PROYECTO DE FIN DE CARRERA

Facultad de Informática

Universidad de Murcia

ANÁLISIS DEL ENTORNO SIMULINK / RTW PARA EL DISEÑO DE SISTEMAS DE CONTROL CON LA TARJETA DSPACE 1102

AUTOR:

Joaquín Cervera López

DIRECTORES:

Alfonso Baños Torrico

Aurelio Arenas Dalla-Vecchia

Mayo de 1999

ÍNDICE

1. INTRODUCCIÓN

En este proyecto hemos abordado el estudio de un entorno CACSD (*Computer Aided Control System Design)* para el diseño e implementación de sistemas de control. Está formado por un lenguaje de simulación dotado de una amplia biblioteca de bloques (SIMULINK [12]), una herramienta de generación automática de programas de control (Real Time Workshop [13]), y de un interfaz para la adaptación de estos programas a un hardware específico (Real Time Interface [4]). El hardware sobre el que se van a implementar los controladores consiste de un procesador digital de señal, junto con canales analógicos/digitales y digitales/analógicos, así como otros subsistemas, que conforman la tarjeta de control dSPACE 1102 ([6]). El objetivo del proyecto es la puesta en marcha de las diferentes herramientas, así como su análisis y aplicación a un caso práctico.

Esta herramienta CACSD facilita en teoría la labor al ingeniero de control, al permitirle automatizar diferentes pasos del proceso de diseño. Resaltaremos dos aspectos fundamentales, que han motivado la realización del proyecto:

- El primero es que le permite describir y simular en un lenguaje de muy alto nivel la estructura de los sistemas con los que trabaja. Se trata de un lenguaje a nivel de diagramas de bloque, editable en un entorno visual, con todas las facilidades de cara a la edición que caracterizan a este tipo de entornos. Este lenguaje incorpora librerías de bloques básicos predefinidos utilizados con frecuencia, así como la posibilidad de crear nuevas librerías o expandir las ya existentes, de modo que se fomenta la reutilización de componentes. Además permite la jerarquización de bloques, con lo que se facilita la estructuración de los modelos.

- El segundo, es que le permite la generación automática, a partir de la descripción de un controlador en ese lenguaje de alto nivel, del programa de control asociado, ejecutable directamente sobre un procesador específico. En particular, se puede generar un ejecutable para el DSP (Digital Signal Processor) que incorpora la tarjeta de control dSPACE DS 1102, que puede ser conectada

directamente a la planta a controlar. Con esta generación automática de controladores ejecutables a partir de la descripción del control a implementar a nivel de diagrama de bloques se consigue un notable ahorro de tiempo y se evitan totalmente los errores de programación. Esta rapidez en la generación del programa permite al diseñador probar sobre la planta real sucesivas modificaciones (refinamientos) sobre el controlador en un tiempo mínimo, lo que se conoce como *prototipado rápido de controladores* RPC (Rapid Control Prototyping). Además, la Tarjeta de Control puede interaccionar con el PC a la vez que ejecuta un controlador, lo que permite realizar tareas de monitorización e incluso modificar parámetros del controlador desde del PC sin detener el control. Esto constituye una herramienta muy potente de cara a la sintonización de parámetros.

El lenguaje de descripción de sistemas a nivel de bloques mencionado es el formalismo gráfico utilizado por *SIMULINK*, que es un paquete de software para definición, simulación y análisis de sistemas dinámicos. SIMULINK está integrado en *MATLAB*, que actúa como motor de cálculo de las simulaciones desarrolladas en SIMULINK. MATLAB es un lenguaje interactivo para cálculo numérico, especializado en cálculo matricial, y que facilita mucho al usuario la programación de este tipo de algoritmos. Esto tanto por su sintaxis y estructuras de tipos, especialmente adecuadas para estos propósitos, como por incorporar un gran número de funciones predefinidas que realizan cálculos complejos frecuentemente utilizados. La potencia de MATLAB se ve ampliada por la incorporación de librerías de funciones muy especializadas en ciertos temas, como puedan ser robótica, control, proceso de señales, lógica fuzzy, etc. Estas librerías se denominan *toolboxes* (literalmente, “cajas de herramientas”).

En base a un modelo de una planta se podría diseñar y simular un controlador directamente en SIMULINK, sin necesidad de generar un programa a partir del controlador para realizar la simulación: SIMULINK, interactuando con MATLAB, se encarga de ello. Pero si además queremos probar el controlador sobre la planta real, es necesario generar un código que se ejecute sobre un procesador (en nuestro caso el de la Tarjeta de Control) para actuar como controlador sobre la planta. Es aquí donde entra en juego el proceso de generación automática de código mencionado. Su piedra angular el programa *RTW* (*Real Time Workshop*), capaz de generar a partir de un modelo en

SIMULINK un programa ejecutable cuya ejecución reproduce en tiempo real el comportamiento del modelo SIMULINK. El código generado es específico del sistema sobre el que se vaya a ejecutar. RTW genera un fichero C como paso intermedio entre el modelo SIMULINK y el ejecutable. La generación del ejecutable requiere de algunos ficheros de configuración que indican a RTW las características del hardware destino, un compilador de C que genere código para ese hardware a partir del fichero intermedio, y posiblemente algunas librerías particulares de ese hardware que el compilador enlazará al programa principal para generar el ejecutable. Este esquema está abierto a la creación de código para numerosas plataformas distintas de destino, siempre que se suministren a RTW estos elementos específicos de esa plataforma.

A grandes rasgos, el proceso de generación automática de código descrito hasta ahora sería así:

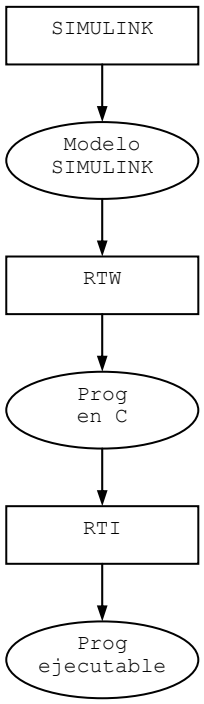

Fig. 1: cadena de generación automática de código a partir de modelos SIMULINK

Las potencialidades clave de este entorno han sido ya descritas. Pero además nos brinda un funcionalidad muy útil en el proceso de diseño de controladores basado en modelos, que no por menos novedosa es menos importante. Para diseñar un control sobre un modelo precisamente necesitamos partir de un modelo de la planta a controlar. Si no lo tenemos, antes del diseño del controlador es necesaria una etapa de

identificación de la planta. Este proceso consiste en aplicarle una señal de control con unas ciertas características en deducir de la respuesta obtenida qué modelo podría aproximar el comportamiento del sistema.

En el Capítulo 3, se realiza un análisis de los aspectos más importantes de SIMULINK y RTW. En el Capítulo 4, se describe con detalle la tarjeta de control dSPACE 1102, sobre la cual se ejecutará el programa de control, así como las características más importantes del RTI, siempre desde el punto de vista de control. Finalmente, en el Capítulo 5 se presenta un caso práctico de diseño, el control de un péndulo invertido, basado en las herramientas anteriormente descritas.

2. SIMULINK Y REAL TIME WORKSHOP

2.1. Definición de sistemas con SIMULINK

Como hemos comentado, SIMULINK es un paquete de software para definición, simulación y análisis de sistemas dinámicos. En este proyecto sólo nos vamos a interesar por la definición de sistemas y su posterior simulación.

Para definir sistemas, SIMULINK proporciona un interfaz gráfico en el que el usuario puede editar los modelos en forma de diagramas de bloques interconectados por líneas que representan señales. Los bloques pueden corresponder a sistemas con entradas y salidas, a fuentes de señales (como un generador de onda cuadrada) o ser "sumideros" de señales: bloques que reciben una señal y hacen algo con ella, como por ejemplo representarla de cara al usuario frente al tiempo (bloque *scope)*, pero no tienen ninguna salida. El usuario parte de unas librerías de bloques básicos parametrizados con los que construir sus modelos. Para conseguir incorporar a nuestro modelo una instancia de un bloque de una librería tan sólo es necesario arrastrar el icono de ese bloque desde la ventana de la librería en la que se encuentra hasta la ventana en que estamos construyendo nuestro modelo. Por ejemplo, sobre un modelo vacío, que llamaremos "modelo_1", insertemos dos instancias del bloque *sine wave* ("onda senoidal", de la librería *sources*, "fuentes"), una del componente *sum* ("suma", de la librería *linear*, "lineal") y por último otra del ya mencionado *scope* (de la librería *sinks*, "sumideros"):

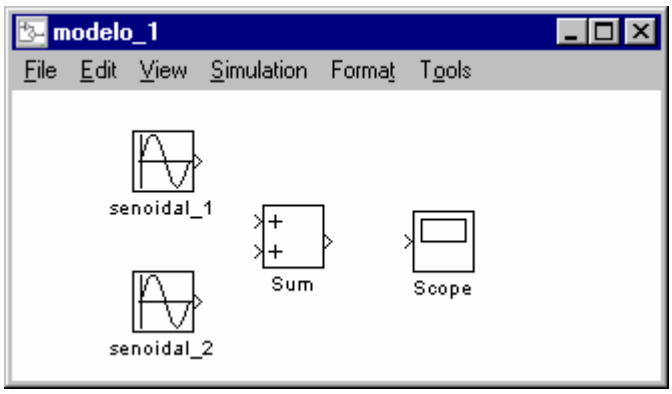

Fig. 2: algunos bloques SIMULINK para construir un modelo de ejemplo

Haciendo doble click sobre las instancias de los bloques podemos configurar sus parámetros. Configuraremos las fuentes de ondas senoidales de modo que *senoidal_1*

tenga una frecuencia de 1 Hz y *senoidal_2* 10 Hz. Vemos que ambas fuentes generan una señal (tienen una flecha saliente), que el bloque *scope* recibe una señal (flecha entrante) y que el bloque *sum* recibe dos señales y devuelve una (realiza la suma de las dos señales que recibe).

Para indicar a SIMULINK que la salida de un bloque queda conectada a la entrada de otro no hay más que arrastrar con el ratón la flecha de salida hasta la de entrada. Aparece una línea desde la salida a la entrada, que termina en una flecha en la entrada. Esta línea es una *señal*, y se le pude asignar una etiqueta. En el ejemplo anterior, realizamos las siguientes conexiones y asignaciones de etiquetas:

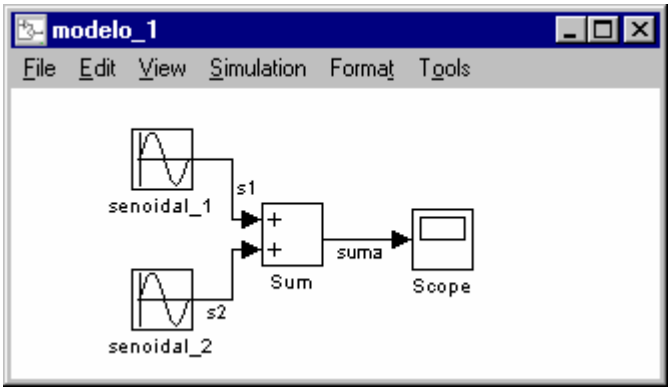

Fig. 3: conexión de los bloques para el ejemplo de modelo SIMULINK

SIMULINK cuenta con una librería de bloques bastante completa, que incluye bloques para sistemas continuos, discretos, muestreados, lineales y no lineales, etc. Aun así, si el usuario necesita nuevos bloques siempre puede construirlos él mismo y añadirlos a alguna librería o incluso crear nuevas librerías para ellos (lo que facilita la reutilización).

Una característica muy importante de la definición de sistemas en SIMULINK es que se permite crear una jerarquía de bloques mediante el concepto de *subsistema*. Un subsistema es un bloque que a su vez agrupa en su interior una serie de bloques que constituyen por sí solos un modelo de un sistema. Eso permite diseñar los modelos en base a una estructura de capas superpuestas de niveles de abstracción. Y esto en las dos direcciones posibles: de abajo hacia arriba (diseño *botton-up*), agrupando un modelo que ya hemos creado para usarlo como bloque en un modelo de mayor nivel de abstracción; o de arriba hacia abajo (diseño *top-down*), empezando a diseñar el nivel

superior de abstracción en base a bloques indefinidos, cuya implementación no se especifica en principio, sino que se hará en una etapa posterior de refinamiento, al describir el siguiente nivel de abstracción. Veamos como ejemplo la encapsulación de nuestras dos fuentes senoidales y el bloque que sumaba sus señales en un solo bloque que llamaremos *2_senoidales*. A la izquierda, cómo queda ahora *modelo_1*, y a la derecha, la expansión del bloque *2_senoidales*:

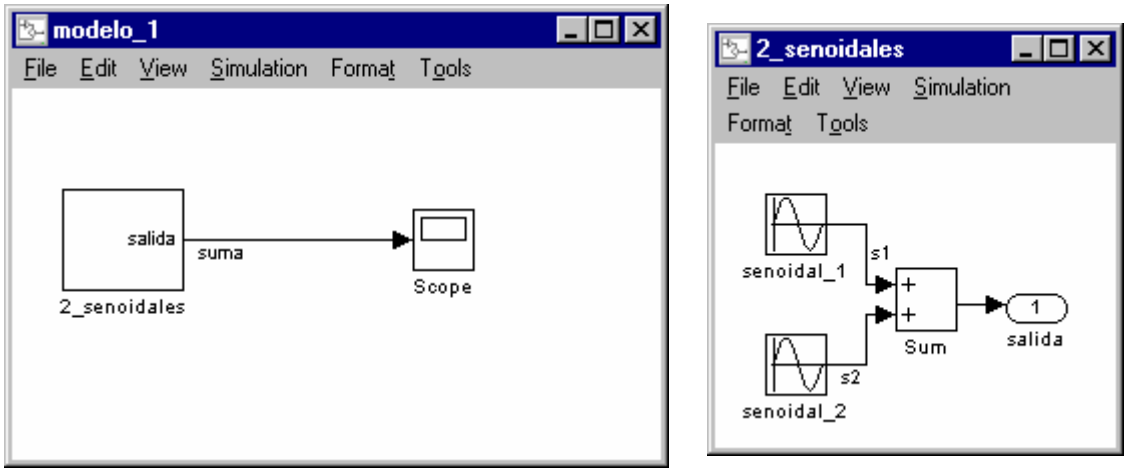

Fig. 4: jerarquización de bloques mediante el uso de subsistemas

Vemos en la ventana de la derecha que dentro del subsistema la salida del bloque *Sum* es entrada de un bloque etiquetado como *salida*. Dentro de un subsistema expandido este tipo de bloques, denominados "out" (salida), tienen la misión de sacar una señal interna del subsistema al exterior, de modo que la señal interna se convertirá en una de las señales de salida del subsistema considerado como bloque. Vemos que la etiqueta del bloque de salida coincide con la que tiene la salida del subsistema en el diagrama de la izquierda. Esta etiqueta da la correspondencia entre señales internas y externas en el subsistema. Se pueden asociar de forma parecida señales de entrada.

No comentaremos todos y cada uno de los bloques disponibles en las librerías de SIMULINK, sino que sólo explicaremos en qué consisten aquellos que se vayan utilizando a lo largo de la exposición del presente proyecto, a medida que sean necesarios. Sin embargo merece la pena mencionar aquí cuatro porque ponen claramente de relieve la fuerte interacción que hay entre SIMULINK y MATLAB:

- Dos fuentes de señal: *from file* (desde fichero) y *from workspace* (desde el espacio de trabajo). El primero genera una señal que corresponde a una

secuencia de parejas (tiempo, magnitud señal) leída de un fichero generado por MATLAB (interacción a través de fichero). La segunda directamente lee los valores de una variable del espacio de trabajo de MATLAB (interacción directa, por compartición de memoria).

- Dos sumideros de señal: *to file* (hacia fichero) y *to workspace* (hacia el espacio de trabajo). Hacen lo propio, pero en sentido contrario: leen la señal que les llega en el modelo SIMULINK y la envían respectivamente a un fichero MATLAB o directamente a una variable del espacio de trabajo MATLAB.

2.2. Simulación de sistemas con SIMULINK

Una simulación de un modelo SIMULINK puede llevarse a cabo mediante la invocación de comandos en modo texto desde la ventana de MATLAB o bien en modo interactivo desde los menús del interfaz gráfico del propio SIMULINK. Describiremos esta segunda opción, ya que es más intuitiva y es suficiente para nuestro propósito.

Antes de ordenar a SIMULINK que simule nuestro modelo de sistema, es necesario configurar una serie de opciones que le indiquen cómo hacerlo. Las únicas que nos van a interesar en principio son las que se encuentran en la pestaña *solver* (resolutor) del menú *simulation/parameters*. En esta pantalla elegimos básicamente dos cosas: el tiempo de simulación y el resolutor (o motor de simulación) a utilizar, junto con los valores de los parámetros asociados al resolutor elegido (el conjunto de parámetros es distinto para uno).

El tiempo de simulación (*simulation time*) establece el tiempo de comienzo y fin de la simulación. Tiempo en términos de la propia simulación, que no coincide con el transcurso del tiempo real: si los cálculos implicados en la simulación son sencillos, simular 20 segundos de experimento puede tardar tan sólo 1 segundo. O al revés si los cálculos son muy complejos.

En cuanto a la elección del resolutor, ésta dependerá del tipo concreto de modelo con que estemos tratando. Debido a la diversidad de comportamientos de los sistemas dinámicos, SIMULINK nos da a elegir entre una amplia gama de resolutores especializados en un tipo de comportamiento particular. Una primera cuestión a elegir

sobre el resolutor es si será de paso fijo o de paso variable. Los de paso variable pueden modificar durante la simulación su paso, lo que les permite ciertas ventajas como la posibilidad de detectar los cortes con cero de una señal. Los de paso fijo no aportan esas posibilidades, pero son los únicos que podemos elegir si queremos traducir el modelo a un programa ejecutable, por lo que nos centraremos en ellos. Para un resolutor de paso fijo sólo hay un parámetro asociado a elegir: el tamaño de paso fijo (*fixed step size*).

Al margen de que usemos un resolutor de paso fijo o variable, nuestro modelo puede tener o no estados continuos. Si no los tiene podemos usar el resolutor *discrete*, que no lleva a cabo ningún tipo de integración, y que está disponible en las dos versiones (paso fijo y paso variable). Si nuestro modelo sí contiene estados continuos, sí que necesita integración, por lo que será necesario elegir un resolutor no discreto. Dentro de éstos se nos ofrecen múltiples opciones, entre las cuales hay uno genérico, que funciona bien con la mayoría de los problemas, que se llama *ode45* en el caso de paso variable y *ode5* en el caso de paso fijo. Está basado en la fórmula explícita Runge-Kutta (4,5) [12]. *ODE* es el acrónimo de *Ordinary Differential Equations* (resolutor de Ecuaciones Diferenciales Ordinarias).

