Affordable Control Platform with MPC Application

Álan C. E SOUSA[1], Valter J. S. LEITE[1]*, Ignacio RUBIO SCOLA[2]

[1] Centro Federal de Educação Tecnológica de Minas Gerais (CEFET-MG, campus Divinópolis),
Rua Alvares de Azevedo, 400, Divinópolis, 35.503-822, Brazil
[2] Centro Internacional Franco Argentino de Ciencias de la Información y de Sistemas, CIFASIS-CONICET,
Ocampo y Esmeralda, S2000EZP, Rosario, Argentina
acristoffers@ieee.org, valter@ieee.org (**Corresponding author*), ignacio.rubioscola@gmail.com

Abstract: This paper presents a control platform developed to interface with various hardware, allowing the design and rapid implementation even of advanced controllers, both on academic and industrial systems. The code of the controllers is written in the open-source Python language, facilitating the translation of code usually written in commercial software. The proposed platform can use from Arduinos to Programmable Logic Computers (PLCs). Beyond the research and tests on industrial facilities, the simplicity of the proposed platform allows its use also for educational and training purposes. Therefore, the proposed platform can help students focus on system analysis and control theory instead of hardware interfacing issues, while using low cost hardware. Developed in a client-server scheme, the platform can run in affordable computers while taking advantage of high-level mathematical and graphical tools available in Python language, allowing rapid implementation of advanced controllers. The use of this platform is illustrated with an implementation of a model predictive control (MPC) of a level control in a laboratory-scale process. A PLC is used to take the level measures, to dispatch control signals, and also for interlocking secure tasks. The controller runs on a Raspberry Pi computer that communicates with the PLC through an ethernet link.

Keywords: Control platform, MPC, Affordable computer, Optimal control.

1. Introduction

A common workflow in Control Theory research and also for students in this field is to develop the hardware and write software to interface with and control it. However, a lot of effort is spent in writing the same code for different hardware: input/output interfacing; sampling timing; data acquisition, manipulation, storage and presentation; etc. These issues introduce undesirable layers that must be traversed to achieve the primary goal of real-time controller experimentation. Moreover, in industrial facilities, the test of new control strategies can be a challenging task because of the code and, eventually, hardware changes required. This certainly limits the enthusiasm of industrial engineers and managers to evaluate advanced control techniques.

The rise of affordable computers makes it possible to substitute microcontrollers circuits with microprocessor powered devices, which have much more processing power and memory, and makes it easier to interface with complex peripherals, like USB, Ethernet and WIFI.

Due to the presence of a full-stack operating system, it's possible to use various languages to write control software, even interpreted ones, which are not available in embedded devices. A solution which consists of a control platform that carries out those tasks and let the student focus on learning was then proposed.

Thanks to projects like NumPy, SciPy and Python-control, the Python language becomes a good alternative for mathematical computing. Those projects allow the use of well know and tested math libraries to do high-level math with optimized algorithms, close to the metal, while allowing the developer to use high-level APIs and a dynamic language to represent and coordinate the calculations.

The academic and industrial effort to bring simple solutions for advanced control test is not new as it can be seen in [1,2]. In this direction, the Python language was created to be a system language, meaning it would coordinate complex tasks that would be carried out by the system in a simple way. Using Python to write controllers comes in-line with that philosophy, as Python describes the algorithm to be executed and the actual computations are done by low-level optimized math libraries. In this manner complex tasks can be described simply in an easy-to-use language while keeping a good performance.

Controllers that make use of evolutive mechanisms [3], neural networks [4] a model free discrete time neural network control is designed for the trajectory tracking of a kind of nonlinear processes. The introduced control has three main characteristics: (1 or model prediction [5, 6], for example, can be implemented easily using Python, as the language offers various facilities

https://doi.org/10.24846/v27i3y201802

to manipulate generic data, like easy to use lists and dictionaries, generators, comprehensions, and offers good support to object-oriented and functional programming. Advanced control techniques like state-feedback [7] or even those with intensive computational requirements like fuzzy-control[8] and model predictive control [9,10] can then be easily implemented.

A control platform is proposed to circumvent the issues associated with the controller experimentation by yielding an easy and rapid way to evaluate new control strategies or control algorithms. The platform, developed in Python, is composed of two applications working in a client-server structure, which allows modularity. The backend can be installed in affordable computers, like the Raspberry Pi, and execute complex control strategies implemented in a simple yet powerful language. The system was design to aid in system analysis and control design classes, as well as in advanced control tests in industrial environment. A use is to connect an affordable computer to already installed *PLC*s through standard network in order to replace conventional industrial controllers during advanced control evaluation. This scheme is of interest in the implementation of advanced control techniques as they require some processing power and their algorithms are easy to implement in a high-level language like Python.

2. Control Platform

The main contribution of this work is to provide a low-cost solution to the following scenario: a control platform that allows the development and testing of controllers as well as running open-loop tests to aid in learning and to offer a simple and safe way for advanced control tests in industrial environment.

