Capítulo 4

Diseño de controladores empotrados jerárquicos

En este capítulo se muestran las ventajas de la implementación jerárquica de controladores y se explica una estructura básica para un controlador jerárquico. Estas ideas se aplican en la realización de un controlador para estabilizar en un punto de operación a un sistema altamente inestable: un doble integrador.

4.1. Diseño de controladores difusos jerárquicos

Una jerarquía es básicamente una interconexión en serie y/o paralelo de distintos módulos básicos. La interconexión de módulos básicos es más simple en cuanto al número de reglas que un sistema no jerárquico. El problema fundamental de los sistemas jerárquicos es que son más dificiles de diseñar, sin embargo *Xfuzzy 3* hace que esta tarea sea más sencilla por las siguientes razones:

a) *Xfuzzy 3* admite la definición de sistemas jerárquicos con módulos difuso y no difusos:

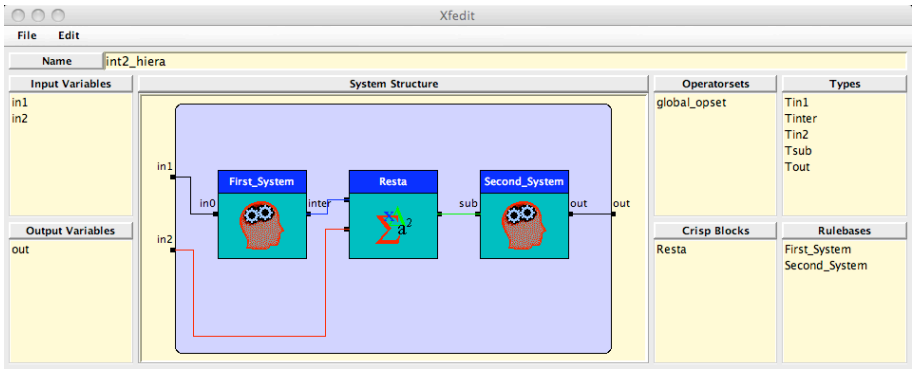


Figura 25. Definición de un sistema jerárquico usando módulos difusos: *First_System* y *Second_System* y de módulos no difusos: *Resta*

b) Xfuzzy permite ajustar sistemas jerárquicos a un conjunto dado de datos numéricos porque permite aplicar algoritmos de aprendizaje sin derivadas o algortimos con derivadas pero en los que la derivada se puede aproximar por la secante.

c) Por último, la herramienta de síntesis hardware Xfsg de Xfuzzy 3 permite hacer una implementación hardware automática de sistemas jerárquicos.

4.2. Estructura básica de un controlador jerárquico difuso - *PWA*

En esta sección se procede a explicar la estructura jerárquica que se va propone en este Proyecto. Se van a utilizar dos sistemas difusos, cada uno de ellos con una entrada que utilizan funciones de pertenencia triangulares solapadas al 50% y un método de defuzzificación *FuzzyMean* y una salida conectadas en serie y con un módulo no difuso, *Resta*, entre ellas.

Vemos el comportamiento de esta estructura mediante un ejemplo sencillo. Para ello, se considera un sistema con dos entradas, x_1 y x_2, y una salida y, como el que vemos en la *Figura 26*.

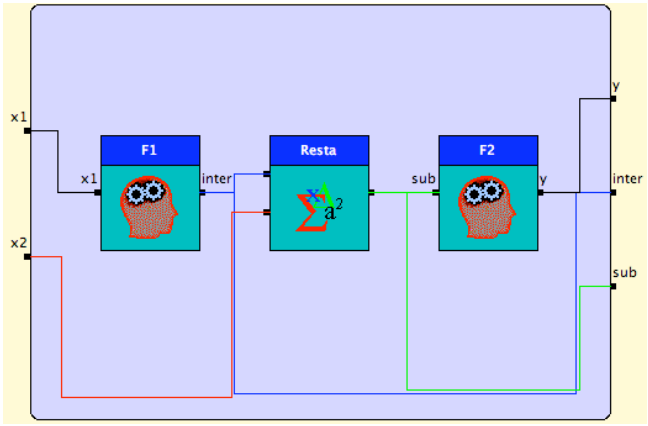
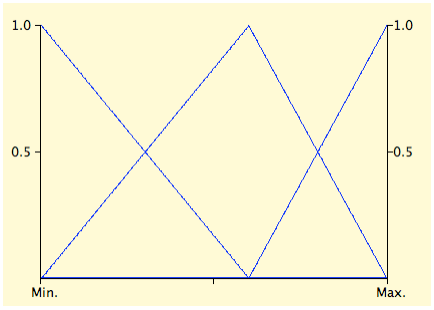
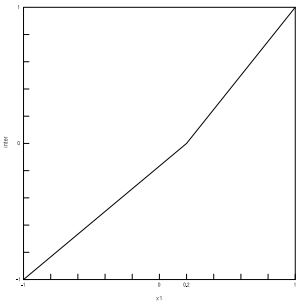


Figura 26. Ejemplo de un sistema jerárquico

En el ejemplo de la *Figura 26* el primer módulo emplea 3 funciones de pertenencia triangulares para cubrir la entrada x_1 (*Figura 27a*) e implementa la base de reglas mostrada en la *Figura 27b*. La partición de la *Figura 27a* distingue dos intervalos en la variable x_1: $[-1,0.2]$ y $[0.2,1]$ (como puede verse en la *Figura 27c)* para que los que la salida inter es lineal. En el primer intervalo, la recta depende de los valores de los consecuentes de las reglas F1a y F1b., mientras que en el segundo intervalo, la recta depende de los valores de los consecuentes de las reglas F1b y F1c ($c_{1b}\mu_M + c_{1c}\mu_L$)



	x_1	inter
Regla F1a	S	-1 (c_{1a})
Regla F1b	M	0 (c_{1b})
Regla F1c	L	1 (c_{1c})

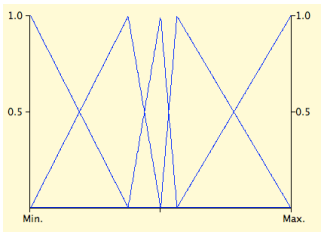


(a) **(b)** **(c)**

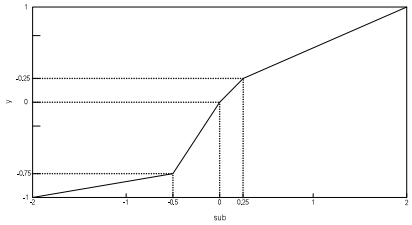
Figura 27. Sistema F1: Funciones de pertenencia de la varible x_1 (a) Base de reglas (b) Representación de inter frente a x_1 (c)

El segundo módulo del ejemplo de la *Figura 26* emplea 5 funciones de pertenencia triangulares para cubrir la entrada sub (*Figura 28a*) e implementa la base de reglas mostrada en la *Figura 29b.* La partición de la *Figura 28a* distingue 4 intervalos en la variable sub: $[-2,-0.5]$, $[-0.5,0]$, $[0,0.25]$ y $[0.25,2]$ (como puede verse en la *Figura 28c*) y para los que la salida, y, es lineal. En el primer

intervalo, la recta depende de los valores de los consecuentes de las reglas F2a y F2b, mientras que en el segundo intervalo, la recta depende de los valores de los consecuentes de las reglas F2b y F2c, en el tercer intervalo, la recta depende de los valores de los consecuentes de las reglas F2c y F2d y en el cuarto intervalo depende de F2d y F2e.



	sub	y
F2a	XS	-1 (c_{2a})
F2b	X	-0.75 (c_{2b})
F2c	M	0 (c_{2c})
F2d	L	0.25 (c_{2d})
F2e	XL	1 (c_{2e})

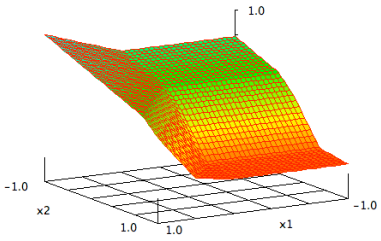


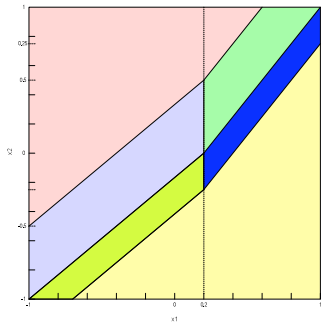
(a) **(b)** **(c)**

Figura 28. Sistema F2: Funciones de pertenencia de la varible sub (a) Base de reglas (b) Representación de y frente a sub (c)

En la *Figura 29a* se muestra la salida total del sistema, y, frente a las entradas globales, x_1 y x_2. La *Figura 29b* muestra los politopos que se distinguen en el espacio (x_1, x_2). Por ejemplo, para el politopo marcado en azul la salida global sería:

$$y = F2(sub) = c_{2c}\mu_L + c_{2c}\mu_{XL} = f_1 sub + h_1 = f_1[F1(x) - x_2] + h_1$$
$$= f_1[c_{1b}\mu_M + c_{1c}\mu_L - x_2] + h_1 = f_1 \cdot [f_2 x_1 + h_2 - x_2] + h_1 = f_1 f_2 x_1 - f_1 x_2 + f_1 h_2 + h_1$$





(a) (b)

Figura 29. Superficie de control del sistema jerárquico de ejemplo de la *Figura 26* (a) y su división del dominio en politopos (b)

La estructura jerárquica ya ha sido empleada con anterioridad en sistemas de control para robótica. El planteamiento de este Proyecto ha sido demostrar la aplicabilidad de esta estructura a otros casos de estudio, en particular, al doble integrador que se analiza en la siguiente sección.

