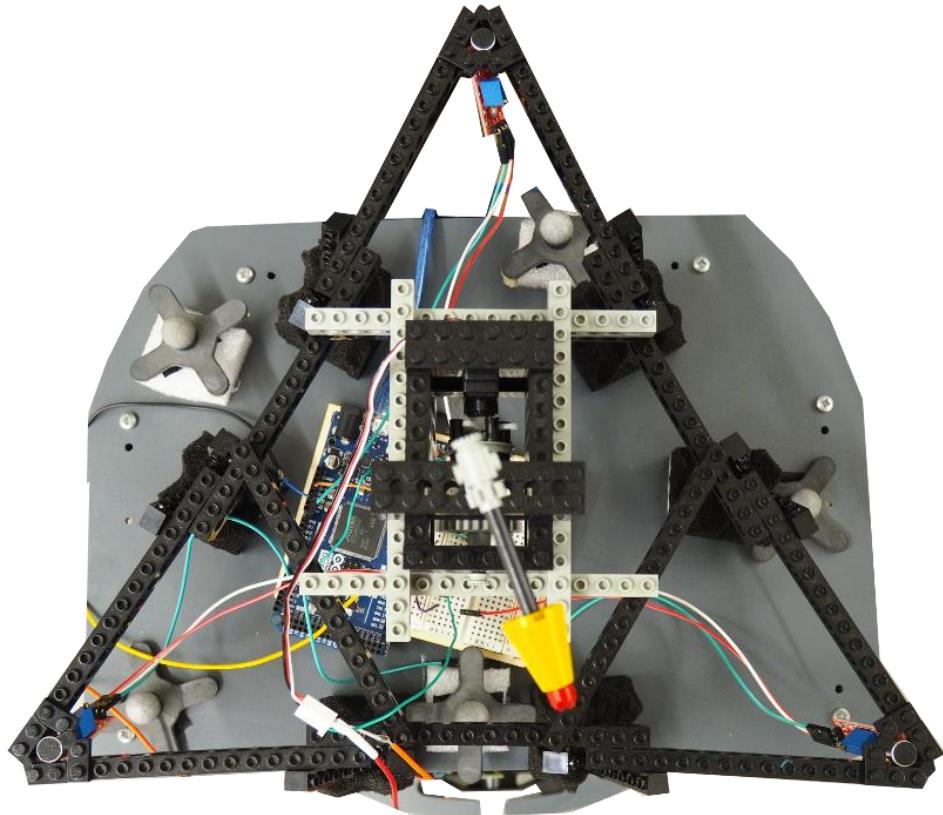


**Universidad Pablo de Olavide  
Grado en Ingeniería Informática en Sistemas de Información**

# **Robot de navegación autónoma guiado por un pulso sonoro**



**Trabajo de Fin de Grado  
Autor:  
Marco Corzetto Conflan**

**Tutores:**  
Manuel Béjar Domínguez  
José Antonio Cobano Suárez



## *Agradecimientos*

Llegue mi agradecimiento a los profesores Manuel Béjar Domínguez y José Antonio Cobano Suárez, tutores de este trabajo de fin de grado, por su asesoramiento y apoyo constante.

También a Luis Merino Cabañas, profesor titular de la materia de Robótica y Visión Artificial de la Universidad Pablo de Olavide, que de manera inspiradora me puso en contacto con el apasionante mundo de la robótica.

Por el clima enriquecedor en el que pude desarrollar este proyecto, agradezco a mis compañeros del Laboratorio de Robótica de la UPO, especialmente a David Alejo Teissière, Ignacio Pérez Hurtado de Mendoza, Álvaro Caballero Gómez, Macarena Mérida Floriano y Noé Pérez Higueras.

Todos ellos contribuyeron a que este trabajo se convirtiera en el primer paso de una rica experiencia, que aspiro ampliar especializándome a nivel de postgrado en el área de robótica.

Un agradecimiento especial a Victoria León Aranda, entrañable y solidaria amiga, que colaboró en la presentación de este trabajo; y a mi familia, por su amor y apoyo constante.

Por último, quiero dedicar este trabajo a Lisandro, que disfrutó jugando con el subsistema 01.



# ÍNDICE

Índice de tablas .....	9
Índice de figuras .....	13
Índice de fotografías .....	17
RESUMEN .....	19
1. INTRODUCCIÓN .....	21
2. ANÁLISIS .....	25
2.1. Definición del sistema .....	27
2.1.1. Alcance del sistema .....	27
2.1.2. Entorno tecnológico .....	27
2.2. Objetivos .....	28
2.3. Especificación de requisitos .....	29
2.3.1. Catálogo de requisitos .....	29
2.3.1.1. Requisitos funcionales .....	29
2.3.1.2. Requisitos no funcionales .....	33
2.3.2. Matriz de trazabilidad requisitos-objetivos .....	34
2.3.3. Especificación de casos de uso .....	36
2.3.3.1. Diagrama de casos de uso .....	36
2.3.3.2. Descripción de casos de uso .....	37
2.4. Análisis de los subsistemas .....	38
2.4.1. Análisis del subsistema 01: detección de un pulso sonoro .....	38
2.4.2. Análisis del subsistema 02: navegación del robot .....	39
2.5. Análisis del uso de <i>MATLAB Simulink Coder</i> con placas <i>Arduino</i> ..	42
2.6. Plan de pruebas .....	43
2.6.1. Entorno de pruebas .....	43
2.6.2. Alcance de las pruebas .....	43
2.6.2.1. Pruebas de odometría en línea recta .....	44
2.6.2.2. Pruebas bidireccionales de odometría en recorrido cuadrado <i>UBMark</i> .....	44
2.6.2.3. Pruebas de lecturas de los <i>sonars</i> .....	44
2.6.2.4. Pruebas de orientación del pulso sonoro .....	45
2.6.2.5. Pruebas de funcionamiento del apuntador .....	45
2.6.2.6. Pruebas de activación de los <i>LED</i> .....	45
2.6.3. Pruebas de aceptación del sistema .....	45
3. DISEÑO .....	47
3.1. Arquitectura del sistema .....	49
3.2. Componentes .....	51
3.3. Diseño del subsistema 01: detección de un pulso sonoro .....	60
3.3.1. Funciones del subsistema 01 .....	64
3.4. Diseño del subsistema 02: navegación del robot .....	67
3.4.1. Diseño de la odometría .....	67

3.4.2. Diseño del algoritmo de navegación .....	70
3.4.3. Estructuras de datos del subsistema 02 .....	76
3.4.4. Funciones del subsistema 02 .....	77
3.5. Diseño de los <i>drivers</i> de <i>Magabot</i> .....	89
3.5.1. Clase <i>Magabot</i> .....	90
3.5.2. Métodos del paquete <i>Magabot.h</i> .....	91
3.6. Diseño de los <i>drivers</i> con <i>Simulink</i> .....	94
3.6.1. Diseño de los <i>drivers</i> de control de los motores de <i>Magabot</i> ...	95
3.6.2. Diseño de los <i>drivers</i> de lectura de los <i>encoders</i> de <i>Magabot</i> ...	96
<b>4. IMPLEMENTACIÓN .....</b>	<b>99</b>
4.1. Construcción del <i>hardware</i> .....	101
4.2. Implementación del <i>software</i> .....	107
4.3. Realización de los <i>drivers</i> con <i>MATLAB Simulink</i> .....	111
4.3.1. Introducción al uso de <i>Simulink</i> con <i>Arduino</i> .....	111
4.3.2. Modelado de los <i>drivers</i> de control de los motores de <i>Magabot</i> con <i>Simulink</i> .....	116
4.3.3. Modelado de los <i>drivers</i> de lectura de los <i>encoders</i> de <i>Magabot</i> con <i>Simulink</i> .....	124
4.3.3.1. Subsistema <i>initializeEncoders</i> .....	125
4.3.3.2. Subsistema <i>encoderReadings</i> .....	126
4.3.3.3. Subsistema <i>resetEncoderState</i> .....	127
4.3.3.4. Subsistema <i>returnReadings</i> .....	128
4.4. Inclusión del código <i>C++</i> de los <i>drivers</i> en <i>MATLAB Simulink</i> ....	129
4.4.1. Inclusión en <i>Simulink</i> del código en <i>C++</i> de los <i>drivers</i> de control de los motores .....	129
4.4.2. Inclusión en <i>Simulink</i> del código en <i>C++</i> de los <i>drivers</i> de lectura de los <i>encoders</i> .....	133
4.5. <i>Profiling</i> del código en <i>Simulink</i> .....	135
<b>5. PRUEBAS .....</b>	<b>137</b>
5.1. Pruebas de odometría en línea recta .....	139
5.2. Pruebas bidireccionales de odometría en recorrido cuadrado <i>UBMark</i> .....	140
5.3. Pruebas de lecturas de los <i>sonars</i> .....	141
5.4. Pruebas de orientación del pulso sonoro .....	142
5.5. Pruebas de funcionamiento del apuntador .....	143
5.6. Pruebas de activación de los <i>LED</i> .....	144
5.7. Pruebas de aceptación del sistema .....	145
5.8. Demostración didáctica para alumnos de educación secundaria ....	145
<b>6. CONCLUSIONES Y DESARROLLOS FUTUROS .....</b>	<b>147</b>
<b>7. BIBLIOGRAFÍA .....</b>	<b>151</b>
<b>8. ANEXOS .....</b>	<b>155</b>
8.1. Anexo I: códigos .....	157
8.1.1. Código del subsistema 01: detección de un pulso sonoro - <i>SoundDetector.ino</i> .....	159
8.1.2. Código del subsistema 02: navegación del robot- <i>RobotNavigation.ino</i> .....	163
8.1.3. Códigos de los <i>drivers</i> de <i>Magabot</i> .....	177

8.1.3.1. Código del fichero <i>Magabot.h</i> .....	177
8.1.3.2. Código del fichero <i>Magabot.cpp</i> .....	179
8.2. Anexo II: plan de proyecto .....	183



# Índice de tablas

## 2. ANÁLISIS

### 2.2. Objetivos

OBJ-01: Navegación autónoma en dirección de un pulso sonoro ..... 28

OBJ-02: Implementación de los *drivers* de *Magabot* con *MATLAB Simulink Coder* ..... 28

### 2.3. Especificación de requisitos

#### 2.3.1. Catálogo de requisitos

##### 2.3.1.1. Requisitos funcionales

RF-01: Control de los motores ..... 29

RF-02: Lectura de los *encoders* ..... 29

RF-03: Lectura de los *sonars* ..... 30

RF-04: Lectura de los *bumpers* ..... 30

RF-05: Control de los *LED* ..... 30

RF-06: Control del apuntador ..... 30

RF-07: Detección de la orientación del pulso sonoro ..... 31

RF-08: Implementación de la odometría ..... 31

RF-09: Movimiento en dirección del origen de un pulso sonoro ..... 31

RF-10: Prevención de colisiones ..... 32

RF-11: Detención por colisiones ..... 32

RF-12: Detención por pulso sonoro ..... 32

##### 2.3.1.2. Requisitos no funcionales

RNF-01: Precisión de la odometría ..... 33

RNF-02: Indicadores de estado y *debugging* ..... 33

RNF 03: Comparar el tiempo de ejecución del código de los *drivers* en C++ con el código generado en *MATLAB Simulink* ..... 33

RNF-04: Crear una guía para el uso de *MATLAB Simulink* con *Arduino* ..... 34

Matriz de trazabilidad requisitos-objetivos ..... 35

### 2.3.3. Especificación de casos de uso

#### 2.3.3.2. Descripción de casos de uso

CU-01: Activar el robot mediante un pulso sonoro ..... 37

CU-02: Detener el robot mediante un pulso sonoro ..... 37

## 3. DISEÑO

### 3.2. Componentes

*Arduino Due* ..... 51

<i>Arduino Mega 2560</i> .....	52
<i>Magabot</i> .....	53
<i>Batería XCell Ni-MH Racing Pack 2500 mah 7,2 V</i> .....	54
<i>Micrófonos KY-037</i> .....	55
<i>Sensor de temperatura TMP36GZ</i> .....	56
<i>Cables Dupont</i> .....	57
<i>Placa solderless breadboard</i> .....	57
<i>Piezas Lego Technic</i> .....	58
<i>PC Acer Aspire E3-111-C5GL</i> .....	59
<b>3.3. Diseño del subsistema 01: detección de un pulso sonoro</b>	
<b>3.3.1. Funciones del subsistema 01</b>	
<i>void setup()</i> .....	64
<i>void loop()</i> .....	65
<i>void blink()</i> .....	66
<i>void set_sound_speed()</i> .....	66
<i>void reset()</i> .....	66
<b>3.4. Diseño del subsistema 02: navegación del robot</b>	
<b>3.4.3. Estructuras de datos del subsistema 02</b>	
<i>MapPoint</i> .....	76
<i>WorldPoint</i> .....	76
<i>Position</i> .....	76
<i>uint8_t direction_map [MAP_SIZE][MAP_SIZE]</i> .....	76
<i>uint8_t obstacles_map [MAP_SIZE][MAP_SIZE]</i> .....	77
<b>3.4.4. Funciones del subsistema 02</b>	
<i>void setup()</i> .....	77
<i>void stopRobot()</i> .....	78
<i>void setSonarsTransform()</i> .....	78
<i>void loop()</i> .....	79
<i>void checkLeds()</i> .....	80
<i>void checkSerial()</i> .....	80
<i>void checkBumpers()</i> .....	81
<i>void updateRobotPosition()</i> .....	81
<i>void checkBorder()</i> .....	82
<i>void detectObstacles()</i> .....	82
<i>WorldPoint getNextPoint()</i> .....	83
<i>bool checkLocalMinimum(MapPoint nextPoint)</i> .....	83
<i>void pointGoal()</i> .....	84
<i>void moveTo(MapPoint nextPoint)</i> .....	84
<i>uint16_t getForce(MapPoint p)</i> .....	84
<i>void movePointer(double angle)</i> .....	85
<i>void addObstacle(WorldPoint obstacle_worldPoint)</i> .....	85
<i>WorldPoint transformPoint(Position pos, WorldPoint p)</i> .....	85
<i>double euclideanDistance(WorldPoint p1, WorldPoint p2)</i> .....	86
<i>void setDirectionMap()</i> .....	86

<i>WorldPoint getWorldPoint(MapPoint mPoint)</i> .....	87
<i>MapPoint getMapPoint(WorldPoint wPoint)</i> .....	87
<i>void resetMapCenter()</i> .....	87
<i>void blinkRed()</i> .....	88
<i>void blinkGreen()</i> .....	88
<i>void blinkBlue()</i> .....	88
<i>void blinkPurple()</i> .....	89
<b>3.5. Diseño de los <i>drivers</i> de <i>Magabot</i></b>	
<b>3.5.1. Clase <i>Magabot</i></b>	
<i>Magabot</i> .....	91
<b>3.5.2. Métodos del paquete <i>Magabot.h</i></b>	
<i>Magabot::Magabot()</i> .....	91
<i>void Magabot::actuateMotors(int vel1, int vel2)</i> .....	92
<i>void Magabot::actuateLEDs(int Red, int Green, int Blue)</i> .....	92
<i>uint8_t Magabot::getSonarReadings(float* sonarReadings)</i> .....	93
<i>bool Magabot::readBumpers()</i> .....	93
<i>void Magabot::readClicks(float *wheelsRotation)</i> .....	94
<b>4. IMPLEMENTACIÓN</b>	
<b>4.5. Profiling del código en <i>Simulink</i></b>	
<i>Profiling</i> de los <i>drivers</i> de los motores .....	136
<i>Profiling</i> de los <i>drivers</i> de los <i>encoders</i> .....	136
<b>5. PRUEBAS</b>	
<b>5.1. Pruebas de odometría en línea recta</b>	
Pruebas de odometría en línea recta .....	139
<b>5.2. Pruebas bidireccionales de odometría en recorrido cuadrado</b>	
<b><i>UBMark</i></b>	
Pruebas de odometría en recorrido cuadrado de 2m x 2m	
en sentido horario .....	140
Pruebas de odometría en recorrido cuadrado de 2m x 2m	
en sentido antihorario .....	140
<b>5.3. Pruebas de lecturas de los <i>sonars</i></b>	
Pruebas de lecturas de los <i>sonars</i> .....	141
<b>5.4. Pruebas de orientación del pulso sonoro</b>	
Pruebas de orientación del pulso sonoro .....	142
<b>5.5. Pruebas de funcionamiento del apuntador</b>	
Pruebas de funcionamiento del apuntador .....	143
<b>5.6. Pruebas de activación de los <i>LED</i></b>	
Pruebas de activación de los <i>LED</i> .....	144



# Índice de figuras

Figura 1: Diagrama de casos de uso .....	36
Figura 2: Diagrama de componentes .....	38
Figura 3: Navegación por campos potenciales .....	40
Figura 4: Algoritmo de navegación <i>Wavefront Planner</i> .....	41
Figura 5: Test <i>UBMark</i> .....	44
Figura 6: Esquema de conexiones <i>hardware</i> .....	50
Figura 7: Disposición de los micrófonos en la parte superior del robot	60
Figura 8: Uso de los micrófonos según el ángulo de origen del pulso sonoro .....	61
Figura 9: Diagrama de flujo del subsistema 01 .....	63
Figura 10: Configuración diferencial de <i>Magabot</i> .....	67
Figura 11: Estructura interna de un <i>encoder</i> .....	68
Figura 12: Mediciones del desplazamiento de <i>Magabot</i> .....	69
Figura 13: Matriz de dirección hacia el objetivo .....	71
Figura 14: Matrices de dirección y de obstáculos .....	72
Figura 15: Desplazamiento de los mapas de orientación y de obstáculos .....	73
Figura 16: Limitaciones físicas de los <i>sonars</i> .....	74
Figura 17: Diagrama de flujo del subsistema 02: navegación del robot .....	75
Figura 18: Clases del paquete de <i>drivers</i> de <i>Magabot</i> realizados por los fabricantes .....	90
Figura 19: Diseño de alto nivel de los modelos de los <i>drivers</i> de <i>Magabot</i> con <i>MATLAB Simulink</i> .....	95
Figura 20: Diseño del modelo <i>motorsSimulink</i> .....	96
Figura 21: Diseño del modelo <i>motorsSimulinkRef</i> .....	96
Figura 22: Diseño del modelo <i>encodersSimulink</i> .....	97
Figura 23: Diseño del modelo <i>encodersSimulinkRef</i> .....	97
Figura 24: Lecturas de los tres micrófonos al detectar un pulso sonoro .....	108
Figura 25: Representación gráfica de la matriz de dirección .....	109
Figura 26: Representación gráfica de la matriz de obstáculos .....	110
Figura 27: Suma de las matrices de dirección y obstáculos .....	110
Figura 28: Suma de las matrices tras la detección de un mínimo local .....	111
Figura 29: Menú <i>Home</i> de <i>MATLAB</i> .....	112

Figura 30: Paquetes de <i>MATLAB</i> y <i>Simulink</i> para placas <i>Arduino</i> ...	112
Figura 31: Opción <i>Blank Model</i> .....	113
Figura 32: Campo <i>Hardware Board</i> .....	113
Figura 33: Configuración del <i>Solver</i> .....	114
Figura 34: Opción <i>Library browser</i> .....	114
Figura 35: <i>Simulink/Commonly Used Blocks</i> .....	115
Figura 36: Bloques del paquete de <i>drivers</i> de <i>Simulink</i> para <i>Arduino</i> .....	116
Figura 37: Modelo del sistema <i>motorsSimulink</i> .....	116
Figura 38: Parámetros del generador de pulsos .....	117
Figura 39: Opción <i>Subsystem &amp; Model Reference</i> .....	117
Figura 40: Conversión de un subsistema a modelo referenciado .....	118
Figura 41: Opción <i>Block Parameters (ModelReference)</i> .....	118
Figura 42: Parámetros de un sistema referenciado .....	119
Figura 43: Opción <i>Model Referencing</i> .....	119
Figura 44: Configuración de <i>Code Generation</i> .....	120
Figura 45: Modelo referenciado <i>motorsSimulinkRef</i> .....	120
Figura 46: Configuración de <i>Code Generation</i> .....	121
Figura 47: <i>Set Objectives - Code Generation Adviser</i> .....	121
Figura 48: Opción <i>Create code generation report</i> .....	122
Figura 49: Configuración <i>Code Generation -&gt; Interface</i> .....	122
Figura 50: Configuración <i>Code Profiling</i> .....	123
Figura 51: Modalidad de ejecución normal .....	123
Figura 52: Modelo <i>encodersSimulink</i> .....	124
Figura 53: Modelo <i>encodersSimulinkRef</i> .....	124
Figura 54: Parámetros del bloque <i>Pulse Generator</i> .....	125
Figura 55: Subsistema <i>initilizeEncoders</i> .....	126
Figura 56: Subsistema <i>encoderReadings</i> .....	127
Figura 57: Función <i>displacement</i> .....	127
Figura 58: Subsistema <i>resetEncoderState</i> .....	128
Figura 59: Subsistema <i>returnReadings</i> .....	128
Figura 60: Diseño del modelo <i>motorsArduino</i> .....	129
Figura 61: Diseño del modelo <i>motorsArduinoRef</i> .....	130
Figura 62: Campo <i>S-function name</i> .....	130
Figura 63: Definición de puertos de entrada de <i>moveMotorsSfunction</i> .....	131
Figura 64: Definición del tipo de atributos de <i>moveMotorsSfunction</i> .....	131
Figura 65: Declaración de paquetes de <i>moveMotorsSfunction</i> .....	132
Figura 66: Código en lenguaje <i>C</i> de <i>moveMotorsSfunction</i> .....	132
Figura 67: Edición de <i>moveMotorsSfunction_wrapper.c</i> .....	132
Figura 68: Definición de puertos de salida de <i>encodersSfunction</i> ....	133

Figura 69: Definición del tipo de atributos de <i>encodersSfunction</i> ....	134
Figura 70: Código en lenguaje C de <i>encodersSfunction</i> .....	134
Figura 71: Edición de <i>encodersSfunction_wrapper.c</i> .....	134
Figura 72: Tabla resultado del <i>profiling</i> .....	135



# Índice de fotografías

Fotografía 1: Laboratorio de robótica de la Universidad Pablo de Olavide, Sevilla .....	43
Fotografía 2: Placa <i>Arduino Due</i> .....	51
Fotografía 3: Placa <i>Arduino Mega 2560</i> .....	52
Fotografía 4: Plataforma <i>Magabot</i> .....	53
Fotografía 5: <i>Magabot shield</i> .....	54
Fotografía 6: Batería <i>XCell Ni-MH Racing Pack 2500 mah 7,2 V</i> ..	54
Fotografía 7: Módulo micrófono <i>KY-037</i> .....	55
Fotografía 8: Sensor de temperatura <i>TMP36</i> .....	56
Fotografía 9: Cables Dupont .....	57
Fotografía 10: Placa <i>solderless breadboard</i> .....	57
Fotografía 11: Piezas de <i>Lego Technic</i> .....	58
Fotografía 12: <i>PC Acer Aspire E3-111-C5GL</i> .....	59
Fotografía 13: <i>Magabot</i> con soportes .....	101
Fotografía 14: Estructura superior del robot .....	102
Fotografía 15: Detalle del micrófono enganchado en la estructura de <i>Lego</i> .....	103
Fotografía 16: Soportes en goma espuma .....	103
Fotografía 17: Apuntador .....	104
Fotografía 18: Servo del apuntador .....	104
Fotografía 19: Engranajes del apuntador .....	105
Fotografía 20: <i>Ardino Due</i> y <i>breadboard</i> .....	105
Fotografía 21: Conexión entre placa <i>Arduino Mega 2560</i> y <i>shield</i> ..	106
Fotografía 22: Instalación completa del robot .....	107
Fotografía 23: Demostración del funcionamiento del robot frente a alumnos del IES Virgen de Valme, I .....	146
Fotografía 24: Demostración del funcionamiento del robot frente a alumnos del IES Virgen de Valme, II .....	146



# RESUMEN

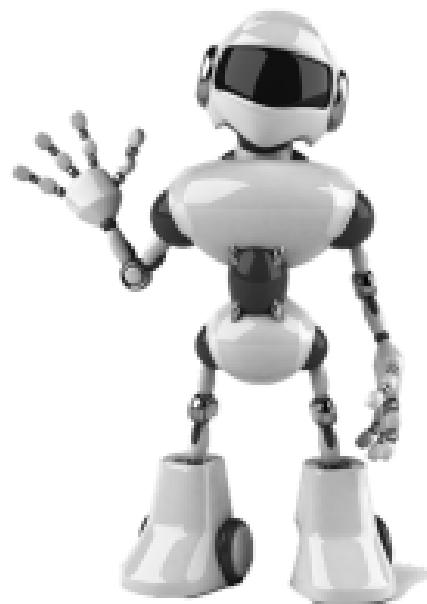
---

En la primera parte de este trabajo presentamos un sistema robótico de navegación autónoma en interiores, implementado por un algoritmo de tipo reactivo, que le permite al robot desplazarse en dirección de un pulso sonoro, esquivando obstáculos. Para el desarrollo de este sistema de navegación utilizamos la plataforma móvil *Magabot*, sobre la que hemos creado dos subsistemas implementados en dos placas *Arduino –Arduino Due* y *Arduino Mega 2560*, conectadas entre sí.

En lo que respecta al sistema robótico de control de bajo nivel que diseñamos para la plataforma *Magabot*, empleamos el software *MATLAB Simulink* para crear modelos que generen el código necesario para ejecutar esos *drivers* en la placa *Arduino Mega 2560*. Los modelos de *Simulink* se utilizarán, asimismo, para comparar los tiempos de ejecución en el *hardware* del código escrito en *C++* del paquete *Magabot.h* y del código generado por *Simulink Coder*.



# Introducción





# 1 INTRODUCCIÓN

---

El desarrollo científico y tecnológico de las últimas décadas ha derivado en el actual auge de los estudios de robótica. Al ritmo de fuertes debates – entre quienes demonizan a los robots augurando un futuro catastrófico para la humanidad y entre quienes aplauden su desarrollo pronosticando una vida más fácil para los seres humanos–, estos estudios avanzan a pasos agigantados.

Actualmente, podemos clasificar a los robots en tres grandes categorías: operados, automáticos y autónomos. Los robots operados, que se usan principalmente para potenciar las capacidades humanas en la realización de un amplio abanico de tareas, se caracterizan por su incapacidad de funcionar de forma independiente de los operadores que los manejan. Controlados a distancia, algunos robots operados se utilizan, por ejemplo, para desactivar artefactos explosivos, evitando que los artificieros pongan en peligro su vida. Dentro de este tipo de robots, podemos encontrar también a robots quirúrgicos, que además de mejorar la precisión del cirujano posibilitan que las operaciones sean menos invasivas para los pacientes.

A diferencia de este primer tipo de robots, los robots automáticos no necesitan de operadores que controlen su funcionamiento. Los robots automáticos, que se caracterizan por realizar únicamente tareas predeterminadas en un entorno estructurado y bien definido, son ampliamente utilizados en la industria, por ejemplo, en los procesos automatizados de las cadenas de montaje.

En la tercera categoría encontramos a los robots autónomos. Al igual que los automáticos, los robots autónomos pueden operar sin la mediación de un operador, pero se diferencian de los primeros por disponer de sistemas inteligentes que posibilitan al robot operar en un entorno no estructurado y tomar decisiones para llevar a cabo sus tareas sin la necesidad de intervención humana. La navegación de vehículos autónomos es uno de los campos de la robótica que más esfuerzos ha concentrado en los últimos años. En algunas ciudades de Estados Unidos, por ejemplo, podemos ver circular vehículos no tripulados capaces de moverse de forma autónoma, respetando las normas de tránsito y evitando colisiones con innumerables tipos de obstáculos. También en nuestros hogares se está volviendo habitual la presencia de este tipo de robot. Un ejemplo muy conocido es *Roomba*, el robot aspirador de *iRobot*, capaz de recorrer espacios interiores limpiando el suelo y de volver autónomamente a la estación de carga de batería.

Dentro de esta tercera categorización enmarcamos nuestro trabajo, que tiene como primer objetivo crear un sistema robótico de navegación

autónoma, guiado por un pulso sonoro. Para montar este sistema utilizaremos la plataforma móvil *Magabot* –fabricada en conjunto por las empresas portuguesas *IDMind* y *Artica*–, y dos tipos diferentes de placas *Arduino*: una *Arduino Due* y una *Arduino Mega 2560*, que funcionarán conectadas entre sí.

Para que el robot pueda navegar autónomamente, implementaremos un algoritmo de navegación que permita el desplazamiento del robot hacia la dirección del pulso sonoro y que, al mismo tiempo, le posibilite evitar obstáculos durante el trayecto.

Existen principalmente dos tipos de planificación de caminos dependiendo del conocimiento del entorno: la planificación global, que sirve para encontrar el mejor camino en un entorno conocido, y la planificación local o reactiva que, al no disponer de información previa sobre el entorno, utiliza los datos recibidos por los sensores para planificar un camino libre de obstáculos. En los sistemas de navegación autónomos se suelen utilizar ambos planificadores de forma conjunta. De esa manera, mientras que el primer tipo de planificador define una trayectoria a seguir en un mapa conocido, el segundo evita la colisión con los obstáculos que detecta el robot durante su desplazamiento.

En el caso de nuestro robot, que no dispone de un mapa para su desplazamiento, toda la información sobre el entorno –el posicionamiento del objetivo y la presencia de obstáculos– proviene de los sensores. Dada la carencia de conocimiento previo del ambiente, implementaremos un algoritmo de navegación de tipo reactivo.

El siguiente objetivo de este trabajo será emplear la metodología de diseño basado en modelo para generar el *software* de control de bajo nivel de *Magabot*. Con tal fin, utilizaremos el entorno de trabajo *MATLAB Simulink* para generar los modelos que capturen las funcionalidades requeridas. Estos modelos permitirán generar, posteriormente, el código de los *drivers* de *Magabot* que deben ser ejecutados en la placa *Arduino Mega 2560*. Sobre la base del código generado para esos modelos, mediante los *Coders* de *Simulink*, llevaremos a cabo una comparación de los tiempos de ejecución con el *software* de control escrito en *C++* (paquete de *drivers Magabot.h*).

Durante el desarrollo de esta segunda parte del proyecto nos ocuparemos, además, de documentar el flujo de trabajo realizado con este *software*, con el propósito de elaborar una guía que pueda servir a quienes que estén interesados en integrar las funcionalidades de *MATLAB Simulink* y *Arduino*.

# Análisis





## 2 ANÁLISIS

---

En este apartado presentamos el alcance y los objetivos del proyecto, así como también, los requisitos funcionales y no funcionales. Luego, describimos la arquitectura del sistema e identificamos a los subsistemas que lo integran. Asimismo, esbozamos el estado del arte sobre nuestro tema de investigación, y definimos las categorías de análisis principales de nuestro proyecto. Por último, detallamos nuestro plan de pruebas.

### 2.1 DEFINICIÓN DEL SISTEMA

En esta sección definimos el alcance del sistema y el entorno tecnológico que disponemos para su implementación.

#### 2.1.1 Alcance del sistema

El objetivo principal de este trabajo es crear un robot de navegación autónomo capaz de detectar el ángulo de origen de un pulso sonoro –como por ejemplo una palmada– y moverse en esa dirección. El robot debe desplazarse autónomamente en un terreno liso y sin desniveles, evitando la colisión con obstáculos estáticos. Sin embargo, en el caso de que se produzca una colisión, los motores del robot deben detenerse.

En una segunda parte de este trabajo, se procede a utilizar el *software MATLAB*, con las extensiones *Embedded Coder* y *Simulink*, para generar código de los *drivers* de los sensores y actuadores del robot, y analizar el rendimiento del código autogenerado. Este segundo objetivo tiene la finalidad de explorar las funcionalidades avanzadas de *MATLAB* para su uso con las placas *Arduino*.

#### 2.1.2 Entorno tecnológico

Para realizar el trabajo se dispone de la plataforma de *open hardware Magabot*, producida por las empresas portuguesas *IDMind* y *Artica*. *Magabot* es un robot móvil –con tracción diferencial, dos motores DC, dos *encoders*, tres sensores *IR*, dos *bumpers*, cinco *sonars* y tres *LED RGB*– basado en *Arduino Uno*. Para el desarrollo de este proyecto, esta última placa será sustituida por una *Arduino Mega 2560*, que posee una memoria de *SRAM* de *8KB* y cuatro puertos de comunicación serie, a diferencia del modelo *Uno*, que dispone de un único puerto serie y una memoria de apenas *2KB*.

Al *Magabot* se añadirán micrófonos que posibiliten la localización de la fuente sonora. La lectura de los micrófonos se realizará con una placa *Arduino Due*, que posee una frecuencia de *clock* y una velocidad de lectura analógica que permiten obtener lecturas precisas del tiempo de llegada de la onda sonora.

## 2.2 OBJETIVOS

En las siguientes tablas se describen los objetivos principales del trabajo.

<b>OBJ–01: Navegación autónoma en dirección de un pulso sonoro</b>	
<b>Descripción</b>	Crear un sistema robótico de navegación autónoma capaz de detectar el origen de un pulso sonoro y moverse hacia esa dirección. En su trayectoria, el robot debe evitar colisiones con obstáculos estáticos.
<b>Importancia</b>	Alta
<b>Comentarios</b>	El algoritmo de navegación, la localización del sonido y los <i>drivers</i> , se implementan en placas de desarrollo <i>Arduino Due</i> y <i>Arduino Mega 2560</i> .

<b>OBJ–02: Implementación de los <i>drivers</i> de <i>Magabot</i> con <i>MATLAB Simulink Coder</i></b>	
<b>Descripción</b>	<ul style="list-style-type: none"> <li>- Crear modelos de <i>MATLAB Simulink</i> que generen el código necesario para la ejecución de los <i>drivers</i> de <i>Magabot</i>.</li> <li>- Realizar un análisis que compare el rendimiento del código de los <i>drivers</i> en <i>C++</i>, con los códigos generados con la herramienta <i>Embedded Coder</i> de <i>MATLAB</i>.</li> </ul>
<b>Importancia</b>	Alta
<b>Comentarios</b>	Considerando que esta propuesta tiene la finalidad de investigar las funcionalidades avanzadas de <i>MATLAB</i> para la programación de placas <i>Arduino</i> , se creará una guía para futuros proyectos que pretendan integrar las funcionalidades de <i>Simulink</i> y <i>Arduino</i> .

## **2.3 ESPECIFICACIÓN DE REQUISITOS**

En este apartado describiremos los requisitos funcionales y no funcionales del proyecto. Incluiremos, asimismo, la matriz de trazabilidad de requisitos y objetivos. Por último, identificaremos los casos de uso del sistema.

### **2.3.1 Catálogo de requisitos**

En las siguientes tablas se detallan los requisitos funcionales y no funcionales del proyecto.

#### ***2.3.1.1 Requisitos funcionales***

<b>RF-01: Control de los motores</b>	
<b>Objetivos asociados</b>	OBJ-01: Navegación autónoma en dirección de un pulso sonoro. OBJ-02: Implementación de <i>drivers</i> de <i>Magabot</i> con <i>MATLAB Simulink Coder</i> .
<b>Descripción</b>	Se implementan los <i>drivers</i> de <i>Magabot</i> para controlar el movimiento de los dos motores <i>DC</i> del robot.
<b>Comentarios</b>	Además, se implementan los <i>drivers</i> de control de los motores usando las herramientas <i>Embedded Coder</i> y <i>Simulink</i> de <i>MATLAB</i> .

<b>RF-02: Lectura de los encoders</b>	
<b>Objetivos asociados</b>	OBJ-01: Navegación autónoma en dirección de un pulso sonoro. OBJ-02: Implementación de <i>drivers</i> de <i>Magabot</i> con <i>MATLAB Simulink Coder</i> .
<b>Descripción</b>	Se implementan los <i>drivers</i> de <i>Magabot</i> para leer los datos de los dos <i>encoders</i> , que se encuentran posicionados en los ejes de las ruedas motrices del robot.
<b>Comentarios</b>	Además, se implementan los <i>drivers</i> de activación y lectura de los <i>encoders</i> usando las herramientas <i>Embedded Coder</i> y <i>Simulink</i> de <i>MATLAB</i> .

RF-03: Lectura de los <i>sonars</i>	
<b>Objetivos asociados</b>	OBJ-01: Navegación autónoma en dirección de un pulso sonoro.
<b>Descripción</b>	Se implementan los <i>drivers</i> de <i>Magabot</i> para activar los cinco <i>sonars</i> y leer sus mediciones.
<b>Comentarios</b>	--

RF-04: Lectura de los <i>bumpers</i>	
<b>Objetivos asociados</b>	OBJ-01: Navegación autónoma en dirección de un pulso sonoro.
<b>Descripción</b>	Se implementan los <i>drivers</i> de <i>Magabot</i> para leer los dos <i>bumpers</i> , que se encuentran en la parte frontal del robot.
<b>Comentarios</b>	--

RF-05: Control de los <i>LED</i>	
<b>Objetivos asociados</b>	OBJ-01: Navegación autónoma en dirección de un pulso sonoro.
<b>Descripción</b>	Se implementan los <i>drivers</i> de <i>Magabot</i> para activar los tres <i>LED RGB</i> del robot.
<b>Comentarios</b>	--

RF-06: Control del apuntador	
<b>Objetivos asociados</b>	OBJ-01: Navegación autónoma en dirección de un pulso sonoro.
<b>Descripción</b>	Se instala un apuntador en la parte superior de <i>Magabot</i> para señalar el lugar de origen del pulso sonoro y, además, indicar la dirección hacia la cual se debe dirigir el robot.
<b>Comentarios</b>	--

<b>RF-07: Detección de la orientación del pulso sonoro</b>	
<b>Objetivos asociados</b>	OBJ-01: Navegación autónoma en dirección de un pulso sonoro.
<b>Descripción</b>	Se utilizan tres micrófonos para calcular la dirección de un pulso sonoro en un plano bidimensional, sobre la base de la diferencia del tiempo de llegada del frente de onda sonora a cada uno de los micrófonos.
<b>Comentarios</b>	--

<b>RF-08: Implementación de la odometría</b>	
<b>Objetivos asociados</b>	OBJ-01: Navegación autónoma en dirección de un pulso sonoro.
<b>Descripción</b>	Se calcula el desplazamiento del robot en el sistema de referencia.
<b>Comentarios</b>	La navegación utilizando la odometría suele producir muchos errores. Para minimizarlos, es necesario realizar mediciones precisas de las dimensiones de las ruedas y de la distancia entre ambas. Además, el robot debe realizar movimientos suaves para evitar el deslizamiento de las ruedas, que es una de las principales fuentes de error.

<b>RF-09: Movimiento en dirección del origen de un pulso sonoro</b>	
<b>Objetivos asociados</b>	OBJ-01: Navegación autónoma en dirección de un pulso sonoro.
<b>Descripción</b>	Se implementa un algoritmo de navegación autónoma para que el robot se mueva en dirección de un pulso sonoro.
<b>Comentarios</b>	En la selección del algoritmo de navegación se deben tener en cuenta las limitaciones <i>hardware</i> de la placa <i>Arduino</i> .

RF-10: Prevención de colisiones	
<b>Objetivos asociados</b>	OBJ-01: Navegación autónoma en dirección de un pulso sonoro.
<b>Descripción</b>	Se usan las lecturas de los <i>sonars</i> para identificar el posicionamiento de obstáculos, y se implementa un algoritmo para evitar las colisiones con objetos estáticos.
<b>Comentarios</b>	En la selección del algoritmo de prevención de colisiones se deben tener en cuenta las limitaciones <i>hardware</i> de la placa <i>Arduino</i> .

RF-11: Detención por colisiones	
<b>Objetivos asociados</b>	OBJ-01: Navegación autónoma en dirección de un pulso sonoro.
<b>Descripción</b>	El <i>Magabot</i> dispone de dos <i>bumpers</i> en la parte frontal. En caso de colisión del robot, éstos se activan y el sistema debe detenerse.
<b>Comentarios</b>	--

RF-12: Detención por pulso sonoro	
<b>Objetivos asociados</b>	OBJ-01: Navegación autónoma en dirección de un pulso sonoro.
<b>Descripción</b>	Si el operador emite un pulso sonoro mientras el robot está en movimiento, éste se tiene que detener.
<b>Comentarios</b>	Se puede frenar el robot dando una palmada, y éste queda en espera de otro pulso sonoro para volver a moverse.

### 2.3.1.2 Requisitos no funcionales

RNF-01: Precisión de la odometría	
<b>Objetivos asociados</b>	OBJ-01: Navegación autónoma en dirección de un pulso sonoro.
<b>Descripción</b>	Es necesario reducir al mínimo los errores en la estimación del desplazamiento del robot.
<b>Comentarios</b>	Se realizarán pruebas de las lecturas de los <i>encoders</i> para calibrar las mediciones de los desplazamientos.

RNF-02: Indicadores de estado y <i>debugging</i>	
<b>Objetivos asociados</b>	OBJ-01: Navegación autónoma en dirección de un pulso sonoro.
<b>Descripción</b>	El <i>Magabot</i> dispone de <i>LEDs</i> que se utilizan para mostrar estados del sistema. Se conecta un ordenador con la placa <i>Arduino</i> para realizar el <i>debugging</i> y obtener información durante la ejecución del código.
<b>Comentarios</b>	--

RNF-03: Comparar el tiempo de ejecución del código de los drivers en C++ con el código generado en <i>MATLAB Simulink</i>	
<b>Objetivos asociados</b>	OBJ-02: Implementación de los <i>drivers</i> de <i>Magabot</i> con <i>MATLAB Simulink Coder</i> .
<b>Descripción</b>	Se comparan los tiempos de ejecución de las funciones de los <i>drivers</i> escritos en C++, con el tiempo de ejecución del código generado con <i>Simulink Coder</i> , a partir de los diseños basados en modelo.
<b>Comentarios</b>	--

<b>RNF-04: Crear una guía para el uso de <i>MATLAB Simulink</i> con <i>Arduino</i></b>	
<b>Objetivos asociados</b>	OBJ-02: Implementación de los <i>drivers</i> de <i>Magabot</i> con <i>MATLAB Simulink Coder</i> .
<b>Descripción</b>	Se escribe una guía que detalle los pasos seguidos para implementar los <i>drivers</i> del <i>Magabot</i> con <i>MATLAB Simulink Coder</i> .
<b>Comentarios</b>	--

### 2.3.2 Matriz de trazabilidad requisitos-objetivos

En la siguiente tabla se muestra la matriz de trazabilidad entre los requisitos funcionales como no funcionales, y los objetivos del trabajo.

<b>Matriz de trazabilidad requisitos- objetivos-</b>	<b>OBJ–01:</b> Navegación autónoma en dirección de un pulso sonoro	<b>OBJ–02:</b> Implementación de los drivers de <i>Magabot</i> con <i>MATLAB Simulink Coder</i>
RF-01 Control de los motores		
RF-02 Lectura de los <i>encoders</i>		
RF-03 Lectura de los <i>sonars</i>		
RF-04 Lectura de los <i>bumpers</i>		
RF-05 Control de los <i>LED</i>		
RF-06 Control del apuntador		
RF-07 Detección de la orientación del pulso sonoro		
RF-08 Implementación de la odometría		
RF-09 Movimiento en dirección del origen de un pulso sonoro		
RF-10 Prevención de colisiones		
RF-11 Detención por colisiones		
RF-12 Detención por pulso sonoro		
RNF-01 Precisión de la odometría		
RNF-02 Indicadores de estado y <i>debugging</i>		
RNF-03 Comparar el tiempo de ejecución del código de los drivers en C++ con el código generado en <i>MATLAB Simulink</i>		
RNF-04 Crear una guía para el uso de <i>MATLAB Simulink</i> con Arduino		

### 2.3.3 Especificación de casos de uso

A continuación, se especifican los casos de uso del robot, que son ejecutados por un único actor: el operador.

#### 2.3.3.1 *Diagrama de casos de uso*

En la Figura 1 representamos en forma gráfica el diagrama de casos de uso de nuestro sistema.

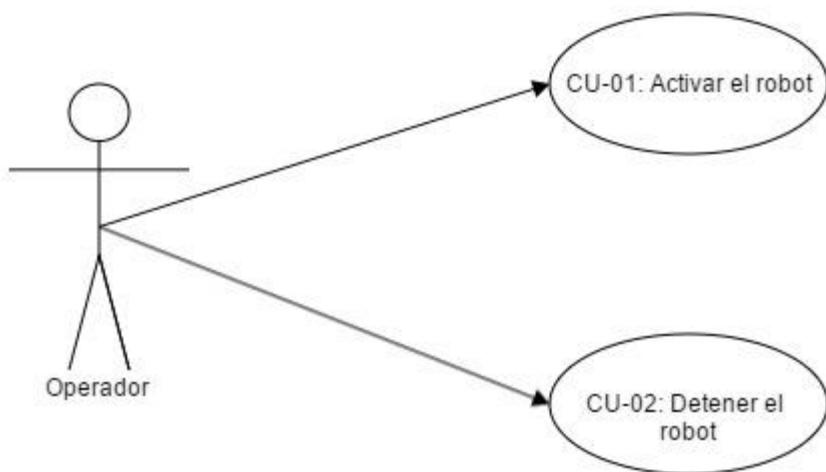


Figura 1: Diagrama de casos de uso

### 2.3.3.2 Descripción de casos de uso

En las siguientes tablas describimos los dos casos de uso que vamos a implementar en nuestro sistema.

