University of South-Eastern Norway

www.usn.no

FMH606 Master's Thesis 2023

Industrial IT and Automation

Advanced Control Implementation with Modelica

Draft

<optional figure>

fig/USN_logo_en.pdf

Carl Magnus Bøe

Faculty of Technology, Natural Sciences and Maritime Sciences

Campus Porsgrunn

University of South-Eastern Norway

www.usn.no

Course: FMH606 Master's Thesis 2023

Title: *Advanced Control Implementation with Modelica*

Pages: *100*

Keywords: *Modelica external objects, Optimization, MPC, model predictive control, Modelica, ...*

Student: *Carl Magnus Bøe*

Supervisor: *Finn Aakre Haugen*

External partner: *Anushka Perera - Yara Porsgrunn*

Draft

Summary:

Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Ut purus elit, vestibulum ut, placerat ac, adipiscing vitae, felis. Curabitur dictum gravida mauris. Nam arcu libero, nonummy eget, consectetuer id, vulputate a, magna. Donec vehicula augue eu neque. Pellentesque habitant morbi tristique senectus et netus et malesuada fames ac turpis egestas. Mauris ut leo. Cras viverra metus rhoncus sem. Nulla et lectus vestibulum urna fringilla ultrices. Phasellus eu tellus sit amet tortor gravida placerat. Integer sapien est, iaculis in, pretium quis, viverra ac, nunc. Praesent eget sem vel leo ultrices bibendum. Aenean faucibus. Morbi dolor nulla, malesuada eu, pulvinar at, mollis ac, nulla. Curabitur auctor semper nulla. Donec varius orci eget risus. Duis nibh mi, congue eu, accumsan eleifend, sagittis quis, diam. Duis eget orci sit amet orci dignissim rutrum.

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 a result of a cooperation between USN campus Porsgrunn and Yara Porsgrunn. Yara International was established as Norsk Hydro in 1905 and has its headquarters in Oslo. Yara International is listed on the Oslo stock exchange and has more than 17 000 employees with operations in over 60 countries. Yara International is the leading crop nutrition company in the world with a focus on the global challenges with regards to CO_2 and greenhouse gas emissions. Yara focuses not only on reducing their own emissions in production but also helping the green shift by supporting emission reductions all the way into the fields where the crop nutrition is applied [1].

Yara Porsgrunn is one of several plants within Yara International and is located at Herøya in Porsgrunn Norway. Yara Porsgrunn manufactures products based on nitrogen and has the highest production capacity of nitrophosphate fertilizer in Europe. In addition, the plant also produces a wide range of chemicals and gases used in industry applications. [2].

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
Dietmar Winkler	USN - Campus Porsgrunn
Finn Aakre Haugen	USN - Campus Porsgrunn
Neste person	Neste org

Porsgrunn, 24th September 2023

Carl Magnus Bøe

Contents

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
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

Yara's current workflow involves generating models in Modelica, followed by exporting these models as functional mock-up units (FMU) to MATLAB. Within MATLAB, the MPC block in the Simulink package is utilized to compute the control signal. To streamline this process, Yara aims to consolidate the entire system within Modelica. One of the options is integrating an MPC block directly into the Modelica environment.

1.3 Method and Structure

The thesis begins by giving background information about concepts like modeling, Modelica, controllers, and model predictive control. Then, it evaluates different models used in the thesis' implementations.

Next, there is a chapter that shows numerous ways advanced control methods are applied to Modelica. Some non-essential parts of source code for things like plotting are left out in the main text to focus on the control implementation. The complete source code for all the source code presented in this thesis is available at the GitHub repository (Link to repository).

Each part of source code presented in this thesis includes a direct link in the figure caption, directing you to the specific code in the GitHub repository. Similar linking is also provided in 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

2 Background Theory

In this chapter, the software applications employed 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

By creating a model of a system, one can run tests on the system to learn about its properties and behavior. The reason for making models and simulating experiments before or sometimes instead of doing physical tests is because conducting such tests on a physical model has some disadvantages [3]:

- High cost
- Dangerous
- System not yet built
- Slow changes
- Complex and time consuming to update system

Modelling and simulation of systems can also lead to some errors mainly based on the human part of the analysis. The main items to be aware of is not to fall in love with the model, and by that forgetting that it is only a model that may not take into account all the real-world parameters since the model could be constructed without taking care of all real-world conditions. Another aspect to take into account is that the accuracy

and usability of the results is highly based on the complexity of the model used for the experiments[3].

A system based on mathematical modelling displays the relationship between different variables and how they change as part of a complete system. One example is Hook's law that is mathematically represented by $F = -kx$.

Various methods exist for representing models, but in this report the focus is on the first-principal mathematical models. The main types of equations used for mathematical models are[3]:

Differential equation with time derivative

$$\frac{du}{dt} = a \cdot u + 10$$

Algebraic equation does not include differentiated variables

$$Z^2 = x^2 + Y^2$$

Partial differential equations contain derivatives with respect to variables other than time

$$\frac{\partial a}{\partial t} = \frac{\partial^2 a}{\partial z^2}$$

Difference equations can display relations between variables at different point in time

$$u(t+3 = 4u(t) + 42$$

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 [4]. 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 used to present the steady state where the output of the system is stable if the input signals are unchanged.

One example of the differences between static and dynamic is displayed in Figure 2.1 where one can observe both a resistor and a capacitor being exited with a step change in current at $t = 4[s]$ Since the resistor follows Ohm law given as $U = R \cdot I$ the response to the step change in current is only given as a linear function of the value compared to the step change, 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 [3]. 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 [5].

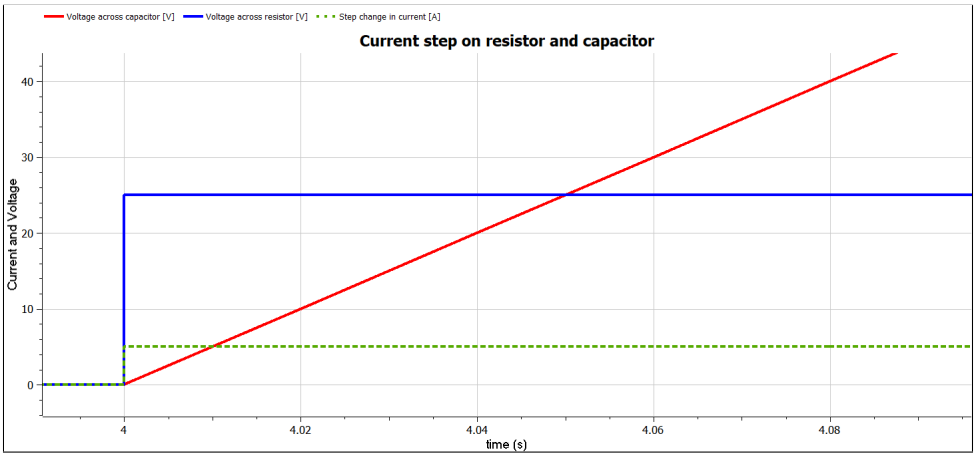

Figure 2.1: Plot of current step response in resistor and capacitor.

A causal system is defined as when a system output is independent of the future inputs, whereas an acausal system is dependent on input values for any instance in time [5].

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, while the discrete system only changes values for some given points in time. One practical example of this could 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 [3]. 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 some 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 some discontinuity at a specific point in time [3].

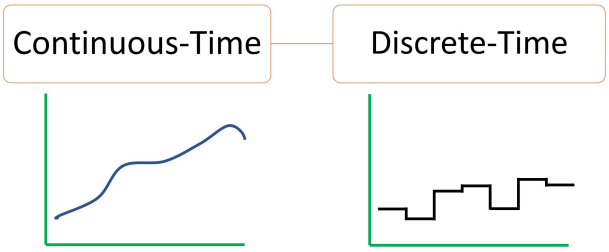

Figure 2.2: Difference between continuous and discrete time models

2.2 Modelica

In 1996, a collaboration between application experts and computer scientists initiated a project [3] called the "Simulation in Europe Basic Research Working Group". Their objective was to develop a language that can be applied for physical modelling. Some of the key goals were to construct the language so that it would include the advantages from object-oriented programming and by this making exchange of libraries and models relatively easy. Despite the existence of several other alternatives in the market at the time, all of the options were based on proprietary and more specific applications [6].

The Modelica Association was founded as an independent non-profit organization in 2000 to promote development of the Modelica language and the standard libraries [3]. The Modelica Association also oversees the five open-source standards displayed in Figure 2.3 [7].

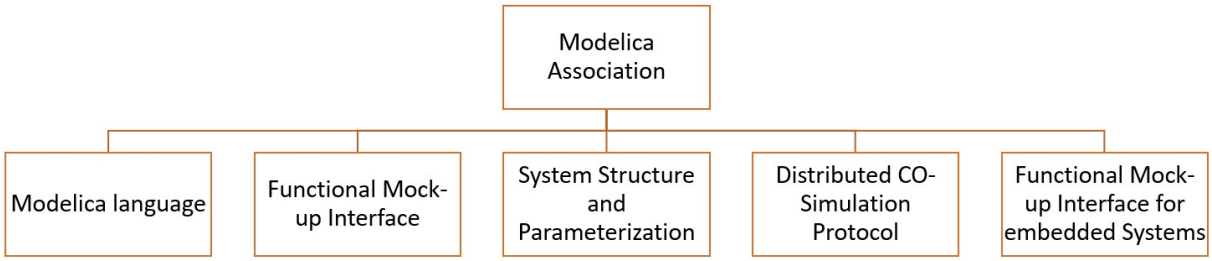

Figure 2.3: Standards governed by the Modelica Association [7].

2.2.1 Modelica Language

The Modelica language is the basis for the open-source modeling and simulation environment named Open Modelica and a range of different other commercially available environments. Open Modelica is the prominent open-source alternative, in addition to several commercial alternatives as displayed in Table 2.1.

Table 2.1: Commercial available software based on Modelica[8]

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™	

The language is based on modelling of first-principle equations that do not require any pre-processing of the equations before implementation into the code, since one can simply add equations straight from e.g., a textbook into the text editor and then be able to simulate the system.

The Modelica language is an object-oriented language with focus on the possibility to reuse models for both text based and block-based modelling [8]. The way Modelica differs from other procedural object-oriented programming languages like C is that Modelica uses the object-oriented concept as a structuring method to manage large system descriptions. Modelica is a declarative programming language where dynamic models are expressed through equations instead of assignment statements as for procedural languages where one must declare a stepwise algorithm onto how to reach the desired goal. Same as in the way standard object-oriented languages Modelica also has classes as templates for objects, inheritance of variables, equations, and functions between classes [3].

One of the unique features when modelling systems based on Modelica is that one can set up combined systems based on elements from several different domains. This can be a mix of electrical, mechanical, hydraulic, and process-oriented components all in one simulation model.

Modelica has a large set of different models in the Modelica library. The open-source library has more than 1600 components and 1350 distinct functions. The different com-

mercial versions of Modelica have an even greater number of libraries with different components and functions[8].

Modelica is widely used in the automotive industry to perform energy analysis and used by ABB and Siemens for modelling of power plants[8].

Execution of Modelica Models

The flow of how Modelica models are implemented and executed is displayed in Figure 2.4.

Modelica Source Code: The first step in the process is passing the Modelica model into the translator.

Translator: The translator parses the code from Modelica source code into machine language. In the processes through the translator the code is analyzed, type checked, inheritance and expansion of classes and conversions of equations. The output of the translator is a flat model code. The flat model represents a flat set of equations, functions, and variables. There is no longer any trace of the object-oriented structure.

Analyzer: The equations are then sorted based on data-flow dependencies between the different equations. DAEs are not only sorted, but also transformed in block lower triangular form to simplify the equations structure for a more effective numerical solving.

Optimizer: The optimizer module performs algebraic simplification by eliminating most equations, and just keeping the minimal set needed to produce a numerical solution.

Code Generator: Generates the C code.

C Compiler: Takes in C code and links it with a numeric equation solver that can solve the now reduced equations system.

Simulation: Initial values are approximated or taken from values specified in the source code.

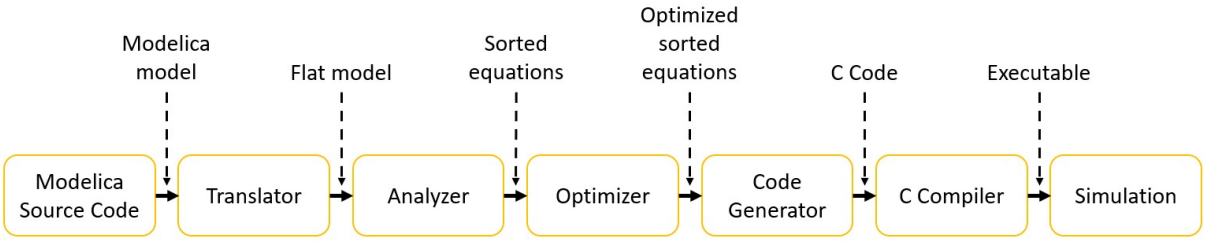

Figure 2.4: Flow displaying how Modelica models is translated and executed [3].

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 used for tasks other than solving equations, like 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 Functional Mock-Up Interface

The functional mock-up interface is used 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 .ZIP where the file includes a .XML file that holds the description of the parameters, derivatives, variables, and model structure. The FMU also contains equations and shared library in C source code. [9].

FMI defines three distinct types of interfaces.

Model Exchange

The model exchange is an interface for solving ordinary differential equations solely based 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 some 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].

The algorithms take care of the overall advance in simulation time, exchange of input and output from the individual FMUs, handle events and triggering of clocks.

