Developing Networked Control Labs: A Matlab and Easy Java Simulations Approach

Gonzalo Farias, Robin De Keyser, Sebastián Dormido, and Francisco Esquembre

***Abstract*—The new information technologies provide great opportunities in control education. One of them is the use of remote control labs to teach the behavior of control systems through a network. In this paper, a new approach to create interactive networked control labs is described. Two main software tools are used: Matlab and Easy Java Simulations. The first one is a widely used tool in the control community, whereas the second one is an authoring tool, designed to build interactive applications in Java without special programming skills. The remote labs created by this approach give to students the opportunity to face the effects of network delays on the controlled system and also to specify on the fly their own control algorithm.**

***Index Terms*—Control education, Easy Java Simulations (EJS), Matlab, networked control labs, virtual labs.**

I. Introduction

CONTROL education has to adapt to the new scenario that the information technologies provide [1], [2]. In this context, interactive remote labs take advantage of these new possibilities to improve the understanding of the system behavior [3], [4]. Interactivity is a crucial aspect when designing software applications that are to be used in the field of control engineering for pedagogical purposes. This interactivity should allow the student to simultaneously visualize the response of the system on the fly to any change, such as a step or a new PID configuration, introduced by the user. This immediate observation of the change in the system output, as response to user interaction, is what really helps the student to get useful practical insight into control system theory [5].

This paper is focused on providing new tools to teachers that want to use networked control labs as part of the educational process in the course on industrial electronic and/or control engineering. A networked control system is a remote lab wherein the control loops are closed through a network [6], [7]. The location of the controller at the server side, i.e., next to the real plant, is a widely used architecture for the design of remote control labs (see, for instance, [4] and [8]). Our approach uses an alternative architecture, considering the controller at the client side. The insertion of the communication network in the feedback control loop makes the analysis and design of these systems complex. Network delays in control loops can impose severe degradation on system performance. These delays have interesting consequences from the control education point of view, and they will be analyzed for a particular plant.

The approach presented here considers mainly two software tools: Matlab [9] and Easy Java Simulations (EJS) [10], [11]. Matlab is widely used software in the control community, being one of the reasons why we decided to use this tool to control a real plant. On the other hand, EJS is an authoring tool which is designed to create interactive applications in Java without special programming skills. By using these tools, teachers can build networked control labs with a high degree of interactivity and visualization.

Commonly, remote labs come with predefined controllers that can only be tuned or mixed by end users in order to produce a customized controller (see, for example, [4], [6], and [8]). However, and taking advantage that Matlab script is an interpreted language, our approach enhances the customization, also allowing students to define on the fly their own control algorithm, which provides great flexibility for the designing of many control strategies.

This paper is organized as follows. In Section II, the connection between EJS and Matlab is presented. Section III describes the required process to build a networked control lab. A real implementation of a lab is described in Section IV. Finally, some conclusions of the work are presented.

II. Linking EJS and Matlab

In this section, the connection between EJS and Matlab will be described in detail.

A. EJS

EJS is a free software tool for rapid creation of applications in Java with high-level graphical capabilities and with an increased degree of interactivity. The applications created by EJS can be standalone Java applications or applets, and for simplicity, we call them here EJS applications or simply applications. The source files of the EJS applications are saved in a customized *xml* format. EJS is different from most other authoring tools in that EJS is not designed to make life easier for professional programmers but has been conceived for science students and teachers.

Manuscript received December 28, 2008; revised May 28, 2009 and July 31, 2009; accepted December 23, 2009. Date of publication February 8, 2010; date of current version September 10, 2010.

G. Farias and S. Dormido are with the Department of Computer Science and Automatic Control, Universidad Nacional de Educación a Distancia, 28040 Madrid, Spain (e-mail: gfarias@bec.uned.es; sdormido@dia.uned.es).

R. De Keyser is with the Department of Electrical energy, Systems and Automation, Ghent University, 9000 Ghent, Belgium (e-mail: rdk@autoctrl.UGent.be).

F. Esquembre is with the Department of Mathematics, University of Murcia, 30100 Murcia, Spain (e-mail: fem@um.es).

Color versions of one or more of the figures in this paper are available online at http://ieeexplore.ieee.org.

Digital Object Identifier 10.1109/TIE.2010.2041130

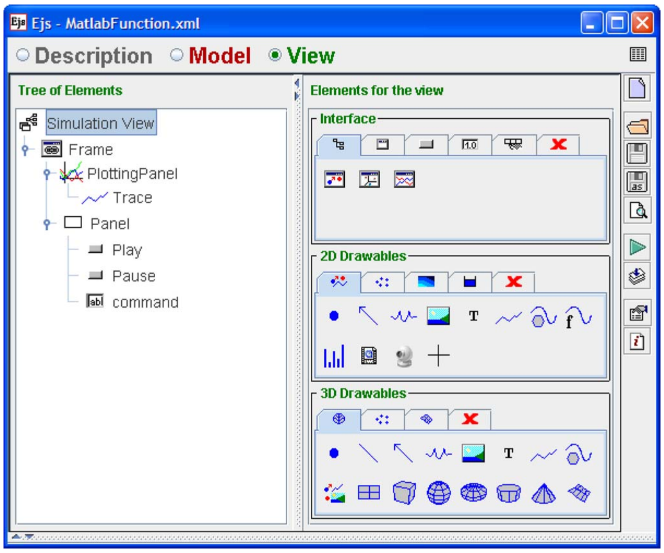


Fig. 1. GUI of the view section of EJS.

EJS structures the application in two main parts, the model and the view (see Fig. 1). The model can be described by means of pages of Java code and ordinary differential equations or by connecting to external applications (such as Matlab). The view provides the visualization of the application and also the user interface elements required for user interaction. These view elements can be chosen from a set of predefined components to build a treelike structure. Model and view can be easily interconnected so that any change in the model state is automatically reflected by the view, and vice versa.

