NORWEGIAN UNIVERSITY OF SCIENCE AND TECHNOLOGY

TKP4580 - CHEMICAL ENGINEERING, SPECIALIZATION PROJECT

Combining robust model predictive control with sensitivity analysis

Author:
Haakon Løvdokken

Supervisor:
Johannes Jäschke

Co-supervisor:
Halvor Aarnes Krog

NTNU | Faculty of Natural Sciences
Department of Chemical Engineering

December 23, 2022

Abstract

Uncertainty is always an issue when dealing with models. This is also the case for model predictive control (MPC), which is a control scheme that uses a model of some system for predicting its future behavior. The nominal MPC does not consider uncertainty in its predictions, leading to plant-model mismatch. If we do consider uncertainty, we then have robust model predictive control (RMPC). Here, one method of RMPC is the scenario-tree based MPC, where we create different scenarios based on the most influential parameters. In order to find these parameters, we should conduct a sensitivity analysis (SA). However, in today's applications of RMPC, when the most sensitive parameters are found, they are stuck with to the end of time. Is this reasonable? We suspect that this is not the case, especially for batch processes. In this project, the aim was to study MPC, RMPC and SA, as well as implementing the closed-loop MPC and open-loop MPC, and using SA on the open-loop MPC. This was done for a simple fermentation process in a batch bioreactor. The case study was restricted to only the Sobol' method as SA, where the simulation was done in *Python* through the use of *CasADi*. It was found that, with respect to the constraint on the biomass X_s, i.e., $X_s \leq 3.7$, that closed-loop and open-loop MPC have constraint violations when considering parametric uncertainty, i.e., $\theta_i \sim U(95\%\ \theta_i, 105\%\ \theta_i)$. Here, the greatest violations were 3.723 and 3.948, respectively, from 100 iterations, which is unacceptable for a hard constraint. Hence, we want to use the scenario-tree based MPC. Firstly, we need to identify the most sensitive parameters. Using Sobol' method, the most sensitive parameters were identified as μ_m, Y_x and S_{in}, and the least sensitive parameters were identified as k_m, k_i, ν and Y_p. The best result was acquired for $N = 2^{17}$ number of samples, but this simulation lasted 6 hours, 46 minutes and 25 seconds. It was concluded that the computational expenses was too high. Future work should consider trying other methods of SA, as well as implementing the scenario-tree based MPC itself.

Preface

This report was written in conjunction with the course TKP4580 - Chemical Engineering, Specialization Project, at the Norwegian University of Science and Technology. The work presented here was conducted Autumn 2022 at the Department of Chemical Engineering.

I have learned a lot through working on this specialization project, and I would like to thank my supervisors, Johannes Jäschke and Halvor Aarnes Krog, for providing guidance on the project and expertise on different topics, as well as for being enthusiastic helpers.

Table of Contents

List of Figures

List of Tables

Abbreviations

CV	Controlled variable
DAE	Algebraic differential equation
DOF	Degrees of freedom
DV	Disturbance variable
EMPC	Economic model predictive control
FF	Factor Fixing
FM	Factor Mapping
FP	Factor Prioritization
GSA	Global sensitivity analysis
HDMR	High-dimensional model represenation
LHS	Latin hypercube sampling
LSA	Local sensitivity analysis
MC	Monte Carlo
MCF	Monte Carlo filtering
MIMO	Multi-input multi-output
MPC	Model predictive control
MS-MPC	Multi-scenario model predictive control
MV	Manipulated variable
NLP	Non-linear program
ODE	Ordinary differential equation
RMPC	Robust model predictive control
SA	Sensitivity analysis
SISO	Singe-input single-output
SMPC	Scenario-based MPC
VC	Variance Cutting

Chapter 1

Introduction

1.1 Motivation

Uncertainty is always an issue when dealing with models. This is also the case for model predictive control (MPC). MPC is a common control scheme in which a model of some system is used for predicting the future behavior of the system. MPC solves an online optimization problem, that is, it minimizes or maximizes an objective function to obtain the optimal control action[12]. However, the nominal MPC does not consider uncertainty in its predictions, leading to plant-model mismatch. That is, even if the MPC has feedback, it still needs back-off if there are hard constraints in the system.

The solution to this problem is introducing robustness into the MPC, i.e., we instead have robust model predictive control (RMPC) that considers uncertainty. There are several methods of RMPC, where one of them is the scenario-tree based method[12]. We consider different scenarios in the MPC to enhance robustness, and we create these scenarios based on the most sensitive parameter. In today's applications of RMPC, this chosen parameters is stuck with to the end of time. Is this reasonable? We suspect that this is not the case, especially for batch processes. For instance, some process in a batch bioreactor, such as an ethanol fermentation, might have great differences in the yeast growth rate from one stage in the process to another. This growth rate depends on some of the plant parameters, and some of the outputs depend directly on the growth rate. Here, say that we have some hard constraint on one of the outputs. We could implement a large back-off to ensure that this constraint is satisfied, or we could try the scenario-tree based MPC. That is, we could create an algorithm that uses sensitivity analysis (SA) for the scenario-tree branching along the time-horizon, and for every iteration of the MPC. Because of high computational costs related to the RMPC and SA, it is natural to limit ourselves to only one, or maybe a few, uncertain parameters. Thus, in this project, where the case study is a fermentation process in a batch bioreactor, the focus is on applying SA to an important constraint of the MPC.

1.2 Thesis structure

The aim of this specialization project was to study MPC, RMPC, and various SA methods, as well as implementing the closed-loop and open-loop MPC, and using the Sobol' method on the open-loop MPC. This was done for a fermentation process in a batch bioreactor.

Firstly, theory on MPC is presented in Chapter 2. Here we start with introducing the general MPC in Section 2.1, before talking about orthogonal collation in Section 2.1.1. After that, we take on RMPC in Section 2.2 and scenario-tree based MPC in Section 2.2.1.

Secondly, theory on SA is presented in Chapter 3. Here we start with introducing the concept of SA in Section 3.1, before talking about local SA in Section 3.1.1, settings of SA in Section 3.1.2 and global SA in Section 3.1.3. After that, we talk about the Sobol'method in Section 3.2 and Saltelli's modification in Section 3.2.1. Other methods are taken on in Section 3.3, i.e., Morris screening in Section 3.3.1, Monte Carlo filtering in Section 3.3.2 and FORM/SORM in Section 3.3.3.

The case study and optimization problem is presented in Chapter 4. Here we start by introducing the case study in Section 4.1, before formulating this as MPC in Section 4.1.1, and describing how we would use SA for this in Section 4.1.2.

The results on MPC and SA are presented and discussed in Chapter 5. Here we start by presenting and discussing results for the closed-loop MPC in Section 5.1, before doing the same for open-loop MPC in Section 5.2 and SA of the open-loop MPC in Section 5.3.

Finally, conclusions are made in Chapter 6. Here we start by concluding the discussed results in Section 6.1, before talking about possible future work in Section 6.2.

Furthermore, *Python* codes that were used for closed-loop MPC and open-loop MPC, and as well for the Sobol' method, are attached in the Appendix.

Chapter 2

Model predictive control

2.1 Model predictive control

Model predictive control (MPC) is a common control scheme in which a model of some system is used for predicting the future behavior of the system[12]. MPC solves an online optimization problem, that is, it minimizes or maximizes an objective function to obtain optimal control action that drives the predicted output trajectory to the reference trajectory. There are several advantages with MPC when compared to the typical PID-controllers[15]: (i) ability to handle multi-input multi-output (MIMO) systems that may have interactions between inputs and outputs, (ii) providing a systematic way of handling constraints upon inputs and outputs, (iii) being able to coordinate control calculations with the calculation of optimum set points, and (iv) ability to provide early warnings of potential problems if the model is accurate. There are also disadvantages with MPC, that is: (i) requirement of an accurate process model, (ii) online complexity, (iii) model might be difficult to maintain, (iv) commissioning costs of the modeling, and (v) less transparent control algorithm[15].

In general control theory, the outputs are called controlled variables (CVs), whilst the inputs are called manipulated variables (MVs), and the disturbances are called feedforward variables (DVs). The overall objectives of MPC, ranked by importance, are typically[11]:

1. Prevent violations of input and output constraints.
2. Drive the CVs to their steady-state optimal values.
3. Drive the MVs to their steady-state optimal values using remaining DOF.
4. Prevent excessive movement of MVs.
5. When signals and actuators fail, control as much of the plant as possible.

Here, DOF is an abbreviation for the degrees of freedom. A block diagram for the general MPC controller is shown in Figure 2.1. A process model is used for predicting the current output. The differences between the actual and predicted outputs, referred to as the residuals, makes the feedback signal to the Prediction block. These acquired predictions are used for set-point calculations and control calculations. Inequality constraints can be

required on both of these calculations. The set points (targets) for the control calculations are found based on the steady-state optimization of the process. The typical optimization objectives are maximizing profit, minimizing cost, or maximizing production. Moreover, the control calculations are based on the current measured output and the predicted output. The objective is to determine the sequence of control actions, so that the predicted output response adjusts optimally to the target[15]. This is shown in Figure 2.2, where the process is simulated discretely over the prediction horizon n_p, and control actions are allowed over the control horizon n_m. It is required that the control horizon cannot surpass the prediction horizon, i.e., $1 \leq n_m \leq n_p < \infty$, and the process must return to the steady state[15].

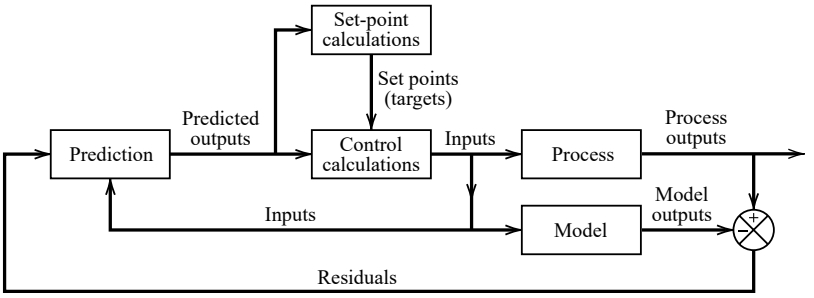

Figure 2.1: General MPC block diagram[15].

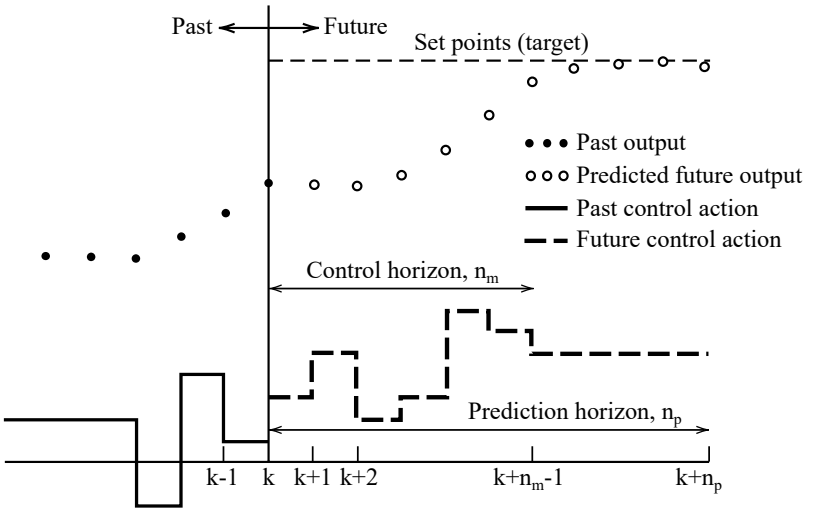

Figure 2.2: Concept of single-input single output (SISO) general MPC[15].

The figures 2.1 and 2.2 represent the general set-point tracking MPC. Here, the typical objective function for this kind of control scheme can be formulated as the following[4]:

$$\min_{x,u} \sum_{k=1}^{n_p} \underbrace{(x_k - x_{\mathrm{SP},k})^T Q (x_k - x_{\mathrm{SP},k})}_{\text{state set-point tracking}} + \underbrace{\sum_{k=1}^{n_m} \underbrace{(u_k - u_{\mathrm{ref},k})^T R_1 (u_k - u_{\mathrm{ref},k})}_{\text{input set-point tracking}} + \sum_{k=1}^{n_m} \underbrace{\Delta u_k^T R_2 \Delta u_k}_{\text{input usage penalty}}}_{\text{regularization terms}}. \tag{2.1}$$

However, for this specialization project, instead of using the general set-point tracking MPC scheme, we instead have an economic MPC (EMPC). That is, the set-point tracking is neglected, and the real-time optimization (RTO) is done together with the MPC instead of in the above control layer. The optimization is done with respect to a cost function, $J(x_k, u_k)$, which typically maximizes product. The objective of an EMPC can be[4]

$$\min_{x,u} \sum_{k=0}^{n_p} J(x_k, u_k) + \sum_{k=1}^{n_m} \Delta u_k^T R \Delta u_k, \tag{2.2}$$

which is an unconstrained optimization problem. Now, having inequality constraints on inputs and outputs is an important benefit of MPC. For example, a given flow rate (MV) has the lower limit of zero and some upper limit determined by pumps, control valves and piping characteristics, whilst the product quality (CV) in a distillation column has the lower limit of zero and some upper limit determined by dynamics or customers' demand[15]. Additionally, if one included penalty on the magnitude of the manipulated variable steps, then the constrained optimization problem for an economic MPC can be written as[4]

$$\min_{x,u} \sum_{k=0}^{n_p} J(x_k, u_k) + \sum_{k=1}^{n_m} \Delta u_k^T R \Delta u_k \tag{2.3a}$$

subject to

$$x_{k+1} = F(x_k, u_k, \theta_k), \qquad k = 0, \ldots, n_p - 1 \tag{2.3b}$$
$$g(x_k, u_k, \theta_k) \leq 0, \qquad k = 1, \ldots, n_p \tag{2.3c}$$
$$x_{min} \leq x_k \leq x_{max}, \qquad k = 1, \ldots, n_p \tag{2.3d}$$
$$u_{min} \leq u_k \leq u_{max}, \qquad k = 1, \ldots, n_m \tag{2.3e}$$
$$-\Delta u_{max} \leq \Delta u_k \leq \Delta u_{max}, \qquad k = 1, \ldots, n_m \tag{2.3f}$$

where

$$x_0 = x(0), \tag{2.3g}$$
$$\Delta u_k = u_k - u_{k-1}, \qquad k = 1, \ldots, n_m \tag{2.3h}$$
$$\Delta u_k = 0, \qquad k = n_m + 1, \ldots, n_p \tag{2.3i}$$

where $J(x_k, u_k)$ is the cost function, x_k is the measured output, allowed between x_{min} and x_{max}, and u_k is the calculated input, allowed between u_{min} and u_{max}. Furthermore, Δu_k denotes the input movement, which is allowed to vary between Δu_{min} and Δu_{max}. The predicted state, x_{k+1}, is a found from the integrator, $F(x_k, u_k, \theta_k)$, where θ_k is the

measured parameters. The nonlinear inequality constraints on the system are denoted by $g(x_k, u_k, \theta_k)$, and the input movement penalization matrix is denoted by the R matrix[4].

2.1.1 Orthogonal collocation

Orthogonal collocation on finite elements is a direct transcription method that allows for a simultaneous approach of an optimization problem. That is, instead of using an ODE/DAE solver, the integration is done together with the optimizer. Put differently, "one write out the integrator equations" and solve them together with the other constraints in the nonlinear program (NLP)[1]. As a result, one obtain very large NLPs, but with sparse structures that can be exploited by the NLP solver. For simplification, consider now the ODE:

$$\dot{x} = f(x), \qquad x(0) = x_0. \tag{2.4}$$

Assume that the solution $x(t)$ can be approximated by the $K+1$ order polynomial:

$$x_i^K(t) = \alpha_0 + \alpha_1 t + \alpha_2 t^2 + \cdots \alpha_K t^K, \tag{2.5}$$

valid on the finite-time elements $t \in [t_i, t_{i+1}]$. Using Lagrange interpolation polynomials and the $j = 0, \ldots, K$ interpolation points $(t_j, x_{i,j})$ in the interval $[t_i, t_{i+1}]$, results in[1]

$$x_i^K(t) = \sum_{j=0}^{K} l_j(\tau) x_{i,j}, \tag{2.6}$$

where $l_j(\tau)$ is the Lagrangrian basis polynomial with dimensionless time $\tau \in [0, 1]$[1]:

$$l_j(\tau) = \prod_{k=0, k\neq j}^{K} \frac{\tau - \tau_k}{\tau_j - \tau_k}, \qquad \tau = \frac{t - t_t}{\Delta t_i}, \qquad \Delta t_i = t_{i+1} - t_i. \tag{2.7}$$

It is important to note that the basis polynomial $l_j(\tau)$ is defined such that $l_j(\tau_j) = 1$ and $l_j(\tau_i) = 0$ for all the interpolation points where $i \neq j$. Such a polynomial ensures that $x^K(t_{i,j}) = x_{i,j}$, and the polynomial is fitted to all the finite elements, see Figure 2.3. Finally, the integration equations to be used in the optimizer, can be formulated as[1]

$$\sum_{j=0}^{K} \underbrace{\frac{dl_j}{d\tau}\bigg|_{\tau_k}}_{a_{j,k}} \frac{x_{i,j}}{\Delta t} = f(x_{i,k}), \qquad k = 1, \ldots, K, \tag{2.8}$$

where $a_{j,k}$ are constants that can be pre-computed. Furthermore, there is one equation missing, that is needed to ensure ensure continuity between the finite elements, which is[1]

$$x_{i+1,0} = x_i^K(t_{i+1}) = \sum_{j=0}^{K} \underbrace{l_j(1) x_{i,j}}_{d_j}, \tag{2.9}$$

where, similarly to the collocation coefficients $a_{j,k}$, the continuity coefficients d_j can be pre-computed. There are different approaches to orthogonal collocation, all with varying number of collocation points and positions. The most common ones are Gauss-Lobatta, Gauss-Legendre and Gauss-Radau, where the last two approaches are shown in Table 2.1.

