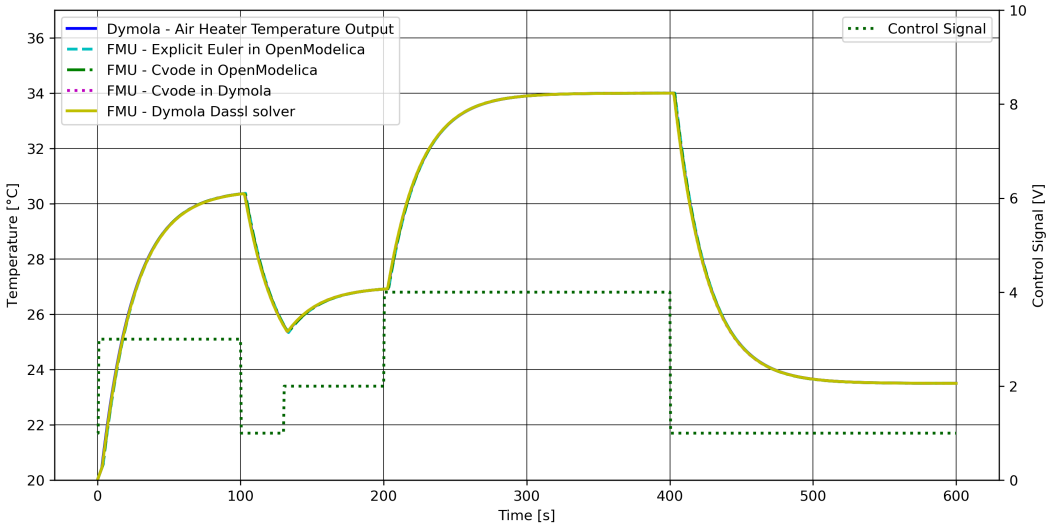
University of South-Eastern Norway

www.usn.no

FMH606 Master's Thesis 2023

Industrial IT and Automation

Advanced Control Implementations with Modelica



Carl Magnus Bøe

Faculty of Technology, Natural Sciences and Maritime Sciences

Campus Porsgrunn

University of South-Eastern Norway

www.usn.no

Course: FMH606 Master's Thesis 2023

Title: *Advanced Control Implementations with Modelica*

Pages: *93*

Keywords: *Modelica external objects, optimization, MPC, model predictive control, Modelica, C, Python, modelling, advanced control, FMPy, NLopt.*

Student: *Carl Magnus Bøe*

Supervisor: *Finn Aakre Haugen*

External partner: *Anushka Perera - Yara Porsgrunn*

Summary:

This thesis explores advanced control implementation in Modelica, focusing on two methods: calling external C code via the Modelica external objects class and utilizing Functional Mock-up Units (FMUs) exported from Modelica for Python simulation. The Model Predictive Control (MPC) chapter covers various models and the optimization processes.

Practical implementations involve evaluating an air heater model using first-principles, transfer functions, and state-space models. The thesis integrates a C noise generator into Modelica and utilizes the NLopt library for optimization. Chapters extend the study to Python, illustrating PI-controller simulation using an FMU from OpenModelica and its integration into an MPC framework. The results are presented through Python source code, statistical measures, and visual comparisons.

Serving as a user manual, the thesis provides detailed implementation descriptions for both methods and ensures transparency and accessibility through GitHub. In conclusion, it not only offers insights into advanced control strategies in Modelica but also serves as a practical guide for implementation.

Preface

As part of the mandatory subjects in the education plan for the Master of Science program in industrial IT and automation at University of South East Norway, the student must produce a master's thesis based on individual work. The master's thesis should include both theoretical and experimental work produced by the student. The work accounts for a total of 30 credits.

This master's thesis is the outcome of a collaboration between USN Campus Porsgrunn and Yara Porsgrunn, one of numerous plants within Yara International. Situated at Herøya in Porsgrunn, Norway, Yara Porsgrunn specializes in nitrogen-based products and boasts the highest production capacity of nitrophosphate fertilizer in Europe. Additionally, the plant manufactures a diverse range of chemicals and gases for industrial applications [1].

I would like to express my gratitude to the following people for taking the time to help me out on this master's thesis. The list is presented in alphabetic order.

Name:	Organization:
Anushka Perera	Yara Porsgrunn
Finn Aakre Haugen	USN - Campus Porsgrunn
Ole Anders Bøe	Family
Turid Eirin Lund	Family

Porsgrunn, 30th October 2023

Carl Magnus Bøe

Contents

Nomenclature

Symbol:	**Explanation:**
MPC	Model predictive control
USN	University of Southeast Norway
FMI	Functional Mock-up Interface
FMU	Functional Mock-up Units
PID	Proportional Integral Differential
SISO	Single input single output
MIMO	Multiple input multiple output
DYMOLA	Dynamic Modelling Laboratory
ODYS	Optimization of Dynamical Systems
GUI	Graphical User Interface

1 Introduction

This chapter provides general information about the master's thesis, including background information and the description of the thesis structure. The task description is included in Appendix A but has been revised as displayed in Chapter 1.1.

1.1 Revised Scope and Title

The initial scope of this thesis was to address the project specifications observable in Attachment A. However, as the research progressed, an unforeseen aspect emerged that significantly affected the project's trajectory. This unforeseen aspect centered around the level of knowledge in C-programming required to perform the implementation of MPC in Modelica with the use of external objects. Because of this aspect the scope and title were changed with consent from both external and internal supervisors in addition to the USN coordinator resulting in a thesis that will present a wider and more general presentation of possible implementations of advanced control in Modelica.

1.2 Background Information

In the absence of inherent support for advanced control methods in Modelica, the current workflow as displayed in Figure 1.1, consisting of generating models within Modelica. Subsequently, the models are exported as functional mock-up units (FMUs) to SIMULINK. Within SIMULINK the MPC block in the model predictive control toolbox is utilized to compute the control signal. Yara's goal is to enhance efficiency by centralizing the entire

Figure 1.1: Current workflow.

system within Modelica. An option under consideration involves the integration of an MPC block directly into the Modelica environment.

1.3 Method and Structure

The thesis begins by providing background information on fundamental concepts, including modeling, Modelica, and model predictive control. Subsequently, it assesses various models implemented.

Following this, the next chapters illustrate various implementations of advanced control methods in Modelica. Non-essential segments of the source code, such as those related to plotting, are removed to emphasize the control implementations. The entire source code for all the implementations in this thesis can be found in the GitHub repository (Link to repository).

Each segment of source code in the thesis is accompanied by a direct link in the caption, guiding readers to the corresponding code in the GitHub repository. Similar linking is also incorporated into the text to reference external source code and complete repositories.

In this thesis, references to source code and repositories are displayed as follows:

1. Link to source code
2. Link to repository

To increase readability, programming-related terms such as 'functions', 'variables', 'classes', and 'models' are enclosed in quotation marks.

2 Background Theory

In this chapter there is a general presentation of modelling, an introduction to the Modelica modelling language, and finally an introduction to model predictive control (MPC).

2.1 Modeling

Creating a model of a system facilitates the execution of tests to understand the systems dynamics. The decision to model and simulate experiments instead of conducting physical tests may present challenges due to one or more of the following factors [2]:

- High cost associated with physical testing
- Potential dangers in experimental setups
- Nonexistent system availability for testing
- Slow system dynamics
- Complexity and time-consuming nature of system updates

Modeling and simulating systems may introduce errors, predominantly influenced by human analysis. It is crucial to avoid becoming overly attached to the model, recognizing that it remains a representation that might not include all real-world attributes. Additionally, the accuracy and usability of results depend significantly on the complexity of the model employed in the experiments [2].

The differences between static and dynamic behavior are displayed in Figure 2.1. In the scenario, both a resistor and a capacitor are excited with a step change in current at $t = 4[s]$. In the ideal case, assuming no transient conditions the resistor follows Ohm law given as $U = R \cdot I$ where the response to the step change in current is only given as a linear function of the value, while the step change in current for the capacitor is given by $I = C \cdot \frac{dV}{dt}$. Because of the time derivative in the expression of the voltage, the charge in the capacitor is accumulated over time and by this representing a dynamic model [2]. A system is static if the output depends solely on the present input. In contrast, when the system output also relies on past inputs, the system is considered dynamic [3].

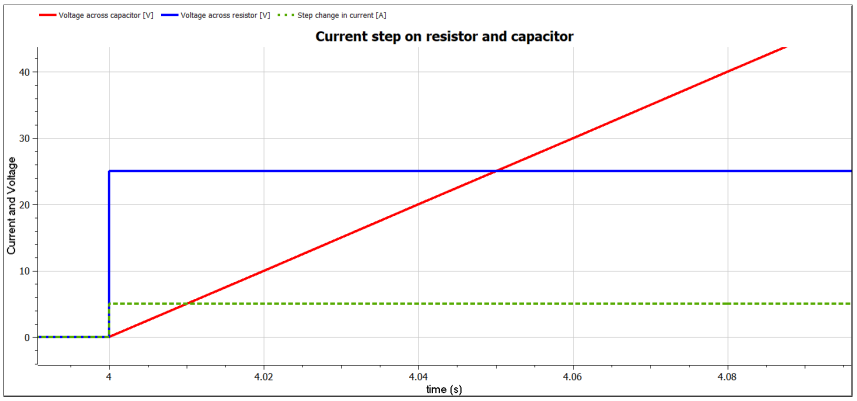


Figure 2.1: Plot of current step response in resistor and capacitor.

The two primary types of dynamic models are continuous-time and discrete-time models. An illustration of the differences is presented in Figure 2.2. Whereas the value of a continuous-time model changes continuously over time, the discrete system only changes values for a range of given points in time. In a practical scenario a physical system where the movement of any physical components can be considered to have continuous-system changes, the set point for controlling the position might be in discrete-time [2]. Considering a physical system as a discrete-time system reduces model complexity, and simulation time. This can in various cases be achieved if the physical system undergoes rapid changes, approaching near-instantaneous behavior, or if the system can experience discontinuity at a specific point in time [2].

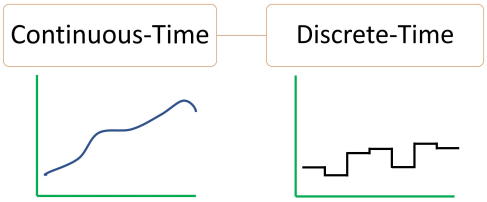


Figure 2.2: Difference between continuous and discrete time models.

2.2 Modelica

Modeling and simulating the dynamics of a physical system may present challenges due to one or more of the following factors:

- The system is intricate and of considerable scale
- Involvement of multiple physical domains
- There is a combination of continuous and discrete behaviors
- Formulating models into systems of ordinary differential equations (ODEs) is necessary to align with a specified solver/integrator
- There is a requirement to engage multiple model developers

The Modelica language, also known simply as Modelica, serves as an effective solution for efficiently addressing these challenges in dynamic system modeling and simulation. The Modelica Association was founded as an independent non-profit organization in 2000 to promote development of the Modelica modeling language and the Modelica standard libraries [2]. The Modelica Association also oversees the five open-source standards displayed in Figure 2.3 [4].

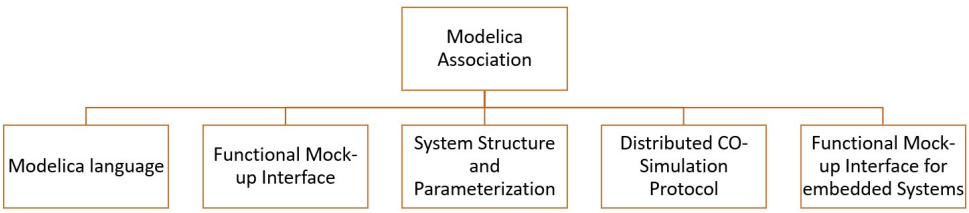


Figure 2.3: Standards governed by the Modelica Association [4].

2.2.1 Modelica Language

The Modelica modeling language is an object-oriented language designed for modeling complex multi-domain systems [5]. Modelica is a declarative modeling language in which dynamic models are expressed through equations, contrasting with procedural languages where typically step-wise algorithms are formulated to achieve a desired goal. Like standard object-oriented languages, Modelica includes classes as templates for objects, incorporating inheritance of variables, equations, and functions among classes [2].

One distinctive feature of modeling systems with Modelica is the ability to model combined systems incorporating elements from diverse domains. Such systems can include a mix of electrical, mechanical, hydraulic, and process-oriented components within a single simulation model. Modelica has an extensive collection of models in the Modelica standard library, which consists of more than 1600 components and 1350 distinct functions in the open-source version. Additionally, various commercial versions of Modelica offer an even greater number of libraries with different components and functions[5].

The Modelica modeling language is the basis for the open-source modeling and simulation environment named OpenModelica and a range of different other commercially available environments as displayed in Table 2.1.

Table 2.1: Commercially available software based on Modelica[5].

Commercially available software	
Altair - solidThinking Activate	Modelon - Modelon Impact
ANSYS - Simplorer	Siemens PLM Software - Simcenter Amesim
Dassault Systèmes - Dymola	Suzhou Tongyuan - MWorks
ESI ITI GmBh - SimulationX	Wolfram - Wolfram SystemModeler®
Maplesoft - MapleSim™	

Execution of Modelica Models

The Modelica source code initiates the process by passing through a series of stages illustrated in Figure 2.4. The translator parses the Modelica code into machine language, conducting analyses, type checks, and converting equations. The output is a flat model

code lacking object-oriented structure. The analyzer organizes equations based on data-flow dependencies, transforming DAEs into a block lower triangular form for efficient numerical solving. The optimizer further simplifies algebraically, retaining a minimal set of equations. The code generator produces C code, and the C compiler links it with a numeric equation solver. Finally, in the simulation stage, initial values are approximated or derived from those specified in the Modelica source code.

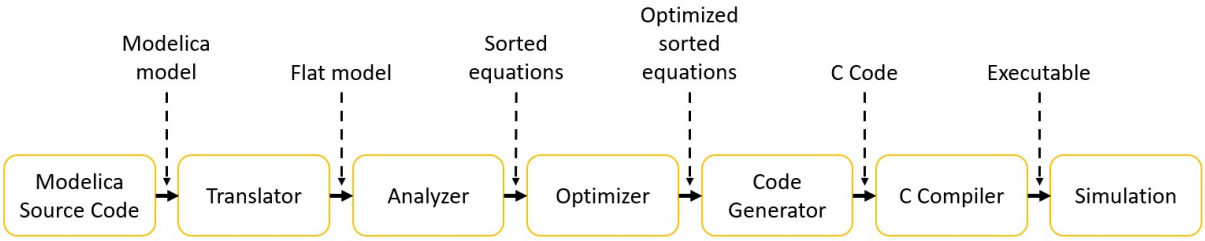


Figure 2.4: Flow displaying how Modelica models are translated and executed. [2].

2.2.2 External Objects

The use of the external object class in Modelica provides a method to store variables and parameters in externally allocated memory, enabling shared data access for Modelica, C, and Fortran 77 environments. This is particularly valuable in scenarios involving multiple function calls where there is a need to store values between these calls [6].

The class 'ExternalObjects', is defined to include only two functions; a 'constructor' and a 'destructor'. The constructor is called once before the initial use of the object, returning a single output argument. The destructor is called after the final use of the external object, with no output argument. Explicitly calling either of these functions is not permitted from the Modelica standard [7].

2.2.3 Functional Mock-Up Interface (FMI)

FMI represents a domain independent standard to enable exchange of dynamic models among different simulation and modelling tools. Based on the FMI there can be generated a vendor neutral functional mock-up unit (FMU), that is supported by an excess of 100 different modelling tools. The generated FMU is a compressed .ZIP file that includes an

.XML file that holds the description of the parameters, derivatives, variables, and model structure in addition to equations and a shared C library [8].

Two of the distinct types of interfaces is outlined below:

- Model exchange
- Co-simulation

Model Exchange

The model exchange interface contains the complete simulation model including the differential equations and the solver. The interface manages the overall advance in simulation time, and the exchange of input and output from the individual FMUs, computing continuous state variables, handling events, and trigger clocks [9].

The model exchange interface is a compact representation with a complete simulation package stored in a single FMU.

Co-Simulation

With the use of the co-simulation interface only part of the simulation model is included. The co-simulation FMU serves as a black box with respect to the external solver, that is required to simulate the system. The FMU receives the simulation results from the external solver, and then integrates these results into the simulation for the current time-step.

Data exchange between the co-simulations is performed at discrete points from one communication point in time, to the next point in time, while the different sub-systems running inside the individual FMUs are solved at their respectively points in time [9].

The co-simulation method has a greater flexibility compared to the model exchange interface because of the possibility to choose different solvers for specific components.

2.3 Model Predictive Control

The PID controller is a versatile choice for controlling both linear and nonlinear processes making it universally applicable for most SISO (Single-input single-output) control scenarios. When the controller needs to handle MIMO (Multiple-input multiple-output) systems and expand control from regulation to optimization to satisfy system constraints, the limitations of single-loop PID controllers become apparent, primarily due to their lack in ability to perform prediction for future dynamic responses in the process [10].

MPC offers a more advanced approach by utilizing a model of the process and thus enables predictions in the process future response when subjected to a sequence of inputs across given future time-steps. The optimization algorithm within the MPC ensures the selection of an optimal input sequence based on a performance index.

A general diagram presenting the process flow with the use of MPC is illustrated in the block diagram in Figure 2.5. The model block within the MPC block displayed in Figure 2.5 represents the model of the process to be controlled, together with an optimizer. The future dynamic response can be predicted using the model. In the optimization block the process model is used to calculate the optimal control signal strategy for the output from the MPC controller, given as u based on the input reference r. The output variable from the process being controlled given by y is fed back to the MPC controller to be utilized in the optimization process in the next time-step [11].

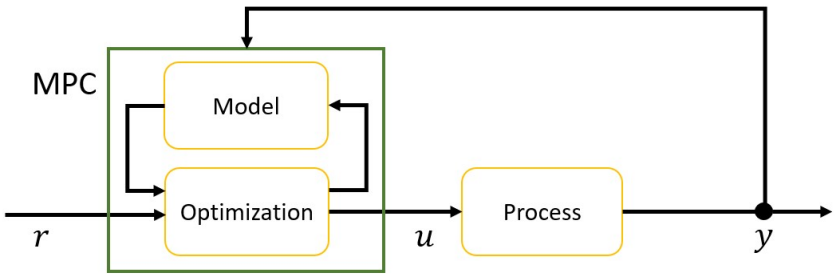


Figure 2.5: Block diagram for MPC. [11].

The MPC considers any constraints while calculating the optimal control signal. Constraints in the optimization process may include parameters such as maximum capacity for heating power of a furnace, span, and speed in opening of a valve, or total volume of

a tank. These are several of the unique features that is possible to achieve with the use of MPC.

An analogy to the MPC has been presented by A.Bemporad in [12] by comparing MPC with a game of chess. When it is your turn to move, you try to find the optimal control strategy several steps forward in time based on the position of the pieces at current time and in addition considering the constraints of how the pieces are allowed to move. The longer your prediction horizon is, the better you are. Still, you are not able to apply more than the first step from the strategy. Based on the opponents move you either continue based on the previous estimate, or you re-evaluate the complete control strategy before the first move of the newly calculated strategy of the sequence is applied [12].

There exists a substantial number of different MPC algorithms, but they all share numerous common characteristics in how they work, based on three main principles [10]:

1. Prediction model
2. Rolling optimization
3. Feedback correction

2.3.1 Prediction Model

The prediction model is utilized to predict the future response of outputs and possible internal states based on historical information and assumed future control inputs. A prediction model can be both linear and nonlinear e.g., based on a transfer function, empirical data, state space equations, or impulse functions [10].

2.3.2 Rolling Optimization

MPC employs rolling optimization to continuously refine the future control actions by iteratively solving the optimization problem at each time step. This dynamic process makes sure the optimization adapts to the process conditions, making sure the control strategy remains responsive to real-time changes.

2.3.3 Feedback Correction

In most practical applications there will exist uncertainties and discrepancies between the model and the physical process along with the potential for various process disturbances to occur. To mitigate these elements the MPC must include a type of closed-loop mechanism to correct this.

