

TKP4580 - Chemical Process Technology, Specialization project

A Comparison of Multiple Shooting and Collocation Approaches using Nonlinear Model Predictive Control

Agnes Camilla Tysland

Submission date: 18.12.2020

Supervisor: Johannes Jäschke, IKP

Co-supervisor: Caroline Satye Nakama, IKP



Department of Chemical Engineering
Norwegian University of Science and Technology

Summary

This specialization project focuses on two different simultaneous approaches used in nonlinear model predictive controlling (MPC), namely multiple shooting and orthogonal collocation. The two methods are widely used in industry, and both computational time and accuracy are important factors to consider when choosing specifications for an MPC. Therefore, the aim of this project is to compare the methods with focus on their performance. A simple continuous stirred tank reactor is considered, for comparing the methods when tracking a state setpoint trajectory. The discretized system has been simulated for a time period of 60 hours using `julia`, with one-, three- and five point collocation methods in addition to multiple shooting. The length of the finite elements has also been varied in order to investigate the impact on the optimization. Two different implementations have been compared, one where the model for each method is built and optimized at each finite element, and one where the models are built only prior and then updated before the optimization at every finite element.

For the system investigated, all of the methods performed well. The method with the lowest computational time for this case was one point collocation and the highest computational time was multiple shooting. Thus, for the system investigated, one point collocation is sufficient. For this system, both three- and five point collocation methods were as accurate at multiple shooting, but one point collocation proved to be slightly less accurate for a more coarse discretization.

The length of the finite elements affects both accuracy and computational time for the four methods investigated. The approximation of the true solution remarkably improves for increasing number of finite elements, however, so does the NLP size and therefore also the computational time.

The difference between updating an existing model and building a new model at every time step showed a small, but not very significant, effect on the computational time. However, for a larger, more nonlinear system, this difference may have a larger influence. For further work, it could therefore be interesting to introduce a more nonlinear model to the system, in order to investigate this difference in addition to the general progression of the methods when handling more difficult systems.

In conclusion, if accuracy is a key criterion, multiple shooting or a higher order collocation approach is recommended. Nonetheless, for simple systems, one point orthogonal collocation may be the fastest approach with decent accuracy.

Preface

Declaration of Compliance

I, Agnes Camilla Tysland, hereby declare that this is an independent work according to the exam regulations of the Norwegian University of Science and Technology (NTNU).

Signature:

A handwritten signature in black ink, reading "Agnes Camilla Tysland". The signature is written in a cursive style with a horizontal line above the last name.

Place and Date: Trondheim - Gløshaugen, December 2020

Table of Contents

| | |
|--|------------|
| Summary | i |
| Preface | ii |
| Table of Contents | iv |
| List of Figures | vi |
| Nomenclature | vii |
| 1 Introduction | 1 |
| 1.1 Motivation | 1 |
| 1.2 Specialization Project Goal | 2 |
| 1.3 Scope of Work | 2 |
| 2 Background and theory | 3 |
| 2.1 Model Predictive Control | 3 |
| 2.2 Multiple Shooting | 5 |
| 2.3 Orthogonal Collocation on Finite Elements | 7 |
| 3 Methodology | 11 |
| 3.1 Optimization of a biochemical reactor | 11 |
| 3.1.1 Dynamic model | 11 |
| 3.1.2 NMPC formulation | 12 |
| 3.2 Implementation | 14 |
| 4 Results & Discussion | 17 |
| 4.1 Comparing the Methods | 17 |
| 4.2 Changing the Length of the Finite Element | 21 |
| 4.3 Implementation - Making a new model vs. Updating the model | 27 |
| 5 Final Remarks | 29 |

| | |
|------------------------------|-----------|
| Bibliography | 31 |
| A Julia Codes | 33 |
| A.1 Main.jl | 33 |
| A.2 methods.jl | 37 |
| A.3 updatemodel.jl | 41 |
| A.4 Plant.jl | 42 |
| A.5 newMethod.jl | 42 |

List of Figures

| | |
|---|----|
| 2.1.1 MPC block diagram[1]. | 4 |
| 2.1.2 Illustration of MPC concept for single-input single-output (SISO) control, showing past and predicted future outputs, y and \hat{y} , previous and future control inputs, u , control horizon, M , prediction horizon, P , and sampling instants, k (Seborg et. al, 2017: 370). | 4 |
| 2.2.1 Simple block diagram for multiple shooting[2]. | 5 |
| 2.3.1 Simple block diagram for orthogonal collocation[2]. | 7 |
| 2.3.2 A function $z(t)$ parameterized using collocation approach. h_i shows the length of one finite element between t_{i-1} and t_i , and τ_j denotes the collocation points within the finite element(Biegler, 2010: 289). | 8 |
| 3.1.1 A simplified flowsheet of the biochemical reactor. | 11 |
| 3.1.2 Biomass concentration setpoint trajectory for the NMPC. | 13 |
| 4.1.1 Plot of biomass and substrate for all four methods. State x_1 represents the biomass concentration tracking the setpoint trajectory and state x_2 represents the substrate concentration. Number of finite elements is 60, and prediction and control horizon is set to five and three hours, respectively. | 18 |
| 4.1.2 CPU time used for all methods. Top plot and middle plot are the same, except that the initial value for the multiple shooting approach has been removed from the bottom plot, in order to better compare the values. The top and bottom plot is the same as well, except that the y-axis is set to 2 seconds in the bottom one. Number of finite elements is 60, and prediction and control horizon is set to five and three hours, respectively. | 19 |
| 4.2.1 Biomass state plotted against time for various values of N and using the different methods. Results from one-, three- and five point orthogonal collocation approach is shown in Plot (a), Plot (b) and Plot (c), respectively. Multiple shooting approach is shown in Plot (d). | 21 |

| | |
|--|----|
| 4.2.2 Biomass and substrate plotted against time for all four methods. State x_1 represents the biomass concentration tracking the setpoint trajectory and state x_2 represents the substrate concentration. Plot (a) shows all methods for $N = 30$ finite elements, Plot (b) for $N = 60$, Plot (c) for $N = 120$ and Plot (d) for $N = 180$ | 23 |
| 4.2.3 Computational time for all methods at different levels of discretization. The top plot in each subfigure is the computational time plotted against time. The middle plot in each subfigure is computational time plotted against time, but the initial value for the multiple shooting approach has been removed, in order to better investigate the plots. The bottom plot in each subfigure is the computational time plotted against time, and the y-axis is fixed to 2 seconds, hence a trend between the plots may be more easily observed. | 25 |
| 4.3.1 Plot of states for all methods with 60 finite elements and control and prediction horizon set to five and three hours respectively. | 27 |
| 4.3.2 CPU time for all methods with 60 finite elements and control and prediction horizon set to five and three hours respectively. | 27 |

Nomenclature

Acronyms

| | |
|------|------------------------------------|
| CPU | central processing unit |
| CV | controlled variable |
| CSTR | continuous stirred tank reactor |
| DAE | differential algebraic equations |
| DOF | degree of freedom |
| DV | disturbance variable |
| MPC | model predictive control |
| MV | manipulated variable |
| NLP | nonlinear programming |
| NMPC | nonlinear model predictive control |
| ODE | ordinary differential equation |
| PDE | partial differential equation |
| SISO | single-input single-output |

| <i>Symbol</i> | <i>Definition</i> | <i>Unit</i> |
|---------------|------------------------------|--------------------------------|
| D | Dilution rate | hr^{-1} |
| D_{min} | Minimum dilution rate | hr^{-1} |
| D_{max} | Maximum dilution rate | hr^{-1} |
| F | Volumetric flowrate | $\frac{m^3}{hr}$ |
| F_{in} | Inlet volumetric flowrate | $\frac{m^3}{hr}$ |
| F_{out} | Outlet volumetric flowrate | $\frac{m^3}{hr}$ |
| k_m | Monod saturation constant | $\frac{g}{L}$ |
| V | Volume of bioreactor | m^3 |
| x_i | System states concentration | $\frac{mass of cells}{volume}$ |
| x_1 | Biomass concentration | $\frac{mass of cells}{volume}$ |
| $x_{1,f}$ | Biomass feed concentration | $\frac{mass of cells}{volume}$ |
| x_2 | Substrate concentration | $\frac{mass of cells}{volume}$ |
| $x_{2,f}$ | Substrate feed concentration | $\frac{mass of cells}{volume}$ |
| Y | Process yield | - |
| μ | Specific growth rate | hr^{-1} |
| μ_{max} | Maximum specific growth rate | hr^{-1} |

Introduction

1.1 Motivation

Several industrial disciplines may, by obtaining and implementing the optimal solution to a control problem, achieve a more optimized control of the system overall. Optimal control may result in lowered total energy demand and reduced cost or increased production, and is therefore highly desired. A way of obtaining optimal control is by using Model Predictive Control (MPC). An MPC uses a dynamic model of the process to predict a systems behaviour and response, for example to a disturbance or setpoint change. An MPC has one or several decision variables to manipulate in order to obtain the wanted outcome. Its main goal is to find a vector of optimal control inputs or functions in order to maximize or minimize a certain objective, subject to constraints[4].

The optimal control problem may be transformed into a nonlinear programming problem (NLP) using a direct method[2]. The time period of interest is discretized into several smaller finite elements, and the state trajectory may then be obtained by parameterization. The dynamic model of the system, e.g. the ordinary differential equations (ODEs), may be given as equality constraints[5]. The resulting NLP may be solved using either a sequential or a simultaneous approach. When a simultaneous approach is used, the states and the chosen manipulated variables are considered to be optimization variables in the NLP. This gives the simultaneous approach the advantage that the jacobian matrix, i.e. the derivatives of the constraints with respect to the problem variables, becomes very sparse compared to those of sequential approaches.

Multiple shooting and orthogonal collocation are two different simultaneous approaches used for dynamic optimization problem solving. There is currently a controversy as to which method is the superior one. Some strongly believe that multiple shooting gives the most accurate and therefore best approximation while others are convinced that the solution simply is to add more collocation points. The two methods differ when it comes to the integration of the systems ODEs. Both methods discretize the continuous optimal control problem into finite elements, but where multiple shooting uses an embedded integrator, orthogonal collocation uses the optimizer to do the integration by evaluating the ODEs at

specific points in time.

1.2 Specialization Project Goal

Depending on the system at hand and the reason the MPC is being used, both accuracy and computational time may be critical. The specialization project goal is therefore to compare multiple shooting and orthogonal collocation, with focus on their accuracy when tracking a setpoint trajectory in addition to the processing time.

1.3 Scope of Work

A simple biochemical continuous stirred tank reactor (CSTR) is considered, and a given setpoint trajectory is used to compare the two methods. An MPC is built using `julia`, with functions for the two methods to be compared. The states, manipulated variables, deviations from setpoint trajectory and the computational time used is evaluated in order to compare the two methods. In addition, two different implementations will be considered and compared; one where the model for each method is made and optimized at every time step, and one where the models are made only in the beginning and then updated before the optimization at every time step. This is to investigate the difference in computational time between the two different ways of implementing.

The relevant theory regarding MPC and the methods is provided, followed by a description of the CSTR dynamic model. The results from the MPC application is presented and discussed. In the end, some final remarks are given. The `julia` codes are given in Appendix A.

Background and theory

2.1 Model Predictive Control

Model predictive control (MPC) is an important advanced control strategy for difficult control problems. The MPC's ability to handle multivariable dynamic models alongside the possibility of adding bounding constraints is what raises this controller above several alternatives. This method allows for imposing constraints both on the input and output variables, and has been used in industry from the late 1980's[6]. Qin and Bagwell (2003: 749) nicely formulated the objectives and features of an MPC as[7]

1. prevent violation of input and output constraints;
2. drive the control variables to their steady-state optimal values (dynamic output optimization);
3. drive the manipulated variables to their steady-state optimal values using remaining degrees of freedom (dynamic input optimization);
4. prevent excessive movement of manipulated variables;
5. when signals and actuators fail, control as much of the plant as possible.

A typical block diagram for MPC is shown in Figure 2.1.1. Given a process model, the current and future value of one or several output variables are predicted. A feedback containing the residuals is given to a prediction block at every time instant. The predicted outputs may then be used for both setpoint calculations and control action calculations for the next time instant.

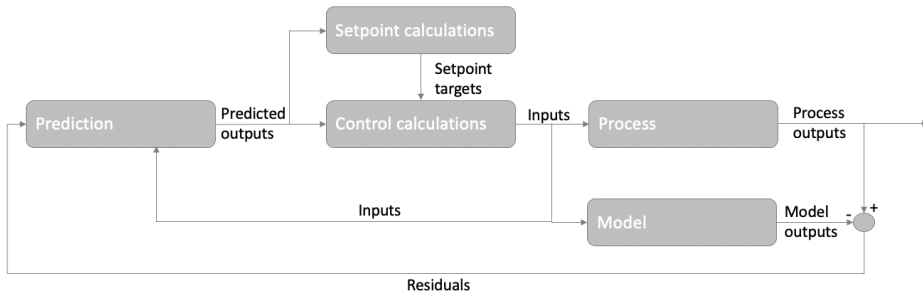


Figure 2.1.1: MPC block diagram[1].

Based on past control outputs, y , and inputs, u , the process is simulated over a certain time period, known as the prediction horizon or the output horizon, n_p . In the beginning of this horizon the manipulated variables (MVs) are allowed to vary. This period is called control horizon or input horizon, n_m . The control horizon cannot surpass the prediction horizon ($1 \leq n_m \leq n_p$) as shown in Figure 2.1.2, and during the output horizon the process must return to steady state[1].

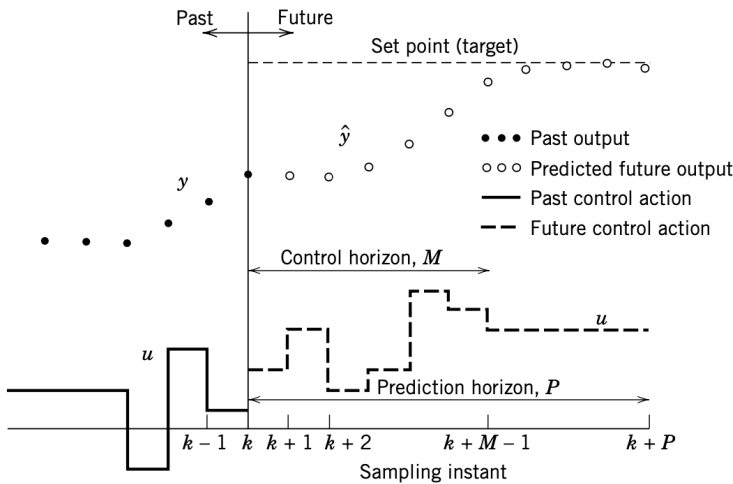


Figure 2.1.2: Illustration of MPC concept for single-input single-output (SISO) control, showing past and predicted future outputs, y and \hat{y} , previous and future control inputs, u , control horizon, M , prediction horizon, P , and sampling instants, k (Seborg et. al, 2017: 370).

This method relies on dynamic models of the process, either linear or nonlinear, which are obtained by system identification within separate time periods. A quadratic objective function is used to stabilize the profiles of the controlled variables (CVs) and to approximate them back to a setpoint. At each time step, a vector of optimal control input functions or values are found, using the control input and measured state from the previous step, and the first control input is given to the system. This way, the quadratic programming problem

is solved online[3].

