UNIVERSITAT POLITÈCNICA DE VALÈNCIA

Escuela Técnica Superior de Ingeniería Aeroespacial y Diseño Industrial

Implementación de una arquitectura funcional para robots móviles en Matlab

Trabajo Fin de Grado

Grado en Ingeniería Electrónica Industrial y Automática

AUTOR/A: Peñarrubia Zamora, Laura

Tutor/a: Zotovic Stanisic, Ranko

CURSO ACADÉMICO: 2023/2024

ÍNDICE DE DOCUMENTOS

1. Memoria
2. Planos
3. Pliego de condiciones
4. Presupuesto

1. Memoria

UNIVERSITAT POLITÈCNICA DE VALÈNCIA

Escuela Técnica Superior de Ingeniería Aeroespacial y Diseño Industrial

Memoria

Implementación de una arquitectura funcional para robots móviles en Matlab

Grado en Ingeniería Electrónica Industrial y Automática

AUTORA: Peñarrubia Zamora, Laura

TUTOR: Zotovic Stanisic, Ranko

CURSO ACADÉMICO: 2023/2024

Resumen

En las últimas décadas, la robótica ha evolucionado considerablemente, destacando la rama de los robots móviles por su capacidad para operar autónomamente en diversos entornos. Estos robots integran sensores para percibir su entorno y sistemas de control que gestionan sus movimientos, como el ajuste de trayectoria y velocidad. Su desarrollo combina disciplinas como la ingeniería electrónica, la informática y la inteligencia artificial. Este trabajo se enfoca en la creación de una arquitectura funcional de un robot móvil 2D en MATLAB, complementado con un sensor láser para reconocer el entorno. Se incluyen demostraciones de las simulaciones que ilustran los comportamientos que ejecuta el robot, así como la integración de un piloto automático y un planificador para lograr que el robot se desplace de forma autónoma por un entorno.

Abstract

In recent decades, robotics has evolved considerably, with mobile robots standing out for their ability to operate autonomously in diverse environments. These robots integrate sensors to perceive their surroundings and control systems that manage their movements, including trajectory adjustments and speed control. Their development combines disciplines such as electronics engineering, computer science, and artificial intelligence. This work focuses on creating a functional architecture of a 2D mobile robot in MATLAB, enhanced with a laser sensor for environment recognition. It includes demonstrations of simulations illustrating the robot's behaviours, along with the integration of an autopilot and planner to enable autonomous navigation in its environment.

Resum

En les últimes dècades, la robòtica ha evolucionat considerablement, destacant els robots mòbils per la seua capacitat per a operar autònomament en diversos entorns. Aquests robots integren sensors per a percebre el seu entorn i sistemes de control que gestionen els seus moviments, com l'ajust de trajectòria i velocitat. El seu desenvolupament combina disciplines com l'enginyeria electrònica, la informàtica i la intel·ligència artificial. Aquest treball s'enfoca en la creació d'una arquitectura funcional d'un robot mòbil 2D en MATLAB, complementat amb un sensor làser per a reconéixer l'entorn. S'inclouen demostracions de les simulacions que il·lustren els comportaments que executa el robot, així com la integració d'un pilot automàtic i un planificador per a aconseguir que el robot es desplace de manera autònoma per un entorn.

ÍNDICE

1. Introducción 10
2. Objetivo 12
3. Estado del arte 13
3.1. Robots móviles 14
3.2. Arquitectura de un robot 16
3.3. Sensor láser 19
4. Herramientas 20
4.1. MATLAB 20
5. Diseño e implementación de la arquitectura 21
5.1. Clase sensor láser 21
5.1.1. Propiedades 21
5.1.2. Cálculo de los puntos de intersección con el entorno 26
5.2. Clase robot 31
5.2.1. Propiedades 32
5.2.2. Funciones de movimiento 36
5.2.3. Comportamientos 43
5.2.4. Planificador 61
5.2.5. Piloto 65
6. Resultados 69
6.1. Simulación “Objetivo” 69
6.2. Simulación “Evitar Obstáculos” 73
6.3. Simulación “Seguir pared” 77
6.3.1. Pared derecha 80
6.3.2. Pared izquierda 81
6.4. Simulación “Seguir pasillo” 83
6.5. Simulación “Atravesar puerta” 87
6.6. Simulación “Aparcar” 91
6.7. Simulación “Piloto” 95
7. Conclusiones 106
8. Bibliografía 108
9. Anexo 112
9.1. ODS 112
9.2. Arquitectura basada en campos potenciales 113
9.3. Cálculo de los puntos de intersección 116

9.4. Cálculo de vectores 119
9.4.1. Vector de coordenadas finales 119
9.4.2. Vector de velocidad lineal 121
9.4.3. Vector de velocidad angular 122
9.4.4. Vector seguir recto 123
9.4.5. Vector objetivo 123
9.4.6. Vector evita obstáculos 124
9.4.7. Vector orientación pared 125
9.4.8. Vector orientación pasillo 126
9.4.9. Vector infinito 126
9.4.10. Vector aparcamiento 127
9.5. Manual de usuario 128
9.5.1. Modificaciones en *crear_sensor_láser* 128
9.5.2. Modificaciones en *crear_robot* 130
9.5.3. Modificaciones en *main* 133
9.5.4. Consejos, uso en *Command Window* y Workspace 135
9.6. Código 136
9.6.1. Clase robot 136
9.6.2. Clase sensor láser 156
9.6.3. Main 160
9.6.4. Algoritmo de Dijkstra y relacionados 161
9.6.5. Ficheros de dibujo y líneas 164
9.6.6. Ficheros de matrices y coordenadas 165
9.6.7. Entornos 166

ÍNDICE DE FIGURAS

Figura 1. Ejemplo de un AGV. 13
Figura 2. William Grey Walter con uno de sus robots tortuga.......... 14
Figura 3. Robot Shakey. 15
Figura 4. Ciclo SMPE.......... 15
Figura 5. Trayectoria de un robot en una arquitectura de campos potenciales.......... 17
Figura 6. Esquema de AuRA. 18
Figura 7. Esquema de una arquitectura híbrida de 3 niveles.......... 18
Figura 8. Esquema de un sensor láser.......... 19
Figura 9. Logo de MATLAB.......... 20
Figura 10. Propiedades del sensor láser. 21
Figura 11. Dibujo del sensor láser en Matlab. 23
Figura 12. Diagrama de bloques de algunas propiedades de la clase *sensor_laser.* 25
Figura 13. Representación del escaneo del láser con el robot. 26
Figura 14. Parámetros para calcular los puntos de intersección con el entorno. 27
Figura 15. Punto de intersección con el entorno. 28
Figura 16. Diagrama de flujo de la función *Intersección.* 29
Figura 17. Resultado de la función *calcular_interseccion_multiple.* 30
Figura 18. Diagrama de flujo de la función *calcular_interseccion_multiple.* 31
Figura 19. Dibujo del robot con el sensor láser en MATLAB. 32
Figura 20. Representación de las coordenadas cartesianas del robot en el entorno. 33
Figura 21. Matriz homogénea del robot. 34
Figura 22. Diagrama de bloques de algunas propiedades de la clase *robot.* 36
Figura 23. Representación del antes y después de ejecutar *mover_robot.* 37
Figura 24. Diagrama de flujo de *mover_robot.* 38
Figura 25. Resultado función *simulación_giro_robot.* 39
Figura 26. Diagrama de flujo de *simulacion_giro_robot.* 40
Figura 27. Representación de algunos pasos de la función *girar_robot.* 41
Figura 28. Diagrama de flujo de *girar_robot.* 42
Figura 29. Diagrama de flujo de la función *objetivo.* 44
Figura 30. Diagrama de flujo de la función *Vect_objetivo.* 45
Figura 31. Ángulos seleccionados para la evitación de obstáculos 46
Figura 32. Diagrama de flujo de la función *evita_obstaculos.* 47

Figura 33. Ángulos para la pared derecha. 49
Figura 34. Ángulos para la pared izquierda........ 49
Figura 35. Diagrama de flujo de la función *orientacion_pared.*........ 50
Figura 36. Diagrama de flujo de la función *seguir_pared.* 52
Figura 37. Representación de los ángulos utilizados en la función *seguir_pasillo.*........ 53
Figura 38. Ejemplo de cuando uno de los sensores laterales de infinito. 54
Figura 39. Diagrama de flujo de la función *seguir_pasillo*. 56
Figura 40. Ángulos para la función *atravesar_puertas.* 57
Figura 41. Diagrama de flujo de la función *infinito.*........ 58
Figura 42. Diagrama de flujo de la función *atravesar_puertas.* 59
Figura 43. Diagrama de flujo de la función *vector_aparcamiento.*........ 60
Figura 44. Diagrama de flujo de la función *aparcar.*........ 61
Figura 45. Ejemplo de un entorno para el planificador. 62
Figura 46. Diagrama de flujo del fichero *Dijkstra.*........ 64
Figura 47. Diagrama de flujo de la función *piloto.* 68
Figura 48. Situación inicial en *simulacion_objetivo.* 70
Figura 49. Diagrama de flujo de *simulacion_objetivo.*........ 71
Figura 50. Resultado de la simulación del comportamiento *objetivo.* 72
Figura 51. Resultado de otra simulación del comportamiento *objetivo.*........ 73
Figura 52. Entorno de la simulación del comportamiento de evitación de obstáculos 74
Figura 53. Diagrama de flujo de la simulación de evitar obstáculos. 75
Figura 54. Resultados de la simulación del comportamiento *evitar_obstaculos.*........ 76
Figura 55. Resultado de otra simulación del comportamiento *evita_obstáculos*........ 77
Figura 56. Entorno de la simulación del comportamiento seguir pared. 78
Figura 57. Diagrama de flujo de la simulación de seguir la pared. 79
Figura 58. Resultado de la simulación *seguir_pared* con la pared derecha........ 80
Figura 59. Resultado de otra simulación del comportamiento de seguir la pared derecha 81
Figura 60. Resultado de la simulación *seguir_pared* con la pared izquierda. 82
Figura 61. Resultado de otra simulación del comportamiento *seguir_pared* con la pared izquierda. 83
Figura 62. Entorno de la simulación del comportamiento seguir pasillo. 84
Figura 63. Diagrama de flujo de la función *simulacion_seguir_pasillo.*........ 85
Figura 64. Resultado de la simulación *seguir_pasillo.*........ 86

Figura 65. Resultado de la simulación del comportamiento seguir pasillo con otras coordenadas. 87
Figura 66. Entorno de la simulación *atravesar_puertas.* 88
Figura 67. Diagrama de flujo de la simulación de *atravesar_puertas.* 89
Figura 68. Resultados de la simulación *atravesar_puertas.* 90
Figura 69. Resultado de la simulación del comportamiento *atravesar puertas* con otras coordenadas. 91
Figura 70. Entorno de la simulación aparcar. 92
Figura 71. Diagrama de flujo de la función *simulacion_aparcar.* 93
Figura 72. Resultado de la simulación de la función *aparcar.* 94
Figura 73. Resultado de la simulación del comportamiento *aparcar* con otras coordenadas. 95
Figura 74. Diagrama de flujo del código de simulación *piloto*. 96
Figura 75. Entorno del piloto. 97
Figura 76. Entorno de la simulación piloto con los nodos y el robot. 98
Figura 77. Resultado de la simulación de la función *piloto.* 102
Figura 78. Simulación del piloto con otro comportamiento (seguir pasillo). 103
Figura 79. Simulación del piloto con otro comportamiento (seguir la pared derecha). 104
Figura 80. Resultado de la simulación piloto con otro comportamiento (aparcar) 105
Figura 81. Campos potenciales primitivos. 113
Figura 82. Campos potenciales en un entorno. 114
Figura 83. Camino seguido por el robot en un campo potencial. 115
Figura 84. Campos potenciales de algunos comportamientos del robot. 116
Figura 85. Parámetros ajustables del sensor láser. 129
Figura 86. Comentario para dibujar el haz del sensor. 129
Figura 87. Comentario para dibujar los puntos de intersección. 130
Figura 88. Parámetros que se pueden modificar del robot. 131
Figura 89. Ejemplo de las constantes de simulación. 132
Figura 90. Ejemplo de la definición de las coordenadas del robot en un comportamiento. 132
Figura 91. Ejemplo de la definición de los ejes y entorno en un comportamiento. 132
Figura 92. Main para probar la simulación. 134
Figura 93. Ejemplo de uso del Command Window para acceder a las propiedades del robot. 135

Figura 94. Ejemplo de uso del Command Window para acceder a las propiedades del sensor láser. 135
Figura 95. Ejemplo del Workspace. 136

ÍNDICE DE TABLAS

Tabla 1. Ejemplo matriz de puntos de intersección. 24
Tabla 2. Ejemplo de las coordenadas del robot. 33
Tabla 3. Matriz de costes. 63
Tabla 4. Ejemplo de la matriz de tipos de comportamientos. 66
Tabla 5. Tipos de comportamientos y sus números en *m_tipos.* 67
Tabla 6. Posiciones de los nodos en la simulación del piloto. 97
Tabla 7. Matriz de costes de la simulación de la función piloto. 99
Tabla 8. Matriz de tipos de comportamiento de la simulación de la función piloto. 100
Tabla 9. Objetivos de Desarrollo Sostenibles 112

ÍNDICE DE ECUACIONES

Ecuación 1. Comprobación de que el láser apunta a la pared. 117
Ecuación 2. Ecuación paramétrica del segmento de la pared. 117
Ecuación 3. Ecuación paramétrica del rayo del sensor láser. 117
Ecuación 4. Sistema de ecuaciones paramétricas para encontrar un punto de intersección. 117
Ecuación 5. Ecuación paramétrica simplificada. 118
Ecuación 6. Ecuación para encontrar un punto de intersección con el láser. 118
Ecuación 7. Ecuación del punto de intersección. 118
Ecuación 8. Cálculo de las coordenadas de desplazamiento del robot. 119
Ecuación 9. Cálculo de las coordenadas de desplazamiento del robot simplificado....... 121
Ecuación 10. Cálculo de la velocidad lineal. 121
Ecuación 11. Cálculo del ángulo para la velocidad angular.. 122
Ecuación 12. Cálculo de la velocidad angular. 122
Ecuación 13. Vector seguir recto. 123
Ecuación 14. Cálculo del vector objetivo.. 123
Ecuación 15. Cálculo del vector de evitación de obstáculos. 124
Ecuación 16. Cálculo del vector de orientación a la pared. 125
Ecuación 17. Cálculo de la diferencia entre las distancias. 125
Ecuación 18. Cálculo del vector infinito.. 126

1. Introducción

En las últimas décadas, el campo de la robótica ha experimentado avances significativos, transformándose en una rama clave dentro de la ingeniería y la tecnología debido al aumento de densidad de robots a nivel mundial.

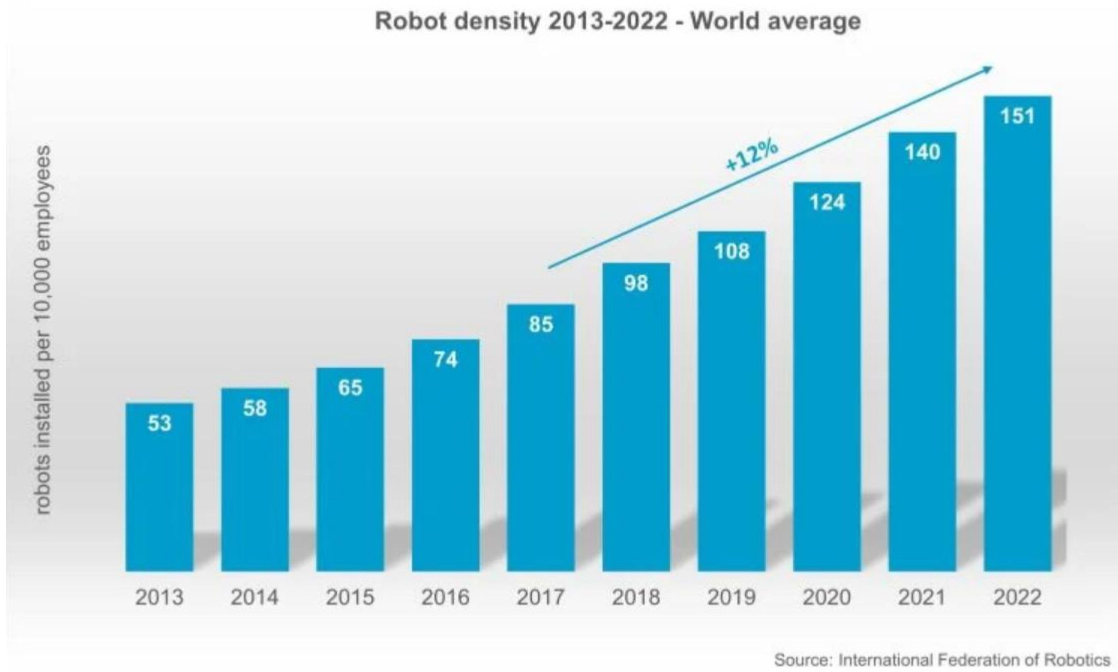


Figura 1. Gráfica del aumento de instalaciones de robots industriales a nivel mundial.
Fuente. [1].

Dentro de este campo, los robots móviles han emergido como una categoría fundamental en la base de la robótica, destacando por su capacidad de desplazarse de manera autónoma en diversos entornos y cumplir tareas complejas que antes eran exclusivas de los seres humanos. Su desarrollo involucra una combinación de disciplinas que incluyen la ingeniería mecánica, la informática y la inteligencia artificial.

El funcionamiento de un robot móvil depende tanto de la integración de los sensores que lo constituyen como de los sistemas de control que gestionan y coordinan los movimientos del robot. Los sensores son elementos clave para que el robot pueda percibir dónde se encuentra, ya que recogen datos sobre obstáculos y otras características del entorno. Estos datos son procesados por algoritmos que interpretan la información de los sensores y deciden las acciones que el robot debe tomar.

Los sistemas de control son responsables de convertir en movimientos las decisiones tomadas por los algoritmos. Estos sistemas deben ser capaces de gestionar los actuadores del robot, como por ejemplo las ruedas, para que éste se desplace de manera eficiente y segura, ajustando su trayectoria y velocidad según sea necesario.

Este trabajo se centra en la importancia de la integración efectiva de los componentes que permiten el funcionamiento autónomo de un robot móvil. Se explorarán los diferentes tipos de robots, así como las arquitecturas para dotarlos de autonomía con distintos comportamientos. A través de simulaciones en 2D en una variedad de escenarios, se demostrará cómo la combinación de sensores y una arquitectura robótica resulta en un robot completamente funcional y autónomo.

Todo este proyecto se ha realizado con programación orientada a objetos, lo que permite que la arquitectura se adapte a distintos entornos y utilice el comportamiento adecuado para cada situación. El enfoque de este tipo de programación facilita el desarrollo futuro del proyecto, haciendo que la implementación de dimensiones adicionales, como el tercer eje cartesiano, u otros tipos de comportamientos, sea más sencilla.

Finalmente, este trabajo parte de la práctica 5 del máster de Informática Industrial y Automática de la Universidad Politécnica de Valencia, impartida por el profesor Ranko Zotovic Stasinic [24].

2. Objetivo

Este proyecto tiene como objetivo desarrollar una arquitectura de un robot móvil en MATLAB. Para ello, se simulará un robot móvil en 2D que se desplazará siguiendo unos comportamientos específicos en diversos entornos. Este robot incluye un sensor láser que escaneará los escenarios para identificar y reconocer obstáculos o paredes.

Para conseguir el objetivo principal, se plantean las siguientes metas:

- Realizar un estudio sobre los tipos de arquitecturas más comunes dentro del sector de la robótica. Con esto se consigue información sobre el funcionamiento de los robots autónomos.

- Representar el robot junto con el sensor láser en MATLAB para poder visualizar su posición y trayectoria en todo momento.

- Implementar el escaneo del entorno con el sensor láser. De esta forma, el robot podrá identificar los obstáculos y/o paredes que se encuentren en un entorno.

- Simular de manera individual los comportamientos de la arquitectura en entornos específicos para cada uno de ellos. Esto permitirá realizar las comprobaciones y ajustes necesarios para asegurarse de que cada comportamiento está correctamente implementado.

- Simular el comportamiento del planificador con el algoritmo de Dijkstra [16], que determina la ruta más corta desde un punto hasta el destino final. Con esto, el robot será capaz de moverse de un punto a otro eficientemente, ya que siempre seguirá el camino más corto hacia su objetivo.

- Simular el comportamiento del piloto integrado en el robot, que elige cómo, cuándo y qué comportamiento tiene que seguir el robot. Al estar implementado dentro de la arquitectura del robot, esto hace que sea completamente autónomo.

3. Estado del arte

La robótica y el desarrollo de robots ha crecido y evolucionado significativamente desde los primeros intentos de robots en la antigüedad hasta la tecnología actual. Aunque la idea de un robot como concepto se remonta a tiempos antiguos, el término robot se acuñó en el 1920, en una obra de teatro donde se representaba a los robots (llamados *robota* que significa trabajo forzado en checo) como seres fabricados para que realizasen cualquier tipo de trabajo desagradable. La noción moderna de robot comenzó a desarrollarse con la Revolución Industrial, cuando se avanzó en la mecanización y automatización de tareas. Estos avances sentaron las bases para los sistemas robóticos que hoy en día se utilizan en múltiples industrias y aplicaciones.

La definición de robot aparece en la RAE como "*Máquina o ingenio electrónico programable que es capaz de manipular objetos y realizar diversas operaciones* [35]*.*" En la actualidad, existen diversos tipos de robots y se pueden clasificar según su aplicación, entorno, movilidad etc. En el caso de este estudio, el robot es un AGV (vehículo de guiado automático), que es un tipo de robot móvil que se desplaza siguiendo rutas predefinidas y utilizando diversos sistemas de guiado, como por ejemplo un sensor láser. Se utilizan sobre todo en industrias para transportar mercancía.

Figura 2. Ejemplo de un AGV.

Fuente. [2].

En la siguiente sección, se detallarán las características y funciones de los robots móviles.

3.1. Robots móviles

Un robot móvil es un tipo de robot capaz de desplazarse por su entorno de manera autónoma o semi-autónoma. Estos robots están equipados con sensores y sistemas de navegación que les permiten percibir su entorno, tomar decisiones y ejecutar movimientos para alcanzar objetivos específicos. Los robots móviles pueden operar en diversos entornos, como interiores, exteriores, industriales o domésticos, y se utilizan en aplicaciones como la logística, la exploración, el servicio y la manufactura. Ejemplos comunes incluyen robots de limpieza, vehículos autónomos y drones.

Los primeros robots móviles con comportamientos complejos fueron creados en el 1948 por William Grey Walter, un neurofisiólogo y pionero de la robótica que se propuso demostrar que, con un pequeño número de células cerebrales bien conectadas, se podían desarrollar comportamientos muy complejos. De esta forma, pretendía corroborar que el secreto del funcionamiento del cerebro radicaba en la forma en la que están conectadas las células cerebrales.

Así pues, creó a dos robots llamados Elmer y Elsie hechos a base de electroválvulas, tenían 5 tipos de comportamiento: búsqueda de luz, atracción hacia la luz débil, repulsión de la luz fuerte, girar y empujar y recargar la batería. Estos robots, conocidos como *robots tortuga* por su caparazón y movimientos lentos, se movían sobre tres ruedas, tenían una célula fotoeléctrica móvil que hacía la función de vista y actuaba sobre un motor que guiaba la dirección del movimiento.

Figura 3. William Grey Walter con uno de sus robots tortuga.

Fuente. [6].

El primer robot móvil capaz de razonar sobre sus propias acciones fue creado en la década de los 60 por la universidad de Standford. Este robot, llamado Shakey, fue el primero en incluir inteligencia artificial. Podía navegar autónomamente desde un sitio a otro, mapear entornos y calcular el camino más corto de un punto a otro mediante un algoritmo de búsqueda llamado A*. Por otra parte, el robot Shakey fue el precursor del paradigma SMPE(Sense-Model-Plan-Execute), un enfoque en el que el robot primero recoge datos del entorno, luego crea un modelo del mismo, planifica las acciones a seguir y finalmente ejecuta dichas acciones.

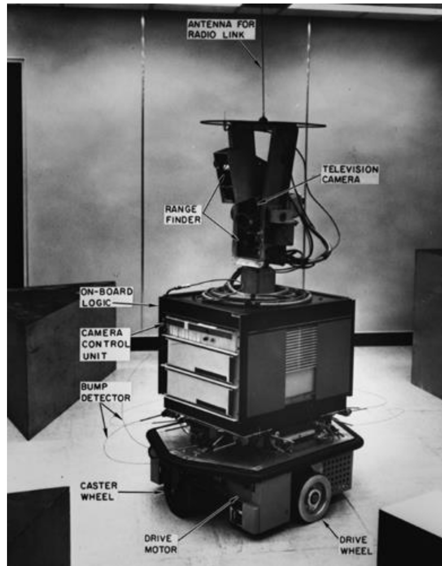


Figura 4. Robot Shakey.

Fuente. [7].

Shakey supuso un antes y un después en la historia de la robótica. Gracias a la creación del paradigma SMPE, se comenzaron a diseñar arquitecturas que organizasen la programación de los robots.

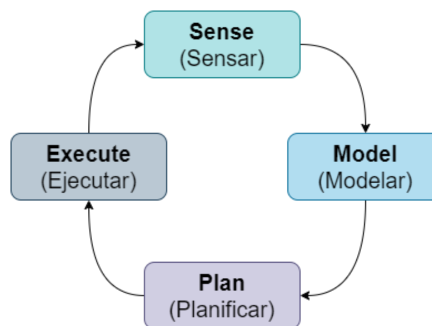


Figura 5. Ciclo SMPE.

Fuente. Propia.

En el siguiente apartado, Apartado 3.2, se explicarán las arquitecturas de los robots y sus tipos.

3.2. Arquitectura de un robot

La arquitectura de un robot es la estructura y organización del sistema de software y hardware que permite al robot funcionar. Esta arquitectura define cómo se integran y comunican los diversos componentes del robot, como los sensores, actuadores, procesadores y módulos de control, para lograr un comportamiento autónomo y coordinado.

Hay distintos tipos de arquitecturas:

- **Las arquitecturas jerárquicas:** están basadas en el paradigma S-M-P-E, y cuentan con varios niveles. Los altos se encargan de enviar comandos a los bajos y los bajos envían información a los altos.

- **Las arquitecturas reactivas:** están basadas en comportamientos. Funcionan mediante estímulo/respuesta. Para ello se empareja el sensor que mide el estímulo con el actuador que ejecuta la respuesta. Hay 3 tipos de arquitectura reactiva: la arquitectura de Braitenberg, la arquitectura de subsunción y la arquitectura basada en campos potenciales.

  La arquitectura basada en campos potenciales se usa parcialmente en este estudio. Se basa en la idea de que el robot se mueve dentro de un campo de fuerzas virtuales, donde cada punto en el espacio tiene un valor de potencial que guía el movimiento del robot. De esta forma, cada comportamiento devuelve un vector y el comportamiento resultante, que es el que ejecuta el robot, es la suma ponderada de varios vectores. Los comportamientos de este estudio vienen definidos por esos mismos vectores resultantes. En la Figura 6 aparece una representación del movimiento de un robot en un entorno de campos potenciales, donde:

  - **R** es el robot
  - **O** es un obstáculo
  - **G** es el objetivo hacia el que tiene que ir el robot
  - **t** es la trayectoria del robot
  - **las flechas** representan las fuerzas de atracción o repulsión del robot en el entorno.

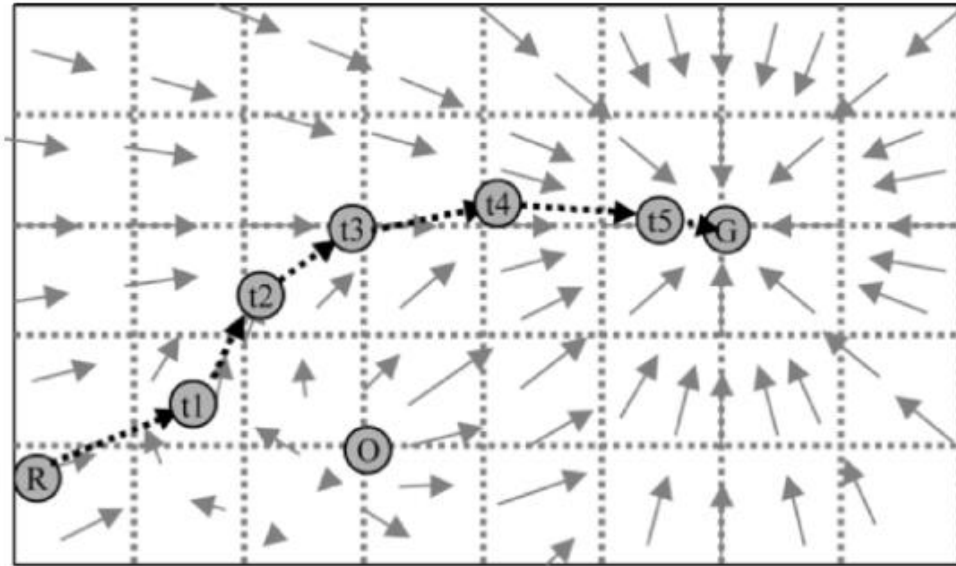


Figura 6. Trayectoria de un robot en una arquitectura de campos potenciales.

Fuente.[14].

La arquitectura de campos potenciales está explicada en detalle en el Anexo 9.2.

- **Las arquitecturas híbridas:** combinan las arquitecturas reactivas y las híbridas. Estas arquitecturas incluyen varios niveles, con el nivel más bajo siendo reactivo y basado en comportamientos simples e inmediatos. Dentro de estas arquitecturas, hay 3 que modelos: AuRA, DAMN y SAPHIRA. Para este proyecto es necesario conocer la arquitectura AuRa (Autonomous Robot Architecture), que combina elementos jerárquicos y reactivos, utilizando un enfoque orientado a objetos. La parte planificada de AuRA tiene tres niveles, similares a la arquitectura jerárquica. La parte reactiva incluye esquemas perceptuales y motores, cuya salida se obtiene como una suma de vectores. En la Figura 7 se muestra un esquema de esta arquitectura.

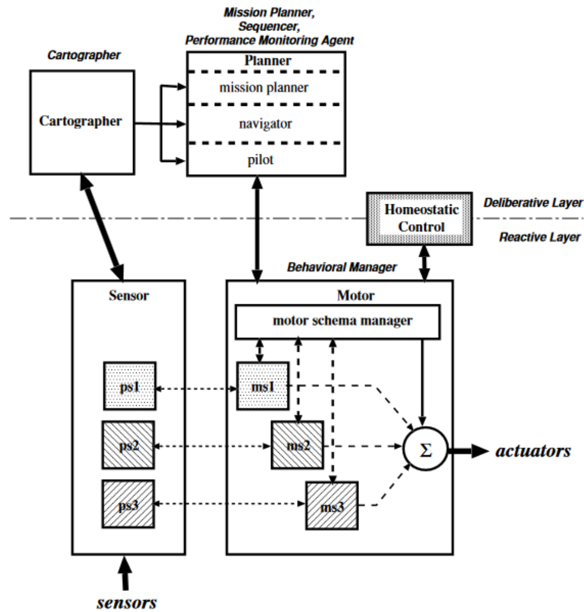


Figura 7. Esquema de AuRA.

Fuente. [13].

Para este estudio se utilizará una arquitectura híbrida. En este caso, la arquitectura es un prototipo de tres niveles, basado en AuRA, donde la primera capa es el planificador (determina la ruta a seguir), la segunda capa es el piloto (decide qué ejecutar) y la tercera corresponde a los comportamientos del robot, basados en sumas de vectores (movimiento del robot). En la Figura 8 se puede ver un esquema con las diferentes capas de esta arquitectura.

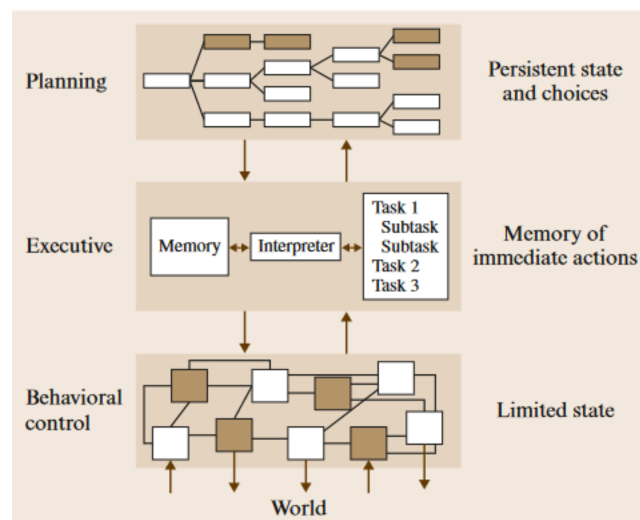


Figura 8. Esquema de una arquitectura híbrida de 3 niveles.

Fuente. [20].

3.3. Sensor láser

Un sensor láser es un dispositivo electrónico que emite un haz de luz para detectar la presencia, ausencia o distancia de un objeto. Funciona detectando el momento en que el haz de luz emitido se refleja en un objeto y regresa al receptor del sensor, permitiendo calcular la distancia o detectar la presencia del objeto.

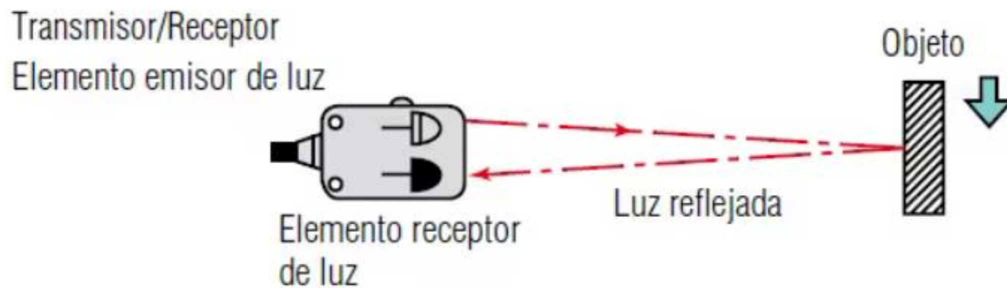


Figura 9. Esquema de un sensor láser.

Fuente. [17].

Los sensores láser tienen dos formas de medir la distancia hacia un objeto. Estos sistemas son:

- **Sistema de triangulación**
  La distancia se calcula basándose en la geometría del triángulo formado por el emisor, el objeto y el receptor.

- **Sistema de medición de tiempo**
  Mide el tiempo que tarda el haz de luz láser en viajar desde el sensor hasta el objeto y regresar al receptor.

En este proyecto no se utiliza ninguno de esos sistemas de medición, ya que se simula un sensor láser, específicamente el modelo UST-20LX (UUST004) [22] de Hokuyo Automatic. Para medir las distancias, se emplea geometría básica: se calcula la distancia directa entre dos puntos, en este caso, entre los objetos del entorno donde se encuentra el robot y el propio robot. El cálculo de la distancia está detallado en el Anexo 9.3.

4. Herramientas

En este apartado se introduce MATLAB, software desarrollado por MathWorks. Este programa se ha utilizado para la programación y simulación de esta arquitectura.

4.1. MATLAB

Según su propia página web: "*MATLAB es una plataforma de programación y cálculo numérico utilizada por millones de ingenieros y científicos para analizar datos, desarrollar algoritmos y crear modelos*" [25]. La versión de MATLAB utilizada para este trabajo es del año 2023.

El código de esta arquitectura se ha desarrollado enteramente en ficheros de MATLAB. Para ello, se han creado dos clases: la clase *crear_sensor_laser* (Anexo 9.4.2.) y la clase *crear_robot* (Anexo 9.4.1.). Al definir una clase, se crea un objeto que puede tener propiedades (datos) y métodos (funciones). En este caso, los objetos son el sensor láser y el robot. Las clases permiten organizar el código de manera que sea más fácil de manejar y reutilizar. Además, se utilizan estructuras de control comunes, como if, if-else y switch, para ejecutar diferentes acciones según las condiciones específicas que se encuentren en el programa.

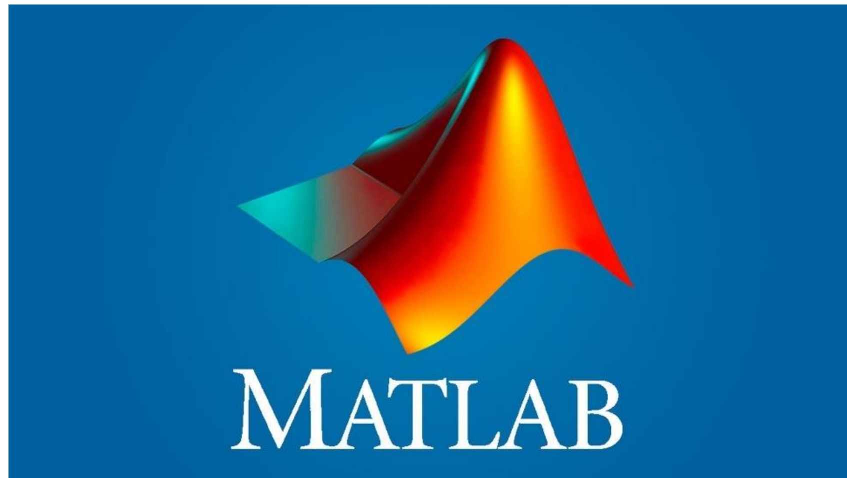


Figura 10. Logo de MATLAB.
Fuente. [18].

5. Diseño e implementación de la arquitectura

5.1. Clase sensor láser

En esta sección, se describe la clase *crear_sensor_laser* basada en el sensor láser UST-20LX , nombrada anteriormente en el apartado 3.3. Aunque la simulación del láser se podría haber hecho con la ToolBox Lidar de MATLAB, se optó por no usar librerías externas para mantener la simplicidad del código.

El objetivo principal de esta clase es escanear el entorno y devolver los puntos de intersección con el mismo. En total, esta clase cuenta con 10 funciones. Sin embargo, solo se desarrollarán las propiedades asociadas al láser y las funciones que calculan los puntos de intersección con el entorno. Para más información, el resto de las funciones del sensor se encuentran referenciadas en el Anexo 9.6.2.

5.1.1. Propiedades

Siguiendo la hoja de datos del sensor UST-20LX, los parámetros para la implementación del sensor son los siguientes:

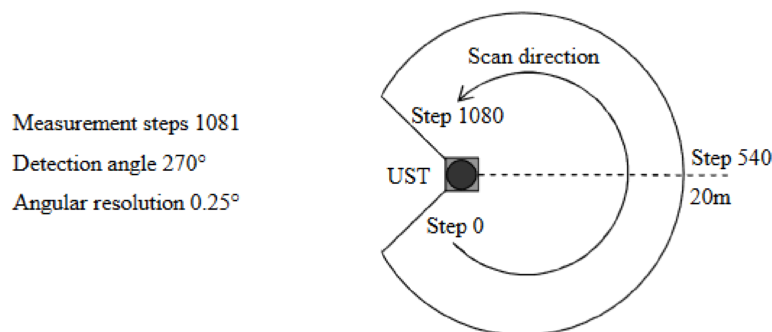


Figura 11. Propiedades del sensor láser.

Fuente. [22].

- **Ángulo inicial**

  Ángulo por el cual el sensor empieza a escanear, en este caso 225º del círculo unitario. Este valor se inicializa en 0º en el constructor de la clase por simplificaciones de código, ya que originalmente el sensor escanea de izquierda a derecha. Luego en la función que calcula los puntos de intersección, el ángulo de escaneo se modifica

para sumarle los 225º. En el código aparece definido como *angulo_inicial*.

- **Ángulo final**

  Ángulo en el que el sensor termina de escanear, en este caso, 135º. Al igual que en el caso anterior, se inicializa en 270º para que el rango de escaneo total sea de 0 a 270º, como en la hoja de datos. En el código aparece definido como *angulo_final*.

- **Resolución angular**

  Paso entre grado y grado del sensor láser. Siguiendo la hoja de datos, el valor de la resolución angular de 0.25º. Por ejemplo, si el sensor mide en el ángulo 90º, el ángulo de lectura anterior es de 89.75º y el posterior es de 90.25º. En el código aparece definido como *resolucion*.

- **Valor máximo de lectura**

  Valor máximo de lectura del sensor, en este caso 20 metros para todos sus ángulos. A partir de 20 metros, el sensor devuelve el punto de intersección como Infinito. En el código aparece definido como *valor_maximo*.

- **Valor mínimo de lectura**

  Valor mínimo de lectura del sensor, en este caso 0.06 metros para todos sus ángulos. Aunque está definido en el código como *valor_minimo*, no se utiliza en esta arquitectura. Por tanto, se asume que el robot no tiene valor mínimo de detección.

Por otra parte, para dibujar correctamente en el entorno el sensor y guardar correctamente los puntos de intersección con el entorno, se han creado las siguientes propiedades:

- **Dibujo**

  Guarda en una matriz las líneas que representan al sensor láser en el entorno. La matriz tiene una dimensión de 1x4 en el que cada columna guarda la representación de una de las líneas del sensor láser. Esta matriz se utiliza en funciones como *dibuagv*, referenciada en el Anexo 9.6.5., para poder dibujar el objeto en otras coordenadas.

  El dibujo del sensor está representado de forma que la parte roja simboliza la parte delantera y las líneas verdes el resto del sensor.

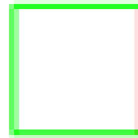


Figura 12. Dibujo del sensor láser en Matlab.
Fuente. Propia.

- **Posición relativa del sensor respecto al robot**

  Este parámetro sitúa al sensor en el centro del robot, y por simplificaciones de cálculo, se mantendrá en esa misma posición, que es [0,0,0]. Aunque esté en el centro, se considera arbitraria la distancia entre el sensor y el robot. En el código aparece definido como *posición_sensor_rel*.

- **Posición absoluta del sensor en el entorno**

  Este parámetro informa sobre la posición del sensor en el entorno. En el código aparece definido como *posición_sensor_abs.* Se calcula mediante la función *calcular_posicion_absoluta*.

- **Matriz del sensor láser**

  Almacena en una matriz de 1x4 la posición absoluta del sensor láser en el entorno junto con el valor máximo de detección (20 m).

Esta matriz sirve para calcular los puntos de intersección en el entorno, ya que para ello es necesario conocer la ubicación en el entorno tanto del robot, como del láser y su valor máximo de detección. En el código aparece definido como *matriz_sensor_laser.*

- **Dirección x, dirección y**

  Estas variables indican hacia donde está apuntando el láser y por tanto el robot. Son necesarias para saber si el robot está apuntando hacia una pared u obstáculo. En el código aparece definido como *direccion_x, direccion_y.*

- **Puntos de intersección**

  Es una matriz generalmente de 1081x4, donde la primera columna representa el ángulo de medición, la segunda y tercera columnas son las coordenadas x e y del punto detectado en la pared, y la cuarta columna es la distancia hasta ese punto. En la Tabla 1 se muestra un ejemplo de cómo es la matriz que guarda los puntos de intersección.

Ángulo	Coordenada X	Coordenada Y	Distancia
334	5	-2.292	5.22
334.250	5	-2.267	5.21
334.50	5	-2.241	5.20

Tabla 1. Ejemplo matriz de puntos de intersección.

Fuente. Propia.

En algunas funciones, la matriz se reduce a un número especifico de filas porque se usan solamente unos ángulos en específico para tomar medidas. En el código aparece definido como *puntos_interseccion.*

- **Robot**

  Es el objeto creado por la clase robot. Este objeto se le pasa como argumento a varias funciones del láser para poder acceder a las

propiedades del robot, como por ejemplo su posición en el entorno o las dimensiones del mismo definido por las líneas.

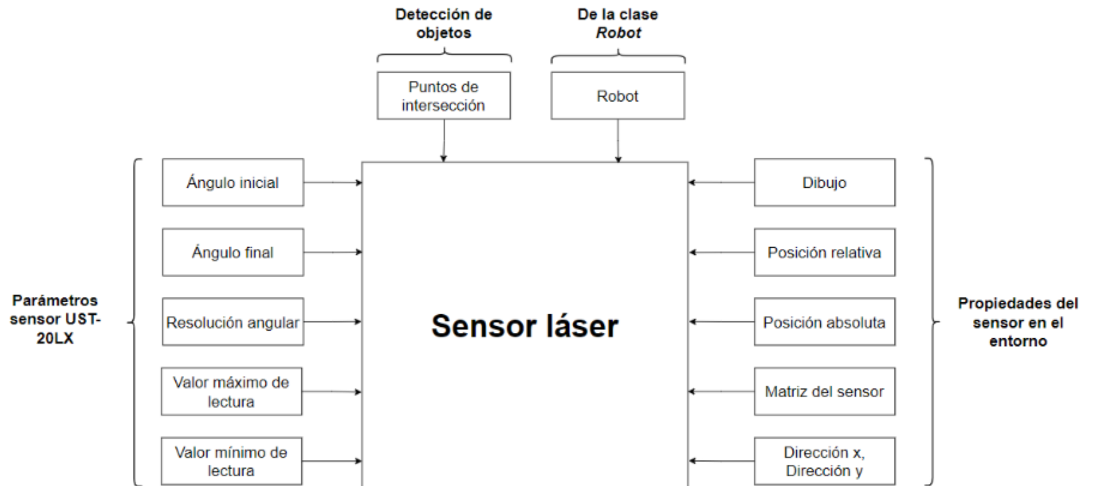


Figura 13. Diagrama de bloques de algunas propiedades de la clase *sensor_laser.*
Fuente. Propia.

