



TRABAJO DE FIN DE GRADO

---

# Diseño electrónico de un robot autónomo

---

*Autor:*

Javier MACÍAS SOLÁ

*Tutor:*

Dr. Leopoldo ACOSTA  
SÁNCHEZ

Departamento de Ingeniería Informática y de Sistemas

29 de junio de 2018



# *Abstract*

This project proposes the design and implementation of an autonomous, low-cost differential robot using Arduino and Raspberry Pi as control boards. Although the global project aims at obtaining NDVI images from the vehicle, this document only covers basic aspects of this idea. Nevertheless, it explains the solutions given for two different implementations of an autonomous rover that is meant to navigate through the waypoints provided. Despite the route being clear from obstacles, the rover should be able to deal with them in case of finding any. For this purpose, the first one covers a basic reactive approach from the starting morphology whilst the second is based on the ideas published in *the bubble rebound obstacle avoidance algorithm for mobile robots* [22].

Starting off from the same considerations, both of them use similar sensors and actuators that are described along with their principle of operation and the actual implementation that forms the whole electronic structure. A bang-bang and proportional-integral controller is compared from one iteration to the other, proving the suitability of the second over the first one in this application. Interestingly, sensor location was proved to be crucial for obtaining useful and accurate data to feed the algorithm. This idea is combined with average filtering of values to minimize noise. Also, descriptions of the functions designed are provided, as well as the whole final code in the appendices.

The issues faced during the development stage of each version are documented and discussed, as well as the schematics and blueprints needed to understand the ideas shown. At the end, some future ideas and improvements are suggested for the next coming iterations.



# Índice general

<b>Abstract</b>	<b>III</b>
<b>Índice general</b>	<b>V</b>
<b>Índice de figuras</b>	<b>IX</b>
<b>Índice de cuadros</b>	<b>XI</b>
<b>Índice de códigos</b>	<b>XII</b>
<b>Lista de abreviaturas</b>	<b>XV</b>
<b>1. Introducción</b>	<b>1</b>
1.1. Descripción del proyecto . . . . .	1
1.1.1. Motivación . . . . .	1
1.1.2. Descripción del lugar . . . . .	3
1.2. Estudio previo y bases de diseño . . . . .	4
1.2.1. Cinemática básica . . . . .	7
1.3. Punto de partida . . . . .	8
<b>2. Iteración 1</b>	<b>11</b>
2.1. Electrónica general . . . . .	11
2.1.1. Baterías . . . . .	11
2.1.2. Cargador . . . . .	13
2.1.3. Controlador . . . . .	13
2.2. Actuadores . . . . .	15
2.2.1. Motores . . . . .	16
2.2.1.1. Controlador de motores . . . . .	18
2.3. Sensores . . . . .	20
2.3.1. Brújula . . . . .	21

2.3.2. GPS . . . . .	23
2.3.3. WiFi . . . . .	25
2.3.4. Ultrasonidos . . . . .	26
2.4. Joystick . . . . .	29
2.4.1. Lector del joystick . . . . .	30
2.5. Interfaces y protocolos de comunicación . . . . .	31
2.5.1. I <sup>2</sup> C . . . . .	32
2.5.2. PWM . . . . .	32
2.5.3. Serial . . . . .	33
2.5.4. UDP . . . . .	33
<b>3. Implementación de la iteración 1</b>	<b>35</b>
3.1. Cálculo cinemático . . . . .	36
3.2. Modo manual . . . . .	37
3.2.1. Cliente . . . . .	38
3.2.2. Servidor . . . . .	39
3.3. Modo automático . . . . .	40
3.3.1. Algoritmo para evitar los obstáculos . . . . .	40
3.4. Joystick . . . . .	45
3.5. Presupuesto . . . . .	51
3.6. Discusión de resultados . . . . .	51
3.6.1. Problemas modo manual . . . . .	52
3.6.2. Problemas modo automático . . . . .	53
<b>4. Iteración 2</b>	<b>55</b>
4.1. Cambios estructurales . . . . .	55
4.2. Cambios de electrónica . . . . .	58
4.2.1. Raspberry Pi . . . . .	60
4.2.2. Cámara . . . . .	63
<b>5. Implementación de la iteración 2</b>	<b>65</b>
5.1. Alto nivel - Raspberry Pi . . . . .	65
5.2. Bajo nivel - Arduino . . . . .	69
5.2.1. Modo manual . . . . .	69
5.2.2. Modo automático . . . . .	70
5.2.2.1. Implementación del algoritmo . . . . .	73

5.2.2.2. Movimientos . . . . .	84
5.3. Seguridad y protección . . . . .	89
5.4. Presupuesto . . . . .	90
5.5. Discusión de resultados . . . . .	91
5.5.1. Problemas modo automático . . . . .	91
5.5.2. Problemas modo manual . . . . .	92
5.6. Conclusiones . . . . .	93
<b>6. Mejoras futuras</b>	<b>95</b>
6.1. Integración . . . . .	95
6.2. Comunicación y conectividad . . . . .	96
<b>A. Programas</b>	<b>99</b>
A.1. Código Arduino para la iteración 1 . . . . .	99
A.2. Cliente UDP . . . . .	117
A.3. Control de motores con el <i>joystick</i> . . . . .	121
A.4. Obtención de imágenes NDVI . . . . .	127
A.5. Código Arduino para la iteración 2 . . . . .	131
<b>B. Esquemáticos</b>	<b>151</b>
B.1. Iteración 1 . . . . .	153
B.2. <i>Joystick</i> . . . . .	155
B.3. Control remoto . . . . .	157
B.4. Iteración 2 . . . . .	159
<b>C. Planos</b>	<b>161</b>
C.1. Vista de planta (iter. 1) . . . . .	163
C.2. Vista de planta (iter. 2) . . . . .	165
C.3. Visión del robot (iter. 2) . . . . .	167
C.4. Soporte Raspberry Pi 3 Model B . . . . .	169
C.5. Soporte convertidor DC-DC . . . . .	171
C.6. Soporte Arduino Mega 2560 . . . . .	173
C.7. Tapa trasera para ultrasonidos HC-SR04 . . . . .	175
C.8. Tapa frontal para ultrasonidos HC-SR04 . . . . .	177
<b>Referencias y bibliografía</b>	<b>179</b>



# Índice de figuras

1.1.	Funcionamiento NDVI . . . . .	2
1.2.	Tipos de caminos . . . . .	4
1.3.	Vista local . . . . .	6
1.4.	Estructura mecánica inicial . . . . .	9
2.1.	<i>Pinout</i> del Arduino MEGA 2560 R3 . . . . .	15
2.2.	Motor de limpiaparabrisas tipo saturno . . . . .	16
2.3.	Interior de un motor limpiaparabrisas . . . . .	17
2.4.	Controlador de motores IMS-1 . . . . .	19
2.5.	Conexionado del controlador de motores . . . . .	20
2.6.	Brújula GY-271 - HMC5883L . . . . .	21
2.7.	Módulo de GPS GY-GPS6MV2 . . . . .	23
2.8.	<i>Pinout</i> del ESP8266 (ESP-01) . . . . .	26
2.9.	Sensor ultrasonido HC-SR04 . . . . .	27
2.10.	Test práctico del sensor HC-SR04 . . . . .	27
2.11.	Joystick Quickshot QS-203 de fábrica . . . . .	29
2.12.	Joystick adaptado . . . . .	30
2.13.	<i>Pinout</i> del Arduino Nano . . . . .	31
3.1.	Estructura electrónica de la iteración 1 . . . . .	36
3.2.	Movimientos de un robot controlado por acción diferencial . . . . .	37
3.3.	Diagrama de bloques del cliente UDP . . . . .	39
3.4.	Diagrama de bloques del servidor UDP . . . . .	39
3.5.	Algoritmo para evitar obstáculos de la iteración 1 . . . . .	41
3.6.	Cálculo del ángulo del obstáculo . . . . .	41
3.7.	Diagrama de acción de las ruedas en función de las coordenadas del joystick . . . . .	45
3.8.	Diagrama de bloques del cálculo de las coordenadas de diamante . . . . .	47

3.9. Espacio de diamante. Fuente: [23]. . . . .	49
4.1. Electrónica de la iteración 2 . . . . .	57
4.2. Estado del robot en la iteración 2 . . . . .	58
4.3. Estructura electrónica de la iteración 2 . . . . .	59
4.4. Raspberry Pi 3 Model B . . . . .	61
4.5. <i>Buck converter</i> a 5 V . . . . .	62
4.6. Cámara con sensor OV5647 . . . . .	63
5.1. Diagrama de bloques para la navegación . . . . .	70
5.2. Algoritmo para evitar obstáculos en la iteración 2 . . . . .	74
5.3. Diagrama de bloques del controlador PI discretizado . . . . .	88

# Índice de cuadros

2.1. Modo de control del IMS-1. Fuente: [10]. . . . .	19
3.1. Ejemplo de <i>string</i> codificada. . . . .	39
3.2. Movimientos de las ruedas según subplanos. . . . .	50
3.3. Casos extremos. . . . .	50
3.4. Presupuesto iteración 1 . . . . .	51
5.1. Asignación de pines según placa . . . . .	66
5.2. Resultados de la medida de los ultrasonidos sin saltar . . . . .	78
5.3. Resultados de la medida de los ultrasonidos salteados . . . . .	79
5.4. Presupuesto iteración 2 . . . . .	90



# Índice de códigos

2.1.	Ejemplo minimalista para el funcionamiento de los motores . . . . .	20
2.2.	Ejemplo minimalista para el funcionamiento de la brújula . . . . .	22
2.3.	Ejemplo minimalista para el funcionamiento del GPS . . . . .	24
2.4.	Ejemplo minimalista para medir distancias con un ultrasonido . . . . .	27
3.1.	Construcción del <i>string</i> a enviar . . . . .	39
3.2.	Cálculo del ángulo al que se encuentra el obstáculo . . . . .	42
3.3.	Cálculo de rumbos correctos . . . . .	42
3.4.	Navegación en espacios abiertos . . . . .	43
3.5.	Filtrado de valores analógicos de los ejes del <i>joystick</i> . . . . .	46
3.6.	Cálculo de las coordenadas de diamante . . . . .	47
5.1.	Configuración de los pines de control del Arduino . . . . .	66
5.2.	Programa bash para inicializar los GPIO . . . . .	67
5.3.	Configuración de alias . . . . .	68
5.4.	Programa Python para leer el puerto Serial del Arduino . . . . .	69
5.5.	Comprobación de que la señal GPS es correcta . . . . .	71
5.6.	Obtención de las coordenadas GPS . . . . .	71
5.7.	Configuración de los pines de los ultrasonidos . . . . .	75
5.8.	Lectura del <i>array</i> de ultrasonidos . . . . .	76
5.9.	Filtrado de las lecturas de los ultrasonidos . . . . .	77
5.10.	Comprobación de la existencia de obstáculos peligrosos . . . . .	79
5.11.	Cálculo del ángulo de rebote . . . . .	80
5.12.	Rotación del robot al rumbo dado . . . . .	81
5.13.	Comprobación de la visibilidad del objetivo . . . . .	82
5.14.	Funciones de movimiento del robot . . . . .	84
5.15.	Controlador PI para rotación pura . . . . .	87
A.1.	Programa principal de la iteración 1 . . . . .	99
A.2.	Programa principal para el control remoto con <i>joystick</i> . . . . .	117
A.3.	Programa para el control local de motores . . . . .	121

A.4. Programa para obtener imágenes NDVI usando OpenCV . . . . .	127
A.5. Programa principal de la iteración 2 . . . . .	131

# **Lista de abreviaturas**

<b>TFG</b>	Trabajo de Fin de Grado
<b>GPS</b>	Global Positioning System
<b>PWM</b>	Pulse Width Modulation
<b>NMOS</b>	Negative Metal Oxide Semiconductor
<b>RF</b>	Radio Frequency
<b>NMEA</b>	National Marine Electronics Association
<b>UTC</b>	Universal Time Coordinated
<b>UT</b>	Universal Time
<b>PCB</b>	Printed Circuit Board
<b>GPIO</b>	General Purpose In/Out
<b>ADC</b>	Analog to Digital Converter
<b>DAC</b>	Digital to Analog Converter
<b>NDVI</b>	Normalized Difference Vegetation Index
<b>ADSL</b>	Asymmetric Digital Subscriber Line
<b>AP</b>	Access Point
<b>UDP</b>	User Datagram Protocol
<b>DHCP</b>	Dynamic Host Configuration Protocol
<b>SBC</b>	Single Board Computer
<b>SO</b>	Sistema Operativo
<b>FFC</b>	Flexible Flat Cable
<b>ROS</b>	Robot Operating System
<b>CCD</b>	Charge-Coupled Device
<b>SIMO</b>	Single Input, Multiple Output
<b>SISO</b>	Single Input, Single Output
<b>IoT</b>	Internet of Things



*En agradecimiento a todos los que  
me han enseñado para llegar  
hasta aquí. En especial a mi  
familia por su apoyo  
incondicional.*



# **Capítulo 1**

## **Introducción**

### **1.1 Descripción del proyecto**

El objetivo global del proyecto que se presenta en este documento es el de crear un prototipo capaz de moverse de forma autónoma desde su ubicación inicial a un punto dado desde una interfaz web, para poder realizar las acciones que le sean encomendadas.

Este TFG pretende realizar una selección de los componentes electrónicos necesarios para que el vehículo pueda moverse de forma autónoma, así como su montaje e implementación en su controlador. Dado el carácter de prototipo del proyecto, para poder conseguir este objetivo es necesario pasar por un proceso iterativo de mejora continua retroalimentado por la experiencia obtenida de la versión anterior. Estos avances son documentados en esta memoria, presentando la descripción, el funcionamiento y los criterios de selección de cada módulo usado. También se aporta una propuesta para la implementación de cada iteración con las conclusiones y observaciones que aportan un análisis de los aciertos y errores que se han cometido.

#### **1.1.1 Motivación**

La idea nace de la necesidad de poder monitorizar y controlar el estado de una finca remota sin tener que acudir físicamente a ella. Para ello, se pretende desarrollar un prototipo modular, de bajo coste y que sea escalable para poder mejorar sus prestaciones en el futuro. Además de poder ser integrado junto con otros procesos automáticos que ya hay instalados.

El interés de este proyecto reside, en buena parte, en ser capaz de analizar con una cámara el estado de la finca, y más concretamente, los distintos cultivos para poder ver si se está regando correctamente, o si existe alguna plaga que requiera una acción *in-situ*.

Para ello, una de las técnicas que permiten observar dichos fenómenos está basado en la obtención de imágenes NDVI. Estas permiten comprobar el estado de salud de los cultivos que allí se encuentren [24]. El principio en el que se basa esta tecnología consiste en que, midiendo la cantidad de luz reflejada por la vegetación en cada zona del espectro, es posible determinar el estado de salud general de la vegetación. Esto se debe a que el crecimiento de la planta es proporcional a la cantidad de clorofila que produce. Como se refleja en la imagen 1.1, las plantas absorben la mayoría del espectro visible para convertirlo en comida por medio de la fotosíntesis. A diferencia del azul y el rojo, se observa que el color que más se refleja es el verde, y es por esto que la vegetación más sana es de este tono. Sin embargo, destaca que la mayor cantidad de reflexión se produce en la zona del infrarrojo cercano.

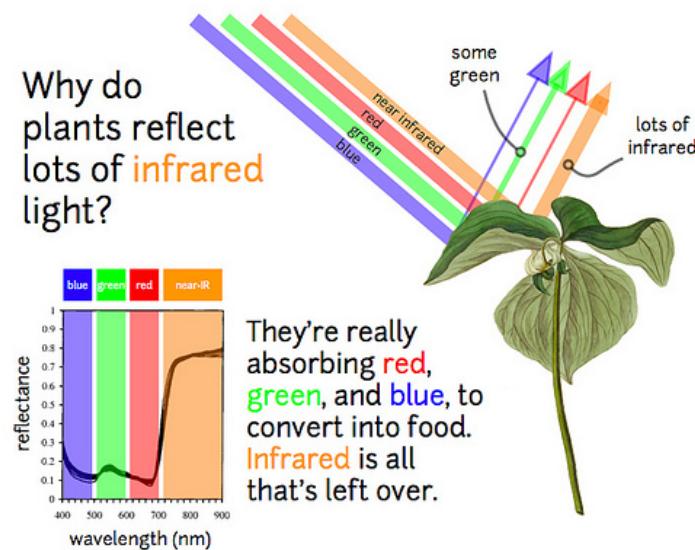


FIGURA 1.1: Funcionamiento NDVI. Fuente: publiclab.org

La idea final, entonces, consiste en filtrar la luz visible frente al infrarrojo cercano por medio de una cámara que sea capaz de capturarlo. Este método de análisis suele ser aplicado desde drones para controlar grandes extensiones de cultivos y bosques, obteniendo así una imagen general de las zonas más sanas. No obstante, la aplicación en

este proyecto tiene por objetivo ser más concreto hacia qué cultivo se quiere analizar, introduciendo un rover en la zona concreta deseada.

### 1.1.2 Descripción del lugar

La ubicación donde el vehículo desarrolla su trabajo es una finca situada en el valle de Güímar, en el sur de Tenerife. En este lugar destacan algunas características, como el viento o el calor, que son factores ambientales clave a tener en cuenta en el diseño del prototipo.

Esta finca está dividida en diversos sectores en función del tipo de cultivo existente. De esta manera, se puede encontrar tanto árboles frutales como naranjeros y mangos, como otras plantas de menor altura como vid. Además, en la zona de cultivo el robot debe convivir con gallinas y otros animales sueltos, por tanto, ser capaz de detectar este tipo de obstáculos y moverse entre medio de las hileras sin afectar al entorno. Actualmente existe una separación de 1.40 m entre las hileras de parras y 3.5 m entre los árboles, por lo que un límite superior para el ancho del vehículo viene representado por estos datos.

En cuanto a otras condiciones operativas, el desnivel que presenta todo el área es poco pronunciado, con un terreno despejado en los caminos y con pocos obstáculos. Así, el máximo desnivel existente posee una pendiente de 11°, aproximadamente. También está dividida en dos partes, una de cultivo, y por lo tanto más complicada para ser transitada, y otra con un firme más uniforme (ver figura 1.2). Este último es el lugar de partida y de reposo del vehículo, especialmente por la necesidad de disponer de una toma de corriente y un lugar protegido para su almacenamiento en el futuro.

Es importante comentar que en la actualidad no existe posibilidad de una conexión de ADSL o fibra que permita el acceso a internet. Sin embargo, para poder utilizar otras herramientas que ya están implementadas sí que existe un móvil con una tarjeta SIM configurado como punto de acceso. Esto permite que algunos dispositivos de IoT puedan enviar y recibir datos de un servidor y ejecutar las acciones deseadas. Por tanto, para el éxito de este proyecto es vital que exista esta red inalámbrica y que además sea fiable.

Dado que la cobertura del AP de un móvil no posee bastante alcance, en un futuro se pretende configurar un *router* con OpenWRT (o similar) como repetidor WiFi, o cablear



(A) Camino tipo 1.



(B) Camino tipo 2.

FIGURA 1.2: Tipos de caminos

varios puntos que extiendan la cobertura a toda la zona. No obstante, estas consideraciones quedan, de momento, fuera del alcance de este TFG.

## 1.2 Estudio previo y bases de diseño

Aunque el diseño mecánico no sea objeto principal de este trabajo, en base a lo descrito en la sección 1.1.2, y de las situaciones que tendrá que afrontar, se pueden establecer unos criterios mínimos de para que sea útil y operativo en su misión.

Durante sus tareas el vehículo tiene que poder girar sobre sí mismo para ganar maniobrabilidad, en especial cuando esté en un espacio reducido, como dentro de una hilera. Además, en esta zona, el terreno está compuesto por tierra y piedras de pequeño tamaño, por lo que para sortearlas se debe usar un diámetro adecuado para las ruedas, y

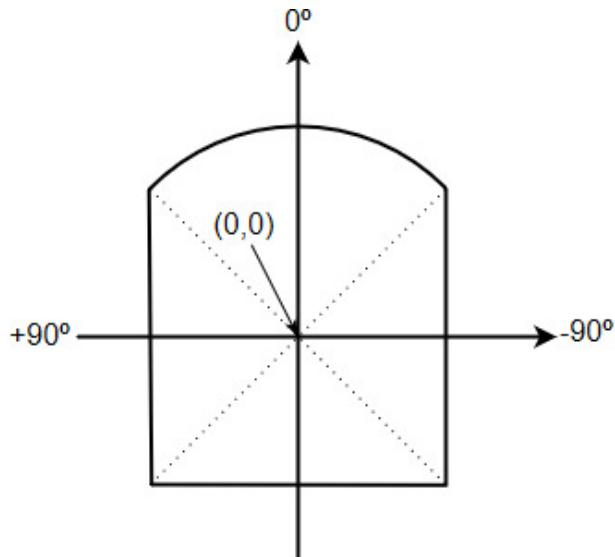
unos motores con el par suficiente para mover todo el peso. Por tanto, desde este punto de vista, parece razonable diseñar un vehículo que se mueva por acción diferencial de ambas ruedas. Esta morfología es típica de los robots de interiores dado que ofrecen bastante maniobrabilidad y capacidad para moverse en lugares de espacio reducido. No obstante, este proyecto desarrolla su actividad en un lugar a la intemperie, como se ha descrito en la sección 1.1.2.

Por un lado, el primer problema al que se enfrenta un robot móvil es el de la navegación. Este concepto está relacionado con la idea de conocer la ubicación actual del robot, idear, y seguir una ruta. Por tanto, está compuesto de dos componentes diferenciados: la localización y la planificación [4]. Conceptualmente, esta última idea consiste en encontrar una trayectoria óptima y libre de obstáculos que permite una navegación segura. En este proyecto este punto no resulta un problema considerable dado que el camino que se le indica al robot es conocido y seguro.

Sin embargo, para que pueda navegar de forma autónoma, hay que dotarlo de los sensores necesarios para detectar los obstáculos cercanos y evitar que golpee con ellos, comandando los motores desde un cerebro central. Esto cobra especial relevancia si tenemos en cuenta el primer problema de la robótica móvil, la localización. Este concepto se relaciona con saber dónde se encuentra el robot con respecto al entorno de forma precisa. Durante el desarrollo de este TFG, la aproximación a este problema se intenta abordar de manera puramente sensorial por medio de sistemas GPS. Por tanto, para saber a dónde se dirige solo hace falta añadir una brújula como sensor necesario para cubrir todas las variables. Es decir, estos dos módulos pretenden solventar el problema de que existan tres grados de libertad para poder posicionar perfectamente un robot móvil. Estos son las coordenadas  $x$  e  $y$  en el espacio, y  $\theta$  que es la orientación. Las primeras se pueden medir en coordenadas decimales, pero las segundas poseen un rango que usualmente va de 0 a  $360^\circ$ . Por tanto, es necesario adaptar los datos de rumbo para que siempre estén en dicho rango. Siguiendo el convenio que se suele adoptar, los  $0^\circ$  muestran el norte, los  $90^\circ$  el este, y así sucesivamente hasta completar una vuelta.

Las ideas mostradas anteriormente corresponden a la ubicación de un robot con respecto al mundo global, sin embargo, este también posee unas coordenadas propias en su mundo local. La suposición que se puede tomar en este punto es que, idealmente, el origen de coordenadas local debe estar ubicado en el centro geométrico del vehículo. No obstante, este tipo de detalles están influenciados por la posibilidad real de ubicar

un sensor en ese punto. Partiendo de la idea de que se cumplen las premisas anteriores, se puede definir desde el punto de vista del robot que un rumbo de  $0^\circ$  corresponde al eje Y. Es decir, que uno de  $90^\circ$  se corresponde a su izquierda, y uno de  $-90^\circ$  sería hacia la derecha (ver figura 1.3).



---

FIGURA 1.3: Vista local

Por otro lado, puesto que el robot necesita recibir unas órdenes, tiene que estar conectado de forma inalámbrica a una red local que le de salida a internet, de donde recibirá los comandos. Aunque es deseable que esta conexión tenga poca latencia, dado el alcance limitado de este TFG, esto no representa una necesidad por el momento. Esto es debido a que la información que manejada no es muy pesada, y tampoco necesita ser en tiempo real.

Se sabe, además, que en un futuro se pretende utilizar el vehículo para tomar imágenes de la finca, por lo que resulta lógico proveer al robot con una cámara que permita realizar tal operación. De cara al futuro, también es importante garantizar que se puedan seguir añadiendo mejoras en el prototipo, dejando el espacio adecuado para ello.

Desde el punto de vista de la mecánica, el centro de gravedad debe estar ubicado en el punto más bajo posible, y con una distancia entre ejes que garantice la estabilidad en los giros cerrados.

Por último, teniendo en cuenta las distancias que deberá recorrer, es deseable que tenga un mínimo de autonomía que permita ir y volver al lugar de carga. No obstante, este

último requisito no se tiene que cumplir, necesariamente, en la fase de prototipado en la que se encuentra.

- Maniobrabilidad. Posibilidad de rotar sobre si mismo.
- Ocupar poco espacio (<1 m de radio).
- Posibilidad de transitar por tierra.
- Detectar y evitar obstáculos cercanos.
- Poder orientarse para encontrar el camino.
- Posibilidad de sacar imágenes.
- Comunicación inalámbrica a internet.
- Estabilidad.
- Espacio para futuras mejoras y/o implementaciones.

Además, es deseable realizarlo modular, al menor coste e impacto ecológico posible.

### 1.2.1 Cinemática básica

Desde el punto de vista cinemático, este robot es no holonómico, es decir, no se puede desplazar en cualquier dirección del espacio. Esto significa que no es capaz de moverse lateralmente, pero sí puede rotar sobre sí mismo, o desplazarse con y sin rotación. Por tanto, para realizar un movimiento lateral debe combinar los movimientos anteriores.

Los movimientos descritos anteriormente vienen dados por la velocidad angular de las ruedas, y por la existencia de una diferencia de avance entre ambas. Esto significa que, para sufrir rotación, una rueda debe girar más rápida que la otra, mientras que, si ambas se mueven igual, el vehículo se estaría trasladando sin cambiar su orientación. Suponiendo que no exista deslizamiento con el suelo, el modelo matemático queda en función de las expresiones 1.1 y 1.2.

$$v = \frac{v_i + v_d}{2} = \frac{r(\omega_i + \omega_d)}{2} \quad (1.1)$$

$$\omega = \frac{v_d - v_i}{L} = \frac{r(\omega_d - \omega_i)}{L} \quad (1.2)$$

donde  $L$  es el ancho del eje,  $r$  el radio de las ruedas,  $\omega$  la velocidad angular y  $v$  la lineal.. A partir de las ecuaciones anteriores se puede deducir la pose del robot con las ecuaciones 1.3, 1.4 y 1.5. Se observa que esta varía con la velocidad lineal y angular que lleve el vehículo, y el incremento de tiempo.

$$x(t) = \int_{t_1}^{t_2} v \cos(\theta) dt \quad (1.3)$$

$$y(t) = \int_{t_1}^{t_2} v \sin(\theta) dt \quad (1.4)$$

$$\theta(t) = \int_{t_1}^{t_2} \omega dt \quad (1.5)$$

donde  $x$ ,  $y$  y  $\theta$  representan cada grado de libertad. De forma análoga, las ecuaciones anteriores pueden ser representadas en incrementos de tiempo, como se muestra en las ecuaciones 1.6, 1.7 y 1.8.

$$x(k+1) = x(k) + v(k) \cdot \cos(\theta(k)) \cdot \Delta t \quad (1.6)$$

$$y(k+1) = y(k) + v(k) \cdot \sin(\theta(k)) \cdot \Delta t \quad (1.7)$$

$$\theta(k+1) = \theta(k) + \omega(k) \cdot \Delta t \quad (1.8)$$

Esta última manera de expresarlas resulta de utilidad a la hora de implementarlas en un controlador digital para hacer una previsión de la pose. Por tanto, el conjunto de ecuaciones anteriores sirven para representar el robot en una situación ideal en la que no existen perturbaciones como el eventual deslizamiento de las ruedas, entre otras.

### 1.3 Punto de partida

A parte de las condiciones del lugar comentadas anteriormente en la sección 1.1.2, el proyecto desde donde se empieza parte de un diseño básico que responde, en buena

parte, a las condiciones descritas en la sección 1.2. En este punto, lo único montado es una estructura con una base rectangular de 55 cm de ancho por 70 cm de largo, con dos ruedas de bicicleta de 26 pulgadas formando un eje de 38 cm de ancho. Para que sea estable, además, se cogió la horquilla de la misma bicicleta y se le colocó una rueda de menor diámetro para que apoyara sobre tres puntos, de tal manera que esta última quedaría loca.



---

FIGURA 1.4: Estructura mecánica inicial



# **Capítulo 2**

## **Iteración 1**

El objetivo de este capítulo es realizar una propuesta de los componentes necesarios para el correcto funcionamiento del robot. Para ello, la implementación se compone de distintos módulos que se coordinan entre sí para ofrecer la información necesaria para poder navegar. Este capítulo hace especial hincapié en la descripción del *hardware*, así como de algunos ejemplos minimalistas de cómo integrarlos desde el punto de vista del *software*. Esto incluye también la justificación del conexionado necesario que recogido en el anexo B.1. En definitiva, la intención es ofrecer una vista previa de qué funciones se implementan, por qué, y las ventajas de un módulo frente a otras soluciones.

### **2.1 Electrónica general**

En una primera versión, la electrónica ha sido dividida en módulos que son controlados desde un mismo controlador que actúa como cerebro. Este contiene el algoritmo para evitar los obstáculos que, en función de la información recibida de los sensores, comanda a los motores la acción necesaria para evitar el impacto, quedando un sistema en lazo cerrado.

#### **2.1.1 Baterías**

La alimentación de un circuito eléctrico es aquella que aporta una diferencia potencial entre dos puntos interconectados entre sí. Esta premisa es necesaria para que funcione, y aunque usualmente esto se consigue obteniendo la energía desde cualquier toma de

corriente común, por la naturaleza móvil del proyecto, esta opción no es realizable. Por tanto, la solución consiste en obtener este recurso de una fuente local que almacene y distribuya la electricidad según se demande.

Esta es la función de las baterías, alimentar eléctricamente todo el circuito del vehículo. Desde el punto de vista conceptual, resulta ideal montar baterías que tengan, entre otras prestaciones, una densidad energética más alta, esto es, una ratio  $\text{kJ}/\text{kg}$  máxima. No obstante, debido al carácter de prototipo de este proyecto, se ha optado por utilizar una reciclada de una moto a la que ya no le servía como alimentación de arranque. En concreto, el modelo utilizado es el GS-GT12A-BS fabricado por GS [8]. Esta batería es de plomo ácido, y posee una capacidad de 10 Ah a un voltaje aproximado de 12 V. La ventaja de montar esta batería no solo tiene relación directa con su voltaje nominal, sino con su reducido peso y la facilidad para adaptarla al espacio físico disponible. Por un lado, aunque seguramente existan otras químicas más adecuadas para estas utilidades, para la fase de desarrollo resulta suficiente para realizar las pruebas. Por otro, es posible que en un futuro sea necesario instalar una de mayor capacidad para alcanzar una autonomía mayor.

El modelo utilizado está compuesto de seis celdas que aportan 2.1 V cada una. Entonces, si estas se colocan en serie, se alcanza el voltaje nominal típico de 12.6 V en circuito abierto. Cabe destacar este último detalle ya que este nivel de voltaje representa también el estado de carga de la misma. Otra característica importante es que la mayoría de los componentes del vehículo, aunque nominalmente trabajen a 12 V, en realidad tienen una tolerancia que hacen compatible este nivel. No obstante, experimentalmente se ha podido comprobar que el voltaje de la batería cae a un valor máximo de 12.3 V cuando se cierra el circuito. Sin embargo, este valor disminuye algo más cuando aumenta la demanda de corriente de los motores, que son el módulo que más requiere (ver 2.2.1).

Otra ventaja de esta decisión reside en que es posible reducir los costes y aumentar la ecología al reciclar aquellas baterías que ya no sirven para arrancar un vehículo de combustión interna, pero que, sin embargo, tienen cabida para satisfacer las necesidades de un robot móvil.

### 2.1.2 Cargador

Para poder utilizar la batería en numerosas ocasiones es necesario dotarla de carga previamente, para lo que hay que provocar la reacción química en el otro sentido, aportando un voltaje mínimo. Para este proyecto, se ha utilizado un cargador provisional reciclado de la fuente de alimentación de otro equipo de 17 V.

Aunque, como se comentó en la sección 2.1.1, el voltaje nominal de las baterías es de 12.6 V en circuito abierto, este tipo de química necesita de un voltaje estable de, al menos, 2.3 V por celda para poder realizar la carga. Sin embargo, si se conectara esta fuente de alimentación directamente, es bastante probable que la batería se dañe por sobrevoltaje, ya que la caída de tensión por celda sería de 2.83 V aproximadamente. Para evitar este problema, se ha optado por utilizar un LM317T como regulador de tensión ajustable a la salida de la fuente de alimentación, situando el nivel en 13.8 V en circuito abierto (ver anexo B.1). Además, este regulador suministra una corriente de salida que está limitada internamente a 2.2 A [1], que es un valor suficiente para realizar la carga.

Dado que es un regulador ajustable, es necesario utilizar dos resistencias para polarizarlo al nivel adecuado, siguiendo el *datasheet* del fabricante. De esta forma, se instaló un R1 y R2 de  $270\Omega$  y  $2700\Omega$ , respectivamente. Además, como el *dropout voltage* de este regulador está por debajo de la caída que regula, se aporta cierta estabilidad al conjunto del cargador. En pruebas experimentales se ha comprobado que funciona correctamente aunque el nivel de voltaje regulado varía con la demanda de corriente de la batería.

### 2.1.3 Controlador

El módulo encargado de coordinar las acciones a partir de la información recibida de los sensores es el controlador. Para este proyecto es necesario que sea capaz de leer bastantes entradas para luego escribir a los actuadores, dada su complejidad. Por este motivo, el módulo elegido es el Arduino MEGA 2560, no solo por su bajo coste sino también por su facilidad de uso y extensión. Este posee un ATmega 2560, un microcontrolador de 8 bits con una frecuencia de hasta 16 MHz fabricado por Atmel. Siguiendo el criterio de escalabilidad del proyecto, una ventaja está en que permite implementar

el protocolo I<sup>2</sup>C, así como varios PWM y más de otras 20 entradas/salidas digitales que se suman a las analógicas.

- Alimentación con regulador a 5 V.
- 4x Puertos Serial.
- 2x I<sup>2</sup>C.
- 15x Conexiones PWM dedicadas.
- 16x Entradas analógicas.
- 15x I/O digitales.
- ADC de 10 bits (1024 niveles) y DAC de 8 bits.

Todas estas salidas trabajan a 5 V por lo que será necesario adaptarlas con algún circuito intermedio en aquellos módulos que trabajen a distinto nivel.

Es importante destacar que es recomendable que el controlador solamente maneje hasta un máximo de 20 mA por pin. Por tanto, este modulo no suministra potencia a ninguna parte del robot, únicamente señal. La solución es, entonces, enviar señales de control a los módulos que sí que contienen componentes que son capaces de manejar más potencia y pueden suministrar la corriente necesaria.

Este controlador forma un sistema que lee y ejecuta una serie de instrucciones de forma lineal, programadas en Arduino, un lenguaje bastante parecido a C/C++. Estas placas tienen un esquema básico de funcionamiento en el que se pueden distinguir tres partes: la zona de declaración de variables y/o funciones; el *setup*, donde se pueden programar instrucciones que solo se ejecutan una vez durante el inicio; y el *loop*, que es el bucle principal encargado de ejecutar las instrucciones programadas de forma iterativa hasta que se desconecte de la corriente.

Con esta estructura es posible plantear una programación que inicialice los módulos en el *setup*, y luego simplemente haga las acciones necesarias en el *loop*. De lo contrario los módulos se inicializarían cada vez que acaba de iterar un ciclo, haciendo que funcione de una manera muy poco eficiente.

Dado que este microcontrolador ejecuta una sola lista de instrucciones de principio a fin, no existe posibilidad de concurrencia real que permita manejar varios *threads*.

