Capítulo 4

Modelo del ROMEO-4R y Simulador del HAM

4. MODELO DEL ROMEO-4R Y SIMULADOR DEL HAM

En el capítulo anterior se vio una descripción completa del *Hardware Abstraction Module*, el cual, se encarga del movimiento del vehículo, utilizando los sensores correspondientes y aplicando las leyes de control. Todo el código que se genera para el ROMEO-4R debe ser probado antes de utilizarlo en el vehículo en sí, ya que un fallo en la programación podría provocar severos daños con el vehículo en movimiento. Para ello se ha creado un simulador del HAM que permitirá aproximar el modelo del ROMEO-4R de una forma muy próxima a la realidad.

En los siguientes apartados se verá una descripción del modelo y del simulador que se acaba de mencionar.

4.1. Modelo del ROMEO-4R

En el Capítulo 1 se estudió la configuración Ackerman en la que se basa el ROMEO-4R. A partir de las ecuaciones que definen a ésta, se obtiene el correspondiente modelo cinemático del vehículo, que nos proveerá de la información necesaria acerca del movimiento del mismo. Además del modelo cinemático, será necesaria la definición de un modelo que tenga en cuenta la dinámica del sistema, lo cual permitirá al vehículo obtener un comportamiento en simulación muy próximo a la realidad.

Dichos modelos, cinemático y dinámico, así como sus correspondientes sistemas de ecuaciones, serán estudiados a continuación.

4.1.1. Modelo cinemático

El modelo cinemático del vehículo proporciona el movimiento del mismo en función su curvatura instantánea y su velocidad de desplazamiento, suponiendo que no existe deslizamiento.

El ROMEO-4R está definido mediante una configuración Ackerman, ya que se trata de un vehículo convencional de cuatro ruedas, donde el centro de guiado (origen del sistema de referencia local) se encuentra en la mitad del eje de las ruedas de tracción (ruedas traseras). No obstante, por simplificación en las ecuaciones, esta configuración se puede asemejar a la de un triciclo. Sin embargo, hay que tener en cuenta que la velocidad real de las cuatro ruedas es diferente y, por consiguiente, el cálculo de la velocidad del centro de guiado del vehículo a partir de la velocidad de una rueda da lugar a errores. A su vez, el radio de curvatura tampoco puede medirse de forma directa a partir de la curvatura de las ruedas directrices ya que cada una de ellas se mueve según un arco de diferente radio.

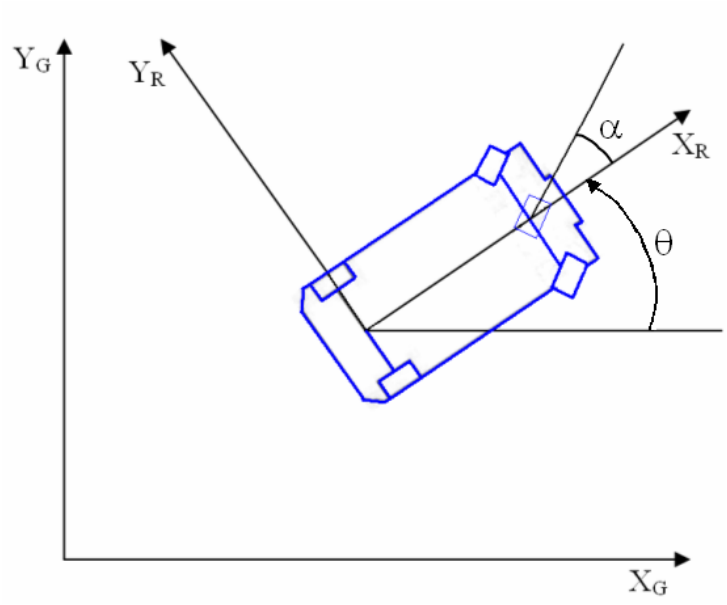

Figura 4.1.1-1: Sistema de referencia en el ROMEO-4R

Según lo anteriormente dicho y observando la Figura 4.1.1-1, la velocidad en el origen del eje de coordenadas local (x, y) sólo tiene componente en la dirección X_R, y el movimiento del vehículo respecto al eje de coordenadas global (X_G ,Y_G) queda descrito por las siguientes ecuaciones:

$$\dot{x} = -v \sin\theta$$

$$\dot{y} = v \cos\theta$$

$$\dot{\theta} = v\gamma$$

donde:

γ: curvatura del vehículo (m^{-1})

v : velocidad de desplazamiento del vehículo (m/s)

4.1.2. Modelo dinámico

El modelo dinámico del vehículo proporciona las referencias necesarias que se deben enviar a los actuadores del sistema para su control, teniendo en cuenta las aceleraciones y masas del mismo.

