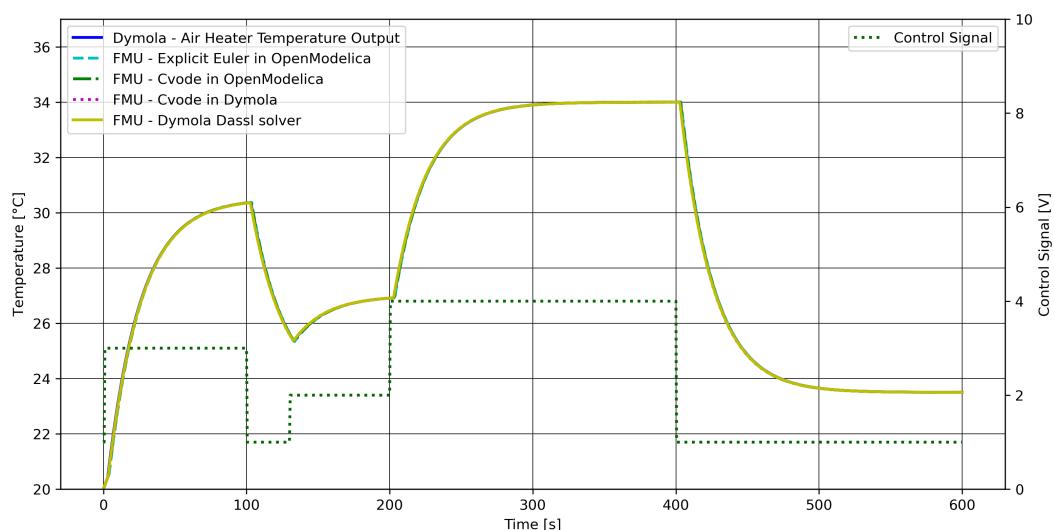


**FMH606 Master's Thesis 2023**  
**Industrial IT and Automation**

# **Advanced Control Implementations with Modelica**



Carl Magnus Bøe

Faculty of Technology, Natural Sciences and Maritime Sciences  
Campus Porsgrunn

**Course:** FMH606 Master's Thesis 2023

**Title:** *Advanced Control Implementations with Modelica*

**Pages:** 99

**Keywords:** *Modelica external objects, Optimization, MPC, model predictive control, Modelica, C, Python, Modelling, advanced control.*

**Student:** *Carl Magnus Bøe*

**Supervisor:** *Finn Aakre Haugen*

**External partner:** *Anushka Perera - Yara Porsgrunn*

**Summary:**

test-test-test]

Draft

# 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 several 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 person's 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, 18th October 2023

Carl Magnus Bøe

# Contents

Preface	3
Contents	6
<b>1 Introduction</b>	<b>8</b>
1.1 Revised Scope and Title . . . . .	8
1.2 Background Information . . . . .	8
1.3 Method and Structure . . . . .	9
<b>2 Background Theory</b>	<b>10</b>
2.1 Modelling . . . . .	10
2.2 Modelica . . . . .	12
2.2.1 Modelica Language . . . . .	13
2.2.2 External Objects . . . . .	15
2.2.3 Functional Mock-Up Interface (FMI) . . . . .	15
2.3 Model Predictive Control . . . . .	16
2.3.1 Prediction Model . . . . .	18
2.3.2 Rolling Optimization . . . . .	18
2.3.3 Feedback Correction . . . . .	19
2.3.4 Types of Models Utilized in MPC . . . . .	19
2.3.5 Optimization in MPC . . . . .	23
<b>3 Review of Existing Options</b>	<b>30</b>
3.1 Linear MPC Modelica Library . . . . .	31
3.2 Master's Thesis - Carles Buqueras Carbonell . . . . .	31
3.3 Article - Model Predictive Control Under Weather Forecast Uncertainty for HVAC Systems in University Buildings . . . . .	32

Draft

3.4	TACO - Tool Chain for Automated Control and Optimization . . . . .	32
3.5	ODYS . . . . .	33
3.5.1	ODYS Software Architecture . . . . .	34
3.6	Book - Modeling, Simulation and Control . . . . .	34
3.7	FMPy - Dassault Systèmes . . . . .	35
3.8	NLopt . . . . .	36
<b>4</b>	<b>Implemented Models</b>	<b>38</b>
4.1	Model of Air Heater Based on First-principles Model . . . . .	38
4.1.1	Model of Air Heater in Modelica . . . . .	39
4.1.2	Calculate PI Parameters for the Air Heater . . . . .	41
4.2	Model of Air Heater Based on Transfer Function Model . . . . .	43
4.3	Model of Air Heater Based on State Space Model . . . . .	44
4.4	Data Validation and Analysis . . . . .	47
4.5	Continuous Stirred-Tank Reactor (CSTR) . . . . .	47
<b>5</b>	<b>Different Methods of Calling External C Code in Modelica</b>	<b>52</b>
5.1	Initial Setup in Modelica . . . . .	53
5.2	Modelica Function . . . . .	54
5.2.1	External C Function . . . . .	55
5.2.2	Modelica Function . . . . .	55
5.2.3	Modelica Function Call Wrapper . . . . .	56
5.2.4	Modelica Simulation Model . . . . .	56
5.3	Modelica External Objects . . . . .	57
5.3.1	Modelica Record and C Struct . . . . .	58
5.3.2	External Object Class - Constructor and Destructor in Modelica and C .	59
5.3.3	Modelica Function Call . . . . .	61
5.4	Data Validation and Analysis . . . . .	62
<b>6</b>	<b>Calling NLopt as External Object in OpenModelica</b>	<b>64</b>
6.1	Installation of the NLopt Library on Windows . . . . .	65
6.2	Implementation of the NLopt Version Block . . . . .	66
6.3	Implementation of the NLopt Univariate Optimization . . . . .	67
6.3.1	Objective Function . . . . .	67
6.3.2	Optimization Parameters . . . . .	68

Draft

6.3.3	Implemented NLOpt Optimization in C . . . . .	71
6.3.4	Simulation Results in OpenModelica . . . . .	73
6.4	Implementation of the NLOpt Multivariate Optimization . . . . .	73
6.4.1	Objective Function . . . . .	73
6.4.2	Optimization Parameters . . . . .	74
6.4.3	Implemented NLOpt Optimization in C . . . . .	75
6.4.4	Simulation Results in OpenModelica . . . . .	75
<b>7</b>	<b>FMU of Air Heater in Python</b>	<b>77</b>
7.1	Implementation in Python . . . . .	78
7.2	Results . . . . .	80
<b>8</b>	<b>FMU of Air Heater with PI Controller in Python</b>	<b>84</b>
8.1	Implementation in Python . . . . .	84
8.2	Results . . . . .	86
<b>9</b>	<b>FMU of Air Heater with MPC in Python</b>	<b>89</b>
9.1	Implementation in Python . . . . .	89
9.2	Results . . . . .	91
<b>10</b>	<b>Conclusion</b>	<b>94</b>
	<b>Bibliography</b>	<b>95</b>
<b>A</b>	<b>Task Description</b>	<b>98</b>

Draft

# Nomenclature

Symbol	Explanation
MPC	Model predictive control
EKF	Extended Kalman filter
USN	University of Southeast Norway
FMI	Functional Mock-up Interface
SSP	System Structure and Parametrization
DCP	Distributed Co-Simulation Protocol
eFMI	Functional Mock-up Interface for embedded Systems
FMU	Functional Mock-up Units
EMPHYSIS	Embedded systems with physical models in the production code software
DEA	Differential algebraic equation
PID	Proportional Integral Differential
SISO	Single input single output
MIMO	Multiple input multiple output
MES	Manufacturing Execution System
NMPC	Nonlinear Model Predictive Control
ARX	Auto Regressive Exogenous
DYMOLA	Dynamic Modelling Laboratory
ODYS	Optimization of Dynamical Systems
NLMPC	Non-linear Model Predictive Control
GUI	Graphical User Interface

# 1 Introduction

---

This chapter provides details about the master's thesis, including background information and 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 presented 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, consists of generating models within Modelica. Subsequently, these models are exported as functional mock-up units (FMUs) to SIMULINK. Within SIMULINK the MPC block in the SIMULINK model predictive control toolbox



Figure 1.1: Current workflow

is utilized to compute the control signal. Yara endeavors to enhance efficiency by centralizing the entire 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 controllers. Subsequently, it assesses various models employed in the implementations presented in this thesis.

Following this, the next chapter illustrates various implementations of advanced control methods in Modelica. Non-essential segments of the source code, such as those related to plotting, are omitted in the main text to emphasize the control implementation. The entire source code for all instances presented in this thesis can be found in the GitHub repository ([Link to repository](#)).

Each segment of source code presented in this thesis is accompanied by a direct link in the figure 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](#)

Draft

## 2 Background Theory

---

In this chapter the software applications utilized in the project are presented, alongside the outcomes of a literature survey exploring existing alternatives for implementing advanced control implementations within the Modelica framework.

---

### 2.1 Modelling

Creating a model of a system facilitates the execution of tests to understand its properties and behavior. 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].

All natural and human-made systems can be considered to be dynamic in a sense that they represent a change over time, since the systems has some inertia working against change [3]. This dynamic is represented by including time as a variable in the model. By eliminating the time as a variable one has constructed a static system that can be utilized to present the steady state where the output of the system is stable if the input signals are unchanged.

One example highlighting the differences between static and dynamic behavior is displayed in Figure 2.1. In this scenario one can observe both a resistor and a capacitor being 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]. Thus, a system is said to be static if the output only depends on the present input, while if the system output also depends on past inputs, the system can be considered to be dynamic [4].

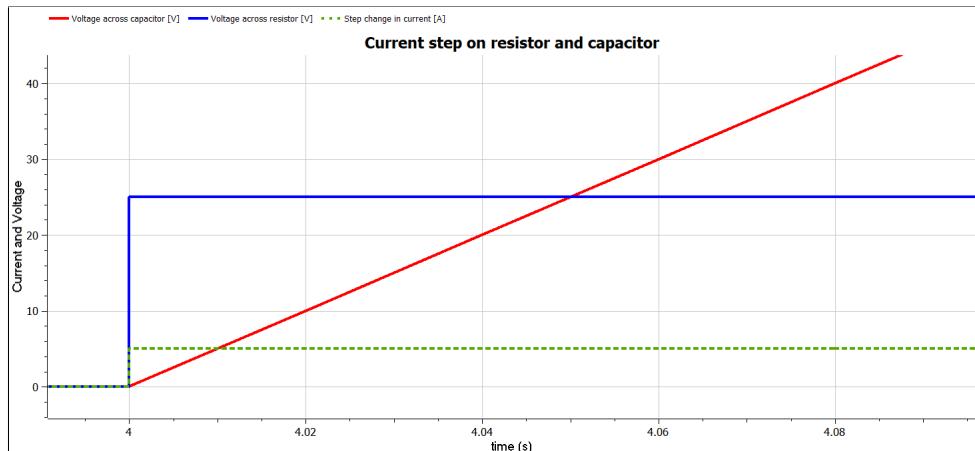


Figure 2.1: Plot of current step response in resistor and capacitor.

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. One practical example of this can be a physical system where the movement of any physical components can be considered to have continuous-system changes, but the set point for controlling the position might be in

discrete-time [2]. By considering a physical system as a discrete-time system one will reduce the model complexity and possibly speed up a simulation. This can in various cases be done if the physical system has changes so fast it can be considered to be close to instantaneous, or if the system can experience discontinuity at a specific point in time [2].

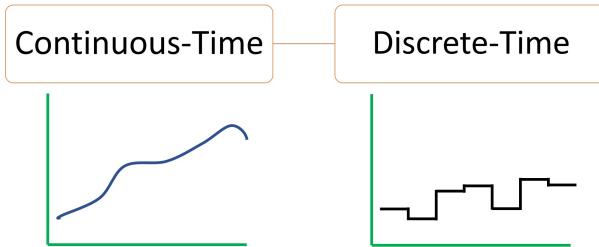


Figure 2.2: Difference between continuous and discrete time models

## 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 [5].

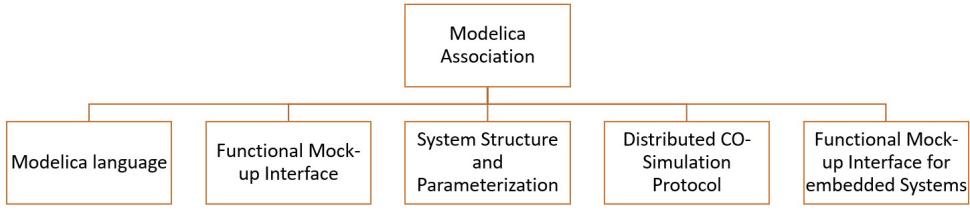


Figure 2.3: Standards governed by the Modelica Association [5].

### 2.2.1 Modelica Language

The Modelica modeling language is an object-oriented language designed for modeling complex multi-domain systems [6]. Modelica is a declarative modeling language in which dynamic models are expressed through equations, contrasting with procedural languages where one typically formulates step-wise algorithms to achieve a desired goal. Analogous to 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 blend 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[6].

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. OpenModelica is the prominent open-source alternative, in addition to several commercial alternatives as displayed in Table 2.1.

Table 2.1: Commercially available software based on Modelica[6]

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.

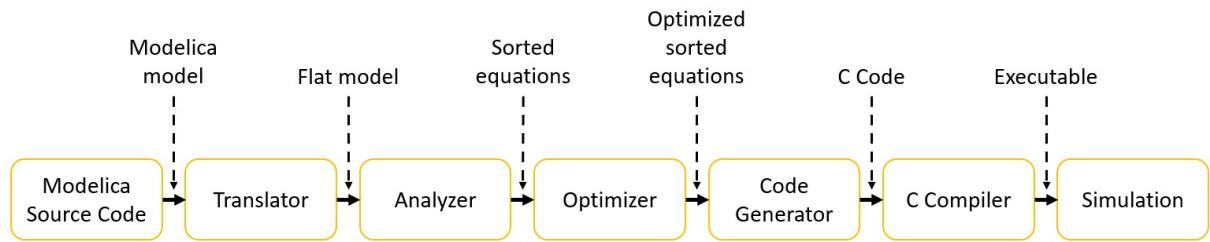


Figure 2.4: Flow displaying how Modelica models is translated and executed [2].

## Difference Between Using Equations and Algorithms

One of the most common methods for performing calculations in Modelica involves using equations. The equations represent the mathematical relationships that describe the model and are often based on physical laws and principles relevant to describing the model. When using the equation-based method there is no fixed sequence for the order of how the equations are solved. When using equations, the number of equations must match the number of variables in the model for successful simulation. The equations are typically solved by the simulation environment when the simulation is performed.

When using algorithms, one maintains the order for how the code is called by following the given sequence in the code. Algorithms are mainly utilized for tasks other than solving equations, such as calling external functions, implementing control logic, or performing specialized calculations. The algorithms need to be explicitly called to perform a specific action.

## 2.2.2 External Objects

The use of the external objects class in Modelica provides a method to store variables and parameters in external memory, enabling data access for Modelica, C, and Fortran 77 environments. This becomes particularly valuable when multiple function calls are involved, and there is a need to store the values between these calls [7].

The class "ExternalObjects," is defined to include 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 [8].

## 2.2.3 Functional Mock-Up Interface (FMI)

The functional mock-up interface is utilized in the industry as a domain independent standard to exchange dynamic models between different simulation and modelling tools. The FMI generates a vendor neutral functional mock-up unit (FMU), that is supported by an excess of 100 different modelling tools. The generated FMU file type is a compressed .ZIP file that includes an .XML file that holds the description of the parameters, derivatives, variables, and model structure. The FMU also contains equations and a shared library in C code [9].

FMI defines three distinct types of interfaces.

