ESCUELA TÉCNICA SUPERIOR DE INGENIEROS
INDUSTRIALES Y DE TELECOMUNICACIÓN

UNIVERSIDAD DE CANTABRIA

Trabajo Fin de Carrera

SISTEMA DE CONTROL DE TIEMPO REAL BASADO EN RECONOCIMIENTO DE IMÁGENES

Para acceder al Titulo de

INGENIERO DE TELECOMUNICACIÓN

Autor: Raúl Arnau Prieto

Mayo – 2005

En primer lugar quiero agradecerle a mi familia todo su apoyo y comprensión, que ha sido mucho.

A toda la gente del CTR, muchas gracias.

A mis compañeros de clase, a María, y a todos los demás que saben que debería haber puesto aquí su nombre, gracias.

ÍNDICE

RESUMEN

Este proyecto es una aplicación de tiempo real sobre el sistema operativo MaRTE, desarrollado en la Universidad de Cantabria, con el objetivo de poner a punto estrategias de control basadas en el análisis de imágenes. La aplicación concreta que se ha elegido es un demostrador para estas técnicas consistente en un plano que se puede inclinar en dos direcciones, y una bola que rueda sobre él.

El proyecto consta de dos partes diferenciadas: una relacionada con la visión y el procesamiento de imágenes, y la otra con sistemas automáticos de control. Este proyecto se desarrolla en la parte concerniente al algoritmo de control.

La finalidad del sistema es controlar la posición de la bola en el plano inclinado, para lo que (aquí entra la parte de visión) se calcula el centro de la misma a partir de imágenes capturadas por una cámara analógica. Obtenido el centro, y en función del estado del sistema (posición origen, velocidad, posición objetivo, inclinación de la plataforma, etc), se calcula la nueva inclinación de la plataforma para que la bola ruede hasta la posición deseada (mediante la parte del control). La parte de visión utiliza un driver específico que captura las imágenes y permite controlar su formato, sus dimensiones, etc.

El algoritmo de control implementado, basado en la teoría clásica de control, es un controlador PID. Si bien simplemente con un controlador PD se consigue tanto en la teoría como en las simulaciones una buena respuesta, en la práctica necesitamos añadir elementos adicionales a este algoritmo básico para que su comportamiento sea aceptable. Las más destacadas son:

- Con el objetivo de eliminar en lo posible el ruido proveniente de las capturas hemos introducido un filtro para limpiar la señal que realimenta nuestro sistema.
- Debido a la discretización debida a la cámara, también hemos necesitado introducir una lógica adicional, de forma que el controlador no intente 'sobre controlar' la posición de la bola cuando está próxima a su objetivo.

Además se ha diseñado un algoritmo para determinar la posición objetivo de la bola en función de la trayectoria deseada y de la posición que tenga en cada instante. Para ello primero generamos un conjunto ordenado de puntos sobre la superficie de la plataforma, de modo que la bola pase por ellos (o un entorno cercano) y recorra así la trayectoria que deseemos, con la velocidad que se establezca.

El movimiento de la plataforma se realiza mediante dos servomotores de corriente continua de los empleados en sistemas de radiocontrol. El control de la posición angular de los servomotores se realiza mediante pulsos periódicos de determinada anchura, técnica conocida como PWM ('Pulse Width Modulation'), y de amplitud constante e igual a 5V. Los servomotores están alimentados por una fuente también de 5V.

Para realizar la escritura de los pulsos se utiliza el puerto paralelo, mediante el driver correspondiente de MaRTE OS, usando las dos líneas de datos menos significativos del puerto. Para controlar que el tiempo en que los pulsos están a nivel alto se corresponda con el ángulo en que se deben posicionar, hacemos que el programa encargado de escribir el valor correspondiente se duerma, desde que pone la línea a nivel alto hasta que la vuelve a poner a cero, un tiempo igual a la duración de los pulsos.

Para que todos los componentes que forman el proyecto trabajen conjuntamente, hemos creado tareas Ada periódicas que se ejecutan de forma concurrente, para realizar acciones dependientes (datos compartidos), de forma sincronizada, pero con periodos distintos, ya que la adquisición de las imágenes tiene un periodo dos veces mayor que la escritura de los pulsos de los servos.

Toda la parte de visión (tanto la captura como el procesamiento de la imagen) está escrita en C, mientras que el programa principal, el control, el código de las tareas, la escritura de los pulsos, los paquetes para conversión de tipos, la instanciación de paquetes con módulos auxiliares, etc, están escritos en Ada.

Una parte muy importante de este proyecto ha sido la medición de tiempos de ejecución, tanto de las tareas periódicas como de otros procedimientos, especialmente la obtención de cotas superiores fiables de estos tiempos. El objetivo de estas medidas es que seamos capaces de planificar el sistema para que su comportamiento temporal sea predecible.

INTRODUCCIÓN

Este proyecto se centrará en el desarrollo de un controlador para el demostrador que se ejecutará en el sistema operativo de tiempo real MaRTE OS, para poner a prueba estrategias de control basadas en el análisis de imágenes. La finalidad del sistema es controlar la posición de una bola, que se encuentra sobre un plano que debemos inclinar para que ruede. El problema consiste en mantener la bola en una posición fija y moverla de una posición a otra del plano controlando los ángulos de inclinación del plano. Es por tanto necesario hacer una breve introducción a los sistemas de control, así como a los sistemas operativos y, especialmente aquellos que, como MaRTE OS, poseen ciertas características que, a diferencia de los sistemas operativos de propósito general, les permite la ejecución de aplicaciones en tiempo real.

1.1. Sistemas de control

En los últimos años, los sistemas de control automático han adoptado un papel de creciente importancia en el desarrollo y avance de la civilización y tecnología modernas. A nivel doméstico, los controles automáticos en los sistemas de calefacción y acondicionamiento de aire regulan la temperatura y la humedad de los hogares modernos para conseguir ambientes confortables. Lavadoras, vitrocerámicas, frigoríficos, lavavajillas, planchas, máquinas fotográficas, etc son otros ejemplos de sistemas de control que podemos encontrar en el hogar. En la industria, los sistemas de control automáticos se encuentran en numerosas aplicaciones, tales como el control de calidad de productos manufacturados, la automatización, control de máquinas, herramientas, sistemas modernos de tecnología espacial y de armas (guiado de proyectiles), sistemas de computadores, sistemas de transporte (pilotaje de aviones) y la robótica. Incluso problemas tales como el control de almacenes, control de sistemas sociales y económicos, y control de sistemas ambientales e hidrológicos, pueden enfocarse también desde el punto de vista de la teoría del control automático.

Un sistema de control es un conjunto de componentes (no tienen porqué ser físicos) conectados de tal manera que él mismo pueda comandar, dirigir o regular, a sí mismo o a otro sistema.

Veamos los tipos básicos de sistemas automáticos de control:

1.1.1. Sistemas de control de lazo abierto y de lazo cerrado

1.1.1.1. Sistemas de control de lazo cerrado

La siguiente figura representa de forma esquemática un sistema de control realimentado:

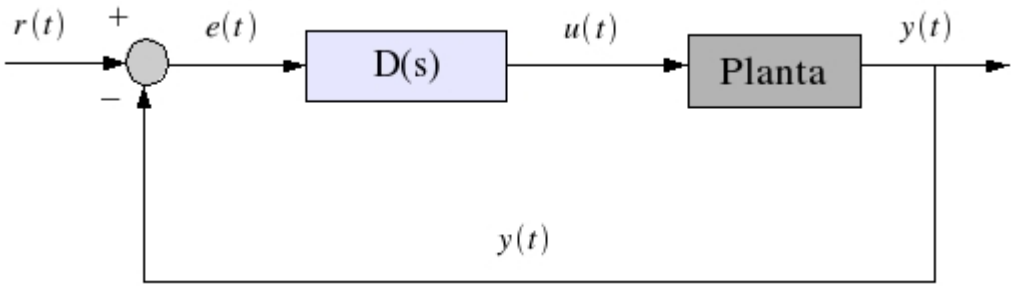

Figura 1.1. Sistema de control de lazo cerrado.

En estos sistemas la señal de salida, y(t) tiene un efecto directo sobre la acción del control. También se les conoce como sistemas realimentados. La señal de error e (t), que es la diferencia entre la señal de entrada, r(t) y la señal realimentada, y(t) (que a su vez debe ser la señal de salida, o una función de la señal de salida), realimenta el controlador, (representado en la figura por su función de transferencia D(s)) de forma que éste intenta reducir el error y llevar la salida del sistema a un valor deseado.

1.1.1.2. Sistemas de control de lazo abierto

En oposición a los sistemas de lazo cerrado, en estos sistemas la salida no tiene ningún efecto sobre el control. Esto es, la salida ni es medida, ni realimentada para compararla con la entrada. Como no realizan ninguna comparación, para cada entrada de referencia corresponde una condición de operación fija. Por lo tanto la precisión del sistema depende de su calibrado (no sólo deben calibrarse cuidadosamente, sino que para que sean útiles deben permanecer calibrados). En presencia de interferencias, un sistema de control en lazo abierto no realiza la tarea deseada. Los sistemas de control en lazo abierto sólo pueden usarse en la práctica si la relación entre la entrada y la salida es conocida, y si no hay interferencias, tanto internas como externas.

1.1.2. Controladores directos frente a indirectos

Para obtener el mejor resultado es deseable medir y controlar directamente la variable que indica el estado del sistema. El problema es que en ocasiones es difícil o

imposible de medir. Si esto ocurre, necesitamos medir otra variable (control indirecto), que puede afectar a la relación entre la calidad y la variable medida (por lo que el control indirecto no es tan efectivo como el control directo).

1.1.3. Sistemas de control adaptativos

Las características dinámicas de la mayoría de los sistemas de control no son constantes debido a muchas razones, p.ej. el deterioro de los componentes con el tiempo, o los cambios en los parámetros y del entorno. Aunque el efecto de pequeños cambios en las características dinámicas se ven atenuados en los sistemas de control realimentados, si los cambios son significativos, para una respuesta satisfactoria el sistema debe tener la habilidad de adaptarse. La adaptación implica ser capaz de autoajustarse o automodificarse de acuerdo a cambios impredecibles en las condiciones, en la estructura o en el entorno.

1.2. Sistemas operativos de tiempo real: SOTR

Un sistema operativo de tiempo real es un sistema basado en computador, el cual está íntimamente ligado a otros sistemas con los que intercambia datos y señales y sobre los que realiza funciones de control [GTI]. El aspecto fundamental que diferencia a los sistemas de tiempo real del resto de los sistemas "convencionales", es el hecho de que además de ser necesaria la satisfacción de todos los requerimientos computacionales normales, la corrección del sistema depende también del tiempo en que son producidos esos resultados. En otras palabras, un resultado correcto pierde todo significado si el tiempo de espera por el mismo sobrepasa ciertos límites preestablecidos.

Las aplicaciones de tiempo real son muy variadas y continuamente surgen nuevos campos de utilización para los mismos, siendo los más comunes los asociados a los sistemas de telecomunicaciones, transporte, robótica, multimedia, control de procesos industriales y sistemas espaciales, entre otros. Algunos ejemplos de sistemas de tiempo real que utilizamos en nuestra vida cotidiana son los que ayudan a volar a los aviones, a rodar a los trenes, los que controlan el motor o los frenos de nuestro automóvil, etc.

Todos estos sistemas tienen algo en común: están íntimamente ligados a otros sistemas con los cuales se relacionan continuamente intercambiando diversos tipos de información y efectuando sobre ellos funciones de control. Podemos considerar la existencia de un único sistema formado por otros dos subsistemas: el sistema controlado o entorno y el sistema de control.

El sistema controlado cuenta con cierto esquema que describe las magnitudes

físicas y los distintos eventos a producirse en él, la forma o secuencia según la cual éstos irán evolucionando o se irán produciendo y las relaciones existentes entre cada uno de ellos. Este esquema se halla normalmente caracterizado por procesos físicos e intervalos de tiempo más o menos bien definidos entre los eventos.

Los sistemas de control son de naturaleza más flexible, pudiéndose implementar diversos esquemas sobre diversos soportes, que pueden ir desde un sistema puramente analógico, pasando por un *hardware* microcontrolador dedicado, hasta un computador de aplicación general. El aspecto a tener en cuenta es que no solamente importa proporcionar un valor correcto a partir del estado de las variables de entrada, sino principalmente que dichos resultados sean entregados "a tiempo".

Lo que caracteriza un sistema operativo de tiempo real es su respuesta de duración acotada ante eventos internos o externos, tales como interrupciones hardware (externas), interrupciones software (internas) o interrupciones de reloj (internas). Es decir, la capacidad de predecir su comportamiento temporal.

La presencia de requisitos temporales hace que la construcción de los sistemas de tiempo real sea mucho más difícil y complicada, ya que la mayoría de los métodos y herramientas utilizados para la construcción de sistemas convencionales no contemplan tales restricciones, características en los sistemas de tiempo real.

1.2.1. Clasificación de los SOTR

En sistemas de tiempo real, a cada una de las actividades concurrentes encargadas de responder a un evento o conjunto de eventos generados por el entorno, se le denomina tarea [ALD03]. El comportamiento temporal de la totalidad del sistema puede describirse en términos de ciertas características [LED], ligadas a las tareas. Estas características comprenden:

- El *esquema de activación* de la tarea: describe cuando se debe ejecutar la tarea, pudiendo ser periódica o aperiódica.
- El *plazo de ejecución* de la actividad: indica el intervalo de tiempo máximo para la ejecución de la tarea. También llamadas *"deadlines"*.

Estas características proporcionan un criterio para la clasificación de los requisitos temporales:

- **Tiempo Real Estricto** (Hard Real Time) [GTI][REA]:

  Se considera que un sistema es de tiempo real estricto si se cumplen ***todas*** las restricciones temporales impuestas por el mundo exterior (*"deadlines"*). De no ser así, los resultados son catastróficos. Por ejemplo, un robot en un coche, que posponga activar el ABS en una situación de riesgo por bajar las

ventanillas.

- **Tiempo Real Flexible** (Soft Real Time):
  Un sistema es de tiempo real flexible si las restricciones de tiempo real impuestas por el mundo exterior, se cumplen de forma ***estadística***. De no cumplir su plazo, la tarea sigue siendo válida, si bien su valor va decreciendo con el tiempo. Un ejemplo puede ser un editor de texto, en el que no importa la rapidez con la que tecleemos mientras podamos ver los caracteres que acabamos de teclear. Un retardo de unos pocos milisegundos es permisible.
- **Tiempo Real Firme** (Firm Real Time):
  Es una mezcla de tiempo real estricto y flexible, en el que las propiedades estadísticas de la desviación temporal tienen que cumplir ciertos requerimientos. Si llegado su plazo no ha terminado, se descarta la tarea sin producir ningún resultado. Por ejemplo, un sistema de videoconferencia, que permite que no actualicemos una imagen de cada mil. A pesar de tener un plazo suave requiere una resolución temporal muy fina.

Estas definiciones son cualitativas. En un mismo sistema puede haber componentes con diferentes tipos de requisitos temporales.

1.2.2. Características de Rendimiento de un SOTR

Un evento es cualquier tipo de situación detectable desde el sistema, tanto interna, como externa, que requiera una determinada respuesta [GAY]. Existen dos medidas fundamentales del rendimiento de un Sistema Operativo de Tiempo Real, la latencia y el jitter:

La latencia es el tiempo transcurrido desde que ocurre el evento y comienza su tratamiento. Si se trata de una interrupción hardware, es el tiempo desde que se produce hasta que se ejecuta la primera instrucción de la rutina de tratamiento. Puede haber retrasos debido al acceso al bus. Una interrupción software es el tiempo desde que la señal es generada hasta que se ejecuta la primera instrucción de la tarea. Este valor sólo depende del tiempo de acceso a los registros del procesador. El jitter es la variación en el periodo normal de una tarea cuando se ejecuta de forma repetitiva.

Podemos observas muchas otras diferencias entre un sistema operativo de propósito general y uno de tiempo real, como la planificación, cambio de contexto, la gestión de memoria, resolución temporal, etc. Brevemente:

Un algoritmo (o política) de planificación es un conjunto de reglas que determinan qué tareas tienen que ser ejecutadas en cada momento [ALD03].
En sistemas de tiempo real se utilizan algoritmos de planificación en tiempo de

ejecución basados en prioridades, en los que cada tarea tiene asignada una prioridad atendiendo a la cual se resolverán los posibles conflictos para acceder al procesador. Pueden estar basados en prioridades estáticas o dinámicas (si cambian en tiempo de ejecución). Pueden atender a políticas expulsoras (si cuando una tarea que se está ejecutando tiene que dejar de hacerlo en cuanto aparezca otra de mayor prioridad lista para ejecutarse), o no expulsoras (si pude seguir ejecutándose hasta que quiera, o se quede bloqueada esperando algún recurso).

En políticas expulsoras, cuando una tarea de baja prioridad pierde la CPU en favor de otra de mayor prioridad, debe encontrarlo todo como estaba antes de haber sido expulsada. A esta operación se la conoce como cambio de contexto, y es relativamente costosa, ya que hay que grabar la información de la tarea para cuando ésta vuelva a ejecutarse.

Para preservar la información de los diferentes recursos (datos en memoria, almacenamiento secundario, etc), se debe evitar que sean utilizados concurrentemente por varias tareas, lo que se garantiza evitando que la tarea que lo usa sea expulsada por otra, aunque tenga mayor prioridad. Esta solución, denominada exclusión mutua, puede provocar que una tarea de alta prioridad bloqueada en un recurso tenga que esperar a la tarea de baja prioridad dueña del recurso, así como a que se ejecuten tareas de prioridad intermedia que puedan expulsar de la CPU a la tarea de baja prioridad: inversión de prioridad no acotada [CTR_SO][ALD03]. Los protocolos de sincronización modifican temporalmente las propiedades del planificador de tareas, resolviendo este problema.

Otros efectos indeseados son el bloqueo en cadena y el bloqueo mutuo [ALD03]. En el primero, una tarea de alta prioridad intenta acceder de forma consecutiva a varios recursos, encontrándolos todos ocupados, por lo que debe esperar en cada uno de ellos (supone hacer muchos cambios de contexto). El segundo se produce cuando ninguna de las tarea de un conjunto que comparte recursos puede ejecutar, ya que todas necesitan un recurso que se encuentra en posesión de otra tarea del mismo grupo.

Muchos SOTR incluyen también formas especiales de gestión de memoria que limitan la posibilidad de fragmentación de la memoria y aseguran un límite superior mínimo para los tiempos de asignación y liberación de la memoria. La velocidad del reparto es importante. Los sistemas de reparto de memoria estándar no son aceptables ya que el reparto de la memoria debe ocurrir en un tiempo fijo en el SOTR.

Otro problema es que la memoria puede fragmentarse cuando las regiones libres se pueden separar por regiones que están en uso. Para usar la memoria libre una solución es tener una lista vinculada LIFO de bloques de memoria de tamaño fijo.

Un reloj es un objeto que mide el paso del tiempo [CTR_SO]. La resolución temporal es el intervalo de tiempo más pequeño que un reloj puede medir. Los relojes de tiempo real miden el tiempo transcurrido desde época (00:00 del 1/1/70). Tienen una resolución mínima de 1ns y máxima de 20ms. Los temporizadores son objetos que pueden avisar a un proceso o tarea de si se ha alcanzado cierta hora o de si ha transcurrido cierto intervalo de tiempo. Cada temporizador está asociado a un reloj. En SOTR los relojes y temporizadores son de vital importancia para controlar con precisión los instantes en que deben producirse determinados eventos.

1.2.3. Núcleo de los SOTR

El núcleo es la parte del sistema operativo que proporciona la mayoría de los recursos básicos a las aplicaciones que corren en un procesador [LIN].

El núcleo de un sistema operativo de tiempo real provee una capa de abstracción que oculta al software de la aplicación los detalles del hardware del procesador sobre el que corre.

Proveyendo esta capa de abstracción, el núcleo del SO suministra a la aplicación cinco categorías principales de servicios.

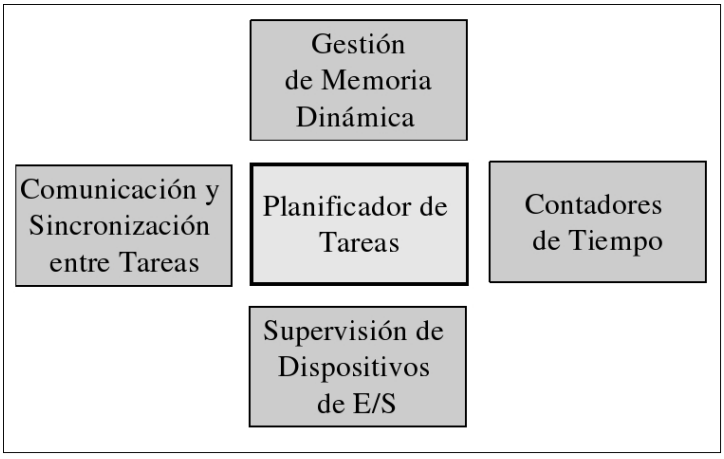

Figura 1.3: Servicios del núcleo.

La Gestión o Planificación de Tareas [LIN], en el centro de la figura (1.3), permite que los desarrolladores aplicaciones diseñen su software como un conjunto de pedazos de código, llamados tareas, cada uno con un propósito o plazo de ejecución distinto. El planificador de tareas controla la ejecución de las tareas, haciendo que se ejecuten según las prioridades que tengan asignadas.

La comunicación y sincronización entre tareas hace posible que se intercambien información unas a otras, y se coordinen para poder cooperar.
La mayoría de los núcleos de tiempo real proveen servicios de contadores de tiempo o 'timers'.

La Asignación Dinámica de Memoria permite a las tareas tomar prestados trozos de memoria RAM para usar de forma temporal. Estos pedazos de memoria se pueden pasar de una tarea a otra para comunicar rápidamente grandes cantidades de información.

El supervisor de dispositivos de entrada/salida otorga un marco de trabajo uniforme para organizar y acceder a la mayoría del hardware que se halle en el sistema operativo.

Además existen un conjunto añadido de componentes opcionales para servicios de alto nivel, como organización del sistema de ficheros, comunicación por red, gerencia de la redes, bases de datos, interfaz de usuario gráfica, etc. Aunque estos componentes añadidos a veces son más grandes y complejos que el propio núcleo, se aprovechan de sus servicios básicos.

1.3. El sistema operativo MaRTE OS

MaRTE es un sistema operativo de Tiempo Real para aplicaciones empotradas. Sigue el estándar "Minimal Real-Time POSIX.13". La mayoría de su código está escrito en Ada, con algunas partes en C y en Ensamblador.

Permite desarrollo software cruzado de aplicaciones en Ada y C usando los compiladores GNAT y GCC de GNU. En el caso de GNAT, su librería de tiempo real GNARL ha sido adaptada para correr en el núcleo de MaRTE.

MaRTE puede correr en arquitecturas *"x86"*, *"Linux"* y *"Linux_lib"*. Sólo usando una arquitectura *"x86"* podemos alcanzar realmente comportamiento de Tiempo Real Estricto. En las arquitecturas sobre Linux el programa *"mprogram"* se ejecuta como cualquier otro proceso y comparte CPU con el resto. Por lo tanto puede ser expulsado, verse afectado por el intercambio de memoria (o paginación), por el planificador de tareas, por las actividades del núcleo de Linux, etc.

La parte central del MaRTE OS es su núcleo, donde se implementa su funcionalidad básica. El núcleo tiene una interfaz abstracta de bajo nivel para acceder al hardware. Esta interfaz encapsula operaciones para el manejo de interrupciones, relojes, contadores de tiempo y cambio de contexto para los threads.

1.4. Objetivos del proyecto

Nuestro propósito general es hacer un demostrador que permita poner a punto técnicas de control por procesamiento de imagen usando el sistema operativo MaRTE OS.

La idea sobre la aplicación concreta nos la ha dado un laberinto similar al que pensamos construir, en el que una persona es la que controla el proceso: el objetivo es controlar la posición de una bola dentro de un plano cuadrado. Se actúa sobre la inclinación de esta plataforma mediante dos mandos. El usuario posee el conocimiento de la posición de la bola gracias a la visión, y mediante habilidad y práctica puede llegar a controlar la bola y hacerla recorrer un laberinto hasta la salida.

Pretendemos sustituir la vista de la persona por visión artificial, y el razonamiento y actuación por un controlador, que funcionen sin intervención externa. Entonces, el objetivo del proyecto es realizar una aplicación automatizada que corra en el sistema operativo de tiempo real MaRTE OS, y que controle la posición de una bola en la superficie del plano. Luego, si fuese posible, intentaremos complicar el problema haciendo que se recorra alguna trayectoria. Esta aplicación se va a componer de dos partes fundamentales que vamos a tratar de forma independiente, ambas imprescindibles:

- Captura de Imágenes y cálculo del Centro de masas de la Bola
- Algoritmo para Controlar la Posición de la bola en el Plano.

Este texto se centra en la segunda parte, mientras que la primera pertenece al proyecto fin de carrera de María Campo-Cossío Gutiérrez.

Para poder realizar la captura de las imágenes, se utilizará el “driver de capturadora de vídeo basadas en los chipsets BT848, BT849, BT878 y BT879”, portada al sistema operativo MaRTE por José Luis Mantecón. La posición actual la obtendremos a partir de las imágenes capturadas por la cámara, procesadas para obtener su centro de masas, mientras que para calcular la posición deseada deberemos implementar un programa que haga lo siguiente:

- Cree una trayectoria o cargue una predefinida.
- Dada la posición actual de la bola, debe calcular la siguiente a la que debe dirigirse, para que recorra la trayectoria de forma que en la siguiente imagen, la posición obtenida por la cámara esté próxima a la deseada.

De esta forma, colocando los puntos más o menos alejados, podremos controlar la velocidad con la que queremos que la bola recorra la trayectoria.

Podemos deducir de lo anterior que la posición a la que queremos que vaya la bola en la siguiente iteración no tiene porqué pertenecer necesariamente al conjunto de posiciones que forman la trayectoria, especialmente si la bola está lejos. De esta forma pretendemos hacer que la bola se mueva por el plano, ya sea recorriendo la trayectoria o aproximándose a ella, con velocidad constante.

Llegados a este punto podemos suponer que siempre vamos a conocer la posición en la que se encuentra la bola y la posición a la que queremos dirigirla. Esto no es exactamente cierto, ya que sólo tendremos conocimiento de estos datos a partir de la cámara (ya que, como hemos dicho, la posición deseada va a ser función de la posición actual), y la cámara en sí podemos tratarla como un conversor Analógico/Digital, ya que a partir de ella obtenemos muestras periódicas de una señal continua (la posición de la bola). De forma que sólo conocemos la posición en los instantes de muestreo (suponiendo despreciable cualquier retraso interno o de procesado).

A partir de las hipotéticas y recientemente calculadas posiciones de origen y destino, tenemos que actuar para que la plataforma se incline con el sentido y magnitud angular apropiados, haciendo que la bola se dirija en la dirección deseada. El encargado de hacer esto es el Algoritmo de Control.

Además, el Algoritmo de Control deberá tener en cuenta de alguna manera la velocidad de la bola, ya que como veremos, el sistema no puede ser sólo proporcional al error entre la posición actual y la deseada. En caso de ser así, sería inestable. De todas formas, a simple vista resulta evidente que si no tenemos en cuenta la velocidad de la bola a la hora de calcular la inclinación de la plataforma, no vamos a poder usar el algoritmo para acelerarla o frenarla en el caso de que necesitemos contrarrestar su velocidad, sobre todo si tenemos que hacerla cambiar de sentido.

Se deberá controlar el ángulo de inclinación del plano mediante dos servomotores. Cada servomotor está conectado al punto medio de uno de los lados del cuadrado, de forma que uno controla la inclinación en el eje 'x', y el otro en el 'y'. Para que funcione bien, el movimiento tendrá que ser independiente en los dos ejes, para lo que hará falta añadirle un contorno al plano cuadrado, que irá fijado en uno sólo de los ejes. Uno de los motores irá unido al plano, mientras que el otro irá unido al marco. Los ejes de ambos serán perpendiculares.

1.5. Descripción del sistema mecánico: bola y plano inclinado

Durante el diseño de la aplicación, pensamos que lo mejor sería utilizar materiales que no fuesen muy pesados. Así los servomotores podrían moverse más fácilmente, ya que tienen que soportar el peso de la plataforma y de su marco. Sin embargo, tienen que ser los suficientemente robustos para no doblarse ni abombarse, ya que la aplicación va a estar sometida a movimientos (posiblemente) bruscos. Además, si la superficie no es lisa o está abombada, el algoritmo de control va a funcionar peor: no controlaríamos la inclinación de la plataforma de forma uniforme en todos los puntos de su superficie, sino que sería dependiente de la forma en que esté deformada en cada punto.

Además, el plano no tiene que brillar demasiado. Esto es así porque, en caso de que la bola vaya directamente sobre la superficie, la parte encargada de hacer las capturas y el procesado de las imágenes tiene que ser capaz de discernir lo que es bola de lo que no, incluyendo los posibles brillos de los focos sobre la superficie.
Vemos un detalle de la plataforma, con el marco, patas, etc.

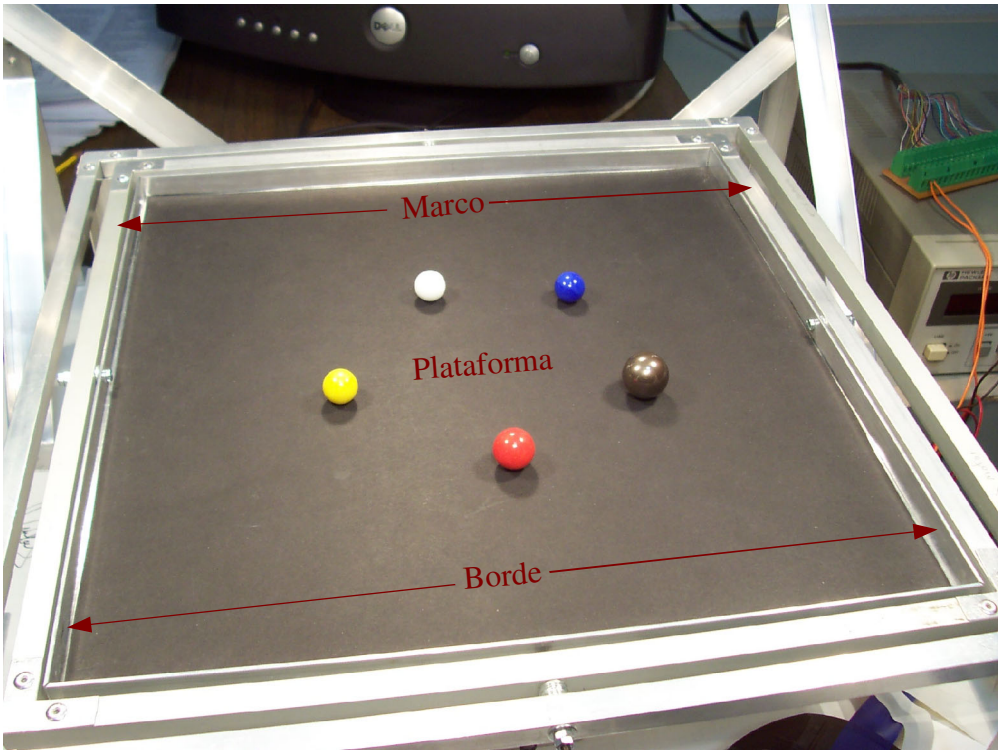

Figura 1.4: Plataforma, borde y marco.

