NTNU – Trondheim
Norwegian University of
Science and Technology

Efficient solving of Quadratic Programming with ramp functions for QP-MPC in Python

Ingvild Rustad Haugen

December 2022

SPECIALIZATION PROJECT
Department of Engineering Cybernetics
Norwegian University of Science and Technology

Supervisor 1: Morten Hovd

Supervisor 2: Sverre Hendseth

Preface

This thesis is written as a part of the requirements for the 5th year at Cybernetics and Robotics at Norwegian University of Science and Technology, Trondheim, Norway. The project was developed trough the autumn semester 2022 and delivered in December 2022.

The thesis builds upon the research paper published by professor Morten Hovd (NTNU) and professor Giorgio Valmorbida (Université Paris-Sacalay) *Quadratic programming with ramp functions and fast QP-MPC solutions* [1], which was submitted to the research journal Automatica.

Executive summary

This report presents the implementation of a numerical algorithm for solving quadratic programming (QP) problems from model predictive control (MPC) using the Python programming language and the NumPy package. The report begins with a comparison of the Matlab and Python code for the algorithm, followed by a discussion of tools for code analysis. In addition, the transferability of the discoveries from the code optimization is discussed, and some recommendations of further work is given. Finally, the report presents proposals for improvements to the implementation, along with comparisons to previous versions and the original Matlab code.

Abbreviations

Abbreviation	Description
MPC	Model Predictive Control
QP	Quadratic programming
NumPy	Numerical Python (Python Package)

Contents

Abstract i
Executive summary ii
Abbreviations iii

1 Introduction **1**
1.1 Background and motivation 1
1.2 Main objectives of the project 1
1.3 Delimitations 2
1.4 Structure of the report 2

2 Theory **3**
2.1 Mathematical optimization theory 3
2.1.1 The optimization problem and feasible set 3
2.1.2 Quadratic programming 4
2.1.3 Active set 4
2.1.4 First order necessary conditions 5
2.2 Model Predictive Control 5
2.2.1 Objective functions for discrete time systems 5
2.2.2 MPC-QP optimization problem 6
2.2.3 The MPC concept 6
2.3 The algorithm 7
2.3.1 Ramp functions to define the active set 8
2.3.2 Solution to (2.19) by ramp functions 9
2.4 Python as a programming language 12
2.4.1 NumPy 13
2.5 Analyzing and evaluating Python code 14
2.5.1 *Line Profiler* 14
2.6 Matlab as a programming language 15

3 Methods and tools **16**
3.1 Models used 16
3.2 NumPy 17
3.3 Code translation 17

3.3.1 Code structure 18
3.3.2 The use of debugger 18
3.3.3 Verification of the Python version 18
3.4 Measurements of runtime 19
3.5 Runtime analysis with the use of *Line Profiler* 19

4 Results and improvements **21**
4.1 Validation of the versions 21
4.1.1 Analysis with the use of *LineProfiler* 29

5 Discussion and further work **33**

6 Conclusion and further work **36**

References **37**

Chapter 1

Introduction

1.1 Background and motivation

Model Predictive Control (MPC) is one of the most popular type of controllers for robust control in the process industry [2]. MPC introduces the advantage of including constraints on input and output variables and the possibility of having non-linearity in both the model and the constraints. Despite its advantages, MPC can be computationally demanding as it requires solving an optimization provlem at each sampling interval. This has historically limited its use to slower processes, such as chemical processes. [3].

Since the beginning of MPC controllers, it has been desired to increase the computational performance through more efficient algorithms so that the application field can be expanded. This project aims to contribute to this development with a new method for solving the MPC-QP problem.

1.2 Main objectives of the project

The present project is a preparatory study for the Master thesis project in the first semester of 2023. The objectives of the master thesis project will be to code and document a computationally efficient QP solver based on a newly developed algorithm based on ramp functions. The tasks of the present project are:

1. Obtain detailed knowledge of the algorithm that solves Quadratic programming with ramp functions presented in [1]

2. Optimize the numerical algorithm proposed in Quadratic programming with ramp functions and fast QP-MPC solutions in Python

3. Learn to use tools to analyze the complexity of a program trough Profiling

Although Matlab is seen as a more efficient and convenient programming language for numerical algorithms, the decision was to implement the algorithm in Python in this project. [4] This was with the objective of learning about optimization of numerical algorithms in a language well known for the student so that when implementing the code in a more efficient language later, preferably C, bugs could be avoided. Due to the complexity and lack of experience in C, it was considered that this order of the tasks would be most efficient and lead to a better outcome.

1.3 Delimitations

The task have been limited to optimizing the code in Python in regards of runtime. The memory consumption was not intended optimized. The concept of sparse matrices, used in the original Matlab code to save memory space was neglected in the Python translation. This will be considered later, when implementing the algorithm in C in the Master thesis. The Python version was run with two different models which was decided to be enough to do the analysis objected in this project. The code is written with a goal of maintainability, so one could easily change the models if desired. Additionally, the code was only tested and run on one single computer. The code was written with a goal of efficiency, the code quality where not given a high priority. The MPC-QP controller was not optimized in regards of fast responses and fast convergence to the steady state of systems.

1.4 Structure of the report

First, the theory is presented in chapter 2, which provides the necessary theoretical knowledge for the reader.The methods and tools used when carrying out the project is described in chapter 3. In chapter 4, the results from the applied methods is presented along with the implemented code improvements. The discussion is to be found in chapter 5 which also provides recommendations for further work. Finally, in chapter 6the conclusion of the project outcome is presented.

Chapter 2

Theory

MPC is an advanced but yet popular controller with a great success in the industry. The controller require a solution to a mathematical optimization problem in every time-step and many MPC applications require a rapid solution of the optimization problem. In addition, this is often implemented on computers with a limited memory capacity and computation capacity. The method used in this project aims to contribute with an efficient, minimal space-consuming solution. The method is based on solving the KKT conditions, which are necessary conditions for optimal solutions and sufficient conditions for convex optimization problems.

Consequently, this chapter will give an introduction to mathematical optimization and the KKT conditions for optimal solutions. The chapter will in addition give the reader necessary knowledge of MPC controllers, QP-problems and the algorithm proposed for efficient solving. The objective is to optimize the runtime of the algorithm and thus some theory of code analysis tools and efficiency in Matlab and Python is presented.

2.1 Mathematical optimization theory

2.1.1 The optimization problem and feasible set

Mathematical optimization problems consists of

- *decision variables*, z, the variables whose values need to be determined
- an *objective function*, $f(z)$, that provides a numerical value for the quality or desirability of a specific choice of values for the devision variables, and
- *constraints* that specifies conditions the decision variables have to fulfill in order for the values of the decision variables to be acceptable.

[5]. Constraints are divided into two types, equality constraints and inequality constraints. This allows the definition of an optimization problem

$$\min_{z \in \mathbb{R}^n} f(z)$$

subject to

$$c_i(z) = 0, \quad i \in \mathscr{E}$$
$$c_i(z) \geq 0, \quad i \in \mathscr{I}$$

The objective function f takes an n-dimensional vector, $z \in \mathbb{R}^n$, and projects it onto the real axis. $\mathscr{E}$ and $\mathscr{I}$ are disjunct index sets for the constraints. The solution for z may be selected from a subset of $\mathbb{R}^n$ called the feasible region. The feasible region consists of all solutions satisfying the constrains c_i.

$$\Omega = \left\{ z \in \mathbb{R}^n \mid (c_i(z) = 0, i \in \mathscr{E}) \wedge (c_i(z) \geq 0, i \in \mathscr{I}) \right\} \tag{2.1}$$

An optimization problem is said to be feasible if it exist a solution satisfying all the constraints, if no such solution exist, the problem is said to be infeasible.

2.1.2 Quadratic programming

Optimization problems can be categorized in different groups by their objective function and constraints. This project will focus on solving Quadratic programming problems, where the objective function is quadratic and the constraints are linear [6]. A QP standard problem is formulated by

$$\min_z 0.5 z^T \widetilde{H} z + c^T z \tag{2.2}$$

subject to the constraints

$$Lz \leq b \tag{2.3}$$

Where z is a vector of free variables in the optimization, $\widetilde{H}$ known as the Hessian matrix, c describes the linear part of the objective function, and L and b describe the linear constraints. To ensure that there exists a unique optimal solution to the optimization problem, the Hessian matrix has to be positive definite.

2.1.3 Active set

An active constraint is defined as a constraint for which $c_i(z) = 0$, which implies that all equality constraints are active at a feasible point. An inequality constraint may be active or inactive at a feasible point. The active set is defined as the set of active constraints at the solution, that is

$$\mathscr{A}(z) = \mathscr{E} \cup \{ i \in \mathscr{I} \| c_i(z) = 0 \} \tag{2.4}$$

2.1.4 First order necessary conditions

The Karush-Kuhn-Tucker (KKT) conditions define necessary conditions for optimal solutions to minimization problems. A Lagrange function is defined as [BLE DET RIKTIG?]

$$\mathscr{L}(z,\lambda) = f(z) + \sum_{i\in\mathscr{E}} \lambda_i c_i(z) + \sum_{i\in\mathscr{I}} \lambda_i c_i(z) \tag{2.5}$$

where the λ_i are the Lagrange multipliers. The KKT conditions is a key result in constrained optimization.

Theorem 1. *Assume that z is a local solution of (2.5) and that f and all c_i are differentiable and their derivatives are continuous. Further, assume that all the active constraint gradients are linearly independent at z. Then there exists Lagrange multipliers λ_i^* for $i \in \mathscr{E} \cup \mathscr{I}$ such that the following conditions (called the KKT conditions) hold at (x^*, λ^*).*

$$\begin{aligned} \nabla_z \mathscr{L}\left(z^*, \lambda^*\right) &= 0 \\ c_i\left(z^*\right) &= 0, \quad i \in \mathscr{E} \\ c_i\left(z^*\right) &\geq 0, \quad i \in \mathscr{I} \\ \lambda_i^* &\geq 0, \quad i \in \mathscr{I} \\ \lambda_i c_i\left(z^*\right) &= 0, \quad i \in \mathscr{I} \end{aligned} \tag{2.6}$$

2.2 Model Predictive Control

Model Predictive Control, MPC, is a method for process control used to control a process while satisfying a set of constraints. [6]. It is a multivariable control algorithm that uses an internal dynamic model of the process, a cost function to minimize over the moving horizon and an optimization algorithm that minimize the objective function with respect to the control input u. The optimization problem is typically formulated as Linear programming (LP) or Quadratic programming (QP). LP problems are more efficiently solved than QP problems, but QP problems leads to a smoother control action and is also what is used in the algorithm in [1]. This theory section will for this reason descibe the formulation of a QP-MPC problem.

