UNIVERSIDAD DE VALLADOLID

ESCUELA DE INGENIERIAS INDUSTRIALES

Grado en Ingeniería en Electrónica Industrial y Automática

Control de robots autónomos mediante microcontrolador Arduino

Autor:
Misiego López, Ana

Tutor 1:
Zalama Casanova, Eduardo
Tutor 2:
Gómez García-Bermejo, Jaime

Valladolid, Septiembre 2015

Agradecimientos:

RESUMEN

El principal objetivo de este proyecto es realizar el control de bajo nivel de un robot móvil educacional. El control se ha realizado mediante un sistema distribuido de tres microcontroladores al que se han conectados los diferentes actuadores y sensores.

Para evitar el choque contra obstáculos, se ha realizado la lectura de los ultrasonidos, bumpers, además de controlar el estado de la batería. Después, a partir de la lectura de los codificadores incrementales, se ha generado un control PID para obtener un movimiento de los motores adecuado y dando la consigna adecuada de velocidad a los amplificadores.

Para la comunicación entre los microcontroladores se ha realizado una comunicación tipo I2C y una tipo Puerto Serial para la comunicación entre bajo y alto nivel.

Finalmente, se ha realizado un módulo ROS (Robot Operating System) de alto nivel de gestión de dispositivos lo que permite la utilización de una gran variedad de programas de control, navegación y planificación de software abierto con el robot.

Palabras clave: Robótica. Control. Arduino. Automática. Electrónica.

Contenido

1. INTRODUCCIÓN

1.1. Marco del proyecto

El objetivo del presente proyecto es el desarrollo de un control de bajo nivel para una plataforma robótica móvil educacional, el desarrollo de un sistema de gestión de dispositivos de alto nivel en ROS (Robot Operating System) para que junto con el sistema de comunicación se puedan utilizar la gran variedad de programas de software libre de navegación, planificación y control disponibles en ROS.

Cabe destacar que se trata de una plataforma robótica desarrollada con el objetivo de ser utilizada en ámbitos de enseñanza e investigación y así no tener la necesidad de adquirir otro tipo de plataformas robóticas ya existentes en el mercado de elevado coste y con pocas posibilidades de mejora y ampliación. Por estos motivos, el robot Edubot U7 fue diseñado de forma genérica, siendo capaz de desempeñar una gran diversidad de funciones y facilitando de este modo el posterior desarrollo sobre el mismo.

Cuando se habla de un diseño genérico, se engloban todo tipo de características, desde los rasgos físicos y mecánicos hasta las características relacionadas con el control del robot. Actualmente, el mismo está caracterizado por una serie de sensores y actuadores cuyo control puede ser modificado y mejorado, siendo posible la incorporación de más tipos de sensores y actuadores con el fin de que futuros estudiantes puedan desarrollar sus conocimientos sobre la ingeniería en el mismo. Además, todas las plataformas de control utilizadas en el robot han sido y son compatibles con futuras ampliaciones, siendo todas ellas, obviamente, de software libre.

El uso de software libre en una plataforma robótica destinada a la enseñanza y la investigación resulta de vital importancia, ya que de usar un software propietario, las posibilidades de aprendizaje, mejora y ampliación se verían notablemente reducidas, además de los inconvenientes económicos, de ahí, la elección del uso de microcontroladores Arduino.

Además de ser de software libre y ser muy asequible económicamente hablando, Arduino tiene otras muchas ventajas. Es multi-plataforma, es decir, es capaz de funcionar en gran número de entornos cuando otro tipo de microcontroladores similares a Arduino sólo son capaces de funcionar en Microsoft. Gracias a esto, se podrá trabajar en entornos como Linux y ROS, donde este último, ofrece multitud de herramientas y librerías para el desarrollo y creación de aplicaciones para robots.

Tiene la capacidad de ser ampliable, es decir, si fuera necesaria la introducción de más microcontroladores, esto podría hacerse sin problemas, pudiendo comunicar los mismos entre ellos. Además de ser de software libre, es de hardware libre, facilitando el desarrollo de estas placas de forma que actualmente cuentan con una gran gama de productos, pudiendo elegir el que mejor se adapte a las condiciones de trabajo.

Por todos estos motivos, se ha considerado el uso de microcontroladores Arduino para el control de bajo nivel del presente proyecto.

Además del carácter educativo del robot Edubot, al tratarse de un robot móvil muy genérico, sus posibles aplicaciones finales son inmensas.

Este tipo de robots son capaces de suplir algunas de las limitaciones con las que se puede encontrar el ser humano, pudiendo desplazarse por todo tipo de terrenos por los cuales una persona no puede o puede por un tiempo limitado. Este tipo de terrenos pueden ser desde entornos no aptos para el ser humano (radiactivos, excesivamente fríos o calientes, sin oxígeno…) hasta lugares de muy difícil acceso que entraña grandes peligros para la vida humana (incendios, derrumbamientos, terremotos…). Además de todas las aplicaciones anteriormente citadas, existen otras muchas como la robótica médica, capaz de realizar intervenciones muchos menos invasivas y más precisas que las tradicionales, la robótica orientada al transporte, a la construcción, a la seguridad, a la producción, al entretenimiento, entre otras muchas.

Teniendo en cuenta todas estas posibles aplicaciones, se puede deducir la importancia del desarrollo de este tipo de tecnología de manera correcta, ya que la misma puede salvar vidas, facilitar el día a día del ser humano e incluso realizar importantes descubrimientos. También se puede deducir la dificultad del desarrollo de todos estos robots, puesto que muchos de ellos deben ser muy precisos y deben ser capaces de desenvolverse en entornos muy hostiles donde se pueden encontrar con cualquier tipo de obstáculo y contratiempo que deben ser capaces de subsanar.

1.2. Objetivos

El objetivo global del presente proyecto es el desarrollo del control de bajo nivel del robot Edubot, de forma que el mismo pueda comunicarse con el control de alto nivel, ROS, el cual le irá indicando a qué velocidad debe moverse cada motor en función de la información recogida y enviada de los diferentes sensores.

Para poder conseguir este objetivo final, será necesario el desarrollo de una serie de subobjetivos de manera independiente, los cuales se citan a continuación:

- Estudio previo del robot Edubot para conocer en profundidad todos y cada uno de los elementos que lo forman y así posteriormente poder utilizarlos correctamente.
- Desarrollo del nuevo diseño electrónico con la incorporación de microcontroladores Arduino y su correspondiente nuevo cableado.
- Lectura de los ultrasonidos implantados en el robot, de forma que se conozca en todo momento la distancia a la que están los diferentes obstáculos que se puede ir encontrando a lo largo de su trayectoria, y así, evitarlos.
- Lectura de los bumpers situados en la parte delantera y trasera del robot, de manera que se sepa cuando se ha producido una colisión, pudiendo cortar la alimentación a los motores como medida de seguridad.
- Lectura de la carga de la batería, gestionando una serie de alarmas en función de la misma y representado este valor de manera visual gracias a una serie de leds.
- Control de la velocidad de los motores, para que los mismo giren de manera uniforme y constante. Para ello, será necesario:
    - Lectura de los codificadores incrementales, a partir de la cuales, se conocerá la velocidad a la que gira el motor.
    - Uso adecuado de los puentes H, de forma que se gire en el sentido correcto los motores.
    - Desarrollo de un control PID, con el que se conseguirá alcanzar la velocidad adecuada y mantenerla a lo largo del tiempo.
- Diseño de la comunicación I2C y RS232 entre los diferentes microcontroladores, de forma que puedan intercambiar información entre ellos, siendo necesaria la integración de un sistema distribuido jerarquizado de los mismos.
- Compatibilizar este sistema con el control de alto nivel ya desarrollado, ROS, de forma que pueda comunicarse con él, siendo necesario aplicar las tramas de comunicación pertinentes. El robot deberá informar al control de alto nivel, en unos periodos de tiempo fijados, de su estado, de forma que dicho control pueda conocer:
    - La distancia a la que se encuentra un obstáculo.
    - Si el robot ha chocado.
    - El nivel de carga de la batería.

- La velocidad a la que se mueven los motores y las vueltas dadas por cada rueda.

Gracias a toda esta información, ROS sabrá como deberá reaccionar el robot en función de todas estas variables, indicándole la velocidad más adecuada para cada motor. Además, ROS es capaz de conocer la localización del robot en cada instante, gracias a la localización odométrica desarrollada.

1.3. Estructura de la memoria

Conocidos todos los objetivos presentes en el proyecto, se pueden diferenciar claramente todas las partes que forman la memoria.

A modo resumen, la memoria está formada por los siguientes capítulos:

- **Capítulo 1. Introducción.** Es el capítulo dónde se realizará una introducción al robot Edubot, indicando las razones de su creación, los objetivos necesarios para superar el proyecto con éxito y adjuntado un breve resumen de toda la memoria.
- **Capítulo 2. Diseño previo.** En este capítulo se explicarán las bases de las que parte este proyecto. Se desarrollará tanto el diseño mecánico, explicando la forma del chasis, la configuración cinemática elegida para el robot y los motores utilizados, como el diseño electrónico, donde se explicará la elección y el funcionamiento de los sensores y actuadores utilizados.
- **Capítulo 3. Diseño actual.** A lo largo del capítulo se explicará el proceso de adaptación del robot a los microcontroladores Arduino, indicando el nuevo cableado realizado y el control que deberá realizar cada microcontrolador. Además, se adjuntará una imagen donde podrá verse todo el montaje físico completo.
- **Capítulo 4. Control del robot Edubot.** En este capítulo se desarrollará todo el proceso de control y lectura de los sensores, actuadores, motores y localización odométrica en gran detalle, explicando el código escrito y su evolución y adjuntado a los mismos unos diagramas de flujo con el fin de facilitar la comprensión del control.
- **Capítulo 5. Comunicación del Arduino Mega.** En este capítulo, se explicará cual ha sido el método de comunicación del microcontrolador principal con el resto de dispositivos, es decir, entre el resto de microcontroladores y el control de alto nivel.

- **Capítulo 6. Programas de comprobación del robot Edubot.** Para poder comprobar muchos de los controles desarrollados en el capítulo 4, fue necesaria la creación de una serie de programas de control de alto nivel con el fin de validar los códigos escritos en Arduino. A lo largo de este capítulo, se desarrollarán todos estos programas.
- **Capítulo 7. Resultados experimentales.** En este capítulo se indicarán todos los resultados obtenidos a lo largo del desarrollo del proyecto.
- **Capítulo 8. Estudio económico.** En este capítulo, se realiza un estudio del coste económico que ha tenido todo el proyecto, teniendo en cuenta tanto los costes directos como los indirectos.
- **Capítulo 9. Conclusiones.** El último capítulo está destinado a las conclusiones obtenidas tras la realización del proyecto, sugiriendo una serie de posibles mejoras.

2. DISEÑO PREVIO

2.1. Introducción

Aunque el presente proyecto se fija en la integración de la plataforma Arduino sobre el robot Edubot, es de gran importancia conocer el trabajo previo realizado sobre el mismo *(Verdejo.D (2011))*.

Gracias a la plataforma Arduino, es posible el control de diferentes tipos de sensores con los cuales podemos lograr el correcto funcionamiento del robot Edubot, moviéndose por diferentes tipos de superficies sin que choque contra ningún obstáculo.

Para ello, ha sido necesario el diseño y construcción del robot físicamente, es decir, el diseño del chasis y la elección de los materiales idóneos para su posterior construcción. A continuación, teniendo en cuenta el objetivo didáctico y de investigación que tiene, se tuvieron que elegir los componentes más adecuados para poder poner en funcionamiento el chasis ya diseñado y construido. Para ello, se tuvo que elegir entre numerosos tipos de piezas, tanto mecánicas como electrónicas, y realizar la configuración idónea de las mismas para que el robot funcione correctamente.

Sin un estudio previo de todo este trabajo, no será posible el correcto control de todos los sensores de los que se disponen y por consiguiente, el robot ya diseñado no podrá ser controlado de manera correcta. Además, sin este proyecto previo, el trabajo que ha sido desarrollado en la actualidad no sería posible. Por estos motivos, a lo largo de este capítulo se va a abordar parte del trabajo elaborado previamente.

En primer lugar, se hablará sobre el diseño mecánico, explicando el diseño del chasis, la configuración cinemática y el tipo de motores que se encuentran en el robot Edubot. Posteriormente, se desarrollará el diseño electrónico del robot, explicando los diferentes tipos de sensores que se encuentran en el mismo y su sistema de alimentación.

Todo lo que se va a explicar a lo largo de este capítulo no ha sido modificado, es decir, se ha trabajado sobre el diseño mecánico y electrónico ya desarrollado.

2.2. Diseño mecánico

En primer lugar, es importante tener presente que se trata de un robot de carácter general, por lo que a la hora de diseñarlo, se tuvo en cuenta la posibilidad de realizar sobre el mismo diferentes tipos de mejoras y la integración de nuevas aplicaciones. Una de estas mejoras que se tuvieron en cuenta a la hora de realizar el diseño fue la posibilidad de intercambiar el tipo de rueda a colocar, ya que aunque en la actualidad el robot no pueda circular por terrenos irregulares, se podrían colocar en el mismo otro tipo de ruedas que sí le permitan circular por este tipo de localizaciones.

Hay que tener en cuenta, que el robot tendrá que soportar un peso determinado, ya que dentro de él se encontrarán elementos de pesos muy variables que pueden ir desde un pequeño ultrasonido hasta una batería. Además, sobre el mismo se deberán poder colocar cargas, que aunque no se ha diseñado como robot de transporte de cargas deberá soportar el peso de un ordenador portátil.

Parte del diseño mecánico se encarga de la elección y disposición de los motores, los cuales deben ser escogidos con la suficiente potencia como para poder desplazar el peso completo del robot (chasis, sensores, motores, batería, etc…) y de posibles cargas que se coloquen dentro o sobre el mismo, y que además, pueda superar las pendientes que se pueda encontrar en el terreno.

Por último, antes de comenzar a desarrollar en mayor profundidad el diseño mecánico, mencionar que para unir las diferentes piezas del robot se han utilizado siempre que se ha podido las uniones atornilladas, facilitando su montaje y desmontaje y cuando era necesaria una unión más segura y estable, se han utilizado tuercas autoblocantes y arandelas entadas.

Chasis

El chasis es una de las partes principales del diseño mecánico, su diseño es decisivo para el posterior desarrollo del robot. Si el chasis realizado es incorrecto, no será capaz de soportar todos los componentes o incluso, siendo capaz de soportarlos, si el chasis no es el adecuado, el robot no podrá desplazarse de manera adecuada o no se

Tiene una forma robusta y hexagonal. Con una tapadera superior divida en dos, para poder acceder al interior del robot con gran facilidad. No sólo la tapa está dividida en dos, si se observa el interior del cuerpo del robot, se pueden observar dos zonas claramente diferenciadas, una delantera y otra trasera.

La mitad trasera está destinada a la colocación de los motores, encoders, engranajes, batería y circuitería necesaria para el control de los motores y la mitad delantera para las dos placas Arduino. La tapa, se ha diseñado lisa para facilitar la colocación del ordenador portátil o de cualquier otro tipo de carga. En los laterales, irán colocados de manera simétrica diferentes tipos de sensores para facilitar la geolocalización del robot.

Dicha distribución se ha realizado con el fin de mantener un determinado orden y para equilibrar el peso del robot, aunque la parte trasera sea algo más pesada. Debido a esto último, para que el robot pueda desplazarse de manera correcta, se colocará la batería lo más centrada posible, para evitar que una rueda tenga mayor agarre con el suelo que la otra, dando lugar a un posible desvío del robot cuando sus ruedas comiencen a girar.

En la Figura 1, se puede observar la planta y el alzado del chasis.

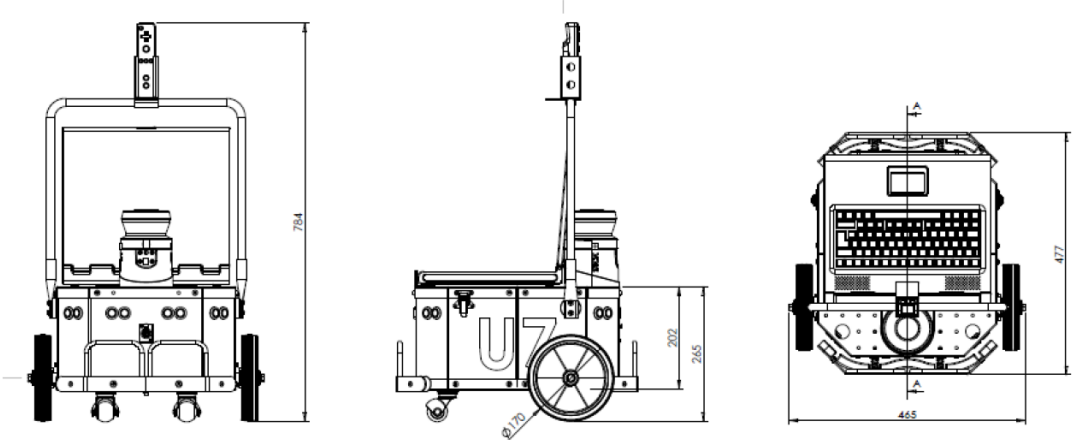


Figura 1. Chasis en 2D.

Configuración cinemática

En primer lugar, se van a introducir una serie de conceptos elementales para poder entender las ventajas y desventajas de los diferentes tipos de configuraciones cinemáticas y así comprender la elección que se a la hora de diseñar el robot.

Existen diferentes tipos de ruedas:

- Rueda motriz, es la que transmite un esfuerzo de tracción al suelo, haciendo posible el movimiento del vehículo.
- Rueda directriz, son ruedas de orientación controlable.

- Ruedas fijas, no tienen tracción, es decir, no están controladas, pero sólo pueden girar entrono a su eje.
- Ruedas locas o ruedas de castor, son ruedas no controladas y que pueden girar en cualquier dirección, es decir, son orientables.

El centro instantáneo de rotación, es el punto de intersección de todos los ejes de las ruedas, y por tanto, el centro de giro del robot.

El último concepto previo a introducir, antes de explicar los diferentes tipos de configuraciones es el de restricción no holónoma, que implica que el robot puede moverse hacia adelante y hacia atrás pero no lateralmente.

A continuación se van a explicar una serie de configuraciones cinemáticas, entre las cuales se eligió la que actualmente utiliza el robot Edubot.

Configuración diferencial

Se caracteriza por no tener ruedas directrices, por lo que el cambio de dirección se realiza modificando la velocidad de cada rueda. Se trata de una configuración barata y fácil de diseñar, pero por el contrario, es difícil de controlar y requiere de gran precisión para trayectorias rectas. Es muy importante que las ruedas sean lo más parecidas posibles, ya que una pequeña diferencia en el diámetro de las mismas puede suponer grandes desvíos. En la Figura 2, se puede observar un claro ejemplo de este tipo de configuración.

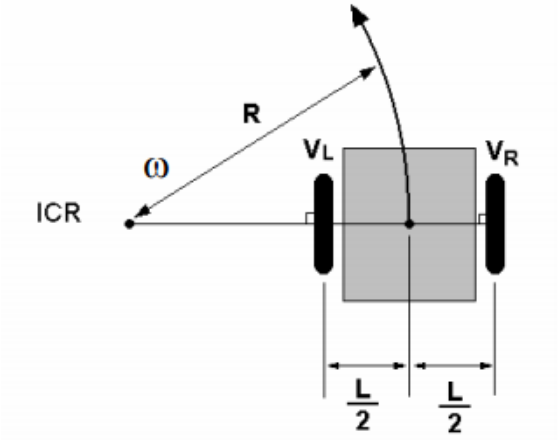


Figura 2. Configuración diferencial.

Configuración síncrona

Todas las ruedas actúan de forma síncrona por un mismo motor, el cual define la velocidad de las mismas. Hay un segundo motor, que estable la orientación de las ruedas. La existencia de dos motores, uno para la translación y otro para la rotación simplifican el control del desplazamiento y el desplazamiento en línea recta está garantizado, sin embargo, como se puede observar al describir este tipo de configuración, su diseño y construcción son muy complejos. En la Figura 3, se puede observar un ejemplo de esta configuración.

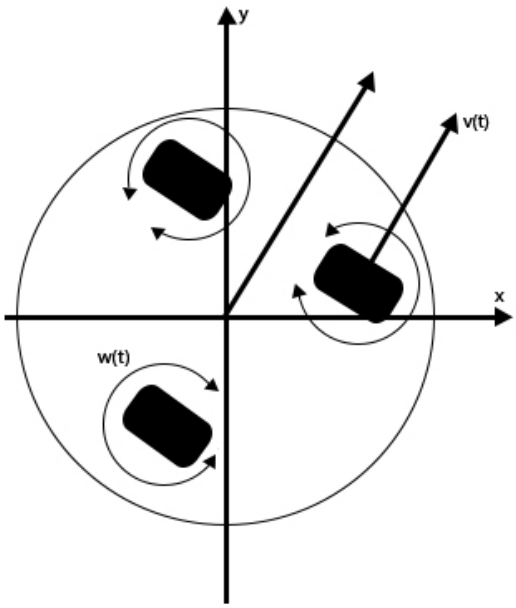


Figura 3. Configuración síncrona.

Configuración triciclo

Al igual que con la configuración diferencial, es un sistema de fácil diseño y construcción, además de que el deslizamiento se ve altamente reducido en este caso. Sin embargo, se encuentra muy restringido a la hora de realizar múltiples movimientos, además de dotar al robot de gran inestabilidad. Para ver visualmente este tipo de configuración véase la Figura 4.

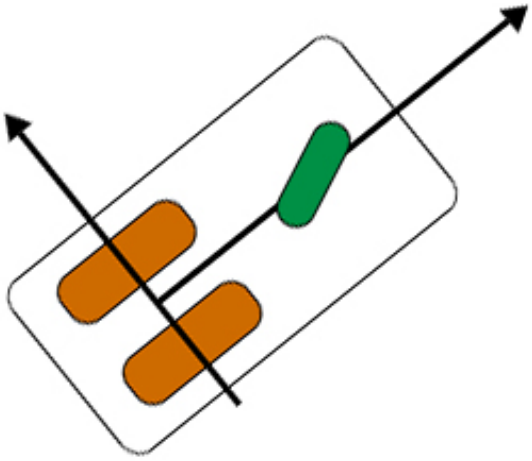


Figura 4. Configuración triciclo.

Configuración Ackerman

Muy frecuente en la industria del automóvil, está caracterizada por estar dotada de cuatro ruedas, dos de tracción trasera y otras dos de dirección delantera. Se trata de una configuración de fácil implementación, con reducido deslizamiento y compatible con altas velocidades, aunque, sus movimientos se encuentran algo restringidos. En la Figura 5 puede verse representado un claro ejemplo de este tipo de configuración.

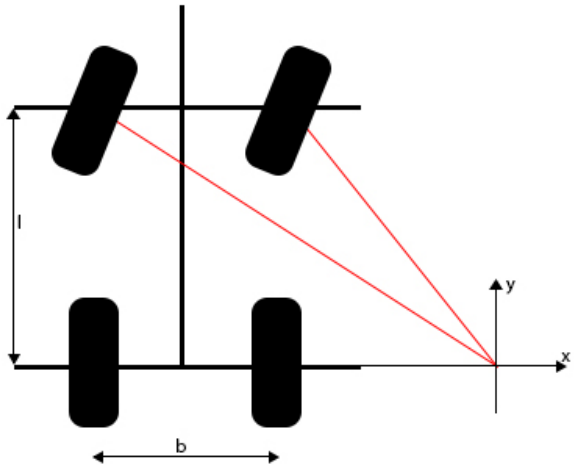


Figura 5. Configuración Ackerman.

Configuración omnidireccional

Son diseños algo complejos pero que tienen una gran ventaja, tienen una gran libertad de movilidad, permitiendo la realización de movimientos complicados gracias a la gran reducción de restricciones cinemáticas. El movimiento en línea recta no está garantizado, siendo necesario un control previo para lograrlo. Véase la Figura 6 para ver un ejemplo de esta configuración.

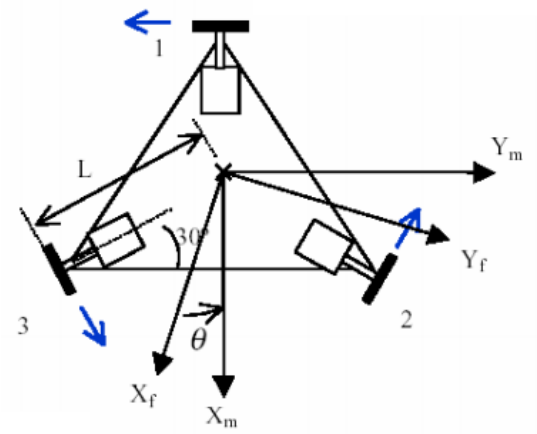


Figura 6. Configuración omnidireccional.

Una vez explicados brevemente los diferentes tipos de configuraciones, debido a su sencillez a la hora de diseñar su configuración, al tener uno de los controles más simples, al ser económicamente rentable y a la facilidad de cálculo para conocer la posición del robot, la configuración elegida es la configuración cinemática diferencial.

Se ha realizado un cuadro resumen, véase Tabla 1, con las características más decisivas a la hora de elegir la configuración idónea, señalando en rojo los aspectos negativos, en naranja los aspectos de gravedad intermedia y en verde las características más adecuadas para el robot Edubot.

En este robot en concreto, se ha elegido una configuración diferencial con dos ruedas motrices, una a cada lado del robot y cuyos ejes estás alineados. Debido a que con dos ruedas, no se puede proporcionar la estabilidad suficiente, se han colocado dos ruedas locas detrás de las ruedas motrices.

		Características		
		Implementación	Coste	Control de posición
Tipos de configuración	Diferencial	Sencilla	Bajo	Sencilla
	Síncrona	Difícil	Elevado	Difícil
	Triciclo	Media	Medio	Sencilla
	Ackerman	Media	Medio	Media
	Omnidireccional	Difícil	Elevado	Difícil

Tabla 1. Tipos de configuración.

En este robot en concreto, se ha elegido una configuración diferencial con dos ruedas motrices, una a cada lado del robot y cuyos ejes estás alineados. Debido a que con dos ruedas, no se puede proporcionar la estabilidad suficiente, se han colocado dos ruedas locas detrás de las ruedas motrices.

Las ruedas motrices se encuentran en la mitad de la parte posterior del robot, de forma que el emplazamiento de los motores no interfiera al de la batería. Además, las ruedas no estarán empotradas a las ruedas, para tener la posibilidad de colocar otro tipo de ruedas en un futuro.

Las ruedas locas, están centradas en la parte delantera del chasis, cuya separación es notablemente menor que la existente entre las ruedas motrices. El motivo de esta separación se encuentra en que se ha contemplado la posibilidad de que alguna de las dos ruedas se quede en el aire. Si esto ocurriera, la otra rueda podría soportar todo el peso delantero, evitando que el robot se inclinará hacia el lado de la rueda que no hace contacto con el suelo.

A continuación, se enumeran las características de la configuración implementada y se visualiza en la Figura 8 la configuración definitiva del robot Edubot:

- Diámetro de las ruedas motrices, 16 cm.
- Distancia entre las ruedas motrices, 42 cm.
- Altura libre al suelo, 6 cm.
- Distancia entre ruedas locas, 22.5 cm.
- Distancia entre ejes, 24.5 cm.

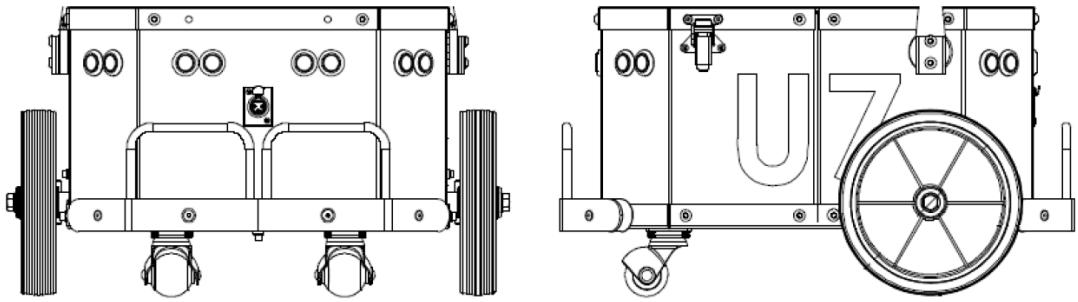


Figura 7. Configuración definitiva del robot Edubot.

Motores

A la hora de elegir el tipo de motor a utilizar en el robot Edubot se tuvieron en consideración diversos tipos, entre los que se encuentran los motores de corriente continua, los motores de corriente alterna, los motores paso a paso y los servomotores. A continuación se van a explicar brevemente las principales características de cada uno de ellos para así poder comprender el motivo de la elección del motor que se deberá controlar.

Motor de corriente continua

Son motores de gran utilidad cuando necesitamos controlar un amplio rango de velocidades y necesitamos entregar un par nominal relativamente grande a velocidades reducidas. Este tipo de motores se alimenta a través de dos bornes, y se puede cambiar la dirección de giro invirtiendo la polaridad de su alimentación.

Dentro de los motores de corriente continua es posible elegir entre los motores de imán permanente y los motores de campo devanado, siendo preferible el uso de los primeros debido a que los segundos tienen un peso mayor y requieren de circuitería adicional. Además, los motores de corriente continua de imán permanente proporcionan un amplio rango de potencias.

Motores de corriente alterna

Se trata de motores mucho más sencillos de manejar que unos motores de corriente continua y los costes de mantenimiento son muy reducidos. Sin embargo, aunque

puedan proporcionar unas potencias bastante elevadas, necesitan una serie de dispositivos adicionales que aumentan su consumo y disminuyen su eficiencia.

Existen dos tipos, los motores síncronos y los asíncronos. Los primeros, tienen un rotor que se mueve a la misma velocidad que gira el campo eléctrico producido por el estator, al contrario que los segundos. Los motores asíncronos, suelen preferirse a los síncronos debido a que son mucho más simples y tienen menos piezas sometidas a desgaste.

Motores pasó a paso

La gran diferencia de este tipo de motores respecto a los de corriente continua se encuentra en que pueden posicionar sus ejes en unos pasos fijos, por lo que pueden mantener la posición. Esto hace que los motores paso a paso sean ideales para robots que requieran movimientos muy precisos. Además, están dotados de un torque de detención, eliminando la necesidad de un mecanismo de frenado.

La velocidad de los motores paso a paso se controla variando el tiempo que nos encontramos en cada uno de los pasos, siendo más precisos cuanto mayor número de pulsos tengan por vuelta.

Al igual que en los motores de corriente continua, existen dos tipos, los motores de imán permanente y los motores de campo devanado.

Servomotores

Se tratan de unos dispositivos muy similares a los motores de corriente continua que pueden moverse hasta una determinada posición dentro de su rango y mantenerse en la misma de manera estable. A diferencia de los motores paso a paso, los servomotores pueden ser indicados a que posición deben moverse de manera directa, sin necesidad de ir alimentado a las bobinas del motor en una determinada secuencia hasta que el eje se posicione en el lugar deseado.

Entre todos los motores explicados, el elegido fue el motor de corriente continua debido a que cumplía numerosas cualidades, las cuales se enumeran a continuación.

- Bajo coste.
- Fáciles de manejar en todos los rangos de velocidades.
- Pequeñas dimensiones.
- Entrega elevada potencia.
- Alta eficiencia.

Se ha incluido un cuadro resumen, véase Tabla 2, para facilitar la visualización de la elección del motor.

		Características		
		Potencia	Coste	Control
Tipos de motores	Corriente continua	Elevada	Bajo	Media
	Corriente alterna	Elevada	Elevado	Sencillo
	Paso a paso	Baja	Bajo	Sencillo
	Servomotores	Elevada	Elevado	Sencillo

Tabla 2. Tipos de motores.

Sin embargo, existen dos desventajas las cuales se deberán subsanar. En primer lugar, debido a que los motores de corriente continua proporcionan elevadas velocidades será necesaria la colocación de reductores a su salida, que además de reducir la velocidad máxima de salida proporcionará un par más elevado. La segunda desventaja se encuentra en el control, ya que no hay forma de saber la velocidad y la posición del eje del motor de corriente continua. Para poder tener el control del motor, se colocarán unos codificadores incrementales en los motores.

2.3. Diseño electrónico

El robot está formado por un conjunto de sensores y actuadores que funcionan de manera independiente, es decir, no tienen ninguna capacidad de decisión. Esto implica la necesidad de un elemento que controle todos estos dispositivos para poder, posteriormente, mover el robot de manera correcta.

Antes de comenzar el desarrollo del diseño electrónico, se deben diferenciar los diferentes niveles de los cuales está dotado el robot Edubot. Existe un control de alto nivel, un control de bajo nivel y el hardware del robot. En la Figura 8 se puede observar una representación de esta estructuración de niveles.

El hardware del robot es el nivel más bajo que existe. En él se encuentran los sensores y los actuadores, los cuales proporcionan datos sobre el entorno en el cual se desplaza el robot o bien ejecutan una determinada orden.

El control de bajo nivel es el nivel intermedio. En él se procesan todos los datos obtenidos del hardware del robot o bien los datos recibidos del control de alto nivel. Todo este control será implementado a través de placas Arduino, tres placas controladoras en el caso del proyecto anterior.

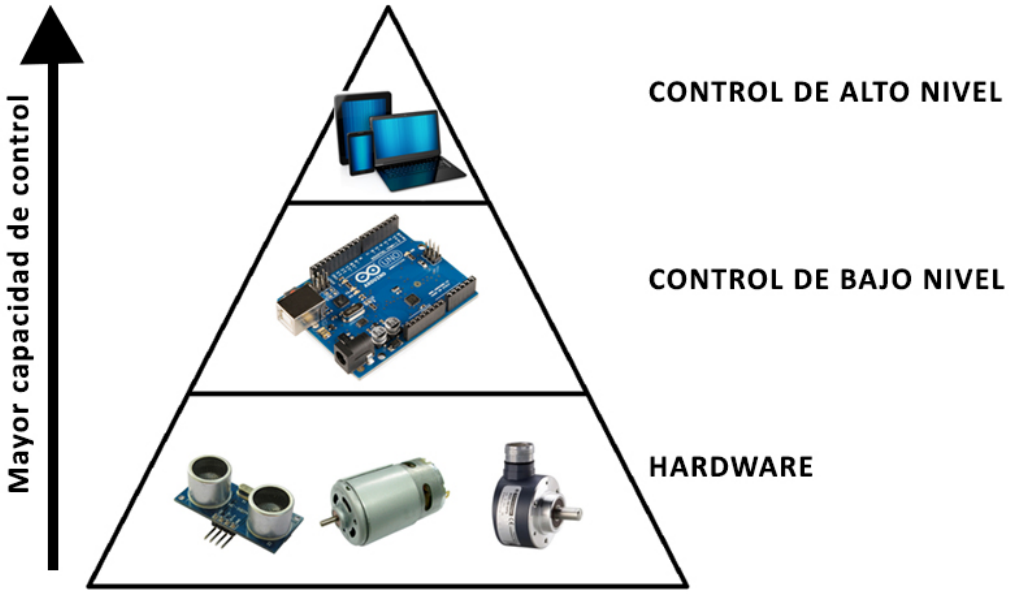


Figura 8. Niveles de diseño.

Por último, el control de alto nivel es el nivel más alto. En él se procesan los datos recibidos del control de bajo nivel para que en función de los resultados, se realizan diferentes funciones. Además, también tiene la capacidad de realizar una serie de órdenes que debe ejecutar el hardware.

Sensores

Ultrasonidos

El robot incluye ocho ultrasonidos *"Paralax PING"*, véase Figura 9, cuatro colocados en la parte delantera y otros cuatro en la parte trasera, estando los sonars de los extremos situados en un ángulo de 45º, pudiendo de esta manera, detectar obstáculos en un rango angular mayor.

Estos dispositivos sirven para detectar objetos que se encuentren alrededor del robot, sabiendo a que distancia están y así poder esquivarlos sin dificultad.

Los ultrasonidos utilizados son capaces de detectar objetos que se encuentran a una distancia de entre 2 cm a 3 m mediante señales de alta frecuencia, 40Hz. Está dota de tres pines, uno de tierra (GND), otro de alimentación (5V) y un último pin para transmitir la información (SIG).

Figura 9. Ultrasonido Parallax PING.

Su funcionamiento es muy sencillo. El sensor emite unas señales de alta frecuencia por un lado, cuando el control de alto o bajo nivel se lo indican por el pin SIG, es decir, se manda un pulso de inicio al ultrasonido. Estas señales chocan contra el objeto que tenga a su alcance, produciéndose una señal de eco, la cual es recibida en el otro lado del sensor. El sensor emite un pulso de salida al control de bajo nivel, que comienza cuando emite la señal de alta frecuencia y termina cuando recibe el eco, produciéndose un pulso con un ancho equivalente a la distancia del obstáculo. Su funcionamiento puede verse de una manera más visual en la Figura 10.

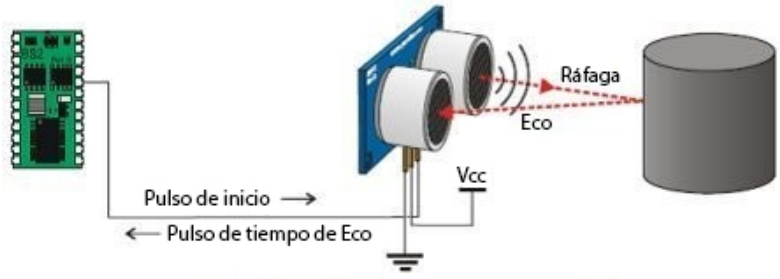


Figura 10. Funcionamiento de un ultrasonido.

Bumpers

El robot tiene dos bumpers, uno en frontal y otro trasero, ambos colocados en la zona inferior. Cada bumper está formado por dos muelles que al ser comprimidos activan un interruptor. Cuando los dos interruptores son presionados, es decir, cuando hay un

choque frontal, se emite un pulso al control de bajo nivel, Arduino, pudiendo saber de esta manera que el robot ha chocado contra un obstáculo.

Además de detectar un choque, también sirven como amortiguadores, ya que los mismos sobresalen de la parte delantera y trasera del robot, evitando de esta forma un golpe directo sobre el mismo. En la figura 11 se puede observar uno de los bumpers colocados en el robot.

Figura 11. Bumper colocado en el robot Edubot.

Sistema de alimentación

A la hora de elegir el dispositivo de alimentación se tuvieron en cuenta dos características:

- El dispositivo debe ser independiente, es decir, no debe estar enchufado a una toma de corriente. De esta forma, el robot podrá moverse libremente,
- El dispositivo debe tener autonomía suficiente como para que el robot pueda estar en movimiento durante un tiempo relativamente largo.

Dispositivos como placas solares, generadores o baterías cumplen con los requisitos citados en el párrafo anterior, por lo que para poder elegir entre uno de ellos se tuvieron más características en cuenta, aunque de menor importancia, entre las cuales se encuentran:

- Sencillo de manejar.
- Posibilidad de recargar.
- Bajo coste.
- Fácil de integrar dentro del robot.

Teniendo en cuenta todas las características necesarias que debe cumplir el sistema de alimentación de robot Edubot, se optó por una alimentación mediante baterías eléctricas.

Una vez elegido el sistema de alimentación hay que elegir la batería idónea para el robot. Existen múltiples tipos de baterías eléctricas como pueden ser las de Plomo-ácido (Pb), Níquel-Cadmio (Ni-Cd), Níquel e hidruro metálico (Ni-MH), iones de litio (Li-Ion).

Hay que evitar las baterías con efecto memoria ya que se reduce progresivamente la capacidad de las mismas debido a sobrecargas repetidas y descargas parciales. Por este motivo, de las baterías anteriormente citadas descartamos las de Ni-Cd y las de Ni-MH.

Entre las baterías de Níquel y de Plomo, decir que las más ligeras son las de Níquel, sin embargo tienen un coste muy elevado que hace que se deba descartar. Por tanto, la batería escogida fue la de Plomo-ácido. Además, este tipo de batería puede ser recargada con cualquier fuente de tensión de corriente continua que proporcione una tensión mayor que la de la batería para forzar la corriente de carga.

La batería del robot Edubot es una *"Yuasa NP17-12I FR"*, Figura 12, proporciona 12 V y tiene una capacidad de 17Amperios-hora.

Figura 12. Batería robot Edubot.

La parte principal del sistema de alimentación del robot es la batería, sin embargo, hay otros elementos muy importantes necesarios en el robot y que forman parte de su sistema de alimentación. Dichos elementos son el sistema de carga y los elementos de protección.

Sistema de carga

Para mayor comodidad, se instaló en el robot un sistema de carga para poder cargar la batería sin tener que sacar la misma del robot. El conector del sistema de carga no va directamente conectado a la batería, entre medias hay un fusible encargado de proteger la batería y el resto de componentes electrónicos. En la Figura 13 se puede observar el conector de carga del robot.

Figura 13. Sistema de carga.

Elementos de protección

Estos elementos de protección son los fusibles. Hay cuatro entre la batería y resto de componentes electrónicos para protegerlos de posibles cortocircuitos. De estos cuatro, en este proyecto sólo se utiliza uno, el del conector de carga.

La caja de fusibles se encuentra situada en la mitad del chasis debido a que es el mejor lugar para intercalar los fusibles entre la alimentación que se encuentra en la parte posterior y el resto de circuitos electrónicos que se encuentran en la parte delantera.

En la figura 14 se puede observar el esquema eléctrico de los fusibles del robot Edubot actual.

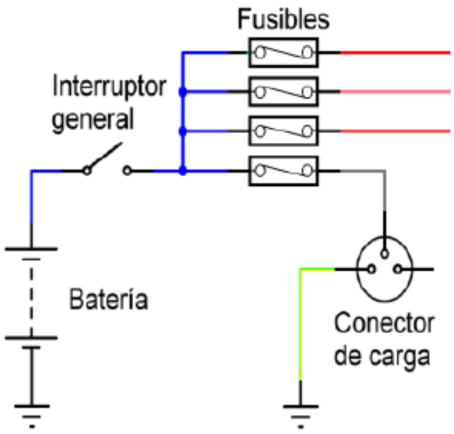


Figura 14. Esquema eléctrico de los fusibles.

Controlador GPMRC

En el proyecto anterior a este se utilizó un controlador GPMRC como controlador de bajo nivel el cual será sustituido en el presente proyecto por dos placas Arduino que posteriormente se explicarán, con el fin de realizar el control del robot con un menor número de placas, reduciendo de esta manera peso, espacio y cableado dentro del robot.

Comentar brevemente que dicho controlador GPMRC se encargaba de:

- La comunicación con el dispositivo de control de alto nivel.
- Conocer el estado del hardware del robot, centralizar toda la información e informar de la misma.
- Transformar las órdenes de la controladora de alto nivel para poder ejecutarlas en el hardware del robot.
- Medir la tensión de alimentación e informar de su estado.
- Detectar los errores y funcionamientos extraños y advertir de ellos.
- Captar y contabilizar los pulsos generados por los codificadores incrementales.

Puentes H

En los motores eléctricos de corriente continua no se puede controlar el sentido de giro del mismo de manera independiente. Es decir, es necesaria la utilización de un dispositivo adjunto entre el motor y el controlador de bajo nivel que controle el sentido de giro.

Dicho dispositivo es un puente H. Se trata de un circuito electrónico que está compuesto por cuatro interruptores cableados de manera estratégica para que, dependiendo de los interruptores que estén abiertos o cerrados, se permita el paso de tensión positiva o negativa. Para facilitar la compresión de su funcionamiento, se adjunta una imagen, véase Figura 15, donde se representa la estructura simplificada de un puente H junto a una tabla, Tabla 3, donde se pueden ver todos los posibles funcionamientos del motor en función del estado de cada interruptor. Si el interruptor está abierto, su estado se representa mediante un 0, por el contrario, el estado del interruptor cerrado se representa mediante un 1.

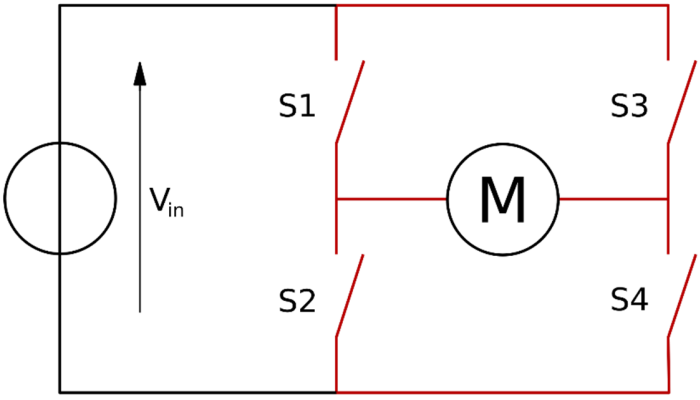


Figura 15. Esquema de un puente H.

Interruptor				Funcionamiento
S1	S2	S3	S4	
1	0	0	1	Avance
0	1	1	0	Retroceso
0	0	0	0	Frenado bajo la propia inercia del motor
1	0	1	0	Frenado brusco

Tabla 3. Funcionamiento Puente H.