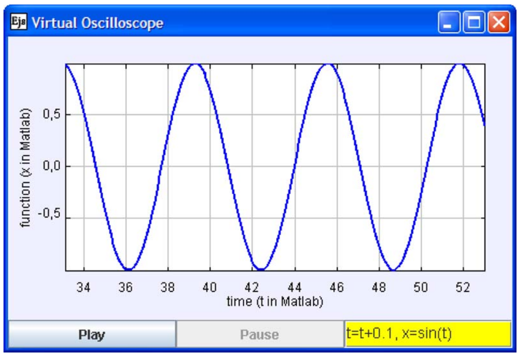
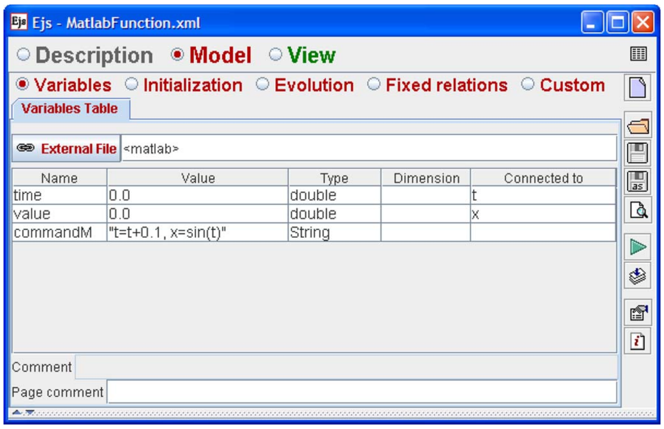


Fig. 2. GUI of a simple virtual oscilloscope which uses Matlab to evaluate a function entered by user.



External File <matlab>

Name	Value	Type	Dimension	Connected to
time	0.0	double		t
value	0.0	double		x
commandM	"t=t+0.1, x=sin(t)"	String		

Comment
Page comment

Fig. 3. Defining Matlab as external application in EJS. Notice that the EJS variables *time* and *value* are connected to the Matlab variables t and x, respectively.

B. Using Matlab as an External Application in EJS

EJS can set a link with external applications like Scilab, Sysquake, and Matlab/Simulink [12]. The connection with Matlab is quite interesting because Matlab is a well-known technical software tool in the engineering community. This link uses the JMatlink library [13] and the Engine library to control Matlab from EJS applications.

Authors can create virtual or remote labs (i.e., pedagogical applications) with a high level of interactivity and visualization using the EJS–Matlab link. In this case, Matlab is used for computing and processing data by means of the toolboxes, whereas EJS provides the graphical support and the user interaction of the Java application. The procedure to connect the application and Matlab can be summarized in four steps.

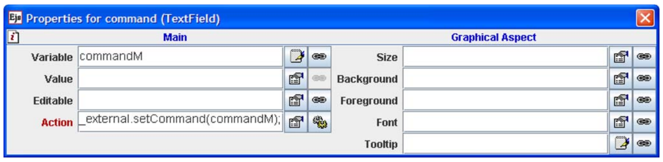
1) Select Matlab as an external application in EJS.
2) Create and connect the EJS variables with Matlab.
3) Control the execution of Matlab code.
4) Define the view and the user interaction.

A simple example will be used to explain in more detail the link between Matlab and the Java application created by EJS. The example is a virtual oscilloscope which shows a plot of a function versus time. The idea is that end users enter a command which will be evaluated in Matlab, and the application depicts in a plot (see Fig. 2) the values of two Matlab variables (x and t). As the continuous iteration of the command changes the values of the Matlab variables, the virtual oscilloscope plots the corresponding function.

The required steps to build the virtual oscilloscope in EJS will be presented in detail in the following.

Steps 1 and 2) To select Matlab as an external application, the authors simply need to introduce the string $\langle matlab \rangle$ in the text field *External File* in the subsection *Variables* of the *Model* (see Fig. 3). After that, authors have to create and connect EJS variables with Matlab variables using the column *connected to*. In this case, *time* and *value* are EJS variables which are connected to Matlab variables t and x, respectively. There are also built-in functions in EJS that directly read or write variables in the Matlab workspace.

Step 3) To control the execution of Matlab code from the Java application, EJS provides different built-in functions. One of them is *_external.step()*, which allows the execution of a specific command. Normally, this built-in function is located in the subsection *Evolution* to execute the code in an iterative way. The command has to be previously defined using the built-in function *_external.setCommand(commandM)*. Notice that this function needs a string variable (in this case, *commandM*) whose value is the command that will be executed by *_external.step()*. In the example (see Fig. 3), the command to execute has the following initial value: $t = t + 0.1$ and $x = \sin(t)$.


Properties for command (TextField)

Main		Graphical Aspect	
Variable	commandM	Size	
Value		Background	
Editable		Foreground	
Action	_external.setCommand(commandM);	Font	
		Tooltip	

Fig. 4. Properties of the text field used by end users to enter a function. Notice that the function will be evaluated in Matlab when a user presses the return key (property *Action*) in the text field.

Step 4) The initial command can be modified by end users using the text field of the view, which is highlighted in Fig. 2. To add this interaction, the text field is connected to the variable *commandM* and to the built-in function *_external.setCommand(commandM)* using the properties of the text field (see Fig. 4). This interaction means that, when end users modify the value of the string in the text field, the value of the variable *commandM* will also be modified, and then, the new command will be executed by the built-in function *_external.step()*. Thus, finally, the plot will show the graphic of the modified function.

EJS also has built-in functions that allow interacting at low level with Matlab. The three most important functions are as follows.

1) *_external.eval(String)*: A method to execute a string variable in Matlab (similar to function *eval* of Matlab).
2) *_external.getDouble ("MatlabVariable")*: A method that returns the value of a Matlab variable.
3) *_external.setValue ("MatlabVariable," Value)*: A method that modifies the value of a Matlab variable.

In [11], [12], and [14], a more detailed description is presented about the connection between EJS and Matlab.

C. Jim Server

The previous link between Matlab and the Java application is direct if both software are in the same computer. However, if end users do not have Matlab in their computers, they are not able to execute the Java application which uses the EJS–Matlab link. For this reason, when Matlab and the application are located in different computers, we need to use Jim server. This free software tool, written in Java [14], [15], allows for an EJS application to use a remote Matlab server. Therefore, using Jim, end users can execute applications which use Matlab, even though users do not have Matlab in their computers. In order to set the extended link, Jim has to be located in the remote computer where Matlab is installed. In addition, the application has to be modified to indicate the remote service.

The remote connection uses a network protocol to support the exchange of information between Jim and the EJS application. The scheme of the connection between EJS applications, Jim, and Matlab is shown in Fig. 5. At the client side, there is an EJS application connected to Jim server by TCP/IP protocol. At the server side, Matlab is controlled by Jim server in order to response to the requests of the EJS application. Similar to the local link case, the library JMatlink is used to control Matlab from Jim server. Normally, a Java application that uses this scheme does most of the computation at the server side (i.e., by Matlab), whereas the interface is used to show the results and to support the user interaction.

