Master's thesis

NTNU
Norwegian University of Science and Technology
Faculty of Natural Sciences
Department of Chemical Engineering

Erling Nicolay Selvig

Comparative analysis of NMPC performance for a semi-batch reactor optimisation problem with uncertain variables using state estimators Extended Kalman Filter and Moving Horizon Estimator

Master's thesis in Chemical Engineering
Supervisor: Johannes Jäschke
Co-supervisor: Simen Bjorvand
June 2023

NTNU
Norwegian University of
Science and Technology

Erling Nicolay Selvig

Comparative analysis of NMPC performance for a semi-batch reactor optimisation problem with uncertain variables using state estimators Extended Kalman Filter and Moving Horizon Estimator

Master's thesis in Chemical Engineering
Supervisor: Johannes Jäschke
Co-supervisor: Simen Bjorvand
June 2023

Norwegian University of Science and Technology
Faculty of Natural Sciences
Department of Chemical Engineering

PREFACE

This project was conducted as a master thesis at the Department of Chemical Engineering at NTNU and is a continuation of a specialisation project. The specialisation project was undertaken the previous semester and acts and preparatory for this thesis which aims to improve the concepts within state estimation and optimisation that were introduced. This thesis was performed in the last semester of the 5 year master program Industrial Chemistry and Biotechnology with a workload equivalent of 20 weeks.

I would like to thank my co-supervisor Simen Bjorvand for invaluable guidance throughout this work period which has been quite the learning experience for me. I am very grateful of his guidance and I am certain this thesis would not be the same without his involvement. Additionally, I would like to thank my supervisor Johannes Jäschke and the Department of Chemical Engineering for sparking my interest within process optimisation with their courses throughout my study period. Finally, I would express my thanks to my family and friends throughout the study period for their support and motivation.

ABSTRACT

The nonlinear model predictive controll (NMPC) is an optimisation technique which aims to compute control actions that ensures optimal operation of a process. This is done by calculating a sequence of control actions which purpose is to control predicted behaviour of the system and thereby provide operational stability. In order to perform this, the NMPC requires information of the process, often given in the form as ordinary differential equations (ODE), and the imposed constraints which limit the optimisation problem.

However, obtaining information of the process may not be straightforward in practice. For instance, uncertainty caused by external factors in the form of process and measurement noise could complicate this and result in operational instability if this type of random behaviour is not accounted in the NMPC. Another example are when states are simply immeasurable, either because they are economically expensive or physically impossible to measure, which for instance can be the case for a temperature sensor inside a rocket engine. If information about the optimisation problem is unattainable, the NMPC has to be provided state estimates instead and there are several methodologies to achieve this.

This thesis therefore aims to conduct a comparative study between two state estimators, Extended Kalman Filter (EKF) and Moving Horizon Estimator (MHE), which aims to provide filtered estimates to the NMPC based off noisy measurements of the states. This study is conducted on a semi-batch reactor, where the aim is to produce the product C from reactants A and B, where measurement data of the concentrations are unavailable. Finally, this thesis aims to include noise related uncertainty that had previously not been adressed in the preparatory work.

Based off the results, the EKF appeared to outperform the MHE as the most cost effective choice of state estimator for the NMPC as it was able to produce more C than the latter with less CPU time. However, the main cause for the lower performance of the MHE remains somewhat unclear, and further work towards improving the MHE, most notably regarding the choice of method for estimating the arrival cost, is encouraged.

SAMMENDRAG

Nonlinear model predictive control (NMPC) er en optimaliseringsteknikk hvor hovedmålet er å beregne kontrollhandlinger som forsikrer optimal drift av en prosess. Dette gjøres ved å beregne kontrollhandlingsekvenser som sørger for trygg og pålitelig drift av prosessen for fremtiden. Dersom NMPC-en skal beregne disse sekvensene trenger den informasjon om prosessen den skal kontrollere og dette oppgis som regel i form av ordinære differentialligninger og beskrankninger som er påført optimaliseringsproblemet.

I praksis er det derimot ikke alltid mulig å tilføre NMPC-en informasjonen den krever, enten fordi tilstandene er umålelige eller usikre. Usikkerhet i en prosess kan skyldes blant annet prosess- og målestøy og dersom NMPC-en ikke tar dette i betrakning når den løser optimaliseringsproblemet kan dette føre til sub-optimal eller upresis kontroll av prosessen. Dersom informasjon om tilstandene som NMPC-en kontrollerer er utiljengelige benyttes estimerte tilstander istedenfor og disse beregnes med hjelp av tilstandmålinger som inkluderer støy.

Et Prosjektarbeid ble utført forrige semester og hadde som hovedmål å implementere en NMPC på et allerede eksisterende optimaliseringsproblem av en semibatch reaktor med mål om å maksimalisere produksjon av et stoff, C, ut ifra reaktantene A og B. I praksis benytter modellen målinger som inkluderer målestøy og som dermed introduserer usikkerhet til NMPC-en, hvilket forble uadressert i prosjektarbeidet. I tillegg er to av tilstandene i reaktoren, konsentrasjonene av A og B, umålelige og av den grunn må tilstandestimatorer benyttes for å adresse nevnt usikkerhet. Hovedmålet for denne masteroppgaven er av den grunn å utføre et sammenligningsstudie mellom tilstandestimatorene, Extended Kalman Filter (EKF) og Moving Horizon Estimator (MHE) og undersøke hvilken av dem som er best egnet for modellen. Dette gjøres ved å undersøke hvilken av dem som resulterer i størst produksjon av C og som bidrar til forutsigbar og stabil drift av prosessen.

Ut ifra resultatene fremstår EKF som det mest kosteffektive alternativet grunnet en høyere produksjon av C med kortere CPU tid sammenlignet med MHE. Hva underpresteringen til MHE skyldes er uklart, men hvordan terminalkostnaden, Γ_k, har blitt estimert kan være en mulig forklaring. Av den grunn oppfordres videre arbeid til forskning innen andre måter å estimere denne variablen på, noe som forhåpentligvis kan utgjøre en positiv forskjell for ytelsen til MHE sammenlignet med EKF.

CONTENTS

LIST OF FIGURES

LIST OF TABLES

ABBREVIATIONS

	Description
CPU	Central Processing Unit (Time)
DAE	Differential Algebraic Equation
EKF	Extended Kalman Filter
KF	Kalman Filter
LQ	Linear Quadratic (Control)
MHE	Moving Horizon Estimator
MPC	Model Predictive Control
NMPC	Nonlinear Model Predictive Control
NLP	Nonlinear Programming
OC	Orthogonal Collocation
ODE	Ordinary Differential Equation
PDF	Probability Density Function

NOMENCLATURE

Latin letters

Symbol	Unit	Description
A_W	m^2	Inner surface area covered with reaction mixture
C_A	[mol/L]	Concentration of A
C_B	[mol/L]	Concentration of B
$C_{B,\text{in}}$	[mol/L]	Input concentration of B
C_C	[mol/L]	Concentration of C
C_P	[kJ/gK]	Specific heat capacity of the reactor contents
$C_{C,0}$	[mol/L]	Initial concentration of C
dt	[h]	Sampling instant
e_k	[-]	Innovation variable
h	[-]	Scaling parameter (Orthogonal Collocation)
H	[kJ/mol]	Reaction Enthalpy
k	[L/molh]	Reaction constant
KG	[-]	Kalman Gain
N	[-]	Maximum length of estimation window for MHE
$NFE0$	[-]	Length of NMPC prediction horizon
P_0	[-]	Parametric variables
$\dot{Q}_\text{K}$	[kJ/h]	Cooling input
r	[m]	Radius of cross-section of inner reactor
T	[m]	Length of estimation for MHE at sampling instant k
T_in	[K]	Temperature of inflow to reactor
T_J	[K]	Reactor Jacket Temperature
T_R	[K]	Reactor Temperature
v	[-]	Measurement noise
V_R	[L]	Reactor volume
u_k	[-]	Control inputs at each sampling instant
$\dot{V}_\text{in}$	[L/h]	Inflow input
$\check{x}$	[-]	A priori guess of state for EKF
$\hat{x}_k$	[-]	Estimated state at sampling instant k
x_k	[-]	Real state at sampling instant k
y_k	[-]	Measurement of states at each sampling instant k

Greek letters

Symbol	Unit	Description
α	[L/h]	heat-transfer coefficient between the reactor and jacket
Γ_{T-N}	[-]	Arrival cost for MHE
$\Delta\dot{Q}_K$	[kJ/h]	Variable cooling input at each sampling instant
$\Delta\dot{V}_{\text{in}}$	[L/h]	Variable inflow input at each sampling instant
ϵ_k	[-]	Slack variable at each sampling instant
π	[-]	Pi, Mathematical constant
ρ	[g/L]	Density of reactor contents
σ	[-]	Standard deviation
ω	[-]	Process noise

Matrices

Symbol	Description
A_k	Transition matrix
C	Measurement matrix
G	Process gain matrix
H	Measurement gain matrix
M	Weighting Matrix (Orthogonal collocation)
Π_k	State estimate error covariance matrix
Q	Process noise covariance matrix
R	Measurement noise covariance matrix

Units

Unit	Description
g	Gram
h	Hour
J	Joule
k	Kilo
K	Kelvin
kW	Kilowatt
L	Litre
m	Meter
mol	Mol

CHAPTER
ONE

INTRODUCTION

1.1 Background

For an industrial process there is always a desire to maximise the profit of a product while keeping the cost of producing it at a minimum. In the chemical industry, this approach is often labelled as process optimisation and mainly revolves around improving the efficiency and profitability of a chemical process by identifying sub-optimal behaviour and implementing changes that results in an increase of the operational goal. Such goals can for instance be directed towards improving the purity of a product, or recovering more value from the waste of an industrial process. However, reducing the carbon footprint while simultaneously ensuring economic sustainability remains the most important goal for large companies which aims to transition to a more sustainable business models. [1] This is especially the case for the energy sector with large corporations that are currently attempting to reduce their footprint by investigating alternative technology, such as carbon capture and electrification of their processes.

It is therefore important that industrial processes are optimised, either economically or emission wise. Additionally, processes often do have either soft or hard constraints imposed on them, such as economic budgets or safety measures that can for instance include temperature or pressure ranges to operate within. The process therefore has to be controlled while at the same time operate at optimal conditions. The MPC, along with its nonlinear variant, NMPC, attempts to achieve this by predicting future behaviour of a process and calculate control inputs that ensures optimal operation. This control method remains a popular choice of control in the industry and has displayed robust behaviour and a great ability to incorporate imposed constraints for a process. [2]

In order to operate at optimal conditions, the MPC needs to have knowledge about the states, for instance the temperature and volume of a process. These states may be straightforward to measure, but this may not be the case for other states that can prove to be more cumbersome to measure. The concentration, and therefore the quantity of a specific component in a mixture, can for instance be difficult to measure with sufficient accuracy. For instance, the formation of bi-products and the presence of multi-stage phases can complicate the measuring of the concentrations to the point where it may become economically infeasible to perform. Additionally, random process and measurement noise are present for every real process, which introduces a grade of uncertainty of the process and further exacerbate the difficulty of providing sufficiently accurate information to the MPC.

A proposed method to address these challenges is to estimate the states of the system and there are various methods which can achieve this, each with their respective advantages and disadvantages. The extended Kalman Filter (EKF) and moving horizon estimator (MHE) are two alternatives that are capable of producing estimates of unknown variables based off past measurements that are uncertain, or noisy. These methods are already well-established in the process industry as they have proven to produce usable estimates of unknown variables while maintaining a low computational cost compared to other estimation methods. [3][4]

Since their approaches are mathematically different, this thesis therefore aims to implement these methods on a specific optimisation problem in an effort to investigate which is the most cost effective choice when addressing the uncertainty of an optimisation problem. As there are many factors involved, a general and absolute conclusion for all processes will not make sense as each method has their own appeal for different processes. The aim of this thesis is however to indicate the different results and comment on the similarities and differences in an effort to highlight potential shortcomings of each method and encourage future research towards addressing these.

1.2 Literature review

Both the EKF and MHE are well-known state estimator techniques that have existed for several decades and extensive research have been conducted on each.[5][6] However, comparative analysis between the two methods seems to remain largely an unexplored topic. An exception to this is a notable research paper written by Eric. Haseltine and James Rawlings [7] which performs a critical evaluation of each method and compare their performance. Based off this study, it was concluded that the MHE overall provide improved state estimates compared to the EKF and is more robust to poor initial conditions. This come at the expense of an increased computational burden so a cost effective trade-off between accuracy and time do exist for the two methods.

However, this thesis, and similar comparative analyses of the state estimators, have primarily focused on solely producing state estimates based previous measurements. The main goal of this thesis is however to compare the performance of a control method, NMPC, based off state estimates which the EKF and MHE are to provide. In other words, this thesis aims towards investigating the practical consequences which the state estimators introduce to a specific control method, and threfore an optimisation problem as well.

1.3 Scope of project

In this thesis, the main workload is directed towards coding and implementation of the state estimators for the NMPC and compare the closed-loop performances of each. This task on its own requires significant time and effort, and in order to keep the workload at a realistic amount before the deadline, this thesis will use an already existing, optimisation problem derived by Sakthi Thangavel et. al. [8] It is also worth mentioning that throughout this thesis a series of assumptions and simplifications are to be made and what specifically these are is to be elaborated in later sections. The point is that this project, and most likely the results, will consist of a series inaccuracies based of assumptions that have been made in an effort to direct the scope solely towards the state estimators and their effect on the NMPC performance.

CHAPTER

TWO

BACKGROUND

2.1 Nonlinear Model Predictive Control

The main objective of an MPC is to calculate and provide control input sequences to the system which it monitors for optimal operation. It achieves this by solving an optimisation problem with given information of system dynamics and constraints through the use of a software optimiser. Despite calculating an entire sequence with finite length, it is only the very first element which the MPC calculates that is to be used. The idea behind this is that the MPC should be able to predict future behaviour, but only use the upcoming control action. This sequence is to be continuously updated with the goal of accounting future states during operation, thus ensuring predictability and operational stability of the process. An illustration of the overall objectives of an MPC is given in Figure 2.1.1:

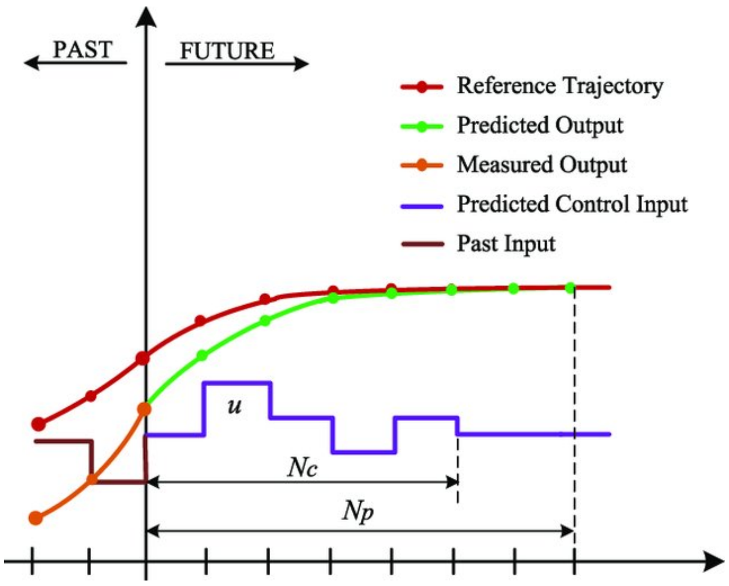


Figure 2.1.1: Illustration of the overall objectives of an MPC. With the dynamic model and constraints, the MPC predicts future behaviour of the system and calculates a control input sequence that ensures optimal operation in the future.[9]

With the main objective of the MPC presented, it makes sense to explain how the MPC calculates control input in a more mathematical sense. An example of an applied MPC can for instance be a car that has to follow a reference trajectory as a safety measure. In order to explain the concept of state and inputs, an illustration of the car example is given in Figure 2.1.2:

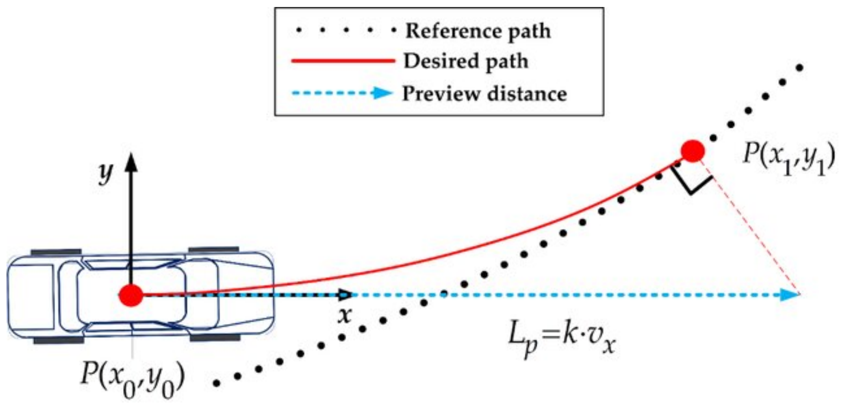


Figure 2.1.2: Example of the use of a MPC. The MPC attempts to control the trajectory of the car based of its current states, x_k. It does this by providing control actions that results in a minimisation of the deviation between the reference and real trajectory. [10]

For this example, the states of the dynamic optimisation problem will simply be its position along the respective x and y-axis. The states, or x_k, will therefore describes the current behaviour of the optimisation problem that is to be solved. With a reference trajectory, the MPC will, as mentioned previously, calculate a control action, or input sequence u_k^T, and execute the first element which for the car example ensures optimal operation, or safe driving.

If the MPC is to calculate control actions, it has to be provided an objective function which describes the dynamic optimisation problem. A generic formulation of a quadratic objective function using a linear model is given in Equation 2.1:

$$f(z) = \sum_{N}^{k=0} f_k(x_{k+1}, u_k) = \sum_{N-1}^{k=0} \frac{1}{2} x_{k+1}^T Q_{k+1} x_{k+1} + \frac{1}{2} u_k^T R_k u_k \tag{2.1}$$

With the objective function defined, the MPC will attempt to solve the following optimisation problem:

$$\min_{z \in \mathbb{R}^n} f(z) = \sum_{N-1}^{k=0} \frac{1}{2} x_{k+1}^T Q_{k+1} x_{k+1} + \frac{1}{2} u_k^T R_k u_k \tag{2.2}$$

subject to the following constraints:

$$
\begin{aligned}
x_{k+1} &= A_k x_k + B_k u_k, && k = 0, ..., N-1 && \text{(2.3a)}\\
x_0, u_{-1} &= \text{Given}, && && \text{(2.3b)}\\
x^{low} &\le x_k \le x^{high}, && k = 0, ..., N && \text{(2.3c)}\\
u^{low} &\le u_k \le u^{high}, && k = 0, ..., N-1 && \text{(2.3d)}\\
-\Delta u^{high} &\le u_k \le -\Delta u^{low}, && k = 0, ..., N-1 && \text{(2.3e)}\\
& && && \text{(2.3f)}
\end{aligned}
$$

where

$$
\begin{aligned}
Q_k &\ge 0, && k = 0, ..., N && \text{(2.3g)}\\
R_k &\ge 0, && k = 0, ..., N-1 && \text{(2.3h)}\\
\Delta u_k &= u_k - u_{k-1} && && \text{(2.3i)}\\
z^T &= (x_1^T, ..., x_N^T, u_0^T, ..., u_{N-1}^T) && && \text{(2.3j)}\\
& && && \text{(2.3k)}
\end{aligned}
$$

The matrices Q_k and R_k are most often diagonal and are considered as tuning parameters for the states and inputs respectively. The weighting $\frac{1}{2}$ of each term in the cost function may seem random, but this is conventionally used for a linear quadratic (LQ) control which the aforementioned equations are based off. [11]

However, the dynamic model that is provided to the MPC often consists of ordinary differential equations (ODE), which is an issue as they have to be integrated in order to provide useful information about the states. This is usually addressed with various collocation methods such as orthogonal collocation on finite elements, single and multiple shooting which aim to calculate numerical solutions to the ODEs with different methods. For this thesis, the former has been used as it has been proven to be efficient in generating accurate solutions within a short time frame [12]. A quick summary of orthogonal collocation can be found in Appendix 6.1.

As the previously described MPC utilises linear dynamic models, the accuracy of the controller be questionable as industrial processes often do not inherit ideal behaviour that can be considered as linear. A nonlinear system is considered nonlinear when it does not obey the principle of superposition, or in other words, when its output is not directly proportional to the input. [13] This is most often the case for a real process and in order to address this, the nonlinear variant of the MPC, NMPC, is proposed. Nonlinear problems are however significantly more challenging to solve compared to linear problems as the former can have multiple local solutions, rather than a single global solution. [14] Figure 2.1.3 is an illustration of how the solutions of an arbitrary nonlinear problem could look like:

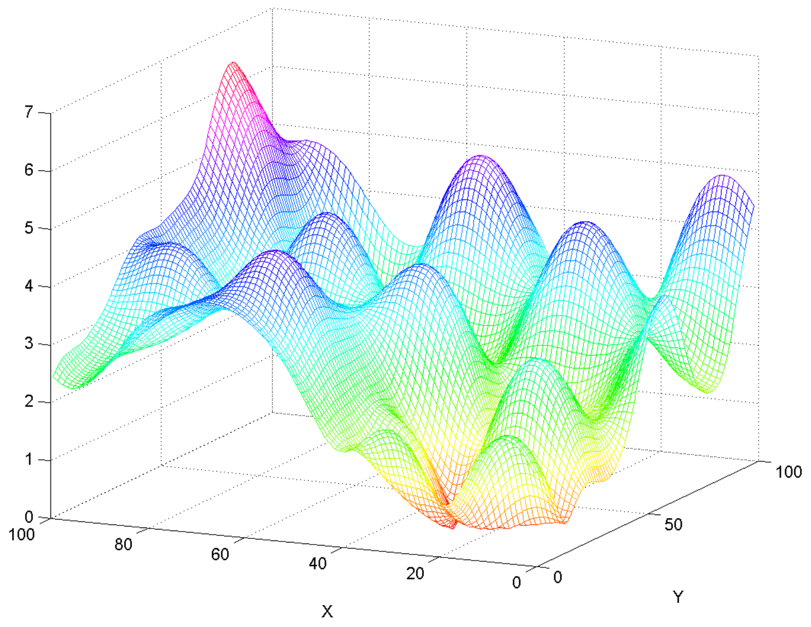


Figure 2.1.3: An illustration of the multiple solutions of an arbitrary nonlinear problem. [15]

Because of the multiple feasible solutions, it is often impractical to compute a global solution within a reasonable time period as the optimiser will struggle to settle for a specific solution. To address this, it is therefore suggested to use regularisation terms which purpose is to force the optimiser towards one specific local solution, rather continuously evaluate multiple when solving an optimisation problem. These terms are calculated based off subsequent control inputs and penalise the NMPC if these are significantly different .Without these terms, the optimiser would continuously evaluate all of the local solutions in the feasible set, something that can be considered counterproductive when trying to achieve numerical convergence towards a unique solution.

With a calculated input, the NMPC will provide the optimal input to a Plant which will, along with prior states, calculate the next real state of the optimisation problem. The plant can be defined with different methods, but in this thesis a standard ODE solver from the DifferentialEquations package in Julia is to be used for this purpose.

2.2 Process Description

A simple benchmark semi-batch reactor derived by Sakthi Thangavel [8] has been chosen as the basis for this thesis. The semi-batch reactor share many similarities to a batch-reactor, as both operate in a single stirred tank that has no outflow during operation. However, the main difference between them is that the former has the ability to use an inflow input during operation while the latter do not. [16] Additionally, both can be equipped with a reactor jacket which enables heat transfer which makes it possible to regulate the thermal energy of the system.

The semi-batch described by Sakthi Thangavel consists of a series of states, differential algebraic equations (DAE) and inputs. The operational goal of the process is to maximise the production of a desired substance, C, which is dependent on the concentrations of reactants A and B in the reactor. The chemical reaction that takes place in the reactor is:

$$A + B \rightarrow C \tag{2.4}$$

This reaction, and therefore the production of C, is dependent on the states,inputs and DAEs of the system. The states are; the reactor volume, V_R, concentration of reactant A, C_A, concentration of reactant B, C_B, the temperature of the reactor, T_R, and the temperature of the encapsulated heating jacket T_J. It is given in Thangavel's paper that the aforementioned reaction is exothermic, which means that heat is generated throughout the operation. The purpose of the jacket is thereforee to counter this by applying cooling through heat transfer with the reactor. The cooling power, Q_K, is therefore one of the two control inputs that can directly affect the optimisation problem.

V_{in} on the other hand denotes the inflow input, which is to be used when the concentration of the reactants needs to be regulated in order to ensure optimal operation. Finally, the concentration of C, C_C, and the inner surface area that is covered with reaction mixture, A_W, are the differential algebraic equations of the system that explicitly depend on some of the states.

Finally, the semi-batch reactor is to operate for an hour with a sampling instant $dt = 0.05$. This means that throughout the operation, the NMPC is to solve the optimisation problem every third minute. A visual representation of the semi-batch reactor is given in Figure 2.2.1:

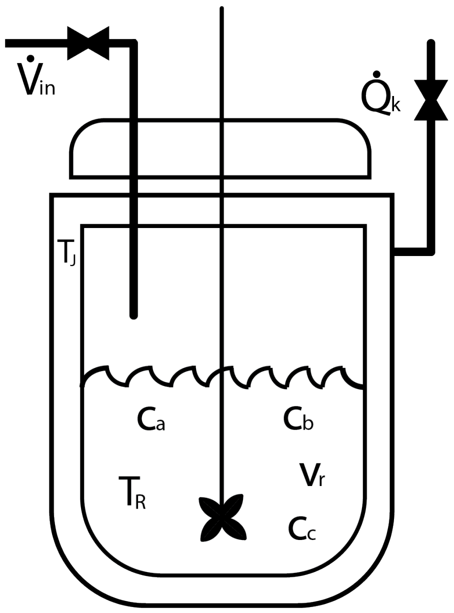


Figure 2.2.1: Illustration of the semi-batch reactor along with its states and inputs

2.3 Dynamic model

In order to solve the upcoming optimisation problem, the NMPC requires information about the semi-batch reactor, which is provided with the the dynamic algebraic (DAE) model derived by Sakthi Thangavel. This model consists of a series of differential equations which are:

$$\dot{V}_{\mathrm{R}} = \dot{V}_{\mathrm{in}} \tag{2.5a}$$

$$\dot{c}_{\mathrm{A}} = -\frac{\dot{V}_{\mathrm{in}}}{V_R} c_A - k c_A c_B \tag{2.5b}$$

$$\dot{c}_{\mathrm{B}} = \frac{\dot{V}_{\mathrm{in}}}{V_R} (c_{\mathrm{B,in}} - c_B) \tag{2.5c}$$

$$\dot{T}_R = \frac{\dot{V}_{\mathrm{in}}}{V_R} (T_{\mathrm{in}} - T_R) - \frac{\alpha A_W (T_{\mathrm{R}} - T_J)}{\rho V_{\mathrm{R}} c_p} - \frac{k c_A c_B H}{\rho c_p} \tag{2.5d}$$

$$\dot{T}_J = \frac{\dot{Q}_K + \alpha A_W (T_R - T_J)}{\rho V_J c_p}, \tag{2.5e}$$

and the algebraic equations which are explicitly dependent of the states V_R and C_A

$$C_C = \frac{c_{A,0} V_{R,0} + c_{C,0} V_{R,0} - c_A V_R}{V_R} \tag{2.6a}$$

$$A_W = \pi r^2 + \frac{0.002 V_R}{r} \tag{2.6b}$$

The DAEs are parameterised by a set of constants and are given in Table 2.3.1.

Table 2.3.1: Plant Model parameters

Parameter	**Description**	**Value**	**Unit**
α	heat-transfer coefficient	1700	$\mathrm{kJK^{-1}h^{-1}m^{-2}}$
r	cross-section radius of inner reactor	0.092	m
ρ	density of the reactor contents	1000	$\mathrm{gL^{-1}}$
C_P	heat capacity of reactor contents	$4.2 \cdot 10^{-3}$	$\mathrm{kJg^{-1}K^{-1}}$
$C_{\mathrm{B,in}}$	input concentration of reactant B	3	$\mathrm{molL^{-1}}$
V_J	content volume in cooling jacket	2.22	L
T_{in}	temperature of inflow	300	K
$C_{C,0}$	initial concentration of the product C	0	$\mathrm{molL^{-1}}$

Additionally, there are two parameters, k and H, which occurs in Equation 2.5b and 2.5d. These denotes the respective reaction constant and enthalpy of the chemical reaction and it is given that the values of these are not known precisely. In his paper, Sakthi Thangel originally investigates the cost effectiveness of implementing a multi-stage NMPC that estimates the values of these unknown parameters. The focus of this thesis is however the implementation of the EKF and MHE, and this uncertainty issue is chosen to remain out of scope. For this thesis, the parameters have been fixed at the specific initial values that are given in the article:

$$\mathbf{p}_0 = \begin{pmatrix} H \\ k \end{pmatrix} = \begin{pmatrix} -355\frac{\mathrm{kJ}}{\mathrm{mol}} \\ 1.205\frac{\mathrm{l}}{\mathrm{mol\ h}} \end{pmatrix} \tag{2.7}$$

2.4 Constraints and cost function

This section has been taken from the preparatory work.[17] For the semi-batch reactor there are a series of constraints, both physical and operational, that limit the production of C. The physical constraints of the states along with proposed initial conditions x_0 are summarised in Table 2.4.1

Table 2.4.1: Physical constraints of the states along with proposed initial conditions

State	**Intial Condition**	**Lower Bound**	**Upper Bound**	**Unit**
V_R	3.5	0	8	L
C_A	2	0	5	mol $\mathrm{L^{-1}}$
C_B	0	0	5	mol $\mathrm{L^{-1}}$
T_R	325	273	350	K
T_J	325	273	350	K

In addition to the physical constraints, there are operational constraints on the control inputs that are given to be;

Table 2.4.2: Upper and lower bounds of control inputs

Control	Initial	Lower Bound	Upper Bound	Unit
$\dot{V}_{\text{in}}$	0.0	0	32.4	L h^{-1}
$\dot{Q}_{\text{K}}$	0.0	-9000	0	kJ h^{-1}

Finally, the optimisation problem also have operational constraints on the volume and temperature in the reactor. These are considered soft constraints, and violations of these are not as critical compared to the physical as they are imposed from mostly an economic point of view. The operational constraints are;

Table 2.4.3: Operational constraints of the states

State	Lower Bound	Upper Bound	Unit
T_R	322	326	K
V_R	[-]	7	L

and the cost function of the optimisation problem is given as;

$$\min_{\mathbf{x}_k,\mathbf{u}_k,\epsilon_k} \sum_{k=1}^{N} -c_C V_R + 0.0154(\Delta\dot{V}_{in,k})^2 + 5.5\times10^{-5}(\Delta\dot{Q}_k)^2 + 10^6 {\epsilon_{k,[1]}}^2 + 10^{10}{\epsilon_{k,[2]}}^2 \quad (2.8)$$

where $\Delta\dot{V}_{\text{in,k}} = \dot{V}_{\text{in,k}} - \dot{V}_{\text{in,k-1}}$ and $\Delta\dot{Q}_k = \dot{Q}_k - \dot{Q}_{k-1}$ and represents the regularisation terms. Additionally, slack variables are introduced in the cost function as ϵ_1 and ϵ_2, which purpose is to provide flexibility to the optimiser. The slack variables are available for the optimiser, although the use of them are heavily penalised for cost function with their significant weighting. ϵ_1 and ϵ_2 therefore provide a slight increase or decrease to the T_R and V_R constraints which may make the difference for the optimiser to calculate an optimal solution.

