ISSN 0280–5316
ISRN LUTFD2/TFRT--5534--SE

Desarrollo de Herramientas de Control Adaptativo para el sistema SCM

Stefan Svensson

Department of Automatic Control
Lund Institute of Technology
July 1995

Department of Automatic Control
Lund Institute of Technology
P.O. Box 118
S-221 00 Lund Sweden

Document name
MASTER THESIS

Date of issue
July 1995

Document Number
ISRN LUTFD2/TFRT--5534--SE

Author(s)
Stefan Svensson

Supervisor
J. L. Navarro Herrero, Valencia, K J Åström, LTH

Sponsoring organisation

Title and subtitle
Desarrollo de Herramientas de Control Adaptativo para el sistema SCM. (Object Oriented Tools for Adaptive Control) (Objektorienterade verktyg för adaptiv reglering).

Abstract

This project was done at the department of Systems, Computers and Automation (Departamento de Ingeniería de Sistemas, Computadores y Automática) in Valencia under the HCM project Euraco. The thesis deals with the development of object oriented tools for adaptive control. The project has resulted in an extension of the distributed control system, SCM, which is a control system developed at the department. It has facilities to control, simulate and monitor industrial processes. The system is implemented in Smalltalk, which is an interpreted language. Therefore the system requires slow sampling rates in the range of 1 Hz. Classes for adaptive control using different adaptive control algorithms have been created and connected to SCM with a graphical user interface. It has been possible to switch between different algorithms for estimation and controller design.

The report, which is written in Spanish, describes and analysis the problem in an object-oriented way. The classes implemented in Smalltalk, are used as a basis for the presentation. An introduction to Smalltalk and object oriented programming in general is given. A brief presentation of adaptive control is also included together with a detailed study of an indirect self-tuning controller. Some practical aspects of the implementation of adaptive regulators are discussed. A summary of the report in English is included.

Key words

Classification system and/or index terms (if any)

Supplementary bibliographical information

ISSN and key title
0280–5316

ISBN

Language
Spanish

Number of pages
60

Recipient's notes

Security classification

English summary

During the last few years computers have provided us with an enormous computing capacity. There have also been made large progresses in the field of adaptive control. It is therefore interesting to implement adaptive regulators in high level programming languages. Some people also say that the object orientation is the *definitive* way to develop large systems. All this together give us a good reason for the implementation of an adaptive regulator in a distributed control system in the programming language *Smalltalk*.

The main part of this work was done at the department of Systems, Computers and Automation (*Departamento de Ingeniería de Sistemas, Computadores y Automática*) at UPV, (Universidad Politécnica de Valencia) in Valencia, Spain. Since the main report is written in Spanish, I will here give a short summary in English.

In the summary I will begin by presenting the distributed control system *SCM, Sistema de Control Multiproceso* and the basic ideas in *Smalltalk*. Then I will discuss some issues of adaptive control: What adaptive algorithms should be used, and how they should be implemented. We will discuss how object oriented methodology can be used to structure the problem. Finally we will look upon a few of the objects with their attributes and operations.

SCM — *Sistema de Control Multiproceso*

SCM is a distributed control system developed in an earlier project at the *Universidad Politécnica de Valencia* [7]. The idea is to have a general tool for control of different kinds of industrial plants. It has facilities for control, simulation and monitoring. SCM was implemented in Smalltalk under OS/2 on a personal computer.

Smalltalk

Smalltalk is a purely object oriented language. It has all the typical object oriented characteristics: identity, classification, encapsulation, polymorfism and inheritance. Different types does not exist, everything in Smalltalk is an object. In Smalltalk it is not possible to write code apart from the methods in the classes. This leads us to the conclusion that programming in Smalltalk is to augment a system with more classes that helps to solve the problem. Smalltalk uses inheritance in full. A sub-class inherits all attributes and methods from all its super-classes.

There are different versions of Smalltalk. Normally it comes as a complete system with tools for text editing, search and debugging. It also has a complete library to build user friendly graphical interfaces.

Smalltalk is a high level language. The programmer does not explicitly have to work with pointers or reserve memory for variables. Smalltalk is an interpreted language, which means that the code is not compiled into machine code. This gives a slow execution which is the major drawback of Smalltalk. For large control systems it restricts sampling rates to the order of 1 Hz.

Smalltalk supports concurrent programming. We can create various processes from the class *Process*. The class *Semaphore* helps with synchronization between processes.

Task

The task, to start off, was to extend SCM with object oriented tools for adaptive control. An indirect self-tuning regulator with one or two possible estimators and with one or two possible regulator designs was implemented. The tools should be general so that they can be applied to any kind of process, that is to any order of the process model. It should also be possible to extend the system with more estimators and regulator designs. The tools should have a graphical user interface, where the user can define the control parameters.

The major classes

Figure 1 shows a block diagram of an indirect self-tuning regulator.

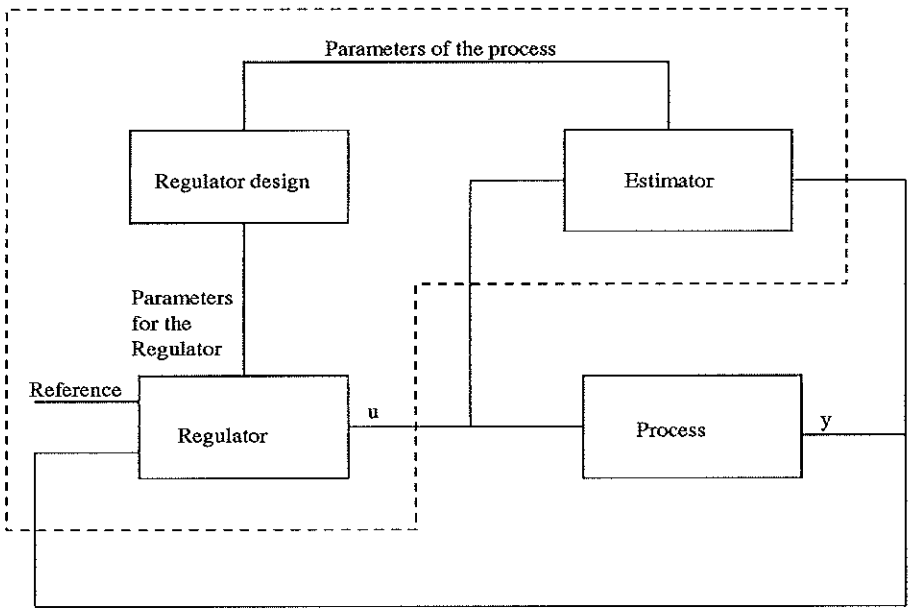

Figure 1: *A process with an indirect self-tuning regulator. Within the dashed line we see the components that form the indirect self-tuning regulator.*

If we try to analyze the problem in an object oriented way we should start by identifying the different objects. In Figure 2 we see some of the objects. They are estimator, regulator design and regulator. To connect the controller to industrial plants we can choose to modelize other objects like sensor, actuator, signal and sampling card. For the implementation of the regulator algorithms we need some mathematical tools that allows us to deal with matrices, polynomials and transfer functions.

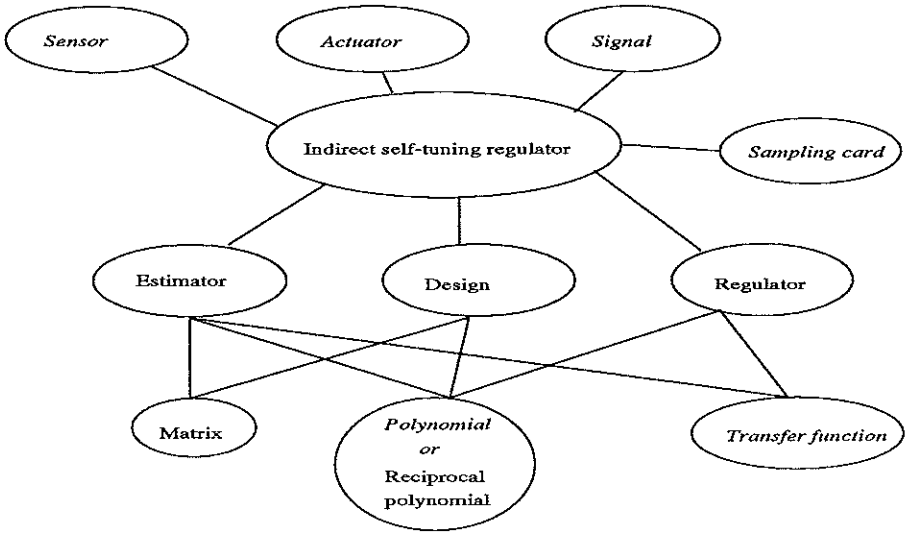

Figure 2: *Some of the objects needed to modelize an indirect self-tuning regulator. Objects written in cursive already existed in the system.*

Once all different objects are identified they have to be classified. We must try to see what different objects have in common: What attributes do they share, and which operations are needed? This structuring is a difficult and an important part of the work. If we want to keep the door open for a further augmentation of the system we have to get a large class hierarchy from the beginning. The idea is that a part of the work already should be done if we decide to implement another controller algorithm. What save work in this case is the inheritance between classes. On the other hand that may result in a system that is difficult to overview. We also want to keep the objects general, because we might want to use them in some other application.

To continue with the indirect self-tuning regulator we start by analyzing the object *Estimator*. If we want to classify different kinds of estimation algorithms we could classify them into two groups depending on how we choose to model the process. We could either choose a model in the form of a transfer function or a state model. If we choose to work with transfer function models they can be divided into different categories, regression models and other models. Regression models can be divided into stochastic and deterministic models. Two different kinds of estimators have been implemented, the *Recursive Least Squares estimator* and a *Projection algorithm*. They are both regression models. Therefore we will not go any further than to define an abstract class — a class that will not have any instances — *Estimator* with the two sub-classes *Rls* and *Projection*. *Estimator* contains attributes like name, order, delay, parameter vector and regression vector and methods to define parameters and to update the regression vector. The instances created will be instances of *Rls* or *Projection*. These classes will have attributes for specific parameters of each

algorithm and has methods to update the parameters.

In an indirect self-tuning regulator we make a regulator design every sampling instant which is based on the estimated process parameters. The design can be any kind of regulator design. Basically we can divide different designs based on transfer function models into two categories: pole placement and designs based on optimization, for example a linear quadratic regulator. I have chosen to implement a pole placement design with possibilities to cancel or not to cancel the process zeros. Analyzing the different designs we get the abstract class *RegulatorDesign* that represents all designs based on transfer function models. It will have attributes like name, order and pole excess and methods to define the attributes. *RegulatorDesign* will have the sub-class *RstPolePlacement* that is also an abstract class containing the attributes A_m, A_o, R, S and T among others. The sub-classes to *RstPolePlacement* will be the classes *AllZeroCancellation* and *NoZeroCancellation* from witch we create instances. They will have attributes representing the parameters and methods to obtain the regulator polynomials.

The regulators are divided into different kinds of regulators using the abstract class *RegulatorTypes*. Its subclasses will be the other abstract classes *Pid* and *PolynomialRst*. *PolynomialRst* will have the attributes R, S and T. Its subclasses will be the two classes *Rst* and *RstAntiWindUp*. The first one representing an algorithm that does not compensate for the actuators physical range and the second one that does. These two classes will have methods to calculate the control signal. Due to the fact that it always takes some time to calculate the control signal it is not that good to do all calculations as one step, because this will cause unnecessary calculation delays. A better way of doing it is to divide the calculations into two steps. Between the A–D and the D–A conversions we calculate only what has to be calculated to obtain the control signal. After that we do the rest of the calculations after sending the control signal to the actuator.

In Table 1 we see the hierarchy of classes that represent the estimator, the design and the regulator in Figure 2.

We also need a class to represent the indirect self-tuning regulator itself. We get an abstract class *ReguladoresPlanta* that represents all different types of regulators used in the system. It will have the attributes name, type and loop. Its subclasses will be *Regulador*, representing a PID, and *ReguladorAdaptativo*, an abstract class representing different kinds of adaptive regulators. *ReguladorAdaptativo* will in this project have one sub-class representing an indirect self-tuning regulator, *ISTR*. This class will contain attributes to represent different kinds of estimators, designs and regulators. We see these classes in Table 2.

Finally, to be able to implement the classes above we need the mathematical objects: *Matrix*, *SignalVector* and *Reciprocal* listed in Table 3.

User interface — Connection to SCM

When all classes for the adaptive controller are constructed they should have a graphical user interface, they should also be connected to SCM. The user interface has the form of dialoguewindows that pops up from the main window

Estimator
    Projection
    Rls
RegulatorDesign
    RstPolePlacement
        AllZeroCancellation
        NoZeroCancellation
RegulatorTypes
    Pid
        Clasico
        WindUp
    PolynomialRst
        Rst
        RstAntiWindUp

Table 1: *Classes needed for the estimator, design and regulator. Classes in cursive already existed in the system.*

ReguladoresPlanta
    Regulador
    ReguladorAdaptativo
        ISTR

Table 2: *Classes needed for representing all regulators of the plant. The class Regulador already existed in the system.*

of the SCM. The windows can be drawn in a graphical editor. Later we connect all logic in a class representing the window.

The last part of the problem is to integrate it to SCM. This requires a good knowledge of the system and it required a few changes in some of the existing classes. It could have been done differently, but I tried to do it in such a way that the changes of the system should be as few as possible. Problems can always be solved in different ways and it is difficult to consider all possible future extensions when designing a system. It is important to document and comment the code as well as possible even if this is tedious. This will make it it easier for other programmers to understand how the system works.

Description of a few classes

Appendix B lists the class definitions of *Estimator*, *Rls*, *Matrix*, *DlgDiseno*. I will describe the attributes and the methods of the three first ones. The last one, *DlgDiseno*, is shown as a sample of a class representing a user interface window. Programmer style in Smalltalk is that an access function or a set procedure should have the same name as the attribute.

Funcion
    Matrix
    SignalVector
Polinomio
    Reciprocal

Table 3: *Other classes needed for the implementation of some of the classes in Table 1. Classes in cursive already existed in the system.*

The class *Estimator*

The class **Estimator** is an abstract class representing an estimator that estimates parameters of a regression model.

It defines the public instance variables **regressor** and **parameters** to represent the process model in discrete time. It also have the variables **order** and **delay**. We give the estimator a name using **name**.

We have methods to set and get the values of the variables and a method, **updateRegressorWithU:withY:** that we use each sampling instant in order to update the regressionvector. The methods **a** and **b** return the reciprocal polynomials representing the denominator and numerator respectively.

The class *Rls*

The class **Rls** is a sub-class of the class *Estimator* for representing a recursive least squares estimator.

The class has the instance variables **initVariances**, **defaultInitVariances**, **pMatrix** and **lambda**. The variances represents the initial values of the covariance matrix **pMatrix**. In order to be able to track changes of the process dynamics we use a forgetting factor, **lambda**.

