Departamento de Informática y Automática

UNED

Ph.D. Dissertation

Contributions to the analysis, simulation, and experimentation on event-based PID control systems

Jesús Chacón Sombría
M.Sc. Systems and Control Engineering

E.T.S. de Ingeniería Informática
Universidad Nacional de Educación a Distancia

MADRID, 2014

Department	Informática y Automática
Faculty	E.T.S. de Ingeniería Informática
Dissertation Title	Contributions to the analysis, simulation, and experimentation on event-based PID control systems
Author	Jesús Chacón Sombría
Academic Degree	M.Sc. Systems and Control Engineering Univ. Nacional de Educación a Distancia
Advisors	Dr. Sebastián Dormido Bencomo Dr. José Sánchez Moreno

To my father

Acknowledgements

I would like to thank all the people that have been in my life during these years and that, in one way or another, have contributed to the realization of this thesis. I hope not to forget anyone, and if so, please do not be bothered ;).

First of all, thanks to my advisors José Sánchez and Sebastián Dormido, for giving me the opportunity to do this work, and for their constant support, advise, and guidance. It is difficult to express in words my gratitude for all the time and the experiences I have been able to live during these years in the UNED. Thank you so much.

Also many thanks to Prof. Francisco (Paco) Vázquez, who encouraged me to begin this adventure in Madrid. I would like to thank the whole research team of the Department of Informática and Automática, specially to Sebastián Dormido-Canto, Raquel Dormido-Canto, Natividad Duro, Fernando Morilla, Joaquín Aranda and Jose Manuel Díaz. Thanks to Miguel Angel, Victorino, David, Alejandro, Luis, Juan Carlos, Oscar, Agustín, Daniel, and Ernesto. Thanks to Dictino for his always interesting, but sometimes strange ideas. Thank you, Maria, for helping me with the english and the reviews of the thesis. I am sure we shall win the paddle tournament some day ;). Thank you, Ernesto and Puri, for your friendship. Thanks to Hector and Gonzalo, for sharing your work and ideas with me and for the good times we had before you went back to Chile. And thank you, Pilar, for you are always making our lives easier with your work.

My most sincere gratitude to Prof. Antonio Visioli, for giving me the opportunity to collaborate with his team and for making my stay in Brescia an enjoyable time. Also thanks to Manuel Beschi and Fabrizio Padula. Many thanks to Prof. Luis Yebra, for giving me the opportunity to stay one week in Almería. Thanks to Juan Luis, who helped me with the experiments on the Acurex system.

Thanks to my friends, Javi, Antonio and Mari Carmen, Natalia, Miguel Angel, Adrián, and Gema. And to all people from Kenwakai and Gimnasio Nicaragua.

Thanks to my family, for being there always, Tia Pili, Abuela, Juan, Jose y Mª José, Sandra y Josito. Thanks to Manolo and Amalia, Alberto and Amalia, and Ana.

Thanks to my mother, Mari Carmen, for being such a good friend. Always ;).

Thanks to my father, Jose Antonio, for making me the way I am. I wish you were here.

And last, but not least, thanks, Marián, for your unconditional love and support during these years.

Contents

List of Tables

List of Figures

1 Introduction

Event-based control is a control paradigm which arises naturally as is similar to how people act. The fundamental idea is the concept of event, that is, something that happens in the system and that somehow involves the need for action. This is a significant change compared to other approaches such as analog control, which is updated continuously, or digital control, which is updated periodically with a fixed time base. However, this form of control system is more intuitive because we humans act somewhat like this: we respond to external events that make us to take decissions about how to act. Moreover, while in other control areas as classical control there is a well-known mathematical theory that provides a systematic approach to problem solving, event-based control has traditionally been used based on heuristics or empirical knowledge, to solve particular problems. However, given the interest and contributions of many researchers in the field, with different formalisms and theoretical frameworks, this situation has changed and, although it is still far from reaching the maturity level of other branches of control, there are many important results. For this reason, it is important not only to research in analysis and synthesis techniques for event-based controllers, but there is an increasing need to incorporate these issues into the curriculum of students, maybe not in initial control courses, but for sure in more advanced courses.

Moreover, today Internet is a ubiquitous tool in our lives, through broadband connections available in the workplace and in most households, and even, increasingly, being continuously connected to the network with mobile devices like smartphones or tablets. In the field of education, students use Internet continuously as a source of knowledge, to the detriment of other more traditional media like books. This new way of dealing with the teaching has led educators to exploit the benefits of interactive tools for learning, allowing leverage technology for students with ways to reach things unthinkable in the past.

The use of hands-on laboratories is essential for experimental sciences where students, in addition to have an understanding of the theory related to the experiment, must learn to deal with practical issues, such as saturations in the actuators, limited precission of the sensor, etc. However, the creation and maintenance of labs is expensive, and in many cases it is not affordable by many universities. Besides the cost, the operation of laboratories depends of the availability of the teacher to supervise and be present during the experiments, leading to a restricted schedule and under-utilization of resources. It is therefore becoming more common, especially in open universities, where usually the number of students is greater, to incorporate remote labs that allow students to carry out experiments through a connection from home or from another place, not necessarily located near the lab. Thus, a remote laboratory allows on the one hand, to eliminate the need for the student to move physically to where practices are performed, and on the other hand, to have a greater utilization of resources.

Within the interactive tools, we can also consider the virtual labs that allow students to experiment on a simulated system, which mimics the behavior of a real system. Although the simulation can not capture all the richness of the behavior of the real system, it has other advantages such as to allow to carry out any type of experience without the possibility to provoke physical damage on the hardware. For this reason, they are very suitable for being presented to the student as a first contact with the plant, to acquire knowledge about it. After having passed successfully the first step, with the virtual lab, they can be allowed to experiment with the real system. This, in turn, also lead to a more rational use of resources and a higher degree of concurrency, because even if the access to the remote laboratory is limited, the virtual system is not. Still, the development of virtual and remote laboratories requires an effort. Moreover, it is often the same educator, not necessarily an expert developer, who must sacrifice time devoted to designing experiences and educational material, to cope with problems related to the practical implementation. So, in general, it is desirable to have patterns and development tools to design these laboratories that facilitate the process and reduce the design time. Particularized to the case of event-driven systems, it is possible to abstract features common to these systems and encapsulate them in software componenents. Furthermore, this method stimulates the construction of more robust solutions and benefits from the experience obtained previously by other developers.

In summary, it is important to study event-based control systems both in its theoretical aspects, to obtain results that help in the analysis and design of controllers that solve practical problems in areas where they have proven to have advantages over traditional techniques, as in the case of networked control systems, multi-agent systems, etc. It is also necessary to contribute to the development of tools to build frameworks to experiment on event-based control systems for research on new control algorithms and to use them in education.

1.1 Objectives

The general objective of this Thesis is to investigate, design, and implement event-based PID control systems. The theoretical study of two general event-based structures and the limit cycles associated to these structures is adressed, and also the simplification of the methods to develop experimentation platforms aimed to teach event-based control.

The following specific objectives are considered in this Thesis:

- Study of two general event-based control structures, focusing on the properties (amplitude and period) of the limit cycles presented in these structures.
- Confirmation of the existence of these limit cycles in simulation and real plants.
- Development of software components to allow a rapid development of virtual and remote laboratories, encapsulating the event-based communication between the client and the server applications.
- Implementation of virtual and remote laboratories making use of the developed components, to assess their performance and facility of use.

1.2 Outline and contributions

The organization of this Thesis is as follows:

Chapter 2. In the first part of the chapter, the two control structures explored in this Thesis are described. These schemes correspond to two cases frequently used in control systems which are based on wireless transmission. The rest of the chapter presents the problem of the limit cycles that may appear in this kind of systems and the mathematical framework that is used in the analysis. The main contribution of this chapter is an algorithm that allow to compute the properties of possible limit cycles in a given process when it is controlled by the considered event-based structures.

Chapter 3. A new library of Java classes and EJS elements is presented, to rapidly develop virtual laboratories for teaching control of industrial processes and, in particular, event-based control systems.

Chapter 4. The creation of remote laboratories is discussed focusing on event-based transmissions related issues. A new architecture is presented, based on the use of EJS elements in the client side, which allows a rapid development and guarantees robustness of the applications.

Chapter 5. The first part of the chapter presents a virtual and remote laboratory to control a quadruple tank plant. The second part discusses the development of a Virtual and Remote laboratory that allows to control a Flexible Link plant.

Chapter 6. Conclusions and future lines of works are given.

1.3 Publications

Journal papers:

1. J. Chacón, J. Sánchez, A.Visioli, L.Yebra, S.Dormido. "Characterization of limit cycles for self-regulating and integral processes with PI control and send-on-delta sampling". *Journal of Process Control*, Vol. 23, No. 6, pp. 826-838, 2013. IF: 1.805.

2. D. Chaos, J. Chacón, J.A. López-Orozco, S. Dormido. "Virtual and Remote Robotic Laboratory Using EJS, MATLAB and LabVIEW". *Sensors*, 13, 2, pp. 2595-2612, doi:10.3390/s130202595, 2013. IF: 1.953.

3. J. Chacón, H. Vargas, G. Farias, J. Sánchez, and S. Dormido. "EJS, JIL and LabVIEW: How to build a remote lab in the blink of an eye". *Transactions on Industrial Electronics.* IF: 5.165. (submitted)

Conference papers:

1. J. Chacón, J. Sánchez, S. Dormido. ”Experimental study of an event-based PID controller in a wireless system”. *9th Portuguese Conference on Automatic Control (Controlo 2010)*, Coimbra, pp. 142-147, 2010.

2. J. Chacón, J. Sánchez, S. Dormido, A. Visioli, ”Design of an event-based feed-forward strategy for SOPTD processes”, *50th IEEE Conference on Decision and Control and European Control Conference*, Orlando (FL), pp. 5431-5436, 2011.

3. J. Chacón, J. Sánchez, A. Visioli, S. Dormido. “Decentralised control of a quadruple tank with a decoupled event-based strategy”, *IFAC Conference on Advances in PID Control*, Brescia (I), pp. 424-429, 2012.

4. J. Chacón, J. Sánchez, A. Visioli, S. Dormido. “Analysis of the limit cycles in the PI control of IPD processes with send-on-delta sampling”, *IEEE International Conference on Control Applications*, Dubrovnik (HR), pp. 1609-1614, 2012.

5. D. Chaos, J. Chacón, J.A. López-Orozco, S. Dormido. ”Enseñando robótica movil con laboratorios remotos.” *XXII Jornadas de Automática*, Vigo, pp. 769-774, 2012.

6. J. Chacón, J. Sánchez, L. Yebra, A. Visioli, S. Dormido. “Experimental study of two event-based PI controllers in a solar distributed collector field”, *European Control Conference*, Zurich (CH), pp. 626-631, 2013.

7. J. Chacón, J. Sánchez, A. Visioli, S. Dormido. “Building process control simulations with Easy Java Simulations elements”, *IFAC Symposium on Advances in Control Education, Sheffield (UK)*, pp. 138-143, 2013.

8. J. Chacón, D. Chaos, H. Vargas, G. Farias, J. Sánchez, and S. Dormido. "A new methodology to build remote laboratories with EJS Elements". *Multimedia on Physics Teaching and Learning (MPTL'18)*, Madrid, 2013.

1.4 Research projects

The results obtained in the framework of this dissertation have been supported by different research projects:

- Event-based modeling, simulation, and control (2007-2012). Spanish Ministry of Science and Technology, CICYT (Ref. DPI2007-61068). Participants: UNED (Spain), University of Murcia (Spain). Directed by Prof. Sebastián Dormido Bencomo.

- MACROBIO: Modeling, simulation, control and optimization of photobiorreactors (2012-2014). Spanish Ministry of Economy and Competitiveness, CICYT (Ref. DPI2011-27818-C02-2). Participants: UNED (Spain). Directed by Prof. José Sánchez Moreno.

Part I

PART I: Analysis

2 Analysis of the Event-based Control Schemes

2.1 Introduction

Traditionally, the control systems used in different engineering areas have been based on periodic sampling, for which there is a wide range of mathematical tools and well established theoretical results (for example, [AW96]). On the contrary, event-based sampling schemes appear in a natural way in wireless sensor networks (WSN) where the controller or the sensors send their outputs according to the violation of some condition. Also, event-based strategies have been used for long time in areas such as control of industrial processes [KKLP99], robot path planning [TXB96], and engine control [HJCV94]. However, event-based control has been used mainly in an *ad-hoc* way, due to the lack of theoretical results, which have begun to be available only in the last years (for example [Å08, AB02, HNX07]). Recently, event-based control is also being used in multi-agent [DL12, GDJ$^+$13] and distributed systems [WAJ12, Gui13, GFF$^+$13].

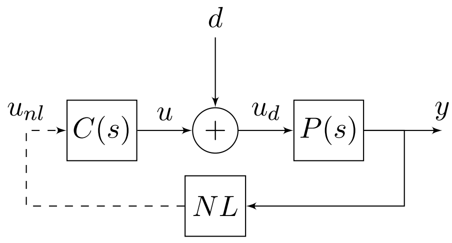
Event-based systems are frequently classified into two types, *event-triggered* systems [YA11], which usually do not have a model of the plant, where the changes measured in the state provoke sporadic updates of the controller, and *self-triggered* systems [Tab07, ASP10], where the controller calculates the time when the next event is triggered, based on an estimation of the state of the plant obtained from a model.

Some works related to event-triggered systems are [WAJ12], where authors develop a distributed event-triggered estimation algorithm for networked systems, or [HD12, HD13], where it is used a periodic event-triggered strategy, which is a variant of event-triggered where the event condition is verified at a constant sampling period. In [MJC12], authors implement a distributed event-triggered control system based on local event generators and prove practical stability. Finally, in [LLJ12], authors analyze the relationship between sampled-data control and two event-triggered strategies, namely, deadband control, in which the controller acts when the process output is outside of a predefined band, and model-based event-triggered control.

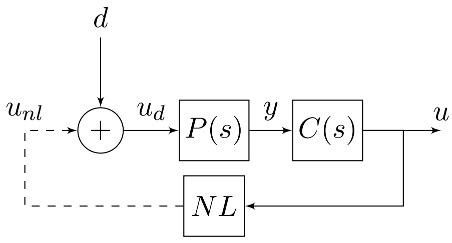
With respect to self-triggered systems, a simple self-triggered sampler for nonlinear systems is presented in [TJ12]. Another self-triggered sampler for nonlinear systems is developed in [AT09, AT10], under the assumption that the closed-loop system with continuous-time control is input-to-state stable (ISS) with respect to measurement errors. In [WL09], authors present a scheme that ensures finite-gain L2 stability of the resulting self-triggered feedback systems.

It is very well-known that Proportional-Integral-Derivative (PID) controllers are widespread in industry, mainly because of their satisfactory performance for many processes and because they are relatively easy to design and tune. Thus, in recent times many authors have addressed the design and implementation of event-based PID controllers [Å99, VK06]. A comprehensive survey of the different methods proposed in the literature for event based PID control can be found in [SVD12].

This work described in this Thesis is focused on *event-triggered* sampling, and in particular on the level crossing or *send-on-delta* sampling [Å99], which consists in taking a new sample when a change greater than a predefined threshold δ is detected



(a) Sampler at the process.



(b) Sampler at the controller.

Figure 2.1: Event-based control schemes. The block diagrams correspond to the two proposed configurations, (a) the event-based sampler at the process output and (b) at the controller output.

in the signal. In practice, this scheme is equivalent to introduce a non-linearity that can be characterized as a quantization of n levels with hysteresis.

The behaviour of a control scheme based on level crossing sampling is studied considering two possible structures, the first one is when the sampler is located after the process output (Figure 2.1a), and the second one after the controller output (Figure 2.1b). Each case represents a configuration of a control scheme based on wireless transmissions, and has different properties. The first case corresponds to a plant with a wireless sensor which takes measures of the process variable and sends them to the controller, physically separated from the sensor but connected to the actuator. The second case is the opposite, where the controller is directly connected to the sensor and the actuator is in other place.

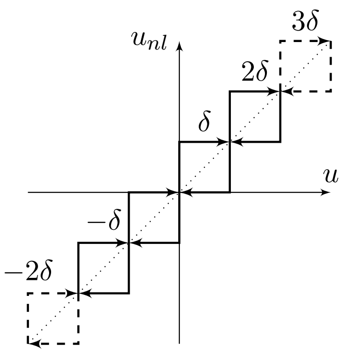
The interest is to characterize with a systematic approach the behaviour of the two event-based control structures with a set of processes such as integrator processes plus time delay process (IPTD), first order processes plus time delay (FOPTD), and second-order processes plus time delay (SOPTD). The analysis is focused on the conditions for the existence of limit cycles, their period and amplitude, the effect of external disturbances, and the wind-up phenomena in the process due to the saturation of the actuators.

There are other works in the literature concerned to the study of limit cycles in event-based systems. In [Å95], authors analyze the existence, properties and stability of limit cycles in relay systems. Related results can also be found in [CA07], where an event-based control scheme with a simple threshold detector is investigated, first in a double integrator and afterwards in the general case. Another approach can be found in [LJ12], where the proposed structure is a model-based event-triggered PI in which the events are generated by the difference between the plant and the model implemented in the controller/sensor. The effect of actuator saturation is also investigated. In [Gon00], a new method to perform the global stability analysis in Piecewise Linear Systems (PLS) is proposed. The method is based on the use of quadratics Lyapunov functions in the switching surfaces. Finally, in [BDSV12a, BDSV12b], authors analize symmetric limit cycles in send-on-delta PI controllers, focusing on the stability of FOPTD processes and proposing tuning rules for this kind of controllers.

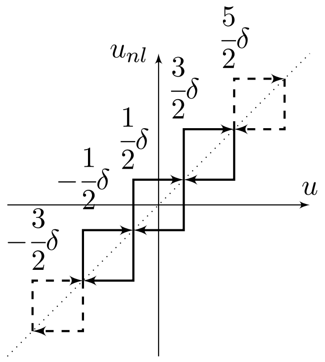
The main contributions of this chapter are the proposition of two general event-based schemes, and a new method for the analysis of the limit cycles that appear in the two presented schemes when they are applied to Linear Time Invariant (LTI) systems with time delay. The method has been applied to study a set of the most common industrial processes, obtaining results that have been confirmed in practice. In particular, a set of experiments carried out in the Distributed Collector Solar Field at the Solar Platform of Almería (PSA) showed the expected behaviour.

2.2 The LTI and SOD sampler blocks

In the event-based control structure used in this work, three elements are present: the multilevel non-linearity with hysteresis represented by the SOD sampler, the process (IPTD, FOPTD or SOPTD), and the PI controller. In order to redefine the event-based system as a PLS, first it is necessary to group the dynamics of the two linear elements in one block, that is, process and controller. Once this is done,



(a) Non-centered sampling.



(b) Centered sampling.

Figure 2.2: Two cases of send-on-delta sampling with different offset.

the redefinition of the system as a PLS is simple since feedbacking the new linear block with the non-linearity is equivalent to introduce a rule to switch between the n systems obtained by the combination of the dynamics of the linear block and the n output levels of the sampler.

Figure 2.2a shows the non-linearity corresponding to the non-centered level crossing with saturation, and Figure 2.2b represents a centered sampling with saturation. The dotted lines in both plots mean that the sampler has a saturation, but in general the levels can be extended in both directions.

2.2.1 The non-linear block: the SOD sampler

The *event-triggered* control systems analysed in this work use a level crossing sampling strategy, where, depending on the sampler location, the sensor sends information to the controller only when the sampled signal crosses certain predefined levels, or the controller sends the new values of the control action to the actuator when there is a significative change with respect to the previous value. The level crossing is considered the event that triggers the capture and the sending of a new sample.

Formally, a SOD sampler can be thought of as a block which has a continuous signal $u(t)$ as input and generates a sampled signal $u_{nl}(t)$ as output, which is a piecewise constant signal with $u_{nl}(t) = u(t_k)$, $\forall t \in [t_k, t_{k+1})$. Each t_k is denoted as *event time*, and it holds $t_{k+1} = \inf\{t|\ t > t_k \wedge |u(t) - u(t_k)| \geq \delta\}$, where δ is the sampling threshold, i.e. the minimum change that triggers the taking of a new sample. The previous condition holds for all t_k, except for t_0, which is assumed to be the time instant when the block is initialized as $u_{nl}(t_0) = u(t_0)$.

In addition, this signal can be saturated due to the limitations in the sensors or in the actuators. As said before, the pattern resulting from sampling a signal with a SOD sampler can be described as a non-linearity of n levels with hysteresis. It can be characterized as

$$u_{nl}(t) = \begin{cases} (i+\alpha+1)\delta & \text{if } u(t) \geq (i+1)\delta \wedge u_{nl}(t^-) = (i+\alpha)\delta \\ (i+\alpha)\delta & \text{if } u(t) \in ((i+\alpha-1)\delta, (i+\alpha+1)\delta) \wedge u_{nl}(t^-) = (i+\alpha)\delta \\ (i+\alpha-1)\delta & \text{if } u(t) \leq (i-1)\delta \wedge u_{nl}(t^-) = (i+\alpha)\delta \end{cases} \tag{2.1}$$

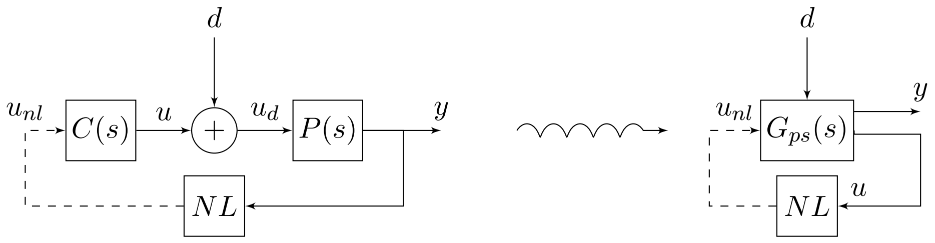
where $\alpha \in [0, 1)$ is the offset with respect to the origin, $i \in \mathbb{Z}$ if the sampler is without saturation, and $i \in [i_{min}, i_{max}]$ when the sampler is with saturation.

Depending on the initial value, the non-linearity introduced could have an offset with respect to the origin, $\alpha = u(t_0) - i\delta$, where $i = \lfloor u(t_0)/\delta \rfloor$. The level crossing sampling with offset is formally defined in the following paragraphs.

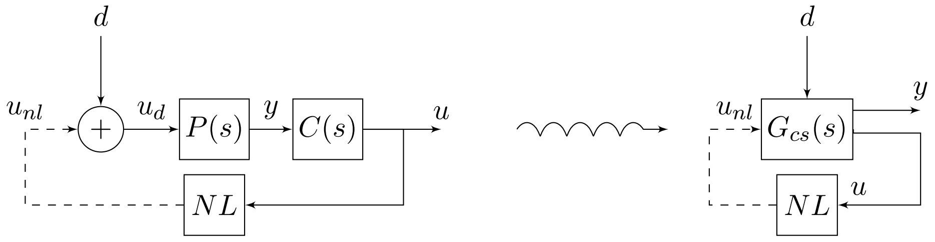
Definition 1. *Let $T = \{t_0, ..., t_n\}$, with $t_k \in \mathbb{R}$ and $t_{k-1} < t_k$, be a set of sampling times, and $U = \{u(t_0), ..., u(t_n)\}$ a set of samples. Thus, U is a level crossing sampling of $u(t)$ if, and only if, $|u(t_k) - u(t_{k-1})| = \delta$, $k = 0, 1, ..., n$.*

Note that every sample can be expressed as $y_s(t_k) = (i_k + \alpha)\delta$, where $i_k \in \mathbb{Z}$ is the crossed level by the input signal $y(t)$, and $\alpha \in [0, 1) \subset \mathbb{R}$ is the sampling offset which depends on the initial sample, $y(t_0)$.

Definition 2. *The order of a non-linearity with hysteresis is defined by subtracting 1 to the number of crossing levels.*



(a) Sampler at the process output.



(b) Sampler at the controller output.

Figure 2.3: Event-based control schemes. The block diagrams on the left correspond to the two proposed configurations, (a) the event-based sampler at the process output and (b) at the controller output. On the right, the continuous dynamics have been grouped in one block to simplify the analysis. Solid lines represent a continuous data flow, while dotted lines mean a discontinuous data flow.

For example, a two-level non-linearity with hysteresis and zero offset owns three crossing levels, that is, $-\delta$, 0, and δ. It must be noticed that if the non-linearity had zero offset the number of levels would be always even, and odd in the opposite case.

2.2.2 The linear blocks: the process and the controller

As Figure 2.3 shows, the linear dynamics of the process and the PI controller can be joined in two different ways by placing the non-linear block at the process output (Figure 2.3a) or at the controller output (Figure 2.3b). Depending on the combination, two different linear blocks G_{ps} and G_{cs} are obtained and, once the loop is closed with the sampler, two PLS are produced with different limit cycles features. The dynamics of the G_{ps} and G_{cs} blocks will be represented by the augmented state

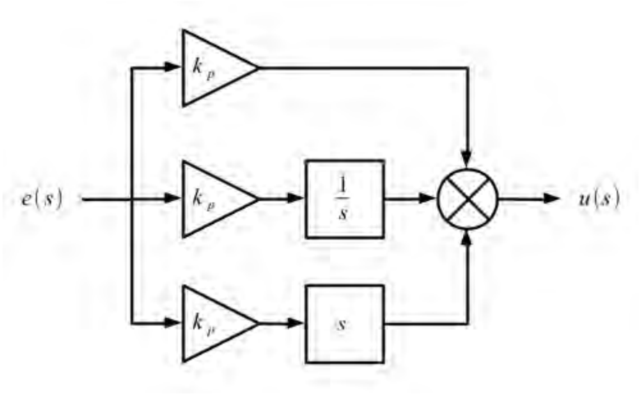


Figure 2.4: Block diagram of the PID controller in the parallel form.

matrices obtained by combining process and controller.

2.2.2.1 State-space representation of the controller

Though there are different implementations of the PID algorithm in literature, all of them are esentially equivalent. In this work, the parallel form of the PID algorithm (see Figure 2.4) is considered. It can be represented by the transfer function,

$$u(s) = \left(k_p + \frac{k_d}{s} + k_d s\right) e(s). \tag{2.2}$$

The PID transfer function has a high gain for high frequencies, due to the derivative term. To avoid problems with noisy signals, in practical implementations it is common to filter the derivative with a first-order filter. With this consideration, the resulting transfer function is,

$$u(s) = \left(k_p + \frac{k_d}{s} + \frac{k_d s}{1 + \frac{k_d}{kN} s}\right) e(s). \tag{2.3}$$

Let us start obtaining the matrices corresponding to the continuous case, that is, without considering the SOD sampler. No delays are considered now. Assume that the process $P(s)$ and the controller $C(s)$ are described by the time-invariant state-space systems

$$\begin{aligned} P(s) &\sim \begin{cases} \dot{x}_p(t) = A_p x_p(t) + B_p(u(t) + d(t)) \\ y(t) = C_p x_p(t) \end{cases} \\ C(s) &\sim \begin{cases} \dot{x}_c(t) = A_c x_c(t) + B_c e(t) \\ u(t) = C_c x_c(t) + D_c e(t), \end{cases} \end{aligned} \tag{2.4}$$

where $x_p \in \mathbb{R}^m$ is the state of the process, $x_c \in \mathbb{R}^n$ is the state of the controller, A_p is non-singular, and $e(t) = r(t) - y(t)$ is the control error.

Combining the two parts of (2.4), the whole system is,

$$\begin{aligned} \begin{bmatrix} \dot{x}_p(t) \\ \dot{x}_c(t) \end{bmatrix} &= \begin{bmatrix} A_p - B_p D_c C_p & B_p C_c \\ -B_c C_p & A_c \end{bmatrix} \begin{bmatrix} x_p(t) \\ x_c(t) \end{bmatrix} + \begin{bmatrix} B_p D_c & B_p \\ B_c & 0 \end{bmatrix} \begin{bmatrix} r \\ d \end{bmatrix} \\ \begin{bmatrix} y(t) \\ u(t) \end{bmatrix} &= \begin{bmatrix} C_p & 0 \\ -D_c C_p & C_c \end{bmatrix} \begin{bmatrix} x_p(t) \\ x_c(t) \end{bmatrix} + \begin{bmatrix} 0 & 0 \\ D_c & 0 \end{bmatrix} \begin{bmatrix} r \\ d \end{bmatrix}. \end{aligned} \tag{2.5}$$

2.2.2.2 Sampling the process variable: the G_{ps} block

The introduction of the SOD sampler in the PID control loop modifies the expressions presented in the previous section. Depending on where the sampler is placed, either the controller or the process input only changes at certain times.

To include the effect of the SOD sampler in (2.4), a new variable is introduced: u_{nl}, the non-linear output of the sampler. At the sampling instants t_k, $u_{nl} := y(t_k)$ if the sampler is at the process variable, and $u_{nl} := u(t_k)$ in the opposite case. The expressions corresponding to the control loop with the SOD sampler are obtained introducing u_{nl} in (2.4), which yields,

$$\begin{bmatrix} \dot{x}_p \\ \dot{x}_c \end{bmatrix} = \begin{bmatrix} A_p & B_p C_c \\ 0 & A_c \end{bmatrix} \begin{bmatrix} x_p \\ x_c \end{bmatrix} + \begin{bmatrix} -B_p D_c & B_p D_c & B_p \\ -B_c & B_c & 0 \end{bmatrix} \begin{bmatrix} u_{nl} \\ r \\ d \end{bmatrix}$$

$$\begin{bmatrix} y \\ u \end{bmatrix} = \begin{bmatrix} C_p & 0 \\ 0 & C_c \end{bmatrix} \begin{bmatrix} x_p \\ x_c \end{bmatrix} + \begin{bmatrix} 0 & 0 & 0 \\ -D_c & D_c & 0 \end{bmatrix} \begin{bmatrix} u_{nl} \\ r \\ d \end{bmatrix}. \tag{2.6}$$

2.2.2.3 Sampling the control variable: the G_{cs} block

The procedure to obtain the augmented state matrices of the G_{cs} block is similar to the previous case but taking into account that now the input to the controller $u(t)$ is the process output $y(t)$, and the input to the process is the signal resulting of adding the disturbance $d(t)$ to an input named $u_s(t)$. The resulting system $G_{cs}(s)$ is

$$\begin{aligned} \begin{bmatrix} \dot{x}_p \\ \dot{x}_c \end{bmatrix} &= \begin{bmatrix} A_p & 0 \\ -B_cC_p & A_c \end{bmatrix} \begin{bmatrix} x_p \\ x_c \end{bmatrix} + \begin{bmatrix} B_p & 0 & B_p \\ 0 & B_c & 0 \end{bmatrix} \begin{bmatrix} u_{nl} \\ r \\ d \end{bmatrix} \\ \begin{bmatrix} y \\ u \end{bmatrix} &= \begin{bmatrix} C_p & 0 \\ -D_cC_p & C_c \end{bmatrix} \begin{bmatrix} x_p \\ x_c \end{bmatrix} + \begin{bmatrix} 0 & 0 & 0 \\ 0 & D_c & 0 \end{bmatrix} \begin{bmatrix} u_{nl} \\ r \\ d \end{bmatrix}. \end{aligned} \tag{2.7}$$

To simplify the analysis of the previous expressions, from now on the setpoint is assumed to be null, i.e. $r = 0$. This is done without loss of generality, as shown in the following Proposition.

Proposition 1. *Consider the systems,*

$$G_{ps}(s) \sim \begin{cases} \dot{x}_{ps} = A_{ps}x_{ps} + B_{ps}u_{ps} \\ y_{ps} = C_{ps}x_{ps} + D_{ps}u_{ps}, \end{cases} \tag{2.8}$$

where

$$A_{ps} = \begin{bmatrix} A_p & B_pC_c \\ 0 & A_c \end{bmatrix} \quad B_{ps} = \begin{bmatrix} -B_pD_c & B_p \\ -B_c & 0 \end{bmatrix}$$

$$C_{ps} = \begin{bmatrix} C_p & 0 \\ 0 & C_c \end{bmatrix} \quad D_{ps} = \begin{bmatrix} 0 & 0 \\ -D_c & 0 \end{bmatrix}, \tag{2.9}$$

and

$$G_{cs}(s) \sim \begin{cases} \dot{x}_{cs} = A_{cs}x_{cs} + B_{cs}u_{cs} \\ y_{cs} = C_{cs}x_{cs}, \end{cases} \tag{2.10}$$

where

$$A_{cs} = \begin{bmatrix} A_p & 0 \\ -B_cC_p & A_c \end{bmatrix} \quad B_{cs} = \begin{bmatrix} B_p & B_p \\ 0 & 0 \end{bmatrix}$$

$$C_{cs} = \begin{bmatrix} C_p & 0 \\ -D_cC_p & C_c \end{bmatrix}.$$

These systems are equivalent to (2.6) and (2.7). Therefore, the setpoint can be assumed to be null without loss of generality.

Proof. Consider that the sampler is placed at the process output. Note that defining $(\bar{x}_p, \bar{x}_c, \bar{u}, \bar{y}, \bar{u}_{nl}, \bar{d}) := (x_p, x_c, u, y, u_{nl} - r, d)$, the system can be equivalently written as (2.6). Now, consider that the sampler is placed at the controller output. Defining $(\tilde{x}_p, \tilde{x}_c, \tilde{u}, \tilde{y}, \tilde{u}_{nl}, \tilde{d}) := (x_p - \frac{C_p^T}{\|C_p\|^2}r, x_c, u, y - r, u_{nl} - \frac{B_p^T A_p C_p^T}{\|B_p\|^2\|C_p\|^2}r, d)$, the system can be written equivalently as (2.7). In both cases, the introduced variables only differ from the original by a constant value. Therefore, it can be assumed without loss of generality that $r = 0$. To make the notation more clear, from now on the variables are written without the symbols ˜ and ¯.

□

2.2.3 The P, I, PI, PD, and PID controllers

In this section, the expressions (2.6) and (2.7) are particularized to the most frequently forms of the PID controller, to serve as a reference for the analysis done in the rest of the thesis. Though the PI is considered in most of the examples, because it is the most extended controller, the propositions and algorithms presented in this chapter are in general form, so they can be applied to any of the enumerated cases by choosing the approppriate matrices.

2.2.3.1 Proportional controller (P)

The proportional or P controller is a controller with a feedback based only in the current error of the process. Its associated state space matrices are $A_c = B_c = C_c = 0$, and $D_c = k_p$. Thus,

$$\begin{aligned} A_{ps} &= \begin{bmatrix} A_p & 0 \\ 0 & 0 \end{bmatrix} & B_{ps} &= \begin{bmatrix} -k_pB_p & B_p \\ 0 & 0 \end{bmatrix} \\ C_{ps} &= \begin{bmatrix} C_p & 0 \\ 0 & 0 \end{bmatrix} & D_{ps} &= \begin{bmatrix} 0 & 0 \\ -k_p & 0 \end{bmatrix}, \end{aligned} \tag{2.11}$$

and,

$$\begin{aligned} A_{cs} &= \begin{bmatrix} A_p & 0 \\ 0 & 0 \end{bmatrix} & B_{cs} &= \begin{bmatrix} B_p & B_p \\ 0 & 0 \end{bmatrix} \\ C_{cs} &= \begin{bmatrix} C_p & 0 \\ -k_pC_p & 0 \end{bmatrix}. \end{aligned} \tag{2.12}$$

2.2.3.2 Integral controller (I)

The transfer function of the integrator or I controller is $C(s) = k_i e(s)$, therefore one of its possible representations in the state space is given by $A_c = 0$, $B_c = 1$, $C_c = k_i$, and $D_c = 0$. Thus,

$$\begin{aligned} A_{ps} &= \begin{bmatrix} A_p & k_iB_p \\ 0 & 0 \end{bmatrix} & B_{ps} &= \begin{bmatrix} 0 & B_p \\ -1 & 0 \end{bmatrix} \\ C_{ps} &= \begin{bmatrix} C_p & 0 \\ 0 & k_i \end{bmatrix}, \end{aligned} \tag{2.13}$$

and,

$$\begin{aligned} A_{cs} &= \begin{bmatrix} A_p & 0 \\ -C_p & 0 \end{bmatrix} \quad B_{cs} = \begin{bmatrix} B_p & B_p \\ 0 & 0 \end{bmatrix} \\ C_{cs} &= \begin{bmatrix} C_p & 0 \\ 0 & k_i \end{bmatrix}. \end{aligned} \tag{2.14}$$

2.2.3.3 Proportional-Integral controller (PI)

The PI controller state-space matrices are $A_c = 0$, $B_c = 1$, $C_c = k_i$, and $D_c = k_p$. Thus,

$$\begin{aligned} A_{ps} &= \begin{bmatrix} A_p & k_i B_p \\ 0 & 0 \end{bmatrix} \quad B_{ps} = \begin{bmatrix} -k_p B_p & B_p \\ -1 & 0 \end{bmatrix} \\ C_{ps} &= \begin{bmatrix} C_p & 0 \\ 0 & k_i \end{bmatrix} \quad D_{ps} = \begin{bmatrix} 0 & 0 \\ -k_p & 0 \end{bmatrix}, \end{aligned} \tag{2.15}$$

and, when the sampler is at the control variable,

$$\begin{aligned} A_{cs} &= \begin{bmatrix} A_p & 0 \\ -C_p & 0 \end{bmatrix} \quad B_{cs} = \begin{bmatrix} B_p & B_p \\ 0 & 0 \end{bmatrix} \\ C_{cs} &= \begin{bmatrix} C_p & 0 \\ -k_p C_p & k_i \end{bmatrix}. \end{aligned} \tag{2.16}$$

2.2.3.4 Proportional-Derivative controller (PD)

The PD controller is $A_c = -n\frac{k_p}{k_d}$, $B_c = n\frac{k_p}{k_d}$, $C_c = k_d$, and $D_c = k_p(1+n)$. Thus,

$$\begin{aligned} A_{ps} &= \begin{bmatrix} A_p & k_d B_p \\ 0 & -n\frac{k_p}{k_d} \end{bmatrix} \quad B_{ps} = \begin{bmatrix} -(1+n)k_p B_p & B_p \\ -n\frac{k_p}{k_d} & 0 \end{bmatrix} \\ C_{ps} &= \begin{bmatrix} C_p & 0 \\ 0 & k_d \end{bmatrix} \quad D_{ps} = \begin{bmatrix} 0 & 0 \\ -(1+n)k_p & 0 \end{bmatrix}, \end{aligned} \tag{2.17}$$

and, when the sampler is at the control variable, the expressions are,

$$\begin{aligned} A_{cs} &= \begin{bmatrix} A_p & 0 \\ -n\frac{k_p}{k_d}C_p & A_c \end{bmatrix} \qquad B_{cs} = \begin{bmatrix} B_p & B_p \\ 0 & 0 \end{bmatrix} \\ C_{cs} &= \begin{bmatrix} C_p & 0 \\ -(1+n)k_pC_p & k_d \end{bmatrix}. \end{aligned} \tag{2.18}$$

2.2.3.5 PID with derivative filter

The PID controller with derivative filter has $A_c = \begin{bmatrix} 0 & 0 \\ 0 & -n\frac{k_p}{k_d} \end{bmatrix}$, $B_c = \begin{bmatrix} 1 \\ n\frac{k_p}{k_d} \end{bmatrix}$, $C_c = \begin{bmatrix} k_i & k_d \end{bmatrix}$, and $D_c = (1+n)k_p$. Thus,

$$\begin{aligned} A_{ps} &= \begin{bmatrix} A_p & B_pk_i & B_pk_d \\ 0 & 0 & 0 \\ 0 & 0 & -n\frac{k_p}{k_d} \end{bmatrix} \qquad B_{ps} = \begin{bmatrix} -B_p(1+n)k_p & B_p \\ -1 & 0 \\ -n\frac{k_p}{k_d} & 0 \end{bmatrix} \\ C_{ps} &= \begin{bmatrix} C_p & 0 & 0 \\ 0 & k_i & k_d \end{bmatrix} \qquad D_{ps} = \begin{bmatrix} 0 & 0 \\ -(1+n)k_p & 0 \end{bmatrix}, \end{aligned} \tag{2.19}$$

where $A_p \in \mathbb{R}^{n\times n}$, and, with $m = n+2$, $A_{ps} \in \mathbb{R}^{m\times m}$, $B_{ps} \in \mathbb{R}^{m\times 2}$, $C_{ps} \in \mathbb{R}^{2\times m}$, and $B_{ps} \in \mathbb{R}^{2\times 2}$.

When the sampler is at the control variable, the expressions are,

$$\begin{aligned} A_{cs} &= \begin{bmatrix} A_p & 0 & 0 \\ -C_p & 0 & 0 \\ -n\frac{k_p}{k_d}C_p & 0 & -n\frac{k_p}{k_d} \end{bmatrix} \qquad B_{cs} = \begin{bmatrix} B_p & B_p \\ 0 & 0 \end{bmatrix} \\ C_{cs} &= \begin{bmatrix} C_p & 0 & 0 \\ -(1+n)k_pC_p & k_i & k_d \end{bmatrix}. \end{aligned} \tag{2.20}$$

where $A_{cs} \in \mathbb{R}^{m\times m}$, $B_{cs} \in \mathbb{R}^{m\times 2}$, and $C_{cs} \in \mathbb{R}^{2\times m}$.

2.3 Defining an event-based system as a PLS

The results obtained in this section do not depend on where the sampler is placed. Therefore, the subindexes representing the position of the sampler, ps and cs, have been dropped to simplify the notation. For example, A is equivalent to A_{ps}, if the process variable is being sample, or to A_{cs} in the other case.

Let us consider that the input to $G(s)$ is delayed in time by τ and the loop is closed by adding the non-linearity represented by the SOD sampler. Then,

$$G(s) \sim \begin{cases} \dot{X}(t) = AX(t) + BU(t-\tau) \\ Y(t) = CX(t) + DU(t-\tau). \end{cases} \tag{2.21}$$

Now, the resulting system is an infinite dimensional system because of the time delay. However, it can be simplified because closing the loop with the non-linearity makes the input signal piecewise constant. So, in the matrix U the input $u_{nl}(t)$ is redefined as

$$u_{nl_k} = (j_k + \alpha)\delta,$$

where $j_k \in \mathbb{Z}$ is the crossed level by the input signal $u_{nl}(t)$, and $\alpha \in [0,1) \in \mathbb{R}$ is the sampling offset which depends on the initial sample, $u_{nl}(t_0)$. Let us define the switching times,

Definition 3. *Consider a limit cycle composed of n switchings, and assume $T = \{t_0, ..., t_n\}$ are the sampling times, as in Definition 1. Then the switching times are defined as $t_i^* = t_i - t_{i-i}$ (see Figure 2.5).*

And the order of the limit cycle,

Definition 4. *Consider a limit cycle in a symmetric non-linearity of order n ($n+1$ crossing levels, as in Definition 2). Then, the number of switchings of the limit cycle is $2n$.*

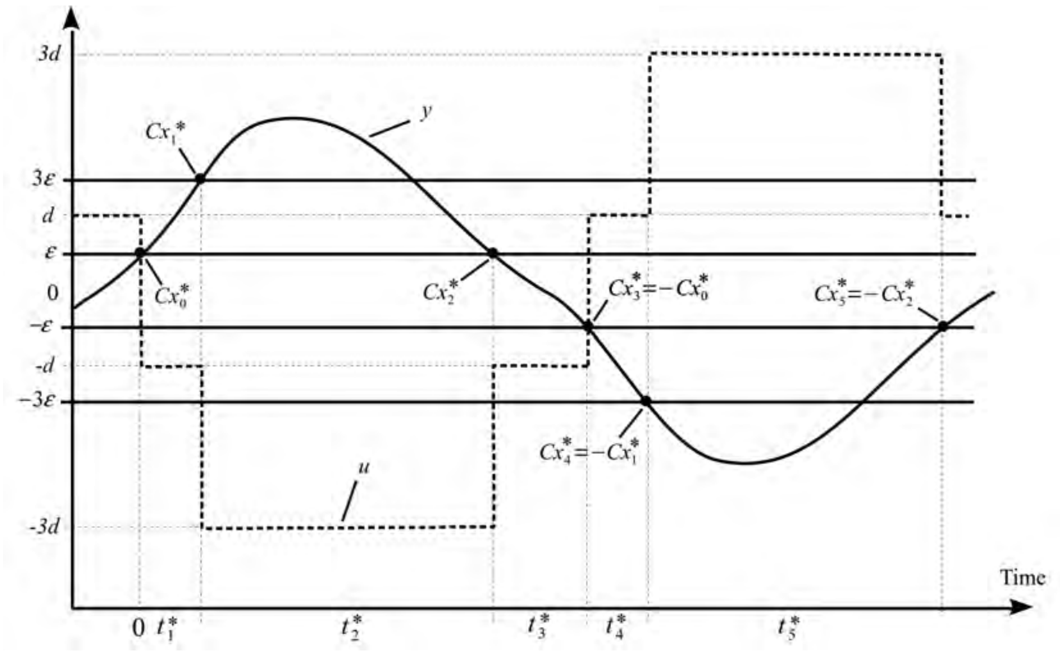


Figure 2.5: Trajectory of a solution of a PLS system corresponding to a limit cycle. The solid line represent the process output, and the dashed line the control input.