The server, called Moirai [11], is written in Python and makes use of a small library, called *AHIO* [12], to interface with various hardware, like Arduino, Raspberry's *GPIO* and Siemens S7 *PLC* series. This application is responsible for the timing and processing of the supplied controller, as well as for controlling the reading and the writing of sensors and actuators. Mathematical calibration expressions can be given to convert sensors and actuators values before writing or after reading. Interlocking conditions can be used to set outputs when the conditions are met. It can be used, for example, to prevent overflow of a tank. The language was chosen for being a high-level general-purpose language with high quality mathematical support through the SciPy libraries.

The client, named Lachesis [13], communicates with the server through a simple *HTTP API* and works as a front-end, allowing the user to configure the server, save controllers and tests, start/stop tasks, see the resulting graphics while the test is running and to export the data of finished tasks to *JSON*, *CSV* or *MAT* formats, therefore facilitating the data analysis in a number of software. The client-server approach was chosen to allow the backend to use all hardware resources running the controller, while the front-end can be installed in a faster computer to draw the live graphics.

In what follows, we describe each of the three modules developed to provide a solution for affordable computation in control and a simple and safe way to test advanced controllers in industry.

AHIO [12]

AHIO is a python library that exposes a single interface to interact with different hardware. It's designed to work with drivers, so you can choose which driver to use, configure its inputs and outputs, without worrying anymore about driver-specific details. When changing the interfacing hardware for the same application, it suffices to change the preamble to configure the new hardware. The rest of the code stays unchanged.

Moirai [11]

The objective of this application is to interface with the hardware and the database, control the sampling time and code execution, so that the user's provided controller is executed in the right moment and all scanned inputs, outputs and logged variables are saved in the database.

It uses Flask to setup a *JSON* web-service which can read and write all configurations to the database and can return the results of tests, as well as execute CRUD operations on controllers, graphs and system response tests.

Because it needs to execute the controller in a fixed interval with precision, threads were needed. However, Python does not offer a real thread API. All of its thread functionality is emulated by the interpreter. To work around this issue the application was split into three submodules: one for the main process, one for the hardware and one

for the web-server. The main process module will spawn two new processes for the other modules. In this manner the operating system can distribute the processes across the processor's cores and make them run truly in parallel. The main process acts as a link between the other processes, which communicate using pipes to start the execution of a controller or test and to signal the processes to quit.

A test is another feature this application offers. It can set one output of the physical system and log the inputs. This eases the execution of open-loop tests, like step, ramp, stairs, pulse, etc.

The database of choice is MongoDB. It is a NoSQL document store which is fast and stores documents in a *JSON* like format named *BSON*. The objects used to interface with the database are Python dictionaries, lists and basic types, so there is little to no overhead when using the objects or converting them to JSON. As the platform was developed to work with affordable computers, it needs to support the 32-bit ARM architecture, which is the most used in that hardware. Because MongoDB dropped support to its 32-bit versions last year, it was necessary to find an alternative. As the system was already developed to work with MongoDB, it was decided not to change that and, instead, add an alternative to systems where it cannot be installed. The chosen alternative was MySQL, which can be easily installed in most systems. But there's a performance penalty since the data structure returned by the database needs to be converted to and from the *JSON* structure used by the web-service. This overhead is noticeable when retrieving points from the database (used by Lachesis to plot) and when exporting the results to *JSON*, *CSV* or *MAT*. There's no significant difference when saving data during controller/test execution.

Because execution timing is important, two methods of controlling it were tested. In the first method the controller code execution is timed, and the wait-time is calculated as sampling time minus execution time. The second method calculates the next multiple of the sampling time based on when the users commanded the start of the test, and then computes the wait time as that time minus current time, using the *time.time()* function. With 6000 samples, collected by running a controller test for 60 seconds with 0.01 as sampling time, the error mean and standard deviation of the first method were 0.0502 and 0.0286 seconds respectively. The second method was better, with 0.0071 and 0.0005 error mean and standard deviation respectively, showing that the second method has a much smaller timing error than the first one.

Lachesis [13]

The front-end was made separated in order to unload the back-end. In this manner the interface is processed in the user machine and does not add processing time to the controlling hardware. In case a standard computer is used to do both controlling and data processing, there's no noticeable difference, but when using affordable computers to run the controllers it is important to use all resources for this sole end.

Because of this the front-end could be designed using a heavier framework that allows a more appealing interface. By using *Electron*, standard web technology can be used without the browser safety limitations.

Microsoft's *Typescript* was chosen as the programming language because of its type-checking capability. It eases development by catching potential bugs during compilation, what gives some safety and assurance of correctness to the code.

Angular was used to organize the code. It takes care of updating the *DOM*, which unburdens the developer from this repetitive task, and enforces separation of concerns by dividing the application into smaller components. The framework also uses the *RxJs* library, which turns the *JavaScript* development reactive. This eliminates the *callback-hell* problem from which the language suffers and simplify the implementation of concurrency. For a better visual style, the *Angular Material* set of components was used, which implements common components that follows *Google*'s *Material Design* principles.

