Universidad de Valladolid

ESCUELA DE INGENIERÍAS INDUSTRIALES

UNIVERSIDAD DE VALLADOLID

ESCUELA DE INGENIERIAS INDUSTRIALES

Grado en Ingeniería Electrónica Industrial y Automática

Título del TFG

Desarrollo de algoritmos de control avanzado para dispositivos programables industriales

Autor:

De Frutos Bolzoni, Laura

Tutor:

Mazaeda Echevarría, Rogelio
Martí Martínez, Rubén
Ingeniería de Sistemas y Automática

Valladolid, Septiembre de 2015

Resumen:

El objetivo del proyecto es el desarrollo de una aplicación de auto-sintonía de controladores PID para el CERN (Organización Europea para la Investigación Nuclear). La herramienta diseñada deberá contener diferentes métodos de sintonía que puedan cubrir diferentes problemáticas. Por ejemplo, habrán métodos cuya función sea la de realizar una estimación inicial de los parámetros del controlador y otros que se encarguen de optimizar el funcionamiento del lazo de control, partiendo de la sintonía ya existente en el PID. Además, será recomendable que los métodos seleccionados sean capaces de trabajar con procesos de dinámica difícil, puesto que en el CERN existen procesos de estas características y serán los que más necesiten del uso de la herramienta. Dicha aplicación deberá estar embebida en el entorno de trabajo de la institución (UNICOS) para poder estar disponible en el ajuste de todos los lazos de control que lo requieran.

Palabras clave:

- Control automático
- Autómatas programables
- Sistemas SCADA
- PID
- Métodos de sintonía de PIDs

ÍNDICE GENERAL

ÍNDICE DE FIGURAS

CAPITULO I:

INTRODUCCIÓN

1. Motivación y objetivos

Un controlador se puede entender como un operador que, en función de la salida deseada de la planta y de la salida real medida, proporciona la acción de control que se debe aplicar sobre el sistema. Aunque existen numerosos tipos de controlador, el que mayor aceptación tiene en el entorno industrial es el regulador PID (Proporcional, Integral, Derivativo). Según una estimación dada por Åström, el 95% de los bucles de control industriales son de tipo PID, y fundamentalmente PI.

Si bien el controlador PID tiene una alta implantación en la industria, en la mayoría de los casos su funcionamiento no está optimizado. De este modo, lazos de control que podrían tener un funcionamiento excelente, se encuentran trabajando de forma mediocre o insatisfactoria.

El mal funcionamiento es debido, en muchos casos, a que los parámetros del PID han sido sintonizados manualmente sin la realización de un estudio del proceso que se pretende controlar. Cuando la sintonía se realiza de este modo, las posibilidades de acierto en la selección de los parámetros dependen únicamente de la experiencia del operador.

El CERN comparte, en este aspecto, esta misma problemática que se encuentra en cualquier entorno industrial. Sólo en la sección que abarca la criogenia del LHC contiene unas 50.000 E/S, 11.000 actuadores y unos 5.000 lazos de control, y más en general, repartidos entre los diversos experimentos que están desarrollando hay implementados unos 8.000 controladores.

Además de los problemas habituales en la sintonía de PIDs, en el caso del CERN hay que añadir aquellos derivados de la complejidad en la dinámica de la respuesta de los procesos con los que trabajan, como es el caso de los lazos de temperatura que conforman la sección de criogenia del LHC mencionada anteriormente.

Por estas razones, se hace necesario el desarrollo de una aplicación de autosintonía que optimice el funcionamiento de cada controlador en función del proceso que trate y que se pueda ejecutar de manera simultánea para varios reguladores.

La herramienta deberá ir embebida en el *framework* UNICOS, creado por el CERN para poder desarrollar con facilidad y homogeneidad aplicaciones industriales de control de procesos. Lejos de ser una complicación añadida, esto supone una gran ayuda, puesto que la estructura para todos los procesos

está unificada. Además, cuando la aplicación esté desarrollada, podrá ser implantada simultáneamente para estar disponible en todos ellos.

Durante el desarrollo del proyecto se entrará a comprender más en profundidad la estructura de UNICOS y el modo en que se introducirá la herramienta dentro de su jerarquía, así como los métodos de autosintonía más adecuados para implementar.

2. CERN

El CERN es la Organización Europea para la Investigación Nuclear, sus siglas se corresponden con su nombre en francés "Conseil Européen pour la Recherche Nucléaire". Se trata del mayor laboratorio de investigación en física de partículas del mundo, situado en la frontera entre Francia y Suiza. En sus instalaciones cuenta con un complejo de aceleradores de partículas entre los que destaca el LHC (Large Hadron Collider), un acelerador y colisionador de partículas.

Algunos de los aceleradores que forman parte de las instalaciones del CERN son el ya desmantelado LEP (Large Electron-Positron Collider), el PS ((Proton Synchrontron)), el SPS (Super Proton Synchrontron), los aceleradores lineales LINAC1, 2, etc. Un esquema del complejo de aceleradores puede verse en la *Figura 1 – Estructura de aceleradores del CERN*.

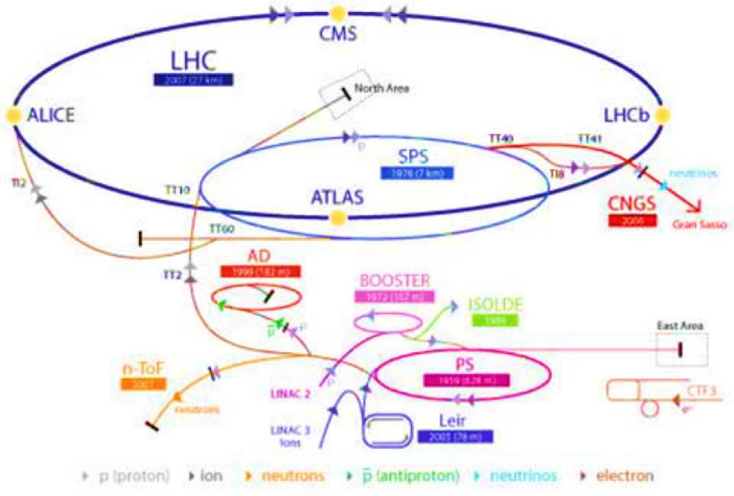


Figura 1 – Estructura de aceleradores del CERN

3. LHC: Large Hadron Collider

Los aceleradores de partículas fueron inventados para proporcionar energía a las partículas para investigar la estructura de los núcleos atómicos. Desde entonces, han sido utilizados para investigar muchos aspectos de la física de partículas. Su tarea es acelerar y proporcionar energía a un haz de partículas generando campos eléctricos que aceleran las partículas, y campos magnéticos que las conducen.

El LHC (Large Hadron Collider) es un acelerador de partículas utilizado por físicos para estudiar las partículas más pequeñas conocidas, es decir, las partículas elementales de la materia.

Dos haces de partículas subatómicas llamadas 'hadrones' (pueden ser protones o iones) viajan en direcciones opuestas dentro de un acelerador circular, ganando energía con cada vuelta. Los físicos utilizan el LHC para recrear las condiciones posteriores al Big Bang, haciendo colisionar los dos haces a una energía muy alta. Equipos de físicos de todo el mundo analizan las partículas creadas en las colisiones utilizando unos detectores especiales en varios experimentos dedicados al LHC (ATLAS, CMS, ALICE y LHCb).

El LHC es el acelerador de partículas más grande y energético del mundo. Usa el túnel de 27 km de circunferencia creado para el Gran Colisionador de Electrones y Positrones (LEP).

3.1. Cómo funciona el LHC

El LHC consiste en un anillo de 27 km de circunferencia de imanes superconductores con gran número de estructuras aceleradoras para incrementar la energía de las partículas a lo largo del camino.

Dentro del acelerador, dos haces de partículas viajan a una velocidad cercana a la velocidad de la luz con altas energías antes de chocar entre ellos. Los haces viajan en direcciones opuestas a través de distintas tuberías, que mantienen el vacío. Estos son guiados a lo largo del anillo del acelerador por un campo magnético muy fuerte, alcanzado utilizando electroimanes superconductores. Dichos electroimanes están construidos a partir de bobinas de un cable eléctrico especial que funciona en estado superconductor, conduciendo energía eléctrica de forma eficiente sin resistencia o perdida de energía. Esto requiere el enfriamiento de los imanes a temperaturas de aproximadamente -271ºC, el encargado de dicho enfriamiento es el sistema de criogenia. Debido a esto, el acelerador está conectado a un sistema de distribución de helio líquido, que se encarga del enfriamiento de los imanes.

CAPITULO II:

UNICOS

1. Introducción

UNICOS (**UN**ified **I**ndustrial **CO**ntrol **S**ystem) es un *framework* desarrollado por el CERN (**C**onseil **E**uropéen pour la **R**echerche **N**ucléaire) para el desarrollo de aplicaciones de control industriales. Este sistema comprende las dos capas superiores de un sistema de control clásico: supervisión y control. La propuesta de UNICOS es un método para diseñar y desarrollar aplicaciones de control que funcionarán en sistemas comerciales de PLCs y SCADAs.

El desarrollo de UNICOS comenzó en 1988 como un elemento necesario para el sistema de criogenia del LHC (**L**arge **H**adron **C**ollider), pero de hecho ha terminado convirtiéndose en el estándar de desarrollo de sistemas de control industrial para el CERN.

Como se muestra en la Figura 2, el framework incluye las capas de supervisión, control y campo utilizando elementos comerciales, así como la creación de las comunicaciones necesarias entre ellas.

Figura 2 – Estructura de capas de UNICOS

El objetivo de UNICOS es estandarizar el desarrollo de aplicaciones de control en el CERN para:

- Hacer hincapié en las buenas prácticas de diseño y operación de las aplicaciones de control de procesos continuos
- Reducir el coste de la automatización de procesos continuos (por ejemplo, refrigeración, climatización …)
- Optimizar los esfuerzos de ingeniería del ciclo de vida (por ejemplo, el uso de herramientas de generación automática de código)

La aplicación del sistema UNICOS aporta una gran cantidad de ventajas, tanto a nivel de desarrollo como de utilización.

Para los operadores UNICOS proporciona:

- Modos de operación (manual, automático, forzado…).
- Simulación de sensores.
- Manipulación de actuadores.
- Diferentes tipos de accesos al sistema de control (experto, operador, monitorización).

Para los desarrolladores proporciona:

- Independencia entre las diferentes capas del sistema de control, lo que permite desarrollar las capas por separado.
- Una metodología de trabajo con dos partes fundamentales:
  - Documentación y especificaciones necesarias para realizar el análisis funcional del proceso a controlar.
  - Documentación para el diseño y desarrollo del código de control.

En resumen, el sistema UNICOS ofrece herramientas de generación automática de código de control basado en objetos con las siguientes funcionalidades:

- Generación de objetos PLC y SCADA.
- Configuración automática de la comunicación entre el SCADA y el PLC.

La metodología de diseño y desarrollo de sistemas de control que propone UNICOS es una modelización del proceso basada en una jerarquía de objetos. Estos objetos son usados como un lenguaje común entre los ingenieros de proceso y programadores para definir el análisis funcional del proceso.

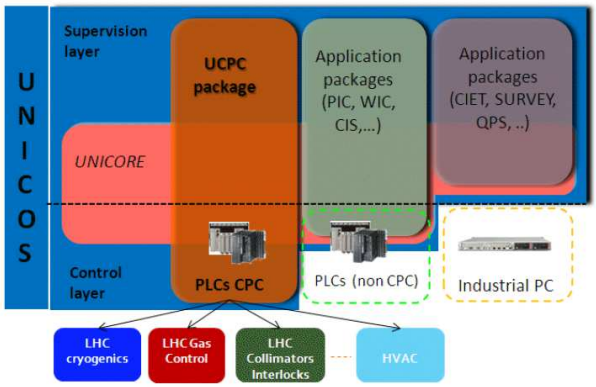


Figura 3 – Arquitectura de UNICOS

Además, las herramientas de generación proporcionadas automatizan la instanciación de los objetos en las capas de control y de supervisión, y también generan el esqueleto del programa de control para los PLCs.

En la Figura 3 se puede observar un gráfico donde queda definida toda la arquitectura de UNICOS.

Como se puede ver, es una herramienta de generación muy útil para procesos repetitivos favoreciendo considerablemente la reducción del tiempo de desarrollo y de mantenimiento del sistema de control

2. UNICOS Continuous Process Control

El UCPC (UNICOS Continuous Process Control) es un *framework* para crear aplicaciones de control de procesos. Como se puede ver en la Figura 3, es uno de los componentes de UNICOS.

En el CERN, algunos ejemplos de procesos, donde se ha aplicado UCPC, son:

- Sistemas criogénicos que proporcionan helio frío para cargas térmicas como imanes superconductores
- Sistemas de gas que proporcionan mezclas de gases para detectores de partículas
- Los sistemas de vacío, donde las bombas equipo especial recintos
- Sistemas que proporcionan potencia de refrigeración de equipos y detectores de refrigeración (utilizando agua, CFC, PFC, etc.)
- Sistemas HVAC proporcionan la ventilación en edificios experimentales y zonas subterráneas

UCPC proporciona una metodología, una librería de objetos y una serie de herramientas para el diseño y la implementación de aplicaciones de control industrial, tanto en las capas control como de supervisión. Incluyen en la capa de control bibliotecas para PLCs Siemens y Schneider y en la capa de supervisión para WinCC OA (antes PVSS) y WINCC Flexible.

2.1. Metodología UNICOS

La metodología de UNICOS proporciona los pasos a seguir para el desarrollo de aplicaciones CPC. Contiene los siguientes pasos:

1. Análisis funcional: el ingeniero de proceso transfiere su conocimiento sobre el proceso en el documento llamado UNICOS Functional Analysis.
2. Diseño de lógica UNICOS: el ingeniero de proceso y el ingeniero de control trabajan juntos para definir el comportamiento del proceso en lenguaje UNICOS.
3. Configuración de las especificaciones: todos los objetos UNICOS son definidos y parametrizados utilizando un archivo Excel/XML.
4. Generación automática: el código de instancias y de lógica para el PLC y el archivo de configuración para WinCC OA se generan automáticamente utilizando las herramientas de generación.
5. Configuración de la lógica: completar la generación automática de lógica si fuese necesario, siguiendo la lógica de diseño UNICOS.
6. Compilación del código de control.
7. Test de comprobación.
8. Puesta en marcha.
9. Operación.

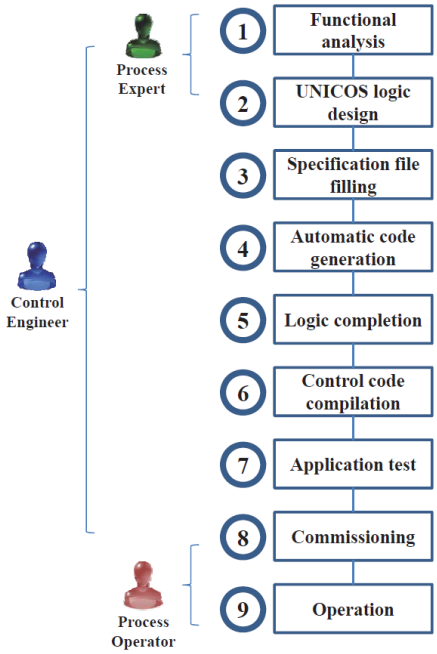


Figura 4 – Metodología de UNICOS

2.2. Estructura de los objetos UNICOS

La jerarquía de UNICOS está estructurada en base a una serie de objetos elementales, en los cuales quedan identificados cada uno de los elementos que componen el proceso a controlar. Estos objetos incluyen la forma de interrelacionarse entre sí y, además, los datos relativos a la parte del proceso que controlan.

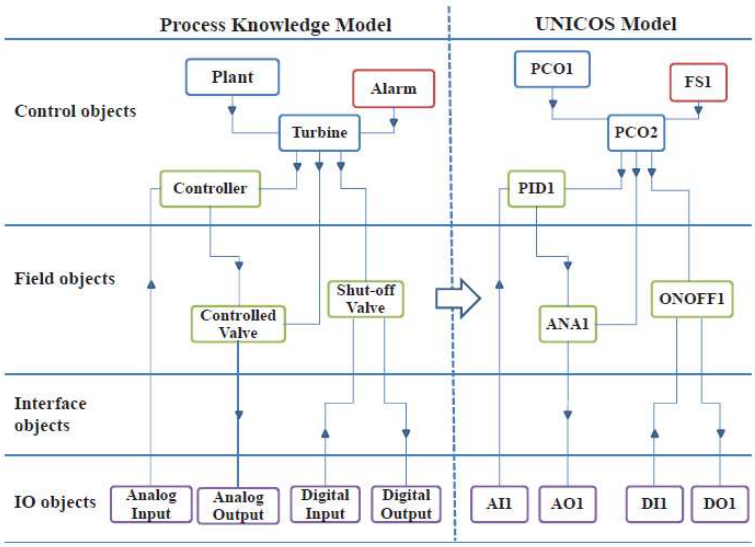


Figura 5 – Ejemplo de la jerarquía de UNICOS

El paquete UNICOS-CPC contiene un conjunto de objetos clasificados por su funcionalidad. Hay cuatro categorías principales de objetos: objetos de E/S, de interfaz, de campo y de control:

- **Objetos de E/S:** Entrada/Salida de información. Estos pueden referirse a los PLC de la periferia, los buses de campo o simplemente la memoria interna.
  - *Digital Input*
  - *Analog Input*: el formato de entrada es un entero en el PLC
  - *Analog Input Real*: la entrada es un dato real en el PLC
  - *Digital Output*
  - *Analog Output*: la salida es un número entero en el PLC
  - *Analog Output Real*: la salida es un dato real en el PLC

- **Objetos interfaz:** parametrización y el estado (también pueden incluir información de la periferia). Estos objetos son más ligeros que los objetos de E/S.
  - *Analog Parameter:* objeto para enviar un valor analógico del SCADA al PLC
  - *Digital Parameter:* objeto de enviar un valor digital del SCADA al PLC
  - *Word Parameter:* objeto de enviar un valor de palabra del SCADA al PLC
  - *Analog Status:* objeto para enviar un valor analógico del PLC al SCADA
  - *Word Status:* objeto para enviar un valor de palabra del PLC al SCADA

- **Objetos de campo:** Modelo de los equipos físicos de campo (por ejemplo: válvulas, motores,…). Los objetos de campo están siempre conectados a los objetos de E/S.
  - *Local:* representa solamente equipos accionados en modo local (válvulas manuales).
  - *OnOff:* representa equipos gobernados por una señal digital (válvulas digitales, motores).
  - *Analog:* representa equipos gobernados por una señal analógica (válvulas de control).
  - *Anadig:* representa equipos con una entrada analógica y una salida digital (calentador impulsado por PWM: modulación por ancho de pulso).
  - *AnaDO:* representa equipos gobernados por una señal digital y por una señal analógica para el posicionamiento (calentador eléctrico activado por una señal de booleana y posicionado por una referencia analógica, variador de frecuencia, …)
  - *MFC:* representa equipos específicos gobernados por una señal analógica (por ejemplo: Flujo de Set Point) con una señal de salida digital y una medición de flujo. (Controlador de flujo másico)

- **Objetos de control:** las acciones de control lógico proceso de realización, alarmas y enclavamientos y control de retroalimentación. Ellos siempre están actuando en Objetos de campo
  - *PCO:* Objetos de control de procesos dan órdenes a sus campos afectados o a objetos dependientes PCO.
  - *Controller:* Los objetos que contiene un mecanismo de regulación de la retroalimentación. El método por defecto es el algoritmo.

- *Digital alarm:* Objetos que disparan una señal de alarma de un valor booleano
- *Analog alarm:* Objetos que disparan una señal de alarma con respecto a un valor analógico (incluye alarma (HH / LL) y advertencia (H / L) umbrales)

Todos los objetos que han sido desarrollados para PLCs Schneider y Siemens están descritos aquí. En los PLCs de Schneider están representados por DFBs gen éricos con variables asociadas. En S7 cada objeto está representado por FB gen éricos que serán inicializados en un DB.

En esta lista se pueden identificar todos los objetos que forman UNICOS-CPC6. Para diseñar una aplicación basada en un proceso determinado se deben identificar cada uno de los elementos y decidir que bloque lo identifica, no solo los elementos físicos, sino también las partes del proceso que no están asociadas a ningún elemento físico, como podría ser una alarma.

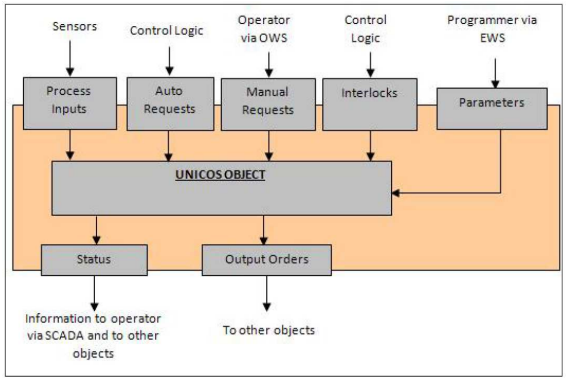


Figura 6 – Estructura "tipo" de un objeto UNICOS

De la lista de objetos descrita, sólo se detallarán a continuación el PCO (objeto de control de proceso) y el Controller (controlador), por ser los más relevantes para el tema tratar en el proyecto.

2.2.1. Objeto PCO

Los objetos PCOs son los más relevantes en el proceso, debido a que según la jerarquía UNICOS son los que encabezarán y administrarán todo el proceso. La estructura general de estos bloques puede verse en la Figura 7.

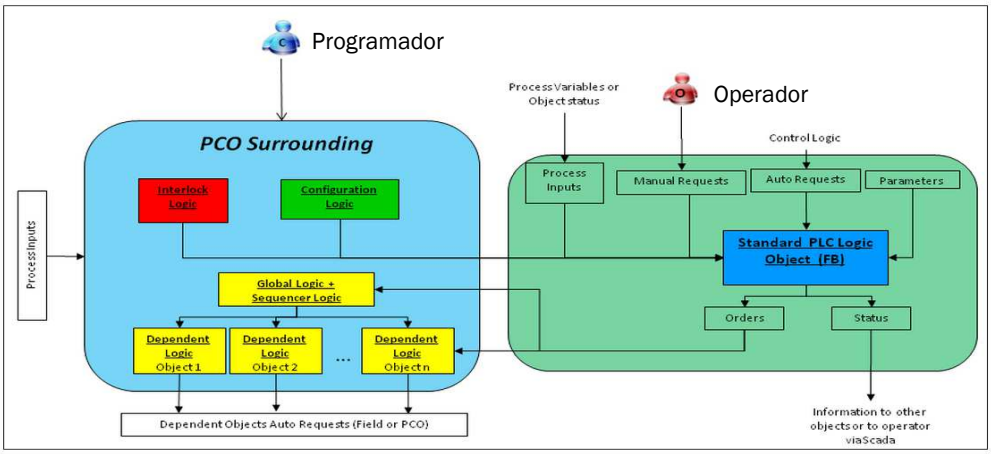


Figura 7 – Estructura del objeto PCO

La lógica de los PCOs se divide en varias sub-funciones: lógica de *interlocks*, lógica de configuración, lógica estándar del objeto y lógica específica del proceso. A continuación se detallan sus características:

- Lógica de *interlocks*:
  - Definición de *interlock*: condición asíncrona que permite a un actuador o a una unidad posicionarse en su posición segura o prevenir su activación por motivos de seguridad. Un *interlock* no debe ser utilizado para funcionamiento normal, solo comportamientos anormales. Los *interlocks* se implementan en el PLC y no garantizan la seguridad de las máquinas o de las personas. Todos los *interlocks* están disponibles en el PLC y en SCADA.
  - *Interlock* de arranque (evita el arranque del objeto);
  - *Interlock* de parada total (para el objeto), el re-arranque requiere la eliminación del interlock y su reconocimiento por parte del operador;
  - *Interlock* de parada temporal (para el objeto), el re-arranque no requiere la intervención del operador, a no ser que se encuentre en modo manual o forzado.

- Lógica de configuración: esta lógica determina si el objeto estará activado o desactivado dependiendo de los estados de los objetos dependientes.

- Lógica estándar del objeto: existen ocho modos posibles de funcionamiento que pueden ser activados por la lógica o por el operador. Si se para el proceso, se puede ir de un modo a otro sin restricciones. Por el contrario, si el proceso está activo, el paso de un modo a otro debe ser autorizado.

- Lógica específica del objeto: el punto de funcionamiento del proceso depende de la lógica de *interlock*s, la de configuración y la estándar. Un comando es enviado a la lógica específica del proceso.

2.2.2. Objeto Controller

Los objetos Controlador son objetos que encierran un algoritmo de regulación, aunque el más utilizado es el regulador PID. Existe la posibilidad de modificar el algoritmo de regulación mediante las características presentadas aquí. Los objetos Controlador deben estar asociados a un objeto analógico (objeto de campo).

Este objeto permite:

- Administrar los procedimientos
- Controlar la salida:
  - Por medio de la regulación y sus instrucciones asociadas
  - Por solicitud de la lógica o el operador, independientemente del regulador
- Tener en cuenta los límites de velocidad con los que las instrucciones pueden cambiarse
- Tener en cuenta los límites que se ocupan de las salidas y las instrucciones
- Definir los parámetros del regulador
- Actualizar los estados incluyendo:
  - Las instrucciones y sus indicadores asociados
  - Los límites de velocidad de cambio en las instrucciones y sus indicadores asociados
  - Las señales de salida y sus indicadores asociados
  - Los límites de las señales de salida y sus indicadores asociados
  - Los parámetros del regulador y sus indicadores asociados
  - Procedimientos
  - Avisos de error y simulación
  - El valor medido a la entrada
  - La posición de salida si existe

Los controladores utilizados por defecto en UNICOS son los PID, proporcional integral derivativo con filtro en el término derivativo.

$$PID_{OUT} = K_p\left(1 + \frac{1}{T_i s} + \frac{sT_d}{1 + \frac{sT_d}{T_{ds}}}\right)E(s) \qquad (1)$$

Los controladores PID están organizados según la norma IEC 61131-3, cuya estructura puede verse en la Figura 8.

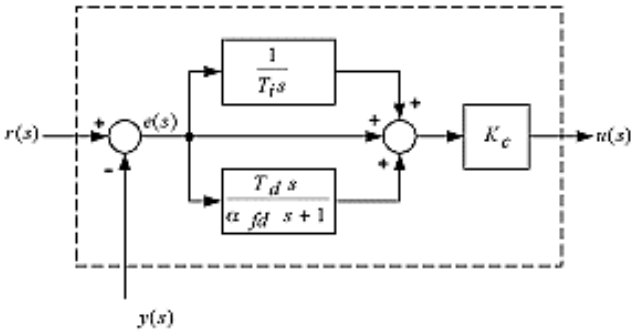


Figura 8 – Estructura del PID utilizado

El funcionamiento del bloque Controlador dependerá del modo de operación activo:

- Cuando se activa una regulación, el *set-point* varía desde el valor actual del proceso hasta el *set-point* deseado con una rampa para evitar discontinuidades en el *set-point*.
- Si se para la regulación (PID en modo manual), la salida del PID se sitúa en una posición predefinida o permanece en su último valor y el *set-point* es igual al valor del proceso.
- Cuando el actuador del controlador está en estado manual o forzado, la salida del PID está rastreando la posición del actuador y el *set-point* es igual al valor del proceso. Esto permite evitar saltos en el orden del actuador controlado.

Los diferentes modos de operación se explican más en profundidad a continuación.

2.3. Modos de operación

El modo de operación del objeto identifica si bien el objeto es manejado por el operador, por la lógica de control o por el proceso. Solo un modo puede estar activo al mismo tiempo. La activación de uno modo desactiva automáticamente los demás.

Los modos de operación soportados por UNICOS-CPC son los siguientes:

- ***Modo Automático (Auto Mode):*** el objeto es manejado por la lógica de control por un objeto más alto en la jerarquía.
  - Puede activarse mediante solicitud automática (*Auto Auto Mode Request*).
  - Puede activarse mediante solicitud manual (*Manual Auto Mode Request*).

- ***Modo Manual (Manual Mode):*** el operador maneja el objeto a través de la capa de supervisión.
  - Puede activarse solo mediante una solicitud del operador (*Manual Manual Mode Request*).
  - Si una solicitud especial automática Auto Inhibición de Modo Manual (*Auto Inhibit Manual Mode*) está activa mientras el objeto está en modo automático, el modo manual no puede ser activado mediante solicitud manual.
  - El retorno al modo automático es posible mediante la lógica de control (*Auto Auto Mode Request*).
  - Se aplican los *interlocks*.

- ***Modo forzado (Forced Mode):*** el operador maneja el objeto a través de la capa de supervisión.
  - Puede activarse solo mediante una solicitud del operador (*Manual Forced Mode Request*).
  - Si una solicitud especial automática Auto Inhibición de Modo Forzado (*Auto Inhibit Forced Mode*) está activa mientras el objeto está en modo automático, el modo forzado no puede ser activado mediante solicitud manual
  - El retorno automático al modo automático es imposible mediante la lógica de control.
  - Se aplican los *interlocks*.

- *Modo de manejo Local (Local Drive Mode):* el objeto es manejado localmente en la capa del proceso. Este modo es utilizado en objetos de campo y de E/S cuando el objeto es manejado por un comando proveniente del campo del proceso.
    - El parámetro de hardware local (*Parameter Hardware Local Drive*) tiene que ser activado para permitir el modo local.
    - Este modo es activado por una entrada digital conectada a un conmutador instalado en el campo de proceso: *Hardware Feedback Local Drive*.
    - Este modo está pensado para propósitos de mantenimiento. No afecta a la jerarquía de los modos normales de funcionamiento (Automático, Manual y Forzado) pero los anula.

3. UNICOS Application Builder

La base de la filosofía UNICOS de unificación y simplificación en el diseño y desarrollo de aplicaciones CPC se encuentra en la herramienta *UNICOS Application Builder* (UAB). La herramienta UAB es un generador que, a través de una hoja de especificaciones introducida por el usuario, genera la estructura para el funcionamiento de un proceso, así como las instancias y la lógica que lo definen.

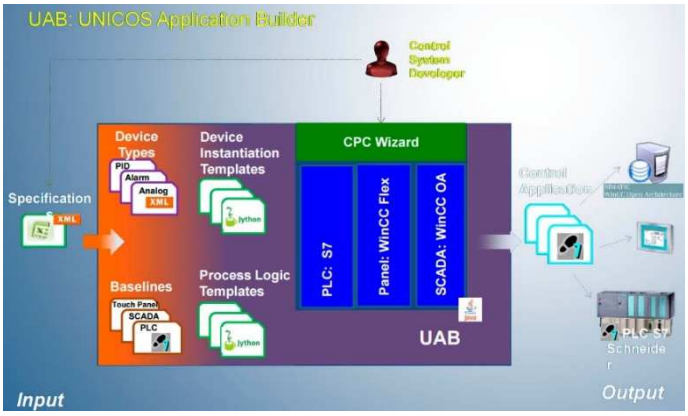


Figura 9 – Estructura del UAB

La hoja de especificaciones consta de una pestaña dedicada a cada tipo de objeto de la lista anterior. De este modo, una vez identificada la estructura de la planta en objetos elementales de UNICOS, sólo resta parametrizar cada una de dichas variables siguiendo las instrucciones particulares de cada elemento.

Una vez completado este paso, sólo resta ejecutar el asistente UAB. El generador UAB emplea los siguientes *plugins* para la creación de todos los archivos necesarios para el correcto funcionamiento tanto de la capa de supervisión como de la capa de control de un sistema continuo.

- ***Generador de instancias S7*** *(S7 Instance Code Generator Plugin)**:*** encargado de generar código de las instancias para el PLC Siemens.

- ***Generador de lógica S7*** *(S7 Logic Generator Plugin)**:*** encargado de generar el código de lógica para el PLC Siemens, es decir, todas las interconexiones entre los PCOs y los objetos de campo del proyecto a controlar

- ***Generador de instancias PVSS*** *(PVSS Instances Generator)**:*** encargado de crear el fichero de importación para el SCADA.

4. Nivel de implantación del algoritmo

La funcionalidad que se pretende desarrollar en el presente proyecto es una aplicación que contenga diversos métodos de autosintonía para controladores PID, por lo que el primer punto a tratar debe ser la localización de dicha aplicación dentro de la estructura UNICOS.

Analizando detenidamente los tipos de objetos elementales, resulta evidente que la herramienta de autosintonía deberá estar relacionada directamente con el objeto *Controller*, por ser éste el encargado de gobernar de manera directa los parámetros PID.

Por otra parte, como se ha mencionado en ocasiones anteriores, UNICOS presenta una estructura que se divide en los niveles de automatización clásicos y establece el sistema de comunicaciones entre ellos. Esto quiere decir, que existe la posibilidad de acceder al controlador tanto desde la capa de control, o sea desde el autómata, como desde la capa de supervisión, es decir desde el SCADA, de manera que será necesario valorar las posibilidades de cada uno de ellos.

4.1. Capa de control: PLC

El control se lleva a cabo mediante el uso de Controladores Lógicos Programables (PLC). Las CPUs de los PLCs utilizadas para el control deben ser compatibles con el *framework* UNICOS:

- Schneider Premium y M340 CPU compatibles con Unity V4 o superiores
- Siemens S7 300 y S7 400 compatibles con Step 7 Pro (ETM200 también puede ser utilizada para aplicaciones muy pequeñas)

4.1.1. Lenguajes de programación

Los lenguajes de programación utilizados para el programa de control son los específicos de los PLC Siemens y Schneider. El código generado sigue la norma IEC 61499, se trata de un código orientado a objetos, debido a que se crean instancias de *Function Blocks* que definen a cada objeto UNICOS.

Para la creación de un programa lógico que determine el funcionamiento del PLC, SIMATIC STEP7 permite trabajar en diferentes lenguajes de programación.

4.1.1.1. KOP y FUP – Lenguajes de programación gráficos

FUP es un lenguaje gráfico de Step7 que utiliza los cuadros del álgebra booleana para representar la lógica. Asimismo, permite representar funciones complejas (p.ej. funciones matemáticas) mediante cuadros lógicos.