El material elegido para la fabricación física del plano es una chapa cuadrada de aluminio mate de 1 mm de espesor y 30 cm de lado. Utilizamos molduras también de aluminio para el borde que delimita su superficie (para evitar que se salga la bola) y para las patas sobre las que está suspendida, de 11 cm de largo.

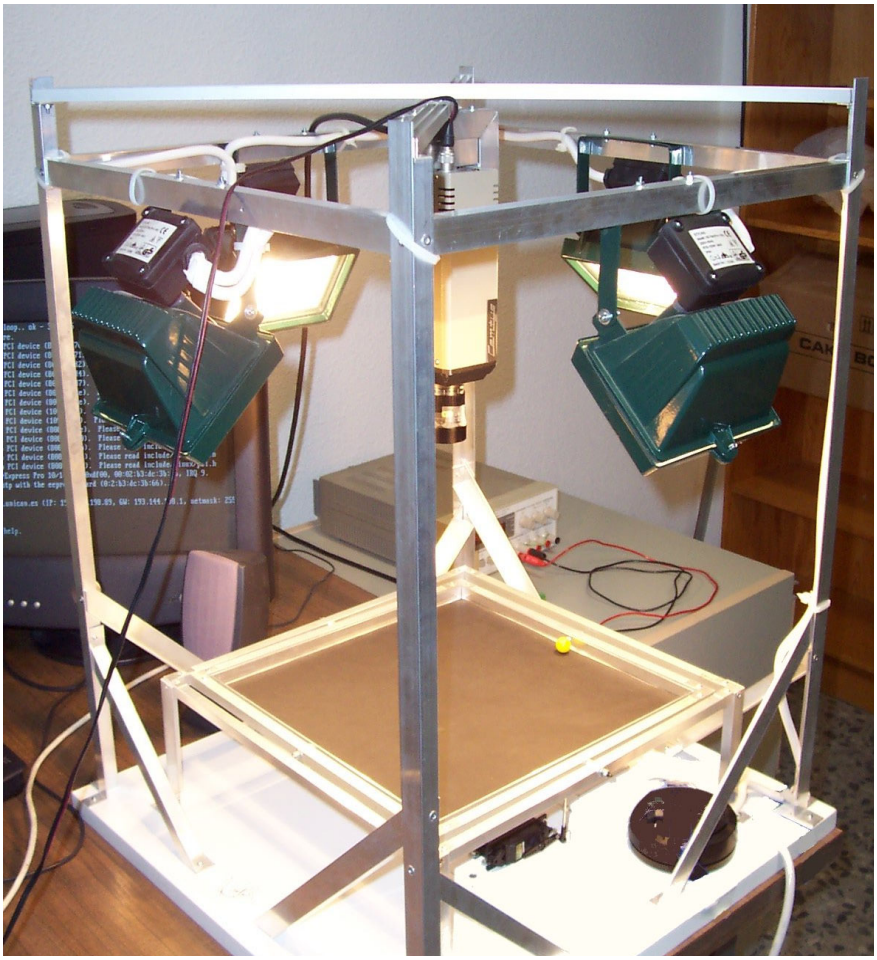

Figura 1.5: Foto de la estructura completa [(1)].

Nota [(1)]: a José María Salmón, nuestro agradecimiento especial por la realización de la estructura.

Para el doble marco del plano, que permite moverle en ambos sentidos de forma independiente, y para el marco más externo, que va sujeto al piso mediante las patas, hemos optado por utilizar varillas huecas de sección cuadradas, de 1 cm de lado.

La estructura que nos permite colgar la cámara se tiene que levantar alrededor del plano. Tiene que ser lo mas sencilla posible, para que podamos acceder fácilmente al plano y a los motores, pero debe tener la suficiente rigidez como para aguantar el peso de la cámara y de los focos del sistema de iluminación.

Para ello decidimos construir las aristas de un cubo mediante molduras de aluminio, reforzándolas en las bases con más molduras de las mismas características, y además colocamos dos molduras cruzadas en la parte superior, no sólo para que no se curve el cubo, sino porque en el punto donde se cruzan va a ir colgada la cámara.

1.6. Algoritmo de control

Se tiene que encargar de tomar los datos calculados en la parte del procesado de la imagen y convertirlo en una respuesta del sistema adecuada. Como hemos dicho, es deseable que el sistema de control sea realimentado para hacer el sistema insensible (en la medida de lo posible) tanto a las perturbaciones externas como a las variaciones de los parámetros del sistema. Sin embargo, el sistema tiene que ser estable. Idealmente, debería calcular un par de ángulos tales que, moviendo los motores a esa posición, la bola ruede hacia donde nosotros queramos.

Conocer la posición de la bola no es suficiente para poder controlarla, sino que vamos a necesitar más información. A priori necesitamos conocer:

- Su Velocidad: calculada a partir del espacio recorrido por la bola en un tiempo determinado (seguramente será el intervalo entre dos capturas).
- Su Siguiente Posición: que generamos nosotros de acuerdo a una trayectoria prefijada, dependiente de la posición y velocidad actual.

Para calcular la velocidad, necesitamos conocer las posiciones anterior y actual, además de sus respectivos instantes de muestreo.

Existen muchos tipos de algoritmos de control, desde la teoría de control clásica, pasando por el control óptimo y adaptativo, hasta los novedosos sistemas de control implementados mediante redes neuronales o lógica difusa. El algoritmo de control elegido debe ser estable, además de cumplir ciertas especificaciones como pueden ser el tanto por ciento de sobredisparo, error en estado estacionario o el tiempo de establecimiento. Además, el sistema de control debe ser capaz de rechazar o reducir

el efecto de cualquier interferencia a la salida de la planta.

Otra característica de un sistema de control es la insensibilidad a los errores en el modelo de la planta. El modelo de la planta puede contener parámetros cuyo valor no sea conocido con exactitud (de hecho, es muy probable que así sea). Si el algoritmo es lo suficientemente robusto, el sistema de control debe funcionar correctamente a pesar de estos valores incorrectos.

1.7. Esquema de esta memoria

Esta memoria está estructurada como se describe a continuación:

Capítulo 2: en él se describe el diseño del algoritmo de control, las simulaciones del sistema sólo y junto a diversos controladores, además de diversas mejoras al algoritmo.

Capítulo 3: trata sobre los servomotores utilizados para mover la plataforma, cómo funcionan y cómo controlarlos en MaRTE mediante el puerto paralelo.

Capítulo 4: describe brevemente la parte del sistema de reconocimiento de imágenes.

Capítulo 5: recoge los algoritmos necesarios para que la bola recorra una trayectoria.

Capítulo 6: habla sobre la implementación física de los programas, tiempos de ejecución, periodos, módulos auxiliares, comunicación entre tareas, interfaz con los programas de reconocimiento de imagen, etc, y el funcionamiento del sistema completo, diagramas de flujo de las tareas y los resultados obtenidos.

Capítulo 7: muestra las conclusiones obtenidas y el trabajo futuro para mejorar el sistema.

Capítulo 8: bibliografía de esta memoria.

DISEÑO DEL ALGORITMO DE CONTROL

2.1. Planteamiento del problema

Antes de entrar propiamente con la parte matemática y física del modelo, vemos una pequeña introducción con los elementos que vamos a considerar, así como de las suposiciones y aproximaciones que se han hecho para simplificar el trabajo de modelado:

Tenemos una bola en un plano. Para facilitar la descripción, suponemos que sólo tenemos un grado de libertad, es decir, que la bola sólo puede rodar a lo largo del eje 'x' del plano, manteniendo fija su posición en el eje 'y'. Luego en la práctica bastará pasar el problema a dos dimensiones usando cálculos vectoriales.

Tenemos un servomotor unido al borde de la superficie mediante el brazo del servo y una varilla. Cuando el servo gira un ángulo θ, la palanca hace que el plano se incline un ángulo α. Cuando el ángulo es distinto de cero, la gravedad hace que la bola se desplace rodando a lo largo del plano. Vamos a suponer que la bola rueda sin deslizar, y que la fricción dinámica entre las dos superficies es insignificante. Recordemos que queremos diseñar un algoritmo de control que nos permita manejar la posición de la bola.

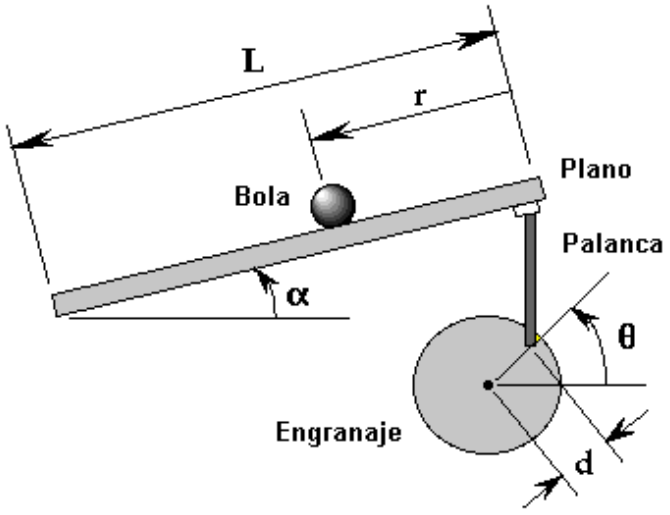

Figura 2.1: Planta del Sistema.

En la figura 2.1 aparece un dibujo esquemático del problema que debemos resolver y a continuación se exponen las constantes físicas y las variables sobre las que se va a modelar matemáticamente el sistema:

M	:	masa de la bola	=	$3\,gm$
R	:	radio de la bola	=	$7.5\,mm$
d	:	longitud de la palanca	=	$2\,cm$
g	:	aceleración de la gravedad	=	$9.8\,m/s^2$
L	:	longitud del plano	=	$15\,cm$
J	:	momento de inercia de la bola : $\frac{2}{5}\times M\times R^2$	=	$6.75\cdot 10^{-8}\,kgm^2$
r	:	posición de la bola		
α	:	ángulo que forma el plano con la horizontal		
θ	:	ángulo del servo		

2.2. Modelado del sistema

Ecuación lagrangiana del movimiento para la bola [ENG][LAU93]:

$$0=(\frac{J}{R^2}+m)\cdot\ddot{r}+m\cdot g\cdot\sin(\alpha)-m\cdot r\cdot(\ddot{\alpha})^2 \tag{2.1}$$

Aproximando esta ecuación para el ángulo $\alpha = 0$:

$$(\frac{J}{R^2}+m)\cdot\ddot{r}=-m\cdot g\cdot\alpha \tag{2.2}$$

Podemos aproximar la ecuación que relaciona el ángulo del plano con el del servomotor, mediante la siguiente ecuación lineal:

$$\alpha=\frac{d}{L}\cdot\theta \tag{2.3}$$

Sustituyendo esto en la ecuación anterior, tenemos:

$$(\frac{J}{R^2}+m)\cdot\ddot{r}=-m\cdot g\cdot\frac{d}{L}\cdot\theta \tag{2.4}$$

2.3. Función de transferencia

Tomando transformadas de Laplace en la ecuación anterior, y asumiendo que las condiciones iniciales son nulas, obtenemos:

$$(\frac{J}{R^2}+m)\cdot s^2\cdot R(s) \;=\; -m\cdot g\cdot\frac{d}{L}\cdot\theta(s) \tag{2.5}$$

Despejando, obtenemos la función de transferencia:

$$\frac{R(s)}{\theta(s)} \;=\; \frac{-m\cdot g\cdot d}{L\cdot(\frac{J}{R^2}+m)}\cdot\frac{1}{s^2} \tag{2.6}$$

Como puede verse, la función de transferencia anterior es un integrador doble. Por lo tanto, es condicionalmente estable.

2.4. Espacio de estados

Podemos poner las ecuaciones lineales del sistema en forma de espacio de estados [ENG][LAU93]. Se puede hacer utilizando como variables de estado la posición de la bola (r) y la velocidad ($\dot{r}$), y como entrada el ángulo del servo (θ). Queda:

$$\begin{bmatrix}\dot{r}\\ \ddot{r}\end{bmatrix} \;=\; \begin{bmatrix}0\,1\\ 0\,0\end{bmatrix}\begin{bmatrix}r\\ \dot{r}\end{bmatrix} + \begin{bmatrix}0\\ \dfrac{m\cdot g\cdot d}{L\cdot(\frac{J}{R^2}+m)}\end{bmatrix}\cdot\theta \tag{2.7}$$

Despejando la aceleración ($\ddot{r}$), obtenemos:

$$\ddot{r} \;=\; \frac{m\cdot g\cdot d}{L\cdot(\frac{J}{R^2}+m)}\cdot\theta \tag{2.8}$$

2.5. Simulación en MatLab®

Antes de programar en Ada el procedimiento que nos permita obtener la inclinación que tienen que tener los motores para que la bola alcance la posición deseada en un tiempo T (intervalo entre imágenes), simulamos el comportamiento del sistema completo en MatLab®. La simulación nos va a permitir saber si el sistema es estable o inestable.

Simulamos la función de transferencia del sistema obtenida anteriormente:

$$\frac{R(s)}{\theta(s)} = \frac{-m \cdot g \cdot d}{L \cdot (\frac{J}{R^2} + m)} \cdot \frac{1}{s^2} \tag{2.9}$$

Creamos un m-file con el siguiente código [ENG]:

```
% Constantes del sistema
m = 3e-3;      % masa de la bola   = 3gm
R = 7.5e-3;    % radio = 7.5mm
g = -9.8;
L = 15e-2;     % longitud plano hasta el eje
d = 2e-2;      % offset servo = 2cm
J = 2/5*m*R^2; % momento de inercia bola

k = (m*g*d)/(L*(J/R^2+m));   % numerador

num = [-k];
den = [1 0 0];
ball = tf(num,den); % FT continua
Ts = 40e-3; % periodo
ball_d = c2d(ball,Ts,'zoh'); % FT discreta

% Respuesta en lazo abierto
% -------------------------
[x,t] = step(1e-3*ball_d,100);
figure(1);
stairs(t,x); title(' balls response to a step input of 1mm');
xlabel('Time (sec)'); ylabel('Amplitude'); pause;
```

El resultado de la simulación del sistema se puede ver en la siguiente gráfica, que corresponde a la respuesta al impulso de la planta para una entrada escalón de 1mm de amplitud:

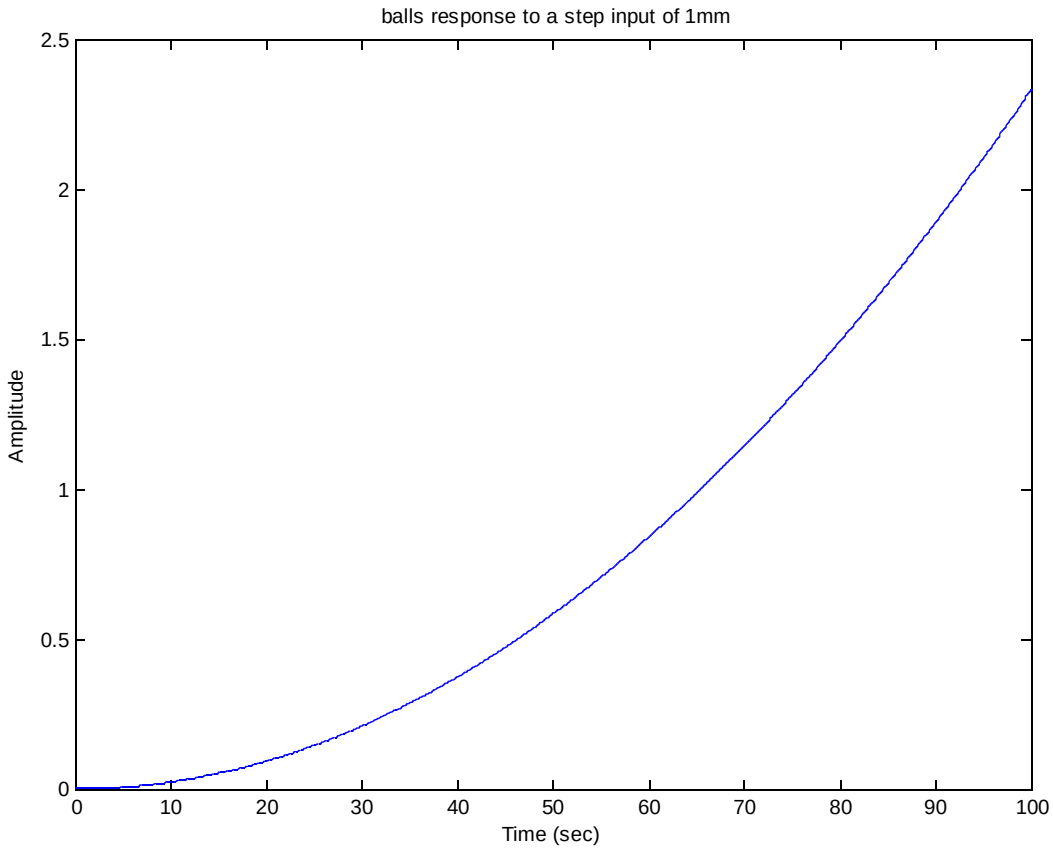

Figura 2.2: Respuesta al escalón de la planta.

En esta gráfica queda claro que el sistema es inestable en lazo abierto, por lo que la bola va a rodar hasta el final de la plataforma. Vamos a necesitar algún método para controlar la posición de la bola dentro del plano.

El método elegido ha sido un controlador PID. Su simplicidad hace que sea sencillo de programar, así como de simular.

Además, sólo depende de tres parámetros (en el caso peor, en que necesitemos hacer uso de todos ellos) así que es otra ventaja, en el supuesto de que necesitemos ajustar estos parámetros una vez hayamos implementado el controlador, sobre todo frente a otros algoritmos de control (como por ejemplo el de los espacios de estados, que sólo funciona bien si tenemos un modelo de la planta lo suficientemente preciso como para no necesitar ajustes posteriores).

2.6. Controlador PID

En el siguiente diagrama de bloques se muestra la representación del sistema junto con el controlador PID en cascada. La salida (posición de la bola) sirve también para realimentar el sistema:

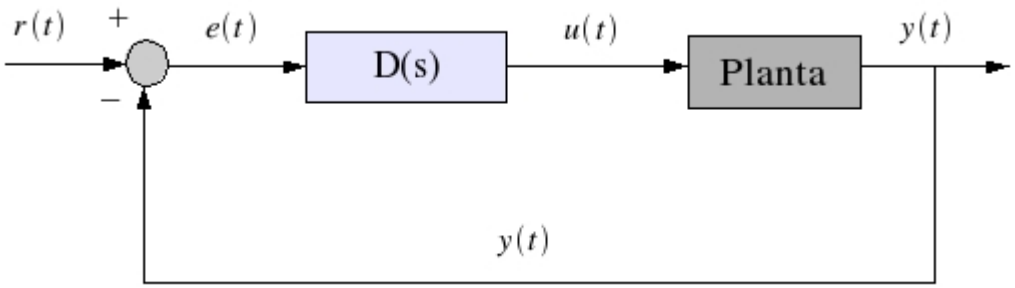

Figura 2.3: Conjunto Sistema + Controlador, con realimentación unidad.

En la figura 2.3, r(t) es la *entrada de referencia*. Si r(t) es cero todo el tiempo, entonces se conoce a D(s) como regulador, y al diseño de D(s) es conocido como problema de regulación [KUO75][OGA70]. En ese caso, el propósito de D(s) es mantener la salida y(t) a cero en el supuesto de que hubiese interferencias externas en el sistema.

Una generalización del problema del regulador es el problema del *punto fijo*. En este caso, r(t) es una constante, y el propósito de D(s) es mantener la salida a un valor de referencia constante, llamado el *punto fijo*, oponiéndose a las interferencias externas del sistema. Si r(t) no es constante durante todo el tiempo, entonces a D(s) se le llama compensador o **controlador**, y al diseño de D(s) se le llama problema de diseño de *servo*. La señal e(t) es la señal de error entre la entrada de referencia y la salida actual de la planta, y u(t) es la señal de entrada de la planta, generada por el compensador para hacer que la planta se comporte de la forma en que deseamos.

La función de transferencia del controlador PID viene dada por la expresión:

$$K_P + \frac{K_I}{s} + K_D \cdot s = \frac{K_D \cdot s^2 + K_P \cdot s + K_I}{s} \qquad (2.10)$$

La realimentación tiene módulo unidad y signo negativo, por lo que la entrada sobre la que está trabajando realmente el PID no es la posición de la bola, sino que trabaja sobre la diferencia entre el valor de entrada deseado y el valor de salida. A esta diferencia se le llama *error de seguimiento* [VAC95].

Esta señal de error se envía al controlador PID, que calcula tanto la derivada

como la integral del error. La señal a la salida del controlador es igual a K_P (ganancia proporcional) veces la magnitud del error, más K_I (ganancia integral) veces la integral del error, más K_D (ganancia derivativa) veces la derivada del error:

$$u = K_P \cdot e + K_I \int e \partial t + K_D \frac{\partial e}{\partial t} \qquad (2.11)$$

La señal *u(t)* se envía a la planta, obteniéndose la nueva salida. Esta nueva salida se vuelve a enviar al sensor para calcular la nueva señal de error. El controlador toma este error y vuelve a calcular su integral y su derivada, etc.

2.6.1. Características de los controladores PID

El control proporcional K_P tiene el efecto de reducir el tiempo de subida, y de reducir también, pero no eliminar, el error de estado estacionario. El control integrativo K_I se reduce el error de estado estacionario, pero empeora sin embargo la respuesta transitoria. Por su parte, el control derivativo K_D tiene el efecto de incrementar la estabilidad del sistema, reduciendo el sobredisparo y mejorando la respuesta transitoria.

La siguiente tabla resume los efectos que tiene cada controlador en un sistema en lazo cerrado [ENG]:

Respuesta en Lazo Cerrado	*Tiempo de Subida*	*Sobredisparo*	*Tiempo de Establecimiento*	*Error de Estado Estacionario*
K_P	*Disminuye*	*Aumenta*	~	*Disminuye*
K_I	*Disminuye*	*Aumenta*	*Aumenta*	*Aumenta*
K_D	~	*Disminuye*	*Disminuye*	~

Tabla 2.1: Efectos de las constantes PID en un sistema en lazo cerrado.

Las relaciones de la tabla anterior deben ser usadas sólo a modo de referencia, ya que las variables Kp, Kd y Ki son interdependientes. De hecho, si cambiamos el valor de una de estas variables, puede que cambie el efecto que producían las otras dos en el sistema, teniendo que reajustar su valor.

Es una buena idea intentar que el controlador sea lo más sencillo posible. Por

ejemplo, si nuestro sistema da una respuesta suficientemente buena con un controlador PI, no es necesario que implementemos el controlador derivativo, puesto que puede suponer mucho tiempo realizando los ajustes oportunos, y el resultado no mejorar proporcionalmente en la medida del esfuerzo invertido.

En la Figura (2.10) se muestra la influencia de la constante K_P en la respuesta del sistema a un escalón (apdo. 2.8).

2.6.2. Controladores PID digitales

La siguiente figura muestra el típico sistema continuo de controlador y planta realimentados, que pude implementarse en la mayoría de los casos mediante electrónica analógica.

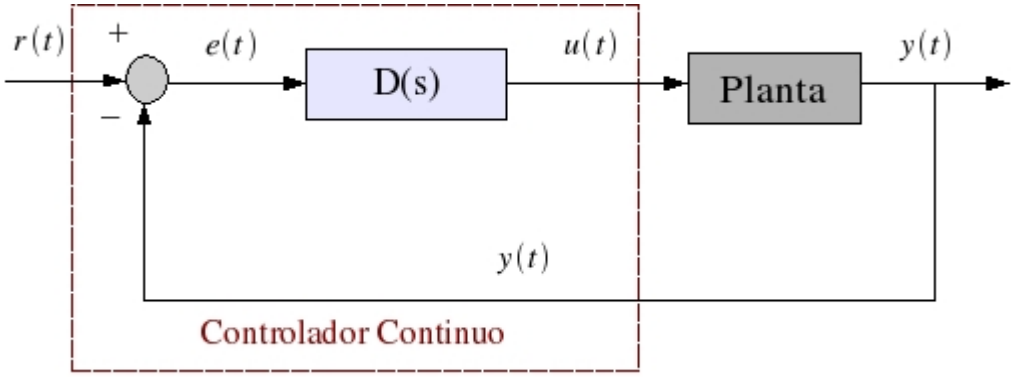

Figura 2.4: Controlador Continuo.

El controlador continuo (dentro de la línea punteada, en el dibujo anterior) puede ser sustituido por un controlador discreto que realice las mismas tareas de control, tal como se muestra en el siguiente esquema:

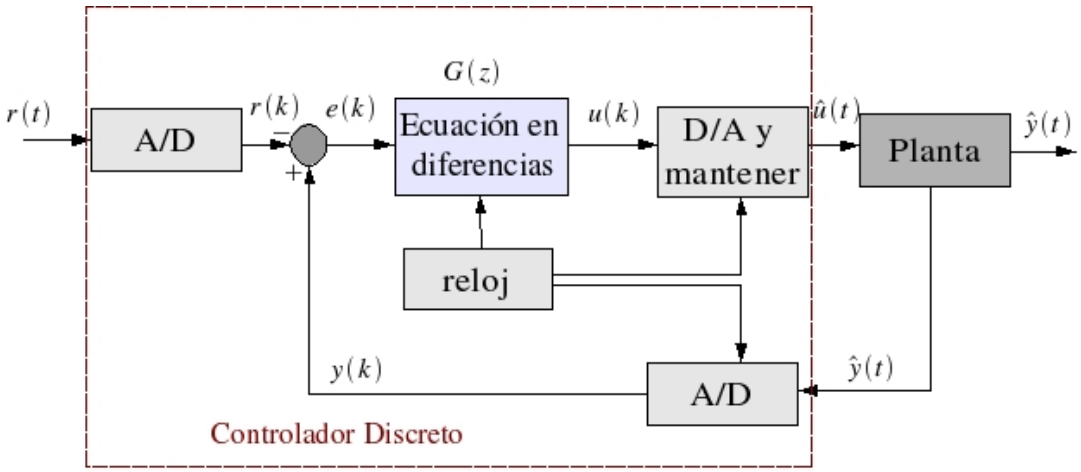

Figura 2.5: Controlador Discreto.

La diferencia básica entre estos sistemas es que el controlador analógico trabaja con señales continuas, mientras que el sistema digital opera con señales discretas, obtenidas a partir de muestras de la señal continua.

Nosotros no sabemos en todo momento la posición de la bola, por lo que no conocemos y(t). Sin embargo, recibimos muestras de esta señal cada vez que se realiza la adquisición de una imagen y los programas encargados del procesado de la misma nos dan el centro de la bola en ese instante.

Esto es equivalente a tener la señal y[k], formada a partir de muestras equiespaciadas de la señal analógica y(t), que es una señal continua en el tiempo, de la posición real de la bola.

El primer paso para diseñar un controlador discreto en el tiempo es obtener un modelo de la planta también discreto en el tiempo. Después tenemos que diseñar un compensador discreto para el modelo discreto de la planta.

El sistema de control digital se obtiene entonces conectando el compensador discreto en el tiempo a la planta analógica mediante conversores A/D y D/A.

Podemos expresar la función de transferencia del controlador PID en el dominio del tiempo continuo (ec.14), en el dominio discreto del tiempo:

$$K_P + K_I \cdot \frac{z}{z-1} + K_D \cdot \frac{z-1}{z} = \frac{(K_P+K_I+K_D)\cdot z^2-(K_P+2\cdot K_D)\cdot z+K_D}{z^2-z} \qquad (2.12)$$

Si el algoritmo de control es un PID en el dominio discreto, con una función de transferencia como en (ec.15), podemos representar la señal que llega a la planta con la siguiente ecuación:

$$u[k] = (r[k]-y[k])\cdot Kp + \left(\frac{(r[k]-y[k])-(r[k-1]-y[k-1])}{T}\right)\cdot Kd \qquad (2.13)$$

Puede observarse que el primer término, multiplicado por la constante proporcional K_P, es el error entre la posición deseada y la posición actual de la bola (o, más concretamente, el centro que obtenemos a partir de la cámara y el procesado de imágenes, ya que puede que exista un retardo desde el instante de adquisición y el momento en que se realiza el procesado).

El segundo término, multiplicando a la constante derivativa, no es más que la

derivada discreta del error, que obtenemos a partir del error anterior y el tiempo transcurrido entre dos imágenes.

2.7. Algoritmo de control PD en función de la velocidad

El algoritmo de control anterior presenta un problema: tiene un derivador a la entrada. Por lo tanto, si queremos ver su respuesta a un pulso de entrada, a la planta va a llegarle la derivada de ese pulso, que es una delta en el origen. Esto hace que la respuesta del sistema sea de la forma:

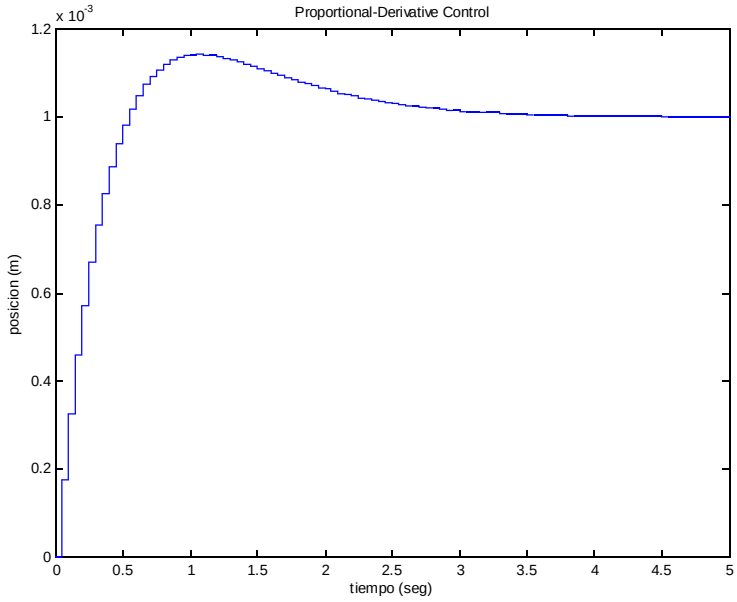

Figura 2.6: Respuesta al Escalón del controlador PD $= K_P + K_D \cdot s$.

Puede observarse que en el origen, en un entorno de $t = 0$, la bola parte con una velocidad mayor que cero, y que la magnitud de esta velocidad es bastante grande.

Esto quiere decir que el sistema tiene que comenzar regulando mucho al principio, ya que la bola parte con velocidad elevada. En la simulación de este sistema no se ve de forma clara el comportamiento real que describe la bola en el plano, sino que representa un caso hipotético en que la bola parte con velocidad inicial no nula. Por lo tanto, como se ve en la gráfica anterior, el sistema tiene que comenzar frenando. En otras palabras, el derivador que tiene este sistema a la entrada hace que su ecuación característica tenga un cero, como vamos a ver, para $s = -K_P/K_D$.

Por el contrario, si las condiciones de arranque de la bola son una posición alejada de su objetivo y con velocidad nula, es evidente que el sistema tiene que

comenzar moviendo el plano de forma que el sentido de la aceleración que le imprima a la bola la haga moverse hacia su posición objetivo.

