Title: Framework in PYOMO for the assessment and implementation of (as)NMPC controllers

Author: Federico Lozano Santamaría Jorge M. Gómez

PII: S0098-1354(16)30153-3
DOI: http://dx.doi.org/doi:10.1016/j.compchemeng.2016.05.005
Reference: CACE 5465

To appear in: *Computers and Chemical Engineering*

Received date: 22-1-2016
Revised date: 5-5-2016
Accepted date: 9-5-2016

Please cite this article as: Santamaría, Federico Lozano., & Gómez, Jorge M., Framework in PYOMO for the assessment and implementation of (as)NMPC controllers.*Computers and Chemical Engineering* http://dx.doi.org/10.1016/j.compchemeng.2016.05.005

Framework in PYOMO for the assessment and implementation of (as)NMPC controllers.

Federico Lozano Santamaría and Jorge M. Gómez[*].

Grupo de Diseño de Productos y Procesos, Departamento de Ingeniería Química, Universidad de los Andes.

Carrera 1 No. 18A-10, Bogotá, Colombia.

[*] To whom correspondence should be addressed. E-mail: jorgomez@uniandes.edu.co

Highlights:

This paper presents a software framework developed in Pyomo, a mathematical modelling language embedded in Python, for the assessment and implementation of ideal nonlinear MPC and advanced step nonlinear MPC. The framework automates many of the aspects of MPC defining new classes in Pyomo for ideal NMPC (iNMPC) and advanced step NMPC (asNMPC). The user only has to define the prediction model of the process using new classes for manipulated variables, disturbances and initial conditions, and the real plant function to access to the states of the process in a similar way of other algebraic modelling languages. The model discretizaion, controller set-up, receding horizon and solution are done automatically. Three examples are presented in detail for explaining the use and advantages of the framework to evaluate iNMPC and asNMPC controllers. The software is freely available upon request and in the future it is expected to be an official extension of Pyomo.

Key words

asNMPC, Model predictive control, Pyomo, NLP sensitivity, Dynamic optimization, Python.

Abstract

Model predictive control (MPC) is an advanced control strategy that has a growing interest for research and applications because of its good performance in many kind of processes and its ability to handle constraints, perform optimization, and consider economic aspects and nonlinearities of the process. However, its design, evaluation and implementation require a high level of expertise which might restrict the developments in this area. This paper presents a software framework developed in Pyomo, a mathematical modelling language embedded in Python, for the assessment and implementation of ideal nonlinear MPC and advanced step nonlinear MPC. The framework automates many of the aspects of MPC defining new classes in Pyomo for ideal NMPC (iNMPC) and advanced step NMPC (asNMPC). The user only has to define the prediction model of the process using new classes for manipulated variables, disturbances and initial conditions, and the real plant function to access to the states of the process in a similar way of other algebraic modelling languages. The model discretizaion, controller set-up, receding horizon and solution are done automatically. Three examples are presented in detail for explaining the use and advantages of the framework to evaluate iNMPC and asNMPC controllers. The software is freely available upon request and in the future it is expected to be an official extension of Pyomo.

1. Introduction

Model predictive control (MPC) and non-linear model predictive control (NMPC) are advanced control strategies that solve an optimal control problem (OCP) in a receding horizon scheme to define the feedback value of the manipulated variables for each sampling interval. This control strategy has been widely used in the chemical process industry which systems exhibit slow dynamics. However, the advantages of this control strategy (e.g. it is capable of handling the MIMO problem directly, it considers inequality constraints and it is based on dynamic optimization) and the current developments in computing and algorithms have expanded the application of this kind of controllers to many systems and processes, even those with fast dynamics [1,2]. These increasing number of applications of (N)MPC and their success have motivated the research in this field and some specific areas that shown interest are: the development of algorithms to perform rapid optimization [3,4], the development of better modelling strategies using more accurate and simple models [5,6], and new alternatives and variations of this control strategy that improve the closed-loop performance or reduce the computational time required for solving the optimization problem (e.g. adaptive MPC [7] and explicit MPC [8]). Even with all these advances in the field the implementation of any (N)MPC requires a high level of expertise.

A typical implementation of an (N)MPC controller requires seven steps. 1) Define a dynamic model that represents accurately enough the process. 2) Define an objective function for the controller and it is common to define a tracking quadratic function. 3) Discretize the model so the OCP can be solved using QP or NLP algorithms which can exploit the structure of the model. 4) Read the current states of the system and process the information. 5) Solve the OCP for the

current time interval. 6) Process the OCP solution and define the value of the manipulated variables that are feedback to the process. 7) Move to the next sampling period. The step 5 is critical for the closed loop performance of the controller because it can introduce a significant delay produced by the time required for solving the dynamic optimization problem, especially when a large-scale nonlinear model is used for prediction. One alternative for reducing this time delay and improve the controller performance is to use NLP sensitivity to update online and fast the values of the manipulated variables. This integration of a NMPC with a fast estimation of the optimal solution of the OCP based on small parametric changes of the problem is called the advanced step NMPC (asNMPC) and its nominal stability properties and robustness have been proven [9]. The NLP sensitivity theory is a powerful tool that can provide fast and accurate estimation of the solution of a nonlinear optimization problem when the parameters of the problem are perturbed in a small region. In addition, it can also be used in the state estimation problem (step 4) for obtaining fast solutions online when a moving horizon estimator (MHE) is implemented [10]. The importance of NLP sensitivity in the whole NMPC scheme makes it necessary to consider within a general framework for designing this kind of controllers.

Most of the steps mentioned can be automated for any variation of (N)MPC controller so the analyst only cares for a good definition of the dynamic model and the level of expertise required might be reduced. Thus the design, assess, and implementation of this kind of controllers can be expanded and done in an easier way. The Model Predictive Toolbox of Matlab [11] is a program that is currently available for doing this. Using this toolbox the user can easily define the controller specifying only the sampling time, prediction horizon, control horizon, input-output constraints and penalization weights. It is also compatible with all the tools available in Simulink, so it can handles disturbances, white noise and set-point changes. However, this

toolbox is only available for linear models, even when the real process can be non-linear it does a linearization at a specific point of operation. Some of the limitations of the Matlab Model Predictive Control Toolbox are: it assumes a standard design of MPC in which the objective function is a quadratic tracking function, the inequality constraints are bounds on the states or manipulated variables, it does not admit general objective functions such as economic functions or other inequalities that can be relations between the states of the process, and it assumes that the solution time of the OCP is negligible or that the dynamic optimization problem is solved instantaneously. The MPC toolbox of Matlab is useful for many applications but it is restricted to the standard MPC. The purpose of this paper is to describe a framework for the design, evaluation and implementation of MPC and NMPC controllers that is not restricted to a specific scheme so it can handle non-linear models and any kind of objective function and inequality constraints. The framework must also be capable of handling the computational delay introduced by the solution of the OCP using NLP sensitivity. We expect that this framework facilitates future research in this area and the implementation of NMPC controllers.

The NMPC framework is developed in Pyomo, an open source-tool for modelling optimization problems and applications in Python. The definition of models for optimization in Pyomo is very similar to current Algebraic Programming Languages (AML), but this platform has the advantage of using many third-party libraries from Python and all the Python data structures to define them giving Pyomo models more flexibility, extensibility and maintainability [12]. In addition, Pyomo has been developed in a modular structure so the users can easily access to different parts of the software or customize them if necessary without the risk of altering the core functionalities. The users can also customize modelling components or the optimization process according to their needs for certain applications [13]. An important example of Pyomo

capabilities to support new developments is its DAE module that allows a new level of abstraction in the modelling of dynamic optimization problems. Using this module the user does not have to discretize manually the differential equations because it is done automatically providing access to advanced discretization schemes and avoiding the human error in the implementation of a discretization [14]. All these important advantages of Pyomo make it attractive for developing a framework for NMPC controllers in which the user only defines the dynamic model of the process and the rest is done automatically.

The remainder of the paper is organized as follows: Section 2 presents an overview of the design goals and requirements of the framework proposed to evaluate and implement NMPC controllers, Section 3 describes the algorithms implemented in the framework and the mathematical background of ideal NMPC (iNMPC) controllers and advanced step NMPC (asNMPC) controllers, Section 4 presents the application of the framework in three examples showing how this simplifies the implementation of the NMPC and how this new module can handle non-standard formulation of the OCP. Finally, Section 5 draws the conclusion of the paper and future challenges.

2. Design goals and requirements

The main purpose of this framework for NMPC controllers is to avoid complex connections between different programs and to reduce the level of expertise required for their implementation and evaluation. Researchers and engineers working in this area can take advantage of this for future developments.

The goals of the framework presented in this paper are described next along with some of the software requirements for this to work properly.