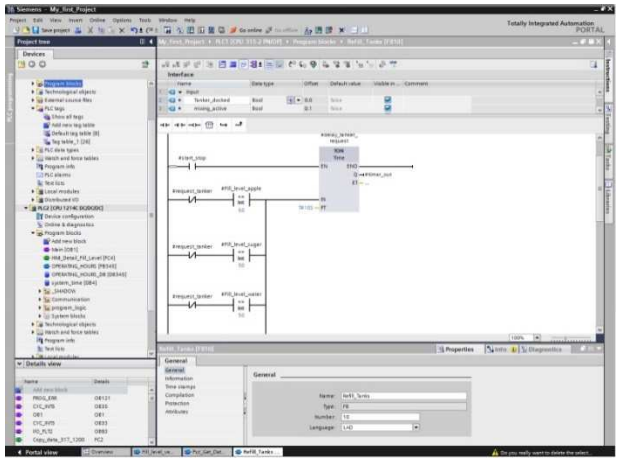


Figura 10 – Ejemplo de programación gráfica

KOP es un lenguaje de esquema de contactos, escalera o ladder. Es un lenguaje gráfico de Step 7 y probablemente el más extendido en todos los lenguajes de programación y por tanto el más similar a otros.

Tiene la ventaja de ver agrupados por bloques las diferentes lógicas y tener bloques complejos. Cuando hay mucha lógica booleana en serie suele ser más compacto y más fácil de ver el segmento completo.

4.1.1.2. GRAPH (SFC) – Programación secuencial

La programación *Sequential Function Chart* (SFC) se usa para describir los procesos secuenciales con secuencias de pasos alternativas o paralelas. Los procesos son proyectados y programados de forma clara y rápida siguiendo un modo de representación estandarizado (según IEC 61131-3, DIN EN 61131). El proceso se describe gráficamente y se divide para ello en los distintos pasos con una representación clara del alcance de las funciones.

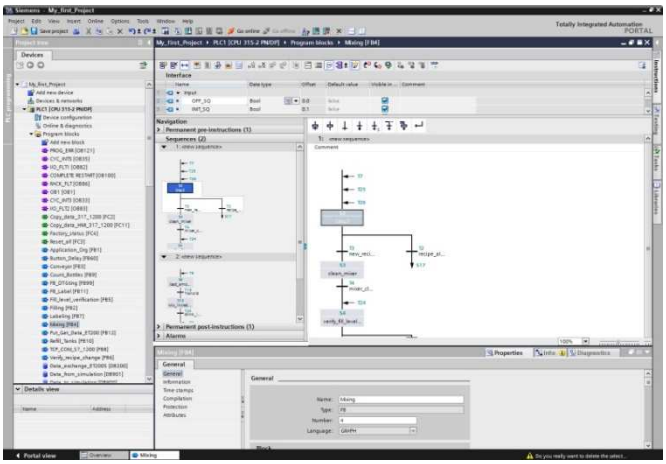


Figura 11 – Ejemplo de programación GRAPH

4.1.1.3. SCL – Programación de algoritmos complejos

El lenguaje de control estructurado (*Structured Control Language*) corresponde con el lenguaje de programación textual ST (*Structured Text*) definido en la norma IEC 61131-3 y cumple el nivel básico (*Base Level*) y el nivel de reutilizado (*Reusability Level*) según PLCopen.

SCL es ideal sobre todo para una programación rápida de algoritmos complejos y funciones matemáticas o para las tareas planteadas del área del procesamiento de datos. El código SCL al ser más corto y claro es mucho fácil de manejar y realizar.

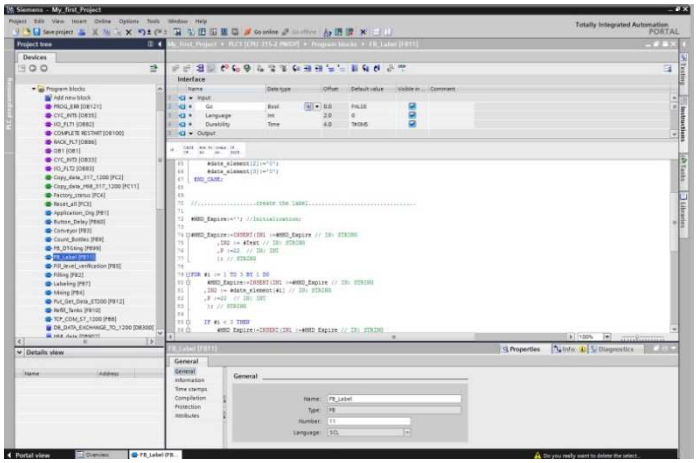


Figura 12 – Ejemplo de programación SCL

4.1.1.4. AWL – Lista de instrucciones

AWL es un lenguaje de programación textual orientado a la máquina. En un programa creado en AWL, las instrucciones equivalen en gran medida a los pasos con los que la CPU ejecuta el programa. Para facilitar la programación, AWL se ha ampliado con estructuras de lenguajes de alto nivel (tales como accesos estructurados a datos y parámetros de bloques).

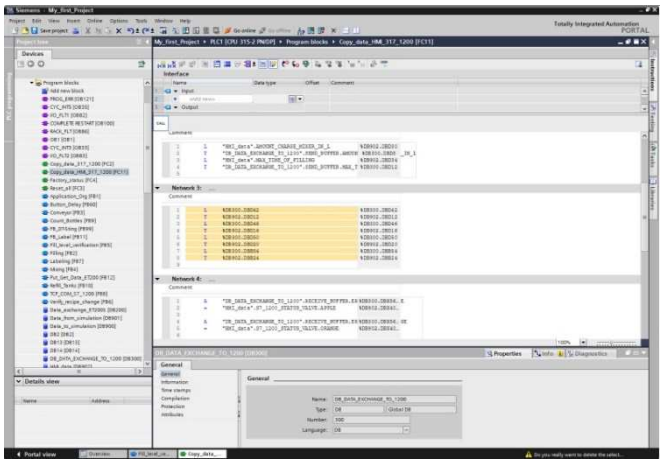


Figura 13 – Ejemplo de programación AWL

Es el más completo y el más complejo visualmente de seguir. Para instrucciones sencillas es muy útil pero cuando se quiere hacer una lógica un poco compleja el trabajo de seguimiento y de depuración es complicado y fácilmente susceptible de cometer errores.

4.2. Capa de supervisión: SCADA

La capa de supervisión está diseñada con el SCADA WINCC OA, aunque también es posible incluir la supervisión local mediante la utilización de un panel táctil.

Dentro de este entorno, la mejor opción es diseñar algún tipo de librería que contenga los algoritmos de autosintonía que se quieran implantar. De este modo, cualquiera de los lazos de regulación activos en el proceso podrá realizar una llamada a la herramienta de autosintonía y ser ejecutado por varios lazos en paralelo.

Además de la programación mediante los parámetros definidos para cada objeto, WinCC Open Architecture también está equipado con una interfaz de programación en un lenguaje de alto nivel denominado *Control*.

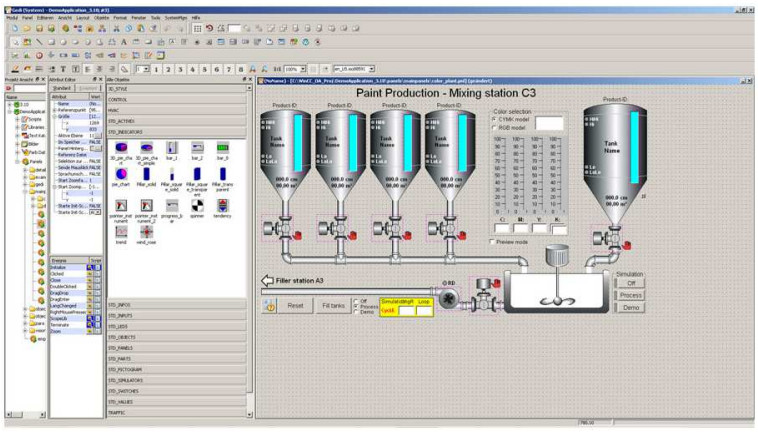


Figura 14 – Diseño de SCADA con WinCC OA

4.2.1. Control Script

Control es un lenguaje de alto nivel con una sintaxis sencilla y fácil de entender basada en el lenguaje de programación ANSI-C. Un programa Control ("s*cript*") se utiliza para especificar la respuesta de WinCC OA a un evento.

Cuenta con numerosas funciones que facilitan el trabajar con los elementos propios del entorno WINCC OA (*Data Points*), aunque a nivel de tratamiento matemático de información el número de herramientas a las que se puede acceder es bastante limitado.

Los *Control Scripts* son programas escritos en el lenguaje interno "C*ontrol*". Los procedimientos son interpretados por el sistema y, por tanto, no es necesaria la compilación, lo que significa que los cambios pueden ser probados inmediatamente.

Estos scripts pueden usarse para vincular los cambios en los estados de proceso con respuestas definidas, por ejemplo, un cambio en la configuración del dispositivo o los cambios en la pantalla de visualización.

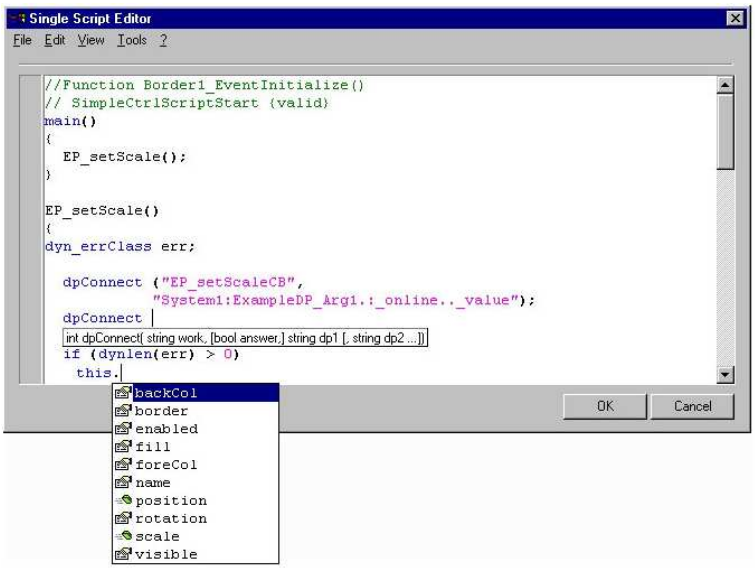


Figura 15 – Ejemplo de Control Script

También es posible crear librerías independientes a las que serán accesibles para todos los *Control Script* del proyecto. Las bibliotecas pueden ser modificadas por el usuario desde cualquier editor de texto o abriéndolo desde el propio editor de scripts del módulo Gedi.

4.2.2. DLL (Dynamic Link Library)

Un archivo DLL es una biblioteca que contiene código y datos que pueden utilizarse por más de un programa al mismo tiempo. Son archivos de código ejecutable que ejecuta el sistema operativo, que necesariamente será Windows, por petición de un programa.

Mediante el uso de un archivo DLL, un programa puede ser modularizado en componentes separados. Cada módulo puede cargarse en el programa principal en tiempo de ejecución, si está instalado el módulo. Dado que los módulos son independientes, el tiempo de carga del programa es más rápido y un módulo sólo se carga cuando se solicita esa funcionalidad.

```
// SampleDLL.cpp
//

#include "stdafx.h"
#define EXPORTING_DLL
#include "sampleDLL.h"

BOOL APIENTRY DllMain( HANDLE hModule,
                       DWORD  ul_reason_for_call,
                       LPVOID lpReserved
                                        )
{
    return TRUE;
}

void HelloWorld()
{
        MessageBox( NULL, TEXT("Hello World"), TEXT("In a DLL"), MB_OK);
}
```

```
// File: SampleDLL.h
//
#ifndef INDLL_H
#define INDLL_H

#ifdef EXPORTING_DLL
extern __declspec(dllexport) void HelloWorld() ;
#else
extern __declspec(dllimport) void HelloWorld() ;
#endif

#endif
```

Figura 16 – Ejemplo de código DLL

Esto, unido al hecho de que su uso se encuentra muy extendido hace que su utilización resulte ventajosa, puesto que ya existe una gran variedad de librerías desarrolladas para este entorno.

4.3. Conclusión

Entre las posibilidades de programación que ofrece Step 7 la que más puede adecuarse a las necesidades de la herramienta que se quiere desarrollar es el lenguaje SCL. Por tratarse de un lenguaje de programación de alto nivel permite una mayor facilidad a la hora de realizar cálculos matemáticos que, indudablemente, habrá que introducir.

Pero, aunque se disponga de un lenguaje que pueda dar soporte a la aplicación, es importante considerar que el PLC no es un dispositivo diseñado para el almacenamiento grandes cantidades de datos ni para el procesamiento matemático. Por esta razón, es probable que a nivel de memoria y tratamiento de datos fuera insuficiente.

Además, el hecho de que los PLCs se encuentren ubicados en la capa de control hace posible que, si la herramienta de autosintonía se implantara en este nivel, pudiera evitar los *interlocks* generados por la capa de supervisión y creando, por tanto, posibles problemas de seguridad en la planta.

Por todas las razones mencionadas se descarta el PLC como soporte para la implementación de la aplicación.

Si subimos al nivel de supervisión automáticamente desaparece el problema de la escasez de recursos de memoria, puesto que cualquiera de las posibilidades planteadas tendrá como soporte un ordenador.

Las DLLs, como se mencionó anteriormente, tienen la gran ventaja de contar con una gran base de librerías de funciones ya creadas, lo cual puede ser muy beneficioso a la hora de hacer el tratamiento matemático de los datos, porque serán necesarias inversiones de matrices, filtros discretos, etc,...

Además, mientras que los *scripts* son interpretados por el sistema, las DLLs requieren compilación, de modo que el proceso de comprobación y validación de cada una de ellas es mucho más lento para estas últimas.

El gran inconveniente de las DLLs se encuentra en el hecho de requerir un sistema operativo Windows para ejecutarse, siendo incompatible, por ejemplo, con Linux. Debido a que en el CERN es bastante utilizada una variante de la versión *Scientific Linux* denominada SLC, la opción de las DLLs debe ser descartada.

Utilizando los *Scripts* salvamos este problema y, además de asegurar el funcionamiento en los equipos del CERN, se deja la puerta abierta a una posible comercialización para un público general de la aplicación, asegurando la máxima compatibilidad con cualquier equipo.

CAPITULO III:

AUTOSINTONÍA DE CONTROLADORES PID

1. Introducción: Controlador PID

El controlador PID es un mecanismo de control basado en la realimentación ampliamente utilizado en sistemas de control industrial.

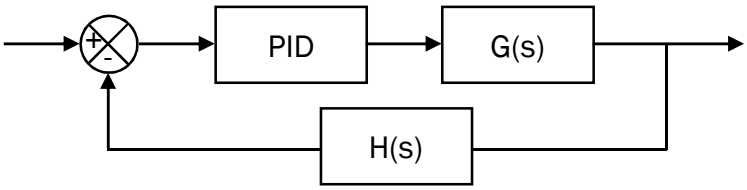


Figura 17 – Lazo cerrado de control

El algoritmo de control consta de tres términos, denominados proporcional, integral y derivativo, relacionados con la diferencia entre un valor de referencia fijado y la salida real del sistema según la siguiente expresión.

$$\mathrm{u(t)} = \mathrm{K_p} e(t) + \int_0^t \frac{e(\tau)}{T_i} d\tau + T_d \frac{e(t)}{dt} \qquad (2)$$

$$(t) = r(t) - y(t) \qquad (3)$$

De la estructura completa del regulador PID se pueden obtener variaciones consistentes en la no inclusión de alguno de sus términos, siendo perfectamente válidas configuraciones del tipo P, PI, PD,... dependiendo de las necesidades concretas del proceso.

Por otra parte, se trata de un controlador que no requiere un conocimiento explícito del proceso, si no que basa su funcionamiento en la señal del error, por lo que permite obtener resultados satisfactorios para una amplia gama de procesos, aún con presencia de perturbaciones.

1.1. Selección del tipo de controlador

El controlador PID combina las acciones proporcional, integral y derivativa según el siguiente algoritmo:

$$\mathrm{u(t)} = \mathrm{K_p} e(t) + \int_0^t \frac{e(\tau)}{T_i} d\tau + T_d \frac{e(t)}{dt} = P + I + D \qquad (4)$$

- El término ***proporcional*** aporta una reducción al error estacionario, pero puede conllevar un aumento excesivo de la sobreoscilación y una tendencia a la inestabilidad.

- La ***acción integral*** aumenta el tipo del sistema, y por tanto, anulará en todo caso el error permanente ante un salto en la referencia.

- La ***acción derivativa*** realiza una predicción, en cierto modo, del error futuro permitiendo que la acción de control se anticipe a la respuesta.

La primera decisión que se debe tomar al diseñar un control PID es el tipo del controlador. Para ello, se pueden tener en cuenta las siguientes consideraciones:

- ***Controlador P:*** En ciertos tipos de procesos es posible trabajar con una ganancia elevada sin tener ningún problema de estabilidad en el controlador. Algunos procesos que poseen una constante de tiempo dominante o son integradores puros caen en esta categoría. Una alta ganancia en un controlador P significa que el error en estado estacionario será pequeño y no se necesitará incluir la acción integral.

- ***Controlador PD:*** El control PD puede ser apropiado cuando el proceso a controlar incorpore ya un integrador. Con esta clase de procesos es posible trabajar con ganancias elevadas en el controlador sin que sea necesario introducir la acción integral. La acción derivada es sensible al ruido ya que a altas frecuencias tiene una ganancia relativamente elevada, por lo tanto, en presencia de altos niveles de ruido se debe limitar dicha ganancia, o prescindir de la acción derivativa.

- ***Controlador PI:*** Es la estructura más usual del controlador. La introducción de la acción integral es la forma más simple de eliminar el error en estacionario. Es recomendable desconectar la acción derivativa es cuando el proceso está contaminado con niveles de ruido elevados. Como primera medida, se debería filtrar el ruido existente, pero en algunas ocasiones con esto no es suficiente. También es común utilizar la estructura PI hay retardos en el proceso, ya que la acción derivativa no resulta apropiada en este tipo de sistemas.

- ***Controlador PID:*** La acción derivativa suele mejorar el comportamiento del controlador, ya que permite aumentar las acciones proporcional e integral. Se emplea para mejorar el comportamiento de procesos que no poseen grandes retardos pero que presentan grandes desfases. Este es el caso típico de procesos con múltiples constantes de tiempo.

2. Sintonía de controladores PID

Una vez determinado el tipo de controlador que se va a implementar, se debe efectuar el ajuste de los parámetros (sintonía) para que la respuesta del sistema en lazo cerrado tenga unas características determinadas (criterio de sintonía). El ajuste de parámetros se convierte así en una tarea muy frecuente en plantas industriales, no sólo en los trabajos de puesta en marcha, sino también cuando se detectan cambios sustanciales de comportamiento en el proceso controlado.

En las primeras aplicaciones de control PID, el ajuste se basaba únicamente en la propia experiencia del usuario o en métodos analíticos. En 1942, Ziegler y Nichols propusieron técnicas empíricas que tuvieron buena aceptación, y que han servido de base a métodos más recientes.

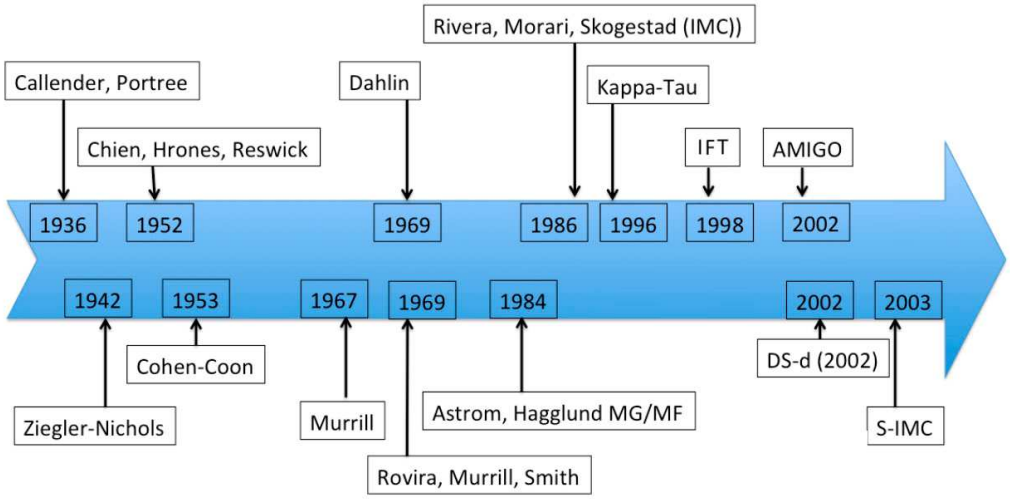


Figura 18 – Evolución de los métodos de autosintonía

Los métodos empíricos o experimentales de ajuste de parámetros están especialmente orientados al mundo industrial, donde obtener una descripción analítica de los procesos es complicado. Estos métodos constan fundamentalmente de dos pasos:

1. Estimación de ciertas características de la dinámica del proceso a controlar. La estimación se puede efectuar en lazo abierto o en lazo cerrado.
2. Calculo de los parámetros del controlador. Para ello se aplican las fórmulas de sintonía, es decir, relaciones empíricas entre los parámetros del controlador seleccionado y las características del proceso estimadas en el paso anterior.

El hecho de que estos métodos proporcionen sólo valores aproximados para los parámetros del controlador hace, generalmente, necesario un tercer paso. Con los parámetros de sintonía calculados y mediante observación de la respuesta en lazo cerrado, se realizará ajuste fino.

La diferencia entre los distintos métodos empíricos citados en la literatura radica en la forma de combinar las técnicas de estimación y las fórmulas de sintonía.

2.1. Criterios de sintonía

La sintonía de controladores PID para procesos industriales está basada normalmente en especificaciones nominales sobre determinadas características de la respuesta del sistema a cambios bruscos en el punto de consigna o en la carga.

Estas características no tienen por qué ser necesariamente una identificación del sistema o su aproximación por algún modelo, si no que puede tratarse de localizar y caracterizar algún punto particular de la respuesta, como ocurre en ciertos métodos de respuesta en frecuencia que se tratarán posteriormente.

También es usual basar el diseño en criterios de optimización sobre la señal de error, tratando de minimizar alguna de las cuatro integrales típicas de la señal de error: la integral del error (IE), la integral del cuadrado del error (ISE), la integral del valor absoluto del error (IAE) y la integral del valor absoluto del error ponderado en el tiempo (ITAE).

2.1.1. Métodos en lazo abierto

En general, no es posible describir completamente un proceso industrial, de ahí que se empleen para ello técnicas de aproximación en las cuales se asemeja la respuesta del sistema por el de un modelo matemático que se estime conveniente para el caso. Estas técnicas se basan en el hecho de que la mayoría de los procesos industriales son estables en lazo abierto y que la respuesta del proceso a ciertas señales de entrada puede aportar en muchos casos información suficiente para poder diseñar un controlador satisfactorio.

Los primeros precedentes en este tipo de diseño de controladores los podemos encontrar en los métodos de Ziegler-Nichols en lazo abierto.

En primer lugar, se requiere una aproximación del modelo por un sistema de primer orden con retardo para después, en base a los parámetros obtenidos en la identificación obtener los parámetros del controlador.

El caso de Ziegler-Nichols se diseñó con el fin de lograr controlar variaciones en el sistema provocadas por perturbaciones, no por cambios en la referencia, y obtener una respuesta tal que el segundo sobrepaso de la variable de salida sea menor que el 25% del sobrepaso inicial.

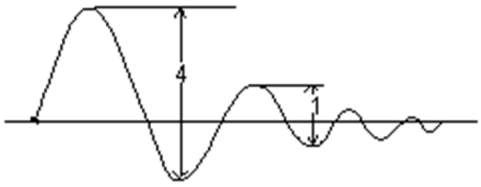


Figura 19 – Objetivo de Ziegler-Nichols en lazo abierto

Esta es sólo una opción de diseño. Existen multitud de tablas de sintonía en función del ajuste que se pretenda realizar y la respuesta del modelo que se busque.

El S-IMC de Skogestad que se explicará en posteriores apartados es un ejemplo de la evolución de esta metodología.

2.1.2. Métodos en lazo cerrado

Los métodos en lazo cerrado se basan en el hecho de que, ciertas características dinámicas de los procesos se pueden determinar a partir de su respuesta en frecuencia y que, por otro lado, hay un gran número de procesos que pueden llegar a ser oscilantes para obtener las características mencionadas.

Existen varios métodos experimentales para la determinación indirecta de un punto de la respuesta en frecuencia, concretamente para determinar la ganancia última (K_u) y el periodo de oscilación mantenida (T_u), definidos respectivamente como: la ganancia de un controlador proporcional a partir de la cual el sistema en lazo cerrado deja de ser estable, y el periodo de la oscilación que se consigue con ese valor de ganancia.

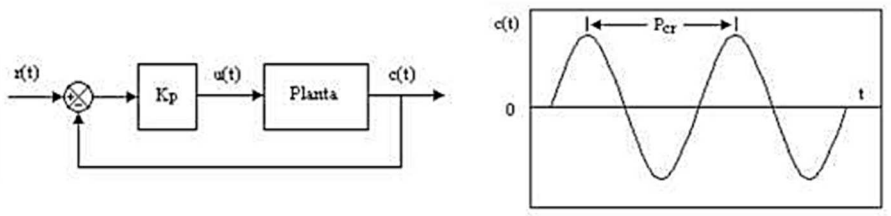


Figura 20 – Método de Ziegler-Nichols en lazo cerrado

Posteriormente, en función del valor que caracteriza a dicho punto, proporcionan el valor de los parámetros del controlador PID.

Los métodos más conocidos para la obtención de dichas características frecuenciales son el método de la oscilación mantenida, propuesto en 1942 por Ziegler y Nichols, y el método de identificación del relé, propuesto por Åström y Hagglund en 1984. Ambos serán analizados en profundidad más adelante.

2.1.3. Ajuste preciso: Métodos de optimización

En el apartado anterior de describió el ajuste preciso como un paso manual y posterior a la sintonía, pero esto no tiene por qué ser necesariamente así.

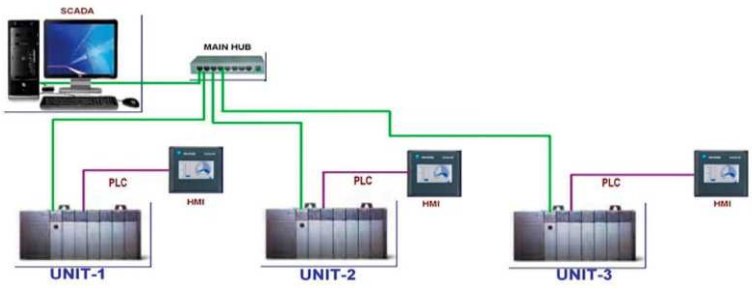


Figura 21 – Control supervisor: SCADA

El hecho de que los sistemas de regulación y control estén conectados a ordenadores a través de los sistemas SCADA ha posibilitado un aumento notorio en las posibilidades de almacenamiento de datos y ejecución cálculos. Por esta razón, hoy en día es posible aplicar métodos de optimización que supongan una alta carga computacional como pueden ser los métodos iterativos.

Los problemas de optimización pueden clasificarse en función de diferentes factores como son su complejidad, la existencia o no de restricciones, su carácter estático o dinámico, lineal o no lineal, mono-objetivo o multi-objetivo.

En cuanto a las técnicas de búsqueda, estas se pueden clasificar en función de si aseguran obtener el resultado óptimo (técnicas globales) o si por el contrario permiten obtener soluciones óptimas cercanas al punto de inicio (técnicas locales).

Las técnicas locales necesitan una condición de partida que no esté excesivamente alejada de donde se halla el óptimo, de manera que son una solución para automatizar el "ajuste preciso" del que se hablaba.

El método IFT, *Iterative Feedback Tuning*, de Hjalmarsson es un método mixto basado en la optimización matemática al que se añade una componente empírica.

3. S-IMC de Skogestad

El método SIMC se basa en las ideas clásicas de Ziegler-Nichols (1942), el IMC de Rivera (1986), y las reglas de sintonización de Smith y Corripio (1985). Los ajustes de Ziegler-Nichols dan como resultado una buena respuesta ante perturbaciones para los procesos con integradores. Se conocen otras maneras para dar lugar a ajustes más agresivos (Tyreus y Luyben 1992) (Åström y Hägglund 1995), aunque no proporcionan buenos resultados para los procesos con retraso dominante. Por otro lado, los ajustes de IMC de Rivera (1986) son conocidos por dar lugar a una mala respuesta ante perturbaciones para procesos integradores (Chien y Fruehauf 1990), (Horn, 1996), pero son robustos y generalmente dan muy buenas respuestas para los cambios de consigna. La regla de sintonía SIMC presenta un buen funcionamiento tanto para procesos integradores como para procesos con grandes retardos, y en ambos casos, tanto en variaciones de consigna como perturbaciones de carga.

El método S-IMC define un procedimiento de dos pasos para realizar la sintonía de un sistema.

1. Obtención de un modelo del sistema de primer o segundo orden con retardo (FOPDT/ SOPDT).
2. Cálculo de los parámetros de controlador PI, si se trata de un modelo FOPDT, o PID, si el modelo es SOPDT.

3.1. Paso 1. Obtención del modelo del sistema

El primer paso en el procedimiento de diseño del S-IMC es la obtención de un modelo aproximado de primer o segundo orden con retardo de la forma:

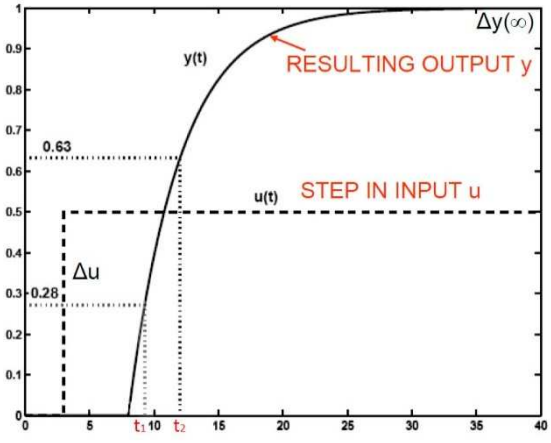
$$G(s) = \frac{k}{\tau_1 s + 1} e^{-\theta s} \tag{5}$$

$$G(s) = \frac{k}{(\tau_1 s + 1)(\tau_2 s + 1)} e^{-\theta s} \tag{6}$$

Donde:

- k ⟶ Ganancia de la planta
- τ_1 ⟶ Constante de tiempo del sistema
- θ ⟶ Tiempo de retardo (efectivo)
- τ_2 ⟶ (Opcional) Polo de segundo orden. Constante de tiempo τ_2

En la práctica, los parámetros del modelo de respuesta de primer orden son obtenidos a partir de la respuesta del sistema en lazo abierto ante una entrada escalón, tal y como se muestra en la Figura 22.



$$k = \frac{\Delta y}{\Delta u}$$

$$\tau_1 = 1.5 \cdot (t_2 - t_1)$$

$$\theta = t_2 - \tau_1$$

Figura 22 – Obtención de un modelo de primer orden con retardo

3.2. Paso 2. Obtención de los parámetros del controlador

El método SIMC establece que el controlador debe tomar una configuración PI para el caso de un modelo de un sistema de primer orden con retardo y PID si se trata de un sistema de segundo orden con retardo.

Para un sistema de primer orden de la forma:

$$G(s) = \frac{k}{\tau_1 s + 1} e^{-\theta s} \tag{7}$$

El método SIMC propone un controlador PI con los siguientes parámetros:

$$K_c = \frac{1}{k} \frac{\tau_1}{\tau_c + \theta} \tag{8}$$

$$\tau_I = \min\{\tau_1, 4(\tau_c + \theta)\} \tag{9}$$

Y en el caso de un modelo de segundo orden con retardo:

$$G(s) = \frac{k}{(\tau_1 s + 1)(\tau_2 s + 1)} e^{-\theta s} \tag{10}$$

Se sugiere un controlador PID con la configuración siguiente:

$$K_c = \frac{1}{k} \frac{\tau_1}{\tau_c + \theta} \tag{11}$$

$$\tau_I = \min\{\tau_1, 4(\tau_c + \theta)\} \tag{12}$$

$$\tau_D = \tau_2 \tag{13}$$

Se puede ver que en el cálculo de los parámetros del controlador aparece un único parámetro de configuración, denominado *constante de tiempo de lazo cerrado* τ_c. El valor de τ_c puede ser elegido libremente dentro del rango $-\theta < \tau_c < \infty$ para obtener una ganancia positiva y no nula. Un valor aceptable que respeta el compromiso existente entre la rapidez y la robustez de la respuesta es $\tau_c=\theta$. Fuera de ese valor, deberemos considerar que un desplazamiento hacia $-\theta$ generará una respuesta más rápida y oscilatoria y hacia infinito una respuesta robusta pero lenta.