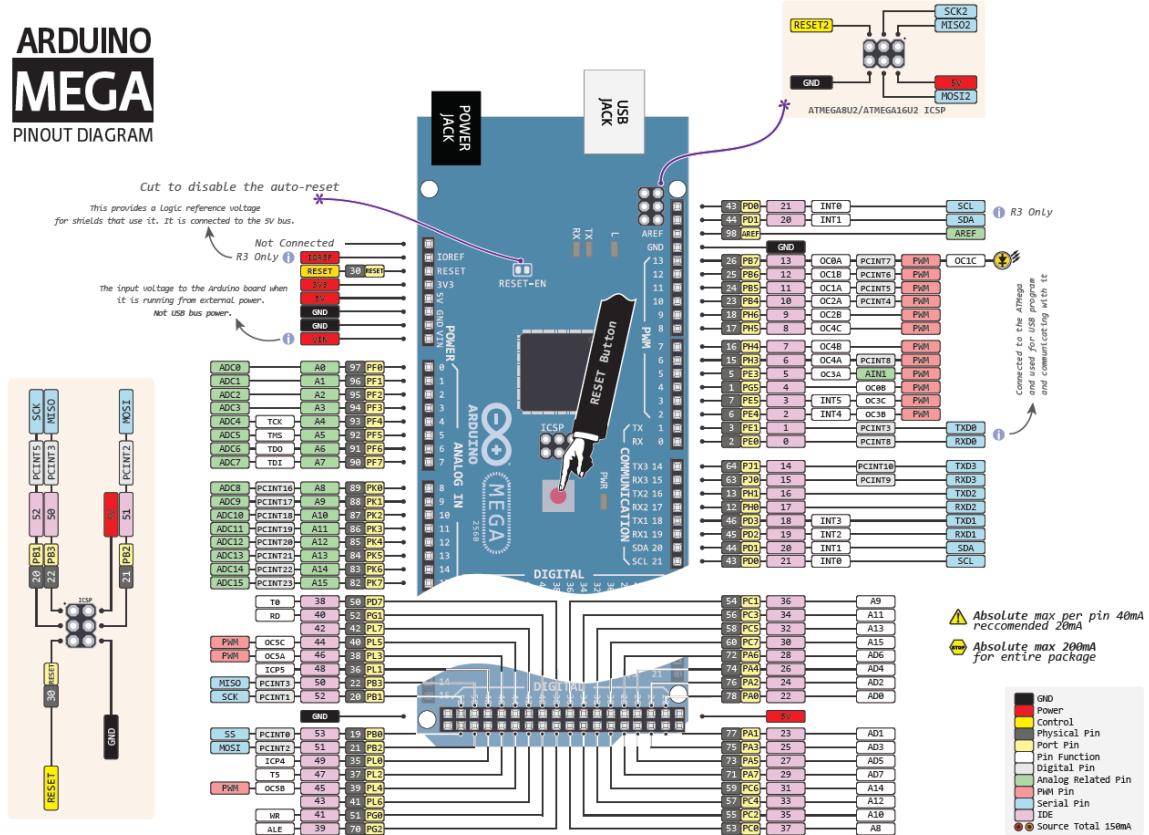


FIGURA 2.1: Pinout del Arduino MEGA 2560 R3. Fuente: [makerstore.com.au](http://makerstore.com.au)

Para obtener algo cercano a esta idea, seguramente habría que recurrir a métodos de programación con interrupciones, pero que están fuera del objetivo de este TFG.

## 2.2 Actuadores

Una característica fundamental de un robot móvil es la de ser capaz de desplazarse de un lugar a otro. Para cumplir este cometido se utilizan actuadores que son aquellos que permiten la movilidad física del vehículo. En este caso, puesto que hace falta bastante agilidad y la posibilidad de pivotar sobre sí mismo, la solución adoptada pasa por implementar dos motores controlados de forma independiente.

Una característica propia de estos módulos es que demandan una gran cantidad de corriente, y por tanto, consumen bastante energía que se transforma en par y finalmente

en un movimiento efectivo.

### 2.2.1 Motores

Los únicos actuadores que se utilizan en este proyecto son los motores. Para realizar esta función se han reutilizado unos de limpiaparabrisas, en concreto el modelo TGE589A fabricado por Magneti Marelli, para los Fiat Panda. Una ventaja que presentan estos motores para este montaje es que son de corriente continua y del voltaje de la batería. De esta manera se pueden controlar de manera más sencilla ya que el circuito necesario para ello es un puente H controlado por PWM (ver 2.2.1.1). Además, poseen en su interior un tornillo sin fin que aporta el par necesario para mover todo el vehículo, mientras sacrifica velocidad (ver figura 2.3).

Dado que los motores que seleccionados son reciclados, no se ha podido encontrar una hoja de datos con las especificaciones exactas. Sin embargo, estos consumen entre 0.9 A hasta 4 A, en función del par que esté desarrollando [20]. Además, de acuerdo con esta misma fuente, el par motor puede alcanzar los 18.3 N m a altas velocidades. Experimentalmente se ha podido comprobar que en llano cada motor consume 0.9 A aproximadamente, llegando hasta 8 A cuando se le aplica una carga. Aunque algunas fuentes ubican la intensidad de cortocircuito de estos motores en 12 A, esto no ha podido ser comprobado.



---

FIGURA 2.2: Motor de limpiaparabrisas tipo saturno. Fuente: [rctankcombat.com](http://rctankcombat.com)

En el tipo de motor ilustrado en la figura 2.2, las bobinas que polarizan el motor para que gire se encuentran en el cilindro negro. No obstante, el conexionado se realiza por medio de unos bornes ubicados en la zona plástica anterior que se encuentra alineada

con el rotor. En el caso del TGE589A existen cuatro bornes además de la tierra, que está conectada a la carcasa. El conexionado es relativamente sencillo, solamente es necesario conectar el positivo del controlador al borne INT, y luego el otro polo a la carcasa.



(A) Engranajes.

(B) Circuito interior.

FIGURA 2.3: Interior de un motor limpiaparabrisas

Como se comentó anteriormente, dado que no se encuentran hojas de datos para este motor, se deben hacer medidas experimentales para calcular algunos parámetros básicos. La velocidad angular del rotor sin carga es de aproximadamente 66.67 rpm. Sin embargo, este valor no se transmite directamente a las ruedas, sino que se utiliza una cadena que con dos piñones de distinto tamaño transmite esta potencia. El uso de este tipo de mecanismos permiten sacrificar velocidad lineal en pro de aumentar el par proporcionalmente.

Partiendo de esta configuración y de los datos teóricos encontrados, es posible calcular el peso máximo que es capaz de llevar el robot. Este se obtiene del estudio de las fuerzas que sufre el motor extendiéndolo al otro punto donde ocurre el trabajo. Si suponemos un sistema ideal sin pérdidas en el que ningún sistema intermedio pierde potencia entre el rotor y la rueda, para el caso de un solo motor, el peso máximo teórico que puede mover se obtiene aplicando la ecuación 2.1.

$$P_{max} = \frac{T_{rotor}}{R_{rotor}(\mu \cdot \cos(\alpha) + \sin(\alpha))} \quad (2.1)$$

donde  $R_{rotor} = 0.015$  m en este montaje. Con el dato anterior el cálculo de la masa a la que corresponde dicha fuerza se obtiene al aplicar la ecuación 2.2.

$$m_{max} = \frac{P_{max}}{g} \quad (2.2)$$

Aplicado a este caso práctico, se obtiene que la masa máxima que puede mover cada motor con esta configuración, suponiendo que cada uno pueda ejercer 18.3 N m como mucho, es de 87.94 kg. Dado que el montaje planteado se compone de dos motores, la masa máxima teórica es de 175.88 kg para una pendiente de 45°.

Por tanto, en teoría, estos motores ejercen un par más que suficiente para realizar el trabajo. De hecho, también podemos ver las ecuaciones anteriores a la inversa y calcular el par que necesita realizar cada motor para mover un peso dado. Visto de esta manera, la ecuación 2.1 queda en la forma mostrada en la ecuación 2.3.

$$T_{rotor} = \frac{R_{rotor} \cdot P(\mu \cdot \cos(\alpha) + \sin(\alpha))}{2} \quad (2.3)$$

Aplicando los datos pertinentes se obtiene que cada motor ejerce 1.04 N m de par para mover el vehículo en una pendiente de 45° con coeficiente de rozamiento máximo.

### 2.2.1.1 Controlador de motores

Con el objetivo de obtener una respuesta personalizada de los motores es necesario implementar un controlador en los motores. Este módulo es el que suministra la potencia al motor obedeciendo las órdenes del controlador. En este caso se ha elegido el IMS-1 en su versión 2 [10]. Este trabaja a un voltaje variable entre 3 y 12 V, hasta un máximo de 15 V, y puede suministrar hasta 50 A. Esta elección fue escogida en vistas a garantizar la escalabilidad del proyecto al menor coste por lo que si en algún momento se deseara cambiar los motores por unos de mayor potencia, no habría que cambiar este componente. Fundamentalmente este módulo se compone de un puente H con cuatro NMOS que se excitan con la señal PWM del controlador (ver sección 2.5.2).

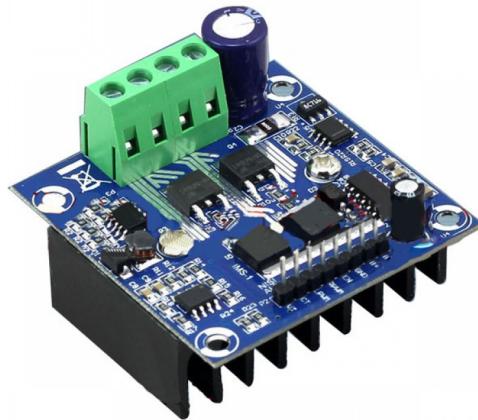


FIGURA 2.4: Controlador de motores IMS-1. Fuente: thanksbuyer.com

Gracias a los aspectos constructivos descritos anteriormente, este controlador resulta adecuado para el control de los motores de corriente continua. Este permite la movilidad de las ruedas en ambos sentidos, además de variar su velocidad en función de la señal comandada. A efectos prácticos, esta modulación permite suministrar un voltaje medio variable que se traduce en una variación proporcional de la velocidad angular.

De acuerdo con el *datasheet*, se ha implementado el modo de control 1 en el que el PWM se aplica a la señal *FORWARD*, y *REVERSE*, en vez del *ENABLE*. De forma esquemática, la tabla de control de los motores queda representada en la tabla 2.1.

	<b>Forward</b>	<b>Reverse</b>	<b>Brake</b>	<b>Close</b>
RPWM	1	0	0	X
LPWM	0	1	0	X
EN	1	1	1	0

CUADRO 2.1: Modo de control del IMS-1. Fuente: [10].

En cuanto al conexionado, el controlador sigue el modo 2 especificado en el *datasheet*, de tal manera que los pines de PWM utilizados son los correspondientes a LPWM y RPWM.

La implementación en el Arduino se realiza con la librería proporcionada por el fabricante para este controlador [7], pero modificada para ajustarse al *setup* implementado en el robot. Estos ajustes tienen que ver con asignaciones de pines, principalmente.

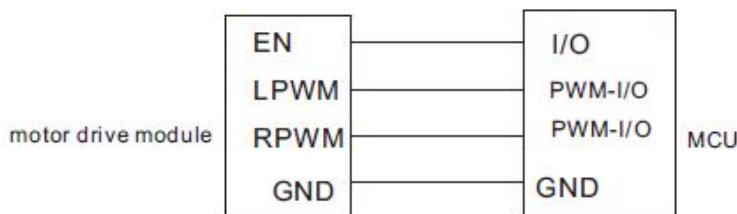


FIGURA 2.5: Conexionado del controlador de motores. Fuente: [10].

El comando enviado a los motores desde cada controlador es un valor proporcional a la velocidad angular asignada al mismo, y por tanto, también a la lineal.

Con el objetivo de ilustrar el funcionamiento de esta librería, el código 2.1 muestra un ejemplo minimalista que mueve los dos motores hacia delante a máxima velocidad.

```

1 #include <MOTOR.h> // Controlling motors with IMS-1 driver modules
2
3 void setup() {
4     // Initializing motors drivers
5     motor.begin();
6 }
7
8 void loop() {
9     // Move both motors full speed forward
10    motor.set(A, 255, FORWARD);
11    motor.set(B, 255, FORWARD);
12    motor_delay(10);
13 }
```

CÓDIGO 2.1: Ejemplo minimalista para el funcionamiento de los motores

## 2.3 Sensores

Aparte de los módulos encargados de realizar cambios en la pose del vehículo, es necesario obtener datos actualizados del estado del mismo cada cierto intervalo de tiempo.

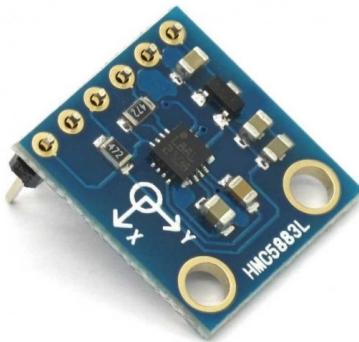
Los módulos que realizan esta actualización son los sensores, y constituyen una parte clave del robot. Estos permiten conocer el estado, permitiendo cerrar el lazo de re-alimentación del sistema entero para que pueda ser controlado. Por tanto, no solo es necesario que la información que aporten sea precisa, sino que además la ubicación física de los mismos es clave para que desempeñen su función.

Es también característico de este tipo de módulos una demanda de corriente bastante pequeña, ya que lo único que se transmite es información en forma de niveles de voltaje entre el controlador y el propio sensor.

### 2.3.1 Brújula

Para solucionar el problema de no conocer los valores de cada uno de los grados de libertad de la pose del vehículo es necesario utilizar una brújula. Estos módulos permiten orientarse en el espacio global por medio de la medición del campo magnético terrestre.

En el caso del robot, el módulo utilizado es el GY-271 que monta el HMC5883L [2], un magnetómetro de tres ejes ubicado en la parte superior del mismo. Como es lógico pensar, cualquier elemento metálico que sea capaz de alterar el campo magnético colindante es susceptible de generar una medida errónea en este sensor. Por tanto, el uso de este tipo de sistemas hacen necesario que se ubiquen en lugares especiales alejados de cualquier otro elemento metálico cercano para garantizar que las lecturas del rumbo son correctas.



---

FIGURA 2.6: Brújula GY-271 - HMC5883L. Fuente: njuskalo.hr

Dado que el campo magnético varía según la ubicación terrestre, es necesario calibrar el módulo al lugar geográfico en donde se encuentra. Para ello, la librería desarrollada por sleemanj que se ha decidido utilizar, HMC5883L\_Simple [21], ya posee funciones para esto (ver código 2.2).

---

```
1 #include <Arduino.h> // Libraries for compass HMC5883L
2 #include <Wire.h>
3 #include <HMC5883L_Simple.h>
4
5 HMC5883L_Simple Compass; // Create a compass
6
7 void setup() {
8     // Initializing compass
9     Wire.begin();
10    Compass.SetDeclination(-5, 1, 'W'); //La Laguna
11    Compass.SetSamplingMode(COMPASS_SINGLE);
12    Compass.setScale(COMPASS_SCALE_130);
13    Compass.setOrientation(COMPASS_HORIZONTAL_X_NORTH);
14 }
15
16 void loop() {
17     float heading = Compass.GetHeadingDegrees(); //Calculate heading
18 }
```

---

CÓDIGO 2.2: Ejemplo minimalista para el funcionamiento de la brújula

Esta calibración es necesaria para solventar el problema de que el norte magnético no coincide con el geográfico. Este desfase que existe entre uno y otro, y varía con el tiempo, recibe el nombre de declinación magnética. Para obtener el valor adecuado para el punto geográfico donde se desarrolla la actividad se ha utilizado la página <http://www.magnetic-declination.com/>. Después, usando las funciones facilitadas por dicha librería se han ajustado los parámetros a los datos locales. SetSamplingMode() es una función que permite configurar la brújula en modo sencillo en el que solo realizará una medición cuando el Arduino lo solicite. Luego, el parámetro SetScale() se configuró para que no generara demasiado ruido en las lecturas, de acuerdo con lo descrito por el desarrollador de la librería. Por último, para decirle al módulo la orientación física en la que está, se utiliza SetOrientation().

Cabe comentar que dos de los ejes del *compass* deben estar lo más paralelos a la superficie terrestre posible. En este caso, dejando la cara de componentes del módulo mirando hacia arriba, este devuelve  $0^\circ$  cuando el eje  $x$  de la serigrafía apunta al norte magnético. Por tanto, este eje se ubica apuntando hacia el frente del vehículo.

En cuanto a la comunicación con el controlador, este módulo utiliza el protocolo I<sup>2</sup>C (ver 2.5.1) que está ya programado en la función `GetHeadingDegrees()`. Además, cabe comentar que, aunque el módulo HMC5883L funciona con un nivel de 3.3 V, es posible conectarlo directamente al Arduino ya que en la variante que se está montando ya existe un regulador de tensión que adapta las señales a este voltaje.

### 2.3.2 GPS

Una vez se conoce la orientación del vehículo las otras dos variables necesarias para conocer la pose del mismo se resuelven con un módulo GPS que permite ubicar el robot en el mundo global. Para obtener esta información un sistema GPS triangula la posición de acuerdo con la distancia a los satélites cercanos. Es decir, este método ofrece un desempeño mejor en lugares abiertos y donde existan condiciones atmosféricas favorables, ya que la propia atmósfera afecta a estas señales.

En este caso, el módulo elegido es el GY-GPS6MV2. Se compone fundamentalmente de un chip NEO-6 [16] con algunos reguladores para adaptar los niveles de tensión, ya que funciona a 3.3 V, y una antena RF que recibe la señal de los satélites. Aunque el NEO-6 se puede comunicar por medio de diversas interfaces, la PCB sobre la que está montado ofrece comunicación Serial por medio de los pines de RX y TX. Por ello, y dado que el MEGA posee numerosas puertas dedicadas con esta interfaz, este es el método de comunicación elegido.



FIGURA 2.7: Módulo de GPS GY-GPS6MV2. Fuente: dx.com

En cuanto a la implementación del módulo en el Arduino, la librería usada se llama TinyGPS [14]. Esta permite obtener los valores de GPS por serial y guardarlos en variables independientes para poder realizar los cálculos dentro del programa. Resulta importante destacar que el GPS utiliza el estándar NMEA0183, desarrollado por NMEA, en el que se reciben unas sentencias codificadas con la información. Es por esto por lo que la librería resulta de gran ayuda para obtener los valores de latitud, longitud, etc.

Con el objetivo de ilustrar estas sentencias, podemos tomar el siguiente ejemplo extraído del documento de Parallax [17]:

\$GPRMC,032606,A,3410.2358,N,11819.0865,W,0.0,207.2,180211,13.5,E,A\*32

Como se puede apreciar, la primera cadena de caracteres hace referencia al comando estándar que se manda, y luego se acompaña con la información complementaria separada por comas.

- Hora UTC: 03 horas, 26 minutos, 06 segundos.
- A = OK, V = Warning.
- Latitud: 34 grados, 10.2358 minutos, Norte.
- Longitud: 118 grados, 19.0865 minutos, Oeste.
- Velocidad: 0.0 nudos.
- Dirección de la brújula: 207.2 grados.
- Fecha UT: 18/02/2011.
- Declinación magnética: 13.5 grados Este.
- Checksum.

Aunque los valores de latitud y longitud en este último ejemplo se muestran en grados y minutos, es habitual abandonar esta expresión para utilizar grados decimales en su lugar. Esta es la forma obtenida en el código 2.3 donde se muestra un ejemplo minimalista para guardar la latitud y la longitud del robot como *float*, junto con la antigüedad de la lectura con respecto al valor anterior. Esto último permite determinar si un valor es actualizado o no.

---

<sup>1</sup> `#include <TinyGPS.h>`

<sup>2</sup>

```
3 float latitude, longitude;
4 unsigned long fix_age;
5
6 // Create a gps object
7 TinyGPS gps;
8
9 void setup(){
10     Serial.begin(115200);
11
12     // Initialize GPS serial communication
13     Serial1.begin(9600);
14 }
15
16 void loop(){
17     while (Serial1.available()){
18         int c = Serial1.read();
19         if (gps.encode(c)){
20             gps.f_get_position(&latitude, &longitude, &fix_age);
21         }
22     }
23 }
```

---

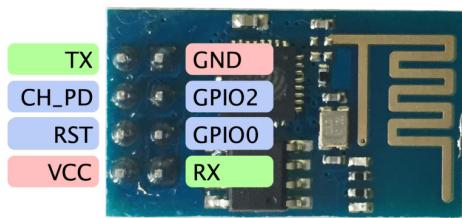
CÓDIGO 2.3: Ejemplo minimalista para el funcionamiento del GPS

### 2.3.3 WiFi

Para poder realizar la comunicación inalámbrica se ha usado un módulo WiFi. Este permite tanto crear una red local propia con su punto de acceso, como conectarse a una ya existente. Por tanto, en este caso es utilizada tanto para la comunicación local como de internet, dependiendo de la disponibilidad de una red. Para la realización de esta tarea el módulo utilizado es el ESP8266 en su versión más básica, el ESP-01.

Este componente viene equipado con una antena sobre la propia PCB, y ocho pines. Algunos de ellos están dedicados a la alimentación del módulo y la comunicación serial, mientras que otros son configurables para otras tareas al ser GPIOs. Puesto que el chip trabaja a 3.3 V, es necesario adaptar el nivel de voltaje con un regulador de tensión para la alimentación principal. Para esto se ha usado el mismo modelo que el planteado

para el cargador (ver sección 2.1.2). Sin embargo, esta vez se polariza otro diferente ya que ambas partes trabajan a niveles de voltaje diferente. En este caso, las resistencias escogidas fueron de  $200\Omega$  y  $330\Omega$  para R1 y R2, respectivamente. Además, como el Arduino emite a 5 V, es necesario utilizar un partidor de tensión en el TX del Arduino que adapte la señal de escritura a 3.3 V. Estas configuraciones quedan volcadas en el anexo (B.1).



---

FIGURA 2.8: *Pinout* del ESP8266 (ESP-01). Fuente: rubensm.com

Este módulo es implementado usando librería WiFiEsp [6] que permite varios protocolos de comunicación con otros equipos.

#### 2.3.4 Ultrasonidos

En caso de que hayan obstáculos es necesario, primero, detectarlos. Con este objetivo se instalan sensores ultrasonidos que permiten conocer la distancia a la que se encuentra un objeto. El principio de operación consiste en emitir una señal de 40 kHz desde el *trigger*, y escuchar lo que tarda en volver esta señal desde el *echo*. Dependiendo del tiempo que tarde se calcula la distancia sabiendo la velocidad de propagación del sonido a través del aire.

Una característica de estos módulos que es importante tener en cuenta es una resolución espacial muy reducida. Esto ocurre porque, simplemente, son capaces de detectar que existe un objeto en la visión del sensor pero sin especificar el lugar concreto. Como es evidente, cuanto más cerca se encuentre el objeto, menor es la incertidumbre del lugar exacto donde están. Además, dado que utilizan ondas de presión que se pueden disipar con facilidad, son módulos que operan mejor a corta distancia y en lugares donde hay poco viento, como se ha podido comprobar experimentalmente. Otro aspecto importante relativo a su desempeño es que el eco de otro ultrasonidos puede provocar falsas lecturas.

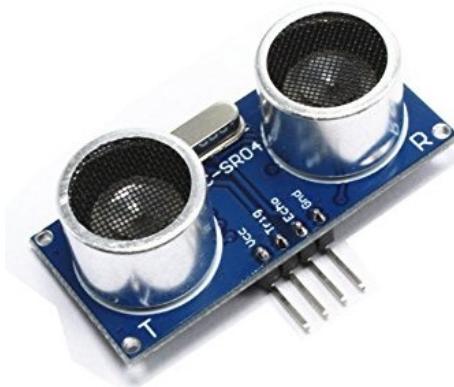


FIGURA 2.9: Sensor ultrasonido HC-SR04. Fuente: amazon.co.uk

Para este proyecto, se ha utilizado el modelo HC-SR04 [25], un ultrasonido que trabaja a 5 V, y que puede medir hasta un máximo de 5 m a una frecuencia de 20 Hz (50 ms mínimo entre muestrerios). Además, puede medir de forma efectiva hasta 22.5 grados hacia cada lado, aunque se considera que a 15 obtiene los mejores resultados (ver figura 2.10).

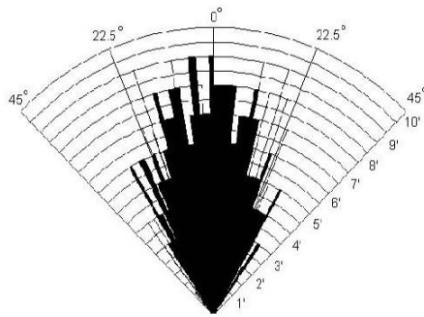


FIGURA 2.10: Test práctico del sensor HC-SR04. Fuente: [25]

El conexionado implica conectar la alimentación al voltaje adecuado y configurar los pines digitales según su función (ver código 2.4).

```

1 const int EchoPin_front_center = 34; //Echo pin in 34
2 const int TriggerPin_front_center = 28; //Trigger pin in 28
3
4 void setup() {

```

```
5 // Configure pins
6 pinMode(TriggerPin_front_center, OUTPUT);
7 pinMode(EchoPin_front_center, INPUT);
8 }
9
10 void loop() {
11     int cm_front_center =
12         calculate_distance(TriggerPin_front_center,
13             EchoPin_front_center);
14     delay(500); //Stabilization
15 }
16
17
18 int calculate_distance(int TriggerPin, int EchoPin){
19     long duration, distanceCm;
20
21     //Setting to LOW for 4us for cleaner reading
22     digitalWrite(TriggerPin, LOW);
23     delayMicroseconds(4);
24     //Triggering for 10us
25     digitalWrite(TriggerPin, HIGH);
26     delayMicroseconds(10);
27     digitalWrite(TriggerPin, LOW);
28     //Measuring the time between pulses in microseconds
29     duration = pulseIn(EchoPin, HIGH);
30     distanceCm = duration * 10 / 292 / 2; //Converting to cm
31     return distanceCm;
32 }
```

---

CÓDIGO 2.4: Ejemplo minimalista para medir distancias con un ultrasonido

Para este módulo no es necesario usar ninguna librería, dado que resulta sencillo de manejar. En su lugar, se ha creado una función llamada `calculate_distance()` que es invocada cada vez que hay medir alguna distancia con un ultrasonido.

El HC-SR04 necesita que el pin *trigger* esté a un uno lógico durante 10 µs para entonces enviar un tren de ocho pulsos de sonido. Inmediatamente, el módulo empieza a escuchar por el pin *echo* y mide el tiempo que pasa. Teniendo en cuenta la velocidad

del sonido a través del aire, se calcula el tiempo que tarda en avanzar un centímetro. Teniendo en cuenta que  $v_{sonido} = 343.2 \text{ m/s}$ , la ecuación 2.4 muestra la expresión utilizada para realizar este cálculo.

$$d_{cm} = \frac{\text{tiempo}}{29,2 \cdot 2} \quad (2.4)$$

## 2.4 Joystick

Con el objetivo de poder controlar el robot manualmente con precisión, y ubicarlo en el lugar deseado, se ha modificado un *josystick*.



(A) Joystick Quickshot QS-203.



(B) Detalle PCB del joystick.

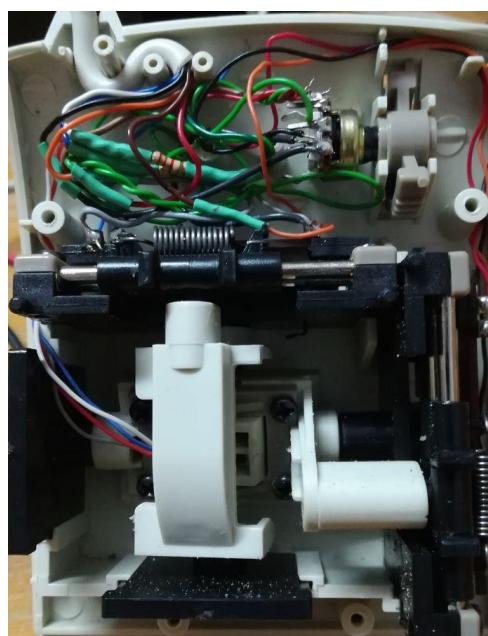
FIGURA 2.11: Joystick Quickshot QS-203 de fábrica

El modelo más básico se compone de dos potenciómetros que representan cada eje que están, a su vez, acoplados a un manipulador vertical. A partir de la lectura de sus valores el robot manda el comando adecuado a los motores.

En este caso se ha reciclado un Quickshot QS-203 (ver figura 2.11a) que posee tres potenciómetros, dos *pushbuttons* y un *switch*. Este tipo de joysticks antiguos se comunicaban con los ordenadores por un puerto de 15 pines dedicado llamado *Game Port*. Este estándar tenía definido un conector DA-15 con funciones concretas para cada pin,

y en su interior se encontraba la electrónica necesaria, generalmente condensadores y resistencias (ver figura 2.11b).

Para poder hacerlo compatible con el montaje propuesto es necesario modificar su circuito interior para que se ajuste a las necesidades del Arduino. Para ello, cada uno de los componentes electrónicos de su interior se han dispuesto en paralelo de tal manera que quedan polarizados a 5 V, además de leer el valor de los potenciómetros por las entradas analógicas del controlador. En cuanto a las entradas digitales, estas tienen una resistencia en serie de  $10\text{ k}\Omega$  para limitar la corriente cuando se cierre el circuito, a modo de *pull-up resistor* (ver anexo B.2).



---

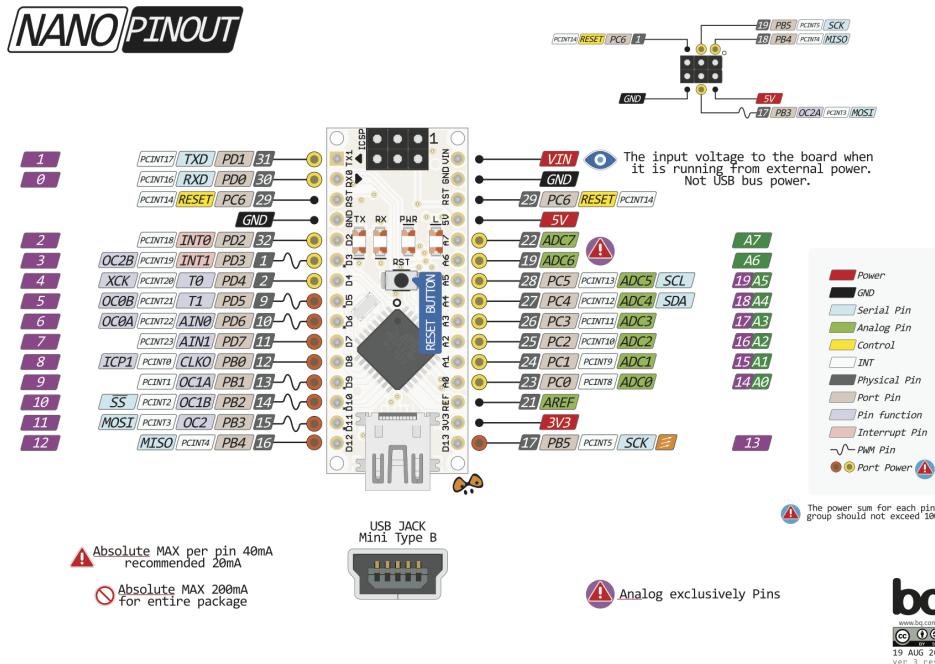
FIGURA 2.12: Joystick adaptado.

Con la intención de mantener el conector original del *joystick*, y poder enchufarlo en un sitio y otro con mayor facilidad, se reutilizó el propio puerto DA-15, modificando solamente el circuito interior (ver figura 2.12).

#### 2.4.1 Lector del joystick

Cuando el *joystick* se use para controlar el vehículo remotamente se utiliza un Arduino Nano para leer los valores del mismo y mandarlos por WiFi. Como se puede observar en el diagrama 2.13, esta placa posee tanto conexiones digitales como analógicas

ubicadas en los pines laterales.



El motivo principal por el que se escoge este módulo es su reducido tamaño y que cumple sobradamente con el número de entradas y salidas necesarias para llevar acabo su misión (ver anexo B.3).

## 2.5 Interfaces y protocolos de comunicación

Como se comentó en la sección 2.1.3, el controlador se comunica con cada módulo emitiendo señales en forma de niveles de voltaje y mínima corriente por uno o varios pines. Para que este proceso sea efectivo, se usan distintas interfaces de comunicación en función de los requerimientos de cada módulo.

El objetivo de esta sección es ofrecer una vista previa de cómo funcionan cada una de las utilizadas, con qué módulos, y las ventajas que ofrecen.

### 2.5.1 I<sup>2</sup>C

Ideado por Phillips, este protocolo síncrono se basa en dos líneas de comunicación con los módulos, una para el reloj (SCL), y otra para los datos (SDA). Una de las ventajas que tiene es que se puede configurar para tener varios sensores conectados a la misma línea SDA y SCL, comportándose como esclavos del controlador, el maestro. Entonces, para que se pueda distinguir entre los distintos dispositivos conectados a estas dos líneas, a cada uno se le asigna una dirección en hexadecimal que permite discernir para quién va la información. Desde el punto de vista constructivo, también es recomendable utilizar un *pull-up resistor* en cada una de las dos líneas cuando se utilizan varios módulos con este protocolo. En el caso de este proyecto, como el único que lo utiliza es la brújula, no es necesario.

Gracias a este protocolo es posible ahorrar bastante cableado, ya que es posible tener hasta 7, 8 y 10 bits de direcciones, aunque pueden existir otras limitaciones físicas que hacen que no sea realizable utilizar todas.

En este proyecto el único sensor que utiliza este protocolo es el magnetómetro con dirección 0x1E (ver sección 2.3.1). La conexión basta con alimentarlos al voltaje asignado y conectarlos a las entradas SDA y SCL del microcontrolador que vaya a realizar las lecturas, tal y como se explicó inicialmente.

### 2.5.2 PWM

Aunque no sea un protocolo de comunicación como tal, por la naturaleza de los motores cobra bastante importancia explicar la modulación por PWM como método para asignar velocidades variables a los motores por medio del controlador (ver 2.2.1.1). El funcionamiento está basado en establecer una consigna por medio de un valor proporcional de 8 bits de longitud sin signo. Esto se traduce en que, siguiendo el reloj del controlador, se ejecuta un pulso cuadrado cuyo ancho varía entre el valor mínimo y máximo en función del porcentaje de potencia asignada en ese instante.

Este tipo de métodos son especialmente necesarios en sistemas digitales en los que no es posible asignar valores intermedios, sino todo o nada. Así se consigue otorgar un voltaje medio que se ubica en el punto deseado, aceptando el límite de resolución propio de la longitud de bits utilizada para la cuantificación. Esta será menor cuanto

mayor sea la diferencia de voltaje que puede variar entre el mínimo y el máximo de consigna. La ecuación 2.5 muestra un cálculo aproximado de la resolución.

$$Q = \frac{\Delta V}{2^{bits}} = \frac{12}{2^8} = 46.875 \text{ mV} \quad (2.5)$$

### 2.5.3 Serial

La mayoría de los dispositivos conectados al controlador usan comunicación Serial para trasladar la información. Esta manda los datos bit a bit para luego reconstruir la sentencia en el otro extremo. Aparte de la propia consola del Arduino, el GPS (ver sección 2.3.2), y el WiFi (ver sección 2.3.3) son los módulos que utilizan este protocolo. La asignación durante esta primera iteración corresponde a Serial1 y Serial2, respectivamente. Por tanto, Serial0 corresponde al USB que se conecta al ordenador para imprimir por pantalla mensajes que sirven como *debug*.

Esta conexión se compone de una línea de transmisión (TX) y otra de recepción (RX) que se conectan según la función de cada uno. Es decir, la transmisión de uno se conecta a la recepción del otro y viceversa. La comunicación se realiza en forma de pulsos a una velocidad dada. Dependiendo del módulo, esta varía entre un *baudrate* de 9600 baudios para el GPS y el WiFi, hasta 115200 para el Serial0.

### 2.5.4 UDP

Este protocolo de comunicación inalámbrica es utilizado para el modo manual de control del robot (ver sección 3.2). La función de esta es transmitir los datos desde el *joystick* al vehículo, de tal manera que el primero actúa como si fuera un cliente que escribe en el otro que hace la función de servidor.

La utilización de este protocolo frente a otros como TCP/IP permite una comunicación más viable, dado que un requisito es que sean en tiempo real. Esta idea se basa en que el implementado no requiere de confirmación por parte del otro de que ha recibido el mensaje, sino que directamente sigue mandando el siguiente paquete de datos. Esta forma de operar es más rápida, aún cuando se pierdan mensajes. Es decir, si se mandan con la suficiente frecuencia, las consecuencias reales pueden ser imperceptibles. Por ello es usado en la comunicación de videojuegos *on-line*.

Para utilizarlo, se implementa con la librería WiFiEsp [6] que ofrece ambos protocolos compatibles con el módulo ESP8266.

## Capítulo 3

### Implementación de la iteración 1

Una vez comentados los módulos que pueden hacer falta, en este capítulo se presenta la implementación del primer prototipo. La idea principal de esta primera fase ha sido realizar una primera aproximación al algoritmo que detectará y evitará los obstáculos, comandando a los motores las consignas adecuadas. Por tanto, la idea de la localización que se comentaba durante la sección 1.2 se afronta solamente desde el punto de vista de la orientación. También se ha preparado un sistema de control manual con el *joystick* para que este pueda ser manejado de forma remota por UDP.

Los planos de esta versión se pueden consultar en el anexo C.1, mientras que el conexionado queda recogido en el B.1 y en la figura 3.1. En esta última, se observa que el Arduino actúa como cerebro central al que se conectan los sensores y actuadores, así como el módulo encargado de la conexión remota. Por tanto, todos dependen del controlador, tanto el bajo nivel que lee los sensores o comanda los motores, como el alto que se comunica con el exterior.

Dadas las dimensiones del vehículo, existen cuatro ultrasonidos ubicados de tal manera que dos cubren los extremos delanteros, es decir, el ancho, y los otros dos justo el otro lado de la esquina. El objetivo de esta ubicación es evitar obstáculos que se encuentren directamente delante, y posteriormente ir añadiendo más alrededor si hicieran falta.

En cuanto al software, el Arduino contiene la programación equivalente al diagrama de bloques de la figura 3.5, entre otros. Por tanto, la idea de esta implementación es también obtener dos modos de control según convenga. Uno de ellos es automático, donde el vehículo es capaz de seguir una dirección e intenta evitar los obstáculos que hayan en el camino (ver sección 3.3), y otro manual, en el que es controlado manualmente desde un *joystick* con otro Arduino remoto (ver sección 3.2).