Estas propiedades son algunas más importantes de la clase *sensor_laser*. El resto de las propiedades de la clase están detalladas en el Anexo 9.6.2.

Teniendo en cuenta todas las propiedades anteriores y las del robot, que se explican en el apartado 5.1.1. , en la Figura 14 se muestra la representación del rango, la dirección y algunos ángulos de escaneo de esta arquitectura.

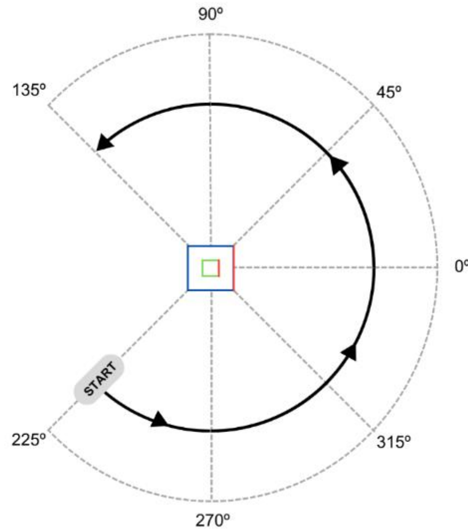


Figura 14. Representación del escaneo del láser con el robot.

Fuente. Propia.

5.1.2. Cálculo de los puntos de intersección con el entorno

Para saber si un sensor va a detectar una pared es necesario conocer los siguientes datos:

- La posición de la pared.
- El ángulo de inicio y el de fin de medida del sensor.
- El máximo alcance (rango) del sensor. Este dato y el anterior están definidos como propiedades del sensor.

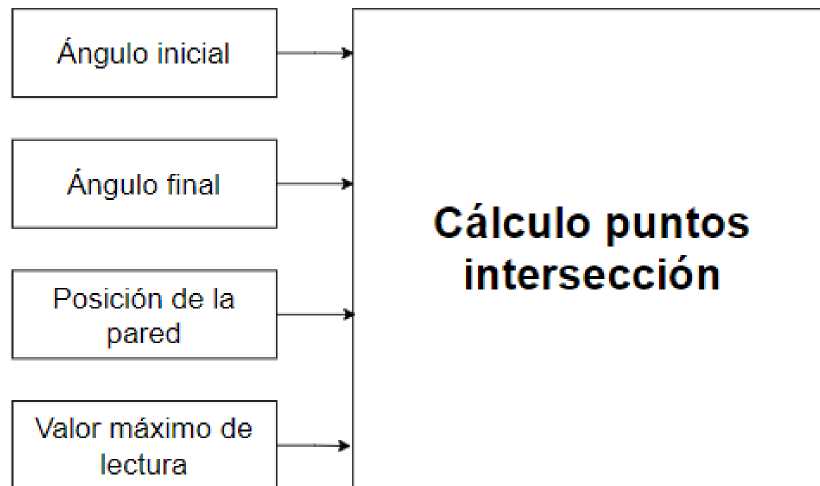


Figura 15. Parámetros para calcular los puntos de intersección con el entorno.
Fuente. Propia.

Para que el sensor láser detecte y devuelva la distancia y los valores de los puntos de intersección con la pared, se han creado dos funciones:

- **[distancia, punto] = interseccion(obj, robot)**

Esta función almacena los valores de la distancia y un único punto de intersección con la pared. Para ello, primero crea una matriz con la posición del robot y la del sensor y luego para cada línea del entorno (definidas en su propia matriz), extrae las coordenadas de la pared. A partir de ahí calcula el punto de intersección asegurándose de que está en el segmento de la pared y de que la distancia es menor a la del rango máximo del sensor. Por último, dibuja el punto de intersección con un círculo en rojo. Para los comportamientos del robot, el círculo rojo que representa el punto de intersección con la pared está desactivado. Esto se debe a que el rendimiento del programa disminuye considerablemente si, además de escanear el entorno, dibuja todos los puntos de intersección. Para probar solamente esta función, consultar el manual de usuario en el Anexo 9.5.

En la Figura 16 se muestra el resultado de la función *intersección()* de la clase *sensor_laser*, donde se puede observar el punto en rojo sobre la línea negra que representa una pared.

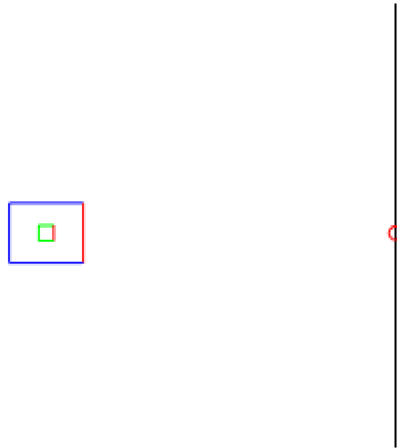


Figura 16. Punto de intersección con el entorno.

Fuente. Propia.

Por último, el diagrama de flujo que detalla el funcionamiento de esta función se muestra en la Figura 17.

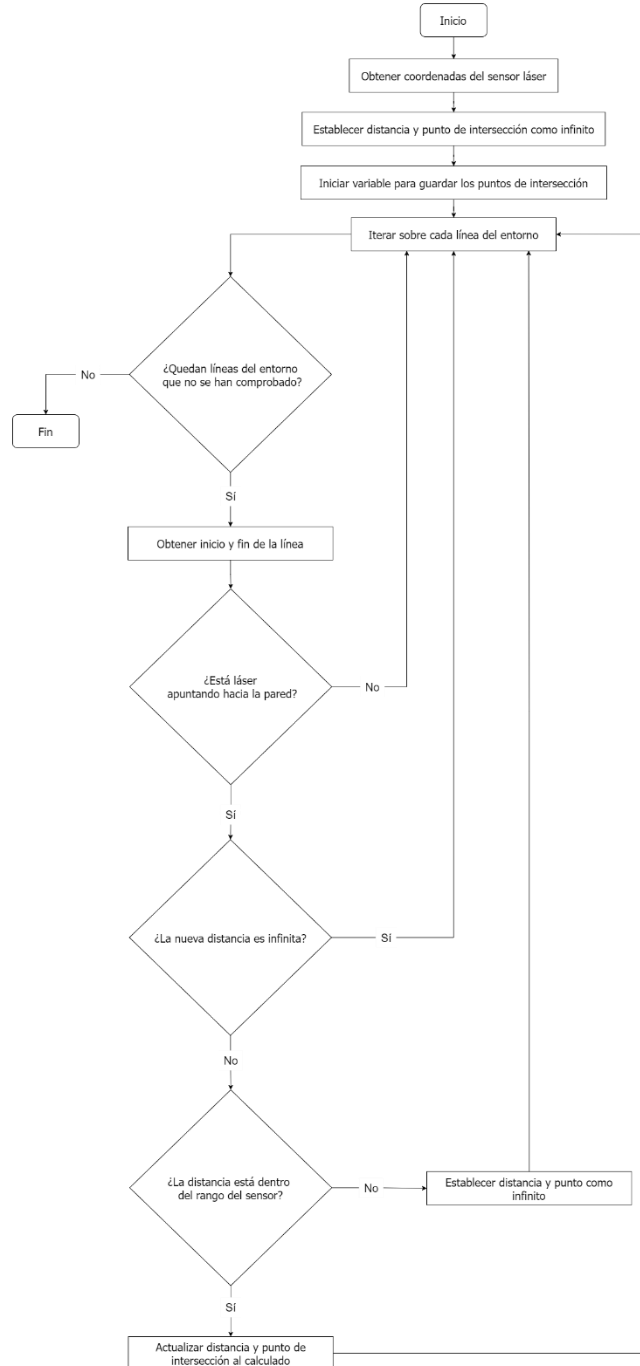


Figura 17. Diagrama de flujo de la función *Intersección.*

Fuente: Propia.

- **puntos_interseccion = calcular_interseccion_multiple(obj, robot)**

Esta función almacena en una matriz el ángulo de medida, la distancia y las coordenadas de los puntos de intersección que detecta el sensor. Para ello,

desde el ángulo inicial hasta el ángulo final, llama a la función *interseccion* para que calcule la distancia y las coordenadas del punto de intersección, aumentando el ángulo según la resolución angular para cada iteración del bucle.

Los puntos de intersección calculados mediante esta función se guardan en la propiedad del objeto llamada *puntos_interseccion.* En la Figura 18 se muestra un ejemplo de la representación de todos los puntos de intersección que puede detectar el sensor láser en una pared

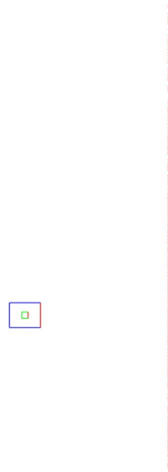


Figura 18. Resultado de la función *calcular_interseccion_multiple.*

Fuente. Propia.

Por último, el diagrama de flujo de la función *calcular_interseccion_multiple* se muestra en la Figura 19.

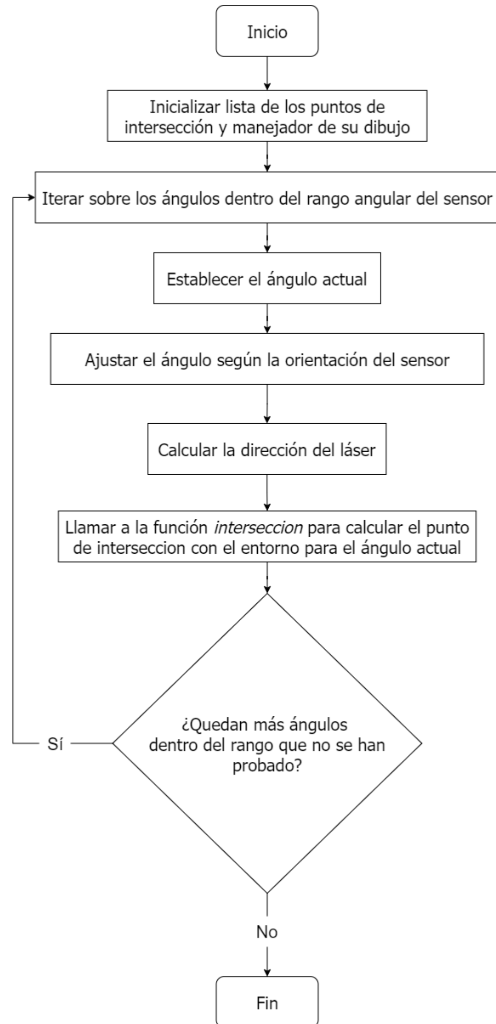


Figura 19. Diagrama de flujo de la función *calcular_interseccion_multiple.*

Fuente. Propia.

5.2. Clase robot

A diferencia del sensor láser, el robot no está basado en ningún modelo en específico. Su diseño se trata en un robot simple de forma cuadrada, con un sensor láser en el centro que escanea el entorno.

El objetivo principal de esta clase es hacer que el robot, junto con el sensor, se mueva siguiendo los diferentes comportamientos que se explicarán en este apartado. Para ello, se han implementado un total de 26 funciones de diverso carácter: para actualizar parámetros, dibujar el robot en el entorno, simulaciones de los comportamientos por separado, etc.

En esta sección se explicarán solamente las funciones referentes al movimiento del robot, los comportamientos y el piloto (arquitectura final del robot).

5.2.1. Propiedades

Para que el robot funcione adecuadamente, se han creado distintas propiedades dentro de la clase robot. Las más importantes son las siguientes:

- **Dibujo**
  Cumple la misma función que el sensor láser, pero aplicada al robot. Además, la parte delantera está representada con una línea roja y el resto del robot con líneas azules.

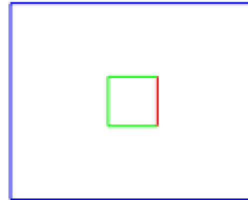


Figura 20. Dibujo del robot con el sensor láser en MATLAB.
Fuente. Propia.

- **Entorno**
  Esta propiedad guarda la matriz del entorno que se vaya a utilizar. Sirve para poder pasarla como argumento en algunas funciones con el fin de calcular los puntos de intersección con las paredes u obstáculos del entorno.

  Los entornos utilizados en esta arquitectura se encuentran referenciados en el Anexo 9.6.7.

- **Coordenadas iniciales y coordenadas finales**

  Estas son las coordenadas del robot en el entorno, almacenadas en un vector de 1x3, que representan las coordenadas cartesianas *x*, *y* y la orientación del robot, respectivamente.

Coordenada X	Coordenada Y	Ángulo de orientación del robot
1	1	π/4

Tabla 2. Ejemplo de las coordenadas del robot.

Fuente. Propia.

En la siguiente figura, la Figura 21, se puede ver una representación del robot en el entorno con las coordenadas de ejemplo de la Tabla 2, teniendo en cuenta que *xv* y *yv* son las coordenadas X e Y y θ es la orientación del robot.

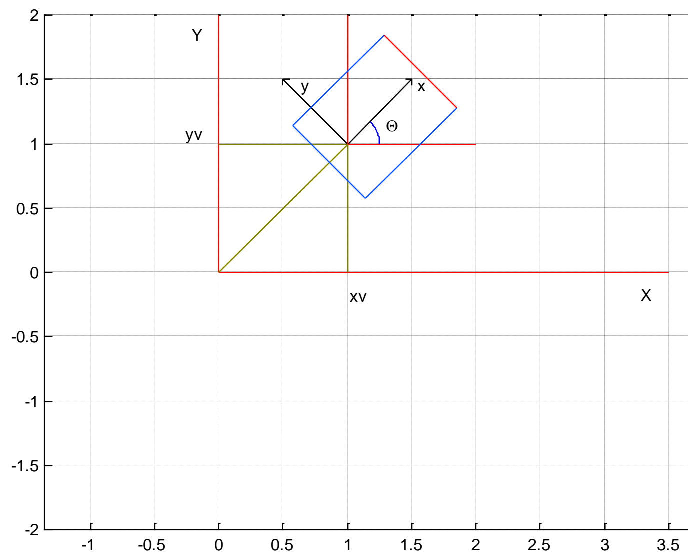


Figura 21. Representación de las coordenadas cartesianas del robot en el entorno.
Fuente. [24].

Las coordenadas iniciales indican la posición inicial del robot, mientras que las coordenadas finales representan el destino hacia donde se debe mover el robot. Generalmente, estas suelen tener el mismo valor debido a la función *mover_robot*, referenciada en el apartado 5.2.2, ya que, para calcular un nuevo movimiento del robot con unas nuevas coordenadas finales, las coordenadas iniciales se

actualizan a las coordenadas finales previas. Estas propiedades aparecen en el código de la clase robot como *coordenadas_iniciales* y *coordenadas_finales*.

- **Matriz del robot**

  Es la matriz homogénea del robot, de 4x4 que representa los cambios de rotación y traslación del robot. En la siguiente figura se muestra una representación de esta matriz con las coordenadas del robot *xv, xy,* θ mencionadas en el anterior apartado de coordenadas iniciales y finales.

$$M = \begin{bmatrix} \cos\theta & -\operatorname{sen}\theta & 0 & x_v \\ \operatorname{sen}\theta & \cos\theta & 0 & y_v \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Figura 22. Matriz homogénea del robot.

Fuente. [24].

  Para calcular esta matriz, se utiliza la función *XaMH*, referenciada en el Anexo 9.4.6, y se introducen las coordenadas finales para determinar los cambios en la posición y orientación del robot. En el código aparece como *matriz_robot*,

- **Sensor láser**

  Es el objeto creado por la clase *sensor_laser.* Se utiliza para acceder a las propiedades y funciones del láser, permitiendo, por ejemplo, conocer los puntos de intersección en el entorno como ejecutar la función que los calcula. En el código aparece como *sensor_laser.*

- **Constantes**

  Para poder simular el movimiento del robot, son necesarias algunas constantes:

  - **Tiempo**

Es el intervalo de tiempo, en segundos, durante el cual se está calculando el cambio en la posición y orientación del robot, es decir, es el tiempo que pasa entre la actualización de las coordenadas del robot. Generalmente, este intervalo está en el orden de centésimas de segundo. En el código aparece como *T*.

- **Constante de velocidad**

  Es la constante que regula la rapidez, en metros por segundo, con la que el robot se mueve entre las coordenadas iniciales y las finales. En el código aparece como *kr*.

- **Constate de rapidez de giro**

  Es la constante que regula la rapidez, en radianes por segundo, con la que el robot gira entre las coordenadas iniciales y las finales. En el código aparece como *kw*.

- **Velocidad lineal**

  Representa la velocidad lineal del robot, es decir, la rapidez con la que el robot se desplaza en línea recta en metros por segundo.

  Esta variable se utiliza para calcular la distancia que el robot recorre en un intervalo de tiempo determinado (*T*). Se calcula a partir del vector resultante, explicado en la siguiente sección. En el código aparece como *velocidad*

- **Velocidad angular**

  Representa la velocidad angular del robot, es decir, la rapidez con la que el robot gira alrededor de su eje vertical en radianes por segundo.

  Esta variable se utiliza para calcular el cambio en la orientación del robot en un intervalo de tiempo determinado

(*T*). Se calcula a partir del vector resultante, explicado en la siguiente sección. En el código aparece como *w*.

- **Vectores**

En este contexto, los vectores se utilizan para mover al robot según el comportamiento. Describen hacia dónde se debe desplazar el robot en los ejes *x* e *y*, determinando la dirección y la magnitud del movimiento.

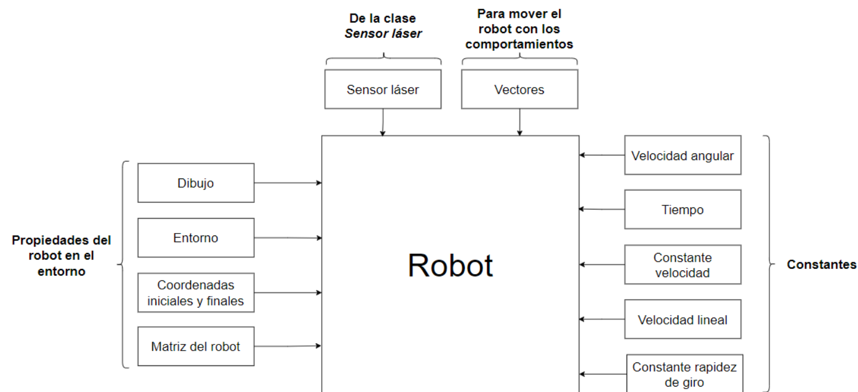


Figura 23. Diagrama de bloques de algunas propiedades de la clase *robot.*
Fuente. Propia.

Estas propiedades son algunas más importantes de la clase *robot*. El resto de las propiedades de la clase están detalladas en el Anexo 9.6.1.

5.2.2. Funciones de movimiento

El robot puede desplazarse en cualquier dirección a lo largo del eje *x* e *y* y también girar 360º en cualquier sentido.

Para simplificar los grados de giro, se utilizó la función *mod* [34] de MATLAB en las funciones de movimiento de modo que los grados de la orientación del robot (tercer elemento de las coordenadas) se mantuvieran dentro del rango de 0 a 2π radianes o lo que es lo mismo, de 0 a 360º.

En este apartado se desarrollarán las diferentes funciones de la arquitectura que hacen que el robot se mueva.

- ***mover_robot(obj, coordenadas_finales)***

  Esta función mueve el robot junto con el láser a unas coordenadas definidas por *coordenadas_finales*.

  Además, una vez el robot se encuentra en la posición de esas coordenadas, actualiza todos los parámetros necesarios (matriz del robot, matriz del sensor láser, dirección del láser, etc) para poder mover el robot a otras coordenadas en futuras llamadas a esta función.

  Por otra parte, esta función también tiene implementada la llamada a calcular los puntos de intersección con el láser, por lo que, si el robot se mueve, escanea el entorno y devuelve los puntos de intersección (si hay) con el mismo.

  En la Figura 24 se muestra una comparación del robot antes y después de usar la función *mover_robot*. Inicialment el antes muestra al robot en la posición [0,0,0] y en el después se muestra cómo cambia a [1,1,pi/2].

Antes **Después**

Figura 24. Representación del antes y después de ejecutar *mover_robot.*
Fuente. Propia.

En la Figura 25 se muestra el diagrama de flujo de la función.

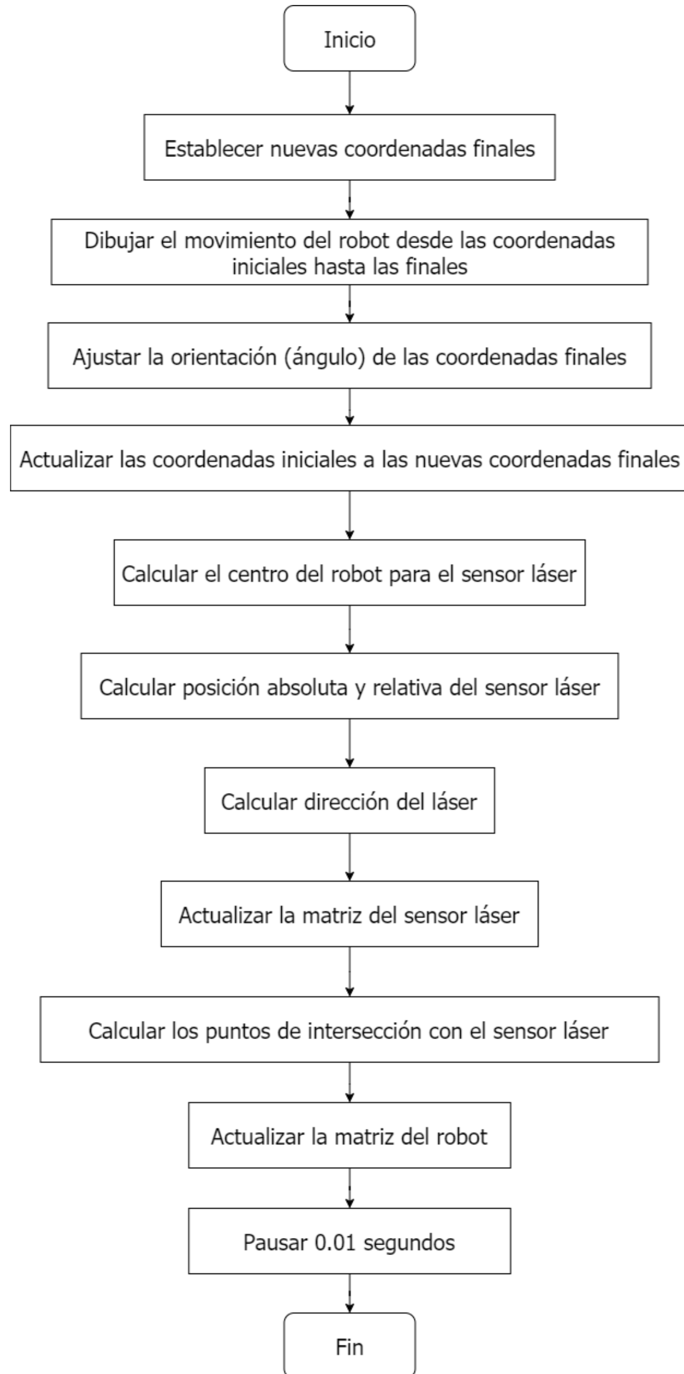


Figura 25. Diagrama de flujo de *mover_robot.*

Fuente. Propia.

La función aparece detallada en el Anexo 9.6.1 de las líneas 80 a la 96.

- ***simulación_giro_robot(obj, velocidad, radio)***

  Esta función mueve el robot siguiendo un radio a una velocidad constante. Para ello, se le pasa como argumentos la velocidad y el radio que tiene que seguir.

  A diferencia de *mover_robot* cuando esta función se ejecuta, se puede ver paso a paso como el robot se mueve. Esto se debe a que dentro de la función se calculan las coordenadas de giro poco a poco y se actualizan con las propiedades del robot (*T, w,* etc) mencionadas anteriormente en el apartado 5.2.1.

  Además, también escanea el entorno para encontrar puntos de intersección, ya que utiliza la función *mover_robot* que tiene implícita la llamada a *calcular_interseccion_multiple* de la clase *sensor_laser*.

  En la siguiente figura, la Figura 26, se muestra la representación al ejecutar la función *simulación_giro_robot* con una velocidad de 0.5 metros por segundo y un radio de 1 metro.

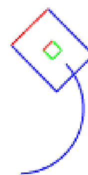


Figura 26. Resultado función *simulación_giro_robot.*
Fuente. Propia.

  El diagrama de flujo de esta función está representado en la Figura 27.

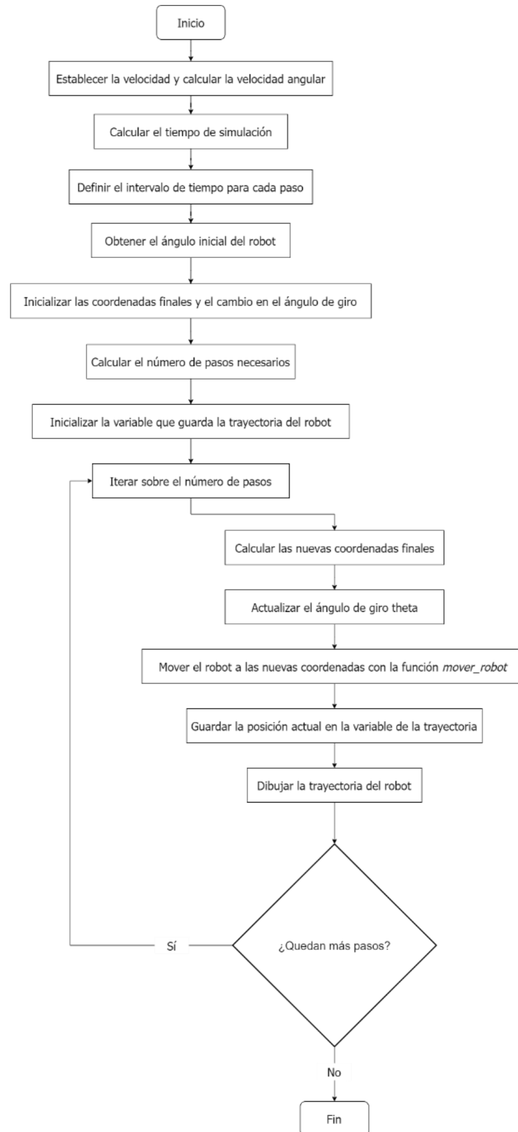


Figura 27. Diagrama de flujo de *simulacion_giro_robot.*

Fuente. Propia.

La función aparece detallada en el Anexo 9.6.1 de la línea 98 hasta la 117.

- ***girar_robot(obj, grados)***

  Esta función gira al robot sobre su propio eje hasta alcanzar los grados definido por la variable *grados.* Además, puede girar al robot en el sentido más corto hasta llegar a los grados especificados; es decir, puede girar en sentido horario o antihorario, dependiendo de cuál ruta sea más corta.

Al igual que *simulación_giro_robot* se utilizan los parámetros mencionados en el apartado 5.2.1. para poder ver paso a paso como el robot se mueve. También escanea el entorno con los puntos de intersección, ya que utiliza la función *mover_robot*.

En la siguiente figura, Figura 28, se muestra el resultado de *girar_robot* a 90º (inicialmente estaba a 0º) con algunos pasos intermedios hasta alcanzar los grados deseados.

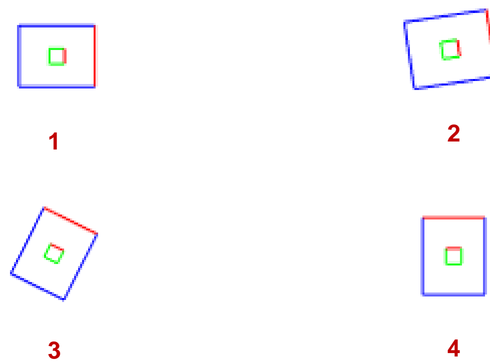


Figura 28. Representación de algunos pasos de la función *girar_robot.*
Fuente. Propia.

El diagrama de flujo de la función *girar_robot* está representado en la Figura 29.

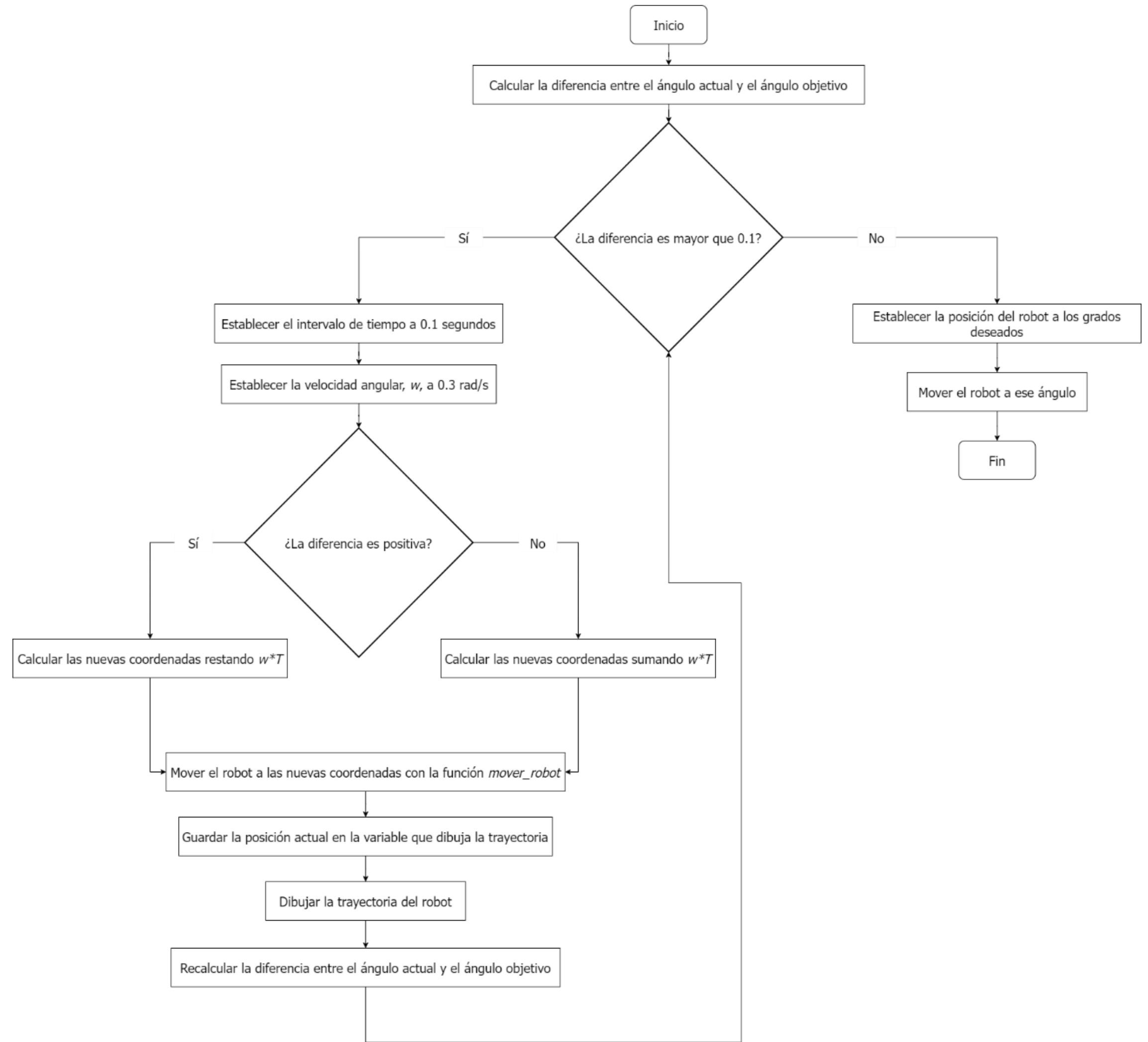


Figura 29. Diagrama de flujo de *girar_robot.*

Fuente. Propia.

La función *girar_robot* aparece detallada en el Anexo 9.6.1. de las líneas 1160 hasta la 1184.

Con todas estas funciones que mueven el robot, se han implementado los comportamientos que se describen en el siguiente apartado.

5.2.3. Comportamientos

El objetivo principal de este proyecto es poder implementar una arquitectura funcional de un robot móvil en MATLAB. Con el fin de cumplirlo, se han desarrollado diferentes comportamientos que tiene que seguir el robot en un entorno de MATLAB. Para ello, cada comportamiento devuelve un vector que indica hacia donde tiene que moverse el robot. En este apartado, se explican los comportamientos que puede seguir el robot.

- **Ir hacia un objetivo**

  Este comportamiento hace que el robot vaya hacia un objetivo, es decir, hacia unas coordenadas en concreto. La función de este comportamiento se llama *objetivo* de la clase *robot*, referenciada en el Anexo 9.6.1 de la línea 208 hasta la 220.

  Para que el robot se mueva poco a poco hasta el objetivo, se siguen los siguientes pasos:

  1. Se declaran las coordenadas del objetivo
  2. Se calcula la distancia del robot hasta el objetivo.
  3. Se declara la distancia mínima al objetivo a la que el robot tiene que parar.
  4. Se calcula un vector de atracción hacia el objetivo.
  5. Se mueve el robot a las nuevas coordenadas obtenidas con el vector objetivo siguiendo la ecuación de movilidad del robot, referenciada en el Anexo en el apartado tal.

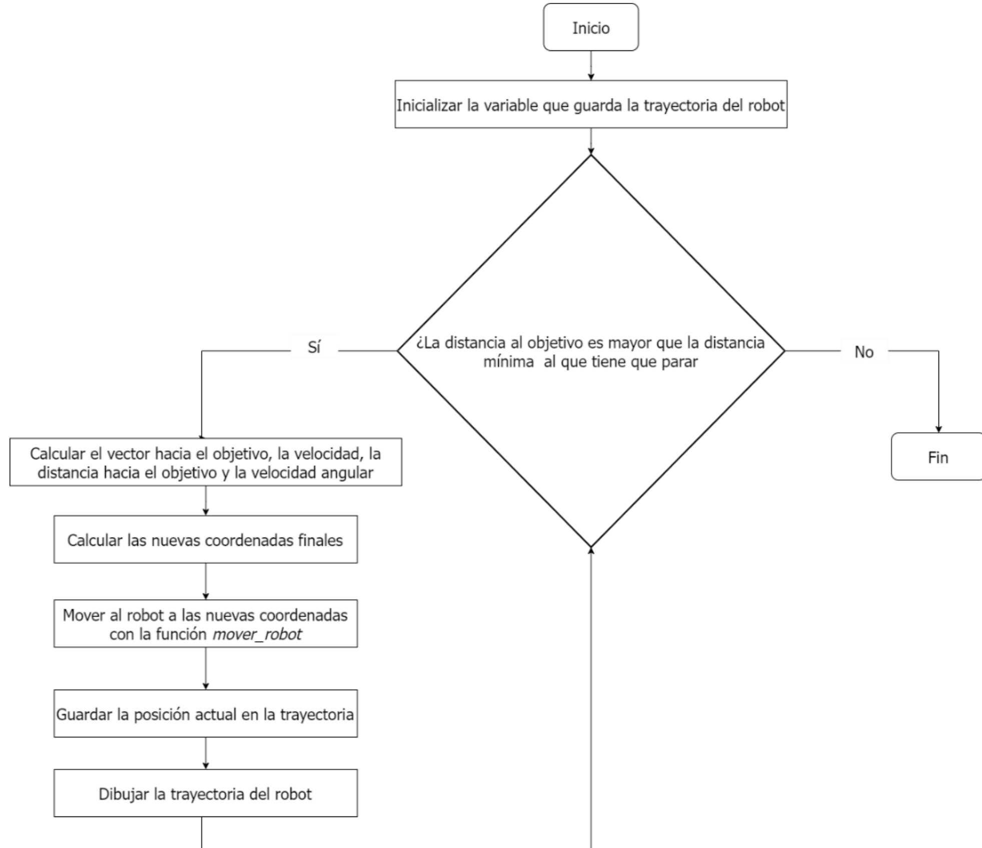


Figura 30. Diagrama de flujo de la función *objetivo.*

Fuente. Propia.

El vector de este comportamiento se calcula con la función *Vect_objetivo()* de la clase robot, referenciada en el Anexo 9.6.1 de la línea 187 hasta la 206. Por último, el cálculo para obtener el vector se encuentra en el Anexo 9.4.5.

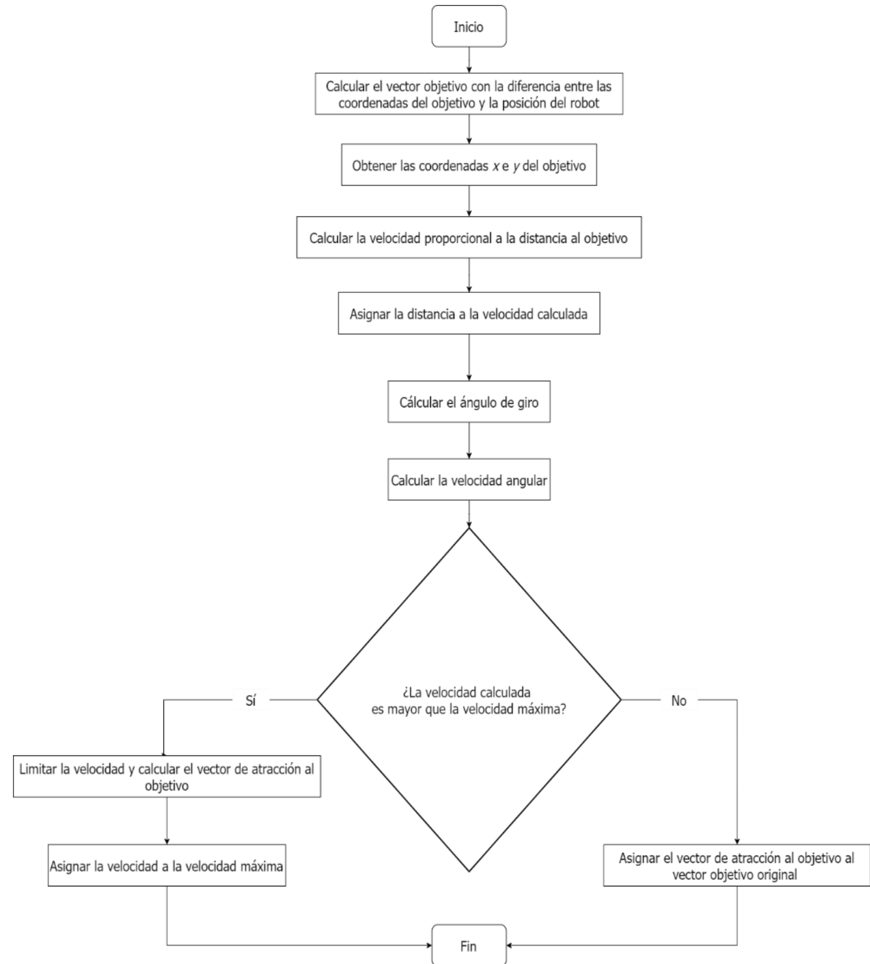


Figura 31. Diagrama de flujo de la función *Vect_objetivo.*

Fuente. Propia.

Con esto, el robot es capaz de orientarse y moverse hasta llegar a un objetivo definido por unas coordenadas.

- **Evitar obstáculos**

  Para que el robot pueda evitar obstáculos, se plantea el mismo principio que en el comportamiento de ir hacia un objetivo, es decir, el comportamiento debe de devolver un vector que indique hacia donde se tiene que mover el robot.

  Para ello, primero se definen los ángulos específicos del sensor láser con el fin de conocer en todo momento la ubicación de los objetos en el entorno. Estos ángulos permiten

al robot escanear el entorno de manera similar a cómo funcionan otros tipos de sensores, como los ultrasonidos. La selección de estos ángulos como la de sus pesos ha sido realizada por prueba y error. Los ángulos seleccionados se pueden ver en la siguiente imagen, Figura 32.

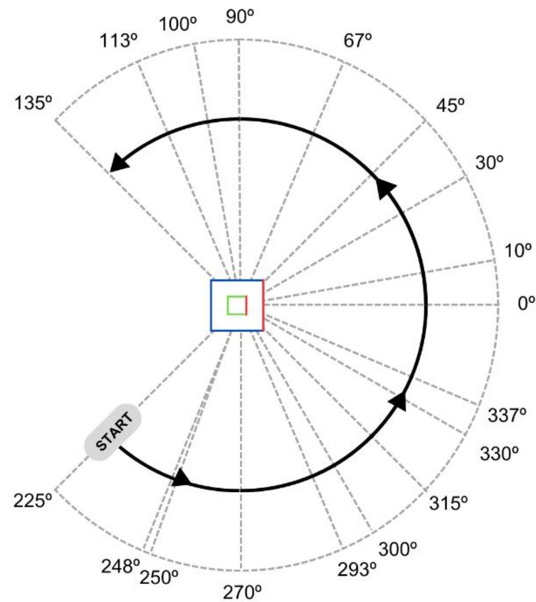


Figura 32. Ángulos seleccionados para la evitación de obstáculos
Fuente. Propia.

Una vez definidos los ángulos, para cada uno de ellos se le asigna un peso. Por ejemplo, los ángulos que corresponden a la parte trasera del robot tienen menor peso que los de la parte delantera o los laterales. Así se asegura que el robot puede esquivar obstáculos de forma eficiente. Por último, se calcula el vector de repulsión de obstáculos para cada ángulo.

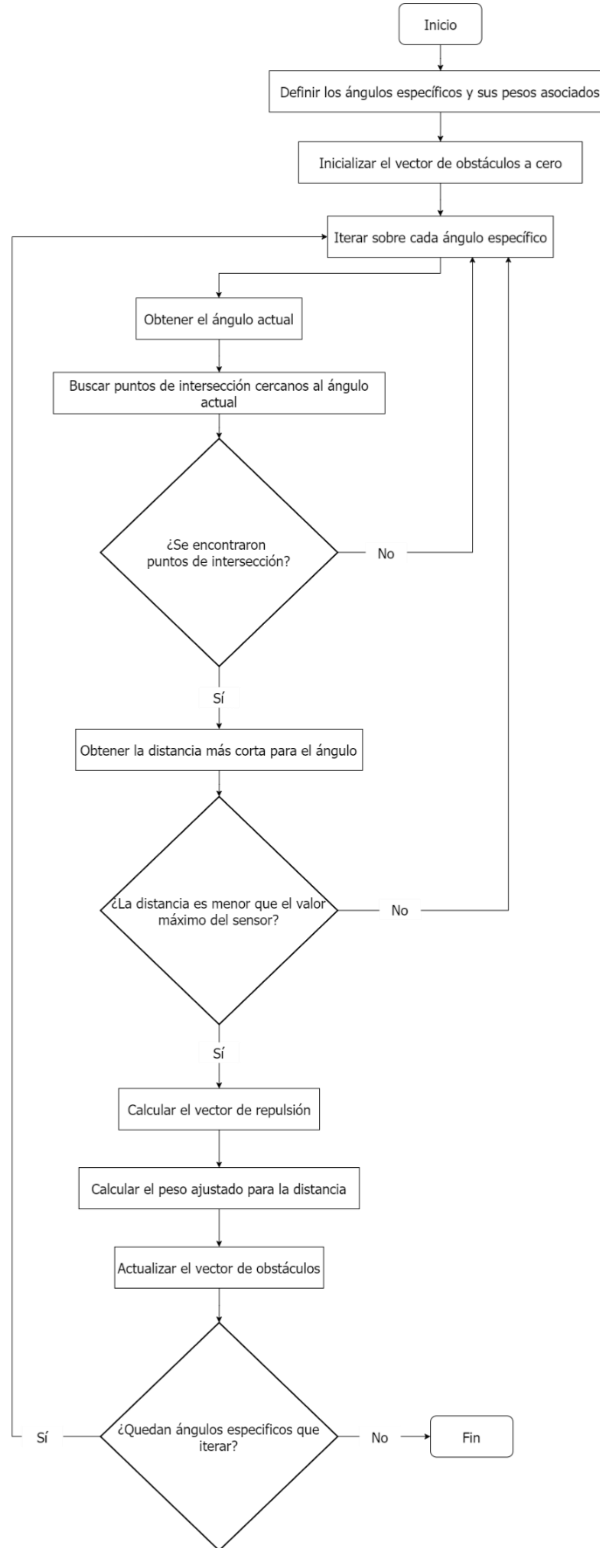


Figura 33. Diagrama de flujo de la función *evita_obstaculos.*

Fuente. Propia.