Scheduled Execution

With the scheduled execution interface individual parts of the models is exposed so that the scheduler for the importer can gain control with regards to execution of the individual parts giving the interface possibility to perform concurrent computation of the individual parts of the exposed models. Concurrent computation is currently not supported by the FMI standard but can be used for computations within the respectively FMUs [10].

2.2.3 Functional Mock-up Interface for Embedded Systems

eFMI standard is a result of the EMPHYSIS project running from 2017 to 2021 where the objective was to develop a standard to reduce the cost and time spent to enable exchange of physics-based models and simulations with software development environments based on embedded systems like electronic control units and micro controllers. With the development of the eFMI standard one can extend the FMI standard onto embedded units and by this prototyping the whole chain from physical model to code developed for the embedded device [11].

FMI is made to run on a few standard platforms like Windows and Linux where resource allocation and run-time performance is taken care of by the operating system. On embedded devices these constraints must be taken into account together with support for different compilers. While the FMI standard is based on C code with a fixed interface for the standard platforms, embedded devices need to support numerous compilers. This is enabled by eFMI where one can include several implementations of C code for different interfaces stored in a manifest file into the production code [12].

2.2.4 System Structure and Parameterization

The SSP is a tool to structure and define the signal flow including unit checking between a network of several FMUs. The structure can be based on hierarchies of sub-systems to generate a complete system topology [13].

The development of SSP was initiated based on the FMI standard but can now be applied for a wide set of different interfaces. The motivation for the SSP standard was to create a network of one or more FMUs with the possibility to separate, map and change parameters from different FMUs, and to be able to store a structured network of FMUs [14].

The main intent of developing the SSP was to set up a standard schema where any tool specific data can be stored to simplify exchange of simulation systems. The exchange is designed to be as simple as possible, but still conserving the essential information.

2.2.5 Distributed CO-Simulation Protocol

The goal of DCP is to deliver an independent communication protocol to help reduce the required effort to perform integration between different simulation environments and external real-time systems [15] and/or non-real time system into one co-simulation[16].

In a market with several suppliers using their proprietary interfaces there is much to gain moving from a vertical to a horizontal approach as DCP represents, by reducing the effort to connect subsystems together[17].

The DCP protocol was designed to be compatible with FMI. Whereas FMI represents the API, DCP represents a communication protocol that is not part of the FMI. The reason to develop DCP was with the goal to reduce the required effort when coupling and integrating real-time systems into simulation environments. [16]. DCP specifies a model for the data, a finite state machine, protocols for data units and communication. The protocol is applied in several different domains, such as architecture, maritime, aerospace, automotive, with applications such as: [15].

- Distributed simulations
- Co-Simulations
- Process automation
- Hardware in-the-loop
- Software in-the-loop
- Model in-the-loop

DCP is set up with a master/slave approach that gives a flexibility to stop any slave instances and switch between a real hardware setup, and a model [16]. The DCP master configures and organizes the different DCP slaves, so that the given scenario can be realized as specified. In the scenario the different DCP slaves represent individual subsystem [18]. The communication protocol defined by DCP is located on the application layer and relies on the use of standardized transport protocols such as Bluetooth, USB, TCP, UDP or CAN [15]. With this approach the DCP can still support future transport protocols [16].

2.3 Controllers

In automation systems the feedback/closed-loop controller calculates the error by comparing the measured output value with a given reference value. This relationship can be given as $e = r - y$ where e represents the system error, and r as the reference set point with y as the measured system variable.

Controllers based on this principle can be divided into three main types [19].

Classical Controller

There exist two main types of classical controllers with the first one being the PID controller and the second type being the on/off (bang-bang) controller. These types of controllers are considered to be reactive in the sense that they only consider past and current system state when performing control actions [19].

Predictive Controller

Uses a model of the given system to predict future behavior of the system and thus being able to predict future deviations from the reference [19]. This allows this controller principle to work in a pre-emptive manner taking actions before the error between the measured output and the reference value/set-point has occurred.

Repetitive Controller

Analyses the systems behavior in the previous cycle and based on that information calculates the trajectory for the next cycle [19].

2.3.1 Predictive Control

Predictive control was developed as an advanced control algorithm for complex industrial processes in the mid-1970s. The main methodological principles are to predict future system dynamics based on a given series of control actions using a model of the system with given constraints, and to optimize the control sequence. For industrial control methods the industry mainly had its focus on closed loop control principles such as the well-known PID controller. The PID controller is suitable for both linear and non-linear processes making it the universal choice for most applications where the requirement of the controller is SISO (Single-input single-output). However, when the process control requires to take into account MIMO (Multiple-input multiple-output) in addition to extending from regulation into optimization based on system constraints; the single loop PID falls short due to the lack of knowledge of the process dynamics [20].

Some of the earliest algorithms used for predictive control in industrial applications, developed back in the 1970s, were based on a process step response obtained from existing process data. Using this model, the controller could be designed without requiring further identification of the process dynamics. Since the academic field of control engineering was limited during this early stage, there was a heavy reliance on domain knowledge of the actual process being controlled to facilitate the design of the controller. In the 1980s, the second stage of algorithm development was based on the adaptive control field. This led to the creation of algorithms such as Model Predictive Control (MPC), which were more familiar to the control community. However, these algorithms still posed challenges in analysis due to the absence of analytical expressions for optimal solutions.

With significant shifts in research ideas during the 1990s, predictive control theory became the mainstream in predictive control research. Despite this, a gap between theoretical development and practical application persisted. Research on MPC experienced rapid development, yielding significant insights into the relationship between MPC and optimal control [20].

There exists a substantial number of different predictive control algorithms, but they all share some common characteristics in how they work, based on three main principles [20]:

1.Prediction Model

The prediction model is the basis of optimization control where the model is used to predict the future response for outputs and possible internal states based on historical information and assumed future control signals. A prediction model can be both linear and non-linear e.g., based on transfer functions, empirical data, state space equations or impulse functions. Since the prediction model provides the basis for comparing the quality of different control policies, it is a prerequisite for optimization control [20].

2.Rolling Optimization

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

Predictive control uses on-line finite horizon optimization also known as rolling optimization appose to traditional optimal control that is based on off-line infinite horizon calculation [20].

A general basic comparison between optimal control, predictive control and PID is presented in Table 2.2.

3.Feedback Correction

In most practical applications there will exist some uncertainties and discrepancies between the model and the physical process along with the potential for some process disturbances to occur. To mitigate these elements the predictive control must have some type of closed-loop mechanism.

Table 2.2: Comparison between optimal control, PID and predictive control [20]

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

The feedback correction can be implemented as a function that waits until real-time values from the physical system are recorded. It then utilizes 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 [20].

2.3.2 Application of Predictive Control in Industrial Processes

Predictive control did not originate from within the field of control theory; rather, it emerged as a response to the demands posed by complex industrial processes. It also presented one of the few approaches capable of addressing multivariate constrained optimization challenges. Predictive control was implemented with remarkable success due to its capability to manage complex optimization problems with constraints. Thanks to advancements in commercially available predictive control software, this approach has found widespread adoption in the process industry, being implemented in over a thousand industrial plants [20].

2.3.3 Applications of Predictive Control Outside the Industry

Initially driven by demands within the process industry, predictive control has more recently expanded its scope into other fields. Examples of dynamic fields benefiting from predictive control's optimization include path planning for robots and production as well as resource planning in a Manufacturing Execution System (MES). While offline solutions are commonly used for these instances, predictive control offers superior performance in environments characterized by uncertainty."

General control problems of this nature cannot be effectively addressed solely through dynamic mathematical models and existing control theory. Consequently, various optimization algorithms need to be explored based on the specific problem's characteristics. By performing a study of the system one can find a description and with that find the optimal solution. The optimal solution will only be valid over time if the model and environment is unchanged. The solution will then not be able to take into account if there are changes in production demand or conditions, issues in supply chain or other elements affecting the production planning. In addition, the model of the system must have a high accuracy to be able to perform the optimization, in addition it might be demanding to compute. Through the adoption of predictive control, the system model's accuracy requirements are mitigated due to the introduction of a feedback mechanism. Additionally, the utilization of a finite horizon leads to a reduction in computational load[20].

2.4 Model Predictive Control

MPC uses a model of a given system enabling the MPC to observe the future system response if one applies a sequence of inputs for future system time-steps. The optimization algorithm in the MPC will make sure the preferred sequence is chosen based on a performance index.

The basic function of MPC is visualized 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 one wishes to control, together with an optimizer. With the use of the model one can 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 used in the optimization process in the next time-step [19].

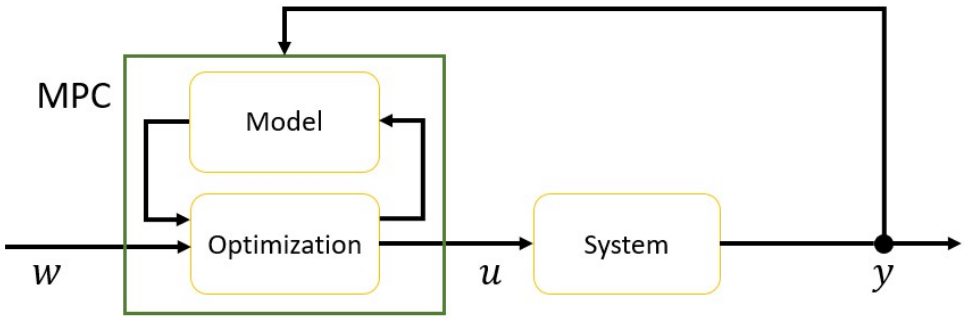

Figure 2.5: Basic block diagram for MPC. [19].

In addition to calculating the optimal control signal the optimizer also takes into account any given system constraints. These are some of the unique features that is possible to achieve with the use of MPC. 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.

An analogy to the MPC has been presented by A.Bemporad in [21] 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 [21].

2.4.1 Types of Models Used for MPC

Various forms of models are utilized 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 [22].

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

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

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

The model can have more than one control input (u), state (x) and output (y). In such a case these properties can be presented as vectors [22].

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 some random process noise given by v and some measurement noise in w. The two noise terms are uncorrelated with zero mean and with some given variance.

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

The matrices given by (A_d), (B_d), (C_d) and (D_d) 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 cases like these, these properties can be represented as vectors and can also be presented in a simplified form as presented in Equation 2.3

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

Polynomial Models

The polynomial / input-output models are identified as models where the current output is dependent on previous outputs and/or past inputs from a given process. These types of models do not include any states (x_k) as were introduced for the state space model.

Nonlinear polynomial models can be expressed as presented in Equation 2.4.

$$y_k = h(y_{k-1},\, y_{k-2},\, \cdots,\, y_{k-m_y},\, u_{k-1},\, u_{k-2},\, \cdots,\, u_{k-m_u}) \tag{2.4}$$

Where (h) represents any nonlinear function of the model output given by (y_k) with (m_y) number of previous outputs, and (u_k) with (m_u) number of previous control inputs.

Linear polynomial models can have different forms, where ARX (auto regressive exogenous) is one example presented in Equation 2.5. The current output of an ARX model is also only dependent on previous outputs and inputs.

$$\begin{aligned} y_k = &-A_1 \cdot y_{k-1} - A_2 \cdot y_{k-2} - \cdots - A_{m_y} \cdot y_{k-m_y} \\ &+B_1 \cdot u_{k-1} + B_2 \cdot u_{k-2} + \cdots + B_{m_u} \cdot u_{k-m_u} \end{aligned} \tag{2.5}$$

Where (A) and (B) are the past output and input coefficients respectively [22].

Continuous Time Systems

First principle / mechanistic models generated from conservation laws often lead to non-linear 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 2.6.

$$\begin{aligned} \frac{dx}{dt} &= f(x,u,t) \quad \leftarrow State\ equation \\ y &= g(x,u,t) \quad \leftarrow Measurement\ equation \end{aligned} \tag{2.6}$$

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 2.7.

$$\begin{aligned} \frac{dx}{dt} &= A_c \cdot x(t) + B_c \cdot u(t) \quad \leftarrow State\ equation \\ y(t) &= C_c \cdot x(t) + D_c \cdot u(t) \quad \leftarrow Measurement\ equation \end{aligned} \tag{2.7}$$

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 [22].

Transfer Function Model

Models can also be expressed as transfer functions with an example converted to the frequency domain as presented in Equation 2.8.

$$H(s) = \frac{y(s)}{u(s)} \tag{2.8}$$

Where the inputs $u(s)$ can be transferred to outputs $y(s)$ with the help of the transfer function given as $H(s)$ [22].

2.4.2 Linearization of Nonlinear Models

Nonlinear models can be linearized around some operating point. In most cases the model will then have a relatively high accuracy around the point of linearization, but the quality of the linearized model will gradually decrease the further one moves away from the chosen operating point used in the linearization process [22].

2.4.3 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 regarding 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 examples where one on 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 done if the outputs have a varying level of importance, and 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 some 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.

Generally, the input weights are presented in 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.

2.4.4 Optimization in MPC

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

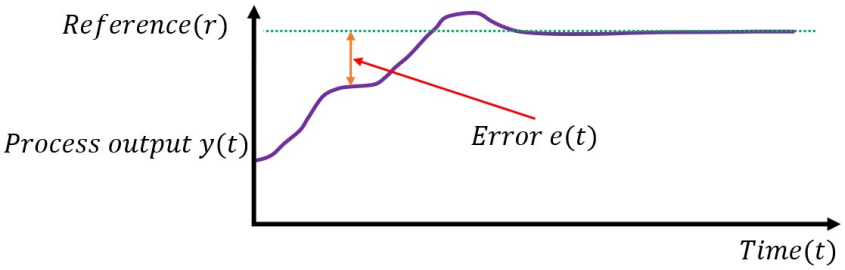

Figure 2.6: Optimization of reference tracking. [22].