Comentar, que los interruptores S1 y S2 no deberán estar cerrados nunca al mismo tiempo, al igual que los interruptores S3 y S4, de ser así, la fuente de tensión se vería cortocircuitada.

Codificadores incrementales

Como ya se dijo anteriormente, el robot está dotado de unos motores de corriente continua con los cuales no podemos medir la posición de las ruedas. Para poder hacerlo, los motores llevan acoplados a sus ejes unos codificadores incrementales.

Los codificadores colocados en el robot Edubot son incrementales y de dos canales de efecto Hall, los cuales tienen una resolución de 20 pulsos por vuelta. Sin embargo, la resolución total del conjunto motor-engranaje reductor es de 980 pulsos por vuelta, ya que los engranajes tienen una relación de 1:49.

Este tipo de codificador está formado por un disco con ranuras radiales que giran frente a un fotosensor, de forma que se van generando pulsos cuando la luz atraviesa una ranura. De esta forma podemos conocer la dirección y la velocidad del eje y su posición desde una posición de referencia dada.

En la Figura 16, se puede observar el codificador incremental acoplado al motor de corriente continua colocado en el robot.

Figura 16. Codificador incremental acoplado al motor.

3. DISEÑO ACTUAL

3.1. Introducción

En el capítulo anterior se explicó desde que base se partía en el presente proyecto. Por lo que una vez explicada tanto la estructura mecánica como los sensores y actuadores que hay que controlar, se va a proceder con el desarrollo de este proyecto.

A lo largo de este capítulo se va a explicar la adaptación que ha sido necesaria hacer en el diseño electrónico para poder sustituir la controladora GPMRC por tres microcontroladores Arduino.

También se explicará la necesidad de utilizar tres microcontroladores en vez de uno y que funciones realiza cada uno.

Para finalizar, se representará de manera esquemática la arquitectura hardware del robot.

3.2. Diseño electrónico

Como ya se ha dicho, es necesario un control de bajo nivel que sirva de intermediario entre el controlador de alto nivel, en este caso un ordenador, y el hardware del robot Edubot. Este control de bajo nivel ha sido implementado por tres microcontroladores Arduino, por todos los motivos citados en el *Capítulo 1*.

Actualmente, en el mercado existe una gran variedad de microcontroladores Arduino, desde el famoso Arduino Uno hasta el Arduino Leonardo, pasando por el Arduino Nano, entre otros muchos. La principal diferencia entre todos ellos es el número de pines analógicos y digitales que tienen, por lo que la elección del mismo se realizará en función del número de pines que necesitemos.

Antes de comenzar con la elección del tipo de microcontrolador Arduino que se va a necesitar, es de gran importancia recalcar el hecho de que se necesitarán tres microcontroladores Arduino.

A modo de breve introducción, ya se explicará el control con mayor detalle en el siguiente capítulo, el control de bajo nivel debe analizar de manera constante y en periodos de tiempo muy reducido las señales recibidas de los ultrasonidos y de los bumpers. Lo mismo ocurre con el control del estado de los motores.

Si todos los controles se realizaran en un mismo microcontrolador, y por tanto en bucles en serie, los periodos de control se alargarían, corriendo el riesgo de no

detectar un obstáculo a tiempo y/o de tener un control sobre los motores deficiente. Para evitar esto, la mejor solución es que estos bucles no se hagan en serie, sino en paralelo, de esta forma, el control de los ultrasonidos y de los bumpers se haría a la vez que el control del motor derecho y todo esto a su vez, al mismo tiempo que el control del motor izquierdo.

Para poder implementar estos tres controles en paralelos será necesario que cada control se realice en un microcontrolador diferente, de aquí, la necesidad de tener tres microcontroladores.

Un microcontrolador estará destinado al control del motor izquierdo, otro para el análisis de las señales recibidas de los ultrasonidos y los bumpers, y un tercero para el resto de controles y lecturas, es decir, para el control del motor derecho, para la lectura y análisis del estado de la batería y para la comunicación con el control de alto nivel.

Elección de los microcontroladores

Una vez conocido el uso de cada microcontrolador, se deberán estudiar cuantos pines y cuanta potencia se necesitarán para un control óptimo del robot.

Se va a empezar por el microcontrolador encargado del control de los ultrasonidos y los bumpers.

Como ya se detalló en el capítulo anterior, los ultrasonidos utilizados en este robot están dotados de tres pines (5V, SIG, GND). El pines de señal, SIG, es un pin digital, por lo que al haber ocho ultrasonidos, necesitaremos ochos pines digitales. El resto de pines, los de 5V y GND, pueden agruparse, es decir, los pines de 5V de todos los ultrasonidos se cablearán de forma que todos estén conectados, saliendo un único cable que irá al pin de 5V del microcontrolador Arduino. Lo mismo se hará con los pines GND.

Por otro lado están los bumpers, los cuales tienen dos pines cada uno, uno de GND y otro de señal. El pin de señal es un pin digital, por lo que al haber dos bumpers, serán necesarios otros dos pines digitales más. De nuevo, los pines GND se podrán unificar, ocupando un único pin GND en el microcontrolador Arduino.

Por último, será necesario comunicar este microcontrolador con el microcontrolador maestro, por lo que para dicha comunicación será necesaria la utilización de un pin SDA, un pin SCL y otros dos pines de 5V y GND independientes no utilizados anteriormente.

Si se hace un recuento de todos los pines necesarios en este microcontroladores:

- 10 pines digitales
- 2 pines GND
- 2 pines 5V
- 1 pin SDA
- 1 pin SCL

Una vez realizado el estudio de pines de una de las placas microcontroladoras, se va a proceder al estudio de los pines necesarios de las otras dos placas.

En ambas se debe realizar el control de los motores, y además, en una de ellas se deberá de tener en cuenta el análisis de la lectura de la batería.

Para este primer control, hay que tener presente que el mismo está dividido en dos *subcontroles*, uno referente a la velocidad del motor y otros a los datos recibidos del codificador incremental.

Para el control de velocidad de cada motor se necesitan tres pines, uno de habilitación, DIS, otro de dirección, DIR, y uno de modulación de ancho de pulso, PWM. Estos tres pines son digitales, aunque el último, debe ser específicamente un pin digital PWM. Para el control de los codificadores se necesitan cuatro pines por encoder, uno GND, otro 5V, y los otros dos pines restantes son los referentes a los canales A y B de cada encoder. Estos dos últimos pines son pines digitales con interrupción. Por último, será necesario el uso de dos pines de comunicación puerto serie, RX y TX, con su correspondiente pin GND común al Arduino maestro.

Si se hace un recuento de los pines necesarios para los dos microcontroladores:

- 5 pines digitales, de los cuales 1 PWM y 2 con interrupción.
- 2 pines GND
- 1 pin 5V
- 1 pin RX
- 1 pin TX

Además, uno de ellos debe tener pines suficientes para la lectura de baterías y su correspondiente respuesta (encendido de leds). Este análisis se va a realizar a través de un pin analógico, el de la batería, tres pines digitales, los cuales corresponden a los tres leds (rojo, amarillo y verde) que nos indican visualmente en el chasis del robot el estado de la batería y un pin GND.

Si se hace un recuento de los pines adicionales que necesita el Arduino maestro:

- 3 pines digitales
- 1 pin analógico
- 1 pin GND

Por último, es necesario recalcar la necesidad de una capacidad de procesamiento relativamente alta para conseguir un control de los motores óptimo.

Una vez realizado este estudio, se puede proceder a la elección de los microcontroladores Arduino más adecuados para este proyecto.

Se han tenido en cuenta, a la hora de realizar esta elección los Arduinos Uno, Due, Leonardo, Meda 2560, y Nano.

Se ha realizado una tabla comparativa, véase Tabla 4, donde se muestran las características de estos microcontroladores que interesan en este proyecto, para facilitar el proceso de exclusión de microcontroladores y poder ver de manera más visual el microcontrolador adecuado.

		Características					
		Pines Digitales			Pines Anal.	Puertos serie	Potencia
		Normal	Con interrup.	PWM			
Tipo de Arduino	Uno	14	2	6	6	1	Baja
	Leonardo	20	5	7	12	1	Media
	Mega 2560	54	6	15	16	4	Alta
	Nano	14	2	6	8	1	Baja

Tabla 4. Tipos de Arduino.

Para el primer microcontrolador, el que se encarga del control de los ultrasonidos y los bumpers, basta con 10 pines digitales, por lo que cualquiera de ellos nos valdría. El microcontrolador elegido es el Arduino Uno, ya que no son necesarios un número excesivo de pines digitales y analógicos, ni una potencia excesiva.

Para los microcontroladores encargados del control de los motores se ha optado por el Arduino Mega, por su elevada capacidad de procesamiento y por el número de puertos serie que posee, pudiendo utilizar uno de ellos como maestro. Este último, al tener hasta 4 puertos serie, podrá comunicarse con numerosos dispositivos ya sea por comunicación I2C o por comunicación Puerto Serie.

Arduino Uno

El Arduino Uno está destinado al control de los ultrasonidos y de los bumpers, pasando toda esta información a través de una comunicación I2C mediante los pines SCL y SDA al Arduino Mega (toda esta comunicación se explicará más adelante).

En la Figura 17, se puede observar un microcontrolador como el usado en el proyecto, cuyo esquema se encuentra adjunto al finalizar la memoria en la sección Anexos, *Anexo 1. Esquema Arduino Uno,* donde se encuentran detallados todos los pines que han sido utilizados para el control de los sensores, adjuntado a la imagen una tabla, véase Tabla 5, con la misión que tiene cada conexión.

Todos los sensores tienen una señal SIG que va conectada a un pin digital del Arduino Uno. En la tabla adjunta se pueden observar que ultrasonidos están conectados con los pines del microcontrolador, y para mayor aclaración, se ha adjuntado un esquema de la situación de cada ultrasonido y la numeración que se le ha impuesto a lo largo del desarrollo del proyecto, véase Figura 18.

Todos los ultrasonidos tienen una señal GND, que por motivos de orden en el interior del robot, se han unificado todos los cables de esta señal en uno sólo, junto con los cables que transmiten esta señal en los bumpers. Lo mismo se ha hecho con los cables que transmiten la señal 5V de los ultrasonidos.

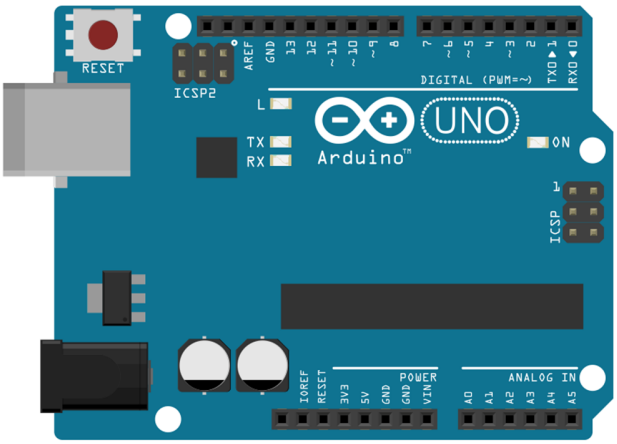


Figura 17. Arduino Uno.

Arduino Uno	
Pin	**Descripción**
2	Pin digital. Conexión con SIG sonar 1
3	Pin digital. Conexión con SIG sonar 2
4	Pin digital. Conexión con SIG sonar 3
5	Pin digital. Conexión con SIG sonar 4
6	Pin digital. Conexión con SIG bumper delantero
7	Pin digital. Conexión con SIG sonar 5
8	Pin digital. Conexión con SIG sonar 6
9	Pin digital. Conexión con SIG sonar 7
10	Pin digital. Conexión con SIG sonar 8
11	Pin digital. Conexión con SIG bumper trasero
GND (1)	Conexión GND para comunicación I2C
GND (2)	Conexión GND para sensores
IOREF	Conexión 5V para comunicación I2C
5V	Conexión 5V para sensores
A4	Pin analógico. Conexión SDA para comunicación I2C
A5	Pin analógico. Conexión SCL para comunicación I2C

Tabla 5. Arduino Uno.

Figura 18. Mapa de ultrasonidos del robot.

Para realizar la comunicación entre este Arduino y el Arduino Mega, ha sido necesario un cableado extra, comunicando el pin SCL del Arduino Mega con el pin SCL del

Arduino Uno, e igualmente con el pin SDA. Para realizar esta comunicación, también es necesario que ambos Arduinos tengan un GND y un 5V común, por lo que a través de otro pin GND no usado, el Arduino Uno dispone de hasta 3 pines GND, se ha extendido cableado hacia otro pin libre GND del Arduino Mega con el fin de unificarlos. Lo mismo hay que hacer con los 5V. Visiblemente, el Arduino Uno parece tener un solo pin de 5V, sin embargo, posee el pin IOREF que tiene la misión de suministrar 5V.

Arduino Mega

En el control de bajo nivel hay dos Arduinos Mega, uno de ellos destinado exclusivamente al control del motor izquierdo, pasando parte de la información procesada al otro Arduino Mega, el maestro, a través de comunicación Puerto Serie, RX/TX (toda esta comunicación se explicará más adelante) y otro dónde, aparte de controlar los motores, deberá gestionar toda la comunicación con el control de alto nivel y analizar el estado de la batería.

En la Figura 19, se puede observar un microcontrolador como los usados en el proyecto, cuyo esquema se encuentra adjunto al finalizar la memoria en la sección Anexos, *Anexo 2. Esquema Arduino Mega,* donde se encuentran detallados todos los pines que han sido utilizados para el control de cada motor, adjuntado a la imagen dos tablas, véase Tabla 6 y Tabla 7, con la misión que tiene cada conexión en el Arduino Mega esclavo y en el Arduino Mega maestro, respectivamente.

Se debe prestar especial atención a la Tabla 6, dónde se puede observar cómo ha sido necesario conectar los canales A y B de los codificadores incrementales a pines digitales con interrupción.

Comentar, que el Arduino Mega no va directamente conectado al motor, entre medias hay conectado un puente H, uno por cada motor, que hace posible el control del sentido del motor.

Además, para realizar la comunicación entre el Arduino Mega esclavo y el Arduino Mega maestro ha sido necesario un cableado extra, conectando los pines RX y TX del Arduino Mega esclavo con los pin TX y RX del Arduino Mega maestro, respectivamente. Por último, comentar la necesidad de que ambos Arduinos tengan un GND común, por lo que a través de otro pin GND no usado, se ha extendido cableado hacia otro pin libre GND del Arduino Mega con el fin de unificarlos.

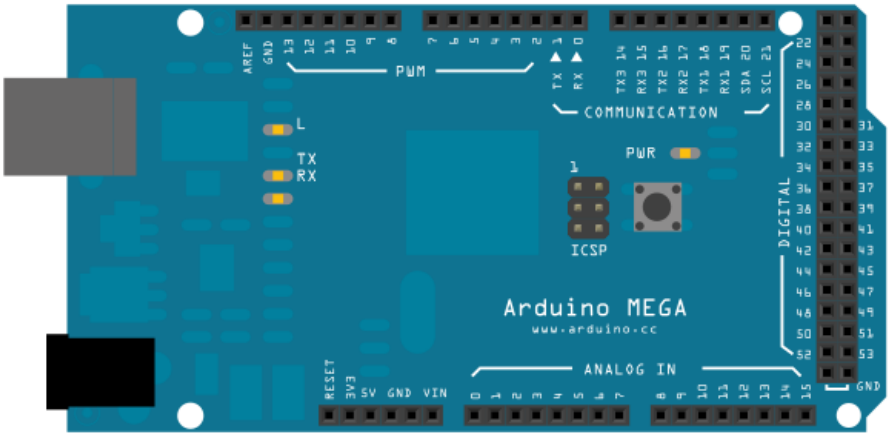


Figura 19. Arduino Mega.

Arduino Mega esclavo	
Pin	**Descripción**
0	Pin RX para la comunicación Puerto Serie
1	Pin TX para la comunicación Puerto Serie
2	Pin digital con interrupción. Conexión con canal A del encoder
3	Pin digital con interrupción. Conexión con canal B del encoder
9	Pin digital. Conexión con pin DIR del puente H del motor
10	Pin digital. Conexión con pin PWM del puente H del motor
11	Pin digital. Conexión de habilitación del motor derecho
GND	Conexión GND para actuadores
5V	Conexión 5V para actuadores
GND(2)	Conexión GND para la comunicación Puerto Serie

Tabla 6. Arduino Mega esclavo.

Arduino Mega maestro	
Pin	**Descripción**
0	Pin RX para la comunicación Puerto Serie
1	Pin TX para la comunicación Puerto Serie
2	Pin digital con interrupción. Conexión con canal A del encoder
3	Pin digital con interrupción. Conexión con canal B del encoder
5	Pin digital. Conexión con led verde
6	Pin digital. Conexión con led amarillo
7	Pin digital. Conexión con led rojo
9	Pin digital. Conexión con pin DIR del puente H del motor
10	Pin digital. Conexión con pin PWM del puente H del motor
11	Pin digital. Conexión de habilitación del motor derecho
A2	Pin analógico. Conexión a batería
20	Pin digital. Conexión SDA para comunicación I2C
21	Pin digital. Conexión SCL para comunicación I2C
GND	Conexión GND para actuadores
GND(2)	Conexión GND para la comunicación Puerto Serie
GND (3)	Conexión GND para comunicación I2C
5V	Conexión 5V para actuadores

Tabla 7. Arduino Mega maestro.

3.3. Arquitectura hardware del robot Edubot

Se ha encontrado útil la representación de todo el hardware del robot Edubot de manera esquemática, indicando con que pines cuenta cada dispositivo que forma parte del robot.

Por mayor claridad, en primer lugar se representará cada dispositivo por separado y posteriormente se representará el esquema de la arquitectura global del robot, para poder visualizar toda la estructura de un solo vistazo.

Ultrasonidos – Arduino Uno

Ya se ha explicado qué es un ultrasonido, su funcionamiento y su estructura. Se trata de un sensor muy sencillo de utilizar, y como se puede ver en la Figura 20, su conexionado es muy simple de implementar.

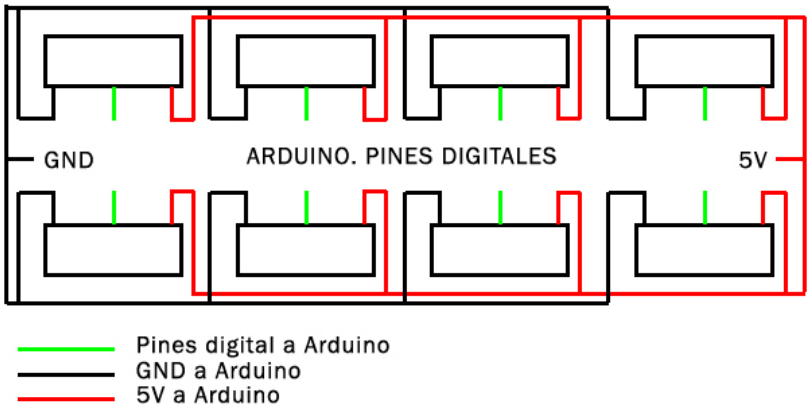


Figura 20. Esquema conexionado ultrasonidos.

Bumpers – Arduino Uno

Al igual que en los ultrasonidos, ya se explicó el funcionamiento y aplicación de estos sencillos dispositivos, que también son muy sencillos de implementar. En la Figura 21, se puede ver un esquema del cableado de los bumpers.

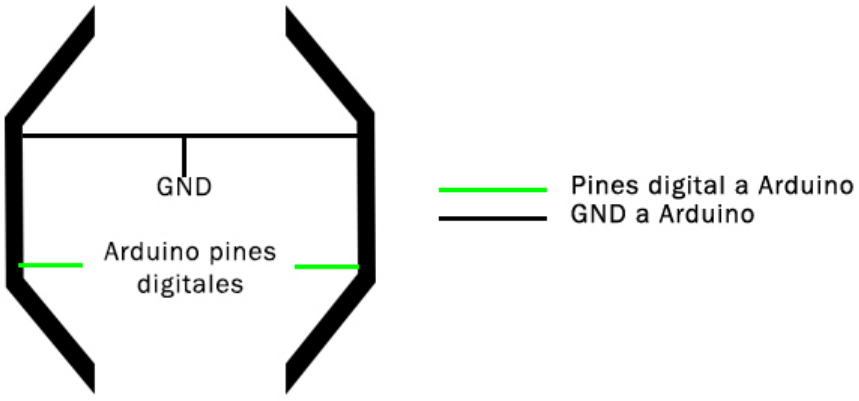


Figura 21. Esquema conexionado bumpers.

Batería – Arduino Mega

En el capítulo anterior se explicó el proceso de selección de la batería a utilizar pero no su conexionado con el control de bajo nivel.

En primer lugar, hay que prestar especial atención en que la batería elegida puede llegar a suministrar hasta 12 V y que el Arduino Mega puede recibir como máximo 5V. Por este motivo, para poder controlar el nivel de carga de la batería será necesario adaptar las tensiones para evitar quemar el microcontrolador.

Es necesario colocar un dispositivo entre el microcontrolador y los bornes de la batería que reduzca la tensión como mínimo a 5 V. Este dispositivo es un divisor de tensión.

Un divisor de tensión es una configuración de circuito eléctrico que reparte la tensión de una fuente entre una o más impedancias conectadas en serie. En la Figura 22 se muestra un esquema de un divisor resistivo igual al implantado en el robot Edubot.

La fórmula de este tipo de divisor de tensión se puede observar a continuación:

$$V_o = \frac{R_2}{R_1 + R_2} \cdot V_i \qquad \text{(Ec.1)}$$

En el caso de este robot, se conocen los siguientes datos:

- V_o = 5 V
- V_i = 12 V

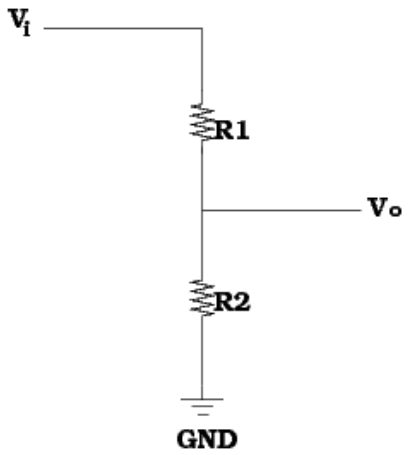


Figura 22. Divisor de tensión resistivo.

Conociendo los valores estándar de las resistencias que se comercializan, los valores de las resistencias elegidas fueron:

- R1 = 150 KΩ
- R2 = 100 KΩ

Esto implica que cuando la batería esté completamente cargada, al microcontrolador le llegarán 5V, cuando tenga media carga le llegarán 2.5V y cuando esté sin ninguna carga 0V.

Teniendo en cuenta todo lo dicho a lo largo de este apartado, el esquema final de la conexión de la batería se puede ver en la Figura 23. Se puede observar, que se ha tenido en cuenta, a parte del divisor de tensión, la caja de fusibles y los puentes H, ya que están comunicados también con la batería.

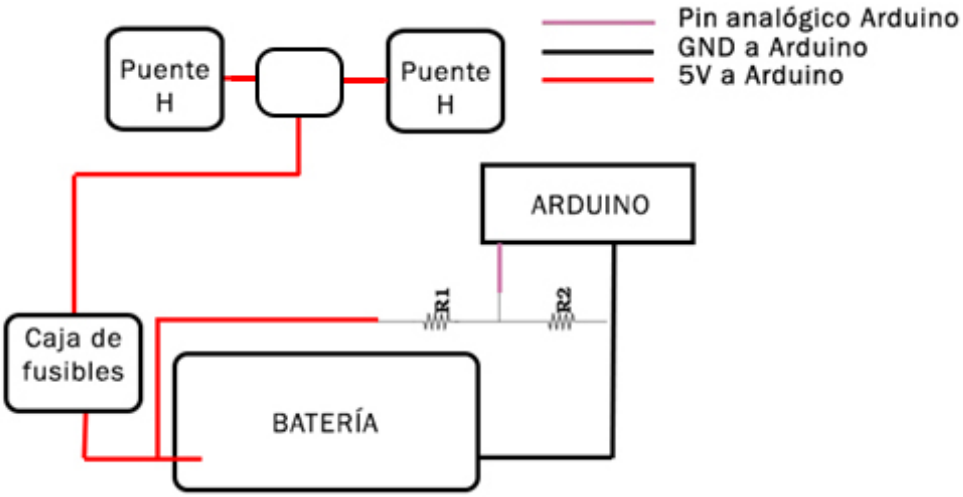


Figura 23. Esquema cableado batería.

Motor – Encoder – Puente H – Arduino Mega

El hardware referente a los motores es el que más pines y por tanto, cableado tiene. Por lo que es de vital importancia explicar claramente qué pines salen de qué dispositivos y que función tienen.

La alimentación que sale de la caja de fusibles no alimenta directamente a los dos puentes H. Los dos cables de alimentación que salen de la caja de fusibles, GND y 12 V, van a un dispositivo intermedio el cual alimenta a los dos puentes H, cediendo a cada puente H la misma tensión que le llega. Es decir, si al dispositivo intermedio le

llegan 12V, este mandará 12V a cada puente H. Por tanto, este circuito tiene dos entradas, GND y 12V, y cuatro salidas, GND y 12V para cada puente H.

Por otro lado se tienen dos puentes H que tienen cuatro salidas cada uno, 12V, GND, y las conexiones a los dos bornes del motor. Además, tienen otras cuatro entradas, una de habilitación de movimiento, DIS, otra de tierra, GND, y otras dos para los canales A y B. El canal A indica el sentido de giro y el canal B el PWM, y ambos van conectados a pines digitales del Arduino Uno, siendo necesario que los canales B de cada puente H vayan conectados a pines digitales dotados de PWM.

Por último está el encoder, el cual va fijado al motor y tiene cuatro pines. Dos de ellos están destinados a la alimentación, 5V y GND, y los otros dos son las salidas del dispositivo, los canales A y B.

Para mayor claridad, se ha representado todo lo explicado en un esquema gráfico, el cual puede observarse en la Figura 24.

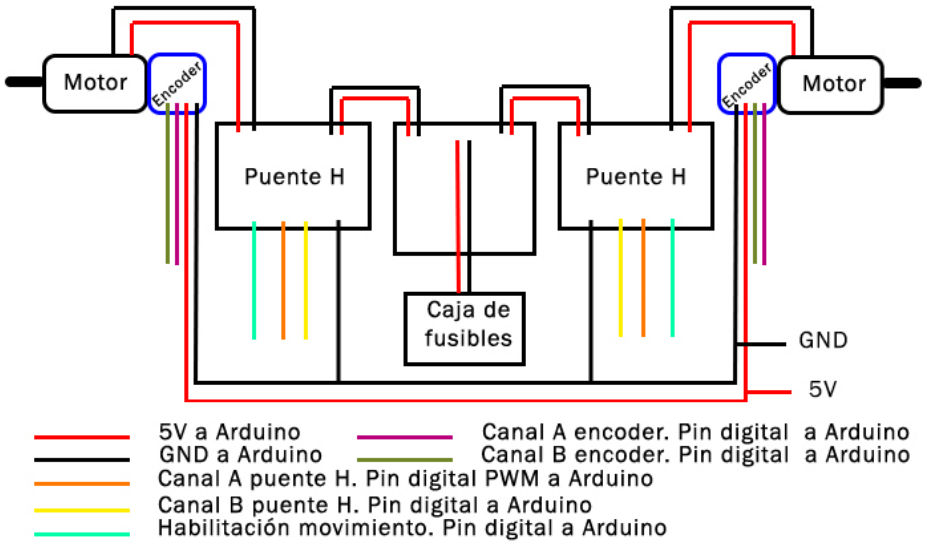


Figura 24. Esquema cableado motores.

Montaje total

Una vez desarrollado todo el esquema eléctrico y de conexionado de todos los dispositivos por separado, se va a representar el esquema global del robot Edubot, para poder ver de manera clara y rápida todas las conexiones realizadas.

Dicho esquema se puede ver representado en la Figura 25. Como se puede observar, en todos los esquemas se ha mantenido la misma *nomenclatura* en cuanto a los

colores utilizados para realizar el cableado y se ha adjuntado una pequeña leyenda para poder comprender con mayor claridad lo que representa cada cable.

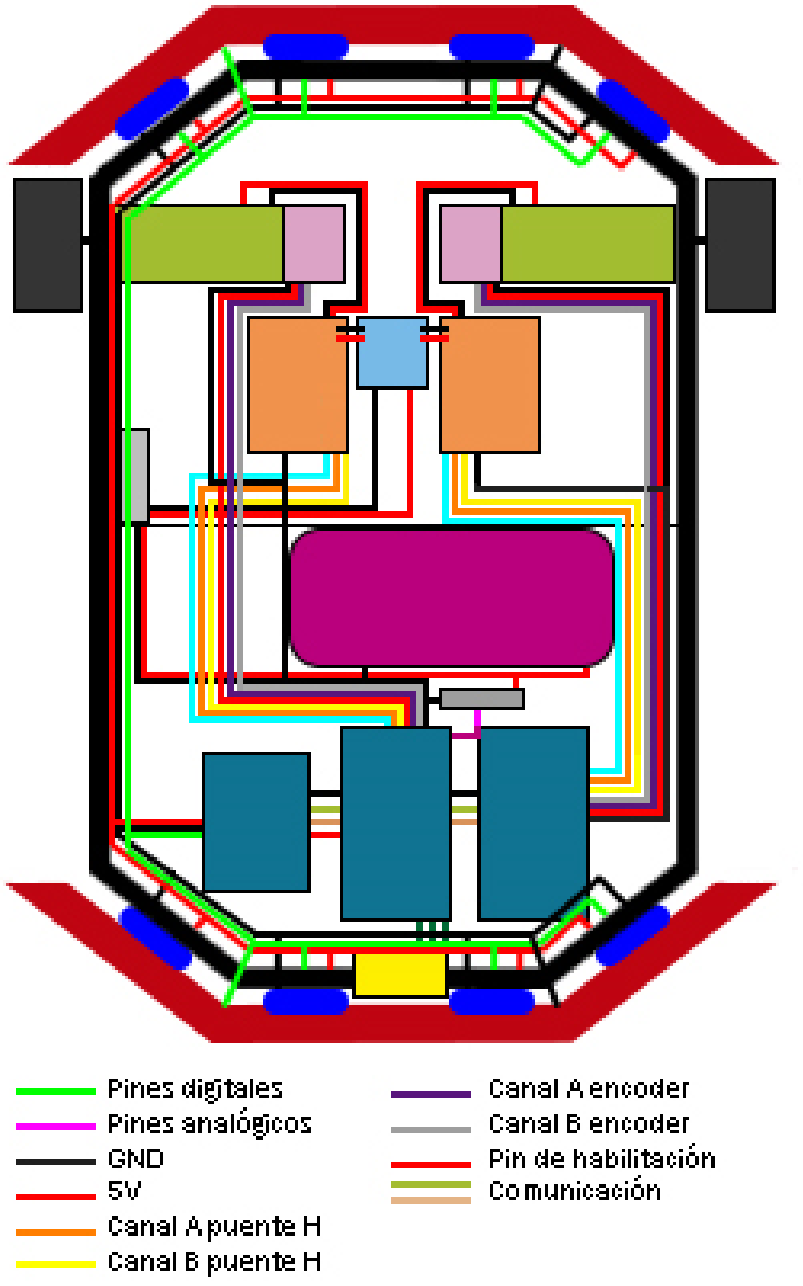


Figura 25. Esquema eléctrico global.

4. CONTROL DEL ROBOT EDUBOT

4.1. Introducción

En los dos capítulos anteriores se habló el diseño actual del robot Edubot, explicando con que dispositivos se realizan los diferentes controles del mismo. A lo largo del presente capítulo, se desarrollará el control del robot centrándose en mayor profundidad en el software realizado, representando los bucles programados para mayor claridad del lector.

En primer lugar, se procederá a la explicación del control de los bumpers y ultrasonidos, para posteriormente centrarnos en el control del estado de la batería y la gestión de las alarmas.

Después se procederá al desarrollo del control de la velocidad. Por motivos de orden, complejidad y claridad, se ha decido explicar todo este control de forma progresiva, según se ha ido avanzado en la realidad con dicho control.

Por último, aunque finalmente no haya sido desarrollada en el control de bajo nivel, se explicará la localización odométrica del robot.

Los códigos desarrollados, aunque si se encuentran explicados, no se han adjuntado para evitar un exceso de carga a lo largo del capítulo. Por este motivo, se ha decido adjuntarlos al final del proyecto en la sección de Anexos. Todos ellos se encuentran perfectamente comentados y utilizan variables nombradas de la manera más autodescriptiva posible para facilitar la comprensión del código.

4.2. Recepción de ultrasonidos y bumpers

El análisis del estado de los ultrasonidos y bumpers se realiza en el microcontrolador Arduino Uno.

Para su control, se lee de manera continuada las señales recibidas por parte de los sensores. Por motivos de claridad, en el bucle principal se llama a dos funciones a parte, *"Lectura_sonar()"* y *"Lectura_bumper()"*, las cuales analizan estas señales recibidas.

En la función *"Lectura_sonar()"* se realiza un bucle para tratar la señal de cada ultrasonido de manera consecutiva. Para ello, en primer lugar se manda al ultrasonido un pulso por el pin SIG para indicarle que emita una ráfaga de ondas ultrasónicas. Después, el sensor mandará al microcontrolador un pulso que finalizará cuando reciba

el *eco* de la señal mandada. Hay que tener en cuenta que el ultrasonido devuelve al microcontrolador el tiempo que ha tardado en recibir la señal de eco, por lo que será necesaria una conversión de este tiempo en centímetros. Dicha conversión la realiza la función *"microsecondsToCentimeters()"*.

Por otro lado, la función *"Lectura_bumper()"*realiza un bucle en el que lee el estado de la señal digital referente a cada bumper. En el caso de que la señal digital se encuentre en alto, significará que el mismo no ha sido pulsado, sin embargo, si la señal digital se encuentra en bajo, significará que el bumper ha sido pulsado y por tanto que ha habido un choque.

Se ha adjuntado un esquema, véase Diagrama de flujo 1, del control de los bumpers y los ultrasonidos, que facilitará su compresión. Recordar, que aunque en este capítulo no se adjuntan los códigos, el lector puede consultarlos al final de la memoria, en la sección de Anexos, *Anexo 3. Recepción de ultrasonidos y bumpers.*

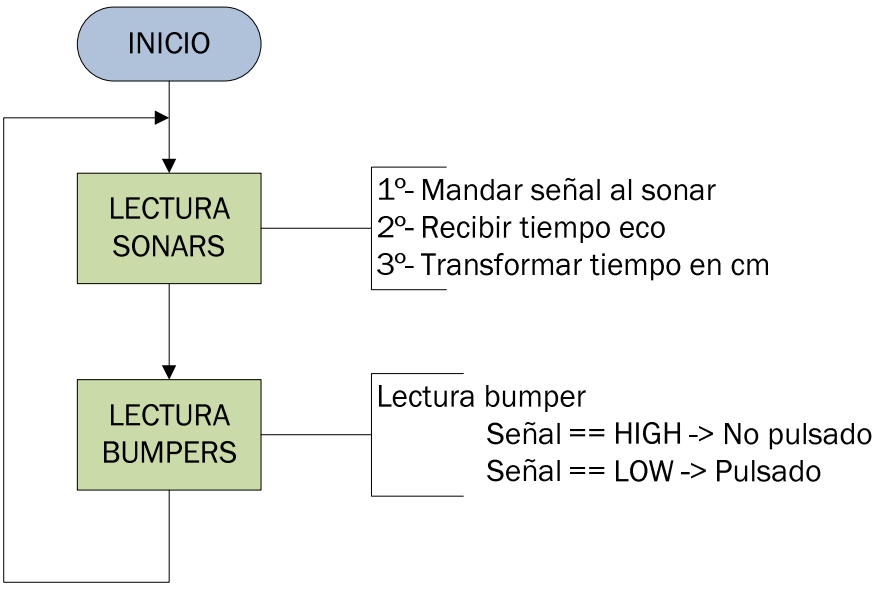


Diagrama de flujo 1. Control de ultrasonidos y bumpers.

4.3. Estado de la batería

El estado de la batería se realiza en el Arduino Mega mediante dos funciones, *"Lectura_bateria()"* y *"Control_estado()",* para evitar saturar el bucle principal de código.

La función *"Lectura_bateria()"* se encarga de leer del pin analógico el valor obtenido a la salida del divisor de tensión. Sin embargo dicho valor no es con el que queremos tratar, es decir, cuando la batería está completamente cargada, el pin analógico no nos da un valor de 12V, sino de 1023. Esto es debido a que los pines analógicos están divididos en 1024 partes, o lo que es lo mismo, los pines analógicos del microcontrolador Arduino pueden distinguir entre 1024 niveles en el rango de 0V a 5V.

Para convertir el valor leído del pin digital en uno que sea útil y más sencillo de tratar, basta con realizar una simple regla de tres, véase Tabla 8.

Valor leído del pin analógico	Salida divisor de tensión	Nivel de carga de la batería
1023	5V	12,5V
x	y	z

Tabla 8. Valor batería.

$$z = \frac{y \cdot 12.5V}{5V}$$

$$y = \frac{5V \cdot x}{1023}$$

Sustituyendo la segunda ecuación en la primera, se obtiene el valor real de la carga de la batería.

$$z = \frac{5V \cdot x \cdot 12.5V}{5V \cdot 1023} = \frac{x \cdot 12.5V}{1023}$$

Una vez se ha conseguido calcular el valor real del nivel de carga de la batería, se deben encender los leds y activar o desactivar las alarmas en función del valor resultante calculado. De ello se encarga la función *"Control_estado()"*, mediante tres bucles *if.*

Si el nivel de carga de la batería es mayor de 11.3V, se encenderá sólo el led verde y no se activará ninguna alarma. Sin embargo, si el nivel de carga disminuye a un valor que se encuentre entre los 11.3V y los 10.8V, se encenderá sólo el led amarillo y se

activará la alarma de batería baja. Sin el nivel de carga disminuye aún más, a un voltaje inferior a los 10.8V, se encenderá sólo el led rojo y se activará la alarma de batería crítica.

Al igual que se hizo con el control del estado de los bumpers y los ultrasonidos, se ha representado mediante un esquema el control del estado de la batería, véase Diagrama de flujo 2, pudiendo comprender rápidamente la estructura del código. Recordar, que aunque en este capítulo no se adjuntan los códigos, el lector puede consultarlos al final de la memoria, en la sección de Anexos, *Anexo 4. Estado de la batería.*

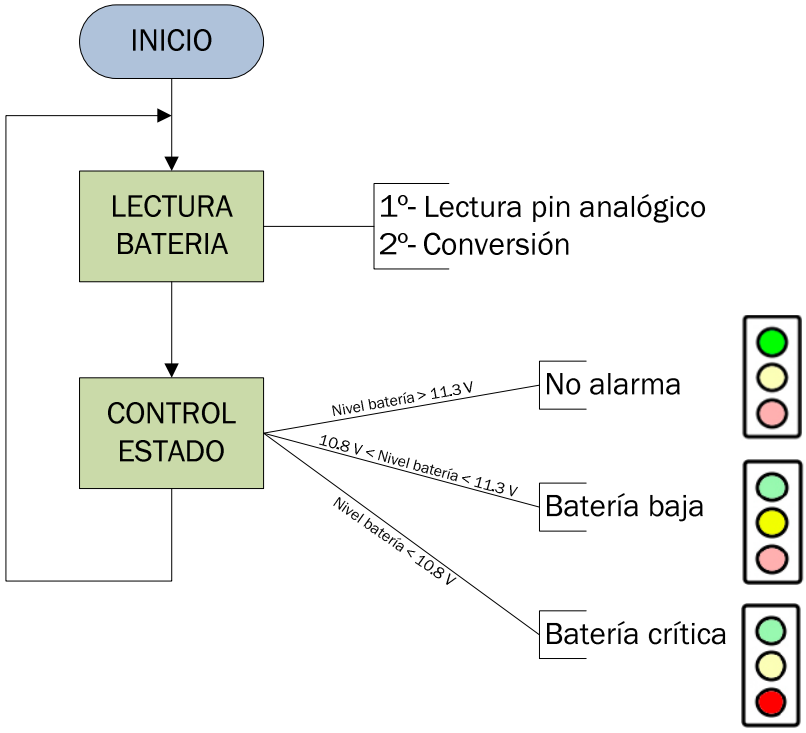


Diagrama de flujo 2. Control del estado de la batería.

4.4. Gestión de alarmas

Aunque pueden programarse más tipos de alarmas, actualmente el robot Edubot ha sido programado para que el control de bajo nivel de una señal de alarma al control de alto nivel en el caso del voltaje de alimentación.

Como ya se ha explicado en el apartado anterior, si la carga de la batería se encuentra entre los 11.3V y los 10.8V se da una alarma de batería baja y si la carga es inferior a 10.8V se da una señal de batería critica.

En el momento en el que el estado de alguna alarma varíe, se deberá notificar al control de alto nivel.

4.5. Control de motores

A continuación, se va a desarrollar el control de los motores de manera gradual, de forma correspondiente al avance que se ha ido realizando a lo largo del proyecto.

Debido a la complejidad de este control, no se pudo desarrollar el mismo de una sola vez, sino que se fueron controlando poco a poco cada uno de los elementos del motor. Con esto se quiere decir que, para controlar los motores del robot no basta con controlar el motor de corriente continua de manera independiente, es necesario controlar también los puentes H y los codificadores incrementales.

En primer lugar, se desarrolló todo el control de forma independiente para cada motor, empezando con el control de los codificadores incrementales, para después controlar los puentes H y finalmente controlar la velocidad de cada motor mediante un control PID. Una vez controlado cada motor de manera independiente, se procedió al control de los dos motores a la vez.

Lectura de codificadores incrementales

En el capítulo 2, ya se explicó el funcionamiento de los codificadores incrementales, sin embargo, no se profundizó demasiado en las señales que mandan al microcontrolador por los canales A y B. Por este motivo, a continuación se desarrolla este tema.

Los codificadores incrementales, suelen tener dos salidas digitales, cada cual es enviada por un canal diferente, canal A y canal B. Cuando el encoder gira, se produce un pulso de onda cuadrado que se envía por estos dos canales y que representa cuánto ha girado el eje del encoder. Si se cuentan el número de pulsos generados desde un determinado momento, se podrá conocer cuánto ha girado el encoder.

Por lo explicado en el párrafo anterior, parece ser sólo necesario el uso de un solo canal, sin embargo, si se utiliza un canal, sólo se conocerá la fracción de giro del encoder pero no en el sentido de giro. Para conocer la distancia recorrida por el robot,

se deberán conocer tanto la distancia recorrida como el sentido del movimiento, por lo que se necesitarán los dos canales.

Para poder conocer el sentido de giro del encoder, o lo que es lo mismo, el sentido de giro de la rueda, se debe tener presente que las señales recibidas de los canales están desfasadas 90º la una de la otra, tal y como se observa en la Figura 26.

Considerando el canal A como señal de referencia, de él contaremos los pulsos para conocer la distancia recorrida por el robot, y para conocer la dirección de giro habrá que fijarse en la posición de los pulsos del canal B respecto al canal A.

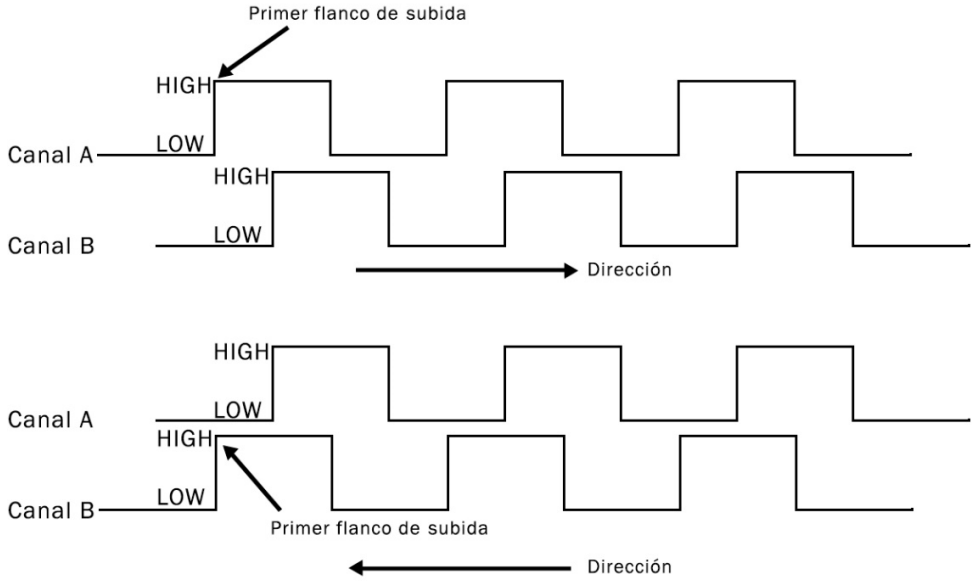


Figura 26. Canales codificador incremental.

Se sabrá que el motor gira hacia delante si cuando se da el flanco de subida en el canal A, el valor de la señal del canal B es un cero lógico. Sin embargo, el motor girará en sentido contrario si cuando se da el flanco de subida en la señal de referencia, la señal del canal B está en alto.