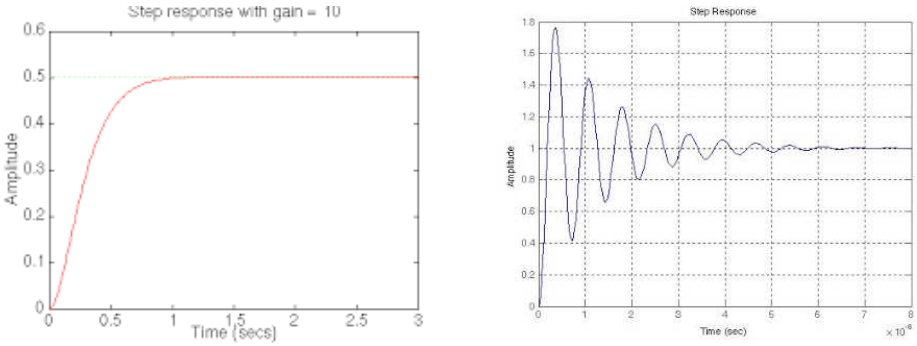


Figura 23 – Respuesta de primer/segundo orden

La estructura del controlador para la que se diseñó el método es un PID serie (14), de tal modo que si el procedimiento se va aplicar sobre un PID paralelo o ideal (15), se deberán aplicar las siguientes relaciones de conversión (16).

$$R_s(s) = K_c \cdot \left(1 + \frac{1}{\tau_I s}\right)(1 + \tau_D s) \tag{14}$$

$$R_p(s) = K_c + \frac{1}{\tau_I s} + \tau_D s \tag{15}$$

$$f = 1 + \frac{\tau_d}{\tau_I} = 1.6 \quad K'_c = K_c \cdot f \quad \tau'_I = \tau_I \cdot f \quad \tau'_d = \frac{\tau_d}{f} \tag{16}$$

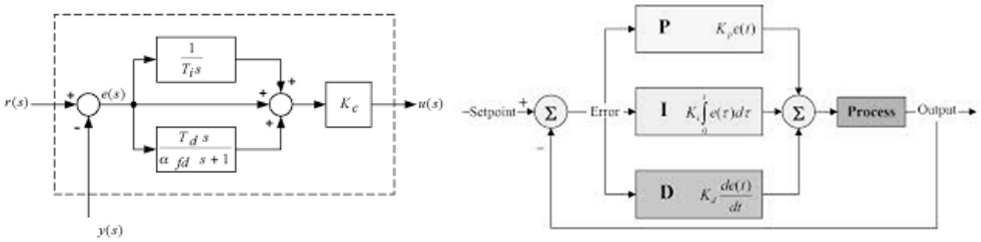


Figura 24 – Estructura de PID serie/paralelo

4. Método del relé

El método del relé, propuesto por Åström y Hagglund en 1984, constituye una forma indirecta de automatizar el método tradicional de la oscilación mantenida o método de Ziegler-Nichols en lazo cerrado.

4.1. Método de Ziegler-Nichols en lazo cerrado

El método de Ziegler-Nichols en lazo cerrado tiene como fin alcanzar el límite de estabilidad de la planta, donde el sistema se mantiene con oscilaciones sostenidas, o expresado de otro modo, identificar el punto del diagrama de Nyquist que coincide con la circunferencia de radio -1.

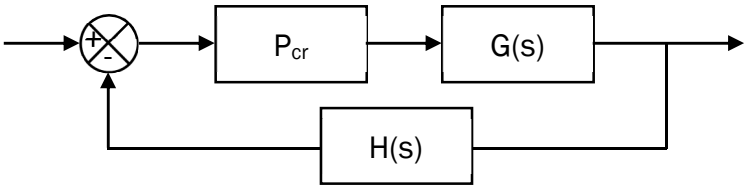


Figura 25 – Lazo de control con realimentación

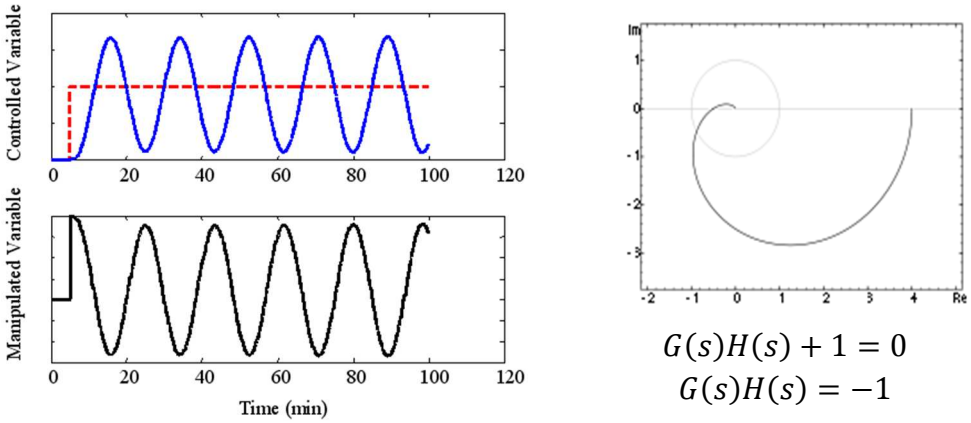
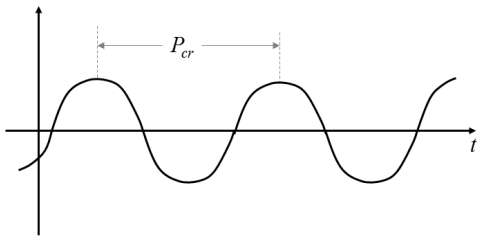


Figura 26 – Ganancia crítica del sistema

Para lograr dicho estado, el sistema debe funcionar en lazo cerrado con un controlador proporcional simple. La ganancia proporcional se aumenta manualmente hasta alcanzar el límite de estabilidad. En ese punto, se identifican la ganancia crítica K_{cr} y el periodo crítico P_{cr} con los que posteriormente se sintonizará el regulador.



Tipo	Ganancia K_p	Tiempo integral	Tiempo derivativo
P	0.5 K_c		
PI	0.45 K_c	T/1.2	
PID paralelo	0.75 K_c	T/1.6	T/10
PID serie	0.6 K_c	T/2	T/8

Figura 27 – Metodología de Ziegler-Nichols en lazo cerrado

El método de Ziegler-Nichols en lazo cerrado generalmente es arriesgado, ya que fuerza a la planta a operar cerca de la inestabilidad, además de que la amplitud de las oscilaciones no es controlable.

La inclusión en el lazo de control de un elemento no lineal, como puede ser un relé, permite igualmente alcanzar un ciclo límite que tendrá aproximadamente el mismo periodo T_c que la oscilación mantenida, y permitirá salvar algunos de los inconvenientes anteriormente mencionados. Esta técnica se denomina método de la función descriptiva.

4.2. Método de la función descriptiva

El método de la función descriptiva permite determinar la existencia de una oscilación en un sistema no lineal compuesto por un elemento lineal y de una no linealidad. El uso de la no linealidad genera armónicos y, si existe realimentación, los armónicos se propagan.

En caso de que la no linealidad sea estática, ésta puede ser aproximada por una ganancia, denominada N, equivalente y después usar las técnicas desarrolladas para los sistemas lineales, tales como el método de respuesta de frecuencia.

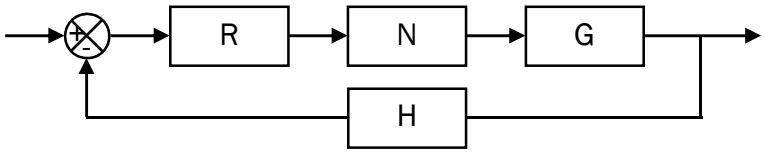


Figura 28 – Diagrama de bloques de la función descriptiva

Algunas de las no linealidades que pueden ser utilizadas para generar la respuesta deseada son: saturación, zona-muerta, relé, histéresis,... Entre ellos, la opción que tradicionalmente ha tenido una mayor aceptación es el relé.

4.3. Método del relé

El método del relé consiste en provocar un ciclo límite mediante la inclusión en el lazo de control de un elemento no lineal como es el relé, este ciclo límite tendrá aproximadamente el mismo periodo que la oscilación mantenida.

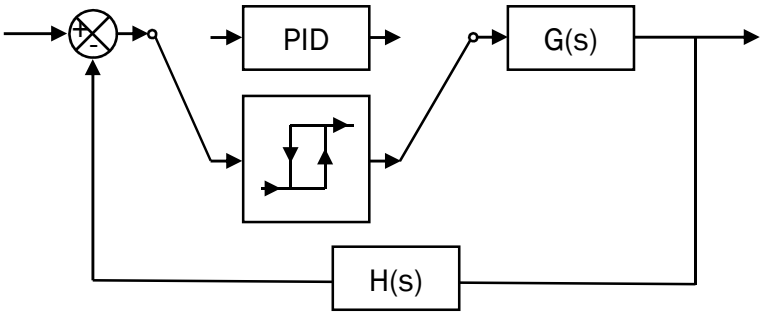


Figura 29 – Diagrama de bloques del método del relé

En la Figura 29, se observa el diagrama de bloques que representa al método propuesto. Para muchos sistemas la inclusión del relé provoca un comportamiento oscilatorio en el que la acción de control adopta conmuta del valor máximo al mínimo de una forma periódica.

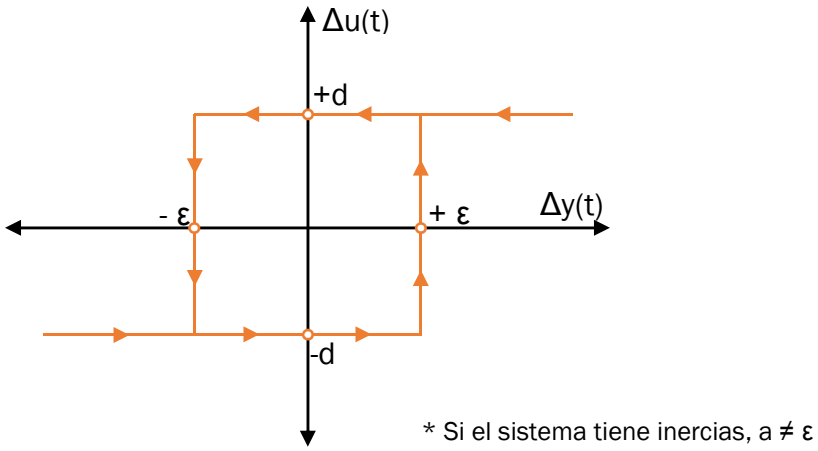


Figura 30 – Ciclo de histéresis para el método del relé

El uso un relé con histéresis presenta ciertas ventajas. Con un relé sin histéresis, el sistema será muy sensible al ruido, ya que un nivel de ruido reducido puede hacer conmutar de forma aleatoria la salida del relé. Introduciendo histéresis, el ruido debe ser mayor que la histéresis para forzar una conmutación en el relé.

Si *d* representa el incremento aplicado a la variable controlada, *ε* la amplitud de la variable manipulada en el ciclo de histéresis y *a* la amplitud de las oscilaciones resultantes (Véase Figura 30), la inversa de la función descriptiva viene dada por la siguiente expresión:

$$\frac{1}{N(A)} = \frac{\pi}{4d}\sqrt{(a^2 - \varepsilon^2)} - j\left(\frac{\pi\varepsilon}{4d}\right) \qquad (17)$$

Al ser la no linealidad dinámica, la función descriptiva tiene parte imaginaria. La parte imaginaria de la inversa de la función descriptiva resulta ser independiente de la amplitud a: $^{\pi\varepsilon}/_{4d}$.

De esto se deduce que la representación de – 1/N(a) en el plano complejo es una semirrecta paralela al eje real. Eligiendo convenientemente la relación entre la amplitud del relé y de la histéresis es posible determinar distintos puntos de corte con la función de trasferencia del sistema. De esta forma se pueden obtener puntos aproximados de la función de transferencia del sistema.

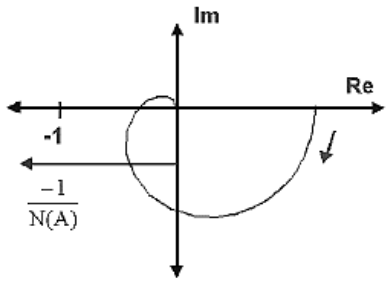


Figura 31 – Diagrama deNyquist del método del relé con histéresis

De la aplicación del método, se obtienen los datos de la ganancia crítica y el periodo crítico del sistema, a partir de los cuales se puede recurrir a las tablas de Ziegler-Nichols en lazo cerrado (Figura 27) para obtener los parámetros de sintonía del regulador PID.

La amplitud de la oscilación en la salida se puede controlar con facilidad eligiendo convenientemente la magnitud de la salida del relé, evitando las posibles situaciones sobrepaso de valores límite y controlando la posible tendencia a la inestabilidad del sistema.

$$K_{Cr} = \frac{4d}{\pi\sqrt{(a^2 - \varepsilon^2)}}$$

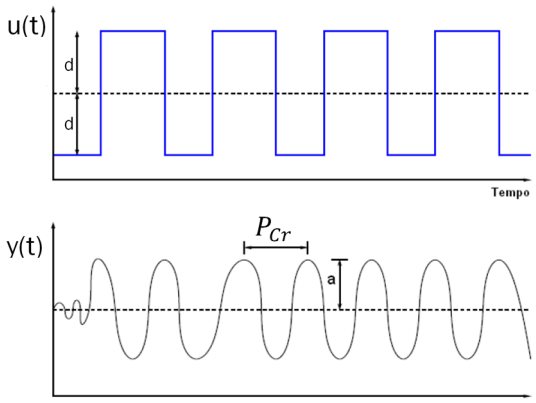


Figura 32 – Metodo del relé con histéresis

5. Método IFT

El *Iterative Feedback Tuning* (H. Hjalmarsson, 1994, 95, 98) se trata de un método de autosintonía en lazo cerrado para controladores PID, basado en la obtención de los parámetros del regulador a partir de la ejecución de una serie de experimentos y la posterior optimización de una función de costes.

El mayor logro del IFT se encuentra en el hecho de no requerir en ningún momento un conocimiento explícito del sistema, es decir, no es necesario ningún tipo de modelo previo ni se realiza ningún tipo de identificación durante el desarrollo, sino que, directamente, se obtienen los parámetros de configuración del regulador mediante la optimización iterativa de la función de costes.

5.1. Controlador de dos grados de libertad

A continuación, se detallan los desarrollos referidos a un sistema SISO de dos grados de libertad, aunque el procedimiento es adaptable a un solo grado de libertad.

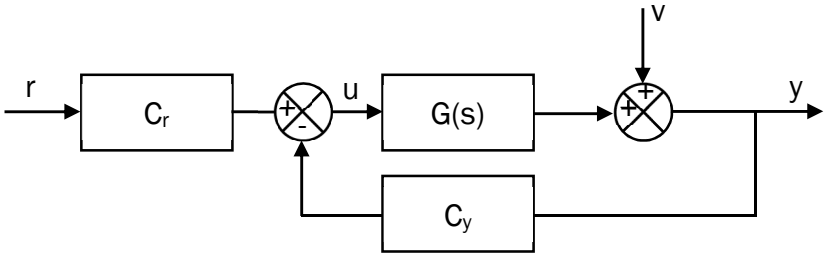


Figura 33 –Controlador de dos grados de libertad

Sea la función de costes una ecuación que depende del error de la variable controlada respecto del valor de consigna y del esfuerzo de control requerido según la siguiente expresión:

$$J(\rho) = \int_{t=0}^{N} e(t)^2 \cdot dt + \lambda \int_{t=0}^{N} u(t)^2 \cdot dt \tag{18}$$

$$\rho = [\mathrm{K_c}, \mathrm{T_i}, \mathrm{T_d}] \tag{19}$$

En la cual el error puede ser ponderado según el criterio ISE, integral del error cuadrático, o ITSE, integral del error cuadrático por el tiempo, en función de que se pretenda penalizar en mayor o menor medida el error en el estacionario.

$$ISE = \int_{t=0}^{N} e(t)^2 \cdot dt \qquad ITSE = \int_{t=0}^{N} t \cdot e(t)^2 \cdot dt \tag{20}$$

El término correspondiente a la variable manipulada se pondera mediante el coeficiente λ, que permite determinar el peso del esfuerzo de control en el total del coste.

Matemáticamente, el problema que se trata de resolver es la búsqueda de los parámetros ρ que minimizan dicha función de costes.

$$\rho^* = \arg\min J(\rho) \tag{21}$$

La manera de obtener estos valores será mediante una búsqueda iterativa basada en el método del gradiente descendiente.

$$\rho_{i+1} = \rho_i - \gamma_i \cdot H_i^{-1} \cdot \frac{\partial J(\rho_i)}{\partial \rho} \tag{22}$$

Donde, en cada iteración se proponen unos nuevos parámetros, ρ_{i+1}, en base a los existentes, ρ_i, al inverso del Hessiano de la función de costes, $\mathrm{H_i^{-1}}$, y a gradiente de ésta, $\partial \mathrm{J}(\rho_i)/\partial \rho$. La velocidad de convergencia del método en cada paso se puede regular mediante el parámetro γ.

Se trata de un algoritmo de optimización de primer orden. Para identificar mínimos locales, se van tomando pasos de aproximación descendentes en la dirección del gradiente o de una aproximación de éste.

No hay que olvidar el carácter local de este método, puesto que si la sintonía inicial contiene un mínimo local en su vecindad, no convergerá hacia el mínimo global de la función, aunque sí se puede garantizar, si se parte configuración adecuada, una mejora en la sintonía inicial del controlador en un número reducido de iteraciones.

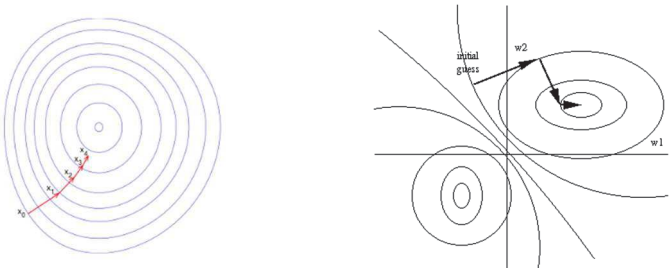
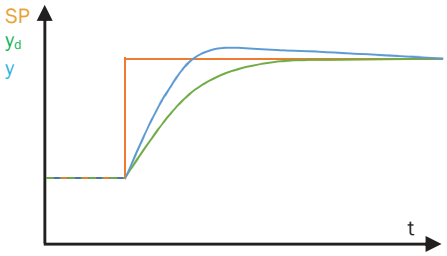


Figura 34 – Optimización local: un solo mínimo o varios mínimos

Si expresamos la función de costes en términos de una función discreta tenemos la siguiente expresión, en la cual el error ha pasado a ser expresado como la diferencia entre la salida del sistema en lazo cerrado y la salida deseada.

$$J(\rho) = \frac{1}{2N}\left[\sum_{t=1}^{N} \tilde{y}_t(\rho)^2 \; + \; \lambda \sum_{t=1}^{N} u_t(\rho)^2\right] \qquad (23)$$



$$\tilde{y}(\rho) = y(\rho) - y_d(\rho) \qquad (24)$$

$$y_d = \frac{1-\alpha}{1-\alpha \cdot z} \qquad (25)$$

Figura 35 – Respuesta real del sistema frente a respuesta deseada

Para poder obtener los nuevos valores del vector ρ, es necesario conocer el gradiente de la función de costes, o en este caso, una estimación de él.

$$est\left[\frac{\partial J(\rho)}{\partial \rho}\right] = \frac{1}{N}\left[\sum_{t=1}^{N} \tilde{y}_t(\rho) \cdot est\left(\frac{\partial \tilde{y}_t(\rho)}{\partial \rho}\right) + \lambda \sum_{t=1}^{N} u_t(\rho) \cdot est\left(\frac{\partial u_t(\rho)}{\partial \rho}\right)\right] \qquad (26)$$

Es importante recalcar que, puesto que $\tilde{y} = y - y_d$ e y_d es constante, se verifica que $\partial\tilde{y}(\rho)/\partial\rho = \partial y\,(\rho)/\partial\rho$.

De la expresión anterior se deduce que para obtener el gradiente de la función de costes, se requieren los gradientes de las variables controlada y manipulada del sistema.

Partiendo de las expresiones matemáticas de cada una de ellas se obtienen los gradientes del siguiente modo:

$$y(\rho) = \frac{C_r(\rho)\,G_0}{1 + C_y(\rho)\,G_0}\cdot r + \frac{1}{1 + C_y(\rho)\,G_0}\cdot v = T_0 \cdot r + S_0 \cdot v \qquad (27)$$

Si calculamos el gradiente de la variable de salida del sistema, obtenemos:

$$\frac{\partial y(\rho)}{\partial\rho} = \frac{G_0}{1 + C_y(\rho)\,G_0}\frac{\partial C_r(\rho)}{\partial\rho}\,r - \frac{C_r(\rho)\,G_0^2}{\left(1 + C_y(\rho)\,G_0\right)^2}\frac{\partial C_y(\rho)}{\partial\rho}\,r - \frac{G_0}{\left(1 + C_y(\rho)\,G_0\right)^2}\frac{\partial C_y(\rho)}{\partial\rho}\,v$$

$$= \frac{1}{C_r}\cdot T_0 \cdot \frac{\partial C_r(\rho)}{\partial\rho}\cdot r - \frac{1}{C_r}\cdot T_0^2 \cdot \frac{\partial C_y(\rho)}{\partial\rho}\cdot r - \frac{1}{C_r}\cdot T_0 \cdot S_0 \cdot \frac{\partial C_y(\rho)}{\partial\rho}\cdot v \qquad (28)$$

Donde:

$$[T_0]^2 \cdot r + T_0 \cdot S_0 \cdot v = T_0 \cdot y \qquad (29)$$

De modo que la expresión (28) puede reescribirse como:

$$\frac{\partial y(\rho)}{\partial\rho} = \frac{1}{C_r}\left[\left(\frac{\partial C_r(\rho)}{\partial\rho} - \frac{\partial C_y(\rho)}{\partial\rho}\right)\cdot T_0 \cdot r + \frac{\partial C_y(\rho)}{\partial\rho}\cdot T_0 \cdot (r - y)\right] \qquad (30)$$

Análogamente se puede obtener el gradiente correspondiente a la variable de control del sistema:

$$u(\rho) = \frac{C_r(\rho)}{1 + C_y(\rho)\,G_0}\cdot r - \frac{C_y(\rho)}{1 + C_y(\rho)\,G_0}\cdot v$$

$$= C_r(\rho)\cdot S_0 \cdot r - C_y(\rho)\cdot S_0 \cdot v \qquad (31)$$

Calculando el gradiente obtenemos:

$$\frac{\partial S_0(\rho)}{\partial \rho} = -\frac{1}{C_r} \cdot T_0 \cdot S_0 \cdot \frac{\partial C_y(\rho)}{\partial \rho} \tag{32}$$

$$\frac{\partial u(\rho)}{\partial \rho} = \mathrm{S}_0 \cdot \frac{\partial C_r(\rho)}{\partial \rho} \cdot r - T_0 \cdot S_0 \cdot \frac{\partial C_y(\rho)}{\partial \rho} \cdot r - \mathrm{S}_0 \cdot \frac{\partial C_y(\rho)}{\partial \rho} \cdot v + \frac{C_y}{C_r} \cdot T_0 \cdot S_0 \cdot \frac{\partial C_y(\rho)}{\partial \rho} \cdot v$$

$$= S_0(\rho) \left[\frac{\partial C_r(\rho)}{\partial \rho} \cdot r - \frac{\partial C_y(\rho)}{\partial \rho} (T_0(\rho) \cdot r + S_0(\rho) \cdot v) \right] \tag{33}$$

Donde:

$$y = T_0(\rho) \cdot r + S_0(\rho) \cdot v \tag{34}$$

$$\frac{\partial u(\rho)}{\partial \rho} = S_0(\rho) \left[\frac{\partial C_r(\rho)}{\partial \rho} \cdot r - \frac{\partial C_y(\rho)}{\partial \rho} y \right] \tag{35}$$

$$\frac{\partial u(\rho)}{\partial \rho} = S_0 \cdot \left[\frac{\partial C_r(\rho)}{\partial \rho} - \frac{\partial C_y(\rho)}{\partial \rho} \cdot r + \frac{\partial C_y(\rho)}{\partial \rho} \cdot (r - y) \right] \tag{36}$$

Como se puede observar, los gradientes dependen de la desconocida función de transferencia de la planta G_0, por medio de S_0 y T_0. Los creadores del algoritmo IFT sugieren, basándose en la forma de las ecuaciones anteriores, que ambos gradientes podrían ser estimados mediante una serie ensayos sobre el sistema.

La metodología propuesta para la realización de dichos experimentos es la que sigue:

1. Se realiza un primer experimento en el que se aplica una entrada Δr sobre el valor inicial de la referencia y se recogen N muestras de la variable de control (u_1) y la salida del sistema (y_1).

2. Se realiza un segundo experimento en el cual se vuelven a recoger N muestras u_2 e y_2, con la diferencia de que la referencia varía para cada una de ellas en la forma (Δr- y_1).

3. Se realiza un tercer experimento con las mismas condiciones que el experimento inicial, aplicando una entrada Δr y recogiendo N muestras u_3 e y_3.

4. Se calculan las estimaciones de los gradientes de las señales de control y de salida del sistema aplicando los filtros que se señalan a continuación, dependientes tan solo de los parámetros del controlador, y por tanto, conocidos.

$$est\ \frac{\partial y(\rho)}{\partial \rho} = \frac{1}{C_r(\rho)}\left[\left(\frac{\partial C_r}{\partial \rho} - \frac{\partial C_y}{\partial \rho}\right) y_3 \ + \frac{\partial C_y}{\partial \rho} y_2 \right] \qquad (37)$$

$$est \frac{\partial u(\rho)}{\partial \rho} = \frac{1}{C_r(\rho)}\left[\left(\frac{\partial C_r}{\partial \rho} - \frac{\partial C_y}{\partial \rho}\right) u_3 \ + \frac{\partial C_y}{\partial \rho} u_2 \right] \qquad (38)$$

5. Una vez conocidos estos gradientes, es inmediata la obtención del valor del gradiente de la función de costes y, por tanto, la aplicación del algoritmo iterativo de optimización.

Es importante esclarecer el por qué es necesario realizar un tercer experimento. Se pudiera pensar en utilizar en lugar de y3, el resultado del primer experimento (y1) que fue realizado en definitiva, bajo las mismas condiciones.

En realidad no han sido exactamente las mismas condiciones porque estaban presenten otras perturbaciones en el lazo. La derivación formal del método IFT asume que las perturbaciones están compuestas por ruido aleatorio y que las perturbaciones de un experimento no está correlacionas con la del siguiente. Para el caso de 2DoF la única manera de garantizar que el estimador obtenido sea no sesgado pasa por la condición de no reutilizar el resultado del experimento 1 sino de realizar otro nuevo.

5.2. Controlador de un grado de libertad

Para el caso de controladores de un solo grado se libertad, el algoritmo sufre las siguientes modificaciones.

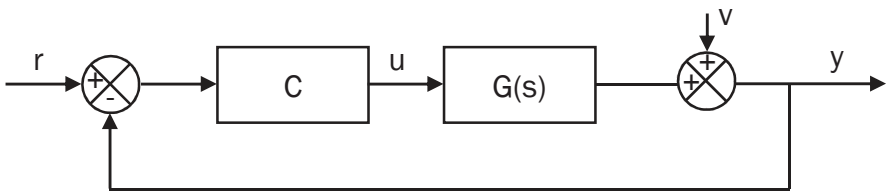


Figura 36 – Controlador de un grado de libertad

De este modo, las nuevas funciones de transferencia T_0 y S_0 son ahora:

$$T_0 = \frac{C(\rho)\, G_0}{1 + C(\rho)\, G_0} \qquad\qquad S_0 = \frac{1}{1 + C(\rho)\, G_0} \tag{39}$$

Comparando con las obtenidas para el supuesto del controlador de dos grados de libertad, podemos concluir que ambas expresiones son equivalentes si se aplica la siguiente relación:

$$C_y = C_r = C \tag{40}$$

Partiendo de las ecuaciones (30) y (35) y aplicando la relación descrita en (40), se ven simplificados los gradientes de las señales de control y salida del sistema de la siguiente manera.

$$\frac{\partial y(\rho)}{\partial \rho} = \frac{1}{\mathrm{C}}\left[\frac{\partial \mathrm{C}}{\partial \rho} \cdot T_0 \cdot (r - y)\right] \tag{41}$$

$$\frac{\partial u(\rho)}{\partial \rho} = \left[\frac{\partial \mathrm{C}}{\partial \rho} \cdot S_0 \cdot (r - y)\right] \tag{42}$$

En este caso, se puede prescindir del tercer experimento, quedando la metodología resumida a los dos primeros.

1. Se realiza un primer experimento en el que se aplica una entrada Δr sobre el valor inicial de la referencia y se recogen N muestras de la variable de control (u_1) y la salida del sistema (y_1).

2. Se realiza un segundo experimento en el cual se vuelven a recoger N muestras u_2 e y_2, con la diferencia de que la referencia varía para cada una de ellas en la forma ($\Delta r - y_1$).

3. Se calculan las estimaciones de los gradientes de las señales de control y de salida del sistema aplicando los filtros que se señalan a continuación, dependientes tan solo de los parámetros del controlador, y por tanto, conocidos.

$$est\ \frac{\partial y}{\partial \rho} = \frac{1}{C}\left[\frac{\partial C}{\partial \rho} y_2\right] \tag{43}$$

$$est\ \frac{\partial u}{\partial \rho} = \frac{1}{C}\left[\frac{\partial C}{\partial \rho} u_2\right] \tag{44}$$

4. Una vez conocidos estos gradientes, es inmediata la obtención del valor del gradiente de la función de costes y, por tanto, la aplicación del algoritmo iterativo de optimización.

A continuación, se muestra un ejemplo de la ejecución del algoritmo.

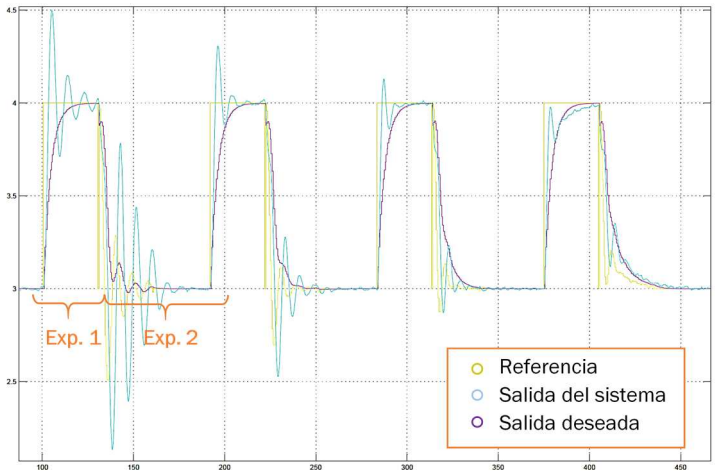


Figura 37 – Ejemplo de ejecución del IFT

CAPITULO IV:

IMPLEMENTACIÓN DE LA HERRAMIENTA DE AUTOSINTONÍA

1. Introducción: Filosofía

Es importante no olvidar que la aplicación de autosintonía que se va a desarrollar va a tener un uso industrial, por lo que se deberá cumplir con ciertas pautas en todo momento.

En primer lugar, hay que tener en cuenta que la herramienta irá embebida en el sistema UNICOS. Esto implica que se deberá respetar su jerarquía y considerar su estructura para no incurrir en contradicciones con su lógica ni generar fallos de seguridad.

A parte de los requisitos de UNICOS, el diseño del sistema debe ser suficientemente robusto para trabajar con cualquier proceso de forma segura. La mayoría de los métodos de autosintonía necesitan realizar experimentos que llevarán al sistema fuera de su punto de trabajo. Cuando esto ocurra, el procedimiento debe ser capaz de detectar si las variables del sistema exceden los límites propuestos o si está experimentando una tendencia a la inestabilidad.

Y en última instancia, se debe tener en cuenta al usuario de la aplicación. La herramienta de autosintonía debe estar lo más simplificada posible y ser visual e intuitiva en su utilización.

2. Estructura de funcionamiento del controlador

La Figura 38 muestra una estructura completa del funcionamiento del objeto *Controlador* tal y como lo define UNICOS.

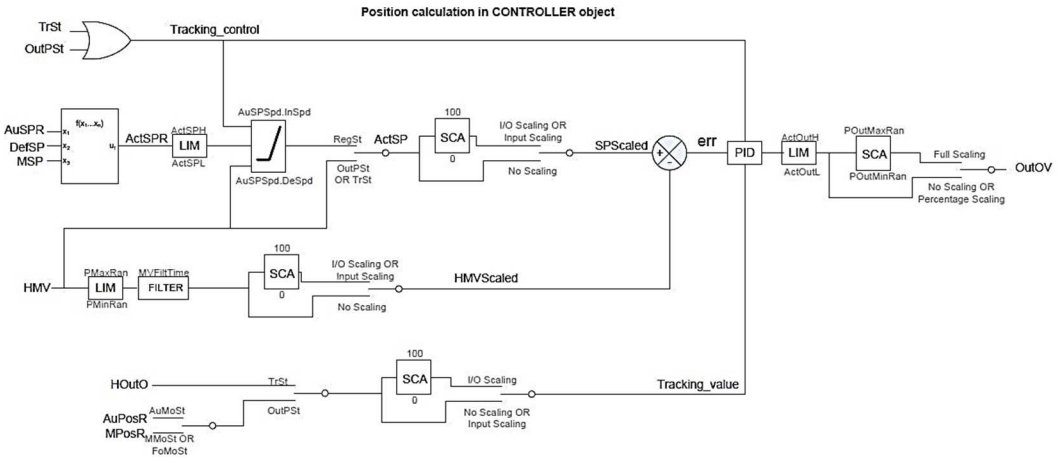


Figura 38 – Esquema de funcionamiento del controlador

En el esquema se distinguen distintos modos de funcionamiento en dependencia del modo de operación y el estado de trabajo seleccionado.

2.1. Modos de operación

En el Capítulo II se trató el tema de los modos de operación de forma genérica, es decir, aplicado a cualquier objeto UNICOS. Si nos referimos en particular al objeto *Controlador* hay algunas consideraciones particulares que se deben tener también en cuenta.

- *Modo Automático:*
  - Cuando se cambia a modo Automático:
    - Se mantienen los parámetros *Activos*.
  - No es posible cambiar estos parámetros
  - No es posible guardar/restaurar estos parámetros
  - No es posible conmutar entre los estados de trabajo *Regulación/Posicionamiento de Salida* manualmente.
  - El modo *Seguimiento* se habilita y deshabilita automáticamente.

- *Modo Manual/Forzado:*
  - Cuando se cambia a modo *Manual/Forzado*:
    - Se mantienen los parámetros *Activos*.
    - Se mantiene el último estado de trabajo (*Regulación/Posicionamiento de la salida*).
  - Todos los parámetros pueden ser modificados con solicitudes manuales si no hay inhibiciones.
  - Es posible guardar nuevos parámetros por defecto desde el panel principal del PVSS.
  - Es posible restaurar los parámetros por defecto desde el panel principal del PVSS.
  - Es posible conmutar entre los estados de trabajo *Regulación/Posicionamiento de Salida* con una solicitud manual.
  - Al cambiar el estado de trabajo a *Posicionamiento de Salida*, el valor de inicialización de la variable de control será el del último valor de salida activo.
  - El modo *Seguimiento* es habilitado y deshabilitado de forma automática.