El controlador PD de la siguiente figura resuelve este problema al aplicar el control derivativo directamente a la salida del sistema (esto es, a la posición actual de la esfera), con lo que en lugar de controlar con la derivada del error, vamos a usar la velocidad instantánea de la bola:

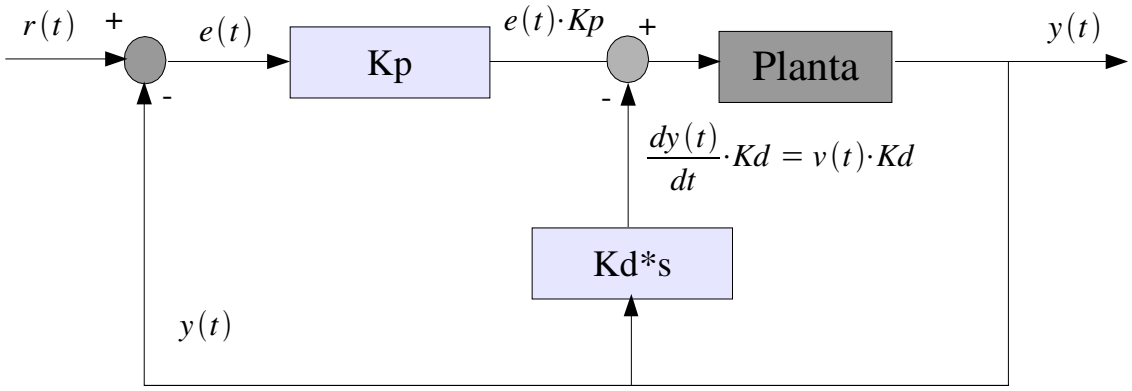

Figura 2.7: Controlador en función de la velocidad.

La nueva señal que llega a la planta responde a la siguiente expresión:

$$u(t) = \left(y(t) - r(t)\right)\cdot Kp - \frac{d}{dt}\cdot\left(y(t)\right)\cdot Kd \qquad (2.14)$$

Y teniendo en cuenta que nuestro sistema es digital, ya que trabajamos con muestras de la señal de entrada, para cada instante discreto de tiempo, tenemos la siguiente ecuación para modelar nuestro algoritmo de control:

$$u[k] = \left(y[k] - r[k]\right)\cdot Kp - \frac{\left(y[k] - y[k-1]\right)}{T}\cdot Kd \qquad (2.15)$$

En la que la velocidad y el error no son magnitudes dependientes del tiempo de forma constante, sino que las obtenemos a base de muestrear la posición instantánea de la bola. A continuación vemos una gráfica con la respuesta al escalón de nuestro algoritmo de control:

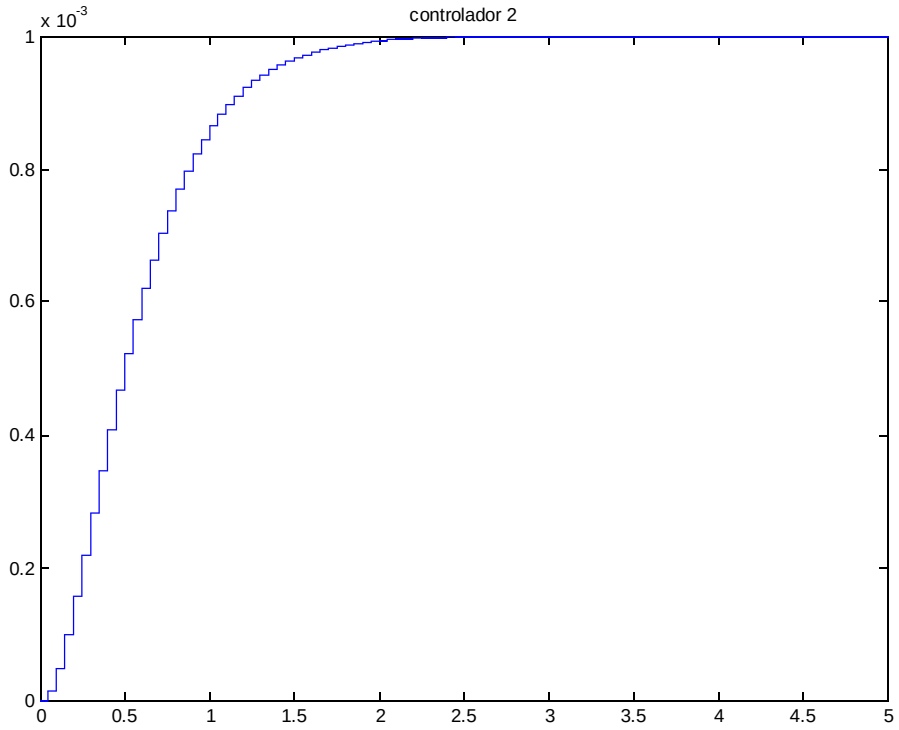

Figura 2.8: Respuesta al Escalón del sistema con controlador en función de la derivada de la señal de salida.

Comparando esta respuesta con la del algoritmo de control PD convencional de la figura 2.6, vemos que la ausencia del cero en $s = -K_P/K_D$ hace que el sobredisparo del sistema sea mucho menor en el algoritmo de control PD en el que la derivada se toma directamente de la señal de realimentación y(t), que es la principal mejora que introduce.

Aunque este otro algoritmo de control, dependiente de la derivada de la posición, consigue hacer que el sobredisparo sea menor sin la necesidad de tener que introducir el control integral, también tiene la desventaja de que hace que la convergencia sea más lenta que con el otro algoritmo (siempre que las constantes KP y KD sean las mismas), dependiente de la derivada del error.

Para la simulación de este algoritmo en MatLab®, podemos ver el controlador como un sistema doblemente realimentado:

- Por un lado la planta está realimentada por el control derivativo, que se aplica a la señal de salida. A la planta le llega la velocidad estimada de la bola, calculada a partir de dos instantes de muestreo consecutivos.
- Por otro, el sistema completo tiene realimentación unidad negativa, y a la entrada multiplicamos la señal de error de posición obtenida por la constante

proporcional.

Como vemos en la figura, la consigna de los motores (señal a la entrada de la planta) se calcula restando estas dos magnitudes:

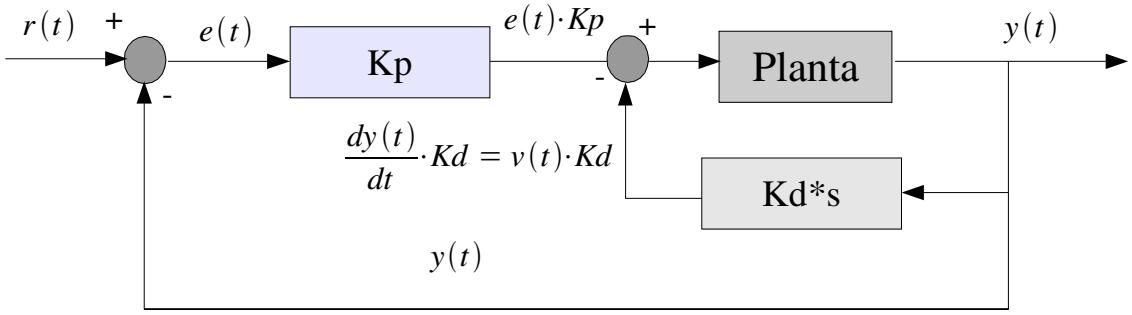

Figura 2.9: Controlador PD doblemente realimentado.

2.8. Simulación del conjunto "sistema – controlador"

Añadimos el controlador al programa MatLab® que hemos usado anteriormente para la simulación de la planta. Como están en cascada, obtenemos la función de transferencia del sistema completo usando la función *feedback*, multiplicando las funciones de transferencia de la planta y del controlador, y usamos realimentación unidad negativa.

A continuación se muestra la gráfica de la respuesta al impulso para un pequeño barrido de la constante proporcional para ver su efecto en el tiempo de establecimiento.

En esta gráfica puede observarse que aumentando la constante proporcional conseguimos que el tiempo de respuesta del sistema disminuya. Al haber añadido la constante derivativa conseguimos que disminuya el sobredisparo. De hecho, no se aprecia en la figura.

La simulación en concreto corresponde a un algoritmo de control función de la velocidad, en el dominio discreto, para 4 segundos con un periodo de muestreo de 40ms (de ahí las 100 muestras).

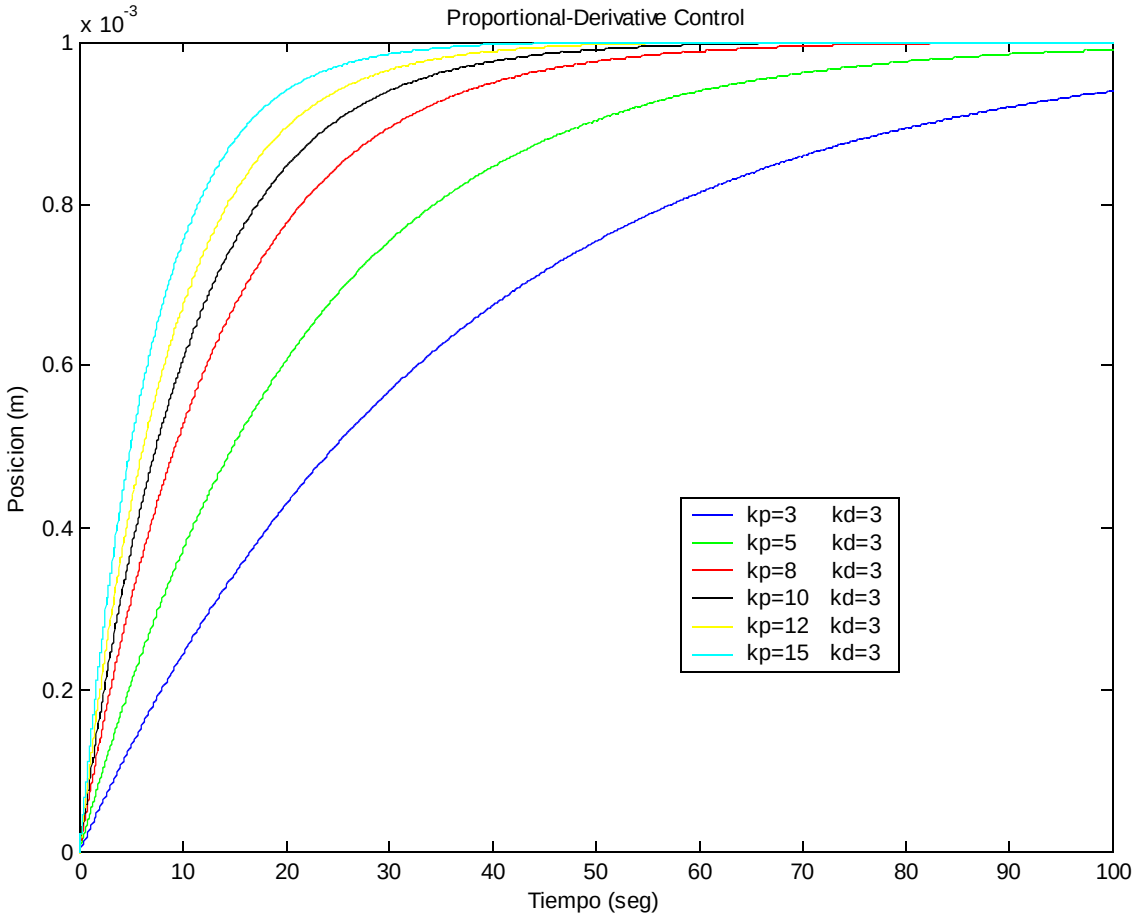

Figura 2.10: Influencia del Control Proporcional en la Respuesta al Escalón.

2.9. Cálculo de las constantes del algoritmo

La simulación del algoritmo PID mediante MatLab® nos permite encontrar las constantes del mismo de forma gráfica, que nos den el comportamiento que mejor se aproxime a nuestros requerimientos. Sin embargo, tenemos mucha libertad a la hora de elegir el valor de estas constantes, y esta simulación no nos da información de la estabilidad del sistema.

También podemos encontrar el valor óptimo para nuestro sistema de forma matemática, despejando el valor de estas constantes para colocar los polos y ceros del sistema en la posición que consideremos más favorable.

La posición de estos polos es dependiente del tiempo de convergencia.

A continuación se muestran los cálculos para obtener los polos del sistema, tanto de la planta como del conjunto formado por la planta y el controlador:

Partimos, como siempre, de la ecuación de la función de transferencia continua de la planta:

$$\frac{R(s)}{\theta(s)} = \frac{-m \cdot g \cdot d}{L \cdot (\frac{J}{R^2} + m)} \cdot \frac{1}{s^2} \tag{2.16}$$

Sustituyendo el momento de inercia por su expresión correspondiente en el caso de una esfera, tenemos:

$$\frac{R(s)}{\theta(s)} = \frac{m \cdot g \cdot d}{L \cdot (\frac{2}{5} \cdot m \cdot \frac{R^2}{R^2} + m)} \cdot \frac{1}{s^2} = \frac{g \cdot d}{L \cdot (\frac{2}{5} + 1)} \cdot \frac{1}{s^2} = \frac{5}{7} \cdot \frac{g \cdot d}{L} \cdot \frac{1}{s^2} = Gp(s) \tag{2.17}$$

Nótese que la ecuación anterior sólo depende de dos constantes conocidas, que son la longitud del plano hasta el punto sobre el que gira, L, y la longitud del brazo del servo, d.

Calculamos ahora la función de transferencia del sistema con controlador:

$$M(s) = \frac{Gp(s) \cdot Gc(s)}{1 + Gp(s) \cdot Gc(s)} \tag{2.18}$$

Donde Gp y Gc son las funciones de transferencia de la planta y del controlador, respectivamente.
Conocemos la función de transferencia del controlador:

$$Gc(s) = Kp + Kd \cdot s \tag{2.19}$$

Y para simplificar la función de transferencia de la planta:

$$Gp(s) = \frac{5}{7} \cdot \frac{g \cdot d}{L} \cdot \frac{1}{s^2} = \frac{a}{s^2} \tag{2.20}$$

Por lo tanto, M(s) queda:

$$M(s) = \frac{Gp(s)\cdot Gc(s)}{1 + Gp(s)\cdot Gc(s)} = \frac{(Kp + Kd\cdot s)\cdot\frac{a}{s^2}}{1 + (Kp + Kd\cdot s)\cdot\frac{a}{s^2}} = \frac{(Kp + Kd\cdot s)\cdot a}{s^2 + a\cdot(Kp + Kd\cdot s)} = \ldots$$

$$\ldots = \frac{(Kp + Kd\cdot s)\cdot a}{s^2 + Kd\cdot a\cdot s + Kp\cdot a} = \frac{Z(s)}{P(s)} \qquad (2.21)$$

Análogamente, para el controlador en función de la velocidad obtenemos una ecuación característica similar, en la que el denominador no cambia, mientras que conseguimos eliminar el cero del numerador:

$$M_2(s) = \frac{Kp\cdot a}{s^2 + Kd\cdot a\cdot s + Kp\cdot a} = \frac{K}{P(s)} \qquad (2.21.bis)$$

Por lo tanto, el resto de cálculos va a ser aplicable a ambos algoritmos. Es decir, que ambos algoritmos van a tener incluso el mismo tiempo de subida (función de la posición del polo dominante), y en lo único en que se van a diferenciar va a ser en la forma en que convergen al valor final.
Calculamos los polos:

$$P = \frac{-Kd\cdot a \pm \sqrt{(Kd\cdot a)^2 - 4\cdot Kp\cdot a}}{2} \qquad (2.22)$$

Ahora tenemos que elegir donde queremos colocar los polos del sistema en lazo cerrado.

Empezamos viendo la localización de los polos de la planta (sistema en lazo abierto sin controlador). La representación en el plano 's' de las localizaciones posibles de los polos y ceros se denomina lugar de las raíces, o rlocus. Veamos una simulación en MatLab® del lugar de las raíces de la planta:

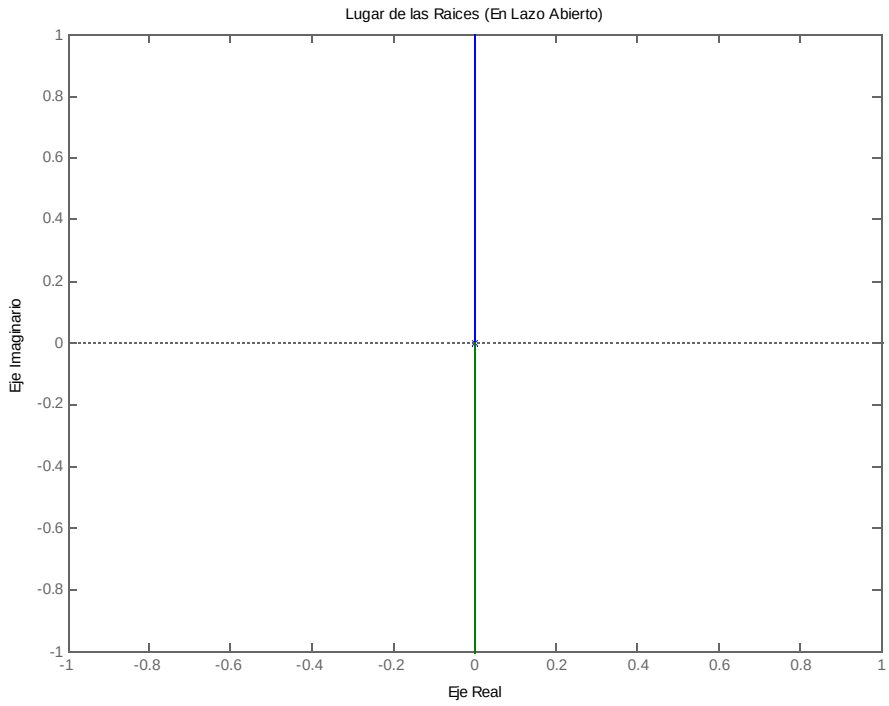

Figura 2.11: Lugar de las raíces de la planta.

El sistema tiene dos polos en el origen, que van al infinito a lo largo del eje imaginario. Por lo tanto, por el criterio de estabilidad de Nyquist, el sistema es inestable, ya que los polos no se encuentran en el semiplano izquierdo.

Si tuviésemos algún polo en el semiplano derecho, el sistema en lazo cerrado sería inestable.

Los polos que estén más cerca del eje imaginario son los que tienen la mayor influencia en la respuesta en lazo cerrado, así que, aunque el sistema tenga muchos polos, puede que se comporte como un sistema de orden uno o dos, en función de la localización de los polos dominantes.

La localización de los polos depende de nuestro criterio de diseño:

- Sobredisparo
- Tiempo de establecimiento.

El tanto por ciento de sobredisparo y el tiempo de establecimiento se definen a partir de la función de transferencia del sistema de segundo orden:

$$G(s) = \frac{\omega_n^2}{s^2 + 2 \cdot \xi \cdot \omega_n \cdot s + \omega_n^2} \tag{2.23}$$

En ocasiones se conoce este sistema como sistema de segundo orden estándar. El parámetro ξ es el damping ratio y ω_n es la frecuencia natural. Asumimos que $0 < \xi \leq 1$. En la siguiente figura se muestra una respuesta típica de G(s), en la que se incluye en tanto por ciento de sobredisparo y el tiempo de establecimiento:

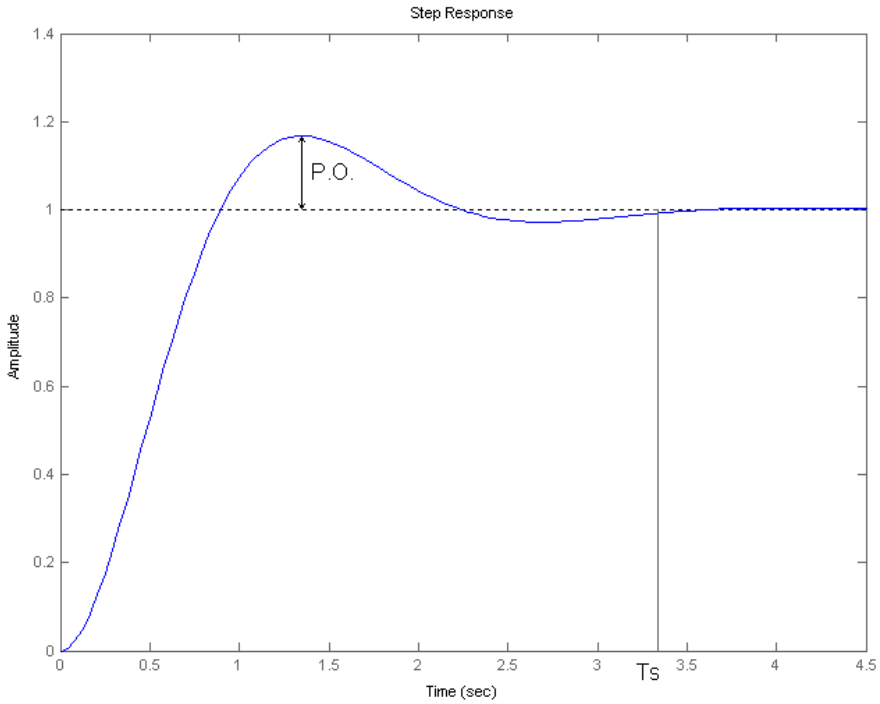

Figura 2.12: Respuesta al impulso típica de un sistema de segundo orden.

Los polos de G(s) están localizados en:

$$s_{1,2} = -\sigma \pm j \cdot \omega_d \tag{2.24}$$

donde:

$$\sigma = \xi \cdot \omega_n$$

$$\omega_d = \omega_n \sqrt{(1 - \xi^2)}$$

Aplicando transformadas de Laplace, se puede comprobar que la respuesta al impulso de G(s) es:

$$y(t) = 1 - e^{-\sigma t}\left(\cos(\omega_d t) + \frac{\sigma}{\omega_d}\cdot\sin(\omega_d t)\right) \tag{2.25}$$

Utilizando esta respuesta al impulso, podemos expresar el tanto por ciento de sobredisparo y el tiempo de establecimiento en función de ω_n y de ξ:

$$Ts \approx \frac{4.5}{\xi\cdot\omega_n} \tag{2.26}$$

$$P.O. = 100\cdot\exp\left(\frac{-\xi\cdot\pi}{\sqrt{(1-\xi^2)}}\right), \qquad 0 < \xi \leq 1. \tag{2.27}$$

Estas expresiones muestran que la velocidad de respuesta (Ts) es gobernada por la parte real de la localización de los polos (una respuesta rápida requiere que los polos estén "lejos" dentro del semiplano izquierdo), y que el sobredisparo sólo es función de ξ.

Aunque estos valores podemos fijarlos libremente dentro de un rango, el comportamiento del sistema va a depender mucho de los valores que tomemos. En nuestro caso parece más que razonable que, teniendo en cuenta que el control se realiza cada 40ms, el tiempo de establecimiento no sea mayor a 3 segundos, mientras que intentaremos acotar el sobredisparo por un valor lo suficientemente pequeño, como por ejemplo el 5% del valor máximo. Este criterio de diseño es posible que en el futuro no nos sirva, por lo que tendríamos que cambiarle y volver a colocar los polos del sistema para cumplir los nuevos requerimientos, que se traducirá, como vamos a ver, en cambiar las constates KP y KD de nuestro algoritmo.

Una vez fijado el criterio de diseño, podemos dibujar la zona del plano 's' en que podemos localizar los polos. Para dibujar esto en MatLab®, utilizamos la función *sgrid*, para lo que tenemos que despejar ξ y ωn de las ecuaciones anteriores. Utilizando Ts = 3seg y P.O. = 5%, obtenemos $\xi = 0.7$ y $\omega_n = 1.9$.

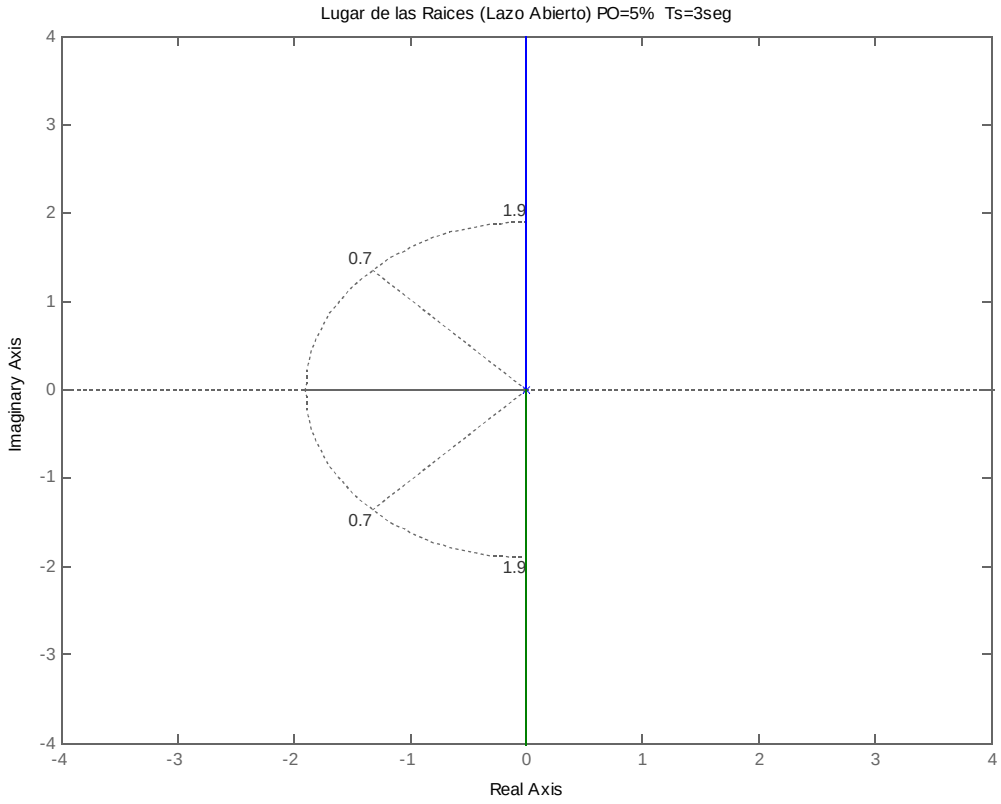

Figura 2.13: lugar de las raíces de la planta junto con el criterio de diseño.

- La zona del plano entre las dos diagonales representa localizaciones de los polos que harían que el tanto por ciento de sobredisparo sea menor que un 5%.

- El área fuera de la línea curva representa las localizaciones para las que el tiempo de establecimiento es menor a 3 segundos.

Ahora simulamos la planta junto con nuestro controlador PD, y calculamos el lugar de las raíces del sistema completo. La gráfica correspondiente se muestra a continuación. Hay que notar que para dibujar el lugar de las raíces es necesario conocer las constantes del algoritmo de control K_P y K_D, a las que hemos dado un valor arbitrario, ya que son precisamente los valores solución de nuestros cálculos.

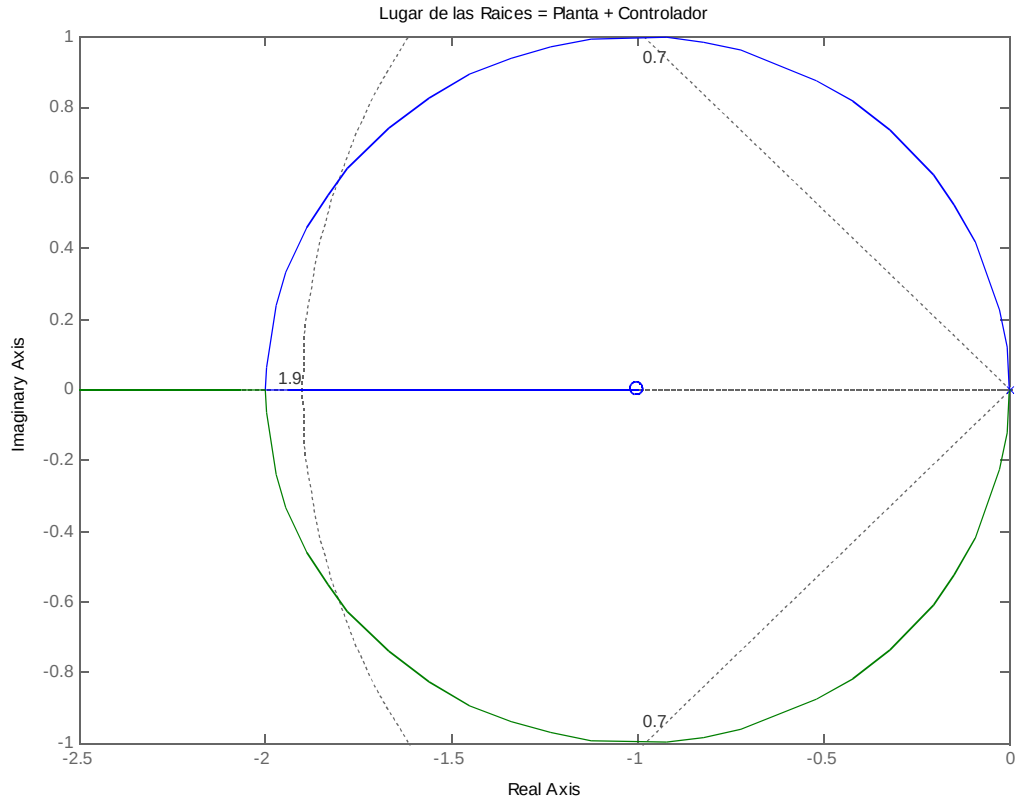

Figura 2.14: lugar de las raíces del conjunto planta-controlador (para $K_P = K_D = 3$)

Esta gráfica es muy útil para ver cómo influyen entre sí las constantes del algoritmo y los requerimientos temporales:

- Si hacemos que el tiempo de establecimiento sea menor, la convergencia se hace más rápida, pero el semicírculo se haría también más grande, por lo que para tener ambos polos fuera de este semicírculo, tendríamos que hacer que el punto en el que los dos polos son reales e iguales, estuviese situado mucho más a la izquierda. Esto se consigue aumentando la constante proporcional.

- Si por el contrario relajásemos más el tiempo de establecimiento, el semicírculo se haría más pequeño, por lo que podríamos colocar los polos a la izquierda del semicírculo para un valor menor de K_P.

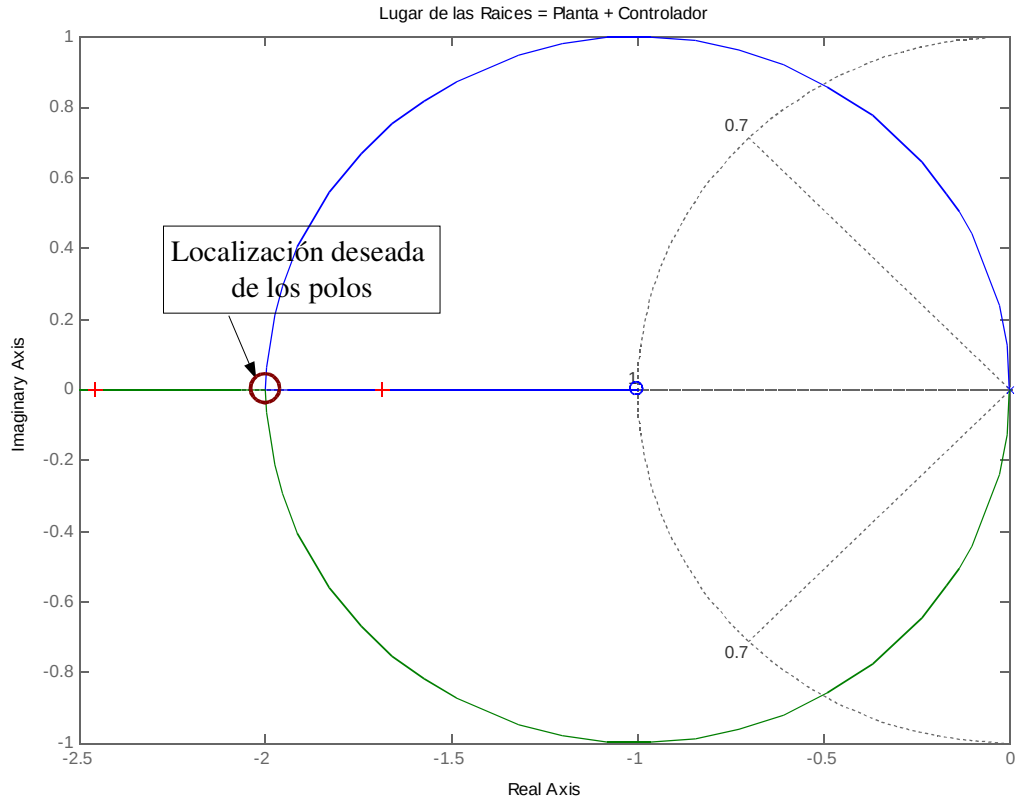

Figura 2.15: Lugar de las raíces con tiempo de establecimiento grande.

