Capítulo 4

Herramientas para la Simulación y el código asociado

4.1 Introducción

En este capítulo analizaremos las simulaciones realizadas y los instrumentos con los que éstas se realizaron. Por eso estudiaremos los sistemas que en el capítulo 2 decíamos que en realidad simulaban a sistemas reales. Además del código en C++, se elaboraron también una serie de programas en Matlab aprovechando la potencia de esta herramienta para el diseño previo del controlador LQR y para la interpretación de datos.

· Código en C++:

4.2 DSP Simulator

En realidad, en la carpeta `./hero/DSP_Simulator/` se encuentra todo el código elaborado para la simulación, incluidos los dos modelos de helicóptero, que veremos en los dos sucesivos puntos. Ahora nos centraremos en la simulación del DSP en sí mismo en lo relativo a la gestión de las comunicaciones con el PC. La función de control se simulará en verdad sólo para el modelo más real, pero se estudiará junto con dicho modelo, ya que en realidad se implementa en una clase asociada a éste.

Aunque todo lo que veamos en este capítulo queda fuera de la arquitectura general, ya que todas las simulaciones serán un paso previo a la práctica real, se ha intentado seguir el esquema de programación que hemos desarrollado a lo largo de todo el capítulo anterior. Esto se hace así por comodidad y porque estamos convencidos de que dicho esquema es correcto. Por lo tanto, seguiremos teniendo distintos hilos que se comunican por medio de una sección crítica, y clases que simplifican lo máximo posible la funcionalidad de dichos hilos.

Algunas de las clases que utilizaremos para la simulación de las comunicaciones con el PC las conocemos, precisamente del estudio anterior de su interlocutor, es decir, del Bridge Module. Este es al caso de la clase `PuertoSerie` y del template `Cola`. Además se volverá a utilizar la clase `Cparser` para cargar la configuración desde el archivo de configuración `dspsim.conf`, que determina el periodo del bucle de control en [ms] y el puerto serie a utilizar en una cadena.

El uso de `PuertoSerie` es obviamente el de interfaz para la comunicación vía puerto serie. La `Cola` emulará el funcionamiento de los buffers internos del DSP; como ya se vio resulta más eficiente que, por ejemplo, la `CList`, y en este caso es además más realista: la `Cola` se desbordará como lo podría hacer un buffer, mientras que en una `Clist` el concepto de capacidad es mucho más flexible.

En cuanto a estas clases, pensamos que se definieron con suficiente detenimiento en su momento, por lo que no las volveremos a ver en los siguientes puntos.

4.2.1 Implementación software del DSP Simulator

La implementación del DSP Simulator la lleva a cabo el proceso multi-hilo `DSPSIM`, que necesita de cinco parámetros en su ejecución. El primero vuelve a ser el código de identificación del robot (como para `BRIDGEMOD` y `STATUSMOD`); los tres siguientes dan posición inicial del robot, y el último la orientación de éste respecto al Norte. Tiene un total de 6 hilos (incluido el `main`) que iremos viendo.

Clase `CDSPSimulatorCriticalSection` (en `dspsimulatorcritical-section.h/ dspsimulatorcriticalsection.cpp`)

Una vez más, nos encontramos ante la clase que implementa la sección crítica del proceso. Como en el caso del Bridge Module, la sección crítica está subdividida en varias, con distintos mutex:

(*CS1*) Comunicación hilo de procesamiento - hilo de lectura puerto serie

Tanto los comandos como los *waypoints* provenientes de la RIL se encolarán para ir siendo tomados por el hilo de procesamiento.

(*CS2*) Comunicación hilo de procesamiento - hilo de escritura puerto serie

Para una única `Cola` que almacena los mensajes para la RIL.

(*CS3*) Comunicación hilo de procesamiento - hilo simulación robot

Que protege la coherencia de:

- Una variable con el estado del robot, proveniente del hilo de simulación. Es lógico que no se encole, lo que nos interesa es el dato más actual.

- Otra variable con el flag de control de flujo. En un principio se encolaba, con la intención de poder usar el propio flag como asentimiento de *waypoints*, pero se descartó ya que el verdadero DSP no implementa dicha función.

- Una cola con los *ack*. En la actualidad está en desuso, ya que el hilo de procesamiento genera automáticamente el asentimiento en cuanto recibe el comando, y lo coloca directamente en la *CS2*, con lo que se gana en tiempo. Por coherencia, se ha dejado esta `Cola`, que podría utilizarse si se decide hacer cambios en la simulación del DSP.

- Una cola con los *waypoints* que tomará el hilo de simulación. Esta es la `Cola` que realmente simula el buffer del DSP, ya que es su estado el que decide el valor del flag de petición de transmisión.

- Una última cola con los errores del DSP, a la que también tendrá acceso el hilo generador de errores. De hecho, actualmente los errores de DSP sólo pueden generarse en el mismo hilo de procesamiento o mediante dicho hilo de generación, ya que en el de simulación no se han considerado por el momento posibles errores, ni siquiera en su faceta de controlador.

(*CS4*) Tiempo

Sólo accesible por el hilo de simulación, la variable `dsp_time` no tendría porqué estar protegida, pero se hace por seguir con la metodología. Además, nunca se sabe si en una futura ampliación del proyecto otro hilo la podría necesitar. Su valor llega al puerto serie, pero a través de la estructura `simulated_dsp_status`, del tipo `ESTADO_HELICOPTERO`, que simula es estado del helicóptero.

(*CS5*) Puertos Virtuales

Asegura la coherencia de las variables que emulan a los puertos virtuales del DSP, por lo que lógicamente no se encolan. Por el momento se encuentran implementados los puertos virtuales correspondientes a:

- `16bits PV 10000`: Máscara de estados, que ya hemos visto.

- `16bits PV 10121`: Modo despegue. Por el momento, se interpreta el valor `1` como una petición para pasar a modo despegue. Esto sólo es válido si el helicóptero está en tierra, y en el momento en el que se ha alcanzado la altura de despegue su valor pasará de nuevo a `0`.

- `16bits PV 10122`: Modo de aterrizaje, análogo al anterior.

- `32bits PV 10119`: Velocidad asociada al *waypoint*. Llegados aquí, creemos conveniente comentar la problemática que esta variable, al estar implementada en un puerto virtual, trae asociada. El principal problema es la falta de sincronismo, es decir, que es imposible asegurar que las coordenadas de un *waypoint* y su velocidad lleguen al mismo tiempo al DSP. Por lo tanto, el DSP es incapaz de distinguir qué velocidad corresponde a qué *waypoint*, y lo único que puede hacer es suponer que el valor actual de la variable en este puerto virtual corresponde a la deseada para el punto objetivo actual. Esto no suele ser un verdadero problema cuando se envía un punto y después otro, pero la cuestión se complica para paquetes de *waypoints*: la velocidad del primer punto suele ser respetada, pero mientras que se llega a él se van cargando las velocidades sucesivas de forma que lo más probable es que para cuando nos empecemos

a dirigir a los siguientes puntos, el valor de la variable en el puerto virtual sea ya el correspondiente a la velocidad deseada para el último de los puntos.

Ante la imposibilidad de encolar las velocidades, que es lo que se pensó en un principio, para solucionar ese problema se propone que todos los *waypoints* de un mismo paquete tengan la misma velocidad asociada: es decir, hacer que la velocidad constante en todos los puntos sea una propiedad de todos los paquetes de *waypoints*.

El otro problema se presenta porque, en el caso del modelo más realista de simulador de helicóptero, el control del helicóptero no se realiza en velocidad sino en posición. Es muy posible por tanto que la velocidad deseada no sea alcanzable, a pesar de que se tenga en cuenta de alguna forma, por ejemplo dando consignas más lejanas para velocidades mayores. Además incluso en el caso del modelo primitivo, velocidades muy altas pueden llevar a la inestabilización del sistema, aunque ya se han tomado medidas que previenen de ello.

(*CS6*) Hover flag

Protege exclusivamente a la variable `hover`, que indica si el helicóptero, que se encuentra en movimiento, debe detenerse y quedarse flotando en torno a la posición actual. Esto se da, por ejemplo, cuando se aborta una tarea que consiste en ir a un punto mientras que el vehículo se está efectivamente dirigiendo a él.

Esta sección crítica vuelve a ser bastante compleja, pero puede entenderse mejor a través del esquema de la Figura 4.2-1. En dicha figura podemos ver que al "mundo exterior" del DSP Simulator sólo se puede acceder a través del puerto serie. Esto es así porque la información de salida será generada mediante simulaciones.

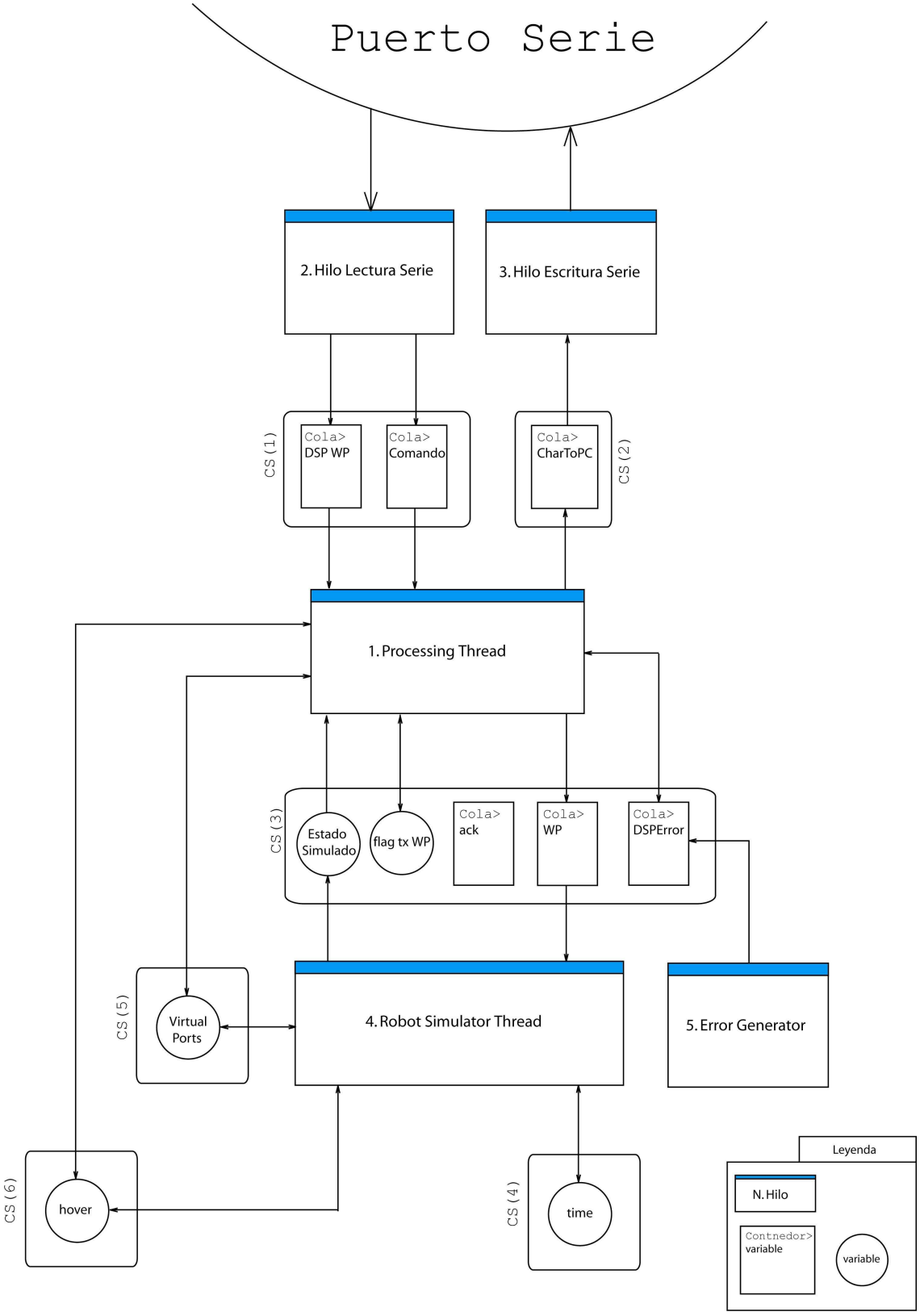

Figura 4.2-1: Esquema de hilos y secciones críticas (CS) del DSP Simulator

Clase `PCComm` (en `PCComm.h/ PCComm.cpp`)

Es la clase que implementa el protocolo de comunicaciones entre PC y DSP, esta vez visto desde el punto de vista del DSP. Se creó a partir del código que se tenía del DSP, y define las estructuras de datos y las funciones necesarias para recibir y transmitir datos al Bridge Module.