Una vez explicadas las señales que va a recibir el microcontrolador, ya se puede proceder a la explicación del código desarrollado donde se interpretan estas señales.

Los motores, y por tanto los codificadores, se desplazan rápidamente, dando en una única vuelta 490 pulsos. Para poder contarlos todos será necesaria la utilización de interrupciones. Estas interrupciones son utilizadas por Arduino para que cuando se de una determinada circunstancia, en este caso un flanco de subida en las señales

enviadas por el encoder, el microprocesador abandone la tarea que estuviera realizando en ese momento para atender la interrupción. Una vez atendida la interrupción, es decir, una vez se halla actualizado el contador de pulsos, el microcontrolador continúa la tarea que dejó inacabada.

En primer lugar, se desarrolló un programa que tan sólo cuenta los pulsos enviados al microcontrolador. Gracias a este programa, se pudo comprobar que se estaban contando correctamente los pulsos, verificando los siguientes puntos:

- ✓ Moviendo la rueda hacia delante se suman pulsos.
- ✓ Moviendo la rueda hacia atrás se restan pulsos.
- ✓ Se obtienen siempre el mismo número de pulsos tras dar una vuelta completa.
- ✓ Se obtienen el mismo número de pulsos dando una vuelta completa a la rueda en sentido horario y en sentido anti-horario.

En el *Anexo 5. Encoders – Contador de pulsos*, se puede consultar el código desarrollado, en el cual se podrá observar cómo se utilizan las interrupciones para contar el número de pulsos recibidos.

Para ello, se utilizarán las interrupciones con las que cuenta el Arduino Uno. Una vez que al microcontrolador le llega la interrupción, se va a dirigir de manera automática a la función *"Contar ()"*.

En esta función se hace lo siguiente:

- Se leen las señales del canal A y del canal B en el momento de la interrupción. Está señal puede ser o bien HIGH o bien LOW:
  - Si el canal A está a HIGH, pueden darse dos situaciones:
    - Que el canal B esté a LOW, en este caso estaríamos avanzando, por lo que sumaría un pulso al contador.
    - Que el canal B esté a HIGH, en este caso estaríamos retrocediendo, por lo que restaría un pulso al contador.
  - Si el canal A está a LOW, pueden darse dos situaciones:
    - Que el canal B esté a HIGH, en este caso estaríamos avanzando, por lo que sumaría un pulso al contador.
    - Que el canal B esté a LOW, en este caso estaríamos retrocediendo, por lo que restaría un pulso al contador.

Una vez verificados los puntos anteriores se procedió a desarrollar un poco más el código y probarlo moviendo el motor, sin necesidad de mover la rueda con las manos, puesto que los microcontroladores deberán leer correctamente los codificadores de esta forma.

Se trata de un código muy sencillo, el cual se puede ver en el *Anexo 6. Encoder – Contar vuelta*. En este programa se hace dar una vuelta completa a la rueda en un sentido, 490 pulsos, y una vez dada la vuelta completa se cambia el sentido de giro para que la rueda de otra vuelta completa, volviendo de esta manera a la posición inicial.

Control de los puentes H

Una vez verificado el funcionamiento de los codificadores incrementales, se procedió al control de la velocidad del motor y su sentido de giro. A lo largo de este punto, cuando se hable de control de velocidad, se referirá al control de la tensión que se le envía al motor para que el mismo varíe de velocidad y de sentido. En el siguiente punto, se hablará del control de la velocidad en el sentido de mantenerla constante.

Se ha de dejar claro que el puente H sólo sirve para controlar el sentido de giro del motor, en ningún momento controlará el voltaje que le llega al motor para que gire a diferentes velocidades. Siempre funciona a plena potencia. El control del voltaje se realizará a través del PWM del microcontrolador Arduino.

Cada puente H tendrá cuatro entradas y cuatro salidas. Las cuatro entradas están conectadas al microcontrolador Arduino Uno y las cuatro salidas al motor. Estas cuatro entradas son:

- Entrada de habilitación del movimiento, ENABLE.
- GND.
- Pin de modulación de ancho de pulso, PWM.
- Pin de sentido de giro.

A través de la entrada digital de habilitación, se controlarán cuando se pueden mover los motores. Si está entrada está a LOW, su motor correspondiente no podrá moverse, sin embargo, si está a HIGH, si podrá.

Mediante el pin PWM podremos controlar la velocidad del motor. Cuanto mayor sea el valor introducido por este canal, mayor será la velocidad a la que gire la rueda. Este valor estará limitado, ya que la máxima velocidad permitida en el motor integrado en el robot es de 255 en ambos sentidos.

Por último, el pin de sentido de giro será el que esté comunicado con los transistores que forman el puente H. Se trata de un pin digital que al estar en estado LOW o bien permanecerá parado o bien girará en sentido inverso, es decir, marcha atrás. Sin embargo, si está un valor de HIGH, el motor girará en sentido contrario, o lo que es lo mismo, hacía delante.

Este control se ha realizado de manera independiente para cada motor, es decir, hay un programa para el motor derecho y otro para el motor izquierdo y ambos se pueden ver en el *Anexo 7. Control de puentes H.*

En ambos programas el motor comienza en estado de reposo y al leer el Arduino del puerto serie actúa en función del comando mandado:

- 0, el motor se para.
- 1, aumenta su velocidad.
- 2, reduce su velocidad.

Si la velocidad resultante es positiva, girará en sentido de avance y si la velocidad es negativa el motor girará en sentido contrario.

Control PID de la velocidad

Sabiendo controlar la velocidad del motor y su sentido de giro, es necesario mantener estable dicha velocidad. Cuando el robot comience a andar por el suelo, es muy importante que ambas ruedas giren a la velocidad establecida de la manera más constante posible, ya que es necesario que el robot ande en línea recta en caso de que sea necesario. Para que esto pueda suceder, ambas ruedas deberán alcanzar la velocidad establecida de la manera más rápida e igualitaria posible, además de mantener dicha velocidad de manera indefinida. Para poder realizar todo esto, se ha optado por un control PID.

Un controlador PID es un mecanismo de control por realimentación, cuyo diagrama de bloques se podría representar según se observa en la Figura 27.

Como se observa en la Figura adjuntada, al PID le llega una señal de error, la cual se obtiene comparando el valor real y el valor de referencia. Con este error, el PID actúa de la forma programada para minimizarlo lo más posible.

El control PID corrige este error mediante la combinación de tres acciones diferentes, acción integral, proporcional y derivativa.

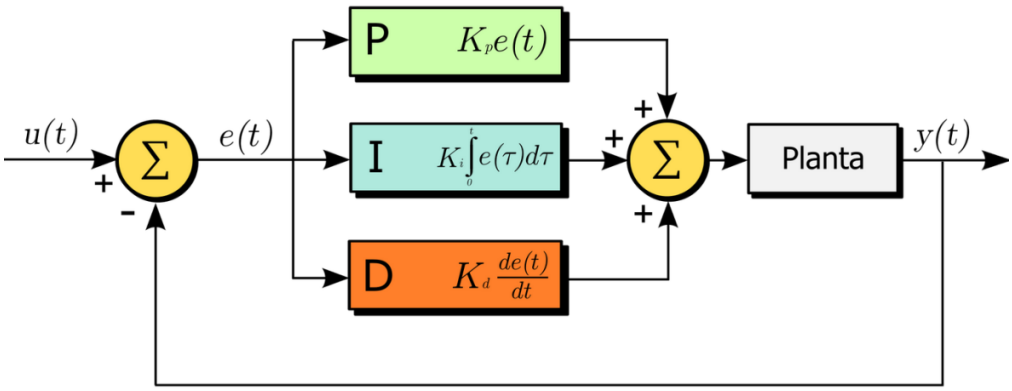


Figura 27. Control PID.

Acción proporcional

La acción proporcional posiblemente es la acción más importante. De hecho, existen controles PID cuya única acción que interviene es la proporcional. Está acción da como resultado una salida que es proporcional al error, multiplicándolo por un valor, llamado ganancia o constante proporcional, K_p.

Partiendo de los parámetros que se observan en la Figura 27, si sólo actuará la acción proporcional, la salida del controlador sería la que se representa en la siguiente ecuación.

$$y(t) = e(t) \cdot K_p \qquad \text{(Ec.2)}$$

Gracias a esta acción, conseguiremos que los motores alcancen el valor de referencia rápidamente. Sin embargo, hay que tener cuidado al dar valores a la constante proporcional, ya que si este valor es excesivamente alta, se correrá el riesgo de que haya un sobre-amortiguamiento, es decir, el PID hará que el motor gire a una velocidad excesiva.

Un claro ejemplo de lo anteriormente explicado se puede observar en la Figura 28. Se puede observar como al aumentar la acción proporcional, la señal alcanza el valor estacionario más rápidamente, sin embargo, valores demasiado elevados pueden provocar sobre-amortiguamiento.

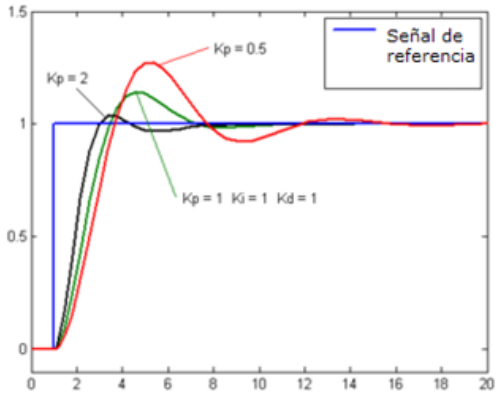


Figura 28. Control proporcional.

Acción integral

Esta acción da como resultado una salida que es proporcional al error acumulado. Para poder obtener este resultado será necesario integrar el valor del error durante un tiempo determinado y multiplicarlo por la constante integral, K_i.

Partiendo de los parámetros que se observan en la Figura 27, si sólo actuará la acción integral, la salida del controlador sería la que se representa en la siguiente ecuación.

$$y(t) = K_i \cdot \int e(t) \cdot d\tau \qquad \text{(Ec.3)}$$

Gracias a esta acción, se conseguirá eliminar el error existente en estado estacionario respecto al valor de referencia. Sin embargo, hay que aplicar esta acción con precaución, ya que se obtiene una respuesta más lenta y el periodo de oscilación aumenta. Todo lo explicado se puede observar en la Figura 29.

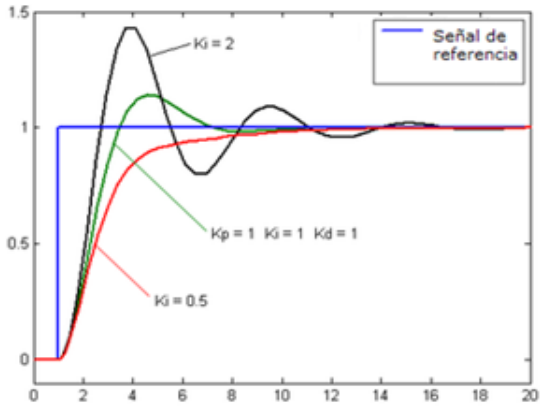


Figura 29. Control integral.

Acción derivativa

Por último, está la acción derivativa, la cual da como resultado un valor proporcional a la derivada del error. Dicha derivada representa la velocidad a la que cambia el error existente. Para poder obtener este resultado será necesario derivar el error absoluto para posteriormente multiplicarlo por la constante derivativa, K_d. Gracias a esta acción, se conseguirá amortiguar las oscilaciones producidas.

Partiendo de los parámetros que se observan en la Figura 27, si sólo actuará la acción derivativa, la salida del controlador sería la que se representa en la siguiente ecuación.

$$y(t) = K_d \cdot \frac{de(t)}{dt} \qquad \text{(Ec.4)}$$

En la Figura 30 se puede observar cómo actúa la acción derivativa en función de la constante derivativa.

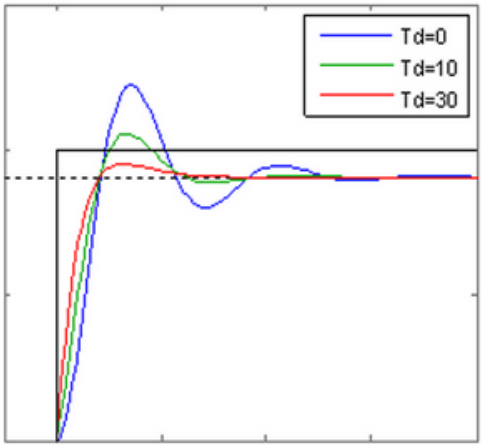


Figura 30. Control derivativo.

Mediante la combinación de todas estas acciones, se conseguirá un control de la velocidad de los motores adecuado. A la hora de programar este control, hay que tener muy presentes todas las ventajas y desventajas que se pueden producir al aumentar o disminuir los valores de todas las constantes, por lo que se han realizado dos tablas resumen, véase Tablas 9 y 10, con las ventajas e inconvenientes de cada control y con la repercusión que tiene cada constante.

	Ventajas	Desventajas
Control proporcional	Error en estado estacionario disminuye. El proceso responde más rápidamente.	La sobre-amortiguación y las oscilaciones aumentan.
Control integral	Se elimina el error en estado estacionario.	Cuanto menor sea la constante, habrá más oscilaciones. Da inestabilidad.
Control derivativo	Amortiguación de las oscilaciones.	Gran sensibilidad al ruido.

Tabla 9. Ventajas y desventajas de los controles PID.

	K_p ↑	K_i ↑	K_d ↑
Estabilidad	Disminuye	Disminuye	Aumenta
Velocidad	Aumenta	Aumenta	Aumenta
Error en estado estacionario	No se elimina	Se elimina	No se elimina

Tabla 10. Variación de las constantes PID.

Una vez explicado el funcionamiento del control PID, ya se puede proceder a explicar la forma en como se ha implementado dicho control en el robot Edubot.

Este control se ha realizado de manera independiente para cada motor, es decir, se ha desarrollado un programa para el motor derecho y otro para el motor izquierdo, para posteriormente unificarlos (esto se desarrollará en el siguiente punto).

En primer lugar, comentar que el programa trabaja con velocidades angulares, y el motor se inicializa para que se mueva a una velocidad angular de referencia no nula.

El bucle principal consta de dos funciones principales, *"Compute ()"* y *"Girar (Velocidad)"*. La función *"Compute ()"* se encarga del control PID y la función *"Girar (Velocidad)"* se encarga de mover el motor en el sentido y a la velocidad adecuada.

Función *"Compute ()"*

Esta función está dividida en dos secciones. La primera, la cual se ejecuta cada 0.2 segundos, encargada de actualizar el valor de la velocidad angular real y la segunda, dedicada al cálculo del PID.

Para calcular el valor de la velocidad angular hay que seguir los siguientes pasos de manera ordenada:

- En primer lugar, se calcula el diferencial de tiempo, el cual sigue la siguiente expresión:

$$dt = Tiempo\ actual - Tiempo\ anterior \quad \text{(Ec.5)}$$

- Posteriormente, se calcula el diferencial de pulsos, cuyo resultado se obtiene siguiendo la siguiente ecuación:

$$dpulsos = Contador\ de\ pulsos\ actual - Contador\ de\ pulsos\ anterior \quad \text{(Ec.6)}$$

- Por último, se calcula la velocidad angular de la siguiente manera:

$$\omega = \frac{dpulsos}{dt} \quad \text{(Ec.7)}$$

Como se puede observar, es necesario conocer cuanto a girado la rueda, por lo que será necesario el uso de la función ya explicada anteriormente *"Contar()"*.

Una vez conocida la velocidad angular real a la que gira la rueda, se puede proceder al control PID, comparando este valor obtenido con el valor de referencia.

El control PID se ha programado completo, es decir, se ha programado tanto la acción proporcional, como la integral y la derivativa. Su uso o no uso se podrá elegir posteriormente al dar valores a las constantes proporcional, integral y derivativa. Para ello, se ha seguido el siguiente esquema:

- Primero, se calcula el error absoluto sobre el cual, intervendrán todas las acciones de control.

$$Error\ absoluto = Valor\ de\ referencia - Valor\ real \quad \text{(Ec.8)}$$

- En segundo lugar, se calcula el error proporcional siguiendo la siguiente expresión:

$$Error\ proporcional = Error\ absoluto \cdot \ K_p \quad \text{(Ec.9)}$$

- En tercer lugar, se procede al cálculo del error integral.

$$Error\ integral += Error\ absoluto \cdot dT \cdot K_i \quad \text{(Ec.10)}$$

- En cuarto lugar, se calcula el error derivativo.

$$Error\ derivativo = \frac{Error\ abs\ actual - Error\ abs\ anterior}{dT} \cdot K_d \quad \text{(Ec.11)}$$

- Una vez calculados todos los errores, se suman, dando lugar al error total del PID del motor que está siendo analizado, el cual equivale a la velocidad que hay que introducir al motor para que la misma se mantenga constante.

$$Error\ total = Error\ proporcional + Error\ integral + Error\ derivativo \quad \text{(Ec.12)}$$

Una vez explicados todos los pasos para el control PID, es necesario mencionar tres detalles.

En primer lugar, la acción integral no debe realizarse siempre. No se sumará en aquellos casos en los que dando un valor de referencia no nulo, los motores no se

muevan. De esta manera, se evitará que la acción integral aumente de manera desproporcionada al apagar la batería o bien al darle valores muy pequeños en los que el motor es incapaz de moverse por el poco voltaje que le llega.

En segundo lugar, aun aplicando las precauciones tomadas en el punto anterior, puedo ocurrir que el error integral incremente su valor hasta valores excesivos, eso provoca el efecto llamada *"windup".* Otra forma de evitarlo, es recalculando el error integral. Cuando el error integral supera los márgenes de velocidad permitidos, es decir, cuando la salida se satura, igualamos el error total a este valor límite. De esta forma, proporcionamos una salida en el límite de la saturación.

Además, para valores PWM muy bajos en los cuales el motor no es capaz de moverse, se ha establecido un error integral nulo. De esta forma, se evitan comportamientos anómalos a bajas velocidades.

Por último, al calcular el error total del PID, o lo que es lo mismo, al calcular la velocidad que se ha de introducir al motor, se ha tenido en cuenta la posibilidad de que la esta supere los márgenes admisibles por el motor, por lo que se ha limitado el resultado en caso de que esto ocurriese.

A continuación, se ha representado todo este control mediante un diagrama de bloques, véase Diagrama de flujo 3, para poder ver la estructura del programa de manera clara y sencilla y en el *Anexo 8. Control de la velocidad de cada motor*, se podrá ver el código desarrollado.

Se puede observar como todo lo desarrollado a lo largo de este punto, Control de motores, se encuentra en este código, empezando desde la lectura de los codificadores incrementales, *"Interrupción debida al encoder",* continuando por el control de los puentes H, *"Giro de motor"*, y finalizando por el control PID.

En el *Capítulo 7. Resultados experimentales* se explicará cómo se obtuvieron las constantes de control PID y cuáles fueron los resultados obtenidos.

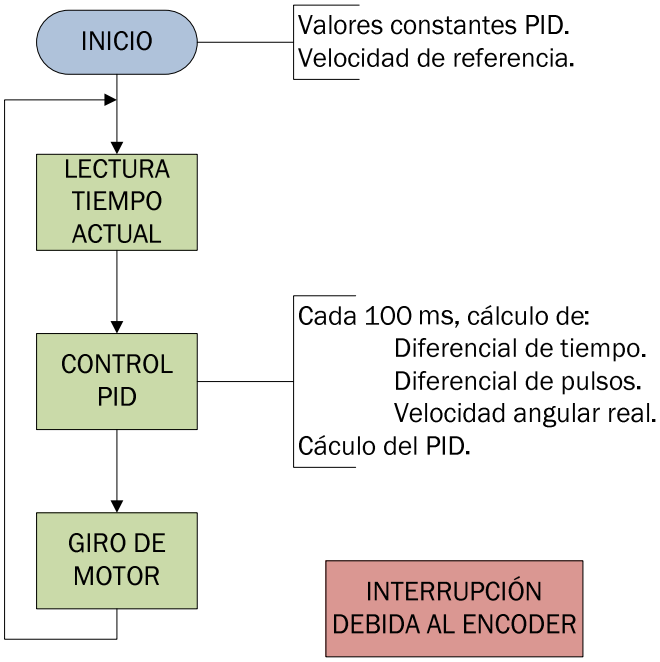


Diagrama de flujo 3. Control de motores individual.

4.6. Control global

Por último, se unificaron todos los controles en tres códigos, uno destinado al Arduino Uno, otro destinado al Arduino Mega del motor izquierdo y el resto al Arduino Mega maestro.

El código del Arduino Uno es el descrito en el punto *4.2 Recepción de ultrasonidos y bumpers*, al cual se le ha añadido la comunicación I2C para poder enviar los datos obtenidos al Arduino Mega maestro (comunicación que se explicará en el próximo capítulo). Este código puede verse desarrollado en el *Anexo 9. Código Arduino Uno.*

El código Arduino Mega esclavo, es el descrito en el punto *4.5 Control de PID de la velocidad*, al cual se le ha añadido la comunicación Puerto Serie para poder enviar la velocidad medida y el número de pulsos dados al Arduino Mega maestro (comunicación que también se explicará en el próximo capítulo).

Por último, está el código que debe ser introducido al Arduino Mega maestro, microcontrolador principal, en el cual se tuvieron que implantar:

- Estado de las baterías.
- Gestión de alarmas.
- Control del motor derecho.

Además de la unificación de todos estos controles, se tuvieron que añadir dos tipos de comunicación:

- Comunicación I2C, para comunicar el Arduino Mega con el Arduino Uno.
- Comunicación puerto serie, para comunicar el Arduino Mega con el alto nivel (ordenador) y con el otro Arduino Mega.

Toda esta comunicación, se explica más detalladamente en el próximo capítulo *5. Comunicación del Arduino Mega*.
Comentar que en el *Anexo 10. Código Arduino Mega esclavo* y en el *Anexo 11.Código Arduino Mega Maestro*, se encuentra el código final introducido a estos dos microcontroladores.

4.7. Localización odométrica.

Aunque la información que se va a abordar a continuación no se realiza a bajo nivel, se ha considerado importante explicar la localización odométrica del robot Edubot, debido a que ocupa un lugar de suma importancia dentro del control de cualquier robot.

La odometría estima la posición de cualquier vehículo con ruedas durante su movimiento, conociendo la rotación de cada rueda a lo largo del tiempo. Se trata de un sistema de localización muy barato y sencillo de implementar, que puede llegar a resultar muy preciso en periodos cortos de tiempo y en lugares donde el robot pueda desplazarse sin dificultad.

Para poder realizar esta estimación, será necesaria la utilización de ecuaciones diferenciales, de forma que se podrá saber la posición del robot dentro del plano XY.

Para poder comprender todo el desarrollo y uso de las ecuaciones que se van a analizar a continuación, hay que tener presente la localización de las ruedas y sus formas de movimiento. En la Figura 31, se puede recordar que el robot está dotado de dos ruedas motrices, las cuales se encargan del movimiento del robot, y dos ruedas locas.

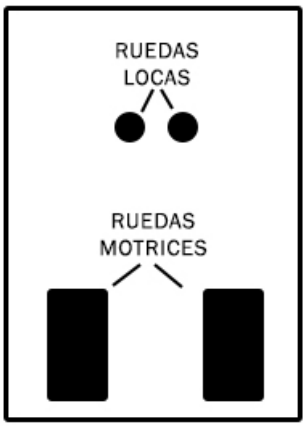


Figura 31. Localización de las ruedas.

Las ruedas motrices sólo pueden moverse hacia delante y hacia atrás, por lo que para girar será necesario variar las velocidades de cada rueda de forma que si una gira más rápido que la otra, el robot tenderá hacia el lado de la rueda más lenta, siendo este giro más brusco cuanto mayor sea la diferencia de velocidad entre las ruedas.

A continuación, se va a abordar el desarrollo de la cinemática diferencial, para posteriormente comprender todos los posibles errores que se pueden cometer con este tipo de localización.

Cinemática diferencial

El robot envía al control de bajo nivel los pulsos dados en un periodo de tiempo determinado gracias a los codificadores implantados en cada motor. Hay que tener en cuenta que la información obtenida de los mismos, nos indica cuanto ha girado el eje de cada motor, es decir, se trata de información del movimiento angular de la rueda. Por este motivo, será necesario que el control de alto nivel realice una conversión de este movimiento angular a un movimiento lineal, de forma que, conociendo cuanto recorre la rueda por cada pulso avanzado por el eje, se sabrá el desplazamiento realizado en los pulsos dados en un periodo de tiempo. Esta conversión se realiza mediante la siguiente ecuación:

$$\frac{Desplazamiento\ lineal}{pulso} = \frac{2 \cdot \pi \cdot R}{Resolución} \qquad \text{(Ec.13)}$$

Donde:

- R, es el radio de la rueda.
- Resolución, es la resolución total del encoder, es decir, teniendo en cuenta la resolución del encoder y la reducción aplicada.

Una vez conocido el desplazamiento realizado por cada pulso girado, se podrá saber fácilmente el desplazamiento de cada rueda en un periodo de tiempo determinado, teniendo en cuenta que en el control de bajo nivel ya se ha calculado el diferencial de pulsos. De esta forma, la distancia lineal recorrida en un intervalo de tiempo por cada rueda vendrá representada mediante las siguientes ecuaciones, rueda derecha y rueda izquierda respectivamente.

$$\frac{d\ despl\ lineal\ der}{dt} = \frac{2 \cdot \pi \cdot R}{Resolución} \cdot dpulsos\ derecho$$
$$\frac{d\ despl\ lineal\ izq}{dt} = \frac{2 \cdot \pi \cdot R}{Resolución} \cdot dpulsos\ izquierdo \qquad \text{(Ec.14)}$$

Una vez conocido el desplazamiento lineal de cada rueda, es necesario conocer el desplazamiento lineal del centro del robot y el ángulo girado. Todo ello se puede representar mediante las dos siguientes fórmulas.

$$\frac{d\ despl\ lineal}{dt} = \frac{\frac{d\ despl\ lineal\ der}{dt} + \frac{d\ despl\ lineal\ izq}{dt}}{2}$$
$$\frac{d\ giro\ del\ robot}{dt} = \frac{d\theta}{dt} = \frac{\frac{d\ despl\ lineal\ der}{dt} + \frac{d\ despl\ lineal\ izq}{dt}}{L} \qquad \text{(Ec.15)}$$

Donde L es la longitud del eje que une las dos ruedas.

Conocidos los desplazamientos lineal y angular del robot, es muy sencillo obtener a partir de trigonometría el ángulo girado y los desplazamientos X e Y del robot.

$$X = X_{anterior} + \frac{d\ desplazamiento\ lineal\ del\ robot}{dt} \cdot cos\theta$$
$$Y = Y_{anterior} + \frac{d\ desplazamiento\ lineal\ del\ robot}{dt} \cdot sen\theta \qquad \text{(Ec.16)}$$
$$\theta = \theta_{anterior} + \frac{d\theta}{dt}$$

Posibles errores

Vista la forma de calcular la odometría, se comprende la sencillez de este método para conocer la localización del robot. Sin embargo, también es fácil deducir el porqué de su gran exactitud sólo en periodos de tiempo pequeños. Al estar tratando continuamente con ecuaciones diferenciales, los errores se van acumulando de manera proporcional a la distancia recorrida.

Hay que tener en cuenta todas las variables introducidas a lo largo de todas las ecuaciones anteriores, para poder observar la gran cantidad de errores que se pueden acumular. A continuación, se van a resumir una serie de errores muy comunes que pueden dar lugar a una localización del robot incorrecta:

- A la hora de calcular el desplazamiento lineal por pulso se ha utilizado como variable el radio de la rueda. Hay que tener en cuenta que la medida del diámetro de la rueda suele diferir, en mayor o menor medida, de la medida dada de fábrica, por lo que además de este error, hay que añadir la posibilidad de que la rueda derecha y la izquierda tengan diámetros diferentes.
- Para calcular el ángulo de giro, ha sido necesario introducir la longitud existente entre las ruedas, por lo que hay que tener en cuenta la posibilidad de que las ruedas no estén perfectamente alineadas, dando lugar a ángulos incorrectos.
- También hay que fijarse en que los datos obtenidos del encoder no son del todo exactos, ya que tanto su resolución como su muestreo es discreto.

A todos estos errores sistemáticos, se deben añadir otra serie de errores no sistemáticos, que dependen del terreno de desplazamiento del robot.

- Si el robot se desplaza por suelos con desniveles, al no haber contacto de la rueda con el suelo, aunque el robot continúe avanzando, al no girar una de las ruedas, el posicionamiento sería incorrecto.
- También se puede dar el caso de que alguna de las ruedas patine, con su consecuente error de localización.

Como el lector puede deducir, hay algunos errores, los sistemáticos, que se pueden corregir modificando ligeramente el valor de algunas variables del código. Estas variables son las referentes al radio de las ruedas y la distancia existente entre ellas.

Todo este conjunto de modificaciones es la denominada calibración de la odometría. Consta de varios procesos de calibrado que deben realizarse en orden. Si una de estas calibraciones falla, deberá volver a ejecutarse todo el proceso de calibración desde el principio.

Calibrar distancia

En este proceso de calibrado se deben comparar dos datos, uno real y otro calculado. Dichos datos hacen referencia a la distancia recorrida por el robot.

Para realizar esta calibración, se debe colocar al robot en una pista recta, de longitud conocida, y programarle para que circule en línea recta dicha distancia. Una vez realiza esta trayectoria pueden ocurrir tres cosas:

- La distancia de la pista y la distancia recorrida sean las mismas. En este caso no es necesario modificar ningún parámetro.
- La distancia recorrida es menor que la de la pista. Esto significa que el control de alto nivel piensa que la distancia ya ha sido alcanzada sin ser cierto, es decir, las longitudes por vuelta de las ruedas, *desplazamiento lineal del robot*, son demasiado grandes. Para corregirlo, habrá que disminuir el valor de las variables que representan los radios de las ruedas derecha e izquierda.
- La distancia recorrida es mayor que la de la pista. Esto significa que el control de alto nivel piensa que la distancia todavía no ha sido alcanzada cuando realmente sí que había sido alcanzada, es decir, las longitudes por vuelta de las ruedas, *desplazamiento lineal del robot*, son demasiado pequeñas. Para corregirlo, habrá que aumentar el valor de las variables que representan los radios de las ruedas derecha e izquierda.

En la Figura 32, se puede ver representado este proceso de calibración.

Figura 32. Calibración distancia.

Calibrar desviaciones

Tras calibrar correctamente la distancia, es necesario comprobar si en la odometría calculada el robot se desvía, es decir, si cuando debe ir en línea recta, los datos calculados indican que está girando a la derecha o a la izquierda. Normalmente, estas

desviaciones suelen ser muy livianas, sin embargo, al tratarse de errores que se van acumulando a lo largo de tiempo, el error que al principio parecía casi insignificante, pasado un tiempo este error puede ser muy grande. Por este motivo, es de vital importancia corregirlo.

De nuevo, se programa al robot para que circule en línea recta, y se visualiza la trayectoria calculada por el control de alto nivel. Finalizada la prueba, pueden ocurrir 3 cosas:

- La trayectoria calculada sea en línea recta. En este caso, no será necesario modificar ninguna variable.
- La trayectoria calculada se desvía hacia la derecha. Esto significa que el control de alto nivel piensa que la rueda derecha ha recorrido una menor distancia lineal que la rueda izquierda. Para corregir este error, será necesario disminuir la variable que representa el radio de la rueda derecha.
- La trayectoria calculada se desvía hacia la izquierda. Esto significa que el control de alto nivel piensa que la rueda izquierda ha recorrido una menor distancia lineal que la rueda derecha. Para corregir este error, será necesario disminuir la variable que representa el radio de la rueda izquierda.

En la Figura 33, se representa este proceso de calibrado.

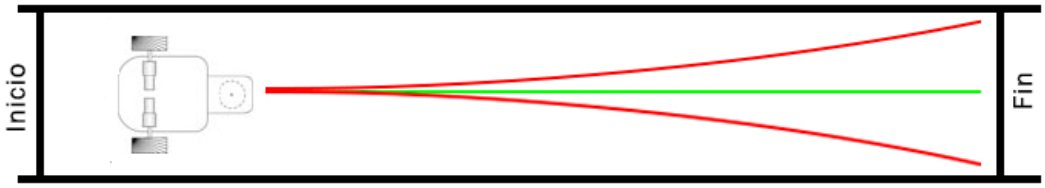


Figura 33. Calibración desvíos.

Calibrar giros

Llegados a este punto, se presupone la correcta calibración de las variables relacionadas con los tamaños de las ruedas, por lo que ya se puede programar al robot para que realice trayectorias más complejas que una línea recta.

Para calibrar el ángulo girado, se programará al robot para que realice una trayectoria cuadrada, tal y como se observa en la Figura 38. Una vez terminada la ejecución de esta tarea, pueden ocurrir tres cosas:

- Que la trayectoria realizada sea la ideal, es decir, acaba el recorrido en el mismo punto en el que empezó. En este caso, no es necesario modificar ninguna variable.
- Que la trayectoria realizada sea abierta, es decir, acaba el recorrido en un punto fuera del cuadrado. Esto significa que el control de alto nivel piensa que el ángulo realizado es mayor al realmente realizado. Si el lector analiza la ecuación anteriormente explicada, Ec.15, puede observar como el ángulo de giro depende tanto de los desplazamientos lineales realizados por cada rueda, es decir, del radio de cada rueda, y de la distancia entre ruedas. Suponiendo que los radios de las ruedas ya han sido corregidos, el parámetro erróneo y que debe modificarse es el de la distancia entre ruedas. Puesto que el ángulo calculado es mayor que el real, para disminuirlo, será necesario aumentar la variable que representa la distancia entre ruedas.
- Que la trayectoria realizada sea cerrada, es decir, acaba el recorrido en un punto dentro del cuadrado. Esto significa que el control de alto nivel piensa que el ángulo realizado es menor al realmente realizado. Para corregir este error, siguiendo el mismo razonamiento anterior, será necesario disminuir la variable que representa la distancia entre ruedas.

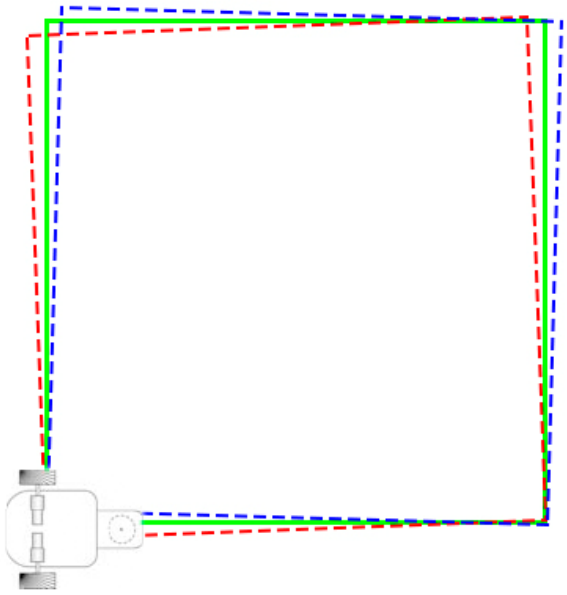


Figura 34. Calibración giros.

Por último, para mayor rapidez a la hora de realizar todos procesos de calibrado, puede resultar de gran utilidad realizar una tabla resumen donde se especifiquen

todos los procesos de calibrado y que modificaciones se deben realizar en función del resultado obtenido. Dicha resumen puede observarse en la Tabla 11 adjunta.

Proceso de calibrado	Resultado obtenido	Solución
Calibración de distancias	Distancia recorrida = Distancia de la pista	No modificar nada
	Distancia recorrida > Distancia de la pista	↑ Radios
	Distancia recorrida < Distancia de la pista	↓ Radios
Calibración de desvíos	No hay desvío	No modificar nada
	Desvío hacia la derecha	↑ Radio rueda derecha
	Desvío hacia la izquierda	↑ Radio rueda izquierda
Calibración de giros	Trayectoria ideal	No modificar nada
	Trayectoria cerrada	↓ Distancia entre ruedas
	Trayectoria abierta	↑ Distancia entre ruedas

Tabla 11. Cuadro resumen de proceso de calibración.

5. COMUNICACIÓN DEL ARDUINO MEGA

Como ya se ha dicho a lo largo de la presente memoria, existen dos controles, el de bajo nivel, formado por dos microcontroladores Arduino, y el de alto nivel, ejecutado por un ordenador.

Entre los tres microcontroladores que forman el control de bajo nivel, hay uno en el que se ejecutan muchas de las funciones principales, Arduino Mega maestro. Este microcontrolador es el principal, es el que se comunica con el control de alto nivel y el que recibe información del Arduino Mega esclavo y del Arduino Uno.

A continuación, se va a explicar cómo se ha realizado la comunicación entre los tres Arduinos, desde el método utilizado hasta la información intercambiada, y después se expondrá la comunicación entre el Arduino Mega y el ordenador, haciendo especial hincapié en el protocolo de comunicación.

5.1. Comunicación Arduino Mega maestro – Arduino Uno

Comunicación I2C

La comunicación existente entre el Arduino Mega maestro y el Arduino Uno es de tipo I2C.

I2C es un bus de comunicación de tipo multi-maestro, permitiendo que haya varios dispositivos que actúen como maestro y varios dispositivos que actúen como esclavo dentro del mismo bus. En este caso, habrá un maestro y un esclavo, los cuales serán el Arduino Mega y el Arduino Uno, respectivamente.

El bus I2C está formado por tres líneas:

- Masa, GND. Esta línea debe ser común para todos los dispositivos que estén conectados al bus I2C.
- Datos, SDA. Por esta línea circulan los datos que se envían entre los dispositivos.
- Reloj, SCL. Esta línea sirve de sincronización mediante pulsos de reloj.

Un posible esquema de este tipo de bus se puede observar en la Figura 35, en el cual sólo hay conectados un maestro y un esclavo, pudiendo existir múltiples dispositivos maestro y esclavo conectados al bus.

Al existir una única línea de datos, hay que indicar siempre hacía donde va dirigido cada dato y hay que realizar un control de acceso al bus, puesto que el bus sólo puede transmitir un dato.

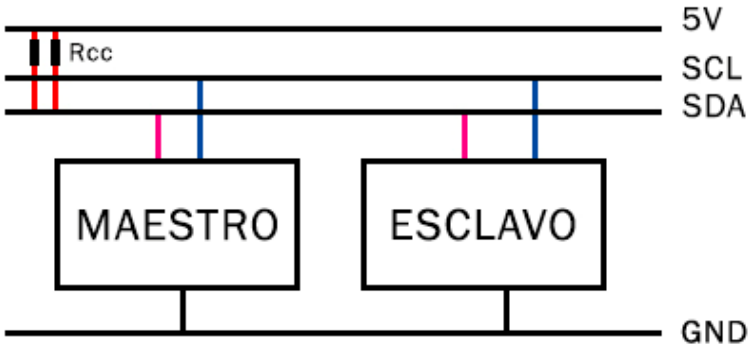


Figura 35. Comunicación I2C.

Los dispositivos maestro son los únicos que pueden iniciar una comunicación, o lo que es lo mismo, son los únicos que pueden controlar la línea SCL, por tanto, los dispositivos esclavo siempre están a la espera de recibir una petición de datos, para poder enviar información al maestro por la línea SDA.

Al sólo poder enviar un solo dato por el bus I2C, para que un dispositivo maestro pueda inicializar la comunicación, el bus debe estar libre, lo cual se indicará mediante las líneas SDA y SCL. Cuando estas dos líneas estén inactivas, es decir, en nivel lógico alto, el bus estará libre.

Si un maestro quiere empezar a comunicarse con otro dispositivo y están las líneas SDA y SCL inactivas, deberá ocupar el bus, indicándoselo al resto de dispositivos conectados al bus activando la línea SDA. Hasta que este dispositivo no libere el bus, ningún otro dispositivo podrá empezar a comunicarse. Este proceso puede verse en la Figura 36.

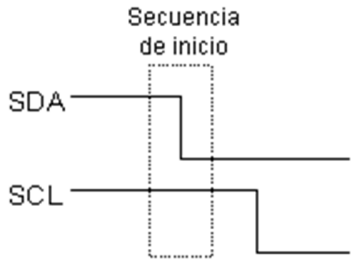


Figura 36. Inicio de comunicación I2C.

Una vez inicializada la comunicación, el maestro envía un byte formado por 8 bits, véase Figura 37, de los cuales 7 sirven para indicar la dirección del dispositivo con el

que quiere intercambiar información y el octavo bit sirve para indicar el tipo de acción que se quiere realizar, a saber, lectura (nivel alto) o escritura(nivel bajo). Si el esclavo se encuentra en disposición de comunicarse, envía una señal ACK a nivel bajo, indicándole al maestro que puede iniciar la transferencia de datos. En caso contrario, la señal de ACK mandada estaría a nivel alto, y el maestro no podría intercambiar información.

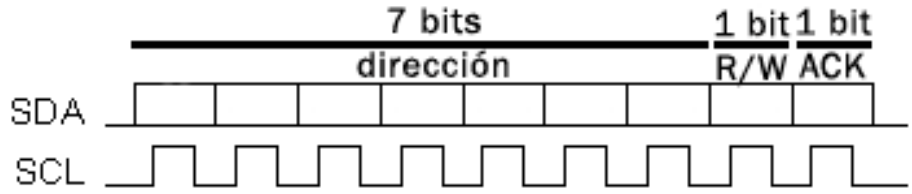


Figura 37. Bits de direccionamiento.

Si el bit ACK que le llega al maestro está a nivel bajo, existen dos opciones:

- El bit de lectura/escritura, R/W, enviado por el maestro está en modo escritura. En este caso, el maestro escribe en el bus I2C los datos que quiere transmitir, recibiendo por parte del esclavo señales ACK, sabiendo de esta forma si los datos se han enviado o no correctamente.
- El bit de lectura/escritura, R/W, enviado por el maestro está en modo lectura. En este caso, el maestro genera pulsos de reloj de forma que el dispositivo esclavo pueda enviar los datos que el maestro quiera leer. Al igual que ocurría en el caso anterior, el que recibe los datos, en este caso el maestro, va generando señales ACK para indicar al esclavo que está recibiendo correctamente los datos.

Una vez terminada toda la transmisión de datos, el maestro debe liberar el bus para que otros puedan utilizarlo. Para liberarlo será necesario poner la línea SDA en nivel lógico alto, tal y como se observa en la Figura 38.

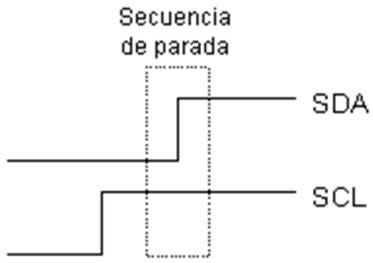


Figura 38. Fin de comunicación I2C.

Durante todo el proceso de comunicación hay que tener en cuenta que las únicas sentencias en las que está permitida la modificación del estado de la línea SDA mientras la línea SCL está también en alto son las sentencias de inicio y fin de comunicación. En el resto de los casos, la línea SDA debe permanecer estable mientras la línea SCL está en valor lógico alto.

Como se ha dicho anteriormente, y se puede observar en la Figura 38, existen 7 bits de direccionamiento. Un número de 7 bits puede abarcar desde el 0 hasta el número 127, esto significa que el bus puede diferenciar entre 128 direcciones diferentes, por tanto, se pueden conectar hasta 128 dispositivos diferentes al bus I2C.

$$2^6 + 2^5 + 2^4 + 2^3 + 2^2 + 2^1 + 2^0 = 127$$

A modo resumen, el protocolo seguido en una comunicación I2C sigue los siguientes puntos:

1) El maestro envía una secuencia de inicio.
2) El maestro envía la dirección de esclavo con el que quiere intercambiar información y el tipo de acción que quiere llevar a cabo.
3) El esclavo envía ACK, indicándole al maestro si está o no dispuesto a realizar la comunicación.
4) Si el esclavo da el visto bueno, el maestro indica el número de registro al que va a escribir o desde el que quiere leer.
5) El maestro envía al esclavo el byte de datos.
6) Una vez terminado el proceso de comunicación, el maestro envía una secuencia de parada.

Comunicación I2C en Arduino

Explicada la comunicación I2C de forma teórica, a continuación se procede a la explicación de la implementación de este tipo de comunicación en el robot Edubot.

En primer lugar, hay que realizar el conexionado adecuado para una correcta comunicación. Como ya se ha visto, es necesario el uso de cuatro líneas, SDA, SCL, 5V y GND. La conexión de los dos Arduinos a las dos últimas líneas es muy sencillo, basta con conectar entre ellos los pines de 5V y GND. La conexión de las líneas de datos y de pulso de reloj es igual de sencilla, sin embargo, hay que tener en cuenta que en el Arduino Uno las líneas ACK y SCL se encuentran en los pines analógicos 4 y 5, respectivamente, y en el Arduino Mega en los pines digitales 20 y 21, respectivamente. Para mayor aclaración, se ha representado este cableado en la Figura 39.

