Master's thesis

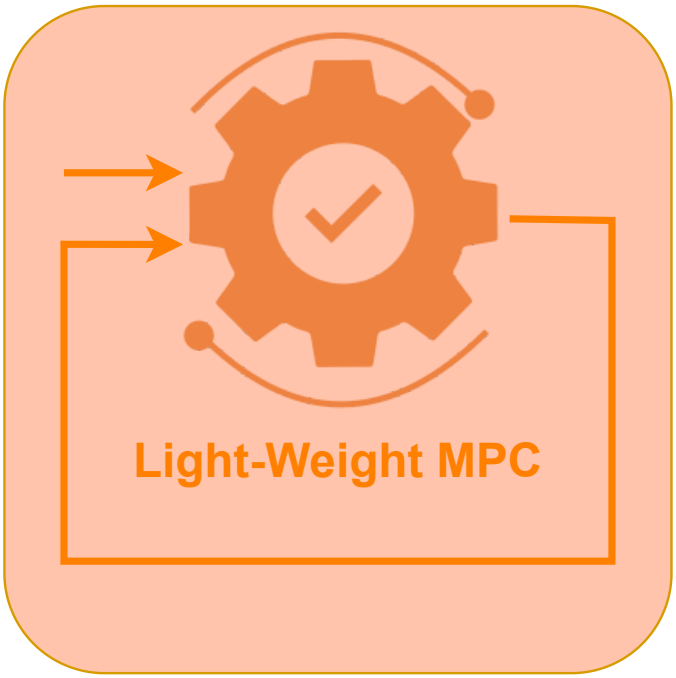
NTNU
Norwegian University of Science and Technology
Faculty of Information Technology and Electrical Engineering
Department of Engineering Cybernetics

Geir Ola Tvinnereim

Light-Weight MPC

A lightweight simulation software implementing linear Model Predictive Control on Finite Step-Response Models

Master's thesis in Cybernetics and Robotics
Supervisor: Prof. Lars Struen Imsland
Co-supervisor: Dr. Gisle Otto Eikrem
June 2023



NTNU
Norwegian University of Science and Technology

equinor

Geir Ola Tvinnereim

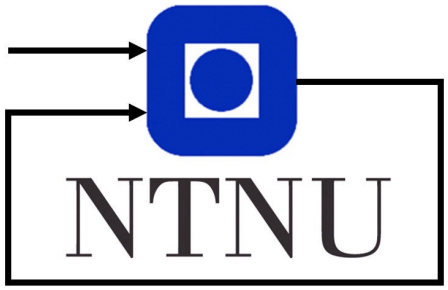
Light-Weight MPC

A lightweight simulation software implementing linear Model Predictive Control on Finite Step-Response Models

Master's thesis in Cybernetics and Robotics
Supervisor: Prof. Lars Struen Imsland
Co-supervisor: Dr. Gisle Otto Eikrem
June 2023

Norwegian University of Science and Technology
Faculty of Information Technology and Electrical Engineering
Department of Engineering Cybernetics

NTNU
Norwegian University of
Science and Technology



Department of Cybernetics Engineering

TTK4900 - Engineering Cybernetics, Master's Thesis

Light-Weight MPC

Author:
Geir Ola Tvinnereim

Supervisor:
Prof. Lars Struen Imsland

Co-supervisor:
Dr. Gisle Otto Eikrem

19th June 2023

Abstract

Model Predictive Control (MPC) has become increasingly important due to its ability to optimally control complex systems. By taking predictions and system constraints into account, the control law can improve the reliability and safety of industrial operations. MPC can also help industries comply with environmental regulations by optimizing energy consumption and reducing climate emissions. This makes it superior to classical approaches, as it can improve overall productivity. With the increasing demand for sustainable and efficient solutions, the controller is applicable to a wide range of industries, including manufacturing, process, robotics and energy. MPC has become a critical component of industrial control systems and is considered the main modern control approach.

Light-Weight MPC is an open-source simulation framework exploiting the performance of linear Model Predictive Controllers. As a lightweight software, it has minimal resource requirements, a small memory footprint, and efficient execution prioritising a simple interface. Different scenarios can be addressed by defining a system description and a controller tuning. Simulation data is neatly structured using JavaScript object notation (JSON), which can be interfaced with several other frameworks. The simulator is C++ implemented using the Operator Splitting Quadratic Solver (OSQP) software as the controller optimizer. The algorithm addressed is based on the formulation derived from *High-performance Industrial Embedded Model Predictive Control* (Kufoalor, Imsland and Johansen, 2015). The derived controller considers slack constraints and bias correction as the fundamental components of MPC functionality required for lightweight software. The corresponding system description addressed is the Finite Step-Response Model (FSRM). A widely used model in the process industry. This model serves as both the control and plant model, hence the simulator does not take model errors into account.

In order to make the software accessible to many and easy to use for a wide range of devices, an interfacing Create-React-App-based application was developed. The simulator is ported to the app using Emscripten Compiler Frontend and Webassembly (Wasm), a high-performance binary instruction format designed for the web. This setup allows the application as a whole to be executed in the browser without the need for server-side callbacks. The simulation software serves as an academic tool to learn, test and explore linear Model Predictive Controllers. The framework is open-source distributed and licensed under *BSD 3-Clause*: https://github.com/orgs/Light-weight-MPC/repositories.

Samandrag

Modell-prediktiv regulering har vorte eit viktig verktøy i moderne industri. Mykje på grunn av dugleiken til å optimalt styre avanserte system. Ved å ta prediksjonar og systembegrensingar i betraktning kan styringslova heve grad av sikkerheit og pålitelegheit for industrielle operasjonar. Den modellbasserte styringsmetoden kan også hjelpe industriar med å etterkomme miljøkrav ved å optimalisere energibruk og redusere klimautslepp. Dette gjer den overlegen i forhold til klassiske regulatorar, sidan den kan forebetre produktiviteten på fleire plan. Med ein aukande etterspurnad etter berekraftige og effektive løysingar, kan styringsmetoden nyttast innan ei rekke ulike industriar, til dømes innanfor produksjon, prosess, robotikk og energi. Modell-prediktiv regulering har vorte ein kritisk komponent i industrielle styringssystem og er antatt å vere ein sentral styringsmetode i dagens samfunn.

Light-Weight MPC er eit simuleringsverktøy nytta til å vurdere ytinga til den modellbasserte regulatoren. Som ei lettvektsklasse programvare brukar den minimalt med ressursar slik som minne. Grensesnittet er enkelt oppbygd og brukarvennleg. I tillegg, skal programmet køyre effektivt. Forskjellege senario kan undersøkast ved å definere ein system-modell og styrings-tuner. Simuleringsdataen er ryddig strukturert i JavaScript object notation (JSON), og er lett tilgjengeleg for analyse i fleire andre rammeverk. Simulatoren er programmert i C++ og importerar Operator Splitting Quadratic Solver (OSQP) programvara til å optimalisere styringa. Algoritma nytta er basert på formuleringa skildra i doktor-avhandlinga *High-performance Industrial Embedded Model Predictive Control* (Kufoalor, Imsland og Johansen, 2015). Denne utleda stryingsformuleringa antek slakk variablar og feilkorrigering som kjernefunksjonaliteten i ein Modell-prediktiv regulator. Styringsmetoden basserar seg på endelege sprangresponsemodellar, som er ein mykje brukt modell i prosess-industrien. Desse modellane vert både brukt til styring i tillegg til å skildre modellen. Dermed tek ikkje simulatoren modellfeil i betrakning.

For å gjere verktøyet tilgjengeleg og brukarvenleg for mange ulike apparat, vart ein Create-React-App-bassert nettsideløysing utvikla. Simulatorlogikken blir overført til applikasjonen ved bruk av Emscripten Compiler Frontend og Webassembly (Wasm). Wasm er eit høg-effektivt binært instruksjonsformat designa for nettsider. Dette oppsettet er uavhengig frå bruken av serverkall sidan heile applikasjonen vert køyrt i nettlesaren. Simuleringsprogrammvara er eit akademisk verktøy for å lære, teste og utforske lineære Modell-prediktive regulatorar. Rammeverket er ope kjelde publisert og lisensiert under *BSD 3-Clause*: https://github.com/orgs/Light-weight-MPC/repositories.

Preface

This master's thesis serves as the final assignment for my integrated Master of Science (M.Sc.) in Cybernetics and Robotics at the *Norwegian University of Science and Technology* (NTNU). The project description is handed out by *Equinor* and formulated in collaboration with the *Department of Engineering Cybernetics* (ITK). The work has been supervised and co-supervised by respectively *Prof. Lars Struen Imsland* and *Dr. Gisle Otto Eikrem*. Thank you so much for your invaluable assistance in helping me complete this project with a time stamp of one year. You provided constructive and swift feedback whenever I was in need of guidance. I would point out a special thank you to *Prof. Imsland* for lending me your book *A First Course in Predictive Control* by *J.A. Rossiter* along with other high-quality academic material for me to get a thorough understanding of the complex controller strategy. I am also immersively thankful for the follow-up from *Dr. Eikrem* whenever my knowledge about web development and software deployment came too short.

It was really interesting taking on such an implementation-heavy project. I have always been fascinated with how technology can be used for problem-solving and wanted to contribute to this development myself. With the aim of becoming a proficient software engineer, I found it motivating to design and assess software libraries in order to achieve the best possible implementation. I hope that *Light-Weight MPC* serves its purpose of being an industrial and academic simulation tool for tuning and testing linear Model Predictive Controllers. Unless otherwise stated, all figures and illustrations have been created by the author.

Geir Ola Tvinnereim

Trondheim, 19th June 2023

Table of Contents

List of Figures

List of Tables

Acronyms

MPC Model Predictive Control

MPC Model Predictive Controller

PID proportional-integral-derivative

SEPTIC *Estimation and Prediction Tool for Identification and Control*

FSRM Finite Step-Response Model

MIMO Multiple Input Multiple Output

SISO Single Input Single Output

CV controlled variable

MV manipulated variable

QP Quadratic Program

MPC-FSRM Model Predictive Controller for Finite Step-Response Model

IO input-output

OS operating system

GCC GNU compiler collection

JSON JavaScript object notation

OSQP *Operating Splitting Quadratic Solver*

SSM State Space Model

API Application programming interface

CLI command-line interface

UI user-interface

UX user-experience

CRA *Create React App*

Wasm Webassembly

emcc *Emscripten Compiler Frontend*

HTTP Hypertext Transport Protocol

FMU Functional Mock-up Interface

1 Introduction

1.1 Project background and goal

The field of process control has evolved significantly over the past few decades. With the growing complexity of industrial systems, there has been a corresponding increase in the demand for more advanced and sophisticated control techniques. One such technique that has received considerable attention in recent years is Model Predictive Control. MPC is a control strategy that exploits mathematical models to predict system behaviour over a receding horizon. The predictions are used to optimize the control inputs to fulfil desired performance objectives. Predictive control approaches have become the de facto norm in the industry. Especially, in cases where aspects such as constraints are considered and/or if small improvements in performance can result in huge increases in profit. For such industrial plants, the extra expense of a model-based controller approach can be justified [23].

Even though the controller is considered a state-of-the-art method, the approach is typically unrecognized by individuals. MPC design can be significantly more complex compared to classical control methods such as the proportional-integral-derivative (PID) controller. Implementing an Model Predictive Controller successfully often involves extensive knowledge of system dynamics, mathematical modelling and optimization theory. This expertise may not be readily available in all industrial settings, making it challenging to adopt MPC in many applications.

Over the last decades, there has been an exponential growth of computing power. This has increased the applicability of more powerful technological solutions, such as the addressed controller method. With the advancement in computing power, MPC has become feasible for handling more intricate and detailed models. It is now possible to efficiently control complex systems that involve a large number of states, constraints, and variables. Such models enable more accurate predictions and precise control actions, leading to improved system performance. Additionally, the growth of computing power allows longer prediction horizons. By considering longer horizons, a broader perspective of future system behaviour can be obtained, enabling more informed decision-making and leading to enhanced stability properties. MPC can consider complete dynamics, leading to better control strategies that optimize performance over extended periods. Not to mention the impact this movement has had on the available optimizers. Complex optimization problems associated with MPC, such as quadratic and nonlinear programming, can be solved quickly allowing real-time control implementation. This is particularly beneficial for fast-changing systems where real-time decision-making and control execution time are crucial.

Equinor is an international energy company that maintains its headquarters in Norway. The organization conducts operations both onshore and offshore. Its industries encompass a variety of sectors, including oil and gas, solar energy, offshore wind, hydrogen, and others. Such systems often yield complex multi-variable system dynamics, which can be challenging to control efficiently with the use of classical control approaches such as PID control. Therefore, the predictive controller strategy plays a central role in many of *Equinor's* processes and is widely used. In order to apply Model Predictive Control in practice,

the company has its own in-house control software called *Estimation and Prediction Tool for Identification and Control* (SEPTIC), with its first release in 1997. According to [9], today the control software is used in applications such as oil refining, gas processing and offshore production. However, MPC has found applicability in emerging fields such as CO2 Capture and Storage (CSS) and energy management. MPC is continuously addressed to maintain and further develop *Equinor's* running applications both on and offshore. [9] also concludes that the utilization of the control approach can increase production up to 5%, which is a considerable growth in business. The controller enables this advancement by optimising the use of resources in line with a given cost. In other words, the controller expresses a preference for a production system that is environmentally and socially sustainable.

Clearly, both industry and society themselves benefit from using this advanced control method and it should be considered in every reasonable use case. This applies not only to existing applications but also to the ones yet to be discovered. Furthermore, as a result of recent technological advancements, modern solutions consistently surpass classical approaches. Therefore, opting for a predictive control approach offers even greater advantages. Consequently, there is a growing need for simulation software to effectively identify and address relevant use cases. By automating MPC scenarios, one can easily assess the performance and compare control data from different controllers. This simplifies the control engineering design and decision-making process. Additionally, having automated simulation software available, engineers can predict and optimize behaviour achieving the pre-eminent controller. The development of such simulation software in a lightweight manner is the aim of this master's thesis.

1.2 Problem description

The purpose of this master thesis is to create an open-source lightweight simulation software for linear MPC implementations. In the realm of software development, lightweight software refers to applications or programs that have minimal resource requirements, a small memory footprint, and efficient execution. *Light-Weight MPC* shall prioritize simplicity, fast performance, and low system overhead, making it suitable for resource-constrained environments. The revised version emphasizes the key characteristics of lightweight MPC and acknowledges the potential challenge of high run-time complexity. It also highlights the importance of managing computational efforts when dealing with such complexity. The simulation software targets the core functionality of SEPTIC, implementing central Model Predictive Controller elements. The simulator shall fulfil the following criteria:

- Based on suitable input files, the software should generate the controller and produce corresponding model simulation data on a discrete-time horizon, T, with constant controller objectives.
- The simulated controller is specifically designed for process industry applications, utilizing the Finite Step-Response Model (FSRM) as the system description.
- Provide functionality to analyse controller performance.

- Emphasise run-time and a memory-efficient C/C++ implementation.
- High-quality open-source code following principles of software development, i.e. modularity, readability, testability, etc.
- Easily portable to web applications.

With an aim to make the code more user-friendly, a web application integrating the simulator shall be developed. This frontend will help automate the tuning and testing of the Model Predictive Controller providing a tailored interface. The web application shall fulfil the following criteria:

- Emphasise user-interface (UI), user-experience (UX) and web performance.
- Being available on different devices supporting multiple web browsers.
- Provide functionality to analyse controller performance.
- Being an industrial and academic framework for tuning and testing MPC.

As stated in 1.1, MPC is a complex controller being unrecognized by many industries. Therefore, to expose the inner workings of the algorithm, the application layout will include informative pages explaining the controller principle and tuning mechanisms. Thus, the application will target both industrial usages as well as being an academic framework for linear Model Predictive Controller.

This master thesis continues the specialization project [29]. During the pre-project, the theory behind and implementation of a minimal lightweight MPC was derived along with corresponding testing routines and simulation results. However, in this thesis, the theory and implementations are further improved and extended. Additionally, while the pre-project concerned itself with the simulator, this thesis focuses on the web interface and simulator applications to a greater extent.

Due to the clear dissection between the simulator and the web application, both in terms of functionality and implementation language, it was sensible to divide the software into separate repositories. The repositories are named *Web-application* and *MPC-simulator*. The combination of these code bases is hereby referred to as *Light-Weight MPC*.

1.3 Contribution

The main contribution of this thesis is the lightweight implementation of an MPC simulation software. The controller is developed based on contemporary theory and modern software libraries aiming to emphasise run-time complexity and memory management. The corresponding web application automates the sequence of producing simulation data. Some highlights are:

- Designing a Unix-based toolchain using GNU compiler collection (GCC) and GNU Make to deploy C/C++ source code. The toolchain uses *Conda* as package and environment manager.

- Deriving and implementing an MPC with and without slack variables using FSRM as control model and *Operating Splitting Quadratic Solver* (OSQP) as optimizer.
- Using Webassembly to port C/C++ code to a web application such as the native-run-time is preserved.
- Developing a portable *React* web application providing a tailored interface to the simulator.

1.4 Outline

The master thesis is structured in the following manner aiming to describe the design choices and the implementation of the simulation software. After the introductory section, presenting the background and project description, the theory behind an MPC is outlined. This theory is sufficient in order to understand the MPC principle and the impact the tuning variables have on the controller performance. Additionally, a background is needed in order to properly utilize the simulation software. The theory section also covers the FSRM, which serves as the main model description. With this foundation established, the Model Predictive Controller for Finite Step-Response Model (MPC-FSRM) is derived. The theoretical background is built upon the foundational theories elucidated in *High-performance Industrial Embedded Model Predictive Control* [12] and *Process Dynamics and Control* [25].

Following the theory, the implementation of the simulator and the web application is described. Starting by describing the functionality of the simulator, a sensible toolchain is designed and implemented making the simulator portable to Unix devices. Moreover, input-output (IO) data formats are outlined to pass sufficient data to the simulator and recall them for further analysis. The relevant implementation languages and relevant libraries used are also addressed. After defining the objectives of the web application, the text elaborates on how to combine the two different repositories in a simulation pipeline using Webassembly. The frontend design regarding the UI and the UX is discussed in relation to the application objectives.

After having elaborated on the methodology, some simulation results are presented to substantiate the application objectives stated in Section 1.2. In order to test the software, dynamic system descriptions are needed. For this purpose two FSRMs with relevance to the process industry are described. These models serve to be a typical use case for an end user, pursuing to control these plants using MPC. The results reveal how to utilize the simulation software and its many functionalities to obtain the best pre-eminent MPC-FSRM. Subsequent to the results, a discussion follows where different software aspects are brought to light. Central to this section is to address how the implemented functionality aligns with the problem description. The conclusion points out possible ways to bring the application further on the path to emerge from the lightweight category.

2 Theoretical background

The theoretical section of this thesis is a continuation of the theory presented in the corresponding project thesis [29]. Hence, some of the sections might appear similar or even identical. The theory chapter is divided into three sections: Finite Step-Response Model, Model Predictive Control and Model Predictive Control for Finite Step-Response Model. The first two sections were also covered in the pre-project, hence they resemble. However, in instances requiring clarification or the inclusion of missing details, certain sections were appended. The controller presented in Section 2.3 is an extended version of the one derived in the pre-project.

2.1 Finite Step-Response Model

An MPC algorithm is dependent on a control model formulation of the dynamical system. In general, a plant can be defined by multiple inputs and outputs yielding a Multiple Input Multiple Output (MIMO) system description. However, for simpler model representations, the input and output can appear singular. This class of systems are characterized as a Single Input Single Output (SISO) system. The MPC formulation is not limited to one system description, but the problem formulation is extended with the complexity of the model, yielding ambitious computation and implementation. Therefore, in the choice of a model for MPC control, one needs to consider model complexity. This section will study an important model in process control, the FSRM.

The Finite Step-Response Model is an advantageous representation of stable linear systems. The model representation is rapidly used within the process industry, due to the fact that many process plants are linear and open-loop stable. In order to describe the system dynamics, step-response coefficients s_i are utilized. These describe the output value of a linear system when a unit step response is applied. By having this system representation, the response can be singly described by a vector of $s_1, s_2, \ldots, s_N$. The number of step-response coefficients, denoted as the model horizon N, is chosen large enough in order to accurately describe the exponential response of a stable linear system. Due to the stability property, all step-response models feature $s_{N+1} \approx s_N$.

2.1.1 Single Input Single Output system description

The formulations of the FSRM are taken from the book *Process Dynamics and Control* (Chapter 20 [25]). This section, describing the model, denotes a lowercase mathematical symbol as a single value. Uppercase and bold uppercase notation denote respectively a vector and a matrix expression. These different notations make up the nomenclature. The output y is dependent on two terms, a summation of actuation changes and an offset. The

SISO representation of an FSRM at a time step k can be expressed as,

$$\begin{aligned} y(k) &= \sum_{i=1}^{N-1} s_i \Delta u(k-i) + s_N u(k-N) \\ &= \begin{bmatrix} s_1 & s_2 & \cdots & s_N \end{bmatrix} \begin{bmatrix} \Delta u_1 \\ \Delta u_2 \\ \vdots \\ \Delta u_{N-1} \end{bmatrix} + s_N u_N \\ &= f(S, \Delta U, u) \end{aligned} \tag{1}$$

In contrast to other system models, the step response formulation is not based on states but describes the dynamic of the systems by linear combining step coefficients with the corresponding change in actuation. Due to the usage of actuation steps and not the current actuation, an $N-1$ element large vector of actuation steps needs to be monitored throughout the simulation of an FSRM. Along with the final actuation, $u(k-N)$, these values represent the internal state of the system. The final actuation is also only a system parameter, and shall not be confused with the applied actuation, u, which is expressed as $u = \sum_{i=1}^{N-1} \Delta u(k-i) + u(k-N)$. Notice that $\Delta u(k)$ denotes the first element in the vector and that the iteration goes backwards in time. Hence, the previous actuation step is multiplied by the first step-response coefficient added by the later step with a later coefficient and so on.

Assume a step in actuation, Δu_* occurring at time k. The corresponding model output is then described by right-shifting the step into the actuation vector. The past actuation step, Δu_{N-1}, is then disregarded from the fixed-sized actuation vector, but rather included in the updated final actuation, $u_N = u_N + u_{N-1}$. To conclude, the effect of an actuation step is represented by right-shifting the actuation vector and updating the offset. This is one of the key properties of using the FSRM for MPC, as calculating predictions is a fairly simple operation.

$$\begin{aligned} y(k+1) &= \sum_{i=1}^{N-1} s_i \Delta u(k-i+1) + s_N u(k-N+1) \\ &= \begin{bmatrix} s_1 & s_2 & \cdots & s_N \end{bmatrix} \begin{bmatrix} \Delta u_* \\ \Delta u_1 \\ \vdots \\ \Delta u_{N-2} \end{bmatrix} + s_N (u_N + u_{N-1}) \end{aligned} \tag{2}$$

The expression (1) is a discrete-time system representation, where k and i are respectively the current and fixed time step. By denoting j as a future time step, the output $\hat{y}(k+j)$ describes a predicted output j time steps forward in time. The general SISO predicted output equation is formulated by splitting up the different step responses in future and

past time intervals, respectively for $i \in [0, j]$ and $i \in [j+1, N]$, yielding,

$$\begin{aligned}\hat{y}(k+j) = & \underbrace{\sum_{i=1}^{j} s_i \Delta u(k+j-i)}_{\text{Effect of future control actions}} \\ & + \underbrace{\sum_{i=j+1}^{N-1} s_i \Delta u(k+j-i) + s_N u(k+j-N)}_{\text{Effect of past control actions}},\end{aligned} \tag{3}$$

where the past term is reformulated as,

$$\hat{y}^o(k+j) \triangleq \sum_{i=j+1}^{N-1} s_i \Delta u(k+j-i) + s_N u(k+j-N), \tag{4}$$

yielding the SISO predicted output equation:

$$\hat{y}(k+j) = \sum_{i=1}^{j} s_i \Delta u(k+j-i) + \hat{y}^o(k+j). \tag{5}$$

2.1.2 Model prediction

Given the system description, one can determine up to N model predictions simply from a linear equation. For an arbitrary prediction P steps into the future, the P first step-response coefficients are used to estimate the output. The remaining $N-1-P$ step coefficients are used to represent the effect of already chosen Δu. Analogously, this procedure can be seen as P right-shift operations of the actuation vector. By expanding these expressions one can formulate a vector of P predictions, $\hat{Y}$. Similarly, this can be done for the past outputs and actuation steps by also taking the control horizon M into account, yielding the expressions (6).

$$\hat{Y}(k+P) \triangleq \begin{bmatrix}\hat{y}(k+1) & \hat{y}(k+2) & \dots & \hat{y}(k+P)\end{bmatrix}^T \in \mathbb{R}^P \tag{6a}$$

$$\hat{Y}^o(k+P) \triangleq \begin{bmatrix}\hat{y}^o(k+1) & \hat{y}^o(k+2) & \dots & \hat{y}^o(k+P)\end{bmatrix}^T \in \mathbb{R}^P \tag{6b}$$

$$\Delta U(k+(M-1)) \triangleq \begin{bmatrix}\Delta u(k) & \Delta u(k+1) & \dots & \Delta u(k+M-1)\end{bmatrix}^T \in \mathbb{R}^M \tag{6c}$$

Due to the linear form of the FSRM, one can express P model predictions ahead using M control inputs. This mapping from control inputs to predictions is achieved by defining the lower triangular matrix $\boldsymbol{S} \in \mathbb{R}^{P \times M}$.

$$\hat{Y}(k+P) = \boldsymbol{S}\Delta U(k+M) + \hat{Y}^o(k+P), \tag{7a}$$

$$\boldsymbol{S} \triangleq \begin{bmatrix} s_1 & 0 & \cdots & 0 \\ s_2 & s_1 & 0 & \vdots \\ \vdots & \vdots & \ddots & 0 \\ s_M & s_{M-1} & \cdots & s_1 \\ s_{M+1} & s_M & \cdots & s_2 \\ \vdots & \vdots & \ddots & \vdots \\ s_P & s_{P-1} & \cdots & s_{P-M} \end{bmatrix}. \tag{7b}$$

2.1.3 Multiple Input Multiple Output system description

The MIMO system description is obtained using the superposition principle. A MIMO system output is formulated as a linear combination of several SISO systems, hence the MIMO representation is a straightforward extension of the system (7). For an industrial process plant, the MIMO system description is governed by multiple outputs Y and inputs U. In this case, the response of an output variable can be caused by multiple input variables. Following the superposition principle, the total response is equivalent to the summation of the sub-responses.

Whenever MPC is applied to a model representation, another methodology is used to describe the model variables. The term controlled variable (CV) describe a variable to be controlled, yielding an output variable. Such variables often have a set point to target and the dynamics are governed by soft constraints. Similarly, a manipulated variable (MV) refers to a variable the controller can determine, yielding an input variable. In contrast to the controlled variables these are governed by hard constraints, constraints which cannot be stepped over. This methodology, renaming the model's inputs and outputs, is typical for industrial applications. Hence, the use of MV and CVs is naturally included in the implementation of *Light-Weight MPC* .

Assume $n_{CV} = 1$ and $n_{MV} = 2$, referring to the number of respectively controlled and manipulated variables. The output, $\hat{Y}(k+P)$, is composed of two SISO FSRMs governed by Equation (7). As seen in Equation (8), knowledge of both systems is required, implying having determined the step-response coefficients for both sub-systems. In the general case with n_{CV} controlled variable and n_{MV} manipulated variables, the overall system description can be described in a linear form (9a). Extending the MIMO matrix definitions is formed by stacking the SISO components upon each other. For instance, one element of the MIMO $\boldsymbol{\Theta}$-matrix (9b) describes one SISO response, $\boldsymbol{S}$-matrix defined in Equation (7b).

$$\begin{aligned} \hat{Y}(k+P) &= \boldsymbol{S}_{11}\Delta U_1(k+M) + \hat{Y}^o_{11}(k+P) \\ &\quad + \boldsymbol{S}_{12}\Delta U_2(k+M) + \hat{Y}^o_{12}(k+P) \\ &= \boldsymbol{\Theta}\Delta U(k+M) + \hat{Y}^o(k+P) \end{aligned} \tag{8}$$

$$\hat{\boldsymbol{Y}}(k+P) = \boldsymbol{\Theta}\Delta\boldsymbol{U}(k+M-1) + \hat{Y}^o(k+P), \tag{9a}$$

$$\text{where} \quad \boldsymbol{\Theta} \triangleq \begin{bmatrix} \boldsymbol{S}_{11} & \boldsymbol{S}_{12} & \cdots & \boldsymbol{S}_{1n_{MV}} \\ \boldsymbol{S}_{21} & \cdots & \cdots & \boldsymbol{S}_{2n_{MV}} \\ \vdots & \vdots & \vdots & \vdots \\ \boldsymbol{S}_{n_{CV}1} & \cdots & \cdots & \boldsymbol{S}_{n_{CV}n_{MV}} \end{bmatrix} \in \mathbb{R}^{n_{CV}\cdot P \times n_{MV}\cdot M}, \tag{9b}$$

$$\hat{Y}^o(k+P) \triangleq \begin{bmatrix} \hat{Y}^{\boldsymbol{o}}_{11} & \hat{Y}^{\boldsymbol{o}}_{12} & \cdots & \hat{Y}^{\boldsymbol{o}}_{1n_{MV}} & \cdots & \hat{Y}^{\boldsymbol{o}}_{n_{CV}1} & \cdots & \cdots & \hat{Y}^{\boldsymbol{o}}_{n_{CV}n_{MV}} \end{bmatrix}^T, \tag{9c}$$

$$\Delta U(k+M-1) \triangleq \begin{bmatrix} \Delta U_{11}(k+M-1) \\ \Delta U_{12}(k+M-1) \\ \vdots \\ \Delta U_{1n_{MV}}(k+M-1) \\ \vdots \\ \Delta U_{n_{CV}n_{MV}}(k+M-1) \end{bmatrix}. \tag{9d}$$

2.1.4 Simulating a Finite Step-Response Model

When digitally simulating an FSRM, time must be discretized into smaller steps. The sampling period, Δt, denoting one discrete time step is firmly related to the model horizon N. The model horizon should be chosen so that

$$N = \frac{t_s}{\Delta t}, \tag{10}$$

where t_s is the settling time for the open-loop response [25]. This value describes the time needed for the process to reach a steady state when a step response is applied. Most important, when deriving an FSRM, is to use the same Δt for all input-output responses. Only by enforcing this property, a valid simulation model can be acquired. Otherwise, the controller actuates the same controlled variable using different time stamps. Assuming a MIMO FSRM description and a constant Δt, the model horizons may differ due to varying settling times. Typically, $N \in [30, 120]$ according to [25]. The simulator implemented solves the problem by enforcing an equal number of step response coefficients. By defining N_* as the highest number of coefficients present in the model. The simulator utilizes the property, $s_{N+1} \approx s_N$, to pad every smaller step-response coefficient vector to the size N_*.

Since the notion of time is abstracted into the number of coefficients, it does not make much sense to produce a simulation for an MPC horizon in a unit of seconds. Therefore the MPC horizon, T is a unitless variable describing the total number of steps used in a simulation. One specific step is denoted k such that a whole simulation can be described as $k \in [0, \ldots, T]$. The impact of one simulation step, $k \to k+1$, represents the Δt used when deriving the FSRM.

2.2 Model Predictive Control

Model Predictive Control is a state-of-the-art control technique, utilizing a model and optimization theory to predict and determine the optimal control actuation. By having the system model present in the controller, the algorithm can take multiple different dynamics into account to achieve better performance. Hence the controller method has most applications facing difficult multi-variable system plants. Additionally, as constraints are a natural part of an optimization problem, system constraints can easily be accounted for in the control loop. The objective of the controller can vary and is defined by the optimization problem. However, a typical MPC aims to stabilize the model output to a set point or make them follow a trajectory.

2.2.1 Classical versus predictive control

Classical controllers such as PID control are the most commonly adopted structure in the industry. Their success is linked to the three parameters K_p, K_I and K_d, as seen in Equation (11), being intuitive and simple to tune in many cases. The PID controller is a computationally fast controller, assessing the control error in the current time stamp t.

$$K(t) = K_p e(t) + K_I \int_{t_0}^{t} e(\tau) d\tau + K_d \dot{e}(t) \tag{11}$$

It is shown (e.g., see [23] Chapter 1) that PID control is challenging to implement for several system classes. Furthermore, it is also difficult to achieve a good performance using a PID controller if the system is prone to large time delays or has a non-minimum phase zero. It is not surprising that the scope of a PID controller is limited as the model is too simplistic to cater for the challenging dynamics of some system descriptions. Predictive controllers, on the other hand, have a more complex structure involving multiple controller parameters. An MPC is a more flexible controller and a better choice for controlling multi-variable systems and heavily constrained systems.

Nevertheless, many of these systems can maintain high-quality control performance by combining PID control with a human operator. So what aspects does a human account for which cannot be perceived by classical controllers?

The true value the predictive controllers lies in their ability to anticipate the future. This is in fact the natural approach humans use to control their surroundings. Consider the process of driving a car. A good driver has a lifted head and eyes glancing several meters ahead. This is in order to anticipate any potential dangers, such that he can account for these at an early stage by reducing the driving velocity or positioning himself accordingly. In other words, the driver uses predictions in order to achieve a reasonable control law, and the car moves forward as the road within our vision moves. In general, how far the prediction goes is denoted by the prediction horizon and this horizon recedes in time. Figure 1 illustrates how the receding horizon moves by the time step k and prediction horizon P. As the horizon moves in time, the operator picks up new information used to update the control action. This is the core principle of a predictive controller.

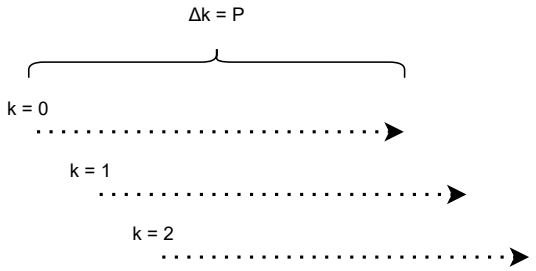


Figure 1: Illustration of the receding horizon. Even though the process changes with time, the prediction horizon is held constant, predicting new information for each step k.