4.3. Caso de estudio: Doble integrador

El doble integrador es un sistema típico que se aborda en control ya que es un sistema altamente inestable.

4.3.1. Descripción de los datos: Doble integrador

Se considera el doble integrador tal y como se muestra en la siguiente ecuación:

$$y(t) = \frac{1}{s^2} u(t) \qquad \textbf{Ecuación 30}$$

donde $y(t)$ es la salida del integrador y $u(t)$ representa la acción de control. La representación en el espacio de estados de su equivalente discreto es la siguiente:

$$\begin{aligned} x(t+1) &= \begin{bmatrix} 1 & 1 \\ 0 & 1 \end{bmatrix} x(t) + \begin{bmatrix} 0 \\ 1 \end{bmatrix} u(t) \\ y(t) &= \begin{bmatrix} 1 & 0 \end{bmatrix} x(t) \end{aligned} \qquad \textbf{Ecuación 31}$$

Obtenido ajustando:

$$\begin{aligned} \ddot{y}(t) &\approx \frac{\dot{y}(t+T) - \dot{y}(t)}{T} \\ \dot{y}(t) &\approx \frac{y(t+T) - y(T)}{T} \\ T &= 1s \end{aligned} \qquad \textbf{Ecuación 32}$$

El objetivo del controlador es regular el sistema hacia el origen partiendo de cualquier estado inicial. La técnica propuesta por *A. Bemporad* para diseñar un controlador *PWA* que resuelve este problema se basa en minimizar la siguiente función cuadrática:

$$\sum_{r=0}^{\infty} y'(t) y(t) + \frac{1}{10} u^2(t) \qquad \textbf{Ecuación 33}$$

sujeta a las siguientes restricciones:

$$-1 \le u(t) \le 1 \qquad \textbf{Ecuación 34}$$

Utilizando esta técnica, que se automiza con *Hybrid Toolbox* para *Matlab* [26] se obtiene un controlador *PWA* explícito para unos rangos determinados de las variables de estado.

El sistema controlador tiene por tanto dos entradas a las que se llamarán $in1$ e $in2$, que corresponden a las variables de estado del doble integrador y una salida, a la que llamaremos out, que corresponde a la entrada del doble integrador, u.

4.3.2. Diseño de un controlador difuso sin jerarquía

Se procede a utilizar la herramienta de *Xfuzzy Xfdm* para la extracción de la base de reglas de un controlador difuso. Para ello se utiliza el método de tipo grid basado en el algoritmo de Wang y Mendel.

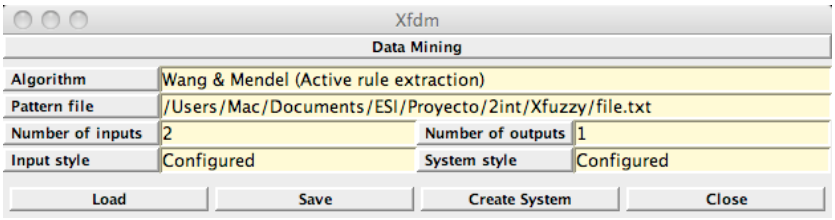


Figura 30. Configuración de la herramienta *Xfdm*

Se configurarán las entradas, $in1$ e $in2$, con 7 funciones de pertenencia de tipo familia triangular.

El sistema quedará de la siguiente manera:

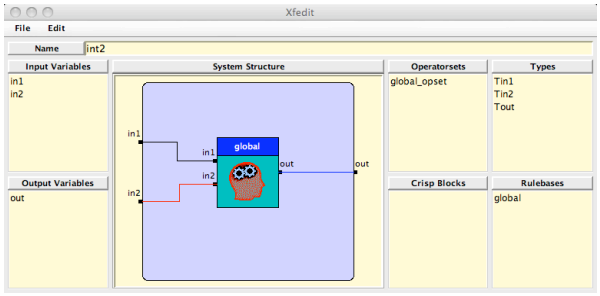


Figura 31. Vista del controlador doble integrador con la herramienta *Xfedit*

En la siguiente figura se puede observar la superfice de control tras utilizar la herramienta *Xfdm* de *Xfuzzy:*

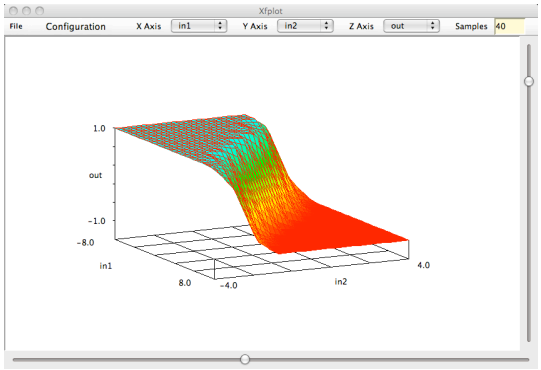


Figura 32. Vista de la superficie de control con la herramienta *Xfplot*

A continuación se realiza un aprendizaje supervisado del sistema mediante la herramienta *Xfsl*. Utilizando como fichero de datos numéricos el mismo que se ha usado en la identificación. Para ello se utiliza el método de optimización numérica de tipo *Quasi-Newton*, concretamente *Marquardt-Levenberg* que como ya se explicó en el apartado 3.1.3 es el más adecuado.

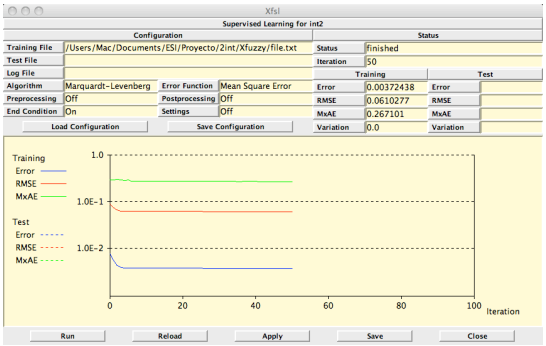


Figura 33. Resultado de utilizar la herramienta *Xfsl*

En la *Tabla 5* se muestran los errores antes y después de utilizar la herramienta de aprendizaje supervisado. El RMSE se mejora en 25,63%

	Antes	Después
Error	0.00751063	0.00372438
RMSE	0.0866638	0.0610277
MxAE	0.286113	0.267101
Variación	-	0

Tabla 5. Errores antes y después de utilizar la herramienta *XfsI*

En la *Figura 34* se muestra la superficie generada después del aprendizaje (que visualmente es similar a la mostrada en la *Figura 32*):

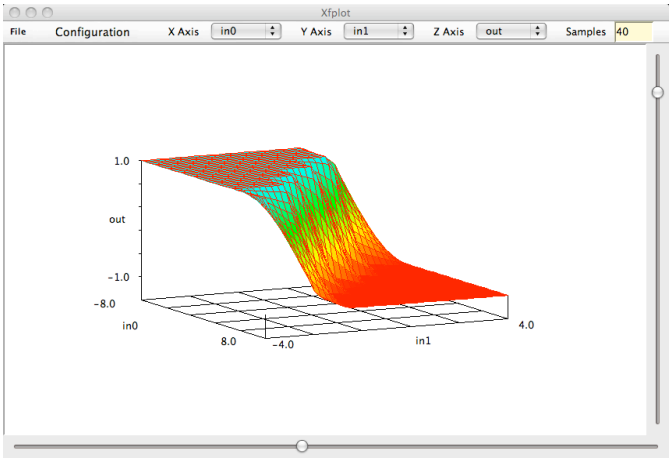


Figura 34. Superficie de control después de utilizar la herramienta *XfsI*

El sistema tiene una complejidad de $7+7=14$ funciones de pertenencia de entrada y de 49 *Singletons* de salida (es decir, 49 reglas). Para reducir la complejidad del sistema, se procede a hacer una simplificación de las reglas. Para ello se utiliza la herramienta de simplificación de *Xfuzzy* (*Xfsp)*. Se realiza un clustering de los *Singletons* de salida a 15 grupos puesto que visualmente se distinguen 15 valores en los consecuentes.

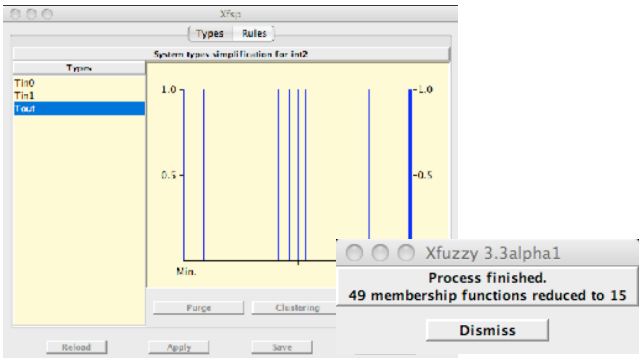


Figura 35. Herramienta de simplificación *Xfsp*

La *Figura 35* ilustra cómo se reduce el numero de *Singletons* de la salida de 49 a 15. A continuación, procedemos a aplicar una simplificación tabular de la base de las reglas, reduciéndose a 25 el número de reglas. También se intenta reducir el número de funciones de pertenencia de las entradas, sin embargo no se consigue. La superficie generada se muestra en la siguiente figura:

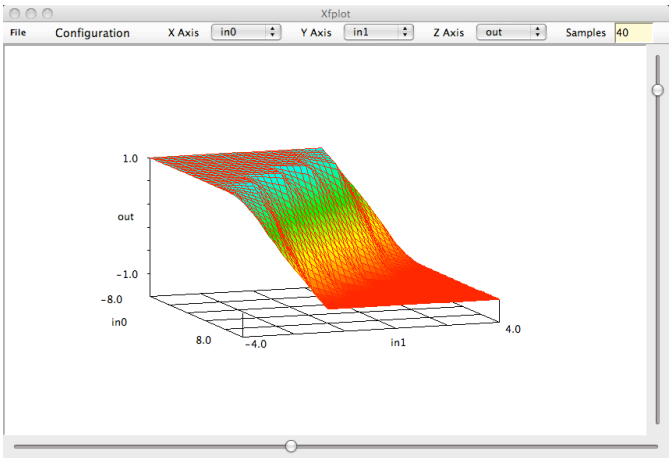


Figura 36. Superficie de control después de usar la herramienta *Xfsp*

Al resultado de la simplificación se le hace un aprendizaje supervisado, para ello se aplica el mismo algoritmo que en el caso anterior. Después de realizar el aprendizaje supervisado vemos que tenemos los valores de errores mostrados en la *Tabla 6*.