This application acts as front-end to *Moirai*, so everything *Moirai* can do can be accessed and manipulated by it. The first screen after launching *Lachesis* is the login screen, in which you select which server to connect to. After logging in the *Connection* panel changes to allow the user to disconnect, change *Moirai*'s password or backup/restore the whole database.

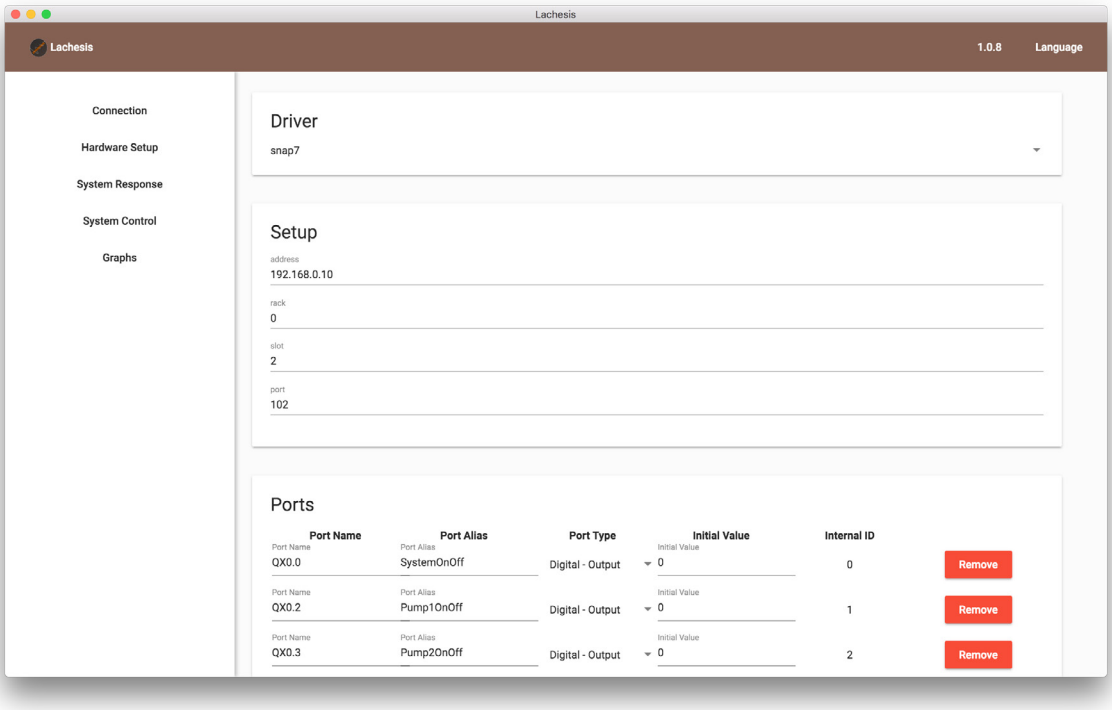

Figure 1. Hardware Setup component

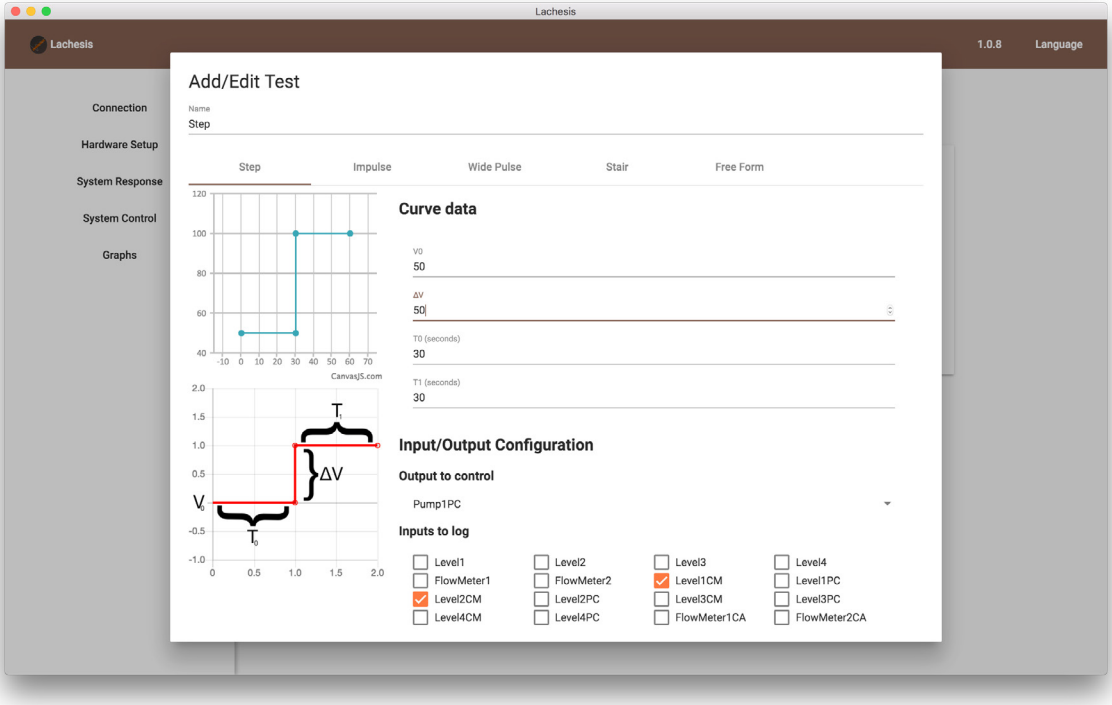

Figure 2. System Response component

The second panel is the *Hardware Setup*, seen in Figure 1, where the user can select to what hardware *Moirai* will connect. This panel also allows the configuration of the connection, the inputs and outputs, which can receive meaningful names. Calibration expressions and interlocking rules can also be provided. The user will see the I/O names (aliases) in all other configuration windows and will be able to access those values by code in the controllers. If the user accesses a

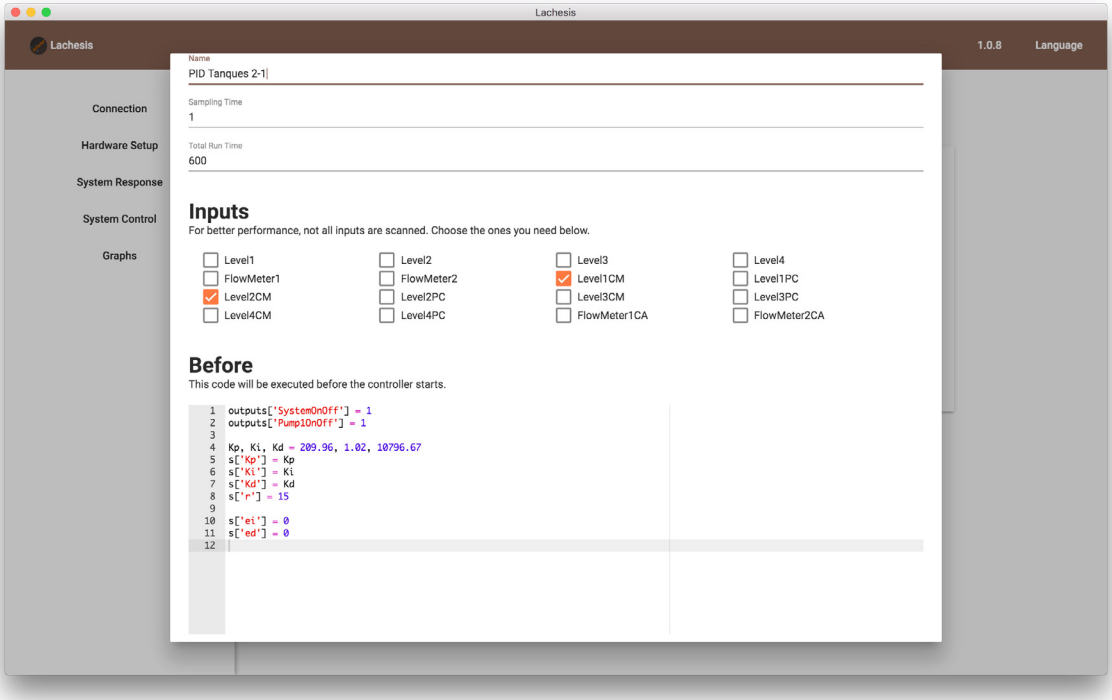

Figure 3. Control component

calibration alias, the returned value is the input reading processed by the expression. If a value is written to it, the value is evaluated by the expression before being written to the output.

The second component is called *System Response*, as shown in Figure 2. It allows the user to create open-loop tests with an easy to use interface. For common types of inputs, like step and stair, the user can simply fill a form and the wave will be generated automatically. If the user needs more control of the wave, he can enter a list of time-point pairs or import them from other software in CSV file format.

The third component is named *System Control* and can be seen in Figure 3. The user can add the controller, written in *Python*, in three text-edit boxes. They are named *Before*, *Controller*, and *After*. *Before* can be used to set constants, initial value of "global" variables, pre-calculate gain matrices, etc., as well as setting outputs to turn the system on. *Controller* will be executed every sampling time and should be used to actually calculate the needed outputs. *After* should be used to turn the system off. If an exception is raised inside Controller, the execution is halted and *After* is executed.

Some special variables exist in those scopes, like *s* (state, a dictionary that holds its value between executions and scopes, acting as a global variable), *log* (a dictionary, every key will act as a new variable) and *t* (holds the current time since test started). The dictionaries *inputs* and *outputs* hold the values of selected inputs (not all are scanned for performance reasons) and allows to set outputs. In Figure 3 you can see the system and the pump 1 being turned on, as well as the PID gains, reference and initial value of the integrator accumulator and derivative last value being set.