En el ROMEO-4R, los motores son los actuadores encargados de generar el movimiento de los ejes de tracción y de la caña de dirección del vehículo, de forma que controlan la velocidad y curvatura deseadas en el mismo.

El modelo dinámico que se emplea aquí se puede dividir, básicamente, en dos partes:

- *Modelo del sistema de tracción*: está formado por un motor que mueve el eje de tracción, un sensor que permite medir la velocidad de desplazamiento del vehículo, y un regulador (como puede ser un PID) que permite controlar dicha velocidad en bucle cerrado.

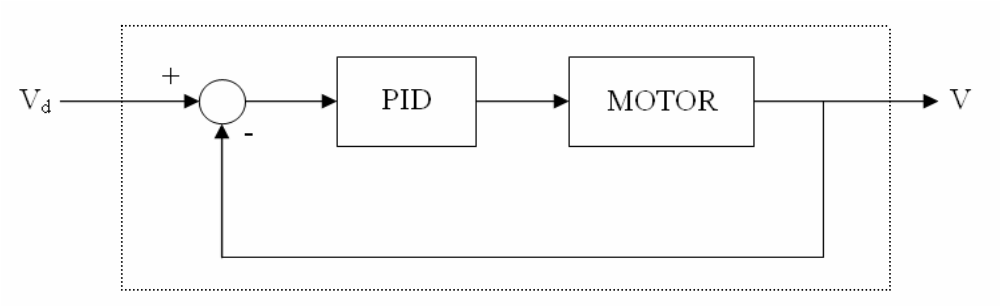

Figura 4.1.2-1: Modelo del sistema de tracción del ROMEO-4R

Este sistema se puede modelar mediante un sistema dinámico de primer orden con una constante de tiempo (T_v), por lo que puede describirse por la siguiente ecuación diferencial:

$$\dot{v} = \frac{1}{T_v}(v_d - v)$$

donde:

v : velocidad real del vehículo (m/s)
v_d : velocidad deseada del vehículo (m/s)

- *Modelo del sistema de dirección*: está formado por un motor conectado a la caña de dirección, un sensor que permite medir el ángulo de dicha caña o de las ruedas del vehículo (α), y un regulador (como puede ser un PID) que permite controlar dicho ángulo en bucle cerrado.

  No obstante, para el diseño del sistema de guiado del vehículo es más interesante utilizar la curvatura (γ) que describe como la variable de control, en lugar de utilizar el ángulo de las ruedas. Para ello será necesaria una transformación (T) entre curvatura y ángulo, que se puede aproximar de la siguiente forma:

$$\gamma = \frac{\tan\alpha}{D}$$

donde D es la distancia existente entre ejes.

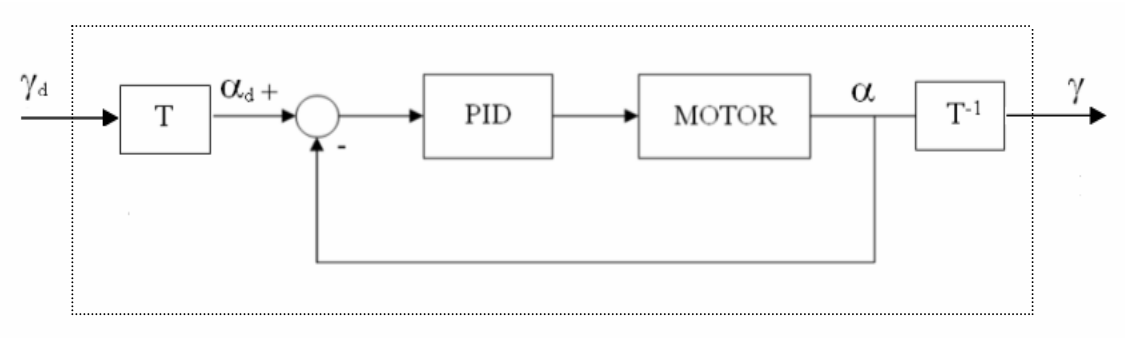

Figura 4.1.2-2: Modelo del sistema de dirección del ROMEO-4R

Este sistema se puede modelar mediante un sistema dinámico de primer orden con una constante de tiempo (T_γ), por lo que puede describirse por la siguiente ecuación diferencial:

$$\dot{\gamma} = \frac{1}{T_\gamma}(\gamma_d - \gamma)$$

donde:

γ : curvatura real del vehículo (m^{-1})

γ_d : curvatura deseada del vehículo (m^{-1})

4.2. Hardware Abstraction Module Simulator

Este módulo se ha implementado basándose en la creación de un modelo cinemático y dinámico del vehículo, que permitirá obtener resultados en simulación muy válidos, de forma que al ejecutar los programas que se creen en el simulador, estos respondan de una forma muy fiel en la realidad.