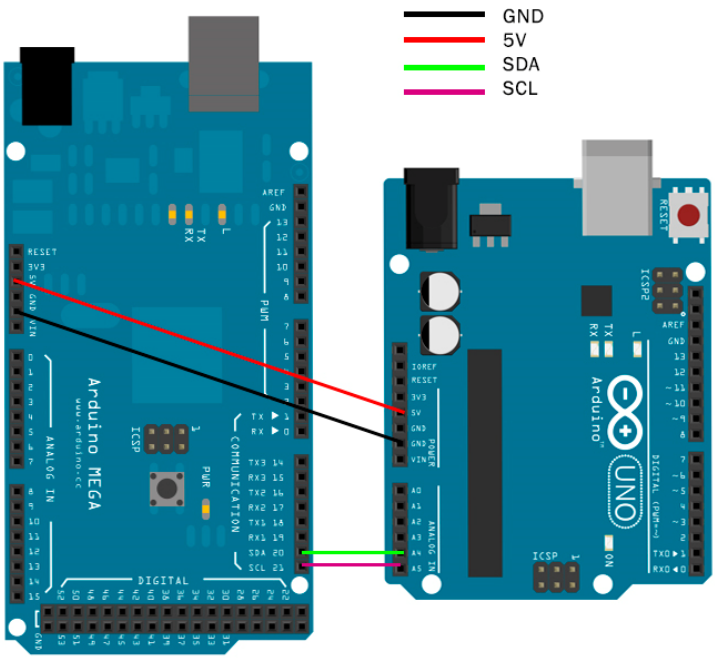


Figura 39. Conexionado comunicación I2C.

Una vez realizado el cableado, se debe implementar toda la comunicación anteriormente descrita en los códigos. Para ello, se debe incluir la librería Wire, *#include<Wire.h>* y definir el papel de cada Arduino dentro de esta comunicación.

En este caso, el Arduino Mega pide cada 100 ms los datos leídos de los ultrasonidos y los bumpers por el Arduino Uno. Por tanto, el Arduino Mega actúa como maestro y el Arduino Uno como esclavo.

Al ser el Arduino Mega el maestro, será este microcontrolador el que inicie la comunicación en el bucle *setup* mediante el comando:

Wire.begin();

Una vez inicializada la comunicación, los dos microcontroladores pueden intercambiar información a través de los siguientes comandos:

- *Wire.beginTransmission(dirección)*. Inicializa el bus.
- *Wire.endTransmission(dirección)*.Finaliza la comunicación y deja libre el bus.
- *Wire.write(dato)*. Escribe datos en la línea SDA.
- *Wire.read(dato)*. Lee datos de la línea SDA.
- *Wire.onRequest(evento petición).* Activación del evento de petición de datos.
- *Wire.onReceive(evento recibir).* Activación del evento de lectura de datos.
- *Wire.requestFrom(dirección, número de bytes).* Petición de datos.

Aunque en los Anexos 9 y 11 se puede ver toda está comunicación descrita en los códigos, para mayor aclaración se ha diseñado un esquema, véase Figura 40, con todo el proceso de comunicación entre los dos Arduinos.

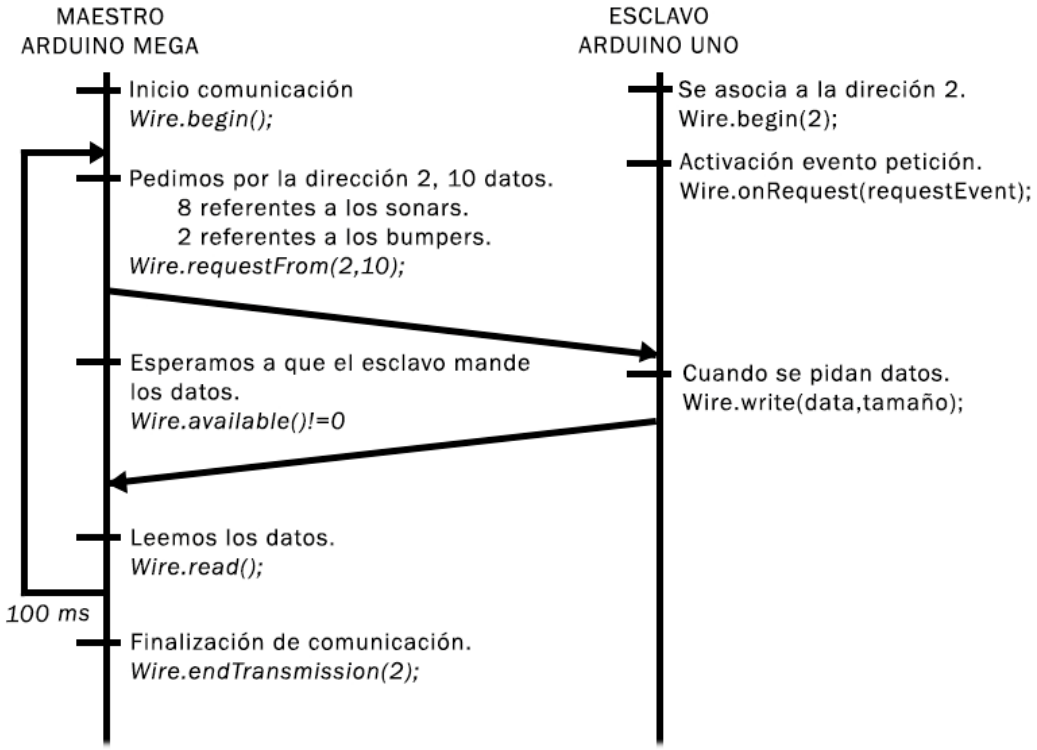


Figura 40. Esquema de comunicación robot Edubot.

5.2. Comunicación Arduino Mega maestro – Ordenador/Arduino Mega esclavo

La comunicación entre Arduino y el ordenador se basa en puertos serie.

Un puerto es un interfaz, físico o virtual, que permite la comunicación entre dos dispositivos. Más en concreto, un puerto serie, envía la información mediante una secuencia de bits mediante dos conectores, RX (recepción) y TX (transmisión). El Arduino Mega, dispone de un conector USB conectado a uno de los cuatro puertos serie de los que dispone, facilitando el proceso de comunicación con el ordenador. Se han recogido en la Tabla 12 todos los pines que pueden ser usados como puerto serie, especificando cuál de todos es usado por el conector USB. Como se puede observar en

la tabla, los pines utilizados por el conector USB son el 0(RX) y el 1(TX), por lo que hay que tener en cuenta que estos dos pines no podrán ser utilizados para otros fines.

	Pin RX	Pin TX	¿USB?
Puerto Serie 1	0	1	Si
Puerto Serie 2	19	18	No
Puerto Serie 3	17	16	No
Puerto Serie 4	15	14	No

Tabla 12. Puertos serie Arduino Mega.

Para realizar la comunicación entre el Arduino Mega maestro y el Arduino Mega esclavo, será necesario conectar:

- El pin RX de un Arduino con el pin TX del otro Arduino.
- El pin TX de un Arduino con el pin RX del otro Arduino.
- El pin GND de un Arduino con el pin GND del otro Arduino, de forma que tengan una tierra común.

Por otro lado, para realizar el proceso de comunicación con el control de alto nivel basta con conectar mediante un cable el microcontrolador con el ordenador, pudiendo de esta manera mandar y recibir información del microcontrolador. Para ello, es necesario un interface, que aunque más adelante se utilice el sistema operativo ROS, por ahora se utilizará el propio software de Arduino o Matlab.

Si se abre el software de Arduino, el lector se encontrará la pantalla que se observa en la Figura 41, en la cual se puede ver la pestaña “Monitor Serie”. Si se picha en esa pestaña, se abrirá una segunda ventana, como la que se observa en la Figura 42. En esta nueva pestaña se pueden observar dos zonas claramente diferenciadas. La zona superior es por donde el usuario le puede mandar información al microcontrolador, y la zona inferior es por donde se visualiza la información que recibe el ordenador por parte del microcontrolador.

Figura 41. Interface Arduino.

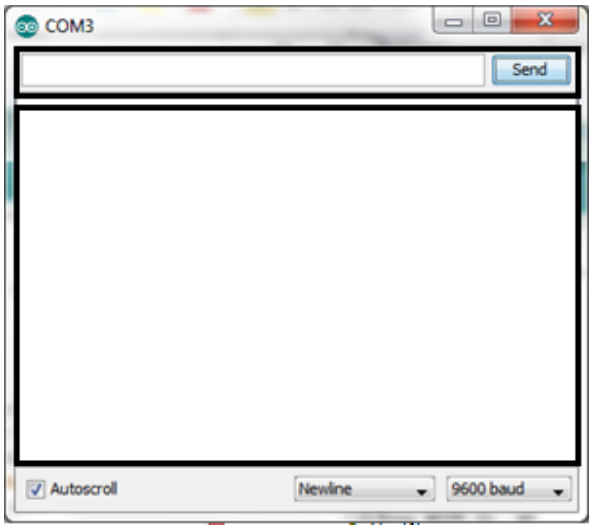


Figura 42. Serial Monitor.

Una vez explicado el método de comunicación a nivel físico y software, queda explicar el código necesario para que esta comunicación sea posible.

En primer lugar, será necesario abrir el puerto serie y fijar la velocidad en baudios para la transmisión de datos en serie. El valor utilizado en este caso es el de 115200 baudios, y esta apertura se realizará en el bucle *void setup()*, mediante el comando:

*Serial.begin(*115200*);*

Abierto el puerto serie, tan sólo queda mandar y recibir datos. Para enviar será necesario imprimir los datos en el puerto serie mediante dos tipos de comandos:

Serial.print(data);

Serial.println(data);

Ambos funcionan de la misma manera, imprimen el dato colocado entre paréntesis por el puerto serie, sin embargo, el segundo comando además de imprimir el dato, imprime un salto de línea.

Para recibir un dato se utilizarán los siguientes comandos:

Serial.available();

Serial.read();

El primer comando devuelve un número entero con el número de caracteres disponibles en el buffer (capacidad máxima de 128 byte), es decir, el número de datos enviados. En el caso de que no se enviará ningún dato, el comando devuelve un 0. En caso contrario, será necesario leer los datos recibido mediante el segundo comando escrito, el cual es capaz de leer un solo byte del puerto serie.

Una vez conseguida la comunicación entre el microcontrolador y el ordenador, hay que tener en cuenta que todo el proceso de control del robot Edubot debe comunicarse con un programa ya realizado en ROS, el cual sigue un protocolo para enviar y recibir datos. Por este motivo, será necesario implantar este protocolo de comunicación en el microcontrolador Arduino Mega maestro, para compatibilizar este control con ROS. Todo este protocolo se explica a continuación.

Tramas de comunicación

Todas las tramas de comunicación empiezan por *"$"* y terminan por *"#"*. Después del símbolo *"$"* se escribirá un código formado por tres caracteres que servirán para identificar la información que se está mandando o recibiendo y tras estos tres caracteres se escribirán dos puntos *":"*. Entre los dos puntos y la almohadilla se escribirán los datos que se desean mandar entre corchetes *"[ ]"*. Por último, tras las almohadilla se deben escribir dos símbolos, uno de salto de línea, *"\n"*, y otro de retorno de carro, *"\r"*. Por ejemplo, una posible trama de comunicación podría ser la siguiente:

$ABC:[dato1,dato2,dato3]#\n\r

Los datos que se deben mandar al ordenador desde el Arduino Mega son los que se citan a continuación:

- Nivel de la batería.
- Estado de los motores.
- Estado de los bumpers.
- Estado de los ultra sonidos.
- Alarmas.

A parte, el Arduino Mega recibirá del ordenador referencias de movimiento para los motores.

En cuanto a los datos referidos al nivel de la batería, se deberá informar al ordenador cada 2000 ms de cuantas baterías dispone el robot, en este caso una, y del valor de la carga de la batería. Los tres comandos que identifican esta trama de comunicación son *VLT* y la trama de comunicación completa sería:

$VLT:[Número de baterías, Valor de la carga de la batería]#\r\n

Respecto al estado de los motores, hay que indicar al ordenador el número de motores que se están controlando y la velocidad angular y el número de pulsos actuales de cada motor. Esta trama de comunicación hay que enviarla cada 100 ms al ordenador mediante los caracteres *MOT*. La trama de comunicación completa sería:

$MOT:[Número de motores, Velocidad motor 1, Número de pulsos motor 1, Velocidad motor 2, Número de pulsos motor 2]#\n\r

El estado de los dos bumpers debe notificarse al ordenador cada 100 ms a través de los caracteres *BMP*, indicando el número de bumpers y el estado de cada uno de ellos, siendo este estado equivalente a 0 si el bumper no ha sido pulsado y en caso contrario, el estado valdrá 1. La trama de comunicación completa sería:

$BMP:[Número de bumpers, estado bumper delantero, estado bumper trasero]#\n\r

El estado de los ocho ultrasonidos, al igual que con los bumpers, debe notificarse cada 100 ms al ordenador. En esta ocasión, los tres caracteres que identifican esta trama son *SON* y en ella deben indicarse el número de ultrasonidos y la distancia leída por cada uno de ellos. La trama de comunicación sería:

$SON:[Número de sonars, dist sonar 1, dist sonar 2, dist sonar 3, dist sonar 4, dist sonar 5, dist sonar 6, dist sonar 7, dist sonar 8]#\n\r

La numeración de los ultrasonidos indicada en la trama sigue la localización ya explicada en el capítulo *3. Diseño Actual - Figura 18*.

Es necesario notificar al ordenador posibles alarmas cuya gestión ya se explicó en el capítulo anterior. Dichas notificaciones deben enviarse todas en una misma trama de

comunicación cada 2000 ms o cada vez que se produzca un cambio. Esta trama de comunicación será identificada mediante los caracteres *AL_* e indicarán si están o no activas las alarmas de batería baja o de batería crítica, siendo estos estados iguales a 1 en caso afirmativo y 0 en caso contrario. La trama de comunicación sería:

$AL_:[Alarma batería baja, Alarma batería crítica]#\n\r

Por último, el ordenador puede enviar una trama de comunicación al Arduino Mega indicándole a qué velocidad debe ir cada motor. Esta trama de comunicación se indicará a través de los comandos *REF* y enviará un número de referencia, 3, para que el microcontrolador sepa que la información recibida se refiere a la velocidad a la que deben ir los dos motores y la velocidad de cada uno de los motores. Para mayor aclaración, se indica la trama de comunicación a continuación:

$REF:[3, velocidad motor derecho, velocidad motor izquierdo]#\r\n

Una vez explicadas todas las tramas de comunicación y todo el proceso de control, se procede a ilustrar, Diagrama de flujo 5, un diagrama de bloques completo de todo el código del proceso de control y comunicación del proyecto.

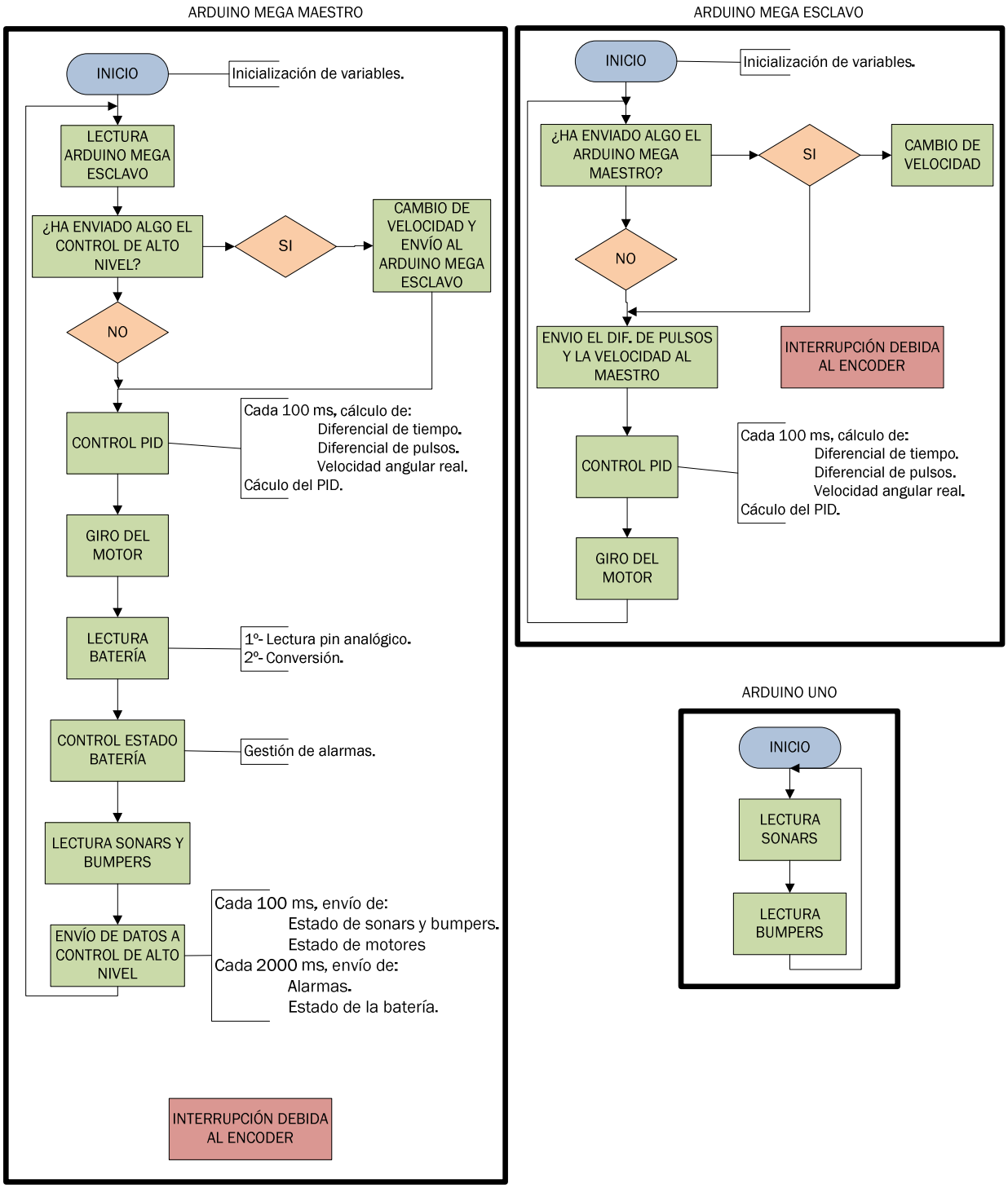


Diagrama de flujo 4. Control total de robot Edubot.

6. PROGRAMAS DE COMPROBACIÓN DEL ROBOT EDUBOT

6.1. Introducción

Parte del objetivo de toda la programación desarrollada en Arduino es su futura comunicación con el control de alto nivel en el sistema operativo ROS, el cual estaba implementado en el proyecto precedente a este. Sin embargo, antes de compatibilizar el código desarrollado en Arduino y el código ya existente en ROS, fue necesario comprobar múltiples características del robot Edubot, verificando el correcto funcionamiento del mismo. Es decir, antes de compatibilizar la comunicación entre el control de alto y bajo nivel, es necesario comprobar que todo lo desarrollado en bajo nivel funciona correctamente. Para ello se usó un programa llamado MATLAB.

MATLAB es una herramienta de software matemático que ofrece un entorno de desarrollo integrado con un lenguaje propio de programación.

A lo largo del presente capítulo se explicaran todos los programas desarrollados en MATLAB para comprobar el correcto funcionamiento del código desarrollado en Arduino para controlar los diferentes dispositivos del robot, entre los cuales se encuentran:

- La comprobación de lectura de todos los bumpers y ultrasonidos.
- La comprobación del movimiento de los motores.
- La correcta lectura de los encoders.
- Representación gráfica de los PIDs.
- Representación gráfica de la localización del robot.

6.2. Desarrollo de los programas

Para que todo lo programado en los tres Arduinos se pueda tratar y visualizar en Matlab, es necesario hacer una inicialización previa, de modo que sea posible la comunicación entre el control de bajo nivel y el de alto nivel.

Para ello, es necesario realizar los siguientes pasos:

1) Borrado de todas las conexiones previas que pueda haber.

```
Delete(instfind({'Port'},{'COMX'}));
```

2) Indicarle al programa cuál es el puerto del Arduino del que llegarán los datos.

```
Puerto_serial = serial('COMX');
```

3) Indicarle la velocidad en baudios para la transmisión de datos en serie.

Puerto_serial.BaudRate = 115200*;*

4) Por último, se debe abrir el puerto serie para poder comenzar con la interacción de datos.

fopen(Puerto_serial);

Una vez realizada toda la inicialización de la comunicación entre Arduino y Matlab, se puede proceder a la ejecución de los diferentes programas que servirán para verificar el correcto funcionamiento de todos los códigos desarrollados en el control de bajo nivel y como simulación del propio sistema operativo ROS.

Terminada la ejecución del programa de comprobación, es muy importante cerrar la comunicación con el puerto serie, ya que de no eliminarla, no se podrá utilizar la misma para otras acciones, por ejemplo, modificar el código ya presente en el microcontrolador.

Para finalizar la comunicación es necesario seguir estos dos simples pasos:

1) Cerrar el puerto serie.

fclose(Puerto_serial);

2) Borrar la conexión con el puerto serie.

Delete(Puerto_serial);

Explicados todos los pasos iniciales y finales más importantes, se procede al desarrollo de los programas.

PID

Una vez desarrollado el programa en Arduino ya explicado en el apartado *Control PID de la velocidad* del capítulo *4. Control del robot Edubot,* la propia interface de Arduino nos indicaba tanto la velocidad angular actual del motor, como el error proporcional e integral (el control derivativo no se ha usado). Sin embargo, visualizando estos datos resulta muy complejo comprender cuál está siendo la evolución de todos estos parámetros a lo largo del tiempo. Por este motivo, resulta conveniente realizar dos gráficas en la cuales se visualicen por un lado la evolución de la velocidad angular y por otro lado los errores calculados por el PID.

Para ello, en primer lugar se deben crear un par de ventanas gráficas mediante el comando:

Nombre de la ventana gráfica = figure('Name','Nombre');

Se han configurado de tal manera que representen hasta 400 muestras, las cuales han parecido un número suficiente para analizar la evolución, tanto en estado transitorio como en estado estacionario, del PID.

Una vez creadas las ventanas gráficas, se han puesto nombre a los ejes y mediante un bucle *while* se han representado las 400 muestras anteriormente mencionadas.

Dentro del bucle, se irán leyendo los 400 valores de la velocidad angular actual, los 400 valores del error proporcional y los 400 valores del error integral del PID, todo ello mediante el comando:

fscanf(puerto_serial, '%g')

El comando *figure* tiene la peculiaridad de que al representar un nuevo valor, todo lo que estuviera anteriormente representado se borrará. Para evitar este borrado indeseado se usará el comando *hold on*, consiguiendo que cada nuevo dato se represente sobre todo lo anterior.

Todos los valores leídos se irán representando mediante el comando *plot* indicando antes en que gráfica deben representarse de la siguiente manera:

Figure(Nombre de la ventana gráfica);

Plot(x,y);

Con el comando *plot*, pasados unos segundos, se da un pequeño retardo de tiempo entre que se lee el valor y se dibuja en la gráfica, dejando de ser una representación en tiempo real. Para evitar esto, se añadirá el comando *drawnow* para que todos los valores representados se dibujen en tiempo real en la gráfica.

Por último comentar que el código explicado puede verse en el *Anexo 12. Representación PID en Matlab* y en el capítulo 7 se analizarán todos los resultados recogidos de estás gráficas, consiguiendo ajustar los parámetros del PID.

Prueba

El correcto funcionamiento del robot Edubot depende de muchos factores, entre ellos se encuentran algunos citados en la siguiente lista:

- Detección de obstáculos por parte de todos los ultrasonidos presentes en el robot.
- Funcionamiento de los dos bumpers.
- Correcto funcionamiento del PID de los motores en todos los sentidos posibles.
- Lectura correcta de los encoders.

Si al oprimirse alguno de los bumpers el control de bajo nivel no lo detecta, al chocarse el robot contra un obstáculo, no cambiará la dirección del movimiento, consiguiendo que el robot no avance. Lo mismo ocurre si alguno de los ultrasonidos no funciona, ya que de ser así, el microcontrolador no sabrá que se aproxima hacía un obstáculo y no podrá evitarlo.

Por otro lado, si los encoders no leen correctamente el número de vueltas dadas, se calculará mal la velocidad angular que lleva la rueda en ese momento, provocando un incorrecto cálculo del PID y por tanto un cálculo incorrecto de la velocidad que se debe introducir al motor para que esta se mantenga constante. Igualmente ocurre un cálculo incorrecto del PID debido a una escritura de código incorrecta o por una asignación de valores a las constantes errónea. Todo ello provoca un funcionamiento anómalo de los motores, produciéndose aceleramientos y frenadas no deseadas.

Por todos estos motivos, se ha considerado de gran importancia comprobar todos estos factores antes de dejar el robot en el suelo en funcionamiento.

Para ello, se ha diseñado un programa tanto en Matlab como en Arduino que verifique todos los factores anteriores antes de poner en funcionamiento el robot Edubot.

Matlab

Se trata de un programa interactivo con el usuario, ya que en el control de alto nivel se ha creado un menú dónde el usuario puede elegir las comprobaciones que desea realizar.

Pero antes de comenzar con todo el programa de comprobación, para poder comunicarse con el microcontrolador Arduino, es necesario inicializar el puerto serie como ya se indicó anteriormente en este capítulo.

Una vez inicializado y abierto el puerto serie, se abre el siguiente menú, donde el usuario puede elegir el tipo de comprobación a realizar:

¿Quiere realizar alguna comprobación antes de empezar a usar el robot?(S/N)

¿Qué quiere comprobar?

- *Encoder derecho (Pulse 1)*
- *Encoder izquierdo (Pulse 2)*
- *Bumpers y Sonars (Pulse 3)*
- *Visualización del PID derecho, hacia delante (Pulse 4)*
- *Visualización del PID derecho, hacia atrás (Pulse 5)*
- *Visualización del PID izquierdo, hacia delante (Puse 6)*
- *Visualización del PID izquierdo, hacia atrás (Pulse 7)*

En función del carácter que introduzca el usuario, se realizarán diferentes acciones, a saber:

- Si a la primera pregunta se introduce una “N” o una “n”, es decir, se indica que no se quiere realizar ninguna comprobación, aparecerá por pantalla la frase *Comenzamos con el funcionamiento del robot* y se sale del programa.
- Si a la primera pregunta se introduce una letra que no es “N”, “n”, “S” o “s”, se indica por pantalla que la tecla introducida es errónea y vuelve a dar la posibilidad de introducir de nuevo el carácter.
- Si a la primera pregunta se introduce una “S” o una “s”, o lo que es lo mismo, se indica el deseo de realizar una o varias comprobaciones, aparece el menú con todas las comprobaciones permitidas. En este punto, se pueden realizar varias acciones:
    - Pulsar el número 1. Se indicará al control de bajo nivel de esta acción, obteniendo como respuesta el movimiento del motor derecho. El usuario debe comprobar como realiza dos vueltas completas, una en un sentido y otra en sentido contrario. Si las vueltas completas no fueran del todo exactas o no cambiara el sentido de giro, indicará al usuario un incorrecto funcionamiento de los motores o de los encoders.
    - Pulsar el número 2. Se indicará al control de bajo nivel de esta acción, obteniendo como respuesta el movimiento del motor izquierdo. El usuario debe comprobar como realiza dos vueltas exactas, una en un sentido y otra en sentido contrario. Si las vueltas completas no fueran del todo exactas o no cambiara el sentido de giro, indicará al usuario un incorrecto funcionamiento de los motores o de los encoders.
    - Pulsar el número 3. Se indicará al control de bajo nivel el deseo de comprobar los bumpers y los ultrasonidos. El usuario deberá acercar la mano a todos y cada uno de los ultrasonidos del robot, indicándole por pantalla si se ha detectado la mano. En caso de que un ultrasonido no detecte la mano o de cómo mano detectada sin haber acercado la mano, significará que ese ultrasonido no funciona correctamente. Igualmente ocurrirá con los bumpers. El usuario debe presionarlos de forma que por pantalla se indicará que dicha presión ha sido detectada. En caso de no detectar la presión realizada o en caso de detectarla sin que el usuario haya presionado el bumper, indicará un incorrecto funcionamiento del bumper.

- Pulsar el número 4. Se indicará al control de bajo nivel el deseo de comprobar el correcto funcionamiento del PID cuando el motor derecho está girando hacia delante. Modificando levemente el programa ya descrito de PID, se representará por pantalla la velocidad angular actual del robot, y los errores proporcional e integral del PID. De esta forma, el usuario podrá ver la evolución del movimiento del motor a lo largo del tiempo, deducir si este funcionamiento es correcto o incorrecto y si es necesario cambiar el valor de algún parámetro.
- Pulsar el número 5. Se indicará al control de bajo nivel el deseo de comprobar el correcto funcionamiento del PID cuando el motor derecho está girando hacia atrás. Las acciones seguidas son las mismas que en al pulsar el número 4, por lo que no se volverá a explicar para evitar ser repetitivo.
- Pulsar el número 6. Se indicará al control de bajo nivel el deseo de comprobar el correcto funcionamiento del PID cuando el motor izquierdo está girando hacia delante. Las acciones seguidas son las mismas que en al pulsar el número 4 o 5, por lo que no se volverá a explicar para evitar ser repetitivo.
- Pulsar el número 7. Se indicará al control de bajo nivel el deseo de comprobar el correcto funcionamiento del PID cuando el motor izquierdo está girando hacia atrás. Las acciones seguidas son las mismas que en al pulsar el número 4, 5 o 6, por lo que no se volverá a explicar para evitar ser repetitivo.
- Pulsar un número que no corresponde con ninguno de los anteriores. En este caso, el usuario habrá introducido un número erróneo, indicándoselo el programa y dándole de nuevo la opción de volver a introducir un número válido.

Una vez finalizado el programa, se procederá a cerrar la conexión con el puerto serie y a eliminar todas las variables anteriormente creadas.

Todo el código descrito podrá leerse con mayor detenimiento en el anexo adjunto *Anexo 13. Programa de comprobación. Matlab.*

Arduino

Por otro lado está el control de bajo nivel, sin el cual, el control de alto nivel realizado en Matlab no tendría ningún sentido.

Mediante la comunicación Arduino-Matlab, anteriormente explicada, el microcontrolador se encuentra a la espera de recibir uno de los anteriores números, para saber que ordenar a los diferentes dispositivos del robot. Comentar que todo el código utilizado ha sido reciclado de los códigos explicados a los largo del capítulo 4, por lo que en este capítulo no se explicará de nuevo todo el código en profundidad.

Definidos e inicializados todos los parámetros y variables, comienza la ejecución del programa en el bucle *loop*.

Por el puerto serie puede recibir 7 tipos de caracteres diferentes, y en función del carácter recibido, el microcontrolador realizará 7 acciones diferentes, a saber:

- El carácter recibido es un 1. En ella se llama a la función *Comprobacion_motor_der()*, en la cual primero girará hacía atrás la rueda derecha hasta que dé una vuelta completa, 490 pulsos. Llegados a este punto, el motor parará para a continuación comenzar a girar en sentido contrario hasta dar otra vuelta completa. Una vez se han dado las dos vueltas completas, el motor se parará a la espera de una nueva orden.
- El carácter recibido es un 2. En ella se llama a la función *Comprobacion_motor_izq()*, en la cual primero girará hacía atrás la rueda izquierda hasta que dé una vuelta completa, 490 pulsos. Llegados a este punto, el motor parará para a continuación comenzar a girar en sentido contrario hasta dar otra vuelta completa. Una vez se han dado las dos vueltas completas, el motor se parará a la espera de una nueva orden.
- El carácter recibido es un 3. En este caso, mediante comunicación I2C se pedirán las lecturas obtenidas por parte de los bumpers y los ultrasonidos, indicando en caso de ser pulsado o de detección de obstáculo este hecho.
- Si el carácter recibido es un 4 o un 5, se moverá el motor derecho hacia delante o hacía atrás, respectivamente, mandando al control de alto nivel los valores de la velocidad angular y los errores proporcional e integral.
- Si el carácter recibido es un 6 o 7, se moverá el motor izquierdo hacia delante o hacía atrás, respectivamente, mandando al control de alto nivel los valores de la velocidad angular y los errores proporcional e integral.

Por último, comentar que todo este código se encuentra adjunto al final de la memoria en el *Anexo 14. Programa de comprobación. Arduino Mega.* El código referente al Arduino Uno, lectura de bumpers y ultrasonidos, no se ha adjuntado, puesto que es el mismo que el utilizado anteriormente y puesto que en ningún caso los dos motores se mueven a la vez, se ha realizado todo el conexionado y programación en un solo Arduino Mega.

Odometría

Parte del objetivo del proyecto es el cálculo de la odometría del robot. Dicho cálculo debe realizarse en el control de alto nivel, sin embargo, en primer lugar se programó en el control de bajo nivel. Este código puede consultarse en el *Anexo 15. Odometría. Arduino Mega.*

En este código podrá observare como prácticamente todo el código es igual al definitivo ya visto anteriormente en el capítulo 4, salvo por una nueva función, *Odometria()*, en la cual se han implantado las ecuaciones odométricas ya explicadas en el punto *4.7 Control odométrico* de este informe y porque en esta ocasión las ruedas no se mueven, es decir, para simular el movimiento del robot el usuario deberá mover las ruedas a placer. Al igual que en el caso anterior, como no hay que realizar el control PID de los dos motores a la vez, se ha realizado todo el cableado y programación en un solo Arduino Mega.

En la interface de Arduino, se puede ir mostrando en qué posición del eje X y del eje Y se encuentra el robot de manera numérica, sin embargo, de esta forma el usuario no puede saber realmente si la odometría se está calculando correctamente. Por este motivo, se decidió desarrollar un programa en Matlab que representará en un plano XY el movimiento del robot. De esta forma, se podrá simular una vista aérea de la trayectoria del robot y el usuario podrá deducir fácilmente si la odometría es o no correcta.

Como siempre, para implementar este programa en Matlab es necesario inicializar el puerto serie y abrirle, para posteriormente ejecutar el programa.

Para ello, será necesario crear una ventana gráfica como ya se explicó anteriormente, sin embargo, en esta ocasión no se recurrirá al comando *plot* para dibujar la gráfica, si no al comando *line*. El motivo de este cambio es que el comando *plot*, pasados unos segundos, comienza a dibujar los puntos de la gráfica con cierto retraso, cosa que no pasa con el comando *line*, mucha más eficiente para este caso.

Una vez creada la ventana gráfica, se procede a la lectura y representación de los datos mediante un bucle *while true*, para representar los datos de manera ininterrumpida.

Dentro de este bucle se irán leyendo los puntos X e Y ya calculados por el control de bajo nivel, para después guardarles en vectores y por último, ir añadiendo estos puntos a la gráfica mediante el comando *set*.

Todo este programa se puede observar con mayor detenimiento en el *Anexo 16. Odometría. Matlab*., adjuntado al final de la memoria.

Simulación ROS

Por último, una vez verificado el correcto control de los dispositivos, antes de comunicar el control de bajo nivel con el sistema operativo ROS, se simuló el mismo en Matlab.

En este punto, todo el control de bajo nivel está ya completo. Es decir, ya se ha programado todo lo explicado a lo largo de la memoria, a saber:

- Lectura de bumpers y ultrasonidos en el Arduino Uno.
- Comunicación entre el Arduino Mega maestro con el Arduino Mega esclavo y el Arduino Uno.
- Control de motores en cada Arduino Mega.
    - Control de encoders.
    - Control de puentes H.
    - Control del PID.
- Control del estado de la batería en el Arduino Mega.
- Gestión de alarmas.
- Comunicación entre el Arduino Mega y el control de alto nivel.
- Implementación de las tramas de comunicación.

Aunque ya se han comprobado gran parte de los puntos citados, faltan por comprobar la implementación de las tramas de comunicación y la gestión de alarmas. Por otro lado, también se pretende probar la programación de la odometría en alto nivel y la simulación de evitar obstáculos. Todo esto se verá en el siguiente programa de Matlab que pretende simular la interacción del sistema de control de bajo nivel con el sistema operativo ROS.

Sin embargo, aunque en el momento en el que se vaya comunicar el Robot Edubot con ROS se van a implantar las tramas de comunicación completas, véase *Anexo 11. Código Arduino Mega Maestro*, en este caso se va a prescindir de las letras que contienen dichas tramas. Por ejemplo, en el caso de la trama de la trama de comunicación que informa sobre el estado de la batería:

$VLT:[Número de baterías, Valor de la carga de la batería]#\r\n

Se va a enviar desde el Arduino Mega sólo el número de baterías y el valor de la carga de la batería, sin el "*$VLT:[*" y el "*]#\r\n*".

Esta simplificación es debida a que con este programa lo que se quiere comprobar de las tramas de comunicación es que llegan todos los datos que deben llegar, en el orden y tiempo adecuados.

La simulación de evitar obstáculos se debe realizar en el control de alto nivel, dónde, a partir de los datos obtenidos de los ultrasonidos y sensores, manda al control de bajo nivel la velocidad más adecuada para cada rueda. Sin embargo, en este programa se programará un código en bajo nivel de forma que sea capaz de evitar algún obstáculo, con el fin de probarlo. Por este motivo, en este código utilizado para la simulación, tampoco se ha tenido en cuenta la comunicación que va del ordenador al Arduino Mega, es decir, no se ha tenido en cuenta la trama de comunicación:

$REF:[3, velocidad motor derecho, velocidad motor izquierdo]#\r\n

A la hora de programar el sorteo de obstáculos se ha tenido solamente en cuenta los obstáculos que se pudiera encontrar el robot de frente, obviando los posibles obstáculos traseros. Esto se ha hecho debido a que sólo se trata de una simulación. En el control de alto nivel estará programado de forma que el robot pueda evitar los obstáculos que se encuentren en cualquier dirección. En el cuadro que se adjunta a continuación, véase Tabla 13, se ha querido resumir brevemente la respuesta de los motores ante diferentes obstáculos.

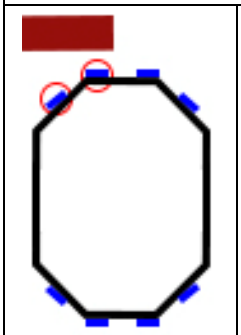
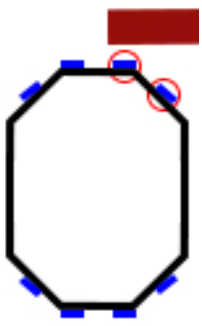
Obstáculos	Velocidad rueda derecha	Velocidad rueda izquierda
	↓	↑
	↑	↓

Tabla 13. Comportamiento ante obstáculos.

Para poder simular correctamente el envío de estas tramas de comunicación en los tiempos requeridos, por limitaciones de Matlab, siempre se enviarán todos los datos cada 100 ms, indicándole a través de una variable *imprimir*, cuando pasan 2000 ms o bien cuando hay un cambio en las alarmas. De esta forma, se imprimirán por pantalla el estado de las alarmas y baterías en los momentos adecuados.

Con toda esta información, queda por comentar antes de comenzar con el desarrollo del programa en Matlab que el código de Arduino para realizar esta simulación se encuentra adjunto en el *Anexo 17. Simulación final. Arduino Maestro.*

Matlab

Como siempre, se comienza con la inicialización y apertura del puerto serie para posteriormente continuar con el programa en sí.

En él, dentro de un bucle *while true*, para que funcione de manera indefinida, se irán leyendo todas las tramas de comunicación enviadas por el Arduino Mega en el siguiente orden:

1. Orden de impresión. Si está variable toma el valor 1, significa que se deben imprimir las alarmas y las baterías, en caso de valer 0, no se imprimirán.
2. Estado de las baterías:
   2.1. Número de baterías, en este caso 1.
   2.2. Voltaje de la batería, en voltios. Por ejemplo 12.6.
3. Gestión de alarmas:
   3.1. Alarma baja. Recordar que se daba está alarma si el voltaje era inferior a 11.3 V.
   3.2. Alarma crítica. Recordar que se daba está alarma si el voltaje era inferior a 10.8 V.
4. Estado de los ultrasonidos:
   4.1. Número de ultrasonidos, en este caso 8.
   4.2. Valor obtenido por cada sensor.
5. Estado de los bumpers:
   5.1. Número de bumpers, en este caso 2.
   5.2. Valor obtenido por cada bumper. Recordar que este valor podía tener un valor 1 si el bumper pulsado y 0 en caso contrario.
6. Estado de los motores:
   6.1. Número de motores, en este caso 2.
   6.2. Velocidad angular del motor derecho.
   6.3. Diferencial de pulsos del encoder derecho.
   6.4. Velocidad angular del motor izquierdo.
   6.5. Diferencial de pulsos del encoder izquierdo.

Una vez leída toda la información necesaria para el control de alto nivel, se procede a la impresión por pantalla de los datos en el siguiente orden:

1. Cada 2000 ms o en momento de cambio, esto se indica en Matlab mediante la orden de impresión:
   1.1. Alarma de batería baja modificada. *Estado de la alarma.*
   1.2. Alarma de batería crítica modificada. *Estado de la alarma.*
2. El estado de la batería, cada 2000 ms.
3. La lectura dada por cada ultrasonido.
4. La lectura de cada bumper.
5. La velocidad angular del motor derecho.
6. La velocidad angular del motor izquierdo.

Posteriormente, se llama a la función *Odometría* para que calcule la posición en la que se encuentra el robot. El código que forma esta función es muy similar al ya programado en Arduino, tal y como el lector puede observar si consulta el *Anexo 18. Simulación final. Matlab.*

Calculada la posición, el programa procede a imprimir por pantalla la gráfica XY, ya explicada en el punto anterior, donde se puede observar en tiempo real la trayectoria tomada por el robot.

7. RESULTADOS EXPERIMENTALES

7.1. Introducción

A lo largo de los capítulos anteriores, se han ido describiendo y desarrollando los diferentes códigos realizados. Se ha hablado de códigos escritos tanto en Arduino como en Matlab. De hecho, se han desarrollado códigos en Matlab con el fin de comprobar el funcionamiento de las diferentes partes del robot y se han ido escribiendo los códigos de Arduino de forma secuencial y aislada para cada uno de los dispositivos del robot Edubot, de forma que no se continuará avanzando hasta no haber comprobado el correcto funcionamiento.

Sin embargo, en ningún momento se han descrito los resultados obtenidos al poner a prueba todos y cada uno de los códigos. Por este motivo, se ha realizado el presente capítulo para abordar todos estos resultados.

7.2. Ultrasonidos y bumpers

En primer lugar se realizó el programa referente al control de ultrasonidos y bumpers, el cual se puede consultar en el *Anexo 3. Recepción de sonars y bumpers*.

A través de este código se deben comprobar dos puntos. El primero y más obvio es el correcto funcionamiento del código y en segundo lugar, se debe comprobar el correcto funcionamiento de los sensores, es decir, que los ultrasonidos lean la distancia correcta y que la entrada de cada bumper se modifique al pulsarlo.

Tras ejecutarlo e ir acercando la mano a todos y cada uno de los ultrasonidos y presionar los bumpers, se pudo comprobar el correcto funcionamiento del código y de los bumpers.

Para comprobar que los ultrasonidos leían las distancias correctamente, se usó una regla en la cual había acoplado un pequeño cartón deslizante, para que el sensor lo detectara como obstáculo. Colocando la regla sobre la superficie superior del robot y moviendo el cartón a lo largo de la misma, pudiendo saber de esta manera la distancia a la que está el obstáculo, se comprobó una a una las lecturas dadas por cada ultrasonido. En la tabla 14 adjunta, se pueden observar los datos obtenidos de esta prueba. Recordar que la numeración de los sensores sigue la ya vista en la Figura 18, de capítulo 3.

En la tabla se puede observar como los valores leídos por los ultrasonidos han sido siempre los reales salvo en los casos en los que el obstáculo se encuentra muy cercano. A distancias menores de 5 cm no se garantiza el buen funcionamiento de los sensores.

		Sensores							
		1	2	3	4	5	6	7	8
Distancias [cm]	25	✓	✓	✓	✓	✓	✓	✓	✓
	20	✓	✓	✓	✓	✓	✓	✓	✓
	10	✓	✓	✓	✓	✓	✓	✓	✓
	5	✓	✓	✓	✓	✓	✓	✓	✓
	0	×	×	×	×	×	×	×	×

Tabla 14. Verificación de distancias ultrasonidos.

7.3. Motores

Tras la realización del código de los ultrasonidos y bumpers se desarrolló el código para controlar los motores, que como ya se explicó anteriormente, se siguió el siguiente orden de desarrollo:

- Lectura de codificadores incrementales. *Anexo 5. Encoders – Contador de pulsos.*
- Control de puentes H. *Anexo 7. Control de puentes H.*
- Control PID de la velocidad. *Anexo 8. Control de la velocidad de cada motor*.

Lectura de codificadores incrementales

Antes de probar el correcto funcionamiento de los codificadores incrementales, es necesario comprobar que el código introducido en el microcontrolador es correcto. Para ello, se giró manualmente la rueda hacia delante y hacia atrás. Al realizar esta acción, el contador de pulsos comenzó a sumar cuando se giraba la rueda hacia delante y a restar cuando se giraba la rueda hacia atrás. De esta forma, se pudo comprobar el correcto funcionamiento del código.