Consider again the car driving process but without a human operator. The car is driving down an alley in order to reach its destination, the system needs to turn by the next intersection. A predictive controller would plan the scenario by reducing the vehicle speed before the act of turning. Hence, maintaining the reference within the constraints. This controller performance is given due to predictions far enough ahead to large enough to anticipate the turn. If any other aspects such as traffic lights and pedestrians could be perceived, this could also be accounted for in the control action.

This example clearly shows the use of predictive controllers. However, in order to describe the predictions, additional models must be accounted for in the controller algorithm leading to higher run-time complexity. In general, the optimal controller can be obtained by finding the sweet spot between performance and complexity. Often simple fast systems will benefit from classical control approaches, while slow complex system description rather is the typical use case for predictive controllers.

2.2.2 Optimization theory

In order to understand the linear MPC algorithm, a background in optimization theory is required. The following sections provide a grasp overview of the basic principle behind the controller. Mathematical optimization is the theory behind finding a set of optimum values, $z \in \mathbb{R}^n$, for a problem given a set of constraints. The definition of a general constrained optimization problem is formulated as (cited from [6])

$$\min_{z \in \mathbb{R}^n} f(z), \tag{12a}$$

$$\text{subject to} \quad c_i(z) = 0, \quad i \in \mathcal{E}, \tag{12b}$$

$$c_i(z) \geq 0, \quad i \in \mathcal{I}, \tag{12c}$$

where f is a scalar objective function describing the properties we want to optimize. $\mathcal{E}$ and $\mathcal{I}$ are the disjunct index sets respectively to the equality and inequality constraints. Based on the constraints the feasible set $\mathcal{X}$ is denoted as

$$\mathcal{X} = \{z \in \mathbb{R}^n \mid (c_i(z) = 0, i \in \mathcal{E}) \wedge (c_i(z) \geq 0, i \in \mathcal{I})\}, \tag{13}$$

The solution to the optimization problem is a $z^* \in \mathcal{X}$ minimizing the objective function. Hence, $f(z^*) < f(z), \quad \forall z \neq z^* \in \mathcal{X}$. There exist a diversity of approaches in order to determine z^*. A suitable optimizer depends on the specific problem and along with other factors such as memory requirements and speed of convergence. The problem formulation is critical when using optimization algorithms. The problem should be formulated in a way that is well-suited to the optimizer being used and the characteristics of the problem being solved. A well-formulated problem can greatly improve the performance of the optimizer and the quality of the solution obtained.

2.2.3 Standard Quadratic Program formulation

A Quadratic Program (QP) arises frequently in optimization-based control. The objective is to find a control signal that minimizes a quadratic cost subject to constraints on the control inputs and state variables. The benefit of such a formulation is that the convexity of the problem is easily characterized. The structure of such a QP is defined as [6]:

$$\min_{z \in \mathbb{R}^n} \frac{1}{2} z^T \boldsymbol{G} z + q^T z, \quad \boldsymbol{G} \in \mathbb{R}^{n \times n}, q \in \mathbb{R}^n \tag{14a}$$

$$\text{subject to} \quad c_i(z) = a_i^T - b_i = 0, \quad i \in \mathcal{E} \tag{14b}$$

$$c_i(z) = a_i^T - b_i \geq 0. \quad i \in \mathcal{I} \tag{14c}$$

The standard quadratic program is characterised by a quadratic cost function constrained by affine elements. If the hessian matrix, $\boldsymbol{G}$, is a positive definite matrix, yielding, $\boldsymbol{G} = \boldsymbol{G}^T \succ 0$ the problem has convex properties. The importance of convexity in optimization problems stems from the fact that convex functions and sets have desirable properties that make optimization easier. It can be shown that convexity implies a uniquely global solution for the optimization problem (14). Such a global minimum in the solution space allows solvers to converge to the optimal solution efficiently since no local minima exist in the solution space. In the case when the hessian matrix is positive semi-definite, yielding $\boldsymbol{G} = \boldsymbol{G}^T \succeq 0$, the optimization problem is still relatively easy to obtain a solution. However, this solution might not be a global solution due to the existence of local minima in the solution space.

2.2.4 Minimal controller formulation

Assume a controller, assessing a SISO plant with manipulated variable Δu and controlled variable y for the time horizon $k = [0, \cdots, j, \cdots, T]$. Linear MPC is formulated as a convex optimization problem minimizing an objective function:

$$\min \sum_{j=W+1}^{P} \|y(k+j) - y_{ref}(k+j)\|_{\mathbf{Q}}^2 + \sum_{j=0}^{M-1} \|\Delta u(k+j)\|_{\mathbf{R}}^2, \tag{15}$$

such that

$$y(k+j) = f(\hat{p}, \Delta u(k+j)) \tag{16}$$

where $\sum_0^I \|y\|_{\mathbf{Q}}^2 = y^T \mathbf{Q} y \in \mathbb{R}, \mathbf{Q} \in \mathbb{R}^{I \times I}$ indicates a scaled L2-norm where each expression is scaled by the elements q_i of the positive definite matrix Q. By utilizing the L2-norm the cost function is assumed to be convex. Additionally, by assuming a linear model description (16), the overall problem is convex using the same analogy as in Section 2.2.3. Furthermore, taking additional linear constraints into account will not change the overall convexity of the quadratic problem.

This is a linear MPC due to the linear property of the dynamical model. Furthermore, this problem can be solved using quadratic optimizers. However, if the plant is nonlinear modelled (16) the optimization problem becomes more challenging to solve. This complicates the solution procedure significantly and a nonlinear solver is needed instead of a quadratic. Such a controller formulation is denoted as a nonlinear MPC and due to its increased complexity, this type of problem will not be further addressed in this thesis. For further reading $f(\hat{p}, \Delta u(k+j))$ is assumed to be linear.

The SISO formulation, Equation (15) and (16), can easily be extended to a MIMO system description. This is done by representing y and Δu in vector cases with the respective lengths of n_{CV} and n_{MV}. By rewriting the summations as vector-matrix multiplications extracting the optimization variables in a vector z, the optimization problem takes the form of (14). The model variable $\hat{p}$ depends on the specific system model representation. If $f(\ldots, \Delta u)$ is represented by a step-response model this variable may consist of measured outputs and previous control inputs.

One question remains. How can the controller be interpreted? The first summation evaluated a controlled variable error, between the model output and the given reference y_{ref}. This reference could either be a constant term or a trajectory to be followed. The performance tuning is represented by the matrices $\mathbf{Q} = \mathbf{Q^T} \succeq 0$ and $\mathbf{R} = \mathbf{R^T} \succeq 0$. The $\mathbf{Q}$ matrix is a tuning matrix that penalizes the controlled variable error. By increasing the q-weights the controller will emphasize tracking the reference signal. Similarly, the later summation assesses the change in actuation penalised by the $\mathbf{R}$-matrix. By tuning these values higher, the controller emphasises a small change in actuation. Additional signals can be taken into account simply by adding an L2-norm expression. The general tuning problem is not that challenging for an MPC controller, since every tuning factor penalises a certain signal.

2.2.5 Model Predictive Control principle

As elaborated in Section 2.2.1, the basic controller principle builds upon future model predictions. Given models, the controller can predict the controlled variables and use this information further to plan an optimal sequence of manipulated variables. Note that the horizons P, M, $T > 0$ and $W \geq 0$ denote scalars even tho they are written in uppercase. For each time step k, a tuned optimization problem is solved for a prediction horizon, P, and a control horizon M. This notion is also called dynamical optimization, solving a set of different optimization problems with the same underlying structure for every time

stamp. The different horizons specify how many time steps into the future the controller takes into account when optimizing. Particularly, the control horizon M refers to the number of future control actions that the MPC plans ahead. It determines the length of the control sequence considered when optimizing. On the other hand, the prediction horizon, P, specifies the number of future time steps over which the system behaviour is predicted. It determines how far into the future the MPC controller looks when making control decisions. After each optimization, the first predicted actuation, $u(k+1)$ is fed back into the plant, proceeded by estimating the system plant using this information to further predict the next optimal action. Algorithm 1 describes the MPC procedure using output feedback.

How do the controller parameters affect the performance? Consider the control and prediction horizons, M and P. As the control horizon increases, the MPC controller tends to become more aggressive and the required computational effort increases. M is often selected such that $M \leq P$. A rule of thumb according to [25] is $\frac{N}{3} < M < \frac{N}{2}$, and different values of M can be tied to every MV. The same case goes for the prediction horizon P if the settling times, t_s are quite different. However, due to simplicity, M and P are assumed equal for every CV and MV response present in a MIMO FSRM. This assumption is also done in thesis [12], but can easily be extended by parsing multiple horizons. if implemented, these additional can be reflected in the definition of the $\mathbf{\Theta}$-matrix yielding the dimensions $\mathbb{R}^{\sum_{i=0}^{n_{CV}} P_i \times \sum_{j=0}^{n_{MV}} M_j}$. The prediction horizon is often selected to be $P \leq N+M$ so that the full effect of the last MV move is taken into account. A decrease in this value tends to make the controller more aggressive. To avoid an aggressive response one can also consider the prediction horizon infinite, $P \to \infty$. It can be shown that this controller assembles a Linear Quadratic Regulator (LQR) if additional constraints are disregarded. Such a controller definition will however not be regarded in this project.

For process plants having a delayed response time, it might be reasonable to account for this delay in the controller definition. The start horizon, W, describes the first predicted output deviation to account for in the cost function. Especially, when regarding industrial process plants, such systems do usually have slow dynamics with a certain time delay, τ between control input and system output. Due to this system property, it would be dubious to try to control the plant for $t \leq \tau$ corresponding to $k \leq W$ in relation to the MPC horizon. Only taking predictions $k \geq W$ into account will also contribute to reducing the computational effort, since fewer terms are considered.

2.2.6 Implementing a Model Predictive Controller

Having covered the control principle and the model, this section will elaborate on how a controller simulation can be implemented. Figure 3 illustrates how the controller is related to the plant, which is being controlled. The control method is illustrated in the blue marked area, where an optimizer and a control model are used to calculate the optimal actuation. For a simulation case, there is no physical plant to apply the actuation on, hence both the control model and plant must be mathematically described. Considering controlling an FSRM plant, the control model will resemble the plant model exactly. This is not the case for real-time control, as the control model only is an estimated model, and model errors are present. For every time step k, the controller solves an optimization problem where

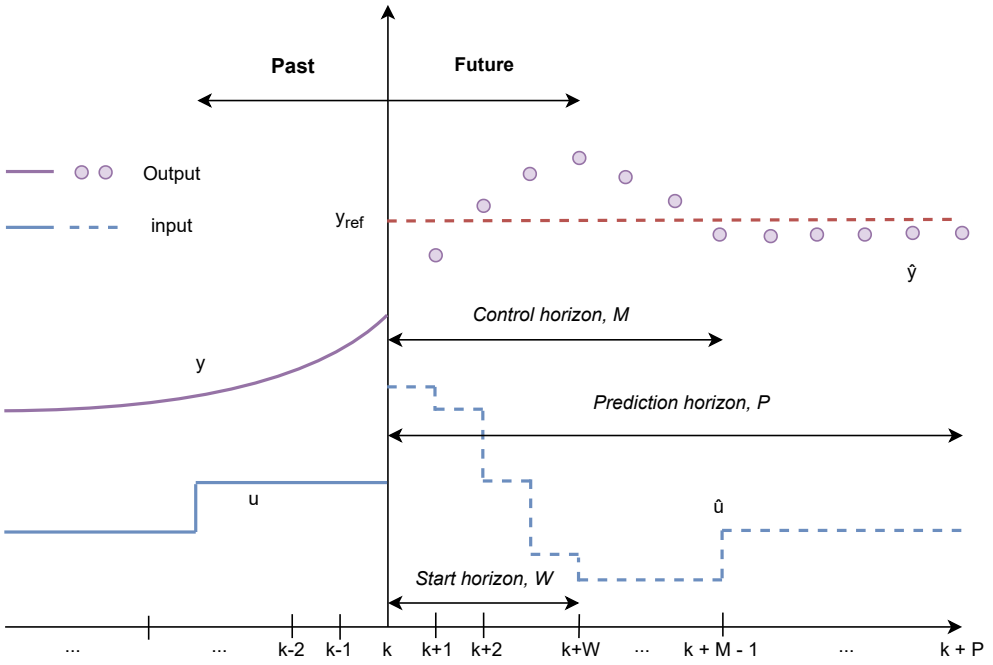


Figure 2: Illustration of a Model Predictive Control scenario, inspired by figure 20.2 [25]. This scenario exemplifies the impact the horizons have on the predicted signals.

the control model is used as an equality constraint. Algorithm 1 describes the simulation procedure used in *MPC-simulator*.

Algorithm 1 Output feedback MPC procedure [6]

for $k = 0, \ldots, T$ **do**
  Compute an estimate of the current output $\hat{y}_k$ based on the measured data up until time k.
  Solve a dynamic optimization problem on the prediction horizon P from k to $k + P$ using M actuation predictions with x_k as initial condition.
  Apply the first control move u_k from the solution above.
end for

MPC-simulator aims to implement the core functionality of an MPC. Of course, the basic principle enters into this purpose. However, there are other features often found in the realm of predictive control. These features contribute to improving the flexibility of the controller and are also addressed in the lightweight implementation.

2.2.7 Soft constraints, slack variables

According to [25], applying inequality constraints on the input and output variables was the primary motivation for the early development of MPC. This property could for instance non-saturating actuators or constraining process outputs such as temperature, flow, pressure and volume seen from a process industry view. However, inequality constraints might lead to feasibility issues for a solver algorithm.

For practical applications, a small feasible set, described in Equation (13), might not be

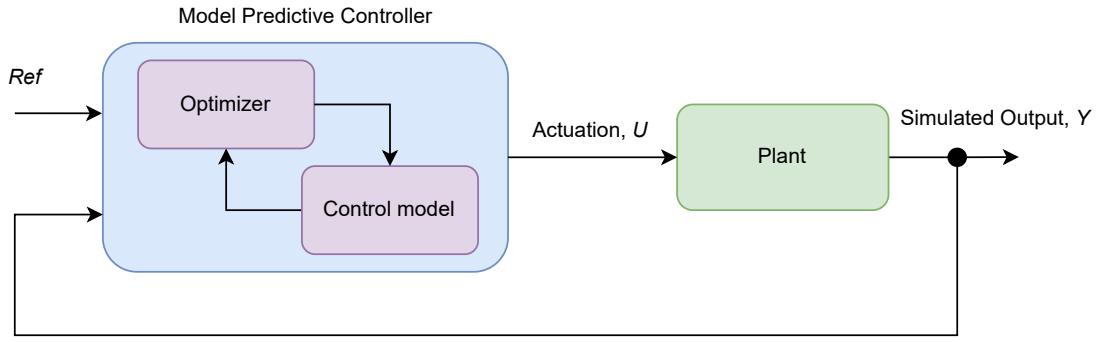


Figure 3: General Model Predictive simulation loop. For each simulation step, the output of the plant is feedback to the controller optimizing the next actuation with the use of the prediction model and reference.

sufficiently feasible for stabilizing controller feedback. In the case of controlling physical plants, model errors and disturbances are always present. These phenomena may break any of the inequality constraints, such that no feasible solution is obtainable for the controller in order to stabilize the plant. By introducing soft constraints in the objective function, the inequality constraints are softened, increasing the set of feasible solutions (13). This approach is beneficial in the case where disturbances or unforeseen phenomena are impacting the plant. Soft constraint introduced in the cost function emphasises robustness in the controller.

Soft constraints, also known as slack variables, are implemented by adding and subtracting an offset, ϵ, to respectively the upper and lower inequality constraint for the output variable. As indicated in Figure 3, the output is denoted as Y. Equation (17) shows how slack variables can be mathematically interpreted. This offset is further penalized in the cost function, in which the offset is minimized. For the presents of multiple slack variables, the cost takes a linear form, $\rho^T \epsilon$. The offset is also required to be non-negative allowing only an increase of the feasible set. To enforce this property, additional inequality constraints for the offset are added to the optimization problem. The tuning of the soft constraint represents the hardness of the constraint. By increasing ρ, the constraint is hardened. Conversely, softening occur through a reduction. Soft constraints are typically employed only on the controlled variables.

$$\underline{Y} - \epsilon_l \leq Y \leq \overline{Y} + \epsilon_h, \quad \epsilon_h \geq 0, \quad \epsilon_l \geq 0. \tag{17}$$

2.2.8 Bias correction

Due to inaccuracy and unknown disturbances, the predictions used in the control algorithms might be unrealistic and error-prone. Prediction accuracy can be improved using the latest measurements in the predictions. The idea is to define a bias correction defined as in Equation (18). This bias is an estimate of the control model error and can further be added to the predicted output yielding a corrected prediction (19). The measured bias $b(k)$ measured in the current time step k is added to every future prediction. Hence, the method can be seen as an update in the model controller offset.

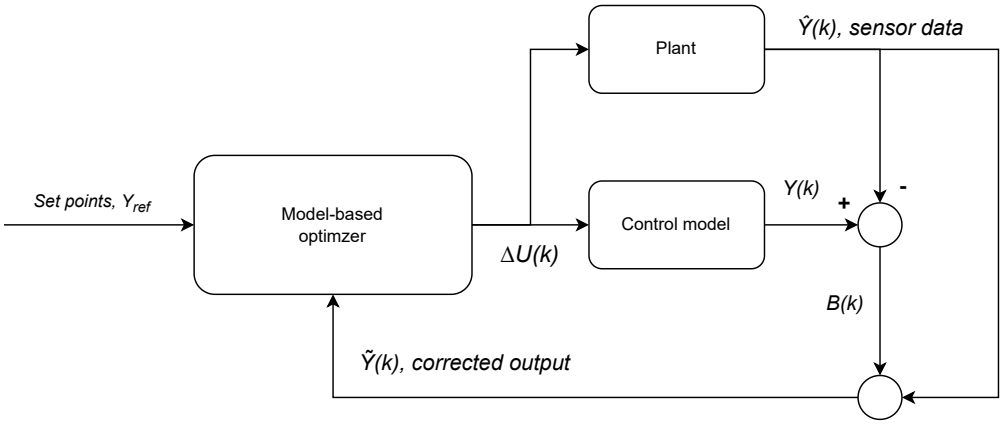


Figure 4: Block diagram for model predictive control with output feedback. The real-world deceptive model information is used to correct the control model in the feedback loop.

$$b(k) = y(k) - \hat{y}(k) \tag{18}$$

$$\begin{aligned} \tilde{y}(k+j) &\triangleq \hat{y}(k+j) + b(k) \\ &= \hat{y}(k+j) + [y(k) - \hat{y}(k)] \end{aligned} \tag{19}$$

This strategy is known as output feedback or bias correction, utilizing measurements in the control loop. The expressions are taken from [25]. The corresponding block diagram for an output feedback MPC controller is shown in Figure 4. The bias correction term is called residual and is seen in most outer feedback. In order to implement this control loop software-wise, a mathematical model assembling a real-world plant needs to be simulated. A typical control challenge when controlling an actual plant is the impact the process noise has on the control loop. Hence, in order to make the control loop as realistic as possible in a simulation environment, noise should be present in the plant model simulations. By this means, the model simulation assembles real sensor data.

Having such a plant model present, in addition to the derived control model, one can also use the residual, $B(k)$, as an estimate of how well the control model assembles the underlying plant. If this deviation is too great, one might need to reconsider the control model. Usually, the control model is a linearized model, yielding a simpler model description. The linearization approach is often made in order to analytically derive control laws. The *MPC-simulator* assumes the Finite Step Response Model description as the control and prediction model. However, as described in Section 2.1 about the model, it can only represent linear stable plants. In the case where the plant deviates too far from this assumption, the controller will perform poorly due to the overwhelming model errors present in the control loop.

2.2.9 Model Predictive Controller formulation

Taking all the MPC features mentioned into account, one can represent the MPC controller by the optimization problem described by the Equations (21). The problem formulation takes a MIMO system description into account, hence every value that occurred in the SISO formulation (15) is now represented as a vector. The vector nomenclature is described by capitalized letters.
There are in total six tuning variables making up the prediction mechanism and controller performance. The horizons: P, M and W and penalizing variables $\mathbf{Q}, \mathbf{R}$ and ρ. The closed-loop stability of MPC can be acquired by the choice of $\mathbf{Q}, \mathbf{R}, P$ and W. A rule of thumb is to set the prediction horizon large enough to cover the dominant dynamics. By predicting far ahead, the algorithm may overcome any future instability by depicting early deviations. Achieving closed-loop stability is in practice no major issue giving a sound MPC tuning. Especially when all dynamical models addressed in *Light-weight MPC* are open-loop stable, however, as addressed in Section 2.2.8 model errors are unavoidable, and stability should always be taken into account when tuning. To summarise, the selection of appropriate values for the six tuning variables depends on the specific system dynamics, control objectives, and computational constraints. It often requires a combination of theoretical analysis, simulation studies, and empirical tuning to find optimal values that result in satisfactory control performance for a given application.

$$\min \sum_{j=W+1}^{P} \left\| \tilde{Y}(k+j \mid k) - Y_{ref}(k+j) \right\|_{\mathbf{Q}}^2 + \sum_{j=0}^{M-1} \|\Delta U(k+j)\|_{\mathbf{R}}^2 + \rho^T \epsilon, \tag{20a}$$

$$\text{where} \quad \rho = \begin{bmatrix} \rho_h \\ \rho_l \end{bmatrix}, \epsilon = \begin{bmatrix} \epsilon_h \\ \epsilon_l \end{bmatrix}, \tag{20b}$$

such that

$$\tilde{Y}(k+j) = F(\hat{p}, \Delta u(k+j)) + B(k+j), \quad B(k) = Y(k) - \hat{Y}(k), \quad j \in \{W+1, \ldots, P\}, \tag{21a}$$

$$U(k+j) = U(k+j-1) + \Delta U(k+j), \quad j \in \{0, \ldots, M-1\}, \tag{21b}$$

$$\Delta \underline{U} \leq \Delta U(k+j) \leq \overline{\Delta U}, \tag{21c}$$

$$\underline{Y} - \epsilon_l \leq \tilde{Y}(k+j) \leq \overline{Y} + \epsilon_h, \quad \epsilon_h \geq 0, \quad \epsilon_l \geq 0. \tag{21d}$$

2.3 Model Predictive Control for Finite Step-Response Model

The theory behind the MPC principle and the FSRM description has been covered in the respective Section 2.2 and 2.1. What remains is how to design a suitable MPC for the purpose of controlling such a dynamic model. More importantly, how to emphasise run-time complexity in the design. Both MPC-FSRM formulations discussed in this chapter are inspired by the doctor thesis *High-performance Industrial Embedded Model Predictive Control* [12].

2.3.1 Standard Quadratic Program formulation

The aim of this section is to derive the standard QP problem formulation for a MIMO step response model, constraining the output error and the change in input. As mentioned in Section 2.2.3, when designing optimization-based controllers, the run-time depends on the optimization problem's complexity. Fast physical systems require real-time controller performance to achieve the desired dynamics. Hence, the standard formulation is often used to assure fast optimization.

In order to gain a complete understanding of the derivation, with all the different matrix expressions, it is reasonable first to show the formulation for a SISO system description, extending to a MIMO, before deriving the standard QP. For a SISO system description, the MPC problem can be formulated as follows:

$$\min \sum_{j=W+1}^{P} q_j(\tilde{y}(k+j) - y_{ref}(k+j))^2 + \sum_{j=0}^{M-1} r_j \Delta u(k+j)^2 + \rho_h \epsilon_h + \rho_l \epsilon_l \tag{22}$$

subject to

$$\tilde{y}(k+j) = y(k+j) + b(k), \quad b(k) = y(k) - \hat{y}(k), \quad j \in \{W+1, \dots, P\}, \tag{23a}$$
$$u(k+j) = u(k+j-1) + \Delta u(k+j), \quad j \in \{0, \dots, M-1\}, \tag{23b}$$
$$\Delta \underline{u} \leq \Delta u(k+j) \leq \Delta \overline{u}, \quad j \in \{0, \dots, M-1\}, \tag{23c}$$
$$\underline{u} \leq u(k+j) \leq \overline{u}, \quad j \in \{0, \dots, M-1\}, \tag{23d}$$
$$\underline{y} - \epsilon_l \leq \tilde{y}(k+j) \leq \overline{y} + \epsilon_h, \quad \epsilon_h \geq 0, \epsilon_l \geq 0, \quad j \in \{W+1, \dots, P\}. \tag{23e}$$

$\rho_h, \rho_l \geq 0$ represent the tuning of the soft constraint described in Section 2.2.7. By representing the summations in terms of dense matrices and stacking the variables into vectors, as done in Section 2.1.3, the following MPC formulation is achieved:

$$\begin{aligned} \min \tilde{Y}(k+(P-W))^T \overline{\boldsymbol{Q}} \tilde{Y}(k+(P-W)) + \Delta U(k+(M-1))^T \overline{\boldsymbol{R}} \Delta U(k+(M-1)) \\ -2\mathcal{T}(k)^T \overline{\boldsymbol{Q}} \tilde{Y}(k+(P-W)) + \rho_h^T \epsilon_h + \rho_l^T \epsilon_l, \end{aligned} \tag{24}$$

where

$$\begin{aligned}\tilde{Y}(k+(P-W)) &= \boldsymbol{\Theta}\Delta U(k+(M-1)) + \boldsymbol{\Phi}\Delta\tilde{U}(k) + \Psi\tilde{U}(k-N) + B(k),\\ &= \boldsymbol{\Theta}\Delta U(k) + \hat{\boldsymbol{Y}}^{\boldsymbol{o}}(k+P) + B(k),\\ &= \boldsymbol{\Theta}\Delta U(k) + \boldsymbol{\Lambda}(k),\end{aligned} \tag{25}$$

$$\begin{aligned}\overline{\boldsymbol{Q}} &= \text{blkdiag}\left(\mathbf{Q}, n_{CV}\right) \in \mathbb{R}^{n_{CV}\cdot(P-W)\times n_{CV}\cdot(P-W)},\\ \overline{\boldsymbol{R}} &= \text{blkdiag}\left(\boldsymbol{R}, n_{MV}\right) \in \mathbb{R}^{n_{MV}\cdot M\times n_{MV}\cdot M},\end{aligned} \tag{26}$$

such that

$$U(k) = \boldsymbol{K}^{-1}(\Gamma\tilde{U}(k-1) + \Delta U(k)), \quad B(k) = Y(k) - \hat{Y}(k), \quad \boldsymbol{K} \succ 0 \tag{27a}$$

$$\tilde{Y}(k+j) = Y(k+j) + B(k), \quad j \in \{W+1, \ldots, P\}, \tag{27b}$$

$$\Delta\underline{U} \leq \Delta U(k+j) \leq \Delta\overline{U}, \quad j \in \{0, \ldots, M-1\}, \tag{27c}$$

$$\underline{U} \leq U(k+j) \leq \overline{U}, \quad j \in \{0, \ldots, M-1\}. \tag{27d}$$

$$\underline{Y} - \epsilon_l \leq \tilde{Y}(k+j) \leq \overline{Y} + \epsilon_h, \quad \epsilon_h \geq 0, \epsilon_l \geq 0, \quad j \in \{W+1, \ldots, P\}, \tag{27e}$$

This formulation describes the model-based predictive controller for an Finite Step-Response Model. The blkdiag function appearing in Equation (26) is described in the appendix Section A. For esthetical reasons, the output reference signals Y_{ref} are described by the letter $\mathcal{T}(k)$. This notation is also more descriptive since the reference also could be given by a trajectory. The cost function consists of two quadratic expressions constraining all CVs and MVs, respectively denoted by Y and ΔU. In addition, three linear terms occur considering the trajectory tracking and the upper and lower soft constraints respectively. The tuning matrices are diagonalized by the number of CV and MV present in this representation block.

By using the same approach as in Section 2.2 the Equation (25) can be simplified using the terms of future and past control actuation. This is realized by introducing $\boldsymbol{\Lambda}$ in the equation. As seen in the Equation (25), the past term $\hat{\boldsymbol{Y}}^{\boldsymbol{o}}(k+P)$ can be reformulated by separating the past change in actuation and final actuation. This is expressed by defining the two step-response matrices $\boldsymbol{\Phi}$ and $\boldsymbol{\Psi}$. The matrices are mathematically expressed in the Equations (28) and (29). This reformulation is important in the reduction of the standard QP problem. Additionally, for a general FSR model, it is not given that every SISO model. Furthermore, Λ-matrix cannot be formed without having a defined size. Define N_* as the highest valued N considering every step response included in the model definition. Due to the structure of the Finite Step Response Model, one can pad the step response vector, defined in (1), with an arbitrary number of s_N. The padded SISO response will be mathematically equivalent to the original, but larger in terms of model parameters.

$$
\begin{aligned}
\boldsymbol{\Phi} &= \begin{bmatrix} \boldsymbol{\Phi_{1,1}} & \boldsymbol{\Phi_{1,2}} & \cdots & \boldsymbol{\Phi_{1,n_{MV}}} \\ \boldsymbol{\Phi_{2,1}} & \boldsymbol{\Phi_{2,2}} & \cdots & \boldsymbol{\Phi_{2,n_{MV}}} \\ \vdots & \vdots & \ddots & \vdots \\ \boldsymbol{\Phi_{n_{CV},1}} & \boldsymbol{\Phi_{n_{CV},2}} & \cdots & \boldsymbol{\Phi_{n_{CV},n_{MV}}} \end{bmatrix}_{n_{CV}(P-W)\times\sum_{j=1}^{n_{MV}}(N_*-W-1)} \\
\boldsymbol{\Phi_{i,j}} &= \begin{bmatrix} s_{W+1} & s_{W+2} & \cdots & s_{N-2} & s_{N-1} \\ s_{W+2} & s_{W+3} & \cdots & s_{N-1} & s_N \\ \vdots & \vdots & \vdots & \vdots & \vdots \\ s_{P+1} & s_{P+2} & \cdots & s_N & s_N \end{bmatrix}_{(P-W)\times N_*-W-1}
\end{aligned}
\tag{28}
$$

$$
\begin{aligned}
\boldsymbol{\Psi} &= \begin{bmatrix} \Psi_{1,1} & \Psi_{1,2} & \cdots & \Psi_{1,n_{MV}} \\ \Psi_{2,1} & \Psi_{2,2} & \cdots & \Psi_{2,n_{MV}} \\ \vdots & \vdots & \ddots & \vdots \\ \Psi_{n_{CV},1} & \Psi_{n_{CV},2} & \cdots & \Psi_{n_{CV},n_{MV}} \end{bmatrix}_{n_{CV}(P-W)\times n_{MV}} \\
\Psi_{i,j} &= \begin{bmatrix} s_N \\ s_N \\ \vdots \\ s_N \end{bmatrix}_{(P-W)\times 1}
\end{aligned}
\tag{29}
$$

Notice that these matrices define the predictions used in the controller's cost function. Hence all of the matrix dimensions are dependent on the horizons variables P, M and W. The same goes for the $\boldsymbol{\Theta}$-matrix described in Equation (30). By setting $W = 0$, one obtain an FSRM description suited for simulation purposes.

$$
\begin{aligned}
\boldsymbol{\Theta} &= \begin{bmatrix} \boldsymbol{S_{1,1}} & \boldsymbol{S_{1,2}} & \cdots & \boldsymbol{S_{1,n_{MV}}} \\ \boldsymbol{S_{2,1}} & \cdots & \cdots & \boldsymbol{S_{2,n_{MV}}} \\ \vdots & \vdots & \vdots & \vdots \\ \boldsymbol{S_{n_{CV},1}} & \cdots & \cdots & \boldsymbol{S_{n_{CV},n_{MV}}} \end{bmatrix}_{n_{CV}\cdot(P-W)\times n_{MV}\cdot M}, \\
\boldsymbol{S_{i,j}} &= \begin{bmatrix} s_W & s_{W-1} & \cdots & 0 \\ s_{W+1} & s_W & \ddots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ s_M & s_{M-1} & \cdots & s_1 \\ s_{M+1} & s_M & \cdots & s_2 \\ \vdots & \vdots & \ddots & \vdots \\ s_P & s_{P-1} & \cdots & s_{P-M} \end{bmatrix}_{(P-W)\times M}.
\end{aligned}
\tag{30}
$$

The decoupling of actuation in the MIMO case is described in (27a). Here the two matrices $\boldsymbol{K}$ and $\boldsymbol{\Gamma}$ take the form of Equation (31) using the block diagonal transform described in the appendix Section A. Since $\boldsymbol{K}$ is a positive definite matrix, it is invertible. It can be shown that $\boldsymbol{K^{-1}}$ takes the form of a lower triangular matrix, taking only the value 1 (33).

$$\begin{aligned}\boldsymbol{K} &= \text{blkdiag}\left(\boldsymbol{K_1}, \boldsymbol{K_2}, \ldots, \boldsymbol{K_{n_{MV}}}\right) \in \mathbb{R}^{M \cdot n_{MV} \times M \cdot n_{MV}}, \boldsymbol{K} \succ 0, \\ \boldsymbol{\Gamma} &= \text{blkdiag}\left(\Gamma_1, \Gamma_2, \ldots, \Gamma_{n_{MV}}\right) \in \mathbb{R}^{M \cdot n_{MV} \times n_{MV}},\end{aligned} \tag{31}$$

$$\boldsymbol{K_j} = \begin{bmatrix} 1 & 0 & \ldots & 0 & 0 \\ -1 & 1 & \ddots & 0 & 0 \\ 0 & -1 & \ddots & 0 & 0 \\ \vdots & \vdots & \ddots & \ddots & \vdots \\ 0 & 0 & \ldots & -1 & 1 \end{bmatrix}, \quad \Gamma_j = \begin{bmatrix} 1 \\ 0 \\ 0 \\ \vdots \\ 0 \end{bmatrix}, \tag{32}$$

$$\boldsymbol{K^{-1}} = \begin{bmatrix} 1 & 0 & \ldots & 0 & 0 \\ 1 & 1 & \ddots & 0 & 0 \\ 1 & 1 & \ddots & 0 & 0 \\ \vdots & \vdots & \ddots & \ddots & \vdots \\ 1 & 1 & \ldots & 1 & 1 \end{bmatrix} \in \mathbb{R}^{M \cdot n_{MV} \times M \cdot n_{MV}}. \tag{33}$$