Mathematically, an MPC problem can be formulated as follows:

$$\min_{u^k, y^k} \sum_{k=1}^{n_p} (\hat{y}^k - y^{sp})^T Q_y (\hat{y}^k - y^{sp}) + \sum_{k=1}^{n_m} (u^k - u^{k-1})^T Q_u (u^k - u^{k-1}) \quad (2.1.1)$$

$$s.t. \quad \hat{y}^{k+1} = F(y^k, u^k) \quad k = 1, \dots, n_p \quad (2.1.2)$$

$$u^k = u^{n_c} \quad k = n_c + 1, \dots, n_p \quad (2.1.3)$$

$$u^{min} \leq u^k \leq u^{max} \quad k = 1, \dots, n_p \quad (2.1.4)$$

$$-\Delta u^{max} \leq u^k - u^{k-1} \leq \Delta u^{max} \quad k = 1, \dots, n_p \quad (2.1.5)$$

Here, the predicted future outputs, \hat{y}^k , is dependent on the previous given inputs and measured outputs, u^k and y^k . Deviations from the setpoint, y^{sp} , and manipulated variable movement, $u^k - u^{k-1}$, is minimized using penalty matrices in the objective function, Equation 2.1.1. Q_y and Q_u are penalty matrices for the output variables and manipulated variables, respectively. Equation 2.1.3 is present to keep the manipulated variable constant after the control horizon. Δu^{max} is the maximum difference the manipulated variable may have from one time instant to the next. u^{min} and u^{max} are boundary values for the MV. The MPC problem solves the entire prediction horizon at each time step, and the optimal control input from the first time step is implemented in the process.

2.2 Multiple Shooting

Direct multiple shooting is a simultaneous approach used for dynamic optimization problem solving, as it simultaneously solves the optimization problem and the system ODEs. A simple block diagram is shown in Figure 2.2.1.

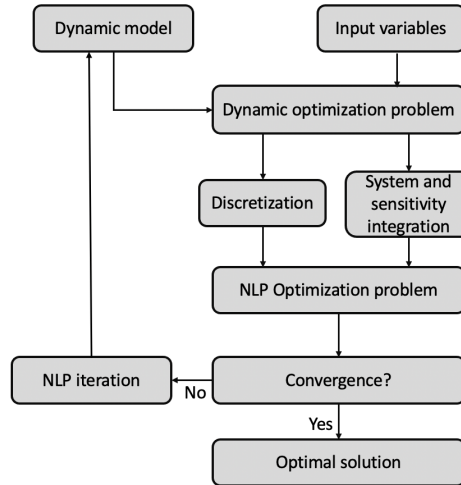


Figure 2.2.1: Simple block diagram for multiple shooting[2].

The numerical method solves boundary value problems by discretizing a finite interval into several smaller subintervals,

$$t_\alpha = t_0 < t_1 < \dots < t_{n-1} < t_n = t_\beta$$

where t_α and t_β are the times at the beginning and end of the finite interval respectively, and n is the number of subintervals. At each subinterval, the control function, $u(t)$, is parameterized

$$u(t) = \nu_k, t \in [t_k, t_{k+1}] \quad (2.2.1)$$

for $k = 1, \dots, n-1$, as well as the initial conditions of the state vectors,

$$y(t_k) = h_k \quad (2.2.2)$$

The state trajectories are evaluated in each subinterval,

$$\dot{y}_k(t) = f(y_k(t), \nu_k, t), t \in [t_k, t_{k+1}] \quad (2.2.3)$$

$$y_k(t_k) = h_k \quad (2.2.4)$$

To ensure continuity in the state trajectory, the continuity equation, also called matching conditions, needs to be satisfied. This controls the next shooting interval to start where the previous solution left off. Mathematically this may be described as

$$h_{k+1} - y_k(t_{k+1}; h_k, \nu_k) = 0 \quad (2.2.5)$$

This method integrates over a small shooting interval, starting at h_k . An advantage of the approach thus arises when, regardless of the integration strategy, the integration function $y(h_k, \nu_k)$ becomes asymptotically linear as the number of finite elements increases, in other words as $t_{k+1} - t_k$ goes to zero[8]. The integrator function is only moderately nonlinear and both the inputs, $u(t)$, and the states, $y(t)$, are used as decision variables. In multiple shooting the parameter vector for all subintervals then becomes

$$\mathbf{w} = [h_0, \nu_0, h_1, \nu_1, \dots, h_{n-1}, \nu_{n-1}, h_n] \quad (2.2.6)$$

Thus, the method considers a continuous optimal control problem

$$\min \quad \int_{t_0}^{t_f} \ell(\mathbf{y}(t), \mathbf{u}(t)) dt \quad (2.2.7)$$

$$\text{s.t.} \quad \dot{\mathbf{y}}(t) = \mathbf{f}(\mathbf{y}(t), \mathbf{u}(t)) \quad (2.2.8)$$

$$\mathbf{g}(\mathbf{y}(t), \mathbf{u}(t)) \leq 0 \quad (2.2.9)$$

$$\mathbf{y}(t_0) = \mathbf{y}_0 \quad (2.2.10)$$

$$(2.2.11)$$

and by discretization, the continuous problem is transformed into an NLP of the form

$$\min \quad \phi(\mathbf{w}) \quad (2.2.12)$$

$$\text{s.t.} \quad \mathbf{c}(\mathbf{w}) = 0 \quad (2.2.13)$$

$$\mathbf{g}(\mathbf{w}) \leq 0 \quad (2.2.14)$$

where \mathbf{c} and \mathbf{g} are the matrices

$$\mathbf{c}(\mathbf{w}) = \begin{bmatrix} y_0 - h_0 \\ y(h_0, \nu_0) - h_1 \\ y(h_1, \nu_1) - h_2 \\ \vdots \\ y(h_{n-1}, \nu_{n-1}) - h_n \end{bmatrix} = 0 \quad (2.2.15)$$

$$\mathbf{g}(\mathbf{w}) = \begin{bmatrix} g(h_0, \nu_0) \\ g(h_1, \nu_1) \\ \vdots \\ g(h_{n-1}, \nu_{n-1}) \\ g(h_n, \nu_n) \end{bmatrix} \leq 0 \quad (2.2.16)$$

A nonlinear optimizer can then be used to solve the resulting NLP problem. The integrator may be CVODES, using implicit Euler, Runge-Kutta or others.

2.3 Orthogonal Collocation on Finite Elements

Orthogonal collocation is a direct transcription method that, similarly to multiple shooting, allows for a simultaneous approach of an optimization problem. A simple block diagram for the method is shown in Figure 2.3.1. Unlike multiple shooting, orthogonal collocation considers a large nonlinear problem without an embedded differential algebraic equation (DAE) solver. In stead, the method uses the optimizer to do the integration, like shown in the block diagram below.

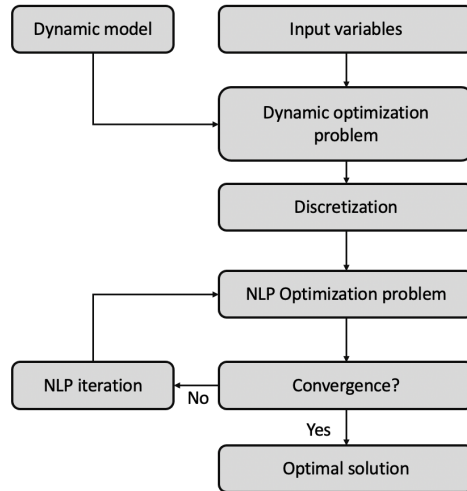


Figure 2.3.1: Simple block diagram for orthogonal collocation[2].

The time period of interest is discretized into several smaller finite elements and, within these elements, the system ODEs are evaluated at specific points in time, namely collocation points. This is illustrated in Figure 2.3.2, where τ_j denotes the collocation points within a finite element and h_i indicates the length of one finite element.

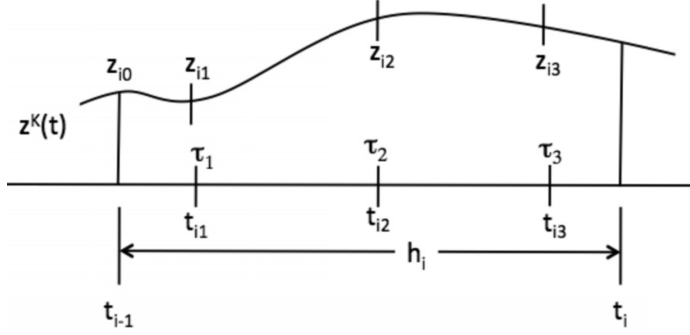


Figure 2.3.2: A function $z(t)$ parameterized using collocation approach. h_i shows the length of one finite element between t_{i-1} and t_i , and τ_j denotes the collocation points within the finite element (Biegler, 2010: 289).

The integrator equations are written out in order to be solved as constraints in the NLP. Therefore, the method only requires smooth profiles within the finite elements. The result should be a very large NLP with a sparse structure. This way, the nonlinearity of the problem is traded for the increasing problem size [3]. Different collocation approaches are distinguished by the number and positioning of the collocation points, and differ in approximation accuracy. Typical positions are Gauss-Lobatto, Gauss-Legendre and Gauss-Radau. The latter two are shown in Table 2.1. Derivation of these are shown in Biegler 2010 ([3], p. 288-295).

Table 2.1: Shifted Gauss–Legendre and Radau roots (Biegler, 2010, p. 292).

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

Consider the ODE

$$\frac{dz}{dt} = f(z(t), t), z(0) = z_0 \quad (2.3.1)$$

A polynomial approximation is considered for the state variable of the form

$$z^K(t) = \alpha_0 + \alpha_1 t + \alpha_2 t^2 + \dots + \alpha_K t^K \quad (2.3.2)$$

of order K over a single finite element. Using Lagrange interpolation polynomials and $K + 1$ interpolation points in each finite element i the state in this element may be given by

$$z^K(t) = \sum_{j=0}^K \ell_j(\tau) z_{ij} \quad (2.3.3)$$

where

$$\ell_j(t) = \prod_{k=0, k \neq j}^K \frac{\tau - \tau_K}{\tau_j - \tau_K} \quad (2.3.4)$$

for the time

$$t = t_{i-1} + h_i \tau, t \in [t_{i-1}, t_i] \quad (2.3.5)$$

As before, h_i is the length of the finite element and the interpolation points are

$$\tau \in [0, 1], \tau_0 = 0, \tau_j < \tau_{j+1} \quad (2.3.6)$$

for $j = 0, \dots, K - 1$.

The collocation equation for each finite element i may then be given as

$$\sum_{j=0}^K z_{ij} \frac{d\ell_j(\tau_k)}{d\tau} = h_i f(z_{ik}, t_{ik}), k = 1, \dots, K \quad (2.3.7)$$

which, for $K=3$ and using Radau roots can be written out as[3]

$$\begin{aligned}
 & z_{i,0}(-30\tau_k^2 + 36\tau_k - 9) + z_{i,1}(46.7423\tau_k^2 - 51.2592\tau_k + 10.0488) \\
 & + z_{i,2}(-26.7423\tau_k^2 + 20.5925\tau_k - 1.38214) + z_{i,3}(10\tau_k^2 - \frac{16}{3}\tau_k + \frac{1}{3}) \quad (2.3.8) \\
 & = h_i f(z_{i,k}, t_{i-1} + \tau_k)
 \end{aligned}$$

where the constants associated with z_0, \dots, z_K can be represented in a square matrix:

$$adot(k, k) = \begin{bmatrix} \dot{\ell}_0(\tau_0) & \dots & \dot{\ell}_0(\tau_k) \\ \vdots & \ddots & \vdots \\ \dot{\ell}_k(\tau_0) & \dots & \dot{\ell}_k(\tau_k) \end{bmatrix} \quad (2.3.9)$$

As the system has been discretized, continuity constraints must be enforced to ensure continuity. Depending on how the collocation approach is implemented, this may be enforced by using the collocation endpoint of one subinterval as the beginning of the next. This may be done by adding the equality constraint

$$z_{i+1,0} = z_{i,K} \quad (2.3.10)$$

constraining the next subinterval to start from the end of the previous one.

Methodology

3.1 Optimization of a biochemical reactor

3.1.1 Dynamic model

In this project a chemical bioreactor is considered. A simple flowsheet is presented in Figure 3.1.1. The reactor is modelled as a continuous stirred-tank reactor (CSTR), and it is assumed only two components are present; biomass, state x_1 , and substrate, state x_2 .

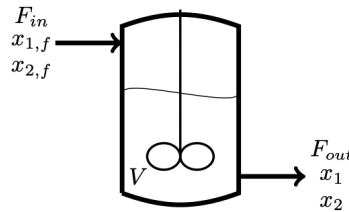


Figure 3.1.1: A simplified flowsheet of the biochemical reactor.

The biomass and substrate material balances are described as

$$\frac{dVx_1}{dt} = F_{in}x_{1,f} - F_{out}x_1 + Vr_1 \quad (3.1.1)$$

$$\frac{dVx_2}{dt} = F_{in}x_{2,f} - F_{out}x_2 - Vr_2 \quad (3.1.2)$$

where V is the reactor volume, x_i is the state concentration for $i = \{1,2\}$, t is time, F_{in} is the inlet volumetric flowrate, $x_{i,f}$ is the state concentration at the inlet, F_{out} is the outlet volumetric flowrate, r_1 is the reaction rate of biomass produced and r_2 is the reaction rate of substrate consumption.

The growth rate is further given by the equation

$$r_1 = \mu x_1 \quad (3.1.3)$$

Substrate inhibition is taken into account through μ by the substrate inhibition relationship

$$\mu = \frac{\mu_{max} x_2}{k_m + x_2 + k_1 x_2^2} \quad (3.1.4)$$

where μ_{max} is specific growth rate and k_m is the Monod saturation constant and k_1 is the inverse of the inhibition constant[9]. For the purpose of this project, approximated values are considered, given in Table 3.1.

The process yield is given by:

$$Y = \frac{\text{rate of biomass growth}}{\text{rate of substrate consumption}} = \frac{r_1}{r_2} \quad (3.1.5)$$

As the expression for r_1 is known, r_2 may be obtained by rearranging:

$$r_2 = \frac{r_1}{Y} = \frac{\mu x_1}{Y} \quad (3.1.6)$$

For this purpose, a process yield is assumed to be $Y = 40\% = 0.4$.

For the purpose of simplification, it is assumed that the volume is constant, hence feed stream in is equal to feed stream out, that there is no biomass in the feed stream and that yield is constant. In addition, the dilution rate is defined as:

$$D = \frac{F}{V} \quad (3.1.7)$$

The dynamic model for the process then becomes:

$$\begin{aligned} \frac{dx_1}{dt} &= -Dx_1 + \mu x_1 = x_1(\mu - D) \\ \frac{dx_2}{dt} &= Dx_{2,f} - Dx_2 - \frac{\mu x_1}{Y} = D(x_{2,f} - x_2) - \frac{\mu x_1}{Y} \end{aligned} \quad (3.1.8)$$

3.1.2 NMPC formulation

The goal is to design a nonlinear model predictive controller (NMPC) for the bioreactor, that tracks a reference trajectory of biomass concentration. The trajectory is shown in Figure 3.1.2.

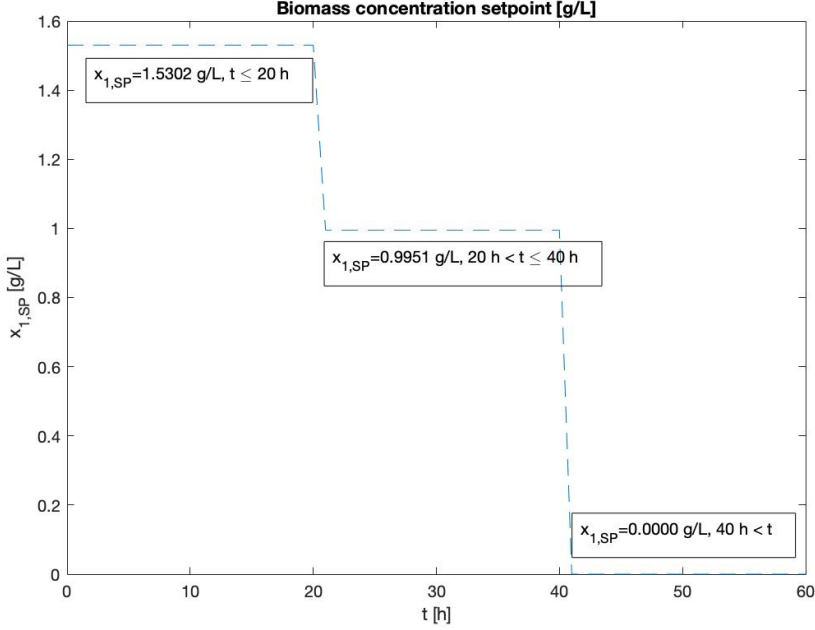


Figure 3.1.2: Biomass concentration setpoint trajectory for the NMPC.