Una vez que hemos visto cómo se configura una simulación, haremos una de ejemplo sobre el modelo que hemos diseñado en el apartado 2.1. Mostramos para los parámetros elegidos qué aspecto tiene la ventana en que se hace esta elección:

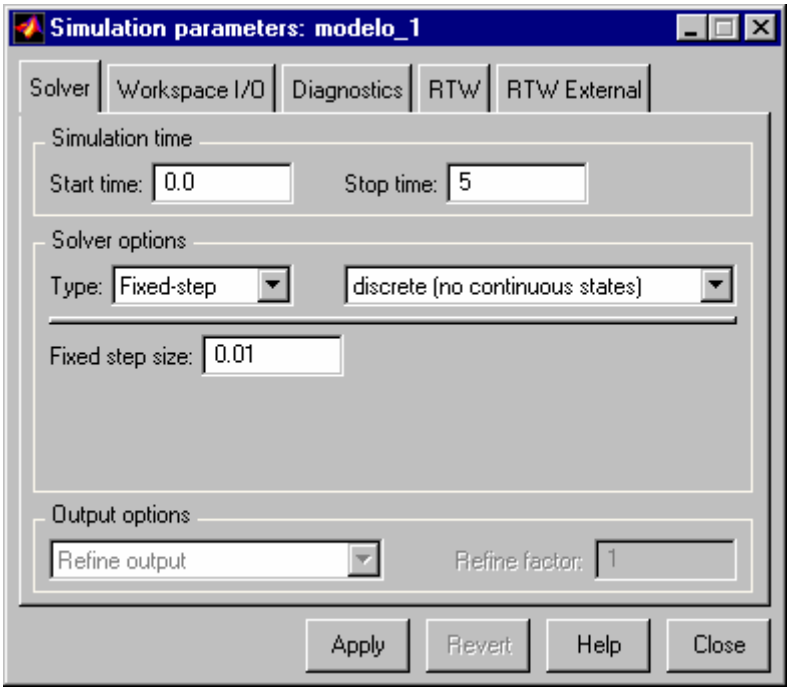

Fig. 5: ventana de selección de parámetros del resolutor de SIMULINK

Hemos tomado como tiempos de comienzo y fin de la simulación 0 y 5 segundos respectivamente, como resolutor el discreto de paso fijo (ya que no hay estados continuos en nuestro modelo), y como tamaño de paso 0.01 segundos.

Para ejecutar la simulación elegimos el elemento de menú *simulation/start*. Tras terminar de ejecutarse, haciendo doble click sobre el bloque *scope* podemos observar el la señal simulada resultante de la suma de las dos senoidales:

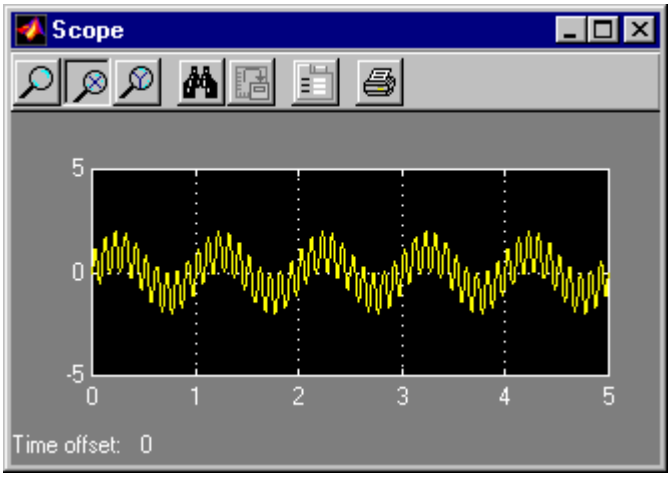

Fig. 6: simulación de la suma de dos senoidales, una de frecuencia diez veces la de la otra

Hay una serie de parámetros de los bloques que pueden ser cambiados interactivamente mientras se ejecuta la simulación, de modo que podemos observar el efecto de estos cambios inmediatamente y de esa forma por ejemplo llevar a cabo un ajuste de parámetros. Los cambios deben ser tales que no modifiquen la estructura del modelo. Por ejemplo, no podemos desconectar una señal de un bloque, o modificar el tamaño de paso de simulación sin interrumpir ésta. Pero si podemos, por ejemplo, cambiar la amplitud de una de las senoidales. Veamos la señal resultante de dividir por 5 la amplitud de la senoidal de mayor frecuencia a mitad de simulación:

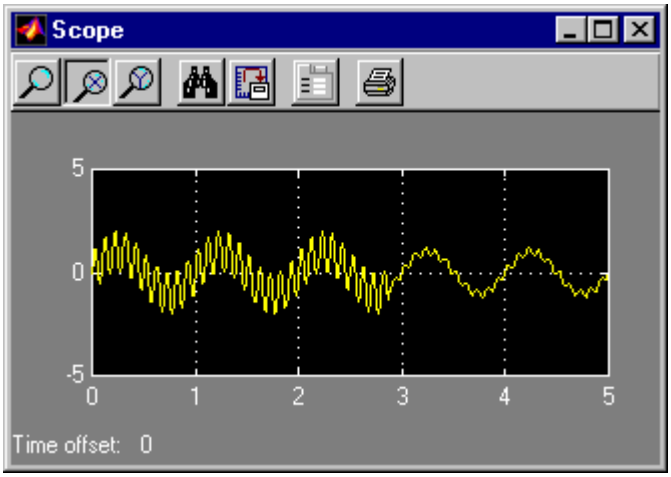

Fig. 7: modificación de la amplitud de la senodal de mayor frecuencia durante la simulación

2.3. Generación automática de programas de tiempo real mediante RTW

No entraremos en una descripción detallada de todas sus posibilidades, pero sí ha de quedar claro que es un programa que traduce modelos SIMULINK/MATLAB a un programa ejecutable en un hardware específico. RTW está basado en una arquitectura abierta que permite que este hardware de destino sea configurable. Además, está diseñado para recibir una variedad amplia de aplicaciones, al margen de que en el presente proyecto nos centremos en el desarrollo de controladores:

- Control en tiempo real: se puede diseñar un controlador utilizando MATLAB y SIMULINK y generar código a partir de su modelo en forma de diagrama de bloques. Ese código puede ser compilado y cargado directamente al hardware de destino. Este es el uso que vamos a dar a RTW en este proyecto.

- Procesado de señales en tiempo real: se puede diseñar un algoritmo de procesado de señales mediante MATLAB y SIMULINK, y generar/cargar con RTW el código ejecutable de la misma forma que se hace para un controlador.

- Simulaciones HIL (hardware-in-the-loop): se puede generar con MATLAB y SIMULINK un modelo de comportamiento de un sistema real, que lo imite. Si compilamos ese modelo con RTW y lo ejecutamos sobre el hardware de destino, éste se comportará de cara al exterior como ese modelo, de forma que podremos utilizarlo para aplicaciones como validación de sistemas de control, simulaciones de cara a entrenamiento de personas (por ejemplo de pilotos), tests de fatiga de materiales, etc.

- Sintonizado de parámetros interactivo en tiempo real: se puede usar SIMULINK como un panel que nos permita interactuar con el programa en ejecución en tiempo real. Esto nos permite cambiar parámetros del programa en ejecución y observar directamente, en tiempo real, los efectos de esos cambios. En este proyecto no usaremos esta posibilidad de RTW, sino que utilizaremos uno de los programas que acompañan a la tarjeta de control con

la que hemos trabajado, COCKPIT, que nos da esta posibilidad de una forma más sencilla y con un interfaz más agradable e intuitivo.

- Simulaciones independientes a alta velocidad: RTW puede generar ejecutables que simulen el comportamiento de un modelo no en tiempo real (corriendo en procesadores especializados en ejecución en tiempo real) sino a alta velocidad, corriendo en procesadores especializados en obtener altas velocidades de cálculo.

- Generación de código C portable, para exportar a otros programas de simulación.

Para generar un ejecutable con RTW a partir de un modelo SIMULINK es necesario configurar previamente una serie de parámetros. Esto se hace desde la ventana SIMULINK en que tenemos el modelo que vamos a compilar. En primer lugar, hay que indicar, de la misma forma que se hacía para una simulación, los parámetros de la pestaña *solver* de la ventana *simulation/parameters*. Estos parámetros afectan de igual forma a una simulación que se active a partir de ese momento como al código generado si se invoca a RTW. Por otra parte, hay que indicar a RTW cómo generar el código para el hardware específico que elijamos. Esto se hace en la pestaña *RTW*. La instalación del *Real Time Interface* que acompaña a la tarjeta de control configura por defecto los parámetros de esta ventana para que el código generado sea para ejecutarse en la tarjeta:

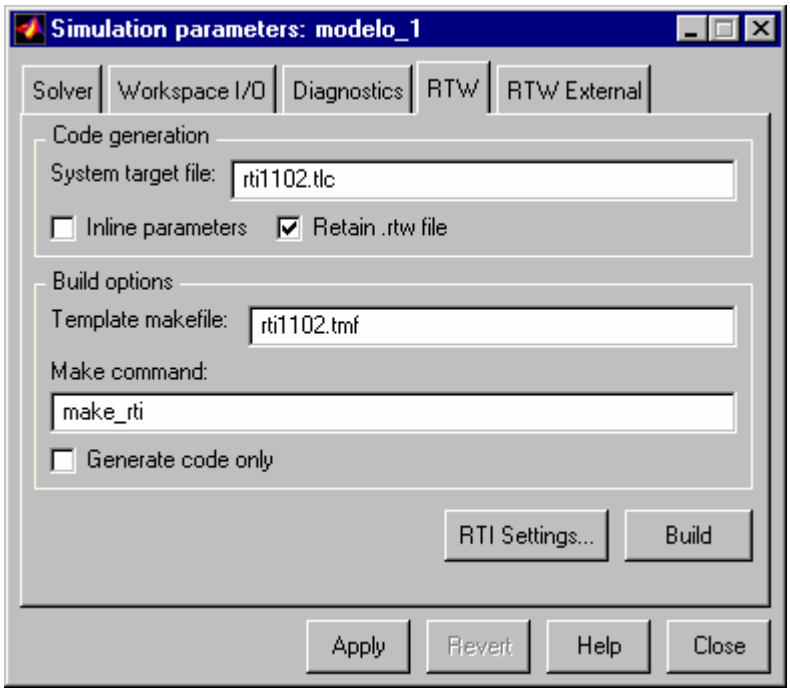

Fig. 8: ventana de parámetros del Real Time Workshop

No entraremos a describir el cometido de cada uno de los ficheros de los campos de edición de texto. Son específicos de la tarjeta (todos llevan la cadena "RTI" como prefijo o sufijo), los ha diseñado el fabricante de la tarjeta e indican a RTW cómo generar código para ella a partir de un modelo. La opción *generate code only*, cuando está habilitada, hace que RTW no cargue y ejecute el programa en el hardware específico una vez que lo ha compilado.

La pestaña *RTW External* permite configurar la opción de que SIMULINK actúe como fachada del programa que corre en la tarjeta de cara a la sintonización interactiva de parámetros, pero como ya se ha dicho no utilizaremos esta opción en este proyecto.

Una vez configurados los parámetros necesarios, la generación de código es tan simple como elegir en el menú de nuestro modelo SIMULINK el comando *tools/rtwbuild*. Esto desencadena todo el proceso de traducción, que culmina con la carga y ejecución d a la tarjeta. En la ventana de MATLAB van apareciendo los siguientes mensajes:

```
*** Starting RTI build procedure with RTI1102 3.1 (23-Apr-1998)

### Starting RTW build procedure for model: modelo_1
### Invoking Target Language Compiler on modelo_1.RTW
tlc -r modelo_1.RTW d:\dsp_cit\MATLAB\rti1102\tlc\rti1102.tlc -O. -
Id:\dsp_cit\MATLAB\rti1102\tlc -IC:\MATLAB\RTW\c\tlc -aInlineParameters=0
### Creating project marker file: RTW_proj.tmw
### Creating modelo_1.mk from rti1102.tmf
### Building modelo_1: dsmake -f modelo_1.mk

BUILDING PROGRAM (single timer task mode)

Initial SimState: default (defined in srtframe.c)
[srtframe.c]
[modelo_1.c]
[rt_sim.c]
[rt_hypot.c]
[rt_look.c]
[rt_matrx.c]
[rt_rand.c]
[rt_sgn.c]
[rt_zcfcn.c]

LINKING PROGRAM ...

LOADING PROGRAM ...

LD31 - DS1102 Controller Board Loader, Vs 3.4 - 32, (C) 1997 by dSPACE GmbH

Loading object module MODELO_1.OBJ ...

DSP started ...
DOWNLOAD SUCCEEDED

### Successful completion of RTW build procedure for model: modelo_1

*** Finished RTI build procedure for model modelo_1
```

Fig. 9: mensajes emitidos por RTW durante la generación automática de un ejecutable

Para ver que efectivamente el programa ha sido cargado a la tarjeta y está ejecutándose, recurriremos a TRACE [3], una de las herramientas software que acompañan a la tarjeta, y que será introducida con más detalle en el apartado siguiente. La vamos a utilizar a modo de osciloscopio para estudiar los valores que va adoptando la salida del bloque "2_senoidales":

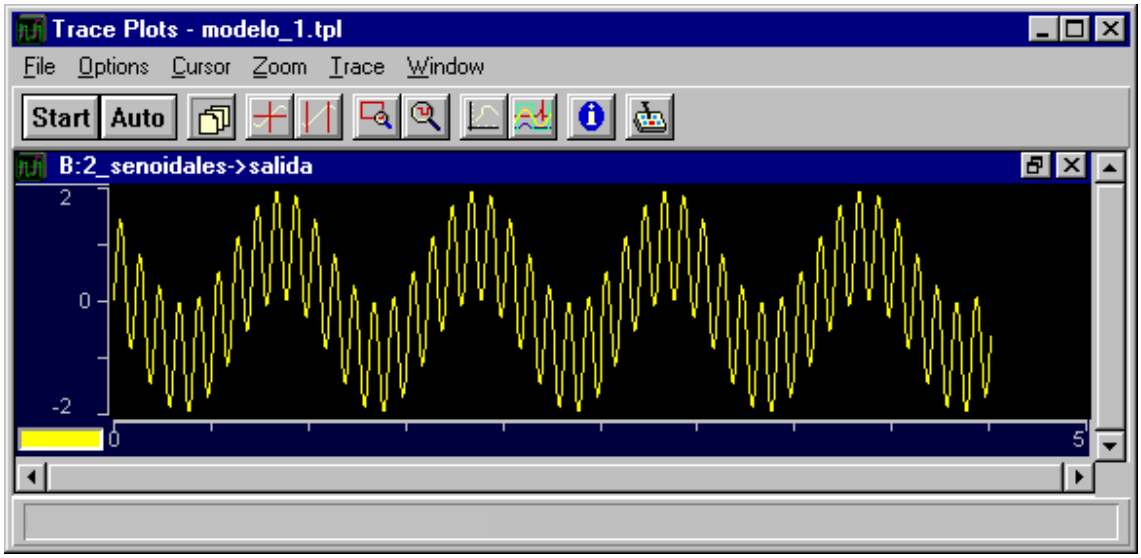

Fig.10: señal resultante de la ejecución en la tarjeta del programa que suma las dos senoidales (monitorizada con TRACE)

3. TARJETA DE CONTROL DSPACE DS 1102

3.1. Arquitectura y funcionalidades básicas.

La DS 1102 [1,5] es una tarjeta específicamente diseñada para el desarrollo de controladores y simulaciones en tiempo real en campos diversos como robótica, actuadores servohidráulicos y eléctricos, control de vehículos, etc. También se ajusta bien al procesado genérico de señales y tareas relacionadas.

Está basada en el DSP (Digital Signal Processor) de punto flotante Texas Instruments TMS320C1, que constituye su unidad principal de proceso. Este procesador proporciona un ciclo rápido de instrucción para algoritmos numéricos intensivos. La potencia del procesador se ve reforzada por una memoria externa que es lo suficientemente rápida para permitir que el procesador nunca deba esperarla ni un solo ciclo al realizar una lectura o escritura. Además esta memoria puede ser accedida desde el host mientras el programa de tiempo real está en ejecución, sin interrumpir la actividad del DSP ni interferir con ella, posibilitando de esta forma una fácil monitorización de interacción con el programa en ejecución. Esto es gracias a que se trata de memoria de doble puerto.

Alrededor del procesador, bajo su control, se sitúan los periféricos que posibilitan la interacción de la tarjeta con el sistema al que se conecta. Para empezar cuenta con cuatro DAC's y cuatro ADC's, directamente controlados por el procesador. También cuenta con dos interfaces de incremental encoder, capaces de leer las señales en forma de tren de pulsos generadas por estos dispositivos y convertirlas directamente, mediante hardware, en un valor de ángulo. Utilizaremos estos interfaces en la identificación y el control del péndulo invertido. Por último, cuenta con un subsistema de entrada salida digital gobernado por un procesador auxiliar, el Texas Instruments TMS320P14. El procesador auxiliar se encarga directamente de la gestión de los periféricos incluidos en este subsistema, descargando así al procesador principal de esta tarea. La forma en que actúa este procesador auxiliar viene determinada por el programa que ejecuta, que puede ser el básico (por defecto), que lee de una EPROM incluida en la

tarjeta, o bien uno distinto. El programa básico permite el acceso a los periféricos (bits de E/S digital, un puerto serie, un subsistema de comparación de bits y captura de eventos) y poco más. Para conseguir comportamientos más sofisticados se puede cargar sobre este procesador un programa que le haga realizar una tarea concreta más elaborada. La tarjeta viene acompañada de una serie de programas para el procesador auxiliar que implementan algunos de estos comportamientos, utilizados frecuentemente, como pueda ser la medición de frecuencias, la generación de PWM's (que explicaremos y utilizaremos en el control del péndulo), etc.

