DAS Departamento de Automação e Sistemas
CTC Centro Tecnológico
UFSC Universidade Federal de Santa Catarina

Satisficing Infinite-horizon Model Predictive Control Implementation and Simulation

Relatório submetido à Universidade Federal de Santa Catarina
como requisito para a aprovação da disciplina:
DAS 5501: Estágio em Controle e Automação

Ígor Assis Rocha Yamamoto

Florianópolis, 2017

Satisficing Infinite-horizon Model Predictive Control Implementation and Simulation

Ígor Assis Rocha Yamamoto

Esta monografia foi julgada no contexto da disciplina

DAS 5501: Estágio em Controle e Automação Industrial

e aprovada na sua forma final pelo

Curso de Engenharia de Controle e Automação

Prof. Eduardo Camponogara

Advisor

Abstract

The purpose of this work is to give guidelines for the implementation of Model Predictive Control (MPC) algorithms with an infinite output horizon. MPC is an advanced control technique that is adopted frequently in industry. Different algorithms of MPC are explored here, starting with the basic Generalized Predictive Control (GPC) until the more complex ones with infinite output horizon (IHMPC) and the Distributed Satisficing Predictive Control proposed by Lima [1]. The final algorithm to be developed (S-IHMPC) intends to combine the benefits of the last two approaches: guarantees of nominal stability given by the infinite output horizon and the automatic tunning delivered by the distributed satisficing approach. All implementations and simulations use free software and they are developed with Python. This report shows the initial results of a long-term project and brings as result blocks of reusable code and tools that can be useful both in academia and industry. The code of the GPC algorithm is fully developed and presented here and, for the S-IHMPC, the initial code is presented alongside the guidelines for future works.

Keywords: Model Predictive Control, Infinite horizon.

List of Figures

Contents

1 Introduction

Model Predictive Control (MPC) is an advanced control technique that is widely adopted in industry. An MPC algorithm computes control inputs with the solution of a real-time optimization problem that takes into consideration the system's current state, future targets, possible disturbances and constraints that ensure the integrity of the system. A lot of advantages are associated with MPC: for multiple-input multiple-output (MIMO) systems, MPC takes advantage of the subsystems coupling to enhance its performance (rather than deploying multiple single-input single-output (SISO) PID's that treat other subsystems as disturbances); MPC handles dead-time process intrinsically by its construction, among other benefits [2].

Despite all of MPC advantages, there are some problems regarding practical implementation limitations. The tunning of the controller's parameters and the setting of weights in the cost function (*categoric tunning*) is not an easy task and can lead to system's instability if not properly tuned. One way to guarantee nominal stability of the system is to consider an infinite output horizon (IHMPC) [3]. For the practical implementation of IHMPC, a scheme based on the Output Prediction Oriented Model (OPOM) developed by Rodrigues and Odloak [4] is used here. In addition to the IHMPC, a Distributed Satisficing Predictive Control approach developed by Lima et al. [1] that delivers automatic tuning and dynamically adjusts the weights of each input of cost function (*situational tunning*) can be considered.

All the algorithms are implemented using free software. The open source programming language Python is the core, optimization libraries are used in order to solve quadratic programming problems and libraries for efficient mathematical implementation are adopted.

This work contributed with the development of computational tools for the control of industrial processes. It also produced a framework to implement MPC controllers based on the object oriented paradigm that will be used to continue this work. This report presents the internship activities developed in the first semester of 2017. The work will be continued for the next semester as part of a long-term project. The main activities of the internship include:

- Study MPC and optimization topics.
- Learn and research Python packages for the implementations.
- Study and implement the Generalized Predictive Control (GPC) algorithm.

- Study and implement the OPOM for the IHMPC scheme.
- Apply the algorithms in a case study of a process in the chemical industry.

This work is structured as follows: Chapter 2 gives the necessary background on MPC theory to understand the concepts behind the algorithms; Chapter 3 presents the case study and shows the algorithm implementation and simulation. First, a GPC algorithm is presented, in order to serve as a standard of comparison for the more complex algorithms to be developed. After that the OPOM implementation for the IHMPC is presented and guidelines for the S-IHMPC are given; Chapter 4 gives a conclusion to the work.

2 Model Predictive Control

2.1 MPC Overview

The term model predictive control (MPC) does not designate a specific control strategy but a very ample range of control methods that make explicit use of a model of the process to obtain the control signal by minimizing an objective function. The principal ideas of the predictive control family are [5]:

- Explicit use of a model to predict the process output at future time instants (horizon).
- Calculation of a control sequence minimizing an objective function.
- Receding strategy, so that at each instant the horizon is displaced towards the future, which involves the application of the first control signal of the sequence calculated at each step.

2.1.1 MPC strategy

The methodology of all the controllers belonging to the MPC family is characterized by the following strategy illustrated in Figure 1:

1. The future outputs for a defined horizon N, called the prediction horizon, are predicted at each instant k using the process model. These predicted outputs $\hat{y}(k+j|k)$ for $j = 1, \ldots, N$ depend on the known values up to instant k (past inputs and outputs) and on the future control signals $u(k+j|k), j = 0, \ldots, N-1$, which are to be sent to the system and to be calculated.
2. The set of future control signals are calculated by optimizing a determined criterion in order to keep the process as close as possible to the reference trajectory $w(k+j)$ (which can be the set-point itself or a close approximation of it). This criterion usually takes the form of a quadratic function of the errors between the predicted output signal and the predicted reference trajectory. The control effort is included in the objective function in most cases. An explicit solution can be obtained if the criterion is quadratic, the model is linear and there are no constraints, otherwise an iterative optimization method has to be used. Some assumptions about the structure of the future control law are also made in some cases, such as that it will be constant from a given instant.

3. The control signal $u(k|k)$ is sent to the process whilst, the next control signals calculated are neglected, because at the next sampling instant $y(k+1)$ will already be known, and step 1 will be repeated with this new value and all the sequences will be brought up to date. Thus the $u(k+1|k+1)$ is calculated (which in principle will be different to $u(k+1|k)$ because of the new information available) using the receding horizon concept.

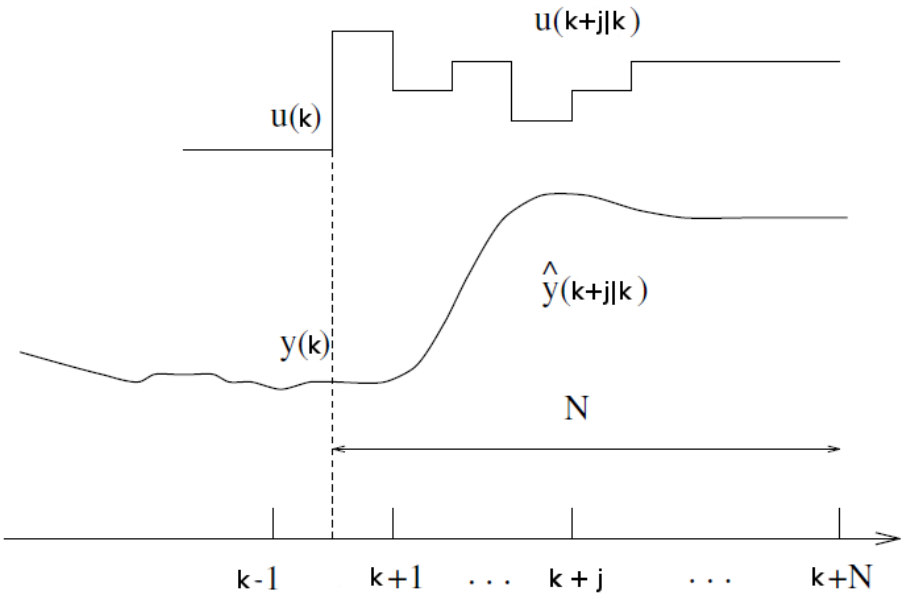


Figure 1 – MPC idea

2.1.2 MPC elements

All the MPC algorithms possess common elements and different options can be chosen for each one of these elements, which gives rise to different algorithms. These elements are

- **The prediction model**

  The process model plays a decisive role in the controller because it is necessary to calculate the predicted output at future instants $\hat{y}(k+j|k)$. The model can be divided into two parts: the actual process model and the disturbance model. Both parts are needed for the prediction.

  – Process model

    Practically every possible form of modeling a process appears in a given MPC formulation, the following being the most commonly used:

    * Impulse response

  - Step response
  - Transfer function

  In this work, step response and transfer function will be the main focus.

- **The objective function**

  The objective function sums up the difference between reference and predicted output through all the horizon. Terms including the weighted sum of the squared inputs and input moves along the control horizon are also considered, so one can prioritize one output of the system.

- **The procedure to obtain the control law.**

  As the MPC optimization problem is a quadratic program, it is solved with the interior point method or Newton method for example.

  – Constraints

    Possible constraints that can be applied to the problem can be on the output or the control action and generally, they represent physical constraints. These constraints are typically, Δu, u_{max}, u_{min}, saturation and operation limits for which the model are valid or security constraints.

2.2 Generalized Predictive Control

Generalized Predictive Control (GPC) is a scheme for MPC that has been widely accepted in both industry and academia. It can handle many different control problems for a wide range of plants, including unstable and integrating processes.