If soft constraints are to be regarded in the controller formulation, the output variables Y, need to be slacked by ϵ_l and ϵ_h. One way to accomplish this mathematically is to enlarge the optimization vector, z_{st}, with additionally $Y(k)$ as formulated in (35). Using this formulation, each output variable is slacked twice. One time representing upper slack and one time representing lower slack. However, one additional matrix is needed in order to represent the slack property. In general, depending on the prediction horizon, there are in total $(P - W) \cdot n_{CV}$ output variables. Still, there are only a total of n_{CV} slack variables - one for each output. To make the dimensions become mathematical valid for subtraction and addition, the n_{CV} slack variables must be mapped to $(P-W) \cdot n_{CV}$. This is accomplished using the scaling matrix (34).

$$\mathbf{1} = \begin{bmatrix} 1_{0,1} & 0 & \ldots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 1_{(P-W),1} & 0 & \ddots & 0 \\ 0 & 1_{0,2} & \ddots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 1_{(P-W),2} & \ddots & 0 \\ \vdots & \ddots & \ddots & \vdots \\ 0 & 0 & 0 & 1_{0,n_{CV}} \\ \vdots & \vdots & \vdots & \vdots \\ 0 & 0 & 0 & 1_{(P-W),n_{CV}} \end{bmatrix} \in \mathbb{R}^{(P-W) \cdot n_{CV} \times n_{CV}}. \tag{34}$$

$$\underline{Y} - \mathbf{1}\epsilon_l \leq \tilde{Y} \leq \overline{Y} + \mathbf{1}\epsilon_h \Leftrightarrow \begin{cases} -\infty \leq \tilde{Y} \leq \overline{Y} + \mathbf{1}\epsilon_h \\ \underline{Y} - \mathbf{1}\epsilon_l \leq \tilde{Y} \leq \infty \end{cases} \Leftrightarrow \begin{cases} -\infty \leq \tilde{Y} - \mathbf{1}\epsilon_h \leq \overline{Y} \\ \underline{Y} \leq \tilde{Y} + \mathbf{1}\epsilon_l \leq \infty \end{cases} \tag{35}$$

By grouping the optimization variables $\Delta U, \tilde{U}, Y, \epsilon_h$ and ϵ_l together in a vector z_{st}, the overall problem leads to the standard QP problem formulation (14). This can be concluded by comparing the standard QP formulation to the cost function (36) and corresponding linear constraints (37). It can be shown that $\overline{\boldsymbol{R}}, \overline{\boldsymbol{Q}} \succ 0 \implies \boldsymbol{G_{st}} \succeq 0$. For simplicity, the inequality constraints are left out since they are not used for further derivation. The standard quadratic program formulation is the base formulation deriving the MPC-FSRM.

$$
\begin{aligned}
&\min_{z_{st}} \frac{1}{2} z_{st}^T \mathbf{G} z_{zt} + q(k)^T z_{st} \\
&= \min_{z_{st}} \frac{1}{2} \begin{bmatrix} \Delta U \\ U \\ \tilde{Y} - \epsilon_h \\ \tilde{Y} + \epsilon_l \\ \epsilon_h \\ \epsilon_l \end{bmatrix}^T \begin{bmatrix} 2\overline{\boldsymbol{R}} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} \\ \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} \\ \mathbf{0} & \mathbf{0} & 2\overline{\boldsymbol{Q}} & \mathbf{0} & \mathbf{0} & \mathbf{0} \\ \mathbf{0} & \mathbf{0} & \mathbf{0} & 2\overline{\boldsymbol{Q}} & \mathbf{0} & \mathbf{0} \\ \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} \\ \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} \end{bmatrix} \begin{bmatrix} \Delta U \\ U \\ \tilde{Y} - \epsilon_h \\ \tilde{Y} + \epsilon_l \\ \epsilon_h \\ \epsilon_l \end{bmatrix} \\
&\qquad + \begin{bmatrix} \mathbf{0} & \mathbf{0} & -2\mathcal{T}(k)^T \overline{\boldsymbol{Q}} & -2\mathcal{T}(k)^T \overline{\boldsymbol{Q}} & \rho_h^T & \rho_l^T \end{bmatrix} \begin{bmatrix} \Delta U \\ U \\ \tilde{Y} - \epsilon_h \\ \tilde{Y} + \epsilon_l \\ \epsilon_h \\ \epsilon_l \end{bmatrix}
\end{aligned}
\tag{36}
$$

such that

$$
\begin{aligned}
\boldsymbol{E} z_{st} = \begin{bmatrix} -\boldsymbol{I} & \boldsymbol{K} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} \\ -\boldsymbol{\Theta} & \mathbf{0} & \boldsymbol{I} & \mathbf{0} & \mathbf{1} & \mathbf{0} \\ -\boldsymbol{\Theta} & \mathbf{0} & \mathbf{0} & \boldsymbol{I} & \mathbf{0} & -\mathbf{1} \end{bmatrix} \begin{bmatrix} \Delta U \\ U \\ \tilde{Y} - \epsilon_h \\ \tilde{Y} + \epsilon_l \\ \epsilon_h \\ \epsilon_l \end{bmatrix} = \\
\begin{bmatrix} \boldsymbol{\Gamma} \tilde{U}(k-1) \\ \boldsymbol{\Phi} \Delta \tilde{U}(k) + \boldsymbol{\Psi} \tilde{U}(k-N) + B(k) \\ \boldsymbol{\Phi} \Delta \tilde{U}(k) + \boldsymbol{\Psi} \tilde{U}(k-N) + B(k) \end{bmatrix} = f
\end{aligned}
\tag{37}
$$

2.3.2 Condensed formulation

The condensed formulation targets a QP on the form:

$$
\begin{aligned}
\min_{z} \quad & \frac{1}{2} z^T \boldsymbol{G} z + q^T z, \quad \boldsymbol{G} = \boldsymbol{G}^T \succeq 0, \\
\text{s.t.} \quad & l \leq \boldsymbol{A} z \leq u.
\end{aligned}
\tag{38}
$$

This formulation differs from the standard quadratic formulation in which there are no linear constraints present. The constraints are rather formulated as a linear system of inequalities upper and lower bounded. In order to get to the target QP (38), a reformulation

of the constraints is needed. Recall the structure of the optimization vector used in (36):

$$z_{st} = \begin{bmatrix} \Delta U \\ U \\ \tilde{Y} - \epsilon_h \\ \tilde{Y} + \epsilon_l \\ \epsilon_h \\ \epsilon_l \end{bmatrix} \in \mathbb{R}^{2\cdot(M\cdot n_{MV}+\cdot P\cdot n_{CV}+n_{CV})}. \tag{39}$$

As formulated in Equation (25), both $Y(k)$, and $U(k)$ are expressions of $\Delta U(k)$ and by use of these equality constraints the number of optimization variables can be reduced. This method is called the *Null Space Method* described in [16] (Chapter 16.2). The principle is that the equality constraints present in the optimization problem can be folded into the cost function, reducing the function to its null space. Hence, a reduced MPC formulation can be achieved yielding a condensed formulation. For the standard QP formulation described in Equation (36) the optimization vector (39) is reduced to

$$z_{cd} = \begin{bmatrix} \Delta U \\ \epsilon_h \\ \epsilon_l \end{bmatrix} \in \mathbb{R}^{M\cdot n_{MV}+2\cdot n_{CV}}. \tag{40}$$

The fewer optimization variables, the lower yields the run-time complexity of the MPC algorithm. Hence the condensed formulation is preferable for this project. In order to derive the condensed formulation using the *Nullspace method*, Equation (25) and (27a) is utilized to describe $U(k)$ and $Y(k)$ in terms of $\Delta U(k)$. The aim is to represent the optimization vector as a linear combination of the condensed vector, yielding $z_{st} = \boldsymbol{A}z_{cd} + C$. The expression has the property such that $\boldsymbol{EA} = \mathbf{0}$ and $\boldsymbol{A}$ being invertible. Using this expression, one can reduce the standard quadratic program cost function (14). This derivation is shown in Equation (41).

$$\begin{aligned} &\min_{z_{st}} \frac{1}{2} z_{st}^T \mathbf{G} z_{zt} + q^T z_{st} \\ &= \min_{z_{cd}} \frac{1}{2} (\mathbf{A}z_{cd} + C)^T \mathbf{G}(\mathbf{A}z_{cd} + C) + q^T(\mathbf{A}z_{cd} + C) \\ &= \frac{1}{2} z_{cd}^T \mathbf{A}^T \mathbf{G}\mathbf{A} z_{cd} + (C^T \mathbf{G}\mathbf{A} + q^T \mathbf{A}) z_{cd} \\ &= \frac{1}{2} z_{cd}^T \boldsymbol{G_{cd}} z_{cd} + q_{cd}^T z_{cd} \end{aligned} \tag{41}$$

The matrix $\mathbf{A}$ and vector C are derived in Equation (42).

$$
\begin{aligned}
z_{st} = \begin{bmatrix} \Delta U \\ U \\ \tilde{Y} - \epsilon_h \\ \tilde{Y} + \epsilon_l \\ \epsilon_h \\ \epsilon_l \end{bmatrix} &= \begin{bmatrix} \Delta U \\ \mathbf{K}^{-1}\boldsymbol{\Gamma}\tilde{U}(k-1) + \mathbf{K}^{-1}\Delta U(k) \\ \Lambda(k) + \boldsymbol{\Theta}\Delta U(k) - \epsilon_h \\ \Lambda(k) + \boldsymbol{\Theta}\Delta U(k) + \epsilon_l \\ \epsilon_h \\ \epsilon_l \end{bmatrix} \\
&= \begin{bmatrix} \mathbf{I} & \mathbf{0} & \mathbf{0} \\ \mathbf{K^{-1}} & \mathbf{0} & \mathbf{0} \\ \boldsymbol{\Theta} & \mathbf{-1} & \mathbf{0} \\ \boldsymbol{\Theta} & \mathbf{0} & \mathbf{1} \\ \mathbf{0} & \mathbf{I} & \mathbf{0} \\ \mathbf{0} & \mathbf{0} & \mathbf{I} \end{bmatrix} \begin{bmatrix} \Delta U \\ \epsilon_h \\ \epsilon_l \end{bmatrix} + \begin{bmatrix} 0 \\ \mathbf{K}^{-1}\boldsymbol{\Gamma}\tilde{U}(k-1) \\ \Lambda(k) \\ \Lambda(k) \\ 0 \\ 0 \end{bmatrix} \\
&= \mathbf{A}z_{cd} + C(k)
\end{aligned} \tag{42}
$$

Hence, the condensed MPC formulation can be summarized as,

$$
\min_{z_{cd}} \frac{1}{2} z_{cd}^T \boldsymbol{G_{cd}} z_{cd} + q_{cd}^T(k) z_{cd}, \tag{43a}
$$

$$
\boldsymbol{G_{cd}} = \mathbf{A}^T \mathbf{G} \mathbf{A} = 2 \cdot \begin{bmatrix} \overline{\boldsymbol{R}} + 2\boldsymbol{\Theta}^T\overline{\boldsymbol{Q}}\boldsymbol{\Theta} & -\boldsymbol{\Theta}^T\overline{\boldsymbol{Q}}\mathbf{1} & \boldsymbol{\Theta}^T\overline{\boldsymbol{Q}}\mathbf{1} \\ -\mathbf{1}^{\boldsymbol{T}}\overline{\boldsymbol{Q}}\boldsymbol{\Theta} & \mathbf{1}^{\boldsymbol{T}}\overline{\boldsymbol{Q}}\mathbf{1} & \mathbf{0} \\ \mathbf{1}^{\boldsymbol{T}}\overline{\boldsymbol{Q}}\boldsymbol{\Theta} & \mathbf{0} & \mathbf{1}^{\boldsymbol{T}}\overline{\boldsymbol{Q}}\mathbf{1} \end{bmatrix} \succeq 0, \tag{43b}
$$

$$
q_{cd}(k) = C(k)^T \mathbf{G}\mathbf{A} + q^T \mathbf{A} = 2 \cdot \begin{bmatrix} 2 \cdot \boldsymbol{\Theta}^T\overline{\boldsymbol{Q}}(\Lambda(k) - \mathcal{T}(k)) \\ -\mathbf{1}^T\overline{\boldsymbol{Q}}(\Lambda(k) - \mathcal{T}(k)) + \rho_h \\ \mathbf{1}^T\overline{\boldsymbol{Q}}(\Lambda(k) - \mathcal{T}(k)) + \rho_l \end{bmatrix}, \tag{43c}
$$

such that

$$
\underline{z_{cd}}(k) \leq \boldsymbol{A} z_{cd} \leq \overline{z_{cd}}(k), \tag{44a}
$$

yielding,

$$
\begin{bmatrix} \Delta\underline{U} \\ \underline{U} \\ -\infty \\ \underline{Y} \\ 0 \\ 0 \end{bmatrix} - \begin{bmatrix} 0 \\ \mathbf{K}^{-1}\boldsymbol{\Gamma}\tilde{U}(k-1) \\ \Lambda(k) \\ \Lambda(k) \\ 0 \\ 0 \end{bmatrix} \leq \begin{bmatrix} \mathbf{I} & \mathbf{0} & \mathbf{0} \\ \mathbf{K^{-1}} & \mathbf{0} & \mathbf{0} \\ \boldsymbol{\Theta} & \mathbf{-1} & \mathbf{0} \\ \boldsymbol{\Theta} & \mathbf{0} & \mathbf{1} \\ \mathbf{0} & \mathbf{I} & \mathbf{0} \\ \mathbf{0} & \mathbf{0} & \mathbf{I} \end{bmatrix} z_{cd} \leq \begin{bmatrix} \Delta\overline{U} \\ \overline{U} \\ \overline{Y} \\ \infty \\ \infty \\ \infty \end{bmatrix} - \begin{bmatrix} 0 \\ \mathbf{K}^{-1}\boldsymbol{\Gamma}\tilde{U}(k-1) \\ \Lambda(k) \\ \Lambda(k) \\ 0 \\ 0 \end{bmatrix}. \tag{44b}
$$

2.3.3 Properties of the condensed formulation

Due to the positive semi-definite property of the standard QP (36), yielding $\boldsymbol{G} \succeq 0$. It can be shown that this property also is passed to $\boldsymbol{G_{cd}}$ (43b). This is a result of positive semi-definiteness being closed under a congruence transform. For $\boldsymbol{G}$ being positive semi-definite:

$$
\boldsymbol{G} \succeq 0 \iff y^T \boldsymbol{G} y \geq 0, \quad \forall y. \tag{45}
$$

Table 1: Condensed form matrix dimensions

Matrix expression	Condensed dimensions
z_{cd}	$M \cdot n_{MV} + 2 \cdot n_{CV}$
q_{cd}	$M \cdot n_{MV} + 2 \cdot n_{CV}$
$\mathcal{T}$	$(P - W) \cdot n_{CV}$
Λ	$(P - W) \cdot n_{CV}$
$\overline{\boldsymbol{Q}}$	$n_{CV} \cdot (P - W) \times n_{CV} \cdot (P - W)$
$\overline{\boldsymbol{R}}$	$M \cdot n_{MV} \times M \cdot n_{MV}$
$\boldsymbol{\Theta}$	$n_{CV} \cdot (P - W) \times M \cdot n_{MV}$
$\boldsymbol{\Phi}$	$n_{CV}\,(P - W) \times \sum_{j=1}^{n_{MV}} (N_* - W - 1)$
$\boldsymbol{\Psi}$	$n_{CV}\,(P - W) \times n_{MV}$
$\boldsymbol{K}/\boldsymbol{K^{-1}}$	$M \cdot n_{MV} \times M \cdot n_{MV}$
$\boldsymbol{\Gamma}$	$M \cdot n_{MV} \times n_{MV}$
$\boldsymbol{A}$	$2M \cdot n_{MV} + 2 \cdot (P - W) \cdot n_{CV} + 2 \cdot n_{CV} \times M \cdot n_{MV} + 2 \cdot n_{CV}$
$\boldsymbol{G_{cd}}$	$M \cdot n_{MV} + 2 \cdot n_{CV} \times M \cdot n_{MV} + 2 \cdot n_{CV}$

Define $y = \boldsymbol{A}x$.

$$x^T \boldsymbol{A}^T \boldsymbol{G} \boldsymbol{A} x \geq 0, \quad \forall x \implies \boldsymbol{A}^T \boldsymbol{G} \boldsymbol{A} = \boldsymbol{G_{cd}} \succeq 0 \quad \square \tag{46}$$

Hence, one can conclude that the condensed formulation is a convex program with the properties stated in Section 2.2.3.

Clearly, this problem formulation is a more compact representation compared to the standard MPC. The problem size can be characterised by the number of optimization variables, n and the number of constraints, m. One can show that:

$$n = M \cdot n_{MV} + 2 \cdot n_{CV} = dim(z_{cd}), \tag{47a}$$

$$m = 2 \cdot M \cdot n_{MV} + 2 \cdot (P - W) \cdot n_{CV} + 2 \cdot n_{CV}. \tag{47b}$$

Moreover, one can see that the problem size is proportional not only to the number of model variables but also to the horizons. Hence, an MPC based on a MIMO FSRM is a fairly memory-demanding system description. An overview of other matrix dimensions used in the formulation is given in Table 1. The condensed problem formulation stated in the Equations (43) serves as the main MPC algorithm in *MPC-simulator*, as it is the least memory-consuming MPC. For scenarios where slack variables are disregarded, the controller (58) is rather simulated. For simulation purposes, an overview of all the parameters needed to simulate the condensed controllers is represented in Table 2. There are in total fourteen different scalars and vectors needed to simulate the MPC-FSRM derived in this section.

Table 2: Controller parameters. Description of all parameters needed in order to simulate the condensed Model Predictive Controller. The identifiers are also used in the implementation of *MPC-simulator*.

Parameter	Description
P	Prediction Horizon
M	Control Horizon
W	Start Horizon
$\boldsymbol{Q}$	Output error penalty
$\boldsymbol{R}$	Actuation penalty
ρ_h	Upper slack penalty
ρ_l	Lower slack penalty
$\overline{z}$	Upper constraints
$\underline{z}$	Lower constraints

3 Implementation

Section 3.1 describes the implementation of the simulator software. This design is a continuation of the software presented in the pre-project. Therefore, Section 3.1 resembles the software design Chapter 5 in [29]. Consequently, to the outline of the simulator implementation, the design of the *Web-application*, Section 3.2, is addressed.

3.1 *MPC-simulator*

The *MPC-simulator* is the foundation of *Light-Weight MPC*. As stated in the problem description, Section 1.2, this tool incorporates MPC and conducts closed-loop simulations using an FSRM. Once all the necessary information is parsed, the condensed controller defined by Equation (43) can be instantiated for simulation. The controller performance is directly related to the trace-back of the output trajectories. Therefore, it is essential to store this information in a structured manner to enable further assessment The same applies to controller tuning and system data. Considering that the software exists within a digital realm, the plant must also be simulated alongside the controller. Therefore, the simulation step is the most intricate part of the project, as it brings together all the necessary components to generate simulation data. The complexity of the step is highly related to the complexity of the controller itself. Hence, a C/C++ implementation is considered to handle potentially large run-time requirements efficiently. Additionally, the software should account for errors such as invalid data and simulation failures, adopting defensive implementation practices and providing meaningful feedback to the end user. Like any framework, the simulator should accommodate various scenarios, making generic implementations the preferred approach.

To gain insights into the functionality of simulation software for control algorithms, initial research was conducted on SEPTIC. Understanding this software seemed appropriate since the simulator aimed to encompass its core functionality. However, unlike SEPTIC, *Light-Weight MPC* emphasises portability over advanced MPC features. The lightweight software is distributed as open source and built using modern software practices. The final

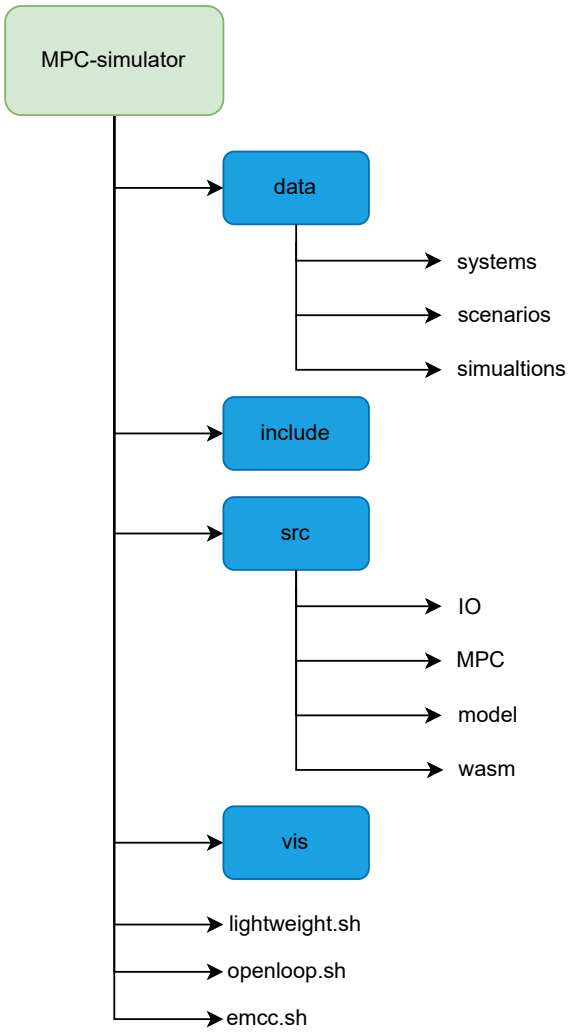


Figure 5: Illustration of the *MPC-simulator* folder structure

simulator's folder structure is illustrated in Figure 5.

3.1.1 Toolchain

Every software relies on a platform to run effectively. To efficiently build and compile the simulator, it was necessary to design a suitable toolchain. Since C/C++ is a compiled language and not an interpreted, the build procedure is a bit more involved as the compilation is platform-dependent. In contrast to an interpreted language, a compiler assess the whole code base in the executable files produced. This makes the code execution time comparatively less since the program is translated into machine code. Therefore, C/C++ are preferred implementation language for heavily computational tasks. The process of producing machine code from source code involves four essential steps: preprocessing, compiling, assembling, and linking of the source code [28]. Besides accounting for these different build steps the toolchain should also be handy in use. The toolchain was intended to be cross-platform, hence runnable on multiple OSs. Considering the hardware limitations, the toolchain release is specifically designed for Unix-based operating systems like

macOS and Linux. The implementation relies on the GNU toolchain [8], incorporating *GNU Make* and GCC. Additionally, to automate the compilation process, *CMake* has been integrated into the toolchain.

The principle behind this building software is as follows: Based on a text file, *CMake* allows the user to specify different targets such as compiler flags, the programming language version, external libraries etc. Dependent on the flag configuration, the software generates a suitable *Makefile* describing how the compiler shall compile and link the code base together. By enhancing a strict folder structure and clearly separating the source code from the headers, shown in Figure 5, *CMake* is able to notice any changes in the code base. It consequently notifies the compiler to add this advancement to the executable files. In contrast to GCC, *CMake* is cross-platform and can target other compilers, making the extension to Windows OS possible. Followed by the generation of the *Makefiles*, *GNU Make* can interpret the file and call GCC to compile the software consequently.

In addition to local software, standardised packages and libraries were needed in order to implement code in a structured and readable manner. Instead of importing several header files describing the interface of a C/C++ library, a package manager was utilized. *Conda* [2] is an open-source package, dependency and environment manager for a waste set of programming languages, including C/C++. *Conda* allows programming environments, which is a major benefit, making dependency issues easy to handle. *MPC-simulator* is based on several different C/C++ packages, and instead of downloading these separately into *Conda*, one can simply download an environment instead. A programming environment defines a collection of dependencies targeting a specific application. This design choice is especially handy as packages become deprecated with time. The *MPC-simulator* offers two downloadable environments for different operating systems. The first environment, named "linux", is designed for Linux-based OSes. The second environment is tailored for macOS, aligning with its corresponding naming convention. Despite being a highly diverse package manager, it proved to be easily integratable into the toolchain. Figure 6 shows an overview of the toolchain.

3.1.2 Data flow

As indicated in Table 2, there are numerous parameters that contribute to the diversity of an MPC. Additionally, in order to produce the controller simulation, the definition of the plant model is also needed. These two components are needed to complete the simulation loop Figure 3. Since the *MPC-simulator* is designed to accommodate various scenarios involving different controllers and FSRMs, the input data may vary in size. Therefore, it was necessary to structure the controller and plant data in a logical and practical manner. The IO formats are described in the appendix Section B. The input files are divided into two formats: scenario and system files. The system file provides a detailed description of the specific model representation being simulated. On the other hand, the scenario file defines the corresponding controller and establishes a link to the relevant system file. This highlights the fact that simulating a controller in isolation is meaningless without understanding the underlying system. While the system and simulation files can possess diverse structural properties, the goal of implementing a lightweight interface led to specific design considerations. The system file is customized for the FSRM, while the scenario file

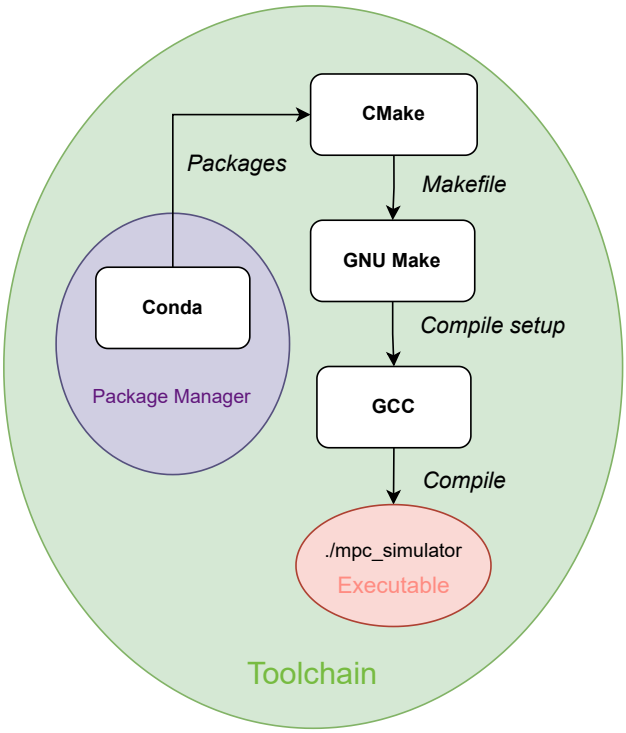


Figure 6: Illustration of the *MPC-simulator* toolchain. The components represent the software used to realise the different C/C++ build steps.

is tailored for the MPC-FSRM. In future implementations, there may be opportunities to reuse the same configuration for other controller and model definitions.

For this purpose, the lightweight data-interchange format JavaScript object notation (JSON) format was chosen [19]. Due to the easily readable interface and programming language support, JSON is widely used in the software community. Moreover, since the format is so usual, there exists a diversity of different software for parsing and serializing such formats. This also emphasises the portability of the data to other frameworks, which can readily parse the simulation data for further applications. All of these properties clearly made JSON a much-preferred data format. The detailed structure of the scenario and system files is described in the appendix, Section B.2 and B.3.

Similarly, an output format was needed in order to be able to interpret the simulation data and analyse the controller performance. Unlike the input formats, determining the output format was not as straightforward or obvious. The selected output format drew inspiration from SEPTIC. The stored data includes the optimal actuation values (U), the reference trajectory to be followed ($\mathcal{T}$), and the simulated output of the model among others. For boundary analysis, the constraints are also added to the simulation format. The output data was customised in order to exploit different aspects of the controller performance. Similarly to the input format, this format can also for future reference be customised to include certain data for the specific scenario. The description of the file is found in the appendix Section B.1 and Figure 7 illustrates the simulator's dataflow.

To fulfil the requirement of parsing and serializing JSON-formatted files with logic compatible with C/C++, external software was imported. The emphasis during this process

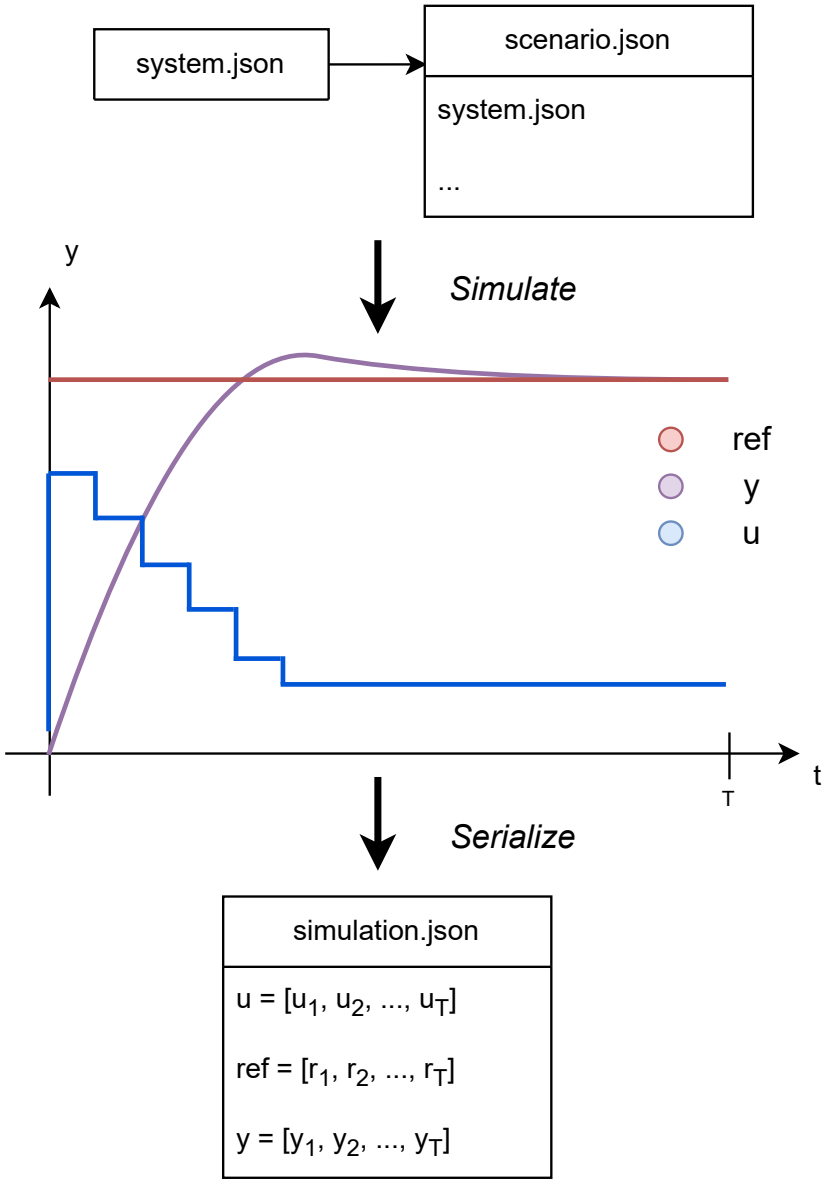


Figure 7: Illustration of the *MPC-simulator* data flow, defining the different steps from processing input to producing output. The input files are represented by the system and scenario files. The simulation results are serialized into a simulation file.

was on achieving efficiency and readability. The open-source JSON parser, *nlohmann* [13], was a popular option for this purpose written in modern C++. Besides the software, a well-documented Application programming interface (API) followed with the purpose of aiding implementation. The software could also be installed via *Conda*, which fitted perfectly into the toolchain described in Section 3.1.1. The utilization of the *nlohmann* software primarily involves including a `nlohmann-json` object, which behaves like a C++ container. By defining the data, one can easily pass the information to the container, and it will generate the corresponding JSON format. However, when reading a JSON file, the data needs to be converted into standard C++ types. For this objective, several C++ classes and structs were implemented in order to categorize the data efficiently from the specified input format.

3.1.3 Finite Step-Response Model

The implementation of the FSRM proved to be the most challenging aspect of the simulator. Despite the model itself being composed of a series of step-responses, there are multiple matrices that need to be constructed and updated during simulation. Equation (25) serves as an example of one such matrix. To express the matrices software-wise involves allocating memory and several index operations. If the simulator was to regard other MPC and models it would be preferable to import a standard linear algebra library. *Eigen* [10] is a C++ template library used for linear algebra. The framework provided a simple interface to define vectors and matrices along with efficient linear algebra operations such as matrix multiplication, transpose, inversion and others. The library is fast and reliable with good compiler support [10]. Similarly to the *nlohmann* parser, the *Eigen* software is also well-documented with an existing API. This library was a clear choice to import into *MPC-simulator*.

The implementation of the FSRM is found in the file `FSRModel.h`. `FSRModel` is a C++ class with a constructor taking a data object described in Section 3.1.2 as an argument. The object has member variables corresponding to horizon parameters from Table 2 and the matrices needed to calculate the predicted output Equation (25). In the general case, the Finite Step-Response Model could include an unknown amount of CVs and MVs hence the object is constructed at run-time, allowing dynamic memory management. Additionally, n_{CV} and n_{MV} could possibly take on large values making these member variables consume a lot of the program memory. The most memory-consuming variables were often sparse matrices, matrices containing a high number of zero elements. Such variables were therefore implemented using a memory-optimized type `Eigen::SparseMatrixXd`, which is an efficient data type for describing dense matrices. Pointers were also utilized in the model implementation with the same aim of reducing memory consumption. By using pointer expressions instead, the software references a certain predefined data type instead of traditionally copying it. Avoiding copying the program circumvents unnecessary usage of program memory. Poor memory handling can lead to stack overflow and system crash and should be dealt with cautiously. Therefore is the object encapsulated, with a destructor releasing the dynamically allocated memory.

3.1.4 Condensed Model Predictive Controller

Up until this point, the methods used to solve the MPC optimization problem (43) have not been addressed. The project description does not require an implementation of an open-source solver. To derive and implement such a solver is a very involved and complicated task. Hence, several optimizer frameworks were addressed for this purpose. All addressed optimizers had the ability to solve a convex QP in a fast manner.