Estructuras:

- `PC_COMANDO`: Estructura que almacena un comando recibido desde el Bridge Module. Contiene información sobre el tipo de comando y los parámetros que éste trae asociados, que como vimos pueden ser de distintos tamaños.

- `PC_DATO_CAMINO_XYZ`: Estructura que almacena un punto del camino dado por el Bridge Module en coordenadas cartesianas, que finalmente fueron descartadas para la expresión de *waypoints*.

- `PC_DATO_CAMINO_GEO`: Coordenadas geográficas, que serán las realmente utilizadas. Los campos que se refieren a la altitud y a la longitud resultan ser enteros, por lo que realmente expresan diezmillonésimas de grado.

- `Comm`: Estructura que almacena el mensaje a enviar al Bridge Module, particular para esta clase.

Funciones de la interfaz:

- `codifica_estado`: Función que empaqueta el estado del vehículo en caracteres que forman un mensaje para enviar al Bridge Module. El resultado es una cadena de caracteres que quedan en un buffer interno de una estructura de tipo `Comm`. La longitud del mensaje depende de los datos que se quieran enviar, es decir, de la máscara de estados, que es enviada desde el Bridge Module.

- `codifica_mensaje_control_de_flujo`: Empaqueta un mensaje de control de flujo. Si se le pasa un `1`, el mensaje permitirá al Bridge Module la transmisión de un nuevo punto objetivo. Si se le pasa un `0`, el mensaje indica que el buffer de *waypoints* está lleno y que no se deben transmitir más por el momento. Ambos mensajes podrían actuar como asentimiento del punto anteriormente recibido, pero dicha condición se descarta, ya que valores futuros sustituirán a valores anteriores, con lo que sólo se pueden tener en cuenta cambios en el valor de la variable. Como vimos en el capítulo 3 al estudiar el protocolo desde el punto de vista del PC, tenemos que asumir que los *waypoints* no son asentidos por el DSP.

- `codifica_mensaje_de_error`: Empaqueta un mensaje con el código de error que se pasa como parámetro.

- `codifica_mensaje_de_asentimiento`: Función a la que se le pasa como parámetro el código de comando recibido y empaqueta un mensaje de asentimiento de ese comando, con los datos de respuesta asociados que le correspondan.

Todas estas funciones que empaquetan mensajes para el Bridge Module devuelve en la llamada el número de caracteres del mensaje.

- `procesa_caracter_del_PC`: Función a la que se le pasa un carácter recibido de la línea de comunicación con el Bridge Module para que lo procese. Cuando le llega el último carácter del mensaje, devuelve un `1` si es un mensaje de comando, un `2` si el mensaje es un *waypoint*. En cada caso, la decodificación del mensaje se encontrará en la variable interna del tipo `PC_COMANDO` o `PC_DATO_CAMINO_GEO` respectivamente. En cualquier otro caso, devuelve un `0`.

- `GetMensajeToPC`: Mete la información correspondiente al mensaje codificado en la variable interna de tipo `Comm` en la estructura de tipo `MessageToPC` cuya referencia se le pasa como parámetro. De esta forma, nos permite tener acceso al mensaje que el DSP Simulator acaba de codificar para el Bridge Module.

- `GetPCComando`: Este *get* nos da acceso a la variable interna que almacena el comando recibido desde el Bridge Module, ya decodificado.

- `GetPCCaminoGeo`: De igual forma, éste nos da acceso al *waypoint* expresado en coordenadas geográficas.

- `GetPCCaminoXYZ`: Ídem para coordenadas cartesianas, que en principio se descartan como sabemos.

- `SetSimulatedStatus`: Modifica el valor del estado del helicóptero, contenido en una estructura de tipo `ESTADO_HELICOPTERO`, que el protocolo codificará a continuación en un mensaje para el Bridge Module.

- `UpdateMask`: Actualiza la máscara de estados que contiene la clase, para así codificar correctamente el mensaje de estado.

Clase `DSPCommManager` (en `DSPCommManager.h/ DSPCommManager.cpp`)

Como veremos, esta clase implementa toda la funcionalidad del hilo de procesamiento con una cierta "inteligencia". Lo hace mediante cuatro funciones cuyas funciones resultan fácilmente interpretables por sus nombres:

- `ReceiveAndDoWhatCommandSays`: Toma un comando de la sección crítica *CS1* y, tras haberlo ejecutado, genera un asentimiento (*ack*) y lo mete en la sección crítica *CS2*.

- `ReceiveWayPointIfThereIsOne`: toma un *waypoint* de la sección crítica *CS1* y lo intenta meter en la *CS3*. En ese momento entra en juego el flag de petición de transmisión de *waypoints*: mientras que no se puedan meter más puntos en la `Cola` de *waypoints* de la *CS3*, el flag valdrá cero y la RIL no nos podrá mandar más. Es por eso que decimos que es esta última cola la que realmente emula al buffer que posee el DSP con este fin.

- `SendEstadoIfTxIsOn`: En caso de que la transmisión de estado esté activada, esta función toma el estado del helicóptero de la *CS3* y empaqueta un mensaje para la RIL que pasa a la *CS2*, teniendo siempre en cuenta la máscara de estados.

- `SendErrorIfThereIsOne`: En caso de que haya algún error encolado en la *CS3*, codifica un mensaje de error para la RIL y lo mete en la *CS2*.

De forma interna, esta clase realiza además la tarea de empaquetar los *waypoints* que permiten despegar y aterrizar cuando se escribe en el puerto virtual correspondiente. Dichos puntos, pasan directamente a la *CS3*.

Cabecera `DSP_error_code.h`

Contiene las definiciones de los códigos de error del DSP, que se estudiaron anteriormente y que podemos encontrar en el Cuadro 3.2-1 del capítulo 3.

Cabecera `DSP_data.h`

Contiene las definiciones de algunas estructuras que son propias del DSP Simulator. De especial interés son las estructuras `MessageToPC` y `ESTADO_HELICOPTERO`. La primera corresponde al tipo de mensaje que el hilo de escritura en puerto serie sabe interpretar y escribir correctamente. La segunda corresponde al tipo en el que el hilo de simulación, sea cual sea el modelo de helicóptero que use, generará un estado simulado del robot. Cabe destacar que esta estructura es idéntica a la que genera el DSP real, y que se ha querido por esto respetar, a pesar de que por ello se han tenido que realizar cambios de tipo y *casts* que añaden complejidad al seguimiento del código.

Archivo de configuración `dspsim.conf`

En este caso lo único que se configura es el periodo del bucle de control (es decir, el intervalo de tiempo entre dos pasos de la simulación) y la dirección del puerto serie que se quiere utilizar en la comunicación con la RIL.

Hilo principal (en `main.cpp`)

Una vez más, el hilo principal se limitará a levantar el resto de hilos. Un manejador se encargará de terminar la ejecución a la recepción de la señal `SIGINT`, generada por el usuario al pulsar Ctrl+C. La clase `CParser` servirá para cargar el puerto serie que se debe de utilizar en las comunicaciones con el Bridge Module, que se le pasará como parámetro a los hilos de lectura y escritura, y la posición y orientación inicial del robot se le pasarán al hilo de simulación dentro de una estructura de tipo `HomeParameters`.

`ProcessingThread` (en `dspsimulatorprocessingthread.cpp`)

La funcionalidad de este hilo se encuentra totalmente implementada en la clase DSPCommManager. Esto quiere decir que en el archivo `dspsimulatorprocess-ingthread.cpp` sólo encontraremos una serie de llamadas a funciones públicas de

esta clase. De forma que se reciben comandos y *waypoints*, y se envían estados y errores, si procede en cada caso.

HiloLecturaPuertoSerie (en `hilolecturapuertoserie.cpp`)

De la misma forma que lo hacía su homónimo en el Bridge Module, su misión será la de recibir por puerto serie los mensajes que este último le manda. De nuevo la clase `PuertoSerie` servirá en tal tarea, y en cada iteración se harán llamadas a `lee_caracter_puerto_serie`, que bloquea al hilo mientras que no hay caracteres, y a `procesa_caracter_del_PC`, función de la clase `PCComm`. Ésta devuelve un entero que indica qué tipo de mensaje nos ha llegado. Vemos cómo este procedimiento es realmente idéntico al que ya vimos, sólo cambian los tipos de mensaje que se pueden recibir:

`INCOMPLETE`: El mensaje está incompleto, debemos seguir iterando porque las estructuras aún no están convenientemente asignadas.

`ES_COMANDO`: Se toma el comando, ya decodificado, con la función `GetPCComando` del protocolo e inmediatamente se introduce en la sección crítica. El comando ha sido recibido.

`ES_WP`: Se hace lo propio con el *waypoint*: se toma con la función `GetPCCaminoGeo` y se mete en la sección crítica.

`ES_ERROR`: Se ha producido algún error en el procesamiento del mensaje recibido, ya sea de *checksum* o de desconocimiento de cabecera: sólo hay que tomar el mensaje de error generado por el protocolo y meterlo en la sección crítica.

HiloEscrituraPuertoSerie (en `hiloescriturapuertoserie.cpp`)

El procedimiento será igualmente análogo al caso del Bridge Module, sólo cambia la estructura que almacena los mensajes:

·1) En el caso de que haya algún `MessageToPC` en la sección crítica, lo escribe en el puerto serie carácter a carácter.

·2) En caso contrario, "duerme" un rato (20 ms).

·3) Vuelve al punto 1.

RobotSimulatorThread (en `uavsimulatorthread.cpp`)

El hilo de simulación del robot (al menos el encargado de lo que respecta a control y modelado) se limita a inicializar el tiempo local a `0`, pasarle las coordenadas iniciales al modelo de robot, y a programar un temporizador de disparo cíclico (de la clase `CHeroTimer`) según el tiempo asignado a un periodo de control en el archivo de configuración `dspsim.conf`. Cada vez que se dispare este temporizador, se realizará un paso de la simulación. Si queremos simulación en tiempo real, para el actual `UAVRealSimulator`, se debe imponer $T = 50$ms.

Después lo único que hace es mandar los datos sobre el movimiento y generar el estado correspondiente, simulado por el modelo de "controlador + helicóptero" tras cada uno de esos pasos de simulación.

Lo realmente interesante es que en este hilo se puede elegir fácilmente qué modelo queremos utilizar en nuestras simulaciones, en este caso a escoger entre el `UAVSimulator` y el `UAVRealSimulator`, clases que estudiaremos en los puntos 4.3 y 4.4 respectivamente. Esto se puede hacer así gracias a que hemos diseñado ambas clases con la misma interfaz, es decir, que las funciones públicas de ambas clases tienen exactamente el mismo prototipo (nombre y tipos de parámetros de entrada y de salida).