Dilution rate is considered to be the manipulated variable (MV), u , and biomass concentration is the controlled variable (CV), x_1 . It is assumed that both states, x_1 and x_2 , are measured.

The discretized control problem is described as follows:

$$\min_{x,u} \quad \frac{1}{2} \left(\sum_{i=1}^{n_p} (x_{1,i} - x_{1,SP})^T Q (x_{1,i} - x_{1,SP}) + \sum_{i=1}^{n_m} \Delta u_i^T R \Delta u_i \right) \quad (3.1.9)$$

$$s.t. \quad x_0 = x(0) \quad (3.1.10)$$

$$x_{i+1} = F(x_i, u_i) \quad i = 0, \dots, n_p - 1 \quad (3.1.11)$$

$$u_{min} \leq u_i \leq u_{max} \quad i = 0, \dots, n_p \quad (3.1.12)$$

$$-\Delta u_{max} \leq \Delta u \leq \Delta u_{max} \quad i = 0, \dots, n_m \quad (3.1.13)$$

$$\Delta u_i = (u_i - u_{i-1}) = 0 \quad i = n_m + 1, \dots, n_p \quad (3.1.14)$$

$$x_1 \leq x_{1,max} \quad (3.1.15)$$

where $x_{1,i}$ is the predicted biomass concentration output at subinterval i , $x_{1,SP}$ is the biomass reference value and Q is a matrix that penalizes biomass deviations from setpoint. Similarly, R is a parameter that penalizes the magnitude of the manipulated variable step, Δu . $x_{i+1} = F(x_i, u_i)$ is the discretized system dynamics, here the differential equations given in Equation 3.1.8. n_p and n_m is the number of hours for the prediction horizon and the control horizon, respectively. $x_{1,max}$ is an upper bound for the biomass concentration.


The values of the control parameters are shown in Table 3.1.

Table 3.1: Control parameters and optimization bounds.

| Symbol | Value | Unit |
|------------------|--|------------|
| μ_{max} | 0.4 | |
| k_m | 0.12 | |
| k_1 | 0.4545 | |
| Y | 0.4 | |
| $x_{2,f}$ | 4.0 | |
| n_p | 5 | [-] |
| n_m | 3 | [-] |
| Q | $\begin{bmatrix} 1 & 0 \\ 0 & 0 \end{bmatrix}$ | [-] |
| R | 0.5 | [-] |
| u_{min} | 0 | $[h^{-1}]$ |
| u_{max} | 1 | $[h^{-1}]$ |
| Δu_{min} | 0.05 | $[h^{-1}]$ |

The system is simulated for 60 hours, with a control sampling time of 1 hour. The initial condition are $x_0 = [1, 1]^T$ [g/L] and $u_0 = 0.3$ [h⁻¹].

3.2 Implementation

The implementation of the MPC has been done in  using the packages JuMP, Ipopt, DifferentialEquations, Polynomials, PyPlot and LaTeXStrings. Four scripts were made, namely *Main.jl*, *Plant.jl*, *methods.jl* and *updatemodel.jl*, which are given in Appendix A.

In the script *methods.jl* found in A.2 there are three functions. The first function, *adot_matrix()*, is for making the collocation matrix shown in Equation 2.3.9. This function takes a number c as input, either 1, 3 or 5, uses collocation points to find Lagrange polynomials and its derivatives, and in the end returns the square collocation matrix needed for the collocation approaches.

The second function, *ColMod()*, takes as input the prediction and control horizon as an array, the length of the finite element and the number of collocation points, and uses this to generate the dynamic collocation model for optimization. All variables and constraints are here attached to the model.

Similarly, the third function, *MultMod()*, makes the multiple shooting model ready for optimization. It also takes horizons and finite element length as inputs, and returns the multiple shooting model containing all variables and constraints. This method, as mentioned in the previous chapter, uses an integrator to evaluate the ODEs at the end of each finite element. Therefore, this function includes another function that takes the state values in the beginning of the single finite element as input, and after evaluating the ODEs using the classic Runge-Kutta approach, returns the value of the states at the end of the single finite element.

The script *updatemodel.jl* in A.3 includes one function, *update_model()*. As input, this function takes the model to be updated, the previous states values, the previous input variable and the setpoint for state x_1 . The function then updates the values and optimizes the model using IPOPT. The function then returns the optimal input variable found and the Central Processing Unit (CPU) time used to find optimal solution.

The third script *Plant.jl* in A.4 is a script for simulating the system behaviour. The script contains two functions. The function *Plant()* takes as arguments the previous states, previous manipulated variable and finite element length, calls the integrator and returns the resulting states. The function *F()* is the integrator integrating the system ODEs and returning the integrated state values.

The last script *Main.jl* in A.1 is the script that simulates the entire optimization problem. It includes the three other scripts, makes the models and then, in a for-loop, updates the models, simulates the plant and in the end plots the results.

In addition, a script was made, which includes all of the characteristics of the above scripts in one, but differs as instead of updating the model, a new model is made at every time step. This script is named *newMethod.jl*.

Results & Discussion

4.1 Comparing the Methods

The biochemical reactor control problem have been solved using NMPC with one-, three- and five point collocation and multiple shooting for optimizing the control input. A time frame of 60 hours was considered, using $N = 60$ finite elements. Prediction horizon was set to $n_p = 5$ hours and control horizon was set to $n_m = 3$ hours.

A comparison of all the methods have been made, and is shown in Figure 4.1.1. As indicated by the legend, one point orthogonal collocation (OC1) is shown in blue, three point (OC3) in red, five point (OC5) in orange and multiple shooting (MS) in bright blue. This color scheme will be consistent through all the following plots in this section. Similarly, where the states are represented by a line, biomass state x_1 will be represented by the dashed line and substrate state x_2 by the solid line. Each method has, as indicated by the legend, markers shaped as triangles facing different directions in all of the state plots. The setpoint trajectory is plotted as a purple, dotted line.

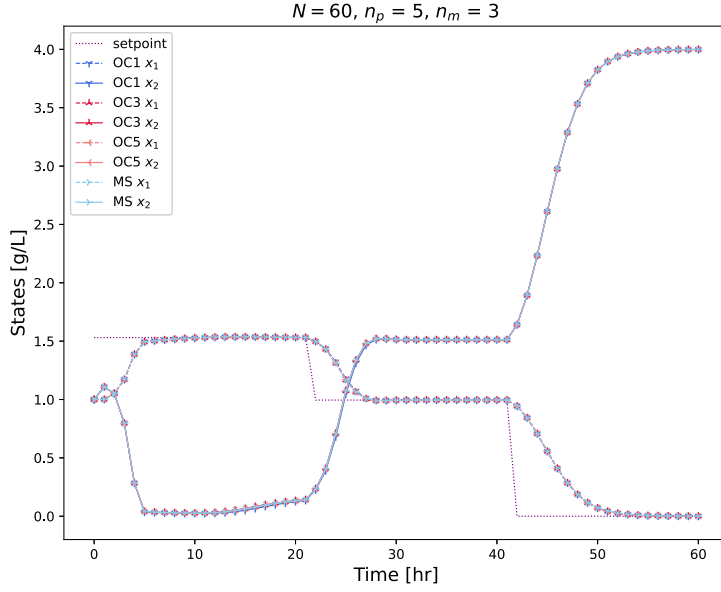


Figure 4.1.1: Plot of biomass and substrate for all four methods. State x_1 represents the biomass concentration tracking the setpoint trajectory and state x_2 represents the substrate concentration. Number of finite elements is 60, and prediction and control horizon is set to five and three hours, respectively.

Figure 4.1.1 shows biomass, state x_1 , and substrate, state x_2 , plotted against time for each method. All methods perform equally well for this case, and track the biomass trajectory presented in Figure 3.1.2 well.

The CPU time for all methods is shown in Figure 4.1.2. The number of finite elements is 60, and prediction and control horizon is set to five and three hours, respectively. The three plots all show computational time against time for all of the four methods, but with minor adjustments for the two bottom ones. The top and middle plot are the same, except that the initial value for the multiple shooting approach has been removed from the middle plot, in order to better view the values. The bottom plot is the same as the top plot as well, except that the y-axis has been set to 2 seconds. This is in order for the plots to be easier to compare in the next section.

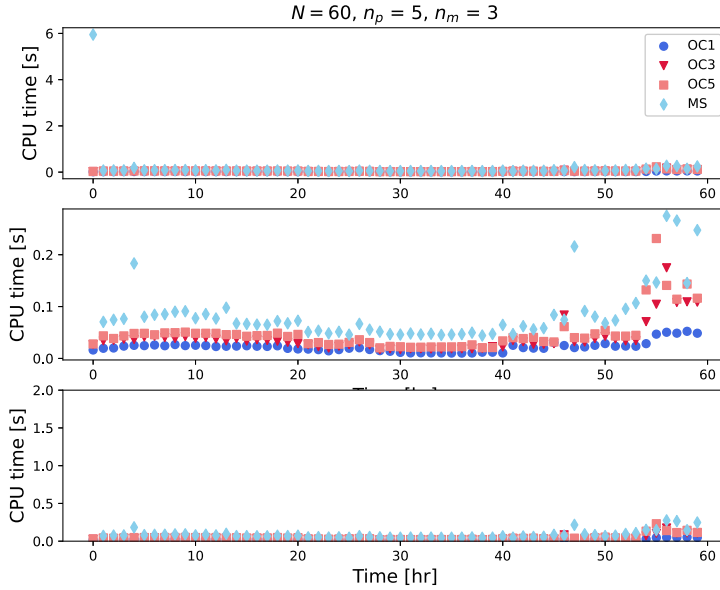


Figure 4.1.2: CPU time used for all methods. Top plot and middle plot are the same, except that the initial value for the multiple shooting approach has been removed from the bottom plot, in order to better compare the values. The top and bottom plot is the same as well, except that the y-axis is set to 2 seconds in the bottom one. Number of finite elements is 60, and prediction and control horizon is set to five and three hours, respectively.

From the middle plot in Figure 4.1.2 it is clear that the approach with the lowest computational time is one point collocation, followed by three point and then five. This is expected, as the number of collocation points indicates the order of the polynomial approximation in each finite element. The higher the order, the more variables are being used, thus increasing the size of the problem. A higher order is expected to give a more accurate solution, but also to use more computational time than a possibly more inaccurate, simpler approximation. The approach with the highest computational time is multiple shooting, which is also as expected as this method uses an embedded integrator.

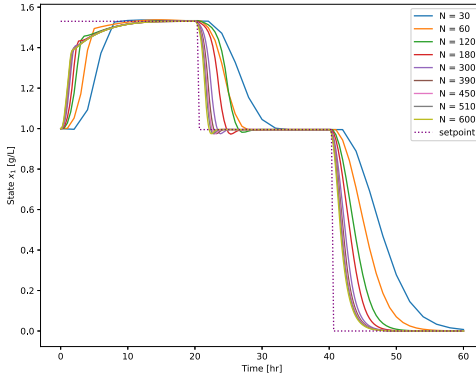
In the top plot of Figure 4.1.2 the CPU time for the initial multiple shooting optimiza-

tion clearly stands out. The CPU time for this point is close to 6 seconds while the majority of the other subintervals are optimized at under 0.2 seconds. This is likely due to the time it takes to allocate the memory when solving the ODEs in the multiple shooting approach. This approach uses an integrator from the DifferentialEquations package, which the first time it is called needs to allocate the memory needed for the integration, while for the rest of the subintervals this memory space may be overwritten as the values do not need to be saved.

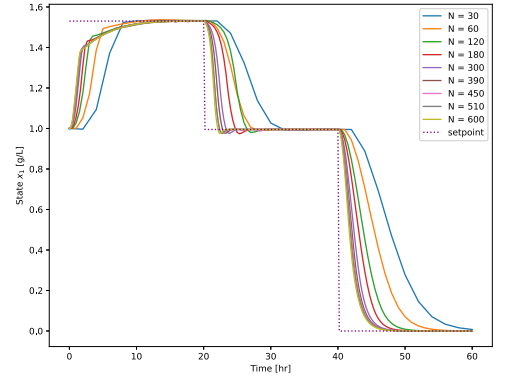
The computational time for the last five hours also stands out for all methods when considering the bottom plot in Figure 4.1.2. Looking closer at the plotted biomass states in Figure 4.1.1 this corresponds to the time when biomass concentration reaches its setpoint of zero. This is also a boundary constraint for the system, and the optimizer needs more iterations. Hence, the computational time is increased.

4.2 Changing the Length of the Finite Element

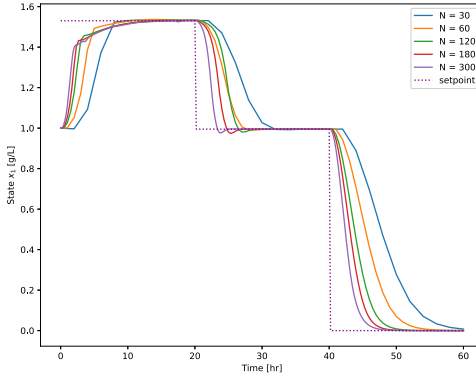
To examine the impact of the length of the finite elements, the biochemical reactor control problem was also solved using various number of finite elements. The prediction and control horizons have been changed accordingly, in order to keep the prediction time window approximately constant. The cases where $N = 30$, $N = 120$ and $N = 180$ finite elements will be closer investigated later in this section, but first each method is presented in its own plot, for different values of N . This is presented in Figure 4.2.1, with the purpose of looking solely at the difference in performance.



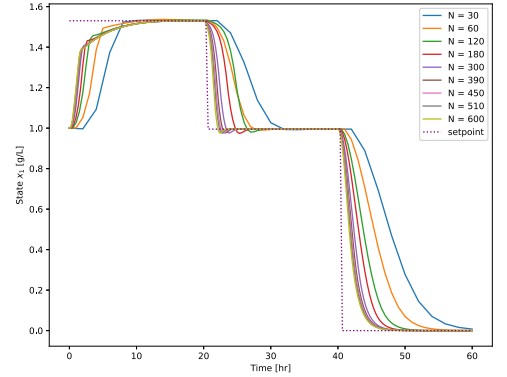
(a) One point orthogonal collocation



(b) Three point orthogonal collocation



(c) Five point orthogonal collocation



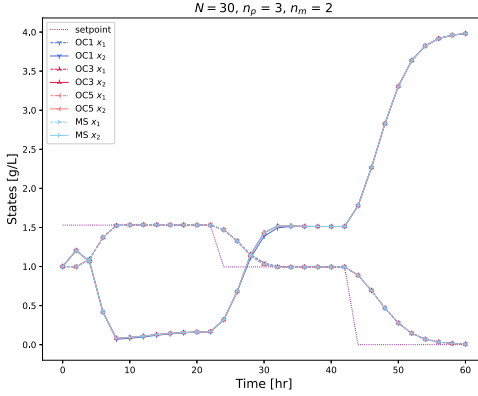
(d) Multiple shooting

Figure 4.2.1: Biomass state plotted against time for various values of N and using the different methods. Results from one-, three- and five point orthogonal collocation approach is shown in Plot (a), Plot (b) and Plot (c), respectively. Multiple shooting approach is shown in Plot (d).

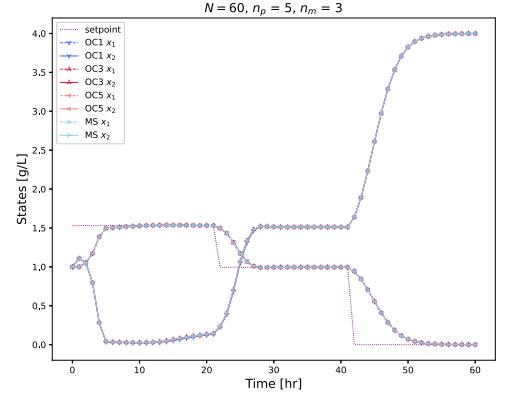
Figure 4.2.1 shows the biomass state plotted against time for all the methods. Here, it can be seen that the amount of finite elements clearly affect time it takes the biomass state to reach the new setpoint after a change. It is also a pattern for all the methods that the difference in solution for e.g. $N = 30$ to $N = 60$ is greater than the difference between $N = 450$ and $N = 510$. This is because as the length of the finite element decreases, the solution is approximating the true solution. Therefore, when the difference in solution becomes insignificant when increasing the number of finite elements, this may be considered as the "true solution".

