress of the state estimation, scaled according to nominal value and time. The dashed line represents the real plant behavior. The **top plot** represents one of the process states, the enthalpy at the inlet of first evaporator. The **bottom plot** represents the introduced disturbance state, with an initial error.

Test 4: The controller progress

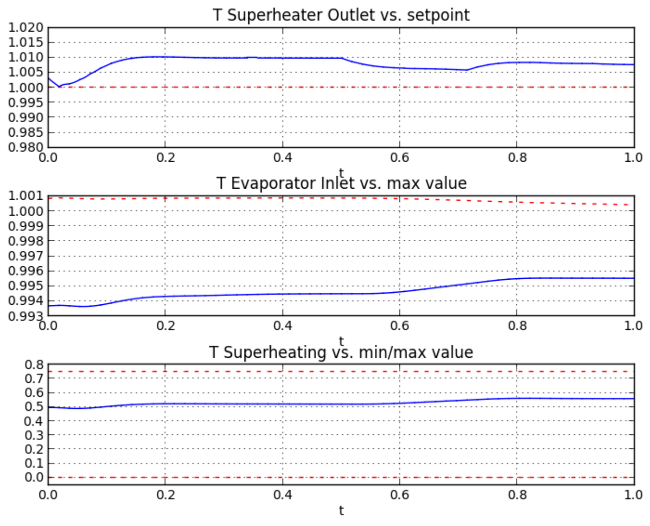

Figure 7.15: The control objective, scaled according to nominal value and time. The **top plot** displays the temperature setpoint control, the dashed line represent the setpoint. The **middle plot** displays the subcooling control, the dashed line represents the maximum value for the temperature at the economizer outlet (pressure dependent). The **bottom plot** displays the superheating control, the dashed lines represents the minimum/maximum degree of superheating.

Test 4: The control actions

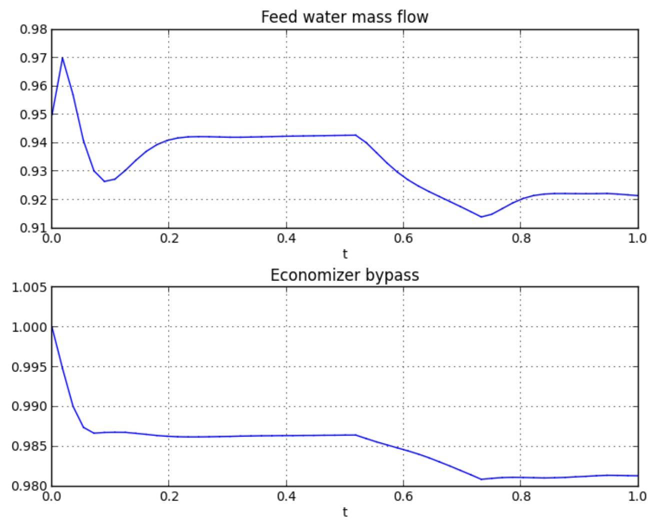

Figure 7.16: The control actions, scaled according to nominal value and time. The **top plot** displays the feed water mass flow. The **bottom plot** displays the opening of the bypass valve, 1 corresponds to a closed valve.

Test 4: The measurement data

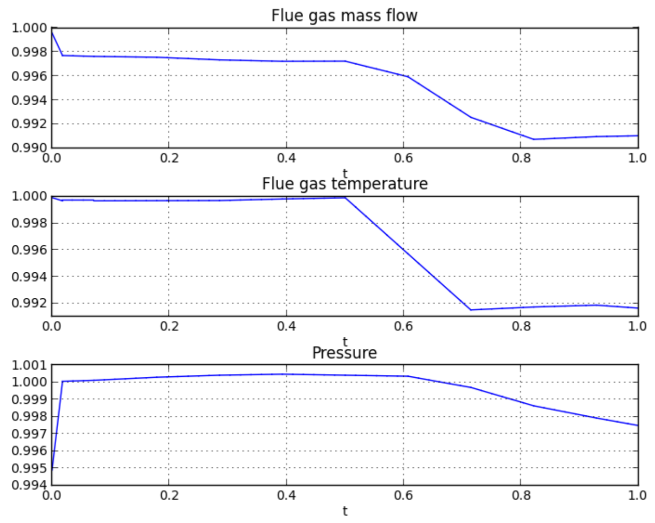

Figure 7.17: The measurement data, scaled according to nominal value and time. The **top plot** displays the flue gas mass flow measurements. The **middle plot** displays the flue gas temperature measurements. The **bottom plot** displays the pressure measurements.