Un código probado en simulación, pasa rápidamente a ser probado en la realidad con tan solo cambiar los módulos. De esta forma, el módulo del simulador del HAM, sería sustituido por el módulo del HAM, para lo cual únicamente es necesario arrancar el programa *main* del simulador (en caso de que se quiera probar código en simulación) o del HAM (en caso de querer utilizar el código en la realidad)

La compilación de este módulo, se realiza mediante el fichero *Makefile* (*./romeo/ril/HAMsimulator/Makefile*), que contiene una lista de todos los archivos objeto del simulador del HAM, así como una relación de dependencias de éstos con los archivos fuente, de manera que podemos ahorrar tiempo de compilación gracias a la utilización del comando *make*.

Esto proporcionará un programa ejecutable, al que llamaremos mediante el siguiente comando:

>> ./ROMEO_HAM_SIMULATOR [robot id] [x_position] [y_position] [orientation in degrees]

donde:

[robot id] : identificador del robot con el que estemos trabajando.
[x_position]: coordenada inicial X del vehículo.
[y_position]: coordenada inicial Y del vehículo.
[orientation in degrees]: orientación inicial del vehículo.

La estructura del módulo de simulación del HAM viene definida por el siguiente esquema:

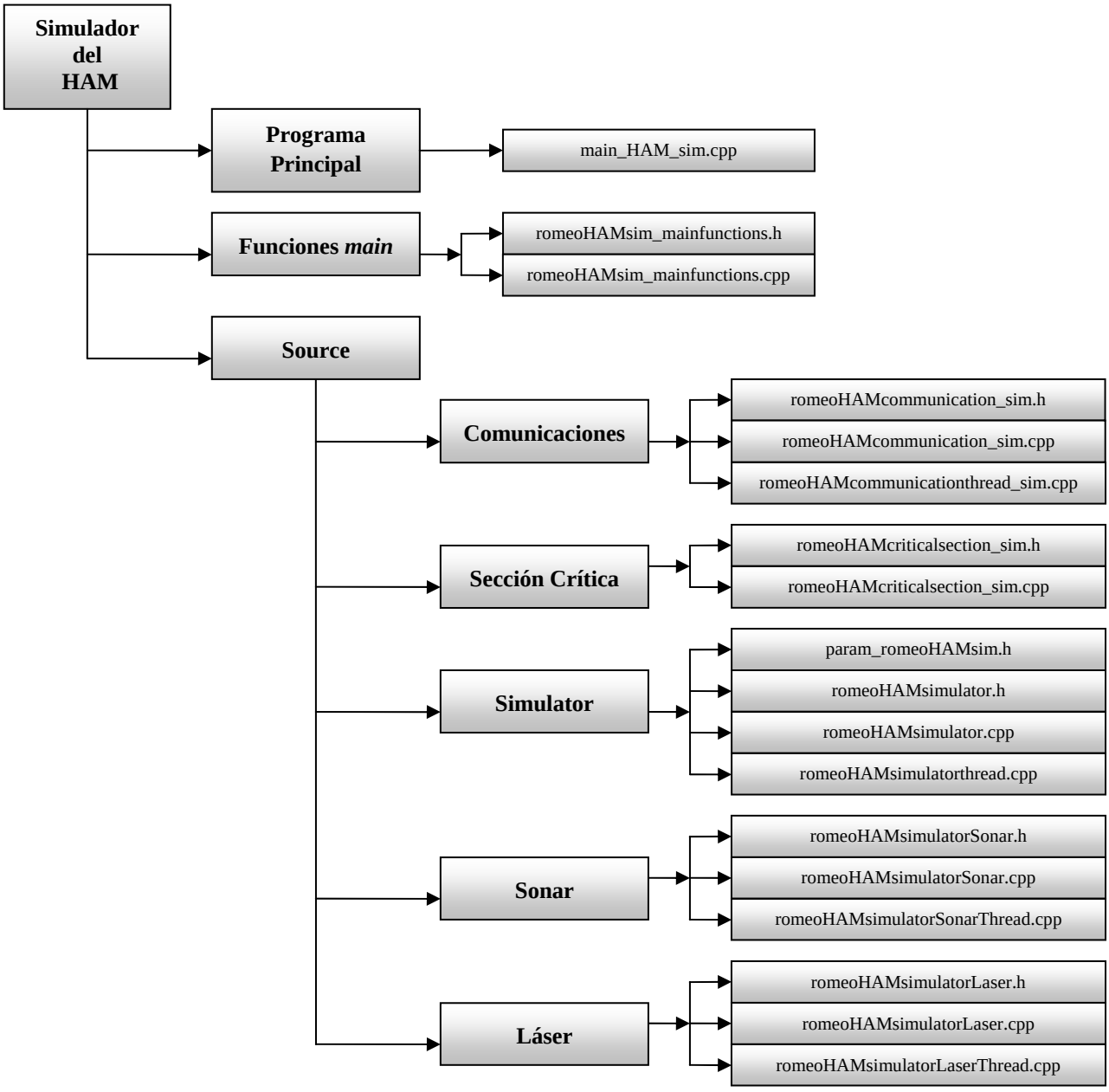

Figura 4.2-1: Esquema del módulo de simulación del HAM

Por último, comentar que este módulo de simulación tiene como entradas las referencias de control para los actuadores, y como salidas, los datos simulados de los dispositivos. Obviamente, este módulo no tiene las entradas y salidas correspondientes a la interactuación con el hardware del robot, al no existir tal en la simulación.