- *Open-source:* even though there are many open-source optimization solvers, for example those found in the package COIN-OR, there are only few open-source programs for developing optimization models and even less for doing (N)MPC. The framework presented in this paper is embedded in an open-source modelling language and it is developed as an extension of it, naturally it is also open-source. The bigger motivation for making it open-source is that the future users can modify and customize the code according to their needs and we can also receive some feedback from them regarding the performance or debugging of the core functionalities of the framework.
- *Avoid communication between programs:* usually the implementation of a (N)MPC control scheme requires a complex interaction between different programs each one serving a specific purpose. Figure 1 shows an example of how this can be implemented. Note that there is an interaction of three different programming languages where a high level language is used to communicate the different layers and to post and preprocess the information available. The flow of information is complex and in some cases might present compatibility restrictions depending on the programs. The framework presented in this paper integrates all the different layers involved in the implementation of (N)MPC controllers in Python which, together with Pyomo, allows to perform all the actions required. Using only Python and its extensions it is possible to integrate all the layers of (N)MPC controllers in a single handler function without calling additional programming languages. In addition Python has many third-party libraries which facilitates the implementation and testing of different functions required in the set-up of a (N)MPC controller.

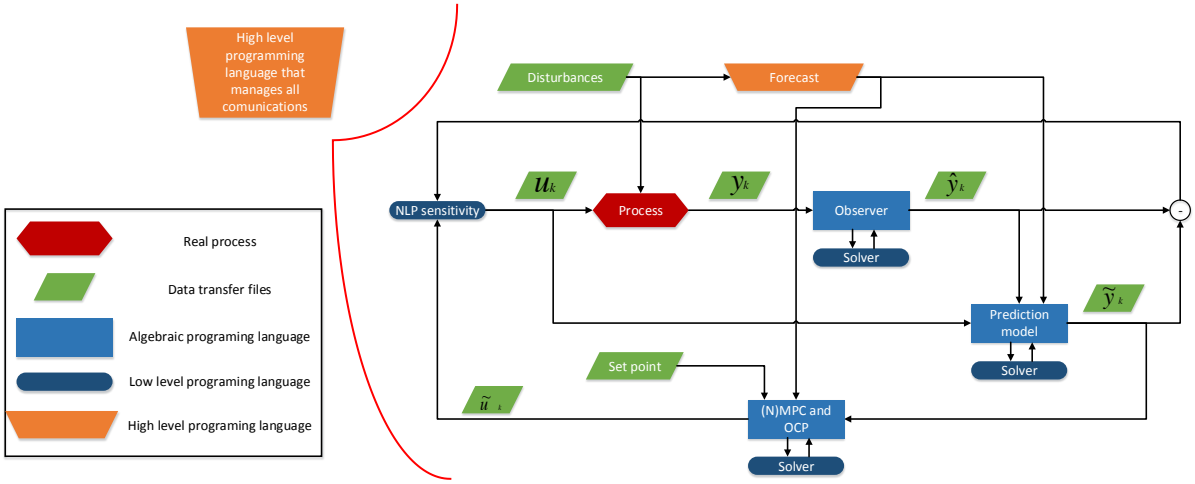


Figure 1. Representative example of an implementation of a (N)MPC controller.

Currently the (N)MPC framework is working with Pyomo 4.1.10159 (http://www.pyomo.org/) in Python 2.7 and it uses the libraries NumPy (http://www.numpy.org/), SciPy (http://www.scipy.org/) and the pseudo solver gjh (http://ampl.com/resources/hooking-your-solver-to-ampl/) for the sensitivity calculations in asNMPC (Section 3.2).

- *Automatation of the controller set-up:* the main idea behind this framework is to reduce the level of expertise required in the evaluation and implementation of (N)MPC controllers, therefore it has to automatically set-up the controller after the user defines some simple instructions. It is expected for the user to know about defining models in AMLs so he/she can define the dynamic model used in the OCP within Pyomo. This is the main input of the framework and afterwards the additional steps in the (N)MPC implementation are done automatically.

In order to facilitates the definition of dynamic models and their direct transcriptions for optimization the framework makes an extensive use of the DAE module of Pyomo (*pyomo.dae*). This module allows the user to incorporate easily differential equations within the model and thus the optimization problem can be defined as a set of differential equations (ODE) or

differential-algebraic equations (DAE). The *pyomo.dae* module includes important modelling classes such as: *ContinuousSet* to represent the continuous domains of the variables (e.g. time, spatial coordinates) and *DerivativeVar* which defines the derivatives of the variables with respect to the independent variable defined in a *ContinousSet* and includes them in the model. Another important aspect of the *pyomo.dae* module, is its ability to transform a dynamic model defined in a continuous time representation to a model in a discrete time representation, which is equivalent to an algebraic model. For doing so, it uses a simultaneous discretization approach and the currently discretization schemes available in this module are: backward finite differences, forward finite differences, central finite differences, orthogonal collocation in finite elements with Radau roots or Legendre roots [15]. The user can easily change the discretization scheme and once the model has been discretized it is ready to be sent to the solver.

In summary, the proposed NMPC framework discretizes the whole model after a discretization scheme has been selected, it defines the control horizon constraints, it defines the manipulated variables as piece-wise functions for each sampling time, it does sensitivity calculations to correct model plant mismatch if necessary, it communicates directly with the solver to pass or extract the results and it presents the results dynamically for a better visualization and testing.

- *Compatibility with different models:* one of the limitations of the current toolboxes available for (N)MPC is their compatibility with different models, some of them are restricted to linear models [11] or to models related with a specific application [16]. In addition, the user cannot customize these models easily because it is necessary to access the source code of the toolboxes. For some processes it is necessary to use non-linear models to describe their dynamic behavior accurately or the (N)MPC formulation might include non-standard elements and this level of flexibility is allowed in the our framework for (N)MPC. Using this framework the user

can define any kind of model as long as it is stated as a mathematical program. For instance, it can include non-linear equality and inequality constraints, binary variables or economic objective functions. The only requirement is for the user to have installed an adequate solver for the type of model that is solving, for example a QP solver for linear MPC, a NLP solver for NMPC and a MIP solver for linear MPC with binary variables. Additionally, Pyomo can be integrated with any optimization solver via tightly coupled modelling tools or loosely coupled modelling tools allowing it to use a variety of solvers independently of their interfaces [14].

3. Description of the framework for iNMPC/asNMPC in Pyomo

The development of the software framework aims to provide a free tool for the assessment, design and implementation of (N)MPC controllers. It is an extension of Pyomo, so it can be used in any Python script by importing the module and it also inherits all the functions of Pyomo. In this section we describe the (N)MPC problem, the advanced step controller and the framework developed to address these problems.

3.1. Background of ideal NMPC (iNMPC)

Figure 2 presents a simple example of how model predictive control works. At the current time (t_k) an optimal control problem (OCP) that uses the model representation of the process is solved online in a future prediction horizon (N_p) where only a number of control actions are allowed (N_u). The solution of the optimization problem is the prediction of the future behavior of the process under the current circumstances and the future control actions necessary to achieve the desired operation. However, only the first value of the manipulated variables is feedback to the process and the rest future actions predicted are discarded.

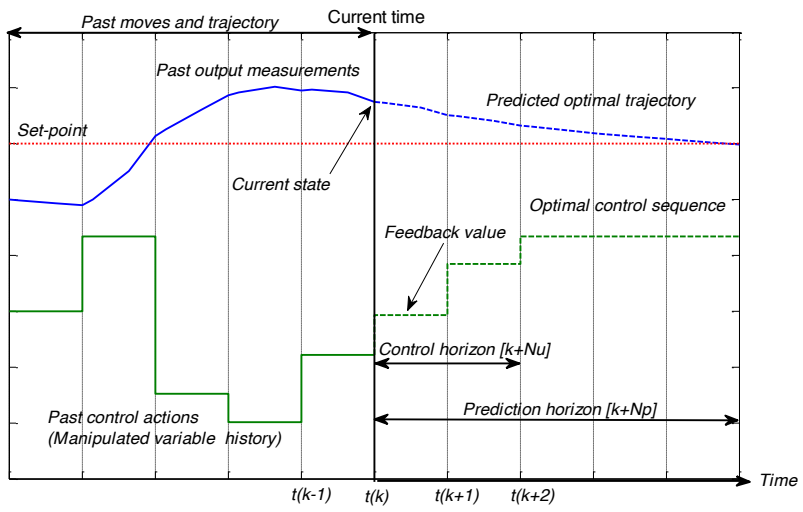


Figure 2. Representation of the receding horizon strategy in (N)MPC

The critical step in the implementation of (N)MPC controllers is the solution of the OCP because it has to be solved periodically and in less time than a sampling time interval. An OCP given by Eq. (1), where x are the differential states, y the algebraic states, w the disturbances and u the manipulated variables, has to be solved each time the controller is asked to determine the new values of the manipulated variables. The objective function Eq. (1a) is composed by two terms: a running cost (ϕ) which is usually associated with a quadratic tracking function, but it can also be an energy or economic function, and a terminal cost (ϕ_f) that is only a function of the states at the final time of the prediction horizon. The optimization problem is subject to a set of differential and algebraic equations Eqs. (1b) – (1c) representing the dynamic model for the evolution of the process. This set of equations can be linear defining a quadratic programming problem (QP), or nonlinear defining a nonlinear programming problem (NLP). The inequalities of the problem Eq. (1e) are normally associated with variable bounds or bounds on their rate of change, but they can also be any nonlinear expressions.