CU-01: Activar el robot mediante un pulso sonoro		
<b>Actor</b>	Operador	
<b>Precondición</b>	El robot tiene que encontrarse detenido	
<b>Postcondición</b>	El robot se mueve en dirección de un pulso sonoro	
<b>Descripción</b>	El operador emite un sonido fuerte como, por ejemplo, una palmada, y el robot se mueve en dirección de ese pulso sonoro	
<b>Flujo normal</b>	<b>Paso</b>	Acción
	1	El operador emite un pulso sonoro fuerte
	2	El robot calcula la dirección del pulso sonoro
	3	El apuntador indica la dirección del pulso sonoro
	4	El robot se mueve en dirección del pulso sonoro evitando los obstáculos

CU-02: Detener el robot mediante un pulso sonoro		
<b>Actor</b>	Operador	
<b>Precondición</b>	El robot tiene que encontrarse en movimiento	
<b>Postcondición</b>	El robot se detiene	
<b>Descripción</b>	El operador emite un sonido fuerte como, por ejemplo, una palmada, y el robot se detiene	
<b>Flujo normal</b>	<b>Paso</b>	Acción
	1	El operador emite un pulso sonoro fuerte
	2	El robot se detiene y espera un nuevo pulso sonoro para volver a moverse

## 2.4 ANÁLISIS DE LOS SUBSISTEMAS

El sistema se divide en dos subsistemas. Mientras que el primer subsistema tiene la función de detectar el origen del pulso sonoro, el segundo se encarga de la navegación del robot.

En el siguiente diagrama de componentes se muestra la arquitectura del sistema (Figura 2).

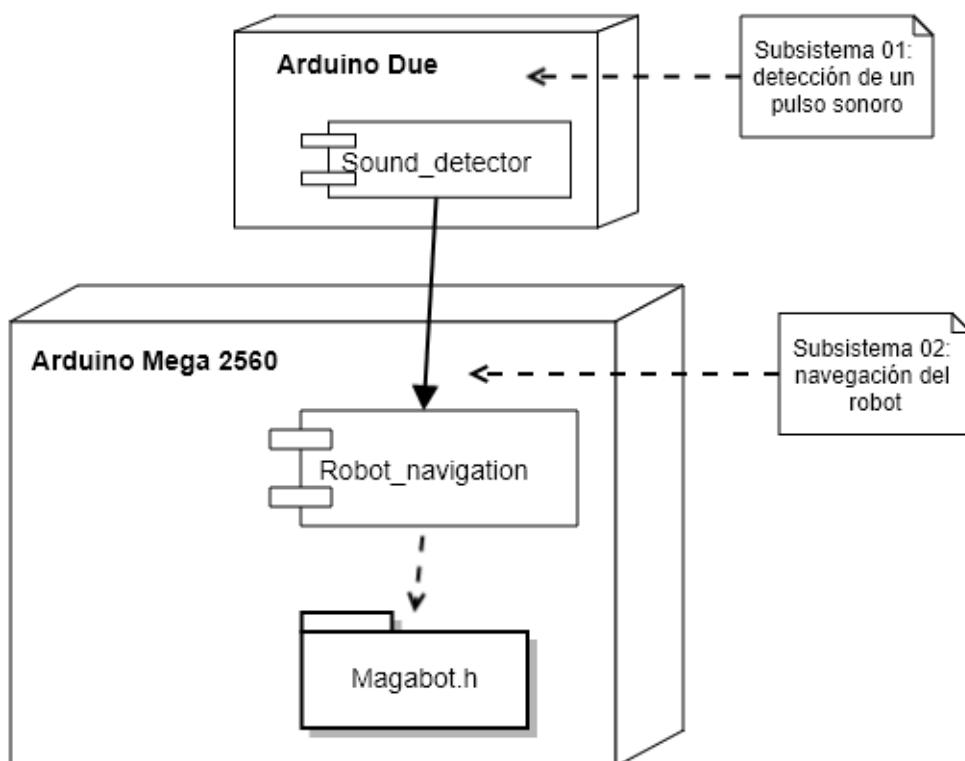


Figura 2: Diagrama de componentes

### 2.4.1 Análisis del subsistema 01: detección de un pulso sonoro

El cerebro humano, para calcular la dirección de proveniencia de un sonido, analiza la diferencia del tiempo interaural (ITD), la diferencia de fase interaural (IPD) y la diferencia de intensidad interaural (IID) (Plack, 2005:173-192). ITD e IPD se deben a la diferencia del tiempo de llegada del

sonido a cada oído. La diferencia de intensidad se usa para detectar principalmente la orientación de sonidos agudos o muy cercanos al oyente.

Todas las técnicas aplicadas a la detección del origen de un sonido se caracterizan por requerir más de un micrófono. Nuestro robot tiene que identificar sólo el ángulo acimutal del sonido, es decir, el ángulo en el plano horizontal. Para ello, se utilizan tres micrófonos y se calcula la dirección del sonido sobre la base de la diferencia del tiempo de llegada del frente de onda sonora a cada uno de los micrófonos. Esta técnica usa el ITD, lo que es equivalente al IPD para el primer ciclo de la onda sonora.

Una vez que se ha calculado el ángulo de origen de un pulso sonoro, este valor es comunicado al subsistema 02, que es el encargado de la navegación autónoma del robot.

#### 2.4.2 Análisis del subsistema 02: navegación del robot

El subsistema 02 es el encargado de mover el robot en dirección de un pulso sonoro. El robot tiene que navegar de forma autónoma sin disponer de ningún tipo de mapa del entorno. El robot implementa un algoritmo de tipo reactivo para alcanzar su objetivo, y utiliza cinco *sonars* para detectar la presencia de obstáculos en su trayecto.

Para evitar obstáculos en un entorno desconocido, suelen ser muy utilizados los llamados *algoritmos de rodeo reactivo* o *bug algorithms*. Si bien estos algoritmos tienen numerosas variantes, en general su comportamiento consiste en mover el robot en línea recta hacia un objetivo determinado. En caso de encontrar algún obstáculo en su trayecto, el robot lo rodea hasta encontrar un nuevo camino libre. Estos tipos de algoritmos de navegación reactivos tienen la ventaja de ser muy ligeros computacionalmente, pero en ciertos casos no resultan adecuados para que el robot alcance su objetivo.

Un algoritmo muy eficaz para eludir obstáculos es el llamado *algoritmo de navegación por campos potenciales*. Para entender su funcionamiento podemos suponer que el robot es una partícula con una carga eléctrica igual a la de los obstáculos, mientras que el objetivo tiene una carga opuesta. De esa manera, el robot se movería atraído por la carga opuesta del objetivo, y en su trayecto lograría evitar los obstáculos, al ser empujado lejos de estos por la carga de igual signo. La dirección y la velocidad del robot son dadas por el vector resultante de la suma de las fuerzas atractivas y repulsivas presentes en cada punto del mapa. En este tipo de algoritmos, el robot encuentra su camino realizando un descenso del gradiente hasta encontrar el punto mínimo, que es el objetivo de la navegación (Figura 3).

El principal inconveniente de los algoritmos basados en campos potenciales es la presencia de mínimos locales, es decir, de puntos en el mapa que tienen un potencial nulo, ya que las fuerzas repulsivas contrastan a la fuerza atractiva. Cuando el robot se encuentra en un mínimo local, queda detenido. Para evadir a los mínimos locales, se pueden utilizar distintas técnicas, como seguir un punto aleatorio durante un tiempo, volver hacia atrás, bordear a los obstáculos o añadir campos de fuerzas virtuales.

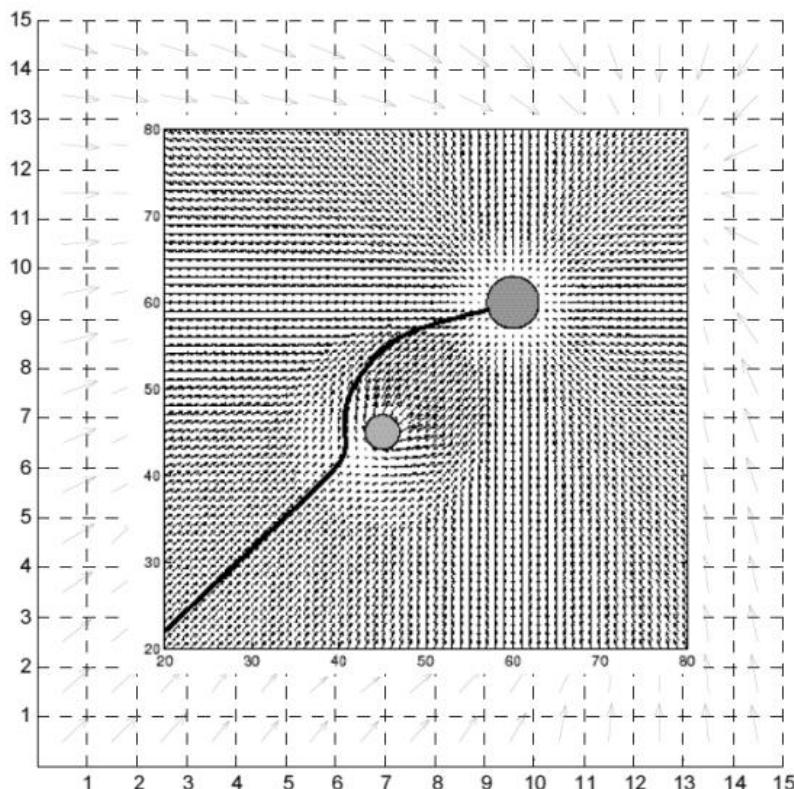


Figura 3: Navegación por campos potenciales.

Fuente: Merino Cabañas, Luís y Heredia Benot, Guillermo. “Robot Navigation, reactive methods. Apuntes de la materia Robótica y Visión Artificial del Grado en Ingeniería Informática de la Universidad Pablo de Olavide”, Año Académico 2016/2017, p. 21.

En los casos en los que el entorno es conocido, se utilizan algoritmos de navegación planificados como, por ejemplo, el llamado *Wavefront Planner* (Reyes *et al*, 2015:1915). Este algoritmo divide el mapa conocido en celdas. Al proponer un nuevo destino para el robot, el algoritmo asigna a la celda objetivo el valor cero. A las celdas adyacentes al objetivo se la otorga el valor uno, mientras que las celdas contiguas van recibiendo un número inmediatamente superior hasta llenar completamente el mapa. El robot se mueve a través del mapa recorriendo las celdas en orden decreciente hasta el objetivo. Si bien este algoritmo es conveniente en un entorno estático y conocido, no es adecuado para esquivar obstáculos no previstos. Asimismo,

resulta computacionalmente poco eficiente actualizar todo el mapa cada vez que el robot encuentra un nuevo obstáculo en su camino (Figura 4).

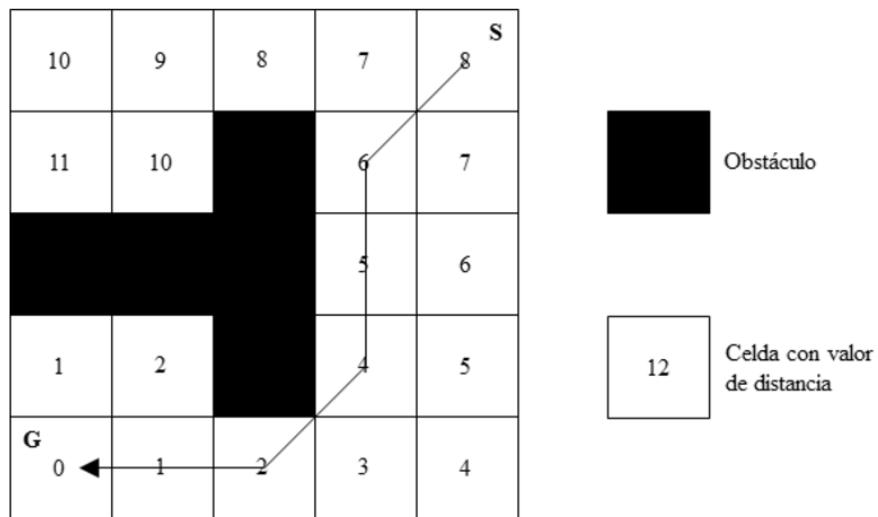


Figura 4: Algoritmo de navegación *Wavefront Planner*

Fuente: Reyes, D.; Millán, G.; Osorio, R. y Lefranc, G. “Mobile Robot Navigation Assisted by GPS”, *IEEE Latin America Transactions*, Vol. 13, N. 6 (2015), p. 1917.

Como veremos más adelante, nuestro subsistema de navegación utiliza un algoritmo que conjuga elementos de los *campos potenciales* y del *wavefront planner*.

Por último, debemos aclarar que nuestro subsistema de navegación requiere *drivers* que permitan comunicarse en tiempo real con los sensores y los actuadores de la plataforma *Magabot*. *IDMind* y *Artica*, fabricantes de *Magabot*, proporcionan un paquete de *drivers* para *Arduino*, llamado *Magabot.h*, escrito en lenguaje *C++*. Para el desarrollo de nuestro proyecto, vamos a modificar el paquete original, reescribiendo en lenguaje *C++* las funciones que mueven los motores, encienden los *LED* y leen los cinco *sonars*, los dos *encoders*, y los cuatro *inputs* de dos *bumpers* (cada *bumper* dispone de dos botones).

## **2.5 ANÁLISIS DEL USO DE MATLAB SIMULINK CODER CON PLACAS ARDUINO**

Como ya mencionamos, el segundo objetivo de este trabajo es analizar las potencialidades de la herramienta *MATLAB Simulink* para controlar las placas *Arduino*, y generar código en lenguaje *C++* con el paquete *Embedded Coder*.

*MATLAB* es un software propietario, escrito en *C*, de la empresa *MathWorks*, que está disponible para sistemas operativos *Unix*, *Linux*, *Windows* y *Mac OS X*. *MATLAB* posee un entorno de desarrollo (IDE) que utiliza un lenguaje de programación interpretado homónimo. Se utiliza para realizar análisis estadísticos, cálculos matemáticos, permite manipular matrices, visualizar funciones y datos, etc.

*Simulink* es un software para la modelación, simulación y análisis de sistemas dinámicos, que está integrado en *MATLAB*. *Simulink* es un instrumento de programación gráfica en diagramas de bloques, con librerías personalizables de bloques. Su uso está en expansión en el mundo de las ingenierías, y su atractivo principal es que permite unificar los procesos de diseño y desarrollo. Cabe señalar que, en *MATLAB Simulink* es reciente, y da lugar a fallos, la integración de componentes que permiten generar código para sistemas embebidos de bajo coste, como el caso de *Arduino*.

*Embedded Coder* es un paquete opcional de *MATLAB* que permite generar código en *C* o *C++* para su implementación en los sistemas embebidos, y además ofrece numerosas posibilidades de análisis y personalización del código.

En este trabajo, se utiliza *Simulink* y *Embedded Coder* para crear los *drivers* de los motores y de los *encoders* de *Magabot*. A través de la herramienta de *profiling*, se analizará la ejecución del código en *C++* de *Magabot* en la placa *Arduino Mega 2560* en tiempo real. Conviene señalar en este punto que la placa *Arduino Mega 2560* es compatible con *MATLAB Simulink*.<sup>1</sup>

Por otra parte, se generarán modelos de *Simulink* con una funcionalidad equivalente a la de los *drivers* de *Magabot*, y compararemos el tiempo de ejecución del código generado a partir de dichos modelos con el código en *C++* de los *drivers*.

---

<sup>1</sup> Sobre la compatibilidad de las placas *Arduino* con *Simulink*, se puede consultar: <https://www.mathworks.com/hardware-support/arduino-simulink.html>

## **2.6 PLAN DE PRUEBAS**

En este apartado detallamos todos los tipos de pruebas que realizaremos sobre nuestro sistema, y describiremos también el entorno en el que estás tendrán lugar.

### **2.6.1 Entorno de pruebas**

Las diferentes pruebas programadas se llevarán a cabo en el Laboratorio de Robótica de la Universidad Pablo de Olavide, situado en el edificio 43 (Fotografía 1). El laboratorio cuenta con un sistema de cámaras llamado *Optitrack*, que es un sistema de cámaras de captura de movimiento (MOCAP).



Fotografía 1: Laboratorio de robótica de la Universidad Pablo de Olavide, Sevilla.

### **2.6.2 Alcance de las pruebas**

A continuación, se describen cada una de las pruebas de funcionamiento que se realizarán.

### 2.6.2.1 Pruebas de odometría en línea recta

Se realizan pruebas de movimiento del robot en línea recta, para determinar la diferencia entre el desplazamiento real y el desplazamiento medido por los *encoders*. En caso de discrepancias significativas entre ambos valores, se calibrará el movimiento de los motores y/o las funciones de odometría.

### 2.6.2.2 Pruebas bidireccionales de odometría en recorrido cuadrado *UBMark*

Se realiza un desplazamiento programado en un recorrido cuadrado de 2m x 2m, en sentido horario y antihorario. Se medirá la diferencia entre el punto de llegada estimado y el punto de llegada real, para poder comparar el desplazamiento calculado usando las lecturas de los *encoders* con el desplazamiento real del robot. Estas mediciones permitirán cuantificar los errores cometidos en el cómputo de la odometría (Borenstein y Feng, L, 1996:869-880)

La Figura 5 muestra una representación gráfica de la prueba.

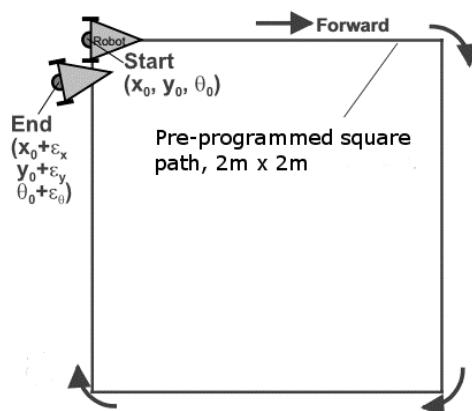


Figura 5: Test *UBMark*.

Fuente: Borenstein, J. y Feng, L. "Measurement and correction of systematic odometry errors in mobile robots". *IEEE Transactions on Robotics and Automation*. Vol. 12, N. 6 (1996), pp. 869-880.

### 2.6.2.3 Pruebas de lecturas de los sonars

Se pondrán objetos delante del robot, y se analizá su capacidad para la detección de obstáculos. Es fundamental para la realización de estas pruebas utilizar objetos de distintas formas como, por ejemplo, cajas rectangulares, patas de sillas y objetos cilíndricos. Determinaremos, además, si la distancia medida por los *sonars* es correcta.

#### **2.6.2.4 Pruebas de orientación del pulso sonoro**

Se cuantificará el error entre el ángulo real del pulso sonoro y el ángulo medido por el robot.

#### **2.6.2.5 Pruebas de control del apuntador.**

Durante las pruebas de funcionamiento del apuntador, se enviarán al servo una secuencia de ángulos, y se comprobará que el apuntador se oriente en la dirección correcta.

#### **2.6.2.6 Pruebas de activación de los LED**

*Magabot* dispone de tres *LED RGB*, dispuestos del siguiente modo: uno en la parte frontal, uno en el costado izquierdo, y otro en el costado derecho.

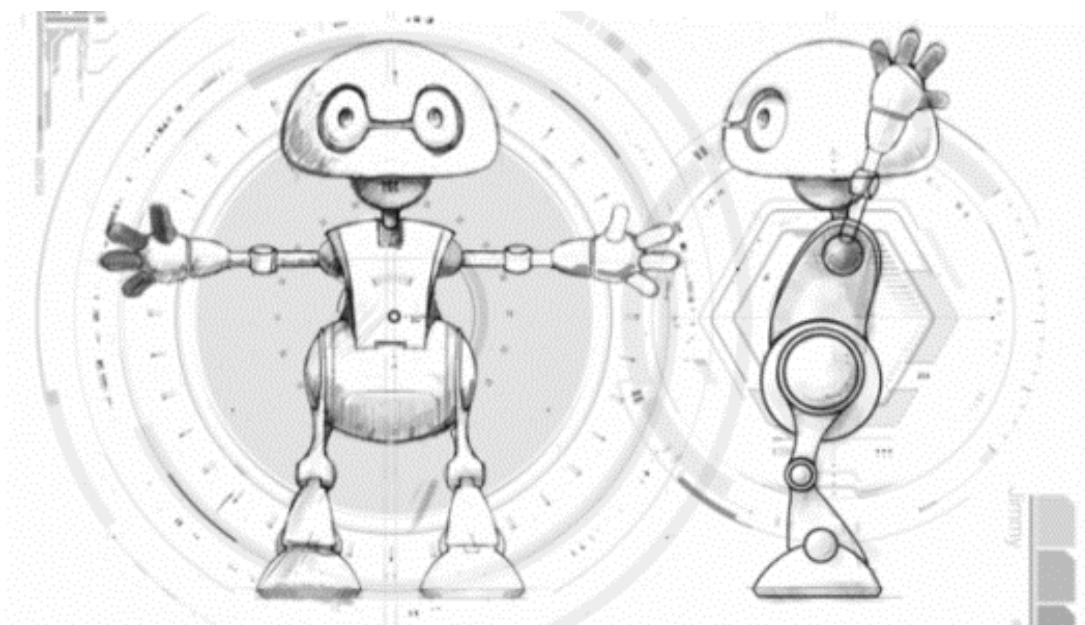
Se realizan pruebas para comprobar si estos *LED* muestran el color correspondiente a los valores de rojo, verde y azul. Para efectuar esta prueba enviaremos, a través de la placa *Arduino Mega 2560*, valores de rojo, verde y azul (*RGB*), en un rango de [0, 255], y comprobaremos que los *LED* muestren el color esperado.

### **2.6.3 Pruebas de aceptación del sistema**

Las pruebas de aceptación consisten en hacer seguir al robot un pulso sonoro, interponiendo en el camino obstáculos fijos. El robot tiene que moverse en dirección del pulso sonoro, evitando colisiones.



# Diseño





## 3 DISEÑO

---

En el presente apartado mostramos los componentes *hardware* y sus conexiones, y describimos el diseño del *software* de los sistemas 01 y 02. Por último, exponemos los diseños de los modelos realizados con *MATLAB Simulink* para implementar los *drivers* de los *encoders* y de los motores de *Magabot*.

### 3.1 ARQUITECTURA DEL SISTEMA

Nuestro sistema robótico hace uso de dos placas *Arduino*: una *Arduino Due* y una *Arduino Mega 2560*, que trabajan conectadas entre sí.

La placa *Arduino Due* se utiliza para realizar la lectura de los tres micrófonos que hemos instalado en la plataforma superior de *Magabot*, con el propósito de determinar el ángulo de origen del pulso sonoro. La *Arduino Due* tiene una frecuencia de *clock* y una velocidad de lectura de puertos analógicos suficientemente rápidas como para poder localizar el origen del sonido con precisión. En la misma placa, además, instalamos un termómetro que mide la temperatura ambiental, factor importante a la hora de calcular la velocidad del sonido. Para poder transmitir el ángulo de proveniencia del sonido, la placa *Arduino Due* se conecta por puerto serie con la placa *Arduino Mega 2560*, encargada de controlar el subsistema de navegación.

Cabe aclarar que la plataforma móvil *Magabot* dispone de una *Arduino Uno*, y de una *shield* fabricada por *IDMind*, que sirve para controlar los motores, los *encoders* y los *sonars*. Esa *shield* está diseñada para la *Arduino Uno*, cuyos pines funcionan con una tensión de 5v. Los pines de la *Arduino Due*, en cambio, funcionan a 3.3v. Por ello, las conexiones de la *shield*, de voltaje superior, podrían dañar la placa. Por esta razón, no podemos usar la *Arduino Due* para realizar la navegación, y decidimos hacer funcionar los dos subsistemas en dos placas distintas.

Por otra parte, la placa *Arduino Uno* no dispone de memoria suficiente para ejecutar el algoritmo de navegación, y tiene sólo un puerto serie. Por este motivo, la sustituimos por una placa *Arduino Mega 2560*, que posee el cuádruple de memoria disponible y cuatro puertos serie. Uno de estos puertos se utiliza para recibir el ángulo desde la placa *Arduino Due*, mientras que otro se conecta con un PC, con el fin de poder realizar el *debugging* y obtener información sobre la ejecución del código. La placa *Arduino Mega 2560* se puede adaptar a la *shield* de *Magabot*, ya que los pines de ambas funcionan con 5v de tensión.

En la Figura 6 mostramos un esquema de las conexiones del sistema.

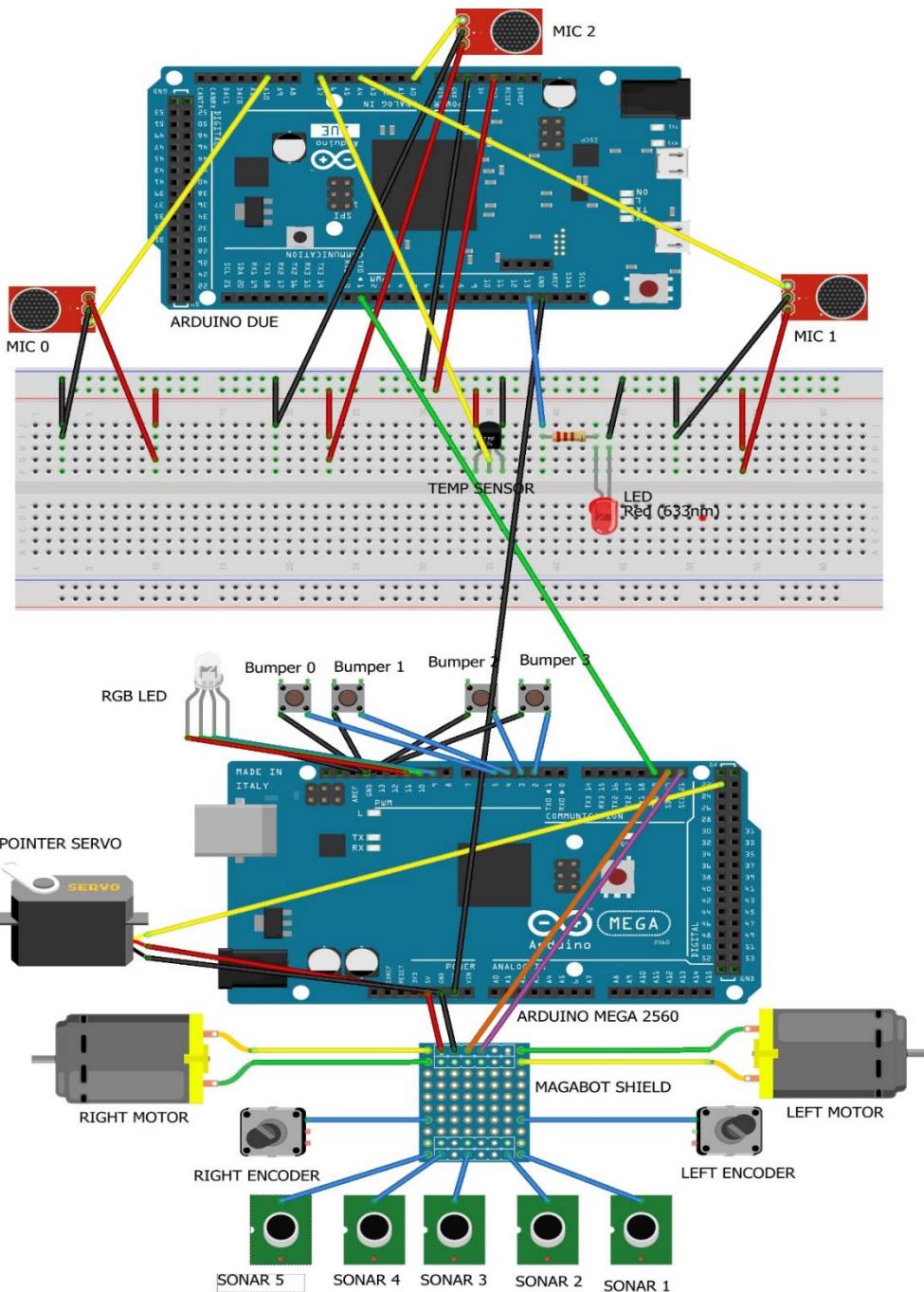
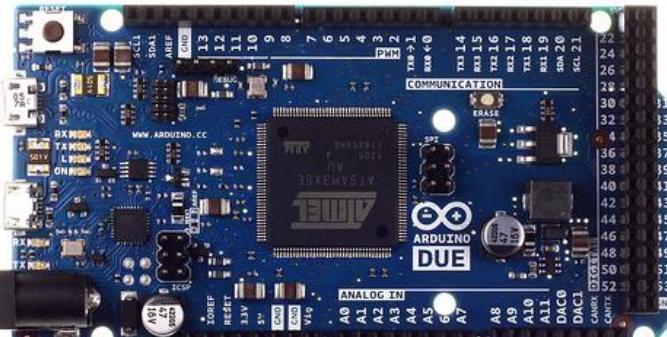


Figura 6: Esquema de conexiones *hardware*

### **3.2 COMPONENTES**

En este apartado proporcionamos una lista que detalla los principales componentes *hardware* necesarios para la realización del proyecto.

<i>Arduino Due</i>		
<b>Descripción</b>	<i>Arduino Due</i> es una placa de desarrollo <i>open hardware</i> . Es la primera placa de la familia <i>Arduino</i> que dispone de un microcontrolador ARM de 32 bits. Esta placa posee una frecuencia de <i>clock</i> de 84 MHz.	
<b>Especificaciones</b>	Microcontrolador	AT91SAM3X8E (ARM 32 bits)
	Velocidad de <i>clock</i>	84 MHz
	SRAM	96 KB
	Memoria <i>flash</i>	512 KB disponibles
	Pines digitales	54 (de los cuales 12 PWM)
	Entradas analógicas	12
	Salidas Analógicas	2
	Puertos serie	4
	Voltaje de operación	3.3 V
	Puertos <i>I2C</i>	2
<b>Precio</b>	37 €	
<b>Cantidad necesaria</b>	1	
<b>Imagen</b>	 The image shows the Arduino Due development board, a blue printed circuit board with a central ATSAM3X8E microcontroller. It features various pins, connectors, and components. The board is labeled "ARDUINO DUE" and "ATSAM3X8E".	

Fotografía 2: Placa *Arduino Due*

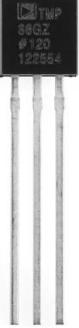
<b><i>Arduino Mega 2560</i></b>		
<b>Descripción</b>	<i>Arduino Mega 2560</i> es una placa de desarrollo <i>open hardware</i> . En comparación con la placa <i>Arduino Uno</i> , dispone de más puertos y más memoria.	
<b>Especificaciones</b>	Microcontrolador	ATmega2560
	Velocidad de <i>clock</i>	16 MHz
	<i>SRAM</i>	8 KB
	<i>EEPROM</i>	4 KB
	Memoria <i>flash</i>	248KB disponibles
	Pines digitales	54 (de los cuales 12 PWM)
	Entradas analógicas	16
	Puertos serie	4
	Voltaje de operación	5 V
	Puertos <i>I2C</i>	2
<b>Precio</b>	41.50 €	
<b>Cantidad necesaria</b>	1	
<b>Imagen</b>	 Fotografía 3: Placa Arduino Mega 2560	

<b><i>Magabot</i></b>																		
<b>Descripción</b>	<p><i>Magabot</i> es una plataforma robot móvil <i>open hardware</i> basada en <i>Arduino</i>. La producen en conjunto las empresas portuguesas <i>IDMind</i> y <i>Artica</i>. Se trata de un robot diferencial con dos ruedas motrices y una rueda “loca” giratoria.</p> <p>La plataforma dispone dos motores <i>DC</i>, dos <i>encoders</i>, dos <i>bumpers</i>, tres <i>LED RGB</i>, cinco <i>sonars</i>, una placa <i>Arduino Uno</i> y una <i>shield</i> de control. Es posible instalar una estructura con estantes para el transporte de una <i>netbook</i>.</p>																	
<b>Especificaciones</b>	<table border="1"> <tr> <td>Tipo de plataforma</td><td>Robot móvil diferencial</td></tr> <tr> <td>Motores</td><td>2 motores <i>DC</i></td></tr> <tr> <td>Ruedas</td><td>2 ruedas motrices. Radio: 4.5cm</td></tr> <tr> <td>Eje</td><td>34 cm</td></tr> <tr> <td><i>Sonars</i></td><td>5, frontales</td></tr> <tr> <td><i>LED</i></td><td>3 <i>LED RGB</i></td></tr> <tr> <td><i>Encoders</i></td><td>2 <i>encoders</i>. Resolución: 3900 ticks por vuelta</td></tr> <tr> <td>Circuitos</td><td><i>Magabot shield</i>, controlador de motores, <i>encoders</i> y <i>sonars</i>.</td></tr> </table>		Tipo de plataforma	Robot móvil diferencial	Motores	2 motores <i>DC</i>	Ruedas	2 ruedas motrices. Radio: 4.5cm	Eje	34 cm	<i>Sonars</i>	5, frontales	<i>LED</i>	3 <i>LED RGB</i>	<i>Encoders</i>	2 <i>encoders</i> . Resolución: 3900 ticks por vuelta	Circuitos	<i>Magabot shield</i> , controlador de motores, <i>encoders</i> y <i>sonars</i> .
Tipo de plataforma	Robot móvil diferencial																	
Motores	2 motores <i>DC</i>																	
Ruedas	2 ruedas motrices. Radio: 4.5cm																	
Eje	34 cm																	
<i>Sonars</i>	5, frontales																	
<i>LED</i>	3 <i>LED RGB</i>																	
<i>Encoders</i>	2 <i>encoders</i> . Resolución: 3900 ticks por vuelta																	
Circuitos	<i>Magabot shield</i> , controlador de motores, <i>encoders</i> y <i>sonars</i> .																	
<b>Precio</b>	400 €																	
<b>Cantidad necesaria</b>	1																	
<b>Imágenes</b>	 <p>Fotografía 4: Plataforma Magabot</p>																	

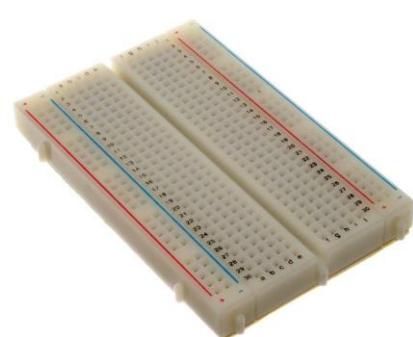
	
Fotografía 5: <i>Magabot shield</i>	

<b>Batería XCell Ni-MH Racing Pack 2500 mah 7,2 V</b>		
<b>Descripción</b>	Pack de baterías para alimentar los motores de <i>Magabot</i> .	
<b>Especificaciones</b>	Capacidad	2500mah
	Voltaje	7,2 V
	Tipo	NiMH
	Conexión	Tamiya
	Dimensiones	132mm/45mm/23mm
	Peso	307g
<b>Precio</b>	18€ c.u.	
<b>Cantidad necesaria</b>	1	
<b>Imagen</b>		
Fotografía 6: Batería XCell Ni-MH Racing Pack 2500 mah 7,2 V		

Micrófonos <i>KY-037</i>								
<b>Descripción</b>	<p>Este módulo está compuesto por un micrófono sensible <i>KY-037</i>.</p> <p>El módulo tiene dos salidas: una analógica y una digital. La salida analógica no proporciona valores absolutos (dB), sino relativos. El módulo dispone de un potenciómetro para regular el umbral de activación de la salida digital.</p>							
<b>Especificaciones</b>	<table border="1"> <tr> <td>Salidas</td><td>1 salida analógica (AO) + 1 salida digital (DO).</td></tr> <tr> <td>Voltaje</td><td>3,3 / 5 v</td></tr> <tr> <td>Accesorios</td><td>Potenciómetro</td></tr> </table>		Salidas	1 salida analógica (AO) + 1 salida digital (DO).	Voltaje	3,3 / 5 v	Accesorios	Potenciómetro
Salidas	1 salida analógica (AO) + 1 salida digital (DO).							
Voltaje	3,3 / 5 v							
Accesorios	Potenciómetro							
<b>Precio</b>	2€ c.u.	3						
<b>Cantidad necesaria</b>								
<b>Imagen</b>								
								
	<p>Fotografía 7: Módulo micrófono <i>KY-037</i></p>							

Sensor de temperatura <b>TMP36GZ</b>		
<b>Descripción</b>	Se trata de un sensor de temperatura de bajo voltaje. No requiere calibración.	
<b>Especificaciones</b>	Salidas	1 salida analógica.
	Voltaje	2,7 – 5,5 V.
	Rango de operación	Desde - 40°C, hasta +125°C.
	<i>Output</i>	10 mV/°C
	Precisión	2°C
	Linealidad	±0.5°C
<b>Precio</b>	1€ c.u.	
<b>Cantidad necesaria</b>	1	
<b>Imagen</b>		
Fotografía 8: Sensor de temperatura <i>TMP36</i>		

Cables Dupont		
<b>Descripción</b>	Cables idóneos para <i>protoboards</i> , que no requieren de soldaduras para conectarse. Se utilizan para realizar prototipos.	
<b>Especificaciones</b>	Tamaño	20cm
	Pines	Macho – hembra
<b>Precio</b>	2,80€ /40pcs.	
<b>Cantidad necesaria</b>	40	
<b>Imagen</b>		
Fotografía 9: Cables Dupont		

Placa <i>solderless breadboard</i>		
<b>Descripción</b>	Placa para realizar prototipos de proyectos de electrónica. No necesita soldaduras.	
<b>Precio</b>	10€	
<b>Cantidad necesaria</b>	1	
<b>Imagen</b>		
Fotografía 10: Placa <i>solderless breadboard</i>		

<b>Piezas Lego Technic</b>	
<b>Descripción</b>	Piezas mixtas de un kit de construcciones <i>Lego Technic</i> . Estas piezas son ideales para construir estructuras sencillas con facilidad.
<b>Precio</b>	n.d.
<b>Imagen</b>	 <p>Fotografía 11: Piezas de <i>Lego Technic</i></p>

<b><i>PC Acer Aspire E3-111-C5GL</i></b>		
<b>Descripción</b>	La <i>netbook Acer Aspire E3-111-C5GL</i> tiene instalado el sistema operativo <i>Ubuntu 14.04</i> . En este proyecto, se usan sus puertos <i>USB</i> para alimentar a las placas <i>Arduino Due</i> y <i>Arduino Mega 2560</i> . El <i>PC</i> también es utilizado para depurar el código de las placas <i>Arduino</i> .	
<b>Especificaciones</b>	Procesador	Intel Celeron N2930
	Sistema Operativo	Ubuntu 14.04 LTS
	Pantalla	11.6"
	Memoria	4GB
	Almacenamiento	500GB
<b>Precio</b>	250€	
<b>Cantidad necesaria</b>	1	
<b>Imagen</b>		
Fotografía 12: <i>PC Acer Aspire E3-111-C5GL</i>		

### **3.3 DISEÑO DEL SUBSISTEMA 01: DETECCIÓN DE UN PULSO SONORO**

Para calcular la dirección del origen del pulso sonoro utilizamos tres micrófonos *KY-037* posicionados de forma equidistante entre sí en la parte superior del robot. Registraremos el tiempo de llegada del pulso sonoro a cada uno de los micrófonos, con el fin de calcular el ángulo acimutal de proveniencia del mismo (Figura 7)

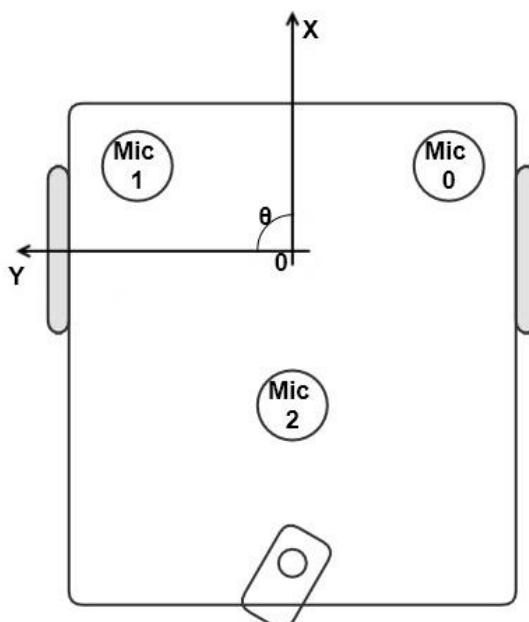


Figura 7: Disposición de los micrófonos en la parte superior del robot

Cabe aclarar que, antes de recibir un pulso sonoro, el robot se encuentra estático. En esta posición se define el sistema referencial externo, el cual tiene su punto de origen en el centro del eje de las ruedas. La dirección positiva del eje *x* va desde centro del robot hacia el frente, mientras que la dirección positiva del eje *y* va desde el centro del robot hacia la izquierda. Al desplazarse, el robot se aleja del origen del sistema referencial.

La lectura analógica de los micrófonos *KY-037* devuelve valores comprendidos en un rango de [0, 1023]. Al iniciar la ejecución, el programa realiza varias lecturas de los micrófonos. Esas lecturas son promediadas con el propósito de conocer el nivel de ruido ambiental. Para cada micrófono

registramos el tiempo, en microsegundos, del instante en el que su lectura analógica difiere del nivel de ruido por 10 unidades.

La forma de interpretar la lectura de estos tres micrófonos varía de acuerdo al tiempo de llegada del pulso sonoro a cada uno de ellos. Sobre la base del cálculo de la diferencia entre los tiempos de llegada del pulso sonoro a los tres micrófonos, el sistema selecciona a la pareja de micrófonos que ha obtenido una diferencia menor de tiempo entre los valores registrados. Esos dos micrófonos son utilizados para medir el ángulo acimutal.

El tercer micrófono, que ha registrado el valor de tiempo más diferenciado con respecto a los otros dos micrófonos, se utiliza para invertir el signo del ángulo calculado mediante una rotación de  $\pi$  radianes, en los casos en los que el sonido lo ha alcanzado antes que a los otros dos.

En la Figura 8, podemos observar la disposición de los tres micrófonos (Mic 0, Mic 1 y Mic 2). Realizamos este gráfico con la finalidad de mostrar qué pareja de micrófonos (Mic 0-1, Mic 1-2, Mic 2-0) será utilizada para la medición del ángulo acimutal, según el origen del pulso sonoro.

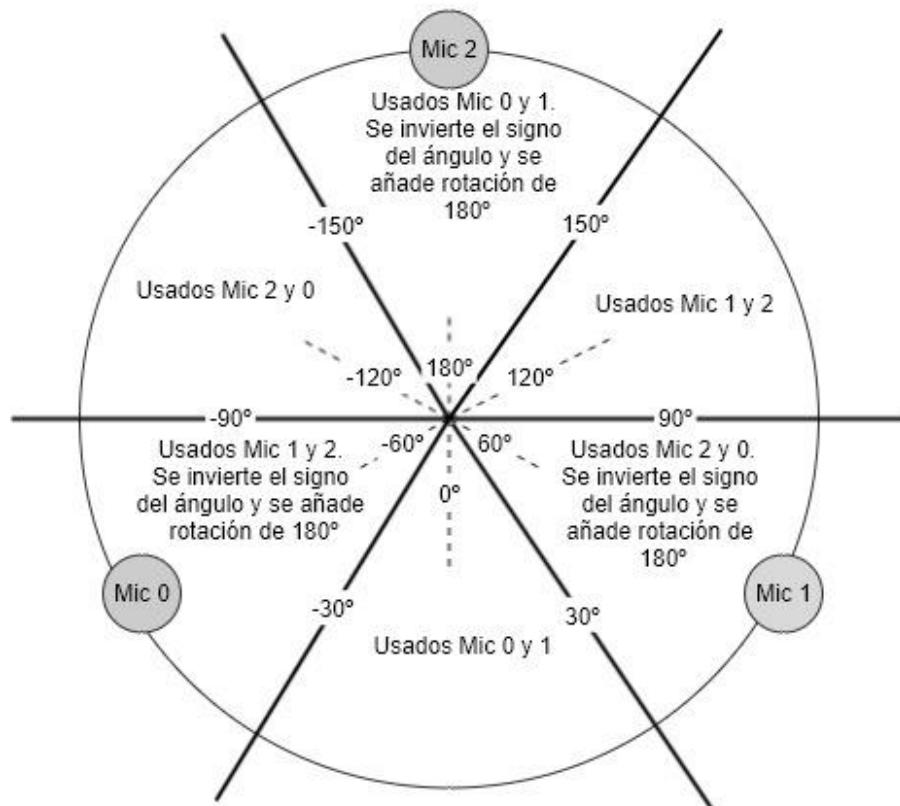


Figura 8: Uso de los micrófonos según el ángulo de origen del pulso sonoro

Para determinar el ángulo de llegada de un pulso sonoro usando dos micrófonos aplicamos la siguiente fórmula:

$$\theta = \arcsin\left(\frac{\Delta t * c}{mic\_distance}\right)$$

$\theta$  es el ángulo en radianes,  $\Delta t$  la diferencia de tiempo de llegada de la onda sonora a cada micrófono,  $c$  la velocidad del sonido, y  $mic\_distance$  es la distancia entre los dos micrófonos utilizados en el cómputo.

Por su parte, a la velocidad de propagación del sonido en el aire la determinamos con la siguiente fórmula:

$$c = \sqrt{k R T}$$

$K$  es el calor específico,  $R$  la constante individual del gas, y  $T$  la temperatura absoluta en grados Kelvin.

Considerando como constantes a  $k = 1.4$  y a  $R = 286.9 \text{ J/Kg}$  para el aire con una atmósfera de presión, solo falta medir la temperatura ambiental con el fin de poder calcular la velocidad del sonido. Para ello, como dijimos, instalamos en el robot un termómetro *TMP36*.