El esquema de la arquitectura básica de la tarjeta sería algo así:

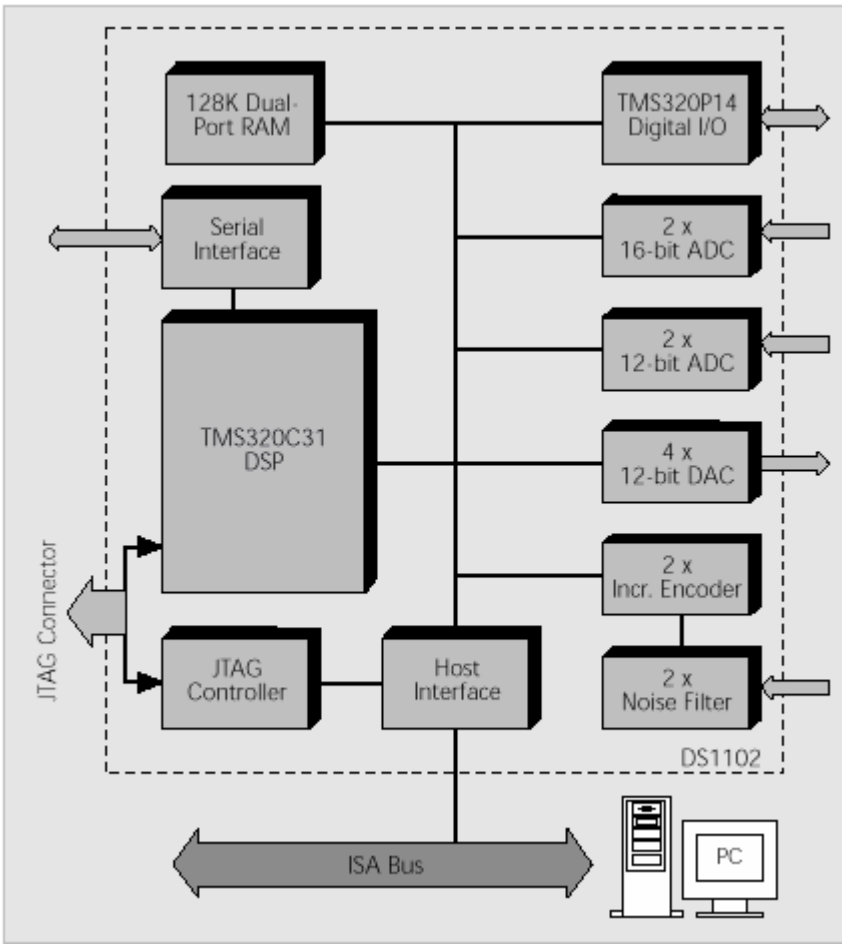

Fig. 11: arquitectura básica de la tarjeta dSPACE DS1102

En él podemos observar dos bloques que no han sido comentados anteriormente, por carecer de relevancia a efectos del presente proyecto: un puerto serie que permite conectar varias tarjetas de dSPACE entre sí y un interfaz JTAG, que implementa un superconjunto del estándar de puerto de emulación JTAG IEEE 1149.1.

3.2. Programación directa de la tarjeta en C (programación tradicional).

La tarjeta dSPACE 1002 puede ser programada, además de mediante la generación automática de código, mediante la programación directa tradicional en C [7]. Veamos un ejemplo sencillo de este tipo de programación, que nos permitirá apreciar las ventajas de la programación automática a partir de modelos en SIMULINK, al tiempo que veremos las herramientas software que proporciona el fabricante para permitir este tipo de programación. Se trata del programa de ejemplo *trctst31.c*, que acompaña a los manuales de la tarjeta. Se han hecho algunos recortes en el código original para focalizar la atención del lector en la estructura del algoritmo de control.

Este programa desarrolla la simulación de un sistema de segundo orden muelle-masa-amortiguador, usando el método de integración de Euler. Se basa en la generación periódica de interrupciones para conseguir un periodo de muestreo regular. La variable de entrada se tomará del primer conversor AD, y la salida se volcará al primer conversor DA. Como veremos en el código, en este tipo de programación se realiza a un nivel de abstracción mucho menor que la programación SIMULINK vía RTW: hay que acceder explícitamente a los conversores AD/DA, hay que programar explícitamente el esquema de generación de interrupciones periódicas para conseguir el periodo de muestreo deseado, hay que implementar el método de integración que se desee utilizar, inicializar explícitamente las variables, tratar explícitamente con estados continuos / discretos de los sistemas simulados, etc. Veamos el código, para a continuación comentar qué hace cada parte:

```
#include <brtenv.h>                     /* entorno de tiempo real básico */

#define DT 1.0e-4                       /* tamaño de paso de simulación: 100 us */
#define NSTATES 2                       /* nº de estados */
float x[NSTATES], x_dot[NSTATES];       /* matrices de estado */

#define D_DEF 16.0                      /* amortiguamiento inicial */
#define C_DEF 20000.0                   /* coeficiente inicial del muelle */
#define M_DEF 0.04                      /* masa inicial */

float *d = (float *) (DP_MEM_BASE);     /* amortiguamiento */
float *c = (float *) (DP_MEM_BASE + 1); /* coeficiente del muelle */
float *m = (float *) (DP_MEM_BASE + 2); /* masa */

float u = 0.5;                          /* valor de entrada */
```

```
/* flag de error para CHKERR en la última posición de la memoria de puerto dual */
volatile int *error = (int *) (DP_MEM_BASE + DP_MEM_SIZE - 1);

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

void derivatives()
{
  float fc, fd;

  fd       = *d * x[1];
  fc       = *c * (x[0] - u);
  x_dot[0] = x[1];
  x_dot[1] = (-fd - fc) / *m;
}

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

void integrate_with_euler()
{
  int i;

  derivatives();
  for (i = 0; i < NSTATES; i++)
    x[i] = x[i] + x_dot[i] * DT;
}

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

void isr_t0()                       /* rutina de servicio a la interrupción timer0 */
{
  /* entrada: leer ADC #1... */
  ds1102_ad_start();
  u = ds1102_ad(1)

  /* salida: escribir ADC #1... */
  ds1102_da(1,x[0])

  integrate_with_euler();           /* actualización de las matrices de estado */
}

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

main()
{
  int i;

  *d = D_DEF;                                  /* inicializar parámetros */
  *c = C_DEF;
  *m = M_DEF;
  for (i = 0; i < NSTATES; i++)                /* resetear variables de estado */
    x[i] = 0.0;

  init();                                      /* inicialización del hardware */

  start_isr_t0(DT);                            /* initializar periodo muestreo timer */

  while (1);                                   /* proceso background */
}
```

Fig. 12: código C del ejemplo de programación tradicional de la tarjeta

Empecemos por la función main, al final del código. En ella se inicializan los parámetros del sistema de segundo orden: amortiguamiento, coeficiente del muelle y masa, y se resetean las variables de estado. Se diferencia entre valores iniciales y actuales de los parámetros porque sería posible acceder a estas variables en tiempo de ejecución desde el PC para observar el efecto de un cambio en sus valores. A continuación se invoca a la función *init()*, que forma parte de las librerías C específicas

de la tarjeta aportadas por el fabricante, e incluidas mediante la directiva `#include <brtenv.h>`, situada al principio del fichero. Esta función se encarga de ciertas tareas de inicialización del hardware que hay que realizar siempre antes de comenzar cualquier actividad con la tarjeta, como son puesta a cero de los bits de interrupción de la tarjeta, inicialización del rango de funcionamiento de los conversores DA, etc.

A continuación se invoca a la función *start_isr_t0(DT)*. Esta función también forma parte de las librerías de la tarjeta. El procesador principal de la tarjeta cuenta con dos timers capaz de generar interrupciones periódicas, *timer0* y *timer1*. Esta función inicializa el *timer0* para que genere interrupciones cada *DT* segundos. Implícitamente, la rutina de servicio a esta interrupción se llama *isr_to()*. Por último, el programa principal queda en un bucle infinito que en este caso no hace nada, pero que se podría ocupar de tareas que no necesitan de un periodo de ejecución fijo, como por ejemplo monitorización.

Según lo visto hasta ahora, el procesador de la tarjeta no hará nada, salvo en los instantes periódicos en que se active la interrupción del timer 0. En esas ocasiones se invoca a la función *isr_t0()*. Veamos lo que hace. En las dos primeras líneas se lee sobre la variable *u* la señal que llega al conversor AD número 1. Esta operación se realiza en dos instrucciones, que invocan a sendas funciones de la librería: la primera activa la conversión AD, mientras que la segunda se mantiene a la espera de que termine la conversión, y cuando termina devuelve el valor recién convertido. Tras leer la variable *u*, se escribe la salida correspondiente a ese periodo de muestreo, que en este caso es simplemente la primera variable de estado. Por último se llama a la función *integrate_with_euler*, que recalcula el valor de las variables de estado en conjunción con la función *derivatives()*.

El proceso de compilación y carga a la tarjeta de un programa en C para la tarjeta queda automatizado por el fichero batch *down31* [7], suministrado por el fabricante. La carga de un determinado ejecutable ya compilado se realiza mediante el programa *ld31k* [11], también suministrado por el fabricante. Este programa permite cargar tanto el ejecutable para el procesador principal como el del auxiliar, así como parar / reanudar el proceso activo.

3.3. Programación desde MATLAB/SIMULINK.

El objetivo que vamos persiguiendo es conseguir la traducción automática de un modelo SIMULINK a un programa que se ejecute sobre la tarjeta. Para ello es necesario especificar en el modelo qué elementos funcionales de entrada / salida de la tarjeta vamos a utilizar en el modelo, especificando en el caso de aquellos que se encuentran duplicados en la tarjeta el número de elemento elegido. Esto establece la correspondencia entre las señales del modelo SIMULINK y las entradas y salidas de la tarjeta.

Para conseguir esta finalidad el RTI [4] de la tarjeta proporciona una nueva librería de bloques SIMULINK que representan los distintos elementos de E/S de la tarjeta. Por ejemplo, tenemos un par de bloques que representan, respectivamente, las cuatro entradas y las cuatro salidas analógicas de la tarjeta:

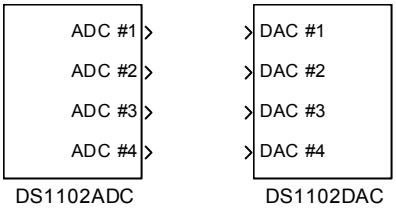

Fig. 13: bloques ADC y DAC del Real Time Interface

Si incluimos el bloque de la izquierda en un modelo SIMULINK y conectamos la salida *ADC #1* a la entrada de otro bloque, llamémoslo *B*, en el programa compilado mediante RTW y ejecutado en la tarjeta el bloque *B* recibe como entrada la señal física que captura el conversor AD número 1. Otro tanto ocurre con el bloque de la derecha: si su entrada *DAC #1* recibe la salida de un bloque *B*, en el programa compilado la salida de *B* sale de la tarjeta por el conversor DA número 1. Estos dos bloques no tienen parámetros que elegir. Cuando se usa el bloque de entradas ADC, aquellas no utilizadas han de ser conectadas a bloques *terminador* (*terminator*), para indicar de cara a la compilación que esas entradas pueden ser ignoradas. En el caso de las salidas DAC las no utilizadas se han de conectar a *tierra* (*ground*). Estos bloques pertenecen a la librería *connections* de SIMULINK, no al RTI de la tarjeta:

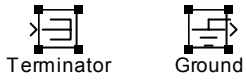

Fig. 14: boques *terminador* y *tierra*

Veamos el primer ejemplo de utilización de estos bloques en un modelo SIMULINK: vamos a sumar las señales que se reciben por las entradas AD 1 y 2, y el resultado va a salir a través de la salida DA 1. El resto de entradas se conectan a un terminador, y el resto de salidas a tierra:

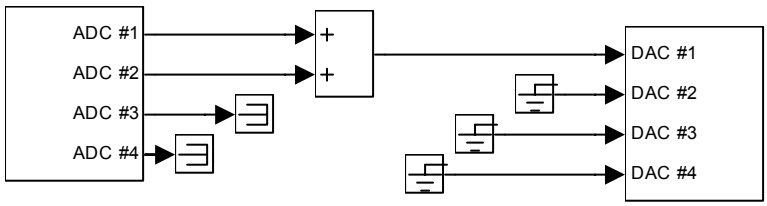

Fig. 15: ejemplo de digarma SIMULINK que utiliza bloques RTI de la tarjeta

Además de los bloques SIMULINK correspondientes a los conversores AD / DA, describiremos aquí algunos más que serán utilizados más adelante...

Bloque *PWM*: el subsistema de E/S digital de la tarjeta es capaz de generar mediante la carga de un programa determinado en el procesador auxiliar hasta seis PWM's independientes (para una descripción de las señales PWM, ver anexo I. Esta funcionalidad queda recogida en el bloque *PWM* que añade el RTI a SIMULINK. Este bloque recibe seis entradas, una por cada PWM a generar. Para cada una de ellas, el valor que tome la señal de entrada indica el valor del *duty cycle* (tanto por ciento del ciclo en que la señal está alta, en rango [0,1]). Este bloque tiene varios parámetros configurables: individualmente para cada una de las señales, los valores inicial y final de duty cycle; y de forma común a todas las señales, la frecuencia de generación de la PWM. Al igual que ocurría con el conversor DA, será necesario conectar a tierra las entradas de las señales PWM que no vayamos a utilizar. Éste es el aspecto del bloque *PWM*:

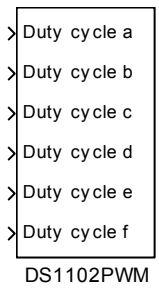

Fig. 16: bloque RTI para la generación de PWM's

Bloque *ENC_POS*: permite el acceso a los valores de ángulo generados por los interfaces de encoder incremental de la tarjeta a partir de los trenes de pulsos desfasados de un encoder incremental conectado a ellos. Para una descripción de la medición de ángulos mediante encoders incrementales, ver anexo II. Este bloque no tiene parámetros. La conversión entre la salida de este bloque y el ángulo que representa, en radianes, obedece a la siguiente lógica: el interfaz del encoder almacena el ángulo actual mediante un contador que se incrementa en una unidad por cada pulso que recibe en una dirección y se decrementa por cada pulso en la dirección contraria. Este contador tiene 24 bits. La salida que da el bloque tiene un rango [-1,1], que corresponde a la interpretación del valor del contador como un entero que varía en el rango [-2^{23},2^{23}-1], y su correspondiente escalado. Además hay que tener en cuenta que cada pulso real del encoder conectado se transforma en realidad en cuatro pulsos en el contador, no en uno. Por todo ello, la conversión a realizar es la siguiente para obtener el ángulo real en radianes:

$$ang\_radianes = 2^{23} \frac{2.\pi}{4 resolución\_encoder} salida\_bloque \qquad (1)$$

Esta conversión se puede llevar a cabo en SIMULINK mediante el uso de un bloque denominado *gain* (ganancia), que permite multiplicar una señal por un valor (parámetro). Este es el aspecto de los bloques *ENC_POS* y *gain* (éste, con valor = 1):

Fig. 17: bloques *enc_pos* y *gain*

Bloques *ENC_INDEX* y *ENC_RESET*: los encoders incrementales no son capaces de determinar el ángulo absoluto que están leyendo, sino sólo incrementos en ese ángulo. Es necesario por tanto fijar cuál es el ángulo cero para poder medir ángulos absolutos. Para ello el RTI aporta dos bloques que trabajan conjuntamente. El primero, *ENC_INDEX*, devuelve un uno cuando el encoder detecta que ha pasado por la posición *index* del encoder. Es preciso habilitar explícitamente la búsqueda de índice mediante la entrada *enable_search* de este bloque. Al contrario que los bloques que hemos visto hasta ahora, en los que cuando un elemento de E/S se duplicaba el bloque correspondiente tenía tantas entradas o salidas como veces estaba duplicado estaba el

elemento, en este bloque hay una sola salida: mediante un parámetro indicaremos a qué canal nos referimos, si al uno o al dos.

El segundo bloque que aporta RTI para determinar la posición del ángulo cero es el *ENC_RESET*, que al recibir un uno lógico en su entrada resetea el contador del interfaz del encoder correspondiente (de nuevo hay que especificar mediante un parámetro a qué canal de los dos existentes nos referimos).

La conjunción de ambos bloques posibilita el reseteo automático del contador de pulsos ante la detección del índice, conectándolos en la siguiente disposición:

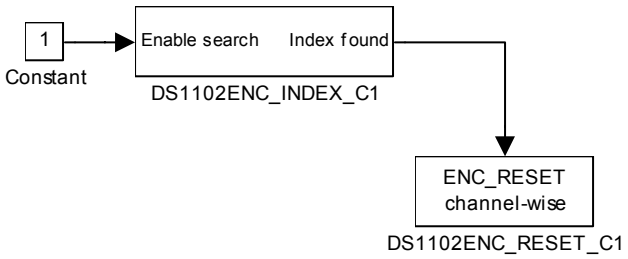

Fig. 18: configuración de bloques RTI para el reseteo del contador de un encoder ante la detección de la señal *index*

3.4. Software para interaccionar con / monitorizar el programa en ejecución.

El fabricante de la tarjeta incluye con ella varias posibilidades software de monitorización de e interacción con los programas que se ejecutan en ella. Por una parte hay dos programas que consiguen este objetivo a través de un interfaz gráfico: son *COCKPIT* [2]y *TRACE* [3]. Por otra, se añaden unas funciones MATLAB que permiten hacer lo mismo desde este entorno: *MLIB* [9] y *MTRACE* [6]. Sólo comentaremos la primera opción, que es la única que se ha utilizado en este proyecto.

COCKPIT permite la creación de un panel de control del programa en ejecución mediante la distribución en una pantalla en blanco de una serie de controles gráficos como barras deslizadoras, indicadores parecidos al velocímetro analógico tradicional de los automóviles, led's, etc. Estos controles se asocian a variables del programa, y pueden ser de entrada (actúan sobre las variables), de salida (sólo muestran el valor de la variable asociada), o de entrada/salida (a la par que actuamos sobre la variable asociada, muestran su valor actual). Una vez creado este panel, se activa, y a partir de ese momento nuestras acciones sobre los controles tendrán repercusión en las variables conectadas a ellos, al tiempo que los controles que visualizan valores serán actualizados continuamente. Este es el aspecto de algunos de los controles que se pueden incorporar a los paneles de control generados con COCKPIT:

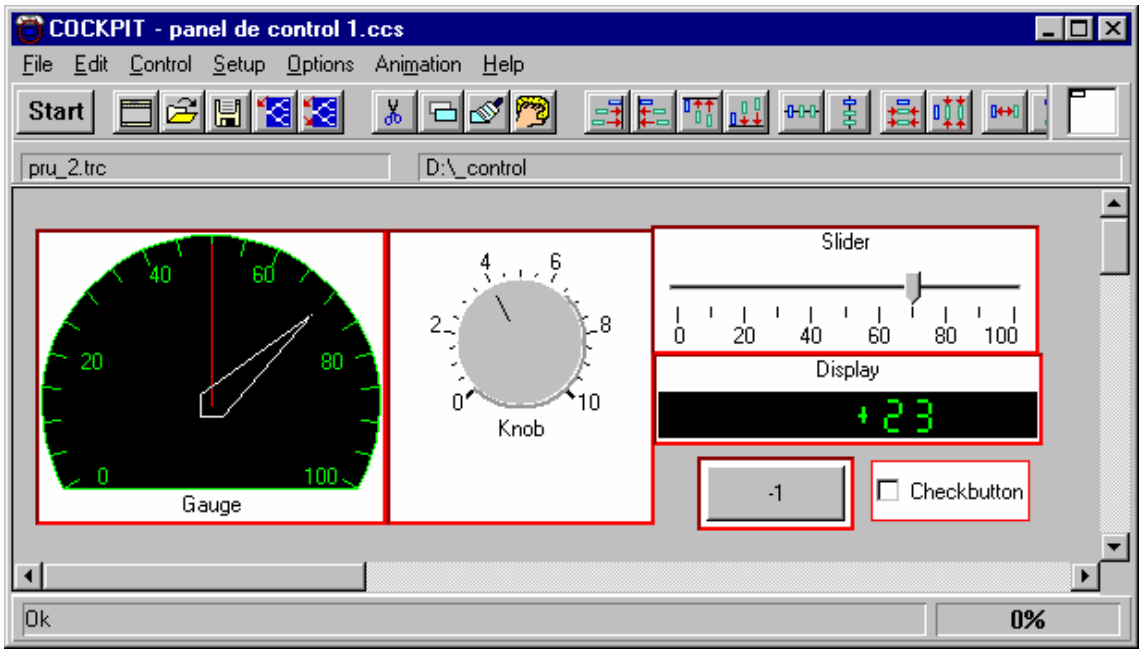

Fig. 19: ejemplo de controles de *COCKPIT*

TRACE . Respecto al seguimiento de valores de variables, *COCKPIT* nos permite saber "más o menos, por dónde están", pero no permite en modo alguno el muestreo en tiempo real de señales. Para desarrollar esta función fabricante de la tarjeta proporciona el programa *TRACE*. En este programa se eligen las variables a muestrear mediante un browser de variables que muestra la jerarquía de bloques SIMULINK. Además se han de especificar los parámetros de muestreo, como son la longitud del intervalo en que se va a tomar la muestra, cada cuántos periodos de muestreo del programa en ejecución se toma una muestra, etc. Tras ello, con el programa en ejecución, cliqueando el botón *start*, se activa el muestreo. Cuando éste finaliza, se pueden realizar todo tipo de operaciones con los datos adquiridos, desde su estudio directo en el propio TRACE, gracias a las facilidades que proporciona de cara al análisis de gráficas (zoom, cursores de todo tipo, superposición de varias señales en la misma gráfica, etc.), como su exportación a un fichero de datos, básicamente a formato MATLAB. En la figura 26 se muestra un ejemplo del seguimiento de variables de un programa en ejecución mediante TRACE.

