Implementación de comportamientos simples con Aurigapy y Makeblock

Memoria de la primera práctica Repositorio:

https://github.com/sistemas-multirobot-grupo-2/practica-1

Roberto Saúl Cova Rocamora
Jonathan Mortes Esquer
Francisco Muñoz Gómez
Yoinel Novelo Téllez
Abel Parodi Fernández
José Miguel Torres Cámara

Sistemas Multirobot
Ingeniería Robótica
Universidad de Alicante
Marzo de 2019

Índice

1. Introducción

En esta práctica hemos desarrollado un *framework* de programación, sensorización, comunicación y control de robots móviles, por tanto, permite crear un sistema multirobot. El lenguaje escogido para este *framework* ha sido Python, debido a la flexibilidad y rapidez para prototipar. El objetivo del trabajo ha sido la organización y programación de un sistema multirobot utilizando dicho *framework* utilizando un hardware limitado tanto en cantidad como en calidad al que teníamos que ceñirnos para realizar una aplicación extensible para varios robots.

2. *Hardware* y *software* utilizados

Los sistemas utilizados son 2 robots tipo mBot Ranger de Makeblock con las siguientes características comunes:

- **Sensor sigue-líneas**
- **Sensores de luz**: LDRs
- **Sensor de temperatura**
- **Sensor de distancia** por ultrasonidos
- **Giroscopio**
- **Emisor de sonido**
- **Motores DC** que permiten un control diferencial del desplazamiento de los robots

Adicionalmente un robot (líder) dispone de una pinza y otro de una brújula y un sensor de distancia (basado en ultrasonidos). Para las comunicaciones entre el PC y el robot, utilizaremos el protocolo *Bluetooth* junto con la librería AurigaPy. Esta última es imprescindible para realizar el control de estos robots con *scripts* en *Python* a través de un puerto serie *Bluetooth* 4.0.

Todo el código, incluyendo la librería AurigaPy, así como su documentación (la elaborada por nosotros), puede encontrarse en nuestro repositorio: https://github.com/sistemas-multirobot-grupo-2/practica-1.

3. Aplicación

Aunque el *framework* permite flexibilidad a la hora de desarrollar una aplicación u otra, se ha escogido un sistema multi-robot donde hay un líder y un seguidor. Ambos se mueven por un espacio cartesiano limitado por una cinta negra, por tanto, tienen la restricción de 1 GDL (Grado De Libertad). El robot líder irá delante del grupo, "ciego" debido a que no tiene sensor de ultrasonidos para detectar movimiento u objetos delante de él y con una pinza. Este esperará a detectar una luz intensa, lo cual será un indicador de que hay un objeto delante. Este abrirá entonces la pinza, cogerá el objeto y comenzará a moverse hacia atrás hasta volver a detectar una luz intensa. En ese momento dejará el objeto y volverá a repetir el ciclo. Por otro lado, los seguidores estarán de apoyo al robot líder siguiéndolo en todo momento en convoy, debiendo preservar la distancia necesaria para no chocarse nunca con el robot que tienen delante. Debido a que no hay suficientes sensores de ultrasonidos, los robots sólo pueden controlar al robot u objeto que está inmediatamente delante de ellos, sin poder fijarse en el que está detrás, lo cual sería muy útil en movimientos marcha atrás.