Five point collocation for larger values of N , resulted in too many variables. As a consequence the model got too big for the RAM memory available when running. Thus, only values up to $N = 300$ is shown for this method.

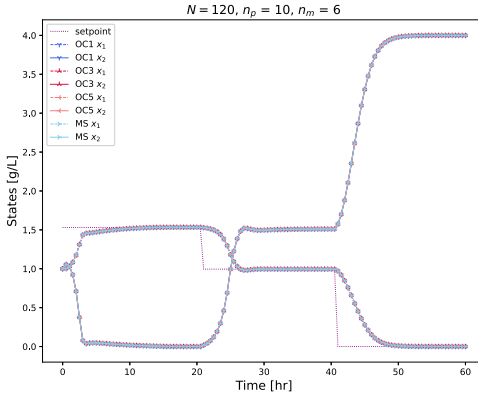
Figure 4.2.2 contains four plots, each representing the solution for all four methods for a specific finite element size.



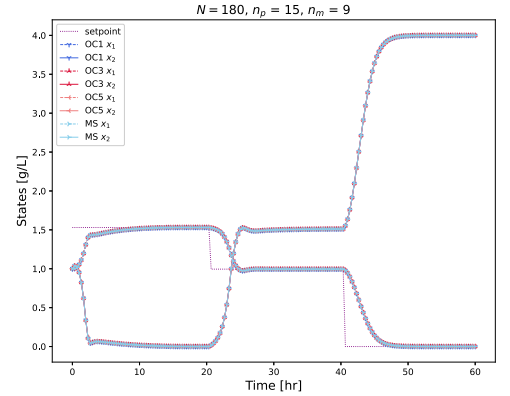
(a) Number of finite elements is 30, and prediction and control horizon is set to three and two hours, respectively.



(b) Number of finite elements is 60, and prediction and control horizon is set to five and three hours, respectively.



(c) Number of finite elements is 120, and prediction and control horizon is set to ten and six hours, respectively.



(d) Number of finite elements is 180, and prediction and control horizon is set to fifteen and nine hours, respectively.

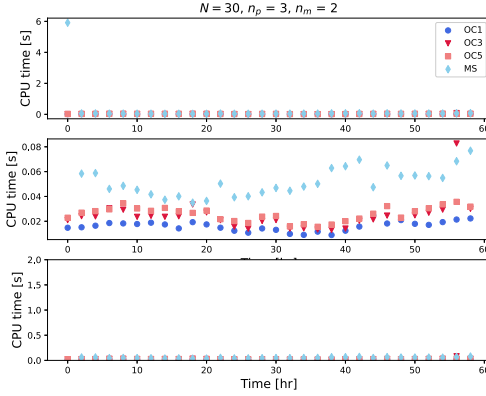
Figure 4.2.2: Biomass and substrate plotted against time for all four methods. State x_1 represents the biomass concentration tracking the setpoint trajectory and state x_2 represents the substrate concentration. Plot (a) shows all methods for $N = 30$ finite elements, Plot (b) for $N = 60$, Plot (c) for $N = 120$ and Plot (d) for $N = 180$.

Comparing Subfigure 4.2.2a and Subfigure 4.2.2b it shows that for both values of N all of the methods perform quite well. When investigating 4.2.2a, it is revealed that one point orthogonal collocation is slightly slower reaching the setpoint after the change at 20 hours, compared to the other three methods. This is expected as one point collocation is

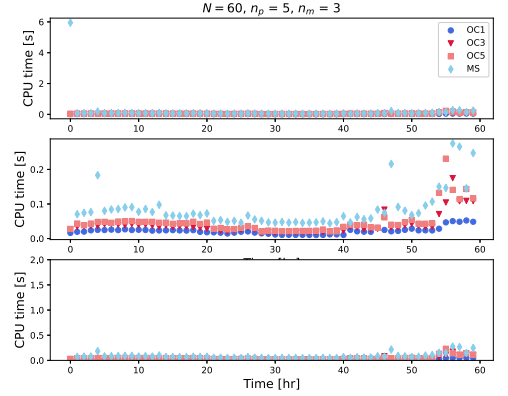
less accurate and coarser than the other methods. Judging by the same plot, three and five point collocation approaches overlap completely with multiple shooting.

For the remaining plots the methods overlap as well, and, as previously discussed, there is a trend towards the true solution as N is increased, which is visible when comparing all the plots in Figure 4.2.2.

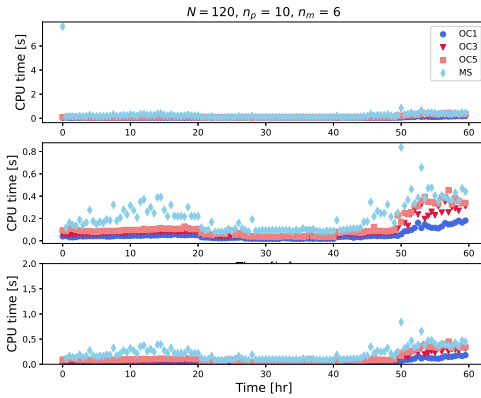
Figure 4.2.3 shows all four methods computational time, divided into four subfigures depending on the value of N . The subfigures have the same base as Figure 4.1.2, where all three plots are computational time against time, middle plot has discarded the first computational time value for multiple shooting, and the bottom plot has a fixed y-axis of 2 seconds.



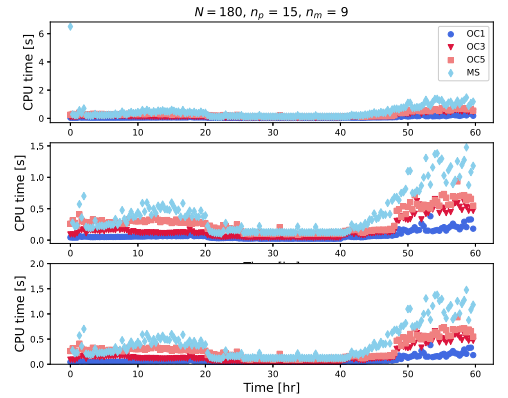
(a) Number of finite elements is 30, and prediction and control horizon is set to three and two hours, respectively.



(b) Number of finite elements is 60, and prediction and control horizon is set to five and three hours, respectively.



(c) Number of finite elements is 120, and prediction and control horizon is set to ten and six hours, respectively.



(d) Number of finite elements is 180, and prediction and control horizon is set to fifteen and nine hours, respectively.

Figure 4.2.3: Computational time for all methods at different levels of discretization. The top plot in each subfigure is the computational time plotted against time. The middle plot in each subfigure is computational time plotted against time, but the initial value for the multiple shooting approach has been removed, in order to better investigate the plots. The bottom plot in each subfigure is the computational time plotted against time, and the y-axis is fixed to 2 seconds, hence a trend between the plots may be more easily observed.

Considering the computational time for 30 finite elements shown in Figure 4.2.3a, the trend is more or less the same as the computational time for 60 finite elements. The plot shows multiple shooting being the slowest, and one point collocation being the fastest. Naturally, three point collocation comes second with regards to computational time, and five point collocation third.

Examining the middle and bottom plot of Subfigure 4.2.3c and Subfigure 4.2.3d, the previously discussed "jump" in computational time as time approaches 60 hours, is present earlier for these values with more finite elements. This is reasonable as the setpoint value of zero is reached earlier for larger values of N , as seen in Figure 4.2.1 and Figure 4.2.2. This is not seen in Subfigure 4.2.3a, indicating that the setpoint value is not reached before the last finite element, if at all. This corresponds with Figure 4.2.2a.

Comparing all of the bottoms plots in each Subfigure in Figure 4.2.3, the trend is quite clear. Computational time is increasing for increasing values of N . This is expected as the more finite elements the system is divided into, the more variables are available for the optimizer, i.e. the size of the NLP increase. A larger problem will take more time to solve.

4.3 Implementation - Making a new model vs. Updating the model

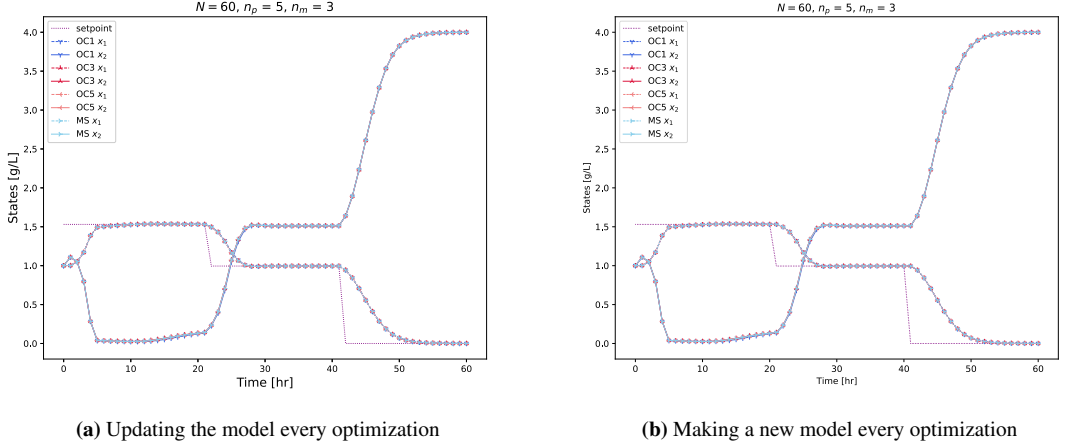


Figure 4.3.1: Plot of states for all methods with 60 finite elements and control and prediction horizon set to five and three hours respectively.

Regardless of whether the model is made at every finite element or updated at every finite element, the results with respect to the states are the same, as shown in Figure 4.3.1.

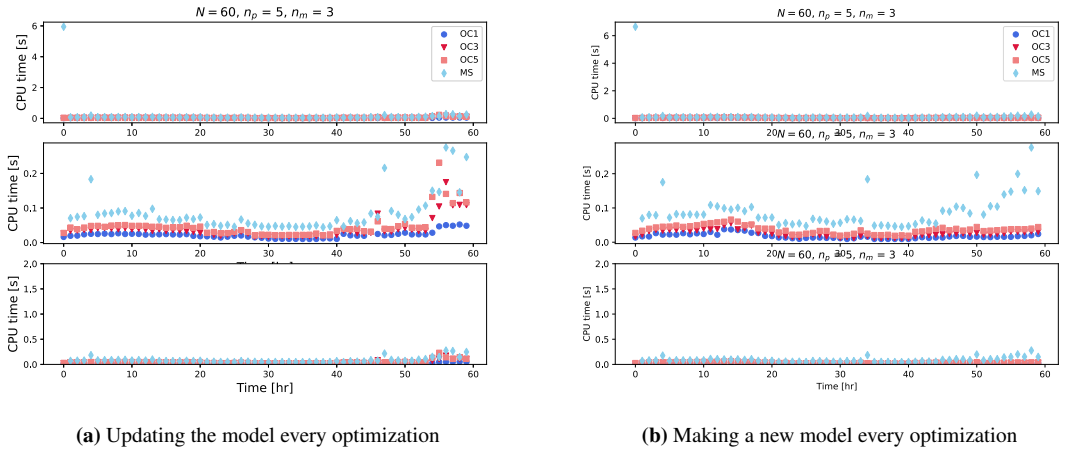


Figure 4.3.2: CPU time for all methods with 60 finite elements and control and prediction horizon set to five and three hours respectively.

Comparing the computational time of updating the same model versus making a new one at each finite element, it shows in Figure 4.3.2 that the computational time for multiple shooting is slightly lower when updating the model. Regarding the collocation approaches, they seem to not be affected significantly by this difference, however, an increase in computational time is noticeable from approximately 10-20 hours. Hence, the computational time is slightly affected by whether or not the model is made prior or at every finite element, nonetheless this difference in computational time is quite insignificant in this case. It might be reasonable to expect this difference to weigh heavier when dealing with a larger, more nonlinear system.

Final Remarks

For the scenario investigated in this specialization project, both multiple shooting and orthogonal collocation approach gives an acceptable result. Multiple shooting method is in most cases investigated the slower approach, nonetheless more accurate than one point orthogonal collocation, and as accurate as three and five point collocation. The fastest approach is one point orthogonal collocation, followed by three- and five point orthogonal collocation. This is expected, as a higher number of collocation point leads to more variables in the NLP, thus increasing the computational time.

The length of the finite elements was shown to affect both the accuracy of all the methods, in addition to the computational time, as expected. More finite elements leads to a better approximation, but the NLP size increases and as follows, also the CPU time. There is a trade off between computational time and accuracy, and depending on the system at hand, either the more accurate or the fastest approach can be chosen.

The difference between making a new model at every time step versus updating an existing, previously built model, did not influence the computational time significantly for the system investigated. However, for a larger, more nonlinear system, it is not unreasonable to expect this difference to have a larger influence on the computational time.

For further work, introducing a more nonlinear model to the system could be interesting. This way, it will be possible to examine the progression of each method, to investigate how they respond to more difficult problems.

In conclusion, for the system considered in this project, one point orthogonal collocation method present good accuracy and is the fastest approach. If accuracy is the most important criterion multiple shooting or a higher order collocation method is preferred, but if computational time is a critical measure, orthogonal collocation may be advantageous.

Bibliography

- [1] Seborg et. al. *Process Dynamics and Control*. Wiley, fourth ed. edition, 2017.
- [2] F. S. Rohman and N. Aziz. Comparison of orthogonal collocation, control vector parameterization and multiple shooting for optimization of acid recovery in batch electrodialysis. *AIP Conference Proceedings*, 2124(1):020013, 2019. doi: 10.1063/1.5117073. URL <https://aip.scitation.org/doi/abs/10.1063/1.5117073>.
- [3] Lorenz T. Biegler. *Nonlinear Programming: Concepts, Algorithms, and Applications to Chemical Processes*. SIAM, 2010.
- [4] Jasem Tamimi and Pu Li. Nonlinear model predictive control using multiple shooting combined with collocation on finite elements. *IFAC Proceedings Volumes*, 42(11):703 – 708, 2009. ISSN 1474-6670. doi: <https://doi.org/10.3182/20090712-4-TR-2008.00114>. URL <http://www.sciencedirect.com/science/article/pii/S1474667015303578>. 7th IFAC Symposium on Advanced Control of Chemical Processes.
- [5] Tor A Johansen. *Introduction to nonlinear model predictive control and moving horizon estimation*, pages 187–239. Bratislava, 2011.
- [6] Mark L. Darby. Industrial mpc of continuous processes. In *Encyclopedia of Systems and Control*, pages 1–10. Springer London, 2013. doi: 10.1007/978-1-4471-5102-9{_}242-1.
- [7] S. Joe Qin and Thomas A. Badgwell. A survey of industrial model predictive control technology. *Control Engineering Practice*, 11:733–764, 2003. doi: [https://doi.org/10.1016/S0967-0661\(02\)00186-7](https://doi.org/10.1016/S0967-0661(02)00186-7).
- [8] Sasa V Raković and William S Levine. *Handbook of Model Predictive Control*. Control engineering. Springer Basel AG, Cham, 2018. ISBN 9783319774886.
- [9] Feng Chenl and Michael R Johns. Relationship between substrate inhibition and maintenance energy of *Chlamydomonas reinhardtii* in heterotrophic culture. Technical report, 1996.