The feedback correction can be implemented as a function that delays the application of the values for the calculations, until the present states from the physical process are recorded. With this approach there is implemented a feedback correction making sure both the prediction and resulting optimization will be closer to the future state of the physical process, improving the prediction of future outputs [10].

2.3.4 Types of Models Utilized in MPC

Various types of models can be used for MPC. Among the most employed are linear and nonlinear models, as outlined below:

- Nonlinear and linear discrete time state space models
  - Nonlinear discrete time state space model
  - Linear discrete time state space model
- Continuous time systems
  - Nonlinear continuous time state space model
  - Linear continuous time state space model
- Transfer function model

Nonlinear and Linear Discrete Time State Space Models

A state space model represents a mathematical model of a physical system that has a set of input and output variables called state variables.

Nonlinear discrete time state space model see Equation 2.1.

$$\begin{aligned} x_{k+1} &= f(t_k, x_k, u_k) \quad \leftarrow \textit{State update equation} \\ y_k &= g(t_k, x_k, y_k) \quad \leftarrow \textit{Measurement equation} \end{aligned} \tag{2.1}$$

Both the state equation f and the measurement equation g represents any nonlinear function of time t_k, state x_k and control input given as u_k where the subscript k represents the discrete time.

The model can have more than one control input u, state x and output y. In such a case these variables can be presented as vectors [13].

Linear discrete time state space model see Equation 2.2.

The states are given by x_k and control input given by u_k where the subscript k represents the discrete time. The linear model also includes random process disturbance given by v and measurement noise in w. The two noise terms are uncorrelated with zero mean and some given variance.

$$\begin{aligned} x_{k+1} &= A_d \cdot x_k + B_d \cdot u_k + v_k \quad \leftarrow \textit{State update equation} \\ y_k &= C_d \cdot x_k + D_d \cdot u_k + w_k \quad \leftarrow \textit{Measurement equation} \end{aligned} \tag{2.2}$$

The matrices given by A_d, B_d, C_d and D_d represent the system matrices with the subscript d indicating that the matrices are discrete time linear model.

Both the nonlinear and linear models can have more than one control input u, state x or output y. In such instances, these attributes can be expressed as vectors and can also be presented in a simplified form as in Equation 2.3

$$\begin{aligned} n_x &\leftarrow Number\ of\ states \\ n_u &\leftarrow Number\ of\ inputs \\ n_y &\leftarrow Number\ of\ outputs \end{aligned} \tag{2.3}$$

Continuous Time Systems

First-principles / mechanistic models generated from conservation laws often lead to nonlinear continuous-time models. These models can be expressed as both nonlinear and linear continuous time state space models.

Nonlinear continuous time state space model see Equation 2.4.

$$\begin{aligned} \frac{dx}{dt} &= f(x,u,t) \quad \leftarrow State\ equation \\ y &= g(x,u,t) \quad \leftarrow Measurement\ equation \end{aligned} \tag{2.4}$$

Where functions f and g are any nonlinear function dependent on the state x, input u, and time t. The model can have several numbers of outputs, states, and inputs.

Linear continuous time state space model see Equation 2.5.

$$\begin{aligned} \frac{dx(t)}{dt} &= A_c \cdot x(t) + B_c \cdot u(t) \quad \leftarrow State\ equation \\ y(t) &= C_c \cdot x(t) + D_c \cdot u(t) \quad \leftarrow Measurement\ equation \end{aligned} \tag{2.5}$$

The matrices given by A_c, B_c, C_c and D_c represent the system matrices with the subscript c indicating that the matrices are for continuous time [13].

Transfer Function Model

Models can also be expressed as transfer functions converted to the Laplace domain as presented in Equation 2.6.

$$y(s) = H(s) \cdot u(s) \tag{2.6}$$

Weighting of Inputs and Outputs

Weighting inputs and outputs in MPC significantly influences behavior, enabling the adjustment of balance for conflicting goals within the optimization process to meet specific objectives.

Weighting of the inputs (also known as manipulated variables) is performed to prioritize certain inputs above others. This can be useful for applications where some inputs have a higher impact on the process dynamic response or have stricter constraints than the other inputs. Weighting of the outputs (also known as controlled variables) can be performed if the outputs have a varying level of importance, then by adding the weights will make the controller achieve the desired response.

Tuning the weights of the input and output is essential to balance out various control objectives in an MPC that has conflicting goals. One example can be that the process requires the controller to prioritize maintaining stability and good setpoint tracking, while the process should also optimize energy usage, but with a lower priority than setpoint tracking.

The input weights are presented as an R-matrix and the output weights in a Q-matrix. The values for these matrices are often set as part of a tuning process where the higher the value for the given weight represents the importance for that individual input or output.

2.3.5 Optimization in MPC

The general optimization problem is based on a objective function to be optimized with the possibility to also include various types of constraints into the optimization problem. An example demonstrating process control reference tracking is presented in Figure 2.6.

In Figure 2.6 the variable r is the process reference/setpoint, $y(t)$ is the measured output of the process, and $e(t)$ is the calculated error between the setpoint and the process output [13].

With the objective to minimize the difference between the process output $y(t)$ and the reference r, a equation for the error can be formulated as displayed in Equation 2.7.

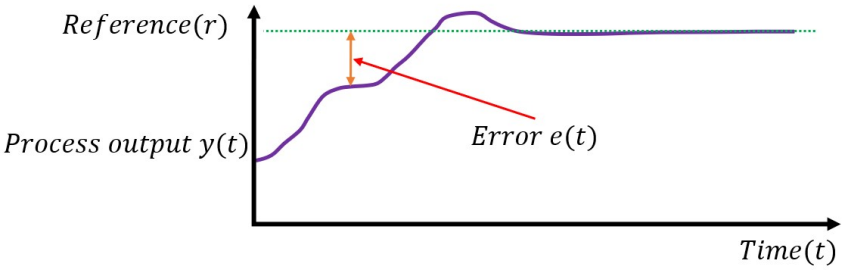


Figure 2.6: Optimization of reference tracking. [13].

$$e = (y - r)^2 \tag{2.7}$$

By summarizing all the calculated errors for the complete prediction horizon given as $k+P$ with k given as one discrete time-step and P representing the complete number of steps for the whole prediction horizon, the optimal control sequence can be calculated for when the sum of squared errors is minimized, and by this find the optimal control sequence for the reference tracking problem displayed in Equation 2.8 [13].

$$e_{Total} = \sum_{k=1}^{P} (y(k) - r)^2 \tag{2.8}$$

Optimization problems are often formulated as minimization problems. The optimization objective function is displayed in Equation 2.9.

$$\min_{U} \quad J(U) \tag{2.9}$$

where U can be a matrix that includes all the different control signals across all the different steps for the whole prediction horizon, and $J(U)$ represents the function to be evaluated. In this case the goal is to generate a matrix U that holds the optimized values for the function $J(U)$ [14].

The optimized control matrix displayed in Equation 2.10 presents all the steps optimized for the complete prediction horizon given as the vector U. This could represent a single input system with only one control signal to be optimized. The total number of steps k in the horizon is expressed as N [14].

$$U = [u_k, u_{k+1}, \ldots, u_{N-1}, u_N] \tag{2.10}$$

From the matrix in Equation 2.11 the optimized control matrix is expanded to display a control matrix for a multiple input process where several control signals are optimized, with the total number of control signals given as r [14].

$$U = \begin{bmatrix} u(1)k & u(1)_{k+1} & \cdots & u(1)_{k+(N-1)} & u(1)_{k+N} \\ u(2)k & u(2)_{k+1} & \cdots & u(2)_{k+(N-1)} & u(2)_{k+N} \\ \vdots & \vdots & \cdots & \vdots & \vdots \\ u(r)k & u(r)_{k+1} & \cdots & u(r)_{k+(N-1)} & u(r)_{k+N} \end{bmatrix} \tag{2.11}$$

Based on a given objective function one can set up a final objective function for an optimal setpoint tracking problem as given in Equation 2.12 [13] :

$$\min_{u} \quad J = \frac{1}{2} \cdot \sum_{k=1}^{N} (e_k^T \cdot Q_k \cdot e_k + u_{k-1}^T \cdot P_{k-1} \cdot u_{k-1}) \tag{2.12}$$

Where the variables for Equation 2.12 are described in Table 2.2.

Table 2.2: Variable description for objective function presented in Equation 2.12 [13].

e_k	Error in setpoint tracking $(y_k - r_k)$
r_k	Setpoint/reference for the complete prediction horizon
Q_k	Weighting matrix for the error signal
u_k	Control input
P_k	Weighting matrix for the control input variables

The optimal control problem can also be formulated to perform the optimization taking into account the rate of change in the control input given as Δu_k as displayed Equation 2.13.

$$\min_{\Delta u} \quad J = \sum_{k=1}^{N} (e_k^T \cdot Q_k \cdot e_k + \Delta u_{k-1}^T \cdot P_{k-1} \cdot \Delta u_{k-1}) \tag{2.13}$$

This can be a relevant formulation for when the process has actuators and control valves that have limitations due to their dynamic performance [13].

Constraints

The optimization problem can also include constraints that must be satisfied, if possible, when performing the optimization. Different constraints include:

- Physical constraints
  - Actuator limits
  - Pump capacity limit
  - Limitations on the rate of change in pump
- Safety constraints
  - Temperature limit
  - Pressure limit
- Environmental constraints
  - Reduction of energy usage
  - Minimization of waste
- Performance
  - Accuracy of setpoint tracking
  - Speed of setpoint tracking

Applying constraints to an optimization problem will limit the solution domain to a feasibility region for which the solution can exist. The constraints can be divided into hard, and soft constraints. Examples of hard constraints are the range of valve opening being limited to maximum 100 %, maximum pump capacity being limited to a given maximum flow, and total volume of a tank. These hard constraints are limited by their general fixed physical attributes. If it is not possible to meet the hard constraints, the

solution can be infeasible. Soft constraints should be satisfied if possible. If it is not possible to satisfy the soft constraints, it is allowed to break them. Examples of possible soft constraints can be temperature in an office space. Even though it is allowed to break the soft constraints, these should be broken in the gentlest way [13].

Equation 2.14 represents a general function that can be subjected to different constraints.

$$\min_{u} f(u) \tag{2.14}$$

The function can be subjected to m number of equality constraints given in Equation 2.15.

$$\begin{aligned} h_i(u) &= 0 \\ i &= (1,2,3,\cdots m) \end{aligned} \tag{2.15}$$

and r number of inequality constraints as presented in Equation 2.16.

$$\begin{aligned} g_j(u) &\leq 0 \\ j &= (1,2,3,\cdots r) \end{aligned} \tag{2.16}$$

Constraints can also be specified as bounds, representing a range of permissible values. This can be expressed as displayed in Equation 2.17 [13].

$$u_L \leq u \leq u_U \tag{2.17}$$

Based on the illustration in Figure 2.6 the process can be extended by applying any general type of constraints. This will limit the feasibility region and as presented in Figure 2.7 result in a different optimized controller strategy.

From the linear constraints applied as the orange dotted line in Figure 2.7 it can be observed that this constraint limits the feasibility region of the responses in the process output. This constraint could be based on the operational speed of a linear actuator that limits the rate of change in the process compared to the reference tracking presented in Figure 2.6.

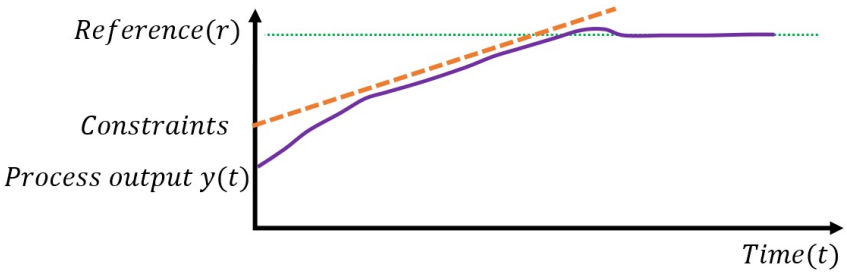


Figure 2.7: Optimization of reference tracking with constraints. [13].

Categories of Optimization Problems

Different optimizations problems include:

- Linear programming problem (LP)
  - Linear objective function
  - Linear constraints
- Quadratic optimization problem (QP)
  - Quadratic objective function
  - Linear constraints
  - Often utilized for linear MPC
- Nonlinear optimization problem (NLP)
  - Nonlinear if objective function or any constraints are nonlinear

Receding Horizon

For each time step forward in time the MPC performs a new optimization cycle to find the optimized trajectory for the output, to be able to set up a control strategy for the control signal based on a given process model. In Figure 2.8 there is an illustration of a reference tracking process for a SISO process.

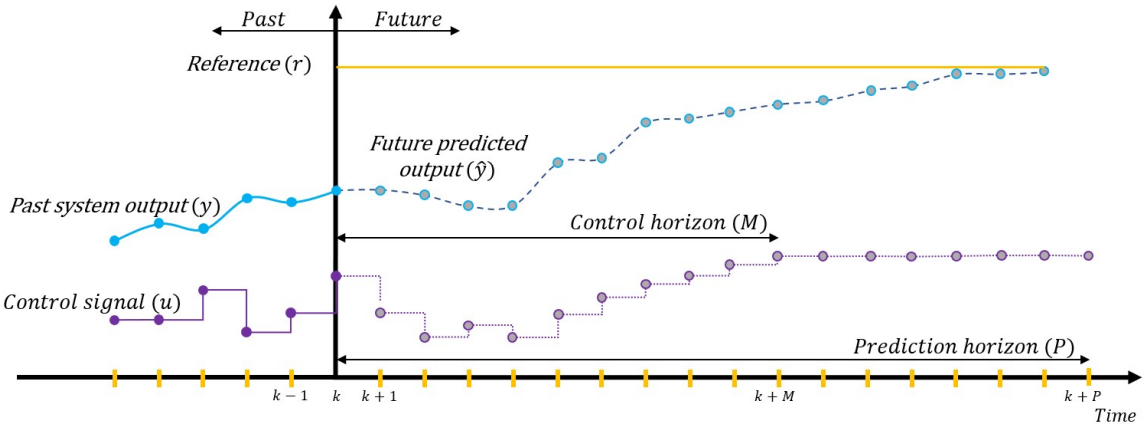


Figure 2.8: Mode of action for MPC with receding horizon [15].

From Figure 2.8 the current point in time is denoted by k. The MPC calculates a set of control inputs along the control horizon given as $k+M$ with the first point being $u(k)$. After $k+M$ number of calculated control signals, the signal is kept constant for the whole duration of the prediction horizon. The total number of control signals for the whole prediction horizon has the duration so that the future predicted output reaches the process set point [15].

Only the first control signal for the whole calculated control horizon is applied to the process, and the remaining signals are discarded. This feature is what is known as the receding/sliding horizon. For the next time-step forward given as $k+1$, the cycle is repeated with a new calculation of the control horizon as $M+1$ and prediction horizon as $P+1$, while still only applying the newly calculated control signal $u+1$.

With the continuous calculation of the optimized trajectory while using the measured present process states from the process, a feed-back component is added to the controller that can adjust for disturbances, signal noise, changes in setpoint, and model inaccuracies [15]. The sliding horizon also introduces a feed-forward function by including the future disturbances and changes in setpoint when solving the optimization problem [13].

3 Review of Existing Options

This chapter presents a review of existing advanced control implementations in Modelica, as well as other implementations based on the use of programming languages that can be connected to Modelica.

A significant portion of the articles investigated are based on JModelica. JModelica is an open-source platform developed by Modelon to perform simulations, optimization, and analysis of complex systems. JModelica is the result of research conducted at Lund University in Sweden, with the goal of creating a platform for exchanging technology between academia and the industry. [16].

The open-source development of the JModelica platform was discontinued in December 2019. Three of the packages were moved to GitHub and are currently still available as open-source projects available in the GitHub repository: (Link to repository) [16]:

- Assimulo
  - A Python-based simulation package available for simulating differential algebraic equations (DAE) and ordinary differential equations (ODE) using explicit Euler or Runge-Kutta methods.
- PyFMI
  - Python package to load and simulate FMUs based on both the model exchange and co-simulation interfaces.
- FMI Library

- C package to load and simulate FMUs based on both the model exchange and co-simulation interfaces.

The compiler and optimization capabilities on the JModelica platform were discontinued as open-source, and further development by Modelon is only available for commercial use with the use of the cloud browser based Modelon Impact. Access to the Modelon Impact gives access to the Modelon commercial version of JModelica called Optimica. The Optimica platform is available inside Modelon Impact with the use of an available Jupyter Lab client.

3.1 Linear MPC Modelica Library

The only open-source general MPC library listed on the Modelica homepage (www.modelica.org/libraries.html) is the now discontinued linear MPC library previously maintained by S. Hoelemann and D.Ablet at RWTH Aachen University.

The goal of the library was to introduce a starting point for implementing advanced control methods in Modelica. The library was developed to implement MPC based on linear process models formulated as a discrete time state space model. The implemented controller manages constraints and disturbances. The library eliminates all dependencies on other software products, and the necessity to generate an FMU of the model [17].

According to the GitHub repository: (Link to repository) the library was last updated in 2016. Since Modelica transitioned to version 4.0, the library is now incompatible due to its reliance on Modelica version 3.2

3.2 Master's Thesis - Carles Buqueras Carbonell

In the master's thesis written in 2010, titled "Model-based Predictive Control using Modelica, and open-source components" by Carles, the suggested approach is to first generate the model in Modelica then compile it into C code with the use of JModelica Python package. Then use the open-source Ipopt optimizer to solve the nonlinear control problem together with a JModelica libraries [18]. With this approach Modelica is only utilized

to define the model and the model parameters. After the model is converted to C code, all the remaining steps of setting up the MPC and the optimizer are performed in C code.

3.3 Article - Model Predictive Control Under Weather Forecast Uncertainty for HVAC Systems in University Buildings

The article investigates a method to improve the performance of an MPC configured to optimize both energy usage and thermal comfort in a building, by implementing an error model for the weather forecast. The building model was constructed in Modelica based on electrical components in a configuration of a RC-network where the R represents the heat resistance and C represents the heat capacitance [19].

The building model was then imported into a Python environment where the model was compiled into a FMU with the use of the JModelica toolbox as discussed in Chapter 3. The FMU is constructed in Python and then utilized in the implemented MPC code. The source code and the .CSV files are available at GitHub: (Link to repository).

3.4 TACO - Tool Chain for Automated Control and Optimization

TACO is a tool chain based on JModelica made to reduce the required engineering skill level and time investment required to implement MPC in building systems. Since the tool has focused on optimizing MPC for building systems it utilizes the near linear structure of the MPC optimization problem, to reduce complexity and computational time.

TACO modifies the JModelica structure by splitting the model equations into two parts. The first part contains all equations that are dependent only on time, and not on state variables, optimization variables, or the model's boundary conditions. The equations are then compiled into an FMU using standard functions from the JModelica platform. These equations do not require to be a part of the optimization problem, and thus reducing computational burden. The other part consists of the remaining equations [20].

The TACO tool chain is a modification of the JModelica platform made to improve MPC for building systems, where the models may contain thousands of state variables and equations.

3.5 ODYS

ODYS is a company founded in 2011 as a spin-off from IMT Lucca. The mission of the company is to bring academic research in advanced control to industrial applications. ODYS currently has three main products in addition to engineering services within the fields of control systems. These products are [21]:

- ODYS QP Solver
- ODYS Embedded MPC
- ODYS Deep learning

The ODYS QP solver is a fast and robust solver coded in plain C code and designed to operate in real-time embedded systems supporting both 32-bit and 64-bit architectures, for any platform that supports floating-point operations . The solver supports both inequality and equality constraints in addition to constraints for the optimization variables. The package can compute the exact worst-case execution time for the solver in advance of the computation [21].

ODYS embedded MPC implements a real-time MPC and state estimator functions in C code with the use of the ODYS QP solver. The MPC handles both linear time-varying and nonlinear prediction models. The MPC supports cost functions, constraints, and degrees of freedom for the optimization, while the estimation is performed by a Kalman filter. The controller is based on a standalone C-code package suitable to run on everything from a desktop computer to embedded computers.

ODYS embedded MPC is a commercially licensed software library. Since the library is built as a standalone C code package without the need of any external libraries it is well suited to be implemented in Modelica with the use of the Modelica function to call external C code from within the Modelica environment [21].