2.2.1 Objective functions for discrete time systems

The algorithm in [1] aims to optimize a dynamic model sampled at discrete points in time. The objective functioon of dynamic optimization problems can be defined as

$$f(x_1, \ldots, x_N, u_0 \ldots, u_N) = \sum_{t=0}^{N-1} f_t(x_{t+1}, u_t) \tag{2.7}$$

The objective function is defined from $t = 0$ to $t = N$ where t is the time index. The time from 0 to N is called the prediction horizon. The objective function is given as the sum of the contribution from all time steps in the prediction horizon. The formulation of a MPC optimization problem begins with

a linear, discrete-time state-space model.

$$x_{t+1} = Ax_t + Bu_t + Ed_t \tag{2.8}$$

$$y_t = Cx_t + Fd_t \tag{2.9}$$

The subscripts define the sampling instants. Like in control literature, the state x, input u, disturbance d and measurement y should be interpreted as deviation variables, that represents the deviation of a set of variables from the obtained model.

2.2.2 MPC-QP optimization problem

A typical optimization problem in MPC is be given by

$$\begin{aligned}\min_{u,x} f(x,u) = \sum_{t=k}^{t=t'+N-1} &\Big\{\left(x_t - x_{ref,t}\right)^T Q\left(x_t - x_{ref,t}\right) \\ &+ \left(u_t - u_{ref,t}\right)^T P\left(u_t - u_{ref,t}\right)^T\Big\} \\ &+ \left(x_{t'+N} - x_{ref,t'+N}\right)^T S\left(x_{t'+N} - x_{ref,t'+N}\right)\end{aligned} \tag{2.10}$$

subject to constraints in (2.8) in addition to

$$\begin{aligned} x_0 &= \text{ given} \\ M_t x_t + N_t u_t &\leq G_t \quad \text{for } 0 \leq t \leq n-1 \\ M_n x_n &\leq G_n \end{aligned} \tag{2.11}$$

The objective function penalize deviation of states and inputs from their desired reference trajectories.

[BYTT K med T merket SOM I FIGUR]

2.2.3 The MPC concept

[LEGG til footnote som sier: i de fleste realistiske tilfeller så vil ikke målinger av tilstandene være tilgjengelig (ikke for alle tilstandene), derfor estimerer man dem og bruker de i stedet. Estimeringsproblemet er utenfor oppgavens scope.]

In MPC, an optimal solution is computed at every time step t to add feedback control [5] as the model is updated with a new initial condition for each time step. At every sampling instant, a finite horizon open-loop optimal control problem is solved. The current state of the plant is the initial state of the optimization problem and the first control in the calculated control sequence is applied to the plant. The MPC concept is illustrated in Figure 2.1. The optimization problem is solved at each time step for a finite horizon, but only the first control input u_t is applied to the plant. Then, in the next

step, MPC recalculates the whole optimization problem again for the horizon from t' to $t' + N$ with the updated state $x_{t'}$ as the initial value. MPC uses a *moving horizon*, which means that the MPC horizon changes at each time step, including the current step t' and the N next steps, where N is the prediction horizon.

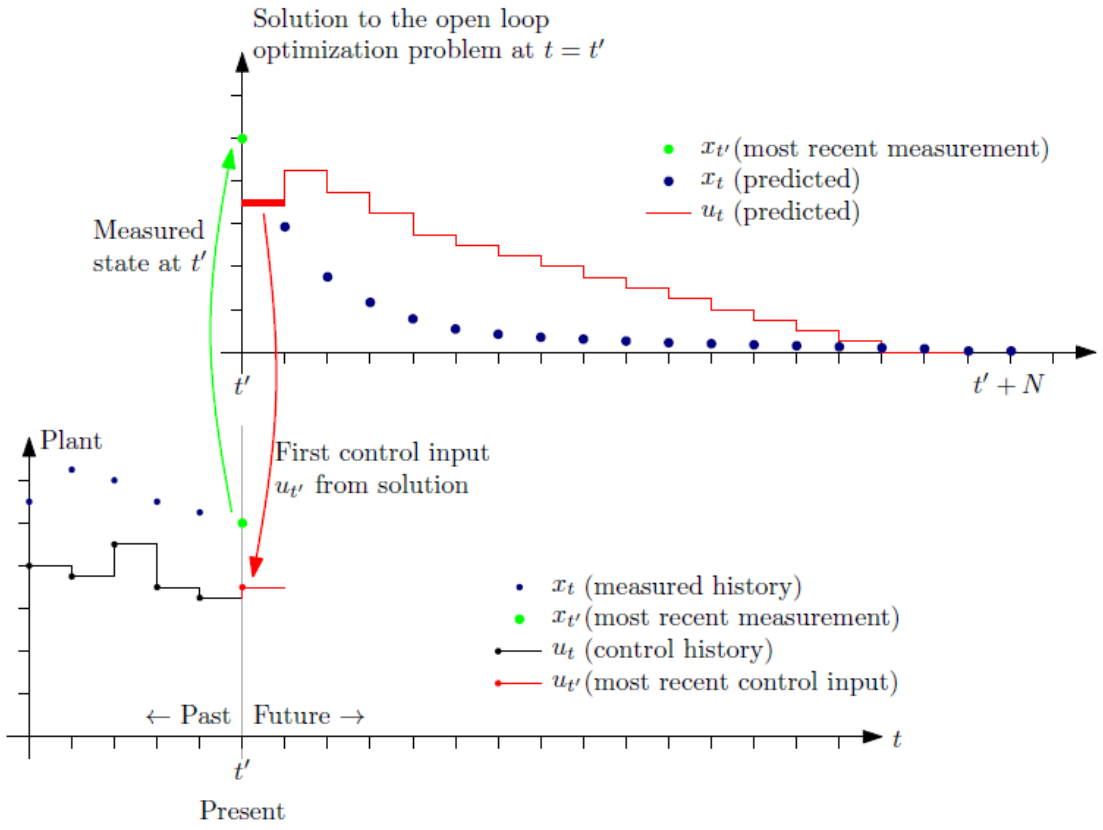

Figure 2.1: Illustration of MPC [5]

2.3 The algorithm

The algorithm described in [1] proposes a new method for solving MPC-QP problems more efficiently. The MPC-QP problem to be solved by the algorithm given in subsection 2.2.2 can be reformulated as in [7] to

$$\begin{aligned} \min_{u} \quad & \mathbf{u}^\top H\mathbf{u} + x(t)^\top F^\top \mathbf{u} \\ \text{subject to} \quad & \\ & G\mathbf{u} \leq S_u x(t) + w \end{aligned} \tag{2.12}$$

By application of the variable change $z = u + H^{-1}Fx(t)$ the formulation of the QP-MPC problem becomes

$$\begin{aligned} &\min_{z} \quad \mathbf{z}^\top H\mathbf{z} \\ &\text{subject to} \\ &\qquad G\mathbf{z} \leq Sx(t) + w \end{aligned} \tag{2.13}$$

where $S = S_u + GH^{-1}F$.

2.3.1 Ramp functions to define the active set

As explained in subsection 2.2.3, the MPC algorithm aims to solve a QP problem for the MPC horizon at each time step. The algorithm solves the optimization problem in (2.13) by the use of ramp functions.

Definition 1. *The ramp function r(y) is given by*

$$r(y) = \begin{cases} 0 & if\, y < 0 \\ y, & if\, y \geq 0 \end{cases} \tag{2.14}$$

Definition 2. *For $y \in \mathbb{R}^m$ we define the vector-valued ramp function $\phi : \mathbb{R}^m \longrightarrow \mathbb{R}^m$ by*

$$\phi(y) = \begin{bmatrix} r(y_1) \\ r(y_2) \\ \vdots \\ r(y_m) \end{bmatrix} \tag{2.15}$$

Theorem 2. *The solution to the QP-MPC problem given in section 2.2 is given by*

$$u(x) = -H^{-1}F^T x - H^{-1}G^T\phi(y) \tag{2.16}$$

$$y = -Sx + (I - GH^{-1}G_T\phi(y) - \omega \tag{2.17}$$

The proof of Theorem 2 is given in [1] and is not repeated here.

To solve the QP-MPC problem in section 2.2, (2.17) needs to be solved. For this, we take advantage from that the ramp function $\phi(y)$ can be expressed as $\phi(y) = I_{\mathscr{A}}\, y$ where $I_{\mathscr{A}} \in \mathbb{D}^m$ with its diagonals verifying $I_{\mathscr{A}\, i,i} \in \{0,1\}$. The binary diagonal matrix $I_{\mathscr{A}}$ is defined by

$$I_{\mathscr{A}\,(i,i)} = \begin{cases} 1 & \text{if } i \in \mathscr{A} \\ 0 & \text{if } i \notin \mathscr{A} \end{cases} \tag{2.18}$$

The set $\mathscr{A}$ is the set of active constraints, i.e. the set for which the values of the multipliers λ are not zero.

Definition 3. *Let $\mathbb{D}^N$ be the set of diagonal matrices of dimension N and $\mathscr{A} \subseteq \{1,2,...,N\}$. Let $I_{\mathscr{A}}$ be a diagonal binary matrix $I_{\mathscr{A}} \subset \mathbb{D}^N$ with $I_{\mathscr{A}} \in \{0,1\}$ where $I_{\mathscr{A}(i,i)} = 1$ if $i \in \mathscr{A}$ and $I_{\mathscr{A}(i,i)} = 0$ if $i \notin \mathscr{A}$. The complementary of the active set, $\mathscr{A}$, is defined as $\mathscr{A}^c = \{1,2,..,N\} \setminus \mathscr{A}$.*

Using the definition of the ramp function and re-arranging (2.17) results in:

$$y - (I - GH^{-1}G^T)I_{\mathscr{A}}y = -Sx - \omega \tag{2.19}$$

In the next section, the algorithm for solving to solve the MPC-QP problem will be presented.

2.3.2 Solution to (2.19) by ramp functions

The algorithm exploits the fact that the multipliers, λ, are given by the ramp function:

$$\lambda = \phi(y) \tag{2.20}$$

With the ramp function defining the multipliers, it yields that if $y_i < 0$, then the corresponding multiplier $\lambda_i = 0$, and in this case the corresponding diagonal elements of $I_{\mathscr{A}}$ is zero, $I_{\mathscr{A}(i,i)} = 0$. In the opposite case, $y_i \geq 0$, then $I_{\mathscr{A}(i,i)} = 1$. This can be summed up in a definition defining the pairs $(y, I_{\mathscr{A}})$.