2.2. Estados de trabajo

El *Controlador* tiene tres estados de trabajo disponibles para todos los modos de operación:

- ***Regulación (Regulation):*** El funcionamiento se corresponde al de un lazo cerrado de control, en el que la señal de control será emitida por el algoritmo PID que se encuentre configurado. Se fija un valor de consigna para la variable controlada y el regulador determina la acción de control necesaria para que la salida del sistema coincida con la referencia.
- ***Posicionamiento de salida (Output Positioning):*** El funcionamiento se corresponde con un bucle de control en lazo abierto, en el que el regulador aplica al sistema un determinado valor de señal de control fijado. El valor de la señal de control se asigna en la variable correspondiente con el modo de operación que se encuentre activo en ese instante, automático o manual.
- ***Seguimiento (Tracking):*** Igualmente el funcionamiento es el de un bucle en lazo abierto, aunque ahora el valor de fijado para la variable de control será el propio valor de salida en el instante de activación de este estado de trabajo. Se trata de un modo de funcionamiento muy útil para las transferencias *bumpless* automático/manual y viceversa.

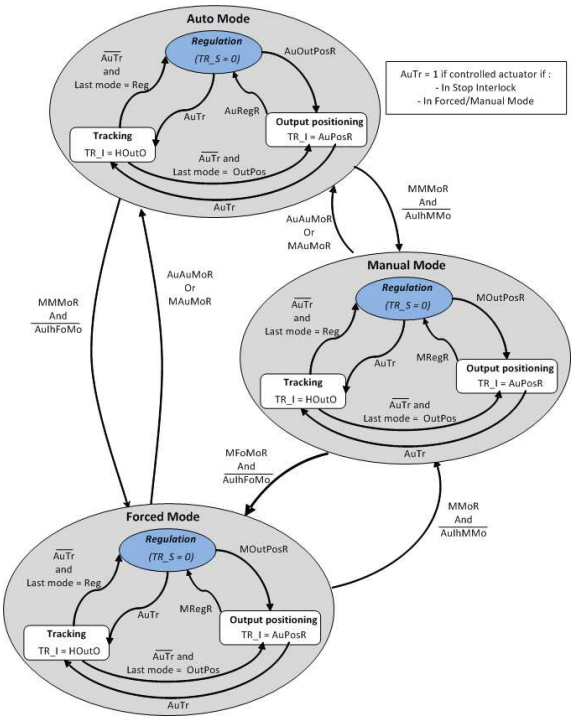


Figura 39 – Relacion entre modos de funcionamiento y estados de trabajo

2.3. Límites del proceso

Al comienzo de la creación del proyecto, cuando se completó la hoja de especificaciones con las variables del proceso, se identificó para cada una de ellas el rango de valores admisible que podía tomar, sus límites físicos.

Aunque estos límites de seguridad para la variable ya existan, no tienen por qué ser los mismos límites que se espera que tenga el lazo de regulación en cuestión a la hora de funcionar. De hecho, lo normal es que no sean los mismos y para los límites del proceso los valores sean más restrictivos que los anteriores.

Por esta razón se define un nuevo intervalo, tanto para las entradas como para las salidas del controlador. Este intervalo es por defecto el definido en el *Template* con que se generó el proyecto, pero puede ser redefinido a un intervalo más reducido.

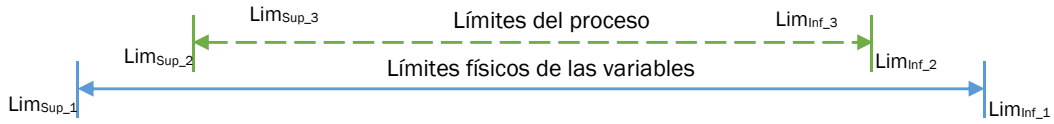


Figura 40 – Límites del proceso

2.4. Factor de escalado

Si nos remitimos de nuevo a la Figura 38 se puede ver que para cualquiera de los estados de trabajo, tanto la variable de control como la de salida pueden estar expresadas en unidades de ingeniería o escaladas y expresadas en tanto por cien. En función de cuáles de estas variables estén escaladas aparecen tres posibilidades:

- ***Opción 1: Entrada al controlador escalada.***
  - El *Set Point* y la variable manipulada están escalados del rango [Lím$_{sup}$, Lím$_{inf}$] → [0, 100]%
  - La variable de salida queda limitada al intervalo [0, 100]%
  - El parámetro K_c no tiene su significado físico, ya que está actuando en valores escalados.

- *Opción 2: Entrada y salida del controlador escalados*
  - El *Set Point* y la variable manipulada están escalados del rango [$Lím_{sup}$, $Lím_{inf}$] → [0, 100]%
  - La variable de salida es escalada de [0, 100]% → [$Lím_{sup}$, $Lím_{inf}$]
  - El parámetro K_c no tiene su significado físico, ya que está actuando en valores escalados.

- *Opción 3: Sin escalado*
  - El *SP* y la variable manipulada se usan directamente, sin escala.
  - La variable de salida queda limitada al intervalo [$Lím_{sup}$, $Lím_{inf}$]
  - El parámetro K_c mantiene su significado físico

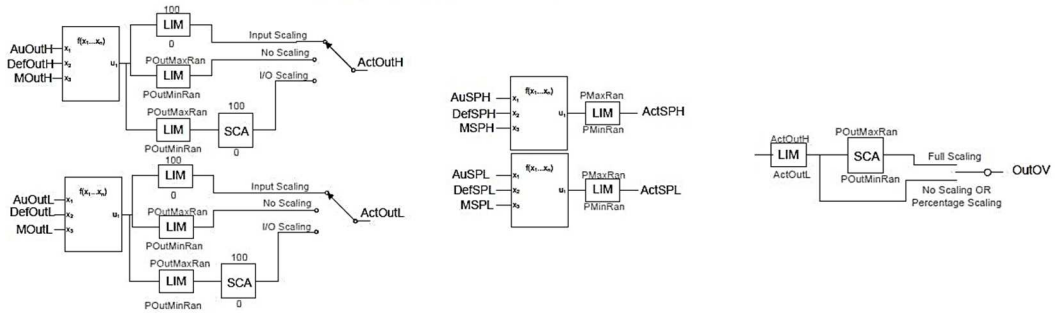


Figura 41 – Limites y escalado de las variables del proceso

3. Implementación de la herramienta de autosintonía

3.1. Consideraciones generales para todos los métodos

Algunas de las características que se deben tener en cuenta, como pueden ser las características de la planta o los parámetros de funcionamiento, son comunes a todos los métodos. Por esta razón, se van a definir de manera general, aunque después cada uno de los algoritmos de sintonía recurrirá a ellos cuando sea preciso.

3.1.1. Estructura del controlador

La estructura del controlador dependerá del sistema que se esté tratando, las posibilidades que se ofrecen son PI o PID. En el Capítulo III se ofrecen unas pautas sobre cómo decidir la estructura.

3.1.2. Límites del proceso de autosintonía

Los límites entre los que puede trabajar el sistema durante la autosintonía son el primer ejemplo de consideraciones generales. Las variables del proceso no podrán superar en ningún momento estos límites. Esto se tiene en cuenta tanto a la hora de introducir los parámetros de configuración de los métodos impidiendo que se acepten datos que excedan el rango permitido, como durante la ejecución de los experimentos.

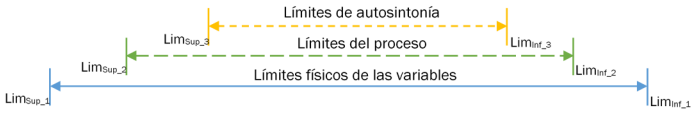


Figura 42 – Límites durante la ejecución de los métodos de sintonía

Los límites para la autosintonía se inicializan con el valor de los límites del proceso, siendo siempre el rango que cubren los primeros menor o igual que estos últimos, como se indica en la Figura 42.

3.1.3. Tiempo de muestreo

Todos los métodos requieren un análisis de la respuesta del sistema, por lo que es importante definir para la recogida de datos el periodo con que se muestrean. Como es bien sabido, el número de datos que se recogen en un experimento es muy significativo, porque tanto el exceso como el defecto pueden dificultar después el análisis de las muestras. Si se toman valores con una frecuencia mayor que la necesaria, se recogerán ruidos innecesarios y, si la frecuencia no es suficiente, se perderán características relevantes de la respuesta del proceso.

El periodo de muestreo es un parámetro que debe estimar el usuario en base a su conocimiento del proceso. Para parametrizarlo correctamente, se sugiere considerar que, en un experimento en lazo abierto en el que se aplique una entrada escalón, se deben recoger unas 30 muestras desde el comienzo y hasta que el sistema se estabilice.

3.1.4. Acción directa/inversa del controlador

El hecho de que la ganancia de un sistema sea positiva o negativa, indica si la relación entre la entrada y la salida de la planta vayan en el mismo sentido (acción inversa) o en sentidos opuestos (acción directa). Esta característica deberá ser considerada en los métodos en los que sea relevante.

3.1.5. Escalado de la entrada/salida

Como se explicó anteriormente, es posible aplicar un escalado tanto a la entrada como a la salida del sistema, y esto puede modificar el significado físico del término proporcional del PID. Si así sucediera, podría ser necesaria una corrección del término proporcional del controlador PID calculado.

Analizando los métodos implementados, se ha concluido que todos ellos absorberán estos escalados como parte del funcionamiento normal de la planta considerándolo, de algún modo, como una ganancia añadida a la función de transferencia del sistema. Por esta razón, esta acción no afectará al resultado final y no será necesario adoptar medidas adicionales para corregirlo.

3.2. Método S-IMC

El S-IMC de Skogestad es un método que parte de una identificación en lazo abierto para después recurrir con los parámetros del sistema a una tabla de sintonía para obtener los parámetros del PID.

En la ejecución de este procedimiento hay dos funciones claramente diferenciadas. Por un lado, se encuentra la identificación del modelo mediante un experimento en lazo abierto y por otro, la obtención de los parámetros PID del regulador a partir de un modelo de función de transferencia conocido.

Estos dos bloques, aunque pertenezcan a un mismo método, se analizarán de forma individual, para así favorecer la adición de nuevos algoritmos de sintonía basados en la identificación de modelos en proyectos futuros.

El primer paso para la implementación será identificar las operaciones unitarias en las que se descompone el método, que en este caso son las siguientes:

a) Identificación

1. ***Detección de estado estacionario:*** la primera condición necesaria previa a comenzar el proceso de identificación del sistema es que la planta se encuentre ya estabilizada en torno al punto de trabajo, es decir, que haya alcanzado el estacionario.
2. ***Entrada escalón:*** a continuación se procede a perturbar el sistema aplicando una entrada escalón de amplitud determinada por el usuario.

3. ***Detección de estado estacionario:*** es necesario que la planta alcance su nuevo estado estacionario.
4. ***Identificación del sistema:*** analizando la forma de la respuesta se realizan los cálculos necesarios para la aproximación del sistema por un modelo de primer orden con retardo.

b) <u>Sintonía</u>

5. ***Cálculo de los parámetros de sintonía del regulador:*** aplicando las reglas de Skogestad se obtienen los nuevos valores de configuración del PID.

Se puede pensar que los pasos 1, 2, 3 y 4 de la identificación no siempre tienen por qué ser necesarios. Si en algún momento anterior se ha obtenido el modelo del sistema, el algoritmo debe ser capaz de reutilizar esta información si se solicita, o si la función de transferencia se conoce previamente se debe de poder introducir al sistema esta información.

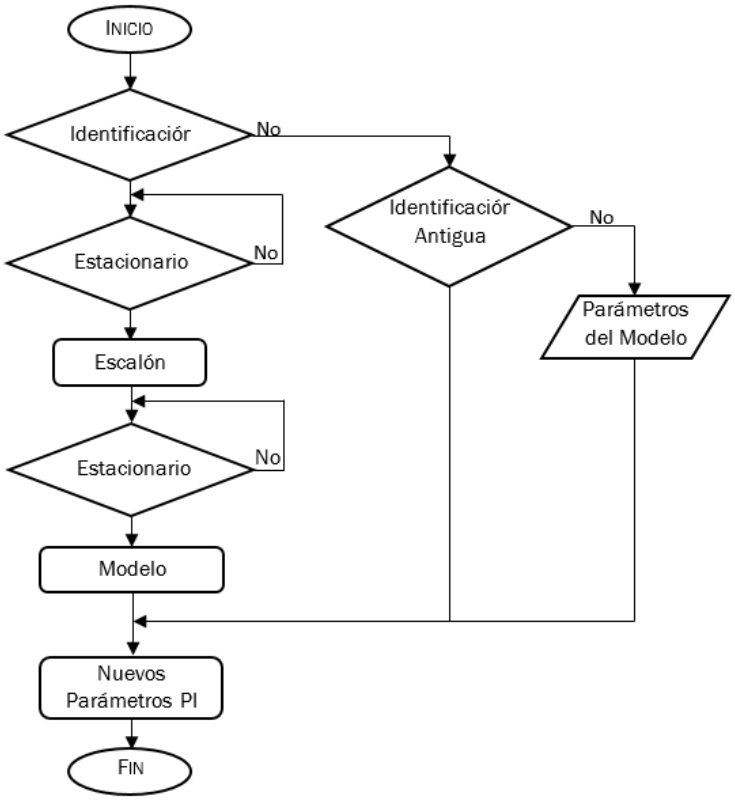


Figura 43 – Diagrama de flujo del método S-IMC

A continuación se analizará la forma de ejecutar cada uno de los pasos de forma más detallada.

3.2.1. Identificación: Aproximación por un modelo FOPD

La identificación es un experimento en lazo abierto que necesitará realizar cambios en el valor de la variable manipulada del proceso, por lo que es necesario que, durante la ejecución el modo de operación sea *Manual* y el estado de trabajo *Output Positioning*.

3.2.1.1. Detección de estado estacionario

Antes de comenzar con la identificación del sistema se debe asegurar que el estado de partida del sistema es estable. Para determinar que la planta está trabajando en estacionario se ha utilizado un algoritmo basado en un criterio estadístico, en concreto utilizando la suma iterativa de las desviaciones al cuadrado (SSID).

Este método, propuesto por R. Russell Rhinehart en la *American Control Conference* en 2013, consiste en calcular la suma de las desviaciones al cuadrado (SSD) de un subconjunto aleatorio de datos (un subconjunto único, seleccionados al azar en cada iteración). La suma de las desviaciones al cuadrado del subconjunto aleatorio (RS SSD) aparecerá como una señal ruidosa que va convergiendo al valor de ruido que tendrá en estacionario según avancen las iteraciones. Se considerará que se ha alcanzado el estacionario cuando RS SDD esté dentro de unas bandas de confianza.

La implementación de este algoritmo requiere la presencia de ruido en el sistema, pero en las plantas del CERN se da el caso de que no lo tienen, debido a que están filtradas. Para solventar el problema, se ha sumado al valor de la variable leída un ruido blanco, que no afecta a las propiedades estadísticas del sistema que interpreta el método.

3.2.1.2. Entrada escalón

Una vez se ha asegurado que la planta está estabilizada, se procede a perturbar el sistema aplicando un escalón a la variable de entrada. El valor del incremento será un valor fijado por el usuario, pero para que el resultado sea válido deben tenerse en cuenta las siguientes consideraciones:

- Si el incremento es demasiado grande, alejará al sistema de su punto de trabajo.

- Si es demasiado pequeño, la perturbación puede ser imperceptible para el sistema si éste es ruidoso.
- Un criterio bastante usual es aplicar un escalón cuyo valor sea el triple del ruido existente.

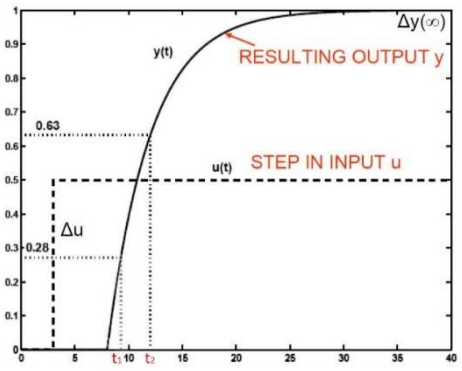
3.2.1.3. Detección de estado estacionario

Al aplicar el escalón hemos sacado al sistema de su estado de estabilidad. Utilizando el mismo algoritmo que se utilizó en el apartado 1 se esperará a que el sistema alcance el nuevo estacionario.

Los datos de la variable de salida en cada iteración, así como el instante en que se toman serán almacenados en sendos vectores ($\bar{y}$ y $\bar{t}$) para ser utilizados en la obtención del modelo.

3.2.1.4. Identificación del sistema

La metodología que se va a aplicar en la obtención del modelo se muestra en la Figura 44.



$$k = \frac{\Delta y}{\Delta u}$$

$$\tau_1 = 1.5 \cdot (t_2 - t_1)$$

$$\theta = t_2 - \tau_1$$

Figura 44 – Obtención del modelo FOPD

En primer lugar, se identifican los valores inicial y final de la controlada del sistema para obtener los incrementos Δy e Δu:

- Δu: valor del escalón aplicado a la señal de control.
- y_{In}: valor de la salida del sistema al inicio del paso 3.
- y_{Fin}: valor de la salida del sistema al final del paso 3.

A continuación, se localizan los puntos t_1 y t_2 en la respuesta del sistema del siguiente modo:

- Se calcula el 63.2% y el 28.3% del Δy.

$$t_1 = 0.283(yExp[N] - yExp[1]) + yExp[1]$$

$$t_2 = 0.632(yExp[N] - yExp[1]) + yExp[1]$$

- Se localiza en el vector $\bar{y}$ los dos valores entre los que se encuentra cada uno de estos datos.
- Se realiza una interpolación lineal para obtener t_1 y t_2 de forma precisa (Figura 45).

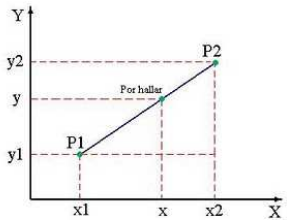


Figura 45 – Interpolación lineal

- Se calcula según las ecuaciones que aparecen en la Figura 44 los parámetros del modelo.
- El retardo de un sistema discreto debe ser mayor o igual que un periodo de muestreo, de modo que si en el cálculo resultara menor, se modifica su valor al mínimo aceptable.

Una vez finalizado e último paso, el sistema recuperará el estado de trabajo *Regulation* y se mostrarán en el panel los parámetros del modelo resultante.

3.2.2. Sintonía del regulador PID

3.2.2.1. Cálculo de los parámetros de sintonía del PID:

La propuesta de sintonía de Skogestad para modelo de primer orden se basa en la elección del valor del parámetro τ_c (Ecuación (45)). En este método en particular, sólo es posible configurar un controlador de tipo PI.

$$K_c = \frac{1}{k}\frac{\tau_1}{\tau_c + \theta} \qquad (45)$$

$$\tau_I = \min\{\tau_1, 4(\tau_c + \theta)$$

La teoría indica que este parámetro se mueve dentro del rango $-\theta < \tau_c < \infty$, generando respuestas agresivas cuando se acerca a $-\theta$ y más robustas cuanto tiende ∞, pero en la realidad acercarse demasiado a los extremos

provoca comportamientos indeseados, por llegar a la inestabilidad o por ser demasiado lentos.

Por esta razón, en la aplicación se ha acotado este rango al intervalo $-0.2 \cdot \theta < \tau_c < 4 \cdot \theta$. Este parámetro de diseño se controla mediante una barra deslizadora y, al mismo tiempo, se muestran los parámetros PI que se corresponden a ese valor de τ_c.

Para facilitar la selección de unos parámetros adecuados a la respuesta deseada, se ha añadido también una gráfica en la que se representa la salida del sistema prevista según la función de transferencia identificada y los parámetros PI del regulador.

La visualización de la respuesta también se actualiza de forma automática al modificar el parámetro τ_c con la barra deslizadora.

3.3. Método del relé

El método del relé se basa, a grandes rasgos, en la búsqueda de un comportamiento oscilatorio mediante la sustitución del regulador por una conmutación sucesiva de la variable manipulada entre dos valores.

Los pasos que se deben completar para la ejecución del método son los siguientes:

1. *Detección de estado estacionario:* es importante para la ejecución de este método que el sistema esté funcionando de forma estable antes de comenzar, o dicho de otra forma, que haya alcanzado el estacionario.

2. *Zona de histéresis:* se trata de un bucle en el que se analiza continuamente el valor del error para, en base a él, determinar el valor que deberá tomar la salida del controlador. Estas variaciones serán las causantes del comportamiento oscilatorio del sistema.

3. *Obtención de las características del sistema:* se identifican sobre la respuesta generada los parámetros que identifican el punto de la respuesta en frecuencia.

4. *Cálculo de los parámetros de sintonía del regulador:* aplicando las reglas de Skogestad se obtienen los nuevos valores de configuración del PID.

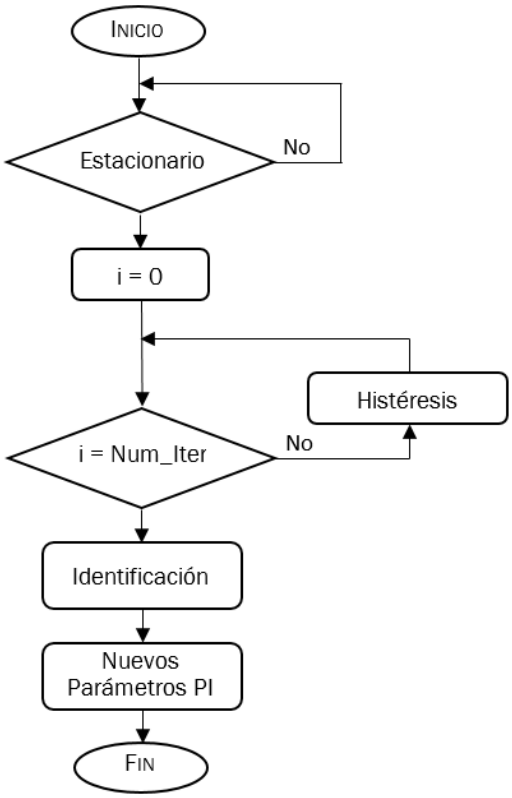


Figura 46 – Diagrama del flujo del método del relé

El método del relé de Åström es, por definición, un algoritmo de sintonía en lazo cerrado aunque, a efectos prácticos, el regulador PID se desconecta de la planta. Por esta razón, el estado de trabajo que se debe activar para su planteamiento debe ser *Output Positioning* y no *Regulation*. De esta manera será posible diseñar un nuevo regulador (el relé con histéresis) en un *Control Script* que tenga el control total de la planta y que gobierne su funcionamiento durante el proceso de sintonía.

A continuación, se detallará cada uno de los pasos y se analizará su implementación.

3.3.1. Detección de estado estacionario

Es imprescindible para asegurar la validez del método cerciorarse de que se parte de un estado de estabilidad, puesto que si no fuera así no se podría asegurar que la respuesta que muestra el sistema se debe únicamente a las variaciones que se están introduciendo de forma voluntaria.

Por ello se ha recurrido de nuevo al método estadístico de *R. Russell Rhinehart* utilizado en ocasiones anteriores para este fin.

3.3.2. Zona de histéresis

La histéresis fue definida en el Capítulo III como una sucesión de cambios simétricos en la variable manipulada del sistema que se movían entre los valores u0+d y u0-d (Figura 30 a)) cuando el error alcanza unos valores ±ε. Realmente, como se plantea en la Figura 30 b), no es necesaria esta simetría en los cambios en la variable manipulada. Además, es más interesante de cara al usuario final de la aplicación que los parámetros entre los que va a oscilar u(t) se introduzcan como valores absolutos y no incrementales. De esta manera, el usuario será plenamente consciente de los valores mínimo y máximo que define.

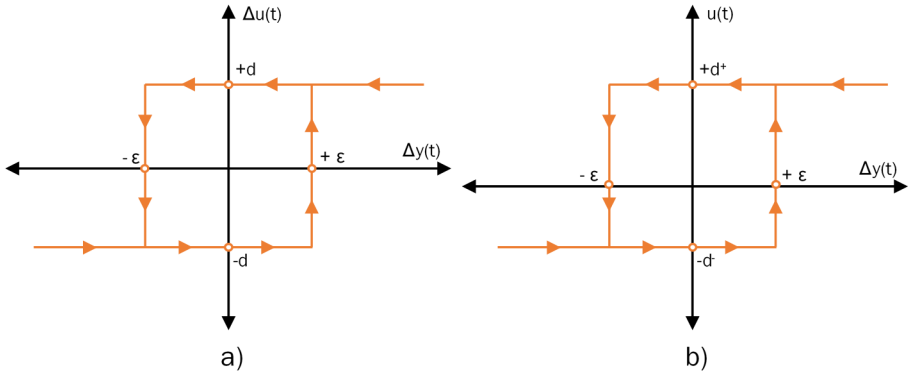


Figura 47 – Histéresis simétrica (a) y desigual (b)

Se ha planteado una alternativa de diseño que para la programación de la histéresis que se descompone en los siguientes pasos:

- Se calcula el error como la diferencia entre la salida del sistema en el instante actual y en el instante inicial, es decir, el valor de la salida al final de la verificación de estado estacionario.
- Si el error es mayor que el límite superior, la salida del regulador conmuta a d^-. En este instante en que se produce el cambio se considera que comienza una nueva iteración.
- Si el error es menor que el límite inferior, la salida del regulador conmuta a d^+.
- Mientras el error esté dentro del intervalo [-ε, +ε] la salida del regulador se mantiene constante.

Para cumplir con la premisa de simplicidad para el usuario de la que se habló al inicio, sólo el parámetro *d* debe ser configurado por él. El otro parámetro ε se genera automáticamente a partir del paso 1 (*Detección de estado estacionario*).

Analizando los datos recogidos al final esa primera fase, cuando ya está prácticamente verificada la condición de estabilidad, se procede a medir el ruido del sistema. Una vez determinado el ruido de la planta, se utiliza la consideración de tomar un ε cuyo valor sea el triple del ruido medido, para asegurar que la histéresis generada tiene amplitud suficiente como para soportar el ruido del sistema.

El diagrama de flujo de la Figura 48, que define de manera gráfica el proceso descrito, podría estar contenido en el bloque denominado *Histéresis* del diagrama de flujo general de la Figura 46.

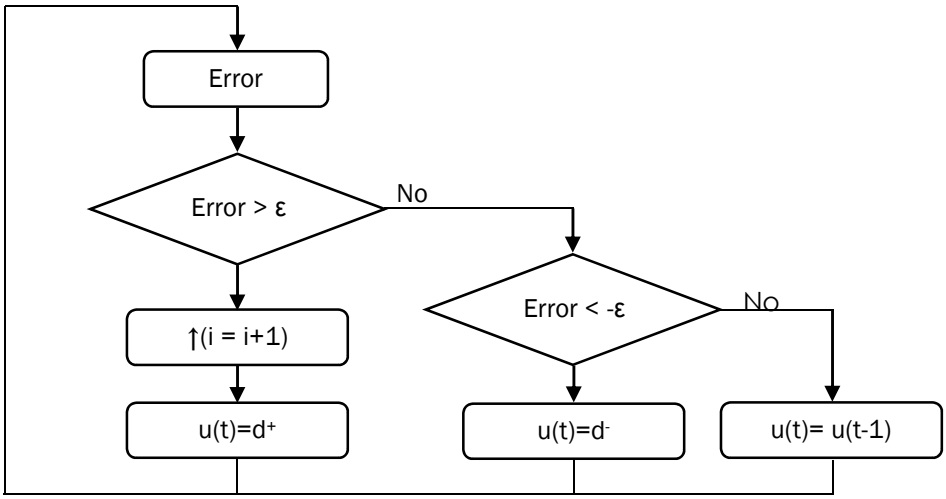


Figura 48 – Diagrama de flujo del ciclo de histéresis

Un aspecto importante a tener en cuenta para este criterio de sintonía es si la ganancia del sistema es de acción inversa (k>0) o directa (k<0). La relevancia de este dato se halla en el hecho de que:

- Si la ***ganancia del sistema*** es ***positiva***, cuando el error sea positivo, la salida del regulador deberá conmutar a d^- y cuando el error sea negativo, lo hará a d^+.
- Si, por el contrario, la ***ganancia del sistema*** es ***negativa***, la salida del regulador deberá actuar al contrario, de tal modo que, cuando el error sea negativo la salida conmutará a d^- y cuando el error sea positivo, lo hará a d^+.

Se terminará con el bucle de oscilaciones mantenidas cuando se considere que estas son estables, es decir, que se repiten en el tiempo. El usuario debe prever al inicio el número de ciclos que serán necesarios para alcanzar el estado de oscilaciones mantenidas que se busca. Normalmente, con 3 iteraciones es suficiente.

3.3.3. Obtención de las características del sistema

Como se explicó en el *Capítulo III*, el método del relé consigue la identificación de un punto concreto de la curva de Nyquist. Los parámetros que modelan ese punto son la ganancia crítica y el periodo crítico del sistema, y se obtienen del siguiente modo.

Se puede ver en la Figura 49 que los parámetros a y Pr deben ser obtenidos del sistema para la determinación de los valores del PID, por lo que se necesita un ciclo de histéresis más para obtenerlo.

$$K_{Cr} = \frac{2 \cdot (d^{+} - d^{-})}{\pi\sqrt{(a^2 - \varepsilon^2)}}$$

Tipo	Ganancia K_p	Tiempo integral	Tiempo derivativo
P	0.5 K_c		
PI	0.45 K_c	T/1.2	
PID paralelo	0.75 K_c	T/1.6	T/10
PID serie	0.6 K_c	T/2	T/8

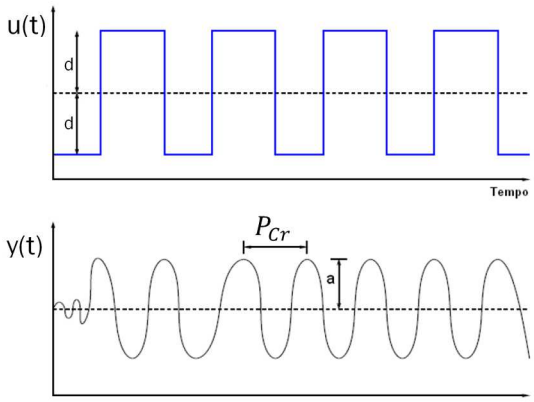


Figura 49 – Parámetros de respuesta en frecuencia y tabla de sintonía

El periodo se consigue grabando el instante de tiempo en el que alcanza un valor máximo, el instante en que alcanza el máximo siguiente y calculando la diferencia entre ambos.

De forma similar, el parámetro *a* es la mitad de la diferencia calculada entre un máximo y un mínimo en la variable de salida del sistema.

3.4. Cálculo de parámetros de sintonía del regulador

Una vez conocidos los parámetros K_{cr} y P_{cr} obtener los parámetros de sintonía del regulador se reduce simplemente a la realización de los cálculos indicados en la tabla de sintonía de Ziegler Nichols para lazo cerrado.

A diferencia del S-IMC, el método del relé ofrece la ventaja de poder elegir entre una configuración PI o PID.

3.5. Método IFT

El IFT es un algoritmo que, de manera iterativa, perturba el sistema para analizar su respuesta y utilizarla para acercar el sistema al comportamiento deseado hasta alcanzar el funcionamiento óptimo.

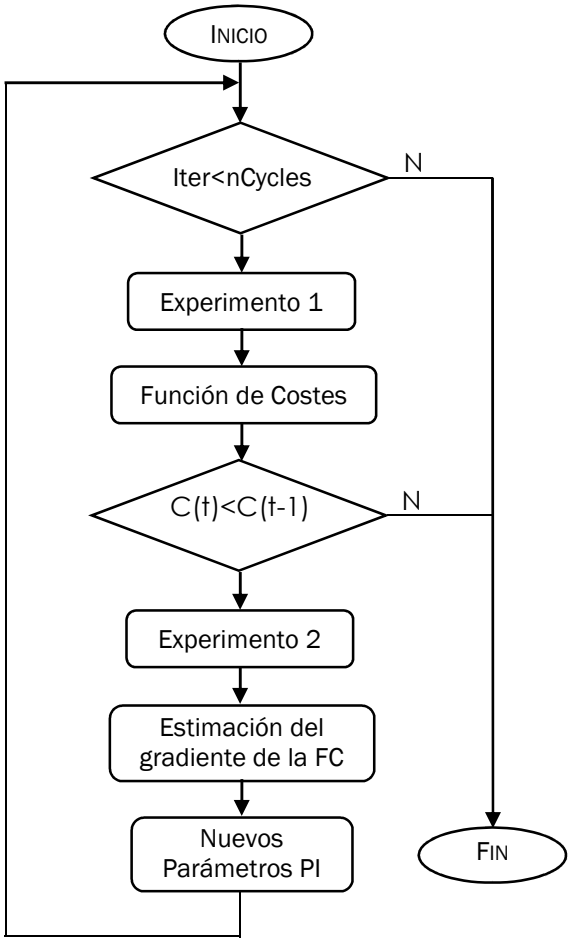


Figura 50 – Diagrama de flujo del método IFT

En este caso se está tratando un proceso cíclico, en el que cada iteración se puede descomponer en los siguientes pasos:

1. ***Primer experimento:*** se aplica una entrada escalón a la referencia del sistema.

2. ***Cálculo del coste:*** se cuantifica el error existente y el esfuerzo de control que supone. Si se detecta que el coste está aumentando en vez de disminuir, se detiene el método y se recuperan los parámetros de la iteración anterior.

3. ***Segundo experimento:*** se aplica una entrada escalón a la referencia del sistema en cada ciclo de muestreo de valor $SP_{Exp2}(i) = u_{Exp1}(i) - y_{Exp1}(i)$

4. ***Estimación del gradiente de la función de costes:*** con los resultados del experimento 2 se puede realizar la estimación indicada.