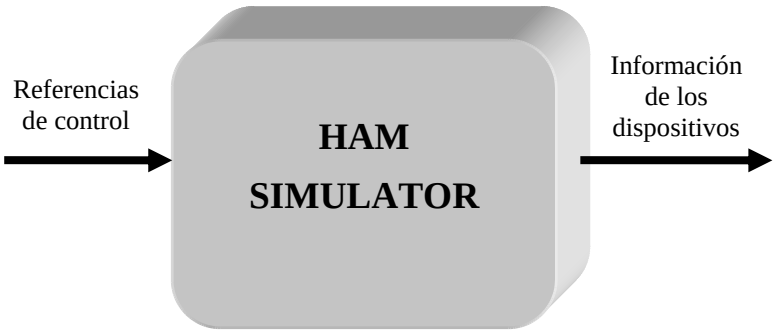

Figura 4.2-2: Entradas y salidas en el módulo de simulación del HAM

A continuación, se verán los elementos de los que consta este módulo.

4.2.1. Programa principal

El programa principal de ejecución del simulador del *Hardware Abstraction Module* (*./romeo/ril/HAMsimulator/main_HAM_sim.cpp*) es el encargado de, básicamente, lanzar el hilo de comunicaciones y los hilos de procesamiento que simulan la cinemática y dinámica del vehículo, así como los distintos sensores, y permanecer a la espera de que se salga del programa pulsando '*Ctrl+C*', para finalizar los hilos.

El módulo de simulación es muy parecido al real, ya que se debe poder cambiar de un módulo a otro sin realizar ningún cambio, y por tanto, conservará casi intacto todo su sistema de comunicaciones y sección crítica, variando únicamente la parte destinada a la simulación de los sensores y del modelo del ROMEO-4R.

Al igual que se vio en el HAM, se contará con dos variables globales y un manejador:

- *end*: variable global que indica la finalización del programa principal y del resto de los hilos.

- *initial_time_ref* : variable global que indica la referencia de tiempo inicial

- *handler_controlC*: manejador al que, una vez que le llega la señal *SIGINT* (provocada cuando el usuario pulsa '*Ctrl+C*'), activa la variable *end*, finalizando todos los hilos que activó el programa principal (hilos de comunicaciones y de procesamiento), así como finalizando también dicho programa.

En este módulo, cabe resaltar la necesidad de programar varias señales distintas para los temporizadores que se van a utilizar en cada hilo de procesamiento, esto es, el hilo destinado a la simulación mediante el modelo cinemático y dinámico llevará asociado un temporizador con una señal determinada (*SIGRTMIN*), mientras que los destinado a los sonares y al láser tendrán asociados idénticos temporizadores con señales distintas (*SIGRTMIN+1* para los sonares y *SIGRTMIN+2* para el láser).

Es importante indicar que se modifica la máscara del proceso, la cual se refiere a las señales que pueden o no interrumpir el funcionamiento de un hilo. Como se ha dicho, se desbloquea la señal *SIGINT* (generada al pulsar '*Ctrl+C*') para que sea tratada por el manejador. A su vez, la máscara de proceso es heredada por los hilos que sean lanzados desde el programa *main*, por lo que se procederá a bloquear las señales *SIGRTMIN*, *SIGRTMIN+1* y *SIGRTMIN+2* en la máscara de este proceso, para desbloquearla en los hilos lanzados por el *main* que requieran del uso de temporizadores, los cuales se programan para un disparo periódico y para esperar dicha señal de forma síncrona.

El código del programa principal tiene el siguiente orden:

- Añade las cabeceras necesarias.
- Crea un objeto de la sección crítica.
- Crea el hilo de comunicaciones y los hilos de procesamiento que simulan la cinemática y dinámica del vehículo, así como los distintos sensores.
- Crea las variables globales anteriormente comentadas.
- Crea y configura el manejador para la señal de finalización del programa.
- Lanza el programa *main*.
- Crea el objeto de la clase que contiene funciones del programa *main*, cuya única función será la de establecer el tiempo inicial de referencia.
- Configura la señal *SIGINT*, desbloqueándola en la máscara del proceso para que sea tratada por el manejador.
- Configura las señales *SIGRTMIN*, *SIGRTMIN+1* y *SIGRTMIN+2*, que estarán asociadas a los temporizadores del hilo de simulación del movimiento del ROMEO-4R, del hilo de simulación del sonar y del hilo de simulación del láser, respectivamente, bloqueándolas en la máscara del proceso.
- Comprueba que se ha llamado correctamente al programa: *>>./ROMEO_HAM [robot id] [x_position] [y_position] [orientation in degrees]*.
- Coloca la referencia de tiempo inicial.