Realizada esta verificación, se pasó a probar la lectura de los codificadores incrementales, obteniendo, al realizar ambas vueltas en varias ocasiones, una lectura

final de los encoder de 490±10 pulsos por vuelta. Téngase en cuenta que esta desviación es debida a que las vueltas que se están haciendo manualmente sobre la rueda son aproximadas, es decir, es muy difícil dar una vuelta exacta a la rueda.

Control de los puentes H

Para poder mover los motores correctamente es muy importante el correcto funcionamiento de los puentes H. Existen diversas combinaciones con las que configurar el puente H de cada motor, sin embargo, muy pocas de esas combinaciones sirven para mover adecuadamente el robot. De hecho, existen algunas combinaciones que nunca deben realizarse, como cerrar a la vez los interruptores de la misma rama, ya que de hacerlo, se produciría un cortocircuito en la fuente de tensión de entrada. Por este motivo, antes de continuar con el control de los motores es muy importante verificar que se han programado correctamente los puentes H.

Para ello, se realizó la siguiente prueba con cada motor. Partiendo con los motores frenados, se comenzó a aumentar los valores PWM, valores positivos, viendo como el motor giraba en sentido de avance. Comprobado que en este sentido funcionaba, se comenzó a disminuir el valor introducido por el pin PWM, disminuyendo con él la velocidad de la rueda hasta llegar al valor 0. En este punto, el motor está parado, verificando así el correcto frenado del motor. Por último, se continuó disminuyendo aún más los valores del PWM, tomando valores negativos y por tanto comenzando a girar en sentido contrario, es decir, en sentido de retroceso.

Control PID de la velocidad

El siguiente paso una vez verificado el correcto control de los puentes H y la lectura de los codificadores incrementales es controlar la velocidad de cada motor por separado. Para ello, es necesario obtener los parámetros PID más adecuados para el robot Edubot.

Con ayuda del código introducido en el microcontrolador y de las gráficas obtenidas a través de Matlab, se pueden conocer las diferentes reacciones de los motores al variar los parámetros PID y de esta forma, saber con cuales se consigue un control de la velocidad óptimo.

En primer lugar, se realizó el estudio sobre el motor derecho girando hacia delante, para una vez encontrados los parámetros adecuados, probar los mismos sobre el mismo motor en sentido inverso y en el motor izquierdo en ambos sentidos.

Se partió de una constante proporcional de 80, con la que se tomaron tres valores diferentes de constante integral, manteniendo la constante derivativa nula. Los resultados obtenidos se pueden observar en la Tabla 15 adjunta.

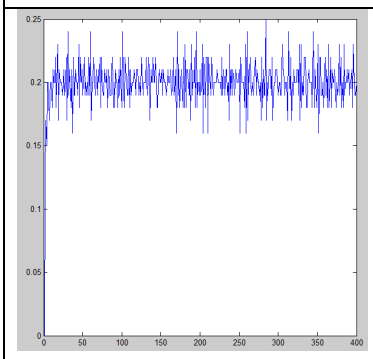
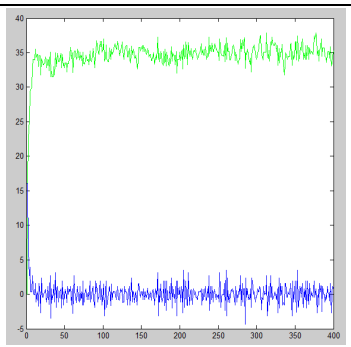
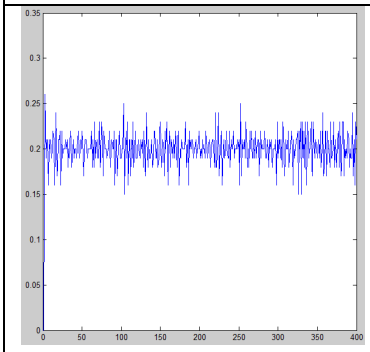
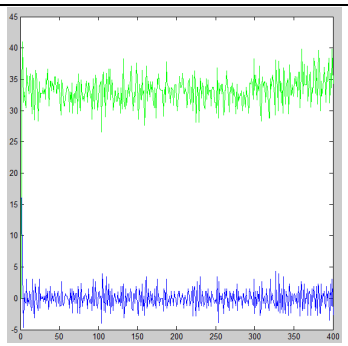
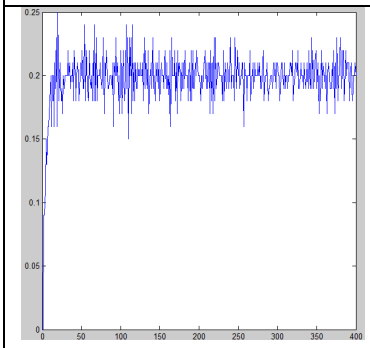
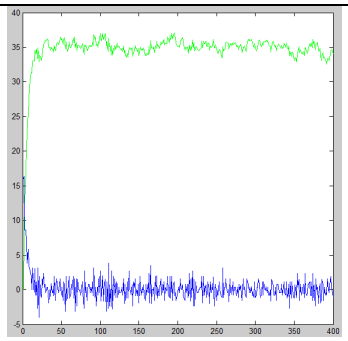
Velocidad actual	Errores P e I	PID	Reacciones
		P = 80 I = 0.01 D = 0	≈ Oscilaciones en la velocidad y errores PI ≈ tiempo en alcanzar estabilidad
		P = 80 I = 0.02 D = 0	↑ Oscilaciones en la velocidad y errores PI ↓ tiempo en alcanzar estabilidad
		P = 80 I = 0.005 D = 0	≈ Oscilaciones en la velocidad y errores PI ↑ tiempo en alcanzar estabilidad

Tabla 15. PID. P=80.

La gráfica de la velocidad angular tomada puede ser de gran utilidad para ver cómo evoluciona la velocidad y si se mantiene constante en el tiempo, sin embargo, es en la curva del error proporcional donde el lector podrá ver cuánto se desvía la velocidad del motor en cada momento de tiempo respecto a la velocidad indicada.

En la tabla 14 se puede observar como partiendo de un valor intermedio para la constante I, al aumentar dicho valor, las oscilaciones en las gráficas se acentúan y el

tiempo de respuesta disminuye, ocurriendo todo lo contrario al disminuir la constante integral, aunque las oscilaciones tampoco disminuyen demasiado en este último caso. En conjunto, el error entre el valor real y el medido se podría considerar significativo, pero disminuyendo las oscilaciones podría eliminarse.

Puesto que con una constante proporcional equivalente a 80 no se produce sobre-amortiguación, se procedió a aumentar la misma hasta un valor de 100, obteniendo los resultados que se observan en la Tabla 16.

En la misma, se puede observar como al aumentar la constante proporcional la velocidad se ha estabilizado un poco más, disminuyéndose ligeramente las oscilaciones y por tanto, el error respecto al valor de referencia. Comentar que se obtuvieron las mismas variaciones en el aumento y disminución de las oscilaciones y del tiempo de estabilización que en el caso anterior.

Tras realizar estos estudios, se ha llegado a la conclusión de que la constante integral más adecuada es la que toma un valor de 0.01, puesto que al aumentar dicha constante las oscilaciones aumentan notablemente y al disminuir dichas variaciones no son muy significativas, sin embargo, la respuesta es mucho más lenta para este último caso.

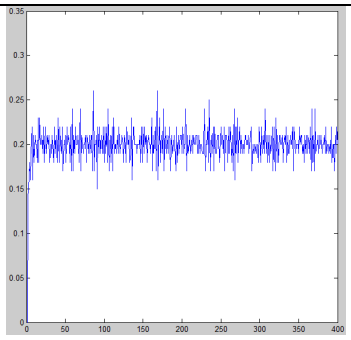
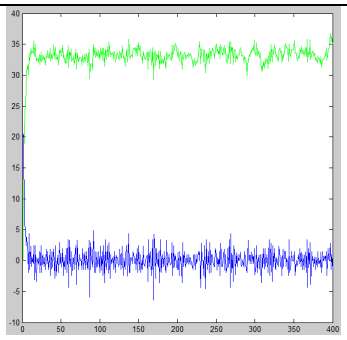
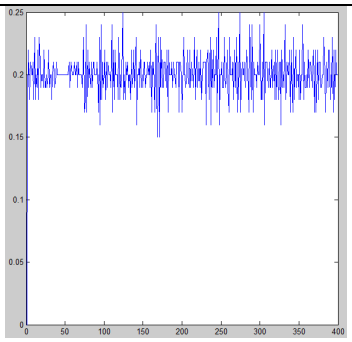
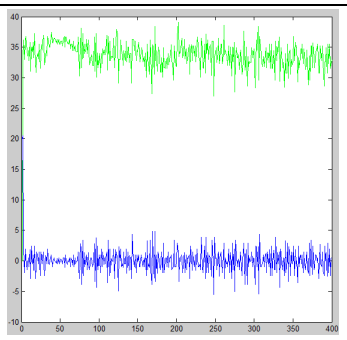
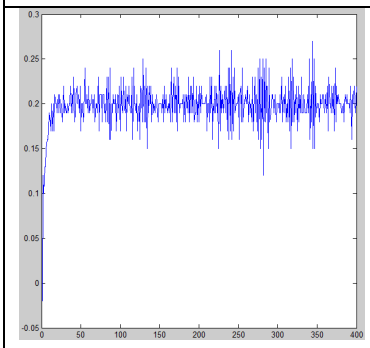
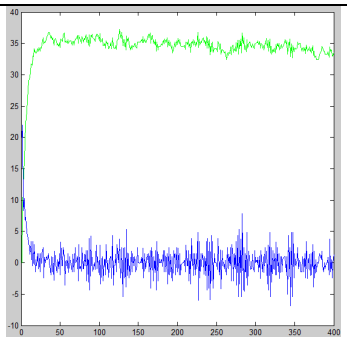
Velocidad actual	Errores P e I	PID	Reacciones
		P = 100 I = 0.01 D = 0	↓ Oscilaciones en la velocidad y errores PI ≈ tiempo en alcanzar estabilidad
		P = 100 I = 0.02 D = 0	↑ Oscilaciones en la velocidad y errores PI ↓ tiempo en alcanzar estabilidad
		P = 100 I = 0.005 D = 0	↓ Oscilaciones en la velocidad y errores PI ↑ tiempo en alcanzar estabilidad

Tabla 16. PID. P=100.

Por último, a la vista de los resultados anteriormente obtenidos, se procedió a aumentar la constante proporcional hasta un valor de 120 y utilizando una constante integral de 0.01, se obtuvieron los resultados que se observan en la Tabla 17. Se puede ver, sobre todo en la gráfica que representa el error proporcional, como el error de la velocidad que toma el robot respecto al valor de referencia es significativamente más pequeña que en los casos anteriores, al igual que las oscilaciones. Si se observa detenidamente la curva de la velocidad medida, el lector puede darse cuenta de que ya se da una pequeña sobre-oscilación, admisible, pero advierte de no aumentar más la constante proporcional.

Por este motivo, los siguientes parámetros se supusieron los mejores para el control de la velocidad del robot Edubot:

- P = 120.
- I = 0.01.
- D = 0.

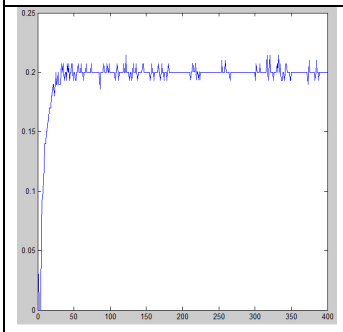
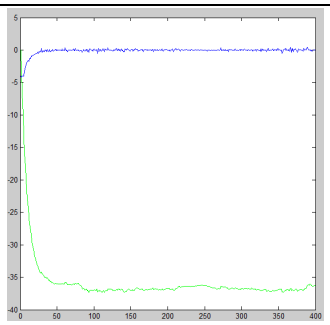
Velocidad actual	Errores P e I	PID	Reacciones
		P = 120 I = 0.01 D = 0	↓↓ Oscilaciones en la velocidad y errores PI ≈ tiempo en alcanzar estabilidad

Tabla 17. PID. P=120, I=0.01.

Para finalizar con este apartado, se adjuntan un par de tablas, véase Tabla 18 y Tabla 19, en la que se puede ver la respuesta de cada motor, derecho e izquierdo respectivamente, moviéndose en ambos sentidos con estos parámetros.

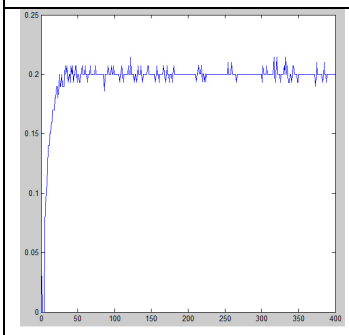
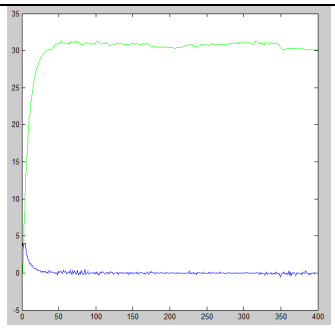
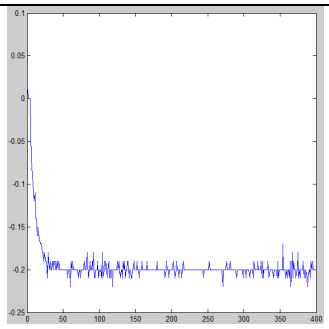
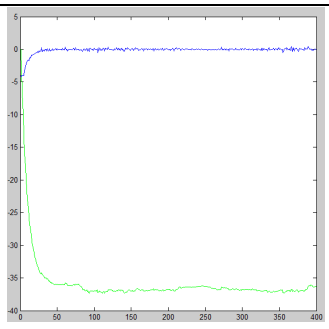
Velocidad actual	Errores P e I	Sentido
		Avance
		Retroceso

Tabla 18. PID Motor derecho.

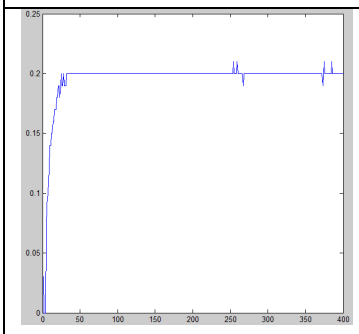
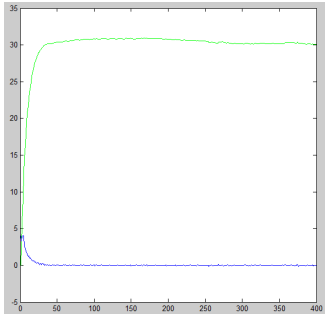
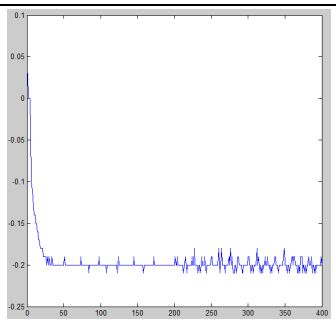
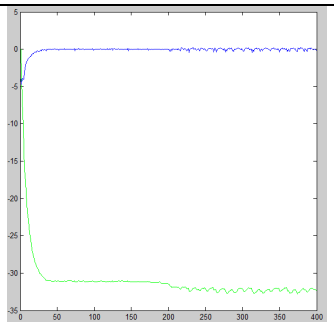
Velocidad actual	Errores P e I	Sentido
		Avance
		Retroceso

Tabla 19.PID. Motor izquierdo.

Se puede observar como los motores responden muy bien ante estos parámetros PID cuando giran en ambos sentidos, dando por concluido este apartado de manera satisfactoria.

7.4. Comprobación Robot Edubot

Como ya se desarrolló en el capítulo anterior, resulto necesaria la realización de una serie de programas para saber el si el robot estaba preparado para funcionar. Estos programas son:

- PID, programa para representar gráficamente los resultados obtenidos del control PID.
- Prueba, un programa para ejecutar antes de poner en el suelo al robot Edubot con el fin de comprobar el correcto funcionamiento de los sensores.
- Odometría, programa con el cual saber si las ecuaciones odométricas se habían desarrollado correctamente, de forma que de manera gráfica, el control de alto nivel nos represente la localización del robot.

- Simulación ROS, programa útil para probar el funcionamiento casi final del robot Edubot.

Puesto que los resultados obtenidos del programa PID ya se han estudiado en las dos últimas secciones de este capítulo, a continuación se van a visualizar los resultados conseguidos a través de los restantes programas.

Prueba

Para la realización de este programa fueron necesarios dos códigos, uno en bajo nivel y otro en alto nivel, los cuales pueden observarse en los Anexos 13 y 14 adjuntos al finalizar esta memoria.

El usuario va a interactuar a través de la interface de Matlab para probar el correcto funcionamiento de todos los sensores.

Nada más ejecutar el programa, el mismo nos preguntará si deseamos realizar algún tipo de comprobación. Si se introduce por teclado "N" saldremos del programa, sin embargo, si se introduce "S", se abrirá un menú donde el usuario podrá elegir entre múltiples comprobaciones, las cuales ya fueron explicadas en el capítulo anterior.

Comentar que el programa funciona correctamente, obteniendo los siguientes resultados en función de los números introducidos por teclado:

- Al introducir un "1" la rueda derecha comenzó a girar, dando primero una vuelta en sentido de avance, para después detenerse y dar otra vuelta completa en sentido de retroceso.
- Al introducir un "2" la rueda derecha comenzó a girar, realizando la misma acción que en el caso anterior.
- Introduciendo un "3", el programa indicó que acercará la mano a cada sonar y que pulsará los bumpers. Según iba acercando la mano a cada ultrasonido, el programa indicaba que había sido detectado. Igualmente ocurría al pulsar los bumpers.
- Introduciendo un "4" o un "5", el motor derecho comenzó a girar en sentido de avance y de retroceso, respectivamente, mostrando por pantalla las gráficas PID ya vistas en apartados anteriores del presente capítulo.
- Al introducir un "6" o un "7", el motor izquierdo comenzó a girar en sentido de avance y de retroceso, respectivamente, ocurriendo lo mismo que en el caso anterior.
- Por último, al introducir cualquier otro carácter de los no indicados, el programa informaba del error cometido, dando la posibilidad de volver a elegir.

Para finalizar con este apartado, se ha querido mostrar el programa ejecutándose en la interface de Matlab, véase Figura 43.

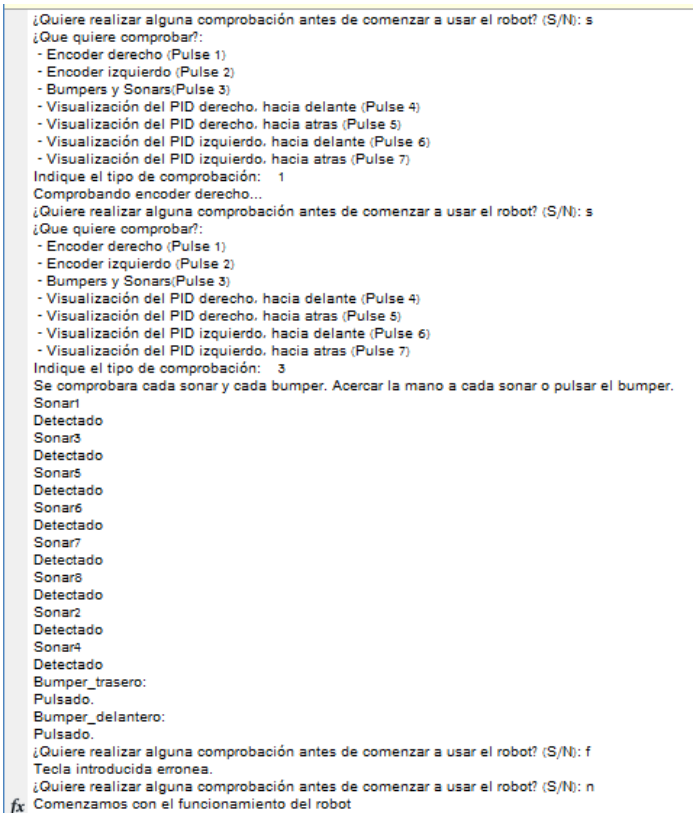


Figura 43. Programa Prueba.

Odometría

Para saber si todo el desarrollo matemático de la localización odométrica es correcto, se ha considerado que no basta con ver numéricamente los valores de las posiciones X e Y y el ángulo girado. Para conocer las coordenadas exactas en un momento determinado estos valores pueden servir, sin embargo, si se quiere conocer la trayectoria seguida por el robot, el análisis de todos los datos obtenidos puede resultar bastante arduo de realizar, aconsejando la representación gráfica de los mismos.

El programa funciona a través de dos códigos, uno en Matlab y otro en Arduino, véase Anexos 15 y 16. Mientras el usuario mueve las ruedas manualmente, en la interface de Matlab se puede ir observando la trayectoria seguida con dichos movimientos, tal y como se observa en la Figura 44.

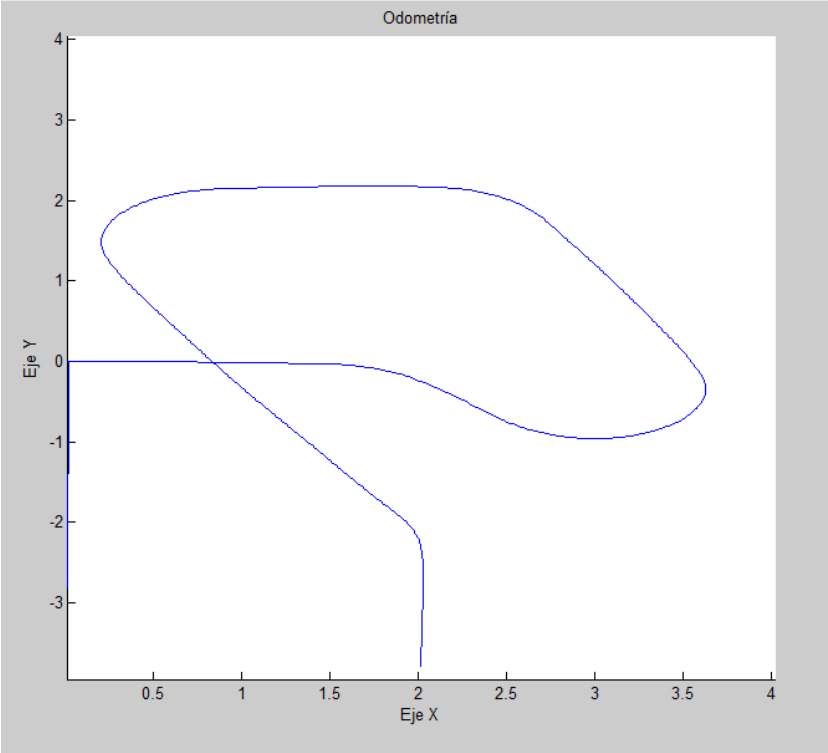


Figura 44. Localización odométrica.

Por último, comentar que la trayectoria representada puede no ser del todo exacta ya que no se ha realizado la calibración odométrica explicada en el punto *4.7 Localización odométrica*.

Simulación ROS

Por último, se va a mostrar el resultado del control de todo el robot Edubot, incluyendo además del control de motores, ultrasonidos y bumpers y odométrico tantas veces desarrollado, el estado de las baterías y la gestión de alarmas.

En este caso, el robot ya anda por el suelo libremente, habiendo introducido en su microcontrolador un pequeño control de obstáculos con el fin de ponerlo a prueba.

Aunque para está simulación las tramas de comunicación no están completas, los tiempos de ejecución son correctos y el robot es capaz de evitar obstáculos mientras por la interface de Matlab se va representando la trayectoria recorrida y los valores oportunos en los intervalos de tiempo estipulados.

En la Figura 45 se muestra la interface de Matlab en un momento determinado en una de las múltiples pruebas realizadas con este programa.

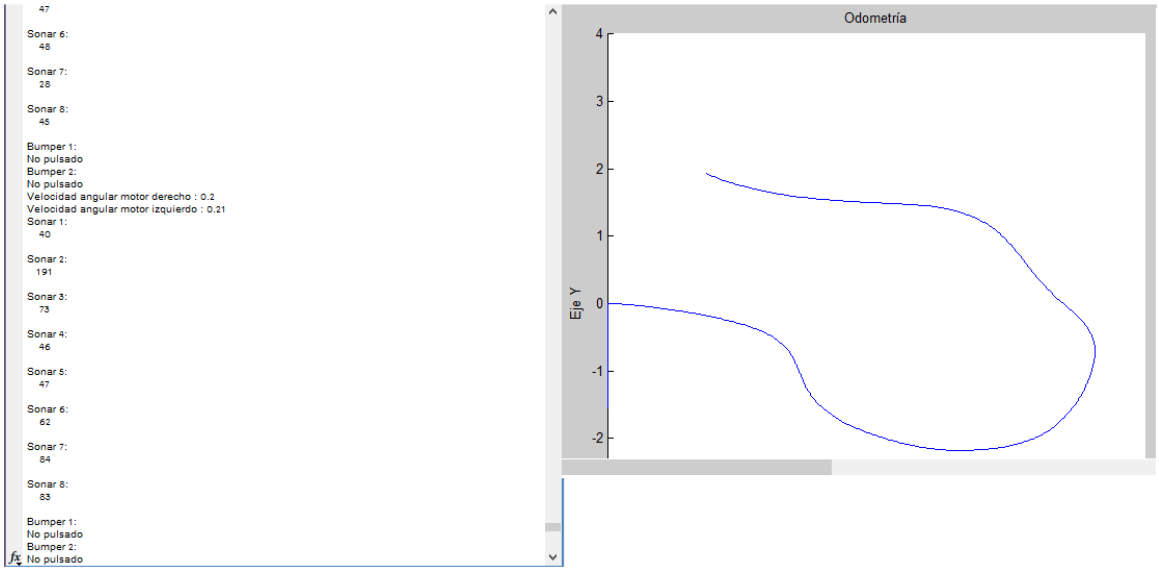


Figura 45. Simulación ROS.

8. ESTUDIO ECONÓMICO

8.1. Introducción

A lo largo de este capítulo se va a realizar el estudio del coste económico que supone la realización del proyecto desarrollado.

Hasta este momento, se ha estudiado todos los aspectos teóricos y técnicos necesarios para la ejecución del robot Edubot, sin embargo, no se ha analizado su viabilidad económica, aspecto también de gran importancia. Por este motivo, se ha escrito un capítulo sólo para este estudio.

En los siguientes apartados se analizarán por separado los costes directos e indirectos para finalmente sumarlos y obtener los costes totales del proyecto.

8.2. Recursos empleados

A lo largo de la realización del proyecto se utilizaron una serie de recursos hardware y software. Cabe destacar que aunque algunos de estos recursos no son gratuitos, su uso no se limita al desarrollo del proyecto, es decir, dicho recursos han sido y seguirán utilizándose durante un periodo indeterminado de tiempo para otros ámbitos tanto académicos como personales, por tanto, es necesario tener presente la amortización del material durante el tiempo que ha sido utilizado en el proyecto, para sólo añadir dicho coste al total.

Los recursos software utilizados han sido:

- Sistema operativo: Windows 8.1.
- Sistema operativo: Windows 7 Professional.
- Software del control de bajo nivel: Arduino v1.6.4.
- Software del control de alto nivel: Matlab R2012a, v7.14.0.739.
- Microsoft Office Professional Plus 2013.

Los recursos hardware utilizados han sido:

- Ordenador portátil: Hacer Aspire E15
- Ordenador de sobremesa: HP Intel Core 2.
- Herramientas de trabajo: Destornilladores, tijeras, entre otros.
- Material de laboratorio: Polímetro, soldador, entre otros.
- Dispositivos electrónicos: Fuente de alimentación y osciloscopio.

8.3. Costes directos

Los costes directos deben evaluarse en tres secciones diferentes, a saber:

- Coste personal.
- Costes amortizables de programas y equipos.
- Coste de materiales directos empleados.

Todos ellos se van a desarrollar a continuación.

Coste de personal

Para la realización del presente proyecto ha sido necesario un ingeniero que llevara a cabo todo el proceso de diseño, montaje eléctrico, programación de los códigos necesarios, realización de pruebas y validación final del robot.

Para calcular el coste de este trabajo, en primer lugar se supondrá un coste anual para un ingeniero y una vez calculado, se adecuará este coste al total de horas trabajadas por el mismo.

Si se tiene en cuenta el sueldo anual bruto con posibles incentivos por su trabajo, y la cotización a la Seguridad Social, la cual es un 35% del sueldo bruto total, se tiene el siguiente coste total anual de un ingeniero:

COSTE ANUAL DE UN INGENIERIO	
Sueldo anual bruto e incentivos	30.000 €
Seguridad Social	10.500 €
Sueldo total	**40.500 €**

Tabla 20. Coste anual de un ingeniero.

Una vez calculado el coste de un ingeniero durante un año, es necesario conocer el número de horas trabajadas por el mismo durante el mismo.

Realizando una estimación de los días trabajados:

DÍAS EFECTIVOS POR AÑO	
Año medio	365,25 días
Sábados y Domingos	-104,35 días
Días de vacaciones efectivos	-20 días
Días festivos reconocidos	-15 días
Días perdidos estimados	-5 días
Total días efectivos estimados	**220,89 días**

Tabla 21.Días efectivos por año.

Considerando que la jornada laboral es de 8 horas diarias, se puede saber cuántas horas trabaja un ingeniero medio al año:

$$220{,}89\,\frac{días}{año}\cdot 8\,\frac{horas}{día}=1.767{,}12\,\frac{horas}{año}$$

Sabiendo que en un año, un ingeniero medio gana 40500 €, se tiene que:

$$\frac{\frac{40.500\ €}{año}}{\frac{1.767{,}12\ horas}{año}}=\mathbf{22.91}\,\frac{€}{\boldsymbol{hora}}$$

A continuación, se adjunta una tabla orientativa del número de horas empleadas en los diferentes sectores que se requieren para el desarrollo del presente proyecto:

DISTRIBUCIÓN TEMPORAL DE TRABAJO	
Formación y documentación	200 horas
Desarrollo del software	350 horas
Adaptación del hardware	20 horas
Elaboración de la documentación	180 horas
Tiempo total	**750 horas**

Tabla 22. Distribución temporal del trabajo.

Por tanto, el coste final de personal es:

$$22.91\,\frac{€}{hora}\cdot 750\ horas=\mathbf{17.182{,}5}€$$

COSTE PERSONAL DIRECTO	17.182,5 €

Costes amortizables de programas y equipos

Para poder calcular estos costes es necesario calcular los costes totales de los programas y equipos amortizables, para posteriormente calcular dicha amortización, es decir, el precio de cada equipo y programa en función del tiempo utilizado y su tiempo de vida útil. En el apartado *8.2 Recursos empleados* se desarrolló una lista detallada de todo el hardware y el software utilizado que es amortizable, sin especificar los precios de los mismos. Por este motivo, en el presente apartado se adjunta una tabla donde el lector puede consultar el precio de cada producto con su correspondiente amortización.

Se ha estimado que el tiempo de amortización considerada para todos los útiles, herramientas, equipos para tratamiento informático, sistemas y programas informáticos ha sido de 5 años, cuyo coeficiente lineal es del 40%.

MATERIAL	IMPORTE	AMORTIZACIÓN
Sistema operativo Windows 8.1.	95.95 €	38,38 €
Sistema operativo Windows 7 Professional.	140 €	56€
Software Arduino v1.6.4.	0 €	0 €
Software Matlab R2012a, v7.14.0.739.	35 €	14 €
Microsoft Office Professional Plus 2013.	119 € x 2 ordenadores = 238 €	95,2 €
Ordenador portátil Acer Aspire E15	500 €	200 €
Ordenador de sobremesa HP Intel Core 2.	600 €	240 €
Polímetro	20 €	8 €
Soldador	20 €	8 €
Total	**1.648,95 €**	**659,58 €**

Tabla 23.Costes amortizables.

Calculado el precio total de todos los materiales amortizables, es posible el cálculo de los mismos por hora mediante la siguiente expresión:

$$\frac{\textbf{Coste final}}{\textbf{hora}} = \frac{659{,}58\ \frac{€}{\text{año}}}{1.767{,}12\ \frac{horas}{año}} \simeq \mathbf{0{,}37}\ \frac{€}{\textbf{hora}}$$

Si por último, se multiplica este dato por el número de horas trabajadas, se obtendrá el coste total de los equipos y programas amortizables.

$$0{,}37\,\frac{€}{hora}\cdot 750\ horas = \mathbf{277{,}5}\ €$$

COSTE DE AMORTIZACIÓN	277,5 €

Coste de materiales directos empleados

En esta sección se van a tener en cuenta aquellos materiales que no son amortizables y que han sido utilizados para ser colocados en el propio robot.

COSTES DE MATERIALES DIRECTOS	
Arduino Uno	21,16 €
Arduino Mega	37,03 € · 2 unidades = 74.06€
Batería	42 €
Motor DC-x42-550	54 €
Encoder 12PPR 42 x 15	32,60 €
Cableado	0,71 €/m · 2m = 1,42 €
Coste total	**225,24 €**

Tabla 24. Costes de materiales directos.

COSTE DE MATERIALES DIRECTOS	225,24 €

Coste derivado de otros materiales

En este apartado se tendrán en cuenta todos los materiales que no se han detallado en apartados anteriores, pero que han sido también necesarios para la realización del proyecto. En ellos, se encuentran materiales como bolis, lápices, gomas, cuadernos, folios, cartuchos de tinta, pegamento, cinta adhesiva, estaño, entre otros.

Se ha estimado que el precio final de todos ellos ha sido de 50 €.

COSTE DE CONSUMIBLES	50 €

Coste directos totales

Realizado todo el desglose de costes directos, se puede calcular el coste directo total del proyecto.

COSTES DIRECTOS TOTALES	
Coste personal directo	17.182,5 €
Coste de amortización	277,5 €
Coste de materiales directos	225,24 €
Coste de consumibles	50 €
Costes directos totales	**17.735,24 €**

Tabla 25. Costes directos totales.

COSTES DIRECTOS	17.735,24 €

8.4. Costes indirectos

Estos costes hacen referencia a aquellos gastos producidos mientras se desarrolla el proyecto y que no pueden incluirse dentro de los gastos directos. En estos gastos se tiene en cuenta el consumo eléctrico (alimentación de dispositivos electrónicos, calefacción, luz…), el consumo telefónico (internet) y los servicios adicionales (transporte, servicios administrativos…).

COSTES INDIRECTOS	
Consumo telefónico	70 €
Consumo eléctrico	180 €
Servicios auxiliares	150 €
Costes indirectos totales	**400 €**

Tabla 26. Costes indirectos.

COSTES INDIRECTOS	400 €

8.5. Costes totales

Una vez calculados todos los costes, basta con sumarlos para obtener el coste total del presente proyecto.

COSTES TOTALES	
Costes directos	17.735,24 €
Costes indirectos	400 €
Costes totales	**18.135,24 €**

Tabla 27. Costes totales.

Es decir, los costes totales del proyecto son:

COSTES TOTALES DEL PROYECTO	18.135,24 €

9. CONCLUSIONES

A lo largo del desarrollo del presente proyecto se ha ido consiguiendo, de manera satisfactoria, el control de los muchos dispositivos que forman el robot, teniendo que haber realizado previamente un estudio de los mismos para conocer su funcionamiento y modo de empleo.

En primer lugar, se consiguió realizar una correcta lectura de los ultrasonidos y bumpers, dando los resultados esperados. También se realizó una correcta lectura e interpretación de los resultados obtenidos por el pin analógico que informa del valor de la carga de la batería, encendiéndose correctamente los leds pertinentes en cada caso de estado de carga.

A continuación, gracias a programas de control de alto nivel como Matlab, se pudo observar la respuesta de los motores y de los codificadores incrementales ante determinados estímulos, tanto manuales (moviendo las ruedas con la mano) como programados (indicándole a través del control de bajo nivel la realización de un determinado movimiento a los motores).

Tras probar los diferentes programas diseñados, se verificó:

- ✓ La correcta lectura de los codificadores incrementales, dando vueltas completas a las ruedas en varias ocasiones y en ambos sentidos, obteniendo el mismo número de pulsos.
- ✓ El correcto control de los puentes H, pudiendo mover los motores en ambos sentidos sin ningún tipo de movimiento anómalo.
- ✓ El diseño adecuado del control PID, consiguiendo una velocidad bastante uniforme de manera independiente.

Además, gracias a la programación realizada tanto en alto como bajo nivel de las ecuaciones odométricas, se pudo observar su correcta implementación al poder visualizar de manera gráfica la trayectoria del robot.

Por último, se verificó la correcta comunicación I2C implementada entre el Arduino Mega maestro y el Arduino Uno, la comunicación Puerto Serie entre los dos Arduinos Mega y la comunicación Puerto Serie entre el Arduino Mega maestro y el control de alto nivel (ya fuera este último un programa como Matlab o la interface de Arduino o bien ROS) enviando y recibiendo correctamente los datos oportunos a través de las tramas de comunicación pertinentes y en los tiempos establecidos.

Al comunicar finalmente con el control de alto nivel ROS, se pudo comprobar como dicho control recibía correctamente las tramas de comunicación en los tiempos

indicados y como el control de bajo nivel respondía correctamente ante la señales recibidas por ROS, manteniéndose inmóvil al no enviarle ninguna referencia de velocidad y moviéndose a la velocidad indicada al enviárselas.

Además, se quiere transmitir que el éxito de un proyecto no depende únicamente de cumplir todos y cada uno de los objetivos marcados, también depende de la capacidad de subsanar los fallos que se han ido encontrando a lo largo de su desarrollo y de todos los conocimientos adquiridos a lo largo del mismo, habiendo sido necesario aplicar diferentes conocimientos aprendidos a lo largo de la carrera, consiguiendo afianzar y ampliar muchos de ellos al ponerlos en práctica físicamente sobre un robot, además de aprender muchos nuevos conceptos.

Respecto a las dificultades que se han ido encontrando a lo largo del proyecto destaca el número de microcontroladores usados para el control de bajo nivel, ya que de forma inicial se pretendía desarrollar este control usando tan sólo dos microcontroladores Arduino, un Arduino Uno para la lectura de los ultrasonidos y bumpers y un Arduino Mega para el resto de lecturas y controles. Mediante esta configuración se conseguían muy buenos resultados en el control PID de manera independiente, sin embargo, al tratar de controlar la velocidad de los dos motores a la vez, este control se volvía mucho más irregular, no pudiendo controlar de manera satisfactoria la velocidad de los motores. Esto sucedía debido a que la capacidad de procesamiento del Arduino Mega es insuficiente para el control de los dos motores a la vez, por lo que se optó por la realización de un control PID paralelo de cada motor, consiguiendo con control de los motores satisfactorio.

Como ya se describió en los primeros temas, se trata de un robot diseñado para la enseñanza y la investigación, el cual tiene inmensas posibilidades de ampliación y mejora.

Una posible mejora puede venir dada del método de localización del robot. La localización odométrica posiblemente sea uno de los métodos más sencillos de implantar, sin embargo, como ya se ha visto, puede dar lugar a muchos errores si el robot no se desplaza por suelos idóneos y si no se calibra la odometría correctamente. Si se quiere que el robot pueda andar por todo tipo de terrenos, será necesario el uso de otro método de localización que aunque sea más complejo, puede resultar mucho más adecuado.

También hay que tener en cuenta que aunque disponga de 8 ultrasonidos, el robot no podrá detectar todos los obstáculos que se le presenten. La localización de los mismos da lugar a muchas zonas muertas que hacen que cualquier obstáculo de pequeñas dimensiones o móvil sea indetectable, no pudiendo evitar el choque contra el mismo.

Por otro lado, el robot móvil desarrollado puede ser usado como base de otros robots, es decir, encima del robot Edubot puede acoplarse cualquier otro tipo de dispositivo que necesite desplazarse. Al concebirse esta posibilidad, el lector puede ver como se abre un gran abanico de posibilidades, puesto que el robot colocado encima puede ser de cualquier tipo, desde un brazo robótico hasta un humanoide.

Además, al microcontrolador Arduino se le pueden adjuntar multitud de dispositivos, pudiéndole dotar de comunicación bluetooth y de esta manera controlar el robot desde cualquier tipo de dispositivo que también tenga bluetooth. Es decir, no es necesario que el control de alto nivel se encuentre en un ordenador, puede encontrarse en nuestro propio dispositivo móvil.

10. BIBLIOGRAFÍA

[1] - ARDUINO. <https://www.arduino.cc/>

[2] - BERMÚDEZ, G. (2002). "Robots Móviles. Teoría, aplicaciones y experiencias." de la *Universidad Distrital Francisco José de Caldas,* vol.5, no10, 17. <http://tecnura.udistrital.edu.co/ojs/index.php/revista/article/view/225/223>

[3] - BRICOGEEK. *Sensor de distancia por ultrasonidos PINK.* <http://tienda.bricogeek.com/descatalogado/189-sensor-de-distancia-por-ultrasonidos-ping.html>

[4] - CARLETTI, E. (2014). *"Comunicación – Bus I2C. Descripción y funcionamiento"* <*http://robots-argentina.com.ar/Comunicacion_busI2C.htm*>

[5] - CUENTOS CUÁNTICOS (2011). "*Robótica: Estimación de posición por odometría."* < http://cuentos-cuanticos.com/2011/12/15/robotica-estimacion-de-posicion-por-odometria/>

[6] - EL CÓDIGO ASCII. <http://www.elcodigoascii.com.ar/>

[7] - ELECTROENSAIMADA. *"Tutorial Arduino: I2C"* <*http://www.electroensaimada.com/i2c.html*>

[8] - FRAILE MORA, J.(2008). *Máquinas Eléctricas.* Madrid: McGraw-Hill.

[9] - GARCÍA, V. (2014). "*El puente H".* <http://www.hispavila.com/3ds/atmega/hpuente.html>

[10] - GONZÁLEZ ARÉVALO, E. (2009). *"Medidor de Velocidad y Posición con un Encoder"* de la *Universidad Tecnológica de la Mixteca.* <http://webs.ono.com/lmoliver/usr_1482483054.pdf>

[11] - GRUPO DE INVESTIGACIÓN SUPRESS (2008). *"El controlador PID básico".* <*http://www.lra.unileon.es/es/book/export/html/268*>

[12] - INSTITUO NACIONAL DE TECNOLOGIAS EDUCATIVAS Y DE FORMACIÓN DEL PROFESORADO. *Los lenguajes de programación.* <http://www.ite.educacion.es/formacion/materiales/47/cd/mod3/3a_1.htm>

[13] - JAVIERSB. (2012) *"Cómo leer datos de Arduino con Matlab a través del puerto USB"* en ZYGZAX, 12 de Septiembre. <http://zygzax.com/como-leer-datos-de-arduino-con-matlab-a-traves-del-puerto-usb/>

[14] - LINARES DIÉGUEZ,J (2015). "*Robot recorre tuberías. Circuito y funcionamiento sensor HC.SR04K."* <http://granabot.es/robot-recorre-tuberias-circuito-y-funcionamiento-sensor-hc-sr04/>

[15] - MORILLA GARCÍA, F.(2007) *"El controlador PID"* de la ETSI de Informática, UNED. <http://www.dia.uned.es/~fmorilla/MaterialDidactico/El%20controlador%20PID.pdf>

[16] - RODRÍGUEZ DEL RÍO R. *"Gráficas con Matlab"* de la Universidad Complutense de Madrid. <http://www.mat.ucm.es/~rrdelrio/documentos/rrrescorial2002.pdf>

[17] - SCOTT C. (2012) *"Simple Arduino Serial Communication"* en Arduino Basics, 4 de Julio. <http://arduinobasics.blogspot.com.es/2012/07/arduino-basics-simple-arduino-serial.html>

[18] - VALENCIA, J. MONTOYA, A. HERNANDO RIOS, L. (2009). *"Modelo cinemático de un robot móvil tipo diferencial y navegación a partir de la estimación odométrica"* de la Universidad Tecnológica de Pereira. No 41, ISSN 0122-1701.

[19] - UNIVERSIDAD DE ALMERÍA. *"Robótica Móvil."* <http://www.ual.es/personal/rgonzalez/papers/Robotica_movil.pdf>

[20] - UNIVERSIDAD DE SEVILLA. "Sistemas de locomoción de robots móviles". <http://www.esi2.us.es/~vivas/ayr2iaei/LOC_MOV.pdf>

[21] - VERDEJO, D. (2011). *"Diseño y construcción de un robot didáctico controlado mediante plataforma Player Stage"*. Proyecto Final de Carrera. Valladolid: Escuela de Ingenierías Industriales.

11. ANEXOS

A lo largo de la presente memoria se han ido indicando una serie de Anexo con el fin de no introducirlos el contenido de los mismos a lo largo de la redacción y así no perder el hilo de la explicación y no cargar de manera innecesaria algunos de los capítulos.

A modo resumen y de índice, los Anexos que se podrán ver a continuación son:

Anexo 1. Esquema Arduino Uno.

Anexo 2. Esquema Arduino Mega.

Anexo 3. Recepción de ultrasonidos y bumpers.

Anexo 4. Estado de la batería.

Anexo 5. Encoders – Contador de pulsos.

Anexo 6. Encoder – Contar vuelta.

Anexo 7. Control de puentes H.

Anexo 8. Control de la velocidad del motor.