	Identificado	Identificado+ aprendizaje	Ident.+simplif. aprendizaje
Error	0.00751063	0.00372438	0.00372452
RMSE	0.0866638	0.0610277	0.0610289
MxAE	0.286113	0.267101	0.270511
Variación	-	0	0

Tabla 6. Errores antes y después de utilizar la herramienta *Xfsl*

Se puede ver que el sistema simplificado es mejor que el inicial puesto que no empeora la mejora del *RMSE* y sí reduce significativamente la base de reglas; 15 en vez de 49 valores de consecuentes y 25 en vez de 49 reglas.

La superficie de control generada después de la simplificación y aprendizaje la podemos ver en la *Figura 37*.

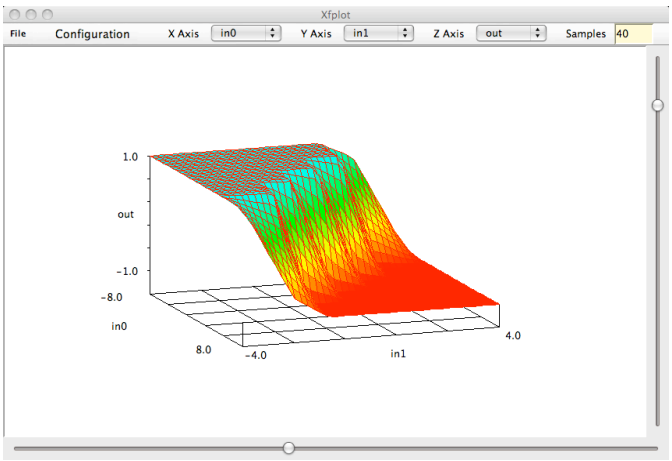


Figura 37. Superficie de control después de utilizar la herramienta *Xfsl*

Vamos a analizar a continuación la solución que se obtendría con un sistema jerárquico.

4.3.3. Diseño de un controlador difuso con jerarquía

Usamos como punto de partida del diseño el sistema jerárquico que se ha introducido en los *apartados 4.1* y *4.2* tal y como se ve en la *Figura 38*.

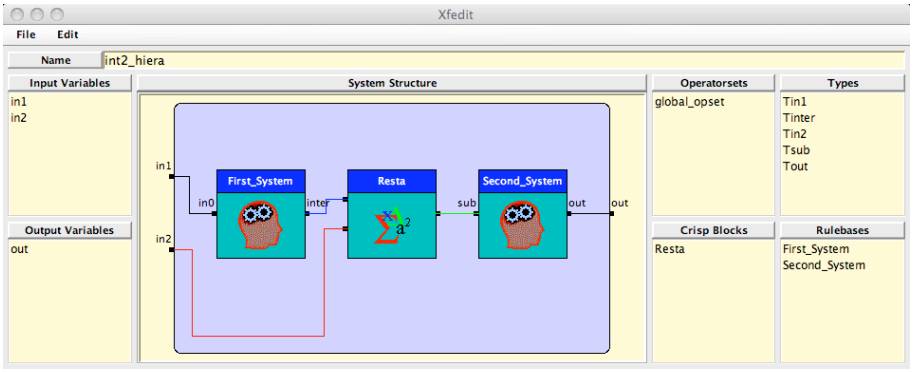


Figura 38. Sistema jerárquico

La salida del primer sistema será una función *PWA* de la entrada $in1$. La salida del segundo sistema será tambien una función *PWA* de su entrada, es decir, de sub. Por lo tanto se tiene:

$$\begin{aligned} \text{int}er &= PWA(in1) \\ sub &= \text{int}er - in2 = PWA(in1) - in2 \\ out &= PWA(sub) = PWA(PWA(in1) - in2) \end{aligned}$$

Ecuación 35

En este sistema las entradas $in1$ e $in2$ tendrán valores comprendidos entre -8 y 8, -4 y 4 respectivamente (que son los rangos para los que se diseñó el controlador *PWA* explícito). La entrada del primer subsistema, $in1$, se cubre con 6 funciones de pretenencia triangulares solapadas al 50%. La salida $\text{int}er$ se cubre con 6 valores *Singletons*.

Después se emplea una bloque tipo *crisp* que realiza la resta entre la salida del primer subsistema y la entrada $in2$. El resultado de esta resta es la variable sub cuyos valores están comprendidos entre -8 y 8, y que se cubre con 4 funciones de pertenencia triangulares solapadas al 50%. La salida de el segundo subsistema, out, podrá tomar dos valores de tipo *Singleton*.

Todos los parámetros del sistema jerárquico se eligen por defecto a unos determinados valores que, en absoluto ofrecen una superficie de control adecuada. Para ajustarlos se entrena este sistema tal y como se hacía en el apartado anterior, es decir, se introduce el fichero de datos numéricos obtenidos con la herramienta como fichero de entrenamiento y se utiliza el método de optimización numérica de tipo *Quasi-Newton*, *Marquardt-Levenberg* estimando la derivada por el método de la secante.

Vemos que se obtienen los valores de la *Tabla 7*. En este caso el aprendizaje sí es significativo puesto que nuestro sistema de partida no contenía ninguna información.

	Antes	Después
Error	0.143309	1.3751E-4
RMSE	0.378562	0.0117265
MxAE	0.622279	0.0709671
Variación	-	3.7644E-8

Tabla 7. Errores antes y después del utilizar la herramienta *Xfsl* en el sistema jerárquico

La gráfica que se muestra en la *Figura 39* de la superficie generada después del aprendizaje:

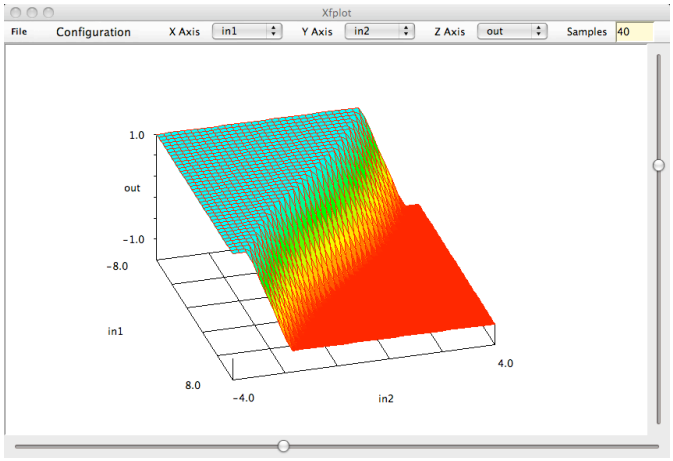


Figura 39. Superfice de control después de utilizar la herramienta *Xfsl* en el sistema jerárquico

De la misma manera que se hace en el *apartado 4.3.2*. se procede a hacer una simplificación de las reglas. Se sigue el mismo procedimiento y se reducen los *Singletons* del segundo subsistema de 4 a 2. Una vez simplificado se hace un aprendizaje supervisado y se obtienen los siguientes valores de error de la *Tabla 8*. De nuevo el sistema simplificado y aprendido resulta ser la mejor solución porque simplifica la base de reglas y no empeora significativamente el *RMSE*.

	Identificado	Identificado+ aprendizaje	Ident.+simplif. aprendizaje
Error	0.143309	1.3751E-4	1.3755E-4
RMSE	0.378562	0.0117265	0.0117282
MxAE	0.622279	0.0709671	0.0709565
Variación	-	3.7644E-8	0

Tabla 8. Tabla de errores antes y después de usar la herramienta Xfsl en el sistema jerárquico simplificado

En la *Figura 40* se observa la superficie de control generada después de realizar el aprendizaje supervisado:

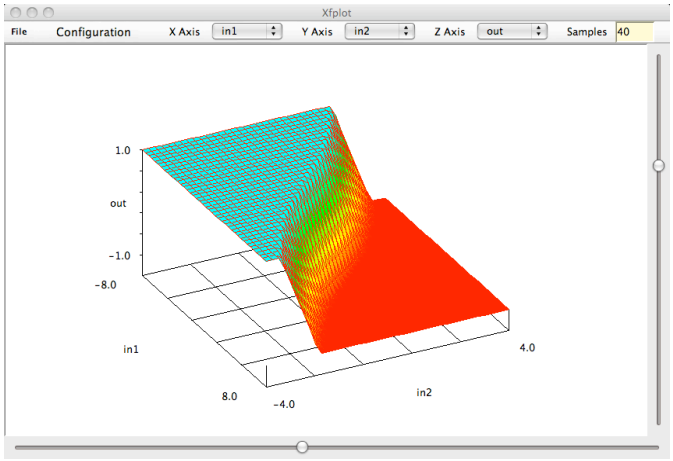


Figura 40. Superficie de control después de utilizar la herramienta *Xfsl* en el sistema jerárquico

El sistema jerárquico consigue una superficie de control bastante ajustada a los datos numéricos con solo 8 reglas mientras que el no jerárquico necesita 25. Con lo cual, seleccionamos la solución jerárquica para diseñar el controlador *PWA*.