Para determinar si la lectura de los micrófonos es válida o no, implementamos comprobaciones preventivas para que se ignoren, por ejemplo, los ruidos o las vibraciones producidos por el mismo robot que activen los micrófonos. La primera medida de seguridad, en este sentido, es utilizar un temporizador que se activa en el momento de la primera lectura de un sonido por uno de los micrófonos, y que se reinicia después de la segunda lectura, por parte de otro de los micrófonos. En el caso que transcurra más tiempo del que necesita el sonido para recorrer la distancia entre dos micrófonos sin que se produzca ninguna lectura, todas las lecturas anteriores son descartadas. Las lecturas también son descartadas si se han producido con tiempos demasiado cercanos.

Una vez que se ha calculado el ángulo de origen del pulso sonoro, el *Arduino Due* envía el resultado, por puerto serie, al *Arduino Mega 2560*.

En la Figura 9, mostramos el diagrama de flujo del software de detección de la orientación del pulso sonoro.

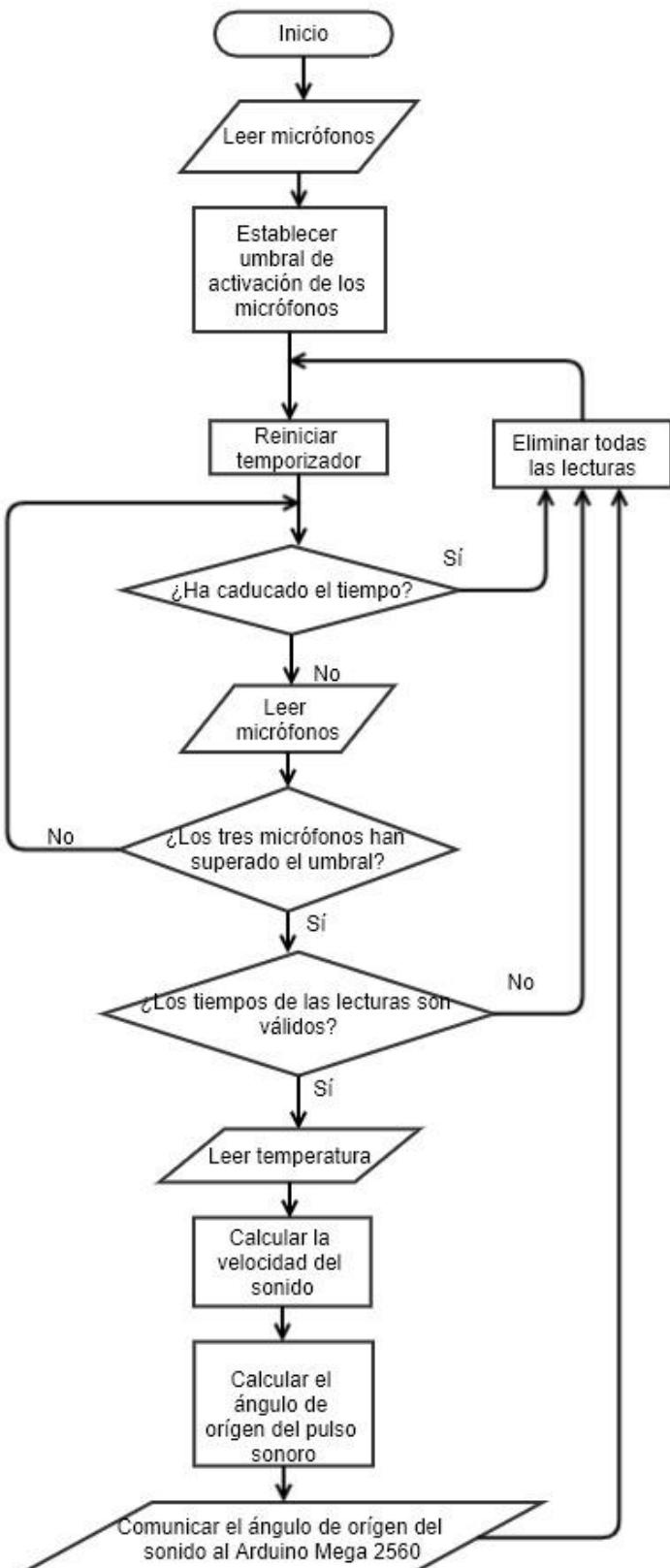


Figura 9: Diagrama de flujo del subsistema 01

### 3.3.1 Funciones del subsistema 01

<i>void setup()</i>	
Descripción	<p>La función <i>setup()</i> es una parte fundamental de los programas de <i>Arduino</i>. Es la primera función en ser ejecutada al iniciar la placa <i>Arduino Due</i>.</p> <p>En esta función se inicializa el puerto serie encargado de comunicar el ángulo de origen del sonido a la placa <i>Arduino Due</i>.</p> <p>Sucesivamente, se llama a la función <i>void set_sound_speed()</i>, la cual calcula la velocidad del sonido en base a la temperatura ambiental.</p> <p>Luego, se realizan mil lecturas de los micrófonos y se promedia el resultado, para conocer cuál es el nivel medio de ruido ambiental al iniciar el sistema.</p> <p>Al final, se hace parpadear un <i>LED</i> rojo con la función <i>blink()</i> para indicar que la función <i>setup()</i> ha terminado su ejecución.</p>
Llamada por	--
Llama a las funciones	<ul style="list-style-type: none"> <li>• <i>Serial.begin(int baudRate)</i></li> <li>• <i>void blink()</i></li> <li>• <i>void set_sound_speed()</i></li> </ul>

<b>void loop()</b>	
<b>Descripción</b>	<p>La función <b>loop()</b> es una parte fundamental de los programas de <i>Arduino</i>. Se ejecuta después de la función <b>setup()</b>, y se repite en un bucle infinito.</p> <p>En este sistema, la función <b>loop()</b> realiza la lectura de los tres micrófonos, verificando si cada uno de ellos ha superado el umbral. En el caso de que un micrófono supere el umbral, se almacena el tiempo en microsegundo en el que ha sido realizada la lectura, y se activa un temporizador, el cual es reiniciado en la siguiente lectura realizada por otro micrófono. Si el temporizador supera el límite establecido, se borran todas las lecturas.</p> <p>Cuando se obtienen tres lecturas válidas antes que caduque el tiempo, se inicia el proceso de cálculo del origen del sonido. Para ello, en primer lugar, se verifica si las tres lecturas tienen tiempos válidos. Si las tres lecturas han sido demasiado rápidas, son descartadas. En caso contrario, se analizan los tiempos de las lecturas para decidir cuáles micrófonos usar para realizar el cómputo del ángulo, y para definir la rotación que se debe aplicar al ángulo calculado. Sucesivamente, se calcula la velocidad del sonido usando la función <b>void set_sound_speed()</b>. Luego, se calcula el ángulo usando la fórmula:</p> $\text{angle} = \text{asin}((\text{recDelay} * \text{sound\_speed}) / \text{mic\_distance}) * \text{frontRear} + \text{angleOffset}.$ <p>En esta fórmula, <i>recDelay</i> es la diferencia de tiempo entre las lecturas de los dos micrófonos seleccionados; <i>sound_speed</i> es la velocidad del sonido; <i>mic_distance</i> es la distancia entre los micrófonos; <i>frontRear</i> es equivalente a 1 o a -1, en el caso de que haya que invertir el ángulo; mientras que <i>angleOffset</i> es la rotación que hay que aplicar al ángulo (el valor de <i>angleOffset</i> y de <i>frontRear</i> dependerá de cuáles fueron los micrófonos seleccionados para el cómputo del ángulo)</p> <p>Una vez realizado el cálculo, el rango del ángulo es restringido al intervalo <math>(-\pi, \pi]</math> y, finalmente, es enviado a la placa <i>Arduino Mega 2560</i>, a través de la función <b>Serial.print(double angle)</b>.</p>
<b>Llamada por</b>	--
<b>Llama a las funciones</b>	<ul style="list-style-type: none"> <li>• <b>size_t Serial.print(double angle)</b></li> <li>• <b>void blink()</b></li> <li>• <b>void set_sound_speed()</b></li> <li>• <b>void reset()</b></li> </ul>

<i>void blink()</i>	
<b>Descripción</b>	Esta función hace parpadear un <i>led</i> rojo.
<b>Llamada por</b>	<ul style="list-style-type: none"> <li>• <i>void setup()</i></li> </ul>
<b>Llama a las funciones</b>	--

<i>void set_sound_speed()</i>	
<b>Descripción</b>	Esta función lee el voltaje del sensor de temperatura, y lo convierte a grados centígrados. Posteriormente, calcula la velocidad del sonido usando la fórmula $sound\_speed = \sqrt{k * R * T}$ , siendo $k$ igual a 1,4, $R$ igual a 286,9 y $T$ la temperatura en grados Kelvin.
<b>Llamada por</b>	<ul style="list-style-type: none"> <li>• <i>void setup()</i></li> <li>• <i>void loop()</i></li> </ul>
<b>Llama a las funciones</b>	--

<i>void reset()</i>	
<b>Descripción</b>	Esta función borra las lecturas de los micrófonos, poniéndolas en cero. De la misma manera, reinicia el contador de lecturas.
<b>Llamada por</b>	<ul style="list-style-type: none"> <li>• <i>void loop()</i></li> </ul>
<b>Llama a las funciones</b>	--

### 3.4 DISEÑO DEL SUBSISTEMA 02: NAVEGACIÓN DEL ROBOT

El *Magabot* es un robot con una configuración de tipo diferencial. En los dos lados se encuentran situadas las ruedas motrices, ambas disponen de un motor independiente. En la parte trasera, se localiza una rueda “loca” giratoria –del tipo de las ruedas de los carritos de los supermercados– que no tiene motor, y cuya única función consiste en sujetar al robot.

#### 3.4.1 Diseño de la odometría

Para que el *Magabot* se mueva en línea recta, es necesario que las ruedas motrices se muevan en la misma dirección y con la misma velocidad, mientras que para curvar se deben mover las ruedas con una velocidad u orientación distintas.

En la Figura 10, podemos observar la configuración diferencial de *Magabot*.

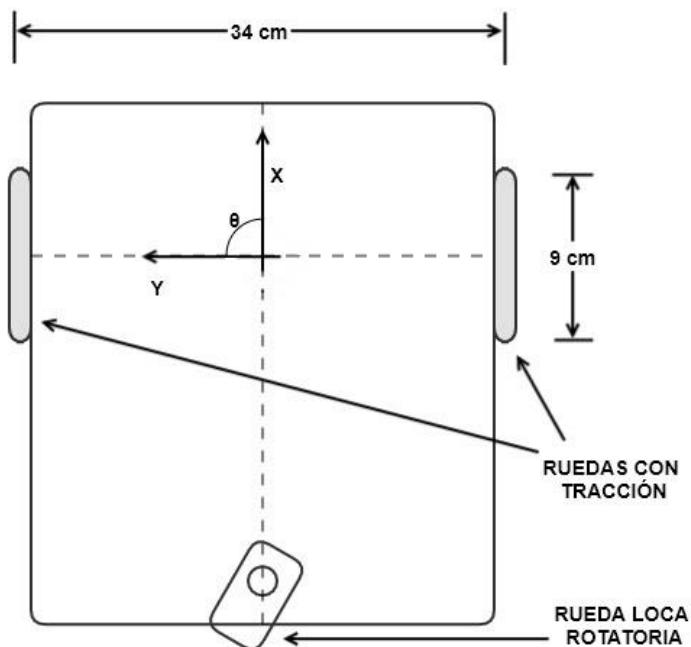


Figura 10: Configuración diferencial de *Magabot*.

Cada rueda motriz dispone de un *encoder*. Este dispositivo permite medir cuánto ha rodado un eje en un determinado lapso de tiempo, a través del cómputo de cuántas veces pasa un haz de luz por un disco agujereado fijado en el eje de la rueda.

En la Figura 11, mostramos la estructura interna de un *encoder*.

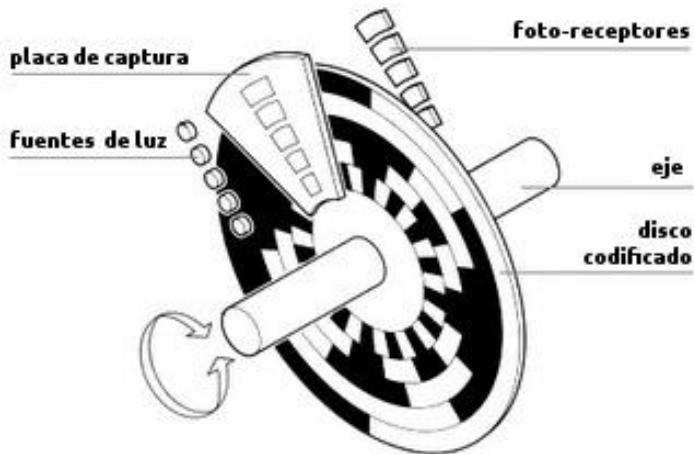


Figura 11: Estructura interna de un *encoder*

Para que el robot logre alcanzar un objetivo, es esencial que conozca en todo momento su posición. Cuando el robot se encuentra detenido, como ya mencionamos, se define el sistema referencial externo, el cual tiene su punto de origen en el centro del eje de las ruedas. Conviene recordar que la dirección positiva del eje  $x$  va desde el centro del robot hacia el frente, y la dirección positiva del eje  $y$  va desde el centro del robot hacia la izquierda. En situación estática se asigna a la posición del robot los valores  $(0,0,0)$ , que representan, respectivamente, a las variables  $x$ ,  $y$ , y  $\theta$ ; siendo  $\theta$  el ángulo.

Para cumplir con la meta de seguir en todo momento la posición del robot, se deben implementar funciones para estimar los desplazamientos mediante la técnica conocida como odometría. Conociendo el radio de las ruedas motrices, la distancia entre ellas, y las lecturas de los *encoders*, podemos medir los cambios de posición del robot.

La distancia lineal recorrida por una rueda en un lapso de tiempo se calcula utilizando la siguiente fórmula:

$$d_{l,r} = \frac{2\pi r t_{l,r}}{k}$$

En esta fórmula,  $d_{l,r}$  es la distancia lineal recorrida por una rueda,  $r$  es el radio de las ruedas;  $t_{l,r}$  es la cantidad de ticks registrados por el *encoder* izquierdo o derecho; y  $k$  es la resolución del *encoder*, medido en ticks por vuelta. La distancia recorrida por el centro del eje de las ruedas del robot ( $d_c$ ) se calcula de la siguiente manera:

$$d_c = \frac{d_r + d_l}{2}$$

$d_r$  y  $d_l$  son las distancias recorridas por la rueda derecha e izquierda, respectivamente.

Para conocer el ángulo de orientación del robot en un determinado momento ( $\theta_i$ ) calculamos:

$$\theta_i = \theta_{i-1} + \frac{d_r + d_l}{L}$$

$L$  es la distancia entre las dos ruedas motrices.

Una vez conocido el ángulo de dirección del robot, podemos calcular las nuevas coordenadas del robot en el sistema de referencia de la siguiente forma:

$$x_i = x_{i-1} + d_c \cos(\theta_i)$$

$$y_i = y_{i-1} + d_c \sin(\theta_i)$$

En la Figura 12, presentamos una representación gráfica de las mediciones realizadas tras un desplazamiento en dos posiciones consecutivas de *Magabot*.

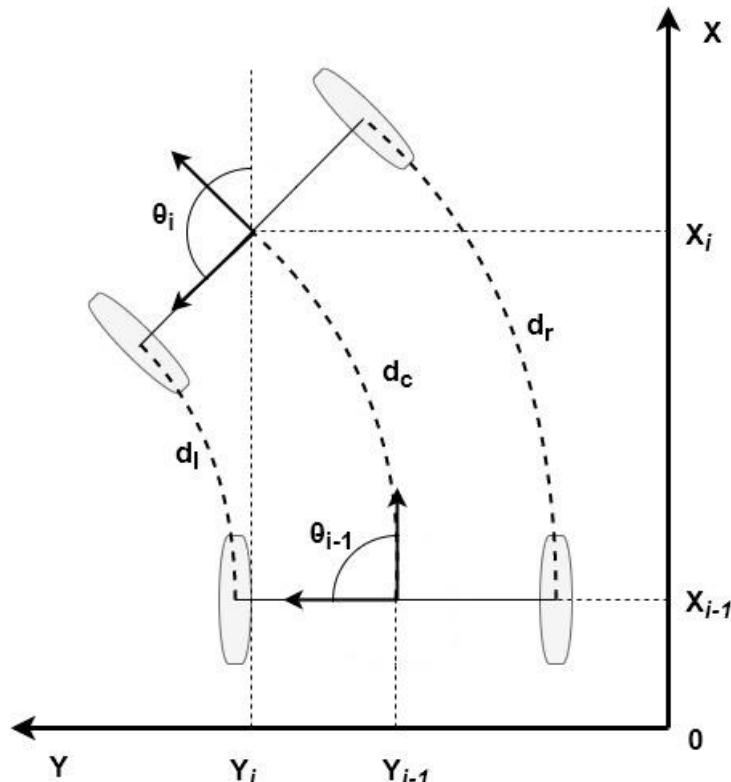


Figura 12: Mediciones del desplazamiento de *Magabot*

La odometría basada en la lectura de los *encoders* conlleva una acumulación de errores. Es necesario tomar algunas medidas para evitar que la imprecisión de la odometría comprometa el funcionamiento correcto del robot. Los movimientos de las ruedas tienen que ser suaves para evitar deslizamientos, además, es necesario resetear la odometría con frecuencia para frenar la acumulación de errores. Otra medida de seguridad que aplicamos en este trabajo consiste en “inflar” o “expandir” los obstáculos, de manera que el robot los considere más grandes de los que en realidad son, y mantenga una distancia de seguridad, dejando un cierto margen de espacio para los errores.

### 3.4.2 Diseño del algoritmo de navegación

Como hemos adelantado en el análisis, en este proyecto utilizamos un algoritmo inspirado en los *campos potenciales* y en el *wavefront planner*, que hace uso de campos potenciales numéricos y no vectoriales.

Cuando el subsistema 02 recibe el ángulo de orientación del pulso sonoro, por un lado, crea una matriz de 50 x 50 celdas que representa un mapa de 5m x 5m alrededor del robot y, por otro, fija como destino el punto más lejano en ese mapa, en dirección del sonido. Luego, en cada celda del mapa se almacena la distancia euclídea, en decímetros, entre la misma celda y el punto objetivo (Figura 13).

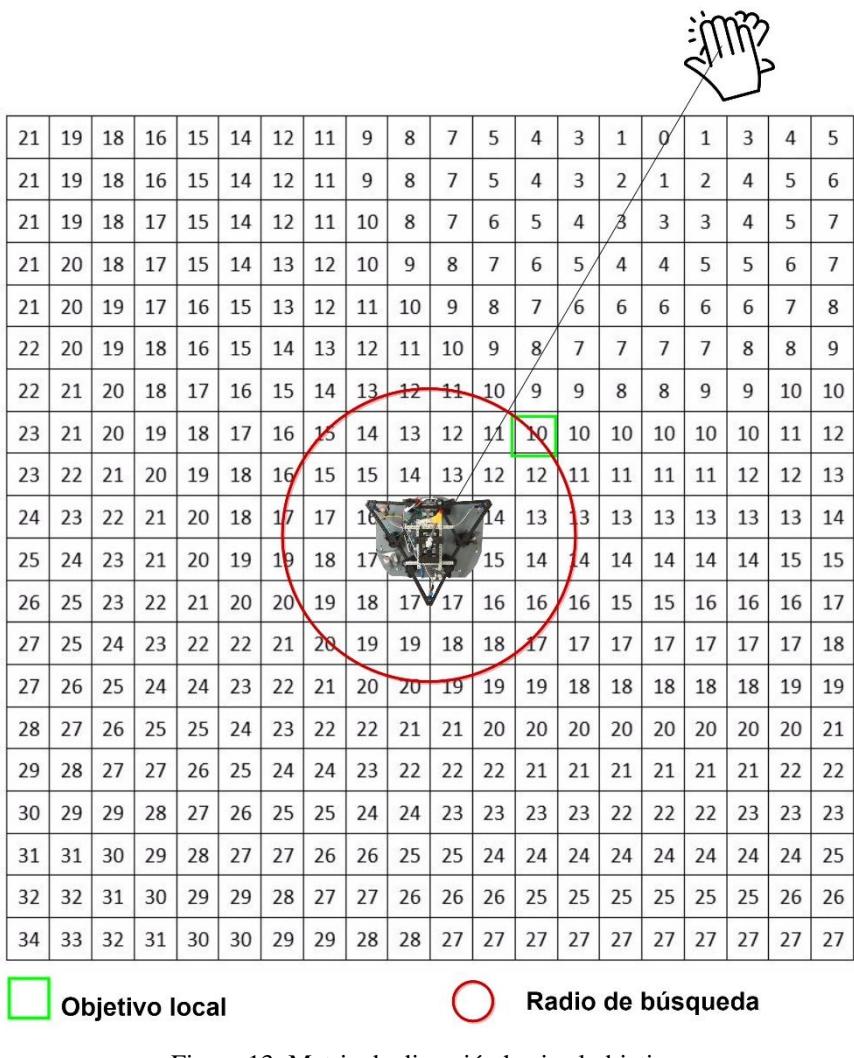


Figura 13: Matriz de dirección hacia el objetivo

Para prevenir colisiones, además de una matriz de dirección, se crea una matriz de obstáculos con todos los valores inicializados con valor cero. Cuando los *sonars* detectan algún objeto en el recorrido, en el mapa de obstáculos se genera un campo de fuerzas ficticias alrededor del objeto localizado.

En todo momento, el robot busca, en un determinado radio alrededor suyo, el punto del mapa que obtenga el menor valor en la suma de la matriz de dirección y la matriz de obstáculos. Posteriormente, fija ese valor como objetivo local. De esta manera, el robot se mueve siguiendo las celdas con menor fuerza.

Cuando el sistema fija como objetivo local una casilla ocupada por el robot, significa que ha sido alcanzado un mínimo local. Para impedir que el

robot quede bloqueado en ese punto, se genera una fuerza ficticia en esta zona para que el robot se aleje.

Para exemplificar lo explicado anteriormente, en la Figura 14 podemos observar tres dibujos de matrices. En la primera de estas imágenes se grafica una matriz con un campo de atracción hacia el objetivo (a). En la segunda se muestra una matriz con campos de repulsión alrededor de los obstáculos (b). En la tercera imagen, se representa la suma de las dos primeras matrices (c). El ejemplo que ilustra esta última imagen, nos sirve para destacar que el campo resultante de la suma de matrices permite al robot moverse en dirección de su objetivo, esquivando al mismo tiempo los obstáculos. Sin embargo, debemos aclarar que, a diferencia del campo aquí representado, el campo de atracción que implementaremos en nuestro algoritmo tendrá una forma de semicono.

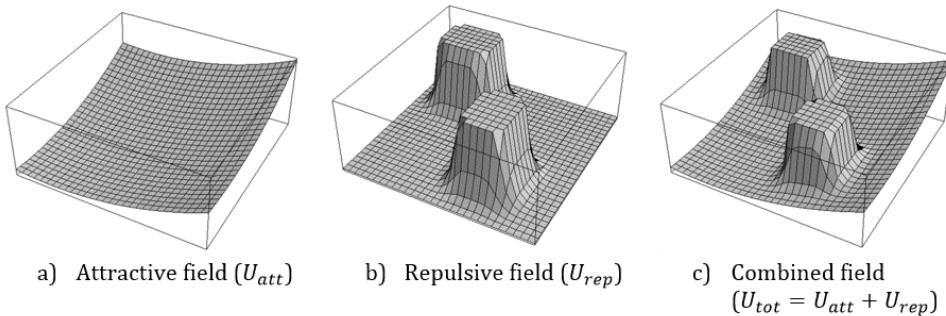


Figura 14: Matrices de dirección y de obstáculos

Fuente: Ferreira, L. et al, “Embedded Motion Control 2015 Group 2”, en  
[http://cstwiki.wtb.tue.nl/index.php?title=Embedded\\_Motion\\_Control\\_2015\\_Group\\_2](http://cstwiki.wtb.tue.nl/index.php?title=Embedded_Motion_Control_2015_Group_2)

El robot no puede moverse hacia puntos que estén fuera del área cubierta por el mapa de dirección, y tampoco puede añadir fuerzas repulsivas fuera del mapa de obstáculos. Por esta razón, cuando el robot se acerca a los límites se reposicionan los mapas de dirección y de obstáculos. En ese proceso se hace coincidir el punto central de ambos mapas con el punto de ubicación del robot en el sistema referencial. Durante todo el trayecto del robot, el sistema referencial permanece fijo, y los ejes de los sucesivos mapas generados siguen manteniendo la orientación asignada al inicio de la navegación. La representación gráfica del reposicionamiento de los mapas se puede observar en la Figura 15.

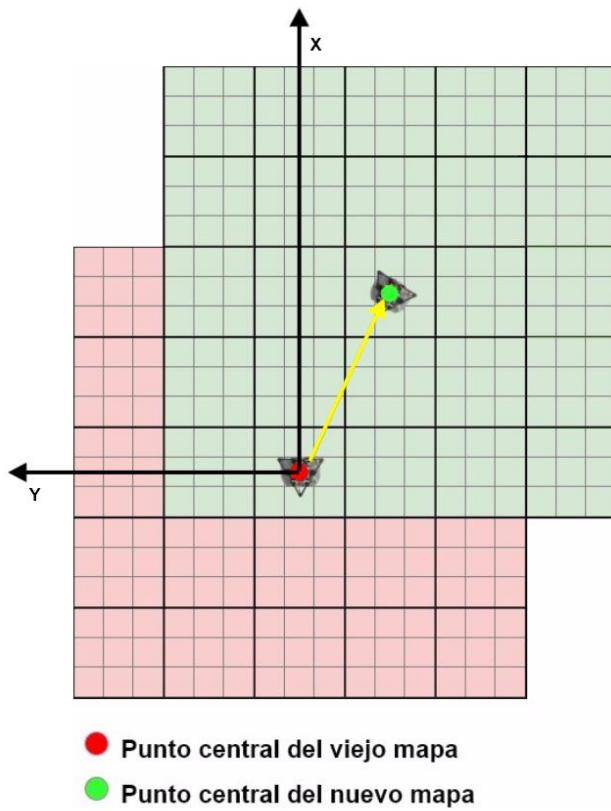


Figura 15: Desplazamiento de los mapas de orientación y de obstáculos

La actualización continua de los mapas sirve también para incrementar la capacidad de elusión de objetos del robot, ya que, en cada actualización, se vuelve a establecer la relación correcta entre la ubicación del robot y la posición del mapa de obstáculos. Esto se convierte en una ventaja importante, especialmente si consideramos que el riesgo de colisión es susceptible de aumentar a causa de la propagación de errores en la odometría.

El robot logra localizar los obstáculos situados en su camino a través de la lectura de cinco *sonars*. Debemos tener en cuenta que se trata de instrumentos poco precisos. En la Figura 16, podemos observar tres tipos de comportamiento de un sónar:

- a)** cuando realiza una correcta detección de un obstáculo,
- b)** cuando el sónar no localiza el objeto debido al ángulo pronunciado del mismo,
- c)** cuando el sónar detecta en posición equivocada un determinado objeto, engañado por la refracción de la onda sonora.

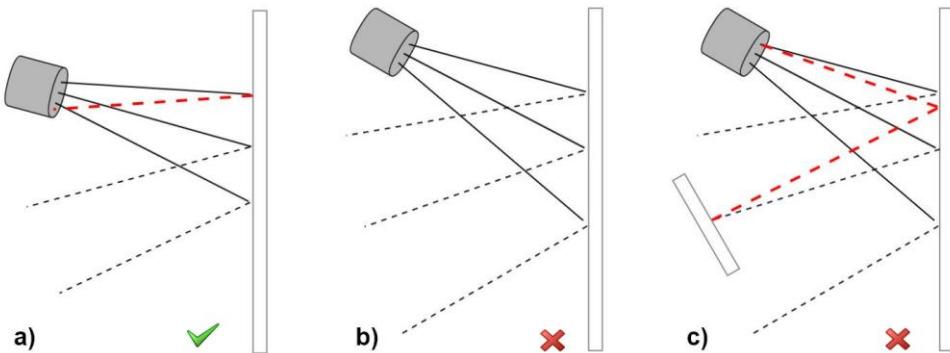


Figura 16: Limitaciones físicas de los *sonars*

En la parte frontal del robot se encuentran dos *bumpers*, estos dispositivos son utilizados para detectar colisiones con algún objeto. En caso de choque, los motores se detienen inmediatamente y se reinician los valores de odometría, los mapas de dirección y los mapas de obstáculos.

En la Figura 17, mostramos el diagrama de flujo que resume el funcionamiento del subsistema de navegación.

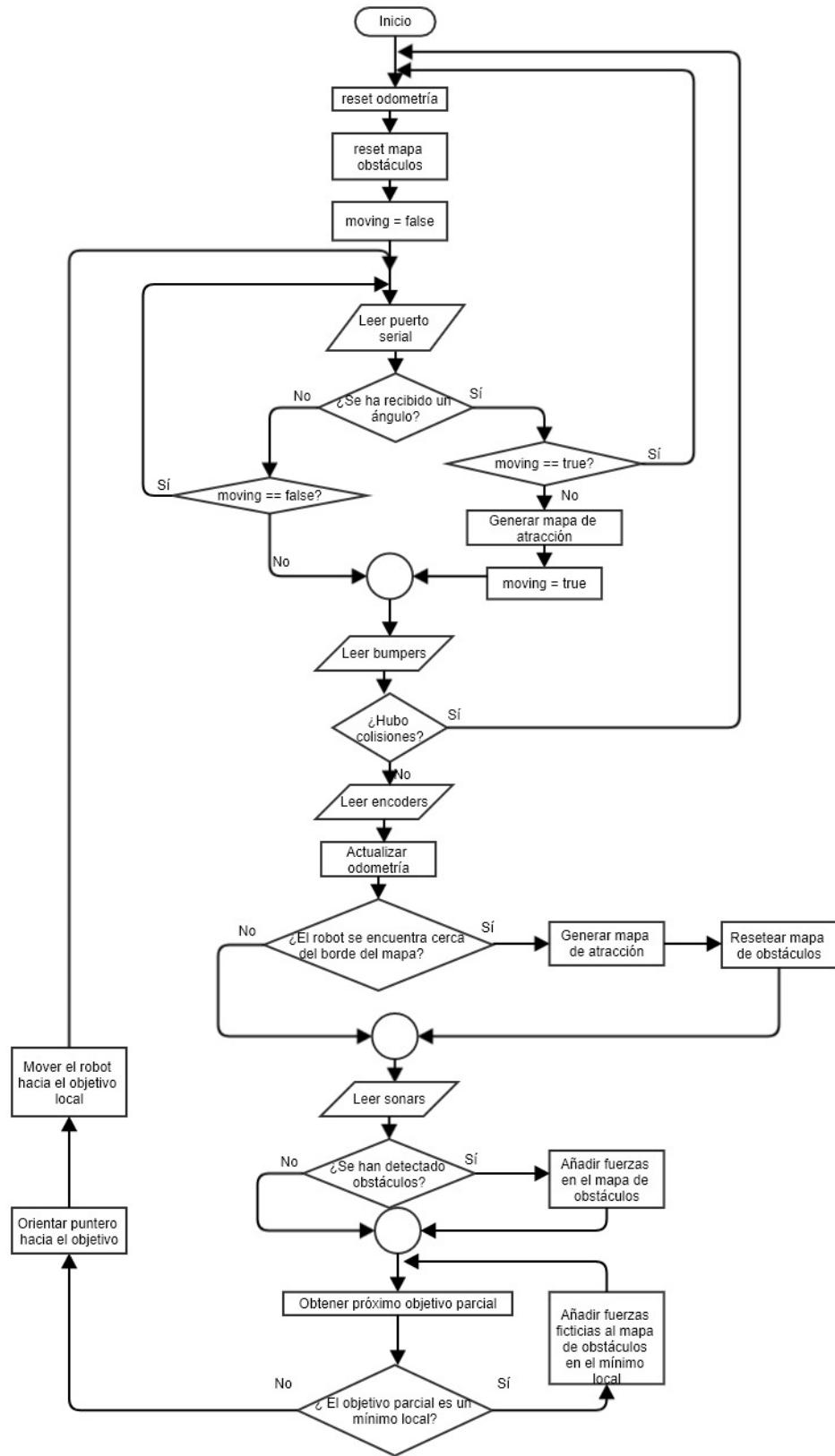


Figura 17: Diagrama de flujo del subsistema 02: navegación del robot

### 3.4.3 Estructuras de datos del subsistema 02

<i>MapPoint</i>	
Campos	<ul style="list-style-type: none"> <li>• <i>uint8_t x</i></li> <li>• <i>uint8_t y</i></li> </ul>
Descripción	La estructura <i>MapPoint</i> representa las coordenadas de un punto en el mapa de dirección o en el mapa de obstáculos.

<i>WorldPoint</i>	
Campos	<ul style="list-style-type: none"> <li>• <i>double x</i></li> <li>• <i>double y</i></li> </ul>
Descripción	La estructura <i>WorldPoint</i> representa un punto en el sistema de referencia.

<i>Position</i>	
Campos	<ul style="list-style-type: none"> <li>• <i>double x</i></li> <li>• <i>double y</i></li> <li>• <i>double theta</i></li> </ul>
Descripción	La estructura <i>Position</i> representa una posición en el sistema de referencia. A diferencia de <i>WorldPoint</i> , la estructura <i>Position</i> dispone del campo <i>theta</i> , el cual indica el ángulo de orientación.

<i>uint8_t direction_map [MAP_SIZE][MAP_SIZE]</i>	
Descripción	La matriz cuadrada <i>direction_map</i> , que tiene la función de mapa, incluye en cada elemento la distancia entre cada punto del mapa y el punto objetivo.

<b><i>uint8_t obstacles_map [MAP_SIZE][MAP_SIZE]</i></b>	
<b>Descripción</b>	La matriz cuadrada <b><i>obstacles_map</i></b> tiene la función de mapa, e incluye en cada elemento la fuerza de repulsión de los obstáculos detectados por los <i>sonars</i> , o las fuerzas ficticias generadas para evitar los mínimos locales.

### 3.4.4 Funciones del subsistema 02

A continuación, detallamos las funciones utilizadas por el subsistema de navegación o subsistema 02. El código completo se puede consultar en el Anexo 1 de este documento.

<b><u>void setup()</u></b>	
<b>Descripción</b>	<p>Como dijimos, la función <b><i>setup()</i></b> es una parte fundamental de los programas de <i>Arduino</i>. Es la primera función en ser ejecutada al iniciar la placa <i>Arduino Mega 2560</i>.</p> <p>En esta función se inicializan los puertos serie encargados de la comunicación con la placa <i>Arduino Due</i> y con el <i>PC</i>. Se inicializa, además, la librería <i>Wire</i>, que es la responsable de las comunicaciones <i>I2C</i> con los motores, los <i>encoders</i> y los <i>sonars</i>.</p> <p>Luego, se llama a la función <b><i>stopRobot()</i></b>, que detiene a los motores, pone a (0,0,0) la posición inicial del robot, e inicializa a cero los valores de la matriz de obstáculos. Además, se llama a la función <b><i>setSonarsTransform()</i></b>, que se encarga de definir la rotación y la traslación de cada <i>sonar</i> respecto al punto central del eje.</p> <p>A continuación, se inicializan el <i>array</i> de lecturas de los <i>sonars</i> y los temporizadores.</p>
<b>Llamada por</b>	--
<b>Llama a las funciones</b>	<ul style="list-style-type: none"> <li>• <b><i>Serial.begin(int baudRate)</i></b></li> <li>• <b><i>Serial1.begin(int baudRate)</i></b></li> <li>• <b><i>Wire.begin()</i></b></li> <li>• void <b><i>stopRobot()</i></b></li> <li>• void <b><i>setSonarsTransform()</i></b></li> </ul>

<u><b>void stopRobot()</b></u>	
<b>Descripción</b>	Llamando a la función <i>actuateMotors(0,0)</i> , del paquete <b>Magabot.h</b> , los motores se detienen de inmediato. Por su parte, <b>blinkBlue()</b> activa los <i>LED</i> de color azul. Posteriormente, se reinicia la posición del robot en <b>x = 0</b> , <b>y = 0</b> y el ángulo <b>θ</b> . Luego, llamando a la función <b>resetMapCenter()</b> , se define como centro del mapa al punto <b>(0,0)</b> . La función <b>resetObstaclesMap()</b> pone todos los valores de la matriz de obstáculos a cero. Al final, se pone a la variable <i>moving</i> el valor <i>false</i> .
<b>Llamada por</b>	<ul style="list-style-type: none"> <li>• <i>void setup()</i></li> <li>• <i>void checkSerial()</i></li> <li>• <i>void checkBumpers()</i></li> </ul>
<b>Llama a las funciones</b>	<ul style="list-style-type: none"> <li>• <i>Magabot.actuateMotors(int leftSpeed, int rightSpeed)</i></li> <li>• <i>void blinkBlue()</i></li> <li>• <i>void resetMapCenter()</i></li> <li>• <i>void resetObstaclesMap()</i></li> </ul>

<u><b>void setSonarsTransform()</b></u>	
<b>Descripción</b>	Esta función define la posición de cada <i>sonar</i> respecto al punto central del eje del robot. Estas medidas son útiles para aplicar la traslación y la rotación a los obstáculos detectados por los <i>sonars</i> .
<b>Llamada por</b>	<ul style="list-style-type: none"> <li>• <u><i>void setup()</i></u></li> </ul>
<b>Llama a las funciones</b>	--

<u><b>void loop()</b></u>	
<b>Descripción</b>	<p>La función <b>loop()</b> es una parte fundamental de los programas de <i>Arduino</i>. Como ya mencionamos, se ejecuta después de la función <b>setup()</b>, y se repite en un bucle infinito.</p> <p>En la función <b>loop()</b> tiene lugar el flujo de la navegación del robot.</p> <p>En cada iteración se ejecuta primero la función <b>checkLeds()</b>, la que, leyendo a un temporizador, controla si es necesario apagar los <i>LED</i>. Luego, la función <b>checkSerial()</b> comprueba si ha sido detectado algún sonido por la <i>Arduino Due</i>. En caso afirmativo, detiene al robot si éste está en movimiento. Si, por el contrario, el robot no está en movimiento, se inicializa la matriz de dirección y el robot inicia a navegar hacia el objetivo.</p> <p>En la función <b>loop()</b>, cuando el robot está en moviendo se ejecuta la función <b>checkBumpers()</b>, que sirve para detener el robot en caso de colisión. Posteriormente, se ejecuta la función <b>updateRobotPosition()</b> para actualizar la odometría. Después, la función <b>checkBorder()</b> comprueba si el robot se encuentra cerca de los límites del mapa. Si este es el caso, crea unos nuevos mapas de dirección y obstáculos, y posiciona al robot en el centro.</p> <p>Luego, la función <b>detectObstacles()</b>, a través de la lectura de los <i>sonars</i>, añade repulsión alrededor de los obstáculos.</p> <p>A continuación, la función <b>getNextPoint()</b> busca en un radio alrededor del robot el punto del mapa con un menor valor en la suma entre la matriz de dirección y la de obstáculos, y lo selecciona como objetivo local. La función <b>checkLocalMinimum()</b> comprueba si el punto se encuentra muy cercano al centro del eje del robot. En caso afirmativo, el robot se encuentra en un mínimo local. Se añade una fuerza de repulsión en la zona del robot y se vuelve a ejecutar la función <b>getNextPoint()</b>, hasta que el robot sale del mínimo local.</p> <p>Después, la función <b>pointGoal()</b> orienta el apuntador del robot hacia el objetivo, y finalmente la función <b>moveTo(nextPoint)</b> envía la orden a los motores de mover al robot hacia el objetivo local.</p>
<b>Llamada por</b>	--
<b>Llama a las funciones</b>	<ul style="list-style-type: none"> <li>• <b>void checkLEDs()</b></li> <li>• <b>void checkSerial()</b></li> <li>• <b>void checkBumpers()</b></li> <li>• <b>void updateRobotPosition()</b></li> <li>• <b>void checkBorder()</b></li> <li>• <b>void detectObstacles()</b></li> <li>• <b>WorldPoint getNextPoint()</b></li> <li>• <b>bool checkLocalMinimum(WorldPoint nextPoint)</b></li> <li>• <b>void pointGoal()</b></li> <li>• <b>void moveTo(WorldPoint nextPoint)</b></li> </ul>

<u><b>void checkLeds()</b></u>	
<b>Descripción</b>	La función <b><i>checkLeds()</i></b> comprueba si ha caducado el <b><i>timer ledsTimer</i></b> . En caso afirmativo, apaga los <b><i>LED</i></b> llamando a la función <b><i>Magabot.actuateLEDs(int r, int g, int b)</i></b> del paquete <b><i>Magabot.h</i></b> , con los valores de <b><i>r</i></b> , <b><i>g</i></b> y <b><i>b</i></b> iguales a cero.
<b>Llamada por</b>	<ul style="list-style-type: none"> <li>• void <b><i>loop()</i></b></li> </ul>
<b>Llama a las funciones</b>	<ul style="list-style-type: none"> <li>• void <b><i>Magabot.actuateLEDs(int r, int g, int b)</i></b></li> </ul>

<u><b>void checkSerial()</b></u>	
<b>Descripción</b>	<p>La función <b><i>checkSerial()</i></b> comprueba si hay mensajes disponibles en el puerto <b><i>Serial1</i></b>, usando la función <b><i>Serial1.available()</i></b>. En caso afirmativo, lee el ángulo enviado por la placa <b><i>Arduino Due</i></b>, aplicando la función <b><i>Serial1.read()</i></b>.</p> <p>Si se ha recibido un ángulo cuando el robot está en moviendo, se ejecuta la función <b><i>stopRobot()</i></b> para detenerlo. En el caso que el robot se haya encontrado detenido, se realizan las siguientes operaciones: la función <b><i>blinkRed()</i></b> enciende a los <b><i>LED</i></b> de color rojo. Se calcula el gradiente del ángulo recibido, posteriormente la función <b><i>movePointer(int soundAngle)</i></b> mueve el apuntador en dirección de la fuente sonora, y <b><i>setDirectionMap()</i></b> genera el mapa de dirección hacia el objetivo.</p>
<b>Llamada por</b>	<ul style="list-style-type: none"> <li>• void <b><i>loop()</i></b></li> </ul>
<b>Llama a las funciones</b>	<ul style="list-style-type: none"> <li>• <b><i>Serial1.available()</i></b></li> <li>• <b><i>Serial1.read()</i></b></li> <li>• void <b><i>stopRobot()</i></b></li> <li>• void <b><i>blinkRed()</i></b></li> <li>• void <b><i>movePointer(soundAngle)</i></b></li> <li>• void <b><i>setDirectionMap()</i></b></li> </ul>

<u><i>void checkBumpers()</i></u>	
<b>Descripción</b>	Esta función llama a la función <i>readBumpers()</i> del paquete <b><i>Magabot.h</i></b> , para comprobar si se ha activado alguno de los <i>bumpers</i> . En caso afirmativo, significa que el robot ha sufrido una colisión y se ejecuta la función <i>stopRobot()</i> , para detener su avance.
<b>Llamada por</b>	<ul style="list-style-type: none"> <li>• void <i>loop()</i></li> </ul>
<b>Llama a las funciones</b>	<ul style="list-style-type: none"> <li>• bool <i>Magabot.readBumpers</i></li> <li>• void <i>stopRobot()</i></li> </ul>

<u><i>void updateRobotPosition()</i></u>	
<b>Descripción</b>	Esta función es la encargada de la odometría. Después de leer los encoders con la función <i>readClicks(float * wheelsRotation)</i> , se utilizan las fórmulas de odometría descritas anteriormente para actualizar la posición del robot( $x, y, \theta$ ) en el sistema de referencia.
<b>Llamada por</b>	<ul style="list-style-type: none"> <li>• void <i>loop()</i></li> </ul>
<b>Llama a las funciones</b>	<ul style="list-style-type: none"> <li>• bool <i>Magabot.readClicks(float * wheelsRotation)</i></li> <li>• void <i>stopRobot()</i></li> </ul>

<u><b>void checkBorder()</b></u>	
<b>Descripción</b>	Esta función controla si el robot se encuentra en una de las cinco posiciones más cercanas al borde del mapa. En caso afirmativo, es necesario reposicionar los mapas de dirección y de obstáculos, para evitar que el programa intente acceder a posiciones de memoria no asignadas. Esta es, asimismo, una medida útil para reducir la acumulación de errores en la odometría. Inicialmente, se activan los <i>LED</i> en color verde con la función <i>blinkGreen()</i> . Luego, se llama a <i>resetMapCenter()</i> , que selecciona a la posición actual del Robot como punto central de los nuevos mapas. Posteriormente, <i>setDirectionMap()</i> vuelve a crear el mapa de dirección, usando la tangente del ángulo de origen del sonido. Finalmente, <i>resetObstaclesMap()</i> pone a cero todos los valores de la matriz de obstáculos.
<b>Llamada por</b>	<ul style="list-style-type: none"> <li>• void <i>loop()</i></li> </ul>
<b>Llama a las funciones</b>	<ul style="list-style-type: none"> <li>• <i>MapPoint getMapPoint(WorldPoint wp)</i></li> <li>• <i>void Magabot.actuateMotors(int leftSpeed, int rightSpeed)</i></li> <li>• <i>void blinkGreen()</i></li> <li>• <i>void resetMapCenter()</i></li> <li>• <i>void setDirectionMap()</i></li> <li>• <i>void resetObstaclesMap()</i></li> </ul>