### Model Exchange

The model exchange is an interface designed for solving ordinary differential equations based only on using the model algorithm and does not require the solution method. The algorithms take care of the overall advance in simulation time, exchange of input and output from the individual FMUs, computing continuous state variables, handling events, and triggering of clocks. The evaluation of the FMU is only performed at specific given instances of time [10].

### Co-Simulation

With the use of the co-simulation interface, one can use a standardized interface for executing simulations of different FMU models. Both connecting tools that perform simulations and connecting exported sub-systems. The algorithm controls both synchronization and controls exchange of data between FMUs. Data exchange between the co-simulations is performed at discrete points from one communication point in time to the next, while the different sub-systems running inside the individual FMUs are solved at their respectively points in time. Based on this there may be a time-delay in the communication between the FMUs that are greater than the time of the solver in the individual systems. In addition to implementing the model algorithm as for the model exchange approach, the co-simulation also requires the solution method [10].

### Scheduled Execution

With the scheduled execution interface, specific parts of the models are exposed, allowing the importers scheduler to gain control over the execution of these individual parts. This interface provides the capability for concurrent computation of the exposed model segments. It's important to note that concurrent computation is currently not supported by the FMI standard, but it can be utilized for computations within the respective FMUs [10].

## 2.3 Model Predictive Control

The PID controller is a versatile choice for controlling both linear and non-linear processes making it universally applicable for most SISO (Single-input single-output) control requirements. When the controller needs to handle MIMO (Multiple-input multiple-output) systems and expand control from regulation to optimization with consideration for system constraints, the limitations of single-loop PID controllers become apparent, primarily due to their lack in ability to perform prediction for future process dynamics [11].

Model Predictive Control (MPC) offers a more advanced approach by utilizing a model of the system and thus enables predictions of the system's future response when subjected to a sequence of inputs across given future time-steps. The optimization algorithm within the MPC ensures the selection of a preferred input sequence based on a performance index. This transition from PID to MPC is particularly beneficial in scenarios where predictive control and optimization are crucial for achieving desired outcomes.

One example of a process flow with the use of MPC is illustrated in the block diagram presented in Figure 2.5. The model block within the MPC box displayed in Figure 2.5 represents the model of the process to be controlled, together with an optimizer. The use of the model enables the option to predict the future response in the system. In the optimization box one utilizes the system model to calculate the optimal control signal strategy for the output from the MPC controller, given as  $u$ . The output parameter from the system being controlled given by  $y$  is fed back to the MPC controller to be utilized in the optimization process in the next time-step [12].

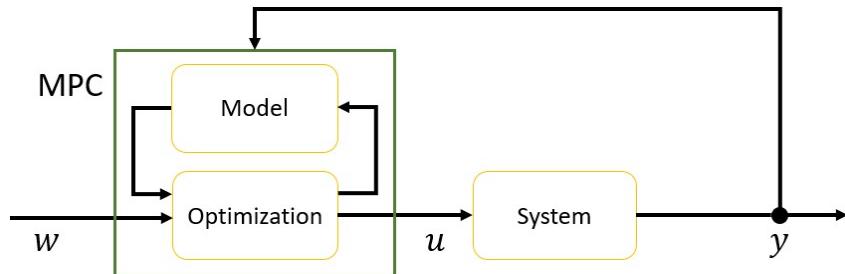


Figure 2.5: Block diagram for MPC. [12].

In addition to calculating the optimal control signal the optimizer also considers any given system constraints. Examples of constraints defined for the optimization process could be 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 [13] 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 taking into account the constraints of how the pieces are allowed to move. The better you are, the longer your prediction horizon will be. 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 [13].

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 [11]:

### **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 non-linear e.g., based on a transfer function, empirical data, state space equations, or impulse functions [11].

### **2.3.2 Rolling Optimization**

MPC utilizes on-line finite horizon optimization also known as rolling optimization oppose to traditional optimal control that is based on off-line infinite horizon calculation [11]. A general basic comparison between optimal control, predictive control and PID is presented in Table 2.2.

As MPC relies on optimization it aims to determine future control actions by optimizing and generating different performance indexes to compare different alternatives. The performance index is determined by the prediction model together with the future control signals over a finite horizon where only the first control signal is applied to the process. At the next time step the complete optimization problem is solved once more to generate a new series of optimal control signals. The performance index calculations can be utilized to minimize the error between the model and the future estimated outputs, while also possibly having constraints on the output.

Draft

Table 2.2: Comparison between optimal control, PID and MPC [11]

	<i>Required Information</i>	<i>Control Style</i>	<i>General Performance</i>
<i>Optimal Control</i>	Accurate information about model and environment	Online implementation and offline optimization	Ideal if there exists no uncertainty
<i>MPC</i>	Real-time outputs and prior information about environment and model	Online finite horizon optimization with rolling implementation	Suitable for uncertain environments
<i>PID Control</i>	Real time information about output	Online direct control	No optimization, suitable for environments with uncertainty

### 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 real-time values from the physical system are recorded. It can then utilize these values to update future behavior, before solving the optimization problem. With this approach there is implemented a feedback correction making sure both the prediction and resulting optimization will be closer to the actual status of the physical system helping increase the prediction of future outputs [11].

### 2.3.4 Types of Models Utilized in MPC

Various types of models are utilized together with MPC. Among the most commonly employed are linear and nonlinear models, as outlined below.

## Nonlinear and Linear Discrete Time State Space Models

For practical applications, the MPC is often implemented onto a micro controller or a computer that are both digital devices operating at discrete time. Systems and software running discrete time are characterized by quantifying steps as displayed in Figure 2.2 [14].

**Nonlinear discrete time state space model** can be expressed as presented in Equation 2.1.

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

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 properties can be presented as vectors [14].

**Linear discrete time state space model** can be expressed as presented in 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 noise given by ( $v$ ) and measurement noise in ( $w$ ). The two noise terms are uncorrelated with zero mean and some given variance.

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

The matrices given by ( $A_d$ ), ( $B_d$ ), ( $C_d$ ) and ( $D_d$ ) represents 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 presented in Equation 2.3

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

### 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** can be expressed as presented in Equation 2.4.

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

Where functions ( $f$ ) and ( $g$ ) is 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** can be expressed as presented in Equation 2.5.

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

The matrices given by ( $A_c$ ), ( $B_c$ ), ( $C_c$ ) and ( $D_c$ ) represents the system matrices with the subscript ( $c$ ) indicating that the matrices are for continuous time [14].

## Transfer Function Model

Models can also be expressed as transfer functions with an example converted to the frequency domain as presented in Equation 2.6.

$$H(s) = \frac{y(s)}{u(s)} \quad (2.6)$$

Where the inputs  $u(s)$  can be transferred to outputs  $y(s)$  with the help of the transfer function given as  $H(s)$  [14].

## Weighting of Inputs and Outputs

Weighting of the inputs and outputs plays a significant role in optimizing the MPC behavior. Based on the weighing of the different inputs and outputs one can adjust the balance for conflicting goals within the optimization process.

Weighting of the inputs (also known as manipulated variables) is performed to prioritize certain inputs above others. This can be useful for applications where one or the inputs has a higher impact on the systems dynamics or has 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 system requires the controller to prioritize maintaining stability and good setpoint tracking, while the system 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. One example based on process control reference tracking is presented in Figure 2.6.

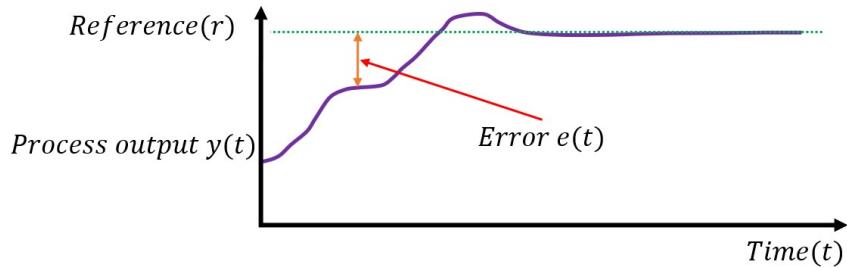


Figure 2.6: Optimization of reference tracking. [14].

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 [14].

Calculation of the minimization problem chosen in this case to be the error for one discrete instance of time can then be formulated as displayed in Equation 2.7.

$$e = (y - r)^2 \quad (2.7)$$

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, one can calculate the optimal control sequence for when the sum of squared errors is closest to 0, and by this find the optimal control sequence for the reference tracking problem displayed in Equation 2.8 [14].

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

Optimization problems are often formulated as minimization problems. An example of

how to form the optimization objective function is displayed in Equation 2.9.

$$\min_U J(U) \quad (2.9)$$

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. For this given example, the goal is to generate a matrix  $U$  that holds the optimized values for the function  $J(U)$  [15].

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  $N$  [15].

$$U = [u_k, u_{k+1}, \dots, u_{N-1}, u_N] \quad (2.10)$$

From the matrix presented in Equation 2.11 the optimized control matrix is expanded to display a control matrix for a multiple input system where several control signals are optimized, with the total number of control signals given as  $r$  [15].

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

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 [14] :

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

Where the variables for Equation 2.12 are described in Table 2.3.

Table 2.3: Variable description for objective function presented in Equation 2.12 [14].

$e_k$	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  $\Delta u_k$  as displayed Equation 2.13.

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

This can be a relevant formulation for when the system has actuators and control valves that have limitations due to their dynamic performance [14].

## Constraints

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

When applying constraints to an optimization problem one 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, one should break them in the gentlest way [14].

As an example of the different constraints one can evaluate a general function presented in Equation 2.14.

$$\min_u f(u) \quad (2.14)$$

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

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

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

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

Constraints can also be specified as bounds, representing a range of permissible values. This can be expressed as displayed in Equation 2.17 [14].

$$u_L \leq u \leq u_U \quad (2.17)$$

Based on the previously presented illustration in Figure 2.6 one can extend this system by applying any general type of constraints to the system. This will limit the feasibility region and as presented in Figure 2.7 result in a different optimized controller strategy.

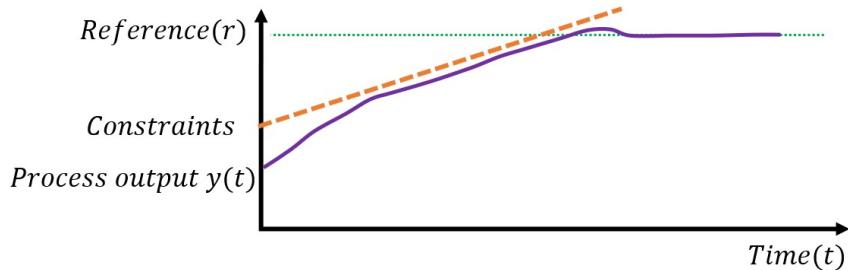


Figure 2.7: Optimization of reference tracking with constraints. [14].

From the linear constraints applied as the orange dotted line in Figure 2.7 one can observe 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 system compared to the reference tracking presented in Figure 2.6.

Draft

### Categories of Optimization Problems

Example of 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

If the optimization problem does not include any constraints, it can be considered to be an unconstrained optimization problem. This can be applicable for all three main types of optimization problems [14].

### 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. An example of a reference tracking process for a SISO system is displayed in Figure 2.8.

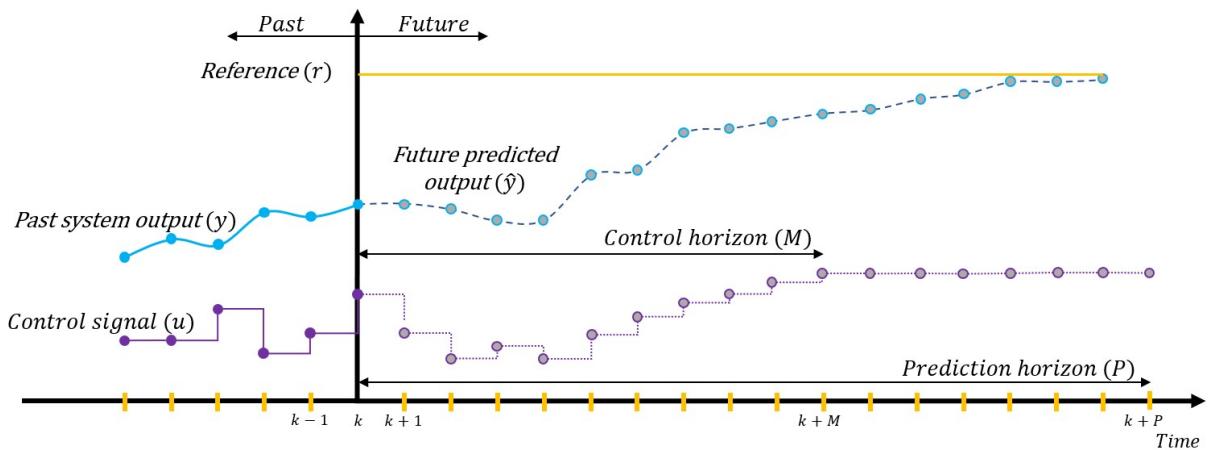


Figure 2.8: Mode of action for MPC with receding horizon [16].

From the Figure 2.8 the current place 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 system set point [16].

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 updated values from the system, one has achieved a feed-back component to the controller that can adjust for disturbances, signal noise, changes in setpoint, and model inaccuracies [16]. The sliding horizon can also introduce a feed-forward function by including the future disturbances and changes in setpoint when solving the optimization problem [14].

Draft

## 3 Review of Existing Options

---

In this chapter, there is a presentation of existing implementations of MPC 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 and presented in this chapter are based on JModelica. JModelica is an open-source platform developed by Modelon for the simulation, optimization, and analysis of complex systems. It is the result of research conducted at Lund University in Sweden, with the goal of creating a platform for exchanging technology between academia and industry. [17].

According to the website [jmodelica.org](http://jmodelica.org), 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:

- Assimulo
  - A Python-based simulation package is available for simulating differential algebraic equations (DAE) and ordinary differential equations (ODE) using explicit Euler and Runge-Kutta of order 4 methods. More information can be found at: [www.jmodelica.org/assimulo](http://www.jmodelica.org/assimulo)
- PyFMI
  - Python package to load and interact with both model exchange and co-simulation FMUs. More information can be found on: [www.jmodelica.org/pyfmi/](http://www.jmodelica.org/pyfmi/)
- FMI Library

- Software package written in C code to enable integration of FMUs into custom applications.

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. With access to the Modelon Impact one gets access to Modelon commercial version of JModelica called Optimica. The Optimica platform is available inside Modelon Impact with the use of an available Jupyter Lab client.

An implementation of MPC with the use of Optimica will be based on Python and have license cost. Given the availability of other free Python alternatives, this thesis will not

### **3.1 Linear MPC Modelica Library**

The only open-source general MPC library listed on the Modelica homepage ([www.modelica.org/libraries.html](http://www.modelica.org/libraries.html)) is the discontinued linear MPC library maintained by S. Hoeleemann and D. Ablet at RWTH Aachen University in 2012. According to the repository on GitHub ([www.github.com/modelica-3rdparty/LinearMPC](https://www.github.com/modelica-3rdparty/LinearMPC)) the library was last updated in 2016. Since then, Modelica has moved to version 4.0 making the library not compatible since it is based on Modelica version 3.2.

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 a linear process model formulated as a discrete time state space model. The controller is also able to manage constraints and disturbances. With the library one avoids all couplings to other software products and the need to generate an FMU of the model [18].

### **3.2 Master's Thesis - Carles Buqueras Carbonell**

In the work of the master's thesis written in 2010 with the title: Model-based predictive control using Modelica, and open-source components Carles suggests to first generate

the model in Modelica then compile this into C code with the use of JModelica Python package. Then use the open-source Ipopt optimizer to solve the non-linear control problem together with a JModelica libraries [19]. 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 improving 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 built in Modelica based on electrical components in a configuration of a RC-network where the R represents the heat resistance and C represent the heat capacitance [20].