2.2.1 Prediction Model

The GPC approach to compute predictions is based on the CARIMA model [6]:

$$A(z^{-1})y(k) = z^{-d}B(z^{-1})u(k-1) + \frac{C(z^{-1})e(k)}{\Delta} \tag{2.1}$$

where $A(z^{-1})$ and $B(z^{-1})$ are polynomials with degrees n_a and n_b, d is the dead time, $\Delta(z^{-1}) = 1 - z^{-1}$, $e(k)$ is a white noise, $C(z^{-1})$ represents the stochastic characteristics of the noise (it is difficult to estimate in practice, a common choice is to set $C(z^{-1}) = 1$).

The output prediction $\hat{y}$ can be formulated from the solution of the following Diophantine Equation:

$$1 = \tilde{A}(z^{-1})E_j(z^{-1}) + z^{-j}F_j(z^{-1}) \tag{2.2}$$

where $\tilde{A}(z^{-1}) = \Delta A(z^{-1})$, $E_j(z^{-1})$ and $F_j(z^{-1})$ are the unknown variables of the equation.

From Equations (2.1) and (2.2), we can find an expression to calculate the output predictions:

$$\hat{y}(k+j|k) = E_j B\,\Delta u(k+j-1) + F_j\,y(k) \tag{2.3}$$

The term E_jB contains the coefficients of the step response of the system and it is normally represented by G. We can group the terms relative to past inputs and outputs as the free response. The predictions for the horizon $p = N_2 - N_1$ and considering the control horizon equal m is therefore obtained by:

$$\begin{bmatrix} \hat{y}(k+N_1|k) \\ \hat{y}(k+N_1+1|k) \\ \cdots \\ \hat{y}(k+N_2|k) \end{bmatrix} = G \begin{bmatrix} \Delta u(k) \\ \Delta u(k+1) \\ \cdots \\ \Delta u(k+m-1) \end{bmatrix} + F_u \begin{bmatrix} \Delta u(k-1) \\ \Delta u(k-2) \\ \cdots \\ \Delta u(k-n_b) \end{bmatrix} + F_y \begin{bmatrix} \hat{y}(k+N_1-1|k) \\ \hat{y}(k+N_1-2|k) \\ \cdots \\ \hat{y}(k+N_1-n_a|k) \end{bmatrix}$$

$$\hat{y} = G\Delta u + f \tag{2.4}$$

2.2.2 Control Algorithm

The objective of a GPC controller is to drive the output as close to the set-point as possible in a least-squares sense with the possibility of the inclusion of a penalty term on the input moves. Therefore, the manipulated variables are selected to minimize a quadratic objective that can consider the minimization of the future errors and the control effort:

$$J = \sum_{j=N_1}^{N_2} \delta(j)[\hat{y}(k+j|k) - \omega(k+j)]^2 + \sum_{j=1}^{N_u} \lambda(j)[\Delta u(k+j-1)]^2 \tag{2.5}$$

$$= (\hat{y}-\omega)^T Q_\delta (\hat{y}-\omega) + \Delta u^T Q_\lambda \Delta u \tag{2.6}$$

where Q_δ and Q_λ are the diagonal weight matrices with values of $\delta(j)$ and $\lambda(j)$, respectively, and ω is the output reference.

If $\hat{y}$, given in Equation (2.4), is introduced in Equation (2.6), then J becomes a function only of the future control increments Δu and the free response f. Rewriting the cost function with the introduction of $\hat{y}$, we have:

$$J = \frac{1}{2}\Delta u^T H \Delta u + q^T \Delta u + f_0 \tag{2.7}$$

where $H = 2(G^T Q_\delta G + Q_\lambda)$, $q = 2G^T Q_\delta (f-\omega)$, $f_0 = (f-\omega)^T Q_\delta (f-\omega)$.

The algorithm to be implemented consists in minimizing Equation (**??**), subject to linear constraints, like $A\Delta u \leq b$. Once the optimization problem is solved with Δu as the decision variable, the next control increments are obtained and the receding horizon strategy is applied.

2.3 Infinite-horizon MPC

One frequently debated issue about MPC is the nominal stability of the strategy. It is of great interest an implementation of a stable controller independently of its tuning. In the literature, one can find basically three approaches concerning guarantees of stability [4]:

- state terminal constraints.
- state terminal set constraints.
- infinite output prediction horizon.

The first two approaches are based on the addition of constraints in the optimization problem in such a way that all the states are forced to zero at the end of the output prediction horizon. It seems to be the simplest and more intuitive methods, however these strategies hold some issues: they reduce drastically the feasible region of the control problem; the attraction region is very sensitive to disturbances or big changes on the set point; and the computational efforts to evaluate the control law at every instant are often impracticable.

The third group of strategies, considering an infinite output prediction horizon (IHMPC), was proposed by Rawlings and Muske (1993) [3] and the stability of the method was shown for stable and unstable systems subject to control input and state constraints. The infinite-horizon MPC is characterized by the objective function given in Equation (2.6) with N_2 going to infinity. In general, the IHMPC approach transforms the infinite output horizon in a finite horizon through a penalty weight matrix at the end of the input horizon. The terminal weight is obtained by solving a discrete-time Lyapunov equation.

IHMPC works very well theoretically, however the terminal weight computation is a huge challenge to be performed on-line. Disturbances and big changes on the set point can make the implementation unfeasible. To solve this issue, a different model for the output prediction in the infinite horizon can be used: the Output Prediction Oriented Model (OPOM).

2.3.1 Output Prediction Oriented Model

OPOM defines a prediction strategy based on a state-space model obtained from the step response of the system. In terms of computational effort, IHMPC with OPOM as prediction model, offers an equivalent solution comparing to terminal state constrained MPC, however one does not need to choose a prediction horizon. Besides ensuring nominal stability independently of the tuning of matrix weights and parameters in the objective function.

The main ideas of the model of OPOM are:

- Divide the states of the system in three parts according to its step response: steady state, poles dynamics and the integrating part. A state for each part of the system is created and its evolution along time is considered. The creation of these states allows the proposition of constraints that ensure the convergence of the response at the end of the control horizon.

- Create an output model that can make predictions continuously along the time at each sample time. Therefore, the objective function can be reformulated as an integral of the output error, which has an analytical solution, eliminating the need to calculate a penalty weigh matrix via the Lyapunov equation.

To clarify how OPOM is built, consider the following example:

- Second order system with one stable pole and one integrator. The system is represented by the transfer function $G(s) = \frac{K}{(\tau s+1)s}$. The corresponding step response is giving by $S(t) = -K\,\tau + K\,\tau\,e^{-t/\tau} + K\,t = d^0 + d^d\,e^{-t/\tau} + d^i\,t$. For this step response expression, we can find a corresponding state space model that represents the OPOM formulation:

$$x_{k+1} = A\,x_k + B\,\Delta u_k \tag{2.8}$$

$$y_k(t) = C(t)x_k \tag{2.9}$$

where $A = \begin{bmatrix} 1 & 0 & Ts \\ 0 & e^{-Ts/\tau} & 0 \\ 0 & 0 & 1 \end{bmatrix}$, $B = \begin{bmatrix} d^0 + d^i\,Ts \\ d^d\,e^{-Ts/\tau} \\ d^i \end{bmatrix}$, $C(t) = \begin{bmatrix} 1 & e^{-t/\tau} & t \end{bmatrix}$, $x = \begin{bmatrix} x^s \\ x^d \\ x^i \end{bmatrix}$, Ts is the sample time.

Observe that, in Equation (2.9), $C(t)$ is not constant, it is time dependent. This property at sampling step k allows the entire output trajectory prediction at that sample.

Regarding the meaning of each state, x^s corresponds to the integrating part introduced by the incremental form of the model, x^d corresponds to the dynamic of the stable pole and x^i corresponds to the integrating pole.

The previously example is an illustration for a simple case where OPOM can be formulated. A generalization of the model is made on [4], [7] and [8] for MIMO systems with multiple poles, integrator and dead time considerations.

2.3.2 Control Algorithm

The IHMPC algorithm calculates the control increment signal by minimizing a quadratic cost function. This cost function can be formulated with an integral of the

output error since the OPOM model allows the predictions continuously along the time at each sample

$$J = \sum_{n=1}^{m} \int_{(n-1)Ts}^{nTs} [e(t)]_k^T Q_\delta [e(t)]_k \, dt + \int_{mTs}^{\infty} [e(t)]_k^T Q_\delta [e(t)]_k \, dt + \\ + \sum_{j=0}^{m-1} \Delta u_{k+j}^T \, Q_\lambda \, \Delta u_{k+j}$$

The integrals from the cost function can be analytically solved as shown in [7]. Therefore a new expression for the optimization problem can be obtained, equivalent to the format of Equation (2.7).

2.4 Satisficing MPC

In general the MPC controllers have what is called categoric tunning, the weights of the subsystems are fixed. On the other hand, the satisficing MPC controller proposed in [1] implements situational altruism, where an equivalent global cost arises from the interaction between controllers. The satisficing controllers try to find a solution that lies in a region that is satisfactory and sufficient for all controllers (satisficing = satisfy + suffice).

In the satisficing approach [9], a minimum performance is defined which is local for each controller. Any solution that leads to a better performance than the minimum is a valid and satisfactory solution. The satisficing controllers are also altruistic since they incorporate the satisfaction of others into their satisfaction.

Situational altruism makes those controllers with worse local performance (measured against the minimum performance) acquire more importance in comparison with those performing better. This reason of importance is adjusted automatically in a dynamic way. A fairer relationship between controllers is thus expected based on local criteria.