Appendix A

Julia Codes

A.1 Main.jl

```
import DifferentialEquations
using JuMP, Ipopt, Polynomials
using PyPlot, LaTeXStrings

include("Plant.jl")
include("methods.jl")
include("updatemodel.jl")

default = Dict(
    :t0 => 0,
    :tend => 60,
    :x0 => [1,1],
    :u0 => 0.3,
    :N => 60,
    #:dt => tend/N,
    :np => 5,
    :nm => 3,
    #:hor => [np,nm],
    :k1 => 0.4545,
    :Dmin => 0,
    :s => 2, # x1, x2
    :Dmax => 1,
    :dumax => 0.05,
    :mumax => 0.4,
    :xlmax => 4.5,
    :km => 0.12,
    :k1 => 0.4545,
    :x2f => 4,
    :Y => 0.4,
    :R => 0.5,
    :Q => 1
)

default
s = default[:s] # no of states
t0 = default[:t0]
tend = default[:tend]
N = default[:N]
```

```
#dt = default[:dt]
dt = tend/N
np = default[:np]
nm = default[:nm]
#hor = default[:hor]
hor = [np, nm]
x0 = default[:x0]
u0 = default[:u0]
dumax = default[:dumax]

# Arrays for 1 pt collocation
uSim1 = Array{Float64,1}(undef, N+1)
xSim1 = Array{Float64,2}(undef, N+1, 2)
x1_dev1 = Array{Float64,1}(undef, N+1)
CPU1Sim = Array{Float64,1}(undef, N)
# Arrays for 3 pt collocation
uSim3 = Array{Float64,1}(undef, N+1)
xSim3 = Array{Float64,2}(undef, N+1, 2)
x1_dev3 = Array{Float64,1}(undef, N+1)
CPU3Sim = Array{Float64,1}(undef, N)
# Arrays for 5 pt collocation
uSim5 = Array{Float64,1}(undef, N+1)
xSim5 = Array{Float64,2}(undef, N+1, 2)
x1_dev5 = Array{Float64,1}(undef, N+1)
CPU5Sim = Array{Float64,1}(undef, N)
# Arrays for multiple shooting
uSimM = Array{Float64,1}(undef, N+1)
xSimM = Array{Float64,2}(undef, N+1, 2)
x1_devM = Array{Float64,1}(undef, N+1)
CPUMSim = Array{Float64,1}(undef, N)
# General arrays
timeSim = Array{Float64,1}(undef, N+1)
x_sp = [1.5302*ones(1,N÷3+1) 0.9951*ones(1,N÷3) 0.0*ones(1,N÷3)]
spSim = Array{Float64,1}(undef, N+1)

#ColMod(hor, dt, c)
colmod1 = ColMod(hor, dt, 1)
colmod3 = ColMod(hor, dt, 3)
colmod5 = ColMod(hor, dt, 5)
multmod = MultMod(hor, dt)

for i = 1:N
    x1_sp = x_sp[i]
    spSim[i+1] = x1_sp
    # Disturbance
    if i==1
        test = Plant(x0,dt,u0)
        timeSim[1] = t0
        spSim[1] = x_sp[1]
        xSim1[1,1:2] .= x0
        xSim3[1,1:2] .= x0
        xSim5[1,1:2] .= x0
        xSimM[1,1:2] .= x0
        x1_dev1[1] = xSim1[1,1] - x1_sp
        x1_dev3[1] = xSim3[1,1] - x1_sp
        x1_dev5[1] = xSim5[1,1] - x1_sp
        x1_devM[1] = xSimM[1,1] - x1_sp
        uSim1[1] = u0
        uSim3[1] = u0
        uSim5[1] = u0
        uSimM[1] = u0
    end

    # Optimal input using NMPC function
    #update_model(m,x_prev,u_prev,x_sp_new)
    uSim1[i+1], CPU1Sim[i] = update_model(colmod1, xSim1[i,1:s],
                                           uSim1[i], x1_sp)
    uSim3[i+1], CPU3Sim[i] = update_model(colmod3, xSim3[i,1:s],
```

```

                                uSim3[i], xl_sp)
uSim5[i+1], CPU5Sim[i] = update_model(colmod5, xSim5[i,1:s],
                                uSim5[i], xl_sp)
uSimM[i+1], CPUMSim[i] = update_model(multmod, xSimM[i,1:s],
                                uSimM[i], xl_sp)
#Simulating the plant behavior during dt
xSim1[i+1,:] = Plant(xSim1[i,:], dt, uSim1[i+1])
xSim3[i+1,:] = Plant(xSim3[i,:], dt, uSim3[i+1])
xSim5[i+1,:] = Plant(xSim5[i,:], dt, uSim5[i+1])
xSimM[i+1,:] = Plant(xSimM[i,:], dt, uSimM[i+1])
# for plotting
#timeSim[i] = (i-1)*dt
timeSim[i+1] = timeSim[i] + dt
xl_dev1[i+1] = xSim1[i+1,1] - x_sp[i]
xl_dev3[i+1] = xSim3[i+1,1] - x_sp[i]
xl_dev5[i+1] = xSim5[i+1,1] - x_sp[i]
xl_devM[i+1] = xSimM[i+1,1] - x_sp[i]
end
#timeSim[end] = tend # GJ RE DETTE BEDRE

u_grid_b1 = Array{Float64,1}(undef, N+1)
u_grid_t1 = Array{Float64,1}(undef, N+1)
u_grid_b3 = Array{Float64,1}(undef, N+1)
u_grid_t3 = Array{Float64,1}(undef, N+1)

u_grid_b1[1] = uSim1[1] - dumax
u_grid_t1[1] = uSim1[1] + dumax
u_grid_b3[1] = uSim3[1] - dumax
u_grid_t3[1] = uSim3[1] + dumax
for i = 1:N
    u_grid_b1[i+1] = uSim1[i] - dumax
    u_grid_t1[i+1] = uSim1[i] + dumax
    u_grid_b3[i+1] = uSim3[i] - dumax
    u_grid_t3[i+1] = uSim3[i] + dumax
end

#####
### PLOTTING
#####
yline(timeSim) = 0*timeSim
uPlot1 = [uSim1 u_grid_t1 u_grid_b1]
uPlot3 = [uSim3 u_grid_t3 u_grid_b3]
xlPlot = [xSim1[:,1], xSim3[:,1], xSimM[:,1], spSim]
xPlot = [xSim1, xSim3, xSimM, spSim]
devPlot = [xl_dev1,xl_dev3,xl_devM,yline]
#plot(timeSim, xPlot)

##### PYPLOT
compPlot = figure(figsize=(9,7))
PyPlot.plot(timeSim,spSim, color="purple",label="setpoint",linestyle=":",lw=0.9)
PyPlot.plot(timeSim,xSim1[:,1], color="royalblue",label=L"OC1 $x_1$",
            linestyle="--",marker="1",lw=0.9)
PyPlot.plot(timeSim,xSim1[:,2], color="royalblue",label=L"OC1 $x_2$",
            linestyle="--",marker="1",lw=0.9)
PyPlot.plot(timeSim,xSim3[:,1], color="crimson",label=L"OC3 $x_1$",
            linestyle="--",marker="2",lw=0.9)
PyPlot.plot(timeSim,xSim3[:,2], color="crimson",label=L"OC3 $x_2$",
            linestyle="--",marker="2",lw=0.9)
PyPlot.plot(timeSim,xSim5[:,1], color="lightcoral",label=L"OC5 $x_1$",
            linestyle="--",marker="3",lw=0.9)
PyPlot.plot(timeSim,xSim5[:,2], color="lightcoral",label=L"OC5 $x_2$",
            linestyle="--",marker="3",lw=0.9)
PyPlot.plot(timeSim,xSimM[:,1], color="skyblue",label=L"MS $x_1$",
            linestyle="--",marker="4",lw=0.9)
PyPlot.plot(timeSim,xSimM[:,2], color="skyblue",label=L"MS $x_2$",
            linestyle="--",marker="4",lw=0.9)
PyPlot.xlabel("Time [hr]",fontsize=14)
PyPlot.ylabel("States [g/L]",fontsize=14)

```

```
PyPlot.legend()
#PyPlot.title("NMPC bioreactor trajectory")
PyPlot.title(L"N = " * string(N) * "L", $n_p$ = " * string(np) * "L", $n_m$ = "
    * string(nm), fontsize=14)
PyPlot.savefig("/Users/agnescamilla/Documents/NTNU/PROSJEKT/Figurer/PlotsFINAL/
    comp_FE=" * string(dt) * "_N=" * string(N) * "_np="
    * string(np) * "_nm=" * string(nm) * ".eps")

MVfig = figure(figsize=(9,7))
PyPlot.step(timeSim,uSim1[:,1], color="royalblue",label=L"OC1",linestyle="--",
    lw=0.9)
PyPlot.step(timeSim,uSim3[:,1], color="crimson",label=L"OC3",linestyle="--",
    lw=0.9)
PyPlot.step(timeSim,uSim5[:,1], color="lightcoral",label=L"OC5",linestyle="--",
    lw=0.9)
PyPlot.step(timeSim,uSimM[:,1], color="skyblue",label=L"MS",linestyle="--",
    lw=0.9)
PyPlot.xlabel("Time [hr]",fontsize=14)
PyPlot.ylabel("Manipulated variable",fontsize=14)
PyPlot.legend()
PyPlot.title(L"N = " * string(N) * "L", $n_p$ = " * string(np) * "L", $n_m$ = "
    * string(nm), fontsize=14)
PyPlot.savefig("/Users/agnescamilla/Documents/NTNU/PROSJEKT/Figurer/PlotsFINAL/
    MVfig_FE=" * string(dt) * "_N=" * string(N)
    * "_np=" * string(np) * "_nm=" * string(nm) * ".eps")

devfig = figure(figsize=(9,7))
PyPlot.scatter(timeSim,xl_dev1,color="royalblue",label="OC1",marker="o",lw=0.7)
PyPlot.scatter(timeSim,xl_dev3,color="crimson",label="OC3",marker="v",lw=0.7)
PyPlot.scatter(timeSim,xl_dev5,color="lightcoral",label="OC5",marker="s",lw=0.7)
PyPlot.scatter(timeSim,xl_devM,color="skyblue",label="MS",marker="d",lw=0.7)
PyPlot.axhline(y=0,xmin=0,xmax=1,label="zero deviation",lw=0.7,color="purple")
PyPlot.xlabel("Time [hr]",fontsize=14)
PyPlot.ylabel(L"x_1 - x_{sp}" * " [g/L]", fontsize=14)
#PyPlot.title("Deviation from setpoint")
PyPlot.title(L"N = " * string(N) * "L", $n_p$ = " * string(np) * "L", $n_m$ = "
    * string(nm), fontsize=14)
PyPlot.legend()
PyPlot.savefig("/Users/agnescamilla/Documents/NTNU/PROSJEKT/Figurer/PlotsFINAL/
    dev_FE=" * string(dt) * "_N=" * string(N)
    * "_np=" * string(np) * "_nm=" * string(nm) * ".eps")

CPUfig = figure(figsize=(9,7))
PyPlot.subplot(311)
PyPlot.scatter(timeSim[1:end-1],CPU1Sim, color="royalblue",label="OC1",marker="o")
PyPlot.scatter(timeSim[1:end-1],CPU3Sim, color="crimson",label="OC3",marker="v")
PyPlot.scatter(timeSim[1:end-1],CPU5Sim, color="lightcoral",label="OC5",marker="s")
PyPlot.scatter(timeSim[1:end-1],CPUMSim, color="skyblue",label="MS",marker="d")
#PyPlot.xlabel("Time [hr]",fontsize=14)
PyPlot.ylabel("CPU time [s]",fontsize=14)
PyPlot.title(L"N = " * string(N) * "L", $n_p$ = " * string(np) * "L", $n_m$ = " * string(nm),
    fontsize=14)
PyPlot.legend()
PyPlot.subplot(312)
PyPlot.scatter(timeSim[1:end-1],CPU1Sim[1:end], color="royalblue",label="OC1",
    marker="o")
PyPlot.scatter(timeSim[1:end-1],CPU3Sim[1:end], color="crimson",label="OC3",
    marker="v")
PyPlot.scatter(timeSim[1:end-1],CPU5Sim[1:end], color="lightcoral",label="OC5",
    marker="s")
PyPlot.scatter(timeSim[2:end-1],CPUMSim[2:end], color="skyblue",label="MS",
    marker="d")
#PyPlot.ylim(0,30)
```

```

PyPlot.ylabel("CPU time [s]", fontsize=14)
PyPlot.xlabel("Time [hr]", fontsize=14)
PyPlot.subplot(313)
PyPlot.scatter(timeSim[1:end-1], CPU1Sim[1:end], color="royalblue", label="OC1",
               marker="o")
PyPlot.scatter(timeSim[1:end-1], CPU3Sim[1:end], color="crimson", label="OC3",
               marker="v")
PyPlot.scatter(timeSim[1:end-1], CPU5Sim[1:end], color="lightcoral", label="OC5",
               marker="s")
PyPlot.scatter(timeSim[2:end-1], CPUMSim[2:end], color="skyblue", label="MS",
               marker="d")

PyPlot.ylim(0,2)
PyPlot.xlabel("Time [hr]", fontsize=14)
PyPlot.ylabel("CPU time [s]", fontsize=14)
#PyPlot.legend()
PyPlot.savefig("/Users/agnescamilla/Documents/NTNU/PROSJEKT/Figurer/PlotsFINAL/
               CPU3_FE="*string(dt)*"_N="*string(N)*
               "_np="*string(np)*"_nm="*string(nm)*".eps")

CPUfig2 = figure(figsize=(9,7))
PyPlot.subplot(211)
PyPlot.scatter(timeSim[1:end-1], CPU1Sim, color="royalblue", label="OC1", marker="o")
PyPlot.scatter(timeSim[1:end-1], CPU3Sim, color="crimson", label="OC3", marker="v")
PyPlot.scatter(timeSim[1:end-1], CPU5Sim, color="lightcoral", label="OC5", marker="s")
PyPlot.scatter(timeSim[1:end-1], CPUMSim, color="skyblue", label="MS", marker="d")
PyPlot.xlabel("Time [hr]", fontsize=14)
PyPlot.ylabel("CPU time [s]", fontsize=14)
PyPlot.title(L"N = "*string(N)*L", $n_p$ = "*string(np)*L", $n_m$ = "
               *string(nm), fontsize=14)

PyPlot.legend()
PyPlot.subplot(212)
PyPlot.scatter(timeSim[1:end-1], CPU1Sim[1:end], color="royalblue", label="OC1",
               marker="o")
PyPlot.scatter(timeSim[1:end-1], CPU3Sim[1:end], color="crimson", label="OC3",
               marker="v")
PyPlot.scatter(timeSim[1:end-1], CPU5Sim[1:end], color="lightcoral", label="OC5",
               marker="s")
PyPlot.scatter(timeSim[2:end-1], CPUMSim[2:end], color="skyblue", label="MS",
               marker="d")

PyPlot.ylim(0,2)
PyPlot.xlabel("Time [hr]", fontsize=14)
PyPlot.ylabel("CPU time [s]", fontsize=14)
PyPlot.legend()
PyPlot.savefig("/Users/agnescamilla/Documents/NTNU/PROSJEKT/Figurer/PlotsFINAL/
               CPU2_FE="*string(dt)*"_N="*string(N)*
               "_np="*string(np)*"_nm="*string(nm)*".eps")

```

A.2 methods.jl

```

import DifferentialEquations
using JuMP, Ipopt, Polynomials

function adot_matrix(c)
    if c == 1
        # Collocation Points Using Radau Roots 3rd degree polynomial
        t0 = 0
        t1 = 1.0
        # Lagrange polynomials
        l0 = fromroots([t1]) / (t0 - t1)
        l1 = fromroots([t0]) / (t1 - t0)
        # 1st derivatives