3.5. Proceso genérico de diseño de un controlador.

El diseño de sistemas de control (véase por ejemplo [17], capítulo 7) responde en la práctica a un proceso iterativo. El objetivo final es la obtención de un sistema controlado que responde a un conjunto de especificaciones previamente fijadas. La Fig. 20 representa un esquema genérico de los diferentes pasos de diseño para la obtención de un sistema de control lineal.

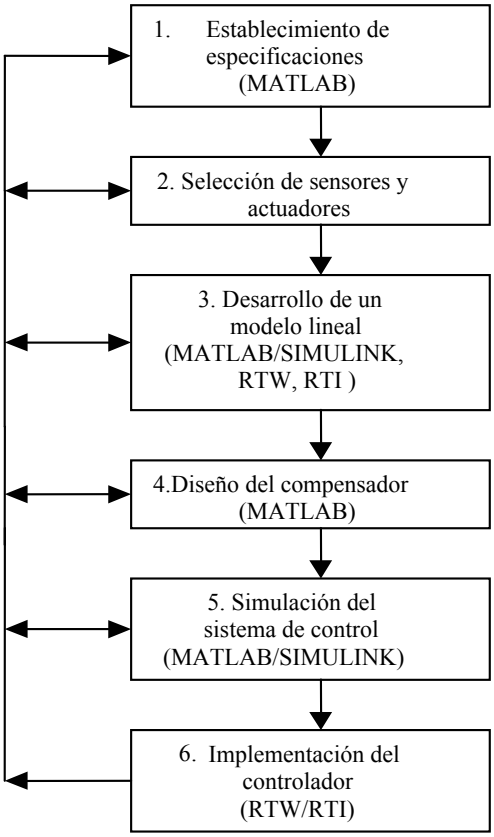

Fig. 20: Diseño de sistemas de control

El primer paso de diseño consiste en la definición de especificaciones, usualmente en el dominio del tiempo y/o en el dominio de la frecuencia, incluyendo también una primera descripción del sistema que se pretende controlar. Una buena comprensión del proceso puede prevenir el fijar especificaciones excesivamente fuertes para el proceso en cuestión, que pudieran requerir un esfuerzo de control excesivo. El segundo paso corresponde a la elección de los sensores y actuadores adecuados para el problema. Esta elección responderá a diferentes tipos de factores entre los que sin duda aparecerán aspectos tecnológicos y económicos. El siguiente paso es la obtención de un

modelo lineal a partir del proceso. Usualmente se parte de un modelo obtenido a partir de primeras leyes (por ejemplo, ecuaciones de Lagrange-Euler para un sistema mecánico), y luego se hace una validación experimental utilizando técnicas de identificación ([15,19]). Una vez obtenido un modelo lineal fiable, se diseña el controlador, por ejemplo utilizando técnicas de respuesta en frecuencia o basadas en el lugar de las raíces ([17]). Después se simula el sistema de control, y si cumple las especificaciones, y el diseño es suficientemente robusto, se implementa en el procesador que se encargará de realizar el control digital del sistema.

El proceso es iterativo porque en los diferentes pasos es inevitable realizar aproximaciones. Si la realización de algún paso de diseño es imposible por obtenerse propiedades indeseables (por ejemplo, el sistema se hace inestable o el esfuerzo de control es inadmisible), entonces hay que dar un paso atrás permitiendo una relajación de las condiciones que producen ese efecto indeseado o el rediseño de algún sistema. Es conveniente enfatizar el hecho de que prácticamente todas las etapas de diseño de un sistema de control quedan cubiertas por la utilización de alguna o varias de las herramientas analizadas en este proyecto, esto es, MATLAB/SIMULINK, RTW y RTI.

A continuación detallamos, por su importancia, dos pasos fundamentales del proceso de diseño: la obtención del modelo de la planta, y el diseño del compensador.

Obtención del modelo de la planta. Identificación:

Después de tener un primer modelo teórico de la planta, o incluso en aquellos casos que por su complejidad sea difícil de obtener un modelo teórico, se debe realizar una identificación del sistema (por ejemplo su función de transferencia) a partir de datos experimentales. Se pueden seguir los siguientes pasos:

1. Generamos la señal de ataque mediante un algoritmo programado en MATLAB. Llamémosla *S*.
2. Generamos un modelo SIMULINK que ataque a la planta con la señal obtenida en el paso anterior, y a la par reciba la salida de la planta como entrada a alguna de sus señales internas, que etiquetaremos para poder identificarla posteriormente.

3. Con RTW y RTI compilamos y ejecutamos el programa sobre la tarjeta (que está conectada a la planta a identificar).
4. Capturamos la salida de la planta con TRACE y la salvamos a un fichero con formato de datos MATLAB. La identificamos gracias a la etiqueta citada en 2.
5. Usamos MATLAB (*Toolbox* de *System Identification*) para analizar la respuesta de la planta y deducir un modelo que la aproxime.

Diseño del controlador a partir del modelo obtenido:

Una vez que tenemos un modelo de la planta podemos diseñar un controlador mediante el siguiente esquema:

1. Diseñamos el controlador en MATLAB (*Toolbox* de Control), utilizando las herramientas que proporciona para esta tarea, como son los diagramas de bode, las funciones relacionadas con el lugar de las raíces, etc.
2. Realizamos una simulación del controlador del paso 1 en SIMULINK, utilizando el modelo que tenemos de la planta. Si los resultados aconsejan rediseñarlo, volveríamos al paso 1. Si no, seguimos...
3. Generamos a partir del modelo SIMULINK del paso 2 un programa de control para la tarjeta. Para ello quitamos del modelo SIMULINK el bloque correspondiente a la planta y lo sustituimos por entradas y salidas de la tarjeta. Tras ello generamos el ejecutable con RTW. Una vez obtenido el ejecutable, podemos probar el controlador diseñado directamente sobre la planta, y observar el comportamiento de ésta mediante las herramientas de monitorización. Si los resultados aconsejan rediseñar el controlador, volveríamos al paso 1. Este paso, al mismo tiempo que permite comprobar si el controlador es efectivo, permite sintonizar sus parámetros a través de COCKPIT.

4. APLICACIÓN: DISEÑO DE UN CONTROLADOR PARA UN PÉNDULO INVERTIDO

El péndulo invertido es un sistema muy utilizado en ambientes académicos debido a que se trata de un sistema muy inestable y por tanto difícil de controlar.

4.1. Planteamiento del problema de control.

Tenemos una plataforma metálica que se desplaza en línea recta sobre un raíl. La denominaremos *carro*. Podemos aplicar al carro una fuerza F en la dirección de desplazamiento y en cualquiera de los dos sentidos gracias a la acción de un motor:

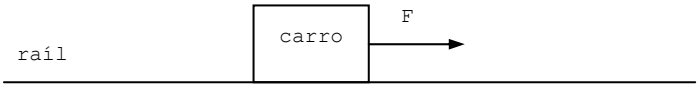

Fig. 21: carro sobre el raíl con fuerza F aplicada

Sobre el carro se sitúa una varilla que bascula sobre un eje perpendicular a la vertical y a la dirección de movimiento. Denominaremos a este eje *pivote*. Esta varilla tiene una pesa en el extremo opuesto al eje de giro:

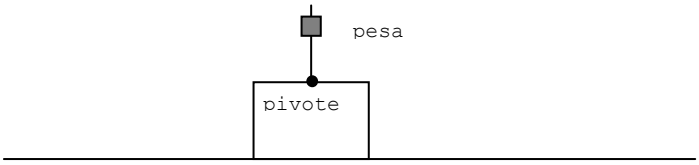

Fig. 22: péndulo invertido sobre el carro

Denominamos *péndulo invertido* al conjunto varilla-pesa basculando sobre el pivote. El péndulo supone un sistema en un equilibrio inestable en su posición vertical: debido a que en esta posición se encuentra en un máximo de la función "energía potencial frente al ángulo con la vertical", la más mínima variación respecto a cero de este ángulo hará que el péndulo tienda a caer. Denominaremos θ al ángulo que forma el péndulo con la vertical:

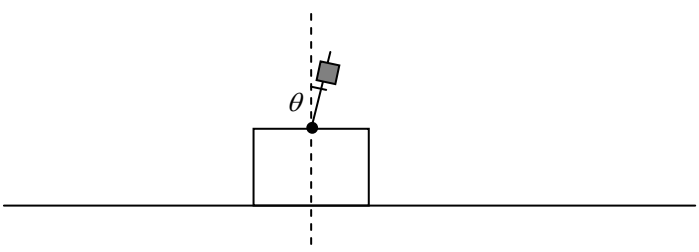

Fig. 23: ángulo θ

El objetivo del problema de control es mantener el péndulo en la vertical, es decir, conseguir que θ sea cero. Para ello será necesario diseñar un controlador que tome como variable de entrada el ángulo θ y que genere una acción de control que determine qué fuerza *F* ha de ser aplicada al carro a través del motor.

Para medir el ángulo θ contamos con un encoder incremental OMROM® E6B2-CWZ3E [20], que tiene una resolución de 500 pulsos por revolución. Ver anexo II para una explicación del funcionamiento de este tipo de dispositivos. Este encoder, conectado a uno de los interfaces de encoder incremental de la tarjeta de control, nos permite la lectura directa del ángulo absoluto θ.

El motor utilizado como actuador es un motor eléctrico de corriente continua, que recibe una alimentación de 12V DC y cuya velocidad será controlada mediante una PWM. Ver anexo I para una descripción de las señales PWM y cómo se han generado mediante un bloque SIMULINK las necesarias para este motor en base a la electrónica de potencia utilizada. Dado que el motor ha de girar en ambos sentidos, ha sido necesario utilizar un amplificador que permita esa posibilidad. Hemos elegido una distribución de transistores en H, que permite el giro en ambos sentidos con una única fuente de alimentación. La filosofía de esta disposición es aplicar la PWM en sólo en la base de un par de transistores para conseguir el giro en un sentido y en la base del par simétrico para conseguir el giro en el otro sentido. La disposición en H tiene este aspecto (*giro* + indica dónde actúa la señal para conseguir giro en un sentido, y *giro* – en el contrario; *M* representa al motor) [18]:

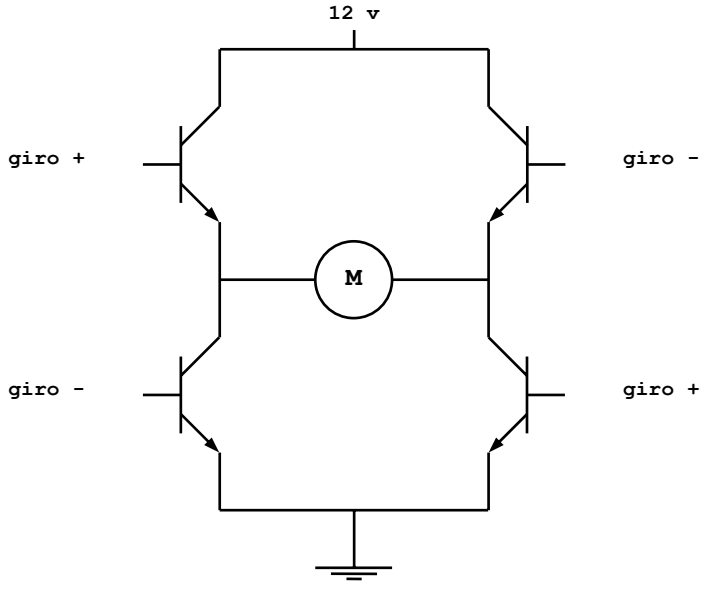

Fig. 24: esquema de la disposición de transistores en H

Para una descripción detallada de la electrónica utilizada y de los problemas encontrados / soluciones aportadas en la interacción con la tarjeta, ver el anexo III.

Para la obtención del modelo de la planta será necesario realizar una identificación empírica de parte de ella, como se verá en el siguiente subapartado. Para ello se ha utilizado otro encoder que gira solidario con el eje del motor. Su resolución es de aproximadamente 60 pulsos por revolución del motor en el eje de la rueda dentada que transforma el giro en traslación sobre el raíl. Un radián de giro sobre este eje supone un desplazamiento de un centímetro en la dirección del raíl. Debido en parte a una avería en sus componentes, ha sido necesario rediseñar la electrónica de este encoder. Para una explicación del nuevo circuito diseñado ver anexo IV.

4.2. Identificación de la planta.

Antes de diseñar un controlador para la planta es necesario tener un buen modelo de ella. Podríamos descomponer la planta de la siguiente forma:

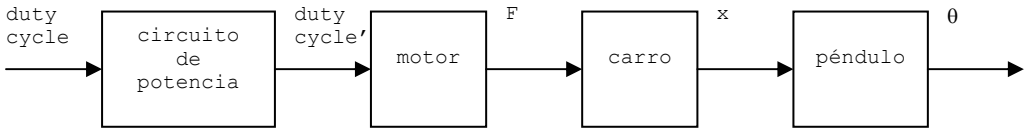

Fig. 25: sistemas que componen la planta

Dado que la masa del péndulo es despreciable frente a la del carro, parece razonable despreciar la oposición que éste ofrece por su inercia al desplazamiento del primero. Por ello podemos desacoplar el sistema en dos bloques, para pequeñas variaciones de θ:

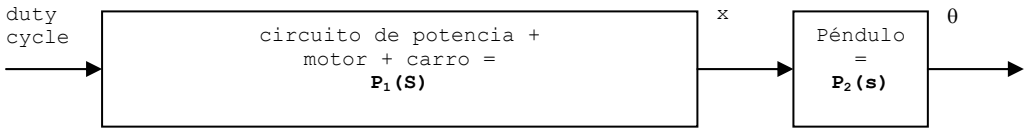

Fig. 26: agrupación de los sistemas de la planta

Vamos a identificar empíricamente el sistema formado por los tres primeros bloques, es decir, la función de transferencia que expresa la relación entre el duty cycle aplicado al circuito de potencia y la posición resultante del carro. A esta función de transferencia la llamaremos $P_1(s)$. Para el último bloque haremos un modelo teórico estudiando la dinámica del péndulo invertido. A la función de transferencia de este segundo bloque la llamaremos $P_2(s)$.

La función de transferencia de la planta completa será $P(s) = P_2(s).P_1(s)$. Veamos la obtención de cada una de estas dos funciones.

Obtención empírica de $P_1(s)$

Para identificar $P_1(s)$ es necesario excitar al sistema con una señal rica en frecuencias y observar su respuesta. A partir de ella podremos deducir un diagrama de Bode, de cuyo perfil podremos deducir la función de transferencia del sistema. Una señal rica en frecuencias ampliamente utilizada la PRBS (Pseudo-Random Binary Sequence). Se trata de una señal que varía bruscamente entre un máximo y un mínimo

de forma pseudo-aleatoria. Al ser aleatoria la distancia entre cambios de nivel, se consiguen barrer un rango de frecuencias muy amplio. Para una explicación sobre este tipo de señales y cómo se han generado con MATLAB, ver anexo V. La PRBS tiene una serie de parámetros a configurar. Para elegirlos es necesario conocer previamente el tiempo de subida del sistema a identificar. Para calcularlo hemos utilizado el siguiente modelo SIMULINK, compilado con RTW y ejecutado en la tarjeta:

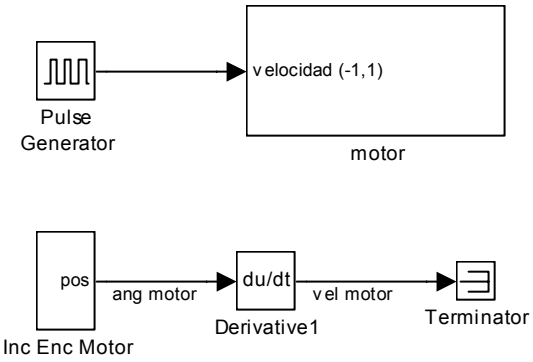

Fig. 27: modelo SIMULINK para el estudio del tiempo de subida del primer bloque de la planta

El bloque *motor* recibe la señal *duty cycle*, que determina la velocidad del motor. Esta señal se mueve en el rango [0,1]. Para una descripción de la comunicación de este bloque con la electrónica de potencia, ver anexo I. Introducir como *duty cycle* un escalón (en realidad una secuencia de escalones, para obtener una repetición automática del experimento de observación de la respuesta escalón. El bloque *Inc Enc Motor* devuelve a su salida la posición actual del motor en radianes (señal *ang motor*). Si diferenciamos esta señal (bloque *derivative*), obtenemos la velocidad del motor (*vel motor*). La monitorización mediante TRACE de esta última señal permite estudiar la respuesta escalón del sistema. De este modo hemos obtenido el dato de que el tiempo de subida del sistema es de entre 0'2 y 0'3 segundos, lo que nos permite ya la generación de una PRBS concreta según el método descrito en el anexo V. Esta PRBS generada es almacenada en el fichero de datos *PRBS.mat*, y tiene el siguiente aspecto (se muestra solamente un fragmento de la señal):

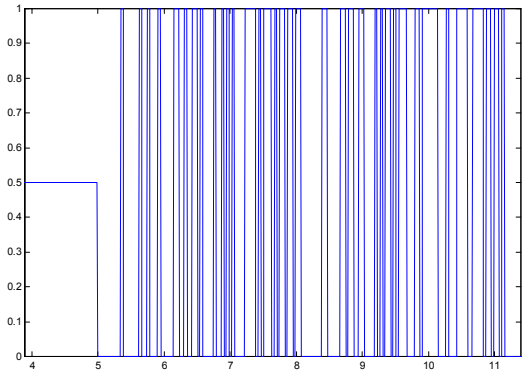

Fig. 28: aspecto de la señal PRBS utilizada para la identificación

El modelo SIMULINK para aplicar esta PRBS al sistema es muy parecido al utilizado para estudiar la respuesta escalón, pero ahora la señal en lugar del tren de pulsos es la PRBS leída desde el fichero *PRBS.mat* (bloque *From File* de SIMULINK), y centrada / escalada de forma que varíe en el rango [-1,1], para conseguir que el motor se mueva en los dos sentidos:

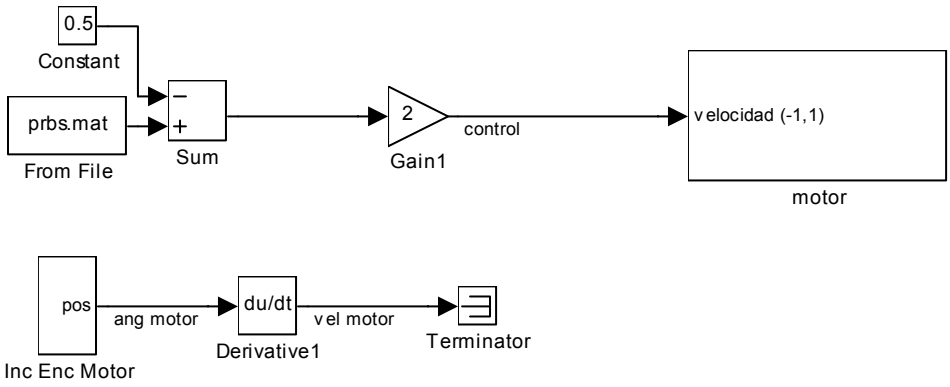

Fig. 29: modelo SIMULINK utilizado para estudiar la respuesta de la planta ante la PRBS

Este modelo es compilado y ejecutado sobre la tarjeta. Mediante TRACE podemos monitorizar la respuesta del motor a la PRBS, y guardarla en un fichero. Repetimos el experimento cinco veces, generando por tanto cinco ficheros de respuestas a la misma PRBS. Con estos ficheros con las respuestas obtenidas, pasamos a la identificación del sistema. Para ello nos valdremos del toolbox de identificación de sistemas de MATLAB, *ident* [15].