OSQP [27] is a numerical optimization package solving convex QPs on the form (48), where x is the optimization variable and $\boldsymbol{G}$ is a positive semi-definite matrix. This problem formulation aligns perfectly with the condensed MPC (43). This is no coincidence since the algorithm is derived with the purpose of fitting to the OSQP software. The optimizer employed in the Model Predictive Controller implementation is based on the alternating direction method of multipliers (ADMM) approach. ADMM is a first-order optimization method known for its emphasis on efficiency. OSQP outperforms most available commercial and academic solvers, such as *GUROBI* and *MOSEK* [27].

$$\text{minimize} \quad \frac{1}{2}x^T\boldsymbol{G}x + q^T x, \quad \boldsymbol{G} = \boldsymbol{G}^T \succeq 0, \tag{48a}$$

$$\text{subject to} \quad l \leq \boldsymbol{A}x \leq u. \tag{48b}$$

The software provides interfaces for programming languages such as C, Python, Julia, and MATLAB. However, the source code was not implemented directly into *MPC-simulator*, as only a community-maintained version of OSQP was available in C++. This open-source software goes with the name *Osqp-Eigen* and is maintained by *Giulio Romualdi* [18]. Additionally, as the name states, the software is also based on the *Eigen* library addressed in Section 3.1.3. Since *Eigen* already was used in the implementation of the control model, *Osqp-Eigen* was the preferred optimizer software. Additionally, the package provided an example code on how to implement MPC on a linear State Space Model (SSM). This contributed to the reduction of the implementation time of *MPC-simulator*.

3.1.5 Simulation

The `SRSolver()` function is an implementation of the MPC stated in Algorithm 1. The abbreviation, "SR", refers to step response as the control model is an FSRM. A pseudo-code of this algorithm is described in Algorithm 2. The pseudo-code contributes to the understanding of the *MPC-simulator* C++ implementation outlined in the appendix Section D.

As demonstrated in the appendix, additional matrices, namely Ω_y and Ω_u, were required to efficiently implement the simulation loop. These matrices are utilized to select the first element of a vector, specifically the initial predicted output and actuation values. The solution of the QP (43) yields a vector z_{cd} described in (40). However, a Model Predictive Controller only applies the first optimized actuation to the simulated plant.

Algorithm 2 `SRSolver()` - the simulation loop

for $k = 0, \ldots, T$ **do**
  `ReadOutput()` *// Bias update*
  $z_{cd} \leftarrow$ `OsqpEigen::Solver.solveProblem()`
  $\Delta U \leftarrow \boldsymbol{\Omega_u} \cdot z_{cd}(0 : M \cdot n_{MV})$ *// Extraxt actuation*
  $Y \leftarrow \boldsymbol{\Omega_y} \cdot$ `SimulateFSRM`(ΔU) *// Update control model*
  `UpdateQP()`
end for

This extracting operation is realized by multiplying $\boldsymbol{\Omega_u}$ (49) to the sliced optimization vector, characterised by the third step in the algorithm loop.

Similarly, having specified the prediction horizon, propagating the calculated actuation through the FSRM yields a $P \cdot n_{CV}$ long output vector Y. This is the result of the prediction Equation (25). Therefore, in order to parse the first prediction, updating the simulated plant, the vector is multiplied with $\boldsymbol{\Omega_y}$ (50). This calculation is described as the fourth step in the simulation loop. Obtaining all P predictions are also interesting to analyse after the simulation. This is in order to see if the controller plans to stabilize the plant.

$$\boldsymbol{\Omega_u} = \begin{bmatrix} 1 & 0 & \ldots & 0 & 0 & \ldots & 0 & 0 & \ldots & 0 \\ 0 & 0 & \ldots & 1_M & 0 & \ldots & 0 & 0 & \ldots & 0 \\ \vdots & \vdots & \ddots & \vdots & \vdots & \ddots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \ldots & 0 & 0 & \ldots & 1_{M\cdot(n_{MV}-1)} & 0 & \ldots & 0 \end{bmatrix}_{n_{MV} \times n_{MV} \cdot M} \tag{49}$$

$$\boldsymbol{\Omega_y} = \begin{bmatrix} 1 & 0 & \ldots & 0 & 0 & \ldots & 0 & 0 & \ldots & 0 \\ 0 & 0 & \ldots & 1_P & 0 & \ldots & 0 & 0 & \ldots & 0 \\ \vdots & \vdots & \ddots & \vdots & \vdots & \ddots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \ldots & 0 & 0 & \ldots & 1_{P\cdot(n_{CV}-1)} & 0 & \ldots & 0 \end{bmatrix}_{n_{CV} \times n_{CV} \cdot P} \tag{50}$$

The interface to the simulator software is implemented using command-line interface (CLI) as UI. One can simply run the simulator by calling the bash script *make.sh* in the command line. Additional simulation flags can be indicated to the CLI to define specific aspects of the simulation. Allowing different flags to be parsed, permits the user to easily run different simulations. Table 3 indicates the different allowed flags. The arguments are parsed into the C++ software using *CLI11* [24]. This is an open-source argument parser available for downloading through *Conda*. The library turned out to be a powerful command line parser with minimal syntax, fitting perfectly the lightweight simulator.

All arguments are indicated with a corresponding flag, a letter describing the argument. Firstly, the MPC horizon, T, needs to be specified. This variable describes the total number of simulation steps or the number of dynamic optimization problems the controller to be solved. This argument's predecessor is the flag $-T$. Furthermore, the system file, defining the plant model, is specified. This argument is interpreted as an `std::string` with a corresponding $-s$ flag. As the simulator only parses the system definition, it is necessary to have a corresponding default scenario file for the specific system, which defines

Table 3: Simulator argument flags. These flags must be specified in order to call the *MPC-simulator*.

Flag	Description
$-T$ $[int]$	MPC Horizon
$-s$ $[std :: string]$	System
$-r$ $[std :: vector < double >]$	Reference vector
$-n$ $[bool]$	New simulation flag

the controller. If no default scenario file exists, an additional flag must be added to the CLI to specify the scenario. This is avoided as the controller definition is a scenario file named "sce_" followed by the system name. The $-r$ flag symbolises the reference sent to the controller definition. The length of the reference vector must coincide with the model description, otherwise, an `std::invalid_argument` error is returned. The targets parsed are constant values in line with the simulator objectives described in Section 1.2. Typically this is not a valid assumption to make. However, since the current implementation of the simulator only simulates FSRM descriptions, this assumption was made. The final argument represents the ability to recall a previous simulation and simulate for further time steps. This continuation may or may not have a different controller turning and change in control targets. Having this functionality present in the software lets the user test how the controller acts on repetitive changes in controller objectives. When entering a simulator call without the new simulation flag, the simulator parses the information from an old simulation. The parsing is needed for the simulator to initialize at the endpoint of the targeted simulation having the same configuration as the previous controller. To summarise, the simulation can be called using the *lightweight.sh* build script with additional argument flags specified. Ex. *sh lightweight.sh -T [MPC horizon] -s [scenario] -r [reference] -n [new simulation].*

When assessing a new model description in the simulation software, it might be useful to simulate the open-loop response to get an impression of the dynamics. Creating an open-loop simulation is a neat approach to getting insight into the behaviour and characteristics of the plant. This information is valuable to identify potential challenges and to design an appropriate control strategy. Due to the open-loop stability of FSRMs, there is no risk of encountering mathematically undefined simulations. Therefore, the functionality of simulating a process without a control law was also implemented. This functionality can be reached from the CLI. The relevant flags needed to perform an open-loop simulation on an FSRM are shown in Table 4. Since the software assumes constant reference values simulating an MPC, a stabilizing controller will produce a constant-valued actuation when the set-point is reached. As seen in the table, the open-loop simulation requires two arguments to define the actuation. The actuation vector, corresponding to the argument flag $-r$, defines the maximum value the actuation can reach. Consequently, the change in the actuation vector argument determines the increase in actuation for each simulation step. Having the functionality defined, the end user can apply a stable actuation response to the linear model. Performing an open-loop simulation is done by typing *sh openloop.sh -T [MPC horizon] -s [system] -r [actuation] -a [change in actuation].*

In the case where slack variables are not needed in the controller definition, the simulator

Table 4: Open loop argument flags. These flags must be specified in order to simulate an open loop process.

Flag	Description
$-T \quad [int]$	MPC Horizon
$-s \quad [std :: string]$	System
$-r \quad [std :: vector < double >]$	Actuation vector
$-a \quad [std :: vector < double >]$	Change in the actuation vector

disables this property. While great slack penalties contribute to hardening the constraints, small values have a counteracting effect. For instance, by defining $\rho_h = \rho_l = 0$, the slack variables are disregarded in the cost function. Hence, the hard output constraints from the QP description (48) have no effect on the controller performance. In the opposite case, where the end-user wants to treat the output constraints as hard constraints, ρ_h and ρ_l must take on extremely large values. When this is desired, the simulator uses the condensed formulation without slack constraints (57). This is achieved by defining these variables as empty arrays.

3.1.6 Visualization tool

As stated in the problem the simulator shall be equipped with functionality to analyse controller performance. Such an analysis tool would provide insight into the simulator, verifying software implementation. This functionality would therefore not only be valuable for the application itself but also for the development progress. From the early stages of development, it was evident that the tool should incorporate visualizations in the form of plots. However, determining the specific formats of the plots and deciding what information to include was not initially clear. In order to settle on a suitable format exploiting the performance, inspiration was taken from SEPTIC. This format separates the MVs from the CVs in a number of $n_{MV} \cdot n_{CV}$ plots. There are in total three signals accounted for in the plots with a standard colour coding. The output, the actuation and the reference are respectively coloured purple, blue and red. An additional axis is also illustrated in the plot to separate past and future predictions and actions. This axis and colour coding resemble Figure 2. Additionally, the lower and upper constraints are also plotted in order to determine if the CVs or MVs surpass their respective constraints.

The visualization tool is implemented using the Python programming language. Python was chosen due to its neat syntax and ability to include *Matplotlib* [11] which is a common plotting library. The programming language is also easily integrable through *Conda*. Since the tool was implemented in another language, it was developed separately from the simulator. As a result, the analysis tool needs to be called externally after each simulation. Similarly to the simulator interface, the analysis tool also allows CLI argument parsing. The only flag implemented is $-s$, which is used to specify the simulation file described in the appendix Section B.3. Additionally, if the user does not prefer the CLI, a *Jupyter Notebook* file can be run in order to produce the same visualizations. *Jupyter Notebook* is an interface allowing Python code to be run in a web browser [1]. If the *Conda* environment is activated in the terminal, *Jupyter* is activated by calling *jupyter*

notebook in the command line. The implementation of the visualization tool is found in the *vis*-folder shown in Figure 5.

3.1.7 Code competency

An important objective of the simulator software is how code competency is reflected in the implementation. The notion refers to design principles within software development assuring i.e. the maintainability, testability and reliability of the source code. Figure 8 outlines the *MPC-simulator* structure. In line with typical software principles objectives, the code base should be divided into simple modules each having its field of responsibility. In order to fulfil this requirement the simulator consists the three modules: IO, model and MPC. In the IO module, the *nlohmann* JSON parser is imported and IO files are parsed and serialized to achieve the wanted data flow outlined in Section 3.1.2. The JSON files are stored locally in the data folder illustrated in Figure 8. The modules, model and MPC, implement the control model and the corresponding condensed controller (43). The naming of the modules is chosen descriptively emphasising the readability and maintainability of the software.

All modules are implemented defensively, meaning that arguments are typed checked and tested before the simulation routine is further executed. Defensive programming is an important software principle, providing meaningful feedback to the end user and warnings if the software configuration will induce faults and simulation errors. Specifically, central procedures are implemented using try-catch statements. By throwing `std::exceptions` when a fault occurs, the programmer can trace the error back to its origin. Furthermore, all modules are implemented using test-driven development (TDD). This software principle deals with the testability of different functionality. Every core function shall easily be testable to validate functionality. The corresponding tests used to validate the software are defined in the *tests.cc* file. To assure the readability of the source code, the *Doxygen* documentation format [4] is described in every module. Every function is documented likewise as the `SRSolver()` function described in the appendix Section D. On a modular level, readability is assured by having a *README.md* file present. This file is typically the first file a programmer faces when browsing a *GitHub* repository. In addition to explaining the purpose of each module, the *README.md* files also elucidate how to download and run the software.

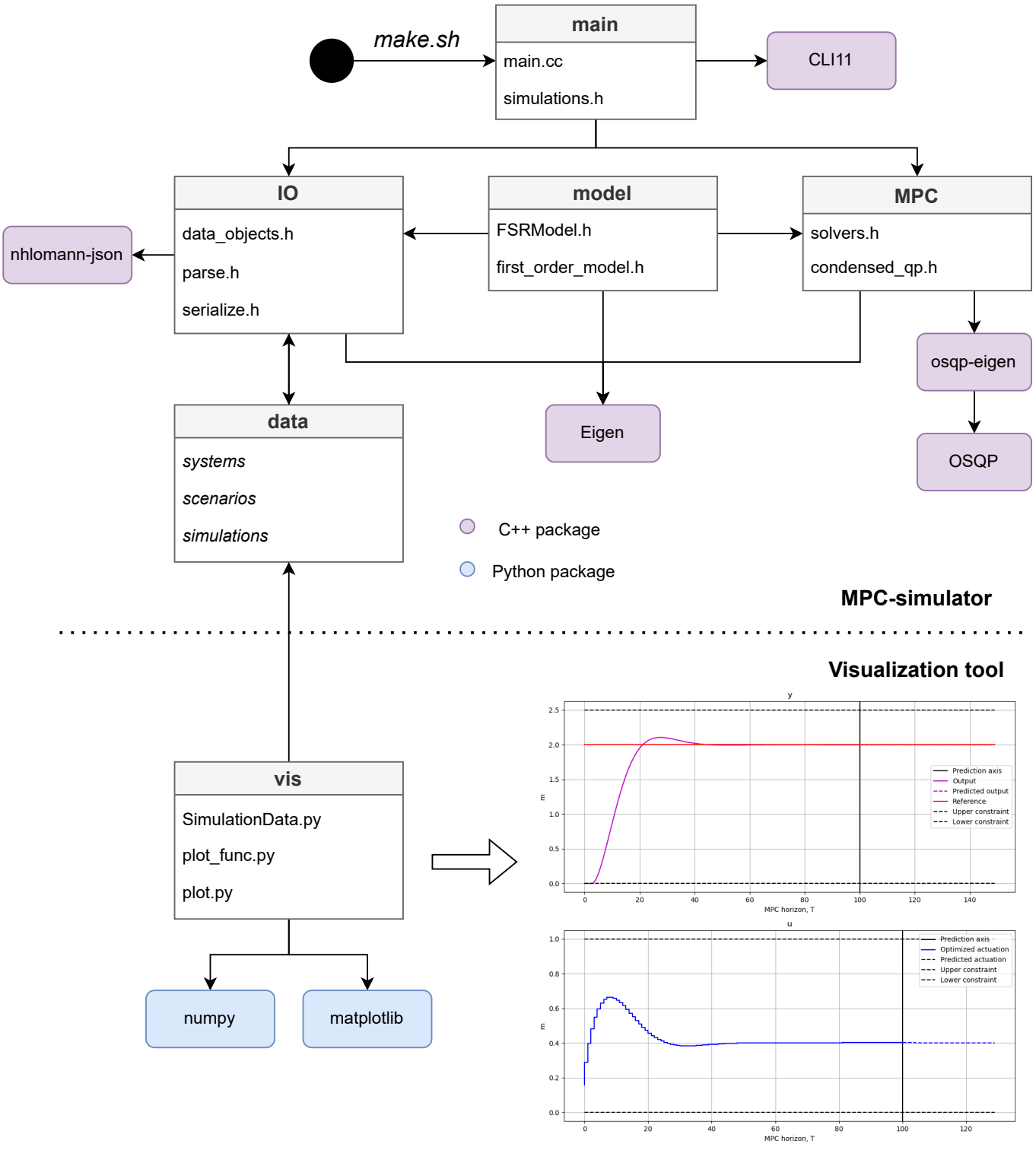


Figure 8: Module diagram describing the *MPC-solver*. Imported Python and C++ packages are also illustrated. The dotted line symbolises the dissection between the simulator and the visualization tool.

3.2 *Web-application*

The *Web-application* serves as a portable, tailored web interface for the implemented controller described in Equation (43). The ability to interface the World Wide Web (WWW) will immersively enhance the simulator's value for users. Supporting communication over the Hypertext Transport Protocol (HTTP) makes the simulation highly accessible and reachable across the continents. The only dependency needed is an Internet connection. In an industrial setting, engineers can utilize the software to assess MPC performance outside the office having remote access. For instance on one of *Equinor's* many oil platforms in the North Sea or other distant facilities. Additionally, the software can be available to many other devices supporting web browsing. Not only stationary PCs but also smaller devices such as tablets, smartphones or other Internet of Things (IoT) devices. This has also an impact on the scalability, letting multiple users access the application. The increased traffic can be tackled by employing a distributed system architecture. Even though this is fully achievable, this is left out in the implementation of *Light-Weight MPC*. However, a static HTTP server is added to the development pipeline for local deployment.

One major motivating factor behind the development of the *Web-application* is the simplification of the simulation procedure. As elaborated in Section 3.1.5, the only interface available is through the command line. By encapsulating the simulator using the *Web-application* as frontend, these dependencies can be abstracted into the interface. By defining a tailored UI, the CLI is replaced with visually appealing and easy-to-navigate user-friendly components.

3.2.1 *React, Create React App*

In order to implement the interactive application, it was early decided to look into *React* and *Create React App* (CRA). *React* is a JavaScript library for defining user-interfaces and is widely used in web development to build interactive applications [22]. The library employed in this project adopts a component-based syntax, where each element of the UI is implemented as a component in JSX format, encapsulating its own logic and rendering. JSX is an extension of JavaScript and can be rendered using a Virtual Document Object Model (DOM). Leveraging a DOM for rendering is a key advantage of utilizing the *React* library. When changes occur in a *React* component, only the necessary parts of the DOM are updated, enhancing performance and rendering efficiency. This rendering approach is particularly well-suited for applications that prioritize web performance. Moreover, as specified in the problem description, Section 1.2, the frontend of the application should support multiple web browsers. According to the documentation [22], *React* is compatible with all modern browsers, including Microsoft Edge, Firefox, Chrome, Safari, and others. In summary, React is a fast library that also offers ease of development and testing, suitable for a lightweight simulation software.

Create React App is a development environment for *React* applications [3]. CRA simplifies the initial setup and configuration, handling necessary dependencies and build processes. The environment's only dependency is *Node.js*. This is an asynchronous event-driven JavaScript run-time [17] used to employ JavaScript projects. Node also provides a default package manager, *Node Package Manger* (npm) which can be used to install open-source

packages needed for implementation. Similarly to *CMake*, CRA manages different build dependencies by enhancing a strict folder structure. Figure 9 illustraits the folder structure of *Web-application* needed to let npm handle package dependencies. The most noticeable folders are the source folder and *node_modules*. The latter folder is a collection of all required *Node*-packages used in the project. All dependencies are stated in the *package.json*. There are multiple ways to deploy the website using the CRA framework. Different approaches are emphasised based on the intention of use. With *Node.js* installed, all that is needed to deploy the software is to call *npm install* in the command line to install the required modules. To initiate a development server, you can execute the command "npm start" as outlined in the installation instructions. This server reacts to changes in the source code updating the layout of the site correspondingly. Another deployment approach, which is more suitable for static web deployment, is to run a production build of the application. This is done by calling *npm run build*. After successfully calling this command, the build folder, visible in Figure 9, is created. Furthermore, this optimized build can be deployed by a static server.

3.2.2 *Webassembly, Emscripten Compiler Frontend*

Even though the setup seems promising, there is still a problem left to be addressed. How to port the simulator to the web? A lot of effort was given into the implementation of *MPC-simulator* and it would be a dissipation if this code was not to be used in the application. The simulator was also implemented using the C++ programming language for efficiency means. Hence, the porting process should be carried out in a manner that ensures efficiency is not compromised.

Wasm is an instruction format designed to run high-performance code on the web [30]. The chosen format aims to be encoded in a binary format that is both efficient in terms of size and load time. This optimization is pursued to attain near-native speed when executing within web browsers. This makes Wasm especially suitable for compute-intensive tasks which JavaScript is not really good at. By using Wasm, one can enlarge the functionality of JavaScript by combining it with low-level programming languages such as C/C++, Rust and Python i.e.

Clearly, the employment of Wasm solves the porting problem between the *MPC-simulator* and *Web-application*. Hence, porting functionality is implemented at both ends. Going back to the folder structure of the application Figure 9, the Wasm compiled simulator interfacing JavaScript is located inside the source folder under the name *mpc_simulator.mjs*. This file easily be imported into a JavaScript function handling the simulator procedure. The simulator file is produced using a compiler called *Emscripten Compiler Frontend* (emcc) [5]. The compiler is a part of the *MPC-simulator* repository, enabling the simulator to easily interface web applications. Even though the Webassembly format is language-agnostic, the emcc is optimized only for C/C++ projects. With the use of *Embind*, one can bind C++ function definitions to JavaScript classes. These *Emscripten bindings* are implemented inside the *wasm* folder in the simulator folder structure illustrated in Figure 5. Furthermore, a CLI is implemented to run the compilation. The compile configuration interfacing *CMake* is implemented in the bash script *emcc.sh*. Figure 10 illustrates the relation the build script has on the lightweight pipeline.

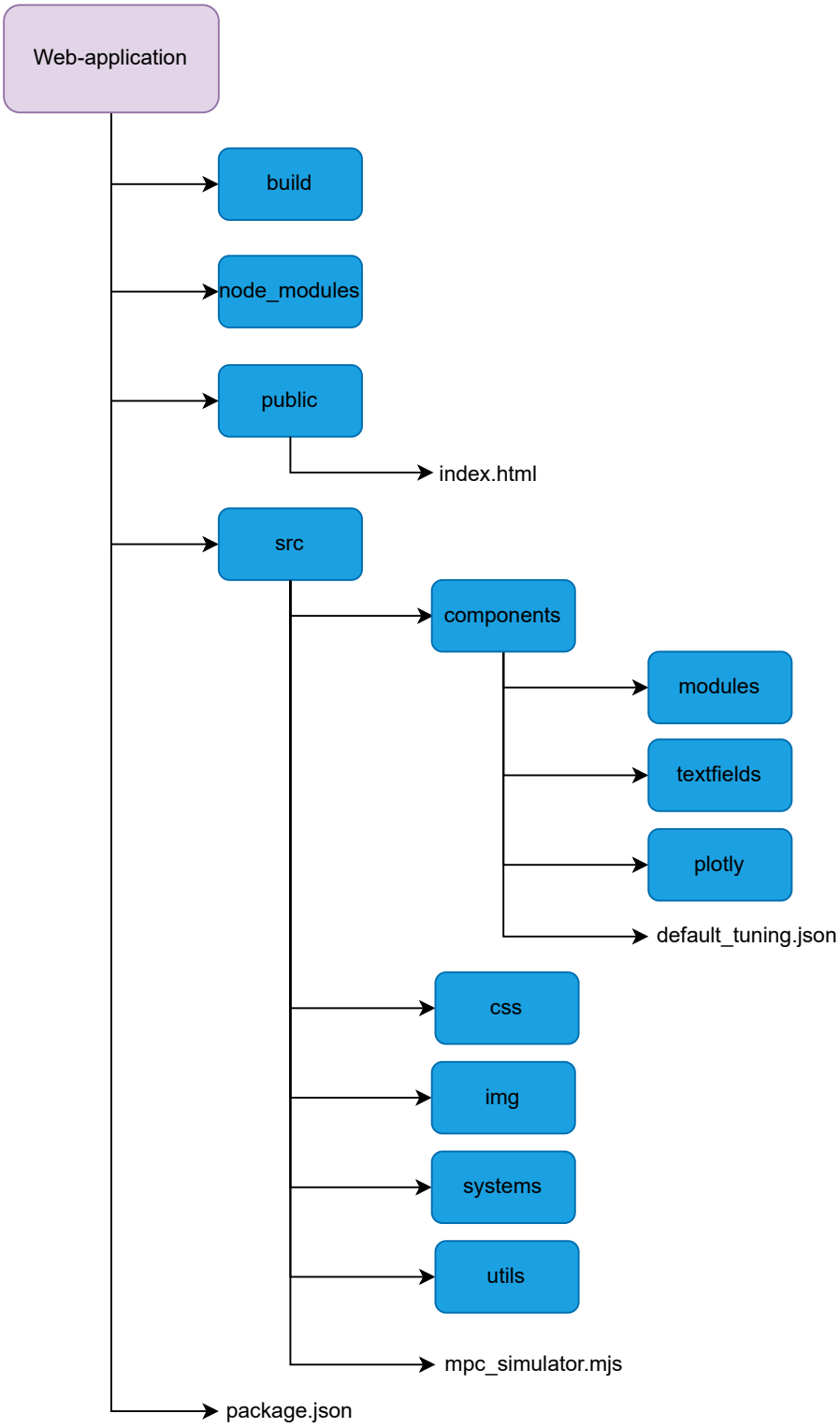


Figure 9: *Web-application* folder structure. The upper hierarchical folders consisting of *build, node_modules, public* and *src* are enforced for dependency management.

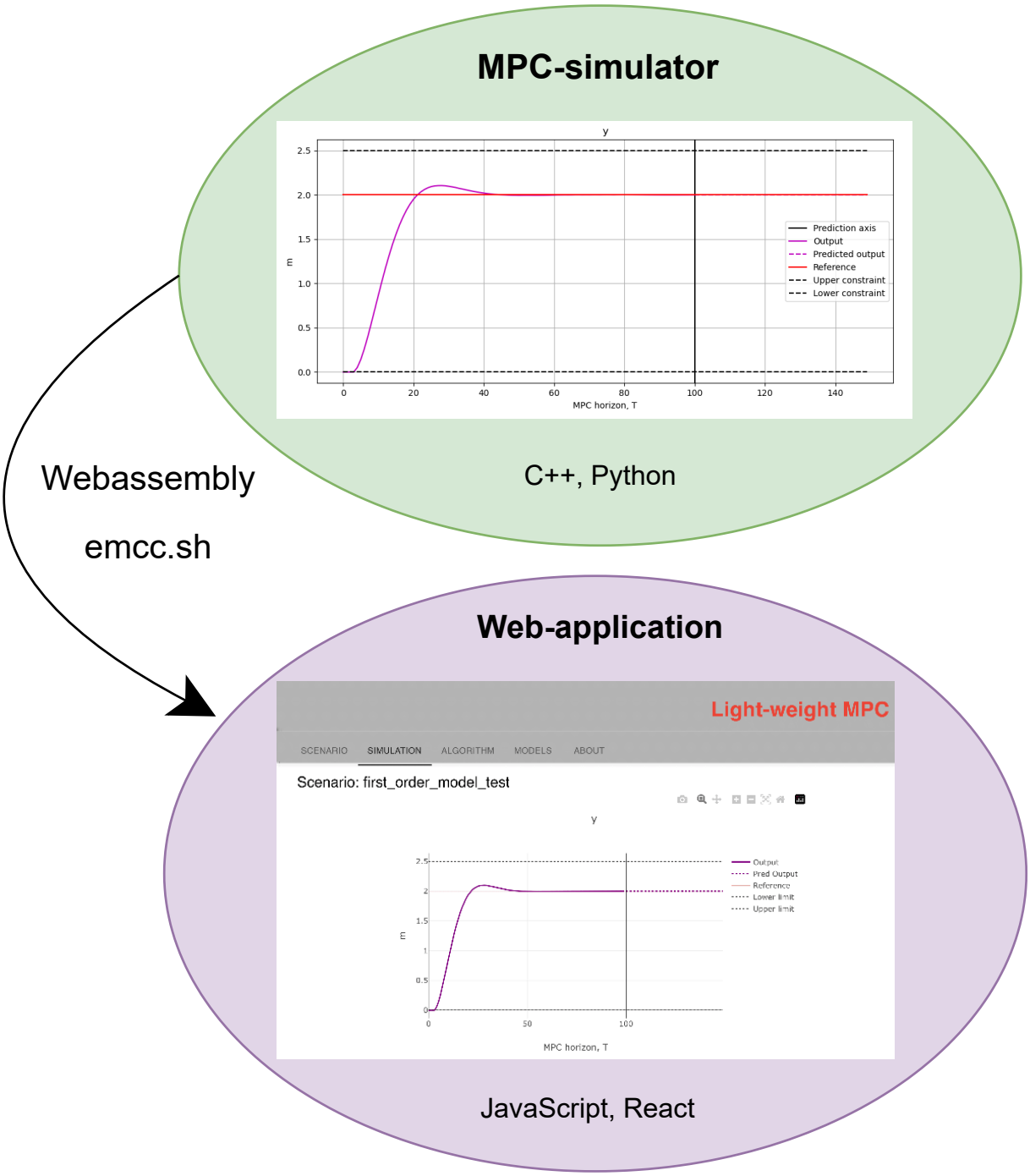


Figure 10: *Empscripten Compiler Frontend* is used to compile the simulator to Wasm. This file can further be imported into the application.

3.2.3 Database

As stated in a previous section, the major benefit of encapsulating the simulator into a web application is that it can simplify the data flow. Even though the scenario files can be defined through interactive components, the simulator still has a strong dependency on system files. In the domain of web development, such external data is normally stored in a database. The *Web-application* can easily interface external databases performing an HTTP request. Such databases are needed when the framework is to simulate many different FSRMs. However, at an early stage of development, only a total of two FSRMs are added to the simulator. Due to time constraints, it was chosen to only implement a local database, leaving the implementation of a system file-based database for future work. The local database resembles the data folder defined inside the simulator's source. The web application parses the content of the local database displaying them inside a *React* component for user selection. The local database is denoted as *systems* in Figure 9.

3.2.4 Plotly

Similarly to the *MPC-simulator* objectives, the *Web-application* also needs the functionality to analyse controller performance. This functionality aims to preserve the visualization tool described in Section 3.1.6. While the original tool was implemented using Python's *matplotlib*, this web functionality was implemented using *React's Plotly* [20]. This is a JavaScript graphing library imitating most of the functionality a visualization framework needs. Similarly to the Python package, a well-documented API was available for implementation specifics, minimising the development time. The *Line Chart* object use to plot the simulation data inside the *React*-application also allows zooming into the graphs for detailed assessments. Additionally, these reactive components can also save the graphs produced in order to address different simulation results.

3.2.5 User experience and user interface

The user-interface of any web application plays a crucial role in its success and user satisfaction. The UI directly impacts the UX and the overall impression of the application. It defines how users interact, navigate and accomplish their tasks using the application. The application's UI is also the main contributing factor to the user's first impression. A positive impression would likely motivate usage and app exploration. Above all the UI should be easy to understand and improve productivity for the user. In order to implement user-friendly components for the project, *Material UI* was imported. This is an open-source *React* component library providing predefined components for development [14]. For instance, interactive *TextFields*, buttons, URL links and more could be imported and integrated into the project. Many of the components are also highly customizable and can be integrated to suit individual tasks.

The *Web-application* consists mainly of three components, controlling several sub-components and corresponding child processes. These components are the Header, Body and Footer, describing the upper, lower and middle parts of the website's layout. Figure 11 shows the layout of the front page. As seen the Header is found in the upper part of the page,

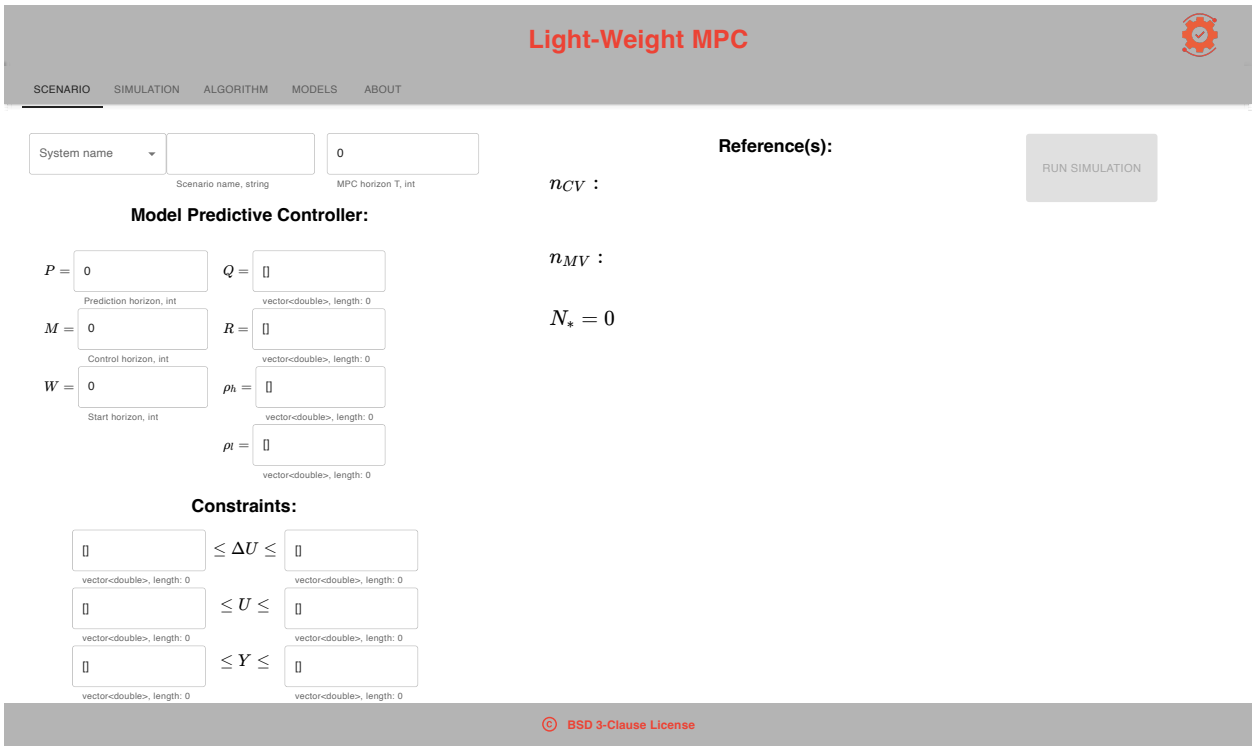


Figure 11: The front page of the *Web-application*. The page consists of interactive React components: A header with a menu bar, a body tailored to the simulator and a footer providing the software licensing. The app is running at a local port in a *Safari* browser.

providing the title and a logo. Correspondingly, the Footer lies at the bottom of the page, providing the software licence. The app functionality is found inside the Body component, inside a menu bar. This bar displays five different modules for the user to select and explore. The front page aims to define the scenario file needed to run the simulator. Hence, the current menu module is called correspondingly. As seen on the page, multiple *TextFields* appear on the left side. These are interactive components, which can store user input. During simulation, the information is serialized into a scenario file and routed to the simulator logic. For simplicity, a default scenario is provided in the application after having selected one of the available models in the local database. Furthermore, the MVs and CVs present in the model are displayed under the "Reference(s)" section along with the corresponding measurement units in order to provide reasonable controller references. This is a quite neat feature of the software since the end user is able to get some insight into the model before attempting to fit a controller tuning. At the far right, a simulation button is located. By pressing this button, a callback function is activated, running the underlying *MPC-simulator* in the browser. However, this button is disabled if the provided input yields an invalid controller scenario. Every *TextField* input is checked if a change occurs, updating each component's error state. A component turns red if its input is invalid for simulation matters. Eventually, when all components yield a valid simulation, the simulation button turns green and a simulation can be produced. This error checking is important for the UX, providing continuous feedback to the user during the engagement. The error-checking criteria for each controller parameter are stated in Table 5.