En cualquier caso, la aproximación de los datos numéricos sólo tiene como objetivo la reducción en el *RMSE* pero no tiene en cuenta ningún objetivo relacionado con el control. Al monitorizar con la herramienta *Xfmt* de *Xfuzzy* ciertos valores de entradas no se obtiene a la salida el resultado deseado, pero esto se soluciona modificando ligeramente las funciones de pertenencia para conseguir la simetría exigida por el control.

4.3.4. Simulación con la herramienta *Xfmt* de *Xfuzzy*

El siguiente paso consiste en hacer una simulación en lazo cerrado con la herramienta *Xfsim* que nos proporciona Xfuzzy. Para ello, lo primero de todo es escribir un modelo de la planta que queremos controlar, es decir, del doble integrador en java:

```
import xfuzzy.PlantModel;
public class DoubleIntModel implements PlantModel {
private double in1;
private double in2;
private double out;
public DoubleIntModel() {
}
public void init() {
        in1=10;
        in2=-5;
}
public void init(double val[]) {
        in1 = val[0];
        in2 = val[1];
}
public double[] state() {
        double state[] = new double[2];
        state[0] = in1;
        state[1] = in2;
        return state;
}
public double[] compute(double val[]) {
        out=val[0];
        double oldin1 = in1;
        double oldin2 = in2;
        in1=oldin1+oldin2;
        in2=oldin2+out;
        return state();
}
}
```

Una vez que se tiene el modelo de la planta hay que hacer una simulación en la que elegimos la siguiente configuración :

```
xfsim_plant("C:\Proyecto\2int\Xfsim\DoubleIntModel.class")
xfsim_init(3.0, -4.0)
xfsim_limit(_n < 100.0)
xfsim_plot(in1,in2,0)
xfsim_plot(_n,out,0)
xfsim_plot(_n,in1,0)
xfsim_plot(_n,in2,0)
```

Las siguientes figuras muestran el resultado de la simulación con la herramienta *Xfsim* de *Xfuzzy*. En la *Figura 41* se puede ver la gráfica del espacio de estados, en la cual se ve cómo desde un

estado inicial determinado $((3,-4)$ en este caso**)**, el sistema se controla hacia el origen como se esperaba.

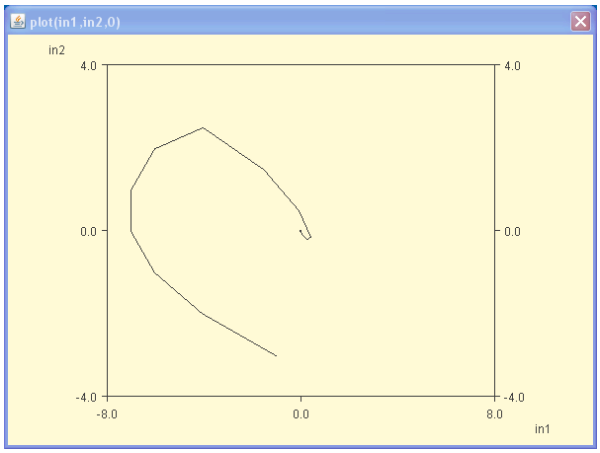


Figura 41. Resultado de la simulación *in2* frente *in1* con *Xfsim*

En la *Figura 42, Figura 43* y la *Figura 44* se ve como tanto las entradas, $in1$ e $in2$, como la salida, out, tienden a 0 en un número pequeño de ciclos de control.

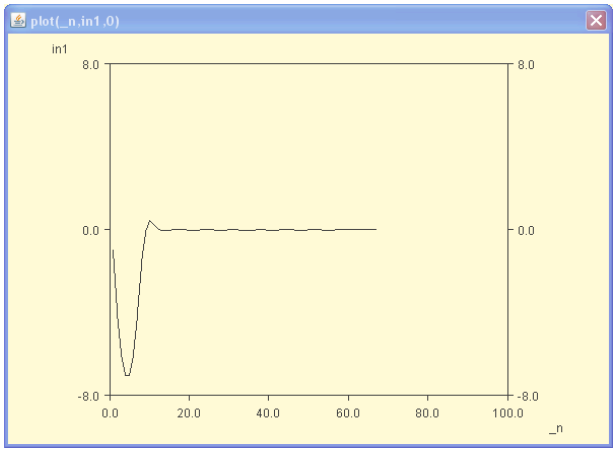


Figura 42. Resultado de la simulación *in1* con *Xfsim*

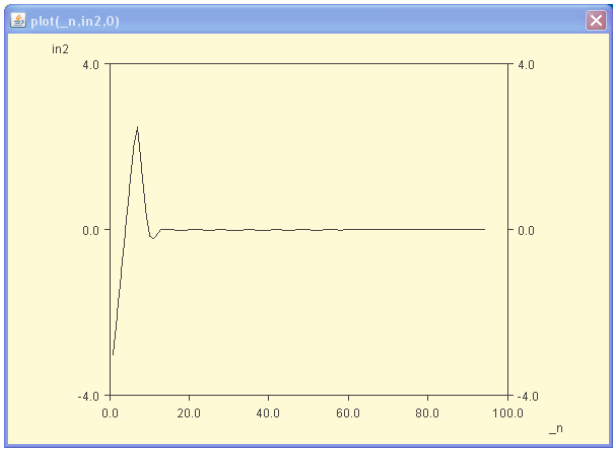


Figura 43. Resultado de la simulación *in2* con *Xfsim*

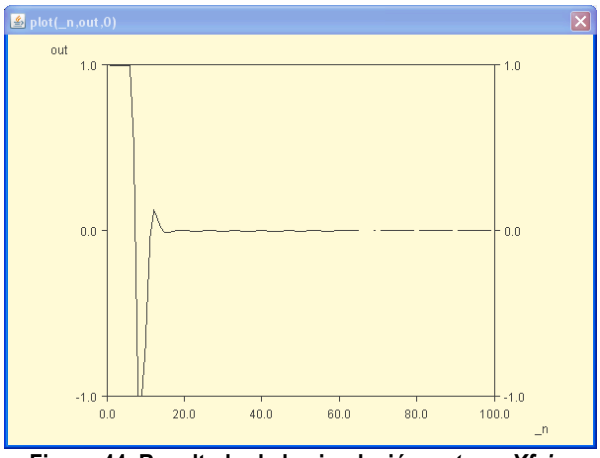


Figura 44. Resultado de la simulación *out* con *Xfsim*

4.3.5. Herramienta *Xfsg* de *Xfuzzy*

El siguiente paso consiste en hacer un modelo en *Simulink* del controlador para poder hacer las simulaciones correspondientes en lazo abierto y lazo cerrado contemplando las características del *hardware*, que hasta ahora no se han contemplado.

En primer lugar, se selecciona el sistema y se arranca la herramienta *Xfsg* de *Xfuzzy* que tiene el aspecto que se ve en la *Figura 45*.

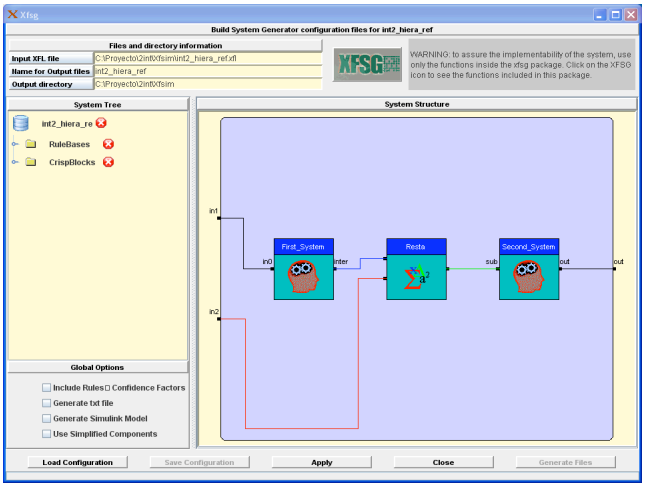


Figura 45. Vista principal de la herramienta *Xfsg*

Para cada base de reglas que aparece en el sistema difuso (‘*First_System’* y ‘*Second_System’*) se suministra a la herramienta el número de bits, así como para el bloque crisp *‘Resta’*. La precisión que se elige para cada sistema, tanto para los bits de entrada, como para los bits de salida será de 12 bits. Para la parte entera de las pendientes de las funciones de pertenencia del primer sistema se elige 3 bits y para la parte entera de las pendientes del segundo sistema se elige 4 bits.

Se generan los siguientes ficheros:

- ***int2_hiera_ref.m*** es un fichero *‘.m’* de *Matlab* que contiene la inicialización de las variables de cada uno de los bloques de la librería *XfuzzyLib* que se usan para implementar el sistema difuso.

- ***int2_hiera_ref_aux.mdl*** contiene el modelo *Simulink* del sistema difuso utilizando los módulos incluidos en la librería *XfuzzyLib*.

- ***int2_hiera_ref.txt*** contiene una descripción en formato texto de las entradas y salidas de cada

base de reglas y bloque *crisp*. También incluye el componente que se utilizará de la librería *XfuzzyLib*. Si no existe dicho componente, se especifica con un *'null'*.