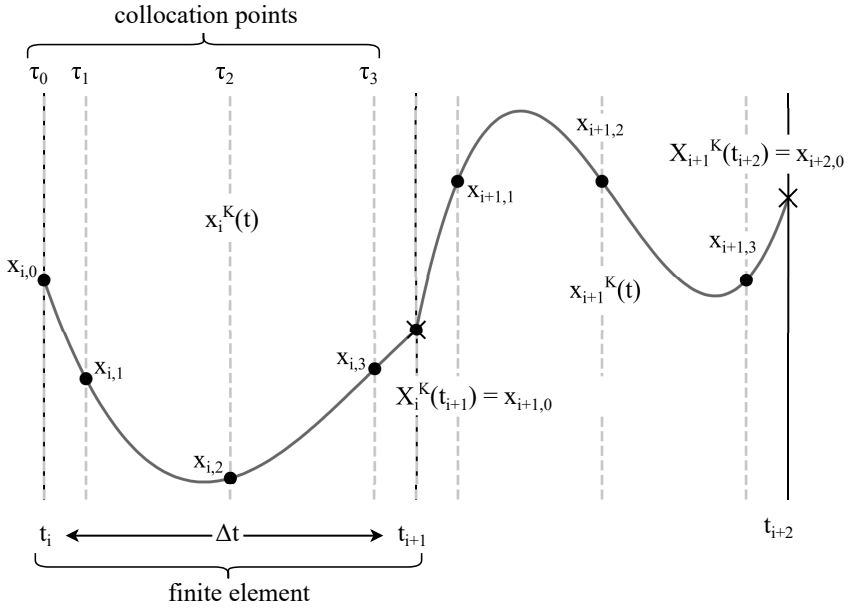

Figure 2.3: Lagrange polynomials to approximate solution of an ODE [10].

Table 2.1: Gauss–Legendre and Gauss-Radau roots as collocation points [1].

Degree K	Gauss–Legendre	Gauss-Radau
1	0.500000	1.000000
2	0.211325	0.333333
	0.788675	1.000000
3	0.112702	0.155051
	0.500000	0.644949
	0.887298	1.000000
4	0.069432	0.088588
	0.330009	0.409467
	0.669991	0.787659
	0.930568	1.000000
5	0.046910	0.057104
	0.230765	0.276843
	0.500000	0.583590
	0.769235	0.860240
	0.953090	1.000000

Finally, one could have reformulated the MPC, i.e., eq. (2.3), to the following[1]:

$$\min_{x_{i,k},x_k,u_k} \sum_{k=0}^{n_p} J(x_k,u_k) + \sum_{k=1}^{n_m} \Delta u_k^T R \Delta u_k \tag{2.10a}$$

subject to

$$\sum_{j=0}^{K} a_{j,k} \frac{x_{i,j}}{\Delta t} = f(x_{i,k}), \qquad k = 1,\dots,K \tag{2.10b}$$

$$g(x_k,u_k,\theta_k) \leq 0, \qquad k = 1,\dots,n_p \tag{2.10c}$$

$$x_{min} \leq x_k \leq x_{max}, \qquad k = 1,\dots,n_p \tag{2.10d}$$

$$u_{min} \leq u_k \leq u_{max}, \qquad k = 1,\dots,n_m \tag{2.10e}$$

$$-\Delta u_{max} \leq \Delta u_k \leq \Delta u_{max}, \qquad k = 1,\dots,n_m \tag{2.10f}$$

where

$$x_0 = x(0), \tag{2.10g}$$

$$\Delta u_k = u_k - u_{k-1}, \qquad k = 1,\dots,n_m \tag{2.10h}$$

$$\Delta u_k = 0, \qquad k = n_m+1,\dots,n_p \tag{2.10i}$$

$$x_{i+1,0} = \sum_{j=0}^{K} d_j x_{i,j}, \qquad i = 1,\dots,K \tag{2.10j}$$

$$x_{i,K} = \sum_{j=0}^{K} d_j x_{K,j}, \qquad x_{1,0} = x(t_0) \tag{2.10k}$$

2.2 Robust model predictive control

In standard MPC formulations, the model sees the world as perfect, when in reality there are disturbances and uncertainties that should be accounted for[12]. If not, this can lead to plant-model mismatch, which gives worse performance and possibly constraint violations. The solution to this problem is introducing robustness into the controller. Robust model predictive control (RMPC) serves this purpose, and it includes various methods of MPC that guarantee to optimal performance while also considering uncertainty in the system. Some methods are the min-max MPC, tube-based MPC and scenario-tree based MPC[6] [7].

The min-max MPC strategy involves optimization of the worst-case performance with respect to the uncertainties[6]. Unfortunately, this results in conservative control and with small domains of feasibility. Solving min-max MPC problems often is too computationally demanding for practical implementation, especially for closed-loop MPC. The tube-based MPC is a more recently developed approach of RMPC, and it focuses more on efficiency. Here, an ancillary feedback controller that acts on the state deviations, is designed in order for keeping the actual state trajectories within an invariant ”tube” around the nominal trajectory, which is calculated by solving the nominal MPC[16]. However, this project focuses on the latter RMPC method mentioned, that is, the scenario-tree based MPC.

2.2.1 Scenario-tree based MPC

Scenario-tree based MPC, which is similar to the likes of scenario-based MPC (SMPC), multi-scenario MPC (MS-MPC), or multi-stage scenario-based MPC, revolves around the idea of introducing different scenarios in the MPC to give robustness[6]. Here, the future uncertainties in the prediction horizon are represented through a scenario-tree. The control trajectories are then computed online for the different scenarios. For instance, for only one robust horizon, i.e., the optimization problem is only branched once, the scenario-tree MPC could look like Figure 2.4. Here, each of the scenarios has its own separate cost, and the objective is to find input sequence $\{u_k, \ldots, u_{k+N}\}$ that minimizes the expected cost.

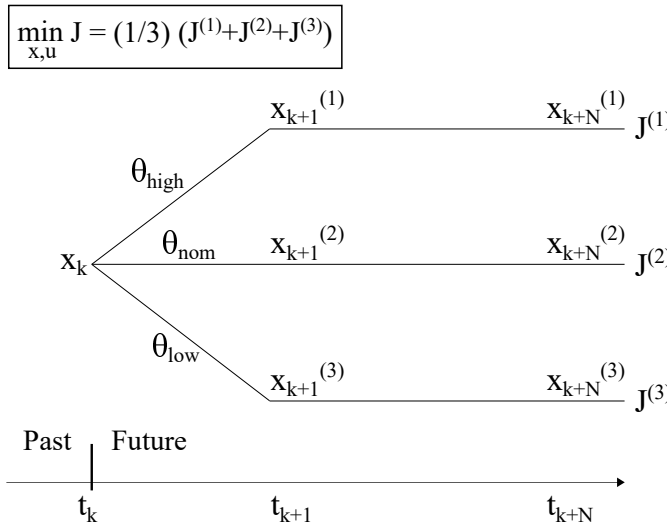

Figure 2.4: Scenario-tree based MPC with one robust horizon.

These scenarios are defined by considering one, or maybe a few, uncertain parameters out of potentially many uncertain plant parameters. For simplicity, say that we only want one uncertain parameter at the time t_k. This is not unreasonable, as we are restricted by computational complexity. Which of the parameters that are chosen for generating the scenarios is done online by a method of sensitivity analysis. In general, one should select the parameter that affects the cost function the most. In Figure 2.4, we have three possible scenarios of the parameter θ, i.e., θ_{high}, θ_{nom} and θ_{low}. How do we choose what scenario that should be realized? For this we have to create an algorithm that selects θ based on J, as well as selecting the weighting of each scenario. In Figure 2.4, the weightings are equal for simplicity. Implementation of these scenario-trees was out of scope for this project. The focus is on sensitivity analysis, which is presented in Chapter 3. We restrain ourselves to only parametric uncertainty, as uncertain inputs would give extra computational costs.

Chapter 3

Sensitivity analysis

3.1 Sensitivity analysis

Sensitivity analysis (SA) is defined as the study of how the uncertainty in the outputs of a model can be apportioned to the different sources of uncertainty in the model inputs[14]. Generally, there are two main approaches to SA: (i) Local sensitivity analysis (LSA) and (ii) Global sensitivity analysis (GSA). LSA methods are usually implemented through calculating the partial derivatives of the outputs with respect to the inputs. GSA methods, however, are carried out by apportioning the output uncertainty to the inputs' uncertainty, using probability distributions for the inputs' entire range[13]. This report focuses on GSA.

3.1.1 Local sensitivity analysis

In literature we are often met with sensitivity defined as based on derivatives. It is indeed, that the partial derivative $\partial Y_j / \partial X_i$ can be used as a definition on the sensitivity of Y_j against X_i. This approach is attractive due to its efficiency in computational time, as the required model executions are generally small[13]. Thus, the sensitivity can be written as

$$S^p_{X_i} = \frac{\partial Y_j}{\partial X_i}, \tag{3.1}$$

where Y_j are the model outputs and X_i are the model inputs. The superscript p denotes "partial derivative". The derivatives $S^p_{X_i}$ are non-normalized, and one way of improving eq. (3.1) is by introducing sigma-normalized derivatives, which can be formulated as[14]

$$S^\sigma_{X_i} = \frac{\sigma_{X_i} \partial Y_j}{\sigma_{Y_j} \partial X_i}. \tag{3.2}$$

However, derivative-based approaches are unwarranted when the model inputs have uncertainty and when the model is nonlinear. For that reason, this project focuses on GSA. We seek to use conditional variances for describing the sensitivities[14].

3.1.2 Settings in sensitivity analysis

In literature we can find cases where different sensitivity methods are used for the same problem in a non-structured practise[13]. This can yield quite different results, e.g., when it comes to ranking the input factors after importance. Here, it is difficult to know for sure what is the true answer. For this issue, we use "settings" as a method for framing the sensitivity task, such that the results can be entrusted. There are different settings, but for selecting the most appropriate one, we have to carefully consider: (i) the output of interest, and (ii) the concept of "importance". Here follows a list of some possible settings[14]:

- Factor Prioritization (FP) setting is used to identify an input factor (or a group of input factors), that when fixed at its true value, gives the largest variance reduction of the output. That is, the identified input factor (or a group of input factors) is the one that accounts for the most of the output variance.
- Factor Fixing (FF) setting is used to identify input factors in the model, which, allowed to vary freely over the range of uncertainty, contributes very little to the output variance. Then, the identified input factors can be fixed at any value within their range of variation, without affecting the output variance.
- Variance Cutting (VC) setting is used to the reduce the output variance to below a given tolerance. Typically, this may be desirable in reliability analysis.
- Factor Mapping (FM) setting is used to identify what values of the input factors leads to model realizations in some given range of the model output space.

Out of the four settings mentioned, the first three are susceptible to variance-based SA, i.e., a form of GSA. The utility of variance-based SA comes from its many applications[14].

3.1.3 Global sensitivity analysis

GSA methods are carried out by apportioning the output uncertainty to the input factors' uncertainty, using probability distributions that cover the input factors' entire range[13]. These ranges are important, as they represent the knowledge that we have or are lacking, with respect to the model and its parameterization. One well-known approach of GSA is the variance-based sensitivity analysis, which uses variance as the basis to find a measure of the input influence on the output variation. This choice feels natural, as variance can be used as a measure of dispersion or variability in the model prediction, indicating its precision due to input variations. Nevertheless, consider now the generic model[14]

$$Y = f(X_1, X_2, \ldots, X_3). \tag{3.3}$$

Here, each X_i has a non-null range of variation or uncertainty. Imagine now that we fix the factor X_i at some value x_i^*. Let $V_{\mathbf{X}_{\sim i}}(Y|X_i = x_i^*)$ be the resulting variance of Y, taken over $\mathbf{X}_{\sim i}$ (i.e., all the factors except X_i). This is called the *conditional variance*, since it is conditional on X_i being fixed to x_i^*. Now, if we average this over all the possible point x_i^*, the dependence on x_i^* disappears. This can be formulated as $E_{X_i}(V_{\mathbf{X}_{\sim i}}(Y|X_i))$. In fact, we always have $E_{X_i}(V_{\mathbf{X}_{\sim i}}(Y|X_i)) \leq V(Y)$, which comes from the equality[14],

$$E_{X_i}(V_{\mathbf{X}_{\sim i}}(Y|X_i)) + V_{X_i}(E_{\mathbf{X}_{\sim i}}(Y|X_i)) = V(Y). \tag{3.4}$$

Hence, from observing eq. (3.1), a small $E_{X_i}(V_{\mathbf{X}_{\sim i}}(Y|X_i))$ or a large $V_{X_i}(E_{\mathbf{X}_{\sim i}}(Y|X_i))$ implies that X_i is an important factor. The conditional variance $V_{X_i}(E_{\mathbf{X}_{\sim i}}(Y|X_i))$ is also called *the first-order effect of X_i on Y*. Likewise, we have the sensitivity measure[14],

$$S_i = \frac{V_{X_i}(E_{\mathbf{X}_{\sim i}}(Y|X_i))}{V(Y)}, \tag{3.5}$$

which is called *the first-order sensitivity index of X_i on Y*, and we must have $S_i \in [0, 1]$. Now, what if the conditional variance had multiple factors instead of one? For instance, say that we have two factors X_i, X_j. Then, the conditional variance can be written as[14]

$$\frac{V(E(Y|X_i, X_j))}{V(Y)}, \tag{3.6}$$

where $i \neq j$, and we dropped the indices of both E and V. Then, the following is true:

$$V(E(Y|X_i, X_j)) = V_i + V_j + V_{ij}, \tag{3.7}$$

where V_i, V_j and V_{ij} can be written as

$$V_i = V(E(Y|X_i)) \tag{3.8a}$$
$$V_j = V(E(Y|X_j)) \tag{3.8b}$$
$$V_{ij} = V(E(Y|X_i, X_j)) - V_i - V_j. \tag{3.8c}$$

Here, V_{ij} represents the interaction between the factors X_i and X_j. A non-linear additive model (e.g., $Y = \sum_i X_i^2$) will not have any V_{ij} terms, while a non-linear non-additive model (e.g., $Y = \prod_i X_i$) will have non-zero V_{ij} terms. That is, even for a non-additive model, we are able to fully understand the model's sensitivities, granted that patience is required for the time consuming computations. For k input factors, we have[14]

$$\sum_i S_i + \sum_i \sum_{j>i} S_{ij} + \sum_i \sum_{j>i} \sum_{l>j} S_{ijl} + \cdots + S_{123\ldots k} = 1. \tag{3.9}$$

Moreover, we can express eq. (3.1) by the total effect index. That is, the total effect accounts for the contribution from the output variation due to the factor X_i (i.e., the first-order effect) plus all the higher effects due to interactions. This can be formulated as[14]

$$S_{T_i} = 1 - \frac{V(E(Y|\mathbf{X}_{\sim i))}}{V(Y)} = \frac{E(V(Y|\mathbf{X}_{\sim i))}}{V(Y)}. \tag{3.10}$$

Up until now, we have assumed that the input factors are independent of each other. The reasoning behind this is quite natural, as dependent input samples are time consuming to generate, and the required sample size for computing sensitivity measures for dependent samples is much higher. Thus, it is advised to work with only uncorrelated samples[14].

In Section 3.1.2 we talked about *settings*. How does settings relate to the first-order sensitivity index S_i and the total sensitivity index S_{T_i}? The short answer is that S_i relates to the Factor Prioritization (FP) and S_{T_i} relates to the Factor Fixing (FF). For instance, when it comes to research prioritization, one could ask the question "Which factor is the most deserving of further analysis?", in which we link S_i to the FP setting. Or when it comes to model simplification, one could ask the question "Can some factors of the model be fixed or simplified?", in which we link S_{T_i} to the FF setting[14]. Such questions are important to ask beforehand. In Section 2.2.1 we talked about scenario-trees, and that these trees should be based on the most sensitive parameter. However, is it S_i, S_{T_i} or both that should be used in the scenario-tree algorithm? For now, we leave this unanswered.

3.2 Sobol' method

The Sobol' method is one such method of variance-based SA, formed by I. M. Sobol'[17]. Now, consider a square-integrable function f over Ω^k, the k-dimensional unit hypercube,

$$\Omega^k = (X \mid 0 \leq x_i \leq 1; \qquad i = 1, ..., k), \tag{3.11}$$

where Sobol' method considers an expansion of f into terms of increasing dimensions[14],

$$f = f_0 + \sum_i f_i + \sum_i \sum_{j>i} f_{ij} + \cdots + f_{12...k}. \tag{3.12}$$

Here, each of the terms are also square integrable over the domain, and they are only functions of the index factors, i.e., $f_i = f_i(X_i)$, and $f_{ij} = f(X_i, X_j)$, and so forth. This is not a series decomposition, as the number of terms is finite. More specifically, it has 2^k terms, where one term is constant (f_0), and there are k first-order functions, and $\binom{k}{2}$ second order functions (f_{ij}), and so forth. This expansion is called a high-dimensional model representation (HDMR), and it is not unique. That is, for some model f, it could be an infinite number of choices for its terms. Furthermore, if each of the terms have mean equal to zero, i.e., $\int f(x_i)dx_i = 0$, then Sobol' proved that all the terms are orthogonal in pairs, i.e., $\int f(x_i)f(x_j)dx_i dx_j = 0$. Consequently, each of these terms can be univocally calculated with the conditional expectation of the model output Y. Thus, it follows that[14]

$$f_0 = E(Y) \tag{3.13a}$$
$$f_i = E(Y|X_i) - E(Y) \tag{3.13b}$$
$$f_{ij} = E(Y|X_i, X_j) - f_i - f_j - E(Y). \tag{3.13c}$$

The conditional expectation $E(Y|X_i)$ can be calculated by slicing the X_i domain and averaging the values of $Y|X_i$. The variance of $E(Y|X_i)$ can be considered as a summary measure of sensitivity. In fact, $V(f_i(X_i))$ is another way of writing $V[E(Y|X_i)]$, so that if we divide by unconditional variance $V(Y)$, we obtain the first-order sensitivity index[14],

$$S_i = \frac{V[E(Y|X_i)]}{V(Y)}, \tag{3.14}$$

which is the main effect contribution of each input factor to the output variance. However, we might have interaction effects too. Two factors interact when their effect on output Y cannot be written as a sum of their individual effects. Decomposing eq. (3.1) gives that[14]

$$V_i = V(f_i(X_i)) = V[E(Y|X_i)] \tag{3.15a}$$
$$V_{ij} = V(f_{ij}(X_i, X_j)) = V(E(Y|X_i, X_j)) - V(E(Y|X_i)) - V(E(Y|X_j)). \tag{3.15b}$$

Here, $V(E(Y|X_i, X_j))$ measures the joint effect of the pair X_i, X_j on the output Y, and $V(f_{ij})$ is equal to this joint effect minus the first-order effects for the same factors. $V(f_{ij})$ is also known as the second-order effect. Similarly, this can be done for higher-order terms. We abbreviate $V(f_i) = V_i$, $V(f_{ij}) = V_{ij}$, and so on, and square integrate each term of the eq. (3.1) over Ω^k, giving a so-called ANOVA-HDMR decomposition[14]:

$$V(Y) = \sum_i V_i + \sum_i \sum_{j>i} V_{ij} + \cdots + V_{12\ldots k}. \tag{3.16}$$

Dividing both sides of the eq. (3.1) by $V(Y)$, results in

$$\sum_i S_i + \sum_i \sum_{j>i} \sum_{l>j} S_{ijl} + \cdots + S_{123\ldots k} = 1. \tag{3.17}$$

Total effects come as consequences of Sobol's variance decomposition. The total effect index accounts for the total contribution to the output variation due to factor X_i, i.e, its first-order effect, plus all higher-order effects. For instance, say that some model has three input factors. Then, the total effect of X_1 is the sum of all terms in eq. (3.1), which is[14]

$$S_{T1} = S_1 + S_{12} + S_{13} + S_{123}. \tag{3.18}$$

Here, the total index consists of four terms, where the latter three terms give useful information on the non-additive features. The unconditional variance is decomposed[14],

$$V(Y) = V(E(Y|X_i)) + E(V(Y|X_i)). \tag{3.19}$$

Another way of defining the total index is by decomposing the output variance $V(Y)$ in terms of main effect and residual, conditioning with respect to all factors except one[14],

$$V(Y) = V(E(Y|\mathbf{X}_{\sim i})) + E(V(Y|\mathbf{X}_{\sim i})). \tag{3.20}$$

Here, $V(Y) - V(E(Y|\mathbf{X}_{\sim i})) = E(V(Y|\mathbf{X}_{\sim i})$ denotes the remaining variance of Y that would be left, on average, if we could find the true values of $\mathbf{X}_{\sim i}$. If we divide this by the unconditional variance $V(Y)$, we can finally obtain the total effect index for $\mathbf{X}_{\sim i}$[14]:

$$S_{T_i} = \frac{E[V(Y|\mathbf{X}_{\sim i})]}{V(Y)} = 1 - \frac{V[E(Y|\mathbf{X}_{\sim i})]}{V(Y)}. \tag{3.21}$$

Now, we want to use the Monte Carlo based numerical procedure for computing the first-order and total-effect indices for a model of k input factors. This is the best available procedure for computing indices purely based on model evaluations[14]. In next section, we propose a shortcut for computing indices more efficiently, i.e., Saltelli's modification.