`ErrorGeneratorThread` (en `errorgeneratorthread.cpp`)

Con el propósito de simular posibles errores del DSP, se incluyó este hilo que, de una forma muy sencilla, los genera a voluntad del usuario. El hilo está atento a la entrada por consola, de forma que la introducción de la secuencia '`e` *`código_de_error`*' mete en la sección crítica un DSPError que se enviará al puerto serie, con lo que el Bridge Module lo reconocerá como si de un error del DSP se tratara.

Si el código numérico no está comprendido entre 1 y 300, los errores no corresponderán al DSP, pero llegarán igualmente al RIL, por lo que esta circunstancia puede ser interesante para probar otros tipos de error, aunque hay que saber qué se está haciendo. En todo caso, este hilo nos permite verificar que los errores efectivamente se traducen y trascienden de forma correcta.

4.3 UAV Simulator

Este modelo de simulación corresponde al sistema que vimos en el punto 2.6.2, y que denominamos entonces como simulador primitivo. Esto es así porque se trata de un simulador cuya única misión fue la de probar el software en fases iniciales del presente proyecto. Sin embargo no se ha eliminado, ya que puede ser útil si, por nuevos cambios en el software, es necesario volver a hacer pruebas en las que no interesa complicar el análisis con un modelo más real.

Este modelo utiliza como sistema aquel definido por el Control Center: el eje y está orientado hacia el norte y el x hacia el este, como en los robots terrestres, y la tercera dimensión z, orientada hacia el cielo como en el WCS. Nuestro sistema sólo tendrá posición (x, y, z), orientación respecto al Norte (θ) y velocidad lineal referida a los planos horizontal y vertical (v_h, v_v).

Un ejemplo de lo primitivo que resulta es el hecho de que las unidades no nos preocupan. Es decir, a excepción de los ángulos que deben estar en radianes para los cálculos, el resto de variables pueden venir expresadas en las unidades que sean, siempre que éstas sean iguales y consecuentes: es decir si la unidad de longitud es el metro y la de tiempo el segundo, la posición estará expresada en [m] y la velocidad en [m/s]. Si por el contrario la unidad de longitud es la pulgada y la de tiempo el periodo del bucle de control, cada una de las coordenadas vendrá expresada en [pulgadas] y las velocidades en [pulgadas/periodo].

La orientación cambia instantáneamente, y el control de todo el movimiento se realiza de una forma sencilla: Imaginemos que nos encontramos en el punto A (x_1, y_1, z_1) y recibimos como objetivo el B (x_2, y_2, z_2). El típico problema $A \rightarrow B$.

Para simular la dinámica, se calculan las distancias en el plano vertical (s_v) y en el plano horizontal (s_h), funciones del punto en el que nos encontramos y el punto objetivo:

$$\left.\begin{aligned} x_{diff} &= x_2 - x_1 \\ y_{diff} &= y_2 - y_1 \\ z_{diff} &= z_2 - z_1 \end{aligned}\right\} \Rightarrow \begin{cases} s_v = z_{diff} \\ s_h = \sqrt{x_{diff}^{\ 2} + y_{diff}^{\ 2}} \end{cases}$$

A partir de estas distancias se podría calcular la pendiente de la recta que une los puntos en el plano vertical que la contiene. El ángulo que forma dicha recta con el plano horizontal (que llamaremos α, pero ojo no confundir con el ángulo de ataque porque no tiene nada que ver) sería el arco cuya tangente es igual a dicha pendiente. De manera análoga, el ángulo que conoceremos como orientación (θ) del helicóptero respecto a la dirección Norte (coincidente con el eje y) será función de x_{diff} e y_{diff}.

$$\alpha = \arctan \frac{s_v}{s_h}$$

$$\theta = -\arctan \frac{x_{diff}}{y_{diff}}$$

Donde es necesario corregir el signo de la orientación ya que se define positiva en sentido antihorario.

Finalmente, con α podemos descomponer la velocidad que recibimos como dato junto con el *waypoint* (v) en velocidad horizontal (v_h) y velocidad vertical (v_v)

$$\begin{cases} v_h = v \cos \alpha \\ v_v = v \sin \alpha \end{cases}$$

Y con las velocidades ya podemos actualizar la posición del robot en cada iteración del bucle de control (cada Δt):

$$\begin{cases} x \leftarrow x - \dfrac{v_h}{\Delta t} \sin(\theta) \\ y \leftarrow y + \dfrac{v_h}{\Delta t} \cos(\theta) \\ z \leftarrow z + \dfrac{v_v}{\Delta t} \end{cases}$$

Y por último, para comprobar si hemos llegado al objetivo, basta con generar un nuevo cubo de tolerancias y verificar que nos encontramos dentro de él.

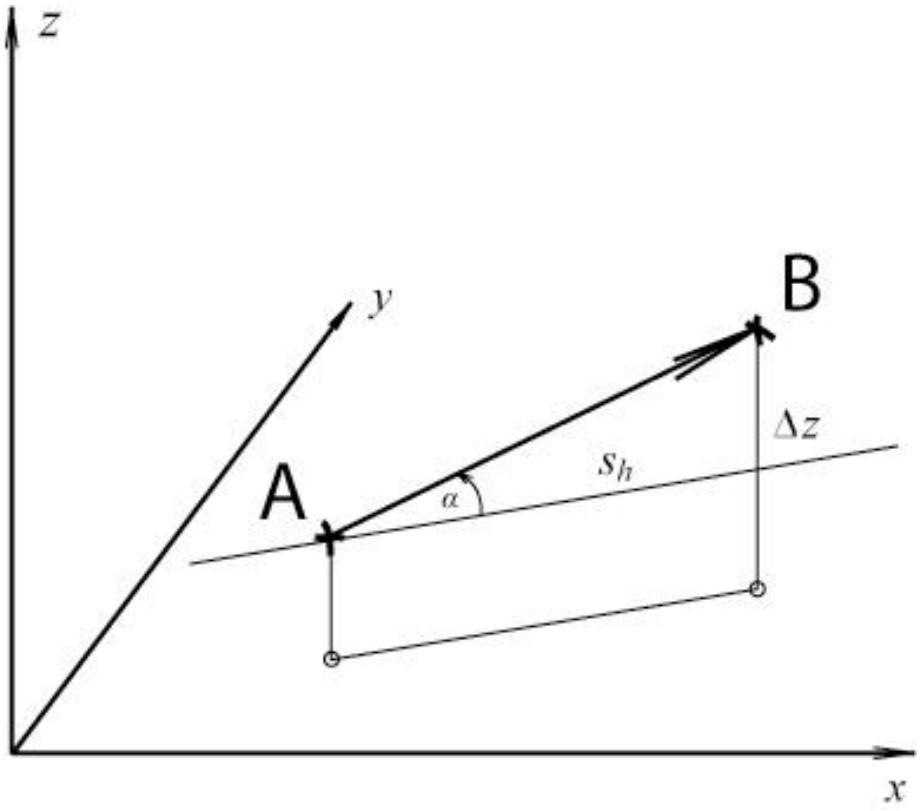

Figura 4.3-1: Trigonometría del problema en el espacio

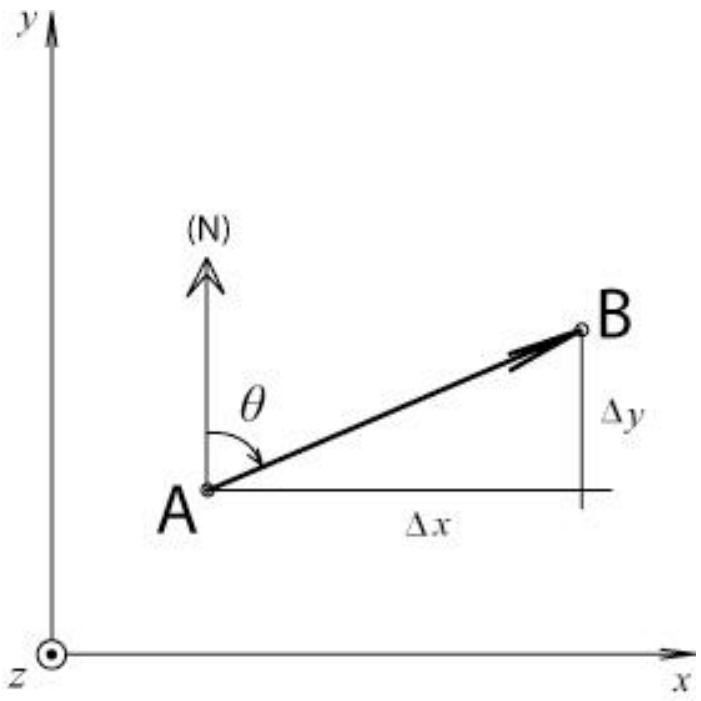

Figura 4.3-2: Trigonometría del problema en el plano

4.3.1 Implementación software del UAV Simulator

Si definimos `MODEL_0` para el hilo de simulación, éste utilizará el modelo primitivo de UAV, es decir, el `UAVSimulator`.

Clase `UAVSimulator` (en `uavsimulator.h/ uavsimulator.cpp`)

Implementa la funcionalidad del hilo de simulación en el caso de ser elegida para ello. Posee otro constructor además del *default* (sin parámetros de entrada) que permite inicializar la posición del helicóptero en base a una variable de tipo `HomeParameters`. De todas maneras, existe también otra función que puede hacer esto en cualquier otro momento que no sea el de construcción (llamada al constructor). De esta forma el punto de partida y la orientación inicial del helicóptero simulado serán los correspondientes a los valores que se le pasan al ejecutable `DSPSIM`.

El resto de funciones relevantes de la clase:

`bool ReceiveMovementData(void)`: Toma un *waypoint* de la sección crítica y lo mete dentro de una variable interna que indicará al modelo el próximo punto al que debe de dirigirse. Además lee la velocidad a la que debe hacerlo de la misma sección crítica, en este caso de la parte que simula a los puertos virtuales del DSP. Vemos claro que es necesaria la introducción de la sección crítica del proceso en esta clase como variable externa.

`void SimulatedDynamic(bool hover)`: Como su propio nombre indica, esta es la función que simula la dinámica del UAV, aunque en realidad lo que hace son llamadas a las funciones protegidas (por tanto no pertenecientes a la interfaz) `ControlBlockCalculateActuation`, `ModelBlockUpdatePosition` y `DotheRobotArriveToWP` en caso de que queramos dirigirnos a otro punto. Aquí es donde entra la variable `hover`, que detiene el helicóptero (impone velocidades nulas) si, por ejemplo, se ha abortado la misión de ir a dicho punto.

`void ControlBlockCalculateActuation(void)`: Realiza la función de bloque de control, o lo que es lo mismo, de controlador. En este caso, para el modelo primitivo de helicóptero, el control es de lo más sencillo: se actualiza la orientación de forma instantánea y se asignan las velocidades en función de la posición del próximo punto objetivo, tal y como vimos al principio de esta sección. La única precaución que se toma es la de frenar progresivamente el vehículo si la velocidad es demasiado elevada en el entorno de nuestro destino, para evitar la inestabilización.

`void ModelBlockUpdatePosition(void)`: Implementa el bloque de modelado, que se limita a actualizar la posición en función de las velocidades impuestas por la función anterior. Esto es así porque, como hemos repetido hasta la saciedad, el modelo empleado en este caso es extremadamente simple. Tanto esta como la anterior función serán las que marquen la gran diferencia, como veremos, entre esta clase y el `UAVRealSimulator`.