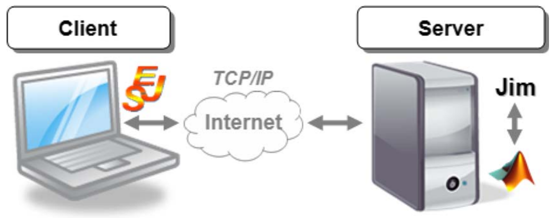
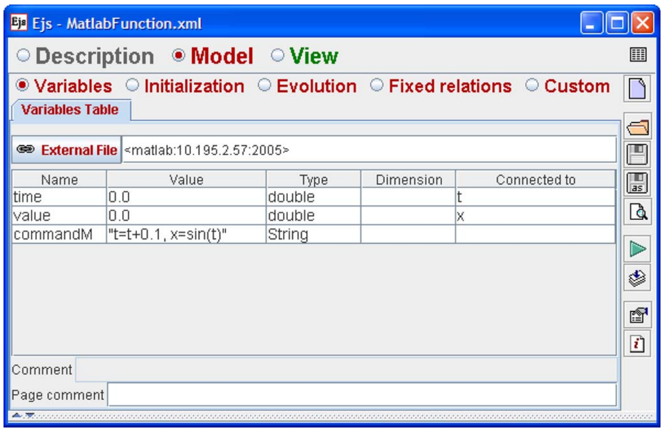


Fig. 5. Scheme of the remote Matlab–EJS connection.


Ejs - MatlabFunction.xml

Description [✓ Model] View

[✓ Variables] Initialization Evolution Fixed relations Custom

Variables Table

External File <matlab:10.195.2.57:2005>

Name	Value	Type	Dimension	Connected to
time	0.0	double		t
value	0.0	double		x
commandM	"t=t+0.1, x=sin(t)"	String		

Comment

Page comment

Fig. 6. Defining remote Matlab as external application in EJS.

D. Types of Links in Jim

There are two kinds of link between Jim and EJS applications, i.e., **synchronous** and **asynchronous**. The synchronous link is quite similar to a local connection between an EJS application and Matlab, i.e., the execution of a command is done in a coordinated way. This implies that the application decides when Matlab executes a command. The performance of a synchronous link is good when the network delays are negligible, as in a local area network. However, in a wide area network, e.g., Internet, the time required to exchange information between client and server could become much longer. This is where the asynchronous link comes in handy, because this link has been developed to minimize the effect of the network delays. In this case, the application does not continuously keep coordination with the remote Matlab, and therefore, the command execution in a remote Matlab is indirectly controlled by EJS application.

One of the advantages of the synchronous link is that, when an author wants to transform an EJS application that uses local Matlab to use a remote Matlab, the required changes are minimal. For instance, if we decide to transform the virtual oscilloscope, in order to use a synchronous link with a remote Matlab, the only one requirement is to modify the source file to enter in the text field *External File* the IP address (e.g., 10.195.2.57) and the Port (e.g., 2005) of the server. This change is shown in Fig. 6. After that, end users are enabled to use the virtual oscilloscope through a network.

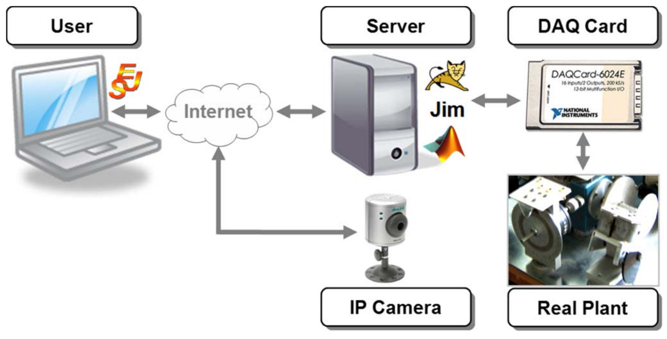


Fig. 7. Elements required for a networked control lab implemented with Matlab–EJS approach.

The case of the asynchronous link is slightly different because, in addition to the previous changes, authors also need to use the built-in function *_external.synchronize()*. Normally, this built-in function has to be executed when end users interact with the view of the application (e.g., moving sliders).

The choice between both links depends on the kind of application that the author wants and also the context where the application is going to be used. From the remote control lab point of view, an asynchronous link could be useful if the controller is located at the server side, which means that the lab is used mainly to monitor a remote plant. Otherwise, if the controller is located at the client side, the synchronous link has to be used. For this reason, the approach of this work uses the synchronous link for the networked control labs. More details about the Jim server can be found in [14].

III. Building Networked Control Labs

In this section, the main tools and actions required to create a networked control lab will be presented. First, the software and hardware issues will be discussed. Later on, the communication process with the plant will be detailed. Finally, the core control code of the lab will be presented.

A. Software and Hardware Required

The approach presented in this work can be summarized in Fig. 7. As shown in this figure, there are other tools in order to put online the control lab for students. Among others, we need at least a Web server, an IP camera, and a data acquisition card (DAQ card).

The Web server is required if authors want to put the remote labs as applets. Moreover, a Web server is needed if the teacher adds some virtual labs, simulations, or documentation in the Web site. A good option to provide this service could be Apache Tomcat. This Web server also offers security features called authorization, authentication, and access control in order to allow only access to authorized students.

To show students a view of the real plant, a simple Web camera can be used. However, it is preferable to use an IP camera since it has a built-in Web server that can stream video images directly to the Internet. Moreover, EJS has a specific visual element to display the video from stream servers, so the access to the streams of IP cameras is quite direct and simple to use in EJS.

There are many options when developers require controlling external hardware (plants) from computers. In our case, different DAQ cards could be used; the only one restriction in this approach is that the selected card has to be compatible with Matlab. The communication process with the DAQ card can be done using the Data Acquisition Toolbox. This toolbox is a collection of M-file functions and Dynamic Link Library that enable users to interface with specific hardware. The toolbox provides users with the following main features:

1) a framework for bringing live measured data into the MATLAB workspace using PC-compatible plug-in data acquisition hardware;
2) support for analog input (AI), analog output (AO), and digital I/O subsystems, including simultaneous analog I/O conversions.

For further information about the toolbox, see [16].