In summary, the cost function of the optimisation problem given in Equation 2.8 subject to the aforementioned constraints can be summarised as;

$$x_{k+1} = f(x, u_k, d_k) \quad (2.9a)$$
$$322 \leq T_{R,k} + \epsilon_{k,[1]} \leq 326 \quad (2.9b)$$
$$V_{R,k} + \epsilon_{k,[2]} \leq 7 \quad (2.9c)$$
$$-1 \leq \epsilon_{k,[1]} \leq 1 \quad (2.9d)$$
$$-0.01 \leq \epsilon_{k,[2]} \leq 0.01 \quad (2.9e)$$
$$u_k^{low} \leq u_k \leq u_k^{high} \quad (2.9f)$$
$$(2.9g)$$

2.5 Presence of process and measurement noise

In reality, the semi-batch reactor will not follow its particular model exactly, as it will be prone to external disturbances. This could for instance be caused by temperature and pressure variations in the room which could affect the chemical reacitons that partake in the reactor. Another example could be a dysfunctional rotor in the reactor that would result in uneven mixing during operation. Process noise ω_k is used to describe the deviation, or uncertainty, that arises from the difference between ideal and real behaviour of the semi-batch reactor [18]. Unlike the unpredictable nature of process noise, measurement noise v_k is mainly caused by the equipment that is used to obtain measurements of the system. For instance, with electric signals, measurement noise is often caused by inference from other electric sources or caused by wear and tear on the sensors. [19]

In order to make the dynamic model more realistic, the uncertainties are to be simulated in an attempt to include the element of randomness by generating random values and adding them to the states. The noise is to follow a Gaussian distribution with an assumed mean $\mu = 0$:

$$\omega \sim N(0, {\sigma_\omega}^2) \tag{2.10a}$$

$$v \sim N(0, {\sigma_v}^2) \tag{2.10b}$$

along with the standard deviations σ_ω and σ_v

$$\sigma_\omega = [\sigma_{\omega,V_R} \sigma_{\omega,C_A} \sigma_{\omega,C_B} \sigma_{\omega,T_R} \sigma_{\omega,T_J}] = [0.01\ 0.05\ 0.05\ 0.5\ 0.5]$$

$$\sigma_v = [\sigma_{v,V_R} \sigma_{v,T_R} \sigma_{v,T_J}] = [0.2\ 1.0\ 1.0]$$

For the semi-batch reactor, every state has process noise associated with it, but this is not the case for the measurement noise. So far it has not mentioned that the C_A and C_B states are actually immeasurable, and therefore result in σ_v only being three dimensional. The cause of this is to be elaborated in later sections, but for now the focus remains towards simply presenting the process and measurement noise that are present in the semi-batch reactor. A Gaussian-distributed probability curve is given in Figure 2.5.1 in an attempt to further explain the behaviour the noise is to inherit;

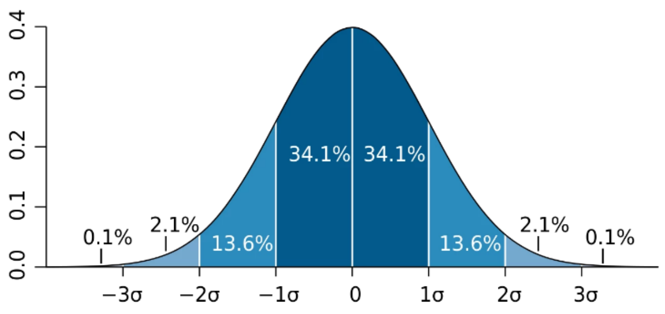


Figure 2.5.1: Example of a Gaussian distributed probability curve. [20]

and the variance of each state will determine the degree of uncertainty associated with their respective distributions. The greater the variance is, the wider the probability curve become, which in turn will result in a single most probable event becoming less apparent. The randomly generated noise of each state will remain within range of their respective variances $\pm 2\sigma_{\omega,v}$ and are to be added to the process and measured states as a term that represents the unpredictable deviation from ideal behaviour. A visual description of the significance of the mean and variance on the probability curve is given in Figure 2.5.2

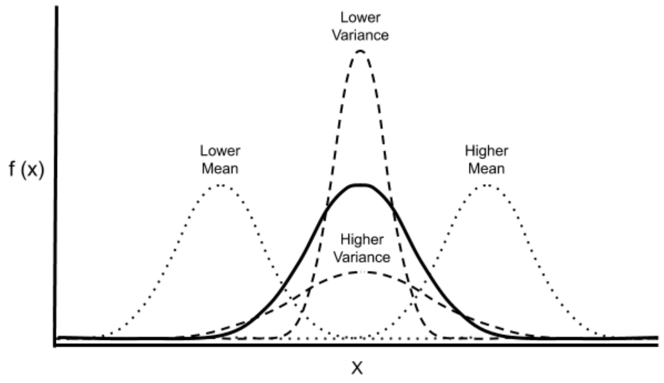


Figure 2.5.2: Visual description of mean and variance in a normal distribution. The greater the variance, the more varied, or uncertain, the process and measurement noise will be [21]

Throughout the operation of the semi-batch reactor, it is assumed that the standard deviations σ_{ω_k} and σ_{v_k} do not change significantly and therefore remain fixed at the given values. This assumption was considered to be reasonable as the semi-batch reactor will only operate for a single hour and not be exposed to external disturbances that would significantly affect the probability distribution. Wear and tear on the measurement devices could obviously affect the measurement noise distributions, but this aspect is considered to be negligible as the semi-batch reactor will only operate for 1 hour.

As there are 5 different states for the semi-batch reactor, the question remains whether the generated noise of these are correlated or not. For multiple correlated states, the concept of multivariate normal distributions is introduced and a visual example of this is given in Figure 2.5.3,

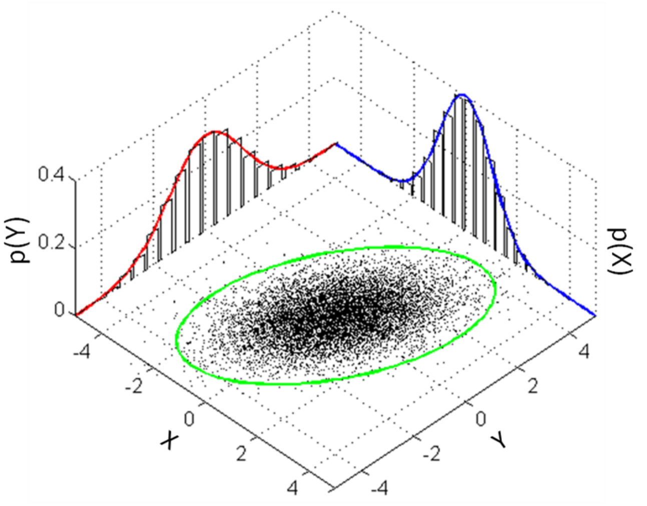


Figure 2.5.3: Visual example of multivariate normal distribution that is used to describe correlated values of random variables. Higher density of random variables, as seen within the green ellipse, indicates the peak of the Gaussian distribution. [22]

and the covariances of σ_{ω_k} and σ_{v_k} are summarised as the respective process and measurement noise-covariance matrices Q and R:

$$Q = \begin{bmatrix} {\sigma_{\omega,V_R}}^2 & 0 & 0 & 0 & 0 \\ 0 & {\sigma_{\omega,C_A}}^2 & 0 & 0 & 0 \\ 0 & 0 & {\sigma_{\omega,C_B}}^2 & 0 & 0 \\ 0 & 0 & 0 & {\sigma_{\omega,T_R}}^2 & 0 \\ 0 & 0 & 0 & 0 & {\sigma_{\omega,T_J}}^2 \end{bmatrix}$$

$$R = \begin{bmatrix} {\sigma_{v,V_R}}^2 & 0 & 0 \\ 0 & {\sigma_{v,T_R}}^2 & 0 \\ 0 & 0 & {\sigma_{v,T_J}}^2 \end{bmatrix}$$

For simplicity, it is assumed that the generated $\sigma_{\omega,v}$ are uncorrelated, and as a result Q and R are diagonal matrices where the diagonal consists of the respective variances. Since the standard deviations are assumed to remain constant throughout the optimisation problem, Q and R will as well. The weighting of these matrices will determine the shape of the Gaussian distributions and therefore the degree of uncertainty associated with the generated ω_k and v_k.

2.6 Previous work and thesis motive

This thesis is largely a continuation of preparatory work that was conducted the previous semester [17], and aims to improve the certain aspects that were introduced. The goal of the project work was to develop a nonlinear model predictive controller (NMPC) on the semi-batch reactor in order to familiarise the student with the theory and implementation of nonlinear optimisation. This was intended to be preparatory work for the upcoming master thesis, and originally the objective of this thesis was, as explained with the parameters k and H, to develop a multi-stage NMPC and investigate the cost effectiveness of it. However, the topic changed from this to be more focused towards addressing uncertainty of the reactor which was a topic that had, because of time constraint, largely remained outside the scope of the project work. This was to be done in an effort to make NMPC operation more realistic and provide results that hopefully could be used for real world applications or future research.

The main results, or the real state trajectories, from the preparatory work are given in Figure 2.6.1:

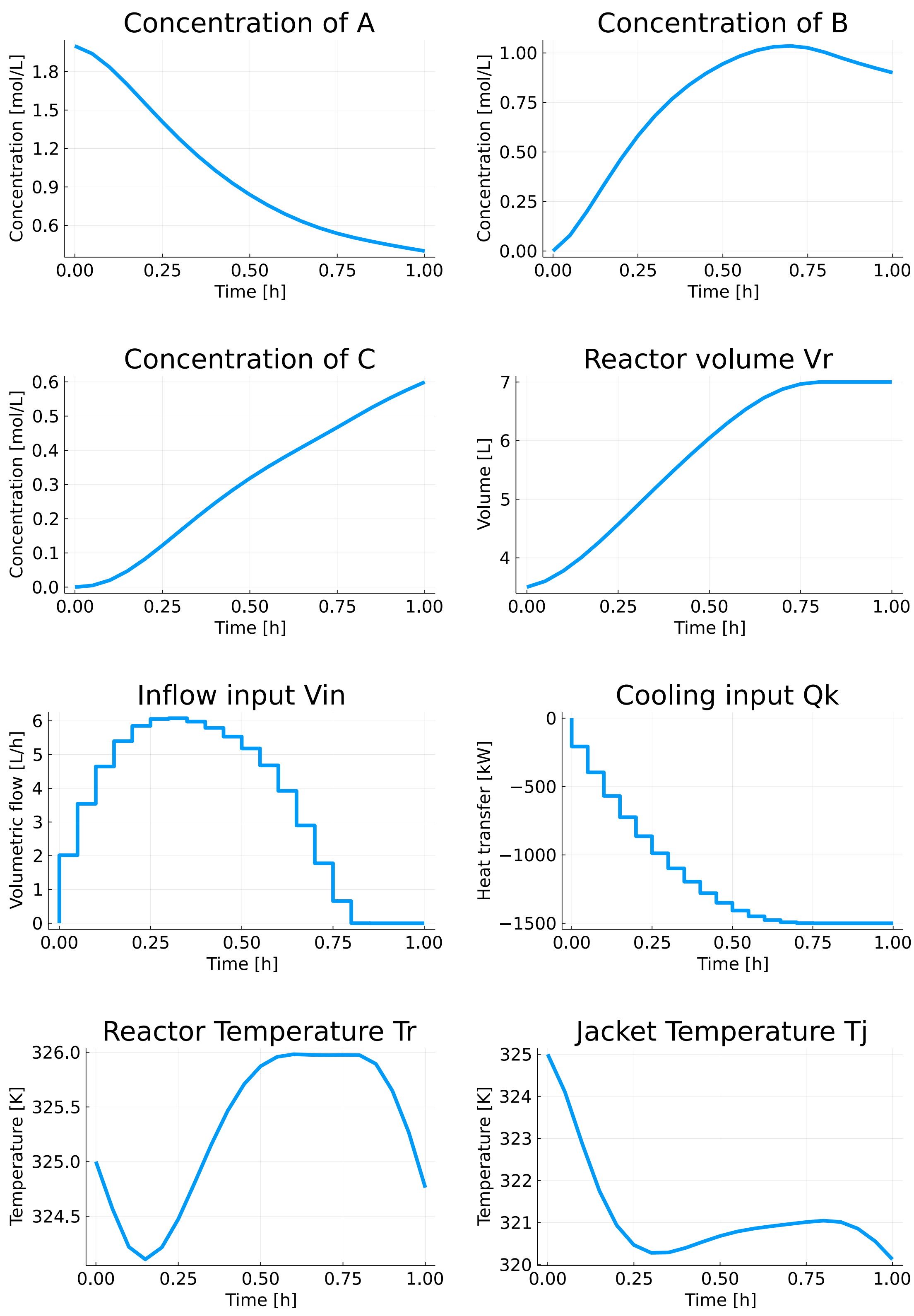


Figure 2.6.1: Real trajectory results from the preparatory work. The operational constraints for V_R and T_R are not violated and the production of C also remains strictly increasing over prediction horizon.

The main takeaway from these results is that the operational objective of maximising the production of C seem to be achieved. The production of C appears to be strictly increasing throughout the simulation while the constraints, most notably V_R and T_R, are not violated. Additionally, the control inputs profiles do not fluctuate, which indicate that the output of the NMPC remain stable throughout the simulation.

However, there are two significant issues associated with the optimisation problem that might question the legitimacy of the results. Firstly, it was assumed that the states of the semi-batch reactor were measured perfectly at every iteration. This means that knowledge of the system is exact and there are no uncertainties associated with it. This assumption results in a inaccurate model that disregards the presence of process and measurement noise which are unavoidable in the real world.

The second issue is that measurements that are related to a chemical quantity, for instance the concentration of specific components, are unrealistic to obtain. [23] This practice is generally considered expensive since it requires precise measurements of minuscule components, such as molecules of specific products. Because of mixing, the specific amount of the components that are consumed and produced becomes a complicated manner that ultimately result in a high computational burden for the measurement sensors which provide information to the NMPC. Because of this, the states are estimated, rather than attempt to measure them with sufficient accuracy. There are various methods that can achieve this, but for this thesis the EKF and MHE are chosen specifically for this purpose. Their respective theory and implementation are to be presented in the upcoming section.

The overall objectives of this thesis are summarised with Figure 2.6.2 and 2.6.3:

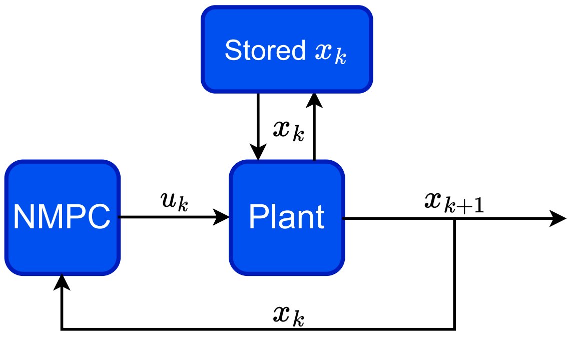


Figure 2.6.2: Flowchart of NMPC algorithm without state estimation. In the project work it was assumed that the states of the semi-batch reactor were measured perfectly, which is unrealistic for real world applications

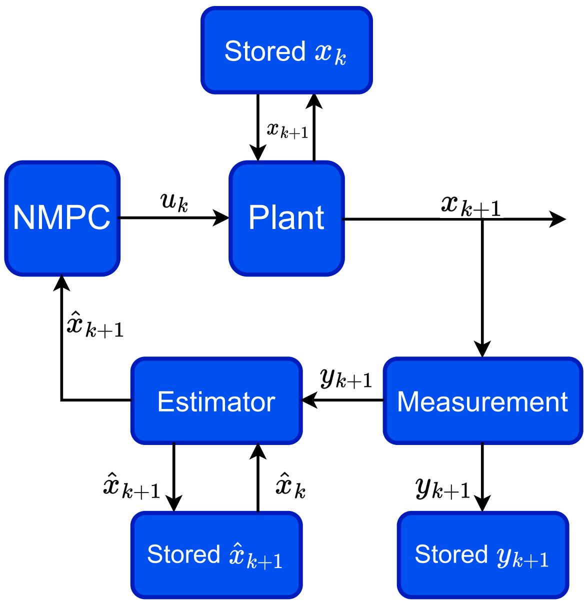


Figure 2.6.3: Flowchart of NMPC algorithm with state estimation. The NMPC is to be provided state estimates for operation that are obtained with noisy measurements.

This thesis therefore aims to improve the NMPC that was developed in the project work and implement state estimators that address uncertainty that stem from process and measurement noise along with estimating C_A and C_B that are immeasurable. Throughout these cycles the real, measured and estimated values are to be stored in global lists in order to plot them and visually interpret the performance of the NMPC using each of the estimators.

CHAPTER
THREE

METHODOLOGY

3.1 State Estimation

Before explaining each of the estimators, the overall objectives of state estimation and the common traits the mentioned techniques share should be presented. A flow diagram of state estimation and the components of it is given in Figure 3.1.1

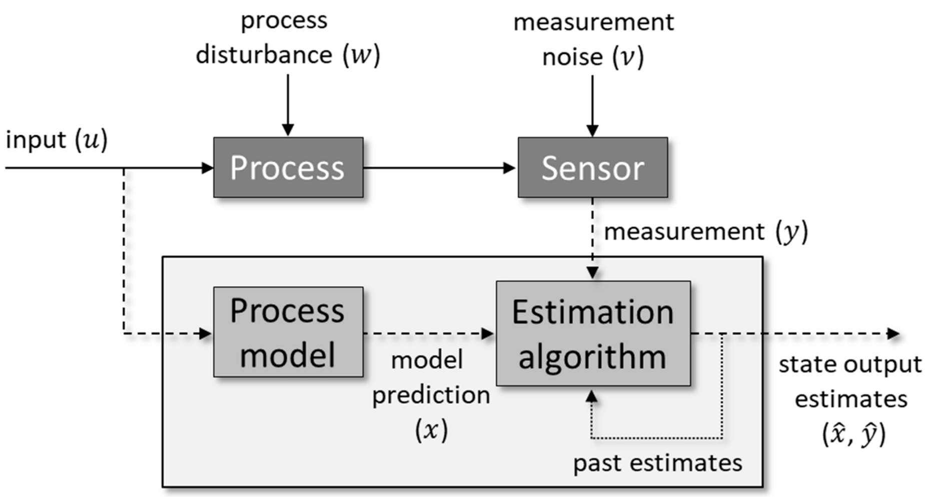


Figure 3.1.1: Flow diagram of state estimation and the components of it. The overall objective is to use observations, or noisy measurements y_k, along with a known input u_k to produce a filtered estimate $\hat{x}_k$ that optimises a given criteria. [24]

The physical system, a semi-batch reactor in this thesis, is driven by a series of inputs and its corresponding outputs that are measured by various devices, such as sensors. Because of this, the knowledge of the system behaviour is given by the known inputs u_k and observed outputs, y_k. However, these observations contain errors, or more specifically process, ω_k, and measurement noise ,v_k, that represent the difference between ideal and real behaviour. These errors therefore introduce an uncertainty to the process and thus makes knowledge about the states not exact. Information about the states is still required for the NMPC and thus the states have to be estimated. The EKF and MHE are to produce, albeit with different approaches, filtered estimates $\hat{x}_k$ based off model predictions, observed state measurements y_k, and past estimates $\hat{x}_{k-n}$, [25] that hopefully provides the NMPC with information of sufficient accuracy in order to solve the optimisation problem.

For the process model, consider a system described by the following discrete model:

$$x_{k+1} = f(x_k, u_k, \omega_k) \tag{3.1a}$$
$$y_k = h(x_k, v_k), \tag{3.1b}$$

where x_k is the system state vector, u_k the input vector, and $f(x_k, u_k)$ the system dynamics vector. The measurements y_k are expressed as a nonlinear function of the states and measurement noise and its respective jacobian with respect to x_k is:

$$C = \frac{\partial h}{\partial x} = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 \end{bmatrix},$$

as A and B are not measured. In the discrete model it seems that the system dynamics f and measurement function are dependent of ω_k and v_k. However, in this thesis it is assumed that these are additive and the discrete model is therefore formulated as:

$$x_{k+1} = f(x_k, u_k) + G\omega_k \tag{3.2a}$$
$$y_k = C(x_k) + H\omega_k + v_k, \tag{3.2b}$$

where G and H are the gain matrices that relates the process noise to the dynamic model and measurements. The latter is introduced as the process noise will affect the states, which in turn will affect measurements of it. It is however, for simplicity, common to assume that the measurements are independent of this regardless [26], which results in H becoming a zero-matrix:

$$H = \begin{bmatrix} 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \end{bmatrix}$$

In the upcoming simulations, the process and measurement noise are to be randomly generated within a Gaussian distribution with a mean equal zero. Because of this, ω_k and v_k are considered to be independent of the dynamic model and adidative, thus resulting in G being equal to:

$$G = I = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 \end{bmatrix}$$

With these assumptions, the discrete model can be written as:

$$x_{k+1} = f(x_k, u_k) + \omega_k \tag{3.3a}$$
$$y_k = C(x_k) + v_k, \tag{3.3b}$$

As described earlier, the ODEs of the semi-batch reactor in itself do not provide useful information to the optimisation problem and have to be integrated. Orthogonal collocation, which is described in Appendix 6.1, is used to numerically integrate the states for the NMPC:

$$x_{k+1} = F(x_k, u_k) + \omega_k \tag{3.4a}$$
$$y_k = C(x_k) + v_k, \tag{3.4b}$$

The equations that are to be used for the estimators depend on whether the dynamic model is continuous or discrete. In this thesis, a discrete model is to be used and the continuous model of the semi-batch reactor therefore has to be discreditized. This is done using the forward Euler method:

$$x_{k+1} = \underbrace{F(x_k, u_k) = x_k + dtf(x_k, u_k)}_{\text{forward Euler method}}, \tag{3.5}$$

where dt is the step length of the discredization. With the discrete model explained, it is time to present the other matrices that are to be used in both methods. Certain steps of each methods revolves around linearising the filtered estimate $\hat{x}_k$ in order to update different parameters and in order to achieve this the transition matrix A_k, which represents a linearised $5x5$ matrix model, is to be used:

$$A_k = \frac{\partial F}{\partial x_k}|_{u_k, \hat{x}_k, 0} \tag{3.6a}$$
$$= \frac{\partial x_k}{\partial x_k} + \frac{\partial (dtf(x_k, u_k))}{\partial x_k} \tag{3.6b}$$
$$= I + dt\frac{\partial f}{\partial x_k}(x_k, u_k), \tag{3.6c}$$

where the final term is:

$$\frac{\partial f}{\partial x_k}(x_k, u_k) =$$
$$\begin{bmatrix} 0 & 0 & 0 & 0 & 0 \\ \frac{V_{in}}{{V_R}^2}C_A & -\frac{V_{in}}{V_R} - kC_B & -kC_A & 0 & 0 \\ -\frac{V_{in}}{{V_R}^2}(C_{Bin} - C_B) & -kC_B & -\frac{V_{in}}{V_R} - kC_A & 0 & 0 \\ \frac{V_{in}}{{V_R}^2}(T_{in} - T_R) + \frac{\alpha A_W(T_R - T_J)}{\rho {V_R}^2 C_P} & -\frac{kC_B H}{\rho C_P} & \frac{kC_A H}{\rho C_P} & -\frac{V_{in}}{V_R} - \frac{\alpha A_W}{\rho V_R C_P} & \frac{\alpha A_W}{\rho V_J C_P} \\ 0 & 0 & 0 & \frac{\alpha A_W}{\rho V_J C_P} & -\frac{\alpha A_W}{\rho V_J C_P} \end{bmatrix}$$

In addition to A_k, the covariance matrices Q and R are also required to update the algorithm. These are to be used in the cost function of MHE and the calculation of updated covariance matrices Π_k. The latter will be a $5x5$ matrix that describes the covariance of the estimation errors.

3.2 Extended Kalman Filter

In essence, the main objective of the Kalman Filter (KF) is to produce optimal estimates based off noisy measurements of the states. The KF revolves around performing consecutive cycles of prediction and filtering with dynamics that are calculated based off Gaussian probability functions (PDFs).

The greatest limitation of the KF is its strict use of linear models, something which the EKF aims to adress. When either the system dynamics or observations are nonlinear, the PDFs that provide the minimum mean square estimate errors becomes non-Gaussian. One way to solve this is to propagate the non-Gaussian functions and evaluate their respective mean, something which introduces a high computational burden and is therefore deemed not optimal [25]. The most common approach to address nonlinear dynamics is to rather linearize the filter dynamics around previous filtered estimates, something which has been performed in this thesis with the help of the previously mentioned transition matrix A_k.

Producing the optimal estimates revolves around minimising the state estimation errors that are derived from the difference between real and estimated states. From a Bayesian viewpoint, the EKF propagates the PDFs of desired quantities (optimisation problem) based off knowledge of actual data, or measurements y_k. With a known input u_k and measurement y_{k+1} at iteration k let

$$p(x_k|y_k, u_{k-1}) \sim \mathcal{N}(\hat{x}(k|k), \Pi(k|k)), \tag{3.8}$$

represent the conditional PDF as a Gaussian PDF where $\hat{x}(k|k)$ denotes the state estimate and $\Pi(k|k)$ the covariance matrix of estimation errors that quantifies the uncertainty of the state estimate. The EKF therefore aims to obtain the "best", or optimal estimate, that optimises a chosen criteria, or in this thesis, the semi-batch reactor optimisation problem.

There are mainly two steps that make up the EKF method: an a priori guess, and an a posteriori correction of the guess. A flow diagram of this cycle is given Figure 3.2.1

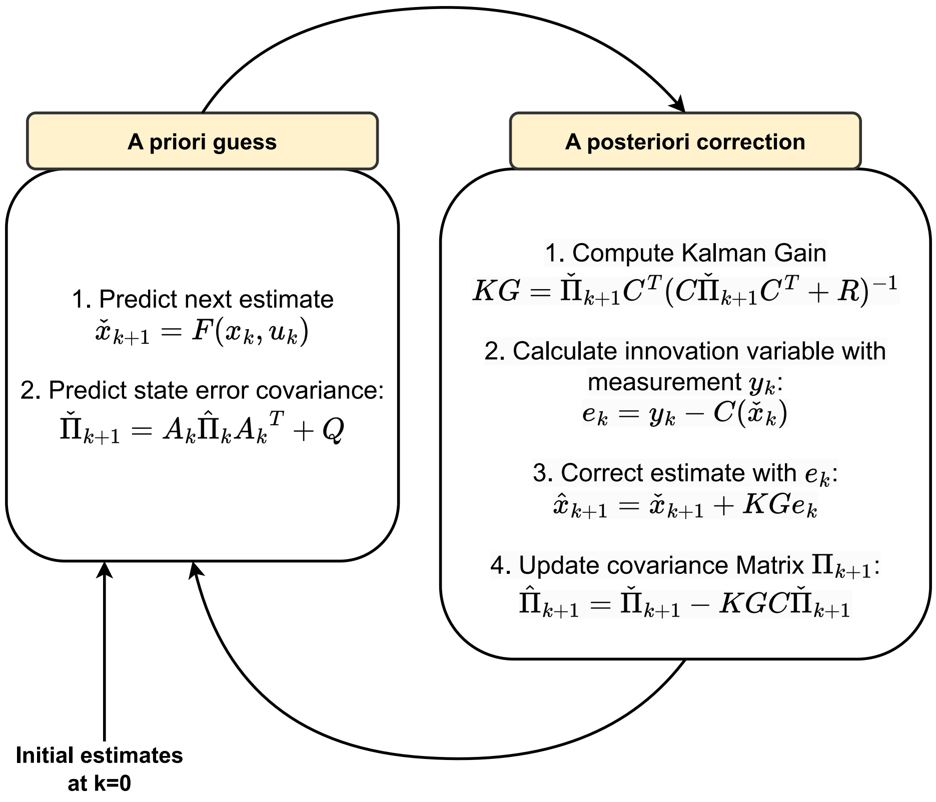

A priori guess
1. Predict next estimate
$\check{x}_{k+1} = F(x_k, u_k)$
2. Predict state error covariance:
$\check{\Pi}_{k+1} = A_k \hat{\Pi}_k {A_k}^T + Q$
A posteriori correction
1. Compute Kalman Gain
$KG = \check{\Pi}_{k+1} C^T (C \check{\Pi}_{k+1} C^T + R)^{-1}$
2. Calculate innovation variable with measurement y_k:
$e_k = y_k - C(\check{x}_k)$
3. Correct estimate with e_k:
$\hat{x}_{k+1} = \check{x}_{k+1} + KG e_k$
4. Update covariance Matrix Π_{k+1}:
$\hat{\Pi}_{k+1} = \check{\Pi}_{k+1} - KGC\check{\Pi}_{k+1}$
Initial estimates at k=0

Figure 3.2.1: Illustration of the EKF cycle

First, the a priori guess, or prediction, of the optimal state estimate $\check{x}_{k+1}$ is made:

$$\check{x}_{k+1} = F(x_k, u_k) \tag{3.9a}$$
$$\check{\Pi}_{k+1} = A_k \hat{\Pi}_k {A_k}^T + Q, \tag{3.9b}$$

where A_k is the transition matrix, $F(x_k, u_k)$ the integrated discrete model, and $\hat{\Pi}_k$ and Q the respective error and process noise covariance matrices. This prediction is then corrected with the Kalman Gain (KG) which uses the predicted error estimate covariance $\check{\Pi}_{k+1}$

$$KG = \check{\Pi}_{k+1} C^T (C \check{\Pi}_{k+1} C^T + R)^{-1} \tag{3.10}$$

The main purpose of KG is to relate the new information obtained from the current measurement y_k to the predicted estimate $\check{x}_{k+1}$. This gain is then multiplied by the innovation variable e_k which represent the difference between the measurement vector y_k, and predicted state $\check{x}_k$:

$$e_k = y_k - C(\check{x}_k). \tag{3.11}$$

The corrected a posteriori state estimate $\hat{x}_{k+1}$ and its corresponding covariance matrix $\hat{\Pi}_{k+1}$ are then:

$$\hat{x}_{k+1} = \check{x}_{k+1} + KGe_k \tag{3.12a}$$
$$\hat{\Pi}_{k+1} = \check{\Pi}_{k+1} - KGC\check{\Pi}_{k+1} \tag{3.12b}$$

For iteration k, the cycle of prediction and correction is then completed and the output is the filtered state estimate $\hat{x}_{k+1}$. It is these estimates that are to be stored and later compared to the real states in order to evaluate the accuracy and performance of the state estimators. With the cycle completed, as indicated in Figure 3.2.1, the iteration shifts and the values of $\hat{x}_k$ and $\hat{\Pi}_k$ are updated.