3.6 Book - Modeling, Simulation and Control

In the book called Modeling, Simulation, and Control [14] there is a reference to a source code in Python for the implementation of a nonlinear MPC, to control a simulated air heater. The code includes the implementation of Kalman filtering, disturbances, and control blocking. The source code is implemented from scratch and is only dependent on basic Python libraries such as NumPy, SciPy, and time.

A possibility is to adapt the source code to a more general form and combine it with the use of an FMU model exported from Modelica.

3.7 FMPy - Dassault Systèmes

The FMPy library enables the importing and simulation of FMUs in Python. FMPy is a free Python library that provides support for both model exchange and co-simulation. FMPy can be found on GitHub in the CATIA Systems repository available on GitHub: (Link to repository). CATIA Systems, a subsidiary of Dassault Systèmes is also the provider of Dymola [22].

The FMPy packaged includes six features given as:

1. Graphical user interface
2. Generate code for Jupyter Notebook
    - Import and simulate FMUs in a Python based GUI
3. Simulation in Python
    - Perform simulations of FMUs imported into a Python environment
4. Simulation in command line
    - Simulate FMUs in Python prompt
5. Create a Jupyter Notebook
    - Created by using the export function in the GUI

6. Create a web app
    - Create a web app that can be shared with anyone that has a web browser

3.7.1 FMPy Functions

The documentation of the library functions is based on three example scrips give below [22]:

1. Coupled clutches - Link to source code
2. Custom input - Link to source code
3. Parameter variation - Link to source code

Table 3.1 presents the basic functions from the FMPy library.

Table 3.1: FMPy functions and definitions.

Function:	Description:
read_model_description()	Reads the description of the imported FMU used in building the FMU object
FMU2Slave()	Builds and configures the FMU object
FMU.doStep()	Takes inn simulation time, and simulates one time step forward
FMU.get*()	Returns a list of values from the FMU
FMU.set*()	Takes in a list of values as an argument to be applied to the FMU simulation
FMU.terminate()	Destroys the simulation object

Table 3.2: Caption

3.8 NLopt

NLopt is an open-source library that enables several different methods of nonlinear optimization for C, C++, Fortran, MATLAB, GNU Octave, Python, among others [23].

The algorithms within NLopt are a collection of several open-source packages contributed by different authors. These packages have been adapted to varying degrees to align with the NLopt framework. The source code is available in the GitHub repository: (Link to repository)

The library includes a wide span of different optimization algorithms as presented:

- DIRECT and DIRECT-L
- Controlled Random Search (CRS)
- Multi-Level Single-Linkage (MLSL)
- StoGO
- AGS
- Improved Stochastic Ranking Evolution Strategy (ISRES)
- Evolutionary algorithm (ESCH)
- Constrained Optimization BY Linear Approximations (COBYLA)
- BOBYQA
- NEWUOA + bound constraints
- PRincipal AXIS (PRAXIS)
- Nelder-Mead Simplex
- Sbplx (based on Subplex)
- Method of Moving Asymptotes (MMA) and CCSA
- SLSQP
- Low-storage BFGS
- Preconditioned truncated Newton
- Shifted limited-memory variable-metric
- Augmented Lagrangian algorithm

The NLopt framework is designed to enable creation of multiple optimizer objects built with different optimization algorithms. This gives the possibility to execute multiple optimization algorithms on the same optimization object within the same source code. This can be used to first run a global optimizer, and then run a consecutive optimization with a local optimizer for improved local accuracy.

The general flow of the optimization process with NLopt is displayed in Figure 3.1.

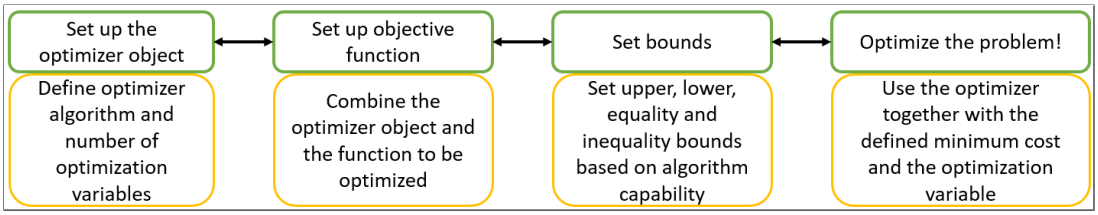


Figure 3.1: NLopt program flow.

The NLopt library can be implemented in C source code enabling utilizing it as an optimizer to be called from Modelica.

4 Implemented Models

This chapter presents several models implemented in Modelica and the evaluation of the models.

4.1 Model of Air Heater Based on First-principles Model

USN has constructed an air heater system that is available in the campus lab station. A key application relevant for this thesis, and the primary motivation for selecting the air heater model is Finn Haugen's Python implementation of MPC for the air heater. The source code for the implementation is available at the web site techteach.no.

A picture of the air heater is presented in Figure 4.1.

Figure 4.1: Picture of air heater [24].

The mathematical model of the air heater is expressed as displayed in Equation 4.1.

$$t_{const} \cdot \frac{dT}{dt} = (T_{amb} - T) + K_h \cdot u(t - t_{delay}) \tag{4.1}$$

With the process dynamics represented by the constants given in Table 4.1.

Table 4.1: Air heater process constants.

$K_h = 3.5[\frac{C}{V}]$	Heater gain
$t_{const} = 23[s]$	Time constant of the air heater
$t_{delay} = 3[s]$	Time delay of the air heater

With the variables given in Table 4.2.

Table 4.2: Air heater process variables.

$T[°C]$	Temperature at the tube outlet
$T_{amb}[°C]$	Air heater ambient temperature
$u[V]$	Air heater control signal
$t[s]$	Time used to add the delay

A more comprehensive presentation of the air heater with additional details is available at the web site: techteach.no and in the report "Demonstrating PID Control Principles using an Air Heater and LabVIEW" from (Haugen, Fjelddalen, Dunia & Edgar, 2007)[25].

4.1.1 Model of Air Heater in Modelica

The air heater was implemented into Modelica as a model, and to observe the dynamic response of the air heaters, it was added an additional 'IF' statements to change the control signal at given points in time. The resulting Code is presented in Code 4.1.

The results of the simulation is displayed in Figure 4.2. The variable u represents the input control signal, and the variable T_{Out} represents the air heater output. The simulation was run for $600[s]$ while the control input was manipulated as displayed in Code 4.1. From Figure 4.2 the process time constant, and the delay between the control signal u, and the output temperature T_{Out}, can be observed.

```
model AirHeaterWithStep

  // Import
  import Modelica.Units.SI;
  import Modelica.Units.NonSI;

  // Constants
  constant SI.Time T_CONST = 23.0              "Time constant";
  constant SI.Time T_DELAY = 3.0               "Time delay";
  constant Real Kh(unit="C/V") = 3.5           "Heater gain";

  // Parameters
  parameter NonSI.Temperature_degC T_amb = 20  "Ambient/Room temperature";

  // Variables
  NonSI.Temperature_degC T_Out(min=T_amb)      "Temperature output from heater";
  SI.Voltage u(min=0, max=5)                   "Control Signal";

initial equation
  T_Out = T_amb;

equation
if time>0 and time<100 then
      u = 3;
    elseif time>=100 and time<130 then
      u = 1;
    elseif time>=130 and time<200 then
      u = 2;
    elseif time>=200 and time<400 then
      u = 4;
    else
      u = 1;
  end if;

  T_CONST * der(T_Out) = (T_amb - T_Out) + Kh * delay(u, 3);

  annotation(experiment(StartTime=0, StopTime=600)

end AirHeaterWithStep;
```

Code 4.1: Air Heater Implemented in Modelica - Link to source code.

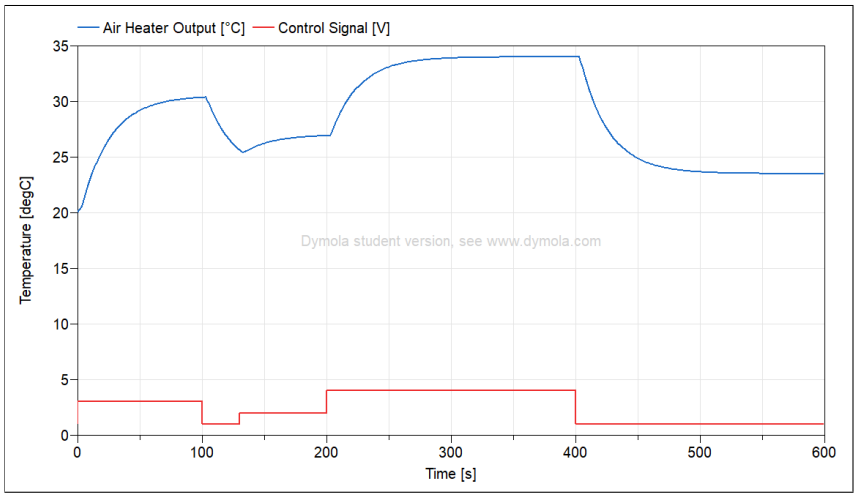


Figure 4.2: Plot of air heater with step changes in control signal.

4.1.2 Calculate PI Parameters for the Air Heater

From a step response introduced to the process as displayed in Figure 4.3, the parameters for a PI-controller were calculated based on the Skogestad-method where the process is approximated by assuming integrator with time delay dynamic response, given by the purple line [14]. The experiment was run with the model named 'AirHeaterTUNING' in the Examples folder of the Modelica source code.

The slope of the step response S was calculated as displayed in Equation 4.2.

$$S = \frac{T_2 - T_1}{t_2 - t_1} = \frac{38 - 25}{239 - 212} = \frac{13}{27} \simeq 0.481 \tag{4.2}$$

The integrator gain was calculated dividing the slope of the step response with the change in control signal displayed in Equation 4.3.

$$K_i = \frac{S}{U} = \frac{\frac{13}{27}}{4.5} = \frac{26}{243} \simeq 0.11 \tag{4.3}$$

The proportional gain of the controller was calculated with the use of the known time delay given as $3[s]$ from Table 4.1 represented as τ displayed in Equation 4.4.

$$K_p = \frac{1}{2 \cdot K_i \cdot \tau} = \frac{1}{2 \cdot \frac{26}{243} \cdot 3} = \frac{81}{52} \simeq 1.56 \tag{4.4}$$

The integral time was calculated as displayed in equation 4.5.

$$T_i = 4 \cdot \tau = 12 \tag{4.5}$$

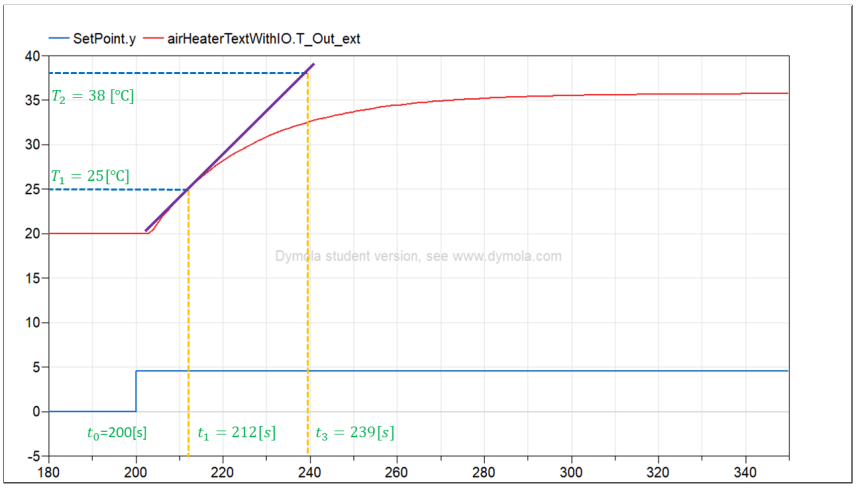


Figure 4.3: Skogestad method for tuning of PI-controller [14].

In Modelica the air heater model was connected to the built-in PID block together with step blocks to simulate step changes in the ambient temperature at $100[s]$ and step change in setpoint at $200[s]$, as displayed in Figure 4.4.

Figure 4.5 displays the air heater's response to both a change in ambient temperature and change in the control signal. The results display oscillations in the air heater output when the step change in setpoint is applied to the controller.

4.2 Model of Air Heater Based on Transfer Function Model

The model parameters from Table 4.1 was utilized in a general 1.order transfer function with time delay given in Equation 4.6. This was implemented in Modelica with the use of a transfer function block from the standard library in addition to a fixed delay block,

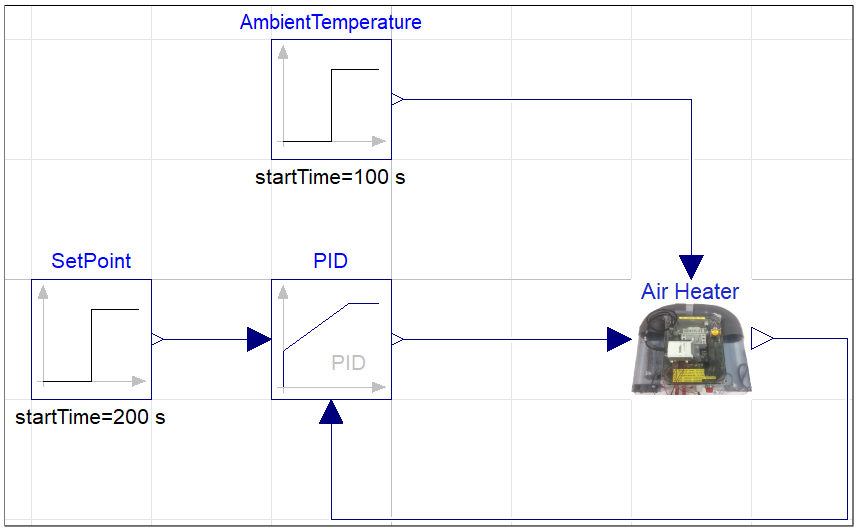


Figure 4.4: Picture of air heater with PID-controller.

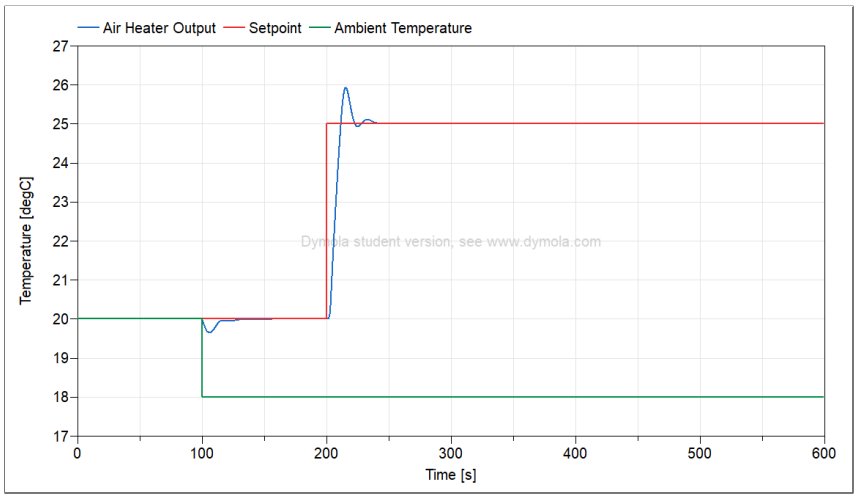


Figure 4.5: Plot of air heater with PID-controller.

as displayed in Figure 4.6. The implemented source code can be found in the GitHub repository (Link to source code)

$$H(s) = \frac{K}{Ts+1} \cdot e^{-\tau \cdot s} \longrightarrow H(s) = \frac{3.5}{23s+1} \cdot e^{-3 \cdot s} \tag{4.6}$$

The resulting plot from the simulation of the transfer function model is presented in Figure 4.7.

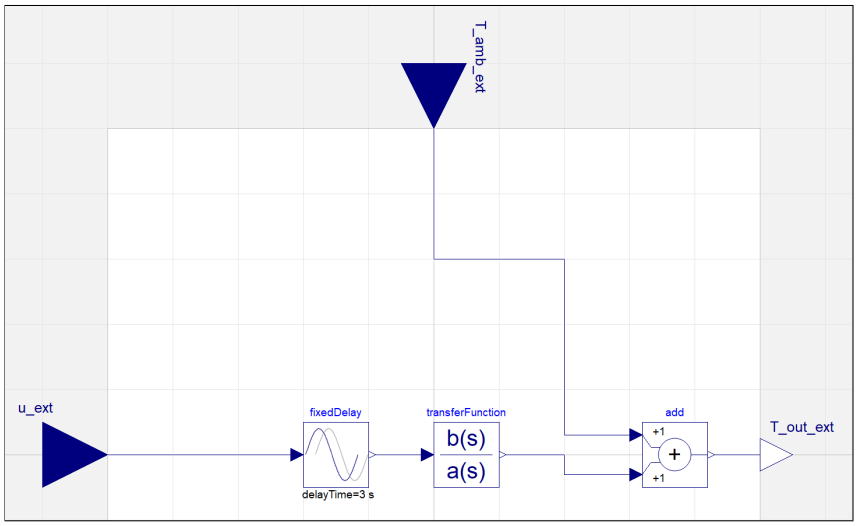


Figure 4.6: Air heater model implemented as a transfer function in Modelica.

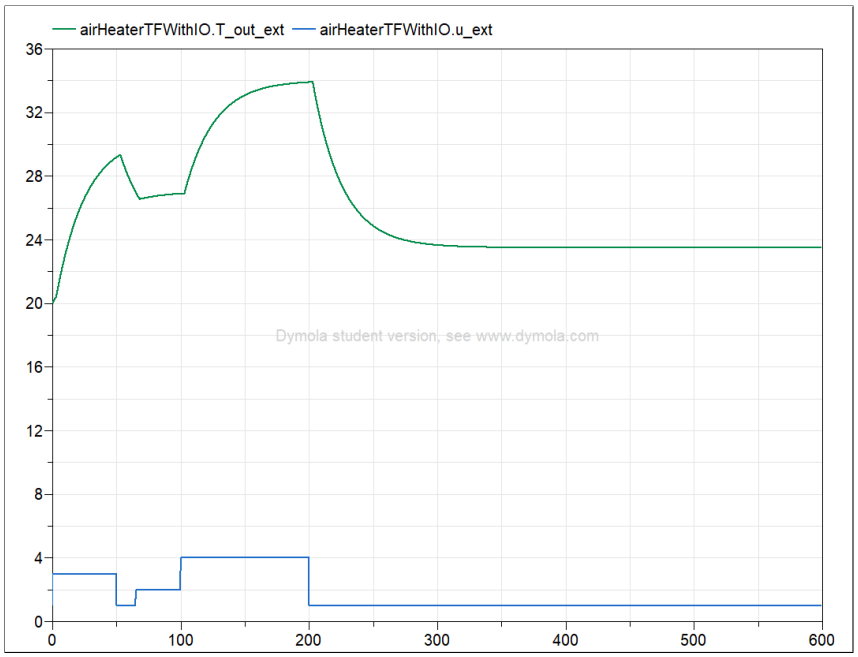


Figure 4.7: Plot from simulating transfer function model of air heater.

4.3 Model of Air Heater Based on State Space Model

The transformation of the model to state-space form was performed with the use of the Python source code that is available on GitHub: (Link to source code). The implementation utilizes the control library and the control MATLAB extension in Python, as

demonstrated in Code 4.2.

A general 1.order system was implemented, then the time delay was implemented with the use of the padé approximation. After the two transfer functions were merged with the Python series function displayed in Line 30, the transfer function model was converted to a continuous state space model in Line 36. A step response was then applied to both the transfer function and state-space models in Line 33 and 39, before the resulting difference between the two step responses was calculated in Line 42. For this current implementation both results were 0.00.