Table 5: Overview of the error-checking criteria for each controller parameter.

Parameter	Criteria
P	$P \in [M, N_*]$
M	$M \in [W, P]$
W	$W \in [0, M]$
T	$T > 0$
$\boldsymbol{Q}$	$\boldsymbol{Q} \geq 0, \quad \boldsymbol{Q}.length = n_{CV}$
$\boldsymbol{R}$	$\boldsymbol{R} \geq 0, \quad \boldsymbol{R}.length = n_{MV}$
ρ_h	$\rho_h \geq 0, \quad \rho_h.length = n_{CV}$
ρ_l	$\rho_l \geq 0, \quad \rho_l.length = n_{CV}$
$\overline{z}$	$\overline{z}_i > \underline{z}_i, \quad i \in \{0, \ldots, 2 \cdot n_{MV} + n_{CV}\} \quad \overline{z}.length = 2 \cdot n_{MV} + n_{CV}$
$\underline{z}$	$\underline{z}_i < \overline{z}_i, \quad i \in \{0, \ldots, 2 \cdot n_{MV} + n_{CV}\} \quad \underline{z}.length = 2 \cdot n_{MV} + n_{CV}$

Figure 12: Simulation module: No simulation results are available for display.

The remaining modules, which can be accessed in the menu bar, are named: *Simulation*, *Algorithm*, *Models* and *About*. Upon rendering these modules in the application, a webpage is displayed. Which page to be displayed is user-defined by interfacing the menu bar. Navigating to the simulation page will display one of two outcomes. Depending on if a simulation has been produced or not, the simulation page displays a simulation result or none. Figure 12 shows the simulation page given no simulation is available. In the other case, the simulation page mimics the plotting format interfacing with the simulator using the *Plotly* library outline in Section 3.2.4. Usually, it takes some time to run the simulator from the app, causing the user to wait. As previously elaborated the run time is dependent on the size of the model description, as described. In order to not let the user wait unknowingly if the application responded to the simulation call or not, a loading page is displayed. Figure 34 shows the layout of this page.

In the given use case, when a user does not achieve acceptable controller performance, they

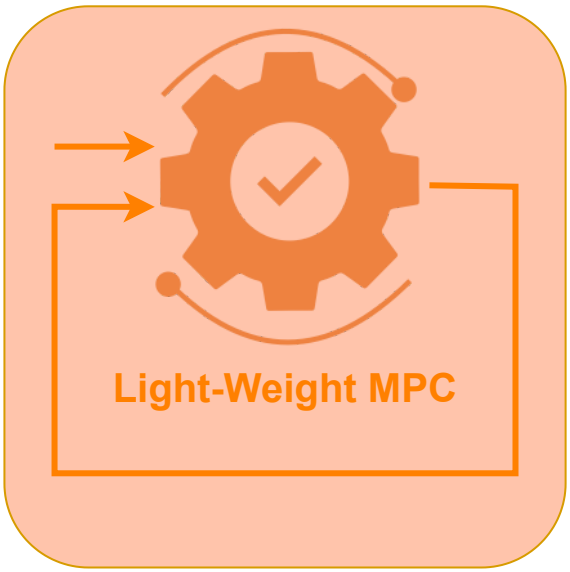


Figure 13: *Light-Weight MPC* logo. The logo is used for the *GitHub organization*.

may remain in this unsatisfactory condition. It might not be intuitive how to approach this further. Therefore, the UI is also designed to provide informative modules covering the controller principle. By rendering these pages the end user can deepen their knowledge about predictive controllers and use the test models to try out theories in simulations. The module *Models*, available in the menu bar, presents the general FSRM description and how this module is used to predict future simulation steps. Additionally, the *Algorithm* module describes how this model is used in the simulated controller algorithm in order to achieve controller objectives. The different algorithms along with tuning principles are also described, such that the end user may succeed in their search for tuning parameters. Designing these modules and integrating them into the menu bar apprise the application's academic perspective. These modules are thoroughly described in the appendix, Section F.

3.2.6 Distribution

The *Light-Weight MPC* simulation pipeline is open-source distributed under the terms of the BSD 3-Clause license. This document provides legally binding guidelines for the use and distribution of the code base. In order to combine the simulation software into an entity, a *GitHub organization* under the name *Light-Weight-MPC* was made. Within this organization three repositories are available for download: *MPC-simulator*, *Web-application* and *Light-Weight-MPC*. The two first repositories are described respectively in Section 3.1 and Section 3.2. The final repository is a code base for development purposes if it is desired to redistribute the application. The organization can be accessed through this link: https://github.com/orgs/Light-weight-MPC/repositories. Additionally, as a means to brand the software, a logo was made. Figure 13 illustrates the logo.

4 Software testing and results

4.1 Control models

In order to test different aspects of the simulation software, several model definitions are needed in order to exploit different functionalities. For this purpose, two models are derived and used for MPC simulations in the further sections.

4.1.1 *first_order_model*

The *first_order_model* is an asymptotically stable mathematical model often found within the process industry. As the name states, the model describes a first-order ordinary differential equation, yielding a SISO system description. In general, step-response coefficients can be obtained either empirically, by machine learning techniques or analytically from the transfer function. Due to the simple structure of the model, the coefficients can easily be derived analytically: Assume a first-order model with a time delay present. The transfer function expressing the response between an input, u and output y can be represented mathematically:

$$\frac{y(s)}{u(s)} = \frac{ke^{-\theta s}}{\tau s + 1}, \quad s = \frac{d}{dt}. \tag{51}$$

The parameters k, τ and θ respectively denote the gain, the time constant and the time delay present in the model. The variable s is a complex variable appearing due to the *Laplace transformation*. Based on the description (51), the coefficients can be derived by calculating the step-response for a sequence of time steps Δt. Calculating the inverse *Laplace transform* of the expression, setting $u(s) = \delta(t - \theta)$, where $\delta(t)$ denotes the *Heaviside step function*, and the coefficients can be described as ([25], Equation 20-4),

$$\left.\begin{array}{ll} s_i = 0 & \text{for } i\Delta t \leq \theta \\ s_i = k\left(1 - e^{-(i\Delta t - \theta)/\tau}\right) & \text{for } i\Delta t > \theta \end{array}\right\}. \tag{52}$$

Here $t = i\Delta t$. $i = 1, 2, \ldots, N$ such that $t_s = N\Delta t = 5\tau + \theta$ is considered the process settling time. Hence, $N = \frac{5\tau+\theta}{\Delta t}$ [25]. Most importantly is that N is chosen large enough such that $s_{N+1} \approx s_N$.

The *first_order_model* is derived using $k = 5, \tau = 15$ min and $\theta = 3$ min. In order to calculate the model horizon needed in order to cover the process settling time, the sampling time Δt needs to be decided. For simplicity, $\Delta t = 1$ min such that each simulation step corresponds to this step in time. If this plant is to be controlled using an MPC-FSRM, an MPC horizon $T = 10$ corresponds to 10 min. Inserting the specified values into Equation (10) yields a model horizon $N = 80$. Hence, in order to simulate the settling time of the system an MPC horizon, $T \geq N = 80$ is needed. There is a lot of research tied to this model description, both in the matter of control and system identification. If one wishes to control a plant but does not possess any corresponding model description. If the system data assembles a first-order system, Skogestad's system identification procedure

[26] can be utilized. The simple internal mode control (SIMC) method also describes a PID tuning approach to control the plant. If the system under consideration exhibits a MIMO configuration, the system identification approach can also be applied to identify and model the MIMO system. Consider a MIMO plant, if one excites one input with a unit step while simultaneously holding all other input constants, one can use this open-loop data to derive an FSRM. This holds given that the responses assemble first-order systems. Taking this approach into account, the simplistic model has a far greater scope of application than first guessed.

4.1.2 *SingleWell*

As stated in the theory, Section 2, the Model Predictive Controller approach is most desirable for complex slow multi-variable system description, where small improvements in performance can result in considerable increases in profit [23]. Clearly, the *first_order_model* is too simplistic to cover this use case. Simulating a SISO system description is also far easier in comparison to a MIMO. To test the simulation software on a more complex model, the *SingleWell* model was derived. *SingleWell* describes mathematically a single oil pipe and the process of leading the flow of oil and gas through the sea floor. The model is a MIMO system description having oil and gas rates as output. These rates or CVs are controlled using two valves or MVs. The first MV is the choke which controls the oil production rate on the upper end of the well. The production rates occur due to the pressure difference between the reservoir and the valves. In order to control the oil rate gas lift is injected as an artificial-lift method to reduce the hydrostatic pressure [9]. The resulting bottom-hole pressure reduction can be an enabler for the production allowing higher production rates. However, the gas lift used to increase oil production is costly and should be applied with minimal waste. The *SingleWell* model is a suitable use case for MPC, optimizing the use of this resource. Figure 14 is an illustration of the *SingleWell* plant. By employing the MPC approach, set points can be specified for the controller, along with appropriate tuning parameters that align with the desired cost function. The controller subsequently ensures the optimal production of set points, adhering to the defined objectives.

The model is originally implemented by *Equinor's* research team using Functional Mock-up Interface (FMU) [9]. To acquire the step-response coefficients for the FMU model, simulations were conducted using the method outlined in Section 4.1.1. This process resulted in a MIMO system description with a model horizon $N = 180$. Due to the two valves present, the model has an underlying nonlinear dynamic. Therefore, the step-response coefficients obtained are not an exact representation of the plant since the coefficients assemble a linearized system. As illustrated in Figure 14, the gas lift valve is a controlled valve which can determine the valve opening based on a targeted flow measured in m^3/h. The choke, on the other hand, is a normal valve with an opening ranging from $[0, 100]\%$. Both of the controlled variable, being the oil and gas rates are measured in m^3/h.

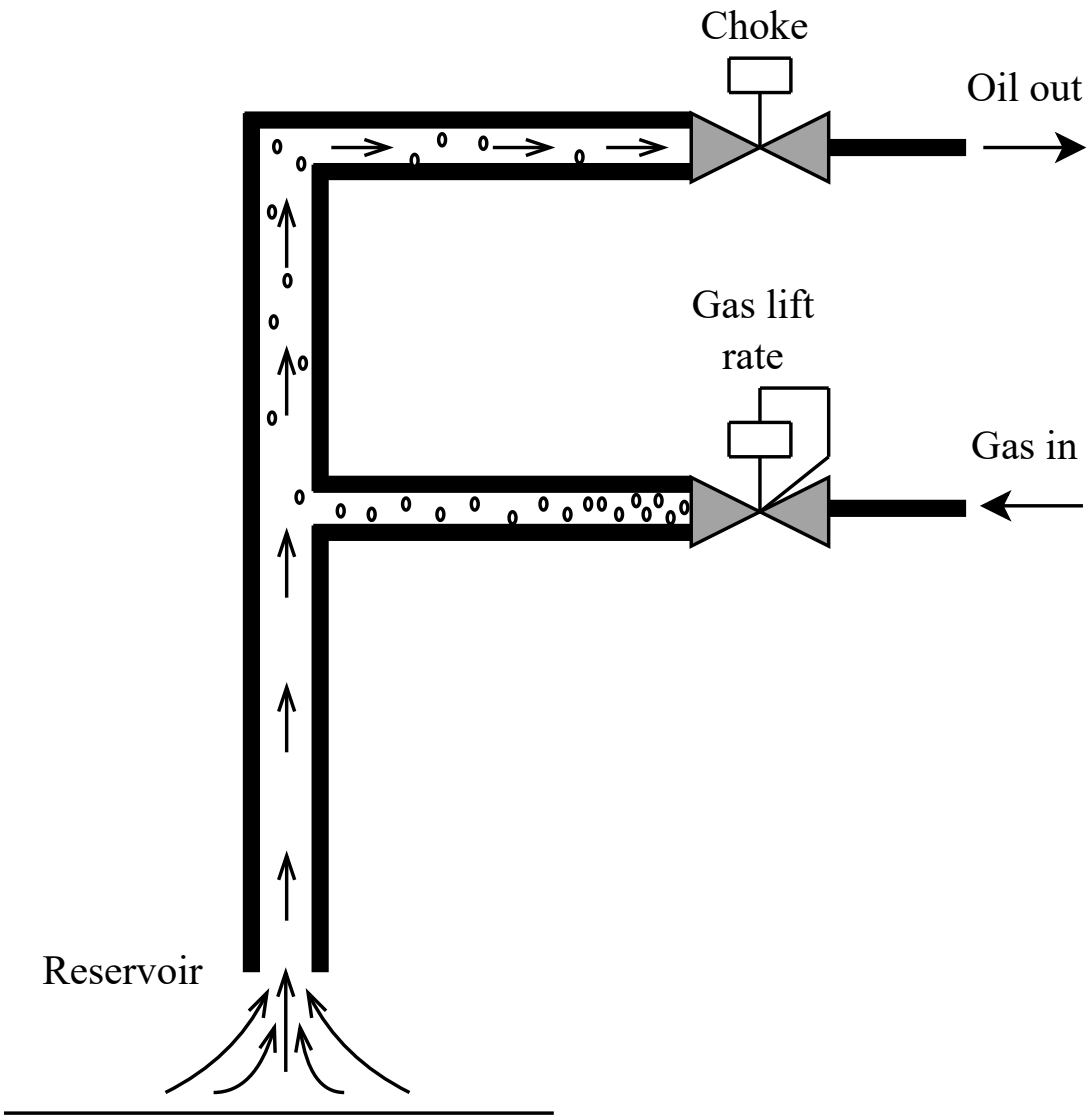


Figure 14: Illustration of the oil pipe *SingleWell* model. Oil is pumped from a reservoir under the sea floor. In order to control the oil rate through the well, gas lift is injected providing an artificial lift.

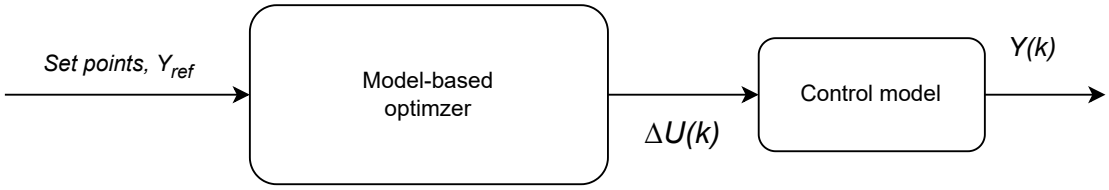


Figure 15: *Light-Weight MPC* simulation loop

4.2 *MPC-simulator* testing and results

As covered in the project thesis [29], the backend has been tested on a simple SISO system description. Those results were concluded to approve the implementation of a simpler version of the MPC algorithm (43). This implementation did not cover the usage of slack constraints nor bias correction in the control law. Since then, the *MPC-simulator* software has been updated and improved. However, due to time constraints simulating a real-world plant description used for bias correction is still left to be implemented. Therefore, the implemented simulation loop does not have any output feedback present, correcting the model errors. Nor is there any reliable model present to point out if errors even occur. Hence, the *Light-Weight MPC* software assumes a perfect control model identical to the plant and a simulation loop illustrated in Figure 15. This new controller implementation needs to be tested to a greater extent due to the advancement in functionality. Luckily, having derived the two FSRMs, *first_order_model* and *SingleWell*, it is possible to test a larger group of scenarios.

Assume the simulator is used to address Model Predictive Control on the *first_order_model*. MPC scenarios are defined by editing the tuning variables for the JSON files inside the *scenarios* folder. Since the addressed model has $N = 80$, it is sensible to set the prediction horizon, $P = N$, yielding $T \geq P$ in order to predict the dominant dynamics. To reduce the run-time memory, M is chosen such that $M \leq P$. By defining the rest of the tuning parameters as identity matrices and setting $W = 0$, only the inequality constraints are left before performing a simulation. Consider the scenario shown in the JSON file, Figure 16, simulated on a Linux distributed system. In order to call the simulator, the *Conda* environment must first be imported and then activated in the CLI: *conda activate linux*. One can call the simulator performing a simulation for an MPC horizon $T = 120$, where the MPC aims to control the plant to a reference $\mathcal{T} = 2$. In the CLI this call translates to *sh make.sh -T* 120 *-s first_order_model -r* [2] *-n* using the flags described in Table 3.

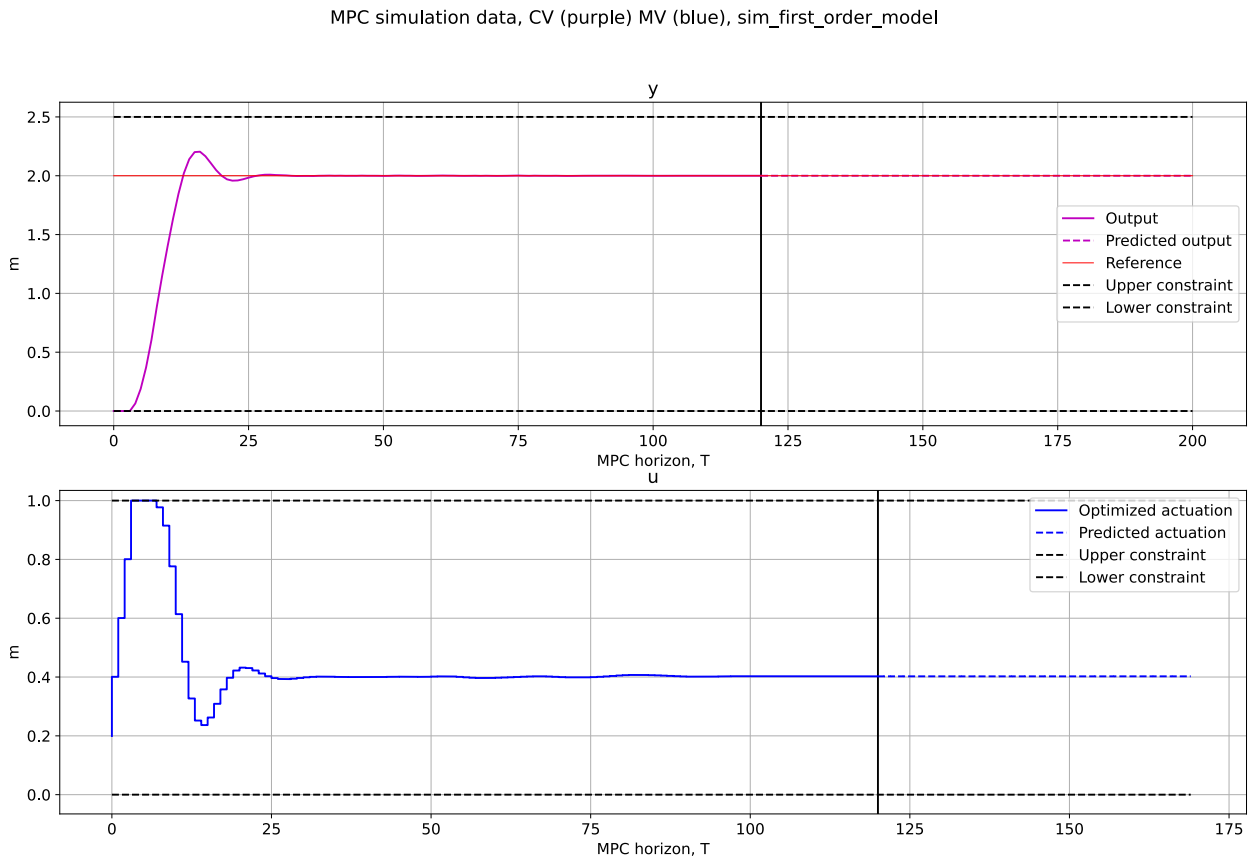


Figure 17: Simulation $T = 120$ and $\mathcal{T} = 2$ on the *first_order_model* plant with a controller description shown in Figure 16.

```
{   "system": "first_order_model",
    "MPC": { "P": 80,
             "M": 50,
             "W": 0,
             "Q": [1],
             "R": [1],
             "RoH": [1],
             "RoL": [1] },
    "c": [
      {"du": [-0.2, 0.2]},
      {"u": [0, 1]},
      {"y": [0, 2.5]} ]
}
```

Figure 16: *first_order_model* scenario file.

In order to analyse the simulation, the visualization tool described in Section 3.1.6 needs to be called. This is achieved by feeding *python3 vis/plot.py -s sim_first_order_model* into the CLI, given that the simulation file was stored with the name *sim_first_order_model.json*. The resulting simulation data is shown in Figure 17. Clearly, the simulation data shows a working Model Predictive Controller managing to control the mathematical model to a given reference within a total number of 25 simulation steps. However, an oscillating response combined with a saturating actuation is not a satisfactory controller performance. Luckily, the software is designed to rapidly consider new scenarios by recalling the software.

In order to achieve better controller performance, one might decide to change the tuning variables $\boldsymbol{Q}$ and $\boldsymbol{R}$. Additionally, one can reduce the simulation time by changing the start horizon W and disregarding the use of slack variables. Slack variables can be set aside since none of the controlled variable constraints is active. By defining such a MPC scenario, the condensed formulation without slack constraints is simulated (57). Hence, ρ_h and ρ_l are undefined vectors, in line with the description of the *MPC-simulator*,Section 3.1.5. Furthermore, as covered in Section 4.1.1, it is reasonable to set $W > 0$ as there is a time delay present in the model. This property is also seen by addressing the graphs. Running the simulator with controller parameters changed to $W = 10$ and $\boldsymbol{Q} = 200$ the output oscillations are damped. This can be seen in the simulation results from Figure 18. Since slack constraints were disabled, a smaller QP was solved during this simulation. Hence, the simulation time was decreased. In order to check if this tuning is resilient to repetitive changes in reference values, one can disable the new simulation flag and simulate the previous scenario for a longer MPC horizon having the reference changed. The total number of simulation steps using this simulator configuration yields $T = 120 + 120 + 80 = 320$. The first 120 steps are taken from the previous simulation, while the latter 200 prediction steps are the results of the new simulation using the changed reference value, $\mathcal{T} = 1$. The simulation result is shown in Figure 19.

A new feature, which was not implemented during the pre-project, is the use of slack constraints in the simulation. In order to validate this functionality, the same scenario can be simulated apart from changing the upper constraint to the value of 1.75. Since the reference yields a value of 2 the controller is forced to break the upper constraint in order to achieve the controller objective. The controller performance, Figure 20, is in line with the theory behind slack variables. The output error penalty, determined by $\boldsymbol{Q} = 200$, is severely larger than the penalty of exceeding the upper constraint, $\rho_h = 1$. The simulation results indicate a softened upper constraint. In this scenario, $W = 0$, in order to avoid an oscillating response when the controller exceeds the constraint. Conversely, the constraint can be hardened by increasing the slack penalty. The simulation result setting $\boldsymbol{Q} = 1$ and $\rho_h = 2000$ is shown in Figure 21.

After having tested the different functionalities provided using the condensed controller definition (43) on the theoretical *first_order_model*, one can try the simulator on a model definition used for industrial applications. The *SingleWell* model covered in Section 4.1.2 describes one oil well where the oil rate is controlled by applying an artificial lift by injecting external gas into the well. Assume that an end user wants to control this process optimally, utilizing *Light-Weight MPC* to address the issue. The model's open-loop response is simulated as a starting point to find a suitable controller definition. This information can be valuable for understanding the dynamics and identifying potential challenges. For

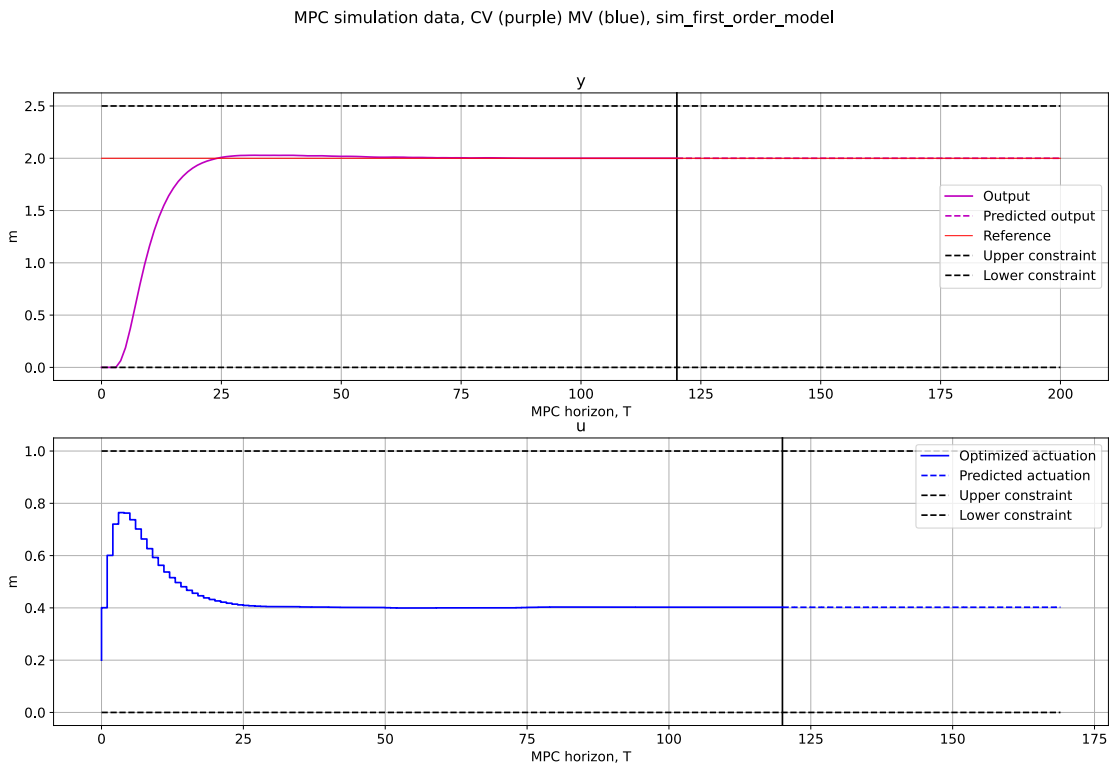


Figure 18: Simulation results considering time delay and output oscillations in the controller tuning.

instance, finding suitable constraint values can be hard to obtain. Certainly, the controller can endure a small output constraint subset having slack constraints present in the optimization problem. However, this is not the case for the constraints regarding the controller actuation. If these values are too heavily constrained, it might be infeasible to fulfil the criteria for reaching the reference value. Consequently, the controller must be tuned to handle these challenges. The open-loop simulation functionality is a neat feature when assessing MIMO systems.

From the model description, outlined in Section 4.1.2, the first manipulated variable could only take on values in the range from $[0, 100]\%$. Therefore, it is sensible to add this property as an inequality constraint. Consequently, one can assume that the valve cannot open faster than 2% in each simulation step to penalise the corresponding change in actuation. By defining the gas lift rate as a response increasing by $25\,\mathrm{m}^3/\mathrm{h}$ for each simulation step up until the final value of $1000\,\mathrm{m}^3/\mathrm{h}$, the response is shown in Figure 22. By inspection, the plot shows an equilibrium at approximately $3900\,\mathrm{m}^3/\mathrm{h}$ and $72\,\mathrm{m}^3/\mathrm{h}$ respectively the gas and oil rate. As the use of artificial gas lift is rather expensive, one can apply a Model Predictive Controller to reduce this cost while simultaneously maintaining the oil rate.

In order to exploit the applicability the controller has on this issue a simulation using identity penalties is produced. In an attempt of reducing the artificial gas the set-point in this simulation are respectively $3800\,\mathrm{m}^3/\mathrm{h}$ and $72\,\mathrm{m}^3/\mathrm{h}$. The resulting simulation is shown in Figure 23. Since the model horizon, $N = 180$, the prediction horizon is chosen correspondingly. Since there is no time delay present in the control model description,

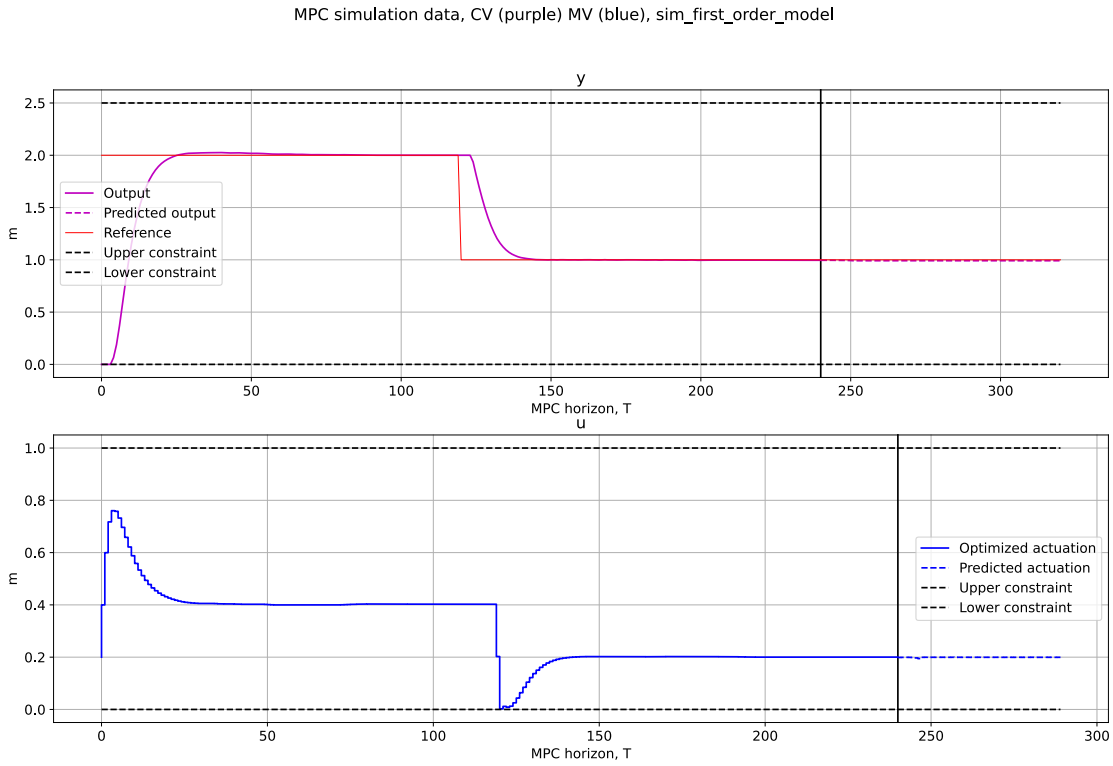


Figure 19: Continuation of simulation 18 with a changed reference value. This functionality can be used to address the performance under a changing simulation environment.

W takes zero value. The simulation plots reveal a working controller solving the tracking problem applied. As seen in the lower left plot, the choke reaches maximum opening quite early in the simulation. This response in combination with the injected gas lift yields an overshoot in the oil rate response. This overshoot is rather undesired as it is advantageous to have a smooth control law. In order to improve the simulation, the tuning parameters described in the JSON file, Figure 25, are used. In this scenario, slack constraints are disabled since the controlled variables are not sufficiently close enough to the constraints to have an impact on the simulation. The tuned Model Predictive Control simulation is displayed in Figure 24. In line with the theory, increasing the output-error penalty, identified as the elements in the $\boldsymbol{Q}$-matrix, will produce a dampened response. Additionally, by also tuning in the $\boldsymbol{R}$-matrix, the penalty on the gas lift can be utterly inflicted. Sadly, the controller tuning yielding the smooth oil rate response affects the tracking of the gas rate due to the coupled model dynamics.