- Envía la posición inicial a la sección crítica.
- Envía el identificador del robot a la sección crítica.
- Lanza el hilo de comunicaciones y los hilos de procesamiento que simulan la cinemática y la dinámica del vehículo, así como los distintos sensores.
- Mientras se están ejecutando los hilos, el programa *main* permanece a la espera de que se pulse '*Ctrl+C*' para finalizar el programa.
- Una vez se active la variable *end*, se finalizarán los hilos y después terminará el programa principal.

4.2.2. Ficheros con funciones para el programa *main*

Es necesario el establecimiento de una referencia de tiempo inicial que permita a los distintos hilos controlar el momento en que le llegan los datos correspondientes. Esta referencia se establece al comienzo de cada hilo, y marcará a su vez el tiempo global de uso del dispositivo.

Esto se implementa mediante la clase *CRomeoHamSimMainFunctions* (*./romeo/ril/HAMsimulator/romeoHAMsim_mainfunctions.h*), que proporcionará al programa *main* una referencia de tiempo inicial, la cual será almacenada en una variable global de dicho programa, llamada *initial_time_ref*, mediante la función *set_initial_time_ref()* (*./romeo/ril/HAMsimulator/romeoHAMsim_mainfunctions.cpp*), de forma que ésta pueda estar disponible para todos los hilos.

La **clase *CRomeoHamSimMainFunctions*** está formada por la siguiente función:

- **set_initial_time_ref**: establece la referencia de tiempo inicial para los distintos hilos de simulación del ROMEO-4R.

  Parámetros:

  - double *init_time_ref: variable en la que se almacenará la referencia de tiempo inicial.

  Devuelve:

  - void.

4.2.3. Source

Al igual que en el resto de los módulos estudiados en el capítulo anterior, en el simulador del HAM existe una carpeta que contiene los ficheros de comunicaciones, los ficheros de la sección crítica y los ficheros de los hilos de procesamiento, que en este caso, son los hilos correspondientes a la simulación de los distintos sensores y del movimiento del ROMEO-4R.

A continuación, se procede a describir el contenido de la carpeta Source.

4.2.3.1. Comunicaciones

Las comunicaciones en el simulador del *Hardware Abstraction Module* tendrán una estructura idéntica a la vista en el HAM, para lo cual solo es necesario ver en el capítulo anterior el apartado *"3.1.4.1. Comunicaciones"*, para conocer la implementación de las mismas en este módulo. Las comunicaciones son idénticas en ambos módulos para facilitar así el cambio de uno por otro al realizar pruebas en simulación o reales, sin tener que cambiar nada más que el programa a ejecutar.

Por tanto, las conexiones, los puertos y los slots empleados en las comunicaciones de este módulo, así como la clase que implementa dichas comunicaciones, son las mismas que usa el módulo real.

4.2.3.2. Sección crítica

Al igual que ocurre con las comunicaciones, la sección crítica en el simulador del *Hardware Abstraction Module* se implementa de forma similar a la existente en el HAM. Para ello, se recomienda ver en el capítulo anterior el apartado *"3.1.4.2. Sección crítica"*, para conocer la implementación de la misma en este módulo.

La **clase *CRomeoCriticalSection_HAM***, que implementa la sección crítica en el simulador del HAM, estará formada por las mismas funciones existentes en la sección crítica del HAM, con la inclusión de las siguientes funciones:

- **SetInitialPosition**: coloca en la sección crítica la posición inicial del ROMEO-4R.

  Parámetros:

  - INIT *initial_position: variable que contiene la posición inicial del ROMEO-4R.

  Devuelve:

  - void.

- **GetInitialPosition**: obtiene de la sección crítica la posición inicial del ROMEO-4R.

  Parámetros:

  - INIT *initial_position: variable que contiene la posición inicial del ROMEO-4R.

  Devuelve:

  - bool: devuelve *true* una vez obtenido el dato.

- **SetRobotID**: coloca en la sección crítica el identificador del robot.

  Parámetros:

  - int *id: variable que contiene el identificador del robot.

  Devuelve:

  - void.

- **GetRobotID**: obtiene de la sección crítica el identificador del robot.

  Parámetros:

  - int *id: variable que contiene el identificador del robot.

  Devuelve:

  - bool: devuelve *true* una vez obtenido el dato.