The continuous state space model was written to individual .CSV files for the A, B, C and D matrices in Line 45-48, and a discrete time state space model using Euler forward and a time-step of $0.01[s]$ was written to .CSV files as displayed in Line 53-56.

```
1  import control as ct
2  import matplotlib.pyplot as plt
3  from control.matlab import *
4  import numpy as np
5  import pandas as pd
6
7  #Process parameters and constants
8  # Process Gain[C/V]
9  K = 3.5
10
11 # Time constant [s]
12 T = 23
13
14 # Process time delay
15 tau = 3
16
17 # Order of the padé approximation
18 N = 10
19
20 #Set up transfer function to represent the time constant dynamics of the air heater
21 num = [K]
22 den = [T, 1]
23 H1 = ct.tf(num,den)
24
25 # Set up transfer function to represent the time delay of air heater
26 [num_pade, den_pade] = ct.pade(tau,N)
27 Hpade = ct.tf(num_pade, den_pade)
28
29 # Connect the time constant and time delay transfer functions together
30 H = ct.series(H1, Hpade)
31
32 # Perform a step response to the transfer function system
33 ttf,ytf = ct.step_response(H)
34
35 # Transfer the model from transfer function model to continuous state space model
36 Hss = ct.matlab.tf2ss(H, dt=0)
37
38 # Performe a step response to the state space system
39 tss,yss = ct.step_response(Hss)
40
41 # print the max and min from the difference between the step response in the transfer function and state space model.
42 print ("The maximum value is: " + str(max(ytf-yss)) + "\nThe minimum value is: " + str(min(ytf-yss) ))
43
44 # Write matrices to .CSV files
45 pd.DataFrame(Hss.A).to_csv("DataFiles/AirHeaterSS_A.csv", sep=',',columns=None, index=False, decimal='.', header=False)
```

```
46 pd.DataFrame(Hss.B).to_csv("DataFiles/AirHeaterSS_B.csv", sep=',',columns=None, index=False, decimal='.', header=False)
47 pd.DataFrame(Hss.C).to_csv("DataFiles/AirHeaterSS_C.csv", sep=',',columns=None, index=False, decimal='.', header=False)
48 pd.DataFrame(Hss.D).to_csv("DataFiles/AirHeaterSS_D.csv", sep=',',columns=None, index=False, decimal='.', header=False)
49
50 # Generate a discrete time state space model
51 HssD = Hss.sample(0.01, method='euler')
52 # Write matrices to .CSV files
53 pd.DataFrame(HssD.A).to_csv("DataFiles/AirHeaterSSD_A.csv", sep=',',columns=None, index=False, decimal='.', header=False)
54 pd.DataFrame(HssD.B).to_csv("DataFiles/AirHeaterSSD_B.csv", sep=',',columns=None, index=False, decimal='.', header=False)
55 pd.DataFrame(HssD.C).to_csv("DataFiles/AirHeaterSSD_C.csv", sep=',',columns=None, index=False, decimal='.', header=False)
56 pd.DataFrame(HssD.D).to_csv("DataFiles/AirHeaterSSD_D.csv", sep=',',columns=None, index=False, decimal='.', header=False)
```

Code 4.2: Transfer function and state space model in Python - Link to source code.

The exported .CSV files generated in the Python script in Code 4.2 was imported into Modelica with the use of a continuous and a discrete time state space model block, as displayed in Figure 4.8. The ambient temperature was added as an input to the model blocks with the use of add blocks.

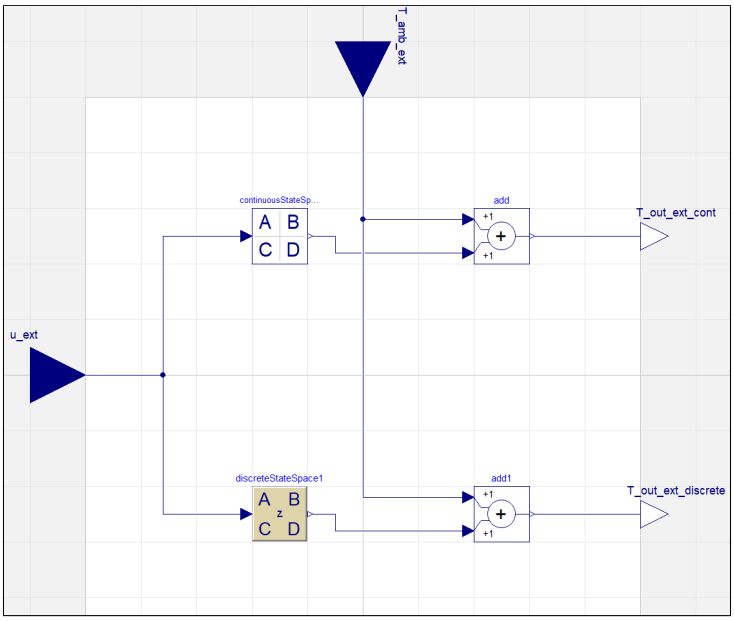


Figure 4.8: Air heater implemented as continuous and discrete time state space models.

4.4 Data Validation and Analysis

A simulation of the three blocks based on first-principles, transfer function and both state space models were set up and excited with the same control signal sequence. The implementation in Modelica is displayed in Figure 4.9.

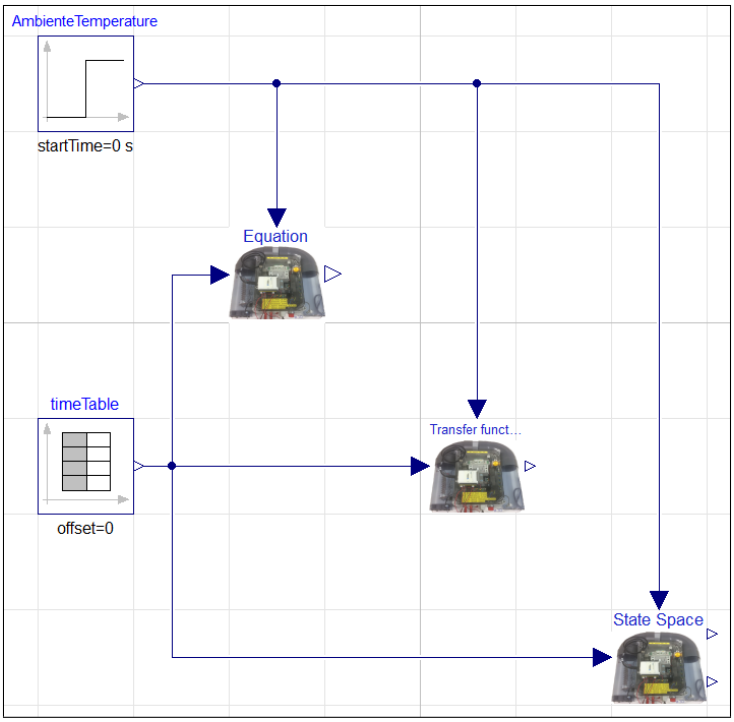


Figure 4.9: Simulation setup of air heater models based on equations, transfer functions, and state space.

The resulting plot in Figure 4.10 displays minor differences in the dynamic response. The reason for the deviation is mainly attributed to a discrepancy at $t = 0^+$, where the state-space models incorporate the time delay using the Padé approximation from $t = 0$, while the other models are initiated with initial conditions.

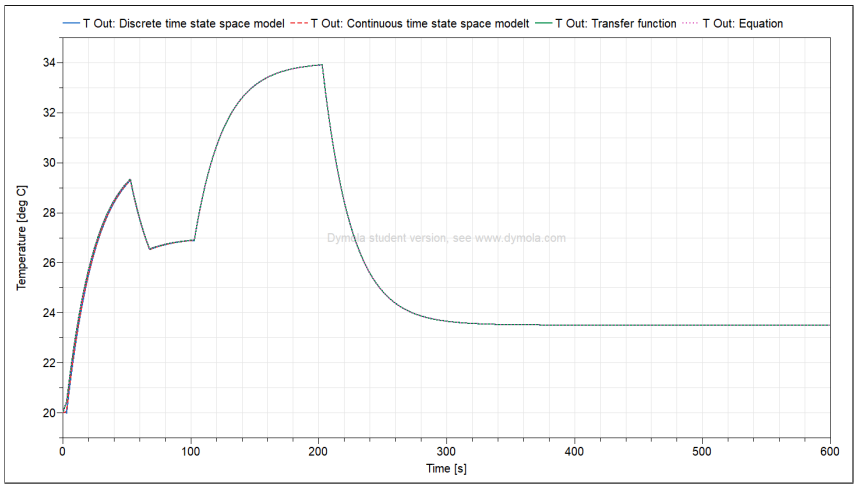


Figure 4.10: Plot of simulation setup of air heater models based on equations, transfer functions, and state space.

5 Different Methods of Calling External C Code in Modelica

This chapter presents the required setup of packages in Modelica and the implementation of two different methods for calling external functions written in C code, from an OpenModelica environment.

The two methods implemented in this chapter are illustrated in Figure 5.1.

Figure 5.1: Different ways to call external C code in Modelica.

Modelica Function Alternative

Represents the most basic implementation alternative. In many cases, it is limited to the use of a single .C file and is best suited for calling relatively small external functions. The Modelica functions cannot be utilized directly within the graphical environment but need to be encapsulated within a block or model wrapper to be callable.

Modelica External Object Alternative

Extends the functionality of external functions to include the capability to store parameters and variables in external memory. The external object incorporates a constructor

for external memory allocation and a destructor for de-allocating memory, in addition to invoking the external function. The external object can be called from both a Modelica model and a Modelica block.

5.1 Initial Setup in Modelica

There exist two options for how Modelica manages packages and files:

1. Store all packages and files in a single file.
2. Replicate the structure from Modelica, including sub-folders, on the host computer.

To replicate the structure when creating a new package in Modelica, un-check the box as shown in Figure 5.2. This approach replicates the file structure from the package on to the computers file system, enabling the option to upload files into the file structure and access it from within the OpenModelica package.

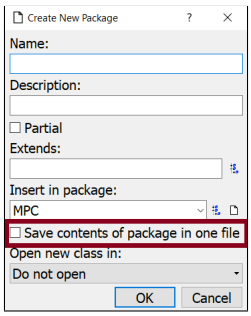


Figure 5.2: Configure Modelica folder structure.

Figure 5.3 illustrates how the file structure in the Modelica package is replicated to the computer file structure. Pictures utilized in Modelica models are stored in the 'Images' folder, while the .C files reside in the 'Source' folder. The .H files is stored within the 'Include' folder, and the .a compiled archive files are located inside the 'library/win64' folder. With this setup the file locations will be according to the Modelica standard search paths, eliminating the need to specify a file path when calling external functions.

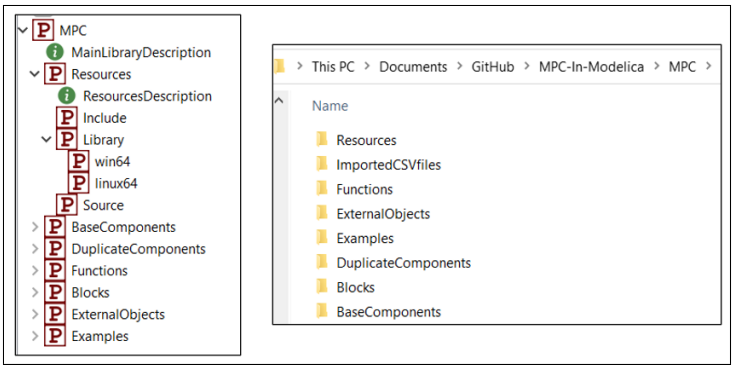


Figure 5.3: Replication of Modelica file structure on host computer.

5.2 Modelica Function

This chapter presents four stages for how the external C function was implemented in Modelica, as presented in Figure 5.4.

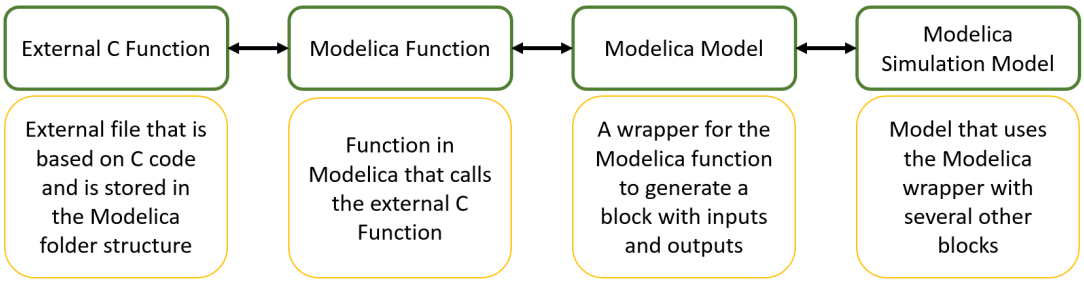


Figure 5.4: Process flow of calling external functions from Modelica.

5.2.1 External C Function

This chapter presents the C function that is called from Modelica. The function is displayed in Code 5.1 and is based on the random function 'rand()'. The function returns a pseudo random number based on a given range as input.

```
#include <stdio.h>
#include <stdlib.h>

double NoiseGenerator(double range)
{
  double result = 0;
  double span = 2*range;

  //Only return the noise component
  result = ((double)(rand() /(RAND_MAX/(span)))-range);
```

```
11
12    return result;
13
14  }
```

Code 5.1: Noise Generator Implemented in C - Link to source code.

5.2.2 Modelica Function

Since external functions cannot be called directly from a Modelica model it is required to wrap the external function call into a Modelica function. The Modelica function is stored in the 'Functions' folder, as displayed in Figure 5.3. The Modelica function code for calling the external function is presented in Code 5.2, where the inputs and outputs are defined, along with the external command that describes the external language and the function name, including both the return value variable and the input value variable.

The 'IncludeDirectory' in Line 8 references the relative file path, with reference to the root package named MPC, while 'NoiseGenerator.C' in Line 9 is the file name residing inside the source folder.

```
1  function extNoiseGenerator "Function that call external c code to calculate random number"
2
3    //Function parameters
4    input Real range     "Total range of noise to be generated centered around input signal";
5    output Real y_ext    "Distorted output signal";
6
7    external "C" y_ext = NoiseGenerator(range)
8      annotation (IncludeDirectory = "modelica://MPC/Resources/Source/",
9                  Include="#include \"NoiseGenerator.c\"");
10
11 end extNoiseGenerator;
```

Code 5.2: Calling External Function in Modelica - Link to source code.

5.2.3 Modelica Function Call Wrapper

The function presented in Code 5.2 is implemented into the Modelica model in Code 5.3. From the code, an external connector is observable in Line 8, enabling the passing of generated noise from the model as an output. Additionally, a sample period of 1.0 has been included within a 'WHEN' statement to determine how often the Modelica function calls the external function.

```
model NoiseGenerator "Simple random noise generator"

  //Parameters
  parameter Real range = 0.5;
  parameter Real samplePeriode = 1.0;

  //External connectors
  Modelica.Blocks.Interfaces.RealOutput noiseGenerator_out
    annotation (Placement(transformation(extent={{100,-10},{120,10}})));

initial equation
  noiseGenerator_out = Functions.extNoiseGenerator(range);

equation
  when sample(0, samplePeriode) then
        noiseGenerator_out = Functions.extNoiseGenerator(range);
  end when;

end NoiseGenerator;
```

Code 5.3: Wrapper for Modelica Function - Link to source code.

5.2.4 Modelica Simulation Model

Figure 5.5 presents a Modelica model where the measurement noise is generated from the external function in the Noise Generator sub-model. The noise is added to the process output measurement by a Modelica add block.

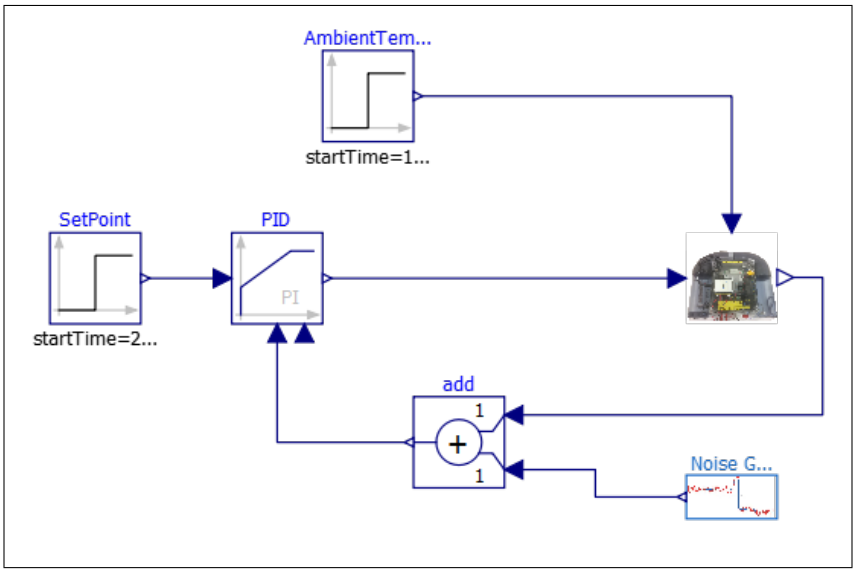


Figure 5.5: Model for simulating noise generated from external C code.

A plot of the process output measurement with and without the added noise is displayed in Figure 5.6. Since the model does not include any filtering the signal noise generated for

each time step is either accumulated or equalized based on the phase differences between the random generated noise and the process measurement.

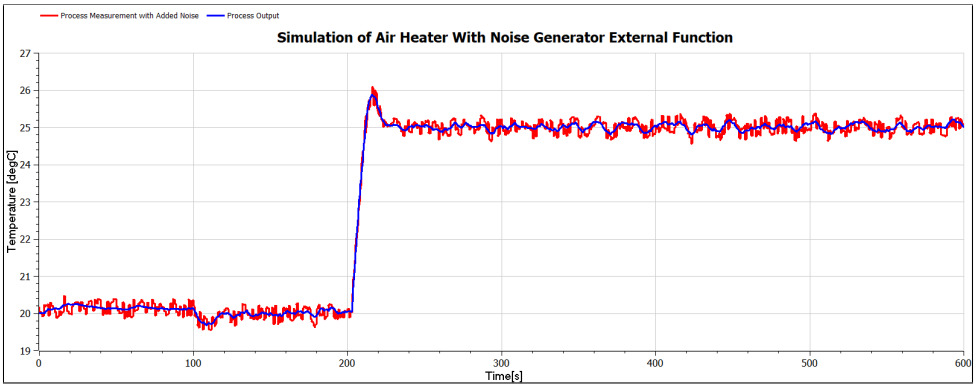


Figure 5.6: Plot of process output with both existing and newly added noise.

5.3 Modelica External Objects

One of the differences between using the Modelica model and a Modelica block is that a block requires that all inputs and outputs have prefixes for all connector variables. The Modelica block lacks internal states or timing conditions, only invoking the C function during block processing.

In Figure 5.7 a sequence diagram, illustrates the fundamental method for implementing external objects in Modelica. The sequence diagram, referencing points 1 to 5, demonstrates the construction of both a Modelica record and a C struct. These structures are utilized to share data between the Modelica and C environments by accessing the same pre-allocated memory space on the computer.

Within the loop in the sequence diagram it is observable that the external C function responsible for generating the noise is accessed and data is communicated back to the Modelica environment. After the Modelica code is finished the external object is destroyed as displayed in point 11 to 13 in Code 5.7.

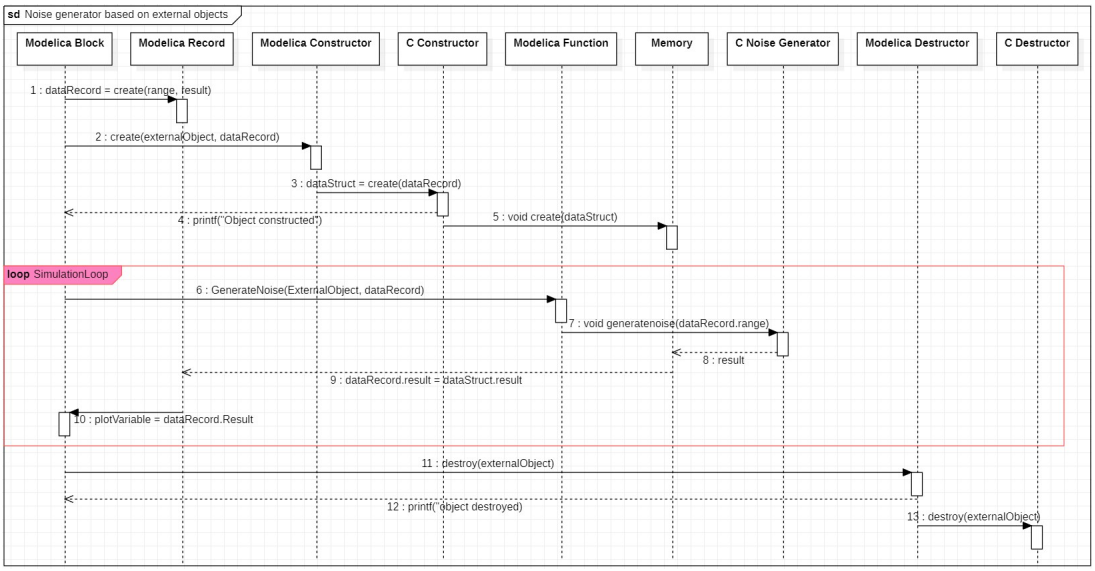


Figure 5.7: Sequence diagram for calling external noise generator with the use of Modelica external objects.