Definition 4. *For a given x, the pair $(y, I_{\mathscr{A}})$ is said to be compatible if $(y, I_{\mathscr{A}})$ satisfy (2.19) and $I_{\mathscr{A}(i,i)} = 1$ if $y_i \geq 0$ and $I_{\mathscr{A}(i,i)} = 0$ if $y_i < 0$.*

Thus the solution to (2.17) is found whenever (2.19) is fulfilled with a compatible pair $(y, I_{\mathscr{A}})$. The algorithm to solve (2.17) searches for the set of active constraints by adding or removing elements to the active set $\mathscr{A}$, i.e. modifying the matrix diagonal of $I_{\mathscr{A}}$ and then update y.

To derive the equations of the solution we need to define the complement of the active set

Definition 5. *The complement of the active set A is defined as*

The derivation of a mathematical algorithm to solve (2.19) starts with defining

$$Q(A) = \left(I_{\mathscr{A}^c} + GH^{-1}G^{\top}I_{\mathscr{A}}\right) \tag{2.21}$$

and use this to write the solution to (2.19) by

$$\begin{aligned} y - \left(I - GH^{-1}G^{\top}\right)I_{\mathscr{A}}y &= -Sx - w \\ Q(\mathscr{A})y &= -Sx - w \\ y &= Q(\mathscr{A})^{-1}(-Sx - w). \end{aligned} \tag{2.22}$$

Consider the case where *i enters the active set* $\mathscr{A}$ i.e $i \notin \mathscr{A}$ and define $\mathscr{A} \cup \{i\}$. With

$$I_{\mathscr{A}_{+i}} = I_{\mathscr{A}} + e_i e_i^\top \tag{2.23}$$

and

$$I_{\mathscr{A}^c_{+i}} = I_{\mathscr{A}^c} - I_{\mathscr{A}^c(\cdot,i)} e_i^\top \tag{2.24}$$

we get

$$Q(\mathscr{A}_{+i}) = Q(\mathscr{A}) - \left(I_{\mathscr{A}^c} - GH^{-1}G^\top\right)_{(\cdot i)} e_i^\top. \tag{2.25}$$

where e_i is the ith vector of the canonical basis of $\mathbb{R}^N$.

The matrix $Q(\mathscr{A}_{+i})$ presents a sum of $Q(\mathscr{A})$ and to get a solution for y as in (2.22) the inverse of $Q(\mathscr{A}_{+i})$ is needed. To find $Q(\mathscr{A}_{+i})$ the matrix inversion lemma is used. For a rank 1 update of a matrix Q, the matrix inversion lemma can be stated as

Lemma 1. *Let $Q \in R^{NxN}$ be an invertible matrix and Q^{-1} be known. Then*

$$\left(Q + q_u q_c q_v\right)^{-1} = Q^{-1} - \left(q_c^{-1} + q_v Q^{-1} q_u\right)^{-1} \left(Q^{-1} q_u q_v Q^{-1}\right) \tag{2.26}$$

where $q_c \in \mathbb{R} \setminus \{0\}$ is a scalar, $q_u \in \mathbb{R}^N$ and $q_v^T \in \mathbb{R}^N$. that is q_u is a column vector and q_v is a row vector.

With the use of Lemma 1 with $q_c = -1, q_v = e_i^T$ and $q_u = \left(I_{\mathscr{A}^c} - GH^{-1}G^\top\right)_{(\cdot,i)}$, the inverse of $Q(\mathscr{A}_{+i})$ becomes

$$\begin{aligned} Q(\mathscr{A}_{+i})^{-1} =& Q(\mathscr{A})^{-1} \\ &- \left(-1 + e_i^\top Q(\mathscr{A})^{-1} \left(I_{\mathscr{A}^c} - GH^{-1}G^\top\right)_{(\cdot,i)}\right)^{-1} \times \\ &\left(Q(\mathscr{A})^{-1} \left(I_{\mathscr{A}^c} - GH^{-1}G^\top\right)_{(\cdot,i)} e_i^\top Q(\mathscr{A})^{-1}\right) \end{aligned} \tag{2.27}$$

By defining

$$v(\mathscr{A}, i) = Q^{-1}(\mathscr{A}) \left(I_{A^c} - GH^{-1}G^\top\right)_{(\cdot,i)}, \tag{2.28}$$

One can rewrite (2.27) as

$$\begin{aligned} Q(\mathscr{A}_{+i})^{-1} =& Q(\mathscr{A})^{-1} \\ &- (-1 + v(\mathscr{A}, i)_i)^{-1} v(\mathscr{A}, i) \left(Q(\mathscr{A})^{-1}\right)_{(i,\cdot)} \end{aligned} \tag{2.29}$$

The update of $y_+[i]$ is given by

$$\begin{aligned} y_+[i] =& Q\left(\mathscr{A}_{+i}\right)^{-1}\left(-Sx-w\right) \\ =& Q(\mathscr{A})^{-1}(-Sx-w) \\ & -\left(-1+e_i^\top Q(\mathscr{A})^{-1}\left(I_{\mathscr{A}^c}-GH^{-1}G^\top\right)_{(\cdot,i)}\right)^{-1}\times \\ & \left(Q(\mathscr{A})^{-1}\left(I_{\mathscr{A}^c}-GH^{-1}G^\top\right)_{(\cdot,i)}e_i^\top Q(\mathscr{A})^{-1}\right)(-Sx-w) \\ =& y-\left(-1+e_i^\top Q(\mathscr{A})^{-1}\left(I_{\mathscr{A}^c}-GH^{-1}G^\top\right)_{(\cdot,i)}\right)^{-1}\times \\ & \left(Q(\mathscr{A})^{-1}\left(I_{\mathscr{A}^c}-GH^{-1}G^\top\right)_{(\cdot,i)}\right)y_i, \end{aligned} \tag{2.30}$$

that is simplified with (2.28) to,

$$y_+[i] = y - y_i\left(-1+v(\mathscr{A},i)_i\right)^{-1}v(\mathscr{A},i) \tag{2.31}$$

which describes the update to y when adding a constraint to the active set, $\mathscr{A}$. Consider now the case where $i\in\mathscr{A}$ and define $\mathscr{A}_{-i}=\mathscr{A}\setminus\{i\}$, when i is removed from the set $\mathscr{A}$,

$$Q\left(A_{-i}\right) = Q(\mathscr{A}) + e_i^\top\left(I_{\mathscr{A}^c}-GH^{-1}G^\top\right)_{(,,i)} \tag{2.32}$$

The matrix $Q\left(\mathscr{A}_{-i}\right)$ presents a sum of $Q(\mathscr{A})$ and with the use of Lemma Theorem 1 with $q_c=1, q_v = q_v = e_i^T$ and $q_u = \left(I_{\mathscr{A}^c}-GH^{-1}G^\top\right)_{(\cdot,i)}$, the inverse of $Q\left(\mathscr{A}_{-i}\right)$ is given by

$$\begin{aligned} Q\left(\mathscr{A}_{-i}\right)^{-1} =& Q(\mathscr{A})^{-1} \\ & -\left(1+v(\mathscr{A},i)_i\right)^{-1}v(\mathscr{A},i)\left(Q(\mathscr{A})^{-1}\right)_{(i,\cdot)} \end{aligned} \tag{2.33}$$

which gives the update of y the following expression

$$y_-[i] = y - y_i\left(1+v(\mathscr{A},i)_i\right)^{-1}v(\mathscr{A},i) \tag{2.34}$$

The removing and adding of constraints from and to $\mathscr{A}$ given in (2.31) and (2.34) can be summed up with the following points

- Remove first from $\mathscr{A}$ constraints corresponding to negative y_i. If there are more than one such constraint, remove first the one corresponding to the most negative y_i.

- Then add to $\mathscr{A}$ a constraint that is not a member of and corresponds to a positive value of y_i. If there is more than one such constraint, add first the one corresponding to the largest y_i.

The steps above are implemented in Algorithm 1 below. Algorithm 1 is called inside Algorithm 2 which returns the input $u(x_k)$ to be applied to the plant.

Algorithm 1 Solution to the algebraic equation (2.17)

Require: $GH^{-1}G, \mathscr{A}, invQ, y$ satisfying $y = invQ(-Sx - \omega)$

while $(\mathscr{A}, y)$ not compatible **do**
  if $ind(sign(y,-1)) \cap \mathscr{A} \neq \emptyset$ **then**
    $L \leftarrow ind(sign(y,-1)) \cap \mathscr{A}$
    $i \leftarrow ind(min(y,L))$
    $v \leftarrow invQ(I_{\mathscr{A}^C} - GH^{-1}G^T))_{(\cdot,i)}$
    $\mathscr{A} \leftarrow \mathscr{A} \setminus i$
    $q_0 \leftarrow 1$
  else if $ind(sign(y,1)) \neq A$ **then**
    $L \leftarrow ind(sign(y,1)) \setminus \mathscr{A}$
    $i \leftarrow ind(max(y,L))$
    $v \leftarrow invQ(I_{\mathscr{A}^C} - GH^{-1}G^T))_{(\cdot,i)}$
    $\mathscr{A} \leftarrow \mathscr{A} \cup i$
    $q_0 \leftarrow -1$
  end if
  $invQ \leftarrow invQ - (q_0 + v_i)^{-1} v(invQ)_{(i,\cdot)}$
  $y \leftarrow y - y_i(q_0 + v_i)^{-1} v$
end while
return $\mathscr{A}, invQ, y$

Algorithm 2 Solution o the QP-MPC in section 2.2

Require: $GH^{-1}G, S, \omega, F$

while MPC running **do**
  Obtain $x(k)$
  $Qinv \leftarrow I$
  $\mathscr{A} \leftarrow \emptyset$
  $y \leftarrow (-Sx(k) - \omega)$
  $y \leftarrow Algorithm1(GH^{-1}G, \mathscr{A}, invQ, y)$
  $u(x_k) \leftarrow -H^{-1}F^T x_k - H^{-1}G^T \phi(y)$
  Apply $u(x_k)$ to the plant
end while

2.4 Python as a programming language

Python is a language with the benefit of easily read syntax bringing comfort and efficiency to the development process [8]. However, when it comes to run-time performance, a good performance is easier to obtain in low-level programming languages like for example C. In difference from Python, C is a compiled language, meaning the whole code is converted directly to machine code and executed by the processor. Python is a interpreted language, running through the code line by line and executing each command. The execution speed is what makes the difference between this two categories, which is why many Python packages are written in the compiled language C.