Este toolbox nos va a permitir de una forma sencilla, desde un interfaz gráfico, para cada una de esas cinco respuestas obtenidas, realizar un preprocesado (eliminación del nivel de continua) y obtener los diagramas de Bode. Una vez obtenidos los cinco diagramas de Bode, los exportamos al workspace de MATLAB, donde realizaremos un promedio de ellos. El diagrama de Bode promediado resultante es el que se muestra el la página siguiente:

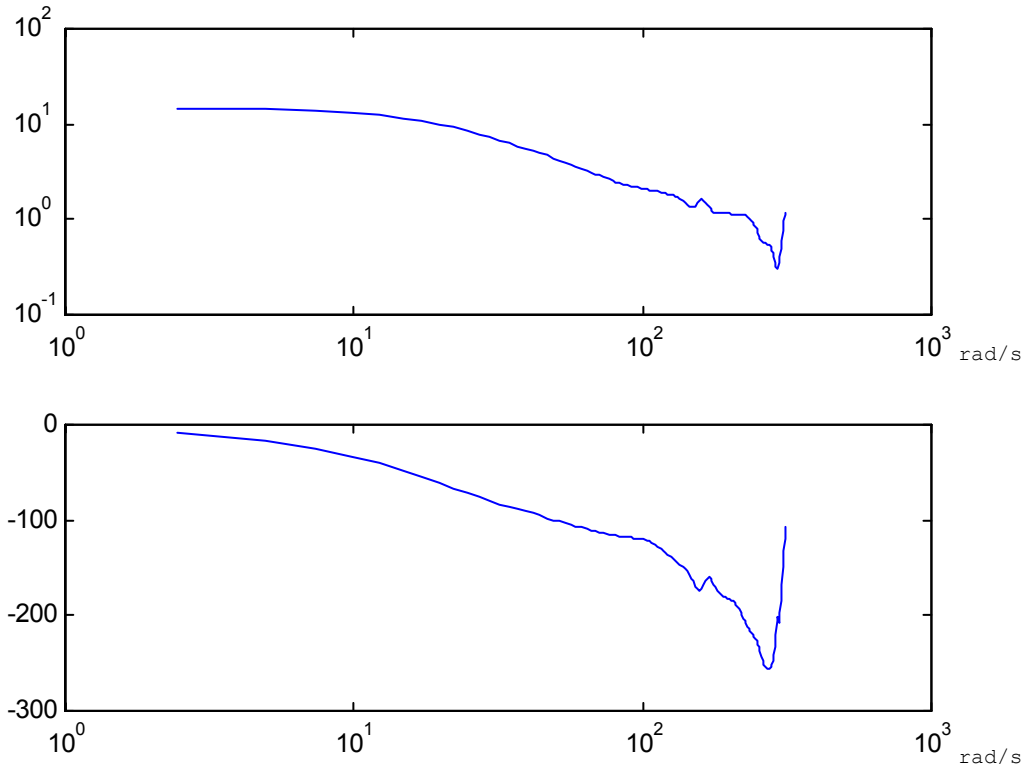

Fig. 30: diagrama de bode obtenido de la planta experimentalmente

La respuesta del motor ha sido obtenida con un periodo de muestreo de 0.01 segundos (frecuencia de muestreo de 100 Hz), $p = 4$ y $N = 10$ (ver anexo V para una expliación de los parámetros de la PRBS). Según el teorema de Nyquist, de esta forma estaríamos registrando las frecuencias de la respuesta de hasta 50 Hz. Éste es el límite teórico, que es realmente aun menor en la práctica. Así que en el diagrama de Bode anterior deberemos despreciar la información sobre las frecuencias más altas, para las que como vemos la respuesta del motor tiene un comportamiento algo extraño: todo sistema físico tiende a disminuir su ganancia al aumentar la frecuencia, y no al revés, como en este caso.

De los diagramas en primer lugar podemos deducir que el sistema no tiene polos en el origen, ya que la magnitud empieza sin pendiente y el ángulo en cero. A continuación observamos la aparición de un polo real que provoca una caída de 20 dB por década "algo más allá de $\omega = 10$". Esto queda corroborado por el diagrama de fase. Haciendo zoom sobre la gráfica del ángulo, y con una escala lineal para las frecuencias, podemos determinar que la frecuencia para la que el ángulo es de 45º es aproximadamente de 13'65 rad/s. Respecto a la ganancia del sistema, haciendo zoom en

el diagrama de amplitud y con escala de amplitudes lineal podemos ver que ésta es de aproximadamente igual a 15 (referencia: ω = 1). El sistema resultante es por tanto:

$$\frac{\text{velocidad(s)}}{\text{duty cycle(s)}} = \frac{15}{\frac{s}{15}+1} = \frac{15}{0.07s+1} \qquad (2)$$

Este sistema tiene el siguiente diagrama de bode (que como se puede observar coincide en su estructura con el experimental):

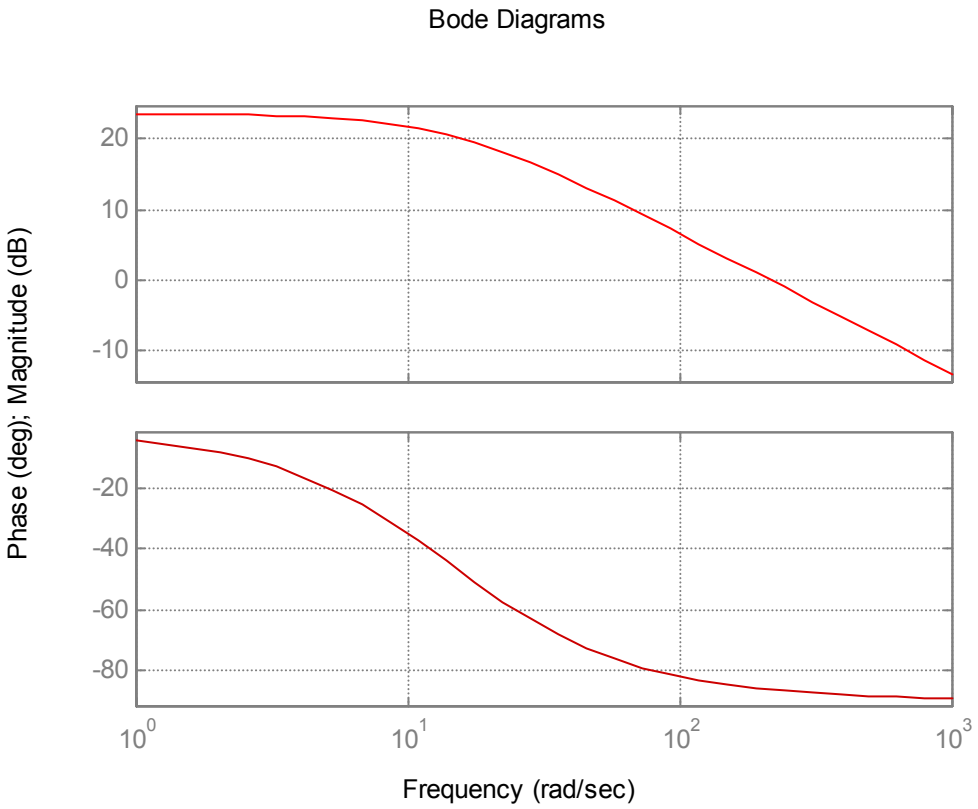

Fig. 31: diagrama de bode del modelo extraído de la planta

Como en realidad $P_1(s)$ expresaba la relación entre *duty cycle* y posición, habrá que multiplicar esta última expresión por 1/s. Y además será necesario multiplicar la ganancia por 100, ya que hemos obtenido este sistema midiendo como respuesta del motor su ángulo de giro en radianes, cuando lo que queremos es el desplazamiento producido en metros (y según hemos expresado anteriormente, en este experimento 1 radián de giro del motor corresponde a 1cm de desplazamiento sobre el raíl). El sistema resultante es:

$$P1(s) = \frac{\text{X(s)}}{\text{duty cycle(s)}} = \frac{0.15}{\text{s.}(0.07s+1)} \qquad (3)$$

Modelo teórico de $P_2(s)$

Consideremos el siguiente diagrama, en el que vemos que consideramos ángulos positivos los de la vertical hacia la derecha, y desplazamientos sobre el raíl positivos, hacia la derecha:

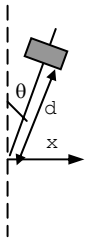

Fig. 32: sentido positivo del desplazamiento sobre el raíl y el ángulo con la vertical

La dinámica del péndulo [17,21,22] quedará establecida por la ecuación fundamental de la dinámica de rotación, que iguala la suma de momentos respecto a un eje (en nuestro caso el pivote) momento de inercia del cuerpo que gira (el péndulo) por su aceleración angular respecto al eje de giro:

$$\sum M = I\,\ddot{\theta} \tag{4}$$

Teniendo en cuenta que la masa de la pesa es muy grande frente a la de la varilla, podemos considerar el péndulo como una masa puntual situada en el centro de la pesa. Haciendo esta consideración, el momento de inercia del péndulo será la masa puntual por la distancia a la que está del pivote al cuadrado:

$$I = md^2 \tag{5}$$

Las fuerzas que actúan sobre el péndulo provocando momento son su propio peso y la fuerza que le imprime el carro. Por el desacoplo que hemos realizado entre las dinámicas del carro y el péndulo, la fuerza del carro sobre el péndulo se calcula directamente como la masa del péndulo por la aceleración del carro:

$$F = m\ddot{x} \tag{6}$$

El peso del péndulo es:

$$P = mg \tag{7}$$

Para calcular los momentos que provocan, veamos el siguiente gráfico:

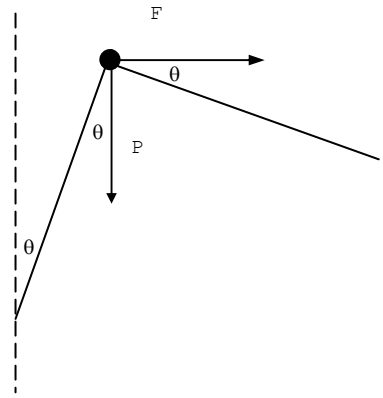

Fig. 33: angulos formados por la fuerza *F* y el peso con la tangente al radio de giro

De él podemos deducir los momentos de esas dos fuerzas:

$$M_P = mgd\,\mathrm{sen}(\theta) \tag{8}$$

$$M_F = m\ddot{x}d\cos(\theta) \tag{9}$$

Si ahora volvemos sobre la ecuación 1, tenemos:

$$mgd\,\mathrm{sen}(\theta) + m\ddot{x}d\cos(\theta) = md^2\ddot{\theta} \tag{10}$$

Para ángulos θ muy pequeños, se puede hacer la aproximación: $\mathrm{sen}(\theta) \approx \theta$ y $\cos(\theta) \approx 0$. En ese caso tenemos:

$$mgd\theta + m\ddot{x}d = md^2\ddot{\theta} \tag{11}$$

Aplicando la transformada de Laplace:

$$mgd\theta(s) + ms^2X(s)d = md^2s^2\theta(s) \tag{12}$$

Operando sobre la ecuación 9, obtenemos:

$$g\theta(s) + s^2 X(s) = md\ \ s^2\theta(s)$$

$$\frac{\theta(s)}{X(s)} = \frac{s^2}{ds^2 - g} \tag{13}$$

El valor de *d* es en principio variable, ya que la pesa no va soldada a la varilla, sino que esta distancia es ajustable. Pero a lo largo de todo el proyecto hemos trabajado con *d* = 15 cm = 0.15 m. Ésta última ecuación es la función de transferencia $P_2(s)$ que estábamos buscando.

Función de transferencia de la planta completa

Según todo lo anterior, la función de transferencia de la planta será $P(s) = P_2(s)P_1(s)$, o lo que es lo mismo:

$$\mathbf{P(s)} = \frac{\theta(s)}{\text{duty cycle(s)}} = \frac{\theta(s)}{X(s)} \cdot \frac{\text{X(s)}}{\text{duty cycle(s)}} = P_2(s)P_1(s) = \\ = \frac{s^2}{0.15s^2 - g} \cdot \frac{0.15}{\text{s}(0.07\text{s} + 1)} = \frac{\mathbf{0.15s}}{\mathbf{(0.07s + 1)(0.15s^2 - g)}} \tag{14}$$

4.3. Diseño de un controlador.

Diseñaremos un controlador en lazo cerrado mediante el método del lugar de las raíces [17] y con la ayuda de MATLAB. A continuación simularemos el comportamiento de la planta controlada por el controlador diseñado. Por último compilaremos el controlador y lo ejecutaremos sobre la tarjeta. Como veremos, hay ciertos cuestiones que ocurren en los experimentos reales que a menudo no son tenidas en cuenta y que en cambio pueden ser cruciales para el buen funcionamiento de un controlador. Este tipo de cuestiones no se suelen detectar en las etapas de simulación, y no es hasta que se prueba el controlador frente a la planta real cuando nos damos cuenta de ellas.

Diseño del controlador mediante el lugar de las raíces

Para un controlador proporcional, obtenemos el siguiente lugar de las raíces (función *rlocus* del toolbox de control de MATLAB):

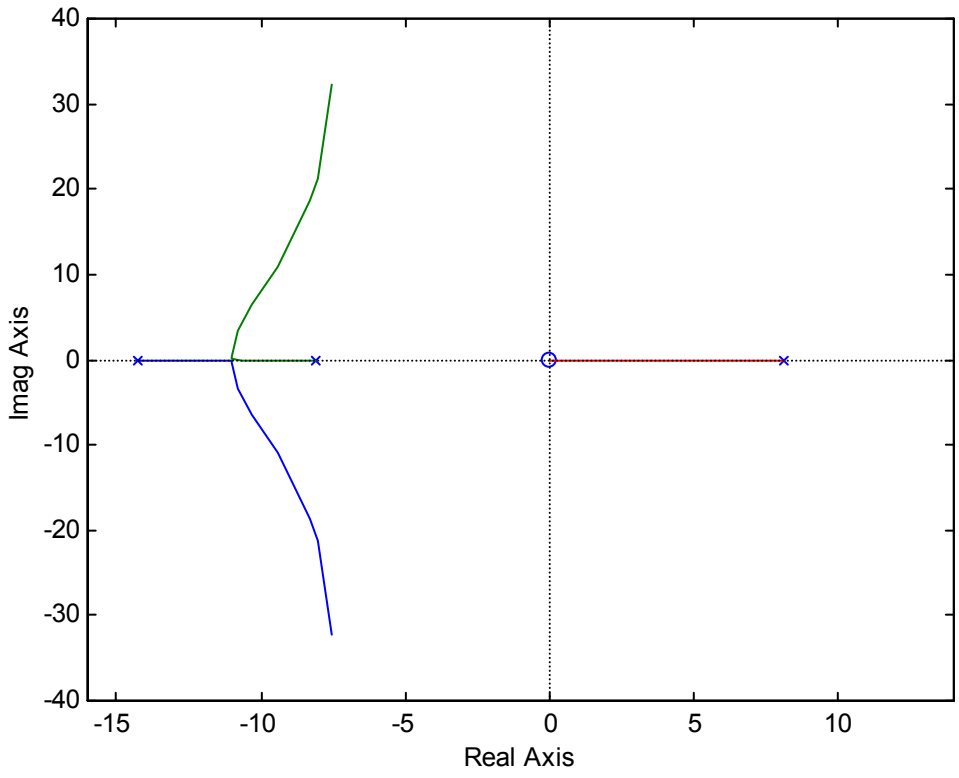

Fig. 34: $\mathrm{rlocus}\left(\frac{0.15s}{(0.07s+1)(0.15s^2-g)}\right)$

En primer lugar, añadiremos al controlador un en s = -8 para simplificar la dinámica del sistema.