However, as every iteration requires information from the previous one, a question of how the algorithm is initialised arises. For the first iteration, there is no prior knowledge of $F(x_k, u_k)$, and thus an estimation has to be made. For the initial prediction, it is assumed that the initial state and error estimation covariance matrix of the semi-batch reactor are:

$$\check{x}_{0+1} \approx x_0 + \omega_0 \tag{3.13a}$$
$$\check{\Pi}_{0+1} \approx Q \tag{3.13b}$$

These are only approximations and their accuracy are debatable, but it stems from the fact that the given initial real state x_0 is known. Because of the certain x_0, the state uncertainty quantified as $\check{\Pi}_1$ will be relatively small. The initial guess of $\check{\Pi}_1$ is therefore assumed to be approximately equal to Q, as the uncertainty associated with the process noise will be more dominant than state estimation errors for the exception $k = 0$. With the a priori guesses $\check{x}_1$ and $\check{\Pi}_1$ estimated, the EKF can proceed as mentioned above.

3.3 Moving Horizon Estimator

The Moving Horizon Estimator (MHE) is a state estimation technique where the current state is estimated based off an optimisation problem that incorporates past measurements and states. Unlike the EKF, the MHE is capable of addressing non-linear dynamics and do not resort to linearising the filtered estimates. Based off various literature [7], there is a general consensus that MHE often outperform the EKF in terms of state estimation and robustness towards poor initial guesses and tuning parameters. This improved performance do however come at the expense of an increased computational burden. This is because the MHE solves an optimisation problem based off a sequence of past measurements, unlike EKF which only use the most recent.

Given the system model, the MHE aims to solve the following optimisation problem to estimate the states at time T with estimation window length N:

$$\min_{w,v,x} \sum_{k=T-N}^{T-1} {w_k}^T {Q_k}^{-1} w_k + \sum_{k=T-N}^{T-1} {v_k}^T {R_k}^{-1} v_k + \Gamma_{T-N}(x_T - N), \tag{3.14a}$$

$$\text{s.t.} \quad x_{k+1} = F(u_k, x_k) + \omega_k \tag{3.14b}$$

$$y_k = C(x_k) + v_k \tag{3.14c}$$

$$x_k \in \mathbb{X} \tag{3.14d}$$

$$w_k \in \mathbb{W}_k, \tag{3.14e}$$

$$\tag{3.14f}$$

or in other words, produce an estimate $\hat{x}_k$ with minimal uncertainty that stem from ω_k, v_k. Q and R remains the tuning matrices of the noise uncertainties and are equal to the ones previously presented for the EKF. The MHE requires prior knowledge of the system and this is provided to it with a series of past measurements. An issue with this approach is the question of how many prior states the MHE should account for. On one hand, the MHE could include all prior measurements in order to output the most optimal estimates that follow prior behaviour of the system. However, the issue with this is that the computational burden would only accumulate with the increasing number of previous measurements to the point where the MHE would not be able to produce estimates within a reasonable time frame. Because of this, a selection of which measurements to account for in the MHE is done with the sliding window concept which is illustrated in Figure 3.3.1:

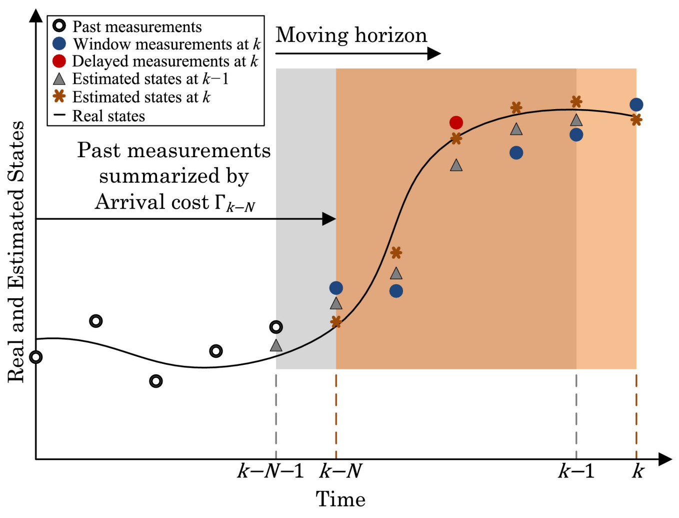

Past measurements
Window measurements at k
Delayed measurements at k
Estimated states at $k-1$
Estimated states at k
Real states
Moving horizon
Past measurements summarized by Arrival cost Γ_{k-N}
Real and Estimated States
$k-N-1$
$k-N$
$k-1$
k
Time

Figure 3.3.1: Overview of MHE. Which of the past measurements that are used to estimate the current state are determined by a shifting window. A terminal cost Γ_{k-N} is added to the cost function to account for previous measurements that are not included. [27]

The arrival cost, $\Gamma_{T-N}(x_T - N)$, acts as a corrective term that attempts to include prior information of the system states outside the sliding window with length N. The ideal arrival cost can be calculated by solving the optimisation problem:

$$\begin{aligned}
\Gamma_{k+1}(x_{k+1}) = \min_{w_k, v_k, x_k} \quad & {w_k}^T {Q_k}^{-1} w_k + {v_k}^T {R_k}^{-1} v_k + \Gamma_k(x_k) && (3.15a)\\
\text{s.t.} \quad & x_{k+1} = F(u_k, x_k, w_k) && (3.15b)\\
& y_k = C(x_k) + v_k && (3.15c)\\
& x_k \in \mathbb{X} && (3.15d)\\
& w_k \in \mathbb{W}_k && (3.15e)\\
& && (3.15f)
\end{aligned}$$

Unfortunately, this method is not particularly useful as it introduces another optimisation problem which will significantly increase the computational burden. In order to adress this, the arrival cost is rather estimated as the following quadratic expression:

$$\Gamma_k(x_k) = (\check{x}_k - x_k)^T {\Pi_k}^{-1} (\check{x}_k - x_k), \tag{3.16}$$

Once again, uncertainty is introduced in the optimisation problem in the form of Π_k that denotes the covariance matrix of the estimation errors. x_k are the state variables that are to be optimised in the MHE cost function and $\check{x}$ denote previously estimated states of the MHE. At iteration k, $\check{x}_k$ represents the most recent estimate outside the estimation window. If Equation 3.16 is expanded, the presented arrival cost can be simplified as:

$$\begin{aligned}
(\check{x}_k - x_k)^T {\Pi_k}^{-1} (\check{x}_k - x_k) &= {x_k}^T \Pi^{-1} x_k - 2\check{x}\Pi^{-1} x + \check{x}^T \Pi^{-1} \check{x}\\
&= {x_k}^T \Pi^{-1} x_k - 2\check{x}\Pi^{-1} x,
\end{aligned}$$

as the latter term do not contain the x_k variable and will therefore be insignificant for the optimisation problem.

As most of components in MHE depend on calculations from previous iterations, the question remains of how the algorithm is initialised. For the initialisation, the optimisation problem given in Equation 3.16 is solved, but this time it will rather be:

$$\begin{aligned}\min_{w,v,x} \sum_{k=T-N}^{T-1} {w_k}^T {Q_k}^{-1} w_k + \sum_{k=T-N}^{T-1} {v_k}^T {R_k}^{-1} v_k + \Gamma_{T-N}(x_T - N) \\ = \min_{v,x} v_0 {R_0}^{-1} v_0 + \Gamma_0(x_0),\end{aligned}$$

where the initial arrival cost estimate is equal to:

$$\Gamma_0 = (\check{x}_0 - x_k)^T {\Pi_0}^{-1} (\check{x}_0 - x_k) \tag{3.19}$$

At the first iteration, $k = 0$, it is assumed that the simulation has not developed significant process noise yet, therefore making the process noise term of the optimisation problem equal to zero. For $k = 0$, $\check{x}$ remains unknown, as no previous filtered state estimate has been produced yet. To address this, $\check{x}_k$ is estimated to be equal to the known initial state x_0 which has been presented earlier in this report. The initial iteration of MHE therefore only depend on the measured state of the system and attempts to solve the optimisation problem based off it. After this initialisation, the MHE will proceed as described earlier until the simulation is complete. The estimation window will have a maximum length of $T = 10$ so for iterations $k \leq 10$, there will be no excluded $\check{x}$ that represent prior knowledge. To address this, it is assumed that, until the estimation window fills up, $\check{x} = x_0 + \omega$.

As the covariance error estimation matrix, Π_0, requires prior information as well, it has to be estimated for the initialisation. Similarly to the EKF, it is assumed for the initialisation that

$$\Pi_0 = Q, \tag{3.20}$$

as the uncertainty associated with the process noise will be more significant than state estimations since x_0 is known. With the the concept of the arrival cost explained, the overall procedure of the MHE can be presented. A flowchart of an iteration in MHE is given in Figure 3.3.2

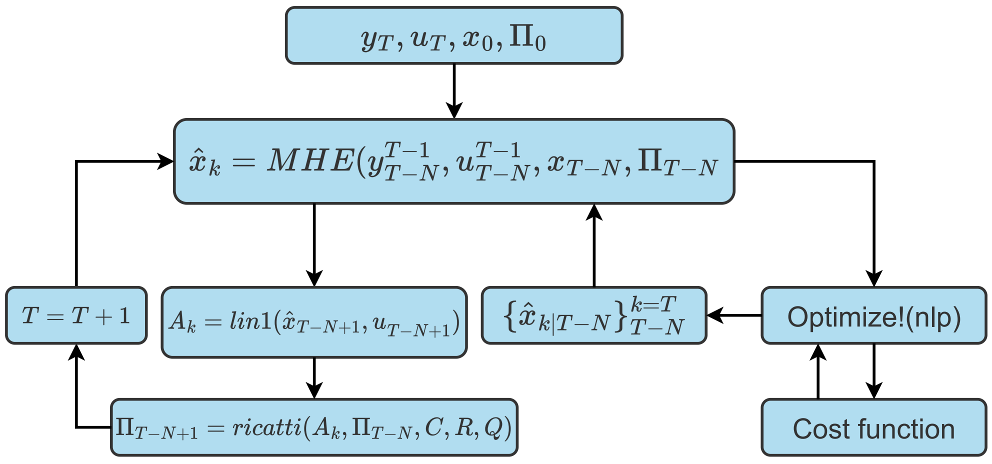

y_T, u_T, x_0, Π_0
$\hat{x}_k = MHE(y_{T-N}^{T-1}, u_{T-N}^{T-1}, x_{T-N}, \Pi_{T-N}$
$T = T+1$
$A_k = lin1(\hat{x}_{T-N+1}, u_{T-N+1})$
$\{\hat{x}_{k|T-N}\}_{T-N}^{k=T}$
Optimize!(nlp)
$\Pi_{T-N+1} = ricatti(A_k, \Pi_{T-N}, C, R, Q)$
Cost function

Figure 3.3.2: Illustration of the MHE cycle

First, as previously described, the algorithm is initialised with the initial inputs u_0, states x_0 and covariance matrix Π_0. With the cost function, previous noisy measurements y_k^T and inputs u_k^T, an optimiser produces an estimate, $\hat{x}_k$ that fulfils the constraints imposed on the system. It is important to note that, unlike the NMPC, the constraints incorporated in the MHE are physical:

$$x_{k+1} = f(x, u_k, d_k) \leq 0 \tag{3.21a}$$
$$V_{R,k} \leq 8L, \tag{3.21b}$$
$$\tag{3.21c}$$

rather than operational. This is done in order to ensure that the state estimations from the MHE remains positive as negative states would not make physical sense. The optimal estimate $\hat{x}_k$ is then stored and the algorithm can shift to the next iteration. However, similarly to the EKF, the parameters, or $\check{x}_k$ and Π_k, have to be updated for the next iteration. The covariance error estimate matrix Π_k is updated using the Ricatti equation:

$$\Pi_{k+1} = A_k(\Pi_k - \Pi_k C^T(C\Pi_k C^T + R)^{-1}C\Pi_k){A_k}^T + GQG^T, \tag{3.22}$$

while $\check{x}_k$ is automatically updated as the estimation window progresses and therefore do not require additional computation to be updated. A_k represents the transition matrix that linearises the dynamic model around the filtered estimate $\hat{x}_k$ and is equal to the one previously used for EKF. With Π_k updated, the estimated, real and measured states of the iteration are stored before the algorithm shifts to the next iteration with $T = T + 1$. This cycle of measuring, estimating and updating of $\check{x}_k$ and Π_k continues until the semi-batch reactor operation is complete. The final results will be a series of measured,real and estimates states that can be plotted in order to visually interpret the performance of the MHE and compare it to the output from the EKF.

CHAPTER
FOUR

RESULTS AND DISCUSSION

4.1 Comparative analysis on overall performance

In this chapter, the output of the optimisation problem and the overall performance of the state estimators are to be presented and discussed. There are three main criteria that are to be used when evaluating the results;

- **State trajectories that appear to be physically realistic**
- **Similarity to the real trajectories from the project work**
- **Production of the desired product C**

The first criteria may be considered to be vague, but the purpose of it to check if the results actually provide useful information. For instance, if the volume V_R at certain iterations is negative, the results would not be useful as negative state values do not make physical sense. This type of error naturally stem from implementation errors in the code framework, but it is regardless important to be critical of the physical behaviour of the state trajectories. For instance, if some of the operational constraints are consistently violated, this could indicate code errors or that there are no feasible solutions.

Ideally, the results, or state trajectories, should appear similar to the results from the project work that were given in Figure 2.6.1. As described earlier, the main difference between the previous project work and this thesis is the implementation of state estimation for the former in the optimisation problem. The NMPC from the project work use real, and not estimated, states and therefore output the real state trajectories. These results therefore represent the real behaviour of the semi-batch reactor and are to be used as metrics to investigate if the state estimators are capable of providing estimates to the NMPC with reasonable accuracy.

Finally, and probably the most important criteria, is how much C is produced. This will indicate which of the estimators are able to provide estimates to the NMPC that results in maximisation of its operational goal.

Since the process and measurement noise are to be random, it is important that these are generated in a consistent manner as this is necessary in order to produce comparative results. Additionally, it is also important to note that the randomness exhibited for each simulation will produce varying results. To achieve representative results despite the element of randomness, the upcoming state trajectories are to represent the mean of 100 seed states, or simulations. Each of the simulation are to follow their respective seed state, or in other words, the first simulation will follow seed state 1, the second seed will follow seed state 2 and so on up to seed state 100. This will ensure consistency of the generated noise as the simulations will follow a pre-defined sequence of randomness which will be unique for each seed state. The mean state and C trajectories obtained with the EKF from seed $1 - 100$ are given in Figure 4.1.1:

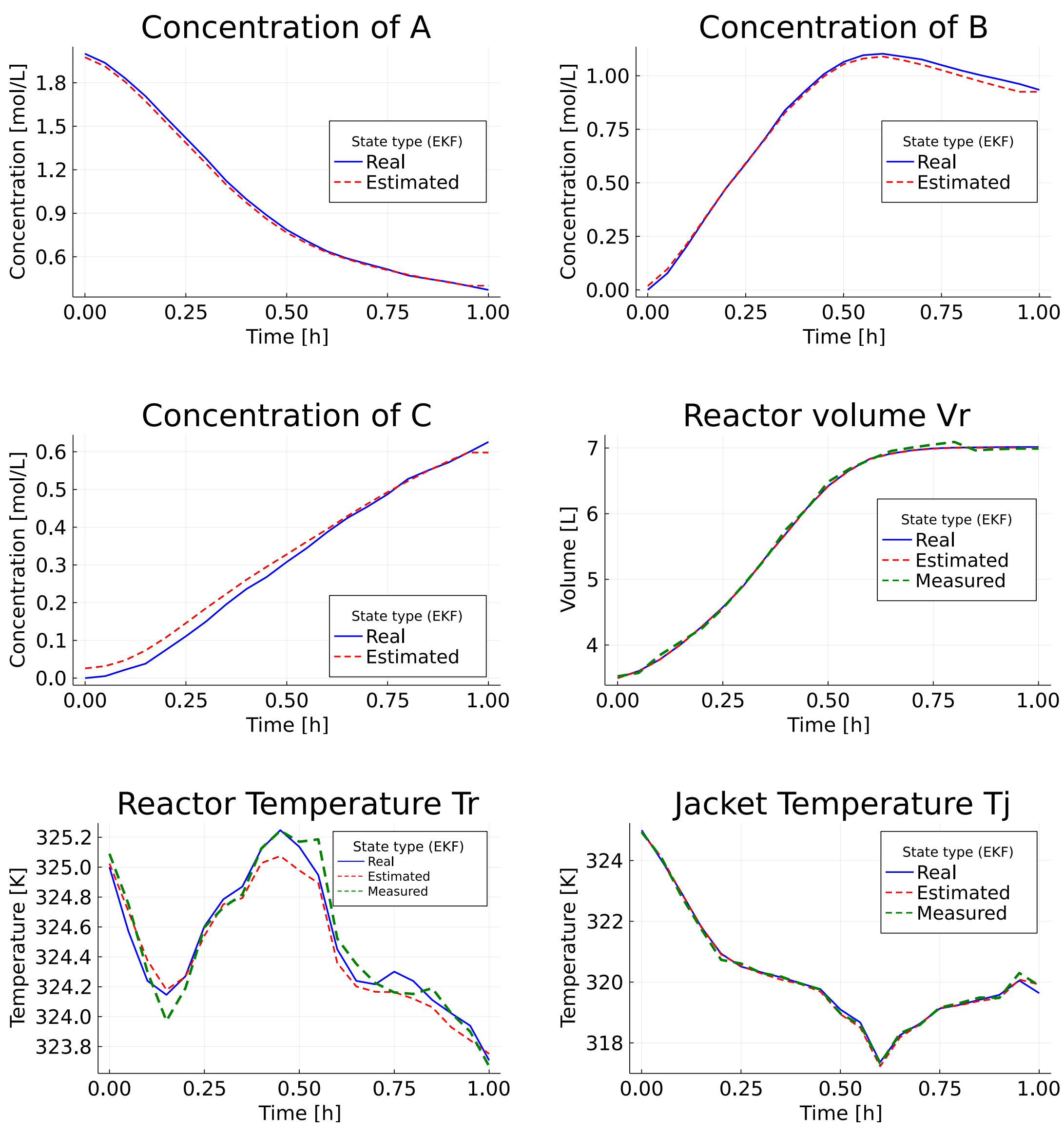


Figure 4.1.1: Main results of the optimisation problem using EKF

Based of these results, it seems that the EKF is functional as it appears to be capable of providing estimates to the NMPC that results in maximisation of C. The trajectory of C_C strictly increases throughout the simulation which indicate that the operational goal was achieved with the EKF. Compared to the results of the project, the estimated states follow approximately the same trajectories of the real states and overall these results confirms the expected performance of the EKF. With the same settings, the optimisation problem was simulated once again with the MHE instead and the results are given in Figure 4.1.2:

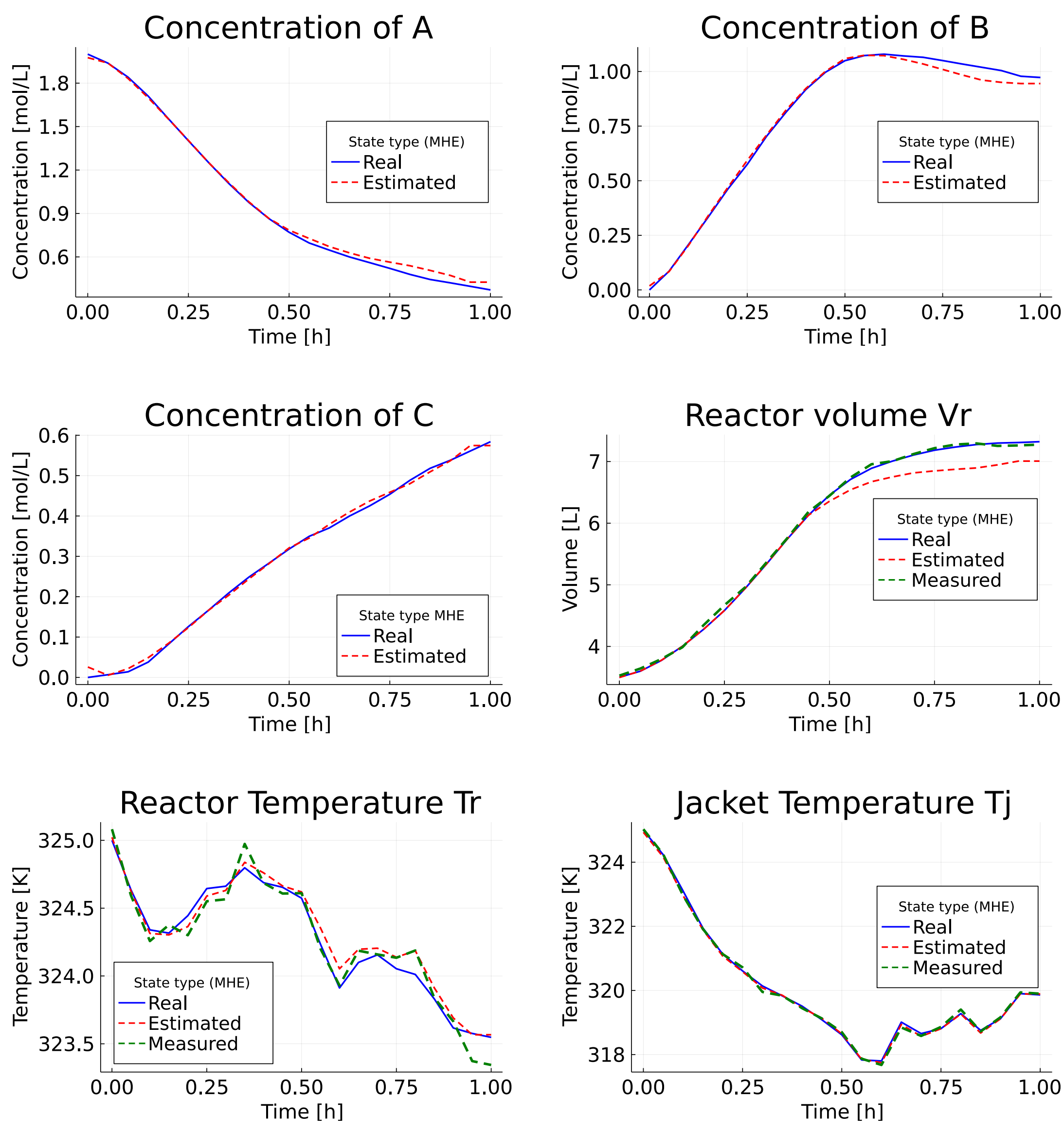


Figure 4.1.2: Main results of optimisation problem using MHE

Similarly to the EKF, the MHE overall display expected behaviour, albeit with minor differences. For instance, the estimated trajectory of T_R do in general follow the real trajectory more closely than EKF. Another difference is the slight deviation at the end of the simulation for the C_B trajectory, although it is debatable how significant this is. The most interesting difference between the two estimators are their respective real V_R trajectories and which constraints they operate from. It appears that the real V_R trajectories of EKF and MHE operate based off the operational $V_R \leq 7L$ and physical $V_R \leq 8L$ constraints respectively. The estimated trajectory of both do however satisfy the former, but this difference is worth noting regardless. Regarding the operation goal of maximising the production of C, it seem that both methods achieve relatively similar output. There is however a minor difference, as it appears that at the final iteration, $k = 20$, the EKF has resulted in a slightly, or ≈ 0.05 [mol/L], higher concentration of C than MHE.

In summary, both estimators provide estimates to the NMPC that results in expected behaviour and fulfilment of the operational goal of maximising C. The state trajectories for each method do however display a difference despite being exposed to the same sequence of generated noise. This is expected as the estimators will provide different state estimates to the NMPC which will calculate control actions, which in turn results in different state trajectories.

Although the aforementioned results appear straightforward, there are however other aspects of the optimisation problem that should be evaluated as well. The previously presented trajectories represent the mean of 100 simulations and it would be interesting to investigate the associated uncertainties of each method. With the mean of the real state trajectories, the standard deviation at each iteration were calculated and the corresponding uncertainties are illustrated in Figure 4.1.3:

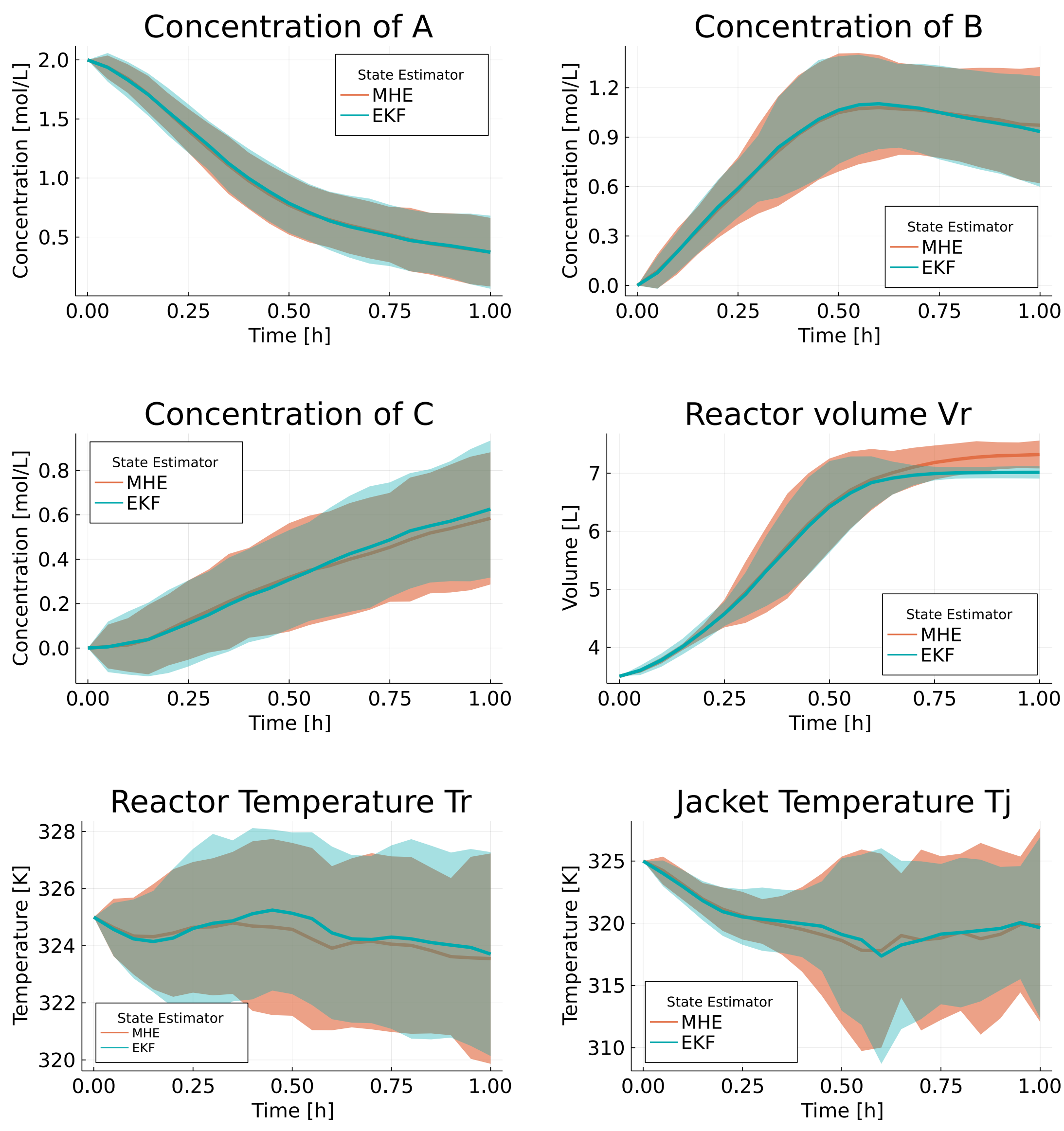


Figure 4.1.3: Comparative analysis of uncertainty with the mean of the real trajectories of EKF and MHE. The shaded area represents the mean ±2 times the standard deviation at each iteration

It appears that the system states are quite prone to uncertainty regardless of the choice of estimator. This is especially the case for the temperature related states T_R and T_J, with the latter having a variance up to $\approx 16K$ for the EKF. These variances are most likely caused by the irregular and excessive use of the cooling input $\dot{Q}_k$, which directly affects the state trajectories for each seed, and thus the mean overall as well. This topic is however to be discussed in greater detail in Section 4.2.6. The most interesting result is probably the uncertainties associated with the V_R trajectories, as there is a significant difference between the EKF and MHE at the end of the simulation. The exact reason for this may be difficult to pinpoint, but a possible explanation could be the estimated arrival cost Γ_k which the MHE uses.