```

```
d10 = derivative(l0)
d11 = derivative(l1)
# Collocation matrix: 1st derivatives evaluated at the collocation points
adot = zeros(2,2)
tau = [t0,t1]
for i = 1:c+1
    adot[1,i] = d10(tau[i])
    adot[2,i] = d11(tau[i])
end
return adot
end
if c == 3
    #Collocation Points Using Radau Roots 3rd degree polynomial
    t0 = 0
    t1 = 0.155051
    t2 = 0.644949
    t3 = 1.000000
    # Lagrange polynomials
    l0 = fromroots([t1,t2,t3])/((t0-t1)*(t0-t2)*(t0-t3))
    l1 = fromroots([t0,t2,t3])/((t1-t0)*(t1-t2)*(t1-t3))
    l2 = fromroots([t0,t1,t3])/((t2-t0)*(t2-t1)*(t2-t3))
    l3 = fromroots([t0,t1,t2])/((t3-t0)*(t3-t1)*(t3-t2))
    # 1st derivatives
    d10 = derivative(l0)
    d11 = derivative(l1)
    d12 = derivative(l2)
    d13 = derivative(l3)
    # Collocation matrix: 1st derivatives evaluated at the
    # collocation points
    adot = zeros(4,4)
    tau = [t0,t1,t2,t3]
    for i = 1:c+1
        adot[1,i] = d10(tau[i])
        adot[2,i] = d11(tau[i])
        adot[3,i] = d12(tau[i])
        adot[4,i] = d13(tau[i])
    end
    return adot
end
if c == 5
    #Collocation Points Using Radau Roots 3rd degree polynomial
    t0 = 0
    t1 = 0.057104
    t2 = 0.276843
    t3 = 0.583590
    t4 = 0.860240
    t5 = 1.000000
    # Lagrange polynomials
    l0 = fromroots([t1,t2,t3,t4,t5])/((t0-t1)*(t0-t2)*(t0-t3)*(t0-t4)*(t0-t5))
    l1 = fromroots([t0,t2,t3,t4,t5])/((t1-t0)*(t1-t2)*(t1-t3)*(t1-t4)*(t1-t5))
    l2 = fromroots([t0,t1,t3,t4,t5])/((t2-t0)*(t2-t1)*(t2-t3)*(t2-t4)*(t2-t5))
    l3 = fromroots([t0,t1,t2,t4,t5])/((t3-t0)*(t3-t1)*(t3-t2)*(t3-t4)*(t3-t5))
    l4 = fromroots([t0,t1,t2,t3,t5])/((t4-t0)*(t4-t1)*(t4-t2)*(t4-t3)*(t4-t5))
    l5 = fromroots([t0,t1,t2,t3,t4])/((t5-t0)*(t5-t1)*(t5-t2)*(t5-t3)*(t5-t4))
    # 1st derivatives
    d10 = derivative(l0)
    d11 = derivative(l1)
    d12 = derivative(l2)
    d13 = derivative(l3)
    d14 = derivative(l4)
    d15 = derivative(l5)
    # Collocation matrix: 1st derivatives evaluated at the
    # collocation points
    adot = zeros(c+1,c+1)
    tau = [t0,t1,t2,t3,t4,t5]
    for i = 1:c+1
        adot[1,i] = d10(tau[i])
        adot[2,i] = d11(tau[i])
```

```

        adot[3,i] = dl2(tau[i])
        adot[4,i] = dl3(tau[i])
        adot[5,i] = dl4(tau[i])
        adot[6,i] = dl5(tau[i])

    end
    return adot
end
error("Wrong number of collocation points")
end

function ColMod(hor, dt, c ;kwargs...)
    Dmin = get(kwargs, :Dmin, default[:Dmin])
    Dmax = get(kwargs, :Dmax, default[:Dmax])
    s = get(kwargs, :s, default[:s]) # x1, x2
    dumax = get(kwargs, :dumax, default[:dumax])
    mumax = get(kwargs, :mumax, default[:mumax])
    xlmax = get(kwargs, :xlmax, default[:xlmax])
    km = get(kwargs, :km, default[:km])
    k1 = get(kwargs, :k1, default[:k1])
    x2f = get(kwargs, :x2f, default[:x2f])
    Y = get(kwargs, :Y, default[:Y])
    #np = get(kwargs, :np, default[:np])
    #nm = get(kwargs, :nm, default[:nm])
    #hor = get(kwargs, :hor, default[:hor])
    R = get(kwargs, :R, default[:R])
    Q = get(kwargs, :Q, default[:Q])
    np, nm = hor

    m = Model(Ipopt.Optimizer)
    set_optimizer_attribute(m, "print_level", 4)
    @variable(m, Dmin <= u[1:np] <= Dmax)
    @variable(m, du[1:np], start = 0)
    @variable(m, 0 <= x[1:s,1:np,1:c+1], start = 1)
    @variable(m, 0 <= mu[1:np,2:c+1] <= mumax, start = 0.23 )
    @variable(m, rk[1:s, 1:np, 1:c+1])
    @variable(m, L[1:np, 1:c+1], start = 0)
    @variable(m, rkL[1:np, 1:c+1])
    @variable(m, uin)
    @variable(m, x_sp)
    adot = adot_matrix(c)

    #=
    @NLobjective(m, Min,
        0.5*(L[np,c+1]+sum(R*du[i]^2
            for i=1:np)))
    =#

    @NLobjective(m, Min, sum(dt*
        (
            0.5*(Q*(x[1,i,end]-x_sp)^2 + R*du[i]^2)
        )
        for i=1:np
    ))

    #####
    # CONSTRAINTS
    #####
    # Initial conds
    @constraint(m, init, L[1,1] == 0)
    @constraint(m, gapConstr0[i=1:s], x[i,1,1] == xk[i])
    @constraint(m, dudiff0, du[1]-(u[1]-uin) == 0)
    #@constraint(m, dudiff0, u[1]-uin == 0)

    @constraint(m, xlconstr[i=1:np,j=1:c+1], x[1,i,j] <= xlmax)
    @constraint(m, duconstr[i=nm+1:np], du[i] == 0)
    @constraint(m, duconstr[i=1:nm], -dumax <= du[i] <= dumax)
    @constraint(m, dudiff[i=2:np], du[i]-(u[i]-u[i-1]) == 0)

    @constraint(m, rkConstr[i=1:s,j=1:np,k=1:c+1], rk[i,j,k] == sum(x[i,j,l]

```



```

        *adot[l,k] for l = 1:c+1))
@NLconstraint(m, muConstr[i=1:np, j=2:c+1], mu[i,j] - mumax*x[2,i,j] /
        (km + x[2,i,j] + k1*x[2,i,j]^2) == 0)

@NLconstraint(m, xldot[i=1:np, j=2:c+1], rk[1,i,j] - dt*((mu[i,j]-u[i])
        *x[1,i,j]) == 0)
@NLconstraint(m, x2dot[i=1:np, j=2:c+1], rk[2,i,j] - dt*(u[i]*(x2f-x[2,i,j])
        -mu[i,j]*x[1,i,j]/Y) == 0)

# Closing shooting gap
@constraint(m, gapConstrs[i=1:s, j=1:np-1], x[i,j,end] - x[i,j+1,1] == 0)
# Objective constraints
@NLconstraint(m, rkLConstr[i=1:np, j=1:c+1], rkL[i,j]==sum(L[i,k]*adot[k,j]
        for k = 1:c+1))
@NLconstraint(m, objConstr[i=1:np, j=2:c+1], rkL[i,j] - dt*((x[1,i,j]-x_sp)^2
        *Q) == 0)
@NLconstraint(m, LgapConstr[i=1:np-1], L[i,end] - L[i+1,1] == 0)
return m
end

function MultMod(hor, dt ;kwargs...)
    Dmin = get(kwargs, :Dmin, default[:Dmin])
    Dmax = get(kwargs, :Dmax, default[:Dmax])
    s = get(kwargs, :s, default[:s]) # x1, x2
    dumax = get(kwargs, :dumax, default[:dumax])
    mumax = get(kwargs, :mumax, default[:mumax])
    x1max = get(kwargs, :x1max, default[:x1max])
    km = get(kwargs, :km, default[:km])
    k1 = get(kwargs, :k1, default[:k1])
    x2f = get(kwargs, :x2f, default[:x2f])
    Y = get(kwargs, :Y, default[:Y])
    #np = get(kwargs, :np, default[:np])
    #nm = get(kwargs, :nm, default[:nm])
    #hor = get(kwargs, :hor, default[:hor])
    R = get(kwargs, :R, default[:R])
    Q = get(kwargs, :Q, default[:Q])
    np, nm = hor

    function Ftrunc(x0_1,x0_2,u,j)
        x0 = [x0_1,x0_2]
        function f(xdot,x,pars,t)
            xdot[1] = ((mumax*x[2]/(km+x[2]+k1*x[2]^2)) - pars)*x[1]
            xdot[2] = pars*(x2f-x[2])-(mumax*x[2]/(km + x[2] + k1*x[2]^2))
                        *x[1]/Y
        end
        tspan = (0.,dt)
        #pars = [mu,u]
        prob = DifferentialEquations.ODEProblem(f,x0,tspan,u)
        sol = DifferentialEquations.solve(prob,DifferentialEquations.RK4(),
            abstol=1e-9, reltol=1e-6)

        xk = sol.u[end]
        return xk[trunc{Int,j}] # return xk[1] or xk[2]
    end

    #####
    # VARIABLES
    #####
    multmod = Model(Ipopt.Optimizer)
    set_optimizer_attribute(multmod,"print_level",0)
    JuMP.register(multmod, :Ftrunc, 4, Ftrunc, autodiff=true)
    @variable(multmod, Dmin <= u[1:np] <= Dmax)
    @variable(multmod, 0 <= x[1:s,1:np], start = 1)
    @variable(multmod, -dumax <= du[1:np] <= dumax)
    @variable(multmod, xk[1:s])
    @variable(multmod, x_sp)
    @variable(multmod, u_in)

    #####
    # OBJECTIVE
    #####

```

```

@NLobjective(multmod, Min, sum(dt*
    (
        0.5*(Q*(x[1,i]-x_sp)^2 + R*du[i]^2)
    )
    for i=1:np
))

#####
# CONSTRAINTS
#####
# Initial conds
@NLconstraint(multmod, eqInit[i=1:s], x[i,1]-Ftrunc(xk[1],xk[2],u[1],i) == 0)
@constraint(multmod, dudiff0, du[1]-(u[1]-uin) == 0)

#@constraint(multmod, init[i=1:s], x[i,1] == xk[i])
#@constraint(multmod, u[1] - uin == 0)

@NLconstraint(multmod, eq[i=1:s,j=2:np], x[i,j]-Ftrunc(x[1,j-1],x[2,j-1],
    u[j],i) == 0)

@constraint(multmod, xlconstr[i=1:np], x[1,i] <= xlmax)
@constraint(multmod, duconstr0[i=nm+1:np], du[i] == 0)
@constraint(multmod, dudiff[i=2:np], du[i]-(u[i]-u[i-1]) == 0)
return multmod
end

```

A.3 updatemodel.jl

```

import JuMP

function update_model(m,x_prev,u_prev,x_sp_new)
    x = m[:x]
    uin = m[:uin]
    u1 = m[:u][1]

    x_sp = m[:x_sp]
    if length(size(x)) == 3

        for i=1:2
            fix(x[i,1,1],x_prev[i], force=true) # only for collocation?
        end
    elseif length(size(x)) == 2
        xk = m[:xk]
        for i=1:2
            fix(xk[i],x_prev[i], force=true) # only for collocation?
        end
    end
    fix(uin,u_prev, force=true)
    fix(x_sp,x_sp_new, force=true)
    set_start_value.(x[1,:,:),x_sp_new)
    set_start_value.(x[2,:,:),1)
    #set_start_value.(rk[1,:,:),x_sp_new)
    sol = JuMP.optimize!(m)
    u_opt = JuMP.value.(u1)
    CPUtime = solve_time(m)
    return u_opt, CPUtime
end

```

A.4 Plant.jl

```
import DifferentialEquations
using JuMP, Ipopt

#####
# INTEGRATOR
#####
function F(x0_1,x0_2,u,dt ;kwargs... )
    x0 = [x0_1,x0_2]
    mumax = get(kwargs, :mumax, default[:mumax])
    km = get(kwargs, :km, default[:km])
    k1 = get(kwargs, :k1, default[:k1])
    x2f = get(kwargs, :x2f, default[:x2f])
    Y = get(kwargs, :Y, default[:Y])
    function f(xdot,x,pars,t)
        # Model equations
        xdot[1] = (mumax*x[2]/(km+x[2]+k1*x[2]^2)) - pars)*x[1]
        xdot[2] = pars*(x2f-x[2])-(mumax*x[2] / (km + x[2] + k1*x[2]^2))*x[1]/Y
    end
    tspan = (0.,dt)
    prob = DifferentialEquations.ODEProblem(f,x0,tspan,u)
    sol = DifferentialEquations.solve(prob,DifferentialEquations.RK4())
    xk = sol.u
    println("xk12 = ",xk)
    println("xk2 = ",xk[end])
    return xk[end] # xk needs to be both x1 and x2!!!
end

#####
# PLANT
#####
function Plant(x0,dt,uk)
    x_new = F(x0[1],x0[2],uk,dt)
    return x_new
end
```

A.5 newMethod.jl

```
import DifferentialEquations
using JuMP, Ipopt, Polynomials
#using Plots
using PyPlot, LaTeXStrings

# parameter values
Y = 0.4
x2f = 4
mumax = 0.4
k1 = 0.4545
km = 0.12
#####
# INTEGRATOR
#####
function F(x0_1,x0_2,u,dt)
    x0 = [x0_1,x0_2]
    function f(xdot,x,pars,t)
        # Model equations
        xdot[1] = (mumax*x[2]/(km+x[2]+k1*x[2]^2)) - pars)*x[1]
```

```

        xdot[2] = pars*(x2f-x[2])-(mumax*x[2] / (km + x[2] + k1*x[2]^2))*x[1]/Y
    end
    tspan = (0.,dt)
    prob = DifferentialEquations.ODEProblem(f,x0,tspan,u)
    sol = DifferentialEquations.solve(prob,DifferentialEquations.RK4())
    xk = sol.u
    #println("xk12 = ",xk)
    #println("xk2 = ",xk[end])
    return xk[end] # xk needs to be both x1 and x2!!!
end

#####
# PLANT
#####
function Plant(x0,dt,uk)
    x_new = F(x0[1],x0[2],uk,dt)
    return x_new
end

#####
# Collocation 1 points
#####
function NMPCColl(xk, uin, x_sp, hor, dt)
    c = 1 # No of col points
    Dmin = 0
    Dmax = 1
    mumax = 0.4
    xlmax = 4.5
    np = hor[1]
    nm = hor[2]
    R = 0.5
    Q = 1
    h = dt
    #Collocation Points Using Radau Roots 3rd degree polynomial
    t0 = 0
    t1 = 1.0
    # Lagrange polynomials
    l0 = fromroots([t1])/(t0-t1)
    l1 = fromroots([t0])/(t1-t0)
    # 1st derivatives
    dl0 = derivative(l0)
    dl1 = derivative(l1)
    # Collocation matrix: 1st derivatives evaluated at the collocation points
    adot = zeros(2,2)
    tau = [t0,t1]
    for i = 1:c+1
        adot[1,i] = dl0(tau[i])
        adot[2,i] = dl1(tau[i])
    end
    #####
    # VARIABLES
    #####
    NMPCColmod1 = Model(Ipopt.Optimizer)
    set_optimizer_attribute(NMPCColmod1,"print_level",0)
    @variable(NMPCColmod1, Dmin <= u[1:np] <= Dmax, start = uin)
    @variable(NMPCColmod1, du[1:np], start = 0)
    @variable(NMPCColmod1, 0 <= x[1:s,1:np,1:c+1], start = 1)
    @variable(NMPCColmod1, 0 <= mu[1:np,2:c+1] <= mumax, start = 0.23 )
    @variable(NMPCColmod1, rk[1:s, 1:np, 1:c+1])
    @variable(NMPCColmod1, L[1:np, 1:c+1], start = 0)
    @variable(NMPCColmod1, rkL[1:np, 1:c+1])

    set_start_value.(x[1,:,:),x_sp)
    set_start_value.(rk[1:s,:,:),x_sp)
    #####
    # OBJECTIVE
    #####
    #=