En segundo lugar, tenemos que resolver un problema: tenemos un cero en el origen y un polo en s = 8.08. Esto hace que exista una rama del lugar de las raíces desde ese polo a ese cero. Eso significa que uno de los polos del sistema en lazo cerrado, el que corre por esa rama, es inestable para cualquier valor de ganancia del controlador. Para evitar esta situación se hace necesario introducir un polo entre ese cero y ese polo, de manera que consigamos que esa rama llegue al semiplano izquierdo antes de llegar a su cero de destino. Situaremos ese polo en s = -2. El controlador será de momento: $C(s) = \frac{s+8}{s-2}$. El lugar de las raíces será ahora:

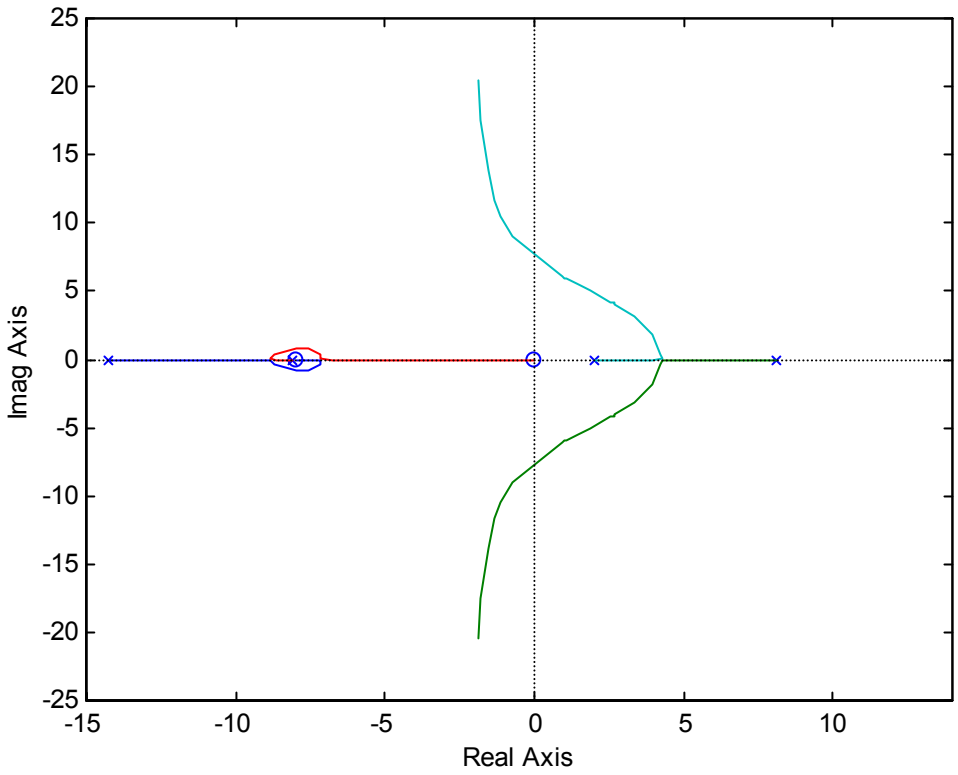

Fig. 35: $\text{rlocus}\left(\frac{s+8}{s-2}\frac{0.15s}{(0.07s+1)(0.15s^2-g)}\right)$

Para tratar de atraer algo más estas dos ramas, que parten de los polos del semiplano derecho, hacia el semiplano izquierdo, añadiremos un cero en s = -10. El nuevo controlador será $C(s) = \frac{(s+8)(s+10)}{s-2}$, y el lugar de las raíces:

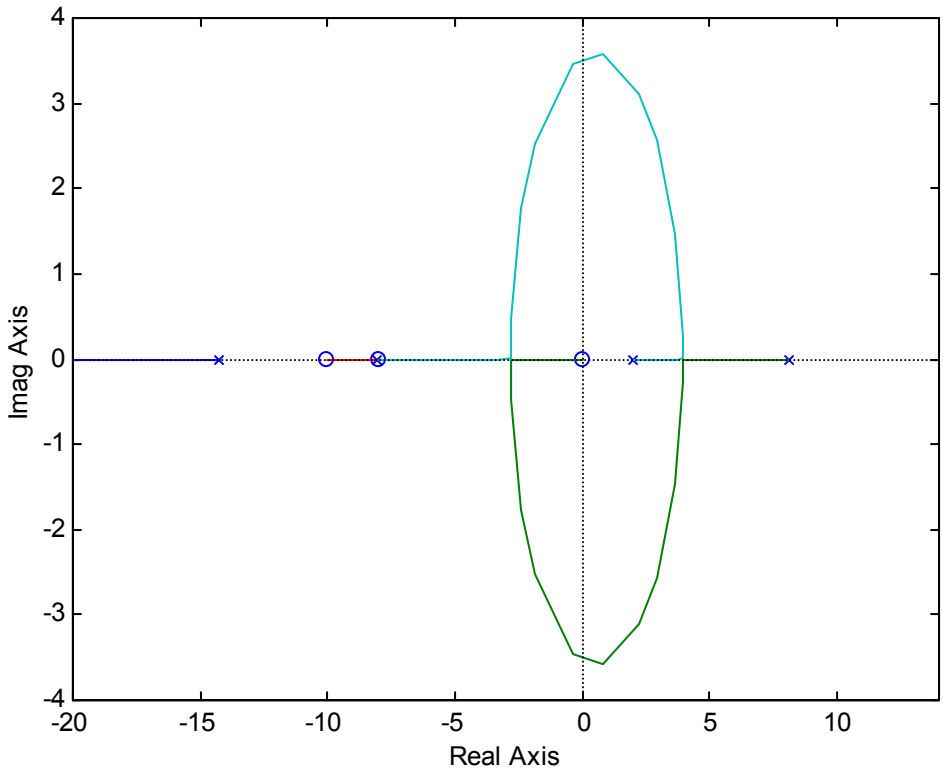

Fig. 36: $\text{rlocus}\left(\frac{(s+8)(s+10)}{s-2}\frac{0.15s}{(0.07s+1)(0.15s^2-g)}\right)$

Por último, para conseguir que el sistema sea propio, añadiremos un polo estable muy lejano (para que no influya en la dinámica del sistema). Digamos en s = -100. El controlador definitivo es $C(s) = \frac{(s+8)(s+10)}{(s-2)(s+100)}$ (sin especificar aún su ganancia).

La ganancia la determinamos utilizando la función *rlocfind*, que nos informa de la *k* elegida al elegir sobre el lugar de las raíces una situación concreta de los polos en lazo cerrado. Hemos obtenido un valor aproximado de *k=150* para posiciones relativamente aceptables de los polos de lazo cerrado.

Simulación en SIMULINK

El siguiente paso en el diseño del controlador es realizar una simulación en SIMULINK de la actuación del controlador sobre el modelo obtenido de la planta, para comprobar que el sistema resultante es estable. Para poner a prueba la estabilidad del sistema en condiciones más o menos reales introduciremos una perturbación aleatoria de frecuencia 1 Hz y amplitud 0.01 (sobre la señal de control). Se hará lo mismo sobre la salida de la planta, de cara a la realimentación, para simular la presencia de ruido en la lectura del sensor. El modelo SIMULINK en que se realiza esta simulación es éste:

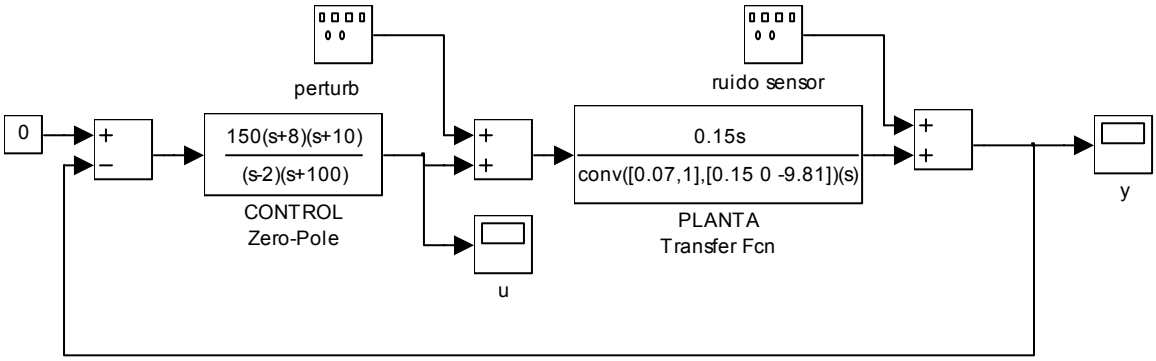

Fig. 37: modelo SIMULINK para simulación de la actuación del controlador sobre el modelo de la planta

En él aparecen tres tipos de bloques SIMULINK nuevos: el *Zero-Pole*, que nos permite especificar una función de transferencia en base a sus ceros, polos y ganancia; el *Transfer Fcn*, que permite hacer lo propio especificando los polinomios numerador y denominador de la función de transferencia asociada; y por último el *signal generator* (generador de señales), utilizado para generar la perturbación y el ruido del sensor.

Éste es el resultado de la simulación del modelo anterior en un intervalo de 2 segundos, observado en los bloques *scope* correspondientes a las variables *u* e *y*:

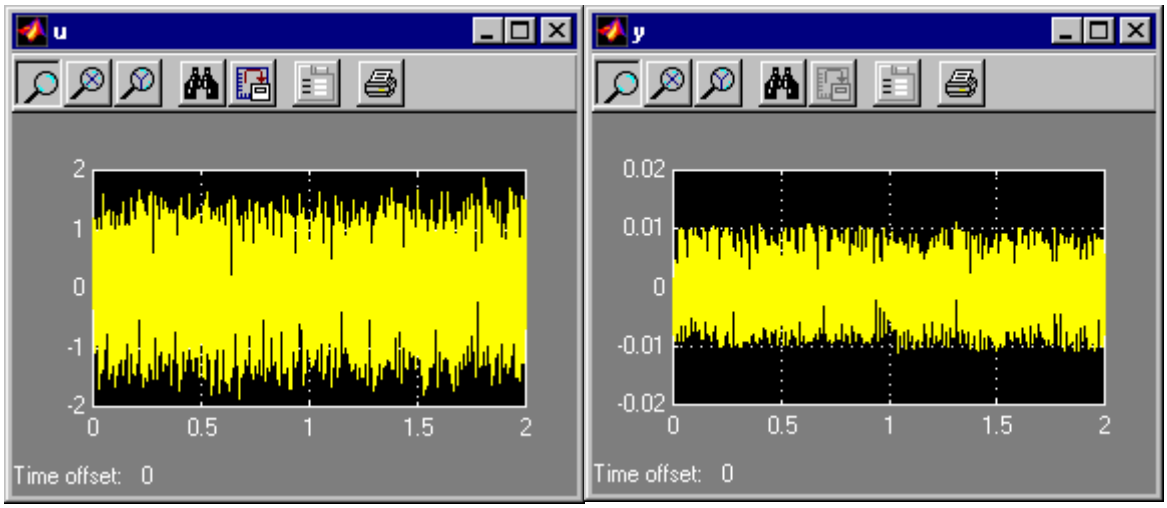

Fig. 38: resultado de la simulación con $k = 150$

Como vemos el controlador es capaz de neutralizar la perturbación y el ruido del sensor. Pero para ello necesita generar una acción de control en el rango [-2,2], que excede las posibilidades reales, [-1,1]. Disminuir ese rango de la acción de control pasa por elegir una ganancia menor para el controlador. Hemos comprobado que para una ganancia de 100 el controlador sigue siendo capaz de absorber las perturbaciones y el ruido del sensor con una acción de control que se mantiene dentro del rango rango [-1,1]:

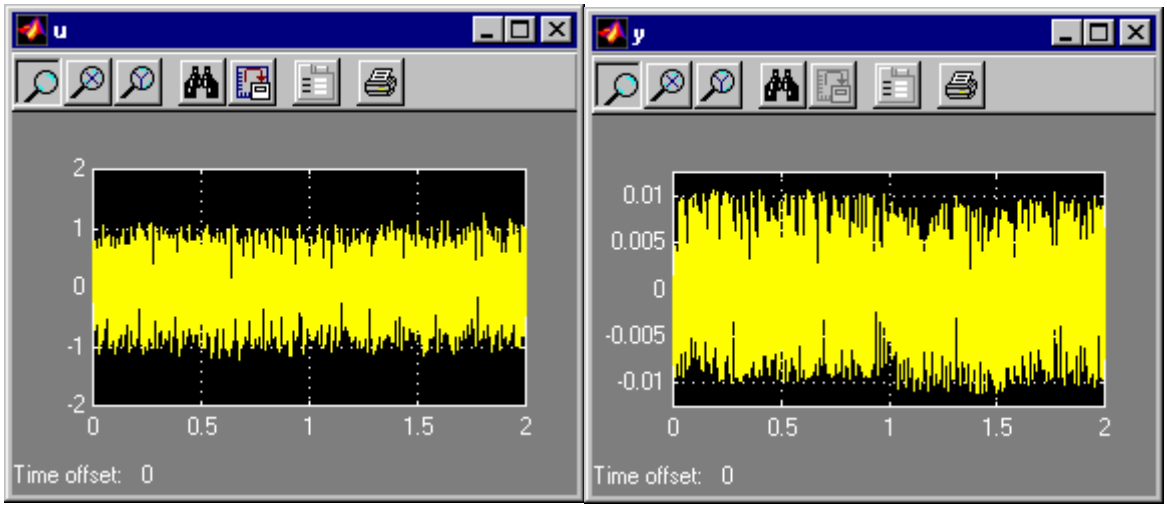

Fig. 39: resultado de la simulación con $k = 100$

Ejecución del controlador en la tarjeta

Para generar un programa que controle la planta real hay que sustituir en el modelo SIMULINK anterior la planta por las entradas y salidas de la tarjeta. Además el controlador será discretizado mediante la función *c2d* de MATLAB. En una primera aproximación, tendríamos el siguiente modelo:

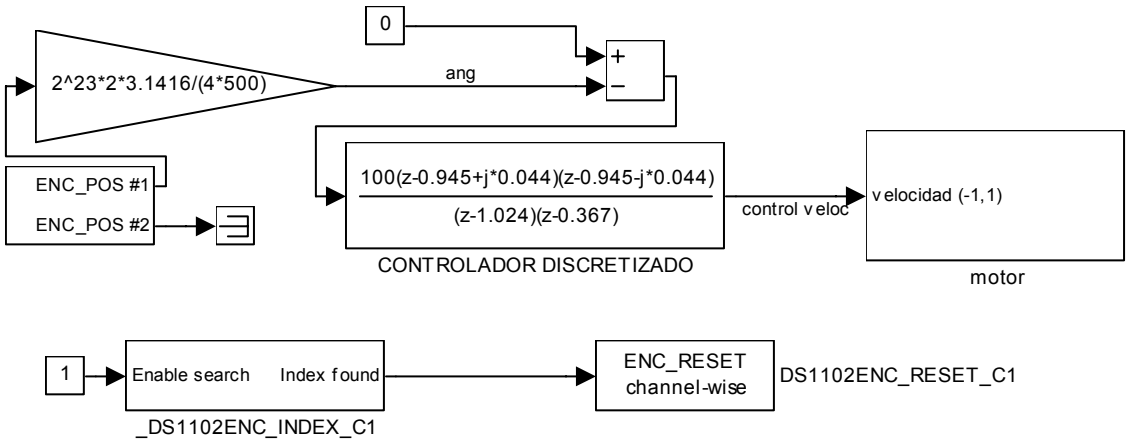

Fig. 40: modelo SIMULINK de la primera aproximación a la generación real del controlador

En él vemos que a la entrada del controlador discretizado llega la lectura del interfaz de encoder 1, y que la acción de control se aplica al bloque *motor*. Como ya sabemos, los bloques de la parte inferior se encargan de resetear el contador del encoder cuando se detecta la señal de índice. El ángulo leído por el encoder no es válido hasta que se detecte el primer paso por index. Se ha de evitar que el controlador comience a realizar cálculos con ángulos no válidos antes de ese primer paso por la posición de índice. Para ello haremos lo siguiente: utilizaremos un bloque de SIMULINK que implementa un biestable *RS* para almacenar la información de que ya se ha detectado el primer paso por el índice. La salida de este biestable condicionará el que el ángulo leído sea el devuelto por el encoder o bien cero (para instantes anteriores al primer paso por el índice). De esta forma conseguimos que el controlador perciba un ángulo cero hasta que la lectura del encoder no sea válida. Este es el modelo SIMULINK resultante:

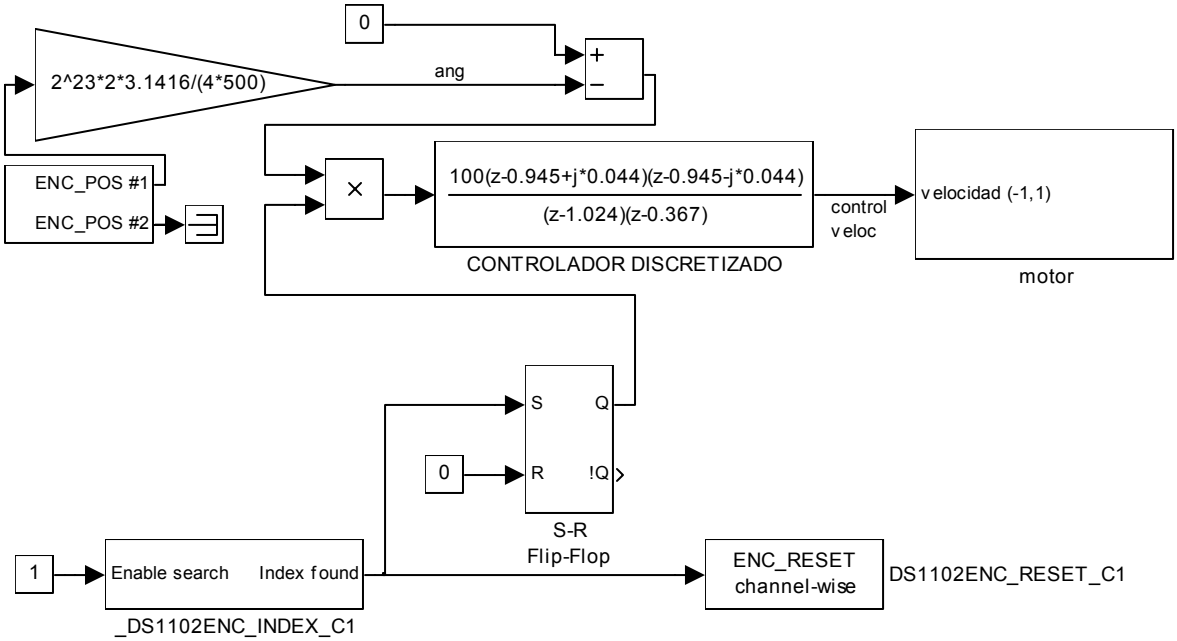

Fig. 41: modelo SIMULINK para la generación real del controlador

Una vez que hemos construido este modelo, lo compilamos con RTW y experimentamos el comportamiento del controlador sobre la planta real.

El resultado de este experimento ha sido que el controlador no consigue mantener el péndulo invertido en la vertical. Tras repetir la experiencia para distintas ganancias de controlador sin obtener mejores resultados, y tras revisar minuciosamente el modelo construido de la planta y el controlador diseñado, se llega a la conclusión de

que en el experimento real debe haber algún factor que no ha sido tenido en cuenta en la simulación.

En primer lugar nos plantemos la posibilidad de que ese factor fuese el hecho de que, en la planta real, la acción de control, cuando es mayor que uno en valor absoluto, se ver recortada a uno (o menos uno si era negativa). Esto no ha sido tenido en cuenta en las simulaciones. Para añadir esta consideración al modelo SIMULINK hemos utilizado un bloque *saturation*, que devuelve la señal que recibe siempre que ésta se mantenga en el rango [-1,1], pero cuando se sale de él, devuelve –1 si la señal de entrada era negativa y 1 si era positiva. Éste es el aspecto de este bloque SIMULINK:

Fig. 42: bloque *saturation*

El bloque se sitúa a la salida del controlador, de forma que la acción de control que recibe la planta queda restringida al rango [-1,1]. Tras realizar esta modificación, observamos que los resultados de las simulaciones no cambian sustancialmente.