2.5 Summary

In this chapter, it was presented a brief review of model predictive control. All the concepts required to understand the implementation of the algorithms in this work were shown. A general intuition on MPC, with the main idea of the strategy and its elements. Followed by specific strategies inside the context of MPC: GPC, an algorithm that can handle unstable and integrating systems; and the IHMPC (using OPOM to compute the predicted output) approach that guarantees stability for MPC.

3 Implementation, Simulation and Results

3.1 Implementation

All the implementations of MPC algorithms in this work are written in Python 3.6 (code can be found in the Appendix). The object oriented paradigm is used to produce a modular piece of software, where all the classes and their methods can be easily reused for different applications. Here we describe a framework that defines how the code must be structured.

The framework consists in the construction of three main classes: one to represent the system (*SystemModel*), one to represent the digital controller (*Controller*) and one to represent the simulation environment (*Simulation*). Other classes are inherited from this core set of classes for each algorithm (GPC and IHMPC). The *GPCController* and the *IHMPCController* are build on top of *Controller*. And to represent the prediction models of each algorithm, the *CARIMA* and *OPOM* are build on top of *SystemModel*. Each controller's algorithm class has a relationship of composition with the prediction model. The class diagram (Figure 2) below represents the conception of this work. Following, the attributes and methods of the core classes are explained and in the next sections the implementation of the algorithms classes is detailed.

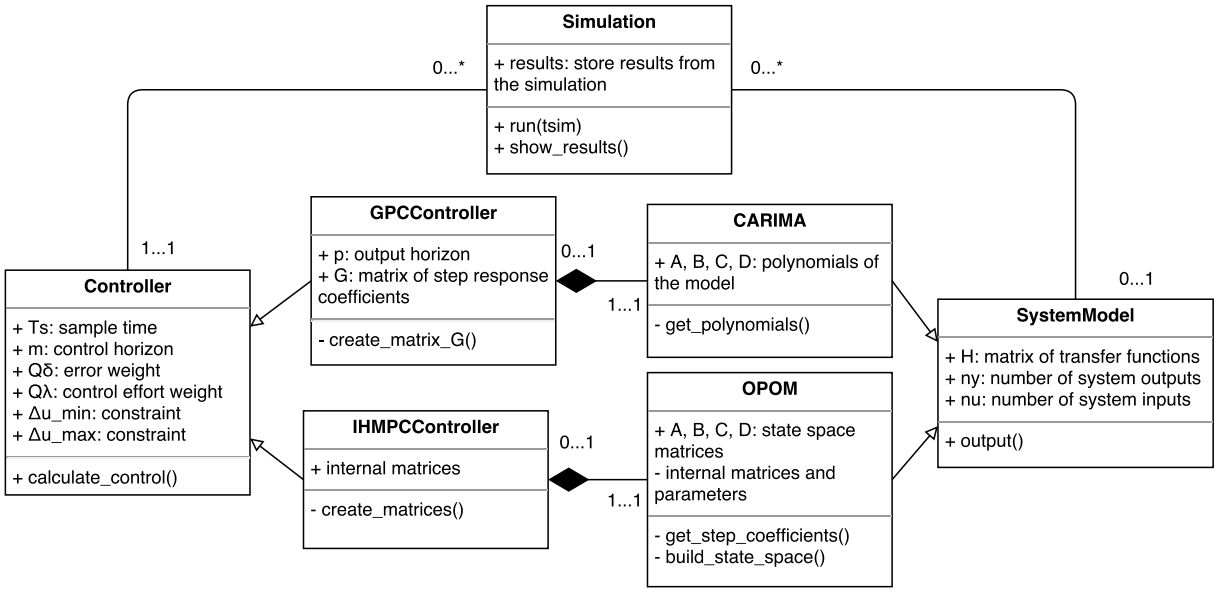


Figure 2 – Class diagram

- Classes:

1. ***SystemModel*:** this class is built from the transfer function of a SISO system

or a matrix of transfer functions of a MIMO system that the user will pass as input to build the prediction model of the control algorithm.

SystemModel
+ H: matrix of transfer functions + ny: number of system outputs + nu: number of system inputs
+ output(): return output given the control signal

Table 1 – Class diagram representation of *SystemModel*

2. ***Controller*:** this class represent the digital controller to be implemented. Its attributes contain the parameters to be tunned and its methods compute the next control action. A child controller class for each MPC algorithm is implemented.

Controller
+ Ts: sample time + m: control horizon + weights + constraints
+ calculate_control(): return next control action

Table 2 – Class diagram representation of Controller

3. ***Simulation*:** this class creates an environment for simulation. It is built giving the following inputs: one object of controller and one optional object of real system model for robustness analysis (if not passed, the model used by the controller is considered the same as the real model). A method is created for simulation running, passing the time of simulation as argument.

Simulation
+ results: store the results from the simulation
+ run(tsim): run simulation given the simulation duration as input + show_results(): displays the last result of the simulation in a graphic

Table 3 – Class diagram representation of Simulation

The following packages, available in the Python community, are required as dependencies of the implementation:

- Python packages

  1. **Numpy:** the fundamental package for scientific computing with Python. It contains an efficient data structure (numpy.array) to handle mathematical operations with N-dimensional data.
  2. **Scipy:** it offers a toolbox for engineering. In this work, the module of the library called *signal* is used for signal processing and operations with transfer functions.
  3. **Cvxopt:** this package offers solvers for optimization problems. In particular, the quadratic programming solver is used for the MPC algorithms.
  4. **Matplotlib:** it is used for visual analysis of the simulations.

3.2 Case Study

In this section, we illustrate an application of the algorithms proposed in this work. This case study is based on the ethylene oxide reactor system (Figure 3), presented by [4]. This is a typical example of the chemical process industry that exhibits stable and integrating poles.

The production of ethylene-oxide (EO) is based on the exothermic reaction

$$C_2H_4 + \frac{1}{2}\,O_2 \rightarrow C_2H_4O \tag{3.1}$$

The complete combustion of ethylene occurs through the undesirable secondary reaction

$$C_2H_4 + 3\,O_2 \rightarrow 2\,CO_2 + 2\,H_2O \tag{3.2}$$

The gaseous mixture that leaves the reactor is sent to an absorber where the ethylene-oxide is extracted. The remaining gas is recycled through other chemical processes and it is mixed with the feed stream. One can observe that this procedure of recycling the gas introduces an integrating dynamic, since the introduction of a feed component in excess tends to increase the concentration of this component in the recycle gas continuously.

Typical process variables considered in the problem are:

- Manipulated variables (inputs)
  - the pure oxygen (O_2) feed flow rate
  - the ethylene (C_2H_4) feed flow rate
  - the nitrogen (N_2) feed flow rate
  - the temperature of cooling oil used in the reactor
  - the combustion inhibitor (EDC) feed flow rate

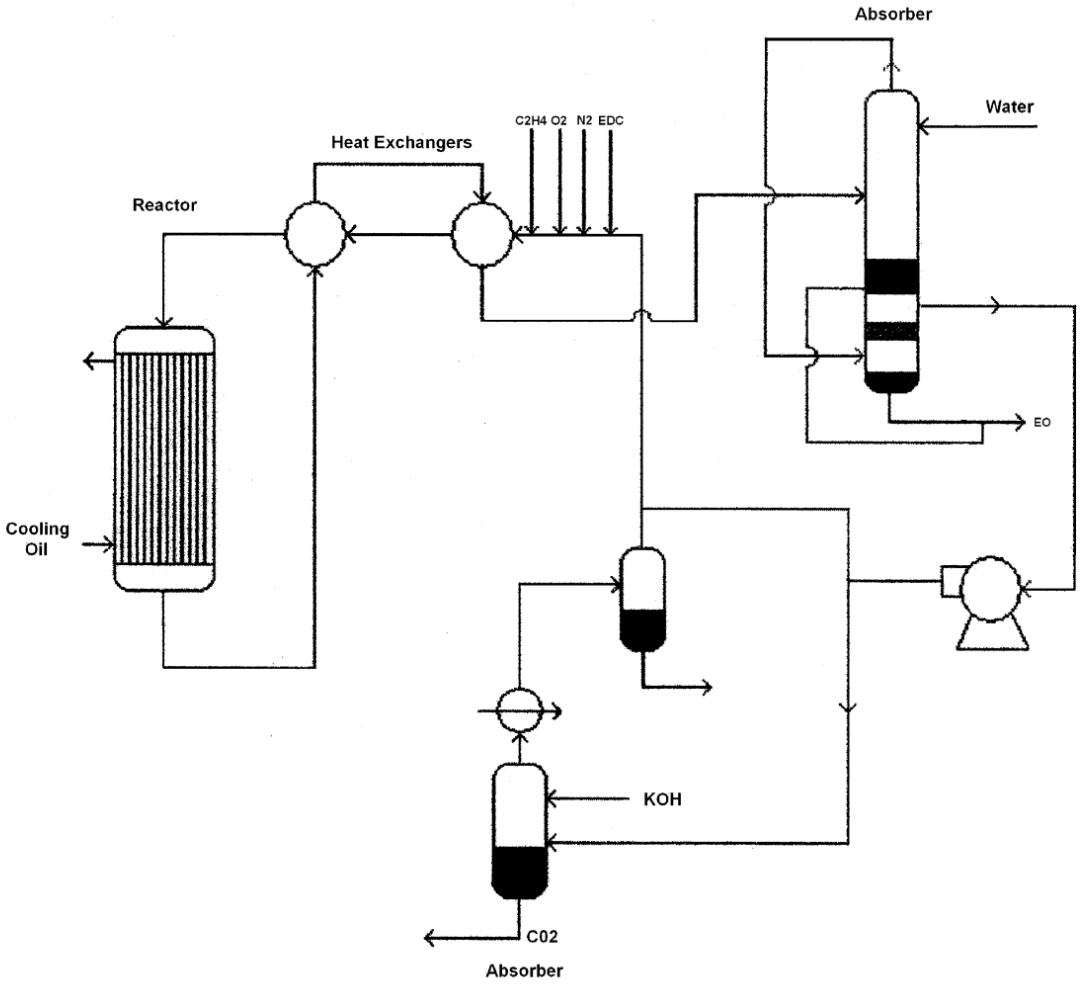


Figure 3 – Schematic representation of an industrial ethylene-oxide plant.