La *Figura 46* muestra la representación del sistema de control mediante un modelo *Simulink*. El modelo incluye dos *FLCs* (*FLC2_P_TS1s* y *FLC1_FMs)* disponibles en *XfuzzyLib*, así como un bloque aritmético, que ha tenido que ser definido con primitivas del *Xilinx Blockset*, que implementa el módulo crisp '*Resta*'. La definición de los parámetros que definen las bases de conocimiento (MFCs y reglas) de los módulos difusos se incluyen en el archivo *int2_hiera_ref.m* generado por *Xfsg*. Utilizando esta información y las distintas facilidades de *Matlab* para definir y visualizar señales se puede verificar el comportamiento entrada/salida del sistema de control, así como la influencia en dicho comportamiento de los diferentes parámetros de diseño.

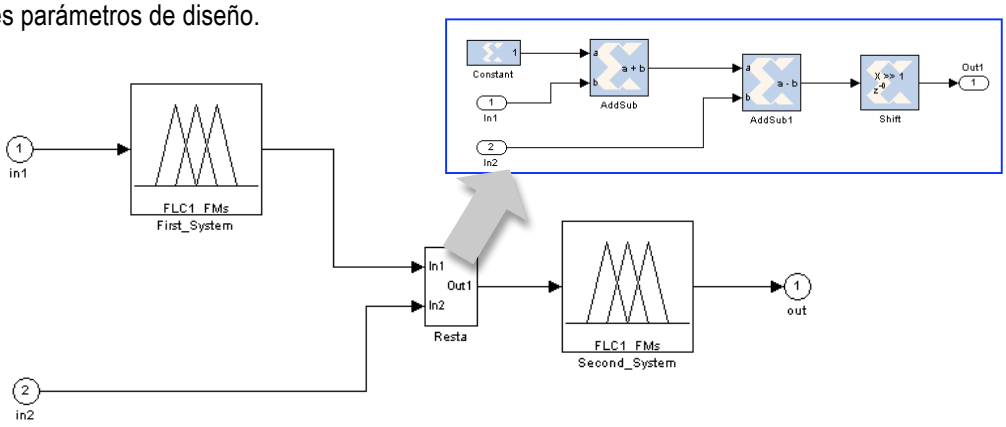


Figura 46. Representación del sistema de control difuso mediante un modelo *Simulink* con módulos de la librería *XfuzzyLib* y el modelo del módulo *Resta*

4.3.5.1. Simulación con *Matlab* y *Simulink*

El siguiente paso consiste en hacer una simulación del modelo del controlador generado por la herramienta *Xfsg*. Para ello, se crea un modelo en *Simulink* que hace un barrido normalizado de las entradas, $in1$ e $in2$, y que se conectan al modelo generado por la herramienta *Xfsg*. Este modelo se puede ver en la *Figura 47*.

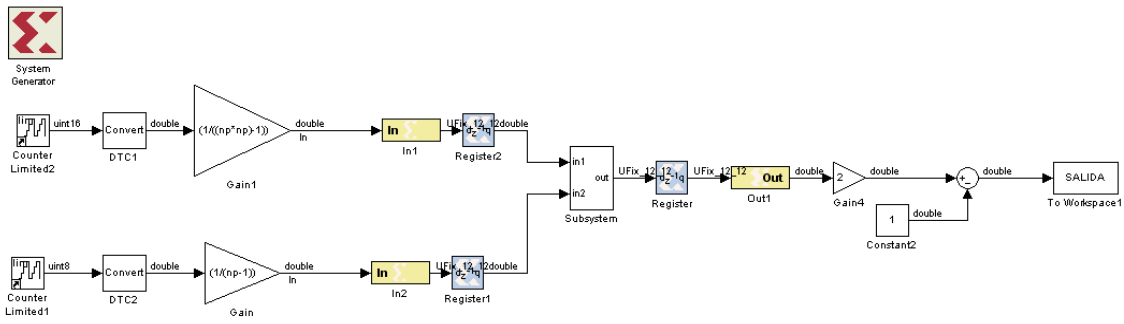


Figura 47. Modelo en *Simulink* para la simulación en lazo abierto del controlador

El resultado de ejecutar esa simulación se muestra en la *Figura 48*.

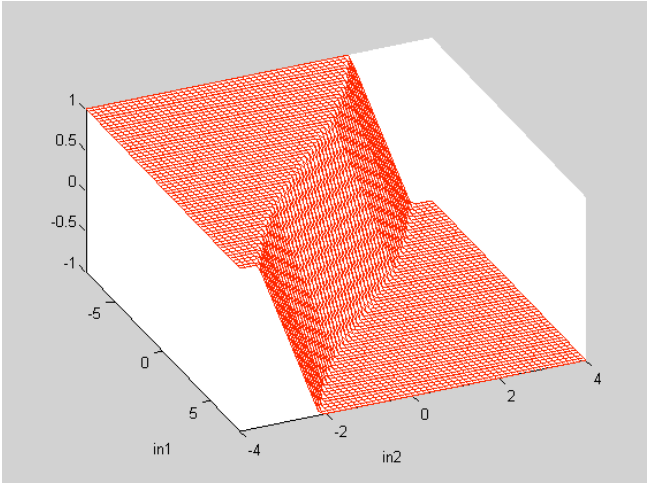


Figura 48. Superficie de control simulada con el modelo generado por la herramienta *Xfsg*

Una vez que se ha realizado el barrido en bucle abierto habrá que realizar una simulación en lazo cerrado del modelo, tal y como haciamos con la herramienta *Xfsim* hay que crear un modelo de la planta. En esta ocasión el modelo será un modelo *Simulink*. Tambien se creará una máscara para configurar los parametros como los valores iniciales de $in1$ e $in2$.

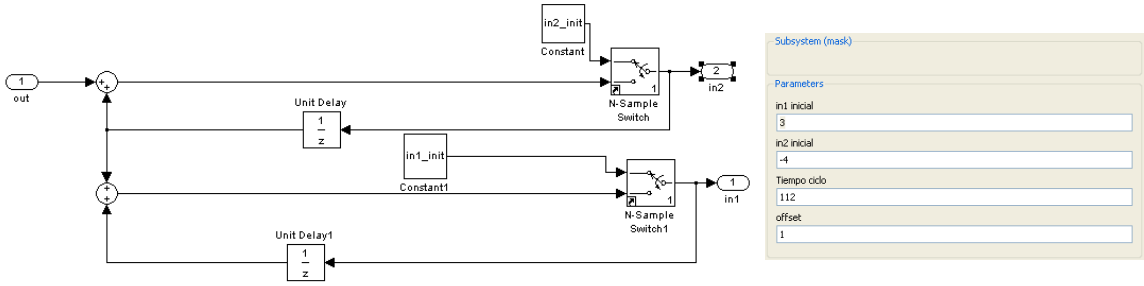


Figura 49. Modelo de la planta en *Simulink* y máscara de definición de parámetros

El siguiente paso es conectar el modulo generado por *Xfsg* y el modelo de la planta poniendo unos bloques que adapten las entradas y salidas de los mismos. En el caso de *out* saldrá del modelo del controlador normalizado por lo que desplazaremos a valores comprendidos entre -1 y 1. En el caso de $in1$ e $in2$, habrá que normalizar los valores que salen del modelo de la planta que serán entre -8 y 8, y entre -4 y 4 respectivamente para acondicionarlos a la entrada del modelo del controlador entre 0 y 1. El modelo queda por tanto:

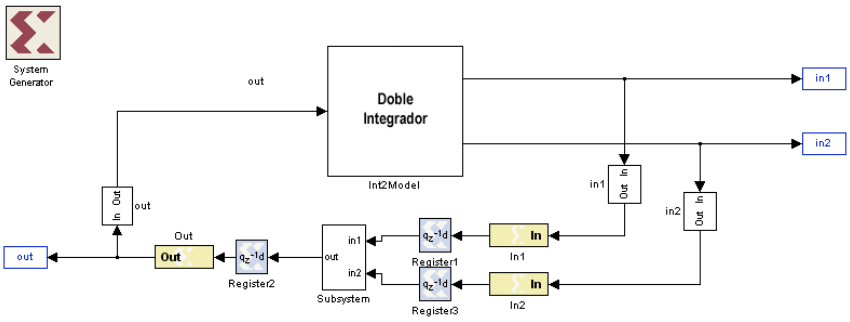


Figura 50. Modelo para la simulación en lazo cerrado

El resultado de la simulación de lazo cerrado es el que se observa a continuación:

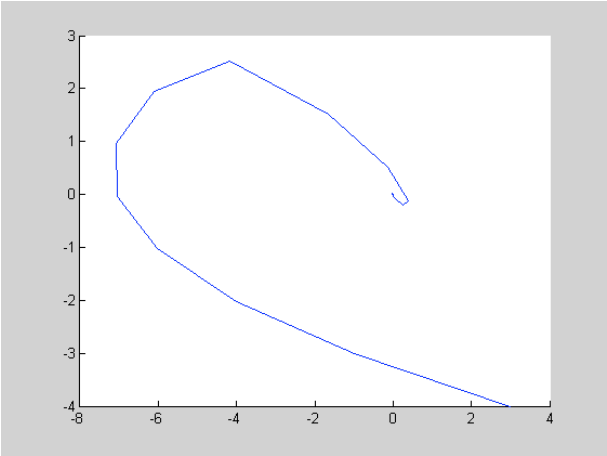


Figura 51. Representación de *in2* frente a *in1* después de la simulación en lazo cerrado

En la *Figura 51* se puede ver la gráfica en el espacio de estados, tal y como se obtenía en la simulación con la herramienta *Xfsim* de *Xfuzzy*. Se vé como desde un estado inicial determinado (de nuevo $(3,-4)$ el sistema se controla hacia el origen como se esparaba. En las *Figura 52*, *Figura 53* y *Figura 54* se puede ver como tanto las variables de entrada del controlador, $in1$ e $in2$, como la de salida, out, tienden a 0.