El cálculo y la fijación de estos sensores se rencuentra en la función *evita_obstaculos* en el Anexo 9.6.1 de la línea 291 hasta la 315. Para más información sobre el cálculo, consultar el Anexo 9.4.6.

- **Seguir una pared**

  Para que el robot pueda seguir una pared, primero es necesario indicar qué pared tiene que seguir. Si por ejemplo se encuentra en un pasillo, puede seguir tanto la pared izquierda como la derecha. Esta elección está reflejada en la propiedad *lado* del robot. Si el lado es igual a 0 sigue la pared derecha y si vale 1, sigue la pared izquierda.

Además de seguir la pared, es capaz de seguirla en paralelo a una distancia de referencia que se le indique, en este caso, esa distancia es una propiedad de la clase *robot* llamada *d_ref*.

Para que pueda seguir la pared eficientemente sin chocarse, al decirle de qué lado está la pared a seguir, se activan los ángulos de medición de ese lado para medir constantemente a que distancia está de la pared.

En esta arquitectura, los ángulos para el comportamiento de seguir la pared están definidos por las propiedades del robot *a, b,* y *c,* que en este caso equivalen a 320º, 270º, 225º para la derecha del robot y 40º, 90º y 135º para la izquierda del robot, respectivamente. Para simplificar este comportamiento, se ha implementado una función que filtra los puntos de intersección del láser a solamente los ángulos definidos por *a, b,* y *c.* Esta función es *filtrar_puntos()* detallada en el Anexo 9.6.1.

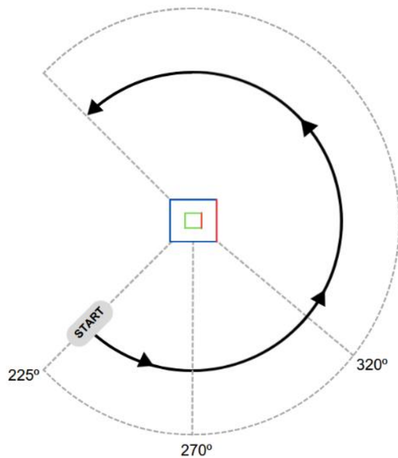


Figura 34. Ángulos para la pared derecha.

Fuente. Propia.

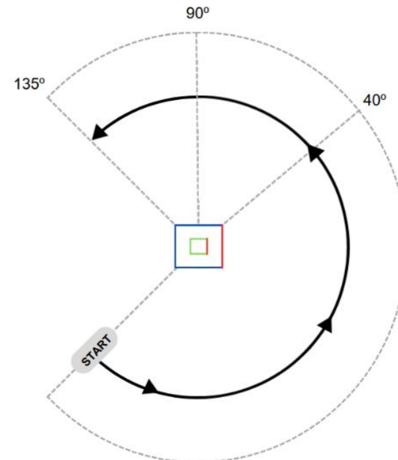


Figura 35. Ángulos para la pared izquierda.

Fuente. Propia.

En este caso, el vector de orientación hacia la pared se calcula mediante la función *orientación_pared()* referenciada en el Anexo 9.6.1 de la línea 515 hasta la línea 578. Esta función extrae las distancias medidas por *a* y *b* y calcula la diferencia entre ellas para mover el robot en esa dirección.

Además, también calcula la diferencia de distancia entre *b* y la distancia de referencia a la pared, ya que el ángulo de medición definido por *b* es perpendicular al robot y por tanto la distancia que mida tiene que ser la misma que la de referencia. Si la diferencia entre *b* y la distancia de referencia es menor a 0.1:

- Se asume que el robot está a la distancia de referencia de la pared
- Se mueve el robot a la coordenada x del punto de intersección de *b* para que la próxima vez que calcule la diferencia entre *b* y *d_ref* esta sea 0.
- Se gira el robot hasta que esté a 90º (paralelo a la pared)

- Se declara como [0,0] el *vector_orientacion*

Todos estos cálculos están explicados en el Anexo 9.4.7.

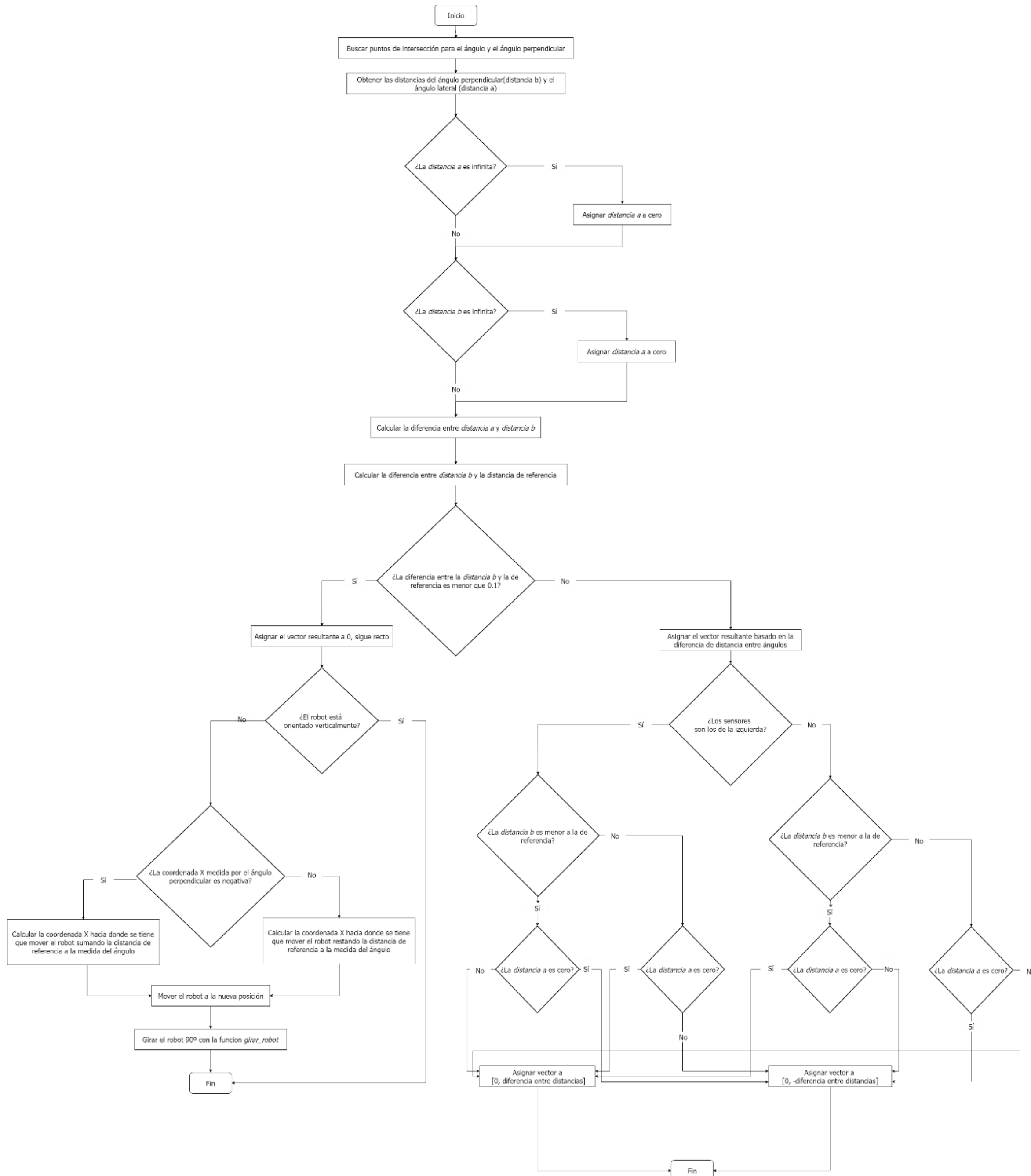


Figura 36. Diagrama de flujo de la función *orientacion_pared.*

Fuente. Propia.

Además, se declara un vector para que el robot pueda avanzar recto, en este caso *vector_seguir_recto*. La explicación de este vector está detallada en el Anexo 9.4.4.

El vector que devuelve este comportamiento se guarda en la propiedad *vector_resultante* de la clase *robot*. Este vector es la suma de dos vectores, *vector_seguir_recto* y *vector_orientacion* multiplicados por sus respectivos pesos *k1* y *k2*.

Con esto, el robot es capaz de seguir en paralelo una pared indicándole la distancia a la pared a la que se tiene que mantener. Este comportamiento está declarado en la función *seguir_pared()* en el Anexo 9.6.1. de las líneas 351 hasta la 423.

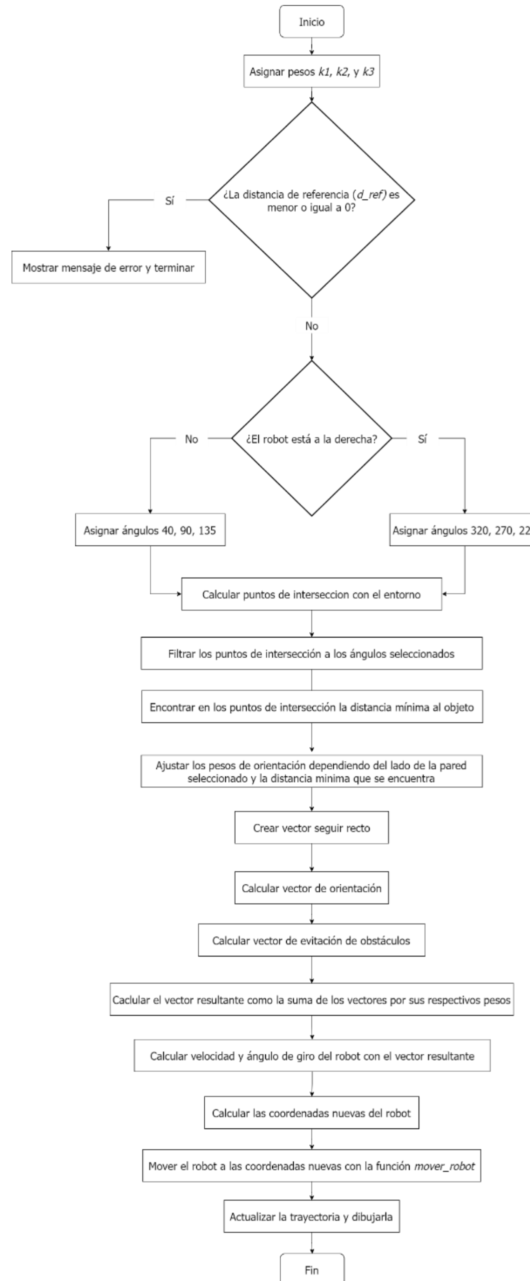


Figura 37. Diagrama de flujo de la función *seguir_pared.*

Fuente. Propia**.**

- **Seguir un pasillo por el centro**

  Para que el robot pueda seguir un pasillo por el centro de este, se sigue el mismo planteamiento que en el apartado anterior, en el comportamiento de seguir pared. La diferencia está en que, en vez de utilizar 3 ángulos de un lado, se utilizan los 3

ángulos de cada lado y el frontal, es decir, en total se usan 7 ángulos que vienen declarados en las propiedades *a, b, c, d, e, f* y *g* de la clase *robot*. Estos ángulos son: 320º, 270º, 225º, 40º, 90º, 135º y 0º, que se corresponden a las propiedades anteriores respectivamente.

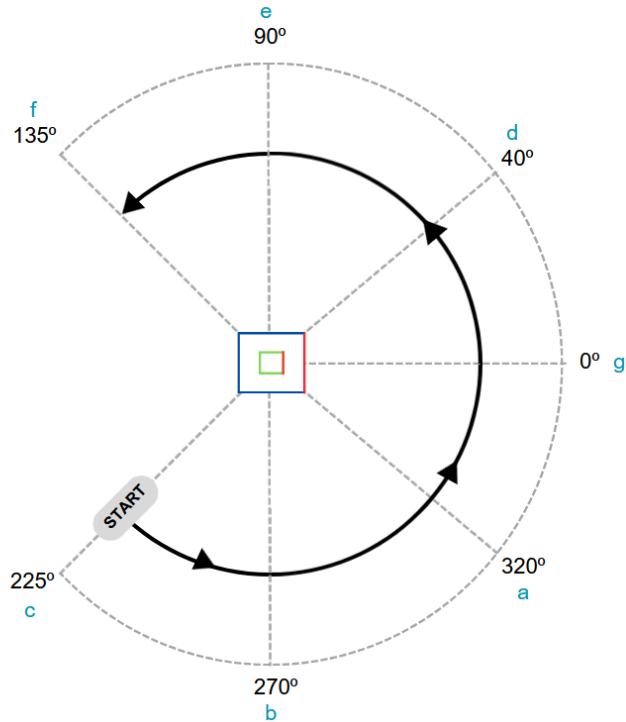


Figura 38. Representación de los ángulos utilizados en la función *seguir_pasillo.*

Fuente. Propia.

Para empezar, primero se comprueba si alguno de los dos ángulos perpendiculares (*b, e)* da infinito, si el sensor frontal detecta algún objeto y si el robot está orientado verticalmente. Esto es para que el robot pueda avanzar (vector resultante = [1, 0]) a la mitad de la distancia del ángulo lateral y frontal en caso de que uno de los sensores no detecte nada.

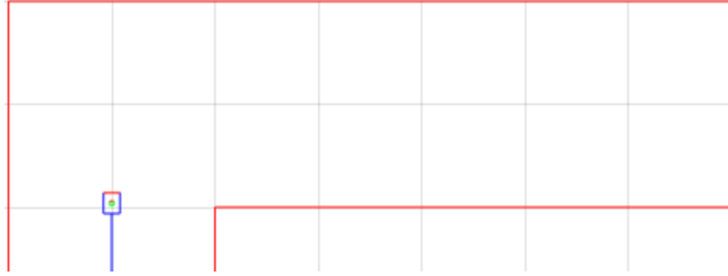


Figura 39. Ejemplo de cuando uno de los sensores laterales de infinito.

Fuente. Propia.

Una vez está a esa distancia, gira para seguir recto por el pasillo hasta que detecte algo en el ángulo lateral que daba infinito. En este caso, el vector resultante sería [1,0] para que pueda seguir recto.

Si ninguna de esas dos condiciones se cumple, se calcula el vector que mueve al robot al centro del pasillo, definido por la función *orientacion_pasillo()*, referenciada en el Anexo 9.6.1 de las líneas 719 hasta la 825.

En esta función, se extraen las distancias medidas por los ángulos 90º y 270º (perpendiculares al robot) para comprobar si son iguales, ya que, si la diferencia entre ellas es 0, se asume que el robot está en el centro del pasillo.

Si la diferencia es menor a 0.08, se asume que el robot ha llegado al centro del pasillo y se comprueba si el robot está en vertical u horizontal. Si lo está, el robot avanza recto (vector_orientacion = [0,0]) y si no, el robot gira 90º y calcula el centro del pasillo para mover el robot hacia allí. Así, la próxima vez que se calcule la diferencia entre las distancias de los ángulos, dará 0, puesto que el robot estará centrado perfectamente en el pasillo.

Si la diferencia entre las distancias no es menor a 0.08, entonces se comprueba que distancia de las dos es mayor para mover al robot a un lado o al otro. Si por ejemplo la distancia de la derecha es mayor, el robot va hacia la derecha. Para ello, si la distancia de la izquierda es mayor, se utilizan las distancias de los ángulos correspondientes a la parte izquierda del robot y se restan para obtener el vector resultante. En cambio, para la derecha, se usan los ángulos situados a la derecha del robot.

Para consultar el diagrama de flujo de esta función, revisar el plano número 12 del documento de planos de esta memoria.

El vector resultante de este comportamiento es la suma del vector que hace que el robot vaya recto (*vector_seguir_recto)* y el vector de la orientación en el pasillo (*vector_orientacion)*. El cálculo del vector de orientación está detallado en el Anexo 9.4.8.

El comportamiento viene definido por la función *seguir_pasillo()* referenciada en el Anexo 9.6.1 de las líneas 628 hasta la 717.

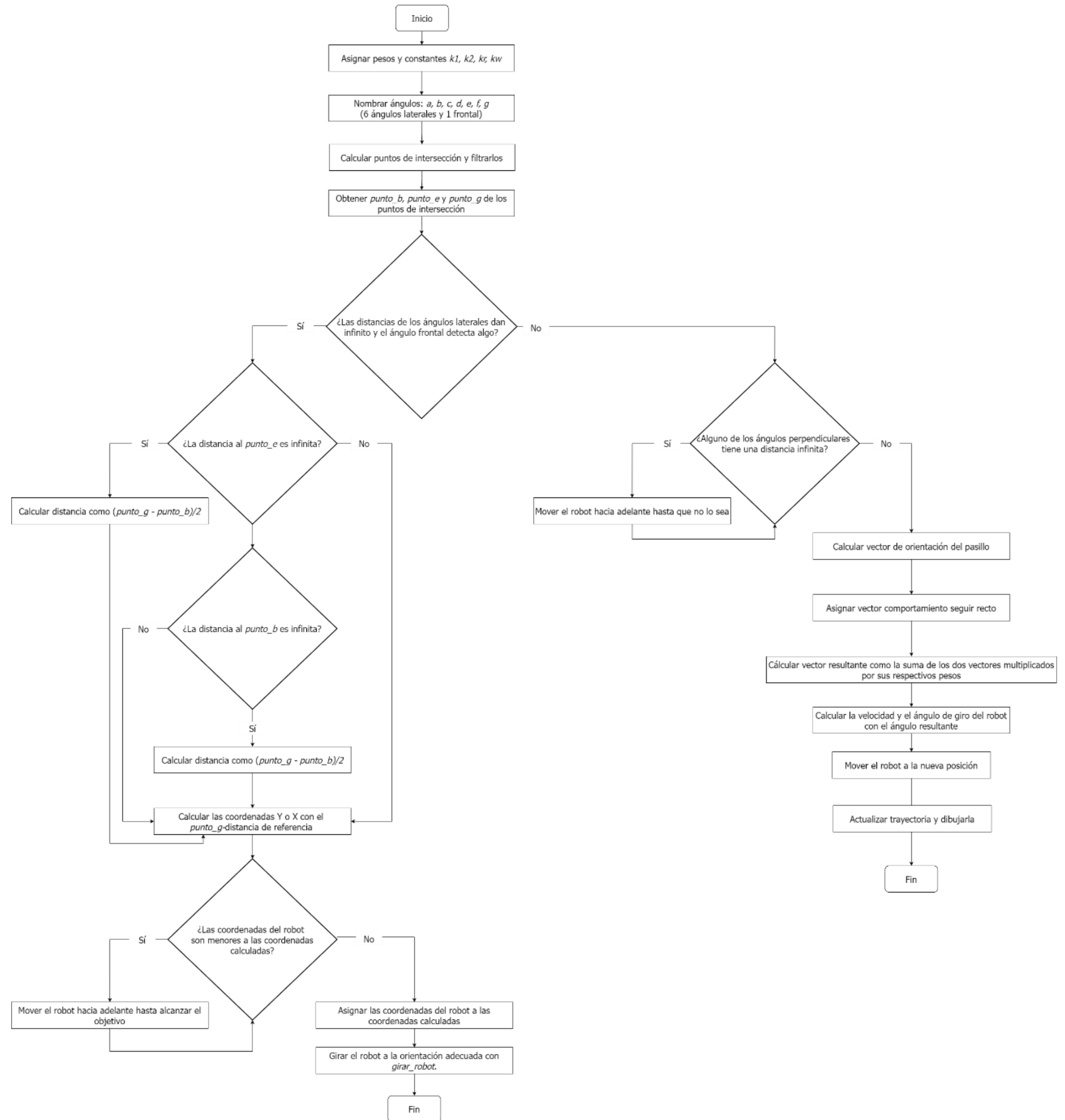


Figura 40. Diagrama de flujo de la función *seguir_pasillo*.

Fuente. Propia.

- **Atravesar puertas**

  Para el comportamiento de atravesar puertas, son necesarios dos vectores: el vector de evitación de obstáculos (para que no se choque con el marco de la puerta) y un vector de atracción al infinito, es decir, hacia donde no hay obstáculos.

El cálculo del vector del infinito se implementa para que el robot se desplace hacia donde no hay ningún obstáculo, en este caso, una puerta. Para ello, se seleccionan unos ángulos en específico: 4 ángulos de la derecha (0, 338, 330, 315) y 4 de la izquierda (67, 45, 30, 10) que son los ángulos frontales del robot. La selección de estos ángulos se debe a que el robot tiene que atravesar la puerta frontalmente y así en todo momento se sabe si los ángulos frontales dan infinito para moverse hacia allí.

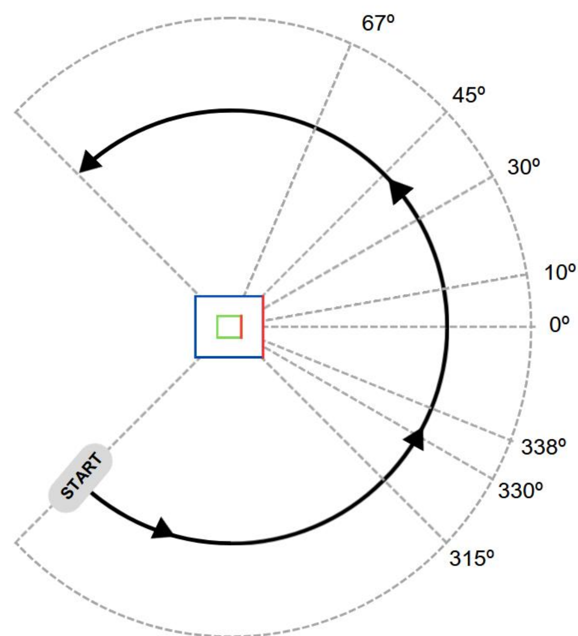


Figura 41. Ángulos para la función *atravesar_puertas.*

Fuente. Propia.

Luego de definir los ángulos, se definen unos pesos específicos para cada uno de ellos y se utilizan para el cálculo del vector de atracción al infinito. Este cálculo del vector está implementado en la función *infinito()* referenciada en el Anexo 9.6.1 de la línea 892 hasta la 920.

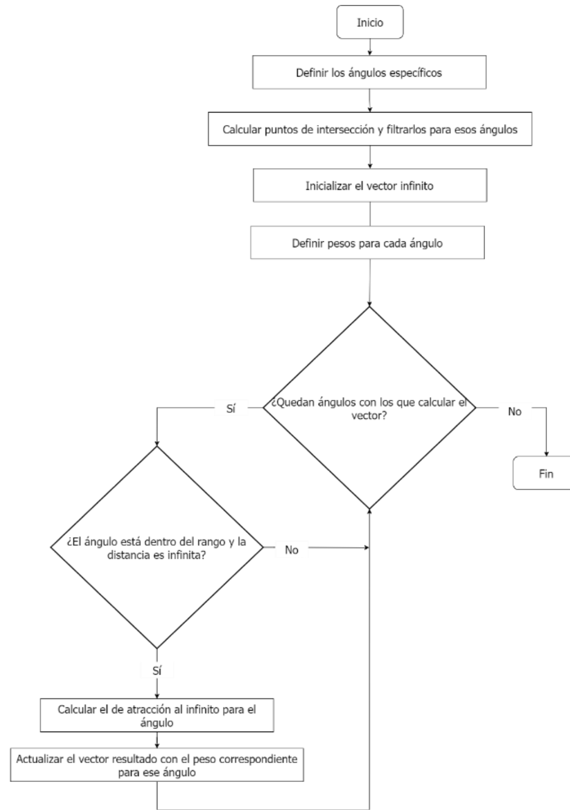


Figura 42. Diagrama de flujo de la función *infinito.*

Fuente. Propia.

Para más detalles sobre el cálculo del vector de atracción al infinito, consultar el Anexo 9.4.9. Una vez calculado el vector de atracción al infinito y el vector de obstáculos, el vector resultante es la suma de los dos con sus respectivos pesos.

La implementación de este comportamiento viene dada por la función *atravesar_puerta(),* referenciada en el Anexo 9.6.1 de la línea 853 hasta la 890.

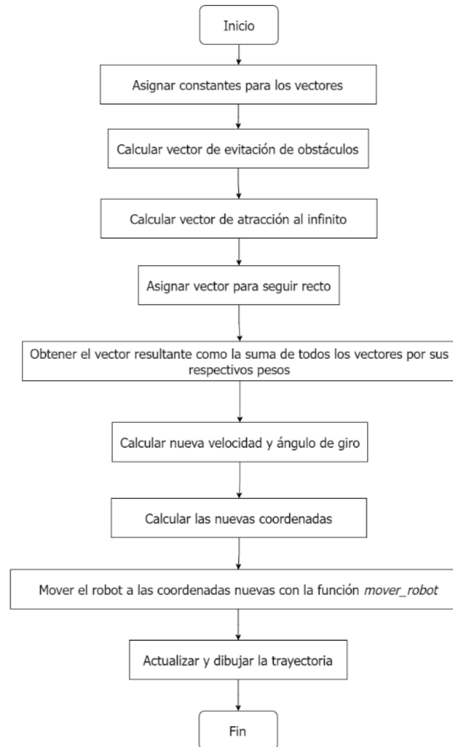


Figura 43. Diagrama de flujo de la función *atravesar_puertas.*

Fuente. Propia.

- **Aparcar**

  Este comportamiento hace que el robot se mueva a un área determinada y gire hasta alcanzar 90º para que simule haber aparcado. El área de aparcamiento viene dada por un punto en el entorno. Desde este punto, se declaran la distancia lateral derecha (*distancia_derecha),* lateral izquierda (*distancia_izquierda)* y frontal (*distancia_forntal)*. Con esas distancias, se trazan las líneas que delimitan el área con la función *vector_aparcamiento()*, referenciada en el Anexo 9.6.1 de la línea 983 hasta la 997*.* Para más detalles sobre el cálculo del vector, consultar el Anexo 9.4.10. El diagrama de la función *vector_aparcamiento()* se encuentra en la Figura 44.

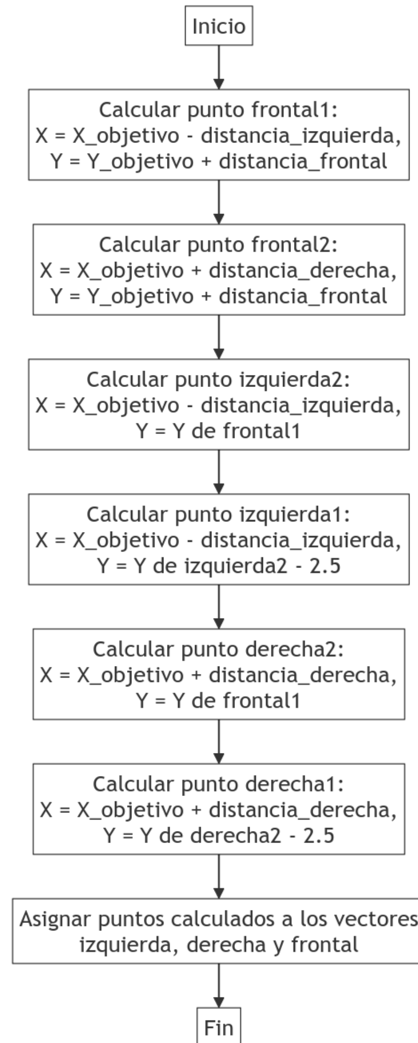


Figura 44. Diagrama de flujo de la función *vector_aparcamiento*.

Fuente. Propia.

Luego, se traza unas coordenadas objetivo, que tienen como componente X la componente X del centro del área y la componente Y como el centro del área menos 2 unidades. De esta forma, el robot activa el comportamiento hacia el objetivo, pero sin cocharse con los laterales del área. Al llegar al objetivo, gira hasta estar posicionado en 90º y sigue recto hasta estar pegado a la distancia frontal del área.

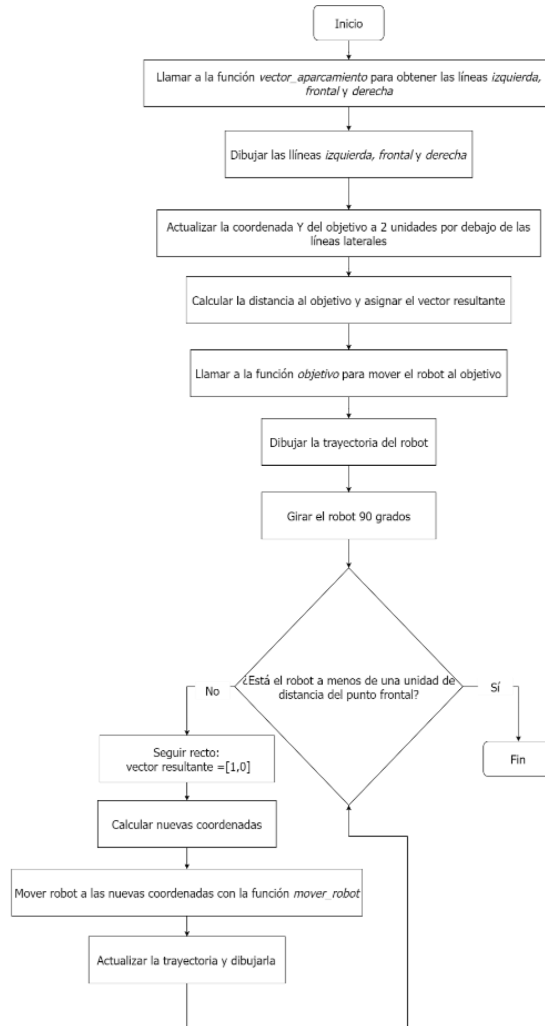


Figura 45. Diagrama de flujo de la función *aparcar.*

Fuente. Propia.

Este comportamiento viene dado por la función *aparcar()* referenciada en el Anexo 9.6.1 de la línea 951 hasta la 981.

5.2.4. Planificador

En esta arquitectura, se ha implementado un planificador Un planificador es un algoritmo que determina la secuencia de acciones necesarias para alcanzar un objetivo, considerando los obstáculos en el camino y optimizando el uso de recursos y tiempo.

En este contexto, el planificador utiliza un mapa topológico cualitativo del entorno, conocido previamente. En la Figura 46, se muestra un ejemplo de un entorno utilizado para el planificador.

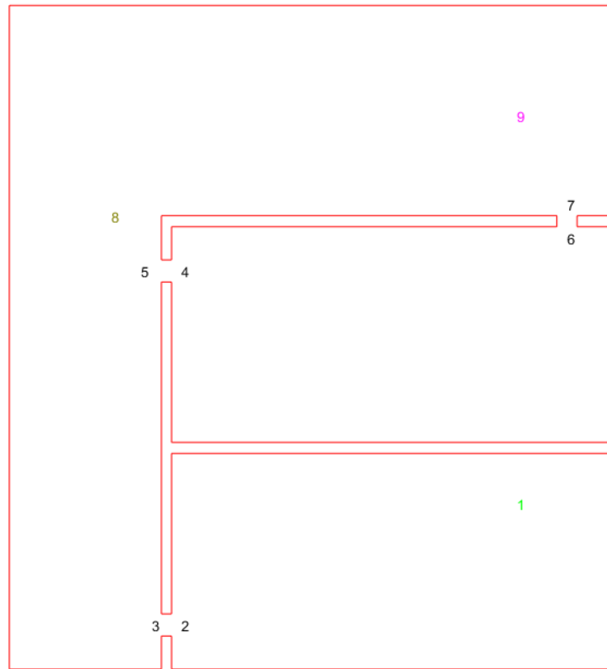


Figura 46. Ejemplo de un entorno para el planificador.
Fuente. Propia.

Como se puede ver, este entorno está definido por números que representan el camino a seguir. Estos números, también llamados nodos, tienen un color en específico que representa el tipo de nodo que es. En este caso, los de color negro (2,3, 4,5, 6,7) son los nodos asociados a las puertas de las habitaciones. El nodo en verde claro (1) representa el punto de partida del robot. El nodo en verde oscuro (8) representa el final del pasillo. Por último, el nodo en morado (9) representa el objetivo al que debe llegar el robot.

Para saber el camino a seguir, se utiliza una matriz de costes. Esta matriz es una representación que describe los costes asociados a moverse de un nodo a otro. Es una matriz

cuadrada de *n* x *n* donde *n* es el número de nodos en el entorno. Suponiendo una matriz de 4x4, un ejemplo de cómo se vería la matriz de costes se encuentra en la Tabla 3.

	NODO 1	**NODO 2**	**NODO 3**	**NODO 4**
NODO 1	Coste de moverse del **nodo 1 al 1**	Coste de moverse del **nodo 1 al 2**	Coste de moverse del **nodo 1 al 3**	Coste de moverse del **nodo 1 al 4**
NODO 2	Coste de moverse del **nodo 1 al 2**	Coste de moverse del **nodo 2 al 2**	Coste de moverse del **nodo 2 al 3**	Coste de moverse del **nodo 2 al 4**
NODO 3	Coste de moverse del **nodo 1 al 3**	Coste de moverse del **nodo 2 al 3**	Coste de moverse del **nodo 3 al 3**	Coste de moverse del **nodo 3 al 4**
NODO 4	Coste de moverse del **nodo 1 al 4**	Coste de moverse del **nodo 2 al 4**	Coste de moverse del **nodo 3 al 4**	Coste de moverse del **nodo 4 al 4**

Tabla 3. Matriz de costes.

Fuente. Propia.

Como se puede ver, cada celda de la matriz representa el coste de moverse entre dos nodos específicos. Si dos nodos no están conectados directamente, su coste se representa con un valor infinito. Por otra parte, si el nodo es el mismo (por ejemplo, moverse de 1 a 1) el coste se representaría como 0.

Para esta arquitectura, la matriz de costes se utiliza en el algoritmo de Dijkstra. Este algoritmo encuentra el camino más corto desde un nodo inicial hasta el resto de los nodos. Para ello, utiliza la matriz de costes para conocer las distancias (o costes) entre los nodos y les asigna un peso, asegurándose

de que este peso no sea negativo. Primero, crea una lista de puntos visitados y no visitados, y asigna costes iniciales para moverse entre ellos. En un bucle, selecciona el punto no visitado con el menor coste, actualiza los costes y caminos para sus puntos vecinos, y mueve el punto a la lista de visitados. Este ciclo se repite hasta que se procesen todos los puntos o se alcance el punto final. Finalmente, el algoritmo construye el camino más corto y calcula su coste total. Este algoritmo se encuentra en el Anexo 9.6.4.

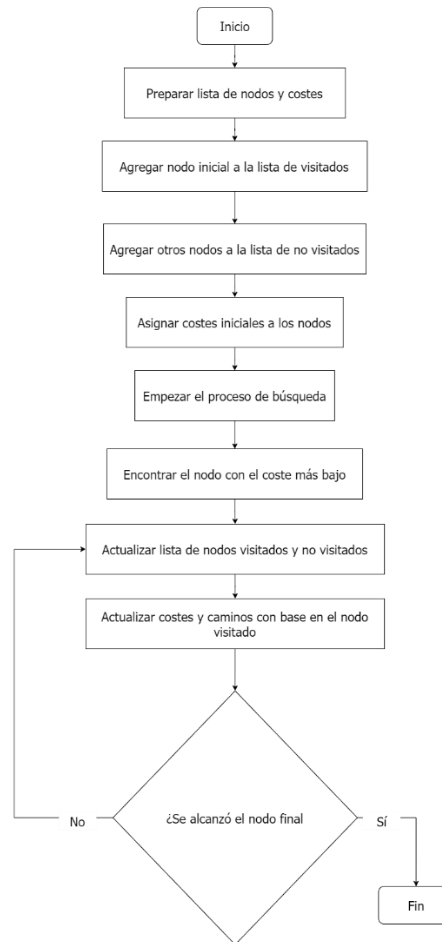


Figura 47. Diagrama de flujo del fichero *Dijkstra.*

Fuente. Propia.

En un robot móvil, el planificador se utiliza con el piloto para determinar el tipo de comportamiento a seguir en la ruta más eficiente hacia el objetivo.

5.2.5. Piloto

En un robot móvil, el piloto es el componente que gestiona la ejecución de los comportamientos necesarios para que el robot alcance su objetivo de manera eficiente y segura. La misión del piloto incluye:

- **Implementación de comportamientos**: ejecución de maniobras específicas como giros, desplazamientos y evitación de obstáculos.

- **Toma de decisiones**: el piloto toma decisiones durante la marcha del robot, como desatascarlo de un mínimo local (una posición en la que el robot podría quedar atrapado).

En esta arquitectura, el piloto no incluye la toma de decisiones durante la marcha del robot. Su función se limita a gestionar los comportamientos según el plan establecido. En este caso, el plan viene definido por la matriz *m_tipos*, una matriz de dimensiones n x n donde n es el número de nodos que hay en el entorno. La matriz contiene el tipo de comportamiento a seguir entre nodo y nodo. En la Tabla 4 se muestra un ejemplo de la matriz *m_tipos.*

	NODO 1	NODO 2	NODO 3	NODO 4
NODO 1	Tipo de comportamiento a seguir del nodo **1 al 1**	Tipo de comportamiento a seguir del nodo **1 al 2**	Tipo de comportamiento a seguir del nodo **1 al 3**	Tipo de comportamiento a seguir del nodo **1 al 4**
NODO 2	Tipo de comportamiento a seguir del nodo **2 al 1**	Tipo de comportamiento a seguir del nodo **2 al 2**	Tipo de comportamiento a seguir del nodo **2 al 3**	Tipo de comportamiento a seguir del nodo **2 al 4**
NODO 3	Tipo de comportamiento a seguir del nodo **3 al 1**	Tipo de comportamiento a seguir del nodo **3 al 2**	Tipo de comportamiento a seguir del nodo **3 al 3**	Tipo de comportamiento a seguir del nodo **3 al 4**
NODO 4	Tipo de comportamiento a seguir del nodo **4 al 1**	Tipo de comportamiento a seguir del nodo **4 al 2**	Tipo de comportamiento a seguir del nodo **4 al 3**	Tipo de comportamiento a seguir del nodo **4 al 4**

Tabla 4. Ejemplo de la matriz de tipos de comportamientos.

Fuente. Propia.

Al igual que la matriz de costes, apartado 5.2.4 :

- Si la ejecución de un comportamiento entre nodo y nodo no es posible, se le asigna un valor infinito a la celda.
- Si el nodo es el mismo (por ejemplo, nodo 1-1) el comportamiento es 0, que indica que el robot no tiene que hacer nada, ya que se encuentra en el nodo a seguir.

En esta arquitectura, los tipos de comportamiento tienen un valor numérico. Para hacer que el robot ejecute un comportamiento u otro,

en la matriz *m_tipos* hay que poner el número correspondiente al comportamiento que se quiera realizar. En la Tabla 5, se muestran los números correspondientes al tipo de comportamiento.

	COMPORTAMIENTOS					
	OBJETIVO	ATRAVESAR PUERTAS	SEGUIR PARED DERECHA	SEGUIR PARED IZQUIERDA	SEGUIR PASILLO	APARCAR
NÚMERO	1	2	3	4	5	6

Tabla 5. Tipos de comportamientos y sus números en *m_tipos.*
Fuente. Propia.

Para este proyecto, la implementación del piloto se ha conseguido de la siguiente forma:

1. Se crea un entorno con sus nodos y la matriz de costes asociada a ellos
2. Se ejecuta el algoritmo de Dijkstra para saber cuál es el camino más corto.
3. Se extrae la posición de los nodos a seguir en el entorno.
4. Se comprueba en la matriz de tipos cual es el tipo de comportamiento a seguir entre nodo y nodo elegidos.
5. Se ejecuta cada comportamiento por individual hasta llegar al siguiente punto. Por ejemplo, si el comportamiento a seguir de 1-3 es el de ir a un objetivo, cuando el robot llega a 3 para y activa el siguiente comportamiento del camino a seguir.

Además, hay que tener en cuenta que después de atravesar puertas, el robot gira 90º hacia arriba para situarse en el pasillo. Por último, el comportamiento de evitar obstáculos está presente para la función de atravesar puertas.

En la Figura 48 se muestra un diagrama de flujo de la función piloto para esta arquitectura.

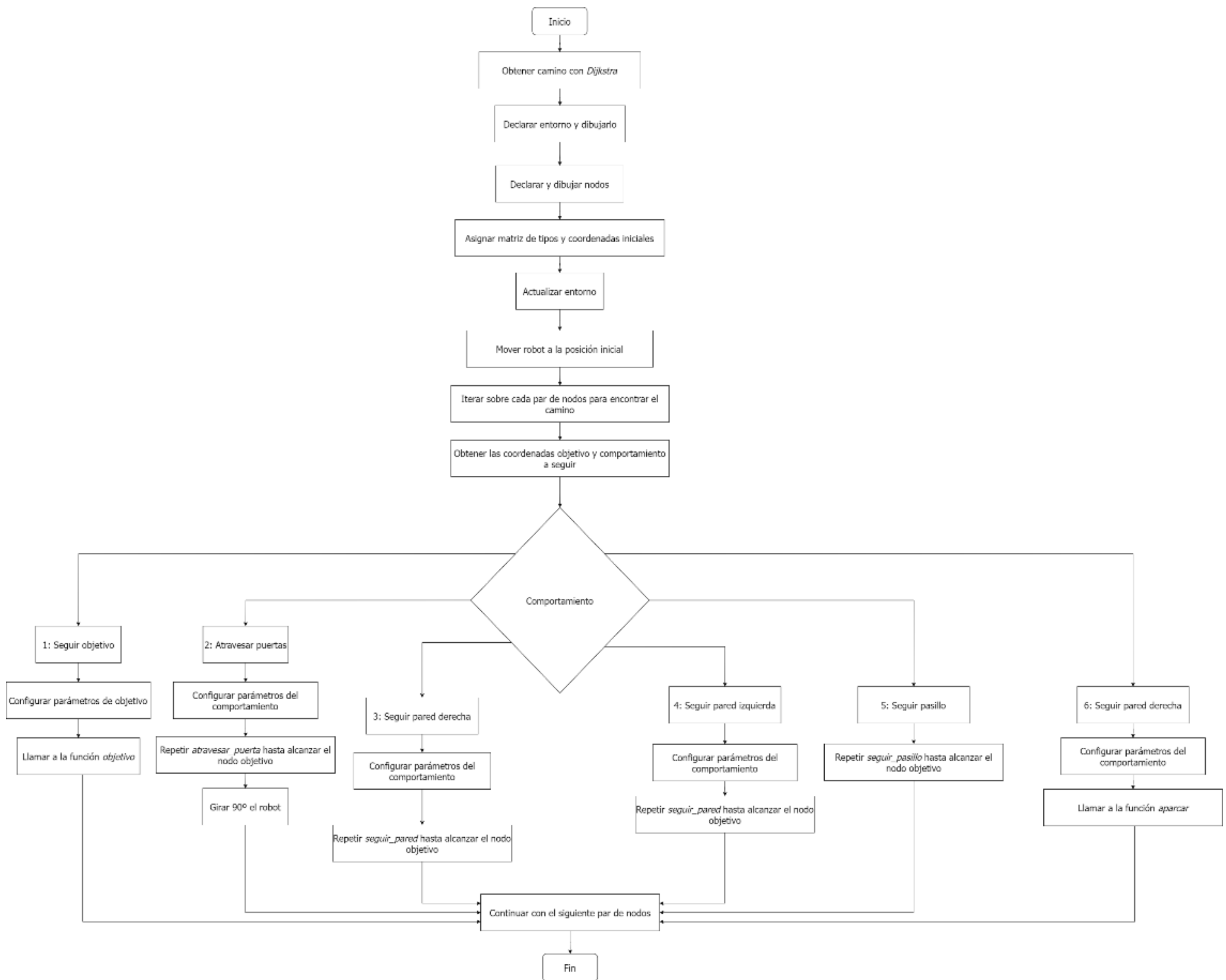


Figura 48. Diagrama de flujo de la función *piloto.*

Fuente. Propia.

6. Resultados

En este apartado se mostrarán las ejecuciones de cada comportamiento, individualmente, en un entorno distinto para cada uno de ellos. Posteriormente, se presentará el entorno final con la implementación del piloto. Hay que tener en cuenta que, al comienzo de cada simulación, se limpia la figura de MATLAB para borrar simulaciones u objetos anteriores. Además, se llama a una función denominada *actualizar_entorno*, que inicializa todas las variables y parámetros de la clase robot y sensor láser. La función *actualizar_entorno* se encuentra en el Anexo 9.6.1 de las líneas 131 hasta la 150. Para poder probar las simulaciones, en el Anexo 9.5 se encuentra un manual de usuario que explica como ejecutarlas una a una.

6.1. Simulación "Objetivo"

En este apartado se mostrarán los resultados de la simulación del robot utilizando el comportamiento de ir hacia un objetivo. Para comprobar que la función *objetivo* funciona, se ha creado un entorno vacío con unas coordenadas a las que el robot tiene que llegar. La función para probar el comportamiento *objetivo* se llama *simulación_objetivo*, referenciada en el Anexo 9.6.1. de la línea 152 a la línea 185.