  - the potassium hydroxide flow rate, introduced in the second absorber

- Process variables (outputs)
  - the reaction selectivity, which is defined as the percentage of the reacted ethylene that is converted to ethylene-oxide
  - the molar fraction of ethylene in the gas stream at the entrance of the reactor
  - the temperature of reaction products in reactor
  - the system pressure
  - the molar fraction of oxygen in the gas stream at the entrance of the reactor

For simplicity and to compare results with published papers [7], we consider a small example of the ethylene-oxide system represented by the following matrix of transfer

function:

$$H(s) = \begin{bmatrix} \frac{-0.19}{s} & \frac{-1.7}{19.5\,s+1} \\ \frac{-0.763}{31.8\,s+1} & \frac{0.235}{s} \end{bmatrix} \tag{3.3}$$

where each element $H_{i,j}$ of $H(s)$ represents the transfer function of the output/input pair (y_i, u_j).

3.3 GPC Implementation

In this section, we will present all the steps for the implementation of an MPC algorithm based on the GPC scheme. First, the construction of the GPC controller is presented, then a simulation for the case study is shown.

3.3.1 Prediction Model

The construction of the prediction model starts with the creation of the prediction model *CARIMA*, based on *SystemModel* class. As stated before, this class stores the matrix of transfer functions of the system and it has a method to compute the output response given a control input. Moreover, the *CARIMA* class needs to store the polynomials of the model, presented in Equation (2.1), and needs an internal method to get them from the transfer functions given as user input.

The next step is to implement the controller's class, the *GPCController* class is built. In the controller initialization all the parameters and weights passed by the user are set up and as the matrix G never changes, it can also be initiated off-line with an internal method to generate it.

Finally, to complete the Equation (2.4) and to become possible the output prediction calculation, the free response can be obtained recursively at each step in the simulation.

3.3.2 GPC Controller

The controller implemented deals with both SISO and MIMO systems. It calculates actions for the nu inputs to control the ny outputs of the system. It uses an output prediction horizon of p samples and a control horizon of m samples. Two matrices of weights Q_δ and Q_λ are used for the prediction error and control increment effort terms of the cost function, respectively. The constraints du_{min} and du_{max} in the magnitude of control increment are also passed to the controller. To sum up, Table 4 shows the GPC controller parameters:

Table 4 – Parameters of GPC controller

Parameter	**Description**
ny	number of system outputs
nu	number of system inputs
p	output prediction horizon
m	control action horizon
Q_δ	matrix of weights for the prediction error
Q_λ	matrix of weights for the control increment effort
Δu_{min}	minimum increment on the control input
Δu_{max}	maximum increment on the control input

The control algorithm returns the next control action given as inputs the current and the past inputs/outputs and the future set-points. This function is implemented as a method of the class *GPCController* and it builds the matrices in Equation (**??**) and gives them as input for the quadratic programming solver, which returns the next control actions.

3.3.3 Case Study Simulation

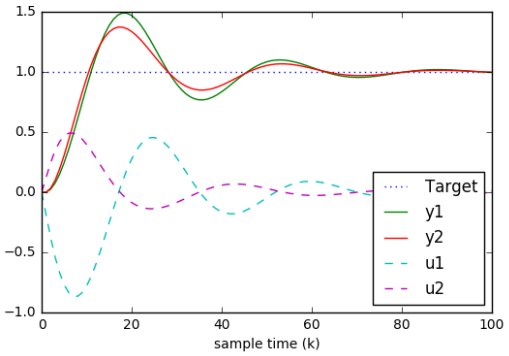
Different scenarios are simulated for the case study. First, the weight matrices are chosen and the prediction horizon p and the control horizon m are adjusted based on the response of a unit step in both output set-points. Once the horizon parameters are obtained, different changes in the set-point and different initial conditions are tested. Lastly, a robustness analysis varying the real gains of the system is done.

The simulations illustrated in Figure 4 show the process of tuning the controller until the system response does not exhibit overshoot. The adjusted controller parameters are presented in Table 5.

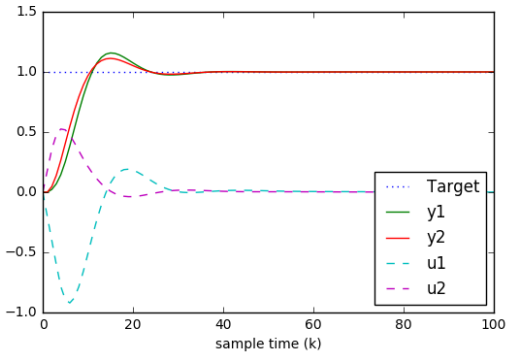
Table 5 – Parameters of Controller

ny	nu	p	m	Ts	Q_δ	Q_λ	Δu_{min}	Δu_{max}
2	2	10	3	1min	diag(1,. . .,1)	diag(10,. . .,10)	-0.2	0.2

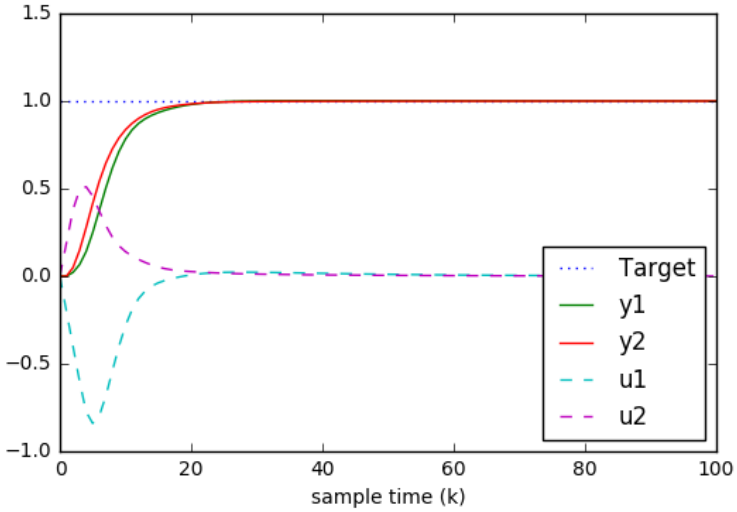
For the simulation in Figure 4c, the plot in Figure 5 reveals the evolution of the primal objective value of the optimization problem along each sample. There is a peak between samples $k = 0$ and $k = 20$ due to the increase in the variation of the control signal.



(a) Horizons: $m = 3$, $p = 5$. Overshoot = 50%



(b) Horizons: $m = 3$, $p = 7$. Overshoot = 15%



(c) Horizons: $m = 3$, $p = 10$. No overshoot

Figure 4 – Controller tuning - simulation of GPC algorithm

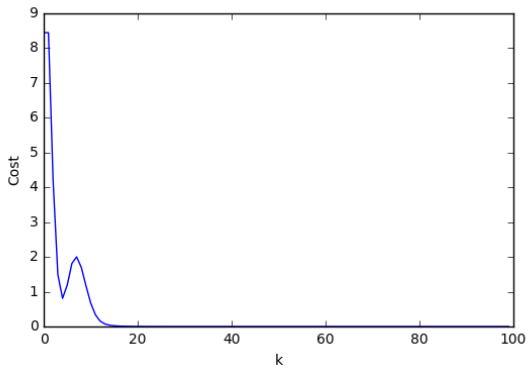
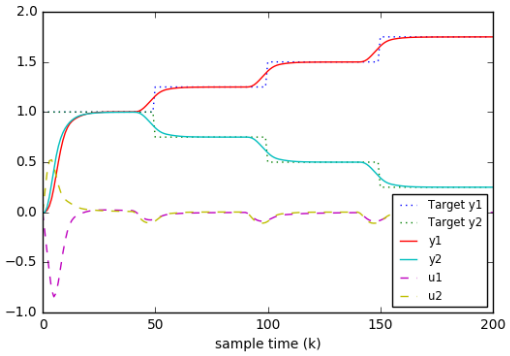


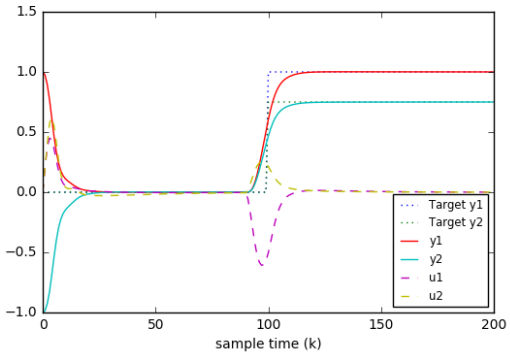
Figure 5 – Primal objective value

After the process of tunning the controller, other simulations were performed (Figure 6) in order to test different scenarios of the case study. First, a sequence of steps on the set-points was applied with output states initially null. Then, the initial states were

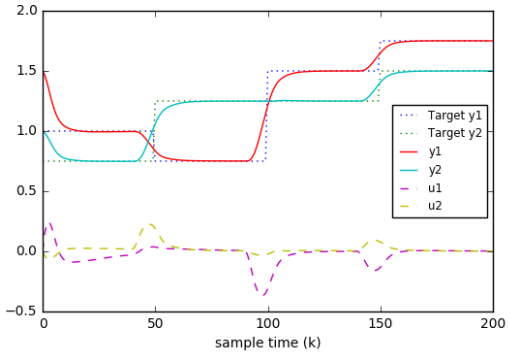
set with different values for the next simulations. At last, a ramp signal was applied on the set-points.