4.2 Case studies

So far the performance of each estimator have yielded rather similar results and it may be difficult to recommend one method over the other. There are a series of factors that could affect these results and it may be interesting to investigate if changing these would result in a more significant difference in performance of the estimators. Because of that, a series of case studies have been conducted and these are summarised in Table 4.2.1:

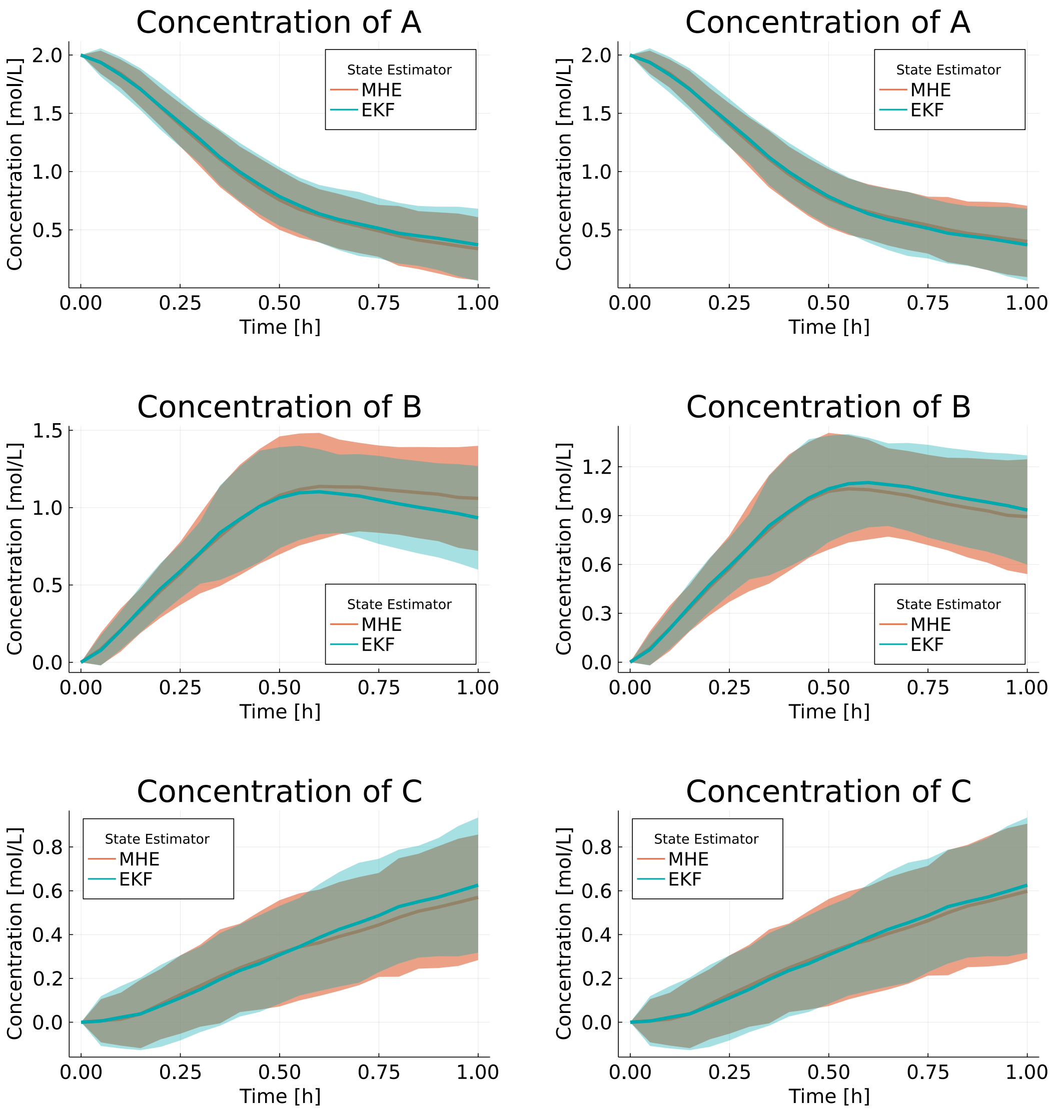
Table 4.2.1: Overview of Case studies

Variable	**Case**	**Value**
Length of estimation window (MHE)	1,2	$N = 5, 20$
Length of prediction horizon (NMPC)	3,4	$NFE = 5, 15$
Seed amount	5,6	$10, 1000$
Initial Guess	7,8	$x_{0,2}, x_{0,3}$
Standard deviation of noise	9,10	$\sigma_{\omega,2}, \sigma_{v,2}, \sigma_{\omega,3}\sigma_{v,3}$
Regularisation terms	11,12	$\Delta\dot{V}_{\text{in},2}, \Delta\dot{Q}_2, \Delta\dot{V}_{\text{in},3}, \Delta\dot{Q}_3$

The generated results in the previous section are considered as the "Base case" and the upcoming case studies are therefore to be compared to it in order to determine whether the change of variables result in a significant change on the output of the optimisation problem.

4.2.1 Significance of length of estimation window

So far the length of the estimation window of the MHE has had a fixed length of $N = 10$. The comparative results given in Figure 4.1.3 indicated significant uncertainty associated with the MHE and it may be interesting to investigate if increasing this to $N = 20$ would improve the performance. It could also be interesting if shortening the prediction could yield approximately the same performance, and therefore prove to be a more cost-effective choice of N. The results of Case 1 and Case 2 with their respective lengths $N = 5$ and $N = 20$ were conducted and the results are given in Figure 4.2.1.


Concentration of A
Concentration of A
Concentration of B
Concentration of B
Concentration of C
Concentration of C
Concentration [mol/L]
Time [h]
State Estimator
MHE
EKF

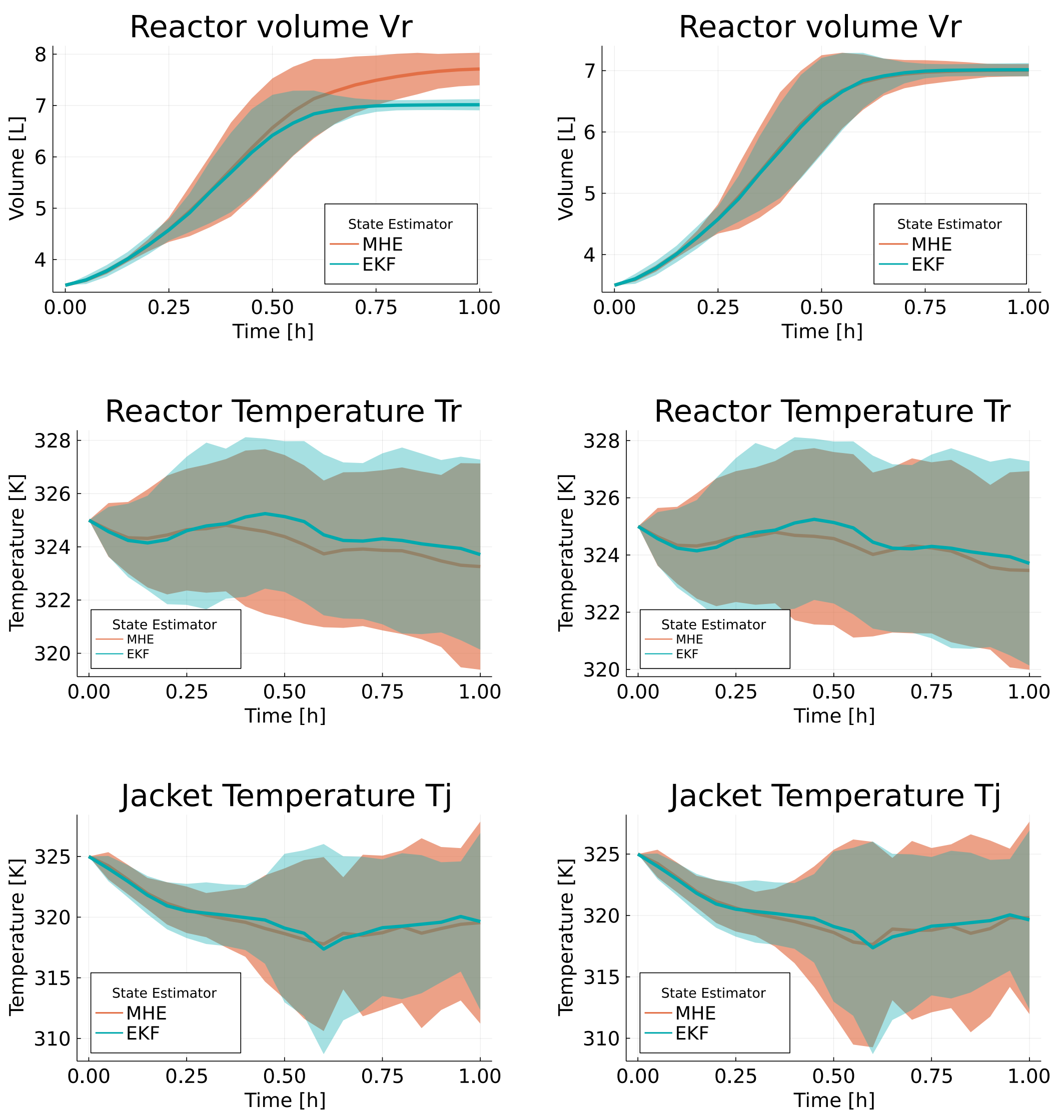
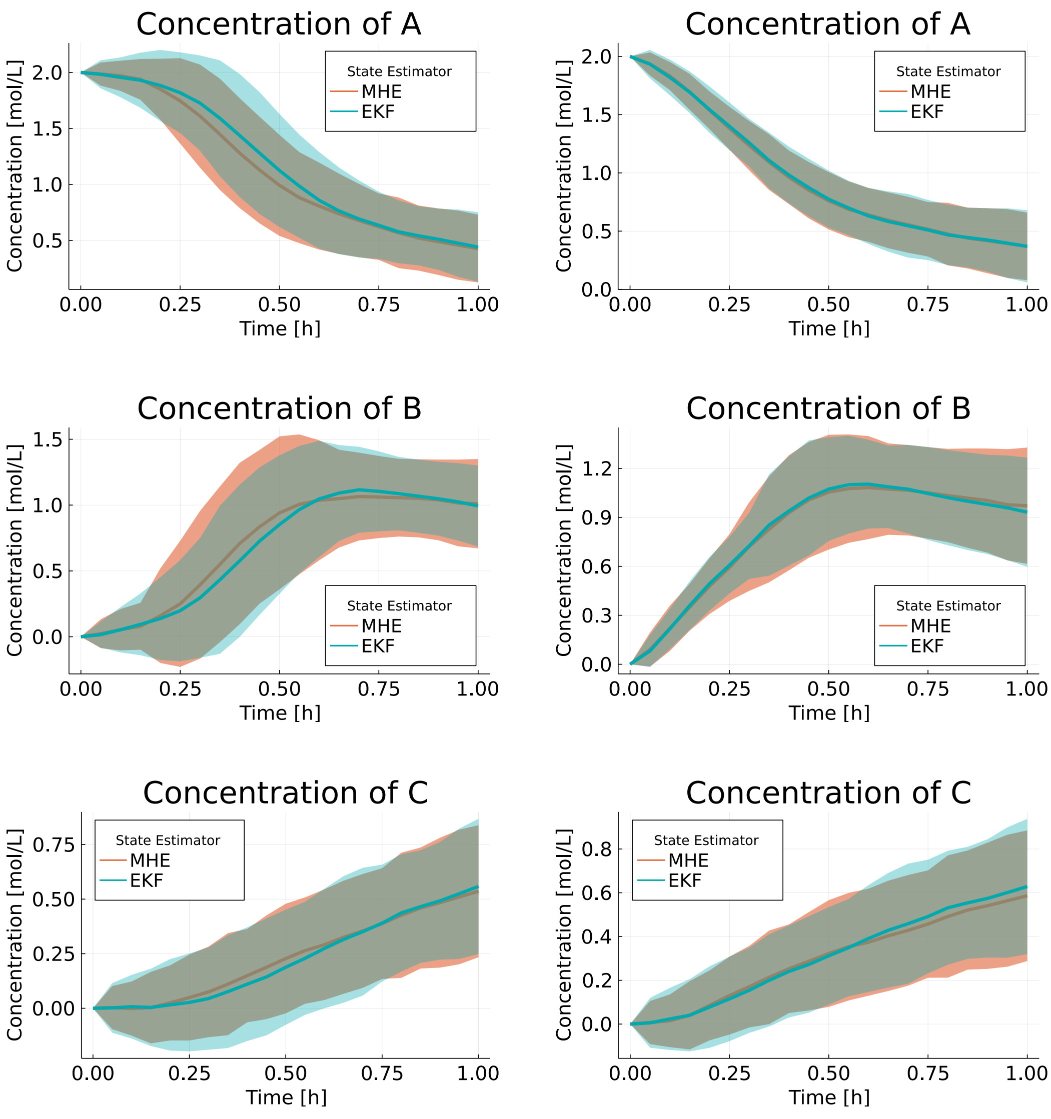


Figure 4.2.1: Results from Case Study 1 and 2. The most notable result is the change of the V_R trajectory with the increased horizon $N = 20$

With an increased CPU time, the uncertainty with $N = 20$ remains largely the same for most of the states compared to the shorter horizon $N = 5$. However, a rather interesting result is the trajectory of V_R that, with the increased length, makes the MHE achieve a performance almost identical to the EKF. This result is preferable, as the V_R trajectory from the MHE will converge towards the operational, and not the physical constraint. Violating the latter, which would result in a spillover, would be problematic for the optimisation problem and it is therefore desirable if the state trajectories rather converge, and remain, at the operational constraint. This increased performance of the MHE do obviously come at the expense of increased CPU time, and the cost effectiveness of this, along with the results of the other case studies, is a topic that is to be evaluated later.

4.2.2 Significance of NMPC prediction length

Similar to the length of the estimation window, the length of the prediction horizon of the NMPC, $NFE0$, could also have a significance on the outputted state trajectories. For the base case, this length was $NFE0 = 20$, but it would be interesting to investigate if whether a more cost effective alternative, such as $NFE0 = 5$ and $NFEO = 15$ could output trajectories with acceptable accuracy. The respective results of these lengths are given in Figure 4.2.2:



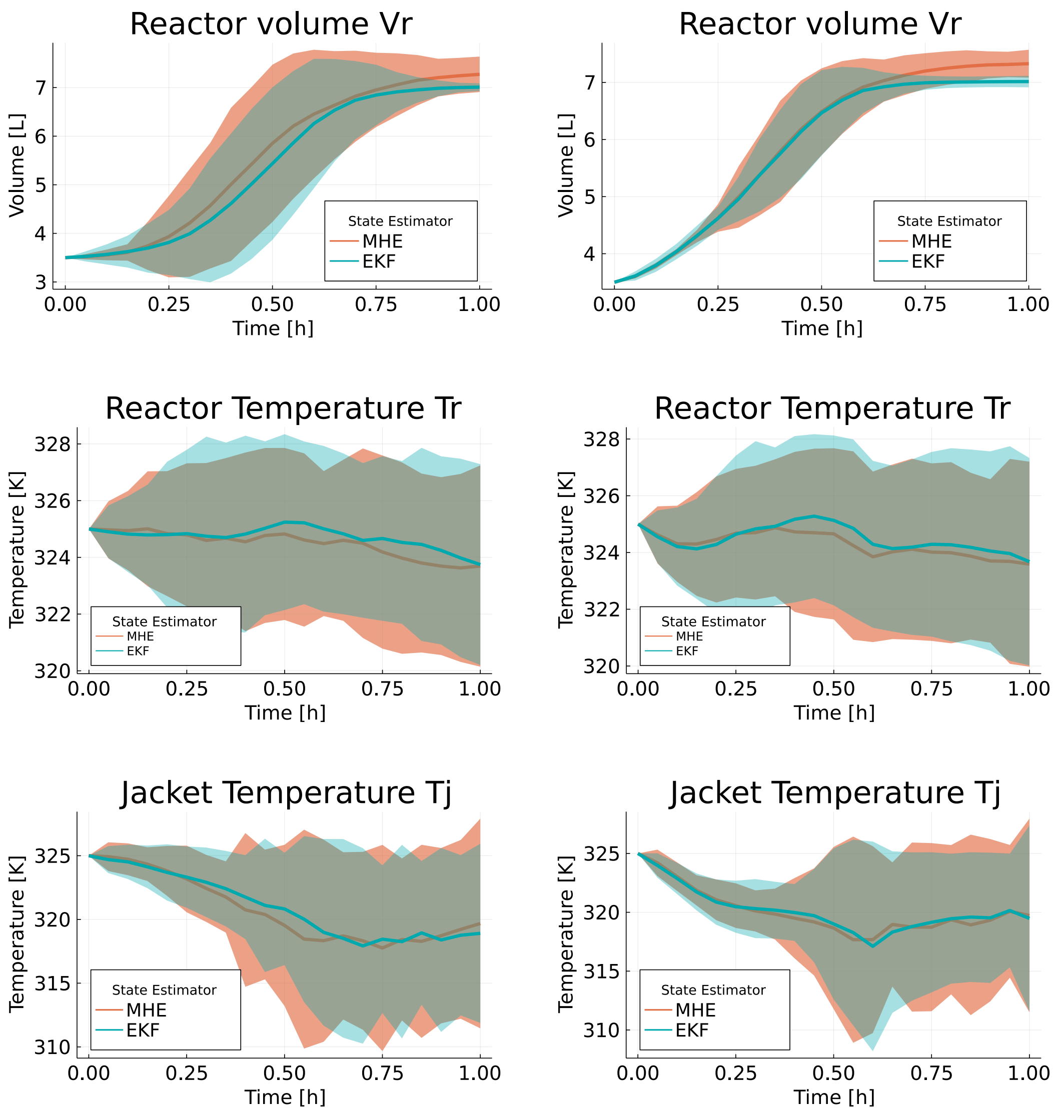
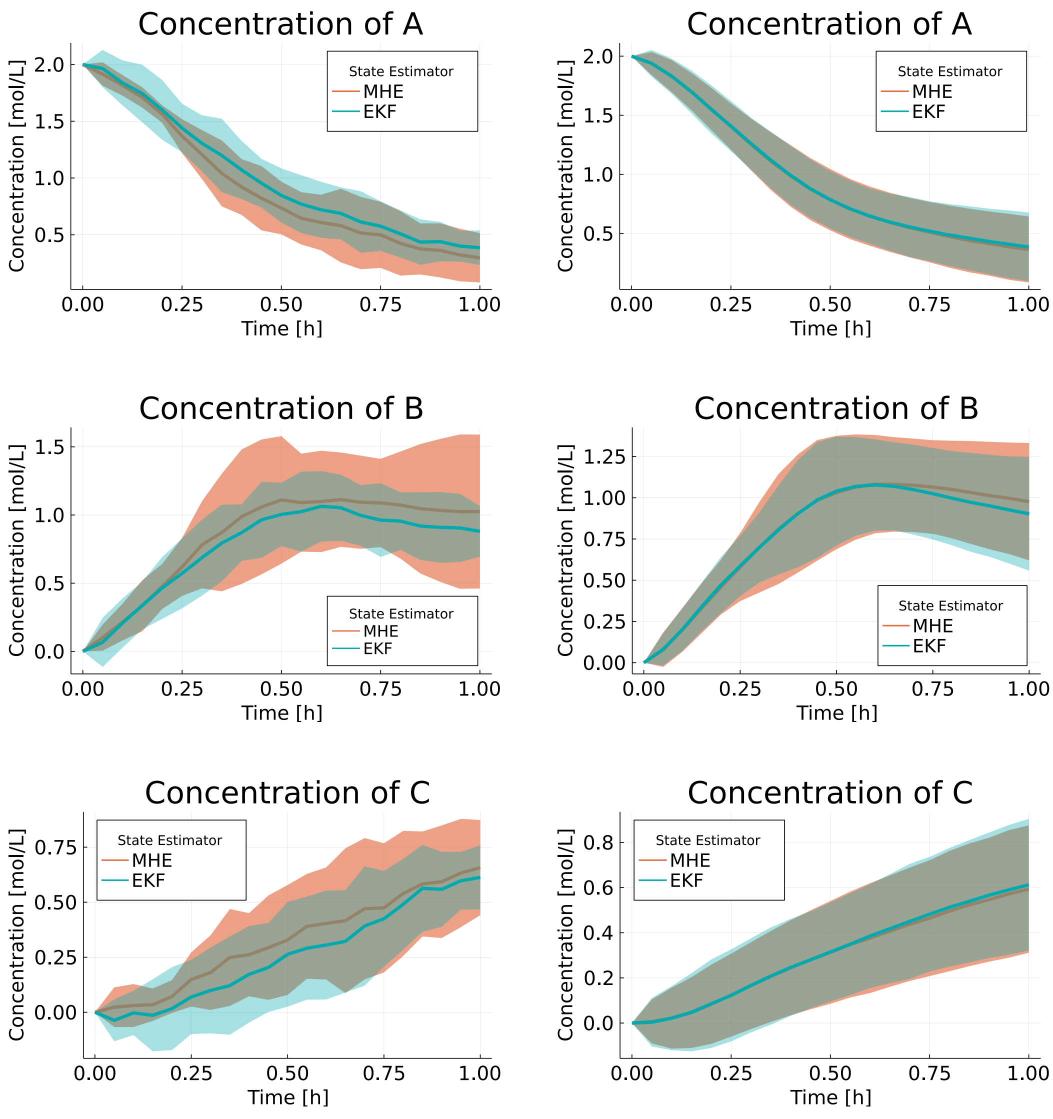


Figure 4.2.2: Results from Case Study 3 and 4.

Based off these results, it is apparent that the short horizon, $NFE0 = 5$ results in greater standard deviations for the EKF, and therefore more uncertainty. This is an expected result, as the short horizon will not be able to predict future state trajectories to the same degree as the longer horizon $NFE0 = 15$. On the other hand, the shorter horizon will require significantly less CPU time compared to the longer alternatives $NFE0 = 15$ and $NFE0 = 20$. Because of this, a tradeoff between accuracy, and therefore operational reliability, and the corresponding CPU time exists. In the case of $NFE0 = 15$, it appears that this length provide almost identical trajectories to $NFE0 = 20$ with the exception of the V_R which regardless output trajectories with reasonable accuracy. Overall, the main takeaway from these cases are therefore that the prediction horizon length $NFE0 = 15$ is a more cost effective alternative than $NFE0 = 20$, as the former output approximately the same state trajectories at the cost of only minor deviations.

4.2.3 Significance of amount of seeds

As mentioned earlier, the base case results given in Figure 4.1.3 represent the mean state trajectories which were calculated from the seed states $1 - 100$. It is therefore questionable if this method yield representative enough results where general trends, or behaviour, can be discussed and conclusions drawn. Case 5 and Case 6 therefore investigate the effect of the amount of data points, or seed states, has on the calculated state trajectories. The results are given in Figure 4.2.3:



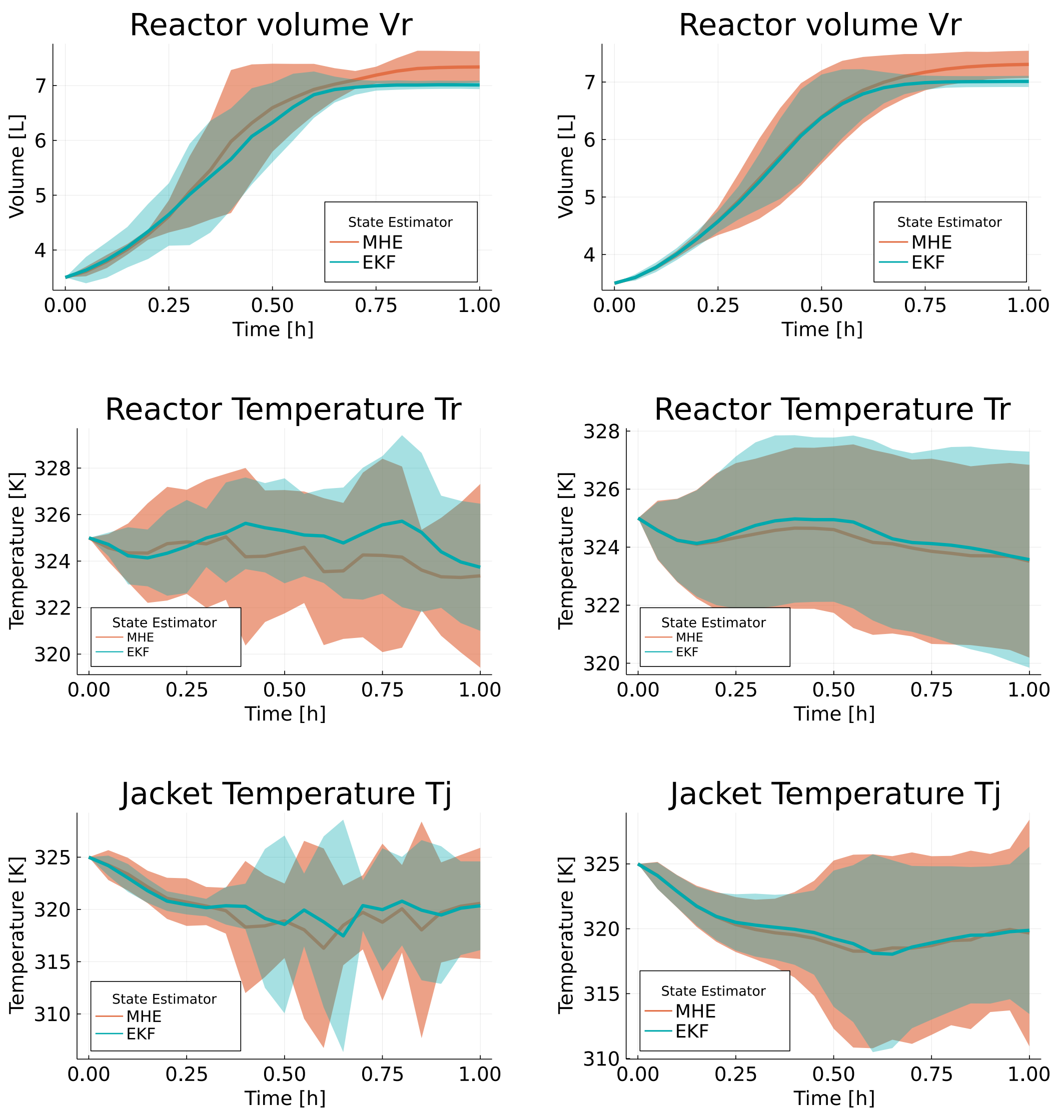


Figure 4.2.3: Results from Case Study 5 and 6.

Unfortunately, with only 10 datapoints, or seed states, the state uncertainties becomes much greater, especially for the MHE. This is expected as fewer datapoints will naturally reflect less knowledge about general trends of the optimisation problem. What could however be interesting was if the fewer datapoints could somehow output state trajectories similar to the base case with 100 datapoints, thereby becoming a more cost effective alternative. This proved to not be case and indicates that state trajectories should at least consist of more than 10, for example 50, datapoints if the results are to be representative of general behaviour.

For Case 6, the amount of data points was incread to 1000 in order to investigate if this resulted in new or interesting trends which the base case with 100 data points could not achieve. As a consequence of this, the CPU time to calculate this increased by a tenfold and unfortunately output almost identical state trajectories to the base case. These results do however indicate that the initial amount of 100 datapoints in the base case appear to provide representative results, as increasing this amount do not yield new results. Using 100 datapoints, or seed states, therefore appears to be a cost effective choice when analysing trends of the optimisation problem.