Así mismo, la localización de los polos depende del valor de las constantes K_P y K_D. En la gráfica anterior parece que podemos elegirlos arbitrariamente, incluso habiendo fijado ya el valor de las constantes, pero esto es así sólo porque el dibujo del lugar de las raíces se hace barriendo una constante que multiplica todo el sistema, con la intención de poder elegir así, gráficamente, la constante del controlador.

Vamos a utilizar el criterio de que nuestros dos polos sean reales e iguales. No es necesario, ya que podríamos haberlos puesto separados, siendo uno de ellos, el más cercano al origen, el polo dominante. Con la localización de este polo podríamos haber encontrado las constantes del algoritmo para el mismo tiempo de subida que con el polo doble.

Despejamos las constantes KP y KD de la ecuación (25) para tener un polo doble, Real y negativo:

$$P = -\frac{Kd \cdot a}{2} \tag{2.28}$$

Entonces, la relación que se tiene que cumplir entre KP y KD para que sea cero la raíz cuadrada de la ecuación (22), que nos garantiza que los polos son reales e iguales:

$$\sqrt{(Kd \cdot a)^2 - 4 \cdot Kp \cdot a} = 0 \;\rightarrow\; Kp = \frac{Kd^2 \cdot a}{4} \tag{2.29}$$

Además, la constante 'a' es conocida:

$$a = \frac{5}{7} \cdot \frac{g \cdot d}{L} = \frac{5}{7} \cdot \frac{9.8 \cdot 2\,cm}{15\,cm} = 0.933 \tag{2.30}$$

Por lo tanto, una vez fijemos la posición del polo doble, tendremos inmediatamente de las ecuaciones (28) y (29) el valor de las constantes KP y KD.

Para la localización de los polos vamos a tener en cuenta en tiempo de subida de la respuesta al escalón. Para un polo doble se puede calcular el tiempo de subida mediante la siguiente expresión:

$$Tr = \frac{3.5}{P} \tag{2.31}$$

Donde el tiempo de subida es el tiempo que tardamos en pasar del 10% al 90% del valor del valor final en la respuesta al escalón del sistema. Vamos a fijar un tiempo de subida de Tr = 1seg. Por lo tanto el polo doble:

$$Tr = \frac{3.5}{P} \;\Rightarrow P = \frac{3.5}{1\,seg} = 3.5 \tag{2.32}$$

Los valores correspondientes de las constantes del controlador son entonces:

$$\begin{aligned} -3.5 = -\frac{Kd \cdot a}{2} \;\Rightarrow\; Kd &= \frac{3.5 \cdot 2}{0.933} \approx 13 \\ Kp &= \frac{Kd^2 \cdot a}{4} \approx 7.5 \end{aligned} \tag{2.33}$$

2.10. Influencia de los polos en el sistema de control discreto

En control digital, vamos ser capaces de especificar el comportamiento del sistema en lazo cerrado sólo en los instantes de muestreo. El sistema de control digital es descrito por una función de transferencia discreta que vamos a llamar G(z). Nos gustaría que la respuesta al impulso de G(z), y[k], sea una versión muestreada de la respuesta al impulso de G(s), y(t). En otras palabras, nos gustaría que:

$$y[k] = y(t)|_{t=kT} \tag{2.34}$$

Nos gustaría obtener la respuesta a la siguiente pregunta: ¿dónde deben localizarse los polos de G(z) para que se satisfaga (XX)?

La respuesta a esta pregunta se desarrolla de la siguiente manera:
La respuesta al impulso de G(z) se obtiene a partir de muestras de de la respuesta al impulso en (XX-respuesta y(t)):

$$y[k] = y(kT) = 1 - (e^{-\sigma T})^k \left(\cos(\omega_d k T) + \frac{\sigma}{\omega_d} \sin(\omega_d K T) \right). \tag{2.35}$$

Para que y[k] sea la respuesta al impulso de G(z), la transformada z de y[k] debe ser el producto de la correspondiente transformada z del pulso de entrada, con G(z):

$$Z\{y[k]\} = \frac{z}{z-1} G(z) \tag{2.36}$$

Si tomamos la transformada z de y[k] en (35) y ponemos el resultado en la misma forma que en la ecuación (36), seremos capaces de encontrar G(z):

$$Z\{y[k]\} = \frac{z}{z-1} - z\left[\frac{n_1(z) + (\sigma/\omega_d) n_2(z)}{z^2 - 2 r z \cos(\omega_d T) + r^2} \right] \quad \text{,donde} \quad r = e^{-\sigma T} \tag{2.37}$$

Y los polinomios $n_1(z)$ y $n_2(z)$ no son importantes para estos cálculos. Simplificando la expresión, llegamos a:

$$Z\{y[k]\} = \frac{z}{z-1}\cdot\left[1 - \frac{(z-1)[n_1(z) + (\sigma/\omega_d)n_2(z)]}{z^2 - 2rz\cos(\omega_d T) + r^2}\right] = \ldots$$

$$\ldots = \frac{z}{z-1}\frac{N(z)}{z^2 - 2rz\cos(\omega_d T) + r^2} \tag{2.38}$$

Que está en la misma forma que (36). Entonces, los polos del sistema discreto son las raíces del denominador de (38):

$$z^2 - 2rz\cos(\omega_d T) + r^2 \text{ , que son:}$$

$$\begin{aligned} z_{1,2} &= r\cos(\omega_d T) \pm jr\sqrt{(1 - \cos^2(\omega_d T))} \\ &= r[\cos(\omega_d T) \pm j\sin(\omega_d T)] \\ &= (e^{-\sigma T})(e^{\pm j\omega_d T}) \\ &= e^{s_{1,2}T} \end{aligned} \tag{2.39}$$

Donde la última línea sigue de la ecuación (30), en la que $s_{1,2}$ son los polos del sistema de segundo orden estándar. Sustituyendo esto en la ecuación anterior (39) vemos que:

$$\left|(z_{1,2})\right| = e^{-\sigma T} \tag{2.40}$$

De forma que $-\sigma$, la parte real de los polos del plano s, directamente controla la magnitud de los polos del plano z. Entonces, una respuesta al impulso rápida requiere que los polos estén lejos del origen y dentro del semiplano izquierdo del plano s, o equivalentemente polos en el plano z de radio pequeño. El tanto por ciento de sobredisparo de la respuesta al impulso depende de ξ, y el efecto de ξ en la localización de los polos en el plano z no queda lo suficientemente clara en (40).

Sin embargo, utilizando los polos de la ecuación (40) podemos dibujar contornos de ξ y ω_n constantes en el plano z, exactamente igual que hemos dibujado el lugar de las raíces en el plano s.

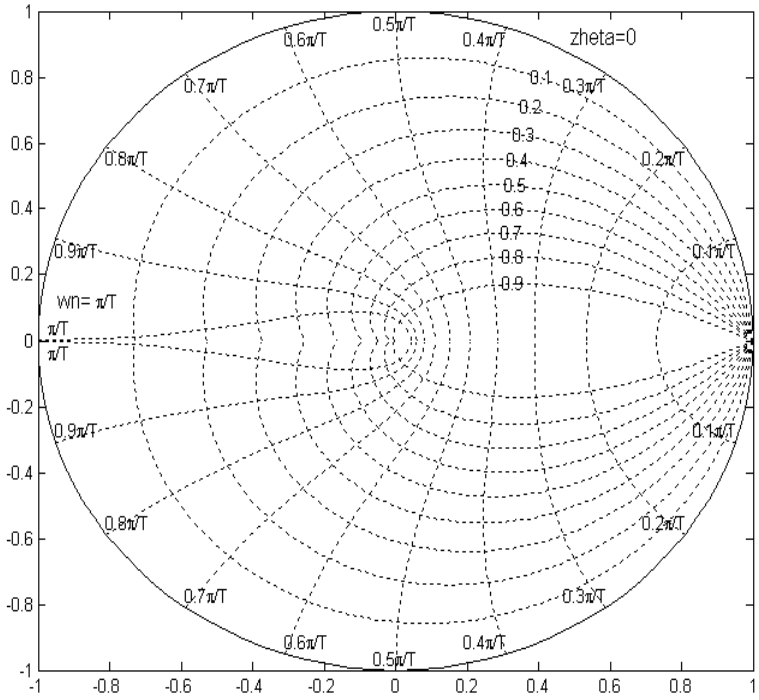

Figura 2.16: Contornos de ξ y ω_n constantes para un sistema de control discretizado con intervalo de muestreo T constante.

En resumen, hemos visto que podemos hacer perfectamente el cálculo de las constantes del sistema en tiempo continuo o discreto, ya que tenemos medios para pasar de uno a otro. Este resultado nos sirve para justificar que en el apartado 2.9 hayamos hecho los cálculos en tiempo continuo (notar que la función de transferencia depende de *'s'*) siendo sin embargo nuestro sistema basado en computador y por tanto discreto.

2.11. Limitaciones y mejoras al algoritmo de control

El sistema compuesto por la bola y el plano aparece con frecuencia como ejemplo típico de sistema dinámico no lineal. Mientras que el sistema ideal bola-plano es en sí no lineal, su implementación física en el laboratorio acarrea no linealidades adicionales, entre las que podemos citar la existencia de la zona muerta y del contragolpe, introducidos por los motores y por la configuración de la caja de los engranajes, la detección de la posición discretizada de la bola, y la existencia de una superficie desigual para rodar.

Estas irregularidades adicionales hacen que el modelo que teníamos del sistema se aleje del modelo matemático del sistema real. Por lo tanto, el comportamiento del

sistema bola-plano más controlador que hemos construido físicamente no va a ser igual al que habíamos simulado.

Para que nuestro sistema vuelva a tener un comportamiento parecido al de la simulación, es decir, una convergencia rápida y poco sobredisparo, es necesario que hagamos algún cambio en nuestro controlador, o que introduzcamos algún elemento más en el bucle de control.

2.11.1. Filtro

Implementamos un filtro en el bucle de realimentación del algoritmo, con la idea de eliminar en lo posible el ruido de la señal de posición de la bola, calculada por el procesado de la imagen.

Sabemos que la señal va a tener fundamentalmente una componente muy fuerte de ruido debido a la cuantificación, ya que la cámara hace capturas con muy poca resolución (320x240).

La baja resolución de la cámara hace que un pixel tenga un tamaño aproximado de:

$$\frac{30\,cm}{204\,pixels} = 1.4706\,mm/pixel. \tag{2.41}$$

El error de cuantificación máximo es igual a la mitad de un escalón [OPP78], donde cada escalón de cuantificación es igual al tamaño de un pixel (2.41). Por lo tanto, si el procesado encuentra el centro exacto de gravedad de la bola, como máximo vamos a cometer errores de 0.75 mm, únicamente debidos al proceso de discretización.

Esto supone que si la posición de la bola es la misma en dos instantes consecutivos (mismo pixel), supondremos que la bola ha estado quieta. Sin embargo, ha podido moverse con una velocidad próxima a:

$$\frac{1.4706\,mm}{40\,ms} = 36.765\,mm/s \simeq 3.5\,cm/s \tag{2.42}$$

Evidentemente, pasa lo mismo si la bola realmente está quieta, y la posición calculada oscila entre dos pixels (por estar cerca del umbral, o porque el procesado es muy dependiente de la iluminación, brillos, etc.). Supondríamos que la bola lleva una velocidad de 3.5 cm/s, cuando en realidad está quieta. La situación se agravaría si en el siguiente instante, aún permaneciendo estática, la velocidad calculada tuviese signo contrario.

El algoritmo de control es dependiente del error de posición y de la velocidad de la bola. Para intentar paliar el efecto que produce el ruido de cuantificación en el algoritmo de control, introducimos un filtro que suavice los cambios bruscos de velocidad, ya que sabemos la aceleración máxima de la bola (conocemos la consigna de los motores).

Funcionamiento del filtro:

Datos de entrada: posición actual y consigna de los motores en la iteración anterior. El filtro calcula la aceleración máxima a partir de la consigna:

$$\text{(aproximadamente} \quad a_{MAX} = 0.933 \cdot \theta \ \text{)}.$$

Si la consigna es pequeña, el ángulo del plano está próximo a cero y la aceleración será muy pequeña. Este valor lo vamos a usar para acotar la aceleración que calculemos mediante otro método más fiable (es posible que no se haya alcanzado el valor de la consigna en el periodo actual), así que no nos interesa que sea demasiado pequeño, de forma que si es inferior al (aproximadamente) 10% del valor angular máximo, al que llamamos consigna mínima, desechamos este dato y nos quedamos con la consigna mínima.

Tenemos guardadas la posición y velocidad calculadas en la iteración anterior, y las vamos a usar para calcular la velocidad y la aceleración actual:

$$\begin{aligned} Velocidad_{MEDIA} &= \frac{Posicion_{ACTUAL} - Posicion_{ANTERIOR}}{T = 40\,ms} \\ Aceleracion_{MEDIA} &= \frac{Velocidad_{MEDIA} - Velocidad_{ANTERIOR}}{T = 40\,ms} \end{aligned} \tag{2.43}$$

La aceleración en realidad es una estima bastante por debajo de la media. Además, si la aceleración es mayor que la aceleración máxima, se satura.

Filtramos la velocidad y la posición mediante el filtro paso bajo:

$$Velocidad_{FILTRADA} = Velocidad_{ANTERIOR} + Aceleracion_{ACTUAL} \cdot T$$

$$Posicion_{FILTRADA} = Posicion_{ANTERIOR} + \frac{(Velocidad_{ANTERIOR} + Velocidad_{FILTRADA})}{2} \cdot T$$

Se recalcula la velocidad en caso de que hayamos saturado la aceleración. En otro caso, permanece igual a la velocidad media.

Como vemos, obtenemos la nueva posición suponiendo que la bola se ha movido en el periodo anterior a una velocidad igual al promedio entre la velocidad anterior y la actual.

Sin embargo, la velocidad actual en realidad es una estima bastante por debajo de la velocidad instantánea, ya que se ha obtenido considerando velocidades y aceleraciones medias como si en realidad fuesen las velocidades y aceleraciones instantáneas al final del periodo, aparte de que es posible que la aceleración esté saturada. Por lo tanto, cabe esperar que el filtro disminuya considerablemente los efectos negativos que tiene el ruido de cuantificación sobre el controlador.

La siguiente figura muestra los resultados de una simulación del sistema completo:

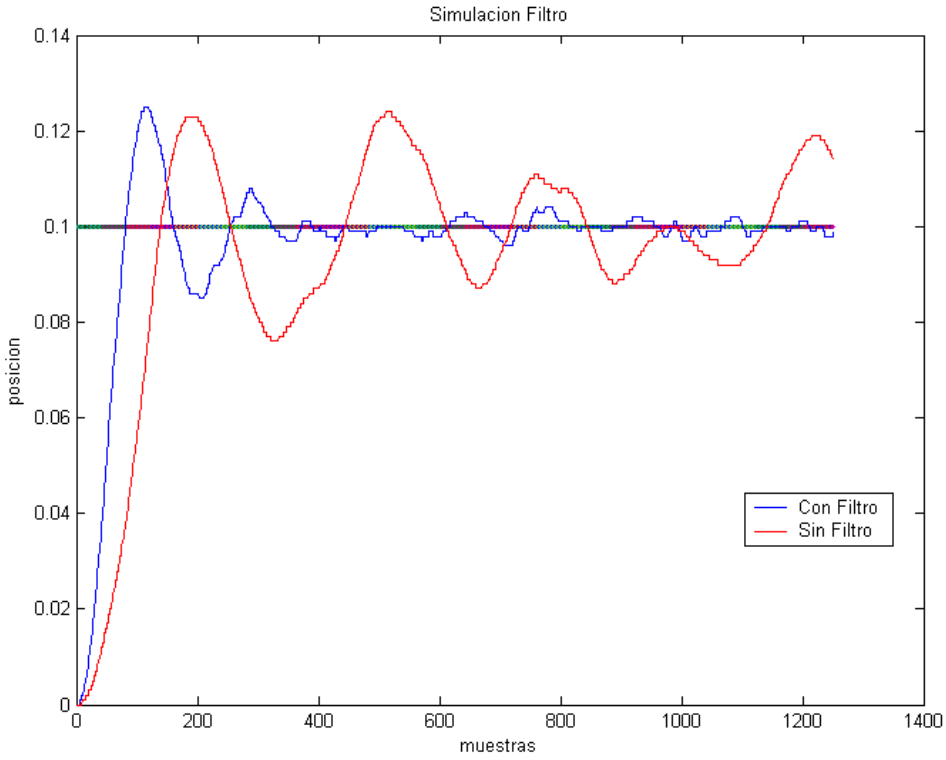

Figura 2.17. Resultado de la simulación: Mejora introducida por el filtro.

Para empezar, podemos observar que la convergencia es más rápida con el filtro que sin él. La posición objetivo es la línea recta a 0.1m.

Sin embargo, donde de verdad notamos los efectos del filtro, en cuanto a suavizado de ruido de cuantificación se refiere, es analizando la varianza de la posición de la

bola después de la convergencia:

Varianza en la posición de la Bola	
Con Filtro	*0.0087750 mm*
Sin Filtro	*0.14779 mm*

Tabla 2.2. Varianza en la posición de la bola, con y sin filtro.

La varianza de la posición de la bola sin filtro, una vez el sistema converge en torno a la posición deseada, es casi veinte veces mayor que con él.

2.11.2. Otras mejoras

Principalmente, se ha implementado un procedimiento que divide la superficie del plano en regiones concéntricas, con centro en la posición deseada. La idea es que cuando la bola vaya acercándose al centro, va cambiando de región. En cada una de estas regiones comparamos la velocidad de la bola con un umbral (más pequeño cuanto más cerca de la posición deseada). En caso de que la velocidad de la bola sea menor que el umbral, para un eje dado, reducimos el ángulo de inclinación de la plataforma en ese eje.

La condición de pertenencia a una región depende tanto de la posición de la bola como de su velocidad, tal como se muestra en la siguiente tabla:

Región	***0***	***1***	***2***	***3***	***4***	***5***	***6***	***7***
Radio = [mm]	> 147	< 147 > 74	< 74 > 37	< 37 > 18	< 18 > 12	< 12 > 7	< 7 > 5	< 5
Vel [m/s]	~	< 3.67	< 1.84	< 0.92	< 0.5	< 0.31	< 0.18	< 0.12
θ	θ	θ	$\theta/2$	$\theta/4$	$\theta/8$	$\theta/12$	$\theta/20$	0
Ki	4	0	0	0	0	2	2.5	0

Tabla 2.3. Mejora al algoritmo de control: Regiones.

De esta forma conseguimos que la bola se pare en un entorno próximo de su posición deseada.

Cuando la bola está muy próxima al centro se activa el control integrativo, de forma que la integral del error de posición va aumentando si existe error. La consigna de los motores va aumentando con la integral, de forma que cuando sea lo suficientemente grande, la bola rodará hacia el centro para anular el error.

En la región más próxima al destino se desactiva el control integrativo, ya que la bola va a estar casi quieta y el error, por pequeño que sea, puede desbordar el tamaño de la variable en que se guarda (siempre va a sumar con el mismo signo). Hacemos la integral igual a cero para que comience a sumar desde cero en caso de que se salga del anillo central y tengamos que volver a activar el control integrativo.

En las regiones intermedias también desactivamos el control integrativo, para que funcione bien el controlador PD.

Cuando la bola no ha comenzado aún a rodar, existe una zona muerta debida al rozamiento estático, o resistencia debida a la compresibilidad de las superficies de contacto y a la deformación entre el cuerpo y el apoyo. La zona muerta implica que hay un ángulo mínimo para que la bola comience a rodar, que podemos salvar gracias al control integrativo.

Experimentalmente, hemos observado que la zona más alejada del centro (Región 0) es en la que con mayor probabilidad va a comenzar el movimiento. Es por esto que en esta zona incluimos el control integrativo, para que en caso de haber obstáculos difíciles de salvar (además del rozamiento estático ya mencionado), el reiterado error de posición haga que aumente la integral, y con ella la consigna que vamos a enviar a los motores.

SERVOMOTORES

3.1. Funcionamiento de los servomotores

Hasta ahora hemos visto que tenemos que mover la plataforma de determinada manera, y en determinados instantes, para que la bola siga la trayectoria deseada.

Hemos supuesto que somos capaces de controlar el ángulo de inclinación del plano, y no hemos tenido en cuenta ningún tipo de limitación. Obviamente esto no va a ser cierto, ya que la velocidad de giro de la plataforma, así como el ángulo máximo o la precisión van a estar supeditados por los motores que usemos.

La fuerza que moverá nuestra plataforma la proporcionan dos servomotores iguales a los que se emplean en modelismo y radio control para hacer girar la dirección de los coches y para mover el timón de los aviones. Gracias al modelismo podemos disponer de estos motores a un precio relativamente asequible.

Los servos de radiocontrol son un tipo especial de motor que se caracterizan por su capacidad para posicionarse rápidamente en cualquier posición dentro de su rango de operación [IME][MID][AUT]. Para ello, el servo espera un tren de pulsos que se corresponden con el movimiento a realizar. Están generalmente formados por un amplificador, un motor, la reducción de engranaje y la realimentación, todo en un misma caja de pequeñas dimensiones. El resultado es un servo de posición con un margen de operación de 180° aproximadamente.

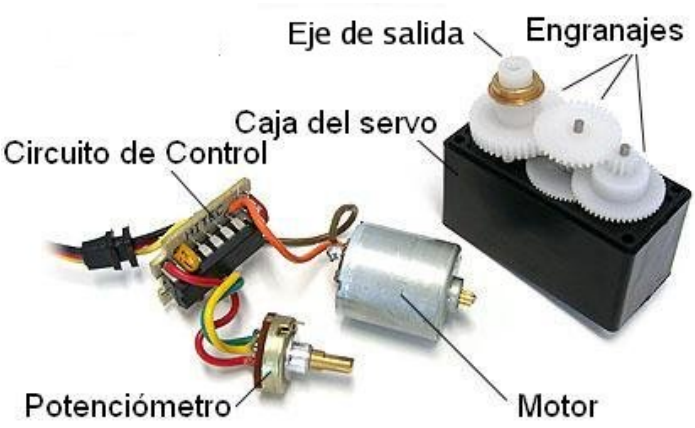

Figura 3.1: Detalle de la Estructura interna de un Servo.

El motor eléctrico en miniatura genera la magnitud que se ha de controlar: el giro y posicionamiento del eje del motor. A su vez, el movimiento de rotación angular del motor modifica la posición de un potenciómetro interno que controla la anchura de los pulsos de un monoestable también integrado en el servo.

El eje del motor puede ser girado hasta una posición angular especifica mediante la señal de control. Mientras se mantenga esta señal de control, la posición del eje del motor se mantendrá. Si la señal de control cambia, también cambia la posición angular del eje.

Disponen de tres conexiones eléctricas: Vcc (roja), GND(negra) y entrada de control (amarilla). Estos colores de identificación y el orden de las conexiones dependen del fabricante del servo. Es importante identificar las conexiones ya que un voltaje de polaridad contraria podría dañar el servo.

En la siguiente tabla podemos ver las características de los "servos" en concreto que hemos usado en la fabricación del sistema:

Características Técnicas de los Servos: "Jamara Car-Power Servo"	
Alimentación	4.8 V – 6 V
Par de Salida	7.6 Kgcm (alimentado a 6 V)
Velocidad de Operación	0.20 seg (alimentado a 6 V)
Dimensiones	20 x 40.5 x 40.5 mm
Peso aproximado	65 g
Engranaje	De Metal
Rodamientos	De Bolitas x 2

Tabla 3.1: características técnicas de los servos "Car-Power Servo", de Jamara.

Los engranajes de metal hacen más duradera la vida de estos servos [JAM], frente a otros servos con los engranajes de de plástico. Los rodamientos de bolas hacen que la fricción sea mínima, haciendo que el funcionamiento sea mucho más suave, además de que mejoran el centrado y ajuste a través del tren de engranajes.

El control de un servo se limita a indicar en que posición se debe situar. La velocidad del motor, así como la dirección de movimiento de los servos se controla mediante una serie de pulsos [IME][MID][AUT]. La magnitud del giro del eje del servo es proporcional a la anchura del pulso que llega por la línea de control. Esta

señal digital se genera aproximadamente cada 20ms, aunque se pueden emplear valores entre 10ms y 30ms. Si el intervalo entre pulso y pulso es inferior al mínimo, puede interferir con la temporización interna del servo, causando un zumbido, y la vibración del brazo de salida. Si es mayor que el máximo, entonces el servo pasará a estado "dormido" entre pulsos. Esto provoca que se mueva con intervalos pequeños y que no esté controlando la posición constantemente. Es importante destacar que para que un servo se mantenga en la misma posición durante un cierto tiempo, es necesario enviarle continuamente el pulso correspondiente. De este modo, si existe alguna fuerza que le obligue a abandonar esta posición, intentará resistirse. Si se deja de enviar pulsos (o el intervalo entre pulsos es mayor del máximo) entonces el servo perderá fuerza y dejará de intentar mantener su posición, de modo que cualquier fuerza externa podría desplazarlo.

Cada servo tiene sus márgenes de operación, que se corresponden con el ancho del pulso máximo y mínimo que el servo entiende. Los valores más generales corresponden con valores entre 1ms y 2ms. Estos valores suelen ser los recomendados, sin embargo, es posible emplear pulsos menores de 1ms o mayores de 2ms, pudiéndose conseguir ángulos mayores de 180º. Aunque la relación entre la anchura del pulso y la posición del eje no está estandarizada, lo normal es que trenes de pulsos de 1.5ms lleven al centro de su rango al eje del servo, mientras que pulsos de 1ms y 2ms corresponden a las posiciones extremas. Esta técnica se conoce como PWM, o modulación por anchura de pulsos (acrónimo de *Pulse Width Modulation*, en inglés).

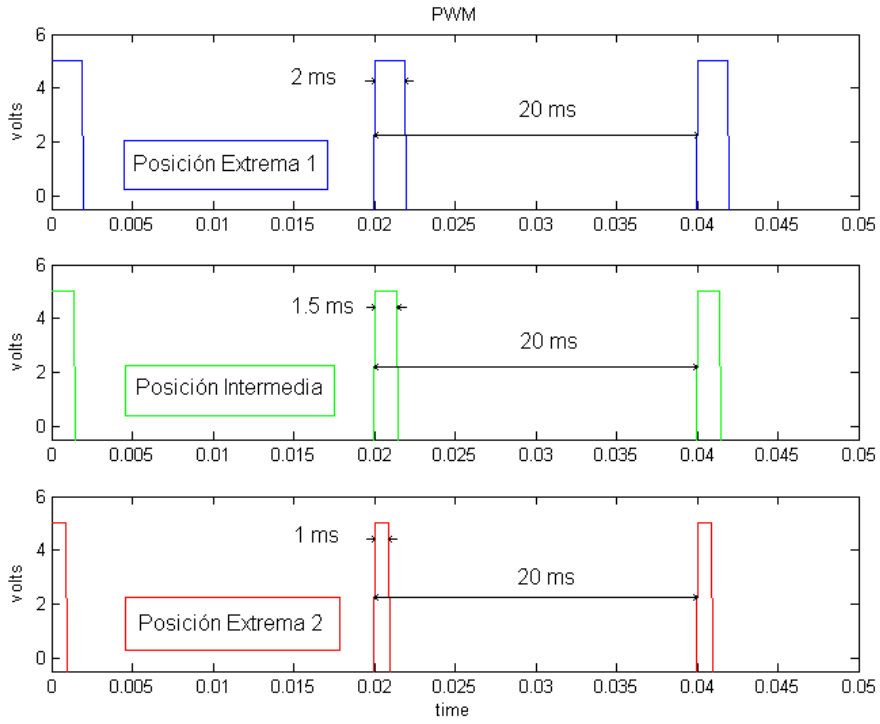

Figura 3.2: Modulación por Anchura de Pulsos.

El servomotor trabaja comparando la anchura del pulso de entrada con la anchura del pulso generado por el reloj o *timer* interno, que es controlada por el potenciómetro conectado al eje del servo. La lógica del servo se encarga de determinar la dirección en la que debe girar el motor para minimizar el error entre las anchuras de ambos pulsos, el de entrada y el interno.

Cuando llega el siguiente pulso se vuelve a realizar la comparación, comprobando de forma continua la posición del eje, y realizando las correcciones necesarias en la posición del mismo.

En la siguiente figura (3.3) podemos ver esquemáticamente la realimentación de la posición del motor, comparación con la posición de entrada y corrección del error mediante un filtro PID:

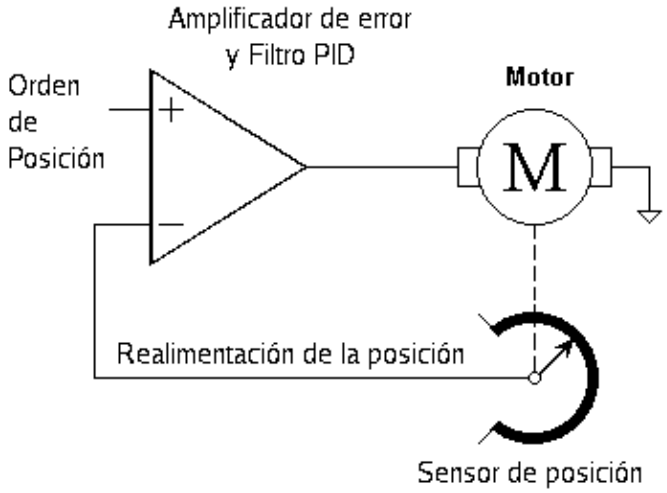

Figura 3.3: Operación de Posicionamiento del Servo.

La precisión al posicionarse depende tanto de la precisión del potenciómetro como de la precisión de la anchura de los pulsos que llegan al motor. La mayoría de los servomotores consiguen una resolución de 0.5 grados.

Cuando la señal de error cae por debajo de un determinado umbral (aproximadamente unos 5 microsegundos), el eje del servo se encuentra en la posición correcta. En ese momento la corriente del motor se apaga. Es decir, cuando el servo está en este rango conocido como zona muerta o banda de guarda (*guard band)*, el servo apaga los drivers del motor.