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 [22].

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 2.9.

$$e = (y - r)^2 \tag{2.9}$$

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 2.10 [22].

$$e_{Total} = \sum_{k=1}^{P} (y(k) - r)^2 \tag{2.10}$$

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

how to form the optimization objective function is displayed in Equation 2.11.

$$\min_{U} \quad J(U) \tag{2.11}$$

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 $J(U)$ [23].

The optimized control matrix displayed in Equation 2.12 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 [23].

$$U = [u_k, u_{k+1}, \ldots, u_{N-1}, u_N] \tag{2.12}$$

From the matrix presented in Equation 2.13 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 [23].

$$U = \begin{bmatrix} u(1)k & u(1)_{k+1} & \cdots & u(1)_{k+(N-1)} & u(1)_{k+N} \\ u(2)k & u(2)_{k+1} & \cdots & u(2)_{k+(N-1)} & u(2)_{k+N} \\ \vdots & \vdots & \cdots & \vdots & \vdots \\ u(r)k & u(r)_{k+1} & \cdots & u(r)_{k+(N-1)} & u(r)_{k+N} \end{bmatrix} \tag{2.13}$$

Based on a given objective function one can set up a final objective function for an optimal setpoint tracking problem as given in Equation 2.14 [22] :

$$\min_{u} \quad J = \frac{1}{2} \cdot \sum_{k=1}^{N} (e_k^T \cdot Q_k \cdot e_k + u_{k-1}^T \cdot P_{k-1} \cdot u_{k-1}) \tag{2.14}$$

Where the variables for Equation 2.14 are described in Table 2.3.

Table 2.3: Variable description for objective function presented in Equation 2.14 [22].

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 with regards to the rate of change in the control input given as Δu_k as displayed Equation 2.15.

$$\min_{\Delta u} \quad J = \sum_{k=1}^{N} (e_k^T \cdot Q_k \cdot e_k + \Delta u_{k-1}^T \cdot P_{k-1} \cdot \Delta u_{k-1}) \tag{2.15}$$

This can be a relevant formulation for when the system has actuators and control valves that have some limitations due to their dynamic performance [22].

The optimization problem can also have some constraints that must be satisfied, if possible, when performing the optimization. Example of different constraints can be:

- Physical
  - Actuator limits
  - Pump capacity limit
  - Limitations to rate of change in pump
- Safety
  - Temperature limit
  - Pressure limit
- Environmental
  - Reduce energy usage
  - Reduce waste

- Performance
  - Accuracy of setpoint tracking
  - Speed of setpoint tracking

When applying constraints to an optimization problem one will in most cases limit the solution domain to a feasibility region for which the solution can exist. The optimizations problem can in some cases become infeasible to solve due to constraints [22].

The constraints can be divided into hard, and soft constraints. Examples of hard constraints are the range of valve opening being maximum 100 %, maximum pump capacity being limited to some maximum flow, and total volume of a tank. These hard constraints are limited by their general fixed physical properties. 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 [22].

As an example to the different constraints one can evaluate a general function presented in Equation 2.16.

$$\min_{u} f(u) \tag{2.16}$$

The function can be subjected to m number of equality constraints given in Equation 2.17.

$$\begin{aligned} h_i(u) &= 0 \\ i &= (1,2,3,\cdots m) \end{aligned} \tag{2.17}$$

and r number of inequality constraints as presented in Equation 2.18.

$$\begin{aligned} g_j(u) &\leq 0 \\ j &= (1,2,3,\cdots r) \end{aligned} \tag{2.18}$$

Constraints can also be specified as bounds, representing a range of permissible values. This can be expressed as displayed in Equation 2.19 [22].

$$u_L \leq u \leq u_U \tag{2.19}$$

Based on the previously presented 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.

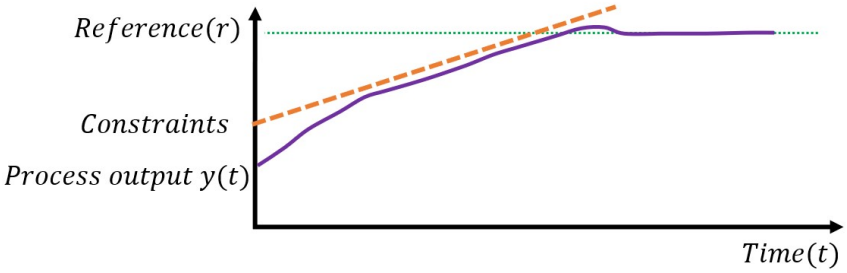

Figure 2.7: Optimization of reference tracking with constraints. [22].

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 some linear actuator that limits the rate of change in the system compared to the reference tracking presented in Figure 2.6.

Categories of Optimization Problems

Three examples of different optimizations problems can be:

- Linear programming problem (LP)
  - Linear objective function
  - Linear constraints
- Quadratic optimization problem (QP)
  - Quadratic objective function
  - Linear constraints
  - Often used 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 [22].

Analytical Method

To solve the optimization problem with an analytical approach one can consider some functions, as exemplified by the function f in Figure 2.8 that has some local minimums and local maximums. By calculating the first derivative of the function and solving for $f'(x) = 0$ one will detect all the points for where there exists either a local minimum or local maximum. To further analyze the results to find out if the given first derivatives is a maximum or minimum one can calculate the second derivative and observe if the result either is greater than or smaller than zero. With a result for the second derivative greater than zero the point is a minimum, while a negative value results in a local maximum [23].

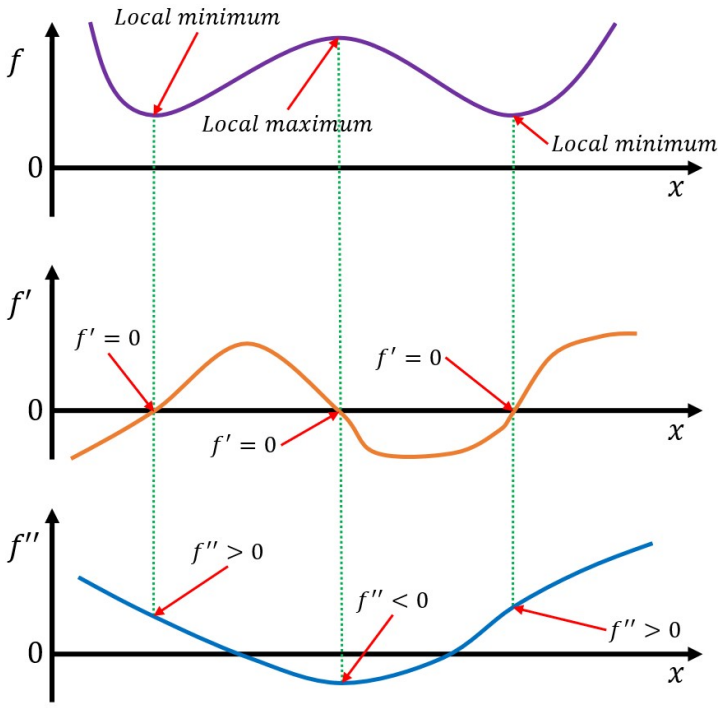

Figure 2.8: Analysis of objective function [23].

Iterative Method

The iterative method is initialized with a guess of the value to be optimized. In the example presented in Figure 2.9 the value of x is optimized to minimize the functional value of $f(x)$. With the use of the iterative method the initial guess of the value is crucial since one could end up at a local minimum or even a local maximum far away from the actual functional minimum or maximum.

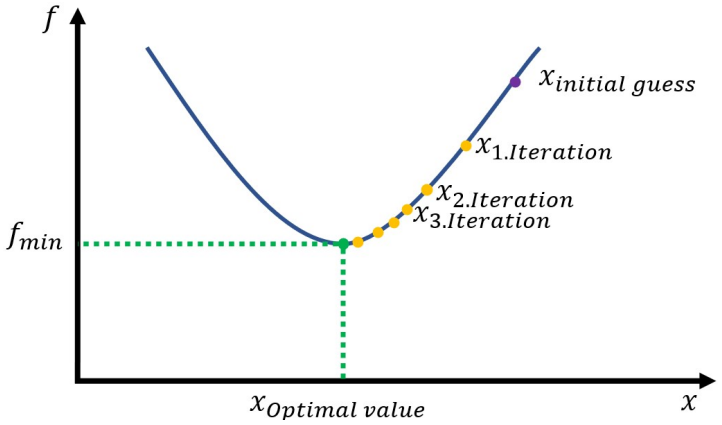

Figure 2.9: Iterative optimization [23].

The steepest decent method is also based on an iterative approach and does as the name indicate have the objective to follow the path of where the reduction of the optimization variable has the largest change possible. One can express a scalar optimization problem of x as displayed in Equation 2.20.

$$x_{k+1} = x_k + \Delta x_k \tag{2.20}$$

With K given as the factor that defines the size of the step and $f'(x)$ as the resulting gradient of the function. The reduction step can then be formulated as shown in Equation 2.21.

$$\Delta x_k = -K f'(x_k) \tag{2.21}$$

One of the limitations of this method is when Δx_k becomes too large, the method can overshoot the actual minimum point leading to an oscillation around the minimum [23].

Newton search method is another iterative method that utilizes the property that when the gradient of the function is zero, one has found either a maximum or a minimum. This can be expressed as displayed in Equation 2.22.

$$f'(x_{possibleMin}) = 0 \tag{2.22}$$

Since this is true for both maximum and minimum one also has to evaluate the double derivative to find out if the value is a maximum or minimum. This can be expressed as displayed in Equation 2.23.

$$f''(x_{possibleMin}) > 0 \tag{2.23}$$

Grid/Brute Force Method

The grid method is relatively easy to implement for most types of optimization problems. The method finds the global minimum and requires no advanced knowledge about optimization theory, as it relies on using a computer to perform all the demanding work.

The method for solving the optimization problem based on the grid method is to perform calculations of the objective function for all combinations of the given optimization variables. This can be performed with nested for loops, where the outer most for loop runs through values for e.g., the optimization parameter $x1$ and a second for loop inside the outer most for loop that calculates the values for the optimization parameter $x2$, and so on.

2.4.5 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 2.10.

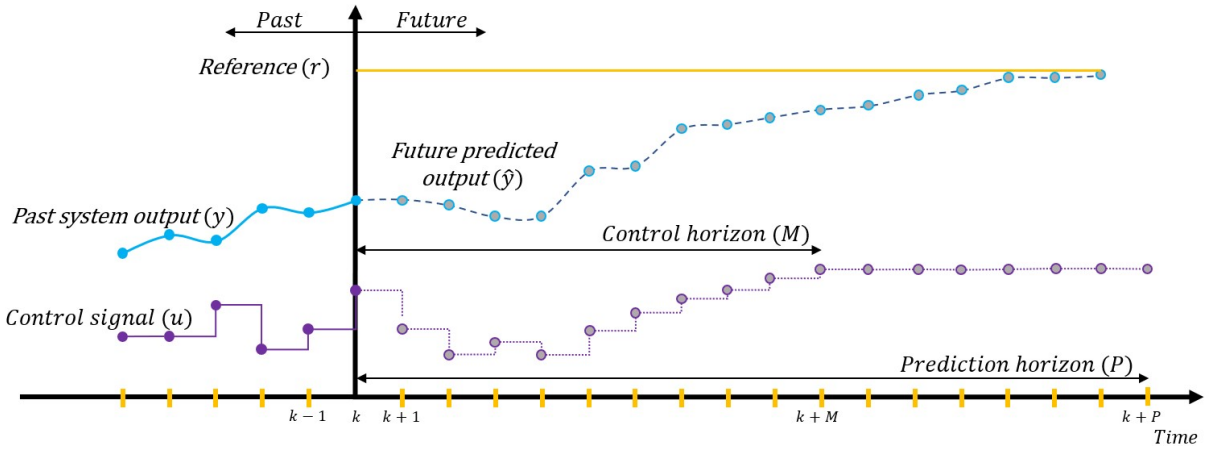

Figure 2.10: Mode of action for MPC with receding horizon [24].

From the Figure 2.10 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 [24].

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 this 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 [24]. The sliding horizon can also introduce a feed-forward function by including the future disturbances and changes in setpoint when solving the optimization problem [22].

3 Implemented Models

In this chapter there is a presentation of the models implemented in Modelica.

3.1 Model of Air Heater Based on First-Principle

USN has built several identical air heater systems that are available in the campus lab station. The air heaters have been used for educational purposes for at least 15 years and has been utilized for a wide range of different applications throughout the time. One of the applications that is highly relevant for this work and the main reason for choosing 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 in the textbook [23] and at the web site techteach.no. With the use of Finns Python script where the MPC for the air heater is implemented from scratch, one can validate the model produced in Modelica against what Finn as already implemented.

A picture of the air heater is presented in Figure 3.1.

The mathematical model of the air heater can be expressed as displayed in Equation 3.1.

$$t_{const} \cdot \frac{dT}{dt} = (T_{amb} - T) + K_h \cdot u(t - t_{delay}) \tag{3.1}$$

With the system dynamics represented by the constants given in Table 3.1.

And the parameters and variables given in Table 3.2.

Figure 3.1: Picture of air heater [25].

Table 3.1: Air heater system constants.

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

Table 3.2: Air heater system parameters and variables

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

A more extensive presentation of the air heater can be found on the TechTeach web site: `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].

3.1.1 Model of Air Heater in Modelica

The air heater was implemented into Modelica as a model. Since the model heater gain is given with $\frac{C}{V}$ 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

```
model AirHeaterWithStep

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

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

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

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

initial equation
  T_Out = T_amb;

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

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

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

end AirHeaterWithStep;
```

Code 3.1: Air Heater Implemented in Modelica - Link to source code

the control signal over time. The resulting Code is presented in Code 3.1.