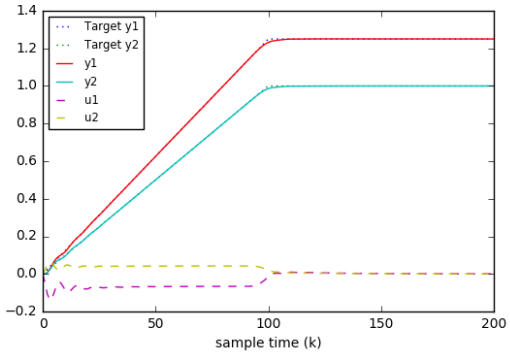
(a) Steps, with $y_1(0) = y_2(0) = 0$



(b) Steps, with $y_1(0) = 1$ and $y_2(0) = -1$



(c) Steps, with $y_1(0) = 1.5$ and $y_2(0) = 1$



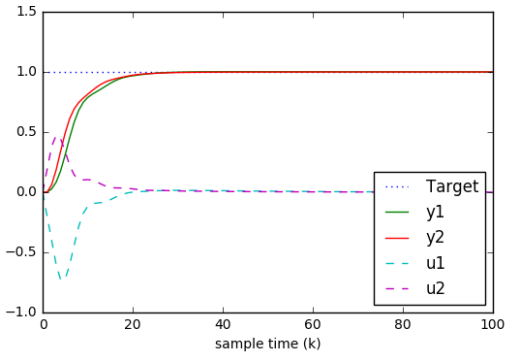
(d) Ramp, with $y_1(0) = y_2(0) = 0$

Figure 6 – Simulations for different changes on set-point

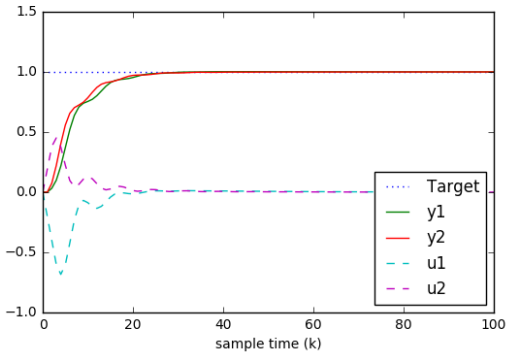
A robustness test is performed for the case where the gains of the system used as model for the controller and the real system are different. Errors of 25%, 50%, 75% and 100% in systems model gains are imposed. The results are shown in Figure 7.

We can conclude that the controller responds well to variations in the gain of the system. The response changes minimally till the 50% of gain error. From the 75% of gain error, the response starts to oscillate till the appearance of a limit cycle when the error reaches 100%.

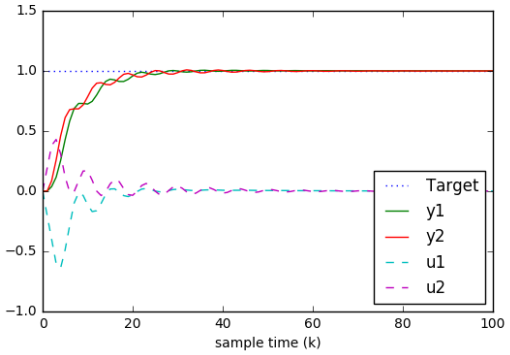
The controller can still be re-tuned in order to obtain better responses for large gain errors and remove the limit cycle. The simulations in Figure 8 show the processes of re-tuning the controller's parameters.



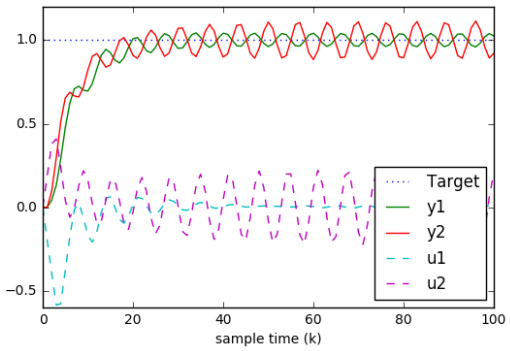
(a) $m = 3$, $p = 10$ and gain error = 25%



(b) $m = 3$, $p = 10$ and gain error = 50%

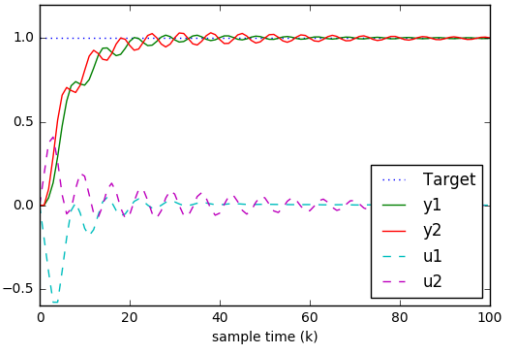


(c) $m = 3$, $p = 10$ and gain error = 75%

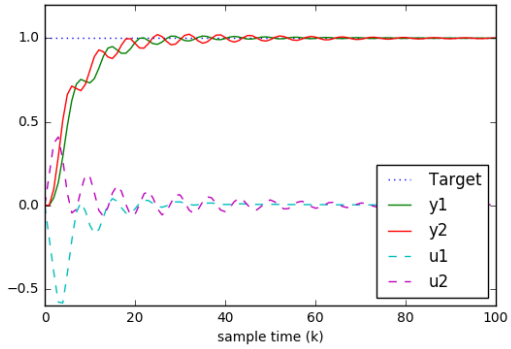


(d) $m = 3$, $p = 10$ and gain error = 100%

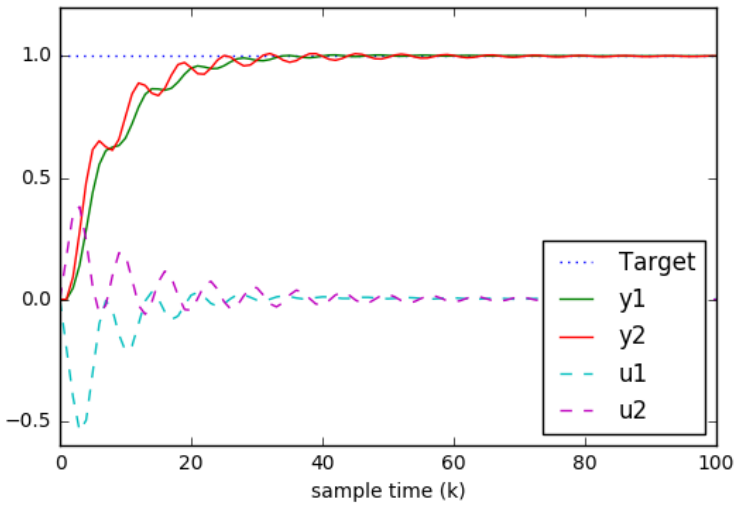
Figure 7 – Robust analysis - simulation of GPC algorithm with gain errors



(a) $m = 4$, $p = 10$ and gain error = 100%



(b) $m = 5$, $p = 10$ and gain error = 100%



(c) $m = 5$, $p = 15$ and gain error = 100%

Figure 8 – Controller re-tuning - simulation of GPC algorithm with 100% of gain error

3.4 IHMPC Implementation

In this section, we will present the initial implementation of the IHMPC algorithm. First, the prediction model OPOM is developed and a unit test is performed in order to validate the model implementation. Next, we discuss the implementation of the IHMPC controller and give guidelines for the continuation of the development.

3.4.1 Prediction Model

The IHMPC algorithm implemented here has the OPOM as the prediction model as explained in the previous chapter. For the implementation, a class named *OPOM* is created on top of the *SystemModel* class. Its attributes hold the state space model (matrices A, B, C and D), a vector X of the current state of the system and internal

parameters that are used to build the state space model. In relation to the methods of the class, *OPOM* needs an internal method to get the coefficients of the analytical expression of the step response from the transfer functions given as user input. Besides that there is a method to create the model, returning the matrices of the state space on the initialization of one instance of the class.

3.4.2 Validation of the Prediction Model

After the design of the *OPOM* class, a unit test was performed in order to assure the correctness of the implementation. The transfer matrix of the case study was used as input to the class and on the initialization of the instance, the following state space system was generated:

$$X_{k+1} = A\,X_k + B\Delta u_k$$
$$y(t)_k = C(t)\,X_k$$

where

$$X = \begin{bmatrix} x_1^s \\ x_2^s \\ x_1^d \\ x_2^d \\ x_1^i \\ x_2^i \end{bmatrix}; \quad A = \begin{bmatrix} 1 & 0 & 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0.950 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0.969 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 \end{bmatrix}; \quad B = \begin{bmatrix} -0.190 & -1.700 \\ -0.763 & 0.235 \\ 0 & 1.615 \\ 0.739 & 0 \\ -0.190 & 0 \\ 0 & 0.235 \end{bmatrix};$$

$$C(t) = \begin{bmatrix} 1 & 0 & e^{-0.05t} & 0 & t & 0 \\ 0 & 1 & 0 & e^{-0.03t} & 0 & t \end{bmatrix}$$

For each output there is a state x^s that corresponds to the integrating state introduced by the incremental form of the input, a state x^d that corresponds to the dynamics of the stable poles and a state x^i for the integrating poles.