Si la señal de error no está por debajo del umbral, la electrónica interna seguirá intentando cancelar el minúsculo error, haciendo girar el motor atrás o adelante en un

movimiento llamado "*hunting*". La electrónica interna tiene como objetivo hacer la anchura de los pulsos del monoestable igual a la anchura de los pulsos de entrada.

Debido a que hay una relación fija entre el ángulo de rotación del potenciómetro y la anchura del pulso interno, podemos controlar directamente el ángulo de rotación del servo mediante la anchura de los pulsos aplicados.

3.2. Puerto paralelo

El mecanismo elegido para comunicarnos con los servomotores es el puerto paralelo. La transmisión en paralelo entre un computador y un periférico, se basa en la transmisión de datos simultáneamente por varios canales, generalmente 8 bits. Por esto se necesitan 8 cables para la transmisión de cada byte, mas otros cables adicionales para conocer el estado del dispositivo y realizar el control del mismo. El numero de estos dependerá del protocolo de transmisión utilizado.

Mediante este puerto, podemos escribir simultáneamente en 8 lineas de datos, es decir, 1 Byte, a diferencia del puerto serie, en el que escribimos un sólo bit cada vez. Esta propiedad justifica que hayamos elegido el puerto paralelo para controlar los motores, ya que necesitamos dos líneas independientes de datos, una para cada servomotor. Esto nos va a permitir escribir las consignas de ambos servos simultáneamente, cosa que resultaría imposible si utilizásemos por ejemplo el puerto serie. El puerto paralelo típico de un PC utiliza un conector hembra de tipo D de 25 patillas (DB-25 S). Éste es el caso más común, aunque existen tres tipos de conectores definidos por el estándar IEEE 1284: tipos A, B, C.

A continuación, un dibujo del conector 1284 tipo A, también llamado DB25:

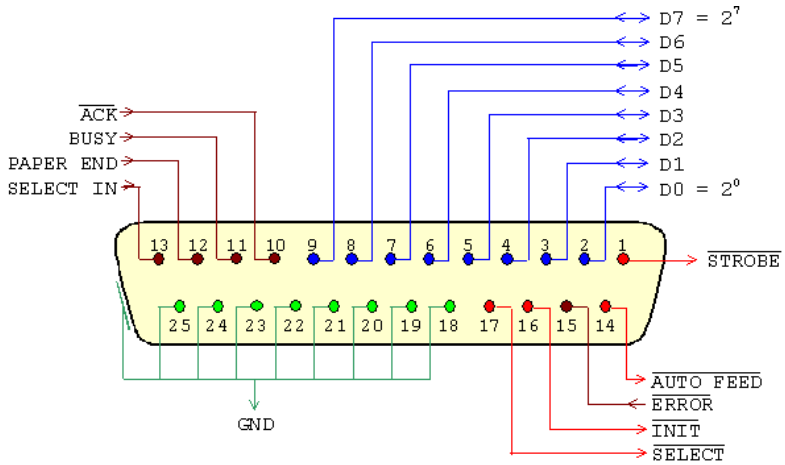

Figura 3.4: conector DB25.

Patilla	***E/S***	***Polaridad Activa***	***Descripción***
1	Salida	*0*	Strobe
2 .. 9	Salida	--	Lineas de Datos (bit 0/patilla 2, bit 7/patilla 9)
10	Entrada	*0*	Línea Acknowledge (activa cuando el sistema remoto toma datos)
11	Entrada	*0*	Línea Busy (si está activa, el sistema remoto no acepta datos)
12	Entrada	*1*	Línea Falta Papel (si está activa, falta papel en la impresora)
13	Entrada	*1*	Línea Select (si está activa, la impresora se ha seleccionado)
14	Salida	*0*	Línea Autofeed (si está activa, la impresora inserta una nueva línea por cada retorno de carro)
15	Entrada	*0*	Línea Error (si está activa, hay un error en la impresora)
16	Salida	*0*	Línea Init (si se mantiene activa al menos durante 50 microsegundos, esta señal inicializa la impresora)
17	Salida	*0*	Línea Select Input (cuando está inactiva, obliga a la impresora a salir de línea)
18 .. 25	--	--	Tierra Eléctrica

Tabla 3.2: Configuración del Puerto Paralelo Estándar

En la tabla 3.2 se muestra la función de cada patilla de este conector.

Sin embargo, de todas las líneas del puerto paralelo sólo vamos a utilizar las de datos y las tierras eléctricas, que aparecen sombreadas en la tabla. Esto en así porque para escribir en cada motor necesitamos una línea para la alimentación (típicamente 5V), otra para la tierra (como tenemos la de la fuente de alimentación y la del puerto paralelo, las vamos a unir para que los motores tengan una sola referencia de tierra) y otra más para la señal de control (línea de datos del puerto paralelo).

Hay tres direcciones de E/S asociadas con un puerto paralelo. Éstas direcciones pertenecen al **registro de datos**, el **registro de estado** y el **registro de control**. El *registro de datos* es un puerto de lectura-escritura de ocho bits. Leer el registro de datos (en la modalidad unidireccional) retorna el último valor escrito en el registro de datos. Los registros de control y estado proveen la interfaz a las otras líneas de E/S. La distribución de las diferentes señales para cada uno de los tres registros de un puerto paralelo esta dada en la siguiente tabla:

Dirección	***Nombre***	***Lectura/Escritura***	***Bit Nº***	***Propiedades***
Base + 0	Puerto de Datos	ESCRITURA	Bit 7	Dato 7
			Bit 6	Dato 6
			Bit 5	Dato 5
			Bit 4	Dato 4
			Bit 3	Dato 3
			Bit 2	Dato 2
			Bit 1	Dato 1
			Bit 0	Dato 0
Base + 1	Puerto de Estado	SÓLO LECTURA	Bit 7	Busy
			Bit 6	Acknowledge
			Bit 5	Falta Papel
			Bit 4	Select In
			Bit 3	Error
			Bit 2	IRQ (Not)
			Bit 1	Reservado
			Bit 0	Reservado

Dirección	*Nombre*	*Lectura/Escritura*	*Bit Nº*	*Propiedades*
Base + 2	Puerto de Control	LECTURA/ESCRITURA	Bit 7	No usado
			Bit 6	No usado
			Bit 5	Permite puerto bidireccional
			Bit 4	Permite IRQ a través de la línea acknowledge
			Bit 3	Selecciona impresora
			Bit 2	Inicializa impresora
			Bit 1	Nueva línea automática
			Bit 0	Strobe

Tabla 3.3: Registros del Puerto Paralelo

Un computador soporta hasta tres puertos paralelos separados. Por lo tanto, puede haber hasta tres juegos de registros en un sistema en un momento dado. Existen tres **direcciones base** para el puerto paralelo asociadas con tres posibles puertos paralelo: 0x3BCh, 0x378h y 0x278h, nos referimos a éstas como las direcciones base para el puerto **LPT1**, **LPT2** y **LPT3**, respectivamente. El registro de datos se localiza siempre en la dirección base de un puerto paralelo (Base + 0), el registro de estado aparece en la dirección Base + 1, y el registro de control aparece en la dirección Base + 2.

Por ejemplo, para un puerto LPT2 localizado en 0x378h, ésta es la dirección del registro de datos, al registro de estado le corresponde la dirección 0x379h y su respectivo registro de control está en la dirección 0x37Ah.

Eléctricamente, el puerto paralelo entrega señales TTL y como tal, teóricamente, se le puede conectar cualquier dispositivo que cumpla con los niveles de voltaje específicos de la lógica TTL. Sin embargo, el hardware del puerto paralelo está muy limitado en cuanto a su capacidad de manejo de corriente, por ésta razón se debe ser muy cuidadoso con el manejo de las señales del puerto: un cortocircuito puede dañar permanentemente la placa base del PC.

Para poder realizar operaciones de Entrada/Salida, como por ejemplo escribir en el puerto serie o paralelo, disponemos en MaRTE de la librería ***IO_Interface.ads***. Esta librería nos permite definir la dirección del puerto de E/S, entre 0x0h y 0xFFFFh, que vamos a usar, además de ofrecer procedimientos para leer/escribir datos de tipo *byte*, *word* (2 bytes), y *long word* (4 bytes), tanto con retardo como sin él. El retardo es bueno porque permite adaptar la velocidad del hardware moderno a la del hardware más antiguo, permitiendo así la comunicación entre los diferentes dispositivos.

El procedimiento en concreto que usamos para escribir la consigna en los servos es Outb_P, que escribe un byte en el puerto paralelo con retardo [MaRTE].

3.3. Escritura mediante pulsos: PWM (*Pulse Width Modulation*)

Este sistema consiste en generar una onda cuadrada en la que se varía el tiempo que el pulso está a nivel alto, manteniendo el mismo período (normalmente), con el objetivo de modificar la posición del servo según se desee. En nuestro caso, se ha elegido que el periodo entre pulsos sea de 20ms, con una anchura variable entre 1ms y 2ms, que se corresponden aproximadamente con las posiciones extremas de los motores. Si bien, estos valores no son exactamente los que aparecen en los programas:

> Aunque nuestros motores son idénticos, usamos anchuras de pulso distintas para llevar cada eje de la plataforma a una posición predefinida de su rango (centro, extremos, etc...). Esto se debe a que nuestros motores no están conectados al mismo punto de la plataforma, sino que la palanca de uno está soldada al marco más externo de la misma, mientras que la del otro servo está un poco más hacia el interior, para conseguir así que la inclinación de la plataforma sea independiente en ambos ejes.

Para escribir en los motores necesitamos una tarea periódica que se ejecute cada 20ms (periodo de la onda cuadrada que enviamos a los servos). Esta tarea utiliza la función de MaRTE *Outb_T*, de la librería *IO_Interface*, para escribir un byte en el puerto paralelo. Del byte de datos enviado, los motores están conectados a los bits de menor peso. Esto es más fácil verlo en las siguiente tablas:

Motor	***Bit Nº***
X (LSB)	*0*
Y	*1*

Valores en los Motores	*Nº escrito en el Puerto Paralelo*
Y = **0**, X = **0**	*0*
Y = **0**, X = **1**	*1*
Y = **1**, X = **0**	*2*
X = **1**, Y = **1**	*3*

Tabla 3.4: Escritura de los motores.

Así, escribiendo estos cuatro números en el momento apropiado, obtenemos las dos señales digitales moduladas en anchura de pulso que necesitamos para mover simultáneamente ambos motores.

A continuación se muestra cómo hemos implementado la escritura de los motores en nuestros programas:

- Lo primero que hacemos es inicializar un temporizador, que nos servirá para controlar el periodo de 20ms.

- Como escribimos a la vez en ambos servos, lo que tenemos que hacer al comienzo del periodo es poner ambos pulsos a ***'1'***, para lo cual tenemos que escribir un ***'3'*** en el puerto. Para que la duración de los pulsos sea lo más fiable posible, necesitamos usar otro temporizador, que hemos inicializado justo antes de escribir el comienzo de los pulsos.

- Luego debemos bajar a ***'0'*** el pulso de menor duración. Para ello necesitamos conocer la anchura de los pulsos en 'X' y en 'Y'. Los comparamos y esperamos hasta el instante correspondiente al inicio de los pulsos, más el pulso de **menor** duración. Escribimos un ***'1'*** o un ***'2'***, si el pulso menor es el 'Y' o el 'X', respectivamente, mientras que si son iguales debemos escribir un ***'0'***.

- Si los pulsos eran iguales, nos ahorramos este paso: esperamos hasta el instante correspondiente al inicio de los pulsos, más el pulso de **mayor** duración, y escribimos un ***'0'***.

- Ahora esperamos hasta el inicio del periodo (primer temporizador) más 20ms, es decir, hasta el comienzo del siguiente periodo.

En el siguiente esquema se puede ver de forma gráfica cómo funciona la escritura de los pulsos.

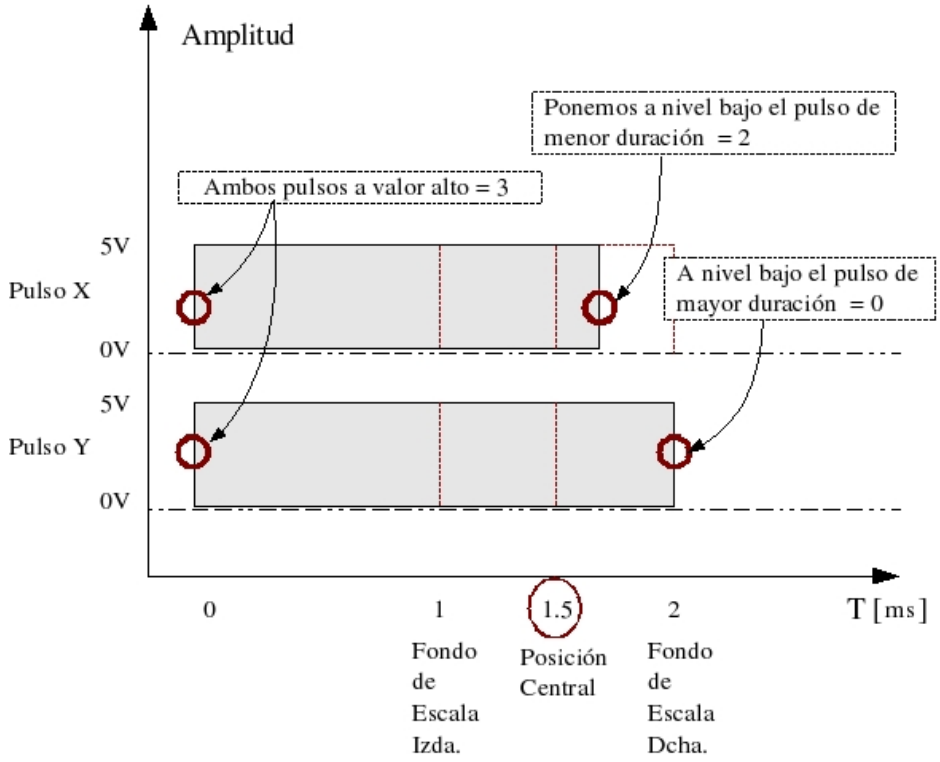

Figura 3.5: Escritura de dos pulsos que comienzan simultáneamente.

Para poder garantizar que el periodo va a ser muy próximo a 20ms, de todas las tareas que se estén ejecutando, la encargada de escribir la consigna de los servos tiene que tener la prioridad más alta. De esta forma, no será expulsada mientras está escribiendo o esperando la duración un pulso para poner la línea correspondiente a nivel bajo. Si pudiese ser interrumpida, generaríamos pulsos de anchura incorrecta, lo que se traduciría en una posición del servo también errónea. Para evitar estas situaciones de error debemos dar a esta tarea la prioridad más alta posible.

Es preciso observar que para conseguir un error máximo de un 1% del fondo de escala en posición, el error máximo de los instantes de subida y bajada del pulso está aproximadamente en 10μs.

Los pulsos que obtendremos tienen que tener la siguiente forma:

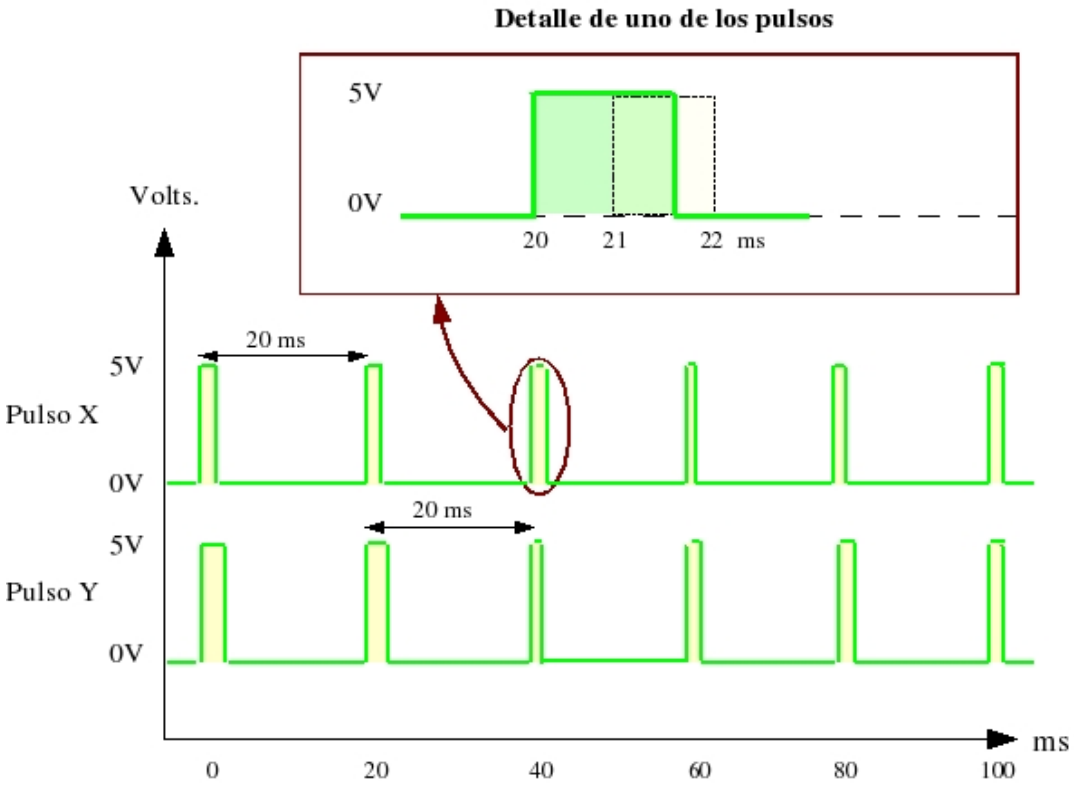

Figura 3.6: Señal modulada por anchura de pulso.

Ahora bien, que escribamos los pulsos de los servos correctamente no depende únicamente del retardo de la lógica del puerto paralelo. En la parte concerniente al retraso que introduce el puerto, no nos importa tanto que la magnitud de este retardo sea grande, como que esté acotado o, mejor aún, que sea constante. En el supuesto de que este retardo sea variable, la precisión de los pulsos no sería muy buena, ya que este retardo podría ser distinto para la señal que escribe el flanco de subida del pulso y la que escribe el flanco de bajada. Además, no podríamos mantener la posición de los servos, ya que cada pulso sería distinto al anterior, por lo que se moverían adelante y atrás continuamente en lugar de estar siempre en torno a la misma posición angular (aunque para suavizar este efecto tenemos un filtro PID en el servo, que compara el valor del error con un umbral, así que sólo tendríamos este problema si el error al escribir la anchura del pulso es mayor que el umbral del comparador del servo).

Afortunadamente, vamos a tener un retardo constante y bastante pequeño. Sabemos que es así porque el puerto paralelo sólo tiene componentes de lógica combinacional, es decir, sólo hardware. El retraso introducido por la lógica del puerto, al tratarse de lógica TTL, va a ser del orden de las decenas de nanosegundos a lo sumo, que podemos considerar despreciable, frente a los errores temporales en el software.

En definitiva, en lo que concierne al puerto paralelo vamos a poder controlar bien la posición del plano. Para comprobar que la señal digital realmente cumple los requerimientos temporales de frecuencia y duración de los pulsos, hemos utilizado un osciloscopio y unos programas de prueba que se describen a continuación, y que también nos han servido para calibrar el rango de operación de los servos.

3.4. Calibrado

El calibrado de los motores es necesario para tener la certeza de que cuando los movamos hacia una determinada posición angular vamos a llegar a esa posición o a un entorno muy próximo de la misma. Nos interesan las anchuras de pulso necesarias para llevar los servos a sus posiciones extremas y central (sobre todo). El resto de posiciones las obtenemos fácilmente ya que la anchura del pulso y la posición angular siguen una relación lineal. Usaremos la posición central para comprobar que se cumple esta relación y para obtener una ajuste más fino, ya que estamos especialmente interesados en que los motores funcionen bien en un pequeño rango en torno a la posición horizontal del plano.

Para realizar el calibrado hemos creado un programa de prueba que, básicamente, es igual al programa que se encarga de escribir en los motores. Consta de una tarea que se ejecuta cada 20ms, respetando el periodo de escritura de los servos, además de otra tarea que pide por teclado la posición de los motores, de menor prioridad que la anterior, para que sea expulsada en caso de que se esté ejecutando cuando tengamos que escribir un pulso. Así garantizamos que el periodo de los pulsos va a ser 20ms.

Introducimos un número en representación de coma flotante entre -10.0 y 10.0, que se corresponden aproximadamente con las posiciones angulares máxima y mínima de los motores. Para ello partimos de las anchuras teóricas de 1ms y 2ms.

Ajustamos hasta conseguir llevar el servo a la posición que queramos. El programa muestra por pantalla la anchura del pulso correspondiente.

Para comprobar que realmente estamos escribiendo pulsos cada 20ms y que tienen la duración apropiada, conectamos un osciloscopio a las líneas de datos 0 y 1, y medimos estos parámetros. Hemos obtenido los siguientes resultados:

Medidas Osciloscopio								
Periodo		**Método medida**	***Duración de Los Pulsos***					
			Mínima		**Media**		**Máxima**	
X	20ms	Duty-Cycle	~	~	7.1%	1420μs	11.1%	2220μs
		Cursores	714μs		1448μs		2210μs	
Y	20ms	Duty-Cycle	~	~	6.6%	1320μs	10.1%	2020μs
		Cursores	706μs		1380μs		2060μs	

Tabla 3.5: Calibrado de los servos con el osciloscopio.

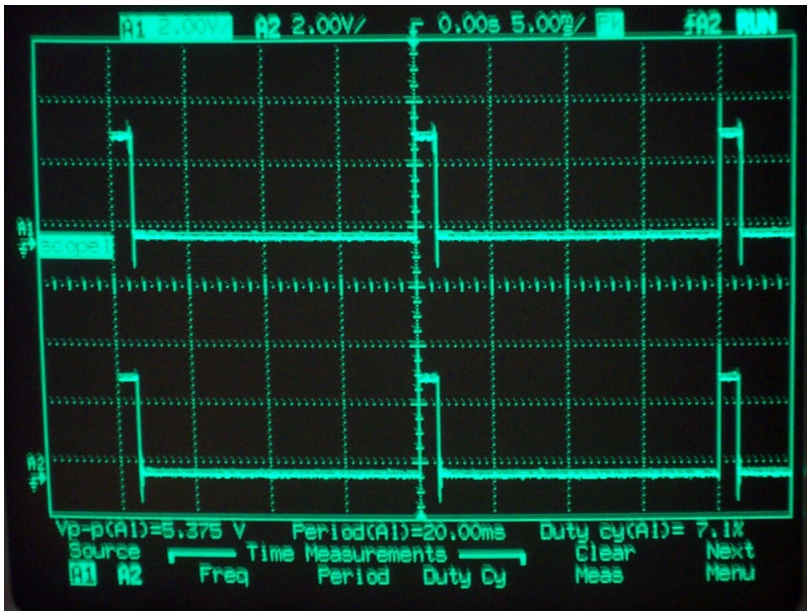

Figura 3.7: Foto osciloscopio de los pulsos de duración intermedia.

La figura anterior es una captura del osciloscopio en la que aparecen los pulsos 'x' e 'y' con duración media (aprox. 1.5ms), para que se la plataforma se posicione en posición horizontal.

En la tabla anterior, lo primero que hay que destacar es la medida del periodo de los pulsos, que es exactamente de 20ms (y es igual tanto para la señal 'x' como para

la 'y', para las posibles duraciones de los pulsos). Sin embargo, este resultado no garantiza que el método utilizado sea exacto, ya que el resultado se muestra con una resolución de 10μs.

Para la medida de la duración de los pulsos hemos usado dos métodos:

- Para medir el tiempo que el pulso está a nivel alto mediante el duty-cycle, necesitamos ver al menos dos pulsos (para que el osciloscopio calcule el periodo).
- Si medimos utilizando cursores, podemos ampliar la forma de onda para tener más exactitud.

La segunda medida es más precisa (como veremos luego, se aproxima más al valor esperado).

De esta forma sabemos el rango de anchuras de pulso en el que queremos que trabajen los motores. Además de eso, las medidas realizadas con el osciloscopio nos permiten saber de forma aproximada la magnitud de los errores que estamos cometiendo en la duración de los pulsos. Para ello basta con comparar las anchuras de los pulsos medidas, con las esperadas (establecidas en el programa de calibrado), como se muestra en la siguiente tabla:

Errores en la duración de los pulsos				
		Mínima	**Media**	**Máxima**
X	Medida	714μs	1448μs	2210μs
	Esperada	700μs	1450μs	2200μs
	Error	**14μs**	**2μs**	**10μs**
Y	Medida	706μs	1380μs	2060μs
	Esperada	700μs	1375μs	2050μs
	Error	**6μs**	**5μs**	**10μs**

Tabla 3.6: Errores en la duración de los pulsos.

Como vemos en la tabla, debido al método utilizado, existe un pequeño retraso no constante (lo que hace pensar que el problema se debe al software), que hace que los pulsos reales que llegan a los motores sean alrededor de 10μs superiores al valor deseado.

Es posible que modifiquemos ligeramente los valores del rango de operación de los motores, para intentar compensar estos errores.

Volviendo a la precisión del método usado para escribir los pulsos, en el apartado anterior vimos que el hardware del puerto paralelo no introduce ningún retraso significativo. Sin embargo, sí que vamos a cometer un error a la hora de escribir los pulsos, que va a afectar tanto al periodo como a la duración de los mismos. Este error se debe al software. Uno de los factores de los que depende, es el tiempo necesario para que se realice el cambio de contexto cuando nuestra tarea entre a ejecutarse, ya que si hay otra tarea en ejecución en ese momento, tiene que ser expulsada.

También podríamos pensar que la resolución del reloj del sistema puede influir de manera significativa. Sin embargo, esto no es así por dos motivos.

- Primero: porque la resolución del reloj del sistema es lo suficientemente alta.
- Segundo: en realidad la resolución del reloj es sólo la precisión con que podemos medir el tiempo. A nosotros nos interesa la resolución del temporizador (timer hardware). Esto es, la precisión con la que se puede programar el temporizador para que avise al sistema de que ha llegado un instante de tiempo determinado (p.e. el final de un delay de una tarea Ada).

En MaRTE, la resolución del reloj es la inversa de la frecuencia del procesador. La resolución del timer depende de la arquitectura que hayamos elegido en la instalación de MaRTE.

En nuestro caso, la resolución del timer es la del *'Programable Interval Timer'* (PIT) del computador.

Resolución del reloj y del timer del sistema	
Velocidad del Procesador	2793253020 Hz
Resolución del Reloj	1/ 2793253020 seg = 35.8 ns (aprox.)
Resolución del Timer	838.1 ns (aprox.)

Tabla 3.7: Resolución del reloj y del timer del sistema.

Por lo tanto, podemos controlar el instante de comienzo de nuestras tareas con una precisión de aproximadamente un microsegundo.

Los retrasos debidos al software los podemos medir, como veremos, al igual que hemos medido el tiempo que tarda cada tarea en ejecutarse. Para ello utilizamos varias variables, con las que anotamos el instante en que se produce determinado evento significativo, como por ejemplo el comienzo de la tarea o del pulso. Tenemos

otra variable con la hora a la que debería producirse el evento. Sabemos esta hora gracias a que siempre usamos instantes relativos al comienzo del periodo. La resta de estos instantes es por tanto el error que cometemos.

Aplicando esta técnica hemos medido los errores que cometemos en los siguientes casos:

- En el instante en que tiene que despertarse la tarea (tiene que tener un periodo de 20ms)
- En el instante en que comienza el pulso (también cada 20ms)
- En el tiempo que tardan en ejecutarse las instrucciones que hay desde el comienzo de la tarea hasta que comienza el pulso (para saber si ese error es constante, ya que influyen en el instante de comienzo del pulso)
- En el tiempo que tardamos en leer cada reloj (para saber si estamos midiendo de forma precisa los errores anteriores)

- Los resultados de este experimento se muestran en la siguiente tabla:

	Máximo	***Mínimo***	***Medio***
Periodo de los Pulsos	*20.02ms*	*20.001ms*	*20.008828ms*
Error en el Instante de Inicio de la Tarea	*- 1μs*	*≈ 0μs*	*-0.39779μs*
Error en el Instante de Inicio del Pulso	*- 1μs*	*≈ 0μs*	*-9.0808μs*
Tiempo de Ejecución de las Primeras Instrucciones	*5μs*	*2μs*	*2.8981μs*
Error Leyendo el Reloj del Sistema (Clock)	*1μs*	*≈ 0μs*	*0.5858μs*

Tabla 3.8 : Medidas Temporales

Cada uno de los valores que contiene la tabla anterior se han obtenido a partir los estadísticos de 100 mediciones consecutivas.

Como se observa en la primera fila de la tabla, podemos considerar que el método de escritura de los pulsos es relativamente bueno, ya que tenemos un error máximo de tan sólo 20μs en un periodo de 20ms, que corresponde a un error del 2%.

Como vemos, el error en el instante de inicio de la tarea y el error en el instante de comienzo de los pulsos son idénticos. Es lógico ya que la escritura de los pulsos está contenida dentro de la tarea, y realizamos una escritura en los motores cada vez que se inicia la tarea, por lo que todo el error que se cometa a la hora de despertar la tarea se va a traducir de forma inmediata en al menos la misma variación temporal en el instante de comienzo de los pulsos.

Medimos también el tiempo que tardan en ejecutarse las primeras instrucciones ya que, en caso de que fuese constante, podríamos usarlo como dato para despertar la tarea siempre un poco antes, con el objetivo de compensar este retraso. Como vemos, este error es de 5μs.

En cuanto al error que cometemos utilizando este método de medida, podemos acotarlo midiendo el máximo tiempo que tardamos en leer el reloj del sistema. Este error es aproximadamente igual a 1μs, por lo que podemos considerar despreciable frente al resto de errores cometidos, ya que son un orden de magnitud superiores.

3.5. Velocidad de los motores

Los motores tardan 0.5 segundos aproximadamente (unos 25 periodos de 20ms) en realizar un movimiento de un extremo al otro de su fondo de escala. Calculamos la velocidad angular de los servos:

$$\begin{aligned} \textit{ÁnguloMáximo} &= \arctan(2\,cm \,/\, 15\,cm) = 0.13255\ rad \\ \textit{VelocidadAngular} &= \textit{ÁnguloMáximo} * 2 \,/\, 0.5\,seg = 0.53021\,rad/seg \\ \textit{ÁnguloPorPeriodo} &= \textit{ÁnguloMáximo} * 2 \,/\, 25\,periodos = 0.010604\,rad \end{aligned} \quad (1)$$

Por lo tanto, es evidente que si el algoritmo de control nos exige una variación angular mayor a 0.0106 radianes ($\approx 0.6^o$), vamos a necesitar más de un periodo para llegar a la inclinación deseada.

Hemos calculado la velocidad de los motores mediante los 25 periodos que tardan en hacer un movimiento de extremo a extremo. Sin embargo, de esta forma no podemos calcular la velocidad de los motores directamente, sino que estamos calculando la velocidad mediante una aproximación.

Podemos acotar la velocidad angular a la que creemos que se puede mover el plano. Una posible cota inferior es suponer que va a tardar 30 periodos en hacer el recorrido de extremo a extremo. Es una magnitud conservadora que nos va a ser útil a la hora de poner los motores en posición horizontal, para que tengan tiempo de sobra para llegar.