In Figure 3.2 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 3.1. From the plot in Figure 3.2 one can observe both the system time constant, and the delay between the control signal u and the output temperature T_{Out}.

3.1.2 Calculate PI Parameters for the Air Heater

From a step response introduced to the system as displayed in Figure 3.3 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

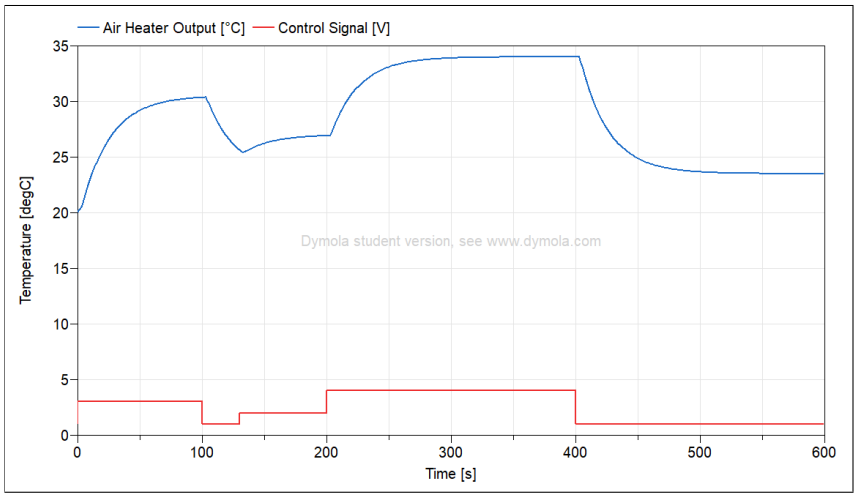

Figure 3.2: Plot of air heater with step changes in control signal

in Figure 3.3 [23]. 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 3.2.

$$S = \frac{T_2 - T_1}{t_2 - t_1} = \frac{38 - 25}{239 - 212} = \frac{13}{27} \simeq 0.481 \tag{3.2}$$

The integrator gain was calculated dividing the slope of the step response with the change in control signal displayed in Equation 3.3.

$$K_i = \frac{S}{U} = \frac{\frac{13}{27}}{4.5} = \frac{26}{243} \simeq 0.11 \tag{3.3}$$

The proportional gain of the controller was calculated with the use of the known time-delay given as $3[s]$ from Table 3.1 represented as τ displayed in Equation 3.4.

$$K_p = \frac{1}{2 \cdot K_i \cdot \tau} = \frac{1}{2 \cdot \frac{26}{243} \cdot 3} = \frac{81}{52} \simeq 1.56 \tag{3.4}$$

The integral time was calculated as displayed in equation 3.5.

$$T_i = 4 \cdot \tau = 12 \tag{3.5}$$

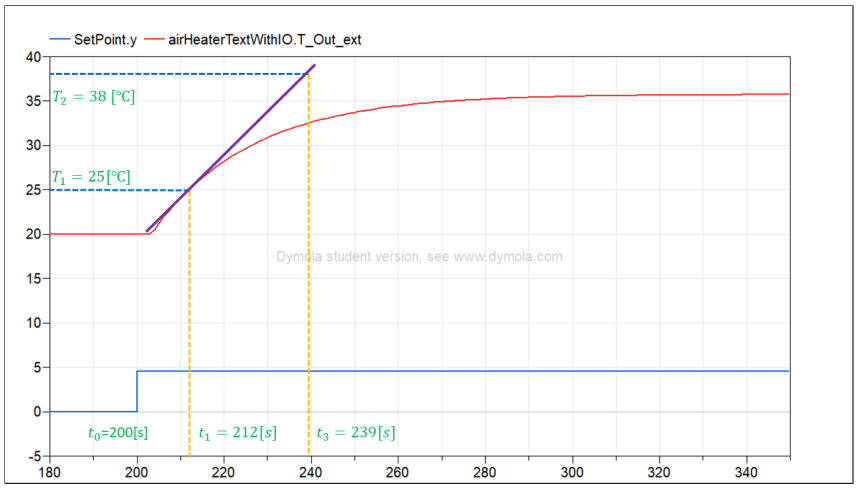

Figure 3.3: Skogestad method for tuning of PI-controller

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 3.4.

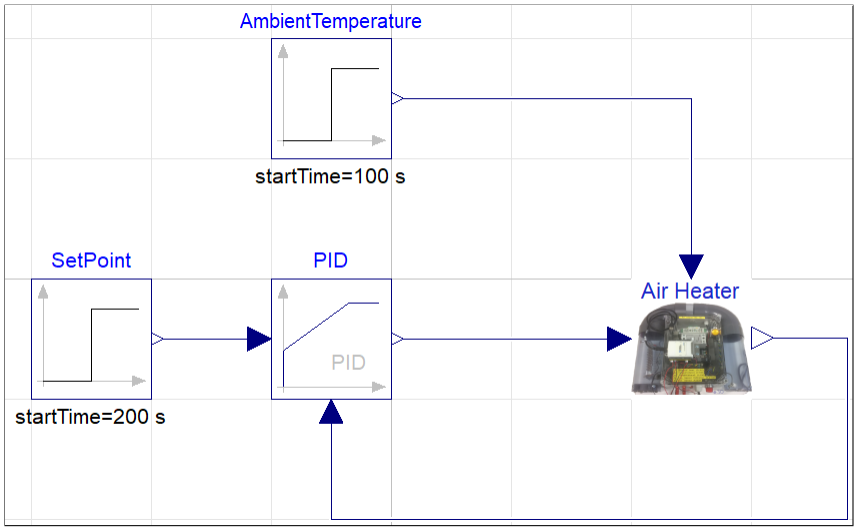

Figure 3.4: Picture of air heater with PID controller

In Figure 3.5 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.

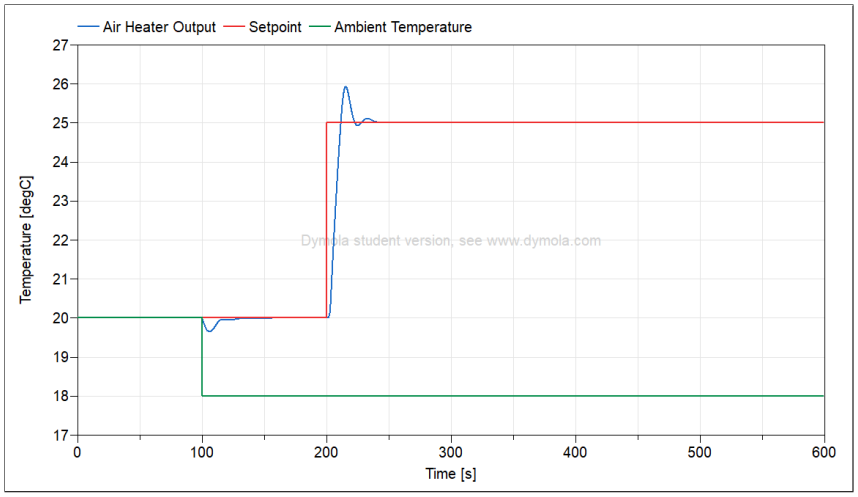

Figure 3.5: Plot of air heater with PID controller

3.2 Model of Air Heater Based on Transfer Function Model

The model parameters presented in Table 3.1 was used in a general 1.order transfer function with time-delay presented in Equation 3.6.

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 3.6. In addition to the model parameters, it was added a reference signal input, ambient temperature input, and a temperature output from the air heater.

$$H(s) = \frac{K}{Ts+1} \cdot e^{-\tau \cdot s} \longrightarrow H(s) = \frac{3.5}{23s+1} \cdot e^{-3 \cdot s} \tag{3.6}$$

The resulting plot from the simulation of the transfer function model is presented in Figure 3.7

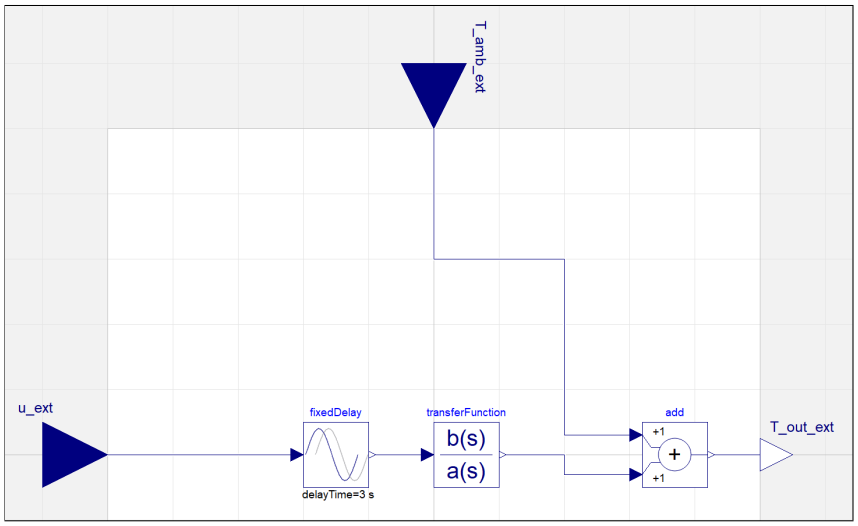

Figure 3.6: Air heater model implemented as a transfer function in Modelica

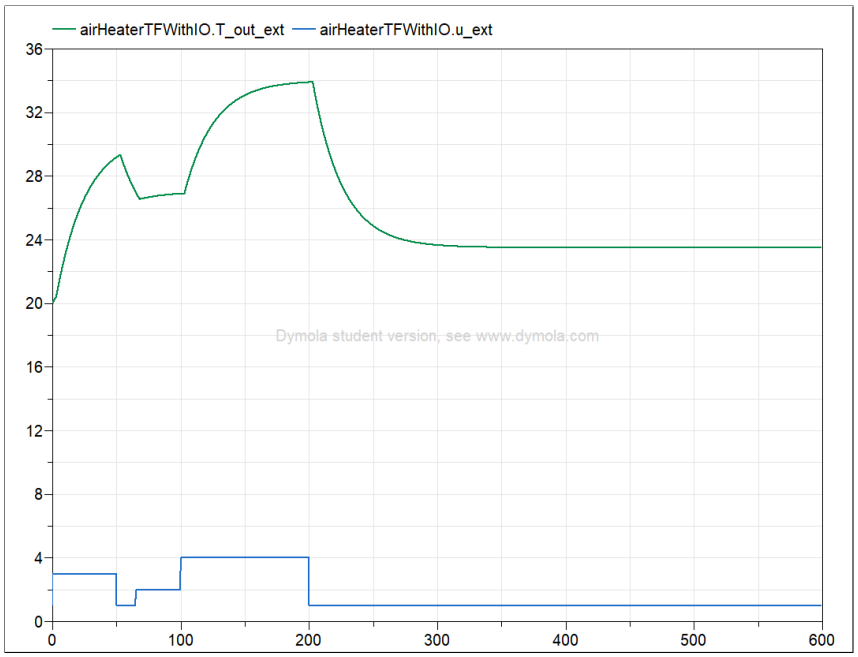

Figure 3.7: Plot from simulating transfer function model of air heater

3.3 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 3.2 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 3.2.

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 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.

```
import control as ct
import matplotlib.pyplot as plt
from control.matlab import *
import numpy as np
import pandas as pd

#System parameters and constants
# System Gain [C/V]
K = 3.5

# Time constant [s]
T = 23

# System time-delay
tau = 3

# Order of the padé approximation
N = 10

#Set up transfer function to represent the time constant dynamics of the air heater
num = [K]
den = [T, 1]
H1 = ct.tf(num,den)

# Set up transfer function to represent the time delay of air heater
[num_pade, den_pade] = ct.pade(tau,N)
Hpade = ct.tf(num_pade, den_pade)

# Connect the time constant and time delay transfer functions together
H = ct.series(H1, Hpade)

# Perform a step response to the transfer function system
ttf,ytf = ct.step_response(H)

# Transfer the model from transfer function model to continuous state space model
Hss = ct.matlab.tf2ss(H, dt=0)

# Performe a step response to the state space system
tss,yss = ct.step_response(Hss)

# print the max and min from the difference between the step response in the transfer function and state space model.
print ("The maximum value is: " + str(max(ytf-yss)) + "\nThe minimum value is: " + str(min(ytf-yss) ))

```

```
# Write matrices to .CSV files
pd.DataFrame(Hss.A).to_csv("DataFiles/AirHeaterSS_A.csv", sep=',',columns=None, index=False, decimal='.', header=False)
pd.DataFrame(Hss.B).to_csv("DataFiles/AirHeaterSS_B.csv", sep=',',columns=None, index=False, decimal='.', header=False)
pd.DataFrame(Hss.C).to_csv("DataFiles/AirHeaterSS_C.csv", sep=',',columns=None, index=False, decimal='.', header=False)
pd.DataFrame(Hss.D).to_csv("DataFiles/AirHeaterSS_D.csv", sep=',',columns=None, index=False, decimal='.', header=False)

# Generate a discrete time state space model
HssD = Hss.sample(0.01, method='euler')
# Write matrices to .CSV files
pd.DataFrame(HssD.A).to_csv("DataFiles/AirHeaterSSD_A.csv", sep=',',columns=None, index=False, decimal='.', header=False)
pd.DataFrame(HssD.B).to_csv("DataFiles/AirHeaterSSD_B.csv", sep=',',columns=None, index=False, decimal='.', header=False)
pd.DataFrame(HssD.C).to_csv("DataFiles/AirHeaterSSD_C.csv", sep=',',columns=None, index=False, decimal='.', header=False)
pd.DataFrame(HssD.D).to_csv("DataFiles/AirHeaterSSD_D.csv", sep=',',columns=None, index=False, decimal='.', header=False)
```

Code 3.2: Transfer function and state space model in Python - Link to source code

The exported .csv files generated in the Python script that is presented in Code 3.2 was imported into Modelica with the use of one continuous and one discrete time state space model blocks as displayed in 3.8. In addition, the ambient temperature was added as an input to the model blocks with the use of two add blocks.