7.5 Test 5

```
Setup:
-------------------------------------------------------------
    Modelica model as real plant, including a modeling error
    n_e = 10
    n_cp = 3
    Long loop duration
    One disturbance state
--> Load change
-------------------------------------------------------------
```

The controller was evaluated for the same setup as in Test 4, but for a data set including a load change.

Test 5: Comments

The performance of the state estimator seems fine for this set of measurements as well. The controller progress is also satisfying, similar to the results of Test 3. The largest difference is observed for the temperature setpoint controller. Initially, the performance is worse, but it looks better towards the end. This could be a coincidence, and it would have been nice to evaluate the controller performance for a longer time and see the behavior once steady state is re-established.

The next natural step is to first include more disturbance states, and it would be preferable to add a disturbance state at the pressure measurement, because it has quite a large effect on the plant behavior.

Test 5: The progress of the state estimation

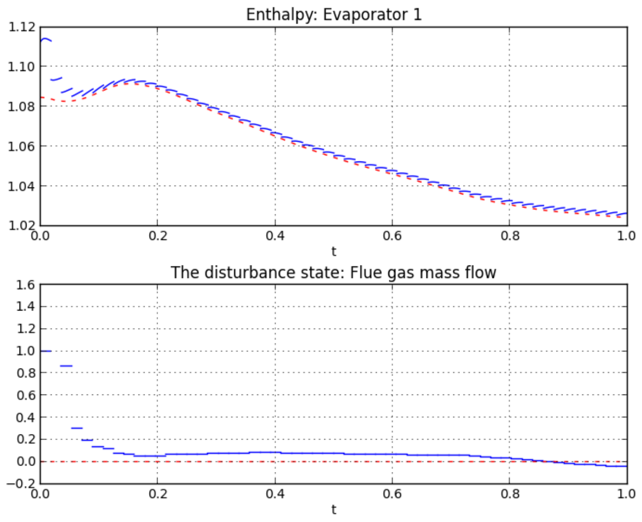

Figure 7.18: The progress of the state estimation, scaled according to nominal value and time. The dashed line represents the real plant behavior. The **top plot** represents one of the process states, the enthalpy at the inlet of first evaporator. The **bottom plot** represents the introduced disturbance state, with an initial error.

Test 5: The controller progress

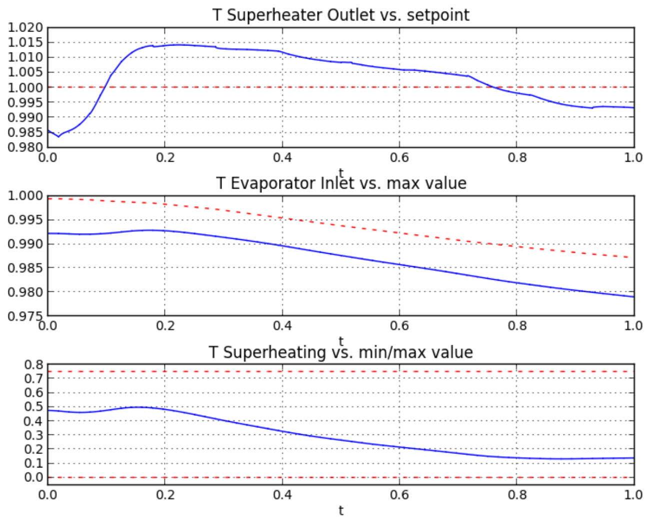

Figure 7.19: The control objective, scaled according to nominal value and time. The **top plot** displays the temperature setpoint control, the dashed line represents the setpoint. The **middle plot** displays the subcooling control, the dashed line represents the maximum value for the temperature at the economizer outlet (pressure dependent). The **bottom plot** displays the superheating control, the dashed lines represents the minimum/maximum degree of superheating.