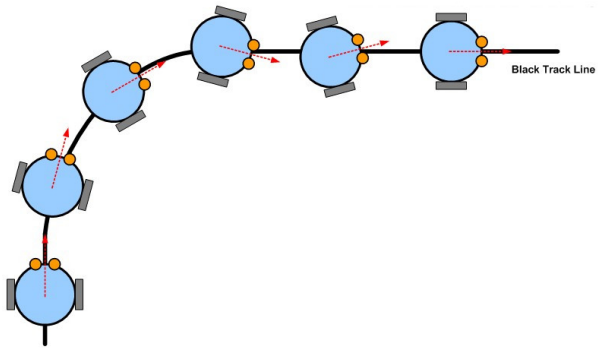

Figura 1: Seguimiento de línea con enderezado

4. Arquitectura

Debido a las características propias de un robot móvil, su programación debe tener en cuenta que se van desenvolver en un mundo estocástico y por tanto, es muy importante captar la información del entorno, procesarla y decidir qué ley de control aplicar en cada momento teniendo en cuenta la seguridad del sistema y del entorno, teniendo en cuenta que los sensores proporcionan información ruidosa. Este hecho se agrava al tratar con un hardware tan limitado. Es por ello que hemos desarrollado una arquitectura o *framework* que es modular y que cumple todas estas restricciones. Además, permite conectarse con los robots reales, o simularlos emulando las entradas de los sensores para así poder probar la lógica del programa del robot sin tener el real. Hay que tener en cuenta que el sistema está definido para soportar a un robot líder y a múltiples robots seguidores.

La arquitectura se basa en la creación de una clase llamada robot (robot.py), de la cual se creará una instancia que contendrá las variables y métodos para el control de cada robot y de sus sensores (ya sea en modo real o simulado). Para la lógica se utiliza una máquina de estados, que será el c̈erebrodel robot. En otro fichero (sensors.py), se encuentran las funciones de lectura de cada uno de los sensores, que se encargan de leer el sensor y comprobar que los valores de cada sensor están dentro del rango adecuado. Este fichero contiene además las funciones necesarias para el procesado de la información de cada sensor, debido que al robot en sí no le interesan los datos del sensor en crudo o raw, si no, la información de alto nivel que se deriva del sensor.

Por otro lado, en otro fichero (controllers.py) se recogen las funciones de control que usará el robot para moverse por el entorno. Todos estos módulos del *framework* son explicados en mayor profundidad en los siguientes apartados. Debido a que el robot debe compartir información con las funciones de sensores y control (que pueden variar dependiendo del robot) se han creado clases auxiliares (similares a los structs de C/C++) que guardan las variables de configuración del robot (velocidad máxima, tiempo de refresco de lectura de sensores, ect), de lectura y procesado de sensores, y de acciones de control. Estas clases se llaman *Config*, *Data*, *Information* y *Actions* respectivamente.

Para hacer un código más claro, se ha creado un fichero de configuración y descripción de las características del robot y de los sensores (description_constants.py) que permite definir las variables globales del robot, como los estados del robot, de la lectura de los sensores (como los rangos, entre otras), las datos de alto nivel que van a devolver los sensores, y las constantes del control.

La creación de cada robot se hace mediante un código principal (main.py). Este código hace uso de *multithreading*, debido a que cada robot no puede ejecutarse de forma secuencial, si no de forma paralela, ya que es posible añadir tantos robots seguidores como sea necesario.

5. Robot

Como ya se ha mencionado en la arquitectura, la clase robot se encarga de gestionar las funciones del robot. Debido a los diferentes sensores que puede tener cada robot así como su rol dentro de la organización, cuando se crea una instancia de esta clase recibe como parámetros:

- El nombre del robot para diferenciarlo de los demás robots.
- El modo en el que se va a ejecutar (simulado o real).
- La ruta *bluetooth* a la que está conectado.
- El rol (líder o seguidor).
- Una lista de sensores.
- Una lista del puerto de cada sensor.

Una vez creada la instancia, el método lee la lista de sensores y de puertos y los añade a una variable de instancia para poder llamar de forma eficiente a todos los sensores y a todas las funciones de procesado. Eso permite tener un código más limpio dentro de la clase.

Por otro lado, el método *run_main* se encarga de llamar a todas las funciones necesarias de forma secuencial: Lectura de los sensores, procesado de la información, actualización de la máquina de estados, ley de control, ejecución de la acción, refresco de la interfaz de usuario. Esta función contendrá un bucle infinito sin *delays* para que se ejecute lo más rápido posible. Para las funciones que necesitan ejecutarse en tiempos específicos se han diseñado temporizadores internos.

En el caso de que la clase detecte un error en la inicialización de la clase, mandará el robot al estado de emergencia y no podrá salir de ahí a menos que se reinicie el robot.