The generated matrices A, B and C match what was presented in [7]. To complete the validation test, the method of the class *OPOM* to predict the output of the system was called with the states zeroed. Applying the same sequence of control signals that the GPC algorithm calculated before (Figure 4) and calling the output prediction method after each application of the control increment, we achieve the same graphic (Figure 9) as before.

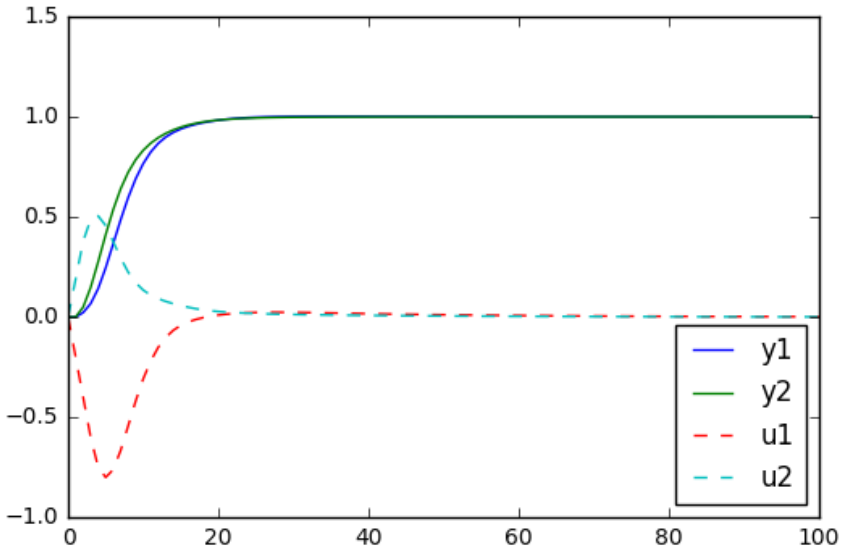


Figure 9 – Simulation to validate the OPOM

3.4.3 IHMPC Controller

The next step of the IHMPC algorithm implementation is the creation of a class on top of the *Controller* class that is named *IHMPCController*. The class has internal matrices as attributes that are used to generate the inputs for the QP solver (matrix H and vector q of Equation (2.7)). On the initialization of an instance of the class, an internal method is called to generate the matrices that can be calculated off-line based on the OPOM model. No additional parameter is needed besides those already passed to *Controller* class.

Finally, to calculate the next control action, the *IHMPCController* method *calculate_control()* receives the current state of the system with the state of the class *OPOM*. Having the states x^s, x^d and x^i, the algorithm generates the inputs for the solver and then calls it to solve the optimization problem on every sample step of the simulation.

The implementation of the controller is in phase of unit tests. For future works, the controller can still be improved with the addition of extra features, like slack variables to overcome possible infeasibilities and targets for the manipulated variable. *OPOM* can also be extend for systems with dead time [8].

3.5 Summary

In this chapter we presented the implementation, simulation and results of the MPC algorithms. First the framework and the implementation planning were exposed and the class diagram of the project was shown. Thereafter the case study was detailed and

the transfer matrix of the system was presented. Afterwards the step by step of the GPC implementation was described and simulations for different scenarios were made. Finally, the IHMPC implementation was described and tests of its prediction model were shown.

4 Conclusion

In this report, we presented guidelines for MPC algorithms implementation. First, we created a programming framework to implement the code in Python. Following this framework, we designed a GPC controller and tested it for the case study, represented by a real problem in the chemical industry. Next, we implemented the OPOM in Python and reproduced the results found in [7]. At last, we discuss the future continuation of the algorithms (IHMPC and S-IHMPC) implementation.

The Python code produced here was designed to be modular and, consequently, easy to implement in real applications. The implementation intends to suit the general case even though the code developed till this point has some parts specifically designed for the case study. The process to build the algorithm is iterative and as the implementation is still in an early version, new features need to be added and code optimization must be implemented.

In the near future, the implementation of the IHMPC will be continued and the development of the S-IHMPC will begin afterward. Suggestions for future works are the consideration of dead time in OPOM, inclusion of slack variables and targets for the manipulated variable and simulations for different processes.

Bibliography

1 LIMA, M. L. de et al. Distributed satisficing MPC. *IEEE Transactions on Control Systems Technology*, v. 23, n. 1, p. 305–312, 2015. Citado 3 vezes nas páginas 2, 6, and 14.

2 CAMACHO, E.; BORDONS, C. *Model Predictive Control*. Springer London, 2004. (Advanced Textbooks in Control and Signal Processing). Disponível em: <https://books.google.com.br/books?id=Sc1H3f3E8CQC>. Citado na página 6.

3 RAWLINGS, J. B.; MUSKE, K. R. The stability of constrained receding horizon control. *IEEE Transactions on Automatic Control*, v. 38, n. 10, p. 1512–1516, Oct 1993. Citado 2 vezes nas páginas 6 and 12.

4 RODRIGUES, M. A.; ODLOAK, D. An infinite horizon model predictive control for stable and integrating processes. *Computers and Chemical Engineering*, v. 27, 2003. Citado 4 vezes nas páginas 6, 12, 13, and 17.

5 CAMACHO, C. B. E. F. *Model Predictive Control*. [S.l.]: Springer, 1999. (Advanced textbooks in control and signal processing). Citado na página 8.

6 NORMEY-RICO, J. E.; CAMACHO, E. *Control of Dead-time Processes*. [S.l.]: Springer, 2007. (Advanced textbooks in control and signal processing). Citado na página 10.

7 CARRAPICO, O. L.; ODLOAK, D. A stable model predictive control for integrating processes. *Computers and Chemical Engineering*, v. 29, 2005. Citado 5 vezes nas páginas 13, 14, 18, 25, and 28.

8 SANTORO, B. F.; ODLOAK, D. Closed-loop stable model predictive control of integrating systems with dead time. *Journal of Process Control*, v. 22, 2012. Citado 2 vezes nas páginas 13 and 26.

9 SIMON, H. A. A behavioral model of rational choice. *The quarterly journal of economics*, v. 69, p. 99–118, 1955. Citado na página 14.

Appendix

APPENDIX A – Implementation Code

A.1 Core classes

```
# -*- coding: utf-8 -*-
"""
Created on Thu Mar 31 17:03 2017

@author: Igor Yamamoto
"""
import numpy as np
import matplotlib.pyplot as plt
import scipy.signal as signal
from cvxopt import matrix, solvers


class SystemModel(object):
    def __init__(self, H):
        self.H = np.array(H)
        self.ny = self.H.shape[0]
        self.nu = self.H.shape[1]

    def output(self, X0=None, T=None, N=None):
        def fun(X02=None, T2=None, N2=None):
            def fun2(sys):
                return signal.step(sys, X02, T2, N2)
            return fun2
        fun3 = fun(X0, T, N)
        step_with_time = list(map(fun3, self.H))
        return np.array([s[1] for s in step_with_time])


class Controller(object):
    def __init__(self, Ts, m, Q, R, du_min, du_max):
        self.Ts = Ts
        self.m = m
        self.Q = Q
```

```
        self.R = R
        self.du_min = du_min
        self.du_max = du_max


class Simulation(object):
    def __init__(self, w, controller, real_system=None):
        self.results = {"y": [], "u": []}
        self.w = w
        self.controller = controller
        if real_system:
            self.real_system = real_system
        else:
            self.real_system = controller.system

    def run(self, tsim):
        # Initialization
        y11 = 0*np.ones(tsim+1)
        y12 = 0*np.ones(tsim+1)
        y21 = 0*np.ones(tsim+1)
        y22 = 0*np.ones(tsim+1)
        u1 = np.zeros(tsim+1)
        u2 = np.zeros(tsim+1)
        du1 = np.zeros(tsim+1)
        du2 = np.zeros(tsim+1)
        y11_past = 0*np.ones(na11)
        y12_past = 0*np.ones(na12)
        y21_past = 0*np.ones(na21)
        y22_past = 0*np.ones(na22)
        u1_past = np.zeros(nb)
        u2_past = np.zeros(nb)

        J = np.zeros(tsim)
        Status = ['']*tsim
        # Control Loop
        for k in range(1,tsim+1):
            y11[k] = -Ar11[1:].dot(y11_past[:-1])
                     + Br11.dot(u1_past)
            y12[k] = -Ar12[1:].dot(y12_past[:-1])
```

```
                + Br12.dot(u2_past)
            y21[k] = -Ar21[1:].dot(y21_past[:-1])
                + Br21.dot(u1_past)
            y22[k] = -Ar22[1:].dot(y22_past[:-1])
                + Br22.dot(u2_past)

            # Select references for the current horizon
            w = np.append(self.w[0][k:k+p], self.w[1][k:k+p])
            du_past = np.array([du1[k-1], du2[k-1]])
            y_past = [y11_past, y12_past, y21_past, y22_past]
            current_du = [np.array([du1[k]]), np.array([du2[k]])]
            dup,j,s =
            self.controller.calculate_control(w,
                                              du_past=du_past,
                                              y_past=y_past,
                                              current_du=current_du)

            du1[k] = dup[0]
            du2[k] = dup[m]
            u1[k] = u1[k-1] + du1[k]
            u2[k] = u2[k-1] + du2[k]

            u1_past = np.append(u1[k],u1_past[:-1])
            u2_past = np.append(u2[k],u2_past[:-1])
            y11_past = np.append(y11[k],y11_past[:-1])
            y12_past = np.append(y12[k],y12_past[:-1])
            y21_past = np.append(y21[k],y21_past[:-1])
            y22_past = np.append(y22[k],y22_past[:-1])

            J[k-1] = abs(j)
            Status[k-1] = s
        self.results['y'] = [y11+y12, y21+y22]
        self.results['u'] = [u1, u2]

    def show_results():
        plt.clf()
        plt.plot(w1[:-p],':', label='Target y1')
        plt.plot(w2[:-p],':', label='Target y2')
        plt.plot(self.results['y'][0], label='y1')
```

```
        plt.plot(self.results['y'][1], label='y2')
        plt.plot(self.results['u'][0],'--', label='u1')
        plt.plot(self.results['u'][0],'--', label='u2')
        plt.legend(loc=0, fontsize='small')
        plt.xlabel('sample time (k)')
        plt.show()
        #plt.savefig('sim8.png')
        return J, Status
```