The solution to the trade-off between development efficiency Python provides and the run-time ef-

ficiency from a compiled language is to use a multi-language model, where a high-level language is used with libraries as an interface to software packages written in low-level languages. Python can be seen as a part of a system of software package solutions that provides a good environment for computational work. It is also significant that the scientific computing libraries are free and open source, that makes it possible for the developer to have insight in how the packages are implemented and with what methods.

2.4.1 NumPy

NumPy, Numerical Python, is a fundamental Python package for scientific computing. The core of NumPy is implemented in C, and thus provides high efficiency for array manipulation and processing. NumPy provides vectorized computations, which eliminates the need for many explicit loops over the array elements. When code is vectorized, it means that the vecotirzed operations are executed in optimized, pre-compiled C code [**?**]. In general, vectorized code comes with the advantages of more concise code, fewer lines and thus fewer bugs.

In NumPy, the arithmetic operations between arrays are done by applying batch operations which enables rapid computations. When avoiding loops, the code obtains higher performance. the power of vectorization is demonstrated with simple program examples given below.

```
from timeit import default_timer as timer
import numpy as np

arr = np.ones((100,1))
arr2 = np.zeros((100,1))
arr4 = np.zeros((100,1))

start = timer()

for i in range(100):
    arr2[i] = arr[i][0]*2
    arr4[i] = arr[i][0]*4

end = timer()
timetot = end-start
print(timetot)
```

Listing 2.1: Python example without vectorizing

```
from timeit import default_timer as timer
import numpy as np

arr = np.ones((100,1))
arr2 = np.zeros((100,1))
arr4 = np.zeros((100,1))

start = timer()

arr2 = arr*2
arr4 = arr*4
```

```
12
13 end = timer()
14 timetot = end-start
15 print(timetot)
```

Listing 2.2: Python example with vectorizing

The first program is written without the advantage of vectorization and takes 1.27e-04 seconds. The second program is written by the use of vectorization and it takes 1.25e-05 seconds to execute. The non vectorized program is 10 times slower. This simple example demonstrates how avoiding loops can affect the runtime and in programs with a big amount of computations the vectorization is essential for efficiency.

2.5 Analyzing and evaluating Python code

To be able to optimize code, it is necessary to have knowledge about what parts of a program that is requiring the most resources i.e. time or memory. For this, one can benefit from using some analyzing tools called Profilers [9]. A profiler creates a report containing details about functions in a program. In this section, the tool *Line Profiler* and *Memory Profiler* will be described.

2.5.1 *Line Profiler*

Line Profiler The *Line Profiler* analysis tool from the LineProfiler package is a tool to analyze the runtime of code. This tool make it possible to discover the runtime bottle neck and thus optimize this parts. The *Line Profiler* returns data for each line in the code in terms of Hits, Time, Per Hit, %Time and the Line Content [9].

The Hits gives the number of times this line was called while the Per Hit column gives the time of each call. The %Time represents the percentage of the total time taken by that specific line and the Line content gives the code of the line. The tool gave an overview of which lines one should attempt to optimize. An example of a terminal output after running *Line Profiler* can bee seen in Figure 2.2. The unit is μs.

```
Timer unit: 1e-06 s

Total time: 0.051297 s
Function: simulate at line 7

Line #      Hits         Time  Per Hit   % Time  Line Contents
==============================================================
     7                                           def simulate(iterations, n=10000):
     8         1        25393  25393.0     49.5      s = step(iterations, n)
     9         1         1914   1914.0      3.7      x = np.cumsum(s, axis=0)
    10         1           22     22.0      0.0      bins = np.arange(-30, 30, 1)
    11         1            4      4.0      0.0      y = np.vstack([np.histogram(x[i,:], bins)[0]
    12         1        23962  23962.0     46.7                     for i in range(iterations)])
    13         1            2      2.0      0.0      return y
```

Figure 2.2: The result of running *Line Profiler* on a function [9]

2.6 Matlab as a programming language

Matlab is an interpreted language that is well-suited for scientific computations, as it comes with various extension libraries that provide solutions to technical tasks [10]. In contrast to other programming languages, where more functions need to be made by the developer, Matlab offers hundreds of built-in functions. Furthermore, there are numerous specialized toolboxes that can be used to solve complex problems in specific fields. This makes MATLAB an ideal language for quickly prototyping new programs. Matlab have the possibility of vectorization, which demonstrated in section 2.4.1 contribute to efficiency. Matlab is faster than Python with Numpy in most arithmetic operations [4]. Matlab uses just-in-time compilation, meaning that it converts the code to machine language just before its executed, improving the speed of the code. However, Matlab, is not that widely used in the industry due to its high cost.

Chapter 3

Methods and tools

In this chapter, the models, methods and tools used in the project will be presented. The decisions and choices for the project will be described and justified. The installation process on the computer of the different tools have been omitted.

3.1 Models used

Two different models where used to run the code and compare with Matlab. The models are the same as in [1] as it made it simplified the possibility of comparison between the Matlab version and the translated version. It was considered to be adequate with two models, one with 2 states and one with 4 states. Two models were considered sufficient to do the desired analysis of runtime and comparison with the Matlab version.

Model 1 is a double integrator example given by the system dynamics

$$x(k+1) = \begin{bmatrix} 1 & 1 \\ 0 & 1 \end{bmatrix} x(k) + \begin{bmatrix} 1 \\ 0.3 \end{bmatrix} u(k) \tag{3.1}$$

with constraints

$$-1 \leq u(k) \leq 1, \quad \begin{bmatrix} -5 \\ -5 \end{bmatrix} \leq x(k) \leq \begin{bmatrix} 5 \\ 5 \end{bmatrix} \tag{3.2}$$

The prediction horizon $N = 10$ is used, the initial state is given by

$$x_0 = \begin{bmatrix} 5 \\ -2 \end{bmatrix} \tag{3.3}$$

Model 2 is a four-state system with discrete-time dynamics given by

$$x(k+1) = \begin{bmatrix} 0.928 & 0.002 & -0.003 & -0.004 \\ 0.041 & 0.954 & 0.012 & 0.006 \\ -0.052 & -0.046 & 0.893 & -0.003 \\ -0.069 & 0.051 & 0.032 & 0.935 \end{bmatrix} x(k) + \begin{bmatrix} 0 & 0.336 \\ 0.183 & 0.007 \\ 0.090 & -0.009 \\ 0.042 & 0.012 \end{bmatrix} u(k) \tag{3.4}$$

with constraints

$$\begin{bmatrix} -1 \\ -1 \end{bmatrix} \leq u(k) \leq \begin{bmatrix} 1 \\ 1 \end{bmatrix}, \begin{bmatrix} -1 \\ -1 \end{bmatrix} \leq Cx(k) \leq \begin{bmatrix} 1 \\ 1 \end{bmatrix}. \tag{3.5}$$

The prediction horizon is $N = 30$ and the initial state is given by

$$x_0 = \begin{bmatrix} 25.5724 \\ 25.3546 \\ 9.7892 \\ 0.2448 \end{bmatrix} \tag{3.6}$$

In the process of inverstigating how to improve the code, the code was run onlu on model 2 and the reults where compared with the results obtained in MATLAB. This was done under the assumption that improving the runtime with model 2 will also improve the runtime with model 1. This assumption was tested at the end of the improvement process and the results can be found in the next chapter.

3.2 NumPy

The Numerical Python library, NumPy, was chosen due to its efficiency described in 2.4.1. There are other packages for scientific computing in Python such as Pandas and SciPy, but NumPy was considered to be the one fitting this project best as it is simple to use and is a popular package with a big amount of resources.

3.3 Code translation

The code was translated from Matlab to Python by the use of NumPy for the array manipulation and the arithmetic operations. The code was attempt translated as directly as possible, so that it was possible to do a comparison of the two versions. This made impact on the code quality, but the objective of the project was efficient QP solving, that is readability was neglected. The source code editor used was Visual Studio code from Microsoft and the interpreter used was Python 3.10.8 64-bit.

In the process of code translation, some problems where encountered. The first problem in the process was a problem with termination for the script running the MPC for model 1. The script where able to run model 2 without problems. This was investigated and it was discovered that the code was stuck in a loop and could not solve the QP problem at one of the time steps. This problem led to a time demanding process with a lot of debugging and research. It turned out that the original Matlab code was a previous version and after receiving a newer version and applying these adjustments to the Python code it terminated without problems for both models.

3.3.1 Code structure

The code where divided into separate files calling each other. In Matlab these files needs to be executed in a specific order, to set the variables needed by the other files. In Python, however, when importing a file dependency at the beginning of the file, it is only necessary to run one file, as this instruction execute the dependent files automatically. All implemented Python files can be seen in [APPENDIX].

MPC_setup.py sets all required matrices and variables for defining the MPC optimization problem for the chosen model. For simplicity, some of the constant matrices where read from the setup file in Matlab, so that the amount of code needed to set variables was decreased. The file *MPC_run.py* runs the MPC algorithm and solves the QP problem for each time step. The QP problem was for each time step solved in a while loop, adding and eliminating constraints from the active set. The MPC algorithm returned an x array with all the states calculated in the MPC and a u array with all calculated inputs to the model. These arrays was initialized with zeros and filled with one element at each iteration of the for loop, i.e. at each time step.

3.3.2 The use of debugger

While implementing the code, the Python debugger in VS code was utilized. The debugger provided the possibility to set breakpoints, step-through the code and observe variables. The tool was utilized to compare the code with the Matlab code, using the debugger and the *Workspace* in Matlab to compare variables. As there was a lot of data to compare for matrices of high dimensions a technique of random sampling was utilized. That is, some values in a random row and random column where compared with the same row and column in the two versions.

3.3.3 Verification of the Python version

To verify that the Python algorithm gave the same results for input and output of the models as in the Matlab algorithm some plots where made. In Python, the package *matplotlib* was used, and the output trajectories *y_m* and input trajectories *u* where plotted. In Matlab, the same trajectories where plotted with the built in function *plot* and compared to the Python plots. The process of making the versions generate the same inputs where challenging and time consuming, as described in section 4.1.

3.4 Measurements of runtime

Both versions was configured to solve the MPC problem for 100 time-steps. The measurements where taken in the exact same lines in each version of the code.

In Python the $default\_timer$ function from the timeit-package was used to get the current time at the beginning and end of the desired code, then the difference between the timestamps where computed.

In Matlab, the function-pair tic and toc where used to measure the elapsed time between their calls.

Both timing methods rely on wall-time, that is, they measure the time spent on the execution, as oppose to f. eks CPU time which measures the time the CPU was busy.

To ensure comparable results, the code is always run on the same computer so that the hardware does not cause differences for the software performance. Two different timers for measuring computation time was implemented in each of the versions.