4.2.3.3. Simulator

Es el hilo encargado de la simulación del movimiento del ROMEO-4R, empleando un modelo cinemático y dinámico del mismo, así como de la simulación del GPS y del giróscopo.

Dicho modelo ya se estudió en el comienzo de este capítulo, obteniéndose las ecuaciones necesarias que lo describen. En este apartado se utilizará una implementación de las mismas, para lo cual se han creado las funciones correspondientes dentro de una clase (*CRomeoHAMsim*) que veremos a continuación.

Los archivos que implementan lo anteriormente dicho son los siguientes:

- *romeoHAMsimulator.cpp* y *romeoHAMsimulator.h*: implementan la clase referente a la simulación del giróscopo, del GPS y del movimiento del ROMEO-4R.

La **clase *CRomeoHAMsim*** está formada por las siguientes funciones:

- **update_sim**: se encarga de la lectura y actualización de las estimaciones del simulador. Primero obtiene los valores del simulador, mediante la función privada *simulator_HAM()*, para después actualizar los valores correspondientes a las estructuras de la DCX, GPS y giróscopo. No se le pasan parámetros, ni devuelve nada, ya que trabaja con funciones y variables privadas de la clase.

  Parámetros:

  - void.

  Devuelve:

  - void.

- **SendDcxData**: coloca en la sección crítica los datos de la DCX para que sean enviados a la red por el hilo de comunicaciones.

  Parámetros:

  - void.

  Devuelve:

  - void.

- **SendGpsData**: coloca en la sección crítica los datos del GPS para que sean enviados a la red por el hilo de comunicaciones.

  Parámetros:

  - void.

  Devuelve:

  - void.

- **SendGyroData**: coloca en la sección crítica los datos del giróscopo para que sean enviados a la red por el hilo de comunicaciones.

  Parámetros:

  - void.

  Devuelve:

  - void.

- **ReceiveRefSpeed**: recibe de la sección crítica la referencia de velocidad que le llega de la red por el hilo de comunicaciones.

  Parámetros:

  - void.

  Devuelve:

  - void.

- **ReceiveRefCurv**: recibe de la sección crítica la referencia de curvatura que le llega de la red por el hilo de comunicaciones.

  Parámetros:

  - void.

  Devuelve:

  - void.

- **ReceiveInitialPosition**: recibe de la sección crítica la posición inicial del ROMEO-4R, que se le pasa al programa principal *main* tras ejecutar el módulo.

  Parámetros:

  - void.

  Devuelve:

  - void.

- **SetInitialPosition**: establece la posición inicial del ROMEO-4R. Para ello, primero recibe de la sección crítica dicha posición inicial, mediante la función *ReceiveInitialPosition()*, que se le pasa al programa principal *main* tras ejecutar el módulo, para luego colocarla en una estructura válida para el modelo cinemático (*CIN_DATA*).

  Parámetros:

  - void.

  Devuelve:

  - void.

- **ForDebugging**: muestra por pantalla las estimaciones del simulador, permitiendo el análisis de los resultados.

  Parámetros:

  - void.

  Devuelve:

  - void.

- **simulator_HAM**: función privada de la clase que se encarga de obtener la salida del simulador. Para ello, ejecuta las funciones que implementan el modelo dinámico (*dinamica()*) y cinemático (*cinematica()*), asigna el resultado a la estructura (*SIM_ DATA*) que contiene los valores cinemáticos (orientación y coordenadas X, Y) y dinámicos (velocidad y curvatura) que salen del simulador, y ejecuta la función que añade ruido a dichos valores (*ruido()*) para que la simulación sea más real.

  Parámetros:

  - void.

  Devuelve:

  - void.

- **dinamica**: función privada de la clase que se encarga del cálculo del modelo dinámico. A partir de las referencias de velocidad y curvatura que le llegan de la red, comprueba que éstas se encuentren dentro de sus límites para, posteriormente, calcular los valores de velocidad y curvatura que adquiere el ROMEO-4R tras aplicar el modelo dinámico (ver el apartado *"4.1.2. Modelo dinámico"*), guardando dichos valores en una estructura válida para dicho modelo (*DIN_DATA*).

  Parámetros:

  - void.

  Devuelve:

  - void.

- **cinematica**: función privada de la clase que se encarga del cálculo del modelo cinemático. A partir de los valores de velocidad y curvatura obtenidos tras aplicar el modelo dinámico (mediante la función *dinamica()*), obtiene los valores de la orientación y coordenadas X, Y del ROMEO-4R tras aplicar el modelo cinemático (ver el apartado *"4.1.1. Modelo cinemático"*), guardando dichos valores en una estructura válida para dicho modelo (*CIN_DATA*).

Parámetros:

- void.

Devuelve:

- void.

- **ruido**: función privada de la clase que se encarga de simular el ruido que se produce en la lectura de los dispositivos, variando por tanto los valores de la estructura de simulación (*SIM_DATA*).

