Nonlinear Model Predictive Control in JModelica.org

Magdalena Axelsson

LUND UNIVERSITY

Department of Automatic Control

MSc Thesis
ISRN LUTFD2/TFRT--5987--SE
ISSN 0280-5316

Department of Automatic Control
Lund University
Box 118
SE-221 00 LUND
Sweden

Printed in Sweden by Tryckeriet i E-huset
Lund 2015

Abstract

In this thesis, a stronger support for Model Predictive Control (MPC) in JModelica.org has been implemented. JModelica.org is an open-source software for simulation and optimization of systems described by Modelica models. MPC is an optimization-based control strategy where one formulates an Optimal Control Problem (OCP) to describe the aim of the controller. At discrete time points the state of the system is estimated and the OCP is solved to find the optimal input to apply to the system. The main goal of this thesis has been to make the time it takes to obtain the optimal input as short as possible and also streamlining the setup of MPC in JModelica.org. This has been done by implementing an MPC class, which utilizes the fact that the structure of the OCP is the same in each consecutive sample for efficiency. Two different benchmarks, one on a smaller problem and one on a larger problem, shows that by using the new MPC framework we obtain similar results as before, but considerably faster. The total average computation time for one sample is decreased by almost 60% for the large problem and by almost 90% for the smaller problem.

Acknowledgements

First and foremost I would like to give a massive thank you to my supervisor, Fredrik Magnusson, who've been a great support during this entire project. Thank you for everything, Fredrik.
I would also like to thank all the people from Modelon AB who've shown interest in this project. Thank you for all your feedback and expertise input on both JModelica.org and MPC.
Lastly I would like to thank Christian Hartlep from Siemens AG, who tested the MPC class during different stages of the development phase. Thank you for providing some very useful feedback on the MPC class from a users point of view and on how it works on much larger systems than the ones I've tested.

Contents

1

Introduction

1.1 Background

Model Predictive Control is an advanced, optimization-based, control strategy which originated some time in the late 1970's. Its popularity quickly grew due to its intuitive, easy-to-tune nature and ability to take constraints into account. It also allows for operation closer to the constraint limits (compared to conventional control), which leads to a more profitable operation. It has been used mainly in the petrochemical and chemical process industry, where control update rates are relatively low and there is plenty of computation time available for the optimization. However, with better computing hardware and increasingly efficient optimization algorithms the application prospects for MPC is broadening.

The main idea of Model Predictive Control (MPC) is to utilize a model of the system dynamics to predict and optimize the future behaviour of the system. The control strategy is based on the repeated on-line solution of an open-loop optimal control problem at discrete timepoints on the horizon. Feedback is incorporated by measureing the state at each of these discrete timepoints and using the measured state as the initial state in the optimal control problem. From each optimization the first input in the optimal control sequence computed is applied to the system. Two of the main advantages of MPC compared to other control methods are:

- it easily extends to multivariable processes.
- it intrinsically handles constraints both on the outputs and inputs.

In general, one distinguishes between linear and nonlinear model predictive control (LMPC/NMPC). In case of linear MPC, where the system model and any constraints imposed upon the system are linear, the optimal control problem can be cast as the solution of a quadratic problem. Quadratic problems can be solved efficiently on-line. In case of nonlinear MPC, the optimal control problem is instead cast as the solution of a nonlinear programming problem, which is more computationally demanding to solve. The long computation time of the optimization, along with the risk that sometimes a feasible solution is not found at all, are two of the main

limiting factors for successful application of NMPC in industry. [Allgöwer et al., 2004].

JModelica.org is an open-source platform for simulation, optimization and analysis of complex dynamic systems. In recent research its use has been proposed for the solution of the optimal control problem for NMPC applications in several different fields, including:

- [Cavey et al., 2014] where a JModelica.org/NMPC scheme was successfully implemented to control the heating system in a building.
- [Berntorp and Magnusson, 2015] where the use of JModelica.org was proposed to solve the NMPC optimal control problem in a hierarchical predictive control scheme for the lane keeping of a vechicle.
- [Larsson et al., 2013] where a case study of the start up of a combined cycle power plant using a JModelica.org/NMPC scheme was made.

Another popular software which can be used for the solution of the optimal control problem in NMPC applications is ACADO Toolkit[1], which is presented in [Houska et al., 2011].

Since the term MPC includes both NMPC and LMPC, we will make no further distinction between MPC and NMPC, and simply use the term MPC in the remainder of this thesis.

1.2 An introductory example

To illustrate what can be achieved using MPC we are going to present a simple example. The example is based on the Hicks-Ray Continuously Stirred Tank Reactor system (CSTR) [Hicks and Ray, 1971]. The system has two states, the concentration c and the temperature T. The input to the system is the temperature of the cooling flow T_c. The chemical reaction in the tank is exothermic, meaning it generates heat. It is also temperature dependent, meaning a high temperature results in a high reaction rate. The system dynamics can be described with the following set of differential equations:

$$\dot{c}(t) = F_0 \cdot \frac{c_0 - c(t)}{V} - k_0 \cdot c^{-\frac{EdivR}{T(t)}} \tag{1.1}$$

$$\dot{T}(t) = F_0 \cdot \frac{T_0 - T(t)}{V} - \frac{dH}{\rho \cdot Cp \cdot k_0 \cdot c^{-EdivR/T(t)}} + \frac{2U}{(r \cdot \rho \cdot C_p) \cdot (T_c(t) - T(t))} \tag{1.2}$$

The important variables to take notice of are c and T, which are the states, and T_c which is the temperature of the cooling flow. F_0, c_0 and T_0 are parameters

[1] **http://acado.github.io/index.html**.

which denote the flow rate, concentration and temperature of the inflowing liquid respectively. All other parameters are different kind of thermodynamic and geometric constants.

The aim of our controller is to take the system from the initial state, where the reactor is cold and the reaction rate is low, to the reference state (c_{ref}, T_{ref}), where both the temperature and the reaction rate are higher. To achieve this we define a cost function which penalizes the deviation of the states from their reference values over an optimization horizon of 150s. To this cost function we also add a penalty on the deviation of T_c from a reference value $T_{c_{ref}}$.

We limit the actuator in the system to yield a cooling temperature from 230°C to 370°C. For safety reasons we also limit the temperature in the reactor to a maximum of 350°C. The optimal control problem thus becomes:

minimize

$$\int_{t_0}^{t_0+150} (c_{ref} - c(t))^2 + (T_{ref} - T(t))^2 + (T_{c_{ref}} - T_c(t))^2 dt \tag{1.3}$$

with respect to

$$c \in \mathbb{R}, \qquad T \in \mathbb{R}, \qquad T_c \in \mathbb{R},$$

subject to

$$\begin{aligned}
&\text{Equations (1.1) and (1.2)} \\
&c(t_0) = c_{init}, \qquad T(t_0) = T_{init}, && (1.4)\\
&230 \leq T_c(t) \leq 370, && (1.5)\\
&T(t) \leq 350, && (1.6)\\
&\forall t \in [t_0, t_0 + 150].
\end{aligned}$$

Where equations (1.1) and (1.2) describe the system dynamics, as defined above, and (1.3) is the cost function. The initial conditions are defined in (1.4), where the state values at the beginning of the optimization horizon are set to the initial condition parameters c_{init} and T_{init}. In each MPC sample, the values of c_{init} and T_{init} are updated with the measurements obtained. (1.5) and (1.6) are the constraints imposed upon the system.

We will use a sample period of 3 seconds and control the system over a horizon of 50 samples, making the control horizon defined in time $3 \cdot 50 = 150$ seconds long. For each sample k, measurements will be obtained by simulating the system from the initial state of sample $k-1$, with the optimal input obtained from the $k-1$:th sample. The input obtained at each sample point is held constant until the next sample point. To emulate model mismatch and disturbances we will add normally distributed noise, with mean value 0 and standard deviation 0.5% of the current state value, to each of the measurements. The result is illustrated in the following plot:

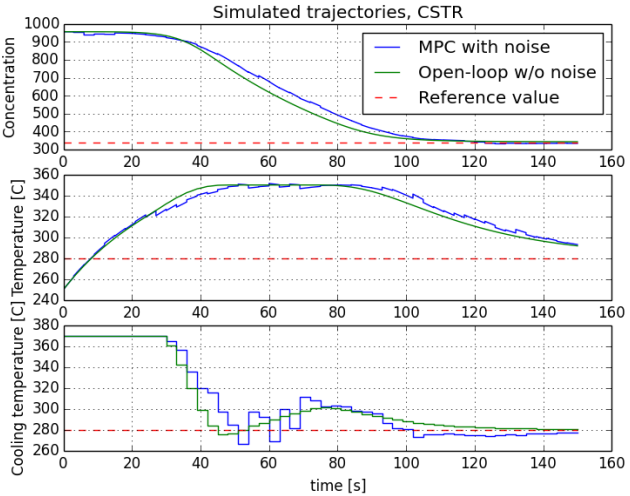

Figure 1.1 CSTR trajectories

The reference values for the states and the control variable are illustrated with the red dashed lines. The blue lines are the MPC-result, that is the result of the first sample period for each of the 50 samples patched together. The little vertical line at each of the sample points illustrates the noise. For comparison, the result of the corresponding open-loop optimization without noise has been plotted in green. In a perfect world, without any noise or model mismatch, the blue and the green lines would have been identical.

Solving this we used the existing optimization algorithm in JModelica.org. The optimization algorithm is divided into three steps; pre-processing, solution and post-processing. In pre-processing the optimization problem is discretized, in solution the problem is solved and in post-processing the result is extracted and presented to the user. The total and average times for the three different steps are printed on the terminal:

```
Total times for 50 samples (average time per sample in
    parenthesis)

Total time: 8.46 seconds                (0.169)
Pre-processing time: 7.13 seconds       (0.143)
Solution time: 0.87 seconds             (0.017)
Post-processing time: 0.46 seconds      (0.009)
```

The Modelica and Optimica code for the CSTR will be presented in section 4.1.

1.3 Aim of thesis

The shortest sample period one can choose for their MPC controller is dependent on the total computational time each sample takes to complete. In turn, each sample is heavily dependent on the total time for the optimal control problem to be solved. The aim of this thesis is to make each consecutive sample as fast as possible. This will allow for shorter sample periods, which will make MPC applicable to systems where faster dynamics require short sample periods.

To achieve this in JModelica.org an MPC framework will be implemented. JModelica.org has a framework for solving open-loop optimal control problems. The MPC framework will utilize this framework and exploit the fact that the only thing that changes in the optimization problem between each consecutive sample is the initial condition. Focus shall also be on usability, and the MPC framework shall therefore also provide methods the user might need when using JModelica.org for MPC applications.

1.4 Thesis outline

In chapter 2 we will present some Model Predictive Control theory and important things to keep in mind when using MPC as a control method. In chapter 3 we will present the optimization algorithm in JModelica.org and describe the theory behind the disctretization scheme used in it to make it possible to solve the optimization problem numerically. In chapter 4 we will present the MPC class implemented in this thesis; both how to use it and what features it includes. In chapter 5 we will run some MPC simulations to test the performance of the MPC class. Finally, in chapter 6, some conclusions will be presented.

2

Model predictive control

In this chapter we present the basic theory of Model Predictive Control and the MPC algorithm. We start by introducing the general concept of Model Predictive Control and move on to describe the framework we will use. Later we present why softening of constraints is important, what warm start is and how to compensate for computation times if they are not negligible.

2.1 Introduction

The term Model Predictive Control is a collection name of advanced control methods which all contain the following elements:

- A prediction model
- An objective function
- A control algorithm

The model is used to predict the output at future time instants. Various types of models can be used and the most important thing is that the model is able to capture the process dynamics; it should represent the relationship between inputs and outputs in the system as accurately as possible. Models that are commonly used are impulse response-, step response- and state-space models [Camacho and Alba, 2013]. These often describe linear, or linearized, systems in discrete time. In this thesis we are going to use non-linear models described by a set of differential algebraic equations (DAE:s) in continuous time. The term Non-linear Model Predictive Control (NMPC) refers to the use of a non-linear model.

In [Maciejowski, 2002, Ch.2], a linear, discrete-time, state-space model is used to describe the system. That is, the model is on the form

$$x(k+1) = Ax(k) + Bu(k) \tag{2.1}$$

where $x \in \mathbb{R}^{n_x}$ denotes the states, $u \in \mathbb{R}^{n_u}$ denotes the control variables and the index k denotes the sample points. This formulation of the model coincides well with that

in other model predictive control literature [Grüne and Pannek, 2011][Camacho and Alba, 2013][Rawlings and Mayne, 2009]. In the formulation above we have omitted the equation defining the measurable states $v(k) = C_v x(k)$. The reason for this is that we assume that an estimate of the states which are not measurable can be obtained. A state estimator uses the values of the measurable variables and inputs up to time k to estimate the complete state. That is, it uses measurements up to $v(k)$ and knowledge of the inputs up to $u(k-1)$ to estimate the value of the state $x_{est}(k)$. This means that new values, whether they be measurements or estimates, can be obtained for the complete state in each sample.

The primary applications for MPC are tracking and stabilization problems, meaning that the controlled outputs should follow a reference trajectory as closely as possible. Therefore, the objective function of the optimization problem is typically formulated so that it penalizes deviations from the reference trajectory. The objective function can also include different forms of penalties on the input.

In practice all processes are subject to constraints. These include physical constraints, such as actuators that have a limited working range and slew rate as well as constructive, safety and environmental constraints imposed on the process to make sure it is operating in a safe and desired manner. Examples of constraints included in the second category are; maximum and/or minimum levels in tanks, flows in pipes, temperatures and pressures. Optimization problems handle constraints intrinsically.

The control algorithm is carried out at each sample point k and consists of the following three actions;

1. Obtain estimates $x_{est}(k)$.

2. Compute the input $u(k)$ that minimizes the objective function.

3. Apply the input $u(k)$ to the system.

There are some special cases in which the input can be found analytically, but in most cases an iterative method of optimization is needed. This means that there will always be a computational delay between step 1 and step 3. For systems with slow dynamics and long sampling intervals this delay is usually negligible. Otherwise, this computational delay needs to be compensated for. How to compensate for computational delays will be described in section 2.4.

When using MPC one also needs to specify a *sample period*. The sample period denotes the amount of time between two sample points, that is the length of one *sample*. The optimal input obtained at each sample point is often assumed to be held constant until the next sample point, where the control algorithm steps are repeated and a new optimal input is obtained.

2.2 Nonlinear model predictive control

Model

In this thesis we will consider prediction models that describe the system dynamics with a set of fully implicit differential algebraic equations. That is, the system dynamics are described by equations of the form

$$F(\dot{x}(t), x(t), u(t), y(t)) = 0, \qquad \forall t \in [t_0, t_f]$$

where $t \in \mathbb{R}$ denotes the time and $y \in \mathbb{R}^{n_y}$ denotes the algebraic variables. Algebraic variables are variables whose derivative is not a part of the system. Initial conditions are given on the form

$$x(t_0) = x_0, \tag{2.2}$$

where x_0 is a parameter representing the initial value of variable x. The reason we introduce the values as parameters will be clear later on but it is linked with the actual implementation, and the fact that we need to be able to change them between optimizations. For ease of notation, we introduce a new time-dependent variable $z = (\dot{x}, x, y, u) \in \mathbb{R}^{2n_x+n_y+n_u}$. This allows us to write the system dynamics equations on the form

$$F(z(t)) = 0 \tag{2.3}$$

where

$$F : \mathbb{R}^{n_z} \to \mathbb{R}^{n_x+n_y} \tag{2.4}$$

The non-free variables ($x(t)$ and $y(t)$) should be determinable from the DAE system given fixed values of the control variables $u(t)$. This means that the DAE system should contain $n_x + n_y$ scalar equations.