Then, if limit cycles of period T with switching times, $t_i^* > \tau$ where $T = t_1^* + t_2^* + \ldots + t_{2n}^*$ are only considered, the input $\{u_{nl}(t), t_i^* - \tau < t \leq t_i^*\}$ is constant and its value is given by the feedback. In this case, the state of the system at the sampling times $t_1, t_2..., t_{2n}$ is uniquely given by $X(t_i)$. Taking into account this consideration, the system can be considered as a PLS defined by

$$\begin{aligned} \dot{X}(t) &= AX(t) + B_i \\ Y(t) &= CX(t) + D_i, \end{aligned} \tag{2.22}$$

where $B_i = B\begin{bmatrix} u_{nl_i} & d \end{bmatrix}^T$, $D_i = D\begin{bmatrix} u_{nl_i} & d \end{bmatrix}^T$, and $u_{nl_i} = (j_i + \alpha)\delta$, for $i = \{0, \pm 1, \ldots, \pm n\}$.

In this PLS the rule to switch between the linear systems is included in the definitions of the non-linearities. It must be noted that the switching rules have

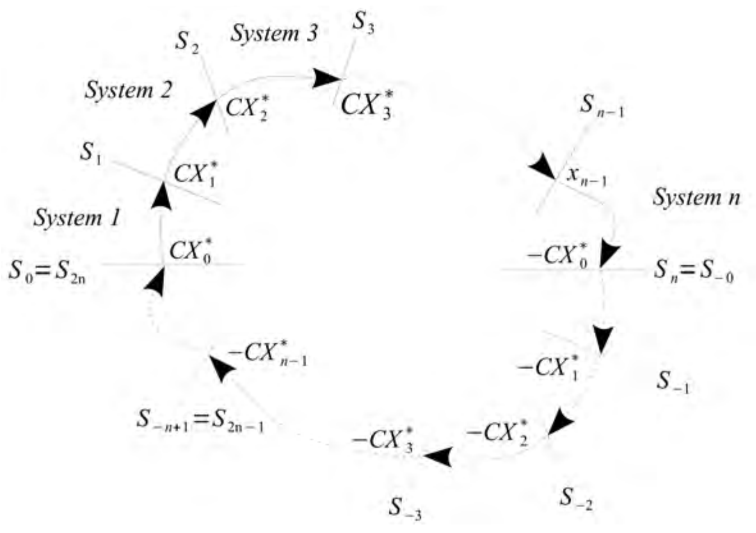


Figure 2.6: Trajectory of a solution of a PLS system. In each region, the dynamics of the system is determined by a different LTI system defined by (2.22). The lines represents the switching surfaces.

memory and the decision of which LTI to use may not depend only on the actual values of the state, but also on their past values. In the state-space, the points in which a rule provokes the switching from the system i to the system j define a surface which is known as *switching surface* (see Figure 2.6). This surfaces consist of hyperplanes of dimension $m-1$, being m the order of the system, i.e. $X \in \mathbb{R}^m$,

$$S_i = \{X \,|CX - u_{nl_i} = 0\} \quad \text{for } i = \{0, \pm 1, \dots, \pm n\}.$$

It is interesting to characterize the limit cycles that can appear in the system due to the effect of the non-linearities introduced with the event-based sampling scheme. The study of these limit cycles can be interesting for several reasons. There is a wide range of processes that will almost surely present limit cycles with the studied control schemes, while for other processes they can be prevented by carefully choosing the controller parameters. In any case, limit cycles mean oscillations, which, depending on the process, may be more or less problematic. For example, a high frequency of oscillation may wear out the actuators. Also, the study of limit cycles can been used for identification purposes. For example, the relay auto tuning method [AH84] is based on the properties of the limit cycle that appears in a process subject to relay feedback (which can be considered as a particular case of the studied event-based scheme). In addition, for the cases when the limit cycles cannot be prevented, it may be important to know about the stability of these limit cycles.

In order to calculate the period and amplitude, first it is assumed that the limit cycle contains $n+1$ levels, and thus it is composed of $2n$ switchings, where the first n correspond to the positive level crossings, and the rest n are the negative ones. Next, it is presented the set of equations that allows us to find the switching times and the values of the states at these switching times. The result stated in the following proposition is a generalization to n levels of the approaches in [CA07], [Å95], and [Gon00].

Proposition 2. *Consider the PLS in (2.22), with a non-linearity defined by the switching surfaces $S_i = \{X \mid CX - (j_i + \alpha)\delta = 0\}$, where $\alpha \in [0,1)$, $j_i \in \mathbb{Z}$, $i \in \{0, \pm 1, \pm n\}$, and $0 < \delta \in \mathbb{R}$. Assume that there exists a symmetric periodic solution γ with $2n$ switching surfaces per period $T = t_1^* + t_2^* + \ldots + t_{2n}^*$, where t_1^* , t_2^* , ... t_{2n}^* are the switching times when the switching surfaces S_1 ,..., S_n ,S_{-1}.. S_{-n} are crossed, respectively (Figure 2.6). Define*

$$f_k(t_1^*, \ldots, t_{2n}^*) = C(I + e^{AT})^{-1} \left[\sum_{i=1}^{2n-1} \Phi_{2n-1} \cdots \Phi_{i+1} (\Phi_i - I) \Lambda_i \right] - E_k, \qquad (2.23)$$

where $\Phi_i = \Phi(t_i) = \mathrm{e}^{At_i}$, and $\Lambda_i = A^{-1}B_i$. Then the following conditions hold

$$\begin{cases} f_1(t_1^*, t_2^*, \ldots, t_{2n}^*) = 0 \\ f_2(t_1^*, t_2^*, \ldots, t_{2n}^*) = 0 \\ \quad \vdots \\ f_{2n}(t_1^*, t_2^*, \ldots, t_{2n}^*) = 0, \end{cases} \qquad (2.24)$$

and

$$\begin{cases} E_i \leq y(t) = CX_i(t) < E_{i+1} & \text{for } 0 \leq t < t_i^* \quad i = 1 \dots n-1 \\ y(t) = CX_n(t) \geq E_{n+1} & \text{for } 0 \leq t < t_n^* \\ E_i \geq y(t) = CX_i(t) > E_{i+1} & \text{for } 0 \leq t < t_i^* \quad i = n+1 \dots 2n, \end{cases} \tag{2.25}$$

where

$$E_i = (j_i + \alpha)\delta, \ \text{ and } X_i(t) = e^{At}X_{i-1}^* - A^{-1}(e^{At} - I)B_i.$$

Furthermore, the limit cycle can be obtained with the initial condition

$$X_0^* = (I + e^{AT})^{-1} \left[\sum_{i=1}^{2n-1} \Phi_{2n-1} \cdots \Phi_{i+1} (\Phi_i - I) \Lambda_i \right]. \tag{2.26}$$

Proof. Assuming that $t_i > \tau$, where t_i is the time elapsed between the crossing of two consecutive switching surfaces, for example i and $i+1$, then the state is obtained by integrating (2.22) from $t = 0$ to $t = t_i$. It gives

$$X_{i+1} = \Phi(t_i)X_i + \Gamma_1(t_i)U_{i-1} + \Gamma_0(t_i)U_i, \tag{2.27}$$

where $\Phi(t) = \mathrm{e}^{At}$ is the state transition matrix, $\Gamma_0 = \int_0^{t-\tau} \Phi(s)ds$ accounts for the effect of the input, and $\Gamma_1 = \int_{t-\tau}^{t} \Phi(s)ds$ is a term which represents the effect of the input because of the time delay of the system.

Then, for a limit cycle involving n switching times, we have a system of equations described by

$$\begin{aligned} \Phi(t_1)X_1 + \Gamma_1(t_1)U_n + \Gamma_0(t_1)U_1 - X_2 &= 0 \\ \cdots& \\ \Phi(t_{n-1})X_{n-1} + \Gamma_1(t_{n-1})U_{n-2} + \Gamma_0(t_{n-1})U_{n-1} - X_n &= 0 \\ \Phi(t_n)X_n + \Gamma_1(t_n)U_{n-1} + \Gamma_0(t_n)U_n - X_1 &= 0. \end{aligned} \tag{2.28}$$

Substituting recursively, we get the following expression,

$$\begin{aligned}
[I - \Phi(t_n)...\Phi(t_1)]X_n &= \Phi(t_n)...\Phi(t_2)(\Gamma_1(t_1)U_n + \Gamma_0(t_1)U_1) \\
&+ \Phi(t_n)...\Phi(t_3)(\Gamma_1(t_2)U_1 + \Gamma_0(t_2)U_2) + ... \\
&+ \Phi(t_n)(\Gamma_1(t_{n-1})U_{n-2} + \Gamma_0(t_{n-1})U_{n-1}) \\
&+ \Gamma_1(t_n)U_{n-1} + \Gamma_0(t_n)U_n.
\end{aligned} \tag{2.29}$$

The previous expression can be written in compact form as,

$$[I - \Phi_n...\Phi_1]X_n = \sum_{i=1}^{2n-1} \Phi_{2n-1}...\Phi_{i+1}[\Gamma_1(t_i)U_{i-1} + \Gamma_0(t_i)U_i], \tag{2.30}$$

and,

$$X_n = [I + \Phi_n...\Phi_1]^{-1} \sum_{i=1}^{2n-1} \left(\prod_{j=1}^{2n-1-j} \Phi_{2n-j} \right) [\Gamma_1(t_i)U_{i-1} + \Gamma_0(t_i)U_i]. \tag{2.31}$$

If we assume that the system matrix A is nonsingular, then the integrals Γ_1 and Γ_0 can be explicitly computed and the state X_n can be solved, yielding the expression of the initial state (2.26). Since we know that at the switching times the state must be at the switching surface, we can combine the previous expression with the switching conditions to get the set of equations given by (2.24). Finally, the conditions (2.25) hold because the state does not cross the switching surface in the interval between two switchings. $\square$

However, if the system matrix is assumed to be singular, neither the state nor the integrals Γ_0 and Γ_1 can be computed explicitly. In this case the system of equations is obtained in the same way, but the computations are more involved. First the functions in (2.24) are redefined as,

$$f_i(X_i^*, t_1^*, ..., t_{2n}^*) = [I - \Phi_{2n}...\Phi_1]X_i^* - \sum_{i=1}^{2n-1} \Phi_{2n-1}...\Phi_{i+1}[\Gamma_1(t_i^*)U_{i-1} + \Gamma_0(t_i^*)U_i]. \tag{2.32}$$

where, in contrast to the previous case, X_i appear as unknowns.

Note that the system of equations given by the functions f_i is composed of n^2 scalar equations and $n^2 + n$ unknowns. Thus, in order to solve it, the system must be completed with n additional equations. These equations are obtained from the switching conditions in the sampler, which fix the values of either the process output or the control input at the event times, depending on where the sampling is placed. Then, the complete set of equations that must be solved to obtain the features of the limit cycle is,

$$\left\{ \begin{aligned} f_1(X_1^*, t_1^*, ..., t_{2n}^*) &= 0 \\ &\vdots \\ f_{2n}(X_{2n}^*, t_1^*, ..., t_{2n}^*) &= 0 \\ CX_1^* - d_1 &= 0 \\ &\vdots \\ CX_{2n}^* - d_{2n} &= 0. \end{aligned} \right. \tag{2.33}$$

How to use this result to analyze the limit cycles is demonstrated with examples in Sections 2.4 and 2.5.

2.3.1 Local stability

The local stability of the limit cycles described in the previous paragraphs can be analized by observing the system at the switching times. The following result, which is a generalization of the approaches in [CA07], [Å95], and [Gon00], can be applied to study the behaviour of the trajectories of the system in the proximities of the limit cycles.

Proposition 3. *Assume that there exists a limit cycle γ with k states in the system (2.22). The limit cycle is locally stable if and only if $W = W_k W_{k-1} ... W_1$ has all its* *eigenvalues inside the unit disk, where* $W_i = (I - \frac{V_i C_i}{C_i V_i})\mathrm{e}^{At_i^*}$, $V_i = AX_i^* + B_i$, t_i^* *are the switching times, and* X_i^* *the state at the switching times.*

Proof. Consider a trajectory with initial condition $x(0) = x_0^*$. In the time interval before the first switching, this trajectory is defined as $x(t) = \Phi(t)x_0^* + \Gamma_1(t)u_{nl_{n-1}} + \Gamma_0(t)u_{nl_n}$. When x reaches the switching surface at time $t_1^* + \delta_1 t_1^*$, we have,

$$x(t_1^* + \delta_1 t_1^*) = \Phi(t_1^* + \delta_1 t_1^*)(x_0^* + \delta_1 x_0^*) + \Gamma_1(t_1^* + \delta_1 t_1^*)u_{nl_{n-1}} + \Gamma_0(t_1^* + \delta_1 t_1^*)u_{nl_n}.$$

The series expansion in $\delta_1 t_1^*$ and $\delta_1 x_0^*$ is,

$$x(t_1^* + \delta_1 t_1^*) = x_1^* + v_1 \delta_1 t_1^* + \Phi(t_1^*)\delta_1 x_0^* + O(\delta_1^2),$$

where $v_1 = Ax_1^* + B_1 = A[\Phi(t_1^*)x_0^* + \Gamma_1(t_1^*)u_{nl_{n-1}} + \Gamma_0(t_1^*)u_{nl_n}] + Bu_{nl_n}$.

Since the solution is on the switching surface at $t_1^* + \delta_1 t_1^*$, we have

$$C_1 x(t_1^* + \delta_1 t_1^*) + d_1 = C_1 x_1^* + C_1 v_1 \delta_1 t_1^* + C_1 \Phi(t_1^*)\delta_1 x_0^* + d_1 = 0,$$

and, therefore, the following equality holds,

$$\delta_1 t_1^* = -\frac{C_1 \Phi(t_1^*)}{C_1 v_1}\delta_1 x_0^*.$$

The rest of the proof follows as in [Gon00]. The state after the first switch is

$$x(t_1^* + \delta_1 t_1^*) = x_1^* + (I - \frac{v_1 C_1}{C_1 v_1})\Phi(t_1^*)\delta_1 X_0 + O(\delta_1^2) = x_1^* + W_1 \delta_1 x_0^* + O(\delta_1^2). \quad (2.34)$$

Taking as initial condition $x_1^* + \delta_2 x_1^*$ and neglecting the $O(\delta^2)$ term we have $x(t_2^* + \delta_2 t_2^*) = x_2^* + W_2 \delta_2 x_1^*$. Combining with (2.34) yields $\delta_2 x_1^* = W_1 \delta_1 x_0^*$. Replacing in the previous expression and applying succesively for k eventually leads to

$$x(t_k^* + \delta_k t_k^*) = x_0^* + W_k W_{k-1} ... W_1 \delta_1 x_0^* + O(\delta_1^2). \quad (2.35)$$

Neglecting the $O(\delta_1^2)$ term, the dynamics of equation 2.35 is stable if and only if the eigenvalues of $W = W_k W_{k-1} ... W_1$ are inside the unit disk. This proves the proposition.

□

2.4 Analysis of the limit cycles

In the following sections the expressions which take into account the effects of the event-based sampling at the output of the process and controller are presented. The method used is based on the grouping of all the continuous dynamics into one block to obtain the state-space matrix (as shown in Figure 2.3), and then to study the effect of the non-linear feedback introduced by the sampling block.

To easily distinguish between the different types of controller and sampling, the following naming convention is used: *controller*-SOD_n-*process*, when the sampler is after the controller output, and *controller*-*process*-SOD_n if the sampler is placed after the process output, where *process* corresponds to the type of process considered (IPTD, SOPDT,...) and *controller* refers to the type of controller (PI, PD, PID,...) The index n refers to the number of hysteresis bands presented in the sampler. For example, PI-SOD_1-IPTD denotes the system composed by an IPTD process controlled by a PI controller with the sampler placed after the controller output, and a limit cycle with one hysteris band (i.e. a system with relay feedback).

There are two directions in which the complexity of the analysis can be increased. The first one is to consider that the order of the process is increasing, i.e. a simple integrator, a double integrator, etc., and the second one is to consider an increase in the number of hysteresis bands of the sampler.

To simplify the analysis, it is worth noting that the solutions of (2.33) are linear on δ, thus the state and control signal can be normalized dividing by δ (note that $\delta > 0$).

2.4.1 Equilibrium points

Consider the system (2.22). The set of equilibrium points if defined as $\mathbb{X} = \{x^* | \dot{x} = 0\}$. An equilibrium point is one in which the derivatives of the states are null, i.e. $x^* | \dot{x}^* = 0$. Since the derivatives are null, all the trajectories which enter into it at t_0 will stay for $t > t_0$. An inmediate necessary condition to have an equilibrium point is that the linear system $Ax + B_i = 0$ has at least one solution. Note that, except for the P and PD controller, it is easy to see that $det(A) = 0$, due to the integrator added by the controller, thus being possible to have a system of equations which is either indetermined or incompatible. The system not have any solution if $rg(A) < rg(A|B_i)$, so an equilibrium point can exist only if $\exists i \in \mathbb{R} \mid rg(A) = rg(A|B_i)$.

Now assume that the output is within the k band of hysteresis, i.e. $x \in \Omega_k = \{x | \delta_k \leq y(t) = Cx(t) < \delta_{k+1}\}$ for some time interval $t_k \leq t < t_{k+1}$.

Proposition 4. *A necessary condition for the system to be ultimately bounded to* Ω_k *is that either* $C[Ax + B_k] = 0$ *or* $C[Ax + B_{k+1}] = 0$.

Proof. Assume that $Cx(t)$ enters into Ω_k at t_0. After a time $t > \tau$, the derivative of the system is $C\dot{x}(t) = C[Ax(t) + B_i]$, for $i \in \{k, k+1\}$. Thus, if $C[Ax(t) + B_i] \neq 0$ for both $i = k$ and $i = k+1$, the process output will eventually cross the boundaries of Ω_k for some t. □

Computing the equilibrium points of the PI control with SOD sampling at the process output (the setpoint is assumed to be null) yields:

$$\begin{bmatrix} \dot{x_p} \\ \dot{x_c} \end{bmatrix} = \begin{bmatrix} A_p & k_i B_p \\ 0 & 0 \end{bmatrix} \begin{bmatrix} x_p \\ x_c \end{bmatrix} + \begin{bmatrix} -k_p B_p & B_p \\ -1 & 0 \end{bmatrix} \begin{bmatrix} \delta_i \\ d \end{bmatrix} = 0. \qquad (2.36)$$

Thus, a necessary condition for the existence of an equilibrium point, from (2.37), is: $\exists i \in \mathbb{Z} | \delta_i = 0$, because otherwise the integrator derivative is a non-null constant. Note that, if the setpoint is not assumed to be null, the condition still holds with a slight modification: $\exists i \in \mathbb{Z} | \delta_i = y_{sp}$, i.e. the setpoint must be an exact multiple of δ. Furthermore, since A_p is a non-singular matrix, the computation of the equilibrium point is straightforward: $x_c = \frac{-d}{k_i}$, $x_p = 0 \in \mathbb{R}^n$.

If the sampler affects to the controller output, then the equilibrium equation is:

$$\begin{bmatrix} \dot{x_p} \\ \dot{x_c} \end{bmatrix} = \begin{bmatrix} A_p & 0 \\ -C_p & 0 \end{bmatrix} \begin{bmatrix} x_p \\ x_c \end{bmatrix} + \begin{bmatrix} B_p & B_p \\ -1 & 0 \end{bmatrix} \begin{bmatrix} \delta_i \\ d \end{bmatrix} = 0, \tag{2.37}$$

and the necessary condition to have an equilibrium point is, $\exists i \in \mathbb{Z} | \delta_i = (1 - C_p A_p^{-1} B_p)^{-1} d$. If this expression holds, then the equilibrium point can be computed as $x_p = \frac{C_p^T}{\|C_p\|}\delta_i$, $x_c = (1 + k_p)\delta_i$.

The rest of this section presents an algorithm to compute the period and amplitude of a limit cycle in a generic process, and then shows examples of application to several common processes.

2.4.2 Algorithm

In this section an algorithm to obtain computationally the period of a limit cycle and the intermediate switching times is outlined. Assume that $\alpha = 0.5$ and that the system presents a limit cycle in which the condition $y(t) = Cx(t) \in ((\alpha - n)\delta, (\alpha + n - 1)\delta)$ holds. The limit cycle is assumed to have only two changes in the sign of the derivative one at the begining of the first semiperiod and the other at the begining of the second semiperiod. Thus, the limit cycle must have $4n - 2$ switchings. Because of the symmetry, the behaviour of the limit cycle can be inferred by studying only the first semiperiod, thus reducing the complexity to $2n - 1$ levels. The algorithm, that can be implemented either in a symbolic or in a numerical computation tool, is

1. Set n as the number of levels crossed within the limit cycle.
2. Fix the values of k_p, k_i, α, δ, d, τ, and the matrices A and B.

3. Calculate $\Phi(t) = e^{At}$, $\Gamma_0(t) = \int_0^{t-\tau} e^{As} ds$ and $\Gamma_1(t) = \int_{t-\tau}^{t} e^{As} ds$.

- To calculate the period:

  1. For i from 1 to $2n$ repeat steps 2-3.
  2. If $i \in (1, n)$, set $j_i = i - \lfloor \frac{n}{2} \rfloor$, else $j_i = \lfloor \frac{n}{2} \rfloor + n - i$.
  3. Set $u_{nl_i} := (j_i + \alpha)\delta$, and,
     - $x_{c_i} = -\frac{u_{nl_i} + k_p x_{p_i}}{k_i}$, if sampling the controller output, or,
     - $x_{p_i} = u_{nl_i}$, if sampling the process output.
  4. Set $eq_i := -X_{i+1} + \Phi(t_i)X_i + \Gamma_1(t_i)U_j + \Gamma_0(t_i)U_i = 0$.
  5. Solve the system of equations given by eq_i, with the unknowns t_i, and x_{p_i} or x_{c_i}.
  6. $T = \sum_{i=1}^{2n} t_i$.

- To calculate the amplitude:

  1. Set $j_{max} = j|(X_j > X_i, \forall i \neq j)$ and $j_{min} = j|(X_j < X_i, \forall i \neq j)$.
  2. Find $t_{min} = \min(\tau, t|C\dot{x}_{j_{min}}(t) = 0)$, and $t_{max} = \min(\tau, t|C\dot{x}_{j_{max}}(t) = 0)$, corresponding to the minimum and maximum values of the output.
  3. Compute the amplitude of the process output, $\Delta_p = C_p[X(t_{max}) - X(t_{min})]$, and the control input, $\Delta_c = C_c[X(t_{max}) - X(t_{min})]$.

2.4.3 Examples of analysis

To illustrate the application of Proposition 2 and the use of the algorithm of Section 2.4.2, the analysis of several systems (IPTD, SOPTD, FOPTD and SOPTD) is detailed in the following lines. The results are summarized in Table 2.1, at the end of the section.

2.4.3.1 IPTD process

Let us consider an IPTD process $P(s) = \frac{k}{s}\mathrm{e}^{-\tau s}$, which can be described by its state-space model in the canonical observable form as,

$$\begin{aligned} \dot{x}(t) &= -ku(t-\tau) + kd(t) \\ y(t) &= x(t) = x_p(t), \end{aligned} \tag{2.38}$$

where k is the process gain, x is the state, u the process input, and y is the process output. First it is presented the approach for the PI-IPTD-SOD$_\mathrm{n}$ structure, and then for the PI-SOD$_\mathrm{n}$-IPTD.

Process variable sampling (PI-IPTD-SOD$_1$) According to (2.6), the state feedback matrix corresponding to the system controlled by a PI with SOD sampling at the process output is,

$$\begin{aligned} \begin{bmatrix} \dot{x}_p(t) \\ \dot{x}_c(t) \end{bmatrix} &= \begin{bmatrix} 0 & kk_i \\ 0 & 0 \end{bmatrix} \begin{bmatrix} x_p(t) \\ x_c(t) \end{bmatrix} + \begin{bmatrix} kk_p & k \\ 1 & 0 \end{bmatrix} \begin{bmatrix} u_{nl_k} \\ d \end{bmatrix} \\ y(t) &= \begin{bmatrix} 1 & 0 \end{bmatrix} \begin{bmatrix} x_p(t) \\ x_c(t) \end{bmatrix}. \end{aligned} \tag{2.39}$$

Assuming there exists a stable limit cycle with two states, then the equations that allow us to obtain the amplitude and period of the oscillations are,

$$\begin{aligned} \Phi(t_1)X_1 + \Gamma_1(t_1)U_2 + \Gamma_0(t_1)U_1 - X_2 = 0 \\ \Phi(t_2)X_2 + \Gamma_1(t_2)U_1 + \Gamma_0(t_2)U_2 - X_1 = 0 \\ x_{p_1} = -u_{nl_1} = \alpha\delta \\ x_{p_2} = -u_{nl_2} = (\alpha-1)\delta, \end{aligned} \tag{2.40}$$

where $X_i = [x_{p_i} \quad x_{c_i}]^T$, and $U_i = [u_{nl_i} \quad d]^T$, are the state and input, respectively.

The matrices Φ, Γ_0, and Γ_1 can be calculated as,

$$\begin{aligned}
\Phi(t) &= \begin{bmatrix} 1 & kk_i t \\ 0 & 1 \end{bmatrix} \\
\Gamma_0(t) &= \begin{bmatrix} kk_p t - kk_p \tau + \frac{1}{2} kk_i t^2 - kk_i t \tau + \frac{1}{2} kk_i \tau^2 & k(t-\tau) \\ t-\tau & 0 \end{bmatrix} \\
\Gamma_1(t) &= \begin{bmatrix} kk_p \tau + kk_i t \tau - \frac{1}{2} kk_i \tau^2 & k\tau \\ \tau & 0 \end{bmatrix}.
\end{aligned} \tag{2.41}$$

Introducing (2.41) in (2.40), and solving the resulting equations, the period of the limit cycle can be obtained (see Table 2.1). Looking at the expression of the period, it can be seen that the symmetry of the limit cycle, i.e. the difference between the two semiperiods t_1 and t_2 depends on the offset of the sampler α. In particular, for $\alpha = 0.5$ the two semiperiods have the same value. When $\alpha = 0$, t_2 vanishes, which can be interpreted as this limit cycle cannot exist. In this case, either the system will reach a steady-state or enter into a limit cycle with higher number of levels. With respect to the disturbance rejection, it can be seen that d does not affect the period. This is because it is rejected by the integrator, which changes its mean value to absorb the disturbance.

With the switching times t_1 and t_2, the amplitudes of the oscillations can be computed. It is necessary to find the maximum and minimum of the output, which correspond to the times when its first derivative is null, i.e. $\dot{x}_p(t) = kk_i x_c(t) + kk_p u_{nl_k} + kd = 0$. By solving the previous expression, the value of x_c and the times when the peaks are reached can be obtained, and so for this case it can be found an analytic expression for the amplitude of the process output, A_{po}, and of the control input, A_{ci} (See Table 2.1).

Controller variable sampling (PI-SOD$_1$-IPTD) When the sampler is placed at the controller output, the description of the system in the state-space is given by expressions (2.7). Thus, for this process, it is obtained

$$\begin{aligned}\begin{bmatrix}\dot{x}_p(t)\\ \dot{x}_c(t)\end{bmatrix} &= \begin{bmatrix}0 & 1\\ 0 & 0\end{bmatrix}\begin{bmatrix}x_p(t)\\ x_c(t)\end{bmatrix} + \begin{bmatrix}0 & 0\\ k & k\end{bmatrix}\begin{bmatrix}u_{nl_k}\\ d\end{bmatrix}\\ y(t) &= \begin{bmatrix}k_p & k_i\end{bmatrix}\begin{bmatrix}x_p(t)\\ x_c(t)\end{bmatrix}.\end{aligned} \tag{2.42}$$

Assuming there exists a stable limit cycle with two states, then the equations that allow to obtain the amplitude and period of the oscillations are

$$\begin{aligned}\Phi(t_1)X_1 + \Gamma_1(t_1)U_2 + \Gamma_0(t_1)U_1 - X_2 = 0\\ \Phi(t_2)X_2 + \Gamma_1(t_2)U_1 + \Gamma_0(t_2)U_2 - X_1 = 0\\ k_p x_{p_1} + k_i x_{c_1} = u_{nl_1} = \alpha\delta\\ k_p x_{p_2} + k_i x_{c_2} = u_{nl_2} = (\alpha - 1)\delta.\end{aligned} \tag{2.43}$$

Here, the amplitude of the limit cycle can be computed directly, since the control input is piecewise constant and the switching times and values are known. The period of the limit cycle can be obtained by solving the system of equations (2.43) (see Table 2.1).

As opposed to the process sampling, here the disturbance appears in the expression of the semiperiods, thus affecting to the simmetry of the limit cycle. It is possible to have oscillations where the process is changing slowly nearly all the time and then to have an abrupt change. Since in one semiperiod the control action is more agressive, this decreases the margin of delay that can be added to the system without reaching the next sampling level.

To obtain the expression corresponding to the amplitude, and since the maximum and minimum values of the process output are reached at times $t_1 + \tau$ and $t_2 + \tau$, integrating (2.42) yields the desired result (see Table 2.1).

2.4.3.2 DIPTD process

It is well known in classic control theory that the double integrator process cannot be stabilized with a continuous PI controller, due to the 90º phase lag of each integrator. It becomes then necessary to introduce the derivative action (PD controller) to compensate this lag. In the same way, either with the PI-SOD$_\mathrm{n}$-DIPTD and PI-SOD$_\mathrm{n}$-DIPTD the system will oscillate with unbounded growing amplitudes. Though it is not in the scope of this work, it should be possible to stabilize this kind of process by a SOD-PD controller. In practice, the implementation of the derivative action must be carefully studied, because it can be problematic specially when the sampler is placed after the process output, since the estimation of the derivative may be poor.

2.4.3.3 FOPTD process

The process considered in this section is a FOPTD process $P(s) = \frac{k}{Ts+1}\mathrm{e}^{-\tau s}$, which is described in the state-space by the following expressions,

$$\begin{aligned} \dot{x}(t) &= -\frac{1}{T}x(t) + \frac{k}{T}u(t-\tau) + d(t) \\ y(t) &= x(t) = x_p(t), \end{aligned} \tag{2.44}$$

where k is the process gain, x is the state, u the process input, y is the process output, and d is an external disturbance.

Process variable sampling (PI-FOPTD-SOD$_1$) The expressions corresponding to the PI-FOPTD-SOD$_1$ are as follows,

$$\begin{bmatrix} \dot{x}_p(t) \\ \dot{x}_c(t) \end{bmatrix} = \begin{bmatrix} \frac{1}{T} & \frac{kk_i}{T} \\ 0 & 0 \end{bmatrix} \begin{bmatrix} x_p(t) \\ x_c(t) \end{bmatrix} + \begin{bmatrix} \frac{kk_p}{T} & \frac{k}{T} \\ 1 & 0 \end{bmatrix} \begin{bmatrix} u_{nl_k} \\ d \end{bmatrix}$$

$$y(t) = \begin{bmatrix} k_p & k_i \end{bmatrix} \begin{bmatrix} x_p(t) \\ x_c(t) \end{bmatrix}. \tag{2.45}$$

From (2.45) the matrices Φ, Γ_0, and Γ_1 can be obtained as,

$$\begin{aligned}
\Phi(t) &= \begin{bmatrix} \mathrm{e}^{\frac{t}{T}} & kk_i(\mathrm{e}^{\frac{t}{T}} - 1) \\ 0 & 1 \end{bmatrix} \\
\Gamma_0(t) &= \begin{bmatrix} -k(k_p + k_iT)(\mathrm{e}^{\frac{t-\tau}{T}} - 1) + kk_i(t-\tau) & T(\mathrm{e}^{\frac{t-\tau}{T}} - 1) \\ t-\tau & 0 \end{bmatrix} \\
\Gamma_1(t) &= \begin{bmatrix} k(k_p + k_iT)(\mathrm{e}^{\frac{t-\tau}{T}} - \mathrm{e}^{\frac{t}{T}}) - kk_i\tau & -T(\mathrm{e}^{\frac{t-\tau}{T}} - \mathrm{e}^{\frac{t}{T}}) \\ \tau & 0 \end{bmatrix}.
\end{aligned} \tag{2.46}$$

As it can be seen in the previous expressions, since the system of equations obtained for the FOPTD contains terms involving exponentials it is not possible to find analytical solutions, as opposed to the case of IPTD processes. Instead of that, the solutions have to be found numerically. However, the algorithm proposed in Section 2.4.2 can be applied.

Controller variable sampling (PI-SOD$_1$-FOPTD) The expressions corresponding to the PI-SOD$_1$-FOPTD are the following ones,

$$\begin{aligned}
\begin{bmatrix} \dot{x}_p(t) \\ \dot{x}_c(t) \end{bmatrix} &= \begin{bmatrix} -\frac{1}{T} & 0 \\ 1 & 0 \end{bmatrix} \begin{bmatrix} x_p(t) \\ x_c(t) \end{bmatrix} + \begin{bmatrix} \frac{k}{T} & \frac{k}{T} \\ 1 & 0 \end{bmatrix} \begin{bmatrix} u_{nl_k} \\ d \end{bmatrix} \\
y(t) &= \begin{bmatrix} k_p & k_i \end{bmatrix} \begin{bmatrix} x_p(t) \\ x_c(t) \end{bmatrix} \\
y_p(t) &= \begin{bmatrix} 1 & 0 \end{bmatrix} \begin{bmatrix} x_p(t) \\ x_c(t) \end{bmatrix}.
\end{aligned} \tag{2.47}$$

As in the previous case, the solutions of the equations must be found by means of numerical tools.

2.4.3.4 SOPTD process

The process considered in this section is a SOPTD $P(s) = \frac{k}{(\tau_1 s+1)(\tau_2 s+1)}\mathrm{e}^{-\tau s}$, which is described in the state-space by the following expressions,

$$\begin{aligned} \ddot{x}(t) &= -\frac{1}{\tau_1\tau_2}x - \frac{\tau_1+\tau_2}{\tau_1\tau_2}\dot{x} + \frac{k}{\tau_1\tau_2}u(t-\tau) + d(t) \\ y(t) &= x(t) = x_p(t), \end{aligned} \tag{2.48}$$

where k is the process gain, τ_1 and τ_2 the time constants, x is the state, u the process input, y is the process output, and d is an external disturbance.

Process variable sampling (PI-SOPTD-SOD$_\mathrm{n}$) The expressions corresponding to the PI-SOPTD-SOD$_\mathrm{n}$ are the following ones,

$$\begin{aligned} \begin{bmatrix} \dot{x}_p(t) \\ \ddot{x}_p(t) \\ \dot{x}_c(t) \end{bmatrix} &= \begin{bmatrix} 0 & 1 & 0 \\ -\frac{1}{\tau_1\tau_2} & -\frac{\tau_1+\tau_2}{\tau_1\tau_2} & \frac{kk_i}{\tau_1\tau_2} \\ 0 & 0 & 0 \end{bmatrix} \begin{bmatrix} x_p(t) \\ \dot{x}_p(t) \\ x_c(t) \end{bmatrix} + \begin{bmatrix} 0 & 0 \\ \frac{kk_p}{\tau_1\tau_2} & \frac{k}{\tau_1\tau_2} \\ 1 & 0 \end{bmatrix} \begin{bmatrix} u_{nl_k} \\ d \end{bmatrix} \\ y(t) &= \begin{bmatrix} 1 & 0 & 0 \\ k_p & 0 & k_i \end{bmatrix} \begin{bmatrix} x_p(t) \\ \dot{x}_p(t) \\ x_c(t) \end{bmatrix}. \end{aligned} \tag{2.49}$$

As in the FOPTD case, for this system it is not possible to find analytical solutions due to the exponentials that appear in the equations. Therefore, numerical methods must be used to find the solutions.

Controller variable sampling (PI-SOD$_\mathrm{n}$-SOPTD) The expressions corresponding to the PI-SOD$_\mathrm{n}$-SOPTD are the following ones,

$$\begin{bmatrix} \dot{x}_p(t) \\ \ddot{x}_p(t) \\ \dot{x}_c(t) \end{bmatrix} = \begin{bmatrix} 0 & 1 & 0 \\ -\frac{1}{\tau_1\tau_2} & -\frac{\tau_1+\tau_2}{\tau_1\tau_2} & 0 \\ 1 & 0 & 0 \end{bmatrix} \begin{bmatrix} x_p(t) \\ \dot{x}_p(t) \\ x_c(t) \end{bmatrix} + \begin{bmatrix} 0 & 0 \\ \frac{k}{\tau_1\tau_2} & \frac{k}{\tau_1\tau_2} \\ 1 & 0 \end{bmatrix} \begin{bmatrix} u_{nl_k} \\ d \end{bmatrix}$$

$P(s)$	PI-process-SOD$_1$	PI-SOD$_1$-process
$\frac{k}{s}e^{-\tau s}$	$T = \frac{1}{2}\frac{kk_i\tau^2 - 2kk_p\tau - 2}{k(k_p - k_i\tau)(\alpha-1)\alpha}$ $t_1 = (1-\alpha)T,\ t_2 = \alpha T$ $\Delta_{po} = \frac{\delta}{2}\frac{kk_i\tau^2 - 2kk_p\tau - 2}{k_p - k_i\tau}$ $\Delta_{co} = \frac{k_p^2 + k_i}{k}\Delta_{po}$	$T = \frac{1}{2}\frac{kk_i\tau^2 - 2kk_p\tau - 2}{k(k_p - k_i\tau)(\alpha - 1 + \frac{d}{\delta})(\alpha + \frac{d}{\delta})}$ $t_1 = (1-\alpha-\frac{d}{\delta})T,\ t_2 = (\alpha + \frac{d}{\delta})T$ $\Delta_{po} = \frac{\delta}{2}\frac{kk_i\tau^2 - 2kk_p\tau - 2}{k_p - k_i\tau}$ $\Delta_{co} = \delta$
$\frac{k}{s^2}e^{-\tau s}$	Limit cycle does not exist	
$\frac{e^{-\tau s}}{\tau_1 s+1}$	T, Δ_{po}, and Δ_{co} can be obtained using numerical methods	
$\frac{e^{-\tau s}}{(\tau_1 s+1)(\tau_2 s+1)}$		

Table 2.1: Summary table with the limit cycle periods and amplitudes. Δ_{po} denotes the amplitude of the process output, and Δ_{co} the amplitude of the controller output.

$$y(t) = \begin{bmatrix} 1 & 0 & 0 \\ k_p & 0 & k_i \end{bmatrix} \begin{bmatrix} x_p(t) \\ \dot{x}_p(t) \\ x_c(t) \end{bmatrix}. \tag{2.50}$$

As in the previous case, the solutions of the equations must be found by means of numerical tools.

2.4.4 Implementation in MATLAB

The algorithm has been implemented in MATLAB to obtain the periods and amplitudes of the simulation examples and the models identified from experimental data. The code is shown in Listing 2.1. First, the system matrices are defined, corresponding to the PI controller with sampling at the process output (lines 7-11), and with the sampling at the controller output (lines 12-16). Then, the type, order, and parameters of the sampler are stored in the variables `sampling`, `delta`, and `alpha` (lines 17-22). Finally, the set of equations if defined and solved in lines 33-38.

```matlab
%% Implementation of the Limit Cycle Finder Algorithm
clear all; clc;
numberOfProcessStates = size(Ap, 1);
numberOfProcessInputs = size(Bp, 1);
numberOfProcessOutputs = size(Cp, 1);
I = eye(numberOfProcessStates+1);
% Process Sampling
Aps = [Ap ki*Bp; zeros(1, numberOfProcessStates+1)];
Bps = [kp*Bp Bp; 1 0];
Cps = [Cp zeros(numberOfProcessOutputs, 1);];
Dps = [zeros(numberOfProcessOutputs, numberOfProcessInputs+1); -kp zeros(1,
        numberOfProcessInputs)];
% Controller Sampling
Acs = [Ap ki*Bp; zeros(1, numberOfProcessStates+1)];
Bcs = [kp*Bp Bp; 1 0];
Ccs = [Cp zeros(numberOfProcessOutputs, 1);];
Dcs = [zeros(numberOfProcessOutputs, numberOfProcessInputs+1); -kp zeros(1,
        numberOfProcessInputs)];
% Type of sampling ('process', 'controller')
sampling = 'process';
delta = 1.0;
alpha = 0.5;
% Hysteresis bands
n = 1;
% Switchings
if (alpha == 0.0)
    m = 2*n-2;
    u = [(alpha-n+1):(alpha+n-2); ones(1,m)*disturbance];
elseif (alpha == 0.5)
    m = 2*n-1;
    u = [(alpha-n+1):(alpha+n-1); ones(1,m)*disturbance];
end

tic,
for tau = 0:0.1:1
   f = @(t) slc(t, u, Aps, Bps, Cps, tau);
   Tguess = 5;
   k = Tguess*rand(1, m);
   [sol, val] = fsolve(f, k);
end
elapsedTime = toc;
```

Listing 2.1: Limit Cycle Finder algorithm.

2.5 Simulation results

This section shows simulations which illustrate the behaviour of the different combinations of control schemes and processes commented in the previous sections.

2.5.1 PI-IPTD-SOD$_n$ and PI-SOD$_n$-IPTD

Let us consider an IPTD process controlled by a PI controller with event-based sampling, and let the values of the plant parameters be $k = 1.0, \tau = 0.2$. The controller gains $k_p = 1.2, k_i = 1.0$ have been chosen to produce a limit cycle with two states, and the sampler $\alpha = 0.5, \delta = 0.1$. The system is described by the following expressions,

$$\begin{bmatrix} \dot{x}_p(t_k) \\ \dot{x}_c(t_k) \end{bmatrix} = \begin{bmatrix} 0 & 0 \\ 1.0 & 0 \end{bmatrix} \begin{bmatrix} x_p(t_k) \\ x_c(t_k) \end{bmatrix} + \begin{bmatrix} 1 & 0 \\ 1.2 & 1 \end{bmatrix} \begin{bmatrix} u_{nl_k} \\ d \end{bmatrix}. \tag{2.51}$$

There exists a symmetric limit cycle with two states (see Figure 2.7.a), with period $T = 5.6093$ and amplitude $\Delta_{po} = 0.2095$, which have been computed with the algorithm presented in Section 2.4.2. With the chosen gains, the system converges to the limit cycle after introducing a step change in the set-point.

When the sampler is placed at the controller output, the limit cycle that appears (see Figure 2.7.c) has the same period $T = 5.6093$ but different amplitude $\Delta_{po} = 0.1402$. While in the first case an external disturbance does not vary the properties of the limit cycle, for the second case it does as it is shown in Figure 2.8.

Varying the parameters of the controller it is possible to obtain limit cycles with higher number of levels. For example, increasing the integral gain of the controller to $k_i = 2.0$, and also the order of the sampler to $n = 2$, the system with the sampler at the process output presents the response shown in Figure 2.7.b and with the sampler at the controller output it has the response of Figure 2.7.d. The period and amplitude of the oscillation computed are $T = 4.9745$ and $\Delta_{po} = 0.3584$, which correspond to the results obtained in the simulation.

