NTNU
Norwegian University of
Science and Technology

# Model-based predictive control using Modelica and open source components

**Carles Buqueras Carbonell**

Master of Science in Engineering Cybernetics
Submission date: June 2010
Supervisor: Lars Imsland, ITK

Norwegian University of Science and Technology
Department of Engineering Cybernetics

# Problem Description

The candidate shall put together a prototype Nonlinear Model Predictive Control (NMPC) tool for models implemented in Modelica, using open-source components. Suggested components are OpenModelica/JModelica.org for modelling and simulation, Ipopt for optimization and Sundials for numerical integration. The NMPC optimization problem should be formulated using a sampled-data (discrete-time) formulation.

Tasks:
1. Give a brief description of NMPC, and the modelling language Modelica.
2. Describe the chosen open-source software packages, and how they can be used in solving a NMPC optimization problem.
3. Implement a "tanks in series"-model in OpenModelica. Investigate how the model can be interfaced from other software using C/C++.
4. Make a C/C++ program that simulates the (Open-)Modelica-model. Start by using forward Euler. Discuss the use of, and if time allows, implement, more advanced integration routines, like e.g. Sundials/CVODE(S).
5. Extend the program to calculate an NMPC objective function. Discuss methods for calculating the gradient of the objective function.
6. Interface the objective function to the Ipopt optimization tool.
7. Suggest and, as far as time permits, implement, a framework for NMPC optimization.

Assignment given: 15. February 2010
Supervisor: Lars Imsland, ITK

## Preface

This work is the master thesis of my Industrial Engineering studies specialization on Automation. It has been done during an academic exchange at Norwegian University of Science and Technology NTNU, being the Polytechnic University of Catalonia UPC the origin university. Because of working in a different country with a different language, an additional effort of communication and integration maked this thesis more difficult but more interesting at the same time. Before start this thesis I had a general scientific knowledge and specifically in automation and control technology, however I didn't know about MPC method. Because of my studies specializaion and because I would like to direct my professional carreer toward this way, the subject of this thesis fits very well in my expectations.

It has been supervised by Professor Lars Imsland along 4 months of work. The main effort has been implementing software and integrating some different open source tools, so this report also explains theoretical bases but the most explanations are in how this implementation has been done.

I would like to express my gratitude to Professor Lars Imsland, who gave me an excellent guidance throughout the project.

Carles Buqueras Carbonell, June 2010

## Abstract

This thesis is about Model Predictive Control (MPC) method for process control. It describes how this method could be implemented using some different open source software components, describing functionalities of each one and showing how the implementation has been done. Finally the code is tested to demonstrate effectiveness of this software in front of this kind of problems and to demonstrate MPC main characteristics. The main goals of this thesis are these last ones, code development and tests, so all mathematical and theoretical background are described but not as in detail as development and tests.

Globally describing, MPC is a process control method where a previous knowledge of the plant is needed, so the controller have a model to simulate and predict the behavior of the system to calculate the best command signal. It has an optimization algorithm determining the optimal trajectory to bring system from initial state to desired state. Optimization is done by iterative simulation and solved online periodically at each sample time, initializing values at each time with measured feedback.

## Contents

1. Problem description

The candidate shall put together a prototype Nonlinear Model Predictive Control (NMPC) tool for models implemented in Modelica, using open-source components. Suggested components are OpenModelica/JModelica.org for modeling and simulation, Ipopt for optimization and Sundials for numerical integration. The NMPC optimization problem should be formulated using a sampled-data (discrete-time) formulation.

Tasks:

1. Give a brief description of NMPC, and the modeling language Modelica.
2. Describe the chosen open-source software packages, and how they can be used in solving a NMPC optimization problem.
3. Implement a "tanks in series"-model in OpenModelica. Investigate how the model can be interfaced from other software using C/C++.
4. Make a C/C++ program that simulates the (Open-)Modelica-model. Start by using forward Euler. Discuss the use of, and if time allows, implement, more advanced integration routines, like e.g. Sundials/CVODE(S).
5. Extend the program to calculate an NMPC objective function. Discuss methods for calculating the gradient of the objective function.
6. Interface the objective function to the Ipopt optimization tool.
7. Suggest and, as far as time permits, implement, a framework for NMPC optimization.

Assignment given: 15 February 2010

Supervisor: Lars Imsland

## 2. Model Predictive Control

MPC is a process control method where the controller uses a model of the plant to simulate and predict system behavior. This prediction is repeated iteratively and used by an optimizer to calculate the optimal command values. These values are applied to the plant and the entire optimization problem is solved again for next sample time. Initial values are always put up to date with measured output. MPC special interests are that it can work with complex and non-linear systems and allow imposing a large number of specifications and constraints. When system is non-linear, MPC is extended to Non-linear Model Predictive Control (NMPC), having the same principles with a little differentiation in mathematical development. Nowadays more and more specifications are required in control field (economic performance, safety constraints or environment goals) being MPC a good tool for that. It has been during the lasts 40 years when MPC has grown up and has acquired more and more popularity.

The most common applications of MPC/NMPC are [11] [15]:

- Distillation column
- Hydrocracker
- Pulp and paper plant
- Solar power plant
- Mobile robot
- …

The principle of MPC is that controller has an optimizer based on Sequential Quadratic Programming. Inside, an objective function is defined with all desired values (reference values) in order to direct the optimization to desired goal. At each sample time optimizer simulate and predict states evolution with different values until converge into an optimal solution. In fact, at each iteration a set of values are tried, system behavior is simulated with these values and the objective function is evaluated in order to know the cost (how much good is this solution). By a line search method, optimizer iterates until finding the optimal values. The result is an input trajectory that bests bring states to their reference values. As MPC works in a discrete-time formulation, only first value of this trajectory is applied to the plant and the entire problem is reformulated again for the next sample time. Initial values are put up to date with measured output. Predictions are done along a finite time length, along what is called a “prediction horizon” as it is the time while future system response is predicted.

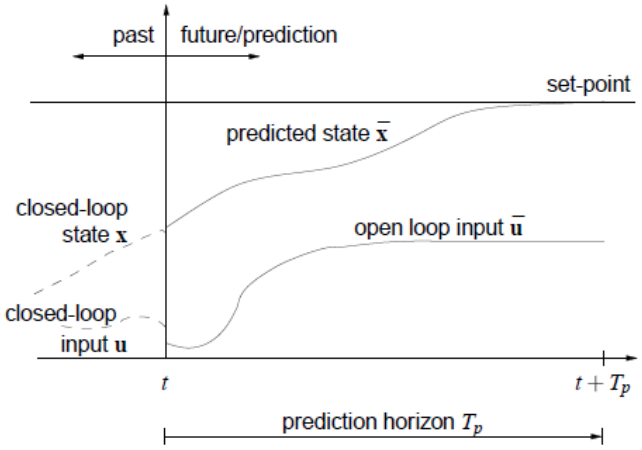



Figure 1. Principle of Model Predictive Control. Along the prediction horizon, an input trajectory $u$ is checked and states evolution is predicted. Optimizer uses this prediction to evaluate objective function and check how much good is the proposed $u$ solution. Different $u$ trajectories are tried until find the optimal one [9].

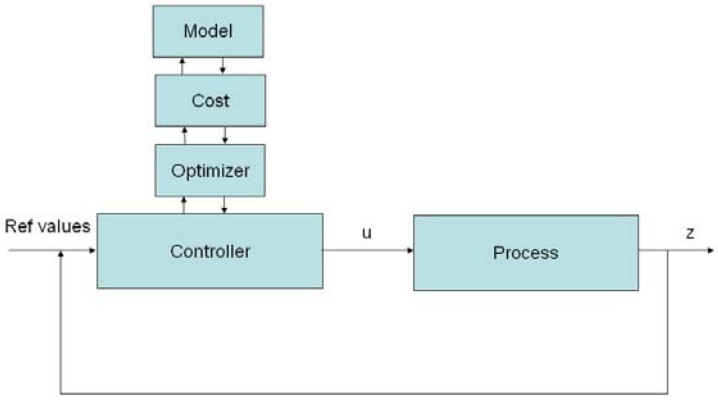



Figure 2. Overview concept of MPC. The controller gets reference values and measured output $z$, it runs the optimizer to find a control input $u$ that best satisfies reference values. Optimizer tries iteratively with different possible solutions and evaluates for each one their cost until finding the optimal one. At each cost evaluation, a simulation is done with the model to predict process behavior and to calculate how much close states are from reference values.

## 2.1. Mathematical formulation

MPC can work with linear or non-linear systems, in this second case we talk about Non-linear Model Predictive Control NMPC. This is an extension on MPC and both use the same principle even if mathematical development is a little bit different. Let's show how is formulated the problem in each case:

MPC

- It uses a linear model $\dot{x} = Ax + Bu$
- Quadratic objective function $J = x^T Qx + u^T Ru$
- Linear constraints $Hx + Gu < 0$
- Quadratic programmation

NMPC

- Non-linear model $\dot{x} = f(x,u)$
- Objective function can be non quadratic $J(x,u)$
- Non-linear constraints $h(x,u) < 0$
- Non-linear programming

Predictive control is a good tool for linear systems as it allows working with large number of constraints and considering a lot of criteria. However it is much more interesting on non-linear systems where own to their non-linearities, control is more difficult with classical methods. In MPC/NMPC, model mathematical representation can be either as ODE or DAE. For simplicity we assume that it is formulated as a continuous ODE. In a Non-linear MPC case, the problem is formulated as:

$$\dot{x} = f(x,u,p) \ , \ y = h(x,u,p) \ , \ z = g(x,u,p) .$$

Where $x$ are state variables, $u$ the controlled inputs, p the parameters, $y$ the measured outputs (not necessarily controlled) and $z$ the controlled outputs. As we are interested in expression above to calculate states and outputs at the next sampling instant, we are interested to work in a discrete-time representation.

$$x_{k+1} = x_k + \int_{t_k}^{t_{k+1}} f\left(x(\tau), u_k, p_k\right) d\tau,$$

$$y_k = h(x_k, u_{k-1}, p_{k-1}),$$

$$z_k = g(x_k, u_{k-1}, p_{k-1}).$$

The integration involved is in general solved by ODE solver routines. As the system above is used for prediction, we are not concerned with the measured outputs, so we can omit these ones (not controlled outputs). Then the time-varying discrete-time system can be represented as

$$x_{k+1} = f_k(x_k, u_k), \quad z_k = g_k(x_k, u_{k-1}).$$

The quadratic objective function to minimize at each sample time is

$$\min J\left(x_0, u_0, u_1, \ldots, u_{N-1}\right) = \frac{1}{2} \sum_{i=0}^{N-1} ((z_{i+1} - z_{ref})^T Q(z_{i+1} - z_{ref}) + (u_i - u_{ref})^T R(u_i - u_{ref})),$$

Where $x_0$ are present measured state variables used as initial state, $u_i$ future manipulated inputs, $z_i$ computed (predicted) output and $Q$ and $R$ are weighting matrices. $Q = Q^T \geq 0$ , $R = R^T > 0$.