Objective function

The objective function describes what we want the controller to minimize. For tracking problems, where the aim is to get the controlled outputs to follow a reference trajectory, one usually adds a quadratic penalty on the error. In other words, the objective function is defined so that it penalizes the deviation of the predicted controlled outputs $w(t)$, from the reference trajectory $w_{ref}(t)$ over the entire prediction horizon H_p. We may choose any of the variables in $z(t)$ to be the controlled outputs. The objective function $f(t_0, t_f, z(t))$ thus becomes,

$$f(t_0, t_f, z(t)) = \int_{t_0}^{t_f} (w_{ref}(t) - w(t))^T Q (w_{ref}(t) - w(t)) dt \tag{2.5}$$

where t_0 is the start time and $t_f = t_0 + H_p$ is the final time of each optimization.

Constraints may also be imposed upon the system. The different types of constraints we consider are *variable bounds*, *path constraints* and *terminal constraints*. Variable bounds are enforced during the entire prediction horizon and have a constant upper and lower limit. Path constraints are also enforced during the entire time

horizon with the difference that they have no constant limit. A variable bound is thus a form of path constraint, or rather, a path constraint is a generalization of a variable bound. The reason we separate these is because bounds can be treated more efficiently by many numerical algorithms. Terminal constraints on the other hand are only enforced at the end of the prediction horizon. Terminal constraints can be used to ensure stability in the closed loop system.

Combining the model, the objective function and the constraints creates a typical Optimal Control Problem (OCP). The goal of an OCP is to find the input trajectory that minimizes the objective function, while at the same time upholding the constraints imposed upon the system. We will denote the OCP to be solved in each sample as P, where P is;

minimize

$$f(t_0, t_f, z(t)) = \int_{t_0}^{t_f} (w_{ref}(t) - w(t))^T Q(w_{ref}(t) - w(t))dt \tag{2.6}$$

with respect to

$$x \in \mathbb{R}^{n_x}, \qquad u \in \mathbb{R}^{n_u}, \qquad y \in \mathbb{R}^{n_y},$$

subject to

$$F(z(t)) = 0, \tag{2.7}$$
$$x(t_0) = x_0, \tag{2.8}$$
$$z_L \leq z(t) \leq z_U, \tag{2.9}$$
$$g(z(t)) \leq 0, \tag{2.10}$$
$$G(z(t_f)) \leq 0, \tag{2.11}$$
$$\forall t \in [t_0, t_f].$$

In this formulation of the optimal control problem (2.6) is the objective function, (2.7) are the model equations, (2.8) are the initial conditions, (2.9) are the variable bounds with lower limit z_L and upper limit z_U, (2.10) are the path constraints and (2.11) are the terminal constraints.

In MPC formulations it is common to enforce the input to be piecewise constant on the prediction horizon, meaning the input is only allowed to change at specific time instants. The result of an optimization will thus be a discrete input sequence $u = \{u_1, u_2, ..., u_N\}$, rather than a continuous input trajectory $u = u(t)$. The *control horizon* denotes during how long time on each prediction horizon the input is allowed to change. Once the control horizon ends no more control actions are allowed and the input is to remain constant for the rest of the prediction horizon. Shortening the control horizon decreases the number of inputs in the resulting control sequence, and is thus a way of decreasing the number of optimization variables in the discretized optimal control problem. In JModelica.org the control horizon is defined by the option 'blocking_factors'. The control horizon will be discussed more in section 3.3.

Control algorithm

The general idea with MPC is that P is to be solved on-line at each sample point t_k on the MPC simulation horizon. The solution to P will determine the input that is to be applied to the system until the next sample point t_{k+1}. The time between two sample points is called the sample period and will be denoted h.

The control algorithm states that at each sample point t_k, the following steps should be carried out:

1. Obtain estimates of the state $x_{est}(t_k)$.
2. Set the initial condition parameters $x_0 = x_{est}(t_k)$, the start time $t_0 = t_k$ and the final time $t_f = t_0 + H_p$. Solve P.
3. Use the first value in the resulting control sequence u_1 as input to the system. Hold the input constant through the entire sample period.

This is the most basic formulation of the control algorithm. In sections 2.4 and 2.5 we will extend this algorithm to include compensation of computation times and warm start of the optimization.

2.3 Softening constraints

A serious problem that can occur when solving an optimization problem is that the solver is faced with an infeasible problem. There are two main reasons that causes infeasibility; either the problem is not well-posed and thus *is* infeasible, or an initial condition has violated a constraint and thus made the problem infeasible. A constraint might be violated if the process is running close to a limit and a particularly large disturbance occurs, or if the model is not good enough and the process behaves differently than predicted. Infeasibility caused by constraint violations can be prevented by 'softening' the constraints. This means that rather than to regard constraints as 'hard' limits which may never be crossed, we 'soften' them by allowing them to be crossed, but only if necessary. Here it is important to make a distinction between constraints on the inputs and constraints on the states. Input constraints usually stem from physical limits on the actuators and therefore they simply cannot be softened. More importantly though, input constraints do not directly cause infeasibility, so there is no need to soften them.

One way of softening an output constraint is by adding a new variable, a so-called "slack variable", to the problem. This slack variable is heavily penalized in the cost function and is defined in such a way that it is non-zero only if the constraint is violated. Going back to the original optimal control problem P, the parts that are affected by the softening are the cost function and the constraint that is to be softened. Assuming they are defined as follows before the softening:

minimize

$$f(z(t))$$

with respect to

$$z \in \mathbb{R}^{n_z}$$

subject to

$$c(z(t)) \leq 0$$

where f is the cost function and c is a constraint. The corresponding optimization problem with the constraint $c(z(t)) \leq 0$ softened would be:

minimize

$$f(z(t)) + \rho \|\varepsilon\|$$

with respect to

$$z \in \mathbb{R}^{n_z}, \varepsilon \in \mathbb{R}$$

subject to

$$c(z(t)) \leq \varepsilon(t) \tag{2.12}$$

$$\varepsilon(t) \geq 0 \tag{2.13}$$

where ρ is the constraint violation weight, ε is the new slack variable and $\|\varepsilon\|$ is typically the 1-norm, 2-norm or infinity-norm. The different norms are defined as

$$\|\varepsilon\| = \begin{cases} \int_{t_0}^{t_f} |\varepsilon| \, dt & \text{1-norm} \\ \sqrt{\int_{t_0}^{t_f} \varepsilon^2 \, dt} & \text{2-norm} \\ |\varepsilon_{max}| & \text{infinity-norm} \end{cases}$$

where ε_{max} is the maximum value $\varepsilon(t)$ obtains over the time horizon.

If softening more than one constraint, one can either use the same slack variable for all constraints or add a separate slack variable for each constraint that is to be softened. One drawback with using the same slack variable is that if one constraint needs to be violated, it will allow all the other constraints to be violated as well. The main advantage though, is that the problem size is only increased with one slack variable and one constraint, generally making the optimization easier. Adding a separate slack variable to each constraint that is to be softened may substantially increase the size of the optimization problem, but it ensures that each constraint is only violated if necessary.

Using $\rho = 0$ makes the problem completely unconstrained, but as $\rho \to \infty$ one will be back at the hard-constrained problem. By using the 1-norm or ∞-norm, it can be shown that a large enough ρ will prevent constraints from being violated unless absolutely necessary. That is, the solver will recover the same solution as with the hard constrained problem, if a feasible solution exists [Maciejowski, 2002, Sec. 3.4.]

2.4 Compensation of computation times

If the computation time needed to solve the optimization problem is not negligible compared to the sample period it needs to be compensated for. One way of compensating for the computation time is by using a state estimator to estimate the state values at a given time τ in the future. The idea is to, given estimates of the state at time t_m, use the state estimator to make a prediction of the state at time $t_k = t_m + \tau$ and solve the optimization problem with the predicted state values $\hat{x}(t_k)$ as initial states. This means that the optimization problem for a given sample point t_k is solved ahead of time, and the optimal input $u(t_k)$ is available by the time it is to be applied[Grüne and Pannek, 2011, Sec. 7.6.] The control algorithm in section 2.2.3. is thus modified to:

1. Obtain estimates of the state $x_{est}(t_m)$.
2. Use the state estimator to estimate the state $\hat{x}(t_k)$ of the system, where $t_k = t_m + \tau$.
3. Set the initial condition parameters $x_0 = \hat{x}(t_k)$, the start time $t_0 = t_k$ and the final time $t_f = t_0 + H_p$. Solve P.
4. Once $t = t_k$: Apply the first value in the resulting control sequence u_1.

This sequence assumes that τ is large enough that the time the state estimator needs to compute an estimate τ_e and the computation time for the solver τ_s combined is smaller than τ. In other words, $\tau_e + \tau_s < \tau$, so that the input u_1 is available by the time $t = t_k$.

2.5 Warm start of the MPC optimization

Step 4 of the modified MPC algorithm in the previous section states that we are to use the first element of the control sequence as input in the next sampling period. The solution obtained however, contains a lot more information than just the optimal control sequence. It also includes a prediction of the variable trajectories obtained by using the calculated optimal input. The MPC algorithm as previously stated does not specify what to do with the rest of the solution. In this thesis we will use an iterative solver to solve the optimal control problem. Iterative solvers only guarantee local convergence, which means that a good initial guess of all optimization variables is usually needed for the solver to converge. A good initial guess will also in most cases reduce the number of iterations needed to find the optimal solution. Due to this, we will use the solution of one optimization as the initial guess for the variables in the following optimization. We thus modify the control algorithm to:

1. Obtain estimates of the state $x_{est}(t_m)$.
2. Use the state estimator to estimate the state $\hat{x}(t_k)$ of the system, where $t_k = t_m + \tau$.
3. Set the initial condition parameters $x_0 = \hat{x}(t_k)$, the start time $t_0 = t_k$ and the final time $t_f = t_0 + H_p$. Set the initial guess of the optimization variables to the solution obtained in the previous optimization. Solve P.
4. Once $t = t_k$: Apply the first value in the resulting control sequence u_1.

For the first optimization, where there is no previous solution, one usually has to define the initial guess in some other way.

3

Optimization in JModelica.org

To solve the optimal control problem stated in the previous chapter we are going to use the existing optimization algorithm in JModelica.org which can be used to solve dynamic optimization problems. Optimal control problems, as well as parameter estimation problems and parameter optimization problems, are all different kinds of dynamic optimization problems. We will keep our focus on the optimal control problem but will henceforth refer to it as the optimization problem.

In this chapter we will present the optimization algorithm in JModelica.org and describe how it solves the optimization problem. We will also describe how it needs to be modified for us to be able to use it for our consecutive MPC-samples efficiently.

3.1 Defining the model and optimization problem

JModelica.org[1] is an extensible, open source platform for optimization, simulation and analysis of complex dynamic systems. It uses Modelica[2] models to describe the dynamics of the system and the Modelica extension Optimica[“Optimica—An Extension of Modelica Supporting Dynamic Optimization”] to formulate the optimal control problem.

Modelica is a freely available, object-oriented, equation based language for modeling of large and complex dynamic systems. The models are described by differential, algebraic and discrete equations and the goal of Modelica is to model the dynamic behavior of the system in a convenient way. The Modelica language has been developed by the non-profit organisation Modelica Association since 1996 and today it is used in a wide variety of fields such as chemistry, mechanics and

[1] **http://www.jmodelica.org/**.
[2] **https://www.modelica.org/**.

electronics. Today the open source Modelica Standard Library contains more than 1200 model components and 900 functions.

Optimica is an extension of Modelica with language constructs that enable the formulation of dynamic optimization problems based on Modelica models. Since Modelica was not developed with optimization in mind it lacks important features desireable for expressing optimization problems. This includes formulation of cost-functions, constraints, variable bounds and initial guesses, all of which can be expressed in Optimica.

Scripts for JModelica.org are written in Python. Python[3] is a general-purpose, open source, object-oriented, high-level programming language with a clear syntax for high code readability. All the code implemented in this thesis will be in Python, that is both the MPC class and the scripts for running it. We will use the Python packages SciPy[4] and NumPy[5] for scientific computations and matplotlib[6] for plotting purposes.

3.2 Collocation theory

To understand the optimization algorithm used in JModelica.org some theory needs to be covered.

Discretization using collocation

The optimization problem needs to be discretized in order for a numerical solver to solve it. By discretizing the problem we let a finite number of discrete time points on the optimization horizon represent the entire optimization problem. This means that we only consider the optimization problem at these discrete time points, rather than for all $t \in [t_0, t_f]$. It is this distinction that separates the infinite-dimensional optimization problem from a finite-dimensional nonlinear programming problem (NLP).

JModelica.org uses direct collocation to transcribe the optimization problem into an NLP. In this thesis we will only give an introductory description of collocation theory and how the transcription is executed. For a more thorough description on these topics the reader is referred to [Magnusson, 2012], [Biegler, 2010].

The collocation methods supported in JModelica.org are Radau and Gauss. They both start with dividing the optimization horizon into n_e number of elements, where the endpoint of each element is called the mesh point. In each element, n_c number of collocation points are placed. The placement of the collocation points depends on which collocation method is to be used. In this thesis we will consider Radau collocation. Radau collocation always places one collocation point at the mesh point,

[3] **https://www.python.org/**.
[4] **http://www.scipy.org/**.
[5] **http://www.numpy.org/**.
[6] **http://matplotlib.org/**.

and the others are placed in a way that maximizes accuracy. The collocation points are placed in the same way in all elements.

The collocation points are denoted $t_{i,k}$, where i specifies the element and k specifies which collocation point in said element we are referring to. The total number of collocation points becomes $n_e \cdot n_c$. We will denote the variables at each of these collocation points $z(t_{i,k}) = z_{i,k}$. This means that for each time-dependent variable in our original optimization problem, we will have a set of $n_e \cdot n_c$ variables in the NLP. That is

$$z(t), \qquad \forall t \in [t_0, t_f],$$

will be approximated by

$$z_{i,k}, \qquad i = 1, ..., n_e, \qquad k = 1, ..., n_c.$$

This in turn means that all constraints in the original optimization problem, which include a time-dependent variable, are transcribed into a set of $n_e \cdot n_c$ constraints in the NLP.

In a similar manner, the cost function is approximated by a sum of the costs at each collocation point. That is

$$f(t_0, t_f, z(t)) \approx \sum_{i=1}^{n_e} \left(l \sum_{k=1}^{n_c} \omega_k (w_{ref}(t_{i,k}) - w_{i,k})^T Q (w_{ref}(t_{i,k}) - w_{i,k}) \right) \tag{3.1}$$

where l is the length of each element and ω_k is called the quadrature weight and depends on the placement of the collocation points in the elements.