<u><b>void detectObstacles()</b></u>	
<b>Descripción</b>	<p>Se ejecuta la función <i>getSonarReadings()</i> del paquete <i>Magabot.h</i>, que efectúa la lectura de los <i>sonars</i>.</p> <p>Si la última lectura se encuentra en un rango establecido, se efectúan las rotaciones y las traslaciones del punto detectado hacia el sistema de referencia. El punto obtenido es enviado a la función <i>addObstacle(WorldPoint obst_odom)</i>, que añade el obstáculo al mapa de obstáculos.</p>
<b>Llamada por</b>	<ul style="list-style-type: none"> <li>• void <i>loop()</i></li> </ul>
<b>Llama a las funciones</b>	<ul style="list-style-type: none"> <li>• <i>int8_t Magabot.getSonarReadings(float *sonarReadings)</i></li> <li>• <i>WorldPoint transformPoint(Position pos, WorldPoint p)</i></li> <li>• <i>void addObstacle(WorldPoint obst_odom)</i></li> </ul>

<u><i>WorldPoint getNextPoint()</i></u>	
<b>Descripción</b>	<p>Esta función es la encargada de encontrar el próximo objetivo local hacia el cual se debe mover el robot.</p> <p>Se busca, en un radio determinado alrededor del robot, el punto en el cual la suma de la matriz de dirección y la matriz de obstáculos es menor. La suma de las matrices en el punto es realizada a través de la función <code>getForce(MapPoint p)</code>.</p>
<b>Llamada por</b>	<ul style="list-style-type: none"> <li>• void <code>loop()</code></li> </ul>
<b>Llama a las funciones</b>	<ul style="list-style-type: none"> <li>• <code>MapPoint getMapPoint(WorldPoint p)</code></li> <li>• <code>uint16_t getForce(MapPoint p)</code></li> <li>• <code>WorldPoint getWorldPoint(MapPoint p)</code></li> </ul>

<u><i>bool checkLocalMinimum(MapPoint nextPoint)</i></u>	
<b>Descripción</b>	<p>Esta función es la responsable de detectar si el robot ha alcanzado un mínimo local. En ese caso, se genera una fuerza ficticia que aleja al robot. Para verificar si el robot se encuentra en un mínimo local, se mide la distancia euclídea entre el objetivo local y el centro del eje del robot, y se comprueba si esta es menor que 20cm. En caso afirmativo, se encienden los <i>LED</i> de color magenta, y se aumenta en una unidad el valor de la matriz de obstáculos en un radio de 20cm. La función devuelve un booleano que indica si el robot se encuentra en un mínimo local.</p>
<b>Llamada por</b>	<ul style="list-style-type: none"> <li>• void <code>loop()</code></li> </ul>
<b>Llama a las funciones</b>	<ul style="list-style-type: none"> <li>• <code>double euclideanDistance(WorldPoint p1, WorldPoint p2)</code></li> <li>• <code>void blinkPurple()</code></li> <li>• <code>MapPoint getMapPoint(WorldPoint p)</code></li> <li>• <code>WorldPoint getWorldPoint(MapPoint p)</code></li> </ul>

<u><b>void pointGoal()</b></u>	
<b>Descripción</b>	Esta función mueve el apuntador hacia el objetivo. Primero, calcula el ángulo entre la posición del robot y el objetivo. Luego, mueve el servo del apuntador con la función <b>movePointer(double goalAngle)</b> .
<b>Llamada por</b>	<ul style="list-style-type: none"> <li>• void <b>loop()</b></li> </ul>
<b>Llama a las funciones</b>	<ul style="list-style-type: none"> <li>• <b>double atan2(double y, double x)</b></li> <li>• <b>void movePointer(double goalAngle)</b></li> </ul>

<u><b>void moveTo(MapPoint nextPoint)</b></u>	
<b>Descripción</b>	<p>Esta función es la encargada de mover al robot en dirección del objetivo local. Inicialmente, se calcula el ángulo entre la posición del robot y el objetivo local.</p> <p>Se define la velocidad angular aplicada al robot, multiplicando la velocidad máxima por el seno del ángulo anteriormente calculado. La velocidad lineal es inversamente proporcional a la velocidad angular, y se calcula usando la siguiente fórmula: <b>linearSpeed = MAX_SPEED - MAX_SPEED * (abs(angularSpeed) / MAX_ANGULAR_SPEED)</b>. A continuación, se llama a la función <b>actuateMotors(int leftSpeed, int rightSpeed)</b> del paquete <b>Magabot.h</b>: <b>leftSpeed</b> es la velocidad lineal menos la velocidad angular, y <b>rightSpeed</b> es la suma entre la velocidad lineal y la velocidad angular.</p>
<b>Llamada por</b>	<ul style="list-style-type: none"> <li>• void <b>loop()</b></li> </ul>
<b>Llama a las funciones</b>	<ul style="list-style-type: none"> <li>• <b>double atan2(double y, double x)</b></li> <li>• <b>void Magabot.actuateMotors(int leftSpeed, int rightSpeed)</b></li> </ul>

<u><b>uint16_t getForce(MapPoint p)</b></u>	
<b>Descripción</b>	Esta función devuelve la suma de las matrices de dirección y de obstáculos en un punto determinado del mapa.
<b>Llamada por</b>	<ul style="list-style-type: none"> <li>• <u><b>WorldPoint getNextPoint()</b></u></li> </ul>
<b>Llama a las funciones</b>	--

<u><i>void movePointer(double angle)</i></u>	
<b>Descripción</b>	Esta función mueve el servo, para orientarlo en la dirección del ángulo que se le proporciona.
<b>Llamada por</b>	<ul style="list-style-type: none"> <li>• <u><i>void checkSerial()</i></u></li> <li>• <u><i>void pointGoal()</i></u></li> </ul>
<b>Llama a las funciones</b>	<ul style="list-style-type: none"> <li>• <i>Servo.write(int servoAngle)</i></li> </ul>

<u><i>void addObstacle(WorldPoint obstacle_worldPoint)</i></u>	
<b>Descripción</b>	<p>Esta función añade fuerzas al mapa de obstáculos, alrededor del punto que recibe como parámetro.</p> <p>Las fuerzas son almacenadas en 8 bits sin signo, con lo cual están incluidas en el rango [0, 255]. A la posición del mapa que incluye el obstáculo se pone el valor 255. En un determinado radio alrededor del obstáculo, se añade fuerza usando la fórmula:</p> $F = \text{round}((ROI - dist) * KFORCE / dist)$ <p><i>ROI</i> es el radio de influencia, <i>dist</i> la distancia entre el punto y el obstáculo, y <b>KFORCE</b> un coeficiente para incrementar la fuerza. El valor obtenido está acotado a 255.</p>
<b>Llamada por</b>	<ul style="list-style-type: none"> <li>• <i>void detectObstacles()</i></li> </ul>
<b>Llama a las funciones</b>	<ul style="list-style-type: none"> <li>• <i>MapPoint getMapPoint(WorldPoint obstacle_worldpoint)</i></li> <li>• <i>double euclideanDistance(WorldPoint wp1, WorldPoint wp2)</i></li> </ul>

<i>WorldPoint transformPoint(Position pos, WorldPoint p)</i>	
<b>Descripción</b>	Esta función realiza la traslación y rotación del punto recibido como segundo parámetro. El atributo <i>pos</i> es la posición del sistema de referencia de origen respecto al sistema de referencia de destino. La función aplica, primero, la rotación, y luego, la traslación del punto.
<b>Llamada por</b>	<ul style="list-style-type: none"> <li>• <i>void detectObstacles()</i></li> </ul>
<b>Llama a las funciones</b>	--

<i>double euclideanDistance(WorldPoint p1, WorldPoint p2)</i>	
<b>Descripción</b>	Calcula la distancia euclídea entre dos puntos.
<b>Llamada por</b>	<ul style="list-style-type: none"> <li>• <i>WorldPoint getNextPoint()</i></li> <li>• <i>bool checkLocalMinimum(WorldPoint nextPoint)</i></li> <li>• <i>void addObstacle(WorldPoint obstacle_worldPoint)</i></li> <li>• <i>void setDirectionMap()</i></li> <li>• <i>WorldPoint getNextPoint()</i></li> <li>• <i>bool checkLocalMinimum(WorldPoint nextPoint)</i></li> </ul>
<b>Llama a las funciones</b>	--

<i>void setDirectionMap()</i>	
<b>Descripción</b>	<p>Esta función inicializa el mapa de dirección alrededor del robot.</p> <p>En primer lugar, dependiendo el ángulo de origen del sonido, se calcula cuál es el punto más lejano en el mapa que esté en dirección del sonido, y se lo elige como objetivo. Sucesivamente, en cada casilla del mapa se almacena la distancia en decímetros entre el punto central de esta y el objetivo.</p>
<b>Llamada por</b>	<ul style="list-style-type: none"> <li>• <i>void checkSerial()</i></li> <li>• <i>void checkBorder()</i></li> </ul>
<b>Llama a las funciones</b>	<ul style="list-style-type: none"> <li>• <i>WorldPoint getWorldPoint(MapPoint p)</i></li> <li>• <i>double euclideanDistance(WorldPoint p1, WorldPoint p2)</i></li> </ul>

<i>WorldPoint getWorldPoint(MapPoint mPoint)</i>	
<b>Descripción</b>	Esta función devuelve el punto del sistema referencial, que se encuentra en las coordenadas del mapa recibidas como parámetro.
<b>Llamada por</b>	<ul style="list-style-type: none"> <li>• <i>WorldPoint getNextPoint()</i></li> <li>• <i>bool checkLocalMinimum( WorldPoint nextPoint)</i></li> <li>• <i>void addObstacle(WorldPoint obstacle_worldPoint)</i></li> <li>• <i>void setDirectionMap()</i></li> </ul>
<b>Llama a las funciones</b>	--

<i>MapPoint getMapPoint(WorldPoint wPoint)</i>	
<b>Descripción</b>	Esta función devuelve la posición del mapa en el cual se encuentra el punto del sistema referencial recibido como parámetro.
<b>Llamada por</b>	<ul style="list-style-type: none"> <li>• <i>void checkBorder()</i></li> <li>• <i>WorldPoint getNextPoint()</i></li> <li>• <i>bool checkLocalMinimum(WorldPoint nextPoint)</i></li> <li>• <i>void addObstacle(WorldPoint obstacle_worldPoint)</i></li> </ul>
<b>Llama a las funciones</b>	--

<i>void resetMapCenter()</i>	
<b>Descripción</b>	Esta función reposiciona el punto central de los mapas de dirección y de obstáculos, haciéndolos coincidir con el punto en el que se encuentra el robot, en ese mismo instante, en el sistema referencial.
<b>Llamada por</b>	<ul style="list-style-type: none"> <li>• <i>void stopRobot()</i></li> <li>• <i>void checkBorder()</i></li> </ul>
<b>Llama a las funciones</b>	--

<i>void blinkRed()</i>	
<b>Descripción</b>	Esta función llama a <i>actuateLEDs(int r, int g, int b)</i> del paquete <b><i>Magabot.h</i></b> , encendiendo los <i>LED RGB</i> de <i>Magabot</i> en color rojo. Se activa un temporizador de un segundo. Transcurrido este tiempo, la función <i>checkLEDs</i> apaga los <i>LED</i> .
<b>Llamada por</b>	<ul style="list-style-type: none"> <li>• <i>void checkSerial()</i></li> </ul>
<b>Llama a las funciones</b>	<ul style="list-style-type: none"> <li>• <i>Magabot.actuateLEDs(int r, int g, int b)</i></li> </ul>

<i>void blinkGreen()</i>	
<b>Descripción</b>	Esta función llama a <i>actuateLEDs(int r, int g, int b)</i> del paquete <b><i>Magabot.h</i></b> , encendiendo los <i>LED RGB</i> de <i>Magabot</i> en color verde. Se activa un temporizador de un segundo. Pasado ese tiempo, la función <i>checkLEDs()</i> apaga los <i>LED</i> .
<b>Llamada por</b>	<ul style="list-style-type: none"> <li>• <i>void checkBorder()</i></li> </ul>
<b>Llama a las funciones</b>	<ul style="list-style-type: none"> <li>• <i>Magabot.actuateLEDs(int r, int g, int b)</i></li> </ul>

<i>void blinkBlue()</i>	
<b>Descripción</b>	Esta función llama a <i>actuateLEDs(int r, int g, int b)</i> del paquete <b><i>Magabot.h</i></b> , encendiendo los <i>LED RGB</i> de <i>Magabot</i> en color azul. Luego, se activa un temporizador de un segundo. La función <i>checkLEDs()</i> apaga los <i>LED</i> , una vez transcurrido el segundo.
<b>Llamada por</b>	<ul style="list-style-type: none"> <li>• <i>void stopRobot()</i></li> </ul>
<b>Llama a las funciones</b>	<ul style="list-style-type: none"> <li>• <i>Magabot.actuateLEDs(int r, int g, int b)</i></li> </ul>

<i>void blinkPurple()</i>	
<b>Descripción</b>	Esta función llama a <i>actuateLEDs(int r, int g, int b)</i> del paquete <b><i>Magabot.h</i></b> , encendiendo los <i>LED RGB</i> de <i>Magabot</i> en color magenta. Se activa un temporizador de medio segundo. Transcurrido este tiempo, la función <i>checkLEDs()</i> apaga los <i>LED</i> .
<b>Llamada por</b>	<ul style="list-style-type: none"> <li>• <i>void stopRobot()</i></li> </ul>
<b>Llama a las funciones</b>	<ul style="list-style-type: none"> <li>• <i>Magabot.actuateLEDs(int r, int g, int b)</i></li> </ul>

### 3.5 DISEÑO DE LOS DRIVERS DE MAGABOT

El control de los sensores y de los actuadores de *Magabot* se realiza a través de un paquete de *drivers*, escrito en lenguaje *C++*, y compatible con las plataformas *Arduino*. El subsistema es modelado como una clase, e incluido en el paquete *Magabot.h* [el código de este paquete se puede consultar en el documento Anexo 1].

Los fabricantes de *Magabot* ofrecen en el repositorio *GitHub* un paquete de *drivers*, cuya estructura se desglosa en la Figura 18 [el repositorio de *drivers* originales de *Magabot* está disponible en el siguiente link: <https://github.com/Magabot/Magabot-Arduino-Library/tree/master/Magabot>]

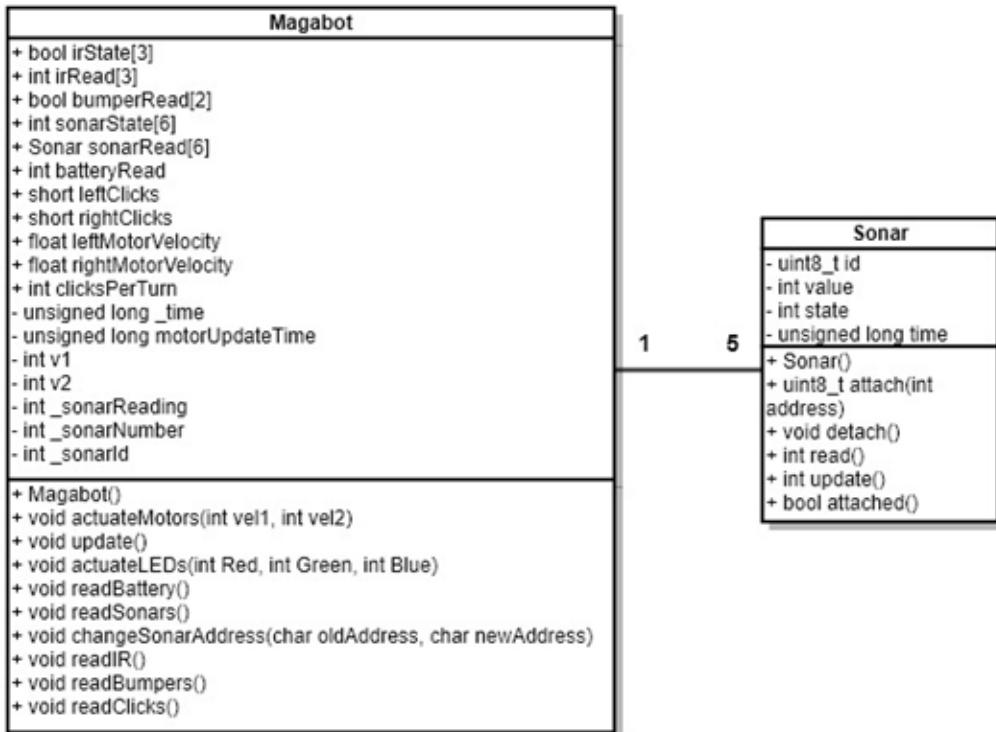


Figura 18: Clases del paquete de *drivers* de *Magabot* realizados por los fabricantes

Guiados con el propósito de simplificar el flujo de nuestro programa de navegación, hemos decidido aportar modificaciones a los *drivers* originales de *Magabot*. Por un lado, cambiamos la estructura de los métodos de la clase *Magabot* y, por otro, eliminamos aquellos elementos que no serán utilizados por nuestro programa como, por ejemplo, la clase *Sonar*. Por último, también limpiamos el código, con el fin de mejorar su legibilidad.

### 3.5.1 Clase *Magabot*

Después de nuestras modificaciones, la clase *Magabot* quedará estructurada como muestra el desglose de la siguiente tabla.

Clase	<i>Magabot</i>
<b>Atributos</b>	<ul style="list-style-type: none"> <li>- <i>_sonarId: int</i></li> <li>- <i>_ping: bool</i></li> <li>- <i>_sonarTimer: unsigned long</i></li> <li>- <i>_clicksPerTurn: int</i></li> </ul>
<b>Métodos</b>	<ul style="list-style-type: none"> <li>+ <i>Magabot()</i></li> <li>+ <i>void actuateMotors(int vel1, int vel2)</i></li> <li>+ <i>void actuateLEDs(int Red, int Green, int Blue)</i></li> <li>+ <i>uint8_t getSonarReadings(float * sonarReadings)</i></li> <li>+ <i>bool readBumpers()</i></li> <li>+ <i>void readClicks(float * wheelsRotation)</i></li> </ul>

### 3.5.2 Métodos del paquete *Magabot.h*

En este apartado describimos el funcionamiento de los métodos de la clase *Magabot*.

<i>Magabot::Magabot()</i>	
<b>Descripción</b>	<p>Se trata del constructor de la clase <i>Magabot.h</i>, que es llamado automáticamente la primera vez que se llama un método de la clase.</p> <p>Su función es la de definir los <i>pines</i> de los <i>LED</i> y de los <i>bumpers</i>, iniciar el protocolo <i>I2C</i>, definir la resolución de los <i>encoders</i> e iniciar el temporizador de los <i>sonars</i>.</p>

***void Magabot::actuateMotors(int vel1, int vel2)***

<b>Descripción</b>	Método público que sirve para mover los dos motores de <i>Magabot</i> . Se reciben como parámetros las velocidades de la rueda izquierda y de la rueda derecha.  El signo de la velocidad derecha es invertido, ya que el motor derecho está montado en dirección opuesta respecto al izquierdo. Posteriormente, a través de operaciones binarias, se obtienen el módulo y las direcciones de las dos velocidades.  Luego, se inicia la comunicación con el protocolo <i>I2C</i> a través de la función <i>Wire.beginTransmission(uint8_t address)</i> . En la que <i>address</i> es la dirección del motor.  Los valores de velocidad y dirección son escritos con <i>Wire.write(uint8_t value)</i> , y la conexión se cierra con <i>Wire.endTransmission()</i> .
--------------------	--

***void Magabot::actuateLEDs(int Red, int Green, int Blue)***

<b>Descripción</b>	Método público que sirve para encender los tres <i>LED RGB</i> de <i>Magabot</i> . Se reciben como parámetros las cantidades de luz roja, verde y azul que se quiere que los <i>LED</i> emitan.  Los valores son escritos en los puertos analógicas asociadas con los <i>LED</i> .
--------------------	--

	<i>uint8_t Magabot::getSonarReadings(float* sonarReadings)</i>
<b>Descripción</b>	<p>Método público que recibe como parámetro a un vector de tipo <i>float</i> con tamaño 5. Este vector debe contener las lecturas de los <i>sonars</i>. El método devuelve la posición del <i>sonar</i> que acaba de ser leído.</p> <p>La comunicación con los <i>sonars</i> se hace a través del protocolo <i>I2C</i>, usando la librería <i>Wire.h</i>.</p> <p>El proceso es el siguiente: primero, se realiza la conexión con un <i>sonar</i>, y se le pide que devuelva el resultado en centímetros. El <i>sonar</i> emite un pulso de ultrasonido y, luego, es necesario esperar al menos sesenta milisegundos antes de poder leer el resultado. Para no interrumpir la ejecución del resto del sistema, se lee solo un <i>sonar</i> a la vez. En la primera llamada de <i>getSonarsReadings()</i>, se pide al <i>sonar 0</i> que envíe el pulso de ultrasonidos. En la llamada sucesiva, si es necesario, se espera que hayan transcurrido sesenta milisegundos y, luego, se lee el resultado del <i>sonar 0</i>. A continuación, se envía la petición al <i>sonar 1</i> y habrá que esperar el tercer llamado de la función para leer el <i>sonar 1</i> y enviar la petición al <i>sonar 2</i>. Este proceso se repite para los cinco <i>sonars</i>. Alterando peticiones y lecturas de los <i>sonars</i>, se evita mantener el sistema en espera.</p>
	<i>bool Magabot::readBumpers()</i>

	<b><i>void Magabot::readClicks(float * wheelsRotation)</i></b>
<b>Descripción</b>	<p>Método público que sirve para medir la rotación de las ruedas, a través de la lectura de los dos <i>encoders</i>.</p> <p>La lectura se hace a través del protocolo <i>I2C</i>, utilizando el paquete <b>Wire.h</b>.</p> <p>El <i>encoder</i> proporciona la información de cuantas marcas ha leído en un período de tiempo determinado. Conociendo la resolución del <i>encoder</i>, que en este caso es de 3900 marcas por giro, se puede calcular la rotación realizada por el eje usando la fórmula:</p> $\text{rotación} = \text{marcas} * 2\pi / \text{resolución}.$ <p>Dado que el <i>encoder</i> izquierdo está montado al revés, hay que invertir el signo del resultado de éste.</p> <p>El método recibe como parámetro un vector de tipo <i>float</i> de tamaño 2. En este vector se almacena el resultado, que es la rotación que ha tenido cada una de las dos ruedas motrices.</p>

### 3.6 DISEÑO DE LOS *DRIVERS* CON *SIMULINK*

Una de las principales ventajas del uso de *MATLAB Simulink* es que permite unificar los procesos de diseño y de implementación. En el entorno de desarrollo de *Simulink* se generan diseños basados en modelos. Estos diseños pueden ser convertidos a código *C* o *C++*, y pueden ejecutarse desde la misma interfaz de *MATLAB Simulink* en los sistemas embebidos.

En este apartado procedemos a mostrar cómo hemos diseñado los modelos *Simulink* para su ejecución en la placa *Arduino Mega 2560*. Más adelante, concretamente en el apartado “4.3 Realización de los *drivers* con *MATLAB Simulink*”, ofreceremos una descripción detallada del proceso de creación de estos modelos, que se desarrollará en la fase de implementación.

En *Simulink* existen cuatro tipos de ejecución de un modelo: *normal*, *Software In the Loop (SIL)*, *External Mode* y *Processor In the Loop (PIL)*. Los primeros dos tipos hacen que el código se ejecute como simulación en la máquina de desarrollo con *MATLAB Simulink*, mientras que los dos últimos, ejecutan el programa en el *hardware* externo, que es objeto de nuestro desarrollo embebido.

El *profiling*, es decir, la medición de los tiempos de ejecución del código se puede realizar solo sobre el código ejecutado en la máquina de desarrollo. *Simulink* permite incluir dentro de un modelo que se ejecuta en modo *normal*, otro modelo que funciona en modo *PIL*, mediante la funcionalidad *Model Reference*.

En la Figura 19, podemos observar el diseño de alto nivel de la estructura que tendrán los controladores de *Magabot* realizados con modelos de *MATLAB Simulink*.

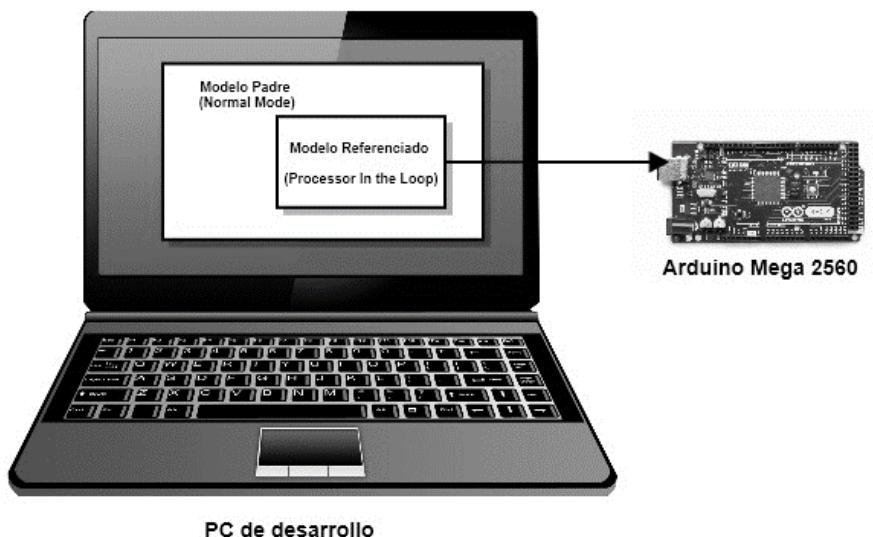


Figura 19: Diseño de alto nivel de los *drivers* de *Magabot* con *MATLAB Simulink*

### 3.6.1 Diseño de los *drivers* de control de los motores de *Magabot*

En *MATLAB Simulink* debemos crear un modelo padre, al que llamaremos *motorsSimulink* (Figura 20). Dentro de este modelo padre, creamos, a su vez, un modelo referenciado, que recibirá el nombre de *motorsSimulinkRef* (Figura 21). Este último modelo es ejecutado dentro de la placa *Arduino Mega 2560*, en modo *Processor In the Loop*.

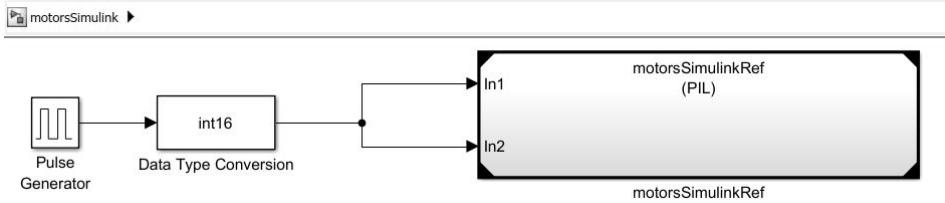


Figura 20: Diseño del modelo *motorsSimulink*

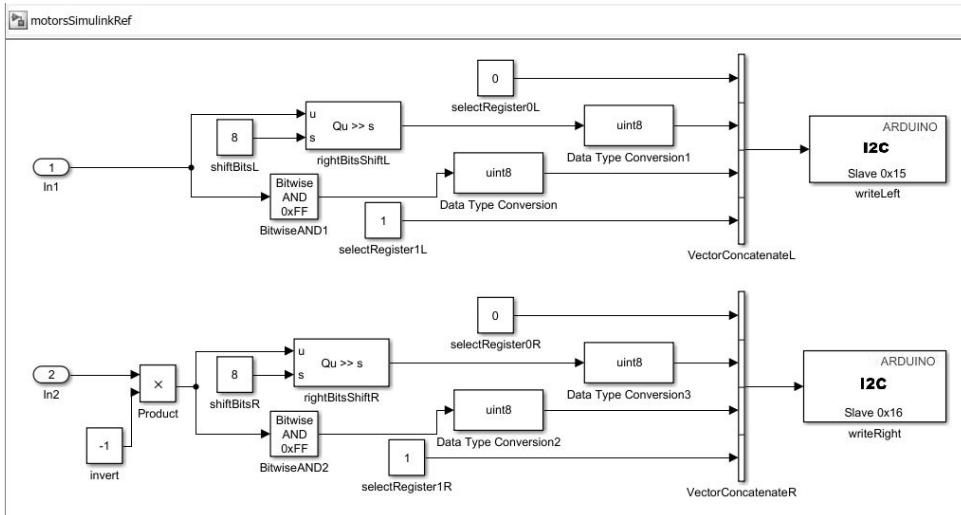


Figura 21: Diseño del modelo *motorsSimulinkRef*

En la fase de implementación, elaboraremos una descripción detallada sobre este proceso, que será incluida en el apartado “4.3.1 Modelado de los *drivers* de control de los motores de *Magabot* con *Simulink*”.

### 3.6.2 Diseño de los *drivers* de lectura de los *encoders* de *Magabot*

Para diseñar los *drivers* de los *encoders* de *Magabot* con *MATLAB Simulink*, utilizamos un procedimiento parecido al que hemos empleado en el diseño de los *drivers* de control de los motores. Para comenzar, diseñamos el modelo *encodersSimulink* (Figura 22), que se ejecutará en el *PC* de desarrollo en modo *normal*. Este modelo debe llamar al modelo referenciado *encodersSimulinkRef* (Figura 23), que se ejecutará en la placa *Arduino Mega 2560*, en modo *Processor In the Loop*.

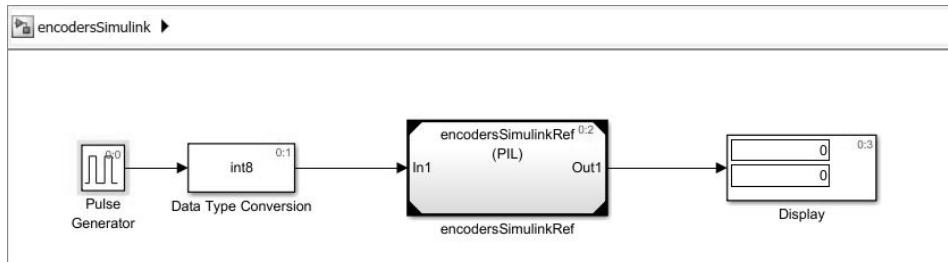


Figura 22: Diseño del modelo *encodersSimulink*

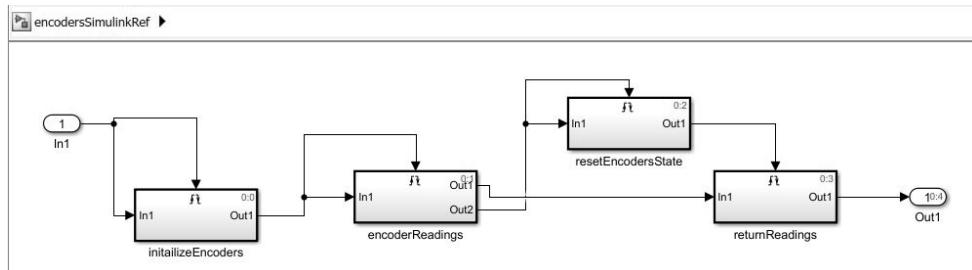


Figura 23: Diseño del modelo *encodersSimulinkRef*

En el momento de llevar a cabo este proceso en la fase de implementación, se confeccionará una guía que desglose cada uno de los pasos seguidos. Esta descripción detallada se incluirá en el apartado titulado “4.3.2 Modelado de los *drivers* de lectura de los *encoders* de *Magabot* con *Simulink*”.



# Implementación





## 4 IMPLEMENTACIÓN

---

En este apartado describimos el proceso de realización de nuestro sistema de navegación, considerando tanto el desarrollo del *software* como los elementos vinculados a la construcción física del robot.

Luego, elaboramos una guía de los pasos que seguimos en el *software MATLAB Simulink* para realizar los modelos de los *drivers* de los motores y de los *encoders* de *Magabot*. Posteriormente, analizamos el rendimiento en la placa *Arduino Mega 2560* del código generado por *MATLAB Embedded Coder*, a partir de los modelos *Simulink*.

### 4.1 CONSTRUCCIÓN DEL HARDWARE

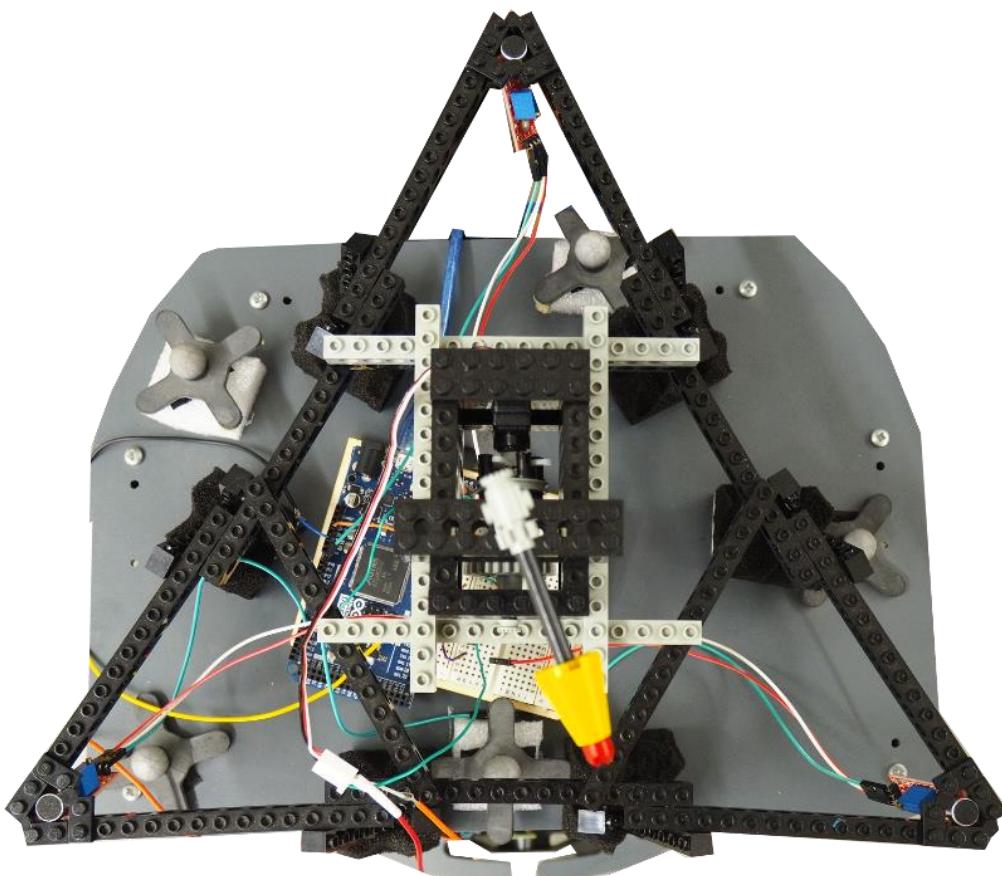
Para la realización del trabajo se dispone de la plataforma móvil *Magabot*, un robot de configuración diferencial.

Como se observa en la Fotografía 13, el *Magabot* dispone también de un soporte con dos estantes, que permiten apoyar accesorios como, por ejemplo, una *netbook*.



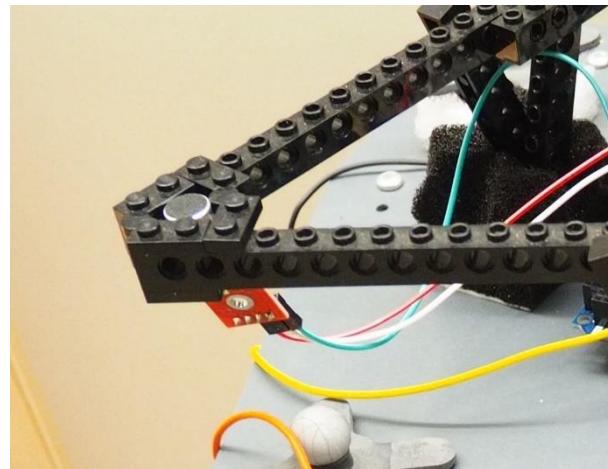
Fotografía 13: *Magabot* con soportes

Para detectar el origen de un sonido, como ya señalamos, hemos montado en la zona superior del robot tres micrófonos *KY-037*. La instalación de los micrófonos se realiza sobre una estructura construida con piezas de *Lego Technic* (Fotografía 14).



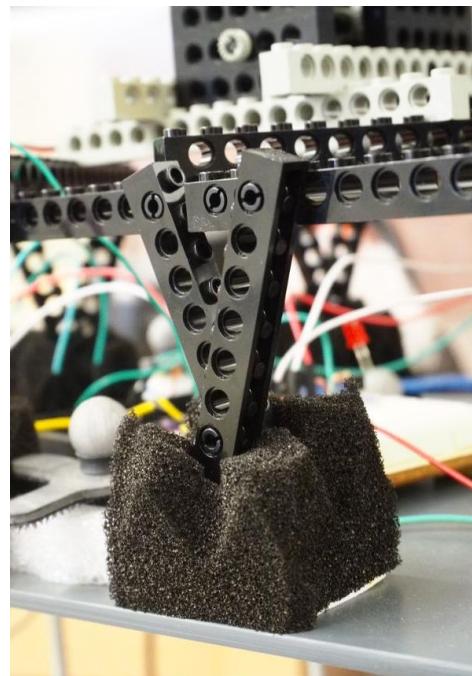
Fotografía 14: Estructura superior del robot

Como podemos observar en la fotografía anterior, los micrófonos están dispuestos en las esquinas de un triángulo equilátero, con una distancia de 31 cm entre sí. Cabe aclarar que, para la elaboración de la estructura, no ha sido necesario soldar ni pegar ninguna pieza (Fotografía 15).



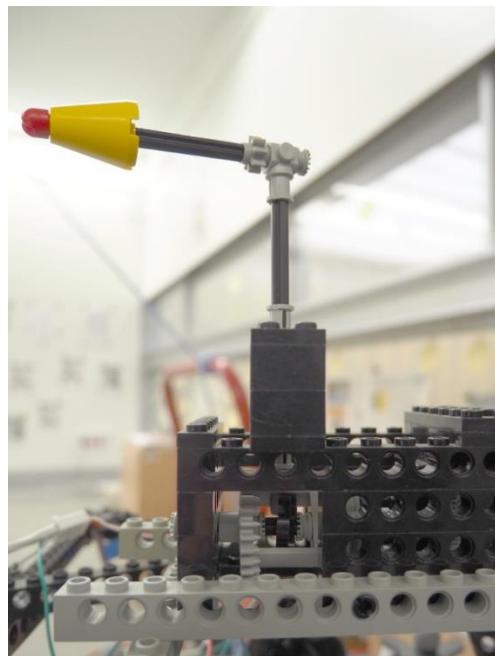
Fotografía 15: Detalle del micrófono enganchado en la estructura de *Lego*

La estructura de *Lego* se apoya en seis pedazos de goma espuma que, al funcionar como amortiguadores, evitan que las vibraciones producidas por el movimiento del robot afecten las lecturas de los micrófonos (Fotografía 16).

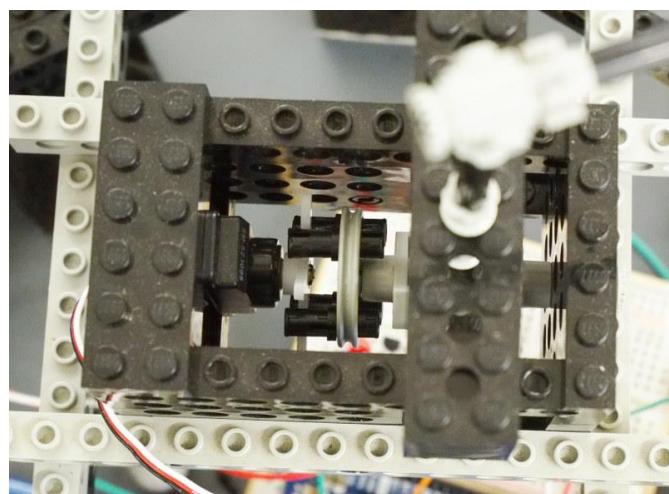


Fotografía 16: Soportes en goma espuma

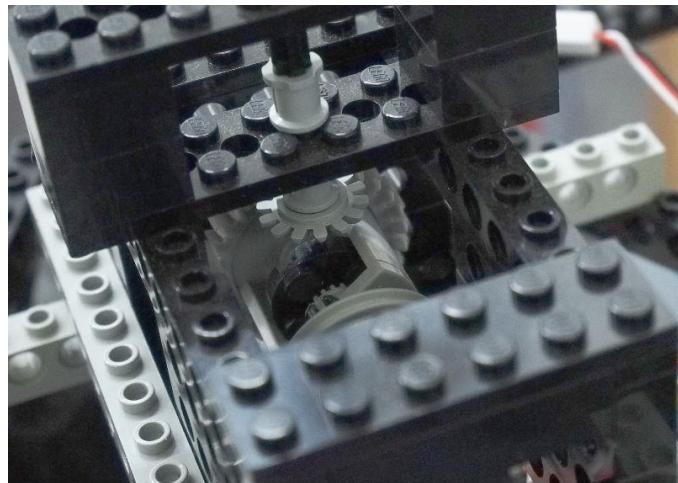
En el centro de la estructura de micrófonos colocamos un apuntador, construido también con piezas de *Lego Technic*, que es utilizado para señalar hacia la dirección de origen del sonido (Fotografía 17). Un motor servo de 180° de rotación es el encargado de mover el apuntador (Fotografía 18). Para que el apuntador pueda rotar a 360°, hemos construido un sistema de engranajes con relación de transmisión de 2:1 (Fotografía 19). Esta relación se obtiene uniendo al eje del servo un engranaje que posea el doble de dientes del engranaje ensamblado al eje del apuntador.



Fotografía 17: Apuntador

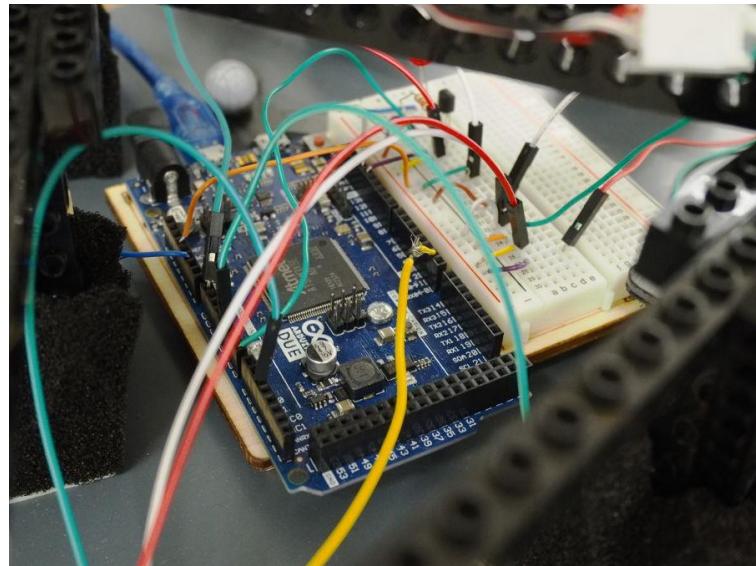


Fotografía 18: Servo del apuntador



Fotografía 19: Engranajes del apuntador

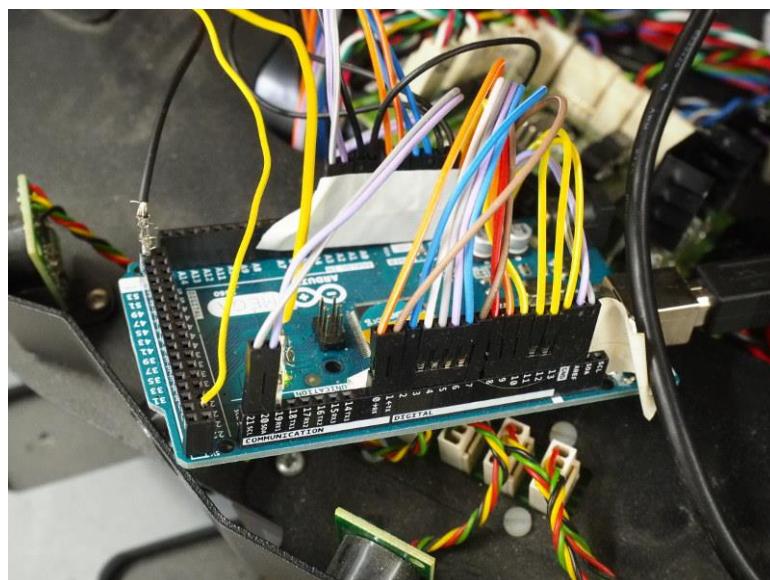
Debajo de la estructura de Lego, se encuentra la placa *Arduino Due*, que tiene la función de realizar la lectura de los micrófonos, calcular el ángulo de origen del sonido y enviarlo a la placa *Arduino Mega 2560*. Las conexiones entre la placa *Arduino Due*, los micrófonos y el termómetro, se realizaron utilizando una placa *solderless breadboard* y cables de tipo *Dupont*, que no necesitan soldaduras (Fotografía 20).