The building model was imported into a Python environment where the model was compiled into a FMU with the use of the JModelica.org platform toolbox as discussed in Section 3. After the FMU is constructed in Python the model is utilized in the implemented MPC code. The source code and the .csc files are available at GitHub (<https://github.com/haoranli1988/Modelica-district-heating-optimization-MPC>).

The implementation of the FMU in addition to the Python coded MPC can be a basis to build on to construct a Python based MPC on more general form.

### **3.4 TACO - Tool Chain for Automated Control and Optimization**

TACO is a tool chain based on JModelica made to reduce the engineering level and time investment required to implement MPC in building systems. Since the tool has a focus on optimizing MPC for building systems it focuses on exploiting 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 on time only and not on state variables or

optimization variables in addition to the model's boundary conditions. The equations are then compiled into an FMU using standard functions from the JModelica platform. These equations do not need to be part of the evaluated part of the optimization problem, and thus reducing computational burden. The other part consists of the remaining equations [21]. 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 [22]:

- 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 bounds for optimization variables. The package is also able to compute the exact worst-case execution time for the solver in advance of the computation [22].

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 state-of-the-art Kalman filtering. The controller is based on a standalone C-code package suitable to run on everything from desktop to embedded computers.

Sometimes a physics based non-linear model might be too complex to develop and maintain. As an alternative one can use a data driven approach by generating a black-box model based on experimental data with the deep learning library available for Python or MATLAB. Based on the trained neural model one can produce C code to be implemented in an NLMPc controller.

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 [22].

### 3.5.1 ODYS Software Architecture

The MPC architecture consists of five distinct groups of functions.

1. Offline functions Functions that are called offline to generate the code for the real-time application.
2. Real-time functions Functions to be utilized for the Real-time implementation
3. Utility functions Functions that implement algorithms for the numerical differentiation and integration of the defined dynamic model
4. User's functions Functions that must be coded by the user. Defining the prediction model and parameters for MPC, observer and QP solver.
5. Post processing functions Analyze the models and the data generated from the MPC and EKF

## 3.6 Book - Modeling, Simulation and Control

In the book written by Finn Haugen called Modeling, Simulation, and Control [15] there is a reference to a source code in Python for the implementation of nonlinear MPC to control a simulated air heater. The example code is implemented in Python 3.7. The implementation also includes Kalman filtering, disturbances, and control blocking.

The source code is implemented from scratch and is only based on import of basic Python libraries such as NumPy, SciPy, and time. One 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 enabled the import and execution of FMUs in Python for Windows, Linux and macOS. FMPy is a free Python library that provides support for both Co-Simulation and Model Exchange for FMI 1.0, 2.0, and 3.0. FMPy can be found on GitHub in the CATIA Systems repository ([Link to repository](#)). CATIA Systems is a subsidiary of Dassault Systèmes, which is also the provider of Dymola [23].

The FMPy packaged includes six key 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

The documentation of the library is manly based on three example scrips presented below [23]:

1. Coupled clutches - [Link to source code](#)
2. Custom input - [Link to source code](#)
3. Parameter variation - [Link to source code](#)

## 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, GNU Guile, Julia, GNU R, Lua, OCaml, Rust and Crystal [24]. The algorithms are a collection of several open-source packages by different authors that have been modified to a greater or lesser extent to fit into the NLOpt framework. The source code is available at 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

Draft

Based on the distinct types of optimization problems it is advised to try different algorithms and to compare accuracy and computational time. Another element to consider when performing global optimization with NLOpt is that when a global optimization problem is to be solved it can be advantageous to run a consecutive optimization with the use of one of the local optimization algorithms for improved local accuracy.

The framework is built to enable the possibility to build several optimizer objects that can include different algorithms. With this one can run several optimizer objects on the same optimization object in the same source code. The general flow of the optimization process with NLOpt is displayed in Figure 3.1.

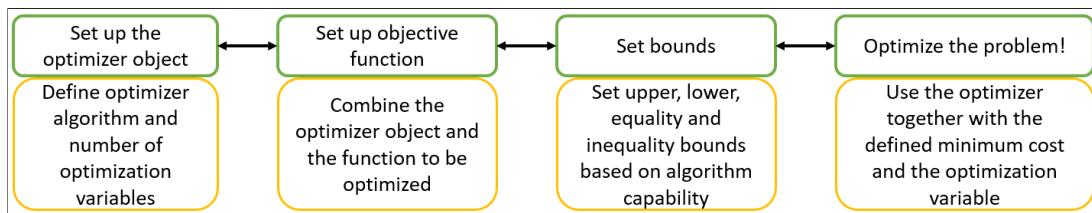


Figure 3.1: NLOpt program flow.

The NLOpt library can be implemented in C source code and with that also be implemented as an optimizer in a script called from Modelica.

Draft

## 4 Implemented Models

---

In this chapter there is a presentation of the different models implemented in Modelica accompanied by a evaluation of the models.

---

### 4.1 Model of Air Heater Based on First-principles Model

USN has constructed several identical air heater systems that are available in the campus lab station. A key application relevant to this study and the primary motivation for selecting the known air heater model is Finn Haugen's Python implementation of MPC for the air heater. The source code for the implementation of MPC in Python is available at the web site [techteach.no](http://techteach.no). With the use of the Python source code where the MPC for the air heater is implemented from scratch one can validate the model produced in Modelica against what Finn has already implemented.

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

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

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

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

And the parameters and variables given in Table 4.2.



Figure 4.1: Picture of air heater [25].

Table 4.1: Air heater system constants.

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

Table 4.2: Air heater system parameters and variables

$T[^{\circ}C]$	Temperature at the tube outlet
$T_{amb}[^{\circ}C]$	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 TechTeach web site: [http://techteach.no/air\\_heater](http://techteach.no/air_heater) and in the report "Demonstrating PID Control Principles using an Air Heater and LabVIEW" from (Haugen, Fjelddalen, Dunia & Edgar, 2007)[26].

#### 4.1.1 Model of Air Heater in Modelica

The air heater was implemented into Modelica as a model. Since the model heater gain is given with  $\frac{^{\circ}C}{V}$  and the use of Celsius as a non-SI unit, this was extended by labeling the value as non-SI unit for the simulation results in addition.

To observe the air heaters dynamics, it was also added additional if statements to change the control signal over time. The resulting Code is presented in Code 4.1.

```

1 model AirHeaterWithStep
2
3 // Import
4 import Modelica.Units.SI;
5 import Modelica.Units.NonSI;
6
7 // Constants
8 constant SI.Time T_CONST = 23.0           "Time constant";
9 constant SI.Time T_DELAY = 3.0             "Time delay";
10 constant Real Kh(unit="°C/V") = 3.5        "Heater gain";
11
12 // Parameters
13 parameter NonSI.Temperature_degC T_amb = 20 "Ambient/Room temperature";
14
15 // Variables
16 NonSI.Temperature_degC T_Out(min=T_amb)      "Temperature output from heater";
17 SI.Voltage u(min=0, max=5)                    "Control Signal";
18
19 initial equation
20   T_Out = T_amb;
21
22 equation
23 if time>0 and time<100 then
24   u = 3;
25 elseif time>=100 and time<130 then
26   u = 1;
27 elseif time>=130 and time<200 then
28   u = 2;
29 elseif time>=200 and time<400 then
30   u = 4;
31 else
32   u = 1;
33 end if;
34
35 T_CONST * der(T_Out) = (T_amb - T_Out) + Kh * delay(u, 3);
36
37 annotation(experiment(StartTime=0, StopTime=600)
38
39 end AirHeaterWithStep;

```

Code 4.1: Air Heater Implemented in Modelica - [Link to source code](#)

In Figure 4.2 one can observe the result of the simulation of the air heater in Modelica. 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 the plot in Figure 4.2 one can observe both the system time constant, and the delay between the control signal  $u$  and the output temperature  $T_{out}$ .

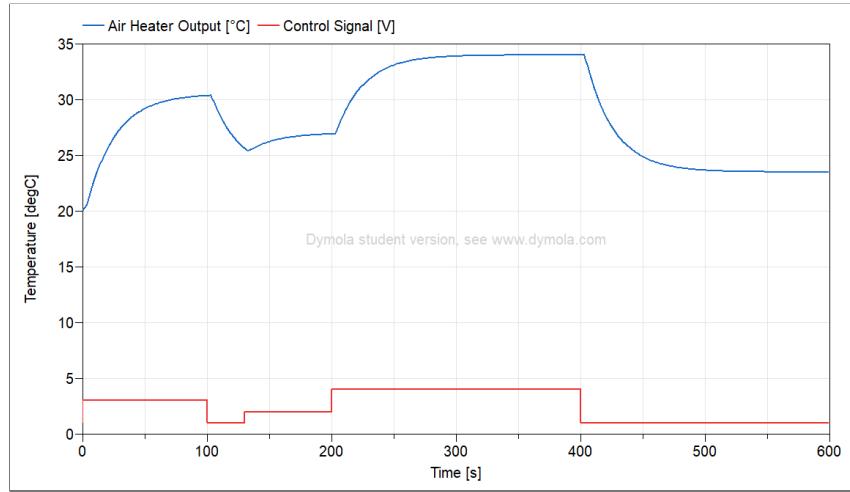


Figure 4.2: Plot of air heater with step changes in control signal

Draft

#### 4.1.2 Calculate PI Parameters for the Air Heater

From a step response introduced to the system as displayed in Figure 4.3 the parameters for a PI controller were calculated based on the Skogestad-method where the system is approximated by assuming integrator with time-delay dynamics given by the purple line in Figure 4.3 [15]. 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.

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

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

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

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  $\tau$  displayed in Equation 4.4.

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

The integral time was calculated as displayed in equation 4.5.

$$T_i = 4 \cdot \tau = 12 \quad (4.5)$$

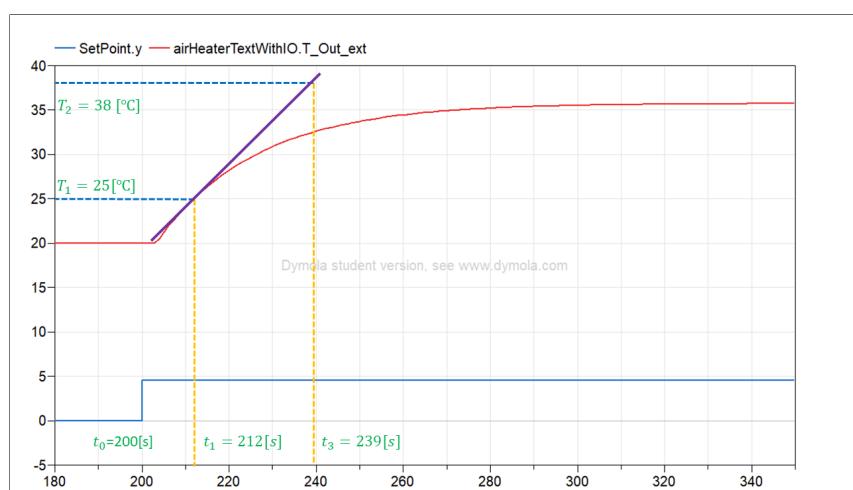


Figure 4.3: Skogestad method for tuning of PI-controller

The air heater model was set up in an example file where it was connected to the Modelica built-in PID block and step blocks for the ambient temperature and setpoint as displayed in Figure 4.4.

In Figure 4.5 one can observe the air heater response to both a change in ambient temperature and a step change in control signal for the air heater connected to a PID controller with the calculated parameters. Based on the results one can observe slight oscillations of the air heater output when performing the step change in setpoint.

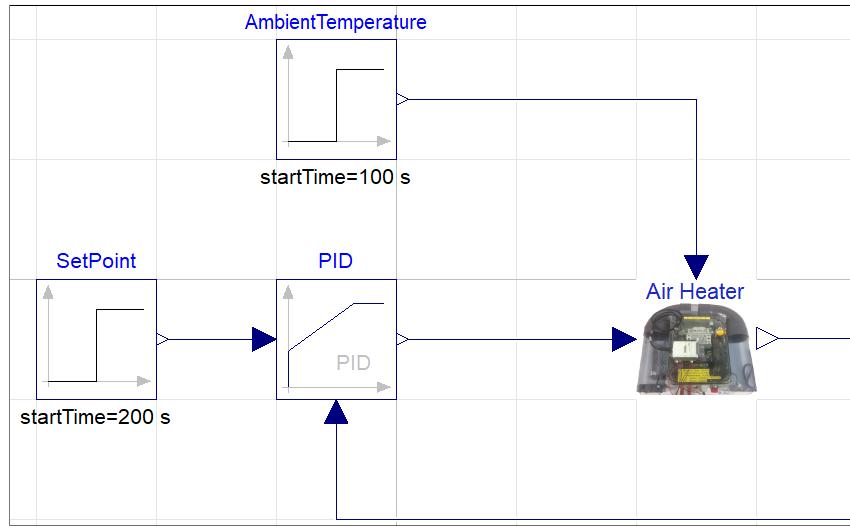
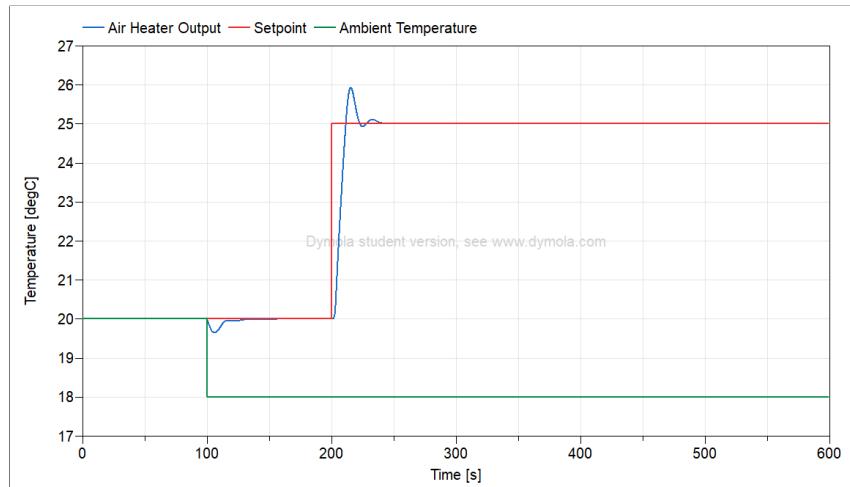


Figure 4.4: Picture of air heater with PID controller



Draft

Figure 4.5: Plot of air heater with PID controller

## 4.2 Model of Air Heater Based on Transfer Function Model

The model parameters presented in Table 4.1 was utilized in a general 1.order transfer function with time-delay presented in Equation 4.6.

Based on the general form the transfer function and the time-delay were implemented in Modelica with the use of a transfer function block and a fixed delay block as displayed in Figure 4.6. In addition to the model parameters, it was added a reference signal input,

ambient temperature input, and a temperature output from the air heater.