3.2.1 Saltelli's modification

We want to generate a $(N, 2k)$ matrix containing random numbers, and we the define the two matrices A and B, each containing half of the random sample (see 3.1 and 3.1). Here, N is the base sample, i.e., typically a few thousands, and k is the number of inputs[14].

$$A = \begin{bmatrix} x_1^{(1)} & x_2^{(1)} & \cdots & x_i^{(1)} & \cdots & x_k^{(1)} \\ x_1^{(2)} & x_2^{(2)} & \cdots & x_i^{(2)} & \cdots & x_k^{(2)} \\ \cdots & \cdots & \cdots & \cdots & \cdots & \cdots \\ x_1^{(N-1)} & x_2^{(N-1)} & \cdots & x_i^{(N-1)} & \cdots & x_k^{(N-1)} \\ x_1^{(N)} & x_2^{(N)} & \cdots & x_i^{(N)} & \cdots & x_k^{(N)} \end{bmatrix} \tag{3.22}$$

$$B = \begin{bmatrix} x_{k+1}^{(1)} & x_{k+2}^{(1)} & \cdots & x_{k+i}^{(1)} & \cdots & x_{2k}^{(1)} \\ x_{k+1}^{(2)} & x_{k+2}^{(2)} & \cdots & x_{k+i}^{(2)} & \cdots & x_{2k}^{(2)} \\ \cdots & \cdots & \cdots & \cdots & \cdots & \cdots \\ x_{k+1}^{(N-1)} & x_{k+2}^{(N-1)} & \cdots & x_{k+i}^{(N-1)} & \cdots & x_{2k}^{(N-1)} \\ x_{k+1}^{(N)} & x_{k+2}^{(N)} & \cdots & x_{k+i}^{(N)} & \cdots & x_{2k}^{(N)} \end{bmatrix} \tag{3.23}$$

Now, matrix C_i is made of all columns in B except from the i'th, which is from A[14]:

$$C = \begin{bmatrix} x_{k+1}^{(1)} & x_{k+2}^{(1)} & \cdots & x_i^{(1)} & \cdots & x_{2k}^{(1)} \\ x_{k+1}^{(2)} & x_{k+2}^{(2)} & \cdots & x_i^{(2)} & \cdots & x_{2k}^{(2)} \\ \cdots & \cdots & \cdots & \cdots & \cdots & \cdots \\ x_{k+1}^{(N-1)} & x_{k+2}^{(N-1)} & \cdots & x_i^{(N-1)} & \cdots & x_{2k}^{(N-1)} \\ x_{k+1}^{(N)} & x_{k+2}^{(N)} & \cdots & x_i^{(N)} & \cdots & x_{2k}^{(N)} \end{bmatrix} \tag{3.24}$$

Using the sample matrices A, B and C_i, we compute the model output for all the sampled input values, giving the following $N \times 1$ dimensional vectors of model outputs[14]:

$$y_A = f(A), \qquad y_B = f(B), \qquad y_{C_i} = f(C_i). \tag{3.25}$$

We assume these vectors are everything needed for finding the first-order indices S_i and total-effect indices S_{T_i}, for some input factor X_i. The total cost is only $N(k+2)$, which is quite lower than the original N^2 amount of iterations for the brute-force method. Now, the recommended formula for estimating the first-order sensitivity indices is[14]

$$S_i = \frac{V[E(Y|X_i)]}{V(Y)} = \frac{y_A \cdot y_{C_i} - f_0^2}{y_A \cdot y_A - f_0^2} = \frac{(1/N)\sum_{j=1}^{N} y_A^{(j)} y_{C_i}^{(j)} - f_0^2}{(1/N)\sum_{j=1}^{N} y_A^{(j)} y_A^{(j)} - f_0^2}, \tag{3.26}$$

where we have the mean defined as

$$f_0^2 = \Big(\frac{1}{N}\sum_{j=1}^{N} y_A^{(j)}\Big)^2. \tag{3.27}$$

Here, $(\cdot)$ is the scalar product of two vectors. Similarly, the total-effect indices are[14]:

$$S_{T_i} = 1 - \frac{V[E(Y|\mathbf{X}_{\sim i})]}{V(Y)} = 1 - \frac{y_B \cdot y_{C_i} - f_0^2}{y_A \cdot y_A - f_0^2} = 1 - \frac{(1/N)\sum_{j=1}^{N} y_B^{(j)} y_{C_i}^{(j)} - f_0^2}{(1/N)\sum_{j=1}^{N} y_A^{(j)} y_A^{(j)} - f_0^2}. \tag{3.28}$$

In the scalar product $y_A \cdot y_{Ci}$, the values of output Y from the matrix A are multiplied by the values of output Y in which all factor except X_i are resampled while the values of the input factor X_i remains fixed. If X_i is non-influential, then high and low values of y_A and y_{Ci} are randomly associated. However, if X_i is influential, then high (or low) values of y_A is multiplied by high (or low) values of y_{Ci}, thus increasing the value of the resulting scalar product. It is by design such that S_{T_i} always is greater than or equal to S_i[14].

There are several methods for generating random samples for A and B. One proposal is the *Latin hypercube sampling* (LHS), which is normal for Monte Carlo (MC) simulations. It might reduce the required iterations significantly.[13] LHS is inspired of the Latin square, having one sample in each row and column of the square. Moreover, *hypercube* means a cube with more dimensions than three. LHS is carried out by dividing some probability distribution into N equal parts, for each of the input factors, and then sampling randomly within that part. This typically performs better than the *random-* or *stratified sampling*[13].

3.3 Other methods

There are other SA methods that are interesting for the scenario-tree based MPC, but were not implemented in this project; in short: (i) Morris screening is interesting due to its low computational time, (ii) Monte Carlo filtering is interesting because it could tell us what realizations violate some constraint, and (iii) FORM/SORM is interesting since it can give the possibility of violating some constraint. Now, let us talk more about these methods.

3.3.1 Morris screening

Morris screening is another important method of GSA. In general, screening methods are used for identifying importance of the input factors, by using a quite small number of runs. This gives somewhat simple sensitivity measures. However, the most useful information lies in the ranking itself instead of in the accuracy of the input factors with respect to the model outputs. Hence, Morris screening is valuable in early SA phases, as it is useful for finding input factors of less importance, which can be left out of further SA[14].

Consider now a model Y of independent inputs X_i, where $i = 1, \ldots, k$, varying in the k-dimensional unit cube over the p chosen levels. For some value of $\mathbf{X} = (X_1, X_2, X_3)$, the elementary effect EE_i for the i'th input factor can be formulated as[14]

$$EE_i = \frac{[Y(X_1, X_2, \ldots, X_{i-1}, X_i + \Delta, \ldots, X_k) - Y(X_1, X_2, \ldots, X_3)]}{\Delta}, \tag{3.29}$$

where p denotes number of levels, and Δ denotes a value $\in \{1/(p-1), \ldots, 1-1/(p-1)\}$. The total sensitivity index S_{T_i} can be used for identifying non-influential inputs factors[14],

$$S_{T_i} = \frac{E_{\mathbf{X}_{\sim i}}(V_{X_i}(Y|\mathbf{X}_{\sim i}))}{V(Y)}, \tag{3.30}$$

but when the computational cost of S_{T_i} is expensive, we instead use an effective substitute. We can use the sensitivity measures proposed by Morris, i.e., μ and σ, as estimates of mean and standard deviation of the input factor distribution, respectively [14]. One could also use another estimated mean, μ^*, proposed by Campolongo [2]. It is advised to use all three of these statistics, in order to obtain more sensitivity information at little extra computations. Calculation of the elementary effects is a sampling based approach. Here, we can sample input factors from the randomized sampling matrix $\mathbf{B}^*$, which can be formulated as

$$\mathbf{B}^* = (\mathbf{J}_{k+1,1}\mathbf{x}^* + (\Delta/2)[(2\mathbf{B} - \mathbf{J}_{k+1,1})\mathbf{D}^* + \mathbf{J}_{k+1,1}])\mathbf{P}^*, \tag{3.31}$$

where we have $\mathbf{J}_{k+1,1}$ as a $(k+1) \times k$ matrix of 1's, and x^* is some random value of $\mathbf{X}$. The diagonal matrix $\mathbf{D}^*$ is of k dimensions, with every element being either $+1$ or -1, and the $k \times k$ random permutation matrix is denoted $\mathbf{P}^*$, where every row contains one element equal to 1, while all others elements are 0, and there are no columns that have 1's in the same positions. The matrix B is a strictly lower triangular matrix of 1's [14].

Say that we have l in the set of $\{1, \ldots, k\}$, and then, if $\mathbf{x}^l$ and $\mathbf{x}^{l+1}$ are samples of the j'th trajectory, differing in their i'th component, the elementary effect of the factor i is

$$EE_i^j(\mathbf{x}^l) = \frac{y(\mathbf{x}^{l+1}) - y(\mathbf{x}^l)}{\Delta}, \tag{3.32}$$

if we have an increase of the i'th component of $\mathbf{x}^l$ by Δ, but

$$EE_i^j(\mathbf{x}^{l+1}) = \frac{y(\mathbf{x}^l) - y(\mathbf{x}^{l+1})}{\Delta}, \tag{3.33}$$

if we have an decrease of the i'th component of $\mathbf{x}^l$ by Δ. When we have r elementary effects per input factor available (EE_i^j, $i = 1, \ldots, k$, $j = 1, \ldots, r$), then μ_i, μ_i^* and σ_i^2, with respect to the distributions, can be computed for every input factor [14],

$$\mu_i = \frac{1}{r}\sum_{j=1}^{r} EE_i^j \tag{3.34a}$$

$$\mu_i^* = \frac{1}{r}\sum_{j=1}^{r} |EE_i^j| \tag{3.34b}$$

$$\sigma_i^2 = \frac{1}{r-1}\sum_{j=1}^{r}(EE_i^j - \mu)^2, \tag{3.34c}$$

in which EE_i^j denotes the elementary effects with respect to the input factor i, along the trajectory j. Now, based on eq. (3.1), we could rank importance of all the input factors [14].

3.3.2 Monte Carlo filtering

Another important method of GSA is the Monte Carlo filtering (MCF). Here, one refrains from identifying a optimal solution of the model output Y, but instead focus on mapping values of the input factors into the output space, then *filtering* out input corresponding to the unacceptable Y values[13]. Thus, elements of the MC sample that classifies as "good" realizations are flagged as *behavioural*, while elements that classifies as "bad" realizations are flagged as *non-behavioural*. The general idea is that, if the subsets "behavioural" and "non-behavioural" are not similar to one another, then the input factor is influential[14].

3.3.3 FORM/SORM

Sometimes we are not interested in the magnitude of the output Y (thus, neither variation), but rather in the probability of Y exceeding some critical value[13]. For instance, say that we have some constraint $Y - Y_{crit} \leq 0$, which gives a hypersurface in the space Ω of the input factors $\mathbf{X}$. Then, the quantity that we are rather interested in, would be the minimum distance between some design point for X and Ω. We denote this distance β for some joint distribution of the input factor $\mathbf{X}$. With a such setting, we can choose β with respect to X as the sensitivity measure. The first-order reliability method provides such measures.

In structural reliability, the first-order reliability method (FORM) and second-order reliability method (SORM) are considered amongst the most reliable methods. Generally, their accuracy depend on three parameters, i.e., (i) the curvature radius at the design point, (ii) the number of random variables, and (iii) the first-order reliability index[13].

FORM tries to identify a design point in Ω that gives the biggest possibility of failure. We denote each uncertain input factor as X_i, in which there are n uncertain factors in the model output Y, and $\mathbf{X}$ denotes the vector of all the n input factors. Then, we can define failure by the performance function $g(\mathbf{X})$. Failure would mean exceeding some critical value when the model is run. What differs FORM and SORM from one another, is that $g(\mathbf{X})$ is linear for FORM, but non-linear for SORM. Otherwise, these methods are alike, in which they utilize an optimization algorithm to identify the point that is the most likely for failure, taking into consideration the input factors and the performance function $g(\mathbf{X})$. When this point (i.e., the design point) is identified, some first-order (second-order) surface is fitted to the point in order for evaluating an approximated probability of failure[13].

Chapter 4

Optimization problem

4.1 The case study

The chosen case study is a fermentation process in a batch bioreactor. A simple flowsheet of this process is shown in Figure 4.1. The system consists of four states (X_s, S_s, P_s, V_s), one input (u) and seven parameters ($\mu_m, K_m, K_i, \nu, Y_p, Y_x, S_{in}$). There is only one flow of substrate feed entering the reactor, and it is assumed that the reactor is perfectly mixed under isothermal conditions. The overall objective of the process is to maximize product.

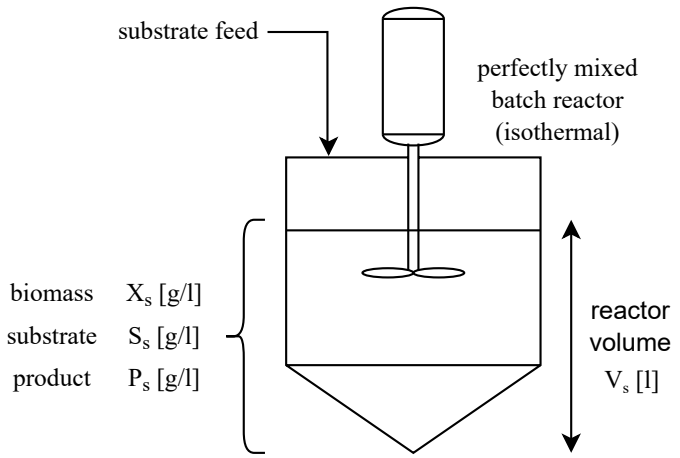

Figure 4.1: Simplified flowsheet of the batch bioreactor[8].

The process is described by the following ordinary differential equation (ODE)[8]:

$$\dot{X}_s = \mu(S_s)X_s - \frac{u}{V_s}X_s \tag{4.1a}$$

$$\dot{S}_s = -\frac{\mu(S_s)X_s}{Y_x} - \frac{\nu X_s}{Y_p} + \frac{u}{V_s}(S_{in} - S_s) \tag{4.1b}$$

$$\dot{P}_s = \nu X_s - \frac{u}{V_s}P_s \tag{4.1c}$$

$$\dot{V}_s = u \tag{4.1d}$$

where

$$\mu(S_s) = \frac{\mu_m S_s}{K_m + S_s + (S_s^2/K_i)} \tag{4.1e}$$

Here, $X_s[\mathrm{g/l}]$ is the biomass concentration, $S_s[\mathrm{g/l}]$ is the substrate concentration, $P_s[\mathrm{g/l}]$ is the product concentration, and $V_s[\mathrm{l}]$ is the reactor volume. The feed flow rate is denoted by $u[\mathrm{m^3/min}]$, and the auxiliary term, $\mu[\mathrm{min^{-1}}]$, is the specific growth rate. Furthermore, $\mu_m[\mathrm{min^{-1}}]$ is the maximum specific growth rate, $K_m[\mathrm{g/l}]$ is the saturation constant, $K_i[\mathrm{g/l}]$ is the inhibition constant, $\nu[\mathrm{g\ product/(g\ cells \cdot min)}]$ is the specific rate of product formation, $Y_p[-]$ is the product yield, $Y_p[-]$ is the biomass yield, and $S_{in}[\mathrm{g/l}]$ is the inlet substrate concentration. The initial values for the states and the input, along with the nominal values for the parameters, are shown in Table 4.1.

Table 4.1: Initial values for states and input, and nominal values for parameters[8].