Last comes the Graphs component, represented in Figure 4. It allows the user to visualize logged data both during and after a controller/test run. If the test is finished, it also allows to export the points to *JSON*, *CSV* or *MAT* formats. The graphs can be zoomed and panned, and they can be synchronized, so zoom and pan operations applied to one graph also apply to all other graphs. This component also allows to stop a running test. The graphs are updated every second and only the points inserted in the database after the last point already fetched are retrieved. This minimizes both the data transfer and the data processing by the server. Because of the client-server structure the

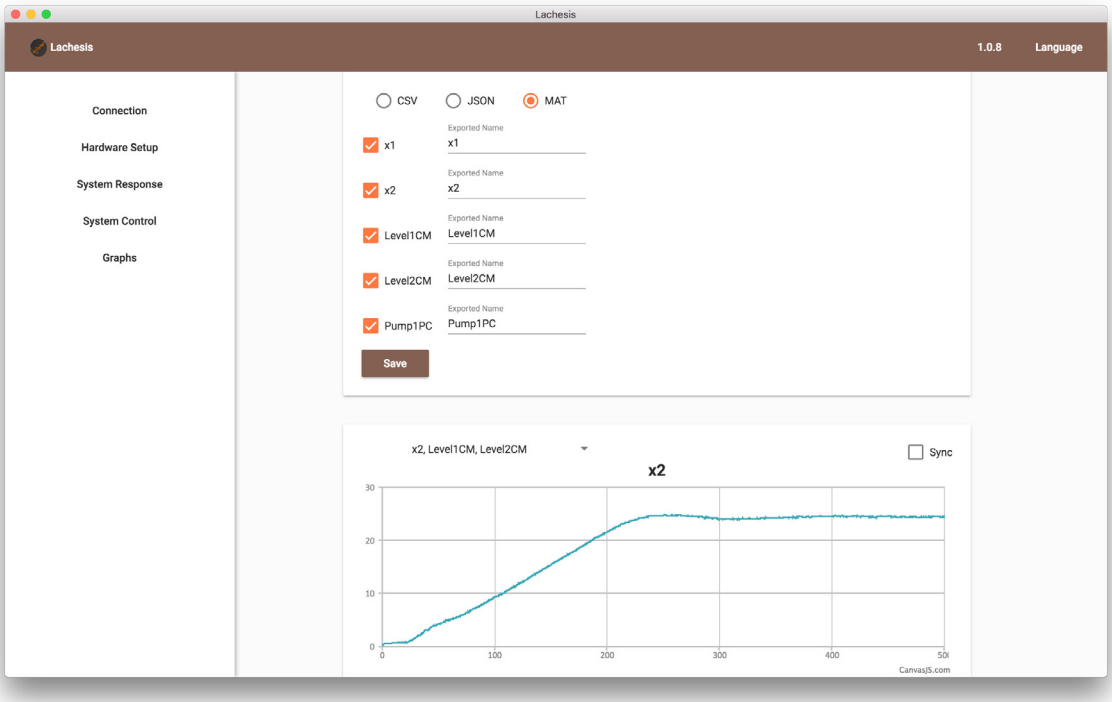

Figure 4. Graphs component

user does not need to keep the client open while the test is running, so simply navigating to another component or closing the application will make the server have more processing power available to the controller.

3. Model Predictive Control

Model Predictive Control is a control technique used in the chemical and petrochemical industries. The main idea behind MPC is to use the model of the controlled process to predict the plant output and use the prediction to optimize the control trajectory [14]. This work considers a very simple form of MPC only to illustrate an application of our main contribution, which is the use of affordable computers with an open source control platform. Albeit simple, this implementation is very computationally intensive.

Then, for a controlled process modeled as:

$$\begin{aligned} x(k+1) &= Ax(k) + Bu(k) \\ y(k) &= Cx(k) + Du(k) \end{aligned} \quad (1)$$

where k is the sample order, $A \in R^{n \times n}$ is the dynamic matrix, $B \in R^{n \times m}$ is the input matrix, $C \in R^{m \times n}$ is the output matrix, x(k) is the system state at sample k, y(k) is the controlled output, and u(k) is control input. Two sets of samples or windows are defined: the prediction and the control horizons with N_p and N_c samples ahead, respectively. Usually N_c is much smaller than N_p.

Because most of the MPC applications refers to regulation processes, an integral action is often introduced by considering a deviation model version of (1). In this case, assuming $\Delta x(k+1) = x(k+1) - x(k)$ and $\Delta u(k+1) = u(k+1) - u(k)$, one can rewrite (1) as [14]

$$\begin{aligned} x(k+1) &= \begin{bmatrix} A & 0 \\ CA & 1 \end{bmatrix} \begin{bmatrix} \Delta x \\ y(k) \end{bmatrix} + \begin{bmatrix} B \\ CB \end{bmatrix} \Delta u(k) \\ y(k) &= [0 \quad \cdots \quad 1] x(k) \end{aligned} \quad (2)$$

Equation (2) is used Np times to predict a set of outputs $Y = [y_k \quad y_{k+1} \quad \cdots \quad y_{k+n}]^T$ which yields

$$Y = Fx(k) + \Phi \Delta U, \quad (3)$$

where $F \in R^{N_p \times n}$ and $\Phi \in R^{N_p \times N_c}$ are given by

$$F = [CA \quad CA^2 \quad CA^3 \quad \cdots \quad CA^{N_p}]^T \quad (4)$$

and

$$\Phi = \begin{bmatrix} CB & \cdots & 0 \\ \vdots & \ddots & \vdots \\ CA^{N_p-1}B & \cdots & CA^{N_p-N_c}B \end{bmatrix}. \quad (5)$$

The MPC objective is to minimize a quadratic cost function

$$J = (R_s - Y)^T (R_s - Y) + \Delta U^T \bar{R} \Delta U \quad (6)$$

with *Rs* being a column vector containing the future setpoints and $\bar{R}$ a matrix that weighs ΔU.