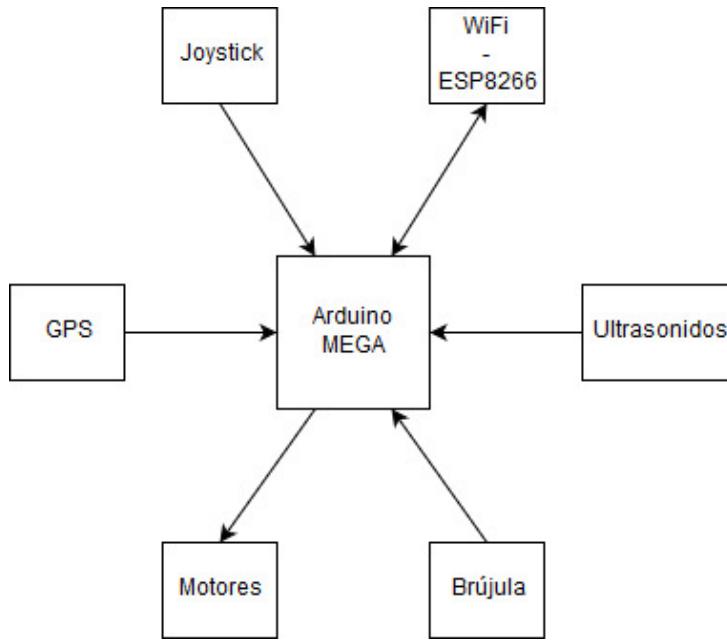


FIGURA 3.1: Estructura electrónica de la iteración 1.

### 3.1 Cálculo cinemático

Siguiendo las ideas expuestas en la sección 1.2.1 es posible realizar los cálculos cinemáticos reales del prototipo propuesto.

Como se comentó en la sección 2.2.1, la transmisión de la potencia del rotor a las ruedas se realiza con una reductora. En el caso de esta implementación, el rotor tiene un piñón de ocho dientes, mientras que el engranaje que está acoplado a la rueda posee dieciocho. Esto significa que la relación de transmisión es de 8/18. Sabiendo que las ruedas poseen un diámetro de 16 pulgadas, la velocidad lineal máxima del vehículo es de 0.624 m/s. La ecuación 3.1 muestra el desarrollo completo del cálculo a partir de la ecuación 1.1.

$$v = \frac{d}{2} \cdot \omega = \frac{16[\text{inch}] \cdot 25.4 \times 10^{-3} [\text{m}]}{2} \cdot \frac{8}{18} \cdot 66,67[\text{rpm}] \cdot \frac{2 \cdot \pi[\text{rad}]}{60[\text{s}]} = 0.624 \text{ m/s} \quad (3.1)$$

Si las ruedas rotan a máxima velocidad sobre sí mismas, obtendríamos una rotación pura de 3.374 rad/s, o 193.31 °/s (ver ecuación 3.2).

$$\omega = \frac{(0,624 - (-0,624))[\text{m/s}]}{37 \times 10^{-2} [\text{m}]} = 3,374 \left[ \frac{\text{rad}}{\text{s}} \right] \frac{360[\circ]}{2\pi[\text{rad}]} = 193.31^\circ/\text{s} \quad (3.2)$$

El significado de este último cálculo significa que cuando el robot vaya a cambiar de dirección, girará esa cantidad por segundo siempre que la velocidad sea máxima. Por tanto, este dato revela que es importante muestrear la orientación con mayor frecuencia, o reducir la velocidad de las ruedas, para así garantizar que el vehículo sea estable a la hora de orientarse, es decir que no se pase el intervalo de tolerancia establecido para el nuevo rumbo.

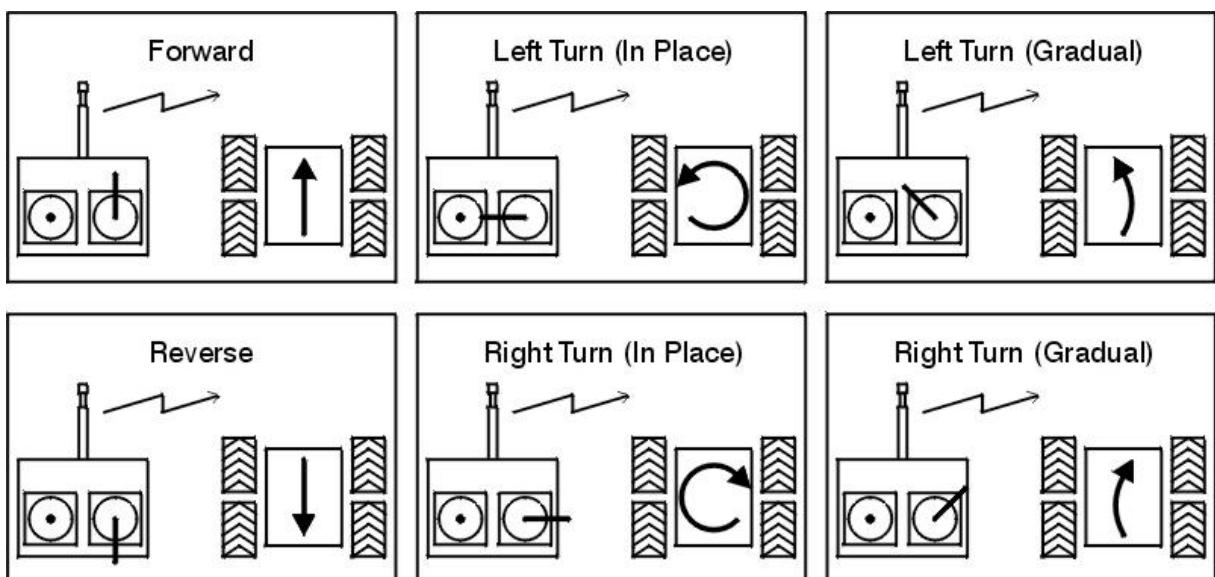


FIGURA 3.2: Movimientos de un robot controlado por acción diferencial.  
Fuente: stackoverflow.com

### 3.2 Modo manual

Este tipo de control pretende manejar remotamente el robot con un *joystick*. Este es especialmente útil para ubicarlo en una posición concreta, dado que el error de los GPS suele ser más grande que la precisión requerida.

Conceptualmente es más sencillo que el automático, ya que su función es reducida a escuchar por un puerto la información de los valores leídos por el *joystick* y su Arduino,

que le envían cíclicamente. Además, para conseguir el mayor éxito posible, debe funcionar con la menor latencia, motivo por el cual se opta por usar UDP frente a TCP/IP (ver sección 2.5.4).

### 3.2.1 Cliente

Dentro de este modo de control, el cliente está ubicado en el lado del *joystick*. Lo primero que realiza este es medir la información de los sensores del *joystick* y la envía. Para ello, el Arduino intenta conectarse al servidor, es decir, el robot, y esperar a que este le asigne una IP. Una vez ha terminado de establecerse una conexión, comienza a realizar la lectura de los valores usando la idea explicada en la sección 2.4, donde se hacen las lecturas analógicas y digitales pertinentes. Posteriormente, esta información se codifica en un *string* de 18 + 1 elementos, como máximo, y se envían por UDP a través del puerto configurado.

El número de elementos viene dado por las longitudes máximas que pueden tomar los valores. El cálculo sale, por un lado, de que el *josystick* tiene tres potenciómetros, cuyo valor de 10 bits es variable entre uno y cuatro dígitos (0 y 1023). Por otro, hay dos botones digitales cuyo estado es binario, por lo que cada uno suma otro dígito. Al final, la sentencia queda codificada como un conjunto de caracteres numéricos que están directamente relacionados con el valor calculado por el Arduino. Dicho esto, es lógico pensar que si se codificara tal cual, habría un problema fundamental para que el servidor asocie cuántos caracteres hay para cada una de las cinco variables, especialmente las de los potenciómetros. No se debe olvidar que la longitud de estas varía, a diferencia de los botones. Aunque seguramente hayan otras maneras para discernir entre el principio y el inicio de una variable dentro de un mismo *string*, la solución adoptada consistió en añadir un espacio entre cada una. De esta manera, el servidor solo almacena cada variable hasta encontrar un espacio, momento en el que interpreta que está completa. Otro de los motivos por los que esto parece una buena solución es que resulta más sencillo de entender o leer en comparación con una cadena seguida de dígitos a la hora de hacer *debug*.

Para ilustrar el párrafo anterior, podemos tomar el ejemplo mostrado en la tabla 3.1.

En la sentencia anterior, el valor del eje *x*, *y* y *z* sería de 1023, y los botones estarían a un uno lógico. El código 3.1 pretende ilustrar cómo se forma el *string* desde el Arduino Nano.

Variable	x	y	z	button1	button2
Valor	1023	1023	1023	1	1

CUADRO 3.1: Ejemplo de *string* codificada.

---

```

1 //Building the string to be sent
2   ReplyString = String(x_val) + " " + String(y_val)+ " " +
      String(z_val) + " " + String(button1_state) + " " +
      String(button2_state) + " ";
3   ReplyString.toCharArray(ReplyBuffer, 19); //max. 18 elements +
    1 needed

```

---

CÓDIGO 3.1: Construcción del *string* a enviar

Por un lado, la implementación del código Arduino sigue el esquema básico de la figura 3.3. Por otro, el anexo A.2 contiene el programa real escrito.

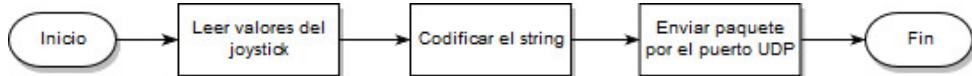


FIGURA 3.3: Diagrama de bloques del cliente UDP.

### 3.2.2 Servidor

En el caso de este vehículo, se considera que el servidor va montado sobre el propio robot, y su misión es estar a la espera hasta que llegue alguna información para comandar los motores.

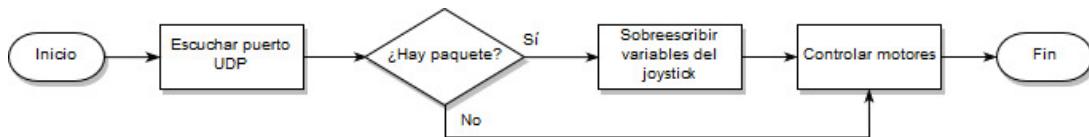


FIGURA 3.4: Diagrama de bloques del servidor UDP.

Para hacer esto, el ESP8266 que está montado sobre el vehículo se configura como punto de acceso, para que se conecte el cliente y lo provea de una IP (tiene DHCP). Después, empieza a escuchar a través del puerto 10002. Una vez recibido y almacenado

el mensaje en un *buffer*, el robot lo analiza y decodifica para guardarla en cada una de sus variables. La manera de realizar esto es, usando la librería `stdio.h` de C, invocar la función `sscanf()` que permite almacenar cada uno de los datos en una variable. Estas luego serán utilizadas para comandar a los motores las consignas adecuadas. Una vez termina este proceso vuelve a iterar desde el principio (ver diagrama 3.4). Resulta importante comentar que cuando no existe un paquete nuevo, por seguridad, se sobreescreiben las variables del *joystick* como si este hubiera mandado a parar. De esta manera se evita que continúe realizando una acción si la comunicación ha caído.

### 3.3 Modo automático

En este tipo de control el vehículo debe poder operar de forma autónoma obedeciendo a una interfaz web que manda los puntos de GPS a los que debe dirigirse. En esta primera iteración, este modo se ha centrado en priorizar la lógica necesaria para identificar los obstáculos.

No obstante, para esta versión se establece un rumbo concreto explícito que debe seguir dentro del código, y se colocan obstáculos para analizar cómo reacciona a ellos. De esta manera, aunque el GPS esté conectado, en ningún momento aporta ningún dato al sistema.

Como es lógico pensar, en este modo sí se utilizan todos los sensores descritos en el capítulo 2, que mantienen actualizado el estado del entorno en el que se mueve el robot.

#### 3.3.1 Algoritmo para evitar los obstáculos

Existen dos ideas básicas de navegación, por un lado, cuando el robot avanza en campo abierto, este puede dirigirse en la dirección adecuada directamente. Por otro, si este encuentra un objeto en su trayectoria debe rodearlo guardando siempre una distancia de seguridad hasta haber pasado el peligro. En ese momento, volvería a apuntar en la dirección asignada siguiendo la misma idea (ver diagrama de la figura 3.5).

En cuanto a la implementación en Arduino, lo primero que hace es leer su rumbo actual y los cuatro ultrasonidos que están implementados. Con esta información básica el controlador ya conoce su entorno, por lo que lo siguiente que debe hacer es tomar

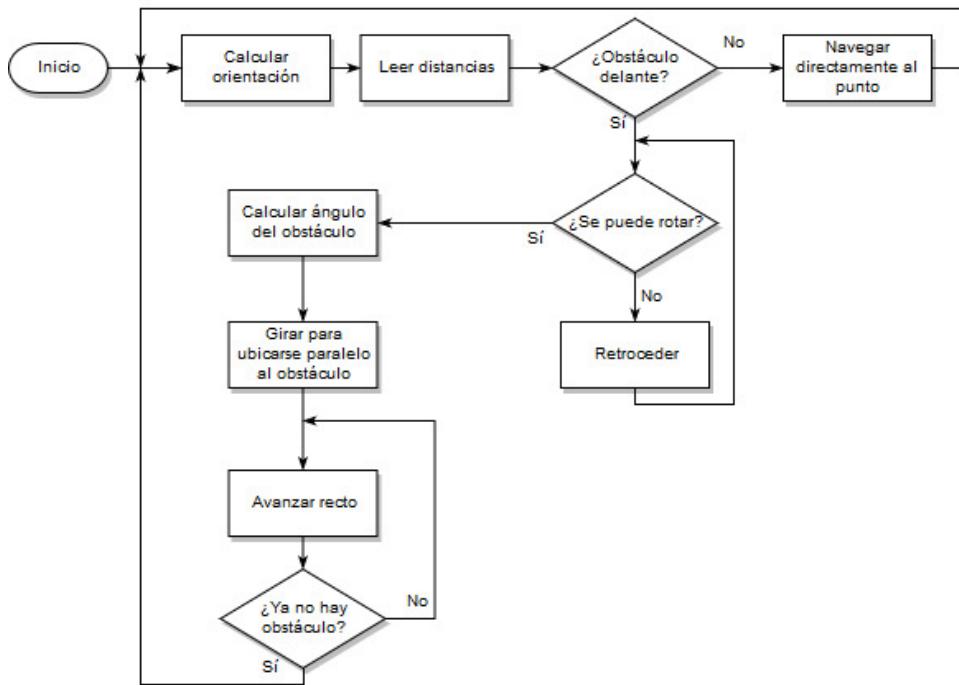


FIGURA 3.5: Algoritmo para evitar obstáculos de la iteración 1.

una decisión. En caso de que detecte un obstáculo que esté por debajo del umbral de seguridad designado, *threshold*, entra en un bucle para tratar de rodearlo. Este espacio de seguridad se ha establecido en un valor arbitrario de 60 cm para poder hacer las primeras pruebas. Una vez detectado el objeto, puede también ocurrir que lo detecte demasiado tarde como para girar sobre sí mismo. En este caso, retrocede hasta que esta acción sea posible sin golpear con ningún objeto.

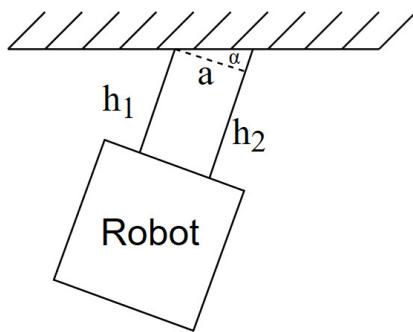


FIGURA 3.6: Cálculo del ángulo del obstáculo.

Después de determinar que está en la zona segura para girar sobre sí mismo, calcula el

ángulo al que se encuentra el obstáculo. Esta operación la puede realizar fácilmente con la función `calculate_obstacle_angle()`, que se basa en relaciones trigonométricas para devolver dicho ángulo en grados (ver código 3.2). Es decir, conociendo la distancia que hay entre ambos sensores, y la que existe entre el objeto y cada uno de ellos, deduce el ángulo al que este se encuentra. Haciendo el complementario de este ángulo, se obtiene la rotación necesaria para ubicarse paralelo al mismo. Este concepto queda ilustrado en la figura 3.6, y como se puede observar, la relación matemática viene dada por la arcotangente de ambos catetos (ver ecuación 3.3).

---

```

1 // Used to calculate the angle at which the obstacle is
2 float calculate_obstacle_angle(float cm_front_left, float
3     cm_front_right) {
4     float diff = abs(cm_front_left - cm_front_right);
5     float alpha = atan2(diff, 49); //49 cm between both sensors
6     float alpha_degrees = alpha * 180 / 3.14159265358979323846;
7     return alpha_degrees;

```

---

CÓDIGO 3.2: Cálculo del ángulo al que se encuentra el obstáculo

$$\alpha = \operatorname{arctg}\left(\frac{h_2 - h_1}{a}\right) \quad (3.3)$$

Por otro lado, la variable que calcula el ángulo que debe girar se llama `avoid_direction()`, que a su vez llama a una función que calcula el nuevo rumbo. Este se calcula añadiendo al valor actual de `heading` al ángulo complementario del obstáculo. No obstante, esta operación no es posible ejecutarla de forma directa ya que el rango en el que se mueve la orientación está entre  $0^\circ$  y  $360^\circ$ . Tras una suma continuada, esto resultaría en un valor fuera del mismo, haciendo que el robot nunca pare de girar. Por tanto, para corregir esto se debe identificar el lado hacia el que rota el robot, para saber si se suma o se resta, y en función de eso ver si esa operación resultaría en un valor negativo, o mayor de  $360^\circ$ . En caso de que así sea, se ajusta el valor adecuadamente. Esta corrección es la realizada en el código 3.3.

---

```

1 // Used to calculate proper heading values
2 float calculate_heading(float heading, float obstacle_angle, int
3     direction) {
4     if(direction == 0) {

```

---

---

```

4     if(heading + obstacle_angle > 360) {
5         return heading + obstacle_angle - 360;
6     }
7     else{
8         return heading + obstacle_angle;
9     }
10    }
11    else{
12        if(heading - obstacle_angle < 0){
13            return heading - obstacle_angle + 360;
14        }
15        else{
16            return heading - obstacle_angle;
17        }
18    }
19 }
```

---

### CÓDIGO 3.3: Cálculo de rumbos correctos

Una vez calculados estos valores, el robot empezará a rotar hasta llegar a un ángulo mayor o igual al calculado, de lo contrario, podría chocar si avanza hacia delante. En esta rotación, el vehículo mide continuamente su dirección actual para poder ser comparada con la que desea obtener. Después de salir de este bucle, vuelve a medir las distancias de sus sensores para conocer si, tras estas operaciones, ya ha salido de la zona de peligro. En caso de que así sea, entonces empieza su marcha hacia delante mientras continua leyendo los sensores de distancia. Si alguno de ellos detectara que tras este avance hay un objeto cercano, el robot saldría del bucle para volver a comenzar desde el principio.

Si por el contrario no detectara ningún obstáculo cercano desde un principio, entraría en otra función denominada `navigate_open_space()` (ver código 3.4). En esta el vehículo se desplaza directo al punto, rotando sobre sí mismo para orientarse y avanzando a la velocidad nominal, ya que esta es la manera más rápida de llegar.

---

```

1 // Autonomous logic for compass when there is open space and no
   obstacles
2 void navigate_open_space(float heading, float heading_command) {
```

```

3     if(heading >= heading_command + 10 || heading <=
4         heading_command - 10){ // Change direction
5         if((heading - heading_command < 0 && abs(heading -
6             heading_command) < 180) || (heading - heading_command > 0
7             && abs(heading - heading_command) > 180)){
8             Serial.println("Turning right to heading_command!");
9             motor.set(A, 130, FORWARD);
10            motor.set(B, 130, REVERSE);
11            motor_delay(10);
12        }
13    } else{
14        Serial.println("Turning left to heading_command!");
15        motor.set(A, 130, REVERSE);
16        motor.set(B, 130, FORWARD);
17        motor_delay(10);
18    }
19    else{ // Continue forward
20        Serial.println("Going forward! @ open space");
21        motor.set(A, 110, FORWARD); //correcting drift
22        motor.set(B, 130, FORWARD);
23        motor_delay(10);
}

```

CÓDIGO 3.4: Navegación en espacios abiertos

Como es lógico pensar, dado que es muy improbable que en el momento en el que se calcula el *heading* coincida con que el robot esté justamente en ese valor, es necesario establecer una tolerancia. En este caso, se ha considerado que una desviación sobre el rumbo menor de 10° era aceptable. Por tanto, se evalúa si la orientación actual está fuera de este rango, de lo contrario, puede continuar avanzando hacia delante. Además, en caso de que vaya a rotar para reorientarse, el robot es capaz de discernir hacia qué lado es más corto llegar al rumbo objetivo.

### 3.4 Joystick

Como se adelantó anteriormente en la sección 3.2, el modo manual necesita de la acción de un *joystick* que permita moverlo de forma intuitiva. El uso de este aparato introduce el problema matemático de que existen dos motores que se mueven independientemente en función de dos variables,  $x$  e  $y$ , que no lo son y que marcan un vector de dirección en un espacio cartesiano (ver figura 3.7). Se puede deducir que este problema no existiría si, por el contrario, asignáramos cada una de las variables a las consignas de cada uno de los motores. No obstante, haciendo esto se perdería el sentido de utilizar este tipo de controles, ya que se pretende que sea intuitivo para el usuario.

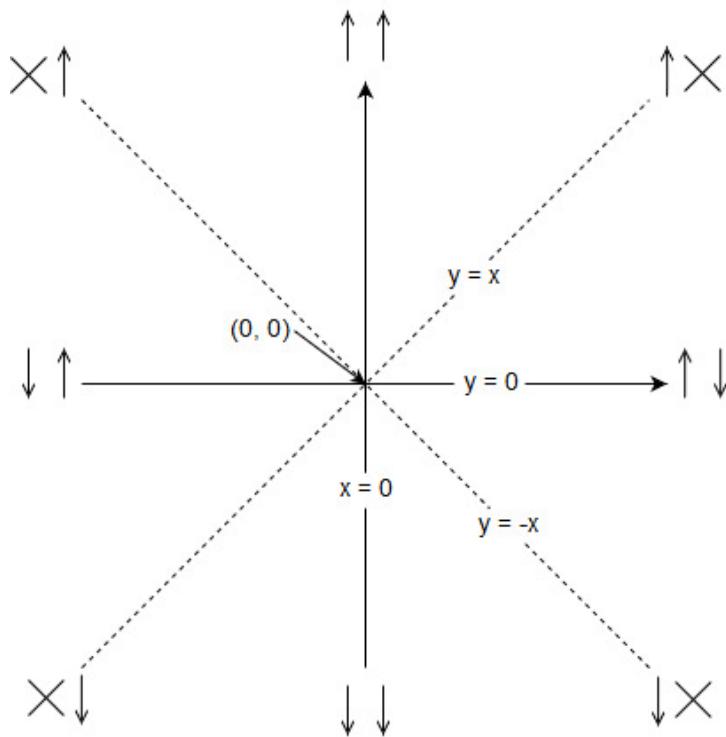


FIGURA 3.7: Diagrama de acción de las ruedas en función de las coordenadas del joystick.

En el caso presentado, como se comenta en la sección 1.2.1, hay un robot que para girar debe mover una rueda más rápida que la otra. En el problema del *joystick*, se puede ver que si vamos a toda potencia hacia delante y quisieramos girar, tendríamos dos opciones: o aumentamos la velocidad de una, más aún; o bien sacrificamos el avance de una para crear esta diferencia que provoca el giro. Evidentemente, lo primero no es

realizable, ya que los motores no pueden girar más rápido que su velocidad máxima, por lo que la segunda opción es más adecuada. Sin embargo, si estuviéramos parados y quisiéramos girar, ocurriría justamente lo contrario: no es posible reducir la velocidad de ninguno de los motores, pero sí que es posible aumentar la de uno de ellos.

Podemos suponer que el joystick tiene un origen de coordenadas en la posición vertical, es decir, es el punto en el que ningún motor se mueve, reciben un comando igual a cero. Desde el punto de vista práctico, los potenciómetros ubican su cero en los extremos por lo que es necesario centrar las medidas restando la mitad a la longitud total que se puede obtener del ADC. De esta manera, en caso de que el valor real sea 512, al restarle la variable *centre* obtendríamos un origen en este punto. Esta misma corrección ocurre para el resto de valores disponibles en el rango de 0 a 1023. Cabe recordar en este punto que el ADC del Arduino, responsable de leer los potenciómetros, es de 10 bits (ver sección 2.1.3).

Es importante puntualizar que, dada la naturaleza analógica de estos potenciómetros, es vital filtrar el ruido generado por la línea para obtener valores más precisos. La solución adoptada ha sido, aprovechando la alta frecuencia del micro (ver sección 2.1.3), realizar la media de varias medidas analógicas en cada ciclo. Aunque es una solución trivial, se demuestra efectiva para los dos ejes principales que utiliza el *joystick*. Este método fue implementado en una función que se invoca cada vez que hay que realizar una medida de un valor analógico (ver código 3.5).

---

```
1 int calculate_average(int axis){  
2     int value;  
3     int num = 10;  
4     // Calculate average of analog measurements  
5     for(int i = 0; i < num; i++){  
6         value += analogRead(axis);  
7     }  
8     int average = value/num;  
9  
10    return average;  
11 }
```

---

CÓDIGO 3.5: Filtrado de valores analógicos de los ejes del *joystick*

Una vez se obtienen los valores de cada eje hay que afrontar el problema que se comentó inicialmente en esta sección. Existen numerosos métodos para realizar la transformación en la que dos entradas dependientes determinan la salida de dos motores independientes que marca la dirección final. Algunas aproximaciones usan el eje *y* del *joystick* para marcar la aceleración y su sentido, mientras que el *x* es usado para marcar la diferencia de velocidad de una rueda frente a otra para provocar el giro.

No obstante, en este caso se ha usado un método que convierte las coordenadas cartesianas del *joystick* a un valor de consigna para cada motor directamente, por medio de coordenadas de diamante [23]. La principal ventaja de este algoritmo está en que al final del mismo se obtiene el valor a escribir en cada actuador de forma directa, por lo que facilita bastante la implementación.

Para transformar las coordenadas directamente toma el punto cartesiano del *joystick* ya centrado, y lo convierte a coordenadas polares para poder rotar el plano 45 grados. Luego, vuelve a coordenadas cartesianas y los valores se multiplican por la raíz de dos, ya que es la hipotenusa del triángulo formado en tanto por uno. Finalmente, se ajusta el rango en el que pueden moverse los valores de consigna. Estos se pueden ubicar entre el intervalo de 0 y 255, dado que la longitud del DAC es de 8 bits (ver sección 2.2.1.1). En la figura 3.8 se representa el diagrama de bloques de este proceso.

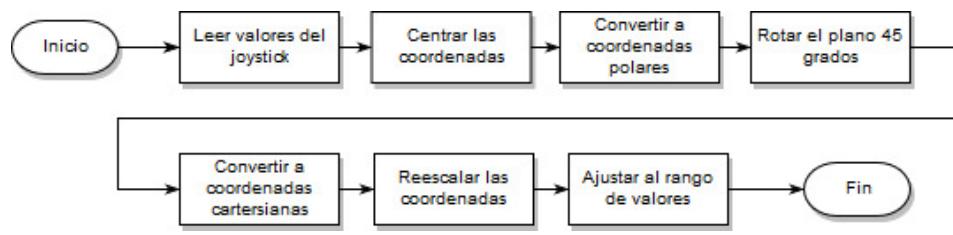


FIGURA 3.8: Diagrama de bloques del cálculo de las coordenadas de diamante.

El código 3.6 representa el programa en C que hace esta transformación, mientras que el anexo A.3 muestra la implementación completa para el control local.

---

```

1 // Read, store and correct raw values
2 x_val = calculate_average(x_axis);
3 y_val = calculate_average(y_axis);
4 button1_state = digitalRead(button1);
5 button2_state = digitalRead(button2);

```

```
6
7 x_val = 1024 - x_val;
8 y_val = 1024 - y_val;
9
10 const int centre = 512;
11
12 int x = x_val - centre;
13 int y = y_val - centre;
14
15 // Computing the speed
16 // Convert to polar
17 float r = hypot(x, y);
18 float t = atan2(y, x);
19
20 // Rotate by 45 degrees
21 t = t - 3.14159265358979323846 / 4;
22
23 // Back to cartesian
24 float left = r * cos(t);
25 float right = r * sin(t);
26
27 // Rescale the new coords
28 left = left * sqrt(2);
29 right = right * sqrt(2);
30
31 // Clamp to -245/+245 to avoid over power
32 left = fmax(-245, fmin(left, 245));
33 right = fmax(-245, fmin(right, 245));
34
35 // Convert to absolute
36 float left_abs = abs(left);
37 float right_abs = abs(right);
```

---

CÓDIGO 3.6: Cálculo de las coordenadas de diamante

Al rotar el plano 45 grados, los ejes laterales del diamante en los dos planos superiores e inferiores se convierten en la consigna de cada uno de los motores. De esta manera, cuando el robot se mueve a toda velocidad hacia delante, o lo que es lo mismo, solo se

aplica el eje  $y$ , los dos motores aplicarían una consigna al 100 % de avance. Sin embargo, si ubicáramos el *joystick* en posición diagonal sobre el primer plano, de tal manera que quisieramos que el robot gire hacia la derecha, la rueda izquierda tendría que girar a máxima velocidad, mientras que la otra está quieta. Esto es lo que justamente se puede observar en la figura 3.9.

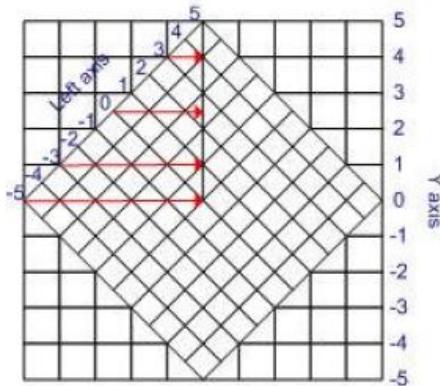


FIGURA 3.9: Espacio de diamante. Fuente: [23].

Una vez configurado, en teoría, los motores deberían responder adecuadamente. No obstante, en la práctica esto no funciona porque al enviar esta información directamente a los motores se observa que en algunos casos no hay movimiento. Esto es debido a que la librería usada en el controlador de los motores (ver sección 2.2.1.1), solo acepta números de 8 bits sin signo, por lo que cuando se intenta enviar una consigna negativa se produce un error. Por esto, el código 3.6 calcula el valor absoluto del resultado del algoritmo. Es decir, la manera de discernir entre si el motor debe moverse hacia delante o hacia atrás viene determinado por el argumento *FORWARD* o *REVERSE* que acompaña al resto de parámetros.

Este problema hace que la implementación real en el modelo no sea tan directa, y haya que discernir entre los distintos casos que se pueden afrontar para establecer si cada rueda va hacia delante o al revés. Visto de esta manera, se puede apreciar que el espacio cartesiano se puede dividir en ocho cuadrantes delimitados por dos funciones según el sentido de giro de las ruedas. Por un lado, en la tabla 3.2 se establecen los movimientos propios de un correcto funcionamiento. Por otro, se pueden distinguir los casos extremos en los que ocurre un cambio en la dirección de la rueda. Esto ocurre

cuando el punto que marca el *joystick* se encuentra justo encima de alguna de las funciones representadas en la figura 3.7. Esto queda reflejado en la tabla 3.3, suponiendo la máxima consigna posible para cada caso.

Cuadrante	Función A	Función B	Movimientos	V. izq.	V. der.
1	$y = x$	$y > 0$	Adelante, derecha	Alta	-Baja
2	$x > 0$	$y = x$	Adelante, derecha	Alta	Baja
3	$x < 0$	$y = -x$	Adelante, izquierda	Baja	Alta
4	$y = -x$	$y > 0$	Adelante, izquierda	-Baja	Alta
5	$y < 0$	$y = x$	Atrás, izquierda	-Baja	-Alta
6	$y = x$	$x < 0$	Atrás, izquierda	-Baja	-Alta
7	$x > 0$	$y = -x$	Atrás, derecha	-Alta	-Baja
8	$y = x$	$y < 0$	Atrás, derecha	-Alta	-Baja

CUADRO 3.2: Movimientos de las ruedas según subplanos.

Cuadrantes	Función	Movimiento	V. izquierda	V. derecha
1, 2	$y = x$	Adelante, derecha	Alta	Parada
2, 3	$y > 0$	Traslación adelante pura	Alta	Alta
3, 4	$y = -x$	Adelante, izquierda	Parada	Alta
4, 5	$x < 0$	Rotación izquierda pura	-Alta	Alta
5, 6	$y = x$	Atrás, izquierda	Parada	-Alta
6, 7	$y < 0$	Traslación atrás pura	-Alta	-Alta
7, 8	$y = -x$	Atrás, derecha	-Alta	Parada
8, 1	$x > 0$	Rotación derecha pura	Alta	-Alta

CUADRO 3.3: Casos extremos.

Siguiendo las tablas anteriores, es posible deducir el programa del anexo A.3 que realiza esta discreción. Es necesario comentar que, dado que los elementos mecánicos del *joystick* poseen una holgura natural, y el ADC del Arduino tiene una resolución relativamente alta en comparación, es necesario establecer unas tolerancias para que el vehículo no se mueva aún cuando el *joystick* esté vertical. En el ejemplo del anexo este umbral está ubicado en 100, un 20 % respecto al total, aproximadamente. Aunque es un valor bastante alto, y experimentalmente la mitad también da buenos resultados, este aporta mayor confianza.

### 3.5 Presupuesto

Otro punto interesante relacionado con el proyecto es su coste. Como se comentó inicialmente, el interés de usar materiales reciclados y/o comunes reside en su economía y la facilidad de recambio en caso de que alguno falle. De esta manera, en un cálculo no exhaustivo, es posible establecer un costo en materiales de 155 €, aproximadamente.

Concepto	Uds.	Precio unitario (€)	Importe (€)
Motor	2	12,00	24,00
Arduino MEGA 2560	1	15,00	15,00
Bicicleta	1	10,00	10,00
Batería	1	40,00	40,00
Tabla madera	1	3,00	3,00
Ángulos de chapa	2	2,00	4,00
Ultrasonidos	4	1,00	4,00
Controlador motor	2	14,00	28,00
Brújula	1	2,00	2,00
GPS	1	5,00	5,00
Joystick	1	10,00	10,00
WiFi	1	2,00	2,00
Varios	1	10,00	10,00
Total			157,00

CUADRO 3.4: Presupuesto iteración 1

Como se puede observar en la tabla 3.4, se han listado solamente los conceptos tangibles del proyecto, sin tener en cuenta el coste de la mano de obra para construirlo o el desarrollo del mismo. Además, para poder entender mejor esta tabla, es necesario comentar que no se ha presupuestado explícitamente el costo del cableado, la tornillería y otros detalles menores del proyecto. Esto es debido a que su coste es bastante reducido para desglosarlo, por lo que en su lugar este tipo de componentes se han incluido en el concepto varios, donde se establece un precio estimado para todos ellos.

### 3.6 Discusión de resultados

Tras probar experimentalmente esta versión se han observado algunas carencias que deberían ser resueltas en la siguiente iteración.

Por un lado, en cuanto a los aspectos de hardware, el módulo WiFi suele generar bastantes problemas para inicializar y crear el AP. Estos pueden estar relacionados con algún falso contacto o problema interno del chip ya que al aislarlo e intentar comunicarse con comandos AT por Serial, a veces aparecen caracteres extraños. Aunque esto último puede ocurrir cuando el *baudrate* no es el mismo entre uno y otro, se observa que en otras ocasiones se comunica sin este problema.

Por otro lado, a la hora de desarrollar el robot es bastante complicado hacer un buen *debug* debido a que cuando trabaja en modo autónomo hay que estar conectados por cable con él. Por tanto, para poder monitorizar el monitor serial es necesario perseguirlo mientras se observa la pantalla, haciendo que sea una tarea bastante difícil.

### 3.6.1 Problemas modo manual

Este modo de control funciona adecuadamente cuando el *joystick* se conecta localmente, a diferencia de cuando se intenta hacer remotamente. Al intentarlo, se aprecia que el vehículo tarda demasiado en responder. Curiosamente, este exceso de latencia no ocurría cuando se realizaron las pruebas en vacío, es decir, el servidor y el cliente montados en una *protoboard* transmitiendo datos sin motores conectados. En estas pruebas, se perdía un 10 % de los paquetes que se mandaban, un valor aceptable dado que el tiempo que tarda en hacer un ciclo es aproximadamente 40 ms. Sin embargo, al pasar al montaje real parece que los paquetes se pierden más a menudo provocando que se pierda el concepto de tiempo real.

En relación a esto último, es posible que otro problema que tenga la implementación realizada durante esta versión sea que el *buffer* usado sea demasiado largo y al recibir instrucciones más rápido de lo que el robot itera por dentro de su propio bucle, este se llena con instrucciones viejas, creando el retardo comentado anteriormente.

Otro error menor que se observa experimentalmente es que si el *joystick* cambia su posición rápidamente existe un cambio brusco de velocidad. Una solución podría ser aplicar un sistema de control con memoria para evitar que el cambio de consigna entre un ciclo y otro sea brusco.

### 3.6.2 Problemas modo automático

En términos generales, este modo funciona como se esperaba. No obstante, se aprecia que en algunas situaciones se comporta de manera poco robusta, especialmente cuando se encuentra en un pasillo o alguna zona estrecha. Se observa que el vehículo queda bloqueado tratando de buscar un lugar sin obstáculos fuera del umbral programado. Este gira y detecta un objeto en un lado, al volver a corregir el rumbo gira más de la cuenta y vuelve a detectar otro en el opuesto, quedando en ese bucle. Aunque es cierto que, seguramente, los sensores de ultrasonidos no sean los más adecuados, otro motivo que hace que no funcione correctamente está relacionado con el tipo de control que se implementado.