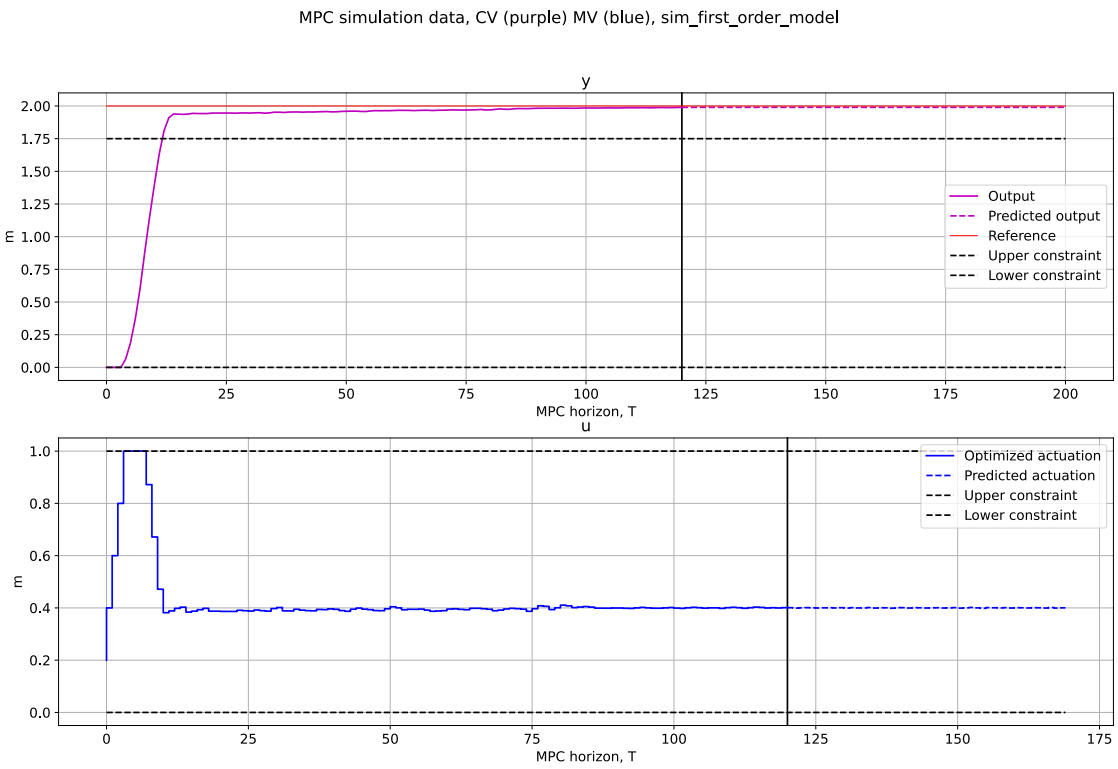


Figure 20: Simulating *first_order_model* determining the reference above the upper output constraint. Output error penalty, $\boldsymbol{Q} = 200$ and upper slack penalty, $\rho_h = 1$.

```
{   "system": "SingleWell",
    "MPC": { "P": 180,
             "M": 120,
             "W": 0,
             "Q": [1, 15000],
             "R": [1, 5000],
             "RoH": [],
             "RoL": [] },
    "c": [
      {"du[1]": [-2, 2]}, {"du[2]": [-10, 10]},
      {"u[1]": [0, 100]}, {"u[2]": [0, 1000]},
      {"y[1]": [0, 4000]}, {"y[2]": [0, 100]} ]
}
```

Figure 25: *SingleWell* scenario file used for testing.

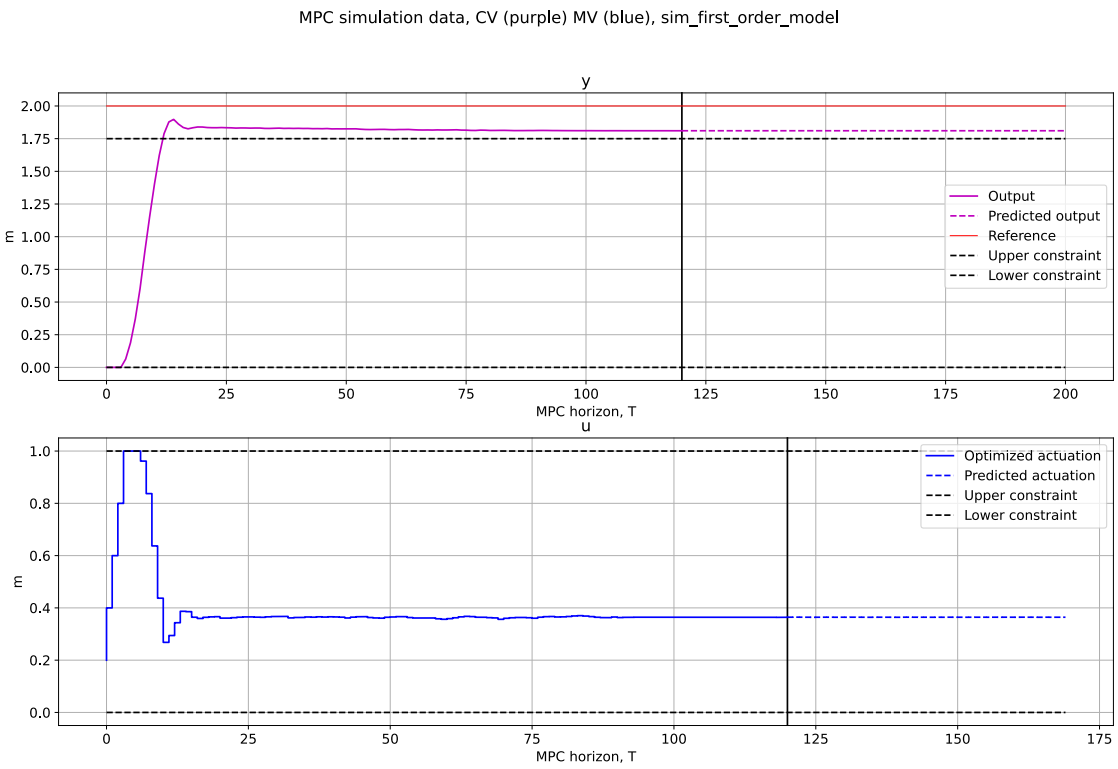


Figure 21: Hardening the upper constraint to validate a correctly implemented slack variable. Output error penalty, $\boldsymbol{Q} = 1$ and upper slack penalty, $\rho_h = 2000$.

4.3 *Web-application* testing and results

The purpose behind the *Web-application* software is to provide a neat interface to the simulator to enhance the automation of the production of simulation data. As covered in Section 3.2, the software is cross-platform and reachable on a diversity of devices as long as an Internet connection is available. With *Node.js* installed, the application can be deployed as described in the implementation section. After deployment, the front page, Figure 11, is the first the user encounters. Assume the user wants to perform the same simulations as produced in Section 4.2. The first step is to define the control model to be simulated. Hence, the user navigates to the *System name* input bar. The available model descriptions stored in the local database are displayed in a drop-down menu as shown in Figure 26. As expected, the two models described in Section 4.1 are present in the menu.

By selecting the *first_order_model* a default MPC tuning appears inside the different *Text-fields*. These tuning parameters are defined in the *default_tuning.json* file shown in the folder structure Figure 9. Additionally, the reference section, which previously appeared empty, has now changed. This is the feature mentioned in the implementation Section 3.2.5 providing the end user model parameters and dimensions. By feeding in the value $\mathcal{T} = 2$ in the reference input field, the scenario assembles the second *first_order_model* simulation covered in Section 4.2. The MPC-scenario is in line with the controller parameter criteria shown in Table 5. This conclusion can also be drawn from the green coloured *RUN SIMULATION* button located on the far right in the Figure 27. By pressing the button the

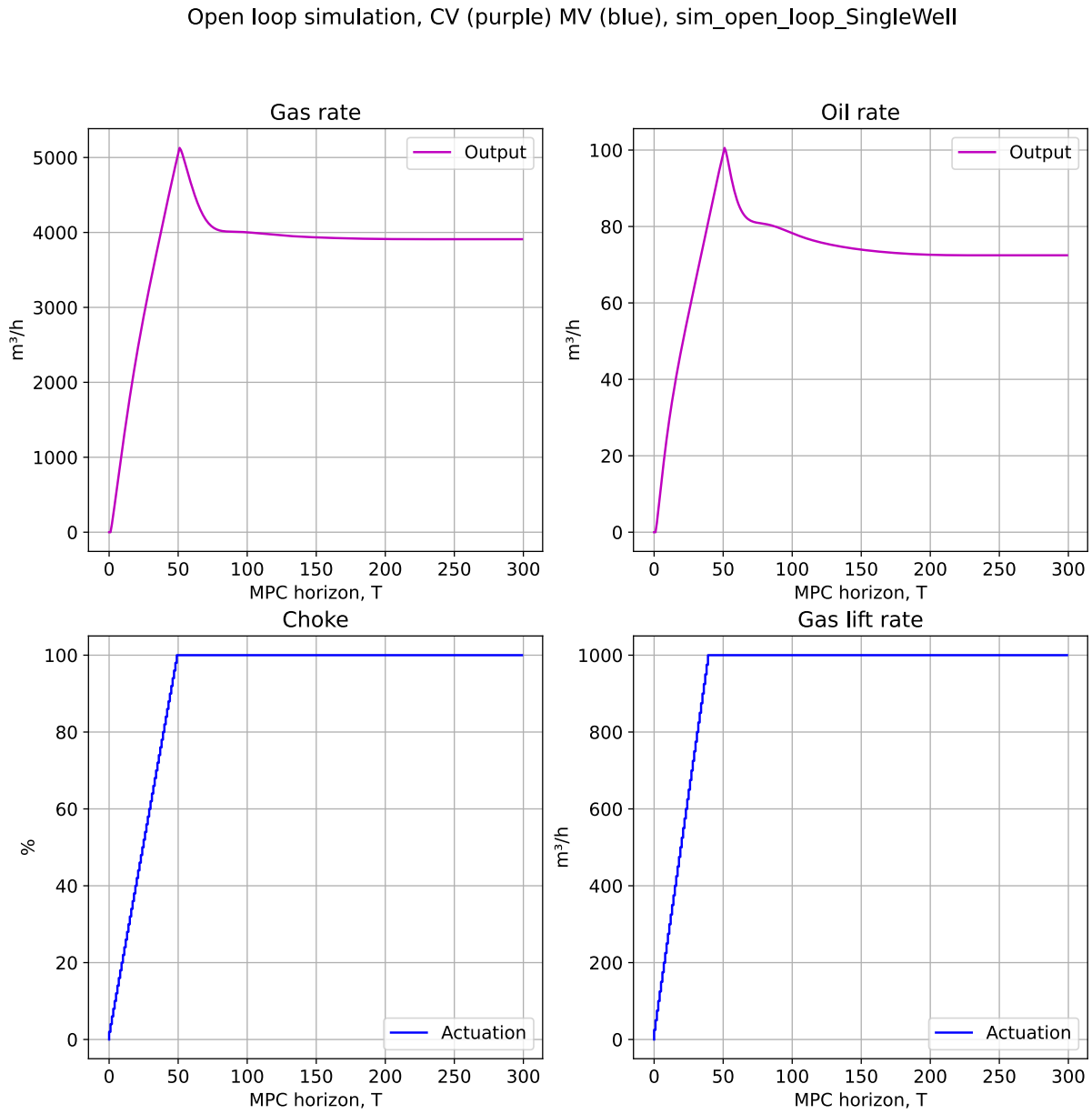


Figure 22: Open-loop simulation of the *SingleWell* model for $T = 300$. The control model reaches its steady state after around 200 simulation steps.

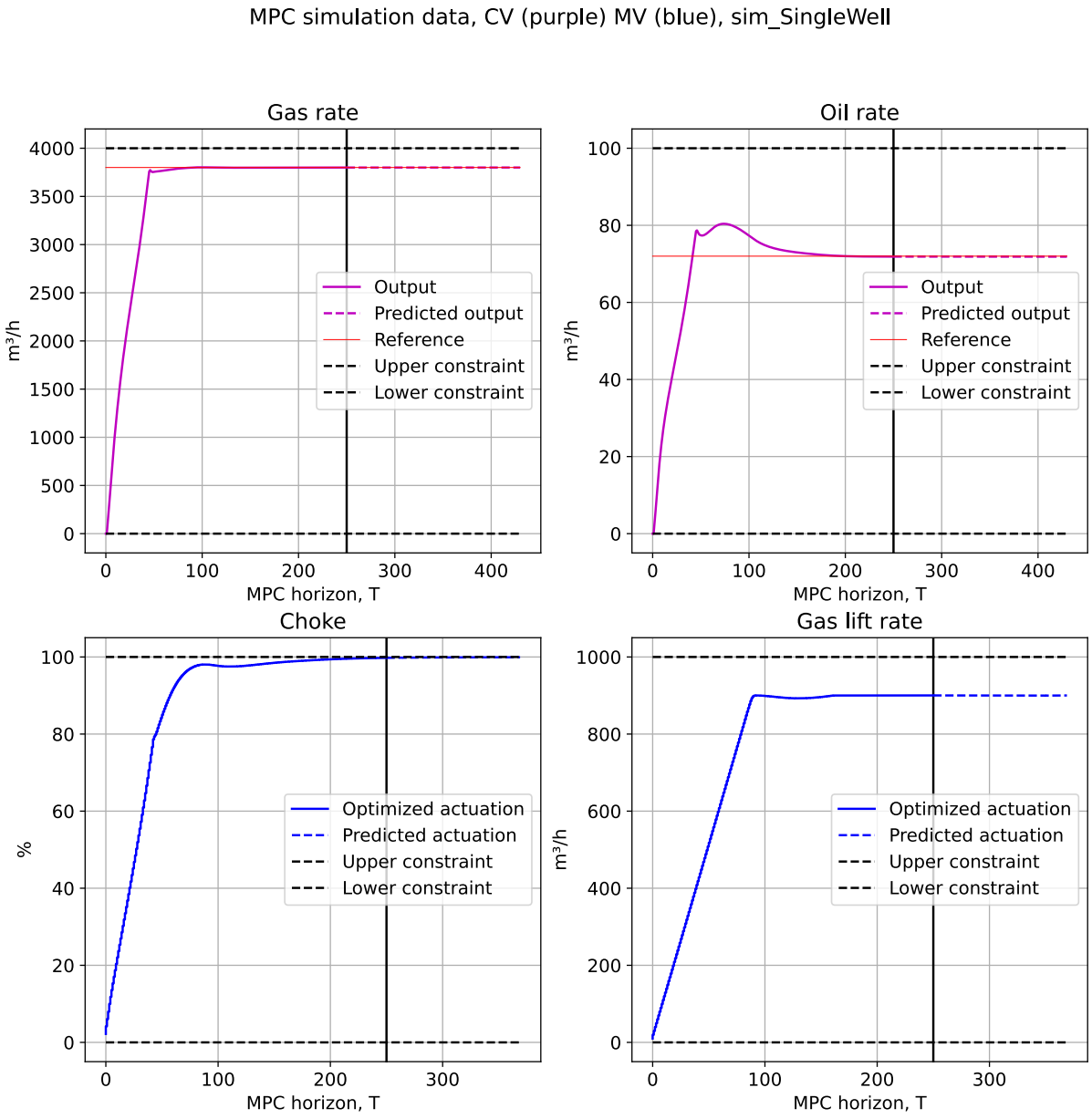


Figure 23: MPC simulation of the *SingleWell* model for $T = 250$, $P = 180$ and $M = 120$. The data shows that the simulated controller achieves the tracking problem while simultaneously reducing the use of artificial gas.

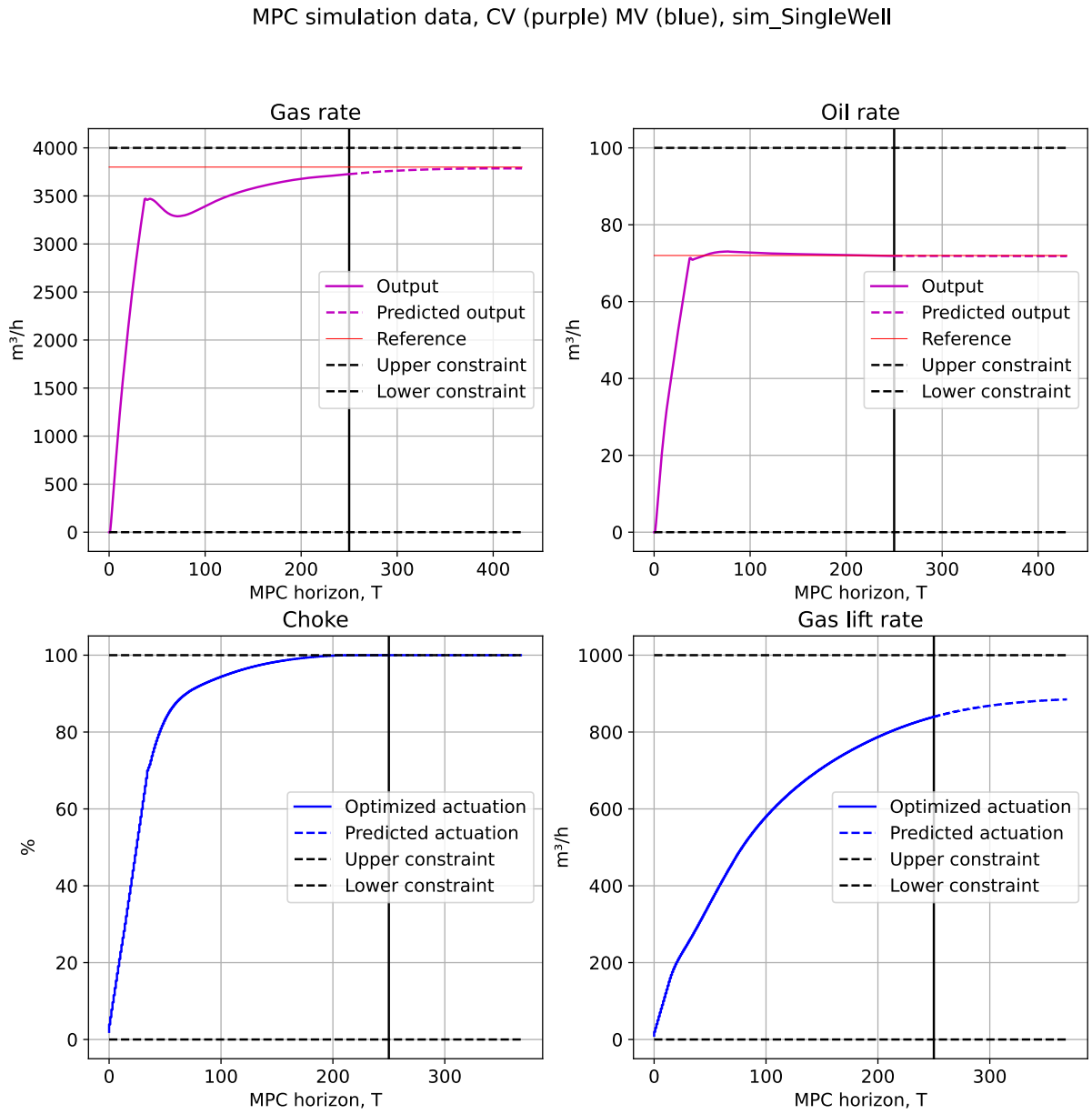


Figure 24: *SingleWell* simulation with an MPC horizon, $T = 250$. The tuning matrices are defined to optimize the oil rate response, $\boldsymbol{Q} = [1, 15000]$ and $\boldsymbol{R} = [1, 5000]$.

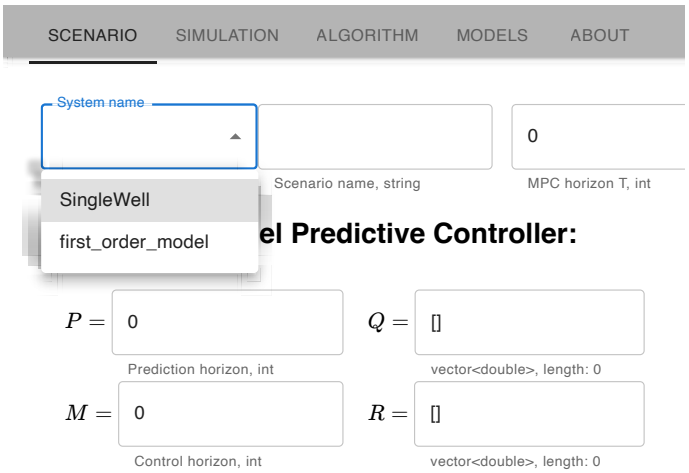


Figure 26: Model descriptions available: The application parses all models stored in the database for selection.

Wasm compiled *MPC-simulator* is called producing the simulator. While the end user waits on the simulator to return the data, the loading page, Figure 34, is displayed.

Assume another user wants to download the application to design an MPC with the aim to control the *SingleWell* model. The user has only Windows deceives available so he chooses to make use of the web application. After having spawned the application and selected the model, the user enters the same tuning parameters he had seen worked well on a SISO system. However, the simulation button is disabled preventing the software to perform simulations. The current state of the application could resemble the case shown in Figure 29. This error is caused due to the controller parameters failing to fulfil the criteria pointed out in Table 5. The specific parameters not passing the criteria check are marked out in red colour. These values cause the disability of the simulation button. After having sorted out all the parameter errors, the user decides to simulate the scenario. Even though the scenario entered is valid, simulation errors might occur for unknown reasons. For instance, when testing out a new model description, there might be some properties in the system file causing simulation faults. To test the application's ability to handle such errors, a scenario is tested on a model description with a wrongly defined parameter. Figure 30 shows the layout after the attempt.

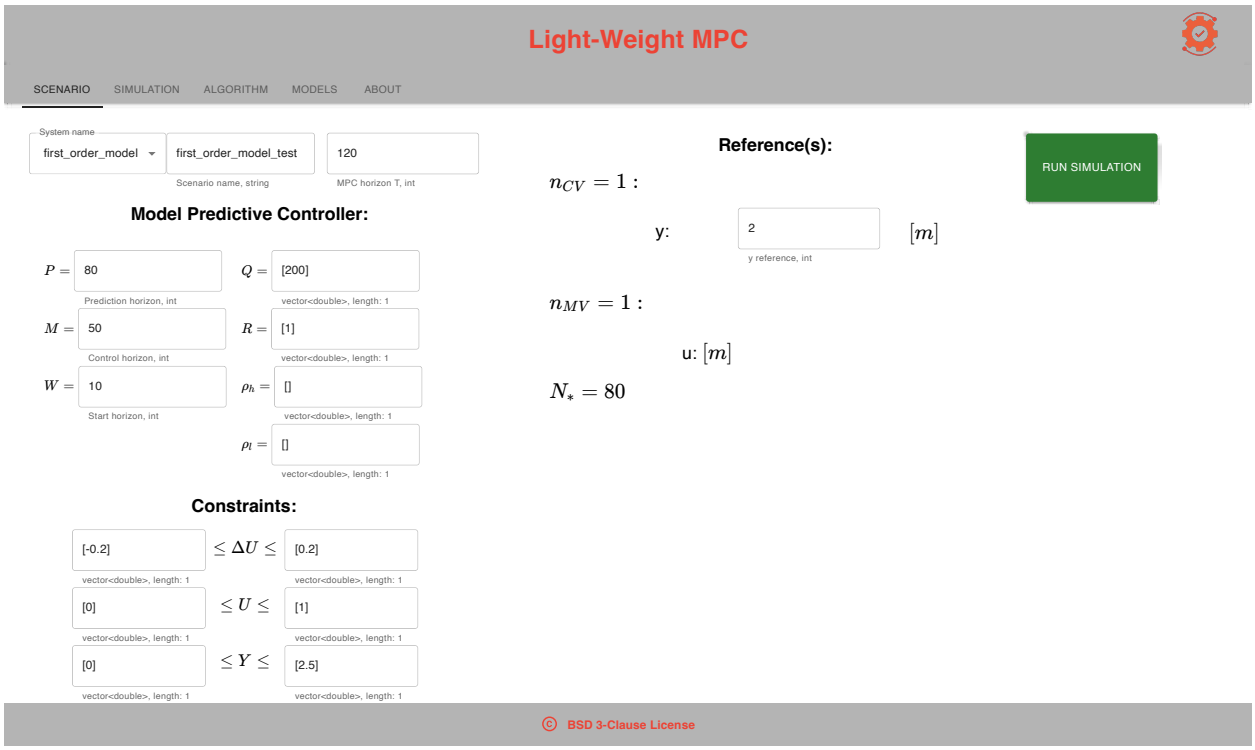


Figure 27: *Web-application* layout after having selected *first_order_model* as control model. Every field except the reference is defined by a default tuning. Feeding in a reference value combined with a valid scenario results in a green *RUN SIMULATION* button.

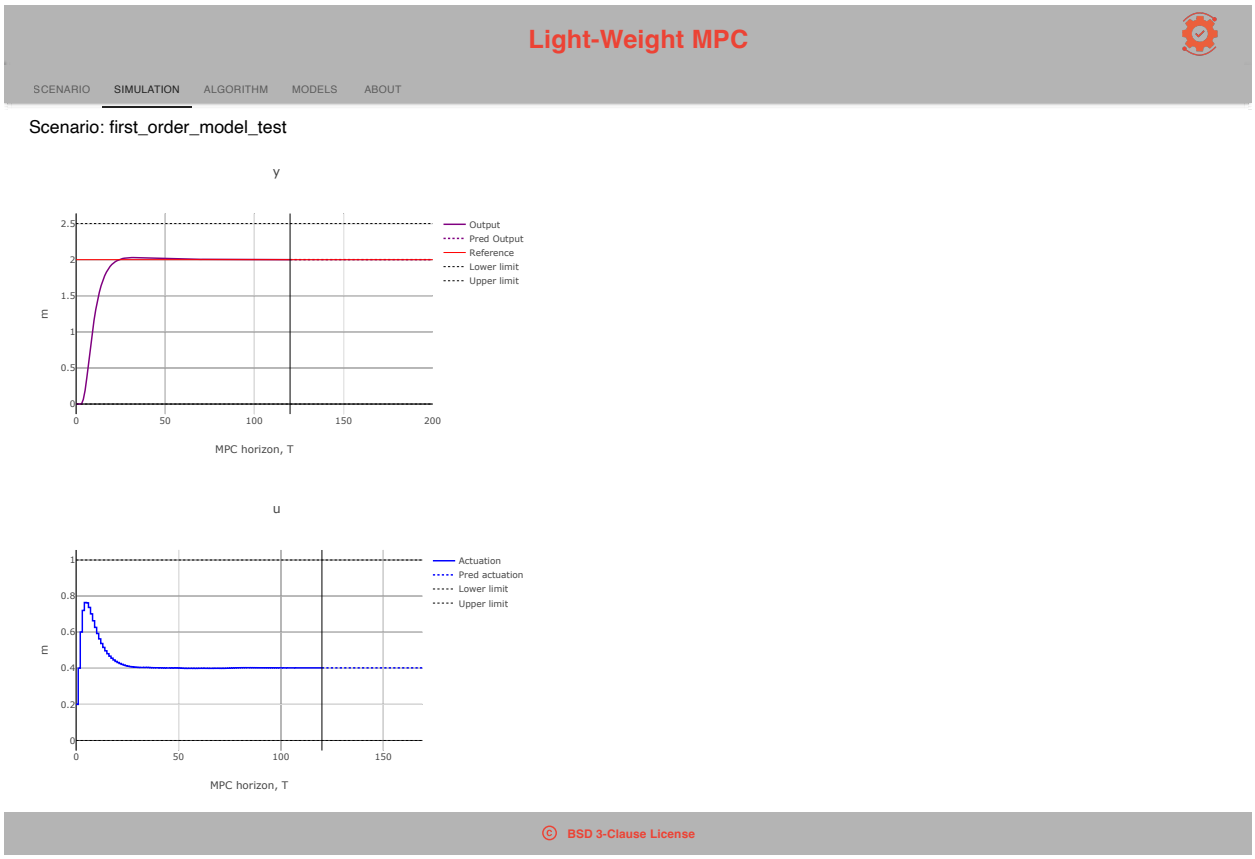


Figure 28: *Web-application* layout after simulating the default tuning corresponding to the selection of the *first_order_model* with a reference value $\mathcal{T} = 2$.

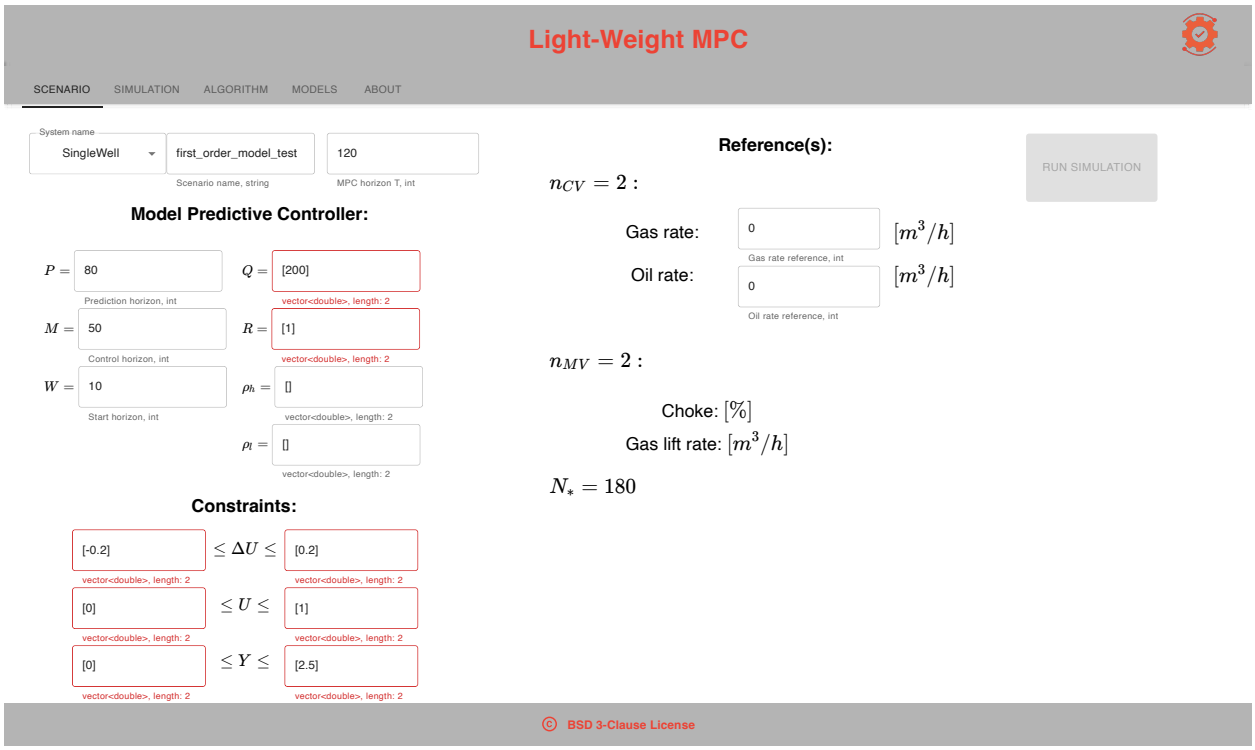


Figure 29: *Web-application* layout when the simulation criteria Table 5 is not fulfilled. The figure is taken from the application running on a *Windows* operating system.

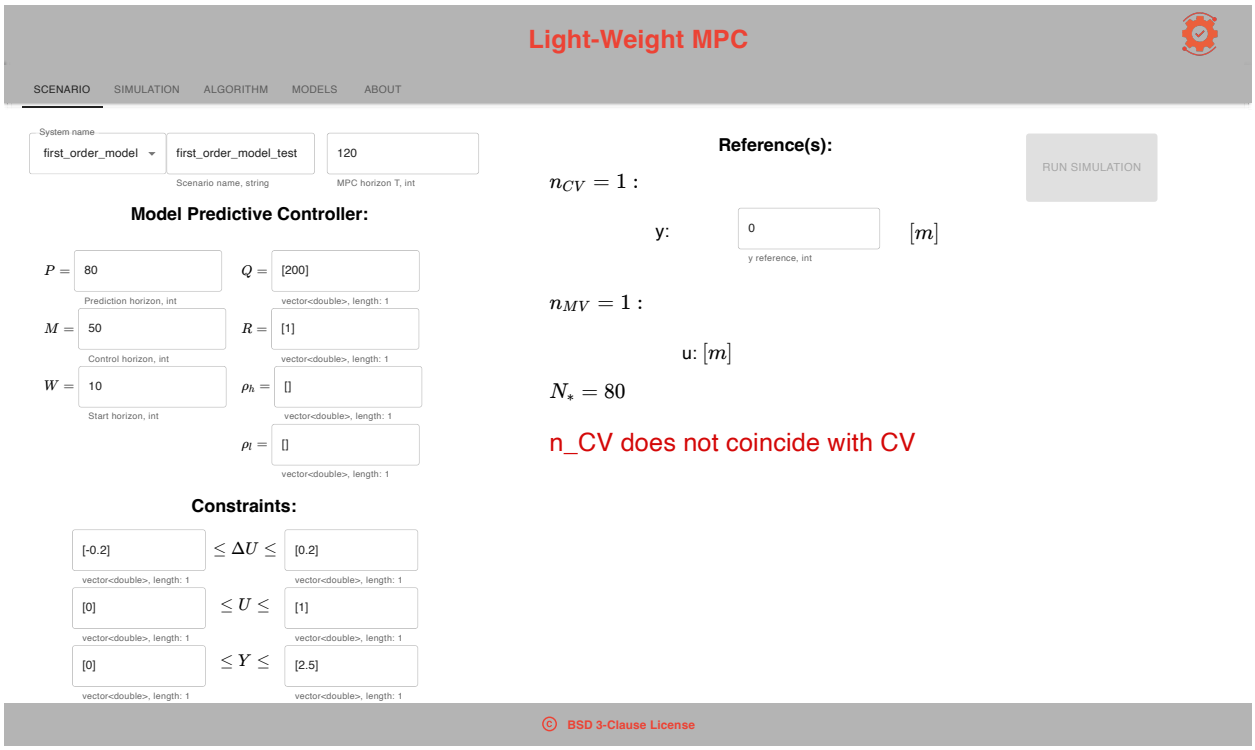


Figure 30: Scenario page layout after enforcing a simulation error. Since the implementation of *MPC-simulator* emphasises defensive programming, the detailed error message caught is displayed in red colour.

5 Discussion

Recall the *Light-Weight MPC* objectives stated in the problem description, Section 1.2. The main objective of the project is to implement a lightweight simulation tool for linear Model Predictive Control. Several components are needed to simulate a controller for this purpose. Firstly, the system description representing the control model must be inputted into the software. The control model is simulated alongside the MPC, which implicitly reuses the model to predict future behaviour. The principle behind the controller method along with the simulation loop was covered in the theory, Section 2, and illustrated in Figure 3. Furthermore, the controller defines the model dynamic based on a set of tuning parameters, which is the second type of data inputted to the software. Implementing a framework realising the simulation loop in a simple manner is what makes the software lightweight.