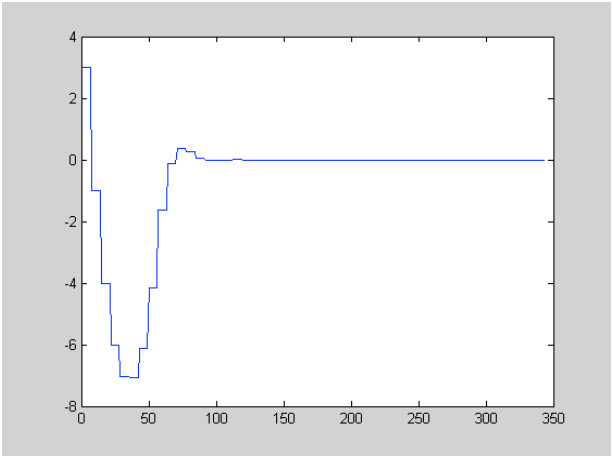


Figura 52. Representación de *in1* después de la simulación en lazo cerrado

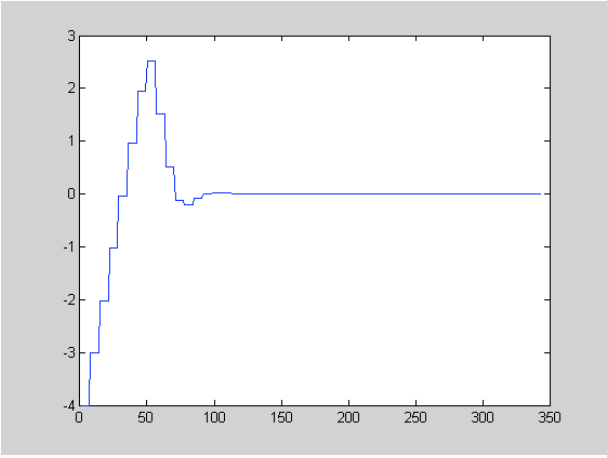


Figura 53. Representación de *in2* después de la simulación en lazo cerrado

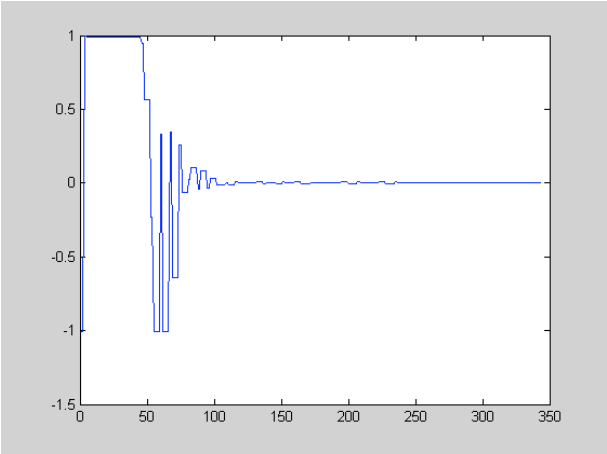


Figura 54. Representación de *out* después de la simulación en lazo cerrado

4.3.5.2. Generación del *bitstream* para *FPGA*

Se va a usar una *FPGA Spartan 3 (xc3s200-5ftp256)* y el diseño va a ser sintetizado por el *software Xilinx ISE 10.1* .

Se abre el modelo de *closed_loop.mdl* y hacemos doble click en *System Generator* y se configura como se indica en la siguiente figura:

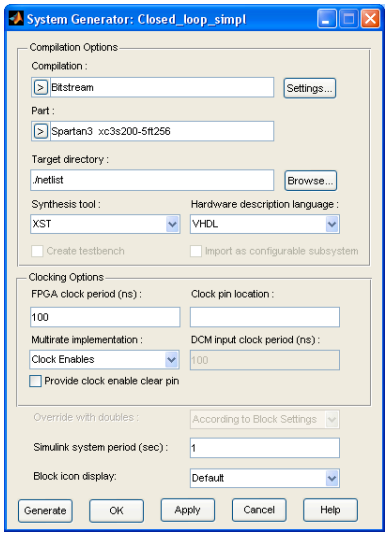


Figura 55. Configuración de bloque System Generator

Se guarda la configuración y después se generan los ficheros. El resultado de seleccionar la opción *Bitstream* es:

- Generación de cores (*Core Generator*; ".\netlist\sysgen\coregen_xxxx")
- Síntesis lógica (*Xst*; ".\netlist\synth_model")
- Implementación (*Xflow*: *ngdbuild*, *map*, *par*; ".\netlist\xflow")
- Generación del fichero '.bit' (*Xflow*:*bitgen*)

Ahora se abre el entorno *ISE 10.1* para re-implementar el fichero de forma interactiva. Para ello se abre el proyecto, *closed_loop_cw.ise*. Se procede a hacer la implementación del diseño en un solo paso por lo que elegimos el nivel *Implement design* de la ventana *Processes.*

En la siguiente figura se puede ver un resumen del diseño de la *FPGA* que resulta de esta ejecución:

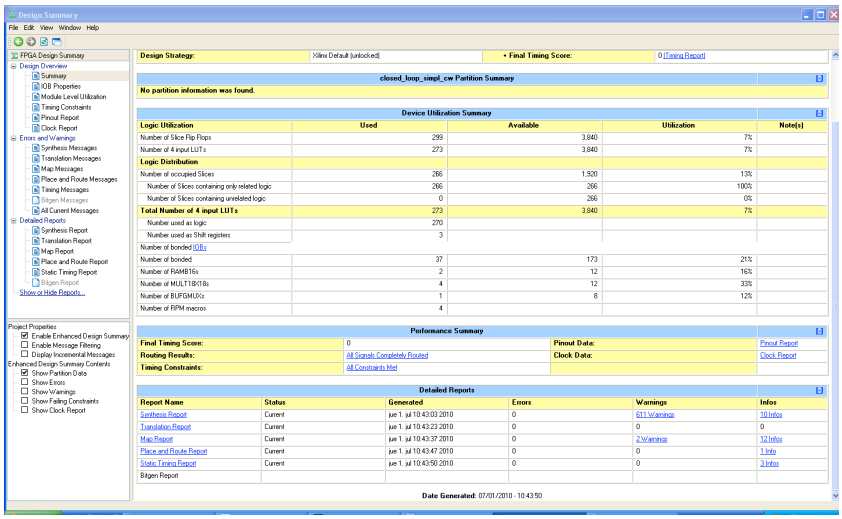


Figura 56. Resumen del diseño de la *FPGA*

El diseño difuso ocupa solo un 7% de la lógica de la *FPGA* y cuatro multiplicadores 18×18, Como la generación de la arquitectura es automática se usan 4 multiplicadores, pero se podrían haber usado 2 si se hubiesen aprovechados los mismos 2 multiplicadores para los 2 módulos difusos. Aplicando *pipeline*, no se necesitan los 4 multiplicadores a la vez, sino sólo 2: primero para el primer módulo y luego para el segundo. Incluso para un mismo módulo no se necesitan los dos multiplicadores a la vez, sino que se usa primero un multiplicador para generar el grado de pertenencia y luego otra vez para el producto por los consecuentes. Es decir, que temporizando adecuadamente el circuito se podría usar un solo multiplicador.

La frecuencia máxima de operación es de $62.39MHz$. Se necesitan 32 ciclos de reloj para obtener el primer dato y 8 ciclos para obtener cada dato.

4.3.5.3. Cosimulación *hardware/software*

Para realizar la cosimulación *Hardware/Software* se conecta una placa *FPGA Spartan 3A*. El concepto de cosimulación *Hardware/Software* de System *Generator* implica que, dado un modelo *Simulink*, la parte implementable del modelo (la que se encuentra entre los gateways de entrada y salida) será sintetizada, implementada y programada sobre *FPGA*, mientras que el resto del modelo seguirá ejecutándose en código *Matlab*. De esta forma, y en relación con la aplicación que se está desarrollando, esto se traducirá en que, al lanzar la simulación, el controlador difuso se implementará sobre la *FPGA* e interactuará con el modelo del doble integrador y las distintas utilidades de visualización.

Se abre el bloque *System Generator*. Se selecciona como opción de compilación *Hardware Co-simulation -> Spartan3AK* y se inicia el proceso. Una vez completado se generará en el directorio *'.\netlist_hw'* la librería *Closed_loop_hwcosim_lib* que contiene el modelo de un bloque tipo *JTAG Co-sim* denominado *Closed_loop hwcosim*.