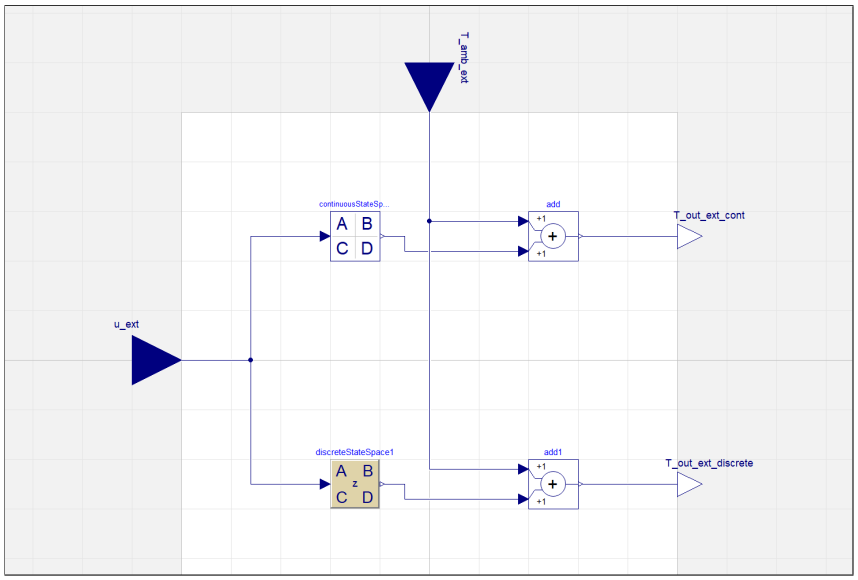

Figure 3.8: Air heater implemented as continuous and discrete time state space models.

3.3.1 Compare the Different Implemented Models of Air Heater

All the three generated blocks based on first principle, transfer function and state space were set up and excited with the same control signal sequence as displayed in Figure 3.9.

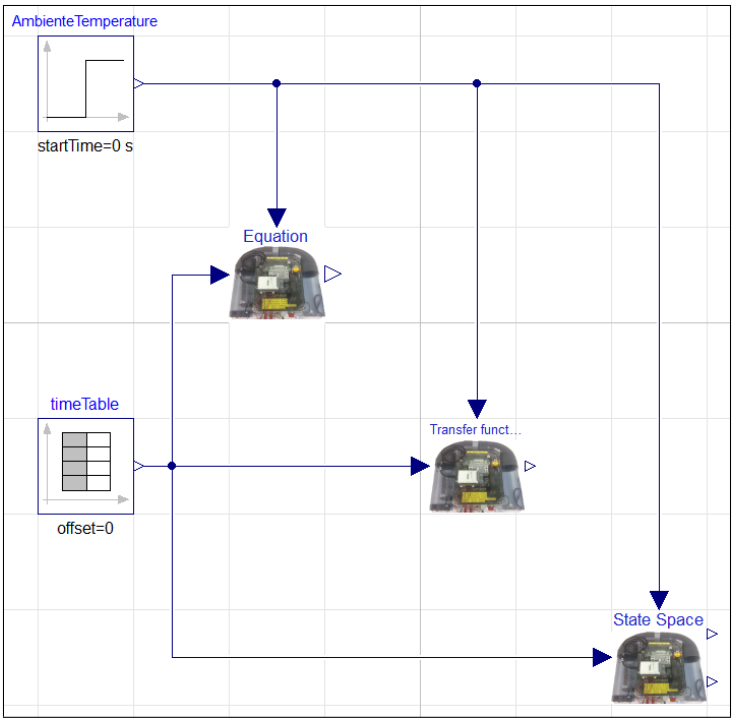

Figure 3.9: Simulation setup of air heater models based on equations, transfer functions, and state space.

The resulting plot is presented in Figure 3.10, 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 $t = 0$.

3.4 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 3.11 represents a relatively simple first-order chemical reaction model when compared to tubular reactors or packed-bed reactors. This simplicity makes the CSTR a convenient choice for illustrating modeling principles for

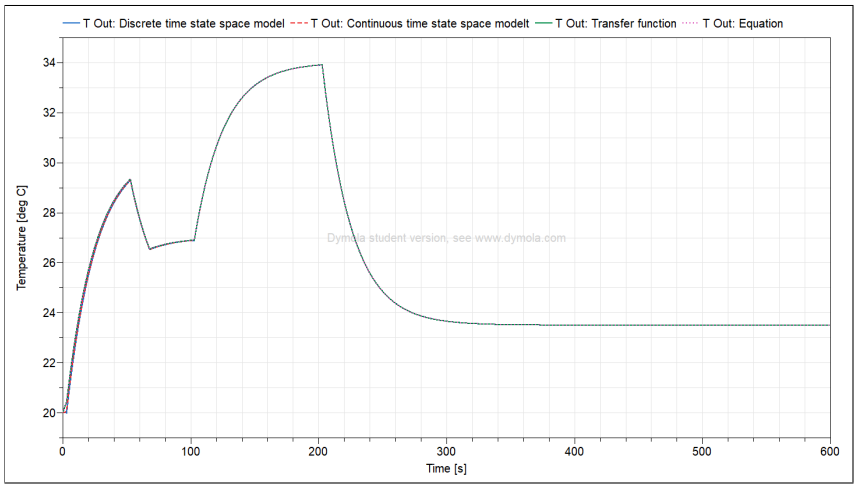

Figure 3.10: Plot of simulation setup of air heater models based on equations, transfer functions, and state space.

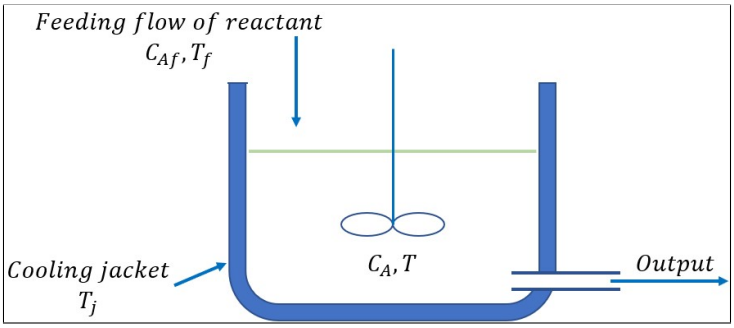

Figure 3.11: Sketch of the CSTR model.

chemical reactors. The implemented model is based on certain assumptions given in Table 3.3.

Table 3.3: 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 $Q = U \cdot A(T_c \cdot T)$
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 temper-

ature, 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 3.4

Table 3.4: 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 3.8 and Equation 3.7 respectively.

$$\dot{C_A} = \frac{F}{V} \cdot (C_{Af} - C_A) - k_0 \cdot \exp\frac{-E}{R \cdot T} \cdot C_A \tag{3.7}$$

$$\dot{T} = \frac{F}{V} \cdot (T_f - T) - \frac{\Delta_H}{\rho \cdot C_p} \cdot k_0 \cdot \exp\frac{-E}{R \cdot T} \cdot C_A + \frac{U \cdot A}{\rho \cdot C_p \cdot V} \cdot (T_j - T) \tag{3.8}$$

With the values and descriptions for the parameters presented in Table 3.5.

In Code 3.3 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 3.12. 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.

```
model ContinuousStirredReactor

    // Import
  import Modelica.Units.SI;

  // Constants
  constant Real R = 1.98589              "gas constant";

  // Parameters
  parameter SI.Temperature V = 1.0       "Liquid volume of reactor";
  parameter Real F = 1.0                 "Volumetric flow rate";
  parameter Real E = 11843.0             "Activation energy";
  parameter Real UA = 150.0              "U=overall heat transfer; A heat transfer area";
  parameter Real delH = -5960.0          "Heat of reaction per mole";
  parameter Real rhoCp = 500.0           "mass dencity and heat capacity";
  parameter Real k0 = 34930800.0         "Frequency factor";
  parameter SI.Temperature Tf = 298.15   "Temperature of liquid";
  parameter Real CAf = 10                "Concentration";

  // Variables
  // Manipulated input
  SI.Temperature Tj     "Jacket cooling temperature";

  //Outputs
  SI.Temperature T       "Temperature of medium in reactor";
  SI.Concentration CA    "Concentration of reactant inside the tank";

equation

if time>0 and time<100 then
      Tj = 280;
    elseif time>=100 and time<130 then
      Tj = 300;
    elseif time>=130 and time<200 then
      Tj = 305;
    elseif time>=200 and time<400 then
      Tj = 250;
    else
      Tj = 320;
end if;

der(T) = (F/V)*(Tf-T)-(delH/rhoCp)*k0*exp(-E/(R*T))*CA+((UA)/(rhoCp*V))*(Tj-T);
der(CA) = (F/V)*(CAf-CA)-k0*exp((-E)/(R*T))*CA;

end ContinuousStirredReactor;
```

Code 3.3: Continuous stirred tank reactor Implemented in Modelica - Link to source code .

Table 3.5: 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
Δ_H	-5960.0	$[\frac{kcal}{kgmol}]$	Heat of reaction per mole
ρC_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

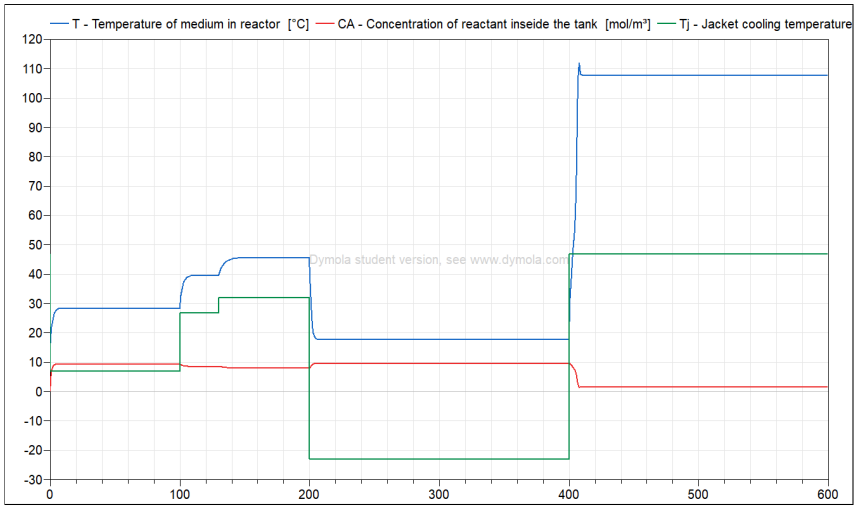

Figure 3.12: Simulation results for the implemented CSTR.

4 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 huge 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. [28].

According to the website 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
- 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/
- 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.

The 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 instead consider those alternatives.

4.1 Linear MPC Modelica Library

The only open-source general MPC library listed on the Modelica homepage (www.modelica.org/libraries.html) is the discontinued linear MPC library maintained by S. Hoelemann and D.Ablet at RWTH Aachen University in 2012. According to the repository on GitHub (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 [29].

4.2 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 [30]. With this approach Modelica is only used 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.

4.3 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 [31].

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 4. After the FMU is constructed in Python the model is used 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.

4.4 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 models 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 [32]. 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.

4.5 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 [33]:

- 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 [33].

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 [33].

4.5.1 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 used 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

4.6 Book - Modeling, Simulation and Control

In the book written by Finn Haugen called Modeling, Simulation, and Control [23] 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.

4.7 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 [34].

The FMPy packaged includes five 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 [34]:

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

4.8 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 [35]. 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

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 in some cases 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 4.1.

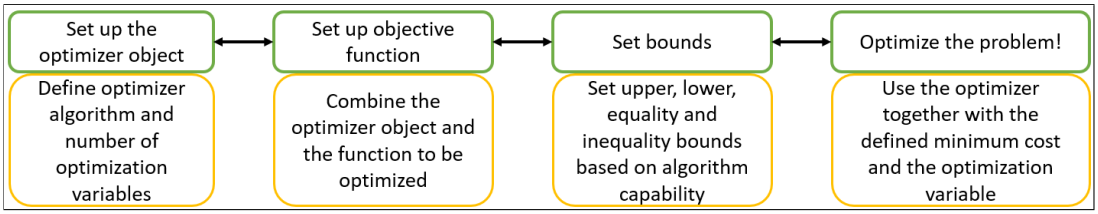

Figure 4.1: NLopt program flow.

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.

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 used 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. You can call the external object from both a Modelica model and a 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. You can choose to store all packages and files in a single file.
2. You can 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 your computer, enabling you to upload C source code into the file structure and access it from within OpenModelica.

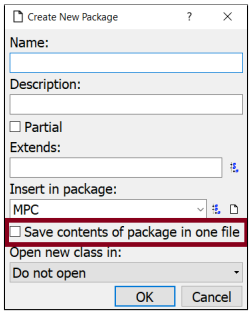

Figure 5.2: Configure Modelica folder structure

Figure 5.3 displays an example of the replicated file structure. Pictures used 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 subchapters used the simulation setup presented in Table 5.1.