4.2.4 Significance of initial guess

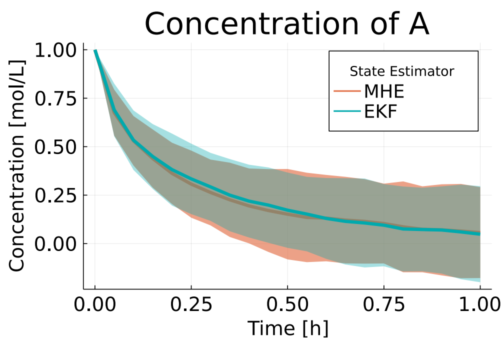
When attempting to solve a nonlinear optimisation problem, the choice of an initial guess will be detrimental as it will initialise the iterative algorithm of the solver. The guess will therefore be significant for both the CPU time and final solution, as it determines how quickly the solver converges towards a global extremum. As nonlinear problem tend to consist of multiple local extrema, the initial guess will also have an affect on which of those the solver will converge towards. [28] For Case 7 and 8, there are 3 aspects that are of interest:

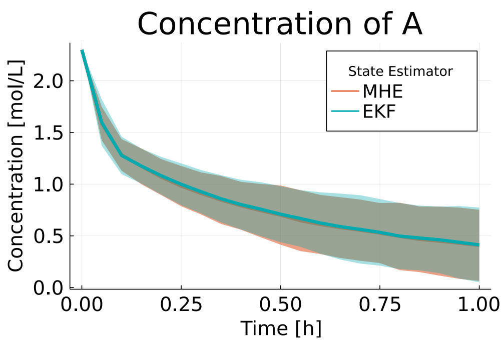
- Sensitivity of the optimisation problem towards initial guess
- Robust performance of NMPC despite "bad" initial guesses
- The effect the initial guess has on the amount of produced C

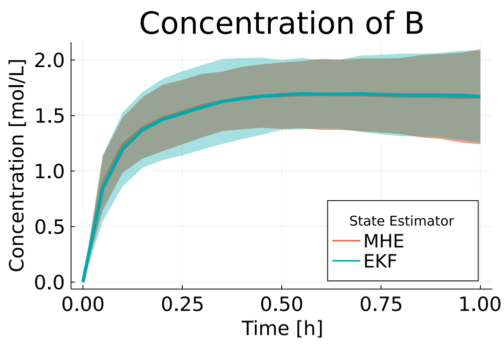
Case 7 has an initial guess, $x_{0,2}$ which consist of minor differences compared to the base initial guess x_0. The purpose of $x_{0,2}$ is to investigate the sensitivity the optimisation problem has towards a minor change of initial guess. Case 8 on the other hand uses an intentional "bad" guess, $x_{0,3}$, which is used to highlight whether the NMPC has robust, or consistent, performance despite given an inaccurate initialisation. The initial guesses for Case 7 and 8 are given in Table 4.2.2 and the corresponding results in Figure 4.2.4:

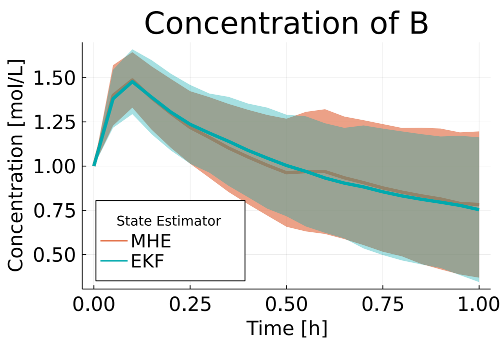
Table 4.2.2: Intial guesses for Case 7 and 8

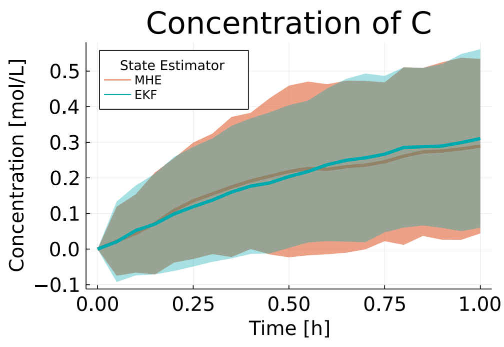
Case	**Initial V_R**	**Initial C_A**	**Initial C_B**	**Initial T_R**	**Initial T_J**
Base	3.5	2.0	0.0	325.0	325.0
7	2.5	1.0	0.0	328.0	328.0
8	4.5	2.3	1.0	330.0	330.0

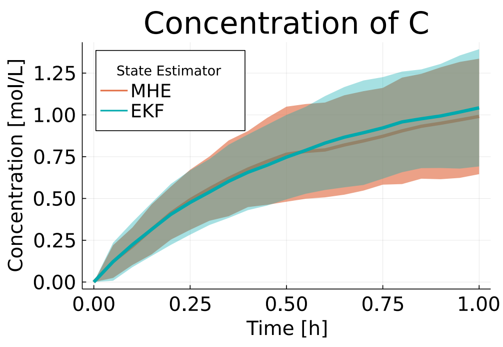

Concentration of A
Concentration [mol/L]
Time [h]
State Estimator
MHE
EKF


Concentration of A
Concentration [mol/L]
Time [h]
State Estimator
MHE
EKF


Concentration of B
Concentration [mol/L]
Time [h]
State Estimator
MHE
EKF


Concentration of B
Concentration [mol/L]
Time [h]
State Estimator
MHE
EKF


Concentration of C
Concentration [mol/L]
Time [h]
State Estimator
MHE
EKF


Concentration of C
Concentration [mol/L]
Time [h]
State Estimator
MHE
EKF

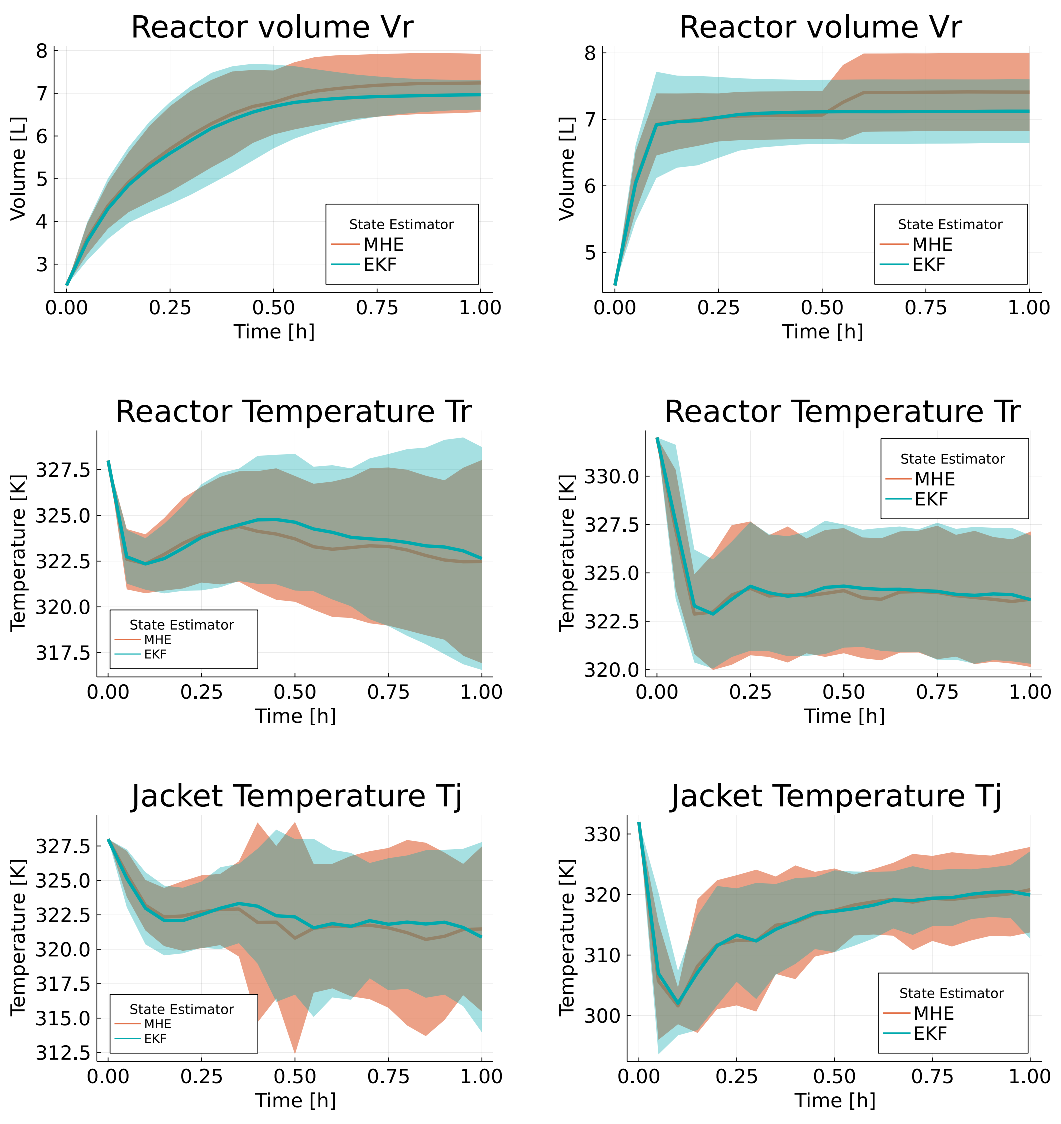


Figure 4.2.4: Results from Case Study 7 and 8.

For Case 7 with the minor perturbation of $x_{0,2}$ proved to not produce results that were of particular interest. The states had approximately the same trajectories compared to the base cases, albeit with shifted values along y-axis. The most interesting results was of the MHE T_J trajectory which consists of peaks at the half point of the simulation. These spikes were not present in the base case and could indicate excessive use of control input Q_k caused by $x_{0,2}$.

For greater perturbations of x_0, such as x_3, it seems that the initial guess overall have a significant effect on the state trajectories. In both cases it appears that the NMPC was able to calculate similar state trajectories compared to the base case. This indicate that the NMPC is robust and functional despite varying initial guesses. For Case 8, the initial T_R was outside the range of the operational constraint, but despite this, the NMPC was able to adjust itself to this inconvenience and eventually output a state trajectory similar to the base case.

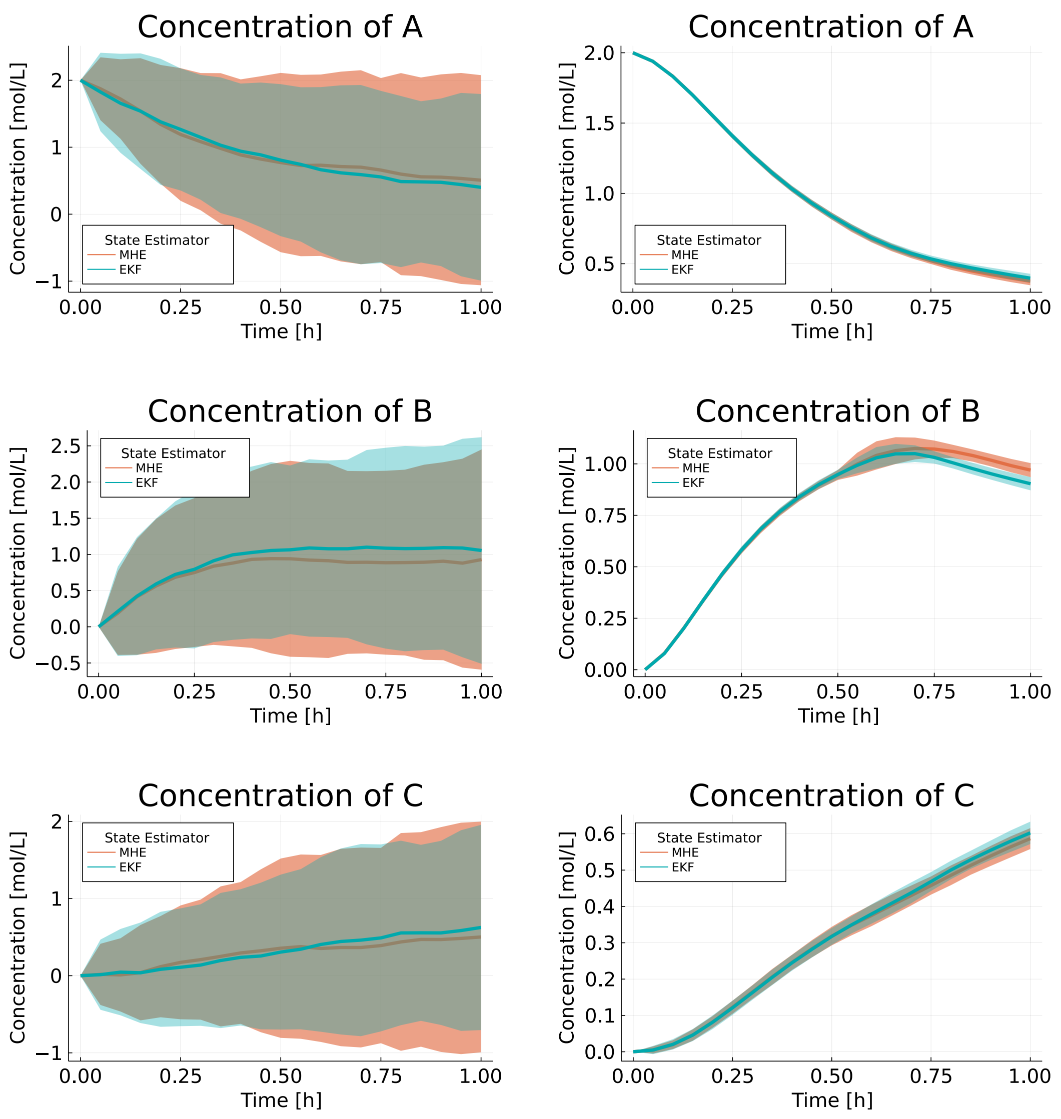
One of the more notable results is the production of C in Case 8, which is significantly higher than previous case studies. This is expected as the initial V_R, and therefore, C_A, and C_B, were greater than the base case which resulted in the semi-batch reactor producing more C throughout the operation. This indicate that inital V_R,C_A, and C_B have a significant effect on the final amount of produced C. Finally, it is worth mentioning that the CPU time to calculate these results in Case 7 and 8 were significantly higher compared to the base case with x_0. This could indicate that the initial guesses $x_{0,2}$ $x_{0,3}$ were inferior CPU time wise compared to x_0, as they resulted in the optimiser requiring more iterations in order to evaluate and calculate feasible solutions.

4.2.5 Significance of standard deviation of generated noise

So far the random variables, or more specifically the process and measurement noise, have been generated within a Gaussian distributions with fixed standard deviations σ_ω,σ_v. The values of the standard deviations determine the degree uncertainty in the optimisation problem and it would be interesting to investigate if changes to these could affect the state trajectories. The values of the respective process and measurement noise standard deviations for Case 9 and 10 are given in Table 4.2.3 and the corresponding results in Figure 4.2.5

Table 4.2.3: Standard deviations for Case 9 and 10

Case	**Variable**	**Value**
Base	$\sigma_{\omega,1}$	$0.01, 0.05, 0.05, 0.5, 0.5$
Base	$\sigma_{v,1}$	$0.4, 1.0, 1.0$
9	$\sigma_{\omega,2}$	$0.05, 0.20, 0.20, 2.0, 2.0$
9	$\sigma_{v,2}$	$0.8, 2.0, 2.0$
10	$\sigma_{\omega,3}$	$0.001, 0.005, 0.005, 0.05, 0.05$
10	$\sigma_{v,3}$	$0.04, 0.1, 0.1$


Concentration of A
Concentration of A
Concentration of B
Concentration of B
Concentration of C
Concentration of C
Concentration [mol/L]
Time [h]
State Estimator
MHE
EKF

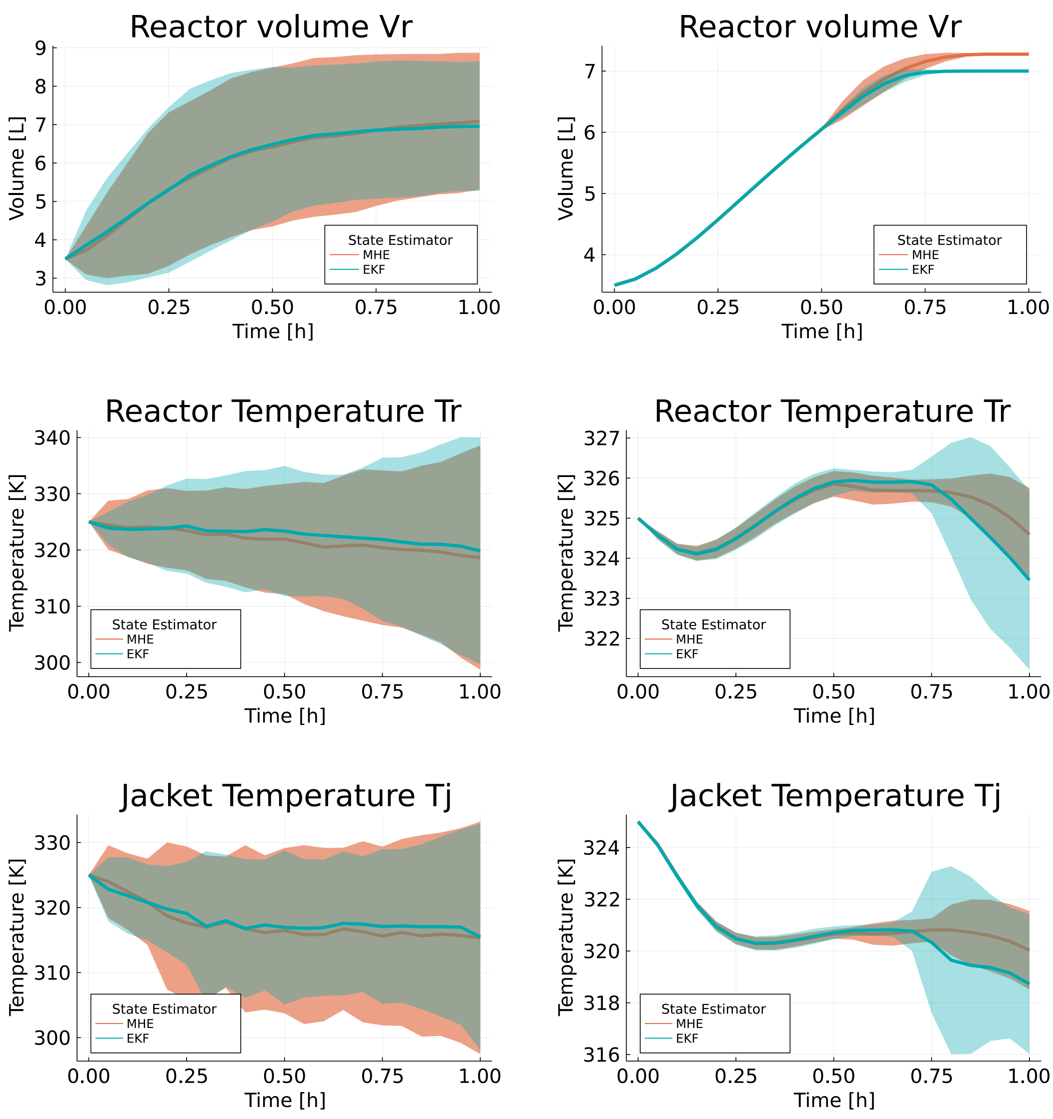


Figure 4.2.5: Results from Case Study 9 and 10.

For Case 9, the standard deviations were increased by 4 and 2 times respectively and as expected this greatly increased the uncertainty of the problem. Regarding Case 10, the tenfold decrease of σ_ω,σ_v did, as expected, result in a significant decrease of the uncertainty.

However, an interesting result are the trajectories of T_R and T_J where the uncertainty significantly increase towards the end of the simulation. With a shrinking horizon towards the end of operation, the NMPC will attempt to further maximise the production of C by adding more reactants A and B with the inflow input. This results in more heat generated which has to compensated by the cooling input, and it appears that how these actions are executed varies greatly for each of the seed states. This uncertainty seems to be much more apparent for Case 10 with the small values of σ_ω,σ_v compared to Case 9.

Despite the varying values of σ_ω,σ_v, the mean of the state trajectories remain roughly the same compared to the base case. Because of that it seems that while the uncertainties of the optimisation problem are sensitive to changes in σ_ω,σ_v, the overall performance and production of C remains roughly the same. This could indicate that the estimators achieves their operational goal of producing filtered estimates for the NMPC despite changes in σ_ω,σ_v.

4.2.6 Significance of regularisation terms

So far in the report, the operational behaviour of the control inputs have not been presented. The reason for this is that the input profiles have displayed irregular behaviour, which is expected as the control actions will vary significantly because of the random variables that are uniquely generated for each seed state. To elaborate this argument further, the control profiles of both MHE and EKF from seed state 1 and 612 are given in Figure 4.2.6 and 4.2.7:

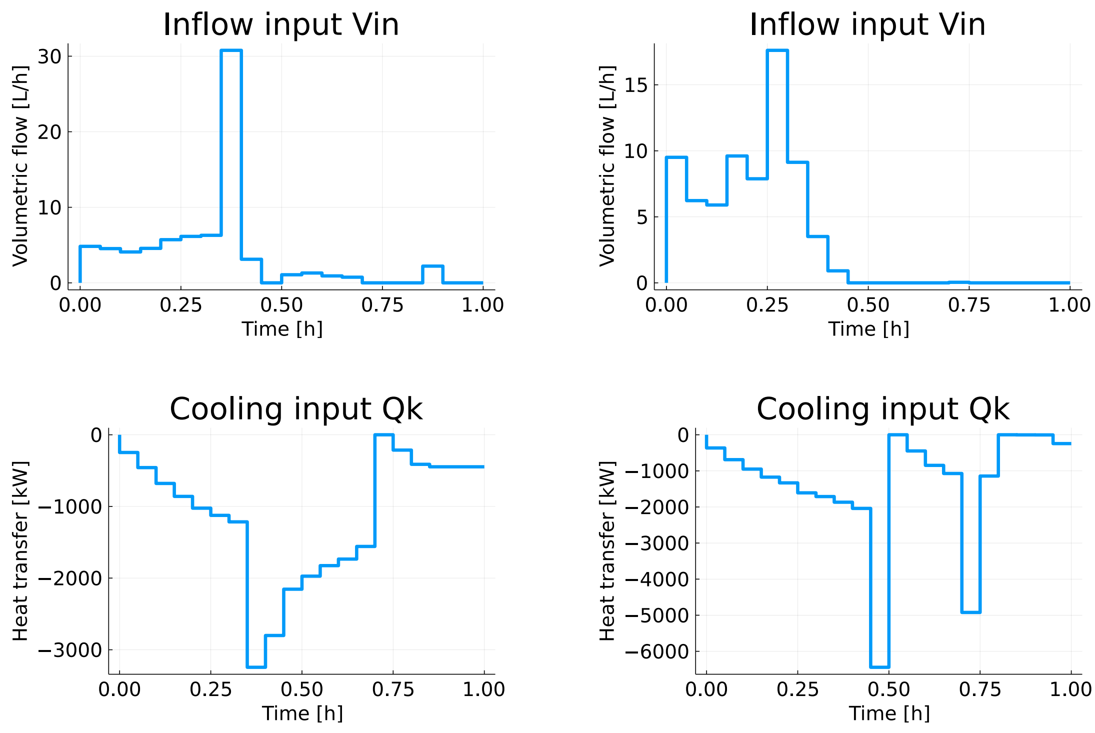


Figure 4.2.6: Example of control input profile taken from Seed 1. To the left are the input profile for the state estimator MHE and to the right from EKF

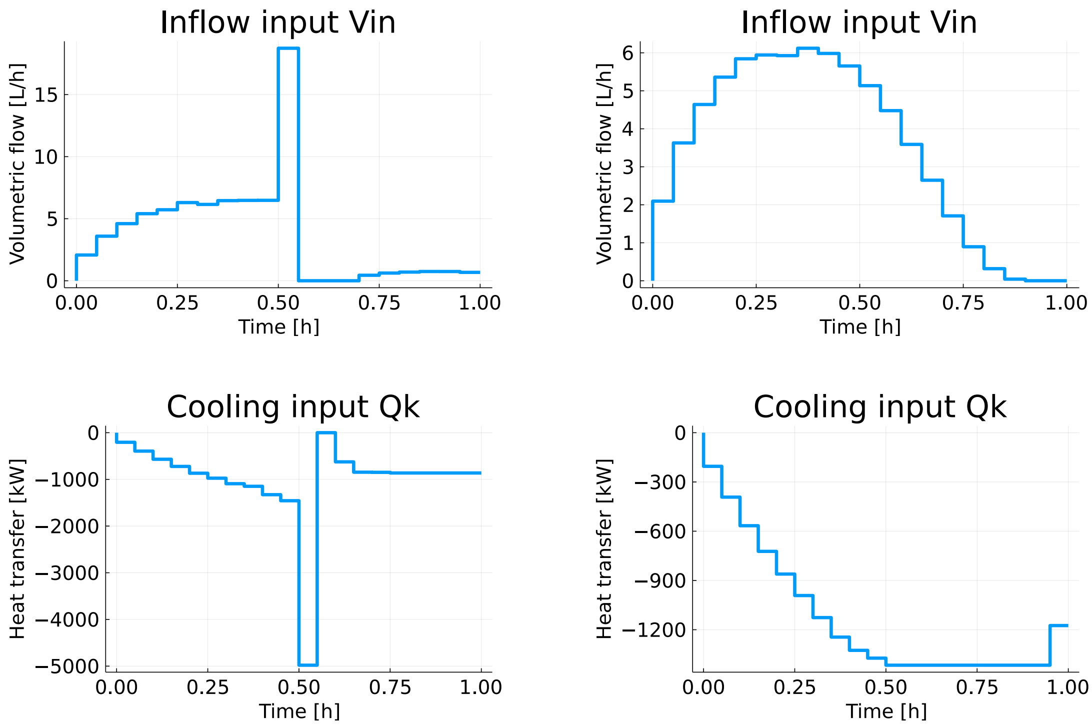


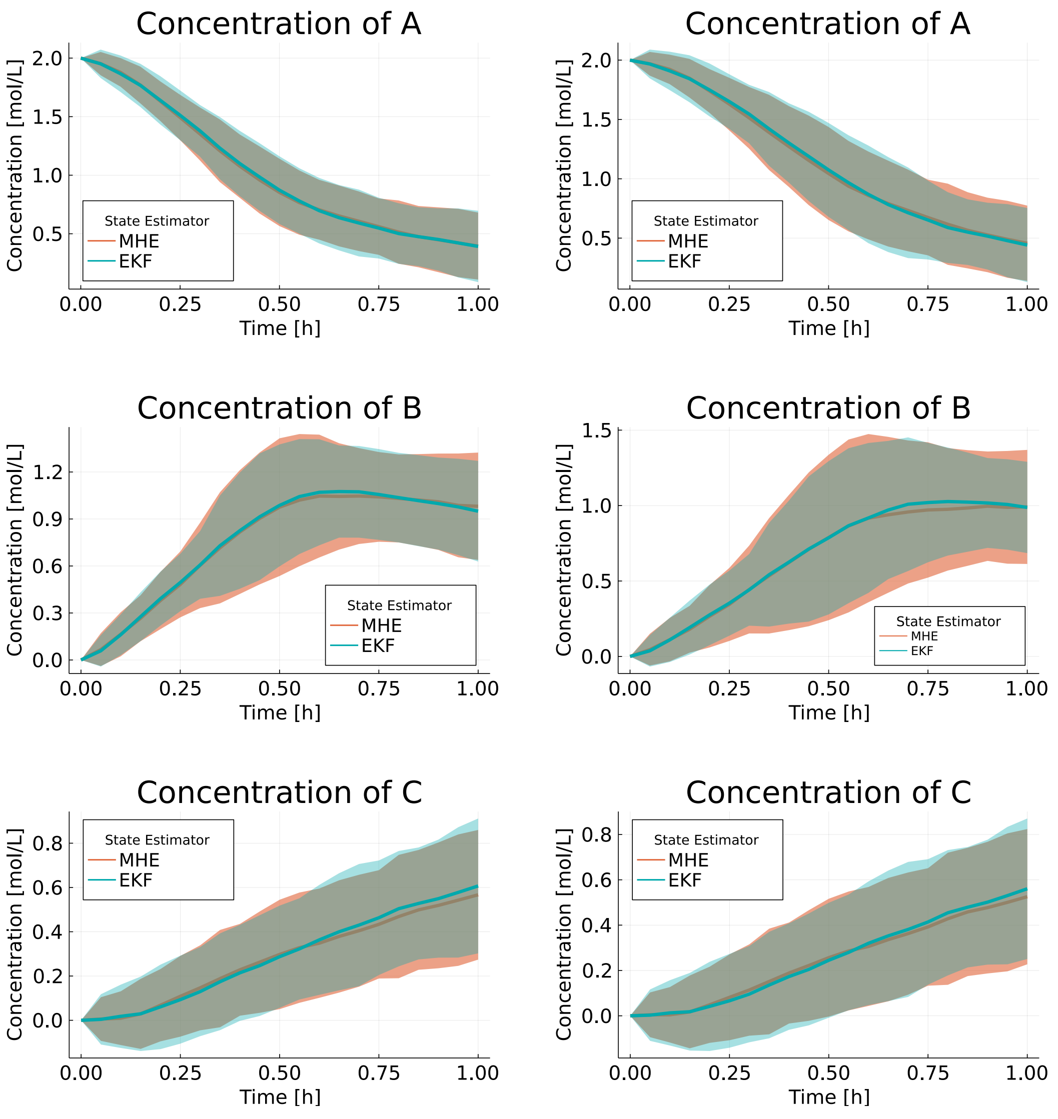
Figure 4.2.7: Another example of the control profiles, this time from seed state 612

Based off these examples, it appears that the NMPC has a tendency to "spike" for some of the iterations. On the other hand, there are many of the seeds who do not display this similar behaviour and have "smooth" control input trajectories, such as seed 612 with EKF. Through trial and error with a series of different seeds, it was observed that the control profiles in general attempted to follow the trajectory where V_{in} would increase to $\approx$ 6.5L before decreasing halfway during operation and Q_k converge towards a steady state value of ≈ -1500kW. The main problem were the inconsistent and sudden spiking the NMPC would execute at approximately halfway through the operation. This is an issue as the control inputs such as these seem to have destabilised, or complicated, the convergence of the input trajectories. It is questionable how problematic this actually is, but it would preferable if the inputs trajectories were converging and thus provided predictability and operational stability.

This erratic behaviour of the NMPC is probably the main cause of the uncertainties that have been presented and discussed so far in this report. These spikes are roughly occurring either once or twice for each of the seed states, and the calculated trajectories will therefore have a significant standard deviation which again results in increased uncertainty of the mean trajectories. Addressing the uncertainty that stem from these spikes are the main motivation for conducting Case study 11 and 12. For each of the cases, the weighting of the regularisation terms of the cost function, $\Delta \dot{V}_{\text{in}}$ and $\Delta \dot{Q}_k$, are to be increased. The motivation of this is that the increased weights would discourage spikes by further penalising the cost function for subsequent control inputs that had a significant difference. The settings of Case 11 and 12 are given in Table 4.2.4 and the results in Figure 4.2.8:

Table 4.2.4: Regularisation terms for Case Study 11 and 12

Case	Variable	Weight
Base	$\Delta\dot{V}_{\text{in}}$	0.0840
Base	$\Delta\dot{Q}_1$	$5.5 * 10^{-5}$
11	$\Delta\dot{V}_{\text{in},2}$	0.1708
11	$\Delta\dot{Q}_2$	$7.5 * 10^{-5}$
12	$\Delta\dot{V}_{\text{in},3}$	0.8054
12	$\Delta\dot{Q}_3$	$9.5 * 10^{-5}$



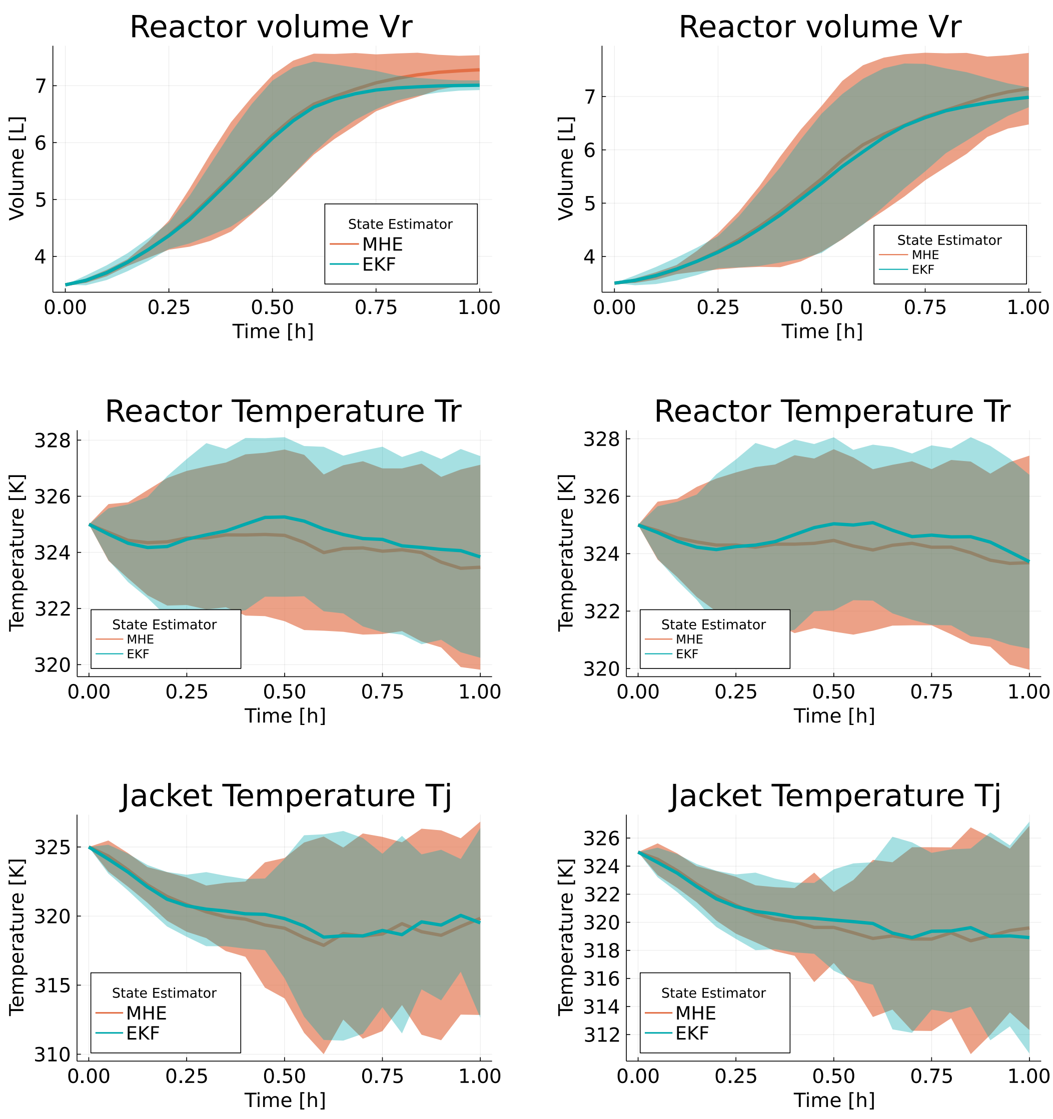


Figure 4.2.8: Results from Case Study 11 and 12.