$$\min \int_{t_k}^{t_k+\tau N_p} \phi(t,x,y,u)dt + \phi_f(x_f,y_f,u_f) \tag{1a}$$

$$s.t.\ \frac{dx}{dt} = f(t,\ x,y,u,w) \tag{1b}$$

$$x(t_k) = x_0 \tag{1c}$$

$$h(t,\ x,\ y,u,w) = 0 \tag{1d}$$

$$g(t,\ x,\ y,u,w) \geq 0 \tag{1e}$$

The OCP problem as it is defined in Eq. (1) cannot be solved directly in its continuous time representation. Using discretization techniques such as finite differences or orthogonal collocation the OCP problem can be transformed in a discrete time representation and the optimization problem is equivalent to a large-scale NLP. This direct transcription is usually done using the sampling time as the time step and the Eq. (2) represents the OCP problem in discrete time.

$$\min \sum_{i=0}^{N_p} \phi(t_i,x_i,y_i,u_i) + \phi_f\left(x_{N_p},y_{N_p},u_{N_p}\right) \tag{2a}$$

$$s.t.\ x_{i+1} = f(t_i,x_i,y_i,u_i,w_i),\ \ \forall\, i \in \{0,1,\dots,N_p\} \tag{2b}$$

$$x_{i=0} = x_0 \tag{2c}$$

$$h(t_i,x_i,y_i,u_i,w_i) = 0,\ \ \forall\, i \in \{0,1,\dots,N_p\} \tag{2d}$$

$$g(t_i,x_i,y_i,u_i,w_i) \geq 0,\ \ \forall\, i \in \{0,1,\dots,N_p\} \tag{2e}$$

If the optimization problem given by Eq. (2) is of large-scale and/or has complex nonlinear terms its solution might take an important amount of time introducing delays and even instabilities in the closed loop control. The ideal NMPC (iNMPC) assumes that the OCP can be solved instantaneously and that all the states are measured, thus our framework is based in these two assumptions for iNMPC and it is useful for testing and evaluating different controller configurations but not for real implementations. The advanced step NMPC (asNMPC) is also included in this framework to make it compatible with real implementations because it handles the model-plant mismatch and the delays introduced by the solution of the OCP. Using this module the user can define a state estimator if necessary.

3.2. *Advanced step NMPC (asNMPC)*

The idea of asNMPC is to solve the OCP problem in advance, which means that at the current time t_k and with the measurements of the current states and manipulated variables, the prediction model is used to predict the states at the next time period t_{k+1} and the OCP for this period is solved between t_k and t_{k+1}. Using this strategy the delays generated due to the computational time are avoided, but there can be still a mismatch between the prediction of the states and the measured values at t_{k+1}, thus a rapid correction using NLP sensitivity has to be done. For a more detailed discussion about asNMPC controller, how it works and its stability properties, the reader is referred to [17,18]. Figure 3 illustrates how the asNMPC strategy works, when the OCP problem is solved and when it is necessary to do a NLP sensitivity calculation.

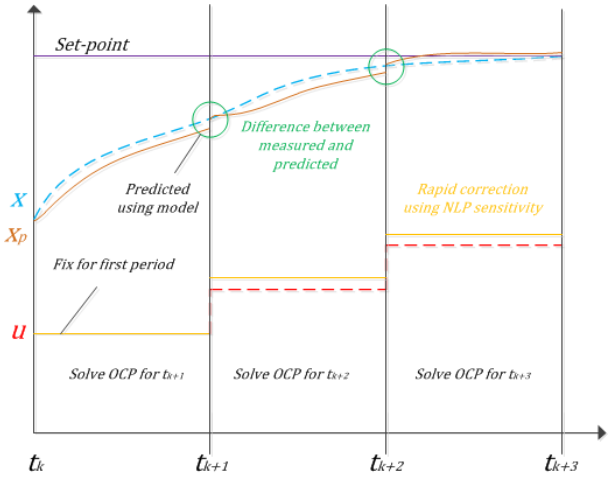


Figure 3. Representation of asNMPC with NLP sensitivity corrections.

Note that at each sampling instant the OCP given by Eq. (2) is very similar to the previous one, the only parameters that change are the initial condition of the states (x_0) and the value of the disturbances (w). This provides two advantages: the first one is that once an OCP is solved its solution is a good starting point for the next OCP facilitating the convergence of the problem, and the second one is that there is a clear identification of the parameters that change in the

definition of the OCP for any NMPC strategy. In addition, within the NMPC framework we introduce two new classes for defining these parameters: the *InitialCondition* and the *Disturbance* classes. Therefore, these parameters are easy to track every time the OCP is solved and the NLP sensitivity calculation can be automated for these controllers.

If the parameters (p) of an optimization problem are associated with artificial constraints of the form $x_p - p = 0$ where x_p is a variable used to storage the value of the parameters, the NLP sensitivity calculations can be reduced to solve the linear system given by Eq. (3) [19]. In this linear equation W is the Hessian matrix of the Lagrange function (L), A is the Jacobian matrix of the equality constraints ($c = 0$), V is a diagonal matrix which contains the dual variables (v), Z is a diagonal matrix which contains all the variables of the problem and λ are the Lagrange multipliers for the equality constraints. Note that this equation is valid when the problem is reformulated for solving it using an interior point strategy (IPOPT) and also that the left hand side matrix corresponds to the KKT matrix at the optimal solution which is available after solving the nominal optimization problem. Additionally, the artificial constraints and variables can be automatically defined within the NMPC framework when the classes *InitialCondition* and *Disturbance* are used to declare these elements of the model.

$$\begin{bmatrix} W & \nabla_{zx_p} L(z, x_p, \lambda, v) & A & -I & 0 \\ \nabla_{x_p z} L(z, x_p, \lambda, v) & \nabla_{x_p x_p} L(z, x_p, \lambda, v) & \nabla_{x_p} c(z, x_p) & 0 & I \\ A^T & \nabla_{x_p} c(z, x_p)^T & 0 & 0 & 0 \\ V & 0 & 0 & Z & 0 \\ 0 & I & 0 & 0 & 0 \end{bmatrix} \begin{bmatrix} \Delta z \\ \Delta x_p \\ \Delta \lambda \\ \Delta v \\ \Delta \bar{\lambda} \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ 0 \\ 0 \\ \Delta p \end{bmatrix} \quad (3)$$

Injecting the updated manipulated variables using NLP sensitivity to the process may causes the active set of constraints to change, which means that inactive inequality constraints are violated in practice and active inequality constraints may become inactive. These changes of active sets, if not handled properly, affect the accuracy and stability of the controller. A simple

and easy approach to handle active set changes during sensitivity calculation is clipping in the first interval (CFI) which basically updates the value of the manipulated variables with only a fraction of the sensitivity step, so the active set of the constraints does not change [20]. CFI only deals with the feasibility of the current manipulated variables and the first states predicted. The fraction of the sensitivity step (τ) used to update the manipulated variables can be found solving the linear optimization problem stated in Eq. (4). This problem only has one variable which is τ and the rest are parameters which are obtained from the solution of the OCP at the current time. In other words, u_0 and z_1 are the values obtained for the manipulated and state variables after solving the OCP for a specific sampling instant and Δu_0 and Δz_1 are the results of the sensitivity step calculation.

$$\max \tau \tag{4a}$$
$$s.t. \; u_{lb} \leq u_0 + \tau \Delta u_0 \leq u_{ub} \tag{4b}$$
$$z_{lb} \leq z_1 + \tau \Delta z_1 \leq z_{ub} \tag{4c}$$
$$0 \leq \tau \leq 1 \tag{4d}$$

This framework uses NLP/QP strategies for solving the optimization problem and for large-scale problems these alternatives are particularly efficient. In other words, the differential equations and all the variables have to be fully discretized so the problem can be formulated as a large-scale NLP. However, the discretization can be done automatically using the DAE module from Pyomo and it also makes the problem compatible for using NLP sensitivity.

3.3. Framework overview

For using this framework within dynamic models it is necessary for the user to call new classes that are not standard from Pyomo and that are developed exclusively to make the implementation of iNMPC/asNMPC controllers easy and intuitive. Figure 4 presents the general structure of the NMPC module, the new classes created for Pyomo and how they interact with each other. The classes *ManipulatedVar*, *Disturbance*, and *InitialCondition* are created especially for this

framework and the user must use them in the same nature of other Pyomo objects (e.g. *Var*, *Param*) to define the manipulated variables, the disturbances and the initial conditions of a differential variable, respectively. The class *DerivativeVar* is inherited from the DAE module and it is used to define the derivative variables of the model (e.g. *dx/dt*). The *ManipulatedVar* class inherits the arguments and key words from the Pyomo *Var* class. The *Disturbance* and *InitialCondition* classes inherit the key words from the Pyomo *Param* class with the exception that both of them only receives one argument, for *Disturbance* is a continuous set that represents the time of the model and for *InitialCondition* is the corresponding differential variable.