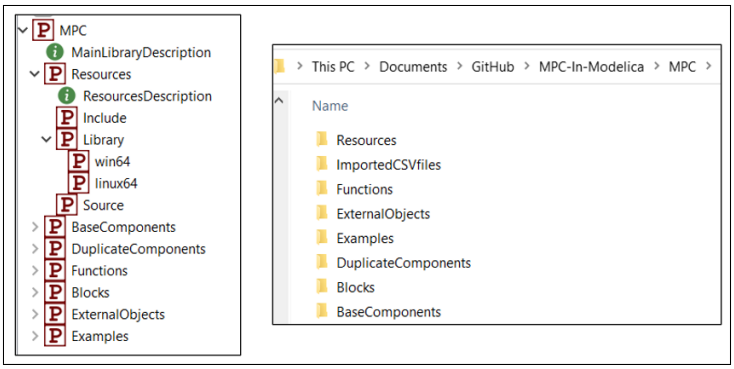

Figure 5.3: Replication of Modelica file structure on host computer

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}

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.

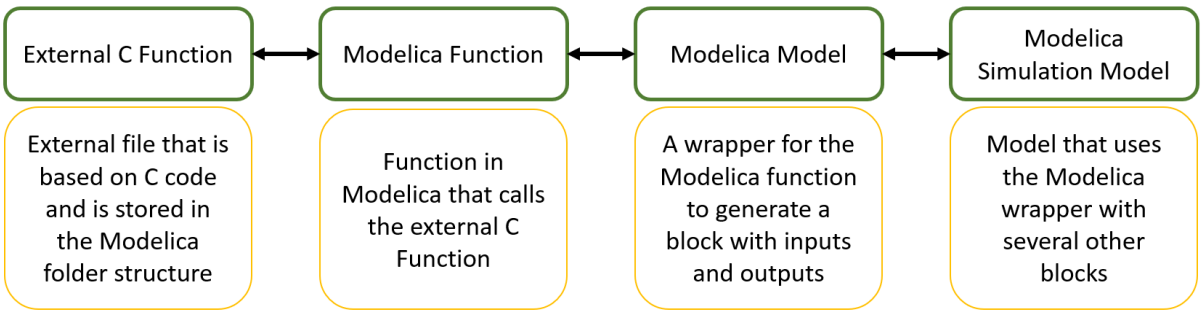

Figure 5.4: Process flow of calling external functions from Modelica

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.

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

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

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

  return result;

}
```

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.

```
function extNoiseGenerator "Function that call external c code to calculate random number"

  //Function parameters
  input Real range     "Total range of noise to be generated centered around input signal";
  output Real y_ext    "Distorted output signal";

  external "C" y_ext = NoiseGenerator(range)
    annotation (IncludeDirectory = "modelica://MPC/Resources/Source/",
                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.

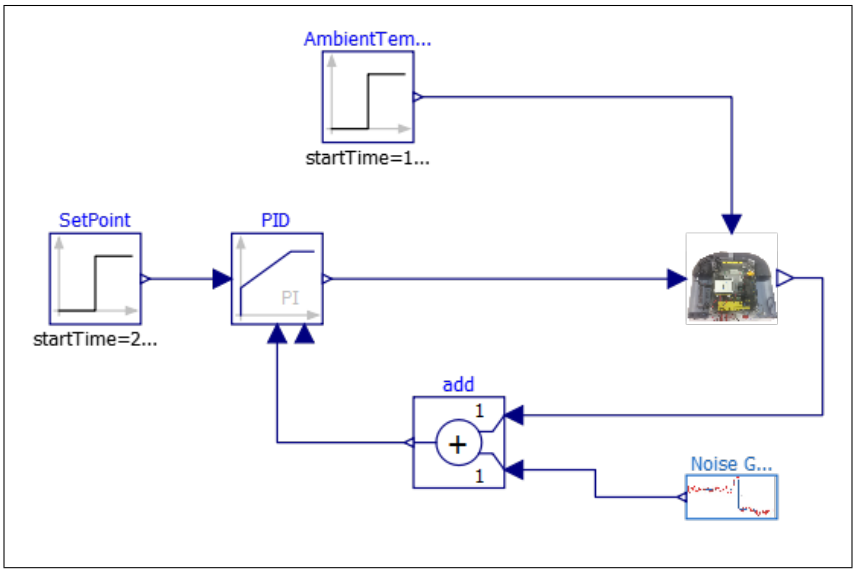

Figure 5.5: Model for simulating noise generated from external C code

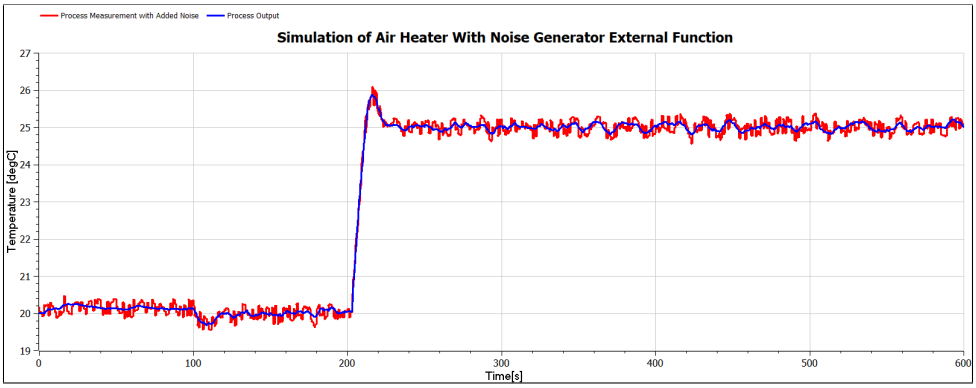

Figure 5.6: Plot of process output with both existing and newly added noise

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.

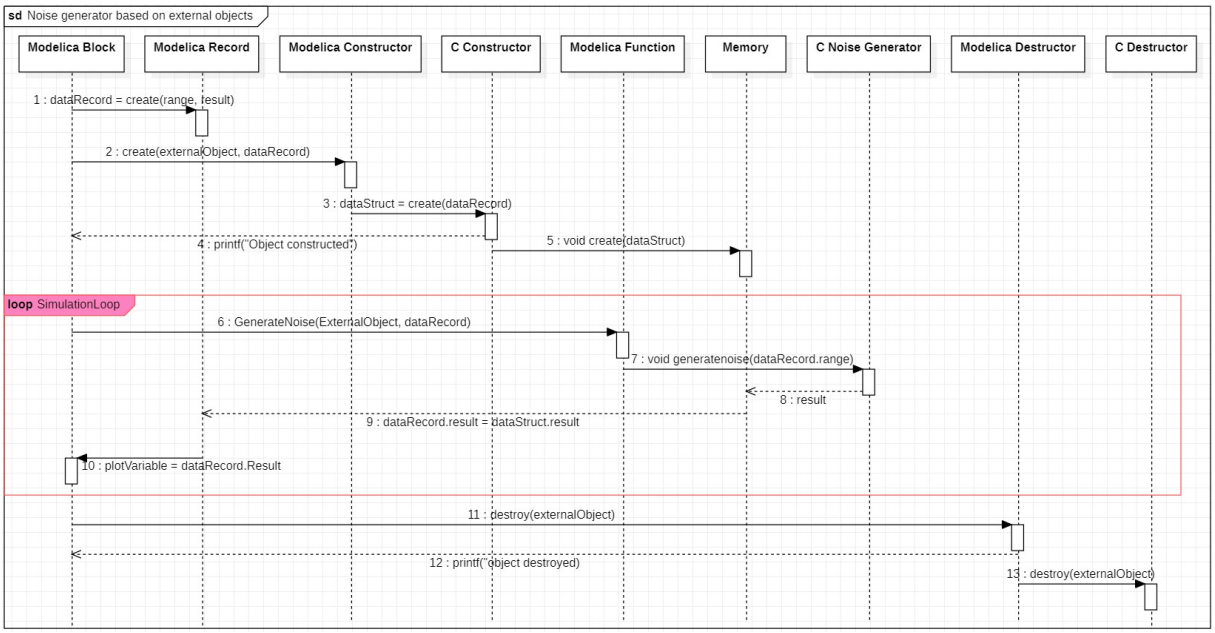

Figure 5.7: Sequence diagram for calling external noise generator with the use of Modelica external objects.

”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 used for the struct in the C source code presented in Code 5.5.

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

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

The corresponding struct in C is presented in Code 5.5.

```
typedef struct{
  double range;
  double result;
}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://MPC/Resources/Include/",
6                  Include = "#include \"NoiseGenerator.c\"");
7
8     end destructor;
9   end NoiseGeneratorExternalObject;
```

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

The corresponding C source code for the destructor is 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://MPC/Resources/Include/",
9                      Include = "#include \"NoiseGenerator.c\"");
10
11  end noiseGenerationCall;
```

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

The corresponding C function that is called from the Modelica function and generates the noise is presented in Code 5.11. The function takes in two void objects in line 1 being the external object and noiseGeneratorData record from OpenModelica. Based on the two external objects there is generated two objects inside the C function where one of the objects references the input values passed inn from the Modelica function, and the second is the return object reference set up to return 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

5.4 Comparison of Implementation Methods

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 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.

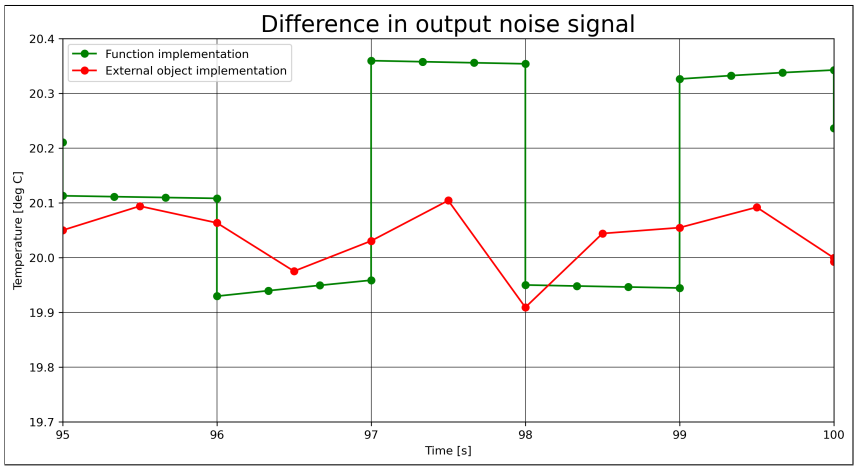

Figure 5.9: Simulation result of PID with all the different implementations in a time interval of 10 seconds.

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

Within this chapter, 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, are presented. This chapter explores three different implementations, each exhibiting 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 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.

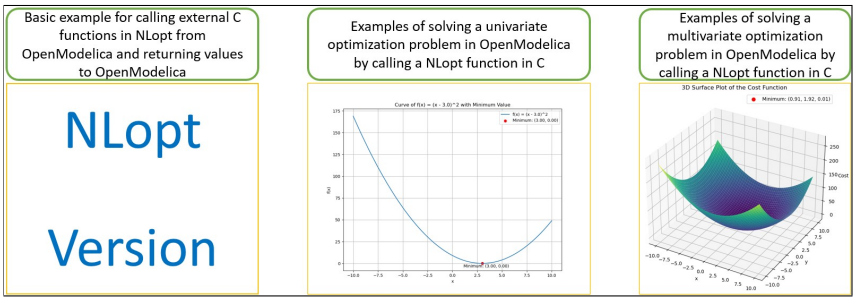

Figure 6.1: Three different blocks implemented in OpenModelica

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:

```
C:\src\vcpkg
C:\dev\vcpkg
```

Clone the repository with this command:

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

The next step is to build the vcpkg with bootstrap:

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

Then finally install the NLopt library for Windows 64bit with:

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

With vcpkg installed in the root directory, the NLopt library can be found in the first line below. The second line provides a reference to the Modelica implementations presented in the GitHub repository, indicating where the file needs to be moved. Access to the repository is available at the following link: Link to repository.

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

The NLopt .h file must be moved from the installed NLopt library with the path displayed in the first line below, to the Modelica folder given by the reference path in second line.

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

6.2 Implementation of the NLopt Version Block - NLoptVersionBlock

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.

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 it 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 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.

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, you can observe the individual variables extracted from the record in OpenModelica, each associated with the corresponding version number of the installed package.

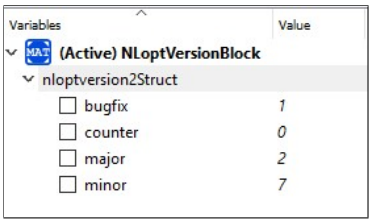

Figure 6.3: Output in OpenModelica simulation environment

6.3 Implementation of the NLopt Univariate Optimization - NLoptUniOptiBlock

The version function as introduced in Chapter 6.2 was 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. The segment of source code display-

ing the implementation of the objective function is presented in the GitHub repository (Link to source code).

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

As a reference an initial optimization was performed 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.

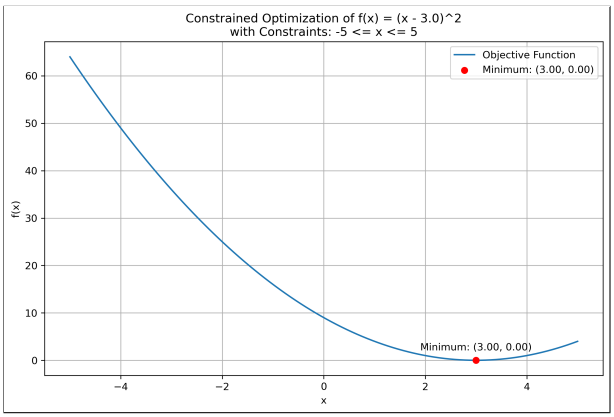

Figure 6.4: Plot of optimized univariate function - Link to source code

6.3.2 Optimization Parameters

In Figure 6.5, the diagram displays the path of 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.

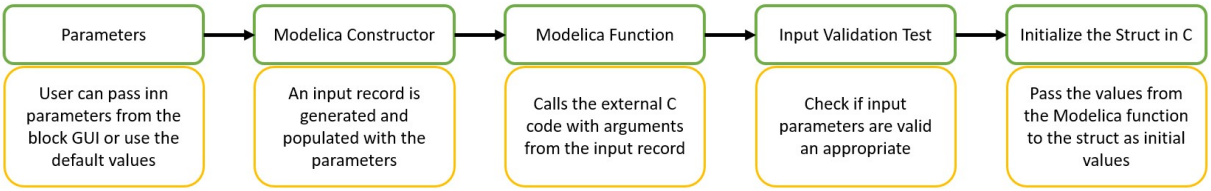

Figure 6.5: Diagram of flow for parameter from GUI all the way to allocated space in memory

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 in OpenModelica is displayed in Figure 6.6.

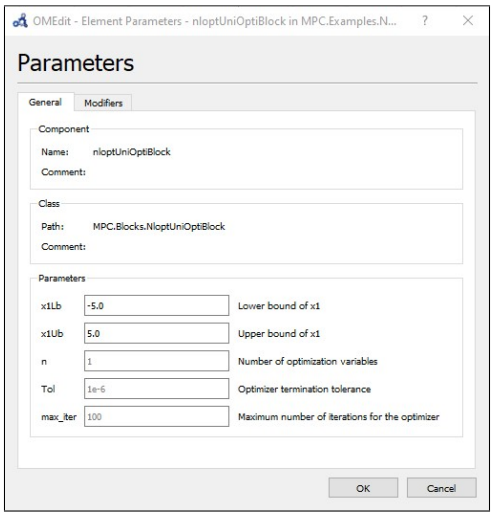

Figure 6.6: Available parameters for the Modelica block

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 used as arguments when calling the external constructor function called ”initialiseUniNloptInput()” in the C code.