Symbol	Initial value	Nominal Value	Unit
X_s	1.0		$[\mathrm{g/l}]$
S_s	0.5		$[\mathrm{g/l}]$
P_s	0.0		$[\mathrm{g/l}]$
V_s	120.0		$[\mathrm{l}]$
u	0.0081		$[\mathrm{m^3/min}]$
μ_m		0.02	$[\mathrm{min^{-1}}]$
K_m		0.05	$[\mathrm{g/l}]$
K_i		5.0	$[\mathrm{g/l}]$
ν		0.004	$[\frac{\mathrm{g\ product}}{\mathrm{g\ cells \cdot min}}]$
Y_p		1.2	$[-]$
Y_x		0.4	$[-]$
S_{in}		200.0	$[\mathrm{g/l}]$

4.1.1 Model predictive control

Using eq. (3.3) for this case study, we can formulate the economic MPC as

$$\min_{x,u} \sum_{k=0}^{n_p} -P_{s,k} + \sum_{k=1}^{n_m} \Delta u_k^T R \Delta u_k \tag{4.2a}$$

subject to

$$x_{k+1} = F(x_k, u_k, \theta_k), \qquad k = 0, \ldots, n_p - 1 \tag{4.2b}$$
$$g(x_k, u_k, \theta_k) \leq 0, \qquad k = 1, \ldots, n_p \tag{4.2c}$$
$$x_{min} \leq x_k \leq x_{max}, \qquad k = 1, \ldots, n_p \tag{4.2d}$$
$$u_{min} \leq u_k \leq u_{max}, \qquad k = 1, \ldots, n_m \tag{4.2e}$$
$$-\Delta u_{max} \leq \Delta u_k \leq \Delta u_{max}, \qquad k = 1, \ldots, n_m \tag{4.2f}$$

where

$$x_0 = x(0), \tag{4.2g}$$
$$\Delta u_k = u_k - u_{k-1}, \qquad k = 1, \ldots, n_m \tag{4.2h}$$
$$\Delta u_k = 0, \qquad k = n_m + 1, \ldots, n_p \tag{4.2i}$$

where the constraints on the outputs, inputs and input changes are shown in Table 4.2.

Table 4.2: Constraints on the outputs, inputs and input changes [8].

Symbol	Lower constraint	Upper constraint	Unit
X_s	0.0	3.7	[g/l]
S_s	0.0	∞	[g/l]
P_s	0.0	3.0	[g/l]
V_s	0.0	∞	[m^3]
u	0.0	0.2	[m^3/min]
Δu	-0.003	0.003	[m^3/min]

Closed-loop and open-loop control parameters are presented in the tables 4.3 and 4.4. We can now simulate the MPC, which can be done in *Python* through the use of *CasADi*, that is, an open-source tool for nonlinear optimization and algorithmic differentiation [3]. As presented in Section 2.2.1, we are using orthogonal collocation as an approach of the optimization problem, in which three Gauss-Radau collocation points per finite element was chosen. The natural choice of the solver for this MPC in *Python*, is the "Interior Point OPTimizer", or *Ipopt*, that uses a search filter method for identifying the local solution [5].

Table 4.3: Closed-loop control parameters.

Symbol	Value	Unit
n_p	20	$[-]$
n_m	3	$[-]$
R	1.0	$[-]$

Table 4.4: Open-loop control parameters.

Symbol	Value	Unit
n_p	150	$[-]$
n_m	150	$[-]$
R	1.0	$[-]$

When an optimal solution of the MPC is found, we plot the control inputs and the states against the discrete time-axis $t \in [0, 150]$, where each interval accounts for one minute. Then, we introduce uncertainty in the system, by sampling parameters θ_i from a uniform distribution, i.e., $\theta_i \sim U(95\%\,\theta_i, 105\%\,\theta_i)$, and doing this for each time t_k. We repeat this N number of times, and obtain N number of uncertainty plots, where we particularly want to study violations of the constraint on X_s, i.e., $X_s \leq 3.7$. If there are great violations of this constraint, it could be worth using sensitivity analysis and scenario-tree based MPC.

This case study was inspired by *do-mpc*[9], which is an open-source toolbox for RMPC, and the developers created an example, i.e., *Batch Bioreactor*, for this toolbox. This case is quite the same as ours. Here, they have implemented a scenario-tree based MPC with Y_x and S_{in} as the uncertain parameters. However, it is not mentioned why these are selected as the uncertain parameters. Neither does it seem like they have used SA, and the scenarios always seem based on Y_x and S_{in} [8]. Thus, we seek to SA for finding answers.

4.1.2 Sensitivity analysis

Of all the sensitivity analysis methods presented in Chapter 3, this project was restricted to implementation of only the Sobol' method. It seems reasonable to use Sobol' method for performing SA on the parameters θ_i with respect to the constraint on X_s, i.e., $X_s \leq 3.7$, as this is an important constraint not to violate. In particular, this regards the FP setting, because we are interested in identifying the parameter θ_i that accounts for the most of the output variance. This is linked with the first-order sensitivity indices S_i, which is what we obtain from the Sobol' method, together with the total-effect indices S_{T_i}. Now, the S_{T_i}'s are rather linked with the FF setting, where we are interested in identifying the parameters θ_i that contributes very little to the output variance[14]. As for now, we assume that both the S_i's and S_{T_i}'s could be valuable in the branching of scenario-trees, even though the S_i's seemingly are more important. Thus, we plot the S_i and S_{T_i} against the discrete time-axis $t \in [0, 150]$ in their respective figures. Here, we sample the uncertain parameters θ_i from a uniform distribution, i.e., $\theta_i \sim U(95\%\,\theta_i, 105\%\,\theta_i)$, with regards to the Latin hypercube sampling approach. The number of samples, N, is an important factor, and we plot S_i and S_{T_i} for an increasing N. Here, we use $N = 2^{15}$, $N = 2^{16}$ and $N = 2^{17}$, and plot using both regular and stacked (i.e., sensitivities are stacked) plots for the best visualization.

Chapter 5

Results and discussion

We have calculated the optimal control inputs and output trajectories for the closed-loop and open-loop MPC. This was done for both nominal parameters and uncertain parameters. The results are shown in Section 5.1 and Section 5.2, respectively. We have also computed Sobol' sensitivity indices for the open-loop MPC, in which the results are in Section 5.3. The scripts that were used for these simulations, are all presented in Appendix.

5.1 Closed-loop MPC

Figure 5.1 shows control inputs and output trajectories for the nominal closed-loop MPC.

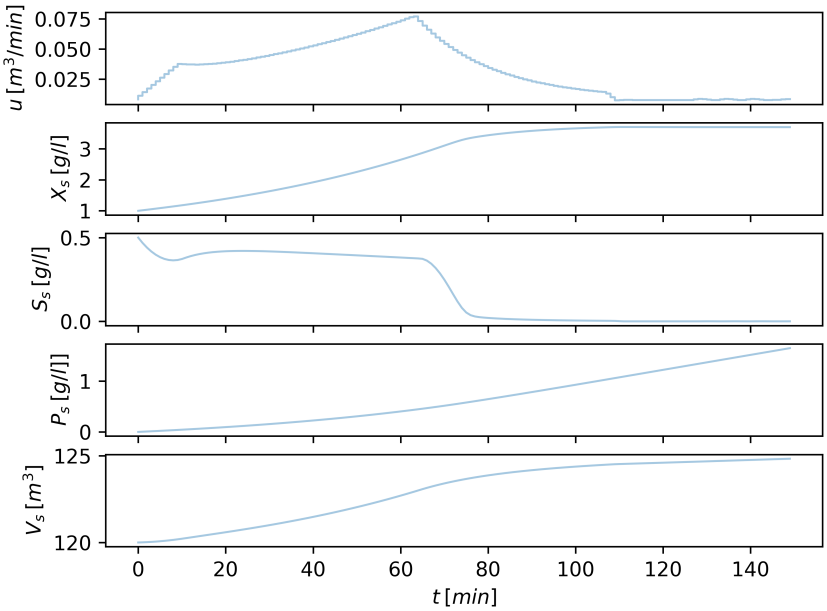

Figure 5.1: Input and output trajectories for the nominal closed-loop MPC.

Figure 5.2 shows control inputs and output trajectories for the uncertain closed-loop MPC. Here, we have randomly sampled the parameters θ_i from a uniform distribution, that is, $\theta_i \sim U(95\%\,\theta_i, 105\%\,\theta_i)$, at every time step, for $N = 100$ number of iterations.

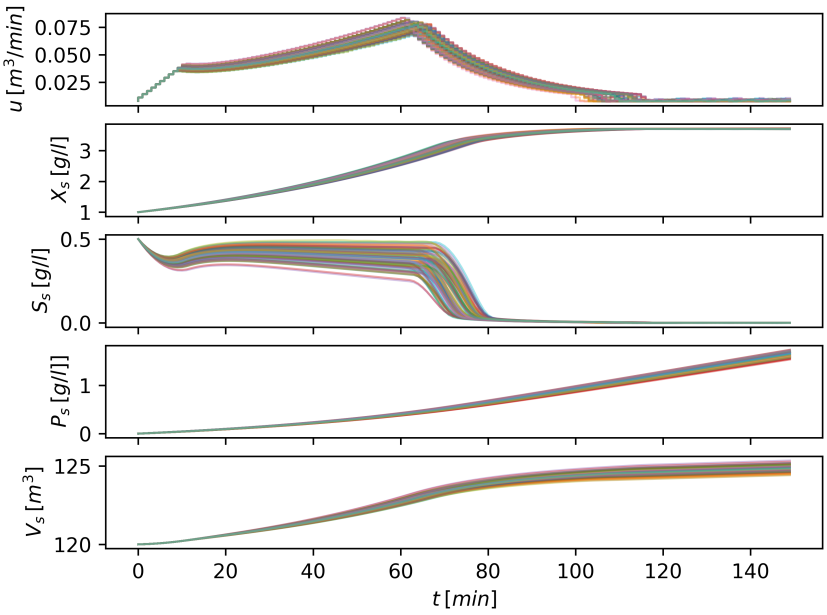

Figure 5.2: Input and output trajectories for the uncertain closed-loop MPC. N=100.

Figure 5.3 shows the output trajectories of biomass X_s for the uncertain closed-loop MPC. Here, we have randomly sampled the parameters θ_i from a uniform distribution, that is, $\theta_i \sim U(95\%\,\theta_i, 105\%\,\theta_i)$, at every time step, for $N = 100$ number of iterations.

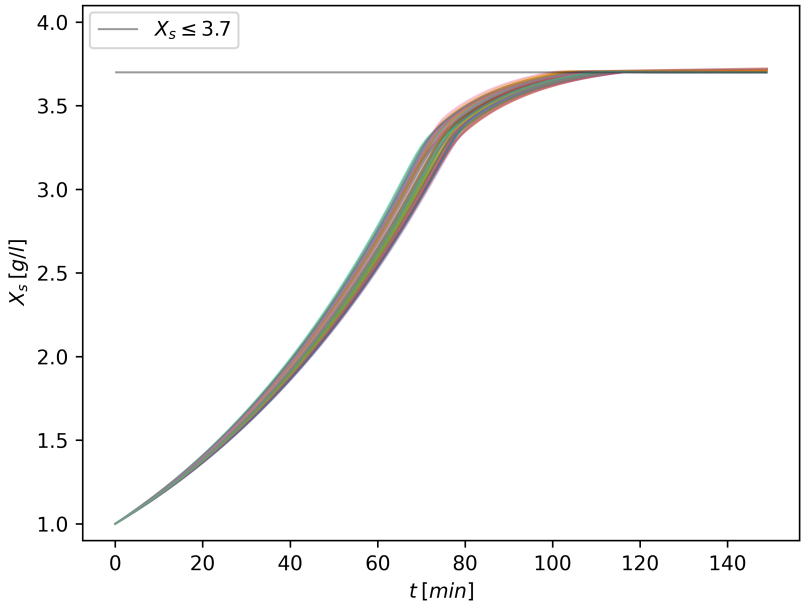

Figure 5.3: X_s output trajectories for the uncertain closed-loop MPC. N=100.

If we compare the control inputs and output trajectories in Figure 5.1 to the example case of *do-mpc*[8], we can confirm similarities. The control inputs u look somewhat alike, as well with the output trajectories for X_x, S_s, P_s and V_s. The relative "small" differences between these plots are due to the MPCs being different. In Figure 5.1, we used a nominal closed-loop MPC, while *do-mpc* had RMPC implemented. They considered uncertainty in the parameters Y_x and S_{in}, and implemented a scenario-tree based MPC with a robust horizon of 1 with 9 scenarios. Thus, it is natural that the plots look somewhat different.

In particular, it is the constraint on the biomass X_s that we are interested in studying, i.e., $X_s \leq 3.7$. In Figure 5.1 for the nominal closed-loop MPC, there are no uncertainties in the parameters, and thus, the constraint on X_s is satisfied. However, what if we do introduce uncertainty? This is done in Figure 5.2. Here, we recognize similarities with the control inputs and output trajectories from Figure 5.1, but as we now have uncertainties, the acquired trajectories vary quite a lot, e.g., look at the substrate S_s. However, it is still the biomass constraint that we are interested in studying, as it is a hard constraint.

Thus, in Figure 5.3, we have used the same results as calculated in Figure 5.2, but only focused on the X_s plot. From this figure, we observe that the biomass constraint $X_s \leq 3.7$ is violated. It is not violated by much, but it is being violated for several of the iterations. At the most, X_s takes the value of 3.723, which is not acceptable for our hard constraint. The solution would be to add back-off to the MPC, or we could make our MPC robust, e.g., doing a scenario-tree based approach, just like they did in the example of *do-mpc*[8]. If we were to include extra back-off to the MPC, this would ensure that we do not violate the constraint, but it could possibly be worse for performance if the back-off is too large. Hence, we seek to the scenario-tree based MPC for answers, but unlike that for *do-mpc*, we want to include SA in selecting the parameters for the scenario-trees. As the Sobol' method got quite computationally expensive, we instead tried implementing this for the open-loop MPC. We will say more about this in the next sections, but the concept remains the same for the open-loop; we want to find the parameter θ_i that is the most sensitive to the constraint on X_s. The results of this can be applied to a scenario-tree based MPC.

5.2 Open-loop MPC

Figure 5.4 shows control inputs and output trajectories for the nominal open-loop MPC.

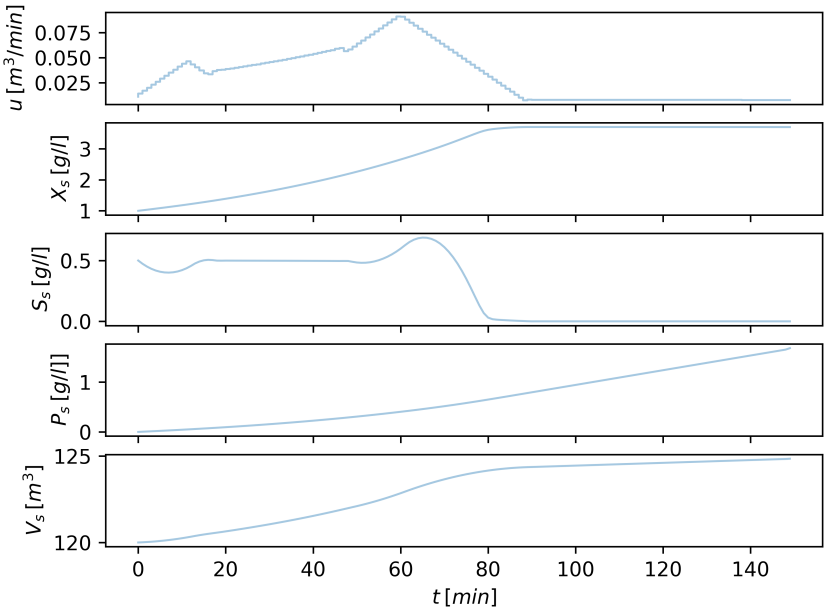

Figure 5.4: Input and output trajectories for the nominal open-loop MPC.

Figure 5.5 shows control inputs and output trajectories for the uncertain open-loop MPC. Here, we have randomly sampled the parameters θ_i from a uniform distribution, that is, $\theta_i \sim U(95\%\,\theta_i, 105\%\,\theta_i)$, at every time step, for $N = 100$ number of iterations.

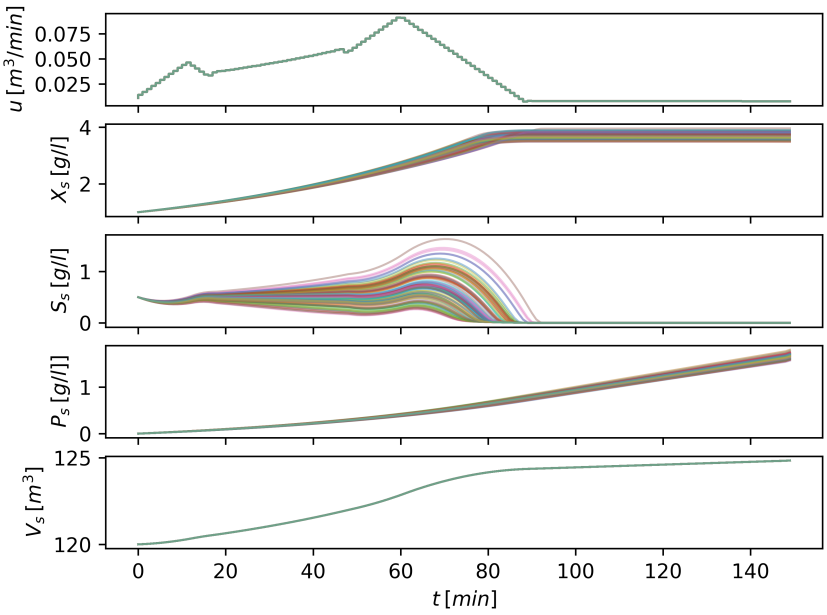

Figure 5.5: Input and output trajectories for the uncertain open-loop MPC. N=100.

Figure 5.6 shows the output trajectories of biomass X_s for the uncertain open-loop MPC. Here, we have randomly sampled the parameters θ_i from a uniform distribution, that is, $\theta_i \sim U(95\%\,\theta_i, 105\%\,\theta_i)$, at every time step, for $N = 100$ number of iterations.