The simulations show that the system, with the chosen parameters, converges to a limit cycle even in presence of constant disturbances. The local stability of the limit cycles can also be proven by applying Proposition 3. As an example, for the two-level limit cycle, looking at the eigenvalues of the matrix $W = W_2 W_1$, where

$$W_i = \begin{bmatrix} 1 - u_{nl_i} & 0 \\ -x_{pi} + 1.2(u_{nl_i} + d) + t_i^* & 1 \end{bmatrix}.$$

After straightforward computations, it can be shown that the eigenvalues of W

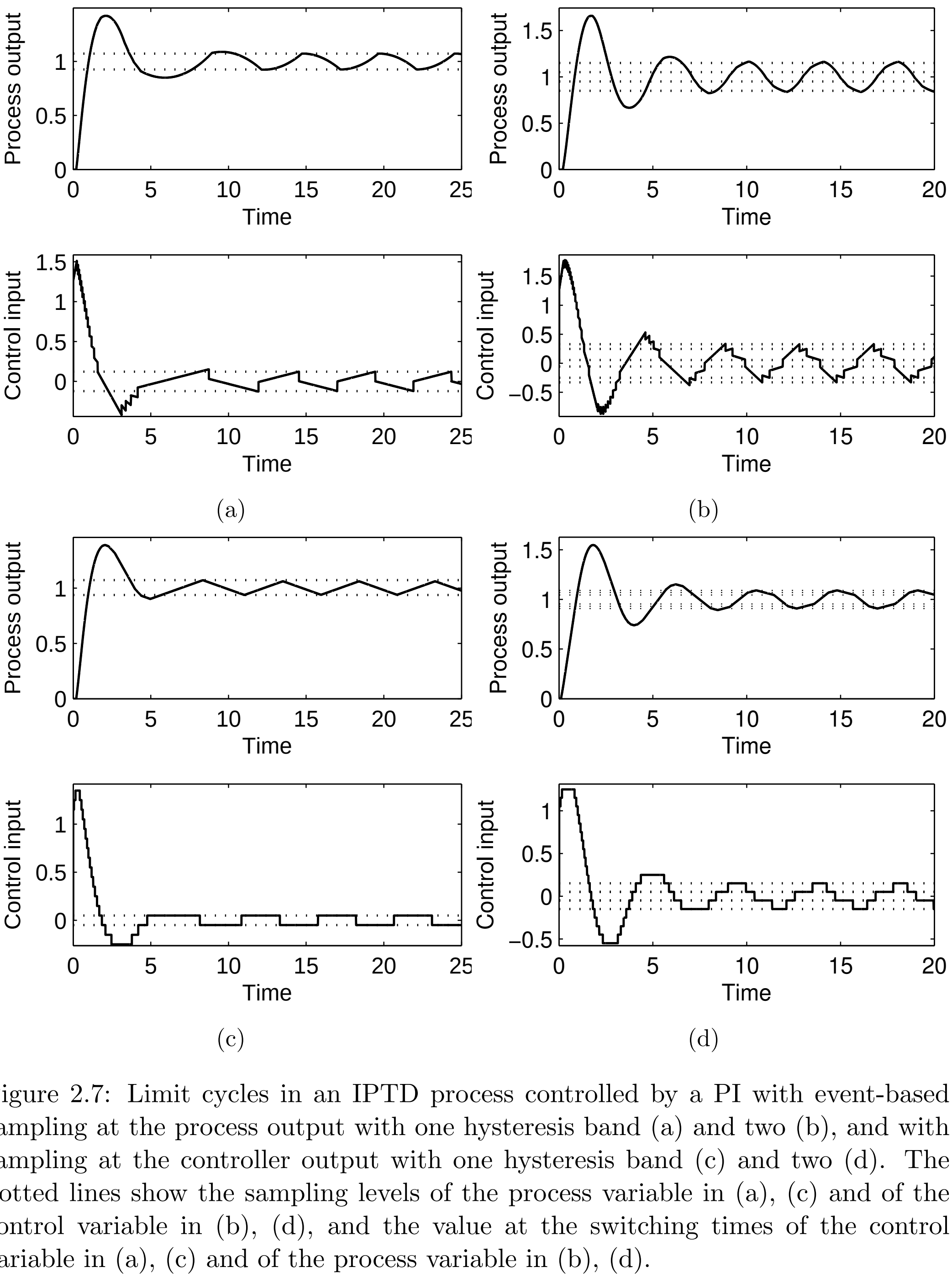


Figure 2.7: Limit cycles in an IPTD process controlled by a PI with event-based sampling at the process output with one hysteresis band (a) and two (b), and with sampling at the controller output with one hysteresis band (c) and two (d). The dotted lines show the sampling levels of the process variable in (a), (c) and of the control variable in (b), (d), and the value at the switching times of the control variable in (a), (c) and of the process variable in (b), (d).

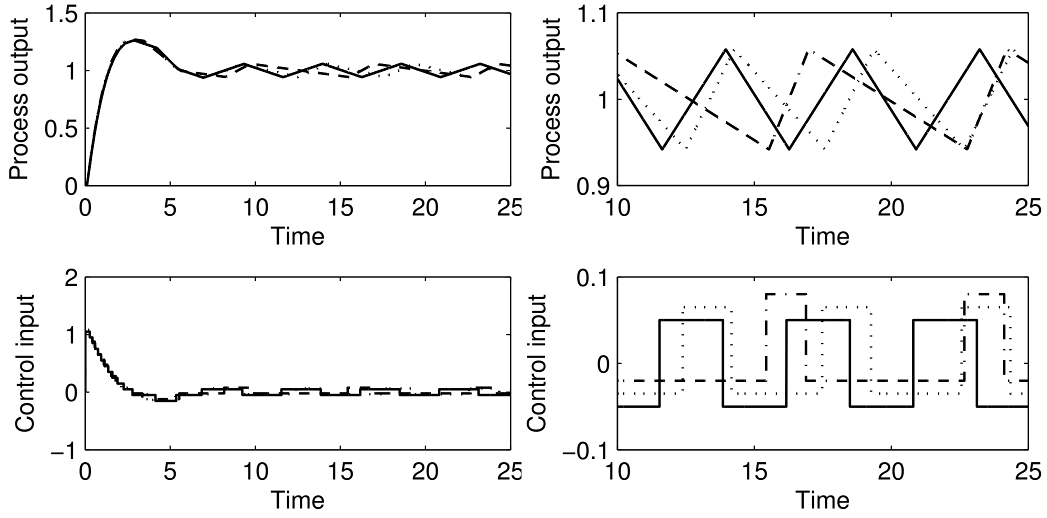


Figure 2.8: Limit cycles in an IPTD process controlled by a PI with event-based sampling at the controller output (a) with an external disturbance $d = 0.0$ (solid line), $d = 0.015$ (dotted line), and $d = .03$ (dashed-dotted line). (b) Detail of the limit cycles.

are $\lambda_1 = 1, \lambda_2 = (1-\alpha)\alpha < 1$, and therefore the limit cycle is locally stable. If limit cycles with $2n$ switchings are considered, then the eigenvalues of W are $\lambda_1 = 1, \lambda_2 = \prod_{i=0}^{2n-1}(1-\alpha-n+i)$. It is easy to verify that $|\lambda_2| \leq 1$ for every α when $n = 1, 2$, i.e. the corresponding limit cycles are locally stable. However, for $n > 2$ the local stability of the limit cycles depends on the particular value of α.

2.5.2 PI-SOPTD-SOD$_n$

Now, consider a SOPTD with parameters $k = 1.0$ (gain), $\tau_1 = 1.0$, $\tau_2 = 0.5$ (time constants), and $\tau = 0.2$ (time delay), which is controlled by a PI with event-based sampling placed at the process output, with $\alpha = 0.5$ and $\delta = 0.1$. Setting the controller gains to $k_p = 1.2$ and $k_i = 1.0$, the matrices A and B of the system are,

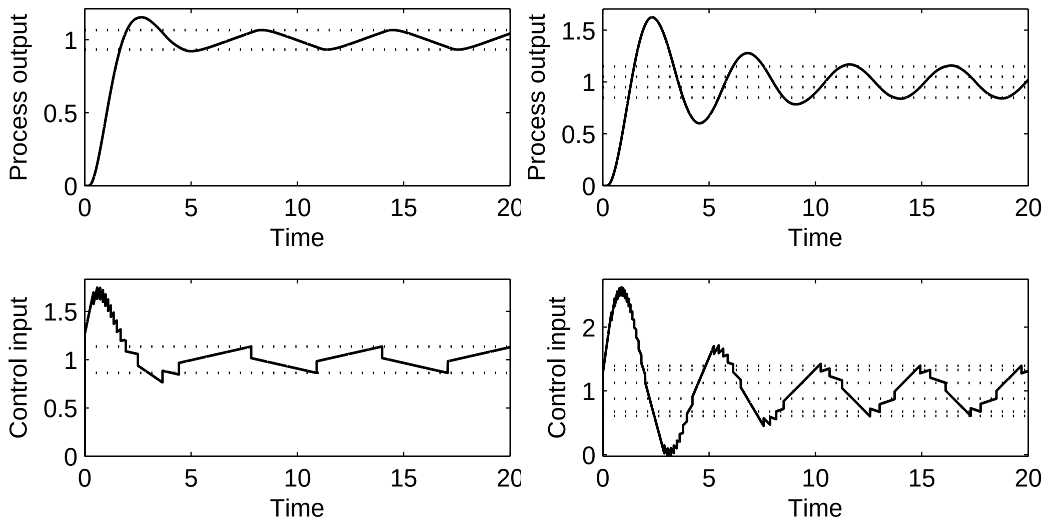


Figure 2.9: Limit cycles in an SOPTD process controlled by a PI with event-based sampling at the controller output, involving (a) one, and (b) two hysteresis bands.

$$A = \begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & 1 \\ 2 & -2 & -3 \end{bmatrix}, \qquad B = \begin{bmatrix} 1 & 0 \\ 0 & 0 \\ 2.4 & 2 \end{bmatrix}. \tag{2.52}$$

Solving the system of equations corresponding to the limit cycle with one band of hysteresis, the values of the switching times and levels are,

$$t_1 \approx 3.0772, t_2 \approx 3.0772,$$
$$\dot{x}_{p_1} \approx -0.0517, \dot{x}_{p_2} \approx 0.0517,$$
$$\dot{x}_{c_1} \approx -0.0669, \dot{x}_{c_2} \approx 0.0669.$$

Finally, the period is $T = t_1 + t_2 \approx 6.1545$ and the amplitude of the process output is $A \approx 0.1338$. This limit cycle, and another composed of two hysteresis bands, are plotted in Figure 2.9.

It is interesting to note that, when the sampler is placed at the process output, the limit cycle does not vary when a constant disturbance affects the system, because it is rejected by the controller.

The local stability of the limit cycle can be studied by looking at the eigenvalues of the matrix W,

$$W = W_2 W_1 = \begin{bmatrix} 0 & 0 & 0 \\ -0.0787 - 2.5953d & 0.0042 & 0.0021 \\ 0.0699 + 2.4253d & -0.0042 & -0.0021 \end{bmatrix}. \tag{2.53}$$

Since the eigenvalues of W are inside the unit disk ($\lambda_1 \approx .0021$, $\lambda_2 \approx 0.0000$,$\lambda_3 \approx 0.0000$), the limit cycle is locally stable.

2.6 Experimental results

This section shows experimental results which clearly evidence the existence of the results derived in the previous sections in a real system. Therefore the experiences carried out were focused on finding limit cycles in the Acurex system to compare them with that predicted by the theory and simulations. As shown in the following paragraphs, it is worth noting that even when the model used is a simplification of the process which ignores many of the complex dynamics existent in the system, the results are very close to those predicted in theory. Below, the Acurex system and the model identified from experimental data are presented, commenting some implementation issues of the controller and, finally, the obtained results are shown and interpreted.

2.6.1 The Acurex system

The experiments have been done with an equipment, known as Acurex, built in 1981 at the Almería Solar Platform (PSA, Spain [SD83]). In this plant (Figure 2.10), two types of collecting systems were considered, a central receiver system (CRS) and

Figure 2.10: Acurex distributed collector system at the Solar Platform of Almería (PSA), Spain.

a distributed collector system (DCS) using parabolic troughs. Parabolic trough systems concentrate sunlight onto a receiver pipe which contains a heat transfer fluid (HTF) that is heated as it flows along the receiver pipe. Then, the HTF is used to produce steam that may be used for example to feed an industrial process. A survey of basic and advanced control approaches for distributed solar collector fields can be found in [CRBV07a, CRBV07b]. For more information on control of solar plants see [CBR97].

2.6.2 The model

The plant was identified as a FOTPD, where the process input is the oil flow (l/s) in the pipes and the process output is the temperature of the oil at the collector field outlet (°C). There are unmodeled dynamics that are considered as external disturbances, such as the oil temperature at the input, or the solar irradiance. However, because of the time scale of the tests performed, which is smaller than the rate of variation of these variables (under clear day conditions without clouds), the model obtained seems to be a valid representation of the process for our purposes.

The transfer function was identified from experimental data obtained from the plant in a set of step tests. The procedure followed in each test is to drive the temperature manually to the working point, and when the process has reached it to introduce a step change in the input, registering the data measured from the sensors until the process stabilizes again. The FOPTD model, obtained by applying a least-squares procedure, is,

$$P(s) = -\frac{6.0715}{103.2723s + 1}\mathrm{e}^{-67.5021s}, \tag{2.54}$$

where the time constant τ_1 and time delay τ are given in seconds, and the gain k in $^{\circ}Cs/l$. The tests were carried out with an input around 8.5 l/s, being the range of the pump from 2 to 12 l/s. Also, the time constants and delays obtained make sense from the previous published works in this field. Notice the minus sign in (2.54), which represents the inverse response of the plant, i.e. a positive change in the flow produces a negative change in the temperature.

2.6.3 Implementation

The Acurex system has a SCADA software developed in LabVIEW. This software provides the user with an interface to execute its own controller implementation in MATLAB code. Thus, one must write a MATLAB callback function which is invoked with a configurable sampling period (it was fixed to $T_s = 15$s). This function receives the measures from the sensors, updates the controller state and, finally, sends the new control action to the actuators.

The controller implementation can be configured to work in three modes, namely,

-*manual*, the control input is set manually,

-*SOD-PI*, the sampler is at the process output, and

-*PI-SOD*, the sampler is at the controller output.

An excerpt of the code of the controller is shown in Listings 2.2 and 2.3.

2.6.3.1 Controller sampling

The first set of tests presented were carried out with the controller in *PI-SOD* mode, with the purpose of reproducing the two-level limit cycles obtained in simulation (with $\alpha = 0.5$) and the three-level (with $\alpha = 0.0$). The procedure followed is the same for each test, first the system temperature is moved to an operating point, and next when the process reaches a steady state a set-point step change is introduced into the controller.

```
% Sampling at the process output
  e = setpoint - output;
  if(~((u_prev >= umax && e > 0) || (u_prev <= umin && e < 0)))
    I = I + e_prev*Ts;
  end

  % event detection
  if (abs(e - e_prev) > delta)
    levels = floor(abs(e - e_prev) / delta);
    e_prev = e_prev + sign(e - e_prev)*delta*levels;
    u_prev = sat(Kp*e_prev + Ki*I, umin, umax);
    I = (u_prev - Kp *e_prev) / Ki;
  end
```

Listing 2.2: Code of the controller with the sampler at the process output.

```
% Sampling at the controller output
  e = setpoint - output;

  % anti-windup
  if(~((u_prev >= umax && e > 0) || (u_prev <= umin && e < 0)))
    I = I + e*Ts;
    u = Kp*e + Ki*I;
  else
    u = u_prev;
  end

  % event detection
  if (abs(u - u_prev) > delta)
    levels = floor(abs(u - u_prev) / delta);
    u_prev = sat(u_prev + sign(u - u_prev)*delta, umin, umax);
  end
```

Listing 2.3: Code of the controller with the sampler at the controller output.

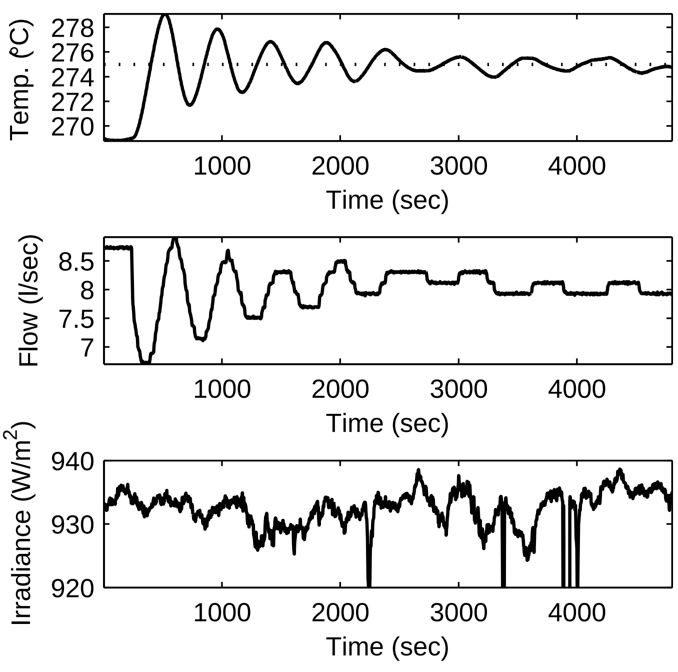


Figure 2.11: Response of the Acurex system (solid line) to a step change in the setpoint, with the event-based sampler placed at the controller output.

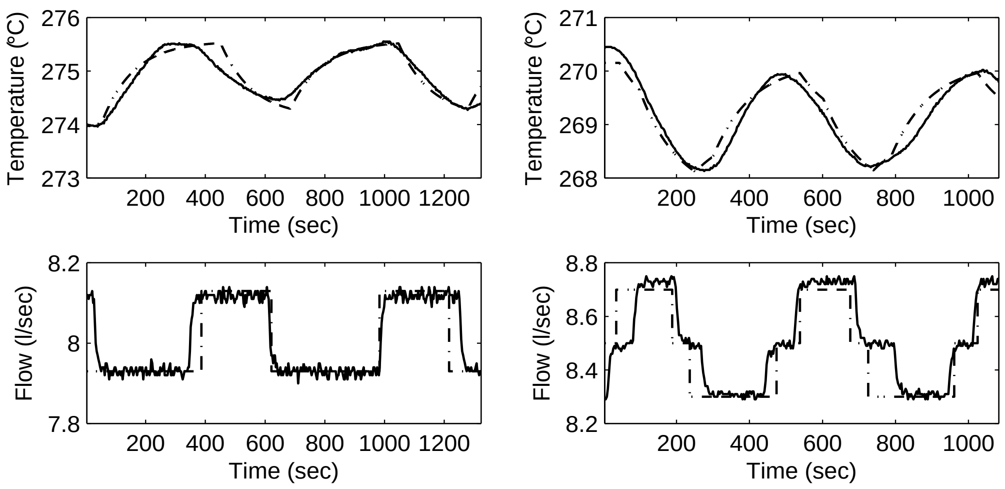


Figure 2.12: Limit cycles in the Acurex system (solid line) and in the simulated model (dashed line) controlled by a PI with SOD sampling placed after the controller output. (a) With two levels, and (b) with three levels.

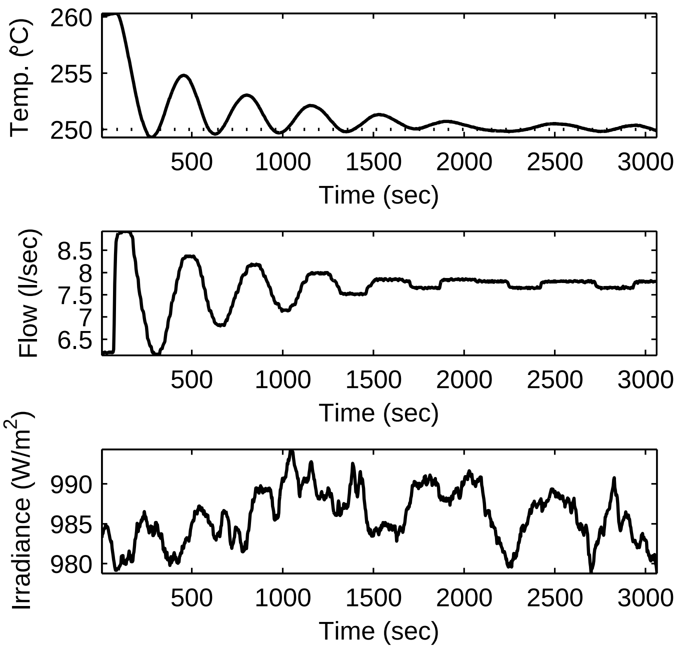


Figure 2.13: Response of the Acurex system (solid line) to a step change in the setpoint, with the event-based sampler placed at the process output.

The response is shown in Figure 2.11, where the oil temperature, the flow, and the solar irradiance during the test are plotted. It can be seen that the system goes into a limit cycle composed of two levels, which is similar to the results obtained in simulation with the FOPTD model.

Figure 2.12a and 2.12b show the comparison of the limit cycles obtained in simulation with the model and the results obtained with the Acurex system. The results are similar both qualitatively (limit cycles with two states), and quantitatively (the period and amplitude are approximately equal).

2.6.3.2 Process sampling

The second set of tests were carried out with the sampler placed at the process output. After verifying that, as in the previous section, the system enters into a limit cycle of two levels (Figure 2.13), the existence of more complex limit cycles

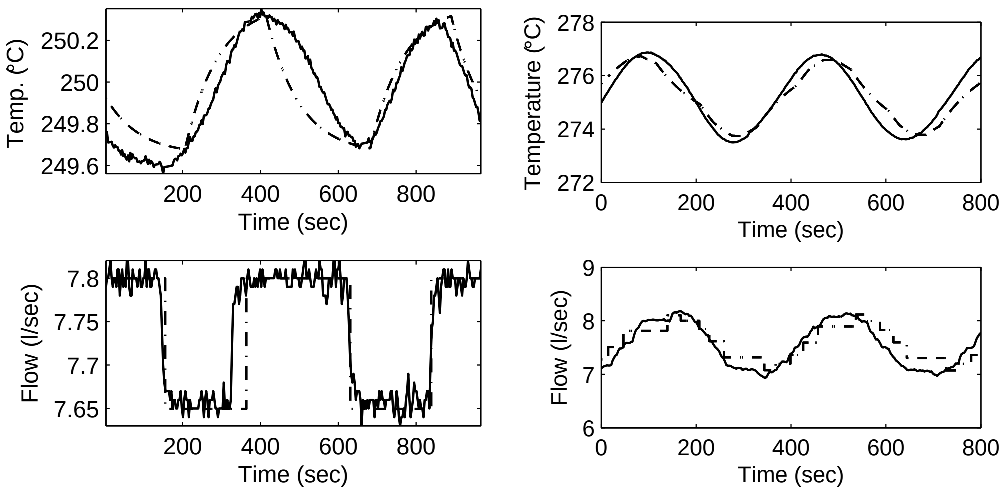


Figure 2.14: Limit cycles in the Acurex system (solid line) and in the simulated model (dashed line) controlled by a PI with SOD sampling placed after the process output. (a) with two levels, and (b) with eight levels.

was investigated. Increasing the proportional gain, a limit cycle with eight different levels was found, which is shown in Figure 2.14. It is remarkable that even in this case, the comparison between the experimental data and the simulated process shows that there are no significative differences in the behaviour.

2.7 Conclusions

The behaviour of a control system based on the use of a level crossing sampling either in the process output or in the control output has been studied. Limit cycles are of particular interest since they are associated to oscillations in processes, and therefore it is worth to have knowledge about them in order to avoid them when possible or to assure that they are not problematic.

When trying to find properties about the limit cycles, it is common to have system of equations which involve transcendent functions and thus it is not possible, in general, to find closed-form solutions. Moreover, due to the combinatorial explosion, it can be computationally expensive to find these solutions, and it becomes harder when higher order process model are considered.

Therefore, an algorithm to analyze the properties of the limit cycles has been proposed; it allows us to introduce some knowledge in the problem statement, so that the complexity can be reduced.

A set of simulation results illustrates the behaviour of the controllers with a set of processes models used very frequently in industrial context, which are the IPTD, the FOPTD, and the SOPTD. Also, this behaviour has been tested and verified in the Acurex Field of the Solar Platform of Almería, Spain. The experiments performed confirm that the simulation results can be extrapolated to real cases, obviously with the divergences due to unmodeled dynamics of the process, disturbances, etc.

3 Building Event-based Virtual Labs with EJS Elements

3.1 Introduction

Since control engineering is an applied science, it becomes important to have experimentation environments to help students assimilating the concepts studied. Ideally, this need is fulfilled by the use of laboratories. However, it is not always possible to have the necessary infrastructures [CTG$^+$04], mainly because they are expensive and not affordable by many universities. As remarked in the literature [Dor04, DDCD$^+$05], in this context interactive learning software tools provide an invaluable help to complement the learning process. This approach, as opposed to physical laboratories, in general does not require a high investment, thus being adequate not only for large universities but also for other with less resources.

Many examples of interactive tools can be found in literature. For example, a complete set of interactive tools for learning automatic control is presented in [GCBD12]. A tool for teaching system identification is presented in [GRDB12]. In [JGA98], authors implement a control learning environment as a collection of small modules, where each module shows a specific concept. In [KK12], an interactive tool for state space control design is developed, and [RMV09] presents a tool for teaching basic control concepts with first order time delay models and PI controllers.

Though there are many different development environments and programming languages that can be used to build virtual laboratories with varying difficulty and efficiency, one of them is *Easy Java Simulations* (EJS), a modeling and authoring tool designed to simplify the development of interactive dynamical simulations [CE07, EJS12]. The use of EJS to develop software tools to teach control engineering concepts has proven to be unvaluable, and it also has been successfully applied in the context of remote labs and hardware-in-the-loop systems. With the features provided by this tool, a simulation can be created without being an expert programmer. For example, EJS implements several solvers (the code which integrates the differential equations) and an editor where the model equations can be directly introduced as ordinary differential equations (ODEs). Recently, a mechanism to simplify the development of simulations has been introduced in EJS: the subpanel *elements*. An element is a ready-to-use component that implements a functionality. For example, a *file chooser* which shows a dialog to choose a file to be loaded in the simulation, or a *PID controller* that can be used to build a control loop.

Using this mechanism, a library of Java classes and EJS elements have been developed to increase the productivity when simulating interactive dynamic control systems with EJS. With this framework, a wide range of dynamic process control simulations can be easily built.

The design of the library is inspired by the block diagram editor provided by other well-known simulation tools as SIMULINK. With these tools, the user creates a model interconnecting different blocks. This approach has several advantages as, for instance, intuitiveness, and robust and modular design.

The library provides the user with the implementation of the most frequently used systems, such as systems described by state-space expressions, PID controllers, or non-linear systems. A mechanism to extend the functionality of the built-in

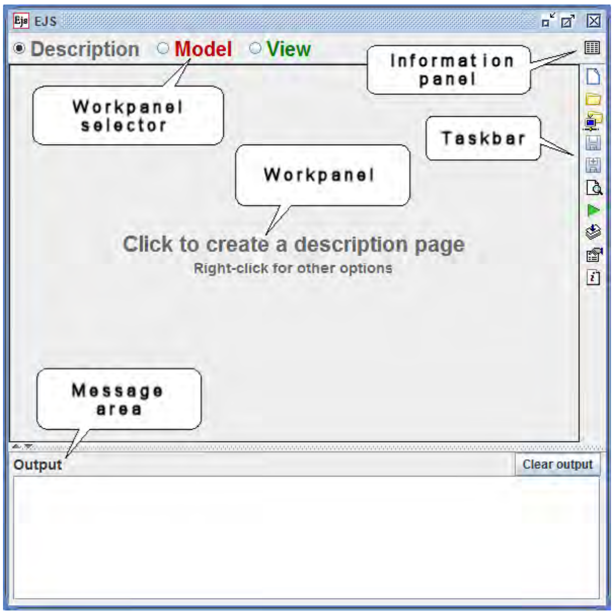


Figure 3.1: Graphical user interface of EJS.

library blocks is provided for advanced users. This can be useful if there is a need to use a specific component that is not implemented in the library.

The use of the library is illustrated with an application example built by combining several of the available elements at the library: the simulation of a control loop composed of a SISO process and a PID controller with send-on-delta sampling.

3.2 A brief on Easy Java Simulations (EJS)

EJS is an open source software tool designed to create simulations in Java with high-level graphical capabilities and with an increased degree of interactivity. The tool provides its own mechanism for describing models of scientific and control engineering phenomena, and, as such, can be used to create virtual laboratories on its own.

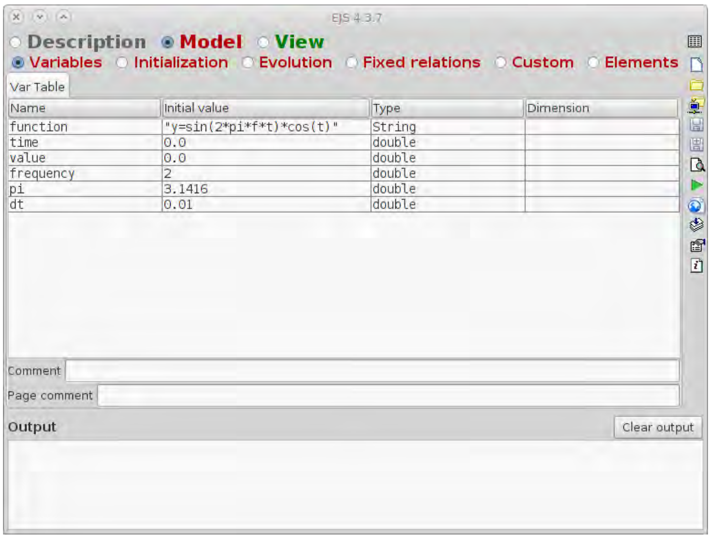
EJS is different from other authoring tools in that it is not designed to make life easier to professional programmers, but it has been conceived for science students and teachers. That is, for people who are more interested in the content of the simulation, the simulated phenomenon itself, and much less in the technical aspects needed to build the simulation.

The tool structures a simulation into two main panels, the **Model** and the **View**. Apart from the Model and the View, there is also an introductory part, named **Description**, to describe the system to be simulated using HTML files (see Figure 3.1).

The Model describes the simulated system by means of variables (both state variables and parameters), that completely characterize the system, and of computer algorithms that state how the system evolves in time and how it responds to user interaction. Authors need to declare the variables using a simple table, and write the Java code needed to specify the algorithms. EJS offers specialized help to solve models based on ODEs by providing an editor to write these equations and automatically generating the code required using the most popular solvers.

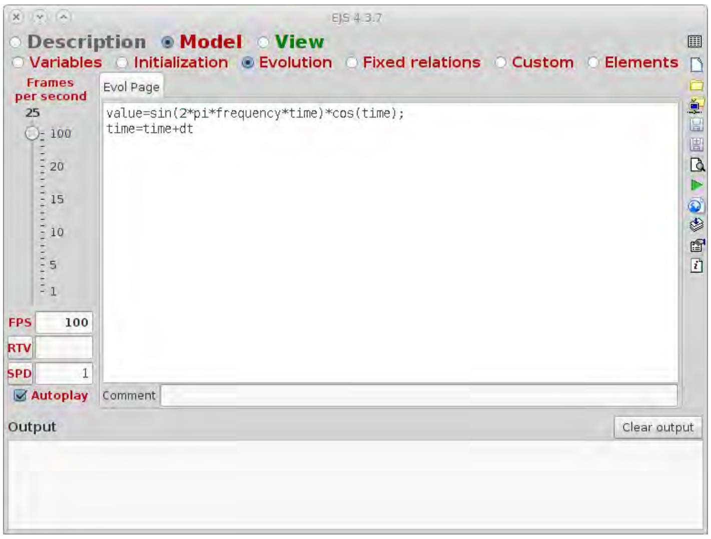
The Model is divided into six subpanels: *Variables*, *Initialization*, *Evolution*, *Fixed relations*, *Custom*, and *Elements*. In the Variables subpanel the global variables of the simulation are declared. The Initialization subpanel allows authors to execute initialization code before stepping the simulation. In the Evolution subpanel authors can input two type of descriptions: pure Java code or ODEs by using the editor. Both types of descriptions are evaluated continuously while the simulation is performed. The Fixed relations subpanel provides an additional way to execute Java code when the user interacts with the view while the simulation is paused. The Custom subpanel can be used by authors to implement their own Java methods. The Elements panel shows the model elements, a mechanism created to facilitate the creation and use in EJS of objects of existing Java classes. Figure 3.2a and Figure 3.2b show the Variables and Evolution subpanels, respectively.

The View provides the visualization of the simulated system, either in a realistic



Name	Initial value	Type	Dimension
function	"y=sin(2*pi*f*t)*cos(t)"	String	
time	0.0	double	
value	0.0	double	
frequency	2	double	
pi	3.1416	double	
dt	0.01	double	

(a) Subpanel Variables of EJS.



```
value=sin(2*pi*frequency*time)*cos(time);
time=time+dt
```

FPS 100
RTV
SPD 1
[✓ Autoplay]

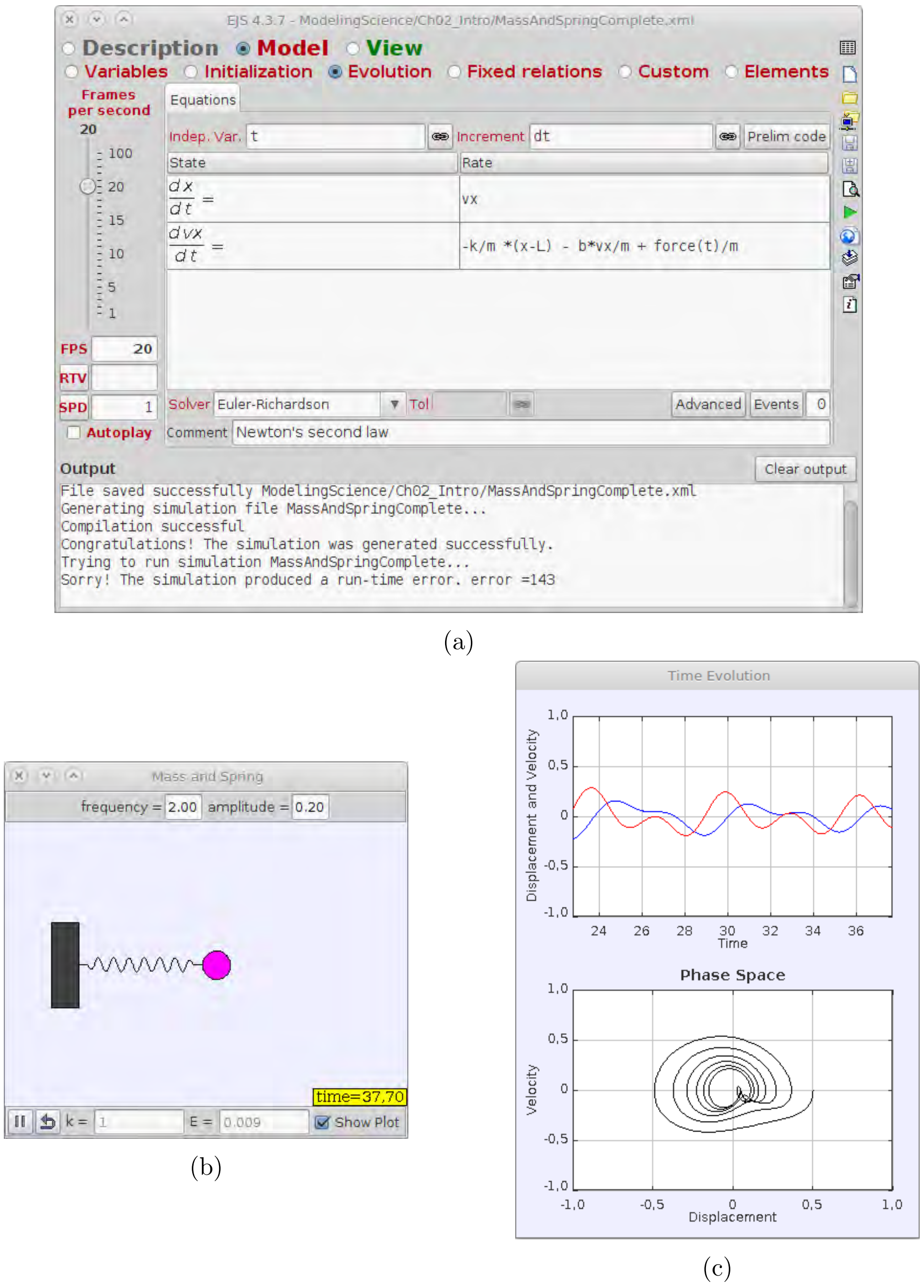
(b) Subpanel Evolution of EJS.

Figure 3.2: Subpanels of EJS.

form or using one or several data graphs, and the user interface elements required for the user interaction. These **view elements** can be chosen from a set of predefined, ready-to-use components, to build a tree-like structure. There are elements of different types. Each type specializes in a given visualization or interaction task, but can also be customized using the so-called **properties**, a set of internal values that modify the aspect and behaviour of the element on the screen. This way, the job of the author when building the view consists in choosing the right elements from those offered and in customizing them to define the level of interaction of the user with the simulation.

Both, model, and view need to be interconnected. Any change in the model state must be immediately reflected by the view in order to keep a dynamic on-the-fly visualization of the system. In turn, any interaction of the user with the view must immediately affect the model so that the desired interactivity is achieved. This communication is based on connecting model variables and view elements properties. This connection is very easily established by typing, in the table of properties of the view elements, the names of the model variables to be connected to the properties. Once the model and the view have been created and the required connections established, EJS creates the ready-to run simulation at a single mouse click, taking care of a good number of technical issues that thus becomes completely transparent to the author. The result is an independent, high performance, interactive simulation which can either be run as a stand-alone Java program, or be embedded as an applet in an HTML page. More description about using Easy Java Simulation, some examples, and the software can be obtained from *http://www.um.es/fem/Ejs/*.

A simple example is the simulation of the motion of a mass m situated at the end of a spring of length l and negligible mass. The reaction of the spring to a displacement dx from the equilibrium point is modeled using Hooke's law, $F(x) = -kdx$, where k is a constant which depends on the physical charasteristics of the spring. Thus, applying Newton's Second Law, the resulting second-order differential equation is $\frac{d^2x}{dt^2} = -\frac{k}{m(x-l)}$. This equation (the model) can be introduced in an *ODE Editor Page* (see Figure 3.3a), and the simulation is computed with the numerical solver of EJS. The view is created with graphical elements provided by EJS, and linked to the model variables. On the one hand, a visual animation of the mass and



EJS 4.3.7 - ModelingScience/Ch02_Intro/MassAndSpringComplete.xml

Description [✓ Model] View

Variables Initialization [✓ Evolution] Fixed relations Custom Elements

Indep. Var. t — Increment dt

State	Rate
$\frac{dx}{dt}$ =	vx
$\frac{dvx}{dt}$ =	-k/m *(x-L) - b*vx/m + force(t)/m

Solver Euler-Richardson

Comment Newton's second law

FPS 20 — SPD 1

Output

```
File saved successfully ModelingScience/Ch02_Intro/MassAndSpringComplete.xml
Generating simulation file MassAndSpringComplete...
Compilation successful
Congratulations! The simulation was generated successfully.
Trying to run simulation MassAndSpringComplete...
Sorry! The simulation produced a run-time error. error =143
```

Mass and Spring

frequency = 2.00 amplitude = 0.20

time=37,70

k = 1 E = 0.009 Show Plot

Figure 3.3: Simulation of a simple mass and spring system. (a) The *model*, and the *view*: (b) a graphical animation and (c) two plots.

spring (see Figure 3.3b) allows students to have an intuitive knowledge by beholding the motion of the system. On the other hand, two plots show the evolution of the variables (see Figure 3.3c).

3.3 The Process Control Elements

3.3.1 The Process Control Library

In control engineering, as it occurs in other areas, it is common to construct complex systems models by using an approach based on hierarchical blocks. A well-known example in the scientific and academic community is MATLAB and its graphical block editor SIMULINK, which can be used to construct complex systems. For instance, in SIMULINK there is a set of tools which are widely used such as a *sum* block, a system defined by a *transfer function* or by *space state* matrices, and so on.

EJS is designed to hide low-level programming details to the user, and therefore it simplifies greatly the development of simulations. However, when one tries to implement a control engineering application in EJS, there are common tasks that must be coped with. For example, to implement a basic feedback loop, composed of a PID controller and a process defined by their transfer functions, a first approach can be to obtain an ODE representation of the system and to introduce them in an ODE page. Though this approach is valid, it has several drawbacks. Not only it requires some previous work to convert the differential equations, but it could be difficult to adapt or modify the simulation later.

Therefore, it would be useful to have a library of control elements which implement the more common blocks that allow to construct a generic control system, and at least a set of tools to perform basic analysis of the system. In this context is where our library fits. In the following paragraphs we present the design, implementation, and examples of use in real case studies.

In the most popular control engineering software tools, such as the *LabVIEW control and simulation toolkit* or *SIMULINK*, it is common to have a division between blocks with continuous dynamics and blocks with discrete dynamics. Thus, we have followed the same paradigm in the design of our library: The *Process Control Library.*

The *Process Control Library* is composed of two conceptually differentiated parts. On the one hand, the core of the library is a framework that contains the Java classes and interfaces with the actual implementation of the components needed to define a control loop. This is done by interconnecting different kinds of blocks, which implement the most frequently used components, such as a state-space model or a PID controller, together with a mechanism to define user-specific blocks.

On the other hand, the second part of the library is the implementation of the EJS elements, which provide an easy way to incorporate the library into EJS and assist the developer with the configuration and usage of the blocks. The architecture of the core of the library, i.e. the classes that provide the functionality, is described, and the elements that allow their use from EJS are presented in the following paragraphs.

3.3.2 Core architecture

The aim is to keep the architecture as simple as possible in order that users do not have to spend more effort than necessary. Though the systems considered in the library can be classified following different criteria (dynamics, number of input and outputs, function, etc.), the most relevant aspect concerning the design of the library is the system dynamics. The evolution of dynamical systems can be described using different frameworks. The two main paradigms are the *continuous* time versus the *discrete* time models. In the former, the dynamics are represented by means of differential equations that allow to compute the rate of change of the state variables (the minimum set of variables to describe the behaviour of a system) as a function

		Discontinuities in the states	
		No	Yes
Continuous Flow	Yes	Continuous	Hybrid
	No	Discrete	Event-Based

Table 3.1: System classification according to the dynamics.

of theirselves, the time and other external signals. In discrete time, the evolution of the system is described by difference equations which define how to compute the new state as a function of the previous state and/or inputs. But there are also systems which are better described by combining continuous and discrete time representations, as for example hybrid systems, which present continuous dynamics but the states (or even the dynamics) may have an abrupt change at certain times.

According to the dynamics of the system, the blocks of the library have been classified into four types (summarized in Table 3.1), namely,

- *Continuous* systems, with dynamics described by differential equations, which are integrated numerically by the solver to obtain the evolution.
- *Discrete* systems, which do not have a continuous flow, but they change their state with a constant sampling period.
- *Event-based* systems, which do not have a continuous flow, but they change their state only when some condition changes.
- *Hybrid* systems, which do have a continuous flow as continuous systems, but which can also change their state and/or their dynamics when some condition changes.

The framework provides the Java interfaces (see Figure 3.4) that define the contract that a class must fulfill to be considered one of the above mentioned kinds of blocks. The main component is the interface *Block*, shown in Listing 3.1, which provides methods needed to properly interact with the solver and/or interconnect

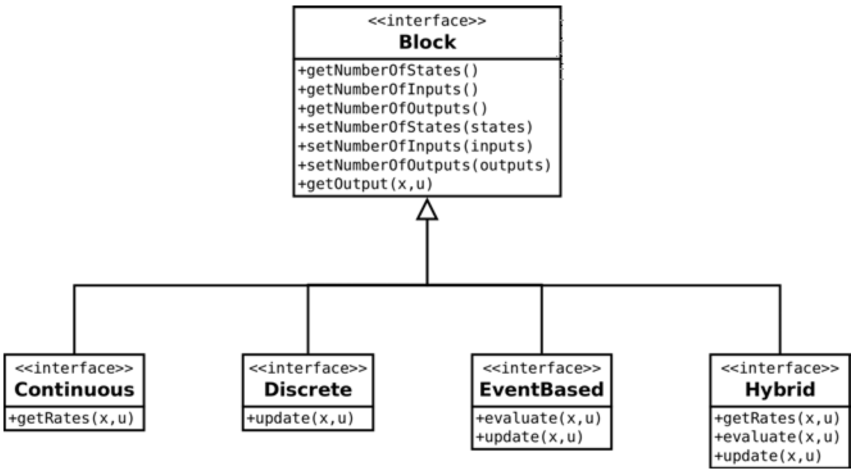


Figure 3.4: The diagram shows the different interfaces of the *Process Control Library* and their relationships.

with other blocks: to obtain and modify the number of states (*getNumberOfStates*, *setNumberOfStates*), inputs (*getNumberOfInputs*, *setNumberOfInputs*) and outputs (*getNumberOfOutputs*, *setNumberOfOutputs*), and to get the value of the outputs (*getOutputs*). This interface is the root of the hierarchy, providing a common interface between the four kinds of blocks mentioned above, and other types which may be defined by user code.