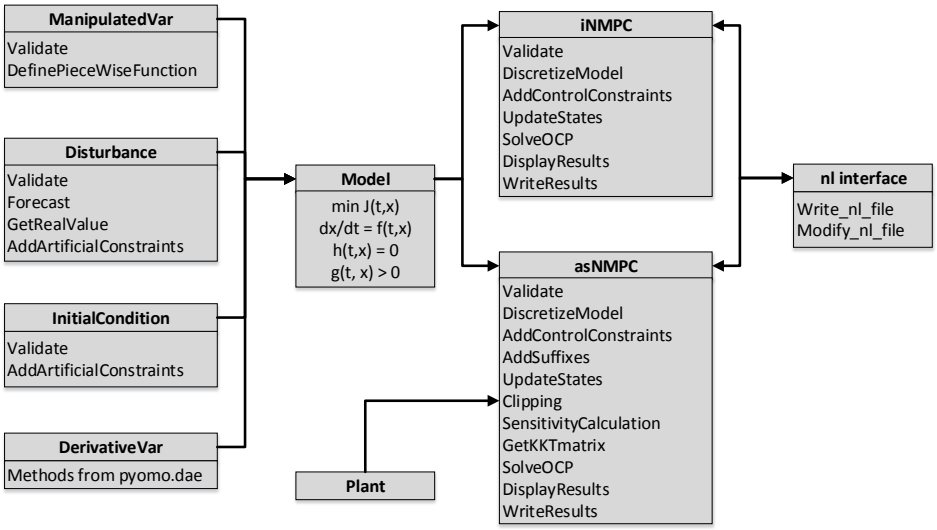


Figure 4. Structure overview of the NMPC framework

After defining the model using these new classes the user can create an object of the class *iNMPC* or *asNMPC* and then call the method *solve* with the adequate arguments. These two classes have the core functionalities of the framework and automate the whole implementation of NMPC controllers. For example, creating an object of the class *iNMPC* or *asNMPC* the model is discretized automatically, the control constraints to fix the manipulated variables after the control horizon are added to the model, the states are updated automatically after each sampling period, and the results are displayed dynamically and written to a text file. The *asNMPC* class has

additional methods which includes: a method for adding suffixes to the model that avoids the elimination of the artificial constraints by preprocessing instructions and allows the storage of the sensitivity results; a method for getting and saving the KKT matrix at the optimal solution; a method for doing NLP sensitivity calculations solving the linear problem defined in Eq. (3), and a method for doing CFI after the sensitivity calculations according to Eq. (4) so active set changes can be handled.

Note that the *asNMPC* class receives the inputs from a generic function *Plant*. This function must return the states of the process at the current time and it has a flexible structure, so it could be a mathematical model, a call to another software or real measurements of the process. In addition, this framework does not handle the state estimation problem, but the user can define a state estimator within the *Plant* function to address this issue.

Finally, there is an additional class called *nl_interface* that interacts with the *iNMPC* and *asNMPC* classes and it is used to generate and modify a ".nl" file of the optimization problem. ".nl" files are text files used to communicate an AML with a solver and they contain the whole representation of the optimization problem including variables, constraints, objectives, suffixes, expression tress and Jacobians, in a codified structure [21]. Each ".nl" begins with a header that gives the problem statistics such as: number of continuous variables, number of discrete variables, number of constraints, number of objectives, number of logical constraints and nonzero elements in the Jacobian and gradients. The header is followed by various segments identified by letters, some of them are: V for variables definition, C for constraint definition using expression graphs, O for objective functions definition using expression graphs, S for suffixes values, x for primal variables initial guess, d for dual variables initial guess, b for bounds on the variable, r for bounds on algebraic constraints, J for the definition of the Jacobian

and G for the definition of the gradient [21]. For a more detail description of ".nl" files, its parts and how to create or edit them the reader is referred to [21].

This *nl_interface* class is necessary in the implementation of NMPC controllers because Pyomo is not efficient writing this type of files for large-scale problems, it might take more than 150 seconds for writing the ".nl" file of a problem with 40000 elements (variables and constraints) [13], and this delay added to the solution of the OCP is prohibitive. Using the *nl_interface* class this framework avoid this additional delay writing the ".nl" file only once when a NMPC object is declared and modifying it at each sampling interval. This can be done relatively easily because the OCP problem does not change between sampling times, only the initial conditions and disturbances change and by identifying those elements in the ".nl" file the update of the OCP formulation can be done instantaneously.

4. Examples and discussion

This section presents three examples of iNMPC and asNMPC controllers to exhibits the use and advantages of the framework. All the examples are solved using Pyomo 4.1.10159 in an Intel Core i5 2.7 GHz, 8.0 GB memory.

4.1. Example 1. Car position problem

This is an example adapted from [22] in which a NMPC controller is used to control the position of a car over a road using its velocity as manipulated variable. For this example we consider an iNMPC controller without model-plant mismatch.

The formulation of the OCP for this example is given by Eq. (5). The starting position is $x = 0, y = 0$ and the final position is $x = 0, y = 1$, also after 150 seconds the car should return to the initial position. This last condition is modelled as a set-point change. The control objective

is reflected in the objective function, Eq. (5a), which is a quadratic function of tracking penalizing big values of the velocity components. Eq. (5b) – (5c) represents the variation of the position in x and y according to the directional component of the velocity, Eq. (5d) is a constraint over the movement of the car that must be restricted to an ellipse, and Eq. (5e) is an upper bound for the magnitude of the velocity.

$$\min \int_{t_k}^{t_k+\tau N_p} \left[\left(x - x_{sp}\right)^2 + 5\left(y - y_{sp}\right)^2 + v_x^2 + v_y^2 \right] dt \tag{5a}$$

$$s.t. \frac{dx}{dt} = v_x, \qquad x(t_k) = x_0 \tag{5b}$$

$$\frac{dy}{dt} = v_y, \qquad y(t_k) = y_0 \tag{5c}$$

$$x^2 - 4(y - 0.5)^2 = 1 \tag{5d}$$

$$v_x^2 + v_y^2 \leq 0.02^2 \tag{5e}$$

In this problem the controlled variable is the position of the car (x, y) and the manipulated variable is the velocity vector (v_x, v_y). Moreover, Eq (5b) and Eq (5c) represent the rate of change of the car position in the x and y coordinate, respectively, Eq (5d) represents the rail that the car must follow and Eq (5e) is the maximum velocity, represented as the norm of the velocity vector, allowed for the system.

For this problem we use a sampling time of one second, a control horizon of 4 and a prediction horizon of 4. Also, the differential equations are discretized using the backward finite differences method, so the integral in the objective function is transform into a summation over all the prediction horizon and the differential equations into algebraic constraints. After discretizing the OCP problem it is composed by 28 variables, 24 equality constrains and 4 inequality constraints for a total number of 4 degrees of freedom which correspond to a velocity component in each discrete time instant.

This example is useful to show the capability of the framework to handle nonlinear equality and inequality constraints.

In order to solve this control problem using NMPC the user has to define the model within Pyomo. First it is necessary to include the Pyomo environment, the Pyomo module for DAE models and the new module for NMPC controllers in the Python script:

```
from pyomo.environ import *
from pyomo.dae import *
from pyomo.iNMPC import *
```

Then, it is necessary to define the variables of the problem:

```
# Model
m = ConcreteModel()
# Sets
m.t = ContinuousSet(bounds=(0, 1.0))
# Variables
m.x = Var(m.t, bounds=(None,None), initialize=0.0)
m.y = Var(m.t, bounds=(None,None), initialize=0.0)
m.vx = ManipulatedVar(m.t, bounds=(None,None), initialize=0.02)
m.vy = ManipulatedVar(m.t, bounds=(None,None), initialize=0.0)
```

Note that the continuous set of time (*m.t*) must start at zero and end at the value of the sampling time. Also the manipulated variables (*ManipulatedVar*) have the same nature of variables (*Var*).