The variables x and y are approximated using Lagrange polynomials. Given a set of distinct points, in our case the collocation points $(t_{i,k}, z_{i,k})$, the corresponding Lagrange polynomial is the polynomial of least degree that at each point $t_{i,k}$ assumes the value $z_{i,k}$. One Lagrange polynomial is used in each element for all the x and y variables. To enforce continuity on the x variables, an extra collocation point is added at the beginning of each element. The extra collocation points are added as collocation points zero, that is $x_{i,0}$. By setting the value of x in these points equal to the value of x at the mesh point of the previous element, continuity is achieved. That is,

$$x_{i,n_c} = x_{i+1,0}. \tag{3.2}$$

An approximation for the x and y variables over the entire optimization horizon is thus obtained by gluing all the Lagrange polynomials together at the mesh points. An approximation of the derivative $\dot{x}$ is obtained by differentiating the polynomials representing the states x with respect to t.

The optimization variables in the resulting NLP, i.e the variables we wish to find a value which minimizes the cost-function for, are all the variables in $z_{i,k}$. All optimization variables are given a symbolic representation using CasADi.

CasADi CasADi (Computer algebra system with Automatic Differentiation) is an open-source, symbolic framework for automatic differentiation. It uses state-of-the-art algorithms for automatic differentiation to, in a fast and efficient way, calculate function derivatives. This is very useful for solving optimization problems. CasADi is used in JModelica.org to transcribe the Modelica and Optimica code to a format which is compatible with the NLP solvers [Lennernäs, 2013]. This is done by giving each optimization variable in the discretized NLP its own symbolic representation. The equations in the NLP are thus built of these symbolic representations of the different variables.

Solving the NLP

After the transcription, the resulting NLP has the general form

minimize

$$f(z) \tag{3.3}$$

with respect to

$$z \in \mathbb{R}^{n_z}, \tag{3.4}$$

subject to

$$g_e(z) = 0, \tag{3.5}$$
$$g_i(z) \leq 0, \tag{3.6}$$

where all the constraints have been categorized depending on whether they are equality constraints g_e or inequality constraints g_i. The model equations and initial conditions are thus included in g_e. To solve the NLP it is important to study the Lagrangian function, which is defined as

$$L(z, \lambda, \nu) = f(z) + \lambda \cdot g_e(z) + \nu \cdot g_i(z), \tag{3.7}$$

where $\lambda \in \mathbb{R}^{n_{g_e}}$ and $\nu \in \mathbb{R}^{n_{g_i}}$ are called the dual variables and z are the optimization variables, also called the primal variables. An optimal solution (z^*, λ^*, ν^*) to the NLP requires the Karush-Kuhn-Tucker (KKT) conditions to be satisfied. The KKT conditions are given by

$$\nabla_z L(z^*, \lambda^*, \nu^*) = \nabla f(z^*) + \lambda^* \cdot \nabla g_e(z^*) + \nu^* \cdot \nabla g_i(z^*) = 0, \tag{3.8}$$

$$g_e(z^*) = 0, \qquad g_i(z^*) \leq 0, \tag{3.9}$$
$$\nu^* \cdot g_i(z^*) = 0, \qquad \nu^* \geq 0. \tag{3.10}$$

Conditions (3.9) ensure that none of the constraints we defined are violated i.e. we have primal feasibility. Condition (3.8) ensures that the solution is a stationary point. Conditions (3.10) ensures that either the dual variable ν^* is zero, in which case the constraint is *inactive*, or that the constraint function $g_i(z^*)$ is zero, in which

case the constraint is *active*. To ensure that the optimal solution is not a local maximum, it also need to satisfy the additional condition

$$p \cdot \nabla^2_{zz} L(z^*, \lambda^*, \nu^*) \cdot p \geq 0, \qquad \forall p \neq 0, \nabla g_e(z^*) \cdot p = 0. \tag{3.11}$$

Which states that the second order derivative of the Lagrangian function needs to be positive, and the derivative of the equality constraints needs to be 0. The conditions mentioned above, together with a constraint qualification condition, are *necessary* for a point (z^*, λ^*, ν^*) to be a local minimum. If the inequality in (3.11) is changed to a strict inequality the conditions are also *sufficient* for the point to be a local minimum.

To solve the resulting NLP, JModelica.org support NLP-solvers IPOPT[7] and WORHP[8]. The two solvers use different algorithms to obtain the optimal solution to a nonlinear programming problem. WORHP uses a Sequential Quadratic Programming (SQP) method on the general non-linear level together with an Interior-Point (IP) method on the quadratic subproblem level, while IPOPT uses an IP method straight away. The differences in the algorithms will not be discussed in any more detail here, the interested reader is referred to [Wächter and Biegler, 2006], [Büskens and Wassel, 2013].

In the majority of this thesis we will use the default solver in JModelica.org, which is IPOPT, to solve the NLP. In IPOPT we will use the linear solver MA27, which is available through a HSL licence[9].

3.3 Optimization algorithm

The optimization algorithm is divided into three steps:

1. Pre-processing.
2. Solution.
3. Post-processing.

Pre-processing

In the pre-processing step, the optimization problem is transcribed into an NLP, as described in section 3.2. All optimization variables are given a symbolic representation using CasADi, while all the parameters are substituted for their respective values. The actual time points that each collocation point corresponds to are also calculated.

[7] **https://projects.coin-or.org/Ipopt/**.
[8] **http://www.worhp.de/**.
[9] **"HSL. A collection of Fortran codes for large scale scientific computation. http://www.hsl.rl.ac.uk/"**.

If the user has provided initial trajectories for the variables, which they should since a good initial guess is required to find a solution, the corresponding values for the variables at the collocation points are extracted and given as the initial guess to the NLP-solver.

Nominal trajectories for the variables are useful for scaling purposes, since it is easier for the solver to find a solution if the problem is well scaled. If the user has provided nominal trajectories, scaling factors for the variables are computed from them.

For the resulting NLP a solver object is created. When creating the solver object the optimization variables, the cost-function and the constraint equations need to be specified. Upper and lower bounds for the optimization variables and constraint equations are given to the solver object as inputs and are specified after the solver object has been created. Giving them to the solver as inputs means that we are able to change them between optimizations.

Solution

The solution step is handled completely by either IPOPT or WORHP and includes the iterative steps the solver takes to find an optimal solution to the NLP.

Post-processing

The solver object gives the result to all optimization variables in one long vector with the size n_z. The post-processing step includes processing the result from the solver so that it is presented to the user in an easy to comprehend way. This includes writing the result to file and creating a `LocalDAECollocationResult` object. The last step in creating the result object includes printing the times for the three different steps on the terminal.

Using the optimization algorithm for MPC

Each time the optimization algorithm is called the three steps mentioned above are executed, in the above order. The last step is almost always the fastest one. Which one of the first two steps that is the most time-consuming depends largely on the problem size, the initial guess provided to the solver and the collocation/optimization options chosen. A larger problem takes longer to transcribe into an NLP, but it will also typically increase the solution time. A good initial guess will decrease the solution time.

Making use of the fact that the only thing that we need to change between each consecutive MPC-sample is the initial condition for the states and the start and final times of the optimization horizon, we will save a lot of time if we use the same solver object for all optimizations. This does, however, require a slight modification to the optimization algorithm.

In the creation of the NLP, any parameters defined in the Modelica and Optimica files are substituted for their respective values. However, if we were to create

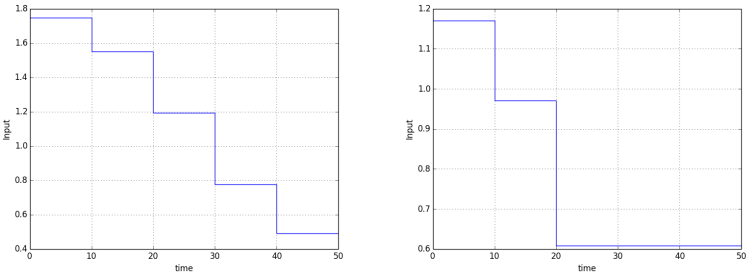

Figure 3.1 The plots show how the input is constrained in JModelica.org using the blocking factor option.

a symbolic representation for them using CasADi, like we do for all optimization variables, we could keep them as parameters in the resulting NLP. Keeping them as parameters in the NLP means that we will be able to change them between optimizations. This is the reason we defined the values of the initial condition as parameters in the system model in section 2.2.1.

To achieve the piecewise constant input which was discussed in section 2.2.2. in JModelica.org we use the optimization option called `'blocking_factors'`. For each input which shall be piecewise constant we define a blocking factor list containing ints, such that the sum of all elements in the list equals the number of collocation elements specified. The blocking factors will make the computed optimal inputs constant during the specified number of collocation elements at a time as defined by the blocking factor list. We illustrate this with an example with 5 collocation elements where each collocation element is 10 seconds long. In Figure 3.1 the plots illustrate the computed optimal input to the same optimization problem. In the left plot, we've used the blocking factor list `bf_list=[1,1,1,1,1]`, while in the right plot we've used `bf_list = [1,1,3]`. The sum of the elements in both cases are equal to 5, which is the number of collocation elements in the mesh, as defined by the collocation option `'n_e'`. We define the end of the control horizon as the point in time where the input is first constrained from changing and is held constant after. The control horizons in the two examples are thus 50 in the left plot and 30 in the right plot. The control horizon ends one element after the last change in input. The computed input sequence has 5 values in the left example and 3 values in the right example. Due to this, with all other options identical, the number of optimization variables in the NLP corresponding to the left optimization problem will contain two optimization variables more than the NLP corresponding to the right optimization problem.

4

MPC class

In this chapter the MPC class implemented in this thesis will be described. The chapter will begin with an example to show how the MPC class is used, and continue with a more in-depth description of the MPC class and the features included in it. At the end of the chapter we will also describe the changes made to the collocation algorithm in order to make the MPC class possible at all, and comment on some features which are not supported, or do not work as intended, with the MPC class.

4.1 MPC example

To show the performance of the MPC class we will return to the CSTR example discussed in chapter 1 and use the MPC class to solve it. The following Modelica code describes the CSTR system:

```
model CSTR "A CSTR"

  parameter Modelica.SIunits.VolumeFlowRate F0=100/1000/60 "Inflow";
  parameter Modelica.SIunits.Concentration c0=1000 "Concentration inflow
      ";
  Modelica.Blocks.Interfaces.RealInput Tc "Cooling temperature";
  parameter Modelica.SIunits.VolumeFlowRate F=100/1000/60 "Outflow";
  parameter Modelica.SIunits.Temp_K T0 = 350;
  parameter Modelica.SIunits.Length r = 0.219;
  parameter Real k0 = 7.2e10/60;
  parameter Real EdivR = 8750;
  parameter Real U = 915.6;
  parameter Real rho = 1000;
  parameter Real Cp = 0.239*1000;
  parameter Real dH = -5e4;
  parameter Modelica.SIunits.Volume V = 100 "Reactor Volume";
  parameter Modelica.SIunits.Concentration c_init = 1000;
  parameter Modelica.SIunits.Temp_K T_init = 350;
  Real c(start=c_init,fixed=true,nominal=c0);
  Real T(start=T_init,fixed=true,nominal=T0);
equation
  der(c) = F0*(c0-c)/V-k0*c*exp(-EdivR/T);
  der(T) = F0*(T0-T)/V-dH/(rho*Cp)*k0*c*exp(-EdivR/T)+2*U/(r*rho*Cp)*(
      Tc-T);
end CSTR;
```

The two equations under the keyword `equation` represents the model equations presented in chapter 1 (1.1), (1.2). The following Optimica code describes the optimization problem:

```
model CSTR_MPC_Model
  extends CSTR;
  parameter Real c_ref = 338.775766;
  parameter Real T_ref = 280.099198;
  parameter Real Tc_ref = 280;
end CSTR_MPC_Model;

optimization CSTR_MPC(objectiveIntegrand = (c_ref-c)^2 + (T_ref-T)^2 + (
    Tc_ref-Tc)^2, startTime=0.0, finalTime=150)

extends CSTR_MPC_Model(Tc(min=230, max=370, nominal=300), T(min=0, max
    =350, nominal=300), c(nominal=500));

end CSTR_MPC;
```

First, the original model have been extended with parameters representing the reference values. Next the optimization problem is defined where the objective function is stated after the keyword `objectiveIntegrand`. The `objectiveIntegrand` here corresponds to the objective function presented in chapter 1 (1.3). Variable bounds are stated using the keyword `min/max` after declaring the variables while constraints are added under the keyword `constraint`. The observant reader may have noticed that this definition of the optimization problem include variable bounds on T. To avoid the risk of infeasible initial conditions, that bound was softened manually in the introductory example. The MPC class however supports automatic softening of variable bounds, so we may proceed with this definition of the optimization problem. The automatic softening of variable bounds will be described in section 4.3.2.

Script

The script used to run the MPC simulation will be presented and described here. The reader is assumed to be somewhat familiar with optimization in JModelica.org, so the comments will mainly be on things that are new with the MPC class. First, initial trajectories are computed by creating a Functional Mock-up Unit (FMU) for the system and simulating it with the constant input `'Tc'=280`. From the initial trajectories, an initial guess for the optimization variables will be obtained.

```
# 1. Compute initial guess trajectories by means of simulation
# Locate the Modelica and Optimica code
file_path = os.path.join(get_files_path(), "CSTR.mop")

# Compile and load the model used for simulation
sim_fmu = compile_fmu("CSTR.CSTR_MPC_Model", file_path,
                      compiler_options={"state_initial_equations":True
                        })
sim_model = load_fmu(sim_fmu)

# Define stationary point A and set initial values and inputs
c_0_A = 956.271352
T_0_A = 250.051971
```

```
sim_model.set('_start_c', c_0_A)
sim_model.set('_start_T', T_0_A)
sim_model.set('Tc', 280)
init_res = sim_model.simulate(start_time=0., final_time=150)
```

Next, the optimization problem is compiled and loaded. Here it is important that the compiler option `'state_initial_equations'` is set to True. This option creates parameters for the initial conditions with the name `'_start_`*stateName*`'`, where *stateName* is the name of the state. This is important since the MPC class needs to know the name of the initial condition parameters to be able to change them. The MPC class assumes that the initial condition parameters are named according to this convention, which is why it is important that `'state_initial_equations'` is set to True.