El entorno de esta simulación viene definido por el fichero *entorno_objetivo*, referenciado en el Anexo 9.6.7. Las coordenadas del objetivo, en este caso, se definen en [15, 35] que representan las coordenadas *x* e *y*, respectivamente. Por otra parte, la postura inicial del robot se define en [-15, 5, pi] por lo que el robot aparecerá en -15 x, 5y y orientado horizontalmente hacia la izquierda. En la Figura 49 se muestra la situación inicial de este entorno.

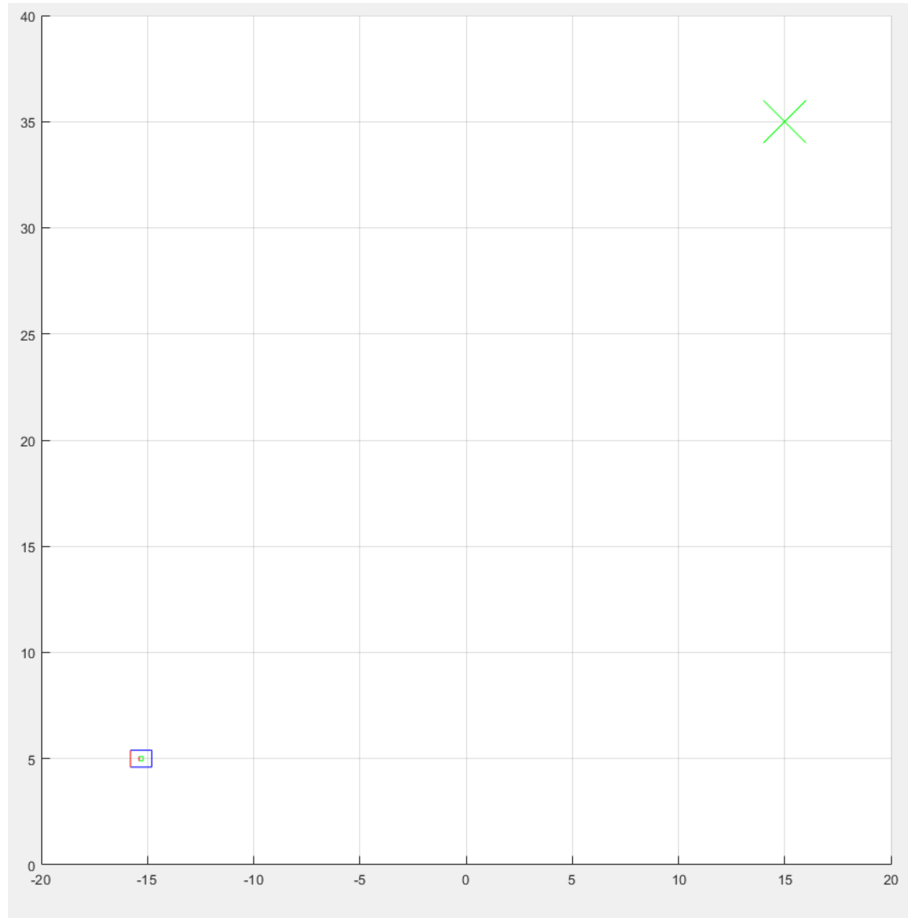


Figura 49. Situación inicial en *simulacion_objetivo.*

Fuente. Propia.

Una vez las coordenadas están dibujadas y el robot está posicionado, se llama a la función *objetivo*. Como ya se explicó en el apartado 5.2.3., este comportamiento se ejecuta mientras la distancia hacia el objetivo es mayor que la distancia a la que se tiene que parar. En la siguiente figura, la Figura 50, se puede observar el diagrama de flujo del código de esta simulación.

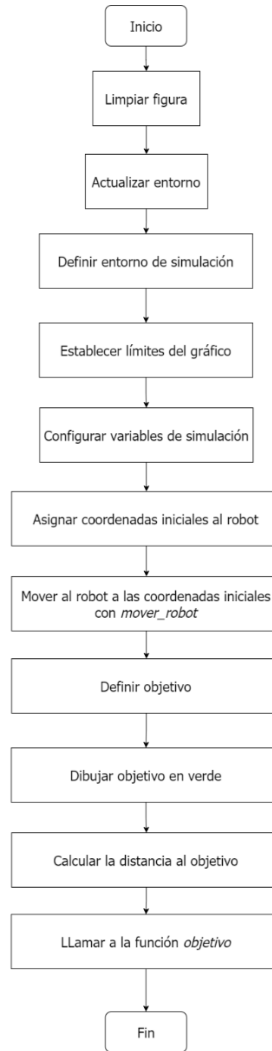


Figura 50. Diagrama de flujo de *simulacion_objetivo.*
Fuente. Propia.

En la siguiente figura se muestran algunos pasos de la simulación del robot, teniendo en cuenta que la trayectoria del robot se representa en azul.

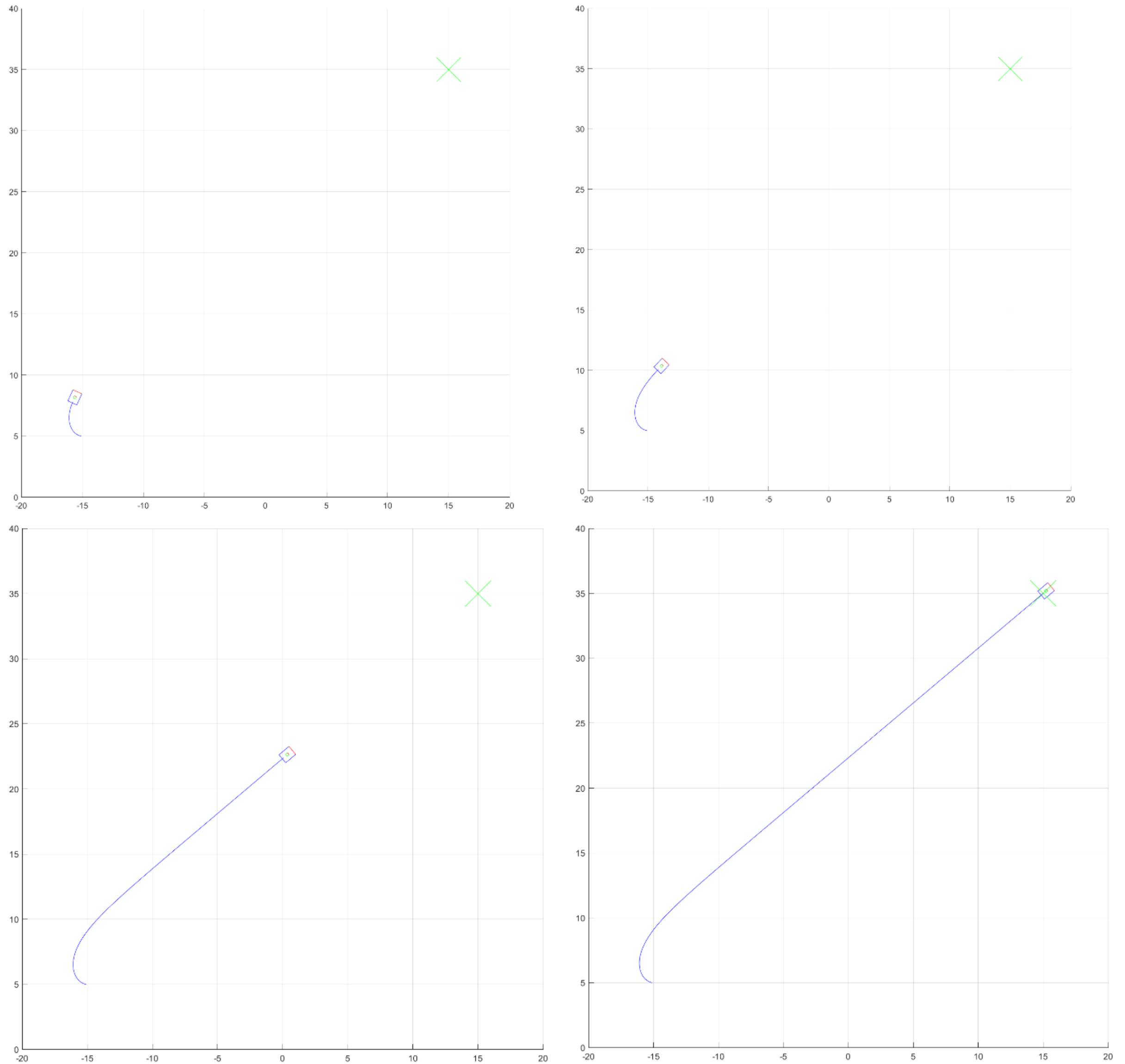


Figura 51. Resultado de la simulación del comportamiento *objetivo.*

Fuente. Propia.

Si se prueba con otras coordenadas como por ejemplo [15, 25, π] para el robot y [-15, 5] para el objetivo, el resultado es el siguiente:

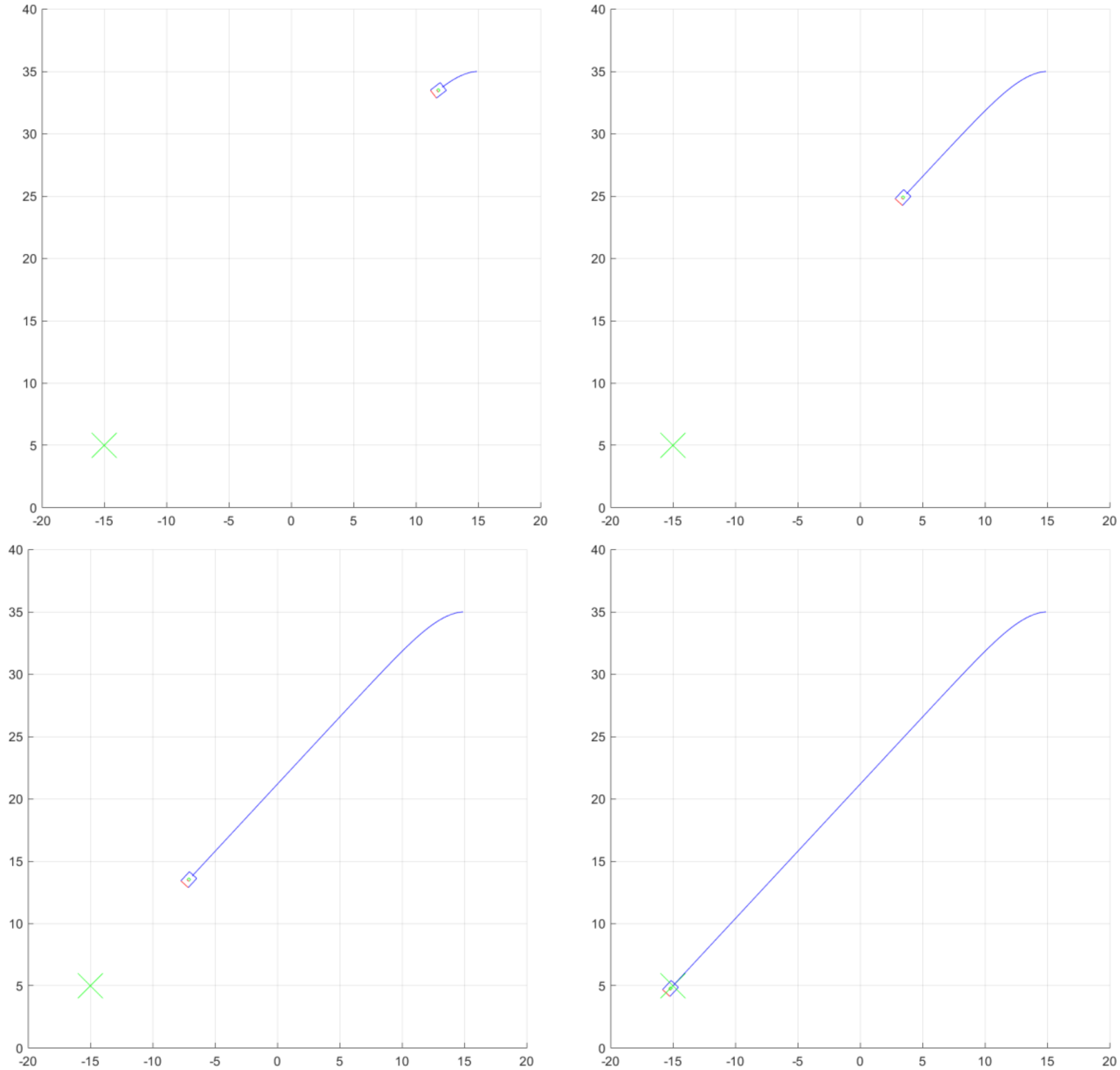


Figura 52. Resultado de otra simulación del comportamiento *objetivo.*
Fuente. Propia.

Así pues, se puede verificar que el comportamiento de ir a hacia un objetivo funciona adecuadamente. Para cambiar la posición del robot o las coordenadas y probar otras simulaciones, consultar el manual de usuario, Anexo 9.5.

6.2. Simulación “Evitar Obstáculos”

En este apartado se mostrarán los resultados de la simulación del robot utilizando el comportamiento de evitación de obstáculos. Para comprobar que la función *evita_obstaculos* funciona, se ha creado un entorno con varios

obstáculos que el robot tiene que evitar. Además, con el fin de que demostrar que funciona, al comportamiento de evitación de obstáculos se la ha añadido el comportamiento de ir hacia un objetivo. De esta manera, se podrá ver que el robot se dirige a un objetivo esquivando los obstáculos que hay en el camino. La función para probar esta simulación se llama *simulación_evitar_obstaculos* referenciada en el Anexo 9.6.1. de la línea 222 hasta la 289.

El entorno de esta simulación viene definido por el fichero *entorno_evita_obstaculos*, referenciado en el Anexo 9.6.7. Las coordenadas del objetivo y las coordenadas del robot inicialmente son las mismas que para la simulación de ir hacia un objetivo, apartado 6.1. , exceptuando la postura del robot que empieza verticalmente. En la Figura 53 se muestra la situación inicial de este entorno con los obstáculos, el objetivo marcado y el robot.

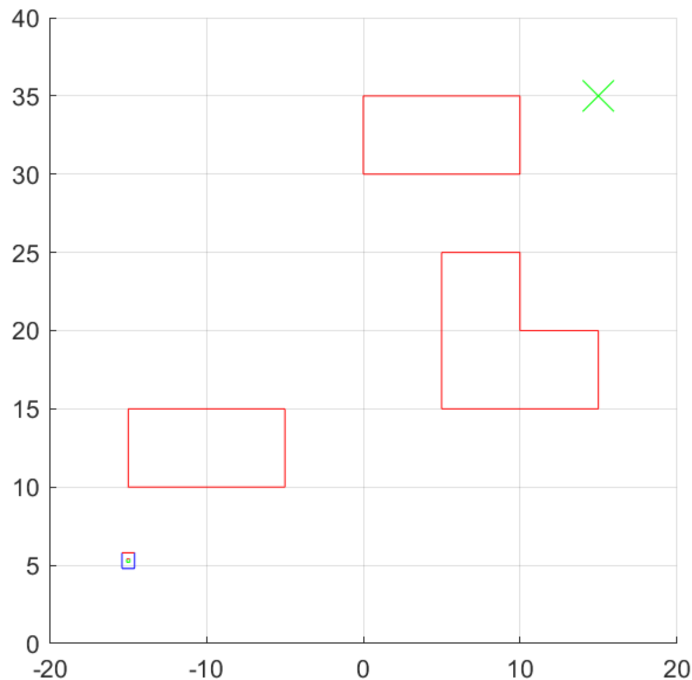


Figura 53. Entorno de la simulación del comportamiento de evitación de obstáculos
Fuente. Propia.

Después de dibujar todos los elementos del entorno, se llama a la función que calcula el vector de evitación de los obstáculos, al vector que calcula la atracción hacia el objetivo y se calcula el vector

resultante como la suma de los dos comportamientos con sus respectivos pesos. En la Figura 54, se encuentra el diagrama de flujo de esta simulación.

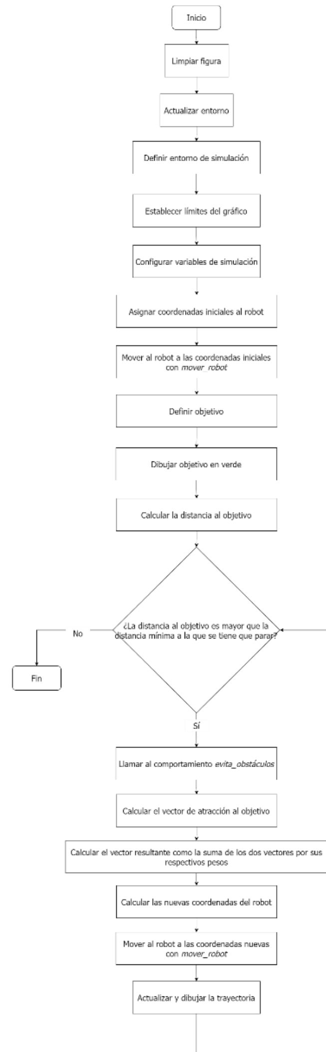


Figura 54. Diagrama de flujo de la simulación de evitar obstáculos.
Fuente. Propia.

Por otra parte, en la siguiente Figura 55 se puede ver los resultados de esta simulación paso a paso, teniendo en cuenta que la línea azul es la trayectoria del robot.

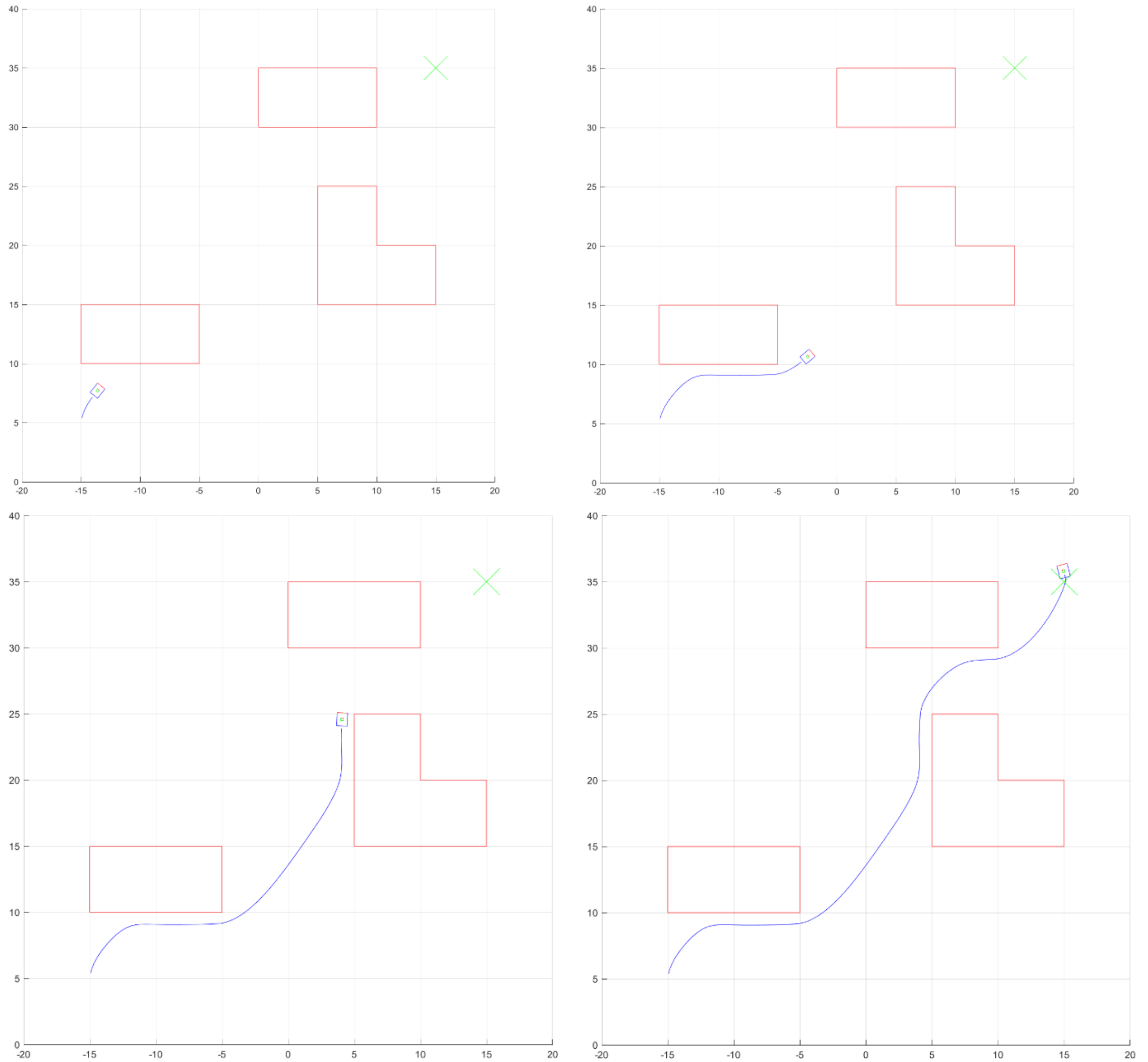


Figura 55. Resultados de la simulación del comportamiento *evitar_obstaculos.*

Fuente. Propia.

Si se prueba una segunda simulación cambiando las coordenadas del robot a [15, 35, π/2] y las del objetivo a [10, 5], el resultado es el siguiente:

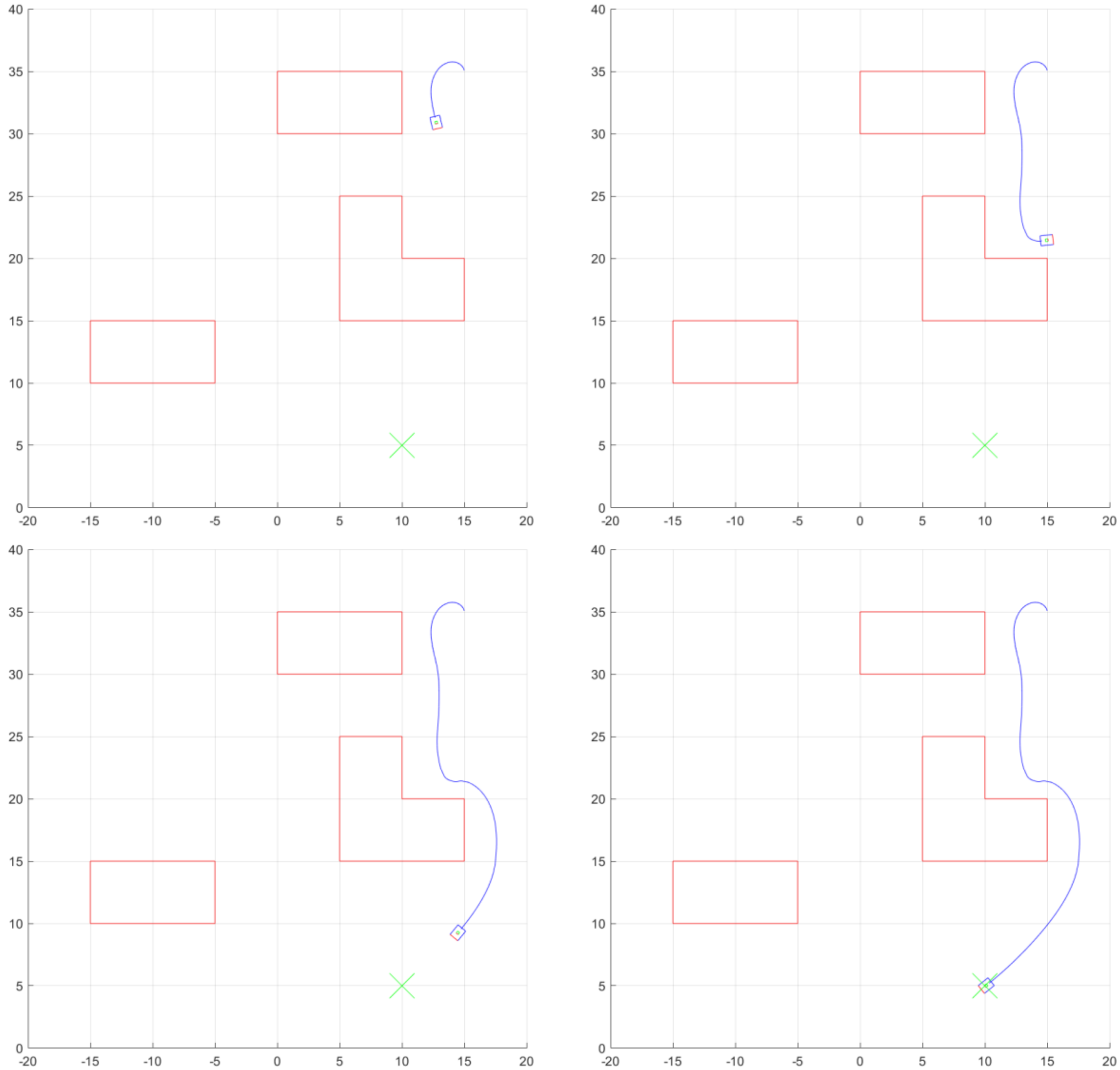


Figura 56. Resultado de otra simulación del comportamiento *evita_obstáculos*
Fuente. Propia.

Por tanto, se puede decir que el comportamiento de evitación de obstáculos del robot funciona correctamente.

6.3. Simulación "Seguir pared"

En este apartado se mostrarán los resultados de la simulación del robot utilizando el comportamiento de seguir las paredes. Para comprobar que la función *seguir_pared* funciona, se ha creado un entorno con dos paredes. La función para probar esta simulación se llama *simulación_seguir_pared* referenciada en el Anexo 9.6.1. de la línea 317 hasta la línea 349.

El entorno de esta simulación viene definido por el fichero *entorno_seguir_pared,* referenciado en el Anexo 9.6.7. Las coordenadas del robot pueden variar dependiendo de a que pared se quiera seguir. Si por ejemplo el robot se encuentra muy alejado, más de 10 metros, de la pared derecha y se pretende seguirla a una distancia de referencia de 1, tardará más en orientarse que si está cerca de ella. Por eso mismo, para comprobar el funcionamiento de ambos lados, se posiciona el robot en el centro del pasillo. En la Figura 57 se muestra la situación inicial del robot con las paredes paralelas a él.

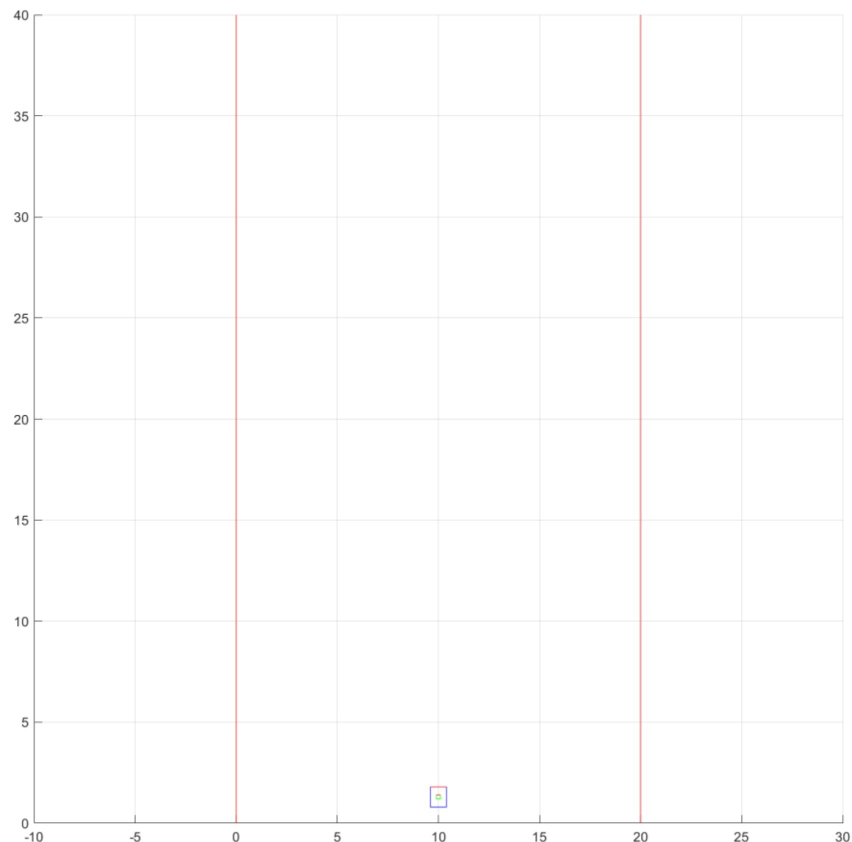


Figura 57. Entorno de la simulación del comportamiento seguir pared.

Fuente. Propia.

Después de dibujar todos los elementos del entorno, se elige el lado y la distancia de referencia a la que se tiene que poner el robot y se

llama a la función *seguir_pared*. La simulación para cuando el robot haya llegado a lo alto del pasillo, en este caso, la coordenada Y 35 del entorno. En la Figura 58, se encuentra el diagrama de flujo de esta simulación.

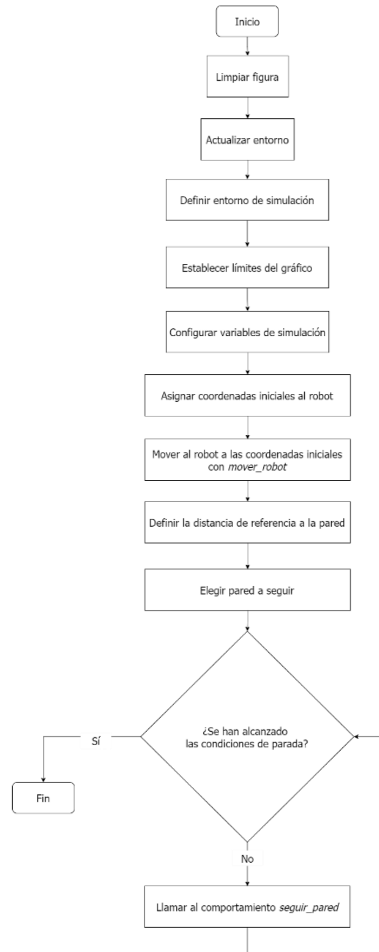


Figura 58. Diagrama de flujo de la simulación de seguir la pared.
Fuente. Propia.

Para comprobar que funciona bien de un lado de la pared y de otro, se divide este apartado en dos subsecciones para mostrar la simulación de la pared derecha y de la pared izquierda.

6.3.1. Pared derecha

La pared derecha se elige cuando la propiedad *lado* de la clase robot, mencionada en el apartado 5.2.3, es igual a 0. Para esta simulación, se ha declarado una distancia de referencia de 5 metros. En la Figura 59, se pueden ver los resultados de la simulación.

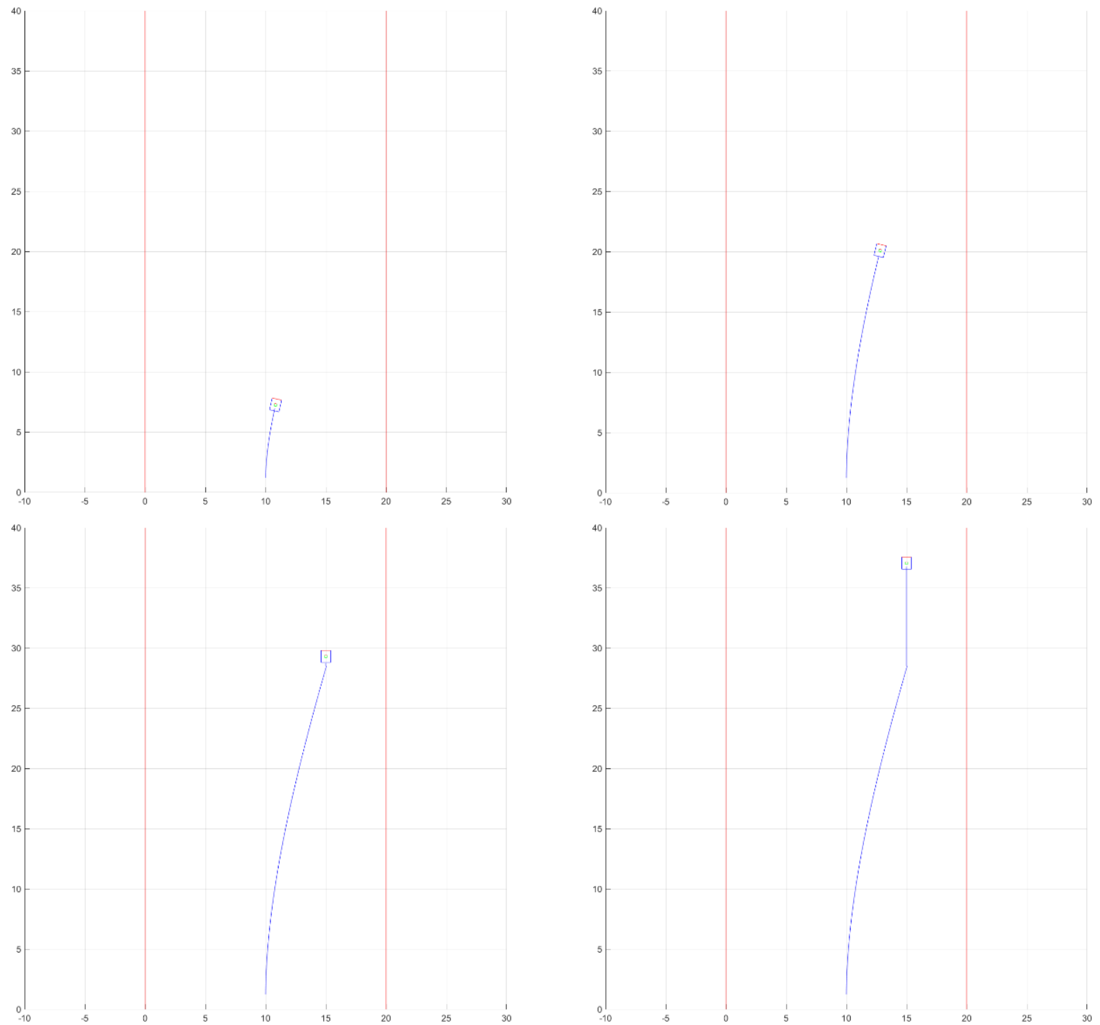


Figura 59. Resultado de la simulación *seguir_pared* con la pared derecha

Fuente. Propia.

Si se prueba con otras distancias, por ejemplo 4, el resultado es el siguiente:

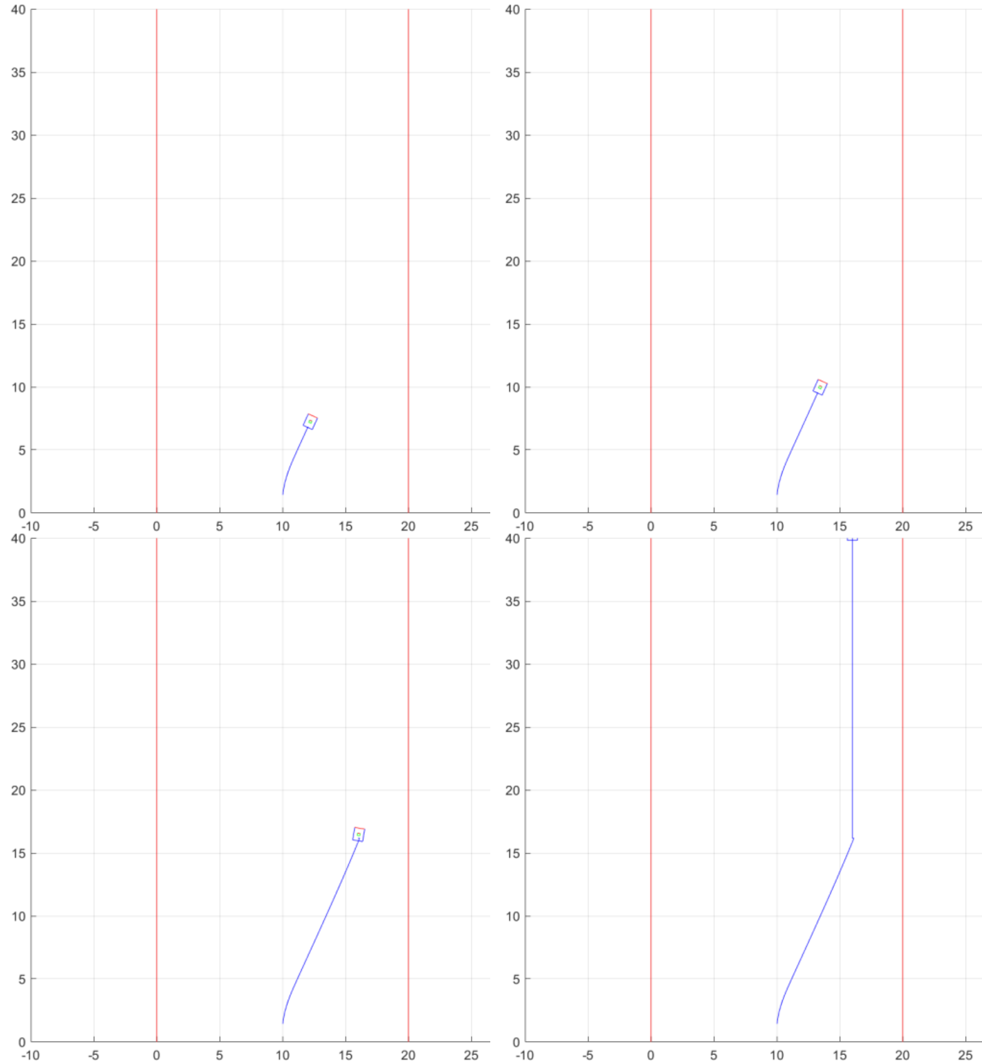


Figura 60. Resultado de otra simulación del comportamiento de seguir la pared derecha
Fuente. Propia.

6.3.2. Pared izquierda

Para probar que el robot sigue en paralelo la pared izquierda, se declara la propiedad de la clase robot con un valor igual a 0. Al igual que con la pared derecha, apartado 6.3.1, la distancia de referencia para esta simulación se declara con una distancia de 5 metros. Los resultados de la simulación se pueden ver en la Figura 61.

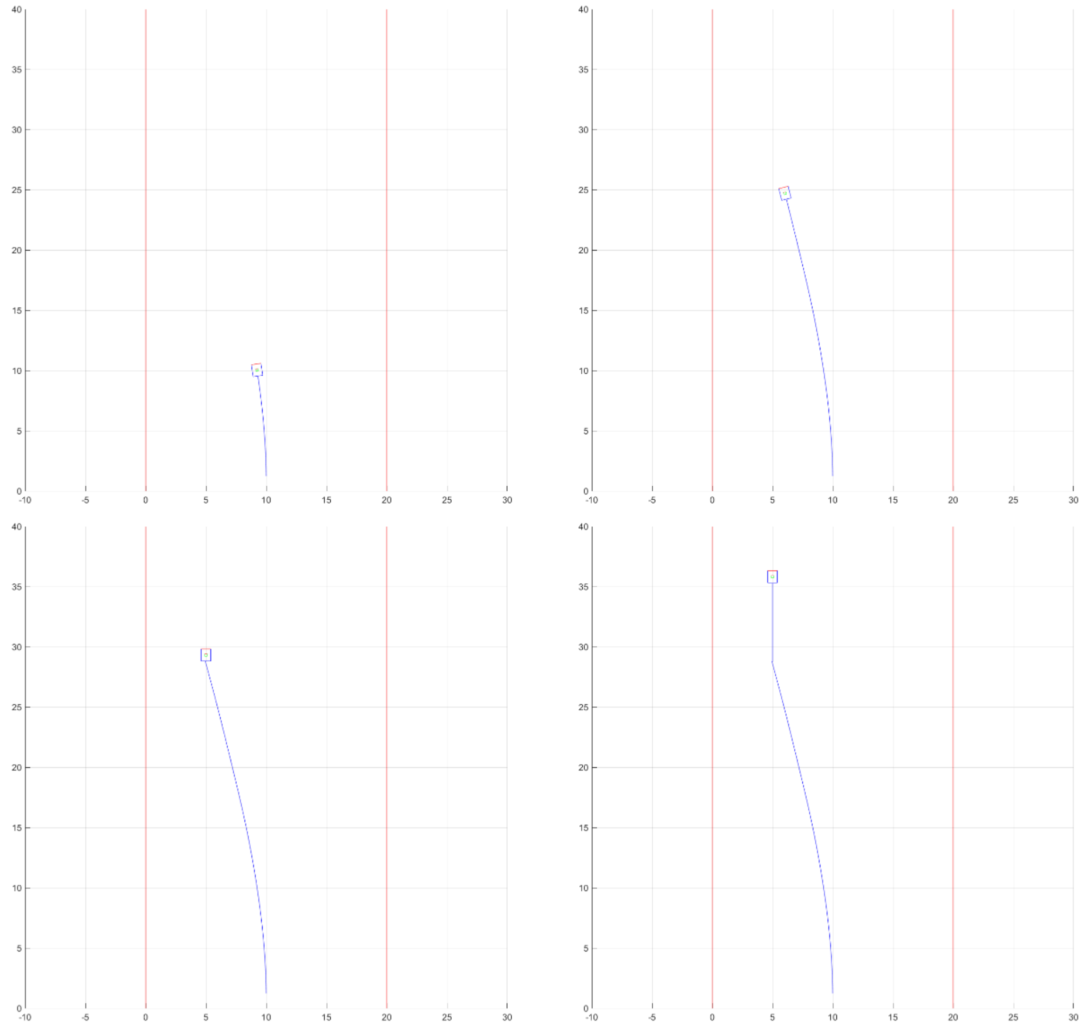


Figura 61. Resultado de la simulación *seguir_pared* con la pared izquierda.

Fuente. Propia.

Si se vuelve a simular este comportamiento, pero con otras coordenadas y otra distancia, por ejemplo, situando al robot en [5, 1, π/2] a una distancia de 3 unidades, el resultado es el siguiente:

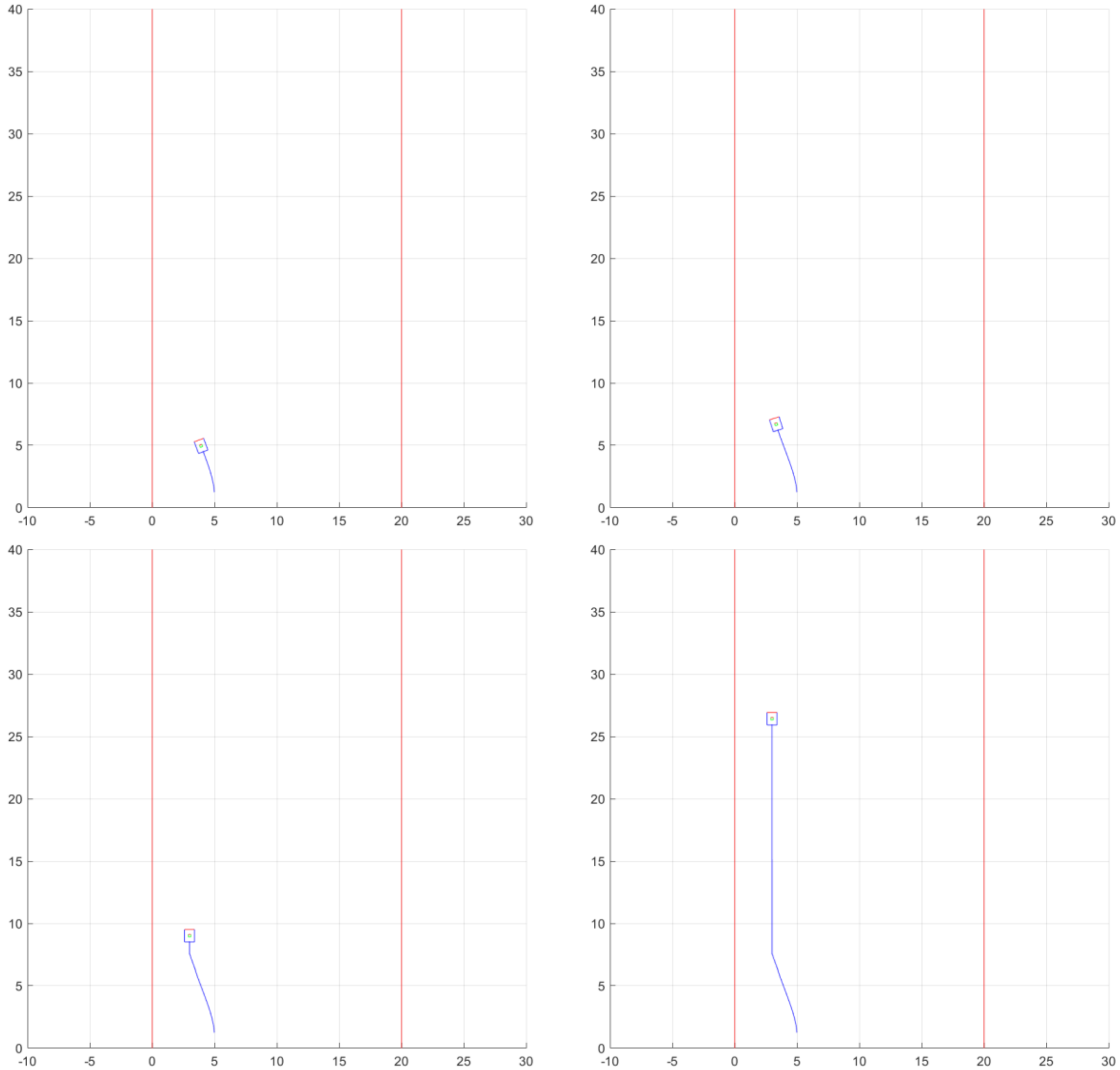


Figura 62. Resultado de otra simulación del comportamiento *seguir_pared* con la pared izquierda.
Fuente. Propia.