1. One timer was implemented to measure the time of each step, that is, the time used to solve the QP problem and compute the next input, $u(x)$, to the system.

2. One timer was implemented to measure the run-time of the complete algorithm for all 100 steps.

These timers where chosen in order to see how the solution to the QP problem evolves for each time step. Although it was possible to calculate the whole runtime by multiplying with the number of time steps, it was considered better to measure the whole algorithm runtime. The results from the Python code was saved to a .txt-file and the data was read by the Matlab function $readmatrix$. This gave the possibility to plot all data with Matlab and compare results.

3.5 Runtime analysis with the use of *Line Profiler*

To investigate what parts of the implemented code that consumed most of the computation time the tool $LineProfiler$ from the $line\_profiler$ Python Module was utilized.

There are several profiling tools available in Python, but after some research of the different options, the *Line Profiler* was chosen due to the data it generated and also the simplicity of use. *Line Profiler* generate data for each line and was considered to be straightforward to understand.

The *LineProfiler* required a function to analyze which lead to some restructuring of the code. The code in *MPC_run.py* was put inside a **main** function which was called at the end of the file. An additional file. *lineproftest.py*, was written to call the *Line Profiler* on the **main** function. The script called to use

Line Profiler is given below.

```
from line_profiler import LineProfiler
from MPC_run import main

lp = LineProfiler()

main = lp(main)
main()
lp.print_stats()
```

Listing 3.1: linproftest.py

When using LineProfiler to run the MPC_run script, the total run-time of the script increases in comparison to running the script directly. However, calling MPC_run from a different script with *LineProfiler* is still a useful tool for run-time analysis.

The strategy for optimizing the code used in this project can be summarized as the following approach in 6 steps:

1. Run *MPC_run.py* with *LineProfiler*
2. Interpret the results and search for lines with high impact on total time i.e. with a high percentage in the **% Time column**
3. Consider options for restructuring and improving the selected line
4. Run *MPC_run.py* with *LineProfiler* and keep the adjustment if the performance is better
5. Verify that the results are the same as expected i. e. run *plots.py* and verify the plots
6. Repeat the algorithm from step 2

The process provides a method to approach the problem, but contains no instruction on how to execute step 3 as this needs to be customized to the line content of the specific line. There is also not a limit on how many times this algorithm could be executed when attempting to improve a program. It will be a reasonable choice to prioritize to optimize the lines with a high **% Time**, but the **Hits** should also affect the prioritization. Eventually, when the runtime has increased considerably it will be reasonable to move on to other optimization techniques.

Chapter 4

Results and improvements

This chapter presents the results of implementing the methods described in the previous chapter, as well as the actions taken to improve the code. The chapter also contains comparisons of the old and the new code.

4.1 Validation of the versions

To validate that the Python translated code worked as expected, the input to the system, $u(x_k$, and the systems state trajectories x was plotted in both Matlab and Python. The Python module $matplotlib$ was used for the Python version, while the Matlab build-in function $plot$ was used for the Matlab version.

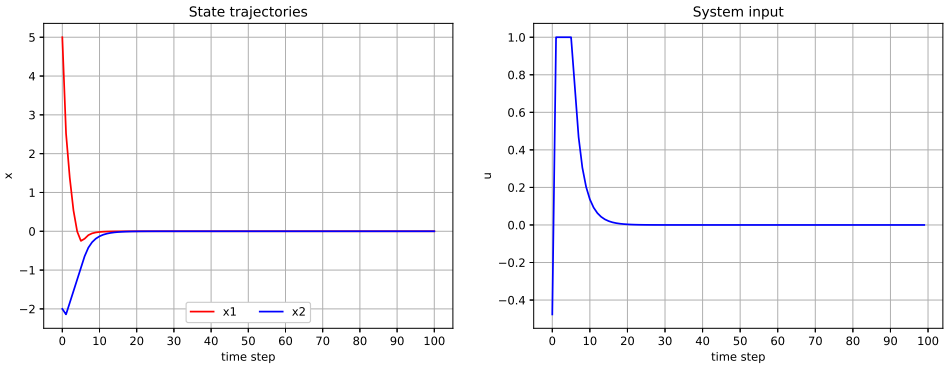

Figure 4.1: Input and state trajectories for the implemented Python translation, model 1

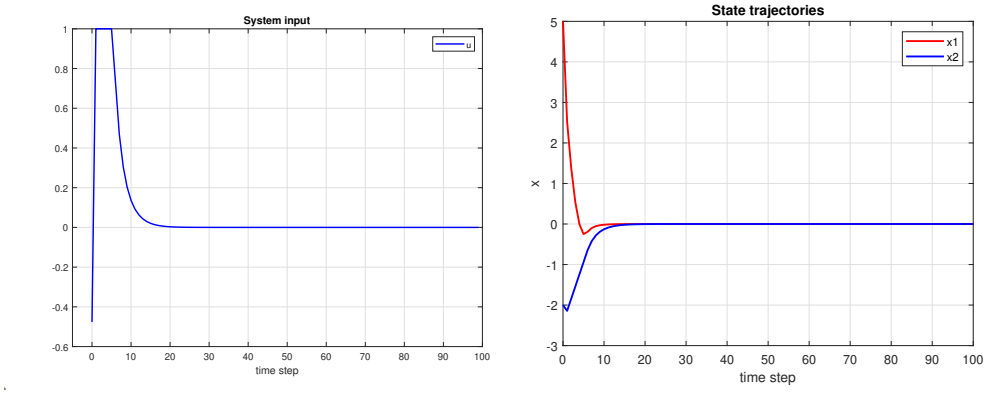

Figure 4.2: Input and state trajectories for the Matlab original, model 1

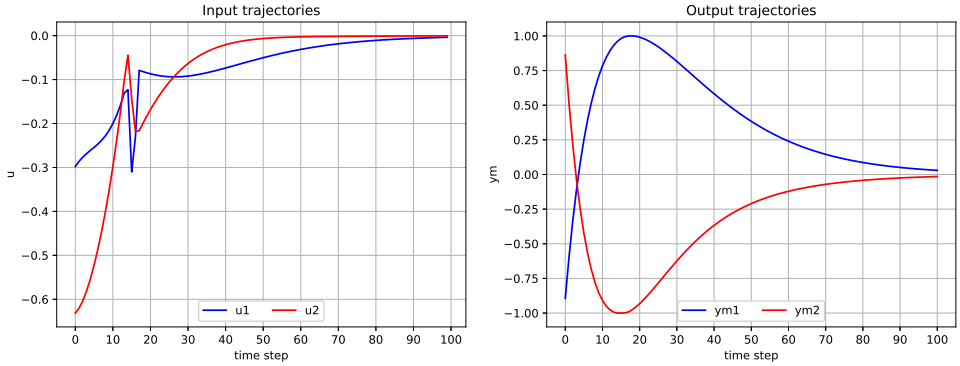

Figure 4.3: Input and output trajectories for the implemented Python translation, model 2

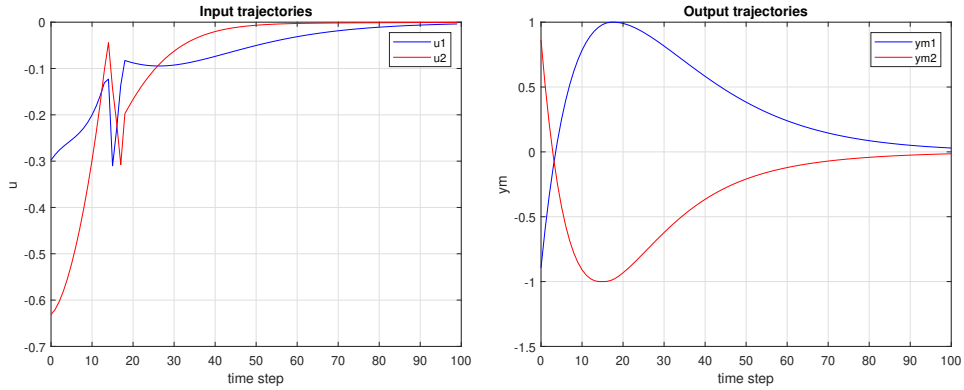

Figure 4.4: Input and output trajectories for Matlab original, model 2

As seen in the plots, the two versions coincide for model 1, but there is a deviation for the input u2 starting in step 17. The deviation biggest at step 17, before the difference gets slightly less noticable towards the end of the computation. This deviation posed a challenge that that persisted until the end of the project timeline. The process of optimizing the code continued, but troubleshooting this issue was given a high priority and a significant amount of time. The solution was eventually found after extensive debugging, stepping through the code, and comparing values in MATLAB and Python.

The problem was known to occur at step 17 in the for-loop after validating the vectors at each time step until step 17. The bug was identified by stepping through both the MATLAB program and the Python program at step 17 and carefully examining the updates to the variables generated by each line of code.

The difference appeared when an if-sentence checked for a variable to not be empty, meaning we wanted to add a constraint to the active set, line 8 in Algorithm 1. In the MATLAB version, in line 14 in Listing 4.1, the variable was evaluated to not be empty and the program went inside the if-sentence, while in the Python program, line 11 Listing 4.2, the if-sentence was evaluated to be false. This difference caused the deviation found in the plots first compared.

[BURDE DETTE FLYTTES TIL DISKUSJONEN?] The problem was caused by a error generated from translating the code from MATLAB to Python. The original MATLAB code whose translation caused the error is seen in Listing 4.1. As Python is zero indexed, the evaluation of the if-sentence in line 4 in Listing 4.2 was incorrect when the index was 0. After discovering the bug, the fix was not complex to implement, the evaluation and the setting of the variables was changed as seen in Listing 4.3. [SKAL PRVE Å ENDRE STILENE SOM MATLAB OG PYTHON ER I HER; SÅ MATLAB SKILLER SEG UT]

```
1  ...
2              if (i2 <= 0)
3                  i2 = [];
4              end
5              if(~isempty(i2))
6                  iz = i2z;
7                  actset(iz) = 1;
8                  qc = -1; % Add constraint to active set
9              else
10                 iz = [];
11             end
12         end
13
14         if(~isempty(iz))
15                 qu = IGis(:,iz);
16                 vA = Qmat0i*qu;
17                 qdiv = qc+vA(iz); %qc = 1/qc
18         ...
19     ...
```

Listing 4.1: The original code snippet from MPC_runṁ

```
1  ...
2              if (i2 <= 0):
3                  i2= []
4              if (i2):
5                  iz = i2z
6                  actset[iz] = 1
7                  qc = -1
8              else:
9                  iz = []
10
```

```
11          if (iz):
12              qu = IGIs[:, iz]
13              vA = np.transpose(np.matrix(Qmat0i@(qu)))
14              qdiv = qc+vA[iz]
15
16          ...
17      ...
```

Listing 4.2: Python code causing deviation due to wrong variable setting

```
1  ...
2              if (i2 <= 0):
3                  i2= "NULL"
4              if (i2 != "NULL"):
5                  iz = i2z
6                  actset[iz] = 1
7                  qc = -1
8              else:
9                  iz = "NULL"
10
11         if (iz != "NULL"):
12             qu = IGIs[:, iz]
13             vA = np.transpose(np.matrix(Qmat0i@(qu)))
14             qdiv = qc+vA[iz]
15         ...
16     ...
```

Listing 4.3: Python code containing the solution after discovering what caused the deviaton

The changes of these lines resulted in a plot seen in Figure 4.5 equal to the one from the MATLAB original seen in Figure 4.4. It can be verified that the states, $x1$ and $x2$ from model 1 and the measurements $ym1$ and $ym2$ behaves as desired for an equilibrium point in the origin as they converge to 0 when time increases. The inputs u does also converge to 0, which is desired behaviour of controllers.