Next, some MPC simulation options are defined. The MPC class requires that sample points and mesh points coincide. This is why we've made the total number of collocation elements, `'n_e'`, dependent on how many collocation elements each sample shall contain, `'n_e_per_sample'`.

```
# 2. Define the optimal control problem and solve it using the MPC class
# Compile and load optimization problem
op = transfer_optimization_problem("CSTR.CSTR_MPC", file_path,
                        compiler_options={"state_initial_equations":True
                            })

# Define MPC options
sample_period = 3                                   # s
horizon = 33                                        # Samples on the horizon
n_e_per_sample = 1                                  # Collocation elements /
     sample
n_e = n_e_per_sample*horizon                        # Total collocation elements
finalTime = 150                                     # s
number_samp_tot = finalTime/sample_period           # Total number of samples to
      do
```

Next, the regular collocation options which are to be used when creating and solving the NLP are defined. A complete list of these options is found in [*JModelica.org user Guide*]. Note that all collocation options are not compatible with the MPC class. Collocation options and their compatibility with the MPC class will be discussed in section 4.2.3

```
# Create blocking factors with quadratic penalty and bound on 'Tc'
bf_list = [1]*horizon
factors = {'Tc': bf_list}
du_quad_pen = {'Tc': 500}
du_bounds = {'Tc': 30}
bf = BlockingFactors(factors, du_bounds, du_quad_pen)

# Set collocation options
opt_opts = op.optimize_options()
opt_opts['n_e'] = n_e
opt_opts['n_cp'] = 2
opt_opts['init_traj'] = init_res
opt_opts['blocking_factors'] = bf
```

To have something to compare the MPC results to we compile and load a new instance of the optimization problem and run an open loop optimization on it. The

open loop optimization shall have an optimization horizon which corresponds to the total running time of the MPC simulation. To achieve this, and get the same resolution in both problems, the number of collocation elements, and in turn the blocking factor list, need to be changed for the open loop problem accordingly.

```
if with_plots:
        # Compile and load a new instance of the op to compare the MPC
            results
        # with an open loop optimization.
        op_open_loop = transfer_optimization_problem(
            "CSTR.CSTR_MPC", file_path,
            compiler_options={"state_initial_equations":True})
        op_open_loop.set('_start_c', float(c_0_A))
        op_open_loop.set('_start_T', float(T_0_A))

        # Copy options from MPC optimization
        open_loop_opts = copy.deepcopy(opt_opts)

        # Change n_e and blocking_factors so op_open_loop gets the same
        # resolution as op
        open_loop_opts['n_e'] = number_samp_tot

        bf_list_ol = [n_e_per_sample]*(number_samp_tot/n_e_per_sample)
        factors_ol = {'Tc': bf_list_ol}
        bf_ol = BlockingFactors(factors_ol, du_bounds, du_quad_pen)
        open_loop_opts['blocking_factors'] = bf_ol
        open_loop_opts['IPOPT_options']['print_level'] = 0
```

The last thing to be defined before creating the MPC object is a constraint violation cost on T. The MPC class will automatically soften hard variable bounds on variables where a constraint violation cost have been specified.

When creating the MPC object we specify the optimization problem, `op`, the collocation options, `opt_opts`, the sample period, `sample_period`, the horizon, `horizon`, the constraint violation costs, `constr_viol_costs`, and a noise seed for the random generator used when emulating noise, `noise_seed`. The first four of these always need to be specified when creating an MPC object, while the last two are optional.

`x_k` is a dictionary containing the new initial value of the states for each sample. Before starting the MPC loop it is assumed that the state is measured and these values are obtained. The MPC loop will be run through a total of `number_samp_tot` times, where `number_samp_tot` is the number of samples on the MPC simulation horizon. Calling `MPC_object.update_state(x_k)` will set the parameters representing the initial value of the states to the values specified in `x_k`. Calling `MPC_object.sample()` will solve the optimization problem and return the optimal input for the first sample in the result, `u_k`. The simulation model is used to simulate the system from the current state, `x_k`, and one sample period forward in time with the optimal input obtained from the optimization, `u_k`. The method `MPC_object.extract_states(sim_res, mean=0, st_dev=0.005)` extracts the state values at the end of the simulation result, `sim_res`, and adds normally distributed noise with the mean value 0 and standard deviation 0.5% of the current state value to each new initial state value.

This method will also save the simulation result presented to it and concatenate it to create a complete patched together result file for the MPC simulation. The user obtains this result in the form of an `MPCAlgResult` object by calling `MPC_object.get_complete_results()`.

```
constr_viol_costs = {'T': 1e6}

# Create the MPC object
MPC_object = MPC(op, opt_opts, sample_period, horizon, constr_viol_costs
    =constr_viol_costs, noise_seed=1)
# Set initial state
x_k = {'_start_c': c_0_A, '_start_T': T_0_A }

# Update the state and optimize number_samp_tot times
for k in range(number_samp_tot):
    # Update the state and compute the optimal input for next sample
        period
    MPC_object.update_state(x_k)
    u_k = MPC_object.sample()

    # Reset the model and set the new initial states before simulating
    # the next sample period with the optimal input u_k
    sim_model.reset()
    sim_model.set(x_k.keys(), x_k.values())
    sim_res = sim_model.simulate(start_time=k*sample_period,
                                 final_time=(k+1)*sample_period,
                                 input=u_k)

    # Extract state at end of sample_period from sim_res and add
        Gaussian
    # noise with mean 0 and standard deviation 0.005*(
        state_current_value)
    x_k = MPC_object.extract_states(sim_res, mean=0, st_dev=0.005)

# Extract variable profiles
complete_result = MPC_object.get_complete_results()
c_res_comp = complete_result['c']
T_res_comp = complete_result['T']
Tc_res_comp = complete_result['Tc']
time_res_comp = complete_result['time']
```

Lastly, the result is visualized with the following script:

```
    # Plot the results
    if with_plots:
        ### 3. Solve the original optimal control problem without MPC
        res = op_open_loop.optimize(options=open_loop_opts)
        c_res = res['c']
        T_res = res['T']
        Tc_res = res['Tc']
        time_res = res['time']

        # Plot
        plt.close('MPC')
        plt.figure('MPC')
        plt.subplot(3, 1, 1)
        plt.plot(time_res_comp, c_res_comp)
        plt.plot(time_res, c_res )
        plt.legend(('MPC with noise', 'Open-loop without noise'))
        plt.grid()
        plt.ylabel('Concentration')

        plt.subplot(3, 1, 2)
```

```
        plt.plot(time_res_comp, T_res_comp)
        plt.plot(time_res, T_res,)
        plt.grid()
        plt.ylabel('Temperature')

        plt.subplot(3, 1, 3)
        plt.step(time_res_comp, Tc_res_comp)
        plt.step(time_res, Tc_res)
        plt.grid()
        plt.ylabel('Cooling temperature')
        plt.xlabel('time')
        plt.show()
```

Output

Running the script will produce two outputs; a plot that visualizes the results and an output on the terminal containing information about the total and average times for the MPC simulation. The plot in Figure 4.1 looks pretty much the same as that

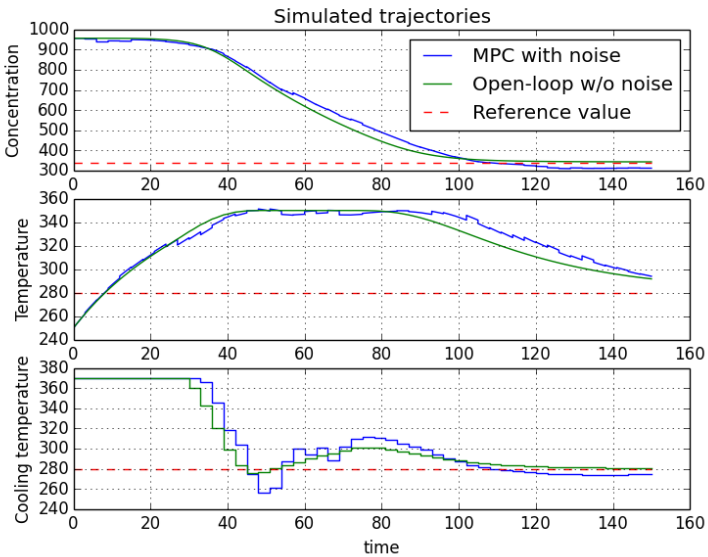

Figure 4.1 Simulated trajectories for the CSTR example using the MPC class.

obtained using the optimization algorithm as done in section 1.2. The blue lines are the patched together MPC simulation result, while the green lines are the corresponding open-loop optimization problem run which thus contains no noise. The dashed red lines are the reference values.

```
Total time for 50 samples (average time in parenthesis).
```

```
Initialization time: 0.15 seconds

Total time: 0.96 seconds              (0.019)
Pre-processing time: 0.05 seconds     (0.001)
Solution time: 0.72 seconds           (0.014)
Post-processing time: 0.20 seconds    (0.004)

Largest total time for one sample (nbr 1): 0.06 seconds
The sample period is 3.00 seconds
```

The thing to take notice of concerning the times is that the initialization time has been separated from the pre-processing time. Included in the initialization time is the time it takes to set up the NLP and create and initialize the solver. Initialization of the solver includes calculating the function derivatives. This is separated from the rest of the times since it is only done once; once the NLP has been set up, we are going to use the same NLP object for all samples.

For all but the first sample pre-processing time includes the time it takes to shift the optimization horizon one sample period forward in time. It also includes the time it takes to obtain the new initial guess for the optimization variables (based on the result from the previous optimization). There are three different methods of obtaining the new initial guess, these will be presented in section 4.3.3. Pre-processing time also includes the time it takes to change the parameter values representing the initial values of the states and setting the initial guess for the variables as inputs to the solver.

The solution time is defined exactly the same as in the optimization algorithm, that is the time it takes for the NLP solver to find a solution to the NLP. The post-processing time includes the time it takes to extract the first optimal input from the result. Notice that, if the user does not specifically ask for it, either by having chosen `initial_guess ='trajectory'` or by calling `get_results_this_samle()` on the MPC object, the MPC class will not create a `LocalDAECollocationAlgResult` object after each optimization. The optimal input is extracted from an internal representation of the result instead. Lastly, we get information on which of the samples that had the largest total time, how long time that was, and the sample period used. The values in parenthesis are the average times for one sample.

4.2 MPC class

In this section we go into more detail about how to use the MPC class. We start off by describing how to create an MPC object; the MPC options available and what is included in the initialization process. We continue with presenting the public methods available in the MPC class and end with a discussion of some collocation options which are to varying degrees incompatible with the MPC class.

Initialization

To create an MPC object there are four things that must be defined and an additional five things that are optional. These are all summarized in the table below.

	Keyword	Default	Description
1	`op`	-	The OptimizationProblem object (op). The op must have been compiled with the compiler option `state_initial_equation` set to True.
2	`options`	-	The collocation options. See 4.2.3.
3	`sample_period`	-	The sample period.
4	`horizon`	-	The number of samples on the prediction horizon.
5	`initial_guess`	`'shift'`	Specifies which method to use when computing the initial guess for the optimization. Can be either: `'trajectory'`, `'shift'` or `'prev'`. See 4.3.3.
6	`create_comp_result`	True	Specifies whether to create the patched together result file for the MPC simulation or not. See 4.3.6.
7	`constr_viol_costs`	{}	A map with the names of variables to soften variable bounds for as keys and their respective constraint violation costs as values. See 4.3.2.
8	`warm_start_options`	{}	A map with the warm start options to use. See 4.3.5.
9	`noise_seed`	None	The noise seed to use on the random generator when emulating noise.

The first thing to happen when initializing an MPC object is that the prediction horizon, in time, is calculated. For an equally spaced mesh (`options['hs']=None`) the prediction horizon is calculated as the sample period times the number of samples on the horizon, that is $H_p = sample\_period \cdot horizon$. How the MPC class calculates the prediction horizon if different element lengths are used is described at the end of this section.

Once the prediction horizon has been calculated, the final time is calculated as the start time plus the prediction horizon. The MPC class then checks if the calculated final time is equal to that defined in the op object. If not, the final time in the op object is set to the calculated final time. This means that the MPC class discards whatever value was defined as the final time in the Optimica file. A warning is printed on the terminal if this happens, telling the user that the final time defined

in Optimica has been changed. The reason for this procedure is to ensure that the sample points will coincide with mesh points.

Next in the initialization process, variables with bounds that are to be softened automatically are softened. This will be explained in more detail in section 4.3.2. but in short the MPC class completes the softening by making some modifications in the op object. Further modifications in the op object are done if any inputs have blocking factor `du_bounds` or `du_quad_pen` applied. An explanation to these modifications and why they are needed will be presented in section 4.3.1. If the user have not specified any blocking factors at all, default blocking factors are applied by the MPC class by changing the blocking factor option in the dictionary containing the collocation options.

Lastly, once all necessary modifications have been made, the op object along with the collocation options are sent to the collocation algorithm and the NLP is created. It should be noticed that once the MPC object has been created, any changes done by the user to the optimization options dictionary will not have any affect on anything. The user is also strongly discouraged from changing anything in the op object after the MPC object has been created. This is because the MPC class continuously goes through the op object to update parameter values, such as the values of the initial condition parameters. If anything in the op structure is changed, or rather if the number of parameters in the op is changed, we will get a dimension mismatch between the vector containing parameter values extracted from the op object and the parameter values input vector the NLP solver was defined with.

Different element lengths. For a mesh with different element lengths the prediction horizon is calculated as the sample period divided by the N first elements in the 'hs' vector, where N is the value at index 0 of the first input with blocking factors applied. Typically, N will be equal to one. That is

$$H_p = \frac{h}{sum(hs[0 : bf[0]])} \tag{4.1}$$

where h is the sample period, hs is the vector specifying the element lengths and bf is the blocking factor list for the first input with blocking factors defined. This is illustrated with an example;

```
blocking_factors = { 'input1': [1,2] }
opt_opts['hs'] = [0.1, 0.3, 0.6]
sample_period = 10
```

The prediction horizon in this case will become

$$H_p = \frac{10}{sum(hs([0 : 1]))} = \frac{10}{0.1} = 100 \tag{4.2}$$

Public methods

In this section we present the public methods available in the MPC class. For each method we first present the Python docstring belonging to the method, and then some further explanations.

`update_state`

```
update_state(self, x_k=None, start_time=None):
"""
Updates the initial condition parameters for the next optimization
to the values in x_k.
Moves the start time of the optimization one sample_period forward,
or to the value specified by start_time.

        Parameters::

        x_k --
        Either a dictionary containing the new values of the initial
        condition parameters or None.
        If None values of the initial condition parameters will be
        extracted automatically from the previous optimization result.
        Default: None

        start_time --
        A float defining the start time of the next optimization.
        If None the start time of the next optimization will be
        calculated as the start time of the last optimization +
        the sample period.
        Defaul: None
"""
```

Calling this method denotes the start of a new sample, meaning this method should always be called first in each MPC-loop. The first thing the method does is that it increases the internal sample number counter with 1 and restarts the timer. When initializing, the internal sample number is defined as 0, which is why `update_state` should be called in the first sample as well.

Next, the method changes the parameters representing the initial values to the values specified in `x_k`. `x_k` may be either a dictionary containing the names of the initial value parameters as keys and their new values as values or None. If `x_k` is None, the new initial values will be extracted from the previous optimization result. The default value of `x_k` is None.

`sample`

```
sample(self):
"""
Extracts parameter values from the op-file, shifts the optimization
     horizon,
redefines the initial guess of the primal variables (for all but the
first sample) and solves the NLP.
Warm start is initiated the second time sample is called.
"""
```

The method starts by recalculating all parameter values and shifting the optimization horizon one sample period forward in time, or as was specified with

start_time in update_state. The method also redefines the initial guess of the optimization variables for the solver (for all samples but the first). For the first sample, the initial guess should have been provided by the user e.g. by the collocation option 'init_traj'. Different ways of redefining the initial guess between MPC samples is presented in 4.3.3.

The second time the method is called, warm start of the solver is initiated. This is done in the second sample since we may get convergence issues if we initiate the warm start in the first sample. More about warm start in 4.3.5.

Next, the method calls the NLP solver to solve the problem and obtains the result. The times for the current sample (pre-proccessing, solution, post-processing and total times) are added to the complete times.