Como se puede observar en todos los pedazos de código adjuntos anteriormente, el tipo de control que se ha aplicado es un todo o nada. Es decir, asigna la consigna más alta o más baja dependiendo del movimiento que quiera realizar. Este tipo de control se implementó porque era el más rápido y directo de programar, pero seguramente sea recomendable utilizar uno proporcional, o proporcional-integral, para poder amortiguar el efecto de la inercia del vehículo. De esta manera, en lugar de establecer un valor fijo para los motores independientemente del ángulo que deba girar, por ejemplo, se le asignaría una consigna inversamente proporcional a lo cerca del ángulo objetivo que esté. Para esto habría que realizar una función que sería llamada cada vez que haya que controlar un motor.

Además de este problema, otra mejora está en optimizar la cantidad de *delay* que se aplica después de cada consigna de motor. Actualmente, está situada en 10 ms, pero quizás con menos ya es suficiente para vencer la inercia y que los motores giren. Una de las ventajas que aporta mejorar este aspecto es que el Arduino evaluaría las condiciones con mayor frecuencia, ya que cada vez que se invoca la función *delay()* este se queda parado en esa sentencia durante ese tiempo. Esta modificación aumentaría la resolución del sistema al aumentar la frecuencia de muestreo. No obstante, al hacer esto hay que tener en cuenta que hay que dejar tiempo suficiente para que los ultrasonidos no generen eco para evitar lecturas erróneas de los sensores colindantes.

En este último aspecto, otro problema que puede existir con la configuración actual es la interferencia que se crea al leer varios sensores que se encuentran en paralelo. Por tanto, es necesario idear otra estrategia que, o bien lea cada uno de forma salteada para

que las ondas del mismo lado no afecten, o cambiar la distribución física para que el eco producido por unos no afecte a los otros.

Desde el punto de vista de la implementación física, se puede observar que existe una pequeña deriva del robot hacia la derecha debido a que uno de los engranajes de la reductora posee un diámetro ligeramente mayor, o un motor no llega a la velocidad nominal del otro. Por tanto, es necesario compensar esta deriva calibrando un valor diferente para cada motor cuando avance hacia delante. Además, la ubicación de los sensores ultrasonidos no es muy adecuada ya que al sobresalir sobre la estructura del vehículo pueden ser golpeados con facilidad contra algún obstáculo.

# **Capítulo 4**

## **Iteración 2**

Una vez visto los errores de la versión anterior, en este capítulo se tratará de realizar las mejoras necesarias para contrarrestar, o eliminar, los problemas contemplados anteriormente. Estos cambios tienen que ver con distintos campos que no se reducen únicamente a la electrónica, pero que son importante comentar.

En términos generales, la iteración 2 está basada en el artículo *The bubble rebound obstacle avoidance algorithm for mobile robots* [22]. En comparación con la versión 1, uno de los cambios más notorios está relacionado con la ubicación de los sensores y la estructura general del robot, al igual que la estrategia programada. Sin embargo, el estudio cinemático del mismo se mantiene inalterado ya que los motores y las ruedas se han mantenido.

La documentación necesaria para esta versión se encuentra disponible en los anexos B.4 y C.2.

### **4.1 Cambios estructurales**

Una de las diferencias más notorias es la forma del chasis. Durante la primera versión, la base sobre la que se partía consistía en una tabla de madera rectangular de 50 cm de ancho por 70 cm de largo, pero esto generaba problemas relacionados con la inercia que hacían el robot difícil de controlar. Esto es debido a que el centro de gravedad no coincidía con el geométrico por lo que el vehículo tiende a seguir una trayectoria debido a la inercia. Además, con esta configuración las cuatro esquinas pueden tropezar con objetos y aumentan el radio que necesita el vehículo para girar, perdiendo maniobrabilidad. Por tanto, para evitar todos estos problemas el sentido de avance del

misma ha cambiado hacia el otro lado, así como el largo del vehículo ha sido reducido. De esta manera el robot pasa a ser de tracción trasera.

Esta nueva morfología sigue el mismo principio de dos motores que funcionan como un diferencial, sin embargo, ahora el chasis tiene forma circular (ver anexo C.2). Para poder reutilizar el chasis de la primera versión, se ha trazado un círculo con origen en el centro del eje diferencial con radio igual a la distancia a una de las esquinas traseras. Por tanto, de esta manera se garantiza que sigue pudiendo cumplir las condiciones dadas en la sección 1.1.2 acerca de poder pasar por pasillos de al menos 1.40 m de ancho. Al realizar este ajuste, el eje de la rueda loca, ahora delantera, está ubicada lo más pegada posible al eje sin que toque ninguna rueda. Este ajuste se ha realizado usando razones trigonométricas, permitiendo dejar 5 cm de seguridad con las otras ruedas. Por tanto, en esta nueva versión se ofrece un vehículo cuyo volumen es mínimo.

En cuanto a la ubicación de la electrónica, esta se encuentra en un mismo compartimento aislado del exterior. Aunque en un inicio esta caja era de categoría IP56, ahora con los orificios para sacar los cables por la parte inferior, debería aún ser capaz de proteger la circuitería del agua en la mayoría de situaciones. Siguiendo la idea de optimizar el espacio al máximo, esta caja ahora se ubica en posición vertical sobre el soporte del eje de la rueda delantera. Esto permite ganar espacio en la zona central del robot donde se pretende colocar baterías de más capacidad que permitan aumentar la autonomía en un futuro. Con estas modificaciones también se colocan algunas interfaces para conectar y desconectar sin necesidad de tocar el cableado interior, así como un interruptor de tres estados para apagar, encender, o cargar el robot.

Dentro de esta caja se han dispuesto verticalmente las tarjetas de los distintos módulos para poder ganar el máximo de espacio, intentando garantizar que los cables sean accesibles en el interior. Para ello se han diseñado e impreso con una impresora 3D unos modelos que actúan de soporte. La idea está basada en la referencia [9] con algunas modificaciones. Los anexos C.4, C.5 y C.6 muestran los planos de estos modelos.

Como se puede ver en la imagen 4.1, el circuito se ha protegido con un fusible rápido de 25 A justo después de la batería. Este valor se ha escogido así ya que la intensidad de rotor bloqueado de los motores es de aproximadamente 12 A (ver sección 2.2.1).

Además, en el plano de planta del anexo C.2 se puede observar que ahora los ultrasónidos se han ubicado de forma radial con un desfase de 20 grados en la parte frontal, y protegidos con una carcasa diseñada específicamente para este proyecto. Como ahora

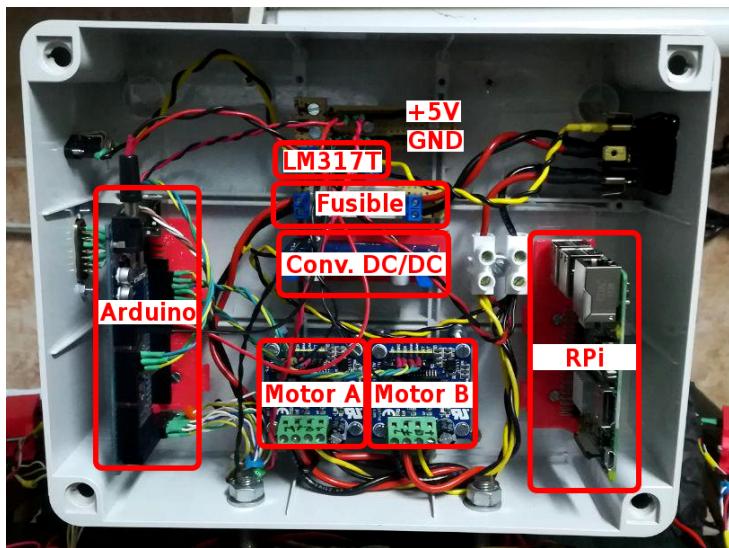


FIGURA 4.1: Electrónica de la iteración 2.

no están en las aristas del robot, los sensores están ahora mucho más protegidos de golpes o el clima. Los planos del anexo los anexos C.7 y C.8 muestran las vistas del modelo 3D diseñado.

El motivo por el que se escogió colocar ocho ultrasonidos equiespaciados por un ángulo de 20 grados está directamente relacionado con la figura 2.10 del capítulo 2. Como se describe allí, el ángulo máximo en el que se obtienen medidas es algo más de 22.5 grados, pero se establece en 15 la apertura más óptima. No obstante, forzando un poco esas características, a 20 también se pueden obtener lecturas adecuadas. Por tanto, la visión del robot queda cubierta en los 180 grados que establece el artículo nombrado al inicio de este capítulo. La visión resultante de esta configuración queda representada en el anexo C.3. Se observa que, aunque cubierta por completa, está en realidad desplazada algo menos de 9 cm hacia delante, además de que existen puntos muertos donde no se detectaría ningún obstáculo. Por tanto, con esto podemos definir que la distancia de seguridad mínima es el punto de cruce entre las visiones de ambos sensores.

Esta nueva localización ha hecho que otros componentes electrónicos que son susceptibles de alterar su medida debido a elementos metálicos cercanos han sido ubicados en una posición alejada para minimizar estos efectos. Este es el caso de la brújula, y en menor medida, del GPS. Se ha podido medir experimentalmente que el primero debe estar separado al menos 30 cm de cualquier otro cuerpo metálico, y que el segundo debe estar ubicado en un lugar donde no lo cubra ningún objeto con la misma propiedad.

Además, es importante recordar que estos elementos deben, en la medida de lo posible, ubicarse lo más cercanos al centro geométrico. Esto es deseable para que cuando el robot gire sobre sí mismo, la medida no sufra traslación y rotación a la vez. Análogamente, es posible decir algo parecido del módulo GPS. Esto es, como este marca las coordenadas que ubican el vehículo frente al espacio, es razonable querer que esté situado lo más al centro posible. La figura 4.2 muestra el estado actual del robot en esta versión.



---

FIGURA 4.2: Estado del robot en la iteración 2.

## 4.2 Cambios de electrónica

En esta sección se tratará de explicar los cambios que han sido oportunos para intentar mejorar el comportamiento del robot desde el punto de vista del hardware. Con este objetivo, algunos módulos han sido eliminados o reemplazados.

Con el planteamiento mostrado en la figura 4.3 se pretende distinguir las tareas básicas de obtención de datos o transmisión de comandos a los actuadores, de aquellas que no tienen una acción tan instantánea, necesariamente. De esta manera se aprecia que ahora

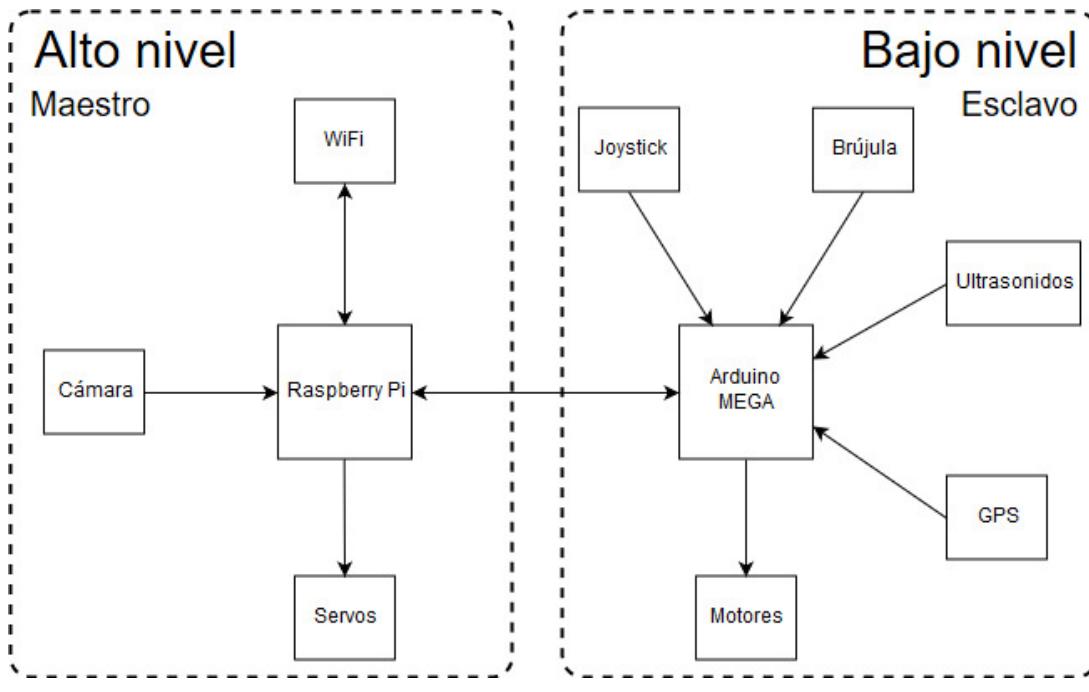


FIGURA 4.3: Estructura electrónica de la iteración 2.

en vez de tener un solo microcontrolador encargado de todas las tareas, se incluye en un nivel jerárquico superior un ordenador que actuará de coordinador. Por tanto, con esta configuración el Arduino pasa a ser un mensajero con reducidas capacidades de decisión de alto nivel.

Otros módulos como el ESP8266 han sido eliminados, mientras que otros han sido añadidos para cumplir con los requisitos planteados en el capítulo 1. Uno de los motivos que han llevado a la supresión del primero es que generaba bastantes problemas de comunicación, e incluso a veces no inicializaba adecuadamente el punto de acceso. Por tanto, la falta de confianza en que el módulo inicie en el momento adecuado, y el problema que supone que el Arduino tenga que lidiar también con interrupciones para recibir comandos por la red inalámbrica, hacen que sustituir el ESP8266 por otra alternativa parezca más razonable. Cabe recordar que el Arduino, como se comentó en la sección 2.1.3, es un dispositivo que solamente es capaz de leer secuencialmente una lista de instrucciones. Es decir, salvo con algunos ajustes de programación por medio de interrupciones, difícilmente es capaz de ser concurrente.

#### 4.2.1 Raspberry Pi

Este módulo que se añade realiza la función de ordenador de a bordo en el robot, y actúa como cerebro central de alto nivel. Esta medida toma especial sentido ya que, aunque tradicionalmente los ordenadores han sido dispositivos voluminosos y costosos, hoy en día existe una oferta amplia de miniordenadores, SBCs, que ofrecen prestaciones suficientes a un precio asequible. Este tipo de computadores ofrece la ventaja de que todo se encuentra en una sola PCB del tamaño de un Arduino, por lo que, además, es fácil alojarlos en espacios reducidos.

A esta idea hay que sumarle que son dispositivos que funcionan con su propios SO y que por su arquitectura son capaces de manejar distintos *threads* y tareas simultáneas. Además, vienen equipados con módulos inalámbricos como WiFi o bluetooth que son controlados por el SO mediante procesos y servicios. Otra ventaja significativa es que, a diferencia de los Arduino, tienen memoria. Esto implica que es posible almacenar información que no va a ser utilizada en el instante de tiempo que se recibe. Incluso se puede guardar para utilizarla cuando las tareas más prioritarias hayan acabado de ejecutarse, consiguiendo así un uso más inteligente y eficiente de los recursos.

Sin embargo, el uso de un SBC también aporta sus lados negativos ya que aumenta el gasto energético del robot, mermando su autonomía. Esto es producido porque, a diferencia de un Arduino, tiene que inicializar más capas de proceso antes de estar listo para operar, por lo que el arranque del ordenador es más lento. Es decir, a la hora de implementarlo, el Arduino tendrá que esperar al ordenador para empezar a ejecutar su *sketch*.

Partiendo de toda esta base, en este proyecto se ha escogido una Raspberry Pi 3 Model B como ordenador de a bordo. El motivo por el que se escogió este SBC fue influenciado por su popularidad, y por la compatibilidad con otros módulos. Al ser famoso existe bastante documentación disponible para desarrollarlo, además de que ofrece prestaciones, a priori, suficientes para este proyecto a un precio asequible.

Algunas de las características de este modelo que resultan interesantes para este proyecto son:

- 40 pines GPIO.
- Alimentación a 5 V.

- 4x USB.
- Ethernet a 100M e interfaz WiFi incorporada.
- Almacenamiento SD hasta 32 GB.
- Conexión FFC para cámara.



---

FIGURA 4.4: Raspberry Pi 3 Model B. Fuente: [raspberrypi.org](http://raspberrypi.org)

Entre las ideas comentadas anteriormente, para el diseño electrónico del vehículo es importante destacar que, aunque la tensión de trabajo es la misma que el Arduino, los requerimientos de potencia son mucho mayores. Por tanto, y como la Raspberry Pi no admite más de 6 V de alimentación, es necesario instalar un regulador a la entrada del mismo que adapte el voltaje a este nivel. El problema es que los reguladores lineales comunes que ya se han usado, como el LM317T, están limitados para suministrar menos corriente [1]. Como esta placa consume entre 2 A y 2.5 A pico en el arranque, la solución adoptada es, entonces, usar un convertidor DC/DC, *buck converter*, que pueda suministrar la potencia suficiente. En concreto, el módulo utilizado es el XY-3606 ya que ofrece varias interfaces que pueden suministrar hasta 5 A en total (ver figura 4.5).

En cuanto al apartado de *software*, se ha instalado una imagen *lite* y personalizada de Raspbian con ROS Kinetic y OpenCV 3.4.1 [19]. Entre los motivos que hacen interesante esta configuración está satisfacer algunos objetivos finales del proyecto comentados en la sección 1.1.1, y porque además simplifica la tarea de preparación del SO. El uso de esta *distro* de Linux ha hecho que sea más sencillo configurar Python3 con las librerías de visión computacional, y otras configuraciones de red, dada la extensa documentación que existe al respecto. Esto se debe a que, al estar basado en Debian y tener soporte oficial, se garantiza que haya documentación actualizada y apoyo de la comunidad para solucionar problemas. Finalmente, tener ROS instalado es, por ahora, un extra que

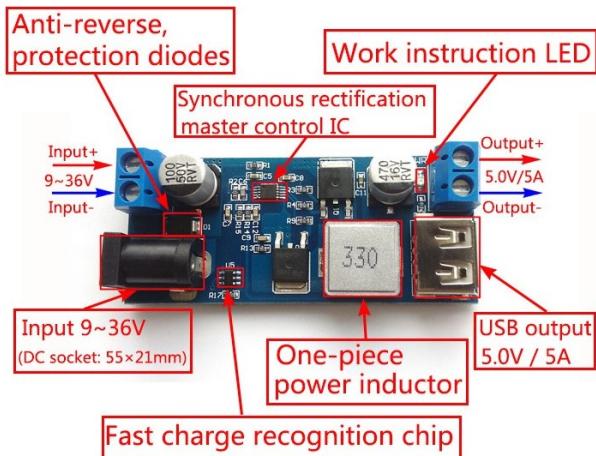


FIGURA 4.5: *Buck converter* a 5 V. Fuente: aliexpress.com

podrá usarse en versiones posteriores del proyecto, pero que por ahora quedan fuera del alcance de este TFG.

Otro aspecto importante que hacen necesaria esta actualización, como se comentó en la sección 3.6, es que con este ordenador es posible depurar el programa de forma más efectiva ya que uno se puede conectar por SSH para poder monitorizar el puerto serial del Arduino sin tener que perseguir el robot. Incluso, aprovechando la memoria del propio SBC, se puede volcar la información de este a un archivo que luego se consulta a modo de histórico.

No obstante, si tenemos en cuenta algunas condiciones que existen en el lugar actualmente (ver sección 1.1.2), es necesario que para poder realizar lo comentado en el párrafo anterior se configure como punto de acceso la interfaz WiFi que viene incluida en la placa. Esto es debido a que no existe una red local con acceso a internet desplegada a la que acoplarse para crear el túnel SSH al robot. La creación de este AP implica que el robot administra su propia red local, pero sin salida a internet para descargar los datos que se le manden. Por tanto, esta solución es solo adoptada para depurar el programa en un lugar en el que no haya una red.

Adicionalmente, y de forma excepcional, se puede añadir una segunda interfaz que se conecta a una red previamente configurada con acceso a internet. Es decir, la que viene por defecto crea un AP, y la segunda daría acceso a internet.

Otra opción más eficiente desde el punto de vista energético sería hacer un programa que compruebe si existe la red preconfigurada. En caso afirmativo se conectaría a esta,

de lo contrario, configura la interfaz para que haga de AP. Finalmente, esta solución ha sido desestimada por no ser prioritaria en este trabajo.

Por tanto, con todo esto es posible prescindir del módulo WiFi debido a que la Raspberry Pi aporta la conexión inalámbrica. Otra ventaja de esta medida es que simplifica el circuito resultante dado que, al eliminar este módulo que funcionaba con 3.3 V, también es posible quitar el regulador y las resistencias que suplen al ESP8266 de su nivel de tensión adecuado.

#### 4.2.2 Cámara

Un complemento necesario para el objetivo de este proyecto es la posibilidad de obtener imágenes del lugar. Para ello, es necesario usar una cámara que vaya montada en la parte delantera, y que además sea capaz de capturar, no solo el espectro visible, sino parte del infrarrojo para poder calcular el NDVI (ver sección 1.1.1).

La importancia de estos parámetros viene dada por el sensor CCD que lleva dicha cámara. En este caso se ha escogido, por conveniencia, una cámara con compatibilidad oficial con Raspberry Pi que utiliza el sensor OV5647 (ver imagen 4.2.2). También puede usarse para visión nocturna, que precisamente funciona con el infrarrojo cercano, por lo que con tratamiento de imágenes debería ser posible de obtener el NDVI.



FIGURA 4.6: Cámara con sensor OV5647. Fuente: aliexpress.com

Este sensor se alimenta con los 3.3 V de la placa, a la que se conecta directamente por el conector FFC disponible al lado de la entrada HDMI. Cuenta con una resolución de 5 megapíxeles y un campo de visión que barre 130 grados. Además, esta versión

posee también la posibilidad de colocar dos luces infrarrojas en los extremos que permiten usar la cámara por la noche tras regular adecuadamente el umbral de luz del fotodetector que las activa.

Para el tratamiento de imágenes se usa la librería OpenCV con Python3 aprovechando el código desarrollado por Robintw [18], al que se le han hecho algunas modificaciones menores para ajustarse a este proyecto. Este código modificado puede consultarse en el anexo A.4.

Sin embargo, pese a que se ha avanzado en estos aspectos, todavía no se ha ubicado físicamente en el modelo real. Este es el motivo por el que en el diagrama de bloques de la figura 4.3 aparecen unos servos que todavía no han sido montados.

# Capítulo 5

## Implementación de la iteración 2

En este capítulo se presenta la implementación realizada para la iteración 2 desde el punto de vista del software. Para ello, se debe tener en cuenta la estructura explicada en el capítulo 4, en la que existen dos niveles de actuación dependiendo de las funciones asignadas.

### 5.1 Alto nivel - Raspberry Pi

La parte encargada de realizar tareas que requieren de varios procesos que corren de forma independiente es denominada en este proyecto como de alto nivel. Ejemplo de estas son el gobierno de los otros niveles o la comunicación remota con el mundo exterior. Por tanto, la idea es conseguir cambiar el comportamiento del robot desde el exterior sin tener que modificar el programa.

Para realizar estas tareas, como se explicó anteriormente, se utiliza una Raspberry Pi que ejerce de maestro sobre el Arduino, que sería el bajo nivel del robot (ver sección 5.2). Existen dos modos de operación principales dentro del vehículo. Por un lado hay un interruptor principal que hace la función de *enable* que permite la actuación del robot, siempre y cuando esté a un uno lógico. La necesidad de este interruptor es debida a que el tiempo de inicio de un Arduino es menor que el de la Raspberry Pi, por lo que el primero empezaría a operar antes de que el segundo esté listo. Además, el uso de este pin no implica el corte de la corriente de la batería, simplemente es un interruptor por *software* que también puede ser pulsado en emergencias. Es importante recordar que para cortar la alimentación de la batería, ya existe un interruptor físico de tres estados designado para esta tarea, entre otras. Por otro lado, el otro pin configurado se

encarga de alternar entre el modo de control automático y manual. La diferencia entre uno y otro es que el primero es capaz de navegar de forma autónoma a partir de un recorrido dado, y el segundo permite controlar el robot localmente, tras conectar el *joystick* por el puerto DA-15.

Como se puede ver en el anexo B.4, con el objetivo de poder alternar entre estos modos de operación, se definen y conectan unos pines concretos en cada placa que funcionan como interruptores. La tabla 5.1 muestra la numeración de los pines asignados para cada una de las placas.

	Arduino	Raspberry Pi
Enable	48	26
Modo	49	19

CUADRO 5.1: Asignación de pines según placa

Por un lado, para elegir los pines del Arduino no hace falta más que escoger entre los libres y asignarlos como *INPUT* en el código (ver código 5.1). Esta configuración permite que estos se pongan en un estado de alta impedancia, es decir, que la corriente que circula por ellos es mínima. Por este motivo es posible conectar los pines directamente, sin pasar por una resistencia, ya que internamente existe una en serie que realiza esta tarea.

---

```

1 // Defining high-level control pins
2 const int enable_pin = 48;
3 const int mode_pin = 49;
4
5 // Configuring high-level control pins
6 pinMode(enable_pin, INPUT);
7 pinMode(mode_pin, INPUT);
```

---

CÓDIGO 5.1: Configuración de los pines de control del Arduino

Sin embargo, la asignación de pines en la Raspberry Pi debe realizarse con mayor cuidado. Esto es, dado que son pines que deben gobernar por encima de la otra placa, es necesario asegurar que en ningún momento del inicio se ponen en un estado desconocido o contrario al deseado. En este proyecto se ha considerado que el *enable* sea activo a altas, por lo que durante el inicio del SO el pin designado debe mantenerse en *low*. En el manual técnico del procesador de la Raspberry Pi [5] se recoge en su sección 6.2

una tabla con los estados por defecto de cada uno de los pines. Por tanto, se asignan los pines GPIO19 y GPIO26 para el modo y el *enable*, respectivamente.

Otro punto importante a tratar es el nivel de voltaje lógico de una y otra placa. Por un lado, el Arduino utiliza una lógica de 5 V, por otro, la Raspberry Pi es de 3.3 V. No obstante, el aspecto relevante es el nivel de voltaje a partir del cual se considera un uno lógico o un cero. Atendiendo a la documentación del Arduino, se observa que el nivel mínimo de voltaje que se establece para un uno es de 3 V por lo que, dado que la Raspberry ofrece 3.3 V cuando se activa a altas, no es necesario adaptar los niveles en ningún momento. De esta manera la interconexión entre ambos módulos es realizada con solo un cable.

Para poder cambiar los estados en el inicio hay que programar un *script* que corra y configure los pines cada vez que se inicia el ordenador. Aunque existen varias maneras de hacerlo, en este caso se ha optado por hacer un *script* en Bash que inicia el pin 19 y 26 y los setea a un estado por defecto (ver código 5.2). Desde un principio se descartó usar un programa en Python ya que, al tener que correr continuamente en segundo plano, no resultaba sencillo encontrar una manera de alterar los valores con una palabra escrita desde la consola. En este sentido, parecía más directo usar bash y configurar unos aliases.

Para que, entonces, el programa pueda correr desde el inicio hay que ejecutarlo desde la dirección /etc/rc.local como *script* de Bash. También es importante arrancarlo con un ampersand para que se ejecute en segundo plano, y evitar así que el SO pueda no continuar el inicio si se cuelga el programa.

---

```
1 #!/bin/bash
2 # Export pins to userspace
3 sudo echo "26" > /sys/class/gpio/export
4 sleep 0.1
5 echo "19" > /sys/class/gpio/export
6 sleep 0.1
7 # Set pins as an output
8 sudo echo "out" > /sys/class/gpio/gpio26/direction
9 sleep 0.1
10 echo "out" > /sys/class/gpio/gpio19/direction
11 sleep 0.1
```

---

```

12 sudo echo "1" > /sys/class/gpio/gpio26/value # Enable = 1
    (default); Disable = 0
13 sleep 0.1
14 echo "0" > /sys/class/gpio/gpio19/value # Manual = 0 (default);
    Automatic = 1

```

---

CÓDIGO 5.2: Programa bash para inicializar los GPIO

Solo tras haber ejecutado el código 5.2, el Arduino puede comenzar a trabajar. No obstante, si luego se desea parar el robot, por ejemplo, habría que escribir un cero en el pin 26. El problema está en que es necesario conectarse al ordenador para mandar un comando que de dicha orden. Dado que en ocasiones no hay acceso a internet, la solución adoptada pasa por crear un punto de acceso local, usando la interfaz wlan0 del ordenador, y conectarse a ella por SSH. De esta manera, la Raspberry Pi ejerce la función de router, ofreciendo las IPs con su DHCP. Para realizar esto se instalaron y configuraron algunas aplicaciones que hacen de servidor DHCP y de *host* para el AP.

Tras conectarse a la Raspberry Pi es posible escribir el comando que cambia este estado. No obstante, resulta útil hacer uso de los alias que ofrece el SO Linux, asignando una palabra clave a un comando concreto para no tener que enviarlo entero a mano. Para hacer esto basta con modificar el archivo .bash\_aliases de la carpeta raíz del usuario que se conecta (ver código 5.3).

---

```

1 alias en='echo "1" > /sys/class/gpio/gpio26/value'
2 alias dis='echo "0" > /sys/class/gpio/gpio26/value'
3 alias manu='echo "0" > /sys/class/gpio/gpio19/value'
4 alias autom='echo "1" > /sys/class/gpio/gpio19/value'

```

---

CÓDIGO 5.3: Configuración de alias

Es importante anotar que los alias que se otorgan no deben coincidir con palabras reservadas dentro del propio SO. Por tanto, una vez que el sistema está inicializado basta con enviar el alias correspondiente a lo que se desee hacer.

Una última mejora que se proponía en la sección 3.6 es tener la capacidad de leer la consola serial del Arduino remotamente, o al menos poder guardar sus datos en un archivo. Para ello se ha configurado un *script* de Python que imprime este puerto por pantalla (ver código 5.4). Otra opción que resulta más útil es redirigir la salida de este

programa a un archivo dentro de la Raspberry donde se vuelca toda la información. De esta manera se guarda la información de cada prueba en un archivo, haciendo posible analizar los datos a posteriori, si fallara algo. Este proceso de *debug* es preferible frente al otro en el que hace falta estar conectado por un cable a un vehículo en movimiento. Particularmente porque los cables USB tienden a fallar si son demasiado largos.

---

```
1 #!/usr/bin/python
2 import serial
3
4 ser = serial.Serial('/dev/ttyACM0', 115200)
5 while True:
6     read_serial=ser.readline()
7     print read_serial
```

---

CÓDIGO 5.4: Programa Python para leer el puerto Serial del Arduino

## 5.2 Bajo nivel - Arduino

En la otra parte dentro de la estructura electrónica de esta versión se encuentra el Arduino como microcontrolador que se conecta a los sensores y los actuadores. La misión de este no es solo obtener o escribir datos, sino también la de llevar el algoritmo para navegar de forma automática y manual.

Por tanto, al iniciar, este espera a que la Raspberry Pi esté lista. Esto es, leer el pin 48 que está configurado como *INPUT*, hasta que se ponga a uno, es decir, cuando el robot esté *enable*. Una vez hecha esta comprobación se leerá el pin 49 que establece si está en modo manual o automático para decidir en qué bucle debe entrar.

### 5.2.1 Modo manual

En esta versión del proyecto el modo manual se mantiene casi inalterado con respecto a la versión anterior. Su objetivo sigue siendo el de obedecer las órdenes de un operario que maneja un *joystick*. No obstante, una diferencia que se puede destacar es la ausencia de control remoto. Este cambio responde, simplemente, a que en esta iteración se eliminó el módulo ESP8266 que se encargaba de la comunicación entre el Arduino del *joystick* y el del robot. Sin embargo, sí se ha mantenido el modo de control local con

el que es manejado tras conectar el *joystick* por el puerto DA-15 disponible. Por tanto, dado que esta parte del trabajo no ha sufrido modificaciones posteriores significativas, no resulta necesario comentar nada más acerca de esta sección.

### 5.2.2 Modo automático

El objetivo de este modo de control sigue siendo el de realizar el camino marcado dentro de un mapa, según la localización de sus puntos. Este camino es supuesto, a priori, libre de obstáculos. No obstante, como medida de precaución, el robot debe ser capaz de sortearlos para evitar golpear con ellos, y seguir el rumbo objetivo. El diagrama de bloques de la figura 5.1 representa el algoritmo que permite navegar por cada punto

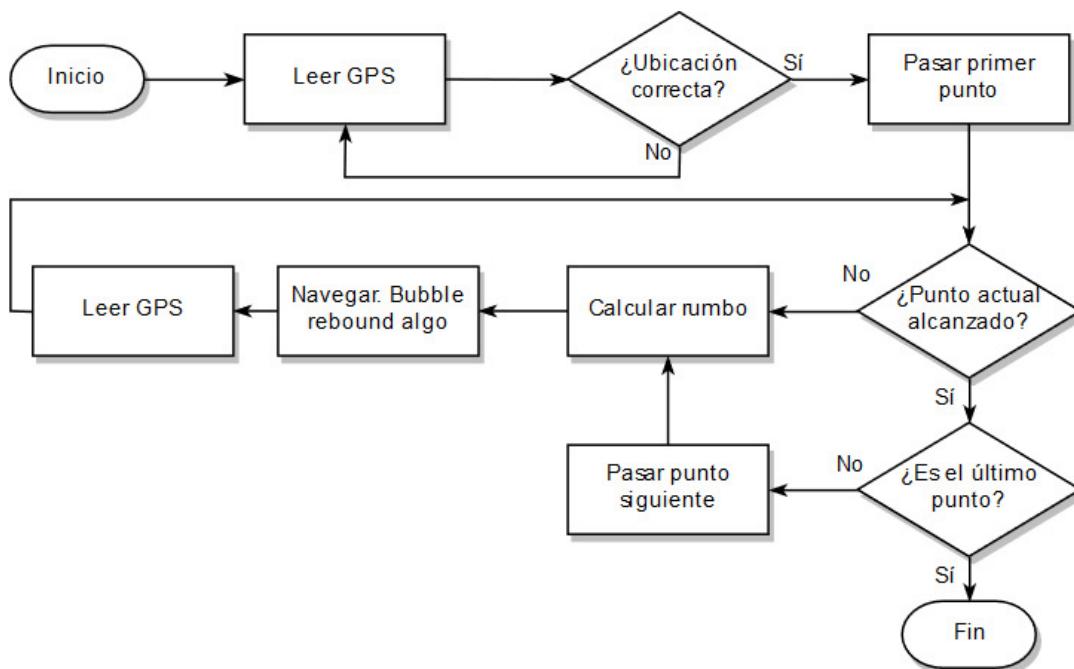


FIGURA 5.1: Diagrama de bloques para la navegación.

En un comienzo, lo primero que realiza es una lectura continua del GPS hasta que recibe una señal correcta. Esto es necesario ya que cuando el módulo inicializa necesita buscar los satélites más cercanos. Dado que se conoce que la ubicación del lugar no es el origen de las coordenadas globales, se puede saber si el valor es el correcto comprobando que este sea diferente de cero. Esta comprobación es la que se muestra en el fragmento de código 5.5.

---

```

1 do{
2     get_location(gps);
3     Serial.print("latitude = ");
4     Serial.print((float)latitude, 6);
5     Serial.print(" ");
6     Serial.print("longitude = ");
7     Serial.println((float)longitude, 6);
8     delay(500);
9 }while(latitude == 0 && longitude == 0);

```

---

CÓDIGO 5.5: Comprobación de que la señal GPS es correcta

La función `get_location()` es utilizada para obtener los valores de latitud y longitud. Para ello, se pasa el objeto GPS y posteriormente se ejecuta la función `f_get_position()`, propia de la librería TinyGPS, que escribe los valores pasados por referencia (ver código 5.6). Los valores obtenidos son en grados decimales, con coma flotante, y hasta seis cifras decimales de resolución.

---

```

1 // Read current position
2 static void get_location(TinyGPS &gps) {
3     while (Serial1.available()) {
4         int c = Serial1.read();
5         if (gps.encode(c)) {
6             // retrieves +/- lat/long in 100000ths of a degree
7             gps.f_get_position(&latitude, &longitude, &fix_age);
8         }
9     }
10 }

```

---

CÓDIGO 5.6: Obtención de las coordenadas GPS

Una vez conocida la ubicación con respecto al entorno, se pueden empezar a pasar las coordenadas de los puntos. Para ello se define una matriz de tantas filas como puntos se requieran, y dos columnas. La primera contiene la latitud, y la segunda la longitud del siguiente punto objetivo. Aunque también podría ser interesante añadir una tercera columna con el rumbo deseado en el punto final, esto no ha sido implementado al no

ser una característica crítica. Por tanto, para hacer el recorrido el controlador itera por cada fila introduciendo los nuevos valores cada vez que ha llegado al punto anterior.

Una pregunta lógica que se deriva del párrafo anterior es en qué momento se considera que el robot ha llegado a un punto. Para hacer frente a esta cuestión se ha definido una función llamada `goal_is_reached()` que devuelve un valor verdadero o falso en función de si hay éxito o no, respectivamente. La manera de discernir esto es comprobar que la diferencia entre el punto objetivo y el actual, en valor absoluto, es menor o igual que un error preestablecido.

Para este proyecto, al igual que para la mayoría de aplicaciones, es suficiente utilizar seis cifras decimales, ya que ofrecen una precisión espacial de 100 mm aproximadamente. No obstante, como se comentó en la sección 2.3.2, la precisión de este tipo de aparatos es afectada por el entorno y el clima. Por tanto, es de esperar que las últimas cifras varíen aún cuando el vehículo esté quieto. Este efecto produce que, si el umbral se ubica en una cifra decimal de mucha precisión, el robot estará eternamente buscando el punto aunque esté sobre él. Por el contrario, si este umbral se ubica en un valor demasiado grande, el efecto es el opuesto. Es decir, se produce una pérdida de resolución espacial que genera una incertidumbre mayor sobre la ubicación real del mismo.