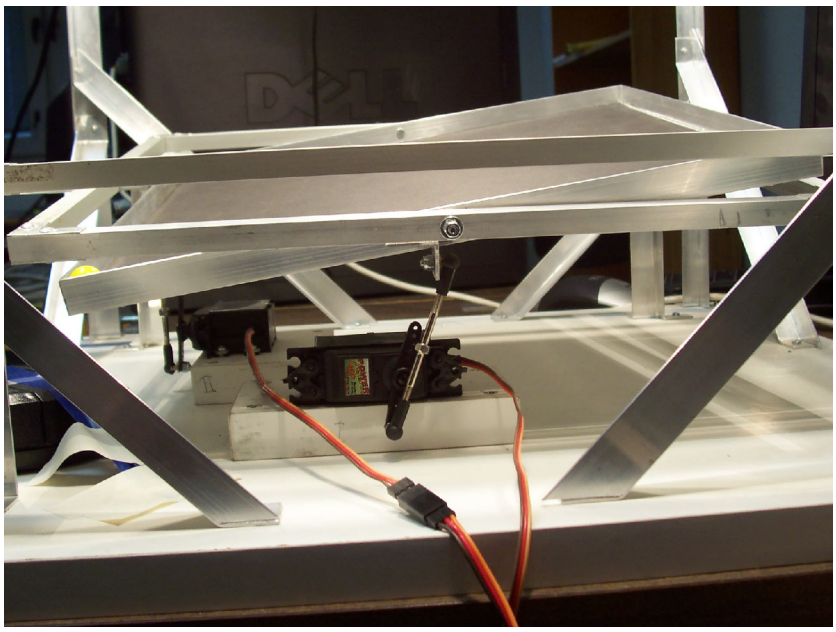

Figura 3.7: Servomotores y plataforma con inclinación máxima.

La velocidad calculada de los motores es una aproximación de su velocidad media, obtenida a partir de su tiempo de respuesta para un movimiento que exige un intervalo temporal grande (de fondo de escala a fondo de escala).

Por lo tanto, para ciertas condiciones de funcionamiento esta aproximación no se va a ajustar a la velocidad instantánea real, principalmente porque los motores no se mueven con velocidad constante cuando arrancan o paran, o cuando se les exija que realicen un cambio de sentido.

CAPTURA DE LAS IMÁGENES

La otra parte del proyecto, realizada por María Campo-Cossío, que se encarga de la captura y procesado de las imágenes, es la que nos tiene que proporcionar la posición de la bola dentro del plano, para lo que debe obtener el centro de masas de la misma, como hemos dicho, mediante la aplicación de ciertas operaciones de procesado y tratamiento de imagen. A continuación vamos a ver la interfaz de este software que vamos a usar, así como las características de funcionamiento:

```
      Obtenido del fichero de especificación: 'procesa_imagen.h'

#include <video_marte.h>

/* ---------------------- *
 * -- Procesa_Imagen.h -- *
 * ---------------------- */

/* Devuelve el instante en el que se
       tomo la foto, la foto y el tamaño:  */

void procesa_imagen (
                 int                 veces,
                 struct xy *         centro_anterior,
                 struct xy *         centro_bola,
                 struct timespec *   instante,
                 int *               salida_error  );

int inicializa_camara ( void );
                                 ...
                                 ...
```

Aunque esta librería tiene otras muchas funciones, aquí sólo vamos a ver las que se usan en la parte del proyecto que estamos tratando:

> Todas las funciones necesarias para realizar el procesado de la imagen, desde la captura de la misma hasta la obtención del resultado final (posición de la bola) están encapsuladas en la función 'procesa_imagen'.

En principio, no nos harían falta más funciones. Sin embargo, en el código anterior aparece otra función: 'inicializa_camara'. Es necesario que usemos esta segunda función porque no se puede inicializar la cámara dentro de 'procesa_imagen'. La inicialización tiene que hacerse desde un módulo que permanezca vivo durante todo el tiempo que esté ejecutándose la aplicación, y

'procesa_imagen' no cumple este requisito ya que es una función. De forma que estas dos funciones son nuestra interfaz con los programas de la parte de imagen.

Para hacer una breve descripción de su funcionamiento, comenzamos por 'inicializa_camara'. Como vemos, no tiene ningún parámetro de entrada, y el entero que devuelve es sólo para indicar si ha habido algún error. La función se limita a inicializar la cámara con el tamaño de las imágenes, formato y modo apropiados, y esperar un tiempo prudencial hasta que podamos usarla.

La función 'procesa_imagen' toma como datos de entrada:

- **`int veces:`** un entero: para saber si tiene que hacer el procesado de la imagen completa (primera vez: `veces = 1`) o sólo de un entorno de la bola (resto).
- **`struct xy * centro_anterior:`** el centro anterior de la bola, para procesar sólo en un entorno de este punto.

Y devuelve:

- **`struct xy *centro_bola:`** el centro actual de la bola, obtenido mediante el procesado de la imagen.
- **`struct timespec *instante:`** el instante de tiempo en que se tomó la imagen.
- **`int *salida_error:`** una salida de error, en la que se almacena el código correspondiente a la función que haya fallado.

Hay que destacar que la función procesa_imagen es bloqueante. Hasta que tiene la imagen y la procesa no se sigue ejecutando el programa que la haya llamado.
En el capítulo 6 se explica cómo usar estas funciones desde el programa principal, y en el subapartado 6.6 se muestran las medidas de los tiempos de ejecución y los periodos de las tareas.
De todas formas, antes de hacer las medidas de estos periodos ya sabemos que van a estar fuertemente correlacionadas con el tiempo de procesado y (sobre todo) adquisición de las imágenes.

TRAYECTORIA DE LA BOLA

Necesitamos conocer en cada instante dónde está la bola, y a dónde queremos que vaya. La posición de la bola la obtenemos de las capturas de la cámara, mediante el procesado de las imágenes. Sin embargo, la posición deseada de la bola en cada instante es dependiente de la trayectoria que tenga que recorrer. Hemos programado un paquete en Ada que resuelve este problema.

Lo primero que hace es crear una trayectoria, que no es más que un conjunto de puntos en el plano. Después de crear la trayectoria, a partir de la posición de la bola y de otros parámetros relevantes (como p.ej. su velocidad), retorna la posición dentro del plano a la que debe dirigirse.

5.1. Creación de la trayectoria

La trayectoria generada se guarda en un array, cuyos elementos son registros de dos campos. Uno de los campos almacena la posición de los puntos de la trayectoria, mientras que en el otro calculamos la velocidad que queremos que tenga la bola en el punto correspondiente al campo Posición:

$$V_{deseada}(i) = \frac{Posición(i+1) - Posición(i)}{T = 40\ ms} \tag{5.1}$$

La magnitud de este vector es precisamente V_{bola}.

El array tiene N elementos. Para recorrerle usamos un índice que es de tipo modular. Así conseguimos que el siguiente elemento después del último, sea de nuevo el primero, lo que es muy útil ya que queremos que la trayectoria que describa la bola sea un bucle cerrado.

Además, como no sabemos el punto de partida de la bola, tampoco podemos saber desde qué punto vamos a empezar a recorrer la trayectoria, de modo que el índice del array aporta información del sentido en que debe recorrerse, pero no del punto inicial.

Hemos decidido que para las pruebas del sistema la trayectoria que va a recorrer la bola va a ser un círculo. El círculo tiene 10 cm de radio y la función que lo crea pide el centro como dato de entrada (por lo que hay que tener cuidado de que la circunferencia no se salga del plano).

Como sabemos la longitud de la circunferencia $(2 \cdot \pi \cdot 10\,cm)$, vamos a tomar puntos de esa circunferencia de forma que la bola pueda recorrer la distancia entre ellos en un periodo de 40ms. Así podemos fijar la velocidad a la que queremos que la bola recorra la circunferencia.

Vamos a fijar esta velocidad para que los puntos estén separados 1 mm, por lo tanto:

$$V_{bola} = \frac{1\ mm}{40\ ms} = \frac{1 \cdot 10^{-3}\ m}{40 \cdot 10^{-3}\ s} = 0.025\ m/s = 25\ mm/s \tag{5.2}$$

En el supuesto de que esta velocidad no sea adecuada, siempre podemos modificarla. Claro que si hacemos esto, el número de puntos de la trayectoria también cambia.

5.2. Algoritmo para el cálculo del siguiente punto de la trayectoria

En este apartado se describe esquemáticamente cómo funciona el programa que calcula el siguiente punto al que debe dirigirse la bola, conociendo la trayectoria que debe recorrer, la posición actual de la bola y su velocidad.

- A -. Caso inicial

Buscamos el punto **λ** de la trayectoria más cercano a la posición actual de la bola. El cálculo de las distancias a todos los puntos de la trayectoria se hace en la forma que refleja el siguiente pseudocódigo:

for i en 1..N Loop

$$Distancia_i = \sqrt{(Pos_{Bola}(x) - PuntoTrayectoria_i(x))^2 + (Pos_{Bola}(y) - PuntoTrayectoria_i(y))^2}$$

end Loop (5.3)

λ se corresponde con el **índice** del array cuya distancia a la bola es mínima:

$$\lambda \in [1, N] \mid distancia_\lambda \leq distancia_i \quad \forall i \in [1, N] \tag{5.4}$$

Hay que notar que λ no es el punto de la trayectoria de menor distancia a la bola, sino su índice.

- B -. Discernir si estamos o no cerca de la trayectoria

Calculamos el tiempo de llegada aproximado a la trayectoria. Definimos un primer tiempo, llamada 'T$_{velocidad}$'.

$$T_{velocidad} = \frac{|V_{Lambda} - V_{Bola}|}{a_{Max}} \tag{5.5}$$

Que se corresponde con el tiempo que tardaríamos en alcanzar la velocidad deseada en el punto λ, suponiendo que la aceleración es máxima (plano inclinado el ángulo máximo). Este tiempo va a predominar cuando la bola esté cerca de la trayectoria: vamos a necesitar variaciones rápidas de velocidad para realizar los correspondientes cambios de dirección.

Definimos otro tiempo, 'T$_{espacio}$', para indicar que no depende de la velocidad, sino de la diferencia de posiciones:

$$T_{espacio} = \frac{|Pos_{Bola} - Pos_{Lambda}|}{V_{Lambda}} \tag{5.6}$$

Es decir, el tiempo que tardaríamos en movernos hasta la posición λ, suponiendo que la bola ya se desplaza a la velocidad deseada. Va a predominar sombre 'T$_{velocidad}$' cuando estemos lejos de la trayectoria.

Calculamos el tiempo que tarda en llegar a la trayectoria como:

$$T_{Llegada} = Max\left(T_{velocidad}, T_{espacio} \right) \tag{5.7}$$

Ahora decidimos si estamos cerca o lejos de la trayectoria:

$$\begin{aligned} &si \;\; T_{Llegada} < T_{Periodo} \cdot CoefAproximación \quad entonces \quad Estamos\,Cerca \\ &si\,no, \quad Estamos\,Lejos \end{aligned} \tag{5.8}$$

Donde el tiempo de un periodo son 40ms, y el coeficiente de aproximación es un real entre uno y dos, cuyo significado es que vamos a llegar a la trayectoria en un tiempo acotado entre uno y dos periodos, en función de su valor.

Si el coeficiente de aproximación vale uno, estamos en el caso más restrictivo, en que suponemos que no estamos cerca hasta que podamos llegar a un punto de la trayectoria en menos de un periodo. En función de la distancia entre los puntos (que

a su vez es función de V_{bola} (1)), es posible que no podamos pasar de un punto de la trayectoria al siguiente y que al mismo tiempo sigamos considerando que el punto de destino está lejos. Esta situación es la que se trata de evitar mediante el coeficiente de aproximación.

- C1 -. Cerca

Si estamos cerca el siguiente punto al que debe dirigirse la bola es precisamente el siguiente punto de la trayectoria. Este punto es λ+1.

$$Pos_{Deseada} = Pos_{TRAYECTORIA}(Lambda + 1) \tag{5.9}$$

Una vez conocida la posición deseada para este caso, tenemos que:

- Recalcular λ (ver apartado "recalcular λ").
- Volver al paso B.

- C2 -. Lejos

El punto más cercano es λ. Sin embargo, para que la bola llegue tangente a la trayectoria, calculamos un punto de destino a "largo plazo":

$$Pos_{ALargoPlazo} = Posición_{TRAYECTORIA}\left[Lambda + Ceiling\left(\frac{T_{Llegada}}{T_{Periodo}}\right)\right] \tag{5.10}$$

(La función 'Ceiling' devuelve el entero mayor)

Es decir, nos dirigiremos a un punto de la trayectoria al que vamos a llegar en un tiempo aproximadamente igual a $T_{llegada}$ (ya que los puntos de la trayectoria están separados un tiempo $T_{periodo}$).

En este caso tenemos dos puntos objetivo: el punto objetivo a largo plazo y λ. Vamos a mover la bola en la dirección del punto objetivo a largo plazo, pero por el momento, λ sigue siendo el punto de la trayectoria más cercano.

Como T_Llegada es mayor que T_Periodo, vamos a necesitar varios periodos para alcanzar este destino a largo plazo. Por eso la función devuelve:

$$Pos_{Deseada} = \frac{Pos_{ALargoPlazo} - Pos_{Bola}}{\left|Pos_{ALargoPlazo} - Pos_{Bola}\right|} \cdot \left|V_{ALargoPlazo}\right| \cdot T_{Periodo} + Pos_{Bola} \tag{5.11}$$

Donde la velocidad a largo plazo es la velocidad deseada en el punto de la trayectoria al que nos dirigimos a largo plazo.

Ahora hay que :

- Recalcular λ (ver el siguiente apartado).
- Volver al paso B.

Recalcular λ

Calculamos las distancias entre la posición de la bola y los siguientes puntos de la trayectoria, comenzando por el punto con el que en la iteración anterior obtuvimos la distancia mínima, es decir, comenzando por λ.

Seguimos calculando las distancias a los sucesivos puntos de la trayectoria, hasta que una de estas distancias sea mayor que la anterior, en cuyo caso dejamos de calcular. Finalmente nos quedamos con el valor del índice que da la menor distancia, y ese es el nuevo λ.

En la figura 5.1 podemos ver con más detalle cómo se realiza el cálculo del nuevo λ:

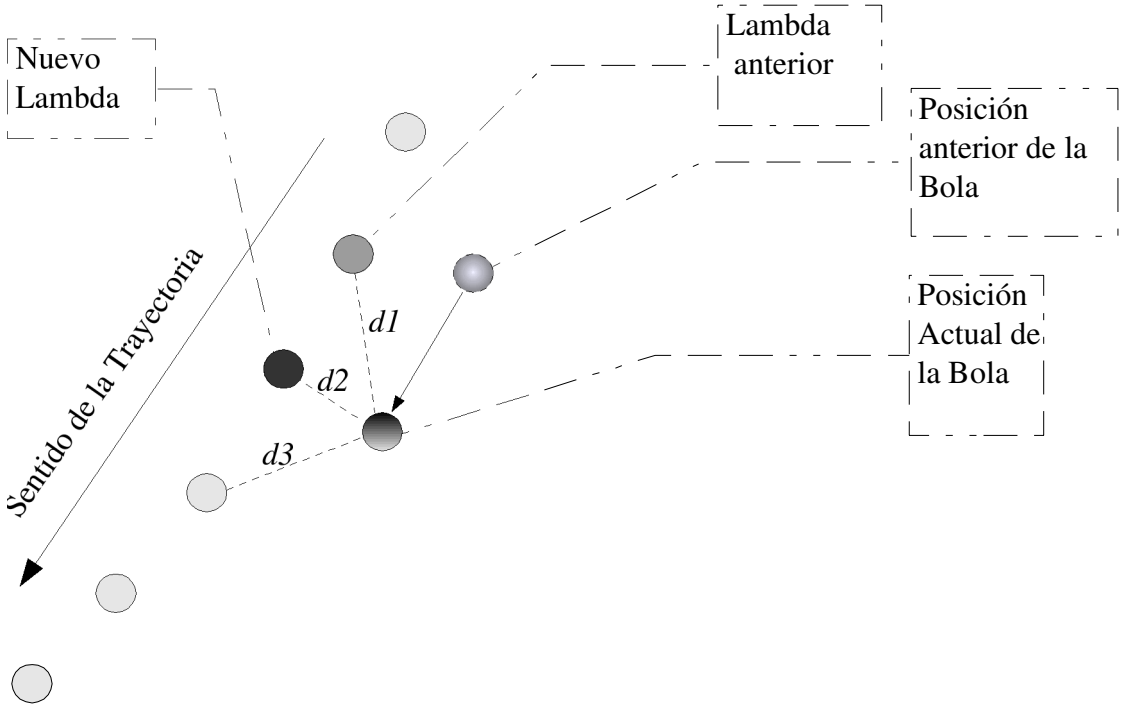

Figura 5.1: Cálculo de λ

- Vemos que para la posición anterior de la bola el punto más cercano (en gris oscuro) era λ.
- Para la nueva posición, calculamos las distancias a los siguientes puntos de la trayectoria, teniendo en cuenta que están ordenados según el sentido indicado por la flecha.
- Dejamos de calcular cuando llegamos a *d3* porque es una distancia mayor a *d2*. Inmediatamente, el punto con distancia *d2* pasa a ser el nuevo λ.

El nuevo λ en este ejemplo es el siguiente del anterior, pero esto no tiene porqué ser así. Si la bola estuviese más lejos de la trayectoria, o si se moviese a una velocidad mayor (aunque no es deseable que pase), el nuevo λ podría haber sido el punto con distancia *d3*, o incluso otro más alejado.

5.3. Simulación del paquete trayectoria

Para comprobar el correcto funcionamiento del paquete *'Trayectoria'*, hemos hecho varias simulaciones. Con los valores obtenidos para la posición de la bola, puntos de la trayectoria, etc. hemos hecho un dibujo en el que se puede observar el comportamiento teórico esperado.

La siguiente figura es un ejemplo del funcionamiento del procedimiento *"Crea_Trayectoria_Circular"* y de la función (no accesible directamente al usuario del paquete) *"Busca_Lambda"*:

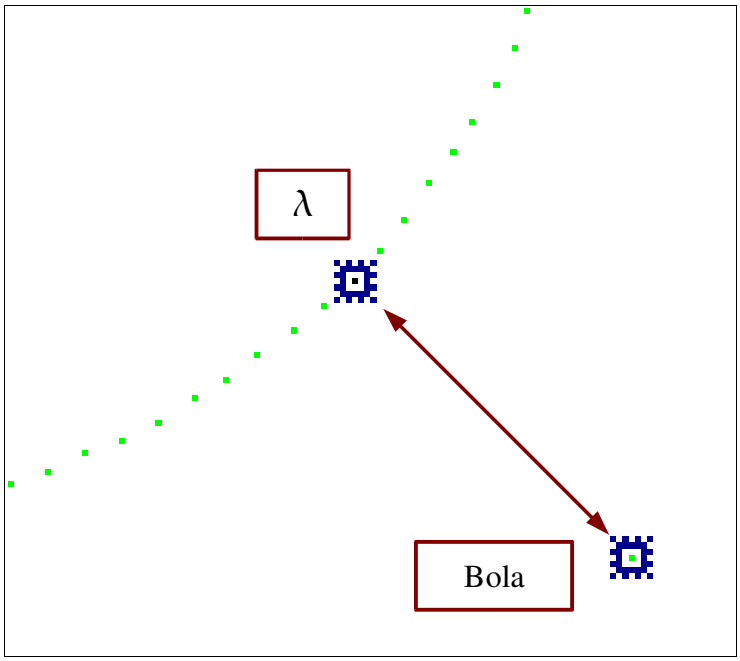

Figura 5.2: Trayectoria Circular y Cálculo de λ.

Suponiendo que en la práctica vamos a poder controlar la posición de la bola de forma perfecta, simulamos la posición a la que llega en cada periodo haciéndola igual a la posición deseada, obtenida mediante el procedimiento *'Siguiente_Posicion'*. Para que se vea mejor en el dibujo, hacemos que llegue a una posición muy cercana, para que no sea el mismo pixel y poder ver ambas en el dibujo.

Hemos utilizado la librería gráfica Gandalf, que contiene facilidades para procesar imágenes, para pintar los puntos de la trayectoria circular y de las posiciones de la esfera según la va recorriendo en nuestra simulación:

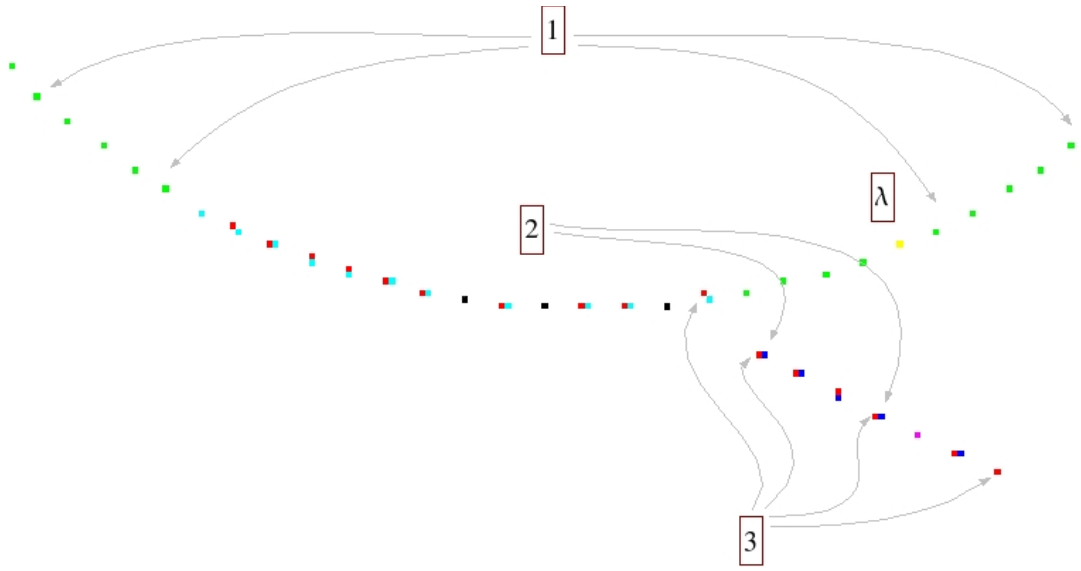

Figura 5.3: Simulación del paquete Trayectoria.

Donde:

1.- Puntos de la trayectoria a recorrer (circunferencia)
2.- Puntos a los que debería ir la bola (posiciones deseadas, dadas por "Siguiente_Posicion")
3.- Posiciones de la bola en cada instante de muestreo
λ.- Punto mas cercano para la primera posición de la bola

Además, si coinciden en la misma posición aparecen puntos de otros colores:

- Los puntos negros son puntos en los que la bola (R), la posición a la que tenía que llegar (B) y un punto de la trayectoria (G) han coincidido exactamente en el mismo pixel.
- Pasa lo mismo con los puntos morados, en los que ha coincidido la bola con la posición a la que tenía que llegar, y con los puntos azules claros, en los que la siguiente posición coincide con un punto de la trayectoria.

– El punto amarillo es el primer λ (punto más cercano).

Como se puede ver en la figura, partimos de una posición alejada de la trayectoria. Calculamos λ. El vector distancia entre la posición de la bola y λ es perpendicular a la tangente de la circunferencia en λ. Por lo tanto λ es el punto más cercano a la bola.

En las siguientes iteraciones vemos que la bola se mueve como esperábamos: se aproxima a la circunferencia, para llegar a ella de forma casi tangencial.

Verificamos que el sistema detecta que la bola está cerca de la trayectoria, ya que cuando llegamos a un entorno próximo de la circunferencia, la siguiente posición a la que debe dirigirse la bola coincide exactamente con el siguiente punto de la trayectoria.

5.4. Conclusiones del paquete trayectoria

En la práctica, la forma en que el paquete trayectoria calcula los puntos a los que debe dirigirse la bola en cada instante está reñida con el algoritmo de control usado.

Esta aparente incompatibilidad se debe a que el algoritmo de control que hemos implementado calcula, a partir de una posición de la bola y de la posición deseada, las consignas necesarias para que la bola llegue a la posición deseada y se pare allí.

Sin embargo, en un trayectoria continua y cerrada, lo que se pretende es que la bola se mueva aproximadamente con una velocidad constante en módulo.

Precisamente con esa intención creamos el paquete trayectoria: queríamos que la la bola pasase por todas las posiciones de la circunferencia con la misma velocidad.

Las posiciones deseadas, obtenidas mediante el algoritmo que calcula el siguiente punto, van a estar próximas a la posición actual de la bola en cada instante, para que pueda llegar en un periodo del control (ya hemos visto que esta distancia es de 1mm).

Sin embargo, como veremos más adelante, esto es más de la precisión con la que podemos encontrar la posición de la bola en el tablero, precisión limitada sobre todo por la resolución de la cámara. Es decir, que cuando le decimos que vaya a la siguiente posición, es posible que ya se haya pasado y tenga que volver.

Más aún, la velocidad a la que movemos los motores es limitada, por lo que estos errores van a introducir cambios que el control no va a ser capaz de seguir.

IMPLEMENTACIÓN DE LOS PROGRAMAS

Todos los programas de la parte del sistema descrita en esta memoria se han escrito en Ada, si bien, se hace uso de librerías y otros paquetes que no hemos programado nosotros, pero que nos han sido de gran ayuda en nuestro trabajo. Por eso, antes de describir nuestros programas vamos a ver cómo funcionan estos paquetes auxiliares, y las facilidades que nos han hecho decidir usarlos.

Así mismo, usamos funciones que no están escritas en el mismo lenguaje de programación que nuestro código, por lo que, gracias a que Ada define un interfaz para realizar programas usando varios lenguajes, podemos importar estas funciones y usarlas como si fuesen cualquier otra función de nuestro programa Ada. Ver manual de referencia de Ada [ADA95].

6.1. Arquitectura de la aplicación

A continuación se muestra un esquema muy simplificado con la arquitectura del sistema. En el esquema aparecen las tareas y demás módulos utilizados, y la forma en que se interrelacionan e intercambian datos. Aparece información de los datos de entrada y salida a los distintos módulos, pero no de la temporización o secuencia en que se debe llamar a esto módulos.

Lo principal es que existen dos tareas, el planificador y el control de los motores. El planificador se encarga de llamar a todas las funciones necesarias desde la inicialización de la cámara (que deberá hacerse sólo la primera iteración), pasando por la obtención del centro de la bola, procesado, algoritmo de control para calcular la inclinación que debe tener el plano para llevar la bola a la posición deseada. Finalmente escribe los datos obtenidos en el objeto protegido para que sean accesibles desde el control de los motores. Todo esto se realiza en un bucle, de forma que se repite indefinidamente (hasta que termine la ejecución).

El control de los motores accede a los datos del objeto protegido y manda la orden a los motores para que se posicionen con la inclinación requerida. En caso de no haber dato válido (si todavía no se ha inicializado la cámara, o si no se ha encontrado la bola) pone el plano en posición horizontal. Esta tarea también se realiza cíclicamente en un bucle infinito.

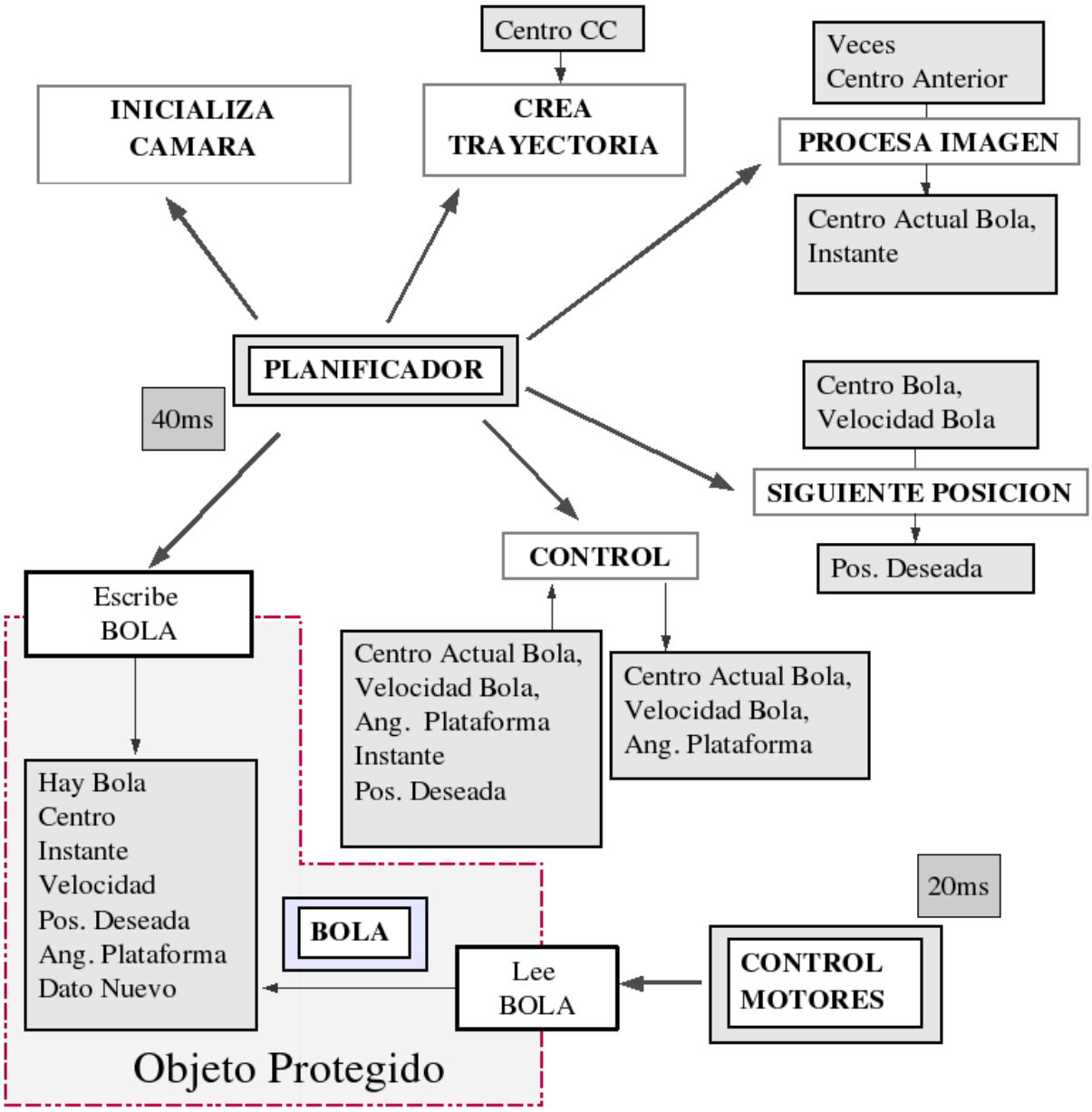

Figura 6.1: Arquitectura del sistema

En el diagrama de la página siguiente se muestra la arquitectura un poco más detallada, en la que aparece un pseudocódigo de las operaciones que realiza el planificador y el control de los motores, así como algunas funciones internas de la parte de visión (que no usamos de forma directa, pero que ayudan a comprender mejor la temporización del sistema):

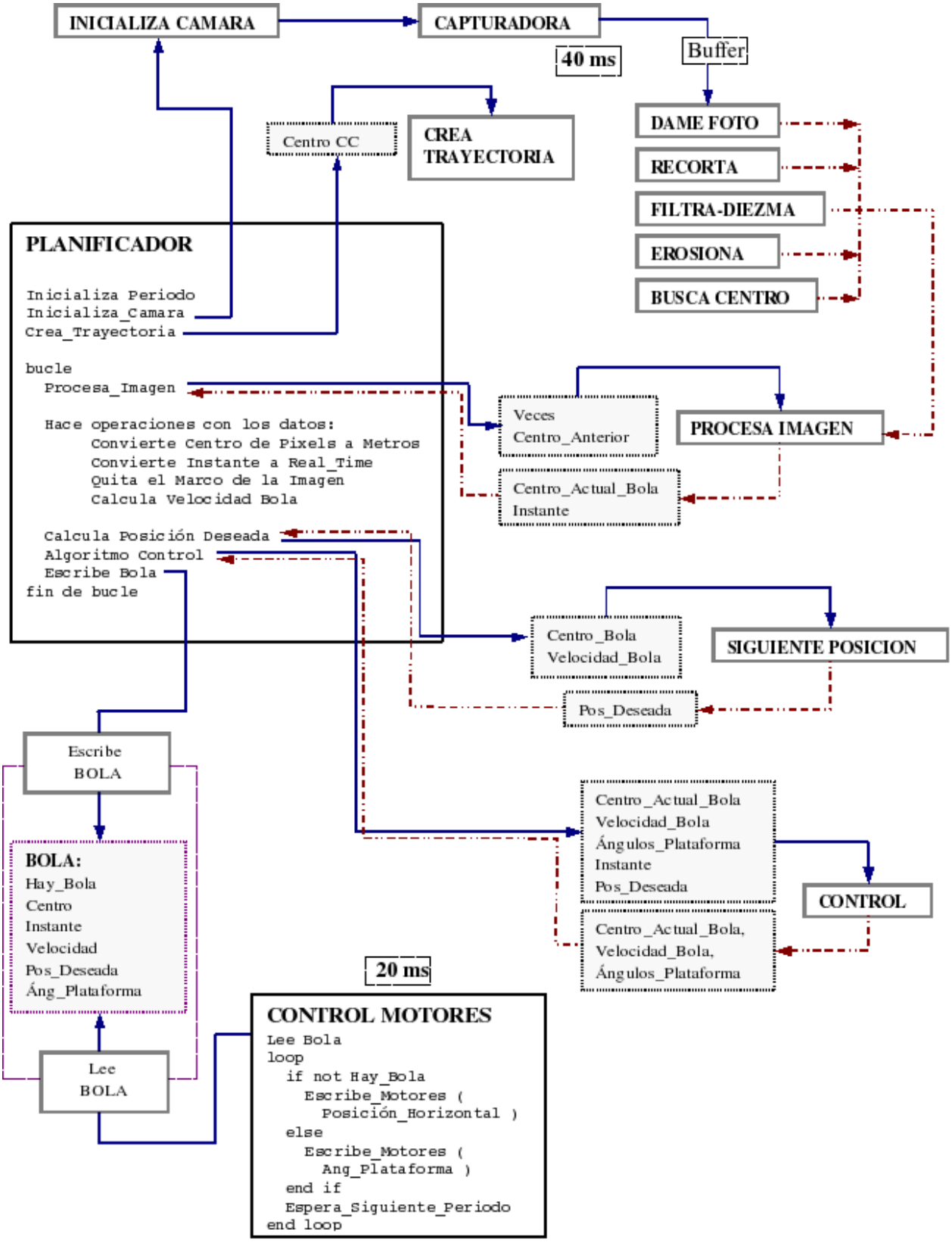

Figura 6.2. Diagrama de las relaciones entre los módulos del programa.