Though all the classes in the library implement this interface through one of its subinterfaces, there are only three classes that implement it directly,

- *AbstractBlock* This class is not directly instantiable, but it can be extended to define new blocks.

```java
/**
 * Block Interface for an object representing a simulation block.
 */
public interface Block {
  public int getNumberOfStates();
  public int getNumberOfInputs();
  public int getNumberOfOutputs();
  public int setNumberOfStates(int states);
  public int setNumberOfInputs(int inputs);
  public int setNumberOfOutputs(int outputs);
  public double[] getOutput(double[] x, double[] u);
  public double getOutput(int i);
}
```

Listing 3.1: Interface *Block*.

```java
/**
 * Class to implement a sum block.
 */
public class Sum extends AbstractBlock {
  /** The signs of the inputs */
  private double[] signs = new double[] { +1, +1 };
  /**
   * Create a new Sum Object.
   * @param signString The sign of the inputs.
   */
  public Sum(String signsString) {
    setSigns(signsString);
    setNumberOfInputs(signs.length);
    setNumberOfOutputs(1);
    setNumberOfStates(0);
  }

  /**
   * Set the sign of the inputs.
   * @param signString The sign of the inputs.
   */
  public void setSigns(String signsString) {
    boolean isValid = Pattern.matches("(\\+|-)+", signsString) &&
                      signsString.length == ninputs;
    if(isValid) {
      int size = signsString.length();
      signs = new Sign[size];
      for(int i=0; i<size; i++) signs[i] = (signsString.charAt(i) == '-') ? -1.0 : +1.0;
    }
  }

  /**
   * Compute the output of the block.
   * @return The signed sum of the inputs.
   */
  @Override
  public double[] getOutput(double[] x, double[] u) {
    double[] y = new double[]{0};
    for(int i=0; i<ninputs; i++) y[0] += signs[i]*u[i];
    return y;
  }
}
```

Listing 3.2: *Sum* Block.

- *Sum* Represents a sum operation, i.e. the output is equal to the sum of its inputs.

- *Saturation* A saturation block whose output is equal to its input if it is within a valid range, or to the saturation values otherwise.

The blocks are described in detail in the following lines.

AbstractBlock The *AbstractBlock* is an abstract (non-instantiable) class which implements the *Block* interface and some general utility methods. Its purpose is to serve as starting point to define new components with a common behaviour and to eliminate the need to repeat non-specific code.

```
/**
 * Interface of a Continuous Block.
 */
public interface Continuous extends Block {
  public double[] getRates(double[] x, double[] u);
  public void linkStates(double[] x);
}
```

Listing 3.3: Interface *Continuous*.

SUM The *Sum* is a utility block representing a signed sum operation. The block has a configurable number of inputs, being also possible to associate a *positive* or *negative* sign to each input, which can be done by means of the method *setSigns*, which receive a *String* with the signs ('+' or '-'). The number of inputs is inferred from the list of signs. Then, the output of the block is computed with the *getOutput* method, as the sum of its inputs premultiplied by their associated signs. An excerpt of the code of the *Sum* class is presented in Listing 3.2.

SATURATION The *Saturation* represents a non-linear function which is linear within the valid range, and saturated to a low/high level saturation when the value is outside. The block is stateless, and has one input and one output. The method *getOutput* return the saturated value of the input.

3.3.2.1 Continuous blocks

The *Continuous Blocks*, which represent systems with continuous dynamics, must implement the interface *Continuous*, given in Listing 3.3.

The most important method of this interface is *getRates* which must compute and return the derivatives of the states, needed to integrate the state by the solver. The other method, *linkStates*, allows to store an internal reference to the state vector of the block to be accessed within the block without the need of passing the reference as parameter. However, this feature must be carefully used since because it can lead to inaccuracies in the solution if it is not properly used. In general it is safer to use only the former method.

Currently, in the library there are three blocks implementing the *Continuous*

```
  /* Returns the derivatives of the states of system as dx(t)=A*x(t)+B*u(t), for the
   given values of x and u. */
  public double[] getRates(double[] x, double[] u) {
    double[] dx = new double[nstates];
    int ninputs = (u != null) ? u.length : 0;
    if(ninputs > this.ninputs) ninputs = this.ninputs;

    for(int i=0; i<nstates; i++) {
      dx[i] = 0;
      for(int j=0; j<nstates; j++) dx[i] += A[i][j]*x[j];
      for(int j=0; j<ninputs; j++) dx[i] += B[i][j]*u[j];
    }

    return dx;
  }


  /* Returns the output of the system as y(t)=C*x(t)+D*u(t), for the given values of x
    and u. */
  public double[] getOutput(double[] x, double[] u) {
    double[] y = new double[noutputs];
    int ninputs = (u != null) ? u.length : 0;
    if(ninputs > this.ninputs) ninputs = this.ninputs;

    for(int i=0; i<noutputs; i++) {
      y[i] = 0;
      for(int j=0; j<nstates; j++) y[i] += C[i][j]*x[j];
      for(int j=0; j<ninputs; j++) y[i] += D[i][j]*u[j];
    }

    return y;
  }
```

Listing 3.4: Code of the *getRates* and *getOutput* methods corresponding to the implementation of system (3.1).

interface. These are:

- *StateSpaceModel*, a linear state space model.
- *SisoPlant*, a specialization of the previous element, constraining it to the particular case of a SISO system.
- *PidController*, a complete implementation of a PID controller with anti-windup mechanism and derivative filter.

StateSpaceModel The *StateSpaceModel* class implements a linear system model represented in the space of states.

$$\begin{aligned} \dot{x}(t) &= Ax(t) + Bu(t) \\ y(t) &= Cx(t) + Du(t). \end{aligned} \tag{3.1}$$

Two important methods of this class are *getRates(double[] x, double[] u)* and

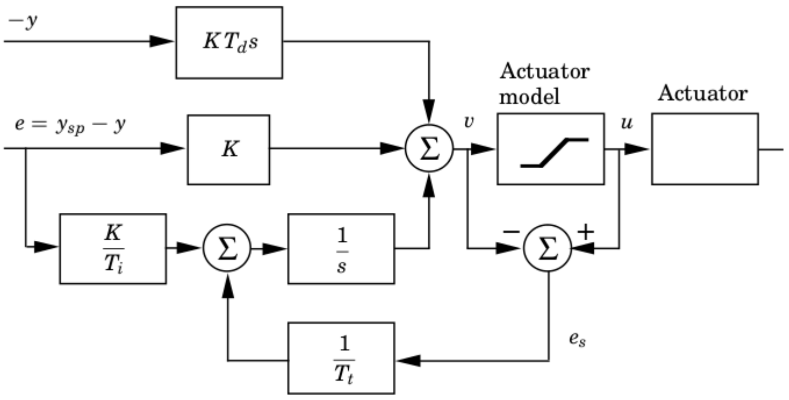


Figure 3.5: Structure of the PID controller with antiwindup mechanism (image taken from the book *Advanced PID Control*, [AH05]).

getOutput(double[] x, double[] u), required by the *Continuous* interface (the latter is inherited from the *Block* interface). The code of these methods, which implement the system described by (3.1), is shown in Listing 3.4.

PidController The *PidController* class implements a continuous PID controller with anti-windup mechanism, derivative filter, and the ability to perform tracking of an external input. The structure of the controller (see Figure 3.5) is similar to the described in [AH05]. The controller has been implemented as a specialization of the *StateSpaceModel* class, using the PID state space representation given in (3.2), which is reproduced here by convenience:

$$A = \begin{bmatrix} 0 & 0 \\ 0 & -\frac{n}{k_d} \end{bmatrix} \; B = \begin{bmatrix} 1 \\ nk_d \end{bmatrix} \; C = \begin{bmatrix} k_i & k_d \end{bmatrix} \; D = \begin{bmatrix} k_p + nk_d \end{bmatrix}. \tag{3.2}$$

The method *getRates(double[] x, double[] u)*, inherited from the *StateSpaceModel* class, has been overriden to add the antiwindup and bumpless transfer mechanism. Listing 3.5 shows the code of this method. The methods *setTracking* and *setAntiwindup* allow to enable or disable the antiwindup and bumpless transfer mechanisms,

```
/**
 * Enable or disable the antiwindup mechanism.
 * @param enabled The new value
 */
public void getRates(boolean enabled) {
  aintiwindup = enabled;
}

/**
 * Enable or disable the tracking mechanism.
 * @param enabled The new value
 */
public void setTracking(boolean enabled) {
  tracking = enabled;
}

/**
 * Compute the derivative of the state.
 * @param x The state
 * @param u The input
 */
@Override
public double[] getRates(double[] x, double[] u) {
  double[] dx = super.getRates(x, u), y = super.getOutput(x, u);

  if(antiwindup) {
    double v = (y[0] < uMin) ? uMin : (y[0] > uMax) ? uMax : y[0];
    dx[0] += ks*(v - y[0]);
  }

  if(tracking) {
    dx[0] += ks*(u[2] - y[0]);
  }

  return dx;
}
```

Listing 3.5: Code of the *getRates()* for the *PidController* class.

respectively.

3.3.2.2 Discrete blocks

The *Discrete Blocks*, which represent systems with discrete dynamics, must implement the interface *Discrete*, described in Listing 3.6. This interface defines one method, *update* which computes the new value of the discrete states of the block. The *Discrete* blocks currently implemented in the library are the following:

- *DiscreteStateSpaceModel*, a linear discrete-time state space model.
- *DiscretePidController*, implements a discrete time PID controller.
- *DiscreteFilteredPidController*, implements a discrete time PID controller with a first-order filter in cascade.

```java
/**
 * Interface of a Discrete Block.
 */
public interface Discrete extends Block {
  public void update(double[] x, double[] u);
}
```

Listing 3.6: Interface *Discrete*.

DISCRETESTATESPACEMODEL The *DiscreteStateSpaceModel* class implements a discrete-time linear system usign the state space representation:

$$\begin{aligned} x(k+1) &= Ax(k) + Bu(k) \\ y(k) &= Cx(k) + Du(k). \end{aligned} \tag{3.3}$$

The *update(double[] x, double[] u)* method implements the difference equation in (3.3) to compute and store the new value of the state which is passed as a parameter to the method, and the *getOutput(double[] x, double[] u)* method returns the output of the system. Both methods are required by the *Discrete* interface (the latter is inherited from the *Block* interface).

DISCRETEPIDCONTROLLER The *DiscretePidController* class implements a discrete-time PID controller with anti-windup mechanism, derivative filter, and the ability to perform tracking of an external input. The controller has been implemented as a specialization of the *DiscreteStateSpaceModel* class, using a discretization based on backward differences of the continuous PID state space representation in (3.2), which yields:

$$A = \begin{bmatrix} k_4 & k_2 & k_3 \\ 0 & 0 & 0 \\ 0 & 1 & 0 \end{bmatrix} \; B = \begin{bmatrix} k_1 \\ 1 \\ 0 \end{bmatrix} \; C = \begin{bmatrix} 1 & 0 & 0 \end{bmatrix} \; D = \begin{bmatrix} 0 \end{bmatrix}, \tag{3.4}$$

where $k_1 = k_p + k_i t_s + k_d/t_s$, $k_2 = k_p + \frac{2k_d}{t_s}$, $k_3 = \frac{k_d}{t_s}$, and $k_4 = \frac{1}{t_s}$. The parameters k_n correspond to the coefficients of the terms in the discrete PID difference equation, and which are computed with the controller gains k_p, k_i, and k_d, as defined

```
t = t + dt;
if(t >= nextTs) {
  pid.update(xc, uc); // xc is the state , and uc is the input
  nextTs = nextTs + pid.getTs(); // compute the next sampling time
}
```

Listing 3.7: Updating the DiscretePidController in an Step Page.

for the continuous PID, and the discretization time t_s. The latter one affects the convertion from the continuous representation to the discrete one, and can be used to dynamically vary the sampling time of the controller during the simulation, while maintaining the same controller gains.

The methods *update(double[] x, double[] u)* and *getOutput(double[] x, double[] u)*, inherited from the *DiscreteStateSpaceModel* class, have been overriden to add the antiwindup mechanism and the external input tracking function.

To use the *DiscretePidController* in a simulation, the method *update* must be called periodically with sampling period t_s, for example into an *Step Page* (see Listing 3.7).

DiscreteFilteredPidController The *DiscreteFilteredPidController* class implements a discrete-time PID controller with a first-order filter connected in cascade. Though it can be built by interconnecting a *DiscreteStateSpaceModel* for the filter and a *DiscretePidController*, it have been also included in the library for practical reasons.

The *DiscreteFilteredPidController* is very similar to the *DiscretePidController*, but the state space representation is slightly different because of the incorporation of the first-order filter:

$$A = \begin{bmatrix} k_4 & k_2 & k_3 \\ 0 & 0 & 0 \\ 0 & 1 & 0 \end{bmatrix} \; B = \begin{bmatrix} k_1 \\ 1 \\ 0 \end{bmatrix} \; C = \begin{bmatrix} 1 & 0 & 0 \end{bmatrix} \; D = \begin{bmatrix} 0 \end{bmatrix}, \tag{3.5}$$

where $\alpha = 1 + \frac{t_f}{t_s}$, $k_1 = \frac{1}{\alpha}(k_p + k_i t_s + k_d/t_s)$, $k_2 = \frac{1}{\alpha}(k_p + \frac{2k_d}{t_s})$, $k_3 = \frac{1}{\alpha}\frac{k_d}{t_s}$, and

```java
/**
 * Interface of an Event-Based Block.
 */
public interface EventBased extends Block {
  public double evaluate(double[] x, double[] u);
  public void update(double[] x);
}
```

Listing 3.8: Interface *EventBased*.

$k_4 = \frac{1}{\alpha}\frac{1}{t_s}$. The parameters k_i, k_p, k_i, k_d, and t_s have the same meaning as in the $DiscretePidController$. In addition, the parameter t_f is the time constant of the filter in cascade with the controller. The higher the value of t_f is, the lower is the cutoff frequency of the filter.

3.3.2.3 Event-based blocks

The *Event-based Blocks* must implement the interface *EventBased*, listed in Listing 3.8. The methods provided by this interface can be used in conjunction with the EJS event detector, which is a mechanism provided by EJS for detecting the zero crossing of a function. The value computed by the method *evaluate* (see Listing 3.8) is expected to be negative if and only if the state of the block must be updated. With the event solver, the very instant when that occurs can be detected, thus updating correctly the state.

At the time of writing this document, there is just one block implementing this interface:

- *SODSampler*, a SISO block, where its output is a send-on-delta sampling of the input ([CSVD12]).

SodSampler The *SODSampler* class implements a send-on-delta sampling. The behaviour of the sampler is determined by two parameters, δ and α, the sampling threshold and the sampling offset, respectively, as described in Section 2.1.1.

Two important methods are $evaluate(double[]x, double[]u)$, which returns the value of the event function, i.e. $f(x, u) = \delta - |x - u|$, and $update(double[]x)$, which updates the state (passed as parameter) with the correct value when a new event is detected. Another relevant method is *init(double[] x, double[] u)*, which initializes the state of the sampler to the correct value.

3.3.3 EJS Elements

The blocks discussed in the previous section implement all the functionality of the library, thus they can be incorporated directly into EJS through Java code. However, to simplify the use, each library block has been encapsulated inside an element. The elements currently developed are:

- *StateSpaceModelElement* This element creates a linear state space model.
- *SisoPlantElement* This element is an specialization of the previous element, constraining it to the particular case of a SISO system.
- *PidControllerElement* This element creates a block with a complete implementation of a PID controller with anti-windup mechanism and derivative filter.
- *DiscretePidControllerElement* This element creates a block with a complete implementation of a discrete time PID controller with anti-windup mechanism and derivative filter.
- *SaturationElement* This element creates a SISO block, where its output is equal to the input if its value is within the valid range, and the saturation value in the opposite case.
- *SourceElement* This element creates a block without inputs, and whose output is the value of an EJS variable.
- *SumElement* This element creates a block whose output is the sum of its inputs. The number of inputs and their associated signs are configurable.
- *SodSamplerElement* This element creates a SISO block, where its output is a send-on-delta sampling of the input.

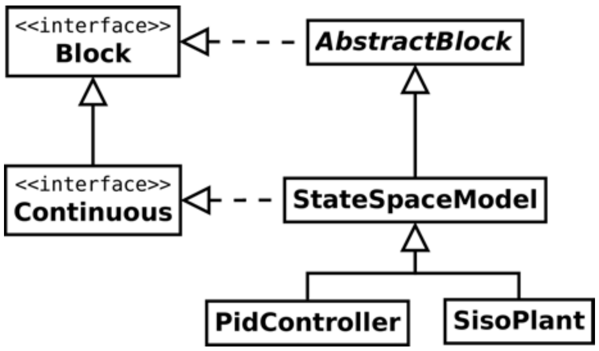


Figure 3.6: The class diagram illustrates the *Process Control Library* extension mechanism. The *AbstractBlock* is extended by the class *StateSpaceModel*, which is also especialized by the *PIDController* and *SisoPlant*.

3.3.4 Extending capabilities of the built-in classes

Though the library contains the basic elements, depending on the user needs it might be necessary to extend the capabilities of the built-in classes. To define a new block, it must implement the *Block* interface (or one of its subinterfaces such as *Discrete* or *Continuous*). This can be done in two ways, that are, by writing a new class that implements the interface or by subclassing a previously existent class (see Figure 3.6). The *AbstractBlock* class provided with the library implements the interface *Block* and some additional utility methods. Thus, the subclassing should be the first-option for defining a new block.

The two methods to extend the library are illustrated in the examples provided in the following paragraphs.

METHOD 1: SUBCLASSING The simplest way to create a new block is by specializing the *AbstractBlock* class. This is illustrated with the step-by-step creation of a new block: a *tank*, which implements the model of a water tank. The physical model of the tank is derived by means of the Bernoulli's laws and mass balances (Referencia). Consider a tank with constant section A, and an outlet hole with cross-sectional area a. Let $q_{in}(t)$ and $q_{out}(t)$ be the inflow and the outflow to the tank,

```
/**
 * Class to implement a Tank block
 */
public class Tank extends AbstractBlock implements Continuous {
  private final double g = 9.81, c = Math.sqrt(2*g);
  private double a, b;
  private double tankSection = 1;
  private double outletArea = 1;
  private double pumpConstant = 1;

  public Tank(double tankSection, double outletArea, double pumpConstant) {
    this.tankSection = tankSection;
    this.outletArea = outletArea;
    this.pumpConstant = pumpConstant;
    this.a = c*outletArea/tankSection;
    this.b = pumpConstant/tankSection;
  }

  @Override
  public double[] getRates(double[] x, double[] u) {
    if(x[0] >= 0) return new double[]{-a*Math.sqrt(x[0]) + b*u};
    return new double[]{k*u};
  }

  @Override
  public double[] getOutput(double[] x, double[] u) {
    return new double[]{x[0]};
  }

  public void setValvePosition(double alpha) {
    if(alpha < 0 || alpha > 1) return;
    this.alpha = alpha;
  }

  public double getSteadyStateInput(double hsp) {
    if(hsp < 0) return 0.0;
    return K * Math.sqrt(hsp);
  }

  public void setTankSection(double tankSection) {
    if(tankSection <= 0) return;
    this.tankSection = tankSection;
  }

  public double getTankSection() {
    return tankSection;
  }
}
```

Listing 3.9: Code of the *Tank* class.

respectively. The dynamics of the tank is described by the mass balance equations:

$$A\dot{h} = q_{in}(t) - q_{out}(t) = -a\sqrt{2gh(t)} + ku(t), \tag{3.6}$$

where h is the height of the water in the tank. Thus the tank block can be seen as a block with two inputs (the inflow and the outflow), and one output (the water level) block.

The *Tank* class is defined as a specialization of the *AbstractBlock* class. The code with the basic functionality is listed in Listing 3.9. The class constructor allows to pass the parameters of the tank model: the *tankSection*, the *outletArea*, and the

Discrete interface.

```
 1 /**
 2  * Class to implement a Zero Order Hold
 3  */
 4 public class ZeroOrderHold implements Discrete {
 5 
 6   public int getNumberOfStates() {
 7     return 1;
 8   }
 9 
10   public int getNumberOfInputs() {
11     return 1;
12   }
13 
14   public int getNumberOfOutputs() {
15     return 1;
16   }
17 
18   public void setNumberOfStates() {}
19 
20   public void setNumberOfInputs() {}
21 
22   public void setNumberOfOutputs() {}
23 
24   public double[] getOutput(double[] x, double[] u) {
25     return new double[]{x[0]};
26   }
27 
28   public void update(double[] x, double[] u) {
29     x[0] = u[0];
30   }
31 
32 }
```

Listing 3.10: Code of the *ZeroOrderHold* class

pumpConstant, for the initialization of the new instance being created. The method *getRates* computes the value of the derivative of liquid level as defined in (3.6), and the *getOutput* method returns the value of the output, which in this case is simply the value of the first state. The class also provides with the *getter* and *setter* methods to obtain or modify the values of the parameters.

Method 2: Implementing the interface Block. To illustrate the second method, let us define the class *ZeroOrderHold*: a block that samples the value of its input and holds it until the next update. The class implements the *Discrete* interface, an specialization of the *Block* interface. The method *update(double[] x, double[] u)* (Listing 3.10, lines 28-30) stores the value of the input in the state vector x passed as first parameter. The method *getOutput(double[] x, double[] u)* (Listing 3.10, lines 24-26) method returns the value of the zero-order hold. The other methods (Listing 3.10, lines 6-22) allow to get and set the dimension of the state, outputs, and inputs of the blocks.

3.4 Use of the library

The library is thought to be used in two different ways: by means of the *Elements*, which allows to easily incorporate the blocks into an EJS simulation, or directly by Java code, using the *OpenSource Physics* (OSP) framework. The former method draw upon the EJS capabilities to make it easier for the user. The latter one may be the preferred way when the complexity of the model is higher or when the simulation does not require interactivity.

3.4.1 Interconnecting the blocks

A model can be built by interconnecting the blocks of the library. All the blocks are assumed to have multiple inputs and multiple outputs of type *double*, which in practice implies that the interface methods work with *double*[].

The state of the *Continuous* blocks is computed by the ODE solver by numerically integrating the state derivatives, which are coded in the corresponding block *getState(double[] x, double[] u)* method. Therefore, for the simplest case, where the output of a block is the input to another block, they can be interconnected simply by passing the output of the first block as the second param to the *getState* method of the second block. An example is shown in Listing 3.11. Of course, not always the interconnections are so simple. Frequently, a block can have different inputs from multiple blocks. For instance, consider a system with three blocks: *block1*, *block2*, and *block3*, where the first output of *block1* is connected to the first input of *block3*, and the second output of *block2* is connected to the second input of *block3*. The code to implement this example is shown in Listing 3.12.

```java
double[] y1 = block1.getOutput(x1, u1);
double[] y2 = block2.getOutput(x2, y1);
double[] dx1 = block1.getRates(x1, u1);
double[] dx2 = block2.getRates(x2, y1);
```

Listing 3.11: Interconnecting two blocks.

```
double[] y1 = block1.getOutput(x1, u1);
double[] y2 = block2.getOutput(x2, u2);
double[] u3 = new double[]{u1[0], u2[0]};
double[] y3 = block3.getOutput(x3, u3);
double[] dx1 = block1.getRates(x1, u1);
double[] dx2 = block2.getRates(x2, u2);
double[] dx2 = block2.getRates(x3, u3);
```

Listing 3.12: Interconnecting three blocks.

In such a way, the model can be built for a number of subsystems and interconnections of arbitrary complexity. Nevertheless, it must be remembered that the extension mechanism provides the possibility to define new blocks, which may be a better option if the complexity of the system grows.

3.4.1.1 Algebraic Loops

There might appear a problematic situation which is known as *algebraic loop.* That occurs when there is a feedback path through blocks that have direct feedthrough (the input has instantaneous effect on the output). For example, consider a state feedback control loop with an state-space block. Assume that $D \neq 0$ so that the block has direct feedthrough. The problem here is that to compute the output of the process, the output itself must be known (see Figure 3.7).

Frequently, the algebraic loops can be solved either by redefining the model or by adding dynamics such as unit step memories. Using the first method, the dynamics system of the example can be expressed as $\dot{x} = (A + BK)x$, with an output $y = (C + DK)x$, and implemented within a single state-space block. Using the second approach, another possible solution may be to store the last step value,

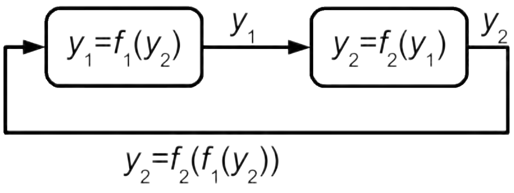


Figure 3.7: Simplified representation of an algebraic loop.

```java
// up contains the output of the controller at the previous step
double[] yp = process.getOutput(xp, up);
double[] yc = controller.getOutput(xc, yp);
double[] dxp = process.getRates(xp, uc);
double[] dxc = controller.getRates(xc, yc);
```

Listing 3.13: Avoiding an algebraic loop by storing the last step value.

as shown in Listing 3.13.

A discussion about algebraic loops including a precise mathematical definition can be found in [hds13].

3.4.2 The elements

The use of the library elements is similar to any other element of EJS: a block can be added to the simulation by dragging and dropping from the palette to the list of elements of the current simulation. Assuming that the control loop has been defined, and it can be constructed with the library built-in blocks, the steps needed to create a simulation with the library are the following ones:

1. Go to the *Process Control Library* located into the *Elements* page of the EJS model.

2. Add the needed blocks to the current simulation by dragging and dropping, and assign a name to each element. The name allows programmers to access the elements from the code.

3. Configure the blocks either with the configuration page of the element or by writing the approppriate code.

4. Define the interconnections between blocks.

5. Add the code to the *ODE*, *Evolution*, and/or *Event* pages to integrate with the solver.

6. Show the desired outputs in the user interface.

This steps are esentially the same in all cases, with independence of the complexity of the model.

3.4.3 The OSP numerics package

Previous sections discussed the use of the *Process Control Library* through the EJS elements that can be incorporated into EJS simulations. The EJS solver relies on the OSP numerics package library and makes it easier to use for the developer. But there are situations in which an advanced developer might find limitations in this method. Two common reasons are:

- *Simplicity.* EJS acts as an interface to the OSP framework that make it easier to use, so the developer delegates some actions to the tool. While this is good in many cases, if the model has many blocks or complex interactions, it could be simpler to take control of the solver and let EJS deal only with the view and the user interactions.
- *Performance.* The EJS solver has to compute the derivatives and the outputs of each block many times during the execution of the simulation at the same time that the view is updated at a fixed frame rate. If no interactivity is required, it is more efficient to simulate the model for the whole time interval. For example, one can compute the response to an input step change for a family of process, and then show all the results in a plot.

The OSP numerics package contains numerical analysis tools, such as different ODE solvers or a framework to deal with matrices, obtain eigenvalues, the inverse matrix, etc. The details of the OSP numerics package can be found in [Chr07]. The next paragraphs discuss the use of the OSP framework to integrate a model and present an example to illustrate the simulation of a continuous feedback control loop.

The actions needed to simulate the model are esentially the following ones:

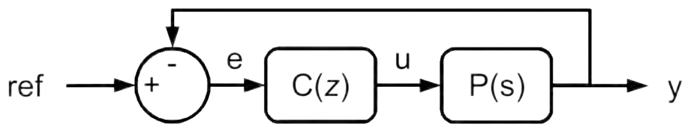


Figure 3.8: Basic output feedback control loop with a discrete controller $C(z)$ and a continuous SISO process $P(s)$.

- Define the ODE.
- Choose the ODE solver.
- Integrate the ODE.
- Store and/or present the results.

The definition of the ODE is done implementing the ODE interface. The integration of the *PCL* blocks with the OSP is similar to the EJS case, commented above.

The control loop is depicted in Figure 3.8. It is a general feedback loop, where the process is a SISO process with continuous dynamics and the controller is a discrete controller. The *FeedbackLoopODE* class contains the definition of the ODE corresponding to the control loop. It is built by combining two different blocks of the PCL: a *Continuous* block and a *Discrete* block. The particular implementation of the block is passed to the class constructor, thus allowing to instanciate the control loop to represent different kinds of systems. The *getRate* method is responsible for the computation of the derivatives, combining the different blocks (Listing 3.14) in the adequate order.

The next step is to choose the solver and integrate the ODE. In this example, the *FeedbackLoopODETest* class has been created to solve the particular case of a first-order process with a PID controller (see Listing 3.15).

```java
/**
 * Class to define the ODE corresponding to a generic feedback control loop
 * composed of a continuous process and controller.
 **/
public class FeedbackLoopODE implements ODE {
  private Continuous process, controller;
  private double[] state;
  private double setpoint, disturbance;

  public FeedbackLoopODE (Continuous process, Continuous controller) {
    this.process = process;
    this.controller = controller;
    state = new double[1+process.getNumberOfStates()+controller.getNumberOfStates()];
    setpoint = 0.0;
    disturbance = 0.0;
  }

  public void getRate(double[] state, double[] rate) {
    double[] xp = new double[process.getNumberOfStates()],
             up = new double[process.getNumberOfInputs()],
             xc = new double[controller.getNumberOfStates()],
             uc = new double[controller.getNumberOfInputs()];

    System.arraycopy(state, 1, xp, 0, process.getNumberOfStates());
    System.arraycopy(state, 1+process.getNumberOfStates(), xc, 0, controller.
     getNumberOfStates());

// Calculate outputs
    uc[0] = setpoint - process.getOutput(xp, up)[0];
    up[0] = controller.getOutput(xc, uc)[0] + disturbance;

// Calculate derivatives
    rate[0] = 1; // time
    double[] dxp = process.getRates(xp, up); // process
    double[] dxc = controller.getRates(xp, up); // controller
    System.arraycopy(dxp, 0, rate, 1, process.getNumberOfStates());
    System.arraycopy(dxc, 0, rate, 1+process.getNumberOfStates(), controller.
     getNumberOfStates());
  }

  public double[] getState() {
    return state;
  }
}
```

Listing 3.14: Control loop definition by code.

```java
/**
 * Class to solve the ODE defined by FeedbackLoopODE.
 **/
public class FeedbackLoopODETest {
  public static void main(String[] args) {
    StateSpaceModel plant = new StateSpaceModel(new double[][]{{-1}}, new double
     [][]{{1}},
                                                new double[][]{{1}}, new double
     [][]{{0}});
    Discrete pid = new DiscreteFilteredPidController(1, 0.1, 0, 1);
    double dt = 0.1;
    FeedbackLoopODE loop = new FeedbackLoopODE(process, controller);
    ODEBisectionEventSolver odeSolver = new ODEBisectionEventSolver(ode, RK45.class);
    odeSolver.initialize(dt);

    loop.setpoint(1);
    double time = 10;
    while(time > 0) {
      double[] states = loop.getState();
      System.out.println("time="+state[0]+",output="+state[1]+",input="+state[2]);
      time -= odeSolver().step();
    }
  }
}
```

Listing 3.15: Code to solve the ODE.

3.5 Application example: PID control of a tank

The first application example is the simulation of a single tank plant controlled by a PID controller which admits several configurations.

The process is composed of a tank with a constant section and an outlet hole with a valve through which the fluid goes out of the tank. The tank is filled through a pump which is assumed to be linear with respect to the control signal. In addition, there is a second pump which can extract water from the tank. Depending on the configuration of the system, i.e., whether the valve is opened or closed and where the control actuates (the first or the second pump), the resulting process can be modeled as an integrator or as a first-order model.

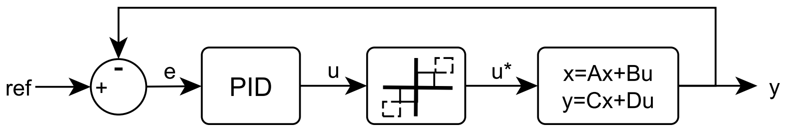
3.5.1 The control loops

The example application provides two control loops, shown in Figure 3.9, which are event-based schemes similar to that described in [CSVD12]. These schemes are composed of the process, which is the above described tank, a continuous PID controller, and a send-on-delta sampler. The send-on-delta sampler is a block that takes a new sample of the input when the sampled signal crosses certain levels, which have a width defined by a threshold δ, with an offset α with respect to the origin. This block can be placed in two positions: sampling the control variable, or sampling the process variable.

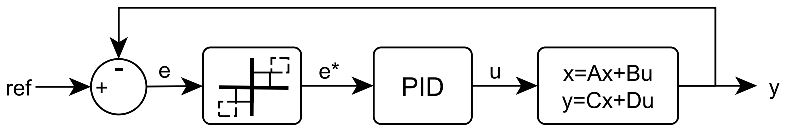
3.5.2 Building the simulation

The features expected from the simulation can be summarized in the following points:

1. Possibility of simulating the two described control loops, showing graphically the variables involved in the loop.



(a)



(b)

Figure 3.9: Two control loops with different location of the send-on-delta sampler, (a) the error signal at the input of the controller is sampled, and (b) the controller output is sampled.

2. Interactivity, i.e. providing the user with the possibility of control the execution of the simulation but also to change the configuration and/or parameters of the systems and reflect immediately those changes.

In the next paragraphs, the building process is described step by step.

3.5.2.1 Step 1: Adding the elements

Once the control loops have been designed, the first step towards the final implementation is to add the elements to the simulation (Figure 3.10). This must be done in order to have them available for the model. Each element can be instantiated one or more times, but each instance must have a unique identificator (a Java valid identificator), since it corresponds to a Java object. The elements with configuration options can be customized in this step by using the *dialog* provided by the element (usually this is the preferred way), or in the initialization code.

The names that have been associated to each instance are: *sum*, *controller*, *sampler*, *process* (in the same order as in Figure 3.9a).

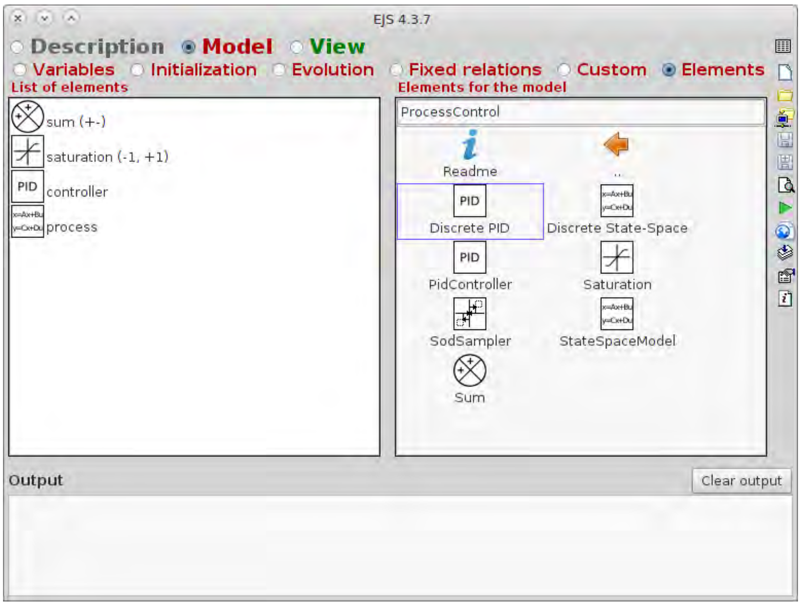


Figure 3.10: *Elements* page of the EJS Model. On the right, the elements of the *Process Control Library*. On the left, the instances of the elements added to the current simulation.

3.5.2.2 Step 2: Interconnecting the blocks

In the second step, after adding all the needed blocks, it is necessary to interconnect them in order to make each block able to obtain its inputs. From a high-level point of view, each block is a black box with inputs indexed from 0 to n_i, and outputs, from 0 to n_o. Since all lines are assumed to be of the same type (*double*), any output can be connected to any input.

The interconnection of the blocks depends on the loop configuration, i.e. whether the controller operates at the input pump or at the output pump. The current configuration is stored in the *boolean* variable `inputPumpControl`, which is defined inside the *Variables* page of the simulation. The corresponding code is listed in Listing 3.16.

This code is executed every integration step in the *Preliminary Code* page of the ODE solver in the EJS simulation, because the outputs must be updated prior to

```
// Compute the output of the tank and its sampler
yp = tank.getOutput(xp, null);
yps = processSampler.getOutput(xps, yp);
po = processSampling ? yps : yp;
uc = sum.getOutput(null, new double[]{setpoint, po[0]});
// Compute  the output of the controller and its sampler
yc = controller.getOutput(xc, uc);
ycs = controllerSampler.getOutput(xcs, yc);
co = controllerSampling ? ycs : yc;
// Select the adequate input
if(pumpInputControl) {
  up[0] = offset + co[0];
} else {
  up[0] = offset - co[0];
}
```

Listing 3.16: Excerpt of code that computes the output of the tank and of the sampler.

computing the state derivatives. If not, the blocks do not know how to obtain their correct inputs and thus the state cannot be properly updated.

At this step, the two control loops have been entirely defined, and thus the last step to complete the simulation is to add the code needed to integrate the model.

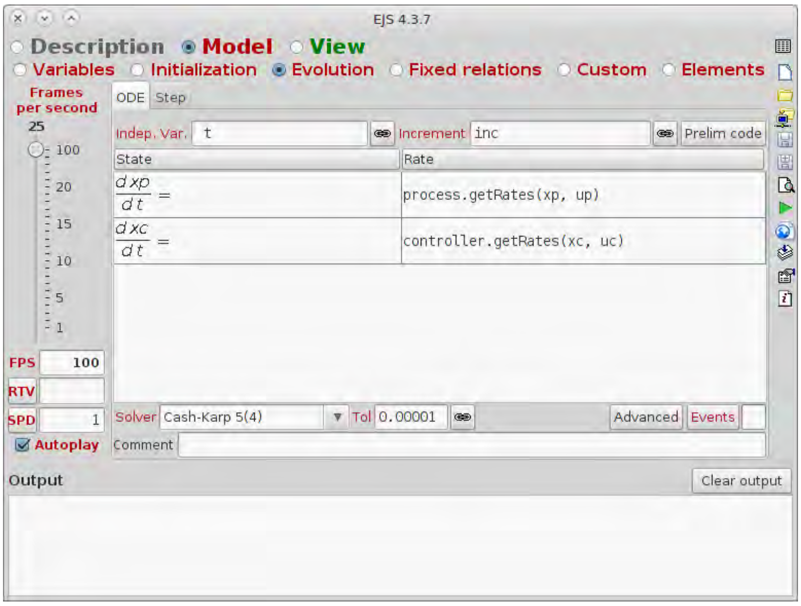
This can be done in several ways, depending on the kind of block. For each block with continuous flow, the dynamics can be added to an *ODE page* (see Figure 3.11). If the block is discrete or event-based, then the updates can be introduced into a *Evolution page* or an *Event page*.

There are two blocks in the loop that have continuous dynamics: the *controller* and the *process*. To integrate the state of this elements with the EJS solver, the derivatives must be added to an *ODE page*, which can be done with *dx/dt = process.getRates()* and with dx_c/dt = *controller.getRates()* (see Figure 3.11).

Note that the blocks with continuous state must link their state vector to an EJS variable, so that the solver can access them properly. This is done with the method *linkStates(double[] x)*, defined in the interface *Continuous*.

3.5.2.3 Step 3: Creation of the GUI

This step is identical to the building of other EJS simulations, and does not depend on the library, so it will not be discussed here. However, the interface has been designed trying to keep it simple and easy to use.



Description [✓ Model] View
Variables Initialization [✓ Evolution] Fixed relations Custom Elements

Frames per second: 25 (100, 20, 15, 10, 5, 1)
FPS: 100
RTV:
SPD: 1
[✓ Autoplay]

Indep. Var.: t
Increment: inc

State	Rate
$\frac{d\,xp}{dt}$ =	process.getRates(xp, up)
$\frac{d\,xc}{dt}$ =	controller.getRates(xc, uc)

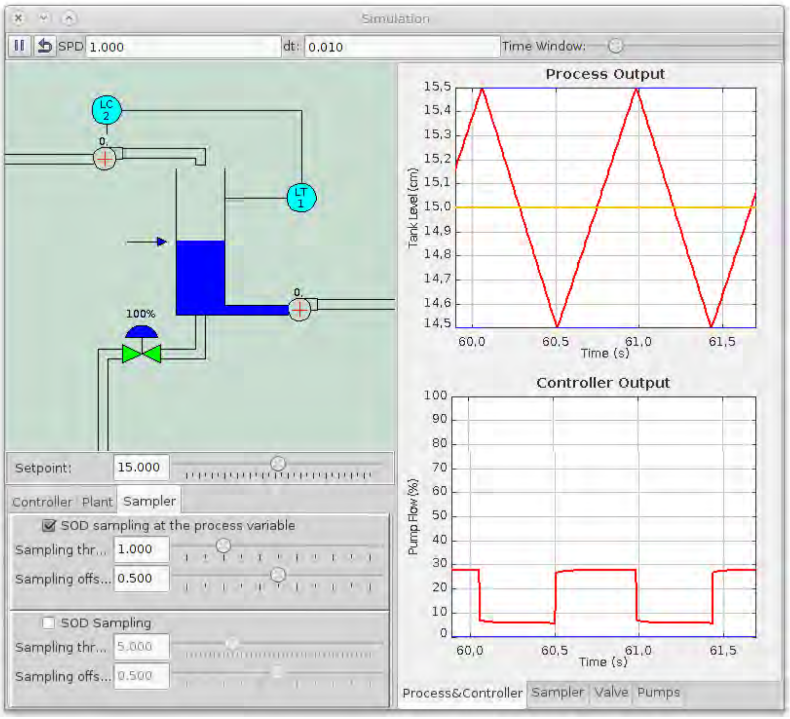
Solver: Cash-Karp 5(4)
Tol: 0.00001
Comment:

Figure 3.11: Definition of the model in EJS. The blocks with continuous dynamics need to have an associated state. The derivatives of the state are obtained by calling the method *getRates()* of the block in an ODE page.

The user interface is shown in Figure 3.12. The state of plant is shown with a visual animation of the tank which varies its water level according to the value of the state. The diagram follows the *PID* symbol convention, which is a standard in the process industry. At the right, the variables of interest (including the plant state) are shown in two plots. The panel under the view contains the controls that allow to modify the parameters of the experiment.

3.6 Application example: Event-based PID control loop

Following the same scheme as the previous example, another interactive simulation has been built with the *Process Control Library*. In this case, the control loop is the same, but instead of the tank the example allows to choose one from four predefined


SPD 1.000 dt: 0.010 Time Window:

Setpoint: 15.000

☑ SOD sampling at the process variable
Sampling thr... 1.000
Sampling offs... 0.500

☐ SOD Sampling
Sampling thr... 5.000
Sampling offs... 0.500

Figure 3.12: User interface of the *PID control of a Tank* example. On the left, the graphical representation of the plant with a *P&ID* based diagram and the controls to modify the parameters of the experiment. On the right the plots show the process and the controller output.

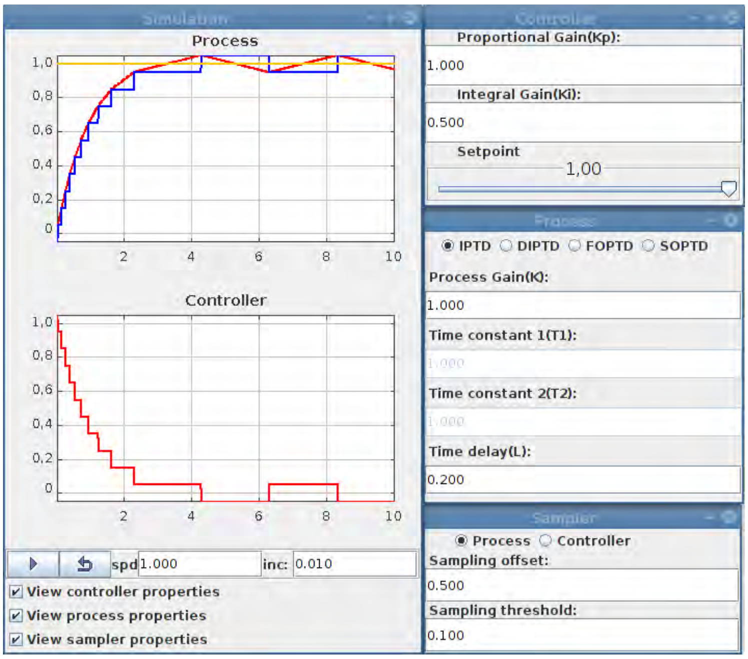
processes: an integrator, a double integrator, a first-order model, and a second-order model.