El segundo factor que se nos ocurrió que podría estar influyendo decisivamente era el error de cuantización en la medida de la salida de la planta introducido por el encoder incremental. Para comprobar esta hipótesis introdujimos en el modelo, en la señal de realimentación (salida de la planta) un bloque *quantizer*:

Fig. 43: bloque *quantizer*

Su misión es precisamente simular un error de cuantización. Tiene un parámetro para indicarle la resolución del dispositivo cuyo error de cuantización tratamos de simular. En nuestro caso, como el encoder genera 500 pulsos en una revolución (en 2*pi radiantes), su resolución es de 2*pi/500 = 0.0126. Este número será el parámetro del bloque. Este factor sí que ha resultado ser determinante respecto al comportamiento del controlador: hemos comprobado que el error de cuantización llega a desestabilizar el sistema en simulación. Valga como muestra el siguiente ejemplo, en el que con un

encoder del doble de resolución el controlador estabiliza el sistema mientras que CON el utilizado realmente no lo consigue:

- ganancia del controlador: 250
- ruido del sensor: onda cuadrada de frecuencia 0.1 Hz, amplitud 0.01.
- error de cuantización del encoder: en el primer caso 0.012 (500 muescas, no estabiliza la planta) y en el segundo 0.006 (1000 muescas, sí la estabiliza).

Resultado de la simulación para el primer caso: a partir de poco más de cinco segundos la planta se desestabiliza (el péndulo cae). La acción de control sube al máximo, uno, intentando volver a estabilizarla, pero ya es imposible:

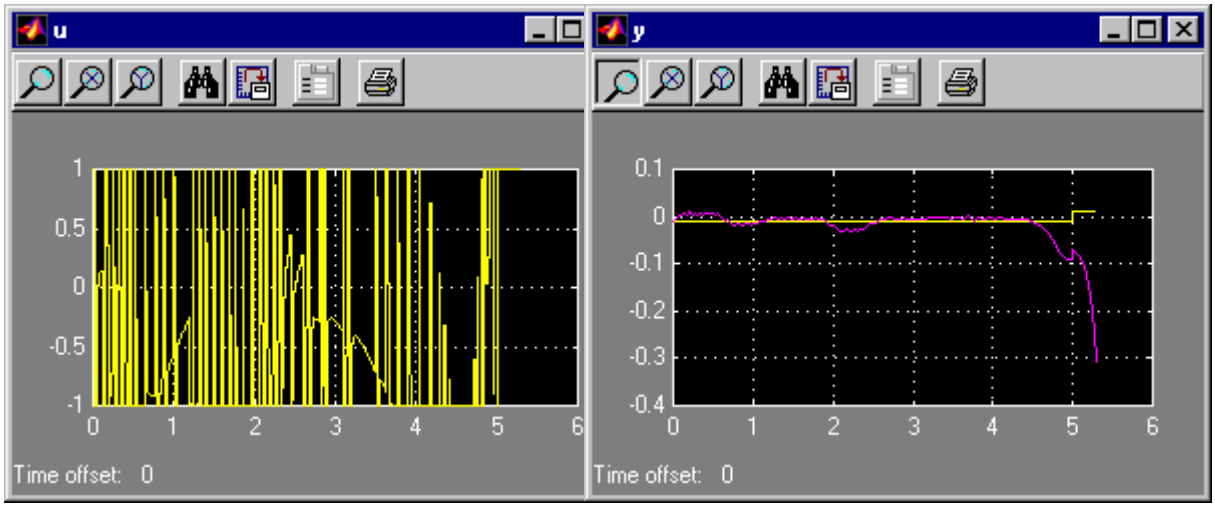

Fig. 44: resultado de la simulación con error de cuantización equivalente al del encoder real

Para el segundo caso, en cambio, la planta queda estabilizada:

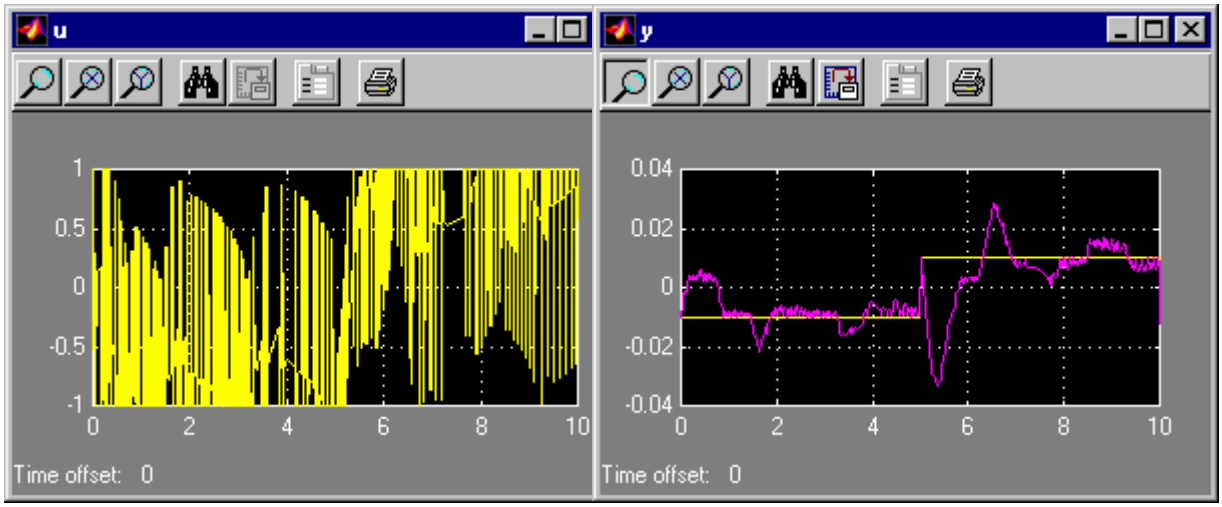

Fig. 45: resultado de la simulación con un error de cuantización correspondiente a un encoder del doble de resolución que el utilizado

La conclusión que obtenemos de estos resultados en que necesitaríamos un encoder de al menos el doble de resolución para controlar el péndulo.

5. CONCLUSIONES

El proceso de diseño de sistemas de control se encuentra en la actualidad fuertemente automatizado. La utilización de herramientas CACSD ha permitido llegar al desarrollo y puesta a punto de sistemas de control en tiempos verdaderamente cortos. En este proyecto se ha analizado una herramienta de este tipo basada en el popular entorno de programación MATLAB. La herramienta está integrada por diferentes programas, cada uno con una determinada funcionalidad: SIMULINK para la definición y simulación de sistemas, RTW/RTI para la generación automática de código para la tarjeta dSPACE 1102.

El objetivo inicial del proyecto era el análisis de las diferentes herramientas CACSD, y su aplicación a un caso práctico, el control de un péndulo invertido. La implementación del sistema de control ha supuesto además el estudio de sensores y actuadores. En particular, se ha diseñado electrónica para un encoder incremental, y la electrónica de potencia para un motor de corriente continua (véanse los Anexos III y IV).

La conclusión principal es que la herramienta se adapta perfectamente al proceso natural de diseño de un sistema de control, sobre todo en los aspectos de identificación y control (véase la Fig. 20 y comentarios relacionados). También es destacable la buena integración entre RTW y RTI, que permite que el usuario puede abstraerse de los detalles de configuración de RTW. RTI, al instalarse, directamente configura todos los parámetros necesarios para la generación automática de código ejecutable en la tarjeta de control dSPACE 1102.

Se han descrito detalladamente las características más importantes de cada uno de los programas, en un estilo tutorial, de forma que esta memoria puede servir como un primer paso para el estudio de la herramienta y su aplicación a casos prácticos de diseño.

El único aspecto importante de diseño que no cubre este conjunto de herramientas es el proceso de modelado. Sería muy deseable completar la herramienta

con un lenguaje de modelado (como por ejemplo DYMOLA o MODELICA), que permitiera una descripción de los sistemas a más alto nivel. Un trabajo futuro que completaría este proyecto sería la creación de un entorno que permita la integración de una herramienta de este tipo.

Respecto al control del péndulo invertido, se ha realizado la identificación y se ha diseñado un primer controlador. La experimentación con SIMULINK ha indicado que sería necesario un encoder de al menos el doble de resolución para controlar propiamente el péndulo.

6. BIBLIOGRAFÍA

1. dSPACE GmbH, *Solutions for Control- Catalog*, 1999.
2. dSPACE GmbH , *DS1102 COCKPIT Instrument Panel*, 1998.
3. dSPACE GmbH, *DS1102 Real-Time TRACE Module*, 1998.
4. dSPACE GmbH, *DS1102 Real-Time Interface to SIMULINK®2*, 1998.
5. dSPACE GmbH, *DS1102 Floating-Point Controller Board*, 1998.
6. dSPACE GmbH, *DS1102 Real-Time TRACE Module for MATLAB® (MTRACE)*, 1998.
7. dSPACE GmbH, *DS1102 Software Environment*, 1998.
8. dSPACE GmbH, *DS1102 DSP Device Driver*, 1998.
9. dSPACE GmbH, *DS1102 MATLAB-DSP Interface Library (MLIB)* , 1998.
10. dSPACE GmbH, *DS1102 General Installation Guide for ISA Bus Systems*, 1998.
11. dSPACE GmbH, *DS1102 Program Loader (LD31)* , 1998.

12. The MathWorks, Inc. , *Using SIMULINK*, 1998.
13. The MathWorks, Inc. , *Real-Time Workshop User's Guide*, 1997.
14. The MathWorks, Inc. , *Using MATLAB*, 1998.
15. The MathWorks, Inc. , *System Identification Toolbox User's Guide*, 1997.
16. The MathWorks, Inc. , *Control Toolbox User's Guide*, 1997.

17. G. F Franklin, J. D. Powell, and A. Emami-Naeini, *Feedback Control of Dynamic Systems*, Addison- Wesley, Wilmington, 1991.
18. Electrocraft Corp.: *DC Motors Speed Control Servo Systems*, Minnesota, 1980
19. Landau, I. D. , *System Identification and Control Design*, Prentice Hall, Englewood-Cliffs, N. J., 1990.

20. OMROM, *Advanced Automation, Catálogo General*, 1995

21. J. P. McKelvey, H. Grotch, *Física para Ciencias e Ingeniería, vol. 1*, Harla, México D.F., 1980

22. R.A. Serway, *Física* , 4ª ed, McGraw-Hill, México, D.F., 1997.

ANEXOS

ANEXO I: Señales PWM (Pulse Width Modulation) .

Una PWM es una señal periódica que se mantiene durante una parte del periodo en un cierto valor de tensión *H* constante, distinto de cero, y durante el resto del periodo a cero. La proporción del periodo total que la señal se mantiene en el valor distinto de cero se denomina *duty cycle*:

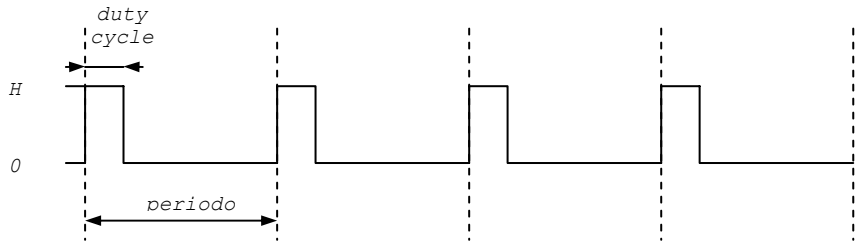

Fig. 46: forma de una PWM

Regulando el *duty cycle* de una PWM podemos controlar la intensidad media que pasa a través del circuito alimentado por la PWM: esa intensidad media será proporcional al *duty cycle*. El rango de variación [0,1] para el *duty cycle* cubre el rango [0,intensidad máxima] para la intensidad.

El *RTI* de la tarjeta proporciona un bloque SIMULINK que nos da la posibilidad de generar hasta 6 PWM's distintas, cada una con su *duty cycle*:

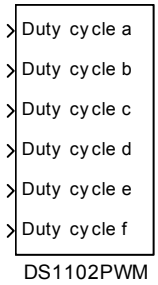

Fig. 47: bloque del RTI para generación de PWM's

Cada una de las señales tiene asociada una entrada de este bloque. A través de esa entrada se indica dinámicamente para esa señal el *duty cycle,* en el rango [0,1]. Las señales son generadas a través del subsistema de entrada / salida digital de la tarjeta. Esto hace que los valores entre los que varía la señal no sean modificables: corresponden a un uno y un cero lógicos, es decir, en esta tarjeta, a 5 y 0 voltios respectivamente.

Con una PWM que varía entre 0 y 5 voltios podemos controlar la velocidad de giro del motor en un sentido, pero no hacerlo girar en los dos sentidos. Para conseguir los dos sentidos de giro recurrimos a una electrónica de amplificación con una distribución en H, descrita en el anexo III, que requiere la generación de dos PWM's idénticas, pero tales que cada una se activa para un sólo sentido de giro. El modelo SIMULINK con que trabajemos generará de cara al motor una señal *duty cycle* que varía en el rango [-1,1], indicando el valor absoluto de este número la velocidad de giro del motor, y su signo el sentido. Además, por cuestiones de la electrónica, será necesario generar cada una de esas PWM's invertida, con lo que resultan cuatro señales en total. El bloque SIMULINK que hemos construido para generar esas cuatro señales a partir de la señal de entrada en el rango [-1,1] es el siguiente:

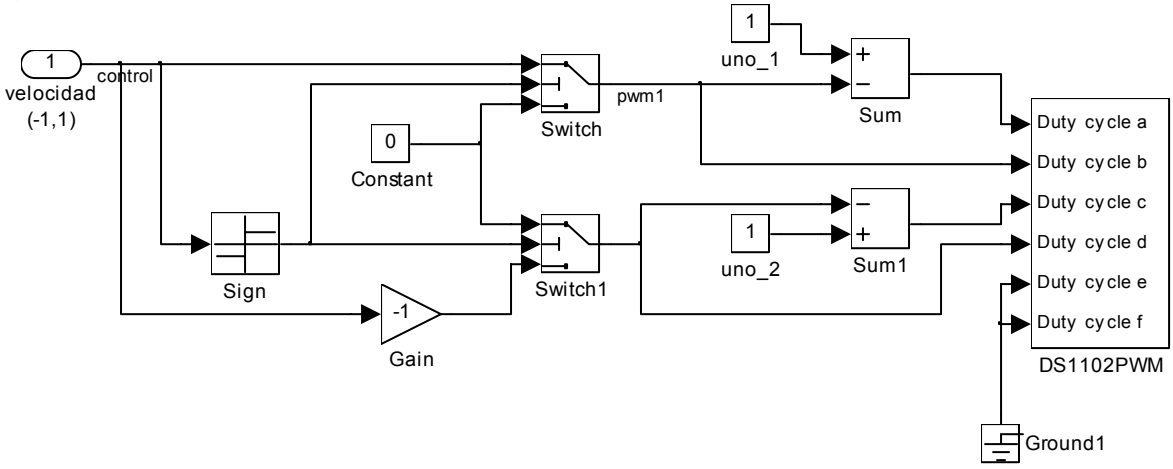

Fig. 48: generación de las PWM'w necesitadas por la electrónica de potencia

En él vemos algunos bloques SIMULINK nuevos. El bloque *sign* (signo) devuelve 1 para entradas positivas, -1 para negativas y 0 para entrada 0. El bloque *switch* selecciona como salida la entrada superior o la inferior en base a que el valor de la central esté a un lado u otro de un umbral configurable. En este caso configuramos el umbral como 0 en ambos bloques.

El funcionamiento del conjunto es de la siguiente forma: el bloque *sign* analiza el signo de la entrada. Según este signo se activará pwm1 o pwm2 (para sentido de giro positivo o negativo). La desactivación de una PWM se lleva a cabo conectándola a un cero a través de un bloque *switch*, con la señal de selección conectada al signo de la entrada. A su vez, cada una de las PWM's se genera directa e invertida. Para invertirlas, se resta la señal original a 1, de forma que cuando PWM = 1, PWM' = (1-1) = 0; y cuando PWM = 0, PWM' = (1-0) = 1.

ANEXO II: Encoders incrementales [20].

Un encoder incremental es un dispositivo utilizado para medir ángulos en el giro de un eje. No son capaces de leer directamente el valor absoluto del ángulo, sino que sólo pueden detectar que se ha producido un incremento discreto de cierta magnitud en éste, gracias a un sistema de marcas situadas a intervalos regulares a lo largo de una circunferencia que gira solidaria con el eje cuyo giro se quiere medir. El paso de una de estas marcas por cierta posición es detectable, y generará mientras sea detectada la marca un valor alto en una señal normalmente baja. Un giro mantenido es un tren de pulsos. Para determinar el sentido de giro se genera una segunda señal desfasada de la primera. Ante la detección de un pulso en la primera, la lectura del valor de la segunda determinará si el pulso detectado correspondía a un incremento del ángulo en uno u otro sentido de giro. En un giro en un sentido, tendríamos...

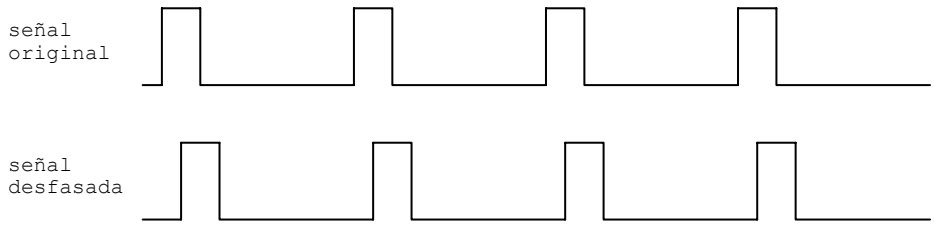

Fig. 49: señales del encoder con un giro posivo

... cuando en la señal original se detecta el principio de una muestra, en la señal desfasada aún no se ha llegado a ella. En cambio, al girar en el sentido contrario...

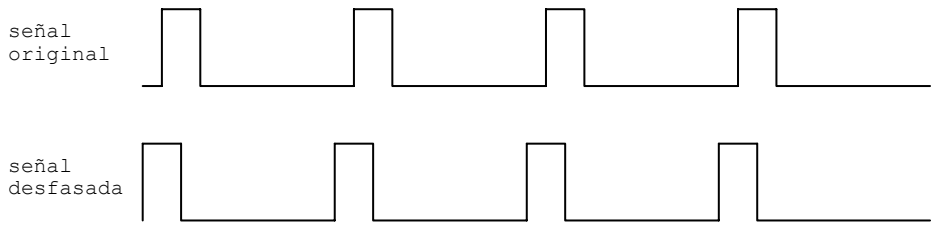

Fig. 50: señales del encoder con un giro negativo

... cuando se detecta el principio de la muestra en la señal original, en la señal desfasada ya hay un nivel alto. Esto nos permite discriminar el sentido de giro.

El número de muescas en la circunferencia se denomina resolución, y va a determinar directamente cuál es el incremento mínimo de ángulo detectable por el encoder (precisión).