The problem is subject to constraints

$$u_{\min} \leq u_k \leq u_{\max}, \qquad k = 0, \ldots N-1$$

$$z_{\min} \leq z_k = g_k(x_k, u_{k-1}) \leq z_{\max}, \qquad k = 1, \ldots N-1$$

Optimal solution is then the command trajectory $u$ that best controls the process. Once the optimal trajectory is obtained, only the first value $u_0$ is sent to the plant, the output is measured and the entire problem is reformulated again for the next sample time. Note that objective function contains all the performance specifications and all the desired criteria, each one weighted according to its importance.

In figure 1 we saw an optimal input trajectory $u$ taking predicted states $x$ to the set point. We could easily believe that applying $u$ trajectory to the plant we would get exactly the predicted state values $x$. Later we will demonstrate that measured output is never exactly as predicted because prediction horizon has a finite length and it ignores future values beyond this horizon. In addition, as integration routines are not exact, and because model mistakes and external perturbations, there is also more error.

This method has a discrete-time formulation, it runs on a cyclic way, sampling the output to get the feedback, solving periodically the optimization problem and sending discrete command values at each sample time. It has the advantage to use all the possibilities of computation that discrete time

technology offers. As MPC is a real-time routine, a special attention to computation effort should be given. Heavy calculations with short sampling periods could make control not fast enough to satisfy real time exigencies.

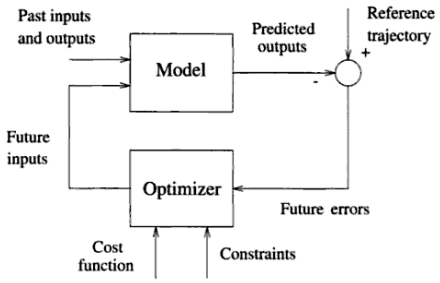



Figure 3. Basic structure of MPC. Optimizer uses the Model to predict outputs, to compare these outputs with reference values and to find a solution in accordance with cost function and constraints.[9]

There are three types of MPC methods using SQP: Single Shooting (or Sequential Approach), Simultaneous Approach and the Multiple Shooting Approach.

### 2.1.1. Single Shooting or Sequential Approach

Here only manipulated variables $u$ are free optimization variables[13]. $x(u)$ and $z(u)$ are implicit functions which are obtained by simulation. In each step of the optimization algorithm, system simulation and optimization are performed sequentially. This procedure is robust only when the system contains stable modes. Otherwise, finding a feasible solution for a given set of control parameters may be difficult. This is the method we used for implementing.

### 2.1.2. Simultaneous Approach

In this case both $u$ , $x$ and $z$ are optimization variables [14]. Constraints can be added on $x$ and $z$ during optimization, for example path constraints to guide trajectories. There's more control so it's better for unstable or highly non-linear systems but as optimization takes more variables, computation is harder.

### 2.1.3. Multiple Shooting Approach

Multiple Shooting is a method between Sequential and Simultaneous approach. The prediction horizon is divided into many sub-horizons and extra equality constraints are added to join the end of each sub-horizon with the next one. Dividing the horizon the non-linearities are spread out on smaller sub-horizons and there is more control over the simulation. Inequality constraints for states and controls can be imposed directly at each horizon begining.

## 2.2. Sequential Quadratic Programming SQP

MPC optimizers normally use a SQP for finding an optimal solution. This is an effective method for optimization problems subject to non-linear constraints, unlike Lagrangians Methods that works with linear constraints. SQP generates a search direction by solving a sequence of quadratic programs (QP) and it iterates until converge into a solution [4]. The operating mode is considering an objective function $J(x)$ subject to constraints $c(x) = 0$.

$$\min J(x)$$

$$\text{Subject to} \qquad c(x) = 0$$

We define current iterate by $x_k$ and next iteration $x_{k+1} = x_k + p_k$ where $p_k$ is the search direction. The Lagrangian function is defined by $L(x, \lambda) = J(x) - \lambda^T c(x)$. Then $A(x)$ is the jacobian matrix of the constraints $A(x)^T = [\nabla c_1(x), \nabla c_2(x), ..., \nabla c_m(x)]$ where $c_i$ is the $i$ th component of the vector $c(x)$. If we had only equality constraints we could use Newton's method. However let's also consider inequality constraints:

$$\min J(x)$$

$$\text{Subject to} \qquad c(x) = 0$$

$$c(x) \geq 0$$

There are two types of SQP: Inequality-constrained Quadratic Programming IQP and Equality-constrained Quadratic Programming EQP.

### 2.2.1. Inequality-constrained Quadratic Programming (IQP)

At each iteration this method solves the quadratic subprogram QP defined by

$$\min_p J_k + \nabla J_k^T p + \frac{1}{2} p^T \nabla_{xx}^2 L_k p$$

and with the linearized constraints

$$\nabla c_i(x_k)^T p + c_i(x_k) = 0$$

$$\nabla c_i(x_k)^T p + c_i(x_k) \geq 0$$

This QP can be solved applying Newton's method to the Karush-Kuhn-Tucker (KKT) conditions ([4] chapter 16). The solution is then used as a guess of the main problem, defining a search direction and speeding the resolution of it.

### 2.2.2. Equality-constrained Quadratic Programming (EQP)

At each iteration this method selects a subset of constraints in accordance of which it detects are active, this subset is called working set. These constraints are imposed as equalities and all other constraints are ignored. The problem is then:

$$\min_p J_k + \nabla J_k^T p + \frac{1}{2} p^T \nabla_{xx}^2 L_k p$$

$$A_k p + c_k = 0$$

Working set is up to date at each sample time by rules based on Lagrange multiplier estimates or by solving an auxiliary subproblem. The advantage of this method is that equality-constrained quadratic subproblems requires in general less computation efford than solve the QP as in IQP. An important thing in this method is how the working set is choosen.

## 2.3. Gradient computation

Optimization algorithm needs to know the derivative values of the objective function. This information should be provide to SQP so it can determinate the searching direction of the optimal solution. For a general function $f(x)$ we can approximate its gradient using the **Forward-difference** formula [4]:

$$\frac{df}{dx_i}(x) \cong \frac{f(x+\varepsilon e_i) - f(x)}{\varepsilon}$$

where $\varepsilon e_i$ represents a small change in the value of a single component of $x$ (the $i$ th component). $\varepsilon$ is the step difference. The Hessian matrix of the Lagrangian Function is also needed. It is expressed as [5]:

$$\sigma_f \nabla^2 f(x_k) + \sum_{i=1}^{m} \lambda_i \nabla^2 g_i(x_k)$$

First term comes from the Hessian of the objective function and the other ones from the Hessian of constraints. As each term evaluates second derivatives, we also use Forward-difference approximation in its double derivative extension [4]. For a general function $f(x)$ this approximation is:

$$\frac{\partial^2 f}{\partial x_i \partial x_j}(x) = \frac{f\left(x+\varepsilon e_i+\varepsilon e_j\right) - f\left(x+\varepsilon e_i\right) - f\left(x+\varepsilon e_j\right) + f\left(x\right)}{\varepsilon^2} + O(\varepsilon)$$

where $O(\varepsilon)$ is the error of the approximation.

However, the optimizer we used in our implementation (Ipopt), offers a method for calculating the Hessian without need of implementation. It is based on **Quasi-Newton** method.

### 2.4. Quasi Newton Method

This method is used to get the Hessian matrix without the need to provide second derivatives and without the need to calculate the whole matrix each time. In fact it provides an up to date of the Hessian at instant k+1 based on the Hessian at instant k and observing changes in gradient of objective function. This is similar to the line search Newton method but here instead of the true Hessian we use an approximate one. For $x_k$ , a quadratic model of the objective function $f_k$ is created by Taylor series form[4]:

$$m_k(p) = f_k + \nabla f_k^T p + \frac{1}{2} p^T B_k p$$

Where $B_k$ is a matrix put up to date at each iteration. Then the Hessian will be obtained by the inverse relation with $B_k$ matrix: $H_k = B_k^{-1}$. $p_k$ is the search direction who defines next iteration as:

$$x_{k+1} = x_k + \alpha_k p_k$$

Where $\alpha_k$ is the step length. The gradient of $m_{k+1}$ match the gradient of the objective function $f$ as:

$$\nabla m_{k+1}(-\alpha_k p_k) = \nabla f_{k+1} - \alpha_k B_{k+1} p_k = \nabla f_k$$

Following equations are then defined:

$$s_k = x_{k+1} - x_k \quad \text{and} \quad y_k = \nabla f_{k+1} - \nabla f_k$$

From here there are some different Quasi Newton methods to finish this problem: BFGS, SR1, DFP and Broyden. Let's only descrive BFGS and SR1 as they are the most used.

### 2.4.1. BFGS

A search direction is computed by

$$p_k = -H_k \nabla f_k$$

and

$$x_{k+1} = x_k + \alpha_k p_k$$

where $\alpha_k$ is computed satisfying Wolfe conditions.

Then $B_{k+1}$ is computed by the formula:

$$B_{k+1} = B_k - \frac{B_k s_k s_k^T B_k}{s_k^T B_k s_k} + \frac{y_k y_k^T}{y_k^T s_k}$$

As Hessian matrix is the inverse of $B$, applying Sherman-Morrisson formula we obtain the updated Hessian matrix

$$H_{k+1} = H_k - \frac{H_k y_k y_k^T H_k}{y_k^T H_k y_k} + \frac{s_k s_k^T}{y_k^T s_k}$$

### 2.4.2. SR1

The solution depends of these 3 cases:

- If $\left(y_k - B_k s_k\right)^T s_k \neq 0$

$$B_{k+1} = B_k + \frac{\left(y_k - B_k s_k\right)\left(y_k - B_k s_k\right)^T}{\left(y_k - B_k s_k\right)^T s_k}$$

Again, Sherman-Morrisson formula is applied to obtain the updated Hessian matrix.

$$H_{k+1} = H_k + \frac{\left(s_k - H_k y_k\right)\left(s_k - H_k y_k\right)^T}{\left(s_k - H_k y_k\right)^T y_k}$$

- If $y_k = B_k s_k$ then

$$B_{k+1} = B_k$$

- And if $y_k \neq B_k s_k$ and $\left(y_k - B_k s_k\right)^T s_k = 0$ BFGS method should be used.

See [4] chapter 6 for a step by step formula demonstration.

## 3. Open source tools

As we saw, MPC needs a model to predic system behavior, so that we need software to modelize, simulate and optimize. As it is a master thesis done in a universitary context, the most appropriate is to try to use open source tools as universities are the main developers. Despite in this thesis there's no intention to provide improvements to this software, it wants to demonstrate its efectiveness and to give a practical application as MPC. Once decided to use open source software, we should determinate which one.