We create an instance sending the message **order:delay:** to the class object. Doing so lambda will initialize to 1 and the variances to 1000 as default values. We reinitialize the estimator sending the message **reset** to the instance. To define and obtain the current variances we send the messages **diagonal:** and **diagonal** respectively.

The class *Matrix*

The class **Matrix** represents a matrix.

We create an instance sending the message **rows:columns:** to the class object. To more easily crate a quadratic matrix we send the message **dimension:**.

We assign values to a matrix with the message **assign:** passing as the argument an *array[array[elements of the first line], array[elements of the second line], ...]*. We can also define or obtain a certain element using the methods **atRow:atColumn:put:** and **atRow:atColumn:** respectively. To obtain the matrix we send the message **matrix**. The following methods are implemented: **+**, **-**, ***, **transpose**, **inverse** and **diagonal**. To obtain the inverse of a matrix we use a numerical method named *Gauss–Jordan algorithm*. It is a numerical

stable algorithm that performs gauss elimination on a linear set of equations [8].

Acknowledgments

I would like to thank my supervisor professor Karl Johan Åström for helping me arrange the project. I would also like to thank professor Pedro Albertos at UPV, who made it possible for me to do the project in Valencia and my supervisor in Valencia Jose Luis Navarro, who always helped me.

The work was done under the auspices of the EU Human Capital and Mobility program on Nonlinear and Adaptive Control. Support from Lund University and *Svensk–Spanska stiftelsen* is also acknowledged.

UNIVERSIDAD POLITÉCNICA DE VALENCIA
Departamento de Ingeniería de Sistemas, Computadores y Automática

Desarrollo de Herramientas de Control Adaptativo para el sistema SCM

PROYECTO FIN DE CARRERA

Presentado por:
Stefan Svensson

Dirigido por:
Jose Luis Navarro Herrero

Valencia
Mayo, 1995

Resumen

El proyecto trata el desarrollo de herramientas de control adaptativo y es una ampliación del sistema de control multiproceso, SCM. Se dan los pasos de describir y analizar el problema en una manera orientada a objetos. El resultado se presenta en forma de una descripción de las clases implementadas en el lenguaje de programación de alto nivel, Smalltalk. Se dan los fundamentos de Smalltalk y la programación orientada a objetos. Brevamente se presenta un resumen del control adaptativo en general y un estudio más ampliado de un regulador auto-sintonizado indirecto. Se discuten los aspectos practicos de la implementación de reguladores adaptativos.

This project concerns the development of tools for adaptive control and is an extension of the multiprocess controlsystem, SCM. Steps describing and analyzing the problem in an object-oriented way are discussed. The classes implemented in the high level programming language, Smalltalk, are used as a basis for the result presentation. An introduction to Smalltalk and object oriented programming in general is given. A brief presentation of adaptive control is also included along with a more detailed study of an indirect self-tuning regulator. Some practical aspects of the implementation of adaptive regulators are discussed.

Contenido

Índice de Figuras

Índice de Tablas

1 Introducción

En los ultimos años ha sido posible la implemetación de reguladores adaptativos en lenguajes de alto nivel. La razón principal es la capacidad de computación que nos ofrecen las computadoras de hoy. Tambien han surgido las técnicas orientadas a objetos como la técnica *definitiva* para el desarrollo de sistemas. Así pues, nuestra aplicación se desarrollará en el lenguaje orientada a objetos *Smalltalk* sobre el sistema operativo *OS/2*.

El sistema de control multiproceso *SCM* es una herramienta genérica tal que puede ser utilizada con cualquier planta industrial. Nos ofrece posibilidades de controlar, simular y monitorizar plantas industriales. Este proyecto resultará en una ampliación de las herramientas de control del sistema *SCM*.

La implementación será de forma de escribir clases con las cuales podamos construir un regulador auto-sintonizado con su interfaz de usuario. Veremos en el proyecto los pasos de describir y analizar el problema. Se presenta el resutado en forma de una descripción de las clases que han sido implementadas. Así pues, primero hablaremos de los conceptos más básicos de *Smalltalk* y de programación con objetos. Tratamos de describir como identificar los objetos y clasificarlos. Hablaremos después sobre el control adaptativo y veremos un regulador auto-sintonizado con su estimador y su diseño del regulador más detallado. También trataremos de describir unos aspectos que siempre hay que tener en cuenta al implementar un regulador por computador.

Los motivos para hacer este proyecto han sido varios, pues me interesa mucho el control adaptativo y la programación. Aunque no tenía ninguna experiencia de *Smalltalk* ni tampoco de programar con objetos, ha sido algo que me ha gustado aprender. También desde hace muchos años me ha gustado aprender el castellano y ahora he tenido la oportunidad de combinar las tres cosas y además conocer Valencia.

Quisiera dar las gracias a mi director del proyecto, el profesor Jose Luis Navarro Herrero, por su orientación de este trabajo y por el tiempo que siempre ha dedicado a ayudarme. Además a dos personas que me han ayudado hacer una realidad este proyecto en Valencia, el catedrático de Universidad Pedro Albertos y el catedrático de Universidad Karl Johan Åström, del Instituto de Tecnología de Lund, LTH (Lunds Tekniska Högskola).

2 Programación orientada a objetos

En los ultimos años las técnicas de orientación a objetos han llegado como la técnica *definitiva* para el desarrollo de aplicaciones y sistemas. Razonablemente la mayor tipos de programas ya deben ser construídos y no solo unica vez. La programación orientada a objetos *OOP* (del inglés Object Oriented Programming) tiene grandes posibilidades de reutilizar código o objetos ya hechos anteriormente. En el porvenir el desarrollo de sistemas pronostican no consistirá en programmmación clasica, sino más en integración de objetos ya hechos. Asi pues, en la próxima fase no hablaremos de programación con objetos, hablaremos de programación de integración.

2.1 ¿ Qué es un objeto ?

Dentro de OOP usamos objetos para modelizar un sistema. Un objeto es un modelo que representa algo real, por ejemplo un cubo, un hombre o un libro. También un objeto puede representar a otras cosas, como un número, un registro o una matriz. Cada objeto tiene sus *atributos* y *operaciones*. Los atributos son valores almacenados dentro del objeto que también pueden ser otros objetos. Usamos las operaciones para crear objetos, cambiar los valores de los atributos, etc. Véase la figura 1; el cubo tiene los atributos volumen, color y posición. Mediante las operaciones cambiar_color, cambiar_volumen y mover podemos cambiar los valores. También podemos tener operaciones para crear y distruir un cubo.

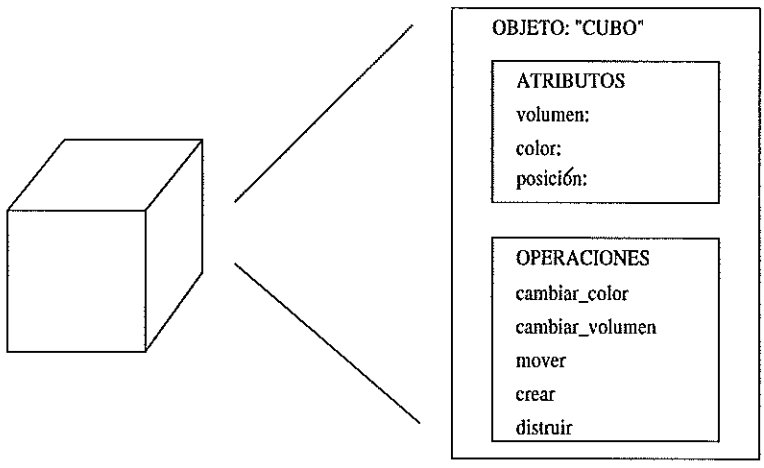

Figura 1: *Un objeto que representa un cubo con sus atributos y operaciones.*

Las características que tienen los objetos son las siguientes:

- Identidad
- Clasificación
- Encapsulación y ocultación de datos
- Polimorfismo
- Herencia

La *identidad* expresa que podemos crear dos objetos exactamente iguales pero distintos entre sí.

La *clasificación* de objetos es una propiedad fundamental para los lenguajes OOP. Nos permite la agrupación de objetos que comparten las mismas propiedades. Mediante la clasificación identificaremos los atributos y las operaciones que los objetos diferentes tienen en común.

La *encapsulación y ocultación* de datos nos ayuda en el desarrollo de sistemas grandes. Es suficiente saber el resultado de una operación, no hace falta saber el código interno. Así será fácil para varios programadores usar los objetos ya hechos.

El *polimorfismo* es una carecterística aportada por la OOP. Expresa la posibilidad de tener varias operaciones con el mismo nombre. El resultado de una operación va a depender de los objetos. Por ejemplo, la operación *suma* tendrá el mismo nombre bien en una operación que tenga la clase que representa los numeros enteros, bien en una operación que tenga la clase que representa las fracciones. El código de las dos operaciones será diferente, pero el resultado será una adición en los dos casos.

La *herencia* nos permite crear estructuras jerárquias de clases. Podemos crear sub-clases que hereden los atributos y las operaciones de todas las clases que estan arriba en la jerarquía. En esta manera podemos ahorrar trabajo que es una propiedad muy importante de la OOP.

2.2 Cómo programar con objetos

La programación orientada a objetos consiste en modelizar el sistema y definir las clases necesitadas para solucionar el problema. Encontraremos los siguientes pasos:

1. Identificar los objetos
2. Definir los atributos de los objetos
3. Definir las operaciones de los objetos
4. Clasificar los objetos en una jerarquía de clases
5. Codificar

Para ver los pasos en detalle, veremos un ejemplo de modelizar una casa. Primero identificamos los objetos habitación, silla, mesa, etc. Luego definimos los atributos y las operaciones de cada objeto. Una habitación puede tener los atributos mesas, sillas y las operaciones mover_silla, mover_mesa, entre otros. Finalmente podemos clasificar las habitaciones en comedor, baño, etc. Una vez hecho esto, podemos empezar a codificar.

2.3 Smalltalk

Smalltalk fue el primer lenguaje OOP. Fue desarrollado en los años setenta y se dice que es el lenguaje OOP más puro. Smalltalk tiene un sintaxis muy simple. Todo está basada en clases y todo en Smalltalk es un objeto.

Smalltalk es un lenguaje de alto nivel. El programador no tiene que trabajar con punteros. Ya existen estructuras implementadas con punteros. También el manejo de memoria se hace automáticamente. No nesecita el programador reservar memoria explicitamente y delitimar memoria explicitamente para sus variables. En Smalltalk el código no se compila al código de instrucciones de máquina. Smalltalk interpreta el código durante ejecución. Esto nos da una ejecución más lenta que otros lenguajes.

Además de un lenguaje, Smalltalk es un ámbito de desarrollo. Tiene herramientas para para editar texto, buscar en el código y encontrar errores (*debugger*), y tiene clases para construir fácilmente un interfaz de usuario.

Un sistema de Smalltalk consiste en un conjunto de clases. Una clase en Smalltalk es una descripción del *objeto de la clase* y las *instancias* — los objetos que creemos durante la ejecución— que pertenecen a la clase. Cuando se ha empezado la ejecución de un programa, se puede verla como un intercambio de mensajes entre objetos. En este intercambio podemos considerar el objeto de la clase como una fábrica que construye instancias.

Todas las clases pertenecen en una cadena de super-clases. Un sub-clase hereda todos los atributos y mensajes de sus super-clases. Todas las clases que defina el programador tendrá la clase *Object* como su superclase, pues todas las clases serán sub-clases de la clase *Object*. Existe el concepto de clases *abstractas*. No existirán instancias de estas clases. Su función es ser super-clase de otras clases en la jerarquía. La clase *Object* es un ejemplo de una clase *abstracta*.

Cada objeto tiene sus atributos, implementado con variables, y sus operaciones, en Smalltalk denominadas *mensajes*. Cuando un objeto reciba un mensaje, ejecuta el *método* cuyo nombre coincide con el nombre del mensaje. Como hay dos tipos de objetos, objetos de clase e instancias, existen dos tipos de mensajes, mensajes de clase y mensajes de instancia. Los mensajes de clase los mandamos a objetos del clase; normalmente son mensajes para crear instancias. Los mensajes de instancia los mandamos a instancias; pueden ser mensajes para cambiar los valores o pueden ser mensajes que da otro objeto como resultado. También hay las variables de clase y las vari-

ables de instancia. Además hay variables compartidas, variables temporales y variables globales.

Al recibir un mensaje, el objeto empieza a buscar el método de ejecutar en su cadena de super-clases. Cuando encuentre uno, terminará de buscar y ejecutará el método. Puede ser que haya métodos con el mismo nombre en clases más alta en la jerarquía que no podemos encontrar. Una manera para empezar a buscar en una clase más arriba es con ayuda de *super*. Esta técnica se usa frecuentamente en Smalltalk.

Como todos expresiones deben tener la forma *objeto mensaje*, encontaremos problemas cuando un objeto quiera usar un mensaje suyo. Para evitar el problema usamos *self* para referirnos al objeto actual.

Se debe acceder a las variables de instancia mediante un mensaje para preservar la encapsulación. Normalmente el estilo de asignar valores u obtener los valores de variables de instancia es tener métodos con el mismo nombre que la variable. Por ejemplo, si tenemos la variable **unaVariable**, el método para asignar un valor tendrá el nombre **unaVariable:** y el método de obtener el valor almacenado será **unaVariable**. Aquí el primer método es de tipo palabra de clave y el segundo es un mensaje unario. Los mensajes de tipo palabra de clave tienen un parámetro o más y los unarios no tienen ningún parámetro. Además hay mensajes binarios que representan operadores y tienen un parámetro. Por ejemplo, $4 + 3$, al objeto 4 le mandamos el mensaje binario + con el parámetro 3, el resultado será un objeto que representa el valor 7. El programador puede definir sus propios mensajes binarios. No existe prioridad entre los mensajes binarios. La prioridad entre los tre tipos es

1. Mensajes unarios
2. Mensajes binarios
3. Mensajes de tipo palabras de clave

Smalltalk suporta la programación concurrente. Tiene la clase *Process* y la clase *Semaphore*. Con *Process* podemos crear tantos procesos como queremos. Con *Semphore* podemos crear semáforos que nos ayudan en la sincronización de procesos múltiples.

3 Control adaptativo

A partir de los anos cincuenta, los progresos en la investigación de los reguladores adaptativos han sido grandes. Como ahora hay ordenadores de suficiente capacidad, ha sido posible la implementación de reguladores adaptativos en lenguajes de alto nivel.

El control adaptativo existe en varias formas. Las formas más conocidos son los sistemas adaptativos de modelo de referencia *MRAS* (del inglés Model-Reference Adaptive Systems) y los reguladores auto-sintonizados *STR* (del inglés Self-Tuning Regulators). En cuanto a los reguladores auto-sintonizado, existen en dos tipos, el directo y el indirecto. En el directo aplicamos el diseño del regulador en el algoritmo de estimación y asi pues obtendremos los parámetros del regulador directo del algoritmo de estimación. En el indirecto la estimación y el diseño son dos cosas distintas. En este proyecto el regulador adaptativo implementado ha sido en forma indirecta, por consiguiente, lo estudiaremos de manera más detallada.