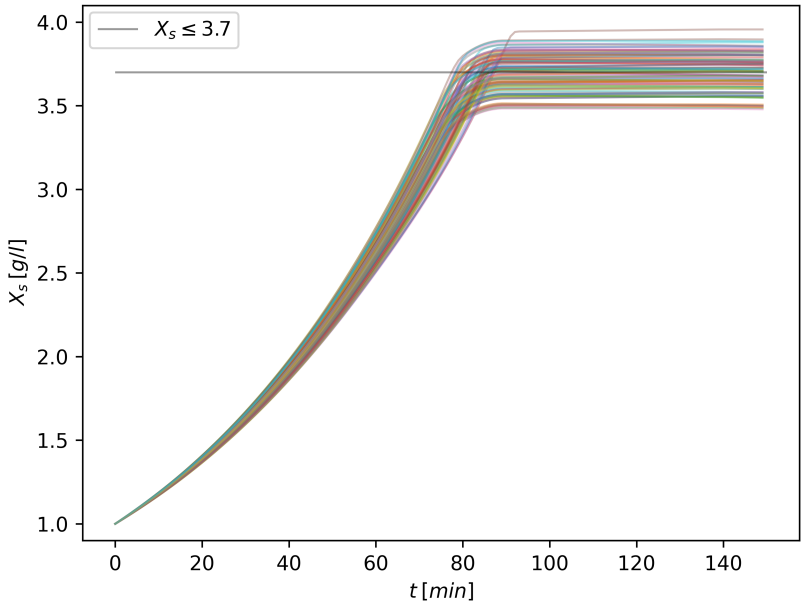

Figure 5.6: X_s output trajectories for the uncertain open-loop MPC. N=100.

Comparing Figure 5.4 with Figure 5.1, it is clear that the closed-loop MPC results in better performance than with the open-loop MPC. This is expected, since the closed-loop MPC has feedback to the plant, whilst the open-loop MPC does not. The control inputs and output trajectories in Figure 5.1 look more similar to the example of *do-mpc*, than for Figure 5.4. This is clear if we look at the control inputs u and the substrate trajectory S_s. Here, in Figure 5.4, the control actions are not as smooth as in Figure 5.1, and the substrate has an increase at around 60 minutes, which is not present for the closed-loop. However, it is still the constraint on X_s that we are interested in studying. In Figure 5.4, we observe that the constraint on X_s is satisfied, which is good. But once again, every model should account for uncertainty, and thus, we introduce uncertainty for the open-loop in Figure 5.5.

From observing the X_s trajectories in Figure 5.5, it is clear that the constraint on X_s, i.e., $X_s \leq 3.7$, is not satisfied. This is another reason of why the closed-loop MPC has better performance than the open-loop MPC. This is a hard constraint, meaning that we want it to be satisfied. Violations of such constraints are typically bad for the economics. In Figure 5.2, the largest violation was $X_s = 3.723$, but in Figure 5.5 we have the largest violation as $X_s = 3.948$. Hence, it is clear that feedback improves the MPC when being exposed to parametric uncertainty. This is illustrated better when comparing Figure 5.3 and Figure 5.6. We observe more deviation from the constraint for the open-loop MPC.

Likewise, as for the closed-loop MPC, the constraint violation is also unacceptable for the open-loop MPC, and to an even higher extent. The solution to this is either adding in a back-off or using a method of RMPC instead, or one could implement a combination of the two. In this project, we have focused on the RMPC solution with a scenario-tree based approach of the MPC. We want to use SA for selecting the uncertain parameter to be considered in the scenarios, and since the Sobol' method got computationally expensive, we decided to use SA on the open-loop MPC. By identifying the most sensitive parameter θ_i to the constraint on X_s, we get valuable information for the scenario-tree based MPC. Due to computational costs, it is not slightly efficient to base the scenario-trees on all the plant parameters. That is why we have to choose one uncertain parameter, or maybe two. We will talk more about the Sobol' method for the open-loop MPC in the next section.

5.3 Sensitivity Analysis

Figure 5.7 and Figure 5.8 show the first-order and total-effect Sobol' indices, respectively, when the number of samples N per parameter θ_i equals 2^{15}, and indices are not stacked.

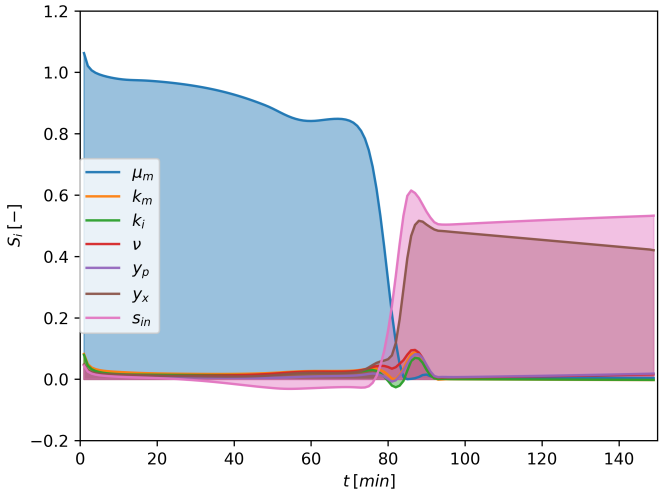

Figure 5.7: First-order Sobol' indices for the plant parameters. $N = 2^{15}$. Not stacked.

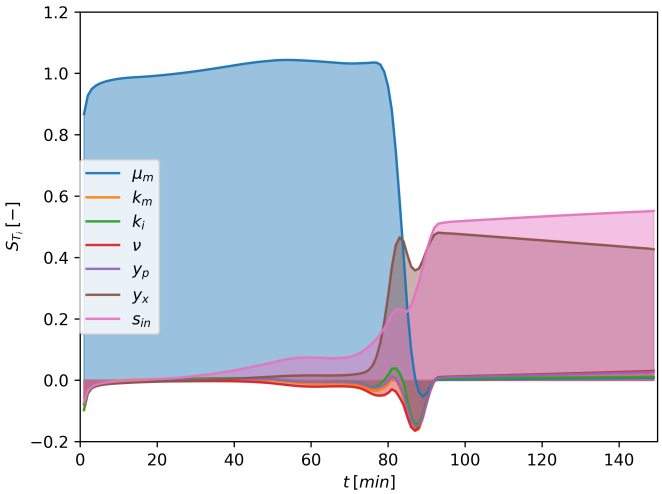

Figure 5.8: Total-effect Sobol' indices for the plant parameters. $N = 2^{15}$. Not stacked.

Figure 5.9 and Figure 5.10 show the first-order and total-effect Sobol' indices, respectively, when the number of samples N per parameter θ_i equals 2^{15}, and indices are stacked.

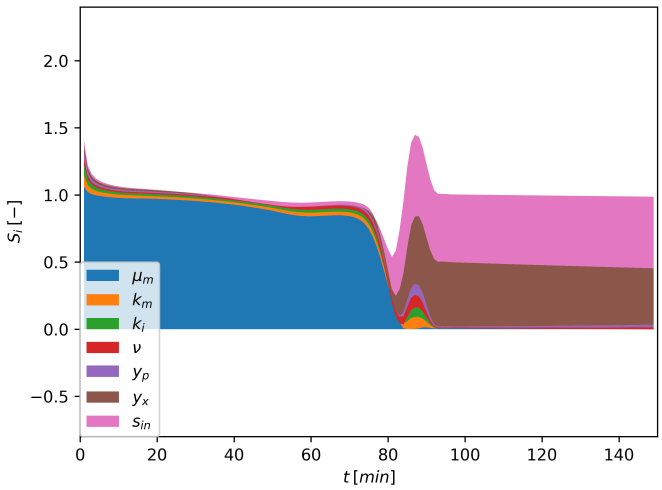

Figure 5.9: First-order Sobol' indices for the plant parameters. $N = 2^{15}$. Stacked.

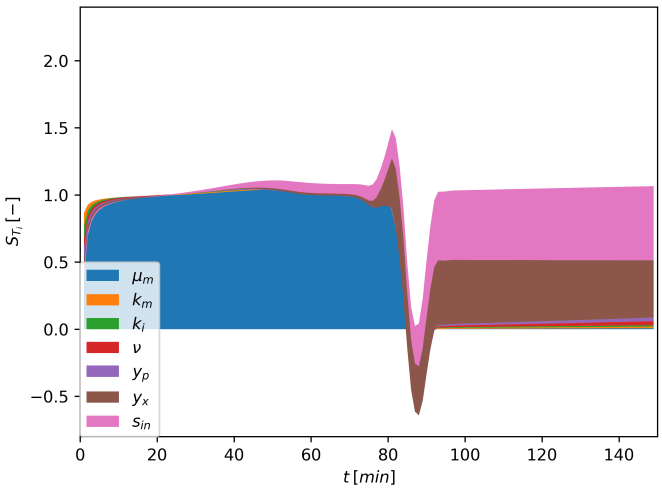

Figure 5.10: Total-effect Sobol' indices for the plant parameters. $N = 2^{15}$. Stacked.

Figure 5.11 and Figure 5.12 show first-order and total-effect Sobol' indices, respectively, when the number of samples N per parameter θ_i equals 2^{16}, and indices are not stacked.

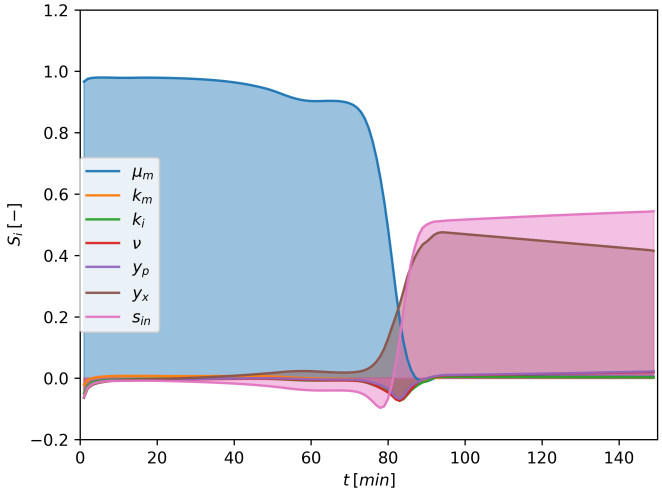

Figure 5.11: First-order Sobol' indices for the plant parameters. $N = 2^{16}$. Not stacked.

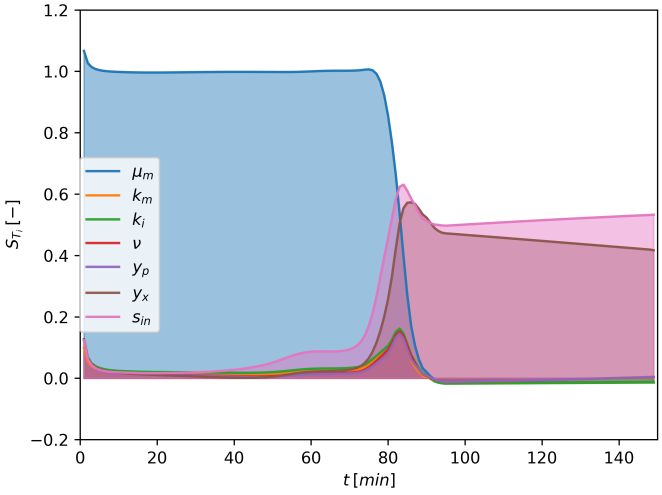

Figure 5.12: Total-effect Sobol' indices for the plant parameters. $N = 2^{16}$. Not stacked.

Figure 5.13 and Figure 5.14 show first-order and total-effect Sobol' indices, respectively, when the number of samples N per parameter θ_i equals 2^{16}, and indices are stacked.

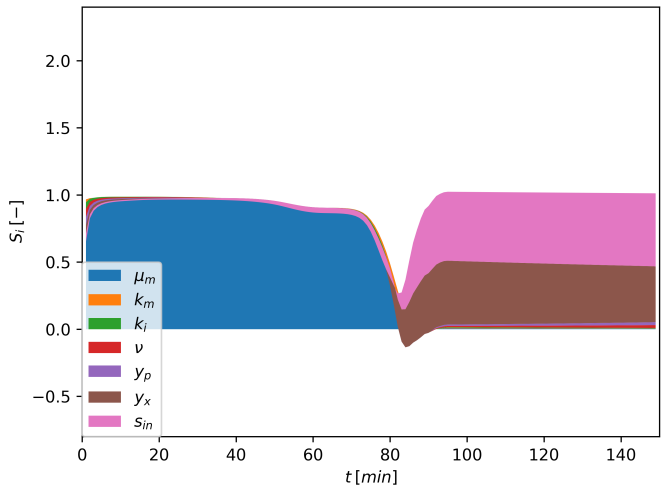

Figure 5.13: First-order Sobol' indices for the plant parameters. $N = 2^{16}$. Stacked.

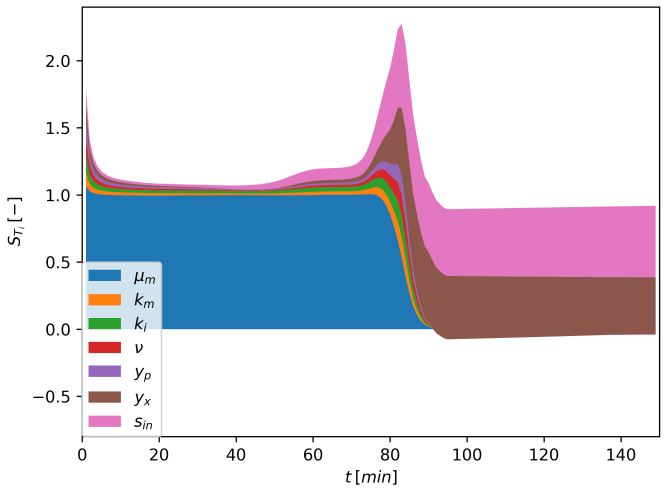

Figure 5.14: Total-effect Sobol' indices for the plant parameters. $N = 2^{16}$. Stacked.

Figure 5.15 and Figure 5.16 show first-order and total-effect Sobol' indices, respectively, when the number of samples N per parameter θ_i equals 2^{17}, and indices are not stacked.

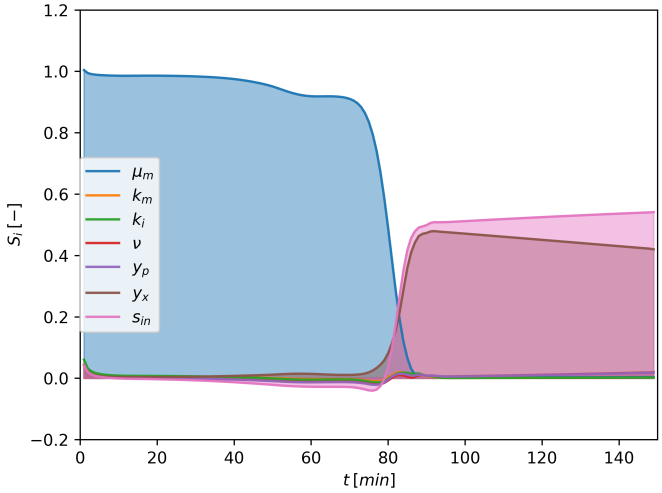

Figure 5.15: First-order Sobol' indices for the plant parameters. $N = 2^{17}$. Not stacked.

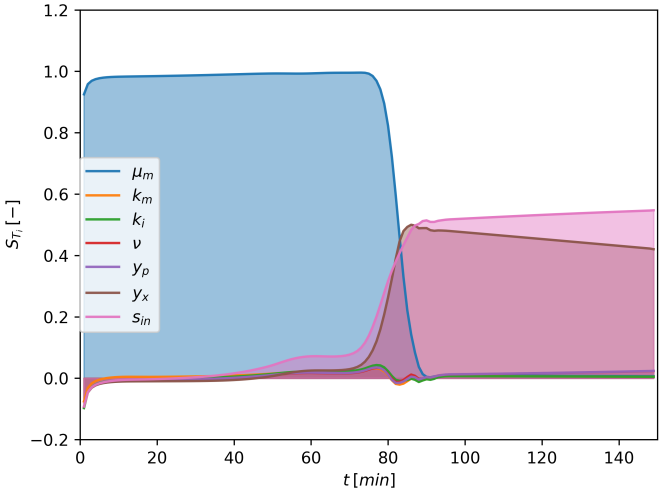

Figure 5.16: Total-effect Sobol' indices for the plant parameters. $N = 2^{17}$. Not stacked.

Figure 5.17 and Figure 5.18 show first-order and total-effect Sobol' indices, respectively, when the number of samples N per parameter θ_i equals 2^{17}, and indices are stacked.

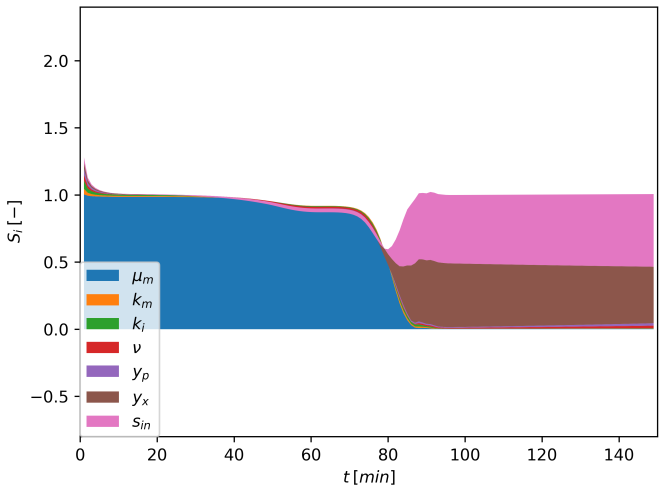

Figure 5.17: First-order Sobol' indices for the plant parameters. $N = 2^{17}$. Stacked.

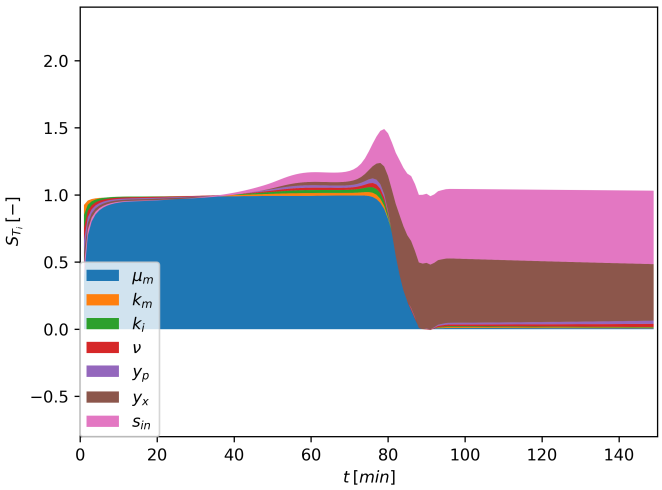

Figure 5.18: Total-effect Sobol' indices for the plant parameters. $N = 2^{17}$. Stacked.