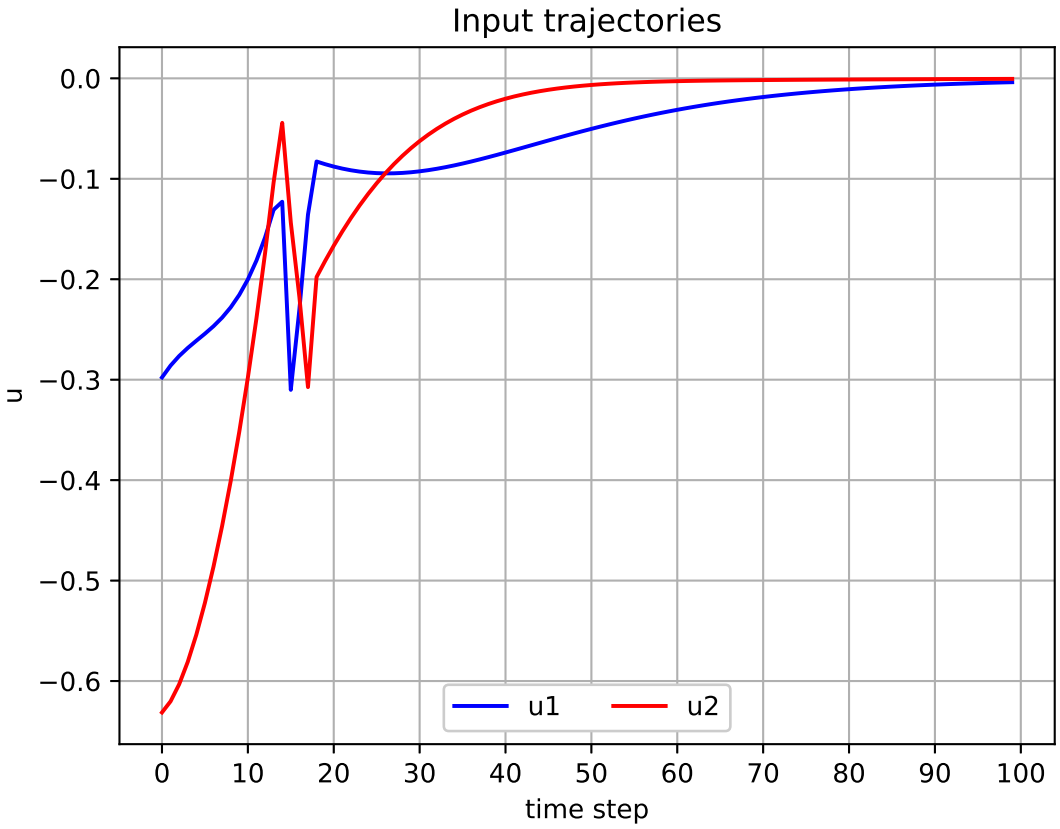

Figure 4.5: Input trajectories for Python translation after fixing the bug causing deviations, model 2

The time spent by the while-loop for each step in the for-loop for the two models are shown in Figure 4.6 and in Figure 4.7. The plots where made logarithmic for better comparison between the two programming languages.

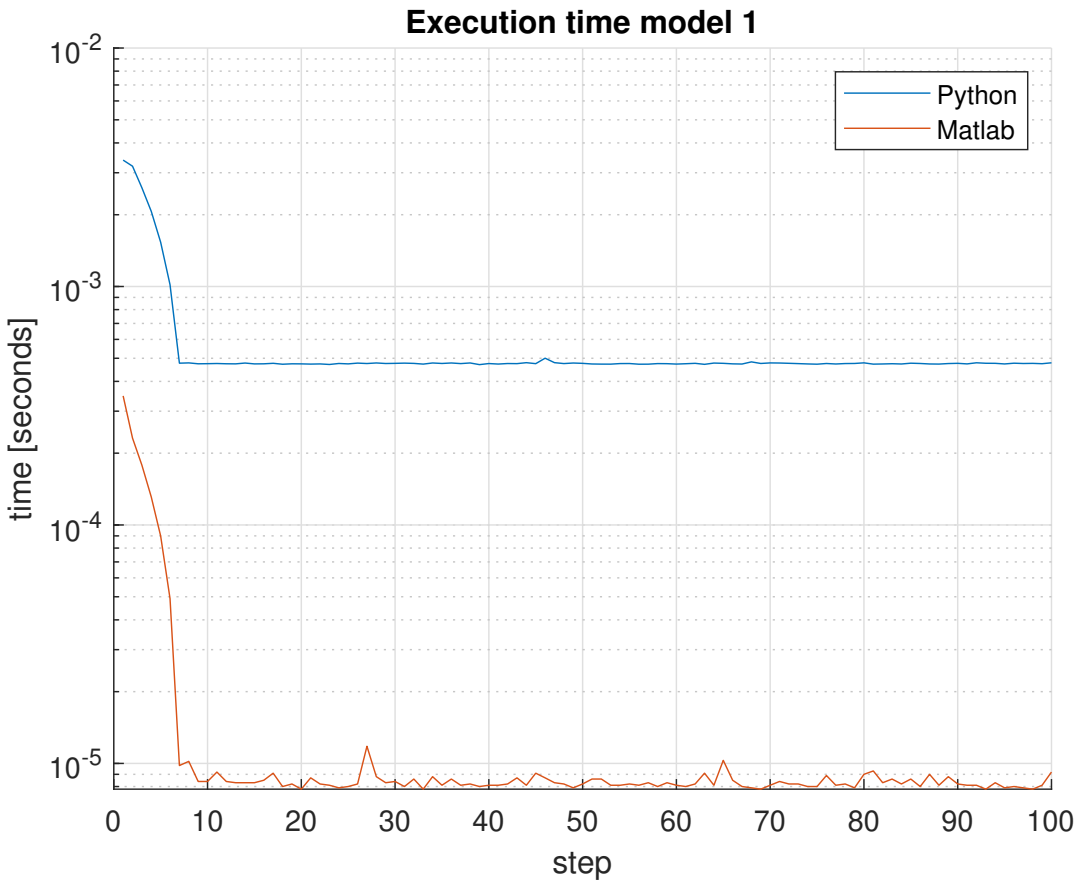

Figure 4.6: Run-time at each step in Python translation and Matlab original for model 1

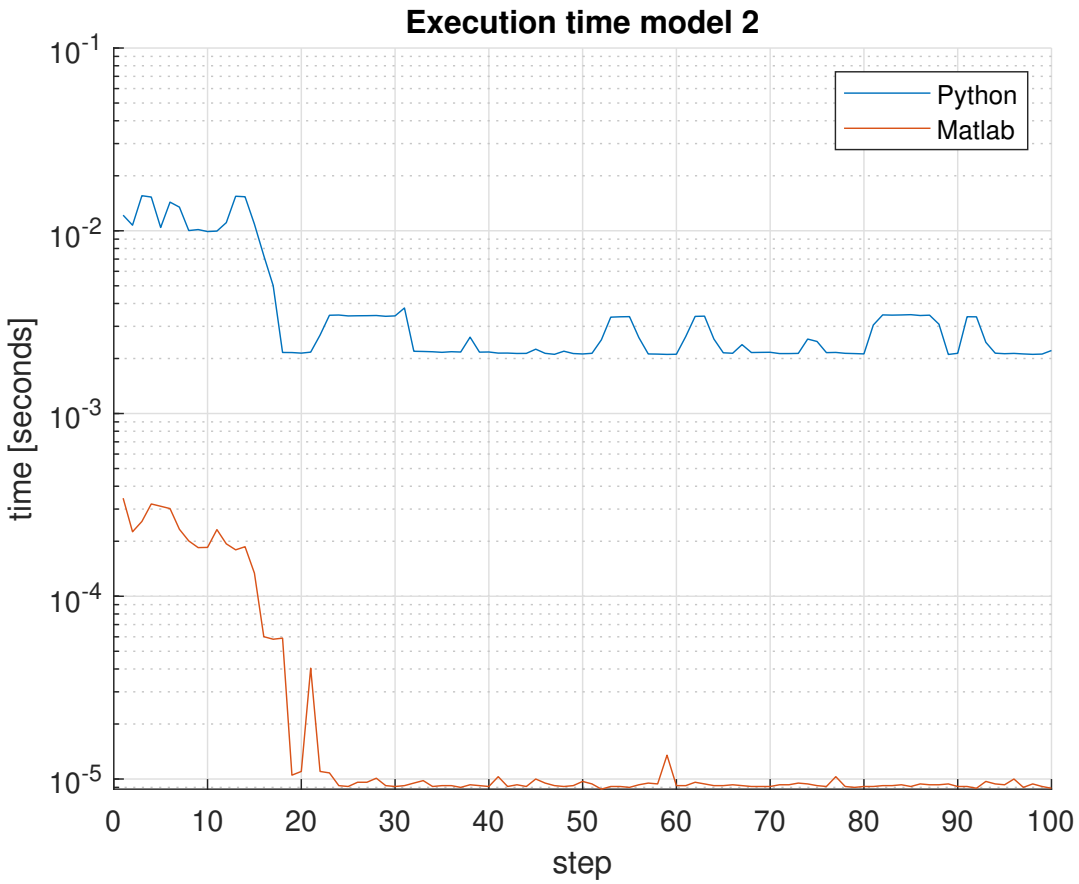

Figure 4.7: Run-time at each step in Python translation and Matlab original for model 2

In addition, the while-loops spent to compute the solution at each time step was saved for each version and plotted, this can be seen in Figure 4.8. The runtime for the Matlab original and the Python translation can be found in Table 4.1 and Table 4.2.