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

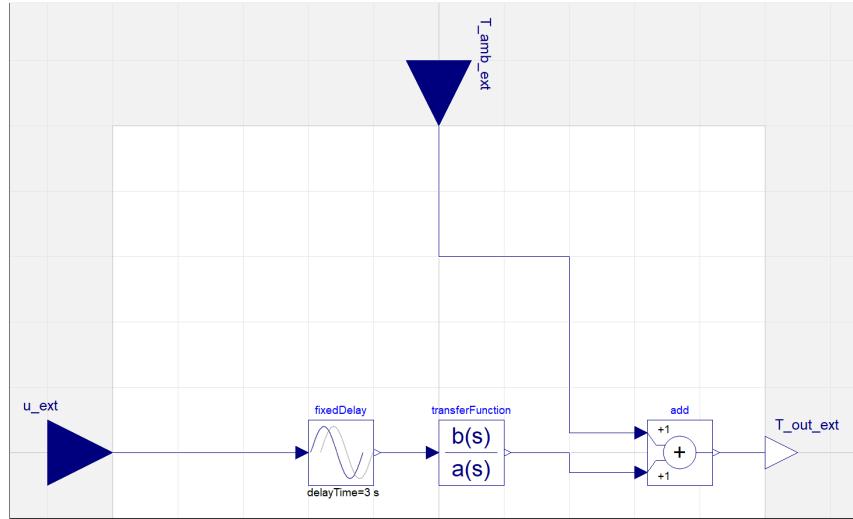


Figure 4.6: Air heater model implemented as a transfer function in Modelica

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

### 4.3 Model of Air Heater Based on State Space Model

To transform the air heater system from a transfer function model to a state-space model, the identical transfer function employed in Section 4.2 was implemented in Python and is available in the GitHub repository Advanced-Control-Implementation-with-Modelica. This implementation utilized the control library and the control MATLAB extension, as demonstrated in Code 4.2.

First 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 put together with the series function, the transfer function model was converted to a continuous state space model. A step response was then applied to both models before the resulting difference between the two step responses was calculated to display the maximum and

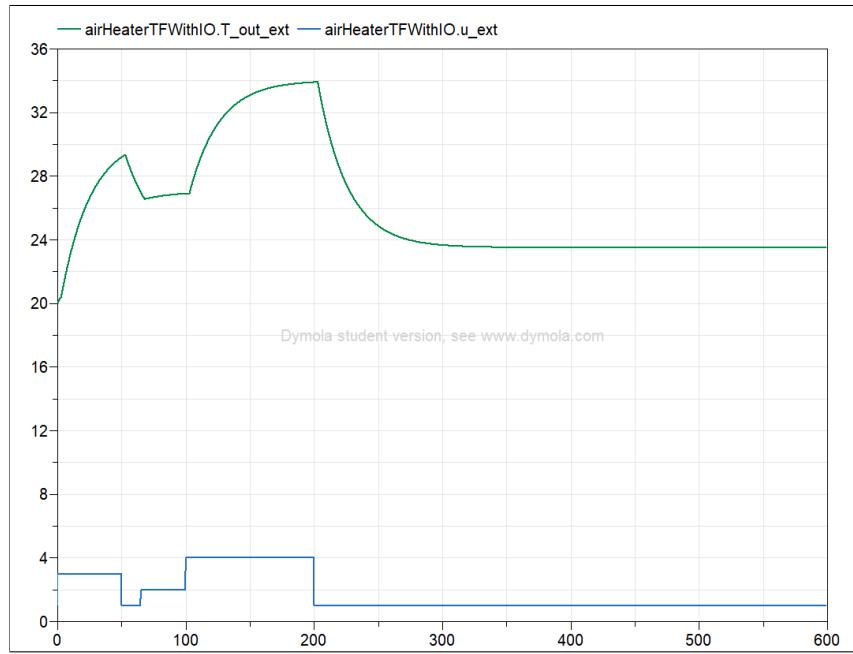


Figure 4.7: Plot from simulating transfer function model of air heater

minimum differences between the two models. For this current implementation both results were 0.00.

Finally, the continuous state space model is written to individual .CSV files for A, B, C and D matrices in addition to calculating a discrete time state space model using Euler and a time-step of 0.01 [s] that was exported the same way as the continuous model.

```

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 #System parameters and constants
8 # System Gain[C/V]
9 K = 3.5
10
11 # Time constant [s]
12 T = 23
13
14 # System 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 # Perform 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](#)

Draft

The exported .CSV files generated in the Python script that is presented in Code 4.2 was imported into Modelica with the use of one continuous and one discrete time state space model blocks as displayed in 4.8. In addition, the ambient temperature was added as an input to the model blocks with the use of two add blocks.

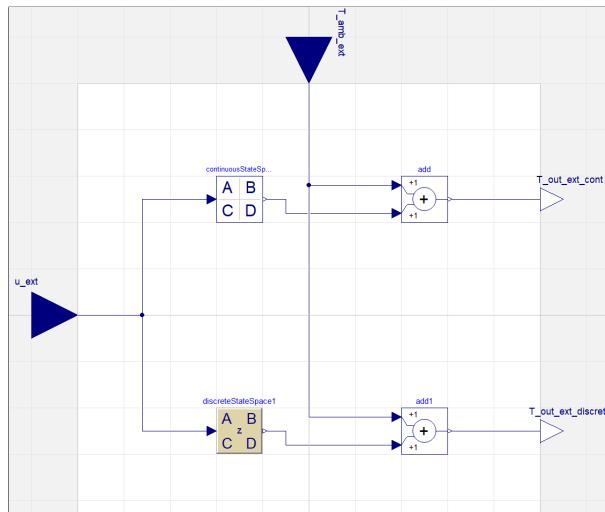


Figure 4.8: Air heater implemented as continuous and discrete time state space models.

## 4.4 Data Validation and Analysis

All the three generated blocks based on first-principles, transfer function and state space were set up and excited with the same control signal sequence as displayed in Figure 4.9.

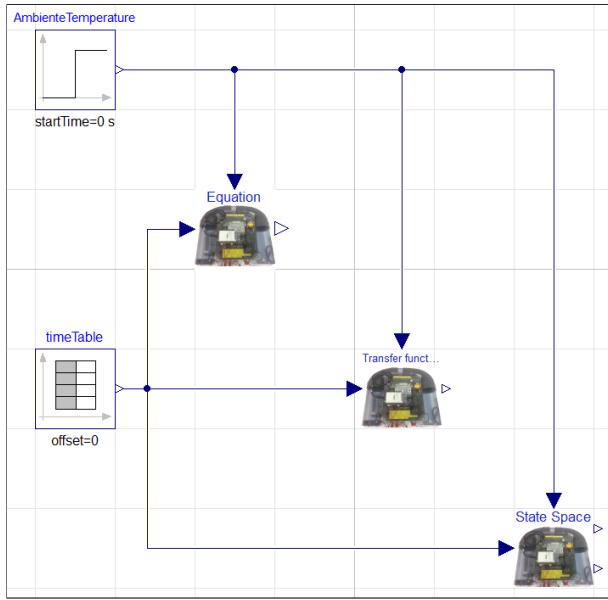


Figure 4.9: Simulation setup of air heater models based on equations, transfer functions, and state space.

The resulting plot is presented in Figure 4.10, where one can observe minor differences in the response. The primary reason for the deviation is attributed to a discrepancy at  $t = 0^+$ , where the state-space models incorporate the time delay using the Padé approximation and, consequently, begin at  $t = 0$ .

## 4.5 Continuous Stirred-Tank Reactor (CSTR)

In this chapter there is a presentation of the implementation of a CSTR model in Modelica that is based on an example given in the book Process Dynamics and Control written by Seborg et al [27].

The CSTR model presented in Figure 4.11 represents a relatively simple first-order chemical reaction model when compared to tubular reactors or packed-bed reactors. This

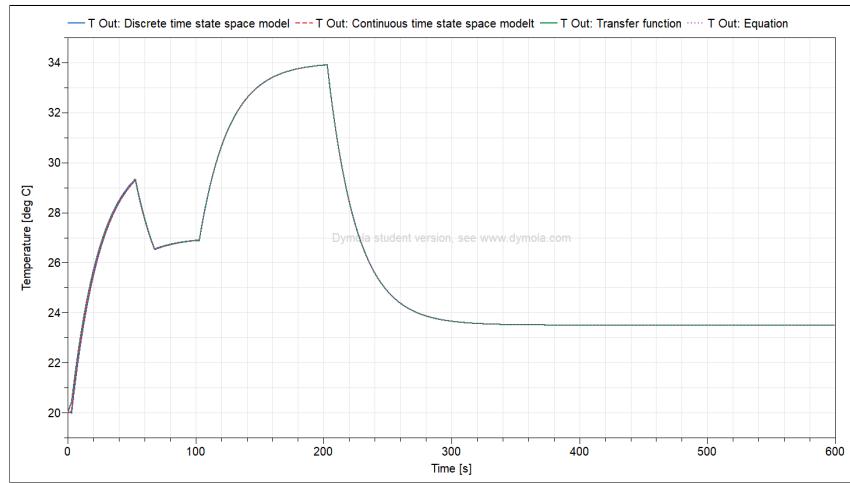


Figure 4.10: Plot of simulation setup of air heater models based on equations, transfer functions, and state space.

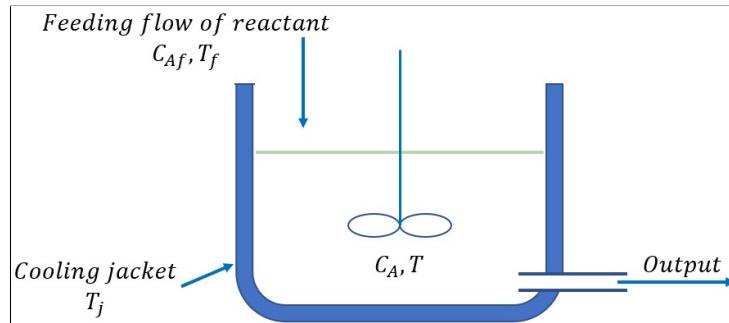


Figure 4.11: Sketch of the CSTR model.

simplicity makes the CSTR a convenient choice for illustrating modeling principles for chemical reactors. The implemented model is based on certain assumptions given in Table 4.3.

Table 4.3: CSTR model assumptions

1.	Perfectly mixed
2.	Mass densities of feed and product are equal and constant
3.	Liquid volume is constant
4.	Thermal capacitances of coolant and cooling coil is negligible compared to liquid in tank
5.	All coolant is at uniform temperature
6.	Rate of heat transfer from content of reactor to coolant is given by $Q = U \cdot A(T_c - T)$
7.	Enthalpy change associated with the mixing of feed and liquid is negligible
8.	Shaft work and heat losses to ambient can be neglected

The CSTR model has one manipulated input that represents the jacket cooling temperature, with an addition of the two states being temperature of reactant inside the tank and the concentration of the reactant. The two states also represent the two outputs of the system. The variables are presented in Table 4.4

Table 4.4: Constant stirred-tank reactor variables

Variable:	Unit:	Description:
$T_j$	[K]	Jacket cooling temperature
$T$	[K]	Temperature of reactant
$C_A$	[ $\frac{kgmol}{m^3}$ ]	Concentration of reactant

The continuous-time dynamics of the system are derived from mass and energy balances presented in Equation 4.8 and Equation 4.7 respectively.

$$\dot{C}_A = \frac{F}{V} \cdot (C_{Af} - C_A) - k_0 \cdot \exp \frac{-E}{R \cdot T} \cdot C_A \quad (4.7)$$

$$\dot{T} = \frac{F}{V} \cdot (T_f - T) - \frac{\Delta_H}{\rho \cdot C_p} \cdot k_0 \cdot \exp \frac{-E}{R \cdot T} \cdot C_A + \frac{U \cdot A}{\rho \cdot C_p \cdot V} \cdot (T_j - T) \quad (4.8)$$

With the values and descriptions for the parameters presented in Table 4.5.

In Code 4.3 one can observe the implementation of the CSTR model in Modelica based on the CSTR parameters, variables, and equations. In addition, there was added an if statement to perform several step changes in the jacket temperature while simulating.

The resulting plot of the inputs and outputs is presented in Figure 4.12. From the figure, one can observe that an increase in the control input, which represents a temperature increase in the jacket, leads to an increase in the temperature of the medium in the reactor and a decrease in concentration. Based on this observation, the two process variables exhibit direct action and reverse action, respectively.