The stacked plots are great for visualizing the importance of the sensitivity indices. The figures 5.9, 5.13 and 5.17 show the first-order Sobol' indices for $N = 2^{15}$, $N = 2^{16}$ and $N = 2^{17}$ number of samples, respectively. Here, it is clear that μ_m, Y_x and S_{in} are the most influential parameters on the X_s constraint, i.e., $X_s \leq 3.7$. We can somewhat verify this result by comparing with the uncertain parameters in the *do-mpc* example, which were Y_x and S_{in}. The reason behind why *do-mpc* did not include μ_m as an uncertain parameter, is probably due to computational expenses. However, this could also be due the fact that the influence of μ_m on X_s decreases a lot after about 80 minutes, and the influence of Y_x and S_{in} increases a lot during this time. That is, when the X_s predictions are getting closer to the constraint, it seems that the importance of Y_x and S_{in} increases. This is an interesting take, as it questions whether it is worth including μ_m in the scenario-trees. However, we presume that it is worth including, due to several uncertainty iterations in Figure 5.6 violating the constraint on X_s before 80 minutes. Another reason for including μ_m in the scenario-trees, is that the behavior of X_s before 80 minutes also is important, which has an effect on the later predictions after 80 minutes. Thus, the figures 5.9, 5.13 and 5.17 show that μ_m, Y_x and S_{in} are the most influential parameters on $X_s \leq 3.7$.

Just now, we talked about the most sensitive parameters, that being μ_m, Y_x and S_{in}. As mentioned in Section 4.1.2, the FP setting is linked with the S_i's and the FF setting is linked with the S_{T_i}'s. So far we have identified the parameters θ_i that account for the most of the X_s variance, according to the FP setting, but we have not yet identified the parameters θ_i that contribute very little to the X_s variance, according to the FF setting. Thus, the stacked plots of the total-effect Sobol' indices are shown in the figures 5.10, 5.14 and 5.18, for $N = 2^{15}$, $N = 2^{16}$ and $N = 2^{17}$ number of samples, respectively. Here, we observe that k_m, k_i, ν and Y_p are the parameters θ_i that contribute the least to the variance in X_s. Thus, we could have excluded these parameters for the scenario-trees, such that the computational expenses decreases.

We have talked about stacked plots for the first-order and total-effect Sobol' indices, but what about the non-stacked plots? These are not as illustrative when comparing the contribution from each parameter, but they are shown in order to illustrate the negative sensitivities that we get. It is so, that the first-order Sobol' indices should sum to 1, and that all indices should be non-negative[14]. If we look at all the stacked-plots for the first-order Sobol' indices, we can clearly see that they do not sum to 1 for each time, but they somewhat tend to either way. The reason behind this is that we have negative first-order sensitivities, which we can see from the figures 5.7, 5.11 and 5.15, for $N = 2^{15}$, $N = 2^{16}$ and $N = 2^{17}$ number of samples, respectively. In theory, this is not possible, but as the Sobol' method with Saltelli's modification is an approximation, we get negative indices from eq. (3.26) if we have $f_0^2 > y_A \cdot y_{C_i}$. That is, we can get negative signs if the Sobol' indices are close to zero (i.e., unimportant parameters). Increasing the number of samples N should give less probability of encountering negative sensitivities. Moreover, the total-effect indices should also be non-negative, and as seen in the figures 5.8, 5.12 and 5.16, they are not. Total-effect indices should always be greater or equal to first-order indices, which we see that are wrong in the stacked plots. This is due to negative first-order indices.

One might ask; ”if it is so that we get less negative Sobol’ indices for greater number of samples N, why would we not just increase N?”. As seen by comparing the figures 5.7-5.10 to the figures 5.11-5.14 to the figures 5.15-5.18, we have less negative Sobol’ indices for greater N. If we increased the number of samples to, e.g., $N = 2^{18}$, we would probably have even less negative indices. The reason why this is inefficient, is because of the computational expenses. Using the Sobol’ method on the open-loop MPC for $N = 2^{15}$ lasted approximately 1 hour, 42 minutes and 27 seconds. Also, for $N = 2^{16}$ it was 3 hours, 13 minutes and 8 seconds, and for $N = 2^{17}$ it was 6 hours, 46 minutes and 25 seconds.

Thus, using the Sobol’ method as SA of the parameters θ_i on the X_s constraint for the open-loop MPC, resulted in a good indicator of what θ_i’s that affects the output variation the most, and what θ_i’s that has little effect on the output variation. However, numerical errors of the estimation, i.e., negative sensitivities, meant that the Sobol’ method resulted in too unreasonable answers for being implemented in a scenario-tree based MPC.

Chapter 6

Conclusion

6.1 Conclusion

With respect to the constraint on the biomass X_s, i.e., $X_s \leq 3.7$, it was found that the closed-loop MPC outperformed the open-loop when parametric uncertainty was present. Both MPCs had violations of the constraint, but the greatest violation of the closed-loop (3.723) was smaller than of the open-loop (3.948). However, as this is a hard constraint, we should add back-off to the MPC or implement a method of RMPC, or a combination of the two. Here, we wanted to study the scenario-tree based MPC as a method of RMPC.

Due to computational expenses, we only want one, or maybe two or three, uncertain parameters to be considered for the scenario-trees. As the constraint on X_s is important to satisfy, we wanted to identify the parameters θ_i that is the most influential to the output variation for X_s, and the parameters θ_i that are the least influential on the output variation. We used Sobol' method as SA for this, and computed first-order and total-effect indices.

It was found that, from the first-order Sobol' indices, that μ_m, Y_x and S_{in} were the most sensitive parameters. From the total-effect indices we found that k_m, k_i, ν and Y_p were the least sensitive parameters. However, we can only use these results as qualitative indicators on sensitivity, as we had numerical errors due to getting negative Sobol' indices.

However, for an increasing amount of samples N, it was found that we obtained less negative indices. If we increased N even further, we would probably have obtained only non-negative sensitivities. The best result was acquired for $N = 2^{17}$, but this simulation lasted 6 hours, 46 minutes and 25 seconds. Hence, it was concluded that the computational expenses was too high; at least for our case study.

6.2 Further work

Further work on this project should include trying other methods of SA for the case study. Typically, this would be the Morris screening, Monte Carlo filtering and FORM/SORM, that were introduced in Section 3.3. Moreover, further work should also include trying to implement an algorithm for the scenario-tree based MPC. Only then may we know if it is worth the extra computational effort, instead of just using a large back-off for the MPC.

Bibliography

[1] Biegler, L.T., 2010. Nonlinear Programming: Concepts, Algorithms, and Applications to Chemical Processes. Society for Industrial and Applied Mathematics, Philadelphia.

[2] Campolongo, F., Cariboni, J., Saltelli, A., 2007. An effective screening design for sensitivity analysis of large models. Environmental Modelling and Software.

[3] CasADi, 2018. Build efficient optimal control software, with minimal effort. https://web.casadi.org/. Last accessed 22 December 2022.

[4] Foss, B., Heirung, T.A.N., 2016. Merging Optimization and Control. Norwegian University of Science and Technology.

[5] Ipopt, 2020. Ipopt documentation. https://coin-or.github.io/Ipopt/. Last accessed 22 December 2022.

[6] Krishnamoorthy, D., Skogestad, S., Jäschke, J., 2019. Multistage Model Predictive Control with Online Scenario Tree Update using Recursive Bayesian Weighting. Norwegian University of Science and Technology.

[7] Krishnamoorthy, D., Suwartadi, E., Foss, B., Skogestad, S., Jäschke, J., 2018. Improving Scenario Decomposition for Multistage MPC using a Sensitivity-based Path-following Algorithm. Norwegian University of Science and Technology.

[8] Lucia, S., Fiedler, F., 2021a. Batch bioreactor. https://www.do-mpc.com/en/latest/example_gallery/batch_reactor.html. Last accessed 22 December 2022.

[9] Lucia, S., Fiedler, F., 2021b. Model predictive control python toolbox. https://www.do-mpc.com/en/latest/index.html. Last accessed 22 December 2022.

[10] Lucia, S., Fiedler, F., 2021c. Orthogonal collocation on finite elements. https://www.do-mpc.com/en/latest/theory_orthogonal_collocation.html. Last accessed 22 December 2022.

[11] Qin, S.J., Badgwell, T.A., 2003. A survey of industrial model predictive control technology. Elsevier.

[12] Rawlings, J.B., Mayne, D.Q., Diehl, M.M., 2022. Model Predictive Control: Theory, Computation, and Design. Nob Hill Publishing.

[13] Saltelli, A., Chan, K., Scott, E.M., 2000. Sensitivity Analysis: Gauging the Worth of Scientific Models. Wiley.

[14] Saltelli, A., Ratto, M., Andres, T., Campolongo, F., Cariboni, J., Gatelli, D., Saisana, M., Tarantola, S., 2008. Global Sensitivity Analysis. Wiley.

[15] Seborg, D.E., Edgar, T.F., Mellichamp, D.A., III, F.J.D., 2017. Process Dynamics and Control. Wiley.

[16] Singh, S., Pavone, M., Slotine, J.J.E., 2016. Tube-Based MPC: a Contraction Theory Approach. Stanford Graduate Fellowship (SGF) and the King Abdulaziz City for Science and Technology (KACST).

[17] Sobol, I.M., 1990. Sensitivity estimates for nonlinear mathematical models. Matematicheskoe Modelirovanie.

Appendix

Code listings closed-loop MPC

Main file for closed-loop MPC without uncertainty (cl_wo_unc.py)

```
import os
import pathlib
import warnings
import numpy as np
import scipy as sc
import casadi as cd
from plant_cl import ode_model
from plant_cl import integrator
from plant_cl import optimizer
import matplotlib.pyplot as plt
from utilities import sample_normal
from utilities import sample_uniform

proj_dir = pathlib.Path(__file__).parent.parent.parent
data_dir = os.path.join(proj_dir, "data")
fpath_t = os.path.join(data_dir, "t.npy")
fpath_u = os.path.join(data_dir, "u.npy")
t, u = np.load(fpath_t), np.load(fpath_u)

plots_dir = os.path.join(proj_dir, "plots")
if not os.path.isdir(plots_dir):
    raise Exception("Exception: can't find path.")

image_dir = os.path.join(plots_dir, "cl_wo_unc")
if not os.path.isdir(image_dir):
    raise Exception("Exception: can't find path.")

u0 = u[0]  # Feed flow rate [m^{3}/min]
xs0 = 1.  # Concentration biomass [g/l]
ss0 = .5  # Concentration substrate [g/l]
ps0 = 0.  # Concentration product [g/l]
vs0 = 120.  # Total volume reactor [m^3]
x0 = np.array([xs0, ss0, ps0, vs0])

mu_m0 = .02  # Kinetic parameter constant guess [unit]
k_m0 = .05  # Kinetic parameter constant guess [unit]
k_i0 = 5.  # Kinetic parameter constant guess [unit
nu0 = .004  # Kinetic parameter constant guess [unit]
yp0 = 1.2  # Yield coefficient constant guess [unit]
yx0 = .4  # Yield coefficient constant guess [unit]
s_in0 = 200.  # Concentration substrate inlet [unit]
p0 = np.array([mu_m0, k_m0, k_i0, nu0, yp0, yx0, s_in0])

f = ode_model()  # Obtain ODE-model
dim_x, dim_u, dim_t = 4, 1, t.shape[0]
x_opts = np.zeros((dim_x, dim_t))
u_opts = np.zeros((dim_u, dim_t))

x_opts[:, 0] = x0.flatten()
u_opts[:, 0] = u0.flatten()

for k in range(1, dim_t):
    dt = np.array([t[k - 1], t[k]])
    uk = np.array(u_opts[:, k - 1])
```

```
    xk = np.array(x_opts[:, k - 1])
    pi = sample_uniform(0.95, 1.05, p0)
    u_opt = optimizer(f, xk, dt, uk, p0)
    x_opt = integrator(f, xk, dt, uk, p0)
    x_opts[:, k] = x_opt  # for plotting
    u_opts[:, k] = u_opt  # for plotting

fig1, ax1 = plt.subplots(5, 1, sharex='all')
plt_kwargs = {"linewidth": 1, "alpha": .4}
ax1[0].step(t, u_opts[0, :], **plt_kwargs)
ax1[1].plot(t, x_opts[0, :], **plt_kwargs)
ax1[2].plot(t, x_opts[1, :], **plt_kwargs)
ax1[3].plot(t, x_opts[2, :], **plt_kwargs)
ax1[4].plot(t, x_opts[3, :], **plt_kwargs)
ax1[0].set_ylabel(r"$u \: [m^{3}/min]$")
ax1[1].set_ylabel(r"$X_s \: [g/l]$")
ax1[2].set_ylabel(r"$S_s \: [g/l]$")
ax1[3].set_ylabel(r"$P_s \: [g/l]]$")
ax1[4].set_ylabel(r"$V_s \: [m^{3}]$")
ax1[4].set_xlabel(r"$t \: [min]$")

fpath_img = os.path.join(image_dir, "image1.png")
plt.savefig(fpath_img, dpi=600)  # save the plot
plt.show()  # plot in SciView
```

Listing 6.1: mpc_closed_loop / cl_wo_unc.py

Main file for closed-loop MPC with uncertainty (cl_w_unc.py)

```
import os
import pathlib
import warnings
import numpy as np
import scipy as sc
import casadi as cd
from plant_cl import ode_model
from plant_cl import integrator
from plant_cl import optimizer
import matplotlib.pyplot as plt
from utilities import sample_normal
from utilities import sample_uniform

proj_dir = pathlib.Path(__file__).parent.parent.parent
data_dir = os.path.join(proj_dir, "data")
fpath_t = os.path.join(data_dir, "t.npy")
fpath_u = os.path.join(data_dir, "u.npy")
t, u = np.load(fpath_t), np.load(fpath_u)

plots_dir = os.path.join(proj_dir, "plots")
if not os.path.isdir(plots_dir):
    raise Exception("Exception: can't find path.")

image_dir = os.path.join(plots_dir, "cl_w_unc")
if not os.path.isdir(image_dir):
    raise Exception("Exception: can't find path.")

u0 = u[0]  # Feed flow rate [m^{3}/min]
xs0 = 1.  # Concentration biomass [g/l]
ss0 = .5  # Concentration substrate [g/l]
ps0 = 0.  # Concentration product [g/l]
vs0 = 120.  # Total volume reactor [m^3]
x0 = np.array([xs0, ss0, ps0, vs0])

mu_m0 = .02  # Kinetic parameter constant guess [unit]
k_m0 = .05  # Kinetic parameter constant guess [unit]
k_i0 = 5.  # Kinetic parameter constant guess [unit
nu0 = .004  # Kinetic parameter constant guess [unit]
```

```
yp0 = 1.2  # Yield coefficient constant guess [unit]
yx0 = .4  # Yield coefficient constant guess [unit]
s_in0 = 200.  # Concentration substrate inlet [unit]
p0 = np.array([mu_m0, k_m0, k_i0, nu0, yp0, yx0, s_in0])

N = 100  # Number of samples taken
f = ode_model()  # Obtain ODE-model
dim_x, dim_u, dim_t = 4, 1, t.shape[0]
x_opts = np.zeros((dim_x, dim_t, N))
u_opts = np.zeros((dim_u, dim_t, N))

for i in range(N):
    x_opts[:, 0, i] = x0.flatten()
    u_opts[:, 0, i] = u0.flatten()
    pi = sample_uniform(.95, 1.05, p0)
    for k in range(1, dim_t):
        tk = np.array([t[k - 1], t[k]])
        uk = np.array(u_opts[:, k - 1, i])
        xk = np.array(x_opts[:, k - 1, i])
        u_opt = optimizer(f, xk, tk, uk, p0)
        x_opt = integrator(f, xk, tk, uk, pi)
        u_opts[:, k, i] = u_opt  # for plotting
        x_opts[:, k, i] = x_opt  # for plotting

fig1, ax1 = plt.subplots(5, 1, sharex='all')
plt_kwargs = {"linewidth": 1, "alpha": .4}
for j in range(N):
    ax1[0].step(t, u_opts[0, :, j], **plt_kwargs)
    ax1[1].plot(t, x_opts[0, :, j], **plt_kwargs)
    ax1[2].plot(t, x_opts[1, :, j], **plt_kwargs)
    ax1[3].plot(t, x_opts[2, :, j], **plt_kwargs)
    ax1[4].plot(t, x_opts[3, :, j], **plt_kwargs)
ax1[0].set_ylabel(r"$u \: [m^{3}/min]$")
ax1[1].set_ylabel(r"$X_s \: [g/l]$")
ax1[2].set_ylabel(r"$S_s \: [g/l]$")
ax1[3].set_ylabel(r"$P_s \: [g/l]]$")
ax1[4].set_ylabel(r"$V_s \: [m^{3}]$")
ax1[4].set_xlabel(r"$t \: [min]$")

fpath_img = os.path.join(image_dir, "image1.png")
plt.savefig(fpath_img, dpi=600)  # save the plot
plt.show()  # plot in SciView

fig2, ax2 = plt.subplots(1, 1, sharex='all')
plt_kwargs = {"linewidth": 1, "alpha": .4}
for j in range(N):
    ax2.plot(t, x_opts[0, :, j], **plt_kwargs)
plt.hlines(3.7, t[0], t[-1], **plt_kwargs,
           color='k', label=r"$X_s \leq 3.7$")
ax2.set_ylabel(r"$X_s \: [g/l]$")
ax2.set_xlabel(r"$t \: [min]$")
plt.legend(loc=(.01, .92))
plt.ylim([.9, 4.1])

fpath_img = os.path.join(image_dir, "image2.png")
plt.savefig(fpath_img, dpi=600)  # save the plot
plt.show()  # plot in SciView
```