B. Acquisition Process From DAQ Cards

A typical code to initiate the data acquisition process using the toolbox of Matlab is shown in code listing 1. In this case, there are two input channels and also two output channels. The input channels are created to read the position and the speed of a servo motor. One output channel is used to send out the command signal to the motor, and the other one is used to feed the position sensor (a potentiometer). After the channels have been created, it is necessary to configure the DAQ card to continuously acquire the data.

Listing 1. Initiate data acquisition process
```
%Create input channels
 AI = analoginput ("nidaq," "1");
 Speed = addchannel (AI,1);
 Position = addchannel (AI,0);
 set (AI, "InputType," "SingleEnded");
 AI.Channel.InputRange = [−10 10];
%Create output channels
 AO = analogoutput ("nidaq," "1");
 VoltOutSpeed = addchannel (AO, 0);
 VoltOutPosition = addchannel (AO,1);
%Configurate sampling and trigger
 set ([AI, AO], "SampleRate," 1/0.05);
 set (AI, "TriggerType," "Immediate");
 set (AO, "TriggerType," "Immediate");
%Acquire data continuously
 set (AI, "SamplesPerTrigger," inf);
%Send out initial volts
 putdata (AO, [0, 1]);
 start ([AO AI]);
```

After the initialization has been done, the channels can be accessed from the EJS application (by means of the EJS–Matlab link). The code listing 2 shows the Java code used to read and write the input/output channels. Notice that the code required to interface the DAQ card is sent to Matlab from the application using the built-in function *_external.eval(String)*.

Listing 2. Read and Write the channels from EJS
```
public double [] getData (){
 double [] data;
 //Read input channels
 _external.eval (“s = [toc, peekdata(AI, 1)];”);
 data = _external.getDoubleArray (“s”);
 return (_data);
}
public double sendOut (double cmd){
 //Saturation of command signal
 cmd = commandLimits (cmd);
 //Write output channels
 _external.eval(“putdata (AO, [“+cmd+,” 1])”);
 _external.eval (“start(AO)”);
 return (cmd);
}
```

C. Control Loop of the Lab

The main loop of the networked control lab is displayed in code listing 3. The code presented is part of the EJS application and has to be located in an *Evolution* page of the *Model* in order to execute the code continuously. The code has three main parts: open loop, closed loop, and getting data. In the open-loop mode, the input, given by the user (*commandUser*), is directly sent out to the real plant. In the closed-loop mode, the control signal is computed using the function *controller*, and then, the signal is sent out to the motor. Finally, after the open- or closed-loop stages have been executed, the data are read from the sensors of the plant. Notice that the functions for reading and writing the channels (*getData* and *sendOut*) have been defined earlier in listing 2.

Listing 3. Main loop of the remote lab in the EJS application
```
//Open Loop
if (isOpenLoop) {
 commandUser = sendOut (commandUser);
 inputApplied = commandUser;
//Closed Loop
} else {
 if (speedControl) cs = controller (reference,speed);
 else cs = controller (reference,position);
 controlSignal = sendOut (cs);
 inputApplied = controlSignal;
}
//Getting data
data = getData ();
position = data [2];
speed = data [1];
time = data [0];
```

IV. Networked Control Lab Implemented

The lab created is a Java application designed to control a simple servo motor through the Internet. This lab uses a real servo motor of FEEDBACK located at Ghent University in Belgium. The selected DAQ card was the 6024E card manufactured by National Instruments. In Fig. 8, a scheme of the implemented lab is presented. Notice that the continuous-time system is controlled by a discrete-time controller with a sampling period of 50 ms. The networked control lab is divided in three sections: the client side, the network, and the server side.

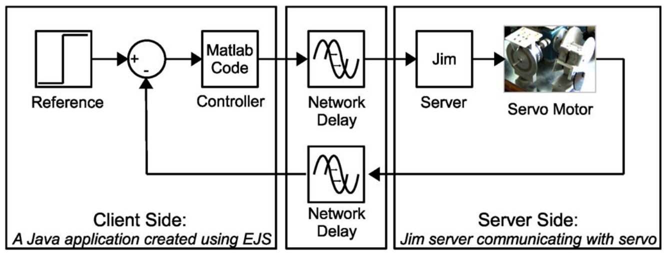


Fig. 8. Scheme of the implemented remote lab.

The implemented lab uses two Matlab sessions, one located at client side to compute the control signal and another one located at the server side to interface the servo motor through the DAQ card 6024E. At each sampling time, the Jim server collects data from the plant by using the Matlab engine at the server side. Then, the Jim server sends data to the EJS application by using standard TCP/IP Java methods. The application uses the data and the local Matlab engine to compute the control signal by means of executing a Matlab script (which can be modified online by the student). Then, the application retrieves the value of the control signal from the local Matlab engine and sends it via TCP/IP to the Jim server. Finally, the Jim server sends out the computed control action to the servo motor by means of the server Matlab engine.

A. Computing the Control Signal

As it was said before, the lab uses a local connection with Matlab to evaluate the control action. This means that students can write on line the Matlab code to control the plant. The code used by default (a PID controller) is shown in listing 4. Here, a Matlab function named $computeControl$ will be called from the $controller$ function of listing 3.

The function $computeControl$ has five input variables and one output variable. The input variables are r (set point or reference), y (controlled variable for position or speed of motor), and three optional variables a, b, and c (in this case, these free variables are used as gain, integral time, and derivative time of the PID controller). The output variable cs is returned to the $controller$ function in order to send out the control action to the real plant (see listing 3).

Listing 4. Default code to calculate a PID controller
```
function cs = computeControl (r, y, a, b, c)
 %r = reference y = output [a b c] = EJSparameters
 %cs = controlsignal
 %Initialization
 persistent Iold Dold yold
 if isempty (Iold)
  Iold = 0; Dold = 0; yold = 0;
 end
 %Calculate Control Signal
 beta = 1; N = 10; h = 0.1;
 P = a * (beta * r - y);
```

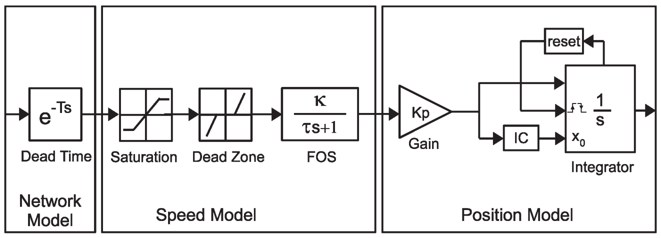


Fig. 9. Model of the networked control lab. The remote servo motor is divided into three parts: the network model, the speed model, and the position model.

```
    I = Iold;
    D = c/(N * h + c) * Dold + N * a * c/(N * h + c) *
(yold − y);
    cs = P + I + D;
    Iold = Iold + a * h/b * (r − y);
    Dold = D;
    yold = y;
```

B. Model of the Remote Servo Motor

From the control point of view, the model of the networked control lab can be divided in three main parts: a model for the network delay, and models for the speed and position of the servo motor (see Fig. 9).