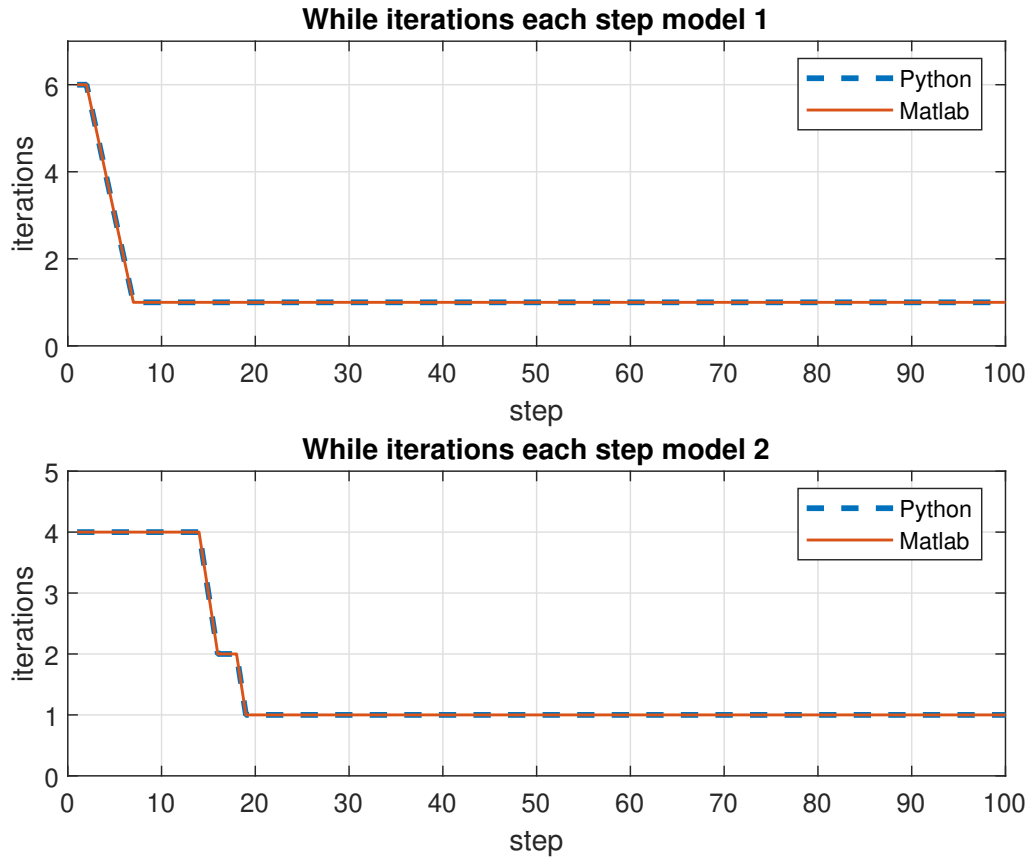

Figure 4.8: Number of while iterations at each time-step for model 1 and model 2

Model	n	N	max. number of iterations	steps with maximum iterations	avg. # of iterations pr time step	avg. runtime for 10 runs [s]
1	2	10	6	2	1.2	0.0608775
2	4	30	4	14	1.46	0.3733584

Table 4.1: Performance of Python translation

Model	n	N	max. # of iterations	steps with maximum iterations	avg. # of iterations pr time step	avg. runtime for 10 runs [s]
1	2	10	6	2	1.2	0.0019658
2	4	30	4	14	1.47	0.0045570

Table 4.2: Performance of Matlab original

4.1.1 Analysis with the use of *LineProfiler*

To investigate what parts of the implemented code that consumes most of the computation time the tool $LineProfiler$ from the $line\_profiler$ Python Module was utilized. A selection of some lines is given below in Table 4.3.

Line #	Hits	Time [μs]	Per Hit [μs]	% Time	Line Contents
38	1	1127.0	1127.0	0.0	$HGz = -Hinv@np.transpose(Gz)$
57	100	261.0	2.6	0.0	solved = False
72	46	18505.0	402.3	0.2	$y = np.subtract((y0), (y0[iz].item()) * vAd)$
76	146	4894693.0	33525.3	46.7	i1 = min(lam)
87	146	4951030.0	33911.2	47.2	i2 = $max(y - lam)$

Table 4.3: Table with LineProfiling results

The method described in section 3.5 was used to improve the Python code with the help from *Line Profiler*. The results and improvements are given in below.

Improvement 1 - the benefit of NumPy

The *LineProfiler* result from the directly translated Python version revealed that line 76 and 87 where particularly time-consuming. These lines are responsible for determining which element to remove or add to the active set, A, respectively line 4 and 10 in Algorithm 1. In addition to being slow, these lines are called more frequently than other lines of code, causing them to have big impact on the total run-time.

After some investigation, a replacement for line 76 was found, the replacement can be seen in Table 4.5. This replacement improved line 76 by a factor of 221 and line 88 by a factor of 170. The change of these two lines decreased the average run-time of the whole script with a significant factor as seen in Table 4.8.

Another line in the code performing a division of a matrix by a constant was improved by replacing *np.mulitply()* with *np.divide()* and doing some mathematical manipulation. The improvement of this was not of big impact on total time, but the **% Time** went from 1.3 to 0.9.

Line #	Hits	Time [μs]	Per Hit [μs]	% Time	Line Contents
76	146	4894693.0	33525.3	46.7	i1 = min(lam)
87	146	4951030.0	33911.2	47.2	i2 = $max(y - lam)$

Table 4.4: Code before improvement nr. 1

Line #	Hits	Time [μs]	Per Hit [μs]	% Time	Line Contents
76	146	21832.0	149.5	0.4	i1=lam.min()
87	146	29073.0	199.1	4.1	i2 = (y-lam).max()

Table 4.5: Code improvement nr. 1

Improvement 2 - dividing complex lines

The improvement changed the results from the *Line Profiler* as the decrease of impact on the total runtime leads to other lines increasing their impact. Another line that got high impact was the line where the update of Q is calculated. The data for this line can be seen in Table 4.6.

Line #	Hits	$Time[\mu s]$	Per Hit [μs]	% Time	Line Contents
108	46	191553.0	4164.2	34.9	Qmat1i = np.subtract(Qmat0i, vAd@np.matrix(Qmat0i[iz,:]))

Table 4.6: Table with extracted LineProfiling results before improvement 1

The improvement suggested was to split this calculation in two parts, one for calculating the subtrahend and one for calculating the difference. This adjustment gave the results shown in Table 4.7.

Line #	Hits	Time [μs]	Per Hit [μs]	% Time	Line Contents
106	46	111791.0	2430.2	19.7	vAdQmat0i = vAd@np.matrix(Qmat0i[iz,:])
107	46	43618.0	948.2	7.7	Qmat1i = np.subtract(Qmat0i, vAdQmat0i)

Table 4.7: Table with extracted LineProfiling results after improvement 2

This gave a new average runtime shown in **??**. After this result of dividing arithmetic operations in smaller parts, some other lines where attempted divided, but with no increased performance, thus the original code was kept. This discovery is discussed in chapter 5.

Improvement 3 - moving lines out of loops

Working with the method described in section section 3.5, another improvement was intended and succeed. The reset of the variable *Qmat0i* was done inside the for-loop with the NumPy call *np.identity(nc)*

consuming 10% of the total runtime. As a solution, the variable *Qreset* was mase outside the loop with the call *np.identity(nc)* which led to this line being called 1 time instead of 100. This led to a small improvement in average runtime, as seen in Table 4.8.

Python translation	After improvement nr. 1	After improvement nr. 2	After improvement nr. 3
0.3733584	0.0369917	0.0246870	0.0203659

Table 4.8: Average run-time of 10 runs of model 2 after each improvement

The total improvement in runtime when running model 1 is presented in Table 4.9. The assumption of improvement in model 2 implies improvements in model 1 was proved to be correct.

Python translation	After all improvements
0.0604434	0.0073402

Table 4.9: Average run-time of 10 runs of model 1 before and after all improvements

The runtime for each step after the improvements to the code can be seen in Figure 4.9 and Figure 4.10.

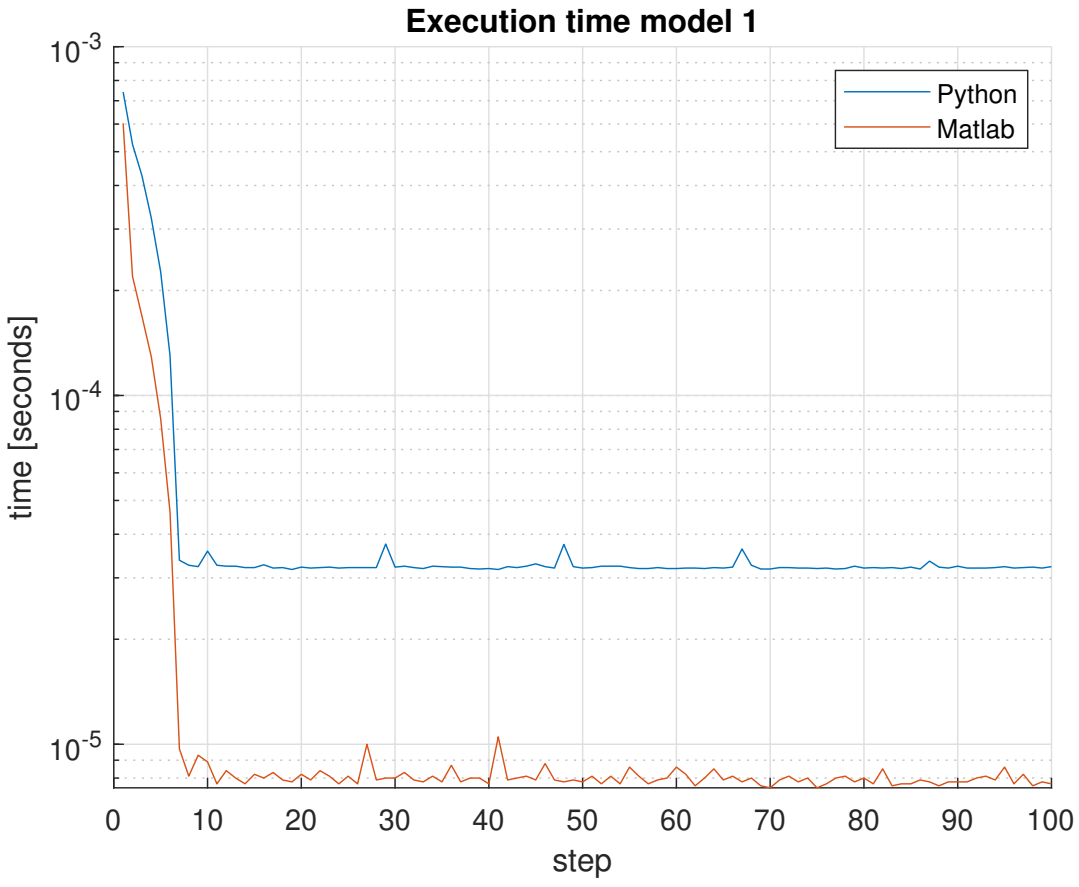

Figure 4.9: Runtime at each step in the improved Python code and Matlab original for model 1