Listing 6.2: mpc_closed_loop / cl_w_unc.py

File for the closed-loop MPC plant and optimizer (plant_cl.py)

```
import os
import pathlib
import warnings
import numpy as np
import scipy as sc
```

```
import casadi as cd

# Declaring states symbolic
xs = cd.SX.sym('xs', 1)
ss = cd.SX.sym('ss', 1)
ps = cd.SX.sym('ps', 1)
vs = cd.SX.sym('vs', 1)
x = cd.vertcat(xs, ss, ps, vs)

# Declaring inputs symbolic
u = cd.SX.sym('u', 1)

# Declaring MV-change symbolic
du = cd.SX.sym('du', 1)

# Declaring time-axis symbolic
t = cd.SX.sym('t', 1)

# Declaring parameters symbolic
mu_m = cd.SX.sym('mu_m', 1)
k_m = cd.SX.sym('k_m', 1)
k_i = cd.SX.sym('k_i', 1)
nu = cd.SX.sym('nu', 1)
yp = cd.SX.sym('yp', 1)
yx = cd.SX.sym('yx', 1)
s_in = cd.SX.sym('s_in', 1)
p = cd.vertcat(mu_m, k_m, k_i, nu, yp, yx, s_in)


# Defining the ODE-system
def ode_system():
    # Declaring the kinetic model
    mu = (mu_m * ss) / (k_m + ss + ((ss ** 2) / k_i))
    # Declaring the biomass equation
    dxs_dt = mu * xs - (u / vs) * xs
    # Declaring the substrate equation
    dss_dt = -(mu * xs) / yx - (nu * xs) / yp + (u / vs) * (s_in - ss)
    # Declaring the product equation
    dps_dt = nu * xs - (u / vs) * ps
    # Declaring the volume equation
    dvs_dt = u
    # Returning these ODEs together
    return cd.vertcat(dxs_dt, dss_dt, dps_dt, dvs_dt)


# Defining the ODE-model
def ode_model():
    sys = ode_system()
    p_aug = cd.vertcat(u, p, t)

    # Declaring the ODE-dictionary
    ode = {"x": x, "p": p_aug, "ode": sys * t}

    # Declaring options dictionary
    opts = {"max_num_steps": 200,
            "abstol": 1e-10, "reltol": 1e-10}

    # Declaring the ODE-integrator
    f = cd.integrator("F", "cvodes", ode, opts)

    # Returning the ODE-integrator
    return f


# Integrating the ODE-model
def integrator(f, x0, tk, u0, p0):
    x0 = cd.vertcat(x0)
    u0 = cd.vertcat(u0)
```

```
    dt = cd.vertcat(tk[1] - tk[0])
    pk = cd.vertcat(u0, p0, dt)
    fend = f(x0=x0, p=pk)
    xf_np = np.array(fend["xf"]).flatten()
    return xf_np


def optimizer(f, x0, tk, u0, p0):
    dg = 3  # orthogonal collocation with 3 points pr element
    tau_root = np.append(0, cd.collocation_points(dg, 'radau'))
    b = np.zeros((dg + 1, 1))
    c = np.zeros((dg + 1, dg + 1))
    d = np.zeros((dg + 1, 1))

    for i in range(dg + 1):
        coeff = 1
        # Construct Lagrange polynomials to get the
        # polynomial basis at the collocation point.
        for r in range(dg + 1):
            if r != i:
                coeff = np.convolve(coeff, [1., -tau_root[r]])
                coeff = coeff / (tau_root[i] - tau_root[r])

        # Evaluate the polynomial at the final time to
        # get coefficients of the continuity equation.
        d[i] = np.polyval(coeff, 1.)

        # Evaluate time derivative of the polynomial at all collocation
        # points to obtain the coefficients of the continuity equation.
        pder = np.polyder(coeff)
        for r in range(dg + 1):
            c[i][r] = np.polyval(pder, tau_root[r])

        # Evaluate the integral of the polynomial to
        # get coefficients of the quadrature function.
        pint = np.polyint(coeff)
        b[i] = np.polyval(pint, 1.)

    # Declare matrix for penalizing SP deviations
    q = np.array([[1., 0., 0., 0.],
                  [0., 0., 0., 0.],
                  [0., 0., 0., 0.],
                  [0., 0., 0., 0.]])

    # Declare matrix for penalizing MV movements
    r = 1.  # the same as do-MPC has used!

    # Declare the objective function
    j = (1 / 2) * (-ps + du.T @ r @ du)

    # Obtaining the ODE-system
    sys = ode_system()

    # Declare the CasADi function
    f = cd.Function('f', [x, u, p, t, du], [sys, j])

    # Declare w,w0,lbw,ubw,g,lbg,ubg,g1,lbg1,ubg1
    w, w0 = cd.vertcat([]), cd.vertcat([])
    g, g1 = cd.vertcat([]), cd.vertcat([])
    lbw, ubw = cd.vertcat([]), cd.vertcat([])
    lbg, ubg = cd.vertcat([]), cd.vertcat([])
    lbg1, ubg1 = cd.vertcat([]), cd.vertcat([])

    # Declare x_plt, u_plt for the plotting part
    x_plt, u_plt = cd.horzcat([]), cd.horzcat([])

    # Declare constraints; u_min, u_max, du_max
    u_min, u_max, du_max = 0., .2, .003
```

```
142
143     # Declare constraints; xs_max, ps_max
144     xs_max, ps_max = 3.7, 3.0
145
146     # Declare the horizons npr, nct
147     npr, nct = 20, 3
148
149     # Initialize the objective
150     j = 0
151
152     # Lift the initial conditions
153     xk = cd.MX.sym('x0', 4)
154     w = cd.vertcat(w, xk)
155     w0 = cd.vertcat(w0, x0)
156     lbw = cd.vertcat(lbw, x0)
157     ubw = cd.vertcat(ubw, x0)
158     x_plt = cd.horzcat(x_plt, xk)
159
160     for k in range(npr):
161         # New NLP variable for the control
162         uk = cd.MX.sym('u_' + str(k))
163         w = cd.vertcat(w, uk)
164         w0 = cd.vertcat(w0, u0)
165         lbw = cd.vertcat(lbw, u_min)
166         ubw = cd.vertcat(ubw, u_max)
167         u_plt = cd.horzcat(u_plt, uk)
168
169         # State at the collocation points
170         xki = [[] * i for i in range(dg)]
171         for i in range(dg):
172             xki[i] = cd.MX.sym('x_' + str(k) + '_' + str(i), 4)
173             w = cd.vertcat(w, xki[i])
174             w0 = cd.vertcat(w0, 1., .5, 0., 120.)
175             lbw = cd.vertcat(lbw, 0., 0., 0., 0.)
176             ubw = cd.vertcat(ubw, np.inf, np.inf, np.inf, np.inf)
177
178         # Loop over collocation points
179         xk_end = d[0] * xk
180
181         # If-sentence for finding duk
182         if k <= (nct - 1):
183             if k == 0:
184                 duk = uk - u0
185             else:
186                 duk = uk - uk0
187             g1 = cd.vertcat(g1, duk)
188             lbg1 = cd.vertcat(lbg1, -du_max)
189             ubg1 = cd.vertcat(ubg1, du_max)
190         else:
191             duk = uk - uk0
192             g1 = cd.vertcat(g1, duk)
193             lbg1 = cd.vertcat(lbg1, 0.)
194             ubg1 = cd.vertcat(ubg1, 0.)
195
196         # If-sentence for finding duk
197         if k != (npr - 1):
198             uk0 = uk
199         else:
200             uk0 = uk0
201
202         dt = tk[1] - tk[0]
203         for i in range(dg):
204             # Expression for state derivative at collocation point
205             xp = c[0, i + 1] * xk
206             for r in range(dg):
207                 xp += c[r + 1, i + 1] * xki[r]
208
209             # Append collocation equations
```

```
            fi, qi = f(xki[i], uk, p0, dt, 0)
            g = cd.vertcat(g, dt * fi - xp)
            lbg = cd.vertcat(lbg, 0., 0., 0., 0.)
            ubg = cd.vertcat(ubg, 0., 0., 0., 0.)

            # Add contribution to the end state
            xk_end += d[i + 1] * xki[i]

            # Add contribution to quad function
            j += b[i + 1] * qi * dt

        # New NLP variable for state at end
        xk = cd.MX.sym('x_' + str(k + 1), 4)
        w = cd.vertcat(w, xk)
        w0 = cd.vertcat(w0, 1., .5, 0., 120.)
        lbw = cd.vertcat(lbw, 0., 0., 0., 0.)
        ubw = cd.vertcat(ubw, xs_max, np.inf, ps_max, np.inf)
        x_plt = cd.horzcat(x_plt, xk)

        # Add equality constraint
        g = cd.vertcat(g, xk_end - xk)
        lbg = cd.vertcat(lbg, 0., 0., 0., 0.)
        ubg = cd.vertcat(ubg, 0., 0., 0., 0.)

    # Formalize it into an NLP problem
    prob = {'x': cd.vertcat(w), 'g': cd.vertcat(g, g1), 'f': j}

    # We may use an options dictionary
    opts = {'ipopt.print_level': 0, 'print_time': 0}

    # Assign solver - IPOPT in this case
    solver = cd.nlpsol('solver', 'ipopt', prob, opts)

    # Converting from CasADi MX to np.array
    w0 = np.array(w0).flatten()
    lbw = np.array(lbw).flatten()
    ubw = np.array(ubw).flatten()
    lbg = np.array(lbg).flatten()
    ubg = np.array(ubg).flatten()
    lbg1 = np.array(lbg1).flatten()
    ubg1 = np.array(ubg1).flatten()
    lbg2 = np.append(lbg, lbg1)
    ubg2 = np.append(ubg, ubg1)

    # Using cd.Function to get the x and u trajectories from w
    trajectories = cd.Function('trajectories', [w], [x_plt, u_plt], ['w'], ['x', 'u'])

    # Solve - using the previous defined initial guess and bounds
    sol = solver(x0=w0, lbx=lbw, ubx=ubw, lbg=lbg2, ubg=ubg2)
    x_opt, u_opt = trajectories(sol['x'])
    x_opt = x_opt.full()  # to numpy array
    u_opt = u_opt.full()  # to numpy array
    return u_opt[0][0]  # only first input
```

Listing 6.3: mpc_closed_loop / plant_cl.py

File for the closed-loop MPC utilities (utilities.py)

```
import os
import pathlib
import warnings
import numpy as np
import scipy as sc
import casadi as cd


def sample_normal(theta_nom):
```

```
    theta_sample = np.random.standard_normal(theta_nom.shape[0])
    return theta_sample


def sample_uniform(low, high, theta_nom):
    theta_low = low * theta_nom
    theta_high = high * theta_nom
    theta_sample = np.random.uniform(theta_low, theta_high, theta_nom.shape[0])
    return theta_sample
```

Listing 6.4: mpc_closed_loop / utilities.py

Code listings open-loop MPC

Main file for open-loop MPC without uncertainty (ol_wo_unc.py)

```
import os
import time
import pathlib
import warnings
import numpy as np
import scipy as sc
import casadi as cd
from plant_ol import ode_model
from plant_ol import integrator
from plant_ol import optimizer
from sobol import func
from sobol import sensitivity
import matplotlib.pyplot as plt
from sobol import uniform_sample

proj_dir = pathlib.Path(__file__).parent.parent.parent
data_dir = os.path.join(proj_dir, "data")
fpath_t = os.path.join(data_dir, "t.npy")
fpath_u = os.path.join(data_dir, "u.npy")
t, u = np.load(fpath_t), np.load(fpath_u)

plots_dir = os.path.join(proj_dir, "plots")
if not os.path.isdir(plots_dir):
    raise Exception("Exception: can't find path.")

image_dir = os.path.join(plots_dir, "ol_wo_unc")
if not os.path.isdir(image_dir):
    raise Exception("Exception: can't find path.")

u0 = u[0]  # Feed flow rate [m^{3}/min]
xs0 = 1.  # Concentration biomass [g/l]
ss0 = .5  # Concentration substrate [g/l]
ps0 = 0.  # Concentration product [g/l]
vs0 = 120.  # Total volume reactor [m^3]
x0 = np.array([xs0, ss0, ps0, vs0])

mu_m0 = .02  # Kinetic parameter constant guess [unit]
k_m0 = .05  # Kinetic parameter constant guess [unit]
k_i0 = 5.  # Kinetic parameter constant guess [unit
nu0 = .004  # Kinetic parameter constant guess [unit]
yp0 = 1.2  # Yield coefficient constant guess [unit]
yx0 = .4  # Yield coefficient constant guess [unit]
s_in0 = 200.  # Concentration substrate inlet [unit]
p0 = np.array([mu_m0, k_m0, k_i0, nu0, yp0, yx0, s_in0])

f = ode_model()  # Obtain ODE-model
tk = np.array([t[0], t[1]])  # t-diff
t = np.linspace(0, 150, 151)  # t-axis
u_opt = optimizer(f, x0, tk, u0, p0)
```

```
x_plt = integrator(f, x0, t, u_opt, p0)

fig1, ax1 = plt.subplots(5, 1, sharex='all')
plt_kwargs = {"linewidth": 1, "alpha": .4}
ax1[0].step(t[:-1], u_opt, **plt_kwargs)
ax1[1].plot(t[:-1], x_plt[0, :], **plt_kwargs)
ax1[2].plot(t[:-1], x_plt[1, :], **plt_kwargs)
ax1[3].plot(t[:-1], x_plt[2, :], **plt_kwargs)
ax1[4].plot(t[:-1], x_plt[3, :], **plt_kwargs)
ax1[0].set_ylabel(r"$u \: [m^{3}/min]$")
ax1[1].set_ylabel(r"$X_s \: [g/l]$")
ax1[2].set_ylabel(r"$S_s \: [g/l]$")
ax1[3].set_ylabel(r"$P_s \: [g/l]]$")
ax1[4].set_ylabel(r"$V_s \: [m^{3}]$")
ax1[4].set_xlabel(r"$t \: [min]$")

fpath_img = os.path.join(image_dir, "image1.png")
plt.savefig(fpath_img, dpi=600)  # save the plot
plt.show()  # plot in SciView

# If we want to compute the sensitivities:
# ---------------------------------------- #
# time_0 = time.time()  # tracks the Sobol time
# sis, stis = sensitivity(f, x0, t, u_opt, p0)
# time_f = time.time()  # tracks the Sobol time
# print(f"Calculation time: {time_f - time_0}")
# ---------------------------------------- #
```

Listing 6.5: mpc_open_loop / ol_wo_unc.py

Main file for open-loop MPC with uncertainty (ol_w_unc.py)

```
import os
import time
import pathlib
import warnings
import numpy as np
import scipy as sc
import casadi as cd
from plant_ol import ode_model
from plant_ol import integrator
from plant_ol import optimizer
from sobol import func
from sobol import sensitivity
import matplotlib.pyplot as plt
from sobol import uniform_sample

proj_dir = pathlib.Path(__file__).parent.parent.parent
data_dir = os.path.join(proj_dir, "data")
fpath_t = os.path.join(data_dir, "t.npy")
fpath_u = os.path.join(data_dir, "u.npy")
t, u = np.load(fpath_t), np.load(fpath_u)

plots_dir = os.path.join(proj_dir, "plots")
if not os.path.isdir(plots_dir):
    raise Exception("Exception: can't find path.")

image_dir = os.path.join(plots_dir, "ol_w_unc")
if not os.path.isdir(image_dir):
    raise Exception("Exception: can't find path.")

u0 = u[0]  # Feed flow rate [m^{3}/min]
xs0 = 1.  # Concentration biomass [g/l]
ss0 = .5  # Concentration substrate [g/l]
ps0 = 0.  # Concentration product [g/l]
vs0 = 120.  # Total volume reactor [m^3]
x0 = np.array([xs0, ss0, ps0, vs0])
```

```
36
37 mu_m0 = .02  # Kinetic parameter constant guess [unit]
38 k_m0 = .05  # Kinetic parameter constant guess [unit]
39 k_i0 = 5.  # Kinetic parameter constant guess [unit
40 nu0 = .004  # Kinetic parameter constant guess [unit]
41 yp0 = 1.2  # Yield coefficient constant guess [unit]
42 yx0 = .4  # Yield coefficient constant guess [unit]
43 s_in0 = 200.  # Concentration substrate inlet [unit]
44 p0 = np.array([mu_m0, k_m0, k_i0, nu0, yp0, yx0, s_in0])
45
46 N = 100  # Number of samples taken
47 f = ode_model()  # Obtain ODE-model
48 dim_x, dim_u, dim_t = 4, 1, t.shape[0]
49 tk = np.array([t[0], t[1]])  # t-diff
50 t = np.linspace(0, 150, 151)  # t-axis
51 u_opt = optimizer(f, x0, tk, u0, p0)
52 x_plt = np.zeros((dim_x, dim_t, N))
53 xs_plt = np.zeros((dim_t, N))
54
55 for i in range(N):
56     pi = uniform_sample(p0)  # sample random params
57     x_plt[:, :, i] = integrator(f, x0, t, u_opt, pi)
58     xs_plt[:, i] = func(f, x0, t, u_opt, pi)
59
60 fig1, ax1 = plt.subplots(5, 1, sharex='all')
61 plt_kwargs = {"linewidth": 1, "alpha": .4}
62 for j in range(N):
63     ax1[0].step(t[:-1], u_opt, **plt_kwargs)
64     ax1[1].plot(t[:-1], x_plt[0, :, j], **plt_kwargs)
65     ax1[2].plot(t[:-1], x_plt[1, :, j], **plt_kwargs)
66     ax1[3].plot(t[:-1], x_plt[2, :, j], **plt_kwargs)
67     ax1[4].plot(t[:-1], x_plt[3, :, j], **plt_kwargs)
68 ax1[0].set_ylabel(r"$u \: [m^{3}/min]$")
69 ax1[1].set_ylabel(r"$X_s \: [g/l]$")
70 ax1[2].set_ylabel(r"$S_s \: [g/l]$")
71 ax1[3].set_ylabel(r"$P_s \: [g/l]]$")
72 ax1[4].set_ylabel(r"$V_s \: [m^{3}]$")
73 ax1[4].set_xlabel(r"$t \: [min]$")
74
75 fpath_img = os.path.join(image_dir, "image1.png")
76 plt.savefig(fpath_img, dpi=600)  # save the plot
77 plt.show()  # plot in SciView
78
79 fig2, ax2 = plt.subplots(1, 1, sharex='all')
80 plt_kwargs = {"linewidth": 1, "alpha": .4}
81 for j in range(N):
82     ax2.plot(t[:-1], xs_plt[:, j], **plt_kwargs)
83 plt.hlines(3.7, t[0], t[-1], **plt_kwargs,
84            color='k', label=r"$X_s \leq 3.7$")
85 ax2.set_ylabel(r"$X_s \: [g/l]$")
86 ax2.set_xlabel(r"$t \: [min]$")
87 plt.legend(loc=(.01, .92))
88 plt.ylim([.9, 4.1])
89
90 fpath_img = os.path.join(image_dir, "image2.png")
91 plt.savefig(fpath_img, dpi=600)  # save the plot
92 plt.show()  # plot in SciView
93
94 # If we want to compute the sensitivities:
95 # time_0 = time.time()  # tracks the Sobol time
96 # sis, stis = sensitivity(f, x0, t, u_opt, p0)
97 # time_f = time.time()  # tracks the Sobol time
98 # print(f"Calculation time: {time_f - time_0}")
```