  Parámetros:

  - void.

  Devuelve:

  - void.

- **get_relative_time**: función privada de la clase que se encarga de devolver el instante de tiempo actual referido al instante inicial (que viene dado por la referencia de tiempo inicial a través de la variable *initial_time_ref*).

  Parámetros:

  - void.

  Devuelve:

  - double: variable que contiene el instante de tiempo actual referido al instante inicial.

Las variables anteriormente mencionadas se encuentran definidas en *./romeo/ril/ HAMsimulator/source/simulator/param_romeoHAMsim.h,* y son descritas a continuación:

```
typedef struct
{
      double x;
      double y;
      double orient;
      double speed;
      double curv;
      double time;

} SIM_DATA;
```

```
typedef struct
{
      double x;
      double y;
      double orient;

} CIN_DATA;


typedef struct
{
      double speed;
      double curv;

} DIN_DATA;


typedef struct
{
      double initial_x;
      double initial_y;
      double initial_orient;

} INIT;
```

Por último, indicar que las variables de la clase son inicializadas en el constructor de la misma.

- *romeoHAMsimulatorthread.cpp*: implementa la ejecución del hilo que se encarga de la simulación del giróscopo, del GPS y del movimiento del ROMEO-4R.

  El código del hilo es el siguiente:

  - Añade las cabeceras necesarias.
  - Crea la variable global externa *end*, que fue creada en el programa principal *main* y que indica la finalización del programa.
  - Crea un objeto a partir de la clase que implementa la sección crítica del módulo, *CRomeoCriticalSection_HAM RomeoHamCS*.
  - Lanza el hilo de simulación (*SimThread*).
  - Asigna la señal correspondiente a este hilo de simulación (*SIGRTMIN*), para que sea usada por el temporizador.
  - Asigna el identificador del robot que se le pasa al programa *main* tras ejecutarse el módulo.
  - Crea un objeto de la clase *CRomeoHAMsim*.
  - Establece la referencia de tiempo inicial.

- Configura el temporizador y lo pone en marcha. Se programa para un disparo periódico y para esperar la señal de forma síncrona. El tiempo de espera será *T_CONTROL,* esto es, 50 ms.

- Comienza el bucle del que se saldrá cuando se active la variable *end*.

- Se reciben las referencias de curvatura y velocidad, para el ROMEO-4R, de la red.

- Se realiza la lectura y la actualización de las estimaciones del simulador, mediante la función *update_sim()*.

- Se envían los datos simulados de la DCX, del GPS y del giróscopo a la red.

- Si se quieren analizar los resultados, se pueden imprimir por pantalla las estimaciones del simulador mediante la función *ForDebugging()*.

- Espera el vencimiento del temporizador mediante la llegada de la señal asociada a dicho temporizador (*SIGRTMIN*).

- Una vez se active la variable *end*, elimina el temporizador y finaliza el hilo.

4.2.3.4. Sónar

Es el hilo encargado de la simulación de los sonares. Este hilo no se encuentra desarrollado en su totalidad, habiéndose establecido las estructuras necesarias para que únicamente se completen las funciones necesarias para el desarrollo de la herramienta de simulación de los sonares.

Los archivos que, en principio, implementan la simulación de los sonares son los siguientes:

- *romeoHAMsimulatorSonar.cpp* y *romeoHAMsimulatorSonar.h*: implementan la clase referente a la simulación de los sonares del ROMEO-4R.

  La **clase *CRomeoHAMsimSonar*** está formada por las siguientes funciones:

  - **update_sim_sonar**: se encarga de la lectura y actualización de las estimaciones de los sonares. En principio, únicamente se encarga de transmitir un valor por defecto a la red de cada uno de los sonares, rellenando por tanto la estructura correspondiente (*SONAR_DATA*). No se le pasan parámetros, ni devuelve nada, ya que trabaja con funciones y variables privadas de la clase.

Parámetros:

- void.

Devuelve:

- void.

- **SendSonarData**: coloca en la sección crítica los datos de los sonares para que sean enviados a la red por el hilo de comunicaciones.

  Parámetros:

  - void.

  Devuelve:

  - void.

- **ForDebuggingSonar**: muestra por pantalla las estimaciones de los sonares, lo que nos permite analizar resultados.

  Parámetros:

  - void.

  Devuelve:

  - void.

- **get_relative_time**: función privada de la clase que se encarga de devolver el instante de tiempo actual referido al instante inicial (que viene dado por la referencia de tiempo inicial a través de la variable *initial_time_ref*).

  Parámetros:

  - void.

  Devuelve:

  - double: variable que contiene el instante de tiempo actual referido al instante inicial.

Por último, indicar que las variables de la clase son inicializadas en el constructor de la misma.

- *romeoHAMsimulatorSonarThread.cpp*: implementa la ejecución del hilo que se encarga de la simulación de los sonares del ROMEO-4R.