Anexo 9. Código Arduino Uno.

Anexo 10. Código Arduino Mega esclavo.

Anexo 11. Código Arduino Mega maestro.

Anexo 12. Representación PID en Matlab.

Anexo 13. Programa de comprobación. Matlab.

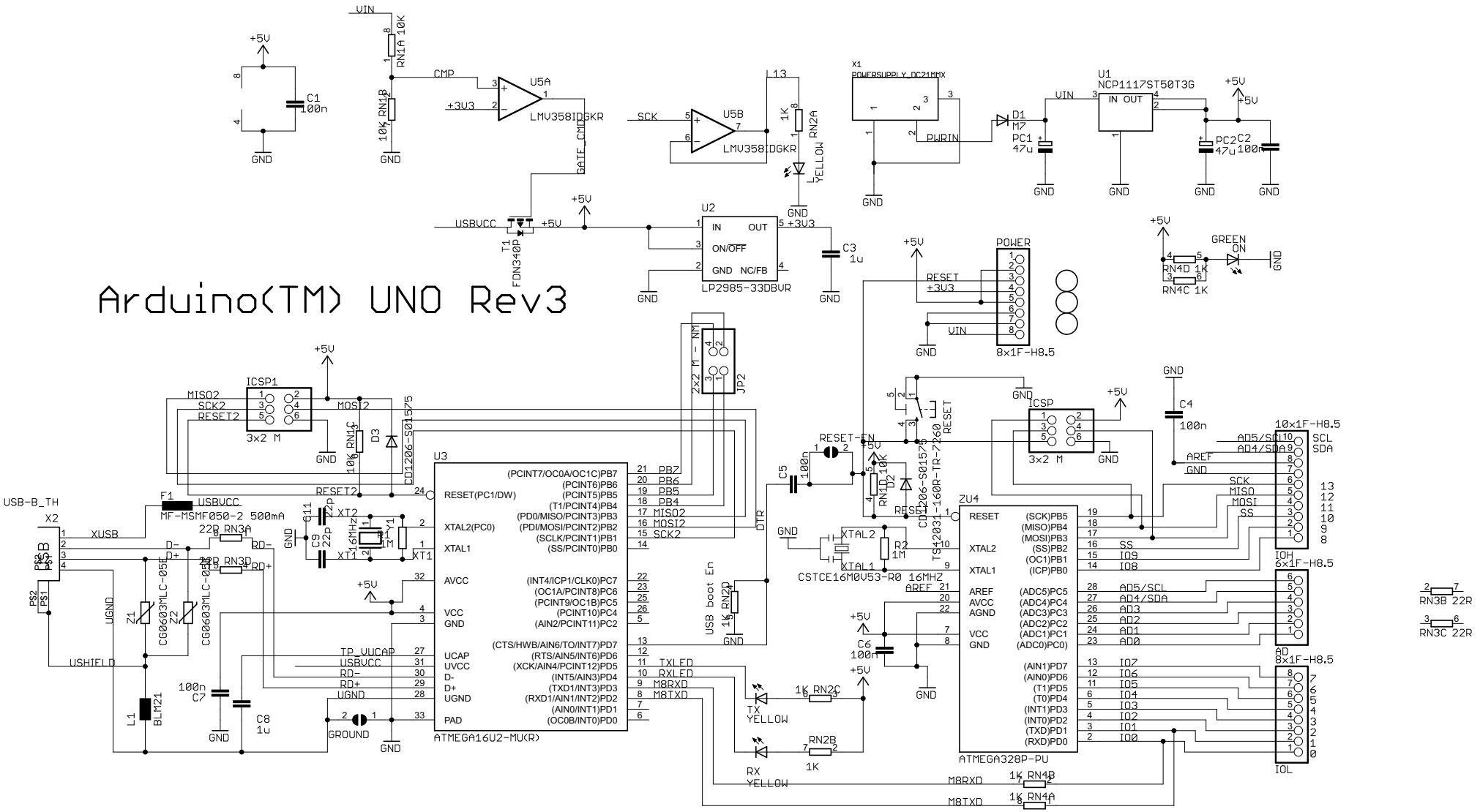
Anexo 14. Programa de comprobación. Arduino Mega.

Anexo 15. Odometría. Arduino.

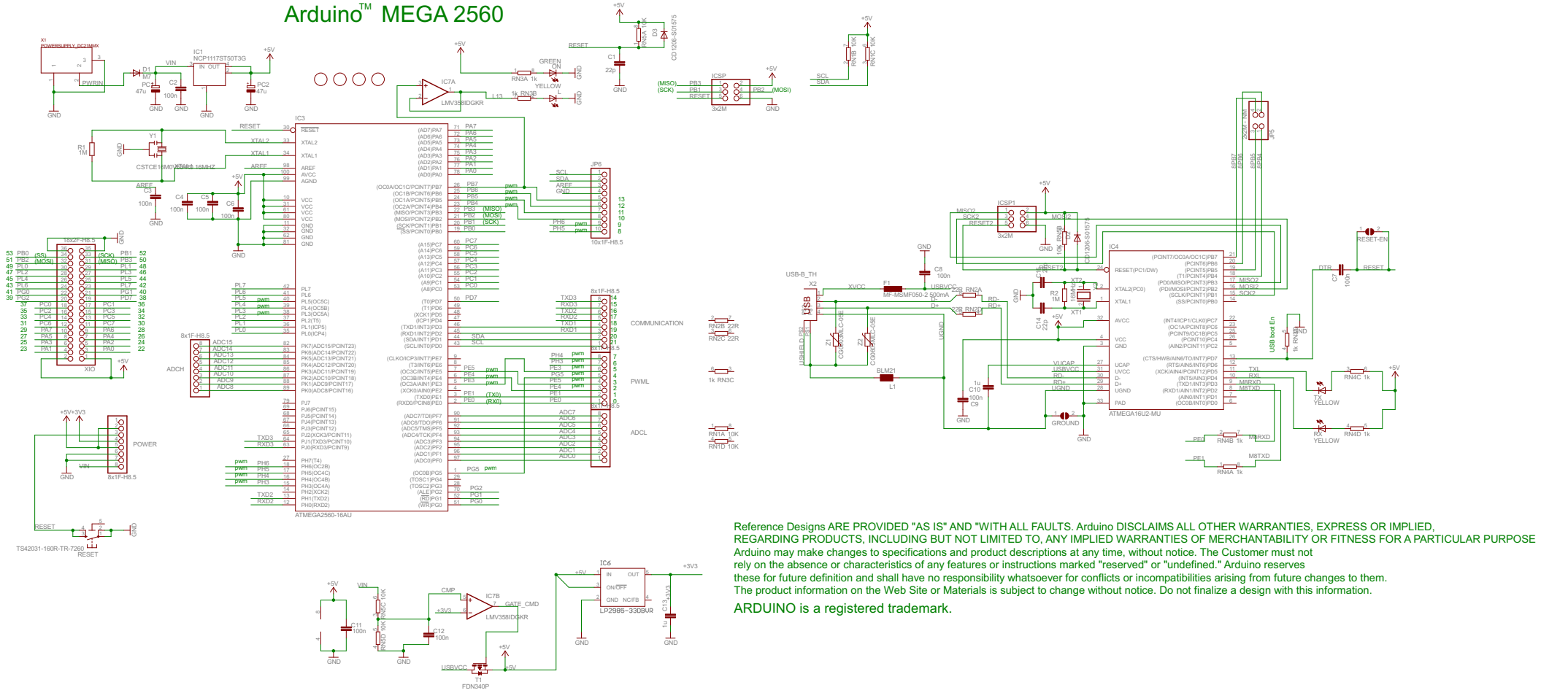
Anexo 16. Odometría. Matlab.

Anexo 17. Simulación final. Arduino maestro.

Anexo 18. Simulación final. Matlab.



Arduino™ MEGA 2560



Reference Designs ARE PROVIDED "AS IS" AND "WITH ALL FAULTS. Arduino DISCLAIMS ALL OTHER WARRANTIES, EXPRESS OR IMPLIED, REGARDING PRODUCTS, INCLUDING BUT NOT LIMITED TO, ANY IMPLIED WARRANTIES OF MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE Arduino may make changes to specifications and product descriptions at any time, without notice. The Customer must not rely on the absence or characteristics of any features or instructions marked "reserved" or "undefined." Arduino reserves these for future definition and shall have no responsibility whatsoever for conflicts or incompatibilities arising from future changes to them. The product information on the Web Site or Materials is subject to change without notice. Do not finalize a design with this information.

ARDUINO is a registered trademark.

```
//*****************************************************************//
//              ANEXO 3. RECEPCIÓN DE SONARS Y BUMPERS             //
//*****************************************************************//

//********************* DECLARACIÓN DE VARIABLES **********************//
// Sonars
//{S1F,S2F,S3F,S4F,S4B,S3B,S2B,S1B}
const int sonar[] = {5, 4, 3, 2, 7, 8, 9, 10};
long duracion_sonar[7];  // Tiempo detectado por el sonar
long int long_sonar[7];  // Tiempo convertido a cm
// Bumpers
const int bumper[] = {6, 11}; // {DELANTE, DETRAS}
int info_bumper[1];   // Estado bumper
int push_bumper[1];   // Estado bumper numérico

// Parámetros necesarios para bucles
int a,b,c,d;

//*****************************************************************//
// Nombre de la función: setup()
// Función: Función principal de inicialización de variables.
//          Se ejecuta una sóla vez. En ella se realiza lo siguiente:
//             - Inicia la comunicación puerto serie.
//             - Define los bumpers como entradas.
//             - Pone en alto la resistencia de Pull-Up de cada bumper.
// Entradas: Ninguna.
// Salidas: Ninguna.
//*****************************************************************//
void setup()
{
  Serial.begin(9600);

  // Entradas
  for(a=0 ; a<2 ; a++)
  {
    pinMode(bumper[a], INPUT);
  }
  // Resistencia de pull-up
  for(b=0 ; b<2 ; b++)
  {
    digitalWrite(bumper[b], HIGH);
  }
}
```

```
//*************************************************************************//
// Nombre de la función: loop()
// Función: Función principal. Actúa como bucle, es decir, se ejecuta
//          cíclicamente.
// Entradas: Ninguna.
// Salidas: Ninguna.
//*************************************************************************//
void loop()
{
  // Lectura sonars
 Lectura_sonar();

  // Lectura bumpers
 Lectura_bumper();

  delay(10);
}

//*************************************************************************//
// Nombre de la función: Lectura_sonar()
// Llamada por: loop()
// Función: Encargada de leer la lectura de todos los ultrasonidos
//          presentes en el robot.
// Entradas: Ninguna.
// Salidas: Imprime por pantalla la distancia del obstáculo más próximo
//          al sensor junto al identificardor del mismo.
//*************************************************************************//
void Lectura_sonar()
{
  for(c=0 ; c<8 ; c++)
  {
    // Lectura del sonar
    pinMode(sonar[c], OUTPUT);
    digitalWrite(sonar[c], LOW);
    delayMicroseconds(2);
    digitalWrite(sonar[c], HIGH);
    delayMicroseconds(5);
    digitalWrite(sonar[c], LOW);
    pinMode(sonar[c], INPUT);
   duracion_sonar[c] = pulseIn(sonar[c], HIGH);
  long_sonar[c] = microsecondsToCentimeters(duracion_sonar[c]);
    delayMicroseconds(2);

    // Impresión por pantalla
    Serial.print("Lectura Sonar ");
    Serial.print(c);
    Serial.print(": ");
    Serial.print(long_sonar[c]);
  }
```

```
}

//*************************************************************//
// Nombre de la función: Lectura_bumper()
// Llamada por: loop()
// Función: Encargada de leer el estado de cada bumper y por tanto, de
//          indicar si el mismo está o no pulsado.
// Entradas: Ninguna.
// Salidas: Imprime por pantalla si está o no pulsado cada bumper.
//*************************************************************//
void Lectura_bumper()
{
  for(d=0 ; d<2 ; d++)
  {
    // Lectura estado bumper
    info_bumper[d] = digitalRead(bumper[d]);
    // Bumper no pulsado
    if (info_bumper[d] == HIGH)
    {
      push_bumper[d] = 0;
      Serial.print("bumper \t");
      Serial.println(d+1);
      Serial.print("no pulsado \n");
    }
    // Bumper pulsado
    else
    {
      push_bumper[d] = 1;
      Serial.print("bumper \t");
      Serial.println(d+1);
      Serial.print("pulsado \n");
    }
  }
}

//*************************************************************//
// Nombre de la función: microsecondsToCentimeters()
// Llamada por: Lectura_sonar()
// Función: Encargada de traducir el tiempo de eco en cm.
// Entradas: El tiempo de eco.
// Salidas: La distancia del objeto en cm.
//*************************************************************//
long microsecondsToCentimeters(long microseconds)
{
  return microseconds / 29 / 2;
}
```

```
//*****************************************************************//
//                  ANEXO 4. ESTADO DE LA BATERÍA                  //
//*****************************************************************//

//********************* DECLARACIÓN DE VARIABLES *********************//
// Variables digitales
int Led_Verde = 13;
int Led_Amarillo = 12;
int Led_Rojo = 11;
// Variable analógica
int Bateria = A0;
// Otras variables utiles
float Lectura_bat; // Valor donde se almacena el valor de la bateria
float Valor_bat;   // Valor convertido de la lectura
int Alarma_baja = 0;    // Alarma=0 Al. inactiva. Alarma=1 Al. activada
int Alarma_critica = 0; // Alarma=0 Al. inactiva. Alarma=1 Al. activada

//*****************************************************************//
// Nombre de la función: setup()
// Función: Función principal de inicialización de variables.
//          Se ejecuta una sóla vez. En ella se realiza lo siguiente:
//             - Inicia la comunicación puerto serie.
//             - Define los leds como salidas.
//             - Apaga todos los leds.
// Entradas: Ninguna.
// Salidas: Ninguna.
//*****************************************************************//
void setup()
{
  Serial.begin(9600);

  // Salidas
  pinMode(Led_Verde, OUTPUT);
  pinMode(Led_Amarillo, OUTPUT);
  pinMode(Led_Rojo, OUTPUT);

  // Inicialmente todas las luces apagadas
  digitalWrite(Led_Verde, LOW);
  digitalWrite(Led_Amarillo, LOW);
  digitalWrite(Led_Rojo, LOW);
}

//*****************************************************************//
// Nombre de la función: loop()
// Función: Función principal. Actúa como bucle, es decir, se ejecuta
//          cíclicamente.
// Entradas: Ninguna.
// Salidas: Ninguna.
//*****************************************************************//
```

```
void loop()
{
  // Lectura del estado de la batería
 Lectura_bateria();

  // Acciones en función del estado de la batería.
 Control_estado();
}

//*****************************************************************//
// Nombre de la función: Lectura_bateria()
// Llamada por: loop()
// Función: Encargada de leer el pin analógico y convertir la lectura
//          en el valor real de la batería (entre 0 y 12.5V).
// Entradas: Ninguna.
// Salidas: Ninguna
//*****************************************************************//
void Lectura_bateria()
{
  Lectura_bat = analogRead(Bateria);

 Valor_bat = (Lectura_bat*12.5)/1023;
}

//*****************************************************************//
// Nombre de la función: Control_estado()
// Llamada por: loop()
// Función: Encargada de actuar en función de la carga de la bateria:
//          - Bateria cargada (12.5 V,11.3 V): Led verde encendido.
//                                             No alarmas.
//          - Bateria media (11.3 V,10.8 V): Led amarillo encendido.
//                                           Alarma bateria baja.
//          - Bateria baja (10.8 V,0 V): Led rojo encendido.
//                                       Alarma bateria crítica.
// Entradas: Ninguna.
// Salidas: Encendido y apagado de los leds.
//*****************************************************************//
void Control_estado()
{
  // Bateria cargada
  if(Valor_bat>11.3)
  {
    digitalWrite(Led_Verde, HIGH);
    digitalWrite(Led_Amarillo, LOW);
    digitalWrite(Led_Rojo, LOW);
    Alarma_baja = 0;
    Alarma_critica = 0;
  }
  // Bateria con carga media
```

```
  if(Valor_bat<=11.3 && Valor_bat>10.8)
  {
    digitalWrite(Led_Verde, LOW);
    digitalWrite(Led_Amarillo, HIGH);
    digitalWrite(Led_Rojo, LOW);
    Alarma_baja = 1;
    Alarma_critica = 0;
  }
  // Bateria con carga crítica
  if(Valor_bat<=10.8)
  {
    digitalWrite(Led_Verde, LOW);
    digitalWrite(Led_Amarillo, LOW);
    digitalWrite(Led_Rojo, HIGH);
    Alarma_baja = 0;
    Alarma_critica = 1;
  }
}
```

```
//*****************************************************************//
//              ANEXO 5. ENCODERS - CONTADOR DE PULSOS             //
//*****************************************************************//

//********************* DECLARACIÓN DE VARIABLES **********************//
// Motor
const int MotorDIR = 4; // Pin DIR del puente H
const int MotorPWM = 5; // Pin PWM del puente H
const int enable = 6;   // Habilitación del motor
// Encoder
const int CanalB = 3; // PIN del canal B del encoder
const int CanalA = 2; // PIN del canal A del encoder
volatile int Contador = 0; // Contador de pulsos del encoder
int LastContador = 0;      // Contador de pulsos del encoder
volatile unsigned int EstadoCanalB = 0;  // Estado del canal B
volatile unsigned int EstadoCanalA = 0;  // Estado del canal A

//*****************************************************************//
// Nombre de la función: setup()
// Función: Función principal de inicialización de variables.
//          Se ejecuta una sóla vez. En ella se realiza lo siguiente:
//             - Inicia la comunicación puerto serie.
//             - Define los pines PWM y de dirección como salidas.
//             - Define los canales de los encoders como entradas.
//             - Pone los canales del encoder a HIGH.
//             - Define el pin de habilitación como entrada.
//             - Pone el pin de habilitación en HIGH.
//             - Activa la interrupción del encoder.
// Entradas: Ninguna.
// Salidas: Ninguna.
//*****************************************************************//
void setup() {
  Serial.begin(115200);

  // Salidas
  pinMode(MotorPWM, OUTPUT);
  pinMode(MotorDIR, OUTPUT);

  // Entradas
  pinMode(CanalB, INPUT);
  digitalWrite(CanalB, HIGH);
  pinMode(CanalA, INPUT);
  digitalWrite(CanalA, HIGH);
  pinMode(enable, INPUT);

  // Habilitación de motor
  digitalWrite(enable,HIGH);

  // Interrupción.
```

```
  //Mega:(interr,pin)[(0,2),(1,3),(2,21),(3,20),(4,19),(5,18)]
  attachInterrupt(0,Contar,CHANGE);
}

//*************************************************************************//
// Nombre de la función: loop()
// Función: Función principal. Actúa como bucle, es decir, se ejecuta
//          cíclicamente.
// Entradas: Ninguna.
// Salidas: Imprime por pantalla los pulsos contandos por el encoder.
//*************************************************************************//
void loop()
{
  // Se imprime por pantalla resultados
  Serial.println("Contador encoder:");
  Serial.println(Contador);

  delay(1000);
}

//*************************************************************************//
// Nombre de la función: Contar()
// Llamada por: la interrupción número 0.
// Función: Encargada de leer los canales del encoder y en función de
//          lo leído, se actualizan los pulsos generados.
// Entradas: Ninguna.
// Salidas: Ninguna.
//*************************************************************************//
void Contar()
{
  EstadoCanalB = digitalRead(CanalB);
  EstadoCanalA = digitalRead(CanalA);

  if(EstadoCanalA == HIGH)
  {
    if(EstadoCanalB == LOW)
      Contador = Contador+1;
    else
      Contador = Contador-1;
  }
  else
  {
    if(EstadoCanalB == LOW)
      Contador = Contador-1;
    else
      Contador = Contador+1;
  }
}
```

```
//*****************************************************************//
//                ANEXO 6. ENCODER - CONTAR VUELTA                 //
//*****************************************************************//

//******************** DECLARACIÓN DE VARIABLES **********************//
// Motor
const int MotorDIR = 4; // Pin DIR del puente H
const int MotorPWM = 5; // Pin PWM del puente H
const int enable = 6;   // Habilitación del motor
double Velocidad_motor = 0; // Velocidad que introducimos al motor
// Canales encoder
const int CanalB = 3; // PIN del canal B del encoder
const int CanalA = 2; // PIN del canal A del encoder
volatile int Contador = 0; // Contador de pulsos del encoder
int LastContador = 0;      // Contador de pulsos del encoder
volatile unsigned int EstadoCanalB = 0;  // Estado del canal B
volatile unsigned int EstadoCanalA = 0;  // Estado del canal A
// Otras variables útiles
int Cambio = 0; // Indicación de cambio de sentido

//*****************************************************************//
// Nombre de la función: setup()
// Función: Función principal de inicialización de variables.
//         Se ejecuta una sóla vez. En ella se realiza lo siguiente:
//            - Inicia la comunicación puerto serie.
//            - Define los pines PWM y de dirección como salidas.
//            - Define los canales del encoder como entradas.
//            - Pone los canales del encoder a HIGH.
//            - Define el pin de habilitación como entrada.
//            - Pone el pin de habilitación en HIGH.
//            - Activa la interrupción del encoder.
// Entradas: Ninguna.
// Salidas: Ninguna.
//*****************************************************************//
void setup(){
  Serial.begin(9600);

  // Salidas
  pinMode(MotorPWM, OUTPUT);
  pinMode(MotorDIR, OUTPUT);

  // Entradas
  pinMode(CanalB, INPUT);
  digitalWrite(CanalB, HIGH);
  pinMode(CanalA, INPUT);
  digitalWrite(CanalA, HIGH);
  pinMode(enable, INPUT);

  // Habilitación de motor
```

```
  digitalWrite(enable,HIGH);

  // Interrupción.
  //Mega:(interr,pin)[(0,2),(1,3),(2,21),(3,20),(4,19),(5,18)]
  attachInterrupt(0,Contar,CHANGE);
}

//*************************************************************************//
// Nombre de la función: loop()
// Función: Función principal. Actúa como bucle, es decir, se ejecuta
//          cíclicamente.
// Entradas: Ninguna.
// Salidas: Ninguna.
//*************************************************************************//
void loop()
{
  // Se imprimen por pantalla los resultados
  Serial.println("Contador encoder:");
  Serial.println(Contador);

  // Gira hacia atrás
  if(Cambio == 0)
  {
    if (Contador >= -490)
      Girar(-20);
    else
    {
      Girar(0);
      Cambio = 1;
    }
  }
  // Gira hacia delante
  if (Cambio == 1)
  {
    if (Contador <= 0)
      Girar(20);
    else
    {
      Girar(0);
      Cambio = 0;
    }
  }
  delay(10);
}

//*************************************************************************//
// Nombre de la función: Contar()
// Llamada por: la interrupción número 0.
// Función: Encargada de leer los canales del encoder y en función de
```

```
//          lo leído, se actualizan los pulsos generados.
// Entradas: Ninguna.
// Salidas: Ninguna.
//*************************************************************************//
void Contar()
{
  EstadoCanalB = digitalRead(CanalB);
  EstadoCanalA = digitalRead(CanalA);

  if(EstadoCanalA == HIGH)
  {
    if(EstadoCanalB == LOW)
      Contador = Contador+1;
    else
      Contador = Contador-1;
  }
  else
  {
    if(EstadoCanalB == LOW)
      Contador = Contador-1;
    else
      Contador = Contador+1;
  }
}

//*************************************************************************//
// Nombre de la función: Girar()
// Llamada por: loop()
// Función: Encargada de girar el motor.
// Entradas: Velocidad a la que debe girar el motor.
// Salidas: Movimiento de motor.
//*************************************************************************//
void Girar(double Velocidad_motor)
{
  Serial.println("Velocidad que entra en la función:");
  Serial.println(Velocidad_motor);

  if(Velocidad_motor==0)
  {
    digitalWrite(MotorDIR,LOW);
    analogWrite(MotorPWM, 0);
  }
  if(Velocidad_motor>0)
  {
    digitalWrite(MotorDIR,LOW);
    analogWrite(MotorPWM, (int)Velocidad_motor);
  }
  if(Velocidad_motor<0)
  {
```

```
    digitalWrite(MotorDIR,HIGH);
    analogWrite(MotorPWM, (int)Velocidad_motor);
  }
}
```

```
//*****************************************************************//
//                    ANEXO 7. CONTROL DE PUENTES H                //
//*****************************************************************//

//********************* DECLARACIÓN DE VARIABLES **********************//
#define VelMax 255 // Máxima velocidad permitida

// Motor
const int MotorDIR = 4; // Pin DIR del puente H
const int MotorPWM = 5; // Pin PWM del puente H
const int enable = 6;   // Habilitación del motor
double Velocidad_ref = 0; // Velocidad introducida por teclado

// Variable donde se almacena el valor introducido por teclado
int  Incoming = 0;

//*****************************************************************//
// Nombre de la función: setup()
// Función: Función principal de inicialización de variables.
//          Se ejecuta una sóla vez. En ella se realiza lo siguiente:
//             - Inicia la comunicación puerto serie.
//             - Define los pines PWM y de dirección como salidas.
//             - Define el pin de habilitación como entrada.
//             - Pone el pin de habilitación en HIGH.
// Entradas: Ninguna.
// Salidas: Ninguna.
//*****************************************************************//
void setup(){
  Serial.begin(9600);

  // Salidas
  pinMode(MotorPWM, OUTPUT);
  pinMode(MotorDIR, OUTPUT);

  // Entrada
  pinMode(enable, INPUT);

  // Habilitación de motor
  digitalWrite(enable,HIGH);
}

//*****************************************************************//
// Nombre de la función: loop()
// Función: Función principal. Actúa como bucle, es decir, se ejecuta
//          cíclicamente.
// Entradas: Ninguna.
// Salidas: Movimiento del motor.
//*****************************************************************//
void loop()
```

```
{
  if(Serial.available()>0)
  {// Lectura del puerto serie
   Incoming = Serial.read()-48;

   if (Incoming==0){
     Velocidad_ref = 0;
     digitalWrite(MotorDIR,LOW);
     analogWrite(MotorPWM, (int)Velocidad_ref);
     Serial.println(Velocidad_ref);
   }
   else if (Incoming==1)
   {
    Velocidad_ref = Velocidad_ref+10;
     Serial.println(Velocidad_ref);
   }
   else if (Incoming==2)
   {
      Velocidad_ref = Velocidad_ref-10;
       Serial.println(Velocidad_ref);
   }

   // ¡¡PRECAUCIÓN!! La velocidad no debe superar lo +-255
   if(Incoming!=0){
     Velocidad_ref = constrain(Velocidad_ref,-VelMax,VelMax);

     if(Velocidad_ref>=0)
     {
       digitalWrite(MotorDIR,LOW);
       analogWrite(MotorPWM, (int)Velocidad_ref);
     }

     if(Velocidad_ref<0)
     {
       digitalWrite(MotorDIR,HIGH);
       analogWrite(MotorPWM, (int)Velocidad_ref);
     }
   }
   delay(10);
  }
}
```

```
//*****************************************************************//
//          ANEXO 8. CONTROL DE LA VELOCIDAD DEL MOTOR             //
//*****************************************************************//

//********************* DECLARACIÓN DE VARIABLES **********************//
#define VelMax 255       // Máxima velocidad permitida

// Motor
const int MotorDIR = 4; // Pin DIR del puente H
const int MotorPWM = 5; // Pin PWM del puente H
const int enable = 6;   // Habilitación del motor
// Encoder
const int CanalB = 3; // PIN del canal B del encoder
const int CanalA = 2; // PIN del canal A del encoder
volatile int Contador = 0; // Contador de pulsos del encoder
int LastContador = 0;      // Contador de pulsos del encoder
volatile unsigned int EstadoCanalB = 0;  // Estado del canal B
volatile unsigned int EstadoCanalA = 0;  // Estado del canal A
// Velocidad
double W_ref = 0;  // Velocidad angular de referencia
double W_act = 0;  // Velocidad que medimos en el motor
int dp = 0;        // Pulsos medidos del encoder
double Velocidad_motor = 0; // Velocidad que introducimos al motor
// Periodos de comunicación
unsigned long  Pmuestreo = 200; // Periodo de muestreo
// Variables temporales
unsigned long LastTime;     // Tiempo anterior
unsigned long now;          // Tiempo actual
unsigned long dT;           // Diferencia de tiempos
//************ Errores PID
// Parámetros PID
double Kp,Ki,Kd;
// Derecho
double Eactual, Eproporcional;  // Error proporcional
double Eint=0, Eintegral;       // Error integral
double Eder, Ederivativo;       // Error derivativo
doubleError,LastError=0;        // Error total

//*****************************************************************//
// Nombre de la función: setup()
// Función: Función principal de inicialización de variables.
//         Se ejecuta una sóla vez. En ella se realiza lo siguiente:
//            - Inicia la comunicación puerto serie.
//            - Define los pines PWM y de dirección como salidas.
//            - Define los canales del encoder como entradas.
//            - Pone los canales del encoder a HIGH.
//            - Define el pin de habilitación como entrada.
//            - Pone el pin de habilitación en HIGH.
//            - Activa la interrupción del encoder.
```

```
//             - Toma el tiempo actual.
//             - Establece una velocidad nula para el motor.
//             - Define los parámetros del PID.
// Entradas: Ninguna.
// Salidas: Ninguna.
//*********************************************************************//
void setup() {
  Serial.begin(9600);

  // Salidas
  pinMode(MotorPWM, OUTPUT);
  pinMode(MotorDIR, OUTPUT);

  // Entradas
  pinMode(CanalB, INPUT);
  digitalWrite(CanalB, HIGH);
  pinMode(CanalA, INPUT);
  digitalWrite(CanalA, HIGH);
  pinMode(enable, INPUT);

  // Habilitación de motor
  digitalWrite(enable,HIGH);

  // Interrupción.
  //Mega:(interr,pin)[(0,2),(1,3),(2,21),(3,20),(4,19),(5,18)]
  attachInterrupt(0,Contar,CHANGE);

  // Tiempo actual
  LastTime = millis();

  // Velocidad inicial nula
  digitalWrite(MotorDIR,LOW);
  analogWrite(MotorPWM, 0);

  //Definición de los parámetros del PID
  Kp = 120;
  Ki = 0.003;
  Kd = 0;
}

//*********************************************************************//
// Nombre de la función: loop()
// Función: Función principal. Actúa como bucle, es decir, se ejecuta
//          cíclicamente.
// Entradas: Ninguna.
// Salidas: Ninguna.
//*********************************************************************//
void loop()
{
```

```
  W_ref = 0.2;            // Se estable velocidad de referencia

  // Estado del motor
  Compute();              // Cálculo del PID
 Girar(Velocidad_motor); // Movimiento del motor
  delay(10);
}

//*********************************************************************//
// Nombre de la función: Contar()
// Llamada por: la interrupción número 0.
// Función: Encargada de leer los canales del encoder y en función de
//          lo leído, se actualizan los pulsos generados.
// Entradas: Ninguna.
// Salidas: Ninguna.
//*********************************************************************//
void Contar()
{
  EstadoCanalB = digitalRead(CanalB);
  EstadoCanalA = digitalRead(CanalA);

  if(EstadoCanalA == HIGH)
  {
    if(EstadoCanalB == LOW)
      Contador = Contador+1;
    else
      Contador = Contador-1;
  }
  else
  {
    if(EstadoCanalB == LOW)
      Contador = Contador-1;
    else
      Contador = Contador+1;
  }
}

//*********************************************************************//
// Nombre de la función: Girar()
// Llamada por: loop()
// Función: Encargada de girar el motor.
// Entradas: Velocidad a la que debe girar el motor.
// Salidas: Movimiento del motor.
//*********************************************************************//
void Girar(double Velocidad_motor)
{
  if(Velocidad_motor==0)
  {
    digitalWrite(MotorDIR,LOW);
```

```
    analogWrite(MotorPWM, 0);
  }
  if(Velocidad_motor>0)
  {
    digitalWrite(MotorDIR,LOW);
    analogWrite(MotorPWM, (int)Velocidad_motor);
  }
  if(Velocidad_motor<0)
  {
    digitalWrite(MotorDIR,HIGH);
    analogWrite(MotorPWM, (int)Velocidad_motor);
  }
}

//*********************************************************************//
// Nombre de la función: Compute()
// Llamada por: loop()
// Función: Encargada de calcular el error PID del motor y de
//          mantener la velocidad estable.
// Entradas: Ninguna.
// Salidas: Error PID y velocidad que debe ser introducida al motor.
//*********************************************************************//
void Compute()
{
  now = millis();

  if(now>LastTime+Pmuestreo)
  {
    // Diferencial de tiempo
    dT = now-LastTime;
    // Diferencial de pulsos
    dp = (Contador-LastContador);
    // Velocidad angular medida
    W_act = (double)dp/dT;

    // Imprimimos por pantalla:
    Serial.println(W_act);
    Serial.println(Eproporcional);
    Serial.println(Eintegral);

    // Actualización de valores
    LastContador = Contador;
    LastTime = now;
   }

   if(W_act==0 && W_ref!=0)
     Eint = 0;

   if(W_ref<=0 && W_ref>=-0.2)
```

```
    {
      digitalWrite(MotorDIR,LOW);
      analogWrite(MotorPWM, 0);
    }

    // CÁLCULO DEL ERROR PID
    // Error proporcional
   Eactual = W_ref-W_act;
   Eproporcional = Eactual*Kp;

    // Error integral
    if(Error >= VelMax && Error > 0)
      Error = VelMax;
    else
      Eint += Eactual*dT;

    if(Error <= VelMax && Error < 0)
      Error = -VelMax;
    else
      Eint += Eactual*dT;

   Eintegral = Eint*Ki;

    // Error derivativo
   Eder = (Eactual-LastError)/dT;
   Ederivativo = Eder*Kd;

    // Error total
  Error = Eproporcional+Eintegral+Ederivativo;

    // Cálculo de la velocidad
   Velocidad_motor = Error;
    // Suponemos una velocidad máxima y mínima, por si se supera
   Velocidad_motor = constrain(Velocidad_motor,-VelMax,VelMax);

    // Actualización de valores
   LastError = Eactual;
}
```

```
//*****************************************************************//
//                    ANEXO 9. CÓDIGO ARDUINO UNO                  //
//*****************************************************************//

//********************* DECLARACIÓN DE VARIABLES **********************//
// Comunicación entre arduinos
#include <Wire.h>
// Sonars
const int sonar[] = {5, 4, 3, 2, 7, 8, 9, 10}; //{S1F,S2F,S3F,S4F,S4B,
                                                //  S3B,S2B,S1B}
long duracion_sonar[7];  // Tiempo detectado por el sonar
long int long_sonar[7];  // Tiempo convertido a cm
// Bumpers
const int bumper[] = {6, 11}; // {DELANTE, DETRAS}
int info_bumper[1];   // Estado bumper
int push_bumper[1];   // Estado bumper numérico
// Almacenamiento
uint8_t data[9];
// Parámetros necesarios para bucles
int a,b,c,d,e,i;

//*****************************************************************//
// Nombre de la función: setup()
// Función: Función principal de inicialización de variables.
//          Se ejecuta una sóla vez. En ella se realiza lo siguiente:
//             - Inicia la comunicación puerto serie.
//             - Inicia la comunicación I2C.
//             - Activa evento de petición.
//             - Define los bumpers como entradas.
//             - Pone en alto la resistencia de Pull-Up de cada bumper.
// Entradas: Ninguna.
// Salidas: Ninguna.
//*****************************************************************//
void setup()
{
  // Comunicación I2C
  Wire.begin(2);                // Dirección 2
  Wire.onRequest(requestEvent); // Activación evento de petición

  // Entradas
  for(a=0 ; a<2 ; a++)
  {
    pinMode(bumper[a], INPUT);
  }
  // Resistencia de pull-up
  for(b=0 ; b<2 ; b++)
  {
    digitalWrite(bumper[b], HIGH);
  }
```

```
}

//*************************************************************************//
// Nombre de la función: loop()
// Función: Función principal. Actúa como bucle, es decir, se ejecuta
//          cíclicamente.
// Entradas: Ninguna.
// Salidas: Ninguna.
//*************************************************************************//
void loop()
{
  // Lectura sonars
  Lectura_sonar();

  // Lectura bumpers
  Lectura_bumper();

  delay(10);
}

//*************************************************************************//
// Nombre de la función: Lectura_sonar()
// Llamada por: loop()
// Función: Encargada de leer la lectura de todos los ultrasonidos
//          presentes en el robot.
// Entradas: Ninguna.
// Salidas: Imprime por pantalla la distancia del obstáculo más próximo
//          al sensor junto al identificardor del mismo.
//*************************************************************************//
void Lectura_sonar()
{
  e = 0;

  for(c=0 ; c<8 ; c++)
  {
    // Lectura del sonar
    pinMode(sonar[c], OUTPUT);
    digitalWrite(sonar[c], LOW);
    delayMicroseconds(2);
    digitalWrite(sonar[c], HIGH);
    delayMicroseconds(5);
    digitalWrite(sonar[c], LOW);
    pinMode(sonar[c], INPUT);
    duracion_sonar[c] = pulseIn(sonar[c], HIGH);
   long_sonar[c] = microsecondsToCentimeters(duracion_sonar[c]);
    delayMicroseconds(2);

    // Almacenamiento en vector a enviar
   data[e] = (uint8_t)long_sonar[c];
```

```
    e = e+1;
  }
}

//*****************************************************************//
// Nombre de la función: Lectura_bumper()
// Llamada por: loop()
// Función: Encargada de leer el estado de cada bumper y por tanto, de
//          indicar si el mismo está o no pulsado.
// Entradas: Ninguna.
// Salidas: Imprime por pantalla si está o no pulsado cada bumper.
//*****************************************************************//
void Lectura_bumper()
{
  for(d=0 ; d<2 ; d++)
  {
    // Lectura estado bumper
    info_bumper[d] = digitalRead(bumper[d]);
    // Bumper no pulsado
    if (info_bumper[d] == HIGH)
      push_bumper[d] = 0;
    // Bumper pulsado
    else
      push_bumper[d] = 1;

    // Almacenamiento en vector a enviar
   data[e] = (uint8_t)push_bumper[d];

    e = e+1;
  }
}

//*****************************************************************//
// Nombre de la función: microsecondsToCentimeters()
// Llamada por: Lectura_sonar()
// Función: Encargada de traducir el tiempo de eco en cm.
// Entradas: El tiempo de eco.
// Salidas: La distancia del objeto en cm.
//*****************************************************************//
long microsecondsToCentimeters(long microseconds)
{
  return microseconds / 29 / 2;
}

//*****************************************************************//
// Nombre de la función: requestEvent()
// Llamada por: Arduino Mega
// Función: Encargada de envíar el vector de datos al Arduino Mega
```

```
//          cuando este se lo pida, mediante comunicación I2C.
// Entradas: Ninguna.
// Salidas: El vector de datos por comunicación I2C.
//*************************************************************//
void requestEvent()
{
  Wire.write(data, (size_t)10);
}
```

```
//*****************************************************************//
//              ANEXO 10. CODIGO ARDUINO MEGA ESCLAVO              //
//*****************************************************************//

//********************* DECLARACIÓN DE VARIABLES **********************//
#define VelMax 255       // Máxima velocidad permitida

// Motor
const int MotorDIR = 9; // Pin DIR del puente H
const int MotorPWM = 10; // Pin PWM del puente H
const int enable = 11;   // Habilitación del motor
// Encoder
const int CanalB = 3; // PIN del canal B del encoder
const int CanalA = 2; // PIN del canal A del encoder
volatile int Contador = 0; // Contador de pulsos del encoder
int LastContador = 0;      // Contador de pulsos del encoder
volatile unsigned int EstadoCanalB = 0;  // Estado del canal B
volatile unsigned int EstadoCanalA = 0;  // Estado del canal A
// Velocidad
double W_ref = 0;  // Velocidad angular de referencia
double W_act = 0;  // Velocidad que medimos en el motor
int dp = 0;        // Pulsos medidos del encoder
double Velocidad_motor = 0; // Velocidad que introducimos al motor
// Periodos de comunicación
unsigned long  Pmuestreo = 100; // Periodo de muestreo
// Variables temporales
unsigned long LastTime;     // Tiempo anterior
unsigned long now;          // Tiempo actual
unsigned long dT;           // Diferencia de tiempos
//************ Errores PID
// Parámetros PID
double Kp,Ki,Kd;
// Derecho
double Eactual, Eproporcional;  // Error proporcional
double Eint=0, Eintegral;       // Error integral
double Eder, Ederivativo;       // Error derivativo
doubleError,LastError=0;        // Error total

//*************** LECTURA PUERTO SERIE ***************//
byte Lectura_puerto;
int Ref = 0, Ref_W = 0;
int Indicador_ref;
boolean mySwitch = true;  // Switch para saber el valor inicial del conjunto
int Negativo_W = 1;

//*****************************************************************//
// Nombre de la función: setup()
// Función: Función principal de inicialización de variables.
//         Se ejecuta una sóla vez. En ella se realiza lo siguiente:
```

```
//             - Inicia la comunicación puerto serie.
//             - Define los pines PWM y de dirección como salidas.
//             - Define los canales del encoder como entradas.
//             - Pone los canales del encoder a HIGH.
//             - Define el pin de habilitación como entrada.
//             - Pone el pin de habilitación en HIGH.
//             - Activa la interrupción del encoder.
//             - Toma el tiempo actual.
//             - Establece una velocidad nula para el motor.
//             - Define los parámetros del PID.
// Entradas: Ninguna.
// Salidas: Ninguna.
//*********************************************************************//
void setup() {
  Serial.begin(115200);

  // Salidas
  pinMode(MotorPWM, OUTPUT);
  pinMode(MotorDIR, OUTPUT);

  // Entradas
  pinMode(CanalB, INPUT);
  digitalWrite(CanalB, HIGH);
  pinMode(CanalA, INPUT);
  digitalWrite(CanalA, HIGH);
  pinMode(enable, INPUT);

  // Habilitación de motor
  digitalWrite(enable,HIGH);

  // Interrupción.
  //Mega:(interr,pin)[(0,2),(1,3),(2,21),(3,20),(4,19),(5,18)]
  attachInterrupt(0,Contar,CHANGE);

  // Tiempo actual
  LastTime = millis();

  // Velocidad inicial nula
  digitalWrite(MotorDIR,LOW);
  analogWrite(MotorPWM, 0);

  //Definición de los parámetros del PID
  Kp = 150;
  Ki = 0.02;
  Kd = 0;
}

void loop()
{
```

```
//******* COMUNICACIÓN ARDUINO MEGA MAESTRO ********//
// NOTA PRÁCTICA.
   // Protocolo de comunicación: [4,W_nueva_izq]
   // En ASCII, los diferentes símbolos tienen una númeración:
   // , =  44 / [ = 91 / ] = 93  / Números entre 49 y 57 / - = 45

if(Serial.available()>0)
 {
   Lectura_puerto = Serial.read();

   // Si recibimos un número o un signo "-" (Número negativo)
   if((Lectura_puerto>47 && Lectura_puerto<58)||Lectura_puerto==45)
   { // Inicialmente mySwitch down
     if(!mySwitch)
      { // Lectura del número de referencia de comunicación
          Ref = (Ref*10)+(Lectura_puerto-48);
      }
      else
      { // Lectura del valor de la velocidad de referencia
        // Se trata de un número negativo
        if(Lectura_puerto==45)
        {
          Ref_W = 0;
          Negativo_W = -1;
        }
        // Se trata de un número positivo
        else
         Ref_W = (Ref_W*10)+(Lectura_puerto-48);
      }
   }

   // FIN DE COMUNICACIÓN. Almacenar valores.
   // El primer valor debe ser un 4.
   if(Lectura_puerto==93)
   {
     Indicador_ref = Ref;
     if(Indicador_ref == 4)
       W_ref = (double)Negativo_W*Ref_W/10;

     // Actualización de variables
     Negativo_W = 1;
     Indicador_ref = 0;
     Ref = 0;
     Ref_W = 0;
     mySwitch = false;
   }
   // Lectura del signo ",". Se debe leer un nuevo dígito
   else if (Lectura_puerto==44)
```

```
      mySwitch = true;

    // INICIO DE COMUNICACIÓN, mySwitch down
    else if (Lectura_puerto==91)
      mySwitch = false;
  }
  // Mandar datos al Arduino Mega Maestro
  else
  {
    int W_mandar = (int)(W_act*100);
    int dp_mandar = (int)dp;

    Serial.print("[");
    Serial.print(6);
    Serial.print(",");
    Serial.print(dp_mandar);
    Serial.print(",");
    Serial.print(W_mandar);
    Serial.print("]");
    delay(100);
    Serial.flush();
  }

  // Estado del motor
  Compute();              // Cálculo del PID
 Girar(Velocidad_motor); // Movimiento del motor
  delay(10);
}

//*************************************************************************//
// Nombre de la función: Contar()
// Llamada por: la interrupción número 0.
// Función: Encargada de leer los canales del encoder y en función de
//          lo leído, se actualizan los pulsos generados.
// Entradas: Ninguna.
// Salidas: Ninguna.
//*************************************************************************//
void Contar()
{
  EstadoCanalB = digitalRead(CanalB);
  EstadoCanalA = digitalRead(CanalA);

  if(EstadoCanalA == HIGH)
  {
    if(EstadoCanalB == LOW)
      Contador = Contador+1;
    else
      Contador = Contador-1;
  }
```

```
  else
  {
    if(EstadoCanalB == LOW)
      Contador = Contador-1;
    else
      Contador = Contador+1;
  }
}

//*****************************************************************//
// Nombre de la función: Girar()
// Llamada por: loop()
// Función: Encargada de girar el motor.
// Entradas: Velocidad a la que debe girar el motor.
// Salidas: Movimiento del motor.
//*****************************************************************//
void Girar(double Velocidad_motor)
{
  if(Velocidad_motor==0)
  {
    digitalWrite(MotorDIR,LOW);
    analogWrite(MotorPWM, 0);
  }
  if(Velocidad_motor>0)
  {
    digitalWrite(MotorDIR,LOW);
    analogWrite(MotorPWM, (int)Velocidad_motor);
  }
  if(Velocidad_motor<0)
  {
    digitalWrite(MotorDIR,HIGH);
    analogWrite(MotorPWM, (int)Velocidad_motor);
  }
}

//*****************************************************************//
// Nombre de la función: Compute()
// Llamada por: loop()
// Función: Encargada de calcular el error PID del motor y de
//          mantener la velocidad estable.
// Entradas: Ninguna.
// Salidas: Error PID y velocidad que debe ser introducida al motor.
//*****************************************************************//
void Compute()
{
  now = millis();

  if(now>LastTime+Pmuestreo)
  {
```

```
 // Diferencial de tiempo
 dT = now-LastTime;
 // Diferencial de pulsos
 dp = (Contador-LastContador);
 // Velocidad angular medida
 W_act = (double)dp/dT;

 // Actualización de valores
 LastContador = Contador;
 LastTime = now;
}

if(W_act==0 && W_ref!=0)
  Eint = 0;

if(W_ref==0 && W_ref>=-0.2)
{
  digitalWrite(MotorDIR,LOW);
  analogWrite(MotorPWM, 0);
}

// CÁLCULO DEL ERROR PID
// Error proporcional
Eactual = W_ref-W_act;
Eproporcional = Eactual*Kp;

// Error integral
if(Error >= VelMax && Error > 0)
  Error = VelMax;
else
  Eint += Eactual*dT;

if(Error <= VelMax && Error < 0)
  Error = -VelMax;
else
  Eint += Eactual*dT;

Eintegral = Eint*Ki;

// Error derivativo
Eder = (Eactual-LastError)/dT;
Ederivativo = Eder*Kd;

// Error total
Error = Eproporcional+Eintegral+Ederivativo;

// Cálculo de la velocidad
Velocidad_motor = Error;
// Suponemos una velocidad máxima y mínima, por si se supera
```

```
    Velocidad_motor = constrain(Velocidad_motor,-VelMax,VelMax);

    // Actualización de valores
    LastError = Eactual;
}
```

```
//*****************************************************************//
//              ANEXO 11. CODIGO ARDUINO MEGA MAESTRO              //
//*****************************************************************//

//********************* DECLARACIÓN DE VARIABLES **********************//
// Comunicación entre Arduino Mega maestro - Arduino Uno
#include <Wire.h>
#define VelMax 255       // Máxima velocidad permitida

//*********** BATERÍA *********//
// Variables digitales
int Led_Verde = 5;
int Led_Amarillo = 6;
int Led_Rojo = 7;
// Variable analógica
int Bateria = A2;
// Otras variables utiles
float Lectura_bat; // Valor donde se almacena el valor de la bateria
float Valor_bat;   // Valor convertido de la lectura
int Alarma_baja = 0;       // Alarma = 0 - Alarma inactiva.
                           // Alarma = 1 - Alarma activada.
int Alarma_critica = 0;    // Alarma = 0 - Alarma inactiva.
                           // Alarma = 1 - Alarma activada.
int Alarma_baja_ant = 0;
int Alarma_critica_ant = 0;
int Num_baterias = 1;
//*********** SONARS Y BUMPERS *********//
int a = 1, b, c; // Parámetros para bucles
int Sonars[8];
int Bumpers[2];
int Num_bumpers = 2;
int Num_sonars = 8;
// Motor
int Num_motores = 2;
const int MotorDIR = 9; // Pin DIR del puente H
const int MotorPWM = 10; // Pin PWM del puente H
const int enable = 11;   // Habilitación del motor
// Encoder
const int CanalB = 3; // PIN del canal B del encoder
const int CanalA = 2; // PIN del canal A del encoder
volatile int Contador = 0; // Contador de pulsos del encoder
int LastContador = 0;      // Contador de pulsos del encoder
volatile unsigned int EstadoCanalB = 0;  // Estado del canal B
volatile unsigned int EstadoCanalA = 0;  // Estado del canal A
// Velocidad
double W_ref = 0;  // Velocidad angular de referencia
double W_act = 0;  // Velocidad que medimos en el motor
int dp = 0;        // Pulsos medidos del encoder
double Velocidad_motor = 0; // Velocidad que introducimos al motor
```

```
// Periodos de comunicación
unsigned long  Pmuestreo = 100; // Periodo de muestreo
unsigned long P_2000 = 2000;    // Periodo de estado 2000 ms
unsigned long P_100 = 300;      // Periodo de estado 100 ms
// Variables temporales
unsigned long LastTime;     // Tiempo anterior
unsigned long now;          // Tiempo actual
unsigned long dT;           // Diferencia de tiempos
unsigned long T_actual;        // Tiempo actual
unsigned long T100_anterior;   // Tiempo anterior para periodo 100 ms
unsigned long T2000_anterior;  // Tiempo anterior para periodo 2000 ms
//************ Errores PID
// Parámetros PID
double Kp,Ki,Kd;
// Derecho
double Eactual, Eproporcional;  // Error proporcional
double Eint=0, Eintegral;       // Error integral
double Eder, Ederivativo;       // Error derivativo
double Error,LastError=0;       // Error total

//*************** LECTURA PUERTO SERIE ***************//
byte Lectura_puerto;      // Almacenamos el valor recibido por el puerto serie
int Ref=0, Ref_der=0, Ref_izq=0; // Útiles para saber que valor analizar
int Num_ref = 1;          // Número de ref del conjunto
int Indicador_ref;        // Número de ref del conjunto
boolean mySwitch = true;  // Switch para saber el valor inicial del conjunto
int Cambio_ref = 0;       // Para cambiar de dígito a leer en el conjunto
int temporizador = 0;     // Temp para saber si el recibimos velocidad de ref
int temporizador1 = 0;    // Temp para saber si el recibimos velocidad de ref
int Lectura = 0;          // Para saber si recibimos velocidad de ref
int Negativo_der = 1;     // Signo de la velocidad de referencia del motor der
int Negativo_izq = 1;     // Signo de la velocidad de referencia del motor izq
int W_ref_izq = 0;        // Valor de la velocidad a enviar al motor izquierdo
int W_ref_der = 0;        // Valor de la velocidad a enviar al motor derecho
//************ LECTURA ARDUINO MEGA ESCLAVO ************//
byte Lectura_puerto1;     // Almacenamos el valor recibido por el puerto serie
int Ref_mega=0, Ref_dp=0, Ref_W=0; // Útiles para saber que valor analizar
int Indicador_ref_mega;   // Número de ref del conjunto
int Num_ref_mega = 1;     // Número de ref del conjunto
boolean mySwitch_puerto = true;  // Switch para saber el valor inicial del con
int Cambio_ref_mega = 0;  // Para cambiar de dígito a leer en el conjunto
int Negativo_dp = 1;      // Signo de la velocidad de referencia del motor der
int Negativo_W = 1;       // Signo de la velocidad de referencia del motor izq
int dp_izq = 0;           // Diferencial de pulsos del motor izquierdo
int W_act_izq = 0;        // Velocidad del motor izquierdo
double W_act_der = 0;     // Velocidad del motor derecho
int dp_der = 0;           // Diferencial de pulsos del motor derecho
int Lectura_mega_1 = 1;   // Para saber si el recibimos datos
```

```
//*****************************************************************//
// Nombre de la función: setup()
// Función: Función principal de inicialización de variables.
//          Se ejecuta una sóla vez. En ella se realiza lo siguiente:
//             - Inicia la comunicación puerto serie.
//             - Inicia la comunicación I2C.
//             - Guarda tiempos actuales.
//             - Define los leds como salidas.
//             - Define los pines PWM y de dirección como salidas.
//             - Define los canales del encoder como entradas.
//             - Define el pin de habilitación como entrada.
//             - Apaga todos los leds.
//             - Pone los canales del encoder a HIGH.
//             - Pone el pin de habilitación a HIGH.
//             - Activa la interrupción del encoder.
//             - Almancena el tiempo actual.
//             - Establece una velocidad nula de motor.
//             - Define los parámetros del PID.
// Entradas: Ninguna.
// Salidas: Ninguna.
//*****************************************************************//
void setup() {
  // Inicialización del bus I2C
  Wire.begin();

  Serial.begin(115200);
  Serial1.begin(115200);

  // Guardamos tiempo actual
  T100_anterior = millis();
  T2000_anterior = millis();

  // SALIDAS
  // Batería
  pinMode(Led_Verde, OUTPUT);
  pinMode(Led_Amarillo, OUTPUT);
  pinMode(Led_Rojo, OUTPUT);
  // Motor derecho
  pinMode(MotorPWM, OUTPUT);
  pinMode(MotorDIR, OUTPUT);

  // Entradas
  pinMode(CanalB, INPUT);
  digitalWrite(CanalB, HIGH);
  pinMode(CanalA, INPUT);
  digitalWrite(CanalA, HIGH);
  pinMode(enable, INPUT);

  // Inicialmente todas las luces apagadas
```

```
  digitalWrite(Led_Verde, LOW);
  digitalWrite(Led_Amarillo, LOW);
  digitalWrite(Led_Rojo, LOW);

  // Habilitación de motor
  digitalWrite(enable,HIGH);

  // Interrupción.
  //Mega:(interr,pin)[(0,2),(1,3),(2,21),(3,20),(4,19),(5,18)]
  attachInterrupt(0,Contar,CHANGE);

  // Tiempo actual
  LastTime = millis();
  temporizador = millis();

  // Velocidad inicial nula
  digitalWrite(MotorDIR,LOW);
  analogWrite(MotorPWM, 0);

  //Definición de los parámetros del PID
  Kp = 150;
  Ki = 0.02;
  Kd = 0;

  Lectura_mega_1 = 1;
}

void loop()
{
  T_actual = millis();

  //****** INICIO COMUNICACIÓN ARDUINO MEGA ESCLAVO ******//
  // NOTA PRÁCTICA.
  // Protocolo de comunicación: [6,dp,W_act]
  // En ASCII, los diferentes símbolos tienen una númeración:
  // , =  44 / [ = 91 / ] = 93  / Números entre 49 y 57 / - = 45

  if(Serial1.available()>0)
  {
    while(Lectura_mega_1 == 1)
    {
       Lectura_puerto1 = Serial1.read(); // Leemos del puerto serie

       // Si recibimos un número o un signo "-" (Número negativo)
       if((Lectura_puerto1>47 && Lectura_puerto1<58)|| Lectura_puerto1==45)
       { // Inicialmente mySwitch down
         if(!mySwitch_puerto)
           // Lectura del número de referencia de comunicación
          Ref_mega = (Ref_mega*10)+(Lectura_puerto1-48);
```

```
  else
  { // Lectura del valor de las velocidad del motor izquierdo
    if(Num_ref_mega==2)
    { // Se trata de un número negativo
      if(Lectura_puerto1==45)
      {
        Ref_W = 0;
        Negativo_W = -1;
      }
      // Se trata de un número positivo
      else
       Ref_W = (Ref_W*10)+(Lectura_puerto1-48);
    }
    // Lectura del número de pulsos del motor izquierdo
    else
    {  // Se trata de un número negativo
      if(Lectura_puerto1==45)
      {
        Ref_dp = 0;
        Negativo_dp = -1;
      }
      // Se trata de un número positivo
      else
       Ref_dp = (Ref_dp*10)+(Lectura_puerto1-48);
    }
  }
}

// FIN DE COMUNICACIÓN. Almacenar valores.
// El primer valor debe ser un 6.
if(Lectura_puerto1==93)
{
 Indicador_ref_mega = Ref_mega;
  if(Indicador_ref_mega==6)
  {
    dp_izq = (double)Negativo_dp*Ref_dp;
    W_act_izq = (double)Negativo_W*Ref_W;
  }

  // Actualización de valores
  Negativo_dp = 1;
  Negativo_W = 1;
  Indicador_ref = 0;
  Ref_mega = 0;
  Ref_dp = 0;
  Ref_W = 0;
  mySwitch_puerto = false;
  Num_ref_mega = 1;
```

```
        Lectura_mega_1 = 0;
      }
      // Lectura del signo ",". Se debe leer un nuevo dígito
      else if (Lectura_puerto1==44)
      { // mySwitch up y sumamos número de referencia
        mySwitch_puerto = true;
        Num_ref_mega = Num_ref_mega+1;
      }
      // INICIO DE COMUNICACIÓN, mySwitch down
      else if (Lectura_puerto1==91)
        mySwitch_puerto = false;
  }
}
//****** FIN COMUNICACIÓN ARDUINO MEGA ESCLAVO ******//

//********** LECTURA CONTROL DE ALTO NIVEL **********//
 // NOTA PRÁCTICA.
// Protocolo de comunicación: $REF[3,vel_der,vel_izq]
// En ASCII, los diferentes símbolos tienen una númeración:
// , =  44 / [ = 91 / ] = 93  / Números entre 49 y 57 / - = 45

// Lectura de tramas de referencia de movimiento recibidas por el puerto ser
temporizador = millis();

if(Serial.available()>0)
{
  temporizador1 = millis();
  Lectura = 1;
  Cambio_ref = 1;

  Lectura_puerto = Serial.read(); // Leemos del puerto serie

  // Si recibimos un número o un signo "-" (Número negativo)
  if((Lectura_puerto>47 && Lectura_puerto<58)|| Lectura_puerto==45)
  {  // Inicialmente mySwitch down
     if(!mySwitch)
       // Lectura del número de referencia de comunicación
      Ref = (Ref*10)+(Lectura_puerto-48);
     else
     { // Lectura del valor de las velocidades de referencia
       if(Num_ref==2)
       { // Se trata de un número negativo
         if(Lectura_puerto==45)
         {
           Ref_der = 0;
           Negativo_der = -1;
         }
         // Se trata de un número positivo
         else
```

```
          Ref_der = (Ref_der*10)+(Lectura_puerto-48);
       }
       else
       { // Se trata de un número negativo
         if(Lectura_puerto==45)
         {
           Ref_izq = 0;
           Negativo_izq = -1;
         }
         // Se trata de un número positivo
         else
          Ref_izq = (Ref_izq*10)+(Lectura_puerto-48);
       }
    }
}

// FIN DE COMUNICACIÓN. Almacenar valores.
// El primer valor debe ser un 3.
if(Lectura_puerto==93)
{
  Indicador_ref = Ref;
  if(Indicador_ref==3)
  {
    W_ref = (double)Negativo_der*Ref_der/10000;
    W_ref_izq = (double)Negativo_izq*Ref_izq/1000;

    // Mandar al Arduino Mega esclavo
    Serial1.print("[");
    Serial1.print(4);
    Serial1.print(",");
    Serial1.print(W_ref_izq);
    Serial1.print("]");
    delay(100);
    Serial1.flush();

    // Actualización de valores
    Negativo_der = 1;
    Negativo_izq = 1;
  }

  // Actualización de variables
  Indicador_ref = 0;
  Ref = 0;
  Ref_der = 0;
  Ref_izq = 0;
  mySwitch = true;
  Num_ref = 1;
}
// Lectura del signo ",". Se debe leer un nuevo dígito
```

```
    else if (Lectura_puerto==44)
    { // mySwitch up y sumamos número de referencia
      mySwitch = true;
      Num_ref = Num_ref+1;
    }
    // INICIO DE COMUNICACIÓN, mySwitch down
    else if (Lectura_puerto==91)
      mySwitch = false;

    Serial.flush();
  }

  // Sólo se deben mover motores si se le indica velocidad
  // por puerto serie.
  if(temporizador-temporizador1 > 2000)
    Lectura = 0;
  if(Lectura == 0)
  {
    W_ref = 0;
    W_ref_izq = 0;
  }
  //****** FIN COMUNICACIÓN CONTROL DE ALTO NIVEL ******//

  // Estado del motor
  Compute();              // Cálculo del PID
 Girar(Velocidad_motor); // Movimiento del motor
  delay(10);

  // Estado de la batería
 Lectura_bateria();
 Control_estado();

  // Estado de bumpers y sonars
 Lectura_sonars_bumpers();
  Wire.endTransmission(2);

  //******* MANDAR AL CONTROL DE ALTO NIVEL ********//
  // Cada 100 ms:
  if(T_actual>T100_anterior+P_100)
  {
    // Cada 2000 ms o alarma:
    if((T_actual>T2000_anterior+P_2000)||(Alarma_baja_ant-Alarma_baja!=0)||
    {
      if((T_actual>T2000_anterior+P_2000))
      { // Estado de las baterias
        Serial.print("$VLT:[");
        Serial.print(Num_baterias);
        Serial.print(",");
        Serial.print(Valor_bat);
```

```
      Serial.print("]#\r\n");

      T2000_anterior = T_actual;
    }

    Serial.print("$AL:[");
    // Alarma bateria baja
    Serial.print(Alarma_baja);
    Serial.print(",");
    // Alarma bateria crítica
    Serial.print(Alarma_critica);
    Serial.print("]#\r\n");

  }

 Alarma_baja_ant = Alarma_baja;
 Alarma_critica_ant = Alarma_critica_ant;

  // Estado sonars
  Serial.print("$SON:[");
  Serial.print(Num_sonars);
  for(b=0 ; b<8 ; b++)
  {
     Serial.print(",");
     Serial.print(Sonars[b]);
  }
  Serial.print("]#\r\n");

  // Estado bumpers
  Serial.print("$BMP:[");
  Serial.print(Num_bumpers);
  for(c=0 ; c<2 ; c++)
  {
     Serial.print(",");
     Serial.print(Bumpers[c]);
  }
   Serial.print("]#\r\n");

   int W_der_mandar =  (int)(W_act_der*10000);
   int W_izq_mandar =  (int)(W_act_izq*10000);
   int dp_der_mandar = (int)dp_der*100;
   int dp_izq_mandar = (int)dp_izq*100;

  // Estado de motores (velocidad y odometría)
   Serial.print("$MOT:[");
   Serial.print(Num_motores);
   Serial.print(",");
  // Motor derecho
  Serial.print(W_der_mandar);
```

```
    Serial.print(",");
    Serial.print(dp_der_mandar);
    Serial.print(",");
    // Motor izquierdo
    Serial.print(W_izq_mandar);
    Serial.print(",");
    Serial.print(dp_izq_mandar);
    Serial.print("]#\r\n");

    T100_anterior = T_actual;
  }
  delay(10);
  //****** FIN COMUNICACIÓN CONTROL DE ALTO NIVEL ******//
  Lectura_mega_1 = 1;
}

//*****************************************************************//
// Nombre de la función: Lectura_bateria()
// Llamada por: loop()
// Función: Encargada de leer el pin analógico y convertir la lectura
//          en el valor real de la batería (entre 0 y 12.5V).
// Entradas: Ninguna.
// Salidas: Ninguna
//*****************************************************************//
void Lectura_bateria()
{
  Lectura_bat = analogRead(Bateria);
  Valor_bat = (Lectura_bat*12.5)/1023;
}

//*****************************************************************//
// Nombre de la función: Control_estado()
// Llamada por: loop()
// Función: Encargada de actuar en función de la carga de la bateria:
//          - Bateria cargada (12.5 V,11.3 V): Led verde encendido.
//                                             No alarmas.
//          - Bateria media (11.3 V,10.8 V): Led amarillo encendido.
//                                           Alarma bateria baja.
//          - Bateria baja (10.8 V,0 V): Led rojo encendido.
//                                       Alarma bateria crítica.
// Entradas: Ninguna.
// Salidas: Ninguna
//*****************************************************************//
void Control_estado()
{
  // Bateria cargada
  if(Valor_bat>11.3)
  {
    digitalWrite(Led_Verde, HIGH);
```

```
    digitalWrite(Led_Amarillo, LOW);
    digitalWrite(Led_Rojo, LOW);
    Alarma_baja = 0;
    Alarma_critica = 0;
  }
  // Bateria con carga media
  if(Valor_bat<=11.3 && Valor_bat>10.8)
  {
    digitalWrite(Led_Verde, LOW);
    digitalWrite(Led_Amarillo, HIGH);
    digitalWrite(Led_Rojo, LOW);
    Alarma_baja = 1;
    Alarma_critica = 0;
  }
  // Bateria con carga crítica
  if(Valor_bat<=10.8)
  {
    digitalWrite(Led_Verde, LOW);
    digitalWrite(Led_Amarillo, LOW);
    digitalWrite(Led_Rojo, HIGH);
    Alarma_baja = 0;
    Alarma_critica = 1;
  }
}

//*****************************************************************//
// Nombre de la función: Lectura_sonars_bumpers()
// Llamada por: loop()
// Función: Encargada de leer la lectura de todos los ultrasonidos y
//          bumpers presentes en el robot por el puerto I2C.
// Entradas: Ninguna.
// Salidas: Ninguna.
//*****************************************************************//
void Lectura_sonars_bumpers()
{
  Wire.requestFrom(2, 10); // Petición de 10 datos por la dirección 2
  while(Wire.available())
  {
    // Inicialización del bucle
    if(a>10)
      a=1;

    // Almacenamiento de dato leido
    int recibe_datos = Wire.read();

    // 8 primeros dígitos - Datos sonars
    if(a<=8)
      Sonars[a-1] = recibe_datos;
```

```
    // 2 últimos dígitos - Datos bumpers
    if(a>8)
      Bumpers[a-9] = recibe_datos;
    a=a+1;
  }
}

//*****************************************************************//
// Nombre de la función: Contar()
// Llamada por: la interrupción número 0.
// Función: Encargada de leer los canales del encoder y en función de
//          lo leído, se actualizan los pulsos generados.
// Entradas: Ninguna.
// Salidas: Ninguna.
//*****************************************************************//
void Contar()
{
  EstadoCanalB = digitalRead(CanalB);
  EstadoCanalA = digitalRead(CanalA);

  if(EstadoCanalA == HIGH)
  {
    if(EstadoCanalB == LOW)
      Contador = Contador+1;
    else
      Contador = Contador-1;
  }
  else
  {
    if(EstadoCanalB == LOW)
      Contador = Contador-1;
    else
      Contador = Contador+1;
  }
}

//*****************************************************************//
// Nombre de la función: Girar()
// Llamada por: loop()
// Función: Encargada de girar el motor.
// Entradas: Velocidad a la que debe girar el motor.
// Salidas: Movimiento del motor.
//*****************************************************************//
void Girar(double Velocidad_motor)
{
  if(Velocidad_motor==0)
  {
    digitalWrite(MotorDIR,LOW);
    analogWrite(MotorPWM, 0);
```

```
  }
  if(Velocidad_motor>0)
  {
    digitalWrite(MotorDIR,LOW);
    analogWrite(MotorPWM, (int)Velocidad_motor);
  }
  if(Velocidad_motor<0)
  {
    digitalWrite(MotorDIR,HIGH);
    analogWrite(MotorPWM, (int)Velocidad_motor);
  }
}

//*********************************************************************//
// Nombre de la función: Compute()
// Llamada por: loop()
// Función: Encargada de calcular el error PID del motor y de
//          mantener la velocidad estable.
// Entradas: Ninguna.
// Salidas: Error PID y velocidad que debe ser introducida al motor.
//*********************************************************************//
void Compute()
{
  now = millis();

  if(now>LastTime+Pmuestreo)
  {
    // Diferencial de tiempo
    dT = now-LastTime;
    // Diferencial de pulsos
    dp = (Contador-LastContador);
    // Velocidad angular medida
    W_act = (double)dp/dT;

    // Para mandar al control de alto nivel
    W_act_der = W_act;
    dp_der = dp;

    // Actualización de valores
    LastContador = Contador;
    LastTime = now;
   }

   if(W_act==0 && W_ref!=0)
     Eint = 0;

   if(W_ref==0 && W_ref>=-0.2)
   {
     digitalWrite(MotorDIR,LOW);
```

```
    analogWrite(MotorPWM, 0);
  }


  // CÁLCULO DEL ERROR PID
  // Error proporcional
  Eactual = W_ref-W_act;
  Eproporcional = Eactual*Kp;

  // Error integral
  if(Error >= VelMax && Error > 0)
    Error = VelMax;
  else
    Eint += Eactual*dT;

  if(Error <= VelMax && Error < 0)
    Error = -VelMax;
  else
    Eint += Eactual*dT;

  Eintegral = Eint*Ki;

  // Error derivativo
  Eder = (Eactual-LastError)/dT;
  Ederivativo = Eder*Kd;

  // Error total
  Error = Eproporcional+Eintegral+Ederivativo;

  // Cálculo de la velocidad
  Velocidad_motor = Error;
  // Suponemos una velocidad máxima y mínima, por si se supera
  Velocidad_motor = constrain(Velocidad_motor,-VelMax,VelMax);

  // Actualización de valores
  LastError = Eactual;
}
```