Se inserta el bloque *JTAG* en el modelo que se tenía para hacer la simulación en lazo abierto, el bloque *Closed_loop hwcosim* como se ve en la *Figura 57.*

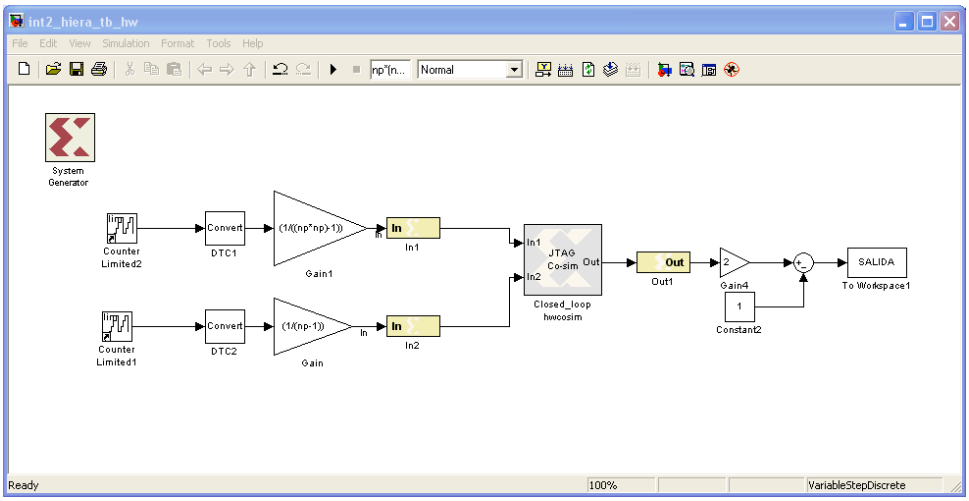


Figura 57. Modelo Simulink para simulación en lazo abierto en hardware

El resultado de esta simulación es:

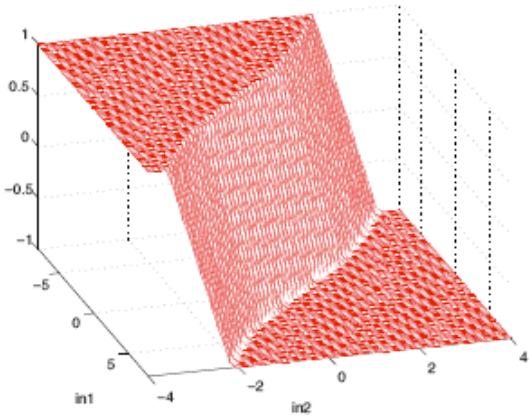


Figura 58. Superficie de control simulada en *hardware*

De la misma manera se hace una simulación en lazo cerrado, en esta ocasión el lazo se cierra con el controlador programado en la *FPGA* y no con el modelo del controlador como en el *apartado 4.3.5.1*. Esta simulación es conocida como *Hardware in the loop*.

La *Figura 59* muestra el modelo en *Simulink* para hacer la *co-simulación Hardware/Software* en lazo cerrado:

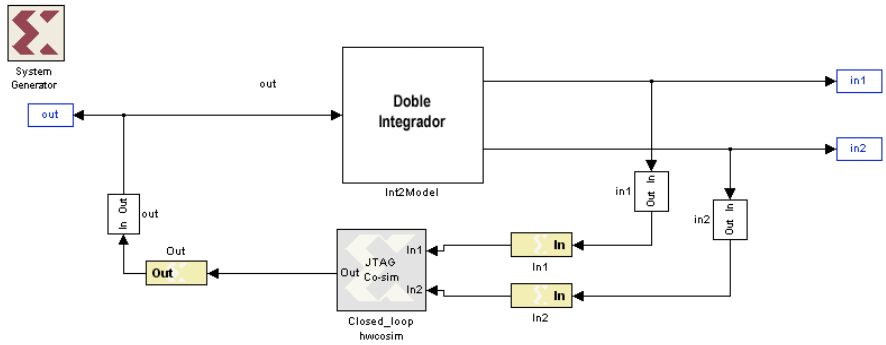


Figura 59. Modelo de co-simulación en lazo cerrado

A continuación se muestran los resultados de la simulación.

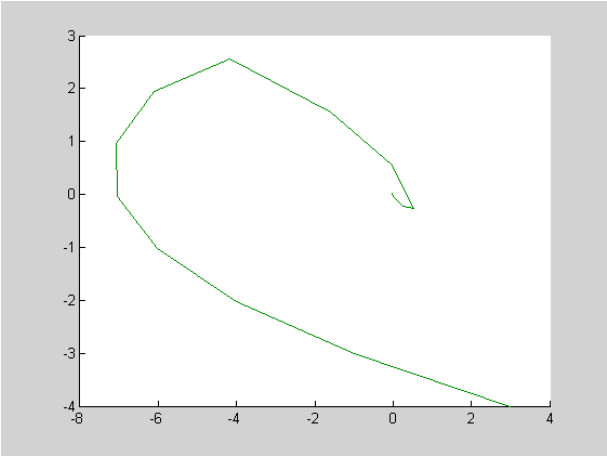


Figura 60. Representación de *in2* frente a *in1* después de la simulación *Hardware in the loop*

En la *Figura 60* se observa como el sistema tiende al origen desde un estado inicial dado tal y como se esperaba. En las *Figura 61*, *Figura 62* y *Figura 63* se ve que tanto las variables de entradas del controlador como la de salida tiende a 0.

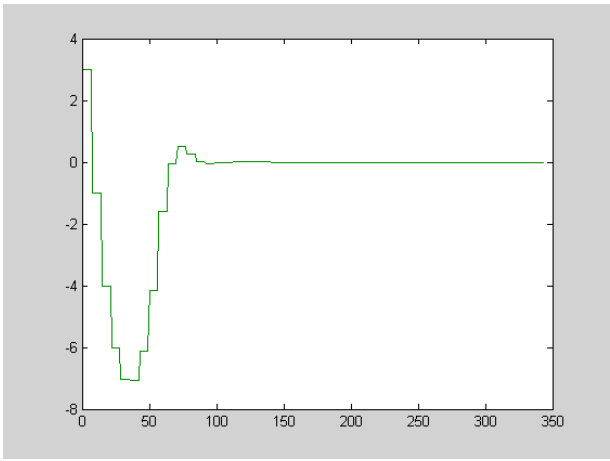


Figura 61. Representación de *in1* después de la simulación *Hardware in the loop*

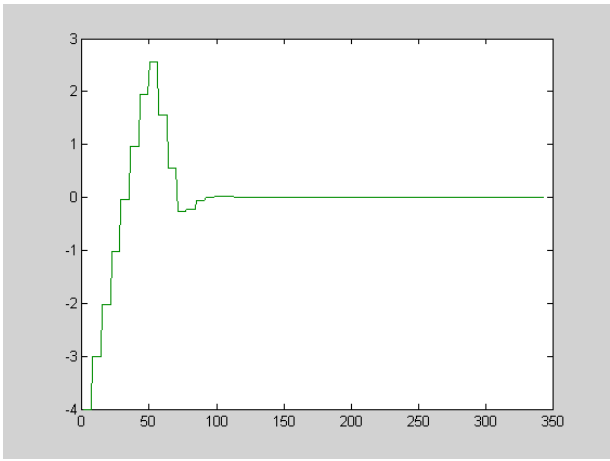


Figura 62. Representación de *in2* después de la simulación *Hardware in the loop*

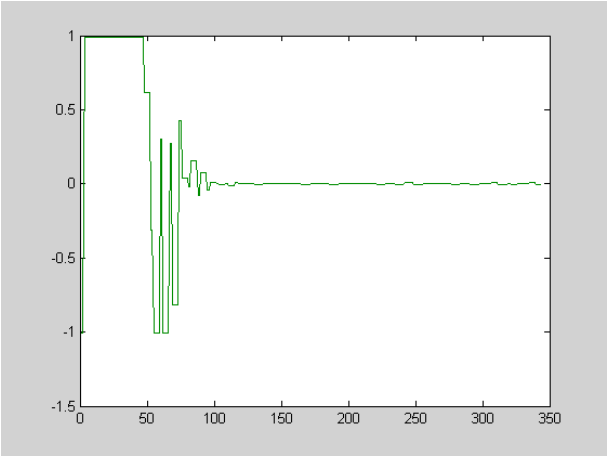


Figura 63. Representación de *out* después de la simulación *Hardware in the loop*

4.3.6. Comparación de resultados

En esta sección se pretende comparar los resultados obtenidos en este proyecto con los resultados obtenidos en [57], en el cual se hace una implementación de funciones *PWA* basándose en árboles de busca binaria.

También se compara con el resultado de implementar este controlador en las arquitecturas serie y paralelo propuestas en [58] para implementar funciones PWAS.

En todos los casos se toma como fichero de partida un archivo generado por una rutina de *Matlab* que describe el caso de estudio y que utiliza la *Hybrid Toolbox*. En el caso de [57] se utiliza también una rutina de *Matlab* que genera *off-line* el bloque de memoria que implementa la máquina de estados del árbol de búsqueda binaria.