A.2 GPC

```
# -*- coding: utf-8 -*-
"""
Created on Thu Mar 31 17:03 2017

@author: Igor Yamamoto
"""
import numpy as np
import matplotlib.pyplot as plt
import scipy.signal as signal
from cvxopt import matrix, solvers


class CARIMA(SystemModel):
    def __init__(self, H):
        super.__init__(H)
        A, A_til, B, C = _get_polynomials()

    def _get_polynomials():
        Bm11 = np.array([-0.19])
        Am11 = np.array([1, -1])
        Am11_til = np.convolve(Am11, [1,-1])

        Bm12 = np.array([-0.08498])
        Am12 = np.array([1, -0.95])
        Am12_til = np.convolve(Am12, [1,-1])

        Bm21 = np.array([-0.02362])
        Am21 = np.array([1, -0.969])
        Am21_til = np.convolve(Am21, [1,-1])
```

```
        Bm22 = np.array([0.235])
        Am22 = np.array([1, -1])
        Am22_til = np.convolve(Am22, [1,-1])

        A = [Am11, Am12, Am21, Am22]
        A_til = [Am11_til, Am12_til, Am21_til, Am22_til]
        B = [Bm11, Bm12, Bm21, Bm22]
        C = 1
        return A, B, C


class GPCController(Controller):
    def __init__(self, Ts, m, Q, R, du_min, du_max):
        super.__init__(self, Ts, m, Q, R, du_min, du_max)
        self.model = ARIMA(H)
        self.p = p
        self.G = self._create_matrix_G()


    def _create_matrix_G(self):

        def create_G(self, g, p, m):
            g = np.append(g[:p],np.zeros(m-1))
            G = np.array([])
            for i in range(p):
                G = np.append(G,[g[i-j] for j in range(m)])
            return np.resize(G,(p,m))

        T = np.array(range(1, 200, self.Ts))
        g11, g12, g21, g22 = ethylene.step_response(T=T)
        G11 = create_G(g11,p,m)
        G12 = create_G(g12,p,m)
        G21 = create_G(g21,p,m)
        G22 = create_G(g22,p,m)
        G1 = np.hstack((G11,G12))
        G2 = np.hstack((G21,G22))
        G = np.vstack((G1,G2))
        return G
```

```
def calculate_control(self, w, **kwargs):
    dm = 0
    y11_f = np.zeros(self.p)
    y12_f = np.zeros(self.p)
    y21_f = np.zeros(self.p)
    y22_f = np.zeros(self.p)
    # Free Response
    du1, du2 = kwargs['current_du']
    du1_f, du2_f = kwargs['du_past']
    y11_aux, y12_aux, y21_aux, y22_aux = kwargs['y_past']
    for j in range(p):
        if j <= dm:
            du1_f = du1
            du2_f = du2
        else:
            du1_f = du2_f = np.array(0)
        y11_f[j] = -y11_aux.dot(self.A_til[0][1:])
                   + du1_f.dot(self.B[0])
        y12_f[j] = -y12_aux.dot(self.A_til[1][1:])
                   + du2_f.dot(self.B[1])
        y21_f[j] = -y21_aux.dot(self.A_til[2][1:])
                   + du1_f.dot(self.B[2])
        y22_f[j] = -y22_aux.dot(self.A_til[3][1:])
                   + du2_f.dot(self.B[3])
        y11_aux = np.append(y11_f[j], y11_aux[:-1])
        y12_aux = np.append(y12_f[j], y12_aux[:-1])
        y21_aux = np.append(y21_f[j], y21_aux[:-1])
        y22_aux = np.append(y22_f[j], y22_aux[:-1])
    f = np.append(y11_f+y12_f, y21_f+y22_f)
    # Solver Inputs
    H =
    matrix((2*(self.G.T.dot(self.Q).dot(self.G)+self.R)).tolist())
    q =
    matrix((2*self.G.T.dot(self.Q).dot(f-w)).tolist())
    A =
    matrix(np.hstack((np.eye(self.nu*self.m),
                      -1*np.eye(self.nu*self.m))).tolist())
    b =
```

```
        matrix([self.du_max]*self.nu*self.m+
                [-self.du_min]*self.nu*self.m)
        # Solve
        sol = solvers.qp(P=H,q=q,G=A,h=b)
        dup = list(sol['x'])
        s = sol['status']
        j = sol['primal objective']
        return dup, j, s
```