3.6.1 Building the simulation

Since the building process of this example has simmilarities with the previous one, only the differences are remarked.

3.6.1.1 Step 1: Adding the elements

Despite of the substitution of the *Tank* element by the *process*, this step is esentially the same as in the previous example.



Proportional Gain(Kp): 1.000

Integral Gain(Ki): 0.500

Setpoint 1,00

[● IPTD] ○ DIPTD ○ FOPTD ○ SOPTD

Process Gain(K): 1.000

Time constant 1(T1): 1.000

Time constant 2(T2): 1.000

Time delay(L): 0.200

[● Process] ○ Controller

Sampling offset: 0.500

Sampling threshold: 0.100

spd 1.000 inc: 0.010

[✓] View controller properties

[✓] View process properties

[✓] View sampler properties

Figure 3.13: User interface of the simulation example. The interface is composed of several windows: On the left, the main window shows the plots corresponding to the event-based sampling of the process and controller. On the right, the other windows allow users the configuration of the different blocks of the two control loops: the process, the controller, and the sampler.

3.6.1.2 Step 2: Interconnecting the blocks

This step is the same as in the previous example.

3.6.1.3 Step 3: Creation of the GUI

This step is identical to the building of other EJS simulations, and does not depend on the library, so it will not be discussed here. However, several aspects to be taken into account for designing the interface are the following ones,

1. *Simplicity.* The interface must be designed trying to keep it simple and easy to use.

2. *Multi-window.* The interface follows the multiwindow paradigm, that means

that conceptually different actions are done in separated windows. Thus, there is a window showing the plots, another window to configure the process, and so on.

The user interface is shown in Figure 3.13. The main window contains two plots which show the process and the controller outputs. Depending on the configuration, the sampler output is plotted either with the process output (when sampling the process variable) or with the controller output (if the control variable is sampled). From this window it is possible to control the execution of the simulation. In addition, there are three additional windows, one with the configuration of the PID controller, another with the configuration of the sampler, and the third one to configure the process.

3.7 Conclusions

A library of Java classes and EJS elements is now available to build simulations related with process control. The PCL aims to facilitate the development of this kind of simulations, inspired by widely used tools such as SIMULINK, and capturing the behaviour of the most common types of systems. In order to build a new simulation, it is not needed to start from the scratch, but by only choosing and interconnecting blocks the application development effort is greatly reduced.

Two examples of interactive simulation tools developed by using the library have also been presented, illustrating the use of the main components and the simplicity of the development. However, it must be remarked that these examples of simulation are actually fully functional interactive tools which allow to experiment with a family of event-based systems.

Though the use of library has been presented in a simulation context, it is also possible to include it in a hardware-in-the-loop application, either by using the

LabVIEW Connector Element for EJS or combining with the real-time support of EJS and the elements already available to access open hardware platforms, such as Arduino or Phidget. This application is discussed in the next Chapter.

4 Adding Event-based Capabilities to Remote Labs

4.1 Introduction

Designing and developing web-enabled remote laboratories for pedagogical purposes is not an easy task. Often, developers (generally, educators who know the subjects they teach but lack of the technical and programming skills required to build Internet-based educational applications) end up discarding the idea of exploring these new teaching and learning possibilities mainly due to the amount of technical issues that must be mastered. To tackle this problem, in this Thesis we present a novel technique to allow developers to create remote labs in a quick, didactical and straightforward way. This framework is based on the use of two well-known software tools in the scope of engineering education: Easy Java Simulations (EJS) and LabVIEW. This new technique exploits one of the new features of EJS, known as EJS *Model Elements*, that enables Java developers to create and integrate their own authoring libraries (elements) into EJS, thus increasing its application possibilities. Particularly, the EJS elemen presented in this chapter allows to control LabVIEW programs from EJS applications through a communication network. This chapter presents the element creation details and how this can be used to create web-enabled experimentation environments for educational purposes. A step by step example is described to illustrate the basic functionality, i.e. execution control of a VI and synchronization of data with the VI.

The integration of new pedagogical methodologies in engineering education is, nowadays, practically mandatory in most universities around the world. This statement is grounded by the number of papers published about these subjects and where the current technological advances have shown the way to follow in this field [GNR05, CPV03]. For instance, in the European case, this has been addressed by introducing the educative community to the European Space for Higher Education (Bologna process), in which Internet plays a key role in university studies [LMA10].

Regarding the aforementioned, hands-on laboratories were one of the first places where the integration of such technological advances was visible. Many engineering faculties expanded the use of these laboratories by offering students opportunities of experimentation with real systems (processes) not only by live classroom training but also remotely through the Internet. These Internet-based educational tools are currently known as *Web-based laboratories*. Web-based laboratories are divided into two categories, according to the system's nature to manipulate: virtual and remote. A *virtual laboratory* simulates a mathematical model of a physical process, whereas a *remote laboratory* provides access to a real physical process located in a remote site on the Internet [DDV+08].

Although simulation is an appropriate way of complementing engineering education, it generally cannot replace experimentation with real processes. For this reason, a full web-based laboratory should offer both training modalities. However, creating the remote version of a web-based lab is still attainable only for educators and research teams who are expert in these matters, mainly due to the amount of technical and programming issues that must be mastered [GB09].

In the literature, many different approaches oriented to the development of remote laboratories can be found, so the examples reviewed are mainly focused on architectures using MATLAB and/or LabVIEW. For instance, a SCADA developed in LabVIEW to control a remote PLC is presented in [PDF$^+$13]. In [SCMS11], authors present a remote laboratory exclusively created by using the LabVIEW platform [Lab13]. Although LabVIEW VIs [1] can be easily made ready for Internet delivery, a LabVIEW Runtime Engine must be installed at the client side. This last step is not recommended when creating remote labs since installing software plugins sometimes can become hard for final users. For this reason, LabVIEW platform is commonly used only for creating the server side of a remote lab. Other software options for the server side can be MATLAB [FDKDE10], Simulink [FFDC$^+$11], C++ [CVJ$^+$], Scicos [MZ12], etc. For example, in [CGGV11, CPV13] authors present a MATLAB-based platform to experiment with mobile robots: the *Automatic Control Telelab.*

On the other hand, Java applets and Flash applications have been the most popular web technologies for developing the client interface for remote labs. In [HMCC08], a virtual laboratory for the analysis and study of the human respiratory system was created. In this example, an applet was developed by using EJS [EJS12]. Two other examples of remote labs for pedagogical purposes were presented in [SMD$^+$02, SDPM04]. In these articles authors present a set of web-based laboratories for teaching automatic control concepts where Java applets to access remotely the training services were used as well. Similarly, Flash applications have found some applications in virtual and remote laboratories design [RJP$^+$03, Gof07]. Unlike Java, Flash has been less used by developers for designing web-based labs mainly for license payment issues.

Despite all these efforts, simple approaches to assist beginner's developers in creating remote labs are not easy to find. In this context, next lines describe the

[1]"LabVIEW programs are called virtual instruments, or VIs, because their appearance and operation imitate physical instruments, such as oscilloscopes and multimeters", *National Instruments* [Lab]

authors proposal in order to contribute in this scope.

At the Spanish National Distance Education University UNED (the authors' institution), distance education courses on automatic control for as many as 300 students each year are offered. Until few years ago, these students had to travel to Madrid from all over the country to attend two-week long laboratories to complete the prescribed hands-on experiments in system identification and control courses. Fortunately, the development of Internet technologies highlighted the importance of Web-based teaching and learning in many research fields, including automatic control. Since more than 10 years, we therefore decided to use Web-based labs in our instruction so that students could minimize their need to physically attend laboratories. The acquired experience for our research team during all this time can be summarized in the following selection of papers [VSJ+11, VSS+09, VSGD09, DVD+08, VSD+08].

Based on the experiences above described, we present a new approach to create remote labs. This framework, which is an update of the work presented in [VSS+09], uses the software tools EJS and LabVIEW. The development framework exploits the new feature of Easy Java Simulations known as *EJS-elements* that enables Java developers to create and integrate their own authoring libraries (elements) into EJS, thus increasing its application possibilities. Particularly, the EJS element here presented allows to LabVIEW programs be controlled from EJS applications through a communication network simply by linking LabVIEW and EJS variables by means of a configuration wizard. The approach hides the low-level communication issues always necessary when creating remote labs, thus simplifying its creation process.

4.2 The Remote Lab architecture

The basic layout of a remote lab is as follows: on the one hand, the plant, whose sensors and actuators allow to interact with it, is connected to the host PC via an

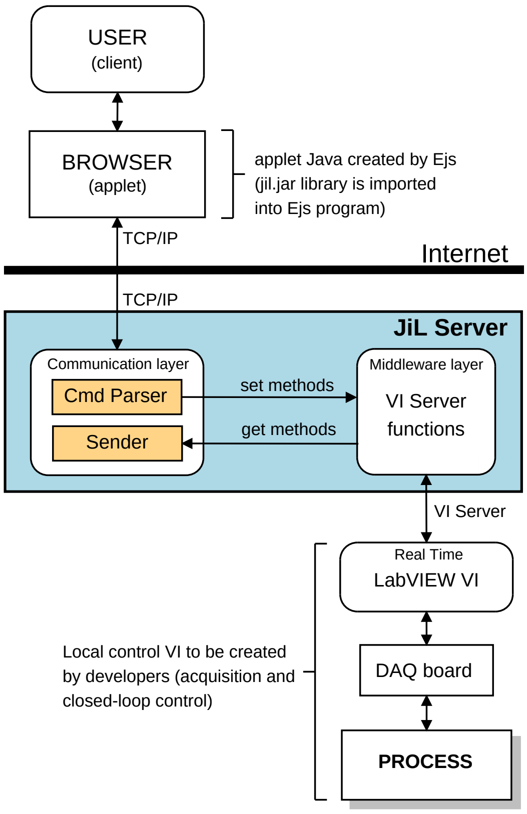


Figure 4.1: Architecture of a remote lab built with EJS, *JIL Server*, and LabVIEW.

acquisition card (DAQ). On the other hand, the graphical user interface (GUI) that allows students the interaction with the plant, and which is an application that runs in the student PC. This solution is usually known as *client/server* architecture.

An improvement to this approach, adopted in this work, is the use of a three-tier architecture (see Figure 4.1). In this solution, a middle-tier is introduced between the client and the server, acting as an intermediary that allows to eliminate or reduce the dependency of the design and implementation of both sides. Thus, the client can focus, for example, on the interface with the user, and the server on the control of the plant, while the middle-tier copes with the data exchanging issues.

Though there are different alternatives to set up the server and client PCs, what we are looking for is a solution that ideally could be applied to any case.

The *National Instrument LabVIEW platform* has been chosen to setup the server PC. As said before, it is a graphical programming tool, widely spread both in industry and academics, which allows for the rapid development of applications. One of the main advantages of LabVIEW is its great hardware support, providing *drivers* and libraries to access DAQ systems, communication protocols, etc.

The middle-tier contains the *JIL Server*, which makes public the controls and indicators of the real-time VI (see Figure 4.1) to be accessed from the client.

Finally, the client tier is the most visible part, because it provides the GUI. EJS is a good option to create the GUI because it has been designed to simplify the development of interactive simulations and graphical interfaces. Also, EJS provides us with a mechanism to extend its capabilities, the *elements* of the model, that let us to encapsulate Java libraries in a way that can be incorporated easily into a simulation.

Here is where the *LabVIEW Connector Element* and the *Audio Player Element* fit, hiding the low-level details of the communication with the LabVIEW VIs via the *JIL Server* and allowing the feedback of audio from the plant. The framework is explained in [Var10] and can be consulted for further details. In the rest of this chapter, we focus on the implementation of the event-based transmission mechanism from the point of view of the two sides: LabVIEW and EJS.

4.3 The LabVIEW VI

There are several implementation problems or decisions to take that frequently appear during the development phase, and must be implemented in the LabVIEW VI at the server side. These problems can be sistematically tackled by dividing the functionalities into several subsystems:

- *Data Acquisition.* It is in charge of the communication with the DAQ to read the level sensors and to send the control actions to the actuators.

- *Local Controller*. This subsystem are includes the implementation of the controller and all the components needed for switching between the different modes of the control system. Though the architecture allows the controller to be implemented in the client side, it is recommended to have a local controller to prevent unsafe operating conditions.

- *Event-based Communication*. As mentioned before, the *JIL Server* cope with the VI execution control and data exchanging. However, the implementation of the event-based sampling schemes and, in particular, the send-on-delta sampling must be done in the LabVIEW VI.

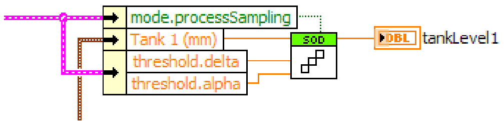
- *Data Logging*. This subsystem contains the blocks to log the data with the experiment evolution.

The *Local Controller*, *Data Acquisition*, and *Data Logging* subsystems are not discussed in detail. The first one because it depends on the plant to control, and the other two because the techniques are not specifically related to event-based systems. With respect to the *Event-based communication* subsystem, the implementation is based on the *SOD Sampler VI*, described in the next paragraph.

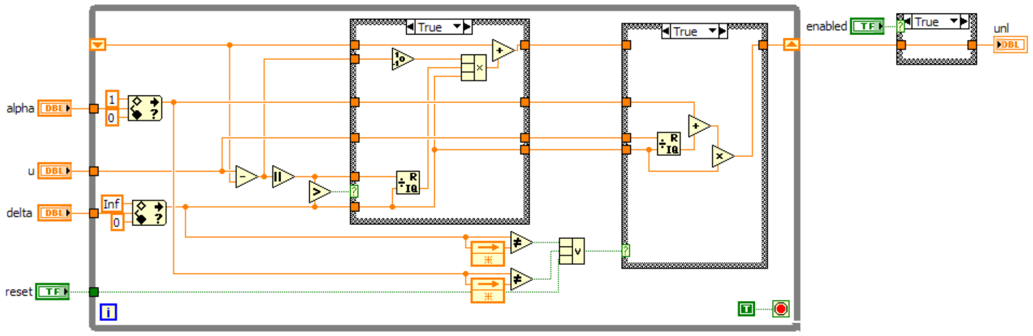
4.3.1 The SOD sampler VI

The SOD sampler, as defined in Section 2.2.1, has been implemented in LabVIEW as a reentrant VI that can be instanciated as many times as needed, depending on the signals being sampled.

Figure 4.2 shows the code of the VI. The implementation is based on a pattern described in [Blu07], which consists of the use of a *while loop* with an unitialized shift register that stores the last sample. The loop *stop condition* is connected to a boolean constant source with *true* value, thus every the subVI is called, there is



(a)



(b)

Figure 4.2: (a) Use of the *VI SOD sampler* to sample a variable, and (b) detail of the implementation of the send-on-delta sampler in LabVIEW.

exactly one execution of the loop. In this way, it can be embedded into a higher level loop or VI. The SOD sampler block admits as input two parameters, `alpha` and `delta`, the signal being sampled, `u`, and two control inputs, `enabled` and `reset`, to enable or disable the sampling and to reset the state, respectively.

4.4 Java and LabVIEW communication

The *JIL Server* software encapsulates the connection with LabVIEW by handling the client requests and providing them with a protocol that allows, on the one hand, to control the execution of a LabVIEW VI and, on the other hand, to read the *indicators* and update the *controls* of the VI.

This means that it does not enforce the use of a specific programming language at the client side. Therefore, though this section presents the implementation in Java, most of the ideas can be directly translated into other languages.

The software framework presented hereafter allows the communication of a Java applet or application with the *JIL Server* with two different levels of abstraction: a low-level protocol and a high-level protocol. Therefore, the framework is divided into:

- The *LabVIEW Connector*, that provides all the low-level functionalities, i.e. to open a connection with the *JIL Server*, to open, run, and/or close a remote *LabVIEW VI*, and to exchange *Java* variables with *VI controls* and *indicators*. This approach allows the user to have an entire control over the communications between the EJS simulation and the *JIL Server*, though it has as drawback that it is required to have a certain understanding of the internals of the *JIL Server*.
- The *LabVIEW Connector Element*, which provides a high level protocol to communicate with the *JIL Server* hiding the details to the user. It is a wrapper that allows the integration of a library into EJS.

The structure of the *LabVIEW Connector* library is represented in Figure 4.3. Note that, though the *element* requires EJS to run, the *LabVIEW Connector* core does not have these bindings, so it can be used by any Java application. In general, the high-level protocol is the recommended method because it is easier for the user. However it does not allow a direct control of the data exchange between the client and the server. Instead of that, all the variables are synchronized in group. Therefore, depending on the number of variables and the communication restrictions, the performance might not be optimal. Both protocols (and components) are explained in the next two sections.

4.4.1 Low-level communication protocol

In this context low-level communication protocol means that the primitives provided are close to the actual commands that the *JIL Server* can understand and process,

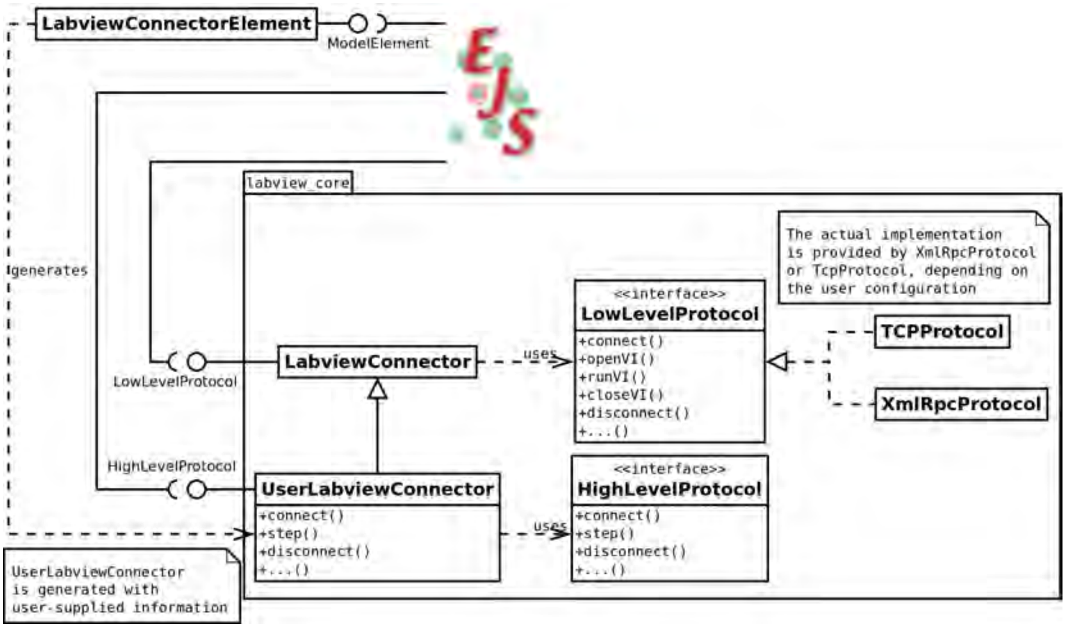


Figure 4.3: Class diagram representing the structure of the *LabVIEW Connector* library for Java and EJS. The low-level primitives are encapsulated into the class *LabviewConnector*. The class *LabviewElement* implements the interface to incorporate the library into EJS simulations.

and there is no user-specific code related to the particular configuration of a remote laboratory. The communications can be done through TCP connections and the *JIL Server* protocol described in [Var10] or as web services via the HTTP and XML-RPC [httb]. This latter protocol has been implemented in the last version of the *JIL Server* to ease the interface with all kind of platforms (there are libraries implementing the XML-RPC protocol in Java, C/C++, PHP, .NET, etc.)

All the functionality over which the communication is built is provided by generic methods, defined by the *LowLevelProtocol* interface (Listing 4.1). Therefore, a particular implementation of a protocol must provide these methods. Nevertheless, a direct use of the low-level protocol interface is in general not recommended because it is necessary to have some knowledge about the logics of the communications with the *JIL Server* and because it can be more error prone due to the necessity of writing

```java
/**
 * Interface to implement the low-level communication protocol
 */
public interface LowLevelProtocol {
  // Class Constructor
  public void LabviewConnector(String url);
  // Connection methods
  public void setServerAddress(String url);
  public boolean connect();
  public boolean disconnect();
  // Execution control methods
  public boolean openVI(String pathToVI);
  public boolean runVI();
  public boolean stopVI();
  public boolean closeVI();
  // Setter methods for the types boolean, int, float, double and String
  public boolean setValue(String name, boolean value) {...}
  public boolean setValue(String name, int value) {...}
  public boolean setValue(String name, float value) {...}
  public boolean setValue(String name, double value) {...}
  public boolean setValue(String name, String value) {...}
  // Getter methods for the types boolean, int, float, double and String
  public boolean getBoolean(String name) {...}
  public int getInt(String name) {...}
  public float getFloat(String name) {...}
  public double getDouble(String name) {...}
  public String getString(String name) {...}
}
```

Listing 4.1: Interface *LowLevelProtocol*.

a higher number of lines of code.

The use of the low-level protocol is summarized in the following steps:

1. Configure the *LabviewConnector* class to know the *url* of the *JIL Server*: method *setServerAddress(url)*.

2. Connect to the server: method *connect()*.

3. Open the remote *VI* specified by *path*: method *openVI(path)*.

4. Run the remote *VI*: method *runVI()*.

5. Repeat until *stop*:

   (a) Update the values of the *VI controls* with the *get*{*type*}*(name)* methods.

   (b) Read the values of the *VI indicators* with the *setValue(name, value)* methods.

6. Stop the remote *VI*: method *stopVI()*.

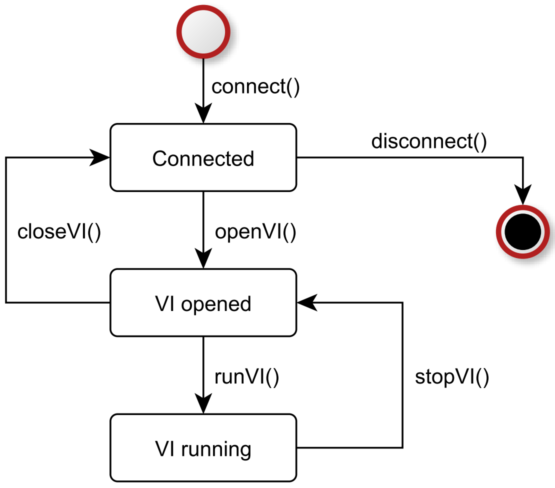


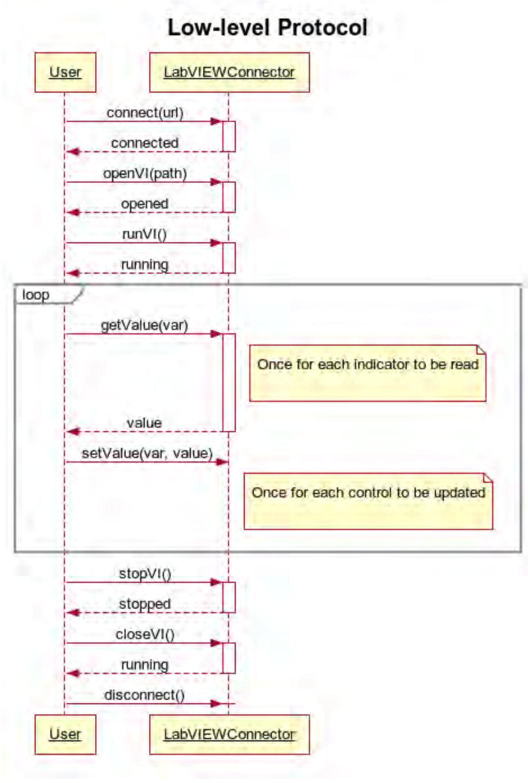
Figure 4.4: State diagram representing the possible states of the connection with the *JIL Server*.

7. Close the remote *VI*: method *closeVI()*.

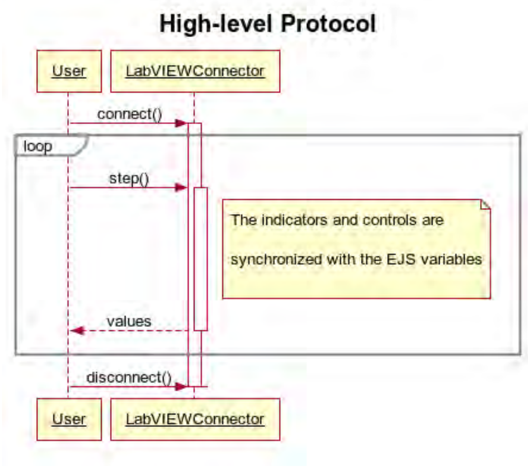
8. Disconnect from the server: method *disconnect()*.

Despite of the actually implemented protocol, its details are hidden through the described *LowLevelProtocol* interface. The other aspects of the connection with the *JIL Server* are encapsulated inside the *LabviewConnector* class, which provides an interface with the methods needed to set up the communication with the *JIL Server*. Furthermore, it implements a state machine that controls the state of the connection. The interaction with the server is summarized in Figure 4.4, which represents the possible states of a connection, and the sequence diagram of Figure 4.5a.

First, the *url* of the *JIL Server* should be provided either in the constructor or in the method *setServerAddress(String url)*. After that, the connection is done with the method *connect()*. The other methods to control the execution are *openVI(String pathToVI)*, used to open a LabVIEW VI that must be accessible in the server, *runVI()* which is used to initiate the execution of the previously opened VI, *stopVI()* to pause the execution of a running VI, and *closeVI()* to dismiss an opened VI. Fi-



(a)



(b)

Figure 4.5: Sequence diagrams for (a) low-level protocol, and (b) high-level protocol.

```java
/**
 * Connect to JIL Server.
 */
public synchronized boolean connect() {
  if (!connected){
    try {
      // create a TCP socket
      jilTCP = new java.net.Socket();
      jilTCP.setSoTimeout(4000);
      jilTCP.setTcpNoDelay(true);
      jilTCP.connect(new InetSocketAddress(SERVICE_IP, SERVICE_PORT), 4000);

      // create an input buffer to receive data
      BufferedInputStream bis = new BufferedInputStream(jilTCP.getInputStream());
      bufferInputTCP = new DataInputStream(bis);

      // create an output buffer to send data
      BufferedOutputStream bos = new BufferedInputStream(jilTCP.getInputStream());
      bufferOutputTCP = new DataOutputStream(bos);

      // create a sender object to manage the output buffer
      cbbs = new CircularByteBuffer(CircularByteBuffer.INFINITE_SIZE);
      sender = new Sender(cbbs, bufferOutputTCP);
      sender.setPriority(Thread.MIN_PRIORITY);
      sender.start();
      connected = true;
    }catch (IOException ioe) {
      System.err.println("connect() method message: IOException = "
                         + ioe.getMessage());
    }catch (Exception e) {
      System.err.println("connect() method message: Exception = "
                         + e.getMessage());
    }
  }

  return connected;
}
```

Listing 4.2: Excerpt of code from the *TcpProtocol* class.

nally, the method *disconnect()* closes the connection with the server and frees the resources. The methods that control the data communication between Java and the *JIL Server* are: *setValues(String name,...)*, which sends a new value to update a control in the VI, and *getDouble(String name),...,getString(String name)*, that obtain the value of an indicator of the *VI*. The method *setValues(...)* must be invoked with two parameters: a *String* containing the name of the control that must be updated, and the value itself. On the other hand, the methods *get{type}(...)* receive only one parameter, a *String* with the name of the VI indicator to be read, and return its value. The exact choice of the method will depend on the type of the indicator to obtain. Since the *JIL Server* currently admits two different implementations of the protocol, the first one over *TCP* and the second one over *XML-RPC*, there are also two classes implementing this interface: the *TcpProtocol* and the *XmlRpcProtocol* classes. Both classes are explored in the following paragraphs.

```
<!-- Client Request -->
POST /RPC2 HTTP/1.0
User-Agent: Frontier/5.1.2 (WinNT)
Host: betty.userland.com
Content-Type: text/xml
Content-length: 181
<?xml version="1.0" encoding="UTF-8"?>
<methodCall><methodName>jil.connect</methodName></methodCall>

<!-- Server Response -->
<?xml version="1.0" encoding="UTF-8"?>
<methodResponse>
  <params>
    <param><value><struct>
        <member>
          <name>version</name>
          <value><string>%JIL-XMLversion%</string></value>
        </member>
        <member>
          <name>sessionID</name>
          <value><i4>%SessionID%</i4></value>
        </member>
      </struct></value></param>
  </params>
</methodResponse>
```

Listing 4.3: Example of a XML-RPC request.

4.4.1.1 The TcpProtocol class

The implementation of the *TCP JIL Server protocol* is based on the *Labview.jar* library, described in [Var10]. An excerpt of the code of the *TcpProtocol* class is shown in Listing 4.2.

In particular, the code corresponds to the *connect()* method which creates a new connection with the *JIL Server*. This code is slightly more complex than the *Xml-RpcProtocol* implementation. First, the lines 6-10 try to create and configure a new connection socket with the specified *ip* and *port* where the server is listening. After the connection is established, the reception and sending buffers are initialized (lines 11-13) and, finally, the *sender* object is created, that is responsible to monitor the output buffer and periodically send the data to the server.

4.4.1.2 The XmlRpcProtocol class

As the *XML-RPC Specification* [htt07] states, XML-RPC is a remote procedure calling protocol that works over the Internet. An XML-RPC message is an HTTP-POST request with the body of the request in XML (see Listing 4.3).

A procedure executes at the server side and the value it returns is also formatted in XML. The *XML-RPC JIL Server protocol* defines methods that can be remotely invoked from the Java code, for instance *jil.connect()*, *jil.openVI()*, *jil.closeVI()*, or *jil.disconnect()*.

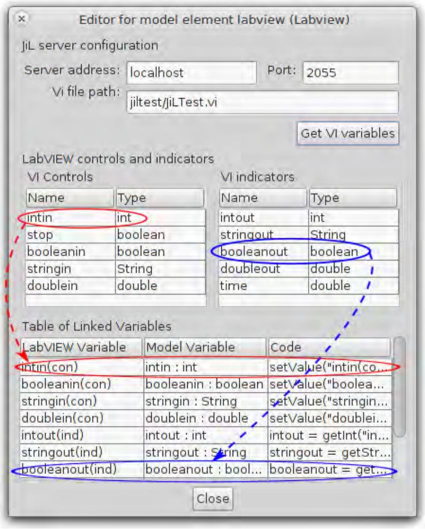
The implementation of the *XML-RPC JIL Server protocol* has been done by means of the Apache XML-RPC library, which generates the adequate XML-RPC request from the name of the methods and the parameters, and provides an interpreter to parse the server response. An example of the code to invoke the *connect()* method is shown in Listing 4.4. It is significantly simpler than the previous case. The remote method *connect()* is invoked by a call to the member method *execute* of the object *client* (line 8), which generates the XML-RPC request with the name and parameters (none in this case) of the remote method.

4.4.2 High-level communication protocol

Though the *LabviewSession* class and the *Protocol* interface are complete implementations that can be used to develop functional applications, it can require an effort to configure the connection and other practical details. Following with the paradigm of using the EJS elements to simplify the development, the idea is to have a high-level communication protocol that allows the user to be unaware of the internals of the

```
 1 /**
 2  * Open a connection with the server
 3  * @returns <i>true</i> if the connection with the server is done, <i>false</i>
        otherwise.
 4  */
 5 public synchronized boolean connect() {
 6   try {
 7     Object[] params = new Object[]{};
 8     HashMap result = (HashMap)client.execute(METHODNAME_CONNECT, params);
 9   } catch (XmlRpcException e) {
10     System.err.println("connect() method message: XmlRpcException = " + e.getMessage())
       ;
11     return false;
12   }
13   return true;
14 }
```

Listing 4.4: Excerpt of code from the *XmlRpcLabviewConnector* class



JiL server configuration

Server address: localhost Port: 2055
Vi file path: jiltest/JiLTest.vi

LabVIEW controls and indicators

VI Controls

Name	Type
intin	int
stop	boolean
booleanin	boolean
stringin	String
doublein	double

VI indicators

Name	Type
intout	int
stringout	String
booleanout	boolean
doubleout	double
time	double

Table of Linked Variables

LabVIEW Variable	Model Variable	Code
intin(con)	intin : int	setValue("intin(co...
booleanin(con)	booleanin : boolean	setValue("boolea...
stringin(con)	stringin : String	setValue("stringin...
doublein(con)	doublein : double	setValue("doublei...
intout(ind)	intout : int	intout = getInt("in...
stringout(ind)	stringout : String	stringout = getStr...
booleanout(ind)	booleanout : bool...	booleanout = get...

Figure 4.6: The configuration window of the *LabVIEW Connector Element* helps the user to configure the connection parameters, i.e. the server address, the path of the VI file, and the linkages between the EJS variables and the controls and indicators of the VI.

communication between the EJS simulation and the *JIL Server*.

This is done by means of the *UserLabviewConnector* class, which is automatically generated with the user-supplied information, i.e. the address of the *JIL Server*, the name of the remote VI and the links between the EJS variables and the LabVIEW VI controls and indicators. The configuration is done within the GUI provided by the element (see Figure 4.6). From a high level point of view, the functionalities that must be provided to the user to allow the interaction with a LabVIEW VI are summarized in the following points:

- Open and run a remote *VI*: *connect()*.
- Synchronize the EJS variables with the LabVIEW VI controls and indicators: *step()*.
- Stop and/or close the remote *VI*: *disconnect()*.

The *UserLabviewConnector* class contains the high-level methods that define a

```java
/**
 * Class to define the user-specific connection to the JIL Server.
 */
public class UserLabviewConnector extends LabviewConnector {
  /**
   * Connect to the server
   */
  public void connect() { }
  /**
   * Synchronize the variables
   */
  public void step() {
    if(isConnected && isRunning) {
      getValues();
      setValues();
    }
  }
  /**
   * Update the indicators
   */
  public void getValues() {
    var1 = getValue("var1");
    var2 = getValue("var2");
  };
  /**
   * Update the controls
   */
  public void setValues() {
    setValue("var1");
    setValue("var2");
  };
  /**
   * Disconnect from the server
   */
  public void disconnect() { }
}
```

Listing 4.5: Class *UserLabviewConnector*.

particular connection with a VI (Listing 4.5). Note that these functions are application specific. For example, each VI has its own *controls* and *indicators*.

Finally, the *LabviewConnectorElement* class provides two functionalities: the implementation of the *ModelElement* interface, which defines the contract required by EJS (Listing 4.6) to incorporate the library in the software tool as an *element*, and the code generator to automatize the definition of the *UserLabviewConnector* class.

4.4.2.1 The LabviewConnectorElement class

The EJS element is implemented by the class *LabviewConnectorElement*. The functionalities provided by this class are listed in the following points:

- Implement the interface *ModelElement*, allowing the class to be recognized as an element and loaded by EJS.

```java
public interface ModelElement {
  public javax.swing.ImageIcon getImageIcon();
  public String getGenericName();
  public String getConstructorName();
  public String getInitializationCode(String _name);
  public String getDestructionCode(String _name);
  public String getImportStatements();
  public String getResourcesRequired();
  public String getPackageList();
  public String getDisplayInfo();
  public String savetoXML();
  public void readfromXML(String _inputXML);
  public String getTooltip();
  public void clear();
  public void setFont(java.awt.Font font);
  public void showHelp(java.awt.Component parentComponent);
  public void showEditor(String _name, java.awt.Component parentComponent,
     ModelElementsCollection list);
  public void refreshEditor(String _name);
  public java.util.List<ModelElementSearch> search (String info, String searchString,
     int mode, String elementName, ModelElementsCollection collection);
}
```

Listing 4.6: *ModelElement* Interface.

- With the configuration provided by the user, it generates an specialized subclass of *LabviewConnector* which implements the high level protocol to communicate with the *JIL Server* and LabVIEW.

The configuration of the element requires only three steps:

1. Add the *LabVIEW Connector Element* to the current simulation by dragging and dropping to the *Model Elements page*.

2. Open the *LabVIEW Connector Element* properties dialog, and introduce the *url* of the server and the path of the VI to be loaded.

3. Link the LabVIEW controls and indicators of the VI with the variables of the EJS simulation.

Thus, the synchronization of the values of the linked EJS variables and LabVIEW controls and indicators is simply done with a call to the method *step()* in the *Evolution page*.

As mentioned before, it is encouraged to do the communication with the high-level protocol unless there is a good reason to use the low-level method.

Regardless of the chosen approach, in most of the applications, the variables can be grouped into two classes, namely,

- *synchronous*, which are the variables that correspond to controls and indicators that must be updated with a constant period, because they have a value that changes frequently. Examples of this kind of variables are the control inputs to the actuators or the readings from the sensors.

- *asynchronous*, which are variables that have the same value the most of the time, only changing sporadically, and they usually correspond to configuration parameters or user commands to interact with the plant.

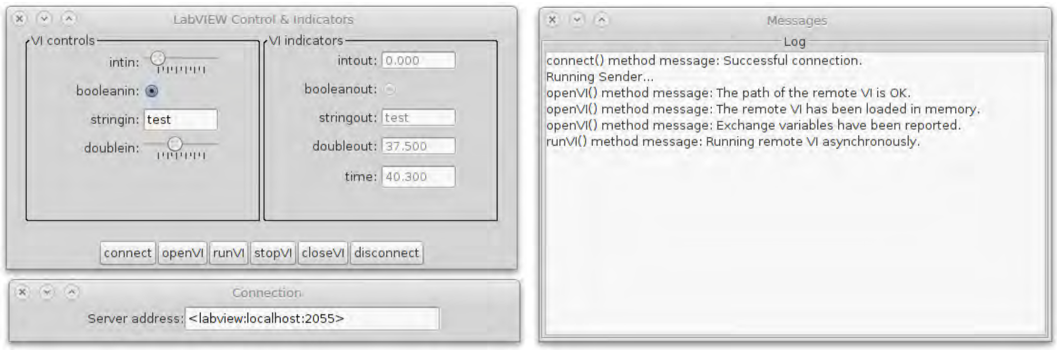
The configuration window of an element allows to mark every linked variable as synchronous or asynchronous variable, which in practice means that if a variable is marked as synchronous, it will be synchronized inside the *step()* method, and in the opposite case the user will be responsible for the variable synchronization.

4.5 Example: JIL Server Test

A simple example is presented to show the use of an element. Though simple, the example illustrates all the steps needed to construct an application which communicates with LabVIEW through the *JIL Server*. The LabVIEW VI of this example contains five controls and indicators, one for each primitive type, i.e. *boolean*, *int*, *float*, *double*, and *String*.

The name of each control and indicator is composed by a word denoting the data type and the *in* suffix for controls or the *out* suffix for indicators. For example, the *boolean* indicator is named `booleanin`, while the name of the control of type *String* is `stringout`.

The client application interface is shown in Figure 4.7. There is a set of EJS controls (*sliders*, *text fields*, and *check buttons*) that allow the user to modify the values of the associated VI controls, and a set of EJS indicators showing the state of the VI indicators.



```
intin:
booleanin:
stringin: test
doublein:

intout: 0.000
booleanout:
stringout: test
doubleout: 37.500
time: 40.300

Server address: <labview:localhost:2055>
```

```
connect() method message: Successful connection.
Running Sender...
openVI() method message: The path of the remote VI is OK.
openVI() method message: The remote VI has been loaded in memory.
openVI() method message: Exchange variables have been reported.
runVI() method message: Running remote VI asynchronously.
```

Figure 4.7: The user interface is divided into three different windows: one with the controllers and indicators, another with the server configuration and the third one with the status messages log.

4.6 Feedback from the plant

EJS incorporates a built-in capability to reproduce video from an IP cam, using either the MPEG video format or the multipart jpeg format (every frame is transmitted as a JPG image). This latter is usually the preferred method in the remote laboratories developed with EJS, mainly because it is provided by most of the video-capture devices, as IP cameras or webcams, and because it is simple to implement since it uses a mature compression standard (JPG).

The audio transmission has not been addressed in other remote labs developed with EJS. However, in certain plants the audio can contribute to create a feeling of proximity to the laboratory. As a particular example, an interesting experience for teaching electrical machinery concepts is to connect in series two rotating machines, the first acting as a motor and the second as a generator. Though the shaft rotation cannot be clearly perceived only with the video transmission, the sound can perfectly transmit the response of the plant to users stimuli.

However, as part of the architecture proposed in this chapter, the *Audio Player Element* that allows to reproduce sound from an IP cam has been developed.

The audio player is divided into two classes (see Figure 4.8), the *MultipartAu-*

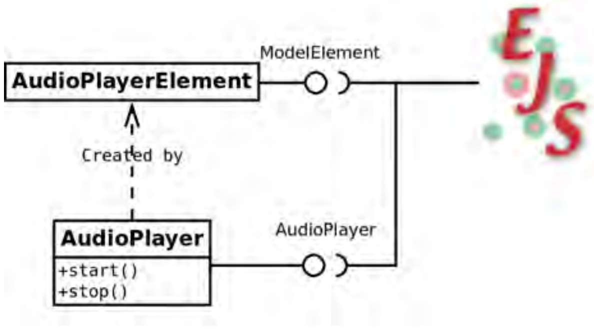


Figure 4.8: The class diagram shows the structure of the *Audio Player Element*. The implementation is divided into two classes: *AudioPlayer* provides the functionality and *AudioPlayerElement* allows for the integration with EJS.

dioPlayer class that actually implement the code needed to play audio, and the *AudioPlayerElement* class that allows to add it to an EJS simulation.

The *MultipartAudioPlayer* class, whose code is shown in Listing 4.7, provides two methods to control the reproduction of audio from the camera: *start()* and *stop()*. The other relevant methods are *setUrl(String url)*, to specify the URL of the audio source, and *setDoubleBufferLength(int len1, int len2)*, to control the size of the buffers.

The sound is transmitted over HTTP with MIME type *multipart/audio* [htta]. An independent thread is responsible for reading the audio frames into a buffer, which will be used to play a continuous audio stream. This thread is defined as an inner class of the *MultipartAudioPlayer*, the *PlayerThread* class, shown in Listing 4.8.

The body of the thread (i.e. the code of the method *run()*), is esentially a continuous loop that read a data stream opened on an specified *URL*. The audio data is then double buffered to provide a smooth audio playing of the stream.

The size of the buffer can be adjusted to accomodate to the quality of the connection, and to the format of the audio being transmitted, which currently can be codified in A-law or μ-law (PCM encoding).

```java
/**
 * MultipartAudioPlayer allows to play multipart audio streaming from an Axis IP cam.
 * The audio formats supported are: u-law and a-law (coming soon: and AAC).
 * @author Jesus Chacon Sombria <jchacon@bec.uned.es>
 */
public class MultipartAudioPlayer {
  /**
   * Create a new MultipartAudioPlayer object
   * @param url
   */
  public MultipartAudioPlayer(String url) {
    setDoubleBufferLength(FIRST_BUFFER_LEN, SECOND_BUFFER_LEN);
    setUrl(url);
  }

  /**
   * Set the size of the double buffer
   * @param first
   * @param second
   */
  public void setDoubleBufferLength(int first, int second) {
    src = new byte[first]; dst = new byte[second];
    audiostream = new ContinuousAudioDataStream(new AudioData(dst));
  }

  /**
   * Set the url source
   * @param url
   */
  public void setUrl(String url) {
    try {
      setUrl(new URL(url));
    } catch(MalformedURLException e) {
      e.printStackTrace();
    }
  }

  /** Start playing audio */
  public void start() {
    if(player == null) player = new PlayerThread();
    if(!player.isAlive()) player.start();
  }

  /** Stop playing audio */
  public void stop() {
    stop = true;
  }
}
```

Listing 4.7: *MultipartAudioPlayer* class.

```java
/**
 * PlayerThread
 */
private class PlayerThread extends Thread {
  public void run() {
    try {
        // Open connection
        URLConnection con = url.openConnection();
        ...
        while(!stop) {
          // Read the audio stream
          int bytesRead = in.read(src);
          ...
        }
        // Stop the player
        AudioPlayer.player.stop(audiostream);
    } catch (Exception e) {
      e.printStackTrace();
    }
  }
}
```

Listing 4.8: Summary of the code of the *PlayerThread* class.