Cabe destacar además la actualización de la variable `flag_volando` perteneciente al estado simulado. Esta sólo tomará el valor `FLYING` tras haber realizado un despegue completo, y no volverá a `NOT_FLYING` hasta que se complete el

aterrizaje. Por despegue completo entendemos aquel que alcanza la altura mínima establecida.

`void DotheRobotArriveToWP(void)`: Calcula si el robot se encuentra ya dentro del cubo definido en torno a nuestro objetivo, y actualiza el flag `lastWaypointReached` en consecuencia.

`void SendSensorStatus(void)`: Mete en la sección crítica el estado simulado, que ya ha sido actualizado en la llamada a `ModelBlockUpdatePosition`.

`bool HaveWeArrivedToLastWaypoint(void)`: Devuelve el valor del flag `lastWaypointReached` actualizado en la llamada a `DotheRobotArriveToWP`, con lo que indica si el robot ha alcanzado o no el último *waypoint* recibido.

`void ForDebugging(void)`: Realiza una salida por consola con los valores más importantes del estado simulado, por lo que se utiliza para ver si el movimiento es correcto en las fases de prueba.

4.4 UAV Real Simulator

El gran avance de este simulador respecto al anterior lo supone la integración de unos bloques de modelado y de control mucho más realistas. Ambos fueron consecuencia de que al fin se disponía de un modelo del helicóptero: obviamente se facilitaba el correcto modelado, pero también ahora se podía empezar a diseñar un control más serio.

El modelo físico del helicóptero al que se llegó, como veremos, describe el comportamiento de nuestro sistema en el espacio de estados de tiempo discreto. Esto es así porque el punto de partida fue el código de un simulador que los alemanes de la TUB tenían para su helicóptero MARVIN.

En dicho código se utilizan una serie de ecuaciones básicas que responden a simplificaciones en la física de vuelo del helicóptero. A continuación se exponen en poco espacio las ecuaciones que nos servirán para el modelado.

Sean las entradas las señales para los servos:

spc: colectivo
spx: cíclico en torno al eje *x* del HCS
spy: cíclico en torno al eje *y* del HCS
sph: rotor de cola
sth: gas

Que se encontrarán debidamente acotadas como veremos. Sea la salida un estado del helicóptero compuesto por las variables:

x, y, z: posición en WCS, expresada en [mm]
v_x, v_y, v_z: velocidad en WCS, expresada en [mm/s]
B_x, B_y, B_z: orientación en WCS, expresada en [pasos/ W_{360}]*
W_x, W_y, W_z: velocidades de giro, referidas al HCS, en [pasos/ W_{360}/s]
W_r: velocidad de giro del rotor principal, en [r.p.m = revoluciones/min]

(*) Una circunferencia completa en este código se dividirá en W_{360} pasos iguales. En el código actual, esta constante $W_{360} = 12^{12} = 4096$, pero la resolución puede cambiarse a placer variando simplemente el valor de esta constante.

Las entradas ya las conocemos de cuando estudiamos los sistemas de nuestro proyecto en el capítulo 2. Vemos que en nuestro helicóptero los cinco actuadores relativos al movimiento se encuentran implementados por servomotores. Estos se suelen controlar por trenes de pulsos de anchura variable, por lo que los valores de las señales de entrada serán proporcionales al ancho de los pulsos de forma que, por ejemplo, los valores extremos de señal correspondan a posiciones extremas de los servos. Esto en realidad no debe preocuparnos, ya que en todo caso se tratará de un simple escalado. Lo importante es saber que los valores de la señal regulan el valor de la actuación de forma proporcional.

Ahora además entendemos por qué el cíclico se divide en dos señales distintas: una variación en el *cíclico x* hará que plato se incline hacia un lado, con lo que el helicóptero se moverá lateralmente. Una variación en el *cíclico y* inclinará el plato hacia adelante o atrás, con lo que el helicóptero seguirá tal tendencia debido a la variación del empuje en el giro de las palas. Cualquier otra dirección puede conseguirse mediante la combinación de ambas entradas.

También en el capítulo 2 estudiamos los sistemas de referencia WCS y HCS. Ahora veremos la simplificación que permite obtener una relación mucho más sencilla entre ambos sistemas: siendo el modelo sólo válido para puntos en torno al *hover* (lógico punto de equilibrio), podemos suponer que los ejes *z* permanecen siempre paralelos. En consecuencia, lo único que diferencia al HCS del WCS es una traslación del origen (correspondiente a la posición del helicóptero) y una rotación respecto al eje *z* (que corresponde a la orientación con respecto al Norte del vehículo).

El código utilizaba las siguientes ecuaciones, que se han dejado en función de los parámetros k_i, E_i y T, que serán constantes que veremos, para el cálculo de los incrementos de las variables de estado en función de valores anteriores y de las entradas:

$$\Delta W_r = k_1 \cdot sth - k_2 \left(\frac{W_r}{E_1} \right)^2 \cdot spc$$

Ec 4.4.1

El incremento de revoluciones del rotor principal depende, por un lado, del gas, con el que es directamente proporcional (ganancia k_1). Por otro lado, un mayor valor para el colectivo hace aumentar la resistencia con el aire, que es función de la velocidad al cuadrado para flujos turbulentos (y por tanto de la misma W_r al cuadrado). Lógicamente, la resistencia se resta, pues tiende a frenar el giro del rotor. El parámetro E_1 es igual a las r.p.m del rotor en un vuelo normal.

$$\Delta v_z = \frac{spc}{\max(spc)} E_3 \cdot g\,T \cdot \frac{W_r}{E_2} - g\,T$$

Ec 4.4.2

La variación de velocidad en el eje *z* será la integración de la aceleración en dicho eje. Esta aceleración está causada por la actuación de una serie de fuerzas sobre el helicóptero. Hacia arriba, consideramos que actúa una fuerza proporcional a un producto entre los valores de colectivo y de velocidad de giro del rotor. Hacia abajo (signo negativo) actúa la gravedad. Para la posición de *hover*, las fuerzas se anulan, la aceleración resultante es cero y el helicóptero no incrementa su velocidad. E_2 son las r.p.m del motor durante la fase de aceleración.

Aquí vemos ya el significado de T: es el paso de integración, es decir, el intervalo de tiempo que trascurre entre dos llamadas a la función de simulación ($T = \Delta t$). Actualmente el código posee una $T = 50$ms.

$$\Delta z = v_z \cdot T$$

Ec 4.4.3

La posición será a su vez la integración de la velocidad en el tiempo.

$$\Delta W_x = spx \cdot \frac{E_{4x}}{E_{5x}}$$

$$\Delta W_y = spy \cdot \frac{E_{4y}}{E_{5y}}$$

Ec 4.4.4a, 4.4.4b

Suponemos que las velocidades angulares en torno a los ejes *x* e *y* del HCS son proporcionales a los valores del cíclico *x* e *y* respectivamente. De hecho el cociente E_4/E_5 es una relación empírica entre valores de cíclico y velocidad angular adquirida. Por ejemplo, si para *spx*=500 se obtiene una velocidad angular W_x de 1024 pasos/s (es decir 90º/s), se tendrá que E_{4x}=1024 y E_{5x}=500.

$$\Delta W_z = \left(\frac{W_r}{E_6} \cdot \frac{spc - E_7}{k_3 \cdot \max(spc)} + \frac{sph - E_8}{\max(sph)} \right) \cdot E_9 \cdot W_{360} \cdot T$$

Ec 4.4.5

En la rotación en torno al eje *z* del HCS, entran en juego otros factores. La velocidad de giro del rotor (W_r) en combinación con el cíclico, generan un momento en torno a este eje que deberá de ser compensado por la actuación del rotor de cola. Los valores de E_6, E_7 y E_8 son los de las actuaciones W_r, *spc* y *sph* respectivamente durante un hover perfecto. Es decir, son tres de las componentes de u^h como veremos.

$$\left.\begin{array}{l} z = 0 \\ v_z = 0 \\ W_x = 0 \\ W_y = 0 \\ W_z = 0 \end{array}\right\} \; para \; z \le 0$$

Ec 4.4.6

Es una simple simulación del suelo: cuando el helicóptero llegue al suelo, *z* no puede ser negativa (sino cero), y las velocidades se hacen nulas. En la realidad, la proximidad del suelo produce además un cambio en el comportamiento del vuelo que se conoce como efecto suelo o efecto tierra.

$$\Delta B_x^{\{HCS\}} = W_x \cdot T$$
$$\Delta B_y^{\{HCS\}} = W_y \cdot T$$

Ec 4.4.7a, 4.4.7b

Un incremento del ángulo de *roll* y *pitch* referidos al sistema HCS del instante anterior resulta de la integración de un paso en t de las velocidades angulares W_x y W_y, que como vimos estaban referidas a ese mismo sistema de coordenadas.

$$\Delta B_z = W_z \cdot T$$

Ec 4.4.8

Para el *yaw* sucede lo mismo, sólo que no hará falta un cambio al WCS, ya que se considera despreciable cualquier rotación que no sea respecto al propio eje z.

$$\begin{cases} B_x = B_y^{\{HCS\}} \cdot \sin\left(W_z \cdot T \cdot \dfrac{2\pi}{W_{360}}\right) + B_x^{\{HCS\}} \cdot \cos\left(W_z \cdot T \cdot \dfrac{2\pi}{W_{360}}\right) \\ B_y = B_y^{\{HCS\}} \cdot \cos\left(W_z \cdot T \cdot \dfrac{2\pi}{W_{360}}\right) - B_x^{\{HCS\}} \cdot \sin\left(W_z \cdot T \cdot \dfrac{2\pi}{W_{360}}\right) \end{cases}$$

Ec 4.4.9

Ahora se realiza una especie de cambio de coordenadas de los ángulos anteriormente calculados. Dicho cambio corresponde a una rotación en el plano xy de un ángulo igual al ΔB_z. Esto es una estimación que puede realizarse gracias a que se espera que los ángulos sean muy pequeños.

$$a_x^{\{HCS\}} = \sin(B_y) \cdot g \approx 2\pi \frac{B_y}{W_{360}} \cdot g$$

Ec 4.4.10

La inclinación del helicóptero respecto al eje y del HCS provoca una aceleración en el eje x. Si suponemos que la aceleración total generada por el rotor es igual a la de la gravedad (lo que corresponde a un *hover* y aproxima el caso de una pequeña inclinación), para obtener la correspondiente al eje basta proyectar g según la inclinación. Por último, ya que suponemos que los ángulos son muy pequeños, podemos además aproximar el seno al ángulo.

$$a_y^{\{HCS\}} = -\sin(B_x)\, g - \frac{sph - k_4 \cdot \max(sph)}{k_4 \cdot \max(sph)} \frac{g}{E_{10}} \approx -2\pi \frac{B_x}{W_{360}} \cdot g - \frac{sph - k_4 \cdot \max(sph)}{k_4 \cdot \max(sph)} \cdot \frac{g}{E_{10}}$$

Ec 4.4.11

Y viceversa, una inclinación del helicóptero respecto al eje x del HCS provoca una aceleración en el eje y. Pero en este eje entra además en juego el rotor de cola, que

realiza un empuje lateral. El valor máximo estimado para la aceleración provocada por el rotor es de $1/E_{10}$ la de la gravedad.

$$\begin{cases} a_x^{\{WCS\}} = a_x^{\{HCS\}} \cdot \cos\left(B_z \cdot \frac{2\pi}{W_{360}}\right) - a_y^{\{HCS\}} \cdot \sin\left(B_z \cdot \frac{2\pi}{W_{360}}\right) \\ a_y^{\{WCS\}} = a_x^{\{HCS\}} \cdot \sin\left(B_z \cdot \frac{2\pi}{W_{360}}\right) + a_y^{\{HCS\}} \cdot \cos\left(B_z \cdot \frac{2\pi}{W_{360}}\right) \end{cases}$$

Ec 4.4.12

Con las aceleraciones es necesario hacer un cambio HCS → WCS, en el que supondremos los valores de B_x y B_y despreciables. Por eso se utiliza directamente el valor de B_z, que es la orientación del HCS respecto al WCS.

$$\Delta v_x = a_x^{\{WCS\}} \cdot T$$
$$\Delta v_y = a_y^{\{WCS\}} \cdot T$$

Ec 4.4.13a, 4.4.13b

Una vez que tenemos las aceleraciones referidas al WCS, el cálculo de las velocidades en este mismo sistema es inmediato.

$$\left.\begin{matrix} v_x = 0 \\ v_y = 0 \end{matrix}\right\} \; para \; z \le 0$$

Ec 4.4.14

Como segunda parte de la simulación del suelo, hay que poner a cero las velocidades recién calculadas si estamos en tierra.

$$\Delta x = v_x \cdot T$$
$$\Delta y = v_y \cdot T$$

Ec 4.4.15a, Ec 4.4.15b

Y al fin obtenemos la posición horizontal del helicóptero a partir de las velocidades.