Fotografía 20: *Ardino Due y breadboard*

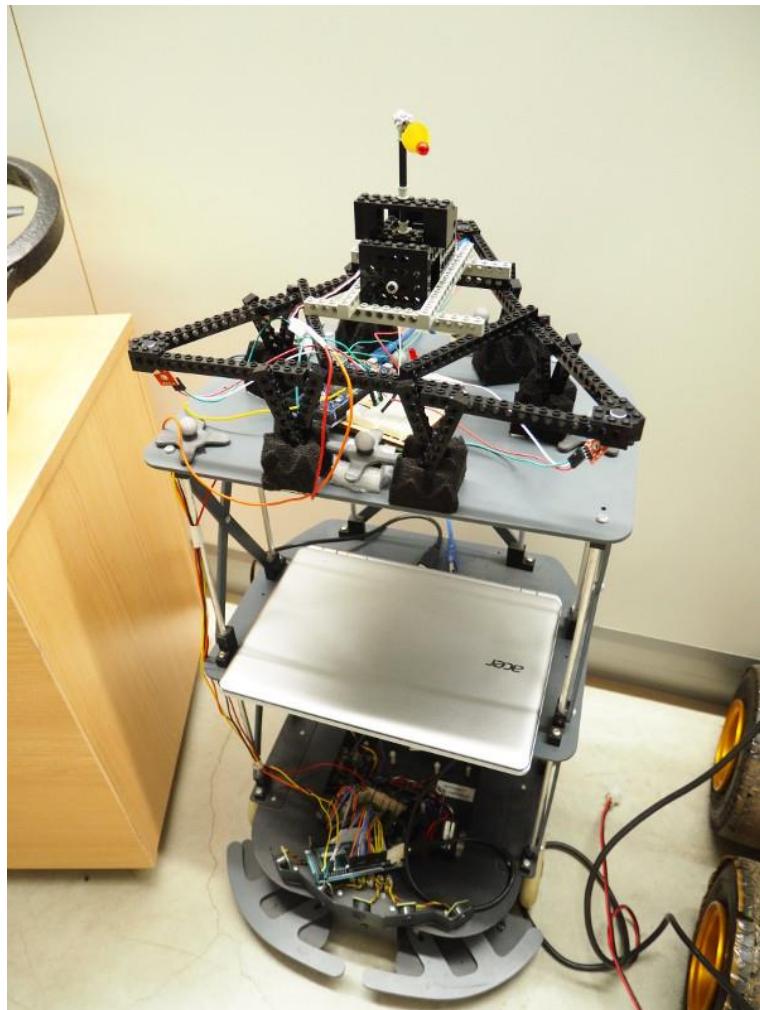
La placa *Arduino Uno*, originariamente montada en *Magabot*, ha sido sustituida por una *Arduino Mega 2560*, ya que esta última, como dijimos, tiene más puertos serie y más memoria.

Sin embargo, la sustitución de la placa original requiere la realización de una adaptación, dado que la *shield* de *Magabot* efectúa la conexión del protocolo *I2C* a través de los puertos *A4 (SDA)* y *A5 (SCL)* de la *Arduino Uno*, mientras que en la placa *Arduino Mega 2560* estos puertos están localizados en los *pines 20 (SDA)* y *21 (SCL)*. Para poder salvar este inconveniente y lograr la adaptación de la *shield* a la nueva placa, hemos utilizado cables *Dupont*, como podemos apreciar en la siguiente la Fotografía 21.



Fotografía 21: Conexión entre placa *Arduino Mega 2560* y *shield*

Una vez terminado el montaje del *hardware*, el robot luce el siguiente aspecto (Fotografía 22).



Fotografía 22: Instalación completa del robot

## 4.2 IMPLEMENTACIÓN DEL SOFTWARE

Realizamos el código de las placas *Arduino* usando el entorno de desarrollo *Arduino IDE*. Esta herramienta nos permite editar el código, subirlo a las placas *Arduino*, incluir paquetes y visualizar los mensajes enviados por puerto serie. Recurrimos a esta última función para depurar el código y para visualizar resultados de la ejecución. Para la realización de los *drivers* en lenguaje *C++*, usamos el editor de texto *Notepad++*.

En la Figura 24, utilizamos *MATLAB* para generar una gráfica donde se representan las lecturas producidas por los tres micrófonos al momento de recibir un pulso sonoro –en este caso, una palmada–. La gráfica muestra con unas estrellas el instante en el que el umbral es superado por cada uno de los micrófonos.

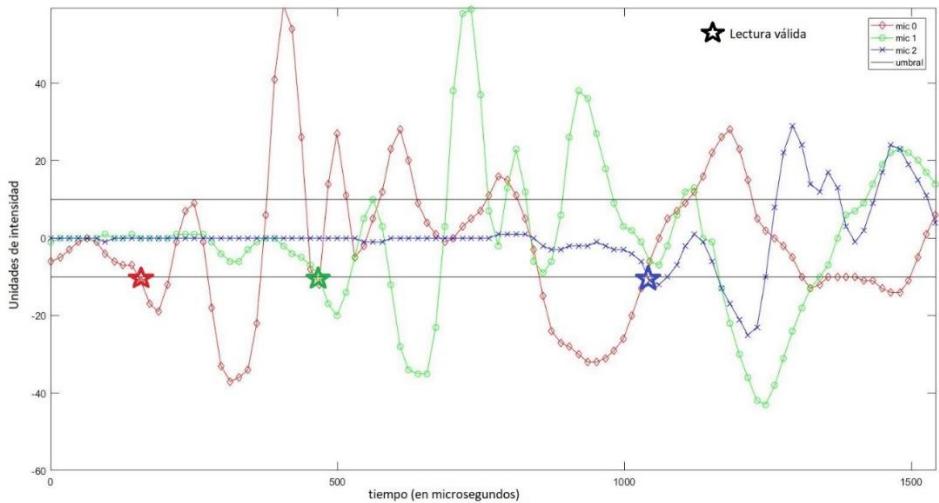


Figura 24: Lecturas de los tres micrófonos al detectar un pulso sonoro

Las lecturas de los micrófonos número 0, 1 y 2 se produjeron a los 157, 484 y 1044 microsegundos, respectivamente, desde el inicio de la grabación. Las dos lecturas más cercanas en el tiempo han sido la del micrófono 0 y la del micrófono 1. Por ello, esos micrófonos serán utilizados para el cálculo del ángulo.

Para calcular el ángulo de origen del sonido recurrimos a la fórmula:

$$\text{angle} = \text{asin}((\text{recDelay} * \text{sound\_speed}) / \text{mic\_distance}) * \text{frontRear} + \text{angleOffset}.$$

En este código, *recDelay* es el tiempo de llegada del sonido al micrófono 0, menos el tiempo de llegada al micrófono 1.

*frontRear* equivale a uno cuando el micrófono no utilizado para el cálculo del ángulo ha sido leído después de los que han sido seleccionados. En caso contrario, es decir, cuando el micrófono ha sido leído antes, *frontRear* equivale a -1.

*angleOffset* es la rotación que se debe aplicar al ángulo. Esa rotación, como ya mencionamos, depende de cuáles han sido los micrófonos seleccionados. Cuando los micrófonos seleccionados son el 0 y el 1, *angleOffset* equivale a cero. Cuando los micrófonos elegidos son el 1 y el 2, *angleOffset* es de  $2\pi/3$  radianes. Cuando los micrófonos son el 2 y el 1, la rotación aplicada es de  $-2\pi/3$  radianes. Además, en el caso de que el primer micrófono en haber sido leído no resulta utilizado para calcular el ángulo, se aplica una ulterior rotación de  $\pi$  radianes al ángulo.

Durante la realización de esta demostración, el termómetro registró una temperatura de 25°C. Por ello, la velocidad del sonido dada por la fórmula  $\text{sound\_speed} = \sqrt{k * R * T}$  tuvo el resultado: 246 m/s.

Para calcular el ángulo, entonces, realizamos el siguiente cálculo:  $\text{angle} = \arcsin(-327*246 / 310000) = -0,2625 \text{ radianes}$ . Por lo tanto, el ángulo de origen del pulso sonoro es de -15°, aproximadamente.

Para estudiar los resultados obtenidos durante la ejecución del algoritmo de navegación, a través del puerto serie, enviamos al PC la matriz de orientación y la matriz de obstáculos. Dichas matrices son respectivamente usadas por el robot para orientarse hacia el objetivo y evitar colisiones. En las siguientes figuras usamos la función *surf()* de MATLAB para representar los valores de ambas matrices.

En la Figura 25, podemos ver la matriz de dirección, que se genera al detectar el ángulo de origen del pulso sonoro. Como se puede observar, el campo generado tiene la forma de semicono, con pendiente hacia el punto objetivo.

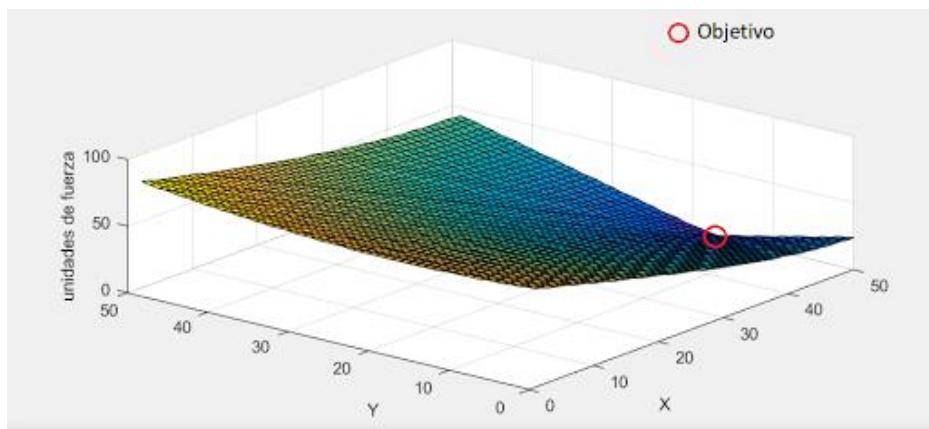


Figura 25: Representación gráfica de la matriz de dirección

La Figura 26 es una gráfica del mapa de obstáculos que se genera al posicionar una caja delante del robot. En esta figura podemos observar una fuerza repulsiva alrededor de los puntos en los cuales los *sonars* han detectado el objeto.

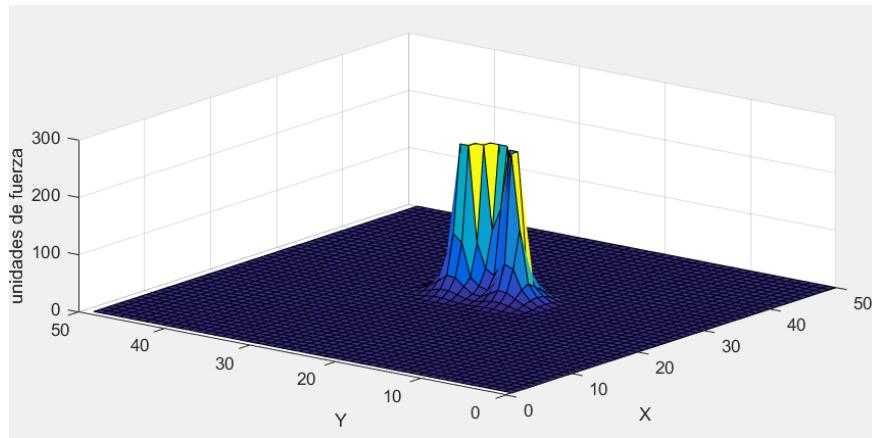


Figura 26: Representación gráfica de la matriz de obstáculos

En la Figura 27 se grafica la suma de las dos matrices. Para navegar, el robot busca en un radio determinado el punto con menor fuerza, lo elige como objetivo local y dirige su movimiento hacia él. Es necesario que el radio de búsqueda no sea demasiado amplio, para evitar que el robot fije como objetivo local un punto ubicado detrás de un obstáculo, lo que causaría una colisión. Sin embargo, el radio de búsqueda tampoco puede ser demasiado corto, ya que el robot avanzaría de forma brusca y con cambios de direcciones continuos. Después de varias pruebas, hemos determinado el tamaño adecuado del radio de búsqueda del objetivo local en 50 cm.

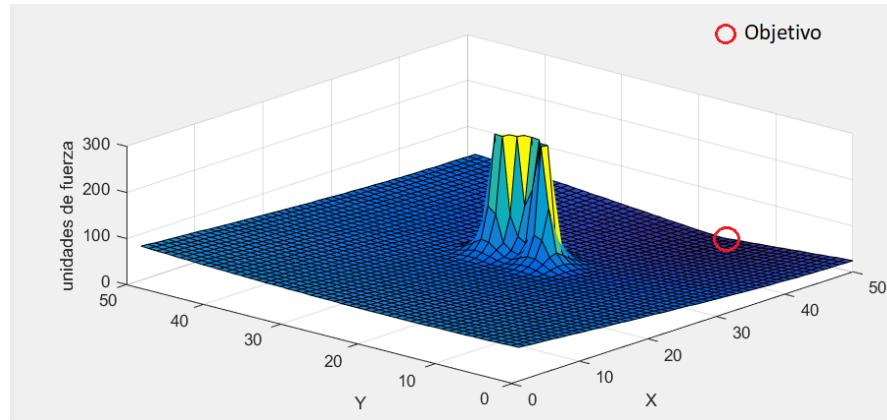


Figura 27: Suma de las matrices de dirección y obstáculos

Al encontrarse frente a un punto mínimo local, el robot no puede avanzar, ya que elige como objetivo local a su propia ubicación. En este caso, se genera una fuerza ficticia que va incrementando de una unidad el valor de la matriz de obstáculos, en un radio de 20 cm alrededor del robot. Ese

incremento se repite hasta que el robot es impulsado fuera de la zona de mínimo local, y puede así esquivar el obstáculo y seguir con la navegación. En la Figura 28, se ha generado una fuerza ficticia en proximidad del obstáculo, la cual ha impulsado al robot hacia la izquierda, saliendo así del mínimo local en el que se encontraba.

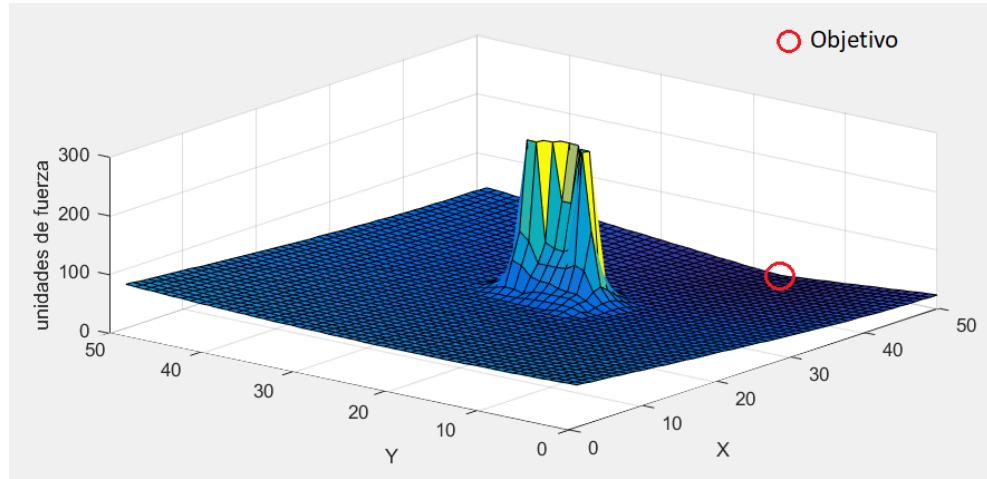


Figura 28: Suma de las matrices tras la detección de un mínimo local

### **4.3 REALIZACIÓN DE LOS *DRIVERS* CON *MATLAB SIMULINK***

En este apartado, elaboramos una guía en la que describimos el proceso de creación de modelos *Simulink* para *Arduino*. Asimismo, mostramos los resultados obtenidos durante el análisis de la ejecución del código en la placa *Arduino Mega 2560*.

*MATLAB Simulink* es un entorno de desarrollo que permite generar diagramas de bloques para representar procesos. Dichos procesos pueden ser ejecutados en un entorno de simulación o en dispositivos embebidos. Los bloques, que pueden tener *inputs* u *outputs*, pueden ser generadores de señales, operadores lógicos, funciones personalizadas o subsistemas enteros.

#### **4.3.1 Introducción al uso de *Simulink* con *Arduino***

Para crear modelos de *Simulink* para *Arduino* disponemos del software *MATLAB 2106b*, y de los paquetes *Simulink* y *Embedded Coder*. Desde el

menú horizontal superior *Home de MATLAB*, accedemos al menú *Add Ons* (Figura 29), para descargar e instalar el paquete de *drivers* de *MATLAB* para placas *Arduino*, y posteriormente el paquete de modelos de *Simulink* para *Arduino* (Figura 30).



Figura 29: Menú *Home de MATLAB*

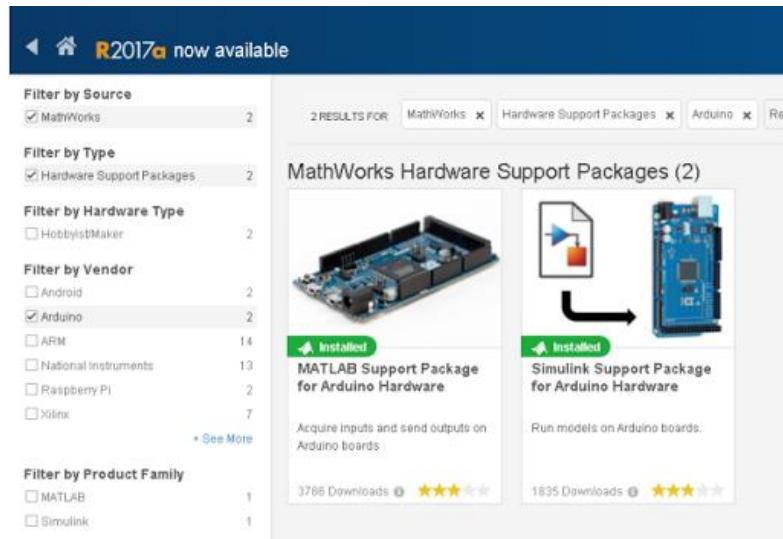


Figura 30: Paquetes de *MATLAB* y *Simulink* para placas *Arduino*

Una vez instalados los *drivers*, podemos abrir *Simulink*, desde el menú *Home de MATLAB*, y crear un nuevo modelo. Para ello, debemos seleccionar la opción *Blank Model* en la pestaña *Simulink* (Figura 31).

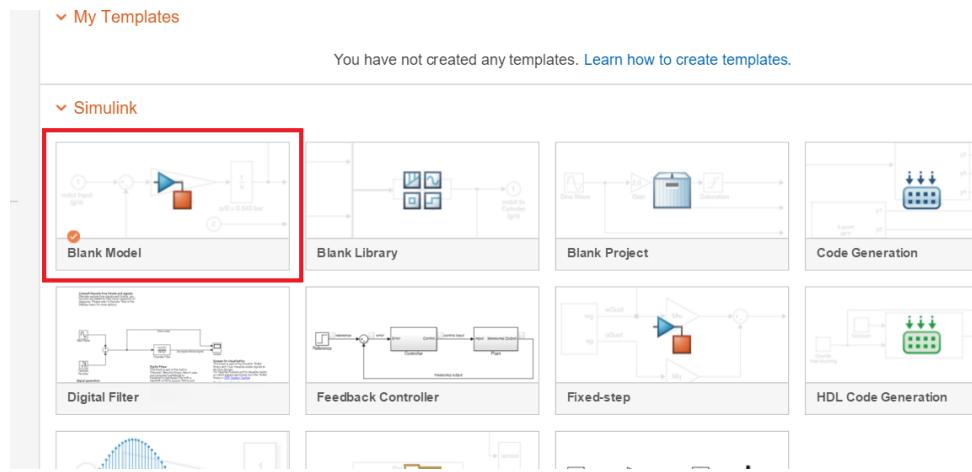


Figura 31: Opción *Blank Model*

Desde el nuevo modelo de *Simulink*, seleccionamos el ícono de configuración (⚙️) en el menú principal. Luego, en el menú de la izquierda vamos a *Hardware Implementation*. Desde allí, seleccionamos el campo *Hardware Board*, y abrimos el menú desplegable para elegir la placa *Arduino Mega 2560* (Figura 32).

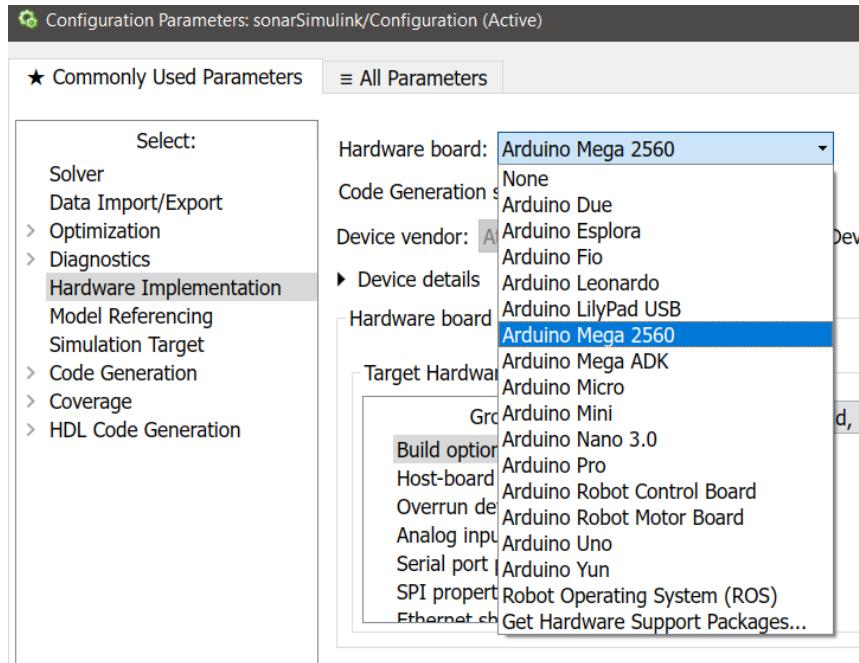


Figura 32: Campo *Hardware Board*

Para configurar las opciones del *Solver*, definimos unos pasos de simulación fijos de 0.01 y dejamos que el programa elija automáticamente el *Solver*. Podemos considerar la función de los pasos de simulación de manera

análoga a los pasos de integración en el cálculo diferencial, es decir, cuanto más pequeños sean los pasos de simulación, más precisa será la simulación. Podemos elegir los tiempos de inicio (*Start Time*) y de fin de la simulación (*Stop Time*), en segundos. Poniendo el valor *inf* en el campo *Stop Time*, la ejecución de la simulación deberá ser terminada manualmente (Figura 33).

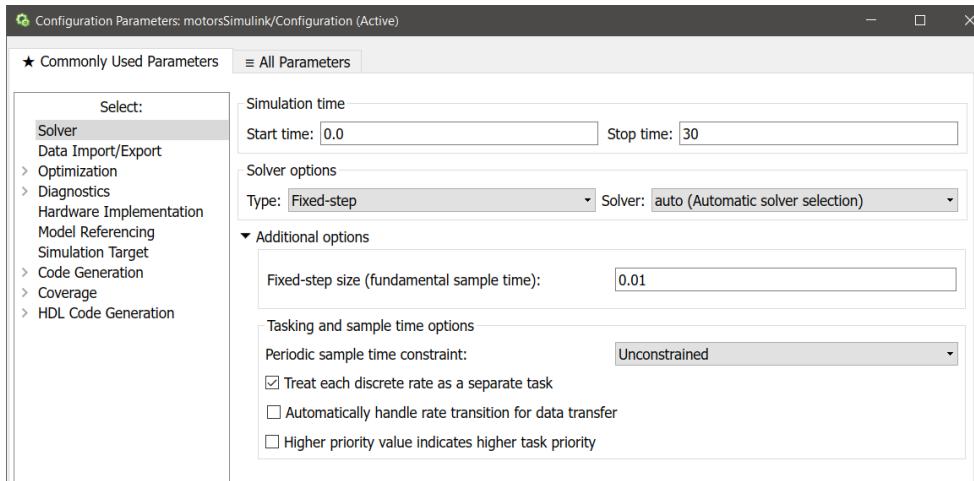


Figura 33: Configuración del *Solver*

Para diseñar los modelos de *Simulink* debemos usar una biblioteca de bloques, que se abre desde el menú principal seleccionando la opción *Library browser* (Figura 34).

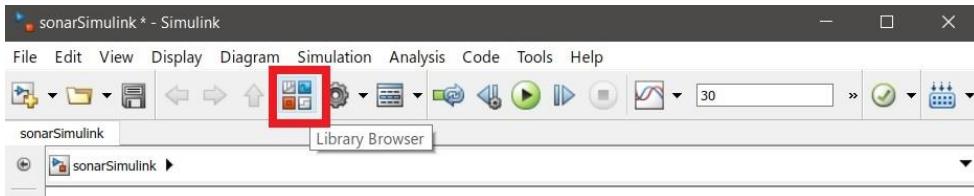


Figura 34: Opción *Library browser*

Esta biblioteca dispone de bloques para fuentes, resultados, conectores y componentes lineales y no lineales. Ofrece, además, la posibilidad de crear bloques con funciones personalizadas (Figura 35)

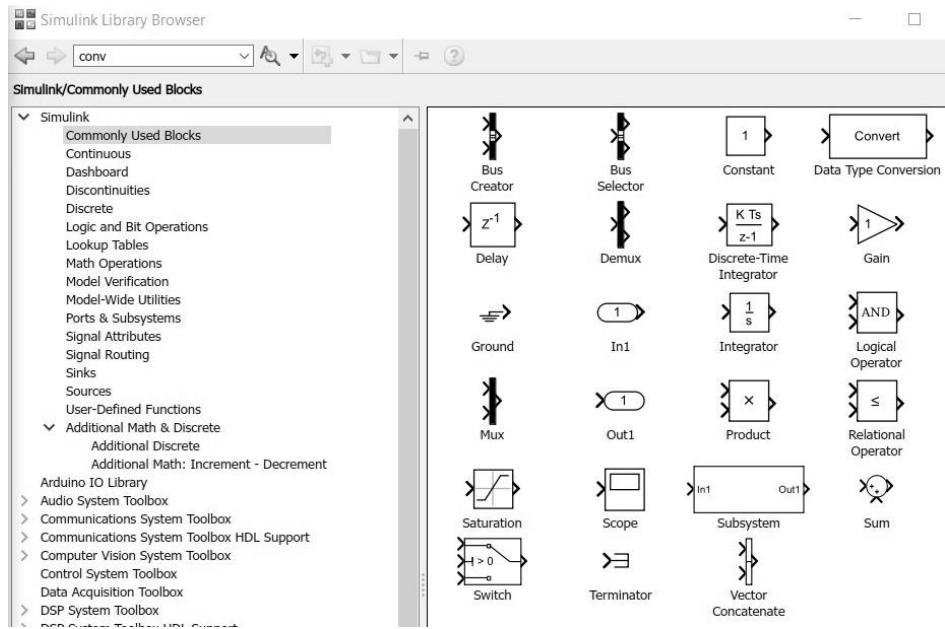


Figura 35: *Simulink/Commonly Used Blocks*

En esta biblioteca, encontramos disponible el paquete de bloques para *Arduino*, que acabamos de instalar: *Simulink Support Package for Arduino Hardware/Common* (Figura 36). En este paquete, encontramos varias opciones de bloques. Mientras que algunos de estos bloques sirven para leer los *inputs* o enviar señales a los puertos analógicos y digitales, otros se encargan de realizar las comunicaciones con los puertos serie, con los protocolos *I2C*, o *SPI*. También, encontramos bloques que sirven para generar señales *PWM* o comandar motores servo.

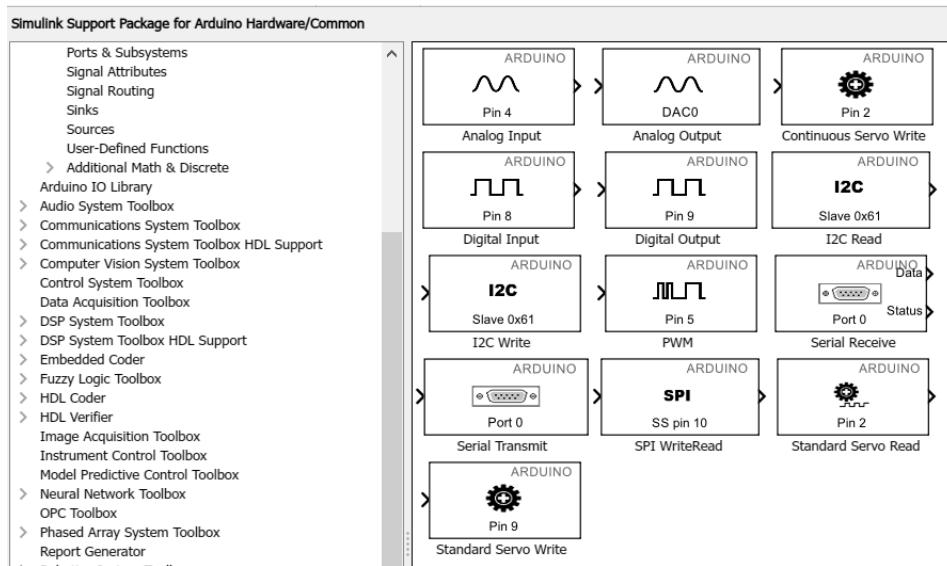


Figura 36: Bloques del paquete de *drivers* de *Simulink* para *Arduino*

#### 4.3.2 Modelado de los *drivers* de control de los motores de *Magabot* con *Simulink*

A continuación, mostramos el proceso que debemos seguir para diseñar los *drivers* de control de los motores de *Magabot* con *Simulink*.

En primer lugar, diseñamos el modelo del sistema *motorsSimulink*, el cual tiene la función de llamar a los *drivers* de los motores, incluidos en el subsistema *motorsSimulinkRef* (Figura 37)

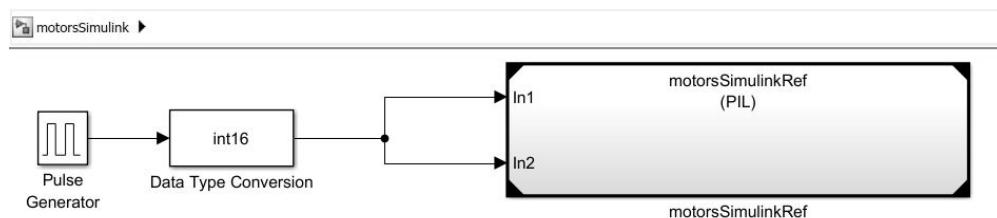


Figura 37: Modelo del sistema *motorsSimulink*

El primer bloque del modelo es un generador de pulsos, al que podemos personalizar. Programamos el generador de pulsos para que envíe una secuencia de *test* que alternará cada 0.1 segundos los valores 0 y 5 (Figura 38).

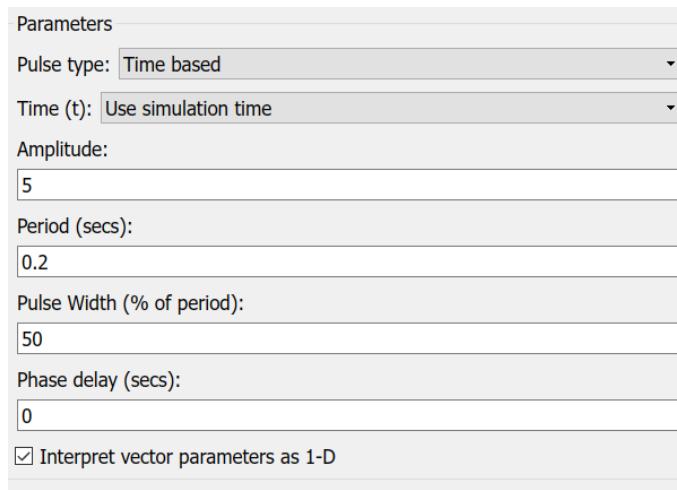


Figura 38: Parámetros del generador de pulsos

Posteriormente, insertamos un bloque de conversión de tipos de datos, con el fin de convertir el valor de 8 bits a 16 bits. Este último valor será el enviado finalmente al subsistema *motorsSimulinkRef*, el cual es un modelo referenciado que se ejecutará en la placa *Arduino Mega 2560*, en modo *Processor In the Loop*.

Para crear el subsistema *motorsSimulinkRef* debemos seleccionar en la biblioteca de bloques el bloque *Subsystem* (ver Figura 35, p. 115). Luego, convertimos ese subsistema en modelo, haciendo click derecho sobre el bloque y seleccionando la opción *Subsystem & Model Reference -> Convert Subsystem to -> Referenced model* (Figura 39).

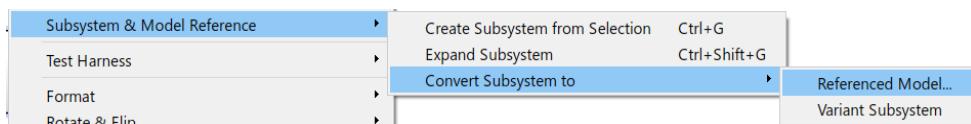


Figura 39: Opción *Subsystem & Model Reference*

En la ventana emergente, *motorSimulink/Subsystem*, ingresamos el nombre del modelo referenciado y pulsamos el botón *Convert* (Figura 40).

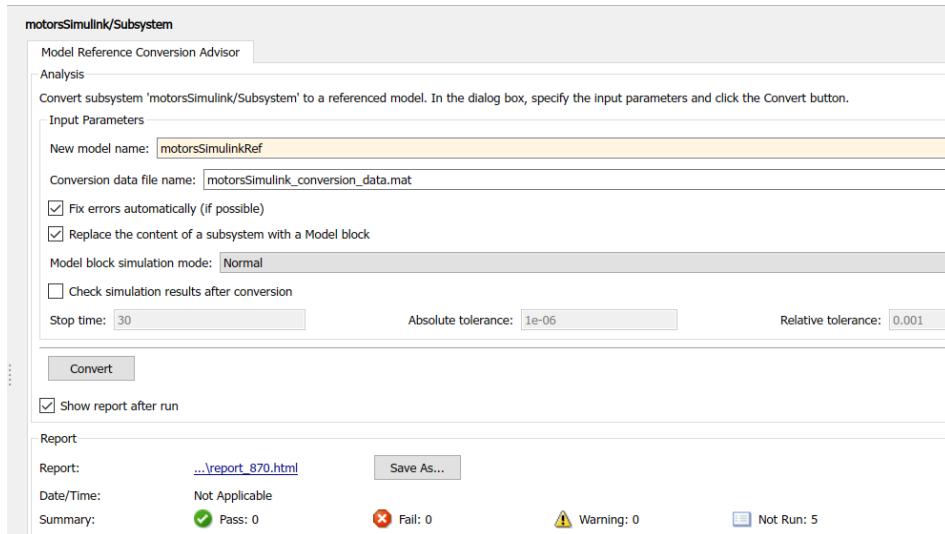


Figura 40: Conversión de un subsistema a modelo referenciado

Luego, hacemos click derecho sobre el modelo referenciado y seleccionamos la opción *Block Parameters (ModelReference)* (Figura 41)

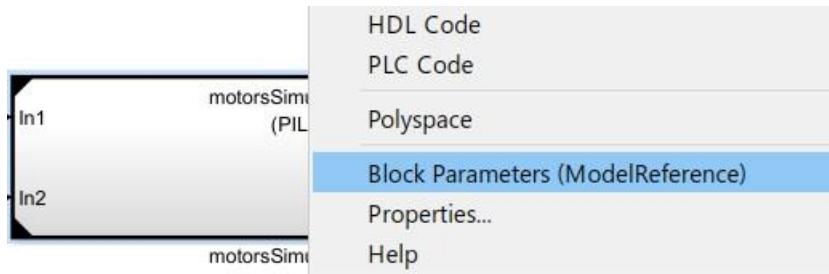


Figura 41: Opción *Block Parameters (ModelReference)*

Aparece una ventana emergente, *Parameters*, en que debemos asignar al campo *Simulation mode* el valor *Processor-in-the-loop*, y al campo *Code interface* el valor *Model reference* (Figura 42)

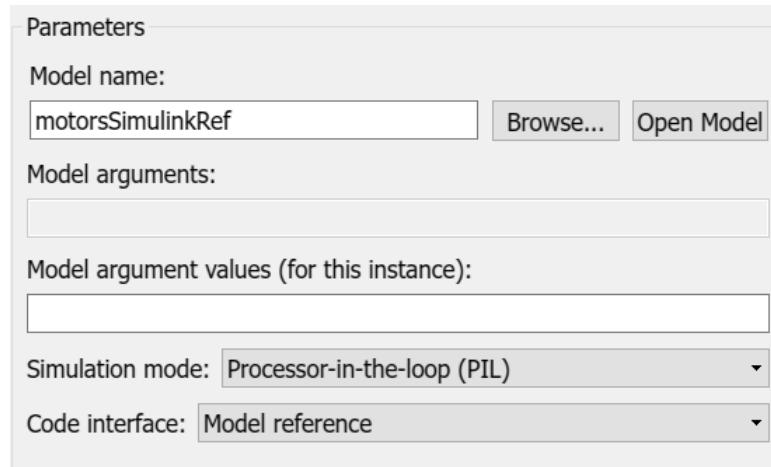


Figura 42: Parámetros de un sistema referenciado

Para que el modelo referenciado funcione correctamente debemos realizar las siguientes operaciones, tanto en el modelo padre como en el modelo referenciado.

En primer lugar, abrimos el menú de configuración (⚙️), desde el menú horizontal superior de *Simulink* (ver Figura 29, p. 112). Desde allí, seleccionamos *Model Referencing*, y asignamos al campo *Propagate sizes of variable-size signals* el valor *During execution* (Figura 43).

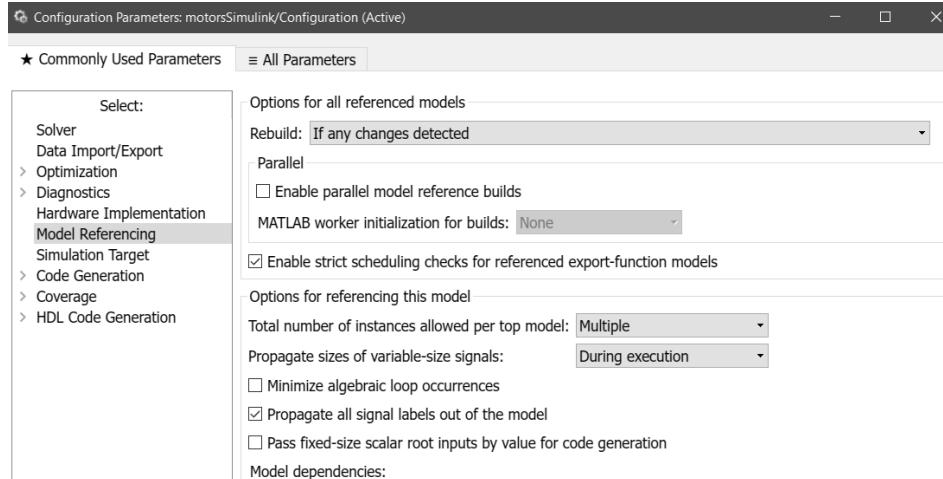


Figura 43: Opción *Model Referencing*

En segundo lugar, para que el código sea generado correctamente, hacemos click en *Code Generation -> Interface -> Software environment*, y seleccionamos la casilla *continuous time* (Figura 44).

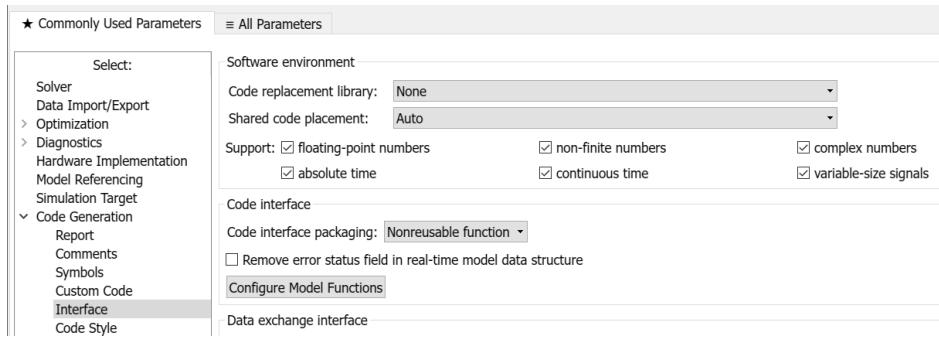


Figura 44: Configuración de *Code Generation*

A continuación, procedemos a diseñar el modelo referenciado *motorsSimulinkRef*, que se encarga de mover de los motores de *Magabot*. Podemos ver el modelo resultante para el diseño en la Figura 45.

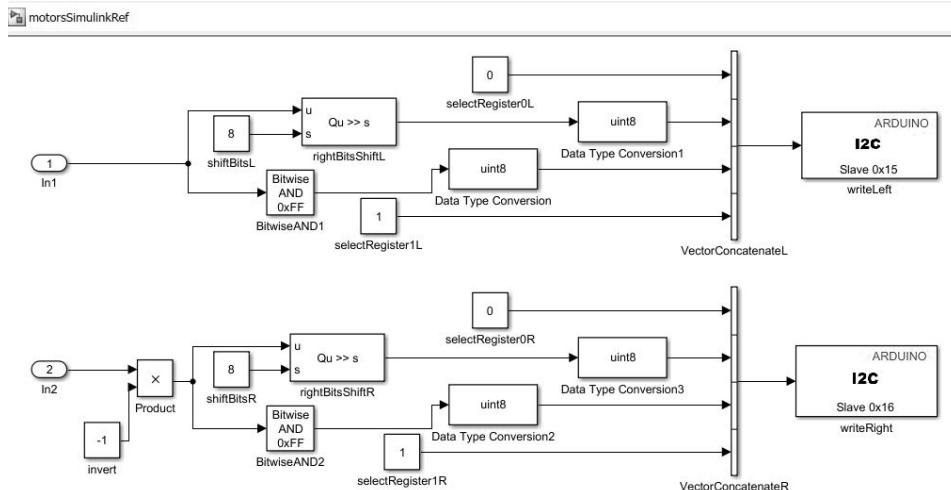


Figura 45: Modelo referenciado *motorsSimulinkRef*

Los bloques *In1* e *In2* son los *inputs* del sistema referenciado, los cuales representan, respectivamente, la velocidad que se debe aplicar a la rueda izquierda y a la rueda derecha. Estos valores son de tipo *uint16*. Utilizando un bloque de tipo *Product* (ver Figura 35, p. 115), debemos invertir el signo del valor *In2*, ya que el motor izquierdo está instalado en posición invertida en el *Magabot*. En la Figura 45 podemos observar cuatro bloques que realizan operaciones lógicas *bit a bit*: mientras dos bloques ejecutan desplazamientos de bits, los otros dos realizan operaciones de tipo *bitwise AND*.

Luego, concatenamos en dos *arrays* los valores obtenidos en las operaciones, junto con las constantes de valor 0 y 1, utilizadas para seleccionar los registros de los motores. Los vectores obtenidos, son enviados

a dos bloques del paquete de *Simulink* para *Arduino*, cuya función es la de trasmitir de forma ordenada los valores a los motores, usando el protocolo *I2C*.

Por otra parte, para configurar la generación del código de estos *drivers*, debemos volver al menú de opciones, y hacer click en *Code Generation*. En el campo *System Target File*, seleccionamos *ert.tlc*, y en *Language* elegimos *C*. En *Toolchain* indicamos *Arduino AVR*, y en *Build configuration*, seleccionamos *Faster Runs* (Figura 46).

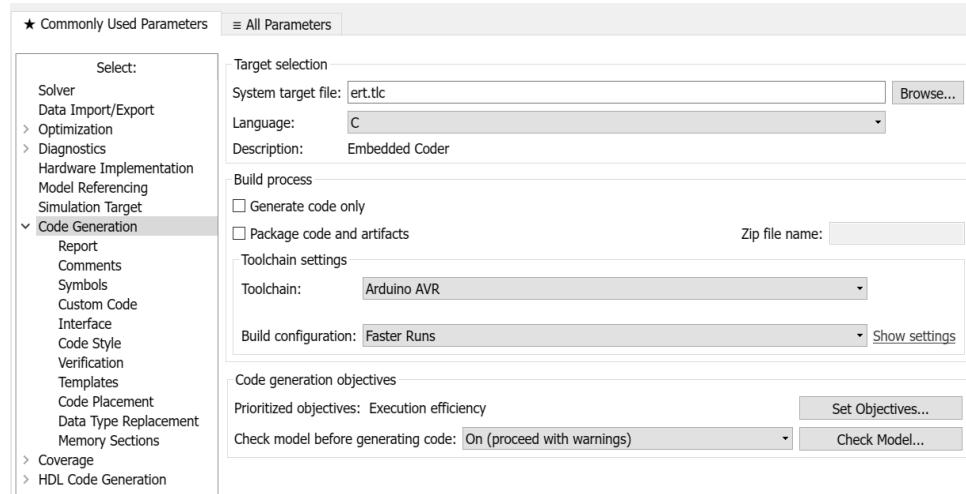


Figura 46: Configuración de *Code Generation*

A continuación, hacemos click en el botón *Set Objectives*. En la ventana emergente *Set Objectives - Code Generation Adviser*, podemos definir los requisitos de eficiencia que queremos priorizar a la hora de generar el código (Figura 47). En nuestro caso, elegimos que el código sea eficiente en cuanto a su velocidad de ejecución.

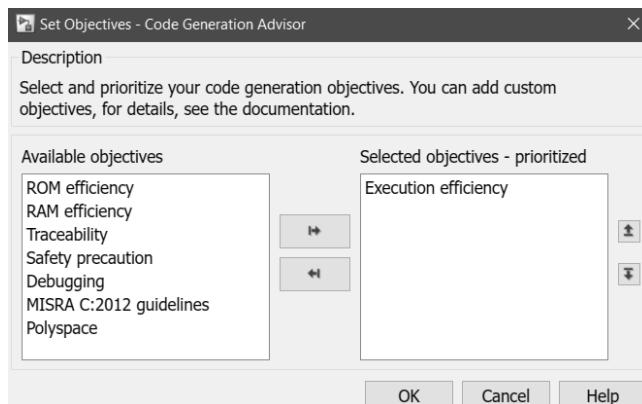


Figura 47: *Set Objectives - Code Generation Adviser*

En *Code Generation -> Report*, podemos configurar *Simulink* para que cree un informe de la generación del código (Figura 48).