5.3.1 Modelica Record and C Struct

The Modelica record was set up as displayed in Code 5.4. The naming of the variables for this record are the same as for the struct in C source Code 5.5.

```
1 record NoiseGeneratorData
2   Real range "Range for noise generation";
3   Real result "Resulting generated noise signal";
4   end NoiseGeneratorData;
```

Code 5.4: Record for storing data - Link to source code.

The corresponding struct in C is presented in Code 5.5.

```
1 typedef struct{
2   double range;
3   double result;
4 }NoiseGeneratorData;
```

Code 5.5: Struct for storing data - Link to source code.

5.3.2 External Object Class - Constructor and Destructor in Modelica and C

The source code for the constructor is presented in Code 5.6 where the extension of the Modelica external objects class in Line 2, is observable. Further the constructor passes

values from the Modelica block input in rangeInn with defined start value and initializes the result variable into the C function call to initialize the C struct with the same values as for the record.

```
1   class NoiseGeneratorExternalObject
2     extends ExternalObject;
3
4     function constructor
5       output NoiseGeneratorExternalObject noiseGeneratorExternalObject;
6       input NoiseGeneratorData noiseGeneratorData(range = rangeInn, result = 0.0);
7
8       // External C function call to initialize the noise generator.
9   external "C" noiseGeneratorExternalObject = initialiseNoiseGenerator(noiseGeneratorData.range,
10                                                                        noiseGeneratorData.result)
11
12      annotation(IncludeDirectory = "modelica://MPC/Resources/Include/",
13                 Include = "#include \"NoiseGenerator.c\"");
14    end constructor;
```

Code 5.6: Constructor of external object in Modelica - Link to source code.

The corresponding constructor in C being called from the Modelica environment is displayed in Code 5.7. The 'malloc()' function in Line 4 allocates space in memory with the size of the number of bytes in the 'NoiseGeneratorData' struct and returns a pointer to the beginning of the allocated memory block. In Line 9 and 10 the initial values are passed from the Modelica environment to the struct. Line 13 is seeding of the random generator with the value of the Unix time, improving the randomness of the function. In Line 15 there is a print statement that will display the text in the OpenModelica terminal window when the code has successfully been executed.

```
1     void* initialiseNoiseGenerator(double range, double result)
2   {
3     // Allocate memory for the optimization data input
4       NoiseGeneratorData* noiseGeneratorDataInput = malloc(sizeof(NoiseGeneratorData));
5     if (noiseGeneratorDataInput == NULL)
6       ModelicaError("Insufficient memory to allocate noiseGeneratorDataInput");
7
8     // Initialize the optimization data input
9     noiseGeneratorDataInput->range = range;
10    noiseGeneratorDataInput->result = result;
11
12      // Seed the random number generator
13      srand(time(NULL));
14
15    printf("Initialization of input successful! \t");
16    return (void *)noiseGeneratorDataInput;
17  }
```

Code 5.7: Constructor of external object in C - Link to source code.

The destructor of the external object is displayed in Code 5.8. The only argument passed into the destructor is the external object from Line 2 in the source code.

```
1     function destructor
2       input NoiseGeneratorExternalObject noiseGeneratorExternalObject;
3
4       external "C" closeNoiseGenerator(noiseGeneratorExternalObject)
5       annotation(IncludeDirectory = "modelica://MPC/Resources/Include/",
6                  Include = "#include \"NoiseGenerator.c\"");
7
8     end destructor;
9   end NoiseGeneratorExternalObject;
```

Code 5.8: Destructor of external object - Link to source code.

The corresponding C source code for the destructor is given in Code 5.9. After the memory has been successfully freed in Line 6, there is a print statement that writes to the terminal in OpenModelica in Line 7.

```
1  void closeNoiseGenerator(void *externalObject)
2  {
3      NoiseGeneratorData* noiseGeneratorDataInput = (NoiseGeneratorData *)externalObject;
4    if(noiseGeneratorDataInput != NULL)
5    {
6      free(noiseGeneratorDataInput);
7          printf("Destruction of input successful!\t");
8    }
9  }
```

Code 5.9: Destructor of external object in C - Link to source code.

5.3.3 Modelica Function Call

When calling external C code with a Modelica function where there is a requirement to return more than one variable, the previously defined record in Code 5.4 can be defined as an output in the Modelica function but passed into the function call as an argument as displayed in Line 7 in Code 5.10. This will update the Modelica record based on the C struct for each function call.

```
1   function noiseGenerationCall
2   input NoiseGeneratorExternalObject noiseGeneratorExternalObject;
3   output NoiseGeneratorData noiseGeneratorData;
4
5   // External C function call to generate the noise.
6           external "C" NoiseGenerator(noiseGeneratorExternalObject,
7                                       noiseGeneratorData)
8           annotation(IncludeDirectory = "modelica://MPC/Resources/Include/",
9                      Include = "#include \"NoiseGenerator.c\"");
10
11  end noiseGenerationCall;
```

Code 5.10: Modelica function that calls external C code - Link to source code.

The corresponding C function that is called from the Modelica function and generates the noise is presented in Code 5.11. The function takes in two void objects in Line 1 being the external object and 'noiseGeneratorData' record from OpenModelica. Based on the two external objects there is generated two objects inside the C function where one of the objects references the input values passed inn from the Modelica function, and the second is the return object reference set up to return calculated values to the Modelica environment. In Line 10 it can be observed that the output object is updated with a new calculated random number based on the range that was given as an input to the function call.

```
1  void NoiseGenerator(void *externalObject, void *externalObject2)
2  {
3    NoiseGeneratorData* noiseGeneratorDataInput = (NoiseGeneratorData *)externalObject;
4    NoiseGeneratorData* noiseGeneratorOutput = (NoiseGeneratorData *)externalObject2;
5
6    double random = 0;
7    random = ((double)rand() / (double)RAND_MAX) - 0.5;
8
9    //Only return the noise component
10   noiseGeneratorOutput->result = random * noiseGeneratorDataInput->range;
11 }
```

Code 5.11: Noise generator function in C code - Link to source code.

5.4 Data Validation and Analysis

Figure 5.8 presents the plot from both implementation methods discussed in Chapter 5.2 and Chapter 5.3. Both noise generator implementations were conducted within a similar model setup as shown in Figure 5.5. Both models were executed in OpenModelica for 600 seconds, utilizing 1200 intervals and the DASSL solver. The figure illustrates the output of the air heater and the signal after the addition of the noise signal.

The data in Figure 5.8 was exported as .CSV files and imported into a Jupyter notebook, using the source code available in the GitHub repository: (Link to source code). By adding data points to the plot for each sample and plotting a total time interval of $5[s]$ seconds a distinction between the output of the external object simulation in red and the function-based simulation in green is observable. The primary difference between these simulations lies in their execution mechanisms where the external object was implemented using the Modelica algorithm, executing code sequentially at specific time steps with a frequency of $0.5[s]$. This frequency is determined by the simulation duration of 600

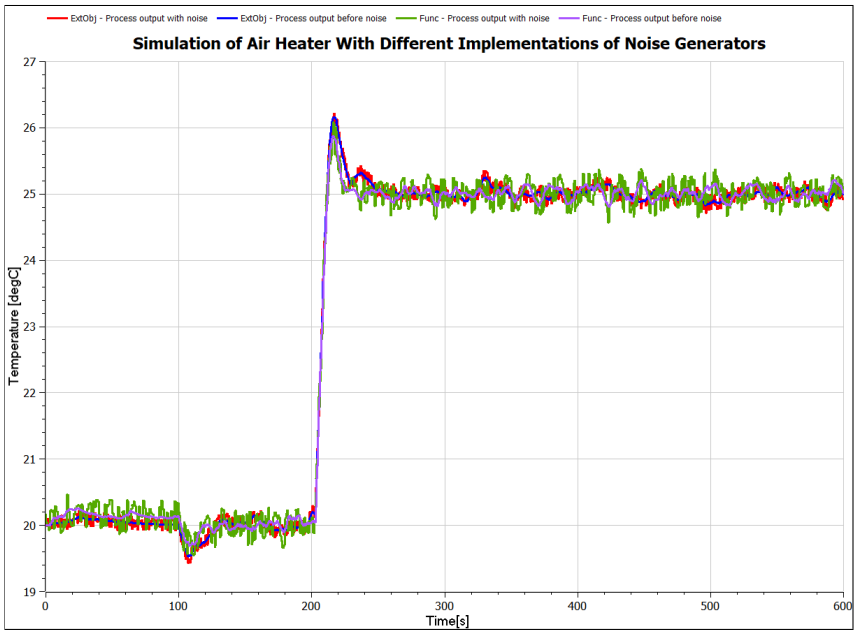


Figure 5.8: Simulation of PI-controller with the different implementations of the noise generator.

seconds and the number of intervals set to 1200. On the other hand, the equation-based implementation utilized in the Modelica function relies on intervals chosen by OpenModelica.

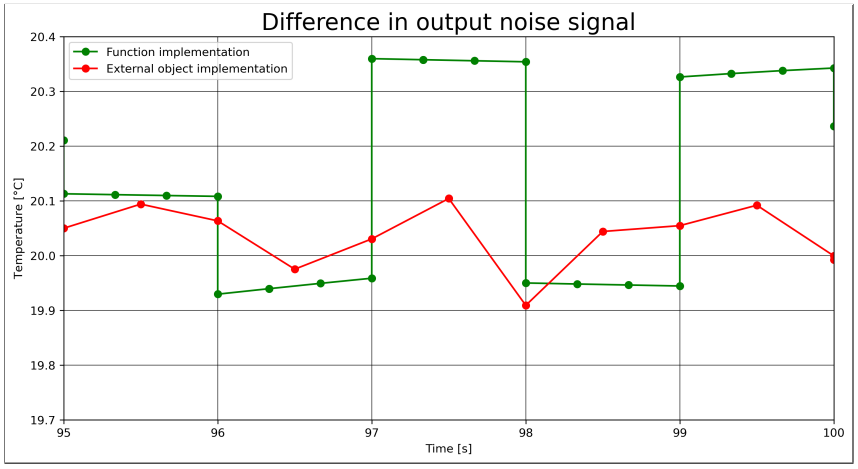


Figure 5.9: Simulation result of PI-controller with all the different implementations in a time interval of $10[s]$.

Table 5.1 displays that the equation-based simulation generated a total of 2926 points, while the algorithm based external object generated a total of 1208 points. The difference in the remaining statistical measures is quite small and a result of rounding.

Table 5.1: Statistical measures for both models.

Variable	Function	External Object
Data points:	2926	1208
Mean:	23.4	23.3
Standard deviation:	2.3	2.4
Min:	19.5	19.4
Max:	26.1	26.2

6 Calling NLopt as External Object in OpenModelica

The chapter presents the setup of the NLopt library on a Windows computer, and the data flow between the NLopt library implemented as an external object inside an OpenModelica block. This chapter further explores three different implementations, each with increased complexity.

Figure 6.1 presents the basic function of the three different blocks implemented in OpenModelica. The first block starting from the left is a basic function that was set up to serve as a test to validate the installation of the NLopt library and the flow of data between the OpenModelica and the C environment. The second and third blocks execute univariate and multivariate optimization, respectively.

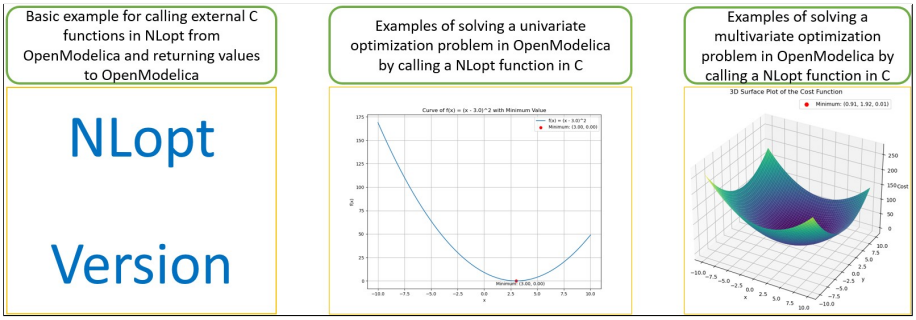


Figure 6.1: Three different blocks implemented in OpenModelica.

6.1 Installation of the NLopt Library on Windows

The NLopt library can be installed using vcpkg, a C/C++ package manager that provides access to over 1500 open-source libraries, including NLopt. The home page of vcpkg with both installation description and search function for the available libraries can be found at this: (Link).

Assuming Git is installed on the computer, the installation process begins by using Windows PowerShell to clone the vcpkg GitHub repository into the local computer. Recommended paths for cloning include the following:

```
1    C:\src\vcpkg
2    C:\dev\vcpkg
```

Clone the repository to the current directory with this command:

```
1    git clone https://github.com/Microsoft/vcpkg.git
```

Build the vcpkg with bootstrap:

```
1    .\vcpkg\bootstrap-vcpkg.bat
```

Install the NLopt library for Windows 64bit with:

```
1    .\vcpkg install nlopt:x64-windows
```

With vcpkg installed in the root directory, the NLopt library given as a .DLL file can be found in the path from Line 1 in the example below. Line 2 provides a reference to the Modelica implementations presented in the GitHub repository, indicating where the .DLL file needs to be moved.

```
1    C:\vcpkg\installed\x64-windows\bin\nlopt.dll
2    <path>\MPC\Resources\Library\win64
```

The NLopt .H file must be copied from the installed NLopt library with the path displayed in Line 1 below, to the Modelica folder given by the reference path in Line 2.

```
1    C:\vcpkg\installed\x64-windows\include\nlopt.h
2    <path>\MPC\Resources\Include
```

6.2 Implementation of the NLopt Version Block

The NLopt version block serves as a test function to validate the functionality of the NLopt library implementation, and the correctness of the data received by the record in OpenModelica. The source code for this implementation closely resembles the one introduced in the external object implementation in Chapter 5.3. The complete Modelica block, including the record, external object class, and the Modelica function that calls the external C function, can be found in the GitHub repository at the following link: (Link to repository). The corresponding C source code is available in the repository at this link: (Link to source code).

In Code 6.1, the definition of the library and include directory is evident from Lines 2 and 4, even though they are part of the standard search path for OpenModelica, the path is added for clarity. To access the NLopt library, the call of the external function must access both the nlop.DLL file and the nlopt.H file. It is worth noting that the header file includes the .H extension, while the library file does not include the .DLL extension in the statements from Line 3 and 5.

```
1  external "C" nloptVersion(nLoptVersionExternalObject, nloptversion2Struct) annotation(
2      LibraryDirectory = "modelica://MPC/Resources/Library/win64/",
3      Library = "nlopt",
4      IncludeDirectory = "modelica://MPC/Resources/Include/",
5      Include = "#include \"nlopt.h\"");
```

Code 6.1: External function call with NLopt - Link to source code.

The C source code features a counter implemented to output a restricted number (in this case, two) of results from the iterations back to the OpenModelica terminal window. These results provide information on the creation and removal of the external object, along with details on the number of iterations and the outcomes achieved during each iteration, as shown in Figure 6.2. This implementation primarily serves debugging purposes.

```
The initialization finished successfully without homotopy method.
### STATISTICS ###
The simulation finished successfully.
Initialisation successful!  nloptVersion function run 1 of 2 - nloptVersion is 2.7.1 -
 nloptVersion function run 2 of 2 - nloptVersion is 2.7.1 - Destruction successful!
```

Figure 6.2: Output in OpenModelica terminal after execution of block.

The results obtained from the simulation in OpenModelica are presented in Figure 6.3. The figure displays the variables extracted from the record in OpenModelica, each associated with the corresponding version number of the installed package.

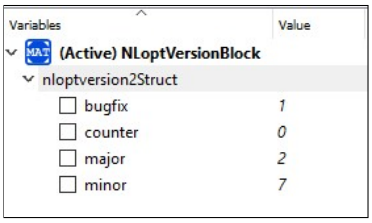


Figure 6.3: Output in OpenModelica simulation environment.

6.3 Implementation of the NLopt Univariate Optimization

The NLopt version function introduced in Chapter 6.2 is in this implementation extended to enable the passing of parameters from the OpenModelica environment into the optimization environment in the C source code. Additional checks were incorporated into the C source code to assess the values supplied to the environment, mitigating the risk of errors in the optimization process.

This chapter references one OpenModelica implementation, and one C implementation. The complete source code for the Modelica block including the records, external object class, and the Modelica function that calls the external C function, can be found in a GitHub repository at the following link: Link to source code. While the implemented C source code with the struct, constructor, destructor and optimization call can be found in the GitHub repository at the following link: Link to source code.

6.3.1 Objective Function

The univariate objective function implemented in the C source code is presented in Equation 6.1. The mathematical function was implemented as part of the C optimization function, which is again called from the Modelica function.

$$f(x) = (x - 3.0)^2 \tag{6.1}$$

A reference optimization was performed utilizing the SciPy library in Python. The resulting plot is displayed in Figure 6.4, illustrating the optimized value with the lower bound at -5 and the upper bound at 5. The resulting calculated optimized value was 3.00.

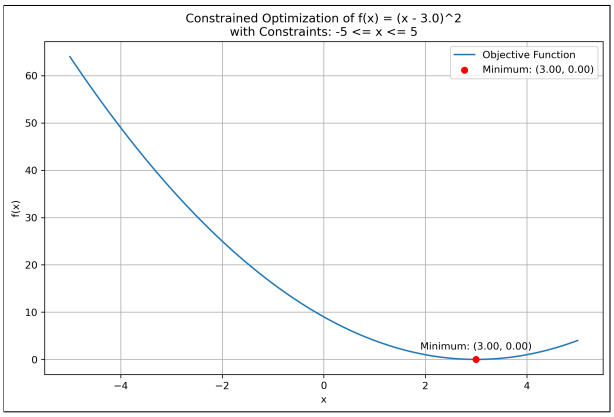


Figure 6.4: Plot of optimized univariate function - Link to source code.

6.3.2 Optimization Parameters

In Figure 6.5 the diagram displays the path of flow for the parameters from the graphical user interface in OpenModelica to the parameters being stored in the computer memory. This chapter will present how this flow was implemented across OpenModelica and C.

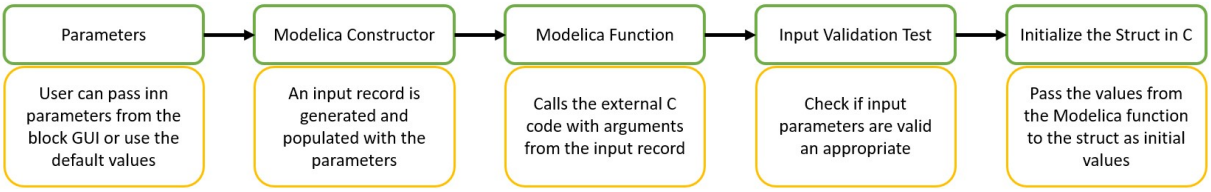


Figure 6.5: Diagram of flow for parameter from GUI all the way to allocated space in memory.

Parameters

The 'NloptUniOptiBlock' was built in OpenModelica with the parameter definition and the initial values as displayed in Code 6.2.

```
1  //Parameters
2    parameter Real x1Lb = -5.0 "Lower bound of x1";
3    parameter Real x1Ub = 5.0 "Upper bound of x1";
4    parameter Integer n = 1 "Number of optimization variables";
5    parameter Real Tol = 1e-6 "Optimizer termination tolerance";
6    parameter Integer max_iter = 100 "Maximum number of iterations for the optimizer";
```

Code 6.2: Definition of parameters in OpenModelica - Link to source code.