Test 5: The control actions

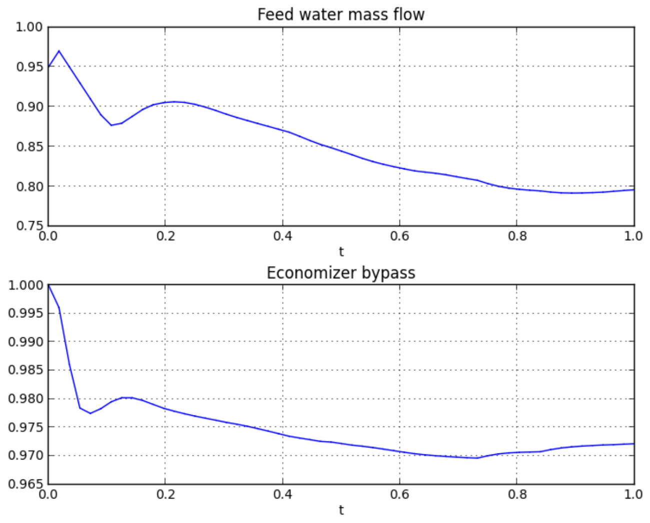

Figure 7.20: The control actions, scaled according to nominal value and time. The **top plot** displays the feed water mass flow. The **bottom plot** displays the opening of the bypass valve, 1 corresponds to a closed valve.

Test 5: The measurement data

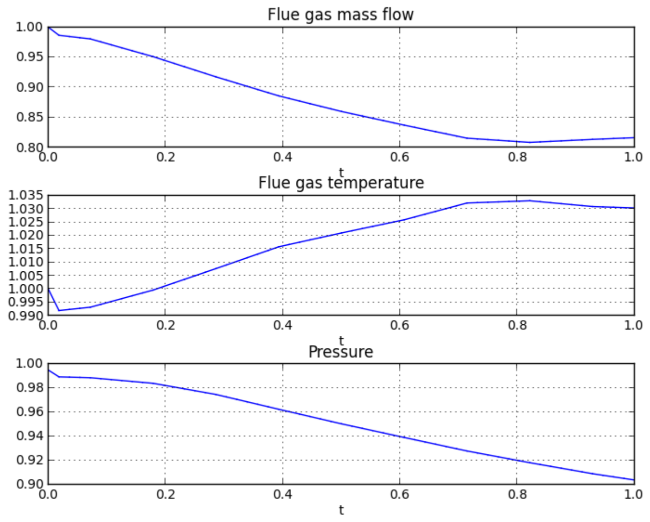

Figure 7.21: The measurement data: scaled according to nominal value and time. The **top plot** displays the flue gas mass flow measurements. The **middle plot** displays the flue gas temperature measurements. The **bottom plot** displays the pressure measurements.

7.6 Test 6

```
Setup:
-------------------------------------------
--> New controller model
--> Modelica model as real plant, new model
    n_e = 10
--> Short loop duration
--> One vs. two disturbance states
--> No load change
-------------------------------------------
```

The model was modified in order to later be able to add a disturbance state for the pressure. The changes made were related to the pressure sink, this resulted in a successful compilation of the JMUs, but a new problem occurred with the number of states. The index reduction made for the JMU in JModelica.org was not sufficient, which resulted in more states in the JMU compared to the FMU. Another effect of this was that the linearization failed.

This was solved by preventing index reduction for the pressure in the heat exchangers, the model went from having integral pressure loss to distributed pressure loss. the script was now able to run, but problems still remained with the convergence of the optimization, because of the added disturbance state for the pressure. A separate model for optimization was introduced, without this disturbance state and the model was instead updated with the corresponding value from the estimation.

It was also necessary to include the measurement table from the model representing the real plant in to the model. Note that the table is not included in the optimization model.

Note that this is a work around solution. The problems could not be solved within the time frame of the project. However, the separate model for optimization might be a good idea, because the optimizer does not need to consider the disturbances as states, only the effects, it only makes the optimization problem larger.

This test evaluates the impact of the pressure disturbance state. The disturbance state for the flue gas mass flow remains, `n_e` was kept at 10 but the control loop was shortened again to only focus on comparing the estimation results of two cases.

7.6.1 Test 6: Comments

First let us explain the spikes in the pressure estimates. This was not present in Test 1-5 and is now, because of the different configuration. The pressure in the pressure sink is different in the real plant model compared to the controller model. The pressure states are algebraically linked to the pressure in the pressure sink. This is where the effect of the added disturbance state for the pressure is visible. The pressure in the pressure sink in the controller model is updated according to the estimation, and the performance of the EKF is improved, see Figure 7.22 and 7.23.