Además se puede ver que tenemos dos tareas periódicas, el planificador y el control de los motores, con periodos de 40ms y 20ms respectivamente. Se comunican a través del objeto protegido 'bola', de forma que se garantiza su mutua exclusión.

El periodo de 40ms de la tarea planificadora viene impuesto por la función Procesa_Imagen, que se bloquea esperando a tener una imagen nueva (en realidad dos, ya que descartamos uno de los campos). Los datos intercambiados por las funciones de procesado están detallados en la siguiente figura:

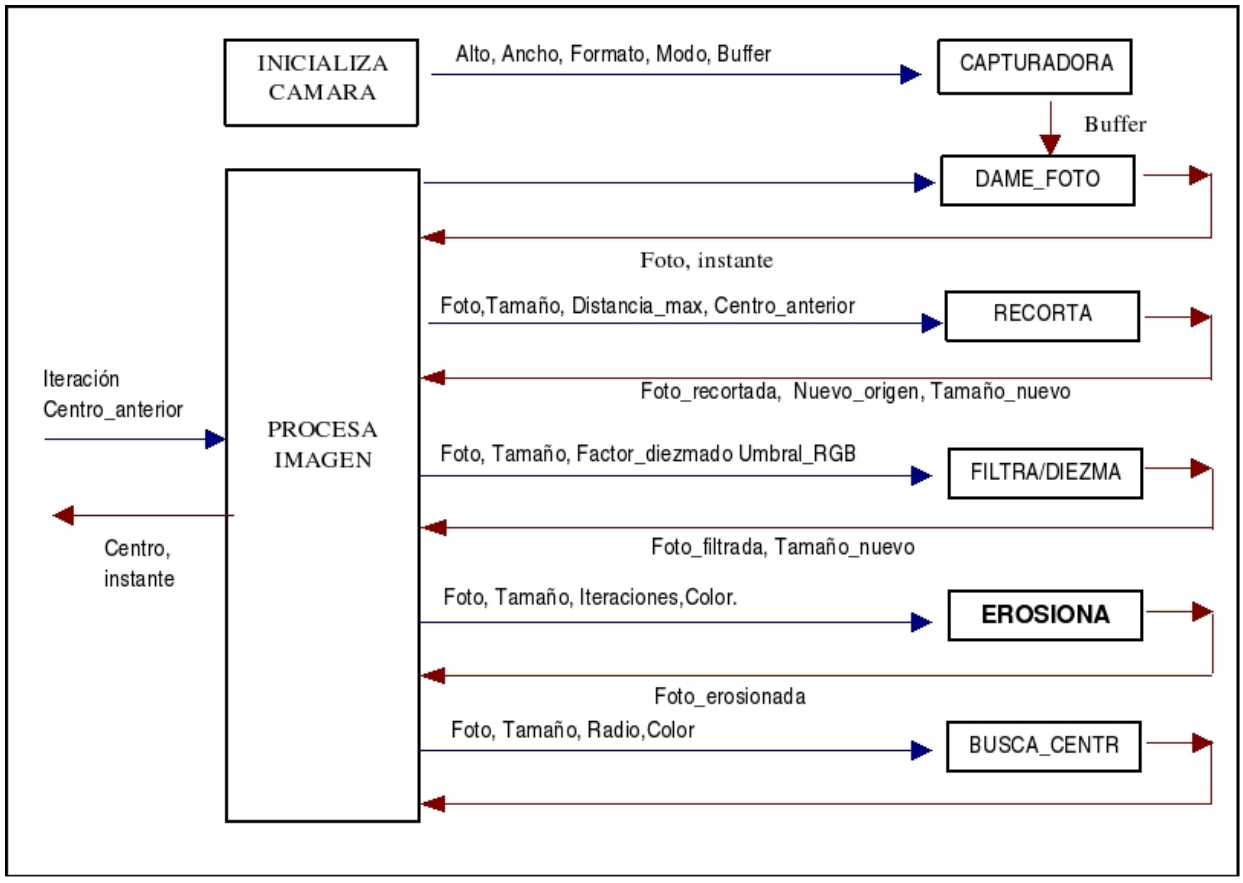

Figura 6.3: Intercambio de datos del procesado de la imagen.

Los datos que aparecen en esta imagen no es necesario que se conozcan desde el programa principal. Sin embargo se muestra para puntualizar que la función que hace que el sistema se quede esperando a la siguiente imagen se ejecuta dentro de 'dame_foto'.

Definimos los siguientes tipos de datos en la especificación del paquete *'Tipos_De_Datos'*:

```
with Matrices; use Matrices; use Matrices.Real_Arrays;

package Tipos_De_Datos is
   subtype Posicion  is  Real_Vector (1..2);
   subtype Velocidad is  Real_Vector (1..2);
   subtype Angulos   is  Real_Vector (1..2);
   subtype Pulsos    is  Real_Vector (1..2);
end Tipos_De_Datos;
```

Todos estos datos,vectores de dos componentes, son subtipos de Real_Vector, definido en *'Matrices.Real_Arrays'*, que aparece de forma más detallada en el apartado 6.5.

Para la comunicación entre las dos tareas se utiliza un objeto protegido que, como veremos, es un mecanismo que garantiza que si una tarea está accediendo al objeto, la otra no pude hacerlo hasta que la primera haya acabado, garantizándose así la integridad de los datos.

Veamos la interfaz del objeto protegido:

```
with Tipos_De_Datos; use Tipos_De_Datos;
with Ada.Real_Time; use Ada.Real_Time;

-- OBJETO PROTEGIDO --
package Paquete_Bola is

   protected Bola is
      -- techo de prioridad
      pragma Priority (10);

      procedure Lee_BOLA (
                Hay_Bola           : out Boolean;
                El_Centro          : out Posicion;
                La_Hora            : out Time;
                La_Velocidad       : out Velocidad;
                Posicion_Deseada   : out Posicion;
                Angulos_Plataforma : out Angulos);

      procedure Escribe_BOLA (
                Hay_Bola           : in Boolean;
                Nuevo_Centro       : in Posicion;
                Nueva_Hora         : in Time;
                Nueva_Velocidad    : in Velocidad;
                Posicion_Deseada   : in Posicion;
                Angulos_Plataforma : in Angulos);
```

```
   private
       ...
       ...
   end Bola;
end Paquete_Bola;
```

Los procedimientos *'Lee_Bola'* y *'Escribe_Bola'* son nuestra interfaz con el objeto BOLA. Simplemente leen o escriben todos los datos del objeto protegido para que sean accesibles desde el exterior. Sin embargo, aunque conozcamos la implementación del objeto protegido, estas funciones son la única manera de acceder a los datos encapsulados por el objeto.

A continuación se muestra la especificación del paquete Planificador:

```
with Tipos_Imagen; use Tipos_Imagen;
with Tipos_De_Datos; use Tipos_De_Datos;
with Ada.Real_Time; use Ada.Real_Time;
with Posix;

package Planificador is

   procedure Procesa_Imagen (
                    Veces           : in     Integer;
                    Centro_Anterior : in out Xy;
                    Centro_Bola     : in out Xy;
                    Instante        : in out Posix.Timespec;
                    Salida          : in out Integer );
   pragma Import (C, Procesa_Imagen, "procesa_imagen");

   function Inicializa_Camara return Integer;
   pragma Import (C, Inicializa_Camara, "inicializa_camara");

   function Calcula_Velocidad_Bola ( Centro_Anterior,
                                     Centro_Actual: in Posicion;
                                     Instante_Anterior,
                                     Instante_Actual: in Time)
                                        return Velocidad;
   task T_Planificador is
      pragma Priority(8);
   end T_Planificador;
end Planificador;
```

En el paquete *'Tipos_Imagen'* tenemos los tipos de datos que utilizan las funciones de C definidos en Ada, como el tipo Xy, que es una estructura de dos números enteros, correspondientes a una posición de la imagen en pixels.

La especificación del paquete Control_Motores, en la que la tarea T_Control_Motores tiene una prioridad más alta que el planificador, ya que tenemos que escribir invariablemente cada 20ms en los motores:

```
with Tipos_De_Datos; use Tipos_De_Datos;
with Io_Interface;

package Control_Motores is

   function Angulos_A_Tiempo (Los_Angulos : in Angulos )
                                                 return Pulsos;
   procedure Escribe_Puerto_Paralelo (Los_Pulsos : in Pulsos);

   task T_Control_Motores is
      pragma Priority (10);
   end T_Control_Motores;
end Control_Motores;
```

6.2. Implementación del Planificador

En lo que respecta al cuerpo de los paquetes, es importante conocer el orden en que se ejecutan los módulos de nuestros programas, dando completitud a la arquitectura vista en el apartado anterior. Por eso vamos a mostrar el pseudocódigo de las tareas, comenzando por el Planificador:

```
Esperas.Inicializa_Periodo;
Inicializa_Camara;
Posicion_Deseada := (others=>15.0e-2);

loop
  Procesa_Imagen (Veces,            -- in Integer --veces
                  Centro_Anterior,  -- in Xy
                  Centro_Bola,      -- in out Xy
                  Instante,         -- in out Timespec
                  Salida);          -- in out Integer

  if Salida = 0 then -- sin error
    Hay_Bola := TRUE;
    if not Primera_Vuelta then
      -- si es la primera vuelta,
      -- la velocidad vale cero =>
      -- calculamos la velocidad de la bola
      Velocidad_Bola := Calcula_Velocidad_Bola(
                              Centro_Anterior_Pos,
                              Centro_Actual_Pos,
                              Instante_Anterior_Time,
                              Instante_Time);
    end if;
```

```
    Algoritmo_Control (Centro_Actual_Pos,
                       Velocidad_Bola,
                       Angulos_Plataforma,
                       Instante_Time,
                       Posicion_Deseada);
  else


    -- Ha habido Error en el procesado
    Hay_Bola := FALSE;
  end if;
  -- ESCRIBIMOS en el OBJETO PROTEGIDO
  Bola.Escribe_Bola (Hay_Bola,
                     Centro_Actual_Pos,
                     Instante_Time,
                     Velocidad_Bola,
                     Posicion_Deseada,
                     Angulos_Plataforma);
end loop;
```

Que podemos describir de forma más clara con el siguiente diagrama de flujo:

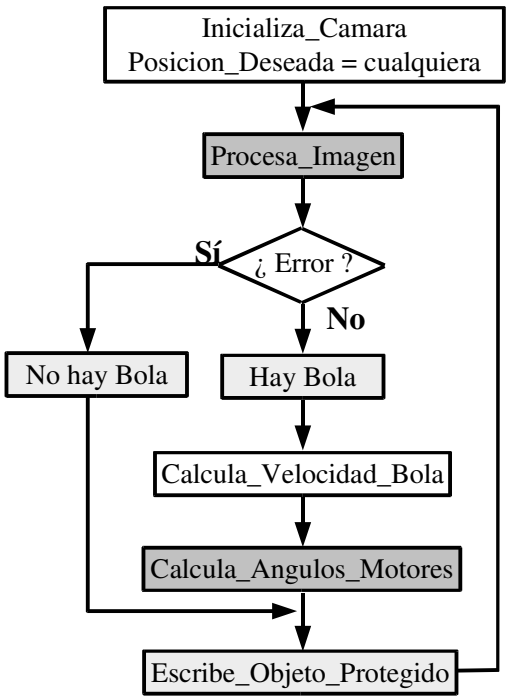

Diagrama de Flujo 6.1. Planificador.

6.3. Implementación del control de los motores

El controlador de los motores se encarga de escribir en el puerto paralelo las inclinaciones en los ejes 'X' e 'Y' que tiene que tener la plataforma. Su funcionalidad queda recogida dentro del siguiente diagrama de flujo, y más detalladamente en el pseudocódigo que le sigue:

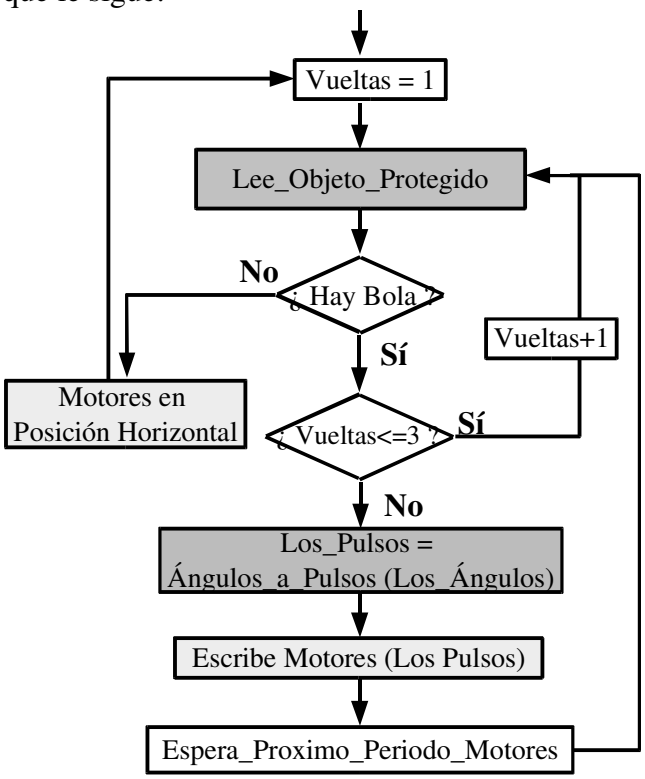

Diagrama de flujo 6.2. Control de los motores

Y su pseudocódigo:

```
loop
  Bola.Lee_BOLA (Hay_Bola,
                 El_Centro,
                 La_Hora,
                 La_Velocidad,
                 Posicion_Deseada,
                 Angulos_Plataforma);
  if not Hay_Bola then
    Los_Pulsos := Angulos_A_Tiempo ((0.0, 0.0));
    for I in 1..15 loop
      Escribe_Puerto_Paralelo(Los_Pulsos);
    end loop;
    Vueltas:=1;
```

```
  else
      -- escribe
      Los_Pulsos := Angulos_A_Tiempo (Angulos_Plataforma);
      Escribe_Puerto_Paralelo (Los_Pulsos);
      Esperas.Espera_Proximo_Periodo_Motores;
    end if;
  end if;
end loop;
```

6.4. Implementación del controlador PD

Para la implementación del controlador necesitamos tener almacenadas las variables que no deben cambiar de una llamada al procedimiento que realiza el control a otra. Por lo tanto, tenemos que declararlas en la especificación del paquete *'Control'*:

```
   -- PID --
   Error_Pos     : Real_Vector (1..2);
   Error_Pos_Ant : Real_Vector (1..2) := (others=>0.0);
   Derivada      : Real_Vector (1..2);
   Integral      : Real_Vector (1..2) := (others=>0.0);
   Consigna      : Real_Vector (1..2) := (others=>0.0);
```

De esta forma son accesibles a todos los procedimientos y funciones declaradas dentro de este paquete, especialmente al filtro y a la función que limita el ángulo calculado por el control en función de la región en que se encuentre la bola. La implementación del controlador es la siguiente:

En la parte declarativa:

```
   T  : constant Real_Type := 40.0e-3; -- 40 ms entre imagenes
   Kp : Real_Vector(1..2)  := (others=>13.0/4.0);
   Kd : Real_Vector(1..2)  := (others=>7.5/2.5);
   Ki : Real_Vector(1..2)  := (others=>0.0);
```

El cálculo de los ángulos de inclinación de los servos:

```
   La_Posicion := Filtro.Filtra_Pos (   La_Posicion,
                                        Consigna, Hay_bola );
   Error_Pos := La_Posicion_Deseada-La_Posicion;
   Derivada  := La_Velocidad;
   Integral  := Integral + (Error_Pos - Error_Pos_Ant)*T/2.0;
   Consigna  := Error_Pos*Kp - Derivada*Kd + Ki*Integral;
   Los_Angulos := -Consigna;

   Fuzzy (Error_Pos, La_Velocidad, Umbral,
          Vmin, Los_Angulos, Cerca);
```

Sin las llamadas a la función *'Filtra'* y al procedimiento *'Fuzzy'* tenemos el algoritmo de control PD sin mejoras. Los valores teóricos calculado para las constantes proporcional y derivativa han tenido que ser ajustados para que el control se realice correctamente.

En cuanto a las mejoras, el filtro toma como datos de entrada:

- La posición de la bola
- La consigna de los motores de la iteración anterior.

Si no tenemos la posición de la bola, por ejemplo si es la primera iteración, no se realiza el filtrado.

Para la corrección de las inclinaciones de los servos realizada por *'Fuzzy'* necesitamos:

- La distancia a la posición deseada (error de posición)
- La velocidad de la bola
- La distancia a la posición deseada para la primera región (es configurable). En nuestro programa es de la distancia equivalente a 100 pixels.
- La correspondiente velocidad máxima, para considerar que estamos en esa región. Hemos fijado este valor a 100 pixels por periodo de 40ms.

El resto de distancias y velocidades se calculan en función a estas.

6.5. Módulos auxiliares

Hemos utilizado el paquete *'Generic_Real_Arrays'*. Este paquete ofrece procedimientos y funciones para trabajar con vectores y matrices de forma sencilla, sin tener que descomponerlos para hacer operaciones básicas, como pueden ser multiplicaciones o sumas.

Podemos instanciar el paquete para usarlo con el tipo de datos flotante de la precisión que queramos. Nosotros hemos usado un *'digits 15'*, para tener la suficiente precisión para almacenar la posición de la bola y su velocidad si perder dígitos significativos. Sin embargo, como hemos visto, debido a la resolución de la cámara esto no era estrictamente necesario (tan sólo tenemos unos 204 pixels en 30 cm de plano, por lo tanto 204^2 posiciones en el plano). Podríamos haber usado simplemente un 'Float'. No hubiésemos perdido precisión y la instanciación del paquete *'Generic_Real_Arrays'* hubiese sido más eficiente.

En el paquete 'Generic_Real_Arrays' se definen los siguientes tipos de datos:

```
   -- TYPES --
type REAL_VECTOR is array (INTEGER range <>) of REAL;
type REAL_MATRIX is array (INTEGER range <>,
                           INTEGER range <>) of REAL;
```

Así como funciones y procedimiento para operar con vectores y matrices.
La instanciación de este paquete para el tipo de datos real con una precisión de 15 dígitos se realiza de la siguiente forma:

```
type Real_Type is digits 15;
 package Real_Arrays is new Generic_Real_Arrays (Real_Type);
 use Real_Arrays;
```

La especificación de los programas correspondientes a la parte de visión usados en la parte del sistema de la que se encarga este proyecto se describen en el capítulo 4, Captura de las imágenes. A continuación sólo se muestra su interfaz:

```
void procesa_imagen (
             int                veces,
             struct xy *        centro_anterior,
             struct xy *        centro_bola,
             struct timespec *  instante,
             int *              salida_error  );

int inicializa_camara ( void );
```

Para usar estas funciones en esta parte del proyecto tenemos que importarlas, como se verá en el apartado 6.5: Integración de la visión y el control.

6.6. Implementación de las tareas

Nuestro programa consta de dos tareas independientes. Estas tareas tienen que ejecutarse concurrentemente. Ambas tienen que tener acceso al objeto protegido, que sirve de mecanismo de comunicación entre las tareas, y contiene la información necesaria para escribir los pulsos correctamente en el puerto paralelo.

La cámara nos proporciona una imagen cada 20ms. La tarea que realiza el procesado de la imagen (el *'Planificador'*) debe tener también un periodo de 20ms. Sin embargo, como hemos visto, si no tomamos siempre el mismo campo, hay un desfase entre el campo par y el impar que hace que el centro calculado por el procesado no nos sea de utilidad a la hora de calcular la inclinación del plano en el controlador. Este desfase es tan grande, y la distancia entre los pixels tan pequeña, que a la velocidad que se mueve la bola, la distancia que recorre en un periodo tiene la misma magnitud que el ruido que proviene de la cámara. El controlador no es capaz de calcular la consigna de los motores, ya que la señal a su entrada está totalmente sumergida en el ruido.

Por tanto, hemos optado por quedarnos siempre con el mismo campo de la imagen. Esto se traduce en que el periodo de adquisición de imágenes se duplica: una imagen cada 40ms.

El periodo con el que se inicia la tarea planificadora también tiene que duplicarse. Sin embargo, como el procesado de la imagen espera a que tengamos la imagen nueva (para que siempre hagamos el procesado a partir de la imagen más reciente), no hace falta que controlemos el periodo de la tarea, que se regula por medio de la espera que introduce la captura de la imagen.

Por otro lado, tenemos la tarea que escribe (invariablemente) en los motores cada 20ms. Independientemente de lo que tardemos en llevar el plano de una posición a otra, siempre esperamos 20ms desde que comenzamos a escribir un pulso hasta que comenzamos a escribir el siguiente. En esta tarea, desde el punto de vista de la planificación temporal, éste es el único dato que nos interesa. Por lo tanto, tenemos dos tareas periódicas. El planificador, con un periodo de 40ms, y el control de los motores, con una frecuencia de 20ms.

El planificador es la tarea productora, puesto que genera el dato necesario para poder escribir en el puerto paralelo al inclinación del plano. El control de los motores es la tarea consumidora. Toma el dato de la inclinación del plano y se encarga de escribirle.

El periodo de la tarea que escribe en los motores es la mitad que el periodo de la tarea que obtiene el nuevo dato a escribir. Por lo tanto, cuando no tenemos dato nuevo (no sabemos la posición de la bola y, en consecuencia, no podemos calcular la nueva inclinación del plano para que la controle) vamos a utilizar el dato anterior para mantener la posición angular de los servos hasta que volvamos a tener un dato nuevo. En resumidas cuentas, aunque escribamos la posición de la plataforma cada 20ms, el control realmente se realiza cada 40ms.

6.7. Integración de la visión y el control

Para que la aplicación completa funcione, tenemos que integrar nuestros programas con los que realizan la captura de las imágenes. Esta parte del proyecto está programada íntegramente en C, por lo que tenemos que importarla desde nuestro programa principal, escrito en Ada. El lenguaje de programación Ada provee una herramienta para importar entidades definidas en otros lenguajes de programación: '*pragma Import*'. En el manual de referencia de Ada 95 [ADA95], dentro del anexo B (*"Interface to Other Languages"*), se describe cómo funciona. A nosotros nos interesa sobre todo el subapartado B.3 (*"Interfacing with C"*), en el que se describen las relaciones existentes entre los diferentes tipos de datos en Ada y C.

Para nosotros, todo el código en C que se encarga de capturar imágenes, procesarlas, etc. y devolver el centro de la bola, no es más que la siguiente función:

```
void procesa_imagen (
          int                 veces,
          struct xy *         centro_anterior,
          struct xy *         centro_bola,
          struct timespec *   instante,
          int *               salida_error  );
```

Desde nuestro programa Ada, definimos los correspondientes tipos de datos, para que sean compatibles con los datos de la función anterior:

```
type Xy is record
   X : Integer;
   Y : Integer;
end record;
```

Sólo hace falta definir este registro ya que el resto de datos también existen en Ada y, aunque por ejemplo el número de bytes de un *'int'* y de un *'Integer'* es dependiente de la implementación (y pueden por tanto no tener el mismo tamaño) en nuestro caso son iguales.

Ahora definimos el procedimiento, siguiendo las reglas definidas en la sección del manual Ada [ADA95] para los distintos tipos de datos entre ambos lenguajes:

```
procedure Procesa_Imagen (
             Veces               : in          Integer;
             Centro_Anterior     : in out      Xy;
             Centro_Bola         : in out      Xy;
             Instante            : in out      Posix.Timespec;
             Salida              : in out      Integer  );

pragma Import (  C, Procesa_Imagen, "procesa_imagen"  );
```

El significado de los datos que usa la función Procesa_Imagen se explican en el capítulo 4, captura de las imágenes, donde se da la interfaz del software usado por esta parte del proyecto con la parte de visión (programada en C).

Como en C no podemos crear procedimientos igual que en Ada, sino que sólo tenemos funciones, cuando se necesite devolver más de una variable es necesario usar punteros, de forma que cambiando el contenido de la dirección de memoria a la que apunta el puntero, podemos tener acceso a ese dato desde el exterior de la función [ADA95][CTR_LAN].

Un puntero a un parámetro en C es equivalente a varios tipos de datos en Ada, como por ejemplo:

- un parámetro de entrada de tipo puntero a objeto del mismo tipo que el puntero en C
- un parámetro de salida o entrada/salida del mismo tipo que el objeto apuntado en C
- un registro en Ada, si el objeto apuntado en C también es un registro
- un puntero a un subprograma en Ada también es equivalente en C a un puntero a función, siempre que el prototipo de la función se corresponda con la especificación del subprograma.
- etc.

Siguiendo este criterio, los punteros en C, como el instante, el centro de la bola o el centro anterior, que son usados en la función *'procesa_imagen'* como parámetros de entrada, son equivalentes en nuestro programa Ada a los tipos de entrada-salida, del mismo tipo que el parámetro apuntado por el puntero correspondiente en C.

Usando las pautas dadas en el manual de referencia, hemos creado el tipo 'Xy' en nuestros programas Ada como un registro de dos enteros, para que sea compatible con el tipo *'struct xy'* de C. El tamaño del entero es dependiente de la implementación. Por tanto, si el tamaño del *'Integer'* de Ada no fuese igual al tamaño del *'int'* de C, tendríamos que usar la librería *'Interfaces.C'* para garantizar que sean iguales: cada uno de los tipos declarados en la librería *'Interfaces.C'* es compatible con el correspondiente tipo de C. Hecho esto, podemos usar la función *'procesa_imagen'* igual que cualquier otro procedimiento Ada.

6.8. Medidas temporales

En sistemas operativos de tiempo real estricto es imprescindible monitorizar el tiempo de ejecución de todas las tareas y detectar situaciones en las que se exceda el tiempo estimado de ejecución de peor caso [GON03].

Muchos algoritmos de planificación de tiempo real flexible requieren la capacidad de medir el tiempo de ejecución y ser capaces de realizar acciones de planificación cuando una cierta cantidad de tiempo se ha consumido (por ejemplo, servidores esporádicos en sistemas por prioridades fijas, o el servidor de ancho de banda constante en sistemas planificables EDF) [GON03].

6.8.1. Tiempos de ejecución de los programas

Hemos medido el tiempo de ejecución de cada función o procedimiento por separado, aunque no es estrictamente necesario desde el punto de vista de la planificación temporal, se ha decidido hacer esta medida de forma detallada para saber si existe algún cuello de botella en nuestro sistema. Así sabremos con precisión qué función es la que requiere más tiempo de CPU, y cuánto es este tiempo.
En función de los valores obtenidos podremos actuar en consecuencia, p.ej. relajando los requerimientos temporales de las partes del sistema que lo requieran.

En la tabla 6.1. podemos ver los tiempo de ejecución de los diferentes módulos de nuestros programas.

Procedimiento / / Función	***Tiempo de Ejecución***		
	Máximo	**Medio**	**Mínimo**
Procesa_Imagen	*Entorno bola:* *1.962ms* *Toda la imagen:* *17.20ms*	*1.612ms* *8.233ms*	*1.280ms* *7.313ms*
Inicialización de la Cámara	*3.4899 seg*	*3.4898 seg*	*3.4897 seg*
Crea_Trayectoria	*1.459ms*	*1.444ms*	*1.389ms*
Siguiente_Posicion	*21μs*	*14.59μs*	*9μs*
Control	*32μs*	*24.8μs*	*16μs*
Control_Motores	*5μs*	*2.8981μs*	*2μs*

Tabla 6.1. Tiempos de ejecución de los programas.

6.8.2. Periodo de las tareas

A priori sabemos que la cámara de vídeo tarda 40ms en capturar cada imagen. Sin embargo, cada uno de los campos de la imagen es capturado cada 20ms. Además tenemos que escribir en los motores cada 20ms. En la siguiente tabla aparecen las medidas realizadas:

Tarea	*Periodo [ms]*		
	Máximo	***Medio***	***Mínimo***
Planificador	*40.113*	*39.996*	*39.886*
Control_Motores	*20.02*	*20.0088*	*20.001*

Tabla 6.2. Periodos de las tareas.

6.8.3. Dependencias temporales

Ahora nos queda saber si vamos a tener tiempo suficiente para realizar el procesado de la imagen, ejecutar el algoritmo de control de la posición de la bola, escribir los pulsos para que se muevan los motores... además de todo el tiempo extra que vamos a necesitar para que se ejecuten el resto de instrucciones, cambios de contexto, escritura en objetos protegidos o variables protegidas, etc.