The window presenting the graphical user interface of the block in OpenModelica is displayed in Figure 6.6.

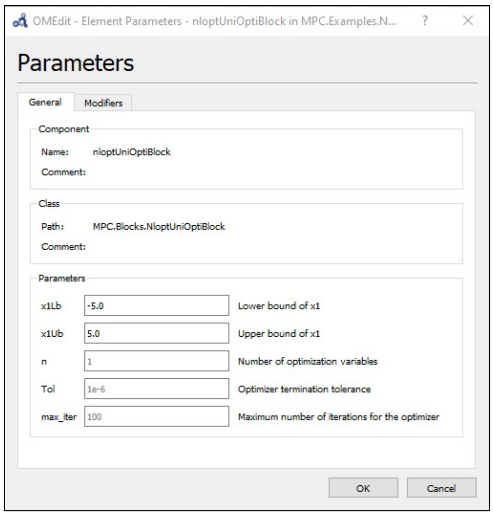


Figure 6.6: Available parameters for the Modelica block.

Modelica Constructor

Code 6.3 presents the initial portion of the OpenModelica implementation of the constructor function within the external object class. Line 5 displays the creation of a record labeled 'optimizationDataInput'. This record is set up with values originating from the parameters in Code 6.2.

```
1 function constructor
2   output NloptUnivariateEO nloptUnivariateEO;
3
4   // Initialize the data struct for shared data between C and Modelica.
5   input OptimizationData optimizationDataInput(x1R = 0, x1LbR = x1Lb, x1UbR = x1Ub, min_costR = 0, nR = n, TolR = Tol,
      max_iterR = max_iter);
```

Code 6.3: Modelica constructor record - Link to source code.

Modelica Function

The second part of the constructor function implemented in OpenModelica is displayed in Code 6.4. In Line 2 the values from the previously constructed record in Code 6.3 are being utilized as arguments when calling the external constructor function called 'initialiseUniNloptInput()' in the C code.

```
1   // External C function call to initialize the NloptUniOptimize input.
2       external "C" nloptUnivariateEO = initialiseUniNloptInput(optimizationDataInput.x1R, optimizationDataInput.x1LbR,
        optimizationDataInput.x1UbR, optimizationDataInput.min_costR, optimizationDataInput.nR, optimizationDataInput.TolR,
        optimizationDataInput.max_iterR)
3       annotation(IncludeDirectory = "modelica://Resources/Include/",
```

```
4                   Include = "#include \"nloptUniOptimize.c\"");
5       end constructor;
```

Code 6.4: Modelica constructor function call - Link to source code.

Input Validation test

The source code in Code 6.5 illustrates tests that were implemented for evaluating the values utilized to initialize the C struct, used when constructing the external object. The 'ModelicaError()' function is a pre-defined Modelica function designed to report error messages back to the Modelica environment.

```
1  // Check input
2    if (x1Lb > x1Ub)
3      ModelicaError("x1Lb must be smaller than x1Ub");
4    if (n <= 0)
5      ModelicaError("n must be larger than 0");
6    if (tol <= 0)
7      ModelicaError("tol must be larger than 0");
8    if (max_iter <= 0)
9      ModelicaError("max_iter must be larger than 0");
```

Code 6.5: Test of input values for initialization of struct - Link to source code.

Initialize the Struct in C

The type definition 'typedef()' of the struct is presented in Code 6.6.

```
1  typedef struct {
2      double x1;          // Optimization variable
3      double x1Lb;        // Lower bound of x1
4      double x1Ub;        // Upper bound of x1
5      double min_cost;    // Minimum value of the objective function after optimization
6      int n;              // Number of optimization variables
7      double tol;         // Termination tolerance
8      int max_iter;     // Maximum number of iterations for the optimizer
9      } OptimizationDataUni;
```

Code 6.6: Definition of struct - Link to source code.

In Code 6.7 there is allocated a space in memory with the function 'malloc()', used in Line 2. The size of the allocated space is the same as required for the 'OptimizationDataUni' struct displayed in Code 6.6. The values passed into the C constructor function from OpenModelica are written to the instance of the 'OptimizationDataUni' object named 'optimizationDataInput' from Line 2. Finally, the constructor returns a 'void' pointer to the memory space allocated by the 'malloc()' function.

```
1   // Allocate memory for the optimization data input
2     OptimizationDataUni* optimizationDataInput = malloc(sizeof(OptimizationDataUni));
3   if (optimizationDataInput == NULL)
4     ModelicaError("Insufficient memory to allocate optimizationDataInput");
5
6   // Initialize the optimization data input
7   optimizationDataInput->x1 = 0;
8   optimizationDataInput->x1Lb = x1Lb;
9   optimizationDataInput->x1Ub = x1Ub;
10  optimizationDataInput->min_cost = 0;
11  optimizationDataInput->n = n;
12  optimizationDataInput->tol = tol;
13  optimizationDataInput->max_iter = max_iter;
14
15  printf("Initialisation of input successful! \t");
16  return (void *)optimizationDataInput;
```

Code 6.7: Allocate memory for struct - Link to source code.

6.3.3 Implemented NLopt Optimization in C

The source code for the implemented C function defining the optimization objective function and process of optimization can be found in a GitHub repository at the following link: Link to source code. This chapter will present and comment on elements of the source code.

Code 6.8 presents the head of the optimization function implemented in C. In Line 1 it can be observed that the C function takes in two arguments that are pointers to objects of type 'OptimizationDataUni' and does not return any values since it writes the values directly to memory.

```
1 void mainFunctionUni(void *externalObject, void *externalObject2){
2   OptimizationDataUni* optimizationDataInput = (OptimizationDataUni *)externalObject;
3   OptimizationDataUni* optimizationDataOutput = (OptimizationDataUni *)externalObject2;
```

Code 6.8: Main optimization function objects - Link to source code.

Code 6.9 presents the implemented C function for the objective function to be optimized. A plot of the objective function is presented in Figure 6.4.

```
1 // Objective function for optimization
2 double objective(unsigned n, const double* x, double* grad, void* data)
3 {
4   // Compute the objective value based on the input variable `x`
5   double result = pow(*x - 3.0, 2);
6   return result;
7 }
```

Code 6.9: Optimization function - Link to source code.

Line 2 in Code 6.10 displays the initialization of the optimization object based on the COBYLA optimizer, and variable 'n' passed from OpenModelica as a parameter representing the number of optimization variables. In Line 5 the objective function is defined together with the optimization object.

```
1   // Create an NLOpt optimizer
2     nlopt_opt optimizer = nlopt_create(NLOPT_LN_COBYLA, optimizationDataInput->n); // Use the LN_COBYLA algorithm
3
4     // Set the objective function
5     nlopt_set_min_objective(optimizer, objective, NULL);
```

Code 6.10: Creation of optimization object - Link to source code.

Code 6.11 illustrates the implementation of the optimization, where the results from the optimization and the minimum cost is written straight to the allocated space in memory.

```
1     // Optimize the problem
2     nlopt_optimize(optimizer, &x, &optimizationDataInput->min_cost);
```

Code 6.11: Optimize the problem - Link to source code.

When the optimization is completed the optimization object is destroyed as displayed in Code 6.12.

```
1     // Destroy the optimizer
2     nlopt_destroy(optimizer);
```

Code 6.12: Destroy the optimization object - Link to source code.

6.3.4 Simulation Results in OpenModelica

Figure 6.7 presents the simulation results after running the 'NloptUniOptiBlock' in OpenModelica. From the record containing both input and output values, a separate record called 'summary' was created so the user can utilize the search function and retrieve the most relevant values more efficiently. The resulting values of the optimization are $x = 3$ and the $min\_cost = 0$ these values match those calculated in the Python implementation from Figure 6.4.

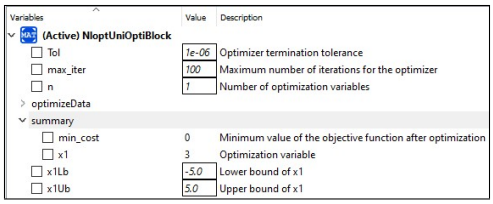


Figure 6.7: Simulation results from the NLopt univariate block.

6.4 Implementation of the NLopt Multivariate Optimization

The implementation of the multivariate optimization is an extension from the univariate implementation from Chapter 6.3, with the addition of one optimization variable and two additional constraints. This chapter presents the differences in implementation between the univariate and multivariate implementations.

The source code for the Modelica block implemented in OpenModelica is available in a GitHub repository at the following link: Link to source code. While the C source code including the optimizer, can be found in a GitHub repository at this link: Link to source code.

6.4.1 Objective Function

The multivariate objective function implemented in the C source code is presented in Equation 6.2, where x represents a vector with variables x_0 and x_1.

$$f(x) = (x_0 - 2)^2 + (x_1 - 3)^2 + 3 \tag{6.2}$$

Figure 6.8 displays a plot generated by performing the optimization in Python utilizing the SciPy library. The lower bound was set to -5 and upper bound to 5 for both parameters. The resulting value for the optimal solution is $x_0 = 2.00$ and $x_1 = 3.00$ with the minimum value of the objective function calculated to be 3.00.

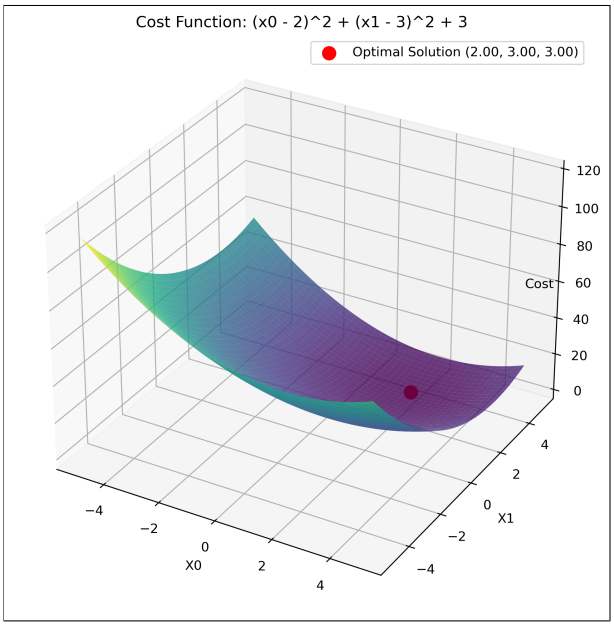


Figure 6.8: Plot of optimized multivariate function - Link to source code.

6.4.2 Optimization Parameters

Due to the absence of support for passing vectors between C and Modelica, the values are written as separate variables into the OpenModelica record utilizing pointers to the struct as displayed in Code 6.13.

```
1  //Passing returnvalues back to Modelica
2  optimizationDataOutput->x1 = x[0];
3  optimizationDataOutput->x2 = x[1];
4  optimizationDataOutput->min_cost = optimizationDataInput->min_cost;
```

Code 6.13: Return optimized values as array element - Link to source code.

Input Validation Test

Code 6.14 includes tests to evaluate the values utilized to initialize the C struct.

```
1  // Check input
2  if (tol <= 0)
3    ModelicaError("The tolerance needs to be > 0");
4  if (max_iter <= 0)
5    ModelicaError("The maximum number of iterations needs to be > 0");
6  if (x1Lb >= x1Ub)
7    ModelicaError("The lower bound of x1 needs to be smaller than the upper bound");
8  if (x2Lb >= x2Ub)
9    ModelicaError("The lower bound of x2 needs to be smaller than the upper bound");
10 if (n <= 0)
11   ModelicaError("The number of variables needs to be > 0");
```

```
12    if (n != 2)
13      ModelicaError("This example only works for two variables");
```

Code 6.14: Test of input values for initialization of struct - Link to source code.

6.4.3 Implemented NLopt Optimization in C

Objective Function

In Code 6.15 there is a presentation of the implemented objective function to be optimized. When handling optimization of multiple variables the calculated return output is a vector.

```
1  // Objective function for the optimization problem
2  double objective(unsigned n, const double *x, double *grad, void *data)
3  {
4      // Compute the cost function (e.g., quadratic cost)
5    double cost = (x[0] - 2) * (x[0] - 2) + (x[1] - 3) * (x[1] - 3) + 3; //Non-zero
6
7    return cost;
8  }
```

Code 6.15: Multivariate objective function - Link to source code.

6.4.4 Simulation Results in OpenModelica

The simulation results obtained by running the model in OpenModelica are displayed in Figure 6.9. The results present relatively minor differences when compared to the Python simulation results in Figure 6.8, this probably caused by floating-point round-off errors.

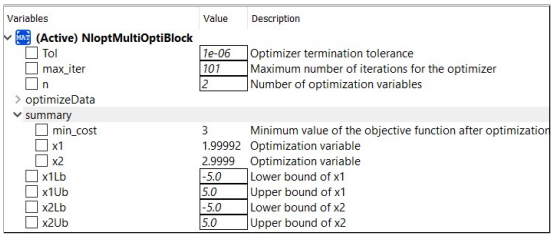


Figure 6.9: Simulation results from the NLopt multivariate block.

7 FMU of Air Heater in Python

This chapter presents the process for importing the exported FMUs into Python and conducting performance comparisons. The chapter contains portions of the source code from the Python implementation using Jupyter Notebook, while the complete source code can be found in the GitHub repository: (Link to repository).

The implementations in this chapter are based on the FMPy example files references in Chapter 3.7. Figure 7.1 provides a visual overview of the setup for evaluating the various exported FMUs.

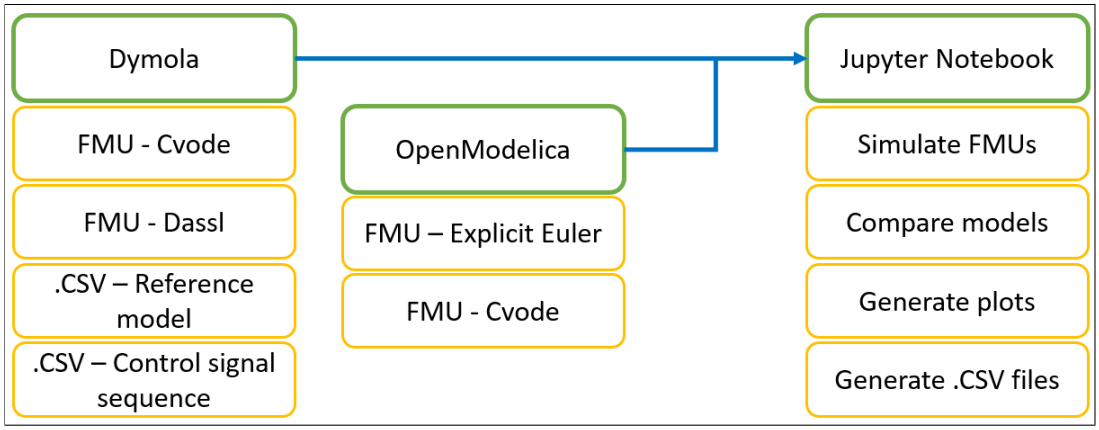


Figure 7.1: Overview of process for generating FMUs and evaluating the FMUs in Python.

The simulation of the reference model of the air heater simulated in Dymola resulted in an .CSV file export. This exported .CSV file serves as a reference for comparison with four other exported FMUs, all based on the same air heater model. The four different FMUs compared to the reference were:

1. OpenModelica - Explicit Euler

2. OpenModelica - Cvode

3. Dymola - Cvode

4. Dymola - Dassl

7.1 Implementation in Python

Code 7.1 illustrates the process of importing the four FMUs into the Python environment. The code includes the setup of initial values for the simulation, the definition of variables to be recorded, and the printing of model information for the various FMUs. Detailed model information for the FMUs is available in the complete source code from the GitHub repository: (Link to source code).

```
import fmpy
import matplotlib.pyplot as plt
import numpy as np
from fmpy import *

# Import FMU model generated with explicit Euler in OpenModelica into a variable
airHeaterOMexpEul = 'AirHeaterFMUexpEul/MPC.DuplicateComponents.AirHeaterWithIO.fmu'

# Import FMU model generated with Cvode in OpenModelica into a variable
airHeaterOMcvode = 'AirHeaterFMUcvode/MPC.DuplicateComponents.AirHeaterWithIO.fmu'

# Import FMU model generated with Cvode in Dymola into a variable
airHeaterDymolaCvode = 'AirHeaterFMUdymolaCvode/MPC_DuplicateComponents_AirHeaterWithIO_64bit_COVDESolver.fmu'

# Import FMU model generated with Dymola Dassl solver in Dymola into a variable
airHeaterDymolaOEM = 'AirHeaterFMUdymolaSolver/MPC_DuplicateComponents_AirHeaterWithIO_64bit_DymolaSolvers.fmu'

# Set the initial values for the inputs
start_values= { 'u_ext': 3.0,
                'T_amb_ext': 20}

# Choose what variables that will be available in plot and export
output = [
    'T_Out_ext',
    'u_ext',
    'T_amb_ext'
]

# Display the FMU model information
print("FMU model generated with explicit Euler in OpenModelica into a variable")
dump(airHeaterOMexpEul)
print("\n FMU model generated with Cvode in OpenModelica into a variable")
dump(airHeaterOMcvode)
print("\n FMU model generated with Cvode in Dymola into a variable")
dump(airHeaterDymolaCvode)
print("\n FMU model generated with Dymola Dassl solver in Dymola into a variable")
dump(airHeaterDymolaOEM)
```

Code 7.1: Import of FMU into Python - Link to source code.

In the source code provided in Code 7.2, from Line 2 the control signal sequence is imported from a .CSV file that originated during the simulation of the reference model in Dymola. This control signal file is utilized in the simulation of all four FMUs in Line 5-8. The simulation results were written in their respective .CSV files in Line 11-14. The time interval for the exported files for both the reference model and all the FMU simulations was configured to $0.5[s]$.

```
1  # Import the CSV file that holds the control signal sequence.
2  input = np.genfromtxt('datafiles/ControlSigna2.csv', delimiter=',', names=True)
3
4  # Simulation of the FMUs based on the imported csv control sequence.
5  result_airHeaterOMexpEul = simulate_fmu(airHeaterOMexpEul, start_values=start_values, output=output, stop_time=600.0, input
       =input, output_interval=0.5)
6  result_airHeaterOMcvode = simulate_fmu(airHeaterOMcvode, start_values=start_values, output=output, stop_time=600.0, input=
       input, output_interval=0.5)
7  result_airHeaterDymolaCvode = simulate_fmu(airHeaterDymolaCvode, start_values=start_values, output=output, stop_time=600.0,
        input=input, output_interval=0.5)
8  result_airHeaterDymolaOEM = simulate_fmu(airHeaterDymolaOEM, start_values=start_values, output=output, stop_time=600.0,
       input=input, output_interval=0.5)
9
10 # Write simulation data to CSV files
11 write_csv('datafiles/airHeaterOMexpEulSimData.csv', result_airHeaterOMexpEul, columns=None)
12 write_csv('datafiles/airHeaterOMcvodeSimData.csv', result_airHeaterOMcvode, columns=None)
13 write_csv('datafiles/airHeaterDymolaCvodeSimData.csv', result_airHeaterDymolaCvode, columns=None)
14 write_csv('datafiles/airHeaterDymolaOEMSimData.csv', result_airHeaterDymolaOEM, columns=None)
```

Code 7.2: Simulate FMUs with control signal sequence - Link to source code.