The controller performance seem to not be affected by the two different scenarios for this test. The effects might be visible for a longer loop duration or for data sets including a load change.

It is also worth noting that the performance of the controller, with this new configuration (also without the disturbance state), remains at the same level as in the previous tests.

Test 6.a: The progress of the state estimation, without a disturbance state for the pressure

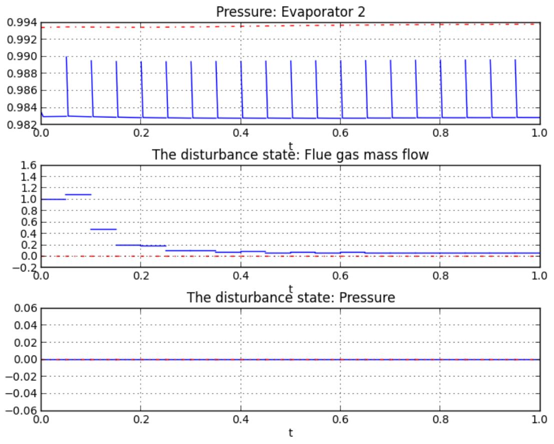

Figure 7.22: The progress of the state estimation without the disturbance state for the pressure, scaled according to nominal value and time. The dashed line represents the real plant behavior. The **top plot** represents one of the process states, the pressure in the middle of the second evaporator. The **middle plot** represents the introduced disturbance state for the mass flow. The **bottom plot** represents "estimation" of the disturbance state of the pressure, here assumed to be 0.

Test 6.b: The progress of the state estimation, with a disturbance state for the pressure

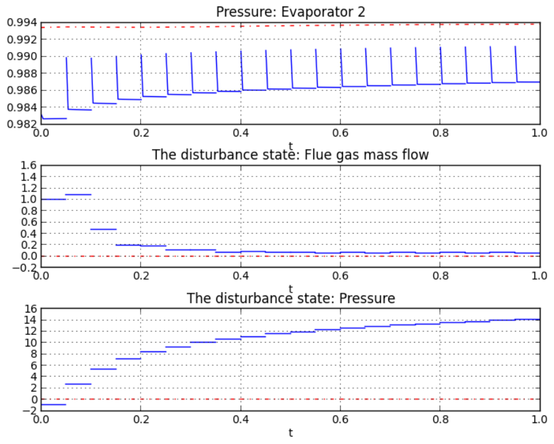

Figure 7.23: The progress of the state estimation with the disturbance state for the pressure, scaled according to nominal value and time. The dashed line represents the real plant behavior. The **top plot** represents one of the process states, the pressure in the middle of the second evaporator. The **middle plot** represents the introduced disturbance state for the mass flow. The **bottom plot** represents the disturbance state for the pressure.

Test 6.a: The controller progress, without a disturbance state for the pressure

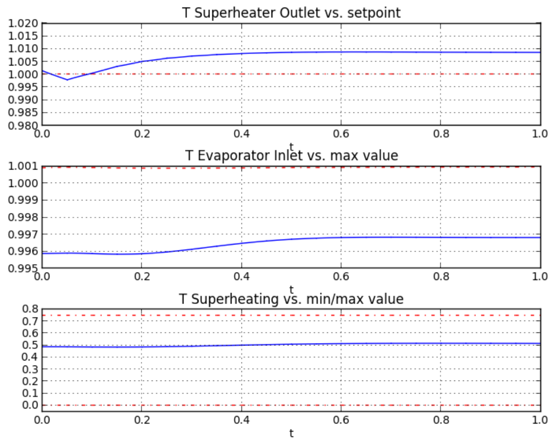

Figure 7.24: The control objective, scaled according to nominal value and time. The **top plot** displays the temperature setpoint control, the dashed line represents the setpoint. The **middle plot** displays the subcooling control, the dashed line represents the maximum value for the temperature at the economizer outlet (pressure dependent). The **bottom plot** displays the superheating control, the dashed lines represents the minimum/maximum degree of superheating.