El valor de las constantes k_i, E_i dependerá de varios aspectos. Principalmente, del vehículo que se quiere simular en particular: peso, envergadura del rotor, potencia del motor... Inicialmente utilizaremos los valores correspondientes al MARVIN con algunos retoques. También sería posible calcular los parámetros que identifican al HERO a través de una serie de experimentos.

Estas ecuaciones realizan una integración del estado en cada iteración, en la que el orden en que se realizan las operaciones es importante, ya que conforme se avanza en los cálculos se van utilizando los valores más actuales de las variables. Esto hace que nuestra descripción en el espacio de estados sea bastante particular y compleja, si queremos que el modelo en el espacio de estados responda realmente al código. Quizá por ello quizá hubiera sido más lógico crear primero el modelo y después el código.

Las ecuaciones equivalentes a las 4.4.1-4.4.15 y que responden a la descripción:

$$\begin{cases} x^{k+1} = f\left(x^k, u^k\right) \\ y^k = g\left(x^k, u^k\right) \end{cases}$$

se encuentran en el Anexo II (Ecs II.1). La *k* corresponde al instante de muestreo y las variables:

$$\begin{array}{lll} u_1 = spc & x_1 = x & y_1 = x \\ u_2 = spx & x_2 = y & y_2 = y \\ u_3 = spy & x_3 = z & y_3 = z \\ u_4 = sph & x_4 = v_x & y_4 = v_x \\ u_5 = sth & x_5 = v_y & y_5 = v_y \\ & x_6 = v_z & y_6 = v_z \\ & x_7 = B_x^{\{HCS\}} & y_7 = B_x \\ & x_8 = B_y^{\{HCS\}} & y_8 = B_y \\ & x_9 = B_z & y_9 = B_z \\ & & y_{10} = W_x \\ & & y_{11} = W_y \\ & x_{12} = W_z & y_{12} = W_z \\ & x_{13} = W_r & y_{13} = W_r \end{array}$$

Esta descripción se ha realizado minimizando los estados, es decir, tomando por estados sólo aquellas variables del problema que dependen de valores propios en instantes anteriores. Así, en realidad los estados x_{10} y x_{11} no existen, ya que ni W_x ni W_y dependen de valores pasados. La nomenclatura se ha mantenido así por comodidad, ya que como vemos las salidas prácticamente son idénticas a los estados. Haciendo recuento, tenemos 5 entradas, 11 estados y 13 salidas.

Para que este sistema que acabamos de definir obedezca las órdenes del Control Center, habrá que implementar un controlador que dé valores a las señales de entrada *u* de forma que el vehículo, por ejemplo, se dirija hacia un *waypoint*.

Para el UAV Real Simulator se ha elegido implementar un control óptimo LQR de horizonte infinito, que para un sistema descrito por:

$$x^{k+1} = Gx^k + Hu^k$$

define una ley de control de realimentación de estados:

$$u^k = -Kx^k$$

que minimiza un índice de bondad del control J expresado como:

$$J = \frac{1}{2}\sum_{k=0}^{\infty}\left((x^k)^T Q x^k + (u^k)^T R u^k\right)$$

Q y R son matrices de ponderación, a las que únicamente se les impone la restricción:

$$Q = Q^T > 0$$
$$R = R^T > 0$$

En definitiva, el problema a resolver es:

$$K = \arg\min_k J$$

que implica la resolución de una ecuación de Ricatti, lo cual no es excesivamente difícil si de dispone de un ordenador (cálculo iterativo) pero que es aún más fácil si se cuenta con Matlab, como en nuestro caso.

Para poder calcular la ganancia K, es necesario por tanto tener un modelo lineal del sistema. Nuestro sistema no lo es, y por lo tanto se pasó a realizar una linealización del mismo. El punto de equilibrio que más nos interesa es el correspondiente al helicóptero flotando sin movimiento, y éste será en torno al cual linealicemos: el *hover* perfecto, del cual se nos ha dado ya algunas pistas con los valores de los parámetros E_6, E_7, E_8. Pero es la condición de equilibrio ($\partial / \partial t = 0$) para todas las variables, la que determina las ecuaciones del hover matemático, que se presentan en el anexo II (Ecs II.2). La mayoría de sus resultados son del todo intuitivos: las velocidades han de ser nulas, así como los ángulos en x e y (el helicóptero ha de estar totalmente horizontal). Las actuaciones *spc* y *sph* vienen determinadas por E_7, E_8 y además se nos dice que W_r^h vale E_6. La posición y la orientación con respecto al norte son libres.

Otros valores para E_i y todos los k_i se nos dan como dato para el caso del MARVIN, pero el *hover* matemático resulta más caprichoso en algunos casos, y hay que jugar con los valores de los parámetros para que las cosas encajen. La documentación no es suficiente para saber qué sucedía en el caso del MARVIN, pero nosotros sí impondremos estabilidad en torno al punto de *hover* (h).

De las ecuaciones primera y sexta de Ecs II.2.1 y de la segunda de Ecs II.2.3:

$$u_5^h = \frac{k_2}{k_1} \cdot \left(\frac{x_{13}^h}{E_1} \right)^2 u_1^h \rightarrow \left\{ \begin{matrix} x_{13}^h = E_6 \\ u_1^h = E_7 \end{matrix} \right\} \rightarrow u_5^h = \frac{k_2}{k_1} \cdot \left(\frac{E_6}{E_1} \right)^2 E_7$$

$$\frac{u_1^h}{\max(u_1)} \cdot E_3 \cdot \frac{x_{13}^h}{E_2} = 1 \rightarrow \left\{ \begin{matrix} x_{13}^h = E_6 \\ u_1^h = E_7 \end{matrix} \right\} \rightarrow E_2 = \frac{E_7}{\max(u_1)} \cdot E_3 \cdot E_6 \quad (\neq 1050)$$

$$u_4^h - k_4 \max(u_4) = 0 \rightarrow \left\{ u_4^h = E_8 \right\} \rightarrow k_4 = \frac{E_8}{\max(u_4)} \quad (\neq 2.7)$$

Tras estas pequeñas correcciones sobre los valores anteriores de los parámetros, ya tenemos los valores numéricos con los que actualmente se están realizando las simulaciones, aunque como veremos se podrían redefinir una vez más para ajustar más el modelo al comportamiento del HERO.

$k_1 = 1/50$	$k_2 = 1/70$
$k_3 = 2$	$k_4 = 0.36$
$E_1 = 1100$	$E_2 = 806.4$
$E_3 = 1.5$	$E_{4x} = 1024 = E_{4y}$
$E_{5x} = 500 = E_{5y}$	$E_6 = 1120$
$E_7 = 720$	$E_8 = 360$
$E_9 = 4.8$	$E_{10} = 10$

El tomar como punto de equilibrio el *hover* nos obligará a realizar desplazamientos lentos, para no salir de él y perder todo parecido al sistema real. Esto en realidad no es ningún inconveniente, puesto que nos interesa un control suave más que veloz. Este hecho se puede además compensar cambiando los pesos Q y R del control.

En definitiva nuestro sistema linealizado quedaría:

$$\left\langle \begin{array}{l} \delta u^k = u^k - u^h \\ \delta x^k = x^k - x^h \\ \delta y^k = y^k - y^h \end{array} \right\rangle \rightarrow \begin{cases} \delta x^{k+1} = A\delta x^k + B\delta u^h \\ \delta y^k = C\delta x^k + D\delta u^h \end{cases}$$

$$A_{i,j} = \left.\frac{\partial f_i}{\partial x_j^k}\right|_h \qquad B_{i,j} = \left.\frac{\partial f_i}{\partial u_j^k}\right|_h$$

$$C_{i,j} = \left.\frac{\partial g_i}{\partial x_j^k}\right|_h \qquad D_{i,j} = \left.\frac{\partial g_i}{\partial u_j^k}\right|_h$$

Ec 4.4.16

donde se ha realizado un cambio que hace nulas todas las variables para el punto de equilibrio, cuyo valor nos es ya matemáticamente conocido.

Finalmente la ley de control para el sistema real, según el control LQR:

$$\delta u^k = -K \cdot \delta x^k \quad \Rightarrow \quad u^k = u^h - K \cdot \left(x^k - x^h\right)$$

Las derivadas parciales en el punto de *hover* (*h*) se encuentran en el Anexo II (Ecs II.3). El hecho de que los valores de tales derivadas dependieran de la posición del punto en que se realiza sería una tragedia, ya que se tendrían controladores distintos para cada uno de los puntos del espacio. Intuitivamente, esto no debe ser así, ya que el Sistema de Referencia es en realidad algo arbitrario, y para hacer un *hover* no debe importar si nos encontramos, por ejemplo, a 5 o a 10 metros de altura. Esta intuición la corroboran nuestras ecuaciones, en las que no se encuentra ninguna dependencia respecto a x_1^h, x_2^h ó x_3^h. Pero no se nos debe escapar que en esas mismas ecuaciones sí se encuentran ciertas dependencias con la orientación (x_9^h), en concreto para las velocidades v_x, v_y, ya que $f_9(x^h,u^h) = x_9^h$. Si nos paramos a pensar, esto es totalmente lógico, ya que tras las suposiciones respecto a los SR, sólo x_9 describe el giro de HCS respecto a WCS. Este giro hace que un cambio de coordenadas sólo afecte a las magnitudes en el plano *xy*. Este cambio afecta por tanto a las aceleraciones, a las velocidades y definitivamente a la posición (*x*,*y*). Sin embargo el cambio de coordenadas que se realiza con el valor del ángulo $W_z \cdot T$ no nos afecta, ya que en un *hover* $W_z = 0$. Estaba claro, las matrices del sistema linealizado, y en consecuencia la ganancia *K* de control, dependían de la orientación con respecto al Norte del helicóptero.

Inicialmente se pensó en resetear los estados (que no las salidas) de forma que el modelo pensara estar siempre orientado hacia el Norte, y así utilizar únicamente la ganancia correspondiente $x_9^h = 0$. Todo esto resultaba poco intuitivo, y no se consiguió ningún resultado aceptable.