En la *Figura 64* se puede ver la partición del dominio del caso *PWA* [57]. Se puede observar que el dominio se divide en 25 politopos. Los politopos con el mismo color comparten la misma expresión afín. En el caso del controlador difuso el dominio se divide en 7 politopos como se ve en la *Figura 65.*

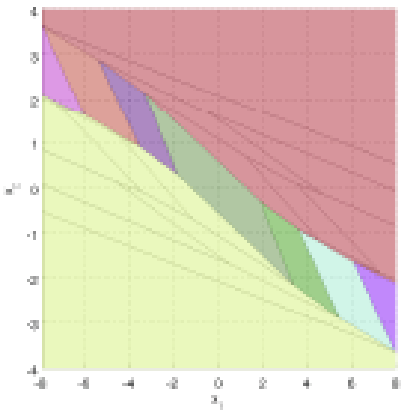


Figura 64. División en politopos del dominio dada por *Hybrid Toolbox*

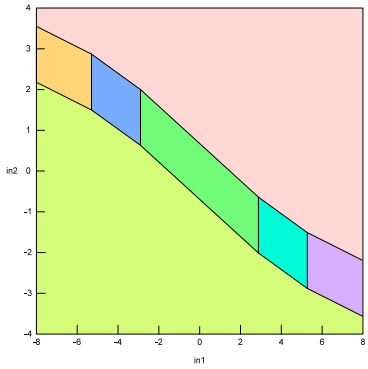


Figura 65. División en politopos del dominio dada por el sistema difuso

A continuación se muestra la superficie de control generada en el caso de [57]:

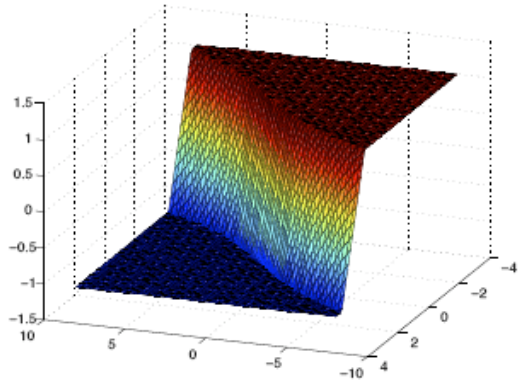


Figura 66. Superficie de control dada por la herramienta de *Bemporad*

En las siguientes gráficas se pretende hacer una comparativa visual de los resultados obtenidos en ambos casos. Para ello se superponen las gráficas de las superficies de control:

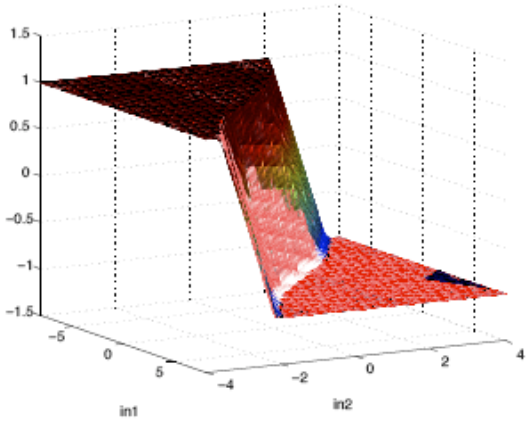


Figura 67. Superficie de control dada por la *Toolbox Hybrid* superpuesta con la resultante de la simulación *hardware* del controlador difuso.

A la vista de la figura se puede ver como el resultado de ambas superficies son bastante parecidas tanto en las zonas en los que la salida tiene un valor constante 1 ó -1, sino que también se adecúan bastante en la zona de transición.

El resultado de la simulación en lazo cerrado con unos valores iniciales dados al ejecutar la H*ybrid Toolbox* [26] se puede ver el la figura siguiente:

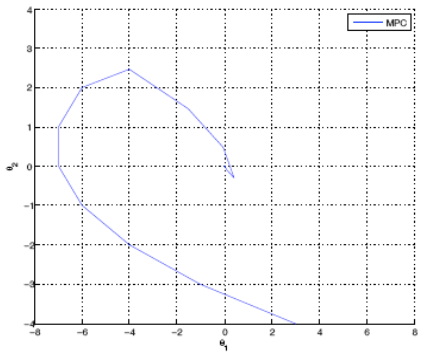


Figura 68. Simulación en lazo cerrado dada por la herramienta *Hybrid Toolbox*

Por último se hace una comparativa superponiendo ambas simulaciones en lazo cerrado en la misma gráfica:

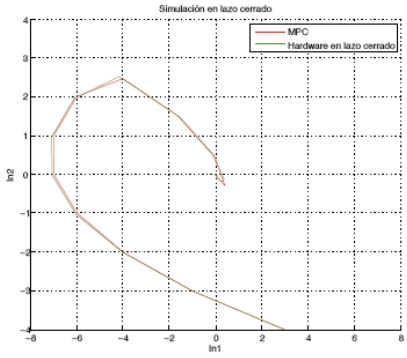
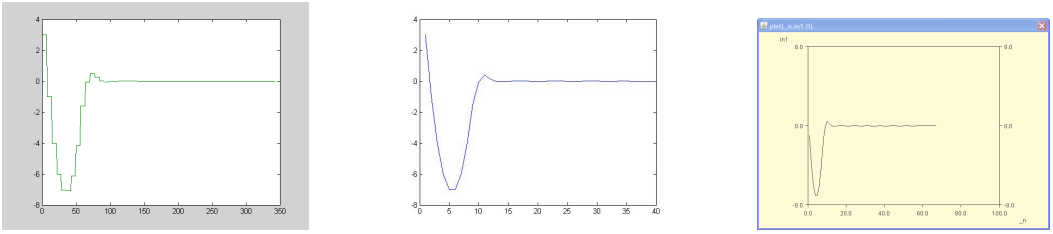


Figura 69. Simulación en lazo dada por dada por la herramienta de *Bemporad* (rojo) superpuesta con la resultante de la simulación *hardware* del controlador difuso (verde).

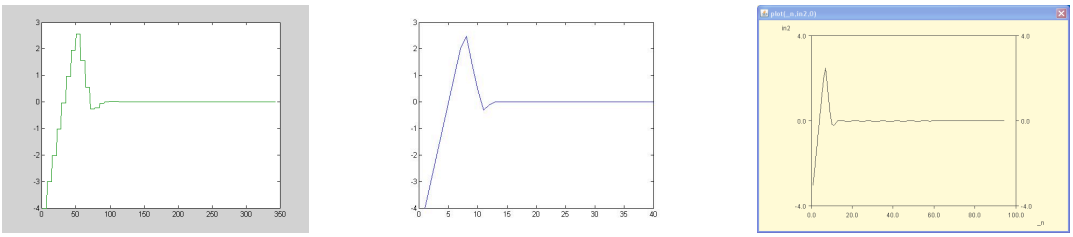
Como conclusión al ver la figura anterior es que ambos casos el comportamiento es similar, aunque se puede ver como la traza en los estados más cercanos al origen en el caso difuso es más directo que en los resultados obtenidos por la *Hybrid Toolbox* de *Bemporad* [26].

En las siguientes figuras se puede ver el comportamiento de las entradas del controlador y la salida del mismo, para la simulación dada por la herramienta *Hybrid Toolbox* de *Bemporad* [26] y por la co-simulación *hardware/software*.



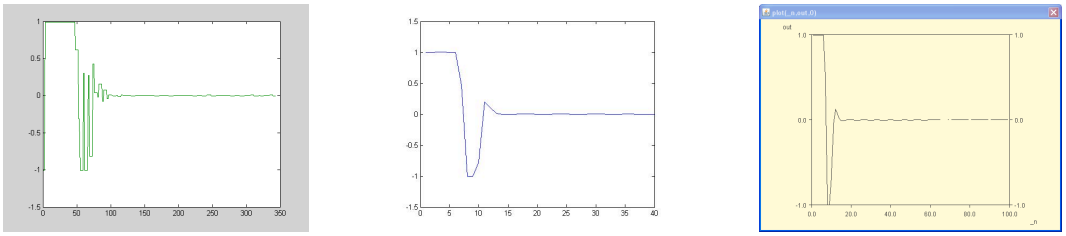
(a) (b) (c)

Figura 70. Representación de *in1* después de la simulación *Hardware in the loop (a),* de la simulación de la herramienta de *Bemporad* (b) y de la simulación de *Xfsim (c)*



(a) (b) (c)

Figura 71. Representación de *in2* después de la simulación *Hardware in the loop (a),* de la simulación de la herramienta de *Bemporad* (b) y de la simulación de *Xfsim (c)*



(a) (b) (c)

Figura 72. Representación de *out* después de la simulación *Hardware in the loop (a),* de la simulación de la herramienta de *Bemporad* (b) y de la simulación de *Xfsim (c)*

En la *Figura 70*, *Figura 71* y *Figura 72* se puede ver que ambos casos el comportamiento es el mismo aunque se puede ver en la *Figura 71* que la entrada del controlador, $in1$, es más estable en el controlador difuso, sin embargo la señal de control, out tiene más sobreoscilaciones en caso difuso que en el de la herramienta de *Bemporad*, sin embargo en el caso de la simulación de *Xfsim* la oscilación es la misma.

El diseño *PWA* [57] ocupa un 11.8% de la lógica de la *FPGA* y un multiplicador 18×18. El diseño difuso ocupa un 7% de la lógica de la *FPGA* y cuatro multiplicadores 18×18, que como se ha explicado en el *apartado 4.3.3.4.* se podría reducir a un único multiplicador temporizando el circuito adecuadamente.

En [57] la frecuencia máxima en ese caso es de $67MHz$ y en el del diseño difuso la frecuencia máxima es de $62.39MHz$ y se necesitan 2 ciclos de reloj para hacer una adquisición de un dato y otros 4 ciclos de reloj para explorar cada nodo del árbol. Además se necesita llegar desde la raíz hasta la hoja, que en el peor de los casos el punto de entrada esté en una hoja de máxima profundidad. En este caso se genera un resultado válido en 26 ciclos de reloj. En el sistema difuso necesita 32 ciclos de reloj para obtener el primer dato y 8 ciclos de reloj para los demás.

El diseño PWAS, en cuanto ocupación de la *FPGA,* se puede decir que es peor que el diseño difuso puesto que si la arquitectura paralela ocupa un 25% de la lógica de la *FPGA* y 3 multiplicadores, y la arquitectura en serie ocupa un 13% de la lógica de la *FPGA* y 1 multiplicador.

4.4. Conclusiones

Los controladores *PWA* jerárquicos evitan el problema de la "*maldición de la dimensionalidad*", y son, por tanto, más sencillos que las PWA sin jerarquía. Aprovechando las herramientas de CAD de Xfuzzy se ha diseñado un controlador jerárquico para estabilizar un doble integrador y los resultados de control son similares a las de un controlador no jerárquico.