Within the variables are the differential variables with their initial conditions. The initial condition definition receives as argument the variable with which it is associated and it inherits its indices, the key word *time_set* indicates the continuous set of time and *initialize* indicates the value of the initial condition for the first sampling period. It is important to mention that *initialize* can also be a function in the case that the variable has more than one index.

```
# Derivative variables
m.dxdt = DerivativeVar(m.x, wrt=m.t)
m.dydt = DerivativeVar(m.y, wrt=m.t)
# Initial conditions
m.x_init = InitialCondition(m.x, time_set=m.t, initialize=0.0)
```

```
m.y_init = InitialCondition(m.y, time_set=m.t, initialize=0.0)
```

The definition of the set point change is done using the disturbance class. The declaration of this class receives the time set as argument, a function that represents the disturbance over time, and a type of forecast that can be perfect or constant:

```
# Set-point change
def sp_change(c_time):
    if c_time <= 150:
        return 1
    else:
        return 0
m.y_sp = Disturbance(m.t, rule=sp_change, forecast='CONSTANT')
```

Finally, the definition of the model is completed adding the constraints and objective:

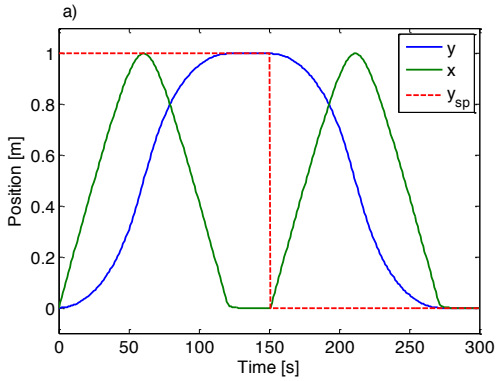
```
## Constraints
#Rail
def _c1(m, l):
    return (m.x[l]**2) + 4*((m.y[l]-0.5)**2) == 1
m.C1 = Constraint(m.t, rule=_c1)
#Max velocity
def _vn(m, l):
    return (m.vx[l]**2) + (m.vy[l]**2) <= (0.02)**2
m.vn = Constraint(m.t, rule=_vn)
#Differential equation 1
def _dx(m, l):
    if l == 0:
        return m.x[0] == m.x_init
    else:
        return m.dxdt[l] == m.vx[l]
m.dx = Constraint(m.t, rule=_dx)
#Differential equation 2
def _dy(m, l):
    if l == 0:
        return m.y[0] ==  m.y_init
    else:
        return m.dydt[l] == m.vy[l]
m.dy = Constraint(m.t, rule=_dy)
# Objective
def _obj(m):
    return sum( (m.x[l]-0.0)**2 + 5*(m.y[l]-m.y_sp[l])**2 +\
           (m.vx[l]**2) + (m.vy[l]**2) for l in m.t)
m.OBJ = Objective(rule=_obj, sense=minimize)
```

Once the model is defined, the NMPC module can be used to create the controller and to solve the whole control problem. The arguments for the definition of an iNMPC controller are the

model, the continuous set of time, the prediction horizon length, the control horizon length, and total number of simulation periods. It also has the option of using a default discretization scheme which is orthogonal collocation in finite elements with three Radau points. The *discretize_model* method receives the same arguments of the *discretize* method from the Pyomo DAE module (*pyomo.dae*) and an additional one to specify the discretization scheme. Finally, the *solve* method can be called specifying the solver and the output file name.

```
## NMPC
CONTROLLER = IdealNMPC(model=m, model_time=m.t, prediction_horizon=4,
                           control_horizon=4, simulation_periods=300,
                           default_discretization=False)
CONTROLLER.discretize_model(discretization_type='FiniteDifferences',
                                  wrt=m.t, scheme='BACKWARD')
CONTROLLER.solve('ipopt', options=[('linear_solver', 'ma27'),
                    file_name='car_iNMPC', dynamic_plot=True)
```

After this the script is completed for the evaluation of the controller and it starts generating the results. At the end the framework writes two text files, one with the computational time required for solving the OCP at each sampling period and other with the time profiles of the state variables, disturbances, and manipulated variables over the whole time of simulation. The results for this example are presented in Figure 5. Note that in average the solution of the OCP problem is achieved in 0.075 seconds which is much lower than the sampling time of the problem (one second).



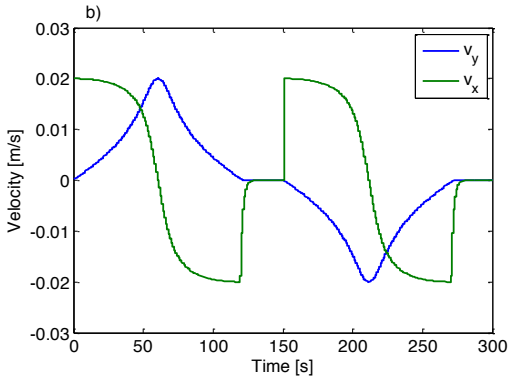


Figure 5. Ideal NMPC results for the car problem. a) States profiles for car position b) Velocity profiles (manipulated variables).

4.2. Example 2. Batch reactor for penicillin production

This example is the temperature control of a batch reactor for the production of penicillin. It is adapted from [23] and the objective is to maximize the penicillin concentration (P) at the final time, thus the objective function of the NMPC is not a standard quadratic function of tracking, but a final cost function. The OCP for this example is represented in Eq. (6) and all the variables are dimensionless, except for the temperature (θ) which is in Celsius degrees. The differential equations Eq. (6b) – (6c) are mass balances that represent the rate of change of the penicillin concentration (P) and the biomass concentration (X). The right hand side of these mass balances are dimensionless reaction rate expressions. The manipulated variable is the temperature (θ) which is bounded according to Eq. (6g). The initial biomass concentration is 0.02, the initial penicillin concentration is 0.0, and the initial temperature is 27°C.

$$\min -P(t_k + \tau N_p) \tag{6a}$$

$$s.t. \frac{dX}{dt} = b_1 X - \frac{b_1}{b_2} X^2 , \qquad X(t_k) = X_0 \tag{6b}$$

$$\frac{dP}{dt} = b_3 X, \qquad P(t_k) = P_0 \tag{6c}$$

$$b_1 = 13.1 \left[\frac{1 - 0.005(\theta - 30)^2}{1 - 0.005(25 - 30)^2} \right] \tag{6d}$$

$$b_2 = 0.94\left[\frac{1-0.005(\theta-30)^2}{1-0.005(25-30)^2}\right] \quad (6e)$$

$$b_3 = 1.71\left[\frac{1-0.005(\theta-20)^2}{1-0.005(25-20)^2}\right] \quad (6f)$$

$$20°C \leq \theta \leq 30°C \quad (6g)$$

For this problem we use a sampling time of 0.02 (dimensionless), a control horizon of 25 and a prediction horizon of 50. Also, the differential equations are discretized using orthogonal collocation in finite elements with three Radau roots, so the differential equations are transformed into algebraic constraints. It is important to mention that the framework uses this discretization by default and that is why it is not specified in the fragment of code shown hereafter. After discretizing the OCP problem it is composed by 755 variables and 730 equality constrains for a total number of 25 degrees of freedom which correspond to the reactor temperature in each sampling instant.

For this problem we consider the asNMPC and introduce a model-plant mismatch using the same model but changing the proportional constant of the expressions for b_1, b_2, and b_3 for 15.1, 0.74, and 1.91 respectively.

The definition of the model in Pyomo using a Python script is very similar to the one presented in the Example 1, the only changes are the import of the asNMPC module (`from pyomo.asNMPC import *`) instead of the iNMPC module and the definition of the controller:

```
## NMPC
CONTROLLER = AdvancedStepNMPC(model=m, model_time=m.t,
                              prediction_horizon=50,
                              control_horizon=25,
                              simulation_periods=50,
                              default_discretization=True)
CONTROLLER.solve('ipopt',
                 options=[('linear_solver', 'ma57'), ('mu_init',1E-4)],
                 file_name='Penicillin_asNMPC', real_plant = plant,
                 dynamic_plot=True)
```

In this case the *solve* method of the controller has two additional key words. The *options* key word is a list of tuples containing the options for the solver. It can also be used in the iNMPC module. The *real_plant* key word is a generic function that returns a dictionary with the states of the process at the current time which corresponds to the initial conditions of the model:

```
def plant(ManVar_dict, IC_dict, current_time):
    """
    A function of the real plant. This method must return a
    dictionary between a string corresponding to the initial
    conditions name and their value
    """
    #Read the current states
    theta = ManVar_dict['theta'][0].value
    X_init = IC_dict['X_init'].value
    P_init = IC_dict['P_init'].value
    #Model
    m = ConcreteModel()
    .
    .
    .
    #Definition, discretization and solution of the 'real' model.
    #It could also be a called to other software or measurements
    #from a process
    .
    .
    .
    #Dictionary file - This dictionary must contain all the initial
    #conditions
    return_IC_dict = {}
    return_IC_dict['X_init'] = m.X[0.02].value
    return_IC_dict['P_init'] = m.P[0.02].value
    return return_IC_dict
```

The arguments of the *plant* function are used inside the asNMPC algorithm and they correspond to the manipulated variables at the optimal solution (*ManVar_dict*), the previous initial conditions used in the OCP (*IC_dict*) and the current time (*current_time*). Note that the first two arguments are dictionaries between a string and the element of the model, so all the functionalities of Pyomo elements are inherit such as getting the value of variables and accessing to suffixes related with these elements. Additionally, if a state estimator is needed it can be defined within this function as the user can easily access the current and previous values of the

state variables and manipulated variables from here. For example, the pseudo code for the moving horizon estimator (MHE) is shown hereafter assuming that the MHE function is fully defined by the user and that it inherits the dynamic model from the NMPC definition.

```
def plant(ManVar_dict, IC_dict, current_time):
    """
    A function of the real plant. This method must return a
    dictionary between a string corresponding to the initial
    conditions name and their value
    """

    #Read the current states and the previous states measured
    #This can be done using the output text file from the NMPC
    #solution at each sampling instant.
    X = ReadStates()
    U = ReadManipulatedVariableValues()

    #Call MHE function. It must inherits the dynamic model from
    # the NMPC definition.
    Xestimated = MHE(X, U)

    #Dictionary file - This dictionary must contain all the
    # initial conditions
    return_IC_dict = {}
    return_IC_dict['X_init'] = Xestimated
    return return_IC_dict
```

Using this script the control problem can be solved and the algorithm automatically updates the initial conditions for each period and does the NLP sensitivity calculations for the correction of the model-plant mismatch. The results for this control problem are presented in Figure 6. Note that even when there is a difference in the biomass concentration over time due to the model-plant mismatch the rapid NLP sensitivity correction of the temperature (manipulated variable) allows the maximization of the penicillin concentration at the final time and there is not a significant difference between the predicted and the real value. Finally, the average solution time for OCP is 0.07 seconds and the sensitivity calculation is done faster (0.02 seconds). The size of this problem is relatively small and thus it is not observed a big difference between the time required for the OCP solution and for the NLP sensitivity calculation.