3.1 Regulador auto-sintonizado indirecto

Un regulador auto-sintonizado consiste en estimar parámetros del proceso y luego hacer un diseño basado de los parámetros estimados. Esto se repite cada período de muestreo. En la figura 2 se muestra un regulador auto-sintonizado indirecto en un diagrama de bloques.

Hay dos lazos del regulador auto-sintonizado. Existe el lazo de realimentación *normal* y lineal, además del lazo de estimación. Las constantes del tiempo no serán iguales para los dos lazos. La constante de tiempo de la estimación será mucho más lenta. Esto será útil en el caso de análisis de estabilidad del regulador. Debemos tener en cuenta que en lazo cerrado el regulador auto-sintonizado será no lineal.

Podemos aplicar cualquier método de estimación y cualquier método de diseño.

3.1.1 Modelo matemático del proceso

Hace falta un modelo matemático del proceso. Puede ser un modelo del tipo entrada–salida o un modelo en el espacio de estados del proceso. Estos dos tipos de modelos se denominan modelo externo y modelo interno respectivamente. Cuando ya tengamos un modelo del proceso, existen métodos o algoritmos para estimar coeficientes o estados del proceso.

El mundo no es perfecto. En la realidad siempre hay pertubaciones; así pues, lo mejor es tratar de conocerlas e incluirlas en los modelos. También hay algoritmos para modelos que incluyen pertubaciones.

Cuando estimamos parámetros de un modelo externo, el modelo puede

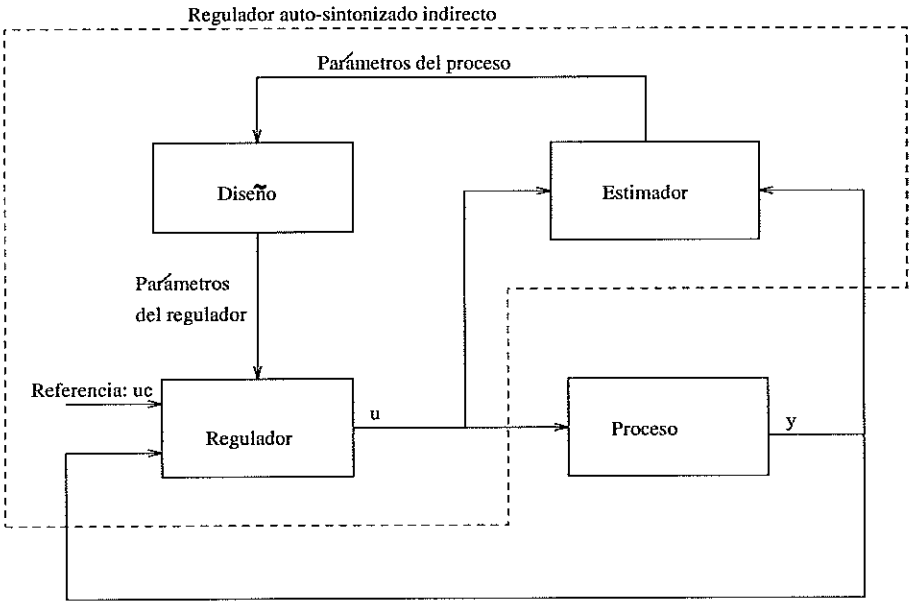

Figura 2: *Un regulador auto-sintonizado indirecto. Dentro de la línea discontinua tenemos las entidades que forman un regulador auto-sintonizado*

escribirse en la forma

$$y(t) = \varphi_1(t)\theta_1 + \varphi_2(t)\theta_2 + \dots \varphi_n(t)\theta_n = \varphi(t)^T\theta$$

donde y es la variable observada, φ funcciones conocidos y θ los parámetros no conocidos. El vector φ^T es el vector de regresión y el vector θ es el vector de los parámetros. Se obtienen datos de y y φ mediante un experimento. Este tipo de modelo se conoce como modelos de regresión. Estos modelos los podemos usar para el caso deterministo (sin calcular con los pertubaciones en el modelo) y para el caso estocástico. Es decir que el proceso puede representarse en un modelo de función de transferencias o en una ecuación de diferencias

$$A(q)y(k) = B(q)u(k) + C(q)e(k)$$

o

$$y(k)+a_1y(k-1)+\dots a_ny(k-n) = b_1u(k-1)+\dots b_nu(k-n)+c_1e(k-1)+\dots c_ne(k-n)$$

donde u es la entrada al proceso, y es la salida del proceso y e es una pertubación de tipo ruido blanco. En el caso determinado ($C = 0$) obtenemos un modelo de regresión, donde

$$\theta = [a_1 \dots a_n b_1 \dots b_n]$$

$$\varphi^T(t) = [-y(k)\ldots\ -y(k-n+1)u(k)\ldots\ u(k-n+1)]$$

.

3.1.2 Estimación de parámetros del proceso

Hay varios métodos de estimar parámetros del proceso. Podemos hacer la estimación en línea o fuera de línea. Si lo hacemos en línea, será conveniente con un algoritmo recursivo. Una razón para implementar los reguladores adaptativos es que queremos cambiar los parámetros del regulador cuando los parámetros del proceso cambian. Por eso introducimos un factor de olvido para eliminar la influencia de datos antiguos en las estimaciones.

Tenemos diferentes algoritmos de estimación para el caso determinista, pues el más conocido es el de mínimos cuadrados formulado por *Gauss* en redonda a finales del siglo XVIII. Este algoritmo en la forma recursiva se denomina *RLS* (del inglés Recursive Least Squares). El *RLS* con un factor de olvido λ viene dado por

$$\hat{\theta}(k) = \hat{\theta}(k-1) + K(k)(y(k) - \varphi^T(k)\hat{\theta}(k-1))$$

$$K(k) = P(k)\varphi(k) = P(k)\varphi(k-1)[I\lambda + \varphi^T(k)P(k-1)\varphi(k)]^{-1}$$

$$P(k) = [I - K(k-1)\varphi(k)]P(k)/\lambda$$

P es una matriz proporcional a la varianza de las estimaciones. Tendremos que calcular $\hat{\theta}$ y P cado período de muestreo. El tiempo más costoso será el tiempo para los calculos de la matriz P.

Para el caso de que tengamos muchos parámetros, un período de muestreo muy corto y una insuficiente capacidad del computador, hay algoritmos más simples. *El algoritmo de proyección de Kacmarz* es un algoritmo que nesecita menos capacidad de computación, a costa de convergencia más lento. Una versión práctica de este algoritmo para el caso deterministo viene dada por

$$\hat{\theta}(k) = \hat{\theta}(k-1) + \frac{\gamma\varphi}{\alpha + \varphi^T(k)\varphi(k)}(y(k) - \varphi^T(k)\hat{\theta}(k-1))$$

donde $\alpha \geq 0$ y $0 < \gamma < 2$

Para el caso general (con pertubaciones calculadas en el modelo) tenemos otros algoritmos. Unos de estos son el mínimos cuadrados ampliados, *ELS* (del inglés Extended Least Squares) o el máxima probabilidad recursivo, *RML* (del inglés Recursive Maximum Likelihood). Para una descripción ampliada de estos o del tema con detalle véase [2] o [3].

3.1.3 Diseño del regulador

Para diseñar un regulador lineal se puede decir que hay dos tipos de diseños. Un tipo son los métodos de optimación de parámetros, como el lineal cuadrático o el método de mínima varianza. El otro es el método de asignación de polos

que cobra muchos casos de diseño incluso el PID. Aquí veremos el método de asignación de polos.

El problema de diseño de un método de asignación de polos será el siguiente. Supongamos que el modelo del proceso es dado por

$$A(q)y(k) = B(q)u(k)$$

Si deseamos encontrar un regulador tal que la función de transferencia desde la señal de mando (u_c) a la salida (y) viene dada por

$$A_m(q)y(k) = B_m(q)u_c(k)$$

un regulador lineal será

$$R(q)u(k) = T(q)u_c(k) - S(q)y(k)$$

donde R_1 y S son las soluciónes a la ecuación diofántica

$$(q-1)^l AR_1 + B^- S = A_o A_m = A_c$$

donde

$$B = B^+ B^-$$

$$B_m = B^- B'_m$$

$$T = A_o B'_m$$

$$R = B^+ R_1$$

El polinomio B es factorizado en el polinomio B^+ que contenga los ceros estables del proceso y el polinomio B^- que contenga los ceros inestables del proceso. El polinomio A_o es el polinomio del observador y l es el numero de integradores.

Existen varias soluciones a la ecuación diofántica cuando los polinomios A y B^- no tengan factores en común. En el caso de sistemas muestreados es útil tener los grados de los polinomios R, S y T iguales para que no introduzcamos retardos no necesarios en el regulador. En el caso de grados mínimos de los polinomios ($graA_m = graA$) y ($graB_m = graB$) obtendremos los grados de los polinomios

$$graAo = graA - graB^+ + l - 1$$

$$graS = graR = graT = graA + l - 1$$

$$graR_1 = graA_o - l$$

La ecuación diofántica puede resolverse tras la identificación de coeficientes de potencias iguales. De esta manera se obtiene una sistema de ecuaciones lineales. Además en el caso de un integrador ($l = 1$) el polinomio

R_1 tiene que tener la condición $R_1(1) = 0$ o lo que es igual $\sum_{i=1}^{graR_1} r_i = -1$. Así pues obtendremos los parametros resolviendo

$$\begin{pmatrix} r_1 \\ \vdots \\ r_k \\ s_0 \\ \vdots \\ s_l \end{pmatrix} = \begin{pmatrix} 1 & 0 & \dots & 0 & b_0 & 0 & \dots & 0 \\ a_1 & 1 & \ddots & \vdots & b_1 & b_0 & \ddots & \vdots \\ a_2 & a_1 & \ddots & 0 & b_2 & b_1 & \ddots & 0 \\ \vdots & \vdots & \ddots & 1 & \vdots & \vdots & \ddots & b_0 \\ a_n & \vdots & & a_1 & b_n & \vdots & & b_1 \\ 0 & a_n & & \vdots & 0 & b_n & & \vdots \\ \vdots & \ddots & \ddots & & \vdots & \ddots & \ddots & \\ 0 & \dots & 0 & a_n & 0 & \dots & 0 & b_n \\ 1 & 1 & \dots & 1 & 0 & 0 & \dots & 0 \end{pmatrix}^{-1} \begin{pmatrix} a_{c1} - a_1 \\ \vdots \\ a_{cn} - a_n \\ a_{cn+1} \\ \vdots \\ a_{ck+l+1} \\ -1 \end{pmatrix}$$

$\underbrace{\qquad\qquad}_{k \text{ columnas}}$ $\underbrace{\qquad\qquad}_{l+1 \text{ columnas}}$

3.2 Aspectos prácticos de la implementación de reguladores adaptativos

Siempre hay diferencias entre un algoritmo que sacamos de un libro y un algoritmo que queremos implementar para regular una planta industrial. Aquí veremos algunos de estos aspectos de reguladores digitales, de los cuales no todos son aspectos de los reguladores adaptativos solamente.

En todos los casos será necesario seleccionar un periodo de muestreo. La selección depende de varias cosas: de la señal, del método de reconstrucción y del proceso. Además, en unos casos hará falta el prefiltrado para evitar el enmascaramiento de frecuencias.

Así mismo, tenemos que tener siempre en cuenta los aspectos numéricos del computador y la precisión que tienen los convertidores.

3.2.1 Aspectos prácticos de la implementación del estimador

Para estimar coeficientes en tiempo real es necesario que la señal entrada al proceso sea suficientemente rica en frecuencias. Cuanto más parámetros estimamos tendremos más rica en frecuencias que tener la señal. Además, en el caso de que tengamos un factor de olvido (λ) < 1 podemos encontrar problemas de *wind-up del estimador*. Esto porque olvidamos datos y, sino obtenemos nueva información del proceso, despues de un ratito ya no tengamos más información del proceso. La matriz P va a crecer bastante y cuando venga nueva información los parámetros podrán cambiarse abruptamente. Hay maneras de evitar este problema. Es posible hacer un *reset* de la matriz P de vez en cuando. Otra manera es incluir condiciones para cuando vayamos a calcular con la información que viene del proceso.

Tenemos que tener un modelo de suficiente orden para obtener estimaciones adecadas.

Desde un punto de vista numérico, en la implementación de un RLS hay métodos que son mejores que el que hemos visto anterioramente. La razón es que en él operamos con el cuadrado de los valores medidos. Lo que puede presentar problemas debida a errores cometidos en las operaciones. Una forma mejor de realizar los cálculos es calcular con la raíz cuadrada de la matriz P. Para una descripción de estos algoritmos véase [2].

Para que no influyan las pertubaciones es mejor prefiltrar las señales. Un filtro que funciona bien (véase [2]) es filtrar con $1/A_m A_o = 1/A_c$.

3.2.2 Aspectos prácticos de la implementación del regulador

Al implementar un regulador utilizando un computador siempre tendremos retardos de cálculo. Esto ocurre porque las conversiones A-D y D-A y los cálculos llevan un tiempo. Para que los retardos sean mínimos, calcularemos solamente lo que es necesario entre las conversiónes A–D y D–A. El resto de los cálculos las podemos hacer después de mandar la señal de actuación al actuador.

En la realidad, los actuadores siempre tienen un rango físico limitado. Para evitar el fenómeno *wind-up* del regulador hay que calcular con los límites calculando la señal de actuación.

Para un regulador con la ley de control

$$R(q)u(k) = T(q)u_c(k) - S(q)y(k)$$

obtenemos un regulador con compensación de saturación y optimizado para retardos lo menor posible mediante los siguientes pasos

1. Leer y y hacer la conversión A-D
2. Calcular $v(t) = t_0 u_c(t) - s_0 y(t) + v_1(t)$
3. Saturar la señal de actuación $u(t) = sat(v(t))$
4. Hacer la conversión D-A y mandar la señal u al actuador
5. Para la próxima vez calcular

$$v_1(t+1) = (1-Ao)v(t+1)+(Ao-R)u(t+1)+(T-t_0)u_c(t+1)-(S-s_0)y(t+1)$$

Aquí hacen falta solamente dos multiplicaciones y tres adiciones entre la conversión A-D y la conversión D-A.

4 Desarrollo de la aplicación

Veremos aquí los pasos que hay que cumplir para el desarrollo de herramientas de control adaptativo. Primero se presenta una descripción muy breve del Sistema de Control Multiproceso, *SCM*. Luego veremos una especificación sobre cómo funcionará el programa, cómo especificaremos los parámetros del control adaptativo, cómo será la interfaz de usuario y qué va a cumplir el sistema. Después, tratarémos de hacer un análisis del problema, que consiste en identificar las clases que necesitaremos para solucionar el problema. Por último, se presenta una descripción de las clases implementadas y las modificaciones de la implementación del sistema *SCM*.