The method returns the optimal inputs of the first sample to the user.

extract_states

```
extract_states(self, sim_res, mean=0, st_dev=0.000):
"""
Extracts the last value of the states from a simulation result object
and adds a noise with mean value and standard deviation as defined.
If 'create_comp_result' is True the method also saves and concatenates
all sim_res to create a complete MPC simulation result file.

Parameters::

        sim_res --
        The simulation result object from which the states are to be
        extracted. If 'create_comp_result' is True, sim_res will be
        added to the complete result.

        mean --
        Mean value of the noise.
        Default: 0

        st_dev --
        Factor to be multiplied with the current value of each state to
        define the stanard deviation of the noise.
        Default: 0.000
"""
```

The method extracts the last values of the states from the simulation result, sim_res, and adds a normally distributed noise with the mean value as specified by mean and standard deviation as specified by st_dev times the state value extracted. If no value of mean and st_dev have been specified, no noise will be added.

If 'create_comp_result' was set to True in the creation of the MPC object this method will also save and concatenate the simulation results provided each time the metod is called to create a complete MPC simulation result object. The complete MPC simulation result object is obtained by calling get_complete_results().

get_results_this_sample

```
get_results_this_sample(self):
"""
```

```
Returns the result object of the last optimization.
(a LocalDAECollocationAlgResult-object).
"""
```

The method returns the `LocalDAECollocationAlgResult` object for the last optimization.

get_complete_results

```
get_complete_results(self):
"""
Creates and returns the patched together resultfile from all
optimizations.
"""
```

The method creates and returns a `MPCAlgResult` object containing the simulation result provided each time `extract_states` was called. That is, the result object returned contains the all the simulation results patched together. The creation of the `MPCAlgResult` object also prints the MPC simulation times on the terminal as shown in section 4.1.2.

set_inittraj

```
set_inittraj(self, init_traj):
"""
Defines the initial trajectories for the next optimization.

Parameters::

    init_traj --
    The result file from which the initial trajectories are to
    be extracted.
"""
```

The method redefines the initial guess for the next optimization by extracting the state values at the collocation points from the trajectories in `init_traj`. Note that the MPC class redefines the initial guess for the next optimization automatically as described in section 4.3.3. This method is only intended to be used if the user created an MPC object without defining an initial trajectory.

set

```
set(self, name, value):
"""
Sets the specified parameters in names to the value in values.

Parameters::

        name --
        List of parameter names whose values are to be changed.
        Type: [string] or string

        value --
        Corresponding new values for the parameters.
        Type: [float] or float
```

```
"""
```

The method sets the parameters in `names` to the corresponding value in `values` in the op object.

`get`

```
get(self, name):
"""
Returns the value of the specified parameter.

Parameters::

        name --
        The name of the parameter whose value is to be returned.
"""
```

The method returns the value of the specified parameter.

`print_solver_stats`

```
print_solver_stats(self):
"""
Prints the sample number, return status, number of iterations and
     solution time for each optimization on the terminal.
"""
```

The method prints the sample number, return status and number of iterations for each optimization on the terminal.

`get_solver_stats`

```
  get_solver_stats(self):
  """
  Returns a dictionary with the solver statistics and times for each
       optimization.
  """
```

The method returns a dictionary with each sample number as keys and the solver statistics and times for each sample as values.

Collocation options

In this section we will discuss some common collocation options and their compatibility with the MPC class. In general the MPC class is completely compatible with the default settings on all options (aside from `n_e`).

To make sure that the sample points coincide with mesh points, `'n_e'` needs to be chosen as a multiple of `horizon`. That is `'n_e'` $= i \cdot$ `horizon`. The MPC class throws an exception if this criterion is not met. We need the sample points to coincide with mesh points because it is assumed that the input is to remain constant through each sample period, as described in section 3.3.4

The collocation options which won't work as desired, or are not compatible at all, with the MPC class are discussed below.

- **Nominal trajectories.** The problem with using nominal trajectories to obtain scaling factors is that the scaling factors are inserted directly into the NLP equations as values and not as parameters. This means that there is no way for the MPC class to shift the scaling factors with the shifting optimization horizon in each sample. In other words, if nominal trajectories are used to obtain scaling factors, each consecutive MPC sample will be scaled exactly the same as the first. For some systems this "incorrect" scaling will just be a technicality and won't actually cause any problems. But in general, a badly scaled problem could cause problems in the solver, which might not be able to find a feasible solution due to the bad scaling.

- **External data.** MPC, or optimization problems in general, support the use of reference trajectories instead of fix reference setpoints. In JModelica.org one would use the option `external_data` to supply a reference trajectory to the optimization algorithm. However, when using this together with the MPC class, the same problem as with nominal trajectories arises. The reference values for each of the collocation points are inserted into the NLP equations as values, and not as paramaters, meaning there is no way to shift them with the shifting optimization horizon.

- **Free element lengths.** Since the MPC class assumes that sample points coincide with mesh points, we can't allow the element lengths to be free, that is `options['hs']`$\neq$`Free`.

4.3 Features

Blocking factors

To achieve a piecewise constant input the MPC class applies blocking factors automatically, have blocking factors not been defined by the user. Default blocking factors are a constant input through each sample on the optimization horizon.

`du_bounds` and `du_quad_pen` are used to limit or penalize the change in input, each time the input changes. The problem with using either of them with MPC is that they are not enabled at the start time of each optimization, only each time the input changes after that. This is not a problem for the first optimization where the first input is allowed to be chosen freely, since we have no value of the input prior to the first MPC optimization. For all other optimizations, though, we have a previous value of the input, the optimal input from the previous optimization. So in order to make `du_bounds` and `du_quad_pen` valid over the total MPC simulation horizon we need to enable them at the beginning of each of the remaining optimizations. The MPC class handles this automatically as follows:

1. A parameter *u*_0 is added to the optimization problem. The parameter keeps track of what the input was in the previous optimization.

2. If `du_bounds` is activated an extra constraint is added to the optimization problem. The extra constraint has has the following form:

$$|u\_0 - u_1| \leq \texttt{du\_bound} \tag{4.3}$$

where u_1 is the first value in the input sequence for u in the result. If `du_quad_pen` is activated an extra term is added to the cost function. The cost function $f(z)$ thus becomes:

$$f(z) + \texttt{du\_quad\_pen} \cdot |u\_0 - u_1|^2 \tag{4.4}$$

where u_1 is the first value in the input sequence for u in the result.

These steps are executed for all inputs which have `du_bounds` or `du_quad_pen` enabled. After each optimization, the value of u_0 is updated to the optimal input obtained.

Softening variable bounds

The MPC class contains a method that automatically softens variable bounds. The variable bounds that are to be softened are defined in the dictionary `constr_viol_costs`, which is an optional input when creating an MPC object. The dictionary shall contain the name of the variables whose bounds are to be softened as keys and the constraint violation weight for those variables as values. The method that softens the bounds creates one slack variable for each variable with a bound to be softened. That means that if a variable has both an upper limit z_U and a lower limit z_L, the same slack variable, z_{slack}, will be used when softening both bounds. The softening is done in four steps:

1. A new input, the slack variable z_{slack}, is added to the optimization problem. The slack variable is bound to be larger than 0 and the nominal value is set to 0.0001 times the nominal value of the base variable. That is,

$$z_{slack} \geq 0 \tag{4.5}$$

$$z_{slack,nominal} = 0.0001 \cdot z_{nominal} \tag{4.6}$$

2. The 1-norm of the slack variable times the constraint violation penalty P_z are added to the cost function. That is, the cost function $f(z)$ is changed to:

$$f(z) + P_z \cdot \int_{t_0}^{t_f} z_{slack}(t)dt \tag{4.7}$$

3. New path constraints for the variable are created and added to the optimization problem:

$$z \leq z_U + z_{slack} \tag{4.8}$$

$$z \geq z_L - z_{slack} \tag{4.9}$$

4. The variable bounds are changed to:

$$z_U \leq +\infty \tag{4.10}$$

$$z_L \geq -\infty \tag{4.11}$$

These four steps are done for all variables that have bounds that are to be softened. If a variable only has either an upper or a lower bound step 3 and 4 are modified accordingly. Ideally, if the initial condition has not violated the constraint, the slack variable should be zero or very close to zero at all times. However, choosing the nominal value of the slack variable has to be done with care. With a nominal value too small, the solver might run into numerical issues if the constraint is suddenly violated. This is why we've made the nominal value of the slack variable proportional to the nominal value of the base variable. The factor of 0.0001 included in the calculation of the slack nominal value was decided through testing.

Defining the next initial guess

There are three different ways to define the initial guess of the primal NLP variables before each optimization.

1. Extracting it from the result object of the last successful optimization.

2. Shifting the values in the result vector the NLP-solver returns after an optimization.

3. Using the previous result vector as it is, without shifting it.

These three methods will be described, and their pros and cons will be discussed, in the next three sections. The last section will discuss how the initial guess of the dual variables are handled. In chapter 5 the different ways of handling the initial guess of the primal and dual variables will be tested.

Extracting from a result object Extracting the initial guess from a result object is done using the same methods the optimization algorithm uses when extracting the initial guess from initial trajectories. Using this method, the values at the collocation points are extracted from the result object trajectories for all collocation elements except those in the last sample. The last collocation element is excluded since the previous optimization result ends one sample period "earlier" than this one. The initial guesses for the last element in each of the variable trajectories will be constantly extrapolated to cover the last sample period.

This way of obtaining the initial guess will always give an accurate initial guess. By accurate we mean that the initial guess corresponds to the previous optimization result exactly in all collocation points except those in the last element. The one drawback with this method is that it is very time consuming.

Shifting the result vector The NLP solver returns the solution of an optimization in one long vector containing the value of each variable at each of the collocation points. The result vector is on the exact same form as the vector corresponding to the initial guess. Looking at the result vector, this method discards all the values included in the first sample period and shifts the rest of the values to cover the voids. That means that all the values corresponding to the second sample period in the result vector will be shifted to the values corresponding to the first sample period in the initial guess vector. This is done for all samples in the result vector. The values of the last sample period in the initial guess vector are all set to the value of the last collocation point from the result vector. These steps are visualized in the following table where we have three collocation elements, two collocation points in each element and each sample period contains one collocation element:

Result vector	$z_{1,1}$	$z_{1,2}$	$z_{2,1}$	$z_{2,2}$	$z_{3,1}$	$z_{3,2}$
Delete values for first sample period	…	…	$z_{2,1}$	$z_{2,2}$	$z_{3,1}$	$z_{3,2}$
Shift the rest of the values	$z_{2,1}$	$z_{2,2}$	$z_{3,1}$	$z_{3,2}$	…	…
Extrapolate over last sample period	$z_{2,1}$	$z_{2,2}$	$z_{3,1}$	$z_{3,2}$	$z_{3,2}$	$z_{3,2}$
New initial guess vector	$z_{2,1}$	$z_{2,2}$	$z_{3,1}$	$z_{3,2}$	$z_{3,2}$	$z_{3,2}$

Inputs with blocking factors are shifted in a similar manner, the first value in the input sequence is discarded and the rest of the values are shifted of step down while the last value is constantly extrapolated:

Result vector	u_1	u_2	u_3
Initial guess vector	u_2	u_3	u_3

This method of shifting the result vector assumes two things:

1. That all collocation elements are of the same length.

2. That the blocking factor lists `bf`, are defined as `bf=[n_e_s]·horizon`, where `n_e_s` is the number of collocation elements per sample period and `horizon` is the number of samples on the prediction horizon.

This means that shifting the result vector this way will yield the exact same initial guess as extracting it from the result object, provided that these two conditions are met. That said, it is still possible to use this method even though one or both of the conditions are not met. The initial guess obtained in that case will not be accurate, but it will still be a good initial guess.

Provided that the two conditions are met, the main advantage of using this method is that it obtains an accurate initial guess in a much faster way than the method extracting the initial guess from a result object. Even if the conditions are not met, the time saved by using this method may still be greater than the extra time needed by the solver due to a less accurate initial guess.

Using the result vector as it is This method sets the new initial guess to the result from the previous optimization directly, without shifting it. Using this method will yield the least accurate initial guess, since all values will be offset by one sample period in time, but it is the least time consuming method of the three mentioned.

Dual variables The result of the dual variables of each optimization are included in the result object, but only on the vector form the NLP-solver returns them on and not as trajectories. This means that it is not possible to extract an initial guess for them in the same way that an initial guess for the primal variables are extracted using the trajectories. In other words, we are not able to obtain an accurate initial guess for the dual variables using the extraction method. The MPC class does not support shifting of the dual variables either, so the only option left is to use the previous result of the dual variables as it is. This means that the initial guess for the dual variables will be offset by one sample period in time. However, if an initial guess for the dual variables is not provided to the NLP-solver, the solver has it's own method of generating an initial guess for them. So we are left with the choice of either providing an initial guess for the dual variables that is offset by one sample period in time, or to not provide an initial guess at all and let the NLP-solver generate it's own initial guess. This will be tested in 5.2.

Unsuccessful optimization

It may happen that the solver terminates without finding a feasible solution to the given problem. In that case the method `sample()` returns the next input in the input sequence of the previous optimization (which was successful). If the next optimization after that is unsuccessful as well, the third input in the input sequence in the last successful optimization is returned. This way of returning optimal inputs from the last successful optimization continues until the solver finds a feasible solution again, or until there are no more values in the last successful optimization to return. In the last case the MPC class throws an exception. If it is the first optimization that is unsuccessful, the MPC class throws an exception as well.

This is how the MPC class handles an unsuccessful optimization. However, it is straightforward to detect if an optimization was successful or not, by looking at the string named `status` in the MPC object, so it is possible for the user to create a custom fallback method instead.

The MPC class counts the return statuses `'Solve_Succeeded'` and `'Solved_To_Acceptable_Level'` as successful optimizations for IPOPT, and the return statuses `'OptimalSolution'`, `'LowPassFilterOptimal'` and `'AcceptableSolution'` as successful optimizations for WORHP. This default setting may be changed by defining the return statuses that are to count as successful in the list called `successful_optimization` in the MPC class.

Warm start of the NLP solver

Warm start of the NLP solver means that we disable the feature where the solver computes it's own initial guess for the dual variables. This means that once warm start is activated we need to provide an initial guess of the dual variables to the solver. The warm start is automatically activated in the beginning of the second sample, since we do not have any values for the dual variables before the first sample. Activating the warm start without providing an initial guess for the dual variables makes it a lot harder for the solver to find an optimal solution.

Default warm start options applied by the MPC class for IPOPT are

```
warm_start_init_point = 'yes'
mu_init = 1e-3.
print_level = 0
```

Default warm start options applied by the MPC class for WORHP are

```
InitialLMest = True
NLPprint = 0
```

The user may change the warm start options to use by providing a dictionary with the warm start options to use under the keyword `'warm_start_options'` when creating the MPC object.

Presenting the result

The MPC class presents the result of an MPC simulation in the new result object type `MPCAlgResult`. The class `MPCAlgResult` is a `JMResultBase` object with the additional attributes `times`, which is a dictionary containing the total times for the MPC simulation, `nbr_samp`, which is the number of samples completed in the MPC simulation and `sample_period`, which is the sample period specified.

The complete result is created by patching together all the simulation results which have been provided to the method `extract_states(sim_res)`. Notice that this means that the user must have simulated the system with the optimal input obtained as described in section 4.1.1. The `MPCAlgResult` object is obtained by calling `get_complete_results()` on the MPC object. Notice that `'create_comp_res'` must be True to be able to call this method. After a result object is created, an output will be printed on the terminal. The output was described in section 4.1.2.

4.4 Alterations to the collocation algorithm

The MPC class is built upon the fact that we use the same NLP for each of the consecutive optimizations. Reusing the NLP, and being able to change the initial conditions prior to each optimization, was made possible due to the fact that we

made some modifications concerning the creation of the NLP in the optimization algorithm.