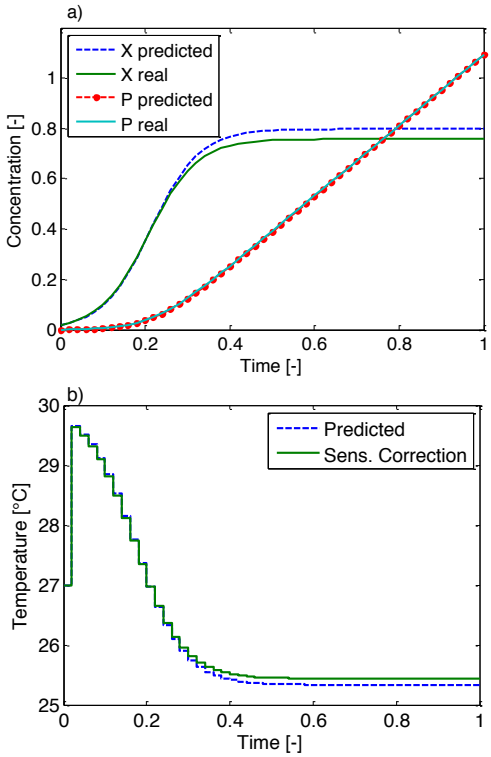


Figure 6. Advanced step NMPC results for the penicillin batch reactor problem. a) States profiles for dimensionless concentrations b) Temperature profile (manipulated variable).

4.3. Example 3. Control of an extractive distillation column

This final example is a large-scale non-linear process that serves to demonstrate how using this framework simplifies the implementation of NMPC controllers and the computational advantage of the asNMPC controller.

This example presents the control of an extractive distillation column to obtain fuel grade ethanol using glycerol as entrainer. The dynamic model and operational conditions are taken from [24] and are summarized in Appendix 1. The control objectives are the maximization of the economic profit and the minimization of a tracking quadratic function Eq. (7) when the process is subject to a sinusoidal disturbance with amplitude of 5% and period of 2 h in the ethanol feed composition. The tracking function is composed by three terms: a set point of the ethanol purity in distillate, a penalization of the changes in the reboiler duty (manipulated variable) with respect

to a reference value and a penalization of the changes in the reflux flow rate (manipulated variable) with respect to a reference value. Additionally, the initial conditions for this problem are obtained solving a steady state economic optimization and model-plant mismatch is considered introducing random noise of 5% in the liquid hold-ups and considering a constant forecast for the sinusoidal disturbance. These model-plant differences are useful for testing the performance and robustness of the asNMPC controller.

$$\min \int_{t_k}^{t_k+\tau N_p} \left[-\left(C_D D_i - C_Q Q_i\right) + \alpha_x \left(x_i^{EtOH} - x_{sp}^{EtOH}\right)^2 + \beta_Q \left(Q_i - Q_{ref}\right)^2 + \beta_R \left(L_{R_i} - L_{R_{ref}}\right)^2 \right] dt \quad (7)$$

The user have to define the dynamic model of the process which in this case is based on the MESH equations, but it is not presented here to avoid confusing the reader with the mathematical details of a high index DAE model. After discretizing the model using orthogonal collation in finite elements with three Radau points, a sampling time of 5 minutes, a control and prediction horizon of 10, the NLP problem has 8552 variables and 8532 equality constraints for a total of 20 degrees of freedom which are the reflux flow rate and the reboiler duty at each sampling instant.

The complete script, with some simplifications, for solving the control problem is as follows:

```
.
.
.
#Solution of steady state problem
.
.
.
#Imports
from pyomo.environ import *
from pyomo.dae import *
from pyomo.asNMPC import *

#Model
m = AbstractModel()
.
.
.
#Disturbance
```

```
def _za_e(c_time):
    return 80 - 5.0*sin(c_time*pi/60)
m.za_e = Disturbance(m.t, rule=_za_e, forecast='CONSTANT')
.
.
.
CONTROLLER = AdvancedStepNMPC(model=m, model_time=m.t,
                              prediction_horizon=10,
                              control_horizon=10,
                              simulation_periods=60,
                              default_discretization=True)

def plant(ManVar_dict, IC_dict, current_time):
    """
    A function of the real plant. This method must return a dictionary
    Between a string corresponding to the initial conditions name and
    their value
    """
    #Dictionary file
    return_IC_dict = {}
    for stage in m.Net:
        return_IC_dict['ML_init['+str(stage)+']'] =
                               m.ML_init[stage].value*(1 + \
                                    random.uniform(-0.05, 0.05))
        for component in m.comp:
            return_IC_dict['x_init['+str(stage)+',
                           '+str(component)+']'] =
                            m.x_init[stage, component].value
    return return_IC_dict

CONTROLLER.solve('ipopt', options=[('linear_solver', 'ma57'),
                              ('mu_init', 1E-5),
                              ('warm_start_init_point', 'yes'),
                              ('warm_start_bound_push', 1E-5),
                              ('warm_start_mult_bound_push', 1E-5)],
                               file_name='EDC_asNMPC_MPM',
                               real_plant = plant, dynamic_plot=True)
```

Note that the *Plant* function uses the states predicted by the model and return the same states introducing white noise.

Using this framework the implementation of the controller can be reduced to only two lines of code, one for declaring the controller object and other for solving the control problem. On the other hand, the definition of the model and the real plant depend completely on the user and can be complex or simple functions according to the user requirements. This gives flexibility to the

framework allowing the definition of many different models as long as they can be formulated as mathematical programs.

Figure 7 presents the results of applying asNMPC control to the extractive distillation column problem. Even when there is a model-plant mismatch and the presence of random noise the asNMPC controller is able to maintain the controlled variables close to their set-points and the changes of the manipulated variables are not too aggressive. Using the NLP solver IPOPT the average solution time of the OCP is 64.6 s which introduces a one minute delay for the iNMPC. However, this delay is avoided using NLP sensitivity which takes an average calculation time of 0.76 s almost hundred times faster than the solution of the dynamic optimization problem. Additionally, the asNMPC controller under these circumstances is able to satisfy the product purity constraint during almost all the operation. Finally, the asNMPC controller gives a very good sensitivity approximation despite the presence of model-plant mismatch and relative large disturbances.

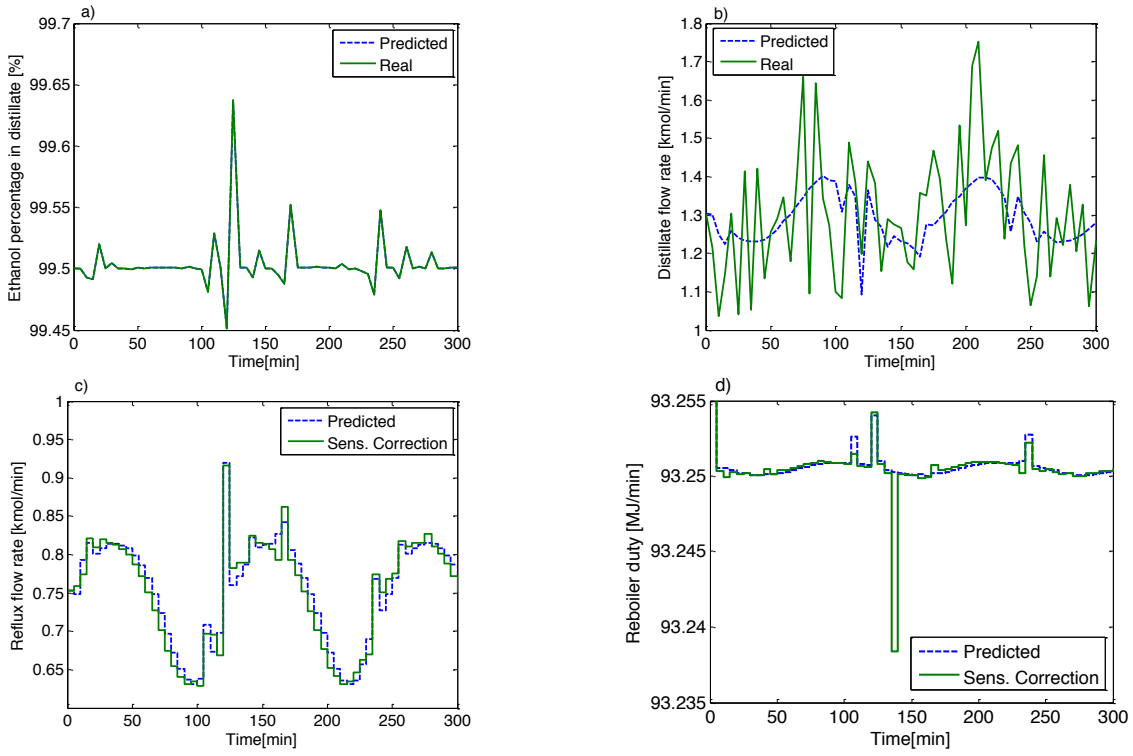


Figure 7. Advanced step NMPC results for the extractive distillation problem. a) Ethanol composition in product b) Product flow rate c) Reflux flow rate (manipulated variable) d) Reboiler duty (manipulated variable)