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%               ANEXO 12. REPRESENTACIÓN PID EN MATLAB                 %
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Nombre de la función: PID
% Función: Representación de los valores de la velocidad angular y de
%          los errores proporcional e integral.
% Entradas: Los valores de la velocidad angular y los errores por el
%           puerto COM3.
% Salidas: Representación gráfica.
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
function  PID
    % Puesta a punto del programa
    close  all;
    clc;

    % Inicialización del puerto serial
    delete (instrfind ({'port' },{'COM3' }));
    puerto_serial  = serial ('COM3' );
    puerto_serial .BaudRate  = 9600 ;
    warning ('off' ,'MATLAB_serial:fscanf:unsuccessfulRead');

    % Abro el puerto serial
    fopen(puerto_serial );

    % Declaro un contador del número de muestras ya tomadas
    contador = 1;
    % Creación de vectores para guardar datos
    y = zeros(1,100000000);
    z = zeros(1,100000000);
    a = zeros(1,100000000);

    % Creo dos ventanas para las gráficas
    h = figure('Name','PID');
    title('PID');
    xlabel('Tiempo');
    ylabel('Velocidad');
    ylim([0 1]);
    x = 1:1:400;
    q = figure('Name','referencia');

    % Bucle para que tome y dibuje las muestras
    while (contador <= 400)
        W_act_izq = fscanf(puerto_serial , '%g');
        Eproporcional_izq  = fscanf(puerto_serial , '%g');
        Eintegral_izq = fscanf(puerto_serial , '%g');

        y(contador) = (W_act_izq(1));
        a(contador) = Eproporcional_izq(1);
        z(contador) = Eintegral_izq(1);

        figure(h);
        plot(x,y(x));
        drawnow

        figure(q);
        plot(x,z(x),'g',x,a(x),'b');
        drawnow

        contador = contador+1;
    end

    % Cierro la conexión con el puerto serial y elimino las variables
    fclose(puerto_serial );
    delete (puerto_serial );
    clear all;
end
```

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%              ANEXO 13. PROGRAMA DE COMPROBACIÓN. MATLAB                    %
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Nombre de la función: Prueba
% Función: Comprobación de todos los sensores y actuadores
%                  del robot Edubot.
% Entradas: Los valores dados por los sensores y actuadores por el
%                    puerto COM3.
% Salidas: Representación gráfica de los resultados.
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
function  Prueba
    % Puesta a punto del programa
    close  all;
    clc;

     % Inicialización del puerto serial
    delete(instrfind({'port'},{'COM3'}));
    puerto_serial  = serial('COM3');
    puerto_serial.BaudRate  = 9600;
    warning('off','MATLAB_serial:fscanf:unsuccessfulRead');

     % Abro el puerto serial
    fopen(puerto_serial );

    % Declaración de variables
    b = 1;          % Bucle menú
    stb = 0;        % Número de bumpers y sonars detectados

    % Menú
    while b==1
        realizar_comp = input('¿Quiere realizar alguna comprobación antes de comenzar a
usar el robot? (S/N): ','s');
        if(realizar_comp == 'S' || realizar_comp == 's')
            disp('¿Que quiere comprobar?:');
            disp(' - Encoder derecho (Pulse 1)');
            disp(' - Encoder izquierdo (Pulse 2)');
            disp(' - Bumpers y Sonars(Pulse 3)');
            disp(' - Visualización del PID derecho, hacia delante (Pulse 4)');
            disp(' - Visualización del PID derecho, hacia atrás (Pulse 5)');
            disp(' - Visualización del PID izquierdo, hacia delante (Pulse 6)');
            disp(' - Visualización del PID izquierdo, hacia atrás (Pulse 7)');
            tipo_comp = input('Indique el tipo de comprobación:    ');

            comp_matlab = num2str(tipo_comp);
            fprintf(puerto_serial ,'%s',comp_matlab);

            switch tipo_comp
                case 1
                    disp('Comprobando encoder derecho...');
                case 2
                    disp('Comprobando encoder izquierdo...');
                case 3
                    disp('Se comprobará cada sonar y cada bumper. Acercar la mano a
cada sonar o pulsar el bumper.');
                    while(stb <= '8')
                        disp(fscanf(puerto_serial ,'%s'))
                        disp(fscanf(puerto_serial ,'%s'));
                        stb = fscanf(puerto_serial ,'%s');
                    end
                case {4,5,6,7}
                       % Creación de vectores para guardar datos
                        y = zeros(1,1000);
                        z = zeros(1,1000);
                        a = zeros(1,1000);
                        % Declaro un contador del número de muestras ya tomadas
                        contador = 1;
                        % Creo dos ventanas para las gráficas
                        h = figure('Name','PID');
                        title('PID');
                        xlabel('Tiempo');
```

```
                    ylabel('Velocidad');
                    ylim([0 1]);
                    x = 1:1:200;
                    q=figure('Name','referencia');

                    if (tipo_comp == 4 || tipo_comp == 5)
                        % Bucle para que tome y dibuje las muestras
                        while (contador<=400)
                            W_act_izq = fscanf(puerto_serial, '%g');
                            W_act_der = fscanf(puerto_serial, '%g');
                            Eproporcional_izq  = fscanf(puerto_serial, '%g');
                            Eproporcional_der  = fscanf(puerto_serial, '%g');
                            Eintegral_izq = fscanf(puerto_serial, '%g');
                            Eintegral_der = fscanf(puerto_serial, '%g');

                            y(contador) = (W_act_der(1));
                            a(contador) = Eproporcional_der(1);
                            z(contador) = Eintegral_der(1);

                            figure(h);
                            plot(x,y(x));
                            drawnow

                            figure(q);
                            plot(x,z(x),'g',x,a(x),'b');
                            drawnow

                            contador = contador+1;
                        end
                    end

                    if (tipo_comp == 6 || tipo_comp == 7)
                        % Bucle para que tome y dibuje las muestras
                        while (contador<=400)
                            W_act_izq = fscanf(puerto_serial, '%g');
                            W_act_der = fscanf(puerto_serial, '%g');
                            Eproporcional_izq  = fscanf(puerto_serial, '%g');
                            Eproporcional_der  = fscanf(puerto_serial, '%g');
                            Eintegral_izq = fscanf(puerto_serial, '%g');
                            Eintegral_der = fscanf(puerto_serial, '%g');

                            y(contador) = (W_act_izq(1));
                            a(contador) = Eproporcional_izq(1);
                            z(contador) = Eintegral_izq(1);

                            figure(h);
                            plot(x,y(x));
                            drawnow

                            figure(q);
                            plot(x,z(x),'g',x,a(x),'b');
                            drawnow

                            contador = contador+1;
                        end
                    end

                otherwise
                    disp('Opción introducida erronea.');
            end
        elseif(realizar_comp == 'N' || realizar_comp == 'n')
            disp('Comenzamos con el funcionamiento del robot');
            b = 2;
        else
            disp('Tecla introducida erronea.');
        end
    end


    % Cierro la conexión con el puerto serial y elimino las variables
    fclose(puerto_serial);
    delete(puerto_serial);
```

```
    clear all;
end
```

```
//*****************************************************************//
//          ANEXO 14. PROGRAMA DE COMPROBACIÓN. ARDUINO MEGA           //
//*****************************************************************//

//********************* DECLARACIÓN DE VARIABLES **********************//
// Comunicación entre arduinos
#include <Wire.h>
#define VelMax 255       // Máxima velocidad permitida
//************ Motores
// Derecho
const int MotorDIR_der = 4; // Pin DIR del puente H derecho
const int MotorPWM_der = 5; // Pin PWM del puente H derecho
const int enable_der = 6;   // Habilitación del motor derecho
// Izquierdo
const int MotorDIR_izq = 7; // Pin DIR del puente H izquierdo
const int MotorPWM_izq = 8; // Pin PWM del puente H izquierdo
const int enable_izq = 9;   // Habilitación del motor izquierdo
//************ Encoders
// Derecho
const int CanalB_der = 3; // PIN del canal B del encoder derecho
const int CanalA_der = 2; // PIN del canal A del encoder derecho
volatile int Contador_der = 0; // Contador de pulsos del encoder
int LastContador_der = 0;      // Contador de pulsos del encoder
volatile unsigned int EstadoCanalB_der = 0;  // Estado del canal B
volatile unsigned int EstadoCanalA_der = 0;  // Estado del canal A
// Izquierdo
const int CanalB_izq = 19; // PIN del canal B del encoder izquierdo
const int CanalA_izq = 18; // PIN del canal A del encoder izquierdo
volatile int Contador_izq = 0; // Contador de pulsos del encoder
int LastContador_izq = 0;      // Contador de pulsos del encoder
volatile unsigned int EstadoCanalB_izq = 0;  // Estado del canal B
volatile unsigned int EstadoCanalA_izq = 0;  // Estado del canal A
//************ Velocidades
// Motor izquierdo
double W_ref_izq = 0;  // Velocidad angular de referencia izquierda
double W_act_izq = 0;  // Velocidad que medimos en el motor izquierdo
int dp_izq = 0;        // Pulsos medidos del encoder izquierdo
double Velocidad_motor_izq = 0; // Velocidad que introducimos al motor izq
// Motor derecho
double W_ref_der = 0;  // Velocidad angular de referencia derecha
double W_act_der = 0;  // Velocidad que medimos en el motor derecho
int dp_der = 0;        // Pulsos medidos del encoder derrecho
double Velocidad_motor_der = 0; // Velocidad que introducimos al motor der
//************ Errores PID
// Parámetros PID
double Kp,Ki,Kd;
// Izquierdo
double Eactual_izq, Eproporcional_izq;  // Error proporcional
double Eint_izq=0, Eintegral_izq;       // Error integral
```

```
double Eder_izq, Ederivativo_izq;       // Error derivativo
doubleError_izq,LastError_izq=0;        // Error total
// Izquierdo
double Eactual_der, Eproporcional_der;  // Error proporcional
double Eint_der=0, Eintegral_der;       // Error integral
double Eder_der, Ederivativo_der;       // Error derivativo
doubleError_der,LastError_der=0;        // Error total
// Periodos de comunicación
unsigned long  Pmuestreo = 200; // Periodo de muestreo
// Variables temporales
unsigned long LastTime; // Tiempo anterior
unsigned long now;          // Tiempo actual
unsigned long dT;       // Diferencia de tiempos
//************ Sonars y bumpers
// Detección ultrasonido
int s1 = 0, s2 = 0, s3 = 0, s4 = 0, s5 = 0,s6 = 0, s7 = 0, s8 = 0;
int b1 = 0, b2 = 0;  // Detección bumpers
int stb = 0;         // Detección total de bumpers + ultrasonidos
int i=1;             // Parámetro para bucle
// Otras variables útiles
int Cambio = 0; // Indicación de cambio de sentido

//*****************************************************************//
// Nombre de la función: setup()
// Función: Función principal de inicialización de variables.
//         Se ejecuta una sóla vez. En ella se realiza lo siguiente:
//            - Inicia la comunicación puerto serie.
//            - Inicia la comunicación I2C.
//            - Define los pines PWM y de dirección como salidas.
//            - Define los canales de los encoders como entradas.
//            - Define los pines de habilitación como entradas.
//            - Pone los canales de los encoders a HIGH.
//            - Pone los pines de habilitación en HIGH.
//            - Activa la interrupción de los encoders.
//            - Almancena el tiempo actual.
//            - Establece una velocidad nula de motores.
//            - Define los parámetros del PID.
// Entradas: Ninguna.
// Salidas: Ninguna.
//*****************************************************************//
void setup() {
  // Inicialización del bus I2C
  Wire.begin();

  Serial.begin(9600);

  // Salidas
  // Motor derecho
  pinMode(MotorPWM_der, OUTPUT);
```

```
  pinMode(MotorDIR_der, OUTPUT);
  // Motor izquierdo
  pinMode(MotorPWM_izq, OUTPUT);
  pinMode(MotorDIR_izq, OUTPUT);

  // Entradas
  // Motor derecho
  pinMode(CanalB_der, INPUT);
  digitalWrite(CanalB_der, HIGH);
  pinMode(CanalA_der, INPUT);
  digitalWrite(CanalA_der, HIGH);
  pinMode(enable_der, INPUT);
  // Motor izquierdo
  pinMode(CanalB_izq, INPUT);
  digitalWrite(CanalB_izq, HIGH);
  pinMode(CanalA_izq, INPUT);
  digitalWrite(CanalA_izq, HIGH);
  pinMode(enable_izq, INPUT);

  // Habilitación de motores
  digitalWrite(enable_der,HIGH);
  digitalWrite(enable_izq,HIGH);

  // Interrupciones.
  //Mega:(interr,pin)[(0,2),(1,3),(2,21),(3,20),(4,19),(5,18)]
  attachInterrupt(4,Contar_izq,CHANGE);
  attachInterrupt(0,Contar_der,CHANGE);

  // Tiempo actual
  LastTime = millis();

  // Velocidad inicial nula en motores
  digitalWrite(MotorDIR_izq,LOW);
  analogWrite(MotorPWM_izq, 0);
  digitalWrite(MotorDIR_der,LOW);
  analogWrite(MotorPWM_der, 0);

  // Definición de los parámetros del PID
  Kp = 120;
  Ki = 0.01;
  Kd = 0;
}

//*************************************************************//
// Nombre de la función: loop()
// Función: Función principal. Actúa como bucle, es decir, se ejecuta
//          cíclicamente.
// Entradas: Ninguna.
// Salidas: Ninguna.
```

```
//*************************************************************************//
void loop() {
  if(Serial.available()>0)
  {
    char comando = Serial.read();
    // Comprobación encoder derecho
    if(comando =='1')
    {
      Cambio = 0;
     Comprobacion_motor_der();
    }
    // Comprobación encoder izquierdo
    else if(comando == '2')
    {
      Cambio = 0;
     Comprobacion_motor_izq();
    }
    // Comprobación bumpers y ultrasonidos
    else if(comando == '3')
    {
      while(stb<=9)
      {
        Wire.requestFrom(2, 10); // Petición de 10 datos por la direc 2
        while(Wire.available())
        {
          // Inicialización del bucle
          if(i>10)
            i=1;

          // Almacenamiento de dato leido
          byte recibe_datos = Wire.read();

          // 8 primeros dígitos - Datos ultrasonido
          if(i<=8)
          {
            if (recibe_datos <= 9)
            {
              if(i==1 && s1==0)
              {
                Serial.println("Sonar1 ");
                Serial.println("Detectado");
                Serial.println(stb);
                s1 = 1;
              }
              if(i==2 && s2==0)
              {
                Serial.println("Sonar2 ");
                Serial.println("Detectado");
                Serial.println(stb);
```

```
      s2 = 1;
    }
    if(i==3 && s3==0)
    {
      Serial.println("Sonar3 ");
      Serial.println("Detectado");
      Serial.println(stb);
      s3 = 1;
    }
    if(i==4 && s4==0)
    {
      Serial.println("Sonar4 ");
      Serial.println("Detectado");
      Serial.println(stb);
      s4 = 1;
    }
    if(i==5 && s5==0)
    {
      Serial.println("Sonar5 ");
      Serial.println("Detectado");
      Serial.println(stb);
      s5 = 1;
    }
    if(i==6 && s6==0)
    {
      Serial.println("Sonar6 ");
      Serial.println("Detectado");
      Serial.println(stb);
      s6 = 1;
    }
    if(i==7 && s7==0)
    {
      Serial.println("Sonar7 ");
      Serial.println("Detectado");
      Serial.println(stb);
      s7 = 1;
    }
    if(i==8 && s8==0)
    {
      Serial.println("Sonar8 ");
      Serial.println("Detectado");
      Serial.println(stb);
      s8 = 1;
    }
  }
}
// 2 últimos dígitos - Datos bumpers
if(i>8)
{
```

```
        if (recibe_datos == 1)
        {
          if(i==9 && b1==0)
          {
            Serial.println("Bumper_delantero:  ");
            Serial.println("Pulsado. ");
            Serial.println(stb);
            b1 = 1;
          }
          if(i==10 && b2==0)
          {
            Serial.println("Bumper_trasero:  ");
            Serial.println("Pulsado. ");
            Serial.println(stb);
            b2 = 1;
          }
        }
       }
       i=i+1;
      stb = s1+s2+s3+s4+s5+s6+s7+s8+b1+b2;
     }
  }
}
// Visualización del PID derecho, hacia delante
else if(comando == '4')
{
  int contador_der = 0;
  while(contador_der<=4000)
  {
    W_ref_der = 0.2;

    Compute();                      // Cálculo del PID
   Girar(Velocidad_motor_der,0);  // Movimiento de motores

    contador_der = contador_der+1;
     delay(10);
  }
  Girar(0,0);
}
// Visualización del PID derecho, hacia atrás
else if(comando == '5')
{
  int contador_der = 0;
  while(contador_der<=4000)
  {
    W_ref_der = -0.2;

    Compute();                      // Cálculo del PID
   Girar(Velocidad_motor_der,0);  // Movimiento de motores
```

```
        contador_der = contador_der+1;
         delay(10);
       }
      Girar(0,0);
    }
    // Visualización del PID izquierdo, hacia delant
    else if(comando == '6')
    {
      int contador_izq = 0;
      while(contador_izq<=4000)
      {
        W_ref_izq = 0.2;

        Compute();                      // Cálculo del PID
       Girar(0,Velocidad_motor_izq);  // Movimiento de motores

        contador_izq = contador_izq+1;
         delay(10);
       }
      Girar(0,0);
    }
    // Visualización del PID izquierdo, hacia atrás
    else if(comando == '7')
    {
      int contador_izq = 0;
      while(contador_izq<=4000)
      {
        W_ref_izq = -0.2;

        Compute();                      // Cálculo del PID
       Girar(0,Velocidad_motor_izq);  // Movimiento de motores

        contador_izq = contador_izq+1;
         delay(10);
       }
      Girar(0,0);
    }
  }
}

//*****************************************************************//
// Nombre de la función: Comprobacion_motor_der()
// Llamada por: loop()
// Función: Encargada de comprobar el correcto mocimiento de la rueda.
// Entradas: Ninguna.
// Salidas: Movimiento de motor derecho.
//*****************************************************************//
void Comprobacion_motor_der()
```

```
{
  // Gira hacia atrás
  if(Cambio == 0)
  {
    if (Contador_der >= -490)
      Girar(-20,0);
    else
    {
      Girar(0,0);
      Cambio = 1;
    }
  }
  // Gira hacia delante
  if (Cambio == 1)
  {
    if (Contador_der <= 0)
      Girar(20,0);
    else
    {
      Girar(0,0);
      Cambio = 0;
    }
  }
  delay(10);
}

//*****************************************************************//
// Nombre de la función: Comprobacion_motor_izq()
// Llamada por: loop()
// Función: Encargada de comprobar el correcto mocimiento de la rueda.
// Entradas: Ninguna.
// Salidas: Movimiento de motor izquierdo.
//*****************************************************************//
void Comprobacion_motor_izq()
{
  // Gira hacia atrás
  while(Cambio == 0)
  {
    if (Contador_izq >= -490)
    {
      Girar(0,-20);
    }
    else
    {
      Girar(0,0);
      Cambio = 1;
    }
  }
  // Gira hacia delante
```

```
  while (Cambio == 1)
  {

    if (Contador_izq <= 0)
    {
      Girar(0,20);
    }
    else
    {
      Girar(0,0);
      Cambio = 0;
    }
  }
  delay(10);
}

//*****************************************************************//
// Nombre de la función: Contar_der()
// Llamada por: la interrupción número 0.
// Función: Encargada de leer los canales del encoder derecho y en
//          función de lo leído, se actualizan los pulsos generados.
// Entradas: Ninguna.
// Salidas: Ninguna.
//*****************************************************************//
void Contar_der()
{
  EstadoCanalB_der = digitalRead(CanalB_der);
  EstadoCanalA_der = digitalRead(CanalA_der);

  if(EstadoCanalA_der == HIGH)
  {
    if(EstadoCanalB_der == LOW)
      Contador_der = Contador_der+1;
    else
      Contador_der = Contador_der-1;
  }
  else
  {
    if(EstadoCanalB_der == LOW)
      Contador_der = Contador_der-1;
    else
      Contador_der = Contador_der+1;
  }
}

//*****************************************************************//
// Nombre de la función: Contar_izq()
// Llamada por: la interrupción número 4.
// Función: Encargada de leer los canales del encoder izquierdo y en
```

```
//         función de lo leído, se actualizan los pulsos generados.
// Entradas: Ninguna.
// Salidas: Ninguna.
//*****************************************************************//
void Contar_izq()
{
  EstadoCanalB_izq = digitalRead(CanalB_izq);
  EstadoCanalA_izq = digitalRead(CanalA_izq);

  if(EstadoCanalA_izq == HIGH)
  {
    if(EstadoCanalB_izq == LOW)
      Contador_izq = Contador_izq+1;
    else
      Contador_izq = Contador_izq-1;
  }
  else
  {
    if(EstadoCanalB_izq == LOW)
      Contador_izq = Contador_izq-1;
    else
      Contador_izq = Contador_izq+1;
  }
}

//*****************************************************************//
// Nombre de la función: Girar()
// Llamada por: loop()
// Función: Encargada de girar los motores.
// Entradas: Velocidad a la que debe girar cada motor.
// Salidas: Movimiento de motores.
//*****************************************************************//
void Girar(double Velocidad_motor_der, double Velocidad_motor_izq)
{
  // Motor derecho
  if(Velocidad_motor_der==0)
  {
    digitalWrite(MotorDIR_der,LOW);
    analogWrite(MotorPWM_der, 0);
  }
  if(Velocidad_motor_der>0)
  {
    digitalWrite(MotorDIR_der,LOW);
    analogWrite(MotorPWM_der, (int)Velocidad_motor_der);
  }
  if(Velocidad_motor_der<0)
  {
    digitalWrite(MotorDIR_der,HIGH);
    analogWrite(MotorPWM_der, (int)Velocidad_motor_der);
```

```
  }

  // Motor izquierdo
  if(Velocidad_motor_izq==0)
  {
    digitalWrite(MotorDIR_izq,LOW);
    analogWrite(MotorPWM_izq, 0);
  }
  if(Velocidad_motor_izq>0)
  {
    digitalWrite(MotorDIR_izq,LOW);
    analogWrite(MotorPWM_izq, (int)Velocidad_motor_izq);
  }
  if(Velocidad_motor_izq<0)
  {
    digitalWrite(MotorDIR_izq,HIGH);
    analogWrite(MotorPWM_izq, (int)Velocidad_motor_izq);
  }
}

//*********************************************************************//
// Nombre de la función: Compute()
// Llamada por: loop()
// Función: Encargada de calcular el error PID de los motores y de
//          mantener la velocidad estable.
// Entradas: Ninguna.
// Salidas: Error PID y velocidad que debe ser introducida a cada motor.
//*********************************************************************//
void Compute()
{
  now = millis();

  if(now>LastTime+Pmuestreo)
  {
    // Diferencial de tiempo
    dT = now-LastTime;
    // Diferencial de pulsos
   dp_der = (Contador_der-LastContador_der);
   dp_izq = (Contador_izq-LastContador_izq);
    // Velocidad angular medida
    W_act_der = (double)dp_der/dT;
    W_act_izq = (double)dp_izq/dT;

    // Actualización de valores
   LastContador_der = Contador_der;
   LastContador_izq = Contador_izq;
    LastTime = now;
   }
```

```
 if((W_act_der==0 && W_ref_der!=0)||((W_act_izq==0 && W_ref_izq!=0)))
 {
   Eint_der = 0;
   Eint_izq = 0;
 }
 if(W_ref_der==0 && W_ref_der>=-0.2)
 {
   digitalWrite(MotorDIR_der,LOW);
   analogWrite(MotorPWM_der, 0);
 }

 if(W_ref_izq == 0 && W_ref_izq>=-0.2)
 {
   digitalWrite(MotorDIR_izq,LOW);
   analogWrite(MotorPWM_izq, 0);
 }

 // CÁLCULO DEL ERROR PID DERECHO
 // Error proporcional
Eactual_der = W_ref_der-W_act_der;
Eproporcional_der = Eactual_der*Kp;

 // Error integral
 if(Error_der >= VelMax && Error_der > 0)
   Error_der = VelMax;
 else
   Eint_der += Eactual_der*dT;

 if(Error_der <= VelMax && Error_der < 0)
   Error_der = -VelMax;
 else
   Eint_der += Eactual_der*dT;

Eintegral_der = Eint_der*Ki;

 // Error derivativo
Eder_der = (Eactual_der-LastError_der)/dT;
Ederivativo_der = Eder_der*Kd;

 // Error total
Error_der = Eproporcional_der+Eintegral_der+Ederivativo_der;

 // Cálculo de la velocidad
Velocidad_motor_der = Error_der;
 // Suponemos una velocidad máxima y mínima, por si se supera
Velocidad_motor_der = constrain(Velocidad_motor_der,-VelMax,VelMax);

 // Actualización de valores
LastError_der = Eactual_der;
```

```
    // CÁLCULO DEL ERROR PID IZQUIERDO
    // Error proporcional
   Eactual_izq = W_ref_izq-W_act_izq;
   Eproporcional_izq = Eactual_izq*Kp;

    // Error integral
    if(Error_izq >= VelMax && Error_izq > 0)
      Error_izq = VelMax;
    else
      Eint_izq += Eactual_izq*dT;

    if(Error_izq <= VelMax && Error_izq < 0)
      Error_izq = -VelMax;
    else
      Eint_izq += Eactual_izq*dT;

   Eintegral_izq = Eint_izq*Ki;

    // Error derivativo
   Eder_izq = (Eactual_izq-LastError_izq)/dT;
   Ederivativo_izq = Eder_izq*Kd;

    // Error total
  Error_izq = Eproporcional_izq+Eintegral_izq+Ederivativo_izq;

    // Cálculo de la velocidad
   Velocidad_motor_izq = Error_izq;

    // Suponemos una velocidad máxima y mínima, por si se supera
   Velocidad_motor_izq = constrain(Velocidad_motor_izq,-VelMax,VelMax);

    // Actualización de valores
   LastError_izq = Eactual_izq;
}
```

```
//*****************************************************************//
//                    ANEXO 15. ODOMETRÍA ARDUINO                  //
//*****************************************************************//

//********************* DECLARACIÓN DE VARIABLES **********************//
#define VelMax 255     // Máxima velocidad permitida
//************ Motores
// Derecho
const int MotorDIR_der = 4; // Pin DIR del puente H derecho
const int MotorPWM_der = 5; // Pin PWM del puente H derecho
const int enable_der = 6;   // Habilitación del motor derecho
// Izquierdo
const int MotorDIR_izq = 7; // Pin DIR del puente H izquierdo
const int MotorPWM_izq = 8; // Pin PWM del puente H izquierdo
const int enable_izq = 9;   // Habilitación del motor izquierdo
//************ Encoders
// Derecho
const int CanalB_der = 3; // PIN del canal B del encoder derecho
const int CanalA_der = 2; // PIN del canal A del encoder derecho
volatile int Contador_der = 0; // Contador de pulsos del encoder
int LastContador_der = 0;      // Contador de pulsos del encoder
volatile unsigned int EstadoCanalB_der = 0;  // Estado del canal B
volatile unsigned int EstadoCanalA_der = 0;  // Estado del canal A
// Izquierdo
const int CanalB_izq = 19; // PIN del canal B del encoder izquierdo
const int CanalA_izq = 18; // PIN del canal A del encoder izquierdo
volatile int Contador_izq = 0; // Contador de pulsos del encoder
int LastContador_izq = 0;      // Contador de pulsos del encoder
volatile unsigned int EstadoCanalB_izq = 0;  // Estado del canal B
volatile unsigned int EstadoCanalA_izq = 0;  // Estado del canal A
//************ Velocidades
// Motor izquierdo
double W_act_izq = 0;  // Velocidad que medimos en el motor izquierdo
int dp_izq = 0;        // Pulsos medidos del encoder izquierdo
// Motor derecho
double W_act_der = 0;  // Velocidad que medimos en el motor derecho
int dp_der = 0;        // Pulsos medidos del encoder derrecho
// Periodos de comunicación
unsigned long  Pmuestreo = 200; // Periodo de muestreo
// Variables temporales
unsigned long LastTime; // Tiempo anterior
unsigned long now;          // Tiempo actual
unsigned long dT;       // Diferencia de tiempos
// Variables odometría
double x_ant = 0, x_actual = 0;  // Posición eje X
double y_ant = 0, y_actual = 0;  // Posición eje Y
double theta_ant = 0, theta_actual = 0;  // Ángulo

//*****************************************************************//
```

```
// Nombre de la función: setup()
// Función: Función principal de inicialización de variables.
//          Se ejecuta una sóla vez. En ella se realiza lo siguiente:
//             - Inicia la comunicación puerto serie.
//             - Define los pines PWM y de dirección como salidas.
//             - Define los canales de los encoders como entradas.
//             - Pone los canales de los encoders a HIGH.
//             - Activa la interrupción de los encoders.
//             - Almancena el tiempo actual.
// Entradas: Ninguna.
// Salidas: Ninguna.
//*************************************************************************//
void setup(){

  Serial.begin(9600);

  // Salidas
  pinMode(MotorPWM_izq, OUTPUT);
  pinMode(MotorDIR_izq, OUTPUT);

  pinMode(MotorPWM_der, OUTPUT);
  pinMode(MotorDIR_der, OUTPUT);

  // Entradas
  // Motor izquierdo
  pinMode(CanalB_izq, INPUT);
  digitalWrite(CanalB_izq, HIGH);
  pinMode(CanalA_izq, INPUT);
  digitalWrite(CanalA_izq, HIGH);
  // Motor derecho
  pinMode(CanalB_der, INPUT);
  digitalWrite(CanalB_der, HIGH);
  pinMode(CanalA_der, INPUT);
  digitalWrite(CanalA_der, HIGH);

  // Interrupciones.
  //Mega:(interr,pin)[(0,2),(1,3),(2,21),(3,20),(4,19),(5,18)]
  attachInterrupt(4,Contar_izq,CHANGE);
  attachInterrupt(0,Contar_der,CHANGE);

  // Tiempo actual
  LastTime = millis();
}

//*************************************************************************//
// Nombre de la función: loop()
// Función: Función principal. Actúa como bucle, es decir, se ejecuta
//          cíclicamente.
// Entradas: Ninguna.
```

```
// Salidas: Ninguna.
//*********************************************************************//
void loop()
{
  Compute();    // Cálculo de pulsos y velcidades
  Odometria();  // Cálculo de la odometría

  delay(100);
}

//*********************************************************************//
// Nombre de la función: Compute()
// Llamada por: loop()
// Función: Encargada de calcular el diferencial de pulsos y la velocidad
// Entradas: Ninguna.
// Salidas: Ninguna.
//*********************************************************************//
void Compute()
{
  now = millis();

  if(now > LastTime+Pmuestreo)
  {
    // Diferencial de tiempo
    dT = now-LastTime;
    // Diferencial de pulsos
   dp_izq = (Contador_izq-LastContador_izq);
   dp_der = (Contador_der-LastContador_der);
    // Velocidad angular medida
    W_act_izq = (double)dp_izq/dT;
    W_act_der = (double)dp_der/dT;

    // Actualización de valores
   LastContador_izq = Contador_izq;
   LastContador_der = Contador_der;
    LastTime = now;
  }

}

//*********************************************************************//
// Nombre de la función: Contar_der()
// Llamada por: la interrupción número 0.
// Función: Encargada de leer los canales del encoder derecho y en
//          función de lo leído, se actualizan los pulsos generados.
// Entradas: Ninguna.
// Salidas: Ninguna.
//*********************************************************************//
void Contar_der()
```

```
{
  EstadoCanalB_der = digitalRead(CanalB_der);
  EstadoCanalA_der = digitalRead(CanalA_der);

  if(EstadoCanalA_der == HIGH)
  {
    if(EstadoCanalB_der == LOW)
      Contador_der = Contador_der+1;
    else
      Contador_der = Contador_der-1;
  }
  else
  {
    if(EstadoCanalB_der == LOW)
      Contador_der = Contador_der-1;
    else
      Contador_der = Contador_der+1;
  }
}

//*****************************************************************//
// Nombre de la función: Contar_izq()
// Llamada por: la interrupción número 4.
// Función: Encargada de leer los canales del encoder izquierdo y en
//          función de lo leído, se actualizan los pulsos generados.
// Entradas: Ninguna.
// Salidas: Ninguna.
//*****************************************************************//
void Contar_izq()
{
  EstadoCanalB_izq = digitalRead(CanalB_izq);
  EstadoCanalA_izq = digitalRead(CanalA_izq);

  if(EstadoCanalA_izq == HIGH)
  {
    if(EstadoCanalB_izq == LOW)
      Contador_izq = Contador_izq+1;
    else
      Contador_izq = Contador_izq-1;
  }
  else
  {
    if(EstadoCanalB_izq == LOW)
      Contador_izq = Contador_izq-1;
    else
      Contador_izq = Contador_izq+1;
  }
}
```

```
//*****************************************************************//
// Nombre de la función: Odometria()
// Llamada por: loop()
// Función: Encargada de generar la odometría del robot.
// Entradas: Ninguna.
// Salidas: Ninguna.
//*****************************************************************//
void Odometria()
{
  // Variables a utilizar
  double L = 0.42;       // Distancia entre ruedas motrices [m]
  double r = 0.085;     // Radio de las ruedas motrices [m]
  double dist_lineal_der, dist_lineal_izq; // Dist recorrida por rueda
  double dist_lineal_media;  // Dist recorrida en el centro del robot
  double ang_medio;          // Ángulo recorrido

  // Distancia recorrida por cada rueda
  dist_lineal_der = 2*W_act_der*r;
  dist_lineal_izq = 2*W_act_izq*r;

  // Distancia y ángulo recorridos globales
  dist_lineal_media = (dist_lineal_der+dist_lineal_izq)/2;
  ang_medio = (dist_lineal_der-dist_lineal_izq)/L;

  // Cálculo de la odometría
  theta_actual = theta_ant+(ang_medio*dT);
  x_actual = x_ant+(dist_lineal_media*dT*cos(theta_actual));
  y_actual = y_ant+(dist_lineal_media*dT*sin(theta_actual));

  // Impresión por pantalla
  Serial.println(x_actual);
  Serial.println(y_actual);
  Serial.println(theta_actual);

  // Actualización de valores
  x_ant = x_actual;
  y_ant = y_actual;
  theta_ant = theta_actual;
}
```