Por tanto, resulta crítico calibrar este valor para que cumpla los requisitos para los que está diseñado. Experimentalmente se ha determinado que un error menor o igual de  $10^{-5}$  grados decimales es un buen valor que permite llegar al punto con solvencia y relativa precisión. Esto se traduce en que el error puede fluctuar un metro a cada lado. Es posible visualizar este problema como un cuadrado ficticio de incertidumbre con un área de  $2 \text{ m}^2$  en el que el sistema puede estar ubicado. El cálculo de la precisión según el número de cifras decimales se obtiene a partir del radio de la Tierra, y la latitud a la que se encuentra el lugar.

Aunque es cierto que esta incertidumbre puede ser demasiado grande para pasar por algunos pasillos y condiciones descritas en la sección 1.1.2, es de esperar que con el algoritmo para evitar los obstáculos pueda sortear el problema.

Una vez conocido el punto al que se debe dirigir el vehículo, el algoritmo calcula el nuevo rumbo para que el robot pueda girar posteriormente. Esto se realiza utilizando la función `gps.course_to()` que viene incluida en la librería TinyGPS. El cálculo del

rumbo en función de dos puntos del espacio se reduce a la ecuación 5.1.

$$\theta = \arctan \left( \frac{\sin(\lambda_2 - \lambda_1) \cdot \cos(\varphi_2)}{\cos(\varphi_1) \cdot \sin(\varphi_2) - \sin(\varphi_1) \cdot \cos(\varphi_2) \cdot \cos(\lambda_2 - \lambda_1)} \right) \quad (5.1)$$

donde  $\lambda$  es latitud,  $\varphi$  longitud en grados decimales, y  $\theta$  el rumbo. Los valores que se obtienen son en radianes, por lo que para ajustarse al sistema de unidades basta con pasarlo a grados para poder navegar con el rumbo correcto.

### 5.2.2.1 Implementación del algoritmo

Para los casos en los que el robot se encuentre con un obstáculo se aplicará el algoritmo para evitar los obstáculos [22]. En la referencia citada se propone un algoritmo sencillo para que un robot móvil *indoor* sea capaz de evitar los obstáculos que se encuentre en el camino usando los ultrasonidos dispuestos en la parte frontal.

La idea principal consiste en seguir un rumbo y comprobar que no existen obstáculos contra los que pueda golpear dada la velocidad y el periodo hasta la siguiente evaluación. Si este no fuera el caso, es decir, si el robot determina que existe posibilidad de que golpee, entonces calcula un ángulo de rebote. Este es aquella dirección hacia la que existe la menor densidad de obstáculos, de acuerdo con las lecturas de los sensores. Por tanto, una vez calculado el nuevo rumbo, el robot girará sobre sí mismo y volverá a evaluar la posibilidad de avanzar hasta que no haya problemas. Una vez esto pase, avanzará recto mientras comprueba que en la dirección objetivo no está bloqueado por ningún obstáculo. En el momento en el que este sea capaz de ver este punto, el robot girará sobre sí mismo para colocarse rumbo al punto designado, hacia el que avanzará recto si no vuelve a encontrarse ningún problema en el camino.

No obstante, existen algunas diferencias importantes entre la implementación original, y la que se propone en este trabajo. Una de ellas es el uso de lógicas diferentes en uno y otro. Esto es, en el robot usado para escribir el artículo se usó un sistema de control con lógica difusa, mientras que en este proyecto se utiliza lógica booleana con ciertas tolerancias. Por tanto, no existe posibilidad de pertenecer en mayor o menor grado a una categoría u otra, como ocurre con la lógica difusa, sino que solo puede ser miembro, o no, de una.

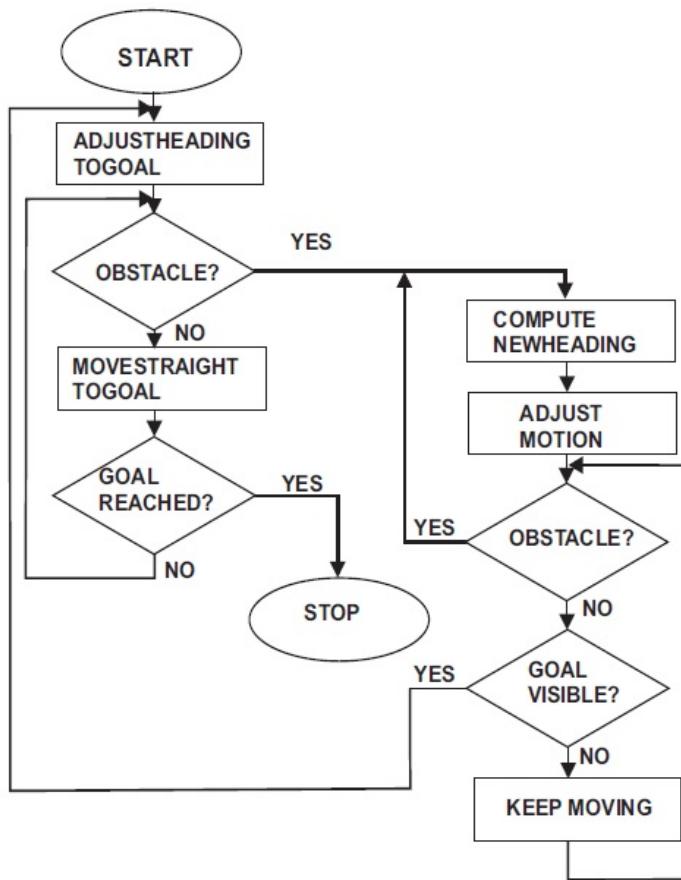


FIGURA 5.2: Algoritmo para evitar obstáculos en la iteración 2. Fuente: [22]

En relación al artículo, cabe comentar que aunque la idea original propone utilizar la velocidad para poder calcular si hay espacio suficiente como para no golpear en la siguiente iteración, estos aspectos no se han tenido en cuenta en este proyecto. Esto es debido a la imposibilidad de obtener un valor preciso y fiable de velocidad ya que no se planteó usar ningún acelerómetro para calcularlo. Por otro lado, también es verdad que se puede utilizar un modelo matemático con el que, sabiendo el comando de las ruedas en ese instante, se pueda estimar la velocidad (ver sección 1.2.1). No obstante, este método, aparte de no estar sujeto a las perturbaciones propias del entorno, necesita conocer también el voltaje de la batería en ese instante. La estrategia matemática sería aprovechar que los motores son de corriente continua y que, por lo tanto, su velocidad de giro es directamente proporcional al voltaje aplicado. Entonces, sabiendo la consigna actual y el voltaje de la batería se puede inducir la tensión en cada motor, y

por tanto, la velocidad lineal y angular del vehículo sin perturbaciones. En cualquier caso, actualmente tampoco existe un partidor de tensión que pueda hacer la función de voltímetro, por lo que esta posibilidad es también descartada con el montaje actual.

Con respecto a la versión 1, esta vez se ha procurado programar el robot manteniendo por un lado funciones propias de la instrumentación o el tratamiento de datos, y por otro aquellas que sirven para asignar comandos en los motores. Es decir, las que producen un cambio físico en la pose del vehículo. Muchas de estas han sido reutilizadas o adaptadas de la versión 1 por lo que no se comentarán más en este capítulo. Este es el caso de `calculate_heading()` y `calculate_distance()` (ver fragmentos de código 3.3 y 2.4).

Antes de empezar a seguir el diagrama del algoritmo expuesto en la figura 5.2, lo primero que realiza el robot es inicializar los módulos. En el montaje que se presenta en esta versión se tiene una cadena de ultrasonidos que barren los 180 grados del espacio delantero del vehículo. La idea principal es, entonces, leer cada uno de ellos y almacenar los datos en un *array*, de tal manera que, si se pintaran, se obtendría un histograma de la distancia en función del ángulo. Sabiendo esto, y que son dos pines digitales los necesarios para el proceso de sensado (ver sección 2.3.4), es lógico asignarle pines consecutivos para poder crear un bucle que itere a lo largo de todos ellos y obtener así un código más óptimo. Es decir, haciendo esto solo se necesita definir un solo sensor ya que los pines del resto van a razón de dos veces el índice del primero. La inicialización queda, por tanto, descrita en el fragmento de código 5.7.

---

```

1 const int EchoPin_0 = 24;
2 const int TriggerPin_0 = 25;
3
4 // Configuring pins
5 for(int i = 0; i < 8; i++) {
6     // Iterate every sensor. Every two is a new sensor pin
7     pinMode(EchoPin_0 + (2 * i), INPUT);
8     pinMode(TriggerPin_0 + (2 * i), OUTPUT);
9     delay(1); // Stabilization
10 }
```

---

CÓDIGO 5.7: Configuración de los pines de los ultrasonidos

La inicialización del resto de módulos es la misma que en la versión 1, por lo que no será necesario comentar nada más acerca de ellos.

En el comienzo del algoritmo lo primero que realiza el robot es calcular su orientación actual y leer los ultrasonidos para conocer la distancia a la que se encuentran los posibles obstáculos. Dado que la información que se recibe de estos sensores es vital para el desarrollo correcto, resulta necesario garantizar que los datos que proveen son precisos y exactos dentro de una tolerancia.

El proceso de sensado de un solo ultrasonido puede tardar, aproximadamente, 38 ms si no existe obstáculo [25], y es igual al de la iteración 1. Por ello, se ha reutilizado la misma función, `calculate_distance()`. Por tanto, si ahora hay una cadena de sensores que se leen sin esperas se podría tardar 304 ms en leerlos todos. No obstante, dado que están equiespaciados por solamente 20 grados entre sí, es probable que el eco de unos afecte a los otros ya que al final son módulos iguales que emiten a la misma frecuencia. Por tanto, otra idea que hay que introducir para obtener buenos valores es la de esperar a que se disipen las ondas de presión que han emitido los otros. Para hacer esto, el *datasheet* establece que la frecuencia máxima debe ser de 20 Hz, o lo que es lo mismo, 50 ms [25]. Por tanto, si introducimos un *delay* de este valor, solo la lectura de los valores llevaría, al menos, 400 ms.

Para poder reducir este periodo de sensado es necesario utilizar otra estrategia que permita aumentar la frecuencia de muestreo sin afectar, significativamente, a las medidas. La manera de solventarlo es tomar medidas de sensores salteados, procurando escoger aquellos que tengan menos probabilidades de verse afectados por las ondas colindantes. Usando esta idea, la secuencia de índices vista desde la planta empieza de derecha a izquierda y va saltando de tres en tres, obteniendo 60 grados de diferencia entre sensores. Así se consigue reducir a 80 ms el periodo entre uno y otro si se establecen 10 entre cada uno, aunque probablemente sea posible reducir este valor aún más. Además, pese a que la medida de los sensores se haga salteada, los valores deben guardarse ordenados según su índice. Siguiendo esta idea, la función que los almacena en la posición adecuada del *array* queda recogida en el código 5.8.

---

```
1 // Store ultrasonic sensor readings. Sequence: 0 3 6 1 4 7 2 5
2 unsigned int *read_sensors(){
3     //Initialize to zero.
4     static unsigned int sonar_readings[8] = {0};
5     int i = 0;
```

```

6     int cont = 0;
7     while(cont < 8){
8         sonar_readings[i] = calculate_distance(TriggerPin_0 + (2 *
9             i), EchoPin_0 + (2 * i));
10        if(cont == 2){
11            i = 1;
12        }
13        else if(cont == 5){
14            i = 2;
15        }
16        else{
17            i += 3;
18        }
19        cont++;
20    }
21    return sonar_readings;
22 }
```

CÓDIGO 5.8: Lectura del *array* de ultrasonidos

Para justificar que esta solución no altera las medidas, una prueba sencilla consiste en calcular el coeficiente de variación de uno y otro método con el mismo tiempo de muestreo, en este caso, 5 ms. Es importante destacar que esta prueba es estática y no tiene por qué reflejar los casos en los que un sensor es reflejado por otro. No obstante sí que demuestra una mejora en la desviación de los datos, siendo 2.85 % en el primero y 2.3 % en el segundo. Las tablas 5.2 y 5.3 recogen los datos de ambos métodos.

De igual manera que se hizo para las lecturas del *joystick* en la iteración 1, para estos sensores también se ha programado una función, *average\_read\_sensors()* que realiza la media de dos lecturas para filtrar los valores. Esta medida fue tomada tras observar que aparecían valores aleatorios de forma esporádica que seguramente estaban influenciados por el ruido de alguna señal. La introducción de este filtrado es realizable al haber conseguido reducir de forma significativa el tiempo de muestreo de los sensores. El fragmento de código 5.9 muestra la función que realiza este proceso.

---

```

1 // Filter sensor readings by doing the average
2 unsigned int *average_read_sensors(){
```

Sensor (i)	0	1	2	3	4	5	6	7
	101	115	105	57	57	76	35	37
	120	113	106	56	55	75	35	37
	99	115	105	57	57	76	35	37
	119	116	106	56	56	90	35	37
	121	114	105	56	57	90	35	37
	93	117	105	57	54	76	35	37
	100	115	105	57	56	89	35	37
	95	116	105	57	57	89	35	37
	100	115	106	57	57	90	35	37
	99	115	105	57	58	90	36	37
	100	118	105	57	56	90	36	37
Media	104,27	115,36	105,27	56,73	56,36	84,64	35,18	37,00
Desviación	9,90	1,30	0,45	0,45	1,07	6,73	0,39	0,00
C. V.	9,49 %	1,13 %	0,42 %	0,79 %	1,90 %	7,95 %	1,10 %	0,00 %
Intervalo	5 ms						Media	2,85 %

CUADRO 5.2: Resultados de la medida de los ultrasonidos sin saltar.

```

3     unsigned int sum[8] = {0}; //Initialize to zero.
4     static unsigned int average[8] = {0}; //Initialize to zero.
5     unsigned int *sonar_readings;
6
7     for(int j = 0; j < 2; j++) {
8         sonar_readings = read_sensors();
9         for(int i = 0; i < 8; i++) {
10             sum[i] += sonar_readings[i];
11         }
12     }
13
14     for(int k = 0; k < 8; k++) {
15         average[k] = sum[k]/2;
16     }
17
18     return average;
19 }
```

CÓDIGO 5.9: Filtrado de las lecturas de los ultrasonidos

Sensor (i)	0	1	2	3	4	5	6	7
	119	100	105	55	56	76	35	36
	119	116	105	55	57	84	35	37
	121	99	105	55	56	84	35	36
	120	116	105	55	55	84	35	37
	119	115	105	55	56	83	35	36
	121	115	105	55	57	84	35	37
	120	116	105	55	56	76	35	37
	121	117	105	55	56	84	35	37
	100	116	105	55	56	84	35	37
	121	118	105	56	55	76	35	36
	122	117	105	55	56	76	35	36
Media	118,45	113,18	105,00	55,09	56,00	81,00	35,00	36,55
Desviación	5,91	6,51	0,00	0,29	0,60	3,79	0,00	0,50
C. V.	4,99 %	5,75 %	0,00 %	0,52 %	1,08 %	4,68 %	0,00 %	1,36 %
Intervalo	5 ms						Media	2,30 %

CUADRO 5.3: Resultados de la medida de los ultrasonidos salteados.

Por tanto, como todas estas funciones están encadenadas, cuando se desea leer los ultrasonidos, basta con ejecutar la función que realiza la media de todas las lecturas, `average_read_sensors()`, que devuelve el *array* guardado en la variable `sonar_readings`.

Otra función importante relacionada con el tratamiento de datos está en conocer si existe un obstáculo delante o no. Esto se puede realizar de forma sencilla comparando cada uno de los valores con un espacio de seguridad. Si cualquiera de ellos fuera menor, la función elevaría un error. Como se comentó al principio, aunque el artículo original establezca que lo más óptimo es calcular si en la siguiente iteración va a golpear o no en función de su velocidad, por los motivos allí explicados se establece una burbuja de valor constante en su lugar, pero que adaptada a la forma del robot. La distancia mínima que debe tener se puede definir como la que existe entre el sensor y el punto mínimo de medida donde tiene visibilidad más un umbral de seguridad extra (ver anexo C.3). Este criterio es el aplicado en los sensores delanteros sobretodo, que son aquellos que perciben el mayor avance con respecto a su zona de medida. Siguiendo este criterio, la función que realiza esta comprobación, `check_for_obstacles()`, se puede consultar en el fragmento de código 5.10.

---

```
1 // Compare whether any reading is below threshold
```

---

```

2 int check_for_obstacles(unsigned int *sonar_readings, const int
3   start, const int end) {
4
5   unsigned int bubble_boundary[8] = {20, 20, 30, 40, 40, 30, 20,
6     20}; // Centimeters
7
8   for(int i = start; i < end; i++) {
9     if(sonar_readings[i] <= bubble_boundary[i]) {
10       return 1;
11     }
12   }
13   return 0;
14 }
```

---

CÓDIGO 5.10: Comprobación de la existencia de obstáculos peligrosos

Una vez realizada esta comprobación, si hay un obstáculo peligroso, el vehículo intentará evitarlo calculando un ángulo de rebote, `alpha_r`, que designará una nueva dirección en la que la densidad de obstáculos es mínima. Esto se ha implementado en la función `compute_rebound_angle()`, disponible en el código 5.11.

---

```

1 // Compute rebound angle
2 float rebound_angle(unsigned int *sonar_readings) {
3   const float alpha_0 = 180/8;
4   float num = 0;
5   float den = 0;
6   for(int i = -4; i < 5; i++) {
7     if(i == 0){ // Zero is not taken.
8       i = 1;
9     }
10    float alpha_i = i * alpha_0;
11    num += alpha_i * sonar_readings[i];
12    den += sonar_readings[i];
13  }
14  float alpha_r = num/den;
15  return alpha_r;
16 }
```

---

CÓDIGO 5.11: Cálculo del ángulo de rebote

Siendo  $D_i$  la distancia dada por un sensor de índice  $i$ , dicha función realiza el cálculo sugerido en el artículo según la ecuación 5.2.

$$\alpha_r = \frac{\sum_{i=-4}^4 \alpha_i D_i}{\sum_{i=-4}^4 D_i} \quad (5.2)$$

Es importante recalcar que el índice cero no se coge ya que eso generaría un deslizamiento de los valores hacia uno de los lados, al no haber un sensor apuntando justamente hacia delante (ver anexo C.3). El ángulo obtenido es, entonces, el incremento sobre el rumbo actual que debe ejecutarse para no golpear con el obstáculo. Además, dado que el cálculo utiliza un índice que va desde -4 que es el sensor 0 del *array*, hasta el 4, se puede distinguir si se debe girar a derecha o a izquierda según el signo obtenido en el resultado de este cálculo.

En caso de ser negativo el robot deberá girar a la izquierda ya que el índice negativo indica que la derecha es el lugar con más obstáculos. La operación que permite al robot girar para reorientarse es `adjust_heading()` que utiliza el rumbo, el ángulo de rebote y el sentido de giro como parámetros de entrada (ver código 5.12).

---

```

1 // Rotate robot to the new heading
2 int adjust_heading(float heading, float alpha_r, int direction) {
3     if(direction == 1) {
4         // Calculate the new heading. Note: 0 for turning right, 1
5         // for left
6         float avoid_direction = calculate_heading(heading,
7             abs(alpha_r), 1);
8         Serial.print("avoid_direction = ");
9         Serial.println(avoid_direction);
10        do{
11            rotate_left(avoid_direction);
12            heading = Compass.GetHeadingDegrees();
13            Serial.print("Heading: ");
14            Serial.println(heading);
15        }while(heading >= avoid_direction + heading_threshold ||
16               heading <= avoid_direction - heading_threshold);

```

```

14     }
15     else{
16         // Calculate the new heading. Note: 0 for turning right, 1
17         // for left
18         float avoid_direction = calculate_heading(heading,
19             abs(alpha_r), 0);
20         Serial.print("avoid_direction = ");
21         Serial.println(avoid_direction);
22         do{
23             rotate_right(avoid_direction);
24             heading = Compass.GetHeadingDegrees();
25             Serial.print("Heading: ");
26             Serial.println(heading);
27         }while(heading >= avoid_direction + heading_threshold ||
28             heading <= avoid_direction - heading_threshold);
29     }
30 }
```

CÓDIGO 5.12: Rotación del robot al rumbo dado

Tras discernir el sentido de giro es necesario calcular la nueva dirección objetivo con respecto al mundo global, ya que `alpha_r` contiene un ángulo con respecto al mundo local. Después gira sobre sí mismo en el sentido dado hasta estar orientado dentro de la tolerancia establecida.

Una vez realizado el giro vuelve a comprobar si siguen habiendo obstáculos, en cuyo caso volvería a calcular un ángulo de rebote igual que anteriormente. En caso contrario comprobará si tras el nuevo cambio de dirección es capaz de ver el obstáculo (ver diagrama de bloques 5.2). Para realizar dicha comprobación basta con saber si existe o no un obstáculo en la dirección hacia la que se encuentra el punto. Para ello hay que discernir hacia qué lado del robot está este punto y ver qué sensores son capaces de verlo. Esto es lo que realiza la función `goal_visible()` (ver código 5.13).

---

```

1 // Check if goal is visible.
2 int goal_visible(unsigned int *sonar_readings){
3     float heading = Compass.GetHeadingDegrees();
4
5     float right_limit = calculate_heading(heading, 90, 0);
```

```

6     float left_limit = calculate_heading(heading, 90, 1);
7
8     // The goal is on the right side.
9     if(heading_command > heading && heading_command < right_limit){
10        Serial.println("Checking obstacles on the right");
11        // if(check_for_obstacles(sonar_readings, 0, 4)){
12        if(search_goal(sonar_readings, 0, 4)){
13            Serial.println("Goal was found on the right");
14            return 0;
15        }
16        else{
17            return 1; // Raise error if goal is not visible
18        }
19    }
20    else{ // The goal is on the left side.
21        Serial.println("Checking obstacles on the left");
22        // if(check_for_obstacles(sonar_readings, 4, 8)){
23        if(search_goal(sonar_readings, 4, 8)){
24            Serial.println("Goal was found on the left");
25            return 0;
26        }
27        else{
28            return 1; // Raise error if goal is not visible
29        }
30    }
31 }
```

CÓDIGO 5.13: Comprobación de la visibilidad del objetivo

Para conocer si hay que visualizar la izquierda o la derecha del robot, se establecen límites según su rumbo actual y el objetivo. Es importante recordar que esto se debe hacer pasando por la función `calculate_heading()` que permite adaptar los valores a los rangos correctos, ya que estos solo pueden estar entre 0 y 360 grados. Tras saber en qué lado se encuentra se evalúa la función `search_goal()` que es fundamentalmente parecida a `check_for_obstacles()` con la excepción de que los valores del *array* de comparación son más holgados para establecer una tolerancia con respecto a lo que se considera un obstáculo o un objetivo visible. Si no se encontrara el objetivo, esta

devuelve un error que es pasado a la función original.

En caso de que este objetivo no fuera encontrado, entonces el robot puede avanzar recto mientras va comprobando si existe algún obstáculo hasta que sea capaz de ver el objetivo. Una vez esto ocurra, entonces empieza a evaluar todo desde el principio.

Partiendo desde el principio, si suponemos que no encontrara un obstáculo en este punto inicial, entonces realiza un giro sobre sí mismo para ubicarse mirando directamente al punto objetivo. Esto solo ocurrirá cuando no esté bien orientado, caso en el que simplemente avanzaría directo al punto. Finalmente, a diferencia de la iteración 1, la tolerancia en cuanto al rumbo ha sido reducida a un error máximo de 5 grados.

### 5.2.2.2 *Movimientos*

Hasta ahora ha sido comentado el algoritmo implementado con cierto detalle, no obstante, otro punto crítico que es necesario tratar es el relativo a la acción propia del robot. Esto es, de qué manera reacciona el vehículo ante la decisión de girar o avanzar. El motivo que hace vital tratar este asunto está relacionado con la estabilidad y el control del sistema en su conjunto. Particularmente para mantener un rumbo constante, o llegar al correcto sin demasiadas oscilaciones.

Como se deduce del algoritmo explicado en la sección 5.2.2.1, para este robot se han planteado dos movimientos básicos. Es decir, el vehículo puede tener traslación pura en la que ambas ruedas se mueven a la misma velocidad; o rotación pura, en la que una rueda se mueve en sentido contrario a la otra pero a igual velocidad. Para ordenar estos movimientos se han definido tres funciones que ejecutan estas acciones (ver código 5.14).

---

```
1 // Rotate left
2 void rotate_left(float goal, double timeChange) {
3     Serial.println("Rotating left!");
4     float command = calculate_command(goal, timeChange);
5     motor.set(A, command, REVERSE);
6     motor.set(B, command, FORWARD);
7     motor_delay(10);
8 }
9
10 // Rotate right
```

```

11 void rotate_right(float goal, double timeChange) {
12     Serial.println("Rotating right!");
13     float command = calculate_command(goal, timeChange);
14     motor.set(A, command, FORWARD);
15     motor.set(B, command, REVERSE);
16     motor_delay(10);
17 }
18
19 // Move forward
20 void move_forward() {
21     Serial.println("Going forward!");
22     motor.set(A, 120, FORWARD); // Drift correction
23     motor.set(B, 130, FORWARD);
24     motor_delay(10);
25 }
```

CÓDIGO 5.14: Funciones de movimiento del robot

Por un lado, `move_forward()` realiza la traslación pura, mientras que, por otro, `rotate_left()` y `rotate_right()` corresponden a la rotación hacia la izquierda y derecha. Es importante notar que después de setear cada motor se introduce una espera de 10 ms en el código. Esto es necesario ya que, de lo contrario, no se vería movimiento alguno en los motores al ser la velocidad del micro mucho mayor con respecto al tiempo que necesita la rueda para empezar a girar. Aunque como se comentó en las observaciones de la sección 3.6, este valor seguramente pueda ser menor. En cualquier caso, cuando se introduce un *delay* en el código no se ejecuta ninguna orden, pero sin embargo el robot está en movimiento, por lo que esto se traduce en un retardo dentro del propio sistema.

Si se toma el robot como un sistema único se puede decir que es un lazo cerrado con dos actuadores que, en función de sus consignas individuales, cambian la orientación y posición con respecto al espacio. Como las ecuaciones matemáticas que determinan la posición siguiente del robot en un tiempo discreto ya fueron definidas en la sección 1.2.1, no resulta necesario comentar nada más acerca de ellas.

El *feedback* de este sistema, por tanto, viene dado por los distintos sensores que tiene dispuestos. En este proyecto, al dirigirse a un punto, el robot intenta mantener un rumbo constante que debe estar dentro de un rango que impone una tolerancia. De lo

contrario, gira sobre sí mismo. Entonces, es importante diseñar un sistema de control que permita, o bien mantenerse dentro del rango marcado, o llegar a él lo más rápido posible en caso de no ubicarse en su interior. Lo primero resulta complicado con la electrónica que diseñada ya que, al no tener *encoders* o un sistema que retroalimente la velocidad instantánea de las ruedas, es imposible hacer un sistema que ajuste el comando de cada una. Esto es, aunque hayan modelos matemáticos que pueden aproximar la velocidad de cada una en función de los radios y la consigna, no es posible determinar cuál es el valor en velocidad. Para comprender esta idea es necesario tener en cuenta que, aunque los motores sean de corriente continua y el voltaje aplicado sea directamente proporcional a la tensión aplicada, no es posible calcular la velocidad con un error aceptable. Esto se debe a que el voltaje que suministra la batería es bastante cambiante en función de la carga demandada, además de que no existe un partidor de tensión que monitorice la carga de la misma. Por este motivo, en el código 5.14 destaca que la función que mueve el robot recto tiene un comando diferente para cada motor. El objetivo de esta diferencia no es otro que una calibración sencilla para compensar la desviación que tiene uno frente a otro. Como es evidente, dado que esto es un valor fijo, no es una manera muy robusta de hacerlo al no compensar perturbaciones que hagan que una de ellas vaya a una velocidad diferente. Para esto haría falta ubicar un sensor en cada una, como se ha explicado a lo largo de este párrafo.

Sin embargo, la opción de diseñar un sistema de control que garantice que el robot gire lo más rápido posible hacia el rumbo sí es realizable con el diseño actual. Es más, resulta deseable implementarlo siempre que sea estable y relativamente robusto. Si recordamos que en rotación pura ambos motores reciben el mismo comando, y que el sensor que introduce la información es solamente uno, la brújula, podemos decir que el sistema pasa de ser SIMO a SISO. Por tanto, como a ambos se les atribuye el mismo comando, pero de signo cambiado, ahora sí es posible diseñar un controlador apropiado. Evidentemente esta suposición menoscopia que una de ellas gira a diferente velocidad, pese a que ambas llevan asignada la misma consigna. No obstante, este hecho resulta despreciable para este proceso ya que el efecto de esta desviación sobre la traslación, en comparación con la rotación, es mucho menor.

El controlador implementado, por tanto, es un proporcional-integral para los movimientos de rotación pura. La parte proporcional se usa principalmente para obtener mejor respuesta del sistema, y la parte integral para eliminar el error en el estacionario y compensar la inercia.

Como se puede ver en el código 5.14, cada vez que se invocan estas funciones, a su vez la función `calculate_command()` es llamada (ver código 5.15). Esta devuelve la consigna de los dos motores en forma de entero sin signo, y luego se setea cada uno con el sentido correcto.

La expresión general de este controlador en un tiempo continuo queda representado por la ecuación 5.3.

$$u = K_p \left( e + \frac{1}{T_i} \int_{t_1}^{t_2} e \cdot dt \right), \quad (5.3)$$

donde  $u$  es el comando de entrada al actuador,  $K_p$  la ganancia única,  $e$  el error y  $T_i$  el tiempo integral. Sin embargo, para aplicarlo a este proyecto es necesario digitalizar la expresión anterior utilizando incrementos con respecto a la muestra anterior. La ecuación 5.4 expresa dicha transformación.

$$\Delta u = K_p \left( \Delta e + \frac{1}{T_i} e \cdot \Delta t \right), \quad (5.4)$$

donde  $\Delta t$  es el tiempo de muestreo. Otra modificación que puede realizarse en la ecuación 5.4 es hacer que el error solamente esté en la parte integral. Por tanto, este queda sustituido en la parte proporcional por el incremento en la variable de proceso. De esta manera, se evita que haya un cambio brusco en el resultado del controlador, si la consigna cambiara repentinamente, por lo que el peso de un cambio en el *setpoint* solo afecta al término integral [12, pp. 52–78]. La ecuación 5.5 representa la expresión final del controlador con dicha modificación.

$$u(k) = K_p \left( \theta(k) - \theta(k-1) + \frac{1}{T_i} (e(k) \cdot \Delta t) \right) + u(k-1) \quad (5.5)$$

---

```

1 // PI controller for rotation of the robot
2 int calculate_command(float goal, double timeChange) {
3     float heading = Compass.GetHeadingDegrees();
4     float eps = abs(heading - goal);
5     float deltaPV = abs(heading - lastHeading);
6
7     // Kp and Ti manually adjusted
8     int Kp = 4;

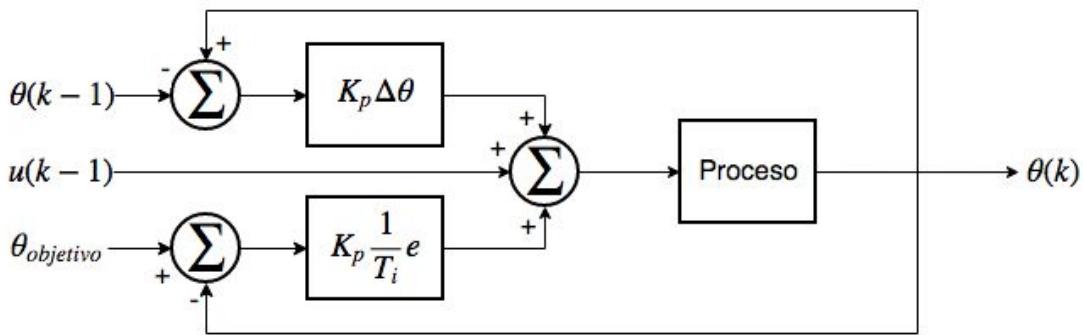
```

```

9     int Ti = 2;
10    float max_command = 130;
11
12    float command = (Kp * (deltaPV + 1/Ti * (eps * timeChange)) +
13                      lastCommand) * (max_command/180);
14
15    // Clamp to avoid values over max_command due to excessive
16    // error/gain
17    command = constrain(command, 60, max_command);
18
19    lastEps = eps;
20    lastCommand = command;
21    lastHeading = heading;
22
23    return command;
24 }
```

CÓDIGO 5.15: Controlador PI para rotación pura

Con el objetivo de facilitar la interpretación de estos comentarios, el diagrama de bloques dibujado en la figura 5.3 representa el controlador implementado en tiempo discreto.



En el código 5.15 se introducen como parámetros de entrada el objetivo a alcanzar, es decir, el *setpoint*, y el incremento de tiempo entre la muestra anterior y la actual. Esto se calcula al principio de cada ciclo en milisegundos con la función `lastInlinemillis()`, y se pasa a segundos para que todos los valores tengan las mismas

unidades. Posteriormente se calcula el rumbo actual y el error, así como la diferencia entre la variable de proceso anterior y actual. Después, el resultado del controlador es calculado con la ecuación 5.5, y el valor final es ajustado al rango necesario. En el caso de este robot, la consigna límite ha de ser 130, es decir, aproximadamente la mitad de la velocidad máxima que alcanza. Además, un comando inferior a 60 hace que el robot apenas se mueva, dado que el par que ejercen los motores con un valor más pequeño es menor que el necesario para generar movimiento. Finalmente, los valores de error, comando y rumbo de esta evaluación son almacenados en las variables globales definidas al principio del programa para poder usarlas en la siguiente iteración.

En cuanto a la calibración del controlador, esta ha sido realizada experimentalmente, probando con distintos valores hasta obtener una reacción que carezca de demasiadas oscilaciones con una respuesta rápida pero estable. Con este método se ha determinado que un valor de  $K_p = 4$  y  $T_i = 2$  son valores óptimos para este sistema.

### 5.3 Seguridad y protección

Durante esta segunda versión del proyecto se ha procurado mantener algunas medidas de seguridad y protección, no solo para las personas, sino también para los componentes electrónicos.

Por un lado, el sistema es eléctricamente seguro para las personas porque el máximo voltaje utilizado se mantiene siempre por debajo de 24 V. Además, no solo hay disponible un interruptor de emergencia por software, el *enable*, sino que también existe un interruptor general visible en la caja principal. Este corta la conexión con la batería, apagando todo el sistema instantáneamente.

Por otro, una medida de protección para la electrónica ha consistido en colocar los componentes dentro de cajas de protección. Esto evita que el polvo, la humedad o el agua no llegue al interior. Para conectar las cajas que están separadas, los agujeros para sacar los cables están ubicados en la parte inferior para minimizar la probabilidad de que les afecte las condiciones ambientales. Además, los sensores ultrasonidos poseen su propia carcasa personalizada que está atornillada en la zona superior del robot. La idea de esta ubicación es evitar los daños en los sensores durante su operación.

Adicionalmente, los cables han sido soldados con el pertinente aislante termorretractil para garantizar que no hayan falsos contactos entre ellos. Junto a esta protección, los

conectores utilizados permiten cambiar los módulos defectuosos con facilidad, haciendo también que la caja principal sea accesible.

## 5.4 Presupuesto

Al igual que en la primera versión es interesante ofrecer un presupuesto no exhaustivo del proyecto, tras los cambios de *hardware* introducidos. Una vez más, solo se tienen en cuenta los materiales y no la mano de obra empleada. Por tanto, las consideraciones comentadas en la sección 3.4 son mantenidas en esta parte del trabajo.

Para esta segunda iteración hay que contabilizar algunos extras como la cámara y la Raspberry Pi que añaden un coste de 56€, así como el módulo que adapta el voltaje para alimentarla y las cajas de protección. Además, en esta versión se ahorra el módulo WiFi, aunque este no tenga un coste significativo sobre el importe total. Finalmente, el presupuesto de esta versión asciende a 237€.

Concepto	Uds.	Precio unitario (€)	Importe (€)
Motor	2	12,00	24,00
Arduino MEGA 2560	1	15,00	15,00
Raspberry Pi 3 Model B	1	42,00	42,00
Cámara RPi 3	1	14,00	14,00
Bicicleta	1	10,00	10,00
Batería	1	40,00	40,00
Tabla	1	3,00	3,00
Ángulos de chapa	2	2,00	4,00
Step-down DC/DC	1	2,00	2,00
Ultrasonidos	8	1,00	8,00
Controlador motor	2	14,00	28,00
Brújula	1	2,00	2,00
GPS	1	5,00	5,00
Joystick	1	10,00	10,00
Varios	1	10,00	10,00
Cajas de protección	1	20,00	20,00
Total			237,00

CUADRO 5.4: Presupuesto iteración 2

## 5.5 Discusión de resultados

Durante esta versión se ha conseguido implementar la mayoría de mejoras planteadas en la sección 3.6. Por un lado, ahora existen dos niveles de trabajo en el que uno es maestro sobre el segundo, manejando los sistemas de comunicación, favoreciendo una manera más sencilla de hacer *debug*, y acceso remoto por SSH. Por otro, en el bajo nivel se han solucionado los problemas mayores de comunicación al eliminar el módulo WiFi y limitar su trabajo al algoritmo para evitar obstáculos y el cumplimiento del recorrido designado.