5. ***Cálculo de los nuevos parámetros PID:*** se recalculan los parámetros para el siguiente ciclo de ejecución del método.

La Figura 50 representa de forma esquemática el procedimiento descrito.

Se trata de un procedimiento que durante toda su ejecución trabaja en lazo cerrado, de modo que el estado de trabajo será *Regulation*. De este modo, el PID funciona normalmente durante toda una iteración hasta que, en la última etapa, se actualizan los parámetros del controlador por los que se acaban de calcular.

3.5.1. Experimento 1

La finalidad de este paso es obtener la respuesta del sistema para después, poder cuantificar de forma analítica la bondad del controlador tratado. El experimento consiste en aplicar un incremento a la referencia del sistema y deberá tener una duración suficiente para permitir que se alcance un nuevo estado estacionario.

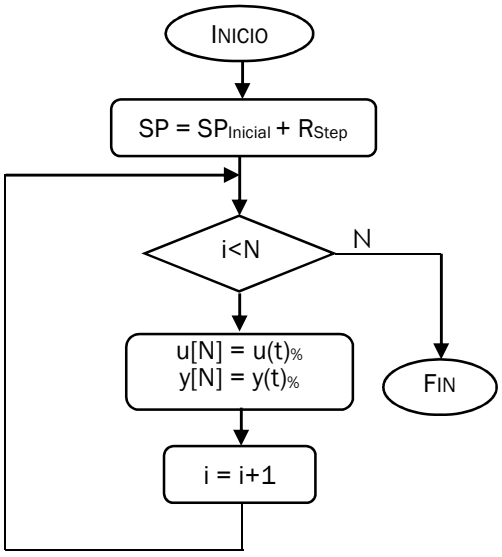


Figura 51 – Diagrama de flujo correspondiente al Experimento 1

La consigna de un sistema puede estar determinada por el usuario o puede venir dada por la salida de otro sistema, como puede ser el caso de una estructura en cascada. Según la estructura de UNICOS, la referencia puede estar definida por el propio controlador, si la controla directamente el usuario, o por un objeto de tipo *Analog* si viene dada por otra parte de la planta.

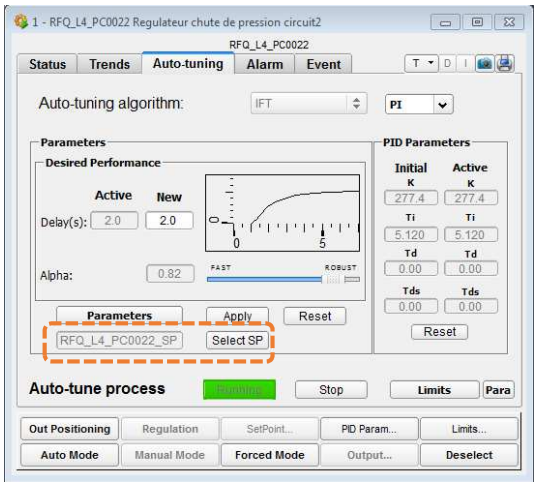


Figura 52 – Selección de la variable de referencia

Si la variable del *set-point* del controlador se encuentra inhabilitada, se hacen visibles las casillas que se han remarcado en la Figura 52. Pulsando el botón *Select SP* se abre una ventana con todas las variables de tipo *Analog* disponibles en el sistema y será el usuario el encargado de seleccionar aquella que fija la referencia del lazo.

Los valores muestreados de las variables de control y de salida del sistema se almacena en sendos vectores (y[N], u[N]). El tamaño de los vectores está fijado por programa (u[40], y[40]) de tal modo que, la manera de asegurar que con ese longitud se abarcará el tiempo que dura un experimento es mediante la elección de un tiempo de muestreo adecuado.

Indistintamente de que se haya aplicado el escalado [0, 100]% a las variables del sistema, su valor se normaliza para expresarlo en tanto por cien antes de ser almacenado. Con esto se consigue un funcionamiento más homogéneo para cualquier tipo de aplicación, independiente del rango de estas variables del sistema.

Para realizar esta conversión se han recogido inicialmente cinco datos de cada una de las variables y se ha calculado su media para obtener un valor estacionario inicial fiable. Después sólo resta aplicar a cada dato la Ecuación (46).

$$y[N] = \frac{y[N] - ySteady_{In}}{ySteady_{In}} \cdot 100$$
$$u[N] = \frac{u[N] - uSteady_{In}}{uSteady_{In}} \cdot 100 \quad (46)$$

La comprobación de que los límites del proceso de autosintonía no se sobrepasan, se realiza para cada muestra recogida, tanto para la variable de control como para la salida del sistema. Si los límites se superan inmediatamente se desactiva la habilitación del método, en cuyo caso se sale de la rutina del experimento, y se recupera el valor de referencia inicial del sistema.

Al salir del experimento, se verifica que se hayan recogido los N datos. Si no es así, como puede suceder en el caso de que se hayan excedido de los límites durante el proceso, se activa una variable que indica que la ejecución de los experimentos no ha sido válida.

Si se produce un error en este paso, indistintamente del número de iteración en el que se encuentre el proceso, se recuperan los valores iniciales del controlador PID con el fin de devolver al sistema a un estado de seguridad.

3.5.2. Cálculo de la función de costes

El coste que supone el experimento 1 se cuantifica con la (47), donde el error se calcula como la diferencia entre la salida real del sistema y la salida que se desea.

$$J(\rho) = \int_{t=0}^{N} e(t)^2 \cdot dt + \lambda \int_{t=0}^{N} u(t)^2 \cdot dt \quad (47)$$

$$e(t) = y(t) - y_d(t) \quad (48)$$

$$y_d = \frac{1-\alpha}{1-\alpha \cdot z}\, r \quad (49)$$

Los parámetros que el usuario debe configurar para diseñar la respuesta final que busca son el λ del esfuerzo de control y el parámetro α de la respuesta deseada.

La variable λ es el peso que se le da a la acción de control o, dicho de otra manera, indica al algoritmo si queremos que tome en cuenta si se requiere una acción de control grande para obtener la respuesta deseada. Su rango de actuación comprende valores dentro del intervalo [0, 1].

Aunque este parámetro no es necesario como tal para diseñar la respuesta que se busca del sistema, siempre es bueno considerarlo para que el regulador no llegue a sintonizarse de manera que genere acciones de control demasiado agresivas. Como se verá en el Capítulo V en la validación de los métodos, se puede conseguir la misma respuesta de un sistema con diferentes configuraciones del regulador PID, siendo unas más robustas que otras.

La respuesta deseada es, matemáticamente, el resultado de aplicar un filtro discreto de primer orden a la consigna del sistema, que para el experimento 1 incluye una entrada escalón. El parámetro de diseño es la α de la Ecuación (49) del filtro.

Hay que tener en cuenta que estos dos parámetros están interrelacionados. Se puede requerir una salida rápida al sistema, pero si se le da una ponderación muy alta a la acción de control, el peso que ésta última tendrá en la función de costes superará al del error y la respuesta final no podrá ser la prevista.

Finalmente, en función de la configuración seleccionada y aplicando la Ecuación (47) se calcula en cada iteración el coste que se le asocia. Lo que se trata en este método es de que en cada iteración el coste se vea reducido hasta alcanzar el mínimo o hasta finalizar el número de iteraciones. Por esta razón se tratará de evitar que cuando el sistema localice el mínimo pueda continuar su tendencia y alejarse de él. Si se detecta, en alguna iteración, que el coste actual ha aumentado respecto del coste de la iteración anterior, el sistema descarta los valores actuales del PID y recupera aquellos con los que obtuvo mejores resultados, es decir, los de la iteración anterior a la actual. Además, termina con el proceso de sintonía.

3.5.3. Experimento 2

El experimento 2 tiene la finalidad de obtener los valores de la salida del sistema y la del controlador para una entrada que varía en cada iteración con la forma:

$$\mathrm{SP[N]} = \mathrm{SP_{Inicial}} + \mathrm{R_{Step}} - \mathrm{y[N]} \qquad (50)$$

La verificación de que los límites no se sobrepasan es similar a la del experimento 1, con el añadido de que, para este caso, también se comprueba en cada iteración que el valor de referencia que se va a solicitar está dentro de los límites.

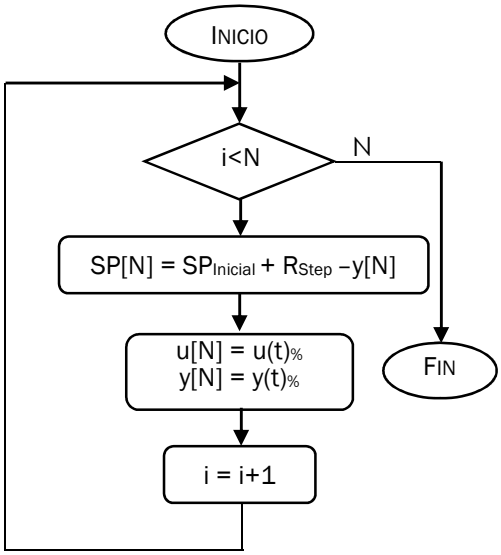


Figura 53 – Diagrama de flujo correspondiente al Experimento 2

3.5.4. Estimación del gradiente de la función de costes

El cálculo de esta estimación, Ecuación (51), se realiza a partir de la estimación de los gradientes de las variables de control y la salida del sistema, Ecuaciones (52) y (53). Estas, a su vez, se consiguen a partir de los resultados que se acaban de obtener del experimento 2.

$$est\left[\frac{\partial J(\rho)}{\partial \rho}\right] = \frac{1}{N}\left[\sum_{t=1}^{N} \tilde{y}_t(\rho) \cdot est\left(\frac{\partial y_t(\rho)}{\partial \rho}\right) + \lambda \sum_{t=1}^{N} u_t(\rho) \cdot est\left(\frac{\partial u_t(\rho)}{\partial \rho}\right)\right] \quad (51)$$

$$est\ \frac{\partial y}{\partial \rho} = \frac{1}{C}\left[\frac{\partial C}{\partial \rho} y_2\right] \quad (52)$$

$$est\ \frac{\partial u}{\partial \rho} = \frac{1}{C}\left[\frac{\partial C}{\partial \rho} u_2\right] \quad (53)$$

El cálculo indicado en las Ecuaciones (52) y (53) no puede ser realizado elemento a elemento, obteniendo el valor de la multiplicación del valor $(\mathrm{C}^{-1}\ \partial \mathrm{C}/\partial\rho)y[i]$ porque, en la realidad, cada componente $\partial y[i]/\partial\rho$ no sólo dependerá de si mismo, sino también de los valores que le preceden.

Matemáticamente, esta operación se corresponde con un filtro discreto.

Un filtro digital se caracteriza por su función de transferencia. La función de transferencia para un filtro discreto lineal, invariante en el tiempo expresada en el dominio Z tiene la siguiente forma:

$$H(z) = \frac{B(z)}{A(z)} = \frac{b_0 + b_1 z^{-1} + b_2 z^{-2} + \cdots + b_N z^{-N}}{1 + a_1 z^{-1} + a_2 z^{-2} + \cdots + a_M z^{-M}} \tag{54}$$

La respuesta impulso es una caracterización del comportamiento del filtro. Los filtros digitales se agrupan en dos categorías: de respuesta de impulso infinito (IIR) y respuesta de impulso finito (FIR). En el caso de filtros lineales invariantes en el tiempo FIR, la respuesta de impulso es exactamente igual a la secuencia de coeficientes de filtro:

$$y_n = \sum_{k=0}^{n-1} h_k x_{n-k} \tag{55}$$

Filtros IIR son recursivos. Cada valor de la salida se obtiene en función de las entradas actuales y anteriores, así como las salidas anteriores. La forma general de un filtro IIR es la siguiente:

$$\sum_{m=0}^{M-1} a_m y_{n-m} = \sum_{k=0}^{n-1} b_k x_{n-k} \tag{56}$$

Finalmente, la ecuación en diferencias resultante es:

$$y[n] = -\sum_{k=1}^{M} a_k y[n-k] + \sum_{k=0}^{N} b_k x[n-k] \tag{57}$$

Esta última expresión (57) será la que se implemente para realizar los cálculos.

Para poder aplicar la expresión que se ha deducido para los filtros, es necesario obtener los coeficientes de la función de transferencia que lo define.

Partiendo de la Ecuación (58)que define el funcionamiento del regulador continuo:

$$PID = K_p\left(1 + \frac{1}{T_i s} + sT_d\right) \tag{58}$$

Las derivadas parciales respecto a cada uno de los parámetros son:

$$\frac{\partial PID}{\partial K_p} = 1 + \frac{1}{T_i \cdot s} + T_d \cdot s \tag{59}$$

$$\frac{\partial PID}{\partial T_i} = -\frac{K_p}{T_i^2 \cdot s} \tag{60}$$

$$\frac{\partial PID}{\partial T_d} = K_p \cdot s \tag{61}$$

Dividiendo las expresiones (59), (60) y (61) entre (58) obtenemos las ecuaciones que se corresponden con $(\mathrm{C}^{-1}\ \partial \mathrm{C}/\partial\rho)$ para los filtros:

$$K_p Filter(s) = \frac{1}{K_p} \tag{62}$$

$$T_i Filter(s) = -\frac{1}{T_i \cdot (T_d \cdot T_i * s^2 + T_i \cdot s + 1)} \tag{63}$$

$$T_d Filter(s) = \frac{T_i \cdot s^2}{T_d \cdot T_i \cdot s^2 + T_i \cdot s + 1} \tag{64}$$

Hasta el momento, se ha mantenido el sistema en el dominio para sistemas continuos de Laplace. Aplicando la aproximación de Tustin se puede obtener su modelo discreto:

$$s = -\frac{2}{T}\frac{\left(\frac{1}{z} - 1\right)}{\left(\frac{1}{z} + 1\right)} \qquad \rightarrow \qquad T: Periodo\ de\ muestreo \tag{65}$$

Sustituyendo, se consiguen las siguientes expresiones:

$$K_p Filter(z) = \frac{1}{\mathrm{K}} \tag{66}$$

$$T_i Filter(z) = -\frac{1}{\mathrm{T_i}\left(\frac{4\mathrm{T_d}\mathrm{T_i}(z^{-1}-1)^2}{\mathrm{T}^2(z^{-1}+1)^2} - \frac{2\mathrm{T_i}(z^{-1}-1)}{\mathrm{T}(z^{-1}+1)} + 1\right)} \tag{67}$$

$$T_d Filter(z) = \frac{4\mathrm{T_i}(z^{-1}-1)^2}{\mathrm{T}^2(z^{-1}+1)^2\left(\frac{4\mathrm{T_d}\mathrm{T_i}(z^{-1}-1)^2}{\mathrm{T}^2(z^{-1}+1)^2} - \frac{2\mathrm{T_i}(z^{-1}-1)}{\mathrm{T}(z^{-1}+1)} + 1\right)} \tag{68}$$

Simplificando:

$$F\_K_p = \frac{1}{\mathrm{K}} \tag{69}$$

$$F\_T_i = \frac{-\mathrm{T}^2\mathrm{z}^2 - 2\cdot\mathrm{T}^2\mathrm{z} - \mathrm{T}^2}{(\mathrm{T}^2\mathrm{T_i} + 2\mathrm{T}\mathrm{T_i}^2 + 4\mathrm{T_d}\mathrm{T_i}^2)\mathrm{z}^2 + (2\mathrm{T}^2\mathrm{T_i} - 8\mathrm{T_d}\mathrm{T_i}^2)\mathrm{z} + \mathrm{T}^2\mathrm{T_i} - 2\mathrm{T}\mathrm{T_i}^2 + 4\mathrm{T_d}\mathrm{T_i}^2} \tag{70}$$

$$F\_T_d = \frac{4\mathrm{T_i}\mathrm{z}^2 - 8\mathrm{T_i}\mathrm{z} + 4\mathrm{T_i}}{(\mathrm{T}^2 + 2\mathrm{T_i}\mathrm{T} + 4\mathrm{T_d}\mathrm{T_i})\mathrm{z}^2 + (2\mathrm{T}^2 - 8\mathrm{T_d}\mathrm{T_i})\mathrm{z} + \mathrm{T}^2 - 2\mathrm{T_i}\mathrm{T} + 4\mathrm{T_d}\mathrm{T_i}} \tag{71}$$

Ahora que ya está disponible la función de los filtros discretos y las funciones de transferencia con expresadas en base a los parámetros del controlador, sólo resta sustituir en la función del filtro con los valores actuales de configuración del PID para obtener los valores numéricos de los coeficientes y calcular las estimaciones de las derivadas parciales de las variables de control y de salida del sistema.

$$\frac{\partial y}{\partial K_p} = Filter(Num\_F\_K_p, Den\_F\_K_p, y_{Exp2})$$

$$\frac{\partial y}{\partial T_i} = Filter(Num\_F\_T_i, Den\_F\_T_i, y_{Exp2})$$

$$\frac{\partial y}{\partial T_d} = Filter(Num\_F\_T_d, Den\_F\_T_d, y_{Exp2})$$

3.5.5. Cálculo de los nuevos parámetros PID

Los nuevos valores para el regulador PID se obtienen a partir de la Ecuación (72).

$$\rho_{i+1} = \rho_i - \gamma_i \cdot H_i^{-1} \cdot \frac{\partial J(\rho_i)}{\partial \rho} \quad (72)$$

Donde:

- ρ_i: son los parámetros PID actuales
- γ: Es la velocidad con la que el algoritmo converge hacia el punto óptimo. Inicialmente vale 1 y se reduce a la mitad con cada iteración.
- H_i: es la matriz Hessiana de la función de costes, Ecuación (73).

$$H = est\left[\frac{\partial^2 J(\rho)}{\partial \rho^2}\right] = \frac{1}{N}\left[\sum_{t=1}^{N} est\left(\frac{\partial y_t(\rho)}{\partial \rho}\right)^2 + \lambda \sum_{t=1}^{N} est\left(\frac{\partial u_t(\rho)}{\partial \rho}\right)^2\right] \quad (73)$$

- El determinante de la matriz Hessiana debe ser no nulo para poder realizar el cálculo de la inversa. En el caso de que lo sea, se tomará la matriz identidad.

Una vez calculado la inversa del Hessiano, ya se tienen todos los datos necesarios para obtener unos nuevos parámetros para el controlador para la nueva iteración.

Para ser consecuentes con la filosofía de seguridad y robustez que se ha propuesto, los nuevos parámetros calculados se someten a una pequeña validación antes de ser aplicados.

Los nuevos parámetros PID que se calculan deben siempre positivos. Incluso la ganancia proporcional, que podría tener signo, es siempre mayor que cero, debido a que lo que indica el signo es el bit "*Direct/Reverse Action*".

Se ha llegado a la conclusión de que, si se alcanzan parámetros negativos, es porque la velocidad de convergencia es demasiado alta y el algoritmo sobrepasa el punto óptimo.

Por esta razón, si se detectan valores negativos, el factor de velocidad *Gamma* se reduce a la mitad y se vuelve a realizar el cálculo de nuevo. Si continuara siendo negativo, se repetirá, hasta un máximo de 10 veces, el proceso de disminuir la velocidad de convergencia y recalcular hasta que se

obtengan parámetros positivos. Si transcurridos estos 10 intentos algún dato continúa siendo negativo, se procede a recuperar los parámetros de la iteración anterior y se termina con la ejecución del método.

4. Diseño de la interfaz de la herramienta

La aplicación de autosintonía, como se ha mencionado en ocasiones anteriores, va implementada como una parte del objeto *Controlador*, incluida en una nueva pestaña en el panel del scada.

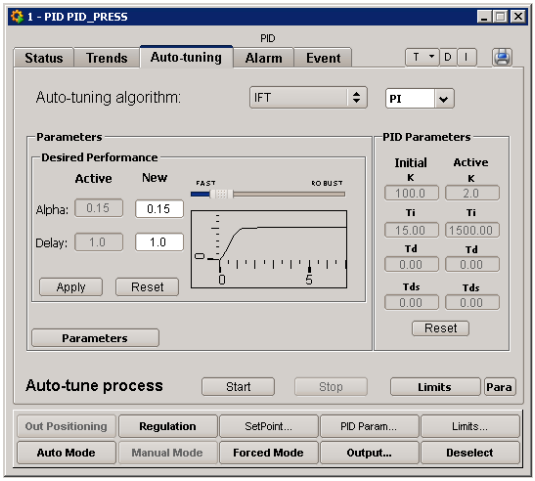


Figura 54 – Panel principal de la herramienta

La ventana tiene, a grandes rasgos, dos partes diferenciadas. Por un lado, en el rectángulo interior, están los métodos que se han ido tratando y, alrededor están los parámetros de configuración que van a ser (más o menos) generales para todos ellos. La Figura 54 muestra la apariencia inicial de la aplicación al ser abierta.

4.1. Parámetros generales de configuración

Los parámetros generales son aquellos que deberían ser definidos antes de comenzar con ningún método de sintonía. A continuación se comentará cada uno de ellos.

4.1.1. Herramienta de autosintonía

Abriendo este menú desplegable se puede seleccionar el panel del método que va a hacer visible en la zona central.

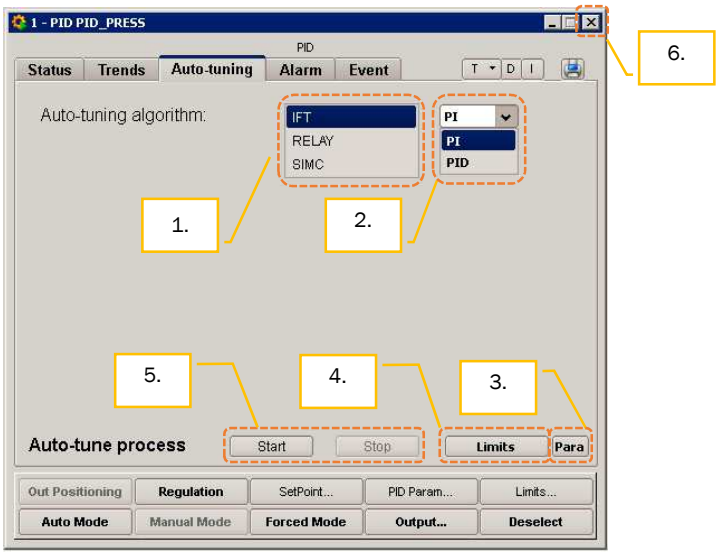


Figura 55 – Características comunes a todos los métodos

4.1.1.1. Algoritmo de autosintonía

Abriendo este menú desplegable se puede seleccionar el panel del método que va a hacer visible en la zona central.

4.1.1.2. Estructura del controlador

Este menú permite seleccionar la estructura que va a tener el controlador que generemos con el algoritmo de autosintonía. Por esta razón, sólo será visible para los casos en los que el método de sintonía ofrezca la posibilidad de diseñar reguladores con distinto número de parámetros, como son el IFT y el relé.

4.1.1.3. Parámetros auxiliares

El botón *Para* es un elemento auxiliar que no existirá en la versión final del programa, para la próxima actualización de UNICOS. El panel que despliega, Figura 56, contiene parámetros del sistema que, a nivel de PLC son conocidos, pero no a nivel de Scada.

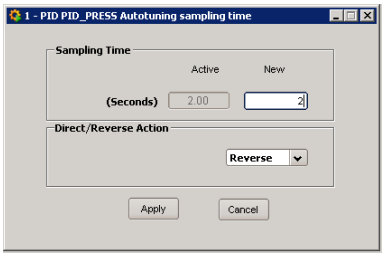


Figura 56 – Panel de parámetros auxiliares

El tiempo de muestreo indica a los métodos cada cuanto tiempo debe hacerse a recogida de datos. Es importante que este valor sea mayor o igual que 2 segundos, al menos si se quiere utilizar el IFT correctamente. El sistema incluye una restricción que introduce un retardo de 2 segundos en el proceso cuando se varía la referencia del sistema, de modo que es el tiempo mínimo entre muestras para el experimento 2 del IFT.

Por otra parte, el menú *Direct/Reverse Action* indica si la ganancia del sistema es menor o mayor que cero. En principio, es un dato que sólo afecta el método del relé.

4.1.1.4. Límites

El valor de los límites, por defecto, se ha configurado para ser el de los límites del proceso. Siempre que se quiera modificar su valor, tendrán que estar dentro de los márgenes que permite el proceso. Si no es así, al pulsar *Apply* para hacerlos activos, se borrarán y aparecerán los valores antiguos.

Pulsando el botón *Restore* se pueden recuperar los parámetros por defecto.

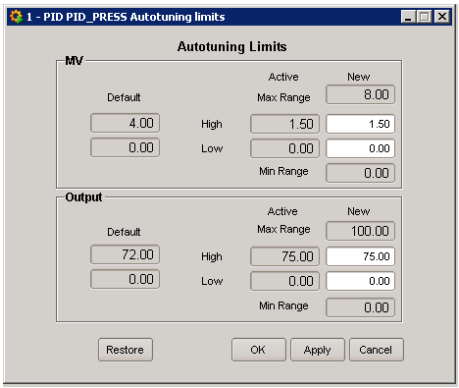


Figura 57 – Panel de configuración de límites para la autosintonía

4.1.1.5. Start y Stop

Son los encargados de gestionar el arranque y la parada de los métodos, por lo que nunca están activos al mismo tiempo. Si no hay ningún método en proceso estará disponible el botón *Start*. En caso contrario, si algún método se está ejecutando, el botón *Start* estará desactivado y parpadeando en verde con el texto *Running* y el botón *Stop* estará activo.

Si se pulsa *Stop* durante la ejecución de cualquier método se producirá una parada inmediata. Si el método hubiera realizado alguna modificación en los parámetros del controlador durante su ejecución: estado de trabajo, variable de control, referencia, configuración PID,... se recuperarán los valores previos al proceso de sintonía.

4.1.1.6. Cerrar panel

Si se trata de cerrar el panel mientras se está ejecutando algún algoritmo de sintonía, aparece la ventana emergente que se muestra en la Figura 58.

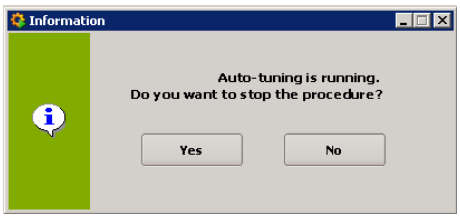


Figura 58 –Confirmación de cierre durante la ejecución

El panel permite detener la ejecución del método, si realmente se quiere abandonar el sistema, o volver al panel del controlador y continuar. Si se decide terminar con la autosintonía, se ejecutará la misma rutina de parada que si se pulsara el botón *Stop* y, después, se cerrará el panel.

4.2. Método S-IMC

La Figura 59 muestra la ventana principal de configuración del S-IMC. La zona de la izquierda, *Identified System*, muestra los parámetros que definen el modelo del sistema. Se plantean dos formas de configurar esta sección:

- ***Editar** (Edit):* si se selecciona esta casilla, los visualizadores de los parámetros del modelo pasan a ver editables y se pueden introducir nuevos valores.

  Los tres valores deben ser siempre positivos. El retardo (*Delay*) y la constante de tiempo del sistema (*Time*) se expresan en segundos. Además, el retardo por definición debe ser mayor o igual que un periodo de muestreo.

- ***Identificar** (Identify):* pulsando este botón se accede al panel de identificación, que se explicará más adelante.

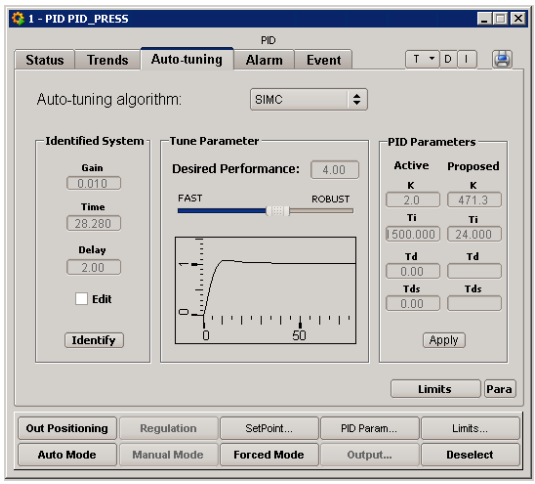


Figura 59 – Panel del método S-IMC

En la zona central, *Tune Parameter,* se encuentra el parámetro de configuración del S-IMC, definido en la teoría como τ_c. Se configura mediante la barra deslizadora y su valor se muestra en el visualizador que hay sobre ella. Es conveniente recordar que la literatura recomienda como valor de configuración $\tau_c=\theta$, que en este caso sería $\tau_c=2s$. De cualquier modo, para facilitar la labor del usuario, en la gráfica inferior se muestra la respuesta del sistema para el parámetro de configuración seleccionado.

A la derecha, *PID Parameters*, se pueden ver, en primer lugar, los parámetros que actualmente tiene activos el controlador y, a continuación, los parámetros que el método está proponiendo en base a la configuración seleccionada.

Cuando los parámetros PID propuestos y la respuesta estimada se consideren adecuados, pulsando el botón *Apply* los nuevos valores PID se cargarán y pasarán a ser los nuevos parámetros activos del controlador.

Puede notarse que en este panel no se muestran los botones de *Start* y *Stop*. Esto es debido a que este método no requiere de la ejecución de ningún experimento. Sólo realiza cálculos matemáticos que se ejecutan automáticamente con el movimiento de la barra deslizadora y se hacen efectivos al pulsar el botón *Apply*.

Retomando el caso de la identificación, al pulsar el botón *Identify* se despliega el panel de la Figura 60.

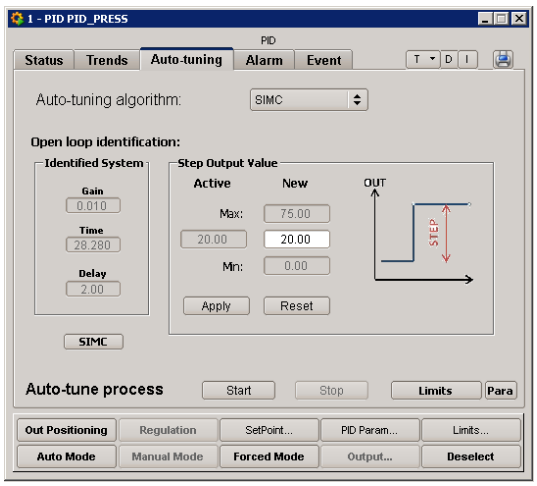


Figura 60 – Panel de la identificación de sistema FOPD

La sección *Identified System* en este caso sólo sirve para visualizar los parámetros del sistema que están almacenados, antes o después de la identificación, aunque ahora no es editable, como sucedía anteriormente.

La zona central, *Step Output Value*, contiene los parámetros de configuración para el experimento de identificación. El único dato que se requiere es la amplitud del escalón que se va a aplicar a la entrada del sistema (la salida del controlador). Como se muestra en la imagen del panel, este valor debe introducirse como incremento respecto del valor de salida actual.

El valor que se introduzca debe estar comprendido entre los límites, mínimo y máximo, de la autosintonía. Si no lo está, al tratar de cargarlo pulsando *Apply* se borrará el valor escrito y aparecerá de nuevo el dato activo. También se puede reemplazar el dato escrito por el activo pulsando *Reset*.

Tras asegurar que el valor del escalón que se quiere aplicar está activo viendo que aparece en la casilla *Active*, se puede pulsar *Start* para dar comienzo al experimento.

Cuando el experimento finaliza, los valores del sistema identificado se actualizan y pasan a mostrarse los que se acaban de calcular. Si se pulsa *Stop* antes de que termine el proceso, continuarán manteniéndose los datos iniciales.

Para volver al panel inicial del S-IMC se deberá pulsar el botón *S-IMC*.

4.3. Método del relé

Cuando se selecciona en el menú desplegable el método del relé se despliega el panel mostrado en la Figura 61.

La sección de la izquierda contiene los parámetros de configuración del método, que en este caso son tres:

- ***d_{Max}:*** indica el valor absoluto (no incremental) máximo que alcanzará la señal de salida del controlador.

- ***d_{Min}:*** indica el valor absoluto (no incremental) mínimo que alcanzará la señal de salida del controlador.

  Las variables d_{Min} y d_{Max} deben estar dentro de los límites fijados para la autosintonía. Si los exceden, al tratar de validarlos pulsando el botón *Apply* serán reemplazados automáticamente por los valores que estén actualmente activos.

- ***Cycles:*** es el número de oscilaciones completas que se van a realizar, incluyendo la última en la que se recogen los datos necesarios.

  El número de iteraciones necesarias para que el proceso genere oscilaciones mantenidas no es siempre el mismo, aunque se puede decir que, normalmente, con cuatro iteraciones se suele conseguir.

Igual que en otras ocasiones, si se desea borrar los datos escritos, pulsando *Reset* éstos son reemplazados por los valores que están activos para el método.

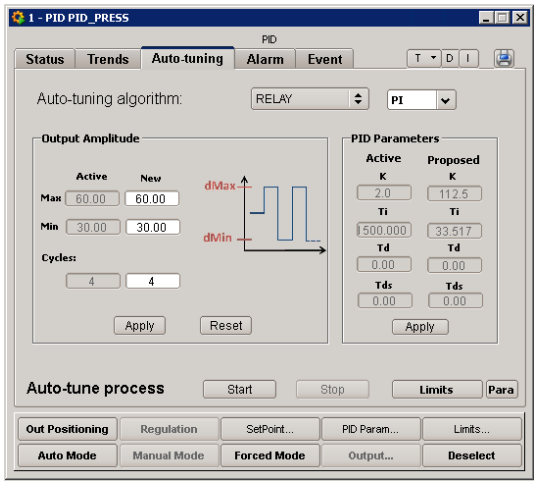


Figura 61 – Panel del método del relé

Una vez que los parámetros estén configurados, pulsando *Start* se puede dar comienzo a la ejecución del método. Cuando el algoritmo termina, en la columna de parámetros propuestos aparecen los nuevos valores de sintonía para el controlador. Si el usuario los considera válidos, pulsando el botón *Apply* pasarán a ser activos.

4.4. Método IFT

El panel del IFT es el que se muestra por defecto al acceder a la pestaña de autosintonía, Figura 62.