The *AudioPlayerElement* class is simpler than the *LabviewConnectorElement* class, because in this case there is no need to provide user configuration, only the *URL* of the audio source. The configuration dialog provides only an input field for the *URL* of the streaming source. As an example, for the *Axis M1031-W* camera, the *URL* to have access to the audio is *http:/camera-ip/axis-cgi/audio/receive.cgi*.

4.7 Conclusions

The main contribution of this chapter is an architecture for rapid development of remote labs. The architecture is based on the use of LabVIEW, the *JIL Server*, and EJS, and allows educators who are not expert programmers to address the development of a remote lab with a minimized learning curve, due to the intuitivity of the graphical tools in the framework.

A significant effort has been dedicated to improve the ease of use, encapsulating all the low-level issues presented at the client side into the *EJS Model Element* mechanism. An *Element* is a wrapper that allows us to easily incorporate Java libraries into EJS simulations, providing with a graphical user interface to help the developer with the configuration and use of the library.

The *LabVIEW Connector Element* allows to configure a connection with a LabVIEW VI, to link EJS variables with the controls and indicators of the VI, and to control the execution of the VI. The *Audio Player Element* allows to play an audio streaming to provide the user with audio feedback from the plant. An important feature of the elements is that reduces the possibility of introducing errors in the code, thus reducing the time and effort needed for the development phase.

Part II

PART II: Assesment

5 Examples of Virtual and Remote Labs

To illustrate the use of the elements described before, in this chapter two examples of virtual and remote laboratories are presented. The first one is based on a Quadruple Tank plant, and the second one on a Flexible Link plant.

5.1 The Quadruple Tank Virtual and Remote Lab

5.1.1 The plant

The plant to be controlled is composed of two Coupled-Tank plants from Quanser, which are used simultaneously and coupled to obtain a more complex Multi-Input-Multi-Output (MIMO) experiment, the quadruple-tank process described in [Joh97, Joh00], depicted in Figure 5.1. It can be shown that the four-interconnected-tank system has an adjustable zero, which can be moved along the real axis in the left- or right-hand-side of the s-plane. Therefore by changing the system parameters, the multivariable zero dynamics can be configured to be either minimum phase or non-minimum phase.

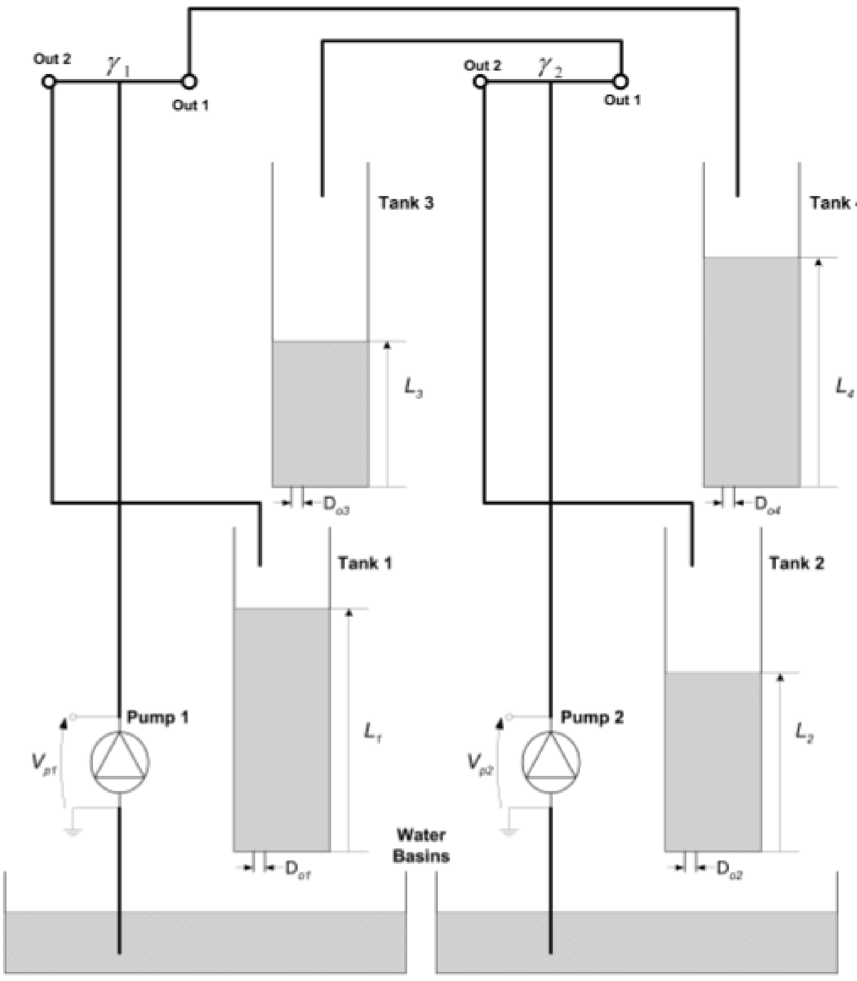


Figure 5.1: Diagram of the quadruple-tank system.

The overall Coupled-Tank frame is made of Plexiglas (see Figure 5.2). The two water tank system is made out of Plexiglas tubes of uniform cross section. The Coupled-Tank pump is a gear pump composed of a 12-Volt DC motor with heat radiating fins. The materials that come into contact with the fluid being pumped are: two molded Delrin gears in a Delrin pump body, stainless steel shafting, a Teflon diaphragm, and a Buna-N seal. It is also equipped with 3/16" ID hose fittings.

The liquid level of the tanks is measured through a pressure sensor. This sensor is located at the bottom of each tank and provides linear level readings over the

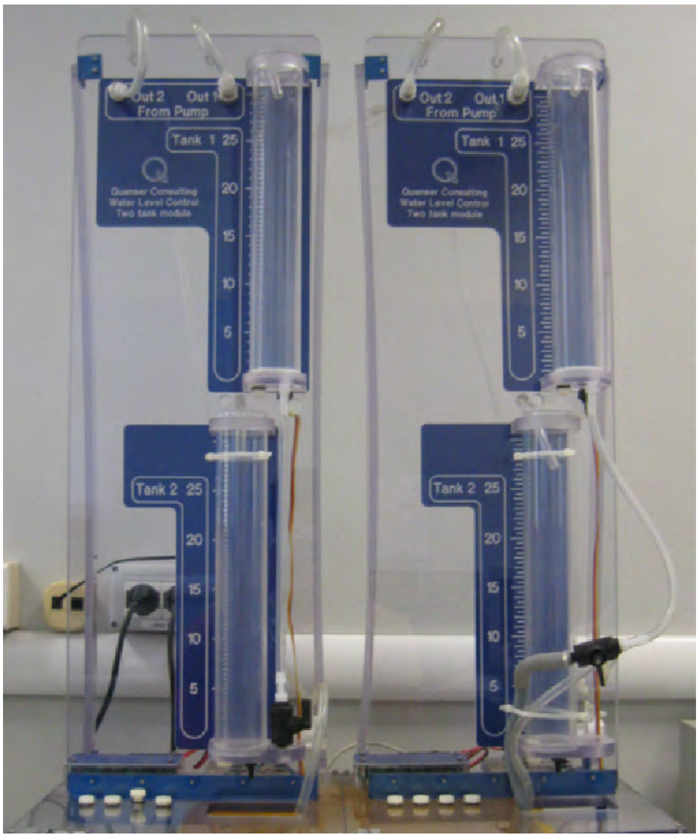
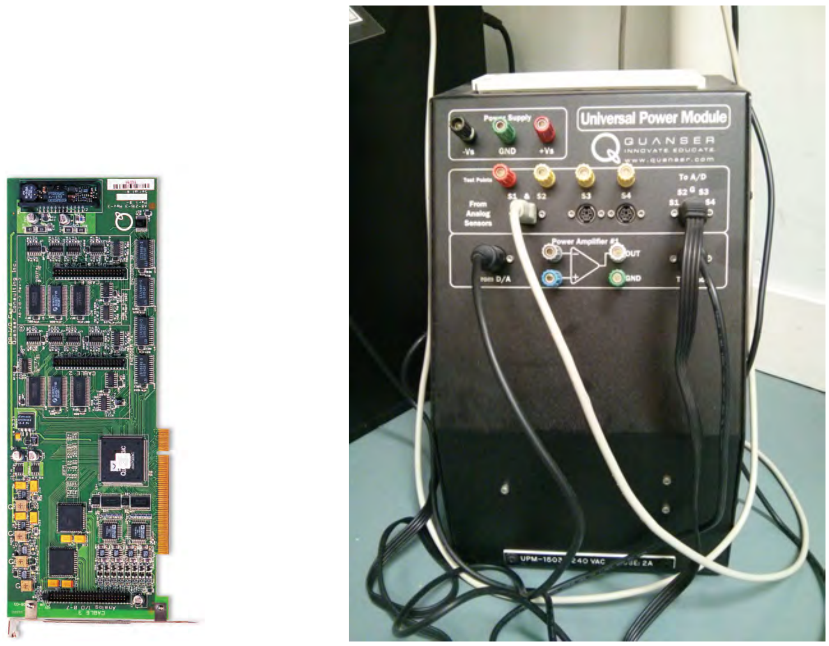


Figure 5.2: Quadruple Tank System from Quanser.

complete liquid vertical level. In other words, the sensor output voltage increases proportionally to the applied pressure. Its output measurement is processed through a signal conditioning board and made available as a 0 to 5V DC signal.

5.1.1.1 Data acquisition system

The data acquisition card is a Quanser Q8 control board (see Figure 5.3a). The Q8 is a HIL (Hardware-in-the-Loop) control board with an extensive range of input and output support. A wide variety of devices with analog and digital sensors as well as quadrature encoders are easily connected to the Q8. This single board solution is ideal for use in control systems and complex measurement applications. The



(a) (b)

Figure 5.3: The Quanser Q8 board (a), and the Universal Power Module UPM-2405 (b).

Quanser's Q8 board is supported by the most important real-time environments, as, for example,The MathWorks xPC Target, LabVIEW RTX and RT-LAB.

5.1.1.2 Power module

The physical connection between the DAQ and the plant (sensors and actuators) is done by means of an Universal Power Module UPM-2405 from Quanser (see Figure 5.3b), which is a power amplifier that provides the adequate voltages and currents to manage the pumps and to adapt the sensor signals to the board ranges.

5.1.1.3 Server PC

There is a PC which acts as a host connected to the system via the Q8 board. This computer runs the LabVIEW application that contains the control system implementation, and the *JIL Server* to add connectivity with the client side. Optionally, the user interface can be executed in this PC, but also in another PC with the Java Virtual Machine and with a network connection to the host PC.

5.1.1.4 The model

From the point of view of control theory, the quadruple tank plant is a MIMO system with two inputs (the pumps flow), and four outputs (the tanks water level). Though there are four outputs, we are more interested in the control of the levels of the two lower tanks. The reason is that if we choose as output the level of the upper tanks and choose the correct input-output pairing, we have two independent SISO system for which there are well-known techniques to control. On the other hand, if we choose the lower tanks levels as outputs, the system owns a richer dynamic behaviour.

A mathematical model for the plant can be derived from mass balances and Bernouilli's law.

Mass balance gives for each of the four tanks

$$\dot{V} = A \cdot \dot{h} = q_{in} - q_{out}, \tag{5.1}$$

where V is the volume of water in the tank, A is the cross-section area of the tank, h is the water level, q_{in} the inflow, and q_{out} the outflow.

By evaluating Bernouilli's law for incompressible liquids

$$\rho + \frac{1}{2}\rho v_w^2 + \rho g h = const. \tag{5.2}$$

at the water surface ($v_w = 0$) and at the bottom of each tank ($h = 0$) and substracting the resulting equations from each other, we obtain for the outflow

$$q_{out} = a \cdot v_w = a\sqrt{2g}\sqrt{h}, \tag{5.3}$$

where a is the cross-section area of an outlet, v_w is the speed of water at the outflow, and g is the acceleration due to the gravity.

The previous expressions applied to the system yield the following differential equations

$$\begin{aligned}
\frac{dh_1}{dt} &= -\frac{a_1}{A_1}\sqrt{2gh_1} + \frac{a_3}{A_1}\sqrt{2gh_3} + \frac{\gamma_1 k_1}{A_1}v_1 \\
\frac{dh_2}{dt} &= -\frac{a_2}{A_2}\sqrt{2gh_2} + \frac{a_4}{A_2}\sqrt{2gh_4} + \frac{\gamma_2 k_2}{A_2}v_2 \\
\frac{dh_3}{dt} &= -\frac{a_3}{A_3}\sqrt{2gh_3} + \frac{(1-\gamma_2)k_2}{A_3}v_2 \\
\frac{dh_4}{dt} &= -\frac{a_4}{A_4}\sqrt{2gh_4} + \frac{(1-\gamma_1)k_1}{A_4}v_1,
\end{aligned} \tag{5.4}$$

where A_i is the cross-section of tank i, a_i is the cross-section of the outlet, and h_i the water level. The voltage applied to the pump i is v_i and the corresponding flow $k_i v_i$. The parameters γ_1 and γ_2 depend on the configuration of the valves. The flow to tank 1 is $\gamma_1 k_1 v_1$ and the flow to tank 4 is $(1-\gamma_1)k_1 v_1$, and similarly for tanks 2 and 3. The acceleration of gravity is denoted by g.

5.1.1.5 Linearization of the model

The mathematical model can be linearized around an operating point. Defining the variables $x_i = h_i - h_i^0$ and $u_i = v_i - v_i^0$, where h_i^0 and v_i^0 are, respectively, the steady state tank level and input flow corresponding to the operating point, the linear system can be represented in state space as

$$\begin{aligned}
\frac{dx}{dt} &= \begin{pmatrix} -\frac{1}{T_1} & 0 & \frac{A_3}{A_1 T_3} & 0 \\ 0 & -\frac{1}{T_2} & 0 & \frac{A_4}{A_2 T_4} \\ 0 & 0 & -\frac{1}{T_3} & 0 \\ 0 & 0 & 0 & -\frac{1}{T_4} \end{pmatrix} x + \begin{pmatrix} -\frac{1}{T_1} & 0 \\ 0 & \frac{\gamma_2 k_2}{A_2} \\ 0 & \frac{(1-\gamma_2)k_2}{A_3} \\ \frac{(1-\gamma_1)k_1}{A_4} & 0 \end{pmatrix} u \\
y &= \begin{pmatrix} k_c & 0 & 0 & 0 \\ 0 & k_c & 0 & 0 \end{pmatrix} x.
\end{aligned} \tag{5.5}$$

The transfer functions matrix is

$$G(s) = \begin{pmatrix} \frac{\gamma_1 c_1}{1+sT_1} & \frac{(1-\gamma_2)c_1}{(1+sT_3)(1+sT_1)} \\ \frac{(1-\gamma_1)c_2}{(1+sT_4)(1+sT_2)} & \frac{\gamma_2 c_2}{1+sT_2} \end{pmatrix}, \tag{5.6}$$

where $c_i = \frac{T_i K_i K_c}{A_i}$ and $T_i = \frac{A_i}{a_i}\sqrt{\frac{2h_i^0}{g}}$.

5.1.1.6 Minimum and Non-minimum phase

One interesting property of the four tank system, from the academic point of view, is that the multivariable system can be of minimum or non-minimum phase depending on the configuration of the distribution valves (γ_1, γ_2). Thus the system can be used for a wide variety of experiments, for example to illustrate the student the difficulty to control a non-mimimum phase system.

As explained in [Joh00], the zeros of the transfer matrix are the zeros of the numerator polynomial of the rational function

$$detG(s) = \frac{c_1 c_2}{\gamma_1 \gamma_2 \prod_{i=1}^{4}(1+sT_i)} \times \left[(1+sT_3)(1+sT_4) - \frac{(1-\gamma_1)(1-\gamma_2)}{\gamma_1\gamma_2}\right]. \tag{5.7}$$

This means that the matrix G has two finite zeros for $\gamma_1, \gamma_2 \in [0,1]$. One of them is always in the left half plane, but the location of the second can be either in the left or in the right half-plane. In particular, it can be showed that the system is minimum phase for

$$1 < \gamma_1 + \gamma_2 \leq 2, \tag{5.8}$$

and non-minimum phase for

$$0 \leq \gamma_1 + \gamma_2 \leq 1. \tag{5.9}$$

There exists a straightforward physical interpretation. If the system is minimum phase ($\gamma_1 + \gamma_2 > 1$), then the flow to the lower tanks is greater than the sum of the flows to the upper tanks, and the system is easier to be controlled. However, if the system is non-minimum phase ($\gamma_1 + \gamma_2 \leq 1$), the flow to the lower tanks is smaller than the sum of the flows to the upper tanks, and in this case the control is more difficult.

5.1.1.7 Relative Gain Array

We can calculate the RGA (Relative Gain Array) from the linearized model (5.6), obtaning the following expression

$$\lambda_{11} = \frac{\gamma_1 c_1 \gamma_2 c_2}{\gamma_1 c_1 \gamma_2 c_2 - (1-\gamma_2)c_1(1-\gamma_1)c_2} = \frac{\gamma_1\gamma_2}{\gamma_1+\gamma_2-1}. \tag{5.10}$$

The first term of the RGA matrix completely determines the others, since the sum of all the elements of a row of the RGA is equal to one, and the same for the columns, so $\lambda_{11} = \lambda_{22}$, and $\lambda_{12} = \lambda_{21} = 1 - \lambda_{11}$. This means that for $\gamma_1 = \gamma_2 = 1$ we have a completely decoupled process, or partially decoupled if only one of (γ_1, γ_2) is equal to one. For the intermediate cases, the RGA can take positive values greater than 1 or negative values. This latter case is the worst in terms of interaction between variables.

5.1.2 The Remote Lab

The platform has been developed with the software tools Easy Java Simulations (EJS) [CE07, EJS12], *JIL Server* [Var10], and LabVIEW, that are combined to allow the interaction with the plant over the network. The controller is entirely in the client side, thus the event-based schemes are adequate because they allow the reduction of the data transmission, thus using more efficiently the network resources.

The remote lab is based on the three-tier architecture presented in Section 4.2 (see Figure 5.4). In the server side, there is a PC connected to the plant through a Data Acquisition Card (DAQ). This PC runs a LabVIEW Virtual Instrument (VI) which implements monitoring funtions and acts as an interface with the plant, i.e.

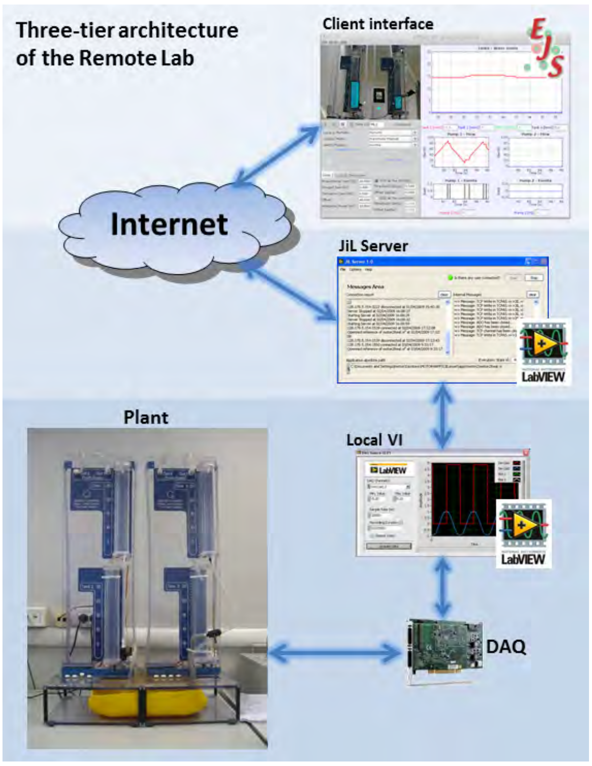


Figure 5.4: The three-tier architecture of the Remote Lab. The *server* is the LabVIEW Virtual Instrument (VI) running in the PC connected to the plant, the *client* is the student interface in *Easy Java Simulations*, and the *middle-tier* is the *JIL Server*, which acts as an interface between the *client* and the *server*.

it allows to obtain the readings from the sensors and sends the control actions to the pumps. Also, there is a webcam to transmit a real-time video and audio streaming of the plant, to allow students to feel more like if they were in a real lab, even if they are remotely connected. The middle-layer is the *JIL Server*, which publishes the variables (controls and indicators) of the VI to make them available over a network connection. Further, the third layer is the EJS application in the client side, which is not only the graphical interface to configure the control system and/or monitor the plant, but it also contains the controller implementation itself.

With regard to the communications, from an abstract point of view each node is composed of two components: a signal-generator and an event-generator. For example, for a sensor node the signal generator can be a zero-order hold that builds the signal from the periodic sensor readings, and the event-generator is the sampling scheme that decides whether to send the data to another node. Note that since the event generator can also be configured to emulate a periodic sampling, this approach is also valid to represent a discrete control system.

From the point of view of the control system, the two control loops depicted in Figure 3.9 are considered. In the first configuration, the sampler is placed at the output of the controller, and in the second one it is situated after the process output.

The student interface has been implemented in EJS based on the use of *Elements*, which allow to facilitate the building of the lab and to assure its reliability.

In addition to the *LabVIEW Connector Element*, which encapsulates the connection with the *JIL Server*, the *Process Control Library* has been used to implement the control system.

To simplify the development of the EJS interface of the virtual and/or the remote labs, it has be divided into five generic steps that can be applied to all the cases,

1. *Adding the elements* that will be used in the code.

2. *Setting up the connection* (only for the remote lab) with the server.

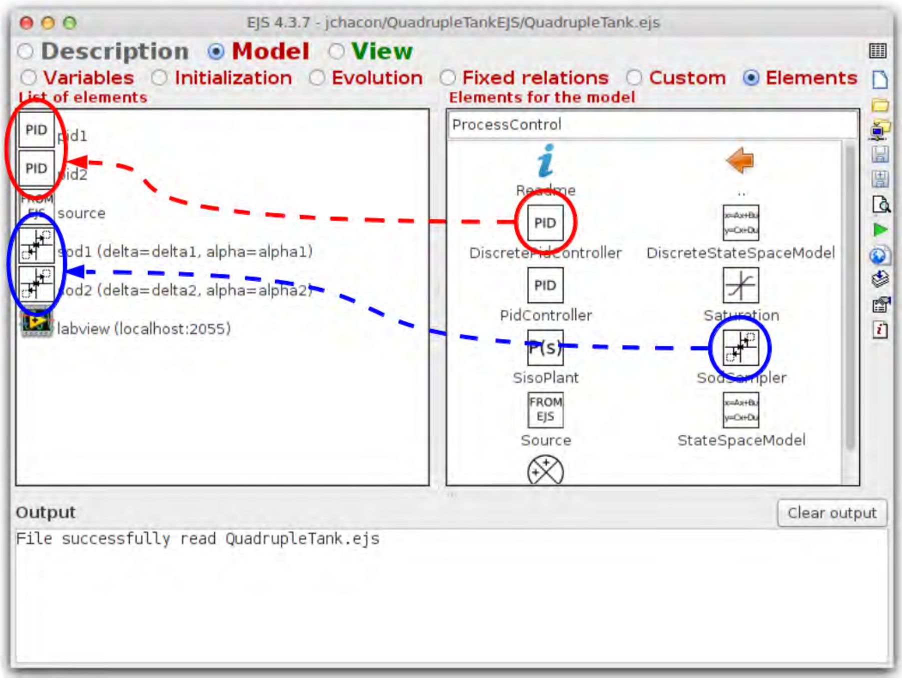
3. *Initialization code* to configure all the elements in a valid initial state.

4. *Evolution code* to integrate the model.

5. *User interface* design.

5.1.2.1 Step 1: Adding the Elements

Once the control loops have been designed, the first step towards the final implementation is to add the *Elements* to the simulation (Figure 5.5). This must be done in order to have them available for the model. Each element can be instantiated one or more times, if it is needed to connect with different servers, but usually only



```
File successfully read QuadrupleTank.ejs
```

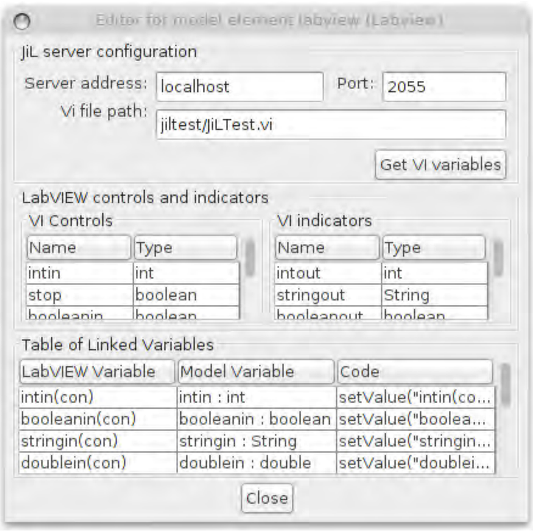
Figure 5.5: At the right of the *Model Elements Page*, the elements available in the libraries appear as an icon with the name of the element. The list at the left of the window shows the instance of the elements incorporated into the simulation, with their instance names and description strings. New instances of the elements are created by dragging and dropping the element icons.

one instance is necessary. The name assigned to the element in this step is used to access the element in the code.

5.1.2.2 Step 2: Setting up the Connection

The *LabVIEW Connector Element* must be configured prior to use it in the code. The basic configuration required is:

- The *url* where the *JIL Server* can be located.
- The *path* of the *VI*.
- The *variables* that will be exchanged with the server.



Editor for model element labview (Labview)

JiL server configuration

Server address: localhost Port: 2055

Vi file path: jiltest/JiLTest.vi

Get VI variables

LabVIEW controls and indicators

VI Controls

Name	Type
intin	int
stop	boolean
booleanin	boolean

VI indicators

Name	Type
intout	int
stringout	String
booleanout	boolean

Table of Linked Variables

LabVIEW Variable	Model Variable	Code
intin(con)	intin : int	setValue("intin(co...
booleanin(con)	booleanin : boolean	setValue("boolea...
stringin(con)	stringin : String	setValue("stringin...
doublein(con)	doublein : double	setValue("doublei...

Close

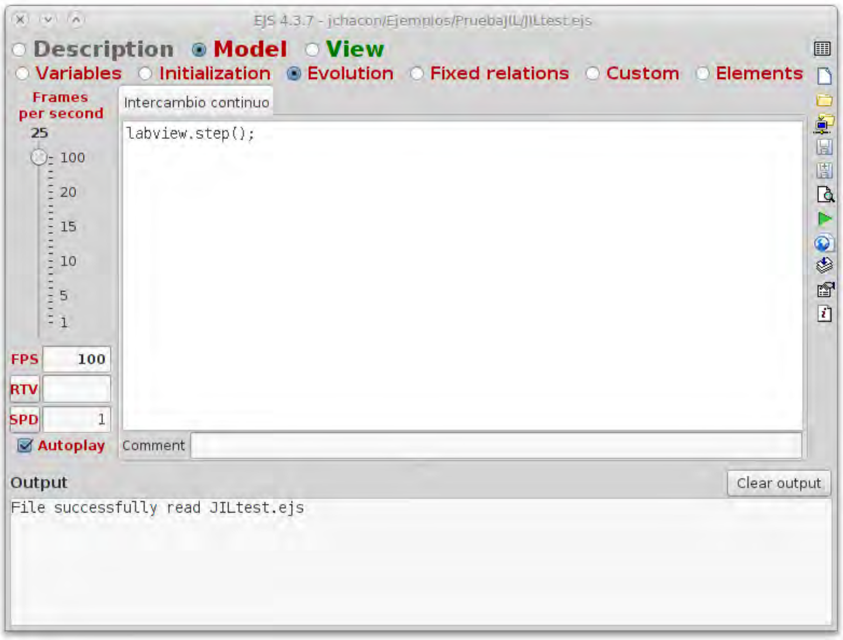
Figure 5.6: The configuration window of the *LabVIEW Connector Element* helps the user to configure the connection parameters, i.e. the server address, the path of the VI file, and the linkages between the EJS variables and the controls and indicators of the VI.

This can be done with the configuration dialog provided by the element (Figure 5.6). The *JIL Server* address and the VI path are introduced as text strings in the text field provided to this end. Then the user should click the button *Get VI variables* to load the controls and indicators of the VIs. Once loaded, the *VI controls* and *indicators* are listed in two tables, where the user can create the links to the *EJS model variables*.

As mentioned before, with the configuration data provided by the user, the *LabVIEW Connector Element* generates a class implementing the high level protocol.

5.1.2.3 Step 3: Initialization Code

The *labview.connect()* method must be invoked to open the connection with the server. Usually this invocation is done either in an *Initialization*, to start the connection automatically, or triggered by a button of the user interface.


EJS 4.3.7 - jchacon/Ejemplos/PruebaJIL/JILtest.ejs
Description [✓ Model] View
Variables Initialization [✓ Evolution] Fixed relations Custom Elements
Frames per second
25
100
20
15
10
5
1
FPS 100
RTV
SPD 1
Autoplay
Intercambio continuo
Comment
Output
Clear output

```
labview.step();
```

```
File successfully read JILtest.ejs
```

Figure 5.7: Synchronizing EJS and LabVIEW with the *labview.step()* method.

5.1.2.4 Step 4: Evolution Code

At this step, the communication with the server is usually done periodically to obtain the new values from the sensors readings, and to send the updates in the control actions or other parameters. These two things can be done with a call to the method *labview.step()* (see Listing 5.1 and Figure 5.7) in an *Evolution Page*.

Note that this approach can be rather unefficient as the number of exchanged variables increases. Frequently, the values of the VI controls correspond to config-

```java
public class LabviewConnector {
  ...
  public boolean step() {
    if (isConnected() && isRunning()) {
      setValues();
      getValues();
      return true;
    }
    return false;
  }
}
```

Listing 5.1: Code of the *step* method of the *LabviewConnector* class.

uration parameters that only changes due to the user interaction. Thus, it can be a better option to invoke only the method *labview.getValues()* periodically, and to call asynchronously the method *labview.setValues()* when needed.

5.1.2.5 Step 5: User Interface

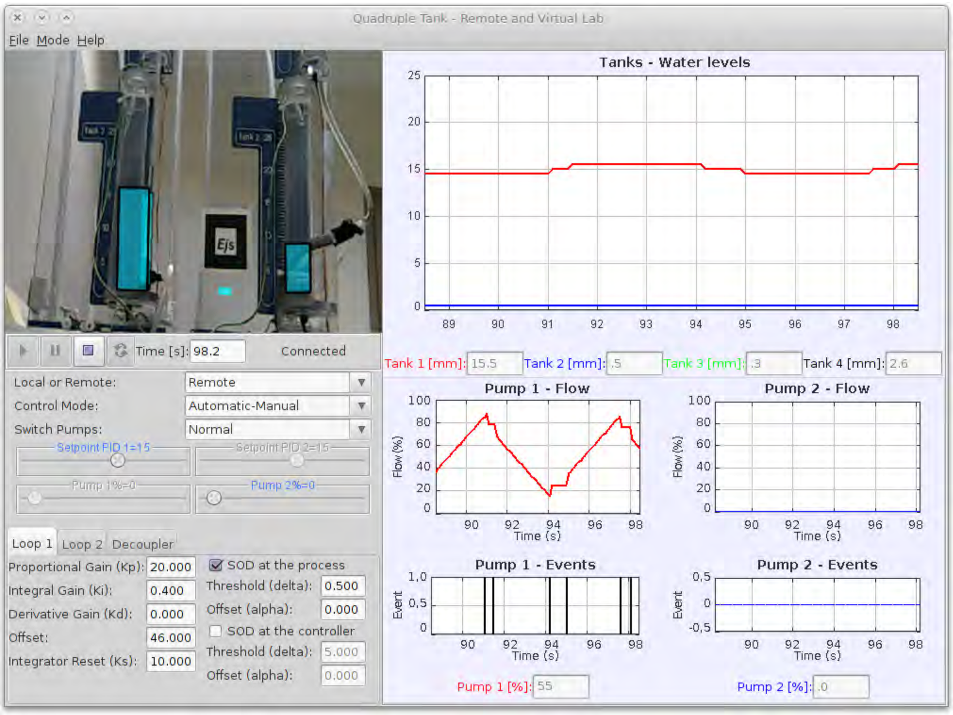
The user interface has been designed considering the information about the state of the plant:

- water levels,
- flow of the pumps,
- events in the controller,

and, also the configuration of the control system,

- connection control,
- automatic (controller) or manual (user) mode,
- PID parameters (gains and thresholds).

Figure 5.8 shows a screenshot of the interface. At the upper left part of the window there is an interactive graphical representation of the four tank system, with the double purpose of presenting to the user the state of the plant and to change the setpoint of the water levels for the bottom tanks. The right half of the window is entirely dedicated to show the evolution of the plant and the controllers, by means of several plots. The plot on the top shows the water level of the four tanks, together with the two set points. Below that, the plots are divided into the left half, with information about the first controller, and the right half corresponding to the second controller. For each controller there is one plot showing the control action and the event times (when the control is updated). Finally, at the bottom



Time [s]: 98.2 Connected

Tank 1 [mm]: 15.5 Tank 2 [mm]: .5 Tank 3 [mm]: .3 Tank 4 [mm]: 2.6

Local or Remote: Remote

Control Mode: Automatic-Manual

Switch Pumps: Normal

Setpoint PID 1=15 Setpoint PID 2=15

Pump 1%=0 Pump 2%=0

Proportional Gain (Kp): 20.000

Integral Gain (Ki): 0.400

Derivative Gain (Kd): 0.000

Offset: 46.000

Integrator Reset (Ks): 10.000

[✓] SOD at the process

Threshold (delta): 0.500

Offset (alpha): 0.000

[ ] SOD at the controller

Threshold (delta): 5.000

Offset (alpha): 0.000

Pump 1 [%]: 55 Pump 2 [%]: .0

Figure 5.8: The interface of the remote lab has been implemented in EJS. The state of the plant is shown by means of the plots at the right, and the image obtained from the webcam with augmented reality at the top-left part of the window. At the bottom-left, the student can configure the control system.

left there are controls which allows the user to configure the controller (gains, event thresholds, decoupling, etc.), and also to perform the conection with the server.

5.1.2.6 Main loop

The main loop of the EJS application, which is continuosly in execution, is defined within an *Evolution* page, and it is divided into three tasks, namely,

- Receive the state of the plant and other data from the server.
- Send the configuration commands to the server.
- To write this data to disk.

```
if(((jil.Jil)vi).isConnected()) {
  PeventPID1 = false;
  IeventPID1 = false;
  PeventPID2 = false;
  IeventPID2 = false;

// Get the state of the plant
  buffer = ((jil.Jil)vi).getDataAvailable();
  if(((jil.Jil)vi).isRunning()) {
    if (buffer > 0) {
      getValues();
    }
  }

// Send the control action
  setValues();
  decString = "";

// log the state
  if(lastTime < time) {
    lastTime = time;
    exp_logstate();
  }
}
```

Listing 5.2: Synchronization code.

The code included in the *Evolution* page is listed in Listing 5.2,

5.1.2.7 Control modes

The system admits four control modes, namely,

- *Pump 1 Manual - Pump 2 Manual.* Both pumps are directly controlled from the EJS side.
- *Pump 1 Manual - Pump 2 Automatic.* The control signal is directly sent to the pump 1 from the EJS side, but the pump 2 is controlled by the PI.
- *Pump 1 Automatic - Pump 2 Manual.* The opposite to the previous case.
- *Pump 1 Automatic - Pump 2 Automatic.* The two PI controllers are active, and the EJS side only acts as a monitor.

These modes allows the user to have more flexibility in the interaction with the plant. For example, the most common case corresponds to the automatic mode for the two tanks, where one can do several experiments with the built-in PI controllers. But thinking in a more demanding user, the manual mode can be used to obtain total control of the signal sent to the actuator. In this way the user can perform a wide range of experiments over the plant, such as identification of the tanks or even the substitution of the controllers with another control law, which in addition can be tested over the network.

5.1.2.8 Switching the pumps

In addition to the modes explained before, the application provides the possibility to switch the pumps, i.e., to send the control signal of the first controller to the second pump and viceversa. In the normal mode the Tank 1 is controlled by the Pump 1 and the Tank 2 by the Pump 2, and in the switched mode the pairing is Pump 1 - Tank 2 and Pump 2 - Tank 1. It must be noted at this point that the effect of switching the pumps can be also obtained by using a decoupling matrix or, from another point of view, when the decoupling matrix is being used the pumps-switch configuration also affects to the system.

5.1.2.9 Decoupling

The system admits three modes of decoupling, namely,

- *Not decoupled.* The control signal is forwarded directly to the pumps. It also can be viewed as if the decoupling matrix is set to the identity.
- *Direct decoupling.* The system is decoupled by using a direct scheme, defined by the transfer functions of the elements D_{ij}.
- *Inverse decoupling.* An inverse scheme defined by the transfer functions of the elements D_{ij} is used to decouple the system.

Due to the fact that the *JIL Server* can not work directly with complex types such as clusters of arrays, it is needed to find another way of sending the decoupling matrix. The method used in our application is to convert the decoupling matrix into an XML string which follows the specification of the LabVIEW XML type definition.

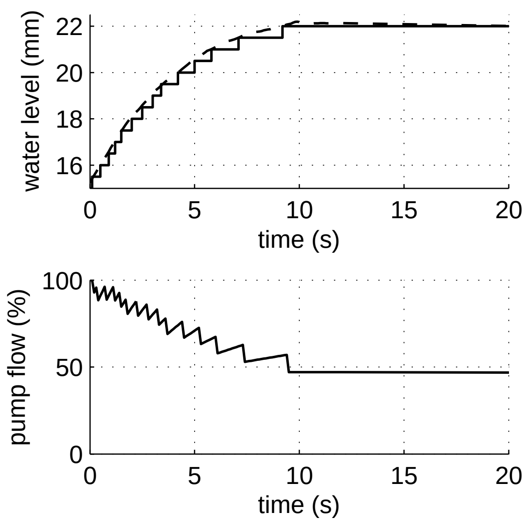


Figure 5.9: Step response of the process controlled by a PI with $k_p = 20$ and $k_i = 0.1$, and with the sampler at the process variable with $\delta = 0.5$ and $\alpha = 0.5$. The top plot shows the process output measured at the process (solid line) and the send-on-delta sampled signal received by the controller. The bottom plot shows the output of the controller.

Then the string is sent via *JIL Server* and unflattened in LabVIEW to rebuild the data as a cluster.

5.1.3 Results

To identify the system as a first-order model, it has been assumed that that operating point is of $15cm$ for the water level, which corresponds to an input of around 50% of the maximum pump flow. A batch of step tests were introduced as input, thus obtaining a set of experimental data which was divided into two subsets, one of them used for identification and the other one for validation. Figure 5.9 shows the response of the system to a step input and the response of the linearized model obtained in the identification. The transfer function identified is,

$$P(s) = \frac{0.63}{26.6s + 1}. \tag{5.11}$$

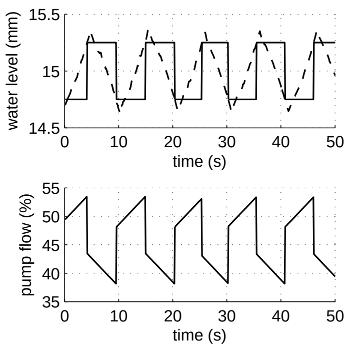
	T_m(s)	T_p(s)	A_m(cm)	A_p(cm)
$k_i = 0.2, \alpha = 0.5$	10.2	9.96	0.6	0.5
$k_i = 0.4, \alpha = 0.0$	9.2	9.02	1.5	1.43
$k_i = 0.8, \alpha = 0.5$	8.9	8.34	1.6	1.51
$k_i = 0.2, \alpha = 0.5$	15.3	15.02	0.44	0.52
$k_i = 1.0, \alpha = 0.5$	8.8	9.4	0.6	0.58
$k_i = 2.0., \alpha = 0.0$	8.2	8.5	0.7	0.64

Table 5.1: T_m, A_m are the measured periods and amplitudes, and T_p, A_p are the computed with the algorithm.

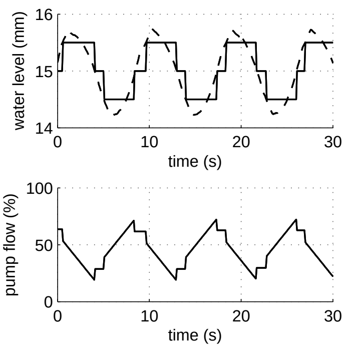
The model was identified with a minimum squares method. The percentage of the process output variation explained by the model is around 90%. An important practical issue is how to choose δ. As mentioned before, in theory the value of δ does not affect to the limit cycle properties. On the other hand, a value excessively small of δ can provoke unwanted events due to the signal noise, while a high value of the threshold can make the control system irresponsive. A rule of thumb is to choose a value to have around 10 samples in a step change. Since the step considered in the experiments ranges from 3 to 7 cm, choosing $\delta = 0.5$ seems reasonable. When the sampler is placed at the controller output, a value of $\delta = 5\%$ has been selected.

The first case considered is the PI controller with SOD sampler at the process output. The sampler parameters are $\alpha = 0$ and $\delta = 0.5$. The PI parameters have been tuned as $k_p = 20$, $k_i = 0.1$. Since these values are inside the stability region, the system does not present limit cycles. This can be seen in Figure 5.9, which shows the response of the system to a step change in the set-point.

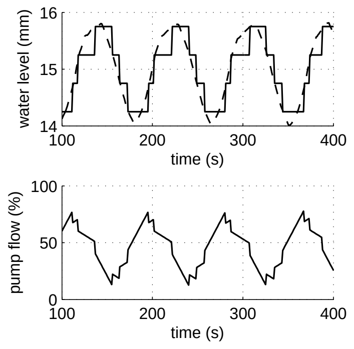
Increasing the integral gain progressively, it can be observed how the trajectories of the system tend to a limit cycle with different number of levels. It is worth to note that, depending on the value of α, the number of level varies. In particular, if $\alpha = 0$ the number of levels that are crossed by the sampled variable in the limit cycle must be odd, and if $\alpha = 0.5$ it must be even. Figure 5.10 shows this behaviour. As an example, for the particular case $k_i = 0.4$, the system presents a limit cycle with two



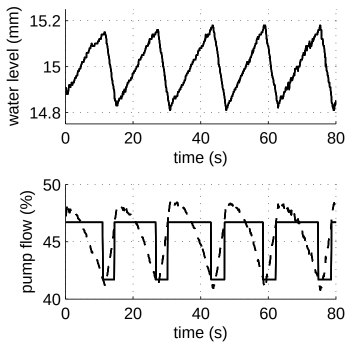
(a) $k_p = 20$, $k_i = 0.2$, $\delta = 0.5$, $\alpha = 0.5$



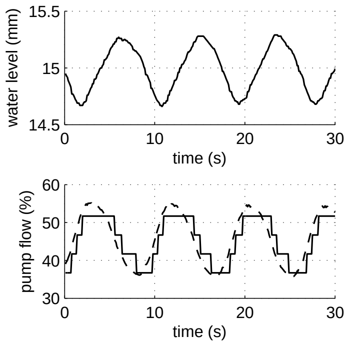
(b) $k_p = 20$, $k_i = 0.4$, $\delta = 0.5$, $\alpha = 0$



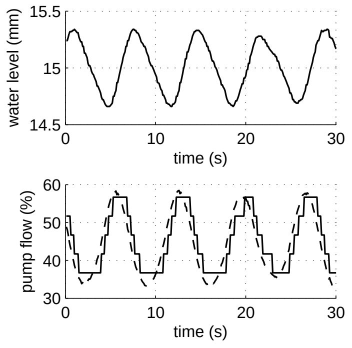
(c) $k_p = 20$, $k_i = 0.8$, $\delta = 0.5$, $\alpha = 0.5$



(d) $k_p = 20$, $k_i = 0.2$, $\delta = 0.5$, $\alpha = 0.5$



(e) $k_p = 20$, $k_i = 1.0$, $\delta = 0.5$, $\alpha = 0.5$



(f) $k_p = 20$, $k_i = 2.0$, $\delta = 0.5$, $\alpha = 0$

Figure 5.10: Step response of the process controlled by a PI with the event-based sampler at the process variable (top plots) and at the controller variable (bottom plots). The plots show the process and the controller output measured locally at their corresponding node (dashed line) and the send-on-delta sampled signal (solid line).

levels. Solving the equations of a limit cycle with two levels, the solutions yields a period of $T \approx 9.96$, while the real limit cycle has a measured period of $T \approx 10.2s$. This means that the error in the prediction is around 3%. The comparison of the measured periods and amplitudes with the values computed with the algorithm is shown in Table 5.1, for all the different considered cases.