5.1. Máquina de Estados

Tal como se ha explicado en el apartado anterior, la lógica de los robots se ha diseñado mediante máquinas de estados. Estas máquinas son las encargadas de gestionar los sensores, el movimiento y los estados de los robots. Este tipo de diseño permite conseguir una arquitectura de control más robusta, fácil de entender a simple vista y escalable en cualquier punto del desarrollo, permitiendo añadir nuevas funcionalidades adaptando mínimamente la lógica de las condiciones ya existentes.

En nuestro caso pasamos por dos iteraciones importantes en el diseño de estas. En un primer lugar, fueron concebidas de tal forma que ambos robots (líder y seguidor) fueran independientes el uno del otro. Sin embargo, este sistema debido a las limitaciones del hardware de detección resultó ser más complejo de implementar en la práctica pues era necesario conocer con mucha precisión y no perder al líder de vista. En la segunda iteración, dicha complejidad fue eliminada gracias al uso del *multithreading*, el cual permite a ambos robots se ejecuten a la vez y accedan a las mismas variables globales. De este modo la máquina de estados del seguidor se vale de la variable *LEADER_STATE_INFORMATION* para conocer en todo momento qué está haciendo el líder y así actuar en consecuencia. Destacar que en este diseño, tenemos un único líder, y a su vez un número indeterminado de posibles seguidores. En todo momento, sea cual sea el tipo de movimiento, habrá un controlador encargado de seguir la línea y evitar que el robot se salga de la misma. A continuación de las figuras correspondientes a las máquinas de estado, explicaremos los distintos comportamientos y pasos por los que pasan las máquinas.

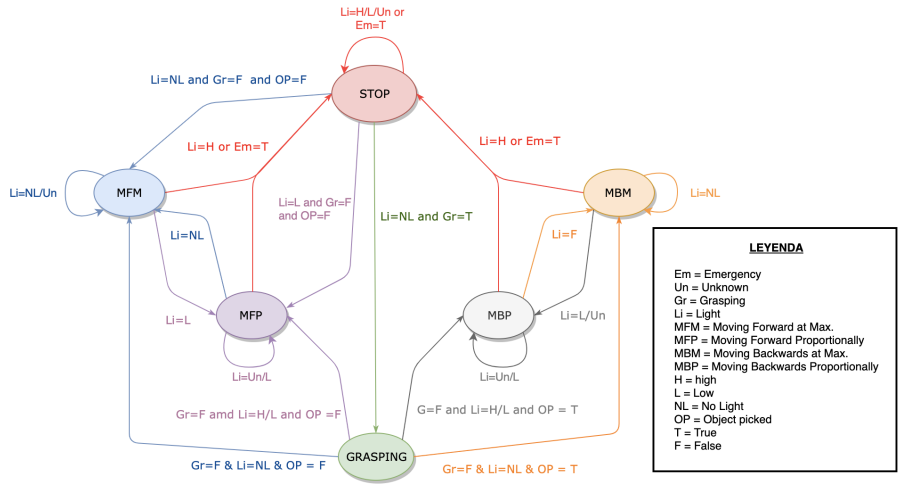

Figura 2: Máquina de estados del robot líder

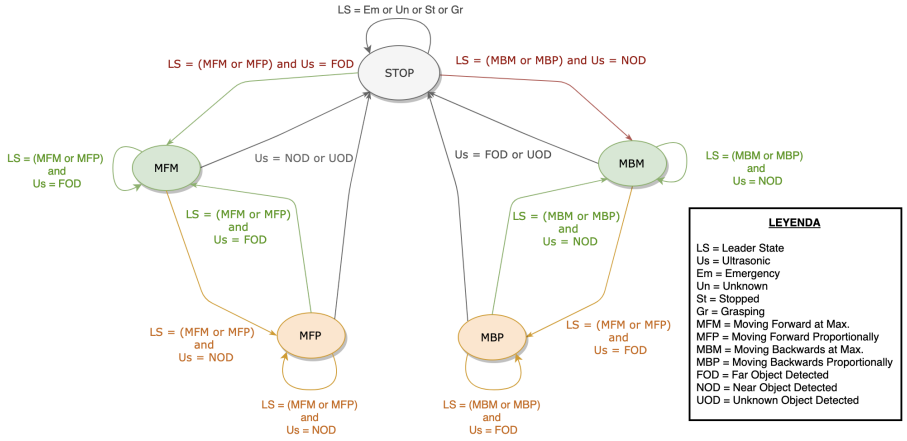

Figura 3: Máquina de estados de un robot seguidor

Todas las máquinas comienzan en el estado *Stop* y permanecerán en este siempre que haya luz, un error en el sensor o una emergencia. En el caso del *follower* tendremos en cuenta el estado en el que se encuentre el líder: emergencia, parada, indeterminado o cogiedo un objeto (*grasping*).