In modelizing world, there are a lot of different languages: **Modelica**, AMPL, APMonitor, Domain-specific modeling (DSM), General Algebraic Modeling System (GAMS), Matlab, etc. Modelica is one of the most popular languages, it's easy and compatible with a large variety of open source components. In addition a lot of libraries and documentation has been developed and there are a lot of modularities already implemented that makes developing easily. Therefore it is a good tool and that's why we choose it. For modeling we did not use an environnment because the simplicity of the chosen problem let us to write model directly in a file without so much problem. This file has a .mo extension and it has all the process information synthesized inside: the dynamic equations, all variables definitions and parameters. If system where more complex, the use of an evironnment would be necessary. Some free environments are **OpenModelica** and SCICOS. Between these two we would choosen OpenModelica as it is the most popular and more tutorials are written about it. OpenModelica it's not only a modeling environment but also an optimizer, compiler and simulator. Once we had the model written in modelica, we tryed to use OpenModelica to compile it into a c file. However in OpenModelica an optimization problem must be defined. As we only wanted the model in c without beeing optimized, we looked for another alternative. In fact, we wanted to have the model file separated of the optimization so it was flexible and easy to integrate to the main code and to work with it. We found **Python** component from **Jmodelica** platform that is able to do that. In addition, Python generated file allows to use the large library JMI of Jmodelica for evaluating the model. The generated C file has all needed information about the system and also some functions already implemented that easily allows accessing all the information and evaluating dynamic equations. The interest of having the model in c is that it becomes compatible with the main control algorithm, already implemented in c. In addition Python deliver some XML files with also information about the model such as name of variables, parameter values, etc. Finally we shoud choose the optimizer, an utiliy able to solve a non-linear problem and flexible enough to integrate into our MPC structure. In this thesis statement, the **Ipopt** tool was suggested. In addition we saw that this software uses approximative methods as Quasi-Newton on computation, so we though it was a fast optimizer, that's why we choose it. However there are some other non-linear optimizers as SNOPT, Trust Region SQP solver, MINOS, CONOPT or KNITRO.

Once we have the model in c, we have JMI libraries to acced this file and we have choosen the optimizer, is time to implement an algorithm that integrate all these elements and who has the control routine for executing control. See the global structure of our implementation in figure 4.

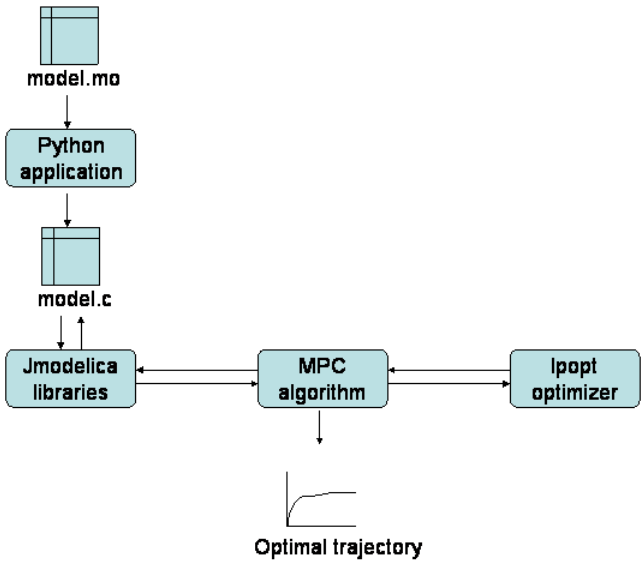



Figure 4. Basic structure of implementation. A Modelica model is written, compiled into a c-file by Python and interfaced with main algorithm through Jmodelica libraries. Main algorithm uses Ipopt optimizer to solve the problem and to get the optimal trajectory. Note that model.c is always evaluated through functions defined in Jmodelica libraries.

Let's now describe in detail each tool.

### 3.1. The Modelica language

Modelica is a modeling language born from the international effort of university community and of some small enterprises. Before Modelica, each one had their own modeling language, it was on 1996 when they did an effort to unify all concepts to create the Modelica language. The main goal was to have the possibility to exchange models, libraries and predevelopped modules to share technology advances in object oriented programming. Nowadays, this language is a good tool for working with complex systems as mechanical, electrical, thermal, hydraulic, pneumatic, fluid, etc. Models can be written in a differential way, algebraic or as discrete equations and developing can be done using a graphical editor like Simforge or just typing directly in a file. There are few Modelica simulation environments, commercial and free. Below there is a list of the most important.

| Free Modelica Simulation Environments (alphabetical list) |
|---|
| • **OpenModelica** from Linköping University, Sweden<br>• **SCICOS** from INRIA, France |
| Commercial Modelica Simulation Environments (alphabetical list) |
| • **CATIA** Systems from Dassault Systèmes<br>• **Dymola** from Dynasim AB,Sweden<br>• **LMS Imagine.Lab AMESim** from LMS International<br>• **MapleSim** from MapleSoft, Canada<br>• **MathMdelica** from MathCore AB,Sweden<br>• **SimulationX** from ITI GmbH, Dresden, Germany |

Table 1. Most important Modelica Simulation Environments classified between free and commercial [a].

The main Modelica libraries are [a]:

- Libraries for electric, electronic and magnetic components
- Libraries for mechanical components
- Libraries for fluid components
- Libraries for control systems
- Libraries for functions

Mode details about libraries on appendix 8.2 .

### 3.2. Open Modelica

Open Modelica is a free compilation and simulation environment for models written in Modelica language. Open Modelica has about 40 different modules that includes from edition, compilation, debug to execution. Here there is the overall architecture of the OpenModelica environment. Arrows denote data and control flow.

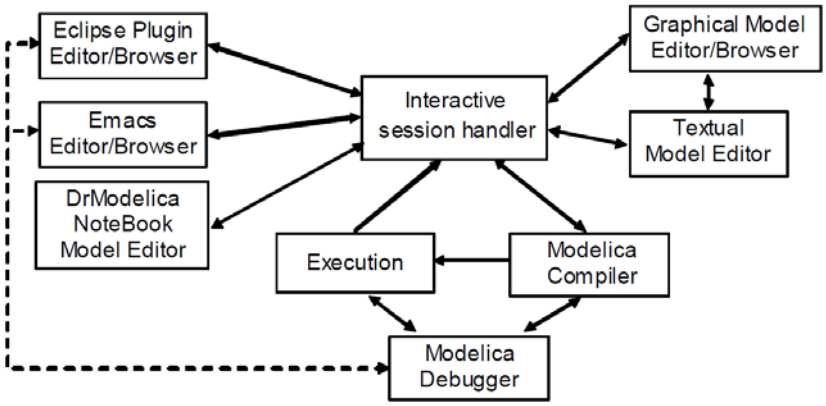



Figure 5. The overall architecture of the OpenModelica environment [c].

Compilation and execution process passes through all these steps:

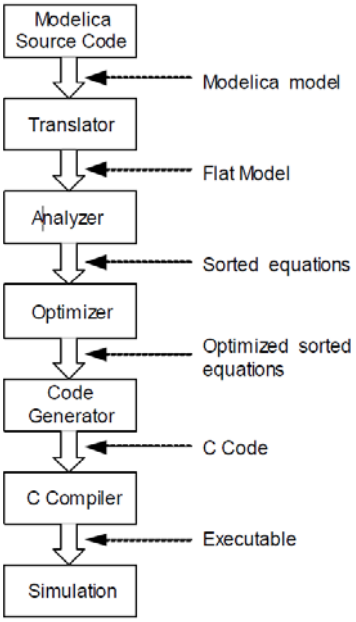



Figure 6. Steps from model to simulation in Open Modelica [c]. The source code is analyzed and as OpenModelica can also optimize problems, it is optimized for then being compiled and simulated.

### 3.3. Jmodelica.org

**Jmodelica.org** is an open source platform composed by some different components working with Modelica language. These different components allow modeling heterogeneous and complex systems, and also compiling and optimizing through Optimica language (an extension of Modelica). The diferent components are:

- Modelica compiler
- Optimica compiler
- JModelica.org Model Interface (JMI) C Runtime library
- Optimization algorithm
- Python user environment
- Functional Mockup Interface (FMI) compliance
- XML export

From this platform we used **Python** and some **JMI Runtime Libraries**. Python is an environment for scripting, developing applications and algorithm integrating. It is divided in packages Numpy and Scipy in order to provide support for numerical computation as matrix and vector operations, basic linear algebra and plotting.

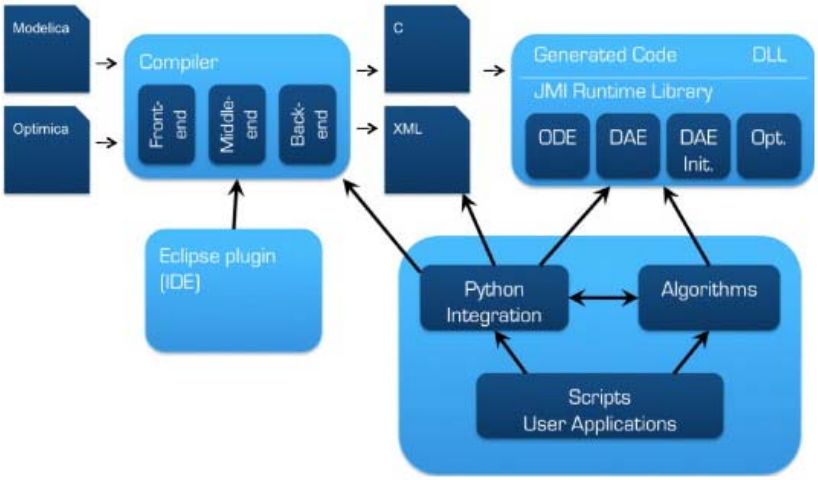



Figure 7. Structure of Jmodelica platform. Model is compiled into a c and xml files and interfaced to user applications through JMI Runtime Library [b].

In figure 7 we can see on the left the Modelica model and the Optimica optimization problem. Jmodelica compiles these files into .c and .xml files. As you can see on the bottom right, users applications can then interface their algorithms with generated c and xml files through the API (Application Programming Interface) and take profit of the JMI libraries (top right). These libraries has implemented functions for create a new jmi structure (an instance of the model into the code), functions for initialize that structure, for evaluate dynamic equations, for optimization, etc. While using Jmodelica during our implementation, we have not used the optimization tools as Optimica language and optimization functions in Jmodelica libraries. In our implementation, Jmodelica is only used to interface the model with all the other parts of the MPC. That is, optimization is done by Ipopt independently of Jmodelica optimization resources.