Otro punto importante a comentar es el cambio físico que ha sufrido el robot. En esta nueva versión se propone una distribución más uniforme que garantiza un centro de masas más próximo al centro geométrico del vehículo, mejorando así su estabilidad. Estos cambios también producen un conjunto más ligero, por lo que los motores necesitan menos par para generar movimiento, ahorrando también en el consumo eléctrico del mismo.

De igual manera, la estabilidad ha sido reforzada con la implementación de un controlador proporcional-integral que aporta cierta rapidez al proceso de orientarse. También ofrece un comportamiento más robusto en comparación con el todo o nada implementado en la primera versión.

### 5.5.1 Problemas modo automático

Como ha sido comentado anteriormente a lo largo del presente capítulo y el anterior, el modo automático ha estado basado en el artículo *The bubble rebound obstacle avoidance algorithm for mobile robots* [22]. En este se propone un algoritmo para que un robot móvil de tracción diferencial pueda evitar obstáculos. En función las lecturas de los sensores, este calcula un ángulo de rebote que deberá girar sobre sí mismo hacia el lado con menor densidad de obstáculos, siempre que no haya peligro de golpear con alguno.

Tras las pruebas el algoritmo ha demostrado ser relativamente fiable con el montaje realizado, en comparación con el ideado en la iteración 1. No obstante, por la naturaleza de los sensores ultrasonidos, en pruebas al aire libre se observa que, pese a realizar la media de varias lecturas de los ultrasonidos, existen demasiados falsos positivos que impiden un avance real y coherente. Esto es debido a la presencia de rachas de viento

que alteran las medidas de las ondas de presión que emiten estos sensores. Por tanto, siguiendo la recomendación del artículo original, una solución a este problema puede ser el uso de un telémetro láser para medir las distancias.

No obstante, las pruebas también demuestran que si se desactivan los sensores de distancia y se marca un camino libre de obstáculos, es posible realizar el recorrido sin mayor problema con el GPS y la brújula. Este aspecto es crítico debido a que una preocupación sobre la localización se encuentra en la incertidumbre que poseen estos sistemas debido a efectos externos. Una solución común para estos problemas es implementar un GPS diferencial en el que, conociendo el valor real de un punto cercano en el mapa, la deriva es calculada y aplicada a la lectura del robot para corregir dicho error.

Otra apreciación con respecto al funcionamiento en espacios reducidos es que los puntos marcados en el mapa no deben estar separados a más de 2 o 3 m. De lo contrario, el error en la medida del GPS del robot hace que desde su punto de vista el objetivo quede rodado, pudiendo chocar con algún obstáculo imprevisto. Una solución para marcar estos puntos con mayor precisión puede consistir en usar el modo manual para guardar rutas predeterminadas con los puntos del mapa.

Desde el punto de vista del control, la pérdida de carga de la batería afecta notoriamente a la capacidad del robot para girar. Esto significa que el controlador necesita mayor ganancia para compensar la falta de par en los motores. Por el momento esta es fija, y calibrada para ser lo más óptima posible con la batería cargada al máximo. Dado que el efecto producido no es de inmovilidad total, sino de respuesta más lenta, esta calibración puede ser aceptable. Sin embargo, una batería demasiado descargada no produce movimiento, aún cuando la consigna de los motores sea mayor de 60.

### 5.5.2 Problemas modo manual

En cuanto a este modo, el control local del *rover* no presenta demasiados problemas desde la versión 1. Sin embargo, tras otras pruebas se observa que el robot desliza cuando cambia la velocidad de los motores en subidas y bajadas. Esto es producido por distintos factores que manifiestan falta de agarre. Uno de ellos puede ser la falta de peso, o que las gomas están demasiado infladas provocando dicha carencia. Una solución que se puede plantear es programar el control manual de tal manera que no hayan cambios bruscos de consignas en los motores para hacer una navegación más suave.

Esto es realizable, por ejemplo, guardando el último valor asignado y comparandolo con el actual. En caso de que la diferencia sea mayor que una cantidad, entonces se aplica un límite a dicho incremento. Otra manera de solucionarlo es reducir el *delay* de los motores a valores inferiores de 10 ms. Con esta solución la frecuencia de muestreo aumenta, por lo que se minimizan los efectos producidos por un cambio brusco de consigna.

Es necesario comentar que, en comparación con la primera versión, en este modo no se ha implementado ningún control remoto. Por tanto, no es posible realizar ninguna comparación en este aspecto.

## 5.6 Conclusions

During the development of this project several problems have been faced regarding mobile robots. Most of them are related to the lack of quality measurements from the sensors that have a direct impact on localization and navigation flow.

1. The physical location of the sensors was proved to be key in order to obtain accurate measurements. Most importantly, the performance of ultrasonic range modules for outdoor environments to measure distances has been rather low for the presented rover. Instead, other methods for accurately locating obstacles might be desirable, such as LiDAR mapping and other technologies. However, the combination of a compass and a GPS sensor sufficiently satisfies the needs of this project in terms of location accuracy. This idea is only achieved if a safe distance from any metallic object around is kept to minimize the effects over the magnetic field being measured.
2. Filtering the readings obtained from the sensors is a recommended implementation for the global success of the robot. This can be achieved by calculating the average of several measurements, thus minimizing noise.
3. Trying to implement high-level communications with lower control structures, such as H-bridges or sensor readings, has been proved to be challenging to obtain good performance for remote manual control mode. Lagging signals represent the main issue that may be caused by the Arduino not being able to handle multiple threads. Instead, a hierachical structure is desirable for each control level.

4. Wiper motors were also proved to be a cheap and successful solution for this differential mobile robot by providing enough torque to move the whole structure. Additionally, the implementation of a PI controller is recommended to obtain a faster and more stable response compared to the bang-bang controller implemented during the first iteration.
5. In order to fix speed differences on each wheel that greatly affect the kinematic response of the vehicle, it is advisable to use encoders or similar sensors on the motors. The use of these is necessary to be able to create a closed-loop control system that keeps the correct orientation of the vehicle when moving forward.
6. The application of the bubble rebound algorithm performed well in most situations where the information provided from the sensors was accurate and precise. Even with the implementation differences from the original article the vehicle managed to avoid most of the obstacles.

# **Capítulo 6**

## **Mejoras futuras**

El objetivo de este capítulo es proponer y discutir algunas mejoras posibles para una tercera versión. Aunque la segunda ya realiza bastante parte del trabajo, esta puede enfocarse más hacia la comunicación externa y la interacción entre la Raspberry Pi y el entorno.

Por tanto, durante la versión 2 ya se aporta una base sólida de control y estabilidad a bajo nivel que puede prepararse como una interfaz a la que el alto nivel ordena las acciones, y el Arduino gestiona las señales necesarias para que eso ocurra. Es decir, el algoritmo ya no lo llevaría el Arduino, sino el ordenador, por lo que el primero queda como un mensajero de doble sentido entre los sensores y los actuadores con el ordenador.

### **6.1 Integración**

En este sentido, una de las mejoras más significativas que se puede aplicar es la integración de ROS dentro del sistema. De esta manera se puede aprovechar que el SO del ordenador ya contiene los paquetes instalados.

Con este punto de vista es posible reducir el peso de los sensores ultrasonidos dentro del algoritmo, para compensarlo con la instalación de un telémetro láser motorizado que permita medir las distancias con los objetos más cercanos [13]. Sin embargo, otro punto de vista que resulta interesante es la combinación de ROS con Kinect para obtener una nube de puntos de los obstáculos, al ser una cámara de profundidad. Lo positivo de esta última combinación es que el mismo sensor haría también la función de cámara para tomar las imágenes NDVI, ya que este es capaz de detectar parte del

infrarrojo cercano [15]. De esta manera es posible prescindir de la cámara propuesta en la versión 2 y el LIDAR, reduciendo el costo del proyecto. La combinación anterior es bastante utilizada por la comunidad, dado que ofrece mejores prestaciones que los ultrasonidos.

Otro añadido que podría ser interesante estudiar para el proyecto es incluir una placa fotovoltaica que permita recargar la batería, o al menos reforzar la alimentación mientras el vehículo está en movimiento.

## 6.2 Comunicación y conectividad

Una vez conseguido que el robot funcione adecuadamente en el lugar de trabajo es necesario permitir que trabaje remotamente. Para ello, las mejoras de comunicación y conectividad están relacionadas con establecer una red local en este sitio. Como se comentó en la sección 1.1.2, la manera de hacer esto puede ser con distintos puntos de acceso que ofrezcan cobertura suficiente.

En cuanto a la parte que se reduce al robot, una vez creada esta red habrá que reconfigurar la interfaz wlan0 para que conecte automáticamente con los AP, en vez de crear su propio punto de acceso local. Dentro de esta red local habrá que configurar los puertos necesarios para crear el túnel de SSH remoto desde donde se mandan los comandos.

Aunque al principio la idea era usar una página web a la que el usuario puede conectarse para enviar los puntos desde un mapa embebido, otra opción realizable es crear un bot de Telegram que corre en la Raspberry Pi para ver si existen mensajes nuevos. De esta manera se almacena un *buffer* en un servidor de IoT del que se obtienen las nuevas instrucciones con un *get*. La ventaja de integrarlo de esta manera es que se aprovechan funciones estandarizadas para enviar ubicaciones de tal manera que solamente hay que extraer las coordenadas de un mensaje estándar. Al final, la utilidad de esto reside en poder seleccionar de un mapa visible el punto geográfico deseado. En cualquier caso, ambos deben tener una opción configurada que realice las acciones que se deseen, como sacar una foto y mandarla a una dirección web.

Paralelamente, el modo manual necesita un método para enviar y recibir los datos. De esta manera tendría que haber una comunicación UDP que envía el *string* de datos del

*joystick* a la dirección del robot. Este último aspecto sería necesario investigar cómo implementarlo en un modelo real por internet.



## Apéndice A

### Programas

En este anexo se recogen todos los programas completos que componen las distintas versiones.

#### A.1 Código Arduino para la iteración 1

Esta sección imprime el programa principal para Arduino de la iteración 1. Consiste en un programa que incluye un modo automático que contiene el algoritmo para evitar obstáculos, y uno manual en el que se inicia un punto de acceso para leer los valores enviados por UDP desde el *joystick*. La explicación precisa de esta implementación queda recogida en los capítulos 2 y 3.

---

```
1 /*  
2 GuimarFarmExplorerServer.ino is the main control sketch of GFE  
3 iteration 1.  
4  
5 Modules implemented:  
6 - Motors (IMS-1)  
7 - HC-SR04 (ultrasonic sensors)  
8 - HMC5883L (magnetometer - GY-273)  
9 - WiFi connection  
10 Author: Javier Macias  
11 */  
12  
13 //---Importing libraries---//
```

```
14 #include <MOTOR.h> // Controlling motors with IMS-1 driver modules
15 #include <Arduino.h> // Libraries for compass HMC5883L
16 #include <Wire.h> // I2C library
17 #include <HMC5883L_Simple.h> // Compass library
18 #include <stdio.h> // I/O management i.e. sscanf()
19 #include <stdlib.h>
20 #include <WiFiEsp.h> // WiFi ESP8266 libraries
21 #include <WiFiEspUdp.h> // UDP protocol for ESP8266
22
23 // Defining WiFi parameters
24 char ssid[] = "ESP8266 Server"; // your network SSID (name)
25 char pass[] = "esp8266server"; // your network password
26 int status = WL_IDLE_STATUS; // the Wifi radio's status
27
28 unsigned int localPort = 10002; // local port to listen on
29
30 char packetBuffer[255]; // buffer to hold incoming packet
31
32 WiFiEspUDP Udp; // Create an UDP instance
33
34 // Create a compass
35 HMC5883L_Simple Compass;
36
37 // Defining joystick inputs
38 const int x_axis = 1; //Blue
39 const int y_axis = 2; //Green
40 const int z_axis = 4; //Yellow
41
42 const int button1 = 5; //Brown
43 const int button2 = 6; //White
44
45 // Defining HC-SR04 inputs
46 const int EchoPin_front_left = 30;
47 const int TriggerPin_front_left = 24;
48 const int EchoPin_front_right = 32;
49 const int TriggerPin_front_right = 26;
50 const int EchoPin_left_front = 36;
```

```
51 const int TriggerPin_left_front = 37;
52 const int EchoPin_right_front = 38;
53 const int TriggerPin_right_front = 39;
54
55 // Joystick-related variables
56 int x_val = 0;
57 int y_val = 0;
58 int z_val = 0;
59 int button1_state = 0;
60 int button2_state = 0;
61
62 const int centre = 1024/2;
63
64 const int AUTOMATIC = 0; // 1 = automatic, 0 = manual.
65 const int threshold = 60; // Centimeters of safe zone
66
67 float heading_command = 330;
68
69 void setup(void) {
70     Serial.begin(115200);
71     Serial.println("GuimarFarmExplorerServer.ino");
72
73     if(!AUTOMATIC) {
74         // Initializing WiFi AP
75         Serial2.begin(9600); // initialize serial for ESP module
76         WiFi.init(&Serial2); // initialize ESP module
77
78         // check for the presence of the shield
79         if (WiFi.status() == WL_NO_SHIELD) {
80             Serial.println("WiFi shield not present");
81             while (true); // don't continue
82         }
83
84         Serial.print("Attempting to start AP ");
85         Serial.println(ssid);
86
87         // start access point
```

```
88     status = WiFi.beginAP(ssid, 10, pass, ENC_TYPE_WPA2_PSK);
89
90     Serial.println("Access point started");
91     printWifiStatus();
92
93     Serial.println("\nStarting connection to server...");
94     // if you get a connection, report back via serial:
95     Udp.begin(localPort);
96
97     Serial.print("Listening on port ");
98     Serial.println(localPort);
99 }
100
101 // Defining joystick pinModes
102 pinMode(button1, INPUT);
103 pinMode(button2, INPUT);
104
105 // Defining HC-SR04 pinModes
106 pinMode(TriggerPin_front_left, OUTPUT);
107 pinMode(EchoPin_front_left, INPUT);
108 pinMode(TriggerPin_front_right, OUTPUT);
109 pinMode(EchoPin_front_right, INPUT);
110 pinMode(TriggerPin_left_front, OUTPUT);
111 pinMode(EchoPin_left_front, INPUT);
112 pinMode(TriggerPin_right_front, OUTPUT);
113 pinMode(EchoPin_right_front, INPUT);
114
115 // Initializing motors drivers
116 motor.begin();
117 /*
118 NOTE:
119 motor.begin() will change the prescaller of the timer0,
120 so the arduino function delay(), millis() and micros() are
121 8 times slow than it should be.
122 Please use motor_delay(), motor_millis(), motor_micros()
123 instead them.
124 */
125
```

```
124
125 // Initializing compass
126 Wire.begin();
127 Compass.SetDeclination(-5, 1, 'W'); //La Laguna
128 Compass.SetSamplingMode(COMPASS_SINGLE);
129 Compass.setScale(COMPASS_SCALE_130);
130 Compass.setOrientation(COMPASS_HORIZONTAL_X_NORTH);
131 }
132
133
134 void loop(void) {
135 //Entering manual or automatic mode
136 if(!AUTOMATIC){
137     // Motores parados por seguridad
138     x_val = 512;
139     y_val = 512;
140     z_val = 512;
141     button1_state = 1;
142     button2_state = 1;
143
144     // if there's data available, read a packet
145     int packetSize = Udp.parsePacket();
146     if (packetSize){
147         Serial.print("Received packet of size ");
148         Serial.println(packetSize);
149         Serial.print("From ");
150         IPAddress remoteIp = Udp.remoteIP();
151         Serial.print(remoteIp);
152         Serial.print(", port ");
153         Serial.println(Udp.remotePort());
154
155         // read the packet into packetBuffer
156         int len = Udp.read(packetBuffer, 255);
157         if (len > 0){
158             packetBuffer[len] = 0;
159         }
160         Serial.println("Contents:");
```

```
161     Serial.println(packetBuffer);
162
163     // Message to decode i.e. 1023 1023 1023 1 1
164     sscanf(packetBuffer, " %d %d %d %d %d ", &x_val,
165             &y_val, &z_val, &button1_state, &button2_state);
166
167     Serial.print("x_val = ");
168     Serial.println(x_val);
169     Serial.print("y_val = ");
170     Serial.println(y_val);
171     Serial.print("z_val = ");
172     Serial.println(z_val);
173     Serial.print("button1_state = ");
174     Serial.println(button1_state);
175     Serial.print("button2_state = ");
176     Serial.println(button2_state);
177
178     joystick_control(x_val, y_val, z_val, button1_state,
179                       button2_state);
180 }
181
182 else{ //Automatic mode
183     float heading = Compass.GetHeadingDegrees();
184     Serial.print("Heading: ");
185     Serial.println(heading);
186
187     int cm_front_left =
188         calculate_distance(TriggerPin_front_left,
189                             EchoPin_front_left);
190     int cm_front_right =
191         calculate_distance(TriggerPin_front_right,
192                            EchoPin_front_right);
193     int cm_left_front =
194         calculate_distance(TriggerPin_left_front,
195                            EchoPin_left_front);
```

```
188     int cm_right_front =
189         calculate_distance(TriggerPin_right_front,
190             EchoPin_right_front);
190     Serial.print(cm_left_front);
191     Serial.print(" ");
192     Serial.print(cm_front_left);
193     Serial.print(" ");
194     Serial.print(cm_front_right);
195     Serial.print(" ");
196     Serial.println(cm_right_front);
197
197     if(cm_front_left < threshold || cm_front_right < threshold){
198         //Try to avoid the object in front
199         Serial.println("Obstacle detected!");
200
200         if((cm_front_left < 35 || cm_front_right < 35)){ //Go
201             reverse until it is safe to do a 90 degrees turn
202             do{
203                 motor.set(A, 110, REVERSE);
204                 motor.set(B, 130, REVERSE);
205                 motor_delay(10);
206                 cm_front_left =
207                     calculate_distance(TriggerPin_front_left,
208                         EchoPin_front_left);
208                 cm_front_right =
209                     calculate_distance(TriggerPin_front_right,
210                         EchoPin_front_right);
210             }while(cm_front_left < 35 || cm_front_right < 35);
211         }
212
213         float obstacle_angle =
214             calculate_obstacle_angle(cm_front_left,
215                 cm_front_right);
215         Serial.print("Obstacle is @ ");
216         Serial.println(obstacle_angle);
217
```

```
214     if(cm_front_left < cm_front_right){ //Left is closer,
215         turn right the degrees necessary to be parallel to
216         object
217         Serial.println("Turning right to avoid contact!");
218         float avoid_direction = calculate_heading(heading, 90
219             - obstacle_angle, 0); // 0 for turning right, 1 for
220             left
221         Serial.print("Heading direction to avoid contact: ");
222         Serial.println(avoid_direction);
223         do{
224             motor.set(A, 130, FORWARD);
225             motor.set(B, 130, REVERSE);
226             motor_delay(10);
227             heading = Compass.GetHeadingDegrees();
228             Serial.print("Heading: ");
229             Serial.println(heading);
230             if(heading > avoid_direction && heading -
231                 avoid_direction <= obstacle_angle){
232                 break;
233             }
234         }while(true);
235
236         cm_front_left =
237             calculate_distance(TriggerPin_front_left,
238             EchoPin_front_left);
239         cm_front_right =
240             calculate_distance(TriggerPin_front_right,
241             EchoPin_front_right);
242         cm_left_front =
243             calculate_distance(TriggerPin_left_front,
244             EchoPin_left_front);
245         cm_right_front =
246             calculate_distance(TriggerPin_right_front,
247             EchoPin_right_front);
248
249         // Go forward for awhile. Otherwise would turn again
250         // against obstacle.
```

```
237     if(cm_front_left > threshold || cm_front_right >
238         threshold) {
239         do{
240             Serial.println("Going forward!");
241             motor.set(A, 110, FORWARD); //correcting drift
242             motor.set(B, 130, FORWARD);
243             motor_delay(10);
244             cm_front_left =
245                 calculate_distance(TriggerPin_front_left,
246                     EchoPin_front_left);
247             cm_front_right =
248                 calculate_distance(TriggerPin_front_right,
249                     EchoPin_front_right);
250             cm_left_front =
251                 calculate_distance(TriggerPin_left_front,
252                     EchoPin_left_front);
253             cm_right_front =
254                 calculate_distance(TriggerPin_right_front,
255                     EchoPin_right_front);
256
257             //Stop going forward and turn if getting closer
258             //to any object
259             if(cm_right_front < 20 || cm_front_right <
260                 threshold || cm_front_left < threshold) {
261                 break;
262             }
263             }while(cm_left_front < threshold);
264         }
265     }
266     else{ //Right is closer, turn left the degrees necessary
267         to be parallel to object
268         Serial.println("Turning left to avoid contact!");
269         float avoid_direction = calculate_heading(heading, 90
270             - obstacle_angle, 1); // 0 for turning right, 1 for
271             left
272         Serial.print("Heading direction to avoid contact: ");
273         Serial.println(avoid_direction);
```

```
260     do {
261         motor.set(A, 130, REVERSE);
262         motor.set(B, 130, FORWARD);
263         motor_delay(10);
264         heading = Compass.GetHeadingDegrees();
265         Serial.print("Heading: ");
266         Serial.println(heading);
267         if(heading < avoid_direction && heading -
268             avoid_direction >= -obstacle_angle) {
269             break;
270         }
271     }while(true);
272
273     cm_front_left =
274         calculate_distance(TriggerPin_front_left,
275             EchoPin_front_left);
276     cm_front_right =
277         calculate_distance(TriggerPin_front_right,
278             EchoPin_front_right);
279     cm_left_front =
280         calculate_distance(TriggerPin_left_front,
281             EchoPin_left_front);
282     cm_right_front =
283         calculate_distance(TriggerPin_right_front,
284             EchoPin_right_front);
285
286     // Go forward for awhile. Otherwise would turn again
287     // against obstacle.
288     if(cm_front_left > threshold || cm_front_right >
289         threshold) {
290         do{
291             Serial.println("Going forward!");
292             motor.set(A, 110, FORWARD); //correcting drift
293             motor.set(B, 130, FORWARD);
294             motor_delay(10);
```

```
284         cm_front_left =
285             calculate_distance(TriggerPin_front_left,
286             EchoPin_front_left);
287         cm_front_right =
288             calculate_distance(TriggerPin_front_right,
289             EchoPin_front_right);
290         cm_left_front =
291             calculate_distance(TriggerPin_left_front,
292             EchoPin_left_front);
293         cm_right_front =
294             calculate_distance(TriggerPin_right_front,
295             EchoPin_right_front);
296
297         //Stop going forward and turn if getting closer
298         //to any object
299         if(cm_left_front < 20 || cm_front_right <
300             threshold || cm_front_left < threshold){
301             break;
302         }
303         }while(cm_right_front < threshold);
304     }
305
306     else{
307         navigate_open_space(heading, heading_command);
308     }
309 }
```

```
302
303 // Used to calculate the angle at which the obstacle is
304 float calculate_obstacle_angle(float cm_front_left, float
305     cm_front_right) {
306     float diff = abs(cm_front_left - cm_front_right);
307     float alpha = atan2(diff, 49); //49 cm between both sensors
308     float alpha_degrees = alpha * 180 / 3.14159265358979323846;
309     return alpha_degrees;
310 }
```

```
310
311 // Used to calculate proper heading values
312 float calculate_heading(float heading, float obstacle_angle, int
313   direction){
314   if(direction == 0){
315     if(heading + obstacle_angle > 360){
316       return heading + obstacle_angle - 360;
317     }
318     else{
319       return heading + obstacle_angle;
320     }
321   }
322   else{
323     if(heading - obstacle_angle < 0){
324       return heading - obstacle_angle + 360;
325     }
326     else{
327       return heading - obstacle_angle;
328     }
329 }
330
331 // Autonomous logic for compass when there is open space and no
332 obstacles
333 void navigate_open_space(float heading, float heading_command){
334   if(heading >= heading_command + 10 || heading <=
335     heading_command - 10){ // Change direction
336     if(((heading - heading_command < 0 && abs(heading -
337       heading_command) < 180) || (heading - heading_command > 0
338       && abs(heading - heading_command) > 180))){
339       Serial.println("Turning right to heading_command!");
340       motor.set(A, 130, FORWARD);
341       motor.set(B, 130, REVERSE);
342       motor_delay(10);
343     }
344   }
345   else{
346     Serial.println("Turning left to heading_command!");
347   }
348 }
```

```
342         motor.set(A, 130, REVERSE);  
343         motor.set(B, 130, FORWARD);  
344         motor_delay(10);  
345     }  
346 }  
347 else{ // Continue forward  
348     Serial.println("Going forward! @ open space");  
349     motor.set(A, 110, FORWARD); //correcting drift  
350     motor.set(B, 130, FORWARD);  
351     motor_delay(10);  
352 }  
353 }  
354  
355 //Calculating distance in cm using HC-SR04 module (unfiltered)  
356 int calculate_distance(int TriggerPin, int EchoPin){  
357     long duration, distanceCm;  
358  
359     digitalWrite(TriggerPin, LOW); //Setting to LOW for 4us for  
            cleaner reading  
360     delayMicroseconds(4);  
361     digitalWrite(TriggerPin, HIGH); //Triggering for 10us  
362     delayMicroseconds(10);  
363     digitalWrite(TriggerPin, LOW);  
364  
365     duration = pulseIn(EchoPin, HIGH); //Measuring the time between  
            pulses in microseconds  
366  
367     distanceCm = duration * 10 / 292 / 2; //Converting to cm  
368  
369     return distanceCm;  
370 }  
371  
372 void joystick_control(int x_val, int y_val, int z_val, int  
        button1_state, int button2_state){  
373     x_val = 1024 - x_val;  
374     Serial.print("x_val = ");  
375     Serial.println(x_val);
```

```
376     y_val = 1024 - y_val;  
377     Serial.print("y_val = ");  
378     Serial.println(y_val);  
379  
380     int x = x_val - centre;  
381     Serial.print("x = ");  
382     Serial.println(x);  
383     int y = y_val - centre;  
384     Serial.print("y = ");  
385     Serial.println(y);  
386  
387     // Computing speed and direction in function of joystick  
     position  
388     // Convert to polar coordinates  
389     float r = hypot(x, y);  
390     float t = atan2(y, x);  
391  
392     // Rotate by 45 degrees  
393     t = t - 3.14159265358979323846 / 4;  
394  
395     // Back to cartesian  
396     float left = r * cos(t);  
397     float right = r * sin(t);  
398  
399     // Rescale the new coords  
400     left = left * sqrt(2);  
401     right = right * sqrt(2);  
402  
403     // Clamp to -245/+245 to avoid over power  
404     left = fmax(-245, fmin(left, 245));  
405     right = fmax(-245, fmin(right, 245));  
406     Serial.print(left);  
407     Serial.print(" ");  
408     Serial.println(right);  
409  
410     float left_abs = abs(left);  
411     float right_abs = abs(right);
```

```
412     Serial.print(left_abs);
413     Serial.print(" ");
414     Serial.println(right_abs);
415
416     // Controlling motors depending on joystick position
417     if(button1_state == 0 || x <= 100 && x >= -100 && y <= 100 && y
418         >= -100){ //Deadzone in centre for full stop
419         Serial.println("Stop!");
420         motor.set(A, 0, FORWARD);
421         motor.set(B, 0, FORWARD);
422         motor_delay(150);
423     }
424
425     else if(x < -100 && y <= 100 && y >= -100){ //Deadzone for
426         stability when joystick is right
427         motor.set(A, left_abs, REVERSE);
428         motor.set(B, right_abs, FORWARD);
429     }
430
431     else if(x > 100 && y <= 100 && y >= -100){ //Deadzone for
432         stability when joystick is left
433         motor.set(A, left_abs, FORWARD);
434         motor.set(B, right_abs, REVERSE);
435     }
436
437     else{
438         if(y >= 0){
439             if(x < 0 && x >= -y || x > 0 && x <= y){ //Upper second
440                 and third quarter
441                 motor.set(A, left_abs, FORWARD);
442                 motor.set(B, right_abs, FORWARD);
443                 motor_delay(20);
444             }
445             else if(x < 0 && x < -y){ //Lower second quarter
446                 motor.set(A, left_abs, REVERSE);
447                 motor.set(B, right_abs, FORWARD);
448                 motor_delay(20);
449             }
450         }
451     }
452 }
```

```
445     }
446     else{ //Lower first quarter
447         motor.set(A, left_abs, FORWARD);
448         motor.set(B, right_abs, REVERSE);
449         motor_delay(20);
450     }
451 }
452 else{
453     // Serial.println("y < 0");
454     if(x < 0 && x >= y || x > 0 && x <= -y){ //Lower third
455         and fourth quarter
456         motor.set(A, right_abs, REVERSE);
457         motor.set(B, left_abs, REVERSE);
458         motor_delay(20);
459     }
460     else if(x < 0 && x <= y){ //Upper third quarter
461         motor.set(A, right_abs, FORWARD);
462         motor.set(B, left_abs, REVERSE);
463         motor_delay(20);
464     }
465     else{ //Upper fourth quarter
466         motor.set(A, right_abs, REVERSE);
467         motor.set(B, left_abs, FORWARD);
468         motor_delay(20);
469     }
470 }
471 }
472
473 // Used to calculate the average of signals received from sensors
474 int calculate_average(int axis){
475     int value;
476     int num = 10;
477     for(int i = 0; i < num; i++){
478         value += analogRead(axis);
479     }
480     int average = value / 10;
```

```
481
482     return average;
483 }
484
485 void printWifiStatus() {
486     // print your WiFi shield's IP address
487     IPAddress ip = WiFi.localIP();
488     Serial.print("IP Address: ");
489     Serial.println(ip);
490
491     // print where to go in the browser
492     Serial.println();
493     Serial.print("SSID: ");
494     Serial.print(ssid);
495     Serial.print("\nIP: ");
496     Serial.print(ip);
497     Serial.println();
498 }
```

---

CÓDIGO A.1: Programa principal de la iteración 1



## A.2 Cliente UDP

Esta sección propone el código Arduino que permite leer, codificar y enviar los valores del *joystick* por UDP al robot.

---

```
1 /*  
2  UDPClient based on UdpSendRecieve example.  
3  
4  This sketch connects to an AP server and encodes a string.  
5  The string is then sent over UDP to the AP server over the local  
6  port specified  
7  
8  Credits to the original author of the examples.  
9  Modifications and further programming by Javier Macias  
9 */  
10  
11  
12 #include <WiFiEsp.h>  
13 #include <WiFiEspUdp.h>  
14  
15 // Emulate Serial1 on pins 6/7 if not present  
16 #ifndef HAVE_HWSERIAL1  
17 #include "SoftwareSerial.h"  
18 SoftwareSerial Serial1(8, 9); // RX, TX  
19 #endif  
20  
21 char ssid[] = "ESP8266 Server"; // your network SSID (name)  
22 char pass[] = "esp8266server"; // your network password  
23  
24 int status = WL_IDLE_STATUS; // the Wifi radio's status  
25  
26 // char server[] = "192.168.1.46"; // Server host  
27 char server[] = "192.168.4.1"; // Server host  
28 unsigned int localPort = 10002; // local port to listen on  
29  
30 char packetBuffer[255]; // Buffer to hold incoming packet  
31 char ReplyBuffer[20]; // Buffer to send string
```

```
32
33 // Defining joystick inputs
34 int x_axis = 1; //Blue
35 int y_axis = 2; //Green
36 int z_axis = 4; //Yellow
37
38 const int button1 = 5; //Brown
39 const int button2 = 6; //White
40
41 int x_val = 0;
42 int y_val = 0;
43 int z_val = 0;
44 int button1_state = 0;
45 int button2_state = 0;
46
47 String ReplyString;
48
49 WiFiEspUDP Udp; //Creating an instance
50
51 void setup(){
52     Serial.begin(115200); // initialize serial for debugging
53     Serial1.begin(9600); // initialize serial for ESP module
54     Serial.println("UDPClient.ino");
55     WiFi.init(&Serial1); // initialize ESP module
56
57     // check for the presence of the shield:
58     if(WiFi.status() == WL_NO_SHIELD) {
59         Serial.println("WiFi shield not present");
60         // don't continue:
61         while(true);
62     }
63
64     // attempt to connect to WiFi network
65     while(status != WL_CONNECTED) {
66         Serial.print("Attempting to connect to WPA SSID: ");
67         Serial.println(ssid);
68         // Connect to WPA/WPA2 network
```

```
69     status = WiFi.begin(ssid, pass);
70 }
71
72 Serial.println("Connected to wifi");
73 printWifiStatus();
74
75 Serial.println("\nStarting connection to server...");
76 // if you get a connection, report back via serial:
77 Udp.begin(localPort);
78
79 Serial.print("Listening on port ");
80 Serial.println(localPort);
81
82 pinMode(button1, INPUT);
83 pinMode(button2, INPUT);
84 }
85
86 void loop(){
87     //Read and store raw values
88     x_val = analogRead(x_axis);
89     y_val = analogRead(y_axis);
90     z_val = analogRead(z_axis);
91     button1_state = digitalRead(button1);
92     button2_state = digitalRead(button2);
93
94     //Building the string to be sent
95     ReplyString = String(x_val) + " " + String(y_val)+ " " +
96                 String(z_val) + " " + String(button1_state) + " " +
97                 String(button2_state) + " ";
98     ReplyString.toCharArray(ReplyBuffer, 19); //max. 18 elements +
99                 1 needed
100 }
101
102 void printWifiStatus(){
```

```
103 // print the SSID of the network you're attached to:  
104 Serial.print("SSID: ");  
105 Serial.println(WiFi.SSID());  
106  
107 // print your WiFi shield's IP address:  
108 IPAddress ip = WiFi.localIP();  
109 Serial.print("IP Address: ");  
110 Serial.println(ip);  
111  
112 // print the received signal strength:  
113 long rssi = WiFi.RSSI();  
114 Serial.print("signal strength (RSSI):");  
115 Serial.print(rssi);  
116 Serial.println(" dBm");  
117 }  
118  
119 // send an UDP package  
120 void sendPacket(char *server, char *ReplyBuffer) {  
121 Serial.print("Sending ");  
122 Serial.print(ReplyBuffer);  
123 Serial.print(" to ");  
124 Serial.print(server);  
125 Serial.print(" on port ");  
126 Serial.println(10002);  
127 Udp.beginPacket(server, 10002);  
128 Udp.write(ReplyBuffer);  
129 Udp.endPacket();  
130 }
```

---

CÓDIGO A.2: Programa principal para el control remoto con *joystick*

### A.3 Control de motores con el *joystick*

En esta sección se presenta el código necesario para controlar el robot con el *joystick* localmente. En él se muestra el proceso entero de transformación de las coordenadas y asignación de los comandos a los motores.

```
1 // Import libraries
2 #include <MOTOR.h>
3
4 // Defining joystick inputs
5 int x_axis = 0; //Blue
6 int y_axis = 1; //Green
7 int z_axis = 2; //Yellow
8
9 const int button1 = 7; //White
10 const int button2 = 6; //Brown
11
12 int x_val = 0;
13 int y_val = 0;
14 int z_val = 0;
15 int button1_state = 0;
16 int button2_state = 0;
17
18 const int centre = 1024/2;
19
20 int motors_wait = 10; //Wait for motors to move
21
22 void setup(){
23     Serial.begin(115200);
24     pinMode(button1, INPUT);
25     pinMode(button2, INPUT);
26
27     // Motor drive initialize
28     motor.begin();
29
30     Serial.println("joystick_motors_7.ino");
31 }
```

```
32
33
34 void loop() {
35     //Read, store and correct raw values
36     x_val = calculate_average(x_axis);
37     Serial.print("x_val = ");
38     Serial.println(x_val);
39     y_val = calculate_average(y_axis);
40     Serial.print("y_val = ");
41     Serial.println(y_val);
42     //For some reason z_axis is not working.
43     z_val = calculate_average(z_axis);
44     Serial.print("z_val = ");
45     Serial.println(z_val);
46     button1_state = digitalRead(button1);
47     button2_state = digitalRead(button2);
48
49     x_val = 1024 - x_val;
50     Serial.print("x_val = ");
51     Serial.println(x_val);
52     y_val = 1024 - y_val;
53     Serial.print("y_val = ");
54     Serial.println(y_val);
55
56     int x = x_val - centre;
57     Serial.print("x = ");
58     Serial.println(x);
59     int y = y_val - centre;
60     Serial.print("y = ");
61     Serial.println(y);
62
63     // Computing speed and direction in function of joystick
64     // position
65     // Convert to polar coordinates
66     float r = hypot(x, y);
67     float t = atan2(y, x);
```

```
68 // Rotate by 45 degrees
69 t = t - 3.14159265358979323846 / 4;
70
71 // Back to cartesian
72 float left = r * cos(t);
73 float right = r * sin(t);
74
75 // Rescale the new coords
76 left = left * sqrt(2);
77 right = right * sqrt(2);
78
79 // Clamp to -245/+245 to avoid over power
80 left = fmax(-245, fmin(left, 245));
81 right = fmax(-245, fmin(right, 245));
82 Serial.print("left: ");
83 Serial.print(left);
84 Serial.print(" right: ");
85 Serial.println(right);
86
87 float left_abs = abs(left);
88 float right_abs = abs(right);
89 Serial.print(left_abs);
90 Serial.print(" ");
91 Serial.println(right_abs);
92
93 // Controlling motors depending on joystick position
94 if(button2_state == 0 || (x <= 100 && x >= -100 && y <= 100 &&
95 y >= -100)){
96     Serial.println("Stop!");
97     motor.set(A, 0, FORWARD);
98     motor.set(B, 0, FORWARD);
99     motor_delay(motors_wait);
100 }
101 else{
102     if(y > 0){
103         if(left > 0 && right > 0){
104             motor.set(A, left, FORWARD);
```

```
104         motor.set(B, right, FORWARD);
105         motor_delay(motors_wait);
106     }
107     else if(left > 0 && right < 0){
108         motor.set(A, left, FORWARD);
109         motor.set(B, right_abs, REVERSE);
110         motor_delay(motors_wait);
111     }
112     else if(left < 0 && right > 0){
113         motor.set(A, left_abs, REVERSE);
114         motor.set(B, right, FORWARD);
115         motor_delay(motors_wait);
116     }
117 }
118 else{
119     if(left < 0 && right < 0){
120         motor.set(A, right_abs, REVERSE);
121         motor.set(B, left_abs, REVERSE);
122         motor_delay(motors_wait);
123     }
124     else if(left > 0 && right < 0){
125         motor.set(A, right_abs, REVERSE);
126         motor.set(B, left, FORWARD);
127         motor_delay(motors_wait);
128     }
129     else{
130         motor.set(A, right, FORWARD);
131         motor.set(B, left_abs, REVERSE);
132         motor_delay(motors_wait);
133     }
134 }
135 }
136 }
137
138 // Used to calculate the average of signals received from sensors
139 int calculate_average(int axis){
140     int value = 0;
```

```
141     int num = 10;  
142     for(int i = 0; i < num; i++) {  
143         value += analogRead(axis);  
144     }  
145     int average = value/num;  
146  
147     return average;  
148 }
```

---

CÓDIGO A.3: Programa para el control local de motores



## A.4 Obtención de imágenes NDVI

La presente sección muestra el código en Python que permite obtener imágenes NDVI usando la librería OpenCV y la cámara de la Raspberry Pi.