Una vez tengamos una variación en la lectura del sensor de luz del líder que indique que ïo hay luz¨(que la intensidad de esta, no supere el umbral establecido) y no tengamos un objeto cogido, pasaremos al estado MFM (*Moving Forward at Max. speed*). En cuanto al *follower*, este comenzará a moverse cuando se detecte que el líder se ha movido más allá de su distancia de lectura máxima (FarObjectDetected) sin importar si el líder está en MFM o MFP. En ambos casos nos mantendremos en este estado siempre que dichas condiciones no cambien.

El siguiente estado principal será el de MFP (*Moving Forward Proportionally*) en el cual reduciremos nuestra velocidad de forma directamente proporcional a la detección de nuestro sensor (Luz para el líder y Ultrasonidos para el *follower*). Accederemos a este estado con el líder cuando la luz comience a detectarse y sea baja. En el caso del seguidor, cuando la distancia con el líder esté entre el *threshold* máximo (FOD) y el mínimo (NearOD) sin importar si el líder está en MFM o MFP. De nuevo siempre que estas sean las condiciones, permaneceremos en dicho estado.

Podremos volver al estado *Stop* desde MFM o MFP si detectamos que el nivel de la luz es alto (modificando *grasping* a *True*) o tenemos una emergencia. En el caso del *follower*, cuando detectemos un NOD o un UnknownOD.

Una vez dejemos de detectar la luz y estando *grasping* en *True*, pasaremos a coger el objeto. Una vez lo tenemos cogido, pasaremos a movernos hacia atrás. Mientras tanto el *follower* permanecerá en

Stop, pues habrá llegado a NOD.

Podremos pasar a MBP (*Moving Backwards Proportionally*), si, teniendo el objeto, hay luz entre los límites establecidos. En el caso del *follower*, pasaremos a este estado cuando el Líder se halle en MBP o MBM y el sensor de ultrasonidos capte un FOD. De nuevo siempre que estas sean las condiciones, permaneceremos en dicho estado.

Pasaremos a MBM (*MB at Max. speed*) cuando no detectemos luz y en el caso del *follower*, cuando se detecte un NOD. De nuevo siempre que estas sean las condiciones, permaneceremos en dicho estado.

Podremos volver al estado *Stop* desde MBM o MBP si detectamos que el nivel de la luz es alto (modificando *grasping* a *True*) o tenemos una emergencia. En el caso del *follower*, cuando detectemos un FOD o un UnknownOD y el líder ya no esté en MBM/P.

De nuevo, una vez dejemos de detectar la luz y estando *grasping* en *True*, pasaremos a dejar el objeto. Una vez lo hayamos depositado, pasaremos a movernos hacia delante en función del nivel de detección o no, de luz. Mientras tanto el *follower* permanecerá en *Stop*. De este modo cerramos el ciclo y pasamos por todos los posibles estados.

6. Sensores

Para el tratamiento de la información que recibimos de los sensores utilizaremos las estructuras y la funciones contenidas en el archivo ***sensors.py***. En él se incluyen la librería ***Aurigapy***, con la que podemos controlar las señales y los movimientos del robot, la librería ***time*** para controlar los tiempos de envío y recepción de los sensores y el archivo ***description_constansts***, con la declaración y asignación de valores de todas las constantes que se usan en los *scripts*.