Unfortunately, it appears that the increased weighting of the regularisation terms did not result in a significant improvement. For Case 12, it seems that the increase of the the V_{in} input actually resulted in more uncertainty which are apparent for the C_B and V_R trajectories. However, for the T_J trajectory, the uncertainty associated with the MHE decreased overall. This could indicate that the significant weighting of V_{in} complicated the optimisation problem, while the moderate weighting for the Q_k input forced the NMPC to perform less spikes, thus decreasing the overall uncertainty.

These results confirms that the weights of the regularisation terms make a difference towards the performance of the NMPC. The main motivation for these cases was however to investigate if changing the weights could result in an immediate improvement of the performance. Finding the optimal weights do however remain outside the scope of this thesis, as it will involve a significant amount of trial and error. The main takeaway from these case studies are therefore to encourage finding more optimal weights of the regularisation terms as further work.

4.2.7 Overall thoughts from case studies

The performance and results of each of the case studies have been quantified and stored in Table 4.2.5 in an effort to generate some overall ideas and conclusions.

Table 4.2.5: Production of C from Case Studies. CPU denotes the time it took to solve the various Case studies.

Unit	[mol/L]	[mol/L]	[min:s]	[min:s]
Case	C_{EKF}	C_{MHE}	CPU$_{EKF}$	CPU$_{MHE}$
1	0.6260	0.5702	02:45	03:27
2	0.6260	0.5980	02:45	04:35
3	0.5582	0.5362	01:19	02:42
4	0.6286	0.5865	02:42	03:52
5	0.6126	0.6573	00:24	00:31
6	0.6124	0.5933	26:29	47:50
7	0.3105	0.2891	04:45	06:38
8	1.042	0.9909	11:22	13:31
9	0.6253	0.5020	07:40	09:46
10	0.6025	0.5867	02:27	03:43
11	0.6070	0.5670	02:49	04:00
12	0.5606	0.5256	02:31	04:21

Overall, it appears that, with the exception of Case 8 and 9, the production of C has remained largely the same at $C \approx 0.60$ [mol/L] despite the different case settings. This indicates that the individual settings may not affect the output of C significantly on their own. However, one of the most interesting results from Case 2 indicated that the base case length of the estimation window, $N = 10$, resulted in sub-optimal operation by MHE compared to EKF, as the former converged towards the physical, $V_R \leq 8L$ and not operational constraint at $V_R \leq 7L$.

According to the results summarised in Table 4.2.5, the EKF consistently outperformed, with the exception of Case 5, the MHE in terms of production of C and required CPU time. This indicates that the EKF is most likely a more cost effective choice of estimator than the MHE, as the latter is supposed to provide more accurate estimates to the NMPC at the expensive of increased computational time. There could be several reasons for the underperformance of the MHE, but it is difficult to pinpoint the exact cause of reason for this behaviour. At this point, the optimisation problem is, with the state estimators, random noise, NMPC, orthogonal collocation and so on, quite complicated and the under performance could either be caused by a single element or an accumulation of different errors or assumptions.

However, based off Case 2, a probable reason for the underperformance of the MHE could be the choice of arrival cost method. As discussed in the methodology chapter, the arrival cost Γ_k is estimated and the accuracy and cost effectiveness of this method remains questionable. [29]

CHAPTER

FIVE

CONCLUSION

5.1 Conclusion

Overall, it seems that the operational goal of maximising the production of C has been achieved despite the challenges posed by the random noise and immeasurable states. Based off the base case results, the concentration of C in the semi-batch reactor strictly increase throughout the simulation and the corresponding state trajectories display realistic behaviour. The constraints of the semi-batch reactor, most notably $V_R \leq 7\text{L}$ and $322K \leq T_R \leq 326K$, are not violated which indicate that the NMPC is able to consistently solve the optimisation problem. It also seem that the NMPC is robust to process and measurement noise of the semi-batch reactor and overall has been able to control the semi-batch reactor in a reliable manner.

However, the NMPC would not able to solve the optimisation problem without the state estimators MHE and EKF, which implementations for the semi-batch reactor have been the main focus of this thesis. Overall, it appears for the state estimators that the operational goal of producing filtered state estimates for the NMPC have been achieved. The estimators produce similar results compared to the real state trajectories, although there is a minor difference in performance between the methods. Based off the case study results given in Table 4.2.5, it could be argued that EKF is a more cost effective choice than MHE since the former consistently results in slightly more C produced than the latter with significantly less CPU time. It was also observed for the MHE that increasing the length of the estimation window from $N = 10$ to $N = 20$ resulted in improved, and almost identical trajectories compared to the EKF.

According to literature [7], the MHE is supposed to produce more accurate estimates than the EKF and the under performance of the former is most likely caused by the various simplifications and estimations that has been made throughout this thesis. Examples of these can be the choice of method for estimating the arrival cost Γ_k, the weights of different variables, or simply implementation error in the code.

CHAPTER

SIX

FURTHER WORK

6.1 Further work

The main objective of this thesis was to perform a comparative analysis of implementing EKF and MHE for the semi-batch reactor in order to address uncertainty and produce state estimates of C_A and C_B which were considered immeasurable. Even though this objective was overall achieved based off the results, there are still a series of factors that has not been properly addressed in this thesis. The most significant is the arrival cost for the MHE, Γ_k, which was calculated using the first estimate, $\hat{x}_{k-N}$, outside the estimation window. As explained earlier in this report, this estimation may be inaccurate and alternative methods, such as NLP sensitivity analysis [29], are currently being researched on.

One of the more worrisome results from the case studies were from Case 10 which investigated the significance of the values of σ_ω and σ_v. For the temperature states, T_R and T_J it was observed that the uncertainty increased significantly towards the end of the simulations, despite σ_ω and σ_v being minimal. It is believed that this is caused by excessive usage of the cooling input, Q_k, in order to counter the increasingly generated heat towards the end of operation, although the exact cause remains uncertain. Investigating the behaviour and cause of the changing behaviour of the NMPC towards the end of the simulation is therefore recommended as further work.

In the dynamic model of the semi-batch reactor the respective reaction constant and enthalpy, k and H, have throughout this thesis been estimated to remain fixed at specific values. This simplification may had a significant impact on the presented results as they are actually uncertain variables as well. Originally, the research paper which this thesis has taken the optimisation problem from, originally investigates a multi-stage NMPC which branches these variables in upper and lower values. In this thesis, k and H were assumed to remain fixed despite this as implementing a multi-stage NMPC remained out of scope. Somehow properly addressing the uncertain parametric variables is therefore encouraged as further work, as this will make the dynamic model more realistic, and hopefully result in decreased uncertainty.

Finally, as presented in Case 11 and 12, addressing the spiking behaviour of the NMPC is advised. Irregular occasions of the control action spikes significantly disrupt the control action trajectories and results in more uncertainty associated with the mean state trajectories. Forcing the NMPC to not perform such spikes is therefore encouraged as further work and could for instance revolve around reevaluating the weighting of the regularisation terms, NMPC settings, or the cost function entirely.

REFERENCES

[1] *101 Companies Comitted To Reducing Their Carbon Footprint.* URL: https://forbes.com/sites/blakemorgan/2019/08/26/101-companies-committed-to-reducing-their-carbon-footprint/?sh=26273f86260b.

[2] Max Schwenzer, Thomas Bergs, and Dirk Abel. "Review on model predictive control: an engineering perspective". In: *The International Journal of Advanced Manufacturing Technology* 117 (2021).

[3] François Auger et al. "Industrial Applications of the Kalman Filter: A Review". In: *IEEE Transactions on Industrial Electronics* (2013).

[4] Andrea Tuveri et al. "Bioprocess Monitoring: A Moving Horizon Estimation Experimental Application". In: *IFAC-PapersOnLine* (2013).

[5] Bashar Alsadik. "Kalman Filter". In: Jan. 2019, pp. 299–326. ISBN: 9780128175880. DOI: 10.1016/B978-0-12-817588-0.00010-6.

[6] Christopher Rao and James Rawlings. "Constrained Process Monitoring: Moving-Horizon Approach". In: *AIChE Journal* 48 (Jan. 2002), pp. 97–109. DOI: 10.1002/aic.690480111.

[7] Eric Haseltine and James Rawlings. "Critical Evalation of Extended Kalman Filtering and Moving-Horizon Estimation". In: *Industrial and Engineering Chemistry Research* 44 (Apr. 2005), pp. 2451–2460. DOI: 10.1021/ie0343081.

[8] Sakthi Thangavel, Radoslav Paulen, and Sebastian Engell. "Multi-stage NMPC using sigma point principles". In: *IFAC-PapersOnLine* 53 (2020), pp. 18–19. ISSN: 2405-8963.

[9] Md. Sohel Rana, Hemanshu Pota, and Ian Petersen. "Performance of sinusoidal scanning with MPC in AFM imaging". In: *IEEE/ASME Transactions on Mechatronics* 20 (Jan. 2014). DOI: 10.1109/TMECH.2013.2295112.

[10] Yonghwan Jeong and Seongjin Yim. "Model Predictive Control-Based Integrated Path Tracking and Velocity Control for Autonomous Vehicle with Four-Wheel Independent Steering and Driving". In: *Electronics* 10 (Nov. 2021), p. 2812. DOI: 10.3390/electronics10222812.

[11] Bjarne Foss and Tor Aksel N. Heirung. *Merging Optimization and Control.* Mar. 2016, pp. 18–19. ISBN: 978-82-7842-201-4.

[12] G.F. Carey and Bruce A. Finlayson. “Orthogonal collocation on finite elements”. In: *Chemical Engineering Science* 30.5 (1975), pp. 587–596. ISSN: 0009-2509.

[13] Shakir Saat, Sing Kiong Nguang, and Alireza Nasiri. “Chapter 1 - Introduction”. In: *Analysis and Synthesis of Polynomial Discrete-Time Systems.* Ed. by Shakir Saat, Sing Kiong Nguang, and Alireza Nasiri. Butterworth-Heinemann, 2017, pp. 1–27. ISBN: 978-0-08-101901-6. DOI: https://doi.org/10.1016/B978-0-08-101901-6.00001-3.

[14] Crina Grosan and Ajith Abraham. “Multiple Solutions for a System of Nonlinear Equations”. In: *International Journal of Innovative Computing, Information and Control* 4 (Sept. 2007).

[15] *Solver for Nonlinear Programming.* URL: https://www.wiomax.com/solver-for-nonlinear-programming (visited on 04/10/2023).

[16] *Semi-Batch or Semi-Continuous Reactor.* URL: https://www.processoperations.com/CKRE/RE_Chp03c.htm (visited on 05/31/2023).

[17] Erling Selvig. *Optimal online control of a semi-batch reactor with varying prediction horizon.* 2022. URL: https://folk.ntnu.no/jaschke/Masters/ProjectTheses/2022/ErlingSelvig/ (visited on 06/15/2023).

[18] *Noise and disturbances in process control.* URL: https://controleng.com/articles/noise-and-disturbances-in-process-control (visited on 06/05/2023).

[19] *Tuning Kalman Filter to Improve State Estimation.* URL: https://mathworks.com/help/fusion/ug/tuning-kalman-filter-to-improve-state-estimation.html (visited on 05/06/2023).

[20] National Cancer Institute. *Learn More about Normal Distribution.* URL: https://dietassessmentprimer.cancer.gov/learn/distribution.html (visited on 06/15/2023).

[21] *Gaussian analysis.* URL: https://ml-science.com/gaussian-analysis (visited on 05/18/2023).

[22] *Multivariate Normal Distribution.* URL: https://en.wikipedia.org/wiki/Multivariate_normal_distribution (visited on 05/18/2023).

[23] Wei He, Songsheng Zhu, and Wei Wang. “A measurement method of batch solution concentration based on normalized compressed sensing”. In: *Measurement and Control* 53 (Jan. 2020), p. 002029401988296. DOI: 10.1177/0020294019882964.

[24] Ronald Alexander et al. “Challenges and Opportunities on Nonlinear State Estimation of Chemical and Biochemical Processes”. In: *Processes* 8 (Nov. 2020), p. 1462. DOI: 10.3390/pr8111462.

[25] Maria Ribeiro. *Kalman and Extended Kalman Filters: Concept, Derivation and Properties.* URL: https://researchgate.net/publication/2888846_Kalman_and_Extended_Kalman_Filters_Concept_Derivation_and_Properties (visited on 05/20/2023).

[26] Finn Haugen. *State estimation withKalman Filter*. 2007. URL: `techteach.no/fag/seky3322/0708/kalmanfilter/kalmanfilter.pdf` (visited on 05/20/2023).

[27] Mohamed Elsheikh et al. "A comparative review of multi-rate moving horizon estimation schemes for bioprocess applications". In: *Computers and Chemical Engineering* 146 (2021). ISSN: 0098-1354.

[28] Rick Wicklin. *How to find an initial guess for an optimization*. 2014. URL: `https://blogs.sas.com/content/iml/2014/06/11/initial-guess-for-optimization.html` (visited on 05/25/2023).

[29] Simen Bjorvand and Johannes Jäschke. "Improving Moving Horizon Estimation using Parametric Nonlinear Programming Sensitivity". In: *FRIPRO Project SensPATH* (2022), pp. 1–2.

APPENDICES

A - ORTHOGONAL COLLOCATION

In short, orthogonal collocation (OC) is a method used to numerically approximate solutions of differential equations. In order for the NMPC to calculate optimal inputs it has to have information regarding the states of the system. For the optimisation problem, the NMPC is provided the dynamic model which consist of a series of ODEs that have to be integrated in order to provide useful information. In this appendix the main ideas of orthogonal collocation are presented.

The main idea of orthogonal collocation is to divide the prediction horizon into finite elements, which again are further divided based off the number of collocation points that is to be used. For this project the Gauss-Radau collocation points, $t = [01151, 0.6449, 1.0000]$, are utilised as they remove the need to interpolate at the end of every finite element since the last collocation point is 1.0.

An example of ODEs that are to be integrated is:

$$M \begin{bmatrix} \dot{x}_1 \\ \dot{x}_2 \\ \dot{x}_3 \end{bmatrix} = \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} - \begin{bmatrix} x_0 \\ x_0 \\ x_0 \end{bmatrix}, \tag{1}$$

where M is the weighting matrix. The state trajectory can be approximated as a polynomial with Equation 1

$$x(t) \approx A + Bt + \frac{1}{2}Ct^2 + \frac{1}{3}Dt^3, \tag{2}$$

and the derivative $\dot{x}$ can be calculated by simply differentiating with the respect to t:

$$\dot{x}(t) \approx B + Ct + Dt^2. \tag{3}$$

An example of an approximation state trajectory, x, is illustrated in Figure A.1.

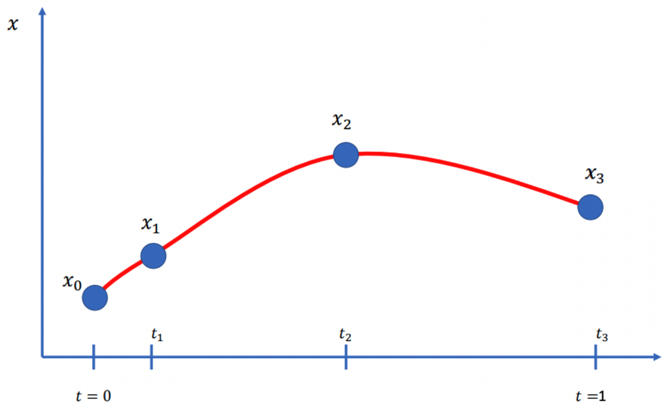


Figure A.1: An example of a polynomial approximated by orthogonal collocation

The goal is then to solve the aforementioned equations with respect to x. To do this the weighting matrix M first has to be calculated. Inserting Equation 2 and 3 into Equation 1 yields

$$M \begin{bmatrix} B + Ct_1 + Dt_1{}^2 \\ B + Ct_2 + Dt_2{}^2 \\ B + Ct_3 + Dt_3{}^2 \end{bmatrix} = \begin{bmatrix} A + Bt_1 + \frac{1}{2}Ct_1{}^2 + \frac{1}{3}Dt_1{}^3 \\ A + Bt_2 + \frac{1}{2}Ct_2{}^2 + \frac{1}{3}Dt_2{}^3 \\ A + Bt_3 + \frac{1}{2}Ct_3{}^2 + \frac{1}{3}Dt_3{}^3 \end{bmatrix} - \begin{bmatrix} x_0 \\ x_0 \\ x_0 \end{bmatrix} \tag{4}$$

If B,C, and D are factorised and the A is set equal to the initial condition, $A = x_0$, Equation 4 can be simplified to:

$$M \begin{bmatrix} 1 + t_1 + t_1{}^2 \\ 2 + t_2 + t_2{}^2 \\ 3 + t_3 + t_3{}^2 \end{bmatrix} \begin{bmatrix} B \\ C \\ D \end{bmatrix} = \begin{bmatrix} t_1 + \frac{1}{2}t_1{}^2 + \frac{1}{3}t_1{}^3 \\ t_2 + \frac{1}{2}t_2{}^2 + \frac{1}{3}t_2{}^3 \\ t_3 + \frac{1}{2}t_3{}^2 + \frac{1}{3}t_3{}^3 \end{bmatrix} \begin{bmatrix} B \\ C \\ D \end{bmatrix} \tag{5}$$

The weighting matrix M can then be calculated to be equal to:

$$M = \begin{bmatrix} t_1 + \frac{1}{2}t_1{}^2 + \frac{1}{3}t_1{}^3 \\ t_2 + \frac{1}{2}t_2{}^2 + \frac{1}{3}t_2{}^3 \\ t_3 + \frac{1}{2}t_3{}^2 + \frac{1}{3}t_3{}^3 \end{bmatrix} \begin{bmatrix} 1 + t_1 + t_1{}^2 \\ 2 + t_2 + t_2{}^2 \\ 3 + t_3 + t_3{}^2 \end{bmatrix}^{-1} \tag{6}$$

The weighting matrix is therefore calculated based on the position of the utilised collocation points. Finally, the state trajectory x can be approximated with M from Equation 6 inserted in to Equation 1.

$$\begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} = \begin{bmatrix} x_0 \\ x_0 \\ x_0 \end{bmatrix} + hM \begin{bmatrix} f(x1, u1, z1, p1) \\ f(x2, u2, z2, p2) \\ f(x3, u3, z3, p3) \end{bmatrix}, \tag{7}$$

where x represents the states, u the inputs, z the algebraic variables, and p the parameters of the process that is to be optimised. h is a scaling parameter that is utilised if the collocation points range is not between 0 and 1. This is however the case for Gauss-Radau collocation points and this parameter is therefore set to be $h = 1$.

B - CODE

.1 Plant Model: *Plant.jl*

```
using JuMP
using Ipopt
using DifferentialEquations
using Plots

# ------Define the model-----------

# Define given constants
α = 1700.0      # Heat transfer coefficient between reactor and jacket   [kJ/Khm
    ^2]
r = 0.092       # Radius of cross section inner part                     [m]
ρ = 1000.0      # Density of the reactor contents                        [g/L]
Cp = 4.2*1e-3   # Specific heat capacity of the reactor contents         [kJ/gK]
Cbin = 3.0      # Input concentration of reactant B                      [mol/L]
Vj = 2.22       # Volume of the contents inside the cooling jacket       [L]
Tin = 300       # Temperature of the flows entering the reactor          [K]

# Define initial conditions
Vr0 = 3.5                                                                # [L]
Ca0 = 2.0                                                                # [mol/
    L]
Cb0 = 0.0                                                                # [mol/
    L]
Cc0 = 0.0                                                                # [mol/
    L]
Tr0 = 325.0                                                              # [K]
Tj0 = 325.0                                                              # [K]
Aw0 = (π*(r^2))+((0.002*Vr0)/(r))                                        # [m^2]
x0 = [Vr0,Ca0,Cb0,Tr0,Tj0]

# ------Define dynamic model-----------
function dotStates(inputStates,p,t)
    Vr,Ca,Cb,Tr,Tj = inputStates
    u = p[1:2]
    par = p[3:4] # par[1] = K par[2] = H
    Aw = (π*(r^2))+((0.002*Vr)/(r))

    VrDot = u[1]
    CaDot = -((u[1]*Ca)/Vr) - (par[1]*Ca*Cb)
    CbDot = ((u[1]/Vr)*(Cbin-Cb)) - (par[1]*Ca*Cb)
    TrDot = ((u[1]/Vr)*(Tin-Tr)) - ((α*Aw*(Tr-Tj))/(ρ*Vr*Cp)) - ((par[1]*Ca*Cb*
        par[2])/(ρ*Cp))
    TjDot = (u[2] + α*Aw*(Tr-Tj))/(ρ*Vj*Cp)
    return [VrDot,CaDot,CbDot,TrDot,TjDot]
end

# ------Define the ODEsolver-----------
function ODEmodel(x,u,t)
    return dotStates(x,u,t)
end
```

```
function PlantODE(x0,u0,dt)
    tspan = (0.0,dt)
    function f(x,u,t)
        return ODEmodel(x,u,t)
    end

    prob = ODEProblem(f,x0,tspan,u0)
    sol = DifferentialEquations.solve(prob)
    x = sol.u
    xk = sol.u[end]
    return xk
end

# Since we can calculate Cc and Aw explicitly from the xstates we do not need to
     use a DAEsolver.
```

.2 Nonlinear Model Predictive Controller (NMPC): *NMPC.jl*

```
include("colMatrix.jl")

# Initialize the NMPC
#This function only builds the nlp as it lacks the sufficient info to solve
function nMPC0(u0List::Vector{Float64},x0List::Vector{Float64},z0List::Vector{
    Float64},pList::Vector{Float64},NFE)
    col = [0, 0.155051,0.644949,1.0] #colpoint 1 is the zeroth colpoint in this
        formulation

    colMat,colCont,colObj = colMatrix.collocationMatrix(col) #colMat is square
        but first row will be ignored

    # setting Timescale and number of steps

    dt = 0.05  #! [hours]
    Nx = 5 # States
    Nu = 2 # Inputs
    Nz = 2 # Algebraic equations
    Np = 2 # Parameters
    NCP = 3 # Collocation points

    # Define initial guesses
    uG = u0List
    xG = x0List
    zG = z0List

    optimizer = optimizer_with_attributes(Ipopt.Optimizer,
                "linear_solver" => "mumps",
                "print_level" => 1, # print_level => 5 output print. Set to 1 to
                     save CPU time
                "max_iter" => 1000,
                "tol" => 1e-8,
                "acceptable_tol" => 1e-8,
                "mu_init" => 10^(-1))
    nlp = Model(optimizer)

    #-------Declare variables----------

    # Lower and upper bounds on  u. Vin and Qk respectively
    ubl = [0.0, -9000.0]
    ubu = [32.4, 0.0]

    #Input variables
    # Declare input and corresponding bounds
    @variable(nlp, ubl[nu] <= u[nu=1:Nu,-1:NFE-1] <= ubu[nu])
```

```
# Declare operational state constraints
xbl = [0.0,0.0,0.0,273.0,273.0]
xbu = [8.0,5.0,5.0,350.0,350.0]

# State variables
# Declare states and corresponding bounds
@variable(nlp, xbl[nx] <= x[nx=1:Nx,1:NFE,0:NCP] <= xbu[nx])
#Declare xend for the collocation
@variable(nlp, xend[nx=1:Nx])

# Declare algebraic variables
@variable(nlp, 0.0 <= z[1:Nz,1:NFE,1:NCP])

# Declare slack variables
@variable(nlp, e1[1,1:NFE,0:NCP])
@variable(nlp, e2[1,1:NFE,0:NCP])

# Setting initial values
for nx in 1:Nx
    set_start_value.(x[nx,:,:],xG[nx])
end

for nu in 1:Nu
    set_start_value.(u[nu,:],uG[nu])
end

for nz in 1:Nz
    set_start_value.(z[nz,:,:],zG[nz])
end

# Parameter variables and initial state
# Declare parameters
@NLparameter(nlp, par[np=1:Np,1:NFE] == pList[np])
#Define initial states
@NLparameter(nlp, x0[nx=1:Nx] == x0List[nx])

## Model equations (Dynamic Model)
# State integration so collocation points are used
@NLconstraints(nlp, begin
    [nfe=1:NFE,ncp=1:NCP], sum(colMat[ncp+1,h+1]*x[1,nfe,h] for h in 0:NCP)
        - dt*((u[1,nfe-1])) == 0 # Vr
    [nfe=1:NFE,ncp=1:NCP], sum(colMat[ncp+1,h+1]*x[2,nfe,h] for h in 0:NCP)
        - dt*(-((u[1,nfe-1]*x[2,nfe,ncp])/x[1,nfe,ncp]) - (par[1,nfe]*x[2,
        nfe,ncp]*x[3,nfe,ncp])) == 0 # Ca
    [nfe=1:NFE,ncp=1:NCP], sum(colMat[ncp+1,h+1]*x[3,nfe,h] for h in 0:NCP)
        - dt*(((u[1,nfe-1]/(x[1,nfe,ncp]))*(Cbin-x[3,nfe,ncp])) - (par[1,
        nfe]*x[2,nfe,ncp]*x[3,nfe,ncp])) == 0 # Cb
    [nfe=1:NFE,ncp=1:NCP], sum(colMat[ncp+1,h+1]*x[4,nfe,h] for h in 0:NCP)
        - dt*(((u[1,nfe-1]/x[1,nfe,ncp])*(Tin-x[4,nfe,ncp])) - ((α*z[2,nfe,
        ncp]*(x[4,nfe,ncp]-x[5,nfe,ncp]))/(ρ*x[1,nfe,ncp]*Cp)) - ((par[1,
        nfe]*x[2,nfe,ncp]*x[3,nfe,ncp]*par[2,nfe])/(ρ*Cp))) == 0 # Tr
    [nfe=1:NFE,ncp=1:NCP], sum(colMat[ncp+1,h+1]*x[5,nfe,h] for h in 0:NCP)
        - dt*(((u[2,nfe-1]+(α*z[2,nfe,ncp]*(x[4,nfe,ncp]-x[5,nfe,ncp])))/(ρ
        *Vj*Cp))) == 0 # Tj
end)

# Model equations (Algebraic Model)
# The algebraic variables are dependent on the states, so collocation points
     has to be used
@NLconstraints(nlp,begin
    [nfe=1:NFE,ncp=1:NCP], z[1,nfe,ncp] - ((Ca0*Vr0 + Cc0*Vr0 - x[2,nfe,ncp]
        *x[1,nfe,ncp])/x[1,nfe,ncp]) == 0
    [nfe=1:NFE,ncp=1:NCP], z[2,nfe,ncp] - (π*(r^2) + ((0.002*x[1,nfe,ncp])/r
        )) == 0
end)

# Constraint 25c, 25d, 25e, 25f
@NLconstraint(nlp, [nfe=1:NFE,ncp=0:NCP], 322.0 <= x[4,nfe,ncp] + e1[1,nfe,
    ncp] <= 326.0) # 25c
@NLconstraint(nlp, [nfe=1:NFE,ncp=0:NCP], x[1,nfe,ncp] + e2[1,nfe,ncp] <= 7
    .0) # 25d
    @NLconstraint(nlp, [nfe=1:NFE,ncp=0:NCP], -1.0 <= e1[1,nfe,ncp] <= 1.0) # 25
        e
    @NLconstraint(nlp, [nfe=1:NFE,ncp=0:NCP], -1.0 <= e2[1,nfe,ncp] <= 1.0) # 25
        f

    # Collocation constraints
    @NLconstraint(nlp, [nx=1:Nx,nfe=2:NFE,ncp=0], x[nx,nfe,ncp] - sum(colCont[h+
        1]*x[nx,nfe-1,h] for h in 0:NCP) == 0)
    @NLconstraint(nlp, [nx=1:Nx,nfe=1,ncp=0], x[nx,nfe,ncp] - x0List[nx] == 0)
    @NLconstraint(nlp, [nx=1:Nx], xend[nx] - sum(colCont[h+1]*x[nx,NFE,h] for h
        in 0:NCP) == 0)

    # Defining initial u0 constraint
    @NLconstraint(nlp, [nu=1:Nu], u[nu,-1] - u0List[nu] == 0)

    # Defining the objective function to be minimized (negative production of Cc
        )
    # I adjusted the weighting of the regularisation terms here through trial
        and error! Literature: 0.0154*Deltau1 and 5.5e-5*Deltau2.
    @NLobjective(nlp, Min, -(NFE/1)*(z[1,NFE,NCP]*x[1,NFE,NCP])
                        + sum(0.0854*(u[1,nfe]-u[1,nfe-1])^2 for nfe in 0:NFE-1)
                        + sum(5.5e-5*(u[2,nfe]-u[2,nfe-1])^2 for nfe in 0:NFE-1)
                        + sum(sum(1e6*e1[1,nfe,ncp]^2 for ncp in 0:NCP) for nfe
                            in 1:NFE)
                        + sum(sum(1e10*e2[1,nfe,ncp]^2 for ncp in 0:NCP) for nfe
                             in 1:NFE))
    return nlp
end

function nMPC(u0List::Vector{Float64},x0List::Array{Float64,1},z0List::Vector{
    Float64},pList::Array{Float64,1},NFE)
    #NFE=20
    Nx = 5
    Nu = 2
    Nz = 2
    Np = 2
    NCP = 3
    nlp = nMPC0(u0List,x0List,z0List,pList,NFE)

    # Initial condition required for integration
    nlp.nlp_data.nlparamvalues[end-Nx+1:end] = x0

    ####Solving part:
    optimize!(nlp)

    ### Return 1 if optimal solution was found
    optFlag = termination_status(nlp) == JuMP.MOI.TerminationStatusCode(4)

    # Return first optimal u and flag that can be used to indicate if optimal
        solution was found
    return value.(nlp[:u][:,0]).data,optFlag
end
```