Dado que el encoder no detecta ángulos absolutos, se hace necesaria la posibilidad de detectar el paso por un ángulo concreto para determinar cuál es el ángulo cero, y así poder transformar la lectura de incrementos de ángulos en ángulos absolutos. Esta señal se denomina *index* o microinterruptor. De los dos encoders utilizados en este proyecto, sólo el utilizado para medir el ángulo del péndulo tiene esta señal.

ANEXO III: Circuito de potencia utilizado[18].

Como ya se ha comentado en la presente memoria, se ha utilizado un amplificador de potencia con una disposición en H de los transistores, para permitir los dos sentidos de giro del motor con una sola fuente de alimentación:

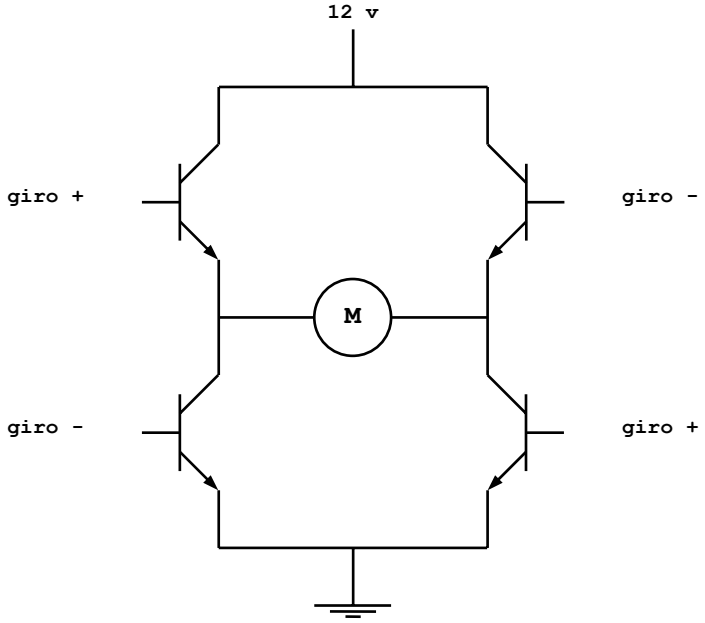

Fig. 51: diagrama base de la disposición en H de transistores

Este esquema necesita la generación de PWM's independientes para el giro en cada sentido. Para un giro positivo, se activa la PWM *giro+*, y se desactiva *giro-*, de modo que los dos transistores alimentados por *giro+* conducen, y los dos alimentados por *giro-* no. De esta forma se genera el siguiente camino para la corriente a través del motor:

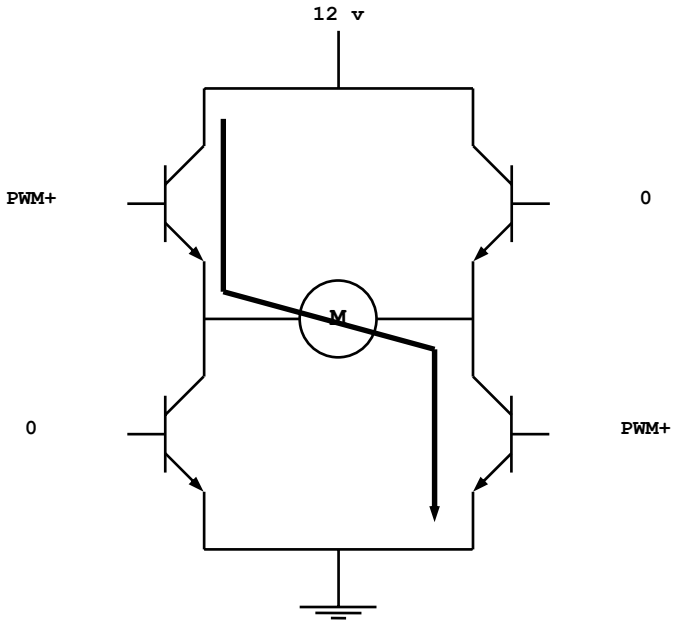

Fig. 52: camino de la corriente para giro positivo

Para giros negativos, se activa *giro-* y se desactiva *giro+*. Ahora la corriente pasa a través del motor en el sentido contrario, por lo que éste girará también en sentido contrario:

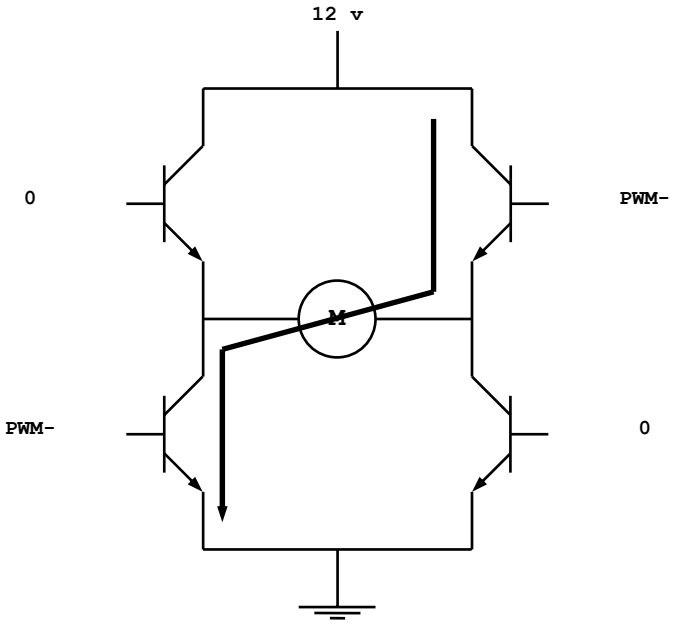

Fig. 53: camino de la corriente para giro negativo

Dado que necesitamos un factor de amplificación muy alto de la señal PWM generada por la tarjeta a la que realmente necesita el motor para moverse, cada uno de los transistores que aparecen en esta figura es en realidad un transistor Darlington formado por dos transistores npn en cascada:

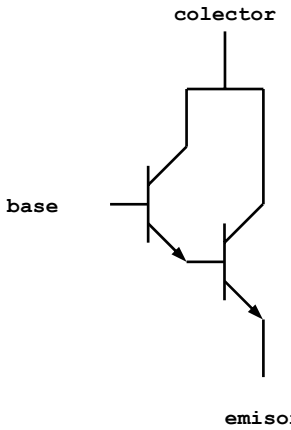

Fig. 54: diagrama de un transistor Darlington

El primero es un *BD137*, y el segundo es un *2N3055*.

Para los transistores más cercanos a la fuente en la disposición en H los 5 voltios generados por la tarjeta como uno lógico no son suficientes para que se saturen completamente, debido a la caída de tensión ocasionada por el motor. Para evitar este problema se hizo necesario elevar la tensión de los unos de la tarjeta hasta 12 voltios. Y ello mediante el siguiente circuito inversor:

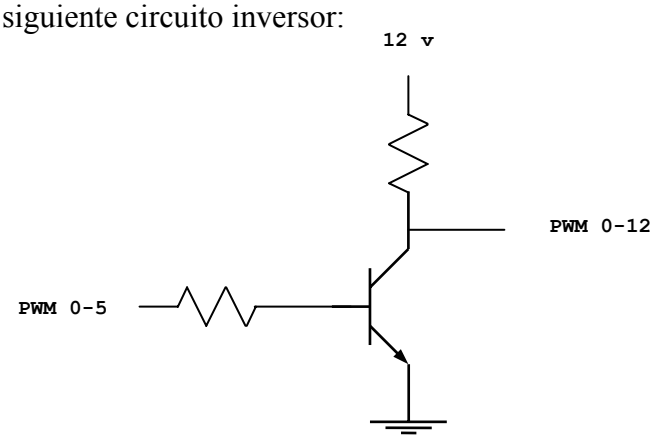

Fig. 55: diagrama del inversor utilizado

Otro problema que surgió es que un uno lógico de la tarjeta no son 0 voltios, sino 1'5 voltios. Esto hace que los transistores que reciben en su base ese supuesto cero lógico no estén realmente en corte cuando tendrían que estarlo. Para evitar ese problema se han añadido a la base de los transistores de los inversores un par de diodos en serie, que provocan la caída de tensión necesaria para conseguir que los transistores estén completamente en corte:

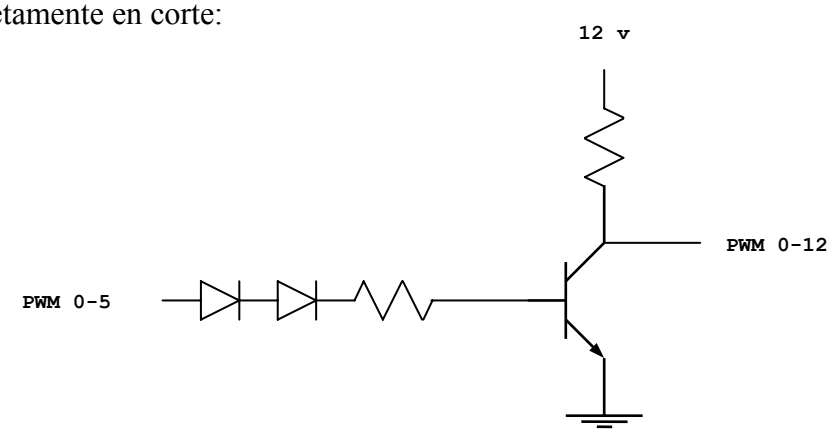

Fig. 56: inversor con dos diodos en serie en la base

Por último, añadir que en cada uno de los cuatro transistores del esquema en H básico se añade un diodo de emisor a colector. Este diodo se encarga de evitar que el transistor sufra en los cambios del sentido de giro en que él deja de conducir el paso de

una gran intensidad a su través en sentido inverso al normal en estado de saturación. Éste es el esquema para cada transistor:

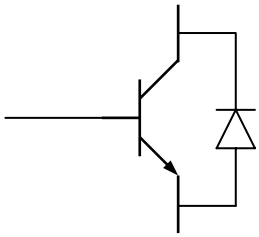

Fig. 57: diodos drenadores de corriente ante inversiones en el sentido de giro

ANEXO IV: Circuito de encoder incremental rediseñado.

El circuito del encoder incremental situado en el eje del motor ha sido rediseñado y construido. El anterior utilizaba led's y fototransistores en componentes separados para detectar el paso entre ellos de una de las aspas que giraban solidarias con el eje del motor. El nuevo circuito utiliza optoacopladores *CNY73*, que incorpora emisor y receptor en una sola pieza en forma de herradura:

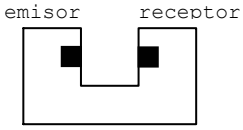

Fig. 58: optoacoplador en forma de herradura

El aspa debe pasar ahora por la ranura de la herradura para interrumpir el haz. Éste es el nuevo circuito (*E = emisor, D = receptor*):

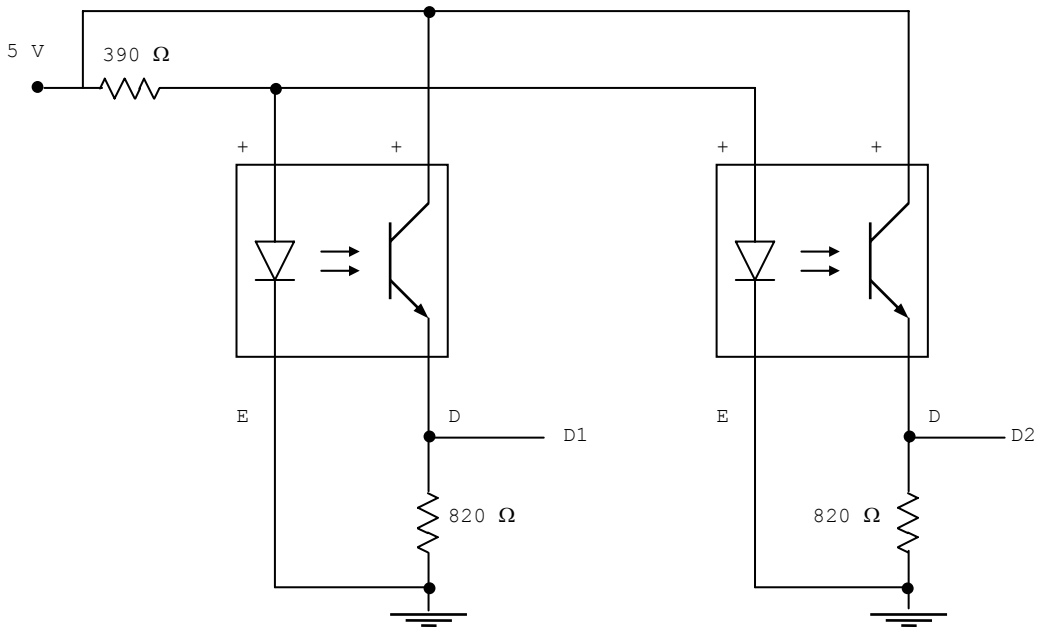

Fig. 59: esquema del nuevo circuito para le encoder incremental

ANEXO V: PRBS's (Pseudo Ramdom Binary Sequence) [19].

Las secuencias binarias pseudo aleatorias son secuencias de pulsos rectangulares, modulados en anchura, que aproximan un ruido blanco en tiempo discreto, de forma que tienen un contenido muy rico de frecuencias. El nombre *pseudo-aleatorias* proviene del hecho de que están caracterizadas por una longitud de secuencia, dentro de la cual los anchos de los pulsos varían aleatoriamente, pero para intervalos grandes de tiempo, son periódicos. Este periodo viene definido por el ancho de la secuencia.

Las PRBS's se generan mediante registros de desplazamiento con retroalimentación. La longitud de la secuencia es 2^N-1, donde *N* es el número de celdas del registro de desplazamiento. Éste sería el esquema de la generación de una PRBS para N=5. Se ha de tener en cuenta que la suma es módulo 2 y que al menos uno de los bits ha de ser inicialmente distinto de cero:

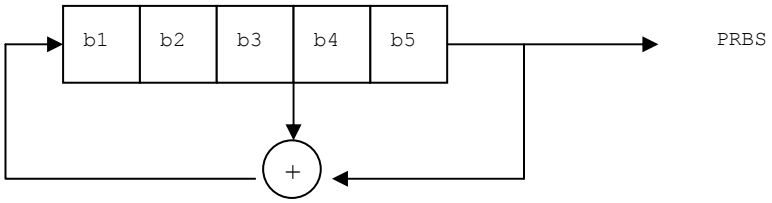

Fig. 60: generación de la PRBS mediante un registro de desplazamiento y suma módulo 2

Veamos como ejemplo la generación de los primeros valores para un registro con valor inicial 10000:

registro	salida	suma mod2 (3,5)	registro desplazado	nuevo registro
10000	**0**	0	01000	01000
01000	**0**	0	00100	00100
00100	**0**	1	00010	10010
10010	**0**	0	01001	01001
01001	**1**	1	00100	10100
10100	**0**	1	01010	11010
11010	**0**	0	01101	01101
01101	**1**	0	00110	00110
00110	**0**	1	00011	10011

Las posiciones a sumar para generar el primer bit del siguiente ciclo dependen de la longitud del registro. He aquí la tabla que da esta relación para longitudes 2-10:

N	bits a sumar
2	1 y 2
3	1 y 3
4	3 y 4
5	3 y 5
6	5 y 6
7	4 y 7
8	2 y 8
9	5 y 9
10	7 y 10

La máxima longitud de un pulso de una PRBS será igual a *N*. Esto debe ser tenido en cuenta a la hora de elegir la PRBS para identificar un sistema concreto. La PRBS deberá ser tal que en su pulso más largo dé tiempo a que el sistema reaccione ante una respuesta escalón, de lo contrario estaríamos perdiendo información. Esto se concreta en la siguiente fórmula: $NT_S > t_{R,}$ es decir, la duración del mayor pulso (N bits de longitud por el tiempo de muestreo) debe ser mayor que el tiempo de subida del sistema. Para aumentar el producto NT_S sin aumentar el tiempo de muestreo es necesario aumentar N. Como la longitud de la secuencia aumenta exponencialmente con N, esto puede llevar a secuencias demasiado largas, que no puedan experimentarse sobre el sistema físico real. Para solucionar este problema se puede generar una PRBS en que el pulso básico no coincida con el tiempo de muestreo, sino que sea un múltiplo de éste: $T_{PRBS} = pT_S$. Y esto para valores de p de hasta 4.

Según lo expuesto hasta aquí, vemos que la generación de una PRBS concreta queda paremetrizada por el tiempo de muestreo, T_S, el factor p y el número de bits del registro generador, *N*. Hemos desarrollado una función MATLAB denominada *genprbs* que devuelve la secuencia de valores de salida de la PRBS correspondiente a los parámetros que recibe. Acepta, además de estos tres, dos parámetros más: *tprev* y *bidir*. *tprev* es el tiempo que la señal generada está inactiva hasta que comienza la PRBS. *bidir* es un booleano que cuando es *true* indica a la función que el tiempo que la señal está inactiva el valor de salida no ha de ser cero, sino 0'5. Esto se utiliza para utilizar la PRBS sobre un sistema como el motor utilizado en el proyecto, que se mueve en los dos sentidos. De este modo, los extremos de la PRBS corresponderán a las máximas velocidades en cada sentido de giro, y el punto medio al punto en que el motor está parado. He aquí la función creada:

```
function PRBS=genprbs(Ts,p,N,bidir,tprev)

%genprbs(Ts,p,N,bidir,tprev)
%
%Ts: tiempo de muestreo (segundos)
%p: freq_PRBS = freq_muestreo/p
%N: nº de bits del generador de PRBS: de 2 a 10
%bidir: booleano: si 0, PRBS usada para motor girando en un sólo
%                      sentido
%                 si 1, PRBS bidireccional: el nivel cero se centra
%tprev: tiempo antes de arrancar la PRBS (segundos)

clear PRBS t t2 trigger input;

pos_a_sumar = [0 1 1 3 3 5 4 3 5 7]; % pag 146 libro "system identifi-
                                     % cation and control design"
if bidir
   niv_cero = 0.5;
else
   niv_cero = 0;
end

generador(1)=1;
PRBS(1) = 0;
for i=2:N
   generador(i) = 0;
   PRBS(i)=0;
end;


for i = 2:(2^N)
   aux = mod(generador(pos_a_sumar(N))+generador(N),2);
   for j = N:-1:2
      generador(j) = generador(j-1);
   end
   generador(1) = aux;
   for j = 0:(p-1)
      PRBS(p*(i-1)+j) = generador(N);
   end
end


t=0:((2^N*p)-1);
t=t*Ts;
t=t+tprev;  %desplazado tprev s par activar TRACE / COCKPIT / fuente
t=[0:Ts:tprev-Ts t];
t2=0:Ts:tprev-Ts;
t2=t2+(2^N*p*Ts)+tprev;
t=[t t2];

PRBS = [(niv_cero*ones(1,tprev/Ts)) PRBS
(niv_cero*ones(1,1+(tprev/Ts)))];

PRBS=[t;PRBS];
```

Fig. 61: función *genprbs.m*