It is important to mention that for this large-scale problem the processing times of some instances can be large. For example, it requires 6.5 s for discretizing the whole model and 146.8 s for the controller set-up, but these operations are only done once after defining the controller. The only recurrent operation is the modification of the ".nl" file for each OCP formulation which takes an average time of 4.1 s.

5. Conclusions and perspectives

This paper presents a framework for the design, evaluation and implementation of iNMPC and asNMPC controllers. The framework is an extension of Pyomo, a mathematical modelling language embedded in Python, that avoids a complex communication between programs in the NMPC implementation and automates the controller set-up, the discretization of differential equations, the feedback of states and the NLP sensitivity calculations; the user only has to define the prediction model as a mathematical program and a function to access the real process states. It is hoped that the framework presented here will help researchers and engineers to address new developments and implementations on the field of NMPC.

The use and performance of the framework are illustrated with three examples, two small examples to present in detail how it is used and one large-scale nonlinear example to show how the framework simplifies the definition of NMPC controllers. In addition, the Examples 2 and 3 show the advantages of using asNMPC control to handle model-plant mismatch and to avoid computational delays when the solution of the OCP requires a significant amount of time.

Future developments for the framework can be included by the users according to their needs or requirements for an application. Among the additional features that can be incorporated to the framework are more detailed strategies to handle constraint violation during the NLP sensitivity calculations such as a path-following approach that uses the directional derivative at the optimal solution to compute the sensitivity step [25]; a more advanced interface with the solvers to access to the KKT matrix more efficiently; and an extension of the ideal (N)MPC module to simulate the computational delay introduce by the solution of the dynamic optimization problem, so it can be appreciated how it compromises the stability and the closed loop performance of the controller.

Acknowledgements

Many thanks to Professor Biegler, Xue Yang and Bethany Nicholson for very useful discussions and for introducing us in Pyomo.

Nomenclature

Subscript

0: initial time

1: first time predicted

$EtOH$: ethanol

f: final time

k: current instant

lb: lower bound

sp: set-point

ub: upper bound

Latin symbols

A: transpose of the Jacobian of the equality constraints in an optimization problem

c: vector of equality constraints of an optimization problem

C_D: economic profit for distillate production in the extractive distillation example

C_Q: energy cost in the extractive distillation example

I: identity matrix

L: Lagrange function

L_R: reflux flow rate in the extractive distillation column example

N_p: prediction horizon

N_u: control horizon

p: vector of parameters in an optimization problem

P: dimensionless penicillin concentration

Q: reboiler duty in the extractive distillation column example

t: time

u: vector of manipulated variable

v: vector of Lagrange multipliers for the inequality constraints

v_x: horizontal velocity in car example

v_y: vertical velocity in car example

V: diagonal matrix which elements are the Lagrange multipliers of the inequality constraints

W: Hessian matrix of the Lagrange function

x: vector of differential states, horizontal position in the car example, molar fraction in the extractive distillation column example.

x_p: artificial variables to save the parameters values in an optimization problem

X: dimensionless biomass concentration

y: vector of algebraic states, vertical position in the car example

z: vector of all the variables of an optimization problem

Z: diagonal matrix which elements are the variables of an optimization problem

Greek symbols

α_x: ponderation weight for the distillation composition in the extractive distillation example

β_Q: ponderation weight for the reboiler duty in the extractive distillation example

β_R: ponderation weight for the reflux flow rate in the extractive distillation example

λ: vector of Lagrange multipliers for the equality constraints

$\bar{\lambda}$: vector of Lagrange multipliers for the artificial equality constraints

Δ: change from predicted value to sensitivity correction

θ: temperature in the batch reactor example

ϕ: running cost of objective function

ϕ_f: terminal cost of objective function

τ: sampling time

Appendix 1

The dynamic model for a distillation column is based on mass and energy balances between the process inlets and outlets at each stage. The thermodynamic equilibrium equations and hydraulic constraints must also be include in the model.

The main assumptions for this model are: 1) thermodynamic equilibrium at each stage assuming that the vapor phase is ideal and the liquid phase is represented with the NRTL model, 2) adiabatic operation, 3) total condenser and partial reboiler, 4) there is no pressure drop in the reboiler, 5) constant mass hold-up in the condenser and reboiler, 6) the vapor hold-up is negligible with respect to the liquid hold-up for all the stages; the density of the liquid phase is much greater than the density of the vapor phase, and 7) the pressure does not vary with time. It helps the problem to be less rigid. Under these assumptions the model equations are as followed:

Total mass balance

$$\frac{dM_{L,n}}{dt} = \begin{cases} V_{n+1} - L_n - D = 0, & n = 1 \\ F_n + L_{n-1} + V_{n+1} - L_n - V_n, & n \in Stages \setminus \{1, NT\} \\ L_{n-1} - L_n - V_n = 0, & n = NT \end{cases} \quad \text{Eq. A1}$$

Partial mass balance

$$M_{L,n}\frac{dx_{n,j}}{dt} = \begin{cases} y_{n+1,j}V_{n+1} - x_{n,j}(L_n + D), & n = 1, j \in Components \\ (z_{n,j} - x_{n,j})F_n + (x_{n-1,j} - x_{n,j})L_{n-1} + (y_{n+1,j} - x_{n,j})V_{n+1} - (y_{n,j} - x_{n,j})V_n, & n \in Stages\setminus\{1, NT\}, j \in Components \\ x_{n-1,j}L_{n-1} - x_{n,j}L_n - y_{n,j}V_n, & n = NT, j \in Components \end{cases} \quad \text{Eq. A2}$$

Liquid flow rate

$$\begin{cases} F_{P,n} = \dfrac{L_n/\rho_L}{V_n/\rho_V}\left(\dfrac{\rho_L[kg/m^3]}{\rho_V[kg/m^3]}\right)^{1/2}, & n \in Stages\setminus\{1, NT\} \\ M_{L,n} = 0.6\rho_L(A_T)(h_W^{0.5})\left(\dfrac{F_{P,n}A_T p}{L_W}\right)^{0.25}, & n \in Stages\setminus\{1, NT\} \end{cases} \quad \text{Eq. A3}$$

Energy balance

$$M_{L,n}\frac{dH_{L,n}}{dt} = \begin{cases} V_{n+1}H_{V,n+1} - (L_n + D)H_{L,n} - Q_C, & n = 1 \\ (H_{F,n} - H_{L,n})F_n + (H_{L,n-1} - H_{L,n})L_{n-1} + (H_{V,n+1} - H_{L,n})V_{n+1} - (H_{V,n} - H_{L,n})V_n, & n \in Stages\setminus\{1, NT\} \\ Q_R + H_{L,n-1}L_{n-1} - H_{L,n}L_n - H_{V,n}V_n, & n = NT \end{cases} \quad \text{Eq. A4}$$

Phase equilibrium

$$y_{n,j} = K_{n,j}x_{n,j} = \frac{\gamma_{n,j}P_{n,j}^{sat}}{P_n}x_{n,j}, \qquad n \in Stages\ , j \in Components \qquad \text{Eq. A5}$$

Phase equilibrium error

$$\sum_{j\in comp} y_{n,j} - x_{n,j} = 0, \qquad n \in Stages \qquad \text{Eq. A6}$$