.3 Orthogonal collocation: *colMatrix.jl*

```julia
# This code was written and handed out by co-supervisor Simon Bjorvand
module colMatrix
    function polEval(pol,x)
        N = length(pol)
        return sum(x^(N-i)*pol[i] for i in 1:N)
    end

    function polDiff(pol)
        N = length(pol)
        diffPol = Array{typeof(pol[1])}(undef,N-1)
        for i = 1:N-1
            diffPol[i] = pol[i]*(N-i)
        end
        return diffPol
    end

    function polInte(pol)
        N = length(pol)
        intePol = Array{typeof(pol[1])}(undef,N+1)
        for i = 1:N
            intePol[i] = pol[i]/(N+1-i)
        end
        intePol[N+1] = 0
        return intePol
    end

    function lagrangeInterpolation(x,j)
        function polMult(pol1,pol2)
            N1 = length(pol1)
            N2 = length(pol2)

            typ = typeof(pol1[1])
            pol = zeros(typ,N1+N2-1)

            for i in 1:N2
                pol[i:(N1+i-1)] += pol1*pol2[i]
            end
            return pol
        end

        function lpol(x,i)
            N = length(x)
            typ = typeof(x[1])
            pol = ones(typ,1)
            for j in 1:N
                if j != i
                    pol = polMult(pol,[1;-x[j]])
                end
            end
            return pol
        end

        tempPol = lpol(x,j)
        return tempPol/polEval(tempPol,x[j])
    end


    function collocationMatrix(col_roots)
        K = length(col_roots)
        colMat = Array{Float64}(undef,K,K)
        colCont = Array{Float64}(undef,K)
        colObj = Array{Float64}(undef,K)


        for i = 1:K
            ltemp = lagrangeInterpolation(col_roots,i)
            dltemp = polDiff(ltemp)
            itemp = polInte(ltemp)
            for j = 1:K
                colMat[j,i] =  polEval(dltemp,col_roots[j])
            end
            colCont[i] = polEval(ltemp,1)
            colObj[i] = polEval(itemp,1)
        end


        return colMat,colCont,colObj
    end
end
```

.4 Functions for EKF: *Kalmanfunctions.jl*

```
using Distributions
using LinearAlgebra

include("plant.jl")

# Returns Jacobian of dynamic system
function dfdx(xk::Vector{Float64},up::Vector{Float64})
    # Need to define explicit Aw
    Aw = (π*(r^2))+((0.002*xk[1])/(r))

    # Define dfdx
    ddx1 = [0 0 0 0 0]
    ddx2 = [up[1]*xk[2]/xk[1]^2 -up[1]/xk[1]-up[3]*xk[3] -up[3]*xk[2] 0 0]
    ddx3 = [(-up[1]*(Cbin-xk[3]))/xk[1]^2 -up[3]*xk[3] -up[1]/xk[1]-up[3]*xk[1]
        0 0]
    ddx4 = [(-up[1]*(Tin-xk[4]))/(xk[1]^2)+(α*Aw*(xk[4]-xk[5])/ρ*(xk[1]^2)*Cp) -
        up[3]*xk[3]*up[4]/ρ*Cp -up[3]*xk[2]*up[4]/ρ*Cp (-up[1]/xk[1])-(α*Aw/ρ*
        xk[1]*Cp) (α*Aw/ρ*xk[1]*Cp)]
    ddx5 = [0 0 0 (α*Aw/ρ*Vj*Cp) -(α*Aw/ρ*Vj*Cp)]

    # Construct matrix
    dfdx = [ddx1;ddx2;ddx3;ddx4;ddx5]
    return dfdx
end

# Perform a priori guess based off previous estimate
function prediction(xekprev::Vector{Float64},up::Vector{Float64},Pekprev::Array{
    Float64})
    xek = PlantODE(xekprev,up,dt)     # Predict next xk based of previous
        estimate
    Ak = IM + dt*dfdx(xekprev,up)     # Define transition matrix A
    G = IM                            # Assumed addidative process noise so G=IM
    Pek = Ak*Pekprev*transpose(Ak) + G*Q*transpose(G)
    return xek,Pek
end

# Obtain measurement of current real states
function measurement(xk::Vector{Float64})
    return C*xk
end

# Correct a priori guess of states with newly obtained information from
    measurement
function correction(yk::Vector{Float64},xek::Vector{Float64},Pek::Matrix{Float64
    })
    KG = Pek*transpose(C)*inv(C*Pek*transpose(C)+R)  # Kalman Gain
    h = measurement(xek)                             # Measure predicted
        estimate
    ek = yk-h                                        # Innovation variable
    xck = xek + KG*ek                                # Perform a posteriori
        correction
    Pck = (IM-KG*C)*Pek                              # Corrected Covariance
        matrix Pk
    return xck,Pck
end

# Obtain corrected x and Pk values with measurement yk, previous estimate
    xekprev and covariance Pekprev
function Kalman(yk::Vector{Float64},xekprev::Vector{Float64},up::Vector{Float64}
    ,Pekprev::Array{Float64})
    xek,Pek = prediction(xekprev,up,Pekprev)
    xck,Pck = correction(yk,xek,Pek)
    return xck,Pck
end
```

.5 Functions for MHE: *MHEfunctions.jl*

```
include("plant.jl")

using Distributions
using LinearAlgebra

# Define unique MHE function for the initial iteration
function MHEinit(x0List::Vector{Float64},ykList::Array{Float64},Pk0::Matrix{
    Float64})

    Nx = 5 # Process states
    Ny = 3 # Measurable states

    optimizer = optimizer_with_attributes(Ipopt.Optimizer,
                "linear_solver" => "mumps",
                "print_level" => 1,
                "max_iter" => 1000,
                "tol" => 1e-8,
                "acceptable_tol" => 1e-8,
                "mu_init" => 10^(-1))
    #optimizer = with_optimizer(Ipopt.Optimizer, linear_solver = "mumps")
    nlp = Model(optimizer)

    #-------Declare variables----------

    # Only need physical constraints, not operational
    xbl = [0.0,0.0,0.0,0,0]

    # State variables
    @variable(nlp, x[nx=1:Nx] >= xbl[nx])

    # Measurement noise variables
    @variable(nlp, v[nx=1:Ny])

    # Measurement constraints
    @NLconstraints(nlp, begin
        ykList[1]-x[1]-v[1]  == 0
        ykList[2]-x[4]-v[2]  == 0
        ykList[3]-x[5]-v[3]  == 0
    end)

    # Approximate initial arrival cost
    AC01 = transpose(x)*inv(Pk0)*x
    AC02 = -2*transpose(x0List)*inv(Pk0)*x
    AC0 = AC01+AC02

    objfunc = transpose(v)*inv(R)*v + AC0

    # Define the objective function that is to be minimised
    @NLobjective(nlp, Min, objfunc)
    return nlp
```

```
end

function MHEinit2(x0::Vector{Float64},ykList::Array{Float64},Pk0::Matrix{Float64
    })

    nlp = MHEinit(x0,ykList,Pk0)

    ####Solving part:
    optimize!(nlp)

    ### Return 1 if optimal solution was found
    optFlag = termination_status(nlp) == JuMP.MOI.TerminationStatusCode(4)


    # Return first optimal u and flag that can be used to indicate if optimal
        solution was found
    return value.(nlp[:x]),optFlag
end

# Build MHE framework
function MHE0(x0List::Vector{Float64},uwindow::Vector{Vector{Float64}},pList::
    Vector{Float64},T::Int64,N::Int64,xkcheck::Vector{Float64},ykList::Vector{
    Vector{Float64}},Pkcheck::Matrix{Float64})
    col = [0, 0.155051,0.644949,1.0] #colpoint 1 is the zeroth colpoint in this
        formulation

    colMat,colCont,colObj = colMatrix.collocationMatrix(col) #colMat is square
        but first row will be ignored

    # Setting Timescale and number of steps
    dt = 0.05                                       #! [hours]
    Nx = 5                                          # Process states
    Ny = 3                                          # Measurable states
    Nz = 2                                          # Algebraic equations
    Np = 2                                          # Parameters
    NCP = 3                                         # Collocation points
    wl = min(T,N)                                   # Define window length wl based of
         T and N

    # Define initial guesses
    xG = x0List

    # Define optimizer settings
    optimizer = optimizer_with_attributes(Ipopt.Optimizer,
                "linear_solver" => "mumps",
                "print_level" => 1,
                "max_iter" => 1000,
                "tol" => 1e-8,
                "acceptable_tol" => 1e-8,
                "mu_init" => 10^(-1))
    #optimizer = with_optimizer(Ipopt.Optimizer, linear_solver = "mumps")
    nlp = Model(optimizer)

    #-------Declare variables---------#

    # Only need physical, not operational constraints,
    xbl = [0.0,0.0,0.0,0.0,0.0]
    xbu = [8.0,Inf,Inf,Inf,Inf]

    # Process noise variable constraints. Chosen to be 5 times standard
        deviation
    ωbl = [-0.05,-0.25,-0.25,-2.50,-2.50]
    ωbu = [0.05,0.25,0.25,2.50,2.50]

    # Measurement noise variable constraints. Chosen to be 5 times standard
        deviation
    vbl = [-2.0,-5.0,-5.0]
    vbu = [2.0,5.0,5.0]

    # State variables
    # Declare states and corresponding bounds
    @variable(nlp, xbl[nx] <= x[nx=1:Nx,1:wl-1,0:NCP] <= xbu[nx])
    #Declare xend for the collocation
```

```
@variable(nlp, xbl[nx] <= xend[nx=1:Nx] <= xbu[nx])

# Process noise variables
@variable(nlp,start=0.0, ωbl[nx] <= ω[nx=1:Nx,1:wl-1] <= ωbu[nx])

# Measurement noise variables
@variable(nlp,start=0.0, vbl[nx] <= v[nx=1:Ny,1:wl] <= vbu[nx])

# Algebraic variables
@variable(nlp, 0.0 <= z[1:Nz,1:wl-1,1:NCP]) # Declare algebraic states

# Setting initial values
for nx in 1:Nx
    set_start_value.(x[nx,:,:],xG[nx])
end

# Parameter variables and initial state
@NLparameter(nlp, par[np=1:Np,1:wl-1] == pList[np]) # Declare parameters


## Model equations (Dynamic Model)
# States are to be integrated so so collocation points are used
@NLconstraints(nlp, begin
    [nfe=1:wl-1,ncp=1:NCP], sum(colMat[ncp+1,h+1]*x[1,nfe,h] for h in 0:NCP)
        - dt*((uwindow[nfe][1])) == 0 # Vr
    [nfe=1:wl-1,ncp=1:NCP], sum(colMat[ncp+1,h+1]*x[2,nfe,h] for h in 0:NCP)
        - dt*(-((uwindow[nfe][1]*x[2,nfe,ncp])/x[1,nfe,ncp]) - (par[1,nfe]
        *x[2,nfe,ncp]*x[3,nfe,ncp])) == 0 # Ca
    [nfe=1:wl-1,ncp=1:NCP], sum(colMat[ncp+1,h+1]*x[3,nfe,h] for h in 0:NCP)
        - dt*(((uwindow[nfe][1]/(x[1,nfe,ncp]))*(Cbin-x[3,nfe,ncp])) - (
        par[1,nfe]*x[2,nfe,ncp]*x[3,nfe,ncp])) == 0 # Cb
    [nfe=1:wl-1,ncp=1:NCP], sum(colMat[ncp+1,h+1]*x[4,nfe,h] for h in 0:NCP)
        - dt*(((uwindow[nfe][1]/x[1,nfe,ncp])*(Tin-x[4,nfe,ncp])) - ((α*z[
        2,nfe,ncp]*(x[4,nfe,ncp]-x[5,nfe,ncp]))/(ρ*x[1,nfe,ncp]*Cp)) - ((
        par[1,nfe]*x[2,nfe,ncp]*x[3,nfe,ncp]*par[2,nfe])/(ρ*Cp))) == 0 # Tr
    [nfe=1:wl-1,ncp=1:NCP], sum(colMat[ncp+1,h+1]*x[5,nfe,h] for h in 0:NCP)
        - dt*(((uwindow[nfe][2]+(α*z[2,nfe,ncp]*(x[4,nfe,ncp]-x[5,nfe,ncp]
        )))/(ρ*Vj*Cp))) == 0 # Tj
end)

# Model equations (Algebraic Model)
# The algebraic variables are dependent on the states, so collocation points
     have to be used
@NLconstraints(nlp,begin
    [nfe=1:wl-1,ncp=1:NCP], z[1,nfe,ncp] - ((Ca0*Vr0 + Cc0*Vr0 - x[2,nfe,ncp
        ]*x[1,nfe,ncp])/x[1,nfe,ncp]) == 0
    [nfe=1:wl-1,ncp=1:NCP], z[2,nfe,ncp] - (π*(r^2) + ((0.002*x[1,nfe,ncp])/
        r)) == 0
end)

# Collocation constraints
@NLconstraint(nlp, [nx=1:Nx,nfe=2:wl-1,ncp=0], x[nx,nfe,ncp] - sum(colCont[h
    +1]*x[nx,nfe-1,h] for h in 0:NCP) - ω[nx,nfe-1] == 0)
# Here we implement the process noise ω in the xk+1=F(xk,uk) constraint.
@NLconstraint(nlp, [nx=1:Nx], xend[nx] - sum(colCont[h+1]*x[nx,wl-1,h] for h
     in 0:NCP) - ω[nx,wl-1] == 0)

# Constraint for measurements yk
@NLconstraints(nlp, begin
    [nfe=1:wl-1], ykList[nfe][1]-x[1,nfe,0]-v[1,nfe] == 0
    [nfe=1:wl-1], ykList[nfe][2]-x[4,nfe,0]-v[2,nfe] == 0
    [nfe=1:wl-1], ykList[nfe][3]-x[5,nfe,0]-v[3,nfe] == 0
end)

# Constraint for xend for measurement yk
@NLconstraints(nlp, begin
    ykList[wl][1]-xend[1]-v[1,wl] == 0
    ykList[wl][2]-xend[4]-v[2,wl] == 0
    ykList[wl][3]-xend[5]-v[3,wl] == 0
end)

# The states rewritten this specific way for the arrival cost AC
```

```
    xk = [x[1:Nx,1,0][1];x[1:Nx,1,0][2];x[1:Nx,1,0][3];x[1:Nx,1,0][4];x[1:Nx,1,0
        ][5]]

    # Define arrival cost AC
    AC1 = transpose(xk)*inv(Pkcheck)*xk
    AC2 = -2*transpose(xkcheck)*inv(Pkcheck)*xk
    AC = AC1+AC2

    term1 = sum(diag(transpose(ω[1:Nx,1:wl-1])*inv(Q)*ω[1:Nx,1:wl-1]))
    term2 = sum(diag(transpose(v[1:Ny,1:wl])*inv(R)*v[1:Ny,1:wl])) + AC

    # Defining the objective function
    objfunc = term1+term2

    @NLobjective(nlp, Min, objfunc)

    return nlp
end

# Perform optimisation that yield state estimate using the MHE framework
function MHE(x0::Vector{Float64},uwindow::Vector{Vector{Float64}},pList::Vector{
    Float64},T::Int64,N::Int64,xkcheck::Vector{Float64},ykList::Vector{Vector{
    Float64}},Pkcheck::Matrix{Float64})

    nlp = MHE0(x0,uwindow,pList,T,N,xkcheck,ykList,Pkcheck)

    ####Solving part:
    optimize!(nlp)

    ### Return 1 if optimal solution was found
    optFlag = termination_status(nlp) == JuMP.MOI.TerminationStatusCode(4)

    # Return first optimal u and flag that can be used to indicate if optimal
        solution was found
    return value.(nlp[:xend]),optFlag
end

# Obtain measurement of current states
function measurement(xk::Vector{Float64})
    return C*xk
end

# Linearizes the produced state estimate around dynamic model. Used for updating
     error estimate covariance matrix Pk
function lin1(xk::Vector{Float64},up::Vector{Float64})
    # Need this defined
    Aw = (π*(r^2))+((0.002*xk[1])/(r))

    # Define dfdx
    ddx1 = [0 0 0 0 0]
    ddx2 = [up[1]*xk[2]/xk[1]^2 -up[1]/xk[1]-up[3]*xk[3] -up[3]*xk[2] 0 0]
    ddx3 = [(-up[1]*(Cbin-xk[3]))/xk[1]^2 -up[3]*xk[3] -up[1]/xk[1]-up[3]*xk[1]
        0 0]
    ddx4 = [(-up[1]*(Tin-xk[4]))/(xk[1]^2)+(α*Aw*(xk[4]-xk[5])/ρ*(xk[1]^2)*Cp) -
        up[3]*xk[3]*up[4]/ρ*Cp -up[3]*xk[2]*up[4]/ρ*Cp (-up[1]/xk[1])-(α*Aw/ρ*
        xk[1]*Cp) (α*Aw/ρ*xk[1]*Cp)]
    ddx5 = [0 0 0 (α*Aw/ρ*Vj*Cp) -(α*Aw/ρ*Vj*Cp)]

    dfdx = [ddx1;ddx2;ddx3;ddx4;ddx5]
    return dfdx
end

# Update state estimation error covariance matrix Pk using Ricatti equation
function ricatti(Ak::Matrix{Float64},Pkprev::Matrix{Float64},Ck::Matrix{Int64},
    Rk::Matrix{Float64},Qk::Matrix{Float64})
    Pk = Ak*(Pkprev-(Pkprev*transpose(Ck)*inv(Ck*Pkprev*transpose(Ck)+Rk)*Ck*
        Pkprev))*transpose(Ak) + Qk
    return Pk
end
```

.6 Code for obtaining EKF results: *resultsEKF.jl*

```julia
include("plant.jl")
include("colMatrix.jl")
include("NMPC.jl")

using Random
using Statistics
using ProgressBars

# Define global variables
# Step length
dt = 0.05
# Intial input
u0 = [0.0,0.0]
# Initial state guess x0
Vr0,Ca0,Cb0,Tr0,Tj0 = [3.5,2.0,0.0,325.0,325.0]
x0 = [Vr0,Ca0,Cb0,Tr0,Tj0]
# Initial DAE guess z0
z0 = [Cc0,Aw0]
# Fixed parameters K and H
p0 = [1.205,-355.0]
# Length of NMPC prediction horizon
NFE0 = 20

# Noise
#Define process and measurement noise. Change these for Case stuudy 9 and 10
# Standard deviation for process noise. Chosen myself
σω = [0.01,0.05,0.05,0.5,0.5]
# Standard deviation for measurement noise. Chosen myself
σv = [0.4,1.0,1.0]
# Assumed mean of Gaussian distributions
μ = 0.0

# Define required matrices
# Measurement Gain matrix. A,B immeasurable
C = [1 0 0 0 0; 0 0 0 1 0; 0 0 0 0 1]

# Identity Matrix
IM = diagm([1.0,1.0,1.0,1.0,1.0])

# Process noise gain Matrix. Process noise is assumed addidative so G=IM
G = IM

# Process noise auto covariance. Assumed no correlation
Q = diagm([σω[1]^2,σω[2]^2,σω[3]^2,σω[4]^2,σω[5]^2])

# Measurement noise auto covariance. Assumed no correlation
R = diagm([σv[1]^2,σv[2]^2,σv[3]^2])

# Initial guess of state estimation covariance error matrix.
# The initial guess x0 is known so Pk0=Q
Pk0 = Q

# Define list that will store the results:
kxekresultsList = Array{Array{Float64,2}}(undef,0)
kxkrresultsList = Array{Array{Float64,2}}(undef,0)
kykresultsList = Array{Array{Float64,2}}(undef,0)
kzkrresultsList = Array{Array{Float64,2}}(undef,0)
kzekresultsList = Array{Array{Float64,2}}(undef,0)


# Obtain results from 100 seeds
for j in ProgressBar(1:100)
    # Generate a specific seed so that the random noise follow the same random
        distribution
    Random.seed!(j)

    # Define global list/matrices that are continously updated for each
        iteration
    xkrList = Array{Array{Float64,1}}(undef,0) # Real x
    xekList = Array{Array{Float64,1}}(undef,0) # Estimated x
    zrList = Array{Array{Float64,1}}(undef,0)  # Real DAEs
    zeList = Array{Array{Float64,1}}(undef,0)  # Estimated DAEs
    PkList = Array{Array{Float64,2}}(undef,0)  # For storing covariance matrices
    ukList = Array{Array{Float64,1}}(undef,0)  # Store inputs
    ykList = Array{Array{Float64,1}}(undef,0)  # Store inputs
    solveList = Array{Bool,1}(undef,0)         # Store flags from optimisation

    for i in 1:NFE0
        NFE = min(20,NFE0-i+1)
        if i == 1 # Initialization
            # Generate random Gaussian distributed process noise
            vrpn = rand(Normal(μ,σω[1]))
            capn = rand(Normal(μ,σω[2]))
            cbpn = rand(Normal(μ,σω[3]))
            trpn = rand(Normal(μ,σω[4]))
            tjpn = rand(Normal(μ,σω[5]))
            ω = [vrpn,capn,cbpn,trpn,tjpn]

            # Generate random Gaussian distributed measurement noise
            vrmn = rand(Normal(μ,σv[1]))
            trmn = rand(Normal(μ,σv[2]))
            tjmn = rand(Normal(μ,σv[3]))
            v = [vrmn,trmn,tjmn]

            # For initialisation, predict what the current state is
            xk0 = x0 + ω
            # Also predict the initial covariance matrix Pk
            Pkprev = Q

            # Implement check here to ensure initial prediction do not result in
                 negative DAE values
            # If negative, these can result in infeasible solutions from NMPC
            if xk0[1] > Vr0
                xk0[1] = Vr0
            end
            if xk0[2] > Ca0
                xk0[2] = Ca0
            end
            if xk0[3] < Cb0
                xk0[3] = Cb0
            end

            # Initial estimate DAE
            Ccr0 = (Ca0*Vr0 + Cc0*Vr0 - xk0[2]*xk0[1])/(xk0[1])
            Awr0 = (π*(r^2))+((0.002*xk0[1])/(r))
            zr0 = [Ccr0,Awr0]

            # Need this for initial point when plotting results
            push!(ukList, u0)
            push!(xkrList,x0)
            push!(zrList,zr0)
            push!(zeList,zr0)

            # Measurement of initial real state
            yk0 = measurement(xk0) + v
            push!(ykList,yk0)

            # Obtain corrected estimate.
            # Since we have done prediction already, for initialisation we only
                need to use the correction function.
            xck,Pck = correction(yk0,xk0,Pkprev)

            # Implement checks on corrected estimate as well
            if xck[1] > Vr0
                xck[1] = Vr0
            end
            if xck[2] > Ca0
                xck[2] = Ca0
            end
            if xck[3] < Cb0
                xck[3] = Cb0
            end

            # Calculate inital estimated DAEs for plotting
            Cce0 = (Ca0*Vr0 + Cc0*Vr0 - xck[2]*xck[1])/(xck[1])
            Awe0 = (π*(r^2))+((0.002*xck[1])/(r))
            ze0 = [Cce0,Awe0]

            # Store covariance
            push!(PkList,Pck)
            # Store estimate
            push!(xekList,xck)
            # Store estimated DAEs
            push!(zeList,ze0)

            # Perform the NMPC with the corrected estimate
            uk,optFlag = nMPC(u0,xck,ze0,p0,NFE)
            print("Iteration: ",i,", Solved to optimailty: ",optFlag)
            # Store flags for optimality
            push!(solveList,optFlag)
            # Store inputs
            push!(ukList,uk)

            # Inject uk into plant and obtain optimal states
            ukp = [uk;p0]
            # Obtain real x
            xkr = PlantODE(x0,ukp,dt) + ω

            # Calculate real DAEs for plotting
            Ccr = (Ca0*Vr0 + Cc0*Vr0 - xkr[2]*xkr[1])/(xkr[1])
            # Check here as well, as inital negative Cc does not make physical
                sense
            if Ccr < 0
                Ccr = 0
            end

            Awr = (π*(r^2))+((0.002*xkr[1])/(r))
            zr = [Ccr,Awr]

            # Store real x
            push!(xkrList, xkr)
            # Store real DAEs
            push!(zrList,zr)

        else
            # Generate random normal-distributed process noise
            vrpn = rand(Normal(μ,σω[1]))
            capn = rand(Normal(μ,σω[2]))
            cbpn = rand(Normal(μ,σω[3]))
            trpn = rand(Normal(μ,σω[4]))
            tjpn = rand(Normal(μ,σω[5]))
            ω = [vrpn,capn,cbpn,trpn,tjpn]

            # Generate random normal-distributed measurement noise
            vrmn = rand(Normal(μ,σv[1]))
            trmn = rand(Normal(μ,σv[2]))
            tjmn = rand(Normal(μ,σv[3]))
            v = [vrmn,trmn,tjmn]

            # Obtain input, previous xek,xkr,Pk
            # Known input
            ukk = ukList[i]
            # Build ukp argument
            ukkp = [ukk;p0]
            # Previous covariance matrix
            Pkprev = PkList[i-1]
            # Previous estimate
            xekprev = xekList[i-1]
            # Real state
            xkrprev = xkrList[i]
            # Current measurement
            yk = measurement(xkrprev) + v
            push!(ykList,yk)
```

```julia
            # Obtain estimated states and co-variance matrix
            xek,Pk = Kalman(yk,xekprev,ukkp,Pkprev)

            # Calculate estimated DAEs for plotting
            Cce = (Ca0*Vr0 + Cc0*Vr0 - xek[2]*xek[1])/(xek[1])
            Awe = (π*(r^2))+((0.002*xek[1])/(r))
            ze = [Cce,Awe]

            # Store covariance
            push!(PkList,Pk)
            # Store estimate
            push!(xekList,xek)
            # Store estimated DAEs
            push!(zeList,ze)

            # Perform the NMPC with the estimate
            uk,optFlag = nMPC(ukk,xek,ze,p0,NFE)
            print("Iteration: ",i,", Solved to optimailty: ",optFlag)
            # Store flags for optimality
            push!(solveList,optFlag)
            # Store optimal inputs
            push!(ukList,uk)

            # Inject uk into plant and obtain optimal states
            ukp = [uk;p0]
            # Obtain real x
            xkr = PlantODE(xkrprev,ukp,dt) + ω

            # Calculate real DAEs for plotting
            Ccr = (Ca0*Vr0 + Cc0*Vr0 - xkr[2]*xkr[1])/(xkr[1])
            # Check here as well, as negative Cc does not make physical sense
                when plotting
            if Ccr < 0
                Ccr = 0
            end
            Awr = (π*(r^2))+((0.002*xkr[1])/(r))
            zr = [Ccr,Awr]

            # Store real x
            push!(xkrList, xkr)
            # Store real DAEs
            push!(zrList,zr)

            # Need to obtain final values for plotting
            if i == NFE0
                xkr = xkrList[end]
                yk = measurement(xkr) + v
                push!(ykList,yk)

                Pkprev = PkList[i-1]                # Previous covariance matrix
                xekprev = xekList[i-1]                # Previous estimate
                ukk = ukList[i]                     # Known input
                ukkp = [ukk;p0]                     # Build ukp argument
                # Obtain estimated states and co-variance matrix
                xek,Pk = Kalman(yk,xekprev,ukkp,Pkprev)
                # Calculate estimated DAEs for plotting
                Cce = (Ca0*Vr0 + Cc0*Vr0 - xek[2]*xek[1])/(xek[1])
                Awe = (π*(r^2))+((0.002*xek[1])/(r))
                ze = [Cce,Awe]

                push!(PkList,Pk)                            # Store covariance
                push!(xekList,xek)                          # Store estimate
                push!(zeList,ze)                            # Store estimated
                    DAEs

                # Perform the NMPC with the estimate
                uk,optFlag = nMPC(ukk,xek,ze,p0,NFE)
                print("Iteration: ",i,", Solved to optimailty: ",optFlag)
                push!(solveList,optFlag)                    # Store flags for
                    optimality
                push!(ukList,uk)                            # Store inputs

                # Inject uk into plant and obtain optimal states
                ukp = [uk;p0]
                xkr = PlantODE(xkrprev,ukp,dt) + ω            # Obtain next x (
                    or real)

                # Calculate real DAEs for plotting
                Ccr = (Ca0*Vr0 + Cc0*Vr0 - xkr[2]*xkr[1])/(xkr[1])
                # Check here as well, as negative Cc does not make physical
                    sense when plotting
                if Ccr < 0
                    Ccr = 0
                end
                Awr = (π*(r^2))+((0.002*xkr[1])/(r))
                zr = [Ccr,Awr]

                push!(xkrList, xkr)                     # Store "real" or
                    optimal x
                push!(zrList,zr)                        # Store real DAEs
            end
        end
    end

    # Have to use hcat to convert Vector{Vector} to Matrix that can be pushed to
         an array
    xekresult = hcat(xekList...)
    xkresult = hcat(xkrList...)
    ykresult = hcat(ykList...)
    zeresult = hcat(zeList...)
    zrresult = hcat(zrList...)
    # Store all results in a matrix
    push!(kxekresultsList,xekresult)
    push!(kxkrresultsList,xkresult)
    push!(kykresultsList,ykresult)
    push!(kzekresultsList,zeresult)
    push!(kzkrresultsList,zrresult)
end

# Plotting EKF trajectories
# Calculate and unpack function can be found in results.jl
# Obtain real, measured and estimated states from result matrix
kvrr,kcar,kcbr,ktrr,ktjr,kccr= calculateμσ(kxkrresultsList)[1]
kvrm, ktrm, ktjm = unpackyk(kykresultsList)
kvre, kcae, kcbe, ktre,ktje,kcce = calculateμσ(kxekresultsList)[1]

# Plotting settings
styles1 = [:solid :dash :dash]
styles2 = [:solid :dash]
colors1 = [:blue :red :green]
colors2 = [:blue :red]
size = [2 2 3]

# Perform EKF plotting
plot(xaxis, [kvrr kvre kvrm], line=(size,styles1),color=(colors1), title="
    Reactor volume Vr", xlabel = "Time [h]",ylabel = "Volume [L]",label=["Real"
     "Estimated" "Measured"],legendtitle="State type (EKF)",legend=:right,
    tickfontsize=16,guidefontsize=16,titlefontsize=25, legendfontsize=15)
plot(xaxis, [kcar kcae], line=(size,styles2),color=(colors2), title = "
    Concentration of A", xlabel = "Time [h]",ylabel = "Concentration [mol/L]",
    label=["Real" "Estimated"],legendtitle="State type (EKF)",legend=:right,
    tickfontsize=16,guidefontsize=16,titlefontsize=25, legendfontsize=15)
plot(xaxis, [kcbr kcbe], line=(size,styles2),color=(colors2), title = "
    Concentration of B", xlabel = "Time [h]",ylabel = "Concentration [mol/L]",
    label=["Real" "Estimated"],legendtitle="State type (EKF)",legend=:right,
    tickfontsize=16,guidefontsize=16,titlefontsize=25, legendfontsize=15)
plot(xaxis, [ktrr ktre ktrm], line=(size,styles1),color=(colors1),title = "
    Reactor Temperature Tr", xlabel = "Time [h]",ylabel = "Temperature [K]",
    label=["Real" "Estimated" "Measured"],legendtitle="State type (EKF)",legend
    =:topright,tickfontsize=16,guidefontsize=16,titlefontsize=25,
    legendfontsize=10)
plot(xaxis, [ktjr ktje ktjm], line=(size,styles1),color=(colors1),title = "
    Jacket Temperature Tj", xlabel = "Time [h]",ylabel = "Temperature [K]" ,
    label=["Real" "Estimated" "Measured"],legendtitle="State type (EKF)",legend
    =:topright,tickfontsize=16,guidefontsize=16,titlefontsize=25,
    legendfontsize=15)
```

```
plot(xaxis, [kccr kcce],line=(size,styles2),color=(colors2), title = "
    Concentration of C", xlabel = "Time [h]",ylabel = "Concentration [mol/L]",
    label=["Real" "Estimated"],legendtitle="State type (EKF)",legend=:
    bottomright,tickfontsize=16,guidefontsize=16,titlefontsize=25,
    legendfontsize=15)
```