The addressed controller formulation was to target a typical process model and represent the core functionality of the control software SEPTIC. Hence, the chosen simulation loop was tailored to control challenges within the process industry. However, for different tasks and objectives, another controller definition or system description might be more suited. For instance, the current simulator implementation only addresses one cost function, even though there might be other costs more desirable. By modularizing the implementation, new functionality can be enhanced by reusing the original simulation loop.

Another important aspect, which needed to be accounted for, was the complexity of the controller. As the strategy is far more complex compared to classical approaches, the efficiency of the controller formulation needed to be addressed to achieve a lightweight design. Developing a realistic controller and simulation environment was yet another factor. This presented a more significant challenge than anticipated, although implementing the most advanced simulation software was never a part of the problem description. A software can only be as useful as its UI regardless of how many simulation aspects it accounts for. Therefore, energy was rather put into developing the web application. The application offers a customized interface for the simulator, streamlining the automation of simulation data. Instead of manually editing JSON files to modify controller parameters, the app provides interactive components such as drop-down menus, buttons, and text fields. This abstraction of the design dependency enhances user-friendliness by eliminating the need for manual read-and-write operations in an editor. Compared to the *MPC-simulator*, which is dependant on Unix systems, the implementation *Web-application* enhances the portability of the simulation software as well.

Even though the simulation setup works, there are simplifications made and functionality left out of the software. Only the simulation results can reveal the usability and applications of the software. The implementation details and results described in Section 4 are discussed in the further sections.

5.1 *MPC-simulator* discussion

5.1.1 Simulations

Recall the test scenarios described in Section 4.2 relating to the control of the *first_order_model*. Based on the simulation data outlined in this section, the controller behaves as expected considering the theory, Section 2.2. Regarding the first simulation Figure 17, the controller achieves getting the plant to track the specified reference $\mathcal{T} = 2$ in about 25 simulation steps. By addressing the plots, all constraints specified in the corresponding scenario file, Figure 16 are obeyed. As seen in the MV plot, the actuation reaches saturation at $U = 1.0$. Since the constraints impacting the actuation are hard, the controller cannot produce a control law exceeding this constraint. The controller manages to keep both signals within these boundaries and simultaneously achieve the controller objective.

This scenario illustrates one of the core benefits of using Model Predictive Control. In a practical scenario, such constraints can assure that actuators are not pressed to their far limit in order to reach a target. Repeated saturation of the equipment can lead to higher maintenance costs as the lifespan is reduced. Saturation may overheat the actuators and cause damage. The same goes for the change in actuation constraints, which also the controller manages to comply with. A too-large actuation step might not be suitable for the process actuators as they may have a preferred operating point. Another aspect of having hard constraints present in the controller formulation relates to the safety requirements of the operating plant. By assuring that constraints are held under any circumstances, reliability is maintained and safety risks are avoided.

Another aspect worth mentioning is the plot layout. By addressing the plots, these formats assemble the description covered in the section about the visualization tool, Section 3.1.6. The formats are designed to easily address if the controller violates any constraints, plotting the constraints at the upper and lower end. The vertical prediction axis appearing at the end of the MPC horizon, $T = 120$, is used to separate the simulation data from future predictions. Even though the simulation data is most important, the planned predictions can exploit the controller's intention. For instance, this can be used to validate the implementation of the controller, as it should always aim to stabilize the plant. If the predicted outputs are not approaching the reference, this might be an implementation error. This property also reflects the importance of a well-designed visualization tool, as it validates the controller principles.

As the previous response was not considered desirable enough due to the output overshoot, another set of tuning parameters was simulated. By tuning in a $W > 0$ and an output-error penalty $\boldsymbol{Q} = 200$, the performance was improved in the simulation shown in Figure 18. The simulation data shows a smoothed simulation due to an optimized use of actuation. Based on the theoretical background the optimizer should address the output error to a greater extent and reduce the damping. Hence, the result is in line with the theory. The empirically obtained controller was also tested in a changed simulation environment where the control objective shifted after $T = 120$.

The fourth and fifth scenarios, respectively in Figure 20 and 21, show the application of having slack constraints present in the controller formulation. As the MPC tries to fulfil a

tracking problem outside the feasible area (13), the plant will encounter a constraint. Forcing the controller outside its constraints without slack values leads to faulty simulations and useless data. This happens even though OSQP returns `OsqpEigen::ErrorExitFlag::NoError`. When the optimizer is not able to obtain a feasible solution, the simulation yields a destabilization of the plant. As covered in the theory, Section 2.2.7, the feasible set can be enlarged by slacking. In the case of trial and testing different scenarios, it is more useful for the user to obtain stabilizing simulations. Therefore, having the ability to enable slack constraints is a major simulator advancement. This functionality can also be used to identify if the constraints are too hard.

In order to mimic a realistic use case of the *Light-Weight MPC* software, the *SingleWell* model was simulated. The tests described in Section 4.2 utilize the open-loop functionality to find realistic set points to make controller objectives. As the dynamic is coupled in this MIMO system description, an arbitrary controller will not be able to track any chosen set of references. By assessing the open-loop simulation, the chosen reference values were respectively $3800\,\mathrm{m}^3/\mathrm{h}$ gas rate and $72\,\mathrm{m}^3/\mathrm{h}$ oil rate. By comparing the open-loop simulation with the best-obtained controller tuning simulation Figure 24, the MPC performs well on the fairly complex MIMO system. The controller approach manages to constrain the costly use gas lift while maintaining the oil rate control objective. This is achieved by penalizing the use of gas lift severely, $\boldsymbol{Q} = [1, 15000]$. The gas rate response has a small-scale oscillation. However, as seen in the predicted output the controller plans to stabilize the response after 300 simulation steps. While this outcome demonstrates the attainability of desired results with a Model Predictive Controller, utilizing a PID controller to achieve the same level of performance would pose a significantly greater challenge. The observed performance signifies an optimized behaviour and a reduction in waste, potentially resulting in substantial resource savings for a company.

5.1.2 Implementation

The *MPC-simulator* tests show a valid implementation of the MPC mathematically described in Equation (43). This formulation is derived with regard to the project description, implementing a lightweight simulation software. Different MPC features such as bias correction and slack constraints were addressed to make an industrial-like controller definition. Based on optimization techniques such as the *Nullspace method* the problem could be reduced to ensure efficient execution. In view of this property, the OSQP software was a clear choice as the preferred optimizer as it is the fastest commercial QP solver available. Other open-source optimizers, such as *qpOASES* [21], were also considered in the implementations. However, besides the efficiency aspect, most solvers considered a larger optimization problem, hence more simulation parameters would be needed. The existing C++ wrapper *Osqp-Eigen* made also simplified the choice of optimizer since this software abstracted optimizer setup into *Eigen* data types.

Besides the controller definition, the implementation also covers the control and plant model. As described in Figure 3 the simulator only takes the FSRM into account under simulation and not the real underlying system. Since the FSRM describes a linear exponentially stable model, this system description is a non-realistic model deviating from the true system. This control model was chosen due to its applicability in the process industry.

Additionally, the model predictions are fairly simple to calculate and account for in an MPC. By having a realistic model definition, bias correction could also be included in the controller yielding more realistic simulation scenarios. The current simulator assumes a perfect model, however, for practical control this is never a valid assumption. For instance, sensors are used in order to estimate the state of the model, and these measurements are noisy and uncertain. Obtaining such a model description has been challenging in the project. Moreover, simulating a complex model accurately requires a lot of computational power. In the case, the model is described by a set of ordinary differential equations, a suitable integrator software must be imported into the software. Due to the lack of model available model definitions, the induced computational overhead and time constraints, bias correction was left a future work.

Since the control model plays a central role in the simulation software, it was implemented as a C++ class. By having a class definition, one can instantiate multiple general FSRMs and define member functions performing calculations. Even though the *MPC-simulator* does not take bias correction into account, an interface is implemented as a public member function. The only aspect left to include bias correction is the calculation of the bias. If this is achieved, the bias can be fed directly into the simulation loop described in Algorithm 2. Since the simulator may require two different FSRM descriptions, dependent on the W value, the class definition assures the readability and reusability of the source code. Speaking of the implementation it was stated in the theory that the models were implemented using `Eigen::MatrixXd` and `Eigen::SparseMatrixXd` which are memory-efficient C++ types. Alongside the datatypes, the *Eigen* software also provided user-friendly syntax to perform mathematical operations such as transpose and inverse calculations.

One implementation detail worth to be discussed is the interpretation of the MPC horizon, T. From the current simulator implementation, the interpretation of the time step is model dependent. How the model horizon effectuates the MPC horizon is elaborated in Section 2.1.4. In an industrial use case, it might be more sensible to simulate a model in a time frame with the SI-unit second. If different Δt are to be addressed, this would imply importing different step-response representations of the same model. Therefore, it would have been a neat feature to derive the needed model at run-time and simulate it for a user-defined Δt. This functionality would have allowed the user to also test different FSRM representations for control. As modelling is not the scope of this project, the framework assumes a predefined model in addition to the user having the background information needed to interpret the simulation results.

5.2 *Web-application* discussion

5.2.1 Simulations

By deploying the application in a web browser, the layout is displayed as expected. However, when opening the app on a smaller device the layout may appear more dense with less space between the interactive *React*-components. Regardless of the density of the layout, the structure and underlying functionality are still intact. When selecting the desired control model, all available models pop up in a drop-down menu, shown in Figure 26. This

is a quite-user friendly feature securing the selection of a valid model description.

Furthermore, by selecting an available model a default tuning is automatically entered into the input fields. The default tuning parameters defined in the source code represents a valid simulation scenario. Such features are added in order to efficiently bring the user up to speed, interfacing with the application. By performing the simulation, and adding the value of 2 to the reference vector, the application navigates to the simulation module, displaying the data. Addressing the data produced using the application, Figure 17, and the simulation data visualized using the *MPC-simulator*, Figure 28, the data resembles. One can therefore conclude, that the porting of the underlying C++ code has been successful using the emcc in the pipeline. By validating this pipeline, the software can be ported and integrated into other professionally maintained software frameworks. The setup can in principle compile any C/C++ function to be accessible in a web application. This architecture deviates from traditional application development as the frontend does not depend on any server-side callbacks. Normally, the backend and frontend communicate through the HTTP protocol and socket programming. If data is requested, the frontend performs an HTTP request and becomes a corresponding response from the backend. This communication overhead is omitted with the use of Wasm as the application can access the needed functionality directly. Hence, a Wasm-based application can run in the browser in an efficient manner without any additional communication overhead. The only dependency is the installation of *Node.js*, which is open-source available on all OSs.

In the case where the application fails to produce simulation data or an invalid scenario is tried to be simulated, the user should be alerted. The implemented UI has an inbuilt error-checking capability assuring UX. Figure 29 shows the layout of the app when the controller parameters do not align with the criteria Table 5. Red markers appear around the related variables. This UX-design aim to provide specific feedback to the user in advance if the application fails to provide a service. However, simulation errors might not only be a cause of the entered scenario. Errors could lay in the porting of the simulator, the model definition or the simulator logic itself, etc. When unforeseen errors occur, informative feedback should also be displayed. As described in Section 3.1.7, the *MPC-simulator* has defence mechanisms such as try-catch statements to handle such errors. The application manages to catch these error messages and display them in the UI. Figure 30 illustrates the scenario when a model parameter is incorrectly defined. The end user then gets alerted where the lies and if possible which parameter to change. In this case, the model parameter n_{CV} did not coincide with the CV defined in the model. The error-handling mechanism benefits both the user, wanting to run simulations, and the programmer, wanting to validate the source code.

5.2.2 Implementation

Addressing the implementation, the use of JavaScript *React* and CRA were obvious choices. The syntax was easy to adapt to and *Material UI* provided an API with informative pages on how to express the targeted frontend design using code. By spawning the available development server, CRA handled the setup and provided error checking of the application. One other great feature, of using *React* is the support the framework has interfacing with web browsers and platforms.

Testing and improving the plant response is relatively easy and the exploration does not take a significant amount of time. This is a major benefit of using the application, as it simplifies the production of data. At least when the controller principle is known to the end user. As stated in the project background, Section 1.1, the controller is typically unrecognized by individuals due to its complexity. In order to operate under this user assumption, the website is equipped with informative pages covering the needed information to understand the control method. These pages are not enough to cover the principle from scratch, but point the user in the right direction. For instance, in the Algorithm module, Figure 36, relevant tuning approaches are outlined. This is valuable information in the event that the user struggle to improve controller performance. It could have been decided to include all background information behind an MPC like Section 2.2. However, pushing too much information out on one page will possibly counteract its purpose, as the user may skip important sections. Therefore, displaying only the core information needed to utilize the app was rather emphasised. Not to mention the computational overhead the first option would have caused, rendering a dozen of pages.

Since the simulator does not account for realistic models and the application has an UX designed for academic use. *Light-Weight MPC* may be at most applicable for teaching purposes. Since the core MPC functionality is present, downloading the application is a great starting point to learn MPC. The software allows enabling and disabling different functionality, such that focus can be drawn on specific control properties. In addition to the neat interface, the application is a perfect test setup to see the impact the controller has on the plant. As of now, the academically recognized *first_order_model* is already present in the software. This model is known to students since it is extensively used in control theory for system identification and PID control. Furthermore, due to the cross-platform nature of the software architecture, incorporating the *Light-Weight MPC* framework into university courses or trainee programs would pose no challenges. This integration would effectively broaden the exposure of controller strategies to a wider audience, thereby enhancing understanding and knowledge in this field.

6 Conclusion

Light-Weight MPC is a simulation software meant to assess the performance of an MPC on typical process step-response models. The simulations can be used to explore new MPC applications as well as to maintain and further develop existing solutions. Simulation results and control data can be obtained by running the *MPC-simulator* directly or spawning the *Web-application* in a browser if the CLI is to be avoided. Such data can serve an important role in the decision-making assessing the performance of a Model Predictive Controlled system. The lightweight property is present in the design of the simulation software emphasising memory management, efficient execution and simplicity.

For the time being, this software can input an arbitrarily FSRM, MPC tuning and MPC horizon addressing a vast set of scenarios and applications. The simulation builds upon algorithm (43), a convex QP with only inequality constraints present. This controller definition covers the core functionality of an MPC, with the weighting matrices $\boldsymbol{Q}$ and $\boldsymbol{R}$ and the controller horizons P, M and W present. Additionally, slack constraints and bias correction were also addressed in the controller formulation, even though the simulator does not account for the latter in simulations. To program the simulator, modern C++ is used along with OSQP, the fastest contemporary open-source optimizer available. Slack constraints can be enabled when defined and the controller can be disabled to address the open-loop behaviour. In order to assure scalability, package dependencies are handled in a sensible manner using *Conda*. The source code is modularized and documented using *Doxygen*, the de facto standard documenting tool, assuring maintainability and readability. The implementation adheres to the objectives of standard software principles, with a strong emphasis on producing high-quality code. Based on SEPTIC, a visualization tool is designed and implemented to analyse the controller performance readily. Plotting is achieved using Python's *matplotlib* and *React Plotly*. The simulator serves as an independent software interfaced by JSON files. With the designed architecture and *Emscripten Compiler Frontend* support, the software can be integrated into many different applications. *Light-Weight MPC* is only one example of an interfacing application.

The *Web-application* is a *React* based interface to the simulator. The application run Wasm directly in the browser in order to preserve computational efficiency interfacing the run-time-heavy simulator. *Node.js* is the only dependency needed with compliance to all standard web browsers such as Safari, Chrome, FireFox, etc. In addition to standard stationary PCs, the application can be utilized from any device having an Internet connection. The layout is designed to tailor the interface to the simulator automating the simulation procedure. Defensive programming is emphasised in both frameworks improving flow and user-experience. Hence, user-friendly feedback is provided whenever misuse or faults occur. To dispel any "black box" perception regarding the MPC, the frontend is designed with informative pages that provide customized descriptions of the underlying algorithm, model, and relevant tuning approaches. This feature aims to enhance transparency and understanding by providing users with comprehensive information about the inner workings of the controller.

To conclude, the software fulfils the project description under the assumption that the core functionality of an MPC is implemented and simulated using constant reference for

a discrete-time horizon T. The lightweight simulation software consists of a total of three repositories implemented using the same number of programming languages. A *Github organization* is established to distribute and brand the framework with a suitable logo. *Light-Weight MPC* is open-source released, granting the user the rights to use and change the software in line with the BSD 3-Clause, stated in Section E. The achieved controller performance elaborated in the test scenarios, Section 4, demonstrates the effectiveness of the lightweight controller implementation, showcasing its potential for waste reduction and resource savings. Moreover, the software can possibly enhance the understanding and knowledge of the controller method contributing to a more sustainable and efficient process industry. Overall, this project has successfully delivered a robust and efficient simulation tool, contributing to the field of predictive control algorithms and paving the way for further advancements in the industry.

6.1 Further work

The *MPC-simulator* has great opportunities to become a full-scale simulation software. By assessing the current version, there is a lot of functionality left to be ported to the *Web-application* as well. The next sections elaborate on the purposed features to bring the simulation software further. However, if all these functionalities were to be added to the project, *Light-Weight MPC* might not be considered a lightweight software anymore.

6.1.1 *MPC-simulator*

Firstly, a really useful feature to integrate into the simulator is bias correction. This could be achieved by parsing a set of differential equations, either explicitly or implicitly defined and simulating the representations using a *Runge-kutta* integrator. Since such models also are mathematically defined, there is no noise present in the output. As discussed in Section 5.1, the presents of noise is mandatory in real-life applications. Hence, to cover this in a simulation environment, white noise could be applied to the simulation output. Furthermore, the controller performance could be assessed under different levels of noise. One other approach to induce real-world features into the software is to interface the FMU model format [7]. This is a common format simulating industrial applications used by *Equinor* among others. The *SingleWell* model is an example of such a model description. Luckily, given the toolchain design, additional packages needed to implement this functionality can easily be added to the base.

Besides using the *Nullspace method*, there are also other formulation techniques to reduce the computational cost of the QP (43). For instance, the optimization vector, z_{cd}, can be utterly reduced using input blocking. The idea is to "block" ΔU, repeatedly for some sequences of the control horizon. The input can be "blocked" by demanding $\Delta U = 0$ for some sampling instants. The length of the blocking sequence grows exponentially, assuming that the controller manages to bring the plant to a steady state. Even though input blocking would have been a favourable aspect to add to the controller the approach was considered too complex to add to the lightweight framework.

Certainly, the simulator has the potential to be expanded to accommodate a wider range

of control models and address different cost functions. By incorporating additional model representations and optimization objectives, the simulator can cater to various control scenarios and provide more comprehensive insights into system behaviour. This flexibility allows for the exploration of different control strategies and the evaluation of their performance under different criteria. With further development, the simulator can become a versatile tool for control algorithm analysis and optimization in various domains. A specific model expansion is to include the State Space Model. This is a model which is often used in academia and in the field of control engineering. In contrast to the FSRM, this model does not assume exponentially stable system descriptions.

6.1.2 *Web-application*

As of now, the control model system descriptions need to be stored locally in the app. Usually, for large-scale applications, huge amounts of data need to be parsed in order to provide the requested service. Such data cannot be stored locally as the server would in time run out of memory. In order to operate on a large set of model descriptions, a database can be used. By utilization of a database, the selected model description can be requested using HTTP, such that the server only needs to address the needed data. An example of one such database that could be interfaced, handling JSON-formatted data, is *MongoDB* [15]. This software was also considered used during development.

As tested in Section 4, the *MPC-simulator* is equipped with far more functionality than accessible in the *Web-application*. For instance, the simulator can produce open-loop simulations and simulate further, using a different configuration. These aspects could be brought into the application. Addressing the first feature, the open-loop simulation functionality, this could require a new module within the menu bar. Possibly, the open-loop feature could be accessed by defining a key in the scenario file, such that every controller parameter is ignored by the underlying simulator. For the second functionality, a timer could be set after the first simulation, simulating a user-defined step size further for a small time delay. having this functionality present, the user would not need to address the MPC horizon, T, as it increases with time.

Bibliography

[1] URL: https://jupyter.org/about (visited on 13th Jan. 2023).

[2] *Anaconda Software Distribution*. Version Vers. 2-2.4.0. 2020. URL: https://docs.anaconda.com/ (visited on 22nd Apr. 2023).

[3] *Create React App*. URL: https://create-react-app.dev/ (visited on 29th May 2023).

[4] *Doxygen software documentation*. URL: https://www.doxygen.nl/ (visited on 8th June 2023).

[5] *Emscripten*. URL: https://emscripten.org/index.html (visited on 29th May 2023).

[6] Bjarne Foss and Tor Aksel N. Heirung. *Merging Optimization and Control*. Mar. 2016. ISBN: 978-82-7842-201-4.

[7] *Functional Mock-Up Interface*. URL: https://fmi-standard.org/ (visited on 5th June 2023).

[8] *GNU compiler toolchain*. URL: https://www.gnu.org (visited on 9th Jan. 2023).

[9] Morten Fredriksen & John-Morten Godhavn. *MPC in Equinor*. NTNU InnoCyPES Training School. Feb. 2023.

[10] Gaël Guennebaud, Benoît Jacob et al. *Eigen v3*. http://eigen.tuxfamily.org. 2010.

[11] J. D. Hunter. 'Matplotlib: A 2D graphics environment'. In: *Computing in Science & Engineering* 9.3 (2007), pp. 90–95. DOI: 10.1109/MCSE.2007.55.

[12] D.K.M. Kufoalor, Lars Imsland and Tor Johansen. 'High-performance Embedded Model Predictive Control using Step Response Models**This work is funded by the Research Council of Norway (NFR) and Statoil through the PETROMAKS project 215684, and also by NFR, Statoil, and DNV through the AMOS project 223254.' In: *IFAC-PapersOnLine* 48 (Dec. 2015), pp. 138–143. DOI: 10.1016/j.ifacol.2015.11.073.

[13] Niels Lohmann. *JSON for Modern C++*. Version 3.10.5. If you use this software, please cite it as below. Jan. 1970. DOI: 10.5281/zenodo.5814096. URL: https://doi.org/10.5281/zenodo.5814096.

[14] *Material UI*. URL: https://mui.com/material-ui/getting-started/overview/ (visited on 31st May 2023).

[15] *MongoDB homepage*. URL: https://www.mongodb.com/ (visited on 18th June 2023).

[16] Jorge Nocedal and Stephen J. Wright. *Numerical Optimization*. Springer New York, NY, 2006. ISBN: 978-0-387-40065-5.

[17] *Node.js*. URL: https://nodejs.org/en (visited on 29th May 2023).

[18] *OSQP-Eigen software*. URL: https://github.com/robotology/osqp-eigen (visited on 12th Jan. 2023).

[19] Felipe Pezoa et al. 'Foundations of JSON schema'. In: *Proceedings of the 25th International Conference on World Wide Web*. International World Wide Web Conferences Steering Committee. 2016, pp. 263–273.

[20] *Plotly*. URL: https://plotly.com/javascript/react/#introduction (visited on 30th May 2023).

[21] *qpOASES User's Manual*. URL: https://www.coin-or.org/qpOASES/doc/3.0/manual.pdf (visited on 16th June 2023).

[22] *React library*. URL: https://react.dev/ (visited on 29th May 2023).

[23] J.A. Rossiter. *A First Course in Predictive Control*. Control Series. CRC Press, 2018. ISBN: 9781351597159. URL: https://books.google.no/books?id=mkxnDwAAQBAJ.

[24] Henry Schreiner et al. *CLIUtils/CLI11: Version 2.3.2: Minor maintenance*. Version v2.3.2. Jan. 2023. DOI: 10.5281/zenodo.7502818. URL: https://doi.org/10.5281/zenodo.7502818.

[25] Dale E. Seborg et al. *Process Dynamics and Control, 3rd Edition*. Wiley, 2011, pp. 414–427. ISBN: 978-1-118-50671-4.

[26] Sigurd Skogestad. 'Simple analytic rules for model reduction and PID controller tuning'. In: *Journal of Process Control* 13.4 (2003), pp. 291–309. ISSN: 0959-1524. DOI: https://doi.org/10.1016/S0959-1524(02)00062-8. URL: https://www.sciencedirect.com/science/article/pii/S0959152402000628.

[27] B. Stellato et al. 'OSQP: an operator splitting solver for quadratic programs'. In: *Mathematical Programming Computation* 12.4 (2020), pp. 637–672. DOI: 10.1007/s12532-020-00179-2. URL: https://doi.org/10.1007/s12532-020-00179-2.

[28] Bjarne Stroustrup. *Programming: Principles and Practice Using C++ (2nd Edition)*. 2nd. Addison-Wesley Professional, 2014. ISBN: 0321992784.

[29] Geir O. Tvinnereim. 'Light-weight MPC. Unpublished, TTK4550 - specialization project'. In: (Jan. 2023).

[30] *Webassembly*. URL: https://webassembly.org/ (visited on 29th May 2023).

Appendix

A Block diagonal transform, blkdiag()

The block diagonal transform, blkdiag as defined in MATLAB, returns the corresponding block diagonal matrix given matrix arguments.

The variables $\boldsymbol{A_1}, \boldsymbol{A_2}, ..., \boldsymbol{A_n}$ represent n matrices. The blkdiag-function will then return a matrix $\boldsymbol{A}$, such that $\boldsymbol{A}$ has the form

$$\boldsymbol{A} = \text{blkdiag}(\boldsymbol{A_1}, \boldsymbol{A_2}, ..., \boldsymbol{A_n}) = \begin{bmatrix} \boldsymbol{A_1} & \boldsymbol{0} & \boldsymbol{0} & \boldsymbol{0} \\ \boldsymbol{0} & \boldsymbol{A_2} & \boldsymbol{0} & \boldsymbol{0} \\ \boldsymbol{0} & \boldsymbol{0} & \ddots & \boldsymbol{0} \\ \boldsymbol{0} & \boldsymbol{0} & \boldsymbol{0} & \boldsymbol{A_n} \end{bmatrix}. \tag{53}$$

An alternative formulation of the block diagonal transform can be derived in the case it is sensible to formulate $\boldsymbol{A}$ using only one matrix argument. For this purpose only two arguments are needed, the number of expansions and the matrix. Assume that the argument matrix $\boldsymbol{A_1}$ has the dimensions $\mathbb{R}^{m \times m}$. The output of the block diagonal transform given the expansion number n is represented in equation 54.

$$\boldsymbol{A} = \text{blkdiag}(\boldsymbol{A_1}, n) = \begin{bmatrix} \boldsymbol{A_1} & \boldsymbol{0} & \boldsymbol{0} & \boldsymbol{0} \\ \boldsymbol{0} & \boldsymbol{A_1} & \boldsymbol{0} & \boldsymbol{0} \\ \boldsymbol{0} & \boldsymbol{0} & \ddots & \boldsymbol{0} \\ \boldsymbol{0} & \boldsymbol{0} & \boldsymbol{0} & \boldsymbol{A_1} \end{bmatrix} \in \mathbb{R}^{m \cdot n \times m \cdot n}. \tag{54}$$

B Input-output JavaScript object notation formats

The following data formats describe the allowed input and produced output for the simulator. The common denominator is that all formats are defined using JSON [19].

The system file is one of two components for the input format. It is structured in the manner described in Section B.1. As described in Section 3.1.2 this file targets the model definition of a Finite Step-Response Model. The second part of the input format is the scenario file, Section B.2. On the other hand, this file targets the defined controller to be simulated on the chosen model description. Similarly as the system file target an FSRM, the scenario file targets a condensed MPC-FSRM (43). Finally, the simulation file is produced by serializing the simulation data along with the parameters needed to interpret it. The simulation format is described here, Section B.3. The corresponding C++ data types used to represent the IO information are `double`, `int` and `std::string`. The arrays are implemented using `std::vector<double>`.

B.1 System file

```
{   "model": {
      "n_CV": int, "n_MV": int, "N": int,  },
   "CV": [
      {  "output": string, "unit": string,
         "init": double,
         "S": [[S11, S12, S13, ... , S1N],
               [S21, S22, S23, ... , S2N],
               ... ,
               [S n_MV, ... , S n_MV N]]    },
         ... ,
      {  "output": string, "unit": string,
         "init": double,
         "S": [[S11, S12, S13, ... , S1N],
               [S21, S22, S23, ... , S2N],
               ... ,
               [S n_MV, ... , S n_MV N]]    }
   ],
   "MV": [
      {  "input": string, "unit": string,
         "init": double, },
         ... ,
      {  "input": string, "unit": string,
         "init": double, }
   ],                          }
```

Figure 31: Template JSON-formatted system file.

B.2 Scenario file

```
{
 "system": "system_name",

 "MPC": {
   "P": int,
   "M": int,
   "W": int,
   "Q": [Q1, Q2, ... , QP],
   "R": [R1, R2, ... , RM],
   "RoH": [Ro1, Ro2, ..., Ro n_CV],
   "RoL": [Ro1, Ro2, ..., Ro n_CV],
 },

 "c": [
   {"du[1]": [double_low, double_high]},
   ...,
   {"du[n_MV]": [double_low, double_high]},
   {"u[1]": [double_low, double_high]},
   ...,
   {"u[n_MV]": [double_low, double_high]},
   {"y[1]": [double_low, double_high]},
   ...,
   {"y[n_CV]": [double_low, double_high]},
  ]
}
```

Figure 32: Template JSON-formatted scenario file.

B.3 Simulation file

While the other two file formats are inputted into the simulation software, the simulation file is only outputted. This file format is used to analyse the MPC-scenario and needs to contain information for performance assessment. It is trivial that the variables describing predictions, actuation, model parameters etc. should be present. These variables are used to produce the plots seen in Section 4. However, the $\Delta\tilde{U}$, identified as the key "du_tilde" in the simulation file, strikes out. This attribute is only read when the simulator is called without the *-n* flag, yielding an extension of an old simulation for future time steps. The information is used to set the simulator in the same controller configuration as the targeted simulation.

```
{ "scenario": "scenario_name",
"T": int,
"n_CV": int, "n_MV": int,
"P": int,  "M": int,
"du_tilde" : [[du11, du12, du13, ... , du1(N-1)],
        [du21, du22, du23, ... , du2(N-1)],
        ... ,
        [du n_MV, ... , du n_MV (N-1)]],
"CV": [
     {  "output": "output_name",
        "unit": string,
        "c": [low, high] (double),
        "y_pred": [y1, y2, y3, ... , y(T+P+1)],
        "ref": [r1, r2, ..., r(T+P+1)] },
        ... ,
     {  "output": "output_name",
        "unit": string,
        "c": [low, high] (double),
        "y_pred": [y1, y2, y3, ... , y(T+P+1)],
        "ref": [r1, r2, ..., r(T+P+1)] } ],
"MV": [
     {  "input": "input_name",
        "unit": string,
        "c": [low, high],
        "u": [u1, u2, u3, ... , u(T+M)] },
        ... ,
     {  "input": "input_name",
        "unit": string,
        "c": [low, high],
        "u": [u1, u2, u3, ... , u(T+M)] } ]
}
```

Figure 33: Template JSON-formatted simulation file.

C Condensed formulation without slack constraints

Given the standard optimization vector without the slack constraints: $z_{st} = \begin{bmatrix} \Delta U \\ U \\ \tilde{Y} \end{bmatrix}$.