Instead of substituting the parameters for their values in the creation of the NLP, we modified the optimization algorithm to create symbolic representations for them using CasADi instead. In the creation of the solver we were then able to define the parameters as NLP parameters in the solver. NLP parameter have the useful property that they are changeable between optimizations.

4.5 Unsupported features and general restrictions

In addition to the collocation options presented in section 4.2.3, which are to varying degrees incompatible with the MPC class, there are a couple of other restrictions and overall behaviour of the MPC class which the user should be aware of.

- **Time dependent dynamics.** For the MPC class to be able to shift the prediction horizon correctly the dynamics of the system must be time independent. That is, the model equations can have time dependent variables but may not be dependent on time directly e.g. $y(t) = x(t) + t$ is not allowed. For MPC applications it is uncommon to have time dependent dynamics, so this is not that serious a problem.
- **Minimum time problems.** An optimization problem which is formulated as a minimum time problem is not supported by the MPC class. This is due to the fact that for minimum time problems, the mesh points of the grid are not defined until an optimal solution has been found, and thus the assumption that mesh points shall coincide with sample points will not be upheld. Also, since the structure of the NLP is the same in all optimizations the length of the collocation elements will decrease with each optimization. As time elapses, each sample period will contain an increasing amount of collocation elements, but the returned input will still only be that of the first collocation element.
- **Returned input.** The optimal input returned when sample is called will always be a function outputting a constant value equal to that in the first collocation point in the mesh. For inputs where no blocking factors have been applied, this means that the optimal input returned will not be equal to the optimal input calculated, since the calculated optimal input is not constant over the sample period.
- **Shifting the dual variables.** The MPC class will set the initial guess of the dual variables to the result obtained for them from the previous optimization, without shifting them. This means that the initial guess of the dual variables defined will always be one sample period offset in time, compared to what the accurate initial guess would have been.

5

Results

To test the performance of the MPC class four different MPC simulation tests will be run. The first three tests aim to show how different settings affect the performance of the MPC class, while the last test is a benchmark which aims to show how much time can be saved by using the MPC class for an MPC simulation compared to not using the MPC class.

5.1 MPC simulation models

To show how the model size affects the performance all MPC simulation tests will be run on both a small-scale problem as well as a larger problem. The optimization problems for both of them will be described in the subsections below. In each of the test sections, a table which summarizes the results will be shown. The tables present the following information:

- **Opt$_{\text{fail}}$**. The sample number of the optimizations which were unsuccessful. For IPOPT it is assumed that the return statuses 'Solve_Succeeded' and 'Solved_To_Accepteble_Level' denote a successful optimization. All other return statuses are counted as unsuccessful.
- **Iterations**. The average number of iterations for one sample and the highest number of iterations for one sample, denoted by avg. and max. respectively.
- **T$_{\text{pre}}$**. The average pre-processing time for one sample.
- **T$_{\text{sol}}$**. The average solution time for one sample and the longest solution time for one sample, denoted by avg. and max. respectively.
- **T$_{\text{post}}$**. The average post-processing time for one sample.
- **T$_{\text{tot}}$**. The average total time for one sample and the longest total time for one sample, denoted by avg. and max. respectively.

Since the number of iterations, the solution time and the total time may fluctuate a lot from one sample to the next, we will show both the average and the largest value for these. In addition to the tables, some plots to describe the simulations in more detail will be presented.

All tests are run with version 1.15+ of JModelica.org together with version 3.11.9 of Ipopt using the MA27 linear solver.

Combined-cycle power plant

The model for the combined-cycle power plant, henceforth referred to as the *CCPP*, was created by F.Casella and describes a combined-cycle power plant during start-up [Casella et al., 2011].

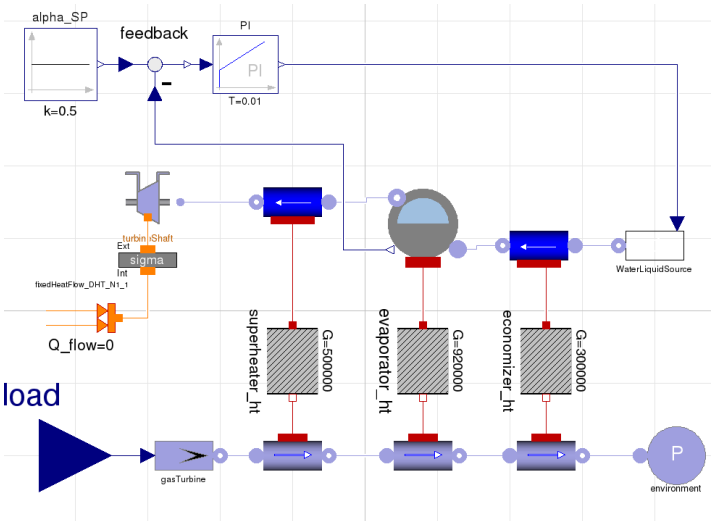

Figure 5.1 Modelica object diagram of the CCPP model.

Starting from the bottom left in Figure 5.1 the input $plant.load$ generates a flow of exhaust gas with a prescribed temperature from the gas turbine. The exhaust gas enter the hot side of a heat exchanger which heats up the water in the evaporator. The feedwater source is controlled by a PI controller to keep the liquid/vapor level in the evaporator constant. The superheated steam enters the steam turbine.

The aim of the MPC controller is to take the system from a shut down state to full capacity. Full capacity is reached once the evaporator pressure, $plant.p$, and the input load, $plant.load$, has reached their reference values of

$$\begin{aligned} plant.p_{ref} &= 8.35\text{MPa}, \\ plant.load_{ref} &= 1. \end{aligned}$$

During the start-up we have an upper bound on the steam turbine stress

$$plant.sigma \leq 260 \text{ MPa}.$$

The MPC class will soften this bound automatically as discussed in section 4.3.2. We are going to extend the model with an integrator at the input by connecting the input *plant.load* with a new state variable u and thus having $\dot{u}$ as the input in the optimization problem. This yields an input which is piecewise linear, rather than piecewise constant. Variable bounds on u and $\dot{u}$ are

$$\begin{aligned} 0 \leq u \leq 1, \\ 0 \leq \dot{u} \leq 0.1/60. \end{aligned}$$

The MPC and collocation options used in all performance tests for the CCPP system are defined below:

```
MPC options
sample_period = 100
horizon = 10
factors = {'du': [1]*10}
bf = BlockingFactors(factors)
op_opts = {'n_e': 10, 'blocking_factors': bf}
constr_viol_costs = {'plant.sigma': 1e5}

nbr_samp_tot = 40
st_dev = 0.004
```

All options not mentioned are at default value or as specified in each of the performance test settings. To emulate noise a normally distributed disturbance, with the mean 0 and the standard deviation as specified by `st_dev` times the current state value, will be added to all the states except for the extra state u.

With the extension of u as a state, and the extra input *sigma_slack* which the MPC class will add to the problem when softening the bound on *plant.sigma*, the resulting optimization problem has 10 states, 123 algebraic variables and 2 inputs. After the transcription, the resulting NLP has 4564 optimization variables.

Continuously stirred tank reactor

Hicks-Ray continuously stirred tank reactor, henceforth referred to as the *CSTR*, was presented in the introductory example in chapter 1. The system has two states, the concentration c and the temperature T. The input to the system is the temperature of the cooling flow Tc. The chemical reaction in the tank is exothermic and temperature dependent. The aim of the MPC controller is to take the system from the initial steady state

$$c_{init} \approx 956,$$
$$T_{init} \approx 250°\text{C},$$

to the reference steady state where

$$c_{ref} \approx 339,$$
$$T_{ref} \approx 280°\text{C},$$
$$Tc_{ref} = 280°\text{C}.$$

The variable bounds in the system are

$$T \leq 350°\text{C}$$
$$230°\text{C} \leq Tc \leq 370°\text{C}.$$

Blocking factors are also applied to the system as defined in the options below. The MPC and collocation options used in all simulations for the CSTR system are defined in the section below:

```
MPC options
sample_period = 3
horizon = 33
factors = {'Tc': [1]*33}
du_quad_pen = {'Tc': 500}
du_bounds = {'Tc': 30}
bf = BlockingFactors(factors, du_bounds, du_quad_pen)
op_opts = {'n_e': 33, 'n_cp': 2, 'blocking_factors': bf}
constr_viol_costs = {'T': 1e6}

nbr_samp_tot = 50
st_dev = 0.005
```

All options not mentioned are at default value or as specified in each of the performance test sections. To emulate noise a normally distributed disturbance, with the mean 0 and the standard deviation specified by `st_dev` times the current state value, will be added to both of the states. Once the MPC class has softened the variable bounds enforced on the temperature T the resulting optimization problem has 2 states, 0 algebraic variables and 2 inputs. After the transcription, the resulting NLP has a total of 432 optimization variables.

5.2 Warm start of IPOPT

In this section we will test how different settings on the warm start options available in IPOPT affects the solution time, the number of iterations and the number of

unsuccessful optimizations for an MPC simulation. The MPC simulations for both the CCPP and the CSTR are run with the additional settings:

```
initial_guess = 'shift'
noise_seed = 1
```

Activating warm start means that IPOPT won't compute an initial guess for the dual variables itself. An initial guess for the dual variables shall in that case be provided by the user. Warm start is activated by setting the solver option `'warm_start_init_point' = 'yes'`.

In interior point methods a barrier parameter μ is used to relax constraints. When warm starting it is also of interest to test different initial values of this barrier parameter `mu_init`. A too large value of `mu_init` might spoil a good initial guess of the optimization variables, while a too small value might make the optimization problem harder to solve. We will therefore test 4 different values of `mu_init`, corresponding to simulation settings 2 through 4. Simulation settings 1 corresponds to using the default values of the warm start options.

1. `warm_start_init_point = 'no'`
   `mu_init = 1e-1`
2. `warm_start_init_point = 'yes'`
   `mu_init = 1e-1`
3. `warm_start_init_point = 'yes'`
   `mu_init = 1e-2`
4. `warm_start_init_point = 'yes'`
   `mu_init = 1e-3`
5. `warm_start_init_point = 'yes'`
   `mu_init = 1e-4`

The results of the MPC simulations for both the CCPP and the CSTR are presented below, where the notation CCPP 1 corresponds to the CCPP system simulated with simulation settings 1 etc.

From Table 5.1 we see that CCPP 1 and CCPP 2 fail to find optimal solutions for 3 and 4 out of the 40 samples respectively, while simulations 3 through 5 find optimal solutions for all samples. Comparing CCPP 3 through 5 it can be seen that CCPP 4 has both the lowest average number of iterations as well as the fastest average solution time. The plots in Figure 5.2 show the number of iterations and solution time for each sample. From them we see that all simulations show roughly the same behaviour. Something seems to have happened at sample number 7 since a clear peak in both number of iterations and solution time is visible there for all simulations. From sample number 14 to 24 there are also some large fluctuations

Table 5.1 Results for the CCPP simulations in the warm start test.

	Opt_{fail} [k]	Iterations [nbr.]		T_{sol} [s]	
		avg.	max.	avg.	max
CCPP 1	7,22,24	41	188	0.83	7.15
CCPP 2	7,8,19,24	39	156	0.75	4.94
CCPP 3	-	41	251	0.69	4.98
CCPP 4	-	37	130	0.54	3.27
CCPP 5	-	43	206	0.81	6.15

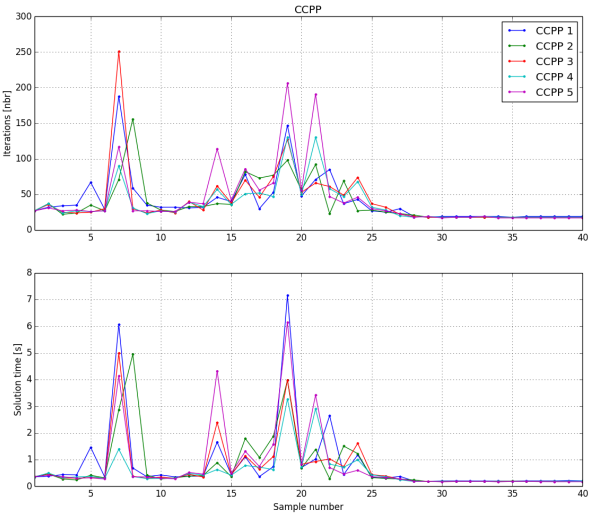

Figure 5.2 The number of iterations and the solution time for each of the samples for the CCPP simulations in the warm start test.

visible for all simulations. This might be due to the constantly extrapolated values for the last sample periods in the prediction horizon becoming a worse initial guess since the variables increase/decrease more rapidly then before. Once we've reached full load, which is around sample number 28 the number of iterations and solution times remain at a low value for all simulations.

A closer look at the CCPP system simulated with the optimal inputs returned from each of the simulations show that all simulations have returned the same optimal inputs for most samples, see Figure 5.3. The deviations visible on the plot are around sample numbers 7, 19 and 24, which are the samples where simulation 1

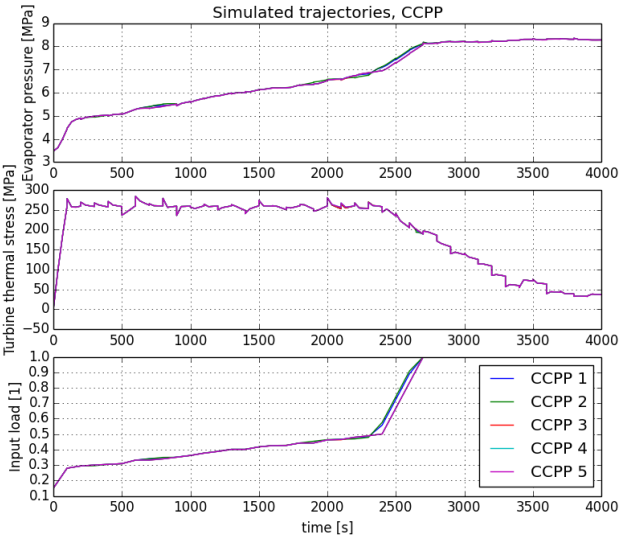

Figure 5.3 The CCPP system simulated with the optimal inputs obtained in each sample in the warm start test.

and 2 failed to find optimal solutions, and thus the MPC class returned the second optimal input from the previous optimization result.

Table 5.2 Results for the CSTR simulations in the warm start test.