4.1 El Sistema de Control Multiproceso SCM

El Sistema de Contol Multiproceso, *SCM*, es un sistema desarrollado en otro proyecto de fin de carrera en la Universidad Politécnica de Valencia, (vease [7]). Es una herramienta genérica, de forma que pueda ser utilizada con cualquier planta industrial. Será suficiente definir los elementos que componen la planta, actuadores, sensores, bucles de control, etc. Definida la planta, la aplicación ya estará lista para funcionar. Podremos realizar monitorización, simulación y control directo de la planta.

El sistema fue implementado en el lenguaje Smalltalk/V sobre el sistema operativo *OS/2*

Iniciaremos la aplicación evaluando dentro de Smalltak/V la expresión

SistemaControl new open

Veremos dos ventanas, el panel de control y la ventana principal. Desde el panel de control podemos cambiar las referencias. Desde la ventana principal podemos especificar la supervisión, la simulación y el control directo del sistema. Es un sistema multitarea, es decir que existirán varias tareas dentro del sistema. El sistema mandará mensajes a la ventana principal sobre su ejecución.

Ya iniciado, el sistema consiste en las variables globales, *Sistema*, *Interfaz* y *PControl* que representan el sistema, la ventana principal y el panel de control respectivamente. Dentro del sistema existen varias clases para representar la planta y los elementos que componen la planta. La clase *PlantaIndustrial* representa la planta global. La clase *Subsitema* representa un sub-proceso de la planta. Además hay clases para representar sensores, actuadores, variables, reguladores, etc. El usuario definirá los elementos que componen la planta en un fichero ASCII (véase [7]).

4.2 Especificación de la aplicación

Se pretende crear herramientas de control adaptativo para el sistema SCM. El usuario podrá especificar los parámetros de control. El interfaz para la especificación será mediante una serie de diálogos desde la ventana principal del SCM. Aquí veremos qué y cómo habrá que especificar, pues así hallaremos qué será posible en cuanto a regulación adaptativa del sistema.

Al seleccionar *control* desde el menú *Especificación* de la ventana principal aparecerá el primer diálogo, (vease la figura 3). En él se mostrará una

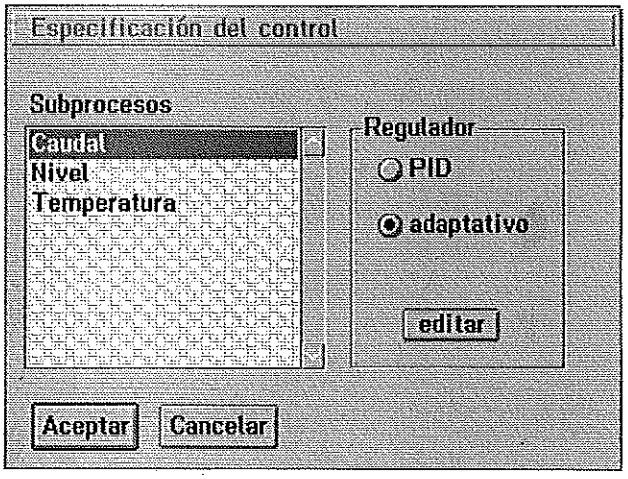

Figura 3: *Primer diálogo en la especificación de los parámetros de control.*

lista de los sub-procesos definidos para la planta. Al seleccionar el nombre de un sub-proceso, veremos cuál es el regulador actual especificado para el bucle de control. Habrá dos tipos de reguladores, PID (ya hecho en el proyecto [7]) y adaptativo, que ha sido implementado en éste proyecto. Cuando ya se tenga un regulador PID y un regulador adaptativo definido, solamente hará falta pulsar el botón *PID* o el botón *adaptativo* para cambiar el regulador del bucle de control.

Al seleccionar el tipo adaptativo y pulsar el botón *editar*, aparecerá el diálogo de la figura 4.

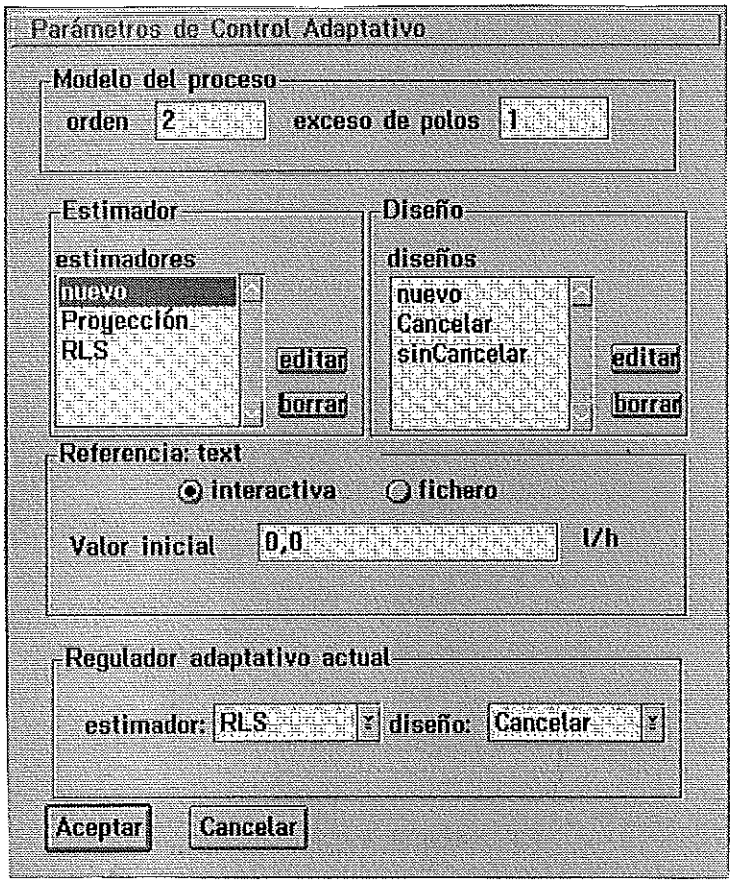

Figura 4: *Diálogo para especificar los parámetros de control adaptativo.*

En él se definirá la referencia y el regulador adaptativo para ese bucle de control. La referencia podrá ser de forma interactiva o de forma predefinida, (vease [7]).

En cuanto al regulador adaptativo, el usuario tendrá que definir el *orden* y el *exceso de polos* del modelo del sub-sistema que quiera modelizar.

Habrá posibilidades de tener varios estimadores y varios diseños. Se definirá el regulador adaptativo actual, que consistirá en un estimador de la lista de estimadores y un diseño de la lista de diseños.

Al elegir *nuevo* o bien un diseño ya definido y pulsar el botón *editar* aparecerá el diálogo de la figura 5.

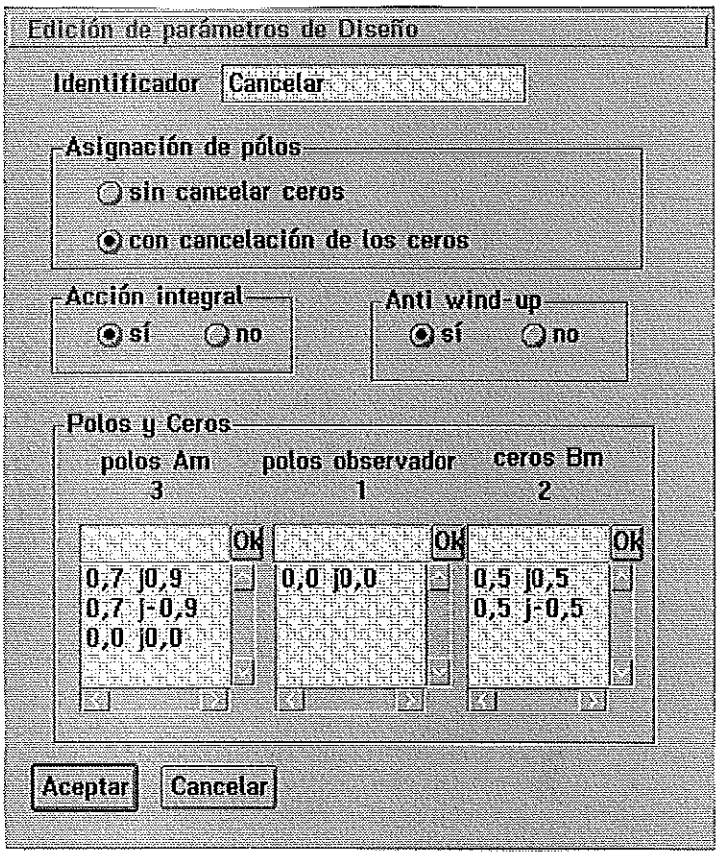

Figura 5: *Diálogo para especificar los parámetros de diseño.*

En él se especificarán los parámetros de diseño. Existirán dos tipos de diseños, uno que no cancelará ningún cero del proceso, el otro que cancelará todos los ceros del proceso. En el caso sin cancelación, se especificará los polos del polinomio, A_m, y los polos del observador, A_o. En el caso con cancelación, además de la especificación de los polinomios A_m y A_o, se especificarán los ceros del polinomio B_m. Los polos y ceros se especificarán en formas discretizadas. En los dos casos, se podrá elegir un diseño, con o sin acción integral. Los números de polos y ceros cambiarán automaticamente a pulsar el boton *sí* o *no* acción integral.

Habrá dos tipos de reguladores, uno que compensará para los rangos físicos de los actuadores, *anti wind-up*, otro que será un regulador lineal.

Al elegir *nuevo* o bien un estimador ya definido y pulsar el botón *editar* aparecerá el diálogo de la figura 6.

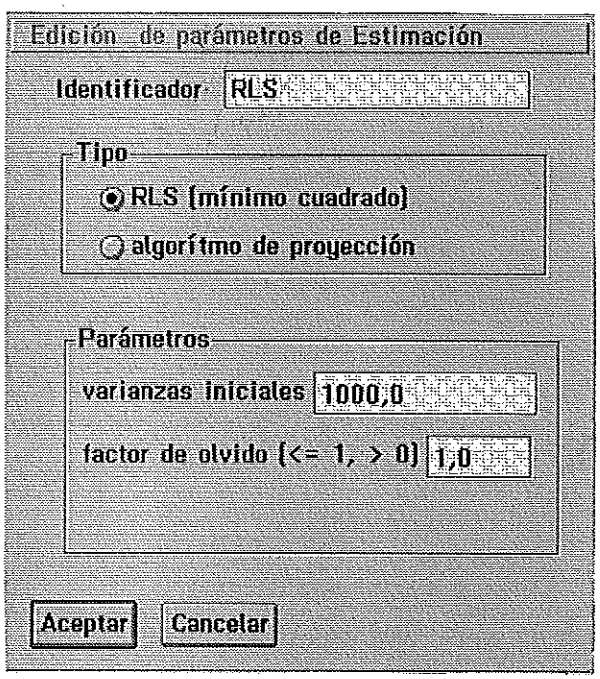

Figura 6: *Diálogo para especificar los parámetros de estimación.*

En él se definirán los parámetros de estimación. Soportará dos tipos de algoritmos, el RLS y un algoritmo de proyección. Para el RLS se especificarán las varianzas iniciales de la matriz P y el factor de olvido λ. Para el algoritmo de proyección se especificará γ (la ganancia) y α.

4.3 Análisis del problema — Cuales serán las clases necesitadas

El desarrollo de la aplicación consistirá en la creación de un conjunto de clases que modelicen las entidades que aparecen en el problema. Así pues, la primera fase será identificar las entidades o los objetos. Una vez conocidas éstas, pasaremos a definir las clases y cómo se relacionan entre sí, es decir donde las ponemos en la jerarquía. Podemos dividir el problema en dos partes, construir las herramientas para el control adaptativo y luego añadirlas al SCM con su interfaz de usuario.

Para encontrar los objetos de un regulador auto-sintonizado indirecto, lo veremos en un esquema de bloques, (véase la figura 2). Primero tendremos el mismo regulador auto-sintonizado como un objeto. Dentro de la línea discontinua tenemos las entidades que forman el regulador auto-sintonizado

indirecto, un estimador, un diseño, un regulador y señales. Además, para acceso a una planta real obtendremos los objetos sensor, actuador y tarjeta de adquisición. Para la construcción de algunos de ellos, obtendremos otros objetos . Por ejemplo para un estimador RLS nesecitarémos matrices; para un diseño de asignación de polos y el regulador, polinomios recíprocos. Smalltalk no soporta estos tipos de entidades. En la figura 7 se muestran los objetos que obtenemos de un regulador auto-sintonizado.

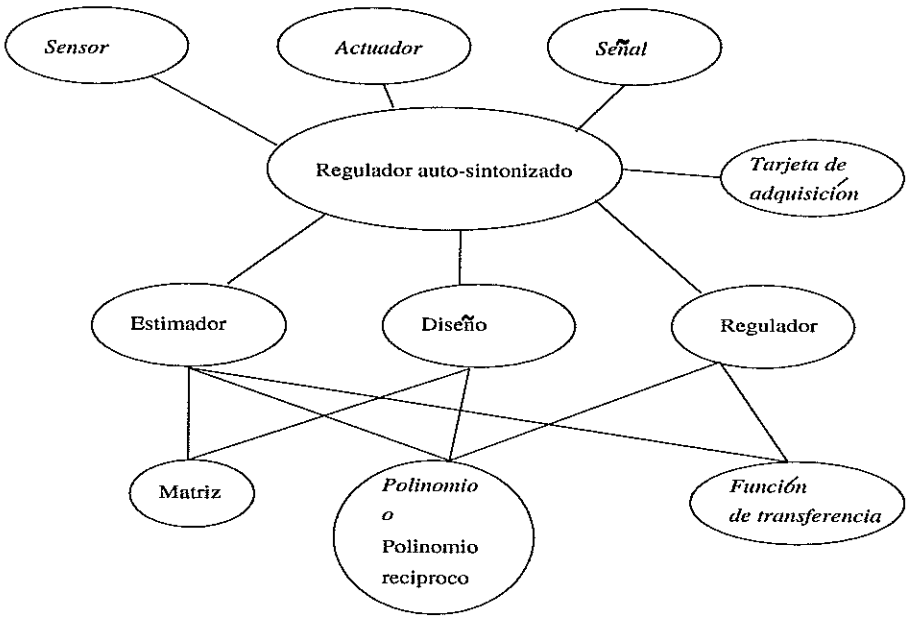

Figura 7: *Los objetos que conforman un regulador auto-sintonizado indirecto y cómo se relacionan entre sí. Las clases para los objetos escritos cursivos ya existían en el sistema*

Ahora sabemos, más o menos, cuáles serán las clases necesitadas. Lo que falta es decidir como vamos a incluirlas en la jerarquía. Debemos pensarlo bien, para que sea fácil la ampliación del sistema. También una jerarquía bien estructurada nos ayuda ahorrar código. Así solamente hace falta escribir única vez los métodos que varios objetos tienen en común.