### 3.4. Ipopt

Ipopt (Interior Point Optimizer) is an open source package for optimizing non-linear problems. It implements a point line search filter, a SQP method, which aims to find a local solution. It has been designed flexible so that it could be used in a large number of applications. The problem is formulated as [5]:

$$\min_{x \in \Re^n} \quad f(x)$$

$$s.t. \quad g^L \leq g(x) \leq g^U$$

$$x^L \leq x \leq x^U$$

It finds a solution of $x$ variables that minimize $f(x)$ objective function while respecting $g(x)$ constraints and bound values $g^L, g^U, x^L$ and $x^U$. $f(x)$ and $g(x)$ functions can be linear or non-linear, convex or non-convex but must be double differentiable.

For solving the problem, Ipopt needs the following information:

1. Problem dimensions
    - number of variables (n)
    - number of constraints (m)
2. Problem bounds
    - variable bounds ( $x^L$ and $x^U$ )
    - constraint bounds ( $g^L$ and $g^U$ )
3. Initial starting point
    - Initial values for the primal x variables
    - Initial values for the multipliers (only required for a warm start option)
4. Problem Structure
    - number of nonzeros in the Jacobian of the constraints
    - number of nonzeros in the Hessian of the Lagrangian function
    - sparsity structure of the Jacobian of the constraints
    - sparsity structure of the Hessian of the Lagrangian function
5. Evaluation of Problem Functions (Information evaluated using a given point $x, \lambda, \sigma_f$ coming from Ipopt)
    - Objective function, $f(x)$
    - Gradient of the objective $\nabla f(x)$
    - Constraint function values, $g(x)$
    - Jacobian of the constraints, $\nabla g(x)^T$
    - Hessian of the Lagrangian function,

      Lagrangian function is $f(x) + g(x)^T \lambda$

      And the Hessian of the Lagrangian is $\sigma_f \nabla^2 f(x_k) + \sum_{i=1}^{m} \lambda_i \nabla^2 g_i(x_k)$

      First term comes from the Hessian of the objective function, while the other ones from the Hessian of constraints. As Ipopt can ask independently for

constraints Hessian, a factor $\sigma_f$ is introduced in front of the objective function to enable it if it's needed. $\lambda_i$ are the multipliers of constraints.

## 4. Implementation

Once the theoretical background is described, we proceed to define a particular problem to implement it. This problem will serve us as an example for testing and to make sure we did a good integration of open source components and a good implementation of MPC.

### 4.1. Tank in series problem

We choose a problem consisting on two tanks communicated in series, see figure 8. The first tank gets an external flow that also corresponds to global system input. Tank one output is regulated by a valve and sends the flow to tank two, this flow is then the input of tank two. The output of the second tank is also regulated by a valve.

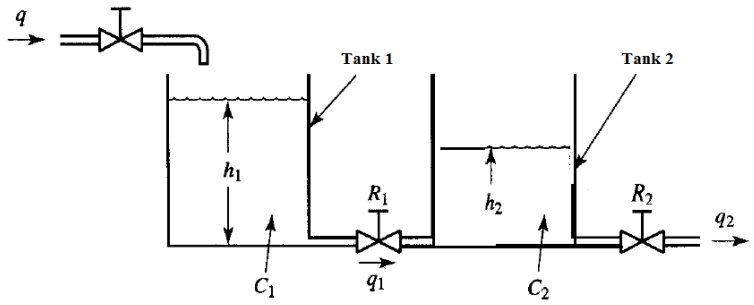



Figure 8. System used in the implementation. It consists in two tanks connected in series. The global system has an input flow going to tank 1, and an exit flow going out from tank 2. Valves are not manipulated variables, they are considered constant parameters to reduce exit flows.

System variables are:

q= tank 1 input flow ($m^3/s$)
$q_1$= tank 1 output flow and tank 2 input flow ($m^3/s$)
$q_2$= tank 2 output flow ($m^3/s$)
$h_1$= tank 1 level (m)
$h_2$= tank 2 level (m)

Parameters:

| | |
|---|---|
| $C_1 = 2$ | tank 1 surface ($m^2$) |
| $C_2 = 2$ | tank 2 surface ($m^2$) |
| $R_1 = 0.5$ | valve 1 opening coefficient (no dimension) |
| $R_2 = 0.5$ | valve 2 opening coefficient (no dimension) |

Dynamic model is represented by the equations:

$$\frac{h_1 - h_2}{R_1} = q_1$$

$$C_1 \frac{dh_1}{dt} = q - q_1$$

$$\frac{h_2}{R_2} = q_2$$

$$C_2 \frac{dh_2}{dt} = q_1 - q_2$$

The goal is here to control $h_2$ level by input q. We want to take $h_2$ level from 0 m initial state to 5 m. Putting together the equations above we gets a system of two equations.

$$\frac{dh_1}{dt} = \frac{q - \frac{h_1 - h_2}{R_1}}{C_1}$$

$$\frac{dh_2}{dt} = \frac{\frac{h_1 - h_2}{R_1} - \frac{h2}{R_2}}{C_2}$$

The objective function should be defined with the purpose to achieve $h_2$=5 m, so a reference value $h_2$_ref =5 should be defined. In addition, in order to optimizer converge faster into a solution, we set reference values for the other variables (q and $h_1$).

The objective function is:

$$f = R\left(q - q\_ref\right)^2 + Q_1\left(h_1 - h_1\_ref\right)^2 + Q_2\left(h_2 - h_2\_ref\right)^2$$

Where $R$, $Q_1$ and $Q_2$ are weighting coefficients determining the importance of each element in the solution. This is one equation with one manipulated variable (q) and two controlled variables ($h_1$ and $h_2$). To be sure this function pretends to achieve a possible state, reference values must also represent a

possible state. If we want to have $h_2$=5 m, q_ref and $h_1$_ref must be set in accordance to system equations in steady state. That is, in steady state derivatives are zero so system is:

$$0 = \frac{q - \frac{h_1 - h_2}{R_1}}{C_1}$$

$$0 = \frac{\frac{h_1 - h_2}{R_1} - \frac{h_2}{R_2}}{C_2}$$

And when $h_2$=5 → $h_1$=10 and q=10, so reference values should be these ones:

q_ref = 10 m.
$h_1$_ref = 10 m.
$h_2$_ref = 5 m.

So we make sure that all reference values can be achieved at the same time, so objective function is pointing to a possible state.

The result of evaluate the objective function is the cost of the solution of the optimization problem, the goal is to minimize this value. We remember that this objective function could be expressed considering some other criteria as safety, economics, etc. To ensure a long life of actuators and to avoid damages on them, we impose an input constraint. That is, we don't want sudden changes on command so the difference between two sequential commands is bounded as:

$$-2 \leq q(k) - q(k-1) \leq 2$$

The problem is now already defined, it should be written in Modelica language for compile and transform into a c file. As we saw, having the model in c language make it compatible with all control algorithm and optimization.

### 4.2. Modelica model

Controller needs the model of two tanks system for simulate and predict the behavior. Modeling according to Modelica standard we obtain:

```
class TwoTanks.TwoTanks

 input Real q "entry flow (m3/s)";
 parameter Real R1 = 0.5 "valve 1 resistance coefficient" ;
 parameter Real R2 = 0.8 "valve 2 resistance coefficient ";
 parameter Real C1 = 3 "tank 1 surface (m2)";
 parameter Real C2 = 2.5 "tank 1 surface (m2)";
 parameter Real h1_init(initial value) = 10;
 parameter Real h2_init(initial value) = 10;
 Real h1(start = h1_init,fixed = true) "water level in tank 1(m)";
 Real h2(start = h2_init,fixed = true) "water level in tank 2(m)";

equation

 der(h1) = ( q - (( h1 - ( h2 )) / ( R1 ))) / ( C1 );
 der(h2) = (( h1 - ( h2 )) / ( R1 ) - (( h2 ) / ( R2 ))) / ( C2 );

end TwoTanks.TwoTanks;
```

Model of the system expressed in Modelica language.

Note that there are two different parts, the first one containing information about variables and parameters, and the second one information about dynamic behavior. Probably modeling is the most complicated part of MPC implementation as systems are often very complex and non-linearities should be identified. Modeling processes could be a long task because of identifying all parameters, dynamic expression, non-linearities, etc., and normally some validation tests should be done. During our implementation we avoid all these steps and we assume that taken equations well represents real process.

### 4.3. Optimization parameters

Let's now implement the optimization problem and define all parameters needed by Ipopt. We want to know the command values that will bring system to desired $h_2$ level in an optimal way. That is, we want to know at every moment the values of q that better controls the system. As a result, Ipopt will give us a trajectory of q input flow expressed as a vector $x[i = 0...n-1]$, each $x[i]$ corresponds to a $q(k)$. n is then the prediction horizon of the MPC strategy.

1. Problem dimensions

- number of variables: n = 15 Prediction horizon.
- number of constraints: m = n — We need a constraint at each sample time to define soft changes of command and avoid actuator damages. For each x(i) we impose that -2<= x(i)-x(i)<=2, so number of constraints is n.

2. Problem bounds

- variable bounds : $x^L$ = 0 — A negative input flow is physically impossible. Lower bound is set at 0.

  $x^U$ = 13 — We consider that input can't be higher than 13 $m^3/s$.

- constraint bounds: $g^L$ = -2

  $g^U$ = 2 — As seen before, each constraint $g(x) = x(i) - x(i-1)$ cannot be bigger than 2 $m^3/s$ or smaller than -2 $m^3/s$ to avoid sudden command changes, so constraints bounds are set on that way.

3. Initial starting point

- Initial values for the primal x variables: $x(i)$ = q_applied

  The problem is initialized with the previous $q$ value sent to the plant $q(k-1)$ =q_applied. If the algorithm is on the first sample time ( $k = 0$ ), q_applied=0.

- Initial values for the multipliers (only required for a warm start option):

  Not defined.

4. Problem Structure

- number of nonzeros in the Jacobian of the constraints: nnz_jac_g = 2m-1

  As we will see later, number of nonzeros in the Jacobian of the constraints is related with the number of constraints m as 2m-1.

- number of nonzeros in the Hessian of the Lagrangian function

  Not defined because Quasi Newton method option is set on. No need to implement it.

- sparsity structure of the Jacobian of the constraints

  Nonzero values are in the diagonal and in the subdiagonal of the matrix.

- sparsity structure of the Hessian of the Lagrangian function

Not defined.

5. Evaluation of Problem Functions (Information evaluated using a given point $x, \lambda, \sigma_f$ coming from Ipopt)

- Objective function:

$$\min f = R(q - q\_ref)^2 + Q_1(h_1 - h_1\_ref)^2 + Q_2(h_2 - h_2\_ref)^2$$

- Gradient of the objective $\nabla f(x)$.