Test 6.b: The controller progress, with a disturbance state for the pressure

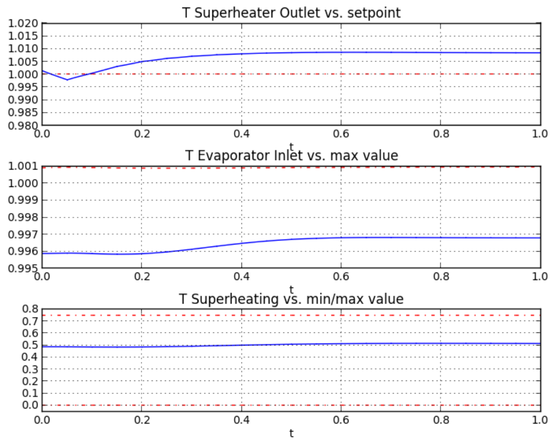

Figure 7.25: The control objective, scaled according to nominal value and time. The **top plot** displays the temperature setpoint control, the dashed line represents the setpoint. The **middle plot** displays the subcooling control, the dashed line represents the maximum value for the temperature at the economizer outlet (pressure dependent). The **bottom plot** displays the superheating control, the dashed lines represents the minimum/maximum degree of superheating.

Test 6.a: The control actions, without a disturbance state for the pressure

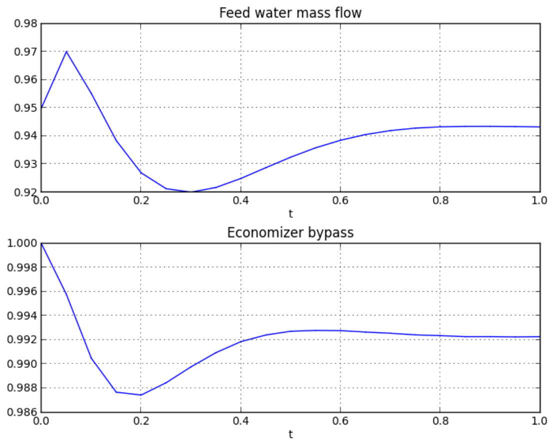

Figure 7.26: The control actions, scaled according to nominal value and time. The **top plot** displays the feed water mass flow. The **bottom plot** displays the opening of the bypass valve, 1 corresponds to a closed valve.

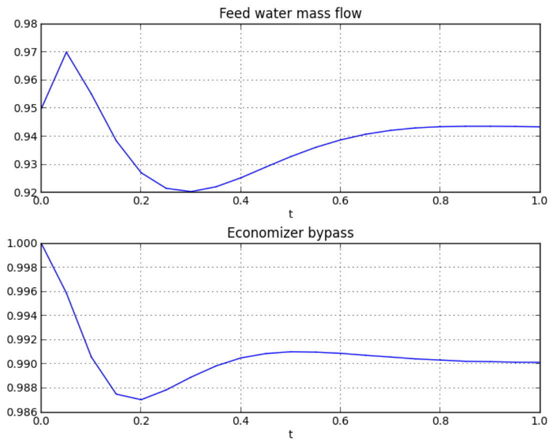

Figure 7.27: The control actions, scaled according to nominal value and time. The **top plot** displays the feed water mass flow. The **bottom plot** displays the opening of the bypass valve, 1 corresponds to a closed valve.