A continuación, podemos observar la **clase *Data***, que contiene todos los datos que se tienen sobre el entorno. Respecto a los métodos, sólo consta de un constructor, en el que se le asigna un valor que informa del desconocimiento que se tiene acerca del valor de dichos valores. Los atributos de la clase son:

- **ultrasensor_distance**: distancia (en cm) proporcionada por el sensor de distancia. ¨1ïsupone un valor indeterminado o de error y está almacenado en la variable IMPOSSIBLE_DISTANCE, definida en *description_constansts*
- **light_sensor_value**: intensidad de luz recibida por el sensor de luz, entre 0 y 1024
- **line_detection**: Detección de línea. Un 0ïmplica que ambos sensores detectan, un ¨1ïque sólo el de la izquierda lo hace, un 2ïsólo el de la derecha y un 3ï; ninguno. De nuevo, ¨1ïsupone un valor indeterminado o de error. Estos valores están definidos en *description_constansts*
- **current_time_ultrasonic**: *timestamp* de la lectura actual del sensor de distancia
- **previous_time_ultrasonic**: *timestamp* de la última lectura realizada por el sensor de distancia

Todos estos datos son captados por las funciones de lectura de datos (desde la línea 34 hasta la 142) y posteriormente procesados con las de procesado (desde la línea 143 en adelante). Tras el procesado de datos entra en juego la **clase *Information***, que almacena la información de alto nivel que se deduce de los datos captados. De nuevo, sólo se tiene un único método, el constructor, que da valores a los atributos:

- **ultrasensor_detection**: indica si no se detecta objeto o si se detecta lejos, cerca o muy cerca (advierte colisión). Se define cada una de las situaciones en función a umbrales. También cabe la posibilidad de que almacene valores indefinidos si ha habido algún error (contemplado en el *script*) o si aún no se le ha asignado valor
- **light_detection**: señala la cantidad (abstracta) de luz detectada, según umbrales de configuración. ésta puede ser alta, baja, nula o desconocida

A continuación, a lo largo de los siguientes subapartados, explicamos los método de lectura y procesado de los sensores.

El **esquema seguido en todas las de lectura** es muy similar. Tras unas pocas líneas donde explicamos los parámetros de entrada y el valor de retorno, se evalúa si se está en modo simulación o real. En caso de darse la primera opción, se le permite al usuario introducir un valor de entrada y, si se está en el segundo, se procede a leer el sensor. Luego se procede a comprobar que no hayan habido errores, considerando que ha sucedido uno si se tiene un valor fuera de los límites del sensor configurados o si no se estuviese ni en modo simulación ni en modo real. Todas estas funciones devuelven *True* si no ha habido ningún problema y *False* en cualquier otro caso (ERROR). También, como se ha mencionado antes, le asignan la lectura a la componente correspondiente del atributo tipo *Data* de la instancia de la clase *Robot* desde la que se ha llamado al método.