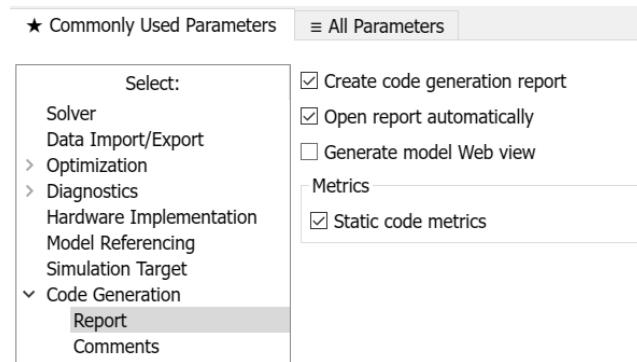


Figura 48: Opción *Create code generation report*

Luego, en *Code Generation -> Interface -> Software environment*, seleccionamos la casilla *continuous time* (Figura 49).

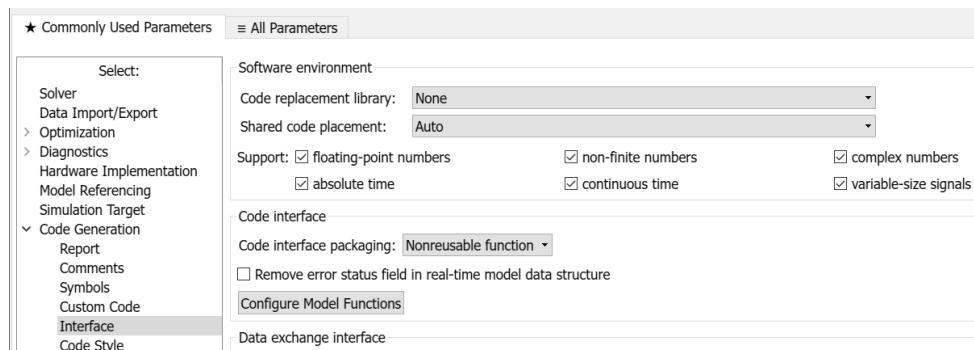


Figura 49: Configuración *Code Generation -> Interface*

Para realizar mediciones de rendimiento de la ejecución del código debemos seleccionar las opciones *Measure task execution time* y *Measure function execution times*, que se encuentran en *Code Generation -> Verification -> Code Profiling* (Figura 50).

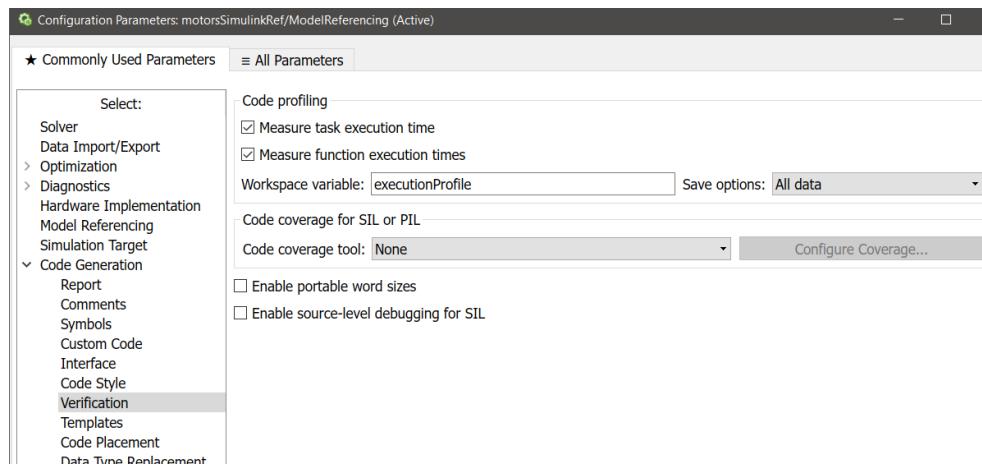


Figura 50: Configuración *Code Profiling*

Para generar el código e iniciar su ejecución en la placa *Arduino Mega 2560*, debemos abrir el modelo *motorsSimulink*, y seleccionar en el menú principal la modalidad de ejecución *normal* (Figura 51).

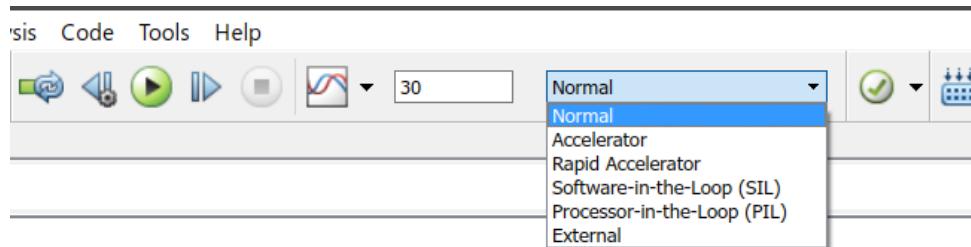


Figura 51: Modalidad de ejecución normal

Luego, pulsamos el botón *Run* (▶) y esperamos que el compilador compile el código, y que aparezcan los informes de la generación del código. Los motores se ponen en funcionamiento, y las dos ruedas de *Magabot* se mueven durante 30 segundos.

Al terminar la ejecución del código, aparece el informe del *profiling*, mostrando unas tablas con los tiempos de ejecución detallados para cada bloque. Estas tablas se mostrarán en un apartado posterior destinado a la comparación de rendimiento de las dos implementaciones consideradas (código en *C++* de los drivers de *Magabot*, y modelo *Simulink* que se utiliza para la generación automática de código equivalente).

### 4.3.3 Modelado de los *drivers* de lectura de los *encoders* de Magabot con Simulink

Para diseñar los *drivers* de lectura de los *encoders* debemos realizar un proceso similar al arriba detallado. En primer lugar, tenemos que crear el modelo *encodersSimulink* (Figura 52).

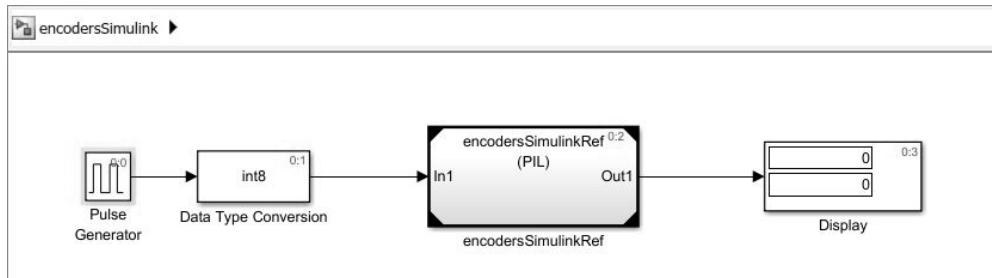


Figura 52: Modelo *encodersSimulink*

El modelo *encodersSimulink* es el encargado de llamar al modelo referenciado *encodersSimulinkRef* (Figura 53).

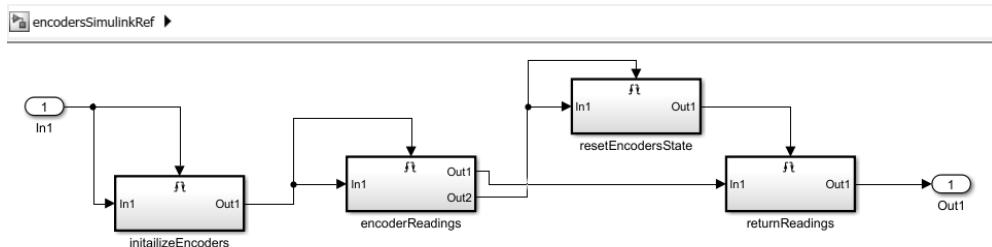


Figura 53: Modelo *encodersSimulinkRef*

Haciendo doble click en el bloque *Pulse Generator*, aparece la ventana emergente *Parameters*. Aplicando la configuración que se muestra en la Figura 54, el bloque generará, cada medio segundo, los valores 0 y 1 de forma alternada. Esta secuencia generada, cabe aclarar, es una secuencia de prueba para verificar el funcionamiento del sistema embebido. En el sistema real, por el contrario, la función de control de los *encoders* es llamada por el subsistema de navegación.

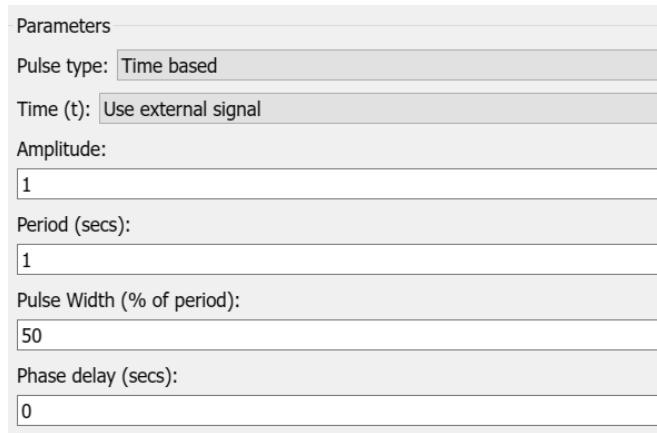


Figura 54: Parámetros del bloque *Pulse Generator*

Los valores 0 y 1 son convertidos al tipo *int8*, y luego transferidos al modelo referenciado *encodersSimulinkRef*. Este último modelo dispone de una salida, que se encarga de trasmitir en radianes el valor de los giros realizados por las ruedas en un *array* bidimensional de tipo *float*. Estos valores son mostrados por un bloque *Display* en tiempo de ejecución (ver Figura 52, p. 124).

El bloque *encodersSimulinkRef* recibe, por la entrada *In1*, los valores 0 y 1 de manera alternada. Estos valores sirven para activar los subsistemas *initilizeEncoders*, *encodersReadings*, *resetEncodersState*, y *returnReadings*.

#### 4.3.3.1 Subsistema *initilizeEncoders*

El subsistema *initilizeEncoders* dispone de un bloque que se llama *Trigger* (Figura 55). El bloque *Trigger* activa el subsistema cada vez que recibe una señal de subida o de bajada. Luego, los dos bloques *I2C* escriben el valor 1 en los *encoders*. La señal activadora del *Trigger* ingresa por el puerto *In1*, y vuelve a salir por el puerto *In2 -> Out1*.

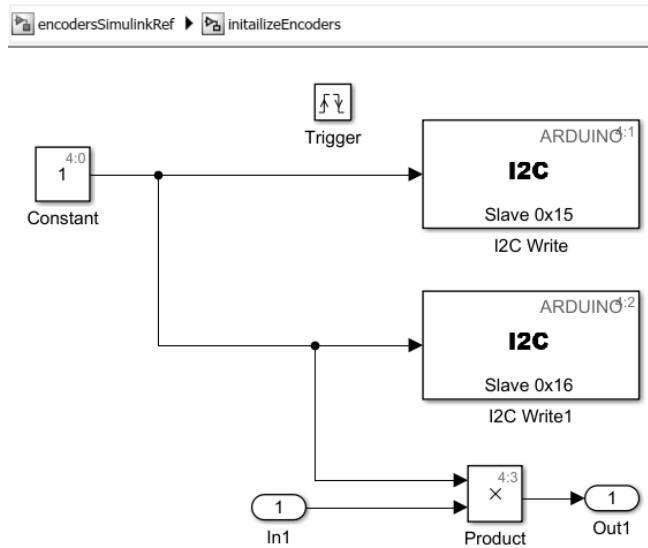


Figura 55: Subsistema *initilizeEncoders*

#### 4.3.3.2 Subsistema *encoderReadings*

El subsistema *encoderReadings* es activado por un bloque de *Trigger* (Figura 56).

La entrada *In1* y la salida *Out2* tienen la función de propagar la señal de activación.

Los dos bloques *I2C* realizan las lecturas de los *encoders*. Luego, la lectura del *encoder* izquierdo es multiplicada por -1, ya que éste *encoder* está montado al revés. Los dos valores son convertidos primero de *int8* a *int16*, luego de *int16* a *float*.

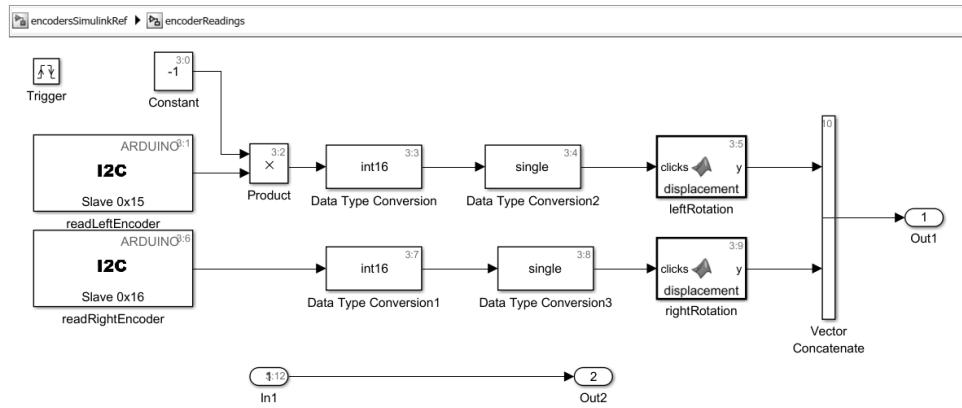


Figura 56: Subsistema *encoderReadings*

*leftRotation* y *rightRotation* son bloques del tipo *MATLAB function*, que permiten incrustar en el modelo una función escrita en *MATLAB*. Haciendo doble click en esos bloques, incluimos la función *displacement*, la cual recibe como parámetro el valor de las lecturas de los *encoders*, y devuelve la rotación del eje en radianes (Figura 57). Cabe aclarar que el número 3900 es la resolución del *encoder*, expresada en *clicks* por vuelta del eje

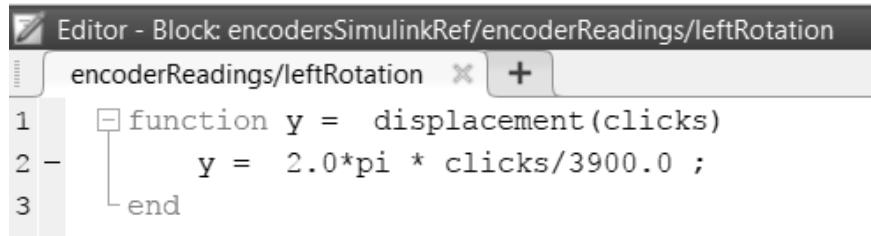


Figura 57: Función *displacement*

Los dos valores devueltos por los bloques *leftRotation* y *rightRotation* son concatenados en un vector y devueltos por la salida *Out1*.

#### 4.3.3.3 Subsistema *resetEncoderState*

La señal de activación pasa ahora al subsistema *resetEncoderState* (Figura 58), el cual escribe un cero en los *encoders* usando dos bloques *I2C*.

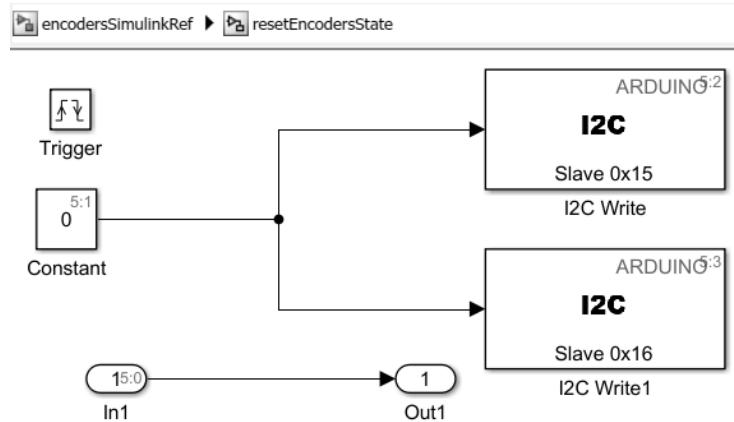


Figura 58: Subsistema *resetEncoderState*

#### 4.3.3.4 Subsistema *returnReadings*

Finalmente, cuando activamos el *Trigger* del subsistema *returnReadings* (Figura 59), este trasmite el resultado al puerto de salida del modelo *encodersSimulinkRef*.

Al ejecutar el modelo *encodersSimulink*, el bloque *Display* muestra en tiempo real, durante 30 segundos, los giros realizados por las ruedas de *Magabot*, expresados en radianes (ver Figura 52, p. 124).

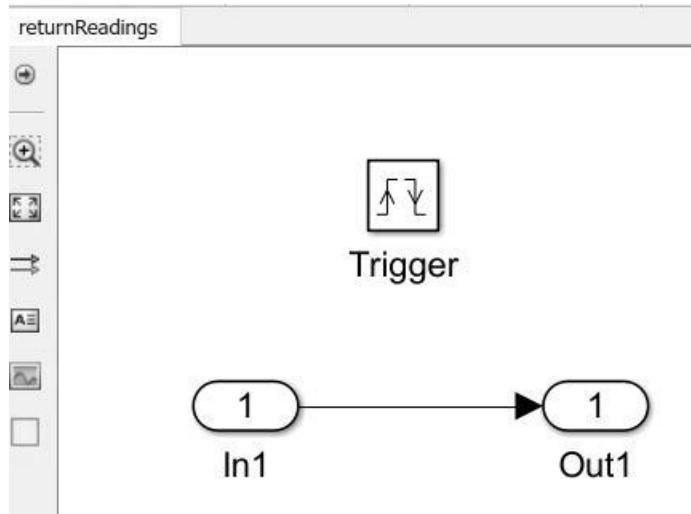


Figura 59: Subsistema *returnReadings*

## 4.4 INCLUSIÓN DEL CÓDIGO C++ DE LOS DRIVERS EN MATLAB SIMULINK

Para poder comparar dentro del entorno *MATLAB Simulink* el tiempo de ejecución del código generado automáticamente por *Embedded Coder* a partir de los modelos anteriores, con el tiempo de ejecución del código en C++, debemos trasladar a *Simulink* el paquete de los *drivers* de *Magabot*, y posibilitar que se llamen sus funciones desde un modelo *Simulink*.

### 4.4.1 Inclusión en *Simulink* del código en C++ de los *drivers* de control de los motores

A continuación, describimos los pasos que seguimos para diseñar el modelo que llama a los *drivers* que hemos creado, a partir de los proporcionados por los fabricantes, para el control de los motores de *Magabot*. El modelo *motorsArduino* (Figura 60), al igual que el modelo *motorsSimulink* anteriormente descrito, tiene la única función de generar una secuencia de prueba para llamar al modelo referenciado *motorsArduinoRef* (Figura 61), ejecutándolo en modo *Processor In the Loop*.

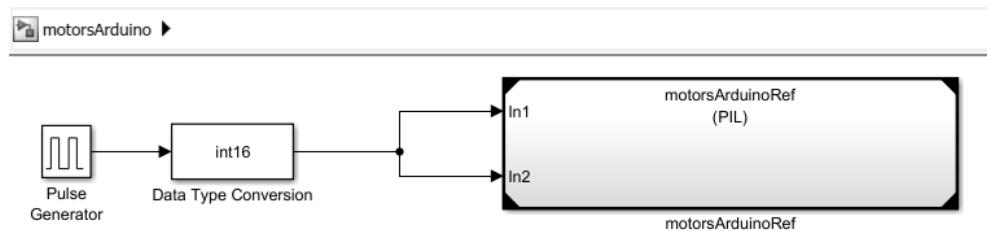


Figura 60: Diseño del modelo *motorsArduino*

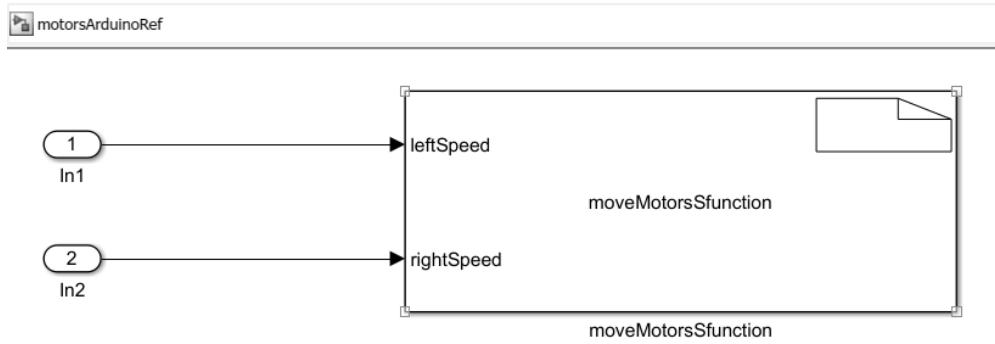


Figura 61: Diseño del modelo *motorsArduinoRef*

En el modelo *motorsArduinoRef*, introducimos un bloque personalizable del tipo *S-Function Builder*, dentro del cual podemos incluir código escrito en lenguaje *C*. Para ello, debemos hacer doble click en el bloque para configurarlo.

En primer lugar, incluimos en la carpeta *Workspace* de *MATLAB* los archivos *Magabot.h* y *Magabot.cpp*, que contienen a los *drivers* de *Magabot*. En *Windows*, esta carpeta *Workspace* de *MATLAB* se encuentra por defecto en la ruta ...\\Documentos\\MATLAB.

Hacemos doble click sobre el bloque *S-Function Builder*, y se nos abre el menú de configuración. En el campo *S-function name* podemos definir el nombre de la función (Figura 62). En nuestro caso, nombramos a esa función: *moveMotorsSfunction*.

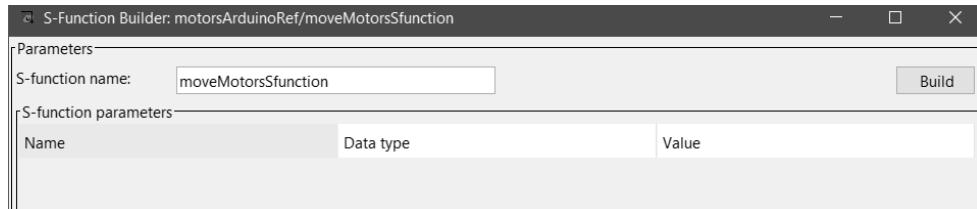


Figura 62: Campo *S-function name*

Luego, seleccionamos la pestaña *Data Properties -> Input ports*, donde declararemos las dos entradas de la función: *leftSpeed* y *rightSpeed*. Asimismo, definimos la dimensión de las mismas, como se puede observar en la Figura 63.

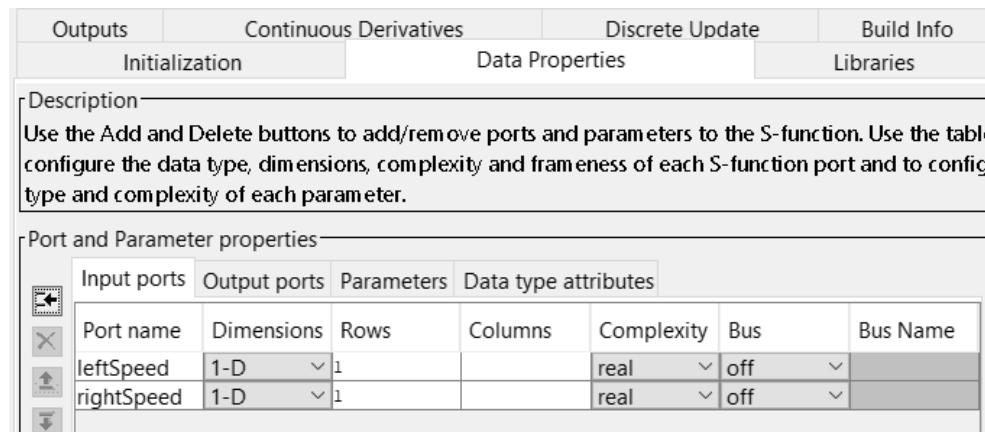


Figura 63: Definición de puertos de entrada de *moveMotorsSfunction*

En *Data Properties -> Data type attributes*, indicamos el tipo de datos de los atributos, como podemos apreciar en la Figura 64.

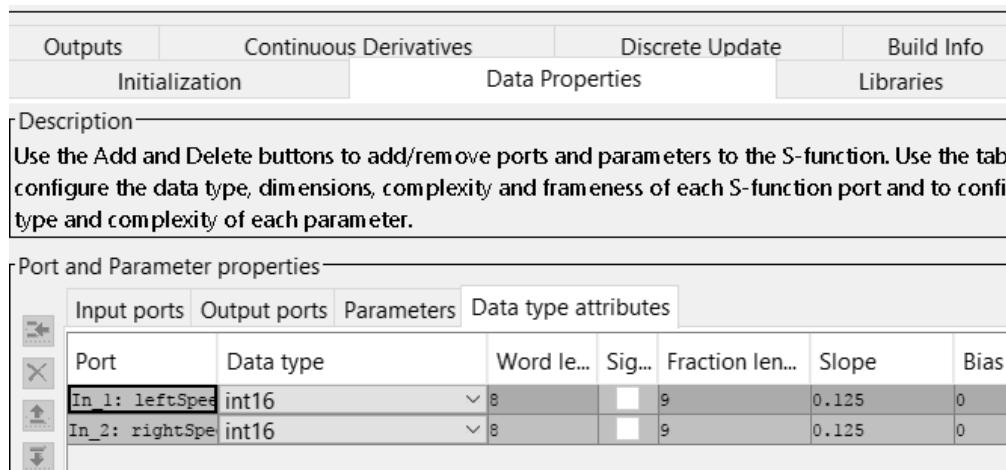


Figura 64: Definición del tipo de atributos de *moveMotorsSfunction*

A continuación, en la pestaña *Libraries*, declaramos el uso de los paquetes *Magabot.cpp* y *Arduino.h* (Figura 65)

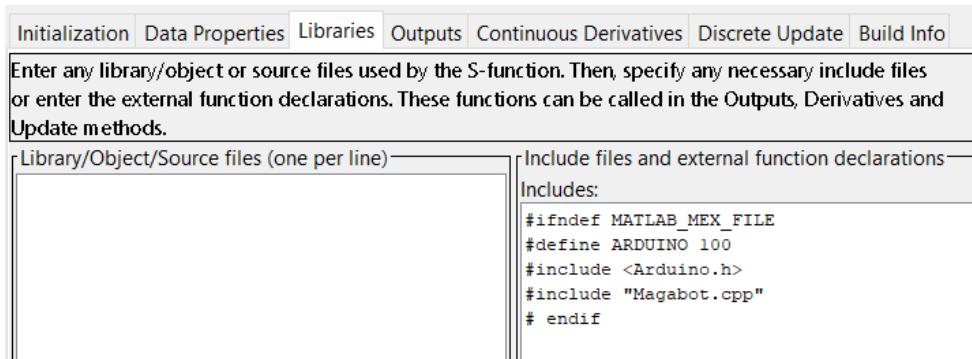


Figura 65: Declaración de paquetes de *moveMotorsSfunction*

Luego, en la pestaña *Outputs*, insertamos el código que sirve para llamar la función *actuateMotors(leftSpeed, rightSpeed)* del paquete *Magabot.h*. Cabe destacar que los atributos *leftSpeed* y *rightSpeed* son las entradas de *moveMotorsSFunction* (Figura 66).

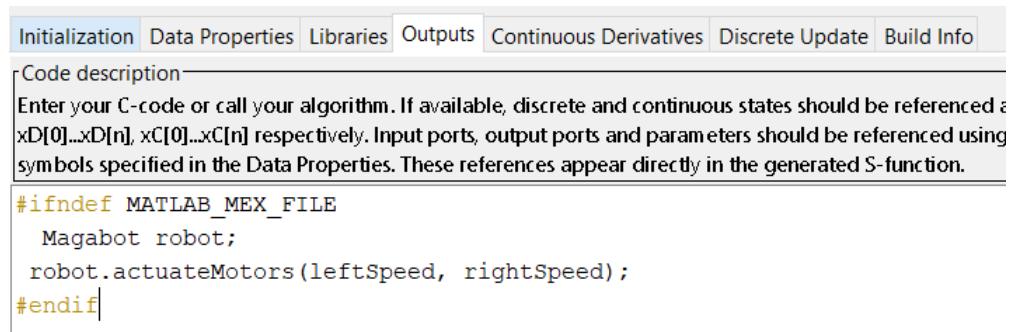


Figura 66: Código en lenguaje C de *moveMotorsSfunction*

Finalmente, pulsamos el botón *Build* que se encuentra en la parte superior derecha. Con esta acción *MATLAB* genera, en el *Workspace*, tres ficheros: *moveMotorsSfunction.mexw64*, *moveMotorsSfunction.tlc* y *moveMotorsSfunction\_wrapper.c*. Tenemos que editar este último fichero, agregando el texto *extern "C"* antes de cada una de las funciones (Figura 67).

```
/*
 * Output functions
 *
 */
extern "C" void moveMotorsSfunction_Outputs_wrapper(const int16_T *leftSpeed,
                                                       const int16_T *rightSpeed)
{
```

Figura 67: Edición de *moveMotorsSfunction\_wrapper.c*

Por último, debemos cambiar la extensión “c” del fichero *moveMotorsSfunction\_wrapper.c*, y renombrarla “cpp”. De esta manera, el nombre final queda del siguiente modo: *moveMotorsSfunction\_wrapper.cpp*.

#### 4.4.2 Inclusión en *Simulink* del código en C++ de los *drivers* de lectura los *encoders*

En este apartado, con el objetivo de crear un modelo de *Simulink* que sirva para leer los *encoders*, llamamos a la función *readClicks()* del paquete de *Magabot*.

Comenzamos, entonces, creando un modelo *encodersArduino* que, haciendo uso de un bloque *Pulse Generator*, llama al bloque *Trigger* del modelo referenciado *encodersArduinoRef*. Ese *Trigger*, se activa sólo cuando le llega una señal de subida.

Dentro del modelo *encodersArduinoRef*, se encuentra el bloque *encodersSfunction*, que es un bloque del tipo *S-Function Builder*. Hacemos doble click sobre el bloque *encodersArduinoRef*, y procedemos a configurarlo configurarlo de manera similar al apartado anterior. Para ello, seleccionamos la pestaña *Data Properties -> Output ports*, y añadimos el parámetro de salida de la función llamado *wheelsRotation*.

Para definir el parámetro *wheelsRotation* como *array* bidimensional, asignamos al campo *Rows* el valor 2 (Figura 68).

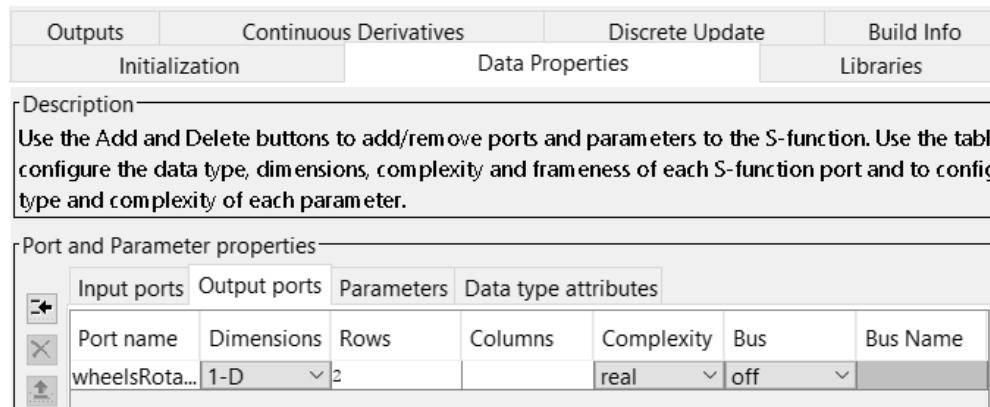


Figura 68: Definición de puertos de salida de *encodersSfunction*

Para definir *float* como *array* *wheelsRotation*, tenemos que seleccionar la pestaña *Data Properties -> Data type attributes*, y asignar al campo *Data type* el valor *single* (Figura 69).

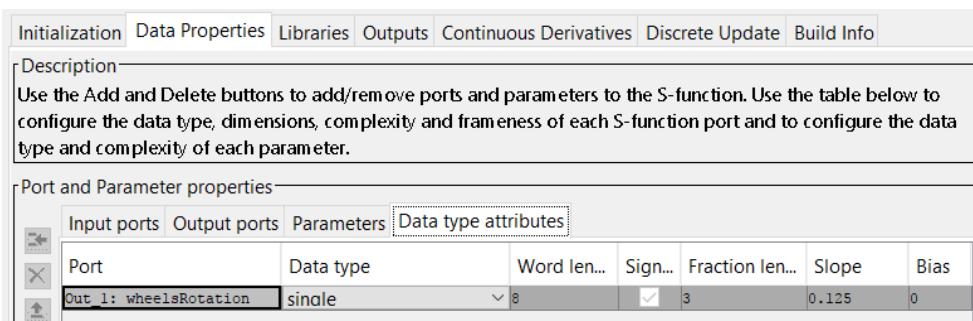


Figura 69: Definición del tipo de atributos de *encodersSfunction*

Seleccionando la pestaña *Libraries*, incluimos el paquete de *drivers* de *Magabot*, siguiendo el mismo procedimiento que describimos en relación al bloque *moveMotorsSfunction* (ver Figura 64, p. 131).

A continuación, en la pestaña *Outputs*, incluimos el código en *C* para leer los *encoders* (Figura 70).

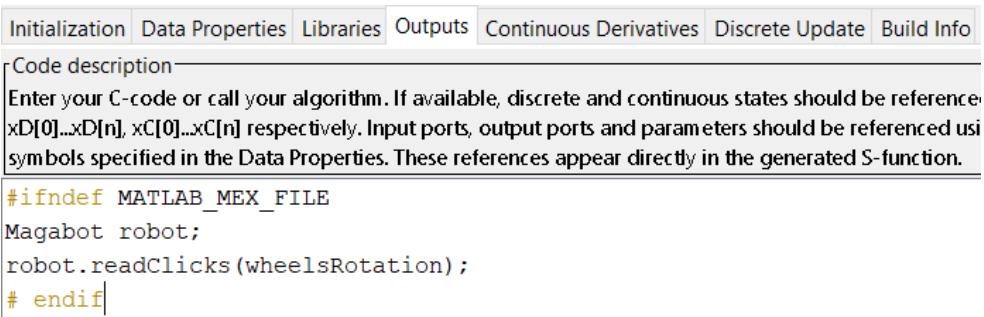


Figura 70: Código en lenguaje *C* de *encodersSfunction*

Después de pulsar el botón *Build*, *MATLAB* genera tres ficheros: *encodersSfunction\_mexw64*, *encodersSfunction.tlc* y *encodersSfunction\_wrapper.c*. Editamos este último fichero, agregando el texto *extern "C"* antes de cada una de las funciones, como podemos observar en la Figura 71.

```
*\n- */\nextern "C" void encodersSfunction_Outputs_wrapper(real32_T *wheelsRotation)\n{\n    /* %%%SFUNWIZ_wrapper_Outputs_Changes_BEGIN --- EDIT HERE TO _END */\n    /* This sample sets the output equal to the input\n       y0[0] = u0[0];\n    For complex signals use: y0[0].re = u0[0].re;\n       y0[0].im = u0[0].im;\n       y1[0].re = u1[0].re;
```

Figura 71: Edición de *encodersSfunction\_wrapper.c*

Como último paso, procedemos a cambiar la extensión “c” del fichero *encodersSfunction\_wrapper.c*, para renombrarla “cpp”. Como resultado, el fichero queda denominado: *encodersSfunction\_wrapper.cpp*.

## 4.5 PROFILING DEL CÓDIGO EN SIMULINK

En este apartado, analizamos el rendimiento del tiempo de ejecución del código de los modelos realizados con *MATLAB Simulink*. Para comenzar, tenemos que comprobar que esté activado el *profiling*. Para ello, seleccionamos las opciones *Measure task execution time* y *Measure function execution times*, que se encuentran en *Configuration Parameters -> Code Generation -> Verification -> Code profiling* (ver Figura 50, p. 123). Una vez que hemos configurado correctamente las opciones de *profiling* en ambos modelos, lanzamos la ejecución del modelo padre desde el menú horizontal superior de *MATLAB Simulink*, pulsando el botón *Run* (▶). Al final de la ejecución, aparece una tabla como la que se muestra en la Figura 72, donde se detallan los tiempos de ejecución de cada bloque en ambos modelos.

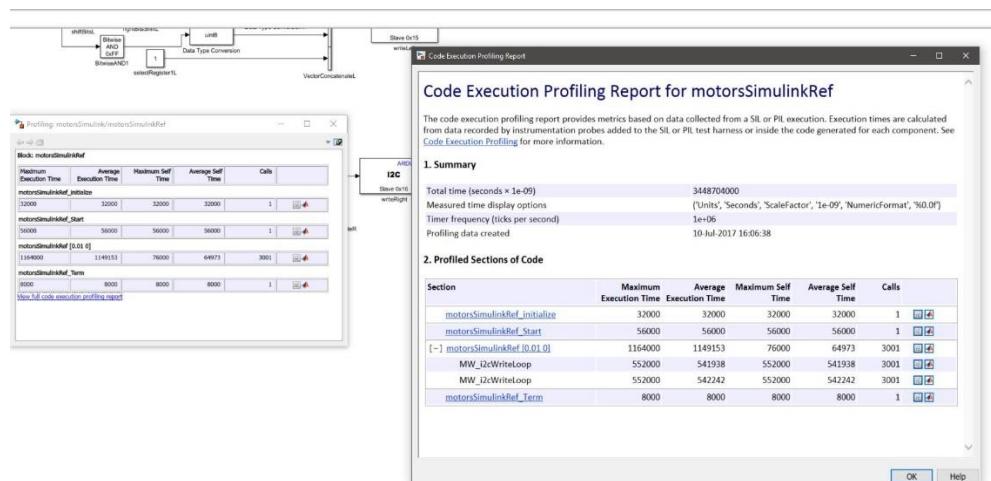


Figura 72: Tabla resultado del *profiling*

En las siguientes tablas, reportamos los tiempos de ejecución de los modelos diseñados en su totalidad con *Simulink* y de los modelos que incluyen al código programado en *C++*, a fin de compararlos.

Profiling de los <i>drivers</i> de los motores			
Nombre del bloque analizado	Tipo de bloque	Tiempo máximo de ejecución	Tiempo medio de ejecución
<b>motorsSimulinkRef</b>	Modelo <i>Simulink</i>	1.164.000 ns	1.149.153 ns
<b>moveMotorsSfunction</b>	<i>S-function</i> con código de <i>Arduino</i>	1.128.000 ns	1.111.528 ns

Profiling de los <i>drivers</i> de los <i>encoders</i>			
Nombre del bloque analizado	Tipo de bloque	Tiempo máximo de ejecución	Tiempo medio de ejecución
<b>encodersSimulinkRef</b>	Modelo <i>Simulink</i>	3.080.000 ns	2.643.345 ns
<b>encodersSfunction</b>	<i>S-function</i> con código de <i>Arduino</i>	2.736.000 ns	2.713.548 ns

A través de estas tablas, podemos apreciar, en primer lugar, que los tiempos medios de ejecución del código escrito en *C++* y el código autogenerado por *MATLAB Simulink* tienen diferencias inferiores a los 0.1 microsegundos. En segundo lugar, observamos que los *drivers* de los motores programados en lenguaje *C* han resultado más eficientes, mientras que, para los *drivers* de los *encoders*, ha sido más rápida la ejecución del modelo realizado con *Simulink*. En definitiva, la generación automática de código en *MATLAB Embedded Coder*, con la configuración específica de acelerar la ejecución, lleva a un resultado muy similar en eficiencia al que se obtiene con la codificación manual. A todo esto se suma la ventaja adicional que supone el diseño basado en modelo. Por todo ello, tras esta comparación podemos concluir que *MATLAB Simulink* y *Embedded Coder* son herramientas válidas para el desarrollo de código eficiente para las placas *Arduino*.

# Pruebas





## 5 PRUEBAS

---

A continuación mostramos los resultados obtenidos durante el desarrollo de las distintas pruebas realizadas con el robot. En la sección “5.7 Pruebas de aceptación del sistema”, hemos incluido el enlace de un vídeo en el que se puede ver a nuestro robot en funcionamiento. Por último, describimos la demostración realizada con el robot frente a un grupo de estudiantes del *IES Virgen de Valme*, que asistieron a un taller de robótica en la Universidad Pablo de Olavide.

### 5.1 PRUEBAS DE ODOMETRÍA EN LÍNEA RECTA

Para esta prueba hemos procedido a mover el robot en línea recta. Al alcanzar la distancia de 2 metros, hemos detenido el robot y mostrado por puerto serie la distancia estimada por odometría. A continuación, hemos comparado esos valores con las mediciones de desplazamiento real del robot.

En la siguiente tabla, reportamos los resultados obtenidos.

Pruebas de odometría en línea recta			
Iteración	Valor odometría	Valor real	Diferencia
1	200,05 cm	200,10 cm	0,05 cm
2	200,08 cm	200,20 cm	0,12 cm
3	200,10 cm	200,10 cm	0,00 cm
4	200,09 cm	200,10 cm	0,01 cm
5	200,07 cm	200,00 cm	0,07 cm

En definitiva, el error de la odometría resultante, en un recorrido de 2m en línea recta, es inferior al 0,1%. Por este motivo, consideramos que este error no influye de manera significativa en las pruebas obtenidas ni en los objetivos de nuestro trabajo.

## 5.2 PRUEBAS BIDIRECCIONALES DE ODOMETRÍA EN RECORRIDO CUADRADO *UBMARK*

Para esta prueba, hacemos desplazar el robot en un recorrido cuadrado de 2m x 2m, en sentido horario y antihorario. Luego, medimos la posición final del robot, y calculamos la distancia entre el punto de llegada estimado por odometría y el punto de llegada real. En las siguientes tablas, mostramos los resultados obtenidos en las diez pruebas realizadas.

Pruebas de odometría en recorrido cuadrado de 2m x 2m en sentido horario			
Iteración	Valor odometría en cm (x, y)	Valor real en cm (x, y)	Distancia euclídea entre los dos valores
1	(-2.2 cm, -4.4 cm)	(-3.5 cm, -2.5 cm)	2,30 cm
2	(-2.2 cm, -4.8 cm)	(-4 cm, -3 cm)	2,28 cm
3	(-2.2 cm, -4.4 cm)	(-4.5 cm, -2.5 cm)	2,98 cm
4	(-2.2 cm, -4.4 cm)	(-3.5 cm, -5.5 cm)	1,70 cm
5	(-2.2 cm, -4.4 cm)	(-3.5 cm, -4 cm)	1,36 cm

Pruebas de odometría en recorrido cuadrado de 2m x 2m en sentido antihorario			
Iteración	Valor odometría en cm (x, y)	Valor real en cm (x, y)	Distancia euclídea entre los dos valores
1	(-2.0 cm, 4.5) cm	(-3.5, 6.5 cm)	2,50 cm
2	(-2.0 cm, 4.7 cm)	(-5.5 cm, 5.5 cm)	3.59 cm
3	(-2.0 cm, 4.5 cm)	(-4.5 cm, 6.5 cm)	3,20 cm
4	(-1.9 cm, 4.5 cm)	(-3.5 cm, 4 cm)	1,68 cm
5	(-2.0 cm, 4.5 cm)	(-4 cm, 5.5 cm)	2,24 cm

En definición, después de un recorrido de 8 metros y de la realización de tres giros de 90°, en ninguna de las pruebas efectuadas el error de estimación de posición ha superado los 4 cm. Sobre la base de la consideración de que los obstáculos son “inflados” o “expandidos” para que el robot mantenga una distancia segura, y que los mapas son actualizados con frecuencia para limitar la acumulación de errores, podemos afirmar que se cumple de forma eficiente con el objetivo de determinar la posición del robot durante la navegación.

### **5.3 PRUEBAS DE LECTURAS DE LOS SONARS**

Para el desarrollo de esta prueba, procedemos a posicionar objetos de distintas formas delante del robot. Luego, verificamos su capacidad para la detección de obstáculos, a través del *monitor* del puerto serie del *Arduino Mega 2560*.

<b>Pruebas de lecturas de los sonars</b>			
<b>Descripción del obstáculo</b>	<b>Valor medido por el sónar</b>	<b>Valor real</b>	<b>Diferencia entre los valores</b>
Caja rectangular de cartón puesta en posición perpendicular al sonar	0.50 m	0.50 m	0.00 m
Caja rectangular de cartón puesta inclinada a 30° respecto al sonar	0.48 m	0.47 m	0.01 m
Caja rectangular de cartón inclinada a 45° respecto al sonar	4.53 m	0.45 m	4.08 m
Caja rectangular de cartón posicionada con una esquina en dirección del sonar	3.19 m	0.50 m	2.69 m
Objeto metálico cilíndrico de 25cm de diámetro, dispuesto en posición vertical	0.50 m	0.50 m	0.00 m
Objeto metálico cilíndrico de 2cm de diámetro, dispuesto en posición vertical	0.50 m	0.50 m	0.00 m

Sobre la base de los resultados obtenidos, podemos afirmar que la capacidad de detección de obstáculos por parte del *sonar* depende de la forma del objeto. De esta manera, cuando posicionamos frente al *sonar* una caja con un ángulo pronunciado, este no es capaz de detectarla, ya que el sonido rebota y vuelve al *sonar* reflejado por otros objetos. Lo mismo acontece cuando se posiciona en dirección del *sonar* la arista de una caja. En este último caso, el sonido es reflejado hacia los costados, y la lectura falla.