'PROCESA_IMAGEN': Analizamos la primera fila de la tabla 6.1, que corresponde a los tiempos de ejecución del procedimiento *'Procesa_Imagen'*. No se tiene en cuenta el tiempo que tarda en esperar a la siguiente imagen hasta que ésta esté lista. De esta forma sólo estamos midiendo el tiempo que tarda en hacer el procesado de la imagen, propiamente dicho. Se aprecia claramente que el tiempo de ejecución es mucho mayor cuando analizamos al imagen completa (toda la foto) y no solo un cuadro alrededor de la última posición de la bola.

Afortunadamente, el tiempo mayor (en torno a 17ms) sólo se va a dar la primera vez que hagamos la captura y procesado de la imagen (siempre que luego seamos capaces de encontrar la bola el resto de iteraciones que dure nuestro programa). En caso contrario, volveríamos a analizar la imagen completa.

'PLANIFICADOR': Para el análisis temporal del planificador, tenemos que tener en cuenta que su periodo depende de los tiempos de ejecución de los procesos que engloba (que son todos los vistos en la tabla excepto *'Control_Motores'*). El que tiene mayor peso es el procesado de la imagen, ya que tiene que esperar a que haya una imagen nueva disponible.

Como vemos en la tabla, el tiempo máximo de ejecución del procesado esta en torno a los 40ms, que es precisamente el tiempo medio de espera para una foto válida. Frente a este periodo, podemos considerar que los tiempos de ejecución del resto de procedimientos son despreciables. Analizamos para el caso peor, en que tuviésemos que esperar a la imagen 40ms, más el tiempo que tardase en realizarse el procesado, unos 17ms, podríamos decir que una cota superior al tiempo de ejecución

es de unos 60ms. Sin embargo, esto sólo es cierto la primera iteración, en que es posible que esperemos 20ms a la primera imagen, la descartemos y esperemos otros 20ms a la segunda. El resto de iteraciones, como el tiempo de ejecución de nuestros procedimientos es pequeño (sumando los tiempos máximos, unos)

'INICIALIZA_CAMARA': Como indica su nombre, esta función pone a funcionar la cámara. Realiza una espera de 3s al final para que el PLL de la cámara tenga tiempo de sobra para engancharse en fase. El tiempo de ejecución de este proceso medido desde el planificador es de aproximadamente 3.5s.

'CONTROL' , 'SIGUIENTE_POSICION' y 'CONTROL_MOTORES': El tiempo de ejecución máximo (que hayamos medido) del algoritmo de control es de 32μs, y tardamos unos 21μs en calcular la siguiente posición. Por lo tanto, pueden considerarse despreciable sin miedo a cometer ningún error ya que son más de tres órdenes de magnitud inferiores al periodo de ejecución de la tarea en la que se ejecutan (*'Planificador'*). Lo mismo ocurre con el control de los motores, que tiene un tiempo de ejecución mucho más pequeño que su propio periodo (la mayoría del tiempo está esperando).

En la figura 6.4 vemos como en la primera iteración puede que no tengamos tiempo suficiente para hacer todas las operaciones dentro del periodo de 40ms, ya que sólo esperar a dos imágenes consecutivas puede tardar este tiempo. Con el tiempo de procesado de la primera imagen podemos llegar hasta casi los 60ms. No nos importa siempre que sólo pase una vez.

Siempre que no haya que hacer el procesado de la imagen completa tenemos tiempo para hacer todas las operaciones del planificador. Sabemos que es así porque los tiempos de de todo el planificador salvo las esperas es menor que 2ms. Por lo tanto, esperaremos 18ms a la imagen del campo que no nos vale, otros 20ms a la imagen que sí, y en total tenemos los 40ms del periodo del planificador. Todavía tenemos margen (casi 18ms) para introducir complejidad al sistema, como procesando imágenes más grandes (de más resolución) o realizando un control más sofisticado.

En la siguiente figura se muestra cómo funcionarán las tareas que ejecutándose de forma concurrente. Como puede verse, el periodo del planificador es la mitad que el periodo del control de los motores. La comunicación entre ellas se realiza mediante el objeto protegido.

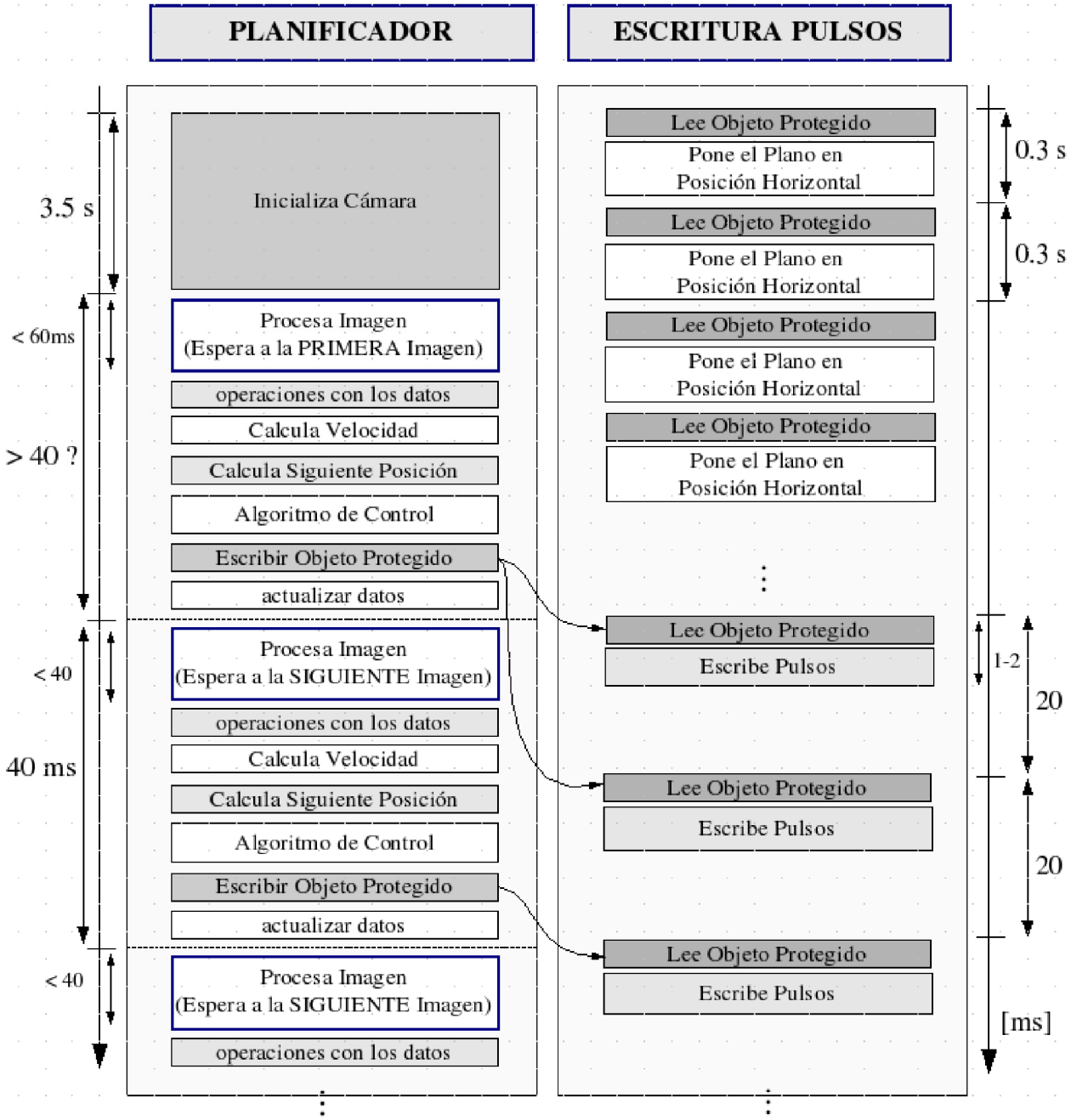

Figura 6.4. Ejecución de las tareas planificadora y de escritura de los pulsos.

Lo más relevante de esta figura es que como ambas tareas cumplen sus periodos después de la primera iteración, se sincronizan, por lo que siempre vamos a leer el objeto protegido aproximadamente en el mismo instante.

6.9. Esperas

Por motivos de sincronización, nuestras tareas van a necesitar ejecutar funciones que las "duerman" durante un determinado intervalo de tiempo. Finalizado este intervalo, la tarea suspendida debe continuar ejecutándose.

Además, podemos aprovechar el mecanismo de esperas para escribir los pulsos en los motores, ya que podemos empezar a contar cuando ponemos ambos pulsos a valor alto, y luego mirar que pulso, 'x' o 'y', hay que poner primero a nivel bajo, para esperar justamente el intervalo temporal que permanezca arriba.

Para simplificar los programas en los que hace falta hacer uso de los mecanismos de temporización, hemos programado el paquete *esperas*. Este paquete implementa todos los procedimientos y funciones que nos han hecho falta para la sincronización entre tareas y para la correcta escritura de los pulsos, entre los que se incluyen esperas, comparaciones temporales, inicialización de relojes (usados como referencia para la sincronización de eventos), etc.

Las medidas que hemos realizado de los tiempos de ejecución de los programas, comentadas anteriormente, están incluidas en este paquete en los valores para los *offsets* temporales, que pueden corresponder directamente al tiempo de ejecución de una tarea en particular, o bien pueden ser la suma de los tiempos de ejecución de un grupo de programas.

Usamos estos *offsets*, por ejemplo, para sincronizar la lectura y la escritura de una variable (ya que no nos interesa leer su contenido hasta que no haya sido actualizado), o para esperar durante la primera vuelta el tiempo necesario para que se inicialice la cámara y para que se ejecute el procesado de la imagen (ya que al tener que realizarse la primera vez de forma exhaustiva, recorre la imagen completa, lo que conlleva un aumento considerable del tiempo de cómputo).

La especificación del paquete *'Esperas'* se muestra a continuación:

```
-----------------
-- Esperas.ads --
-----------------
with Ada.Real_Time; use Ada.Real_Time;

package Esperas is

   Periodo_Motores : constant Time_Span := Milliseconds(20);

   procedure Inicializa_Periodo_Motores;
   procedure Inicializa_Pulsos;
```

```
   procedure Espera_Menor_Tiempo (Tiempo1,
                                  Tiempo2 : in Time_Span);
   procedure Espera_Mayor_Tiempo (Tiempo1,
                                  Tiempo2 : in Time_Span);

   procedure Espera_Proximo_Periodo_Motores; -- 20ms

end Esperas;
```

Este paquete se usa básicamente para realizar correctamente la temporización de la escritura de los pulsos.
Los procedimientos *'Espera_Menor_Tiempo'* y *'Espera_Mayor_Tiempo'* comparan el valor de dos intervalos temporales y esperan el menor o el mayor, respectivamente. Estas esperas son relativas a un instante de tiempo dado por la llamada a *'Inicializa_Pulsos'*.

Esperar que transcurra un intervalo de tiempo desde un instante absoluto facilita la temporización, ya que no tenemos que saber cuánto tiempo tardan en ejecutarse las instrucciones intermedias desde que comienza el periodo y escribimos el comienzo de los pulsos hasta que termina cada uno de ellos.

'Espera_Proximo_Periodo_Motores' hace una espera de 20ms (*Periodo_Motores*) desde el instante en que hayamos inicializado el periodo de los motores, llamando a *'Inicializa_Periodo_Motores'*.

6.10. Ajuste de las imágenes

El centro de la bola, calculado por la parte de este proyecto encargada del procesado de imagen, está expresado en forma de coordenadas o pixels, cuyo origen se sitúa en la esquina superior izquierda de cada imagen. Cuando sólo se procesa una parte de la imagen, el centro devuelto sigue teniendo como referencia el origen de coordenadas de la imagen completa, de modo que para nosotros es como si siempre se procesasen las imágenes en su totalidad.

Ahora bien, el origen de coordenadas a la hora de escribir los ángulos depende de cómo esté colocada la cámara en relación a los motores. En nuestro caso no coinciden, por lo que tenemos que hacer un cambio de coordenadas para que las posiciones de la bola devueltas por el procesado se correspondan luego con las posiciones físicas en el plano, de acuerdo al sistema de coordenadas impuesto por la colocación de los motores.

La cámara está suspendida sobre el centro del plano, por lo que la esquina de la imagen no corresponde a la esquina del tablero: además del cambio de coordenadas debemos hacer una traslación. Para saber la relación existente entre los puntos del plano y los de las imágenes, hemos utilizado la librería gráfica desarrollada por José Luis Mantecón para pintar los puntos correspondientes a las esquinas del tablero, de forma que conocemos los offsets entre el origen de coordenadas de la foto y la esquina superior izquierda del tablero, necesarios para hacer la traslación:

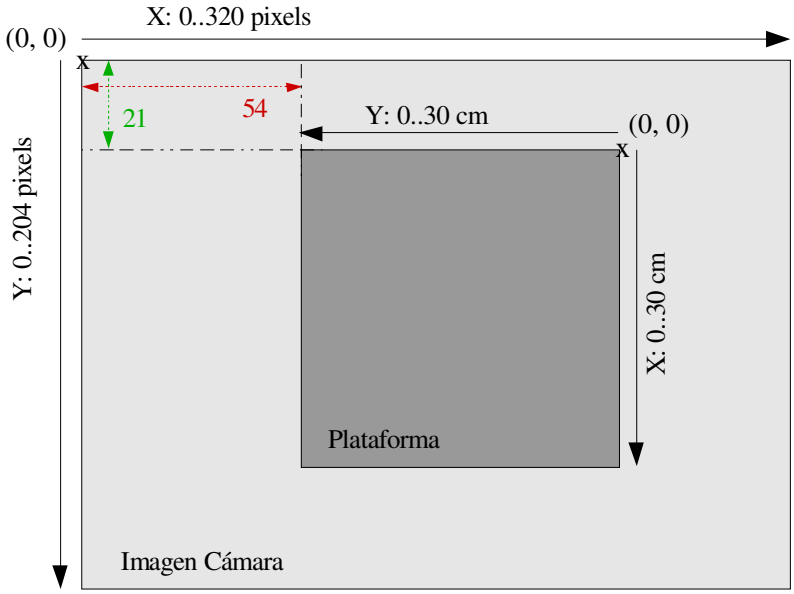

Figura 6.5: Traslación del origen de coordenadas.

Atendiendo a lo expresado en esta figura, podemos escribir la posición de un punto en el tablero de la forma:

Trasladamos el tablero al origen de coordenadas:

$$(X,\ Y)\ \rightarrow\ (X-54,\ Y-21) \tag{4.1}$$

Los ejes están cambiados, y los 30cm del tablero equivalen a 204 pixels:

$$(X-54,\ Y-21)\ \rightarrow\ (Y-21,\ 204-X+54) \tag{4.2}$$

Finalmente, podemos expresar las coordenadas de la bola en el sistema de coordenadas de los motores, en función de las coordenadas dadas por la cámara:

$$(X,\ Y)|_{PLANO}\ =\ (Y-21,\ 258-X)|_{IMAGEN} \tag{4.3}$$

6.11. Resultados

En este apartado se muestra la salida del sistema para las diferentes combinaciones de mejoras al algoritmo de control. A diferencia del apartado 2.11.2, estos datos no se han obtenido mediante simulación, sino a partir de las salida del sistema observable para nosotros, es decir, la posición de la bola obtenida mediante el procesado de imágenes.

El experimento consiste en enviar siempre la misma consigna de posición, que concretamente es el centro de la plataforma (para cada eje esto corresponde a la posición alejada 15cm del origen). El sistema tiene que ser capaz de mover la bola a esa posición y mantenerla. El resultado es el siguiente:

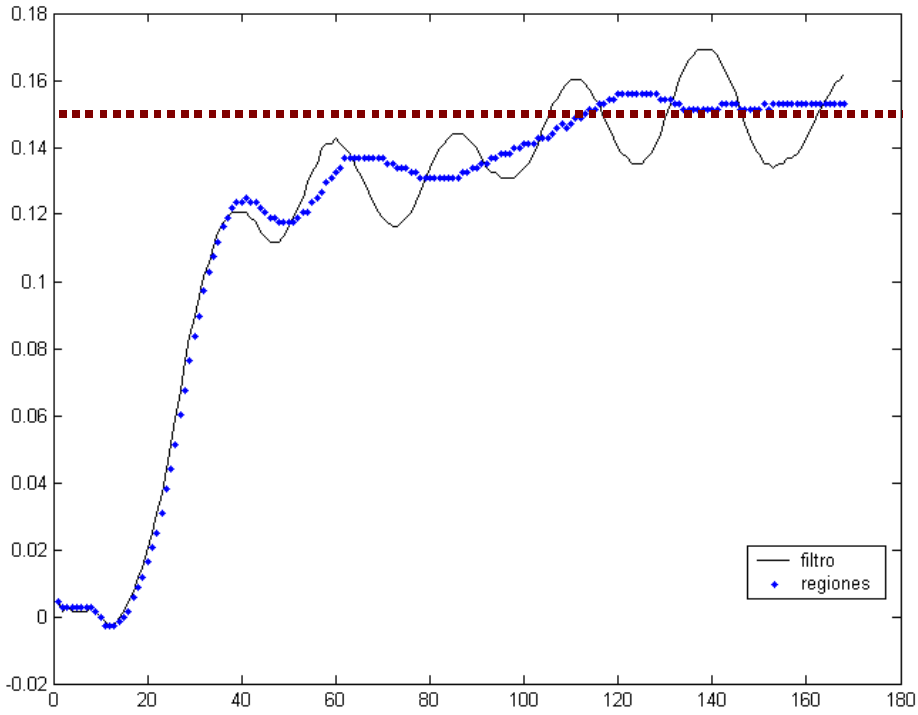

Figura 6.6. Salida con Filtro y salida con regiones de posición y velocidad acotadas.

En esta figura se muestra a modo de comparativa, la salida del sistema con cada una de las dos mejoras mencionadas. Las curvas corresponden a simulaciones independientes. La línea gruesa a trazos es la consigna.

Puede verse que sólo con el filtro (línea continua) el sistema se queda oscilando en torno a la posición deseada, ya que el ruido introducido a la entrada por la

digitalización de las imágenes es elevado.

En la siguiente figura se muestran la salida del sistema sólo con el controlador PD (línea discontinua), frente a las posiciones de la bola implementando tanto el filtro como la división del espacio en regiones en el control.

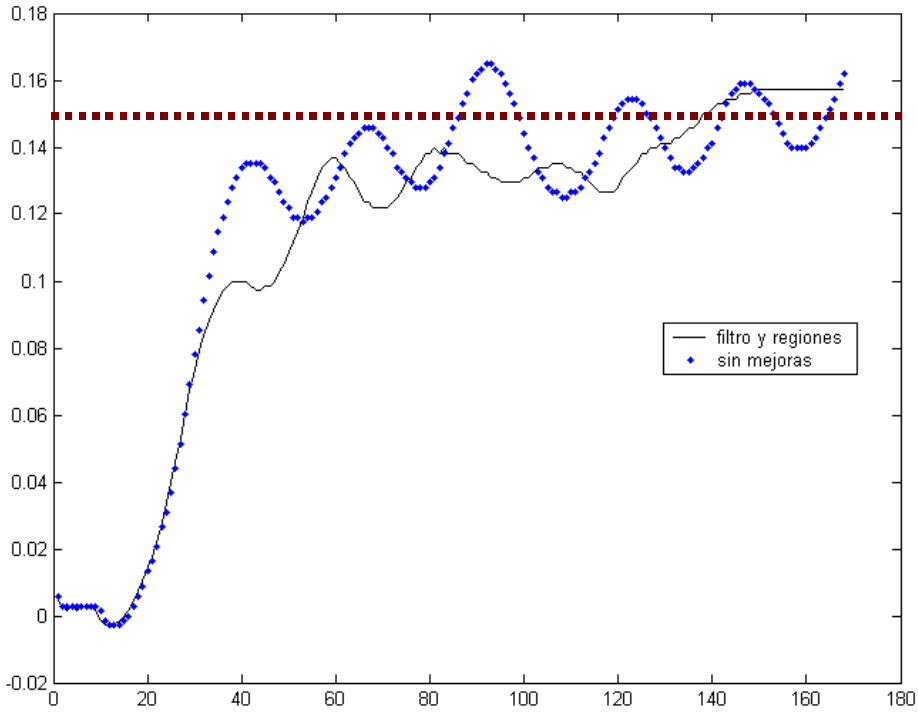

Figura 6.7. Salida del sistema con la mejoras y sin ellas.

El sistema sin mejoras (línea de puntos) llega más rápido a un entorno de la posición deseada, lo cual es lógico ya que no limitando el ángulo de inclinación el sistema tiene un tiempo de subida más pequeño y más sobredisparo. Una vez cerca de la posición deseada el sistema sigue controlando sin llegar a estabilizarse.

Con mejoras (línea continua) la convergencia es más lenta, pero conseguimos que la posición de la bola se estabilice y el sistema deje de controlar cuando estamos muy cerca de la posición objetivo.

CONCLUSIONES Y TRABAJO FUTURO

7.1. Conclusiones

Podemos considerar que hemos alcanzado los objetivos del proyecto, ya que hemos desarrollado una aplicación automatizada en el sistema operativo MaRTE OS, en la que se ejecutan varias tareas de forma concurrente y en la que se utiliza la visión artificial para realizar tareas de control.

El sistema es capaz de encontrar la posición de una bola en cualquier punto de un tablero móvil, y además controlarla. Para encontrar la posición de la bola, un gran inconveniente ha sido la fuerte dependencia con la iluminación, que hace que el calibrado tenga que ser muy preciso y, por lo tanto, complicado. Para el control de la posición, sin embargo, el impedimento fundamental ha sido la baja resolución de la cámara, ya que pequeños errores en la posición de la bola se traducen en grandes correcciones por parte del algoritmo de control, lo que hace que el sistema pierda la estabilidad.

Para solucionar este problema ha sido necesario introducir algún elemento adicional dentro del bucle de control, además del simple controlador, ya que el comportamiento del sistema, si bien no se hacía inestable, sí que distaba mucho de funcionar correctamente. Las causas de este mal funcionamiento son principalmente el ruido añadido a la señal estimada de la posición de la bola, la falta de precisión en el ajuste de los parámetros del algoritmo de control, el tiempo de respuesta de los motores, la incertidumbre en la posición angular instantánea de los mismos, etc. donde los dos primeros son los más significativos.

La principal conclusión es que para poder mejorar el control debemos tener una medida más precisa de las posiciones, tanto en frecuencia como en valor absoluto, lo que implica la mejora de la cámara, pasando de la analógica con entrelazado a digital no entrelazada.

La aplicación construida trabaja de forma totalmente automatizada. No necesita parámetros de entrada (sólo estar calibrada), y sigue funcionando correctamente hasta que se la para. Si actuamos sobre la bola (introduciendo un movimiento impredecible para el sistema), se recupera sin problema. Incluso si la bola desaparece (llevando al sistema a su estado inicial, en el que el pano se coloca en posición horizontal, tenemos un tiempo de espera elevado debido al procesado de la imagen completa, etc), en el momento en que se recupera la visión de la bola, se comienza a controlar su posición.

El tiempo consumido por las tareas para su ejecución es inferior al tiempo disponible, por lo que podemos asegurar que se cumplen todos los requisitos temporales.

Hemos podido despejar satisfactoriamente la duda de si podríamos ser capaces de controlar directamente desde MaRTE OS un controlador de posición, con requisitos temporales en torno a los 10μs, sin usar hardware específico.

7.2. Trabajo futuro

Como líneas futuras, se proponen principalmente algunos cambios en los dispositivos del sistema físico para aumentar el rendimiento:

Lo primero, sería conveniente pasar a una cámara de mayor resolución. Con este cambio conseguimos tener más pixels y de menor tamaño. Disminuye el ruido en la estima de la posición de la bola, ya que disminuye el ruido de cuantificación, que ha sido la principal fuente de error del sistema. Además, como habrá más puntos pertenecientes a la bola, aumenta la eficiencia de las funciones que realizan el procesado de la imagen. Por otro lado, un incremento en el número de pixels hace que el tiempo de procesado aumente, pudiendo llegar a ser crítico.

Aumentando también la frecuencia de adquisición de la cámara mejoraría significativamente el bloque de control, no sólo porque la señal de realimentación está más limpia de ruido, sino porque el periodo de control disminuiría, controlando la posición de la bola cada menos tiempo.
La cámara digital de la que podríamos disponer nos da 100 imágenes completas por segundo, pudiendo llegar hasta 3000 imágenes por segundo si sólo tomamos una porción de la imagen.

La forma en que se realizan las capturas es distinta para una cámara analógica que para una cámara digital. La cámara analógica utiliza un formato conocido como vídeo entrelazado, es decir, captura cada fotograma en dos campos. En teoría la frecuencia de muestreo es el doble. Sin embargo, cada campo tiene la mitad de resolución vertical. La imagen completa (obtenida entrelazando los campos) no sirve para realizar el control, ya que cada campo ha sido capturado en un instante distinto (los objetos que no permanecen estáticos se ven afectados por una distorsión, que es mayor cuanto más haya variado la posición del objeto entre los instantes de captura de los campos par e impar). Esto no ocurriría con la imagen completa, que nos daría una imagen completa cada vez.

Además de la cámara, se podría cambiar el algoritmo de control para que

incluyese la función de transferencia de los motores en la planta. Se espera que al añadir esta mejora, la función de transferencia del modelo del sistema se aproxime más a la función de transferencia real de la planta, ya que la velocidad de los motores no es muy alta e introduce por tanto un retardo apreciable que, sin embargo, no hemos estado considerando en la simulación (suponíamos que los motores eran infinitamente rápidos, consiguiendo incrementos de posición angular instantáneos).

Incluir esta función de transferencia hará que el sistema sea más fácil de controlar, pero a su vez será más difícil simularlo y calcular el controlador adecuado, ya que por lo menos debemos considerar el modelo de segundo orden del servo, para añadir el efecto de la aceleración.

Respecto a lo que al algoritmo de control se refiere, es deseable que el criterio para optimizar el control no parta de una formulación matemática, sino del punto de vista de una aplicación concreta [OK_97] [KB_87]. Con el control óptimo se pretende obtener un sistema que sea el mejor posible respecto a un cierto índice de calidad de funcionamiento o criterio de proyecto.

El concepto de optimización de sistemas de control abarca el mencionado índice de calidad y un diseño que produce el sistema de control óptimo dentro de unos límites impuestos por restricciones físicas. El índice de calidad varía en función del error que se quiera minimizar, planteándose diferentes problemas de optimización: de tiempo mínimo, de regulación lineal, de seguimiento o persecución...

Por otro lado, los sistemas de control adaptativos tienen la capacidad de acomodarse a cambios impredecibles del entorno, siempre que estos cambios lleguen del exterior del sistema. Estos sistemas de control se caracterizan por medir automática y constantemente las características dinámicas (tales como la función de transferencia) de su planta, compararlos con las características dinámicas deseadas, y usar la diferencia para variar los parámetros del sistema ajustables (generalmente características del controlador) o para generar una señal que actúe de forma que el funcionamiento óptimo se mantenga a pesar de los cambios en el entorno.

Por lo tanto, el siguiente paso lógico sería cambiar el algoritmo de control PID por otro controlador de mayores prestaciones, como por ejemplo un controlador óptimo (como por ejemplo un filtro de Kalman), adaptativo, etc, o incluso podrían usarse redes neuronales o lógica difusa para obtener un controlador robusto. La parte negativa es que sacrificaríamos la sencillez y el bajo tiempo de cómputo del controlador PID que, además, sabemos que tiene un comportamiento predecible y fácilmente ajustable en caso de cambio de las condiciones externas a nuestro sistema. Consideramos que una opción más que razonable sería la implantación de un controlador PID óptimo y adaptativo (o PID auto-sintonizado), para llegar a un compromiso entre sencillez y prestaciones del sistema.

Aparte de los cambios ya mencionados, podemos añadir complejidad al sistema de forma que, en caso de que las mejoras anteriores funcionen y el sistema tenga un comportamiento similar al ideal, puedan recorrerse trayectorias más complejas, como trayectorias circulares, elípticas, cuadradas, etc. Es decir, que pueda recorrerse una laberinto formado por cualesquiera secciones de rectas y/o curvas.
Podremos medir el grado de control que tiene el sistema sobre la posición de la bola introduciendo trayectorias que exijan cambios bruscos de dirección.

Otras complicaciones adicionales podrían ser la implementación física de las paredes de un laberinto, el reconocimiento de la trayectoria mediante visión artificial y, si esto es posible, el reconocimiento de obstáculos (como por ejemplo agujeros en la superficie del tablero que haya que evitar para recorrer el laberinto).

BIBLIOGRAFÍA

Sistemas Operativos de Tiempo Real:

[LIN] http://linuxdevices.com/articles/AT4627965573.html
[GAY] http://gayuba1.datsi.fi.upm.es/~dlopez/index.php
[GTI] http://www.gti.det.uvigo.es/~pedro/pub/sodtr
[LED] http://www.led.uc.edu.py/str/primera.htm
[REA] http://www.realtimelinuxfoundation.org/variants/realtime.html
[MaRTE] http://marte.unican.es/
[CTR_SO] http://www.ctr.unican.es/asignaturas/so/
[CTR_LAN] http://www.ctr.unican.es/asignaturas/lan/

[ALD01] Aldea M. y González M.: "MaRTE OS: an Ada kernel for Real-Time Embedded Aplications", International Conference on Reliable Software Technologies, Ada-Europe-2001, LNCS, May 2001.

[ALD03] Aldea M.: "Planificación de Tareas en Sistemas Operativos de Tiempo Real Estricto para Aplicaciones Empotradas", Tesis Doctoral, Dpto. Electrónica y Computadores, Grupo de Tiempo Real, Universidad de Cantabria, Mar. 2003.

[MIC03] Michael González Harbour and Mario Aldea Rivas: "Managing Multiple Execution-Time Timers from a Single Task", Proceedings of the 12th International Real-Time Ada Workshop, Viana do Castelo, Portugal, September, 2003.

Puerto Paralelo:

[MOD] http://www.modelo.edu.mx/univ/virtech/circuito/paralelo.htm
[PCH] http://www.pchardware.org/puertos/paralelo.php
[COM] http://computer.howstuffworks.com/parallel-port2.htm

Servomotores:

[IME] www.ime.eb.br/~pinho/micro/trabalhos/Robot_Bioins_I.pdf
[MID] www.midivid.com
[AUT] http://autric.com/Microbotica%20y%20Mecatronica/servomotores.htm
[JAM] http://www.jamara.de/

Desarrollo de los programas:

[ADA95] http://www.adahome.com/rm95/
[BUR95] Burn A. y Welling A.: “Concurrency in Ada”. Cambridge, 2ª Edición, 1995.

Algoritmo de control:

[ENG] http://www.engin.umich.edu/group/ctm/examples/ball/ball.html
[LAU_93] E. Laukonen and S. Yurkovich, "A Ball and Beam Testbed for Fuzzy Identification and Control Design," The 1993 American Control Conference, San Francisco, CA, June 1993.
[VAC_95] Vaccaro R.: “Digital Control. A State-Space Approach”. McGraw-Hill, 1995.
[OGA_70] Ogata K.: “Modern Control Engineering”. Prentice-Hall, 1970.
[KUO_75] Kuo B.: “Sistemas Automáticos de Control”. Prentice-Hall, 1975.

Procesado de señal:

[OPP_78] Oppenheim A.: “Aplications of digital signal processing”. Prentice-Hall, 1978.