Test 6.a and 6.b: The measurement data

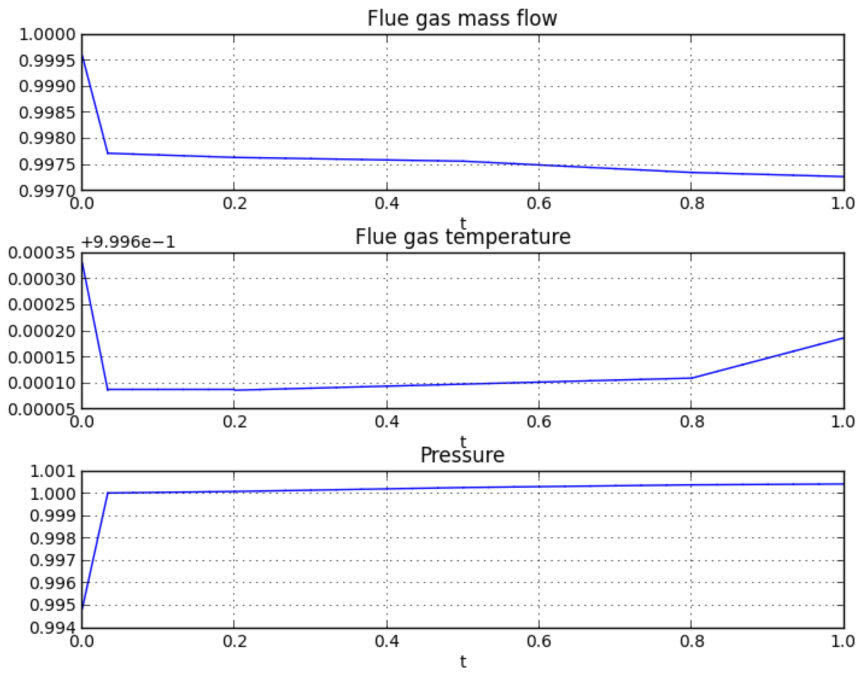

Figure 7.28: The measurement data: scaled according to nominal value and time. The **top plot** displays the flue gas mass flow measurements. The **middle plot** displays the flue gas temperature measurements. The **bottom plot** displays the pressure measurements.

Chapter 8

Concluding comments and future outlook

The NMPC controller, with EKF for state estimation was successfully implemented in JModelica.org. There are however some aspects of the algorithm that should be treated with care, especially the ad hoc solutions. It is good idea to reconsider how these should be handled, or at least to be aware of how they are handled.

- How observability of the nonlinear system is considered.
- How infeasible estimates are handled.
- How time-varying boundary conditions should be handled in the loop (parameter update).
- How the JMU model objects are initialized.
- Where disturbance states should be added.

Also, the current controller algorithm is operating under the assumption that the states that are measurable are pure states. This is not a reasonable assumption when using a separate model as real plant.

In other words, it is of course possible to improve the controller algorithm implemented in this project, and there are some interesting directions to consider, for example:

- Parameter estimation.
- Moving Horizon Estimation (MHE) for state estimation.
- Real time implementation.

- Handling memory issues.
- Improving the model.
- Use Dynaplant model as real plant.

Concerning the boundary conditions: these are obviously necessary approximations, which probably could be improved with parameter estimation. A suggested approach for this is derivative-free optimization, DFO. This is relatively simple to implement in JModelica.org, see [8] as a reference.

It is also suggested that MHE would outperform the EKF as state estimator, since it considers the constraints and does not require the linearization step. The results for the EKF have however been quite satisfying, but it is definitely worth looking into, especially if the plant would be operating close to some constraint. [5]

The controller algorithm implemented in this project is currently far from suitable for real time implementation. The computation time is far too long. It is therefore necessary to investigate how the implementation can be modified in order to make it suitable for this purpose, which really would be an end goal and necessary in order to compete with the currently implemented controller.

The current implementation also has issues concerning memory. It runs out of memory after some iterations. There seem to be some memory leak somewhere. This results in a overall limitation and should really be considered before making any other modifications to the implementation.

An additional approach in order to improve the performance of the control algorithm is to improve with the plant model, since the performance of NMPC is strongly connected to the accuracy of the model. The accuracy can be improved by for example improving the model equations or by adding states in order to better consider modeling errors through the state estimation. The cost function could also be reevaluated.

The control algorithm was never evaluated for control of a process that is different from the model used by the controller (the differences in the tests where very small). A future goal is to apply the algorithm on a model developed in the Siemens in-house tool Dynaplant. The model has been created, and it has been successfully simulated in JModelica (although very slow). The model differences are, however, too large at the moment, but when some of the suggested future steps are considered, e.g parameter estimation, running the algorithm on the Dynaplant model will hopefully be possible.

Bibliography

[1] Kehlhofer, R., Warner, J., Nielsen, H., Backmann, R., *Combined-Cycle Gas and Steam Turbine Power Plants.* ISBN: 0-87814-736-5, second edition, PennWell Publishing Company, Tulsa, Oklahoma, USA, 1999.

[2] Rawlings, J., Mayne, D., *Model Predictive Control: Theory and Design.* ISBN: 978-0-9759377-0-9, first printing, Nob Hill Publishing, LLC, Madison, Wisconsin, USA, 2009.