Finalmente nos planteamos un nuevo objetivo: la obtención de la función κ que nos dé el valor de la ganancia K en función de la orientación del vehículo, es decir:

$$\kappa : Z \rightarrow R^{[5\times 11]} \mid \quad K = \kappa(x_9)$$

En realidad esta dependencia no es tan trágica, ya que x_9, es decir, la orientación de nuestro helicóptero, se normalizará en todo momento entre $(-W_{360}/2, W_{360}/2]$. Por lo tanto sólo puede tomar W_{360} valores distintos, lo cual supone al menos un conjunto finito. Además no todos los 55 elementos de K serán variables con x_9, la lógica nos dice que sólo aquellas que correspondan a las velocidades y posiciones lineales en el plano xy lo serán. Y para simplificar más las cosas, la mitad de los valores de las variables se pueden calcular fácilmente a partir de la otra mitad ya que estos elementos presentan simetría, ya sea:

- respecto al origen: $\kappa(-x_9) = -\kappa(x_9)$

- respecto al eje de ordenadas: $\kappa(-x_9) = \kappa(x_9)$

lo cual es de toda lógica, ya que el signo de la orientación es totalmente arbitrario. Todo esto lo podemos ver en las figuras (Fig III.1) que se adjuntan en el Anexo III. Se observa además cómo, tal y como se esperaba, los valores iniciales y finales dentro del intervalo $(-W_{360}/2, W_{360}/2]$ son iguales, ya que se ha completado una vuelta: si se hubiera prolongado el intervalo, la función sería lógicamente periódica, por lo que normalizando no alteramos el resultado. En las figuras del Anexo III sólo se presentan las $K_{i,j}$ variables con x_9 para unos valores concretos de los pesos Q y R. Al final todo encaja: se tienen variables las ganancias asociadas a x ($K_{i,1}$), y ($K_{i,2}$), v_x ($K_{i,4}$) y v_y ($K_{i,5}$), lo que hace un total de $(2+2)\cdot 5 = 20$ $K_{i,j}$ variables. El resto serán constantes que basta calcular una vez.

El control efectivo del sistema, se realizará dando la consigna en x^h. Es decir, si deseamos que el helicóptero cambie su estado a un x_d (con posición y orientación distintos al actual), basta imponer $x^h = x_d$ para que el sistema se crea fuera del punto de equilibrio y la realimentación de estados se encargue de llevarlo a él. Esto en verdad sólo será posible si x_d está próximo a x^h, ya que es en el entorno de x^h en el que es válida la linealización y por tanto el control LQR calculado.

4.4.1 Implementación software del UAV Real Simulator

Si definimos `MODEL_1` para el hilo de simulación, éste utilizará el modelo primitivo de UAV, es decir, el `UAVRealSimulator`.

Clase `UAVRealSimulator` (en `uavrealsimulator.h`/ `uavrealsimulator.cpp`)

La ganancia K del controlador LQR de declara como variable global. Los valores de cada una de sus componentes dependen exclusivamente del archivo de cabecera `Kis.h`, que se genera mediante un programa de Matlab que veremos. En particular, aquellos valores que dependen de la orientación, serán calculados en cada iteración del control con la función `kappa`, definida en `kappa.h`.

Constructor y destructor desempeñan exactamente las mismas funciones en el caso del `UAVSimulator`. Lo mismo sucede con el resto de las funciones de la clase, con las siguientes salvedades:

En la función `ControlBlockCalculateActuation` lo que se hace ahora es calcular una consigna para cambiar el estado del *hover* (x^h). Esto hará que la actuación, calculada como $u^k = u^h - K\,(x^k - x^h)$, tienda a llevar al helicóptero al estado x^h. Esta consigna se prepara de forma que la diferencia con el estado actual no sea demasiado grande, para evitar brusquedades, de forma que la aproximación al verdadero punto objetivo se hará poco a poco. Además, el trayecto hacia un *waypoint* se hace en dos fases:

1) Primero se llega a la orientación adecuada y a la altura del objetivo.

2) Y después se avanza en el plano horizontal, dando consignas en x e y, mientras se corrige la orientación si es necesario.

Todo esto se puede en realidad interpretar como un pequeño e improvisado generador de trayectorias. Finalmente se hace una llamada al ala función protegida `CalculateU`, que calcula el valor de u^k.

`void CalculateU(void)`: Realiza el cálculo de la K para la orientación actual, y genera $u^k = u^h - K\,(x^k - x^h)$ que después satura para que la señal final esté comprendida entre los valores límites. La única precaución que hay que tomar es la de normalizar la diferencia $(x^k - x^h)$ para el caso concreto x_9. Esto se hace así para describir siempre el menor giro posible para cambiar la orientación. Por ejemplo, sería absurdo que para pasar de una orientación $(\pi - \delta)$ a una $(-\pi + \delta)$, con $(\delta > 0,\ \delta \sim 0)$ se diera un giro completo en lugar de un giro de 2δ.

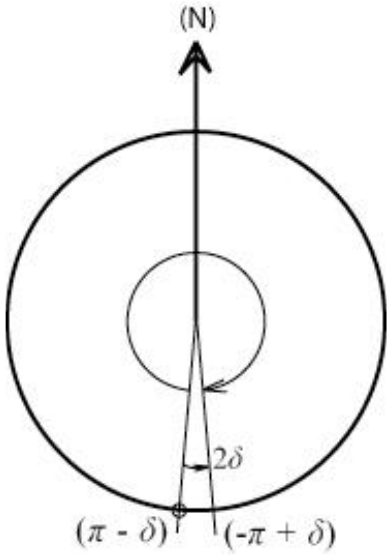

Figura 4.4-1: Ejemplo de la utilidad de normalizar `o_diff`

En `ModelBlockUpdatePosition` la única diferencia está en que lo que se hace es una llamada a la función `simstep` de la clase `HelicopterModel` para generar el estado simulado actual, en lugar de hacerlo directamente imponiendo las velocidades. Esta función realiza un paso en la simulación del helicóptero implementado en dicha clase con las entradas *u* calculadas anteriormente.

La función `DotheRobotArriveToWP` deberá tener ahora en cuenta los dos pasos en los que se descompone la aproximación a un *waypoint*. Así, primero se deberá alcanzar la altura y la orientación (`heightAndAngleReached`) y finalmente el *waypoint* en sí (`lastWaypointReached`).

Por último, se añade una nueva función como protegida: `SimState2Estado` traducirá convenientemente el estado simulado por la clase `HelicopterModel` al formato que generaría el verdadero DSP (`ESTADO_HELICOPTERO`). El sentido de esa conversión (WCS→Sistema Control Center) lo vimos en el capítulo 2 (sección 2.3), pero ahora cabría preguntarnos sobre la conversión de unidades, que veremos en el siguiente punto.

Clase `HelicopterModel` (en `HelicopterModel.h`/ `HelicopterModel.cpp`)

Es la adaptación a una clase del código contenido en `helisim.c`, simulador del MARVIN, sobre el que hemos construido todo nuestro modelo y nuestro controlador. Se han dejado todas las operaciones en función de los valores de los parámetros que vimos anteriormente.

La estructura `SimState`, que será el tipo de la salida de la función `simstep`, tiene por campos las variables que definen el estado simulado del helicóptero (x, y, z, v_x, v_y, v_z, B_x, B_y, B_z, W_x, W_y, W_z, W_r). Dicha función `simstep` realiza paso a paso las operaciones correspondientes a las ecuaciones Ec 4.4.1-4.4.15.

En lo que respecta a la relación entre los sistemas de referencia del simulador y del Control Center (CCS), todas las magnitudes angulares (B_x, B_y, B_z, W_x, W_y, W_z) requieren forzosamente ser convertidas de radianes en pasos y viceversa, mientras que W_r tiene ya las unidades deseadas ([rpm]). Para las magnitudes lineales vimos que la conversión se resumía a un escalado en el que entraba en juego el R_{Veq} de La Tierra. Teniendo en cuenta que la unidad en la que trabaja el simulador es el mm, sería un avance si pudiéramos ahorrarnos tan tedioso escalado.

Se llegaron a resultados concluyentes que harían esto posible, basándose en la linealidad de las ecuaciones en el punto de hover e incluso en puntos cercanos a él de velocidad lineal no nula. Pero para ello había que partir de la premisa de que toda la matriz de trasformación del sistema del Control Center en el sistema del simulador (CCS→WCS) era escalar; es decir, había que suponer que:

$$\exists\, H \in R^{[11\times 11]} \quad | \quad x = H \cdot x^{\{CCS\}} = \alpha \cdot I \cdot x^{\{CCS\}} \qquad \alpha \in R$$

Lo cual no era cierto en nuestro caso. Por lo tanto no parece haber otro remedio que escalar todas y cada una de las entradas en nuestro simulador para después volver a escalar las salidas y que las unidades sean las correctas en todo momento.

Es decir, se calculará la consigna para el controlador a partir del *waypoint*:

$$x_1^d = \frac{2\pi}{360^o} \cdot f^d \cdot R_{Veq} \cdot 10^3 \ [mm]$$
$$x_2^d = -\frac{2\pi}{360^o} \cdot L^d \cdot \cos(f) \cdot R_{Veq} \cdot 10^3 \ [mm]$$
$$x_3^d = z^d \cdot 10^3 \ [mm]$$
$$x_9^d = \frac{W_{360}}{2\pi} \cdot \theta^d \ [1/W_{360}]$$

Y tras cada simulación será necesaria una nueva transformación para enviar los datos definitivamente al DSP Simulator:

$$f = \frac{x_1 \cdot 10^{-3}}{R_{Veq}} \cdot \frac{360}{2\pi} [^o]$$
$$L = -\frac{x_2 \cdot 10^{-3}}{R_{Veq} \cdot \cos(f)} \cdot \frac{360}{2\pi} [^o]$$
$$z = x_3 \cdot 10^{-3} [m]$$
$$\theta = \frac{2\pi}{W_{360}} \cdot x_9 \ [rad]$$

Y análogamente para el resto de magnitudes que, lineales o angulares tendrán expresiones muy parecidas. Sólo el caso de las revoluciones del rotor no necesitará de conversión alguna, ya que vienen siempre expresadas en las mismas unidades, r.p.m.

En todas las expresiones anteriores, tomaremos $R_{Veq} = 6371 \cdot 10^3$ m, cos(f) = cte, tal y como se justificó cuando estudiamos los sistemas de referencia en el capítulo 2.

Continuando ahora con el estudio de la clase `HelicoperModel`, para realizar el cambio de coordenadas HCS→WCS, se define otra función llamada `drotxy` que gira dos vectores un ángulo especificado.

La macrofunción `SERVOSMOOTH` simula además el retardo de los servo-actuadores, por lo que a la definitiva entrada *u* se le impone una dinámica que hace imposible la variación brusca de sus valores.

Se añadieron además una serie de *Set* y *Get* para tener acceso a los estados *x*, que inicialmente no estaban definidos en el código (pero son necesarios para el control) y a las salidas *y* resultantes de la simulación (desde el principio en `SimState`).

Cabecera `kappa.h`

Implementa la función $\kappa(x_9)$ que define los valores de las ganancias del control LQR en función de la orientación, como vimos anteriormente. La operación se realiza mediante un acceso a las tablas contenidas en `Kis.h`, donde el índice será función de x_9, teniendo en cuenta la simetría de κ.

Cabecera `Kis.h`

Contiene las tablas de valores de las ganancias de control para todos los valores de x_9 comprendidos en π radianes ($W_{360}/2$ pasos). Este archivo será generado con el software de simulación desarrollado en Matlab, que veremos en la sección 4.5.

4.4.2 Validación del modelo

En este punto estudiaremos cómo de realista es el modelo del helicóptero cuya implementación acabamos de ver y, lo que es más importante, propondremos una serie de experimentos para la identificación de los parámetros de dicho modelo. Esto permitirá, entre otras cosas, realizar simulaciones con un comportamiento en vuelo más parecido al del HERO real.