Veremos, como un ejemplo, como clasificaremos el objeto *diseño*. Todos los métodos de diseño de modelos de forma entrada–salida tienen, entre otros, los atributos orden y exceso de polos en común. Tienen también en común las operaciones para definir el orden y el exceso de polos. Así pues, obtendremos una clase que tenga estos atributos y estas operaciones. Será una clase abstracta, es decir no creamos instancias de esa. Sub-clases de ella podrán ser otras clases abstractas que representa a un método de diseño más especificado, por ejemplo un método de asignación de polos. Luego las

sub-clases de esa clase serán las clases que tengan los otros atributos y las operaciones especiales para cada tipo de diseño de asignación de polos. Asi pues, las clases necesitadas para el estimador, el diseño y el regulador se muestran en la tabla 1

ObjetosBasicos
- Estimator
  - Projection
  - Rls
- RegulatorDesign
  - RstPolePlacement
    - AllZeroCancellation
    - NoZeroCancellation
- RegulatorTypes
  - *Pid*
    - *Clasico*
    - *WindUp*
  - PolynomialRst
    - Rst
    - RstAntiWindUp

Tabla 1: *Las clases necesitadas para las entidades estimador, diseño y regulador. Las clases en cursiva ya existían dentro del sistema.*

Para los reguladores de la planta, los reguladores adaptativos y los reguladores PID, se muestra las clases necesitadas en la tabla 2

ReguladoresPlanta
- *Regulador*
- ReguladorAdaptativo
  - ISTR

Tabla 2: *Las clases necesitadas para las reguladores de la planta. La clase Regulador ya existía dentro del sistema.*

Por último las demás clases nesecitada para las matrices etc., se muestra en la tabla 3.

La segunda parte del problema será incluir las clases en el sistema con su interfaz de usuario. Trataremos de hacerlo con los menos cambios posibles del sistema ya existente. Ha sido necesario añadir pocos métodos a clases que ya existían. Tambien se han requerido unos cambios dentro de clases y

Funcion
    Matrix
    SignalVector
Polinomio
    Reciprocal

Tabla 3: *Otras clases necesitadas para la implementación de las clases de la tabla 1. Las clases en cursiva ya existían dentro del sistema.*

métodos. Los veremos en el epígrafe 4.5. El interfaz será, como ya hemos visto, en forma de una serie de diálogos. Cada diálogo podemos verlo como un objeto, por consiguiente, será una clase para cada diálogo, (véase el epígrafe 4.4.18).

4.4 La Implementación — Una descripción de las clases

En esta parte se presenta una descripción de todas las clases que han sido implementadas en este proyecto.

4.4.1 La clase *Estimator*

La clase **Estimator** es una clase abstracta que representa un estimador.

En esta clase se definen las variables de instancia **regressor** y **parameters**, para modelizar el proceso en una forma discretizada, además , **order** y **delay**. Tenemos las variable **name** para dar un nombre a cada instancia que creamos . Además, hay variables privadas para filtrar las señales antes de ponerlas en el regresor.

Existen métodos para acceder a las variables y el método **updateRegressorWithU:withY:** que se necesita usar en cada período de muestreo. Mediante los métodos **a** y **b** obtenemos los polinomios con los parámetros estimados del denominador y del numerador del modelo del proceso respectivamente. Los polinomios serán polinomios recíprocos, es decir instancias de la clase *Reciprocal*

Existe diferetes tipos de algoritmos para un estimador. Existirá una sub-clase por cada algoritmo que tengamos. En la aplicación se definen dos sub-clases, una para un algoritmo *Rls*, la otra para un algoritmo *Projection*.

4.4.2 La clase *Projection*

La clase **Projection** es una sub-clase de la clase *Estimator*.

Tiene las variables de instancia **alpha** y **gamma** que son parámetros del algoritmo (vease 3.1.2), así como los métodos para acceder a ellas.

En el método **updateParameters** que ejecutamos en cada período de muestreo se hacen los calculos de los nuevos parámetros.

Creamos una instancia al mandar el mensaje **order:delay:** al objeto de la clase. Por defecto **alpha** se inicializa a 0.1 y **gamma** se inicializa a 1. Mandamos el mensaje **reset** para reinicializar el algoritmo, así **alpha** y **gamma** serán reinicializadas con los valores que haya especificado el usuario.

4.4.3 La clase *Rls*

La clase **Rls** es otra sub-clase de la clase *Estimator.*

Tiene las variables de instancia **pMatrix** y **lambda** (vease 3.1.2), así como los métodos para acceder a ellas.

En el método **updateParameters** que ejecutamos en cada período de muestreo se hace los cálculos de los nuevos parámetros. Los cálculos serán cálculos con matrices y en el caso que haya muchos parámetros a estimar, el tiempo para calcularlos será costoso.

Creamos una instancia al mandar el mensaje **order:delay:** al objeto de la clase. Por defecto los elementos diagonales del **pMatrix** se inicializan a 1000 y **lambda** se inicializa a 1. Mandamos el mensaje **reset** para reinicializar el algoritmo. Las varianzas serán reinicializadas con las varianzas y el lambda que ha especificado el usuario. Para especificar y obtener las varianzas actuales mandamos los mensajes **diagonal:** y **diagonal** respectivamente.

4.4.4 La clase *RegulatorDesign*

La clase **RegulatorDesign** es una clase abstracta que representa un diseño general para un regulador basada en un modelo del proceso de forma entrada–salida.

Para representar el modelo del proceso tenemos las variables de instancia **a**, que representa el denominador, y **b**, que representa el numerador. Estas variables tendrán instancias de la clase *Reciprocal.* Tambien tenemos **order** y **poleExcess.**

Tenemos la variable de instancia **name** para dar un nombre a cada instancia que creamos.

La clase tiene los métodos **a**, **a:**, **b**, **b:** etc. para acceder a las variables de instancia.

4.4.5 La clase *RstPolePlacement*

La clase **RstPolePlacement** es otra clase abstracta que representa los diseños de tipo asignación de polos en forma de polinomios, R, S y T (vease el epígrafe 3.1.3).

La clase tiene las variables de instancia **am** y **ao** que tengan instancias de la clase *Reciprocal.* Las variables **amPoles** y **aoPoles** tendrán cada uno

una instancia de la clase *OrderedCollection*, es decir una lista de los polos en formas discretizados. Además hay las variables **r**, **s** y **t** para representar los polinomios del regulador. Existe la variable **regulator** que tendrá una instancia de *sub-clases de PolynomialRst.*

La clase tiene métodos para acceder a las variables de instancia . Estos métodos son **degAm** y **degAo**, que dan como resulatado los grados de los polinomios *Am* y *Ao*. Tiene el método **reset** para asignar un nuevo regulador.

4.4.6 La clase *AllZeroCancellation*

La clase **allZeroCancellation** es una clase que representa un diseño que cancelará todos los ceros del proceso y diseñará los párametros de un regulador *RST*.

Esta clase tiene las variables de instancia **bm** y **bmZeros** para representar los ceros del proceso que el usuario especificará.

Creamos una instancia al mandar el mensaje **order:poleExcess:integrator:** al objeto de la clase. El parámetro *integrator* lo constituirán los numeros de integradores que queramos en el diseño. Al principio soporta 0 o 1.

Al mandar el mensaje **rst** o **rstAo** obtendremos los polinomios del regulador. Los obtendremos en un *array[r,s,t]* — o un *array[r,s,t,ao]* para el caso de un regulador con *anti wind-up* — donde los polinomios serán instancias de la clase *Reciprocal*. El método para obtener los polinomios es formar la matriz (vease el epígrafe 3.1.3), y luego resolver el sistema de ecuaciones lineales mediante eliminación gaussiana numérica.

Al mandar el mensaje **type** obtenemos la respuesta, una cadena, *allCancel*.

4.4.7 La clase *NoZeroCancellation*

La clase **NoZeroCancellation** es una clase que representa un diseño que no cancelará ningún cero del proceso y diseñará los párametros de un regulador *RST*. Es muy parecida a la clase *AllZeroCancellation*, tiene los mismos métodos, pero difiere en la manera de calcular los polinomios (véase 3.1.3).

Igual que en la clase *AllZeroCancellation*, creamos una instancia al mandar el mensaje **order:poleExcess:integrator:** al objeto de la clase. El parámetro *integrator* lo constituirán los números de integradores que queramos en el diseño. Al principio soporta 0 ó 1.

Al mandar el mensaje **rst** o **rstAo** nos da los polinomios del regulador. Los obtendremos en un *array[r,s,t]* —o un *array[r,s,t,ao]* para el caso de un regulador con *anti wind-up*— donde los polinomios serán instancias de la clase *Reciprocal*. Aquí tambien el método para obtener los polinomios es formar la matriz y luego resolver el sistema de ecuaciones lineales mediante eliminación gaussiana numérico.

Al mandar el mensaje **type** obtenemos la respuesta, una cadena, *noCancel.*

4.4.8 La clase *RegulatorTypes*

La clase **RegulatorTypes** no es más que una clase abstracta que representa todos los tipos de reguladores. Tiene como sus sub-clases las clases *Pid*, *PolynomialRst* etc.

Tiene la única varible de instancia **nombre** para dar un nombre a el regulador.

4.4.9 La clase *PolynomialRst*

La clase **PolynomialRst** es otra clase abstracta que representa los reguladores de tipo *RST* con los mínimos retardos posibles (véase 3.2.2).

Tiene las variables de instancia **r**, **s** y **t** para representar los polinomios del regulador. Especificamos y obtenemos los polinomios R, S y T mediante los mensajes **rst:** y **rst**. El parámetro a pasar como argumento será un *array[r,s,t]* con los tres polinomios de forma recíprocos.

Tiene variables privadas para los cálculos de la variable de actuación **t0**, **s0**, **u**, **y**, **uc**, **ySignals**, **ucSignals** y **uSignals**.

Existirán aquí dos tipos de reguladores, es decir que existirán dos subclases de esa clase. Uno que es un regulador lineal, la clase *Rst*; otro que tiene compensación *anti wind-up*, la clase *RstAntiWindUp*.

4.4.10 La clase *Rst*

La clase **Rst** representa un regulador *RST* sin compensación de los rangos físicos que tienen los actuadores.

Creamos una instancia al mandar el mensaje **new** al objeto de la clase.

Al mandar el mensaje **computeOutputWithY:withUC:** obtendremos como resultado el valor calculado de la variable de actuación. Despues de mandar este valor al actuador mandarémos el mensaje **updateState** para hacer los calculos que podemos hacer para el proximo periodo de muestreo.

Al mandar el mensaje **type** obtendremos la respuesta, una cadena, *rst.*

4.4.11 La clase *RstAntiWindUp*

La clase **RstAntiWindUp** representa a un regulador RST con compensación de los rangos físicos de los actuadores.

Creamos una instancia al mandar el mensaje **new** al objeto de la clase.

La clase tiene los métodos **rstAo:** y **rstAo** para especificar y obtener los polinomios R, S y T. El parámetro a pasar como argumento será un *array[r,s,t,Ao]* con los cuatro polinomios de forma recíproca. El polinomio Ao es necesario para la compensación *anti wind-up*. Especificamos el rango

del actuador mediante el mensaje **outputRange:** donde el parámetro es una instancia de la clase *Intervalo* (véase [7]). Mediante el mensaje **ouputRange** obtendremos el intervalo especificado.

Al mandar el mensaje **computeOutputWithY:withUC:** obtendremos como resultado el valor calculado de la variable de actuación. Después de mandar este valor al actuador mandaremos el mensaje **updateState** para hacer los calculos que podemos hacer para el próximo período de muestreo.

Al mandar el mensaje **type** obtendremos la respuesta; una cadena *rstAntiWindUp*.

4.4.12 La clase *ReguladoresPlanta*

La clase **ReguladoresPlanta** es una clase abstracta que representa a todos los reguladores de la planta, pueden ser reguladores pid, reguladores adaptativos, etc.

Dicha clase tiene las variables de instancia **nombre, tipo** y **loop,** así como los métodos de acceder a ellas. Al mandar el mensaje **tipo** obtendremos el tipo del regulador. Consultando la variable **loop** identificaremos el bucle de regulación. Para dar en nombre a el regulador tenemos la variable **nombre.**

4.4.13 La clase *ReguladorAdaptativo*

La clase **ReguladorAdaptativo** no es más que una clase abstracta para representar todos tipos de reguladores adaptativos. Se usa para el posible ampliación del programa.

La clase no tiene ninguna variable de instancia ni ningún método.

4.4.14 La clase *ISTR*

La clase **ISTR** (del inglés Indirect Self-Tuning Regulator) representa a un regulador auto-sintonizado indirecto.

Creamos una instancia al mandar el mensaje **new.**

La clase **ISTR** tiene las variables de instancia **currentEstimator, currentDesign, estimators** y **designs.** Los dos primeros tendrán un estimador, que será el estimador actual, y un diseño, que será el diseño actual. Los dos últimos tendrán una instancia de la clase *Dictionary* con otros posibles estimadores y diseños. Cambiamos el estimador y el diseño actual mediante los mensajes **currentEstimator:** y **currentDesign:**. Los obtenemos mediante los mensajes **currentEstimator** y **currentDesign.** Para añadir nuevos estimadores posibles o nuevos diseños posibles mandamos los mensajes **addEstimator:** y **addDesign:** y para borrarlos, mandamos los mensajes **removeEstimator** y **removeDesign.**

Al mandar el mensaje **computeOutputWithY:withUC:** obtendremos la variable de actuación y luego al mandar el mensaje **updateStateWithU:-**

withY: hacemos todos los cálculos. Los cálculos consiste en calcular primero los nuevos parámetros del proceso, después hacer un diseño del regulador y calcular todo que se puede de la variable de actuación.

4.4.15 La clase *Matrix*

La clase **Matrix** es una clase que representa una matriz.

Creamos una instancia al mandar el mensaje **rows:columns:** o el mensaje **dimension:**. El primero funciona para cualquier matriz. El segundo es un mensaje para crear más fácilmente una matriz cuadrada.

Asignamos valores a una matriz mediante el mensaje **assign:** donde el parámetro es un *array[array[elementos primera línea], array[elementos segunda línea], ...]*. También se puede especificar un elemento u obtener un elemento mediante los mensajes **atRow:atColumn:put:** y **atRow:atColumn:** respectivamente. Para que nos dará como resultado la matriz mandamos el mensaje **matrix.**

Se han definido los métodos +, −, *, **transpose, inverse** y **diagonal.** El inverso de una matriz obtendremos mediante una eliminación gaussiana. Lo hacemos numérico mediante el algoritmo *Gauss–Jordan*, que es un algoritmo numérico estable (vease [8]). Obtendremos un error de ejecución si tratamos de invertir una matriz singular.