In Code 7.3, five data frames were created using the Pandas Python library in Line 11-20. These data frames were generated for the reference model and the four FMU simulations. Afterward, the statistical measures of all the imported data were written to .CSV files in Line 26-38.

```
1  import pandas as pd
2  import matplotlib.pyplot as plt
3  import numpy
4  from pandas.plotting import table
5
6  # Generate data frames based on exported CSV files
7  # Imported data from simulation in Modelica
8  dfRef = pd.read_csv("datafiles/exportedVariablesFromModelicaREV.csv")
9
10 #FMU model generated with explicit Euler in OpenModelica into a variable
11 dfOMeE = pd.read_csv("datafiles/airHeaterOMexpEulSimData.csv")
12
13 #FMU model generated with Cvode in OpenModelica into a variable
14 dfOMcv = pd.read_csv("datafiles/airHeaterOMcvodeSimData.csv")
15
16 #FMU model generated with Cvode in Dymola into a variable
17 dfDYMcv = pd.read_csv("datafiles/airHeaterDymolaCvodeSimData.csv")
18
19 #FMU model generated with Dymola Dassl solver in Dymola into a variable
20 dfDYMdas = pd.read_csv("datafiles/airHeaterDymolaOEMSimData.csv")
21
22 # Remove the percentiles from the output
23 perc = []
24
25 # Write statistical properties of the imported data from Modelica simulation to a CSV file
26 dfRef.describe(percentiles=perc).to_csv("Tables/describeModelicaReference.csv")
```

```
27
28 # Write statistical properties of the imported FMU model generated with explicit Euler in OpenModelica to a CSV file
29 dfOMeE.describe(percentiles=perc).to_csv("Tables/describeOMexpEul.csv")
30
31 # Write statistical properties of the imported FMU model generated with Cvode in OpenModelica to a CSV file
32 dfOMcv.describe(percentiles=perc).to_csv("Tables/describeOMcvode.csv")
33
34 # Write statistical properties of the imported FMU model generated with Cvode in Dymola to a CSV file
35 dfDYMcv.describe(percentiles=perc).to_csv("Tables/describeDymolaCvode.csv")
36
37 # Write statistical properties of the imported FMU model generated with Dymola Dassl solver in Dymola to a CSV file
38 dfDYMdas.describe(percentiles=perc).to_csv("Tables/describeDymolaDassl.csv")
```

Code 7.3: Import .csv files from simulations of FMUs in Python and Dymola reference simulation - Link to source code.

The differences in value between the reference model and the individual FMUs are calculated in Code 7.4.

```
1  # Generate a new dataframe to evaluate the difference between the different FMUs temperature output and the Dymola
       reference simulation
2  dfTOut_diff = dfRef[['time']].copy()
3
4  # Calculate the difference between the Modelica export and OpenModelica_Explicit_Euler
5  dfTOut_diff['OpenModelica_Explicit_Euler'] = dfRef['T_Out'] - dfOMeE['T_Out_ext']
6
7  # Calculate the difference between the Modelica export and OpenModelica_Cvode
8  dfTOut_diff['OpenModelica_Cvode'] = dfRef['T_Out'] - dfOMcv['T_Out_ext']
9
10 # Calculate the difference between the Modelica export and Dymola_Cvode
11 dfTOut_diff['Dymola_Cvode'] = dfRef['T_Out'] - dfDYMcv['T_Out_ext']
12
13 # Calculate the difference between the Modelica export and Dymola_Dassl
14 dfTOut_diff['Dymola_Dassl'] = dfRef['T_Out'] - dfDYMdas['T_Out_ext']
```

Code 7.4: Build a new data frame for the resulting difference between the models - Link to source code.

The calculated statistical measures of the data frame are written to a new data frame in Code 7.5.

```
1  # Write statistical properties of the difference between the imported FMU models and the Modelica export to a CSV file
2  describe_df_dfTOut_diff =dfTOut_diff.describe(percentiles=perc).to_csv("Tables/describeDifference")
3
4  # Write statistical properties of the difference between the imported FMU models and the Modelica export variable for
       plotting
5  describe_df_dfTOut_diff = dfTOut_diff.describe(percentiles=perc)
6
7  # Remove the variables time, counter, median, and mean
8  describe_df_dfTOut_diff_plt = describe_df_dfTOut_diff.drop(index=['count','50%','mean'], columns='time')
```

Code 7.5: Calculate statistical properties of the difference between FMU models in Python and Modelica reference. - Link to source code.

7.2 Results

The plot in Figure 7.2 illustrates that all models accurately capture the dynamic response of the process.

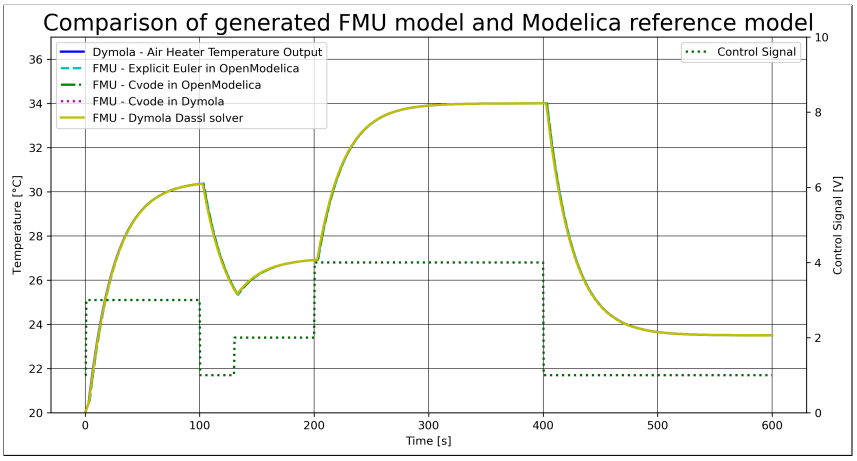


Figure 7.2: Plot of result from simulation of Python and Dymola reference.

In Figure 7.3, the plot illustrates the differences in values between individual FMUs and the reference model exported from OpenModelica. From the plot, it is evident that the FMU based on the explicit Euler method, characterized by a fixed step size and a first-order approximation exhibits a deviation from the reference model when subjected to a step change in the control signal. The FMUs employ the CVODE and DASSL solvers known for their dynamic step size adaptation and higher-order accuracy, presenting an increased ability to adjust to the step change. It is worth noting that reducing the step size for the explicit Euler method could potentially help mitigate the observed differences.

A heat matrix displaying the correlation between the reference model and the individual FMUs is given in Figure 7.4. The 'time' variable displays perfect correlation, and close to perfect correlation when comparing the output temperatures.

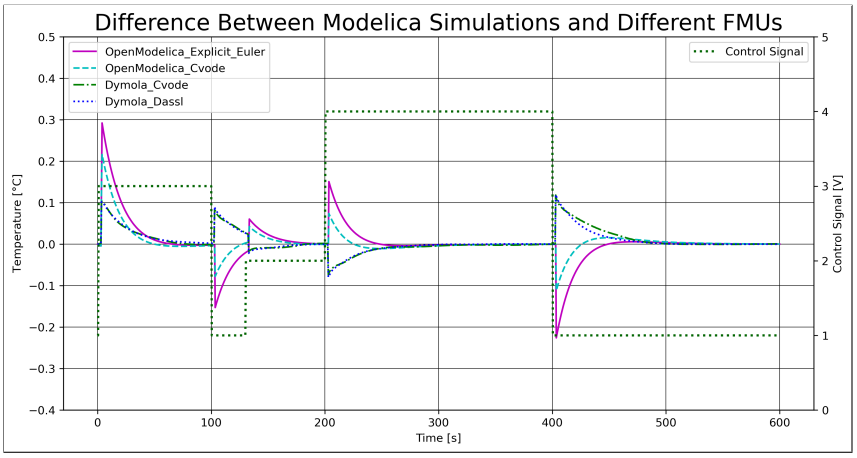


Figure 7.3: Difference in output temperature for FMU simulations in Python and Reference model from OpenModelica.

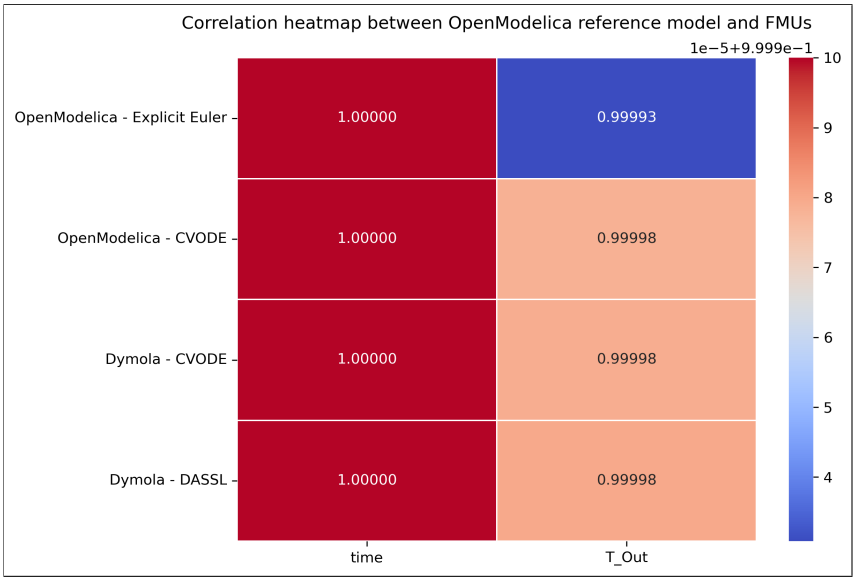


Figure 7.4: Statistical measures for the difference between Python and Modelica model.

8 FMU of Air Heater with PI-Controller in Python

This chapter introduces an implementation of an FMU alongside a PI-controller in Python. A comparison between the simulation results obtained in Python and those performed in OpenModelica is also provided.

The code in this chapter presents components of the Python implementation integrated with a PI-controller, connected to a model of the air heater exported from OpenModelica as an FMU. The import of the FMU into Python is facilitated by using the FMPy package, as presented in Chapter 3.7. The PI-controller implementation is derived from the wood-chip tank with a PI-controller, accessible at the following link: (Link to source code) on TechTeach.no. The complete source code for the implementation presented in this chapter is in the GitHub repository: (Link to repository).

8.1 Implementation in Python

The PI-controller function is implemented in Python as displayed in Code 8.1.

```
def fun_pi_con(y_sp_k, y_k, u_i_km1, contr_params, dt):

    (Kc, Ti, u_man, u_min, u_max) = contr_params
    e_k = y_sp_k - y_k                          # Control error
    u_p_k = Kc*(y_sp_k - y_k)                   # P term
    u_i_k = u_i_km1 + (Kc*dt/Ti)*e_k            # I term
    u_i_min = u_min - u_man
    u_i_max = u_max - u_man
    u_i_k = np.clip(u_i_k, u_i_min, u_i_max)    # Anti windup (Limit ui)
    u_k = u_man + u_p_k + u_i_k                 # PI term + man control
    u_k = np.clip(u_k, u_min, u_max)            # Limitation of control

    return (u_k, u_i_k)                         #Returns control signal and calculated I term
```

Code 8.1: PI-controller function in Python. - Link to source code.

The simulation parameters for the Python simulation in Code 8.2.

```
# define the 'models name and simulation parameters
fmu_filename = 'AirHeater.fmu'
start_time = 0.0
stop_time = 600.0
step_size = 1e-3
```

Code 8.2: Simulation parameters. - Link to source code.

Line 3 in Code 8.3 displays the 'FOR' loop that iterates through the complete set of variables available for the imported instance of the air heater FMU, establishing a mapping between variable names and their corresponding value references. In Lines 7, 8, and 9, the mapped dictionary variables are utilized to assign new variable names. These new names are then employed in the 'fmu.set()' and 'fmu.get()' functions during the simulation.

```
# collect the value references from the FMU
vrs = {}
for variable in model_description.modelVariables:
    vrs[variable.name] = variable.valueReference

# get the value references for the variables we want to get/set
fmu_ControlSignal_Input = vrs['u_ext']                  # Control signal
fmu_AmbientTemperature_Input = vrs['T_amb_ext']         # Ambient air temperature
fmu_OutputTemperature_Output = vrs['T_Out_ext']         # Output temperature from air heater
```

Code 8.3: Linking variables from the FMU - Link to source code.

In Lines 2 and 3 of Code 8.4, the initial values are assigned to the variables. In Lines 6 and 7, the values are written to the FMU variables that were generated in Code 8.3. This is achieved using the library function 'fmu.setReal()'.

```
# Initial simulation conditions
AmbientTemperature = 20.0
InitialControlSignal = 0.0

# Set the initial values in the FMU variable definitions
fmu.setReal([fmu_ControlSignal_Input], [InitialControlSignal])
fmu.setReal([fmu_AmbientTemperature_Input], [AmbienteTemperature])
```

Code 8.4: Initialize the simulation parameters - Link to source code.

In Line 2 of Code 8.5 the simulation of the model initiates at the specified start time, and then advances one step forward in time in Line 5. In Line 8, the defined inputs, outputs, and ambient temperature are recorded using the 'fmu.getReal()' function. In Line 11 these values are added to an array for plotting purposes. In Line 13 the control

signal is calculated based on the implemented PI-controller function. Finally, in Lines 15 and 16 the control signal and ambient temperature are updated using the 'fmu.setReal()' function.

```
1  # perform one step
2      fmu.doStep(currentCommunicationPoint=time, communicationStepSize=step_size)
3
4      # advance the time
5      time += step_size
6
7      # get the values for 'inputs' and 'outputs'
8      inputs, outputs, AmbTemp = fmu.getReal([fmu_ControlSignal_Input, fmu_OutputTemperature_Output,
         fmu_AmbientTemperature_Input])
9
10     # append the results for plotting
11     rows.append((time, inputs, outputs, AmbTemp))
12
13     (controlSignal, u_i_k) = fun_pi_con(setpoint, outputs, prevIterm, contr_params, dt)
14
15     fmu.setReal([fmu_ControlSignal_Input], [controlSignal])
16     fmu.setReal([fmu_AmbientTemperature_Input], [AmbienteTemperature])
```

Code 8.5: FMU simulation loop - Link to source code.

8.2 Results

This chapter presents the results from a comparison of the simulation performed with the FMU in Python and a implementation in OpenModelica, subjected to identical step size, duration, changes in environmental temperature, and changes in temperature setpoint.

Figure 8.1 displays the simulation results in Python. Upon comparison with the results shown in Figure 4.5, it is evident that the plots exhibit similar dynamic response.

The statistical measures of both simulations along with the absolute values of the differences are presented in Table 8.1. From the table it is observable that the resolution of the simulations is relatively high with a step size resulting in 600,000 data points for both simulations. The 7 additional data points in OpenModelica are considered to be an insignificant difference. The maximum output temperature differences with 0.98 [°C] difference between the simulations.

The simulation results in Figure 8.2 displays that the FMU-based Python simulation exhibits a greater overshot in the output temperature compared to the OpenModelica simulation. The difference is also evident in the control signal from the PI-controller, where the OpenModelica simulation reduces the control signal prior to the reduction for

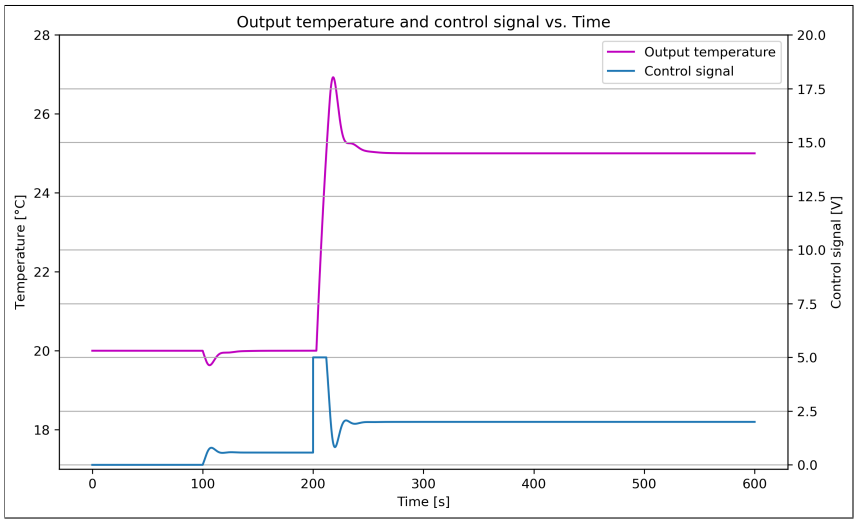


Figure 8.1: Resulting plot for simulation of PI-controller and FMU of a air heater in Python.

Table 8.1: Statistical measures for both simulations.

Measures	Control Signal [V]			Output Temperature [°C]		
	Python	OpenModelica	Diff	Python	OpenModelica	Diff
Data points	600001	600008	7	600001	600008	7
Mean	1.485	1.478	0.01	23.305	23.279	0.03
Standard dev.	0.976	0.947	0.03	2.412	2.386	0.03
Min	0.000	0.000	0.00	19.633	19.633	0.00
Max	5.000	5.000	0.00	26.926	25.942	0.98

the Python simulation. This discrepancy accounts for the 0.98 [°C] difference in Table 8.1. For smaller disturbances such as the drop in ambient temperature at $100[s]$, both simulations have an equal response.

Plot in Figure 8.3 illustrates the calculated difference between the simulations, highlighting variations in both the control signal and the resulting output temperature. The recorded difference is due to the different methods of implementation of the PI-controllers where the controller parameters were calculated based on the OpenModelica implementation.

Correlation results in Figure 8.4 indicate a strong correlation between the two simulations.

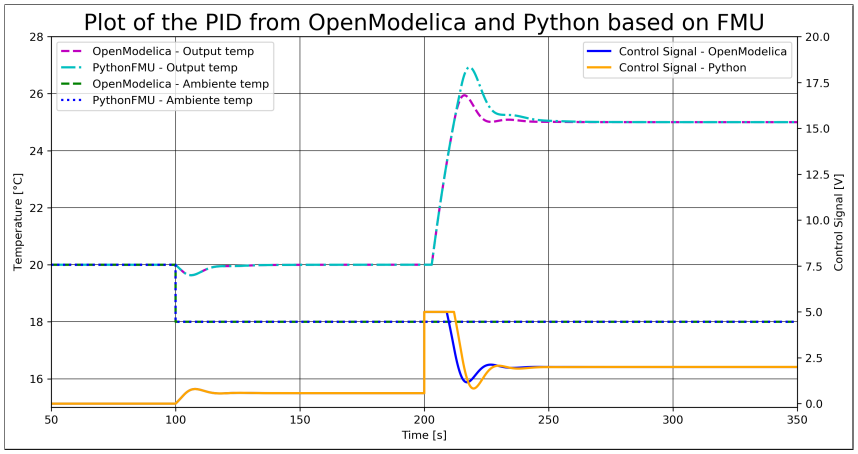


Figure 8.2: Resulting plot for simulation of PI-controller and FMU of an air heater compared with simulation from OpenModelica.

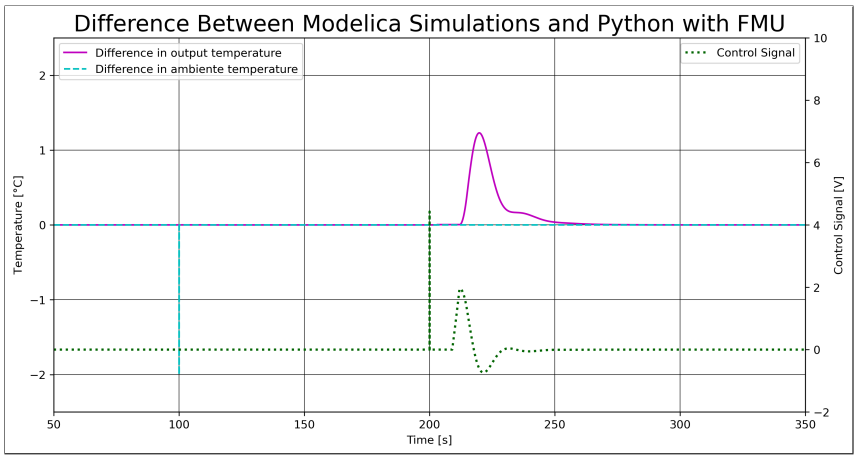


Figure 8.3: Resulting plot comparison of the difference between PI-controller in Python and OpenModelica.