Teniendo en cuenta las dos simulaciones para cada uno de los lados, se puede decir que el robot sigue cumple correctamente el comportamiento de seguir una pared de un lado o de otro.

6.4. Simulación "Seguir pasillo"

En este apartado se mostrarán los resultados de la simulación del robot utilizando el comportamiento de seguir un pasillo. Para comprobar que la función *seguir_pasillo* funciona, se ha creado un entorno con un pasillo vertical y otro horizontal. La función para probar esta simulación se llama

simulación_seguir_pasillo, referenciada en el Anexo 9.6.1. de la línea 598 hasta la línea 626.

El entorno de esta simulación viene definido por el fichero *entorno_pasillo,* referenciado en el Anexo 9.6.7. En la Figura 63 se muestra la situación inicial del robot, con coordenadas [-6, 1, 0] dentro del entorno. Hay que tener en cuenta que, en esta simulación, si el robot no está orientado verticalmente hacia arriba, se fuerza a esa posición, ya que orientarlo correctamente no es el objetivo de esta función.

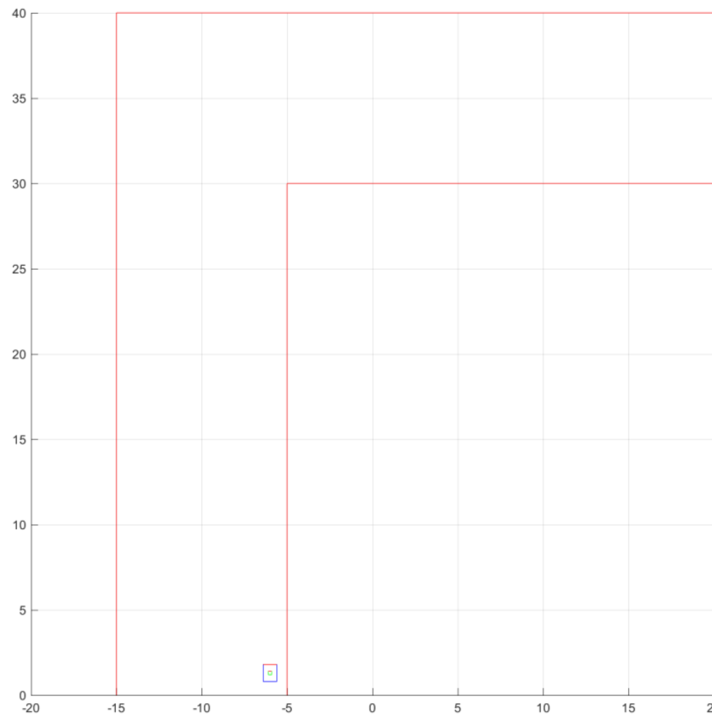


Figura 63. Entorno de la simulación del comportamiento seguir pasillo.

Fuente. Propia.

Por otra parte, en la Figura 64, se puede ver el diagrama de flujo de la función *simulación_pasillo* que se utiliza en este apartado para comprobar que el comportamiento de seguir un pasillo por el centro funciona correctamente.

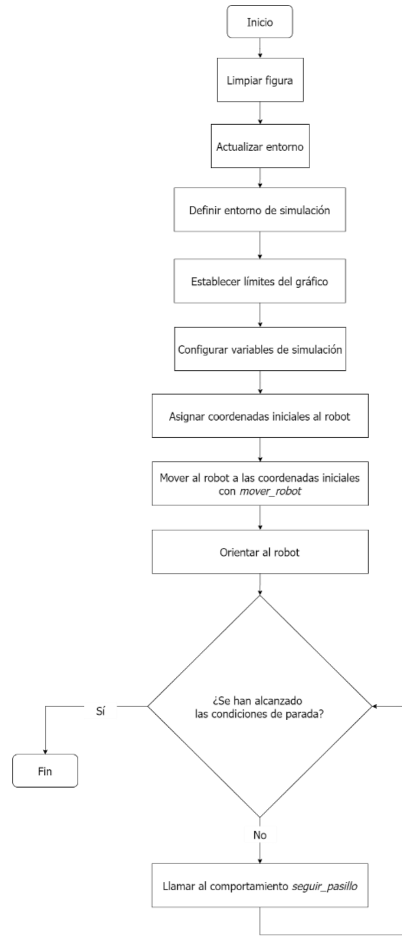


Figura 64. Diagrama de flujo de la función *simulacion_seguir_pasillo.*

Fuente. Propia.

La simulación de este comportamiento termina cuando el robot ha llegado al final del segundo pasillo. En la Figura 65 se puede ver el resultado de esta simulación con algunos pasos.

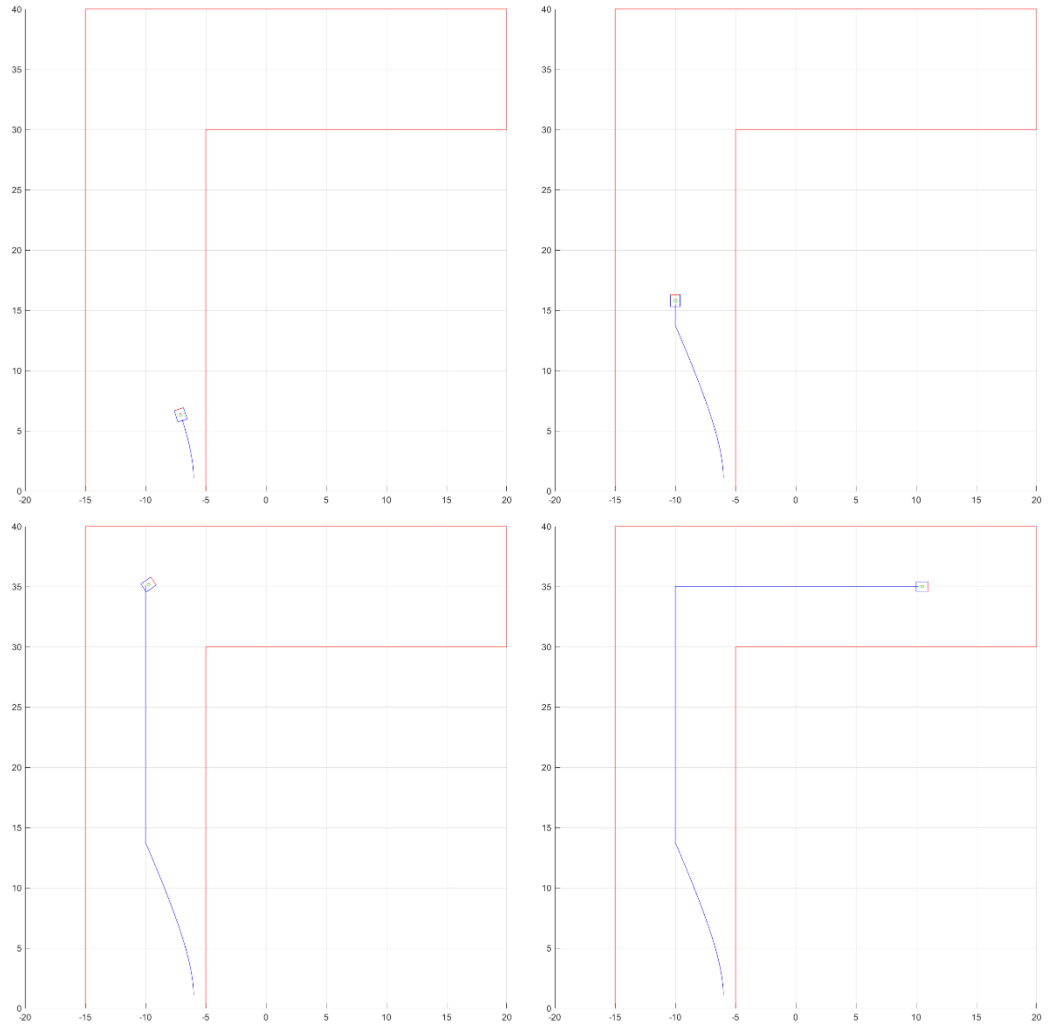


Figura 65. Resultado de la simulación *seguir_pasillo.*

Fuente. Propia.

Si se prueba a simular el comportamiento de nuevo, pero situando al robot en otras coordenadas, como por ejemplo [-12, 15, 0], el resultado es el siguiente:

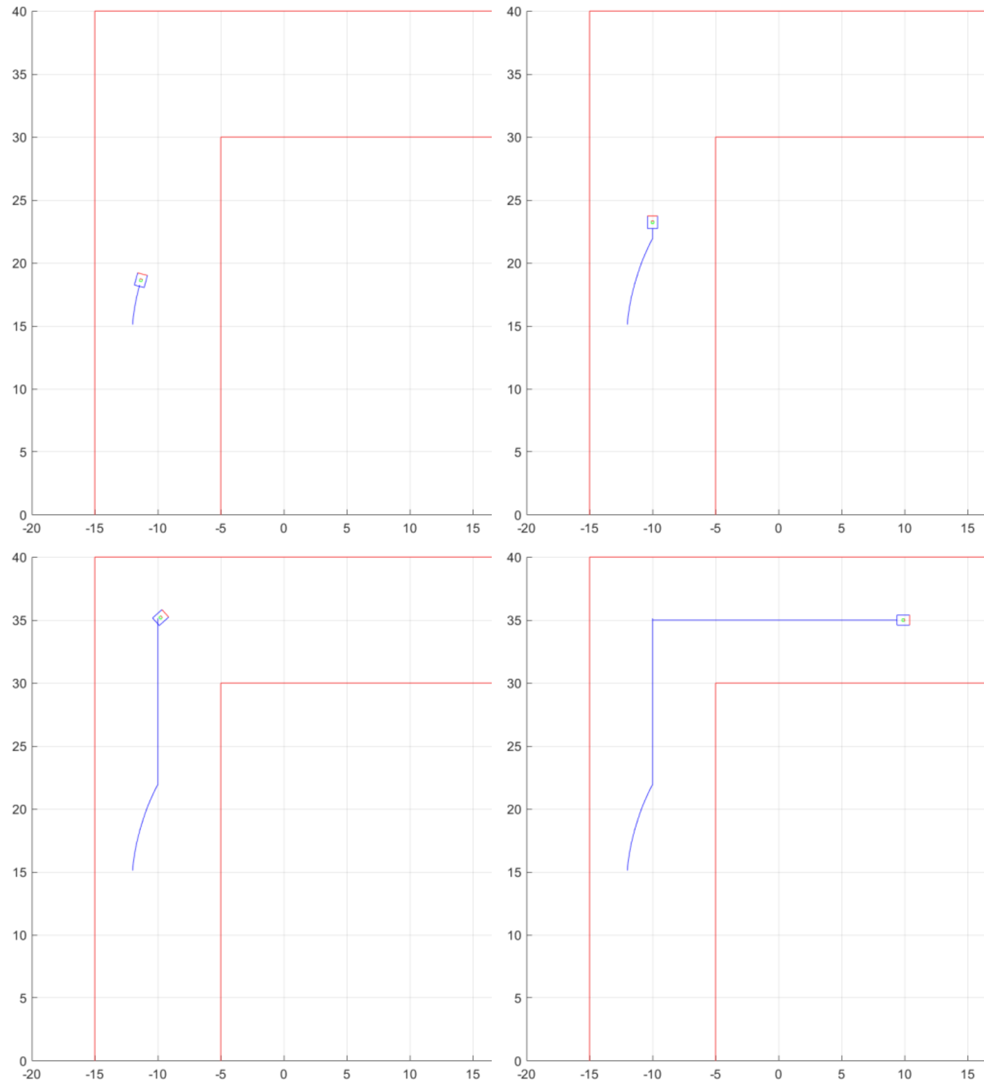


Figura 66. Resultado de la simulación del comportamiento seguir pasillo con otras coordenadas.
Fuente. Propia.

Con todo lo anterior, se puede concluir que el robot sigue correctamente el comportamiento de seguir un pasillo por el centro.

6.5. Simulación “Atravesar puerta”

En este apartado se mostrarán los resultados de la simulación del robot utilizando el comportamiento de atravesar puertas. Para comprobar que la función *atravesar_puerta* funciona, se ha creado un entorno con una habitación y un hueco en una pared que representa una puerta abierta. La

función para probar esta simulación se llama *simulación_atravesar_puertas*, referenciada en el Anexo 9.6.1. de la línea 828 hasta la línea 851.

El entorno de esta simulación viene definido por el fichero *entorno_puerta*, referenciado en el Anexo 9.6.7. En la Figura 67 se muestra la situación inicial del robot, con coordenadas [-10, 1, pi/2] dentro del entorno.

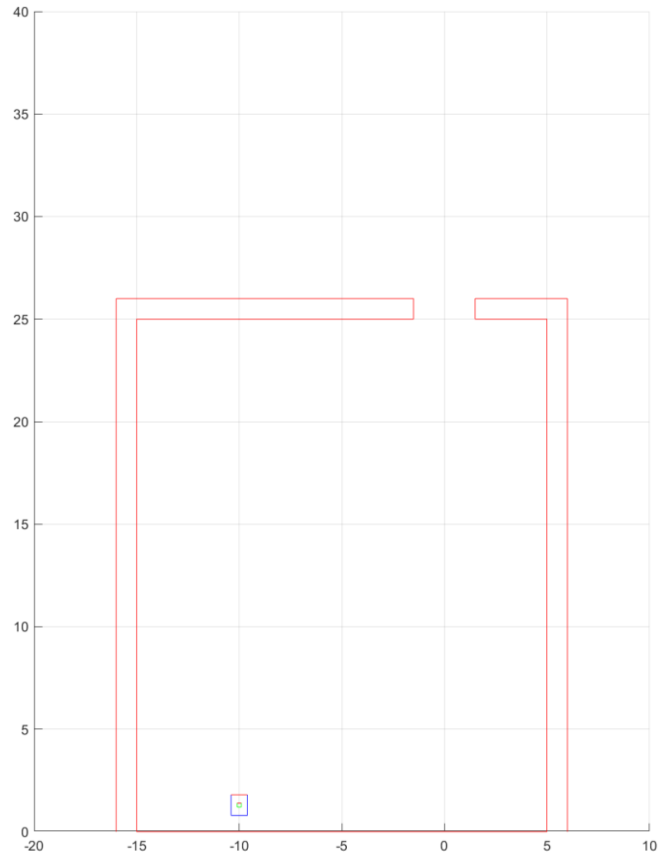


Figura 67. Entorno de la simulación *atravesar_puertas.*
Fuente. Propia.

Por otra parte, en la Figura 67, se puede ver el diagrama de flujo de la función *simulación_atravesar_puertas* que se utiliza en este apartado para comprobar que el comportamiento de salir por la puerta de una habitación funciona correctamente.

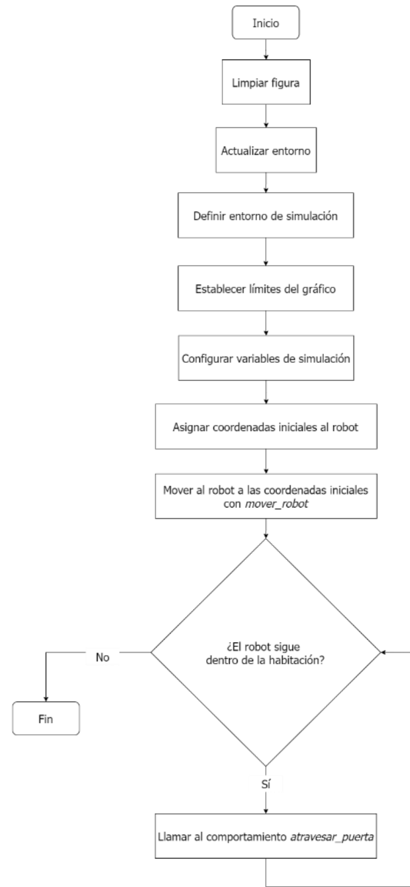


Figura 68. Diagrama de flujo de la simulación de *atravesar_puertas.*

Fuente. Propia.

La simulación de este comportamiento termina cuando el robot ha salido de la habitación. En la Figura 69 se puede ver el resultado de esta simulación con algunos pasos.

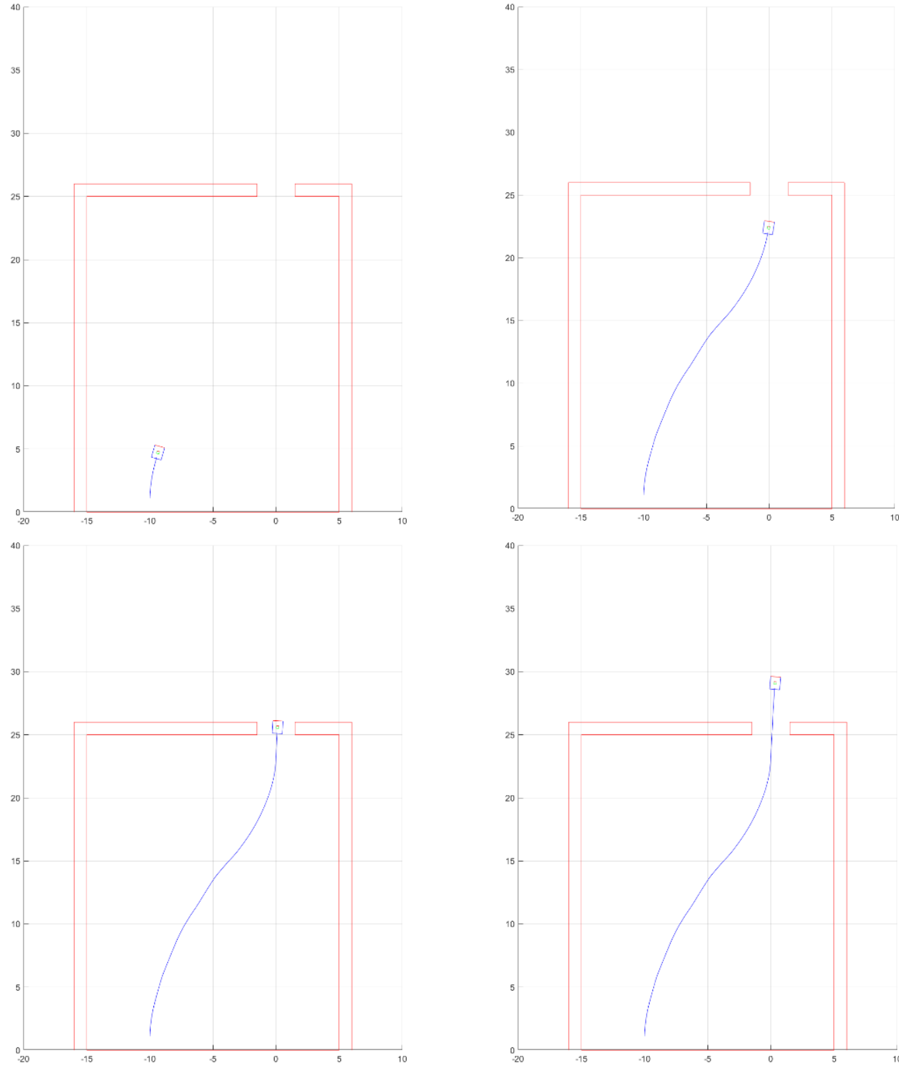


Figura 69. Resultados de la simulación *atravesar_puertas.*

Fuente. Propia.

Si se prueba a simular de nuevo el comportamiento, pero con otras coordenadas, por ejemplo [-10,20, π], el resultado es el siguiente:

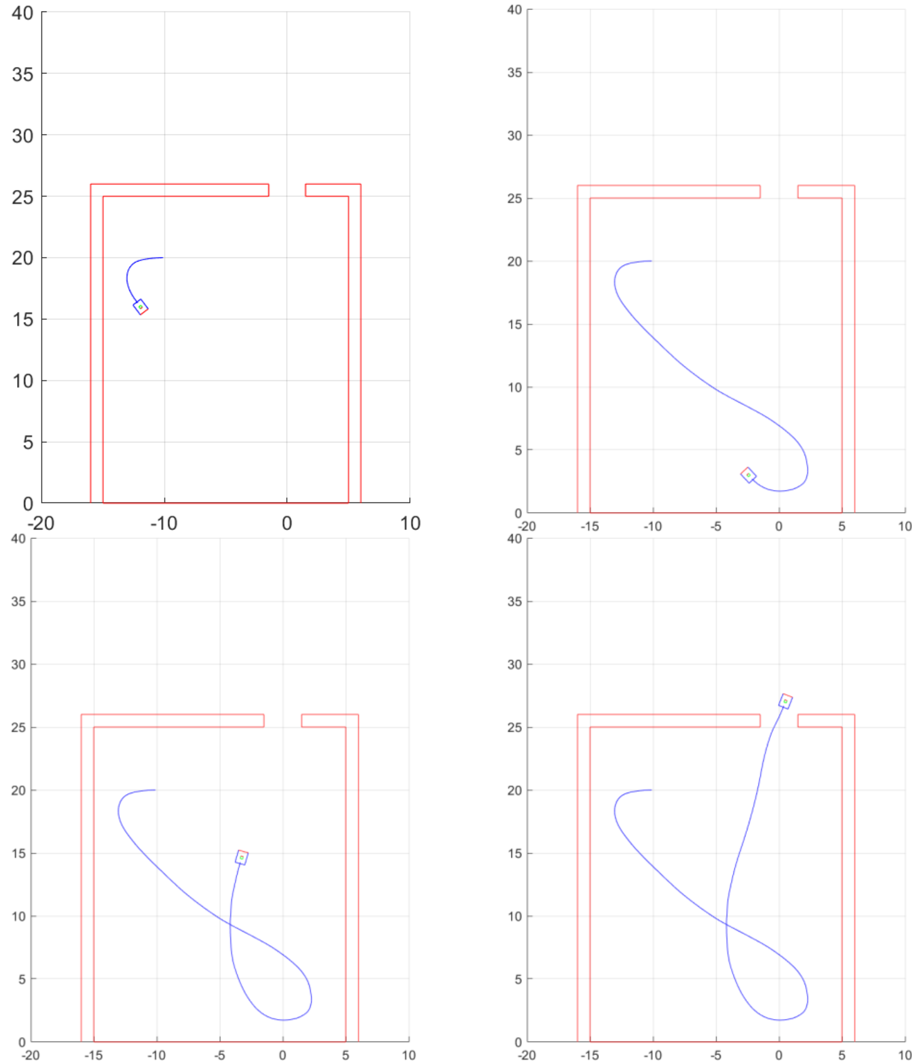


Figura 70. Resultado de la simulación del comportamiento *atravesar puertas* con otras coordenadas.

Fuente. Propia.

Aunque el robot no sigue una trayectoria directa hacia la puerta, es capaz de reorientarse y atravesarla sin problema. Con todo lo anterior, se puede concluir que el robot sigue correctamente el comportamiento de atravesar puertas.

6.6. Simulación "Aparcar"

En este apartado se mostrarán los resultados de la simulación del robot utilizando el comportamiento aparcar. Para comprobar que la función *aparcar* funciona, se ha creado un entorno con una habitación cerrada y se ha elegido

un punto aleatorio de esta habitación, en este caso, [-4, 13]. Desde este punto, se han elegido las distancias laterales y frontales para dibujar un área en la que el robot tiene que aparcar. En este caso, las distancias laterales son de 2 metros y la frontal de 1 metro. La función para probar esta simulación se llama *simulación_aparcar* Anexo 9.6.1. de la línea 922 hasta la línea 949.

El entorno de esta simulación viene definido por el fichero *entorno_aparcar*, referenciado en el Anexo 9.6.7. En la Figura 71 se muestra la situación inicial del robot, con coordenadas [-10, 1, pi/2] dentro del entorno.

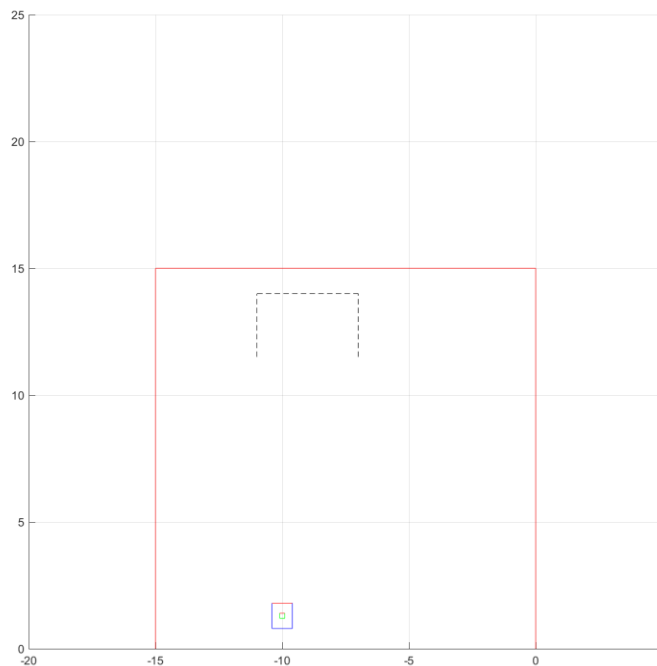


Figura 71. Entorno de la simulación aparcar.

Fuente. Propia.

Por otra parte, en la Figura 72, se puede ver el diagrama de flujo de la función *simulación_aparcar* que se utiliza en este apartado para comprobar que el comportamiento de salir por la puerta de una habitación funciona correctamente.

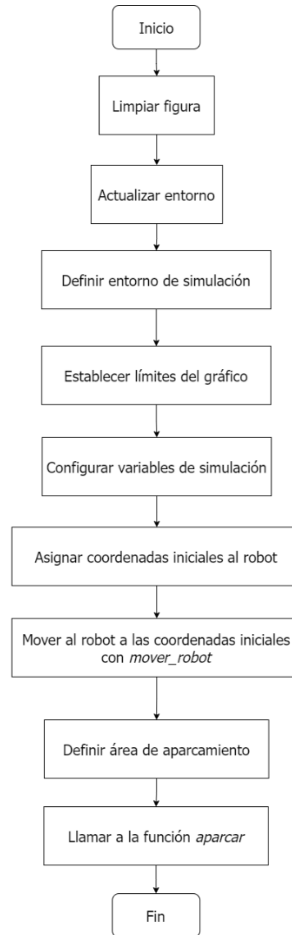


Figura 72. Diagrama de flujo de la función *simulacion_aparcar.*

Fuente. Propia.

La simulación de este comportamiento termina cuando el robot llega casi al límite da la línea frontal del área. En la Figura 73 se puede ver el resultado de esta simulación con algunos pasos.

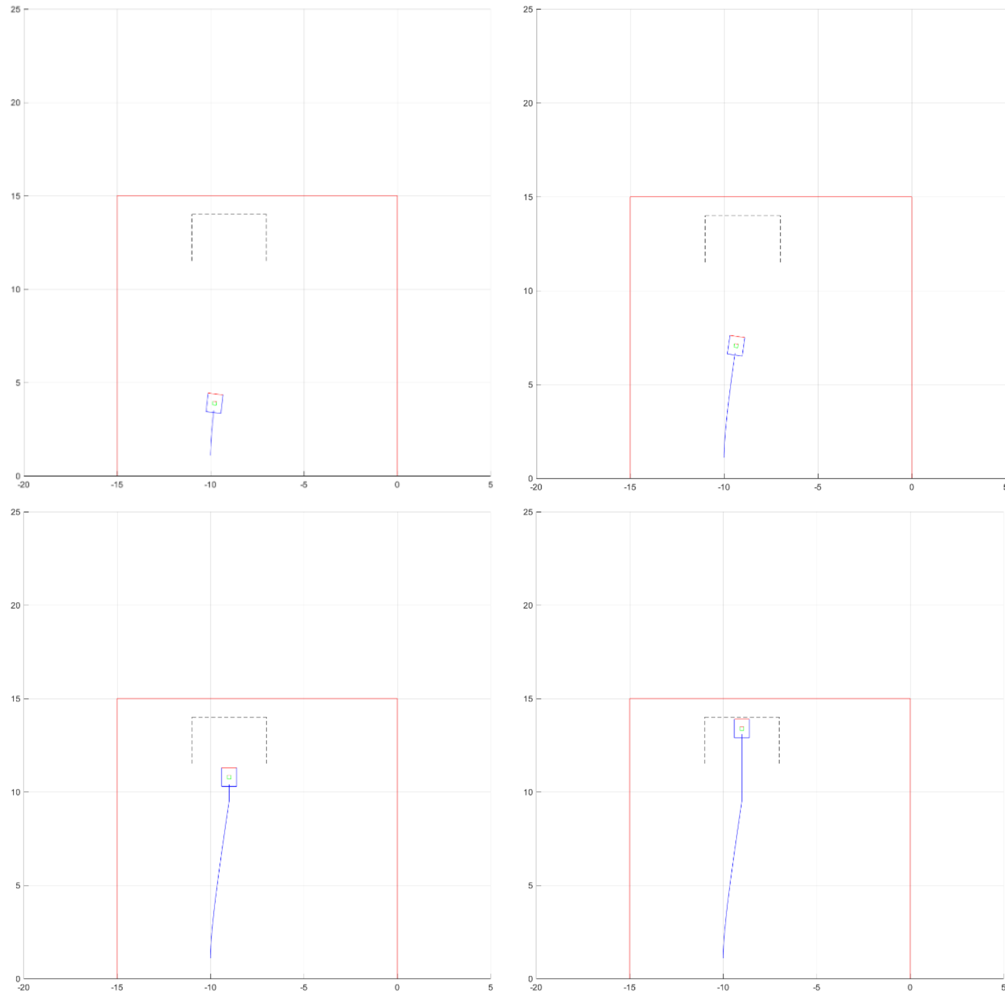


Figura 73. Resultado de la simulación de la función *aparcar.*
Fuente. Propia.

Si se vuelve a ejecutar la simulación, pero poniendo el punto de aparcamiento en [-2, 14] para que el robot aparque en una esquina, el resultado es el siguiente:

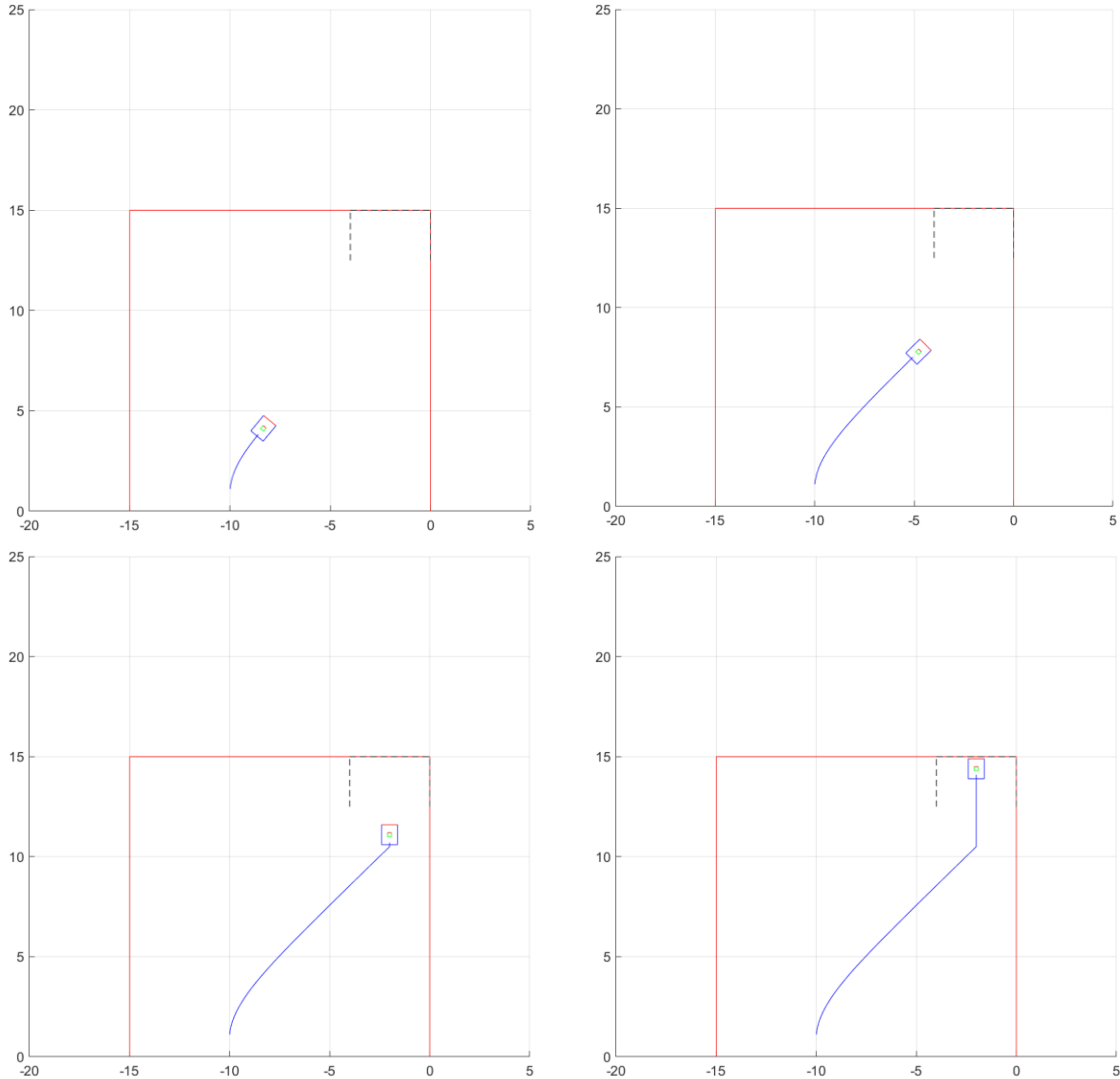


Figura 74. Resultado de la simulación del comportamiento *aparcar* con otras coordenadas.

Fuente. Propia.

Con todo lo anterior, se puede concluir que el robot sigue correctamente el comportamiento de aparcar.

6.7. Simulación "Piloto"

En este apartado se mostrarán los resultados de la simulación del robot utilizando el comportamiento piloto. El diagrama de flujo de esta función se encuentra representado en la Figura 75.

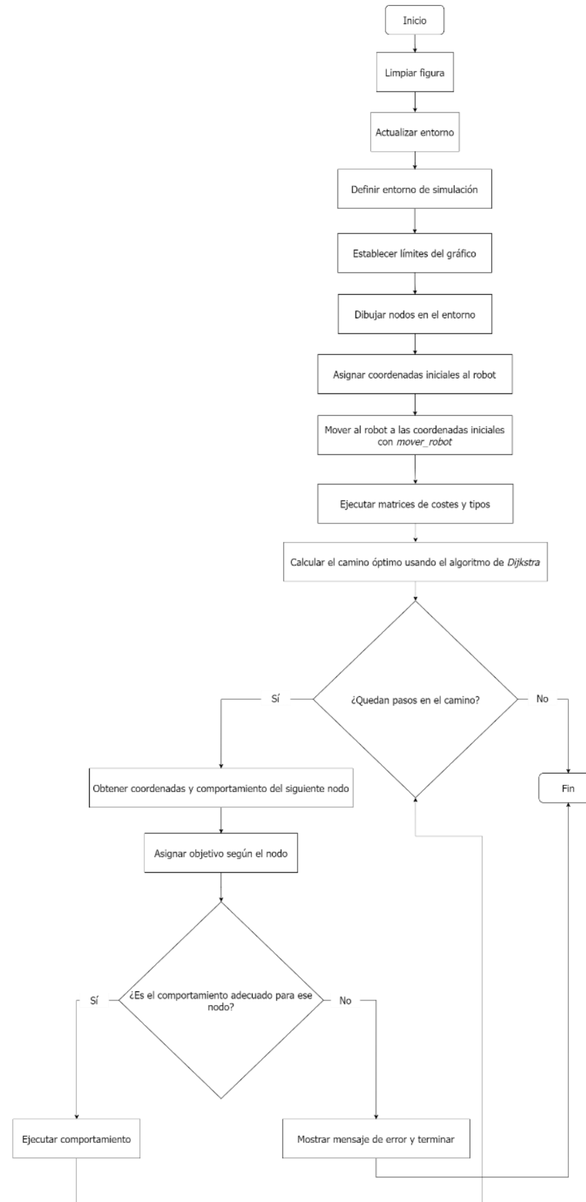


Figura 75. Diagrama de flujo del código de simulación *piloto.*

Fuente. Propia.

Para comprobar el piloto con los comportamientos, se han implementado los parámetros que se explican a continuación. Primero, el entorno de esta simulación viene definido por el fichero *entorno_final*, referenciado en el Anexo 9.6.7. El entorno de esta simulación se muestra en la Figura 76.

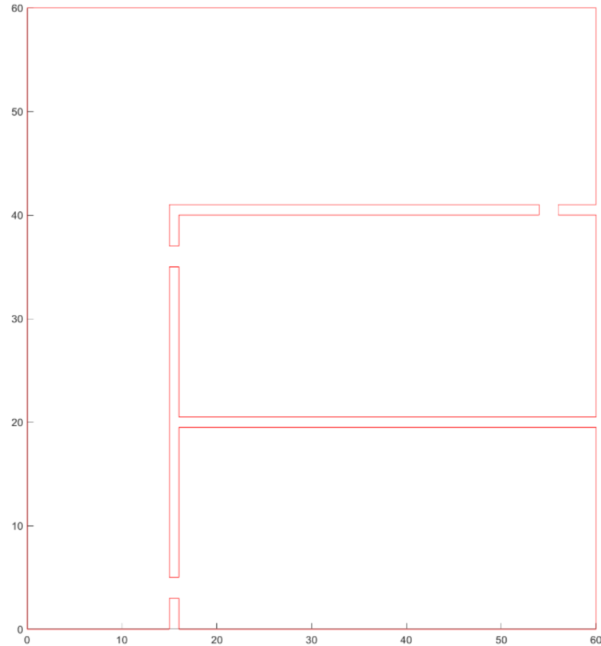


Figura 76. Entorno del piloto.

Fuente. Propia.

Por otra parte, los nodos del entorno se han definido y dibujado utilizando el fichero *Dibuja_nodos,* referenciada en el Anexo 9.6.4. Por otro lado, el número total de nodos es 9 y sus posiciones se ven reflejadas en la Tabla 6.

	Posición	
	Coordenada X	Coordenada Y
Nodo 1	50	15
Nodo 2	17	4
Nodo 3	14	4
Nodo 4	17	36
Nodo 5	13	36
Nodo 6	55	39
Nodo 7	55	42
Nodo 8	10	41
Nodo 9	50	50

Tabla 6. Posiciones de los nodos en la simulación del piloto.

Fuente. Propia.

El nodo 1 es donde se encuentra el robot inicialmente y el nodo 9 es donde tiene que ir. Una vez definidos los nodos, se extrae su posición con la propiedad *position* de un texto en MATLAB. Luego, se dibuja el robot en la posición del nodo 1. En la Figura 77 se muestra la representación del entorno con los nodos y el robot.

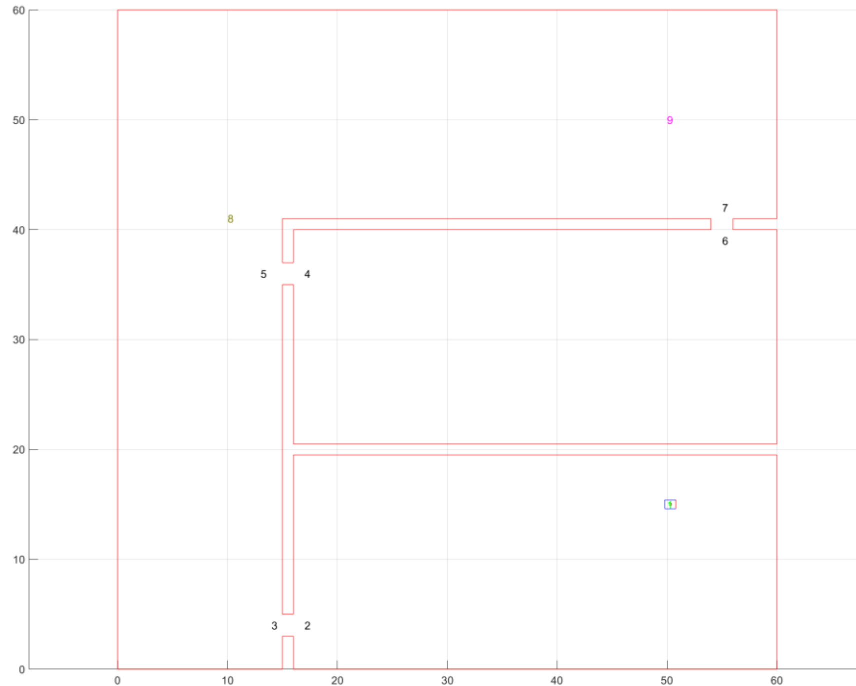


Figura 77. Entorno de la simulación piloto con los nodos y el robot.

Fuente. Propia.

Desde este punto, se ejecutan los ficheros *m_costes* y *m_tipos* que contienen la matriz de costes y la matriz de tipos de comportamiento, respectivamente, asociadas a este entorno. En la Tabla 7 se encuentra la matriz de costes y en la Tabla 8 la matriz de tipos.

	Nodo 1	Nodo 2	Nodo 3	Nodo 4	Nodo 5	Nodo 6	Nodo 7	Nodo 8	Nodo 9
Nodo 1	0	34.7851	Inf	Inf	Inf	Inf	Inf	Inf	Inf
Nodo 2	34.7851	0	4	Inf	Inf	Inf	Inf	Inf	Inf
Nodo 3	Inf	4	0	Inf	32	Inf	Inf	38.1182	Inf
Nodo 4	Inf	Inf	Inf	0	4	38.0526	Inf	Inf	Inf
Nodo 5	Inf	Inf	32	4	0	Inf	Inf	6.7082	Inf
Nodo 6	Inf	Inf	Inf	38.0526	Inf	0	4	Inf	Inf
Nodo 7	Inf	Inf	Inf	Inf	Inf	4	0	45	8.6023
Nodo 8	Inf	Inf	38.11820	Inf	6.7082	Inf	45	0	40.7922
Nodo 9	Inf	Inf	Inf	Inf	Inf	Inf	8.6023	40.7922	0

Tabla 7. Matriz de costes de la simulación de la función piloto.

Fuente. Propia.

	Nodo 1	Nodo 2	Nodo 3	Nodo 4	Nodo 5	Nodo 6	Nodo 7	Nodo 8	Nodo 9
Nodo 1	0	1	Inf	Inf	Inf	Inf	Inf	Inf	Inf
Nodo 2	1	0	2	Inf	Inf	Inf	Inf	Inf	Inf
Nodo 3	Inf	2	0	Inf	3	Inf	Inf	4	Inf
Nodo 4	Inf	Inf	Inf	0	2	1	Inf	Inf	Inf
Nodo 5	Inf	Inf	3	0	Inf	Inf	4	Inf	Inf
Nodo 6	Inf	Inf	Inf	1	Inf	0	2	Inf	Inf
Nodo 7	Inf	Inf	Inf	Inf	Inf	2	0	1	1
Nodo 8	Inf	Inf	4	Inf	3	Inf	1	0	1
Nodo 9	Inf	Inf	Inf	Inf	Inf	Inf	1	1	0

Tabla 8. Matriz de tipos de comportamiento de la simulación de la función piloto.

Fuente. Propia.

En la matriz de tipos los números representan el tipo de comportamiento a seguir, explicado en el apartado 5.2.5.

Una vez definidas las matrices, se ejecuta el algoritmo de Dijkstra con el fichero *Dijkstra* de MATLAB, referenciado en el Anexo 9.6.4. De este fichero se obtiene el camino más corto del nodo inicial al nodo final, que en este caso es 1-2-3-8-9, como se puede observar en la Figura 77. Con el camino a seguir ya marcado, se cogen el nodo inicial y el siguiente (ej: 1-2, 2-3, 3-8, 8-9) y se busca con esa combinación el tipo de comportamiento a seguir, es decir que, si por ejemplo el robot tiene que ir de 3 a 8, se busca en la fila 3, columna 8 de la matriz tipos para extraer el número del comportamiento. De ahí se hace una sentencia *switch* y se ejecuta el comportamiento dependiendo del número extraído de la matriz. En la sentencia switch que selecciona el comportamiento del robot se ajustan los parámetros de la

simulación, como por ejemplo las constantes de velocidad o la distancia de referencia a una pared. El comportamiento termina cuando el robot pasa por el nodo no visitado. Además, para asegurarse de que el robot no ejecuta un comportamiento no deseado, en la extracción del valor de las coordenadas de los nodos no visitados, si el comportamiento extraído no es el esperado, el programa para y avisa por pantalla de que el comportamiento no es el adecuado para la situación del robot.