Las ecuaciones del modelo se fundamentan principalmente en la experimentación. Es decir, reconocen una dependencia de unas variables respecto a otras y proponen una ecuación cuyos parámetros carecen de un sentido físico concreto. O lo que es lo mismo, no se basan en parámetros reconocibles del helicóptero, como puede ser la envergadura de las palas, el grado de potencia del motor, o ni siquiera el propio peso. Esto facilitará mucho las cosas a la hora de definir los experimentos necesarios para la identificación, pero hará que datos técnicos de nuestro vehículo que podamos conocer a priori resulten de poca utilidad para dicho fin.

A su vez, la física en la que se basan dichas ecuaciones es muy general, y no tiene en cuenta fenómenos concretos como la presión asimétrica en el vuelo horizontal o el efecto tierra. Este último en concreto tiene especial importancia ya que, a parte de definir una relación interesante del helicóptero con su entorno, puede resultar vital durante dos de las fases más críticas del vuelo: el despegue y el aterrizaje.

Cerca del suelo, el efecto tierra proporciona un impulso extra al helicóptero, dado que el aire expulsado por las palas rebota contra el suelo y proporciona más potencia al aparato. Asimismo, en el vuelo horizontal se podría aprovechar la presión adicional del aire para obtener más impulso. Esto se debe al incremento en la eficiencia de la pala en su proximidad con el suelo, que se debe a dos fenómenos diferentes:

- El primero y el más importante es la reducción de la velocidad del flujo de aire inducido. Puesto que el suelo interrumpe el flujo de aire bajo el helicóptero, se reduce la velocidad del flujo descendente inducido. El resultado es menos resistencia inducida y sustentación más vertical.

- El segundo fenómeno es la reducción de los vórtices en el extremo del rotor. Cuando se está operando con efecto suelo, la parte externa y descendente del flujo de aire tiende a restringir la generación de los vórtices del extremo del rotor. Esto hace más efectiva la parte externa de las palas y reduce la turbulencia causada por la circulación de los remolinos. Ver Fig IV.2 en Anexo IV.

El efecto suelo puede empezar a considerarse despreciable a partir de una altura del orden del diámetro del rotor, lo que en nuestro caso no llega ni a 2 metros. En todo caso, y dado que el efecto es positivo respecto a nuestros intereses de estabilidad en despegue y aterrizaje, consideraremos que la exclusión de este fenómeno en el modelado no resulta de tanta gravedad. Otros fenómenos pueden considerarse de poca importancia, especialmente si consideramos las bajas velocidades que nuestro control impondrá en todo momento.

Y al fin y al cabo, se obtiene un modelo más realista que el primitivo que creamos al inicio del proyecto: éste nuevo tiene ya prácticamente todas las variables que se considerarán del helicóptero real (posición, velocidades, *roll*, *pitch*, *yaw*...) y un comportamiento menos ideal. Toda una mejora respecto a uno que se orienta al instante e impone una velocidad de vuelo sin más.

Experimentos para la identificación de los parámetros del simulador para el HERO

Todos estos experimentos deben de realizarse partiendo desde la posición de equilibrio, es decir, desde un *hover*. Para no obtener datos que en realidad son relativos al conjunto controlador-helicóptero, todos estos ensayos deberán ser realizados por un piloto en tierra. Los experimentos se podrán realizar uno a continuación de otro, todos seguidos, y estudiar posteriormente los logs obtenidos de cada uno de ellos para obtener la información que nos interesa. Lo ideal es que todas las pruebas se realicen en condiciones óptimas para el vuelo, en ausencia por ejemplo de viento, para que en el modelado no se introduzcan tales perturbaciones.

1) *Hover*. Recogiendo simplemente los valores en las actuaciones durante un *hover* (lo más perfecto posible) se obtienen todos los valores de u en el punto de equilibrio, y por tanto también los valores de los parámetros E_6, E_7 y E_8. Como difícilmente las actuaciones serán constantes, podemos tomar un valor medio, teniendo en cuenta que para el cíclico este valor debería ser siempre nulo.

2) Relación gas - rotor. Se realizan pequeños incrementos en el gas y se mide la respuesta de la velocidad de giro del rotor dada por el tacómetro. Si consideramos desacoplados los efectos del gas y del colectivo sobre dicha velocidad (Ec 4.4.1), podemos calcular sin mucho problema la constante k_4. Esto se puede hacer, por ejemplo, por mínimos cuadrados.

3) Relación colectivo-rotor. Consideramos ahora la segunda pare de la Ec 4.4.1. Si E_1 representa las r.p.m del rotor en vuelo normal, bastará un historial de vuelo para realizar una media y conocer su valor. Para que éste sea cercano al de E_6, los vuelos realizados en ese historial deben de ser muy suaves y cercanos en todo momento al *hover*; si no existe un historial tal, hacer como experimento extra un vuelo suave y en varias direcciones que nos sirva. Suponiendo entonces E_1 conocido, la constante k_2 se calculará

de forma análoga al anterior experimento: tocando sólo el colectivo y midiendo las variaciones en la velocidad de giro del rotor.

4) Movimiento vertical. Haciendo pequeños y suaves ascensos y descensos, y suponiendo que se recogen datos también de las velocidades y aceleraciones lineales, podemos obtener el cociente E_3 / E_2 de la Ec 4.4.2 transformada en:

$$\frac{E_3}{E_2} = \frac{\max(spc)}{spc} \cdot \frac{1}{W_r} \cdot \left(\frac{\Delta v_z}{g\ T} + 1 \right) = \frac{\max(spc)}{spc} \cdot \frac{1}{W_r} \cdot \left(\frac{a_z}{g} + 1 \right)$$

utilizando ya sean las velocidades o las aceleraciones obtenidas, y haciendo una media sobre los valores obtenidos. Dado que lo que realmente nos interesa es al cociente, podemos dejarlo así, o bien darle a E_2 el valor medio de W_r durante este mismo experimento y sacar E_3 en consecuencia.

5) Cíclico. Haciendo pequeños juegos con cada uno de las actuaciones del cíclico por separado (primero sólo cíclico *x*, después sólo cíclico *y*) y midiendo las velocidades angulares inducidas, podemos sacar los valores de los cocientes E_{4x} / E_{5x}, E_{4y} / E_{5y}.

6) Giro con rotor de cola. Si hacemos al helicóptero girar actuando sólo sobre el rotor de cola y no sobre el colectivo, el parámetro E_9 viene de la relación entre esta actuación y la velocidad angular en *z*, según la Ec 4.4.5.

7) Giro con colectivo. Si ahora lo hacemos girar con la actuación sobre el colectivo, de la misma Ec 4.4.5 podemos estimar el valor de k_3.

8) Movimiento lateral. Por último hay que realizar una serie de movimientos laterales con el helicóptero, para medir la influencia del rotor de cola. Aquí tendremos que utilizar el valor anterior de E_{10}, a no ser que se conozca un valor más preciso de la aceleración máxima generada por dicho rotor. Con este valor y las relaciones obtenidas con este experimento entre aceleración lateral relativa al helicóptero ($a_y^{\{HCS\}}$) y actuación sobre el rotor de cola (*sph*), se obtiene el valor de k_4 a través de la Ec 4.4.11.

Una vez finalizados todos estos experimentos y calculados todos los parámetros, hay que someter los valores obtenidos al juicio del *hover* matemático, cuyas ecuaciones se expresan en las Ecs II.2. Esto hará que sea necesario reajustar algunos valores, pero lo razonable sería que no significativamente. Para hacernos una idea de los órdenes de magnitud que tenemos que obtener, sí valdría considerar los datos del MARVIN, ya que las diferencias no son tantas.

· Código en Matlab:

4.5 Programas en Matlab

En la carpeta `Matlab/`, y fuera de toda la estructura de directorios, se encuentran todas las funciones que se crearon en Matlab y que se explican a continuación. En concreto, se uso la versión 6.5 del programa para Windows.

4.5.1 Implementación software

`control.m`

Al ejecutar esta función, se simula el control de vuelo del modelo real con una ganancia LQR diseñada sobre el modelo linealizado.

En cada iteración del bucle de control se le da una consigna al sistema, se simula (`simstep`) y se calcula la ganancia K para el próximo paso, ya sea con `lin(`x_9`)` o con `kappa(`x_9`)`. Veremos cómo la segunda opción es mucho más rápida que la primera. Finalmente el valor que tomará la próxima actuación será la saturación (`satura.m`) del resultado de $u = u^h - K \cdot (x - x^h)$, según el control LQR.

Toda la información respectiva a la simulación se almacena en los vectores t, u, x, y, de forma que nos es sencillo hacer una representación de la evolución en el tiempo de cada una de las variables del problema. Además se realiza una pequeña animación del vuelo, con posición y orientación del helicóptero en toda su trayectoria. Esta animación puede pasarse a video `avi` fácilmente. En el ejemplo `control_05.avi` podemos ver una simulación de un vuelo en forma de ocho de 8x8 metros, aunque las consignas se pueden cambiar para describir la figura que se desee. En las figuras III.2 del Anexo III podemos además observar las representaciones obtenidas con esta función para las entradas y salidas de dicho vuelo, para un valor concreto del control que se especifica en el propio anexo.

`satura.m`(`u = satura(u)`)

En el cálculo de las entradas según la realimentación lineal del estado, dichas entradas pueden tomar cualquier valor. Por eso para ser más realistas es necesario acotar esos valores entre los máximos y mínimos permitidos. Además, en nuestro simulador los valores serán enteros, por lo se hace necesario además redondear dichos valores al entero más próximo. Todo esto lo hace la función `satura(`u`)` que toma un vector de entradas calculado cualquiera y devuelve uno de enteros acotado entre los valores máximos y mínimos.

`lin0.m`

Esta función calcula las matrices del sistema linealizado y la ganancia K del control LQR correspondiente al punto de hover $x^h = [0 \;\; 0 \;\; 0 \;\; 0 \;\; 0 \;\; 0 \;\; 0 \;\; 0 \;\; 0 \;\; 0 \;\; E_6]$; esto como vimos no nos resulta muy útil, al menos que queramos controlar el helicóptero de forma que está permanentemente orientado hacia el Norte. La verdadera

utilidad de esta función es la de dar valores a las variables globales más importantes de todo nuestro problema. Es decir, sólo aquí se le dan valores numéricos a las *ki*, las *Ei*, y demás constantes y, lo más importante de cara al control, las matrices de pesos *Q* y *R*. Así, si decidimos cambiar algún valor numérico, sólo tenemos que hacerlo en esta función y, eso sí, ejecutarla siempre desde un principio.

Para calcular el control LQR se utiliza la función de Matlab `dlqr`, apta para sistemas discretos en el tiempo. Para la dependencia del modelo lineal con x_9 se pensó en usar variables simbólicas, pero por el momento esta función en concreto sólo admite valores numéricos.

Además de eso, esta función realiza un estudio de la controlabilidad y observabilidad del sistema linealizado en función de los rangos de las matrices *M*, *Ms* y *N*. El sistema resulta siempre controlable, incluso en las salidas, y observable. Se entiende que un sistema es de estado controlable si es posible llevarlo de cualquier estado inicial a cualquier estado deseado en un tiempo finito (sin restricciones en el control). Análogamente se define la controlabilidad en la salida. Por tanto, este hecho es una buena noticia.

Aunque quedara fuera del objetivo de este proyecto, se intento ajustar un controlador que fuera lo más suave posible. Para ello se definieron unos pesos que penalizarán mucho toda actuación, especialmente del rotor de cola y, entre los estados, se penalizará más la velocidad angular en *z*. Esto es así porque el vuelo se mostraba especialmente inestable en el giro del helicóptero en torno a sí mismo.