Substituting Y, deriving and equating to zero we have:

$$\Delta U = (\Phi^T \Phi + \bar{R})^{-1} \Phi^T (R_s - Fx(k)) \quad (7)$$

After the computation of the optimal control sequence, only its first component is applied to the process. Next a new state measure or estimation is required, and a new optimal sequence of control signals, ΔU, is computed.

Control Signal Constraint

Using this technique, it's already possible to implement a working controller. However, MPC excels because of its capability to work with constraints in both control signal and output. For that reason, a return to the cost function is needed in order to use the optimization techniques for optimizing it with applied constraints. All constraints must be written in the form:

$$\Delta U_{min} \leq \Delta U \leq \Delta U_{max} \quad (8)$$

Various techniques exist to solve the minimization of the quadratic cost function (6) under (8). The authors decided to use quadratic programming to solve it [14]. Note that a full optimization problem must be solved every sampling time, requiring an additional computation power from the controller CPU. Thus, this control technique is used to illustrate the proposal of the low-cost platform.

For the implementation of such control law, the information of the state vector is needed, but only the system outputs *y(k)* is present. On the other hand, the states and outputs of the system are can be considered corrupted by white Gaussian noises with zero mean, named *w* and *v* respectively:

$$\begin{aligned} x(k+1) &= Ax(k) + Bu(k) + w(k) \\ y(k) &= Cx(k) + Du(k) + v(k) \end{aligned} \quad (9)$$

To handle this case the Kalman filter [15] is used to estimate the state vector and to reduce the covariance of the estimation error:

$$\begin{aligned} \tilde{x}(k+1) &= A\tilde{x}(k) + Bu(k) \\ &+ L\left(y(k) - \left(C\tilde{x}(k) + Du(k)\right)\right) \end{aligned} \quad (10)$$

where $\tilde{x}(k)$ is the state estimation, *L* is the optimal correction gain (in the sense that it minimizes the covariance of the estimation error) computed by

$$L = APC^T (R + CPC^T)^{-1} \quad (11)$$

and *P* is the solution of the following algebraic Riccatti equation (ARE):

$$A(P - PC^T (R + CPC^T)^{-1} CP) A^T + Q = P \quad (12)$$

with *Q* and *R* the covariance matrices of noises *w* and *v* respectively.

After computing *L*, the code to run the MPC controller has to take the following steps at every sample time to compute the control signal:

1. Read the controlled variable and compute the estimated state with (10) and then compute the state variation, Δx.

2. Take the augmented state by appending the last value of the controlled variable to the state variation vector computed in Step 1.

3. Compute *F* and Φ with (4) and (5) respectively, and then $\overline{E} = \Phi^T \Phi + \bar{R}$ and $\overline{F} = \Phi^T (Fx(k) - R_s)$, which are inputs of the QP optimization algorithm.

4. Write the soft constraints on the output, control variable, state and their variations in the form $M\Delta U \leq N$, where *M* and *N* matrices come from the concatenation of the following inequalities:

   a. Constraint on Δu
   $$\begin{bmatrix} -I \\ I \end{bmatrix} \Delta U \leq \begin{bmatrix} -\Delta u_{min} \\ \Delta u_{max} \end{bmatrix}$$

   b. Constraint on *u*
   $$\begin{bmatrix} -C_2 \\ C_2 \end{bmatrix} \Delta U \leq \begin{bmatrix} -u_{min + C_1 u(k-1)} \\ u_{max + C_1 u(k-1)} \end{bmatrix}$$

   c. Constraint on *y*
   $$\begin{bmatrix} -\Phi \\ \Phi \end{bmatrix} \Delta U \leq \begin{bmatrix} -y_{min + Fx(k)} \\ y_{max - Fx(k)} \end{bmatrix}$$

5. Use the QP algorithm to minimize $J = \frac{1}{2} x^T \overline{E} x + x^T \overline{F}$ subject to the constraints in Step 4 to get the optimal control variation vector ΔU. Then implement the control signal $u(k) = u(k-1) + \Delta U_{first\ position}$.

Figure 5. Four tanks interactive system

4. Application

The implementation has entailed a laboratory scale process that consists of four interconnected tanks, each of them with a capacity of 200 liters, and a reservoir with capacity of 1000 liters (see Figure 5). A 1 c.v. pump actioned through a 3-phase inverter controls the inlet flow of the process. A Siemens PLC model Simatic S7 300 is used to get process measures, to dispatch control signals and to perform the interlocking task. All control and measurement consist of 4-20 mA current loops [8,16]. The process allows some different configurations, and, for this application, only tanks T1 and T2 are used, coupled through a fixed manual valve, where the inlet flow arises in tank T1 and the control objective is the level on tank T2. For control purposes, only the level on tank T2, h_2, is measured and controlled.