It is set by Forward-differences formula

$$\frac{df}{dx_i}(x) \cong \frac{f(x + \varepsilon e_i) - f(x)}{\varepsilon}$$

And implemented with the following algorithm

```
Number xe[n];
double eps=1e-6;
int j;

for (Index i=0; i<n; i++) {
       for (j=0;j<n;j++){
         xe[j]=x[j];
       }
       xe[i]=xe[i]+eps;
       grad_f[i] = (cost(n,xe)- cost(n,x))/eps ;
}
```

Forward-differences implemented code.

Where xe is $\varepsilon e_i$, the epsilon difference added only in the $i$ th component of vector $x$. Cost() is the function name that evaluates $f(x)$ and n is the length of $x$ vector.

- Constraint function values

$$-2 \le g(x) = x(i) - x(i-1) \le 2$$

- Jacobian of the constraints,

$$\nabla g(x)^T = \begin{pmatrix} 1 & & 0 \\ -1 & \ddots & \\ & \ddots & \ddots \\ 0 & -1 & 1 \end{pmatrix}$$

So that number of non-zeros in the jacobian of the constraints is 2m-1.

- Hessian of the Lagrangian function, $\sigma_f \nabla^2 f(x_k) + \sum_{i=1}^{m} \lambda_i \nabla^2 g_i(x_k)$

Not defined. We use Quasi-Newton method

### 4.4. Implemented algorithm

All implemented code is separated in two parts; the first corresponding to the controller and all the MPC structure (simulation, optimization and control), and the second one that is a substitutive of the process as we don't dispose of a real one. That is, this second one only simulates real process. Control algorithm puts together and makes compatible all different parts; model, optimization code and objective function evaluation. We consider four logical levels, each one developed in one different file:

- Main.c
- MyNLP.c
- Cost.c
- Model.c

**Main.c**, is the most important and is where the optimization problem is called repeatedly at each sample time. We have decided to execute the program along 40 sample times, so that Main.c calls to solve the optimization problem also 40 times. In a real case, the execution should never stops and runs continuously to ensure a non-stop control.
**MyNLP.c** has all the information about optimization problem (all parameters described in 4.3. It creates the vector x(i=0,…,n-1) and try iteratively with different values. With these values it calls objective function in Cost.c to get the cost value and determine if the proposed values are the optimal ones.
**Cost.c** has then the objective function and the code to simulate and evaluate model. Effectively, whet Cost.c gets vector x, it simulates system behavior and uses objective function to calculate cost value (how much good is proposed x solution). Cost is sent back to MyNLP.c
**Model.c** has the model and some functions to evaluate the dynamic equations. All these functions are called from Cost.c.

When executing, this program generates a **Results.txt** text file with all the values that need to be plotted. With a Matlab script, this file is loaded and results are plotted in a window. As it is not the purpose of this thesis to implement code for plotting, we were looking for the easyest way to do it, even if it means to use non-open-source software as Matlab.

See appendix 8.1 for complete code.

### 4.5. Tests and results

First of all we show that the implemented program runs according to MPC principle. Below there are 4 different graphs corresponding on optimal solution found at different k instants. We can observe that execution length is 40 sample times and the trajectories are only 15, this trajectories length is the predicted horizon. Note that trajectories starts according with the previous one, the optimization problem at instant k initializes with k-1 values.

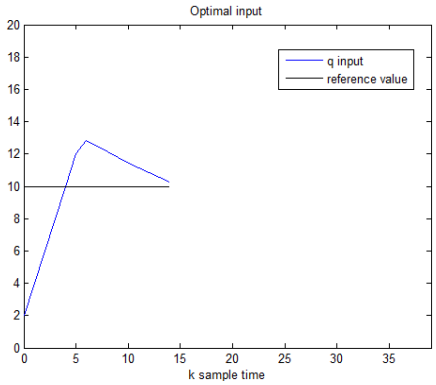



Figure 9. Optimal input trajectory calculated at instant k=0 for a 15 samples horizon.

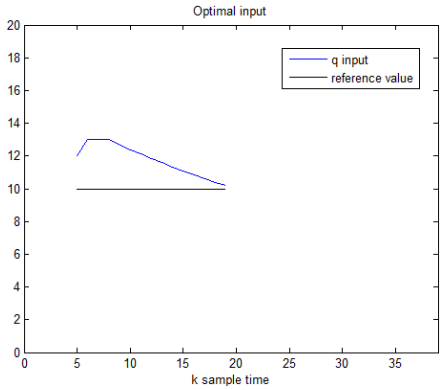



Figure 10. Optimal input trajectory calculated at instant k=5.

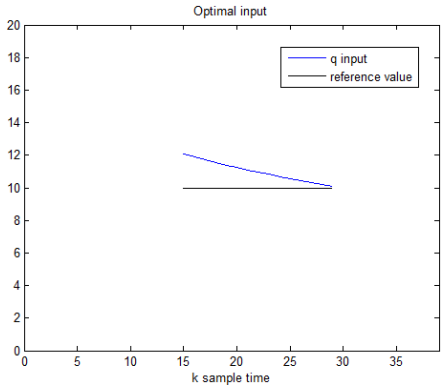



Figure 11. Optimal input trajectory calculated at instant k=15.

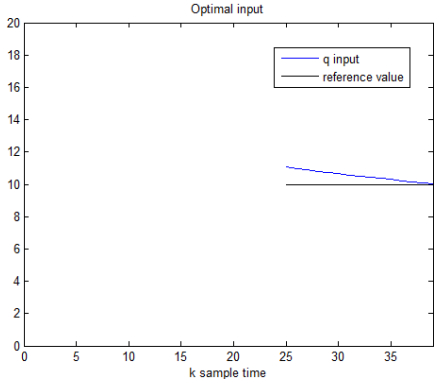



Figure 12. Optimal input trajectory calculated at instant k=25.

First value of each trajectory is sent to the plant, so all the values sent generates also a trajectory along all execution time, see figure 13.

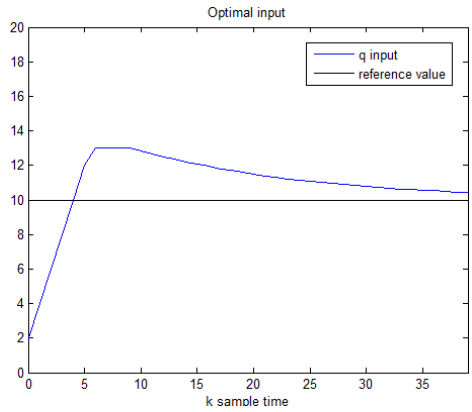



Figure 13. Input sent to the plant along all the execution time. It is generated with the first value of each optimal trajectory calculated.

In figure 14 we compare the predicted behavior of h1 and h2 variables at instant k=0 with real output measured in process. We can see that predicted and real trajectories are not coincident, there's a small error due to the short length of the predicted horizon and integration routines. In fact it is only 15 units so it doesn't consider future values on prediction. Later we will demonstrate that error decreases as predicted horizon length is set longer. Here perturbations are not considered, however in a real case are almost always present.

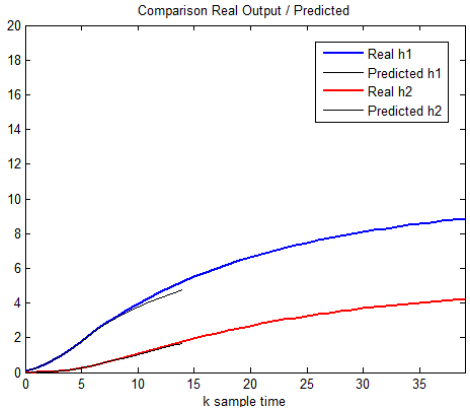



Figure 14. Measured states (colored) compared with predicted states in first optimization (black). Note that trajectories are not coincident.

Up to now, all graphics displayed respected constraints and bounds. Let's have a look what happens when they are not respected.

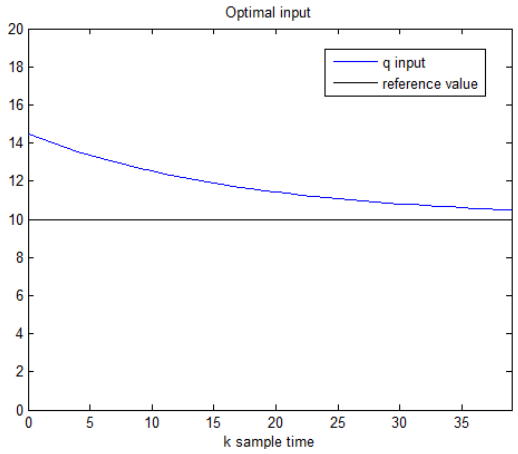



Figure 15. Input sent to the plant when neither constraints nor bounds are defined in problem.

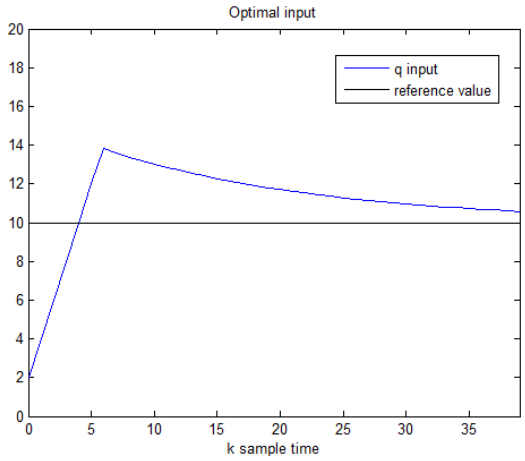



Figure 16. Input sent to the plant with input constraint $-2 \leq q(i) - q(i-1) \leq 2$ but without bounds.

On Figure 15 input q at k=0 is above 14 $m^3/s$ so it does a very sudden change considering that initially it was at $0m^3/s$. To avoid that, on Figure 16 input changes are restricted by a constraint. In addition no graph is bounded so maximal values achieves 14 $m^3/s$ instead of the limit of 13 $m^3/s$ imposed before.

In a real case, model is never perfect and it does not reflect exactly the process. It's easy to make mistakes during modeling and the process can also be subject to interference or perturbation. That's why we will also check how our MPC runs in this case and we will compare it with the ideal case. As we are also using a model to simulate the real process, this model will be the same when testing an ideal case and after we will change it a little bit to have a realistic approach.

I. Ideal case: Control model is exactly the same as process.

We call the same model while optimizing than simulating real process.

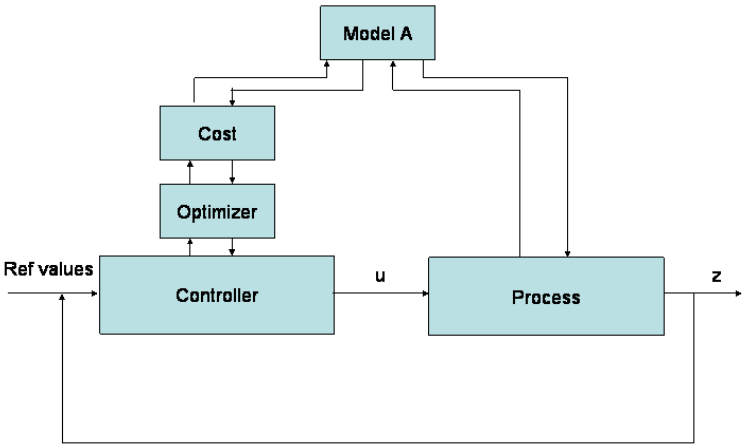



Figure 17. Structure of the algorithm when controller and process are based on the same model. Ideal case.