```

1 model ContinuousStirredReactor
2
3     // Import
4     import Modelica.Units.SI;
5
6     // Constants
7     constant Real R = 1.98589           "gas constant";
8
9     // Parameters
10    parameter SI.Temperature V = 1.0      "Liquid volume of reactor";
11    parameter Real F = 1.0                  "Volumetric flow rate";
12    parameter Real E = 11843.0             "Activation energy";
13    parameter Real UA = 150.0              "U=overall heat transfer; A heat transfer area";
14    parameter Real delH = -5960.0          "Heat of reaction per mole";
15    parameter Real rhoCp = 500.0           "mass dencity and heat capacity";
16    parameter Real k0 = 34930800.0         "Frequency factor";
17    parameter SI.Temperature Tf = 298.15   "Temperature of liquid";
18    parameter Real CAf = 10                "Concentration";
19
20    // Variables
21    // Manipulated input
22    SI.Temperature Tj      "Jacket cooling temperature";
23
24    //Outputs
25    SI.Temperature T      "Temperature of medium in reactor";
26    SI.Concentration CA   "Concentration of reactant inside the tank";
27
28    equation
29
30    if time>0 and time<100 then
31        Tj = 280;
32    elseif time>=100 and time<130 then
33        Tj = 300;
34    elseif time>=130 and time<200 then
35        Tj = 305;
36    elseif time>=200 and time<400 then
37        Tj = 250;
38    else
39        Tj = 320;
40    end if;
41
42    der(T) = (F/V)*(Tf-T)-(delH/rhoCp)*k0*exp(-E/(R*T))*CA+((UA)/(rhoCp*V))*(Tj-T);
43    der(CA) = (F/V)*(CAf-CA)-k0*exp((-E)/(R*T))*CA;
44
45 end ContinuousStirredReactor;

```

Code 4.3: Continuous stirred tank reactor Implemented in Modelica - [Link to source code](#) .

Draft

Table 4.5: Constant stirred-tank reactor parameters

Variable:	Value:	Unit:	Description:
$V$	1.0	[ $m^3$ ]	Liquid volume of reactor
$F$	1.0	[ $\frac{m^3}{hr}$ ]	Volumetric flow rate
$R$	1.98589	[ $\frac{kcal}{kgmolK}$ ]	Gas constant
$E$	11843.0	[ $\frac{kcal}{kgmol}$ ]	Activation energy
$UA$	150.0	[ $\frac{kcal}{(K\cdot hr)}$ ]	U=overall heat transfer; A=heat transfer area
$\Delta_H$	-5960.0	[ $\frac{kcal}{kgmol}$ ]	Heat of reaction per mole
$\rho C_p$	500.0	[ $\frac{kcal}{(m^3 \cdot K)}$ ]	Mass density and heat capacity
$k_0$	34930800.0	[ $\frac{1}{hr}$ ]	Frequency factor
$T_f$	298.15	[ $K$ ]	Temperature of liquid
$C_{Af}$	10.0	[ $\frac{kgmol}{m^3}$ ]	Concentration

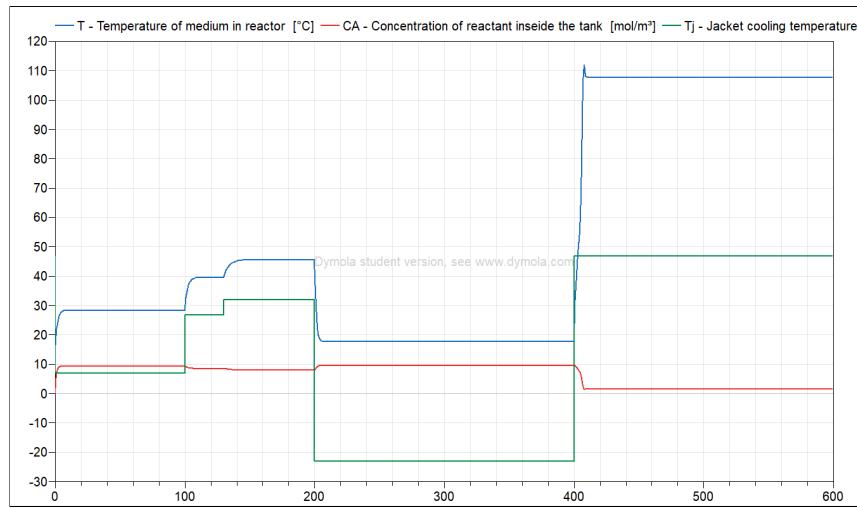


Figure 4.12: Simulation results for the implemented CSTR.

# 5 Different Methods of Calling External C Code in Modelica

---

In this chapter, the implementation of two different methods for calling external functions written in C code from an OpenModelica environment is presented. These two methods 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, simple .C file and is best suited for calling relatively small external functions from C. This function cannot be utilized directly within the graphical environment but needs 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. This external object incorporates a constructor for 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

When creating a new package in Modelica, one will have two options regarding how Modelica manages packages and files:

1. Choose to store all packages and files in a single file.
2. Choose to replicate the file structure presented in Modelica, including sub-folders, on the host computer.

To achieve the second option, simply un-check the box as shown in Figure 5.2. This approach allows for maintaining the same file structure on the computer, enabling the option to upload C source code into the file structure and access it from within OpenModelica.

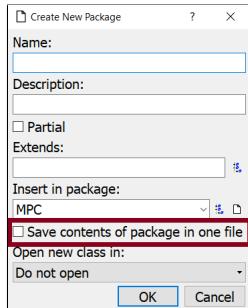


Figure 5.2: Configure Modelica folder structure

Figure 5.3 displays an example of the replicated file structure. Pictures utilized for Modelica models are stored in the 'Images' folder, while the .C files reside in the 'Source' folder. The .H files can be found within the 'Include' folder, and the .a compiled archive files are located inside the 'library/win64' directory. The file locations will be according to the Modelica standard. Utilizing these standard Modelica locations eliminates the need to specify a file path when calling external functions, as these paths are integrated into the Modelica default search paths during program compilation.

All simulations in the following sub-chapters utilized the simulation setup presented in Table 5.1.

Draft

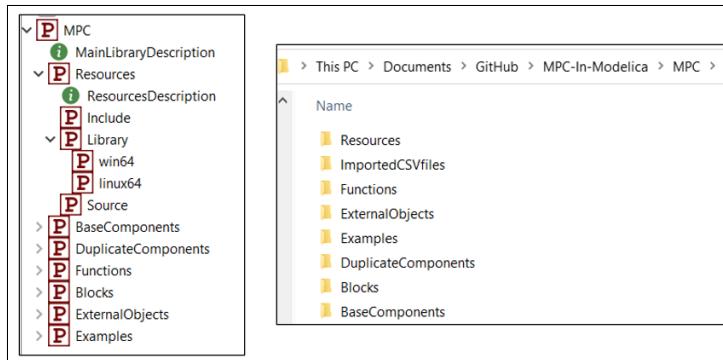


Figure 5.3: Replication of Modelica file structure on host computer

Table 5.1: Modelica simulation setup

Variable:	Value:
Start Time	0.0
Stop Time	600.0
Number of intervals	1000
Method	DASSL
Tolerance	$1^{-6}$

Draft

## 5.2 Modelica Function

In this chapter there is a presentation of the four stages for how the external C function was set up and run in Modelica. The main flow of the four stages in calling external C functions is presented in Figure 5.4.

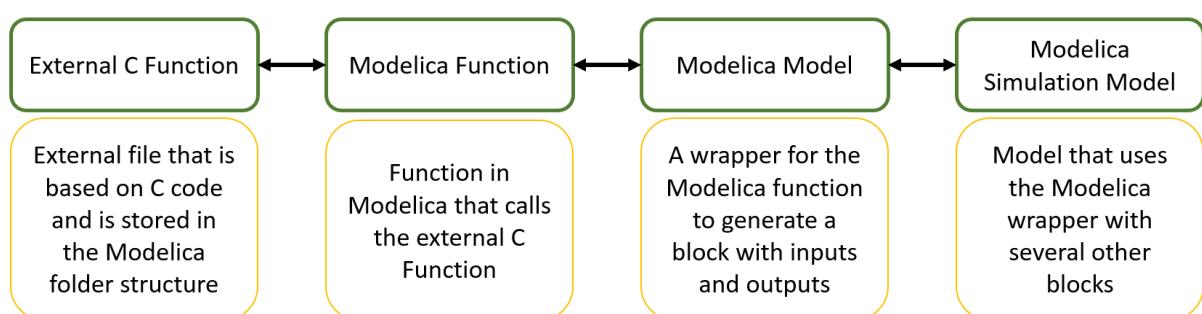


Figure 5.4: Process flow of calling external functions from Modelica

### 5.2.1 External C Function

In this chapter there is a presentation of the C function that was set up to be called from inside Modelica. The C function is displayed in Code 5.1 and is based on the pseudo random function rand() in C. The function returns a pseudo random number based on a given range as input.

```

1 #include <stdio.h>
2 #include <stdlib.h>
3
4 double NoiseGenerator(double range)
5 {
6     double result = 0;
7     double span = 2*range;
8
9     //Only return the noise component
10    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

As external functions cannot be called directly from models, one must set up a Modelica function to invoke the external 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 (return values) 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.

IncludeDirectory shows the relative file path with reference to the root package named MPC in this example. 'NoiseGenerator.C' is the file name residing inside the source folder. While it is not necessary to include the path for this example, it has been added for informational purposes.

```

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\"");
```

```
11 end extNoiseGenerator;
```

Code 5.2: Calling External Function in Modelica - [Link to source code](#)

### 5.2.3 Modelica Function Call Wrapper

The wrapper calling the Modelica function, as presented in Code 5.2, is shown in Code 5.3. From the code, one can observe the addition of an external connector 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.

```
1 model NoiseGenerator "Simple random noise generator"
2
3 //Parameters
4 parameter Real range = 0.5;
5 parameter Real samplePeriode = 1.0;
6
7 //External connectors
8 Modelica.Blocks.Interfaces.RealOutput noiseGenerator_out
9   annotation (Placement(transformation(extent={{100,-10},{120,10}})));
10
11 initial equation
12   noiseGenerator_out = Functions.extNoiseGenerator(range);
13
14 equation
15   when sample(0, samplePeriode) then
16     noiseGenerator_out = Functions.extNoiseGenerator(range);
17   end when;
18
19 end NoiseGenerator;
```

Code 5.3: Wrapper for Modelica Function - [Link to source code](#)

### 5.2.4 Modelica Simulation Model

An example of a Modelica simulation where the measurement noise is generated from the external function is presented in Figure 5.5 where the noise is added to the process output measurement by a Modelica add block.

Both the process output measurement and the signal with the added noise are displayed in Figure 5.6. Since the model does not include any filtering the signal noise is either accumulated or equalized based on the phase differences between the generated noise and the process measurement.

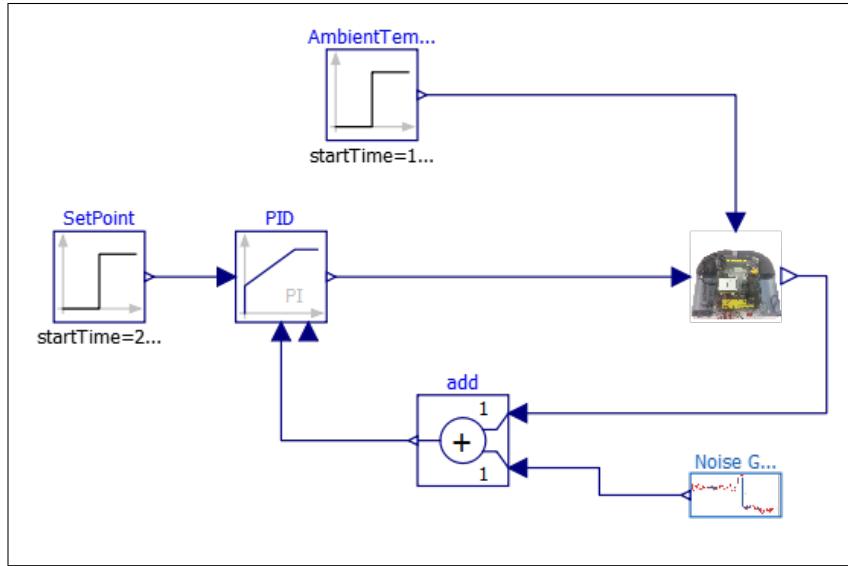


Figure 5.5: Model for simulating noise generated from external C code

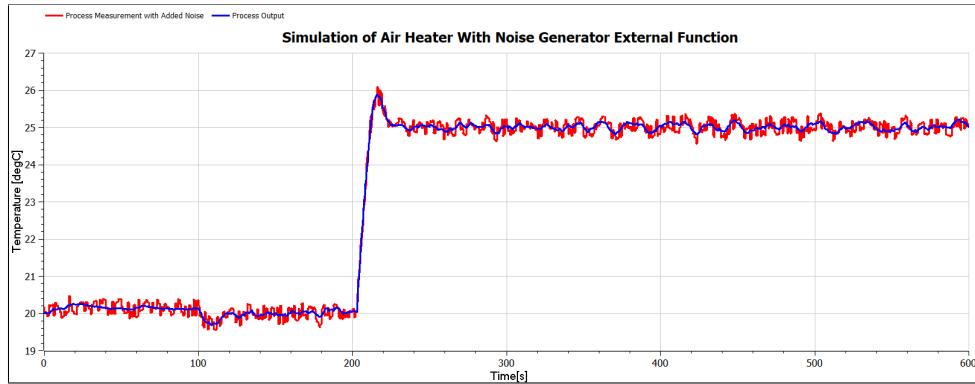


Figure 5.6: Plot of process output with both existing and newly added noise

### 5.3 Modelica External Objects

The main difference between using the Modelica model and the 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. Two primary advantages of employing external objects in Modelica for calling external C code are the capacity to store variables and values in external memory and the assurance of the correct order of initialization and termination of external memory.

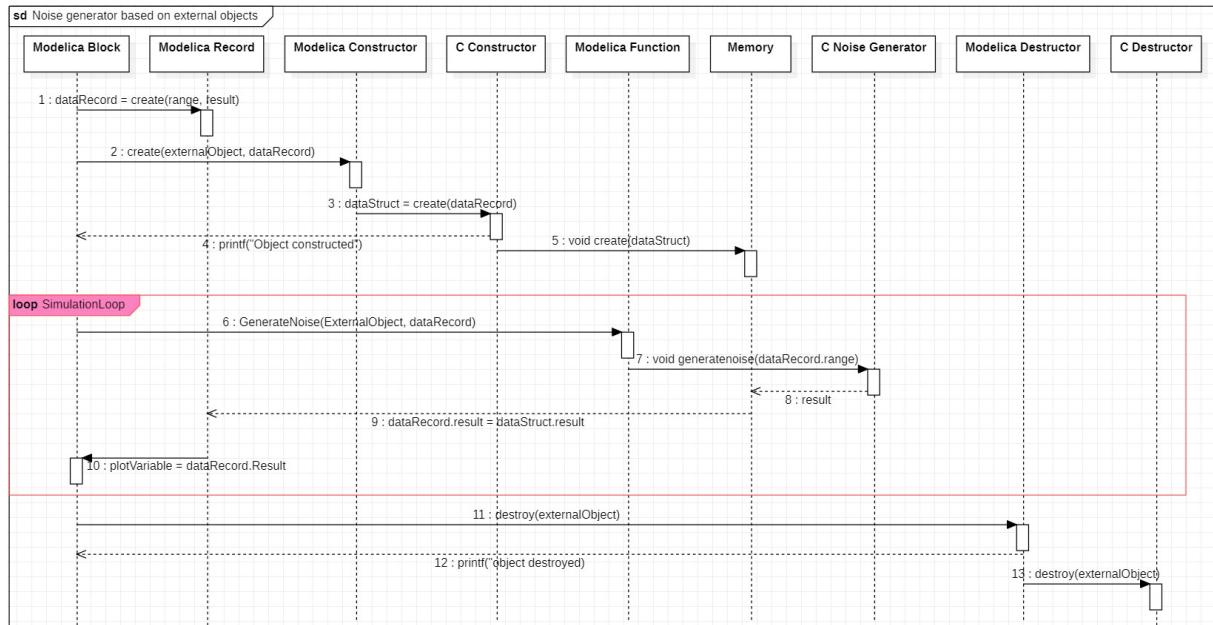


Figure 5.7: Sequence diagram for calling external noise generator with the use of Modelica external objects.

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. It is crucial that both the record and struct use identical variable names.

Within the loop presented in the sequence diagram one can observe 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.

### 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 must be the same as utilized for the struct in the C source code presented in 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 external object class in Modelica should not include anything else but a constructor and a destructor function, based on the Modelica specification. The source code for the constructor is presented in Code 5.6 where one can observe the extension of the Modelica external objects in Line 2. Further one can see that 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 displayed in Code 5.6 is displayed in Code 5.7. The "malloc()" function in Line 4 allocates the space in memory with the size of the NoiseGeneratorData struct. In Line 9 and 10 the initial values passed from the Modelica environment are written as initial values 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:/MFC/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 presented in Code 5.9. After the memory has been successfully freed in Line 6 there is a print statement that writes back 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 one has the requirement to return more than one value / variable the previously defined record 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:/MFC/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 newly calculated values to the Modelica environment. In Line 10 one can observe that the output object is updated with the calculated new random number based on the range that was 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](#)

Draft

## 5.4 Data Validation and Analysis

The resulting plot from both implementation methods discussed in Chapter 5.2 and Chapter 5.3 is presented in Figure 5.8. These 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 noise.

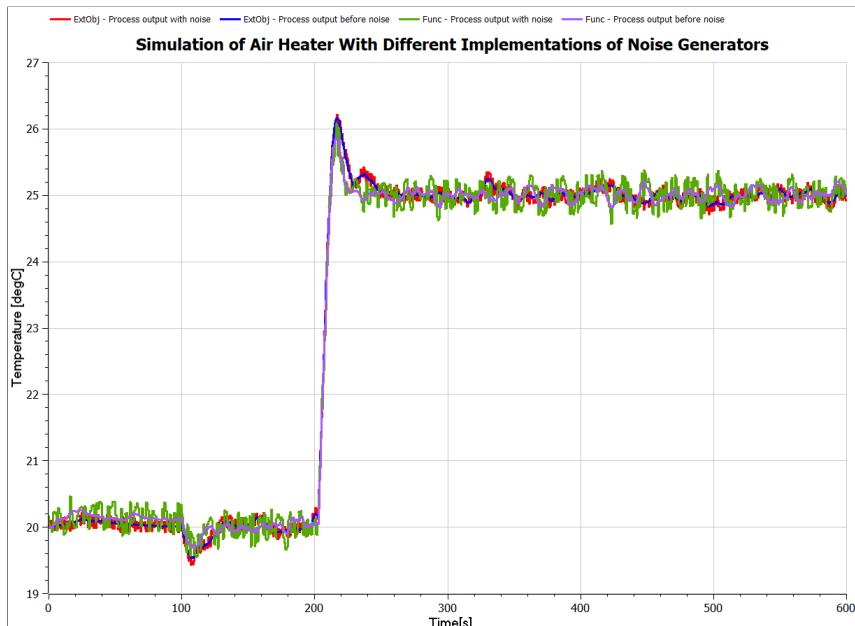


Figure 5.8: Simulation of PID with the different implementations of the noise generator.

The data presented in Figure 5.8 was exported as .CSV files and subsequently imported into a Jupyter notebook using the source code available in a GitHub repository link: ([Link to source code](#)). By adding data points to the plot for each sample and focusing on a time interval of 5 seconds, one can observe a distinction between the output of the external object implementations in red and the function-based implementation in green.

The primary difference between these implementations lies in their execution mechanisms. The external object runs using the Modelica algorithm, executing code sequentially at specific time steps with a frequency of 0.5 seconds. This frequency is determined by the simulation duration of 600 seconds and the number of intervals set at 1200. On the other hand, the equation-based simulation relies on intervals chosen by OpenModelica.