---

```
1 import time
2 import numpy as np
3
4 import cv2
5 import picamera
6 import picamera.array
7
8 def contrast_stretch(im):
9     """
10     Performs a simple contrast stretch of the given image, from
11     5-95%.
12     """
13     in_min = np.percentile(im, 5)
14     in_max = np.percentile(im, 95)
15
16     out_min = 0.0
17     out_max = 255.0
18
19     out = im - in_min
20     out *= ((out_min - out_max) / (in_min - in_max))
21     out += in_min
22
23
24
25 def run():
26     with picamera.PiCamera() as camera:
27         # Set the camera resolution
28         #x = 1600
29         #camera.resolution = (int(1.33 * x), x)
30         camera.resolution = (2592, 1944)
31         # Various optional camera settings below:
```

```
32     # camera framerate = 5
33     # camera.awb_mode = 'off'
34     # camera.awb_gains = (0.5, 0.5)
35
36     # Need to sleep to give the camera time to get set up
37     # properly
38     time.sleep(1)
39
40     with picamera.array.PiRGBArray(camera) as stream:
41         # Grab data from the camera, in colour format
42         # NOTE: This comes in BGR rather than RGB, which is
43         # important
44         # for later!
45
46         # Current time for filename 2012_05_15-15_50_45
47         timestr = time.strftime("%Y_%m_%d-%H_%M_%S")
48         camera.capture(stream, format='bgr')
49         image = stream.array
50
51         # Get the individual colour components of the image
52         b, g, r = cv2.split(image)
53
54         cv2.imwrite('/home/pi/ndvi_b.jpg', b)
55         cv2.imwrite('/home/pi/ndvi_g.jpg', g)
56         cv2.imwrite('/home/pi/ndvi_r.jpg', r)
57
58         # Calculate the NDVI
59
60         # Bottom of fraction
61         bottom = (r.astype(float) + b.astype(float))
62         bottom[bottom == 0] = 0.01 # Make sure we don't divide by
63         # zero!
64
65         ndvi = (r.astype(float) - b) / bottom
66         cv2.imwrite('/home/pi/ndvi_1.jpg', ndvi)
67         ndvi = contrast_stretch(ndvi)
68         cv2.imwrite('/home/pi/ndvi_2.jpg', ndvi)
```

```
66         ndvi = ndvi.astype(np.uint8)
67         cv2.imwrite('/home/pi/ndvi_3.jpg', ndvi)
68
69         # Convert to RGB
70         cv2.cvtColor(ndvi, cv2.COLOR_GRAY2RGB)
71
72         # Save NDVI image
73         cv2.imwrite('/home/pi/ndvi_color.jpg', ndvi);
74
75         # stream.truncate(0)
76
77 if __name__ == '__main__':
78     run()
```

---

CÓDIGO A.4: Programa para obtener imágenes NDVI usando OpenCV



## A.5 Código Arduino para la iteración 2

En esta sección el código de control de la iteración 2 es propuesto. Está basado en el algoritmo *The bubble rebound obstacle avoidance algorithm for mobile robots* [22] explicado en los capítulos 4 y 5. El código permite navegar por un mapa dado, evitando los obstáculos que pueda encontrar en el camino.

---

```
1 /*
2 main.ino is the main control sketch of GFE iteration 2.
3
4 An enable has been configured to wait for RPi to be fully booted.
5 Using the
6 alias it is possible to enable/disable and change the control mode
7 via SSH.
8 The manual mode is programmed to be controlled locally. Make sure
9 to connect the
10 joystick to the DA-15 connector. Otherwise, expect noise and weird
11 movements.
12 The automatic mode is based on the bubble rebound obstacle
13 avoidance algorithm
14 for mobile robots article, published by IOAN SUSNEA, VIOREL MINZU
15 and
16 GRIGORE VASILIU.
17
18 Modules implemented:
19 - Motors (IMS-1 modules)
20 - HC-SR04 (ultrasonic sensors)
21 - HMC5883L (magnetometer - GY-273)
22 - GY-GPS6MV2 (GPS module)
23 - RPi control pins
24
25 Author: Javier Macias
26 */
27 //----Importing libraries---//
28 #include <MOTOR.h> // Controlling motors with IMS-1 driver modules
29 #include <Arduino.h> // Libraries for compass HMC5883L
```

```
25 #include <Wire.h> // I2C library
26 #include <HMC5883L_Simple.h> // Compass library
27 #include <TinyGPS.h> // GPS library
28
29 // Create a compass
30 HMC5883L_Simple Compass;
31
32 // Only need to define two ultrasonic sensor pins.
33 const int EchoPin_0 = 24;
34 const int TriggerPin_0 = 25;
35
36 //----Robot parameters---/
37 const int L = 0.375; // Axle length (meters)
38
39
40 const int threshold = 60; // Centimeters of safe zone
41 const float heading_threshold = 5; // Degrees of tolerance
42 unsigned long lastTime = 0; // Used to store execution time of loop
43
44 // Variables related to GPS
45 float latitude, longitude;
46 unsigned long fix_age;
47 float latitude_command = 0;
48 float longitude_command = 0;
49 // Goal heading
50 // int heading_command = 110;
51 int heading_command = 170;
52
53 // Create a GPS
54 TinyGPS gps;
55
56 // Defining high-level control pins
57 const int enable_pin = 48;
58 const int mode_pin = 49;
59 bool enable = false; // Enable = 1; Disable = 0 if RPi is not ready
60 bool mode = false; // Manual = 0 (default); Automatic = 1
61
```

```
62 // Defining joystick inputs
63 int x_axis = 0; //Blue
64 int y_axis = 1; //Green
65 int z_axis = 2; //Yellow
66
67 const int button1 = 7; //White
68 const int button2 = 6; //Brown
69
70 int x_val = 0;
71 int y_val = 0;
72 int z_val = 0;
73 int button1_state = 0;
74 int button2_state = 0;
75
76 const int centre = 1024/2;
77
78 const int motors_wait = 10; //Wait for motors to move
79
80 // List of waypoints to follow
81 float waypoints[3][2] = {{1.111111, -1.111111}, {2.222222,
-2.222222}, {3.333333, -3.333333}};
82
83 // Declaring controller variables
84 float lastEps = 0;
85 int lastCommand = 0;
86 float lastHeading = 0;
87
88 void setup(){
89     Serial.begin(115200);
90     //Initialize GPS communication
91     Serial1.begin(9600);
92     Serial.println("main.ino");
93
94     // Initializing motor drivers
95     motor.begin();
96     /**
97     NOTE:
```

```
98     motor.begin() will change the prescaller of the timer0,
99     so the arduino function delay() millis() and micros() are
100    8 times slow than it should be.
101    Please use motor_delay(), motor_millis(), motor_micros()
102        instead them.
103
104    // Configuring HC-SR04 pins
105    for(int i = 0; i < 8; i++){
106        // Iterate through every sensor. Every two, next sensor
107        EchoPin and TriggerPin.
108        pinMode(EchoPin_0 + (2 * i), INPUT);
109        pinMode(TriggerPin_0 + (2 * i), OUTPUT);
110        delay(1); // Stabilization
111    }
112
113    // Initializing compass
114    Wire.begin();
115    Compass.SetDeclination(-4, 56, 'W'); // La Laguna
116    Compass.SetSamplingMode(COMPASS_SINGLE);
117    Compass.setScale(COMPASS_SCALE_130);
118    Compass.setOrientation(COMPASS_HORIZONTAL_Y_NORTH);
119
120    // Configuring joystick buttons
121    pinMode(button1, INPUT);
122    pinMode(button2, INPUT);
123
124    // Configuring high-level control pins
125    pinMode(enable_pin, INPUT);
126    pinMode(mode_pin, INPUT);
127
128    // Wait until RPi is ready
129    do{
130        enable = digitalRead(enable_pin);
131        Serial.println("Waiting for RPi to be ready...");
132        motor_delay(1000); // Wait for one second
133    }while(!enable);
```

```
133 }
134
135 void loop(){
136     // Wait for robot to be enabled
137     enable = digitalRead(enable_pin);
138     while(!enable){
139         enable = digitalRead(enable_pin);
140         Serial.println("Robot is disabled...");
141         motor_delay(1000); // Wait for one second
142     }
143
144     mode = digitalRead(mode_pin);
145     // Enter manual mode if mode is false and robot is enabled
146     while(!mode && enable){
147         joystick_control();
148         mode = digitalRead(mode_pin);
149         enable = digitalRead(enable_pin);
150     }
151
152     // Otherwise, automatic mode
153     if(mode && enable){
154         do{
155             get_location(gps);
156             Serial.print("latitude = ");
157             Serial.print((float)latitude, 6);
158             Serial.print(" ");
159             Serial.print("longitude = ");
160             Serial.println((float)longitude, 6);
161             delay(500);
162         }while(latitude == 0 && longitude == 0);
163
164         for(int i = 0; i < 3; i++){
165             // Set the next waypoint
166             latitude_command = waypoints[i][0];
167             Serial.print("latitude_command = ");
168             Serial.print((float)latitude_command, 6);
169             longitude_command = waypoints[i][1];
```

```
170     Serial.print(" longitude_command = ");
171     Serial.println((float)longitude_command, 6);
172
173     do{
174         // Recalculate heading after each iteration
175         heading_command = gps.course_to(latitude, longitude,
176                                         latitude_command, longitude_command);
177         Serial.print("heading_command = ");
178         Serial.println(heading_command);
179         automatic_mode(); //Navigate
180         get_location(gps);
181         Serial.print("latitude = ");
182         Serial.print((float)latitude, 6);
183         Serial.print(" ");
184         Serial.print("longitude = ");
185         Serial.println((float)longitude, 6);
186
187         enable = digitalRead(enable_pin);
188         }while(!goal_is_reached(latitude, longitude,
189                               latitude_command, longitude_command) && enable);
190
191         // Get out if !enable
192         if(!enable){
193             break;
194         }
195     }
196
197 // Read current position
198 static void get_location(TinyGPS &gps){
199     while (Serial1.available()) {
200         int c = Serial1.read();
201         if (gps.encode(c)) {
202             // Retrieves +/- lat/long in 100000ths of a degree
203             gps.f_get_position(&latitude, &longitude, &fix_age);
204         }
205     }
206 }
```

```
205     }
206 }
207
208 // Check if current location is close enough to the objective
209 int goal_is_reached(float latitude, float longitude, float
210   latitude_command, float longitude_command) {
211   if(abs(latitude - latitude_command) <= 0.00001 && abs(longitude
212     - longitude_command) <= 0.00001) {
213     return 1;
214   }
215   else{
216     return 0;
217   }
218 }
219
220 // Automatic mode function
221 void automatic_mode() {
222   // Calculate current and elapsed time for PI controller
223   unsigned long now = millis();
224   double timeChange = (double)(now - lastTime)/1000; //Seconds
225   Serial.print("timeChange = ");
226   Serial.println(timeChange);
227
228   float heading = Compass.GetHeadingDegrees();
229   Serial.print("heading = ");
230   Serial.println(heading);
231
232   unsigned int *sonar_readings;
233   sonar_readings = average_read_sensors();
234   for(int i = 0; i < 8; i++){
235     Serial.print(sonar_readings[i]);
236     Serial.print(" ");
237   }
238
239   // Obstacle avoidance algorithm
240   if(check_for_obstacles(sonar_readings, 0, 8)){
241     Serial.println("Obstacle detected!");
```

```
240
241     float alpha_r = 0;
242     do{
243         Serial.println("Computing rebound angle...");  

244         alpha_r = rebound_angle(sonar_readings);
245         Serial.print("alpha_r = ");
246         Serial.println(alpha_r);
247
248         Serial.println("Adjusting heading.");
249
250         heading = Compass.GetHeadingDegrees();
251         Serial.print("heading = ");
252         Serial.println(heading);
253
254         if(alpha_r < 0){
255             Serial.println("alpha_r < 0. Turning left.");
256             // Turn left. Equivalent to compute new heading +
257             // adjust motion in the paper.
258             adjust_heading(heading, alpha_r, 1, timeChange);
259         }
260         else{
261             Serial.println("alpha_r > 0. Turning right.");
262             adjust_heading(heading, alpha_r, 0, timeChange);
263         }
264
265         recheck_obstacles:
266         sonar_readings = average_read_sensors();
267         for(int i = 0; i < 8; i++){
268             Serial.print(sonar_readings[i]);
269             Serial.print(" ");
270         }
271         enable = digitalRead(enable_pin);
272         }while(check_for_obstacles(sonar_readings, 0, 8) && enable);
273
274         if(!goal_visible(sonar_readings) && enable){
275             Serial.println("Goal is not visible. Move forward!");
move_forward();
```

```
276         goto recheck_obstacles;
277     }
278 }
279 else{
280     // No obstacles, orientate to heading.
281     if(heading >= heading_command + heading_threshold || heading
282         <= heading_command - heading_threshold){
283         do{
284             if(((heading - heading_command < 0 && abs(heading -
285                 heading_command) < 180) || (heading -
286                 heading_command > 0 && abs(heading -
287                 heading_command) > 180))){
288                 Serial.println("No obstacles around, turning right
289                     to heading_command!");
290                 rotate_right(heading_command, timeChange);
291             }
292             else{
293                 Serial.println("No obstacles around, turning left
294                     to heading_command!");
295                 rotate_left(heading_command, timeChange);
296             }
297             heading = Compass.GetHeadingDegrees();
298             Serial.print("heading = ");
299             Serial.println(heading);
300             enable = digitalRead(enable_pin);
301             }while ((heading >= heading_command + heading_threshold
302                 || heading <= heading_command - heading_threshold) &&
303                 enable);
304         }
305         else{ // Continue forward
306             move_forward();
307         }
308     }
309     lastTime = now;
310 }
311 }
```

```
305 // Rotate robot to the new heading
306 int adjust_heading(float heading, float alpha_r, int direction,
307   double timeChange) {
308   if(direction == 1) {
309     // Calculate the new heading. Note: 0 for turning right, 1
310     // for left
311     float avoid_direction = calculate_heading(heading,
312       abs(alpha_r), 1);
313     Serial.print("avoid_direction = ");
314     Serial.println(avoid_direction);
315     do{
316       rotate_left(avoid_direction, timeChange);
317       heading = Compass.GetHeadingDegrees();
318       Serial.print("Heading: ");
319       Serial.println(heading);
320       enable = digitalRead(enable_pin);
321     }while((heading >= avoid_direction + heading_threshold ||
322         heading <= avoid_direction - heading_threshold) &&
323         enable);
324   }
325   else{
326     // Calculate the new heading. Note: 0 for turning right, 1
327     // for left
328     float avoid_direction = calculate_heading(heading,
329       abs(alpha_r), 0);
330     Serial.print("avoid_direction = ");
331     Serial.println(avoid_direction);
332     do{
333       rotate_right(avoid_direction, timeChange);
334       heading = Compass.GetHeadingDegrees();
335       Serial.print("Heading: ");
336       Serial.println(heading);
337       enable = digitalRead(enable_pin);
338     }while((heading >= avoid_direction + heading_threshold ||
339         heading <= avoid_direction - heading_threshold) &&
340         enable);
341   }
342 }
```

```
333 }
334
335 // Check if goal is visible.
336 int goal_visible(unsigned int *sonar_readings){
337     float heading = Compass.GetHeadingDegrees();
338
339     float right_limit = calculate_heading(heading, 90, 0);
340     float left_limit = calculate_heading(heading, 90, 1);
341
342     // The goal is on the right side.
343     if(heading_command > heading && heading_command < right_limit){
344         Serial.println("Checking obstacles on the right");
345         if(search_goal(sonar_readings, 0, 4)){
346             Serial.println("Goal was found on the right");
347             return 0;
348         }
349     else{
350         return 1; // Raise error if goal is not visible
351     }
352 }
353 else{ // The goal is on the left side.
354     Serial.println("Checking obstacles on the left");
355     if(search_goal(sonar_readings, 4, 8)){
356         Serial.println("Goal was found on the left");
357         return 0;
358     }
359     else{
360         return 1; // Raise error if goal is not visible
361     }
362 }
363 }

364 // See if goal is visible from the given sensors
365 int search_goal(unsigned int *sonar_readings, const int start,
366                  const int end){
367     unsigned int goal_boundary[8] = {50, 50, 50, 60, 60, 50, 50,
368                                    50}; // Centimeters
```

```
368
369     for(int i = start; i < end; i++) {
370         if(sonar_readings[i] <= goal_boundary[i]){
371             return 1; // Raise error if goal is not found
372         }
373     }
374     return 0;
375 }
376
377 // Filter sensor readings by doing the average
378 unsigned int *average_read_sensors() {
379     unsigned int sum[8] = {0}; //Initialize to zero.
380     static unsigned int average[8] = {0}; //Initialize to zero.
381     unsigned int *sonar_readings;
382
383     for(int j = 0; j < 2; j++){
384         sonar_readings = read_sensors();
385         for(int i = 0; i < 8; i++){
386             sum[i] += sonar_readings[i];
387         }
388     }
389
390     for(int k = 0; k < 8; k++){
391         average[k] = sum[k]/2;
392     }
393
394     return average;
395 }
396
397 // Store ultrasonic sensor readings. Sequence: 0 3 6 1 4 7 2 5
398 unsigned int *read_sensors(){
399     static unsigned int sonar_readings[8] = {0}; //Initialize to
400     zero.
401     int i = 0;
402     int cont = 0;
403     while(cont < 8){
```

```
403     sonar_readings[i] = calculate_distance(TriggerPin_0 + (2 *
404         i), EchoPin_0 + (2 * i));
405     if(cont == 2){
406         i = 1;
407     }
408     else if(cont == 5){
409         i = 2;
410     }
411     else{
412         i += 3;
413     }
414     cont++;
415     delay(10); // 10 ms delay to avoid false positives
416 }
417 }
418
419 // Compute rebound angle
420 float rebound_angle(unsigned int *sonar_readings){
421     const float alpha_0 = 180/8;
422     float num = 0;
423     float den = 0;
424     for(int i = -4; i < 5; i++){
425         if(i == 0){ // Zero is not taken.
426             i = 1;
427         }
428
429         float alpha_i = i * alpha_0;
430
431         num += alpha_i * sonar_readings[i];
432         den += sonar_readings[i];
433     }
434
435     float alpha_r = num/den;
436
437     return alpha_r;
438 }
```

```
439
440 // Compare whether any reading is below threshold
441 int check_for_obstacles(unsigned int *sonar_readings, const int
442     start, const int end) {
443     unsigned int bubble_boundary[8] = {20, 20, 30, 40, 40, 30, 20,
444         20}; // Centimeters
445
446     for(int i = start; i < end; i++) {
447         if(sonar_readings[i] <= bubble_boundary[i]) {
448             return 1;
449         }
450     }
451
452     return 0;
453 }
454
455 // Read the distance of an ultrasonic sensor. Raw output,
456 // unfiltered.
457 int calculate_distance(const int TriggerPin, const int EchoPin) {
458     long duration, distanceCm;
459
460     digitalWrite(TriggerPin, LOW); //Setting to LOW for 4us for
461         cleaner reading
462     delayMicroseconds(4);
463     digitalWrite(TriggerPin, HIGH); //Triggering for 10us
464     delayMicroseconds(10);
465     digitalWrite(TriggerPin, LOW);
466
467     duration = pulseIn(EchoPin, HIGH); //Measuring the time between
468         pulses in microseconds
469     distanceCm = duration * 10 / 292 / 2; //Converting to cm
470
471     return distanceCm;
472 }
473
474
475 // Used to calculate proper heading values
```

```
470 float calculate_heading(float heading, float alpha_r, int
471     direction) {
472     if(direction == 0) {
473         if(heading + alpha_r > 360) {
474             return heading + alpha_r - 360;
475         }
476         else{
477             return heading + alpha_r;
478         }
479     else{
480         if(heading - alpha_r < 0) {
481             return heading - alpha_r + 360;
482         }
483         else{
484             return heading - alpha_r;
485         }
486     }
487 }
488
489 // PI controller for rotation of the robot
490 int calculate_command(float goal, double timeChange) {
491     float heading = Compass.GetHeadingDegrees();
492     float eps = abs(heading - goal);
493     float deltaPV = abs(heading - lastHeading);
494
495     // Kp and Ti manually adjusted
496     int Kp = 4;
497     int Ti = 2;
498     float max_command = 130;
499
500     float command = (Kp * (deltaPV + 1/Ti * (eps * timeChange)) +
501                     lastCommand) * (max_command/180);
502
503     // Clamp to avoid values over max_command due to excessive
504     // error/gain
505     command = constrain(command, 60, max_command);
```

```
504
505     lastEps = eps;
506     lastCommand = command;
507     lastHeading = heading;
508
509     return command;
510 }
511
512 // Rotate left
513 void rotate_left(float goal, double timeChange) {
514     Serial.println("Rotating left!");
515     float command = calculate_command(goal, timeChange);
516     motor.set(A, command, REVERSE);
517     motor.set(B, command, FORWARD);
518     motor_delay(10);
519 }
520
521 // Rotate right
522 void rotate_right(float goal, double timeChange) {
523     Serial.println("Rotating right!");
524     float command = calculate_command(goal, timeChange);
525     motor.set(A, command, FORWARD);
526     motor.set(B, command, REVERSE);
527     motor_delay(10);
528 }
529
530 // Move forward
531 // IMPLEMENTAR PI(?)
532 void move_forward(){
533     Serial.println("Going forward! @ open space");
534     motor.set(A, 120, FORWARD); // Drift correction
535     motor.set(B, 130, FORWARD);
536     motor_delay(10);
537 }
538
539 void joystick_control(){
540     //Read, store and correct raw values from joystick
```

```
541     x_val = calculate_average(x_axis);
542     Serial.print("x_val = ");
543     Serial.println(x_val);
544     y_val = calculate_average(y_axis);
545     Serial.print("y_val = ");
546     Serial.println(y_val);
547     // For some reason z_axis is not working.
548     // z_val = calculate_average(z_axis);
549     // Serial.print("z_val = ");
550     // Serial.println(z_val);
551     button1_state = digitalRead(button1);
552     button2_state = digitalRead(button2);
553
554     x_val = 1024 - x_val;
555     y_val = 1024 - y_val;
556
557     int x = x_val - centre;
558     int y = y_val - centre;
559
560     // Computing speed and direction in function of joystick
561     // position
562     // Convert to polar coordinates
563     float r = hypot(x, y);
564     float t = atan2(y, x);
565
566     // Rotate by 45 degrees
567     t = t - 3.14159265358979323846 / 4;
568
569     // Back to cartesian
570     float left = r * cos(t);
571     float right = r * sin(t);
572
573     // Rescale the new coords
574     left = left * sqrt(2);
575     right = right * sqrt(2);
576
577     // Clamp to -245/+245 to avoid over power
```

```
577     left = fmax(-245, fmin(left, 245));  
578     right = fmax(-245, fmin(right, 245));  
579  
580     float left_abs = abs(left);  
581     float right_abs = abs(right);  
582  
583     // Controlling the motors  
584     if(button2_state == 0 || (x <= 100 && x >= -100 && y <= 100 &&  
      y >= -100)) {  
585         Serial.println("Stop!");  
586         motor.set(A, 0, FORWARD);  
587         motor.set(B, 0, FORWARD);  
588         motor_delay(motors_wait);  
589     }  
590     else {  
591         if(y > 0) {  
592             if(left > 0 && right > 0) {  
593                 motor.set(A, left, FORWARD);  
594                 motor.set(B, right, FORWARD);  
595                 motor_delay(motors_wait);  
596             }  
597             else if(left > 0 && right < 0) {  
598                 motor.set(A, left, FORWARD);  
599                 motor.set(B, right_abs, REVERSE);  
600                 motor_delay(motors_wait);  
601             }  
602             else if(left < 0 && right > 0) {  
603                 motor.set(A, left_abs, REVERSE);  
604                 motor.set(B, right, FORWARD);  
605                 motor_delay(motors_wait);  
606             }  
607         }  
608         else {  
609             if(left < 0 && right < 0) {  
610                 motor.set(A, right_abs, REVERSE);  
611                 motor.set(B, left_abs, REVERSE);  
612                 motor_delay(motors_wait);  
613             }  
614         }  
615     }  
616 }
```

```
613         }
614     else if(left > 0 && right < 0){
615         motor.set(A, right_abs, REVERSE);
616         motor.set(B, left, FORWARD);
617         motor_delay(motors_wait);
618     }
619     else{
620         motor.set(A, right, FORWARD);
621         motor.set(B, left_abs, REVERSE);
622         motor_delay(motors_wait);
623     }
624 }
625 }
626 }
627
628 // Calculate average of analog values
629 int calculate_average(int axis){
630     int value = 0;
631     int num = 10;
632     for(int i = 0; i < num; i++){
633         value += analogRead(axis);
634     }
635     int average = value/num;
636
637     return average;
638 }
```

---

CÓDIGO A.5: Programa principal de la iteración 2

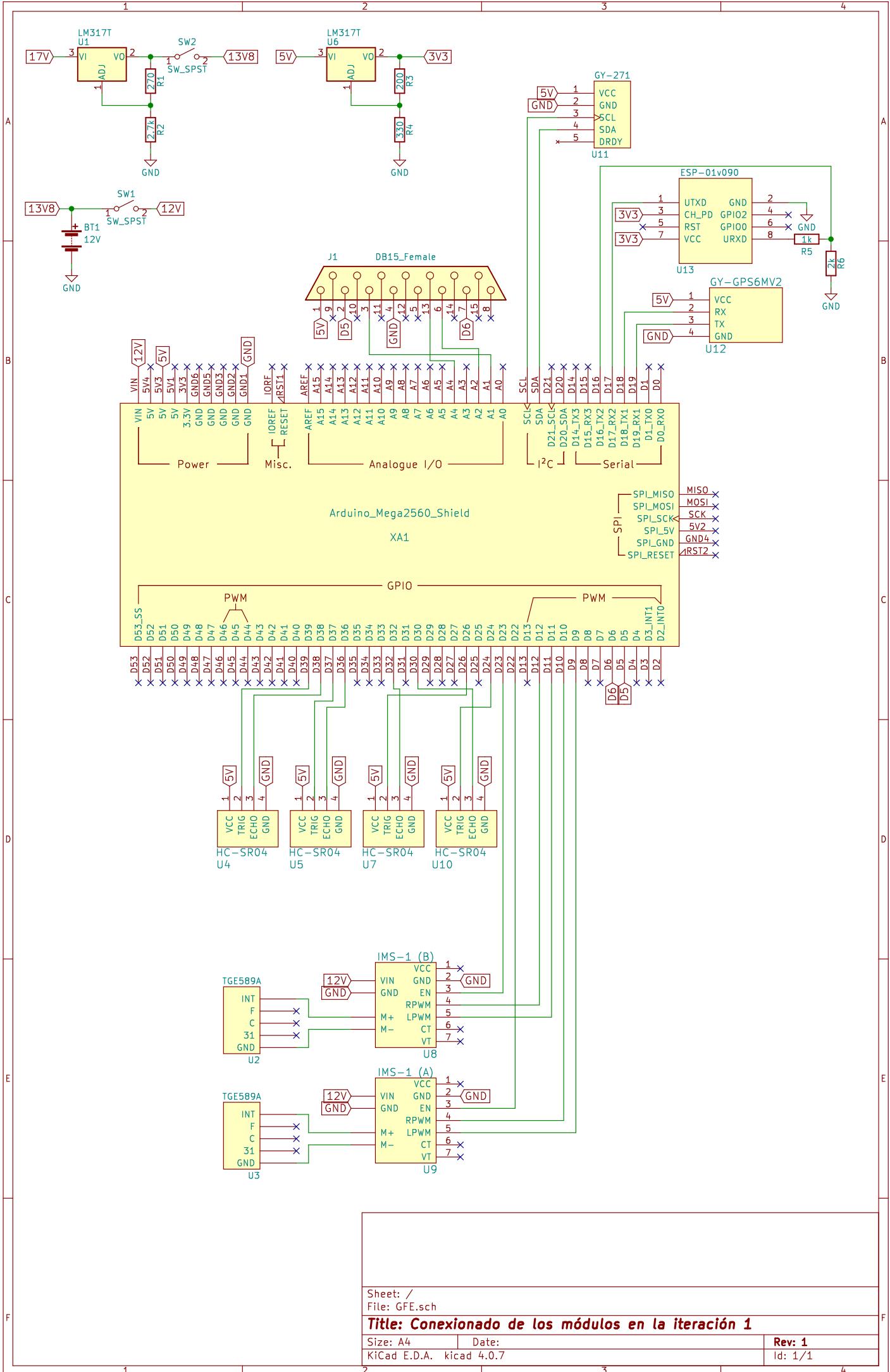


## **Apéndice B**

### **Esquemáticos**

En este anexo se muestran los esquemáticos necesarios para documentar los circuitos electrónicos del vehículo en sus sucesivas iteraciones. Para ello, se han creado, diseñando y anotado los componentes necesarios en KiCAD.







A

B

C

D

E

F

A

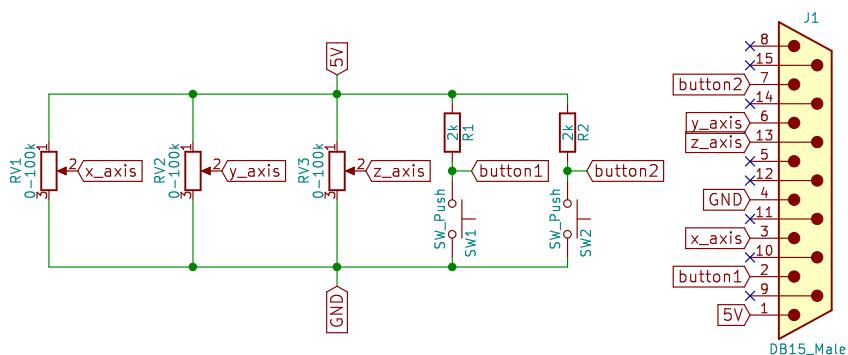
B

C

D

E

F



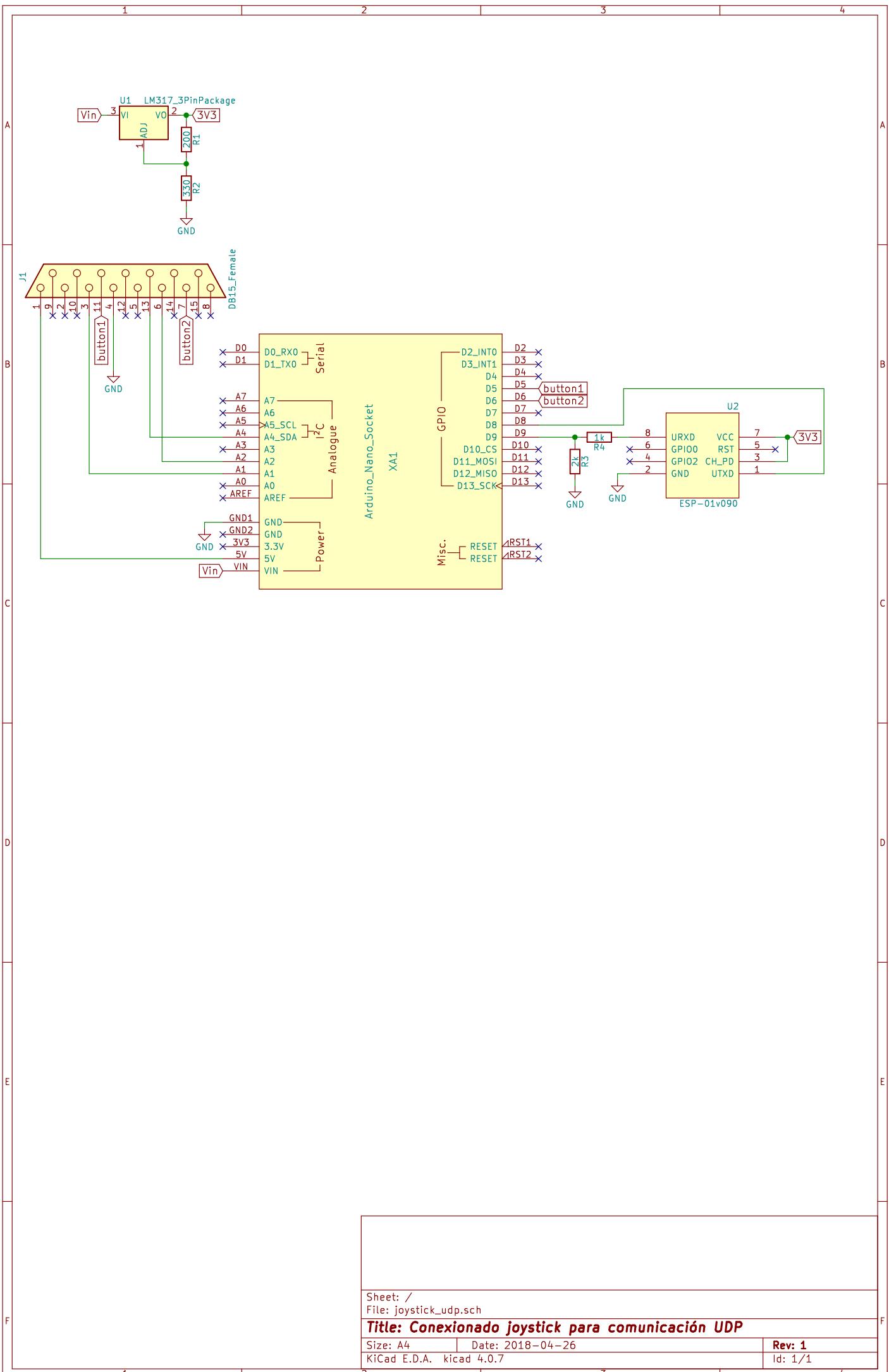
Sheet: /  
File: joystick.sch

**Title: Circuito interior del joystick**

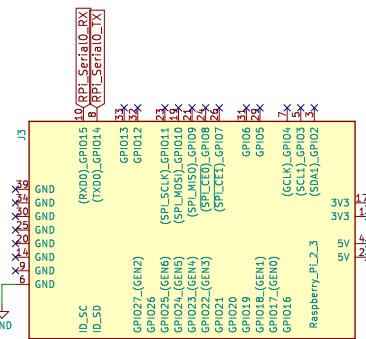
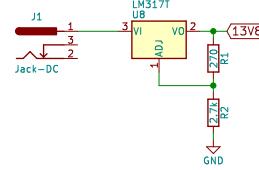
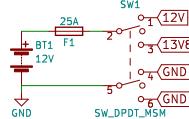
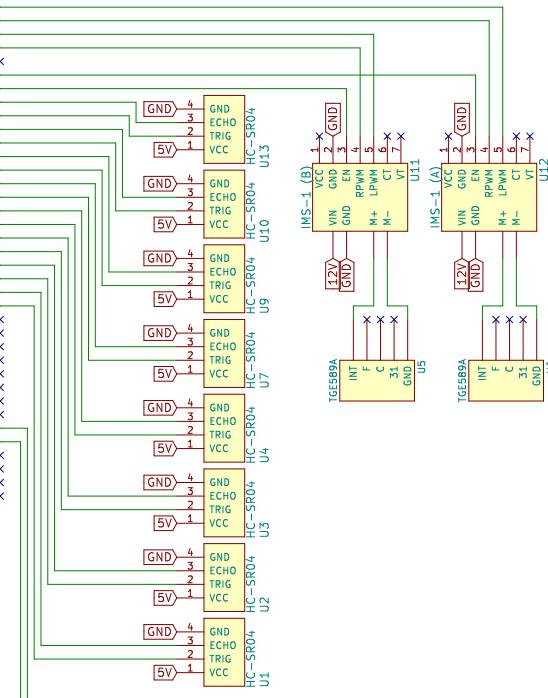
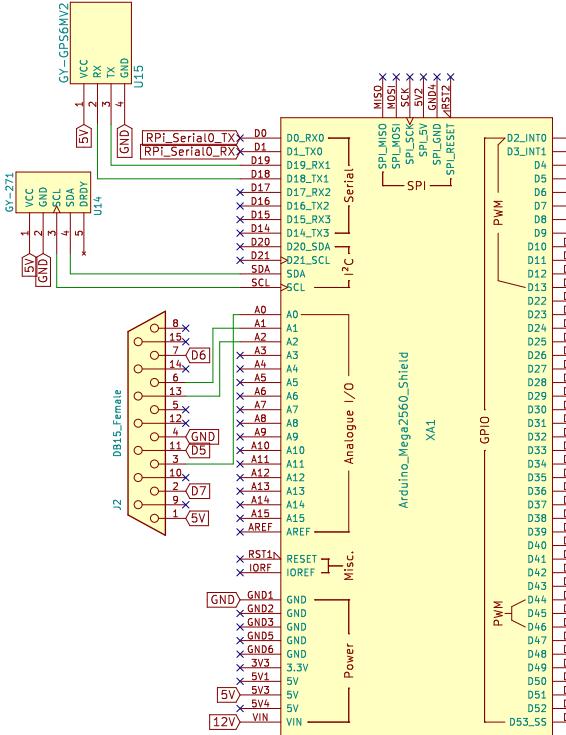
Size: A4 | Date:  
KiCad E.D.A. kicad 4.0.7

Rev: 1  
Id: 1/1









Sheet: /  
File: GFE\_iter2.sch  
**Title: Conexiónado de los módulos en la iteración 2**  
Size: A3 Date: Rev: 1  
KiCad E.D.A. kicad 4.0.7 Id: 1/1

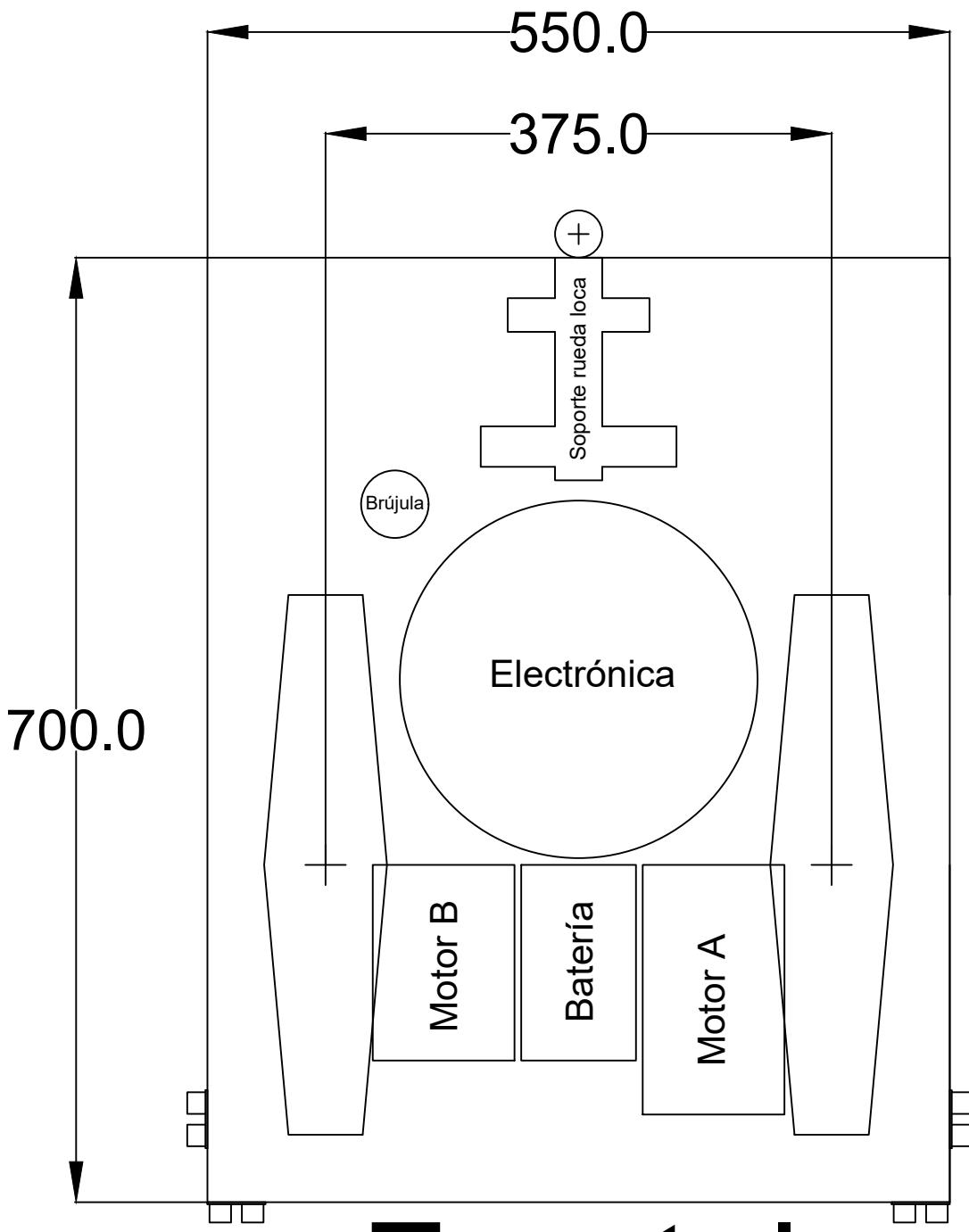


## **Apéndice C**

### **Planos**

En este anexo se muestran los planos necesarios para documentar los modelos físicos que se han tenido que diseñar y crear en las sucesivas versiones.