```

```

@NLOjective(NMPCColmod1, Min, sum(h*
    (
        0.5*(Q*(x[1,i,end]-x_sp)^2 + R*du[i]^2)
    )
    for i=1:np
))

=#
@NLOjective(NMPCColmod1, Min, sum(h*
    (sum(
        (0.5*(Q*(x[1,i,1]-x_sp)^2 + R*du[i]^2))
        *inv(adot[l,k])#^(-1)
        for k = 1:c+1
            for l = 2:c+1
        )
    )
    for i=1:np
))

@NLOjective(NMPCColmod1, Min,
    0.5*(L[np,c+1]+sum(R*du[i]^2
    for i=1:np)))

=#
@NLOjective(NMPCColmod1, Min, sum(dt*
    (
        0.5*(Q*(x[1,i,end]-x_sp)^2 + R*du[i]^2)
    )
    for i=1:np
))

#####
# CONSTRAINTS
#####
# Initial conds
@constraint(NMPCColmod1, init, L[1,1] == 0)
@constraint(NMPCColmod1, gapConstr0[i=1:s], x[i,1,1] == xk[i])
@constraint(NMPCColmod1, dudiff0, du[1]-(u[1]-uin) == 0)

@constraint(NMPCColmod1, xlconstr[i=1:np,j=1:c+1], x[1,i,j] <= xlmax)

@constraint(NMPCColmod1, duconstr0[i=nm+1:np], du[i] == 0)
@constraint(NMPCColmod1, duconstr[i=1:nm], -dumax <= du[i] <= dumax)
@constraint(NMPCColmod1, dudiff[i=2:np], du[i]-(u[i]-u[i-1]) == 0)

@constraint(NMPCColmod1, rkConstr[i=1:s,j=1:np,k=1:c+1], rk[i,j,k] ==
    sum(x[i,j,l]*adot[l,k] for l = 1:c+1))
@NLconstraint(NMPCColmod1, muConstr[i=1:np, j=2:c+1], mu[i,j] - mumax*x[2,i,j]
    / (km + x[2,i,j] + k1*x[2,i,j]^2) == 0)

@NLconstraint(NMPCColmod1, xldot[i=1:np, j=2:c+1], rk[1,i,j]
    - h*((mu[i,j]-u[i])*x[1,i,j]) == 0)
@NLconstraint(NMPCColmod1, x2dot[i=1:np, j=2:c+1], rk[2,i,j]
    - h*(u[i]*(x2f-x[2,i,j])-mu[i,j]
    *x[1,i,j]/Y) == 0)

# Closing shooting gap
@constraint(NMPCColmod1, gapConstrs[i=1:s, j=1:np-1], x[i,j,end]
    - x[i,j+1,1] == 0)

# Objective constraints
@NLconstraint(NMPCColmod1, rkLConstr[i=1:np,j=1:c+1], rkL[i,j]
    ==sum(L[i,k]*adot[k,j] for k = 1:c+1))
@NLconstraint(NMPCColmod1, objConstr[i=1:np,j=2:c+1], rkL[i,j]
    - h*((x[1,i,j]-x_sp)^2*Q) == 0)
@NLconstraint(NMPCColmod1, LgapConstr[i=1:np-1], L[i,end] - L[i+1,1] == 0)
#####
# SOLVE
#####
sol = JuMP.optimize!(NMPCColmod1)
u_opt = JuMP.value.(u[1])
CPUtime = solve_time(NMPCColmod1)

```

```

    return u_opt, CPUtime
end

#####
# Collocation 3 points
#####
function NMPCCol3(xk, uin, x_sp, hor, dt)
    c = 3                # No of col points
    Dmin = 0
    Dmax = 1
    mumax = 0.4
    xlmax = 4.5
    np = hor[1]
    nm = hor[2]
    R = 0.5
    Q = 1
    h = dt
    #Collocation Points Using Radau Roots 3rd degree polynomial
    t0 = 0
    t1 = 0.155051
    t2 = 0.644949
    t3 = 1.000000
    # Lagrange polynomials
    l0 = fromroots([t1,t2,t3])/((t0-t1)*(t0-t2)*(t0-t3))
    l1 = fromroots([t0,t2,t3])/((t1-t0)*(t1-t2)*(t1-t3))
    l2 = fromroots([t0,t1,t3])/((t2-t0)*(t2-t1)*(t2-t3))
    l3 = fromroots([t0,t1,t2])/((t3-t0)*(t3-t1)*(t3-t2))
    # 1st derivatives
    dl0 = derivative(l0)
    dl1 = derivative(l1)
    dl2 = derivative(l2)
    dl3 = derivative(l3)
    # Collocation matrix: 1st derivatives evaluated at the
    # collocation points
    adot = zeros(4,4)
    tau = [t0,t1,t2,t3]
    for i = 1:c+1
        adot[1,i] = dl0(tau[i])
        adot[2,i] = dl1(tau[i])
        adot[3,i] = dl2(tau[i])
        adot[4,i] = dl3(tau[i])
    end
    #####
    # MODEL
    #####
    NMPCColmod3 = Model(Ipopt.Optimizer)
    set_optimizer_attribute(NMPCColmod3,"print_level",0)
    #####
    # VARIABLES
    #####
    @variable(NMPCColmod3, Dmin <= u[1:np] <= Dmax, start = uin)
    @variable(NMPCColmod3, du[1:np], start = 0)
    @variable(NMPCColmod3, 0 <= x[1:s,1:np,1:c+1], start = 1)
    @variable(NMPCColmod3, 0 <= mu[1:np,2:c+1] <= mumax, start = 0.23 )
    @variable(NMPCColmod3, rk[1:s, 1:np, 1:c+1])
    @variable(NMPCColmod3, L[1:np, 1:c+1], start = 0)
    @variable(NMPCColmod3, rkL[1:np, 1:c+1])

    set_start_value.(x[1, :, :], x_sp)
    set_start_value.(rk[1:s, :, :], x_sp)
    #####
    # OBJECTIVE
    #####
    #=
    @NLobjective(NMPCColmod3, Min, sum(h*
        (
            0.5*(Q*(x[1,i,end]-x_sp)^2 + R*du[i]^2)
        )
    ))

```

```

                                for i=1:np
                                ))
=#
#=#
    @NLobjective(NMPCColmod3, Min, sum(h*
        (sum(
            (0.5*(Q*(x[1,i,1]-x_sp)^2 + R*du[i]^2))
            *inv(adot[1,k])#^(-1)
            for k = 1:c+1
                for l = 2:c+1
            )
        )
        for i=1:np
    ))

    @NLobjective(NMPCColmod3, Min,
        0.5*(L[np,c+1]+sum(R*du[i]^2
            for i=1:np)))
    #=#
    @NLobjective(NMPCColmod3, Min, sum(dt*
        (
            0.5*(Q*(x[1,i,end]-x_sp)^2 + R*du[i]^2)
        )
        for i=1:np
    ))

#####
# CONSTRAINTS
#####
# Initial conds
@constraint(NMPCColmod3, init, L[1,1] == 0)
@constraint(NMPCColmod3, gapConstr0[i=1:s], x[i,1,1] == xk[i])
@constraint(NMPCColmod3, dudiff0, du[1]-(u[1]-uin) == 0)

@constraint(NMPCColmod3, xlconstr[i=1:np,j=1:c+1], x[1,i,j] <= xlmax)

@constraint(NMPCColmod3, duconstr0[i=n+1:np], du[i] == 0)
@constraint(NMPCColmod3, duconstr[i=1:n], -dumax <= du[i] <= dumax)
@constraint(NMPCColmod3, dudiff[i=2:np], du[i]-(u[i]-u[i-1]) == 0)

@constraint(NMPCColmod3, rkConstr[i=1:s,j=1:np,k=1:c+1], rk[i,j,k]
    == sum(x[i,j,l]*adot[1,k] for l = 1:c+1))
@NLconstraint(NMPCColmod3, muConstr[i=1:np, j=2:c+1], mu[i,j]
    - mumax*x[2,i,j] / (km + x[2,i,j]
    + k1*x[2,i,j]^2) == 0)

@NLconstraint(NMPCColmod3, xldot[i=1:np, j=2:c+1], rk[1,i,j]
    - h*((mu[i,j]-u[i])*x[1,i,j]) == 0)
@NLconstraint(NMPCColmod3, x2dot[i=1:np, j=2:c+1], rk[2,i,j] - h
    *(u[i]*(x2f-x[2,i,j])-mu[i,j]*x[1,i,j]/Y)
    == 0)

# Closing shooting gap
@constraint(NMPCColmod3, gapConstrs[i=1:s, j=1:np-1], x[i,j,end]
    - x[i,j+1,1] == 0)

# Objective constraints
@NLconstraint(NMPCColmod3, rkLConstr[i=1:np,j=1:c+1], rkL[i,j]
    ==sum(L[i,k]*adot[k,j] for k = 1:c+1))
@NLconstraint(NMPCColmod3, objConstr[i=1:np,j=2:c+1], rkL[i,j]
    - h*((x[1,i,j]-x_sp)^2*Q) == 0)
@NLconstraint(NMPCColmod3, LgapConstr[i=1:np-1], L[i,end] - L[i+1,1] == 0)
#####
# SOLVE
#####
sol = JuMP.optimize!(NMPCColmod3)
u_opt = JuMP.value.(u[1])
CPUtime = solve_time(NMPCColmod3)
return u_opt, CPUtime
end

```

```
#####
# Collocation 5 points
#####
function NMPCCol5(xk, uin, x_sp, hor, dt, x_pre)
    c = 5                # No of col points
    Dmin = 0
    Dmax = 1
    mumax = 0.4
    xlmax = 4.5
    np = hor[1]
    nm = hor[2]
    R = 0.5
    Q = 1
    h = dt
    #Collocation Points Using Radau Roots 3rd degree polynomial
    t0 = 0
    t1 = 0.057104
    t2 = 0.276843
    t3 = 0.583590
    t4 = 0.860240
    t5 = 1.000000
    # Lagrange polynomials
    l0 = fromroots([t1,t2,t3,t4,t5])/((t0-t1)*(t0-t2)*(t0-t3)*(t0-t4)*(t0-t5))
    l1 = fromroots([t0,t2,t3,t4,t5])/((t1-t0)*(t1-t2)*(t1-t3)*(t1-t4)*(t1-t5))
    l2 = fromroots([t0,t1,t3,t4,t5])/((t2-t0)*(t2-t1)*(t2-t3)*(t2-t4)*(t2-t5))
    l3 = fromroots([t0,t1,t2,t4,t5])/((t3-t0)*(t3-t1)*(t3-t2)*(t3-t4)*(t3-t5))
    l4 = fromroots([t0,t1,t2,t3,t5])/((t4-t0)*(t4-t1)*(t4-t2)*(t4-t3)*(t4-t5))
    l5 = fromroots([t0,t1,t2,t3,t4])/((t5-t0)*(t5-t1)*(t5-t2)*(t5-t3)*(t5-t4))
    # 1st derivatives
    dl0 = derivative(l0)
    dl1 = derivative(l1)
    dl2 = derivative(l2)
    dl3 = derivative(l3)
    dl4 = derivative(l4)
    dl5 = derivative(l5)
    # Collocation matrix: 1st derivatives evaluated at the
    # collocation points
    adot = zeros(c+1,c+1)
    tau = [t0,t1,t2,t3,t4,t5]
    for i = 1:c+1
        adot[1,i] = dl0(tau[i])
        adot[2,i] = dl1(tau[i])
        adot[3,i] = dl2(tau[i])
        adot[4,i] = dl3(tau[i])
        adot[5,i] = dl4(tau[i])
        adot[6,i] = dl5(tau[i])
    end
    #####
    # MODEL
    #####
    NMPCColmod5 = Model(Ipopt.Optimizer)
    set_optimizer_attribute(NMPCColmod5,"print_level",4)
    #####
    # VARIABLES
    #####
    @variable(NMPCColmod5, Dmin <= u[1:np] <= Dmax, start = uin)
    @variable(NMPCColmod5, du[1:np], start = 0)
    @variable(NMPCColmod5, 0 <= x[1:s,1:np,1:c+1], start = 1)
    @variable(NMPCColmod5, 0 <= mu[1:np,2:c+1] <= mumax, start = 0.23)
    @variable(NMPCColmod5, rk[1:s, 1:np, 1:c+1])
    @variable(NMPCColmod5, L[1:np, 1:c+1], start = 0)
    @variable(NMPCColmod5, rkL[1:np, 1:c+1])

    #set_start_value.(x[1,:,:],x_sp)
    #set_start_value.(rk[1,:,:],x_sp)
    set_start_value.(x[i=1:s,:,:],x_pre[i])
    #set_start_value.(rk[i=1:s,:,:],x_pre[i])
    #####
end
```



```

# OBJECTIVE
#####
#=
@NLObjective (NMPCColmod5, Min, sum(h*
    (
        0.5*(Q*(x[l,i,end]-x_sp)^2 + R*du[i]^2)
    )
    for i=1:np
))

=#
#=
@NLObjective (NMPCColmod5, Min, sum(h*
    (sum(
        (0.5*(Q*(x[l,i,l]-x_sp)^2 + R*du[i]^2))
        *inv(adot[l,k])#^(-1)
        for k = 1:c+1
            for l = 2:c+1
        )
    )
    for i=1:np
))

@NLObjective (NMPCColmod5, Min,
    0.5*(L[np,c+1]+sum(R*du[i]^2
    for i=1:np)))

=#
@NLObjective (NMPCColmod5, Min, sum(dt*
    (
        0.5*(Q*(x[l,i,end]-x_sp)^2 + R*du[i]^2)
    )
    for i=1:np
))

#####
# CONSTRAINTS
#####
# Initial conds
@constraint (NMPCColmod5, init, L[1,1] == 0)
@constraint (NMPCColmod5, gapConstr0[i=1:s], x[i,1,1] == xk[i])
@constraint (NMPCColmod5, dudiff0, du[1]-(u[1]-uin) == 0)

@constraint (NMPCColmod5, xlconstr[i=1:np,j=1:c+1], x[l,i,j] <= xlmax)

@constraint (NMPCColmod5, duconstr0[i=nm+1:np], du[i] == 0)
@constraint (NMPCColmod5, duconstr[i=1:nm], -dumax <= du[i] <= dumax)
@constraint (NMPCColmod5, dudiff[i=2:np], du[i]-(u[i]-u[i-1]) == 0)

@constraint (NMPCColmod5, rkConstr[i=1:s,j=1:np,k=1:c+1], rk[i,j,k]
    == sum(x[i,j,l]*adot[l,k] for l = 1:c+1))
@NLconstraint (NMPCColmod5, muConstr[i=1:np, j=2:c+1], mu[i,j] - mumax*x[2,i,j]
    / (km + x[2,i,j] + k1*x[2,i,j]^2) == 0)

@NLconstraint (NMPCColmod5, xldot[i=1:np, j=2:c+1], rk[1,i,j]
    - h*((mu[i,j]-u[i])*x[1,i,j]) == 0)
@NLconstraint (NMPCColmod5, x2dot[i=1:np, j=2:c+1], rk[2,i,j]
    - h*(u[i]*(x2f-x[2,i,j])-mu[i,j]
    *x[1,i,j]/Y) == 0)

# Closing shooting gap
@constraint (NMPCColmod5, gapConstrs[i=1:s, j=1:np-1], x[i,j,end]
    - x[i,j+1,1] == 0)

# Objective constraints
@NLconstraint (NMPCColmod5, rkLConstr[i=1:np,j=1:c+1], rkL[i,j]
    ==sum(L[i,k]*adot[k,j] for k = 1:c+1))
@NLconstraint (NMPCColmod5, objConstr[i=1:np,j=2:c+1], rkL[i,j]
    - h*((x[1,i,j]-x_sp)^2*Q) == 0)
@NLconstraint (NMPCColmod5,LgapConstr[i=1:np-1], L[i,end] - L[i+1,1] == 0)
#####