```
1  // External C function call to initialize the NloptUniOptimize input.
2       external "C" nloptUnivariateEO = initialiseUniNloptInput(optimizationDataInput.x1R, optimizationDataInput.x1LbR,
       optimizationDataInput.x1UbR, optimizationDataInput.min_costR, optimizationDataInput.nR, optimizationDataInput.TolR,
       optimizationDataInput.max_iterR)
3       annotation(IncludeDirectory = "modelica://Resources/Include/",
4                  Include = "#include \"nloptUniOptimize.c\"");
5    end constructor;
```

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

Input Validation test

The source code in Code 6.5 demonstrates tests that were implemented for evaluating the values used to initialize the C struct, used when constructing the external object. The ”ModelicaError()” function is a pre-defined Modelica function designed to report error messages back to the Modelica environment. This function should be implemented in all external object implementations to manage cases where there is insufficient available memory to allocate the external object. An example of this implementation can be found in the GitHub repository at the following link: Link to source code

```
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.

```
typedef struct {
    double x1;          // Optimization variable
    double x1Lb;        // Lower bound of x1
    double x1Ub;        // Upper bound of x1
    double min_cost;    // Minimum value of the objective function after optimization
    int n;              // Number of optimization variables
    double tol;         // Termination tolerance
    int max_iter;     // Maximum number of iterations for the optimizer
    } 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 beginning 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_LN_COBYLA, optimizationDataInput->n); // Use the LN_COBYLA algorithm
3
4     // Set the objective function
5     nlopt_set_min_objective(optimizer, objective, NULL);
```

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

The optimization is performed in Code 6.11 where the results from the optimization and the minimum cost is written straight to the allocated place in memory.

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

Code 6.11: Optimize the problem - Link to source code

Finally, 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 faster.

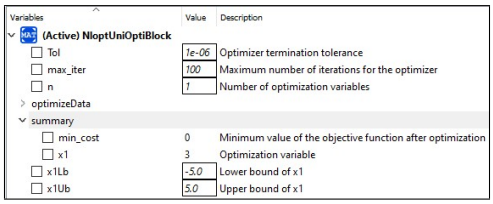

Figure 6.7: Simulation results from the NLopt univariate block

6.4 Implementation of the NLopt Multivariate Optimization - ”NLoptUMultiOptiBlock”

BESKRIVE FORSKJELLENE I IMPLEMENTASJON FRA UNIVARIATE TIL MULTI.

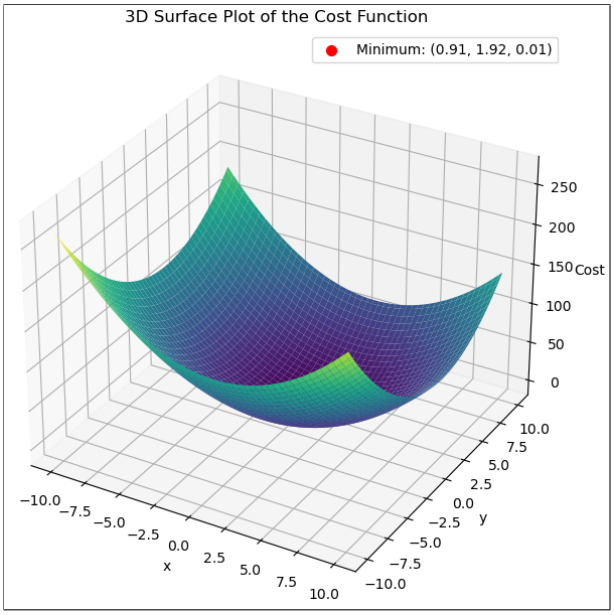

Figure 6.8: Plot of optimized multivariate function

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 4.7. Figure 7.1 provides a visual overview of the setup for evaluating the various exported FMUs.

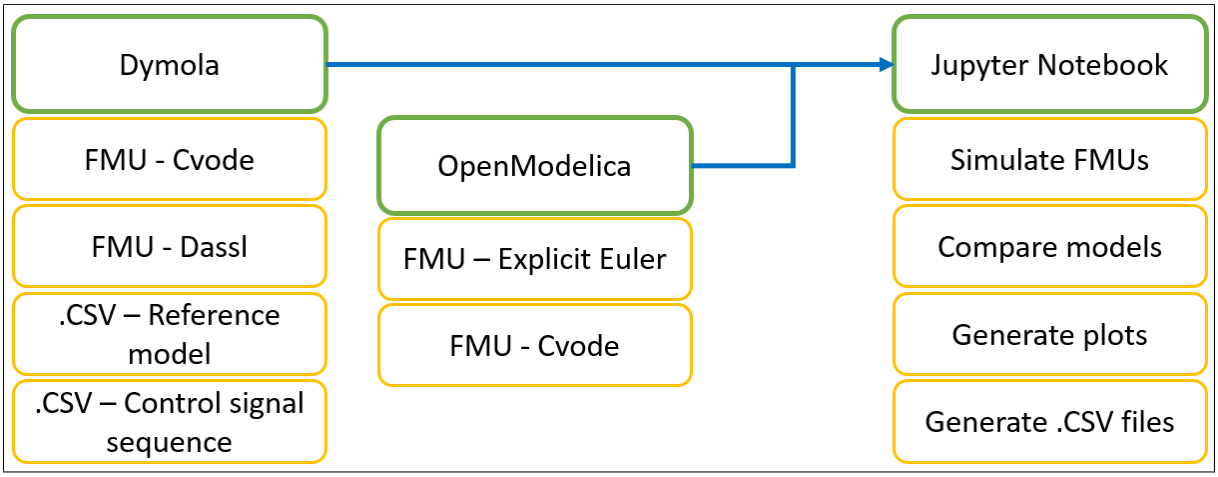

Figure 7.1: Overview of process for generating FMUs and evaluating the FMUs in Python.

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. Open Modelica - Explicit Euler
2. Open Modelica - 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 to source code).


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

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

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

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

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

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

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

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

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

In the source code provided in Code 7.2, 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
2  # Import the CSV file that holds the control signal sequence.
3  input = np.genfromtxt('datafiles/ControlSigna2.csv', delimiter=',', names=True)
4
5  # Simulation of the FMUs based on the imported csv control sequence.
6  result_airHeaterOMexpEul = simulate_fmu(airHeaterOMexpEul, start_values=start_values, output=output, stop_time=600.0, input
       =input, output_interval=0.5)
7  result_airHeaterOMcvode = simulate_fmu(airHeaterOMcvode, start_values=start_values, output=output, stop_time=600.0, input=
       input, output_interval=0.5)
8  result_airHeaterDymolaCvode = simulate_fmu(airHeaterDymolaCvode, start_values=start_values, output=output, stop_time=600.0,
        input=input, output_interval=0.5)
9  result_airHeaterDymolaOEM = simulate_fmu(airHeaterDymolaOEM, start_values=start_values, output=output, stop_time=600.0,
       input=input, output_interval=0.5)
10
11 # Write simulation data to CSV files
12 write_csv('datafiles/airHeaterOMexpEulSimData.csv', result_airHeaterOMexpEul, columns=None)
13 write_csv('datafiles/airHeaterOMcvodeSimData.csv', result_airHeaterOMcvode, columns=None)
14 write_csv('datafiles/airHeaterDymolaCvodeSimData.csv', result_airHeaterDymolaCvode, columns=None)
15 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 Open Modelica into a variable
11 dfOMeE = pd.read_csv("datafiles/airHeaterOMexpEulSimData.csv")
12
13 #FMU model generated with Cvode in Open Modelica into a variable
14 dfOMcv = pd.read_csv("datafiles/airHeaterOMcvodeSimData.csv")
15
16 #FMU model generated with Cvode in Dymola into a variable
17 dfDYMcv = pd.read_csv("datafiles/airHeaterDymolaCvodeSimData.csv")
18
19 #FMU model generated with Dymola Dassl solver in dymola into a variable
20 dfDYMdas = pd.read_csv("datafiles/airHeaterDymolaOEMSimData.csv")
21
22 # Remove the percentiles from the output
23 perc = []
24
25 # Write statistical properties of the imported data from Modelica simulation to a CSV file
```

```
dfRef.describe(percentiles=perc).to_csv("Tables/describeModelicaReference.csv")

# Write statistical properties of the imported FMU model generated with explicit Euler in Open Modelica to a CSV file
dfOMeE.describe(percentiles=perc).to_csv("Tables/describeOMexpEul.csv")

# Write statistical properties of the imported FMU model generated with Cvode in Open Modelica to a CSV file
dfOMcv.describe(percentiles=perc).to_csv("Tables/describeOMcvode.csv")

# Write statistical properties of the imported FMU model generated with Cvode in Dymola to a CSV file
dfDYMcv.describe(percentiles=perc).to_csv("Tables/describeDymolaCvode.csv")

# Write statistical properties of the imported FMU model generated with Dymola Dassl solver in dymola to a CSV file
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

These .csv files were imported into Excel for a preliminary visual inspection of the data in order 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.

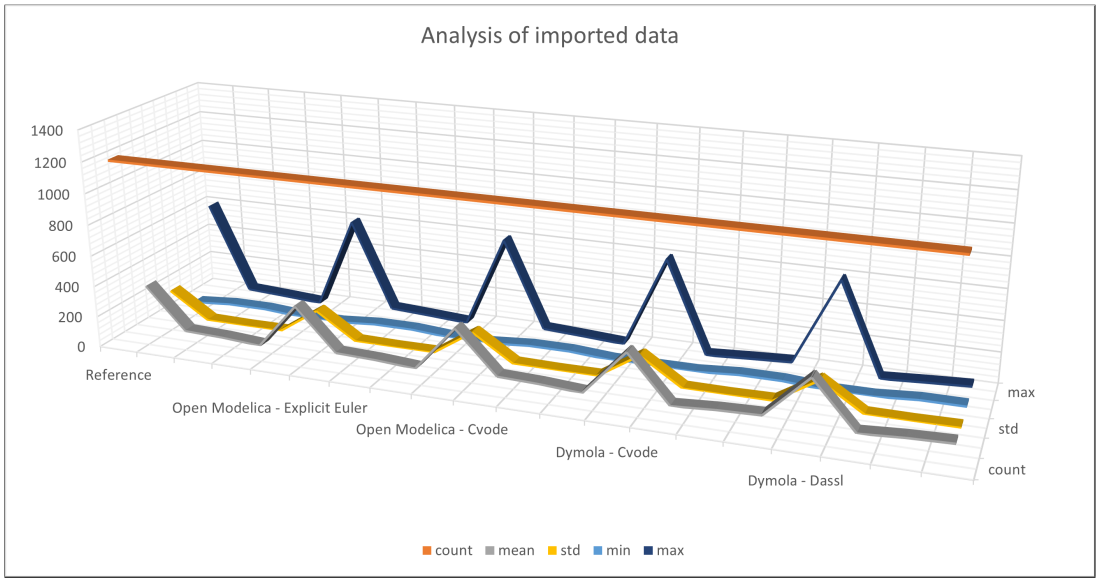

Figure 7.2: Initial analysis of imported simulated data.

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 visualized. The plot also includes the control signal for the simulations.