This model is an index two DAE and it is necessary to implement an index reduction technique to solve it using the integration method for ODE and index one DAE. It consist in differentiating the left hand side of the energy balance (Eq. A4) knowing that the specific liquid enthalpy is defined as $H_{L,n} = \sum_{j\in comp} x_{n,j}\bar{H}_{L,n,j}(T)\,, n \in Stages$. The new algebraic energy balance is:

$$M_{L,n}\left[\sum_{j\in comp} \bar{H}_{L,n,j}\frac{dx_{n,j}}{dt} + \frac{dT_n}{dt}\sum_{j\in comp} x_{n,j}\frac{d\bar{H}_{L,n,j}}{dT}\right] = \begin{cases} V_{n+1}H_{V,n+1} - (L_n + D)H_{L,n} - Q_C, & n = 1 \\ \left(H_{F,n} - H_{L,n}\right)F_n + \left(H_{L,n-1} - H_{L,n}\right)L_{n-1} + \left(H_{V,n+1} - H_{L,n}\right)V_{n+1} - \left(H_{V,n} - H_{L,n}\right)V_n, & n \in Stages \\ Q_R + H_{L,n-1}L_{n-1} - H_{L,n}L_n - H_{V,n}V_n, & n = NT \end{cases} \qquad \text{Eq. A7}$$

where $\frac{d\bar{H}_{L,n,j}}{dT}$ can be calculated analytically and $\frac{dT_n}{dt}$ is obtained from the time derivative of Eq. (A6) when $y_{n,j}$ is replaced by its definition in the VLE equation (Eq. A5).

$$\frac{dT_n}{dt} = \frac{-\sum_{j\in comp} x_{n,j}\sum_{i\in comp}\frac{\partial K_{n,i}}{\partial x_{n,i}}\frac{dx_{n,i}}{dt} - \sum_{j\in comp} K_{n,j}\frac{dx_{n,j}}{dt} + \sum_{j\in comp}\frac{dx_{n,j}}{dt}}{\sum_{j\in comp} x_{n,j}\frac{\partial K_{n,j}}{\partial T_n}}, \qquad n \in Stages \qquad \text{Eq. A8}$$

$$\frac{dT_n}{dt} = \frac{-\frac{1}{P_n}\sum_{i\in comp}\left[x_{n,i}P_{n,i}^{sat}\sum_{j\in comp}\left(\frac{\partial\gamma_{n,i}(x_{n,j},T_n)}{\partial x_{n,j}}\frac{dx_{n,j}}{dt}\right)\right] - \sum_{i\in comp}\left(K_{n,i}\frac{dx_{n,i}}{dt}\right) + \sum_{j\in comp}\frac{dx_{n,j}}{dt}}{\underset{n\in Stages}{\frac{1}{P_n}\left[\sum_{i\in comp}\left(x_{n,i}P_{n,i}^{sat}\frac{\partial\gamma_{n,i}(x_{n,i},T_n)}{\partial T_n}\right) + \sum_{i\in comp}\left(x_{n,i}\gamma_{n,i}\frac{dP_{n,i}^{sat}}{dT_n}\right)\right]}}, \qquad \text{Eq. A9}$$

Finally, when the differential states ($M_{L,n}, x_{n,j}\ n \in Stages, j \in Components$) are fixed for initial condition the subsystem of algebraic equations is nonsingular allowing the definition of consistent initial conditions.

Nomenclature for Appendix 1

Subscript

C: condenser.

j: sub index used to refer to the components of the system.

L: liquid phase.

n: sub index used to refer to the column stages.

R: reboiler.

V: vapor phase.

Latin symbols

A_T: active are of the tray.

D: distillate flow rate.

F_p: flow factor of the stage.

F: feed stream flow rate

h_w: weir height

H: specific molar enthalpy.

K: VLE constant.

L: liquid flow rate of the stage.

L_w: weir length

M: molar hold-up of the stage.

p: pitch of the tray.

P: pressure

Q: heat duty

V: vapor flow rate of the stage.

x: molar fraction in the liquid phase.

y: molar fraction in the vapor phase.

Greek symbols

γ: *activity coefficient*

ρ: molar density

References

[1] D.Q. Mayne, Model predictive control: Recent developments and future promise, Automatica. 50 (2014) 2967–2986. doi:http://dx.doi.org/10.1016/j.automatica.2014.10.128.

[2] X. Yu-Geng, L. De-Wei, L. Shu, Model Predictive Control — Status and Challenges, Acta Autom. Sin. 39 (2013) 222–236. doi:http://dx.doi.org/10.1016/S1874-1029(13)60024-5.

[3] L.T. Biegler, Nonlinear Programming. Concepts, Algorithms and Applications to chemical process, SIAM, Society for Industrial and Applied Mathematics, Philadelphi, 2010.

[4] L.T. Biegler, An overview of simultaneous strategies for dynamic optimization, Chem. Eng. Process. Process Intensif. 46 (2007) 1043–1053. doi:http://dx.doi.org/10.1016/j.cep.2006.06.021.

[5] J.L. Purohit, S.C. Patwardhan, S.M. Mahajani, DAE-EKF-Based Nonlinear Predictive Control of Reactive Distillation Systems Exhibiting Input and Output Multiplicities, Ind. Eng. Chem. Res. 52 (2013) 13699–13716. doi:10.1021/ie4004128.

[6] R. Lopez-Negrete, F.J. D'Amato, L.T. Biegler, A. Kumar, Fast nonlinear model predictive control: Formulation and industrial process applications, Comput. Chem. Eng. 51 (2013) 55–64. doi:http://dx.doi.org/10.1016/j.compchemeng.2012.06.011.

[7] K.K. Sorensen, J. Stourstrup, B. Thomas, Adaptive MPC for a reefer container, Control Eng. Pract. 44 (2015) 55–64. doi:http://dx.doi.org/10.1016/j.conengprac.2015.05.012.

[8] D. Axehill, T. Besselmann, D.M. Raimondo, M. Morari, A parametric branch and bound

approach to suboptimal explicit hybrid MPC, Automatica. 50 (2014) 240–246. doi:http://dx.doi.org/10.1016/j.automatica.2013.10.004.

[9] V.M. Zavala, L.T. Biegler, The advanced-step NMPC controller: Optimality, stability and robustness, Automatica. 45 (2009) 86–93. doi:http://dx.doi.org/10.1016/j.automatica.2008.06.011.

[10] V.M. Zavala, C.D. Laird, L.T. Biegler, A fast moving horizon estimation algorithm based on nonlinear programming sensitivity, J. Process Control. 18 (2008) 876–884. doi:http://dx.doi.org/10.1016/j.jprocont.2008.06.003.

[11] MathWorks, Model predictive control toolbox: Design and simulate model predictive controllers, (2015).

[12] W. Hart, Python Optimization Modeling Objects (Pyomo), in: J. Chinneck, B. Kristjansson, M. Saltzman (Eds.), Oper. Res. Cyber-Infrastructure, Springer US, n.d.: pp. 3–19. doi:10.1007/978-0-387-88843-9_1.

[13] W.E. Hart, J.-P. Watson, D.L. Woodruff, Pyomo: modeling and solving mathematical programs in Python, Math. Program. Comput. 3 (2011) 219–260.

[14] W.E. Hart, C. Laird, J.-P. Watson, D.L. Woodruff, Pyomo--optimization modeling in python, Springer Science \& Business Media, 2012.

[15] B.L. Nicholson, J.-P. Watson, V.M. Zavala, L.T. Biegler, pyomo.dae: A Modeling and Direct Transcription Framework for Optimization with Differential and Algebraic Equations, Submitt. Publ. (2016).

[16] B. Coffey, F. Haghighat, E. Morofsky, E. Kutrowski, A software framework for model

predictive control with GenOpt, Energy Build. 42 (2010) 1084–1092. doi:http://dx.doi.org/10.1016/j.enbuild.2010.01.022.

[17] L.T. Biegler, X. Yang, G.A.G. Fischer, Advances in sensitivity-based nonlinear model predictive control and dynamic real-time optimization, J. Process Control. 30 (2015) 104–116. doi:http://dx.doi.org/10.1016/j.jprocont.2015.02.001.

[18] R. Huang, V.M. Zavala, L.T. Biegler, Advanced step nonlinear model predictive control for air separation units, J. Process Control. 19 (2009) 678–685. doi:http://dx.doi.org/10.1016/j.jprocont.2008.07.006.

[19] H. Pirnay, R. López-Negrete, L. Biegler, Optimal sensitivity based on IPOPT, Math. Program. Comput. 4 (2012) 307–331. doi:10.1007/s12532-012-0043-2.

[20] X. Yang, L.T. Biegler, Advanced-multi-step nonlinear model predictive control, J. Process Control. 23 (2013) 1116–1128. doi:http://dx.doi.org/10.1016/j.jprocont.2013.06.011.

[21] D.M. Gay, Writting .nl files, (2005).

[22] L. Grune, P. Jurgen, Nonlinear Model Predictive Control. Theory and Algorithms, Springer, 2011.

[23] A. Constantinides, N. Mostoufi, Numerical Methods for Chemical Engineers with MATLAB Applications., New Jersey, 1999.

[24] F.L. Santamaría, J.M. Gómez, Economic Oriented NMPC for an Extractive Distillation Column Using an Index Hybrid DAE Model Based on Fundamental Principles, Ind. Eng. Chem. Res. 54 (2015) 6344–6354. doi:10.1021/acs.iecr.5b00853.

[25] J. Jäschke, X. Yang, L.T. Biegler, Fast economic model predictive control based on NLP-sensitivities, J. Process Control. 24 (2014). doi:10.1016/j.jprocont.2014.04.009.