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%                    ANEXO 16. ODOMETRÍA. MATLAB                             %
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Nombre de la función: Odometría
% Función: Comprobación de la correcta aplicación de la localización
%          odométrica.
% Entradas: Los valores de las posiciones X e Y conseguidas a partir
%           del puerto COM3.
% Salidas: Representación gráfica de la trayectoria del robot.
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
function  Odometria

    % Puesta a punto
    close  all;
    clc;

    % Inicialización del puerto serial
    delete (instrfind ({'port' },{'COM3' }));
    puerto_serial  = serial ('COM3' );
    puerto_serial .BaudRate  = 9600 ;
    warning ('off' ,'MATLAB_serial:fscanf:unsuccessfulRead');

    % Abro el puerto serial
    fopen(puerto_serial );

    % Declaro un contador del número de muestras tomadas
    contador = 1;
    % Creación de vectores para guardar datos
    x = zeros(1, 100000);
    y = zeros(1, 100000);

    % Creo una ventana gráfica
    figure('Name','POSICIONAMIENTO DEL ROBOT')
    title('Odometría');
    xlabel('Eje X');
    ylabel('Eje Y');

    grid off;
    hold on;

    % Para una representación en tiempo real
    lHandle = line(nan, nan);

    %Bucle while para que tome y dibuje las muestras
    while true
        posicion_X = fscanf(puerto_serial ,'%g');
        posicion_Y = fscanf(puerto_serial ,'%g');
        angulo = fscanf(puerto_serial ,'%g');

        disp(posicion_X);
        disp(posicion_Y);
        disp(angulo);

        x(contador) = posicion_X;
        y(contador) = posicion_Y;

        X = get(lHandle, 'XData');
        Y = get(lHandle, 'YData');

        x = [X x(contador)];
        y = [Y y(contador)];

        set(lHandle, 'XData', x, 'YData', y);

        drawnow
        contador = contador+1;
    end

    % Cierro la conexión con el puerto serial y elimino las variables
    fclose(puerto_serial );
```

```
    delete(puerto_serial);
    clear all;
end
```

```
//*****************************************************************//
//            ANEXO 17. SIMULACIÓN FINAL. ARDUINO MAESTRO          //
//*****************************************************************//

//********************* DECLARACIÓN DE VARIABLES **********************//
// Comunicación entre Arduino Mega maestro - Arduino Uno
#include <Wire.h>
#define VelMax 255       // Máxima velocidad permitida
#define Alarma_sonar 40  // Distancia a la que se da alarma de choque
//*********** BATERÍA *********//
// Variables digitales
int Led_Verde = 5;
int Led_Amarillo = 6;
int Led_Rojo = 7;
// Variable analógica
int Bateria = A2;
// Otras variables utiles
float Lectura_bat; // Valor donde se almacena el valor de la bateria
float Valor_bat;   // Valor convertido de la lectura
int Alarma_baja = 0;       // Alarma = 0 - Alarma inactiva.
                           // Alarma = 1 - Alarma activada.
int Alarma_critica = 0;    // Alarma = 0 - Alarma inactiva.
                           // Alarma = 1 - Alarma activada.
int Alarma_baja_ant = 0;
int Alarma_critica_ant = 0;
int Num_baterias = 1;
//*********** SONARS Y BUMPERS *********//
int a = 1, b, c; // Parámetros para bucles
int Sonars[8];
int Bumpers[2];
int Num_bumpers = 2;
int Num_sonars = 8;
// Periodos de comunicación
unsigned long  Pmuestreo = 100; // Periodo de muestreo
unsigned long P_2000 = 2000;    // Periodo de estado 2000 ms
unsigned long P_100 = 100;      // Periodo de estado 100 ms
// Variables temporales
unsigned long LastTime;     // Tiempo anterior
unsigned long now;          // Tiempo actual
unsigned long dT;           // Diferencia de tiempos
unsigned long T_actual;        // Tiempo actual
unsigned long T100_anterior;   // Tiempo anterior para periodo 100 ms
unsigned long T2000_anterior;  // Tiempo anterior para periodo 2000 ms
// Motor
int Num_motores = 2;
const int MotorDIR = 9; // Pin DIR del puente H
const int MotorPWM = 10; // Pin PWM del puente H
const int enable = 11;   // Habilitación del motor
// Encoder
```

```
const int CanalB = 3; // PIN del canal B del encoder
const int CanalA = 2; // PIN del canal A del encoder
volatile int Contador = 0; // Contador de pulsos del encoder
int LastContador = 0;      // Contador de pulsos del encoder
volatile unsigned int EstadoCanalB = 0;  // Estado del canal B
volatile unsigned int EstadoCanalA = 0;  // Estado del canal A
// Velocidad
double W_ref = 0;  // Velocidad angular de referencia
double W_act = 0;  // Velocidad que medimos en el motor
int dp = 0;        // Pulsos medidos del encoder
double Velocidad_motor = 0; // Velocidad que introducimos al motor
//************ Errores PID
// Parámetros PID
double Kp,Ki,Kd;
// Derecho
double Eactual, Eproporcional;  // Error proporcional
double Eint=0, Eintegral;       // Error integral
double Eder, Ederivativo;       // Error derivativo
double Error,LastError=0;       // Error total
//************ Dirección del robot
int al_s1 = 0, al_s2 = 0, al_s3 = 0, al_s4 = 0; // Alarmas choque sonars
int al_s5 = 0, al_s6 = 0, al_s7 = 0, al_s8 = 0; // Alarmas choque sonars
int al_bd = 0, al_bt = 0;                       // Alarmas choque bumpers
int al_choque_total = 0;                        // Número de alarmas
//*************** LECTURA PUERTO SERIE ***************//
byte Lectura_puerto;      // Almacenamos el valor recibido por el puerto serie
int Ref=0, Ref_der=0, Ref_izq=0; // Útiles para saber que valor analizar
int Num_ref = 1;          // Número de ref del conjunto
int Indicador_ref;        // Número de ref del conjunto
boolean mySwitch = true;  // Switch para saber el valor inicial del conjunto
int Cambio_ref = 0;       // Para cambiar de dígito a leer en el conjunto
int temporizador = 0;     // Temp para saber si el recibimos velocidad de ref
int temporizador1 = 0;    // Temp para saber si el recibimos velocidad de ref
int Lectura = 0;          // Para saber si recibimos velocidad de ref
int Negativo_der = 1;     // Signo de la velocidad de referencia del motor der
int Negativo_izq = 1;     // Signo de la velocidad de referencia del motor izq
int W_ref_izq = 0;        // Valor de la velocidad a enviar al motor izquierdo
int W_ref_der = 0;        // Valor de la velocidad a enviar al motor derecho
//************ LECTURA ARDUINO MEGA ESCLAVO ************//
byte Lectura_puerto1;     // Almacenamos el valor recibido por el puerto serie
int Ref_mega=0, Ref_dp=0, Ref_W=0; // Útiles para saber que valor analizar
int Indicador_ref_mega;   // Número de ref del conjunto
int Num_ref_mega = 1;     // Número de ref del conjunto
boolean mySwitch_puerto = true;  // Switch para saber el valor inicial del con
int Cambio_ref_mega = 0;  // Para cambiar de dígito a leer en el conjunto
int Negativo_dp = 1;      // Signo de la velocidad de referencia del motor der
int Negativo_W = 1;       // Signo de la velocidad de referencia del motor izq
int dp_izq = 0;           // Diferencial de pulsos del motor izquierdo
int W_act_izq = 0;        // Velocidad del motor izquierdo
```

```
double W_act_der = 0;     // Velocidad del motor derecho
int dp_der = 0;           // Diferencial de pulsos del motor derecho
int Lectura_mega_1 = 1;   // Para saber si el recibimos datos

// Otras variables útiles
int Imprimir;    // Indica a Martlab cuando debe imprimir las alarmas
                 // imprimir=0 - No imprimir/ imprimir=1 - imprimir

//*****************************************************************//
// Nombre de la función: setup()
// Función: Función principal de inicialización de variables.
//          Se ejecuta una sóla vez. En ella se realiza lo siguiente:
//             - Inicia la comunicación puerto serie.
//             - Inicia la comunicación I2C.
//             - Guarda tiempos actuales.
//             - Define los leds como salidas.
//             - Define los pines PWM y de dirección como salidas.
//             - Define los canales del encoder como entradas.
//             - Define el pin de habilitación como entrada.
//             - Apaga todos los leds.
//             - Pone los canales del encoder a HIGH.
//             - Pone el pin de habilitación a HIGH.
//             - Activa la interrupción del encoder.
//             - Almancena el tiempo actual.
//             - Establece una velocidad nula de motor.
//             - Define los parámetros del PID.
// Entradas: Ninguna.
// Salidas: Ninguna.
//*****************************************************************//
void setup() {
  // Inicialización del bus I2C
  Wire.begin();

  Serial.begin(115200);
  Serial1.begin(115200);

  // Guardamos tiempo actual
  T100_anterior = millis();
  T2000_anterior = millis();

  // SALIDAS
  // Batería
  pinMode(Led_Verde, OUTPUT);
  pinMode(Led_Amarillo, OUTPUT);
  pinMode(Led_Rojo, OUTPUT);
  // Motor derecho
  pinMode(MotorPWM, OUTPUT);
  pinMode(MotorDIR, OUTPUT);
```

```
  // Entradas
  pinMode(CanalB, INPUT);
  digitalWrite(CanalB, HIGH);
  pinMode(CanalA, INPUT);
  digitalWrite(CanalA, HIGH);
  pinMode(enable, INPUT);

  // Inicialmente todas las luces apagadas
  digitalWrite(Led_Verde, LOW);
  digitalWrite(Led_Amarillo, LOW);
  digitalWrite(Led_Rojo, LOW);

  // Habilitación de motor
  digitalWrite(enable,HIGH);

  // Interrupción.
  //Mega:(interr,pin)[(0,2),(1,3),(2,21),(3,20),(4,19),(5,18)]
  attachInterrupt(0,Contar,CHANGE);

  // Tiempo actual
  LastTime = millis();
  temporizador = millis();

  // Velocidad inicial nula
  digitalWrite(MotorDIR,LOW);
  analogWrite(MotorPWM, 0);

  //Definición de los parámetros del PID
  Kp = 150;
  Ki = 0.02;
  Kd = 0;

  Lectura_mega_1 = 1;
}

//*****************************************************************//
// Nombre de la función: loop()
// Función: Función principal. Actúa como bucle, es decir, se ejecuta
//          cíclicamente.
// Entradas: Ninguna.
// Salidas: Ninguna.
//*****************************************************************//
void loop()
{
  T_actual = millis();

  //****** INICIO COMUNICACIÓN ARDUINO MEGA ESCLAVO ******//
  // NOTA PRÁCTICA.
  // Protocolo de comunicación: [6,dp,W_act]
```

```
// En ASCII, los diferentes símbolos tienen una númeración:
// , =  44 / [ = 91 / ] = 93  / Números entre 49 y 57 / - = 45

if(Serial1.available()>0)
{
  while(Lectura_mega_1 == 1)
  {
     Lectura_puerto1 = Serial1.read(); // Leemos del puerto serie

     // Si recibimos un número o un signo "-" (Número negativo)
     if((Lectura_puerto1>47 && Lectura_puerto1<58)|| Lectura_puerto1==45)
     { // Inicialmente mySwitch down
       if(!mySwitch_puerto)
         // Lectura del número de referencia de comunicación
        Ref_mega = (Ref_mega*10)+(Lectura_puerto1-48);

       else
       { // Lectura del valor de las velocidad del motor izquierdo
         if(Num_ref_mega==2)
         { // Se trata de un número negativo
           if(Lectura_puerto1==45)
           {
             Ref_W = 0;
             Negativo_W = -1;
           }
           // Se trata de un número positivo
           else
            Ref_W = (Ref_W*10)+(Lectura_puerto1-48);
         }
         // Lectura del número de pulsos del motor izquierdo
         else
         {  // Se trata de un número negativo
           if(Lectura_puerto1==45)
           {
             Ref_dp = 0;
             Negativo_dp = -1;
           }
           // Se trata de un número positivo
           else
            Ref_dp = (Ref_dp*10)+(Lectura_puerto1-48);
         }
       }
     }

     // FIN DE COMUNICACIÓN. Almacenar valores.
     // El primer valor debe ser un 6.
     if(Lectura_puerto1==93)
     {
      Indicador_ref_mega = Ref_mega;
```

```
        if(Indicador_ref_mega==6)
        {
          dp_izq = (double)Negativo_dp*Ref_dp;
          W_act_izq = (double)Negativo_W*Ref_W;
        }

        // Actualización de valores
        Negativo_dp = 1;
        Negativo_W = 1;
        Indicador_ref = 0;
        Ref_mega = 0;
        Ref_dp = 0;
        Ref_W = 0;
        mySwitch_puerto = false;
        Num_ref_mega = 1;
        Lectura_mega_1 = 0;
      }
      // Lectura del signo ",". Se debe leer un nuevo dígito
      else if(Lectura_puerto1==44)
      { // mySwitch up y sumamos número de referencia
        mySwitch_puerto = true;
        Num_ref_mega = Num_ref_mega+1;
      }
      // INICIO DE COMUNICACIÓN, mySwitch down
      else if(Lectura_puerto1==91)
        mySwitch_puerto = false;
  }
}
//****** FIN COMUNICACIÓN ARDUINO MEGA ESCLAVO ******//

// Estado de los motores
if(al_choque_total == 0)// No hay obstáculos
{
  W_ref_der = 0.2;
  W_ref_izq = 0.2;
}
else if(al_choque_total >= 1)  // Si hay obstáculos
  Direccion_robot();   // Recalcula la dirección que debe tomar el robot

Compute();              // Cálculo del PID
Girar(Velocidad_motor); // Movimiento del motor
delay(10);

// Estado de la batería
Lectura_bateria();
Control_estado();

// Estado de bumpers y sonars
Lectura_sonars_bumpers();
```

```
  // Envío de todos los datos al control de alto nivel
  // Cada 100 ms:
  if(T_actual>T100_anterior+P_100)
  {
    Imprimir = 0;  // No se puede imprimir en matlab estado de las alarmas

    // Cada 2000 ms o alarma:
    if((T_actual>T2000_anterior+P_2000))
    {
      Imprimir = 1;  // Se tiene que imprimir en matlab estado de las alarmas
      T2000_anterior = T_actual;
    }

    // Estado de las baterias
    Serial.println(Num_baterias);
    Serial.println(Imprimir);
    Serial.println(Valor_bat);
    // Alarmas
    Serial.println(Alarma_baja);
    Serial.println(Alarma_critica);
    // Estado sonars
    Serial.println(Num_sonars);
    for(b=0 ; b<8 ; b++)
      Serial.println(Sonars[b]);
    // Estado bumpers
    Serial.println(Num_bumpers);
    for(c=0 ; c<2 ; c++)
      Serial.println(Bumpers[c]);
    // Estado de motores (velocidad y odometría)
    Serial.println(Num_motores);
    // Motor derecho
    Serial.println(W_act_der);
    Serial.println(dp_der);
    // Motor izquierdo
    Serial.println(W_act_izq);
    Serial.println(dp_izq);

    T100_anterior = T_actual;
  }
}

//*****************************************************************//
// Nombre de la función: Lectura_bateria()
// Llamada por: loop()
// Función: Encargada de leer el pin analógico y convertir la lectura
//          en el valor real de la batería (entre 0 y 12.5V).
// Entradas: Ninguna.
// Salidas: Ninguna
```

```
//*****************************************************************//
void Lectura_bateria()
{
  Lectura_bat = analogRead(Bateria);
  Valor_bat = (Lectura_bat*12.5)/1023;
}

//*****************************************************************//
// Nombre de la función: Control_estado()
// Llamada por: loop()
// Función: Encargada de actuar en función de la carga de la bateria:
//          - Bateria cargada (12.5 V,11.3 V): Led verde encendido.
//                                             No alarmas.
//          - Bateria media (11.3 V,10.8 V): Led amarillo encendido.
//                                           Alarma bateria baja.
//          - Bateria baja (10.8 V,0 V): Led rojo encendido.
//                                       Alarma bateria crítica.
// Entradas: Ninguna.
// Salidas: Ninguna
//*****************************************************************//
void Control_estado()
{
  // Bateria cargada
  if(Valor_bat>11.3)
  {
    digitalWrite(Led_Verde, HIGH);
    digitalWrite(Led_Amarillo, LOW);
    digitalWrite(Led_Rojo, LOW);
    Alarma_baja = 0;
    Alarma_critica = 0;
  }
  // Bateria con carga media
  if(Valor_bat<=11.3 && Valor_bat>10.8)
  {
    digitalWrite(Led_Verde, LOW);
    digitalWrite(Led_Amarillo, HIGH);
    digitalWrite(Led_Rojo, LOW);
    Alarma_baja = 1;
    Alarma_critica = 0;
  }
  // Bateria con carga crítica
  if(Valor_bat<=10.8)
  {
    digitalWrite(Led_Verde, LOW);
    digitalWrite(Led_Amarillo, LOW);
    digitalWrite(Led_Rojo, HIGH);
    Alarma_baja = 0;
    Alarma_critica = 1;
  }
```

```
}

//*************************************************************************//
// Nombre de la función: Lectura_sonars_bumpers()
// Llamada por: loop()
// Función: Encargada de leer la lectura de todos los ultrasonidos y
//          bumpers presentes en el robot por el puerto I2C.
//          Establece alarma de choque si el obstáculo se encuentra
//          a menos de 40cm.
// Entradas: Ninguna.
// Salidas: Ninguna.
//*************************************************************************//
void Lectura_sonars_bumpers()
{
  Wire.requestFrom(2, 10); // Petición de 10 datos por la dirección 2
  while(Wire.available())
  {
    // Inicialización del bucle
    if(a>10)
      a=1;

    // Almacenamiento de dato leido
    int recibe_datos = Wire.read();

    // 8 primeros dígitos - Datos sonars
    if(a<=8)
    {
     Sonars[a-1] = recibe_datos;

      // ¿Activamos alarma de choque?
      if(Sonars[0]<=Alarma_sonar)
        al_s1 = 1;
      else
        al_s1 = 0;
      if(Sonars[1]<=Alarma_sonar)
        al_s2 = 1;
      else
        al_s2 = 0;
      if(Sonars[2]<=Alarma_sonar)
        al_s3 = 1;
      else
        al_s3 = 0;
      if(Sonars[3]<=Alarma_sonar)
        al_s4 = 1;
      else
        al_s4 = 0;
      if(Sonars[4]<=Alarma_sonar)
        al_s5 = 1;
      else
```

```
        al_s5 = 0;
      if(Sonars[5]<=Alarma_sonar)
        al_s6 = 1;
      else
        al_s6 = 0;
      if(Sonars[6]<=Alarma_sonar)
        al_s7 = 1;
      else
        al_s7 = 0;
      if(Sonars[7]<=Alarma_sonar)
        al_s8 = 1;
      else
        al_s8 = 0;
    }

    // 2 últimos dígitos - Datos bumpers
    if(a>8)
    {
     Bumpers[a-9] = recibe_datos;

      // ¿Activamos alarma de choque?
      if(Bumpers[1]==1)
        al_bd = 1;
      else
        al_bd = 0;
      if(Bumpers[2]==1)
        al_bt = 1;
      else
        al_bt = 0;
    }
    a=a+1;
  }
 al_choque_total = al_s1+al_s2+al_s3+al_s4+al_s5+al_s6+al_s7+al_s8+al_bd+al_b
}

//*****************************************************************//
// Nombre de la función: Contar()
// Llamada por: la interrupción número 0.
// Función: Encargada de leer los canales del encoder y en función de
//          lo leído, se actualizan los pulsos generados.
// Entradas: Ninguna.
// Salidas: Ninguna.
//*****************************************************************//
void Contar()
{
  EstadoCanalB = digitalRead(CanalB);
  EstadoCanalA = digitalRead(CanalA);

  if(EstadoCanalA == HIGH)
```

```
  {
    if(EstadoCanalB == LOW)
      Contador = Contador+1;
    else
      Contador = Contador-1;
  }
  else
  {
    if(EstadoCanalB == LOW)
      Contador = Contador-1;
    else
      Contador = Contador+1;
  }
}

//*************************************************************************//
// Nombre de la función: Girar()
// Llamada por: loop()
// Función: Encargada de girar el motor.
// Entradas: Velocidad a la que debe girar el motor.
// Salidas: Movimiento del motor.
//*************************************************************************//
void Girar(double Velocidad_motor)
{
  if(Velocidad_motor==0)
  {
    digitalWrite(MotorDIR,LOW);
    analogWrite(MotorPWM, 0);
  }
  if(Velocidad_motor>0)
  {
    digitalWrite(MotorDIR,LOW);
    analogWrite(MotorPWM, (int)Velocidad_motor);
  }
  if(Velocidad_motor<0)
  {
    digitalWrite(MotorDIR,HIGH);
    analogWrite(MotorPWM, (int)Velocidad_motor);
  }
}

//*************************************************************************//
// Nombre de la función: Compute()
// Llamada por: loop()
// Función: Encargada de calcular el error PID del motor y de
//          mantener la velocidad estable.
// Entradas: Ninguna.
// Salidas: Error PID y velocidad que debe ser introducida al motor.
//*************************************************************************//
```

```
void Compute()
{
  now = millis();

  if(now>LastTime+Pmuestreo)
  {
    // Diferencial de tiempo
    dT = now-LastTime;
    // Diferencial de pulsos
    dp = (Contador-LastContador);
    // Velocidad angular medida
    W_act = (double)dp/dT;

    // Para mandar al control de alto nivel
    W_act_der = W_act;
    dp_der = dp;

    // Actualización de valores
    LastContador = Contador;
    LastTime = now;
   }

   if(W_act==0 && W_ref!=0)
     Eint = 0;

   if(W_ref==0 && W_ref>=-0.2)
   {
     digitalWrite(MotorDIR,LOW);
     analogWrite(MotorPWM, 0);
   }


   // CÁLCULO DEL ERROR PID
   // Error proporcional
  Eactual = W_ref-W_act;
  Eproporcional = Eactual*Kp;

   // Error integral
   if(Error >= VelMax && Error > 0)
     Error = VelMax;
   else
     Eint += Eactual*dT;

   if(Error <= VelMax && Error < 0)
     Error = -VelMax;
   else
     Eint += Eactual*dT;

   Eintegral = Eint*Ki;
```

```
   // Error derivativo
   Eder = (Eactual-LastError)/dT;
   Ederivativo = Eder*Kd;

   // Error total
   Error = Eproporcional+Eintegral+Ederivativo;

   // Cálculo de la velocidad
   Velocidad_motor = Error;
   // Suponemos una velocidad máxima y mínima, por si se supera
   Velocidad_motor = constrain(Velocidad_motor,-VelMax,VelMax);

   // Actualización de valores
   LastError = Eactual;
}

//*****************************************************************//
// Nombre de la función: Direccion_robot()
// Llamada por: loop()
// Función: Encargada de establecer la velocidad adecuada para motor
//          en función de los obstáculos detectados.
// Entradas: Ninguna.
// Salidas: Velocidad adecuada de cada motor.
//*****************************************************************//
void Direccion_robot()
{
  // Si el obstáculo se encuentra delante
  if((al_s3==1 && al_s2==1 && al_s1==0 && (al_s4==1 || al_s4==0)))
  { // Giramos hacia la derecha
    W_ref = 0.1;
    W_ref_izq = 0.2;
    // Mandar al Arduino Mega esclavo
    Serial1.print("[");
    Serial1.print(4);
    Serial1.print(",");
    Serial1.print(W_ref_izq);
    Serial1.print("]");
    delay(100);
    Serial1.flush();
  }
  else if((al_s3==1 && al_s2==1 && al_s4==0 && (al_s1==1 || al_s1==0)))
  { // Giramos hacia la izquierda
    W_ref_izq = 0.1;
    W_ref = 0.2;
    // Mandar al Arduino Mega esclavo
    Serial1.print("[");
    Serial1.print(4);
    Serial1.print(",");
```

```
  Serial1.print(W_ref_izq);
  Serial1.print("]");
  delay(100);
  Serial1.flush();
}
else if(al_s3==1 && al_s2==1 && al_s4==1 && al_s1==1)
{
  if(Sonars[0]-Sonars[3]>=0)
  {
    W_ref = 0.1;
    W_ref_izq = 0.2;
    // Mandar al Arduino Mega esclavo
    Serial1.print("[");
    Serial1.print(4);
    Serial1.print(",");
    Serial1.print(W_ref_izq);
    Serial1.print("]");
    delay(100);
    Serial1.flush();
  }
  else
  {
    W_ref_izq = 0.1;
    W_ref = 0.2;
    // Mandar al Arduino Mega esclavo
    Serial1.print("[");
    Serial1.print(4);
    Serial1.print(",");
    Serial1.print(W_ref_izq);
    Serial1.print("]");
    delay(100);
    Serial1.flush();
  }
}
// Si el obstáculo se encuentra en el centro y la derecha o a la derecha
else if(((al_s2==1 && al_s1==1) || al_s1==1 || al_s2==1) && al_s3==0 && al_s
{ // Giramos ligeramente hacia la izquierda
  W_ref_izq = 0.1;
  // Mandar al Arduino Mega esclavo
  Serial1.print("[");
  Serial1.print(4);
  Serial1.print(",");
  Serial1.print(W_ref_izq);
  Serial1.print("]");
  delay(100);
  Serial1.flush();
}

// Si el obstáculo se encuentra en el centro y la izquierda o a la izquierda
```

```
  else if(((al_s4==1 && al_s3==1) || al_s4==1 || al_s3==1) && al_s1==0 && al_s
  { // Giramos ligeramente hacia la derecha
    W_ref = 0.1;
  }

  else
  {
    W_ref = 0.2;
    W_ref_izq = 0.2;
    // Mandar al Arduino Mega esclavo
    Serial1.print("[");
    Serial1.print(4);
    Serial1.print(",");
    Serial1.print(W_ref_izq);
    Serial1.print("]");
    delay(100);
    Serial1.flush();
  }

}
```

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%               ANEXO 18. SIMULACIÓN FINAL. MATLAB                      %
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Nombre de la función: Simulación_Final
% Función: Simulación del control de alto nivel final del robot Edubot.
% Entradas: Todos los valores oportuno, conseguidos a partir del
%                   puerto COM3.
% Salidas: Representación gráfica de los datos del robot.
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
function  Simulacion_Final
    % Puesta a punto del programa
    close  all;
    clc;

    % Inicialización del puerto serial
    delete(instrfind({'port'},{'COM3'}));
    puerto_serial  = serial('COM3');
    puerto_serial.BaudRate  = 9600;
    warning('off','MATLAB_serial:fscanf:unsuccessfulRead');

    % Abro el puerto serial
    fopen(puerto_serial );

    % Inicialización de variables
    contador = 1;
    x_ant = 0;
    y_ant = 0;
    theta_ant = 0;
    Alarma_baja_ant = 0;
    Alarma_critica_ant = 0;
    indicador_ref = 3;
    dist_ant_der = 0;
    dist_ant_izq = 0;
    % Creación de vectores para guardar datos
    Lectura_sonars = zeros(1,8);
    Lectura_bumpers = zeros(1,2);
    x = zeros(1, 100000);
    y = zeros(1, 100000);


    % Creo una ventana gráfica
    figure('Name','POSICIONAMIENTO DEL ROBOT')
    title('Odometría');
    xlabel('Eje X');
    ylabel('Eje Y');
    grid off;
    hold on;
    lHandle = line(nan, nan);

    while true
       % Leo del puerto serial
       Imprimir = fscanf(puerto_serial , '%d');

       Num_baterias = fscanf(puerto_serial , '%d');
       Estado_bateria = fscanf(puerto_serial , '%g');

       Alarma_baja = fscanf(puerto_serial , '%d');
       Alarma_critica = fscanf(puerto_serial , '%d');

       Num_sonars = fscanf(puerto_serial ,'%d');
       for a=1:1:8
           Lectura_sonars(a) = fscanf(puerto_serial ,'%g');
       end

       Num_bumpers = fscanf(puerto_serial ,'%d');
       for b=1:1:2
           Lectura_bumpers(b) = fscanf(puerto_serial ,'%g');
       end

       Num_motores = fscanf(puerto_serial ,'%d');
```

```
        W_derecho = fscanf(puerto_serial, '%g');
        dp_derecho = fscanf(puerto_serial, '%d');
        W_izquierdo = fscanf(puerto_serial, '%g');
        dp_izquierdo = fscanf(puerto_serial, '%d');

        % Imprimir por pantalla
        if (Imprimir == 1)
            A=['Estado de la bateria: ', num2str(Estado_bateria)]:
            disp(A);
        end

        % Tratamiento de alarmas
        if (Alarma_baja_ant-Alarma_baja~=0 || Alarma_critica_ant-Alarma_critica~=0 ||
Imprimir_bateria==1)
            B = ['Alarma de baja bateria modificada: ', num2str(Alarma_baja)];
            disp(B);
            C = ['Alarma de bateria critica modificada: ', num2str(Alarma_critica)];
            disp(C);
        end
        Alarma_baja_ant = Alarma_baja;
        Alarma_critica_ant = Alarma_critica;

        for c=1:1:8
            D = ['Sonar ', num2str(c),':'];
            disp(D);
            disp(Lectura_sonars(c));
        end

        for d=1:1:2
            E = ['Bumper ', num2str(d),':'];
            disp(E);
            if(Lectura_bumpers(d) == 1)
                disp('No pulsado');
            else
                disp('Pulsado');
            end
        end

        F = ['Velocidad angular motor derecho : ', num2str(W_derecho)];
        disp(F);
        G = ['Velocidad angular motor izquierdo : ', num2str(W_izquierdo)];
        disp(G);

        % Cálculo de la Odometría
        [posicion_X, posicion_Y, theta_actual, dist_lineal_der, dist_lineal_izq] =
Odometria(dp_derecho, dp_izquierdo, x_ant, y_ant, theta_ant);

        % Actualización de valores
        dist_der = dist_lineal_der + dist_ant_der;
        dist_izq = dist_lineal_izq + dist_ant_izq;

        % Graficar datos
        x(contador) = posicion_X;
        y(contador) = posicion_Y;

        X = get(lHandle, 'XData');
        Y = get(lHandle, 'YData');

        x = [X x(contador)];
        y = [Y y(contador)];

        set(lHandle, 'XData', x, 'YData', y);

        drawnow                 % DRAWNOW es muy importante para la representación en
tiempo real
        contador = contador+1;

        x_ant = posicion_X;
        y_ant = posicion_Y;
        theta_ant = theta_actual;
        dist_ant_der = dist_der;
        dist_ant_izq = dist_izq;
```

```
    end

    % Cierro la conexión con el puerto serial y elimino las variables
    fclose(puerto_serial);
    delete(puerto_serial);
    clear all;

end
```