Por el contrario, cuando colocamos frente a un *sonar* un objeto cilíndrico, o uno plano en posición perpendicular, obtenemos lecturas precisas por parte del *sonar*.

## 5.4 PRUEBAS DE ORIENTACIÓN DEL PULSO SONORO

Con esta prueba pretendemos verificar la capacidad del sistema de detección del pulso sonoro para calcular el ángulo de origen del sonido.

Para ello, nos posicionamos a una distancia del robot de 3 metros, y damos una palmada. Luego, a través de una conexión por puerto serie con la placa *Arduino Due*, verificamos en un *PC* si el ángulo calculado corresponde con el ángulo de origen del sonido.

En la siguiente tabla, mostramos los resultados conseguidos.

Pruebas de orientación del pulso sonoro			
Iteración	Ángulo calculado	Ángulo medido	Diferencia
1	-0,08 rad	0,00 rad	0,08 rad
2	0,47 rad	0,52 rad	0,05 rad
3	0,94 rad	1,05 rad	0,11 rad
4	1,64 rad	1,57 rad	0,07 rad
5	2,15 rad	2,09 rad	0,06 rad
6	- 3,11	3,14 rad	0,03 rad
7	- 2,01 rad	- 2,09 rad	0,08 rad
8	- 1,45 rad	- 1,57 rad	0,12 rad
9	- 1,10 rad	- 1,05 rad	0,05 rad
10	- 0,57 rad	- 0,52 rad	0,05 rad

Esta prueba nos permite concluir que el error promedio en el cálculo del ángulo es de 0,07 radianes. Esto nos permite afirmar que, la capacidad del robot para detectar el ángulo de origen de un pulso sonoro resulta eficaz.

## 5.5 PRUEBAS DE FUNCIONAMIENTO DEL APUNTADOR

En este apartado se describe la prueba realizada para comprobar el funcionamiento del apuntador, que hemos construido con piezas de *Lego Technic*.

El apuntador necesita moverse a 360° para desempeñar el propósito de, por un lado, señalar en dirección del origen del pulso sonoro y, por otro, apuntar hacia el punto objetivo en el mapa de dirección. Para mover el apuntador utilizamos un motor servo que tiene solo 180° de rotación, y un sistema de engranajes con relación de transmisión de 2:1. Este último sistema, como ya mencionamos, le permite al apuntador girar 360°.

En la siguiente prueba, utilizamos en la placa *Arduino Mega 2560* la función `write(int n)` del paquete *Servo* para mover el motor servo. Llamamos a dicha función pasándole distintos valores como parámetro, y comprobamos si el apuntador se orienta en la dirección esperada. Los resultados obtenidos en las pruebas se muestran en la siguiente tabla.

Pruebas de funcionamiento del apuntador		
Valor enviado a la función Servo.write(int n)	Angulo esperado por el puntador	Angulo indicado por el apuntador
0	-180°	-180°
15	-150°	-150°
30	-120°	-120°
45	-90°	-90°
60	-60°	-60°
75	-30°	-30°
90	0°	0°
105	30°	30°
120	60°	60°
135	90°	90°
150	120°	120°
165	150°	150°
179	180°	180°

Como podemos observar en esta tabla, el apuntador se orienta de forma correcta hacia la dirección esperada, sin reportar errores.

## 5.6 PRUEBAS DE ACTIVACIÓN DE LOS LED

Los *LED RGB* pueden mostrar una amplia gama de colores, mezclando únicamente luz roja, verde y azul.

Para llevar a cabo la prueba de activación de los *LED*, utilizaremos la placa *Arduino Mega 2560*. Desde esa placa enviaremos a los *LED* del *Magabot* valores de rojo, verde y azul (*RGB*), en un rango de [0, 255]. De esta manera, podremos comprobar si los *LED* muestran el color esperado.

Pruebas de funcionamiento de los <i>LED</i>		
Valor enviados a los <i>LED</i> ( <i>R,G,B</i> )	Color esperado	Color resultante.
(0,0,0)	Negro	Negro
(255,255,255)	Blanco	Blanco
(255,0,0)	Rojo	Rojo
(0,255,0)	Verde	Verde
(0,0,255)	Azul	Azul
(255,255,0)	Amarillo	Amarillo
(255,0,255)	Magenta	Magenta

Los resultados expuestos en esta tabla, nos permiten concluir que los *LED* muestran los colores de manera adecuada.

## **5.7 PRUEBAS DE ACEPTACIÓN DEL SISTEMA**

Para probar el funcionamiento del robot creamos un recorrido con la presencia de diversos obstáculos, en el Laboratorio de Robótica de la Universidad Pablo de Olavide.

Para efectuar la prueba, hemos creado un pulso sonoro dando una palmada, y el robot ha iniciado correctamente el movimiento en dirección del sonido. Al detectar el primer obstáculo, el robot se encuentra en un punto mínimo local. En este momento, el robot hace parpadear los *LED* en color magenta y genera una fuerza de repulsión en el mapa de obstáculos. Esto hace que el robot se desplace lateralmente para evadir el obstáculo. Al llegar al límite del mapa contenido en memoria, el robot ha hecho parpadear las luces en color verde, y ha creado un nuevo mapa, siguiendo correctamente su camino en dirección del pulso sonoro. Por último, hemos dado otra palmada para frenar el robot, y éste ha detenido el movimiento de los motores.

Un video demostrativo de la prueba de aceptación se encuentra disponible en el siguiente enlace: <https://youtu.be/srv4xhLDheM>.

## **5.8 DEMOSTRACIÓN DIDÁCTICA PARA ALUMNOS DE EDUCACIÓN**

### **SECUNDARIA**

Por otra parte, el día 24 de mayo de 2017, dos cursos del Instituto de Enseñanza Secundaria Virgen de Valme, de la localidad de Dos Hermanas, visitaron las instalaciones de la Universidad Pablo de Olavide, para participar en una actividad organizada por el Laboratorio de Robótica. Durante este encuentro, a través de tareas didácticas y de la presentación de los proyectos de robótica desarrollados en dicha universidad, se buscó despertar la curiosidad e incentivar el interés de los alumnos por la robótica.

En este contexto, además de explicar a los alumnos las características principales de este proyecto, hicimos una demostración del funcionamiento de nuestro robot. De esa manera, tuvimos la oportunidad de probar nuestro robot en un ambiente externo al Laboratorio, es decir, en un espacio reducido y con muchos obstáculos. Durante la demostración, el robot se movió por el aula en la que tuvo lugar el evento. Si bien pudo llegar a su objetivo, sufrió dos colisiones con las patas de las mesas, de base rectangular, debido a que los *sonars* no fueron capaces de detectar sus esquinas. Sustituir los *sonars* por un sistema de detección de obstáculos más precisos como, por ejemplo, *lasers* o infrarrojos, ayudarían al robot a reducir los riesgos de impacto.

Debemos destacar que exponer nuestro robot frente a un público adolescente fue una experiencia enriquecedora, en la que pudimos notar que

gracias también a la utilización de elementos a ellos familiares, como las piezas de *Lego*, fuimos capaces de captar fácilmente su atención y despertar la curiosidad hacia nuestro trabajo (Fotografías 22 y 23).

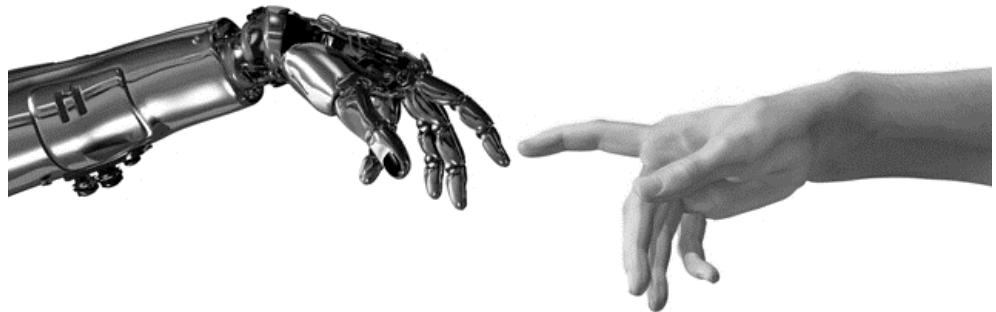


Fotografía 22: Demostración del funcionamiento del robot frente a alumnos del IES Virgen de Valme, I.



Fotografía 23 Demostración del funcionamiento del robot frente a alumnos del IES Virgen de Valme, II

# Conclusiones y desarrollos futuros





## 6 CONCLUSIONES Y DESARROLLOS FUTUROS

---

El primer objetivo establecido para este trabajo de fin de grado fue crear un robot capaz de navegar de forma autónoma en interiores, utilizando la plataforma móvil *Magabot*, basada en *Arduino*. Al elaborar el plan de proyecto, nos propusimos utilizar un pulso sonoro como guía de la trayectoria de nuestro robot.

Para cumplir con este objetivo hemos diseñado dos subsistemas conectados entre sí, basados en el uso de una placa *Arduino Due* y una *Arduino Mega 2560*. Este primer subsistema, fue desarrollado para detectar el ángulo de origen de un pulso sonoro, haciendo uso de tres micrófonos. Para realizar este subsistema, además de investigar sobre cuestiones informáticas, tuvimos que recopilar información sobre acústica, electrónica y mecánica. Si bien el subsistema de detección del ángulo de origen del pulso sonoro cumple de forma precisa con el objetivo planteado en este proyecto, en un futuro nos interesaría mejorarlo integrando las funcionalidades de la librería *OpenCV* para que el robot pueda reconocer comandos vocales, calculando la dirección de origen de una voz humana.

El segundo subsistema fue concebido para que la plataforma robótica se moviera de forma autónoma, evitando la colisión con obstáculos. Para este subsistema hemos creado un algoritmo de navegación reactiva, inspirado en el algoritmo de navegación de campos potenciales y en el *wavefront planner*. Si bien en un principio no nos propusimos el objetivo de elaborar un algoritmo original, los problemas surgidos durante la implementación de la navegación han actuado como el catalizador que nos ha llevado a producir un nuevo algoritmo, funcional y eficiente, para nuestro robot. El funcionamiento correcto del robot ha sido comprobado tanto en las pruebas desarrolladas en el Laboratorio de Robótica de la UPO, como en una demostración realizada frente a alumnos de un colegio secundario. En un próximo trabajo, sin embargo, podríamos mejorar la técnica de evasión de los mínimos locales, por ejemplo, a través del uso de algoritmos planificadores como el *Rapidly-exploring Random Tree Planner*, más conocido por su acrónimo RRT.

El segundo objetivo propuesto fue crear modelos de *MATLAB Simulink* que ejecuten los *drivers* de *Magabot*. Este conjunto de *drivers* conforma un tercer subsistema de control de bajo nivel de *Magabot*, que es el encargado de ejecutar las órdenes que provee el planificador del segundo subsistema, mencionado en el párrafo anterior. Logramos cumplir con este objetivo diseñando diagramas de bloques que pueden ser ejecutados en la placa *Arduino Mega 2560*, que se encuentra embebida en el *Magabot*, ser monitorizados, posteriormente, desde el *PC* de desarrollo con *MATLAB Simulink*. También alcanzamos crear modelos de *Simulink* capaces de llamar a las funciones del paquete de *drivers* de *Magabot*, codificado de manera manual en lenguaje *C++*. De esta forma ha sido posible comparar los tiempos de ejecución en el *hardware* del código en *C++* y del código generado automáticamente por *Simulink Coder*.

Luego de este análisis, podemos concluir que *Simulink* es un *software* que presenta una significativa practicidad no solo en lo que respecta al diseño y a la simulación de sistemas, sino también al desarrollo de código. El principal atractivo de esta herramienta radica en su potencialidad para fomentar trabajos multidisciplinarios, dado que los modelos en bloques pueden ser fácilmente comprendidos por ingenieros de distintas ramas. Asimismo, al unificar el entorno de trabajo para las fases de diseño, desarrollo y pruebas, esta herramienta puede contribuir a acelerar los tiempos de realización de proyectos de ingeniería.

Por último, cabe señalar que durante el desarrollo de la segunda fase de nuestro trabajo, nos hemos percatado de la falta de una bibliografía consolidada sobre la integración de las funcionalidades de *MATLAB Simulink* y *Arduino*. Por ello, hemos decidido documentar las actividades llevadas a cabo con *MATLAB Simulink*, explicando en detalle todo el flujo de trabajo, con el propósito de que otros puedan utilizar este material como guía.

# Bibliografía





## 7 BIBLIOGRAFÍA

---

### LIBROS

- LaValle, S. M. *Planning Algorithms*. Cambridge: Cambridge University Press, 2006
- Hexmoor, H. *Essential Principles for Autonomous Robotics. Synthesis Lectures on Artificial Intelligence and Machine Learning*. Morgan & Claypool Publishers, 2013. (DOI: <http://www.morganclaypool.com/doi/abs/10.2200/S00506ED1V01Y201305AIM021>)
- Plack, C. J. *The Sense of Hearing*. Mahwah, New Jork: Lawrence Erlbaum Associates, 2005.

### ARTÍCULOS CIENTÍFICOS

- Reyes, D.; Millán, G.; Osorio, R. y Lefranc, G. “Mobile Robot Navigation Assisted by GPS”, *IEEE Latin America Transactions*, Vol. 13, N. 6, pp. 1915 – 1920. (2015)
- Borenstein, J. y Feng, L. “Measurement and correction of systematic odometry errors in mobile robots”. *IEEE Transactions on Robotics and Automation*. Vol. 12, N. 6 (1996), pp. 869-880.

### TEXTOS ACADÉMICOS (APUNTES DE CÁTEDRA, TESIS, TFG, ETC.)

- Merino Cabañas, L. y Heredia Benot, G. “Robot Navigation, reactive methods”. Robótica y Visión Artificial. Escuela Politécnica Superior de la Universidad Pablo de Olavide, (A.A 2016/2017)
- “P-A Introducción al entorno *MATLAB-Simulink* de Diseño Basado en Modelo (DBM). Sistemas Informáticos Aeronáuticos, Máster en Ingeniería Informática, Escuela politécnica Superior, Universidad Pablo de Olavide, Sevilla (Año Académico 2016/2017).
- “P-B1 Metodología de Diseño Basado en Modelo (DBM) en el entorno *MATLAB-Simulink*”. Sistemas Informáticos Aeronáuticos, Máster en Ingeniería Informática, Escuela politécnica Superior, Universidad Pablo de Olavide, Sevilla (Año Académico 2016/2017).
- “P-B2 Metodología de Diseño Basado en Modelo (DBM) en el entorno *MATLAB-Simulink*”. Sistemas Informáticos Aeronáuticos, Máster en Ingeniería Informática, Escuela politécnica Superior, Universidad Pablo de Olavide, Sevilla (Año Académico 2016/2017).

- Safadi, H. “Local path planning using virtual potential field”. In “Spatial Representation and Mobile Robotics”, McGill University, School of Computer Science (2007)
- Falconi R., “Introduzione alla Robotica Mobile”. Universitá degli Studi di Bologna (2011)
- De Luca, A. “Corso di Robotica 2 – Metodi”. Universitá Sapienza di Roma, Dipartimento di Informatica e Sistemistica Antonio Ruberti (A.A 2008-2009)
- Wrobel, A. y Grisanti, M. “Acoustic Impulse Marker”. Cornell Univeristy, Electrical & Computer Engineering (2010)
- Lundgren, M. “Path Tracking and Obstacle Avoidance for a Miniature Robot”. Master Thesis, Department of Computer Science, Umeå University. (2003)

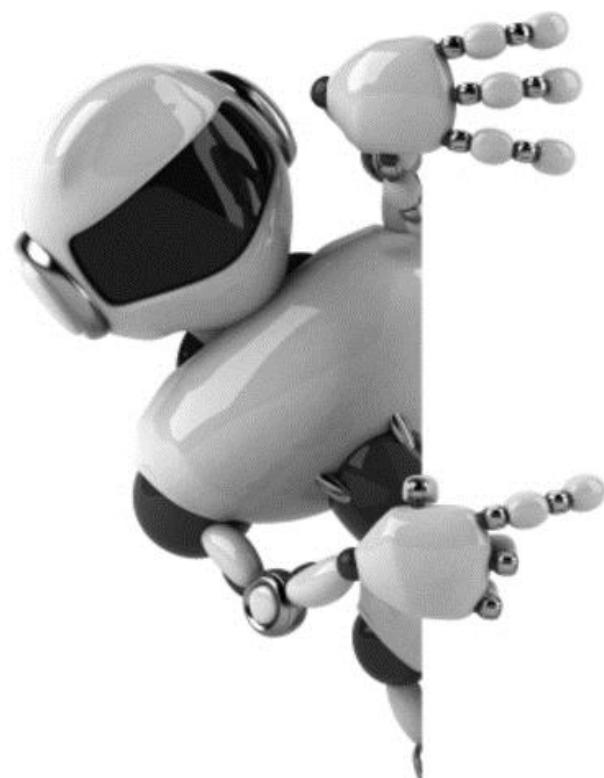
## PÁGINAS WEB

- “Arduino Reference”. En <https://www.arduino.cc/en/Reference>
- “MATLAB documentation”. En <https://www.mathworks.com/help/>
- “Mobile Robots Navigation, Mapping, and Localization”. En <http://what-when-how.com/artificial-intelligence/mobile-robots-navigation-mapping-and-localization-part-i-artificial-intelligence/>
- Ferreira, L. *et al*, “Embedded Motion Control 2015 Group 2”. En [http://cstwiki.wtb.tue.nl/index.php?title=Embedded\\_Motion\\_Control\\_2015\\_Group\\_2](http://cstwiki.wtb.tue.nl/index.php?title=Embedded_Motion_Control_2015_Group_2)
- “Speed of Sound Formulas”. En [http://www.engineeringtoolbox.com/speed-sound-d\\_82.html](http://www.engineeringtoolbox.com/speed-sound-d_82.html)
- “SRF02 Ultrasonic range finder”. En <http://www.robot-electronics.co.uk/htm/srf02tech.htm>

## MATERIAL AUDIOVISUAL

- How to integrate Arduino Libraries with MATLAB Simulink?. En [https://www.youtube.com/watch?v=\\_OLctOFjjYQ](https://www.youtube.com/watch?v=_OLctOFjjYQ)

# Anexos





# **Anexo I**

## **Códigos**





## 8.1 Anexo I: códigos

---

### 8.1.1 CÓDIGO DEL SUBSISTEMA 01: DETECCIÓN DE UN PULSO SONORO - SOUNDDETECTOR.INO

```
short mic0 = 10;
short mic1 = 4;
short mic2 = 0;
const short led = 13;
const short tempSensor = 7; // termómetro usado para
calcular la velocidad del sonido
const short umbral = 10; // diferencia en la lectura
analógica para que se tome en consideración una lectura.
float ambient0 = 0;
float ambient1 = 0;
float ambient2 = 0;
double sound_speed;
const double mic_distance = 310000.0; // distancia entre
micrófonos en um
double max_limit;
double min_limit;
// tiempo de llegada en us
unsigned long t0;
unsigned long t1;
unsigned long t2;
unsigned long timer; // el timer sirve para resetear las
lecturas si pasa mucho tiempo después de la lectura de un
micrófono y las sucesivas. Así se eliminan ciertos errores.
short count; // cuenta si se han realizado 3 lecturas

void setup() {
    Serial.begin(9600);
    blink();
    blink();
    pinMode(led, OUTPUT);
    set_sound_speed();
    max_limit = mic_distance / sound_speed;
    min_limit = mic_distance * cos(PI / 6) / sound_speed;
    count = 0;
    // se mide el ruido ambiental en cada micrófono.
    int s0 = 0;
    int s1 = 0;
    int s2 = 0;
    for (int i = 0; i < 1000; ++i) {
        s0 += analogRead(mic0);
```

```

        s1 += analogRead(mic1);
        s2 += analogRead(mic2);
    }
    ambient0 = s0 / 1000.0 ;
    ambient1 = s1 / 1000.0 ;
    ambient2 = s2 / 1000.0 ;
    blink();
}
void loop() {
    if ( abs(analogRead(mic0) - ambient0) > umbral && t0 == 0)
    {
        t0 = micros();
        ++count;
        timer = t0 + max_limit;
    }
    if ( abs(analogRead(mic1) - ambient1) > umbral && t1 == 0)
    {
        t1 = micros();
        ++count;
        timer = t1 + max_limit;
    }
    if ( abs(analogRead(mic2) - ambient2) > umbral && t2 == 0)
    {
        t2 = micros();
        ++count;
        timer = t2 + max_limit;
    }
    if ( count && micros() > timer ) {
        reset();
    } else if (count == 3) {
        short frontRear = 1; // puede tener valores 1 o -1, para
        invertir el ángulo.
        // diferencias de tiempo de llegada del sonido entre los
        micrófonos
        int delay01 = t0 - t1;
        int delay12 = t1 - t2;
        int delay20 = t2 - t0;
        if (abs(delay01) > 600 || abs(delay12) > 600 ||
abs(delay20) > 600) {
            short excluded; // micrófono excluído del cómputo del
            ángulo
            double angleOffset = 0;
            double angle = 0;
            // decidido los 2 micrófonos que voy a utilizar:
            //micrófonos 0 y 1
            int recDelay = delay01;
            int minDist = abs(recDelay);
            excluded = 2;
            if (t2 < t0) {
                angleOffset = M_PI; // sonido desde atrás
                frontRear = -1;
            } else {

```

```

        angleOffset = 0; // sonido desde adelante
        frontRear = 1;
    }
    if (abs(delay12) < minDist) {
        recDelay = delay12;
        minDist = abs(delay12);
        excluded = 0;
        if (t0 < t1) {
            angleOffset = - M_PI / 3.0;
            frontRear = -1;
        } else {
            angleOffset = 2.0 * M_PI / 3.0;
            frontRear = 1;
        }
    }
    if (abs(delay20) < minDist) {
        recDelay = t2 - t0;
        // minDist = abs(t2 - t0);
        excluded = 1;
        if (t1 < t2) {
            angleOffset = M_PI / 3.0;
            frontRear = -1;
        } else {
            angleOffset = -2.0 * M_PI / 3.0;
            frontRear = 1;
        }
    }
    set_sound_speed();
    // Calculo el ángulo
    angle = asin((recDelay * sound_speed) / mic_distance)
* frontRear + angleOffset;
    // rango del ángulo [ PI y -PI]
    if (angle > M_PI) {
        angle -= 2 * M_PI;
    } else if (angle <= - M_PI) {
        angle += 2 * M_PI;
    }
    Serial.print(angle);
}
delay(2000);
reset();
}
}

/***
    Parpadeo del led
*/
void blink() {
    for (short i = 0; i < 10; ++i) {
        digitalWrite(led, HIGH);
        delay(50);
        digitalWrite(led, LOW);
        delay(50);
}

```

```

        }

    /**
     * defino velocidad del sonido en función de la temperatura
    */
void set_sound_speed() {
    int tempSensorRead = analogRead(tempSensor);
    float tempVolt = 3.3 / 1023 * tempSensorRead;
    float temp = tempVolt * 100 - 50;
    sound_speed = sqrt( 1.4 * 286.9 * (273.15 + temp)); //  

sqrt( k * R * T)
}

/**
 * reinicio los los tiempos y el contador
*/
void reset() {
    t0 = 0;
    t1 = 0;
    t2 = 0;
    count = 0;
}

```

### **8.1.2 CÓDIGO DEL SUBSISTEMA 02: NAVEGACIÓN DEL ROBOT- ROBOTNAVIGATION.INO**

```
#include <Magabot.h>
#include <Servo.h>
#define WORLD_SIZE 5.0 // tamaño del mapa
#define MAP_PRECISION 0.1 // precisión del mapa
#define MAP_SIZE round(WORLD_SIZE/MAP_PRECISION) // lado del
mapa
#define ROI 0.4 // radio de influencia de los obstáculos
#define KFORCE 15 // coeficiente de la repulsión
#define MIN_REPULSION_RADIUS 0.2 // radio de repulsión del
mínimo local
#define LOCAL_MIN_RADIUS 0.2 // radio de detección del
mínimo local
#define POINT_SEARCH_RADIUS 0.4 // radio de búsqueda del
próximo punto
#define SONAR_DISTANCE 0.5 // distancia de lectura de los
sónares
#define MAX_SPEED 12.0 // velocidad linear máxima
#define MAX_ANGULAR_SPEED 8.0 // velocidad angular máxima
#define MAGABOT_WIDTH 0.34 // distancia entre ruedas
#define WHEEL_RADIUS 0.045 // radio de las ruedas
#define servoPin 22 // pin del servo del apuntador

/**
 * estructura que sirve para identificar unas coordenadas
 */
struct MapPoint {
    uint8_t x;
    uint8_t y;
};

/**
 * estructura que sirve para identificar un punto en el
sistema de referencia
 */
struct WorldPoint {
    double x;
    double y;
};

/**
 * estructura que sirve para indicar una posición en el
espacio de referencia,
inclusa la orientación. Se usa también para realizar
transformaciones.
*/
```

```

struct Position {
    double x;
    double y;
    double theta;
};

Servo myServo;
float sonarReadings [5]; // array que almacena las lecturas
de los sónares
Magabot robot; // define el objeto de la clase Magabot del
paquete Magabot.cpp
uint8_t direction_map [MAP_SIZE][MAP_SIZE] ; // matriz que
sirve para crear campos potenciales que atraen hacia el
objetivo
uint8_t obstacles_map [MAP_SIZE][MAP_SIZE] ; // matriz que
sirve para crear campos potenciales que repelen de los
obstáculos
Position sonarTransform[5]; // posiciones de los sónares
respecto al centro del robot
double soundAngle; // ángulo de la fuente del sonido
respecto al punto odom
double gradient; // gradiente de soundAngle
Position robotPosition; // Posición del robot respecto a
odom
WorldPoint mapCenter; // punto central del mapa respecto a
odom
long ledsTimer; // timer para apagar los LED
bool moving; // determina si el robot está en movimiento
String readString; // string donde se almacena la lectura
desde serial
WorldPoint goal; // punto más lejano en el mapa en dirección
hacia el sonido.
bool ping; // el sonar ha enviado el sonido
uint8_t active_sonar; // identificador del sonar activo
WorldPoint nextPoint; // objetivo local

void setup() { // OK
    Serial.begin(19200); // Serial de debug
    Serial1.begin(9600); // Serial para recibir el ángulo.
    Wire.begin();
    for (float rd : sonarReadings) {
        rd = 0;
    }
    stopRobot();
    setSonarsTransform();
    moving = false;
    myServo.attach(servoPin);
    ledsTimer = millis();
}
/**/

```

```

    el robot frena y se resetean el mapa de obstáculos y la
    posición de odom.
*/
void stopRobot() { // ok
    robot.actuateMotors(0, 0);
    blinkBlue();
    robotPosition = {0, 0, 0};
    resetMapCenter();
    resetObstaclesMap();
    moving = false;
}

/**
    función que define las transformaciones estáticas entre
    cada sonar y el centro del robot.
*/
void setSonarsTransform() { //OK
    sonarTransform[0] = {0.11, 0.1, PI / 4};
    sonarTransform[1] = {0.12, 0.05, PI / 6};
    sonarTransform[2] = {0.13, 0, 0};
    sonarTransform[3] = {0.12, -0.05, -PI / 6};
    sonarTransform[4] = {0.11, -0.1, -PI / 4};
}

void loop() { //OK
    checkLEDs();
    checkSerial();

    if (moving) {
        checkBumpers();
        updateRobotPosition();
        checkBorder();
        detectObstacles();
        do {
            nextPoint = getNextPoint();

            } while (checkLocalMinimum(nextPoint));

            pointGoal();
            moveTo(nextPoint);
        }
    }

/**
    función que apaga los LED
*/
void checkLEDs() { //OK
    if (ledsTimer < millis()) {
        robot.actuateLEDs(0, 0, 0);
    }
}

```

```

/**
 * función que escucha si se ha recibido un ángulo por
 * serial.
 * En caso afirmativo, si el robot está quieto, inicia el
 * movimiento hacia la dirección recibida.
 * Si el robot está en movimiento, éste se para.
 */
void checkSerial() { //OK
    while (Serial1.available()) {
        char c = Serial1.read();
        readString += c;
        delay(2);
    }
    if (readString.length() > 0) {
        if (moving) {
            stopRobot();
        } else {
            blinkRed();
            soundAngle = readString.toDouble();
            gradient = tan(soundAngle);
            movePointer(soundAngle);
            setDirectionMap();
            moving = true;
        }
        readString = "";
    }
}

/**
 * Controla si se ha activado algún bumper.
 */
void checkBumpers() { //OK
    if (robot.readBumpers()) {
        stopRobot();
    }
}

/**
 * Función que calcula el desplazamiento linear (Dc) y el
 * desplazamiento angular (Da),
 * en base al desplazamiento de cada rueda (Dr, Dl). Luego
 * actualiza la posición del robot respecto a odom.
 * Las formulas empleadas son las siguientes:
 * Dc = (Dr + Dl)/2
 * Da = (Dr - Dl)/L
 * a' = a + Da
 * x' = x + Dc * cos(Da)
 * y' = y + Dc * sin(Da)
 */
void updateRobotPosition() { //OK
    float wheelsRotation[2];
    robot.readClicks(wheelsRotation);
}

```

```

        double deltaLeft = wheelsRotation[0] * WHEEL_RADIUS;
        double deltaRight = wheelsRotation[1] * WHEEL_RADIUS;
        double deltaLinear = (deltaRight + deltaLeft) / 2.0;
        double deltaTheta = (deltaRight - deltaLeft) /
MAGABOT_WIDTH;
        robotPosition.theta += deltaTheta;
        robotPosition.x += deltaLinear * cos(robotPosition.theta);
        robotPosition.y += deltaLinear * sin(robotPosition.theta);
    }

/**
     Función que comprueba si el robot esá cerca del borde del
mapa.
     En caso afirmativo, se vuelve a crear un mapa alrededor
del robot.
*/
void checkBorder() { //OK
    MapPoint robotMapPoint = getMapPoint({robotPosition.x,
robotPosition.y});
    if (robotMapPoint.x >= MAP_SIZE - 5 || robotMapPoint.y >=
MAP_SIZE - 5 || robotMapPoint.x <= 4 || robotMapPoint.y <=
4) {
        robot.actuateMotors(0, 0);
        blinkGreen();
        resetMapCenter();
        setDirectionMap();
        resetObstaclesMap();
    }
}

/**
     Detecta obstáculos. Si algún sonar detecta obstáculos,
se crea repulsión alrededor de cada uno.
     Es necesario transformar las lecturas de los sónares
primero respecto a la posición del robot, y loego hacia
odom.
*/
void detectObstacles() { //OK
    uint8_t i = robot.getSonarReadings(sonarReadings);
    if (i >= 0 && sonarReadings[i] >= 0.05 && sonarReadings[i]
<= SONAR_DISTANCE) {
        WorldPoint obst_robot =
transformPoint(sonarTransform[i], {sonarReadings[i], 0});
        WorldPoint obst_odom = transformPoint(robotPosition,
obst_robot);
        addObstacle(obst_odom);
    }
}
/***
     busca el próximo punto hacia donde moverse en un radio
POINT_SEARCH_RADIUS
*/
WorldPoint getNextPoint() { //OK

```

```

WorldPoint nextPoint = {robotPosition.x, robotPosition.y};
MapPoint robotMapPoint = getMapPoint(nextPoint);
uint16_t nextPointRepulsion = getForce(robotMapPoint);
for (uint8_t i = robotMapPoint.x -
ceil(POINT_SEARCH_RADIUS / MAP_PRECISION); i <
robotMapPoint.x + ceil(POINT_SEARCH_RADIUS / MAP_PRECISION);
i++) { // desde 1m menos del obstaculo en el eje x
    for (uint8_t j = robotMapPoint.y -
ceil(POINT_SEARCH_RADIUS / MAP_PRECISION); j <
robotMapPoint.y + ceil(POINT_SEARCH_RADIUS / MAP_PRECISION);
j++) { // desde 1m menos del obstaculo en el eje y
        if (i >= 0 && i < MAP_SIZE && j >= 0 && j < MAP_SIZE)
{ // compruebo que no se salga del rango del mapa
            WorldPoint thisPoint = getWorldPoint({i, j});
            if (euclideanDistance(thisPoint, {robotPosition.x,
robotPosition.y}) <= POINT_SEARCH_RADIUS) {
                uint16_t tempRepulsion = getForce({i, j});
                if (tempRepulsion < nextPointRepulsion) {
                    nextPoint = thisPoint;
                    nextPointRepulsion = tempRepulsion;
                }
            }
        }
    }
}
return nextPoint;
}

/**
En el caso de que el próximo punto se encuentre en un
radio LOCAL_MIN_RADIUS alrededor del centro del robot,
se crea repulsión cada medio segundo en un radio
MIN_REPULSION_RADIUS alrededor del robot.
*/
bool checkLocalMinimum( WorldPoint nextPoint) { //OK
    bool isMin = false;
    if (euclideanDistance(nextPoint, {robotPosition.x,
robotPosition.y}) <= LOCAL_MIN_RADIUS) {
        blinkPurple();
        MapPoint robotMapPoint = getMapPoint({robotPosition.x,
robotPosition.y});
        for (uint8_t i = robotMapPoint.x -
ceil(MIN_REPULSION_RADIUS / MAP_PRECISION); i <
robotMapPoint.x + ceil(MIN_REPULSION_RADIUS /
MAP_PRECISION); ++i) {
            for (uint8_t j = robotMapPoint.y -
ceil(MIN_REPULSION_RADIUS / MAP_PRECISION); j <
robotMapPoint.y + ceil(MIN_REPULSION_RADIUS /
MAP_PRECISION); ++j) {
                if (i >= 0 && i < MAP_SIZE && j >= 0 && j <
MAP_SIZE) {
                    MapPoint tempMapPoint = {i, j};

```

```

        WorldPoint tempWorldPoint =
getWorldPoint(tempMapPoint);
        double distance =
euclideanDistance(tempWorldPoint, {robotPosition.x,
robotPosition.y});
        if (obstacles_map[i][j] < 254 && distance <=
MIN_REPULSION_RADIUS) {
            ++obstacles_map[i][j];
        }
    }
}
isMin = true;
}
return isMin;
}

/**
 * el apuntador apunta hacia el punto objetivo
 */
void pointGoal() { //OK
    double goalAngle = atan2(goal.y - robotPosition.y, goal.x
- robotPosition.x) - robotPosition.theta;
    movePointer(goalAngle);
}

/**
 * Función para mover el robot y orientar el apuntador en
dirección del movimiento.
*/
void moveTo(WorldPoint nextPoint) { //OK
    //ángulo del objetivo local, respecto al robot
    double nextPointAngle = atan2(nextPoint.y -
robotPosition.y, nextPoint.x - robotPosition.x) -
robotPosition.theta;
    int angularSpeed = MAX_ANGULAR_SPEED *
sin(nextPointAngle);
    int linearSpeed = MAX_SPEED - MAX_SPEED *
(abs(angularSpeed) / MAX_ANGULAR_SPEED); // limitador de
velocidad en base al ángulo del próximo objetivo.
    robot.actuateMotors(linearSpeed - angularSpeed,
linearSpeed + angularSpeed);
}

/**
 * devuelve la suma de las dos matrices en el punto
indicado.
*/
uint16_t getForce(MapPoint p) { //ok
    return (uint16_t) direction_map [p.x][p.y] +
(uint16_t) obstacles_map [p.x][p.y];
}

```

```

/**
    Mueve el servo del apuntador
*/
void movePointer(double angle) { //ok
    if (angle < -PI) {
        angle += 2 * PI;
    } else if (angle > PI) {
        angle -= 2 * PI;
    }

    double servoAngle = round(((angle * 180 / PI) + 180) / 2);
    if (servoAngle < 0) {
        servoAngle = 0;
    } else if (angle > 179) {
        servoAngle = 179;
    }
    myServo.write(servoAngle);
}

/**
    Se pone repulsión alrededor de un obstáculo en el radio
ROI
*/
void addObstacle(WorldPoint obstacle_worldPoint) {
    MapPoint obstacle_mapPoint =
getMapPoint(obstacle_worldPoint);
    obstacles_map[obstacle_mapPoint.x][obstacle_mapPoint.y] =
255;
    for (uint8_t i = obstacle_mapPoint.x - ceil(ROI /
MAP_PRECISION); i < obstacle_mapPoint.x + ceil(ROI /
MAP_PRECISION); i++) {
        for (uint8_t j = obstacle_mapPoint.y - ceil(ROI /
MAP_PRECISION); j < obstacle_mapPoint.y + ceil(ROI /
MAP_PRECISION); j++) {
            if (i >= 0 && i < MAP_SIZE && j >= 0 && j < MAP_SIZE)
{ // compruebo que no se salga del rango del mapa
                double dist = euclideanDistance(getWorldPoint({i,
j}), obstacle_worldPoint);
                // if (i != obstacle_mapPoint.x && j !=
obstacle_mapPoint.y && dist <= ROI) {
                    if ( dist != 0 && dist <= ROI) {
                        uint16_t force16 = round((ROI - dist) * KFORCE /
dist) ;
                        uint8_t force;
                        if (force16 <= 255) {
                            force = force16;
                        } else {
                            force = 255;
                        }

                        if (obstacles_map[i][j] < force) {
                            obstacles_map[i][j] = force;

```

```

        }
    }
}

}

/***
    función que recibe un punto p y lo transforma,
dependiendo de la posición de pos respecto al origen
*/
WorldPoint transformPoint(Position pos, WorldPoint p) //ok
{
    double s = sin(pos.theta);
    double c = cos(pos.theta);
    // aplico rotación
    double xnew = p.x * c - p.y * s;
    double ynew = p.x * s + p.y * c;
    // traslado el punto
    xnew += pos.x;
    ynew += pos.y;
    return {xnew, ynew};
}

void resetObstaclesMap() { //
    for (uint8_t i = 0; i < MAP_SIZE; i++) {
        for (uint8_t j = 0; j < MAP_SIZE; j++) {
            obstacles_map[i][j] = 0;
        }
    }
}

/***
    distancia euclídea entre dos puntos
*/
double euclideanDistance(WorldPoint p1, WorldPoint p2) {
//OK
    return sqrt(pow(p1.x - p2.x, 2) + pow(p1.y - p2.y, 2));
}

/***
    Función que crea la atracción hacia el sonido.
    Usa como objetivo el punto más lejano en el mapa en
dirección del sonido.
    Para cada punto del mapa, calcula la distancia en dm del
punto al objetivo y lo suma en la matriz direction_map .
    Distancia de P(x0, x1) a recta y = mx
    d = abs(y0 - m * x0)/ sqrt(1+ pow(m,2))
*/
void setDirectionMap() {
    double max_x = mapCenter.x + WORLD_SIZE / 2.0;
    double min_x = mapCenter.x - WORLD_SIZE / 2.0;

```

```

double max_y = mapCenter.y + WORLD_SIZE / 2.0;
double min_y = mapCenter.y - WORLD_SIZE / 2.0;
if (abs(soundAngle) < PI / 2.0) {
    goal = {max_x, max_x * gradient };
} else if ( abs(soundAngle) > PI / 2.0) {
    goal = {min_x, min_x * gradient };
} else if ( soundAngle > 0) {
    goal = {max_y / gradient , max_y};
} else {
    goal = {min_y / gradient , min_y};
}

for (uint8_t i = 0; i < MAP_SIZE; i++) {
    for (uint8_t j = 0; j < MAP_SIZE; j++) {

        WorldPoint p = getWorldPoint({i, j});
        direction_map[i][j] = round(10 * euclideanDistance(p,
goal));
    }
}
/***
    recibe unas coordenadas del mapa y devuelve el punto
central respecto a odom.
*/
WorldPoint getWorldPoint(MapPoint mPoint) {
    WorldPoint wPoint;
    wPoint.x = ((mPoint.x - MAP_SIZE / 2.0) * MAP_PRECISION) +
mapCenter.x ;
    wPoint.y = ((mPoint.y - MAP_SIZE / 2.0) * MAP_PRECISION) +
mapCenter.y;
    return wPoint;
}

/***
    recibe un punto respecto a odom y devuelve las
coordenadas de este punto en el mapa.
*/
MapPoint getMapPoint(WorldPoint wPoint) {
    struct MapPoint mapPoint;
    mapPoint.x = round((wPoint.x - mapCenter.x ) /
MAP_PRECISION) + MAP_SIZE / 2;
    mapPoint.y = round((wPoint.y - mapCenter.y ) /
MAP_PRECISION) + MAP_SIZE / 2;
    return mapPoint;
}

/***
    función que pone como centro del mapa a la posición
actual del robot respecto a odom

```

```

*/
void resetMapCenter() {
    mapCenter = {robotPosition.x, robotPosition.y};
}

/**
     enciende los LED de color rojo
*/
void blinkRed() {
    robot.actuateLEDs(255, 0, 0);
    ledsTimer = millis() + 1000;
}

/**
     enciende los LED de color verde
*/
void blinkGreen() {
    robot.actuateLEDs(0, 255, 0);
    ledsTimer = millis() + 1000;
}

/**
     enciende los LED de color azul
*/
void blinkBlue() {
    robot.actuateLEDs(0, 0, 255);
    ledsTimer = millis() + 1000;
}

/**
     enciende los LED de color magenta
*/
void blinkPurple() {
    robot.actuateLEDs(255, 0, 255);
    ledsTimer = millis() + 500;
}

/**
     Métodos de test
*/
void printPosition() {
    Serial.print(robotPosition.x * 100);
    Serial.print("\t");
    Serial.print(robotPosition.y * 100);
    Serial.print("\t");
    Serial.print(robotPosition.theta);
    Serial.println("");
}

void sonarsTest() {
    robot.getSonarReadings(sonarReadings);
    for (uint8_t i = 0; i < 5; i++) {

```

```

        Serial.print(sonarReadings[i]);
        Serial.print("\t");

    }
    Serial.println("");
}

void printDirectionMap() {
    for (uint8_t i = MAP_SIZE - 1; i >= 0; --i) {
        for (uint8_t j = MAP_SIZE - 1; j >= 0; --j) {
            Serial.print(direction_map[i][j]);
            Serial.print(" ");
        }
        Serial.println("");
    }
}

void printObstaclesMap() {
    MapPoint r = getMapPoint({robotPosition.x,
robotPosition.y});
    obstacles_map[r.x][r.y] = 8;
    Serial.println("");
    Serial.println("");
    for (uint8_t i = MAP_SIZE - 1; i >= 0; --i) {
        for (uint8_t j = MAP_SIZE - 1; j >= 0; --j) {
            Serial.print(obstacles_map[i][j]);
            Serial.print(" ");
        }
        Serial.println("");
    }
    moving = false;
}

void printPointsInMap() {
    MapPoint r = getMapPoint({robotPosition.x,
robotPosition.y});
    obstacles_map[r.x][r.y] = 8;
    for (uint8_t i = MAP_SIZE - 1; i >= 0; --i) {
        for (uint8_t j = MAP_SIZE - 1; j >= 0; --j) {
            WorldPoint wp = getWorldPoint({i, j});
            Serial.print("[");
            Serial.print(wp.x );
            Serial.print(" , ");
            Serial.print("] ");
        }
        Serial.println("");
    }
    moving = false;
}

void printAllMaps() {
    MapPoint r = getMapPoint({robotPosition.x,
robotPosition.y});

```

```

obstacles_map[r.x][r.y] = 8;
Serial.println("");
Serial.println("");
for (uint8_t i = MAP_SIZE - 1; i >= 0; --i) {
    for (uint8_t j = MAP_SIZE - 1; j >= 0; --j) {
        Serial.print((int)obstacles_map[i][j] +
(int)direction_map[i][j]);
        Serial.print(" ");
    }
    Serial.println("");
}
moving = false;

}

void straightMotorsTest(double distance) {
    while (robotPosition.x < distance) {

        robot.actuateMotors(5, 5);
        updateRobotPosition();

    }
    robot.actuateMotors(0, 0);
    updateRobotPosition();

    printPosition();

}
void squareTest(double side, bool clockwise) {
    WorldPoint p0 = {0, 0};
    WorldPoint p1 = {side, 0};
    WorldPoint p2;
    WorldPoint p3;
    if (clockwise) {
        WorldPoint p1 = {side, 0};
        p2 = {side, - side};
        p3 = {0, -side};
    }
    else {
        p2 = {side, side};
        p3 = {0, side};
    }

    WorldPoint corners [] = {p1, p2, p3, p0};
    for (WorldPoint c : corners) {
        int maxspeed = 5;
        while (euclideanDistance({robotPosition.x,
robotPosition.y}, c) > 0.05) {
            updateRobotPosition();
            double nextPointAngle = atan2(c.y - robotPosition.y,
c.x - robotPosition.x) - robotPosition.theta;
            int angularSpeed = ceil(MAX_ANGULAR_SPEED *
sin(nextPointAngle));

```

```
    int linearSpeed = maxspeed - maxspeed *  
    (abs(angularSpeed) / MAX_ANGULAR_SPEED); // limitador de  
    velocidad en base al ángulo del próximo objetivo.  
    robot.actuateMotors(linearSpeed - angularSpeed,  
    linearSpeed + angularSpeed);  
    updateRobotPosition();  
}  
}  
robot.actuateMotors(0, 0);  
updateRobotPosition();  
  
printPosition() ;  
  
while (true) {  
}  
}
```