	Opt_{fail} [k]	Iterations [nbr.]		T_{sol} [s]	
		avg.	max.	avg.	max
CSTR 1	-	10.9	34	0.015	0.052
CSTR 2	-	10.6	34	0.015	0.053
CSTR 3	-	9.5	34	0.014	0.052
CSTR 4	-	9.2	34	0.013	0.052
CSTR 5	-	8.8	34	0.012	0.057

From Table 5.2 we see that optimal solutions were found for all samples in each of the CSTR simulations. For all simulations it was the first optimization which required the largest number of iterations and had the highest solution time. Comparing the simulations to each other we see that the a decreasing value of `mu_init` decreases both the average number of iterations and the solution time where CSTR 5 shows the best results. The plots in Figure 5.4 show this as well. The CSTR 1 and 2 results are clearly higher than the CSTR 4 and 5 results in almost all of the sam-

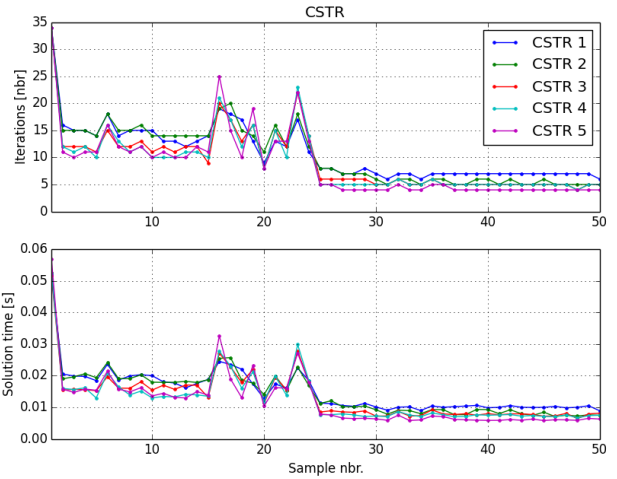

Figure 5.4 The number of iterations and the solution time for each of the samples for the CSTR simulations in the warm start test.

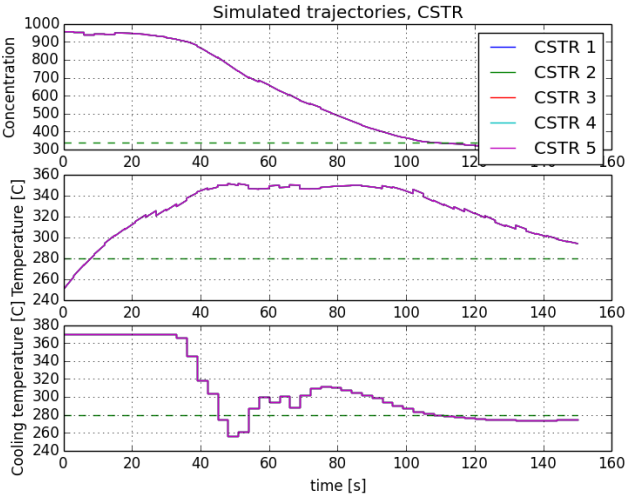

Figure 5.5 The CSTR system simulated with the optimal inputs obtained in each sample in the warm start test.

ples, apart from the region around sample number 20 where the results fluctuates a lot. Taking a closer look at Figure 5.5 we see that all simulated trajectories for the CSTR system are identical, meaning that the same optimal inputs were obtained in all the simulations.

Based on the CCPP and the CSTR results we will set the default warm start options of the MPC class to those corresponding to setting 4, that is:

```
warm_start_init_point = 'yes'
mu_init = 1e-3
```

Using the keyword `'warm_start_options'` when creating the MPC object, allows the user to choose different warm start settings.

5.3 The next initial guess

In this section we will test the 3 different methods of obtaining the next initial guess of the primal variables for the solver. We will look specifically on how the different average times for one sample (total, pre-processing, solution and post-processing) change depending on which method we choose. The MPC simulations for the CCPP and the CSTR are all run with the additional settings;

```
warm_start_options['warm_start_init_point']= 'yes'
warm_start_options['mu_init'] = 1e-3
noise_seed = 1
```

Both the CCPP and the CSTR are simulated with the following different settings:

1. `initial_guess = 'trajectory'`
2. `initial_guess = 'shift'`
3. `initial_guess = 'prev'`

Setting 1 corresponds to extracting the new initial guess for the primal variables from the variable trajectories of a result object while setting 2 corresponds to using the shift method implemented in the MPC class to shift the result vector obtained from the NLP solver one sample period forward in time. Setting 3 corresponds to using the result vector obtained from the NLP solver without compensating for the time offset. The result of the MPC simulations for both the CCPP and the CSTR are presented below.

From Table 5.3 we see that the average pre-processing time for the CCPP system has been decreased by almost 85% from CCPP 1 to CCPP 2 and 3. We also see that there is no difference in average pre-processing time between CCPP 2 and 3. From this we draw the conclusion that compared to all other steps included in the

Table 5.3 Results for the CCPP simulations in the next initial guess test.

	Opt_{fail} (k)	T_{pre} (s)	T_{sol} (s)		T_{post} (s)	T_{tot} (s)	
		avg.	avg.	max.	avg.	avg.	max
CCPP 1	14	0.31	0.74	6.12	0.07	1.12	6.5
CCPP 2	-	0.05	0.54	3.34	0.01	0.60	3.5
CCPP 3	7,8,19	0.05	0.60	6.30	0.01	0.66	6.4

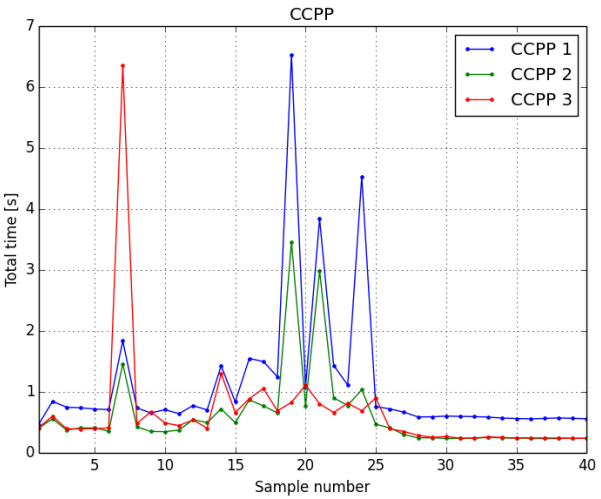

Figure 5.6 Total time for each of the samples in the CCPP simulations in the next initial guess test.

pre-processing time (such as shifting the time horizon), the time it takes to shift the result vector is negligible. Having shifted the result, however, does affect the number of successful optimizations in the simulations. The worse initial guess provided to the solver in CCPP 3 results in unsuccessful optimizations in 3 out of the 40 samples run. The average post-processing time is longer in CCPP 1 compared to the other two due to the fact that a `LocalDAECollocationAlgResult` object, from which the next initial guess shall be extracted, needs to be created after each optimization. This result object is not created in CCPP 2 and 3, unless the user asks for it. The average total time for one sample in this system is decreased by almost a factor 2 by using the shift operation compared to using the extraction method. This is visible in Figure 5.6 as well, where we see that the total time of each sample in CCPP 1 is clearly higher than the total time in CCPP 2 and 3 for almost all samples. Looking at Figure 5.7, where the system was simulated with the optimal inputs obtained in

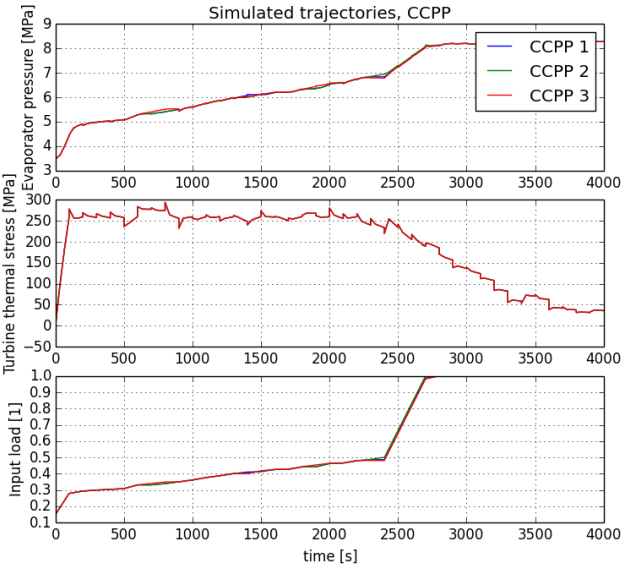

Figure 5.7 The CCPP system simulated with the optimal inputs obtained in each sample in the next initial guess test.

the simulations we see some minor deviations, mainly around the samples where an optimization was unsuccessful.

Another thing to take notice of here is that according to the theory, we should obtain the same initial guess by using the shift operation as by extracting an initial guess from a result object. An identical initial guess as well as identical initial conditions should yield the same solution by IPOPT. However, the results of this test contradicts that, since an optimization fails in CCPP 1 and doesn't in CCPP 2. So either the initial guess obtained by using the shift operation is not identical to using the extraction method, or IPOPT does not yield the exact same solution to identical conditions. A closer investigation of this has been made but no firm conclusion as to why this behaviour occurs has been drawn. It could be that there are some rounding errors involved meaning that the initial guesses are only identical down to a certain number of decimals.

The CSTR system shows the same overall behaviour as the CCPP system does in this test, see Table 5.4, Figure 5.8 and 5.9. That is, the pre-processing time is decreased substantially by using either the shift operation or not shifting the result at all, compared to extracting a new initial guess from a result object. The CSTR system, however, is less affected by the slightly worse initial guess obtained by not compensating for the time offset, as in CSTR 3, than the CCPP was. Feasible

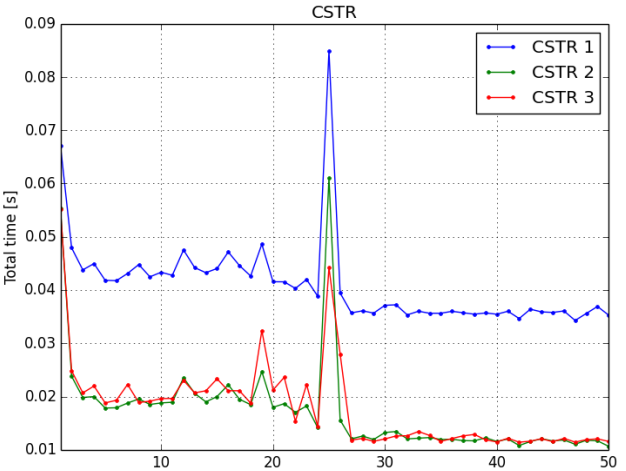

Figure 5.8 Total time for each of the samples in the CSTR simulations in the next initial guess test.

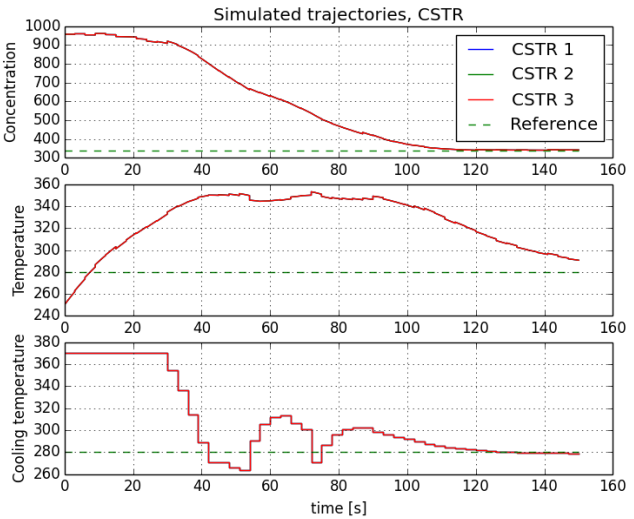

Figure 5.9 The CSTR system simulated with the optimal inputs obtained in each sample in the next initial guess test.

Table 5.4 Results for the CSTR simulations in the next initial guess test.

	Opt_{fail} (k)	T_{pre} (s)	T_{sol} (s)		T_{post} (s)	T_{tot} (s)	
		avg.	avg.	max.	avg.	avg.	max
CSTR 1	-	0.016	0.013	0.053	0.012	0.041	0.08
CSTR 2	-	0.001	0.013	0.051	0.004	0.017	0.06
CSTR 3	-	0.001	0.014	0.051	0.004	0.018	0.06

solutions are found for all samples in all three simulations. The total average total time has decreased by a factor 2 in this system as well.

5.4 IPOPT/WORHP

In this section we will test how the choice of NLP solver affects the performance of an MPC simulation. We will look specifically on the number of unsuccessful optimizations as well as the average solution time and average total times for one sample. Both the CCPP ant the CSTR are simulated with the following settings:

1. `op_opt['solver'] = 'IPOPT'`
2. `op_opts['solver'] = 'WORHP'`

When using WORHP on the CCPP the solver is unable to find a feasible solution to the first sample, meaning the MPC class throws an exception and no results are obtained. If we help the solver along by giving it the correct solution (as computed by IPOPT) as initial guess, WORHP does manage to find a feasible solution to the first sample. However, all optimizations after that fail, so once we are out of next optimal inputs in the first result file, the MPC class will throw an exception and the simulation will terminate. No results for the CCPP are thus presented here.

WORHP is unable to find a feasible solution to the first sample of the CSTR as well, but providing the correct solution as initial guess solves this problem, and the solver is able to find feasible solutions to all but one sample after that.

Table 5.5 Results for the CSTR simulations in the IPOPT/WORHP test.

	Opt_{fail} (k)	T_{sol} (s)		T_{tot} (s)	
		avg.	max.	avg.	max.
IPOPT	-	0.014	0.053	0.019	0.06
WORHP	30	0.223	2.318	0.228	2.32

Looking at the average solution times in Table 5.5 we see that for this system IPOPT is clearly the faster solver. Although, from the plot in Figure 5.10, we see

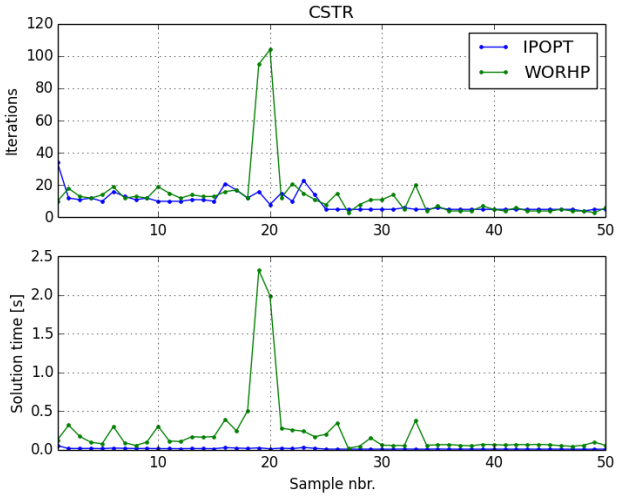

Figure 5.10 The number of iterations and the solution time for each sample in the CSTR simulations in the IPOPT/WORHP test.

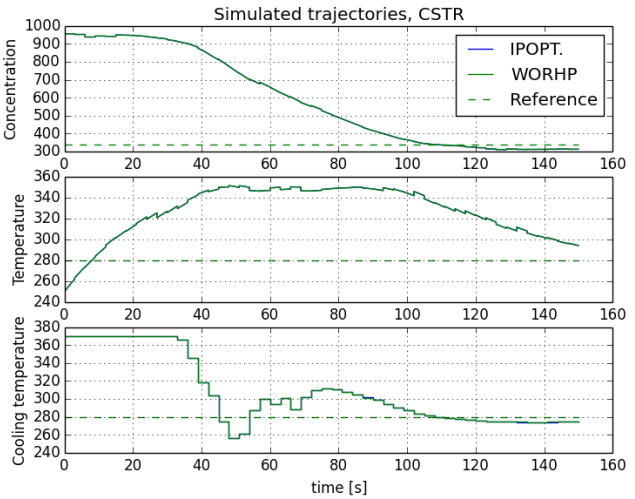

Figure 5.11 The system simulated with the optimal inputs obtained in each sample for the CSTR simulations in the IPOPT/WORHP test.

that the solution time average for WORHP has been greatly increased due to two outliers (sample 19 and 20). From the solution time plot it is clear that IPOPT is the faster solver anyway. Looking at the simulated trajectories in Figure 5.11 we see that the optimal inputs obtained are very close to the same in all samples.