4.4.16 La clase *Reciprocal*

La clase **Reciprocal** es una sub-clase de la clase *Polinomio* que representa un polinomio recíproco. Hereda métodos para sumar,restar, multiplicar y dividir polinomios de la clase *Polinomio.*

Creamos una instancia al mandar el mensaje **orden:** al objeto de la clase. Para asignar el valor de un coeficiente, mandamos el mensaje, **coefOrden:** y para obtener el coeficiente **coefOrden**

Para obtener el valor del polinomio recíproco en un punto enviaremos el mensaje **en:**.

Además, hay el método **shiftDown**, que quita el término directo y devuelve el polinomio recíproco con un grado menos.

4.4.17 La clase *SignalVector*

La clase **SignalVector** es una clase que representa un vector.

Creamos una instancia al mandar el mensaje **new:** al objeto de la clase. Luego podemos cambiar el tamaño mediante el mensaje **dimension:**

Usamos el vector en el presente proyecto para almacenar las señales en el regulador.

La clase tiene la variable de clase **MaxSignalVectorSize** que es el máximo número de señales que cada instancia pueda almacenar.

La clase tiene los métodos **shiftSignal** y **updateSignal:**. El primero cambia todos los elementos un paso a la derecha. El segundo hace lo mismo, pero como el primer elemento pone el parámetro del mensaje.

4.4.18 Las clases de los diálogos

Existe una clase para cada diálogo que tenemos. Los diálogos son sub-clases de la clase *DialogosSCM*. Tenemos las clases **DlgRegul**, **DlgAdaptativo**, **DlgEstimador** y **DlgDiseno**. La descripcion de cada una de éstas no es tan interesante y , por lo tanto, las veremos en una forma más general.

Podemos dibujar la ventana en un editor gráfico y luego añadir el control de la ventana en la clase que la representa. Existirá un método para cada botón que tengamos, el método se ejecutará al pulsar el botón. Hay el método **openOn** o **openOn:** para abrir el diálogo. La ventana estará abierta hasta que mandemos el mensaje **close** al *self*, es decir al pulsar el botón *aceptar* o el botón *cancelar*. Cuando la ventana está abierta , podemos cambiar valores de los parámetros dentro de ella.

4.5 Modificaciones de clases del sistema SCM

Veremos aquí las modificaciones que han sido necesario hacer en el sistema SCM. Los modificaciones han sido precisas por tres razones. Primero, para poder añadir el regulador adaptativo al sistema y crear el proceso encargado de regular la planta. Luego, para poder añadir el regulador adaptativo al interfaz a usuario del sistema y, por último se han requerido cambios en la jerarquía de clases para añadir los clases del regulador adaptativo y pocos cambios en unas clases para añadir sub-clases.

La clase **BucleControl** es una clase que representa un bucle de control, así tiene las variables de instancia **regulador**, **varControl**, **varReferencia** y **varActuacion**. En esta clase se ha añadido la variable de instancia **reguladoresOpcionales**. Esta variable tendrá una instancia de la clase *Dictionary* para representar los diferentes tipos de regualadores definidos para el bucle. Así, tendrá las llaves *pid* y *adaptativo*. La primera llave estará asociada a una instancia de la clase *Regulador*, la segunda llave estará asociada a una instancia de la clase *ISTR*. De esta manera será fácil la ampliación del programa con otros tipos de reguladores.

Los procesos encargados de regular se crean en la clase **SistemaControl**. Esta clase tiene, entre otros, los métodos **controldirecto** y **simulación**. En el modo control directo se ejecutará el primero y en el modo de simulación, el segundo. Se ha cambiado la implementación de estos dos métodos. Se hace una prueba para ver si el regulador del bucle es de tipo *pid* o de tipo *adaptativo*. En el caso de un regulador *pid*, creará un proceso para la regulación pid mediante el mensaje **pid** en modo control directo y **pid2** en el modo simulación. En el caso de un regulador *adaptativo*, creará un proceso

para la regulación adaptativo mediante el mensaje **adaptativo** en modo control directo y **adaptativo2** en el modo simulación.

Han sido necesarios también unos cambios para añadir la interfaz de usuario. Se ha cambiado el primer diálogo para especificar los parámetros de control desde el menú especificación de la ventana principal. Para hacer posible esto, se ha cambiado el método **eControl** de la clase **InterfazSCM** para abrir el nuevo diálogo y para añadir el regulador *PID* a la variable de instancia **reguladoresOpcionales** de la clase **BucleControl**.

Los últimos cambios han sido para cambiar la jeraquía de las clases. Asíi se han añadido las clases abstactas **ReguladoresPlanta** y **RegulatorTypes** (véanse las tablas 1 y 2). Tambien ha habido unos cambios en las clase **FdT**, que representa una funcion de transferencia discretizada, y en la clase **Polinomio** que representa un polinomio matemático.

5 Conclusiones

El problema al empezar fue implementar herramientas para el control adaptativo y añadirlas al sistema *SCM* con su interfaz de usuario. Así han sido implementadas varias clases para el control adaptativo y clases para su interfaz de usuario. Fue posible implementar las clases para representar las entidades de un regulador auto-sintonizado sin ningún conocimiento de donde irían a aplicarse. Este hecho de que podamos integrar los objetos a cualquier aplicación es uno de las ideas de la orientación a objetos. Para obtener esa buena cualidad, el progamador debe pensar bien cómo construir las clases para su posible ampliación futura. Esta es una parte muy difícil de la programación, pero la orientación a objetos y *Smalltalk* nos ayuda a hacerlo. Otro punto de importancia es tratar de comentar muy bien los métodos. Así será más fácil para otro programador usar las clases para otra aplicación. Siempre hace falta un buen conocimiento de un programa para añadir algo. Así para añadir las clases del regulador auto-sintonizado al sistema *SCM* resultó necesario un buen conocimiento del sistema *SCM* en cuanto a su interfaz y como regular con el sistema.

Han sido muchas las ideas que he tenido en cuanto al control adaptativo. Me habría gustado implementar otros estimadores para modelos estocásticos y otros tipos de diseños del regulador. También me habría gustado implementar ventanas para la monitorización de la estimación, donde pudiéramos ver la convergencia de los parámetros, aspectos importantes para los reguladores adaptativos. Además, hubiera querido tener posibilidades de cambiar los parámetros de estimación y diseño durante ejecución para más fácilmente probar las propiedades del sistema de control. Otra cosa importante sería el prefiltrado de las señales antes de aplicarlas el algoritmo de estimación. Pero como no tenía tanta experiencia de programación y ninguna experiencia de *Smalltalk*, se ha hecho más lento el proceso de desarrollo. Como siempre, nos condiciona la falta de tiempo.

Personalmente, pienso que este proyecto ha sido muy interesante. Además me ha gustado mucho conocer Valencia y los valencianos. Los conocimientos que he obtenido de la orientación a objetos y de *Smalltalk* pueden ser de gran utilidad para mi futuro profesional. Tambien la confrontación con el control adaptativo y escribir en el castellano ha sido una gran ayuda para aprender más. Aprendemos cuando nos enfrontamos con los problemas. Así, he mejorado mi castellano y he obtenido conocimientos más profundos del control adaptativo.

A Manual del usuario

En este apéndice se presenta como usar y especificar el regulador adaptativo con su estimador y diseño del regulador dentro del sistema SCM.

Para especificar el control se elige *control* desde el menú *Especificación* de la ventana principal del sistema SCM. Aparecerá el diálogo de la figura 3. Si se quiere definir un regulador adaptativo para algún sub-proceso, primero se elige el nombre y se pulsa el botón *adaptativo* y luego el botón *editar*. Si hay un regulador PID y un regulador adaptativo definido para el sub-proceso, no hace falta más que pulsar el botón *PID* o el botón *adaptativo* para cambiar el regulador. Cuando ya esta listo la especificación se pulsa el botón *aceptar* para verificar la especificación. Para cancelar los cambios se pulsa el botón *cancelar*.

En la figura 4 veremos el diálogo que aparecerá para especificar el control adaptativo. Primero se nesecita especificar el orden y el exceso de polos del modelo del proceso. Hay posibilidades de tener varios estimadores y varios diseños. Para especificar un estimador o un diseño nuevo se elecciona *nuevo* en la lista de estimadores o en la lista de diseños y se pulsa el botón *editar*. Para borrar un estimador o un diseño que ya no quiera se pulsa el botón *borrar*. Si se quiere editar un estimador o un diseño ya existente lo elecciona y presiona el botón *editar*. La referencia se puede especificar de forma interactiva o de forma predefinida en un fichero. Si se quiere especificar una referencia hay un editor de referencia (véase [7]). El regulador adaptativo consiste en un estimador y un diseño. Se puede cambiar el estimador y el diseño actual dentro del rectangulo *Regulador adaptativo actual* de la ventana.

En la figura 5 vemos la ventana donde se puede especificar un diseño para un regulador y el regulador. El diseño debe tener un nombre. Se puede tener un diseño donde todos los ceros del proceso serán cancelados o un diseño que no cancelará ningún cero. Se puede especificar un diseño con o sin acción integral. El regulador se puede especificar con o sin *anti windup*. Los polos y ceros se deben especificar en formas discretizadas. Para el caso sin cancelación de ceros se especificarán los polos de los polinomios A_m y A_o. En el caso con cancelación de ceros, además de estos se especificará los ceros del polinomio B_m. Para especificar un polo o un cero, se elige el polo o cero que se quiere editar y escribe el valor. Luego para verificar el cambio se pulsa el botón *OK*. Todos los polos y ceros deben ser conjugados.

En la figura 6 vemos la ventana donde se puede especificar un estimador. Hace falta dar un nombre al estimador. Se puede elegir entre dos tipos de estimadores. Primero un *Rls* que es un algoritmo recursivo de mínimos cuadrados. Segundo un *algoritmo de Proyección*. El primero será el más útil y en él hacen falta especificar las varianzas iniciales de los parámetros y el factor de olvido λ. El factor de olvido debe estar entre cero y uno. Por defecto las varianzas iniciales se inizializarán a 1000 y el factor de olvido

se inizializará a 1. El segundo, el algoritmo de proyección, es para el caso cuando el tiempo de computación es critico. En él hace falta especificar la ganancia γ y α. La ganancia debe estar entre cero y dos y α debe ser mayor que cero para evitar una división con cero. Por defecto γ se inizializará a 1 y α se inizializará a 0.1.

B Samples of class definitions in Smalltalk

B.1 Definition of class *Estimator*

```
ObjetosBasicos subclass: #Estimator
  instanceVariableNames:
    ' name type order delay parameters regressor estimatorOrder delayFilter '
  classVariableNames: ''
  poolDictionaries: ''   !

!Estimator class methods !

order: anIntegerA delay: anIntegerB
     " Initializes a estimator of order anIntegerA and delay anIntegerB "

     | aEstimator |
     (anIntegerA isInteger) & (anIntegerB isInteger)
     ifTrue: [
        aEstimator := self new.
        aEstimator order: anIntegerA.
        aEstimator delay: anIntegerB.
        ^aEstimator]! !

!Estimator methods !

a
     " Returns the estimated denominator  A polynomial as reciprocal"

     | aReciprocal |
     aReciprocal := Reciprocal grado: order.
     aReciprocal coefOrden: 0 es: 1.
     1 to: order do: [:i |
        aReciprocal coefOrden: i es: (self parameters at: i)].
     ^aReciprocal!

b
     " Returns the estimated numerator, B polynomial as reciprocal"

     | aReciprocal |
     aReciprocal := Reciprocal grado: order.
     0 to: delay - 1do: [:i |
        aReciprocal coefOrden: i es: 0].
     delay  to: order do: [:i |
        aReciprocal coefOrden: i es: (self parameters at: order - delay + 1 + i)].
     ^aReciprocal!

delay
     " Returns the delay of the estimatormodel "

      ^delay!

delay: anInteger
     " Sets the delay of the estimator model to anInteger"

     anInteger isInteger
     ifFalse: [^self].
     (order isNil)
        ifTrue: [
```

```
        delay := anInteger.
        ^self].
        delay := anInteger.
        estimatorOrder := 2 * order - delay + 1.
        self initialize!

initialize
     " Initializes  regressor and parameters. "
     " Sets the initial values of the parameters to 0.9    "
     " Initializes a delay filter that helps us to update the regressor "

     | aArray |
     regressor := Matrix rows: estimatorOrder columns: 1.
     parameters := Matrix rows: estimatorOrder columns: 1.
     1 to: estimatorOrder do: [:i |
        parameters atRow: i atColumn: 1 put: 0.9].
     delayFilter := FdT orden: delay.
     aArray := Array new: delay + 1.
     1 to: delay+1 do: [:i | aArray at: i put:0].
     aArray at: 1 put: 1.
     delayFilter coeficientesDen: aArray.
     aArray := Array new: delay + 1.
     1 to: delay+1 do: [:i | aArray at: i put:0].
     aArray at: delay +1 put: 1.
     delayFilter coeficientesNum: aArray!

name
     " Returns the estimators name"

     ^name!

name: aString
     " Gives the estimator a name"

     aString isString
     ifTrue: [name := aString]!

order
     " Returns the order of the estimator model "

      ^order!

order: anInteger
     " Sets the order of the estimator model to anInteger"

     anInteger isInteger
     ifFalse: [^self].
     (delay isNil)
        ifTrue: [
        order := anInteger.
        ^self].
        order := anInteger.
        estimatorOrder := 2 * order - delay + 1.
        self initialize!

parameters
     " Returns the parameters as anArray  "
```

```
    ^parameters asArray!

parameters: aMatrix
    " Sets the parameters  to aMatrix[estimatedParameters * 1]. "

    (aMatrix rows = estimatorOrder) & (aMatrix columns = 1)
    ifTrue: [parameters := aMatrix]!

regressor
    " Returns the regressor as anArray  "

    ^regressor asArray!

regressor: aMatrix
    " Sets the regressor  to aMatrix[estimatedParameters * 1]. "

    (aMatrix rows = estimatorOrder) & (aMatrix columns = 1)
    ifTrue: [regressor := aMatrix]!

updateRegressorWithU: u withY: y
    " Updates the regressor "

    estimatorOrder to: 2 by: 1 negated do: [:i |
    (regressor matrix at: i) at: 1 put: ((regressor matrix at: (i - 1)) at: 1)].
    (regressor matrix at: 1) at:1 put: y negated.
    (regressor matrix at: (order + 1)) at:1 put:  (delayFilter evalua: u)! !
```