  El código del hilo es el siguiente:

  - Añade las cabeceras necesarias.
  - Crea la variable global externa *end* que fue creada en el programa principal *main* y que indica la finalización del programa.
  - Crea un objeto a partir de la clase que implementa la sección crítica del módulo, *CRomeoCriticalSection_HAM RomeoHamCS*.
  - Lanza el hilo de simulación (*SimSonarThread*).
  - Asigna la señal correspondiente a este hilo de simulación (*SIGRTMIN+1*), para que sea usada por el temporizador.
  - Asigna el identificador del robot que se le pasa al programa *main* tras ejecutarse el módulo.
  - Crea un objeto de la clase *CRomeoHAMsimSonar*.
  - Crea el temporizador y le asocia la señal que se le pasa por parámetro.
  - Configura el temporizador y lo pone en marcha. Se programa para un disparo periódico y para esperar la señal de forma síncrona. El tiempo de espera será *T_CONTROL,* esto es, 50 ms.
  - Comienza el bucle del que se saldrá cuando se active la variable *end*.
  - Se realiza la lectura y actualización de las estimaciones de los sonares, mediante la función *update_sim_sonar().*
  - Se envían los datos simulados de los sonares a la red.
  - Si se quieren analizar los resultados, se pueden imprimir por pantalla las estimaciones del simulador mediante la función *ForDebuggingSonar()*.
  - Espera el vencimiento del temporizador mediante la llegada de la señal asociada a dicho temporizador (*SIGRTMIN+1*).
  - Una vez se active la variable *end*, elimina el temporizador y finaliza el hilo.

4.2.3.5. Láser

Es el hilo encargado de la simulación del láser. Al igual que ocurre con el hilo de los sonares, este hilo no se encuentra desarrollado en su totalidad, habiéndose establecido las estructuras necesarias para que únicamente se completen las funciones necesarias para el desarrollo de la herramienta de simulación del láser.

Los archivos que implementan la clase referente a la simulación del láser del ROMEO-4R son *romeoHAMsimulatorLaser.cpp* y *romeoHAMsimulatorLaser.h*.

La **clase *CRomeoHAMsimLaser*** está formada por las mismas funciones que se implementan para los sonares, cambiando únicamente sus nombres:

- **update_sim_laser**: se encarga de la lectura y actualización de las estimaciones del láser.
- **SendLaserData**: coloca en la sección crítica los datos del láser para que sean enviados a la red por el hilo de comunicaciones.
- **ForDebuggingLaser**: muestra por pantalla las estimaciones del láser, lo que nos permite analizar resultados.
- **get_relative_time**: función privada de la clase que se encarga de devolver el instante de tiempo actual referido al instante inicial (que viene dado por la referencia de tiempo inicial a través de la variable *initial_time_ref*).

Por último, el archivo *romeoHAMsimulatorLaserThread.cpp* implementa la ejecución del hilo que se encarga de la simulación del láser del ROMEO-4R. Su código es idéntico al que se ha comentado para los sonares, resaltando la diferencia en la señal que se asocia a su temporizador, que en este caso es *SIGRTMIN+2*.

4.3. Conclusiones

En este capítulo se han presentado los modelos cinemático y dinámico que definen al ROMEO-4R, para implementarlos posteriormente en el simulador del *Hardware Abstraction Module*, y de esta forma poder obtener el comportamiento del vehículo de una forma muy aproximada a la realidad.

Aunque en muchos casos el modelo cinemático, por sí solo, suele representar de una forma más que aceptable la realidad, en nuestro caso es conveniente el uso del modelo dinámico para darle una mayor importancia a la inercia del vehículo, la cual se tuvo muy en cuenta a la hora de diseñar el algoritmo de seguimiento de caminos que vimos en el anterior capítulo.

Se ha realizado una aproximación de los sistemas de actuación del vehículo mediante sistemas de primer orden, lo cual es bastante válido si se realiza un ajuste apropiado de los reguladores de tracción y dirección. Esta suposición no consiste en una mera simplificación de los sistemas implicados, sino que presenta numerosas ventajas en la práctica, como puede ser el hecho de que tanto la dirección como la tracción funcionen de una forma más suave, sin brusquedades ni oscilaciones, lo que supone una mayor comodidad para posibles pasajeros. Por otra parte, se consigue un menor consumo energético y desgaste mecánico de los actuadores.

En lo que respecta a la simulación propiamente dicha, se ha conseguido implementar un módulo que simula el HAM, lo que nos permite cerrar la arquitectura del vehículo para su uso en modo de pruebas. De esta forma, todo el código que se genere para el ROMEO-4R podrá ser probado en el simulador para ser depurado, de manera que antes de ser utilizado en la realidad tengamos unos resultados bastante aproximados y fiables.