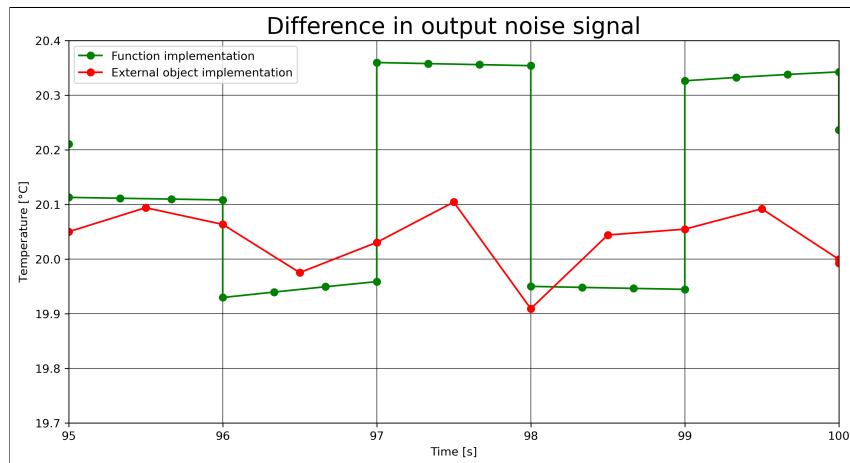


Figure 5.9: Simulation result of PID with all the different implementations in a time interval of 10 seconds.

In Table 5.2 one can observe 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 parameters is quite small and a result of rounding.

Table 5.2: Statistical values 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

---

This 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.

---

In figure 6.1 there is a presentation of 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 as a simple 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.

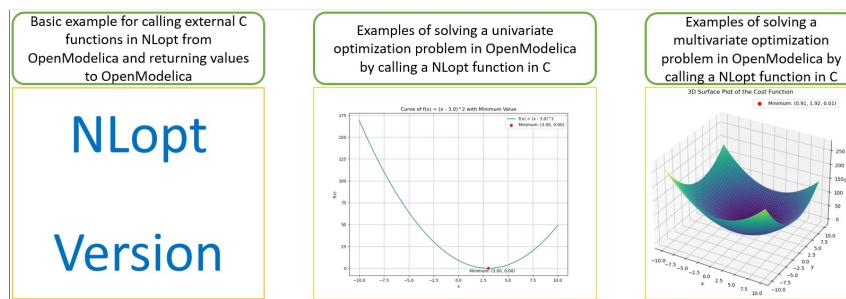


Figure 6.1: Three different blocks implemented in OpenModelica

## 6.1 Installation of the NLOpt Library on Windows

The NLOpt library was 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 current directory. Recommended paths for cloning include the following:

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

Clone the repository with this command:

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

The next step is to build the vcpkg with bootstrap:

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

Then finally 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 presented in Line 1 below. Line 2 provides a reference to the Modelica implementations presented in the GitHub repository, indicating where the .DLL file needs to be moved. Access to the MPC repository is available at the following link:  
[Link to repository](#).

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

The NLOpt .H file must be moved 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 basic 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 a GitHub repository at the following link: [Link to source code](#). The corresponding C source code is also available in the same repository at this link: [Link to source code](#).

From the source code presented in Code 6.1, the definition of the library and include directory is evident in Lines 2 and 4, even though they are part of the standard search path for OpenModelica, the path in this example 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 also features a counter, utilized 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 environment in OpenModelica are presented in Figure 6.3. In this figure one can observe the individual variables extracted from the

record in OpenModelica, each associated with the corresponding version number of the installed package.

Variables	Value
NAZ (Active) NLOptVersionBlock	
nloptversion2Struct	
bugfix	1
counter	0
major	2
minor	7

Figure 6.3: Output in OpenModelica simulation environment

## 6.3 Implementation of the NLOpt Univariate Optimization

The version function as introduced in Chapter 6.2 is in this implementation extended to enable the passing of parameters from the OpenModelica environment into the optimization environment presented in the C source code. Furthermore, additional checks were incorporated into the C source code to assess the values supplied to the C 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 function was implemented as a function within the C optimization function, which is called from the Modelica function.

$$f(x) = (x - 3.0)^2 \quad (6.1)$$

As a reference a 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.

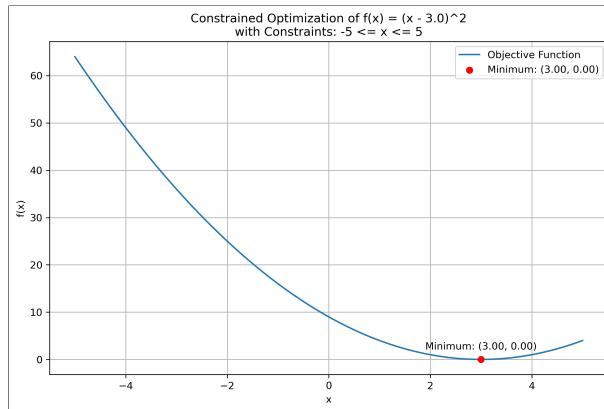


Figure 6.4: Plot of optimized univariate function - [Link to source code](#)

### 6.3.2 Optimization Parameters

In Figure 6.5 the diagram displays the path of flow for the parameters from the graphical user interface when interacting with the "NloptUniOptiBlock" block implemented in OpenModelica to the parameters being stored in the computer memory. This chapter will concentrate on demonstrating how this flow was implemented across OpenModelica and C.

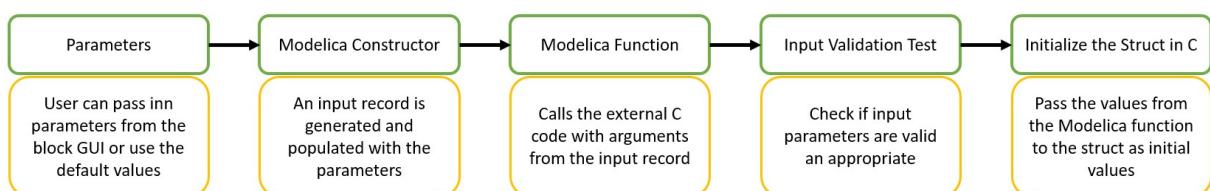


Figure 6.5: Diagram of flow for parameter from GUI all the way to allocated space in memory

## 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.

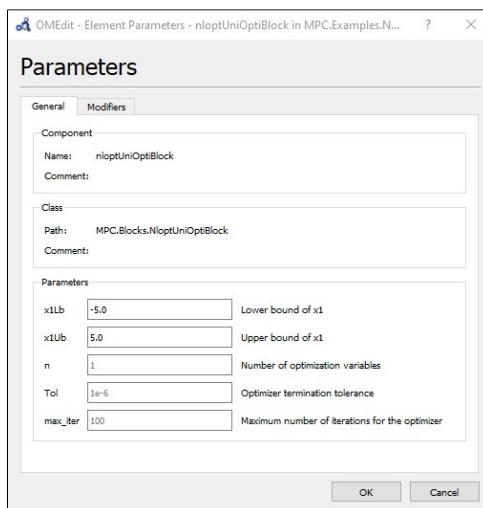


Figure 6.6: Available parameters for the Modelica block

## 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 presented 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 and last part of the constructor function implemented in OpenModelica is displayed in Code 6.4. In Line 2 one can observe the values from the previously constructed record in Code 6.3 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,
3     optimizationDataInput.x1UbR, optimizationDataInput.min_costR, optimizationDataInput.nR, optimizationDataInput.TolR,
4     optimizationDataInput.max_iterR)
5     annotation(IncludeDirectory = "modelica://Resources/Include/",
6             Include = "#include \"nloptUniOptimize.c\"");
7 end constructor;

```

Code 6.4: Modelica constructor function call - [Link to source code](#)

## Input Validation test

The source code presented in Code 6.5 demonstrates tests that were implemented for evaluating the values utilized to initialize the C struct, used when constructing the external object. The "ModelError()" 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 one can observe that there is allocated a place in memory with the function ”malloc()” used in Line 2. The size of the allocated memory 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 complete 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.

The optimization function in Code 6.8 is called through the Modelica ”nloptOptimizationFuncCall()” as presented in Code 6.1. In Code 6.8 one can observe the head of the optimization function implemented in C. In Line 1 one can observe 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](#)

In Code 6.9 there is a presentation of 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](#)

From Line 2 in Code 6.10 one can observe the initialization of the optimization object based on the COBYLA optimizer and variable "n" passed from the OpenModelica parameters representing the number of optimization variables. In Line 5 the objective function as presented in Code 6.9 is defined together with the optimization object.

```

1 // Create an NLOpt optimizer
2 nlopt_opt optimizer = nlopt_create(NLOPT_IN_COBYLA optimizationDataInput->n); // Use the IN_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](#)

The optimization is performed in Code 6.11 where the results from the optimization and the minimum cost is written straight to the previously 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 presented 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

In Figure 6.7, a presentation of the simulation results is shown after running the "NloptUniOptiBlock" in OpenModelica. From the complete 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 are the same values as calculated in the Python implementation presented in 6.4.

Variables	Value	Description
✓ <input checked="" type="checkbox"/> (Active) NloptUniOptiBlock		
<input type="checkbox"/> Tol	1e-06	Optimizer termination tolerance
<input type="checkbox"/> max_iter	100	Maximum number of iterations for the optimizer
<input type="checkbox"/> n	1	Number of optimization variables
> optimizeData		
summary		
<input type="checkbox"/> min_cost	0	Minimum value of the objective function after optimization
<input type="checkbox"/> x1	3	Optimization variable
<input type="checkbox"/> x1Lb	-5.0	Lower bound of x1
<input type="checkbox"/> x1Ub	5.0	Upper bound of x1

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 presented in Chapter 6.3, with the addition of one additional optimization variable and two additional constraints compared to the univariate example. This chapter presents the differences in implementation compared to the univariate implementation.

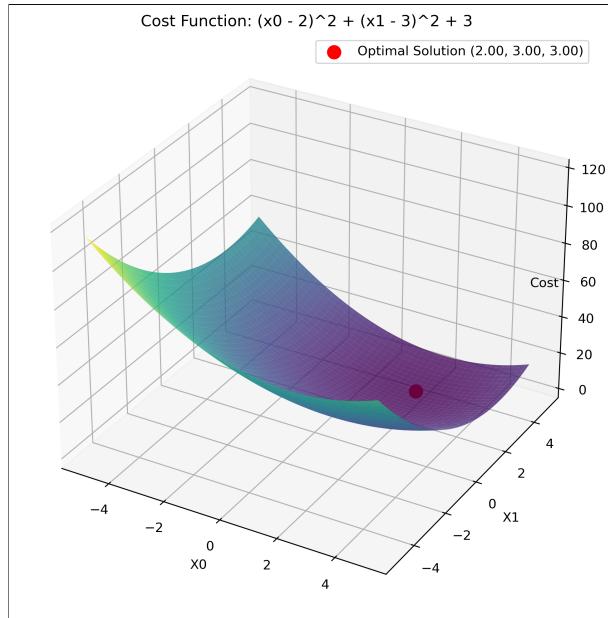
The source code for the Modelica block implemented in OpenModelica is available in a GitHub repository at the following link: [Link to source code](#). While the C source code including the optimizer, can be found in a GitHub repository at this link: [Link to source code](#).

### 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  $x_1$ .

$$f(x) = (x_0 - 2)^2 + (x_1 - 3)^2 + 3 \quad (6.2)$$

Figure 6.8 displays a plot generated by performing the optimization utilizing the SciPy library in Python. The lower bound was set to -5 and upper bound at 5 for both variables. The resulting values for 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.



Draft

Figure 6.8: Plot of optimized multivariate function - [Link to source code](#)

#### 6.4.2 Optimization Parameters

Code 6.13 presents how the referencing of the optimized values given as vector elements within C. Due to the absence of support for passing vectors between C and Modelica, these values are written as separate variables into the OpenModelica record utilizing pointers to the struct.

```

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

The source code presented in Code 6.14 performs 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 optimized output is presented as 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 block in OpenModelica are displayed in Code 6.9. The results present relatively minor differences when compared to the Python simulation results presented in Figure 6.8. The observed differences are likely caused by floating-point round-off errors.

Variables	Value	Description
✓  (Active) NloptMultiOptiBlock		
<input type="checkbox"/> Tol	7e-06	Optimizer termination tolerance
<input type="checkbox"/> max_iter	101	Maximum number of iterations for the optimizer
<input type="checkbox"/> n	2	Number of optimization variables
> optimizeData		
<input type="checkbox"/> summary		
<input type="checkbox"/> min_cost	3	Minimum value of the objective function after optimization
<input type="checkbox"/> x1	1.99992	Optimization variable
<input type="checkbox"/> x2	2.9999	Optimization variable
<input type="checkbox"/> x1Lb	-5.0	Lower bound of x1
<input type="checkbox"/> x1Ub	5.0	Upper bound of x1
<input type="checkbox"/> x2Lb	-5.0	Lower bound of x2
<input type="checkbox"/> x2Ub	5.0	Upper bound of x2

Figure 6.9: Simulation results from the NLOpt multivariate block

## 7 FMU of Air Heater in Python

---

This chapter features a presentation of the process for importing the exported FMUs into Python to conduct performance comparisons. The chapter contains portions of the source code from the Python implementation using Jupyter Notebook. The complete source code and installation information can be found in the GitHub repository ([Link to repository](#)).

---

The implementations presented in this chapter are based in 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.

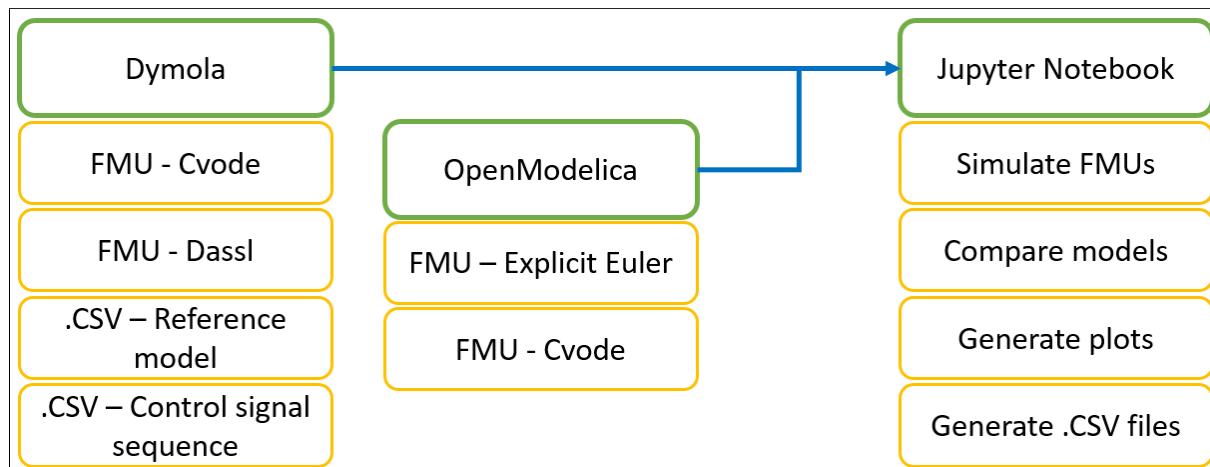


Figure 7.1: Overview of process for generating FMUs and evaluating the FMUs in Python.

The simulation of the reference model of the air heater in Dymola resulted in an .CSV file export. This exported .CSV file served 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

In Code 7.1, the provided source code demonstrates the process of importing the four FMUs into the Python environment. This 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 presented in a GitHub repository link: ([Link to source code](#)).