We can assume that all delays from the network can be represented by a single variable dead time, which can have different values depending mainly on the distance between server and client and the traffic overhead in the network. The speed model is a first-order system (FOS) plus two nonlinearities: a saturation (limiter) and a dead zone. The FOS system represents the dynamics of the amplifier, the motor, the brake, and the tachometer of the servo motor plant. Regarding the dead zone, the range detected in the real plant is located in $[-\delta_1, \delta_1]$, where $\delta_1 = 0.1$ V. In the case of the saturation of the amplifier, it allows input values into the range $[-\delta_2, \delta_2]$, where $\delta_2 = 0.2$ V.

To get the position of the servo motor, a potentiometer was used. The potentiometer is fed by the DAQ card with 1 V. Thus, the voltage obtained indicates the position of the motor. This position model can be represented by a gain plus a pure integrator. However, in order to have a better model of the real plant behavior, an automatic reset can be added to the pure integrator to limit and handle the discontinuity of the servo position sensor, which produces a signal between 0 and 1 V to represent the angular position from 0° to 360°.

The motor dynamics can be modified by means of the position of the magnetic brake. Normally, two positions are used: *partial brake* and *full brake*. For instance, when *full brake* is selected, then the gain κ and the time constant τ are smaller than that when the *partial brake* is used. In the *full brake* case, typical values for κ and τ are about 10 and 1 s, respectively. On the other hand, in the *partial brake* case, normally, the values for κ and τ are about 14 and 2.5 s, respectively.

C. Control Aspects of the Remote Servo Motor

There are two control objectives regarding the implemented lab configuration: speed control and position control. However, in this work, only the position control will be analyzed further, since it is more difficult than speed control. In a feedback control of the remote servo, the model of the motor can be considered as a second-order system with a dead time, a dead zone, and a saturation. The effect of each part of the model will be discussed next.

1) Dead Zone: Consider a second-order system composed only by the FOS model plus a gain (k_p) and a pure integrator. If a P controller (k_c) is also considered, then the open-loop transfer function of this simple model is given by the following equation, where $K = k_c \cdot \kappa \cdot k_p$:

$$GH = \frac{K}{s(\tau s + 1)}. \tag{1}$$

From control point of view, this second-order system can be controlled using only the P controller. However, if the dead zone is taken into account [see Fig. 10(a)], then the control requires an integral action to eliminate the steady-state error. This fact is demonstrated analyzing the obtained error when the control signal $(cs = k_c \cdot e)$ is inside the dead zone $(-\delta_1 \le cs \le \delta_1)$. In this situation, the output of the dead-zone block is zero, so the control loop is broken, and therefore, the position signal is just a constant. It can be demonstrated that, in this case, the steady-state error (e_{ss}) is different from zero, and it is located in a band [see (2)]. The extremes of this band depend on the dead-zone limits and the gain of the proportional controller

$$\frac{-\delta_1}{k_c} \le e_{\mathrm{ss}} \le \frac{\delta_1}{k_c}. \tag{2}$$

2) Dead Zone and Saturation: Consider the previous case plus the saturation. Here, if the saturation is also taken into account, then the integral action required to eliminate the steady-state error has to consider the limits of the saturation block. Thus, if a PI controller is selected, then an antiwindup scheme should be applied to avoid a slow control action [17].

3) Dead Time: If only the linear parts of the motor and the dead time are considered [see Fig. 10(b)], then we get a second-order system plus dead time. A detailed description about dead-time processes can be found in [18]. In this situation, the network delay can affect strongly the control performance. Here, different solutions can be applied; one of them could be the use of a Smith predictor if the delay remains approximately constant during the remote control session. However, if the delay connection is highly variable, then more elaborated algorithms should be required. To simplify the problem, here, we will focus on fixed time-delay systems; further analysis about varying time delay can be found in [18]–[20].

If the network delay is approximately constant and we use a P controller, it is possible to calculate how much delay that the system can tolerate before becoming unstable [21]. To calculate this margin delay first, we compute the open-loop transfer function of the model, which, in this case, is given by (1).

Second, computing the magnitude and the phase margin of the transfer function given by (1) and assuming that $(1/\sqrt{\tau K}) \approx 0$ (which is a common situation when partial or full brake is selected), we can get useful approximations for

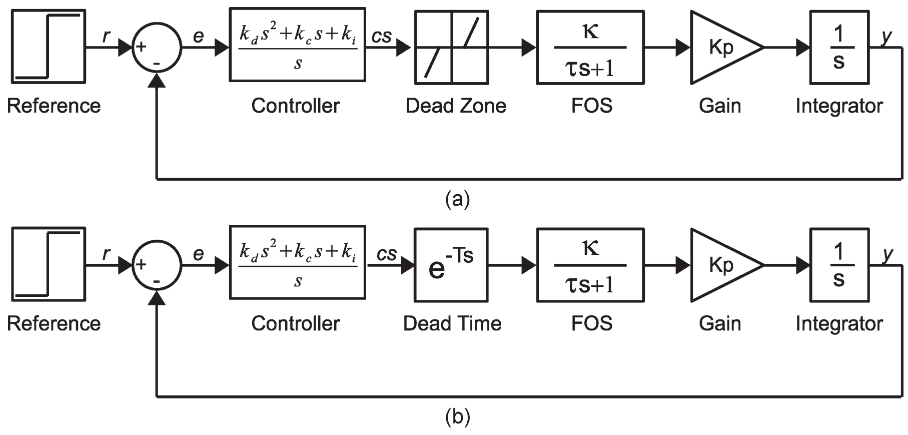


Fig. 10. Remote servo (a) as a second-order system plus dead zone and (b) as a second-order system plus dead time. Note that the block "Controller" represents a PID controller, which can be computed in the remote lab by the Matlab function *computeControl* described in listing 4.