Así pues y teniendo en cuenta que el robot tiene que el camino que tiene que seguir el robot, según el algoritmo de Dijkstra es 1-2-3-8-9 con los comportamientos objetivo-atravesar puertas-seguir pared izquierda(a 3 unidades de distancia)-objetivo, en la Figura 78 se muestra la trayectoria del robot al ejecutar el piloto.

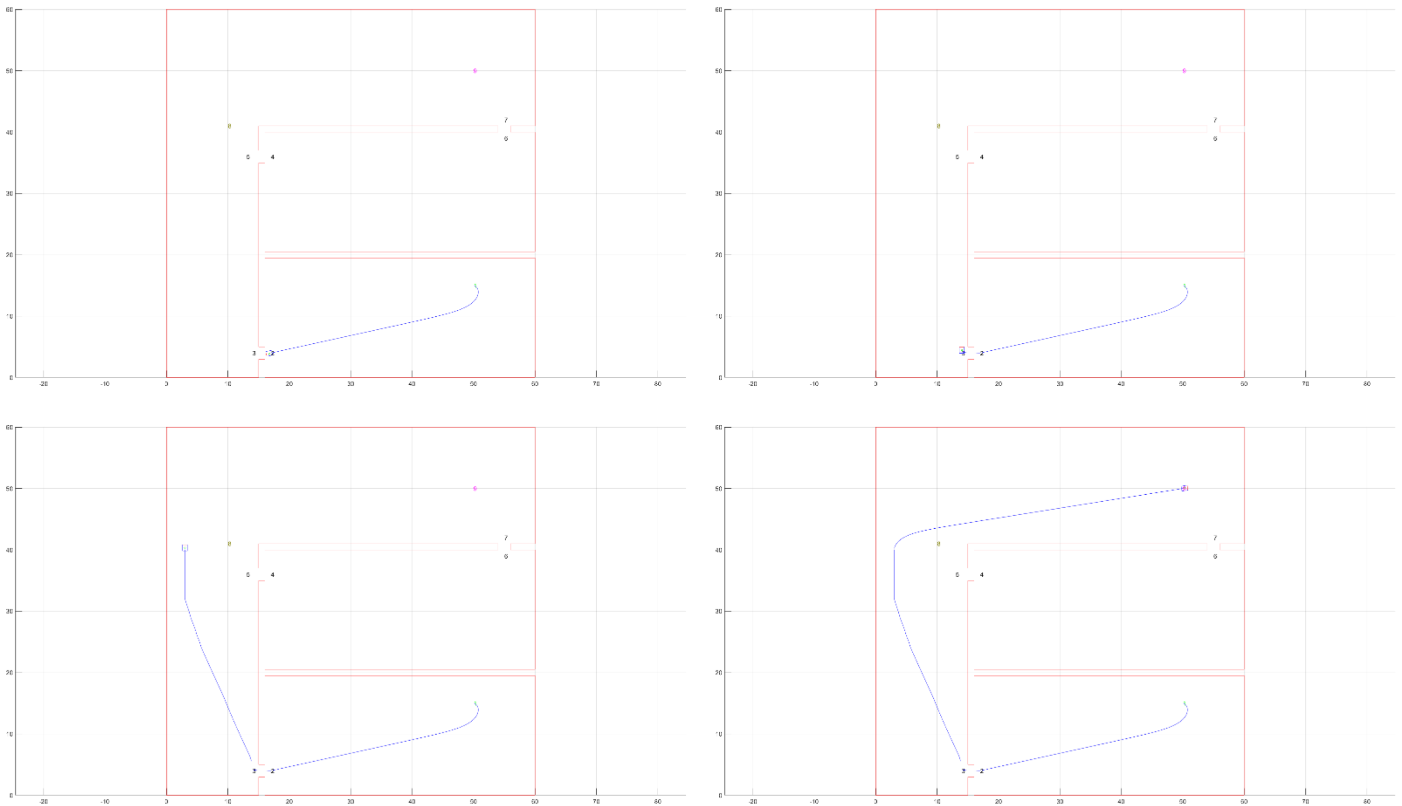


Figura 78. Resultado de la simulación de la función *piloto.*

Fuente. Propia.

Además, se pueden cambiar los comportamientos en la matriz tipos para poder probarlos todos. Por ejemplo, en la Figura 79 se muestra otro resultado de la ejecución de la función piloto pero esta vez con el comportamiento de seguir pasillo, nº5, para el camino 3-8.

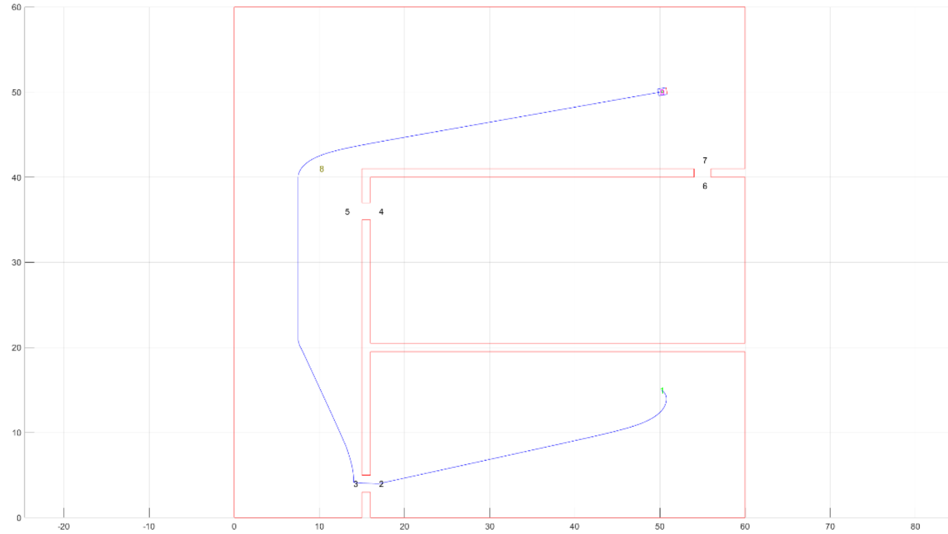


Figura 79. Simulación del piloto con otro comportamiento (seguir pasillo).

Fuente. Propia.

Otro ejemplo sería siguiendo la pared derecha, comportamiento nº3, a una distancia de 6 unidades, mostrado en la Figura 80, donde hay un pequeño error en la trayectoria debido a que el robot al pasar por la puerta no detecta pared y se desvía para encontrar mediciones en los ángulos laterales.

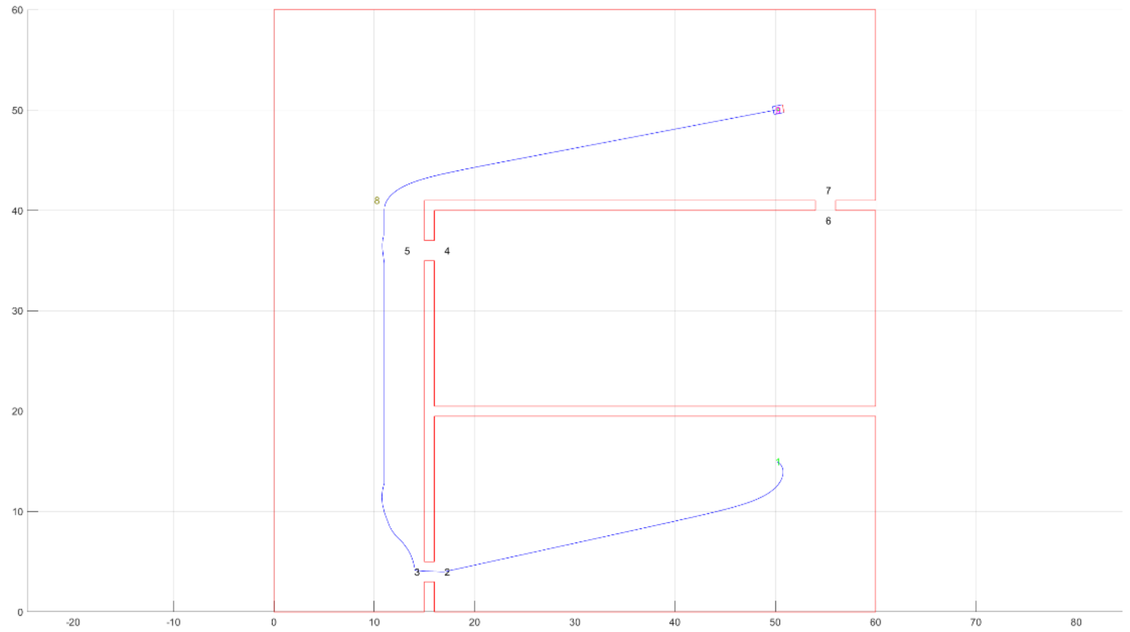


Figura 80. Simulación del piloto con otro comportamiento (seguir la pared derecha).

Fuente. Propia.

Por último, se puede probar del nodo 8 al 9 el comportamiento de aparcar, nº6. El resultado se muestra en la Figura 81.

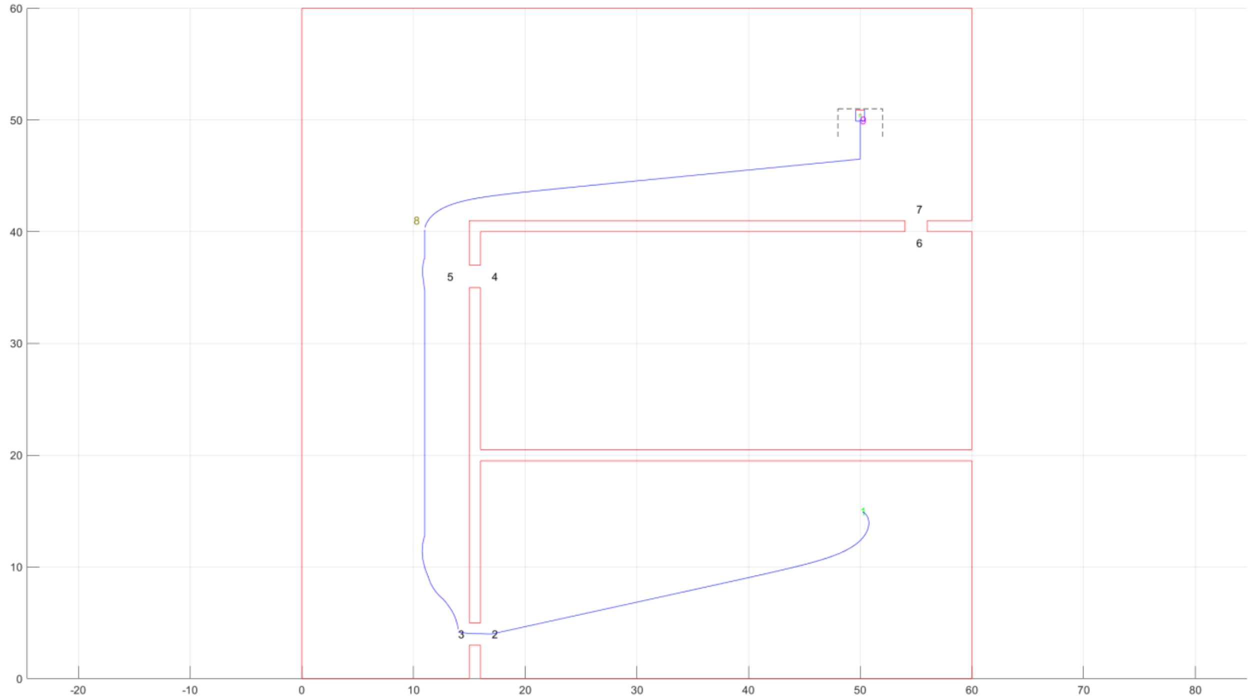


Figura 81. Resultado de la simulación piloto con otro comportamiento (aparcar).

Fuente. Propia.

A falta de comprobar el piloto en otros entornos, se puede concluir que funciona adecuadamente, ya que ejecuta el algoritmo de Dijkstra para encontrar el camino más corto, sitúa al robot en el nodo inicial y puede inicializar y terminar los comportamientos que mueven al robot.

Como se ha podido comprobar, todos los comportamientos, incluido el piloto, funcionan en los entornos de sus respectivas simulaciones. Hay que tener en cuenta que en este estudio no se han probado todas las situaciones posibles en las que el robot puede estar, ya que depende de muchos factores como las coordenadas negativas en un entorno, la posición del robot en determinados comportamientos (como por ejemplo ir hacia abajo siguiendo una pared cambiaría de lado los sensores laterales del robot) y otros factores que solo son posibles de averiguar mediante ensayo y error. Aun así, si un usuario quisiera probar alguna de estas simulaciones, tendría que consultar el manual de usuario en el Anexo 9.3 para saber que parámetros ajustar en función de lo que quiera conseguir o si se encuentra algún error en algún tipo de comportamiento.

7. Conclusiones

En este trabajo, se ha llevado a cabo el desarrollo de un robot móvil en MATLAB, destacando la importancia de los diferentes comportamientos que permiten su funcionamiento autónomo en diversos entornos. Inicialmente, se diseñó un robot equipado con un sensor láser basado en el modelo UST- 20LX (UUST004) [22] capaz de medir puntos de intersección en un rango de 270 grados, lo que proporcionó una base sólida para la percepción del entorno.

Posteriormente, se simularon con éxito varios comportamientos de manera individual: ir hacia un objetivo, evitar obstáculos, seguir paredes (tanto a la derecha como a la izquierda), seguir pasillos, atravesar puertas y aparcar. Cada uno de estos comportamientos demostró ser crucial para la autonomía y adaptabilidad del robot en situaciones específicas.

Una vez validados los comportamientos individuales, se implementó un piloto con un planificador basado en el algoritmo de Dijkstra. Este planificador permitió la integración de los comportamientos en una estrategia de navegación global, proporcionando al robot la capacidad de planificar rutas óptimas.

Las simulaciones en un entorno controlado demostraron que la combinación de estos comportamientos, junto con un planificador, resulta en un robot móvil funcional y autónomo. El robot, con su arquitectura, fue capaz de navegar de manera efectiva y seguir rutas predefinidas por el planificador en un entorno.

Como posible mejora de este proyecto en un futuro, se podría implementar un tercer eje de coordenadas y hacer el espacio 3D. También se podría probar la viabilidad de esta arquitectura en otros entornos o con otros comportamientos.

Este proyecto no ha estado exento de desafíos, ya sea por probar constantes mediante prueba y error o bien por probar la simulación del robot en los entornos teniendo en cuenta las posibles combinaciones entre sus coordenadas y su posición para que los comportamientos se ejecutasen de forma correcta.

En definitiva, la arquitectura de este robot móvil no solo proporciona una sólida base para el aprendizaje en el campo de la robótica móvil, sino que también incluye un manual de usuario, un recurso útil para estudiantes que deseen explorar y comprender tanto los aspectos teóricos como prácticos del diseño, la programación y el funcionamiento autónomo de robots en entornos 2D. Además, esta arquitectura puede servir como punto de partida para desarrollar otros tipos de robots con diferentes sensores y comportamientos, abriendo nuevas posibilidades para verificar el funcionamiento de los robots a través de simulaciones en MATLAB.

8. Bibliografía

[1] International Federation of Robotics. (2024, January 10). *Global robotics race: Korea, Singapore and Germany in the lead*. https://ifr.org/ifr-press-releases/news/global-robotics-race-korea-singapore-and-germany-in-the-lead

[2] SCM Logística. (s.f.). *Vehículos de guiado automático (AGV): tipos y características*. Recuperado de https://www.scmlogistica.es/vehiculos-de-guiado-automatico-agv-tipos-y-caracteristicas/

[3] Robots al Detalle. (s.f.). *Las tortugas robot de William Grey*. Robots al Detalle. Recuperado de: https://robotsaldetalle.es/noticias/las-tortugas-robot-de-william-grey/

[4] Polanco Masa, A. (2015, mayo 17). *Elmer y Elsie, las tortugas robot de 1948*. Tecnología Obsoleta. Recuperado de https://alpoma.net/tecob/?p=11359

[5] Sánchez-Migallón Jiménez, S. (2011, mayo 1). *Las tortugas de Grey Walter*. La Máquina de Von Neumann. Recuperado de: https://vonneumannmachine.wordpress.com/2011/05/01/las-tortugas-de-grey-walter/

[6] Kautz, H. A. (2022). *The third AI summer: AAAI Robert S. Engelmore Memorial Lecture.* AI Magazine, 43(1), 105-125. Recuperado de: https://doi.org/10.1002/aaai.12036

[7] Trilnick, C. (s.f.). *Shakey*. IDIS. Recuperado de: https://proyectoidis.org/shakey/

[8] Sanz, A. (2017, diciembre 18). Shakey: *El primer robot con inteligencia artificial, el abuelo del coche autónomo*. elDiario.es. Recuperado de https://www.eldiario.es/hojaderouter/tecnologia/shakey-robot-inteligencia-artificial-coche-autonomo_1_3466717.html

[9] Faculty of Electrical Engineering, Czech Technical University in Prague. (s.f.). *B4M36UIR Lecture 02: Introduction to Artificial Intelligence*. Recuperado de: https://cw.fel.cvut.cz/b201/_media/courses/b4m36uir/lectures/b4m36uir-lec02- slides.pdf

[10] Zhou, C., Huang, B., & Fränti, P. (2022). *A review of motion planning algorithms for intelligent robots*. Journal of Intelligent Manufacturing, 33, 387- 424. https://doi.org/10.1007/s10845-021-01867-z

[11] Siegwart, R., Arras, K. O., & Nourbakhsh, I. R. (2003). *Adaptive behavior-based control for robot navigation: A multi-robot case study*. Autonomous Robots, 15, 101-110. https://www.researchgate.net/publication/259646134_Adaptive_behavior- based_control_for_robot_navigation_A_multi-robot_case_study

[12] Parker, L. E. (2008). *Potential Fields*. Universidad de Tennessee. Recuperado de https://web.eecs.utk.edu/~leparker/Courses/CS594- fall08/Lectures/Oct- 16- Potential-Fields.pdf

[13] Universidad de la República. (s.f.). *Clase 01: Introducción*. Recuperado de https://eva.fing.edu.uy/pluginfile.php/387182/mod_resource/content/1/clase01_introduccion.pdf

[14] Murphy, R. R. (2000). *Introduction to AI robotics*. MIT Press.

[15] Sciensity. (2013, febrero 14). *Autonomous robot architecture (AuRA)*. Recuperado de https://sciensity.blogspot.com/2013/02/autonomous-robot-architecture-aura.html

[16] Cassingena, E. (s.f). *Algoritmo de la ruta más corta de Dijkstra: introducción gráfica*. FreeCodeCamp. Recuperado, de https://www.freecodecamp.org/espanol/news/algoritmo-de-la-ruta-mas-corta-de-dijkstra-introduccion-grafica/

[17] Keyence Corporation. (s.f.). *¿Qué es un sensor fotoeléctrico?*. Recuperado de https://www.keyence.com.mx/ss/products/sensor/sensorbasics/photoelectric/info/

[18] Logos-World.net. (2024, junio 13). *MATLAB Logo, symbol, meaning, history, PNG, brand*. Recuperado de https://logos-world.net/matlab-logo/

[19] Yang, H., Liu, H., Zou, J., Yin, Z., Liu, L., Yang, G., Ouyang, X., & Wang, Z. (Eds.). (2023). *Intelligent robotics and applications: 16th International Conference, ICIRA 2023, Hangzhou, China, July 5–7, 2023, Proceedings, Part VI*. Springer. (Lecture Notes in Artificial Intelligence, Vol. 14272).

[20] Edjah, A., Coelho, L. S., & Mourele, L. M. (Eds.). (2007). *Mobile Robots: The Evolutionary Approach* (Vol. 50). Springer. ISBN: 978-3-540-49719-6.

[21] Siciliano, B., & Khatib, O. (Eds.). (2016). *Springer Handbook of Robotics* (2nd ed.). Springer. https://doi.org/10.1007/978-3-319-32552-1

[22] Hokuyo Automatic Co., Ltd. (s.f.). *Scanning Laser Range Finder Smart-URG mini UST- 20LX (UUST004) Specification*. https://www.generationrobots.com/media/hokuyo-laser-finder-UST-20LX%20Specification.pdf

[23] Zotovic, R. (2024). *Introducción a la robótica móvil.* Asignatura: Planificación y control avanzado de sistemas robotizados, Máster en Automática e Informática Industrial, Universidad Politécnica de Valencia.

[24] Zotovic, R. (2024). *Práctica 5: Modelado y simulación de robots móviles.* Asignatura: Planificación y control avanzado de sistemas robotizados, Máster en Automática e Informática Industrial, Universidad Politécnica de Valencia.

[25] MathWorks. (2023). *MATLAB* (Versión 2023a) [Software]. MathWorks. https://www.mathworks.com/products/matlab.html

[26] *Canva* [Software]. https://www.canva.com

[27] JGraph Ltd. *draw.io* [Software]. https://www.draw.io

[28] MathWorks. (s.f.). *switch (MATLAB)*. MathWorks. https://es.mathworks.com/help/matlab/ref/switch.html

[29] MathWorks. (s.f.). *plot (MATLAB)*. MathWorks. https://es.mathworks.com/help/matlab/ref/plot.html

[30] MathWorks. (s.f.). *Create a simple class (MATLAB)*. MathWorks. https://es.mathworks.com/help/matlab/matlab_oop/create-a-simple-class.html

[31] MathWorks. (s.f.). *function (MATLAB)*. MathWorks. https://es.mathworks.com/help/matlab/ref/function.html

[32] MathWorks. (s.f.). *transpose (MATLAB)*. MathWorks. https://es.mathworks.com/help/matlab/ref/transpose.html

[33] MathWorks. (s.f.). *mldivide (MATLAB)*. MathWorks. https://es.mathworks.com/help/matlab/ref/mldivide.html

[34] MathWorks. (s.f.). *mod (MATLAB)*. MathWorks. https://es.mathworks.com/help/matlab/ref/mod.html

[35] Real Academia Española. (s.f.). *Robot*. En *Diccionario de la lengua española* (23.ª ed.). Recuperado de https://dle.rae.es/robot

9. Anexo

9.1. ODS

Objetivos de Desarrollo Sostenibles	**Alto**	**Medio**	**Bajo**	**No procede**
ODS 1. Fin de la pobreza.				X
ODS 2. Hambre cero.				X
ODS 3. Salud y bienestar.				X
ODS 4. Educación de calidad.	X			
ODS 5. Igualdad de género.				X
ODS 6. Agua limpia y saneamiento.				X
ODS 7. Energía asequible y no contaminante.				X
ODS 8. Trabajo decente y crecimiento económico.				X
ODS 9. Industria, innovación e infraestructuras.	X			
ODS10. Reducción de las desigualdades.				X
ODS 11. Ciudades y comunidades sostenibles.				X
ODS 12. Producción y consumo responsables.				X
ODS 13. Acción por el clima.				X
ODS 14. Vida submarina.				X
ODS 15. Vida de ecosistemas terrestres.				X
ODS 16. Paz, justica e instituciones sólidas.				X
ODS 17. Alianzas para lograr objetivos.				X

Tabla 9. Objetivos de Desarrollo Sostenibles

Fuente. Propia.

Este proyecto solamente tiene impacto en los *ODS 4. Educación de calidad* y *ODS 9. Industria, innovación e infraestructuras.*

El impacto en el *ODS 4. Educación de calidad* es alto, ya que este trabajo puede servir para demostrar de una forma didáctica y visual cómo funciona la arquitectura de un robot móvil. Puede servir para asignaturas que estén relacionadas con la robótica para reforzar el temario. Además, la base de este proyecto es una práctica de un máster impartido por la UPV.

Por otra parte, el impacto con el *ODS 9. Industria, innovación e infraestructuras,* es alto, debido a que presenta una innovación en el sector de la simulación de robots móviles en MATLAB.

9.2. Arquitectura basada en campos potenciales

Esta arquitectura se basa en la idea de que el robot se mueve dentro de un campo de fuerzas virtuales, donde cada punto en el espacio tiene un valor de "potencial" que guía el movimiento del robot. De esta forma, cada comportamiento devuelve un vector y el comportamiento resultante, que es el que ejecuta el robot, es la suma ponderada de varios vectores.

Existen cinco campos potenciales primitivos: uniforme, perpendicular, atracción, repulsión y tangencial. En la Figura 82, se pueden ver estos campos potenciales.

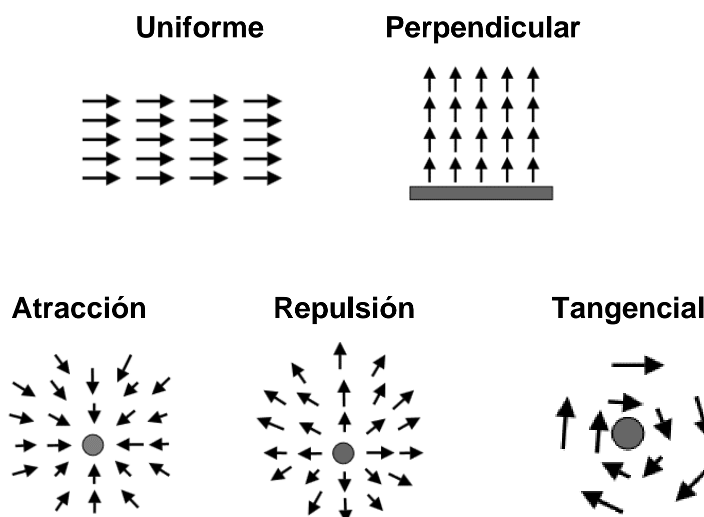
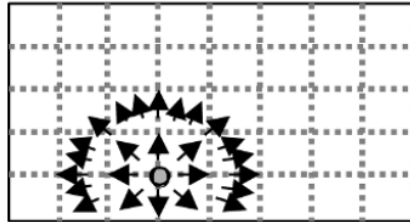


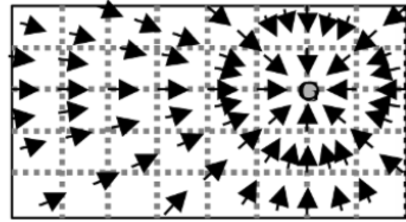
Figura 82. Campos potenciales primitivos.
Fuente. Propia.

Para un robot, la suma de los campos potenciales primitivos resulta en comportamientos. En la figura X, se muestran la fusión entre la repulsión de un obstáculo (O) y la atracción hacia un objetivo (G).

Repulsión desde el obstáculo



Atracción hacia un objetivo



Combinación de los dos

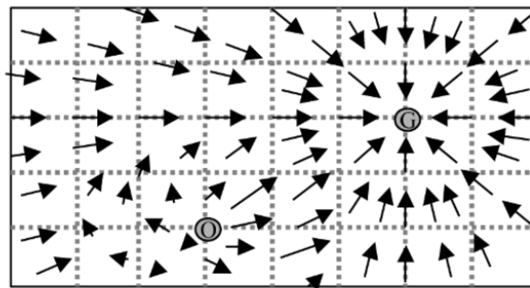


Figura 83. Campos potenciales en un entorno.

Fuente. [14].

Así pues, el camino recorrido por el robo en el ejemplo anterior se puede ver en la Figura 84.

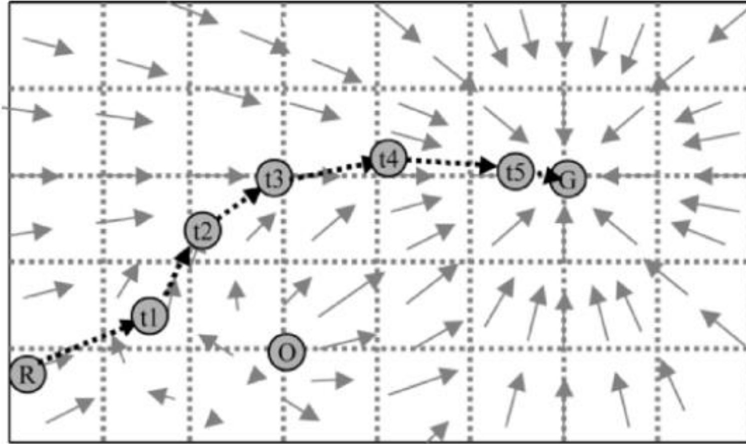
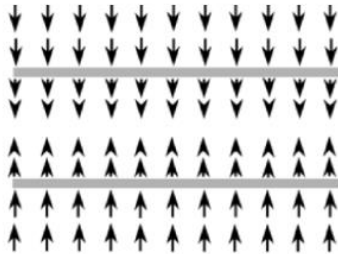


Figura 84. Camino seguido por el robot en un campo potencial.

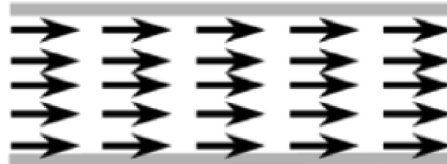
Fuente. [14].

Por último, en la Figura 85 se pueden ver los campos potenciales de algunos comportamientos del robot.

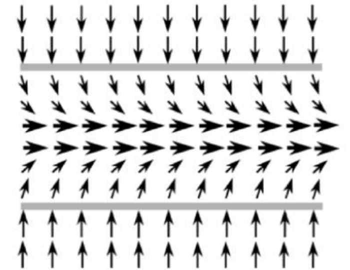
Campo potencial perpendicular a un pasillo



Campo potencial uniforme en un pasillo



Campo del comportamiento seguir pasillo



Campo potencial del comportamiento aparcar con obstáculos

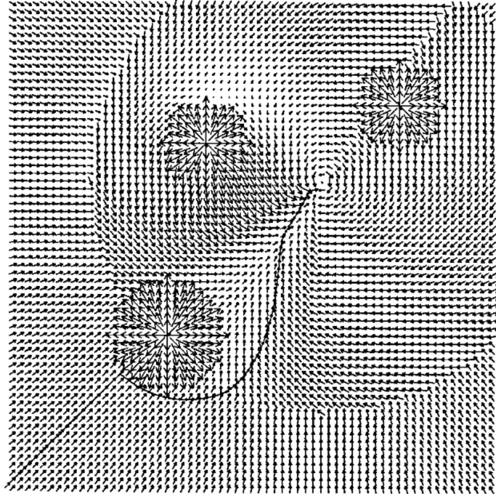


Figura 85. Campos potenciales de algunos comportamientos del robot.

Fuente. [14].

En resumen, las arquitecturas basadas en campos potenciales guían a los robots móviles utilizando campos potenciales. Aunque es un método eficiente computacionalmente, puede presentar problemas como quedar atrapado en mínimos locales y experimentar oscilaciones alrededor de obstáculos.

9.3. Cálculo de los puntos de intersección

Para calcular los puntos de intersección, se utilizan las siguientes variables:

- $(\boldsymbol{p_{x1}}, \boldsymbol{p_{y1}})$ $(\boldsymbol{p_{x2}}, \boldsymbol{p_{y2}})$

  Son los puntos que definen la línea a detectar. Se extraen de la matriz que contiene los obstáculos en el entorno
- $(\boldsymbol{sensor_x}, \boldsymbol{sensor_y})$

  Son las coordenadas del sensor láser en el espacio, extraídas de su matriz homogénea.
- $(\boldsymbol{direccion_x}, \ \boldsymbol{direccion_y})$

  Son las componentes del vector de dirección del rayo del sensor láser.

Primero se comprueba si el láser está apuntando hacia las líneas de la pared. Para ello, se comprueba si la $direccion_x$ del láser está en la misma dirección que las

coordenadas X de la pared (p_{x1}, p_{x2}) y lo mismo con la $direccion_y$ y las coordenadas Y (p_{y1}, p_{y2}) de la pared. Se utiliza la proyección de un vector sobre otro vector, sabiendo que, si el resultado es mayor a 0, el ángulo ente los vectores es agudo y significa que están en una dirección similar.

$$(coordenada_{n1} - coordenada_{n2}) * direccion_n \geq 0$$

Ecuación 1. Comprobación de que el láser apunta a la pared.

Fuente. Propia.

Siendo n la componente *x* o *y*.

Para calcular el punto de intersección entre la pared y el láser, se plantean dos ecuaciones paramétricas:

$$(x, y) = (p_{x1}, p_{y1}) + \lambda\left((p_{x2} - p_{x1}), (p_{y2} - p_{y1})\right)[m]$$

Ecuación 2. Ecuación paramétrica del segmento de la pared.

Fuente. Propia.

Donde λ es el parámetro que recorre desde (p_{x1}, p_{y1}) hasta (p_{x2}, p_{y2}).

$$(x, y) = (sensor_x, sensor_y) + t(direccion_x, direccion_y)[m]$$

Ecuación 3. Ecuación paramétrica del rayo del sensor láser.

Fuente. Propia.

Para encontrar el punto de intersección, se igualan ambas ecuaciones y se resuelve para t obteniendo:

$$sensor_x + t * direccion_x = p_{x1} + \lambda\,(p_{x2} - p_{x1})$$
$$sensor_y + t * direccion_y = p_{y1} + \lambda\,(p_{y2} - p_{y1})$$

Ecuación 4. Sistema de ecuaciones paramétricas para encontrar un punto de intersección.

Fuente. Propia.

Simplificando y eliminando λ multiplicando la primera ecuación por $(p_{y2} - p_{y1})$y la segunda por$(p_{x2} - p_{x1})$ se queda:

$$\left(\left((sensor_x * (p_{y2} - p_{y1})\right) - (sensor_y * (p_{x2} - p_{x1})) + t * (direccion_x * (p_{y2} - p_{y1}) - direccion_y * (p_{x2} - p_{x1})\right) = px_1 * (p_{y2} - p_{y1}) - p_{y1} * (p_{x2} - p_{x1})$$

Ecuación 5. Ecuación paramétrica simplificada.

Fuente. Propia.

De ahí se resuelve para t ya que, si es mayor o igual a 0, significa que el punto de intersección está en la dirección del rayo. Si fuera negativo, estaría detrás del sensor láser.

$$t = \frac{(p_{x1} - sensor_x) * (p_{y2} - p_{y1}) - (p_{y1} - sensor_y) * (p_{x2} - p_{x1})}{direccion_x * (py_2 - p_{y1}) - direccion_y * (p_{x2} - p_{x1})}$$

Ecuación 6. Ecuación para encontrar un punto de intersección con el láser.

Fuente. Propia.

Una vez encontrado t, el punto de intersección se obtiene sumando las coordenadas x e y del sensor (extraídas de la matriz homogénea) y el desplazamiento en la dirección del láser multiplicado por t:

$$punto\ de\ interseccion = matriz(1{:}\,2, 4) + t * [direccion_x, direccion_y]$$

Ecuación 7. Ecuación del punto de intersección.

Fuente. Propia.

En el código aparece directamente la fórmula de t en la línea 208 dentro de la función *interseccion* de la clase sensor láser, Anexo 9.6.2. El punto de intersección aparece calculado en la línea 211 y la comprobación de si el láser está apuntando a la pared aparece en la línea 207 de la clase sensor láser.

Estos cálculos proporcionan tanto las coordenadas x e y del punto de intersección como la distancia en metros entre el sensor y el punto de intersección.

9.4. Cálculo de vectores

En este apartado se detallan los cálculos de los vectores que indican el movimiento del robot.

9.4.1. Vector de coordenadas finales

Como se ha visto previamente en el apartado de comportamientos, sección 5.2.3., las coordenadas finales que indican la posición final del robot se obtienen mediante cálculos.

Primero hay que tener en cuenta la estructura del vector de coordenadas finales, representado en la Tabla 2. Como se puede ver, el vector de coordenadas finales tiene 3 elementos:

$$coordenadas\ finales = [x, y, \theta]$$

Donde los dos primeros elementos (x, y) son las coordenadas de posición en metros y el tercer elemento (θ) es el ángulo con la orientación en radianes. En el código luego se utiliza la transpuesta del vector para utilizarla en los cálculos.

Así pues, para desplazar al robot por el entorno con velocidades, se deben integrar al espacio cartesiano, es decir, se debe pasar de velocidad a coordenadas. La ecuación para ello es la siguiente:

$$\boldsymbol{coordenadas_{finales}} = \overset{a}{\begin{bmatrix} x_{inicial} \\ y_{inical} \\ \theta_{inicial} \end{bmatrix}} + \overset{b}{\begin{bmatrix} velocidad * T * \cos(\theta + \omega * T) \\ velocidad * T * \sin(\theta + \omega * T) \\ \omega * T \end{bmatrix}}$$

Ecuación 8. Cálculo de las coordenadas de desplazamiento del robot.

Fuente. Propia.

Donde:

- a: es la matriz con las coordenadas iniciales del robot
- b: es la matriz con los desplazamientos del robot
  - $\boldsymbol{velocidad}$: es la velocidad del objeto (cuánto se mueve por unidad de tiempo) en metros por segundo.

- $\boldsymbol{\omega}$**:** es la velocidad angular del objeto (rapidez de cambio en el ángulo de orientación) en radianes por segundo.
- $\boldsymbol{T}$: es el intervalo de tiempo en segundos durante el cual se observa el movimiento del objeto.
- $\boldsymbol{\theta}$**:** es el ángulo inicial de orientación del objeto (la dirección en la que se está moviendo inicialmente). Se mide en radianes.

Para entender los desplazamientos, se va a analizar paso por paso los componentes de la matriz b.

- $\mathbf{cos}(\boldsymbol{\theta} + \boldsymbol{\omega} * \boldsymbol{T})$ y $\mathbf{sin}\ (\boldsymbol{\theta} + \boldsymbol{\omega} * \boldsymbol{T})$
  Estos términos calculan la nueva dirección del movimiento del objeto después de un tiempo *T.* La dirección cambia debido a la velocidad angular ω.

- $\boldsymbol{velocidad} * \boldsymbol{T} * \mathbf{cos}(\boldsymbol{\theta} + \boldsymbol{\omega} * \boldsymbol{T})$ y $\boldsymbol{velocidad} * \boldsymbol{T} * \mathbf{sin}(\boldsymbol{\theta} + \boldsymbol{\omega} * \boldsymbol{T})$
  Estos términos calculan el desplazamiento en las direcciones x e y, respectivamente, después de un tiempo *T*. Se multiplica la velocidad por el tiempo *T* para obtener la distancia recorrida y luego se descompone esa distancia en componentes *x* e *y* usando coseno y seno del nuevo ángulo. Para entender mejor la descomposición de seno y coseno, mirar la Figura 21 que muestra la representación de las coordenadas *x* e *y* del robot.

- $\boldsymbol{\omega} * \boldsymbol{T}$
  Este término calcula el cambio en el ángulo de orientación del objeto después de un tiempo *T*.

Este cálculo de coordenadas aparece en todos los comportamientos simplificado de la siguiente forma:

$$\boldsymbol{coordenadas_{finales}} = coordenadas_{iniciales} + [velocidad * T * \cos(\theta + w * T), velocidad * T * \sin(\theta + w * T), w * T]$$

Ecuación 9. Cálculo de las coordenadas de desplazamiento del robot simplificado.

Fuente. Propia.

En la línea 214 del código de la clase robot, apartado 9.6.1, se puede ver un ejemplo de la utilización de este cálculo.

Este método de integración es aproximativo y acumula errores con el tiempo. Algunos métodos similares se utilizan en los vehículos reales para la estimación de la postura basándose en la odometría. Este método es una de las razones del error de posición en los robots móviles.

9.4.2. Vector de velocidad lineal

Al igual que con el cálculo de posición a velocidad, se puede hacer la inversa y pasar de velocidad a posición. Para ello, se sigue la siguiente ecuación para la velocidad lineal:

$$\boldsymbol{velocidad} = kr * \sqrt{v_x^2 + v_y^2}\ [m/s]$$

Ecuación 10. Cálculo de la velocidad lineal.

Fuente. Propia.

Donde:

- $\boldsymbol{kr}$ es la constante de velocidad
- $\boldsymbol{v_x}$ es la componente del vector velocidad en la dirección *x.*
- $\boldsymbol{vy}$ es la componente del vector velocidad en la dirección *y*.

Para entender mejor como se calcula la velocidad, se va a hacer un desglose de la fórmula:

- $\boldsymbol{v_x^2 + v_y^2}$

  Este término es la suma de los cuadrados de las componentes *x* e *y* del vector velocidad. Esto se utiliza para calcular la magnitud del vector en dos dimensiones utilizando el teorema de Pitágoras. Luego se hace la raíz cuadrada de esta suma para obtener la velocidad lineal del objeto.

- $\boldsymbol{kr}$
  Escala la magnitud del vector velocidad para tener en cuenta factores específicos del sistema o del entorno.

Así pues, esta fórmula indica que la velocidad lineal del objeto es proporcional a la magnitud del vector velocidad en el plano *xy*, ajustada por una constante de velocidad *kr*. La constante *kr* ajusta esta relación para adaptarla a las condiciones específicas del sistema.

Este cálculo se realiza en todos los comportamientos, ya que es necesario para calcular las coordenadas finales del robot. Utiliza como vector para el cálculo el vector resultante de cada comportamiento. Un ejemplo de este cálculo se encuentra en el apartado 9.6.1. en la línea 414.

9.4.3. Vector de velocidad angular

Al igual que en el apartado anterior, se puede calcular la velocidad angular a partir de la posición. Para ello, primero se calcula el ángulo con la siguiente fórmula:

$$\boldsymbol{\theta} = \operatorname{atan}\left(\frac{v_y}{v_x}\right) [rad]$$

Ecuación 11. Cálculo del ángulo para la velocidad angular.

Fuente. Propia.

Donde:

- v_x es la componente del vector velocidad en la dirección *x.*
- vy es la componente del vector velocidad en la dirección *y.*

Y al ángulo, se le multiplica la constante de rapidez de giro kw:

$$\boldsymbol{\omega} = kw * \theta \ \ [rad/s]$$

Ecuación 12. Cálculo de la velocidad angular.

Fuente. Propia.

Estos cálculos se utilizan en todos los comportamientos de la arquitectura, ya que son necesarios para calcular las coordenadas hacia donde se tiene

que mover. Un ejemplo de ellos se encuentra en las líneas 415 y 416, en el código de la clase robot del anexo 9.6.1.

9.4.4. Vector seguir recto

En este caso, el vector para hacer que el robot sigua recto es el siguiente:

$$\boldsymbol{vector}_{\boldsymbol{seguir\ recto}} = [1, 0]$$

Ecuación 13. Vector seguir recto.

Fuente. Propia.

Donde:

- **El primer componente (1):** impulsa al robot hacia adelante. Si por ejemplo el robot está orientado verticalmente, este componente hará que el robot se mueva hacia arriba en el plano global.

- **El segundo componente (0):** es el movimiento lateral, perpendicular a la orientación del robot. Al estar a 0, se asegura que no hay movimientos laterales no deseados.

Este vector está siempre declarado con estos valores en el código, como por ejemplo en la línea 395 de la clase robot, en el anexo 9.6.1.

9.4.5. Vector objetivo

El vector objetivo es el vector resultante del comportamiento de ir hacia un objetivo. Se calcula de la siguiente forma:

$$Matriz\ homogénea\ del\ robot * vector_{objetivo} = vector\ con\ las\ coordenadas\ del\ objetivo \rightarrow$$

$$\rightarrow \boldsymbol{vector}_{\boldsymbol{objetivo}} = Matriz\ homogénea\ del\ robot \backslash vector\ de\ coordenadas\ objetivo$$

Ecuación 14. Cálculo del vector objetivo.

Fuente. Propia.

Donde:

- **La matriz homogénea del robot** es una matriz 4x4 que contiene los datos de traslación y rotación del robot.

- **El vector con las coordenadas del objetivo** es un vector 4x1.

- **** es el operador usado en MATLAB para resolver un sistema de ecuaciones lineales.

Este cálculo está representado en la función *Vect_objetivo* de la clase robot, referenciada en el anexo 9.6.1 en las líneas 189 y 190.

9.4.6. Vector evita obstáculos

El vector de repulsión es el vector resultante del comportamiento del robot al evitar un obstáculo. Se calcula de la siguiente forma:

$$\boldsymbol{vector_{repulsion}} = -[cos(\theta), sin(\theta)]$$

$$\boldsymbol{k} = \frac{K(\theta)}{distancia\hat{}n}$$

$$\boldsymbol{v_{obstaculos}} = v_{obstaculos} + k * vector_{repulsion};$$

Ecuación 15. Cálculo del vector de evitación de obstáculos.

Fuente. Propia.

Donde:

- $\boldsymbol{\theta}$**:** es el ángulo actual en grados que indica la dirección hacia el obstáculo desde el robot.

- $\boldsymbol{vector_{repulsion}}$**:** calcula el vector de atracción hacia el obstáculo con el coseno y el seno y le invierte el signo para que apunte a la dirección opuesta, alejándose del obstáculo.