```

1 import fmpy
2 import matplotlib.pyplot as plt
3 import numpy as np
4 from fmpy import *
5
6 # Import FMU model generated with explicit Euler in OpenModelica into a variable
7 airHeaterOMexpEul = 'AirHeaterFMUexpEul/MPC.DuplicateComponents.AirHeaterWithIO.fmu'
8
9 # Import FMU model generated with Cvode in OpenModelica into a variable
10 airHeaterOMcvode = 'AirHeaterFMUcvode/MPC.DuplicateComponents.AirHeaterWithIO.fmu'
11
12 # Import FMU model generated with Cvode in Dymola into a variable
13 airHeaterDymolaCvode = 'AirHeaterFMUdymolaCvode/MPC_DuplicateComponents_AirHeaterWithIO_64bit_COVDESolver.fmu'
14
15 # Import FMU model generated with Dymola Dassl solver in Dymola into a variable
16 airHeaterDymolaOEM = 'AirHeaterFMUdymolaSolver/MPC_DuplicateComponents_AirHeaterWithIO_64bit_DymolaSolvers.fmu'
17
18 # Set the initial values for the inputs
19 start_values= { 'u_ext': 3.0,
20                 'T_amb_ext': 20}
21
22 # Choose what variables that will be available in plot and export
23 output = [
24     'T_Out_ext',
25     'u_ext',
26     'T_amb_ext'
27 ]
28
29 # Display the FMU model information
30 print("FMU model generated with explicit Euler in OpenModelica into a variable")
31 dump(airHeaterOMexpEul)
32 print("\n FMU model generated with Cvode in OpenModelica into a variable")
33 dump(airHeaterOMcvode)
34 print("\n FMU model generated with Cvode in Dymola into a variable")
35 dump(airHeaterDymolaCvode)
36 print("\n FMU model generated with Dymola Dassl solver in Dymola into a variable")
37 dump(airHeaterDymolaOEM)
```

Code 7.1: Import of FMU into Python - [Link to source code](#)

In the source code provided in Code 7.2, 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 was utilized in the simulation of all four FMUs. Subsequently, the simulation results were written in their respective individual .CSV files. The time interval for the exported files for both the reference model and all the FMU simulations was configured to 0.5 seconds.

```

1 # Import the CSV file that holds the control signal sequence.
2 input = np.genfromtxt('datafiles/ControlSignal2.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=
6     =input, output_interval=0.5)
7 result_airHeaterOMcvode = simulate_fmu(airHeaterOMcvode, start_values=start_values, output=output, stop_time=600.0, input=
8     =input, output_interval=0.5)
9 result_airHeaterDymolaCvode = simulate_fmu(airHeaterDymolaCvode, start_values=start_values, output=output, stop_time=600.0,
10    input=input, output_interval=0.5)
11 result_airHeaterDymolaOEM = simulate_fmu(airHeaterDymolaOEM, start_values=start_values, output=output, stop_time=600.0,
12    input=input, output_interval=0.5)
13
14 # Write simulation data to CSV files
15 write_csv('datafiles/airHeaterOMexpEulSimData.csv', result_airHeaterOMexpEul, columns=None)
16 write_csv('datafiles/airHeaterOMcvodeSimData.csv', result_airHeaterOMcvode, columns=None)
17 write_csv('datafiles/airHeaterDymolaCvodeSimData.csv', result_airHeaterDymolaCvode, columns=None)
18 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. These data frames were generated for the reference model and the four FMU simulations, as generated in Code 7.2. Afterward, the statistical properties of all the imported data were written to .CSV files.

```

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 dfOME = 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 dfDYMc = 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](#)

## 7.2 Results

The previously exported .CSV files were imported into Excel for a preliminary visual inspection of the data to identify any initial irregularities. Figure 7.2 displays the resulting plot, which indicates that there are no apparent irregularities, such as outliers or discontinuities, within the data sets.

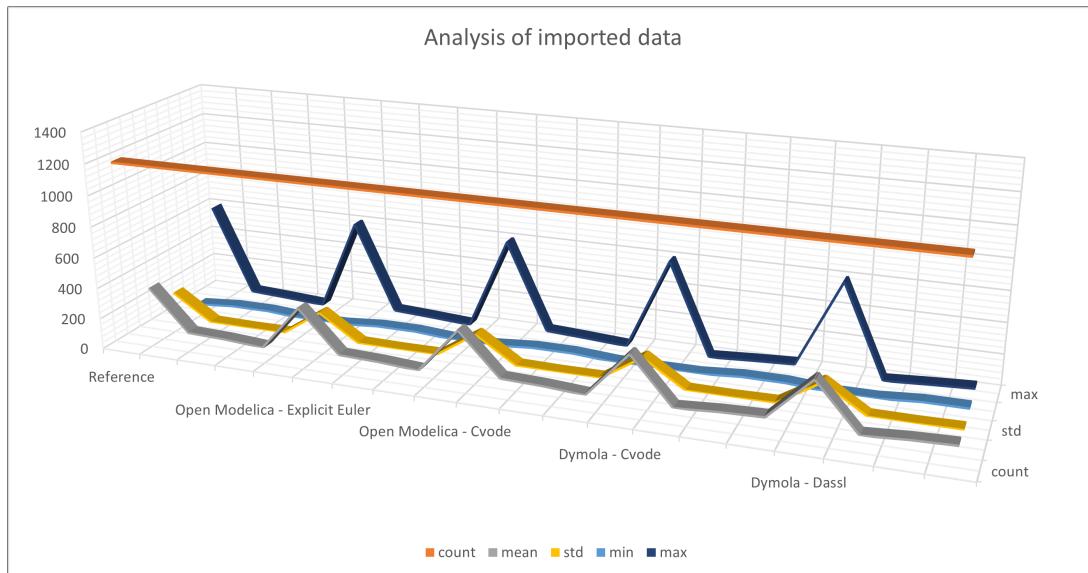


Figure 7.2: Initial analysis of imported simulated data.

In Code 7.4 it was generated a new data frame to store the difference in value between the reference model and the individual FMUs. The resulting plot is presented in Figure

7.3 where the temperature output for the reference model and all the simulated models is illustrated. The plot also includes the control signal for the simulations.

From the plot presented in Figure 7.3 all models represent the model dynamics with relatively good accuracy and precision.

```

1 # Generate a new dataframe to evaluate the difference between the different FMUs temperature output and the Dymola
2     reference simulation
3 dfTOut_diff = dfRef[['time']].copy()
4
5 # Calculate the difference between the Modelica export and OpenModelica_Explicit_Euler
6 dfTOut_diff['OpenModelica_Explicit_Euler'] = dfRef['T_Out'] - dfOMeE['T_Out_ext']
7
8 # Calculate the difference between the Modelica export and OpenModelica_Cvode
9 dfTOut_diff['OpenModelica_Cvode'] = dfRef['T_Out'] - dfOMcv['T_Out_ext']
10
11 # Calculate the difference between the Modelica export and Dymola_Cvode
12 dfTOut_diff['Dymola_Cvode'] = dfRef['T_Out'] - dfDYMcv['T_Out_ext']
13
14 # Calculate the difference between the Modelica export and Dymola_Dassl
15 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](#)

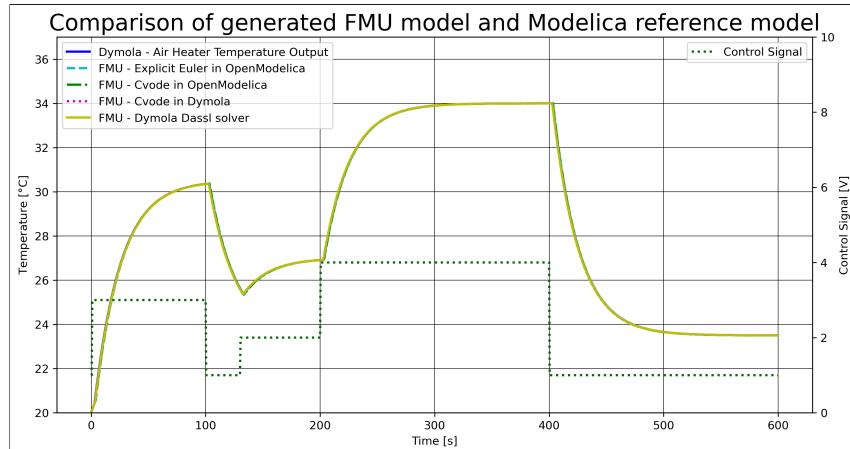


Figure 7.3: Plot of result from simulation of Python and Dymola reference.

In Figure 7.4, 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 more significant deviation from the reference model when subjected to a step change in the control signal. In contrast, FMUs employing the CVODE and DASSL solvers, known for their dynamic step size adaptation and higher-order accuracy, demonstrate a better 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.

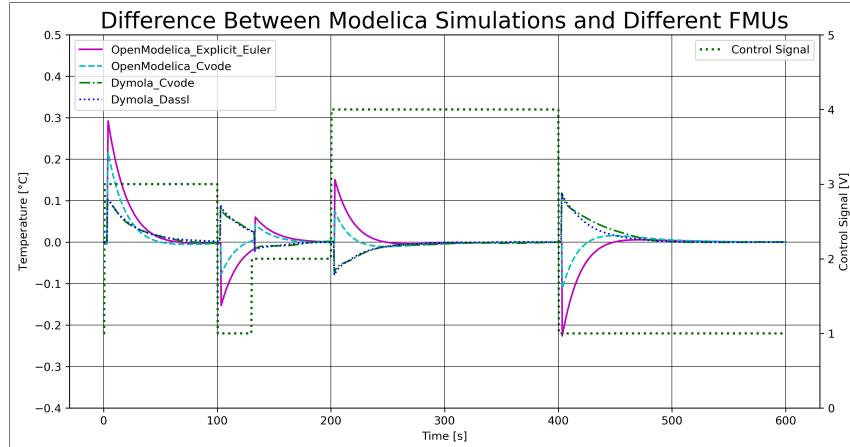


Figure 7.4: Difference in output temperature for FMU simulations in Python and Reference model from OpenModelica.

The calculated statistical properties of the data frame that contains the difference in temperature presented in Figure 7.4 was written to a new data frame presented in Code 7.5. A bar plot of the calculated statistical data frame is presented in Figure 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
5 # plotting
6 describe_df_dftOut_diff = dfTOut_diff.describe(percentiles=perc)
7
8 # Remove the variables time, counter, median, and mean
9 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](#)

Draft

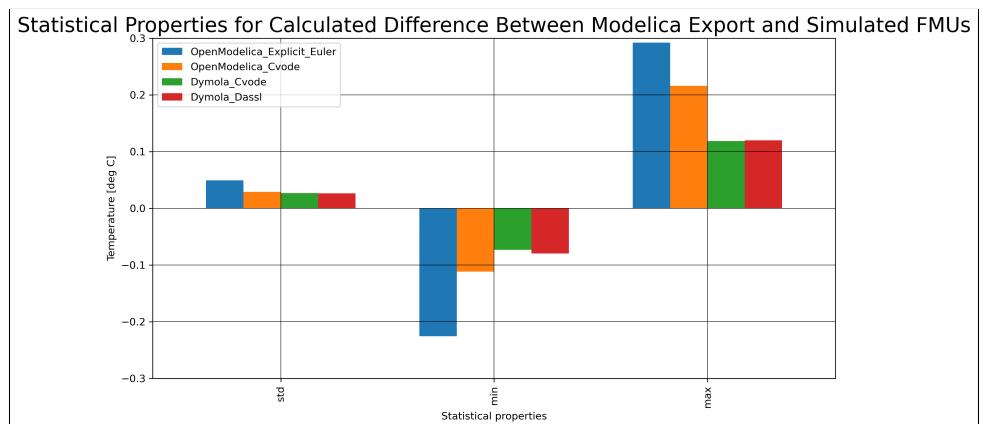


Figure 7.5: Statistical properties for the difference between Python and Modelica model.

# 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 comprehensive source code is available in the GitHub repository ([Link to repository](#)).

---

The code presented 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 this 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.

## 8.1 Implementation in Python

From the source code provided in Code 8.1, one can observe the implementation of a PI controller in Python.

```

1 def fun_pi_con(y_sp_k, y_k, u_i_km1, contr_params, dt):
2
3     (Kc, Ti, u_man, u_min, u_max) = contr_params
4     e_k = y_sp_k - y_k                      # Control error
5     u_p_k = Kc*(y_sp_k - y_k)                # P term
6     u_i_k = u_i_km1 + (Kc*dt/Ti)*e_k        # I term
7     u_i_min = u_min - u_man
8     u_i_max = u_max - u_man
9     u_i_k = np.clip(u_i_k, u_i_min, u_i_max) # Anti windup (Limit ui)
10    u_k = u_man + u_p_k + u_i_k             # PI term + man control
11    u_k = np.clip(u_k, u_min, u_max)         # Limitation of control

```

```

12     return (u_k, u_i_k)                                #Returns control signal and calculated I term
13

```

Code 8.1: PI function in Python. - [Link to source code](#)

The simulation parameters for the Python simulation is presented in Code 8.2.

```

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

```

Code 8.2: Simulation parameters. - [Link to source code](#)

Code 8.3 and Line 2 shows the construction of a dictionary named vrs{}. In Line 3, the for loop 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. This connection effectively links the variable names from the FMU to their numeric 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 set and get functions during the simulation.

```

1 # collect the value references from the FMU
2 vrs = {}
3 for variable in model_description.modelVariables:
4     vrs[variable.name] = variable.valueReference
5
6 # get the value references for the variables we want to get/set
7 fmu_ControlSignal_Input = vrs['u_ext']                      # Control signal
8 fmu_AmbientTemperature_Input = vrs['T_amb_ext']            # Ambient air temperature
9 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. Subsequently, 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.

```

1 # Initial simulation conditions
2 AmbientTemperature = 20.0
3 InitialControlSignal = 0.0
4
5 # Set the initial values in the FMU variable definitions
6 fmu.setReal([fmu_ControlSignal_Input], [InitialControlSignal])
7 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 time advances in Line 5. In Line 8, the defined inputs, outputs, and "AmbTemp" are recorded using the "fmu.getReal()" function reading the values, utilizing the FMU variable definitions. Subsequently, in Line 11, these values are added to an array for plotting purposes.