When using the same model, intuition could make us to expect same results on prediction than in process output, we already demonstrates that it is not true. The length of predicted horizon makes prediction a little bit different from real behavior as it ignores the future values beyond the horizon. In addition, integration routines, external perturbations and model mistakes also casues error. Let's demonstrate now that increasing the horizon, error decreases. We increases from 15 to 80 sample times and we note that goal is achieved before, so control is better.

Figure 18 is only for 15 sample times prediction horizon. h2 is measured at k=50 and its value is h2=4.582.

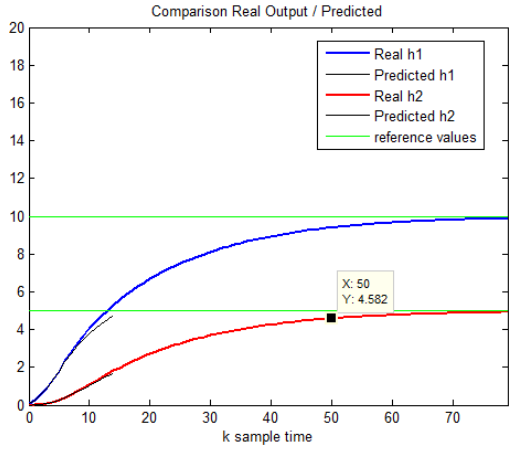
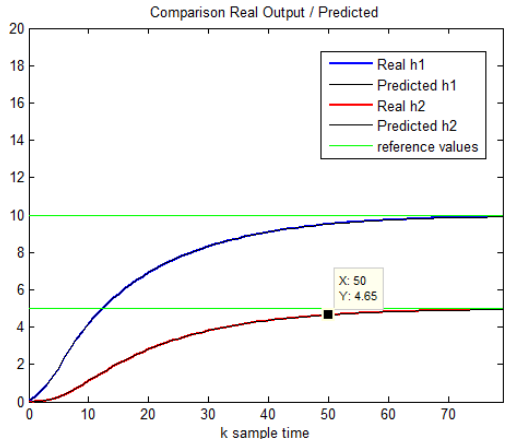



Figure 18. Measured states (in color) copared with predicted states in the first optimization (black). Measured level h2 at k=50: h2=4.582.

We increase now the horizon length to 80 (the same as execution time). Note on figure 19 that real and predicted trajectories are almost overlapped and when k=50 h2=4.65 instead of 4.582 measured before. We agree then that more prediction horizon is long, more quality has the control. However control is

executed continuously without stop while prediction horizon is always finite so there is always error caused by prediction horizon.




Figre 19. Measured states (color) compared with predicted states in the first optimization (black). Here prediction is done with a long horizon, so measured and predicted trajectories are almost coincident.

II. Real case: Control model is not exactly as process

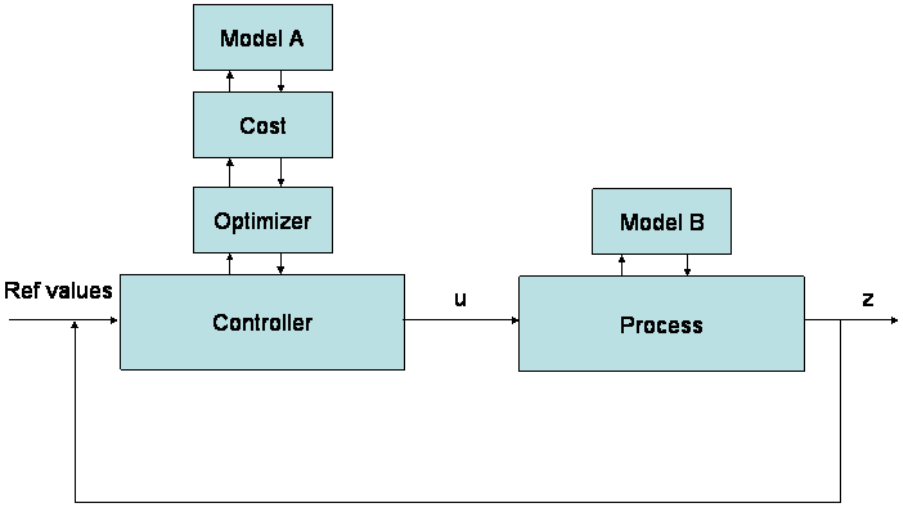



Figure 20. Structure of the algorithm when controller and process are based on different models.

Consider now some modeling mistakes, C2 is set at 2 $m^2$ when real tank 2 surface is 2.1 $m^2$. Consider also that R2 coefficient is set to R2=0.5 when really is R2=0.4. These differences make control more difficult and cause steady state error.

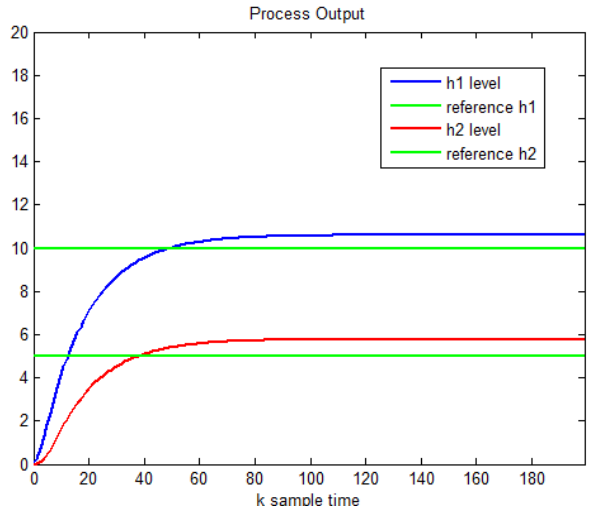



Figure 21. Measured states when control and process models are different. Note that there is a steady state error.

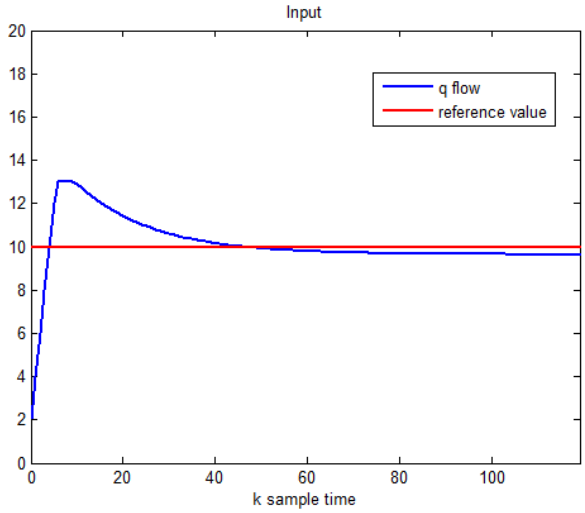



Figure 22. Input when control and process models are different. Trajectory don't achieve reference value.

Note that all state variables and also input have steady state error. That's because when MPC model has mistakes, reference values don't corresponds to a possible state for real process. It is not possible that all variables achieve reference values at the same time. Dinamic equations in model don't reflect real process and previous calculation for setting reference values is also wrong, and objective state is impossible to achieve. As our objective function tries to satisfy as much as possible reference value of all variables, the solution is then a mid point where all variables are close to its reference value but there is no one who really achieved it. We can always set objective function weight parameters different, giving more importance in one variable than others. Is here where performance criteria are set.

## 5. Discussion

Model Predictive Control is an efficient control strategy specially designed for problems subject to a large number of constraints. Is especially interesting its extension to the non-linear world (NMPC) as it allows working with complex systems. In MPC we can set as many performance specifications as needed, related on economical reasons, security, or environmental, what makes this method very useful in practice for a large quantity of applications. It uses a complex computation structure where process behavior is simulated until algorithm finds an optimal control command. Simulating iteratively requires a hard computation, so optimization efficiency is a very important aspect and a line of work for future thesis. Actually this efficiency can be improved by using approximative methods as Forward-difference or Quasi-Newton on derivative calculation. Line search methods will also help to have lightweight computation, more they are better less iterations are needed. It would be interesting to study if it is possible to use MPC in fast processes, where control response must be given fast. Here we only concentrate in software and we talked about approximative methods that simplify computation, but a question remains in the air: Is existent hardware fast enough to run MPC on time? Will real-time exigences be satisfied in all kind of processes?

In this thesis we did both investigation and implementation, so it is a balanced work between theori and practice. Maybe a more detailed work could be done if we developped only one aspect. However I consider more interesting having done both as I saw all method in a global view.

Regarding modeling, open source world offer a large variety of tools and most of them are compatibles each other through Modelica language. A lot of international effort has been done from universities and companies on developing this language and their modeling libraries, so we are in front of a large quantity of knowledge available for everybody. We have seen how a model mistake causes steady state error, what should make us consciously of the existence of gaps in robustness and therefore the importance on modeling. A more detailed modeling study should be done in future works. Together with modeling tools, there is also a set of programs for compiling and optimizing. All this software is a very good help for developers and now for us for implementing the MPC, so all this public knowledge should be preserved and developed for future works.

With the implementation we put in practice MPC method, we also demonstrate the efficiency of all open source tools applied on this type of control. We have also demonstrated a set of properties like dependence between control quality and prediction horizon length: more long is the horizon less error is prodced, however long horizons require more computational effort. We saw the importance to well define the objective function with all criteria weighted according to performance specifications.

## 6. Future Steps

Once the MPC structure is implemented, it would be good to test it with other derivative calculation methods different than Forward-Differences and Quasi-Newton. Try also with some other optimizers different than Ipopt, using some other SQP method. Then do a comparative study, check properties (advantages and disadvantages), quality of solutions and computation time required. Check if due to approximations the quality of results decreases significantly. In addition, a set of tests should be made for checking stability and robustness in all operation rang, detecting instability zones and critical points.

As a model error causes error in control, it would be good to study the possibility to use models with variable coefficients. Adaptive models using the same principle as Kalman filter for model identification, coefficient tuning and correction of steady state error. Measuring the output and observing the error over a certain time could be used to get experience and modify model. It would also be very useful for correcting perturbations. To avoid model mistakes, an acurate invetigation in modeling world would be interesting; look for object oriented programming techniques, parameter identification, non-linearities detection, model validation tests, etc. Even if Modelica is already a well developed language, is always interesting to continue developing new libraries according on technological advances.