This vector corresponds to a reduced standard QP given in equation (36):

$$\begin{aligned} &\min_{z_{st}} \frac{1}{2} z_{st}^T \mathbf{G} z_{zt} + q(k)^T z_{st} \\ &= \frac{1}{2} \begin{bmatrix} \Delta U \\ U \\ \tilde{Y} \end{bmatrix}^T \begin{bmatrix} 2\bar{\boldsymbol{R}} & \mathbf{0} & \mathbf{0} \\ \mathbf{0} & \mathbf{0} & \mathbf{0} \\ \mathbf{0} & \mathbf{0} & 2\bar{\boldsymbol{Q}} \end{bmatrix} \begin{bmatrix} \Delta U \\ U \\ \tilde{Y} \end{bmatrix} + \begin{bmatrix} \mathbf{0} & \mathbf{0} & -2\mathcal{T}(k)^T \bar{\boldsymbol{Q}} \end{bmatrix} \begin{bmatrix} \Delta U \\ U \\ \tilde{Y} \end{bmatrix} \end{aligned} \tag{55}$$

such that

$$\boldsymbol{E} z_{st} = \begin{bmatrix} -\boldsymbol{I} & \boldsymbol{K} & \mathbf{0} \\ -\boldsymbol{\Theta} & \mathbf{0} & \boldsymbol{I} \end{bmatrix} \begin{bmatrix} \Delta U \\ U \\ \tilde{Y} \end{bmatrix} = \begin{bmatrix} \boldsymbol{\Gamma} \tilde{U}(k-1) \\ \boldsymbol{\Phi} \Delta \tilde{U}(k) + \boldsymbol{\Psi} \tilde{U}(k-N) + B(k) \end{bmatrix} = f \tag{56}$$

Furthermore, one can apply the same approach used in section 2.3 to reduce the optimization vector to $z_{cd} = \Delta U$. The reduced system then yields:

$$\begin{aligned} z_{st} = \begin{bmatrix} \Delta U \\ U \\ \tilde{Y} \end{bmatrix} &= \begin{bmatrix} \Delta U \\ \mathbf{K}^{-1} \boldsymbol{\Gamma} \tilde{U}(k-1) + \mathbf{K}^{-1} \Delta U(k) \\ \Lambda(k) + \Theta(k) \Delta U(k) \end{bmatrix} \\ &= \begin{bmatrix} \mathbf{I} \\ \mathbf{K^{-1}} \\ \boldsymbol{\Theta} \end{bmatrix} \Delta U + \begin{bmatrix} 0 \\ \mathbf{K}^{-1} \boldsymbol{\Gamma} \tilde{U}(k-1) \\ \Lambda(k) \end{bmatrix} \\ &= \mathbf{A} \Delta U + C(k) \end{aligned} \tag{57}$$

The condensed MPC without slack variables can therefore be summarised as:

$$\min_{z_{cd}} \frac{1}{2} z_{cd}^T \boldsymbol{G_{cd}} z_{cd} + q_{cd}^T(k) z_{cd}, \tag{58a}$$

$$\boldsymbol{G_{cd}} = \mathbf{A}^T \mathbf{G} \mathbf{A} = 2 \cdot (\boldsymbol{\Theta}^T \bar{\boldsymbol{Q}} \boldsymbol{\Theta} + \bar{\boldsymbol{R}}) \succeq 0, \tag{58b}$$

$$q_{cd}(k) = C(k)^T \mathbf{G} \mathbf{A} + q^T \mathbf{A} = 2 \cdot \boldsymbol{\Theta}^T \bar{\boldsymbol{Q}} (\Lambda(k) - \mathcal{T}(k)), \tag{58c}$$

such that

$$\begin{bmatrix} \Delta \underline{U} \\ \underline{U} \\ \underline{Y} \end{bmatrix} - \begin{bmatrix} 0 \\ \mathbf{K}^{-1} \boldsymbol{\Gamma} \tilde{U}(k-1) \\ \Lambda(k) \end{bmatrix} \leq \begin{bmatrix} \mathbf{I} \\ \mathbf{K^{-1}} \\ \boldsymbol{\Theta} \end{bmatrix} z_{cd} \leq \begin{bmatrix} \Delta \bar{U} \\ \bar{U} \\ \bar{Y} \end{bmatrix} - \begin{bmatrix} 0 \\ \mathbf{K}^{-1} \boldsymbol{\Gamma} \tilde{U}(k-1) \\ \Lambda(k) \end{bmatrix}. \tag{58d}$$

D Condensed MPC-FSRM C++ implementation

Implementation of simulation routine described in Algorithm 2. From line 80, the MPC loop is executed and the following comments align with the algorithm. Prior lines are used to configure the OSQP-solver, allocate run-time variables and check feasibility. Section D shows the corresponding `SRSolver` header file.

solvers.cc

```
1  /**
2   * @file solvers.cc
3   * @author Geir Ola Tvinnereim
4   * @brief MPC-FSRM controller module
5   * @version 0.1
6   * @date 2023-04-19
7   *
8   * @copyright Released under the terms of the BSD 3-Clause License
9   *
10  */
11 #include "MPC/solvers.h"
12 #include "MPC/condensed_qp.h"
13 #include <OsqpEigen/OsqpEigen.h>
14
15 #include <stdexcept>
16 using SparseXd = Eigen::SparseMatrix<double>;
17
18 void SRSolver(int T, MatrixXd& u_mat, MatrixXd& y_pred, FSRModel& fsr_sim,
19                 FSRModel& fsr_cost, const MPCConfig& conf, const VectorXd& z_min,
20                 const VectorXd& z_max, const MatrixXd& ref) {
21     // Initialize solver:
22     OsqpEigen::Solver solver;
23     // Starts primal and dual variables from previous QP
24     solver.settings()->setWarmStart(true);
25     solver.settings()->setVerbosity(false); // Disable printing
26
27     // MPC Scenario variables:
28     const int P = fsr_cost.getP(), M = fsr_cost.getM(), W = fsr_cost.getW();
29     const int n_MV = fsr_cost.getN_MV(), n_CV = fsr_cost.getN_CV();
30     // Define QP sizes:
31     // n = #Optimization variables,
32     const int n = M * n_MV + 2 * n_CV; dim(z_cd)
33     // m = #Constraints, dim(z_st)
34     const int m = 2 * (M * n_MV + (P-W) * n_CV + n_CV);
35     const int a = M * n_MV; // dim(du)
36     const VectorXd z_max_pop = PopulateConstraints(z_max, conf, a, n_MV, n_CV);
37     const VectorXd z_min_pop = PopulateConstraints(z_min, conf, a, n_MV, n_CV);
38     //z_*_pop are lower/upper populated constraints
39
40     solver.data()->setNumberOfVariables(n);
41     solver.data()->setNumberOfConstraints(m);
```

```
42
43      // Define Cost function variables:
44      SparseXd Q_bar, R_bar;
45      const SparseXd Gamma = setGamma(M, n_MV), one = setOneMatrix(P, W, n_CV);
46      const MatrixXd K_inv = setKInv(a), theta = fsr_cost.getTheta();
47      // Dynamic variables:
48      VectorXd q, l = VectorXd::Zero(m), u = VectorXd::Zero(m);
49      // l and u are lower and upper constraints, z_cd
50      VectorXd c_l = ConfigureConstraint(z_min_pop, m, a, false);
51      VectorXd c_u = ConfigureConstraint(z_max_pop, m, a, true);
52
53      // NB! W-dependant
54      setWeightMatrices(Q_bar, R_bar, conf);
55      const SparseXd G = setHessianMatrix(Q_bar, R_bar, one, theta, a, n, n_CV);
56      const SparseXd A = setConstraintMatrix(one, theta, K_inv, m, n, a, n_CV);
57      // Initial gradient
58      setGradientVector(q, fsr_cost, Q_bar, one, ref, conf, n, 0);
59      setConstraintVectors(l, u, fsr_cost, c_l, c_u, K_inv, Gamma, m, a);
60
61      // Set solver data:
62      if (!solver.data()->setHessianMatrix(G)) {
63          throw std::runtime_error("Cannot initialize Hessian"); }
64      if (!solver.data()->setGradient(q)) {
65          throw std::runtime_error("Cannot initialize Gradient"); }
66      if (!solver.data()->setLinearConstraintsMatrix(A)) {
67          throw std::runtime_error("Cannot initialize constraint matrix"); }
68      if (!solver.data()->setLowerBound(l)) {
69          throw std::runtime_error("Cannot initialize lower bound");}
70      if (!solver.data()->setUpperBound(u)) {
71          throw std::runtime_error("Cannot initialize upper bound"); }
72      if (!solver.initSolver()) {
73          throw std::runtime_error("Cannot initialize solver"); }
74
75      u_mat = MatrixXd::Zero(n_MV, T + M);
76      y_pred = MatrixXd::Zero(n_CV, T + P);
77      const SparseXd omega_u = setOmegaU(M, n_MV);
78
79      // MPC loop:
80      for (int k = 0; k <= T; k++) {
81          // Optimize:
82          if (solver.solveProblem() != OsqpEigen::ErrorExitFlag::NoError)
83          { throw std::runtime_error("Cannot solve problem"); }
84
85          // Claim solution:
86          VectorXd z_st = solver.getSolution(); // [dU, eta_h, eta_l]
87          VectorXd z = z_st(Eigen::seq(0, a - 1)); // [dU]
88
89          // Store optimal du and y_pref: Before update!
90          if (k == T) {
91              u_mat.block(0, T, n_MV, M) = (K_inv *
92              z).reshaped<Eigen::RowMajor>(n_MV, M).colwise()
93              + u_mat.col(T-1);
```

```
            y_pred.block(0, k, n_CV, P) = fsr_sim.getY(z, true);
        } else {
            // Store y_pred
            y_pred.col(k) = fsr_sim.getY(z);

            // Propagate FSR models: Update both!
            VectorXd du = omega_u * z; // MPC actuation
            fsr_sim.UpdateU(du);
            fsr_cost.UpdateU(du);
            u_mat.col(k) = fsr_sim.getUK();

            // Update MPC problem:
            setConstraintVectors(l, u, fsr_cost, c_l, c_u, K_inv, Gamma, m, a);
            setGradientVector(q, fsr_cost, Q_bar, one, ref, conf, n, k);

            // Check if bounds are valid:
            if (!solver.updateBounds(l, u)) {
                throw std::runtime_error("Cannot update bounds"); }
            if (!solver.updateGradient(q)) {
                throw std::runtime_error("Cannot update gradient"); }
        }
    }
}
```

solvers.h

```
/**
 * @file solvers.h
 * @author Geir Ola Tvinnereim
 * @brief MPC-FSRM controller module
 * @version 0.1
 * @date 2023-04-19
 *
 * @copyright Released under the terms of the BSD 3-Clause License
 *
 */
#ifndef SOLVERS_H
#define SOLVERS_H

#include "model/FSRModel.h"
#include "IO/data_objects.h"

#include <Eigen/Eigen>
using VectorXd = Eigen::VectorXd;
using MatrixXd = Eigen::MatrixXd;
/**
 * @brief Solving the condensed positive semi-definite optimization problem
 * using OSQP-Eigen for W != 0
 *
 * @param T MPC horizon
```

```
 * @param u_mat Optimized u, filled by reference
 * @param y_pred Predicted y, filled by reference
 * @param fsr_sim Simulation model
 * @param fsr_cost MPC model
 * @param conf MPC configuration
 * @param z_min lower constraint vector
 * @param z_max upper constraint vector
 * @param ref Output reference data
 */
void SRSolver(int T, MatrixXd& u_mat, MatrixXd& y_pred, FSRModel& fsr_sim,
              FSRModel& fsr_cost, const MPCConfig& conf, const VectorXd& z_min,
              const VectorXd& z_max, const MatrixXd& ref);

#endif // SOLVERS_H
```

E BSD 3-Clause License

```
BSD 3-Clause License

Copyright (c) 2022, Fondazione Istituto Italiano di Tecnologia
All rights reserved.

Redistribution and use in source and binary forms, with or without
modification, are permitted provided that the following
conditions are met:

1. Redistributions of source code must retain the above
    copyright notice, this list of conditions and
    the following disclaimer.

2. Redistributions in binary form must reproduce the above
    copyright notice, this list of conditions and the
    following disclaimer in the documentation and/or other
    materials provided with the distribution.

3. Neither the name of the copyright holder nor the names of its
   contributors may be used to endorse or promote products
   derived from this software without specific prior
   written permission.

THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND
CONTRIBUTORS "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES,
INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF
MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED.
IN NO EVENT SHALL THE COPYRIGHT HOLDER OR CONTRIBUTORS BE LIABLE
FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY,
OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO,
PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA,
OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY
THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY,
OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY
OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF
THE POSSIBILITY OF SUCH DAMAGE.
```

F *Web-application* layout

As stated in the implementation Section 3.2, the frontend consists of multiple modules accessible through the menu bar. The default scenario and simulation page are already covered in Section 3.2.5, however, the remaining pages are yet to be described.

After a valid simulation scenario is defined inside the scenario page, a simulation may be produced. Due to the complexity of the controller, the test simulations are executed in about 5 seconds. However, this value varies depending on the model description and horizon. While the application runs the Wasm compiled simulator, the end user is not notified that a simulation is ongoing by displaying the Loading page. Figure 34 demonstrates this site. A circular progress bar along with a string is present in the centre of the page. After the simulator has returned the simulation data, the application navigates to the simulation page plotting the result.

The three last pages are not interfacing the *MPC-simulator* directly, but present to inform the user of the theory behind the software. Starting off with the Algorithm page shown in Figure 36. This page describes the condensed controller formulation (43) solved for each simulation step. In order to achieve the best theoretical controller performance, the page also provides tuning approaches. Similarly, the Model page provides information about the control model used when simulating. This page is demonstrated in Figure 35. Lastly, the About page describes the purpose behind the simulation software, the contribution and the criteria needed to be fulfilled in order to produce a simulation using the web interface. This page is encapsulated in Figure 37. Within this page, a link to the source code is provided in the event a user wants to state a software issue etc. Since bias correction is not addressed in the simulator, this aspect is expressed in the simulation loop illustration. Furthermore, in order to avoid confusion, the controller parameters criteria are stated emphasising UX.

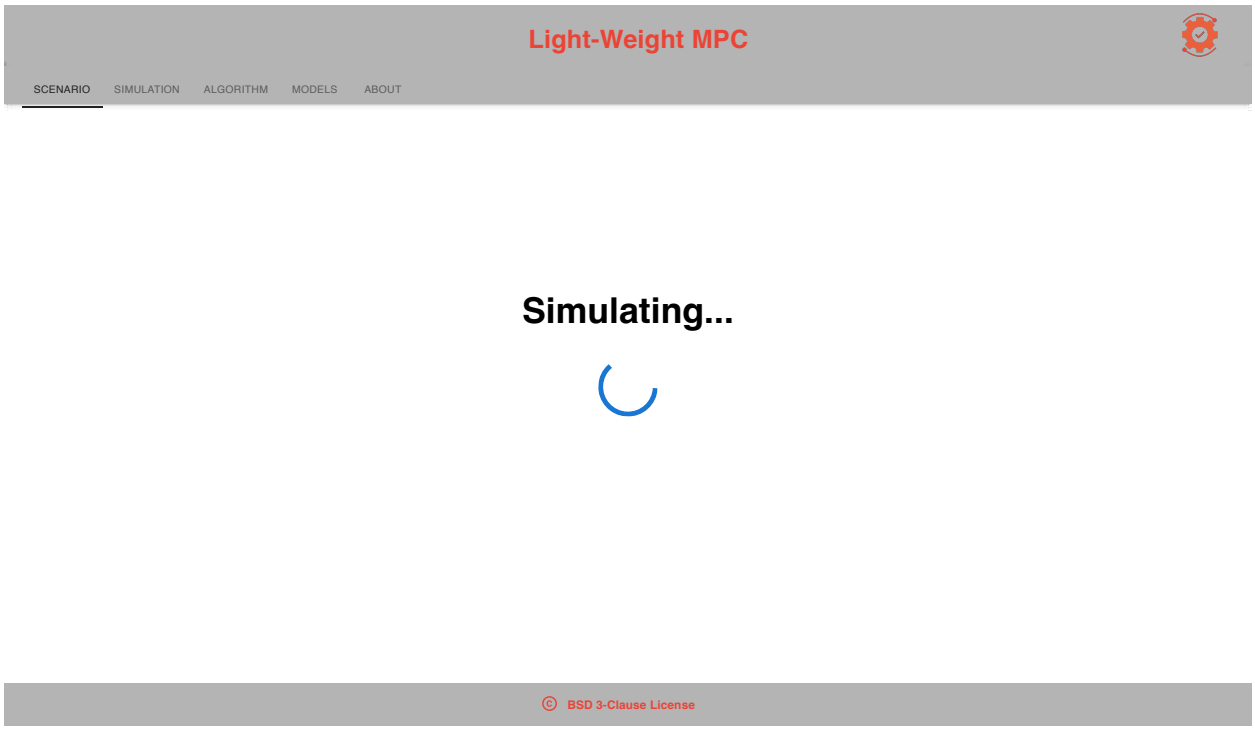


Figure 34: Loading page: while the application is waiting for the simulator to return simulation results, a spinning progress wheel is displayed.

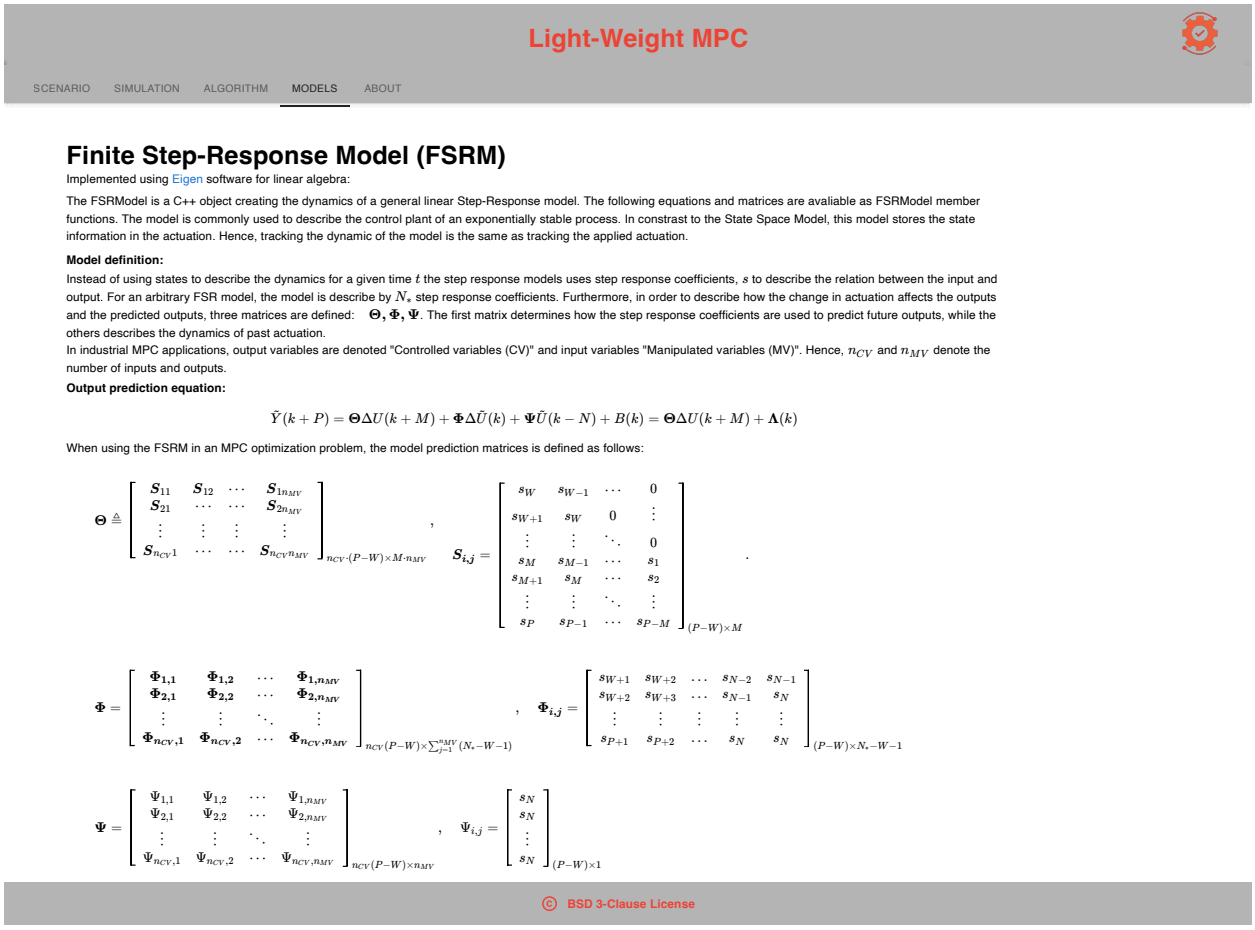


Figure 35: Model page: Informative page displaying the FSRM description.

MPC-FSRM algorithm:

Implemented using osqp-eigen C++ wrapper for the OSQP software: See chapter 2.3: Model Predictive Control for Finite Step-Response Model for more information.

The OSQP, operator splitting QP solver solves the problems of the following form:

$$min \frac{1}{2} z^T \boldsymbol{G} z + q^T z \quad s.t. \quad l \leq \boldsymbol{A} z \leq u$$

Given, $\boldsymbol{G}$ is a positive definite matrix, yielding a convex quadratic program. The OSQP solver uses an custom ADMM-based first order method and is one of the fastest QP solvers avaliable.

Step-Response MPC solver:

This MPC controller is defined as a standard quadratic program (QP) where the cost aims to minimize the error between the output Y and the reference r_y:

$$\min \sum_{j=W+1}^{P} |(Y(k+j \mid k) - \mathcal{T}(k+j))|_{\bar{\boldsymbol{Q}}}^2 + \sum_{j=0}^{(M-1)} |\Delta U(k+j)|_{\bar{\boldsymbol{R}}}^2 + \bar{\rho\epsilon} + \underline{\rho\epsilon}$$

The cost function is constrained by the model definition and relating variables. For the general FSRM-MPC algorithm this cost is constrained by:

$$s.t. \quad Y(k+P) = \boldsymbol{\Theta} \Delta U(k) + \boldsymbol{\Lambda}(k),$$

$$\Delta \underline{U} \leq \Delta U(k+j) \leq \Delta \overline{U}, \quad j \in \{0, \dots, M-1\}$$

$$\underline{U} \leq U(k+j) \leq \bar{U}, \quad j \in \{0, \dots, M-1\}$$

$$\underline{Y} - \epsilon_l \leq \boldsymbol{Y}(k+j) \leq \bar{Y} + \epsilon_h, \quad \epsilon_h \geq 0, \epsilon_l \geq 0, \quad j \in \{W+1, \dots, P\}$$

In order to implement the slack constraints $\epsilon \in \mathbb{R}^{n_{CV}}$, one must map the slack variables to the output $Y \in \mathbb{R}^{P \cdot n_{CV}}$. This is done by defining the scaling matrix $\mathbf{1}$:

$$\mathbf{1} = \begin{bmatrix} 1_{0,1} & 0 & \dots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 1_{(P-W),1} & 0 & \ddots & 0 \\ 0 & 1_{0,2} & \ddots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 1_{(P-W),2} & \ddots & 0 \\ \vdots & \ddots & \ddots & \vdots \\ 0 & 0 & 0 & 1_{0,n_{CV}} \\ \vdots & \vdots & \vdots & \vdots \\ 0 & 0 & 0 & 1_{(P-W),n_{CV}} \end{bmatrix} \in \mathbb{R}^{(P-W) \cdot n_{CV} \times n_{CV}}$$

Tuning approaches:

For this model based controller description, a total of 7 variables are determing the performance. The horizons, $P > 0, M > 0, W \geq 0$, are all present in the cost function. The prediction horizon P determines the closed loop stability. Hence, in order to assure stability, this horizon should be set large enough to cover the domanint dynamics. However, predicting far ahead is comuputationally costly, therefore one can additionally tune the control and time delay horizon, M, W, to decrease the problem size. The prediction horizons starts at W and specifies the number of initial steps in which the deviations of the output from the reference are not penalized. Setting $W > 0$ is particularly useful for systems where there is a time lag between the time of the control action is applied and the time an effect is seen. The control horizon M determines how many prediced actuations are penalized in the cost function.

While the horizons determine how many predictions to be accounted for, the tuning parameters $\bar{\boldsymbol{Q}} \succ 0$ and $\bar{\boldsymbol{R}} \succ 0$ describes the penalization of respectively the output deviation and input value. If the MPC simulation yields large output deviations, an increase in the $\bar{\boldsymbol{Q}}$ matrix shall reduce the error. Having large $\bar{\boldsymbol{Q}}$-gains is typical for tracking control. Consequently, if you wish to avoid large actuation this can be penalized by the $\bar{\boldsymbol{R}}$-matrix tuning. Such functionality comes often handy when controlling process plants where the use of too much actuation is costly. The last tuning parameters, ρ_h, ρ_l corresponds to the output slack constraints. Having slack variables present in the controller formulation, the controller allows output variables to exeed their constraints if neccessary. However, one can prevent this border crossing by tuning these parameters emersively.

Condensed Form:

The condensed formulation solves a smaller optimalization problem, obtained using the Nullspace method. This method reduces the number of optimalization variabled by defining a linear transform. $z_{st} = Az_{cd} + C$ The transform cancels optimalization variables with the given constraints, yielding a computationally easier problem.

The original optimalization vector, $z_{st} = \begin{bmatrix} \Delta U \\ U \\ Y \\ \epsilon_h \\ \epsilon_l \end{bmatrix}$, is reduced to $z_{cd} = \begin{bmatrix} \Delta U \\ \epsilon_h \\ \epsilon_l \end{bmatrix}$.

The consdenced formulation is formulated followingly:

$$\min_{z_{cd}} \quad \frac{1}{2} z_{cd}^T \boldsymbol{G_{cd}} z_{cd} + q_{cd}^T z_{cd}$$

$$\boldsymbol{G_{cd}} = \mathbf{A}^T \mathbf{G} \mathbf{A} = 2 \cdot \begin{bmatrix} \boldsymbol{\Theta}^T \bar{\boldsymbol{Q}} \boldsymbol{\Theta} + \bar{\boldsymbol{R}} & -\boldsymbol{\Theta}^T \overline{\boldsymbol{Q}} \mathbf{1} & \boldsymbol{\Theta}^T \overline{\boldsymbol{Q}} \mathbf{1} \\ -\mathbf{1}^T \overline{\boldsymbol{Q}} \boldsymbol{\Theta} & \mathbf{1}^T \overline{\boldsymbol{Q}} \mathbf{1} & \mathbf{0} \\ \mathbf{1}^T \overline{\boldsymbol{Q}} \boldsymbol{\Theta} & \mathbf{0} & \mathbf{1}^T \overline{\boldsymbol{Q}} \mathbf{1} \end{bmatrix},$$

$$q_{cd}(k) = C(k)^T \mathbf{G} \mathbf{A} + q^T \mathbf{A} = 2 \cdot \begin{bmatrix} 2 \cdot \boldsymbol{\Theta}^T \bar{\boldsymbol{Q}} (\Lambda(k) - \mathcal{T}(k)) \\ -\mathbf{1}^T \overline{\boldsymbol{Q}} (\Lambda(k) - \mathcal{T}(k)) + \rho_h \\ \mathbf{1}^T \overline{\boldsymbol{Q}} (\Lambda(k) - \mathcal{T}(k)) + \rho_l \end{bmatrix},$$

such that:

$$\underline{z}_{cd}(k) = \begin{bmatrix} \Delta \underline{U} \\ \underline{U} \\ -\infty \\ \underline{Y} \\ 0 \\ 0 \end{bmatrix} - \begin{bmatrix} 0 \\ \mathbf{K}^{-1} \boldsymbol{\Gamma} U(k-1) \\ \boldsymbol{\Lambda}(k) \\ \boldsymbol{\Lambda}(k) \\ 0 \\ 0 \end{bmatrix} \leq \begin{bmatrix} \mathbf{I} & \mathbf{0} & \mathbf{0} \\ \mathbf{K}^{-1} & \mathbf{0} & \mathbf{0} \\ \boldsymbol{\Theta} & -\mathbf{1} & \mathbf{0} \\ \boldsymbol{\Theta} & \mathbf{0} & \mathbf{1} \\ \mathbf{0} & \mathbf{I} & \mathbf{0} \\ \mathbf{0} & \mathbf{0} & \mathbf{I} \end{bmatrix} z_{cd} \leq \begin{bmatrix} \Delta \bar{U} \\ \bar{U} \\ \bar{Y} \\ \infty \\ \infty \\ \infty \end{bmatrix} - \begin{bmatrix} 0 \\ \mathbf{K}^{-1} \boldsymbol{\Gamma} U(k-1) \\ \boldsymbol{\Lambda}(k) \\ \boldsymbol{\Lambda}(k) \\ 0 \\ 0 \end{bmatrix} = \bar{z}_{cd}(k)$$

Condensed Formulation without slack variables:

If the tuning parameters RoH and RoL are both null vectors, slack variables are disabled form the Model Predictive Controller. This yields another controller formulation.

The original optimalization vector, $z_{st} = \begin{bmatrix} \Delta U \\ U \\ Y \end{bmatrix}$, is reduced to $z_{cd} = \Delta U$.

The consdenced formulation is formulated followingly:

$$\min_{z_{cd}} \quad \frac{1}{2} z_{cd}^T \boldsymbol{G_{cd}} z_{cd} + q_{cd}^T z_{cd}$$

$$\boldsymbol{G_{cd}} = \mathbf{A}^T \mathbf{G} \mathbf{A} = 2 \cdot (\boldsymbol{\Theta}^T \bar{\boldsymbol{Q}} \boldsymbol{\Theta} + \bar{\boldsymbol{R}}) \succeq 0,$$

$$q_{cd}(k) = C(k)^T \mathbf{G} \mathbf{A} + q^T \mathbf{A} = 2 \cdot \boldsymbol{\Theta}^T \bar{\boldsymbol{Q}} (\Lambda(k) - \mathcal{T}(k))$$

such that:

$$\underline{z}_{cd}(k) = \begin{bmatrix} \Delta \underline{U} \\ \underline{U} \\ \underline{Y} \end{bmatrix} - \begin{bmatrix} 0 \\ \mathbf{K}^{-1} \boldsymbol{\Gamma} U(k-1) \\ \Lambda(k) \end{bmatrix} \leq \begin{bmatrix} \mathbf{I} \\ \mathbf{K}^{-1} \\ \boldsymbol{\Theta} \end{bmatrix} z_{cd} \leq \begin{bmatrix} \Delta \bar{U} \\ \bar{U} \\ \bar{Y} \end{bmatrix} - \begin{bmatrix} 0 \\ \mathbf{K}^{-1} \boldsymbol{\Gamma} U(k-1) \\ \Lambda(k) \end{bmatrix} = \bar{z}_{cd}(k)$$

Figure 36: Algorithm page: Informative page explaining the controller method.

Light-Weight MPC

SCENARIO SIMULATION ALGORITHM MODELS ABOUT

This is a repo for implementing the master's thesis for the study programme Cybernetics & Robotics at NTNU. The thesis is handed out by Equinor, and aims on implementing a simpler software framework simulating optimized control on step-response models. The simulator available through this web application simulates the performance of an MPC on the corresponding control model assuming no model errors. The functionlity to define a plant model is left out for future implementations. Hence, there are no output feedback present in the application. The software reads a model definition defined in the system definition JSON format. Followingly, it parses the UX defined controller tuning and simulates the specified controller method.

Masters thesis description

Master student: Geir Ola Tvinnereim

Supervisors: Prof. Lars Struen Imsland (ITK) & Dr. Gisle Otto Eikrem (Equinor)

The source code can be found at Github and is distributed under the BSD-3-Clause license.

fig: Software logo

fig: Illustration of the Light-Weight MPC simulation loop.

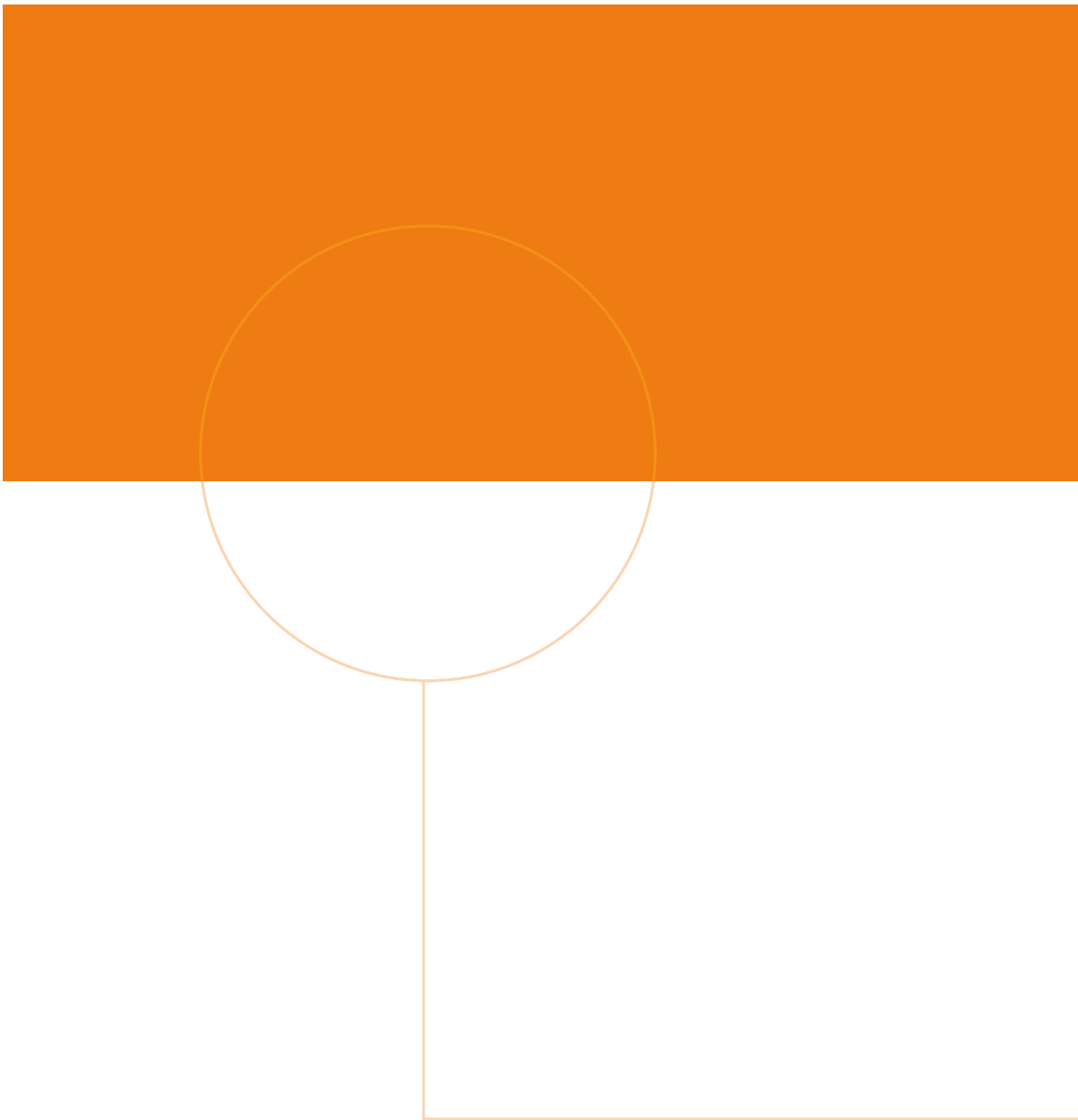
Software usage:

- Navigate to "Scenario" page and define you MPC controller.
- A valid controller definition yields a green "Run Simulation" button.
- After clicking the button, the simulation results shall appear in the "Simulation" page, othervise an error message occurs.
- The mathematical description of the controller and the model structure is respectively defined in the pages "Algorithm" and "Models".

Parameter criteria for valid simulation:

Parameter	Criteria
P	$P \in [M, N_*]$
M	$M \in [W, P]$
W	$W \in [0, M]$
T	$T > 0$
$\boldsymbol{Q}$	$\boldsymbol{Q} \geq 0, \quad \boldsymbol{Q}.length = n_{CV}$
$\boldsymbol{R}$	$\boldsymbol{R} \geq 0, \quad \boldsymbol{R}.length = n_{MV}$
ρ_h	$\rho_h \geq 0, \quad \rho_h.length = n_{CV}$
ρ_l	$\rho_l \geq 0, \quad \rho_l.length = n_{CV}$
$\overline{z}$	$\overline{z}_i > \underline{z}_i, \quad i \in \{0, \ldots, 2 \cdot n_{MV} + n_{CV}\} \quad \overline{z}.length = 2 \cdot n_{MV} + n_{CV}$
$\underline{z}$	$\underline{z}_i < \overline{z}_i, \quad i \in \{0, \ldots, 2 \cdot n_{MV} + n_{CV}\} \quad \underline{z}.length = 2 \cdot n_{MV} + n_{CV}$

Figure 37: About page: Informative page describing the background of the project along with usage and controller parameters criteria.



NTNU
Norwegian University of
Science and Technology

equinor