B.2 Definition of class *Rls*

```
Estimator subclass: #Rls
  instanceVariableNames:
    ' pMatrix lambda defaultInitVariances initVariances '
  classVariableNames: ''
  poolDictionaries: ''  !

!Rls class methods !

order: anIntegerA delay: anIntegerB
    " Creates and initializes a Recursive Least Square estimator "
    " with the estimatormodel: order anIntegerA and delay anIntegerB "

    (anIntegerA isInteger) & (anIntegerB isInteger)
    ifTrue: [^(super order: anIntegerA delay: anIntegerB) initialize]! !

!Rls methods !

defaultInitVariances
    " Returns the defaultInitVariances "

    ^defaultInitVariances!

diagonal
    " Returns the diagonalelements of the pMatrix.                    "
    " Result is given in aMatrix[rows * 1]  "

    | aMatrix |
```

```
     aMatrix := Matrix rows: estimatorOrder columns: 1.
     aMatrix := pMatrix diagonal!

diagonal: aMatrix
     " Resets  the pMatrix                                                    "
     " Sets the diagonalelements to aMatrix, with aMatrix[rows * 1] "

     (aMatrix rows = pMatrix rows) & (aMatrix columns = 1)
     ifTrue: [pMatrix reset; diagonal: aMatrix]!

initialize
     " Initializes a Recursice Least Square estimator "
     " Sets the diagonalelements to aMatrix, with aMatrix[anInteger * 1]"
     " with diagonalelements 1000 as default value "
     " Sets the forgettingfactor lambda to 1 "

     | aMatrix variances |
     defaultInitVariances := 1000.
     variances := defaultInitVariances.
     initVariances isNil ifFalse: [variances := initVariances].
     self pMatrix: estimatorOrder.
     aMatrix := Matrix rows: estimatorOrder columns: 1.
     1 to: estimatorOrder do: [:i |
        aMatrix atRow: i atColumn: 1 put: variances].
     self diagonal: aMatrix.
     self lambda: 1.
     super initialize!

initVariances
     " Returns the initVariances "

     ^initVariances!

initVariances: aNumber
     " Sets the initVariances to aNumber "

     aNumber isNumber
     ifTrue: [initVariances := aNumber].!

lambda
      " Returns the forgettingfactor lambda "

      ^lambda!

lambda: aNumber
     " Sets the forgettingfactor lambda to aNumber between 0 and 1 "

     (aNumber > 0) & (aNumber <= 1)
     ifTrue: [lambda := aNumber]!

pMatrix
     " Returns the pMatrix "

     | aMatrix |
     aMatrix := Matrix dimension: estimatorOrder.
     aMatrix := pMatrix deepCopy.
     ^aMatrix!
```

```
pMatrix: anInteger
     " Sets the dimension of the pMatrix to anInteger * anInteger  "

      anInteger isInteger
      ifTrue: [pMatrix := Matrix dimension: anInteger]!

reset

    | lambd |
    lambd := self lambda.
    self initialize.
    self lambda: lambd.!

type
     " Returns the type of Estimator as a string 'rls' "

    ^'rls'!

updateParameters: y
     " Updates the parameters using a RLS -algoritm "

     | eps den k  |
     eps := y - (regressor transpose * parameters).
     k := pMatrix * regressor.
     den := lambda + (k transpose * regressor).
     parameters := parameters +  (k * (eps / den)).
     pMatrix := pMatrix - (k * k transpose * (1 / den)).
     pMatrix := pMatrix * ( 1 / lambda)! !
```

B.3 Definition of class *Matrix*

```
Funcion subclass: #Matrix
  instanceVariableNames:
    ' matrix rows columns '
  classVariableNames: ''
  poolDictionaries: ''   !

!Matrix class methods !

dimension: anInteger
     " Returns a new matrix of dimension anInteger*anInteger  "

     | aMatrix |
     anInteger isInteger
     ifTrue: [
        aMatrix := self new.
        aMatrix dimension: anInteger.
     ^aMatrix ]!

rows: anIntegerA columns: anIntegerB
  " Returns a new matrix of dimension anIntegerA*anIntegerB "

     | aMatrix |
     (anIntegerA isInteger) & (anIntegerB isInteger)
     ifTrue: [
        aMatrix := self new.
```

```
        aMatrix rows: anIntegerA columns: anIntegerB].
    ^aMatrix! !

!Matrix methods !

* anObject
    " Multiplies reciever matrix with anObject,
      where anObject can be eather aNumber or aMatrix.
      If anObject is aMatrix there are 4 cases available:
      1. Scalar product matrix[1* columns] * aMatrix[rows * 1].
      The result will be a scalar.
      2. Tensor product matrix[rows * 1] * aMatrix[1 * columns].
      The result will be aMatrix[rows * columns].
      3. Multiplication matrix[rows * columns] * aMatrix[rows * columns].
      The  result will be in aMatrix[rows * columns].
      4. Multiplication matrix[rows * columns] *aMatrix[rows * 1].
      Thr result will be in aMatrix[rows * 1].      "

    | sum aMatrix |
    (anObject isKindOf: Number)
    ifTrue: [
       1 to: rows do: [:i |
          1 to: columns do: [:j |
             (matrix at: i) at: j put:(((matrix at:i) at:j) * anObject)]]].
    (anObject isMatrix)
    ifFalse: [^self].

    " scalar product "
    (rows = 1) & (anObject columns = 1) & (columns = anObject rows)
    ifTrue: [
        sum := 0.
        1 to: columns do: [:i |
            sum :=
            sum + (((matrix at: 1) at: i) * (anObject atRow: i atColumn: 1))].
        ^sum].

    " tensor product "
    (columns = 1) & (anObject rows = 1) & (rows = anObject columns)
    ifTrue: [
       aMatrix := self class rows: rows columns: anObject columns.
       1 to: rows do: [:i |
          1 to: anObject columns do: [:j |
             aMatrix atRow: i atColumn: j put:
             (((matrix at:i) at:1) * (anObject atRow: 1 atColumn: j))]].
       ^aMatrix].

     " matrix multiplication; must be of same dimension "
     (rows = anObject rows) & (columns = anObject columns)
    ifTrue: [
       aMatrix := self class rows: rows columns: columns.
       1 to: rows do: [:i |
          1 to: columns  do: [:j |
             1 to: columns  do: [:k |
                 aMatrix atRow: i atColumn: j put:
                 ((aMatrix atRow: i atColumn: j) +
                 (((matrix at: i) at: k) * (anObject atRow: k atColumn: j)))]].
       ^aMatrix].
```

```
    " matrix ( rows * columns ) multiplied by anObject ( rows * 1) "
    (rows = anObject rows) & (rows = columns) & (anObject columns = 1)
    ifTrue: [
       aMatrix := self class rows: rows columns: 1 .
       1 to: rows do: [:i |
          1 to: columns do: [:j |
             aMatrix atRow: i atColumn: 1 put:
             (aMatrix atRow: i atColumn: 1) +
             (((matrix at: i) at: j) * (anObject atRow: j atColumn: 1))]].
    ^aMatrix]!

+ xMatrix
    " Adds xMatrix to matrix. Result is given in aMatrix "

    | aMatrix |
    (xMatrix isMatrix) & (rows = xMatrix rows) &
    (columns = xMatrix columns)
    ifTrue: [
       aMatrix := self class rows: rows columns: columns.
       1 to: rows do: [:i |
           1 to: columns do: [:j |
              aMatrix atRow: i atColumn: j put:
              (((matrix at: i) at:j) + (xMatrix atRow: i atColumn: j))]].
    ^aMatrix]!

- xMatrix
    " Subtractes xMatrix from matrix. Result is given in aMatrix "

    | aMatrix |
    (xMatrix isMatrix) & (rows = xMatrix rows) &
    (columns = xMatrix columns)
    ifTrue: [
       aMatrix := self class rows: rows columns: columns.
       1 to: rows do: [:i |
           1 to: columns do: [:j |
              aMatrix atRow: i atColumn: j put:
              ((matrix at: i) at:j) - (xMatrix atRow: i atColumn: j)]].
    ^aMatrix]!

asArray
    " Converts aMatrix[rows * 1] or aMatrix[1* columns] to anArray"

    | aArray |
    columns = 1
       ifTrue: [
          aArray := Array new: rows.
          1 to: rows do: [:i |
             aArray at: i put: ((matrix at: i) at:1)].
    ^aArray].
    rows = 1
       ifTrue: [
          aArray := Array new: columns.
          1 to: columns do: [:i |
             aArray at: i put: ((matrix at: 1) at: i)].
    ^aArray]!
```

```
assign: aArray
     " Assign values to matrix "

     | sum |
     sum := 0.
     1 to: aArray size do: [:i |  sum := sum + (aArray at: i) size ].
     (rows = aArray size) & (columns = (sum / rows))
     ifTrue: [matrix := aArray]!

atRow: anIntegerA atColumn: anIntegerB
     " Returns matrix element matrix[row,column] "

     (anIntegerA <= rows) & (anIntegerB <= columns)
     ifTrue: [^(matrix at: anIntegerA) at: anIntegerB]!

atRow: anIntegerA atColumn: anIntegerB put: aNumber
     " Sets matrix element matrix[row,column] to aNumber "

    (aNumber isNumber)  & (anIntegerA <= rows) & (anIntegerB <= columns)
     ifTrue: [(matrix at: anIntegerA) at: anIntegerB put: aNumber]!

columns
    " Returns the number of columns in matrix  "

     ^columns!

diagonal
     " Returns the diagonal elements in quadratic matrix."
     " The result is given in a [rows*1] matrix "

     | aMatrix |
     aMatrix := self class rows: rows columns: 1.
     1 to: rows do: [:i | aMatrix atRow: i atColumn: 1 put: ((matrix at: i) at:i)].
     ^aMatrix!

diagonal: aMatrix
     " Sets the diagonal elements in quadratic matrix."
     " aMatrix should be given as a [rows*1] matrix "

      (rows = aMatrix rows) & (1 = aMatrix columns )
      ifTrue: [
         1 to: rows do: [:i |
            (matrix at: i) at: i put: (aMatrix atRow: i atColumn: 1)]]!

dimension
    " Returns the dimension of  matrix  "

     | aArray |
     aArray := Array new:2.
     aArray at: 1 put: rows.
     aArray at: 2 put: columns.
     ^aArray!

dimension: anInteger
    " Sets the dimension of matrix to aninteger*anInteger  "

     anInteger isInteger
```

```
        ifTrue: [ self rows: anInteger columns: anInteger]!

gaussElimination: aMatrix
    " Solvs Ax=B, where A is Reciever matrix [rows * columns] and B "
    " is aMatrix[rows * columns] . Returns x as aMatrix[rows * columns] ."
    " Reciever matrix will contain the inverse of A "
    " Performs gauss elimination through Gauss - Jordans algoritm "

     |n m indxr indxc ipiv icol irow big dum pivinv  temp|
     n := rows.
     m := aMatrix columns.
     indxr := Array new: n.
     indxc :=  Array new: n.
     ipiv :=  Array new: n.
     1 to: n do: [:j | ipiv at: j put: 0].
     1 to: n do: [:i |
        big := 0.
        1 to: n do: [:j |
           (ipiv at: j) ~= 1
              ifTrue: [
                 1 to: n do: [:k |
                     (ipiv at: k) = 0
                        ifTrue: [
                           (self atRow: j atColumn: k) abs >= big
                              ifTrue: [
                                 big := (self atRow: j atColumn: k) abs.
                                 irow := j.
                                 icol := k].
                        ]  " ifTrue"
                        ifFalse: [
                           (ipiv at: k) > 1
                              ifTrue: [self error: 'Singular Matrix' ].
                        ]. "  ifFalse "
                 ].  " k "
              ]. " ifTrue "
        ]. " j "
        ipiv at: icol  put: (ipiv at: icol) + 1.
        irow ~= icol
           ifTrue: [
              1 to: n do: [:l |
                  temp := self atRow: irow atColumn: l.
                  self atRow: irow atColumn: l put: (self atRow: icol atColumn: l).
                  self atRow: icol atColumn: l put: temp].
              1 to: m do: [:l |
                  temp := aMatrix atRow: irow atColumn: l.
                  aMatrix atRow: irow atColumn: l put:
                  (aMatrix atRow: icol atColumn: l).
                  aMatrix atRow: icol atColumn: l put: temp].
           ]. "  ifTrue "
       indxr at: i put: irow.
       indxc at: i put: icol.
       (self atRow: icol atColumn: icol) = 0
          ifTrue: [self error: 'Singular Matrix'].
       pivinv := 1 / (self atRow: icol atColumn: icol).
       self atRow: icol atColumn: icol put: 1.
       1 to: n do: [:l |
            self atRow: icol atColumn: l put:
```

```
            pivinv * (self atRow: icol atColumn: 1)].
        1 to: m do: [:l |
            aMatrix atRow: icol atColumn: l put:
            pivinv * (aMatrix atRow: icol atColumn: 1)].
        1 to: n do: [:ll |
           ll ~= icol
              ifTrue: [
                  dum := self atRow: ll atColumn: icol.
                  self atRow: ll atColumn: icol put: 0.
                  1 to: n do: [:l |
                     self atRow: ll atColumn: l put:
                     ((self atRow: ll atColumn: l) -
                     ((self atRow: icol atColumn: 1) * dum)) ].
                  1 to: m do: [:l |
                     aMatrix atRow: ll atColumn: l put:
                     ((aMatrix atRow: ll atColumn: l) -
                     ((aMatrix atRow: icol atColumn: 1) * dum)) ].
              ]. " ifTrue "
         ]. " ll "
      ]. " i "
      n to: 1 by: 1 negated do: [:l |
         (indxr at: l ) ~= (indxc at: l)
            ifTrue: [
                1 to: n do: [:k |
                    temp := self atRow: k atColumn: (indxr at: l).
                    self atRow: k atColumn: (indxr at: l) put:
                    (self atRow: k atColumn: (indxc at: l )).
                    self atRow: k atColumn: (indxc at: l) put: temp].
            ]. " ifTrue "
      ]. " l "
      ^aMatrix!

inverse
    " Returns the inverse of reciever matrix "
    " Performs gauss elimination through Gauss - Jordans algoritm "

     | a b n m |
     n := rows.
     m := 1.
     a := self class dimension: n.
     a assign: self matrix.
     b := self class rows: n columns: m.
     a gaussElimination: b.
     ^a!

isMatrix
    " Returns true if reciever is of class Matrix
      or any subclass to class Matrix. Else returns false "

    ^true!

matrix
    " Returns matrix "

     ^matrix!

reset
```

```
     " Sets all matrix elements to zero "

     1 to: rows do: [:i |
        1 to: columns do: [:j |
         (matrix at:i) at: j put: 0]]!

rows
    " Returns the number of rows in matrix  "

     ^rows!

rows: anIntegerA columns: anIntegerB
    " Sets the dimension of matrix to anIntegerA*anIntegerB  "

     (anIntegerA isInteger) & (anIntegerB isInteger)
         ifTrue: [
             rows := anIntegerA.
             columns:= anIntegerB.
             matrix := Array new: rows.
             1 to: rows do: [:i |
                matrix at:i put: (Array new: columns)].
              self reset].!

transpose
     "Returns the transpose of matrix. Result is given in aMatrix "

     | aMatrix |
     aMatrix := self class rows: columns columns: rows.
     1 to: rows do: [:i |
        1 to: columns do: [:j  |
           aMatrix atRow: j atColumn: i put:((matrix at: i) at: j)]].
     ^aMatrix! !
```