5.2 Flexible Link Virtual and Remote Lab

In industrial robotics, it is common to assume that the robot structure is built with solid links, i.e. the links do not have deformation during the robot operation, so the study of motion is simplified. However, this is an ideal assumption and may fail for instance because of high payload-to-weight ratio, motion speed or control bandwith. Also, there are applications that need the design of very long and slender arms or require the use of lightweight materials. Flexible structures in motion are present in different domains, such as space manipulators, underwater and underground waste sites, and automated cranes. The structures use in aerospace and other mechanical application fields require a weight reduction to improve system performance. Because of the low damping and high flexibility, the fatigue and instability issues are greater than in rigid bodies. From the control point of view, the neglected link flexibility limits the achievable control performance, because of the vibrations, the steady-state error, etc. In addition, there is the problem of non-colocation between input commands and typical outputs to be controlled.

A survey of the experiments frequently carried out with flexible structures is provided in [SJJ92]. Two major groups are vibration suppression [SPA03, MRR13, SRR13], and slewing/vibration suppression control experiments [LO99, TY93, KS07], but also important are experiments involving the testing of new actuation and sensing concepts, improved sensors and actuators, etc.

This section presents a virtual and a remote laboratory with a Flexible Link

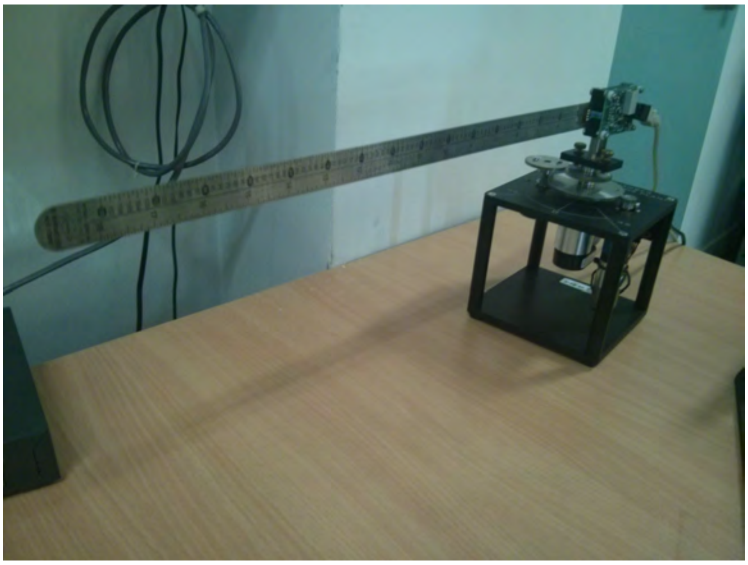


Figure 5.11: Flexible Link plant.

plant, which can be used to illustrate some of the problems that arises in control of flexible structures.

5.2.1 Description of the plant

The plant to be controlled consists of a flexible robotic arm manufactured by Quanser (see Figure 5.11). It is composed by several components, namely, a flexible link (FLEXGAGE) which is oriented in a horizontal position, and coupled to it a servo-motor (SRV02-ET) to apply a rotational movement to the arm.

The rotor position of the servo can be obtained from an encoder coupled to the motor. The encoder is able to distinguish between 4096 angular positions, i.e. it owns a resolution of 0,0015 rad, approximately. The other sensor is a strain gauge situated in the base of the link, which measures the angle of deflection of the arm.

The Flexible Link plant is connected to a PC host with a DAQ card to perform the data acquisition and to run the control and monitoring software. Finally, a power module (UPM 2405) acts as interface between the DAQ and the plant, with the double function of sending commands to the servo and adapting the sensor signals to the range admitted by the card I/O ports. The DAQ system and the power module are the same as in the previous example, so they are not described again.

5.2.1.1 The model

Depending on the requirements, obtaining a simulation model of a Flexible Link might be a non-trivial task. The main reason is because the actual system dynamics is an infinite-dimensional system described by partial differential equations. Frequently, the model is simplified approximating by a system with a finite number of degrees of freedom.

With that approach, a simplified system model can be obtained from the Lagrange-Euler formulation, if it is assumed that the deflection angle of the arm α is small and the displacement of the extreme point of the arm D can be calculated by the geometrical relationship $\alpha = \frac{D}{L}$, where L is the length of the arm. The lagrangian of the system, defined as the difference between kinetic and potential energy is,

$$L = T - V = \frac{1}{2}J_{eq}\dot{\theta}^2 + \frac{1}{2}J_{link}(\dot{\theta} + \dot{\alpha})^2 - \frac{1}{2}J_{eq}\alpha^2, \tag{5.12}$$

where L is the lagrangian, T is the kinetic energy, V the potential energy, and J_{eq} and J_{link} the rotational inertias of the base and of the link.

The system is determined by two generalized coordinates, θ and α, and therefore two equations can be stated,

$$\frac{\partial}{\partial t}\left(\frac{\partial L}{\partial \dot{\theta}}\right) - \frac{\partial L}{\partial \theta} = T_o - B_{eq}\dot{\theta} \tag{5.13}$$

$$\frac{\partial}{\partial t}\left(\frac{\partial L}{\partial \dot{\alpha}}\right) - \frac{\partial L}{\partial \alpha} = 0, \tag{5.14}$$

where T_o is the output torque and B_{eq} is the coefficient of viscous friction.

The output torque of the load from the motor is,

$$T_o = \frac{\eta_m \eta_g K_t K_g (V_m - K_g K_m \dot{\theta})}{R_m}, \tag{5.15}$$

where η_m, η_g, K_t, K_g, K_m, R_m, and k_w are electrical and mechanical parameters which depend on the motor characteristics, and V_m is the voltage applied to the motor. Solving the equations (5.13) and (5.14), and combining with (5.15), the simplified state space model is described by the following matrices,

$$\begin{bmatrix} \dot{\theta} \\ \dot{\alpha} \\ \ddot{\theta} \\ \ddot{\alpha} \end{bmatrix} = \begin{bmatrix} 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & \frac{K_{stiff}}{J_{eq}} & -\frac{\eta_m \eta_g K_t K_m K_g^2 + B_{eq} R_m}{J_{eq} R_m} & 0 \\ 0 & \frac{-K_{stiff}(J_{eq}+J_{arm})}{J_{eq} J_{arm}} & \frac{\eta_m \eta_g K_t K_m K_g^2 + B_{eq} R_m}{J_{eq} R_m} & 0 \end{bmatrix} \begin{bmatrix} \theta \\ \alpha \\ \dot{\theta} \\ \dot{\alpha} \end{bmatrix} + \begin{bmatrix} 0 \\ 0 \\ \frac{\eta_m \eta_g K_t K_g}{J_{eq} R_m} \\ -\frac{\eta_m \eta_g K_t K_g}{J_{eq} R_m} \end{bmatrix} V_m. \tag{5.16}$$

Substituting the parameters with the nominal values given by the vendor for each component, the resulting model is,

$$\begin{bmatrix} \dot{\theta} \\ \dot{\alpha} \\ \ddot{\theta} \\ \ddot{\alpha} \end{bmatrix} = \begin{bmatrix} 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 592 & -32 & 0 \\ 0 & -947.3 & 32 & 0 \end{bmatrix} \begin{bmatrix} \theta \\ \alpha \\ \dot{\theta} \\ \dot{\alpha} \end{bmatrix} + \begin{bmatrix} 0 \\ 0 \\ 56.2 \\ -56.2 \end{bmatrix} V_m. \tag{5.17}$$

However, it must be remarked that though the simplified model described in the previous paragraph can be used as a rough approximation of the plant behaviour, it does not reproduce some important characteristics of the system, such as the natural oscillation frequencies of the arm, which can be problematic to perform the control.

5.2.2 The controllers

Three different controllers are implemented to carry out experiments with the plant, namely, a PID controller, a state feedback controller, and a fuzzy logic controller.

5.2.2.1 The PID controller

The implemented PID controller structure is similar to the described in the previous example. It is implemented with the *Process Control Library*, and it can work as a continuous PID controller or as an event-based one.

5.2.2.2 The state feedback controller

The state feedback controller has been implemented with the *Process Control Library.* The control law is defined as $u(t) = Kx(t)$, where K is the state feedback gain. The controller is initially tuned by calculating the gains to optimize the LQR problem for the model. In the experiments, the gains can be adjusted within a range of $0.5K$ to $1.25K$ around the optimal values.

5.2.2.3 The fuzzy controller

The other controller available is the fuzzy controller, which has been implemented with the help of the *jFuzzyLogic* library [jFu13], a fuzzy logic package written in Java providing an implementation of the Fuzzy Control Language (FCL) to define fuzzy controllers.

The controller defines two input variables, `Position_Error` and `Deflection_Error`, and one output variable, `Control_Input`, and three linguistic variables, `negative`, `positive`, and `zero`. The control law is determined by nine rules (see Listing 5.3). Finally, the output variable is defuzzyfied with the *CenterOfGravity* method.

5.2.3 The Virtual and Remote Labs

5.2.3.1 The LabVIEW VI

The LabVIEW VI, *Flexlink [Top Level].VI*, provides the functionality discussed in Chapter 4.

```
RULEBLOCK first
AND : MIN;
ACT : MIN;
ACCU : MAX;
RULE 0 : IF (Position_Error IS Negative) AND (Deflection_Error IS Negative) THEN
      Control_Input IS Zero;
RULE 1 : IF (Position_Error IS Negative) AND (Deflection_Error IS Zero)     THEN
      Control_Input IS Positive;
RULE 2 : IF (Position_Error IS Negative) AND (Deflection_Error IS Positive) THEN
      Control_Input IS Positive;
RULE 3 : IF (Position_Error IS Zero)     AND (Deflection_Error IS Negative) THEN
      Control_Input IS Negative;
RULE 4 : IF (Position_Error IS Zero)     AND (Deflection_Error IS Zero)     THEN
      Control_Input IS Zero;
RULE 5 : IF (Position_Error IS Zero)     AND (Deflection_Error IS Positive) THEN
      Control_Input IS Positive;
RULE 6 : IF (Position_Error IS Positive) AND (Deflection_Error IS Negative) THEN
      Control_Input IS Negative;
RULE 7 : IF (Position_Error IS Positive) AND (Deflection_Error IS Zero)     THEN
      Control_Input IS Negative;
RULE 8 : IF (Position_Error IS Positive) AND (Deflection_Error IS Positive) THEN
      Control_Input IS Zero;
END_RULEBLOCK
```

Listing 5.3: Specification of the controller rules in the FCL.

The *data acquisition* subsystem is composed of the VIs (see Figure 5.12) that perform the initialization and release of the DAQ card, read the rotor position and strain angle, and write the input voltage to the motor. The readings and writings are done inside the main loop which is running with a period of 10 ms.

The *local controller* is a state feedback one, tuned to smoothly reposition the *Flexible Link* to the origin when the plant is outside of the safety operation area. The control logic is as follows: in normal mode, the control action received from the client side is forwarded to the actuators. Whenever the position of the arm is outside of the safety area ($\pm 45°$), the local *automatic mode* is switched on and the local controller inmediately takes care of the plant to prevent unsafe operating conditions which could lead to damage in the hardware. When the measured position is again near to the origin, the controller switch to *waiting* mode and stays there for at least 3 seconds. Eventually, the system will go back to normal operating mode and the client can send again the control action.

The *event-based communication* consists of two send-on-delta sampler sub VI as described in Chapter 4.

Finally, the *data logging* is constructed with the LabVIEW built-in IO capabilities, to write the plant state and control actions to disk.

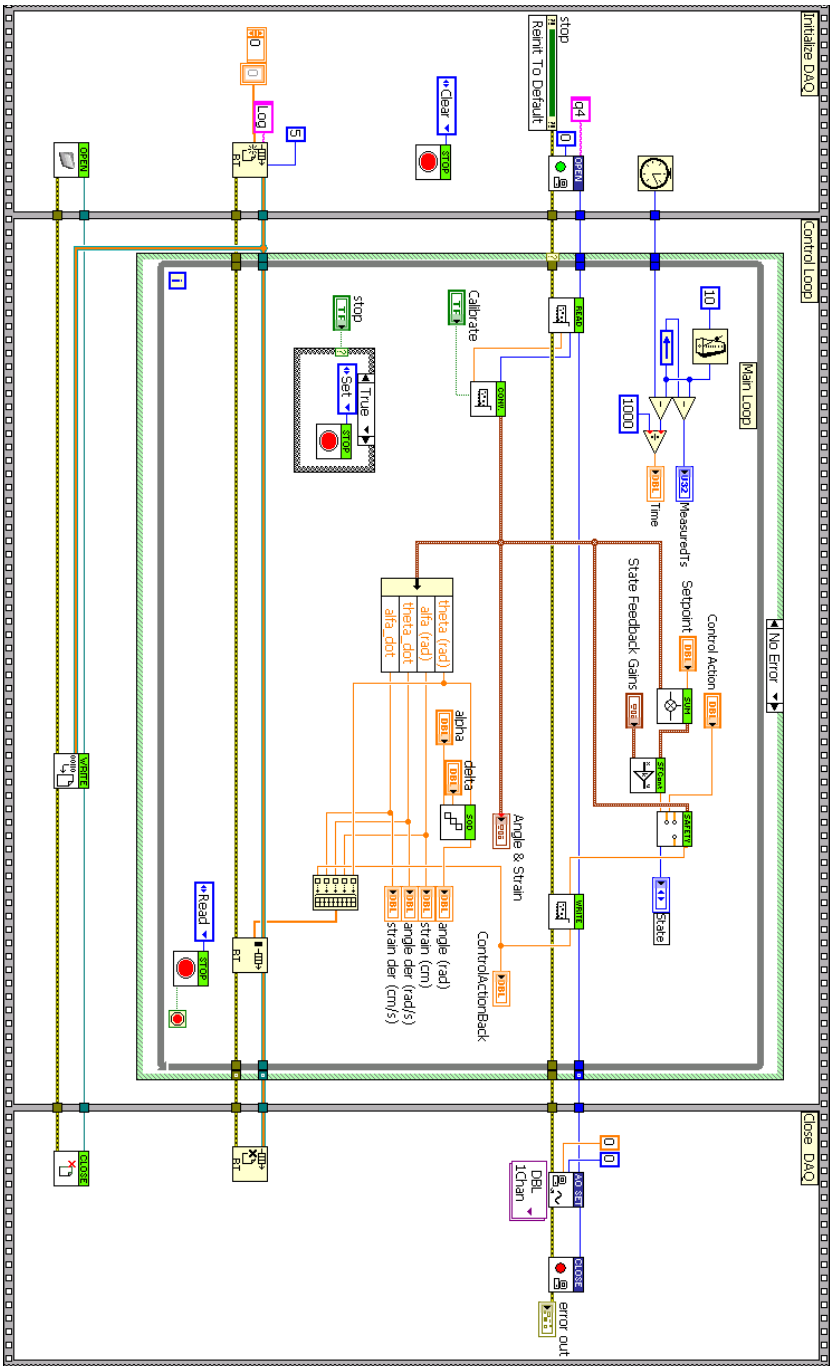


Figure 5.12: Following the division in subsystems discussed in Chapter 4.

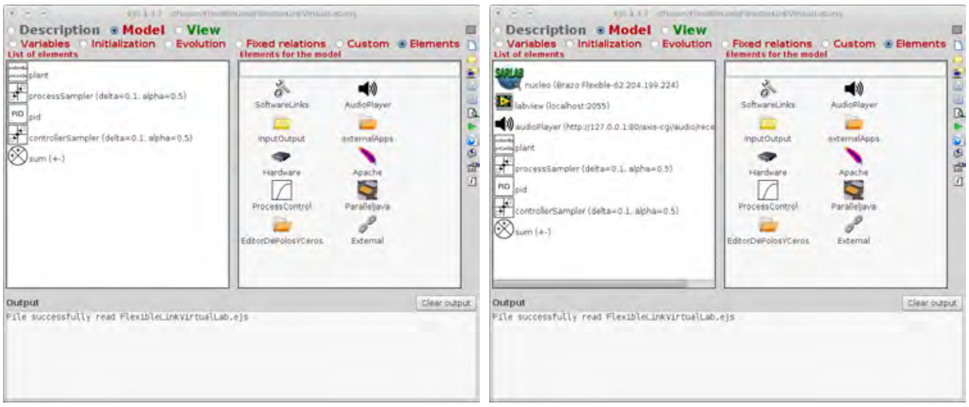
```java
public class StateFeedbackController extends AbstractBlock implements Discrete {
  protected double[] K;
  private double Ts;

  /**
   * Creates a new StateFeedbackController with the specified gains K, and period Ts
   * @param K
   * @param ts
   */
  public StateFeedbackController(double[] K, double ts) {
    setNumberOfOutputs(1);
    this.samplingPeriod = ts;
    setGains(K);
  }

  /**
   * Get the output
   * @param x The state vector
   * @param u The input vector
   * @return
   */
  @Override
  public double[] getOutput(double[] x, double[] u) {
    int n = u.length;
    double[] y = new double[]{0};
    for(int i=0; i<n; i++) { y[0] += K[i]*u[i]; }
    return y;
  }
}
```

Listing 5.4: Code of the class *StateFeedbackController*.

5.2.3.2 The simulation

The model used in the simulation is the fourth order model derived in the previous section. The implementation is based on the *PCL*, in particular the *Plant*, *PIDController*, and *StateFeedbackController* elements. The steps to build the application are commented in the following paragraphs, emphasizing the differences between the virtual and the remote lab.

Step 1: Adding the Elements Since the *StateFeedbackController* Element was not originally included in the *PCE* library, it has been implemented for this application, using the extension mechanism of the library. The class *StateFeedbackController* (see Listing 5.4) contains the implementation of the state feedback controller, extending the class *AbstractBlock*, and implementing the interface *Discrete*. The *StateSpaceModel*, *SOD Sampler*, and *PIDController* elements are added to the virtual and the remote lab applications. However, the *LabVIEW Connector*, and the *Audio Player* elements are only incorporated into the remote lab to add connectivity with LabVIEW and to get the audio feedback from the camera (see Figure 5.13).

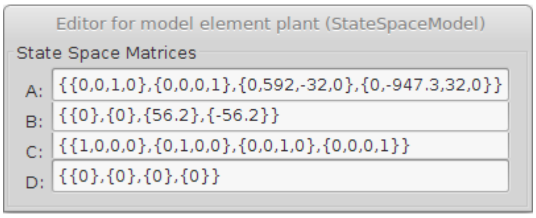


(a) (b)

Figure 5.13: *Model Element page* (a) of the virtual lab, and (b) of the remote lab.

The model of Equation 5.17 is implemented with the *StateSpaceModel* element. The matrices are introduced via the configuration dialog provided by the element (see Figure 5.14).

STEP 2: SETTING UP THE CONNECTION In this case, the remote lab has been integrated into the SARLAB system [MMAS13], developed in the University of Huelva, Spain (UHU), which introduces a middle layer that handles the connections, communication encryption, and other network issues to make the access to the laboratories secure and robust. The integration is achieved with the help of the *SARLAB element*.


Editor for model element plant (StateSpaceModel)

State Space Matrices

A: {{0,0,1,0},{0,0,0,1},{0,592,-32,0},{0,-947.3,32,0}}

B: {{0},{0},{56.2},{-56.2}}

C: {{1,0,0,0},{0,1,0,0},{0,0,1,0},{0,0,0,1}}

D: {{0},{0},{0},{0}}

Figure 5.14: Configuration dialog of the *StateSpaceModel* Element. The system matrices can be introduced either in Java format (the rows are surrounded by braces, {}, and separated by commas), or in MATLAB format (the rows are separated by semicolon (;), and the matrix is enclosed in square brackets, []).

```
1 // Load from 'FCL' file
2 String fileName = "FlexibleLink/controller.fcl";
3 fis = FIS.load(fileName,true);
4 if( fis == null ) {
5   System.err.println("Can't load file: '" + fileName + "'");
6 }
```

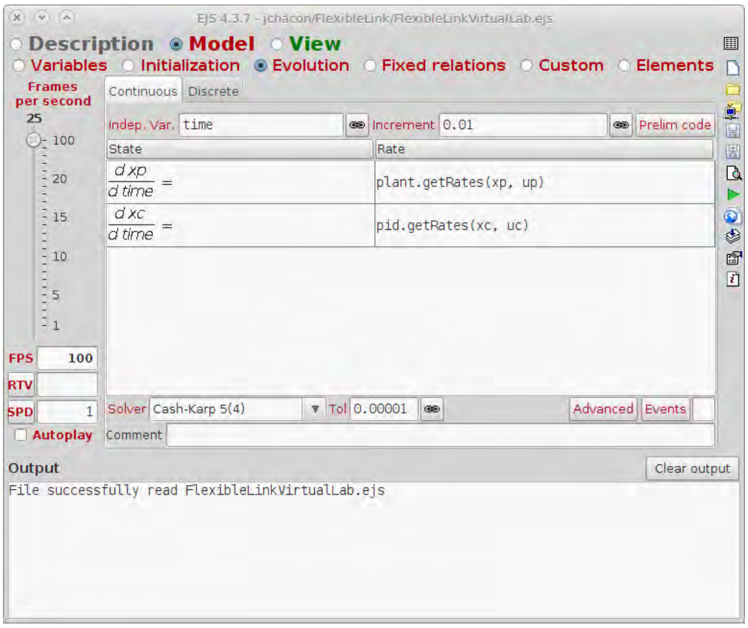
Listing 5.5: Initialization of the fuzzy controller

```
public void updateOutputs () {
  yp = plant.getOutput(xp, null);
  yps = processSampler.getOutput(xps, yp);
  po = processSampling ? yps : yp;
  uc = sum.getOutput(null, new double[]{setpoint, yps[0]});
  yc = pid.getOutput(xc, uc);
  ycs = controllerSampler.getOutput(xcs, yc);
  co = controllerSampling ? ycs : yc;

  switch(_view.controlPanel.getSelectedIndex()) {
    case 0: // StateFeedback
      uc = yp.clone();
      uc[0] -= setpoint;
      up = sf.getOutput(null, uc);
      break;
    case 1: // PID
      ys = sampler.getOutput(xs, yp);
      uc = sum.getOutput(null, new double[]{setpoint, ys[0]});
      up = yc;
      break;
    case 2: // Fuzzy
      up = uFuzzy;
      break;
  }
}
```

Listing 5.6: Code of the method *updateOutputs()*.

Finally, the connection with LabVIEW is done with the *LabVIEW Connector Element*. Because of the integration with SARLAB, the *address* of the server is set to *localhost:2055*, because the port 2055 (used by the *JIL Server*) of the localhost is tunneled through SARLAB to the host in the remote lab IP. For the same reason, the *Audio Player* element is configured to access the url *http:/localhost:8080/axis-cgi/audio*.

STEP 3: INITIALIZATION CODE The code included in the *Initialization page* that is worth mentioning is that related to the fuzzy controller (see Listing 5.5). The controller is loaded (line 3) from the file *FlexibleLink/controller.fcl*, which contains the definition in FCL language, and the reference to the controller is stored in the `fis` variable.



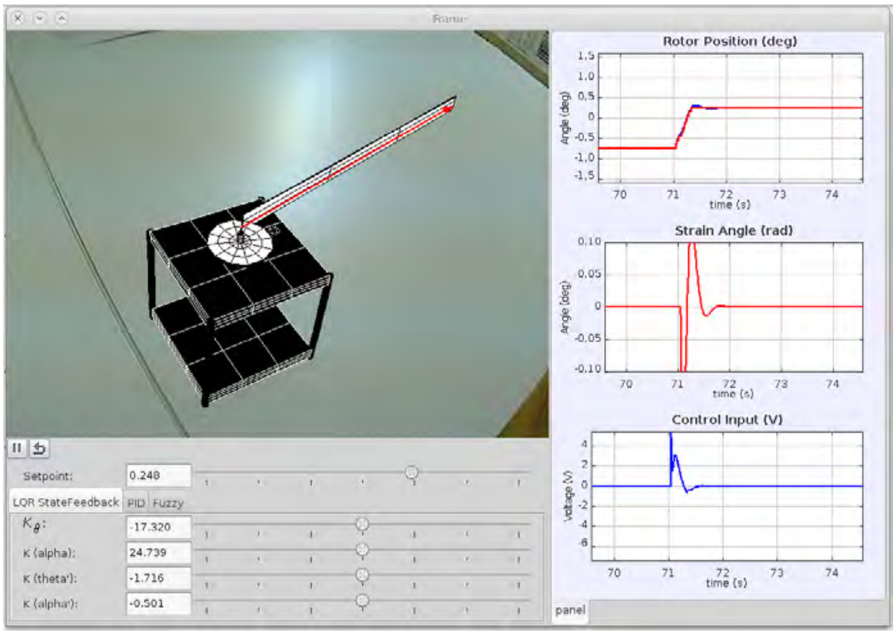
```
Indep. Var. time            Increment 0.01
State                       Rate
d xp / d time =             plant.getRates(xp, up)
d xc / d time =             pid.getRates(xc, uc)
Solver Cash-Karp 5(4)  Tol 0.00001
FPS 100
SPD 1
Output
File successfully read FlexibleLinkVirtualLab.ejs
```

Figure 5.15: *ODE page* of the model in the virtual lab, with an entry for the process, a *StateSpaceModel*, and another for the controller, a *PIDController*.

STEP 4: EVOLUTION CODE The *Evolution code* needed to integrate the simulation model consists of two parts: the first one is an *ODE Page* with the derivatives of the continuous systems (see Figure 5.15), and the second one is a *Code page* with the updates of the discrete blocks. The output of the blocks must be updated before to compute the derivatives. This is done with the code of Listing 5.6, in a *Preliminary Code Page* defined inside the *ODE page*.

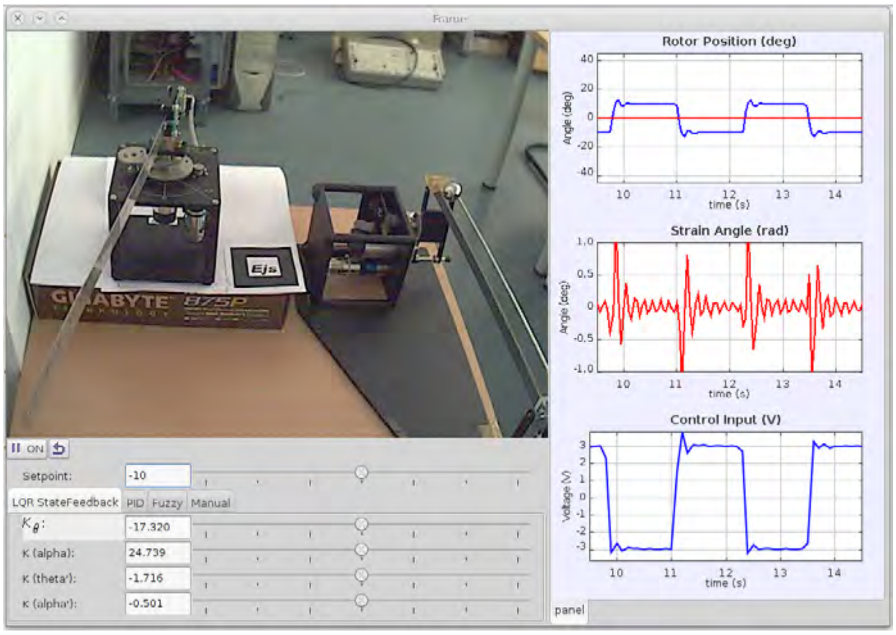
STEP 5: USER INTERFACE The virtual and the remote lab share the same graphical interface (see Figure 5.16), so it is easier for the student to move from one to another, carry on the same experiences and compare the differences between the behaviour of the simulation model and the real plant.

The interface has been designed following the guidelines of the other virtual and remote labs presented in this work. The visual representation or video cam image is shown at the top left part of the window (with a dimension of 640x480 pixels, which



Setpoint: 0.248
LQR StateFeedback | PID | Fuzzy
K_θ: -17.320
K (alpha): 24.739
K (theta'): -1.716
K (alpha'): -0.501

(a)



Setpoint: -10
LQR StateFeedback | PID | Fuzzy | Manual
K_θ: -17.320
K (alpha): 24.739
K (theta'): -1.716
K (alpha'): -0.501

(b)

Figure 5.16: Graphical user interface in EJS (a) of the virtual lab and (b) of the remote lab.

is equivalent to the webcam resolution), the right area of the window is reserved for the plots showing the variables of interest (the rotor position θ and the link deflection α as outputs, and the voltage V of the DC motor as input), and the control panel under the view contains all the parameters that can be adjusted and modified.

To make the simulation more attractive to the user, the Flexible Link has been modeled and animated to emulate the aspect and movement of the real plant. The visualization is built with the 3D capabilities provided by EJS (see Figure 5.16). Initially, the use of the *Java3D* library was considered because of its more advanced features, such as the use of textures that allows a higher degree of realism. However, the first option was finally preferred due to compatibility issues that affected to the portability of the simulations.

5.2.3.3 Integration with Moodle

Moodle is a free Learning Management System (LMS) that has became popular among educators as a tool to create online learning sites for their students. There are many reasons why the integration in a LMS like Moodel of a virtual or remote laboratory has many advantages, both from the point of view of the student, because she have more resources to help them with the study of the subject, and from the point of view of the developer, because there are aspects such as the access control or booking policies can be handle in a more generic and centralized way.

In particular, the Moodle platform is being used by the *UNEDLabs* portal to hold different courses mainly from UNED, but also from other Spanish universities such as the Universidad Complutense of Madrid or the Polytechnic University of Valencia. Thanks to the *EJSApp* set of plugins for Moodle, the integration of an EJS application with this LMS is straightforward, and does not require any modification of the EJS simulation.

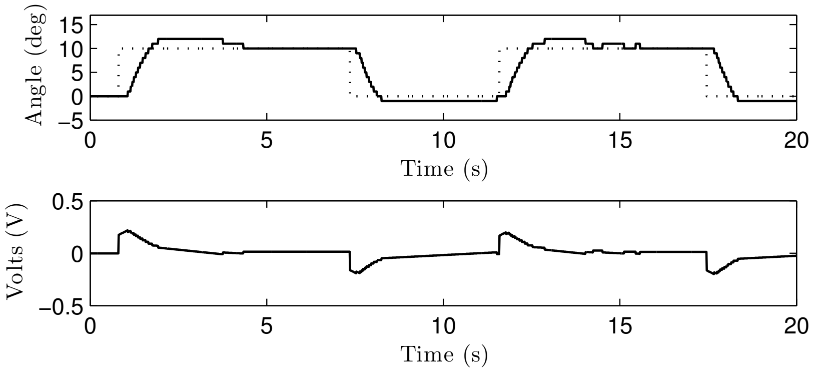
5.3 Results

To protect the hardware, a Safe Operating Area (SOA) has been delimited within the range of ($\pm 45°$). Outside of that area, the local controller takes care of the plant and repositions it to the origin. Also for safety reasons, the input to the motor has been limited to $\pm 5V$, to avoid an excessive speed of the Flexible Link.

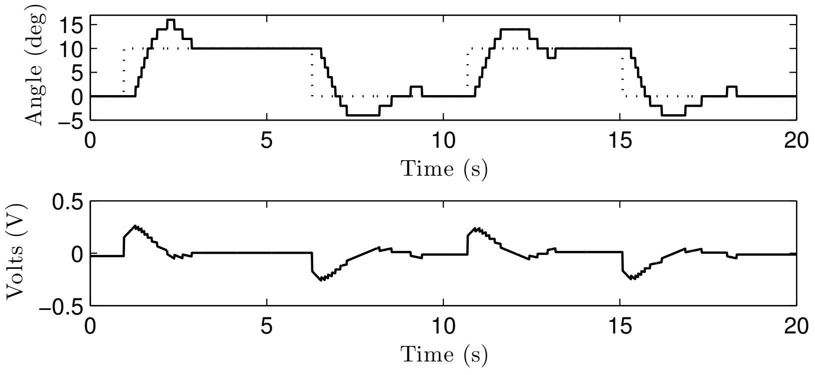
Because in the Quadruple Tank the dynamics is slow, the DAQ works with a sampling rate of $10Hz$, i.e. a period of $T_s = 100ms$, which is a value high enough as to not impose critical limitations to the communications. However, for the case of the Flexible Link the situation is more delicate due to the faster dynamics. For this plant, the DAQ is working with a period of $T_s = 10ms$ (ten times smaller than the previous case), which is fine for the local controller, but problematic to control remotely because of the network induced delays. So, the value of *delta* must be carefully chosen because an excessively small value provoke an excessive traffic, but for greater values the system the error increase, being difficult to stabilize the plant for $\delta > 5$. So, unless it is explicitly stated otherwise, the value of $\delta = 2$ has been used in the experiments with the sampler at the process output.

The PI parameters were adjusted to $k_p = 1, k_i = 0.2$, with the help of the MATLAB PI tuning tool, to obtain a phase margin of $60°$ which is a value frequently used. As opposed to the Quadruple Tank, where a variety of limit cycles with different orders were found, for the Flexible Link the behaviour is slightly different, because an increase in the controller gains rapidly leads the plant to the unstability. The response of the system with the PI controller and SOD sampler at the process output is shown in Figure 5.17a for different values of δ. As it can be seen, the response is very similar for the three cases. With these gains and $\alpha = 0$, the system does not present limit cycles.

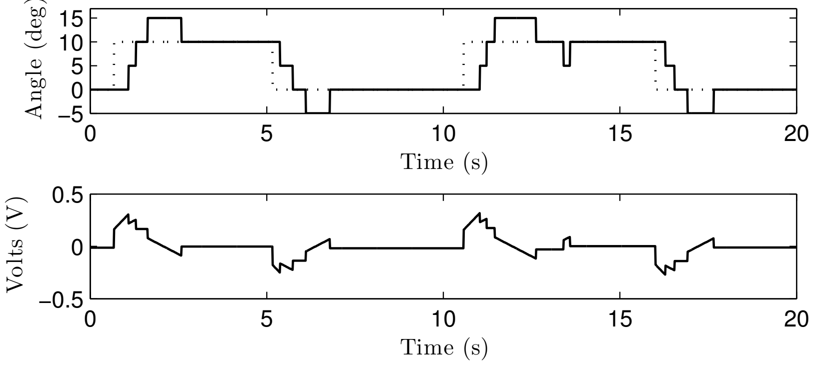
However, changing the value of α to 0.5, i.e. the centered sampler, the behaviour



(a) $k_p = 1$, $k_i = 5$, $\alpha = 0$, $\delta = 1$

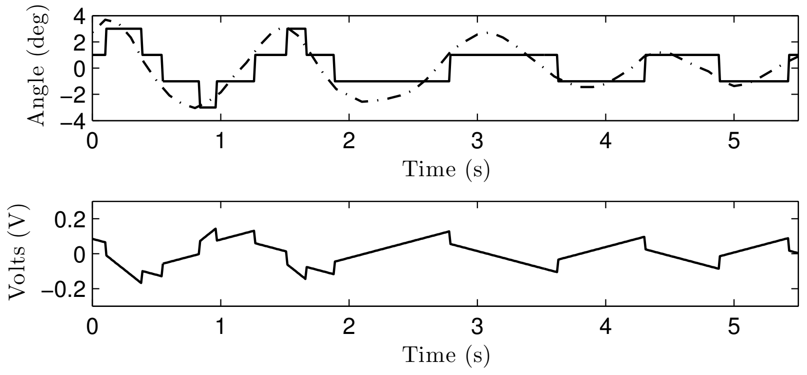


(b) $k_p = 1$, $k_i = 5$, $\alpha = 0$, $\delta = 2$

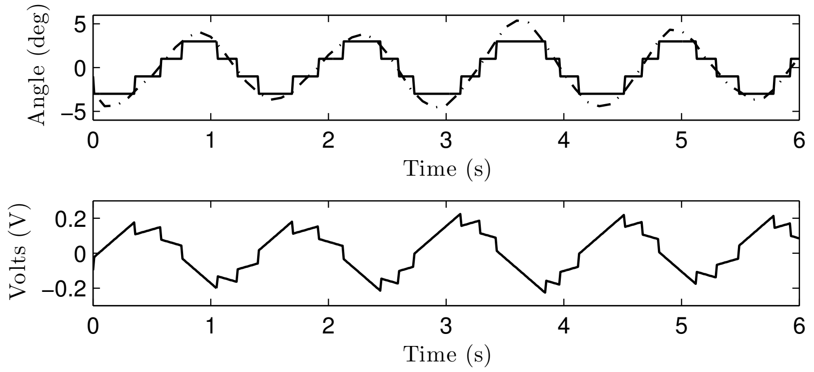


(c) $k_p = 1$, $k_i = 5$, $\alpha = 0$, $\delta = 5$

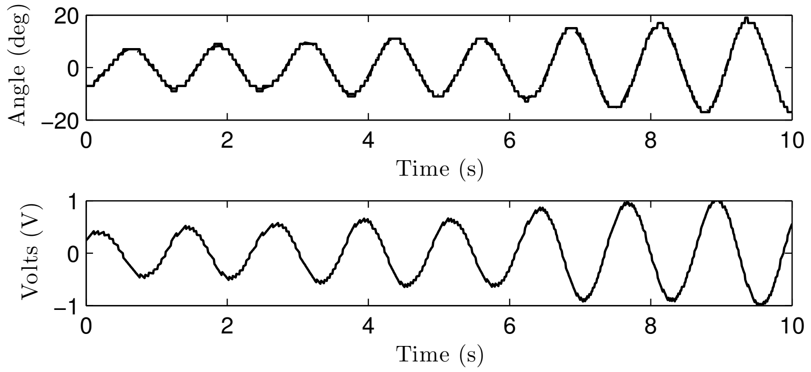
Figure 5.17: Step response of the process controlled by a PI with the event-based sampler at the process variable. The plots show the process and the controller output.



(a) $k_p = 1$, $k_i = 5.0$, $\delta = 2$, $\alpha = 0.5$



(b) $k_p = 1$, $k_i = 5.5$, $\delta = 2$, $\alpha = 0.5$



(c) $k_p = 1$, $k_i = 5.6$, $\delta = 2$, $\alpha = 0.5$

Figure 5.18: Different limit cycles in the process controlled by a PI, with the event-based sampler at the process variable. The plots show the process and the controller output.

of the system is different: the system tends to a stable limit cycle, as shown in Figure 5.18. Increasing the integral gain to $k_i = 5.5$, there exists also a limit cycle composed of four levels (see Figure 5.18b). However, the margin of stability in this case is narrow, a little increment in the integral gain ($k_i = 5.6$) leads to an oscillatory unstability (see Figure 5.18c).

5.4 Conclusions

A new paradigm of remote labs for control education presented in the previous chapter has been assessed by developing two platforms: one based on a Quadruple Tank plant and the other one based on a Flexible Link plant. For the first laboratory, one of its main features is to place the controller in the client side and the plant in the server-side. This architecture allows exploring in an experimental way the properties of event-based controllers. The platform has been used to demonstrate that the limit cycles proposed can be generated in real systems. In particular, the experiments carried out with the experimental plant have reproduced a wide range of these limit cycles as predicted by the theory discussed in Chapter 2. Though the algorithm has been used for a first-order model, it is more general. It can be applied to a LTI system controlled by a general linear controller. The effect of the sampler in the system can be analyzed by considering a modification in the matrices of the state space representation.

With respect to the second lab, the virtual version implements a 3D reproduction of the real plant, providing an atractive visual animation that adds realism to the simulation. On the other hand, the remote lab uses augmented reality to enhance the student experience by showing additional information overlayed to the video feedback obtained from the plant.

The implementation of both labs is done by using an architecture that has been proven to be adequate for remote laboratories. It is based on the use of EJS, *JIL*

Server, and LabVIEW. This platform allows the controller to be physically separated from the plant, and communicating with it through a network connection.

6 Conclusions and Future Works

6.1 Conclusions

The outcomes of the research done in this Thesis can be divided into analytic and experimental results.

6.1.1 Analytic results

The analityc results are related to the study of the behaviour of a PID control system based on the use of a level crossing sampling either in the process output or in the control output. The two proposed control schemes represent two frequent configurations for wireless systems, one with the controller and actuator in the same node, but the sensor is physically separated, and the other one with the controller and sensor in the same place but the actuator situated in another node.

Limit cycles are of particular interest since they are associated to oscillations in processes, and therefore it is worth to have knowledge about them in order to avoid their appearance when possible or to assure that they are not problematic, i.e. they are stable.

When trying to find properties about the limit cycles, it is common to have system of equations which involve transcendent functions and thus it is not possible, in general, to find closed-form solutions. Moreover, if they exist and due to the combinatorial explosion, it can be computationally expensive to find these solutions, and it becomes harder when higher order process models and SOD samplers are considered.

Therefore, an algorithm has been proposed to analyze the properties of the limit cycles, i.e. to obtain computationally the period of a limit cycle and the intermediate switching times. It allows us to introduce some knowledge in the problem statement, so that the complexity can be reduced, and it can be easily implemented either in a symbolic or in a numerical computation tool.

A set of simulation results illustrates the behaviour of the controllers with a set of processes models used very frequently in industrial context, which are the IPTD, the FOPTD, and the SOPTD. Also, this behaviour has been experimentally tested and verified in the Acurex Field of the Solar Platform of Almería, Spain. The experiments carried out confirm that the simulation results can be extrapolated to real cases, obviously with the divergences due to unmodeled dynamics of the process, disturbances, etc.

6.1.2 Experimental results

It is always desirable to validate the analytic results with experiments on real world systems. However, in general, the implementation of the experimentation platforms requires a significant cost, not only economic (the hardware can be expensive), but also in terms of development effort. Some implementation issues are plant specific, but also there are many common problems that can be extrapolated from one system to another, and thus the system design can be simplified. So, the main contribution of this Thesis in the experimental area is an architecture for rapid development of remote labs. The architecture is based on the use of LabVIEW, the *JIL Server*, and EJS, and allows educators who are not expert programmers to address the development of a remote lab with a minimized learning curve, due to the intuitivity of the graphical tools in the framework.

A significant effort has been dedicated to improve the ease of use, encapsulating all the low-level issues presented at the client side into the *EJS Model Element* mechanism, wrappers that allows us to easily incorporate Java libraries into EJS simulations, providing with a graphical user interface to help the developer with the configuration and use of the library.

The *LabVIEW Connector Element* allows to configure a connection with a LabVIEW VI, to link EJS variables with the controls and indicators of the VI, and to control the execution of the VI, and the *Audio Player Element* allows to play an audio streaming to provide the user with audio feedback from the plant. An important feature of the elements is that reduces the possibility of introducing errors in the code, thus reducing the time and effort needed for the development phase.

The result of the research has materialized into the following software components:

- A library of Java classes and EJS elements, the *Process Control Library (PCL)*, is now available to build simulations related with process control. The *PCEL* aims to facilitate the development of this kind of simulations, inspired by widely used tools such as SIMULINK, and capturing the behaviour of the most common types of systems. In order to build a new simulation, it is not needed to start from the scratch, but by only choosing and interconnecting blocks the application development effort is greatly reduced.

- The *Audio Player Element* that adds to EJS the capability to reproduce the sound transmitted by an IP camera through the network.

- The *Labview Connector Element* to connect, in only a few clicks, an EJS application with a LabVIEW VI.

- The MATLAB implementation of the algorithm presented in Chapter 2 to solve the problem of finding the limit cycles properties in a LTI system controlled by the proposed structures.

- The LabVIEW VI *SOD Sampler* that provides the send-on-delta sampling discussed in the Thesis.