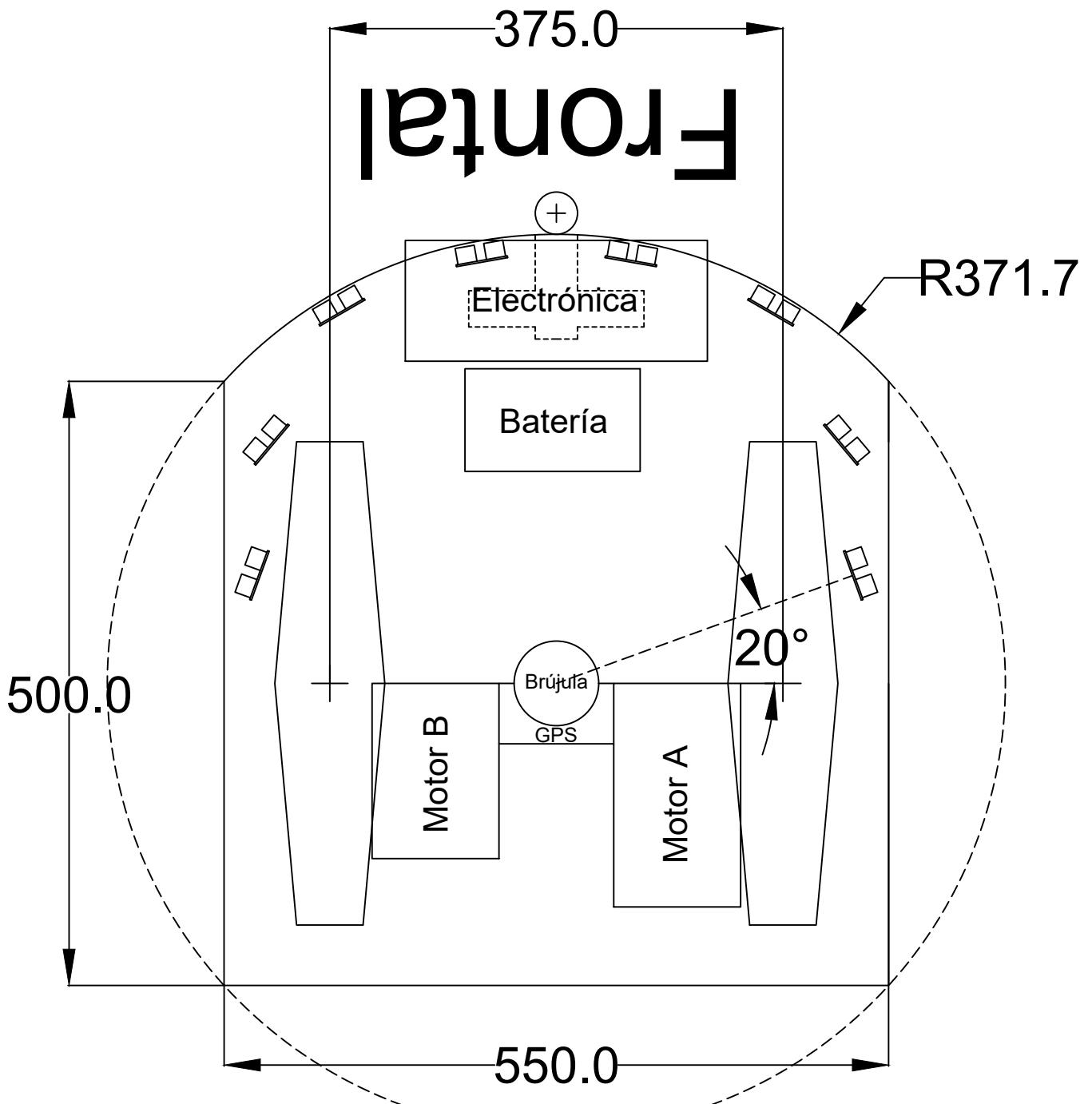


# Frontal

## Diseño electrónico de un vehículo autónomo

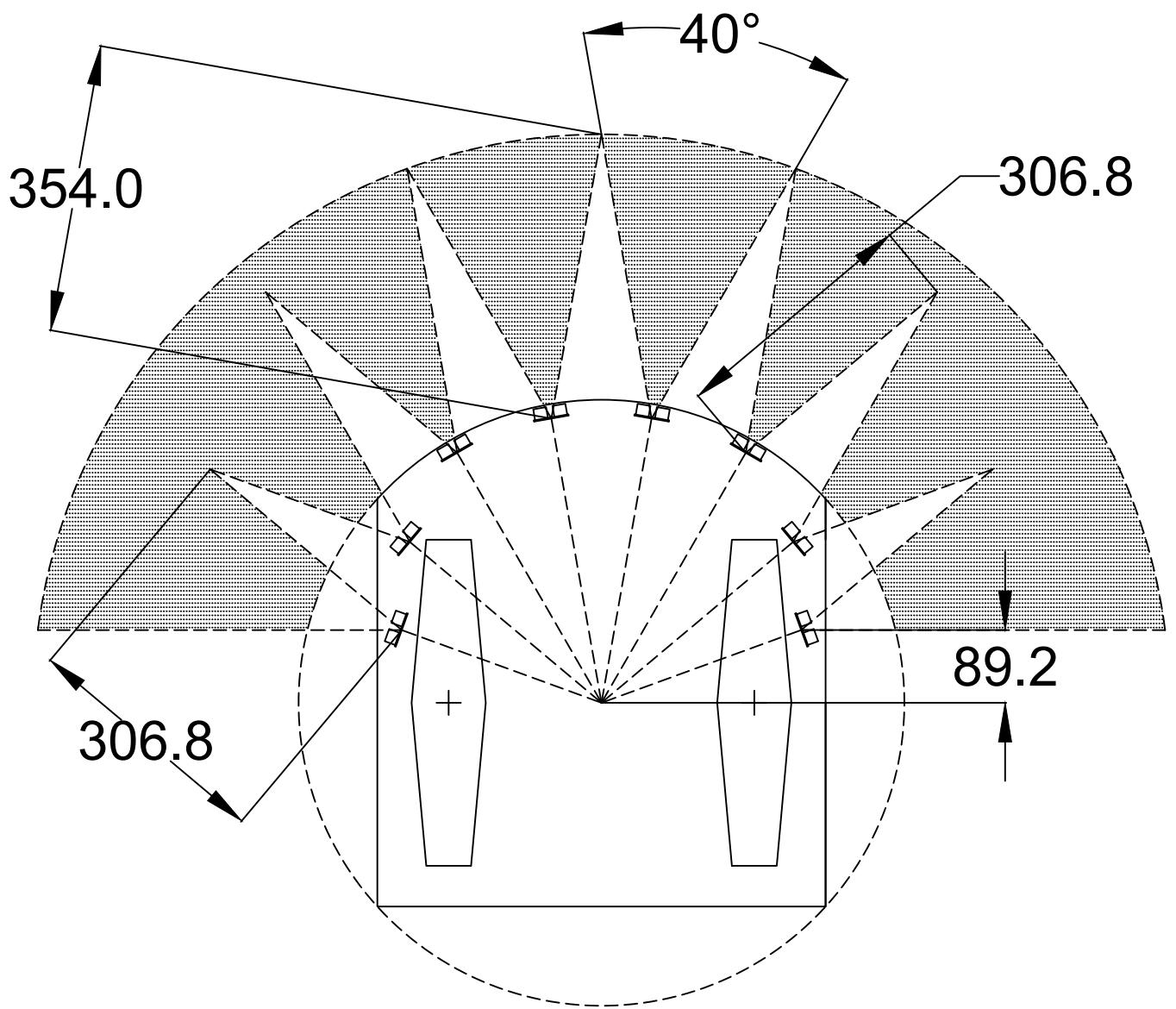
Fecha	Fecha	Autor	<b>ULL</b> Escuela Superior de Ingeniería y Tecnología <i>Grado Ingeniería Electrónica Ind. y Autom.</i> <i>Universidad de La Laguna</i>
<i>Dibujado</i>	<i>Fecha</i>	<i>Javier</i>	
<i>Comprobado</i>	<i>Fecha</i>	<i>Macías Solá</i>	
<i>Id. s. normas</i>	<i>UNE-EN-DIN</i>		
ESCALA: 1:5	Vista de planta del robot (iter. 1)		Nº P.: 1 Nom.Arch: <i>iter_1.dwg</i>





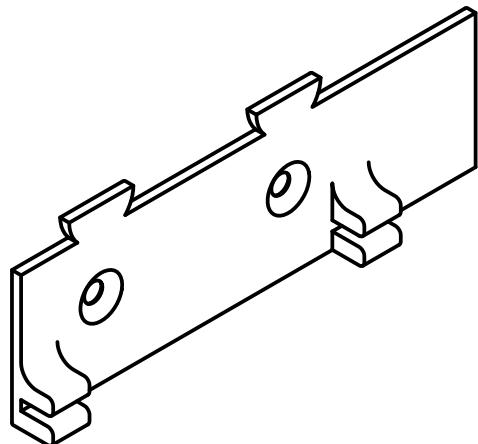
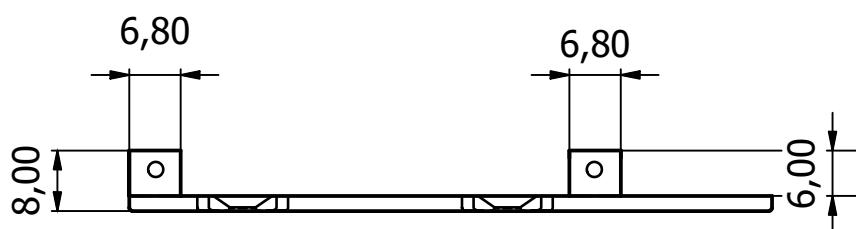
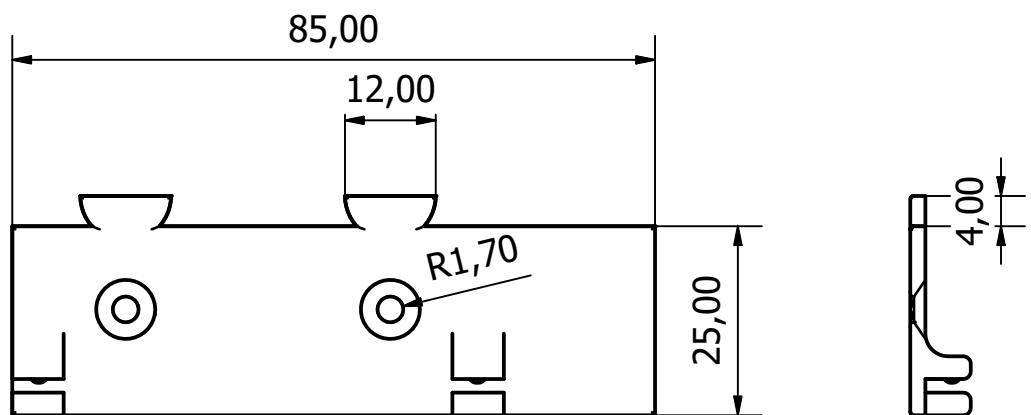
Diseño electrónico de un vehículo autónomo				
Fecha	Fecha	Autor		
Dibujado	30/05/18	Javier	<b>ULL</b> Universidad de La Laguna	Escuela Superior de Ingeniería y Tecnología Grado Ingeniería Electrónica Ind. y Autom. Universidad de La Laguna
Comprobado	30/05/18	Macías Solá		
Id. s. normas	UNE-EN-DIN			
ESCALA: 1:5	Vista de planta del robot (iter. 2)			Nº P.: 2 Nom.Arch: iter_2.dwg





Diseño electrónico de un vehículo autónomo				
Fecha	Fecha	Autor		
Dibujado	30/05/18	Javier	<b>ULL</b> Universidad de La Laguna	Escuela Superior de Ingeniería y Tecnología <i>Grado Ingeniería Electrónica Ind. y Autom.</i> <i>Universidad de La Laguna</i>
Comprobado	30/05/18	Macías Solá		
Id. s. normas	UNE-EN-DIN			
ESCALA: 1:8	Visión del robot (iter. 2)			Nº P.: 3
				Nom.Arch: iter_2.dwg

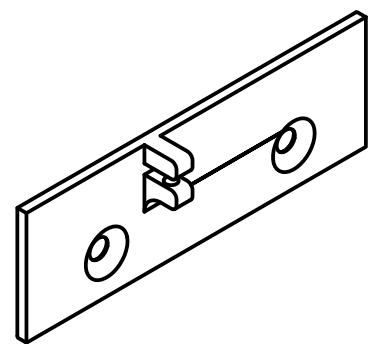
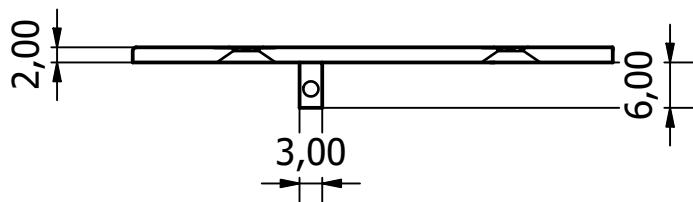
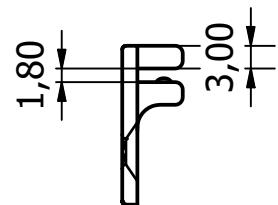
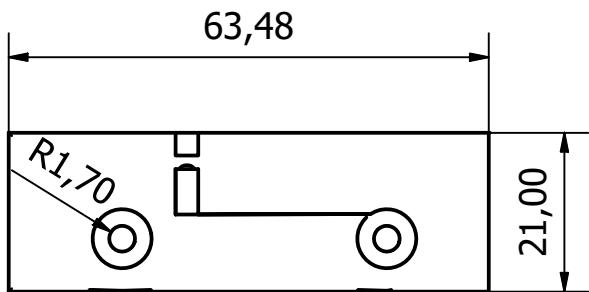




### Diseño electrónico de un vehículo autónomo

Fecha	Fecha	Autor	<b>ULL</b> Universidad de La Laguna	Escuela Superior de Ingeniería y Tecnología <i>Grado Ingeniería Electrónica Ind. y Autom.</i> <i>Universidad de La Laguna</i>
Dibujado	30/05/18	Javier		
Comprobado	30/05/18	Macías Solá		
Id. s. normas	UNE-EN-DIN			
ESCALA: 1:1	Soporte Raspberry Pi 3 Model B		Nº P.: 4	Nom.Arch: rpi_support.blend

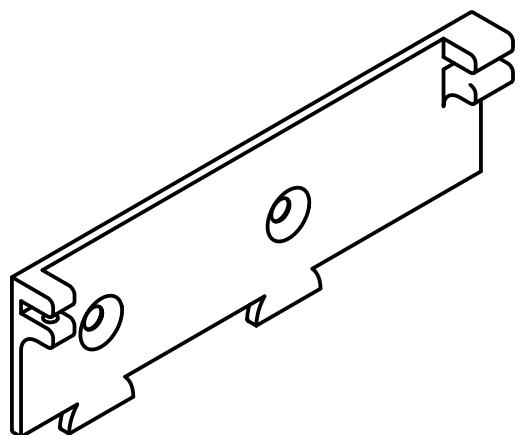
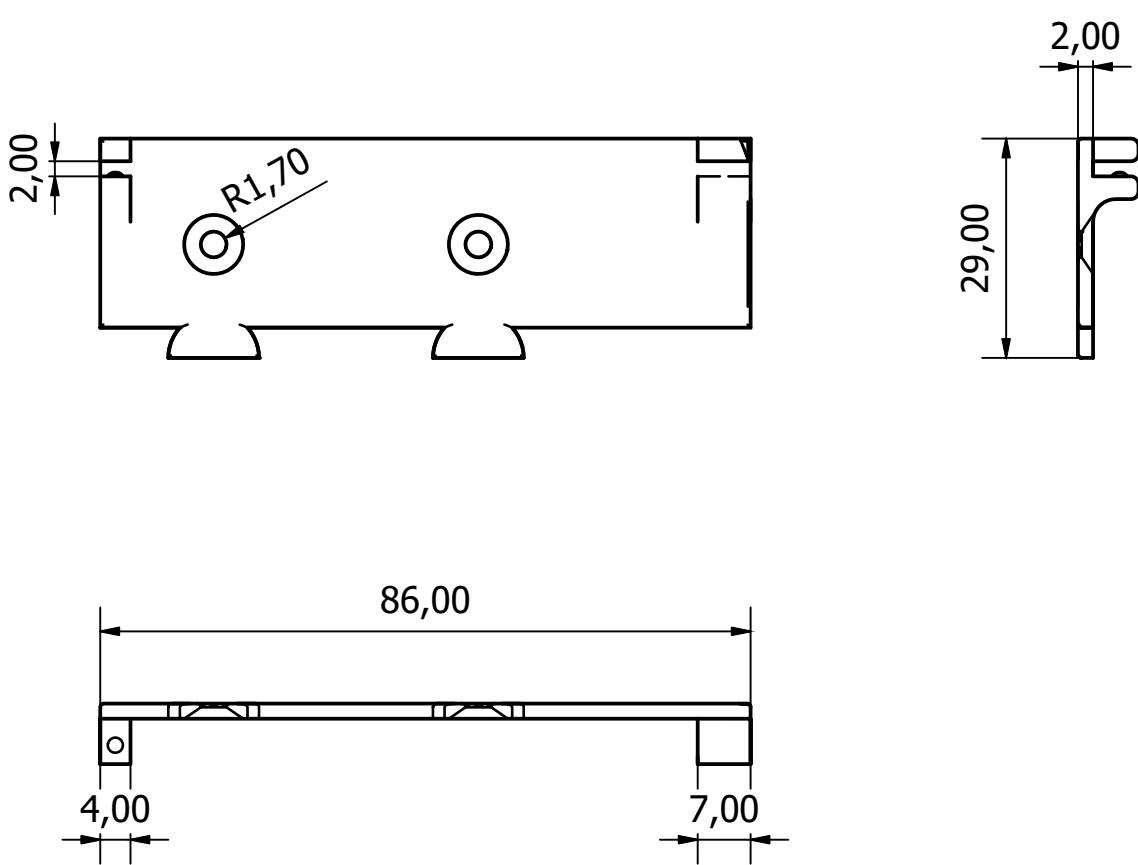




### Diseño electrónico de un vehículo autónomo

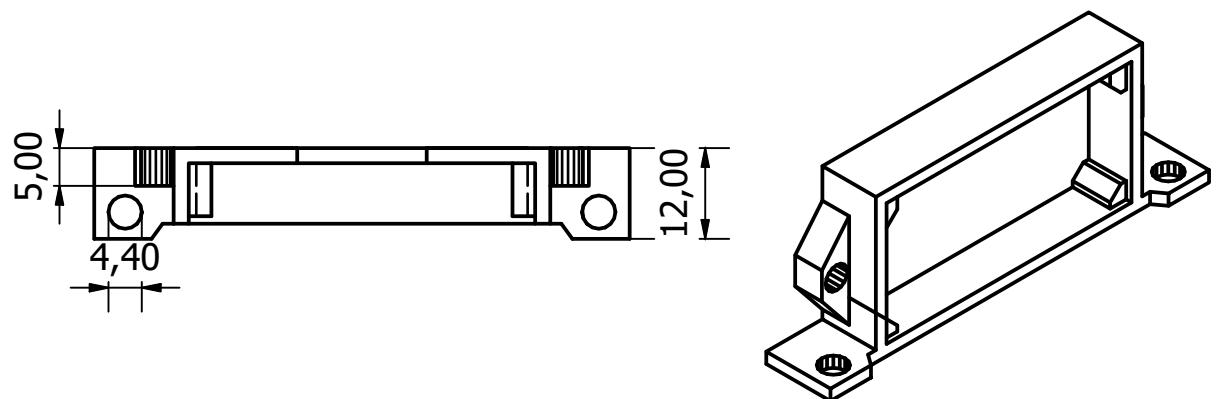
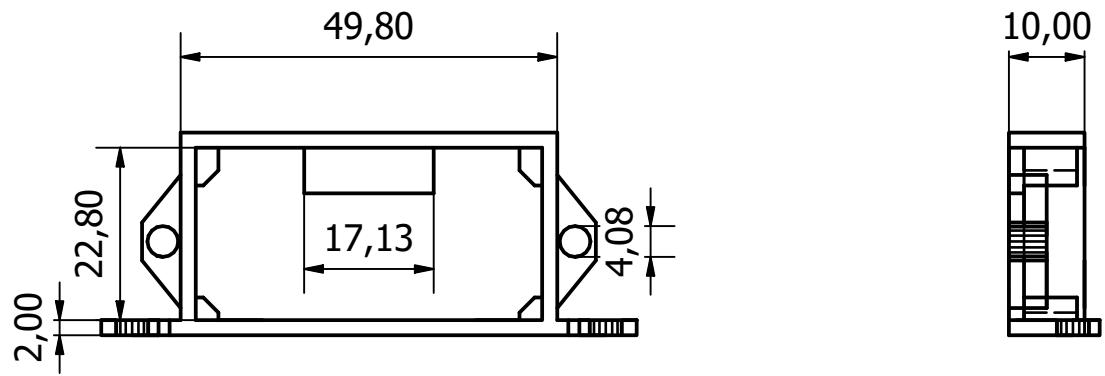
Fecha	Fecha	Autor		
Dibujado	30/05/18	Javier	<b>ULL</b> Universidad de La Laguna	
Comprobado	30/05/18	Macías Solá	Escuela Superior de Ingeniería y Tecnología Grado Ingeniería Electrónica Ind. y Autom. Universidad de La Laguna	
Id. s. normas	UNE-EN-DIN			
ESCALA: 1:1	Soporte convertidor DC-DC			Nº P.: 5 Nom.Arch: dc_dc_support.blend





Diseño electrónico de un vehículo autónomo		
Fecha	Fecha	Autor
Dibujado	30/05/18	Javier
Comprobado	30/05/18	Macías Solá
Id. s. normas	UNE-EN-DIN	
ESCALA: 1:1	Soporte Arduino Mega 2560	Nº P.: 6 Nom.Arch: arduino_support.blend

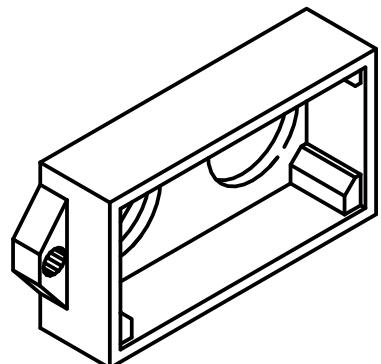
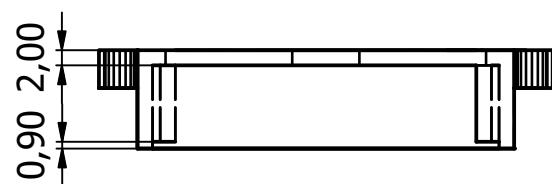
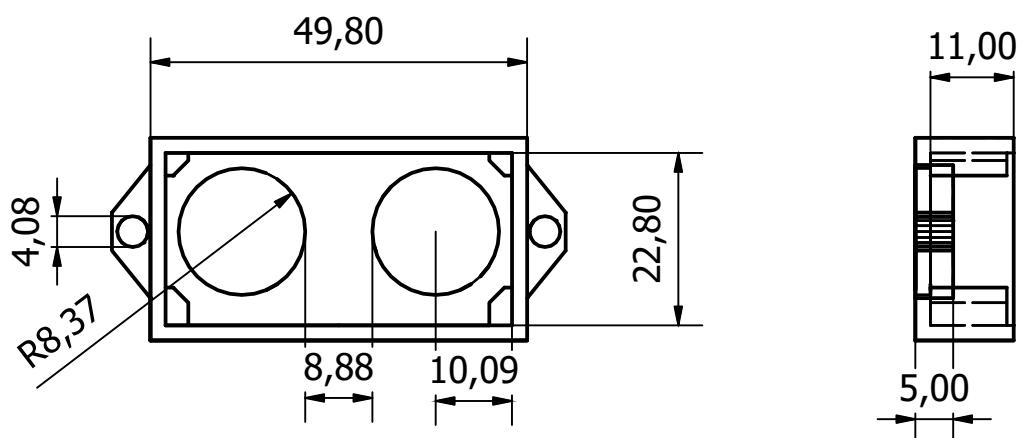




### Diseño electrónico de un vehículo autónomo

Fecha	Fecha	Autor		
Dibujado	30/05/18	Javier	<b>ULL</b> Universidad de La Laguna	
Comprobado	30/05/18	Macías Solá	Escuela Superior de Ingeniería y Tecnología Grado Ingeniería Electrónica Ind. y Autom. Universidad de La Laguna	
Id. s. normas	UNE-EN-DIN			
ESCALA: 1:1	Tapa trasera para ultrasonidos HC-SR04			Nº P.: 7
				Nom.Arch: case.blend





### Diseño electrónico de un vehículo autónomo

Fecha	Fecha	Autor		
Dibujado	30/05/18	Javier		
Comprobado	30/05/18	Macías Solá		
Id. s. normas	UNE-EN-DIN			
ESCALA: 1:1	Tapa frontal para ultrasonidos HC-SR04			Nº P.: 8 Nom.Arch: case.blend
<b>ULL</b> Universidad de La Laguna		Escuela Superior de Ingeniería y Tecnología Grado Ingeniería Electrónica Ind. y Autom. Universidad de La Laguna		



## Referencias y bibliografía

- [1] *1.2 V to 37 V adjustable voltage regulators*, DocID2154, LM217, LM317, Rev. 19, ST Electronics, mar. de 2014. dirección: <http://www.st.com/resource/en/datasheet/lm317.pdf>.
- [2] *3-Axis Digital Compass IC HMC5883L*, HMC5883L, Rev. D, Honeywell, mar. de 2011. dirección: [https://cdn-shop.adafruit.com/datasheets/HMC5883L\\_3-Axis\\_Digital\\_Compass\\_IC.pdf](https://cdn-shop.adafruit.com/datasheets/HMC5883L_3-Axis_Digital_Compass_IC.pdf).
- [3] P. Angelov, *Handbook on Computational Intelligence*, ép. Series on Computational Intelligence. World Scientific, 2016, vol. 1+2, ISBN: 9814675008,9789814675000. dirección: <http://gen.lib.rus.ec/book/index.php?md5=5f1c4656b7aca18fc8f3825113da94ed>.
- [4] A. Atyabi y S. Nefti-Meziani, «Applications of Computational Intelligence to Robotics and Autonomous Systems», en *Handbook on Computational Intelligence*, ép. Series on Computational Intelligence, P. Angelov, ed., vol. 1+2, World Scientific, 2016, cap. 25, págs. 862-902, ISBN: 9814675008,9789814675000. dirección: <http://gen.lib.rus.ec/book/index.php?md5=5f1c4656b7aca18fc8f3825113da94ed>.
- [5] *BCM2835 ARM Peripherals*, Broadcom Europe Ltd., feb. de 2012. dirección: <https://www.raspberrypi.org/app/uploads/2012/02/BCM2835-ARM-Peripherals.pdf>.
- [6] Bportaluri. (2017). Arduino WiFi library for ESP8266 modules, dirección: <https://github.com/bportaluri/WiFiEsp/>.
- [7] Elechouse. (2013). Arduino library for ELECHOUSE 50A Dual-Channel motor drive module, dirección: <https://github.com/elechouse/motor>.
- [8] *GS Technical Data Sheet*, GS GS-GT12A-BS, GS, mayo de 2018. dirección: <https://batterylookupgb.gs-battery.com/datasheet/download/pdf/sku/GS-GT12A-BS>.

- [9] HPBalter. (2018). Modular Raspberry Pi holder, dirección: <https://www.thingiverse.com/thing:2771426>.
- [10] *IMS-1 motor driver module*, IMS-1, Wingxin. dirección: [https://www.xsimulator.net/community/attachments/ims-1\\_en-pdf](https://www.xsimulator.net/community/attachments/ims-1_en-pdf).
- [11] V. M. Ioan Susnea y G. Vasiliu, «Simple, Real-time Obstacle Avoidance Algorithm for Mobile Robots», en *Proceedings of the 8th WSEAS International Conference on Computational Intelligence, Man-machine Systems and Cybernetics*, ép. CIM-MACS'09, Puerto De La Cruz, Tenerife, Canary Islands, Spain: World Scientific, Engineering Academy y Society (WSEAS), 2009, págs. 24-29, ISBN: 978-960-474-144-1. dirección: <http://dl.acm.org/citation.cfm?id=1736097.1736102>.
- [12] M. King, *Process Control: A Practical Approach*. Wiley, 2011, ISBN: 0470975873. dirección: <https://www.wiley.com/en-es/Process+Control:+A+Practical+Approach-p-9780470976661>.
- [13] *Lidar Lite v3 Operation Manual and Technical Specifications*, Lidar Lite v3, Rev. 0A, Garmin, sep. de 2016. dirección: [https://static.garmin.com/pumac/LIDAR\\_Lite\\_v3\\_Operation\\_Manual\\_and\\_Technical\\_Specifications.pdf](https://static.garmin.com/pumac/LIDAR_Lite_v3_Operation_Manual_and_Technical_Specifications.pdf).
- [14] Mikalhart. (2013). A compact Arduino NMEA (GPS) parsing library, dirección: <https://github.com/mikalhart/TinyGPS/>.
- [15] A. K. Nasir, M. Taj y M. F. Khan, «Evaluation of Microsoft Kinect Sensor for Plant Health Monitoring», *IFAC-PapersOnLine*, vol. 49, n.º 16, págs. 221-225, 2016, 5th IFAC Conference on Sensing, Control and Automation Technologies for Agriculture AGRICONTROL 2016, ISSN: 2405-8963. DOI: <https://doi.org/10.1016/j.ifacol.2016.10.041>. dirección: <http://www.sciencedirect.com/science/article/pii/S2405896316316044>.
- [16] *NEO-6 Data Sheet*, GPS.G6-HW-09005-E, Rev. E, u-Blox, dic. de 2011. dirección: [https://www.u-blox.com/sites/default/files/products/documents/NEO-6\\_DataSheet\\_\(GPS.G6-HW-09005\).pdf](https://www.u-blox.com/sites/default/files/products/documents/NEO-6_DataSheet_(GPS.G6-HW-09005).pdf).
- [17] *Real-time GPS Data Reception and Parsing*, AN002, Rev. 1.0, Parallax Semiconductor, 2011. dirección: <https://www.parallax.com/sites/default/files/downloads/AN002-GPS-NMEA0183-v1.0.pdf>.
- [18] Robintw. (2016). Raspberry PI NDVI Code, dirección: <https://github.com/robintw/RPiNDVI>.

- [19] ROSbots. (2018). Ready-to-use Image: Raspbian Stretch + ROS + OpenCV, dirección: <https://medium.com/@rosbots/ready-to-use-image-raspbian-stretch-ros-opencv-324d6f8dc96>.
- [20] Scary-Terry. (2016). Using a wiper motor in your Halloween projects, dirección: <http://www.scary-terry.com/wipmtr/wipmtr.htm>.
- [21] Sleemanj. (2017). Simple to use Arduino library to interface to HMC5883L Magnetometer (Digital Compass), dirección: [https://github.com/sleemanj/HMC5883L\\_Simple](https://github.com/sleemanj/HMC5883L_Simple).
- [22] I. Susnea, A. Filipescu, G. Vasiliu, G. Coman y A. Radaschin, «The bubble rebound obstacle avoidance algorithm for mobile robots», en *IEEE ICCA 2010*, jun. de 2010, págs. 540-545. DOI: 10.1109/ICCA.2010.5524302.
- [23] D. Taylor. (2010). Using Diamond Coordinates to Power a Differential Drive, dirección: <https://github.com/declanshanaghy/JabberBot/raw/master/Docs/Using%20Diamond%20Coordinates%20to%20Power%20a%20Differential%20Drive.pdf>.
- [24] C. J. Tucker, «Red and photographic infrared linear combinations for monitoring vegetation», inf. téc. 2, mayo de 1979, págs. 127-150. dirección: <https://www.sciencedirect.com/science/article/pii/0034425779900130>.
- [25] *Ultrasonic ranging module: HC-SR04*, ITead Studios, nov. de 2010. dirección: <https://www.electroschematics.com/wp-content/uploads/2013/07/HC-SR04-datasheet-version-2.pdf>.
- [26] I. Ünal y M. Topakci, «Design of a Remote-Controlled and GPS-Guided Autonomous Robot for Precision Farming», *International Journal of Advanced Robotic Systems*, vol. 12, n.º 12, pág. 194, 2015. DOI: 10.5772/62059. eprint: <https://doi.org/10.5772/62059>. dirección: <https://doi.org/10.5772/62059>.