A.3 IHMPC

```
# -*- coding: utf-8 -*-
"""
Created on Wed Apr 26 12:07:03 2017

@author: Igor Yamamoto
"""
import numpy as np
import matplotlib.pyplot as plt
from cvxopt import matrix, solvers
from scipy import signal
from scipy.linalg import block_diag


class OPOM(SystemModel):
    def __init__(self, H, Ts):
        super().__init__(H)
        self.Ts = Ts
        self.na = self._max_order()  # max order of Gij
        self.nd = self.ny*self.nu*self.na
        self.nx = 2*self.ny+self.nd
        self.X = np.zeros(self.nx)
        self.A, self.B,
        self.C, self.D = self._build_state_space()

    def __repr__(self):
        return "A=\n%s\n\nB=\n%s\n\nC=\n%s\n\nD=\n%s"
                        % (self.A.__repr__(),
                           self.B.__repr__(),
                           self.C.__repr__(),
```

```
                                self.D.__repr__())

    def _max_order(self):
        na = 0
        for h in self.H.flatten():
            na_h = len(h.den) - 1
            na = max(na, na_h)
        return na

    def _get_coeff(self, b, a):
        # multiply by 1/s (step)
        a = np.append(a, 0)
        # do partial fraction expansion
        r, p, k = signal.residue(b, a)
        # r: Residues
        # p: Poles
        # k: Coefficients of the direct polynomial term
        d_s = np.array([])
        # d_d = np.array([])
        d_d = np.zeros(self.na)
        d_i = np.array([])
        poles = np.zeros(self.na)
        integrador = 0
        i = 0
        for i in range(np.size(p)):
            if (p[i] == 0):
                if (integrador):
                    d_i = np.append(d_i, r[i])
                else:
                    d_s = np.append(d_s, r[i])
                    integrador += 1
            else:
                d_d[i] = r[i]
                poles[i] = p[i]
                i += 1
        if (d_i.size == 0):
            d_i = np.append(d_i, 0)
        return d_s, d_d, d_i, poles
```

```
    def _create_J(self):
        J = np.zeros((self.nu*self.na, self.nu))
        for col in range(self.nu):
            J[col*self.na:col*self.na+self.na, col] = np.ones(self.na)
        return J

    def _create_psi(self):
        Psi = np.zeros((self.ny, self.nd))
        size = self.nu*self.na
        for row in range(self.ny):
            Psi[row, row*size:row*size+size] = np.ones(size)
        return Psi

    def _build_state_space(self):
        D0 = np.zeros((self.nu))
        Dd = np.array([])
        Di = np.zeros((self.nu))
        R = np.array([])
        for i in range(self.ny):
            d0_x = np.array([])
            di_x = np.array([])
            for j in range(self.nu):
                b = self.H[i][j].num
                a = self.H[i][j].den
                d0, dd, di, r = self._get_coeff(b, a)
                d0_x = np.hstack((d0_x, d0))
                Dd = np.append(Dd, dd)
                di_x = np.hstack((di_x, di))
                R = np.append(R, r)
            D0 = np.vstack((D0, d0_x))
            Di = np.vstack((Di, di_x))
        Dd = np.diag(Dd)
        D0 = D0[1:]
        Di = Di[1:]

        # Define matrices F[ndxnd], J[nu.naxnu], N[ndxnu]
        J = self._create_J()

        F = np.diag(np.exp(R))
```

```
        N = J
        for _ in range(self.ny-1):
            N = np.vstack((N, J))

        a1 = np.hstack((np.eye(self.ny),
                        np.zeros((self.ny, self.nd)),
                        self.Ts*np.eye(self.ny)))
        a2 = np.hstack((np.zeros((self.nd, self.ny)),
                        F,
                        np.zeros((self.nd, self.ny))))
        a3 = np.hstack((np.zeros((self.ny, self.ny)),
                        np.zeros((self.ny, self.nd)),
                        np.eye(self.ny)))
        A = np.vstack((a1, a2, a3))

        B = np.vstack((D0+self.Ts*Di, Dd.dot(F).dot(N), Di))

        def psi(t):
            R2 = np.array(list(map(lambda x: np.exp(x*t), R)))
            psi = np.zeros((self.ny, self.nd))
            for i in range(self.ny):
                phi = np.array([])
                for j in range(self.nu):
                    phi = np.append(phi,
                                    R2[(i*self.nu+j)*self.na:
                                    (i*self.nu + j + 1)*self.na])
                psi[i, i*self.nu*self.na:(i+1)*self.nu*self.na] = phi
            return psi

        def C(t):
            return np.hstack((np.eye(self.ny),
                              psi(t),
                              np.eye(self.ny)*t))

        D = np.zeros((self.ny, self.nu))
        return A, B, C, D, D0, Di, Dd, J, F, N, psi, R

    def output(self, du1, du2, samples):
```

```
        U = np.vstack((du1, du2)).T
        X = np.zeros((samples+1, self.nx))
        X[0] = self.X
        Y = np.zeros((samples+1, self.ny))
        Y[0] = self.C(0).dot(X[0])
        for k in range(samples):
            X[k+1] = self.A.dot(X[k]) + self.B.dot(U[k])
            Y[k+1] = self.C(0).dot(X[k+1])

        self.X = X[samples]
        return X, Y


class IHMPCController(Controller):
    def __init__(self, H, Ts, m, Q, R, du_min, du_max):
        super().__init__(H, Ts, m, Q, R, du_min, du_max)
        self.model = OPOM(H, Ts)
        self.internal_matrices = self._create_matrices()

    def _crreate_matrices(self):
        def create_D0n_ou_Di1n(D, n, m):
            D_n = D
            for i in range(m-1):
                if i >= n-1:
                    d_n = np.zeros(D.shape)
                else:
                    d_n = D
                D_n = np.concatenate((D_n, d_n), axis=1)
            return D_n

        def create_Di2n(D, n, m):
            D_n = np.zeros(D.shape)
            for i in range(1, m):
                if i > n:
                    d_n = np.zeros(D.shape)
                else:
                    d_n = i*Ts*D
                D_n = np.concatenate((D_n, d_n), axis=1)
            return D_n
```

```
def create_Wn(F, n, m):
    Wn = np.eye(F.shape[0])
    for i in range(1, m):
        if i > n-1:
            wn = np.zeros(F.shape)
        else:
            f = np.diag(F)
            wn = np.diag(f ** (-i))

        Wn = np.concatenate((Wn, wn), axis=1)
    return Wn

z = self.Dd.dot(self.N)
Z = z
for i in range(self.m - 1):
    Z = block_diag(Z, z)

D0_n = []
for i in range(1, self.m + 1):
    D0_n.append(faz_D0n_ou_Di1n(self.D0, i, self.m))

Di_1n = []
for i in range(1, self.m + 1):
    Di_1n.append(faz_D0n_ou_Di1n(self.Di, i, self.m))

Di_2n = []
for i in range(m):
    Di_2n.append(faz_Di2n(self.Di, i, self.m))

Wn = []
for i in range(1, self.m + 1):
    Wn.append(faz_Wn(self.F, i, self.m))

Di_1m = self.Di
for _ in range(m-1):
    Di_1m = np.hstack((Di_1m, self.Di))

Di_3m = self.m*self.Ts*Di_1m-Di_2n[self.m-1]
```

```
        D0_m = D0_n[self.m-1]
        Di_1m = Di_1n[self.m-1]

        Aeq = np.vstack((D0_m+Di_3m, Di_1m))

        return [Z, D0_n, Di_1n, Di_2n, Wn, Aeq]

    def calculate_control(self):
        def G1(n):
            G = np.zeros((self.ny, self.nd))
            for i in range(self.ny):
                phi = np.array([])
                for j in range(self.nu):
                    r = self.R[i*self.nu+j]
                    if r == 0:
                        g = n
                    else:
                        g = 1/r*(np.exp(r*n)-1)
                    phi = np.append(phi, g)
                G[i, i*self.nu*self.na:(i+1)*self.nu*self.na] = phi
            return G

        def G2(n):
            G = np.array([])
            for y in range(self.ny):
                r = self.R[y*self.nu:y*self.nu+self.nu]
                g = np.zeros((self.nu, self.nu))
                for i in range(self.nu):
                    for j in range(self.nu):
                        gzin = r[i] + r[j]
                        if gzin == 0:
                            g[i, j] = 0
                        else:
                            g[i, j] = 1/gzin*(np.exp(gzin*n)-1)
                G = block_diag(G, g)
            return G[1:]

        def G3(n):
            G = np.zeros((self.ny, self.nd))
```

```
        for i in range(self.ny):
            phi = np.array([])
            for j in range(self.nu):
                r = self.R[i*self.nu+j]
                phi = np.append(phi, r)
            if 0 in phi:
                def aux(x):
                    if x == 0:
                        return n
                    else:
                        return (1/x**2)*np.exp(x*n)*(x*n-1)
                phi = np.array(list(map(aux, phi)))
            else:
                phi = np.zeros(self.nu)
            G[i, i*self.nu*self.na:(i+1)*self.nu*self.na] = phi
        return G

    H_m = 0
    for n in range(self.m):
        a = self.Z.T.dot(self.Wn[n].T)
            .dot(G2(n)-G2(n-1))
            .dot(self.Wn[n]).dot(self.Z)
        b1 = (G1(n) - G1(n-1)).T
            .dot(self.Q)
            .dot(self.D0_n[n]-self.Di_2n[n])
        b2 = (G3(n)-G3(n-1)).T
            .dot(self.Q)
            .dot(self.Di_1n[n])
        b3 = (G1(n)-G1(n-1)).T
            .dot(self.Q)
            .dot(self.Di_1n[n])
        b = 2*self.Z.T.dot(self.Wn[n].T).dot(b1+b2+b3)
        c1 = Ts*(self.D0_n[n]-self.Di_2n[n]).T
            .dot(self.Q)
            .dot(self.D0_n[n]-self.Di_2n[n])
        c2 = 2*(n-1/2)*Ts**2*(self.D0_n[n]-self.Di_2n[n]).T
            .dot(self.Q)
            .dot(self.Di_1n[n])
        c3 = (n**3-n+1/3)*Ts**3*self.Di_1n[n].T
```

```
            .dot(self.Q)
            .dot(self.Di_1n[n])
        c = c1 + c2 + c3
        H_m += a + b + c

    H_inf = self.Z.T
        .dot(self.Wn[m-1].T)
        .dot(G2(float('inf'))-G2(m))
        .dot(self.Wn[m-1])
        .dot(self.Z)

    H = H_m + H_inf

    e_s = np.array([1 - 2.292925, 1 - 0.3075793])

    x_d = np.array([0, -1.39258457, 0.64501061, 0])
    x_i = np.array([-0.133836, -0.165534])

    cf_m = 0
    for n in range(self.m):
        a = (-e_s.T.dot(self.Q)
            .dot(G1(n)-G1(n-1)) +
            x_d.T.dot(G2(n)-G2(n-1)) +
            x_i.T.dot(self.Q).dot(G3(n)-G3(n-1)))
            .dot(self.Wn[n]).dot(self.Z)
        b = (-Ts*e_s.T +
            (n-1/2)*Ts**2*x_i.T +
            x_d.T.dot((G1(n) - G1(n-1)).T))
            .dot(self.Q)
            .dot(self.D0_n[n]-self.Di_2n[n])
        c = (-(n-1/2)*Ts**2*e_s.T +
            (n**3-n+1/3)*Ts**3*x_i.T +
            x_d.T.dot((G3(n)-G3(n-1)).T))
            .dot(self.Q).dot(self.Di_1n[n])
        cf_m += a + b + c

    cf_inf = x_d.T
        .dot(G2(float('inf'))-G2(self.m))
        .dot(self.Wn[m-1])
```

```
            .dot(self.Z)

        cf = cf_m + cf_inf
        beq = np.hstack((e_s - self.m*self.Ts*x_i, -x_i)).T
        sol = solvers.qp(P=matrix(H),
                         q=matrix(cf),
                         A=matrix(self.Aeq),
                         b=matrix(beq))
        du = list(sol['x'])
        s = sol['status']
        j = sol['primal objective']
        return du
```

A.4 Example of Use

```
# Example of Script to Run a Simulation

# Create Transfer Matrix
h11 = signal.TransferFunction([-0.19],[1, 0])
h12 = signal.TransferFunction([-1.7],[19.5, 1])
h21 = signal.TransferFunction([-0.763],[31.8, 1])
h22 = signal.TransferFunction([0.235],[1, 0])
H = [[h11, h12], [h21, h22]]
# Horizons
p = 10
m = 3
# Weights
Q = 1*np.eye(p*ny)
R = 10*np.eye(m*nu)
# Constraints
du_max = 0.2
du_min = -0.2
# Sample Time
Ts = 1
controller = GPCController(H, Ts, p, m, Q, R, du_min, du_max)
# Create Simulation Environment
tsim = 100
sim = Simulation(controller, real_model)
sim.run(tsim)
sim.show_results()
```