[3] Maciejowski, J .M., *Predictive Control with Constraints* . ISBN: 0-201-39823-0, first printing, Pearson Education Limited, England, 2002.

[4] Waechter, A., *Short Tutuorial: Getting Started With Ipopt in 90 Minutes.* http://drops.dagstuhl.de/opus/volltexte/2009/2089, 2009, viewed: 2013-03-26.

[5] Haseltine, E., Rawlings, J., *A Critical Evaluation of Extended Kalman Filtering and Moving Horizon Estimation.* http://jbrwww.che.wisc.edu/tech-reports/twmcc-2002-03.pdf, 2003, viewed: 2013-05-07.

[6] Sandberg, H., *Lecture notes on State Estimation* http://www.cds.caltech.edu/ murray/wiki/images/b/b3/Stateestim.pdf, viewed: 2013-02-26.

[7] Magnuson, F., Akesson, J., *Collocation Methods for Optimization in a Modelica Environment.* In 9th International Modelica Conference, Munich, Germany, September 2012. http://www.control.lth.se/Publication/magn_ake2012modelica.html, viewed: 2013-05-07.

[8] Gedda, S., Andersson, C., Akesson, J., iehl, S., *Derivative-free Parameter Optimization of Functional Mock-up Units.* In 9th International Modelica Conference, Munich, Germany, September 2012, http://www.control.lth.se/Publication/and_etal2012modelica.html , viewed:2013-06-03.

[9] JModelica.org, http://jmodelica.org, viewed: 2013-05-15.

[10] Siemens AG, http://www.siemens.com/press/pool/de/homepage/ Siemens-2013-company- presentation.pdf, 2013-04-13.

[11] Modelon AB, http://www.modelon.com/about-modelon/, viewed: 2013-05-15.

[12] MODRIO, http://www.itea2.org/project/ index/view?project=10114, viewed: 2013-04-09.

[13] Modelica, *Modelica and the Modelica Association.* https://www.modelica.org, viewed: 2013-06-18.

[14] JModelica.org trac, *The stateSelect attribute.* http://trac.jmodelica.org/ticket/1924, viewed: 2013-06-18.

[15] JModelica.org, JModelica.org, viewed: 2013-06-18.

[16] Dassault Systems, Dymola, http://www.3ds.com/products/catia/portfolio/dymola/overview, viewed: 2013-06-18.

[17] JModelica.org, *Assimulo.* http://www.jmodelica.org/assimulo, viewed:2013-05-15.

[18] The FMI standard, https://fmi-standard.org, viewed: 2013-09-01.

Lund University **Department of Automatic Control** **Box 118** **SE-221 00 Lund Sweden**	*Document name* MASTER THESIS *Date of issue* September 2013 *Document Number* ISRN LUTFD2/TFRT--5926--SE
Author(s) Anna Johnsson	*Supervisor* Stéphane Velut, Modelon, Sweden Killian Link, Siemens, Germany Karl-Erik Årzén, Dept. of Automatic Control, Lund University, Sweden (examiner) *Sponsoring organization*

Title and subtitle

Nonlinear Model Predictive Control for Combined Cycle Power Plants (Olinjär MPC för gaskombikraftverk)

Abstract

This master thesis project serves to investigate the possibilities of Nonlinear Model Predictive Control (NMPC) using the example of enthalpy control of the BENSON HRSG (heat recovery steam generator) of a combined cycle power plant (CCPP).

The general idea of NMPC is to solve an optimization problem, to find the next control action, and this optimization problem is based on a model of the system. The models used in the controller implementation are Modelica-based, and the system is described by algebraic differential equations (DAEs).

The controller was implemented in the Python interface of JModelica.org (Modelica-based modeling tool, supporting the Modelica extension Optimica for optimization), together with an extended Kalman filter (EKF) for state estimation.

The control algorithm was only evaluated for a setup where the controller model is very similar to the model representing the real process; both models are simplified representations of the real process.

Keywords

Optimization, Nonlinear Model Predictive Control, Extended Kalman filter, Modelica, Optimica, JModelica.org

Classification system and/ or index terms (if any)

Supplementary bibliographical information

ISSN and key title 0280-5316		*ISBN*
Language English	*Number of pages* 1-90	*Recipient's notes*
Security classification		