ω_{gc} (gain cross over frequency) and ϕ_m (phase margin). These approximations are given by

$$20 \log \frac{K}{\omega_{\mathrm{gc}}\sqrt{\omega_{\mathrm{gc}}^2\tau^2+1}} = 0 \rightarrow \omega_{\mathrm{gc}} \approx \sqrt{\frac{K}{\tau}} \qquad (3)$$

$$\phi_m = \pi - \frac{\pi}{2} - \arctan \omega_{\mathrm{gc}}\tau \rightarrow \phi_m \approx \frac{1}{\sqrt{K\tau}}. \qquad (4)$$

Finally, by using ω_{gc} and ϕ_m, the delay margin (T) can be calculated using (5). Therefore, this last equation tells us that the delay margin is inversely proportional to the product of $k_c \cdot \kappa \cdot k_p$

$$\phi_m = \omega_{\mathrm{gc}}T \rightarrow T \approx \frac{1}{k_c \kappa k_p}. \qquad (5)$$

Notice that even small delays could lead to an unstable closed loop. For example, when the full brake is selected ($\kappa = 10$ and $\tau = 1$) and assuming that $k_p = k_c = 1$, the delay margin is approximately 0.1 s, which is a usual value for Internet delays. This result implies that if the system has less brake, then it is less stable. Obviously, this delay margin can be modified changing the value of k_c in the P controller.

4) Dead Time, Saturation, and Dead Zone: Consider the linear model of the motor plus the dead time, saturation, and the dead zone controlled by a P controller. In this situation, the system could produce limit cycles if there is enough dead time. To effect an approximate limit cycle study of this nonlinear system, the nonlinear elements can be characterized by their describing functions $(N(A,\omega))$ and the linear part by its frequency response function $(GH(j\omega))$. Then, the solution of (6) yields the amplitudes and frequencies of the limit cycle [22]

$$1 + N(A,\omega)GH(j\omega) = 0. \qquad (6)$$

In our case, the linear part is given by (1), and the dead zone, combined with the saturation, has the describing function shown in (7), where A is the amplitude of the limit cycle after the P controller and δ_1 and δ_2 are the limits of the dead zone and saturation, respectively. The function $f(\gamma)$ is called *saturation function*, and its values are given by (8)

$$N(A) = f\left(\frac{\delta_2}{A}\right) - f\left(\frac{\delta_1}{A}\right) \qquad (7)$$

$$f(\gamma) = \begin{cases} -1, & \text{for } \gamma < -1 \\ \frac{2}{\pi}\left(\arcsin\gamma + \gamma\sqrt{1-\gamma^2}\right), & \text{for } |\gamma| \le 1 \\ 1, & \text{for } \gamma > 1. \end{cases} \qquad (8)$$

Solving (6) and assuming that $(1/\sqrt{\tau K N}) \approx 0$, we can find useful approximations for the frequency of the limit cycle (ω) and also for the delay margin (T). These approximations, given by the following equations, can be used for control design purposes:

$$\omega \approx \sqrt{\frac{KN}{\tau}} \quad T \approx \frac{1}{KN}. \qquad (9)$$

Interesting analysis can also be done observing the polar plot (see Fig. 11) of (6). In this figure, we can see that the limit cycles appear when there is an intersection of $GH(j\omega)$ and $-(1/N(A))$. Notice that the curve of $-(1/N(A))$ has been distorted to see both limit cycles. Hence, it is clear that there are no limit cycles if there is no dead time (curve $GH(j\omega)$). However, if there is enough delay (curve $GH_T(j\omega)$), there will be one or two limit cycles. In the case of two limit cycles, one is unstable, and the other one is stable. However, the effect of the unstable cycle limit is constrained by the stable cycle limit.

Observing the polar plot in Fig. 11, it is possible to conclude that at least one limit cycle will occur in the maximum value of $-(1/N(A))$ (i.e., when $A = \delta_2$). The minimum value of the delay required for this situation can be calculated using (9). For example, if $k_p = k_c = 1$, $\delta_1 = 0.1$, $\delta_2 = 0.2$, and the full brake is selected as the previous case, then the minimum delay is about $T \approx (1/10N(0.2)) \approx 0.26$ s.

5) Complete Model: In this case, we are considering the model shown in Fig. 9 controlled by a P controller. The analysis is quite similar to the previous case; the only one difference is the automatic reset mechanism of the integrator.

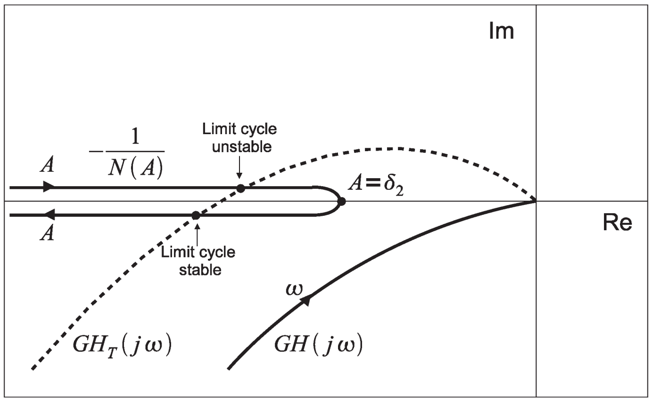


Fig. 11. Polar plot representation to determine the limit cycle conditions.

This mechanism allows us to emulate the nonlinearity of the position sensor (potentiometer), which keeps the position signal between zero and one. For this reason, the behavior of this model is similar to the previous case when the position signal is ranged from zero to one. Otherwise, if the signal crosses these limits, then the model could become unstable. Hence, the limit cycles analysis is useful to determine the delay (T) with which the system would have a limit cycle with an amplitude before the P controller greater than the position range. To do this, we have to use the approximation for time delay given by (9) when $A = 0.5k_c$. Thus, for instance, if the full brake is selected, $k_p = k_c = 1$, $\delta_1 = 0.1$, and $\delta_2 = 0.2$, then the maximum delay is about $T = 0.41$ s.