.7 Code for obtaining MHE results: *resultsMHE.jl*

```
include("plant.jl")
include("colMatrix.jl")
include("NMPC.jl")

using Random
using Statistics
using ProgressBars

# Define initial conditions
dt = 0.05
u0 = [0.0,0.0]

# Initial guess
Vr0,Ca0,Cb0,Tr0,Tj0 = [3.5,2.0,0.0,325.0,325.0]
x0 = [Vr0,Ca0,Cb0,Tr0,Tj0]

# Initial DAE guess
z0 = [Cc0,Aw0]
p0 = [1.205,-355.0]
# Length of NMPC prediction horizon
NFE0 = 20

# Noise
#Define process and measurement noise
# Standard deviation for process noise. Chosen myself
σω = [0.01,0.05,0.05,0.5,0.5]
# Standard deviation for measurement noise. Chosen myself
σv = [0.4,1.0,1.0]
# Assumed mean of Gaussian distributions
μ = 0.0

# Define required matrices
# Measurement Gain matrix. A and B are immeasurable
C = [1 0 0 0 0; 0 0 0 1 0; 0 0 0 0 1]
# Identity Matrix
IM = diagm([1.0,1.0,1.0,1.0,1.0])
# Process Noise Gain Matrix. Process noise is assumed to be addidative so G=IM
G = IM
# Process noise auto covariance. Assume no correlation
Q = diagm([σω[1]^2,σω[2]^2,σω[3]^2,σω[4]^2,σω[5]^2])
# Measurement noise auto covariance. Assume no correlation
R = diagm([σv[1]^2,σv[2]^2,σv[3]^2])

# Initial guess of state estimation covariance error matrix.
# The initial guess x0 is known so Pk0=Q
Pk0 = Q
# Length of MHE estimation window (BASE CASE)
N = 10

# Define list that are to store the results:
xekresultsList = Array{Array{Float64,2}}(undef,0)
xkrresultsList = Array{Array{Float64,2}}(undef,0)
ykresultsList = Array{Array{Float64,2}}(undef,0)
zkrresultsList = Array{Array{Float64,2}}(undef,0)
zekresultsList = Array{Array{Float64,2}}(undef,0)

for j in ProgressBar(1:100)
```

```
# Generate a specific seed so that the random noise follow the same
    distribution
Random.seed!(j)

# Define global list/matrices that are continously updated for each
    iteration
xkrList = Array{Array{Float64,1}}(undef,0) # Real x
xekList = Array{Array{Float64,1}}(undef,0) # Estimated x
zeList = Array{Array{Float64,1}}(undef,0)  # Estimated DAEs
zrList = Array{Array{Float64,1}}(undef,0)  # Real DAEs
PkList = Array{Array{Float64,2}}(undef,0)  # For storing covariance matrices
ukList = Array{Array{Float64,1}}(undef,0)  # Store inputs
ykList = Array{Array{Float64,1}}(undef,0)  # Store inputs
solveList1 = Array{Bool,1}(undef,0)        # Information if optimal solution
     for MHE was found
solveList2 = Array{Bool,1}(undef,0)        # Information if optimal solution
     for NMPC was found

for i in 1:NFE0
    # Change first argument here to change length of NMPC horizon (Case 3
        and 4)
    NFE = min(20,NFE0-i+1)
    if i == 1 # Initialisation
        T = i
        # Generate random normal-distributed process noise
        vrpn = rand(Normal(μ,σω[1]))
        capn = rand(Normal(μ,σω[2]))
        cbpn = rand(Normal(μ,σω[3]))
        trpn = rand(Normal(μ,σω[4]))
        tjpn = rand(Normal(μ,σω[5]))
        ω = [vrpn,capn,cbpn,trpn,tjpn]

        # Generate random normal-distributed measurement noise
        vrmn = rand(Normal(μ,σv[1]))
        trmn = rand(Normal(μ,σv[2]))
        tjmn = rand(Normal(μ,σv[3]))
        v = [vrmn,trmn,tjmn]

        # Predict what the initial state is
        xk0 = x0 + ω

        #Implement check here to ensure initial prediction do not result in
            negative physical values, most importantly Cc and Cb
        if xk0[1] > Vr0
            xk0[1] = Vr0
        end
        if xk0[2] > Ca0
            xk0[2] = Ca0
        end
        if xk0[3] < Cb0
            xk0[3] = Cb0
        end

        # Calculate initial estimate DAE
        Ccr0 = (Ca0*Vr0 + Cc0*Vr0 - xk0[2]*xk0[1])/(xk0[1])
        Awr0 = (π*(rˆ2))+((0.002*xk0[1])/(r))
        ze0 = [Ccr0,Awr0]

        # Calculate initial real DAE
        Ccr0 = (Ca0*Vr0 + Cc0*Vr0 - x0[2]*x0[1])/(x0[1])
        Awr0 = (π*(rˆ2))+((0.002*x0[1])/(r))
        zr0 = [Ccr0,Awr0]

        # Need this for initial point in plotting
        push!(xkrList,x0)
        push!(zrList,zr0)
        push!(zeList,ze0)

        # Obtain measurement of x
        yk0 = measurement(x0) + v
        push!(ykList,yk0)

        # Obtain estimate based off previous estimates
```

```
            xek,optFlag1 = MHEinit2(xk0,yk0,Pk0)
            print("Iteration: ",i,", Solved to optimailty: ",optFlag1)
            push!(solveList1,optFlag1)

            # Checks here as well to make sure the estimate is not negative
            # Negative arguments to the NMPC can result in infeasible solutions
            if xek[1] > Vr0
                xek[1] = Vr0
            end
            if xek[2] > Ca0
                xek[2] = Ca0
            end
            if xek[3] < Cb0
                xek[3] = Cb0
            end

            # Calculate estimated DAEs for plotting
            Cce0 = (Ca0*Vr0 + Cc0*Vr0 - xek[2]*xek[1])/(xek[1])
            Awe0 = (π*(r^2))+((0.002*xek[1])/(r))
            ze0 = [Cce0,Awe0]

            # Store estimate
            push!(xekList,xek)
            # Store estimated DAEs
            push!(zeList,ze0)

            # Perform the NMPC with the estimate
            uk,optFlag2 = nMPC(u0,xek,ze0,p0,NFE)
            print("Iteration: ",i,", Solved to optimailty: ",optFlag2)
            # Store flags for optimality
            push!(solveList2,optFlag2)
            # Store optimal inputs
            push!(ukList,uk)

            # Inject uk into plant and obtain optimal states
            ukp = [uk;p0]
            # Geneerate new noise for real state
            vrpn = rand(Normal(μ,σω[1]))
            capn = rand(Normal(μ,σω[2]))
            cbpn = rand(Normal(μ,σω[3]))
            trpn = rand(Normal(μ,σω[4]))
            tjpn = rand(Normal(μ,σω[5]))
            ω = [vrpn,capn,cbpn,trpn,tjpn]

            # Obtain real x
            xkr = PlantODE(x0,ukp,dt) + ω

            # Calculate real DAEs for plotting
            Ccr = (Ca0*Vr0 + Cc0*Vr0 - xkr[2]*xkr[1])/(xkr[1])
            # Check here as well, as negative Cc does not make physical sense
                when plotting
            if Ccr < 0
                Ccr = 0
            end
            Awr = (π*(r^2))+((0.002*xkr[1])/(r))
            zr = [Ccr,Awr]

            # Store real x
            push!(xkrList, xkr)
            # Store real DAEs
            push!(zrList,zr)

            # Update covariance matrix Pk for next iteration
            # Define transition matrix A
            Ak = IM + dt*lin1(xek,ukp)
            Pk = ricatti(Ak,Pk0,C,R,Q)
            # Store Pk
            push!(PkList,Pk)
        else
            T = i
            # Generate random normal-distributed process noise
            vrpn = rand(Normal(μ,σω[1]))
            capn = rand(Normal(μ,σω[2]))
```

```
    cbpn = rand(Normal(μ,σω[3]))
    trpn = rand(Normal(μ,σω[4]))
    tjpn = rand(Normal(μ,σω[5]))
    ω = [vrpn,capn,cbpn,trpn,tjpn]

    # Generate random normal-distributed measurement noise
    vrmn = rand(Normal(μ,σv[1]))
    trmn = rand(Normal(μ,σv[2]))
    tjmn = rand(Normal(μ,σv[3]))
    v = [vrmn,trmn,tjmn]

    # Obtain input, previous xek,xkr,Pk
    # Known input
    ukk = ukList[i-1]
    # Build ukp argument for PlantODE
    ukkp = [ukk;p0]
    # Previous covariance matrix
    Pkprev = PkList[i-1]
    # Real state
    xkrprev = xkrList[i]
    # Current measurement
    yk = measurement(xkrprev) + v
    push!(ykList,yk)

    # Obtain xkcheck (first xk outside window for AC) and the
        corresponding Pk for that specific iteration
    # Before the  estimation window is filled up, xcheck, Pkcheck = x0,
        Pk0
    if i <= N
        xkcheck = x0
        Pkcheck = Pk0
    else
        xkcheck = xekList[i-N]
        Pkcheck = PkList[i-N]
    end

    # Construct input argument. Only pass the most recent (final 10)
        inputs to the MHE
    uwindow = ukList[max(1,i-N+1):i-1]

    # Update ykList argument. Only pass the most recent (final 10)
        measurements to the MHE
    ykwindow = ykList[max(1,i-N+1):i]

    # Obtain estimate based off previous measurements
    xek,optFlag1 = MHE(xkrprev,uwindow,p0,T,N,xkcheck,ykwindow,Pkcheck)
    print("Iteration: ",i,", Solved to optimailty: ",optFlag1)
    push!(solveList1,optFlag1)

    # Calculate estimated DAEs for plotting
    Cce = (Ca0*Vr0 + Cc0*Vr0 - xek[2]*xek[1])/(xek[1])
    Awe = (π*(r^2))+((0.002*xek[1])/(r))
    ze = [Cce,Awe]

    # Store estimate
    push!(xekList,xek)
    # Store estimated DAEs
    push!(zeList,ze)

    # Perform the NMPC with the estimate
    uk,optFlag2 = nMPC(ukk,xek,ze,p0,NFE)
    print("Iteration: ",i,", Solved to optimailty: ",optFlag2)
    # Store flags for optimality
    push!(solveList2,optFlag2)
    # Store optimal inputs
    push!(ukList,uk)

    # Inject uk into plant and obtain optimal states
    ukp = [uk;p0]
    # Obtain real x
    xkr = PlantODE(xkrprev,ukp,dt) + ω

    # Calculate real DAEs for plotting
```

```
            Ccr = (Ca0*Vr0 + Cc0*Vr0 - xkr[2]*xkr[1])/(xkr[1])
            # Check here as well, as negative Cc does not make physical sense
                when plotting
            if Ccr < 0
                Ccr = 0
            end
            Awr = (π*(r^2))+((0.002*xkr[1])/(r))
            zr = [Ccr,Awr]

            # Update Pk for next iteration
            # Define transition matrix A
            Ak = IM + dt*lin1(xek,ukp)
            # Update Pk for next iteration using Ricatti equation
            Pk = ricatti(Ak,Pkprev,C,R,Q)

            # Store real x
            push!(xkrList,xkr)
            # Store real DAEs
            push!(zrList,zr)
            # Store covariance matrices
            push!(PkList,Pk)

            # Need to measure and estimate final state (for plotting)
            if i == NFE0
                xkrprev = xkrList[end]
                yk = measurement(xkr) + v
                push!(ykList,yk)

                # Construct input argument. Only pass the most recent (final 10)
                     inputs to the MHE
                uwindow = ukList[max(1,i-N+1):i-1]

                # Update ykList argument. Only pass the most recent (final 10)
                    measurements to the MHE
                ykwindow = ykList[max(1,i-N+1):i]

                xkcheck = xekList[i-N]
                Pkcheck = PkList[i-N]
                xek,optFlag1 = MHE(xkrprev,uwindow,p0,T,N,xkcheck,ykwindow,
                    Pkcheck)
                print("Iteration: ",i,", Solved to optimailty: ",optFlag1)
                push!(solveList1,optFlag1)

                # Calculate estimated DAEs for plotting
                Cce = (Ca0*Vr0 + Cc0*Vr0 - xek[2]*xek[1])/(xek[1])
                Awe = (π*(r^2))+((0.002*xek[1])/(r))
                ze = [Cce,Awe]

                # Store estimate
                push!(xekList,xek)
                # Store estimated DAEs
                push!(zeList,ze)

                # Perform the NMPC with the estimate
                uk,optFlag2 = nMPC(ukk,xek,ze,p0,NFE)
                print("Iteration: ",i,", Solved to optimailty: ",optFlag2)
                # Store flags for optimality
                push!(solveList2,optFlag2)
                # Store inputs
                push!(ukList,uk)

                # Inject uk into plant and obtain optimal states
                ukp = [uk;p0]
                # Obtain real x
                xkr = PlantODE(xkrprev,ukp,dt) + ω

                # Calculate real DAEs for plotting
                Ccr = (Ca0*Vr0 + Cc0*Vr0 - xkr[2]*xkr[1])/(xkr[1])
                # Check here as well, as negative Cc does not make physical
                    sense when plotting
                if Ccr < 0
                    Ccr = 0
                end
```

```
                 Awr = (π*(r^2))+((0.002*xkr[1])/(r))
                 zr = [Ccr,Awr]

                 # Define transition matrix A
                 Ak = IM + dt*lin1(xek,ukp)
                 # Update Pk using the Ricatti equation
                 Pk = ricatti(Ak,Pkprev,C,R,Q)

                 # Store real x
                 push!(xkrList,xkr)
                 # Store real DAEs
                 push!(zrList,zr)
                 # Store covariance matrices
                 push!(PkList,Pk)

             end
         end
     end

     # Have to use hcat to convert Vector{Vector} to Matrix that can be pushed to
          an array
     xekresult = hcat(xekList...)
     xkresult = hcat(xkrList...)
     ykresult = hcat(ykList...)
     zeresult = hcat(zeList...)
     zrresult = hcat(zrList...)
     # Store all results in a matrix
     push!(xekresultsList,xekresult)
     push!(xkrresultsList,xkresult)
     push!(ykresultsList,ykresult)
     push!(zekresultsList,zeresult)
     push!(zkrresultsList,zrresult)
end

# Plotting of MHE trajectories
# Calculate and unpack function can be found in results.jl
# Obtain real, measured and estimated states from result matrix
vrr,car,cbr,trr,tjr,ccr = calculateμσ(xkrresultsList)[1]
vrm, trm, tjm = unpackyk(ykresultsList)
vre, cae, cbe, tre,tje,cce = calculateμσ(xekresultsList)[1]

# Plotting settings
styles1 = [:solid :dash :dash]
styles2 = [:solid :dash]
colors1 = [:blue :red :green]
colors2 = [:blue :red]
size = [2 2 3]

# Perform MHE plotting
xaxis = LinRange(0.0, 1.0, 21)

plot(xaxis, [vrr vre vrm], line=(size,styles1),color=(colors1), title="Reactor
    volume Vr", xlabel = "Time [h]",ylabel = "Volume [L]",label=["Real" "
    Estimated" "Measured"],legendtitle="State type (MHE)",legend=:right,
    tickfontsize=16,guidefontsize=16,titlefontsize=25, legendfontsize=15)
plot(xaxis, [car cae], line=(size,styles2),color=(colors2), title = "
    Concentration of A", xlabel = "Time [h]",ylabel = "Concentration [mol/L]",
    label=["Real" "Estimated"],legendtitle="State type (MHE)",legend=:right,
    tickfontsize=16,guidefontsize=16,titlefontsize=25, legendfontsize=15)
plot(xaxis, [cbr cbe], line=(size,styles2),color=(colors2), title = "
    Concentration of B", xlabel = "Time [h]",ylabel = "Concentration [mol/L]",
    label=["Real" "Estimated"],legendtitle="State type (MHE)",legend=:right,
    tickfontsize=16,guidefontsize=16,titlefontsize=25, legendfontsize=15)
plot(xaxis, [trr tre trm], line=(size,styles1),color=(colors1),title = "Reactor
    Temperature Tr", xlabel = "Time [h]",ylabel = "Temperature [K]",label=["
    Real" "Estimated" "Measured"],legendtitle="State type (MHE)",legend=:
    bottomleft,tickfontsize=16,guidefontsize=16,titlefontsize=25,
    legendfontsize=15)
plot(xaxis, [tjr tje tjm], line=(size,styles1),color=(colors1),title = "Jacket
    Temperature Tj", xlabel = "Time [h]",ylabel = "Temperature [K]" ,label=["
    Real" "Estimated" "Measured"],legendtitle="State type (MHE)",legend=:right,
    tickfontsize=16,guidefontsize=16,titlefontsize=25, legendfontsize=15)
```

```
plot(xaxis, [ccr cce],line=(size,styles2),color=(colors2), title = "
    Concentration of C", xlabel = "Time [h]",ylabel = "Concentration [mol/L]",
    label=["Real" "Estimated"],legendtitle="State type MHE",legend=:bottomright
    ,tickfontsize=16,guidefontsize=16,titlefontsize=25, legendfontsize=15)
```

.8 Code for obtaining final comparative results: *results.jl*

```
include("resultsKalman.jl")
include("resultsMHE.jl")


# Define required functions:
# Calculate means and standard deviation for every state over simulation based
    off the results matrix
function calculateμσ(results::Array{Array{Float64,2}})
    # Define required lists

    # For extracing data
    vrresults = Array{Array{Float64,1}}(undef,0)
    caresults = Array{Array{Float64,1}}(undef,0)
    cbresults = Array{Array{Float64,1}}(undef,0)
    trresults = Array{Array{Float64,1}}(undef,0)
    tjresults = Array{Array{Float64,1}}(undef,0)
    ccresults = Array{Array{Float64,1}}(undef,0)


    # For calculating means
    vrmeanList = Array{Float64,1}(undef,0)
    cameanList = Array{Float64,1}(undef,0)
    cbmeanList = Array{Float64,1}(undef,0)
    trmeanList = Array{Float64,1}(undef,0)
    tjmeanList = Array{Float64,1}(undef,0)
    ccmeanList = Array{Float64,1}(undef,0)

    # For calculating standard deviation
    stdvrList = Array{Float64,1}(undef,0)
    stdcaList = Array{Float64,1}(undef,0)
    stdcbList = Array{Float64,1}(undef,0)
    stdtrList = Array{Float64,1}(undef,0)
    stdtjList = Array{Float64,1}(undef,0)
    stdccList = Array{Float64,1}(undef,0)


    # Extract data
    for i in 1:length(results)
        vrseed = Array{Float64,1}(undef,0)
        caseed = Array{Float64,1}(undef,0)
        cbseed = Array{Float64,1}(undef,0)
        trseed = Array{Float64,1}(undef,0)
        tjseed = Array{Float64,1}(undef,0)
        ccseed = Array{Float64,1}(undef,0)
        for j in 1:length(results[i][1,:])
            push!(vrseed,results[i][1,j])
            push!(caseed,results[i][2,j])
            push!(cbseed,results[i][3,j])
            push!(trseed,results[i][4,j])
            push!(tjseed,results[i][5,j])
        end
        push!(vrresults,vrseed)
        push!(caresults,caseed)
        push!(cbresults,cbseed)
        push!(trresults,trseed)
        push!(tjresults,tjseed)
        ccseed = (Ca0.*Vr0.+Cc0.*Vr0.-(caseed.*vrseed))./(vrseed)
        push!(ccresults,ccseed)
    end

    # Easier to sum rows than columns apparently
    vrresults = hcat(vrresults...)
    caresults = hcat(caresults...)
    cbresults = hcat(cbresults...)
    trresults = hcat(trresults...)
    tjresults = hcat(tjresults...)
    ccresults = hcat(ccresults...)

    for i in 1:21
        # Calculate means for every point
        vrsum = sum(vrresults[i,:])
        casum = sum(caresults[i,:])
        cbsum = sum(cbresults[i,:])
        trsum = sum(trresults[i,:])
        tjsum = sum(tjresults[i,:])
        ccsum = sum(ccresults[i,:])
        vrmean = vrsum/length(vrresults[1,:])
        camean = casum/length(caresults[1,:])
        cbmean = cbsum/length(cbresults[1,:])
        trmean = trsum/length(trresults[1,:])
        tjmean = tjsum/length(tjresults[1,:])
        ccmean = ccsum/length(ccresults[1,:])

        # With the mean, ccalculate std for every point
        push!(vrmeanList,vrmean)
        push!(cameanList,camean)
        push!(cbmeanList,cbmean)
        push!(trmeanList,trmean)
        push!(tjmeanList,tjmean)
        push!(ccmeanList,ccmean)

        stdvr = stdm(vrresults[i,:],vrmeanList[i])
        stdca = stdm(caresults[i,:],cameanList[i])
        stdcb = stdm(cbresults[i,:],cbmeanList[i])
        stdtr = stdm(trresults[i,:],trmeanList[i])
        stdtj = stdm(tjresults[i,:],tjmeanList[i])
        stdcc = stdm(ccresults[i,:],ccmeanList[i])

        # Store calculated standard deviations
        push!(stdvrList,stdvr)
        push!(stdcaList,stdca)
        push!(stdcbList,stdcb)
        push!(stdtrList,stdtr)
        push!(stdtjList,stdtj)
        push!(stdccList,stdcc)
    end
    return [[vrmeanList,cameanList,cbmeanList,trmeanList,tjmeanList, ccmeanList]
        ,[stdvrList,stdcaList,stdcbList,stdtrList,stdtjList,stdccList]]
end

# Calculate upper deviation
function calculateupper(mean::Array{Float64,1},std::Array{Float64,1})
    return mean+(2*std)
end

# Calculate lower deviation
function calculatelower(mean::Array{Float64,1},std::Array{Float64,1})
    return mean-(2*std)
end

# Unpack result matrix to obtain mean measured trajectories
function unpackyk(results::Array{Array{Float64,2}})
    # Define required lists
    # For extracing data
    vrresults = Array{Array{Float64,1}}(undef,0)
    trresults = Array{Array{Float64,1}}(undef,0)
    tjresults = Array{Array{Float64,1}}(undef,0)

    # For calculating means
    vrmeanList = Array{Float64,1}(undef,0)
    cameanList = Array{Float64,1}(undef,0)
    cbmeanList = Array{Float64,1}(undef,0)
    trmeanList = Array{Float64,1}(undef,0)
    tjmeanList = Array{Float64,1}(undef,0)

    # Extract data
    for i in 1:length(results)
        vrseed = Array{Float64,1}(undef,0)
        trseed = Array{Float64,1}(undef,0)
        tjseed = Array{Float64,1}(undef,0)
        for j in 1:length(results[i][1,:])
            push!(vrseed,results[i][1,j])
            push!(trseed,results[i][2,j])
            push!(tjseed,results[i][3,j])
        end
        push!(vrresults,vrseed)
        push!(trresults,trseed)
        push!(tjresults,tjseed)
    end

    # Easier to sum rows than columns apparently
    vrresults = hcat(vrresults...)
    trresults = hcat(trresults...)
    tjresults = hcat(tjresults...)

    for i in 1:21
        # Calculate means for every point
        vrsum = sum(vrresults[i,:])
        trsum = sum(trresults[i,:])
        tjsum = sum(tjresults[i,:])
        vrmean = vrsum/length(vrresults[1,:])
        trmean = trsum/length(trresults[1,:])
        tjmean = tjsum/length(tjresults[1,:])

        # With the mean, ccalculate std for every point
        push!(vrmeanList,vrmean)
        push!(trmeanList,trmean)
        push!(tjmeanList,tjmean)
    end
    return [vrmeanList,trmeanList,tjmeanList]
end

# MHE
# Extract  means and standard deviations of the real states from the result
    matrix
vrmeans,cameans,cbmeans,trmeans,tjmeans,ccmeans = calculateμσ(xkrresultsList)[1]
stdvr,stdca,stdcb,stdtr,stdtj,stdcc = calculateμσ(xkrresultsList)[2]

# Define upper deviations
vru = calculateupper(vrmeans,stdvr)
vrl = calculatelower(vrmeans,stdvr)
cau = calculateupper(cameans,stdca)
cal = calculatelower(cameans,stdca)
cbu = calculateupper(cbmeans,stdcb)
cbl = calculatelower(cbmeans,stdcb)
tru = calculateupper(trmeans,stdtr)
trl = calculatelower(trmeans,stdtr)
tju = calculateupper(tjmeans,stdtj)
tjl = calculatelower(tjmeans,stdtj)
ccu = calculateupper(ccmeans,stdcc)
ccl = calculatelower(ccmeans,stdcc)

# Define midpoints(mean value) and vertical deviation around midpoint
midvr = (vrmeans)
wvr = (vrl .- vru) ./ 2
midca = (cameans)
wca = (cal .- cau) ./ 2
midcb = (cbmeans)
wcb = (cbl .- cbu) ./ 2
midtr = (trmeans)
wtr = (trl .- tru) ./ 2
midtj = (tjmeans)
wtj = (tjl .- tju) ./ 2
```

```
midcc = (ccmeans)
wcc = (ccl .- ccu) ./ 2

# EKF
# Extract means and standard deviations of the real states from the result
    matrix
kvrmeans,kcameans,kcbmeans,ktrmeans,ktjmeans,kccmeans = calculateμσ(
    kxkrresultsList)[1]
kstdvr,kstdca,kstdcb,kstdtr,kstdtj,kstdcc = calculateμσ(kxkrresultsList)[2]

kvru = calculateupper(kvrmeans,kstdvr)
kvrl = calculatelower(kvrmeans,kstdvr)
kcau = calculateupper(kcameans,kstdca)
kcal = calculatelower(kcameans,kstdca)
kcbu = calculateupper(kcbmeans,kstdcb)
kcbl = calculatelower(kcbmeans,kstdcb)
ktru = calculateupper(ktrmeans,kstdtr)
ktrl = calculatelower(ktrmeans,kstdtr)
ktju = calculateupper(ktjmeans,kstdtj)
ktjl = calculatelower(ktjmeans,kstdtj)
kccu = calculateupper(kccmeans,kstdcc)
kccl = calculatelower(kccmeans,kstdcc)


kmidvr = (kvrmeans)
kwvr = (kvrl .- kvru) ./ 2
kmidca = (kcameans)
kwca = (kcal .- kcau) ./ 2
kmidcb = (kcbmeans)
kwcb = (kcbl .- kcbu) ./ 2
kmidtr = (ktrmeans)
kwtr = (ktrl .- ktru) ./ 2
kmidtj = (ktjmeans)
kwtj = (ktjl .- ktju) ./ 2
kmidcc = (kccmeans)
kwcc = (kccl .- kccu) ./ 2

# Define x axis
xaxis = LinRange(0.0, 1.0, 21)

# Perform plotting for comparative study
plot(xaxis, [midvr kmidvr], ribbon = [wvr kwvr] , fillalpha = [0.65 0.35], c = [
    2 6], lw = [4 4], label = ["MHE" "EKF"],title="Reactor volume Vr", xlabel =
     "Time [h]",ylabel = "Volume [L]",legendtitle="State Estimator",legend=:
    bottomright, tickfontsize=16,guidefontsize=16,titlefontsize=25,
    legendfontsize=10)
plot(xaxis, [midca kmidca], ribbon = [wca kwca] , fillalpha = [0.65 0.35], c = [
    2 6], lw = [4 4], label = ["MHE" "EKF"],title = "Concentration of A",
    xlabel = "Time [h]",ylabel = "Concentration [mol/L]",legendtitle="State
    Estimator",legend=:bottomleft,tickfontsize=16,guidefontsize=16,
    titlefontsize=25, legendfontsize=15)
plot(xaxis, [midcb kmidcb], ribbon = [wcb kwcb] , fillalpha = [0.65 0.35], c = [
    2 6], lw = [4 4], label = ["MHE" "EKF"],title = "Concentration of B",
    xlabel = "Time [h]",ylabel = "Concentration [mol/L]",legendtitle="State
    Estimator",legend=:bottomright,tickfontsize=16,guidefontsize=16,
    titlefontsize=25, legendfontsize=10)
plot(xaxis, [midtr kmidtr], ribbon = [wtr kwtr] , fillalpha = [0.65 0.35], c = [
    2 6], lw = [4 4], label = ["MHE" "EKF"],title = "Reactor Temperature Tr",
    xlabel = "Time [h]",ylabel = "Temperature [K]",legendtitle="State Estimator
    ",legend=:bottomleft,tickfontsize=16,guidefontsize=16,titlefontsize=25,
    legendfontpointsize=10)
plot(xaxis, [midtj kmidtj], ribbon = [wtj kwtj] , fillalpha = [0.65 0.35], c = [
    2 6], lw = [4 4], label = ["MHE" "EKF"],title = "Jacket Temperature Tj",
    xlabel = "Time [h]",ylabel = "Temperature [K]",legendtitle="State Estimator
    ",legend=:bottomleft,tickfontsize=16,guidefontsize=16,titlefontsize=25,
    legendfontsize=15)
plot(xaxis, [midcc kmidcc], ribbon = [wcc kwcc] , fillalpha = [0.65 0.35], c = [
    2 6], lw = [4 4], label = ["MHE" "EKF"],title = "Concentration of C",
    xlabel = "Time [h]",ylabel = "Concentration [mol/L]",legendtitle="State
    Estimator",legend=:topleft,tickfontsize=16,guidefontsize=16,titlefontsize=
    25, legendfontsize=15)
```