- $\boldsymbol{k}$**:** es la constante de repulsión calculada dividendo la constante específica para el ángulo actual $\boldsymbol{K(\theta)}$ entre la distancia hacia el objeto ($\boldsymbol{distancia}$) elevada a la constante que disminuye la fuerza a de repulsión con la distancia ($\boldsymbol{n}$).

- $\boldsymbol{v_{obstaculos}}$**:** es el vector de evitación de obstáculos, obtenido sumando los anteriores valores para los ángulos y sumando la

multiplicación de la constante de repulsión por el vector de repulsión.

Este cálculo está representado en las líneas 308, 309 y 310 de la función *evita_obstaculos,* referenciada en el anexo 9.6.1.

9.4.7. Vector orientación pared

El vector de orientación hacia la pared se obtiene mediante las medidas de los ángulos laterales. El vector se calcula de la siguiente forma:

$$\boldsymbol{vector_{orientacion\ pared}} = [0\ ,\ diferencia\ entre\ las\ distancias\ medidas];$$

Ecuación 16. Cálculo del vector de orientación a la pared.

Fuente. Propia.

Donde:

- **El primer compontente (0)** indica que el robot no se mueve hacia adelante.

- El segundo componente, la **diferencia entre las distancias medidas** es el resultado de restar la distancia medida de un ángulo lateral menos la otra. Al estar en la segunda posición del vector de orientación, esto hace que el robot se desplace lateralmente, en este caso, hacia la pared elegida mediante el resultado de la diferencia entre las distancias.

Un ejemplo de la diferencia de las distancias es el siguiente:

$$diferencia\ entre\ las\ distancias\ medidas = distancia_a - distancia_b$$

Ecuación 17. Cálculo de la diferencia entre las distancias.

Fuente. Propia.

Dependiendo de si la pared está a la derecha o a la izquierda, la diferencia entre las distancias cambia de signo. Además, se hace una comprobación inicial cada vez que se mueve el robot para saber si está a la distancia de referencia usando el ángulo lateral. Si la distancia del ángulo lateral es igual que la de referencia, el robot está bien situado y entonces el vector pasa a

ser **[0,0]** para indicar que el robot ya no necesita orientarse. En el comportamiento de seguir la pared, el robot sigue recto una vez que se ha orientado gracias a que el vector de orientación se suma al de seguir recto.

Todos estos cambios están reflejados en la función *orientación_pared* de la clase robot, referenciada en el anexo 9.6.1 en las líneas 515 hasta la 518.

9.4.8. Vector orientación pasillo

El vector de orientación pasillo funciona igual que el de orientación para seguir una pared, exceptuando que, en vez de coger los dos ángulos del mismo lado para calcular la orientación, coge los contrarios.

En el caso de esta arquitectura, coge los ángulos de 90º y 270º y calcula la distancia con ellos para saber donde se encuentra. Si por ejemplo el de 90º tiene una distancia mayor que el de 270º, sabe que se encuentra más lejos de la pared izquierda (90º) que de la derecha (270º). Si eso pasa, coge los ángulos laterales del mismo lado para recalcular el vector y orientarse hacia la pared que esté más lejos de la misma forma que en orientación pared: restando las distancias de los ángulos. Una vez está orientado en el centro (los ángulos laterales miden la misma distancia) este vector pasa a ser **[0,0].**

Todos estos cálculos están escritos en la función *orientación_pasillo* de la clase robot, referenciada en el anexo 9.5.1 en las líneas 719 hasta la 825.

9.4.9. Vector infinito

El vector infinito se calcula creado un vector de atracción hacia los ángulos que tengan una medida de escaneo infinita. Para ello:

$$\boldsymbol{vector_{infinito}} = [cos(\theta), sin(\theta)];$$

$$\boldsymbol{vector} = vector + k * vector_{infinito};$$

Ecuación 18. Cálculo del vector infinito.

Fuente. Propia.

Donde:

- $\boldsymbol{\theta}$**:** es el ángulo que apunta hacia el infinito.

- $\boldsymbol{vector_{infinito}}$: es el vector de atracción hacia el infinito para un ángulo en concreto, calculado mediante el coseno y el seno.

- $\boldsymbol{vector}$: es el vector resultante de sumar los anteriores vectores de atracción al infinito $(vector\ )$ con el vector de atracción al infinito actual $(vector_{infinito})$ multiplicado por su constante (k).

Este vector se encuentra calculado en las líneas 916 y 917 de la función *infinito* de la clase robot, referenciada en el anexo 9.6.1.

9.4.10. Vector aparcamiento

Este vector se calcula de una forma distinta a los vectores de repulsión u orientación previos. Aquí se utiliza para guiar al robot hacia una ubicación específica, considerando una zona de aparcamiento definida alrededor de las coordenadas objetivo.

Para empezar, se definen unas coordenadas específicas a alcanzar, y se dibuja un área donde el robot debe aparcar, definida por las distancias laterales y frontales desde las coordenadas del objetivo. Posteriormente, se ajustan las coordenadas objetivo situándolas 2 unidades por debajo de los puntos que definen el área.

Primero, se definen las coordenadas del objetivo, $(x_{objetivo}, y_{objetivo})$ por ejemplo. Luego, se establece un área de aparcamiento alrededor de este punto, considerando distancias laterales y frontales, por ejemplo, 2 unidades laterales y 3 unidades frontales. Una vez definida esta área, se ajustan las coordenadas objetivo para facilitar el aparcamiento del robot. Este ajuste se realiza situando el objetivo 2 unidades por debajo del punto original, resultando en las nuevas coordenadas $(x_{objetivo}, y_{objetivo} - 2)$. A partir de ahí se ejecuta el comportamiento *objetivo* que guía al robot a ese punto con el vector objetivo.

Todos estos cálculos están reflejados en las líneas 952 hasta la 957 del comportamiento *aparcar*, referenciada en el anexo 9.6.1.

9.5. Manual de usuario

El manual de usuario proporciona una guía completa para poder probar las simulaciones de esta arquitectura. A lo largo de este manual, se explorarán las opciones que tiene un usuario a la hora de cambiar o ajustar cosas de la arquitectura, así como probar su funcionamiento.

Nota: No borrar la comilla al final de algunos vectores (ej: [x,y,theta]') **ya que hace la transpuesta del vector y es necesaria para los cálculos.**

9.5.1. Modificaciones en *crear_sensor_láser*

Para poder cambiar parámetros dentro del sensor láser, como el ángulo inicial o la resolución, en el constructor de la clase (líneas 31 a la 57) se encuentran inicializados todos los valores que pueden ser cambiados. No es recomendable bajar el rango de 270º del sensor, pues podría afectar a como se orienta el robot en el entorno.

Por otra parte, las dimensiones del láser se pueden cambiar con las propiedades *obj.lin* que definen las 4 líneas del sensor (líneas 42-45). Como se puede ver, dentro de esta propiedad están las coordenadas x,y,z [punto inicial, punto final] y el color de la línea en RGB [rojo, verde, azul]. Para seleccionar el color, marcar con un 1 la opción de color y con un 0 el resto. Si se quiere representar de color negro, poner todo a 0 y si se quiere representar en blanco, poner todo a 1.

No se recomienda eliminar y/o modificar la línea 39, ya que contiene los datos del robot y son necesarios para calcular todo lo relacionado con el sensor láser.

```
31      function obj = crear_sensor_laser(robot)
32          obj.angulo_inicial = 0;
33          obj.angulo_final = 270;
34          obj.resolucion = 0.25;
35          obj.velocidad = 25;
36          obj.valor_minimo = 0.06;
37          obj.valor_maximo = 20;
38          obj.posicion_sensor_rel = [0, 0, 0]; % Modificar posición relativa
39          obj.robot = robot;
40          obj.puntos_interseccion=[];
41
42          obj.lin1 = line('xdata', [-0.1;0.1]', 'ydata', [0.1;0.1]', 'zdata', [0;0]', 'color', [0 1 0]);
43          obj.lin2 = line('xdata', [0.1;0.1]', 'ydata', [0.1;-0.1]', 'zdata', [0;0]', 'color', [1 0 0]);
44          obj.lin3 = line('xdata', [-0.1;0.1]', 'ydata', [-0.1;-0.1]', 'zdata', [0;0]', 'color', [0 1 0]);
45          obj.lin4 = line('xdata', [-0.1;-0.1]', 'ydata', [-0.1;0.1]', 'zdata', [0;0]', 'color', [0 1 0]);
46          obj.dibujo = [obj.lin1, obj.lin2, obj.lin3, obj.lin4];
47          obj.haz_lines = []; %inicializar la propiedad haz_lines
48
49          %calcular la posición absoluta del sensor y actualizar la matriz
50          [obj.coordenada_x_origen, obj.coordenada_y_origen] = obj.calculo_origen(robot);
51          [obj.posicion_sensor_abs, obj.posicion_sensor_rel] = obj.calcular_posicion_absoluta(robot);
52          [obj.direccion_x, obj.direccion_y] = obj.direccion_laser();
53          obj.matriz_sensor_laser = obj.actualizar_matriz_sensor();
54          %si se quiere probar la función de intersección sola, comentar
55          %la siguiente línea de código:
56          [obj.puntos_interseccion, obj.plotHandle] = obj.calcular_interseccion_multiple(robot);
57      end
58
```

Figura 86. Parámetros ajustables del sensor láser.

Fuente. Propia.

Por otra parte, si se desea ver el haz que traza el sensor con su rango, en *calcular_interseccion_multiple* (líneas 157-187) hay una parte de código comentada que representa el haz del sensor. Esta parte comentada corresponde a las líneas 182 hasta 185 y para utilizarla hay que quitar los % que simbolizan un comentario en MATLAB. No es recomendable usarla en simulaciones donde el robot muestra poco a poco su movimiento, ya que el rendimiento del programa baja considerablemente y puede que se vea lento.

```
157     function [puntos_interseccion, plotHandle] = calcular_interseccion_multiple(obj, robot)
158         %inicializar una lista para almacenar los puntos de intersección
159         puntos_interseccion = [];
160         plotHandle = [];
161         hold on;
162
163         %limpiar las líneas del haz existentes antes de dibujar nuevas
164         if ~isempty(obj.haz_lines)
165             delete(obj.haz_lines);
166             obj.haz_lines = [];
167         end
168
169         %iterar sobre los ángulos dentro del rango angular
170         for angulo_actual = obj.angulo_inicial:obj.resolucion:obj.angulo_final
171             obj.angulo = angulo_actual;
172             %calcular la dirección del láser
173             angulo_ajustado = mod(225 + angulo_actual, 360); %ajustar el ángulo según la orientación del sensor
174             obj.direccion_x = cosd(angulo_ajustado + rad2deg(robot.coordenadas_finales(3)));
175             obj.direccion_y = sind(angulo_ajustado + rad2deg(robot.coordenadas_finales(3)));
176             %calcular la intersección del haz láser con las líneas del entorno
177             [obj.distancia, obj.punto, plotHandles] = obj.interseccion(robot);
178             plotHandle = [plotHandle, plotHandles];
179             puntos_interseccion = [puntos_interseccion; angulo_ajustado, obj.punto(1), obj.punto(2), obj.distancia];
180
181             % Dibujar el haz del sensor láser
182                % x_data = [obj.posicion_sensor_abs(1), obj.posicion_sensor_abs(1) + 20 * cosd(angulo_ajustado + rad2deg(robot.coordenadas_iniciales(3)))];
183                % y_data = [obj.posicion_sensor_abs(2), obj.posicion_sensor_abs(2) + 20 * sind(angulo_ajustado + rad2deg(robot.coordenadas_iniciales(3)))];
184                % haz_line = plot(x_data, y_data, 'b-', 'LineWidth', 0.1); % Línea azul fina
185                % obj.haz_lines = [obj.haz_lines, haz_line]; % Almacenar el handle de la línea del haz
186         end
187     end
```

Figura 87. Comentario para dibujar el haz del sensor.

Fuente. Propia.

Por último, se pueden representar los puntos de intersección con el entorno mediante una línea de código que está comentada dentro de la función *intersección* (líneas 189-233). Las líneas de código que representan los puntos de intersección son de la 225 a la 227, ya que dibujan la propiedad *PlotHandles* que administra el dibujo de los puntos en el entorno. Para poder usarla, hay que quitar los % que comentan la función, como en el caso anterior. Además, para el caso en el que el robot muestre poco a poco su movimiento, no es recomendable usarla, ya que afecta al rendimiento del programa al intentar representar todos los puntos de intersección del láser mientras calcula la nueva posición del robot.

```
205
206         %calcular t para obtener el punto de intersección
207         if (px2 - px1) * obj.direccion_x >= 0 || (py2 - py1) * obj.direccion_y >= 0
208             t = ((px1 - sensor(1, 4)) * (py2 - py1) - (py1 - sensor(2, 4)) * (px2 - px1)) / (obj.direccion_x * (py2 - py1) - obj.direccion_y * (px2 - px1));
209             if t >= 0
210                 %calcular las coordenadas del punto de intersección
211                 punto_interseccion = sensor(1:2, 4) + t * [obj.direccion_x; obj.direccion_y];
212                 %verificar si el punto de intersección está dentro del segmento de la línea
213                 if punto_interseccion(1) >= min(px1, px2) && punto_interseccion(1) <= max(px1, px2) && punto_interseccion(2) >= min(py1, py2) && punto_interseccion(2) <= max(py1, py2)
214                     %calcular la distancia desde el sensor al punto de intersección
215                     distancia_temp = norm(punto_interseccion - sensor(1:2, 4));
216                     if distancia_temp < distancia
217                         if distancia_temp > obj.valor_maximo
218                             distancia=Inf;
219                             punto(1)=Inf;
220                             punto(2)=Inf;
221                         else
222                         distancia = distancia_temp;
223                         punto = punto_interseccion;
224                         end
225                         % hold on;
226                         %plotHandles = [plotHandles, plot(punto(1), punto(2), 'ro', 'MarkerSize', 5)]; %punto rojo
227                         % hold off;
228                     end
229                 end
230             end
231         end
232     end
233 end
```

Figura 88. Comentario para dibujar los puntos de intersección.

Fuente. Propia.

No se recomienda cambiar cualquier otra función o parámetro dentro de este archivo porque puede afectar al funcionamiento del robot.

9.5.2. Modificaciones en *crear_robot*

Al igual que con el sensor láser, las propiedades del robot se pueden cambiar en el constructor (líneas 75 a la 89). Dentro de los parámetros que se pueden cambiar, se recomienda que las coordenadas finales sean las coordenadas donde se quiere situar el robot si no se va a utilizar ninguna otra función.

Además, tampoco se recomienda cambiar y/o modificar la línea 76 que crea el sensor láser, ya que, sin él, el robot no puede orientarse.

Para el entorno del constructor se puede cambiar la línea 64 que llama al entorno que crea una única pared. Si se quisiera utilizar otro entorno, modificar esto por el entorno correspondiente (ej: *obj.entorno* = entorno_evita_obstculos()). Para cambiar el color, la función *dibuent* tiene unos paréntesis que corresponden a los colores RGB: [rojo verde azul]. Poner un 1 en el color que se desea pintar y un 0 a los que no se quieran usar.

Al igual que con el sensor láser, se puede cambiar las dimensiones del robot con los *obj.lin* del robot. Para más información, leer el apartado anterior: Anexo 9.3.1.

```
55  function obj = crear_robot()
56      obj.lin1 = line('xdata', [-0.2;0.8]', 'ydata', [0.4;0.4]', 'zdata', [0;0]', 'color', [0 0 1]);
57      obj.lin2 = line('xdata', [0.8;0.8]', 'ydata', [0.4;-0.4]', 'zdata', [0;0]', 'color', [1 0 0]);
58      obj.lin3 = line('xdata', [-0.2;0.8]', 'ydata', [-0.4;-0.4]', 'zdata', [0;0]', 'color', [0 0 1]);
59      obj.lin4 = line('xdata', [-0.2;-0.2]', 'ydata', [-0.4;0.4]', 'zdata', [0;0]', 'color', [0 0 1]);
60      obj.dibujo=[obj.lin1, obj.lin2, obj.lin3,obj.lin4];
61
62      obj.coordenadas_iniciales=[0, 0, 0]; %se pueden modificar
63      obj.coordenadas_finales=[0, 0, 0];
64      ent_pared = entorno_pared();
65      obj.entorno=ent_pared;
66      dibuent(obj.entorno, [0, 0, 0]);
67      obj.matriz_robot=XaMH(obj.coordenadas_iniciales);
68
69      obj.velocidad =10;
70      obj.w =obj.velocidad/5;
71      obj.tiempo =3*pi/4/abs(obj.w);
72      obj.T=0.1;
73
74      obj.sensor_laser=crear_sensor_laser(obj); %crea un sensor láser
75
76      obj=obj.mover_robot(obj.coordenadas_finales);
77      obj.kr=25;
78      obj.kw=5;
79  end
```

Figura 89. Parámetros que se pueden modificar del robot.

Fuente. Propia.

Dentro de cada comportamiento del robot, se pueden cambiar las constantes de velocidad (*obj.kr*), de giro (*obj.kw*) así como los diferentes pesos de los vectores (*k1, k2, k3…*) y el tiempo de actualización entre coordenadas (*obj.T*). Si alguno de los comportamientos no funciona correctamente y es debido a que el robot va muy rápido, ajustar las constantes de velocidad, de giro y la actualización de coordenadas a menos de lo que estaba definido. Si en cambio el robot se mueve a una velocidad adecuada pero no ejecuta el comportamiento bien, ajustar las constantes de peso de los vectores.

```
845             obj.T=0.01;
846             obj.kw=7;
847             obj.kr=10;
```

Figura 90. Ejemplo de las constantes de simulación.

Fuente. Propia.

Además, dentro de cada comportamiento se pueden cambiar otros valores, como por ejemplo las coordenadas iniciales del robot o el entorno donde se encuentra. La situación inicial del AGV viene definida por *obj.coordenadas_finales* donde el primer componente es la X, el segundo la Y, y el tercero la orientación del robot.

```
839         %coordenadas iniciales del robot
840         obj.coordenadas_iniciales = [0, 0, 0]';
841         obj.coordenadas_finales = [-10, 1, pi/2]'; %situación inicial del AGV
842         obj = obj.mover_robot(obj.coordenadas_finales);
843         obj.path = [];
```

Figura 91. Ejemplo de la definición de las coordenadas del robot en un comportamiento.

Fuente. Propia.

Para cambiar el entorno modificar *obj.entorno* al entorno correspondiente y para cambiar los ejes de la figura, cambiar la variable axis [(x inicial, x final, y inicial, y final)].

```
833             obj.entorno = entorno_puerta;
834             dibuent(obj.entorno, [1 0 0]);
835             axis([-20 10 0 40])
```

Figura 92. Ejemplo de la definición de los ejes y entorno en un comportamiento.

Fuente. Propia.

Por último, para cambiar distancias o valores específicos de los comportamientos, como por ejemplo la distancia de referencia o el lado de la pared a seguir, buscar los objetos dentro de cada simulación:

- *obj.d_ref* (distancia de referencia)
- *obj.pared* (pared a seguir)
- *obj.distancia_frontal, obj.distancia_derecha, obj.distancia_izquierda (*distancias para aparcar)

- *obj.X_objetivo* (coordenadas al objetivo)

No se recomienda cambiar cualquier otro valor dentro de la clase robot debido a que podría afectar al correcto funcionamiento del robot y/o el láser.

9.5.3. Modificaciones en *main*

En el fichero *main* se encuentran todas las simulaciones comentadas para poder probarlas. **Importante: no eliminar ni comentar las líneas de la 2 a la 6, ya que son las encargadas de crear el robot con el láser.** Para probar las simulaciones una a una, quitar el % delante de cada una de ellas. Se recomienda no quitarlos todos a la vez, ya que afecta al rendimiento del programa. Para probar otras simulaciones, se recomienda seguir la estructura de las que ya hay escritas, siendo la siguiente:

robot=función(robot);

Dentro del paréntesis todas las funciones toman como parámetro de entrada el robot, ya que es necesario conocer todos sus valores (posición, valores del sensor láser, distancias..) Para probar otras funciones, dentro del paréntesis de la función, poner el cursor sobre él cuando esté vacío para ver que parámetros de entrada pide. Si el parámetro es un objeto, poner *robot.*

En el main con las funciones que hay, su funcionamiento es el siguiente:

- **mover_robot(robot, coordenadas finales)**
  Esta función sirve para mover el robot a unas coordenadas en concreto en el entorno. Las coordenadas finales tienen que ser 3 y se corresponden a: [x , y, orientación]. En la Figura 93 se muestra un ejemplo de uso.

- **Simulación_giro_robot(robot, velocidad, radio)**

Esta función mueve al robot poco a poco según una velocidad y un radio definidos por el usuario. En la Figura 93 se muestra un ejemplo de uso.

- **girar_robot(robot, grados)**
  Gira al robot sobre su eje hasta alcanzar los grados definidos por el usuario. En la Figura 93 se muestra un ejemplo de uso.

- **robot=puntos_nuevos(robot, factor)**
  Esta función filtra los puntos de intersección según el factor indicado. Si por ejemplo el factor es de 8, filtrará 1 de cada 8 puntos. Se recomiendan valores altos, ya que el sensor detecta 1081 puntos aumentando en 0.25 el ángulo cada vez (90, 90.25, 90.50, 90.75, 91…). En la Figura 93 se muestra un ejemplo de uso.

No se recomienda cambiar y/o quitar cualquier otra línea de código del main.

```
1   %fichero de pruebas
2      clf
3      axis([-10 10 -10 10])
4      axis equal
5      hold on
6      robot=crear_robot();
7
8      %robot=mover_robot(robot, [1, 1, pi/2]);
9      %robot=simulacion_giro_robot(robot, 0.5, 1);
10     %robot=girar_robot(robot, 90);
11
12     robot=puntos_nuevos(robot, 8);
13
14     %robot=simulacion_objetivo(robot);
15     %robot=simulacion_evitar_obstaculos(robot);
16     %robot=simulacion_seguir_pared(robot);
17     %robot=simulacion_seguir_pasillo(robot);
18     %robot=simulacion_atravesar_puerta(robot);
19     %robot=simulacion_aparcar(robot);
20
21     %robot = piloto(robot);
```

Figura 93. Main para probar la simulación.

Fuente. Propia.

9.5.4. Consejos, uso en *Command Window* y Workspace

Para poder descomentar una o varias líneas en MATLAB, se puede usar la combinación Ctrl+Shift+R. En cambio, para poder comentar, se puede usar Ctrl+R. Para buscar palabras en concreto, como por ejemplo la distancia de referencia (*obj.d_ref*) con Ctrl + F se puede buscar por palabras y te muestra las coincidencias en el código. Este comando es útil para buscar rápidamente el parámetro que se quiere cambiar dentro de un comportamiento.

Por otra parte, para poder ver por el Command Window las variables del robot, se puede acceder a ellas insertando:

robot.propiedad a la que se quiera acceder

En la Figura 94 se muestra un ejemplo de uso de esta línea de código.

```
>> robot.coordenadas_iniciales

ans =

     0     0     0
```

Figura 94. Ejemplo de uso del Command Window para acceder a las propiedades del robot.
Fuente. Propia.

Para acceder a las propiedades del sensor láser, es necesario poner antes el objeto robot, ya que el láser está dentro de la clase robot.

```
>> robot.sensor_laser.angulo_inicial

ans =

     0
```

Figura 95. Ejemplo de uso del Command Window para acceder a las propiedades del sensor láser.
Fuente. Propia.

Por último, en la pestaña de la izquierda de MATLAB, está el Workspace, que almacena todos los objetos y los parámetros que crea el código. Si se hace doble click sobre el objeto, se abre otra pestaña donde se pueden ver todos los valores.

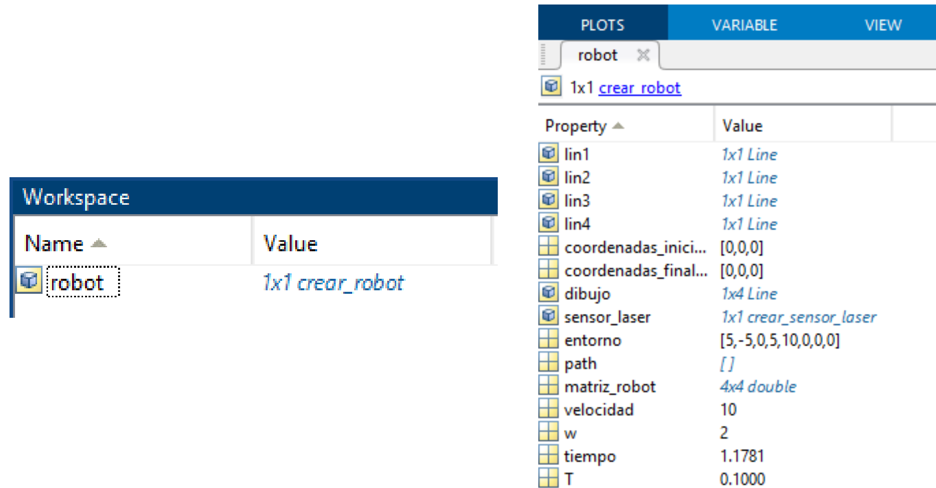


Figura 96. Ejemplo del Workspace.

Fuente. Propia.

9.6. Código

En este apartado se muestran las diferentes clases de esta arquitectura, así como los ficheros externos utilizados para ella.