In WORHPs defence it should be noted that the problem formulation of the CCPP was created to be solved with an IP-method, like the one used in IPOPT. So the problem is formulated in a way that benefits IPOPT, rather than WORHP, which first uses an SQL method and then an IP method to solve the NLP. Also, the warm start feature interface between CasADi and WORHP is not as well developed as CasADis warm start interface to IPOPT. With some more time spent on this test, we might have been able to get WORHP to run successfully on the CCPP system as well.

5.5 Benchmark

In this section we will compare the results of an MPC simulation using the MPC class to an MPC simulation without using the MPC class. We will look specifically on the number of successful optimization as well as the different average times for one sample (pre-processing, solution, post-processing and total). The MPC simulation for the CCPP and the CSTR using the MPC class are run with the following options;

```
initial_guess = "shift"
warm_start_options['warm_start_init_point']= 'yes'
warm_start_options['mu_init'] = 1e-3
noise_seed = 1
```

The variable bounds are softened manually for the MPC simulation running without the MPC class. The manual softening is done in exactly the same way as the MPC class does it. For the CSTR, which have blocking factor `du_bounds` and `du_quad_pen` applied, modifications for these are also done manually (revisit section 4.3.1. for a recap of why this is needed). The resulting MPC problems shall thus be identical in the MPC simulations run with and without the MPC class. The only difference between the problems is that warm start of IPOPT is used in the simulation running with the MPC class, while default values of the warm start options are used in the simulation running MPC class.

From the data in Table 5.6 it can be seen that using the MPC class compared to not using it significantly decreases the total average time. This is also clearly illustrated in the total time plots in Figure 5.12 and 5.13. Looking closer at the average times we can conclude that the majority of the total time saved is in the pre-processing step. There is a slight difference in the average solution time, which might be due to the warm start options used in the simulation running with the MPC class. The post-processing time is also slightly decreased due to the fact that

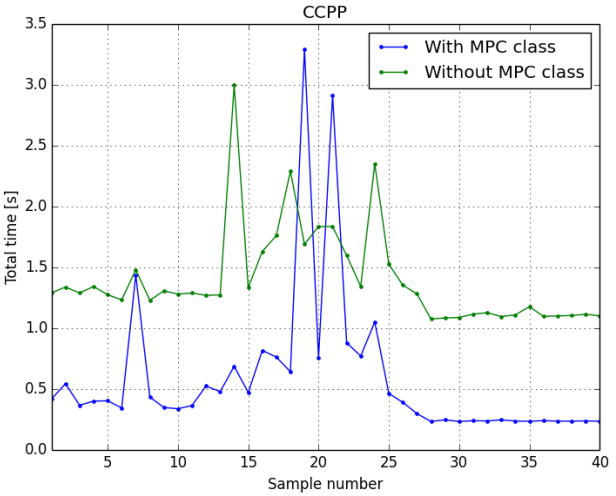

Figure 5.12 Total computation time for each sample in the CCPP benchmark.

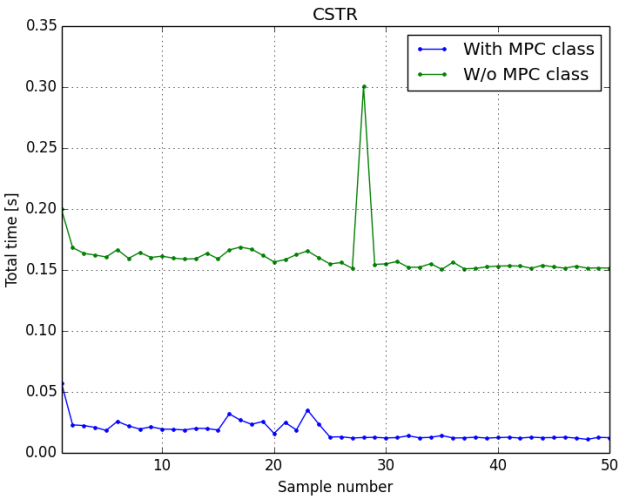

Figure 5.13 Total computation time for each sample in the CSTR benchmark.

Table 5.6 Results for the benchmark for both the CSTR and CCPP systems.

		Opt_{fail} (k)	T_{pre} (s), avg.	T_{sol} (s), avg.	T_{post} (s), avg.	T_{tot} (s), avg.
CCPP	With MPC class	-	0.042	0.527	0.015	0.584
	W/o MPC class	24	0.848	0.490	0.057	1.395
CSTR	With MPC class	-	0.001	0.013	0.004	0.018
	W/o MPC class	-	0.137	0.015	0.008	0.161

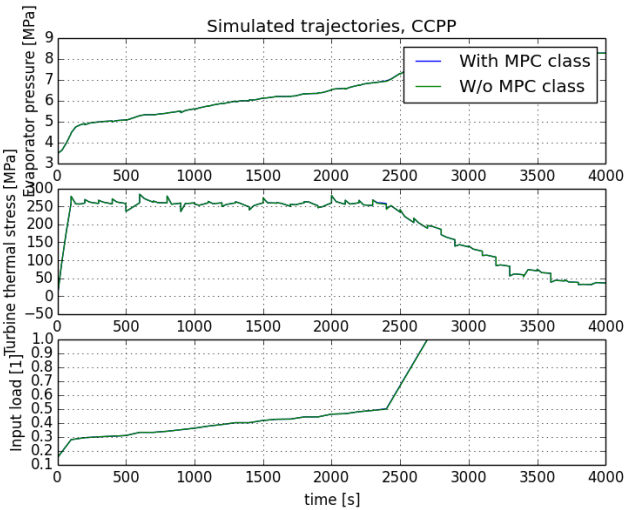

Figure 5.14 The CCPP system simulated with the optimal inputs obtained in both the MPC simulation running with and the MPC simulation running without the MPC class.

the MPC class doesn't create a result object after each optimization. Taking a closer look at the simulated trajectories for both the CCPP in Figure 5.14 and the CSTR in Figure 5.15, we see no difference in the result by using the MPC class compared to not using it. From Table 5.6 however, we do know that the CCPP simulation done without the MPC class failed to find an optimal solution in one of the samples but it is not discernible in the simulated trajectories.

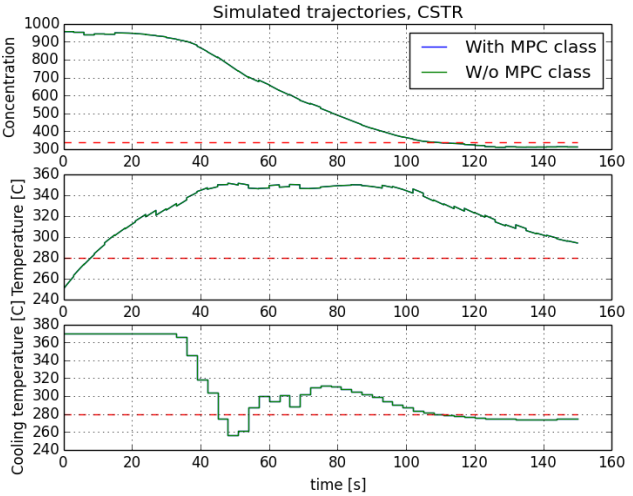

Figure 5.15 The CSTR system simulated with the optimal inputs obtained in both the MPC simulation running with and the MPC simulation running without the MPC class.

6

Concluding remarks

6.1 Conclusions

In this thesis we have implemented an MPC framework in JModelica.org which has shortened the average total computation time for one sample significantly. For the small benchmark problem, the CSTR, the average total time was decreased from $0.154s$ to $0.017s$ by using the MPC framework compared to not using it. Those numbers correspond to a relative decrease of almost 90%. For the large benchmark problem, the CCPP, the average total time for one sample was decreased from $1.395s$ to $0.584s$, a relative decrease of almost 60%.

From both benchmarks we can also conclude that the optimal inputs obtained were the same, within the tolerances used, and thus yielded practically the same simulated trajectories in both cases. Reasons as to why the inputs obtained differed beyond the tolerances include:

- The warm start options chosen on the solver.
- The initial guesses being slightly different (due to a slight difference in the optimal input obtained in the previous sample).

By applying warm start of the solver, we provide an initial guess for the dual variables, while otherwise the solver computes an initial guess for the dual variables itself. The value of other warm start options, such as `'mu_init'`, also affects the solution obtained from the solver. A slight difference in the solution also leads to a slight difference in the initial guess for the next sample, which leads to a slightly different result etc.

Lastly we can summarize the pros and cons of using the MPC class compared to not using it for an MPC simulation as the following. The MPC class

+ is significantly faster.
+ obtains practically the same solution as not using the MPC class does.
+ handles next initial guesses and unsuccessful optimizations internally.

+ can create a complete patched together result file internally.
– has restrictions regarding some collocation options, specifically n_e.
– cannot handle nominal trajectories or external data.
– has restrictions regarding the formulation of the optimization problem.

6.2 Further work

The most important feature which is yet to be supported by the MPC framework is the use of nominal trajectories. Doing this would require quite the implementation effort, though. The collocation algorithm would have to be modified, so that scaling factors are included in the NLP equations as parameters, making it possible to change them between samples. This would increase the number of parameters in the system with the number of optimization variables, since optimization variables would all get a scaling factor parameter each. It would also require a scaling factor shift operation, similar to that implemented for the initial guess of the primal variables, to be implemented in the MPC class. Since this problem was not found until very late into this thesis, support for nominal trajectories was not implemented. The same reasoning applies to the use of external data, which could be used for supplying reference trajectories for variables, rather than reference set points.

To save more time in the solution step, a shift operation for the dual variables could also be implemented. The dual variables however, do not have as clear a structure as the primal variables have i.e. it is not as clear exactly how the vector containing the dual variables should be shifted.

Another feature which could be expanded is the automatic softening of variable bounds. An extension of this would be to let the user choose between different norms to apply to the cost function, as well as choose the number of slack variables to be added to the optimization problem more freely. The feature could also be extended to automatically soften constraints as well, rather than just variable bounds.

In this thesis we've not paid the resulting value of the cost function any mind. To some users the resulting value of the cost function for the complete MPC simulaiton might be of interest, and a function from which it could be obtained would be useful. The value of the cost function is obtainable from the NLP solver for each optimization seperatley. However, the value obtained is the value of the modified cost function, i.e. it includes the terms added to soften constraints and penalize the input. The value obtained is also the resulting value of the cost function over the *entire* optimization horizon. There is no way to choose a specific interval of the optimization horizon and obtaining the value of the cost function in that interval, which would have been useful for creating the patched together result of the cost function.

Bibliography

Å kesson, J. "Optimica—an extension of modelica supporting dynamic optimization". In: *6th International Modelica Conference 2008*. Bielefeld, Germany.

Allgöwer, F., R. Findeisen, and Z. K. Nagy (2004). "Nonlinear model predictive control: from theory to application". *J. Chin. Inst. Chem. Engrs* **35**:3, pp. 299–315.

Berntorp, K. and F. Magnusson (2015). "Hierarchical predictive control for ground-vehicle manuevering". In: *2015 American Control Conference*. Chicago, IL.

Biegler, L. (2010). *Nonlinear Programming: Concepts, Algorithms, and Applications to Chemical Processes*. MOS-SIAM Series on Optimization. Society for Industrial and Applied Mathematics (SIAM, 3600 Market Street, Floor 6, Philadelphia, PA 19104).

Büskens, C. and D. Wassel (2013). "The ESA NLP solver WORHP". In: *Modeling and Optimization in Space Engineering*. Springer, pp. 85–110.

Camacho, E. F. and C. B. Alba (2013). *Model predictive control*. Springer.

Casella, F., F. Donida, and J. Åkesson (2011). "Object-oriented modeling and optimal control: a case study in power plant start-up". In: *18th IFAC World Congress*. Milano, Italy.

Cavey, M. V., R. De Coninck, and L. Helsen (2014). "Setting up a framework for model predictive control with moving horizon state estimation using jmodelica". In: *10th International Modelica Conference 2014*. Lund, Sweden.

Grüne, L. and J. Pannek (2011). *Nonlinear Model Predictive Control*. Springer London.

Hicks, G. and W. Ray (1971). "Approximation methods for optimal control synthesis". *The Canadian Journal of Chemical Engineering* **49**:4, pp. 522–528.

Houska, B., H. J. Ferreau, and M. Diehl (2011). "ACADO toolkit—an open-source framework for automatic control and dynamic optimization". *Optimal Control Applications and Methods* **32**:3, pp. 298–312.

JModelica.org user Guide. Version Version 1.15. Modelon AB. 198 pp.

Larsson, P.-O., F. Casella, F. Magnusson, J. Andersson, M. Diehl, and J. Å kesson (2013). "A framework for nonlinear model-predictive control using object-oriented modeling with a case study in power plant start-up". In: *Computer Aided Control System Design (CACSD), 2013 IEEE Conference on*. Hyderabad, India.

Lennernäs, B. (2013). *A CasADi Based Toolchain For JModelica.org*. Master's Thesis. Department of Automatic Control, Lund University, Sweden.

Maciejowski, J. (2002). *Predictive Control with Constraints*. Prentice-Hall.

Magnusson, F. (2012). *Collocation methods in JModelica.org*. Master's Thesis. Department of Automatic Control, Lund University, Sweden.

Rawlings, J. and D. Mayne (2009). *Model Predictive Control: Theory and Design*. Nob Hill Pub.

Wächter, A. and L. T. Biegler (2006). "On the implementation of an interior-point filter line-search algorithm for large-scale nonlinear programming". *Mathematical programming* **106**:1, pp. 25–57.

Lund University
Department of Automatic Control
Box 118
SE-221 00 Lund Sweden

Document name
MASTER´S THESIS

Date of issue
August 2015

Document Number
ISRN LUTFD2/TFRT--5987--SE

Author(s)
Magdalena Axelsson

Supervisor
Fredrik Magnusson, Dept. of Automatic Control, Lund University, Sweden
Anders Rantzer, Dept. of Automatic Control, Lund University, Sweden (examiner)

Sponsoring organization

Title and subtitle
Nonlinear Model Predictive Control in JModelica.org

Abstract

In this thesis, a stronger support for Model Predictive Control (MPC) in JModelica.org has been implemented. JModelica.org is an open-source software for simulation and optimization of systems described by Modelica models. MPC is an optimization-based control strategy where one formulates an Optimal Control Problem (OCP) to describe the aim of the controller. At discrete time points the state of the system is estimated and the OCP is solved to find the optimal input to apply to the system. The main goal of this thesis has been to make the time it takes to obtain the optimal input as short as possible and also streamlining the setup of MPC in JModelica.org. This has been done by implementing an MPC class, which utilizes the fact that the structure of the OCP is the same in each consecutive sample for efficiency. Two different benchmarks, one on a smaller problem and one on a larger problem, shows that by using the new MPC framework we obtain similar results as before, but considerably faster. The total average computation time for one sample is decreased by almost 60% for the large problem and by almost 90% for the smaller problem.

Keywords

Classification system and/or index terms (if any)

Supplementary bibliographical information

ISSN and key title
0280-5316

ISBN

Language
English

Number of pages
1-70

Recipient's notes

Security classification

http://www.control.lth.se/publications/