With the given resolution of the plot presented in Plot7.3 all models represent the model dynamics with relatively good accuracy and precision.

```
1
2  # Generate a new dataframe to evaluate the difference between the different FMUs temperature output and the Dymola
       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

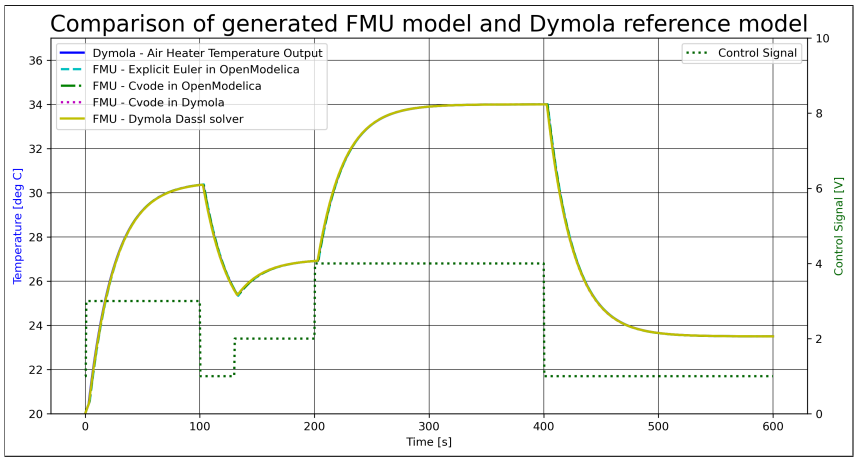

Figure 7.3: Plot of result from simulation of Python and Dymola reference.

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.

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

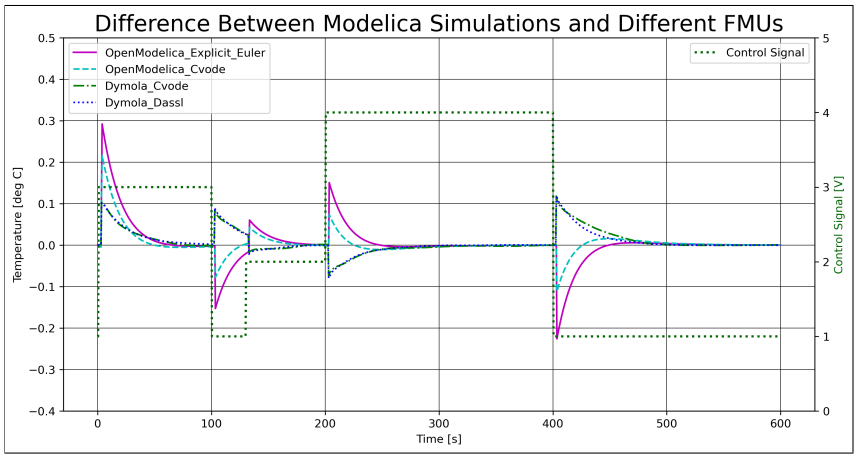

Figure 7.4: Difference in output temperature for FMU simulations in Python and Reference model from OpenModelica.

7.5. A bar plot of the calculated statistical data frame is presented in Figure 7.5.

```
# Write statistical properties of the difference between the imported FMU models and the Modelica export to a CSV file
describe_df_dfTOut_diff =dfTOut_diff.describe(percentiles=perc).to_csv("Tables/describeDifference")

# Write statistical properties of the difference between the imported FMU models and the Modelica export variable for
    plotting
describe_df_dfTOut_diff = dfTOut_diff.describe(percentiles=perc)

# Remove the variables time, counter, median, and mean
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

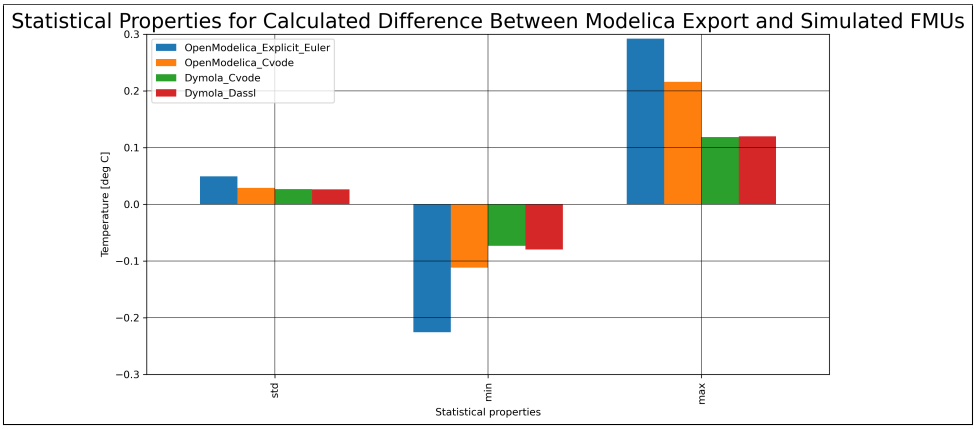

Figure 7.5: Statistical properties for the difference between Python and Modelica model.

8 Implementation of Air Heater and FMU with PI Controller in Python

This chapter presents a implementation of a FMU together with a PI controller in Python. The results from the simulation is also compared with a simulation performed in OpenModelica. The complete source code and installation information can be found in the GitHub repository (Link to repository).

In this chapter there is a presentation of a Python implementation of a PI controller together with a model of the air heater exported from OpenModelica as a FMU. The FMU is then imported into Python with the use of the FMPy package presented in Chapter 4.7. The implemented PI controller is based on the wood-chip tank with PI controller from techteach.no at the following link: Link to source code

8.1 Implementation of FMU and PI in Python

Figure 8.1

8.2 Validation of the results

Figure 8.2

Figure 8.3

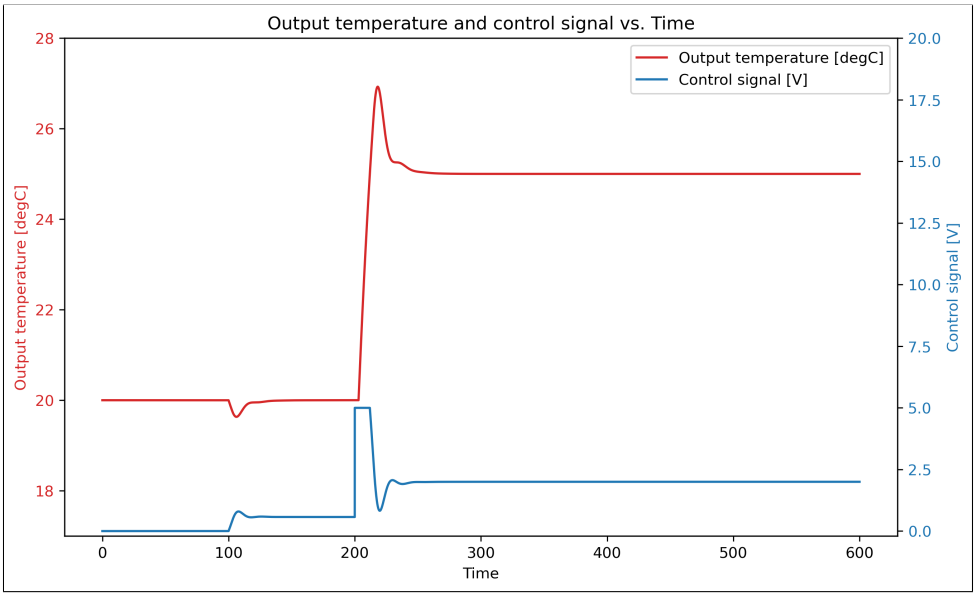

Figure 8.1: Resulting plot for simulation of PI controller and FMU of a air heater in Python.

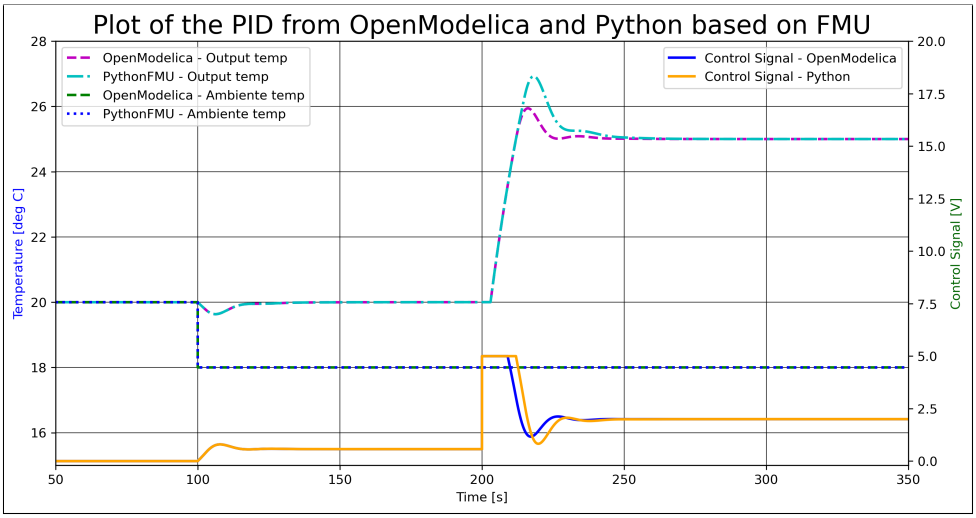

Figure 8.2: Resulting plot for simulation of PI controller and FMU of a air heater compared with simulation from OpenModelica.

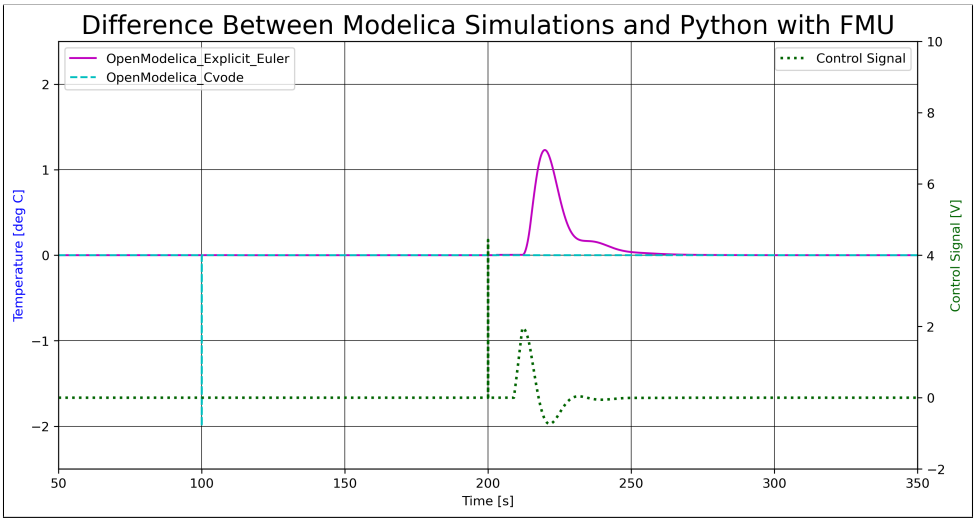

Figure 8.3: Resulting plot comparison of the difference between PI in Python and OpenModelica

9 FMU of Air Heater with MPC in Python

The complete source code and installation information can be found in the GitHub repository (Link to repository).

10 Conclusion

Bibliography

[1] 'Yara at a glance.' (Sep. 2022), [Online]. Available: https://www.yara.com/this-is-yara/yara-at-a-glance/.

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

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

[4] B. Lie, *Modelling of dynamic systems*, Accessed: 2018-12-06, 2019.

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

[6] S. E. Mattsson and H. Elmqvist, 'Modelica - an international effort to design the next generation modeling language,' eng, *IFAC Proceedings Volumes*, vol. 30, no. 4, pp. 151–155, 1997, ISSN: 1474-6670.

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

[8] 'Modelica language.' (Jan. 1), [Online]. Available: https://modelica.org/modelicalanguage.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] 'Itea project emphysis.' (Feb. 2021), [Online]. Available: https://emphysis.github.io/.

[12] 'Functional mock-up interface for embedded systems.' (Feb. 2021), [Online]. Available: https://emphysis.github.io/pages/downloads/efmi_specification_1.0.0-alpha.4.html.

[13] S. development group, *System structure and parameterization standard 1.0*, 2019.

[14] M. D. J. K. P. R. M. K. M. Nagasawa, 'Ma-project "system structure and parameterization" – early insights,' presented at the Japanese Modelica Conference (Tokyo, Japan, 23rd–24th May 2016), 2016.

[15] 'Dcp - distributed co-simulation protocol.' (), [Online]. Available: `https://dcp-standard.org/`.

[16] 'Distributed co-simulation protocol (dcp) 1.0.' (Mar. 2019), [Online]. Available: `https://dcp-standard.org/assets/specification/DCP_Specification_v1.0.pdf`.

[17] C. Horgan and L. van den Borne, 'ACOSAR and its DCP – a protocol for celebration,' *ITEA3 News Item*, Sep. 2018. [Online]. Available: `https://itea3.org/news/acosar-and-its-dcp-a-protocol-for-celebration.html`.

[18] M. Krammer, K. Schuch, C. Kater *et al.*, 'Standardized Integration of Real-Time and Non-Real-Time Systems: The Distributed Co-Simulation Protocol,' in *Proceedings of the 13th International Modelica Conference, Regensburg, Germany, March 4–6, 2019*, vol. 157, Regensburg, Germany: Modelica Association, 2019, pp. 87–96. DOI: `10.3384/ecp1915787`.

[19] 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.

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

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

[22] *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.

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

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

[25] 'Air heater.' (), [Online]. Available: `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.

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

[29] 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.

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

[31] 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.

[32] 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.

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

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

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

Appendix A

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

The signed task description as the basis for the thesis

University of South-Eastern Norway

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

FMH606 Master's Thesis

Title: Development of MPC blocks for Modelica

USN supervisor: Finn Aakre Haugen

External partner: Yara Porsgrunn – Anushka Perera

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

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

Student category: IIA

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

Practical arrangements: N/A

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

Signatures:

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

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

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