Moving to Line 13 the control signal is calculated based on the implemented PI 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,
9   fmu_AmbientTemperature_Input])
10
11 # append the results for plotting
12 rows.append((time, inputs, outputs, AmbTemp))
13
14 (controlSignal, u_i_k) = fun_pi_con(setpoint, outputs, prevIterm, contr_params, dt)
15
16 fmu.setReal([fmu_ControlSignal_Input], [controlSignal])
fmu.setReal([fmu_AmbientTemperature_Input], [AmbienteTemperature])

```

Code 8.5: FMU simulation loop - [Link to source code](#)

## 8.2 Results

In this chapter, there is a presentation of a comparison of the simulation performed with an FMU in Python and an original implementation in OpenModelica with the same changes in environmental temperature, change in temperature setpoint, the same step size, and simulation time. The complete source code can be found in the GitHub repository ([Link to repository](#)).

Figure 8.1 displays a plot of the simulation results in Python. Upon comparison with the results shown in Figure 4.5, it is evident that the plots exhibit somewhat similar dynamics.

The statistical properties of both simulations, along with the absolute values of the differences, are presented in Table 8.1. From the table, one observes that the selected step

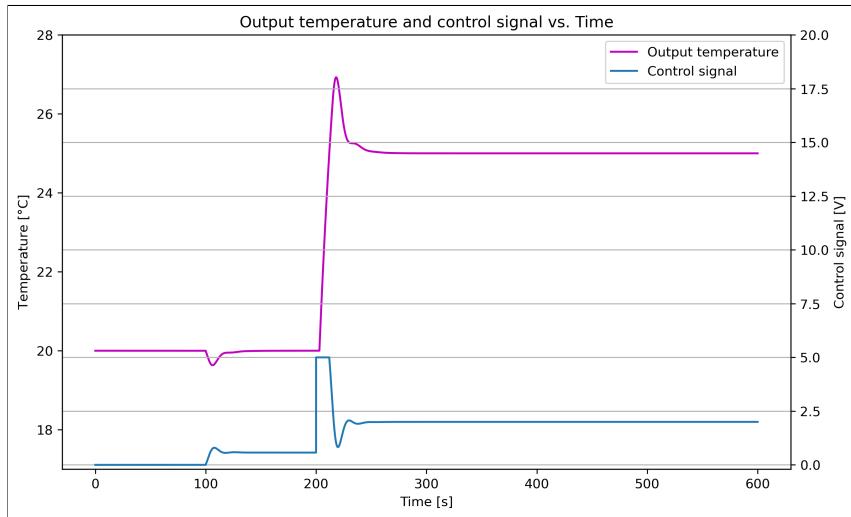


Figure 8.1: Resulting plot for simulation of PI controller and FMU of a air heater in Python.

size is relatively small, resulting in over 600,000 data points for both simulations. The 7 additional points in OpenModelica are considered to be an insignificant difference in this setting. The only difference worth noticing is the maximum output temperature with 0.98 [°C] difference between the simulations.

Table 8.1: Statistical values for both simulations

Variable	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 presented simulation results in Figure 8.2 reveal that the FMU-based Python implementation exhibits a greater overshoot in the output temperature compared to the OpenModelica simulation. This difference is also evident in the control signal from the PI controller, where the OpenModelica simulation reduces the control signal at an earlier time than what happened for the Python implementation. This discrepancy accounts for the 0.98 [°C] difference highlighted in Table 8.1. For smaller controller actions, such as compensating for the drop in ambient temperature at 100[s], both simulations have an equal response.

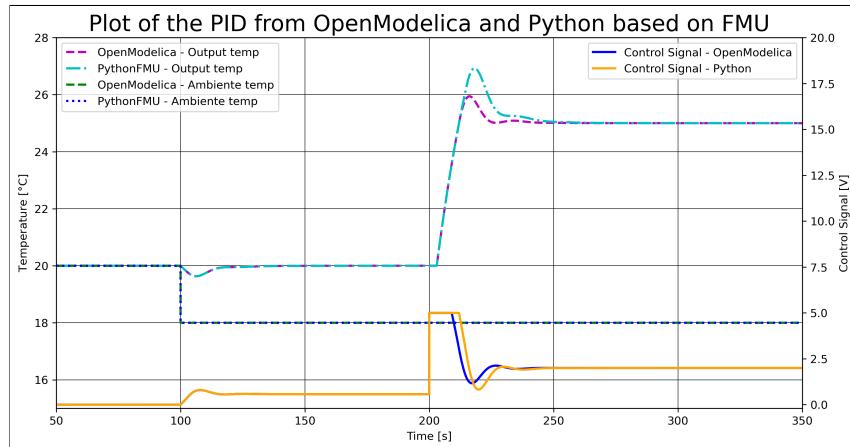


Figure 8.2: Resulting plot for simulation of PI controller and FMU of an air heater compared with simulation from OpenModelica.

The plot in Figure 8.3 shows the contrast between the two simulations, highlighting variations in both the control signal and the resulting output temperature. This difference is probably due to the distinct approaches utilized in implementing the PI controller in their respective programming languages. To bring the Python-implemented controller closer to the OpenModelica simulation, additional parameter tuning may be necessary.

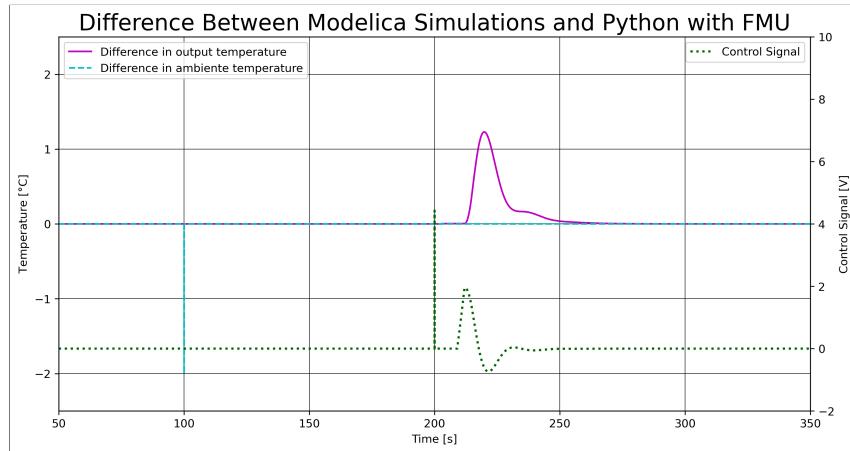


Figure 8.3: Resulting plot comparison of the difference between PI in Python and OpenModelica

# 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 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. Plots from both simulations is presented. The complete source code can be found in the GitHub repository link: ([Link to repository](#)).

---

The implementation described in this chapter is based on the source code for a constrained nonlinear model predictive controller 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: ([Link to source code](#)).

## 9.1 Implementation in Python

From the code presented in Code 9.1 one can observe that the start values is set with the use of Python variable type lists for both which values to make available in the simulation displayed in Line 2, and also setting the initial values in Line 5. In the previous example featuring the PI controller discussed in Chapter 8, these values were stored in individual variables utilizing their respective setReal() functions.

```

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 # 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](#)

The implemented source code, as presented in Code 9.2, resides within the simulation loop. Examining the source code reveals the utilization of both the current time step in the simulation and the sampling time within the fmu.doStep() function, facilitating the simulation of the FMU for the subsequent step. The values employed are identical to those used in simulating the first-principals model. In Line 5, inputs are recorded from the FMU, and in Line 8 the new optimized control signal horizon is calculated before the first control signal is applied to the FMU in Line 17.

```

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 res = minimize(calculate_mpc_objective,
9                 u_guess,
10                method='SLSQP',
11                constraints=[ineq_cons],
12                options={'ftol': 1e-9, 'disp': False},
13                bounds=bounds_u)
14
15 # SET FMU optimized control signal
16 fmu.setReal([fmu_ControlSignal_Input], [u_opt_k])

```

Code 9.2: Simulate FMU - [Link to source code](#)

The source code provided in Code 9.3 illustrates the sequence for applying changes in the temperature setpoint to both the first-principles model and the FMU.

```

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

```

## 9.2 Results

In this chapter, a comparison is presented between simulations conducted MPC together with a FMU and a first-principles model, both simulated at the same time. Table 9.1 displays statistical properties for both the control signal and output temperature in both simulations, along with the absolute value of the difference. The results presented in Table 9.1 show no significant differences in the results.

Table 9.1: Statistical values for both simulations

Variable	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

In Figure 9.1 one can observe both the changes in the temperature setpoint and the resulting process outputs from both the first-principles model and the FMU simulation. The figure illustrates that the bounds were not violated at any point and the performance of both models demonstrates similar characteristics. This is further confirmed in the plot presented in Figure ?? that displays the difference in control signal and temperature output for both simulations, where the maximum difference is below 0.05 [°C].

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

From the plot presented in Figure 9.4, one can observe the rate of change in the control signal and the corresponding bounds. The figure indicates that the bounds were not violated.

Draft

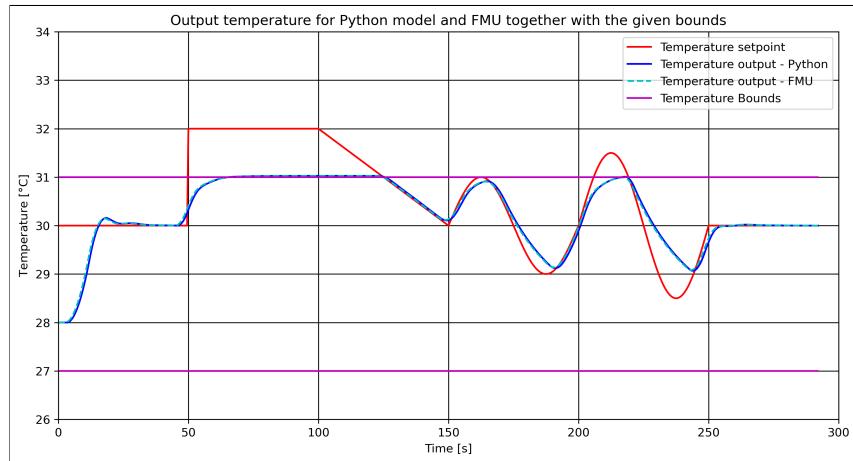


Figure 9.1: Resulting plot comparison of the output temperature difference between FMU and first-principal model

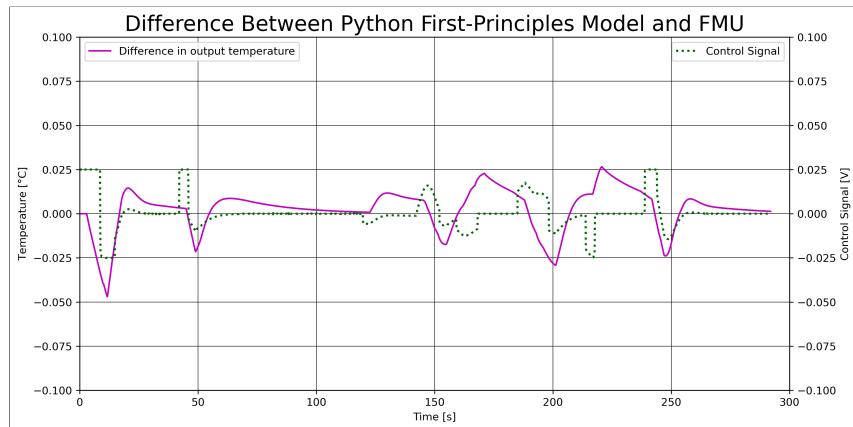


Figure 9.2: Resulting plot of difference between Python first-principles and FMU simulation

For this implementation, the FMU of the air heater produces results that exhibit similar patterns to the first-principles model implemented in the source code.

Draft

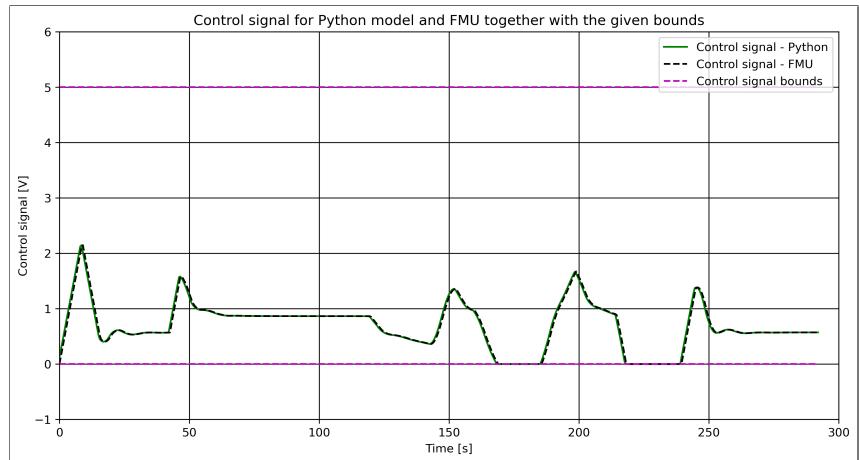


Figure 9.3: Resulting plot of the control signal with bounds

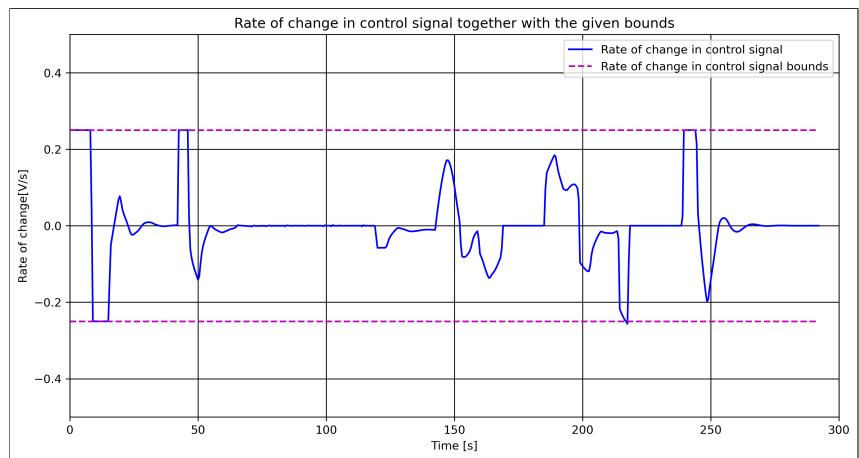


Figure 9.4: Resulting plot of rate of change in control signal

## **10 Conclusion**

Draft

# 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] B. Lie, *Modelling of dynamic systems*, Accessed: 2018-12-06, 2019.
- [4] P. Carreira, *Foundations of multi-paradigm modelling for cyber-physical systems*, eng, Cham, 2020.
- [5] ‘The modelica association.’ (Jan. 1), [Online]. Available: <https://modelica.org/>.
- [6] ‘Modelica language.’ (Jan. 1), [Online]. Available: <https://modelica.org/modelicalanguage.html>.
- [7] M. Association, *Modelica® – a unified object-oriented language for systems modeling*, English, version Version 3.6, Mar. 2023, 365 pp.
- [8] ‘Modelica external objects.’ (), [Online]. Available: <https://doc.modelica.org/om/ModelicaReference.Classes.ExternalObject.html>.
- [9] *Robot operating system (ros) : The complete reference. (volume 5)*, eng, Cham, 2021.
- [10] ‘Functional mock-up interface specification 3.0.’ (May 2022), [Online]. Available: <https://fmi-standard.org/docs/3.0/>.
- [11] Y. Xi, *Predictive control : Fundamentals and developments*, eng, Hoboken, NJ, 2019.
- [12] 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.
- [13] A. Bemporadl. ‘Mpc from basics to learning-based design,’ Youtube. (2022), [Online]. Available: <https://www.youtube.com/watch?v=CNwV5GbTEGM&t=754s>.

Draft

- [14] *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.
- [15] F. A. Haugen, *Modeling, Simulation and Control*. Porsgrunn, 2023.
- [16] *Process dynamics and control*, eng, Hoboken, N.J, 2011.
- [17] M. AB, *Jmodelica.org user guide*, English, version Version 2.2, Modelon AB, Mar. 2018, 164 pp.
- [18] 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>.
- [19] C. B. Corbonell, ‘Model-based predictive control using modelica and open source components,’ M.S. thesis, Norwegian University of Science and Technology, 2010.
- [20] 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>.
- [21] 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.
- [22] ‘Embedded mpc for next-gen controls.’ (), [Online]. Available: <https://www.odys.it/>.
- [23] ‘Fmpy.’ (), [Online]. Available: <https://github.com/CATIA-Systems/FMPy>.
- [24] S. G. Johnson, *The NLOpt nonlinear-optimization package*, <https://github.com/stevengj/nlopt>, 2007.
- [25] ‘Air heater.’ (), [Online]. Available: [https://www.halvorsen.blog/documents/hardware/air\\_heater.php](https://www.halvorsen.blog/documents/hardware/air_heater.php).

- [26] *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.
- [27] D. E. Seborg, T. F. Edgar and D. A. Mellichamp, *Process Dynamics and Control*, 2nd ed. John Wiley & Sons, 2004.

Draft

## Appendix A

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

*The signed task description as the basis for the thesis*

Draft

# 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



CARL MAGNUS BØE

Student (write clearly in all capitalized letters):

2023.01.28 - Carl Magnus Bøe

Student (date and signature):

Draft