```

```

# SOLVE
#####
sol = JuMP.optimize!(NMPCColmod5)
u_opt = JuMP.value.(u[1])
CPUtime = solve_time(NMPCColmod5)
return u_opt, CPUtime
end

#####
# Multiple shooting
#####
function NMPCMult(xk, uin, x_sp, hor, dt)
    s = 2          # x1, x2
    Dmin = 0
    Dmax = 1
    dumax = 0.05
    mumax = 0.4
    xlmax = 4.5
    km = 0.12
    k1 = 0.4545
    x2f = 4
    Y = 0.4
    np = hor[1]
    nm = hor[2]
    R = 0.5
    Q = 1
    h = dt

    function Ftrunc(x0_1,x0_2,u,j)
        x0 = [x0_1,x0_2]
        function f(xdot,x,pars,t)
            xdot[1] = ((mumax*x[2]/(km+x[2]+k1*x[2]^2)) - pars)*x[1]
            xdot[2] = pars*(x2f-x[2])-(mumax*x[2]/(km + x[2]
                                + k1*x[2]^2))*x[1]/Y
        end
        tspan = (0.,h)
        #pars = [mu,u]
        prob = DifferentialEquations.ODEProblem(f,x0,tspan,u)
        sol = DifferentialEquations.solve(prob,DifferentialEquations.RK4())
        xk = sol.u[end]
        return xk[Ftrunc(Int,j)] # return xk[1] or xk[2]
    end

    #####
    # VARIABLES
    #####
    NMPCMultmod = Model(Ipopt.Optimizer)
    set_optimizer_attribute(NMPCMultmod,"print_level",0)
    JuMP.register(NMPCMultmod, :Ftrunc, 4, Ftrunc, autodiff=true)
    @variable(NMPCMultmod, Dmin <= u[1:np] <= Dmax)
    @variable(NMPCMultmod, 0 <= x[1:s,1:np], start = 1)
    @variable(NMPCMultmod, -dumax <= du[1:np] <= dumax)
    set_start_value.(x[1, :, :], x_sp)
    #####
    # OBJECTIVE
    #####
    @NLobjective(NMPCMultmod, Min, sum(h*
        (
            0.5*(Q*(x[1,i]-x_sp)^2 + R*du[i]^2)
        )
        for i=1:np
    ))

    #####
    # CONSTRAINTS
    #####
    # Initial conds
    @NLconstraint(NMPCMultmod, eqInit[i=1:s], x[i,1]-Ftrunc(xk[1],xk[2],u[1],i)

```

```
        == 0)
@constraint(NMPCMultmod, dudiff0, du[1]-(u[1]-uin) == 0)

#@constraint(NMPCMultmod, init[i=1:s], x[i,1] == xk[i])
#@constraint(NMPCMultmod, u[1] - uin == 0)

@NLconstraint(NMPCMultmod, eq[i=1:s,j=2:np], x[i,j]-
        Ftrunc(x[1,j-1],x[2,j-1],u[j],i) == 0)

@constraint(NMPCMultmod, xlconstr[i=1:np], x[1,i] <= xlmax)
@constraint(NMPCMultmod, duconstr0[i=nm+1:np], du[i] == 0)
@constraint(NMPCMultmod, dudiff[i=2:np], du[i]-(u[i]-u[i-1]) == 0)

#####
# SOLVE
#####
sol = JuMP.optimize!(NMPCMultmod)

u_opt = JuMP.value.(u[1])
CPUtime = solve_time(NMPCMultmod)
return u_opt, CPUtime
end

s = 2          # no of states
t0 = 0
tend = 60
N = 60
dt = tend/N
np = 5
nm = 3
hor = [np,nm]
x0 = [1,1]
u0 = 0.3
dumax = 0.05

# Arrays for 1 pt collocation
uSim1 = Array{Float64,1}(undef, N+1)
xSim1 = Array{Float64,2}(undef, N+1, 2)
xl_dev1 = Array{Float64,1}(undef, N+1)
CPU1Sim = Array{Float64,1}(undef, N)
# Arrays for 3 pt collocation
uSim3 = Array{Float64,1}(undef, N+1)
xSim3 = Array{Float64,2}(undef, N+1, 2)
xl_dev3 = Array{Float64,1}(undef, N+1)
CPU3Sim = Array{Float64,1}(undef, N)
# Arrays for 5 pt collocation
uSim5 = Array{Float64,1}(undef, N+1)
xSim5 = Array{Float64,2}(undef, N+1, 2)
xl_dev5 = Array{Float64,1}(undef, N+1)
CPU5Sim = Array{Float64,1}(undef, N)
# Arrays for multiple shooting
uSimM = Array{Float64,1}(undef, N+1)
xSimM = Array{Float64,2}(undef, N+1, 2)
xl_devM = Array{Float64,1}(undef, N+1)
CPUMSim = Array{Float64,1}(undef, N)
# General arrays
timeSim = Array{Float64,1}(undef, N+1)
x_sp = [1.5302*ones(1,N÷3+1) 0.9951*ones(1,N÷3) 0.0*ones(1,N÷3)]
spSim = Array{Float64,1}(undef, N+1)

for i = 1:N
    xl_sp = x_sp[Int(i)]
    spSim[Int(i)] = xl_sp
    # Disturbance
    if Int(i)==1
        test = Plant(x0,dt,u0)
        timeSim[1] = t0
        xSim1[1,1:2] .= x0
```

```

xSim3[1,1:2] .= x0
xSim5[1,1:2] .= x0
xSimM[1,1:2] .= x0
x1_dev1[1] = xSim1[1,1] - x1_sp
x1_dev3[1] = xSim3[1,1] - x1_sp
x1_dev5[1] = xSim5[1,1] - x1_sp
x1_devM[1] = xSimM[1,1] - x1_sp
uSim1[1] = u0
uSim3[1] = u0
uSim5[1] = u0
uSimM[1] = u0
end
# Optimal input using NMPC function
uSim1[Int(i)+1], CPU1Sim[i] = NMPCColl(xSim1[Int(i),1:s], uSim1[Int(i)],
                                       x1_sp, hor, dt)
uSim3[Int(i)+1], CPU3Sim[i] = NMPCCol3(xSim3[Int(i),1:s], uSim3[Int(i)],
                                       x1_sp, hor, dt)
uSim5[Int(i)+1], CPU5Sim[i] = NMPCCol5(xSim5[Int(i),1:s], uSim5[Int(i)],
                                       x1_sp, hor, dt)
uSimM[Int(i)+1], CPUMSim[i] = NMPCMult(xSimM[Int(i),1:s], uSimM[Int(i)],
                                       x1_sp, hor, dt)
#Simulating the plant behavior during dt
xSim1[Int(i)+1,:] = Plant(xSim1[Int(i),:], dt, uSim1[Int(i)+1])
xSim3[Int(i)+1,:] = Plant(xSim3[Int(i),:], dt, uSim3[Int(i)+1])
xSim5[Int(i)+1,:] = Plant(xSim5[Int(i),:], dt, uSim5[Int(i)+1])
xSimM[Int(i)+1,:] = Plant(xSimM[Int(i),:], dt, uSimM[Int(i)+1])
# for plotting
#timeSim[Int(i)] = (Int(i)-1)*dt
timeSim[i+1] = timeSim[i] + dt
x1_dev1[Int(i)+1] = xSim1[Int(i)+1,1] - x_sp[Int(i)]
x1_dev3[Int(i)+1] = xSim3[Int(i)+1,1] - x_sp[Int(i)]
x1_dev5[Int(i)+1] = xSim5[Int(i)+1,1] - x_sp[Int(i)]
x1_devM[Int(i)+1] = xSimM[Int(i)+1,1] - x_sp[Int(i)]
end
#timeSim[end] = tend # GJ RE DETTE BEDRE

u_grid_b1 = Array{Float64,1}(undef, N+1)
u_grid_t1 = Array{Float64,1}(undef, N+1)
u_grid_b3 = Array{Float64,1}(undef, N+1)
u_grid_t3 = Array{Float64,1}(undef, N+1)

u_grid_b1[1] = uSim1[1] - dumax
u_grid_t1[1] = uSim1[1] + dumax
u_grid_b3[1] = uSim3[1] - dumax
u_grid_t3[1] = uSim3[1] + dumax
for i = 1:N
    u_grid_b1[i+1] = uSim1[i] - dumax
    u_grid_t1[i+1] = uSim1[i] + dumax
    u_grid_b3[i+1] = uSim3[i] - dumax
    u_grid_t3[i+1] = uSim3[i] + dumax
end

#####
### PLOTTING
#####
yline(timeSim) = 0*timeSim
uPlot1 = [uSim1 u_grid_t1 u_grid_b1]
uPlot3 = [uSim3 u_grid_t3 u_grid_b3]
x1Plot = [xSim1[:,1], xSim3[:,1], xSimM[:,1], spSim]
xPlot = [xSim1, xSim3, xSimM, spSim]
devPlot = [x1_dev1, x1_dev3, x1_devM, yline]
#plot(timeSim, xPlot)

##### PYPLOT
compPlot = figure(figsize=(9,7))
PyPlot.plot(timeSim, spSim, color="purple", label="setpoint", linestyle=":", lw=0.7)
PyPlot.plot(timeSim, xSim1[:,1], color="royalblue", label=L"OC1 $x_1$",
            linestyle="--", marker="1", lw=0.7)

```

```

PyPlot.plot(timeSim,xSim1[:,2], color="royalblue",label=L"OC1 $x_2$",
            linestyle="--",marker="1",lw=0.7)
PyPlot.plot(timeSim,xSim3[:,1], color="crimson",label=L"OC3 $x_1$",
            linestyle="--",marker="2",lw=0.7)
PyPlot.plot(timeSim,xSim3[:,2], color="crimson",label=L"OC3 $x_2$",
            linestyle="--",marker="2",lw=0.7)
PyPlot.plot(timeSim,xSim5[:,1], color="lightcoral",label=L"OC5 $x_1$",
            linestyle="--",marker="3",lw=0.7)
PyPlot.plot(timeSim,xSim5[:,2], color="lightcoral",label=L"OC5 $x_2$",
            linestyle="--",marker="3",lw=0.7)
PyPlot.plot(timeSim,xSimM[:,1], color="skyblue",label=L"MS $x_1$",
            linestyle="--",marker="4",lw=0.7)
PyPlot.plot(timeSim,xSimM[:,2], color="skyblue",label=L"MS $x_2$",
            linestyle="--",marker="4",lw=0.7)

PyPlot.xlabel("Time [hr]")
PyPlot.ylabel("States [g/L]")
PyPlot.legend()
PyPlot.title(L"N = " * string(N) * "L", $n_p$ = " * string(np) * "L", $n_m$ = " * string(nm))
#PyPlot.savefig("/Users/agnescamilla/Documents/NTNU/PROSJEKT/Figurer/
                #NewModEveryIter/comp_FE=" * string(dt)
                # * "_N=" * string(N) * "_np=" * string(np) * "_nm="
                # * string(nm) * ".pdf")

devfig = figure(figsize=(9,7))
PyPlot.scatter(timeSim,x1_dev1,color="royalblue",label="OC1",marker="o",lw=0.7)
PyPlot.scatter(timeSim,x1_dev3,color="crimson",label="OC3",marker="v",lw=0.7)
PyPlot.scatter(timeSim,x1_dev5,color="lightcoral",label="OC5",marker="s",lw=0.7)
PyPlot.scatter(timeSim,x1_devM,color="skyblue",label="MS",marker="d",lw=0.7)
PyPlot.axhline(y=0,xmin=0,xmax=1,label="zero deviation",lw=0.7)
PyPlot.xlabel("Time [hr]")
PyPlot.ylabel(L"$x_1 - x_{\text{sp}}$ [g/L]")
PyPlot.title(L"N = " * string(N) * "L", $n_p$ = " * string(np) * "L", $n_m$ = " * string(nm))
PyPlot.legend()
#PyPlot.savefig("/Users/agnescamilla/Documents/NTNU/PROSJEKT/Figurer/
                #NewModEveryIter/dev_FE=" * string(dt) * "_N="
                # * string(N) * "_np=" * string(np) * "_nm="
                # * string(nm) * ".pdf")

CPUfig = figure(figsize=(9,7))
PyPlot.subplot(211)
PyPlot.scatter(timeSim[1:end-1],CPU1Sim, color="royalblue",label="OC1",marker="o")
PyPlot.scatter(timeSim[1:end-1],CPU3Sim, color="crimson",label="OC3",marker="v")
PyPlot.scatter(timeSim[1:end-1],CPU5Sim, color="lightcoral",label="OC5",marker="s")
PyPlot.scatter(timeSim[1:end-1],CPUMSim, color="skyblue",label="MS",marker="d")
PyPlot.xlabel("Time [hr]")
PyPlot.ylabel("CPU time [s]")
PyPlot.legend()
PyPlot.subplot(212)
PyPlot.scatter(timeSim[2:end-1],CPU1Sim[2:end], color="royalblue",label="OC1",
            marker="o")
PyPlot.scatter(timeSim[2:end-1],CPU3Sim[2:end], color="crimson",label="OC3",
            marker="v")
PyPlot.scatter(timeSim[2:end-1],CPU5Sim[2:end], color="lightcoral",label="OC5",
            marker="s")
PyPlot.scatter(timeSim[2:end-1],CPUMSim[2:end], color="skyblue",label="MS",
            marker="d")

PyPlot.xlabel("Time [hr]")
PyPlot.ylabel("CPU time [s]")
PyPlot.ylim(0,2)
PyPlot.title(L"N = " * string(N) * "L", $n_p$ = " * string(np) * "L", $n_m$ = " * string(nm))
PyPlot.legend()
#PyPlot.savefig("/Users/agnescamilla/Documents/NTNU/PROSJEKT/Figurer/
                #NewModEveryIter/CPU_FE=" * string(dt)
                # * "_N=" * string(N) * "_np=" * string(np)

```

```

#*_nm="*string(nm)*".pdf")

CPUfig3 = figure(figsize=(9,7))
PyPlot.subplot(311)
PyPlot.title(L"N = "*string(N)*L", $n_p$ = "*string(np)*L", $n_m$ = "*string(nm)")
PyPlot.scatter(timeSim[1:end-1],CPU1Sim, color="royalblue",label="OC1",
               marker="o")
PyPlot.scatter(timeSim[1:end-1],CPU3Sim, color="crimson",label="OC3",marker="v")
PyPlot.scatter(timeSim[1:end-1],CPU5Sim, color="lightcoral",label="OC5",
               marker="s")
PyPlot.scatter(timeSim[1:end-1],CPUMSim, color="skyblue",label="MS",marker="d")
#PyPlot.xlabel("Time [hr]")
PyPlot.ylabel("CPU time [s]")
PyPlot.legend()
PyPlot.subplot(312)
PyPlot.scatter(timeSim[1:end-1],CPU1Sim[1:end], color="royalblue",label="OC1",
               marker="o")
PyPlot.scatter(timeSim[1:end-1],CPU3Sim[1:end], color="crimson",label="OC3",
               marker="v")
PyPlot.scatter(timeSim[1:end-1],CPU5Sim[1:end], color="lightcoral",label="OC5",
               marker="s")
PyPlot.scatter(timeSim[2:end-1],CPUMSim[2:end], color="skyblue",label="MS",
               marker="d")

#PyPlot.xlabel("Time [hr]")
PyPlot.ylabel("CPU time [s]")
PyPlot.title(L"N = "*string(N)*L", $n_p$ = "*string(np)*L", $n_m$ = "*string(nm)")
PyPlot.subplot(313)
PyPlot.scatter(timeSim[1:end-1],CPU1Sim[1:end], color="royalblue",label="OC1",
               marker="o")
PyPlot.scatter(timeSim[1:end-1],CPU3Sim[1:end], color="crimson",label="OC3",
               marker="v")
PyPlot.scatter(timeSim[1:end-1],CPU5Sim[1:end], color="lightcoral",label="OC5",
               marker="s")
PyPlot.scatter(timeSim[2:end-1],CPUMSim[2:end], color="skyblue",label="MS",
               marker="d")

PyPlot.xlabel("Time [hr]")
PyPlot.ylabel("CPU time [s]")
PyPlot.ylim(0,2)
PyPlot.title(L"N = "*string(N)*L", $n_p$ = "*string(np)*L", $n_m$ = "*string(nm)")
#PyPlot.savefig("/Users/agnescamilla/Documents/NTNU/PROSJEKT/Figurer/
               #NewModEveryIter/CPU3_FE="*string(dt)
               #*_N="*string(N)*"_np="*string(np)
               #*_nm="*string(nm)*".pdf")

```