Through mass balance equations, a linearized model of this process can be obtained as:

$$\dot{x} = \frac{1}{A\,R_{12}}\begin{bmatrix} -1 & 1 \\ 1 & -\left(1+\frac{A\,R_{12}}{A\,R_2}\right) \end{bmatrix} x + \begin{bmatrix} \frac{1}{A} \\ 0 \end{bmatrix} u \qquad (13)$$

$$y = [0 \quad 1]x$$

where is the inlet flow on tank T1, A is the area of the cylindric tanks, R_{12} is the flux resistance between T1 and T2, R_2 is the flux resistance of the output of T2, and h_1 and h_2 are the levels on tanks T1 and T2, respectively. Replacing the numerical values, we have:

$$\dot{x} = \begin{bmatrix} -0.015 & 0.015 \\ 0.015 & -0.027 \end{bmatrix} x + \begin{bmatrix} 331 \times 10^{-6} \\ 0 \end{bmatrix} u \qquad (14)$$

Assuming a N_c = 15 samples, the sampling time of $T = 10s$ (and N_p = 150) was used to implement the controller. Also, to evidentiate the possibilities of this affordable platform, the experiments were repeated with $T = 1s$ (and N_p = 1500). Although this last choice is not the natural one, it leads to a higher computational demand that allows to test the proposed affordable platform. The corresponding F and Φ matrices were computed. The observer gain was calculated by using the Kalman Filter, where the covariance of both states and the variation of the states were calculated based on experimental data from 1000 samples of the output yielding $R = 0.060440$ for the sensor and Q as diagonal matrix with 0.783261 and 0.090986 for the state variation. For code implementations details, see https://bit.ly/2uuwecW.

With those values calculated, the MPC can be used to control the level of T2. Two experiments have been performed changing only the sample time as mentioned before. The output and control signal of the system following a setpoint change and adapting to a disturbance at 600 seconds can be seen in Figure 6. The disturbance is performed by actioning a second pump to add water to tank T2. The controller was run in a Raspberry Pi 3 Model B. The hardware controlling the actuators and sensors is a Siemens S7 PLC, which reads the sensors in analogic inputs and controls the pumps through a frequency invertor. The user only needs to know which PLC ports are connected to what hardware, and to give names to those ports in the Hardware Setup component.

It is clear from the experiment data that the controlled output is more effective for a smaller sample time, with almost none overshooting and shorter settling time. On the other hand, the control