## **8.1.3 CÓDIGOS DE LOS DRIVERS DE MAGABOT**

### ***8.1.3.1. Código del fichero Magabot.h***

```
#ifndef Magabot_h
#define Magabot_h
#include <Arduino.h>
#include <Wire.h>
#define REGISTER_CONFIG (16)
#define REGISTER_OUTPUT (16)

class Magabot
{
public:
    Magabot();
    void actuateMotors(int vel1, int vel2);
    void actuateLEDs(int Red, int Green, int Blue);
    void getSonarReadings(float* sonarReadings);
    bool readBumpers();
    void readClicks(float * wheelsRotation);

private:
    int _sonarId;
    bool _ping;
    unsigned long _sonarTimer;
    int _clicksPerTurn;

};

#endif
```



### 8.1.3.2 Código del fichero Magabot.cpp

```
#include "Magabot.h"
#include <inttypes.h>
#include <Math.h>

Magabot::Magabot() {
    // initialize the digital pin as an output.
    //RGB led pins
    pinMode(11, OUTPUT);
    pinMode(10, OUTPUT);
    pinMode(9, OUTPUT);
    //Bumpers
    pinMode(2, INPUT);
    pinMode(3, INPUT);
    pinMode(4, INPUT);
    pinMode(5, INPUT);
    Wire.begin();
    _clicksPerTurn = 3900;
    sonarId = 0;
    _ping = false;
    _sonarTimer = millis();
}

//*****
//*****Actuate motors*****
//*****
void Magabot::actuateMotors(int vel1, int vel2) {
    vel2 = vel2 * -1;
    byte v1b1 = vel1 >> 8;
    byte v1b2 = vel1 & 0xFF;
    byte v2b1 = vel2 >> 8;
    byte v2b2 = vel2 & 0xFF;
    Wire.beginTransmission(0x15);
    Wire.write((byte) 0);
    Wire.write(v1b1);
    Wire.write(v1b2);
    Wire.endTransmission();
    Wire.beginTransmission(0x16);
    Wire.write((byte) 0);
    Wire.write(v2b1);
    Wire.write(v2b2);
    Wire.write(1);
    Wire.endTransmission();
}

//*****
//*****Actuate LEDs*****
//*****
void Magabot::actuateLEDs(int Red, int Green, int Blue) {
    Red = 255 - Red;
    Green = 255 - Green;
```

```

        Blue = 255 - Blue;
        analogWrite(11, (unsigned char) Green);
        analogWrite(10, (unsigned char) Red);
        analogWrite(9, (unsigned char) Blue);
    }

//*****
//*****SONARS READ *****/
//*****


uint8_t Magabot::getSonarReadings(float* sonarReadings) {
    uint8_t returnedID = -1;

    if (_ping) {
        while (millis() < _sonarTimer) {
        }
        Wire.beginTransmission(0x71 + _sonarId); // transmit
        to device #112
        Wire.write(0x02); // sets register pointer to echo #1
        register (0x02)
        Wire.endTransmission(); // stop transmitting
        int value;
        Wire.requestFrom(0x71 + _sonarId, 2); // request 2
        chars from slave device #112
        // receive _sonarReading from sensor
        if (2 <= Wire.available()) // if two chars were
        received
        {
            value = Wire.read(); // receive high char
            (overwrites previous _sonarReading)
            value = value << 8; // shift high char to be high
            8 bits
            value |= Wire.read(); // receive low char as lower
            8 bits
            sonarReadings [_sonarId] = value / 100.0;
        }
        returnedID = _sonarId;
        _sonarId = (_sonarId + 1) % 5;
    }
    Wire.beginTransmission(0x71 + sonarId);
    Wire.write((byte) 0);
    Wire.write(0x51);
    Wire.endTransmission();
    _sonarTimer = millis() + 60;
    _ping = true;
    return returnedID;
}
//*****
//****Front bumpers read function*****/
//*****


bool Magabot::readBumpers() {

```

```

        return digitalRead(2) != 1 || digitalRead(3) != 1 ||
digitalRead(4) != 1 || digitalRead(5) != 1;
}
//*****
//***** Odometer *****/
//***** *****/
void Magabot::readClicks(float * wheelsRotation) {
    short leftClicks = 0;
    short rightClicks = 0;
    Wire.beginTransmission(0x15);
    Wire.write(0x19);
    Wire.write(1);
    Wire.endTransmission();
    delay(1);
    Wire.beginTransmission(0x16);
    Wire.write(0x19);
    Wire.write(1);
    Wire.endTransmission();
    delay(1);
    Wire.beginTransmission(0x15); // transmit to device 0x15
    Wire.write(0x15); // sets register pointer to echo #1
register (0x15)
    Wire.endTransmission();
    Wire.requestFrom(0x15, 2);
    if (2 <= Wire.available()) // if two chars were received
    {
        leftClicks = (short) ((Wire.read() << 8) +
Wire.read());
    }
    Wire.beginTransmission(0x16); // transmit to device 0x16
    Wire.write(0x15); // sets register pointer to echo #1
register (0x15)
    Wire.endTransmission();
    Wire.requestFrom(0x16, 2);
    if (2 <= Wire.available()) // if two chars were received
    {
        rightClicks = (short) ((Wire.read() << 8) +
Wire.read());
    }
    Wire.beginTransmission(0x15);
    Wire.write(0x14);
    Wire.write((byte) 0);
    Wire.endTransmission();
    delay(1);
    Wire.beginTransmission(0x16);
    Wire.write(0x14);
    Wire.write((byte) 0);
    Wire.endTransmission();
    delay(1);
    wheelsRotation[0] = leftClicks * -2 * PI / _clicksPerTurn;
    wheelsRotation[1] = rightClicks * 2 * PI / _clicksPerTurn;
}

```



# **Anexo II**

# **Plan de Proyecto**





## **8.2 ANEXO II: PLAN DE PROYECTO**

---

**Título:** Robot de navegación autónoma guiado por un pulso sonoro

**Autor:** Marco Corzetto Conflan

**Fecha:** 30/01/2017

# ÍNDICE DEL PLAN DE PROYECTO

1. Introducción .....	187
2. Objetivos del proyecto .....	188
3. Organización del proyecto .....	189
3.1. Identificación de los interesados .....	189
3.2. Responsabilidades y funciones de los interesados .....	189
4. Metodología de gestión del proyecto .....	190
5. Programa de trabajo .....	191
5.1. Alcance y objetivos del programa de trabajo .....	191
5.2. Plan de tareas .....	191
5.2.1. Diagrama <i>EDT (EWS)</i> .....	191
5.2.2. Descripción de las tareas .....	193
5.2.2.1. Análisis .....	193
5.2.2.2. Diseño .....	194
5.2.2.3. Implementación .....	196
5.2.2.4. Pruebas .....	197
5.2.2.5. Cierre .....	198
5.2.3. Diagrama de Gantt .....	199
5.2.4. Estimación de coste .....	205
6. Evaluación y planificación de riesgos .....	205
7. Otros aspectos del proyecto .....	207
7.1. Ubicación del equipo de desarrollo .....	207

## 1. INTRODUCCIÓN

Este documento es el plan de desarrollo del proyecto titulado “Robot de navegación autónoma guiado por un pulso sonoro”. La finalidad de este proyecto es realizar el trabajo de fin de grado necesario para la obtención del Grado en Ingeniería Informática en Sistema de Información, de la Escuela Politécnica Superior de la Universidad Pablo de Olavide, Sevilla.

El primer objetivo del proyecto es crear un robot de navegación autónomo que detecte el ángulo de origen de un sonido y se mueve en esa dirección. En su trayectoria, que debe desarrollarse en un terreno liso y sin desniveles, el robot debe esquivar obstáculos fijos.

Para poder cumplir con este objetivo se trabajará sobre la plataforma *Magabot*, producida por la empresa portuguesa *IDMind*. *Magabot* es un robot móvil, basado en la placa *Arduino Uno*, que cuenta, además, con: tracción diferencial, dos motores DC, dos *encoders*, tres sensores *IR*, dos *bumpers*, cinco *sonars* y tres *LED RGB*.

El segundo objetivo de este proyecto es, por un lado, utilizar el software *MATLAB*, con las extensiones *Coder* y *Simulink*, para generar código de los *drivers* de *Magabot*. Por otro lado, se busca realizar un análisis comparado del rendimiento del código en lenguaje *C++* del paquete *Magabot.h* con los códigos autogenerados con *MATLAB Coder*.

## 2. OBJETIVOS DEL PROYECTO

En las siguientes tablas se enuncian los objetivos principales del trabajo.

<b>OBJ–01: Navegación autónoma orientada por un sonido</b>	
<b>Descripción</b>	Crear un sistema robótico de navegación que sea capaz de detectar el origen de un sonido, para moverse hacia él. A medida que avanza en esa dirección, el robot debe evitar colisionar con obstáculos estáticos.
<b>Importancia</b>	Alta
<b>Comentarios</b>	Implementar el algoritmo de navegación, la localización del sonido y los <i>drivers</i> , en placas de desarrollo <i>Arduino</i> .

<b>OBJ–02: Implementación de <i>drivers</i> de <i>Magabot</i> con <i>MATLAB Coder</i> y <i>Simulink</i></b>	
<b>Descripción</b>	Crear modelos de <i>MATLAB Simulink</i> que ejecuten los <i>drivers</i> del <i>Magabot</i> , y generar el código de los mismos en lenguaje <i>C</i> con <i>MATLAB Embedded Coder</i> para realizar un análisis comparado con los <i>drivers</i> escritos en lenguaje <i>C++</i> .
<b>Importancia</b>	Alta
<b>Comentarios</b>	Considerando que esta propuesta tiene la finalidad de investigar las funcionalidades avanzadas de <i>MATLAB</i> para la programación de placas <i>Arduino</i> , se creará una guía para futuros proyectos basados en <i>Simulink</i> y <i>Arduino</i> .

### **3 ORGANIZACIÓN DEL PROYECTO**

Este plan de proyecto fue elaborado para ser producido por un único alumno, quien cumple el rol de responsable principal del proyecto. El alumno en cuestión trabajará bajo la tutorización de dos profesores. De esa manera, el proyecto queda organizado del siguiente modo:

- Alumno responsable del proyecto: Marco Corzetto Conflan
- Tutores: D. Manuel Béjar Domínguez
- D. José Antonio Cobano Suárez

#### **3.1. IDENTIFICACIÓN DE LOS INTERESADOS**

Este plan de proyecto tiene como principales interesados a Marco Corzetto Conflan, quien a través de este proyecto busca obtener el Grado de Ingeniería Informática en Sistema de Información, y a los profesores Béjar Domínguez y Cobano Suárez, quienes actúan como tutores del trabajo de fin de grado.

#### **3.2. RESPONSABILIDADES Y FUNCIONES DE LOS INTERESADOS**

Corzetto Conflan tiene la responsabilidad de cumplir con los objetivos trazados en el proyecto en su totalidad, es decir, debe encargarse de diseñar, desarrollar y probar el robot y, posteriormente, trabajar con el programa *MATLAB Simulink*, para cumplir con los propósitos arriba señalados. Con respecto a los objetivos, cabe aclarar que fueron definidos en forma conjunta con los tutores.

Los profesores Béjar Domínguez y Cobano Suárez, tienen la función de dirigir tanto la planificación como el desarrollo del proyecto. Es necesario especificar que si bien se trata de un único proyecto, la tutorización del primer objetivo está a cargo del profesor Cobano Suárez, mientras que el profesor Béjar Domínguez será el encargado de dirigir el desarrollo de segundo objetivo. Para ello, se realizarán reuniones de seguimiento periódicas, en las que ambos tutores no solo evaluarán los resultados obtenidos, sino que también brindarán asesoría sobre posibles mejoras y resolución de problemas. Asimismo, una vez cumplidas todas las etapas trazadas en este plan de proyecto, deben revisar la memoria final y otorgar su visto bueno para autorizar la defensa del trabajo de fin de grado.

## 4. METODOLOGÍA DE GESTIÓN DEL PROYECTO

Para realizar un seguimiento eficiente de las tareas a desarrollar en este proyecto, se aplicará la metodología ágil denominada *Kanban*. Conviene recordar en este punto que *Kanban* es una herramienta construida para controlar el flujo de trabajo e información de un proceso de desarrollo de manera óptima. Las tres normas básicas que *Kanban* establece –a saber: visualizar el flujo de trabajo, limitar el *Work in Progress*, y medir y mejorar el flujo– guiarán metodológicamente nuestro proyecto. Para ello, se utilizará el tablero *Kanban on line* llamado *KanbanFlow* (disponible en: <https://kanbanflow.com/>), que se personalizará con el fin de organizar las acciones a desarrollar con etiquetas, que permitan visualizar, controlar y optimizar el flujo de trabajo.

Las reuniones de seguimiento se efectuarán con una periodicidad quincenal, preferentemente, en el Laboratorio de Robótica de la UPO, lugar en el que está la plataforma *Magabot*. En esas reuniones Corzetto Conflan presentará los avances, respondiendo a las preguntas de los tutores. Asimismo, deberá exponer el plan de tareas a desarrollar durante la quincena, y se fijará la fecha de la reunión sucesiva. No se ha definido un modelo de informe de seguimiento.

## **5. PROGRAMA DE TRABAJO**

### **5.1. ALCANCE Y OBJETIVOS DEL PROGRAMA DE TRABAJO**

El programa de trabajo que se detalla en este apartado abarca el proyecto en su totalidad, desde la fase de inicio hasta la entrega final.

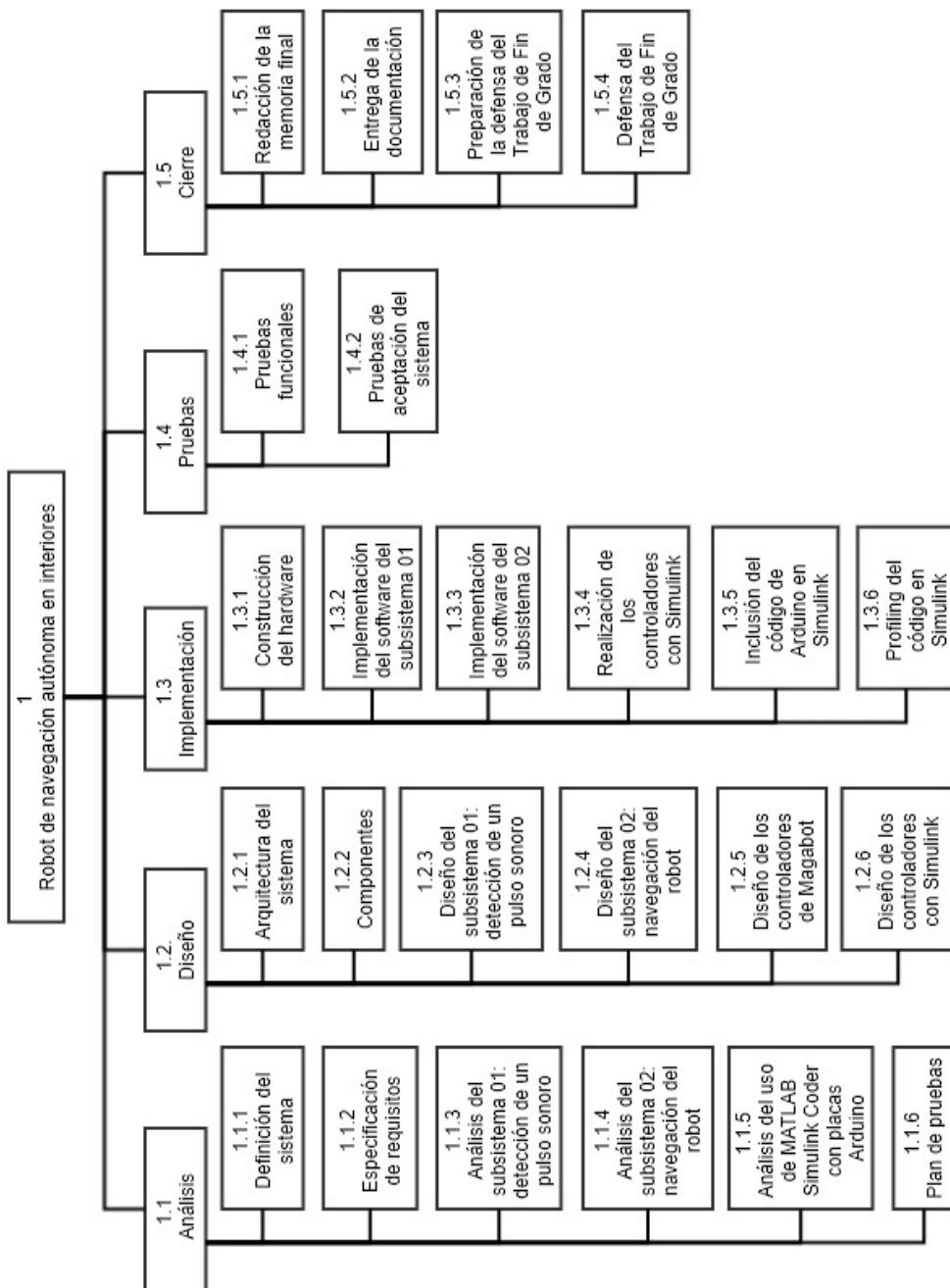
Los objetivos del mismo son los señalados en el apartado 2 de este plan de proyecto.

### **5.2. PLAN DE TAREAS**

#### **5.2.1. Diagrama EDT (EWS)**

A continuación, se incluye el diagrama EDT (WBS) del proyecto, en el que se desglosan las fases y tareas en las que se organizará el flujo de trabajo (Figura 1).

Figura 1: EDT del proyecto “Robot de navegación autónoma guiado por un pulso sonoro”



## 5.2.2. Descripción de las tareas

### 5.2.2.1. Análisis

En la fase de análisis se llevará a cabo una actualización bibliográfica, con el fin de conocer el estado del arte sobre nuestro tema de investigación. Sobre la base de esta pesquisa, se definirán las categorías de análisis principales del proyecto.

Código	Nombre	Estimación
<b>1.1.1</b>	<b>Definición del sistema</b>	3 días

**Descripción**  
Se procederá a definir el alcance del sistema y los objetivos a seguir.

Código	Nombre	Estimación
<b>1.1.2</b>	<b>Establecimiento de los requisitos</b>	2 días

**Descripción**  
Se definirán los requisitos funcionales y no funcionales del sistema.  
Se elaborará una matriz de trazabilidad de requisitos y objetivos.  
Se identificarán los casos de uso, y se procederá a su diagramación y descripción detallada.

Código	Nombre	Estimación
<b>1.1.3</b>	<b>Análisis del subsistema 01: detección de un pulso sonoro</b>	5 días

**Descripción**  
Se investigará sobre acústica y electrónica.  
Se establecerá qué técnicas aplicar para la detección del origen del pulso sonoro.

Código	Nombre	Estimación
<b>1.1.4</b>	<b>Análisis del subsistema 02: navegación del robot</b>	6 días

**Descripción**  
Se realizará una búsqueda bibliográfica sobre algoritmos de navegación autónoma.  
Se elegirá un algoritmo para la implementación de nuestro sistema.

Código	Nombre	Estimación
<b>1.1.5</b>	<b>Análisis del uso de MATLAB Simulink Coder con placas Arduino</b>	6 días
<b>Descripción</b>		
Se estudiarán las funcionalidades que ofrece el software <i>MATLAB Simulink</i> . Se investigarán las posibilidades de integración de este software con las placas <i>Arduino</i> .		

Código	Nombre	Estimación
<b>1.1.6</b>	<b>Plan de pruebas</b>	2 días
<b>Descripción</b>		
Se identificarán cuáles serán las pruebas necesarias para la comprobación del correcto funcionamiento del sistema. Se hará una descripción de cada una de las prueba.		

### 5.2.2.2. Diseño

En la fase del proyecto se diseñarán las características principales del sistema, tanto en lo que se respecta a los componentes como al diseño de la estructura y funcionamiento del programa.

Código	Nombre	Estimación
<b>1.2.1</b>	<b>Arquitectura del sistema</b>	3 días
<b>Descripción</b>		
Se definirán los subsistemas que conformarán la arquitectura del sistema Se establecerá y diagramará el esquema de conexiones del <i>hardware</i> .		

Código	Nombre	Estimación
<b>1.2.2</b>	<b>Componentes</b>	4 días
<b>Descripción</b>		
Se elaborará una lista que desglose cada uno de los elementos necesarios para la composición del <i>hardware</i> , con el detalle de sus características principales.		

Código	Nombre	Estimación
<b>1.2.3</b>	<b>Diseño del subsistema 01: detección de un pulso sonoro</b>	3 días

**Descripción**  
Se diagramará la disposición de los componentes del *hardware*.  
Se crearán el diagrama de flujo del subsistema 01.  
Se definirán y detallarán las funciones del subsistema 01.

Código	Nombre	Estimación
<b>1.2.4</b>	<b>Diseño del subsistema 02: navegación del robot</b>	5 días

**Descripción**  
Se elaborará un diagrama con la configuración diferencial de *Magabot* y de la estructura interna de los *encoders*.  
Se diseñará el diagrama de flujo del subsistema 02.  
Se detallará la estructura de datos del subsistema 02.  
Se definirán y detallarán las funciones del subsistema 02.

Código	Nombre	Estimación
<b>1.2.5</b>	<b>Diseño de los <i>drivers</i> de <i>Magabot</i></b>	1 día

**Descripción**  
Se estudiarán los componentes y métodos del paquete *Magabot.h*.  
Se establecerán los elementos a modificar del paquete de *drivers* de *Magabot*.

Código	Nombre	Estimación
<b>1.2.6</b>	<b>Diseño de los <i>drivers</i> con <i>Simulink</i></b>	7 días

**Descripción**  
Se diagramará el modelado de los *drivers* de *Simulink* para los motores y los *encoders*.

### 5.2.2.3. Implementación

En esta fase, se realizará la construcción del robot y, a posterioridad, se implementará el código. También, durante esta fase, se trabajará con *MATLAB Simulink*.

Código	Nombre	Estimación
1.3.1	Construcción del <i>hardware</i>	4 días
<b>Descripción</b>		
Se procederá a la construcción física del robot. Se montará la estructura y se realizarán las conexiones el <i>hardware</i> .		

Código	Nombre	Estimación
1.3.2	Implementación del software del subsistema 01	8 días
<b>Descripción</b>		
Se implementará el código del subsistema 01, encargado de detectar el origen de un pulso sonoro.		

Código	Nombre	Estimación
1.3.3	Implementación del software del subsistema 02	20 días
<b>Descripción</b>		
Se implementará el código del subsistema 01, encargado de realizar la navegación autónoma del robot.		

Código	Nombre	Estimación
1.3.4	Realización de los <i>drivers</i> con <i>Simulink</i>	14 días
<b>Descripción</b>		
Se utilizará el software <i>MATLAB Simulink</i> para crear los <i>drivers</i> de <i>Magabot</i> , usando diseños basados en modelo. Se redactará una guía de las acciones realizadas con <i>MATLAB Simulink</i> para este proceso.		

Código	Nombre	Estimación
1.3.5	<b>Inclusión del código de Arduino en Simulink</b>	6 días
<b>Descripción</b>		
Se utilizará el software <i>MATLAB Simulink</i> para llamar a las funciones de los <i>drivers</i> de <i>Magabot</i> , escrito en lenguaje <i>C++</i> .		
Se redactará una guía de las acciones realizadas con <i>MATLAB Simulink</i> para este proceso.		

Código	Nombre	Estimación
1.3.6	<b>Profiling del código en Simulink</b>	3 días
<b>Descripción</b>		
Se realizarán pruebas de rendimiento del código generado por <i>Simulink</i> .		
Se comparará el rendimiento del código generado por <i>Simulink</i> con el código en <i>C++</i> de los <i>drivers</i> de <i>Magabot</i> .		

#### 5.2.2.4. Pruebas

Durante esta fase del proyecto, se llevarán a cabo las pruebas definidas en la fase de análisis, con el fin de verificar el funcionamiento del sistema.

Código	Nombre	Estimación
1.4.1	<b>Pruebas funcionales</b>	5 días
<b>Descripción</b>		
Se ejecutarán todas las pruebas necesarias para verificar el correcto funcionamiento del sistema.		

Código	Nombre	Estimación
1.4.2	<b>Pruebas de aceptación del sistema</b>	1 día
<b>Descripción</b>		
Se realizará una prueba exhaustiva del funcionamiento completo del sistema.		

### **5.2.2.5. Cierre**

Durante la última fase del proyecto, se realizará la escritura de la memoria final y se entregará la documentación.

<b>Código</b>	<b>Nombre</b>	<b>Estimación</b>
<b>1.5.1</b>	<b>Redacción de la memoria final</b>	40 días
<b>Descripción</b>		
Se elaborará una memoria final que contenga toda la documentación generada durante el desarrollo del proyecto, y que contenga un resumen, una introducción y conclusiones., etc.		

<b>Código</b>	<b>Nombre</b>	<b>Estimación</b>
<b>1.5.2</b>	<b>Entrega de la documentación (hito)</b>	0 días
<b>Descripción</b>		
Se presentará la memoria final a los tutores del proyecto.		
Se realizarán las correcciones señaladas.		
Se presentará la memoria en la Universidad Pablo de Olavide, con el visto bueno de los tutores.		

<b>Código</b>	<b>Nombre</b>	<b>Estimación</b>
<b>1.5.3</b>	<b>Preparación de la defensa del trabajo de fin de grado</b>	10 días
<b>Descripción</b>		
Se elaborará el texto de defensa.		
Se diseñará el material de apoyo.		

<b>Código</b>	<b>Nombre</b>	<b>Estimación</b>
<b>1.5.4</b>	<b>Defensa del trabajo de fin de grado (hito)</b>	0 días
<b>Descripción</b>		
Se defenderá la memoria ante un jurado formado por profesores de la Universidad Pablo de Olavide.		

### 5.2.3. Diagrama de Gantt

Figura 2: Diagrama de Gantt del proyecto

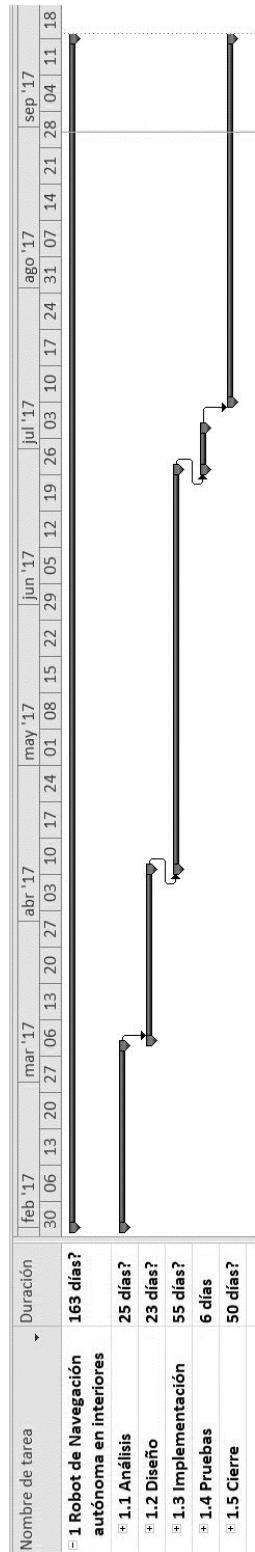


Figura 3: Diagrama de Gantt de la fase de análisis

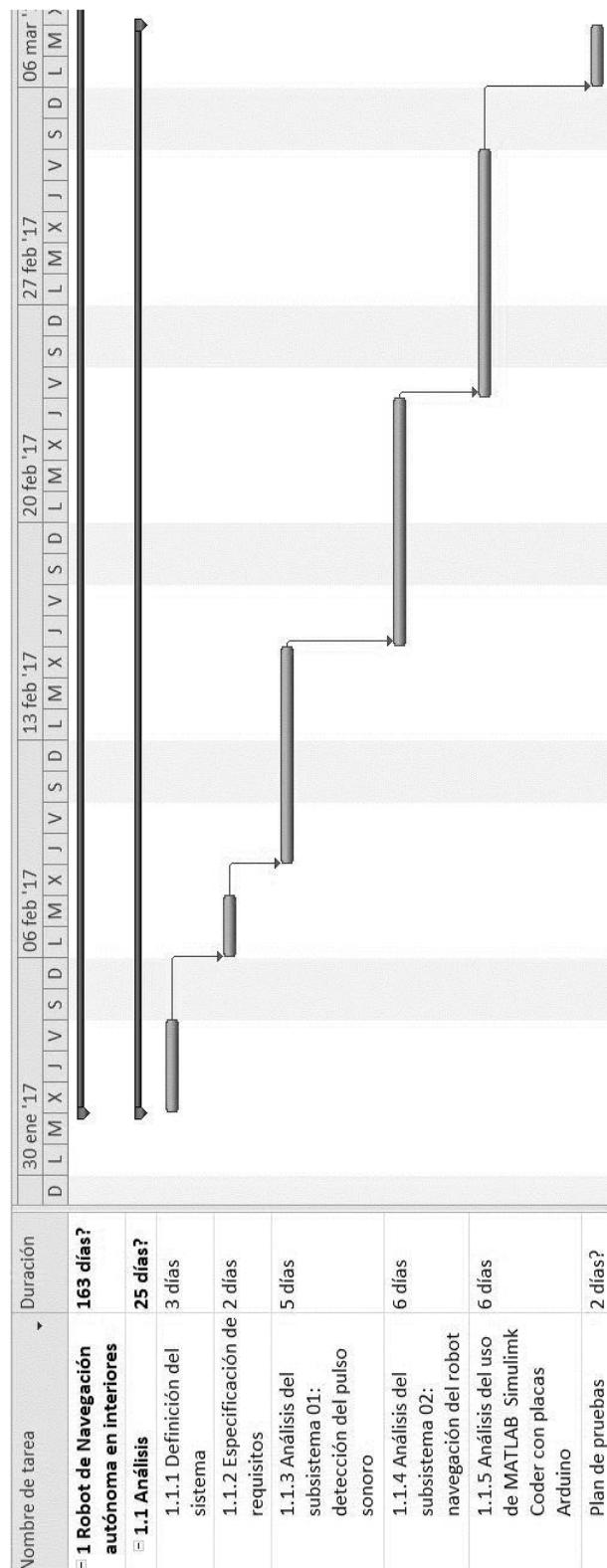


Figura 4: Diagrama de Gantt de la fase de diseño

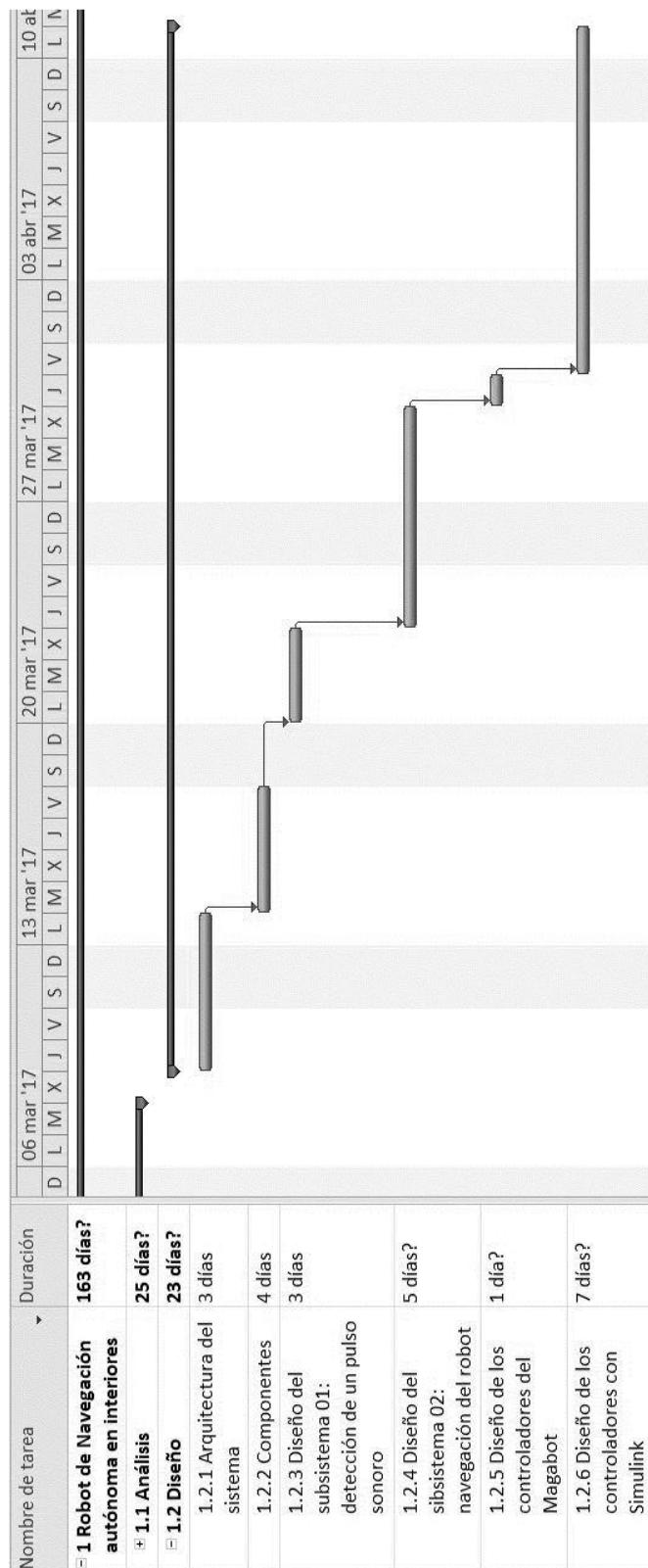


Figura 5: Diagrama de Gantt de la fase de implementación

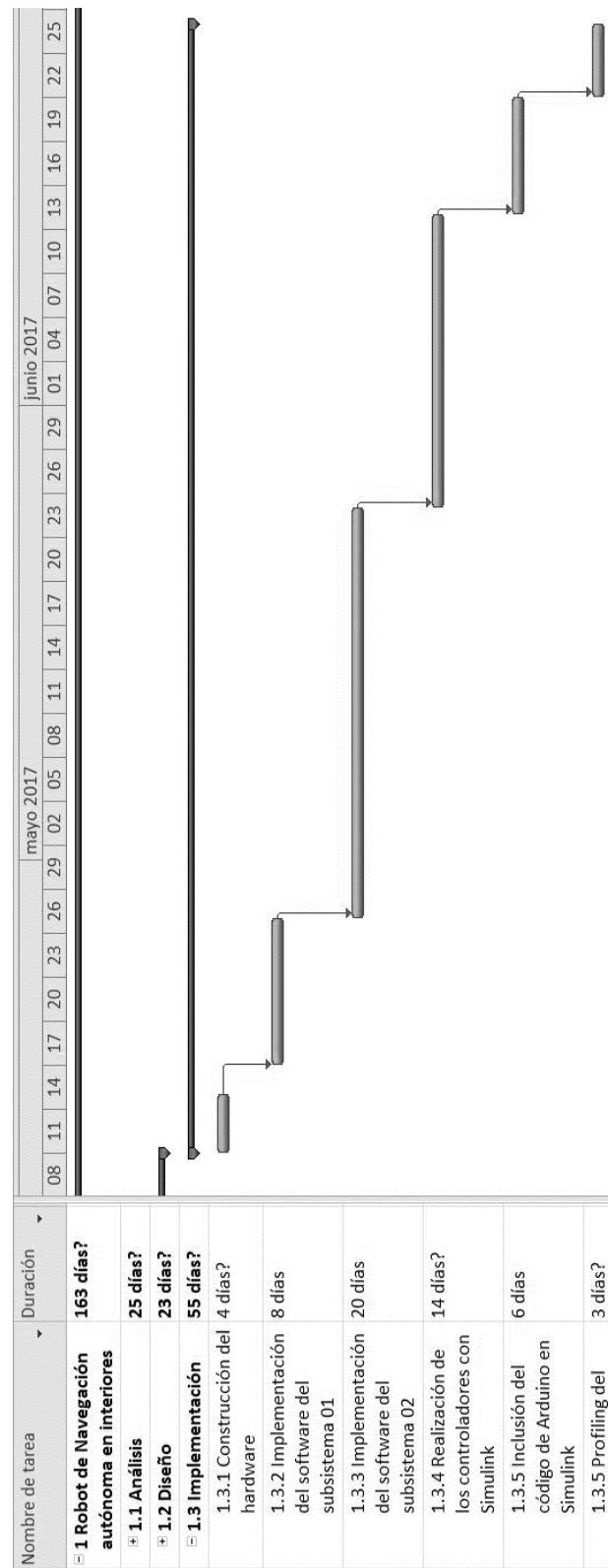


Figura 6: Diagrama de Gantt de la fase de pruebas

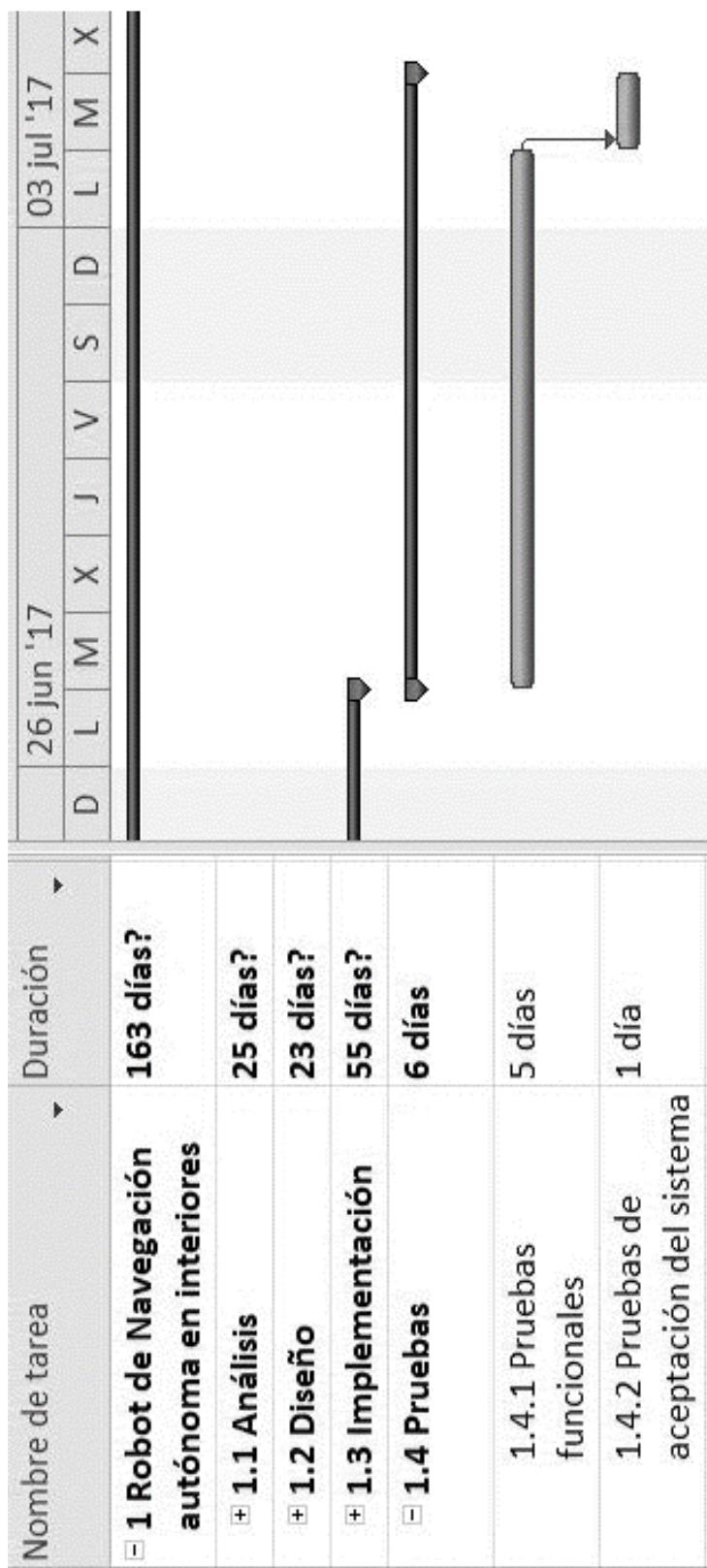
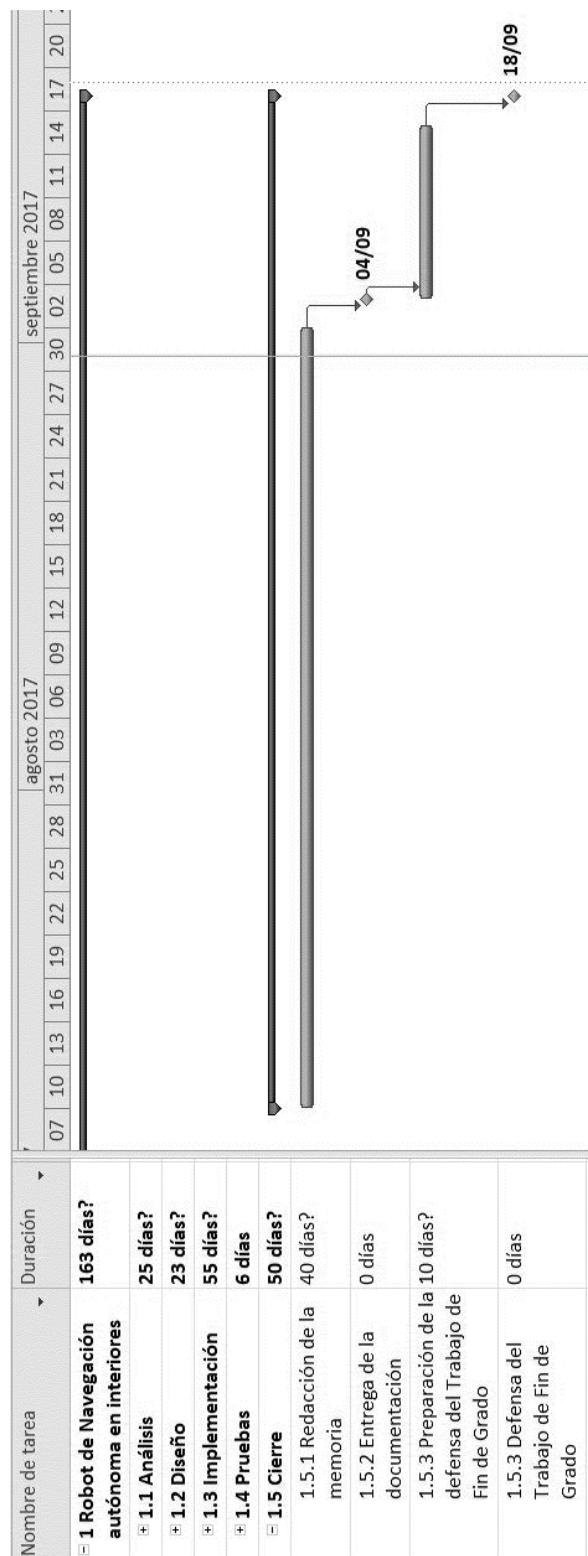


Figura 7: Diagrama de Gantt de la fase de cierre



#### 5.2.4. Estimación de coste

La estimación de coste del proyecto se detalla en la siguiente tabla.

Artículo	Cantidad	Precio por unidad	Coste
<i>Arduino Due</i>	1	37 €	37 €
<i>Arduino Mega 2560</i>	1	41,50 €	41,50 €
<i>Magabot</i>	1	400 €	400 €
Batería	1	18 €	18 €
Micrófono <i>KY-037</i>	3	2 €	6 €
Sensor de temperatura <i>TMP36GZ</i>	1	1 €	1 €
Cables Dupont	40	0,07 €	2,80 €
Placa <i>solderless breadboard</i>	1	10 €	10 €
<i>PC ACER ASPIRE E3-111-C5GL</i>	1	250 €	250 €
<b>Total</b>			<b>766,30 €</b>

## 6. EVALUACIÓN Y PLANIFICACIÓN DE RIESGOS

A continuación, se presenta una tabla en la que se muestran los riesgos para el proyecto.

ID	Descripción	Probabilidad	Severidad	Plan de contingencia	Prioridad
01	Ruptura del Magabot	Baja	Media	Si se rompiere un <i>Magabot</i> , se podría utilizar un segundo <i>Magabot</i> , el cual está disponible en el Laboratorio de Robótica de la Universidad Pablo de Olavide.	Alta
02	Ruptura de un componente electrónico	Baja	Baja	Se deberán sustituir las placas Arduino y los componentes electrónicos que sufrieran rupturas. Por esta razón, se utilizarán para el desarrollo del proyecto materiales de bajo coste, y fáciles de encontrar en el mercado.	Alta
03	Retraso en el proyecto	Media	Media	En el caso de retraso en el proyecto, se dedicarán más horas de las previstas en un inicio. Si esto no fuese posible, se reducirá el alcance del proyecto.	Media
04	Inaccesibilidad del laboratorio	Baja	Media	Si no fuese posible el acceso al Laboratorio de Robótica, se buscará otro lugar de trabajo en la Universidad Pablo de Olavide. Si esto no fuese posible, se solicitará autorización para trabajar con	Alta

				<i>Magabot</i> en nuestro domicilio.	
05	Pérdida de datos	Media	Media	Si se perdiese algún dato sensible del código o de la documentación relativa al trabajo, se accederá a las copias de seguridad, elaboradas periódicamente.	Alta

## 7. OTROS ASPECTOS DEL PROYECTO

### 7.1. UBICACIÓN DEL EQUIPO DE DESARROLLO

El proyecto se llevará a cabo en el Laboratorio de Robótica de la Universidad Pablo de Olavide, situado en el edificio 45. El Laboratorio cuenta con todo el equipamiento necesario para el correcto desarrollo de las actividades planificadas.

A Corzetto Conflan no solo se le asignará un escritorio sino que, además, se le otorgará acceso a la plataforma móvil *Magabot*, una *netbook* y una placa *Arduino Mega 2560*.

En el Laboratorio dispondrá, asimismo, de un espacio apropiado para la realización de las pruebas de sistema.