B.4 Definition of a class representing a dialogue, *DlgDiseno*

```
DialogosSCM subclass: #DlgDiseno
  instanceVariableNames:
    ' noCancel allCancel listaAm listaAo listaBm respuesta rst rstWindUp '
  classVariableNames:
    ' ItemIds '
  poolDictionaries:
    ' PMConstants '   !

!DlgDiseno class methods !

initItemIds
   ItemIds:=Dictionary new.
   ItemIds
     at: 'name' put: 103;
     at: 'textBm' put: 108;
     at: 'listaAo' put: 117;
     at: 'listaBm' put: 118;
     at: 'listaAm' put: 107;
     at: 'numberAo' put: 105;
     at: 'numberBm' put: 106;
     at: 'numberAm' put: 102;
     at: 'bNoCancel' put: 112;
```

```
     at: 'bAllCancel' put: 113;
     at: 'bWindUp' put: 120;
     at: 'bNoWindUp' put: 121;
     at: 'bSiIntegral' put: 115;
     at: 'bNoIntegral' put: 116;
     at: 'entryAm' put: 122;
     at: 'entryAo' put: 124;
     at: 'entryBm' put: 126;
     at: 'okBm' put: 127;
     at: 123 put: #okAm;
     at: 125 put: #okAo;
     at: 127 put: #okBm;
     at: 107 put: #writeListAm;
     at: 117 put: #writeListAo;
     at: 118 put: #writeListBm;
     at: 112 put: #noCancel;
     at: 113 put: #allCancel;
     at: 115 put: #siIntegral;
     at: 116 put: #noIntegral;
     at: 120 put: #siWindUp;
     at: 121 put: #noWindUp;
     at: 263 put: #aceptar;
     at: 264 put: #cancelar.!

new

     ^super new initialize.! !


!DlgDiseno methods !

aceptar
     " Accepts all edits made to the design.
       Checks if defined poles and zeros are conjugated. "

     | name amList bmList aoList complexed |
     name := self queryItemText: (ItemIds at: 'name').
     name = ''
       ifTrue: [
          MessageBox notify: '' withText: 'Nombre no definido'.
          ^nil].
     respuesta name: name.
     amList := OrderedCollection new.
     1 to: listaAm size do: [:i |
         amList add: (listaAm at: i) asComplex].
     complexed := amList select: [:i | i isComplex].
     (self isConjugated: complexed)
        ifFalse: [
           MessageBox notify:'' withText:
           'Los polos del polinómio Am deben ser conjugados.'.
           ^nil].
     aoList := OrderedCollection new.
     1 to: listaAo size do: [:i |
         aoList add: (listaAo at: i) asComplex].
     complexed := aoList select: [:i | i isComplex].
     (self isConjugated: complexed)
```

```
        ifFalse: [
           MessageBox notify:'' withText:
           'Los polos del polinómio Ao deben ser conjugados.'.
           ^nil].
     respuesta type = 'allCancel'
        ifTrue: [
           bmList := OrderedCollection new.
           1 to: listaBm size do: [:i |
              bmList add: (listaBm at: i) asComplex].
          complexed := bmList select: [:i | i isComplex].
          (self isConjugated: complexed)
              ifFalse: [
                 MessageBox notify:'' withText:
                 'Los polos del polinómio Bm deben ser conjugados.'.
           ^nil].
           respuesta bmZeros: bmList].
        respuesta amPoles: amList.
        respuesta aoPoles: aoList.
     self close.!

allCancel
     "Change to a design that cancels all process ceros "

     self showItem: (ItemIds at: 'listaBm').
     self showItem: (ItemIds at: 'numberBm').
     self showItem: (ItemIds at: 'textBm').
     self showItem: (ItemIds at: 'okBm').
     self showItem: (ItemIds at: 'entryBm').
     respuesta := allCancel.
     self setButton: (ItemIds at: 'bNoCancel') value: false.
     self setButton: (ItemIds at: 'bAllCancel') value: true.
     self updateLists.!

cancelar
     "Cancels all edits and closes the dialogue"

     respuesta name: ''.
     self close.!

initialize

     listaAm := OrderedCollection new.
     listaBm := OrderedCollection new.
     listaAo := OrderedCollection new.!

initLists

     1 to: allCancel degBm do: [:i | listaBm add: '0,0 j0,0'].
     respuesta type = 'allCancel'
        ifTrue: [
           1 to: allCancel degAo do: [:i | listaAo add: '0,0 j0,0'].
           1 to: allCancel degAm do: [:i | listaAm add: '0,0 j0,0'].].
     respuesta type = 'noCancel'
        ifTrue: [
           1 to: noCancel degAo do: [:i | listaAo add: '0,0 j0,0'].
           1 to: noCancel degAm do: [:i | listaAm add: '0,0 j0,0'].].!
```

```
isConjugated: aCollection
      " Checks if all poles in aCollection are conjugated "

     | found |
     1 to: aCollection size do: [:i |
         found := false.
         1 to: aCollection size do: [:j |
             ((aCollection at: i) isConjugate: (aCollection at: j))
                ifTrue: [found := true].
         ].
         found
            ifFalse: [^false].
     ].
     ^true!

itemIds
     ^ItemIds.!

noCancel
     "Change to a design that does not cancel any  process cero "

     self hideItem: (ItemIds at: 'listaBm').
     self hideItem: (ItemIds at: 'numberBm').
     self hideItem: (ItemIds at: 'textBm').
     self hideItem: (ItemIds at: 'okBm').
     self hideItem: (ItemIds at: 'entryBm').
     respuesta := noCancel.
     self setButton: (ItemIds at: 'bNoCancel') value: true.
     self setButton: (ItemIds at: 'bAllCancel') value: false.
     self updateLists.!

noIntegral
     "Change to a design without integral action "

     respuesta integrator: 0.
     allCancel integrator: 0.
     noCancel integrator: 0.
     self setButton: (ItemIds at: 'bSiIntegral') value: false.
     self setButton: (ItemIds at: 'bNoIntegral') value: true.
     self updateLists.!

noWindUp
     "Change to a regulator without anti wind-up"

     respuesta regulator: rst.
     allCancel regulator: rst.
     noCancel regulator: rst.
     self setButton: (ItemIds at: 'bWindUp') value: false.
     self setButton: (ItemIds at: 'bNoWindUp') value: true.!

okAm
     "Accepts a edited pole and place it in the list"

     | new position |
     position:=(self querySelectionInListBox: (ItemIds at: 'listaAm')).
     position=LitNone ifTrue: [ ^nil ].
     new := self queryItemText: (ItemIds at: 'entryAm').
```

```
    listaAm at: position + 1 put: new.
    self setItemText: (ItemIds at: 'entryAm') string: ''.
    self showLists.!

okAo
    "Accepts a edited pole and place it in the list"

    | new position |
    position:=(self querySelectionInListBox: (ItemIds at: 'listaAo')).
    position=LitNone ifTrue: [ ^nil ].
    new := self queryItemText: (ItemIds at: 'entryAo').
    listaAo at: position +1 put: new.
    self setItemText: (ItemIds at: 'entryAo') string: ''.
    self showLists.!

okBm
    "Accepts a edited pole and place it in the list"

    | new position |
    position:=(self querySelectionInListBox: (ItemIds at: 'listaBm')).
    position=LitNone ifTrue: [ ^nil ].
    new := self queryItemText: (ItemIds at: 'entryBm').
    listaBm at: position + 1 put: new.
    self setItemText: (ItemIds at: 'entryBm') string: ''.
    self showLists.!

openOn: anArgument
    "Opens the dialoguewindow and initializes it"

    | order poleExcess integrator |
    self fromResFile: 'dlgdisen.res'.
    respuesta := anArgument deepCopy.
    rst := Rst new.
    rstWindUp := RstAntiWindUp new.
    respuesta class = Dictionary
    ifTrue: [                          "A new design "
       allCancel := AllZeroCancellation order: (respuesta at: #order)
                                  poleExcess: (respuesta at: #delay) integrator: 0.
       noCancel := NoZeroCancellation order: (respuesta at: #order)
                                  poleExcess: (respuesta at: #delay) integrator: 0.
       allCancel name: ''.
       noCancel name: ''.
       self noCancel.  "respuesta := noCancel "
       self initLists.
       self noIntegral.
       self siWindUp.
    ]
    ifFalse: [                         "A allready defined design"
       order := respuesta  order.
       poleExcess := respuesta poleExcess.
       integrator := respuesta integrator.
       self setItemText: (ItemIds at: 'name') string: respuesta name.
       1 to: respuesta amPoles size do: [:i |
          listaAm add: (respuesta amPoles at: i) printString].
       1 to: respuesta aoPoles size do: [:i |
          listaAo add: (respuesta aoPoles at: i) printString].
       respuesta type = 'noCancel'
```

```
            ifTrue: [
               noCancel := respuesta deepCopy.
               allCancel := AllZeroCancellation order: order
                                          poleExcess: poleExcess
                                          integrator: integrator.
               allCancel name: ''.
               self noCancel.
            ].
         respuesta type = 'allCancel'
            ifTrue: [
               1 to: respuesta bmZeros size do: [:i |
                  listaBm add: (respuesta bmZeros at: i) printString].
               allCancel := respuesta deepCopy.
               noCancel := NoZeroCancellation order: order
                                       poleExcess: poleExcess
                                    integrator: integrator.
               self allCancel.
               noCancel name: ''.
            ].
         respuesta regulator type = 'rst'
            ifTrue: [self noWindUp]
            ifFalse: [self siWindUp].
         integrator = 1
            ifTrue: [self siIntegral]
            ifFalse: [self noIntegral].
      ]. " ifFalse "
      self showWindow.
      self processInput.
      ^respuesta.!

showLists

      self deleteAllItemsInListBox: (ItemIds at: 'listaAm').
          listaAm do: [ :a |
             self add: a toListBox: (ItemIds at: 'listaAm') at: LitEnd
          ].
      self deleteAllItemsInListBox: (ItemIds at: 'listaAo').
      listaAo do: [ :a |
         self add: a toListBox: (ItemIds at: 'listaAo') at: LitEnd
      ].
      self deleteAllItemsInListBox: (ItemIds at: 'listaBm').
      listaBm do: [ :a |
         self add: a toListBox: (ItemIds at: 'listaBm') at: LitEnd
      ].!

siIntegral
      "Change to a design with integral action"

      respuesta integrator: 1.
      allCancel integrator: 1.
      noCancel integrator: 1.
      self setButton: (ItemIds at: 'bSiIntegral') value: true.
      self setButton: (ItemIds at: 'bNoIntegral') value: false.
      self updateLists.!

siWindUp
       "Change to a regulator with anti wind-up"
```

```
    respuesta regulator: rstWindUp.
    allCancel regulator: rstWindUp.
    noCancel regulator: rstWindUp.
    self setButton: (ItemIds at: 'bWindUp') value: true.
    self setButton: (ItemIds at: 'bNoWindUp') value: false.!

updateLists

    | list |
    respuesta type = 'allCancel'
       ifTrue: [
          self setItemText: (ItemIds at: 'numberAm') string: allCancel degAm printString.
          self setItemText: (ItemIds at: 'numberAo') string: allCancel degAo printString.
          self setItemText: (ItemIds at: 'numberBm') string: allCancel degBm printString.
          allCancel degBm > listaBm size
             ifTrue: [
                (allCancel degBm - listaBm size) timesRepeat: [listaBm add: '0,0 j0,0'].]
             ifFalse: [
                list := listaBm copyFrom: 1 to: allCancel degBm.
                listaBm := list].
          allCancel degAo > listaAo size
             ifTrue: [
                (allCancel degAo - listaAo size) timesRepeat: [listaAo add: '0,0 j0,0'].]
             ifFalse: [
                list := listaAo copyFrom: 1 to: allCancel degAo.
                listaAo := list].
       ].
    respuesta type = 'noCancel'
       ifTrue: [
          self setItemText: (ItemIds at: 'numberAm') string: noCancel degAm printString.
          self setItemText: (ItemIds at: 'numberAo') string: noCancel degAo printString.
          noCancel degAo > listaAo size
             ifTrue: [
                (noCancel degAo - listaAo size) timesRepeat: [listaAo add: '0,0 j0,0'].]
             ifFalse: [
                list := listaAo copyFrom: 1 to: noCancel degAo.
                listaAo := list.].
       ].
    self showLists.!

writeListAm

    | position selected |
   position:=(self querySelectionInListBox: (ItemIds at: 'listaAm')).
   position=LitNone ifTrue: [ ^nil ].
   selected := listaAm at: position + 1.
   self setItemText: (ItemIds at: 'entryAm') string: selected.!

writeListAo

    | position selected |
   position:=(self querySelectionInListBox: (ItemIds at: 'listaAo')).
   position=LitNone ifTrue: [ ^nil ].
   selected := listaAo at: position + 1.
   self setItemText: (ItemIds at: 'entryAo') string: selected.!
```

```
writeListBm

     | position selected |
    position:=(self querySelectionInListBox: (ItemIds at: 'listaBm')).
    position=LitNone ifTrue: [ ^nil ].
    selected := listaBm at: position + 1.
    self setItemText: (ItemIds at: 'entryBm') string: selected.! !
```

Bibliografía

[1] Karl Johan Åström y Björn Wittenmark. *Sistemas controlados por computador.* Paraninfo, Madrid, 1988. En inglés *Computer Controlled Systems—Theory and design.* Prentice–Hall, Englewood Cliffs, New Jersey, second edition, 1990

[2] Karl Johan Åström y Björn Wittenmark. *Adaptive Control.* Addison–Wesley, 1989. Hay una segunda edición, Addison–Wesley, Reading, Massachusetts, second edition, 1995.

[3] Rolf Isermann, K.–H. Lachmann, D. Matko *Adaptive Control Systems.* Prentice Hall, 1992.

[4] Göran Eriksson. *En introduktion till Objectworks/Smalltalk (en sueco).* Lund 1994.

[5] *Manuales de Smalltalk/V.* Digitalk.

[6] Sergio M. Fernández Sastre. *Fundamentos del diseño y la programación orientado a objetos.* McGraw-Hill, Madrid, 1995.

[7] Jorge Bondia Company. *Entorno de desarrollo y simulación de sistemas de control de procesos industriales.* Proyecto de fin de carrera, Universidad Politécnica de Valencia, 1994.

[8] William H. Press, Saul A Teukolsky, William T. Vetterling, Brian P.Flannery. *Numerical Recipies in C.* Second edition, 1992.