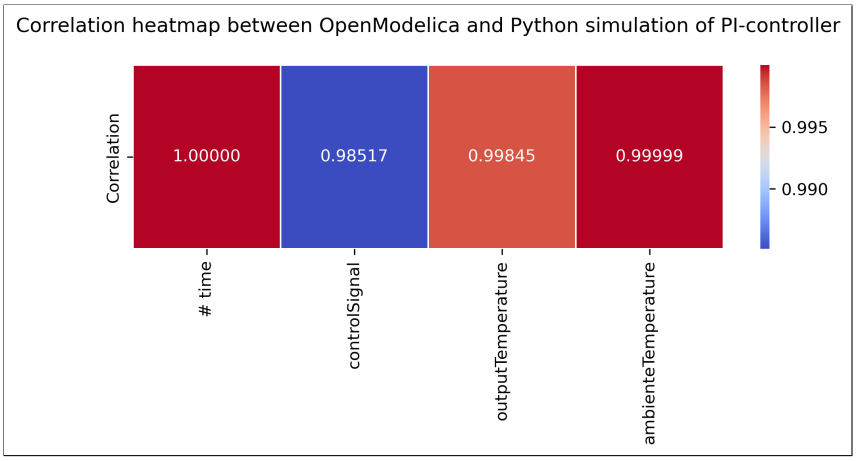


Figure 8.4: Correlation heat map from results of simulation of PI-controller in OpenMOdelica and Python based on FMU.

9 FMU of Air Heater with MPC in Python

This chapter extends the utilization of FMUs, as discussed in Chapter 8. It involves the implementation of an FMU for the air heater model into existing Python source code that already incorporates an MPC based on a first-principles model of the air heater as presented in Chapter 4.1.

The implementation in this chapter is based on the source code for a constrained nonlinear MPC that regulates an air heater model, derived from a first-principles model. The source code is accessible through the following link: (Link to source code) on TechTeach.no. The modified source code, including the implementation of an air heater model based on the FMU running in parallel with the first-principles model, can be found in the GitHub repository: (Link to repository).

9.1 Implementation in Python

In Code 9.1 the start values are set with the use of the Python variable type 'list', displayed in Line 2, and also setting the initial values in Line 5. In the previous example featuring the PI-controller from Chapter 8, these values were stored in individual variables utilizing their respective 'setReal()' functions while for this implementation they are part of a 'list'.

```
1  # Start values in the FMU to be changed
2  start_vrs = [vrs['T_amb_ext'], vrs['T_Out_ext']]
3
4  # Update values for the starting point
5  start_values = [ambient_temperature, ambient_temperature]
6
7  # initialize - Needs to run before every time the simulation is performed
8  fmu.instantiate()
9  fmu.setupExperiment(startTime=t_start)
10
```

```
11 # set the start values in the FMU
12 fmu.setReal(vr=start_vrs, value=start_values)
```

Code 9.1: Instantiate FMU - Link to source code.

Code 9.2 resides within the simulation loop. The 'fmu.doStep()' function in Line 2 utilizes the same time step as for the simulation of the first-principals model. In Line 5, inputs are recorded from the FMU, and in Line 10 the new optimized control signal horizon is calculated before the first control signal is applied to the FMU in Line 18.

```
1     # Simulate the FMU model one step forward in time based on the general sampling time for the simulation
2     fmu.doStep(currentCommunicationPoint=current_time, communicationStepSize=sampling_time)
3
4     # Get the values for 'inputs' and 'outputs' from the FMU
5     inputs, outputs = fmu.getReal([fmu_ControlSignal_Input, fmu_OutputTemperature_Output])
6
7     #---- Omitted code: for clarity ----
8
9     # Optimization
10    res = minimize(calculate_mpc_objective,
11                   u_guess,
12                   method='SLSQP',
13                   constraints=[ineq_cons],
14                   options={'ftol': 1e-9, 'disp': False},
15                   bounds=bounds_u)
16
17    # SET FMU optimized control signal
18    fmu.setReal([fmu_ControlSignal_Input], [u_opt_k])
```

Code 9.2: Simulate FMU - Link to source code.

The source code in Code 9.3 illustrates the setpoint sequence applied to both models.

```
1  #%% Defining sequence for T setpoint
2  T_sp_const = 30.0  # [C]
3  Ampl_step = 2.0  # [C]
4  Slope = -0.04  # [C/s]
5  Ampl_sine = 1.0  # [C]
6  Ampl_sine2 = 1.5  # [C]
7  t_period = 50.0  # [s]
8
9  t_const_start = t_start
10 t_const_stop = 50
11 t_step_start = t_const_stop
12 t_step_stop = 100
13 t_ramp_start = t_step_stop
14 t_ramp_stop = 150
15 t_sine_start = t_ramp_stop
16 t_sine_stop = 200
17 t_sine_start2 = t_sine_stop
18 t_sine_stop2 = 250
19 t_const2_start = t_sine_stop2
20 t_const2_stop = t_stop
```

Code 9.3: - Link to source code.

9.2 Results

This chapter presents a comparison between simulation results from a FMU and a first-principles model. Table 9.1 displays statistical measures for both the control signal and output temperature in both simulations, along with the absolute value of the difference. The results in Table 9.1 show no significant differences in the simulation results.

Table 9.1: Statistical measures for both simulations.

Measures	Control Signal [V]			Output Temperature [°C]		
	Python	FMU	Diff	Python	FMU	Diff
Data points	2921	2921	2921	2921	2921	2921
Mean	0.696	0.696	0.000	30.259	30.256	0.003
Standard dev.	0.417	0.417	0.000	0.692	0.688	0.004
Min	0.000	0.000	0.000	28.000	28.000	0.000
Max	2.175	2.175	0.000	31.005	31.004	0.001

The plot in Figure 9.1 illustrates the changes in temperature setpoint and the resulting process output temperature, from both the FMU and the first-principles model. The plot shows that the bounds were not violated, and the performance of both models exhibit similar characteristics.

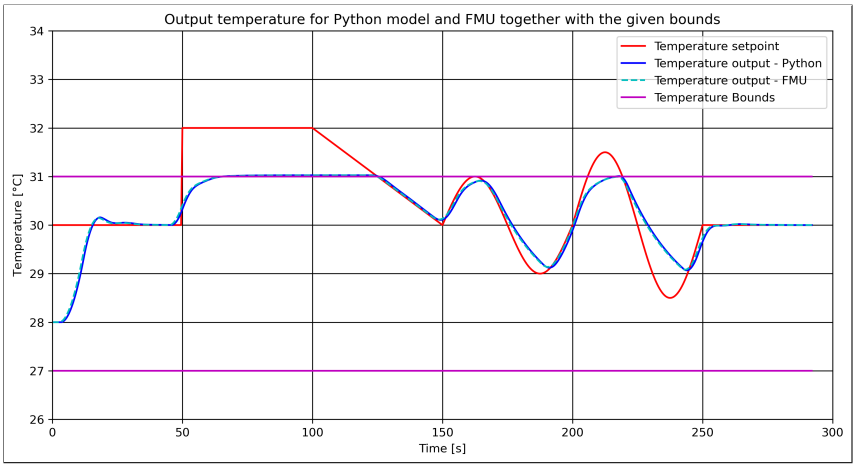


Figure 9.1: Resulting plot comparison of the output temperature difference between FMU and first-principal model.

The similarity in characteristics is further confirmed from the plot in Figure 9.2 that displays the difference in control signal and temperature output for both simulations, where the maximum difference is below 0.05 [°C].

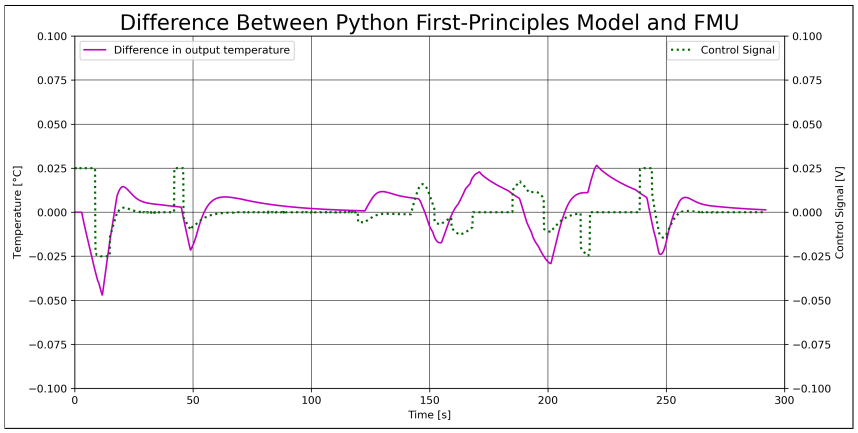


Figure 9.2: Resulting plot of difference between Python first-principles and FMU simulation.

Figure 9.3 presents the control signal for both simulations along with the control signal bounds. The plot indicates that the bounds were not violated, and the same control signal was applied to both models.

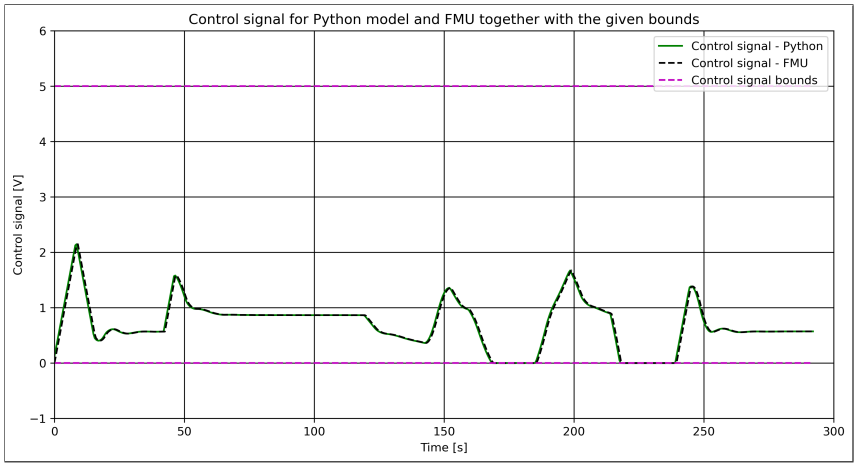


Figure 9.3: Resulting plot of the control signal with bounds.

Plot in Figure 9.4 displays the rate of change in the control signal and the corresponding bounds. The plot indicates that the bounds were not violated.

Figure 9.5 displays a perfect correlation between the temperature output for both the first-principles and the FMU model, and an identical correlation with the simulation setpoint.

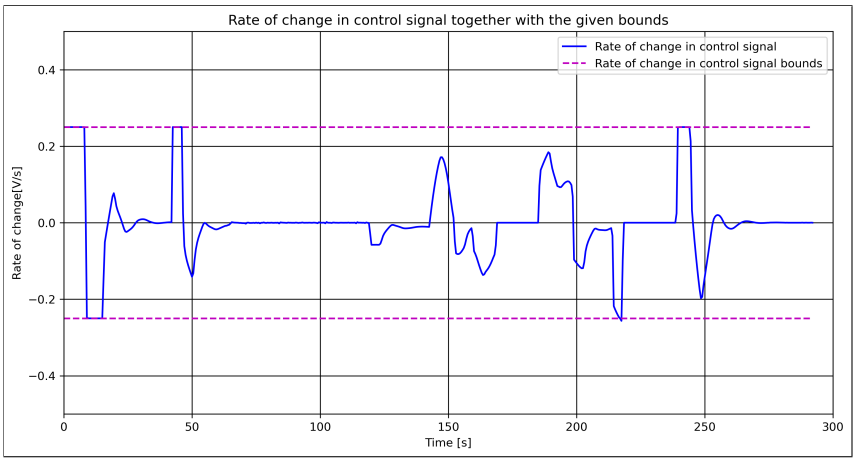


Figure 9.4: Resulting plot of rate of change in control signal.

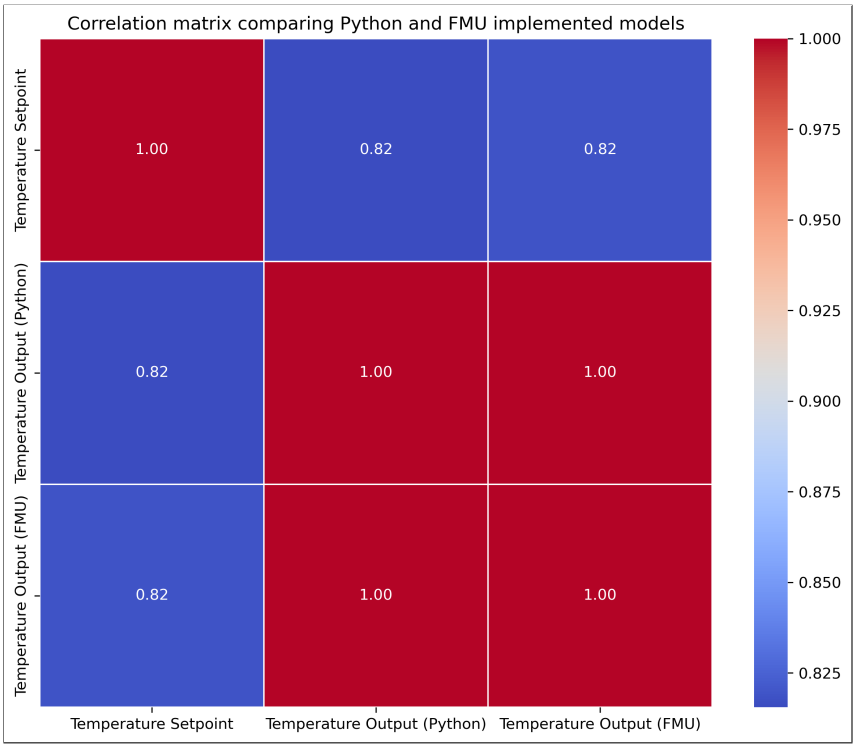


Figure 9.5: Correlation matrix for simulated models.

10 Conclusion

This thesis aimed to explore different methods for implementing advanced control within the Modelica modeling language. Two specific approaches were investigated and implemented: calling external C code using the Modelica external objects class and utilizing FMUs exported from a Modelica environment for simulation in Python.

The advantage of calling external C code lies in its integration within the Modelica framework, streamlining the simulation process and reducing computational time. While, utilizing FMUs in Python provided a versatile platform, leveraging Python's extensive libraries and widespread online support. While conducting simulations in Python based on exported FMUs demanded a higher level of programming knowledge compared to process of inputting values into a Modelica GUI, it required less programming knowledge overall compared to C development required to implement external C code with Modelica. However, it is important to note that simulations in Python are inherently more time-consuming than in C.

The choice between the two methods depends on the specific requirements and expertise of the user. This thesis serves as a user manual, offering detailed descriptions for the implementations of both methods. For the specific experiments conducted within the scope of this thesis, both approaches resulted in identical results when it comes to accuracy. Moreover, the use of GitHub as a documentation platform throughout this research offers a transparent and accessible resource for additional information, and code samples.

In conclusion, this thesis provides insights into advanced control implementation strategies within the Modelica framework and serves as a practical user manual. The comprehensive explanations provided are intended to assist in the implementation of these control strategies together with the associated GitHub repository.

Bibliography

[1] 'Yara porsgrunn.' (Mar. 2018), [Online]. Available: https://www.yara.no/om-yara/om-yara-norge/porsgrunn/.

[2] P. A. Fritzson, *Principles of object-oriented modeling and simulation with modelica 2.1*, eng, Piscataway, New Jersey, 2010.

[3] P. Carreira, *Foundations of multi-paradigm modelling for cyber-physical systems*, eng, Cham, 2020.

[4] 'The modelica association.' (Jan. 1), [Online]. Available: https://modelica.org/.

[5] 'Modelica language.' (Jan. 1), [Online]. Available: https://modelica.org/modelicalanguage.html.

[6] M. Association, *Modelica® – a unified object-oriented language for systems modeling*, English, version Version 3.6, Mar. 2023, 365 pp.

[7] 'Modelica external objects.' (), [Online]. Available: https://doc.modelica.org/om/ModelicaReference.Classes.ExternalObject.html.

[8] *Robot operating system (ros) : The complete reference. (volume 5)*, eng, Cham, 2021.

[9] 'Functional mock-up interface specification 3.0.' (May 2022), [Online]. Available: https://fmi-standard.org/docs/3.0/.

[10] Y. Xi, *Predictive control : Fundamentals and developments*, eng, Hoboken, NJ, 2019.

[11] M. Schwenzer, M. Ay, T. Bergs and D. Abel, 'Review on model predictive control: An engineering perspective,' eng, *International journal of advanced manufacturing technology*, vol. 117, no. 5-6, pp. 1327–1349, 2021, ISSN: 0268-3768.

[12] A. Bemporadl. 'Mpc from basics to learning-based design,' Youtube. (2022), [Online]. Available: https://www.youtube.com/watch?v=CNwV5GbTEGM&t=754s.

[13] *Lecture notes for the course iia 4117: Model predictive control*, Lecture notes on model predictive control, Porsgrunn, Norway: Department of Electrical Engineering, IT and Cybernetics University of South-Eastern Norway, 2019.

[14] F. A. Haugen, *Modeling, Simulation and Control*. Porsgrunn, 2023.

[15] *Process dynamics and control*, eng, Hoboken, N.J, 2011.

[16] M. AB, *Jmodelica.org user guide*, English, version Version 2.2, Modelon AB, Mar. 2018, 164 pp.

[17] S. Hölemann and D. Abel, 'Modelica predictive control – an mpc library for modelica eine mpc-bibliothek für modelica,' *at - Automatisierungstechnik*, vol. 57, no. 4, pp. 187–194, 2009. DOI: doi:10.1524/auto.2009.0766. [Online]. Available: https://doi.org/10.1524/auto.2009.0766.

[18] C. B. Corbonell, 'Model-based predictive control using modelica and open source components,' M.S. thesis, Norwegian University of Science and Technology, 2010.

[19] J. Hou, H. Li, N. Nord and G. Huang, 'Model predictive control under weather forecast uncertainty for hvac systems in university buildings,' *Energy and Buildings*, vol. 257, p. 111 793, 2022, ISSN: 0378-7788. DOI: https://doi.org/10.1016/j.enbuild.2021.111793. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S037877882101077X.

[20] F. Jorissen, W. Boydens and L. Helsen, 'Taco, an automated toolchain for model predictive control of building systems: Implementation and verification,' *Journal of Building Performance Simulation*, vol. 12, no. 2, pp. 180–192, 2019. DOI: 10.1080/19401493.2018.1498537.

[21] 'Embedded mpc for next-gen controls.' (), [Online]. Available: https://www.odys.it/.

[22] 'Fmpy.' (), [Online]. Available: https://github.com/CATIA-Systems/FMPy.

[23] S. G. Johnson, *The NLopt nonlinear-optimization package*, https://github.com/stevengj/nlopt, 2007.

[24] 'Air heater.' (), [Online]. Available: https://www.halvorsen.blog/documents/hardware/air_heater.php.

[25] *Demonstrating pid control principles using an air heater and labview*, Report, Porsgrunn, Norway: Department of Electrical Engineering, IT and Cybernetics University of South-Eastern Norway, 2007.

Appendix A

Task description for master's thesis on developing MPC blocks for Modelica

The signed task description as the basis for the thesis

University of South-Eastern Norway

Faculty of Technology, Natural Sciences and Maritime Sciences, Campus Porsgrunn

FMH606 Master's Thesis

Title: Development of MPC blocks for Modelica

USN supervisor: Finn Aakre Haugen

External partner: Yara Porsgrunn – Anushka Perera

Task background:
It is of interest of the external partner to supplement Modelica's standard library with MPC blocks; it currently contains PID blocks only. There can be many ways of developing MPC blocks, one possibility is to make Simulink MPC blocks available within Modelica using MATLAB and Simulink coders. The coders are used to translate, among others, Simulink MPC blocks into C/C++ source. The generated source code be compiled into DLL's and these DLL's can then be called with Modelica.

Task description:
1. Do a review about existing MPC implementations for Modelica.
2. Develop MPC blocks, like PID blocks, for Modelica using MATLAB/Simulink coders or any other.
3. Test the developed MPC blocks.

Student category: IIA

Is the task suitable for online students (not present at the campus)? Yes

Practical arrangements: N/A

Supervision:
As a general rule, the student is entitled to 15-20 hours of supervision. This includes necessary time for the supervisor to prepare for supervision meetings (reading material to be discussed, etc).

Signatures:

Supervisor (date and signature): 1st February 2023 Finn Haugen

CARL MAGNUS BØE
Student (write clearly in all capitalized letters):

2023.01.28 - Carl Magnus Bøe
Student (date and signature):