In a practical way we should think about how to integrate all this MPC structure into a hardware device. Formulate questions about if it is possible to install MPC in an existing commercial Programmable Controller (like Programmable Interface Controller PIC, Programmable Automation Controller PAC, etc.). Have these devices power enough in computation to run MPC? What are the minimal requirements? Which reliability they have? Would it be safe enough for being used in critical processes? All theory shown in this thesis should be tested on real processes for verifying effectiveness of MPC, robustness and to check compatibility in small hardware structures.

## 7. Conclusion

This thesis is a complete work, where there is both theorethical and practical work so it allowed to learn globally MPC and about open-source tools. We saw that MPC is a powerful control tool but complex at the same time, it requires a big software structure with a perfect and exact model in order to avoid control error. We demonstrate effectiveness of used free software and aproximative methods that simplifies computation. Some questions remain regarding practical application. Future investigation works should continue developping theory but also should check how this method responses in real cases.

# 8. Appendix

## 8.1. Implemented code

### 8.1.1. Main.c

```
//***  Main.c   ***

#include <iostream>
#include "IpIpoptApplication.hpp"
#include "IpSolveStatistics.hpp"
#include "MyNLP.hpp"
#include "jmi.h"

using namespace Ipopt;
int k;  // length of the execution

int main(int argv, char* argc[])
{
  FILE * hFile;
  FILE * iFile;
  FILE * jFile;


  // Create an instance of a Non Linear Problem NLP
  SmartPtr<TNLP> mynlp = new MyNLP();

  // Create an instance of the IpoptApplication
  SmartPtr<IpoptApplication> app = IpoptApplicationFactory();
  app->Options()-> SetStringValue("hessian_approximation","limited-memory");
  ApplicationReturnStatus status;

  // Open files for writing results and plotting
  iFile = fopen( "ZSimulation.txt", "w");
  fclose(iFile);
  jFile = fopen( "ZReal_Input.txt", "w");
  fclose(jFile);
  hFile = fopen( "ZReal_Output.txt", "w");
  fclose(hFile);

  //Execution during 80 sample times
  for (k=0;k<=79;k++){

        // Initialize the IpoptApplication and process the options
        status = app->Initialize();

        if (status != Solve_Succeeded) {
                printf("\n\n*** Error during initialization!\n");
                return (int) status;
        }
        status = app->OptimizeTNLP(mynlp);

        if (status == Solve_Succeeded) {

                // Retrieve some statistics about the solve
                Index iter_count = app->Statistics()->IterationCount();
                Number final_obj = app->Statistics()->FinalObjective();
        }
  }

  system("PAUSE");

  return (int) status;
```

### 8.1.2. MyNLP.c

```
// MyNLP.c
#include <iostream>
#include "MyNLP.hpp"
extern "C" {
#include "cost.h"
}
#include "model.h"
#ifdef HAVE_CSTDIO // for printf
# include <cstdio>
#else
# ifdef HAVE_STDIO_H
#  include <stdio.h>
# else
#  error "don't have header file for stdio"
# endif
#endif

float h1_simulat,h2_simulat,h1_mesured,h2_mesured,u_applied;
float *apuntador_u;
extern int k;
using namespace Ipopt;

/* Constructor. */
MyNLP::MyNLP()
{}
MyNLP::~MyNLP()
{}

bool MyNLP::get_nlp_info(Index& n, Index& m, Index& nnz_jac_g,Index& nnz_h_lag, IndexStyleEnum&
      index_style){

  n = 15; // to input q flow at diferent sample times q[0],q[1],...q[n-1]
  m = n;  // equality constraints g(x)
  nnz_jac_g = 2*m-1;  // nonzeros in the jacobian of the constraints g(x)
  index_style = C_STYLE;  // C index style for row/col entries

  return true;
}

bool MyNLP::get_bounds_info(Index n, Number* x_l, Number* x_u,
                            Index m, Number* g_l, Number* g_u){
  int i;

  for (i=0; i<n; i++) { // lower bounds of the variables
    x_l[i] = 0.0;
  }

  for (i=0; i<n; i++) {  // upper bounds of the variables
              x_u[i] = 13.0;
  }

  // bounds of the constraints
  for (i=0; i<m; i++) {
  g_l[i] = -2.0;
  g_u[i] = 2.0;
  }

  return true;
}


bool MyNLP::get_starting_point(Index n, bool init_x, Number* x,bool init_z, Number* z_L, Number*
      z_U,Index m, bool init_lambda,Number* lambda){

int j;
assert(init_x == true);
assert(init_z == false);
assert(init_lambda == false);
```

```
apuntador_u = &u_applied;

if(k==0){               // initialize to the given starting point
       *apuntador_u = 0;
       h1_mesured=0;
       h2_mesured=0;
}

for (j=0;j<n;j++){
       x[j] = u_applied;
}
  return true;
}

bool MyNLP::eval_f(Index n, const Number* x, bool new_x, Number& obj_value){

       obj_value = cost(n,x);

  return true;
}

bool MyNLP::eval_grad_f(Index n, const Number* x, bool new_x, Number* grad_f){

  Number xe[100];
  double eps=1e-6;
  int j;

  for (Index i=0; i<n; i++) {
         for (j=0;j<n;j++){
         xe[j]=x[j];
         }
       xe[i]=xe[i]+eps;
       grad_f[i] = (cost(n,xe)- cost(n,x))/eps ;
  }
  return true;
}

bool MyNLP::eval_g(Index n, const Number* x, bool new_x, Index m, Number* g)
{
  // return the value of the constraints: g(x).
  int i;

  g[0] = x[0]-u_applied;

  for (i=1; i<m; i++) {
       g[i] = x[i]-x[i-1];
  }

  return true;
}

bool MyNLP::eval_jac_g(Index n, const Number* x, bool new_x,
                       Index m, Index nele_jac, Index* iRow, Index *jCol,
                       Number* values){
int i;

  if (values == NULL) {
    // return the structure of the jacobian
       for (i=0; i<m; i++) {  //setting the structure for the diagonal
              iRow[i] = i; jCol[i] = i;
       }
       for (i=m; i<2*m-1; i++) {  //setting the structure for the sub-diagonal
              iRow[i] = i-m+1; jCol[i] = i-m;
       }
  }else {
    // return the values of the jacobian of the constraints
       for (i=0; i<m; i++) { //filling the values of the diagonal
    values[i] = 1.0;
       }
       for (i=m; i<2*m-1; i++) {
       values[i] = -1.0;
```

```
        }
    }
    return true;
}

bool MyNLP::eval_h(Index n, const Number* x, bool new_x,Number obj_factor, Index m, const Number*
lambda,bool new_lambda, Index nele_hess, Index* iRow,Index* jCol, Number* values){

        return false;
}


void MyNLP::finalize_solution(SolverReturn status,Index n, const Number* x,const Number* z_L,
const Number* z_U,Index m, const Number* g, const Number* lambda,Number obj_value,const poptData*
ip_data,IpoptCalculatedQuantities* ip_cq){

        FILE * jFile;
        int i;

        jFile = fopen( "ZReal_Input.txt", "a");//We write sent input to a txt file for plotting
        fprintf (jFile,"%i %f 10\n",k,x[0]);
        fclose(jFile);

        u_applied=x[0]; //Up to date of sent input for initialize next problem


        escriu_simulacio(n,k,x); //escribim la simul'lació òptima trobada

        //calculem el comportament del sistema real i l'escribim en un fitxer
        calculate_output(n,k,x);

}
```

### 8.1.3. Cost.c

```
#include <iostream>
#include "jmi.h"
#include "model.h"

#define h1_ref 10.0
#define h2_ref 5.0

extern float h1_simulat,h2_simulat,h1_mesured,h2_mesured;

double cost(Index n, const Number* u) {

 jmi_t* jmi;
 double* x;            // states
 double* dx;   // derivatives
 int i,j;              // counters
 double costvalue=0;
 double u_ref=10.0;
 double R[100],Q[2]; //weight vectors

 jmi = (jmi_t*)        calloc(1,sizeof(jmi_t));
 dx  = (double*) calloc(1,N_x*sizeof(double));
 x   = (double*) calloc(1,N_x*sizeof(double));

 jmi_new(&jmi);

 _R1_ = .5;
 _R2_ = .5;
 _C1_ = 2;
 _C2_ = 2;

 // Initial states
 x[0] = h1_mesured;
 x[1] = h2_mesured;
```

```
 //Weight vectors
 Q[0]=1;
 Q[1]=2;
 for (i=0; i<n;i++)
 {
        R[i]=1;
 }

 // Simulate using simple forward differences
 for (i=0; i<n;i++)
 {

    // Set states
    _q_  = u[i];
    _h1_ = x[0];
    _h2_ = x[1];

    // Call model to get derivatives evaluated at these values.
    model_dae_F(jmi,(jmi_ad_var_vec_p) &dx);
    // Update state

        for (j=0; j<N_x; j++){
               x[j] = x[j] + 0.1*dx[j];
        }

        costvalue = costvalue + R[i]*((double)u[i]-u_ref)*((double)u[i]-u_ref) + Q[0]*(x[0] -
h1_ref)*(x[0] - h1_ref) + Q[1]*(x[1] - h2_ref)*(x[1] - h2_ref);

 }



 return costvalue;

}


int calculate_output(Index n,int k,const Number* u){

 FILE * hFile;
 jmi_t* jmi;
 double* x; // states
 double* dx; // derivatives
 int i,j; // Counters

 jmi = (jmi_t*) calloc(1,sizeof(jmi_t));
 dx = (double*) calloc(1,N_x*sizeof(double));
 x = (double*) calloc(1,N_x*sizeof(double));

 jmi_new(&jmi);

 _R1_ = .5;
 _R2_ = .6;
 _C1_ = 2;
 _C2_ = 1,9;

 if (k==0){
        h1_mesured=0;
        h2_mesured=0;
 }

 // Initial states
        x[0] = h1_mesured;
        x[1] = h2_mesured;

        hFile = fopen( "ZReal_Output.txt", "a");

// Simulate using simple forward Differences
               _q_  = u[0];
               _h1_ = x[0];
               _h2_ = x[1];
```

```
// Call model to get derivatives evaluated at these values.
               model_dae_F(jmi,(jmi_ad_var_vec_p) &dx);

// Update state
               for (j=0; j<N_x; j++){
                      x[j] = x[j] + 0.1*dx[j];
               }

       fprintf (hFile,"%i %f %f 10 5\n",k,x[0],x[1]);
       fclose(hFile);
       h1_mesured=x[0];
       h2_mesured=x[1];

return 0;
}



int escriu_simulacio(Index n,int k, const Number* u){

       FILE * hFile;
       jmi_t* jmi;
       double* x; // states
       double* dx; // derivatives
       int i,j; // Counters

       hFile = fopen( "ZSimulation.txt", "a");

       jmi = (jmi_t*) calloc(1,sizeof(jmi_t));
       dx = (double*) calloc(1,N_x*sizeof(double));
       x = (double*) calloc(1,N_x*sizeof(double));

       jmi_new(&jmi);

       _R1_ = .5;
       _R2_ = .5;
       _C1_ = 2;
       _C2_ = 2;

 // These values (parameter values, and also initial states and perhaps other values)
 // should be read from the Modelica-model (via a XML-file?), but I haven't figured which
 // function that does this.


 // Therefore, instead, they are hardcoded here. The defines are defined in the generated C-file.

 // Initial states
       x[0] = h1_mesured;
       x[1] = h2_mesured;

 // Simulate using simple forward Differences
if (k==0){
 for (i=0; i<n;i++)
 {

    _q_  = u[i];
    _h1_ = x[0];
    _h2_ = x[1];

     model_dae_F(jmi,(jmi_ad_var_vec_p) &dx);

       for (j=0; j<N_x; j++){
              x[j] = x[j] + 0.1*dx[j];
       }

       if (hFile == NULL){
              // Error, file not found
       }else{
              // Process & close file
              fprintf (hFile,"%i %i %f %f %f 10\n",k,i,u[i],x[0],x[1]);
```

```
        }
    }
}

        fclose(hFile);

return 0;
}
```