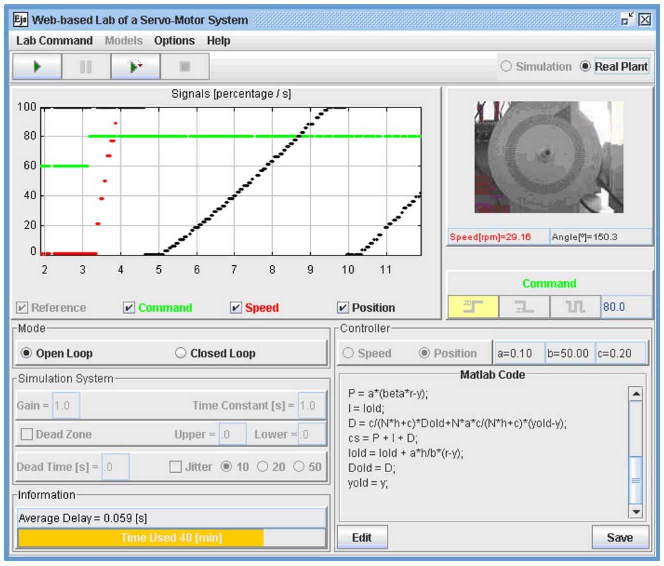
D. GUI of the Networked Control Lab

The many control aspects described earlier demand a graphical user interface (GUI) for the application which offers a high degree of flexibility. However, at the same time, the intended pedagogical use recommends keeping the interaction with the students relatively simple and intuitive. For this reason, we spent a considerable part of our implementation time to discuss and test the possible designs of the interface. The fast-prototyping facilities of EJS were very useful, because they allowed faculty with not much programming expertise to interact directly with the computer at the application design phase, testing the many different possibilities. The result is a modern-looking intuitive GUI that offers a clear view of the process and allows flexible user interaction without frightening the student with too many options.

The interface of the application, shown in Fig. 12, has four main panels, a menu bar, and a small task bar.

The two upper panels of the interface provide a quick view of the motor and a time plot of the signals from and to it. The time plots on the left panel display the speed and position signals of the motor, which are read from the Jim connection to the equipment, as well as the reference and the input to the plant (which is the command or the control signal depending on whether the mode is open or closed loop, respectively).

The top part of the right panel shows a real view of the actual motor at Ghent University, obtained by EJS from an IP camera pointing to it. Although the quality of the video thus obtained



Speed[rpm]=29.16 Angle[º]=150.3

[✓] Reference [✓] Command [✓] Speed [✓] Position

Command: 80.0

Mode: [✓ Open Loop] Closed Loop

Controller: Speed [✓ Position] a=0.10 b=50.00 c=0.20

Simulation System: Gain = 1.0 Time Constant [s] = 1.0; Dead Zone Upper = .0 Lower = .0; Dead Time [s] = .0 Jitter [✓ 10] 20 50

Information: Average Delay = 0.059 [s]; Time Used 48 [min]

Matlab Code

```
P = a*(beta*r-y);
I = Iold;
D = c/(N*h+c)*Dold+N*a*c/(N*h+c)*(yold-y);
cs = P + I + D;
Iold = Iold + a*h/b*(r-y);
Dold = D;
yold = y;
```

Fig. 12. GUI of the implemented remote lab.

depends on the speed of the connection to the Internet, the view of the actual equipment brings students a sense of reality. The view includes information of the speed and position of the servo. In addition, on this right panel, a series of buttons allows students to apply ready-made step up, step down, or square signal as input to the system. The text box allows entering a constant value as input to the plant. In open-loop mode, this input represents a command to the plant, while in closed loop, the input represents a reference to the controller.

The two lower panels are devoted to the control of the motor. The left bottom panel allows the student to select an open- or closed-loop mode of operation, as well as modification of the parameters of the controller in the closed-loop mode. An information bar at the bottom of this panel shows the total delay in each operation cycle. This delay includes the sampling period, the controller computing time, and the round trip communication time between the server and the controller. Finally, a time bar indicates the time elapsed by the current session.

The right bottom panel is used by the student to select if the signal to control is the speed or the position of the motor and to enter the Matlab code that will be used to compute the control signal. This code accepts three free parameters (a, b, and c) that can be used by the student in the control algorithm (see listing 4) and whose values can be modified using three text fields provided by the interface.

The interface is completed with a top task bar that provides buttons to start, pause, step, and stop the application and select whether we want to control the real equipment (if available) or the virtual simulation of the motor.

The components of the interface described earlier provide the basic functionality required to operate the application. A menu bar provides some additional features such as the possibility to specify the range of the axes of the plot, indicate if the values of the signals are displayed in volts or as percentages, send to the Windows clipboard the values of the signals for further analysis, and calibrate the location of the video of the IP camera.

E. Using and Evaluating the Networked Control Lab

From a wide set of tasks to do with the networked control lab, we asked our students to do the following activities.

1) Control the plant indicating suitable values for the controller (by default, a PID controller).
2) Apply a step to the motor input voltage.
3) Identify the main time constant and the gain from the "speed" response.
4) On the basis of this identification exercise, derive a model for the "position" response.
5) On the basis of the position model, design a controller transfer function for position control.
6) Apply the position controller to the motor, and evaluate the result.

This exercise was done in a basic course on control engineering in Ghent University, taken by some 130 students. At the end of the use of the networked control lab, the students were invited to evaluate it using an online poll (which allows students to vote in a confidential way). The results, from teacher and students point of views, were very good specially considering that it was our first experience. The online questionnaire (based on [8]) was rated for students as *strongly agree*, *agree*, *neutral*, *disagree*, and *strongly disagree*. The questions were divided into the following categories.

1) Learning value: The lab helps to learn the relevant contents.
2) Value added: The lab has advantages over traditional learning methods.
3) Usability: The lab was easy to use.
4) Technical functionality: The lab works well from software and hardware points of view.

The results indicate that *Technical Functionality* was not a problem for most of the students because only 11% of them evaluated this category as *disagree*. Regarding the *Usability*, about 84% of the students did not find some difficulty to use the lab. About 55% of the students think that the lab helps to understand the matters (*Learning Value*), while only 13% of them disagree. Regarding the *Value Added*, 30% of the students think that the lab has advantages over other learning methods; however, about 25% of the students think the opposite. Probably, this item indicates the value that students give to the traditional practice in the labs, which means that the remote or virtual labs have to be considered as a complement (and not a replacement) of the industrial electrical teaching.

V. Conclusion and Further Work

In this paper, a new approach to build interactive networked control labs has been presented. The work allows nonprogramming instructors to create innovative pedagogical tools, which can be used to motivate students to apply the theory of automatic control to new challenges, like the remote control of plants. This kind of labs can be used as part of a basic industrial electronic control course or even a benchmark for advanced students since the network delays and nonlinearities add interesting and challenging features from the control point of view.