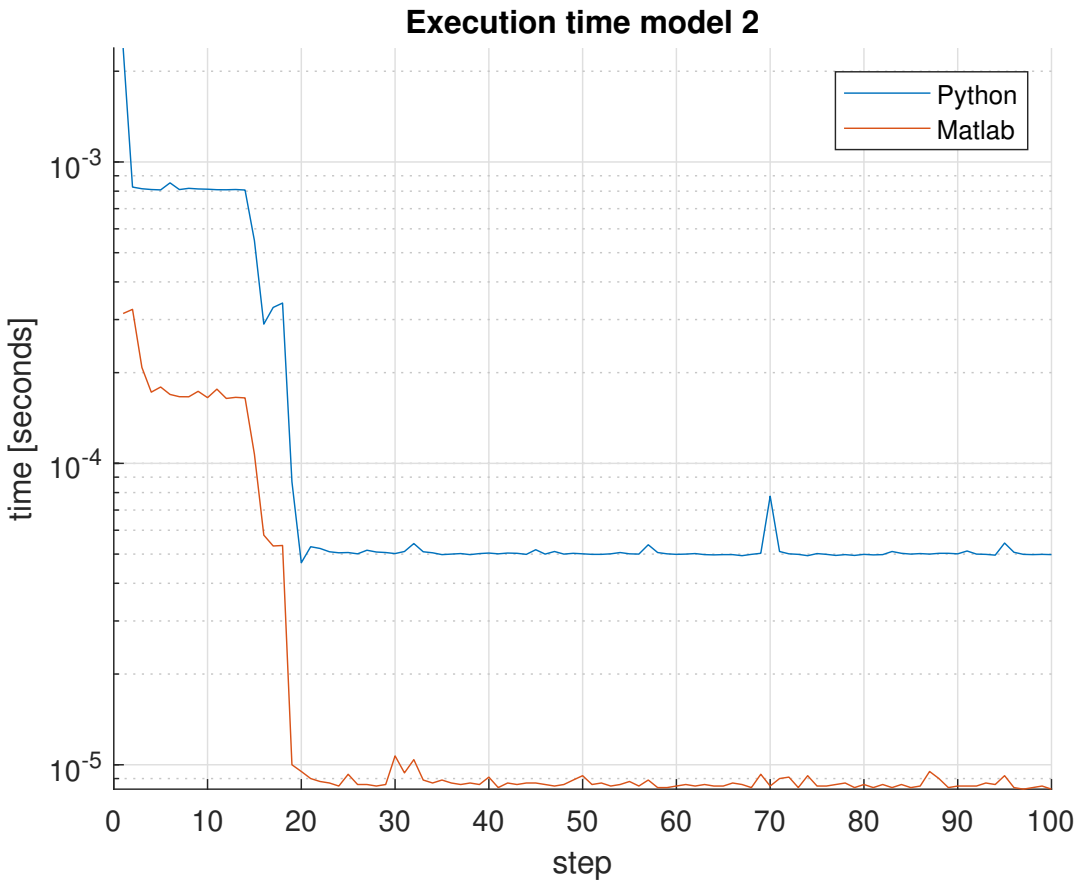

Figure 4.10: Runtime at each step in the improved Python code and Matlab original for model 2

Chapter 5

Discussion and further work

The code was implemented using Python NumPy and was intended runtime optimized. In terms of code quality, implementing complex numerical algorithms in NumPy did not lead to particularly readable code. While code quality was not a primary objective of this project, the lack of readability made the translation process more challenging. For future work, its recommended to carefully balance the trade-off between runtime performance and the readability, making the real-world application as maintainable and fast as possible. In terms of runtime, however, NumPy provided highly efficient mathematical operations, as demonstrated by the improvement in section 4.1.1.

Other alternatives to NumPy, such as Pandas, was considered. NumPy was ultimately chosen due to its simplicity of implementation and the abundance of research papers published on the topic. While Pandas may have offered similar levels of efficiency, the objective of this project was to investigate the algorithm and gain knowledge of analysis tools and optimization. As the code will be implemented in another language in future work, implementing it with NumPy was sufficient for achieving these goals.

One major challenge that arose during the development of the project was the process of accurately translating the MATLAB code to Python. This process was time-consuming and the fist draft resulted in deviations in the results of the controller.When translating code from one language to another, it is essential to carefully consider the differences in syntax, data types, and indexing between the two languages. It is also advisable to run tests on smaller portions of the code in a separate script to ensure that the same results are achieved in both languages. Adopting this approach more rigorously could have helped avoid the problem of deviation and reduced the need for debugging. This experience highlights the importance of thorough testing during code development and the use of small, independent code segments for translating tasks.

The improvements applied to the code led to improved runtime which was one of the objectives. The first improvement in section 4.1.1 proved the difference between Python arithmetic operations and NumPy arithmetic operations and how NumPy contribute to efficiency. This demonstrated how

the implementation of numerical operations can benefit from being implemented in lower level languages like C and give motivation for further work and implementation of the algorithm in C.

The improvement in section 4.1.1 was a realized by dividing a line of complex mathematical operations into two lines. This did not lead to a big decrease of runtime, but the decrease was notable. This method was attempted implemented to other lines with complex mathematical operations but without improvement in regards of runtime. This meant that the splitting of mathematical operations could not be a general method, which was the first intuition after discovering the first improvement. As Python is an interpreted language it would be a great practice to use one-liners when possible, however, as C is used in the NumPy package, the possibility of dividing complex operations have been proved to decrease the runtime as well. Overall, whether dividing lines of code will improve performance or not depends on the specific lines and operations being used, as well as the computer hardware and software environment in which the code is running. This practice must therefore be used together with profiling tools to see how the divided alternatives impacts the use of memory and runtime.

The improvement in section 4.1.1 aligns with the understanding that minimizing loops can improve code performance. This is demonstrated by the benefits of vectorization and the reasonable principle that reducing the number of times a line is executed leads to improved efficiency.

The total improvement of the code was not sufficient to compete with the runtime of Matlab. This was anticipated, as MATLAB has optimized toolboxes for matrix manipulation and arithmetic operations, whereas Python NumPy typically requires more runtime for most operations. However, as the algorithm aims to be a contribution to the industry, a version in Python is a step towards the goal. The project was still successful in achieving its objective of gaining knowledge about the algorithm and code optimization.

The concept of sparse matrices utilized in the Matlab version was not implemented in the Python version. This was due to NumPy not supporting sparse matrices and another package, SciPy, was required for sparse matrices. The SciPy package required different implementation of matrices and arithmetic operations that NumPy, and thus NumPy was kept. Sparse matrices is recommended to be implemented in future work in another programming language due to the memory optimization advantage of sparse matrices. During the work with this project, a tool called memory profiler was discovered, but not used as this was out of scope for the project. However, the use of memory profiler should be considered in future work on the finale code to better understand and optimize memory consumption.

This project has yielded valuable knowledge about code optimization with regard to runtime, which can be applied when implementing a final version of the algorithm in a lower-level language. The profiling analysis provided useful information and should be utilized in future work when implementing the final version of the algorithm. Some of the improvements made in this project are specific to NumPy and may not be applicable as general rules for use in other coding languages. However, other improvements can be transferred to future work and used as guidelines for optimizing code in C.

It is important to note that code optimization is a complex and nuanced field, and the specific approaches and techniques that are most effective can vary depending on the specific algorithms, data, and hardware and software environment in which the code is running. As such, it is essential to carefully analyze the code and experiment with different techniques in order to identify the most effective approaches for improving runtime.

Chapter 6

Conclusion

In conclusion, this study provided valuable knowledge and experience with optimizing a complex algorithm using the Python programming language. The program was significantly improved in regards of runtime by utilizing the NumPy package and various code improvements. The use of profilers as a tool for analyzing and optimizing code was proved to be essential. The knowledge gained from optimizing the code in Python is convertible to other programming languages and will useful in future efforts to implement the code in a lower-level language. Although the optimized code did not match the runtime performance of Matlab, which is specifically designed for efficient numerical calculations, the concepts and techniques learned in this project provide a strong foundation for future work in this area. Overall, this study contributes to our understanding of code optimization and has implications for the design and implementation of efficient algorithms in a variety of contexts. With a main goal of obtaining knowledge of the algorithm and runtime optimization in addition to learn about profilers the goal has been reached. The code result is a MPC-QP solver in Python which makes a step towards the development of a more efficient MPC-QP solver, which was the motivation for the project.

References

[1] G. Valmorbida M. Hovd. *Quadratic programming with ramp functions and fast QP-MPC solutions.* 2022.

[2] C. B. Alba E. F. Camacho. *Model Predictive Control.* Springer Science Business Media, 2013.

[3] A. Bemporad M. Rubagotti, P. Patrinos. *Stabilizing embedded MPC with computational complexity guarantees.* Institute of Electrical and Electronics Engineers, 2013.

[4] A. Elsherbeni A. Weiss. *Computational Performance of MATLAB and Python for Electromagnetic Applications.* 2020.

[5] T. A. N. Heirung B. Foss. *Merging Optimization and Control.* Department of Engineering Cybernetics Norwegian University of Science and Technology — NTNU, 2016.

[6] M. Hovd. *Advanced Chemical Process Control.* Course notes for TTK4210 Advanced control of industrial processes, NTNU.

[7] V. Dua A. Bemporad, M. Morari and E. N. Pistikopoulos. *The explicit linear quadratic regulator for constrained systems.* Automatica, 2002.

[8] R. Johansson. *Numerical Python: A Practical Techniques Approach for Industry.* Apress, Berkeley, CA, 2015.

[9] C. Rossant. *IPython Interactive Computing and Visualization Cookbook.* Packt Publishing, 2018.

[10] S. J. Chapman. *Matlab programming for engineers.* Cengage Learning, 2015.