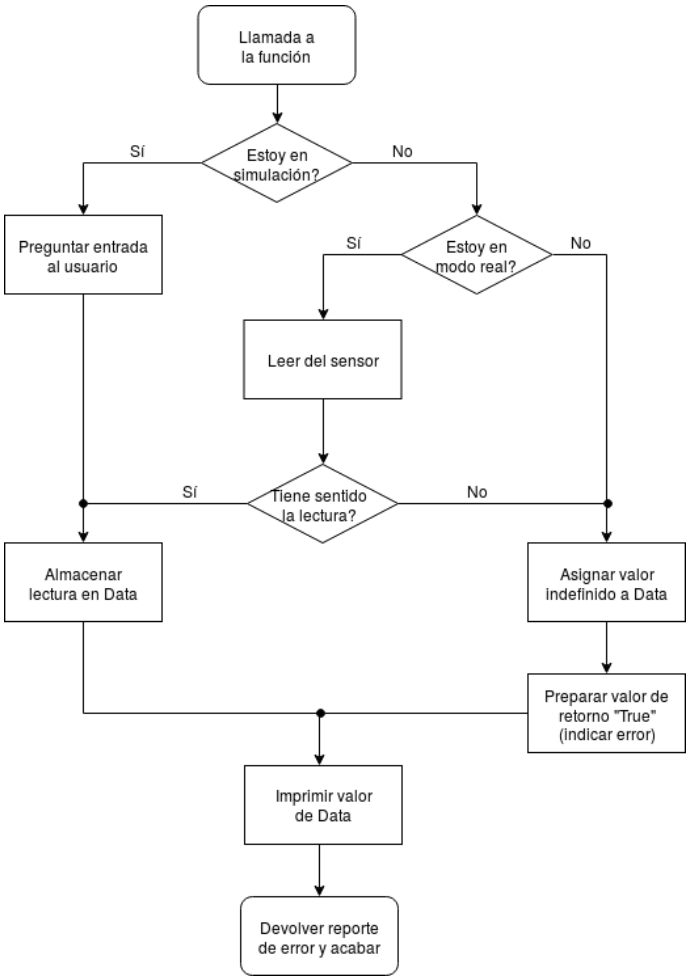

Figura 4: Diagrama de flujo de una función de lectura genérica

Consideramos que éste era el **enfoque más adecuado**, ya que no sólo permite hacer pruebas sin los robots reales, sino que informa de todos los posibles errores de forma general (con el valor de retorno) y de concretando el fallo con los valores de las variables (indicando que X sensor tiene un valor indeterminado), permitiendo desarrollar comportamientos de recuperación o detección de fallos. Del mismo modo, el uso de las clases *Data* e *Information* supone una enorme modularidad y capacidad de escalado, ya que podemos llevar un *tracking* independiente de las lecturas de cada robot que incluyamos en el sistema, así como de las conclusiones de alto nivel que se extraen, pudiendo configurar umbrales distintos para cada uno de ellos o analizar la situación global si así fuese necesario.

6.1. Sensor de distancia

Este sensor se debe leer pasado cierto tiempo respecto de la anterior lectura pues la frecuencia a la que se recibe nueva información es menor a la de lectura del sensor.

La función ***readUltraSensor*** sigue la siguiente lógica: una vez se ha esperado el tiempo suficiente para consultar de nuevo el sensor este se lee y la información se almacena directamente en la clase *Data*, esta función tiene en cuenta casos donde el sensor ofrece datos no válidos.

Posteriormente, se dispone de la función ***processUltrasonicSensorData***, donde se procesan los datos captados para extraer información de alto nivel.

6.2. Sensor sigue-líneas

La información de este sensor es la más fácil de extraer ya que los valores que obtenemos del dispositivo no necesitan procesamiento. Si es 0, significa que detectamos en los dos sensores del sigue-lineas, si es 1, es que el izquierdo está *On* y el derecho *Off*; 2 si es a la inversa y 3 si no detectamos en ninguno. Nosotros incluímos la posibilidad de que se le asigne 1", en caso de error o indeterminación.

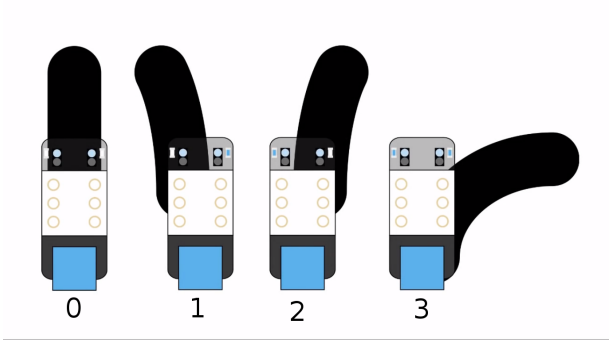

Figura 5: Lecturas del sensor de línea en varias situaciones

La función ***readLineSensor*** comprueba el modo de funcionamiento del programa (simulación o robot real), actualiza los datos y filtra si recibimos algún valor erróneo, asignando entonces un valor declarado como desconocido y devolviendo *False* para indicar el fallo.

6.3. Sensor de luz

Con la función de lectura ***readLightSensor*** podemos leer la información en crudo del sensor de luminosidad para después actualizar la instancia de la clase *Data*, al igual que en el resto de funciones de lectura.