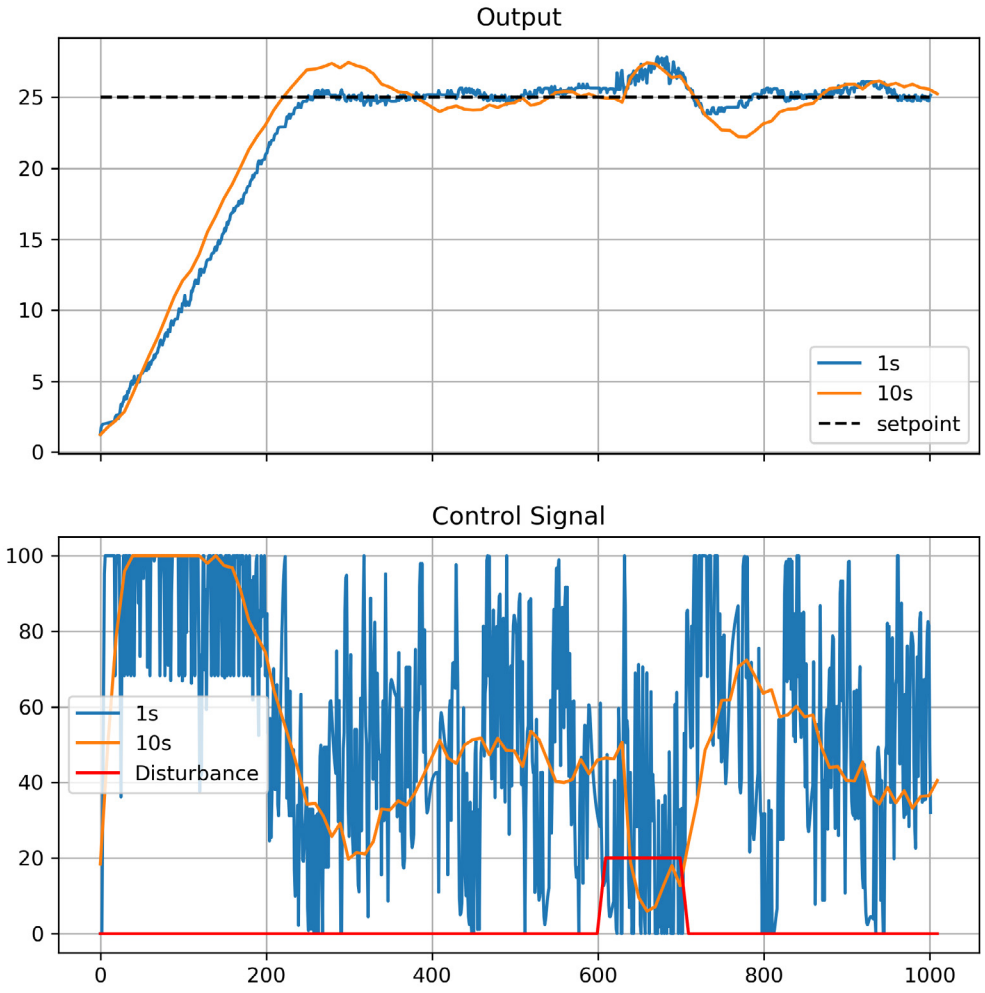

Figure 6. MPC output and control signal

signal has a more significative variance in this case, which may lead to higher maintenance costs. Such an issue can be solved by including a simple first order digital filter in the code. However, the objective of this paper is just to illustrate that a more demanding computational controller can be easily implemented in this affordable platform with a quite simple hardware.

5. Conclusion

An affordable control platform was proposed which allows to run controllers in a low-end computer. It was able to execute a complex, computational expensive controller while saving all readings and writings to a database and estimating states through an observer. The user only needs to fill a form in order to configure the hardware and then he can focus on the system identification, testing and controller development, what gives him more time to study the related concepts. If the student has a non-standard hardware, it's easy to develop a new driver for *AHIO* which can interface with such hardware. The platform will recognize any driver that *AHIO* exports. A real-time control application has been used with MPC to illustrate the potential of the present proposal.

REFERENCES

1. Carlsson, H., Svensson, B., Danielsson, F. & Lennartson, B. (2012). Methods for Reliable Simulation-Based PLC Code Verification, *IEEE Trans. Ind. Informatics*, *8*, 267-278.

2. Schluse, M., Priggemeyer, M., Atorf, L. & Rossmann, J. (2018). Experimentable Digital Twins-Streamlining Simulation-Based Systems Engineering for Industry 4.0, *IEEE Trans. Ind. Informatics*, *14*, 1722-1731.

3. Bazaraa, M. S., Sherali, H. D. & Shetty, C. M. (2006). *Nonlinear programming : theory and algorithms*. Wiley-Interscience.

4. de Jesús Rubio, J. (2018). Discrete time control based in neural networks for pendulums, *Appl. Soft Comput.*, *68*, 821-832.

5. Rawlings, J. B. & Mayne, D. Q. (2009). *Model predictive control : theory and design*. Nob Hill Pub.

6. Boeira, E., Bordignon, V., Eckhard, D. & Campestrini, L. (2018). Comparing MIMO Process Control Methods on a Pilot Plant, *J. Control. Autom. Electr. Syst.*, *29*, 411-425.

7. Barroso, N. F. (2017). *Distributed parameters systems monitoring strategy based on Kalman observers* (in Portuguese). Master thesis at Graduate Program on Electrical Engineering (CEFET-MG). Available at: <goo.gl/oDYCdB>.

8. Lopes, A., Leite, V. & Silva, L. (2018). On the Integral Action of Discrete-time Fuzzy TS Control Under. In *WCCI - World Congress on Computational Intelligence*.

9. Venkatesh, S., Ramkumar, K., Guruprasath, M., Srinivasan, S. & Balas, V. E. (2016). Generalized predictive controller for ball mill grinding circuit in the presence of feed-grindability variations, *Studies in Informatics and Control*, *25*, 29-38.

10. Makhlouf, A., Marhic, B., Delahoche, L., Clérentin, A. & Messaoud, H. (2016). A smart and predictive heating system using data fusion based on the belief theory, *Studies in Informatics and Control*, *25*, 283-292.

11. Sousa, Á. C. (2017). *Moirai*. Available at: <https://github.com/acristoffers/moirai>. Accessed: 23rd May 2018.

12. Sousa, Á. C. (2016). AHIO, *AHIO - Abstract Hardware I/O*. Available at: <https://github.com/acristoffers/ahio>. Accessed: 23rd May 2018.

13. Sousa, Á. C. (2017). *Lachesis*. Available at: <https://github.com/acristoffers/lachesis>. Accessed: 23rd May 2018

14. Wang, L. (2009). Model Predictive Control System Design and Implementation Using MATLAB, *Engineering*. Springer-Verlag London. doi:10.1007/978-1-84882-331-0

15. Kalman, R. E. (1960). A New Approach to Linear Filtering and Prediction Problems, *J. Basic Eng.*, *82*, 35.

16. Oliveira, L., Leite, V., Silva, J. & Gomide, F. (2017). Granular evolving fuzzy robust feedback linearization. In *2017 Evolving and Adaptive Intelligent Systems (EAIS)* (pp. 1-8). IEEE. doi:10.1109/EAIS.2017.7954821.