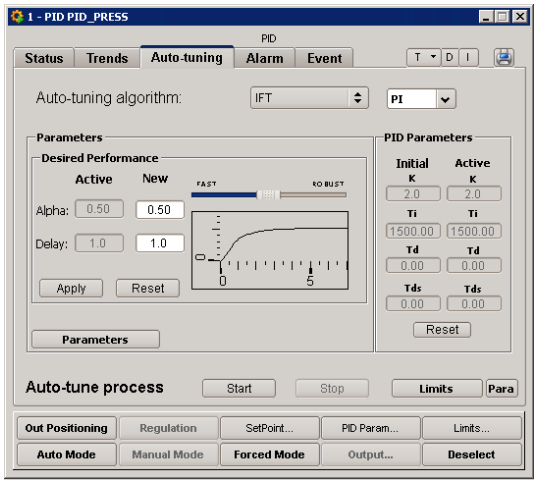


Figura 62 – Panel del método IFT

Los parámetros de configuración del método se han distribuido en dos zonas. En el panel principal se han mantenido los parámetros con los que se configura la respuesta deseada del sistema, entendiendo que serán los que más se puedan modificar.

La respuesta deseada no es un modelo exacto de la respuesta del sistema, sino una referencia que se trata de emular haciendo que el error entre el proceso a sintonizar y ella sea nulo. Esta respuesta deseada se configura con los parámetros:

- ***Retardo (Delay):*** representa el retardo del sistema y debe tener como mínimo el valor de un periodo de muestreo. Se debe introducir el valor numérico en la casilla en blanco y pulsar *Apply* para cargarlo como valor activo.

- ***Alpha:*** hace que la respuesta del sistema sea más agresiva o más robusta. Se configura desde la barra deslizadora, y el valor numérico se puede visualizar en la casilla blanca. De igual modo, para cargarlo como valor activo es necesario aplicar los cambios.

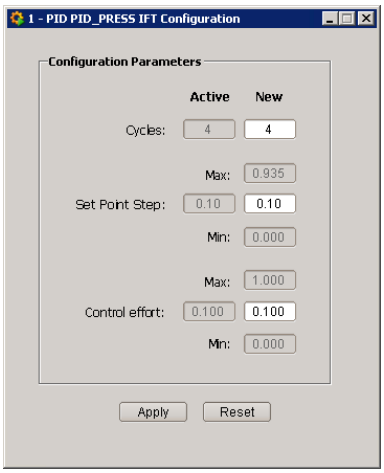


Figura 63 – Parámetros auxiliares del IFT

Los demás parámetros aparecen en la ventana emergente que se muestra en la Figura 63 al pulsar el botón *Parameters*.

- ***Número de ciclos (Cycles):*** indica el número de iteraciones que se van a realizar, a no ser que se detecte que se ha alcanzado el óptimo.

Siempre al terminar los *N* ciclos realiza un experimento 1 más para comprobar el coste de los últimos parámetros PID obtenidos. Normalmente con 4 iteraciones ya se consigue una buena configuración. En el caso de que el número de ciclos sea igual a cero, el método sólo realizará un experimento 1 para realizar el cálculo del coste.

- ***Escalón de la referencia (Set Point Step):*** representa el salto que se aplica en la consigna del sistema en el experimento 1 expresada en las unidades del sistema.
  El salto aplicado debe tener un valor suficiente como para que su efecto pueda diferenciarse del ruido del sistema y, a su vez, no perturbar al proceso más de lo necesario. Para situaciones como esta, es usual recomendar un incremento cuyo valor sea el triple del valor del ruido del sistema.

- ***Esfuerzo de control (Control Effort):*** es la constante lambda que pondera el peso de la acción de control en la función de costes.
  Es conveniente, a la hora de configurar este parámetro, no perder de vista lo que ello conlleva. Asignar un peso pequeño a la variable de control en la función de costes, por ejemplo entre [0.01, 0.1], puede ayudar a controlar acciones excesivamente agresivas que supondrían grandes esfuerzos para los actuadores del sistema. Pero por otro lado, tampoco se pueden buscar respuestas rápidas en el sistema si se penaliza demasiado el esfuerzo de control.

CAPITULO V:

VALIDACIÓN

1. Introducción

La verificación del funcionamiento de la herramienta tiene una doble finalidad. Por una parte, servirá para comprobar que los resultados, en cada caso, se adecúan a lo que sería la respuesta esperada y, por otra, dará una visión general de la metodología que se debe seguir para su utilización.

Los ensayos necesarios han sido realizados sobre diferentes procesos para asegurar lo máximo posible su validez sobre otros sistemas.

En primer lugar se ha utilizado un lazo de presión, al que a partir de ahora llamaremos *Proceso 1*. Esta planta se compone de los siguientes elementos:

- Un autómata Siemens 315-2 PN/DP con módulos auxiliares para entradas y salidas analógicas y digitales.
- Un variador de frecuencia Omron *VS mini J7*.
- Un compresor JUN-AIR *Quiet Air 6-25*.
- Un transductor de presión *Design Insruments TPR-14/N*.

Figura 64 – Proceso 1: lazo de presión

Las demás pruebas se han realizado sobre unas simulaciones proporcionadas por el CERN diseñadas con *Ecosim*. El autómata Siemens permite la posibilidad conectarse mediante un protocolo OPC con las simulaciones e interactuar con ellas a través del PLC como si se tratara de una planta real.

Los modelos se corresponden con un lazo de presión y dos de temperatura, que a partir de ahora denominaremos *Simulación 1, Simulación 2 y Simulación 3*.

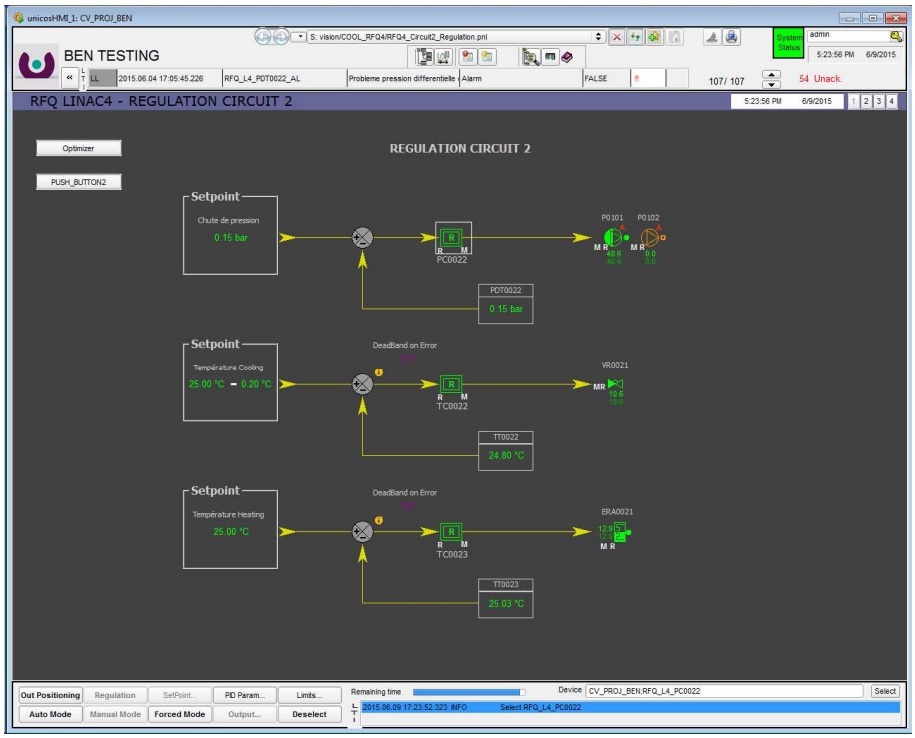


Figura 65 – Simulaciones de lazos de presión y temperatura

1.1. Proceso 1: lazo de presión.

El funcionamiento del proceso consiste en variar la frecuencia de funcionamiento del motor del compresor para así aumentar o disminuir la presión en la salida del sistema.

El primer paso será estudiar las condiciones generales que se deben definir para configurar la aplicación de autosintonía para el sistema descrito.

1.1.1. Consideraciones generales para todos los métodos

1.1.1.1. Estructura del controlador

Los lazos de presión son sistemas rápidos y ruidosos y este caso no se está aplicando ningún filtrado a la salida del proceso. Por ello, la configuración más adecuada será un PI, puesto que si se incluyera el término derivativo sólo se lograría acentuar el ruido del sistema.

1.1.1.2. Límites del proceso de autosintonía

El regulador de la planta se va a sintonizar partiendo de unas condiciones iniciales de variable de control y de salida del sistema de unos 0.8bar y 15Hz. Los límites del proceso, actualmente, están definidos dentro del intervalo [0, 4]bar y [0, 72]Hz. En base al conocimiento del sistema, se estima que durante la ejecución de los procedimientos y partiendo de esas condiciones iniciales no se deberían superar los [0, 1.5]bar.

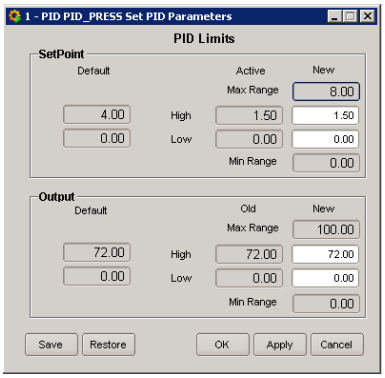


Figura 66 – Configuración de límites

1.1.1.3. Tiempo de muestreo

Haciendo un pequeño ensayo en lazo abierto, Figura 67, se puede ver que en dos minutos el sistema ya se encuentra estabilizado. Con esta información se puede determinar que un periodo de muestreo de 4 segundos es suficiente.

$$\frac{120s}{30S} = 4\,s/S$$

Si se diera el caso de que el sistema tuviera un tiempo de muestreo menor que 2 segundos, habría que abstenerse de utilizar el método IFT.

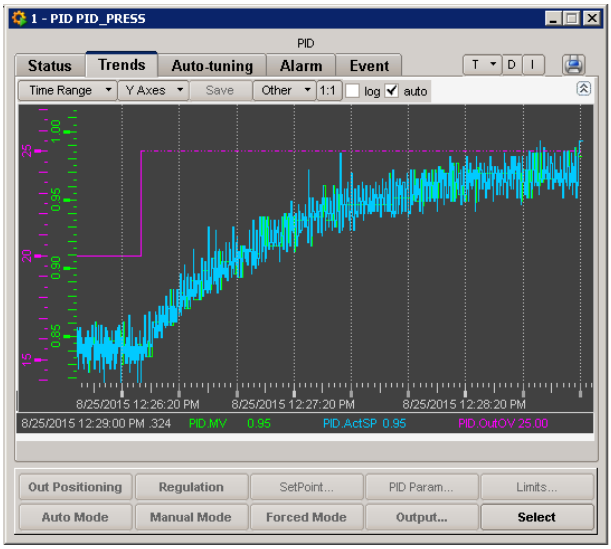


Figura 67 – Ejecución de pre-test

1.1.1.4. Acción directa/inversa del controlador

Si se analiza de nuevo la Figura 67, también se puede ver que la ganancia del sistema es positiva, puesto que un incremento en la salida del regulador conlleva un aumento en la salida del sistema. La acción del controlador es, por tanto, inversa.

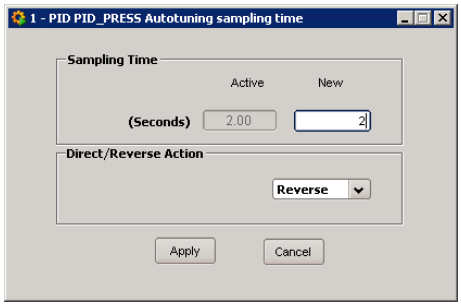


Figura 68 – Configuración de parámetros auxiliares

1.1.2. Método S-IMC

El modelo de función de transferencia de la planta actual no se conoce, por lo que no se puede introducir los valores directamente mediante la opción de editar. La alternativa será realizar la identificación del sistema mediante un ensayo en lazo abierto.

1.1.2.1. Identificación: Aproximación por un modelo FOPD

El único parámetro que se necesita introducir es el incremento que se va a aplicar a la variable manipulada. En base al pre-test que se ha realizado se puede determinar que, con un Δu=5Hz se obtiene un Δy=0.15bar. Ese incremento es suficiente como para obtener la identificación del sistema y no lo perturba demasiado ni lo aleja de su punto de trabajo.

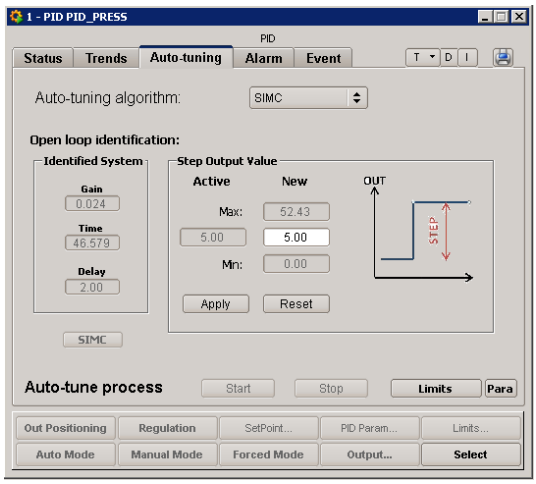


Figura 69 – Configuración de parámetros de la identificación

La ejecución completa del método se muestra en la Figura 70. Se puede distinguir al inicio el comienzo del experimento, cuando el valor de la salida del controlador pasa a ser constante. Después, se empieza la primera comprobación de estacionario y, cuando está verificado, se aplica el escalón. Al detectar el nuevo estacionario el método finaliza y se recuperan las condiciones iniciales de funcionamiento.

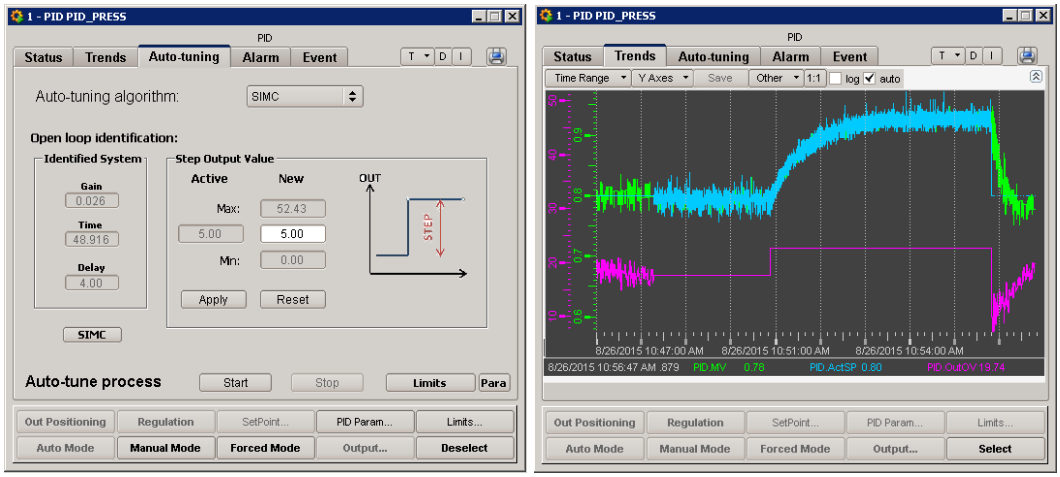


Figura 70 – Ejecución de la identificación y resultados obtenidos

Como el proceso no tiene retardo, se toma el valor mínimo, que es el de un periodo de muestreo. Si se analizan el valor de la ganancia y la constante de tiempo del sistema se puede comprobar a grandes rasgos que son válidos.

$$K_C = \frac{\Delta y}{\Delta u} = \frac{0.93 - 0.8}{5} = 0.026$$

$$0.638 \cdot \Delta y = 0.638 \cdot (0.93 - 0.8) = 0.083 \rightarrow y(\tau) = 0.8 + 0.083 \rightarrow \ \tau \approx 50s$$

Se puede verificar que los parámetros correspondientes a la identificación que se muestran en la Figura 70 – son correctos.

1.1.2.2. Sintonía del regulador PID

Pulsando el botón SIMC podemos redirigirnos al panel inicial del método. El único parámetro que hay que configurar es el valor τ_c. Si aceptamos la recomendación de tomar τ_c =θ obtenemos el siguiente resultado que se muestra en la Figura 71.

La respuesta real del sistema es menos agresiva que lo que se presuponía, pero es bastante rápida. Si no fuera suficiente, se podría repetir el experimento con una nueva configuración.

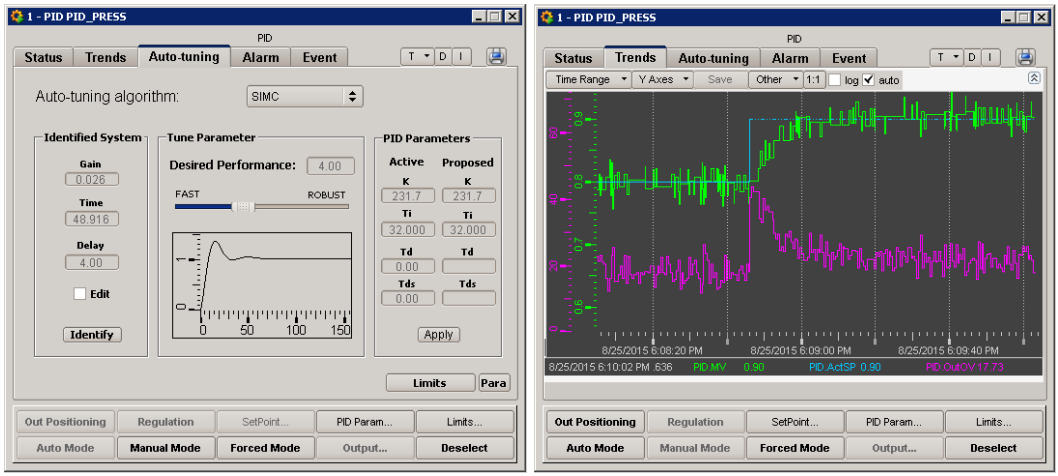


Figura 71 – Obtención de parámetros PID medianteS-IMC y resultados

1.1.3. Método del relé

El método del relé tiene como parámetros de configuración los valores de salida del controlador entre los que se va a hacer oscilar el sistema y el número de iteraciones que se quieren realizar.

Se tiene que seleccionar unos valores con los que el sistema vaya a ser capaz de alcanzar valores de ±3 veces el ruido del sistema. Haciendo referencia al pre-test de la Figura 67, se puede ver que hay un ruido de 0.04bar aproximadamente. Para que los valores de la salida en la histéresis puedan alcanzar estos valores, conociendo la ganancia del sistema, tendrán que tener, al menos los valores:

$$\frac{3 \cdot (\pm 0.04)}{0.026} = \pm 4.6\text{Hz}$$

En base a los cálculos, se configurarán unos valores de ±7.5Hz respecto al valor inicial de salida del controlador para asegurar que se va a alcanzar el error. Si con estos valores no se consiguiera que el sistema alcance el error requerido, se aumentarán.

Normalmente, el número de iteraciones necesarias para conseguir que las oscilaciones sean mantenidas no es mayor que cuatro.

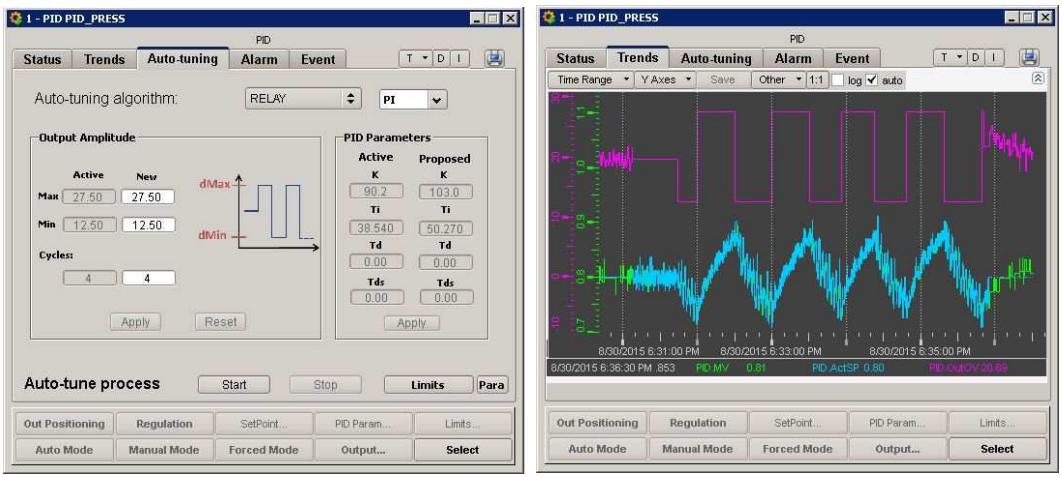


Figura 72 - Configuración de parámetros del método del relé y ejecución del procedimiento

El experimento comienza cuando la salida pasa a ser un valor constante. A partir de ese momento, se realizan los cuatro ciclos de histéresis y se devuelve al sistema a la configuración de funcionamiento inicial.

En el apartado *PID Parameters – Proposed*, aparecen los nuevos valores calculados para el controlador. Pulsando *Apply* los cargamos como valores activos del regulador y, a continuación, realizamos un ensayo aplicando un incremento a la referencia del sistema para así verificar su comportamiento, Figura 73.

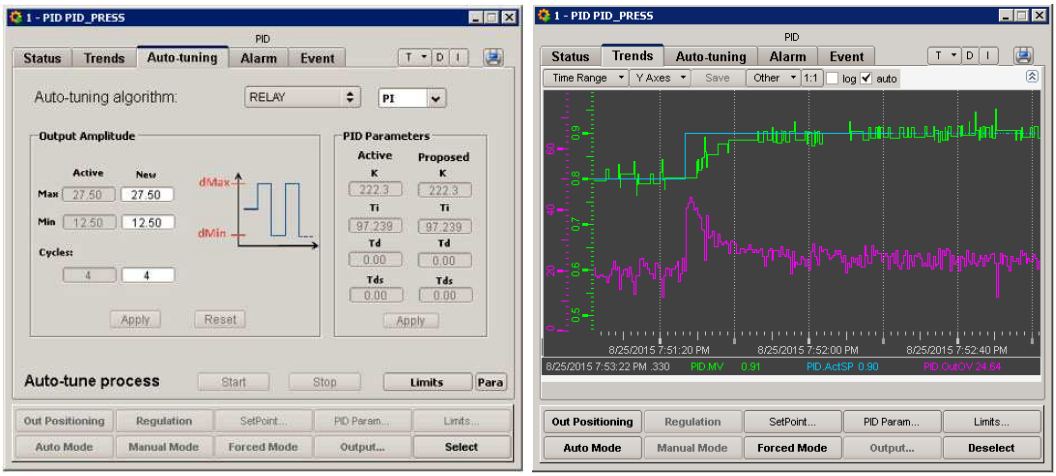


Figura 73 – Aplicación de parámetros PID y resultados

La configuración del regulador PID nos está generando, en este caso, respuestas lentas. A continuación, utilizando el método IFT trataremos de mejorar este comportamiento.

1.1.4. Método IFT

Una consideración importante para este método son los parámetros PID iniciales, porque será lo que determine si se va a converger hacia un punto óptimo o no. Para asegurar que los parámetros PID iniciales son aceptables, comenzaremos con la solución generada por el método del relé.

El método IFT es, en comparación con los demás, el que más parámetros de configuración tiene. Para facilitar la labor del usuario se han dividido en dos partes, Figura 74.

En primer lugar se analizarán aquellos parámetros que, en principio, sólo se van a configurar la primera vez que se ejecute el método y, después, los parámetros con que se diseña la respuesta deseada.

Normalmente, la mayoría de los sistemas con tres o cuatro iteraciones como máximo consiguen alcanzar unos buenos parámetros de funcionamiento.

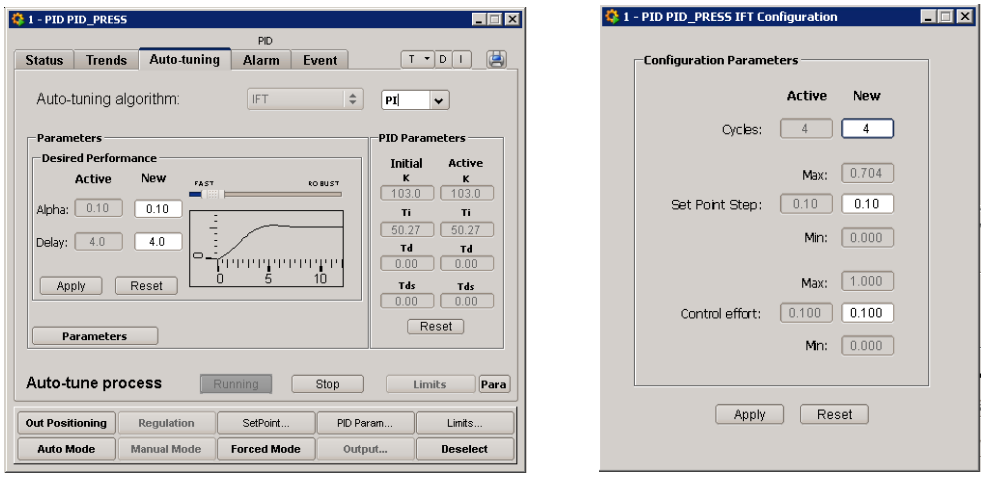


Figura 74 - Configuración de parámetros para el método IFT

El escalón que se aplicará a la referencia en el experimento 1, *Set Point Step*, deberá ser suficientemente alto como para que su efecto pueda diferenciarse del ruido del sistema, por lo que se tomará un Δr=0.1bar.

El peso asignado al esfuerzo de control en la función de costes será de 0.1, de manera que se tenga en cuenta su efecto, pero no tenga prioridad sobre la respuesta deseada.

La respuesta deseada tiene dos datos de diseño. Por un lado el retardo del sistema deberá fijarse, como en ocasiones anteriores, en un periodo de muestreo, puesto que el sistema no tiene retardo.

Por otra parte, la rapidez del sistema se establece con la barra deslizadora que se muestra sobre la gráfica. Se ha fijado un valor de 0.1 para buscar una respuesta rápida.

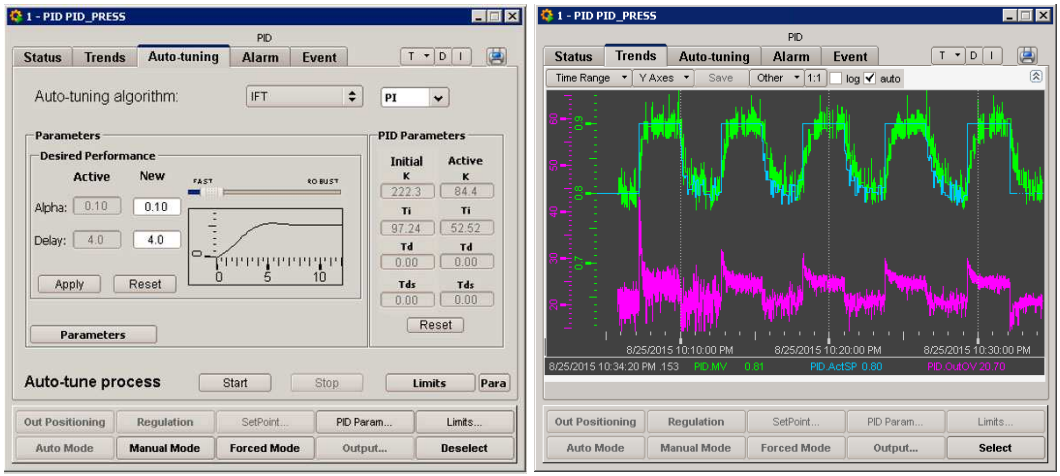


Figura 75 – Ejecución del IFT partiendo de los valores del relé

En la Figura 75 se puede ver la evolución que han sufrido las variables controlada y manipulada del sistema en el proceso de optimización. Los parámetros en cada iteración han sido los siguientes:

		Iteración 1	Iteración 2	Iteración 3	Iteración 4
P	103.013	71.7625	68.748	65.366	63.859
T_i	50.27	44.183	43.773	42.523	42.421
Coste	35.312	34.069	33.569	33.201	33.068

Tabla 1

Se puede apreciar tanto visualmente en la Figura 75 como numéricamente en la tabla anterior, que la variación más significativa se ha producido en la primera iteración. La acción del control en el instante inicial era demasiado agresiva y gracias al ajuste del IFT, se ha conseguido evitar pero en general en este caso el procedimiento no ha supuesto una mejora significativa en cuanto a la forma de la respuesta del sistema.

Si al terminar con la ejecución del método los nuevos parámetros no fueran considerados válidos por el usuario, pulsando el botón *Reset* se pueden recuperar los valores iniciales de configuración del regulador PID.

Para demostrar la importancia que tienen los valores PID de partida, a continuación se va a realizar el mismo experimento aplicado a los valores que se obtuvieron en el S-IMC.

La Figura 76 muestra el resultado de la ejecución del método y la evolución que van siguiendo las variables del sistema.

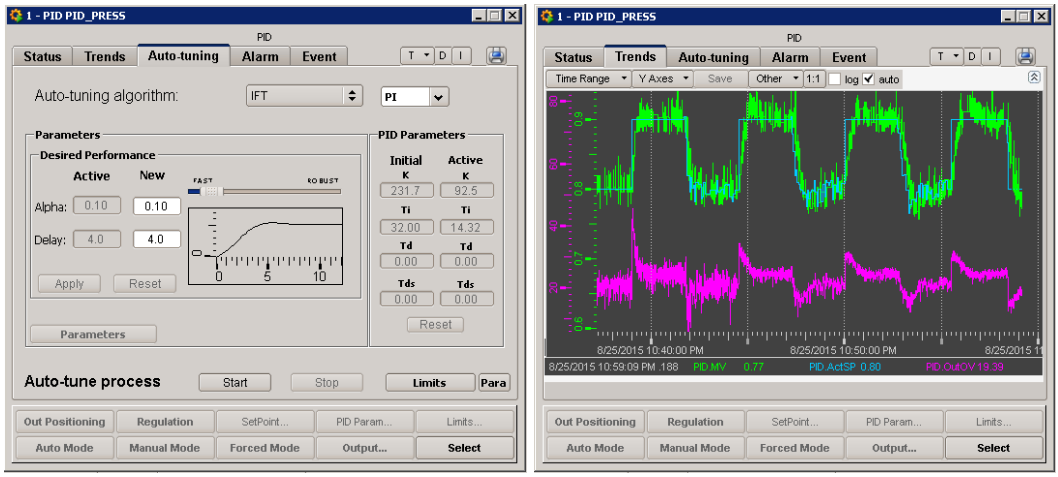


Figura 76 - Ejecución del IFT partiendo de los valores del S-IMC

	Iteración 1	Iteración 2	Iteración 3	Iteración 4
P	231.7	113.22	92.506	~~86.723~~
T_i	32	16.45	14.315	~~13.844~~
Coste	29.020	19.909	16.449	17.565

Tabla 2

Si se analizan, en primer lugar, los resultados numéricos obtenidos en cada iteración que se muestran en la Tabla 2, se puede ver al compararlos con la Tabla 1 que no converge hacia el mismo punto. Esto se debe, como se ha dicho en ocasiones anteriores, a que el IFT localiza los puntos óptimos que se hallen en el entorno del punto de partida. Como en este caso los resultados generados por el S-IMC y el relé diferían bastante, el IFT ha convergido hacia distintos óptimos locales en cada caso.

Otro aspecto a considerar en este último ensayo es que no se han completado las 4 iteraciones prefijadas por el usuario. En el tercer ciclo el método detectó un aumento en la función de costes, que supondría un empeoramiento de las condiciones de funcionamiento anteriores, por lo que los valores del segundo ciclo fueron recuperados y se terminó con la ejecución del método.

Figura 77 – Comparativa entre los dos resultados obtenidos del IFT

En la Figura 77 se muestran, una a continuación de otra, la respuesta del sistema con los parámetros resultantes de los ensayos que se han realizado. A la izquierda, con los obtenidos partiendo del relé y a la derecha, del S-IMC.

El primer caso ha convergido a una respuesta del sistema más robusta y con una acción de control menos agresiva, pero en este caso no era lo que se pretendía, puesto que se había configurado la respuesta deseada para que fuera rápida. El segundo caso cumple mejor con las especificaciones de diseño. Aunque la acción de control es algo más fuerte, la respuesta final del proceso coincide con lo que se buscaba.

1.2. Simulación 1: lazo de presión

El uso de los métodos de identificación, ya sea en lazo abierto o cerrado, es muy útil a la hora de realizar una primera parametrización del regulador de una planta, pero los valores que generan necesitan de un ajuste más preciso a posteriori.

El IFT, en cambio, requiere estar inicialmente posicionado en el entorno del punto óptimo de funcionamiento del controlador, pero una vez fijado el punto de arranque del proceso, obtendrá los valores de sintonía que optimicen el funcionamiento del proceso.

Es por ello que puede utilizarse tras la ejecución de otro método, como el S-IMC o el relé, o usarse de forma exclusiva para mejorar el funcionamiento del sistema o para eliminar algún comportamiento indeseado.

El análisis de las simulaciones se centrará en este segundo aspecto. Se planteará una situación inicial en la que el funcionamiento del sistema no sea el buscado y, aplicando el IFT, se tratará de corregir el comportamiento no deseado. Se realizará una valoración de la respuesta en rasgos generales, sin entrar en detalles numéricos como en los apartados anteriores.

Por ejemplo, en el caso de la simulación de presión, se ha comenzado con un supuesto en el que la acción de control tiene un sobrepico al inicio que no es admisible. Para esta situación el algoritmo IFT permite reajustar forma de la salida del controlador manteniendo el compromiso que existe con la forma de la respuesta deseada.

En la Figura 78 se observa la evolución del sistema durante la ejecución del IFT y cómo, paso a paso, se va suprimiendo el sobrepico del que se hablaba, aunque como es normal, la rapidez del sistema se ve afectada.