El resultado es un control muy suave, que hace que el sistema responda muy lentamente, pero saliendo muy poco del punto de equilibrio (*hover*), por lo que es muy estable.

`lin.m` (K = `lin`(x_9))

Esta función realiza la linealización y el cálculo de la matriz de ganancias *K* de la misma forma que `lin0.m`, y con los mismos valores numéricos, pero esta vez sí en función de *Bz*, es decir, de la orientación del helicóptero con respecto al norte. De esta forma nos será útil para probar un nuevo juego de pesos o de otros parámetros dentro de la función `control.m`, calculando la *K* en cada iteración, sin necesidad de generar todos los valores para κ (x_9). Para esto último, la generación de las tablas de *K* para todos los valores posibles de x_9, la función `gnrKij.m` hará uso precisamente de esta `lin.m`.

`gnrKij.m`

Una vez que tenemos claro que los pesos de las matrices *Q* y *R* son los adecuados (algo que es complicado de saber) para el control, podemos generar los valores de κ (x_9) para todo x_9. De eso se encarga esta función. De estudios realizados analíticamente (ver sección 4.4) y con versiones previas de esta función, pudimos comprobar que aquellas ganancias que eran función de x_9 (K_{i1},K_{i2},K_{i4},K_{i5}) presentaban simetría, ya fuera respecto al eje $x_9 = 0$ ó respecto al origen (donde nuevamente $x_9 = 0$). Sabiendo esto, y que los valores del resto de ganancias es constante, podemos calcular sólo para los valores de x_9 entre 0 y $W_{360}/2$ para después generalizar.

Esto es de toda lógica, ya que el origen y el signo de la orientación es arbitrario: una posición girada 45º hacia la izquierda debe ser simétrica respecto a una girada hacia la derecha.

Si queremos ser exactos, se deben calcular los pesos para todos los valores enteros entre estos dos extremos, pero en el caso de que se quisiera reducir el tamaño de las tablas generadas, sería posible hacer un cálculo más grosero aumentando el valor del paso con el que iteramos (variable `step_x9`). Después se podría interpolar o hacer algún otro tipo de aproximación.

Para los pesos que deben de ser constantes, se toma en realidad una media aritmética de todos los valores resultantes, ya que en ocasiones se observaron pequeñas oscilaciones numéricas.

El tiempo de computación para el cálculo de todos los valores de κ (x_9), considerando los valores de $x_9 \in [0, W_{360}/2]$ es de más de 10 minutos en un Pentium 4 a 2.40GHz con 512MB de RAM (*Motemarte*).

`Kij.m`

Este archivo servirá únicamente para almacenar los valores generados por la función `gnrKij` en las variables globales `K_i_j`. Aquellas que sean variables con x_9 serán un vector [1xW_{360}/2+1] y las que no simples constantes [1x1].

Al ejecutar esta función y dar valores numéricos a las `K_i_j`, ya estamos en disposición de usar la función `kappa(x9)`, implementación en Matlab de la función $\kappa(x_9)$, que resultará mucho más rápida que usar `lin(x9)` en cada iteración, ya que los cálculos ya están hechos. Exactamente este procedimiento será el que se utilice en el código en C++ definitivo: se generará como veremos una cabecera con las tablas para los `K_i_j` variables con x_9 y se tendrá acceso al valor correspondiente mediante un simple indexado a la tabla.

De echo, una vez que estamos convencidos de que el controlador es el adecuado, basta copiar la información correspondiente a los valores de los `K_i_j` en un archivo que llamaremos `Ktext.m` y llamar al ejecutable `gnrKis` para generar un archivo de cabecera `Kis.h` que está casi listo para ser incorporado al código en C++. El procedimiento exacto, que veremos más adelante, hace que el cambiar los valores del controlador sea algo más llevadero.

`gnrKis` es un pequeño ejecutable que hemos programado para pasar los vectores generados en Matlab a un formato aceptado por C++. Por ejemplo, hay que cambiar corchetes ‘[ ]’ por llaves ‘{ }’; pero lo más importante, hay que introducir comas ‘ , ‘ entre los distintos componentes. Ni siquiera lo hemos intentado, pero tiene que ser bastante trabajoso poner a mano un total de 2048x20 comas. El único retoque que sí hay que hacer a mano es especificar el tipo de dato (en nuestro caso `double`) y añadir los corchetes tras el nombre de las variable indicando que se trata de un vector si es el caso. Pero eso no parece tanto trabajo en comparación.

kappa.m (K = kappa(x_9))

Esta función hace uso de las variables K_i_j definidas por Kij.m para implementar de una forma veloz a la función $\kappa(x_9)$ tan deseada. Toma el valor de x_9, que como sabemos debe estar normalizado entre $(-W_{360}/2, W_{360}/2]$ y lo convierte en el índice correspondiente para entrar en las tablas. Por supuesto, para ello lo primero que hay que hacer es convertirlo a entero.

De los valores de K constantes no hay que preocuparse, por lo que la matriz de ganancias se inicializa con dichos valores y el resto a 0. Para los valores que dependen de la orientación (como vimos por el cambio de S.R.) llega el momento de aprovechar la simetría.

Para x9>=0, basta sumarle 1 al propio valor de x9 para obtener el índice justo y entrar en las tablas, es decir `K(1,1) = K_1_1(x9+1);`

Para x9<0, la conversión a un índice válido pasa por un simple cambio de signo a x9, pero el definitivo valor de $K_{i,j}$ depende del tipo de simetría de la función que describe a dicha ganancia en función de x9:

- Simetría respecto al origen: El valor de $K_{i,j}$ correcto es el opuesto del valor al que se accede mediante el índice −x9, es decir: `K(1,1) = -K_1_1(-x9);`

- Simetría respecto el eje $x_9 = 0$: Basta acceder a las tablas con el índice ya corregido en signo, es decir: `K(1,2) = K_1_2(-x9);`

En C++, programaremos algo muy parecido (cambiando los formatos y la indexación *i = i-1*) para el cálculo de $K_{i,j}$ en función de x_9, ya que es lo más rápido.

simstep.m ([x y] = simstep(u,x,y))

No es más que una adaptación al lenguaje de Matlab del código del simulador del helicóptero, contenido en la función de mismo nombre de la clase Helicopter-Model. Así, podremos simular también en Matlab el comportamiento de nuestro modelo real (sigue siendo un modelo, pero ya no está linealizado) ante un determinado controlador. La simulación del control, como vimos, la realizamos con la función control.m.

fi.m, **gi.m** (fi(x, u), gi(x, u))

Simplemente representan a las funciones del modelo que definimos en su momento (Ecs II.1) como fi(x, u), gi(x, u). Serán útiles en el cálculo de las derivadas parciales, ya sea en lin0 o en lin.

También se desarrolló software para la interpretación de Logs en Matlab:

`MrLog.m`

Basta salvar el log que se quiere interpretar, de los creados por el Status Module del RIL, en un archivo de extensión '`.m`'; el nombre del archivo no importa, nosotros lo llamaremos `MrLog.m`. Entonces se declara la variable global `aux` y se inicializa a la matriz formada por el log, sin olvidar comentar la primera línea, correspondiente a los nombres de las variables. Esta variable `aux` será la que utilice la función `intLog` para interpretar el Log. A este respecto, hay que tener la precaución de que los índices de las variables coinciden con las del log, ya que como sabemos podemos cambiar las variables de las que deseamos hacer log. En realidad, basta una pequeña comprobación.

`intLog.m`

La función `intLog` toma los datos del log de la variable global `aux` y representa todas las variables de las que se hizo log en función del tiempo. El log puede ser de una simulación o de una prueba real, esto nos es indiferente.

Hace además una pequeña animación de la trayectoria que puede incluso pasarse a video '`.avi`', tal y como se hacía con la simulación en `control.m`.

Como la cantidad de información que se maneja es tan enorme, para que el procesamiento de ésta se realice en un tiempo razonable y sin necesidad de una gran cantidad de espacio en disco, se ha introducido un sesgo, modificable mediante el valor de la variable `sesgo`, que debe estar entre [0,1]. Cuanto más cercano a 1 sea su valor, desestimaremos mayor cantidad de información de forma aleatoria y uniformemente distribuida.

Y además se realizó un esquema en Simulink:

`sim_discreto.mdl`

En las fases más iniciales de esta parte del proyecto, se construyó un modelo lineal en el espacio de estados y discreto en el tiempo, contenido en el archivo `sim_discreto.mdl`. En cuanto se vio que la linealización del sistema dependía de la orientación dejó de ser útil, ya que se debería de andar actualizando las matrices del sistema linealizado cada vez que cambiara B_z. Pero precisamente en esas fases iniciales, fue utilizado en pruebas para comprobar que, en torno al punto de equilibrio, el comportamiento de ambos sistemas, real y linealizado, era el mismo. Es decir, introduciendo pequeñas perturbaciones en las entradas correspondientes al punto de equilibrio, se comprobaba que las salidas eran idénticas en torno a dicho punto, con lo que la linealización (vimos qué compleja era) estaba correctamente realizada.

4.5.2 Utilización del software

Pasos a seguir para generar `Kis.h`

1) Se introducen los pesos *Q*, *R* deseados en el código de `lin0.m` y se ejecuta dicha función. Esto define todos los parámetros, incluidos *Q* y *R* como variables globales que no serán modificadas en ningún otro punto. Es posible hacer simulaciones del control implementado con la función `control.m`, donde *K* deberá calcularse con la función `lin.m`. En este caso se aconsejan consignas de vuelo bien definidas y tiempos de simulación cortos.

2) Una vez que estamos conformes con el control, se genera la *K* con la función `gnrKij.m`, la cual se tomará unos minutos. Cuando se finaliza, es recomendable salvar el *Workspace* por si algo falla en los siguientes pasos no tener que repetir éste.

3) Ahora hay que copiar la información referente a la *K* con dos finalidades distintas. En `Kij.m` nos interesa tener las variables `K_i_j` definidas como globales, para como veremos poder después realizar simulaciones de control más rápidas. Por otro lado, en `Ktext.m` sólo nos interesan los valores en sí, para generar `Kis.h` en el siguiente paso. En realidad ambos archivos sólo se distinguen por la primera línea, pero lo importante es que la finalidad es distinta en cada caso.

4) Basta ejecutar `gnrKis`, un pequeño programa en C++ cuyo código se encuentra en `gnrKis.cpp`, para dejar la información prácticamente en formato aceptado por C, es decir con '{ }' en vez de '[ ]' y comas entre los componentes.

5) Se introducen los tipos (`double`) y las especificaciones de vector '[ ]' y tenemos `Kis.h` listo para el código del UAV Real Simulator.

Pasos a seguir para simular un control salvado en `Kij.m`

1) Ejecutar `lin0.m` para inicializar los parámetros del problema. Como vemos, ejecutar `lin0` es siempre el primer paso: esto se traducirá en comodidad, ya que será el único sitio donde tendremos que tocar el valor de los parámetros cuando se decida algún cambio numérico en el modelo.

2) Ejecutar `Kij.m` para declarar e inicializar las variables `K_i_j` que utiliza la función `kappa.m` en el siguiente paso.

3) Ejecutar `control.m`, donde en este caso la *K* se deberá calcular con la función `kappa.m`. Esto resulta mucho más rápido que utilizar `lin.m`, ya que se han realizado gran cantidad de cálculos previos en la generación misma de `Kij.m`. Como vimos, `control.m` nos permite realizar una simulación de vuelo, con lo que podemos interpretar si el control es o no bueno.