But also, the new paradigm of remote labs for control education presented in this Thesis, and the mentioned software components have been used to develope the following virtual and/or remote laboratories for the study of limit cycles:

- A remote laboratory based on a Quadruple Tank plant (UNED, UNIBS). One of its main features is to place the controller in the client side and the plant in the server-side. This architecture allows exploring in an experimental way the properties of event-based controllers. The platform has been used to demonstrate that the limit cycles proposed can be generated in real systems. In particular, the experiments carried out with the experimental plant have reproduced a wide range of these limit cycles as predicted by the theory discussed in Chapter 2. Though the algorithm has been used for a first-order model, it is more general. It can be applied to a LTI system controlled by a general linear controller. The effect of the sampler in the system can be analyzed by considering a modification in the matrices of the state space representation.

- A platform based on a Flexible Link (UNED). The virtual version implements a 3D reproduction of the real plant, providing an atractive visual animation that adds realism to the simulation. On the other hand, the remote lab uses augmented reality to enhance the student experience by showing additional information overlayed to the video feedback obtained from the plant. The motivation to use this plant is because it is more complex than the Quadruple

Tank. So, for example, a problem in the communication channel will probably lead the plant to an instability, while in the previous case in general only provokes a decrease in the control performance.

The implementation of both labs is done by using an architecture that has been proven to be adequate for remote laboratories. It is based on the use of EJS, The *JIL Server*, and LabVIEW. This platform allows the controller to be physically separated from the plant, and communicating with it through a network connection.

In addition, the developed software components has been used in the development or enhancement of other remote platforms:

- A platform to teach concepts on electrical machines (UHU). In particular, the *Audio Player* element adds the capability to have audio feedback from the remote lab, which is important to help students have a closer experience with the plant, since the motor and the generator movement cannot be clearly perceived visually.
- A virtual laboratory to teach system identification theory and PI control on a gasoil furnace (ASU). In this case, the *PCE* library provides the implementation of the discrete PI controller with a cascade filter.
- A platform to teach Autonomous Robots subject (UNED), based on *LEGO Mindstorm* robots.

6.2 Future works

Lines of further work can be divided into analytic and experimental. Related to the analytic ones:

- The algorithm to find limit cycles is based on a time approach, but it is also possible to use frequency based approaches, such as the descriptive function, or a combination of both paradigms, to try and find a closed and general solution for n order systems and n order SOD sampler.

- In addition to the previous point, the parallelization of the algorithm could be explored to increase the efficiency to find solutions. In fact, some preliminary tests have been carried out in a five-node cluster with MATLAB, that show a significant reduction in the computational time.

and, on the other hand, in the experimental lines we propose:

- Though the architecture of the library and the most common blocks have been defined, there are more complex systems that could be added to it, to cover a wider range of systems. In addition, though the design of a new model has been simplified by providing built-in blocks that can be combined and interconnected, it is still needed to do the hard-work by code. It will be great to have a graphical tool to design the block diagrams.

- The use of library has been presented in a simulation context, and also in hardware-in-the-loop applications, by using the *LabVIEW connector* element for EJS, but it is also possible to combine with the real-time support of EJS and open hardware platforms, such as Arduino, Phidget, Gnublin, etc. Currently, the possibility of developing a remote laboratory based on a *BeagleBone Black* board is being considered. This board is a low-cost development platform with a *Linux* operating system and advanced I/O capabilities such as analog and digital inputs, built-in *PWM* outputs, *SPI* ad *I2C* buses, and other features that makes it adequate to build experimental platforms.

- The same ideas used in the *LabVIEW Connector* can be used to simplify the interconnection with other engineering tools, for example with MATLAB, Octave, etc. Though it is already possible to do that for most of the commonly used tools, a short term objective is to reach the same level of interconnectivity and platform portability (Linux, Mac OS, and Windows) that is currently available with LabVIEW.

Bibliography

[Å95] K.J. Åström. Oscillations in Systems with Relay Feedback. In *Adaptive Control, Filtering, and Signal Processing, IMA Volumes in Mathematics and its Applications*, volume 74. Springer-Verlag, 1995.

[Å99] K.E. Årzén. A Simple Event-Based PID Controller. In *14th IFAC World Congress*, Beijing, P.R. China, 1999.

[Å08] K.J. Åström. *Event-Based Control.* Springer-Verlag, Berlin, 2008.

[AB02] K.J. Åström and B. Bernhardsson. Comparison of Riemann and Lebesgue Sampling for First Order Stochastic Systems. In *41st IEEE conference on Decision and Control*, pages 2011–2016, Las Vegas NV, 2002. IEEE Control System Society.

[AH84] K.J. Åström and T. Hägglund. Automatic Tuning of Simple Regulators with Specifications on Phase and Amplitude Margins. *Automatica*, 20:645–651, 1984.

[AH05] K.J. Åström and T. Hägglund. *Advanced PID Control.* ISA - The

Instrumentation, Systems, and Automation Society, Research Triangle Park, NC 27709, 2005.

[ASP10] J. Almeida, C. Silvestre, and A.M. Pascoal. Self-Triggered State Feedback Control of Linear Plants under Bounded Disturbances. In *49th IEEE Conference on Decision and Control (CDC)*, pages 7588–7593, December 2010.

[AT09] A. Anta and P. Tabuada. Isochronous Manifolds in Self-Triggered Control. In *48th IEEE Conference on Decision and Control*, pages 3194–3199, 2009.

[AT10] A. Anta and P. Tabuada. Self-Triggered Control for Nonlinear Systems. *IEEE Transactions on Automatic Control*, 2010.

[AW96] K.J. Åström and B. Wittenmark. *Computer-Controlled Systems, Theory and Design.* Prentice-Hall, 1996.

[BDSV12a] M. Beschi, S. Dormido, J. Sánchez, and A. Visioli. Characterization of Symmetric Send-on-Delta PI Controllers. *Journal of Process Control*, 2012.

[BDSV12b] M. Beschi, S. Dormido, J. Sánchez, and A. Visioli. Tuning Rules for Event-Based SSOD-PI Controllers. In *20th Mediterranean Conference on Control and Automation*, Barcelona, 2012.

[Blu07] P.A. Blume. *The LabVIEW Style Book.* Prentice Hall, 2007.

[CA07] A. Cervin and K.J. Åström. On Limit Cycles in Event-Based Control System. In *46th Conference on Decision and Control*, pages 3190–3195, December 2007.

[CBR97] F. Camacho, M. Berenguel, and F.R. Rubio. *Advanced Control of Solar Plants.* Springer-Verlag, 1997.

[CE07] W. Christian and F. Esquembre. Modeling Physics with Easy Java Simulations. *The Physics Teacher*, 45(8):475–480, 2007.

[CGGV11] M. Casini, A. Garulli, A. Giannitrapani, and A. Vicino. A LEGO Mindstorms Multi-Robot Setup in the Automatic Control Telelab. In *18th IFAC World Congress*, 2011.

[Chr07] W. Christian. *Open Source Physics. A User's guide with Examples.* Pearson. Addison Wesley., 2007.

[CPV03] M. Casini, D. Prattichizzo, and A. Vicino. The Automatic Control Telelab: A User-Friendly Interface for Distance Learning. *IEEE Transactions on Education*, 46(2):252–257, 2003.

[CPV13] M. Casini, D. Prattichizzo, and A. Vicino. Remote Pursuer-Evader Experiments with Mobile Robots in the Automatic Control Telelab. In *10th IFAC Symposium on Advances in Control Education*, Sheffield, UK, 2013.

[CRBV07a] E.F. Camacho, F.R. Rubio, M. Berenguel, and L. Valenzuela. A Survey on Control Schemes for Distributed Solar Collector Fields. Part I: Modeling and Basic Control Approaches. *Solar Energy*, 81(10):1252–1272, October 2007.

[CRBV07b] E.F. Camacho, F.R. Rubio, M. Berenguel, and L. Valenzuela. A Survey on Control Schemes for Distributed Solar Collector Fields. Part II: Advanced Control Approaches. *Solar Energy*, 81(10):1252–1272, October 2007.

[CSVD12] J. Chacón, J. Sánchez, A. Visioli, and S. Dormido. Analysis of the Limit Cycles in the PI Control of IPD Processes with Send-on-Delta

Sampling. In *IEEE International Conference on Control Applications, Dubrovnik (HR)*, pages 1609–1614, 2012.

[CTG$^+$04] F. Candelas, F. Torres, P. Gil, F. Ortiz, S. Puente, and J. Pomares. Virtual Remote Laboratory and its Impact Evaluation in Academics. *Latinamerican Journal on Automation and Industrial Informatics*, 1(2):49–57, 2004.

[CVJ$^+$] R. Costa, M. Vallés, L.M. Jiménez, L. Díaz-Guerra, A. Valera, and R. Puerto. Integración de Dispositivos Físicos de un Laboratorio Remoto de Control mediante Diferentes Plataformas: LabVIEW, Matlab y C/C++. *Revista Iberoamericana de Automática e Informática Industrial RIAI.*

[DDCD$^+$05] S. Dormido, S. Dormido-Canto, R. Dormido, J. Sánchez, and N. Duro. The Role of Interactivity in Control Learning. *International Journal of Engineering Education*, 21(6):1122–1133, 2005.

[DDV$^+$08] N. Duro, R. Dormido, H. Vargas, S. Dormido-Canto, J. Sánchez, G. Farias, and S. Dormido. An Integrated Virtual and Remote Control Lab: The Three-Tank System as a Case Study. *Computing in Science and Engineering*, 10(4):50–59, 2008.

[DL12] O. Demir and J. Lunze. Event-Based Synchronisation of Multi-Agent Systems. In *4th IFAC Conference on Analysis and Design of Hybrid Systems (ADHS 12)*, Eindhoven, The Netherlands, 2012.

[Dor04] S. Dormido. Control Learning: Present and Future. In *Annual Reviews in Control*, volume 28, pages 115–136, 2004.

[DVD$^+$08] R. Dormido, H. Vargas, N. Duro, J. Sánchez, S. Dormido-Canto, G. Farias, F. Esquembre, and S. Dormido. Development of a Web-

Based Control Laboratory for Automation Technicians: The Three-Tank System. *IEEE Transactions on Education*, 51(1):34–44, 2008.

[EJS12] EJS. *http://fem.um.es/Ejs*, 2012.

[FDKDE10] G. Farias, R. De Keyser, S. Dormido, and F. Esquembre. Developing Networked Control Labs: a Matlab and Easy Java Simulations Approach. *IEEE Transactions on Industrial Electronics*, 57(10):3266–3275, 2010.

[FFDC+11] E. Fabregas, G. Farias, S. Dormido-Canto, S. Dormido, and F. Esquembre. Developing a Remote Laboratory for Engineering Education. *Computers & Education*, 57:1686–1697, 2011.

[GB09] L. Gomes and S. Bogosyan. Current Trends in Remote Laboratories. *IEEE Transactions on Industrial Electronics*, 56(12):4744–4756, 2009.

[GCBD12] J.L. Guzmán, R. Costa, M. Berenguel, and S. Dormido. *Control Automático con Herramientas Interactivas*. Pearson, May 2012.

[GDJ+13] M. Guinaldo, D.V. Dimarogonas, K.H. Johansson, J. Sánchez, and S. Dormido. Distributed Event-Based Control Strategies for Interconnected Linear Systems. *IET Control Thery & applications*, 7(6):877–886, 2013.

[GFF+13] M. Guinaldo, E. Fabregas, G. Farias, S. Dormido-Canto, D. Chaos, J. Sánchez, and S. Dormido. A Mobile Robots Experimental Environment with Event-Based Wireless Communications. *Sensor*, 13(7):9396–9413, 2013.

[GNR05] D. Gillet, A.V. Nguyen, and Y. Rekik. Collaborative Web-Based Experimentation in Flexible Engineering Education. *IEEE Transactions on Education*, 48:696–704, 2005.

[Gof07] J. Goffart. Design of a Web-Based Remote Lab for a Brewery Process. Master thesis, HAMK University of Applied Sciences, Finland, 2007.

[Gon00] J.M. Gonçalves. *Constructive Global Analysis of Hybrid Systems*. PhD thesis, Massachussetts Institute of Technology, 2000.

[GRDB12] J.L. Guzman, D.E. Rivera, S. Dormido, and M. Berenguel. An Interactive Software Tool for System Identification Education. *Advances in Engineering Software*, 45(1):115–123, 2012.

[Gui13] M. Guinaldo. *Contributions to Networked and Event-Triggered Control of Linear Systems*. PhD thesis, UNED, Madrid, Spain., July 2013.

[HD12] W.P.M.H. Heemels and M.C.F. Donkers. Model-Based Periodic Event-Triggered Control for Linear Systems. *Automatica*, 2012. accepted.

[HD13] W.P.M.H. Heemels and M.C.F. Donkers. Periodic Event-Triggered Control for Linear Systems. *IEEE Transactions on Automatic Control*, April 2013. to appear.

[hds13] http://www.mathworks.es/es/help/simulink/ug/simulating-dynamic systems.html?s_tid=doc_12b#bsn4198. *Simulating Dynamic Systems*, 2013.

[HJCV94] E. Hendricks, A. Jensen, A. Chevalier, and A. Vesterholm. Problems in Event Based Engine Control. In *American Control Conference*, pages 1585–1587, 1994.

[HMCC08] A.M. Hernández, M.A. Maanas, and R. Costa-Castelló. Learning Respiratory System Function in BME Studies by Means of a Virtual Laboratory: RespiLab. *IEEE Transactions on Education*, 51(1):24–34, 2008.

[HNX07] J.P. Hespanha, P. Naghshtabrizi, and Y. Xu. A Survey of Recent Results in Networked Control Systems. In *IEEE*, volume 95, 2007.

[htta] http://www.w3.org/Protocols/rfc1341/7_2_Multipart.html. *RFC1341*.

[httb] http://xmlrpc.scripting.com/default.html. *XML-RPC Site*.

[htt07] http://xmlrpc.scripting.com/spec.html. *XML-RPC Specification*, 2007.

[jFu13] jFuzzyLogic. *http://jfuzzylogic.sourceforge.net/html/index.html*, 2013.

[JGA98] M. Johansson, M. Gafvert, and K.J. Astrom. Interactive Tools for Education in Automatic Control. *Control Systems, IEEE*, 18(3):33 –40, jun 1998.

[Joh97] K.H. Johansson. *Relay Feedback and Multivariable Control*. PhD thesis, Department of Automatic Control, Lund Institute of Technology, Sweden, 1997.

[Joh00] K.H. Johansson. The Quadruple-Tank Process: a Multivariable Laboratory Process with an Adjustable Zero. *Control Systems Technology, IEEE Transactions on*, 8(3):456–465, 2000.

[KK12] R. Krasnansky and A. Kozáková. State Space Control Design: An Interactive Tool for Control Education. In *9th IFAC Symposium Advances in Control Education*, volume 9, Russia, 2012.

[KKLP99] W.H. Kwon, Y.H. Kim, S.J. Lee, and K.N. Paek. Event-Based Modeling and Control for the Burnthrough Point in Sintering Processes. In *IEEE Transactions on Control Systems Technology*, 1999.

[KS07] H. Kojima and W. Singhose. Adaptive Deflection-Limiting Control for

Slewing Flexible Space Structures. *Journal of Guidance Control and Dynamics*, 30:61–67, 2007.

[Lab] LabVIEW White Paper. http://www.ni.com/white-paper/7001/en.

[Lab13] LabVIEW home page. http://www.ni.com/labview, 2013.

[LJ12] D. Lehmann and K.H. Johansson. Event-Triggered PI Control Subject to Actuator Saturation. In *IFAC Conference on Advances in PID Control*, 2012.

[LLJ12] D. Lehmann, J. Lunze, and K.H. Johansson. Comparison Between Sampled-Data Control, Deadband Control and Model-Based Event-Triggered Control. In *4th IFAC Conference on Analysis and Design of Hybrid Systems (ADHS 12)*, Eindhoven, The Netherlands, 2012.

[LMA10] A. Lareki, J. Martínez, and N. Amenabar. Towards an Efficient Training of University Faculty on ICTs. *Computers & Education*, 54(2):491–497, 2010.

[LO99] X. Liu and J. Onoda. Controller Design for Vibration Suppression of Slewing Flexible Structures. *Computers & Structures*, 70(1):119 – 128, 1999.

[MJC12] M. Mazo Jr. and M. Cao. Decentralized Event-Triggered Control with One Bit Communications. In *4th IFAC Conference on Analysis and Design of Hybrid Systems (ADHS 12)*, Eindhoven, The Netherlands, 2012.

[MMAS13] A. Mejías, M.A. Márquez, J.M. Andújar, and M.R. Sánchez. A Complete Solution for Developing Remote Labs. In *10th IFAC Symposium on Advances in Control Education*, Sheffield, UK, 2013.

[MRR13] M. Morlacchi, F. Resta, and F. Ripamonti. An Adaptive Non-Model Based Control Logic for Vibration Suppression in Flexible Structures. In *Mechatronics (ICM), 2013 IEEE International Conference on*, pages 63–68, 2013.

[MZ12] Z. Magyar and K. Zakova. SciLab Based Remote Control of Experiments. *Advances in Control Education*, 9(1):206–211, 2012.

[PDF+13] M.A. Prada, M. Domínguez, J.J. Fuertes, P. Barrientos, C.J. del Canto, and S. García. Remote Laboratory with an Electro-Pneumatic Classification Cell. In *10th IFAC Symposium on Advances in Control Education*, Sheffield, UK, 2013.

[RJP+03] R. Barrett, J. Cona, P. Hyde, B. Ketcham, B. Kinney, and J. Schakelman. *Virtual Microscope*. University of Delaware, 2003.

[RMV09] M.L. Ruz, F. Morilla, and F. Vázquez. Teaching Control with First Order Time Delay Model and PI Controllers. In *8th IFAC Symposium Advances in Control Education*, volume 8, Japan, 2009.

[SCMS11] M. Stefanovic, V. Cvijetkovic, M. Matijevic, and V. Simic. A LabVIEW-Based Remote Laboratory Experiments for Control Engineering Education. *Computer Applications in Engineering Education*, 19(3):538–549, 2011.

[SD83] F.A. Schraub and H. Dehne. Electric Generation System Design: Management, Startup and Operation of IEA Distributed Collector Solar System in Almería, Spain. *Solar Energy*, 31(4):351–354, 1983.

[SDPM04] J. Sánchez, S. Dormido, R. Pastor, and F. Morilla. A Java/Matlab-Based Environment for Remote Control System Laboratories: Illustrated With an Inverted Pendulum. *IEEE Transactions on Education*, 47(3):321–329, 2004.

[SJJ92] D.W. Sparks Jr. and J.N. Juang. Survey of Experiments and Experimental Facilities for Control of Flexible Structures. *Journal of Guidance, Control, and Dynamics*, 15(4), 1992.

[SMD$^+$02] J. Sánchez, F. Morilla, S. Dormido, J. Aranda, and P. Ruipérez. Virtual and Remote Control Labs Using Java: A Qualitative Approach. *IEEE Control System Magazine*, 22(2):8–20, 2002.

[SPA03] S.P. Singh, H.S. Pruthi, and V.P. Agarwal. Efficient Modal Control Strategies for Active Control of Vibrations. *Journal of Sound and Vibration*, 262(3):563 – 575, 2003. ¡ce:title¿2001 India-USA Symposium on Emerging Trends in Vibration and Noise Engineering¡/ce:title¿.

[SRR13] M. Serra, F. Resta, and F. Ripamonti. An Active Control Logic Based on Modal Approach for Vibration Reduction through the Eigenstructure Assignement. In *Mechatronics (ICM), 2013 IEEE International Conference on*, pages 58–62, 2013.

[SVD12] J. Sánchez, A. Visioli, and S. Dormido. *PID Control in the Third Milennium*, chapter Event-based PID control, pages 495–526. Springer, 2012.

[Tab07] P. Tabuada. Event-Triggered Real-Time Scheduling of Stabilizing Control Tasks. *IEEE Transactions on Automatic Control*, 52(9):1680–1685, July 2007.

[TJ12] U. Tiberi and K. H. Johansson. A Simple Self-triggered Sampler for Nonlinear Systems. In *4th IFAC Conference on Analysis and Design of Hybrid Systems (ADHS 12)*, Eindhoven, The Netherlands, 2012.

[TXB96] T.J. Tarn, N. Xi, and A. Bejczy. Path-Based Approach to Integrated Planning and Control for Robotic Systems. *Automatica*, 32(12):1675–1687, 1996.

[TY93] A. Tzes and S. Yurkovich. An Adaptive Input Shaping Control Scheme for Vibration Suppression in Slewing Flexible Structures. *Control Systems Technology, IEEE Transactions on*, 1(2):114–121, 1993.

[Var10] H. Vargas. *An Integral Web-based Environment for Control Engineering Education*. PhD thesis, UNED, Madrid, Spain, 2010.

[VK06] V. Vasyutinskyy and K. Kabitzsch. Implementation of PID Controller with Send-on-Delta Sampling. In *ICC'2006, International Conference on Control*, Glasgow, Scotland, 2006.

[VSD$^+$08] H. Vargas, J. Sánchez, N. Duro, R. Dormido, R. Dormido-Canto, G. Farias, S. Dormido, Ch. Salzmann, and D. Gillet. A Systematic Two-Layer Approach to Develop Web-Based Experimentation Environments for Control Engineering Education. *Intelligent Automation and Soft Computing*, 14(4):505–524, 2008.

[VSGD09] H. Vargas, Ch. Salzmann, D. Gillet, and S. Dormido. Remote Experimentation Mashup. In *Proc. 8th IFAC Symposium on Advances in Control Education (ACE09)*, Kumamoto, 2009.

[VSJ$^+$11] H. Vargas, J. Sánchez, C.A. Jara, F.A. Candelas, F. Torres, and S. Dormido. A Network of Automatic Control Web-Based Laboratories. *IEEE Transactions on Learning Technologies*, 4(3):197–208, 2011.

[VSS$^+$09] H. Vargas, J. Sánchez, Ch. Salzmann, F. Esquembre, D. Gillet, and S. Dormido. Web-Enabled Remote Scientific Environments. *Computing in Science and Engineering*, 11(3):34–46, 2009.

[WAJ12] J. Weimer, J. Araújo, and K.H. Johansson. Distributed Event-Triggered Estimation in Networked Systems. In *4th IFAC Conference on Analysis and Design of Hybrid Systems (ADHS 12)*, Eindhoven, The Netherlands, 2012.

[WL09] X. Wang and M. Lemmon. Self-Triggered Feedback Systems with State-Independent Disturbances. In *American Control Conference*, Riverfront, St. Louis, MO, USA, 2009.

[YA11] H. Yu and P. J. Antsaklis. Event-Triggered Output Feedback Control for Networked Control Systems using Passivity: Time-varying Network Induced Delays. In *50th IEEE Conference on Decision and Control and European Control Conference (CDC-ECC)*, December 2011.

APPENDICES

A
Resumen en español

A.1 Introducción

El control basado en eventos es un paradigma de control que surge de manera natural ya que es similar a la forma de actuar de las personas. Su idea fundamental es el concepto de evento, que es algo que ocurre en el sistema y que, de alguna forma, conlleva la necesidad de actuar. Esto supone un cambio significativo comparado con otros enfoques como el control analógico, donde se actúa de forma continua, o el control digital, donde se actúa de forma periódica con una base de tiempos fija. Sin embargo, esta forma de controlar sistemas resulta más intuitiva, ya que el ser humano, en cierto modo, actúa así: respondemos a eventos que nos llegan del exterior, y que nos hacen elegir una u otra acción. Por otra parte, mientras en otros campos del control existe una amplia teoría matemática bien conocida y consolidada, que nos facilita el enfoque sistemático y la resolución de los problemas de control clásico, el control por eventos ha sido usado tradicionalmente en la práctica, basado en conocimiento empírico y heurísticas o en la resolución de problemas particulares. Sin embargo, dado el interés de muchos investigadores en el campo, y la aportación de diferentes formalismos y marcos de trabajo, esta situación ha cambiado, y, aunque todavía se encuentra lejos de alcanzar el grado de madurez de otras ramas del control, ya hay resultados importantes. Por esta razón, es importante no solo la investigación en técnicas de análisis y síntesis de controladores basados en eventos, sino que cada vez es mayor la necesidad de incorporar estos temas en el currículum del estudiante, si bien no en cursos de iniciación al control, sí en otros más avanzados.

Por otra parte, hoy día Internet es una herramienta omnipresente en nuestras vidas, a través de las conexiones de banda ancha disponibles en los lugares de trabajo y en la mayoría de los hogares, e incluso, cada vez más, estando continuamente conectados a la red con dispositivos móviles como smartphones o tabletas. En el terreno de la educación, el estudiante utiliza Internet continuamente como fuente de conocimientos, en detrimento de otros soportes más tradicionales como los libros. Esta nueva forma de afrontar la enseñanza ha llevado a los educadores a explotar los beneficios de herramientas interactivas para la enseñanza, que permiten sacar partido de la tecnología para conseguir llegar al alumno de formas impensables en otras épocas.

En el caso de las materias experimentales, es conocida la importancia del uso de los laboratorios para aprender a enfrentarse a problemas prácticos que en una clase teórica es muy difícil de transmitir. Sin embargo, la creacion y el mantenimiento de laboratorios es costoso y, en muchos casos, no es afrontable por muchas universidades. Incluso si es posible afrontar el coste económico, la explotación de los laboratorios está en ocasiones limitada por la disponibilidad del profesor para supervisar las experiencias, lo que conlleva una infrautilización de los recursos. Por este motivo, es cada vez más frecuente, sobre todo en universidades a distancia como la UNED, que además posee un número tan elevado de alumnos que agrava aun más el problema con el uso de laboratorios presenciales, la incorporación de laboratorios remotos que permiten al estudiante, a través de una conexión desde su casa o desde otro centro no necesariamente situado cerca del laboratorio. Un laboratorio remoto permite, por un lado, el eliminar la necesidad por parte del alumno de trasladarse físicamente al lugar donde se realizan las prácticas y, por otro lado, permite una mayor utilización de los recursos gracias a la automatización de experiencias, el uso de sistemas de reservas, etc.

Dentro de las herramientas interactivas, también podemos considerar los laboratorios virtuales, que permiten al estudiante realizar experiencias sobre un sistema simulado, que imita el comportamiento de otro sistema real. Aunque la simulación no puede captar toda la riqueza del comportamiento del sistema real, sí que posee otras ventajas como la capacidad de realizar cualquier tipo de experiencia sin temor a provocar daños físicos en la planta. Por este motivo, son muy adecuados para presentar al alumno como primera toma de contacto para adquirir conocimiento sobre un sistema, para una vez que ha superado unos mínimos permitirle acceder a la experimentación con el sistema real, lo que a su vez también permite un uso más racional de los recursos y un mayor grado de concurrencia, ya que aunque el acceso al sistema real está limitado, no ocurre así con el sistema virtual.

Aun así, el desarrollo de laboratorios virtuales y remotos supone también un coste en esfuerzo. Es más, muchas veces es el mismo educador, que no tiene por que ser un experto desarrollador, el que debe sacrificar tiempo dedicado al diseño de las experiencias y el material educativo, para enfrentarse a resolver problemas de implementación prácticos. En general, es deseable contar con patrones de solución y con herramientas de desarrollo dentro de estos laboratorios que faciliten el proceso y disminuyan el tiempo de diseño y, particularizando en el caso de los sistemas por eventos, es interesante la posibilidad de abstraer características comunes a estos sistemas y encapsularlas en componentes de software que permitan lo anterior. Además, de esta forma se favorece la adopción de soluciones más robustas y que se beneficien de la experiencia obtenida anteriormente por otros desarrolladores.

En resumen, es importante el estudio de los sistemas de control basados en eventos tanto en su aspecto teórico, para aportar resultados que ayuden en el análisis y diseño de controladores que resuelvan problemas prácticos en los que han demostrado poseer ventajas con respecto a las técnicas tradicionales, como es el caso de los sistemas de control en red, sistemas multiagente, etc. y, por otra parte, es necesario contribuir con herramientas que faciliten el desarrollo de plataformas de experimentación con sistemas de control basados en eventos tanto para la investigación sobre nuevos algoritmos de control como para su uso en el terreno de la educación.

A.2 Objetivos

El objetivo general de la tesis es investigar, diseñar e implementar sistemas de control PID basados en eventos. Se aborda el estudio de dos estructuras genéricas de control por eventos y sus ciclos límites asociados, así como la simplificación de los métodos existentes para desarrollar plataformas de experimentacion y enseñanza del control basado en eventos.

Los objetivos específicos considerados en esta tesis son:

- El estudio de dos estructuras genéricas de control basado en eventos, centrado en las propiedades de los ciclos límite (amplitud y periodo) que presentan los sistemas controlados por estas estructuras.
- La confirmación de la existencia de estos ciclos límites en simulación y en plantas reales.
- El desarrollo de componentes software que permitan un rápido desarrollo de laboratorios virtuales y remotos, encapsulando la comunicación basada en eventos entre las aplicaciones cliente y servidor.
- La implementación de laboratorios virtuales y remotos con los componentes desarrollados, para garantizar el rendimiento y la facilidad de uso.

A.3 Estructura y contribuciones

En el resto de este capítulo y en la primera parte del siguiente, se describen las dos estructuras de control analizadas en esta tesis. Estos esquemas se corresponden con dos casos frecuentemente usados en el control de sistemas con transmisiones inalámbricas.

Capítulo 2. Presenta el problema de los ciclos límite que pueden aparecer en la clase de sistemas de control estudiados, así como el marco teórico usado en el análisis. La principal contribución de este capítulo es la proposición de un algoritmo que permite obtener computacionalmente las propiedades de los posibles ciclos límite de un proceso controlado por una de las estructuras basadas en eventos consideradas.

Capítulo 3. Presenta una nueva librería de clases Java y elementos EJS para desarrollar simulaciones y laboratorios virtuales dedicados a la enseñanza de control de procesos, y, en particular, de sistemas de control basados en eventos.

Capítulo 4. Discute la creación de laboratorios remotos, centrándose en los aspectos relacionados con las transmisiones basadas en eventos. Se propone una nueva arquitectura basada en el uso de elementos de EJS, que permite un rápido desarrollo de laboratorios remotos.

Capítulo 5. La primera parte del capítulo presenta un laboratorio virtual y remoto para controlar una planta de cuatro tanques acoplados. En la segunda parte se discute el desarrollo de un laboratorio virtual y remoto con un brazo flexible.

Capítulo 6. Se dan las conclusiones y líneas de trabajo futuras.

A.4 Publicaciones

Artículos de revista:

1. J. Chacón, J. Sánchez, A.Visioli, L.Yebra, S.Dormido. "Characterization of limit cycles for self-regulating and integral processes with PI control and send-on-delta sampling". *Journal of Process Control*, Vol. 23, No. 6, pp. 826-838, 2013. IF: 1.805.

2. D. Chaos, J. Chacón, J.A. López-Orozco, S. Dormido. "Virtual and Remote Robotic Laboratory Using EJS, MATLAB and LabVIEW". *Sensors*, 13, 2, pp. 2595-2612, doi:10.3390/s130202595, 2013. IF: 1.953.

3. J. Chacón, H. Vargas, G. Farias, J. Sánchez, and S. Dormido. "EJS, JIL and LabVIEW: How to build a remote lab in the blink of an eye". *Transactions on Industrial Electronics*. IF: 5.165. (submitted)

Artículos de conferencias:

1. J. Chacón, J. Sánchez, S. Dormido. "Experimental study of an event-based PID controller in a wireless system". *9th Portuguese Conference on Automatic Control (Controlo 2010)*, Coimbra, pp. 142-147, 2010.

2. J. Chacón, J. Sánchez, S. Dormido, A. Visioli, "Design of an event-based feed-forward strategy for SOPTD processes", *50th IEEE Conference on Decision and Control and European Control Conference*, Orlando (FL), pp. 5431-5436, 2011.

3. J. Chacón, J. Sánchez, A. Visioli, S. Dormido. "Decentralised control of a quadruple tank with a decoupled event-based strategy", *IFAC Conference on Advances in PID Control*, Brescia (I), pp. 424-429, 2012.

4. J. Chacón, J. Sánchez, A. Visioli, S. Dormido. "Analysis of the limit cycles in the PI control of IPD processes with send-on-delta sampling", *IEEE International Conference on Control Applications*, Dubrovnik (HR), pp. 1609-1614, 2012.

5. D. Chaos, J. Chacón, J.A. López-Orozco, S. Dormido. "Enseñando robótica movil con laboratorios remotos." *XXII Jornadas de Automática*, Vigo, pp. 769-774, 2012.

6. J. Chacón, J. Sánchez, L. Yebra, A. Visioli, S. Dormido. "Experimental study of two event-based PI controllers in a solar distributed collector field", *European Control Conference*, Zurich (CH), pp. 626-631, 2013.

7. J. Chacón, J. Sánchez, A. Visioli, S. Dormido. "Building process control simulations with Easy Java Simulations elements", *IFAC Symposium on Advances in Control Education, Sheffield (UK)*, pp. 138-143, 2013.

8. J. Chacón, D. Chaos, H. Vargas, G. Farias, J. Sánchez, and S. Dormido. "A new methodology to build remote laboratories with EJS Elements". *Multimedia on Physics Teaching and Learning (MPTL'18)*, Madrid, 2013.

A.5 Proyectos de investigación

Los resultados obtenidos durante el desarrollo de esta tesis han sido soportados por diferentes proyectos de investigación:

- Event-based modeling, simulation, and control (2007-2012). Spanish Ministry of Science and Technology, CICYT (Ref. DPI2007-61068). Participantes: UNED (Spain), University of Murcia (Spain). Dirigido por Prof. Sebastián Dormido Bencomo.

- MACROBIO: Modeling, simulation, control and optimization of photobiorreactors (2012-2014). Spanish Ministry of Economy and Competitiveness, CICYT (Ref. DPI2011-27818-C02-2). Participantes: UNED (Spain). Dirigido por Prof. José Sánchez Moreno.

A.6 Conclusiones

Los resultados de la investigación realizada en esta tesis puede dividirse en dos partes, teóricos y experimentales.

A.6.1 Resultados teóricos

Los resultados analíticos están relacionados con el estudio del comportamiento de un sistema de control PID basado en el uso de un muestreo por cruce de nivel, ya sea en la salida del proceso o en la salida del controlador. Los dos esquemas de control propuestos representan dos configuraciones frecuentes para los sistemas inalámbricos, una con el controlador y el actuador en el mismo nodo, pero el sensor físicamente separado, y la otra con el controlador y el sensor en el mismo lugar, pero el actuador situado en otro nodo.

Los ciclos límite son de particular interés ya que están asociados a oscilaciones en los procesos y, por lo tanto, vale la pena tener conocimiento sobre ellos con el fin de evitar su aparición cuando sea posible o, en caso contrario, al menos para asegurar que no son problemáticos, es decir, que son estables.

Al tratar de estudiar analíticamente propiedades sobre los ciclos límite, es frecuente encontrarse con sistemas de ecuaciones que involucran funciones trascendentes y, por lo tanto, no es posible en general encontrar soluciones cerradas. Por otra parte, si es que existen, y debido a la explosión combinatoria, puede ser costoso encontrar estas soluciones computacionalmente, lo que se hace más difícil aun cuando se consideran modelos de proceso y muestreadores SOD de órdenes superiores.

Por lo tanto, se ha propuesto un algoritmo para analizar las propiedades de los ciclos límite, es decir, para obtener computacionalmente el período de un ciclo límite y los tiempos de conmutación intermedios. Esto nos permite introducir conocimiento sobre los sistemas estudiados en el planteamiento del problema, de manera que la complejidad puede ser reducida, lo que a su vez hace que se puede implementar fácilmente, ya sea en una herramienta software de cálculo simbólico o en una de cálculo numérico .

El comportamiento de los controladores estudiados se ilustra con resultados de simulación sobre un conjunto de modelos de procesos que se utilizan con frecuencia en el ámbito industrial, como son el IPTD, el FOPTD, y el SOPTD. Además, este comportamiento ha sido probado y verificado experimentalmente en el sistema Acurex de la Plataforma Solar de Almería, España. Los experimentos llevados a cabo confirman que los resultados de la simulación se pueden extrapolar a los casos reales, obviamente con las divergencias debidas a la dinámica no modelada del proceso, perturbaciones externas, etc.

A.6.2 Resultados experimentales

Siempre es deseable validar los resultados análiticos con experimentos en sistemas del mundo real. Sin embargo, en general, la implementación de plataformas de experimentación conlleva un coste significativo, no sólo en el aspecto económico (el hardware puede ser caro), sino también en términos de esfuerzo de desarrollo. Algunas de las dificultades de implementación son específicas del sistema con el que se trata, pero también existen una serie de problemas comunes que pueden ser extrapolados de un sistema a otro y, por tanto, el diseño puede ser sistematizado hasta cierto punto. Para ayudar en este aspecto, la principal contribución de esta tesis en el terreno experimental es la proposición de una arquitectura que permite un desarrollo rápido de laboratorios remotos. Ésta se basa en el uso de *LabVIEW*, *JIL Server* y EJS, y permite a los educadores, que no tienen por qué ser expertos programadores, acometer el desarrollo de laboratorios remotos con una curva reducida de aprendizaje, gracias al uso intuitivo de las herramientas gráficas en el marco de trabajo propuesto.

Un esfuerzo importante se ha dedicado mejorar la facilidad de uso, encapsulando todos los detalles de bajo nivel que se presentan en el lado del cliente dentro del mecanismo de los *Elementos del Modelo* de EJS, que nos permiten incorporar librerías Java en simulaciones de EJS de un modo sencillo, proporcionando así una interfaz gráfica que ayuda al desarrollador con la configuración y el uso de la librería.

El elemento *LabVIEW Connector* permite configurar una conexión con un VI de LabVIEW, para poder enlazar variables de EJS con los controles e indicadores del VI, así como para controlar la ejecución del VI. Por otra parte, el elemento *Audio Player* permite reproducir un *streaming* de audio para mejorar la sensación de realismo del laboratorio. Una característica importante de los elementos es que reducen la posibilidad de introducir errores en el código, reduciendo el tiempo y esfuerzo necesario en la fase de desarrollo.

Los resultados de la investigación se han plasmado en los siguientes componentes software:

- Una librería de clases Java y elementos de EJS, la *Process Control Library (PCL)*, para construir simulaciones relacionadas con el control de procesos. La *PCL* tiene como objetivo el facilitar el desarrollo de este tipo de simulaciones, inspirada en herramientas ampliamente extendidas como SIMULINK, y captura el comportamiento de los tipos de sistemas más comunes. Así, para construir una nueva simulación no es necesario empezar de cero, sino que es posible seleccionar e interconectar diferentes bloques, con lo que el esfuerzo de desarrollo se ve drásticamente reducido.

- El elemento de EJS *Audio Player*, que añade a EJS la capacidad de reproducir el sonido transmitido por una *webcam* o cámara IP a través de la conexión de red.

- El elemento de EJS *LabVIEW Connector*, que permite conectar, con sólo algunos clicks, una aplicación de EJS con un VI de LabVIEW.

- La implementación en MATLAB del algoritmo presentado en el Capítulo 2, para encontrar las propiedades de los ciclos límite en un sistema LTI controlado por una de las estructuras propuestas.

- El VI de LabVIEW *SOD Sampler*, que implementa el muestreo *send-on-delta* discutido en esta tesis.

Además, el nuevo paradigma de laboratorios remotos para la enseñanza del control de sistemas y los componentes de software presentados en esta tesis se han utilizado para desarrollar los siguientes laboratorios virtuales y/o remotos que permiten el estudio de los ciclos límite:

- Un laboratorio remoto basado en una planta de cuatro tanques (UNED, UNIBS). Una de sus principales características es situar el controlador en el lado del cliente, y la planta en el lado del servidor. Esta arquitectura permite explorar experimentalmente las propiedades de los controladores basados en eventos. La plataforma se ha usado para demostrar que los ciclos límite estudiados pueden aparecer en sistemas reales. En particular, los experimentos han permitido reproducir una amplia gama de estos ciclos límite, tal y como predice la teoría discutida en el Capítulo 2. Aunque el algoritmo se ha usado para un modelo de primer orden, éste es más general. Puede ser aplicado a un sistema LTI controlado por un controlador lineal genérico. El efecto del muestreo en el sistema puede ser analizado considerando una leve modificación en las matrices correspondientes a la representación del sistema en el espacio de estados.

- Una plataforma basada en un brazo flexible (UNED). La version virtual implementa una reproducción en 3D de la planta real, que proporciona una animación visualmente atractiva y que añade realismo a la simulación. Por otro lado, el laboratorio remoto usa realidad aumentada para mejorar la experiencia del estudiante con el laboratorio, mostrando información adicional superpuesta al vídeo obtenido de la planta. La motivación para el uso de esta planta es su mayor grado de complejidad con respecto a la de cuatro tanques. Por ejemplo, un problema en el canal de comunicaciones probablemente lleva a la planta a una desestabilización, mientras que en el caso anterior, en general provocaría solamente un peor rendimiento del control.

La implementación de ambos laboratorios se ha realizado utilizando una arquitectura que ha demostrado ser adecuada para laboratorios remotos. Se basa en el uso de EJS, JIL server, y LabVIEW. Esta plataforma permite que la implementación del controlador se encuentre físicamente separada de la planta, comunicándose a través de una conexión de red.

Además, los componenentes software desarrollados se han usado en la implementación o mejora de otras plataformas remotas:

- Una plataforma para enseñar conceptos de máquinas eléctricas (Universidad de Huelva, UHU). En particular, el elemento *Audio Player* añade la capacidad de obtener audio del laboratorio remoto, lo que es importante para ayudar a los alumnos a tener una experiencia más cercana a la planta, ya que, por razones de construcción, el movimiento del motor y el generador no se perciben visualmente de forma clara.

- Un laboratorio virtual para enseñar teoría de identificación de sistemas y control PI con un horno de gasoil (Arizona State University, ASU). En este caso, la librería *PCE* proporciona la implementación del controlador PI con un filtro de primer orden en cascada.

- Una plataforma para realizar prácticas de la asignatura *Robots Autónomos* (UNED), basada en *LEGO Mindstorm*.

A.7 Líneas de trabajo

Hay dos líneas de trabajo futuro. Por un lado, en lo que respecta a los resultados analíticos:

- El algoritmo para encontrar ciclos límite utiliza un enfoque basado en el dominio del tiempo, pero también es posible recurrir a un enfoque en el dominio de la frecuencia, como la función descriptiva, o una combinación de los dos paradigmas para intentar encontrar una solución cerrada y general para sistemas y muestreadores SOD de orden n.

- Además del punto anterior, es interesante explorar la posibilidad de paralelizar el algoritmo para incrementar la eficiencia en la búsqueda de soluciones. De hecho, se han llevado a cabo algunas pruebas preliminares, en un cluster de computación de cinco nodos con MATLAB, que muestra una reducción significativa en el tiempo de cómputo.

y, por otro lado, en lo que respecta a los resultados experimentales:

- Aunque se ha definido e implementado la arquitectura de la librería, así como los bloques más comúnmente usados, existen sistemas más complejos que pueden añadirse aún, para cubrir un rango mayor de sistemas. Por otro lado, aunque la implementación de los modelos de simulación se ha simplificado, proporcionando componentes que pueden ser combinados e interconectados, todavía es necesarios escribir el código de interconexión a mano. Sería deseable contar con una herramienta gráfica que permita la definición de diagramas de bloques.

- El uso de la librería se ha presentado en un contexto de simulación, y también en sistemas hardware-in-the-loop haciendo uso del elemento *LabVIEW connector* para EJS, pero también es posible combinarla con el soporte de tiempo real de EJS y plataformas open hardware, como Arduino, Phidget, Gnublin, etc. Actualmente, se está considerando la posibilidad de desarrollar un laboratorio remoto basado en una tarjeta *BeagleBone Black*. Ésta es una plataforma de desarrollo de bajo coste con un sistema operativo *Linux* y características E/S avanzadas, con entradas y salidas analógicas y digitales, salidas *PWM*, buses *SPI* y *I2C*, y otras capacidades que la hacen adecuada para construir plataformas experimentales.

- Las mismas ideas usadas en el elemento *LabVIEW Connector* pueden extrapolarse a otras herramientas de ingeniería, para simplificar la interconexión con MATLAB, Octave, etc. Aunque actualmente es posible realizar la interconexión para muchas de las herramientas de uso común, el objetivo a corto plazo es alcanzar el mismo nivel de interconectividad y portabilidad de plataforma (Linux, Mac OS, y Windows) del que actualmente se dispone con LabVIEW.