También disponemos de la función ***processLightSensorData*** se limita a valorar los datos disponibles, trasladándolos al atributo correspondiente de *Information* con las constantes *HIGH_LIGHT_DETECTED*, *LOW_LIGHT_DETECTED*, *NO_LIGHT_DETECTED* y *UNKNOWN_LIGHT_DETECTED* que indican de forma abstracta la cercanía a un foco de luz (que indicará la presencia del objeto)

7. Controladores

Para estructurar los datos de movimiento de nuestro robot hemos creado una clase llamada *Actions* donde guardamos la velocidad de los motores, tanto en el giro como en línea recta, el último sentido de movimiento del robot y la posición de la pinza. Trabajaremos sobre esta clase en las funciones de control.

En primer, lugar hemos diseñado una función llamada *customSpeeds*, a la cual le proporcionamos las velocidades que queremos que tengan los motores y esta, devuelve el resultado de concatenar una secuencia hexadecimal con dichas velocidades.

Para manejar los eventos del sigue-lńeas, hemos creado la función *lineFollower*, la cual recibe el sentido de movimiento que queremos tomar, el valor obtenido del sigue-lńeas, una velocidad base lineal, una para los giros y por último, el último sentido de movimiento del robot. Si el sentido de la dirección que queremos seguir, es hacia delante (ya sea girando o en línea recta) o hacia atrás, establecemos los giros y los asignamos a unas variables auxiliares para llevar un control más claro, ya que uno de los motores está invertido, por lo que la velocidades serán las opuestas del otro. A continuación, en función de la información del sigue-lńeas y de la último sentido de movimiento, usando la función *customSpeeds*, ajustaremos la velocidad de cada motor para que el robot se mueva en el sentido deseado. En el caso de los giros, para evitar la inestabilidad y que comience a zigzaguear, se compensará de forma muy sutil, con una diferencia de 5 rpm en la velocidad de ambos motores.

Además de la función concreta de la aplicación del robot sigue-lńeas, hemos creado unas funciones de control genéricas que pueden utilizarse en varios entornos y que aplican los cambios directamente al objeto robot. En primer lugar, tenemos una función de parada, otra de avanzar a máxima velocidad y una última de retroceder, también a máxima velocidad. Estás 3 funciones actualizan el estado de *Actions* del robot directamente, con la velocidad máxima que hayamos establecido. Pero, además de estas, hemos creado dos funciones para el movimiento proporcional del robot. En el caso del robot líder, esta proporcionalidad irá definida por la intensidad de la fuente de luz, cuanto mayor sea, más lento avanzará o retrocederá y en el caso del seguidor, en función de la distancia. Si con el ultrasonidos detectamos que se aleja el robot de delante, aumentaremos la velocidad, si se detectase al contrario, sería disminuida. Por último, dispondremos de una función para abrir y cerrar la pinza en un intervalo de unos 4 segundos.

8. Trabajos Futuros y Conclusiones

Como trabajos futuros queda hacer totalmente funcional el *framework*, debido a que por errores en los controladores este no ha sido capaz de funcionar. Mejorar la gestión interna de errores, estandarizar la interfaz de usuario, añadir una interfaz visual con los leds para saber en qué estado se encuentra el robot sin tener que mirar el terminal y obtener de forma experimental el número máximo de robots seguidores que es capaz de soportar el *framework* (aunque parece evidente que dependerá de las capacidades de cómputo del PC).

Las conclusiones que hemos extraído han sido la validación de que aunque un programa funcione en simulación, debido al hardware y a la complejidad del mundo real, es muy posible que en estos escenarios no funcione y hay que hacer modificaciones. Por otro lado, la alta complejidad de la programación de un robot y sobre todo un robot móvil hace que si uno de los 3 módulos (arquitectura, sensorización y control) falla el robot no sea capaz de cumplir la tarea adecuadamente. En nuestro caso se ha debido al modulo de control, que no ha podido estar totalmente listo para la fecha límite del trabajo. Por otro lado, aunque en simulación funcionara la lectura y filtrado de sensores, así como la lógica de la máquina de estados, el no poder probarlo en el robot real porque el controlador no estaba listo a tiempo ha hecho que pequeños detalles (pero que a la vez son muy importante) no hayan podido ser testados, como los *thresholds* de distancia a los objetos y otras variables importantes a la hora de programar la lógica de un robot.