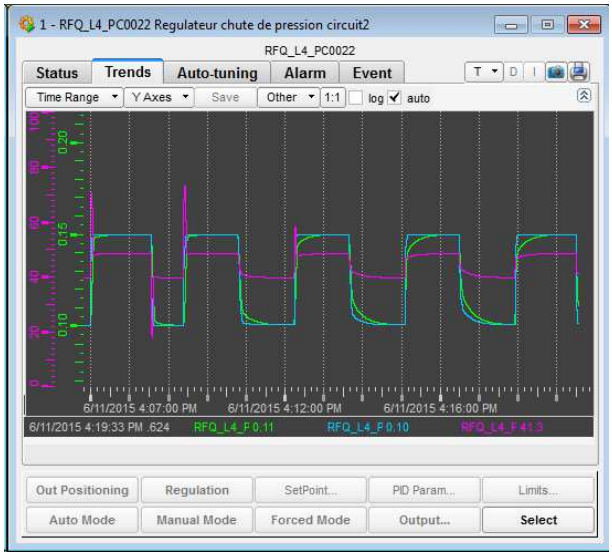


Figura 78 – Ejecución del IFT en simulación de proceso de presión

1.3. Simulación 2: lazo de temperatura

El método IFT se ha aplicado sobre esta simulación con una configuración tal que lo que se busca es una respuesta más robusta del sistema, pero en la cual no se ha tenido en cuenta la acción de control, es decir, se ha considerado $\lambda=0$.

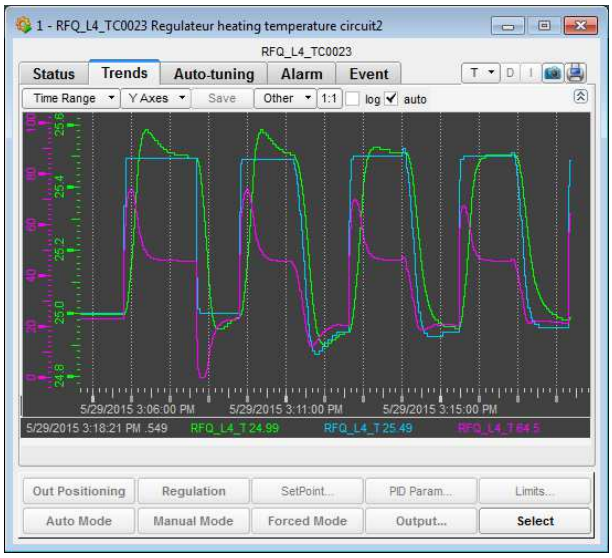


Figura 79 – Ejecución del IFT en simulación de proceso de temperatura

La respuesta inicial del sistema era bastante agresiva, pero se puede ver como en cada ciclo se va eliminando el sobrepico sin afectar por ello a la rapidez del sistema, que mantiene inalterado su tiempo de subida y mejora notablemente el tiempo de establecimiento.

1.4. Conclusiones

Los métodos implementados fueron seleccionados con el fin de ser capaces realizar la autosintonía completa de un sistema, sin necesidad de tener un conocimiento previo del modelo de la planta.

En este capítulo se ha verificado la validez de la metodología desarrollada, basada en la utilización inicial de un método de identificación inicial, seguida de un método de optimización que realice la labor de afinar el ajuste inicial.

También se ha comprobado que el buen funcionamiento del método IFT para mejorar lazos que ya estuvieran previamente sintonizados pero que, por alguna razón, no estuvieran trabajando exactamente del modo en que se deseaba.

CAPITULO VI:

TRABAJO FUTURO

La aplicación desarrollada en este proyecto ha servido para sentar las bases de una herramienta de optimización del funcionamiento de controladores PID. El hecho de que la implantación se haya realizado sobre la capa de supervisión, deja las puertas abiertas a la introducción de una gran gama de mejoras, puesto que el consumo de recursos y la capacidad de cálculo no suponen una restricción importante en este nivel.

A continuación se proponen distintas ramas de continuación para proyectos futuros.

1. Mejoras sobre la aplicación de autosintonía

El primer campo a analizar son los trabajos de mejora sobre la aplicación actual. La herramienta se ha diseñado con la idea de contener diferentes tipos de métodos que fueran diferentes, para poder utilizarse en distintos contextos, y a su vez se complementaran entre ellos. Con este pensamiento se tomó la decisión de incluir los siguientes:

- S-IMC: basado en la identificación en lazo abierto.
- Relé: basado en la identificación en lazo cerrado.
- IFT: basado en la optimización local del funcionamiento.

El S-IMC y el relé están pensados para generar unos primeros valores de sintonía para un lazo, que no tienen por qué ser los ideales, pero que al menos garantizan que no van a generar comportamientos graves en el sistema.

Después, el IFT ya se encarga de optimizar el funcionamiento, aunque eso sí, dentro del entorno del punto de configuración inicial del regulador.

El trabajo futuro estaría en el campo que abarcan los dos primeros métodos, debido a que la utilización de ambos está restringida a los sistemas de primer orden.

En el caso del S-IMC su utilización se puede ampliar a sistemas de segundo orden. Para ello, en primer lugar habría que implantar un nuevo algoritmo de identificación para sistemas se segundo orden. Después, una vez que el modelo estuviera identificado, se aplicarán las reglas de Skogestad para sistemas de segundo orden.

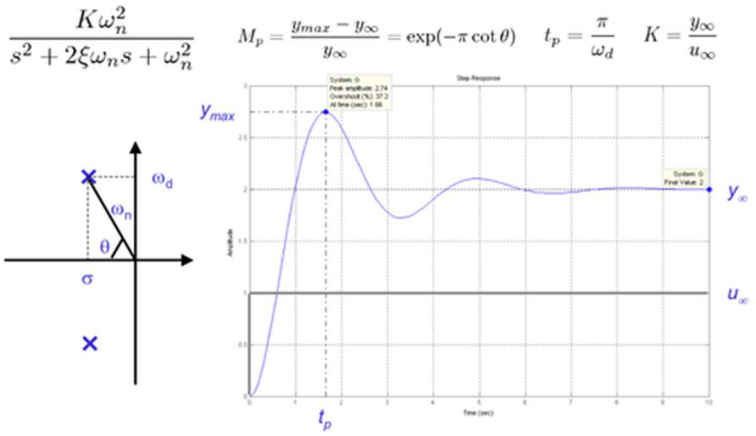


Figura 80 – Identificación de sistemas de segundo orden

El relé, en principio, no tiene una metodología que pueda servir para cualquier sistema de segundo orden (sobreamortiguado, subamortiguado,...). Si se quiere mantener un método que abarque este tipo de sistemas habrá que buscar otras alternativas.

El método IFT no presenta esta problemática, puesto que no depende en ningún momento del modelo de la planta, si no que utiliza en vez de ello la respuesta del sistema.

Otro posible campo para abordar son la gama de sistemas de dinámica difícil. Existe una variedad de sistemas que tienen características en su respuesta que complican la sintonía de sus reguladores, como pueden ser los sistemas con grandes retardos, los de fase no mínima o procesos que contienen integradores. En concreto, los sistemas térmicos suelen presentar la problemática de ser procesos con grandes tiempos de retardo.

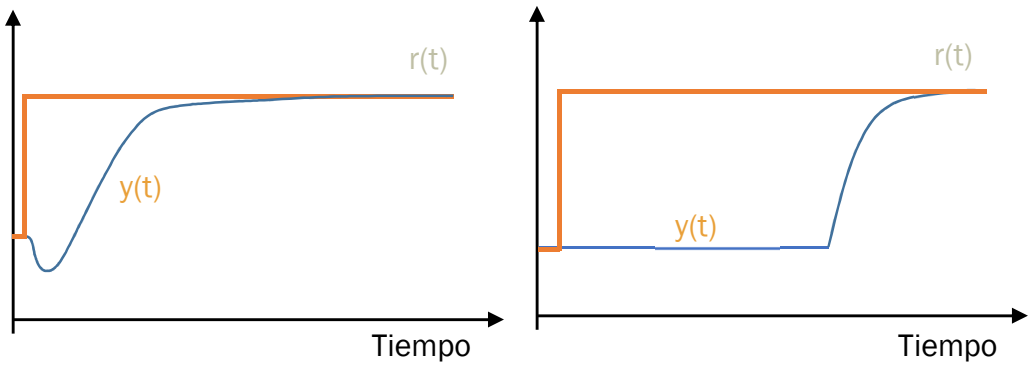


Figura 81 – Sistema de fase no mínima. Sistema con gran retardo

El LHC, *Large Hadron Collider,* utiliza fuertes campos magnéticos generados por electroimanes construidos a partir de bobinas de cable que trabaja en estado superconductor. Esto requiere el enfriamiento de los imanes a temperaturas de, aproximadamente, -271ºC. Por esta razón, sería muy interesante que el algoritmo implementado prestara una especial atención a este tipo de sistemas.

H. Hjalmarsson presenta en su artículo "*Iterative Feedback Tuning: Theory and Applications"* una amplia casuística de modificaciones que se pueden implementar como complemento al algoritmo original y que mejoran su funcionamiento para estos casos particulares de los que se ha hablado.

2. Supervisión de controladores PID

La aplicación de autosintonía se ha diseñado para que su funcionamiento se ejecute a petición del usuario y además, solamente usuarios que tengan los derechos de acceso pueden hacer uso los procedimientos cuando lo crean conveniente.

En el CERN hay varios miles de lazos de regulación, por lo que ejecutar esta tarea de forma manual uno a uno sería una tarea ardua. La mejor solución consistiría en que el propio sistema pudiera auto-diagnostica. En el caso de que se detectara un funcionamiento inadecuado, la autosintonía entraría de forma automática para corregirlo.

Este tipo de aplicación sería distinta a la generada para la autosintonía, puesto que se trataría de un algoritmo que se está ejecutando de forma periódica o continua, pero no a petición del usuario en cualquier caso. Por esta razón, habría que realizar de nuevo el estudio que se hizo en el Capítulo II acerca cuál sería la mejor opción para su implantación dentro de la estructura de UNICOS.

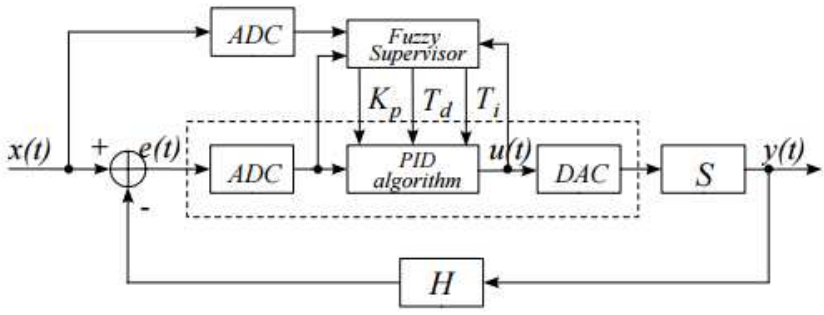


Figura 82 – Aplicación de fuzzy logic a sintonía de PIDs

La lógica difusa o *fuzzy logic* ha demostrado muy buenos resultados dentro del campo de la sintonía de reguladores PID, especialmente dentro de los procesos que se han definido como sistemas de dinámica difícil.

Estos algoritmos difusos combinan la tarea de supervisión y sintonía de controladores PID de forma simultánea, pero habría que estudiar si su funcionamiento es suficientemente robusto para las aplicaciones en que se pretende utilizar.

CAPÍTULO VII:

CONCLUSIONES

El PID es el regulador más ampliamente extendido en el entorno industrial debido a su simplicidad, su robustez y su eficacia pero estas características pueden llegar a ser, en cierto modo, contraproducentes. El hecho de que con una mala configuración de los parámetros de controlador se pueda conseguir que el sistema funcione, aunque no sea en las condiciones más adecuadas, hace que un alto porcentaje de los PIDs que hay implantados en la industria no estén trabajando de manera óptima.

Además, aunque la evolución de la teoría de control haya desarrollado nuevos algoritmos de optimización para el control de procesos, en la base de muchos de ellos continúa estando el PID, de manera que la buena parametrización de esta primera capa de control es trascendental para el buen funcionamiento de cualquier sistema.

El CERN no deja de ser un medio en el que el PID está presente a cada paso, con la característica que le otorgan, además, la complejidad de los procesos con los que trabaja. Por ello, una buena sintonía de sus reguladores es para ellos tan necesaria, al menos, como para cualquier industria.

La mayoría de los sistemas SCADA comerciales, no permiten el acceso a niveles tan profundos de desarrollo como lo hace *WinCC Open Architecture*. El trabajar en este entorno ha posibilitado la creación de una herramienta de autosintonía para el CERN, pero deja las puertas abiertas a que la aplicación continúe creciendo y pueda llegar incluso a ser implementada en otros entornos más usuales en el mundo industrial.

A título personal, este proyecto ha supuesto una apertura a la complejidad del mundo de la regulación de procesos. Es sorprendente ver cómo procedimientos que llevan utilizándose décadas, como es el caso del controlador PID, a día de hoy continúan sin estar aplicados de manera eficiente en la mayoría de los procesos.

Desde el momento en que se comenzó con el análisis de la planta que se quería implementar (nivel de campo), pasando por la programación del *grafcet* y la lógica del funcionamiento de la planta en el autómata (nivel del control), hasta el desarrollo de la aplicación de autosintonía en el SCADA (nivel de supervisión), se ha tenido contacto con todos los niveles de la automatización que abarca UNICOS. Esto ha supuesto una gran ayuda a la hora de comprender lo que abarca la arquitectura de la automatización, así como las implicaciones que conllevan cada uno de los niveles que se definen en su jerarquía: nivel de campo, nivel de control, nivel de supervisión,…

En última instancia, a nivel de implementación, el diseño de la herramienta ha supuesto una posibilidad de aplicar todos los conocimientos teóricos adquiridos durante los años formación académica sobre sistemas discretos, puesto que hasta el momento solamente había tenido contacto con sistemas continuos.

BIBLIOGRAFÍA

- Ph. Gayet and R. Barillère. "UNICOS a framework to build industry like control systems: Principles Methodology".
- 10th ICALEPCS Int. Conf. on Accelerator and Large Expt. Physics Control Systemns. Geneva (Switzerland), 2005.
- L. Evans and P. Bryant. "The CERN Large Hadron Collider: Accelerator and Experiments".Journal of Instrumentation, 2008
- J.G. Ziegler and N.B. Nichols. "Optimum setting for automatic controllers".Transactions of the ASME,64, 1942.
- K. Aström and T. Hägglund. "Automatic tuning of simple regulators on phase and amplitude margins".Automatica, 20, 1984.
- S. Dormido and F. Morilla. "Auto-tuning and Antiwindup methods in PID controllers".UNED Press, 1995
- R. Russell Rhinehart."Automated Steady and Transient State Identification in Noisy Processes". American Control Conference (ACC), Washington (USA), 2013
- O'Dwyer. "Handbook of PI and PID Controller Tuning Rules".Imperial College Press, 2006
- Sigurd Skogestad and Chriss Grimholt. "The SIMC Method for Smooth PIDController Tuning". PIDControl in the Third Millennium.Springer. 2012
- H. Hjalmarsson, M. Gevers and O. Lequin. "Iterative feedback tuning: Theory and applications". IEEE Control Systems Magazine 18(4),1998.
- W.K. Ho, Y. Hong, A. Hansson, H. Hjalmarsson, J.W. Deng. "Relay auto-tuning of PID controllers using iterative feedback tuning". Automatica 39, 2003.
- Rogelio Mazaeada, César de Prada. "IFTtune: a PID automatica tuning software tool". IFAC Conference on Advances in PID Control (PID12), Brescia (Italy),2012.

ANEXO 1:

CÓDIGO FUENTE

```
/*-----------------------------------------------------------------------------*/
// $License: NOLICENSE
// **@file
//  *
//  * This library contains algorithms for auto-tuning
//  *
//  * @author Rubжn Martь (EN-ICE)
//  *         Laura de Frutos
//  * @date   August 2015
//  * @version
//  */
/*-----------------------------------------------------------------------------*/


/*****************************************************************************/
/************************************ IFT ************************************/
/*****************************************************************************/
/* - Steps:                                                                  */
/*      - Calculate the Auxiliar Vectors                                     */
/*      - Experiment 0
*/
/*      - Experiment 1                                                       */
/*      - Calculate the Cost Function                                        */
/*      - Experiment 2                                                       */
/*      - Calculate the partial derivative of PID function                   */
/*      - Calculate the derivative of cost function                          */
/*      - Calculate the new PID parameters                                   */
/*      - End of AutoTunning                                                 */
/*****************************************************************************/


IFT(string deviceName, int N, float RStep, float Delay, float alpha, int nCycles,
float Samp_Time, float Lambda)
{
  dyn_string exceptionInfo;
  bool alreadySent;
  int Disc_Delay;  //System delay (samples)

// Initial variables
  bool enable_IFT;
  float SP_Init, SP_max, SP_min;
  float Out, Out_max, Out_min;
  dyn_float yExp0, uExp0;
  float ySteady, ySteady_In;
  float uSteady, uSteady_In;
  float max_Noise, min_Noise;
  float Ny;

// Experiments
  float y;
  dyn_float yExp1, yExp2;
  float u;
  dyn_float uExp1, uExp2;
  float Uo, SP, R;
  float Gamma = 1;
  dyn_float Cy, Cy_Ant, Cy_Init, Inc_Cy;  // PID parameters
  float J, J_ant;                         // Cost
```

```
  bool Exp_OK = TRUE;
  bool End_Opt = FALSE;
  bool ERROR=FALSE;


// Program variables
  int iter=0;
  int i=1, j=1, k=1;

// Calculation variables
  dyn_float R_vector; // R Vector (Discrete step input)
  dyn_float yd;       // Desired output (R vector filtered)
  dyn_float yhat;     // Error

  dyn_float numK, denK, numTi, denTi, numTd, denTd;
  dyn_dyn_float dy, du;
//   float T=Samp_Time, Te2=pow(T,2);
  float T, Te2;
  dyn_float Cye2;
  dyn_float dJ;
  dyn_dyn_float Hessian;
  dyn_dyn_float Inv_Hessian;
  float det_Hessian;

// Object to modify the Setpoint
  string StringSP;
  string AnalogParameter;
  bool EnableSP;

  dpGet(deviceName + ".AutoTunning.IFT.EnableSP",EnableSP);



/*-------------------------------------------------------------------------------*/
/*                              Auxiliar Vectors                                 */
/*-------------------------------------------------------------------------------*/

 dyn_float Num_Filter, Den_Filter;

 Disc_Delay = floor(Delay/Samp_Time);

 if (Disc_Delay==0)
 {
   Disc_Delay=1;
 }

 dynClear(Num_Filter);
    for (i=1; i<=(Disc_Delay+1); i++)  // Filter numerator
    {
      if (i<=(Disc_Delay))  { Num_Filter[i]=0; }
      else                  { Num_Filter[i]=1-alpha; }
    }

 dynClear(Den_Filter);
    for (i=1; i<=2; i++)            // Filter denominator
    {
      if (i==1)  { Den_Filter[i]=1; }
```

```
      else       { Den_Filter[i]=(-1)*alpha; }
    }

 dyn_dyn_float Ident;  // Identity matrix [Ny,Ny]
 dynClear(Ident);
 for (i=1;i<=Ny;i++){
   for (j=1;j<=Ny;j++){
     if (i==j)  { Ident[i][j]=1; }
     else       { Ident[i][j]=0; }
   }
 }
//  DebugTN("Ident: ", Ident);



/*-------------------------------------------------------------------------------*/
/*                                Experiment 0                                   */
/*-------------------------------------------------------------------------------*/

if (NumParam_Controller.text()=="PI")       { Ny=2; }
else if (NumParam_Controller.text()=="PID") { Ny=3; }
DebugTN("Controller parameters: ",Ny);

if (Samp_Time<2.0)
{
  i=2;
  while (Samp_Time<2.0)
  {
    Samp_Time = Samp_Time*i;
    i++;
  }
}
DebugTN("Samp_Time: ", Samp_Time);
T=Samp_Time;
Te2=pow(T,2);


dynClear(Cy);
dpGet (deviceName+".ProcessInput.ActKc", Cy[1]);
dpGet (deviceName+".ProcessInput.ActTi", Cy[2]);
if (Ny==3)
{
  dpGet (deviceName+".ProcessInput.ActTd",  Cy[3]);
  dpGet (deviceName+".ProcessInput.ActTds", Cy[4]);
}
else
{
  Cy[3]=0;
  Cy[4]=0;
}
Cy_Init = Cy;       // PID parameters -> Iter(0)
Cy_Ant = Cy_Init;   // PID parameters -> Iter(i-1)
DebugTN("Cy_Init: ",Cy_Init);

// Set Point Limits
dpGet (deviceName+".AutoTunning.Limits.SP_Max", SP_max);
DebugTN("SP_max: ",SP_max);
```

```
dpGet (deviceName+".AutoTunning.Limits.SP_Min", SP_min);
DebugTN("SP_min: ",SP_min);

dpGet (deviceName+".ProcessInput.ActSP", SP_Init);
dpSet (deviceName+".AutoTunning.IFT.SP_Init", SP_Init);
DebugTN("SP_Init: ",SP_Init);

// Output Limits
dpGet (deviceName+".AutoTunning.Limits.Out_Max", Out_max);
DebugTN("Out_max: ",Out_max);
dpGet (deviceName+".AutoTunning.Limits.Out_Min", Out_min);
DebugTN("Out_min: ",Out_min);


dpGet(deviceName + ".AutoTunning.IFT.Active",enable_IFT);
// Average value of controlled variable
dynClear(yExp0);
dynClear(uExp0);
for (i=1; i<=10; i++)
{
  dpGet(deviceName + ".AutoTunning.IFT.Active", enable_IFT);
  if (enable_IFT == FALSE) { break; }
  dpGet (deviceName+".ProcessInput.MV", yExp0[i]);
  dpGet (deviceName+".ProcessInput.OutOVSt", uExp0[i]);
  delay(Samp_Time);
}

if (enable_IFT)
{
  ySteady_In = dynAvg(yExp0);
  ySteady = ySteady_In;
  DebugTN("ySteady_In: ", ySteady_In);
  uSteady_In = dynAvg(uExp0);
  uSteady = uSteady_In;
  DebugTN("uSteady: ", uSteady);

  DebugTN("Initial Set Point: ", SP_Init);
  // Step input
  DebugTN("Step: ", RStep);

  // Discrete step input
  dynClear(R_vector);
  for (i=1; i<=N; i++) { R_vector[i]= RStep; }

  // Desired output -> Filtered step input
  dynClear(yd);
  yd = Filter(Num_Filter, Den_Filter, R_vector);

  // Desired output (%)
  for (i=1; i<=N; i++) { yd[i]= 100*yd[i]/ySteady; }


  if ((SP_Init+RStep)>SP_max || (SP_Init+RStep)<SP_min)
  {
    DebugTN("SP out of limits");
    dpSet(deviceName + ".AutoTunning.IFT.Active",FALSE);
    Exp_OK==FALSE;
```

```
  }
  else
  {
    DebugTN("Start IFT");

    //Activate Manual Mode
    unGenericObject_SendPulse(deviceName, ".ProcessOutput.ManReg01",
    CPC_ManReg01_MMMOR, exceptionInfo, alreadySent);

    //Activate Regulation
    unGenericObject_SendPulse(deviceName, ".ProcessOutput.ManReg02",
    CPC_ManReg02_MREGR, exceptionInfo, alreadySent);
  }
}

/*-----------------------------------------------------------------------------*/

dpGet(deviceName + ".AutoTunning.IFT.Active",enable_IFT);
while (enable_IFT==TRUE)
{


/*-----------------------------------------------------------------------------*/
/*                                Experiment 1                                 */
/*-----------------------------------------------------------------------------*/

  DebugTN("иииииииииииииииии First experiment иииииииииииииииии");

//   t1 = getCurrentTime();
  if (SP_Init+RStep>=SP_min && SP_Init+RStep<=SP_max)
  {
    if(EnableSP)
    {
      StringSP = deviceName+".ProcessOutput.MSP";
      //Write and send desired SP value
      dpSet(StringSP, SP_Init+RStep);
      unGenericObject_SendPulse(deviceName, ".ProcessOutput.ManReg01",
      CPC_ManReg01_MNEWSPR, exceptionInfo, alreadySent);
    }
    else
    {
      dpGet(deviceName+".AutoTunning.IFT.AnalogSP",AnalogParameter);
      StringSP = AnalogParameter+"ProcessOutput.MPosR";
      //Write and send desired SP value
      dpSet(StringSP, SP_Init+RStep);
      unGenericObject_SendPulse(AnalogParameter, ".ProcessOutput.ManReg01",
      CPC_ManReg01_MNEWMR, exceptionInfo, alreadySent);
    }
  }
  else
  {
    Exp_OK = FALSE;
  }

  dynClear(yExp1);
  for (i=1; i<=N; i++)
  {
```

```
      dpGet(deviceName + ".AutoTunning.IFT.Active",enable_IFT);
      if (enable_IFT == FALSE) { break; }
      dpGet (deviceName + ".ProcessInput.MV", y);
      dpGet (deviceName+".ProcessInput.OutOVSt", u);
      yExp1[i] = 100*(y-ySteady_In)/ySteady_In;
      uExp1[i] = 100*(u-uSteady_In)/uSteady_In;

      if (u>Out_max || u<Out_min)
      {
        DebugTN("Manipulated Variable out of Limits");
        dpSet(deviceName + ".AutoTunning.IFT.Active", FALSE);
        enable_IFT = FALSE;
        break;
      }

      delay(Samp_Time);
    }

    if  (i>N)
    {
      Exp_OK = TRUE;

  /*-------------------------------------------------------------------------------*/
  /*                                Cost function                                  */
  /*-------------------------------------------------------------------------------*/

      dynClear(yhat);
      for (i=1; i<=N; i++)  // yHat vector -> error
      {
        yhat[i] = (yExp1[i]-yd[i]);
      }

      J=0;
      for (i=1; i<=N; i++)
      {
         J = J + (pow(yhat[i],2) + Lambda*pow(uExp1[i],2))/(2*N);
      }
      DebugTN("Cost (J): ", J);

      // If J(i)>J(i-1) -> END IFT
      if (J<J_ant || iter==0 ) { J_ant = J; }
      else                     { End_Opt = TRUE; }

    }
    else
    {
      Exp_OK = FALSE;
    }

  /*-------------------------------------------------------------------------------*/
    iter++;

    if (iter >= nCycles+1)
    {
      DebugTN("иииииииииииииииии END OF AUTOTUNING иииииииииииииииии");
      dpSet(deviceName + ".AutoTunning.IFT.Active", FALSE);
      //dpGet(deviceName + ".AutoTunning.IFT.Active",enable_IFT);
```

```
      enable_IFT = FALSE;
    }
    else
    {
      DebugTN("Iteration: ", iter);
      DebugTN("nCycles", nCycles);
    }

  /*-------------------------------------------------------------------------------*/
  /*                                Experiment 2                                   */
  /*-------------------------------------------------------------------------------*/


    dpGet(deviceName + ".AutoTunning.IFT.Active", enable_IFT);
    if (enable_IFT==TRUE && End_Opt==FALSE && Exp_OK==TRUE)
    {
      DebugTN("иииииииииииииииии Second experiment иииииииииииииииии");
      dynClear(yExp2);
      for (i=1; i<=N; i++)
      {
        // Write SP value
        R = RStep-ySteady_In*yExp1[i]/100;

        if ((SP_Init+R)>SP_max || (SP_Init+R)<SP_min)
        {
          DebugTN("SP out of limits");
          dpSet(deviceName + ".AutoTunning.IFT.Active", FALSE);
          Exp_OK==FALSE;
        }
        else
        {
          // SP from controller
          if(EnableSP)
          {
            StringSP = deviceName+".ProcessOutput.MSP";
            //Write and send desired SP value
            dpSet(StringSP, SP_Init+R);
            unGenericObject_SendPulse(deviceName, ".ProcessOutput.ManReg01",
            CPC_ManReg01_MNEWSPR, exceptionInfo, alreadySent);
          }
          // SP from analog parameter
          else
          {
            dpGet(deviceName+".AutoTunning.IFT.AnalogSP",AnalogParameter);
            StringSP = AnalogParameter+"ProcessOutput.MPosR";
            //Write and send desired SP value
            dpSet(StringSP, SP_Init+R);
            unGenericObject_SendPulse(AnalogParameter, ".ProcessOutput.ManReg01",
            CPC_ManReg01_MNEWMR, exceptionInfo, alreadySent);
          }

          // Data adquisition
          dpGet (deviceName + ".ProcessInput.MV", y);
          yExp2[i] = 100*(y-ySteady_In)/ySteady_In;
          dpGet (deviceName+".ProcessInput.OutOVSt", u);
          uExp2[i] = 100*(u)/uSteady_In;
          if (Samp_Time>2) {delay(Samp_Time-2);}
```

```
        if (u>Out_max || u<Out_min)
        {
          DebugTN("Manipulated Variable out of Limits");
          dpSet(deviceName + ".AutoTunning.IFT.Active", FALSE);
          enable_IFT = FALSE;
          break;
        }

      }

      dpGet(deviceName + ".AutoTunning.IFT.Active",enable_IFT);
      if (enable_IFT == FALSE) { break; }
    }
    if (i>N)
    {
      Exp_OK = TRUE;
    }
    else
    {
      Exp_OK = FALSE;
    }
  } // If enable_IFT==TRUE


/*-------------------------------------------------------------------------------*/
/*                                Calculations                                   */
/*-------------------------------------------------------------------------------*/

  dpGet(deviceName + ".AutoTunning.IFT.Active",enable_IFT);
  if (enable_IFT==FALSE || Exp_OK==FALSE || End_Opt==TRUE)
  {
    dpSet(deviceName + ".AutoTunning.IFT.Active",FALSE);
    dpGet(deviceName + ".AutoTunning.IFT.Active",enable_IFT);
  }
  else
  {
  DebugTN("Calculations: ");

/*-------------------------------------------------------------------------------*/
/*              Calculate the partial derivative of PID function                 */
/*-------------------------------------------------------------------------------*/

    dynClear(Cye2);
    Cye2 = makeDynFloat(pow(Cy[1], 2), pow(Cy[2], 2), pow(Cy[3], 2));

    dynClear(numK);
    numK  = makeDynFloat(1.0);
    dynClear(denK);
    denK  = makeDynFloat(Cy[1], 0);
    dynClear(numTi);
    numTi = makeDynFloat(-Te2, -2*Te2, -Te2);
    dynClear(denTi);
    denTi = makeDynFloat(Te2*Cy[2]+2*T*Cye2[2]+4*Cy[3]*Cye2[2],
    2*Te2*Cy[2]-8*Cy[3]*Cye2[2], -2*T*Cye2[2]+Te2*Cy[2]+4*Cy[3]*Cye2[2]);

    if (Ny==3)
```

```
    {
      dynClear(numTd);
      numTd = makeDynFloat(4*Cy[2], -8*Cy[2], 4*Cy[2]);
      dynClear(denTd);
      denTd = makeDynFloat(Te2+2*Cy[2]*T+4*Cy[3]*Cy[2], -8*Cy[3]*Cy[2]+2*Te2,
      Te2+4*Cy[3]*Cy[2]-2*T*Cy[2]);
    }

    dynClear(dy);
    dy[1]= Filter(numK,  denK,  yExp2);
    dy[2]= Filter(numTi, denTi, yExp2);

    if (Ny==3)
    {
      dy[3]= Filter(numTd, denTd, yExp2);
    }

    dynClear(du);
    du[1]= Filter(numK,  denK,  uExp2);
    du[2]= Filter(numTi, denTi, uExp2);
    if (Ny==3)
    {
      du[3]= Filter(numTd, denTd, uExp2);
    }


/*-------------------------------------------------------------------------------*/
/*                  Calculate the derivative of cost function                    */
/*-------------------------------------------------------------------------------*/

    // dJ vector
    dynClear(dJ);
    dJ=makeDynFloat(0,0,0);
    for (i=1; i<=Ny; i++)
    { for (j=1; j<=N;j++)
      {
        dJ[i] = dJ[i] + (yhat[j]*dy[i][j] + Lambda*uExp1[j]*du[i][j])/N ;
      }
    }

    // Hessian matrix
    dynClear(Hessian);
    for (i=1; i<=Ny; i++)
    { for (j=1; j<=Ny; j++)
      { for (k=1; k<=N; k++)
        {
          Hessian[i][j] = Hessian[i][j] + (dy[i][k]*dy[j][k] +
          Lambda*du[i][k]*du[j][k])/N;
        }
      }
    }
    DebugTN("Hessian: ", Hessian);


    dynClear(Inv_Hessian);
    det_Hessian = CalcDeterminant(Hessian, Ny);
```

```
    if (det_Hessian==0)
    {
      DebugTN("Ill-conditoned");
      Inv_Hessian = Ident;
    }
    else
    {
      Inv_Hessian = MatrixInversion (Hessian, Ny);
    }

    DebugTN("Inv_Hessian: ", Inv_Hessian);


/*-------------------------------------------------------------------------------*/
/*                       Calculate the new PID parameters                        */
/*-------------------------------------------------------------------------------*/

    Cy_Ant = Cy;
    k=1;

    do
    {
      Cy = Cy_Ant;

      for (i=1;i<=Ny;i++)
      {
        for (j=1;j<=Ny;j++)
        {
          Cy[i] = Cy[i] - Gamma*Inv_Hessian[i][j]*dJ[j];
        }
      }

      if (Ny==3)
      {
        if (Cy[3]<0) { Cy[3]=0; }
        Cy[4] = Cy[3]/10;
      }
      DebugTN("Cy: ", Cy);

      if (Cy_Ant[1]!=0)  { Inc_Cy[1]=fabs(Cy[1]-Cy_Ant[1]); }
      else               { Inc_Cy[1]=0; }
      if (Cy_Ant[2]!=0)  { Inc_Cy[2]=fabs(Cy[2]-Cy_Ant[2]); }
      else               { Inc_Cy[2]=0; }
      if (Cy_Ant[3]!=0)  { Inc_Cy[3]=fabs(Cy[3]-Cy_Ant[3]); }
      else               { Inc_Cy[3]=0; }

      k++;
      Gamma = Gamma/2;

//     } while (k<=10 && (Cy[1]<0 || Cy[2]<0 || Inc_Cy[1]>25 || Inc_Cy[2]>25 ||
Inc_Cy[3]>10));
    } while (k<=10 && (Cy[1]<0 || Cy[2]<0));

    if (k>10)
    {
      ERROR = TRUE;
      dpSet(deviceName + ".AutoTunning.IFT.Active",FALSE);
```

```
    }
    else
    {
      dpSet(deviceName+".ProcessOutput.MKc", Cy[1]);
      unGenericObject_SendPulse(deviceName, ".ProcessOutput.ManReg01",
      CPC_ManReg01_MNEWKPR, exceptionInfo, alreadySent);
      dpSet(deviceName+".ProcessOutput.MTi", Cy[2]);
      unGenericObject_SendPulse(deviceName, ".ProcessOutput.ManReg01",
      CPC_ManReg01_MNEWTIR, exceptionInfo, alreadySent);
      dpSet(deviceName+".ProcessOutput.MTd", Cy[3]);
      unGenericObject_SendPulse(deviceName, ".ProcessOutput.ManReg01",
      CPC_ManReg01_MNEWTDR, exceptionInfo, alreadySent);
      dpSet(deviceName+".ProcessOutput.MTds", Cy[4]);
      unGenericObject_SendPulse(deviceName, ".ProcessOutput.ManReg01",
      CPC_ManReg01_MNEWKDR, exceptionInfo, alreadySent);
    }

    } // if Exp_OK && IFT_ON...
  } // while enable_IFT...


/*-------------------------------------------------------------------------------*/
/*                              End of Autotuning                                */
/*-------------------------------------------------------------------------------*/

  if (Exp_OK==FALSE || End_Opt==TRUE)
  {
    if (Exp_OK==FALSE) {DebugTN("Failed experiments: END OF AUTOTUNING");}
    else               {DebugTN("End of optimization: END OF AUTOTUNING");}

    dpSet(deviceName+".ProcessOutput.MKc", Cy_Ant[1]);
    unGenericObject_SendPulse(deviceName, ".ProcessOutput.ManReg01",
    CPC_ManReg01_MNEWKPR, exceptionInfo, alreadySent);
    dpSet(deviceName+".ProcessOutput.MTi", Cy_Ant[2]);
    unGenericObject_SendPulse(deviceName, ".ProcessOutput.ManReg01",
    CPC_ManReg01_MNEWTIR, exceptionInfo, alreadySent);
    dpSet(deviceName+".ProcessOutput.MTd", Cy_Ant[3]);
    unGenericObject_SendPulse(deviceName, ".ProcessOutput.ManReg01",
    CPC_ManReg01_MNEWTDR, exceptionInfo, alreadySent);
    dpSet(deviceName+".ProcessOutput.MTds", Cy_Ant[4]);
    unGenericObject_SendPulse(deviceName, ".ProcessOutput.ManReg01",
    CPC_ManReg01_MNEWKDR, exceptionInfo, alreadySent);
  }

  if (ERROR==TRUE)
  {
    DebugTN("Error: END OF AUTOTUNING");

    dpSet(deviceName+".ProcessOutput.MKc", Cy_Init[1]);
    unGenericObject_SendPulse(deviceName, ".ProcessOutput.ManReg01",
    CPC_ManReg01_MNEWKPR, exceptionInfo, alreadySent);
    dpSet(deviceName+".ProcessOutput.MTi", Cy_Init[2]);
    unGenericObject_SendPulse(deviceName, ".ProcessOutput.ManReg01",
    CPC_ManReg01_MNEWTIR, exceptionInfo, alreadySent);
    dpSet(deviceName+".ProcessOutput.MTd", Cy_Init[3]);
    unGenericObject_SendPulse(deviceName, ".ProcessOutput.ManReg01",
    CPC_ManReg01_MNEWTDR, exceptionInfo, alreadySent);
```

```
      dpSet(deviceName+".ProcessOutput.MTds", Cy_Init[4]);
      unGenericObject_SendPulse(deviceName, ".ProcessOutput.ManReg01",
      CPC_ManReg01_MNEWKDR, exceptionInfo, alreadySent);
    }

    //Activate Manual Mode
    unGenericObject_SendPulse(deviceName, ".ProcessOutput.ManReg01",
    CPC_ManReg01_MMMOR, exceptionInfo, alreadySent);

    //Activate Regulation
    unGenericObject_SendPulse(deviceName, ".ProcessOutput.ManReg02",
    CPC_ManReg02_MREGR, exceptionInfo, alreadySent);

    //Write and send initial SP
    if(EnableSP)
    {
      StringSP = deviceName+".ProcessOutput.MSP";
      //Write and send desired SP value
      dpSet(StringSP, SP_Init);
      unGenericObject_SendPulse(deviceName, ".ProcessOutput.ManReg01",
      CPC_ManReg01_MNEWSPR, exceptionInfo, alreadySent);
    }
    else
    {
      dpGet(deviceName+".AutoTunning.IFT.AnalogSP",AnalogParameter);
      StringSP = AnalogParameter+"ProcessOutput.MPosR";
      //Write and send desired SP value
      dpSet(StringSP, SP_Init);
      unGenericObject_SendPulse(AnalogParameter, ".ProcessOutput.ManReg01",
      CPC_ManReg01_MNEWMR, exceptionInfo, alreadySent);
    }

  } // function

  /*-------------------------------------------------------------------------------*/
```

```
/*****************************************************************************/
/********************************** RELAY ************************************/
/*****************************************************************************/
/* - Steps:                                                                  */
/*      - Detecting Steady State                                             */
/*      - Calculate Hystereis Parameters                                     */
/*      - Hysteresis Section                                                 */
/*      - Calculate New PID Parameters                                       */
/*      - End of Autotuning                                                  */
/*****************************************************************************/


RELAY(string deviceName, float dmax, float dmin, int nCycles, bool signe)
{
  DebugTN("sign: ", signe);

  dyn_string exceptionInfo;
  bool alreadySent;

  bool Active_Method;
  bool x1 = TRUE;
  bool x2 = FALSE;
  float epsilon = 0;        //Output amplitude
  float SP=0, MV=0, error=0;
  float Uo, Ux, max=0, min=1000;

  bool EnTim = FALSE;
  time t1, t2;
  int Time_1, Time_2, ms1, ms2;
  float Period;

  bool EnCount = TRUE;
  int Count = 0;

  float P_K,P_Ti,P_Td, P_Tds;

  dyn_float Y;

  float MVMax,MVMin;

  float Ny; //Structure Controller (PID - PI)

/*---------------------------------------------------------------------------*/

  //Initializing variables
  dpGet (deviceName+".ProcessInput.OutOVSt", Uo);
  dpGet (deviceName+".ProcessInput.ActSP", SP);
  dpGet (deviceName+".AutoTunning.Limits.SP_Max", MVMax);
  dpGet (deviceName+".AutoTunning.Limits.SP_Min", MVMin);

/*---------------------------------------------------------------------------*/

  //Activate Manual Mode
  unGenericObject_SendPulse(deviceName, ".ProcessOutput.ManReg01",
  CPC_ManReg01_MMMOR, exceptionInfo, alreadySent);

  //Write desired output value
```

```
  dpSet(deviceName+".ProcessOutput.MPosR", Uo);
  //Activate writting
  unGenericObject_SendPulse(deviceName, ".ProcessOutput.ManReg02",
  CPC_ManReg02_MNEWPOSR, exceptionInfo, alreadySent);

  //Activate Out Positioning
  unGenericObject_SendPulse(deviceName, ".ProcessOutput.ManReg02",
  CPC_ManReg02_MOUTPR, exceptionInfo, alreadySent);


/*-----------------------------------------------------------------------------*/
/*                              Detecting Steady State                         */
/*-----------------------------------------------------------------------------*/

  //SSID FILTER Parameters
  float l1 = 0.2;
  float l2 = 0.1;
  float l3 = 0.1;

  float cl1 = 1 - l1;
  float cl2 = 1 - l2;
  float cl3 = 1 - l3;


  float yf = 0;
  float nu2f = 0;
  float delta2f = 0;
  float y_old = 0;
  float ID_Filter = 0.5;
  float R_Filter;
  bool SteadyState;
  float y;
  dyn_float yExp1,YExp;
  float Samp_Time = 2;

/*-----------------------------------------------------------------------------*/

  //Indentify the steady state to change U
  SteadyState = FALSE;
  dpGet(deviceName+".AutoTunning.Relay.Active", Active_Method);
  int i = 1;
  float y_Steady;
  dpGet (deviceName + ".ProcessInput.MV", y_Steady);
  delay(Samp_Time);
  DebugTN("SteadyState", SteadyState);
  DebugTN("Active_Method", Active_Method);

  while(SteadyState==FALSE && Active_Method==TRUE)
  {
    dpGet(deviceName+".AutoTunning.Relay.Active", Active_Method);
    DebugTN("SteadyState", SteadyState);
    DebugTN("Active_Method", Active_Method);
    dpGet (deviceName + ".ProcessInput.MV", y);
    if (y!=y_Steady)
    {
      yExp1[i] = (y-y_Steady)/y_Steady;
      YExp[i] = y;
```

```
        delay(Samp_Time);

        //SSID FILTER
        nu2f = l2*pow(( yExp1[i]-yf),2) + cl2*nu2f;
        yf = l1* yExp1[i] + cl1*yf;
        delta2f = l3*pow((yExp1[i]-y_old),2) +cl3*delta2f;
        y_old = yExp1[i];
        R_Filter = (2-l1)*nu2f/delta2f;
        if (R_Filter > 2.5)
        {
            ID_Filter = 0;
        }
        else
        if(R_Filter < 1 && i>1)
        {
          ID_Filter = 1;
        }
        else
        {
          if(i>1)
          {
            ID_Filter = ID_Filter;
          }
         }

        if(ID_Filter == 1)
        {
          SteadyState = TRUE;
        }
        i++;
      }
    }


/*-------------------------------------------------------------------------------*/
/*                          Calculate Hysteresis Parameters                      */
/*-------------------------------------------------------------------------------*/

    // Calculate hysteresis parameters
    dpGet(deviceName+".AutoTunning.Relay.Active", Active_Method);
    if(Active_Method)
    {
      int j=i;
      float max_noise=YExp[i-1], min_noise=YExp[i-1];
      while (i>=2 && i>=(j-10))
      {
        i--;
        if (YExp[i]>max_noise)
        {
          max_noise=YExp[i];
        }
        if (YExp[i]<min_noise)
        {
          min_noise=YExp[i];
        }
      }
      epsilon=3*(max_noise-min_noise);
```

```
    DebugTN("epsilon; ",epsilon);
  }


/*-------------------------------------------------------------------------------*/
/*                                    RELAY                                      */
/*-------------------------------------------------------------------------------*/

  int aux = 1, aux_1=1;
  dyn_float Y;
  do{

    dpGet(deviceName+".AutoTunning.Relay.Active",Active_Method);
    dpGet (deviceName+".ProcessInput.MV", MV);
    error = MV - SP;
    aux = aux+1;
    //Actual oputput value
    dpGet (deviceName+".ProcessInput.OutOVSt", Ux);

    //Rising Event
    if (Ux <= (dmin))
    {
      EnCount = TRUE;
    }


/*-------------------------------------------------------------------------------*/
/*                              Hysteresis Section                               */
/*-------------------------------------------------------------------------------*/

     DebugTN("Error:",error);
    //Positive zone
    if  (error >= epsilon)
    {
        //Write desired output value
        if(signe)
        {
          dpSet(deviceName+".ProcessOutput.MPosR", dmin);
        }
        else
        {
          dpSet(deviceName+".ProcessOutput.MPosR", dmax);
        }
        //Activate writting
        unGenericObject_SendPulse(deviceName, ".ProcessOutput.ManReg02",
        CPC_ManReg02_MNEWPOSR, exceptionInfo, alreadySent);
        x1=1;
        x2=0;
    }

    //Negative zone
    else
      if  (error <= (-epsilon))
      {
          //Write desired output value
          if(signe)
          {
```

```
          dpSet(deviceName+".ProcessOutput.MPosR", dmax);
        }
        else
        {
          dpSet(deviceName+".ProcessOutput.MPosR", dmin);
        }
        //Activate writting
        unGenericObject_SendPulse(deviceName, ".ProcessOutput.ManReg02",
        CPC_ManReg02_MNEWPOSR, exceptionInfo, alreadySent);
        x1=0;
        x2=1;
    }

    //Intermediate zone
    else
     if (error>((-1)*epsilon) && error<epsilon)
      {
        if (x1==1)
        {
          //Write desired output value
          if(signe)
          {
            dpSet(deviceName+".ProcessOutput.MPosR", dmin);
          }
          else
          {
            dpSet(deviceName+".ProcessOutput.MPosR", dmax);
          }
        //Activate writting
        unGenericObject_SendPulse(deviceName, ".ProcessOutput.ManReg02",
        CPC_ManReg02_MNEWPOSR, exceptionInfo, alreadySent);
        }
        else if (x2==1)
        {
        //Write desired output value
        if(signe)
        {
          dpSet(deviceName+".ProcessOutput.MPosR", dmax);
        }
        else
        {
          dpSet(deviceName+".ProcessOutput.MPosR", dmin);
        }
        //Activate writting
        unGenericObject_SendPulse(deviceName, ".ProcessOutput.ManReg02",
        CPC_ManReg02_MNEWPOSR, exceptionInfo, alreadySent);
        }
      }

dpGet (deviceName+".ProcessInput.OutOVSt", Ux);

if (Ux>=Uo && EnCount==TRUE)
{
  EnCount = FALSE;
  Count ++;
  DebugTN ("Contador: ", Count);
}
```

```
    if (Count==nCycles)
    {
      if (EnTim==FALSE)
      {
        EnTim = TRUE;
        //Countdown
        t1 = getCurrentTime();
      }
      Y[aux_1] = MV;
      aux_1++;
    }

    //Maximun and Minimum Alarm
    if(MV>MVMax | MV<MVMin)
    {
      dpSet(deviceName+".AutoTunning.Relay.Active", FALSE);
    }

  }while(Count<=nCycles & Active_Method);


  //Finish Countdown
  t2 = getCurrentTime();

  //Write desired output value
  dpSet(deviceName+".ProcessOutput.MPosR", Uo);
  //Activate writting
  unGenericObject_SendPulse(deviceName, ".ProcessOutput.ManReg02",
  CPC_ManReg02_MNEWPOSR, exceptionInfo, alreadySent);



/*-------------------------------------------------------------------------------*/
/*                       Calculate the new PID parameters                        */
/*-------------------------------------------------------------------------------*/

  Time_1 = period(t1);
  DebugTN ("Tiempo: ", t1);
  Time_2 = period(t2);
  DebugTN ("Tiempo: ", t2);

  ms1 = milliSecond(t1);
  ms2 = milliSecond(t2);

  Period = Time_2-Time_1 + 0.001*(ms2-ms1);

  DebugTN ("Period(t): ", Period);

  float a;
  a = 0.5*(dynMax(Y)-dynMin(Y));
  DebugTN ("Max (a): ", a);
  dpGet(deviceName+".AutoTunning.Relay.Active",Active_Method);
//   Active_Method = TRUE;
  if (NumParam_Controller.text()=="PI")       { Ny=2; }
  else if (NumParam_Controller.text()=="PID") { Ny=3; }
  if(Active_Method & Ny == 2)
```

```
    {
      // PID parameters Calculatation
      P_Ti = Period/1.2;
      P_K = 0.45*(2*(dmax-dmin))/(3.1416*pow((pow(a, 2) - pow(epsilon, 2)), 0.5));
      DebugTN ("Proposed(K): ", P_K);
      DebugTN ("Proposed(Ti): ", P_Ti);

      dpSet(deviceName+".AutoTunning.KNew", P_K);
      dpSet(deviceName+".AutoTunning.TiNew", P_Ti);
      dpSet(deviceName+".AutoTunning.TdNew", 0);
      dpSet(deviceName+".AutoTunning.TdsNew", 0);
    }
    else
    if(Active_Method & Ny == 3)
    {
       // PID parameters Calculatation
      P_Ti = Period/2.0;
      P_K = 0.6*(2*(dmax-dmin))/(3.1416*pow((pow(a, 2) - pow(epsilon, 2)), 0.5));
      P_Td = Period/8;
      P_Tds = P_Td/10;

      DebugTN ("Proposed(K): ", P_K);
      DebugTN ("Proposed(Ti): ", P_Ti);
      DebugTN ("Proposed(Td): ", P_Td);
      DebugTN ("Proposed(Tds): ", P_Tds);

      dpSet(deviceName+".AutoTunning.KNew", P_K);
      dpSet(deviceName+".AutoTunning.TiNew", P_Ti);
      dpSet(deviceName+".AutoTunning.TdNew", P_Td);
      dpSet(deviceName+".AutoTunning.TdsNew", P_Tds);
    }

  /*-------------------------------------------------------------------------------*/
  /*                              End of Autotuning                                */
  /*-------------------------------------------------------------------------------*/

    //Activate Manual Mode
    unGenericObject_SendPulse(deviceName, ".ProcessOutput.ManReg01",
    CPC_ManReg01_MMMOR, exceptionInfo, alreadySent);

    //Activate Regulation
    unGenericObject_SendPulse(deviceName, ".ProcessOutput.ManReg02",
    CPC_ManReg02_MREGR, exceptionInfo, alreadySent);


  }//end function


  /*-------------------------------------------------------------------------------*/
```

```
/*****************************************************************************/
/******************************** IDENTIFICATION *****************************/
/*****************************************************************************/
/* - Steps:                                                                  */
/*      - Detecting Steady State                                             */
/*      - STEP INPUT                                                         */
/*      - Calculate First Order plus Delay Parameters                        */
/*      - End of Autotuning                                                  */
/*****************************************************************************/


IDENTIFICATION(string deviceName, float DeltaU, float Samp_Time)
{
  DebugTN("Samp_Time: ", Samp_Time);
  dyn_string exceptionInfo;
  bool alreadySent;
  dyn_float yExp1, yExp2, yExp, TimeExp2;
  float y, u, Uo, Time0;
  int i,j; //Auxiliar variables
  bool SteadyState;
  time TimeAux;

  float MVMax,MVMin;

  bool Active_Method;


/*---------------------------------------------------------------------------*/
/*                            Detecting Steady State                         */
/*---------------------------------------------------------------------------*/

  //SSID FILTER Parameters
  float l1 = 0.2;
  float l2 = 0.1;
  float l3 = 0.1;

  float cl1 = 1 - l1;
  float cl2 = 1 - l2;
  float cl3 = 1 - l3;


  float yf = 0;
  float nu2f = 0;
  float delta2f = 0;
  float y_old = 0;
  float ID_Filter = 0.5;
  float R_Filter;

  Uo = 0;
  //User data - Change on U
  float Ustep = DeltaU;

  //Indentified Pant Parameters
  dyn_float Kp;
  dyn_float Theta;
  dyn_float Tau;
```

```
/*-----------------------------------------------------------------------------*/

  //Initializing variables
  dpGet (deviceName+".ProcessInput.OutOVSt", Uo);
  dpGet (deviceName+".AutoTunning.Limits.SP_Max", MVMax);
  dpGet (deviceName+".AutoTunning.Limits.SP_Min", MVMin);

/*-----------------------------------------------------------------------------*/

  //Activate Manual Mode
  unGenericObject_SendPulse(deviceName, ".ProcessOutput.ManReg01",
  CPC_ManReg01_MMMOR, exceptionInfo, alreadySent);

  //Activate Out Positioning
  unGenericObject_SendPulse(deviceName, ".ProcessOutput.ManReg02",
  CPC_ManReg02_MOUTPR, exceptionInfo, alreadySent);

/*-----------------------------------------------------------------------------*/


  //Write desired output value
  dpSet(deviceName+".ProcessOutput.MPosR", Uo);
  //Activate writting
  unGenericObject_SendPulse(deviceName, ".ProcessOutput.ManReg02",
  CPC_ManReg02_MNEWPOSR, exceptionInfo, alreadySent);

  DebugTN("Manipulated Variable:",Uo);
  //Indentify the steady state to change U
  SteadyState = FALSE;
  dpGet(deviceName+".AutoTunning.SIMC.Active", Active_Method);
  i = 1;
  float y_Steady;
  dpGet (deviceName + ".ProcessInput.MV", y_Steady);
  y_Steady=y_Steady+0.000001*rand();
  delay(Samp_Time);

  while(!SteadyState & Active_Method)
  {
    dpGet(deviceName+".AutoTunning.SIMC.Active", Active_Method);
    dpGet (deviceName + ".ProcessInput.MV", y);
    yExp1[i] = (y-y_Steady)/y_Steady;
    delay(Samp_Time);

    //SSID FILTER
    nu2f = l2*pow(( yExp1[i]-yf),2) + cl2*nu2f;
    yf = l1* yExp1[i] + cl1*yf;
    delta2f = l3*pow((yExp1[i]-y_old),2) +cl3*delta2f;
    y_old = yExp1[i];
    R_Filter = (2-l1)*nu2f/delta2f;
    DebugTN("R_Filter:",R_Filter);

    if (R_Filter > 2.5)
    {
        ID_Filter = 0;
    }
    else
```

```
    if(R_Filter < 1)
    {
      ID_Filter = 1;
    }
    else
    {
      if(i>1)
      {
        ID_Filter = ID_Filter;
      }
     }

    if(ID_Filter == 1)
    {
      SteadyState = TRUE;
    }
    i++;
  }

/*-------------------------------------------------------------------------------*/
/*                                  STEP INPUT                                   */
/*-------------------------------------------------------------------------------*/

  DebugTN("Steady State:",SteadyState);
  DebugTN("*****");
  DebugTN("STEP");
  DebugTN("*****");


  //Change on Manipulated Variable
  //Write desired output value
  DebugTN("STEP VALUE:", Uo+Ustep);
  //Write desired output value
  dpSet(deviceName+".ProcessOutput.MPosR", Uo+Ustep);
  //Activate writting
  unGenericObject_SendPulse(deviceName, ".ProcessOutput.ManReg02",
  CPC_ManReg02_MNEWPOSR, exceptionInfo, alreadySent);
  dpGet(deviceName+".AutoTunning.SIMC.Active", Active_Method);



/*-------------------------------------------------------------------------------*/
/*                             Detecting Steady State                            */
/*-------------------------------------------------------------------------------*/

  //Recollect Data until the new Steady State
  //Indentify the steady state to change U
  yf = 0;
  nu2f = 0;
  delta2f = 0;
  y_old = 0;
  ID_Filter = 0.5;
  SteadyState = FALSE;
  i = 1;

/*-------------------------------------------------------------------------------*/
```

```
  while(!SteadyState & Active_Method)
  {
    dpGet(deviceName+".AutoTunning.SIMC.Active", Active_Method);
    dpGet (deviceName + ".ProcessInput.MV", y);
//     yExp2[i] = (y-y_Steady)/y_Steady+rand();
    yExp2[i] = (y-y_Steady)/y_Steady;
    yExp[i] = y;
    TimeAux = getCurrentTime();
    TimeExp2[i] = period(TimeAux)+ 0.001*milliSecond(TimeAux);
    if (i==1)
    {
      Time0=TimeExp2[i];
    }
    TimeExp2[i] = TimeExp2[i] - Time0;

    delay(Samp_Time);

    //SSID FILTER
    nu2f = l2*pow((yExp2[i]-yf),2) + cl2*nu2f;
    yf = l1*yExp2[i] + cl1*yf;
    delta2f = l3*pow((yExp2[i]-y_old),2) +cl3*delta2f;
    y_old = yExp2[i];
    R_Filter = (2-l1)*nu2f/delta2f;
    DebugTN("R_Filter:",R_Filter);
    if (R_Filter > 2.5)
    {
        ID_Filter = 0;
    }
    else
    if(R_Filter < 1)
    {
      ID_Filter = 1;
    }
    else
    {
      if(i>1)
      {
        ID_Filter = ID_Filter;
      }
     }

    if(ID_Filter == 1)
    {
      SteadyState = TRUE;
    }
    i++;

    //Maximun and Minimum Alarm
    if(y>MVMax | y<MVMin)
    {
      DebugTN("End Method: Out of Limits");
      dpSet(deviceName+".AutoTunning.SIMC.Active", FALSE);
    }
  }

  dpGet(deviceName+".AutoTunning.SIMC.Active", Active_Method);
  if(Active_Method)
```

```
  {


/*-------------------------------------------------------------------------------*/
/*                  Calculate First Order plus Delay Parameters                  */
/*-------------------------------------------------------------------------------*/

    //Time at 28.3% Final Steady State
    int N_Sampling = dynlen(yExp);
    float YSteady = yExp[N_Sampling-1];

    float YTau1 = 0.283*(YSteady-yExp[1])+yExp[1];
    float YTau2 = 0.632*(YSteady-yExp[1])+yExp[1];

    DebugTN("YTau1:",YTau1);
    DebugTN("YTau2:",YTau2);

    dyn_float TimeTau1,TimeTau2;
    TimeTau1 = Lagrange_interp_Inv(TimeExp2,yExp,YTau1);
    DebugTN("TimeTau1:",TimeTau1);
    TimeTau2 = Lagrange_interp_Inv(TimeExp2,yExp,YTau2);
    DebugTN("TimeTau2:",TimeTau2);

    //MODEL
    DebugTN("FOPD MODEL");
    int Num_Sol_1 = dynlen(TimeTau1);
    int Num_Sol_2 = dynlen(TimeTau2);

    Kp[1] = (yExp[N_Sampling-1]-yExp[1])/(Ustep); // Have to be %/%
    Tau[1] = 1.5*(TimeTau2[Num_Sol_2] - TimeTau1[1]); // Seconds
    if(TimeTau2[Num_Sol_2] - Tau[1] >0 && TimeTau2[Num_Sol_2] - Tau[1]>Samp_Time)
    {
      Theta[1] = (TimeTau2[Num_Sol_2] - Tau[1]); // Seconds
    }
    else
    {
      Theta[1] = Samp_Time;
    }
    DebugTN("Gain;",Kp);
    dpSet(deviceName+".AutoTunning.Kp", fabs(Kp[1]));
    DebugTN("Theta;",Theta);
    dpSet(deviceName+".AutoTunning.Theta", Theta[1]);
    DebugTN("Tau;",Tau);
    dpSet(deviceName+".AutoTunning.Tau", Tau[1]);
 }



/*-------------------------------------------------------------------------------*/
/*                               End of Autotuning                               */
/*-------------------------------------------------------------------------------*/

  DebugTN("END METHOD");
  //Activate Manual Mode
  unGenericObject_SendPulse(deviceName, ".ProcessOutput.ManReg01",
  CPC_ManReg01_MMMOR, exceptionInfo, alreadySent);
```

```
    //Activate Regulation
    unGenericObject_SendPulse(deviceName, ".ProcessOutput.ManReg02",
    CPC_ManReg02_MREGR, exceptionInfo, alreadySent);
  }
```

```
/**************************************************************************************/
/************************************ s-IMC ***********************************/
/**************************************************************************************/
/* - Steps:                                                                           */
/*      - Calculate the new PID parameters                                            */
/**************************************************************************************/

SIMC()
{
  float K, Tau,Theta;
  string deviceName = unGenericDpFunctions_getDpName($sDpName);

  dpGet(deviceName + ".AutoTunning.Kp", K);
  dpGet(deviceName + ".AutoTunning.Theta", Theta);
  dpGet(deviceName + ".AutoTunning.Tau", Tau);

  float TuneParameter = 0;
  float TuneParameter_User = 0;
  float TuneParameter_min = 0;
  float TuneParameter_max = 0;

  TuneParameter_min = -2*Theta;
  TuneParameter_max = 2*Theta;

  dpGet(deviceName + ".AutoTunning.SIMC.TuneParameters",TuneParameter_User);
  TuneParameter = TuneParameter_User*(TuneParameter_max-TuneParameter_min)/100;

  float Kc = 0;
  float Ti = 0;
  float Ti_aux1 = 0;
  float Ti_aux2 = 0;


/*-------------------------------------------------------------------------------*/
/*                       Calculate the new PID parameters                        */
/*-------------------------------------------------------------------------------*/


  if(K>0)
  {
    Kc = (1/K)*(Tau/(TuneParameter+Theta));
    setValue("KNew", "text", Kc);

    Ti_aux1 = Tau;
    Ti_aux2 = 4*(TuneParameter+Theta);
    if(Ti_aux1>Ti_aux2)
    {
      Ti = Ti_aux2;
    }
    else
    {
      Ti = Ti_aux1;
    }
    setValue("TiNew", "text", Ti);
  }
  else
  {
```

```
            setValue("KNew", "text", "--");
            setValue("TiNew", "text", "--");
          }
        }
```

```
/*****************************************************************************/
/****************************** Auxiliar Functions*****************************/
/*****************************************************************************/



/*****************************************************************************/
/* Matrix Functions **********************************************************/
/*****************************************************************************/
/* - Description: Calculate recursively the determinant and the inverse matrix  */
/* - Functions:                                                                 */
/*      double CalcDeterminant( dyn_dyn_float mat, int order)                   */
/*      dyn_dyn_float GetMinor(dyn_dyn_float src, int row, int col, int order)  */
/*      dyn_dyn_float MatrixInversion(dyn_dyn_float A, int order)               */
/*****************************************************************************/

// Calculate the matrix
dyn_dyn_float MatrixInversion(dyn_dyn_float A, int order)
{
    // get the determinant of a
    dyn_dyn_float Y;
    double det = 1.0/CalcDeterminant(A,order);

    // memory allocation
    dyn_float temp;
    dyn_dyn_float minor;

    for(int j=1;j<=order;j++)
    {
        for(int i=1;i<=order;i++)
        {
            // get the co-factor (matrix) of A(j,i)
            minor = GetMinor( A, j, i, order);
            Y[i][j] = det*CalcDeterminant(minor,order-1);
            if( (i+j)%2 == 1) { Y[i][j] = -Y[i][j]; }
        }
    }

    return Y;

    // release memory
    dynClear(temp);
    dynClear(minor);
}

/*---------------------------------------------------------------------------*/

// Calculate the minors
dyn_dyn_float GetMinor(dyn_dyn_float src, int row, int col, int order)
{
  // indicate which col and row is being copied to dest
  int colCount=1,rowCount=1;
  dyn_dyn_float dest;

  for(int i = 1; i <= order; i++ )
  {
      if( i != row )
```

```
        {
            colCount = 1;
            for(int j = 1; j <= order; j++ )
            {
                // when j is not the element
                if( j != col )
                {
                    dest[rowCount][colCount] = src[i][j];
                    colCount++;
                }
            }
            rowCount++;
        }
    }

    return dest;
}

/*---------------------------------------------------------------------------*/

// Calculate the determinant recursively.
double CalcDeterminant( dyn_dyn_float mat, int order)
{
    // order must be >= 0
    // stop the recursion when matrix is a single element
    if( order == 1 ) { return mat[1][1]; }

    // the determinant value
    float det = 0;

    // allocate the cofactor matrix
    dyn_dyn_float minor;

    for(int i = 1; i <= order; i++ )
    {
        // get minor of element (0,i)
        minor = GetMinor( mat,  1, i , order);
        // the recusion is here!
        det += pow( -1.0, (i-1) ) * mat[1][i] * CalcDeterminant( minor,order-1 );
    }

    // release memory
    dynClear(minor);

    return det;
}

/*---------------------------------------------------------------------------*/
```

```
/*****************************************************************************/
/* Discrete Filter ***********************************************************/
/*****************************************************************************/
/* - Description: Obtein the filtered data by a discrete transfer function   */
/*****************************************************************************/

dyn_float Filter (dyn_float Num_Filter, dyn_float Den_Filter, dyn_float vector)
{
  dyn_float x, y, yd;
  float sum_x=0, sum_y=0, a1=1;
  int Or_Num = dynlen(Num_Filter);
  int Or_Den = dynlen(Den_Filter);
  int i, j;

  if (Or_Den>1) { a1=Den_Filter[1]; }

  for (i=1; i<=dynlen(vector); i++)
  {
    for (j=1;j<=Or_Num;j++)
    {
      if   ((i-j)>=0)   { x[j]=Num_Filter[j]*vector[i-j+1]; }
      else              { x[j]=0.0; }
    }
    for (j=2;j<=Or_Den;j++)
    {
      if   ((i-j)>=0) { y[j]=Den_Filter[j]*yd[i-j+1]; }
      else            { y[j]=0.0; }
    }

    sum_x = dynSum(x);
    if (Or_Den>1) { sum_y = dynSum(y); }

    yd[i]= (sum_x-sum_y)/a1;
  }

  return(yd);

  // Release memory
  dynClear(x);
  dynClear(y);
  dynClear(yd);
}

/*---------------------------------------------------------------------------*/
```

```
/*****************************************************************************/
/* Interpolation *************************************************************/
/*****************************************************************************/

dyn_float Lagrange_interp_Inv(dyn_float x, dyn_float y, float y0)
{
 //Initialize outputs
 int N_Sampling = dynlen(x);
 dyn_float x0;
 dyn_float IDEXACT;
 dyn_float IDINTERP;
 int i;

 //Subtract target level to simplify subsequent code
 dyn_float Y;
 for(int i=1;i<=N_Sampling;i++)
 {
   Y[i] = y[i] - y0;
 }
 //Find exact values
 int counter1 = 0;
 for(i=1;i<=N_Sampling;i++)
 {
   if(Y[i]==0)
   {
     counter1++;
     IDEXACT[counter1] = i;
   }
 }
 //Find crossing
 int counter2 = 0;
 for(i=1;i<=N_Sampling-1;i++)
 {
   if(Y[i]*Y[i+1]<0)
   {
     counter2++;
     IDINTERP[counter2] = i;
   }
 }
 //Compusing the different solution
 if(dynlen(IDEXACT)>0)
 {
   for(i=1;i<=counter1;i++)
   {
     x0[i] = x[IDEXACT[i]];
   }
 }
 if(dynlen(IDINTERP)>0)
 {
   for(i=1;i<=counter2;i++)
   {
     x0[counter1+i] = x[IDINTERP[i]] +
     (x[IDINTERP[i]+1]-x[IDINTERP[i]])*Y[IDINTERP[i]]/(Y[IDINTERP[i]]-Y[IDINTERP[i]+1]
     );
   }
 }
```

```
 return x0;
}

/*------------------------------------------------------------------------------*/
```