### 8.1.4. Model.c

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <jmi.h>

static const int N_ci = 0;
static const int N_cd = 0;
static const int N_pi = 6;
static const int N_pd = 0;
static const int N_dx = 2;
static const int N_x = 2;
static const int N_u = 1;
static const int N_w = 0;
static const int N_eq_F = 2;
static const int N_eq_R = 0;

static const int N_eq_F0 = 2 + 2;
static const int N_eq_F1 = 3;
static const int N_eq_Fp = 0;
static const int N_eq_R0 = 0 + 0;

#define _R1_ ((*(jmi->z))[jmi->offs_pi+0])
#define _R2_ ((*(jmi->z))[jmi->offs_pi+1])
#define _C1_ ((*(jmi->z))[jmi->offs_pi+2])
#define _C2_ ((*(jmi->z))[jmi->offs_pi+3])
#define _h1_init_ ((*(jmi->z))[jmi->offs_pi+4])
#define _h2_init_ ((*(jmi->z))[jmi->offs_pi+5])
#define _der_h1_ ((*(jmi->z))[jmi->offs_dx+0])
#define _der_h2_ ((*(jmi->z))[jmi->offs_dx+1])
#define _h1_ ((*(jmi->z))[jmi->offs_x+0])
#define _h2_ ((*(jmi->z))[jmi->offs_x+1])
#define _q_ ((*(jmi->z))[jmi->offs_u+0])
#define time ((*(jmi->z))[jmi->offs_t])


#define _ci(i) ((*(jmi->z))[jmi->offs_ci+i])
#define _cd(i) ((*(jmi->z))[jmi->offs_cd+i])
#define _pi(i) ((*(jmi->z))[jmi->offs_pi+i])
#define _pd(i) ((*(jmi->z))[jmi->offs_pd+i])
#define _dx(i) ((*(jmi->z))[jmi->offs_dx+i])
#define _x(i) ((*(jmi->z))[jmi->offs_x+i])
#define _u(i) ((*(jmi->z))[jmi->offs_u+i])
#define _w(i) ((*(jmi->z))[jmi->offs_w+i])
#define _t ((*(jmi->z))[jmi->offs_t])
#define _dx_p(j,i) ((*(jmi->z))[jmi->offs_dx_p + \
  j*(jmi->n_dx + jmi->n_x + jmi->n_u + jmi->n_w)+ i])
#define _x_p(j,i) ((*(jmi->z))[jmi->offs_x_p + \
  j*(jmi->n_dx + jmi->n_x + jmi->n_u + jmi->n_w) + i])
#define _u_p(j,i) ((*(jmi->z))[jmi->offs_u_p + \
  j*(jmi->n_dx + jmi->n_x + jmi->n_u + jmi->n_w) + i])
#define _w_p(j,i) ((*(jmi->z))[jmi->offs_w_p + \
  j*(jmi->n_dx + jmi->n_x + jmi->n_u + jmi->n_w) + i])


static int model_dae_F(jmi_t* jmi, jmi_ad_var_vec_p res) {
    (*res)[0] = jmi_divide(_q_ - ( jmi_divide(_h1_ - ( _h2_ ),_R1_,"Divide by zero: ( h1 - ( h2 )
) / ( R1 )") ),_C1_,"Divide by zero: ( q - ( ( h1 - ( h2 ) ) / ( R1 ) ) ) / ( C1 )") -
(_der_h1_);
```

```
    (*res)[1] = jmi_divide(jmi_divide(_h1_ - ( _h2_ ),_R1_,"Divide by zero: ( h1 - ( h2 ) ) / (
R1 )") - ( jmi_divide(_h2_,_R2_,"Divide by zero: ( h2 ) / ( R2 )") ),_C2_,"Divide by zero: ( ( h1
- ( h2 ) ) / ( R1 ) - ( ( h2 ) / ( R2 ) ) ) / ( C2 )") - (_der_h2_);

        return 0;
}


static int model_init_F0(jmi_t* jmi, jmi_ad_var_vec_p res) {
    (*res)[0] = jmi_divide(_q_ - ( jmi_divide(_h1_ - ( _h2_ ),_R1_,"Divide by zero: ( h1 - ( h2 )
) / ( R1 )") ),_C1_,"Divide by zero: ( q - ( ( h1 - ( h2 ) ) / ( R1 ) ) ) / ( C1 )") -
(_der_h1_);
    (*res)[1] = jmi_divide(jmi_divide(_h1_ - ( _h2_ ),_R1_,"Divide by zero: ( h1 - ( h2 ) ) / (
R1 )") - ( jmi_divide(_h2_,_R2_,"Divide by zero: ( h2 ) / ( R2 )") ),_C2_,"Divide by zero: ( ( h1
- ( h2 ) ) / ( R1 ) - ( ( h2 ) / ( R2 ) ) ) / ( C2 )") - (_der_h2_);
    (*res)[2] = _h1_init_ - (_h1_);
    (*res)[3] = _h2_init_ - (_h2_);

        return 0;
}

static int model_init_F1(jmi_t* jmi, jmi_ad_var_vec_p res) {
   (*res)[0] = 0.0 - _q_;
   (*res)[1] = 0.0 - _der_h1_;
   (*res)[2] = 0.0 - _der_h2_;

        return 0;
}


int jmi_new(jmi_t** jmi) {

        jmi_init(jmi, N_ci, N_cd, N_pi, N_pd, N_dx,
                              N_x, N_u, N_w, N_t_p);

        // Initialize the DAE interface
        jmi_dae_init(*jmi, *model_dae_F, N_eq_F, NULL, 0, NULL, NULL,
                     *model_dae_R, N_eq_R, NULL, 0, NULL, NULL);

        // Initialize the Init interface
        jmi_init_init(*jmi, *model_init_F0, N_eq_F0, NULL,
                      0, NULL, NULL,
                      *model_init_F1, N_eq_F1, NULL,
                      0, NULL, NULL,
                      *model_init_Fp, N_eq_Fp, NULL,
                      0, NULL, NULL,
                      *model_init_R0, N_eq_R0, NULL,
                      0, NULL, NULL);


        return 0;
}
```

## 8.2. Modelica main libraries

1. Libraries for electric, electronic and magnetic components
    a. Analog electric and electronic components
    b. Digital electrical components
    c. Electrical machines
    d. Lumped magnetic networks
    e. Controlled electrical machines:

2. Libraries for mechanical components
    a. 1-dim. mechanical, translational systems
    b. 1-dim. mechanical, rotational systems
    c. 3-dim. mechanical systems
    d. Vehicle dynamics:
    e. Power trains and planetary gearboxes:
    f. Flexible beams and FE-based flexible bodies (with stress stiffening):
    g. Flexible bodies from Nastran, Genesis, and Abaqus:
    h. VDLMotorsports library:
    i. Belt drive systems:
3. Libraries for fluid components
    a. Fluid media
    b. 1-dim. thermo-fluid flow
    c. Simple thermo-fluid pipe flow
    d. Thermal power plants:
    e. Air conditioning systems:
    f. Hydraulics Library:
    g. Pneumatics Library:
    h. CombiPlant Library:
    i. Thermal comfort feeling in air conditioning systems:

4. Libraries for control systems
    a. Input/output blocks
    b. Controller blocks with two levels of detail (continuous and discrete)
    c. Hierarchical state diagrams
    d. Safe hierarchical state diagrams with action blocks

5. Libraries for functions
    a. Functions operating on vectors and matrices
    b. functions operating on strings, streams, files
    c. Functions for analysis and synthesis of continuous and discrete linear systems

## 9. References


[1] J. B. Rawlings. Tutorial Overview of Model Predictive Control. IEEE Control Systems Magazine June 2000.

[2] S.J. Qin & T.A. Badgwell. A survey of industrial model predictive control technology. CEP, ca. 2003.

[3] P. Fritzson. Principles of Object-Oriented Modeling and Simulation with Modelica 2.1. IEE 2004.

[4] Nocedal & Wright. Numerical Optimization. 2006.

[5] Kawajir, Laird & Wächter. Introduction to Ipopt: A tutorial for downloading, installing, and using Ipopt. Massachusetts Institute of Technology.

[6] M.Diehl. An Efficient Algorithm for Optimization in Nonlinear Model Predictive Control of Large-Scale Systems. 2002

[7] W. Li and L.T. Biegler. Newton-Type Controllers for Constrained Nonlinear Processes with Uncertainty. Ind. Eng. Chem. Res. 1990

[8] C.E. García, D.M. Prett and M. Morari. Model predictive control: theory and practice. A survey. Automatica 25 (1989).

[9] R. Findeisen, F. Allgöwer. An Introduction to Nonlinear Model Predictive Control. Institute for Systems Theory in Engineering, University of Stuttgart.

[10] Modelica Association. ModelicaTM - A Unified Object-Oriented Language for Physical Systems Modeling. Tutorial. 2000.

[11] P.E. Orukpe. Basics of Model Predictive Control. Imperial College, London.

[12] F. Martinsen, L.T. Biegler, B. Foss. A new optimization algorithm with application to nonlinear MPC. 2004

[13] R.Ringset, L.Imsland, B.Foss. On Gradient Computation in Single-shooting Nonlinear Model Predictive Control.

[14] L.Biegler, A. Cervantes, A. Wächter. Advances in simultaneous strategies for dynamic process optimization. 2001.

[15] E.F.Camacho, C. Bordons. Model Predictive Control. Springer. Second edition 2003.

[a] www.modelica.org/

[b] www.jmodelica.org

[c] www.openmodelica.org

[d] https://projects.coin-or.org/Ipopt

[e] https://trac.ws.dei.polimi.it/simforge/