Two main software tools have been used, Matlab and EJS. Matlab was selected because it is a very well known tool in the control engineering community, whereas EJS allows teachers to build complex Java applications with high levels of interactivity and visualization, but with minimum skills on computer programming.

Future work will mainly consist in adding new functionality and improvements to this approach. In this sense, in order to provide a better experience for users, large delays on far Internet connections should be mitigated until suitable values.

References

[1] B. S. Heck, "Special report: Future directions in control education," *IEEE Control Syst. Mag.*, vol. 19, no. 5, pp. 35–58, Oct. 1999.

[2] S. Dormido, "Control learning: Present and future," *Annu. Control Rev.*, vol. 28, no. 1, pp. 115–136, 2004.

[3] L. Gomes and S. Bogosyan, "Current trends in remote laboratories," *IEEE Trans. Ind. Electron.*, vol. 56, no. 12, pp. 4744–4756, Dec. 2009.

[4] L. Gomes, F. Coito, A. Costa, L. Palma, and P. Almeida, "Remote laboratories support within teaching and learning activities," in *Proc. Int. Conf. REV*, Jun. 25–27, 2007, pp. 1–6.

[5] J. Sánchez, S. Dormido, and F. Esquembre, "The learning of control concepts using interactive tools," *Comput. Appl. Eng. Educ.*, vol. 13, no. 1, pp. 84–98, 2005.

[6] C. Lazar and S. Carari, "A remote-control engineering laboratory," *IEEE Trans. Ind. Electron.*, vol. 55, no. 6, pp. 2368–2375, Jun. 2008.

[7] W. Zhang, M. S. Branicky, and S. M. Phillips, "Stability of networked control systems," *IEEE Control Syst. Mag.*, vol. 21, no. 1, pp. 84–99, Feb. 2001.

[8] 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 Trans. Educ.*, vol. 51, no. 1, pp. 35–44, Feb. 2008.

[9] Mathworks' Home Page. [Online]. Available: http://www.mathworks.com

[10] F. Esquembre, "Easy Java simulations: A software tool to create scientific simulations in Java," *Comput. Phys. Commun.*, vol. 156, no. 2, pp. 199–204, Jan. 2004.

[11] Easy Java Simulations' Home Page. [Online]. Available: http://fem.um.es/Ejs

[12] S. Dormido, F. Esquembre, G. Farias, and J. Sanchez, "Adding interactivity to existing Simulink models using Easy Java Simulations," in *Proc. 44th IEEE Conf. Decision Control*, Seville, Spain, 2005, pp. 4163–4168.

[13] S. Müller, JMatlink's Home Page. [Online]. Available: http://jmatlink.sourceforge.net/

[14] G. Farias, F. Esquembre, J. Sanchez, S. Dormido, H. Vargas, S. Dormido-Canto, R. Dormido, N. Duro, and M. Canto, "Desarrollo de Laboratorios Virtuales, Interactivos y Remotos Utilizando Easy Java Simulations y Modelos Simulink," in *Proc. 12th Latin-Amer. Congr. Autom. Control*, Salvador de Bahı´a, Brazil, 2006.

[15] JIM Server's Home Page. [Online]. Available: http://lab.dia.uned.es/rmatlab/

[16] The MathWorks, Inc., Data Acquisition Toolbox, User's Guide, v.2.5.

[17] K. J. Åström and T. Hägglund, *Advanced PID Control*. Research Triangle Park, NC: ISA, 2005.

[18] J. Normey-Rico and E. Camacho, *Control of Dead-Time Processes*. Berlin, Germany: Springer-Verlag, 2007.

[19] S. Cristea, C. de Prada, and R. De Keyser, "Predictive control of a process with variable dead-time," in *Proc. IFAC 16th World Congr.*, Prague, Czech Republic, 2005.

[20] G. Liu, Y. Xia, J. Chen, D. Rees, and W. Hu, "Networked predictive control of systems with random network delays in both forward and feedback channels," *IEEE Trans. Ind. Electron.*, vol. 54, no. 3, pp. 1282–1297, Jun. 2007.

[21] R. Dorf and R. Bishop, *Modern Control Systems,* 10th ed. Englewood Cliffs, NJ: Prentice-Hall, 2004.

[22] A. Gelb and W. Vander Velde, *Multiple-Input Describing Functions and Nonlinear System Design*. New York: McGraw-Hill, 1968.

Gonzalo Farias received the M.S. degree in computer science from the University de la Frontera, Temuco, Chile, in 2001.

Since 2005, he has been with the Department of Computer Science and Automatic Control, Universidad Nacional de Educación a Distancia, Madrid, Spain. His current research interests include control of dynamic system and virtual and remote labs.

Robin De Keyser received the M.Sc. degree in electromechanical engineering and the Ph.D. degree in control engineering from Ghent University, Ghent, Belgium, in 1974 and 1980, respectively.

He is currently a Full Professor of control engineering with the Faculty of Engineering, Ghent University. He was an External Review expert in several European Commission research programs and is one of the pioneers who produced the original concepts of "predictive control" during the 1980s. He is the author/coauthor of about 250 publications in journals, books, and conference proceedings. His current research interests include model predictive control, autotuning and adaptive control, modeling and simulation, system identification, and pilot implementations in technical and nontechnical systems, such as chemical, steel, marine, mechatronic, seminconductor, power electronics, and biomedical.

Sebastián Dormido received the B.S. degree in physics from Complutense University, Madrid, Spain, in 1968 and the Ph.D. degree in science from the University of the Basque Country, Bilbao, Spain, in 1971.

Since 1981, he has been a Professor of control engineering with the Universidad Nacional de Educación a Distancia, Madrid, Spain. His scientific activities include computer control of industrial processes, model-based predictive control, robust control, and model and simulations of continuous processes. He has authored or coauthored more than 150 technical papers in international journals and conferences. Since 2002, he has promoted academic and industrial relations as President of the Spanish Association of Automatic Control CEA–IFAC (Comité Español de Automática–International Federation on Automatic Control).

Francisco Esquembre received the Ph.D. degree in mathematics from the University of Murcia, Murcia, Spain, in 1991.

Since 1986, he has been with the University of Murcia, where he has been holding the position of Associate Professor since 1994. His academic expertise includes differential equations, dynamical system, and numerical analysis. His research includes computer-assisted teaching and learning.