Listing 6.6: mpc_open_loop / ol_w_unc.py

Main file for the open-loop MPC sensitivities (sensitivity.py)

```
import os
import time
import pathlib
import numpy as np
import casadi as cd
import scipy.stats as sc
import matplotlib.pyplot as plt

proj_dir = pathlib.Path(__file__).parent.parent.parent
data_dir = os.path.join(proj_dir, "data")
fpath_sis = os.path.join(data_dir, "unif_131072_sis.npy")
fpath_stis = os.path.join(data_dir, "unif_131072_stis.npy")
sis, stis = np.load(fpath_sis), np.load(fpath_stis)

plots_dir = os.path.join(proj_dir, "plots")
if not os.path.isdir(plots_dir):
    raise Exception("Exception: can't find path.")

image_dir = os.path.join(plots_dir, "sensitivity")
if not os.path.isdir(image_dir):
    raise Exception("Exception: can't find path.")

ts = np.linspace(0, 150, 151)  # declare the time-axis
clr = (plt.rcParams['axes.prop_cycle'].by_key()['color'])
labels = [r"$\mu_{m}$", r"$k_{m}$", r"$k_{i}$", r"$\nu$",
          r"$y_{p}$", r"$y_{x}$", r"$s_{in}$"]

fig1, ax1 = plt.subplots(1, 1, sharex='all')
for i in range(len(labels)):
    ax1.plot(ts[1:-1], sis[i, 1:], color=clr[i], alpha=.95, label=labels[i])
    ax1.stackplot(ts[1:-1], sis[i, 1:], color=clr[i], alpha=.45)
ax1.set_ylabel(r"$S_{i} \: [-]$")
ax1.set_xlabel(r"$t \: [min]$")
plt.legend(loc=(0., .25))
plt.ylim([-.2, 1.2])
plt.xlim([0., 150.])

fpath_img = os.path.join(image_dir, "image1.png")
plt.savefig(fpath_img, dpi=600)  # save the plot
plt.show()  # plot in SciView

fig2, ax2 = plt.subplots(1, 1, sharex='all')
for i in range(len(labels)):
    ax2.plot(ts[1:-1], stis[i, 1:], color=clr[i], alpha=.95, label=labels[i])
    ax2.stackplot(ts[1:-1], stis[i, 1:], color=clr[i], alpha=.45)
ax2.set_ylabel(r"$S_{T_{i}} \: [-]$")
ax2.set_xlabel(r"$t \: [min]$")
plt.legend(loc=(0., .25))
plt.ylim([-.2, 1.2])
plt.xlim([0., 150.])

fpath_img = os.path.join(image_dir, "image2.png")
plt.savefig(fpath_img, dpi=600)  # save the plot
plt.show()  # plot in SciView

fig3, ax3 = plt.subplots(1, 1, sharex='all')
ax3.stackplot(ts[1:-1], sis[0, 1:], sis[1, 1:], sis[2, 1:], sis[3, 1:],
              sis[4, 1:], sis[5, 1:], sis[6, 1:], labels=labels)
ax3.set_ylabel(r"$S_{i} \: [-]$")
ax3.set_xlabel(r"$t \: [min]$")
plt.legend(loc=(0., 0.))
plt.ylim([-.8, 2.4])
plt.xlim([0., 150.])

fpath_img = os.path.join(image_dir, "image3.png")
```

```
plt.savefig(fpath_img, dpi=600)  # save the plot
plt.show()  # plot in SciView

fig4, ax4 = plt.subplots(1, 1, sharex='all')
ax4.stackplot(ts[1:-1], stis[0, 1:], stis[1, 1:], stis[2, 1:], stis[3, 1:],
              stis[4, 1:], stis[5, 1:], stis[6, 1:], labels=labels)
ax4.set_ylabel(r"$S_{T_{i}} \: [-]$")
ax4.set_xlabel(r"$t \: [min]$")
plt.legend(loc=(0., 0.))
plt.ylim([-.8, 2.4])
plt.xlim([0., 150.])

fpath_img = os.path.join(image_dir, "image4.png")
plt.savefig(fpath_img, dpi=600)  # save the plot
plt.show()  # plot in SciView
```

Listing 6.7: mpc_open_loop / sensitivity.py

File for the open-loop MPC plant and optimizer (plant_ol.py)

```
import os
import time
import pathlib
import numpy as np
import casadi as cd
import scipy.stats as sc
import matplotlib.pyplot as plt

# Declaring states symbolic
xs = cd.SX.sym('xs', 1)
ss = cd.SX.sym('ss', 1)
ps = cd.SX.sym('ps', 1)
vs = cd.SX.sym('vs', 1)
x = cd.vertcat(xs, ss, ps, vs)

# Declaring inputs symbolic
u = cd.SX.sym('u', 1)

# Declaring MV-change symbolic
du = cd.SX.sym('du', 1)

# Declaring time-axis symbolic
t = cd.SX.sym('t', 1)

# Declaring parameters symbolic
mu_m = cd.SX.sym('mu_m', 1)
k_m = cd.SX.sym('k_m', 1)
k_i = cd.SX.sym('k_i', 1)
nu = cd.SX.sym('nu', 1)
yp = cd.SX.sym('yp', 1)
yx = cd.SX.sym('yx', 1)
s_in = cd.SX.sym('s_in', 1)
p = cd.vertcat(mu_m, k_m, k_i, nu, yp, yx, s_in)


# Defining the ODE-system
def ode_system():
    # Declaring the kinetic model
    mu = (mu_m * ss) / (k_m + ss + ((ss ** 2) / k_i))
    # Declaring the biomass equation
    dxs_dt = mu * xs - (u / vs) * xs
    # Declaring the substrate equation
    dss_dt = -(mu * xs) / yx - (nu * xs) / yp + (u / vs) * (s_in - ss)
    # Declaring the product equation
    dps_dt = nu * xs - (u / vs) * ps
    # Declaring the volume equation
    dvs_dt = u
```

```
    # Returning these ODEs together
    return cd.vertcat(dxs_dt, dss_dt, dps_dt, dvs_dt)


# Defining the ODE-model
def ode_model():
    sys = ode_system()
    p_aug = cd.vertcat(u, p, t)

    # Declaring the ODE-dictionary
    ode = {"x": x, "p": p_aug, "ode": sys * t}

    # Declaring options dictionary
    opts = {"max_num_steps": 200,
            "abstol": 1e-10, "reltol": 1e-10}

    # Declaring the ODE-integrator
    f = cd.integrator("F", "cvodes", ode, opts)

    # Returning the ODE-integrator
    return f


# Integrating the ODE-model
def integrator(f, x0, tk, u0, p0):
    x_dim = x0.shape[0]
    t_dim = tk.shape[0]
    u_dim = u0.shape[0]
    p_dim = p0.shape[0]
    assert u_dim == t_dim - 1

    x0 = cd.vertcat(x0)
    u0 = cd.vertcat(u0)
    p0 = cd.vertcat(p0)
    dt = np.zeros(t_dim - 1)
    for i in range(t_dim - 1):
        dt[i] = tk[i + 1] - tk[i]
    dt = cd.vertcat(dt)

    xfs = np.zeros((x_dim, t_dim - 1))
    xfs[:, 0] = np.array(x0).flatten()
    for i in range(t_dim - 1):
        pk = cd.vertcat(u0[i], p0, dt[i])
        ff = f(x0=xfs[:, i], p=pk)
        xf = np.array(ff["xf"]).flatten()
        if i < (t_dim - 2):
            xfs[:, i + 1] = xf
        else:
            xfs[:, i] = xf
    return xfs


def optimizer(f, x0, tk, u0, p0):
    dg = 3  # orthogonal collocation with 3 points pr element
    tau_root = np.append(0, cd.collocation_points(dg, 'radau'))
    b = np.zeros((dg + 1, 1))
    c = np.zeros((dg + 1, dg + 1))
    d = np.zeros((dg + 1, 1))

    for i in range(dg + 1):
        coeff = 1
        # Construct Lagrange polynomials to get the
        # polynomial basis at the collocation point.
        for r in range(dg + 1):
            if r != i:
                coeff = np.convolve(coeff, [1., -tau_root[r]])
                coeff = coeff / (tau_root[i] - tau_root[r])

```

```
        # Evaluate the polynomial at the final time to
        # get coefficients of the continuity equation.
        d[i] = np.polyval(coeff, 1.)

        # Evaluate time derivative of the polynomial at all collocation
        # points to obtain the coefficients of the continuity equation.
        pder = np.polyder(coeff)
        for r in range(dg + 1):
            c[i][r] = np.polyval(pder, tau_root[r])

        # Evaluate the integral of the polynomial to
        # get coefficients of the quadrature function.
        pint = np.polyint(coeff)
        b[i] = np.polyval(pint, 1.)

    # Declare matrix for penalizing SP deviations
    q = np.array([[1., 0., 0., 0.],
                  [0., 0., 0., 0.],
                  [0., 0., 0., 0.],
                  [0., 0., 0., 0.]])

    # Declare matrix for penalizing MV movements
    r = 1.  # the same as do-MPC has used!

    # Declare the objective function
    j = (1 / 2) * (-ps + du.T @ r @ du)

    # Obtaining the ODE-system
    sys = ode_system()

    # Declare the CasADi function
    f = cd.Function('f', [x, u, p, t, du], [sys, j])

    # Declare w,w0,lbw,ubw,g,lbg,ubg,g1,lbg1,ubg1
    w, w0 = cd.vertcat([]), cd.vertcat([])
    g, g1 = cd.vertcat([]), cd.vertcat([])
    lbw, ubw = cd.vertcat([]), cd.vertcat([])
    lbg, ubg = cd.vertcat([]), cd.vertcat([])
    lbg1, ubg1 = cd.vertcat([]), cd.vertcat([])

    # Declare x_plt, u_plt for the plotting part
    x_plt, u_plt = cd.horzcat([]), cd.horzcat([])

    # Declare constraints; u_min, u_max, du_max
    u_min, u_max, du_max = 0., .2, .003

    # Declare constraints; xs_max, ps_max
    xs_max, ps_max = 3.7, 3.0

    # Declare the horizons npr, nct
    npr, nct = 150, 150

    # Initialize the objective
    j = 0

    # Lift the initial conditions
    xk = cd.MX.sym('x0', 4)
    w = cd.vertcat(w, xk)
    w0 = cd.vertcat(w0, x0)
    lbw = cd.vertcat(lbw, x0)
    ubw = cd.vertcat(ubw, x0)
    x_plt = cd.horzcat(x_plt, xk)

    for k in range(npr):
        # New NLP variable for the control
        uk = cd.MX.sym('u_' + str(k))
        w = cd.vertcat(w, uk)
        w0 = cd.vertcat(w0, u0)
```

```
        lbw = cd.vertcat(lbw, u_min)
        ubw = cd.vertcat(ubw, u_max)
        u_plt = cd.horzcat(u_plt, uk)

        # State at the collocation points
        xki = [[] * i for i in range(dg)]
        for i in range(dg):
            xki[i] = cd.MX.sym('x_' + str(k) + '_' + str(i), 4)
            w = cd.vertcat(w, xki[i])
            w0 = cd.vertcat(w0, 1., .5, 0., 120.)
            lbw = cd.vertcat(lbw, 0., 0., 0., 0.)
            ubw = cd.vertcat(ubw, np.inf, np.inf, np.inf, np.inf)

        # Loop over collocation points
        xk_end = d[0] * xk

        # If-sentence for finding duk
        if k <= (nct - 1):
            if k == 0:
                duk = uk - u0
            else:
                duk = uk - uk0
            g1 = cd.vertcat(g1, duk)
            lbg1 = cd.vertcat(lbg1, -du_max)
            ubg1 = cd.vertcat(ubg1, du_max)
        else:
            duk = uk - uk0
            g1 = cd.vertcat(g1, duk)
            lbg1 = cd.vertcat(lbg1, 0.)
            ubg1 = cd.vertcat(ubg1, 0.)

        # If-sentence for finding duk
        if k != (npr - 1):
            uk0 = uk
        else:
            uk0 = uk0

        dt = tk[1] - tk[0]
        for i in range(dg):
            # Expression for state derivative at collocation point
            xp = c[0, i + 1] * xk
            for r in range(dg):
                xp += c[r + 1, i + 1] * xki[r]

            # Append collocation equations
            fi, qi = f(xki[i], uk, p0, dt, 0)
            g = cd.vertcat(g, dt * fi - xp)
            lbg = cd.vertcat(lbg, 0., 0., 0., 0.)
            ubg = cd.vertcat(ubg, 0., 0., 0., 0.)

            # Add contribution to the end state
            xk_end += d[i + 1] * xki[i]

            # Add contribution to quad function
            j += b[i + 1] * qi * dt

        # New NLP variable for state at end
        xk = cd.MX.sym('x_' + str(k + 1), 4)
        w = cd.vertcat(w, xk)
        w0 = cd.vertcat(w0, 1., .5, 0., 120.)
        lbw = cd.vertcat(lbw, 0., 0., 0., 0.)
        ubw = cd.vertcat(ubw, xs_max, np.inf, ps_max, np.inf)
        x_plt = cd.horzcat(x_plt, xk)

        # Add equality constraint
        g = cd.vertcat(g, xk_end - xk)
        lbg = cd.vertcat(lbg, 0., 0., 0., 0.)
        ubg = cd.vertcat(ubg, 0., 0., 0., 0.)
```

```
    # Formalize it into an NLP problem
    prob = {'x': cd.vertcat(w), 'g': cd.vertcat(g, g1), 'f': j}

    # We may use an options dictionary
    opts = {'ipopt.print_level': 0, 'print_time': 0}

    # Assign solver - IPOPT in this case
    solver = cd.nlpsol('solver', 'ipopt', prob, opts)

    # Converting from CasADi MX to np.array
    w0 = np.array(w0).flatten()
    lbw = np.array(lbw).flatten()
    ubw = np.array(ubw).flatten()
    lbg = np.array(lbg).flatten()
    ubg = np.array(ubg).flatten()
    lbg1 = np.array(lbg1).flatten()
    ubg1 = np.array(ubg1).flatten()
    lbg2 = np.append(lbg, lbg1)
    ubg2 = np.append(ubg, ubg1)

    # Using cd.Function to get the x and u trajectories from w
    trajectories = cd.Function('trajectories', [w], [x_plt, u_plt], ['w'], ['x', 'u'])

    # Solve - using the previous defined initial guess and bounds
    sol = solver(x0=w0, lbx=lbw, ubx=ubw, lbg=lbg2, ubg=ubg2)
    x_opt, u_opt = trajectories(sol['x'])
    x_opt = x_opt.full()  # to numpy array
    u_opt = u_opt.full()  # to numpy array
    return u_opt[0, :nct]  # input-sequence
```

Listing 6.8: mpc_open_loop / plant_ol.py

File for the open-loop MPC Sobol' indices (sobol.py)

```
import os
import time
import pathlib
import numpy as np
import casadi as cd
import scipy.stats as sc
from plant_ol import integrator
import matplotlib.pyplot as plt


# Define function
def func(f, x0, tk, u0, p0):
    return integrator(f, x0, tk, u0, p0)[0]


def uniform_bounds(p0):
    p_low = 0.95 * p0
    p_high = 1.05 * p0
    return p_low, p_high


def uniform_sample(p0):
    p_dim = p0.shape[0]
    p_low, p_high = uniform_bounds(p0)
    return np.random.uniform(p_low, p_high, p_dim)


def uniform_dist(p0):
    p_dim = p0.shape[0]
    p_low, p_high = uniform_bounds(p0)
    p_dist = [sc.uniform(loc=p_low[i], scale=(p_high[i] - p_low[i])) for i in range(p_dim)]
    return p_dist
```

```
33
34
35  def sensitivity(f, x0, tk, u0, p0):
36      x_dim = x0.shape[0]
37      t_dim = tk.shape[0]
38      u_dim = u0.shape[0]
39      p_dim = p0.shape[0]
40      assert u_dim == t_dim - 1
41      p_dist = uniform_dist(p0)
42
43      N = 2 ** 6  # ~= 20 sec
44      # N = 2 ** 8  # ~= 55 sec
45      # N = 2 ** 13  # ~= 28 min
46      # N = 2 ** 15  # ~= 6146.46780371666s
47      # N = 2 ** 16  # ~= 11587.509873390198s
48      # N = 2 ** 17  # ~= 24384.82639169693s
49
50      sampler = sc.qmc.LatinHypercube(d=(2 * p_dim))
51      samples = sampler.random(n=N)  # no. samples
52      samples_p = np.zeros((N, 2 * p_dim))
53      for i in range(sampler.d):
54          if i < p_dim:
55              samples_p[:, i] = (p_dist[i].ppf(samples[:, i]))
56          else:
57              samples_p[:, i] = p_dist[i - p_dim].ppf(samples[:, i])
58
59      A = samples_p[:, :p_dim]
60      B = samples_p[:, p_dim:]
61
62      yA = np.zeros((t_dim - 1, N))
63      yB = np.zeros((t_dim - 1, N))
64      yC = np.zeros((t_dim - 1, N))
65
66      sis = np.zeros((p_dim, t_dim - 1))
67      stis = np.zeros((p_dim, t_dim - 1))
68
69      for i in range(N):
70          yA[:, i] = func(f, x0, tk, u0, A[i])
71          yB[:, i] = func(f, x0, tk, u0, B[i])
72          if (i % 1000) == 0:
73              print(f"yA,yB: iteration {i}/{N}")
74
75      for i in range(p_dim):
76          C = B.copy()
77          C[:, i] = A[:, i]
78          for j in range(N):
79              yC[:, j] = func(f, x0, tk, u0, C[j])
80              if (j % 1000) == 0:
81                  print(f"yC{i}: iteration {j}/{N}")
82          for j in range(1, t_dim - 1):
83              f0 = ((1 / N) * np.sum(yA[j, :])) ** 2
84              si = (((1 / N) * (yA[j, :] @ yC[j, :])) - f0) / (
85                      ((1 / N) * (yA[j, :] @ yA[j, :])) - f0)
86              sti = 1 - (((1 / N) * (yB[j, :] @ yC[j, :])) - f0) / (
87                      ((1 / N) * (yA[j, :] @ yA[j, :])) - f0)
88              sis[i, j] = si
89              stis[i, j] = sti
90
91      return sis, stis
```

Listing 6.9: mpc_open_loop / sobol.py