9.6.1. Clase robot

```
1.     classdef crear_robot
2.         properties
3.             lin1
4.             lin2
5.             lin3
6.             lin4
7.             coordenadas_iniciales
8.             coordenadas_finales
9.             dibujo
10.
11.            sensor_laser
12.            entorno
13.            path
14.            matriz_robot
15.
16.            velocidad
17.            w
18.            tiempo
19.            T
```

```
20.             kr
21.             kw
22.             factor
23.             Max_velocidad
24.
25.             X_objetivo
26.             d_obj
27.             d_min
28.             d_ref
29.
30.             vector_final
31.             vector_objetivo
32.             vector_obstaculos
33.             vector_resultante
34.             vector_seguir_recto
35.             vector_orientacion
36.             vector_distancia
37.             vector_infinito
38.
39.             distancia_frontal
40.             distancia_derecha
41.             distancia_izquierda
42.             distancia_objetivo
43.             lado
44.             a
45.             b
46.             c
47.             d
48.             e
49.             f
50.             g
51.             angulos_especificos
52.         end
53.
54.         methods
55.             function obj = crear_robot()
56.                 obj.lin1 = line('xdata', [-0.2;0.8]', 'ydata', [0.4;0.4]',   'zdata', [0;0]', 'color', [0 0 1]);
57.                 obj.lin2 = line('xdata', [0.8;0.8]', 'ydata', [0.4;-0.4]', 'zdata', [0;0]', 'color', [1 0 0]);
58.                 obj.lin3 = line('xdata', [-0.2;0.8]', 'ydata', [-0.4;-0.4]', 'zdata', [0;0]', 'color', [0 0 1]);
59.                 obj.lin4 = line('xdata', [-0.2;-0.2]', 'ydata', [-0.4;0.4]', 'zdata', [0;0]', 'color', [0 0 1]);
60.                 obj.dibujo=[obj.lin1, obj.lin2, obj.lin3,obj.lin4];
61.
62.                 obj.coordenadas_iniciales=[0, 0, 0]; %se pueden modificar
63.                 obj.coordenadas_finales=[0, 0, 0];
64.                 obj.entorno= entorno_pared();
65.                 dibuent(obj.entorno, [0, 0, 0]);
66.                 obj.matriz_robot=XaMH(obj.coordenadas_iniciales);
67.
68.                 obj.velocidad =10;
69.                 obj.w =obj.velocidad/5;
70.                 obj.tiempo =3*pi/4/abs(obj.w);
71.                 obj.T=0.1;
72.
73.                 obj.sensor_laser=sensor_laser(obj); %crea un sensor láser
74.
75.                 obj=obj.mover_robot(obj.coordenadas_finales);
76.                 obj.kr=25;
77.                 obj.kw=5;
78.             end
79.
80.             function obj = mover_robot(obj, coordenadas_finales)
81.                 obj.coordenadas_finales= coordenadas_finales;
82.                 dibuagv(obj.dibujo, obj.coordenadas_iniciales, obj.coordenadas_finales);
```

```
83.             obj.coordenadas_finales(3)=mod(obj.coordenadas_finales(3), 2*pi);
84.             obj.coordenadas_iniciales=obj.coordenadas_finales;
85.             [obj.sensor_laser.coordenada_x_origen, obj.sensor_laser.coordenada_y_origen] =
        obj.sensor_laser.calculo_origen(obj);
86.             [obj.sensor_laser.posicion_sensor_abs,
        obj.sensor_laser.posicion_sensor_rel]=obj.sensor_laser.calcular_posicion_absoluta(obj);
87.             [obj.sensor_laser.direccion_x, obj.sensor_laser.direccion_y]= obj.sensor_laser.direccion_laser();
88.             obj.sensor_laser.matriz_sensor_laser = obj.sensor_laser.actualizar_matriz_sensor();
89.             %para probar la función de intersección sola, comentar
90.             %interseccion multiple y descomentar interseccion
91.             [obj.sensor_laser.puntos_interseccion, obj.sensor_laser.plotHandle] =
        obj.sensor_laser.calcular_interseccion_multiple(obj);
92.             %[obj.sensor_laser.puntos_interseccion(4), obj.sensor_laser.puntos_interseccion,
        obj.sensor_laser.plotHandle] = obj.sensor_laser.interseccion(obj);
93.             obj.matriz_robot=XaMH(obj.coordenadas_finales);
94.             pause(0.01);
95.
96.         end
97.
98.         function obj = simulacion_giro_robot(obj, velocidad, radio)
99.             obj.velocidad=velocidad;
100.            obj.w =obj.velocidad/radio;
101.            obj.tiempo =3*pi/4/abs(obj.w);
102.            obj.T=0.1;
103.            theta=obj.coordenadas_iniciales(3);
104.            obj.coordenadas_finales=obj.coordenadas_iniciales;
105.            dtheta=obj.w*obj.T;
106.            N=obj.tiempo/obj.T;
107.            obj.path=[];
108.
109.            for i =1:N
110.                obj.coordenadas_finales=obj.coordenadas_iniciales+[obj.velocidad * obj.T * cos(theta),
        obj.velocidad * obj.T*sin(theta), dtheta];
111.                theta=theta+dtheta;
112.                obj=obj.mover_robot(obj.coordenadas_finales);
113.                obj.path = [obj.path; obj.coordenadas_iniciales(1), obj.coordenadas_iniciales(2)];
114.                hold on;
115.                plot(obj.path(:,1), obj.path(:,2), 'b'); %dibujamos la trayectoria.
116.            end
117.        end
118.
119.        function obj = puntos_nuevos(obj, factor)
120.            filas_seleccionadas = 1:factor:size(obj.sensor_laser.puntos_interseccion, 1);
121.            obj.sensor_laser.puntos_interseccion = obj.sensor_laser.puntos_interseccion (filas_seleccionadas,
        :);
122.            x = obj.sensor_laser.puntos_interseccion(:, 1);
123.            y = obj.sensor_laser.puntos_interseccion(:, 2);
124.            delete(obj.sensor_laser.plotHandle)
125.            hold on;
126.            axis([-10 10 -10 10]);
127.            axis equal
128.            plot(x, y, 'bo', 'MarkerSize', 5);
129.        end
130.
131.        function obj = actualizar_entorno(obj)
132.            %crear nuevas lineas del robot
133.            obj.lin1 = line('xdata', [-0.2;0.8]', 'ydata', [0.4;0.4]', 'zdata', [0;0]', 'color', [0 0 1]);
134.            obj.lin2 = line('xdata', [0.8;0.8]', 'ydata', [0.4;-0.4]', 'zdata', [0;0]', 'color', [1 0 0]);
135.            obj.lin3 = line('xdata', [-0.2;0.8]', 'ydata', [-0.4;-0.4]', 'zdata', [0;0]', 'color', [0 0 1]);
136.            obj.lin4 = line('xdata', [-0.2;-0.2]', 'ydata', [-0.4;0.4]', 'zdata', [0;0]', 'color', [0 0 1]);
137.
138.            %actualizar la propiedad dibujo si es necesario
139.            obj.dibujo = [obj.lin1, obj.lin2, obj.lin3, obj.lin4];
```

```
140.
141.                %configuración inicial del robot y el sensor
142.                obj.sensor_laser.posicion_sensor_rel = [0, 0, 0];
143.                [obj.sensor_laser.lin1, obj.sensor_laser.lin2, obj.sensor_laser.lin3, obj.sensor_laser.lin4] =
    obj.sensor_laser.dibujar_laser();
144.                obj.sensor_laser.dibujo = [obj.sensor_laser.lin1, obj.sensor_laser.lin2, obj.sensor_laser.lin3,
    obj.sensor_laser.lin4];
145.                [obj.sensor_laser.coordenada_x_origen, obj.sensor_laser.coordenada_y_origen,
    obj.sensor_laser.posicion_sensor_abs, obj.sensor_laser.posicion_sensor_rel, obj.sensor_laser.direccion_x,
    obj.sensor_laser.direccion_y] = obj.sensor_laser.actualizar_laser(obj); % Crear el dibujo del láser
146.                obj.sensor_laser.matriz_sensor_laser = obj.sensor_laser.actualizar_matriz_sensor();
147.
148.                axis equal
149.                grid
150.            end
151.
152.            function obj = simulacion_objetivo(obj)
153.                clf
154.                %crear un nuevo robot desde 0
155.                obj = obj.actualizar_entorno();
156.
157.                %entorno simulacion
158.                obj.entorno=entorno0;
159.                dibuent(obj.entorno,[1 0 0]);
160.                axis([-20 20 0 40])
161.
162.
163.                %variables simulación
164.                obj.kr=25;
165.                obj.kw=5;
166.                obj.T = 0.01;
167.                obj.Max_velocidad=10; %límite de velocidad
168.                obj.d_min=0.1;%distancia del objetivo a la que para.
169.
170.                %coordenadas del robot
171.                obj.coordenadas_iniciales=[0,0,0]';
172.                obj.coordenadas_finales=[15,35, pi]'; %situacion inicial del AGV
173.                obj=obj.mover_robot(obj.coordenadas_finales);
174.
175.                %objetivo
176.                obj.X_objetivo = [-15 5]; %marco el objetivo
177.                line('xdata',[obj.X_objetivo(1)-1;obj.X_objetivo(1)+1],'ydata',[obj.X_objetivo(2)-
    1;obj.X_objetivo(2)+1],'color',[0 1 0]);
178.                line('xdata',[obj.X_objetivo(1)-
    1;obj.X_objetivo(1)+1],'ydata',[obj.X_objetivo(2)+1;obj.X_objetivo(2)-1],'color',[0 1 0]);
179.                obj.vector_final = [obj.X_objetivo 0 1]';
180.                obj.d_obj=sqrt((obj.coordenadas_finales(1)-obj.X_objetivo(1))^2+(obj.coordenadas_finales(2)-
    obj.X_objetivo(2))^2);%distancia al objetivo.
181.
182.                %comportamiento objetivo
183.                obj=obj.objetivo();
184.
185.            end
186.
187.            function [V_atraccion_objetivo,velocidad, distancia, w]=Vect_objetivo(obj)
188.
189.                V_objetivo = (obj.matriz_robot \ obj.vector_final)';
190.                V_objetivo=V_objetivo(1,1:2);
191.                velocidad=obj.kr*sqrt(V_objetivo(1)^2+V_objetivo(2)^2);
192.                distancia=velocidad;
193.
194.                %calcula el angulo de giro en un instante de tpo.
195.                ang=atan2(V_objetivo(2),V_objetivo(1));
```

```
196.            w=obj.kw*ang;
197.
198.            %limita la velocidad.
199.            if velocidad>obj.Max_velocidad
200.                V_atraccion_objetivo=[obj.Max_velocidad*cos(ang) obj.Max_velocidad*sin(ang)];
201.                velocidad=obj.Max_velocidad ;
202.            else
203.                V_atraccion_objetivo=V_objetivo;
204.            end
205.
206.        end
207.
208.        function obj= objetivo(obj)
209.            obj.path=[];
210.            while (obj.d_obj>obj.d_min)
211.                [obj.vector_objetivo, obj.velocidad, obj.d_obj, obj.w]=obj.Vect_objetivo();
212.
213.                %calcula la nueva posición y mueve el AGV
214.
obj.coordenadas_finales=obj.coordenadas_iniciales+[obj.velocidad*obj.T*cos(obj.coordenadas_iniciales(3)+obj.w*ob
j.T) obj.velocidad*obj.T*sin(obj.coordenadas_iniciales(3)+obj.w*obj.T) obj.w*obj.T]';
215.                obj=obj.mover_robot(obj.coordenadas_finales);
216.                obj.path=[obj.path; obj.coordenadas_iniciales(1) obj.coordenadas_iniciales(2)];
217.                hold on;
218.                plot(obj.path(:,1),obj.path(:,2), 'b'); %dibujamos la trayectoria.
219.            end
220.        end
221.
222.        function obj = simulacion_evitar_obstaculos(obj)
223.            clf
224.            %crear un nuevo robot desde 0
225.            obj = obj.actualizar_entorno();
226.
227.            %entorno simulacion
228.            axis([-20 20 0 40]);
229.            obj.entorno=entorno_evita_obstaculos;
230.            dibuent(obj.entorno, [1 0 0]);
231.
232.            %inicializar robot
233.            obj.coordenadas_iniciales = [0, 0, 0]';
234.            obj.coordenadas_finales = [15, 35, pi/2]'; % Situación inicial del AGV
235.            obj = obj.mover_robot(obj.coordenadas_finales);
236.
237.            %objetivo
238.            obj.X_objetivo = [-15 5]; %objetivo
239.            line('xdata', [obj.X_objetivo(1) - 1; obj.X_objetivo(1) + 1], 'ydata', [obj.X_objetivo(2) - 1;
obj.X_objetivo(2) + 1], 'color', [0 1 0]);
240.            line('xdata', [obj.X_objetivo(1) - 1; obj.X_objetivo(1) + 1], 'ydata', [obj.X_objetivo(2) + 1;
obj.X_objetivo(2) - 1], 'color', [0 1 0]);
241.
242.            %Variables simulación
243.            k_obj = 1.0; %peso comportamiento buscar objetivo
244.            k_obs = 1.0; %peso comportamiento evitar obstáculos
245.
246.            obj.kr = 10; %ganancia de velocidad
247.            obj.kw = 5; %ganancia de giro
248.            obj.T = 0.01; %encrementar el paso de tiempo
249.
250.            obj.Max_velocidad = 2; %limite de velocidad
251.            obj.d_min = 5; %distancia mínima al objetivo para detenerse
252.            obj.vector_final = [obj.X_objetivo 0 1]';
253.            obj.d_obj = sqrt((obj.coordenadas_finales(1) - obj.X_objetivo(1))^2 + (obj.coordenadas_finales(2) -
obj.X_objetivo(2))^2); % Distancia al objetivo.
```

```
254.
255.
256.             while (obj.d_obj > obj.d_min)
257.                 %vector comportamiento evitar obstáculos:
258.                 [obj.vector_obstaculos] = obj.evita_obstaculos();
259.                 %vector comportamiento atracción objetivo
260.                 [obj.vector_objetivo, obj.velocidad, obj.d_obj, obj.w] = obj.Vect_objetivo();
261.
262.                 %fusión de comportamientos como suma de vectores
263.                 obj.vector_resultante = k_obj * obj.vector_objetivo + k_obs * obj.vector_obstaculos;
264.
265.                 %normalización del vector resultante para evitar magnitudes extremas
266.                 if norm( obj.vector_resultante) > 0
267.                     obj.vector_resultante =  obj.vector_resultante / norm( obj.vector_resultante);
268.                 end
269.
270.                 %para que el AGV pueda ir hacia atrás
271.                 if  obj.vector_resultante(1) < 0
272.                     obj.velocidad = -obj.velocidad;
273.                 end
274.
275.                 %calcula la nueva velocidad y ángulo de giro
276.                 obj.velocidad = obj.kr * sqrt( obj.vector_resultante(1)^2 +  obj.vector_resultante(2)^2);
277.                 ang = atan2( obj.vector_resultante(2),  obj.vector_resultante(1));
278.                 w1 = obj.kw * ang;
279.
280.
281.                 %calcula la nueva posición y mueve el AGV
282.                 obj.coordenadas_finales = obj.coordenadas_iniciales + [obj.velocidad * obj.T *
cos(obj.coordenadas_iniciales(3) + w1 * obj.T), obj.velocidad * obj.T * sin(obj.coordenadas_iniciales(3) + w1 *
obj.T), w1 * obj.T]';
283.                 obj = obj.mover_robot(obj.coordenadas_finales);
284.                 obj.path = [obj.path;obj.coordenadas_iniciales(1), obj.coordenadas_iniciales(2)];
285.
286.             end
287.             hold on;
288.             plot(obj.path(:,1), obj.path(:,2),'b'); %dibujamos la trayectoria.
289.         end
290.
291.         function [v_obstaculos] = evita_obstaculos(obj)
292.             %angulos específicos y pesos asociados
293.             obj.angulos_especificos = [0, 330, 337, 315, 293, 300, 270, 250, 248, 225, 135, 113, 100, 90, 67,
45, 30, 10];
294.             % pesos de cada ángulo. Ajustado por prueba y error:
295.             k2 = 5.0; k3 = 0.5; k4 = 0.2; k5 = 0.1;
296.             %    0      330     337     315     293     300     270     250     248     225     135     113
100     90      67      45      30      10
297.             K = [k3,    k2,     k3,     k5,     k5,     k5,     k2,     k4,     k4,     k5,     k5,     k4,
k4,     k2,     k4,     k3,     k2,     k2];
298.             n = 2; %exponente para el cálculo de la repulsión
299.
300.             %filtrar y calcular vectores de repulsión
301.             v_obstaculos = [0, 0];
302.             for angulo_idx = 1:length(obj.angulos_especificos)
303.                 angulo_actual = obj.angulos_especificos(angulo_idx);
304.                 idx = find(abs(obj.sensor_laser.puntos_interseccion(:, 1) - angulo_actual) < 0.1);
305.                 if ~isempty(idx)
306.                     distancia = min(obj.sensor_laser.puntos_interseccion(idx, 4));  % Tomar la distancia más
corta para cada ángulo
307.                     if distancia < obj.sensor_laser.valor_maximo
308.                         vector_repulsion = -[cosd(angulo_actual), sind(angulo_actual)];
309.                         k = K(angulo_idx) / distancia^n;
310.                         v_obstaculos = v_obstaculos + k * vector_repulsion;
```

```
311.                     end
312.                 end
313.             end
314.
315.         end
316.
317.         function obj= simulacion_seguir_pared(obj)
318.             clf
319.             %crear un nuevo robot desde 0
320.             obj = obj.actualizar_entorno();
321.
322.             %entorno simulación
323.             axis([-10 30 0 40]);
324.             ent_pared=entorno_seguir_pared;
325.             obj.entorno=ent_pared;
326.             dibuent(ent_pared,[1 0 0]);
327.
328.             %variables simulación
329.             obj.T=0.01;
330.             obj.d_ref=2.5;
331.
332.
333.             obj.kr=15; %cte de velocidad
334.             obj.kw=5; %cte de rapidez de giro
335.
336.             %coordenadas del robot
337.             obj.coordenadas_iniciales=[0,0,0]';
338.             obj.coordenadas_finales=[10, 1, pi/2]'; %situacion inicial del AGV
339.             obj=obj.mover_robot(obj.coordenadas_finales);
340.
341.             obj.path=[];
342.
343.             %elección de pared (1= izquierda, 0=derecha)
344.             obj.lado=0;
345.
346.             while obj.coordenadas_iniciales(1)>0 && obj.coordenadas_iniciales(2)<40 &&
        obj.coordenadas_iniciales(1)<20
347.                 obj=obj.seguir_pared();
348.             end
349.         end
350.
351.         function obj = seguir_pared(obj)
352.             k1=1.0; %peso comportamiento seguir recto
353.             k2=0.5; %peso comportamiento orientación
354.             k4=0.5;
355.
356.             if obj.d_ref==0 || obj.d_ref<0
357.                 disp('No se puede poner el robot a esa distancia');
358.                 return;
359.             end
360.
361.             if obj.lado==0
362.                 obj.a=320;obj.b=270;obj.c=225; %sensores de la derecha
363.             elseif obj.lado==1
364.                 obj.a=40;obj.b=90;obj.c=135; %sensores de la izquierda
365.             end
366.
367.             [obj.sensor_laser.puntos_interseccion, obj.sensor_laser.plotHandle] =
        obj.sensor_laser.calcular_interseccion_multiple(obj);
368.             obj.angulos_especificos = [obj.a,obj.b,obj.c];
369.
370.
371.             %filtrar los puntos de intersección que corresponden a los ángulos específicos
```

```
372.             obj.sensor_laser.puntos_interseccion = obj.filtrar_puntos();
373.
374.             %verificar la distancia al objetivo más cercano
375.             dist_min = min(obj.sensor_laser.puntos_interseccion(:, 4));
376.
377.             if obj.lado==1
378.                 if dist_min >13
379.                     k2=0.01;
380.                 elseif dist_min >10
381.                     k2=0.02;
382.                 end
383.             end
384.
385.             if obj.lado==0
386.                 if dist_min >13
387.                     k2=0.01;
388.                 elseif dist_min >10
389.                     k2=0.02;
390.                 end
391.             end
392.
393.
394.             %creamos vector comportamiento seguir recto:
395.             obj.vector_seguir_recto=[1,0];
396.
397.             %creamos vector comportamiento orientación:
398.             [obj, obj.vector_orientacion]=obj.orientacion_pared();
399.             [obj.sensor_laser.puntos_interseccion, obj.sensor_laser.plotHandle] =
obj.sensor_laser.calcular_interseccion_multiple(obj);
400.             obj.angulos_especificos = [obj.a,obj.b,obj.c];
401.             obj.sensor_laser.puntos_interseccion = obj.filtrar_puntos();
402.             distancia_b=round(obj.sensor_laser.puntos_interseccion(2, 4), 3);
403.             %creamos vector comportamiento distancia:
404.             obj.vector_distancia=obj.distancia();
405.             if distancia_b==obj.d_ref
406.                 k4=0;
407.             end
408.             %creamos vector comportamiento evitar obstaculos:
409.             obj.vector_obstaculos=obj.evita_obstaculos();
410.
411.             v_pared=k1*obj.vector_seguir_recto+k2*obj.vector_orientacion+k4*obj.vector_obstaculos;
412.
413.
414.             obj.velocidad=obj.kr*sqrt(v_pared(1)^2+v_pared(2)^2);
415.             ang=atan2(v_pared(2),v_pared(1));
416.             obj.w=obj.kw*ang;
417.
418.             obj.coordenadas_finales = obj.coordenadas_iniciales + [obj.velocidad * obj.T *
cos(obj.coordenadas_iniciales(3) + obj.w * obj.T), obj.velocidad * obj.T * sin(obj.coordenadas_iniciales(3) +
obj.w * obj.T), obj.w * obj.T]';
419.             obj=obj.mover_robot(obj.coordenadas_finales);
420.             obj.path = [obj.path; obj.coordenadas_iniciales(1), obj.coordenadas_iniciales(2)];
421.             hold on;
422.             plot( obj.path(:,1),  obj.path(:,2), 'b'); %dibujamos la trayectoria.
423.         end
424.
425.         function puntos_filtrados = filtrar_puntos(obj)
426.             puntos_filtrados=[];
427.             for angulo_actual = obj.angulos_especificos
428.                 %encontrar índices cercanos al ángulo deseado
429.                 idx = find(abs(obj.sensor_laser.puntos_interseccion(:, 1) - angulo_actual) < 0.1);
430.                 angulo_anterior = NaN;
431.                 angulo_posterior = NaN;
```

```
432.                    idx_anterior = NaN;
433.                    idx_posterior = NaN;
434.
435.                    %verificar si alguno de los puntos encontrados tiene valores infinitos
436.                    if all(isinf(obj.sensor_laser.puntos_interseccion(idx, 2:3)))
437.                        %buscar el ángulo anterior y posterior válidos que no sean infinitos
438.                        if idx(1)==1081
439.                            for i = idx(1):-1:idx(1)-5
440.                                if ~any(isinf(obj.sensor_laser.puntos_interseccion(i, 2:3)))
441.                                    angulo_anterior = obj.sensor_laser.puntos_interseccion(i, 1);
442.                                    idx_anterior = i;
443.                                    break;
444.                                end
445.                            end
446.                            if isnan(idx_anterior)
447.                                angulo_anterior=NaN;
448.                                angulo_posterior=NaN;
449.                            else
450.                                for i=idx_anterior-1:-1:idx_anterior-5
451.                                    if ~any(isinf(obj.sensor_laser.puntos_interseccion(i, 2:3)))
452.                                        angulo_posterior = obj.sensor_laser.puntos_interseccion(i, 1);
453.                                        idx_posterior = i;
454.                                        break;
455.                                    end
456.                                end
457.                            end
458.                        elseif idx(1)==1
459.                            for i = idx(1):+1:idx(1)+5
460.                                if ~any(isinf(obj.sensor_laser.puntos_interseccion(i, 2:3)))
461.                                    angulo_anterior = obj.sensor_laser.puntos_interseccion(i, 1);
462.                                    idx_anterior = i;
463.                                    break;
464.                                end
465.                            end
466.                            if isnan(idx_anterior)
467.                                angulo_anterior=NaN;
468.                                angulo_posterior=NaN;
469.                            else
470.                                for i=idx_anterior+1:+1:idx_anterior+5
471.                                    if ~any(isinf(obj.sensor_laser.puntos_interseccion(i, 2:3)))
472.                                        angulo_posterior = obj.sensor_laser.puntos_interseccion(i, 1);
473.                                        idx_posterior = i;
474.                                        break;
475.                                    end
476.                                end
477.                            end
478.                        else
479.                            for i = idx(1):-1:idx(1)-5
480.                                if ~any(isinf(obj.sensor_laser.puntos_interseccion(i, 2:3)))
481.                                    angulo_anterior = obj.sensor_laser.puntos_interseccion(i, 1);
482.                                    idx_anterior = i;
483.                                    break;
484.                                end
485.                            end
486.
487.                            for i = idx(1):+1:idx(1)+5
488.                                if ~any(isinf(obj.sensor_laser.puntos_interseccion(i, 2:3)))
489.                                    angulo_posterior = obj.sensor_laser.puntos_interseccion(i, 1);
490.                                    idx_posterior = i;
491.                                    break;
492.                                end
493.                            end
494.                        end
```

```
495.                 end
496. 
497.                 %si se encuentran ambos ángulos válidos, interpolar
498.                 if ~isnan(angulo_anterior) && ~isnan(angulo_posterior)
499.                     punto_anterior = obj.sensor_laser.puntos_interseccion(idx_anterior, :);
500.                     punto_posterior = obj.sensor_laser.puntos_interseccion(idx_posterior, :);
501. 
502.                     %interpolación lineal entre los puntos anterior y posterior
503.                     peso_anterior = (angulo_posterior - angulo_actual) / (angulo_posterior - angulo_anterior);
504.                     peso_posterior = (angulo_actual - angulo_anterior) / (angulo_posterior - angulo_anterior);
505. 
506.                     punto_interpolado = peso_anterior * punto_anterior + peso_posterior * punto_posterior;
507. 
508.                     puntos_filtrados = [puntos_filtrados; punto_interpolado];
509.                 else
510.                     puntos_filtrados = [puntos_filtrados; obj.sensor_laser.puntos_interseccion(idx, :)];
511.                 end
512.             end
513.         end
514. 
515.         function [obj, vector] = orientacion_pared(obj)
516.             idx_a = abs(obj.sensor_laser.puntos_interseccion(:, 1) - obj.a) < 0.1;
517.             idx_b = abs(obj.sensor_laser.puntos_interseccion(:, 1) - obj.b) < 0.1;
518. 
519.             distancia_a = obj.sensor_laser.puntos_interseccion(idx_a, 4);
520.             distancia_b = obj.sensor_laser.puntos_interseccion(idx_b, 4);
521. 
522. 
523.             if distancia_a==Inf
524.                 distancia_a = 0;
525.             end
526. 
527.             if distancia_b==Inf
528.                 distancia_b = 0;
529.             end
530. 
531.             diferencia=distancia_a-distancia_b;
532.             diferencia_objetivo=distancia_b-obj.d_ref;
533. 
534.             if abs(diferencia_objetivo)<0.1
535.                 vector =[0,0];
536.                 if obj.coordenadas_finales(3) ~=pi/2
537.                     if obj.sensor_laser.puntos_interseccion(idx_b, 2) == 0
538.                         obj.coordenadas_finales(1)=abs(obj.d_ref-(obj.sensor_laser.puntos_interseccion(idx_b,
     2)));
539.                     else
540.                         obj.coordenadas_finales(1)=abs((obj.sensor_laser.puntos_interseccion(idx_b, 2))-
     obj.d_ref);
541.                         obj=obj.mover_robot(obj.coordenadas_finales);
542.                     end
543.                     obj=obj.girar_robot(90);
544.                 end
545.             else
546.                 vector = [0, abs(diferencia)];
547.                 if obj.lado==0
548.                     if distancia_b < obj.d_ref
549.                         if distancia_a == 0
550.                             vector = [0, -diferencia];
551.                         else
552.                             vector = [0, diferencia];
553.                         end
554.                     else
555.                         if distancia_a == 0
```

```
556.                             vector = [0, diferencia];
557.                         else
558.                             vector = [0, -diferencia];
559.                         end
560.                     end
561.                 elseif obj.lado==1
562. 
563.                     if distancia_b < obj.d_ref
564.                         if distancia_a == 0
565.                             vector = [0, diferencia];
566.                         else
567.                             vector = [0, -diferencia];
568.                         end
569.                     else
570.                         if distancia_a == 0
571.                             vector = [0, -diferencia];
572.                         else
573.                             vector = [0, diferencia];
574.                         end
575.                     end
576.                 end
577.             end
578.         end
579. 
580.         function vector = distancia(obj)
581.             idx_c = abs(obj.sensor_laser.puntos_interseccion(:, 1) - obj.c) < 0.1;
582.             distancia_c = obj.sensor_laser.puntos_interseccion(idx_c, 4);
583. 
584.             if isinf(distancia_c)
585.                 %si la distancia es infinita, el sensor no detecta nada, seguir recto
586.                 vector = [0, 0];
587.             else
588.                 %calcular el vector de distancia
589.                 vector = [0, (obj.d_ref-distancia_c)/distancia_c];
590. 
591.                 if obj.lado==0
592.                     vector = [0, -(obj.d_ref-distancia_c)/distancia_c];
593.                 end
594.             end
595. 
596.         end
597. 
598.         function obj = simulacion_seguir_pasillo(obj)
599.             clf
600.             %crear un nuevo robot desde 0
601.             obj = obj.actualizar_entorno();
602. 
603.             obj.entorno = entorno_pasillo;
604.             dibuent(obj.entorno, [1 0 0]);
605.             axis([-20 20 0 40]);
606. 
607.             %variables simulación
608.             obj.T = 0.01;
609. 
610.             obj.coordenadas_iniciales = [0, 0, 0]';
611.             obj.coordenadas_finales = [-6, 1, 0]'; %situación inicial del AGV
612.             obj = obj.mover_robot(obj.coordenadas_finales);
613. 
614.             obj.path = [];
615. 
616.             %cibujar el robot en pi/2
617.             if obj.coordenadas_finales(3)~=pi/2
618.                 obj.coordenadas_finales(3)=pi/2;
```

```
619.                    obj = obj.mover_robot(obj.coordenadas_finales);
620.                end
621.
622.                %comportamiento para seguir el pasillo
623.                While obj.coordenadas_iniciales(1) > -15 && obj.coordenadas_iniciales(2) < 40 &&
        obj.coordenadas_iniciales(1) < 10
624.                    obj=obj.seguir_pasillo();
625.                end
626.            end
627.
628.            function obj= seguir_pasillo(obj)
629.                k1 = 1.0; %peso comportamiento seguir recto
630.                k2 = 1.0; %peso comportamiento orientación
631.                obj.kr = 10; %constante de velocidad
632.                obj.kw = 5; %constante de rapidez de giro
633.
634.                %nombramiento de sensores
635.                obj.a = 320; obj.b = 270; obj.c = 225; %sensores de la derecha
636.                obj.d = 40; obj.e = 90; obj.f = 135; %sensores de la izquierda
637.
638.                obj.g =0; %sensor frontal
639.
640.                [obj.sensor_laser.puntos_interseccion, obj.sensor_laser.plotHandle] =
        obj.sensor_laser.calcular_interseccion_multiple(obj);
641.                obj.angulos_especificos = [obj.a, obj.b, obj.c, obj.d, obj.e, obj.f, obj.g];
642.                obj.sensor_laser.puntos_interseccion = obj.filtrar_puntos();
643.
644.                punto_b=obj.sensor_laser.puntos_interseccion(2,:);
645.                punto_e=obj.sensor_laser.puntos_interseccion(5,:);
646.                punto_g=obj.sensor_laser.puntos_interseccion(7,:);
647.
648.
649.                If (punto_b(4)==Inf && abs(obj.coordenadas_iniciales(3)-pi/2)<0.1 &&
        punto_g(3)~=Inf)||(punto_e(4)==Inf && abs(obj.coordenadas_iniciales(3)-pi/2)<0.1 && punto_g(3)~=Inf) %se
        comprueba si alguno de los dos sensores laterales da infinito y si el frontal detecta algo
650.                    if punto_e(4) == Inf
651.                        distancia=round((punto_g(3)-punto_b(3))/2);
652.                    elseif punto_b(4)==Inf
653.                        distancia=round((punto_g(3)-punto_e(3))/2);
654.                    end
655.
656.                    objetivo=punto_g(3)-distancia;
657.
658.                    while obj.coordenadas_iniciales(2) < objetivo
659.                        obj.vector_resultante = [1, 0];
660.                        obj.velocidad = obj.kr * sqrt(obj.vector_resultante(1)^2 + obj.vector_resultante(2)^2);
661.                        ang = atan2(obj.vector_resultante(2), obj.vector_resultante(1));
662.                        obj.w = obj.kw * ang;
663.                        obj.coordenadas_finales = obj.coordenadas_iniciales + [obj.velocidad * obj.T *
        cos(obj.coordenadas_iniciales(3) + obj.w * obj.T), obj.velocidad * obj.T * sin(obj.coordenadas_iniciales(3) +
        obj.w * obj.T), obj.w * obj.T]';
664.                        obj=obj.mover_robot(obj.coordenadas_finales);
665.                        obj.path = [obj.path; obj.coordenadas_iniciales(1), obj.coordenadas_iniciales(2)];
666.                        hold on;
667.                        plot(obj.path(:, 1), obj.path(:, 2), 'b'); % Dibujamos la trayectoria.
668.
669.                        [obj.sensor_laser.puntos_interseccion, obj.sensor_laser.plotHandle] =
        obj.sensor_laser.calcular_interseccion_multiple(obj);
670.                        obj.angulos_especificos = [obj.a, obj.b, obj.c, obj.d, obj.e, obj.f, obj.g];
671.                        obj.sensor_laser.puntos_interseccion = obj.filtrar_puntos();
672.                    end
673.                    obj.coordenadas_finales(2)=objetivo;
674.                    obj=obj.mover_robot(obj.coordenadas_finales);
```

```
675.                 obj=obj.girar_robot(0);
676.
677.             elseif punto_b(4)==Inf
678.                 while punto_b(4)==Inf
679.                     obj.vector_resultante = [1, 0];
680.                     obj.velocidad = obj.kr * sqrt(obj.vector_resultante(1)^2 + obj.vector_resultante(2)^2);
681.                     ang = atan2(obj.vector_resultante(2), obj.vector_resultante(1));
682.                     obj.w = obj.kw * ang;
683.                     obj.coordenadas_finales = obj.coordenadas_iniciales + [obj.velocidad * obj.T *
    cos(obj.coordenadas_iniciales(3) + obj.w * obj.T), obj.velocidad * obj.T * sin(obj.coordenadas_iniciales(3) +
    obj.w * obj.T), obj.w * obj.T]';
684.                     obj=obj.mover_robot(obj.coordenadas_finales);
685.                     obj.path = [obj.path; obj.coordenadas_iniciales(1), obj.coordenadas_iniciales(2)];
686.                     hold on;
687.                     plot(obj.path(:, 1), obj.path(:, 2), 'b'); % Dibujamos la trayectoria.
688.
689.                     [obj.sensor_laser.puntos_interseccion, obj.sensor_laser.plotHandle] =
    obj.sensor_laser.calcular_interseccion_multiple(obj);
690.                     obj.angulos_especificos = [obj.a, obj.b, obj.c, obj.d, obj.e, obj.f, obj.g];
691.                     obj.sensor_laser.puntos_interseccion = obj.filtrar_puntos();
692.                     punto_b=obj.sensor_laser.puntos_interseccion(2,:);
693.                 end
694.             else
695.
696.                 [obj, obj.vector_orientacion] = obj.orientacion_pasillo();
697.
698.                 %crear vector comportamiento seguir recto:
699.                 obj.vector_seguir_recto=[1,0];
700.
701.                 if norm(obj.vector_orientacion)>0 %para evitar magnitudes extremas
702.                     k2=0.05;
703.                 end
704.
705.                 obj.vector_resultante = k1 * obj.vector_seguir_recto + k2 * obj.vector_orientacion;
706.
707.                 obj.velocidad = obj.kr * sqrt(obj.vector_resultante(1)^2 + obj.vector_resultante(2)^2);
708.                 ang = atan2(obj.vector_resultante(2), obj.vector_resultante(1));
709.                 obj.w = obj.kw * ang;
710.                 obj.coordenadas_finales = obj.coordenadas_iniciales + [obj.velocidad * obj.T *
    cos(obj.coordenadas_iniciales(3) + obj.w * obj.T), obj.velocidad * obj.T * sin(obj.coordenadas_iniciales(3) +
    obj.w * obj.T), obj.w * obj.T]';
711.                 obj = obj.mover_robot(obj.coordenadas_finales);
712.
713.                 obj.path = [obj.path; obj.coordenadas_iniciales(1), obj.coordenadas_iniciales(2)];
714.                 hold on;
715.                 plot(obj.path(:, 1), obj.path(:, 2), 'b'); %dibujamos la trayectoria.
716.             end
717.         end
718.
719.         function [obj, vector] = orientacion_pasillo(obj)
720.
721.             punto_a=obj.sensor_laser.puntos_interseccion(1,:);
722.             punto_b=obj.sensor_laser.puntos_interseccion(2,:);
723.             punto_d=obj.sensor_laser.puntos_interseccion(4,:);
724.             punto_e=obj.sensor_laser.puntos_interseccion(5,:);
725.
726.             distancia=punto_b(4)-punto_e(4);
727.
728.             if abs(distancia) <0.08
729.                 if abs(obj.coordenadas_iniciales(3)-pi/2)<0.1 || obj.coordenadas_iniciales(3)<0.1
730.                     vector=[0,0];
731.                     obj.T=0.02;
732.                     obj.kw=0;
```

```
733.                     obj.w=0;
734.                 else
735.                     obj=obj.girar_robot(90);
736.                     [obj.sensor_laser.puntos_interseccion, obj.sensor_laser.plotHandle] =
     obj.sensor_laser.calcular_interseccion_multiple(obj);
737.                     obj.angulos_especificos = [obj.a, obj.b, obj.c, obj.d, obj.e, obj.f, obj.g];
738.                     obj.sensor_laser.puntos_interseccion = obj.filtrar_puntos();
739.                     punto_b=obj.sensor_laser.puntos_interseccion(2, :);
740.                     punto_e=obj.sensor_laser.puntos_interseccion(5, :);
741.
742.                     %calcula el centro del pasillo en x si las y de los puntos son iguales
743.                     if round(punto_b(3)) == round(punto_e(3))
744.                         if punto_b(2)<=0 %si las coordenadas son negativas
745.                             if punto_b(2) >punto_e(2)
746.                                 coordenada_x_centro=-(punto_b(2)-punto_e(2));
747.                             elseif punto_b(2) < punto_e(2)
748.                                 coordenada_x_centro=-(punto_e(2)-punto_b(2));
749.                             else
750.                                 coordenada_x_centro=punto_e(2);
751.                             end
752.                         elseif punto_b(2)>0 %si las coordenadas son positivas
753.                             if punto_b(2) >punto_e(2)
754.                                 if round(punto_e(2))== 0
755.                                     coordenada_x_centro=punto_b(2)/2;
756.                                 else
757.                                     coordenada_x_centro=punto_b(2)-punto_e(2);
758.                                 end
759.                             elseif punto_b(2) < punto_e(2)
760.                                 if round(punto_b(2))== 0
761.                                     coordenada_x_centro=punto_e(2)/2;
762.                                 else
763.                                     coordenada_x_centro=punto_e(2)-punto_b(2);
764.                                 end
765.                             else
766.                                 coordenada_x_centro=punto_e(2);
767.                             end
768.                         end
769.                     elseif round(punto_b(2)) == round(punto_e(2))
770.                         if punto_b(3)<=0 %si las coordenadas son negativas
771.                             if punto_b(3) >punto_e(3)
772.                                 coordenada_x_centro=-(punto_b(3)-punto_e(3));
773.                             elseif punto_b(3) < punto_e(3)
774.                                 coordenada_x_centro=-(punto_e(3)-punto_b(3));
775.                             else
776.                                 coordenada_x_centro=punto_e(3);
777.                             end
778.                         elseif punto_b(3)>0 %si las coordenadas son positivas
779.                             if punto_b(3) >punto_e(3)
780.                                 coordenada_x_centro=punto_b(3)-punto_e(3);
781.                             elseif punto_b(3) < punto_e(3)
782.                                 coordenada_x_centro=punto_e(3)-punto_b(3);
783.                             else
784.                                 coordenada_x_centro=punto_e(3);
785.                             end
786.                         end
787.                     end
788.
789.                     obj.coordenadas_finales(1)=coordenada_x_centro;
790.                     obj=obj.mover_robot(obj.coordenadas_finales);
791.                     obj.kr=30;
792.                     vector=[0,0];
793.                 end
794.             else
```

```
795.                if punto_b(4) < punto_e(4)  %si la distancia de la derecha es menor, se pega a la izquierda
796.                    if punto_e(4)==Inf
797.                        punto_e(4)=0;
798.                    end
799.
800.                    if punto_d(4)==Inf
801.                        punto_d(4)=0;
802.                    end
803.                    distancia = (punto_d(4) - punto_e(4));
804.
805.                    if punto_d(4) ==0
806.                        vector = [0, -(distancia)];
807.                    else
808.                        vector = [0, (distancia)];
809.                    end
810.
811.
812.                elseif punto_b(4) > punto_e(4) %si la distancia de la derecha es mayor, se pega a la derecha
813.                    if punto_a(4)==Inf
814.                        punto_a(4)=0;
815.                    end
816.
817.                    if punto_b(4)==Inf
818.                        punto_b(4)=0;
819.                    end
820.
821.                    distancia = punto_a(4) - punto_b(4);
822.                    vector = [0, -(distancia)];
823.                end
824.            end
825.        end
826.
827.
828.        function obj = simulacion_atravesar_puerta(obj)
829.            clf
830.            %crear un nuevo robot desde 0
831.            obj = obj.actualizar_entorno();
832.
833.            obj.entorno = entorno_puerta;
834.            dibuent(obj.entorno, [1 0 0]);
835.            axis([-20 10 0 40])
836.
837.
838.            %coordenadas iniciales del robot
839.            obj.coordenadas_iniciales = [0, 0, 0]';
840.            obj.coordenadas_finales = [-10, 1, pi/2]'; %situación inicial del AGV
841.            obj = obj.mover_robot(obj.coordenadas_finales);
842.            obj.path = [];
843.
844.            obj.T=0.01;
845.            obj.kw=7;
846.            obj.kr=10;
847.
848.            while (obj.coordenadas_finales(1)<5 && obj.coordenadas_finales(2)<30)
849.                obj=obj.atravesar_puerta();
850.            end
851.        end
852.
853.        function obj=atravesar_puerta(obj)
854.            k_inf=1.0;
855.            k_obs=0.5;
856.            k_seguir_recto=1.0;
857.            %vector comportamiento evitar obstáculos:
```

```
858.            [obj.vector_obstaculos] = obj.evita_obstaculos();
859.            %cector comportamiento atracción objetivo
860.            [obj.vector_infinito, obj.sensor_laser.puntos_interseccion]=obj.infinito();
861.            %cector comportamiento seguir recto
862.            obj.vector_seguir_recto=[1,0];
863.
864.            %fusión de comportamientos como suma de vectores
865.            obj.vector_resultante = k_seguir_recto*obj.vector_seguir_recto+ k_inf * obj.vector_infinito + k_obs
    * obj.vector_obstaculos;
866.
867.            %normalización del vector resultante para evitar magnitudes extremas
868.            if norm(obj.vector_resultante) > 0
869.                obj.vector_resultante = obj.vector_resultante / norm(obj.vector_resultante);
870.            end
871.
872.            %para que el AGV pueda ir hacia atrás
873.            if obj.vector_resultante(1) < 0
874.                obj.velocidad = -obj.velocidad;
875.            end
876.
877.            %calcula la nueva velocidad y ángulo de giro
878.            obj.velocidad = obj.kr * sqrt(obj.vector_resultante(1)^2 + obj.vector_resultante(2)^2);
879.            ang = atan2(obj.vector_resultante(2), obj.vector_resultante(1));
880.            obj.w = obj.kw * ang;
881.
882.
883.            %calcula la nueva posición y mueve el AGV
884.            obj.coordenadas_finales = obj.coordenadas_iniciales + [obj.velocidad * obj.T *
    cos(obj.coordenadas_iniciales(3) + obj.w * obj.T), obj.velocidad * obj.T * sin(obj.coordenadas_iniciales(3) +
    obj.w * obj.T), obj.w * obj.T]';
885.            obj = obj.mover_robot(obj.coordenadas_finales);
886.            obj.path = [obj.path; obj.coordenadas_iniciales(1), obj.coordenadas_iniciales(2)];
887.
888.            hold on;
889.            plot(obj.path(:,1), obj.path(:,2), 'b');
890.        end
891.
892.        function [vector, puntos_filtrados]=infinito(obj)
893.            %definir los ángulos específicos
894.            obj.angulos_especificos = [0, 330, 337, 315, 293, 300, 270, 250, 248, 225, 135, 113, 100, 90, 67,
    45, 30, 10];
895.
896.            puntos_filtrados=obj.filtrar_puntos();
897.
898.            N = size(puntos_filtrados, 1);
899.            vec_infinito = zeros(N, 2);
900.
901.            %pesos de cada ángulo. Ajustado por prueba y error:
902.            k1= 3.0; k2 = 1.0; k3 = 0.5; k4 = 0.2; k5 = 0.1;
903.    %    0      330     337     315     293     300     270     250     248     225     135     113     100     90
    67      45      30      10
904.            K = [k1,    k2,     k2,     k3,     k3,     k5,     k2,     k3,     k3,     k5,     k5,     k4,
    k4,     k2,     k4,     k3,     k2,     k2];
905.
906.            %indices específicos para el cálculo
907.            indices_especificos = [1:4, 16:18];
908.
909.            %inicializar el vector resultado
910.            vector = [0 0];
911.
912.            %calcular el vector de atracción hacia el infinito solo para índices seleccionados y distancia
    infinita
913.            for i = indices_especificos
```

```
914.                if i <= N && isinf(puntos_filtrados(i,4)) % verificar que el índice esté dentro del rango y la
        distancia sea infinita
915.                    angulo_rad = puntos_filtrados(i,1) * pi / 180; %convertir ángulo de grados a radianes
916.                    vec_infinito(i,:) = [cos(angulo_rad), sin(angulo_rad)];
917.                    vector = vector + K(i) * vec_infinito(i,:);
918.                end
919.            end
920.        end
921.
922.        function obj=simulacion_aparcar(obj)
923.            clf
924.            obj=obj.actualizar_entorno();
925.
926.            obj.entorno = entorno_aparcar;
927.            dibuent(obj.entorno, [1 0 0]);
928.            axis([-20 5 0 25])
929.
930.            %coordenadas iniciales del robot
931.            obj.coordenadas_iniciales = [0, 0, 0]';
932.            obj.coordenadas_finales = [-10, 1, pi/2]'; %situación inicial del AGV
933.            obj = obj.mover_robot(obj.coordenadas_finales);
934.            obj.path = [];
935.
936.            obj.T=0.01;
937.            obj.kw=7;
938.            obj.kr=10;
939.
940.            obj.X_objetivo=[-9, 13];
941.            obj.vector_final = [obj.X_objetivo 0 1]';
942.
943.
944.            obj.distancia_frontal = 1;
945.            obj.distancia_izquierda = 2;
946.            obj.distancia_derecha= 2;
947.
948.            obj=obj.aparcar();
949.        end
950.
951.        function obj = aparcar(obj)
952.            [izquierda, frontal, derecha]=vector_aparcamiento(obj);
953.            plot([izquierda(1), izquierda(3)], [izquierda(2), izquierda(4)], '--', 'Color', 'black' );
954.            plot([frontal(1), frontal(3)], [frontal(2), frontal(4)], '--', 'Color', 'black');
955.            plot([derecha(1), derecha(3)], [derecha(2), derecha(4)], '--', 'Color', 'black');
956.
957.            obj.X_objetivo(2)= izquierda(2)-2;
958.            obj.vector_final = [obj.X_objetivo 0 1]';
959.            obj.Max_velocidad=10; %límite de velocidad
960.            obj.d_min=0.1;%distancia del objetivo a la que para.
961.            obj.d_obj=sqrt((obj.coordenadas_finales(1)-obj.X_objetivo(1))^2+(obj.coordenadas_finales(2)-
        obj.X_objetivo(2))^2);%distancia al objetivo.
962.
963.            obj=obj.objetivo();
964.
965.            hold on;
966.            plot(obj.path(:,1),obj.path(:,2), 'b');
967.            obj=obj.girar_robot(90);
968.            while obj.coordenadas_finales(2)<abs(frontal(2)-1)
969.                obj.vector_resultante=[1,0];
970.                obj.velocidad = obj.kr * sqrt(obj.vector_resultante(1)^2 + obj.vector_resultante(2)^2);
971.                ang = atan2(obj.vector_resultante(2), obj.vector_resultante(1));
972.                obj.w = obj.kw * ang;
973.
974.                %calcula la nueva posición y mueve el AGV
```

```
975.                obj.coordenadas_finales = obj.coordenadas_iniciales + [obj.velocidad * obj.T *
        cos(obj.coordenadas_iniciales(3) + obj.w * obj.T), obj.velocidad * obj.T * sin(obj.coordenadas_iniciales(3) +
        obj.w * obj.T), obj.w * obj.T]';
976.                obj=obj.mover_robot(obj.coordenadas_finales);
977.                obj.path = [obj.path; obj.coordenadas_iniciales(1), obj.coordenadas_iniciales(2)];
978.                hold on;
979.                plot(obj.path(:,1), obj.path(:,2), 'b');
980.            end
981.        end
982.
983.        function [izquierda, frontal, derecha]= vector_aparcamiento(obj)
984.
985.            frontal1=[obj.X_objetivo(1)-obj.distancia_izquierda,obj.X_objetivo(2)+obj.distancia_frontal];
986.            frontal2=[obj.X_objetivo(1)+obj.distancia_derecha,obj.X_objetivo(2)+obj.distancia_frontal];
987.
988.            izquierda2=[obj.X_objetivo(1)-obj.distancia_izquierda, frontal1(2)];
989.            izquierda1=[obj.X_objetivo(1)-obj.distancia_izquierda, izquierda2(2)-2.5];
990.
991.            derecha2=[obj.X_objetivo(1)+obj.distancia_derecha,frontal1(2)];
992.            derecha1=[obj.X_objetivo(1)+obj.distancia_derecha,  derecha2(2)-2.5];
993.
994.            izquierda = [izquierda1, izquierda2];
995.            derecha =   [derecha1, derecha2];
996.            frontal =   [frontal1, frontal2];
997.        end
998.
999.        function obj = piloto(obj)
1000.           clf
1001.           %declararar el entorno:
1002.           obj.entorno=entorno_final;
1003.           %dibujarlo:
1004.           dibuent(obj.entorno,'r');
1005.           [nodo1, nodo2, nodo3, nodo4, nodo5, nodo6, nodo7, nodo8, nodo9]=Dibuja_nodos;
1006.
1007.           obj.coordenadas_iniciales=[0,0,0];
1008.           obj.coordenadas_finales=nodo1.Position;
1009.
1010.           obj=obj.actualizar_entorno();
1011.
1012.           obj = obj.mover_robot(obj.coordenadas_finales);
1013.           obj.coordenadas_finales=obj.coordenadas_finales';
1014.           obj.coordenadas_iniciales=obj.coordenadas_iniciales';
1015.
1016.           m_costes;
1017.           m_tipos;
1018.           [camino, ~]=Dijkstra(matriz_costes, 1, 9);
1019.
1020.           for i = 1:length(camino)-1
1021.               coordenadas=[camino(i), camino(i+1)];
1022.               comportamiento=matriz_tipos(coordenadas(1), coordenadas(2));
1023.               switch (coordenadas(2))
1024.                   case 1
1025.                       objetivo=nodo1.Position;
1026.                   case 2
1027.                       objetivo=nodo2.Position;
1028.                       if comportamiento ~=1
1029.                           disp('El comportamiento no es el adecuado en esta situación')
1030.                           return;
1031.                       end
1032.                   case 3
1033.                       objetivo=nodo3.Position;
1034.                       if comportamiento ~=2
1035.                           disp('El comportamiento no es el adecuado en esta situación')
```

```
1036.                           return;
1037.                       end
1038.                   case 4
1039.                       objetivo=nodo4.Position;
1040.                   case 5
1041.                       objetivo=nodo5.Position;
1042.                   case 6
1043.                       objetivo=nodo6.Position;
1044.                   case 7
1045.                       objetivo=nodo7.Position;
1046.                   case 8
1047.                       if comportamiento ==2
1048.                           disp('El comportamiento no es el adecuado en esta situación')
1049.                           return;
1050.                       end
1051.                       objetivo=nodo8.Position;
1052.                   case 9
1053.                       if comportamiento ~=1 && comportamiento ~=6
1054.                           disp('El comportamiento no es el adecuado en esta situación')
1055.                           return;
1056.                       end
1057.                       objetivo=nodo9.Position;
1058.                   otherwise
1059.                       disp('Nodo no encontrado');
1060.               end
1061.
1062.               switch(comportamiento)
1063.                   case 1
1064.                       disp('objetivo');
1065.                       obj.T = 0.02;
1066.                       obj.kr=10;
1067.                       obj.kw=5;
1068.                       obj.vector_final = [objetivo(1) objetivo(2) 0 1]';
1069.
1070.                       obj.Max_velocidad=10; %límite de velocidad
1071.                       obj.d_min=0.05;%distancia del objetivo a la que para.
1072.                       obj.d_obj=sqrt((obj.coordenadas_finales(1)-objetivo(1))^2+(obj.coordenadas_finales(2)-
objetivo(2))^2);%Ddistancia al objetivo.
1073.
1074.                       obj=obj.objetivo();
1075.
1076.                   case 2
1077.                       disp('atravesar puertas');
1078.                       obj.T=0.02;
1079.                       obj.kw=5;
1080.                       obj.kr=10;
1081.                       diferencia_1=obj.coordenadas_finales(1)-objetivo(1);
1082.
1083.                       while (abs(diferencia_1)>0.1)
1084.                           obj=obj.atravesar_puerta();
1085.                           diferencia_1=obj.coordenadas_finales(1)-objetivo(1);
1086.                           obj.path = [obj.path; obj.coordenadas_iniciales(1), obj.coordenadas_iniciales(2)];
1087.                           hold on;
1088.                           plot(obj.path(:,1), obj.path(:,2), 'b');
1089.                       end
1090.                       obj=obj.girar_robot(90);
1091.
1092.
1093.                   case 3
1094.                       disp('pared derecha');
1095.                       obj.lado=0;
1096.                       obj.path=[];
1097.                       obj.T=0.02;
```

```
1098.                        obj.d_ref=4;
1099.
1100.                        obj.kr=10; %cte de velocidad
1101.                        obj.kw=5; %cte de rapidez de giro
1102.
1103.                        diferencia_y=abs(obj.coordenadas_finales(2)- objetivo(2));
1104.
1105.                        while diferencia_y>1
1106.                            obj=obj.seguir_pared();
1107.                            diferencia_y=abs(obj.coordenadas_finales(2)- objetivo(2));
1108.                        end
1109.
1110.                    case 4
1111.                        disp('pared izquierda');
1112.                        obj.lado=1;
1113.                        obj.path=[];
1114.                        obj.T=0.02;
1115.                        obj.d_ref=3;
1116.
1117.                        obj.kr=10; %cte de velocidad
1118.                        obj.kw=5; %cte de rapidez de giro
1119.
1120.                        diferencia_y=abs(obj.coordenadas_finales(2)- objetivo(2));
1121.
1122.                        while diferencia_y>1
1123.                            obj=obj.seguir_pared();
1124.                            diferencia_y=abs(obj.coordenadas_finales(2)- objetivo(2));
1125.                        end
1126.
1127.
1128.                    case 5
1129.                        disp('Seguir pasillo');
1130.                        diferencia_y=abs(obj.coordenadas_finales(2)- objetivo(2));
1131.
1132.                        while diferencia_y>1
1133.                            obj=obj.seguir_pasillo();
1134.                            diferencia_y=abs(obj.coordenadas_finales(2)- objetivo(2));
1135.                        end
1136.
1137.                    case 6
1138.                        disp('Aparcar')
1139.                        obj.T=0.02;
1140.                        obj.kw=5;
1141.                        obj.kr=10;
1142.
1143.                        obj.X_objetivo=[objetivo(1), objetivo(2)];
1144.                        obj.vector_final = [obj.X_objetivo 0 1]';
1145.
1146.
1147.                        obj.distancia_frontal = 1;
1148.                        obj.distancia_izquierda = 2;
1149.                        obj.distancia_derecha= 2;
1150.
1151.                        obj=obj.aparcar();
1152.                    otherwise
1153.                        break;
1154.                end
1155.
1156.            end
1157.
1158.        end
1159.
1160.        function obj=girar_robot(obj, grados)
```

```
1161.              diferencia=obj.coordenadas_iniciales(3)-deg2rad(grados);
1162.
1163.              while abs(diferencia)>0.1
1164.                  T2=0.1;
1165.                  obj.w=0.3;
1166.                  %calcular las nuevas coordenadas
1167.                  if diferencia>0
1168.                      obj.coordenadas_finales(3) = obj.coordenadas_iniciales(3) - obj.w *T2;
1169.                  elseif diferencia<0
1170.                      obj.coordenadas_finales(3) = obj.coordenadas_iniciales(3) + obj.w *T2;
1171.                  end
1172.                  obj=obj.mover_robot(obj.coordenadas_finales);
1173.                  obj.path = [obj.path; obj.coordenadas_iniciales(1), obj.coordenadas_iniciales(2)];
1174.                  hold on;
1175.                  plot(obj.path(:,1), obj.path(:,2), 'b'); %dibujamos la trayectoria.
1176.                  diferencia=obj.coordenadas_iniciales(3)-deg2rad(grados);
1177.              end
1178.              obj.coordenadas_finales(3)=deg2rad(grados);
1179.              obj=obj.mover_robot(obj.coordenadas_finales);
1180.              obj.sensor_laser.puntos_interseccion = obj.filtrar_puntos();
1181.          end
1182.
1183.      end
1184.  end
```

9.6.2. Clase sensor láser

```
1.      classdef crear_sensor_laser
2.          properties
3.              angulo_inicial
4.              angulo_final
5.              resolucion
6.              velocidad
7.              valor_minimo
8.              valor_maximo
9.              posicion_sensor_rel
10.             posicion_sensor_abs
11.             matriz_sensor_laser
12.             lin1
13.             lin2
14.             lin3
15.             lin4
16.             dibujo
17.             coordenada_x_origen
18.             coordenada_y_origen
19.             direccion_x
20.             direccion_y
21.             distancia
22.             punto
23.             puntos_interseccion
24.             robot
25.             plotHandle
26.             angulo
27.             haz_lines
28.         end
29.
30.         methods
```

```
31.         function obj = crear_sensor_laser(robot)
32.             obj.angulo_inicial = 0;
33.             obj.angulo_final = 270;
34.             obj.resolucion = 0.25;
35.             obj.velocidad = 25;
36.             obj.valor_minimo = 0.06;
37.             obj.valor_maximo = 20;
38.             obj.posicion_sensor_rel = [0, 0, 0]; % Modificar posición relativa
39.             obj.robot = robot;
40.             obj.puntos_interseccion=[];
41.
42.             obj.lin1 = line('xdata', [-0.1;0.1]', 'ydata', [0.1;0.1]', 'zdata', [0;0]', 'color', [0 1 0]);
43.             obj.lin2 = line('xdata', [0.1;0.1]', 'ydata', [0.1;-0.1]', 'zdata', [0;0]', 'color', [1 0 0]);
44.             obj.lin3 = line('xdata', [-0.1;0.1]', 'ydata', [-0.1;-0.1]', 'zdata', [0;0]', 'color', [0 1 0]);
45.             obj.lin4 = line('xdata', [-0.1;-0.1]', 'ydata', [-0.1;0.1]', 'zdata', [0;0]', 'color', [0 1 0]);
46.             obj.dibujo = [obj.lin1, obj.lin2, obj.lin3, obj.lin4];
47.             obj.haz_lines = []; %inicializar la propiedad haz_lines
48.
49.             %calcular la posición absoluta del sensor y actualizar la matriz
50.             [obj.coordenada_x_origen, obj.coordenada_y_origen] = obj.calculo_origen(robot);
51.             [obj.posicion_sensor_abs, obj.posicion_sensor_rel] = obj.calcular_posicion_absoluta(robot);
52.             [obj.direccion_x, obj.direccion_y] = obj.direccion_laser();
53.             obj.matriz_sensor_laser = obj.actualizar_matriz_sensor();
54.             %si se quiere probar la función de intersección sola, comentar
55.             %la siguiente línea de código:
56.             [obj.puntos_interseccion, obj.plotHandle] = obj.calcular_interseccion_multiple(robot);
57.         end
58.
59.         function [origen_x, origen_y] = calculo_origen(~, robot)
60.             %obtener las coordenadas de los extremos de las cuatro líneas
61.             xdata1 = get(robot.lin1, 'XData');
62.             ydata1 = get(robot.lin1, 'YData');
63.
64.             xdata2 = get(robot.lin2, 'XData');
65.             ydata2 = get(robot.lin2, 'YData');
66.
67.             xdata3 = get(robot.lin3, 'XData');
68.             ydata3 = get(robot.lin3, 'YData');
69.
70.             xdata4 = get(robot.lin4, 'XData');
71.             ydata4 = get(robot.lin4, 'YData');
72.
73.             %calcular el punto medio de cada línea
74.             x_mid1 = mean(xdata1);
75.             y_mid1 = mean(ydata1);
76.
77.             x_mid2 = mean(xdata2);
78.             y_mid2 = mean(ydata2);
79.
80.             x_mid3 = mean(xdata3);
81.             y_mid3 = mean(ydata3);
82.
83.             x_mid4 = mean(xdata4);
84.             y_mid4 = mean(ydata4);
85.
86.             %calcular el punto medio del cuadrilátero
87.             origen_x = (x_mid1 + x_mid2 + x_mid3 + x_mid4) / 4;
88.             origen_y = (y_mid1 + y_mid2 + y_mid3 + y_mid4) / 4;
89.         end
```

```
90.
91.            function [posicion_abs, posicion_rel] = calcular_posicion_absoluta(obj, robot)
92.                coordenada_abs_x = obj.coordenada_x_origen;
93.                coordenada_abs_y = obj.coordenada_y_origen;
94.
95.                posicion_abs = [coordenada_abs_x, coordenada_abs_y, robot.coordenadas_iniciales(3)];
96.                dibuagv(obj.dibujo, obj.posicion_sensor_rel, posicion_abs);
97.                posicion_rel = posicion_abs;
98.            end
99.
100.           function matriz = actualizar_matriz_sensor(obj)
101.               matriz = [obj.posicion_sensor_abs, obj.valor_maximo];
102.           end
103.
104.           function [direccionx, direcciony] = direccion_laser(obj)
105.               direccionx = 0;
106.               direcciony = 0;
107.               xdata1 = get(obj.lin1, 'XData');
108.               ydata1 = get(obj.lin1, 'YData');
109.
110.               xdata2 = get(obj.lin2, 'XData');
111.               ydata2 = get(obj.lin2, 'YData');
112.
113.               lin1_x1 = xdata1(1);
114.               lin1_x2 = xdata1(2);
115.
116.               lin1_y1 = ydata1(1);
117.               lin1_y2 = ydata1(2);
118.
119.               lin2_x1 = xdata2(1);
120.               lin2_x2 = xdata2(2);
121.
122.               lin2_y1 = ydata2(1);
123.               lin2_y2 = ydata2(2);
124.
125.               %verificar si las líneas son verticales
126.               if lin2_x1 == lin2_x2
127.                   direcciony = 0;
128.                   %verificar la posición relativa en el eje x
129.                   if lin1_x1 < lin2_x1 || lin1_x2 < lin2_x1
130.                       direccionx = 1;
131.                   elseif lin1_x1 > lin2_x1 || lin1_x2 > lin2_x1
132.                       direccionx = -1;
133.                   end
134.               %verificar si las líneas son horizontales
135.               elseif lin2_y1 == lin2_y2
136.                   direccionx = 0;
137.                   %verificar la posición relativa en el eje y
138.                   if lin1_y1 < lin2_y1 || lin1_y2 < lin2_y1
139.                       direcciony = 1;
140.                   elseif lin1_y1 > lin2_y1 || lin1_y2 > lin2_y1
141.                       direcciony = -1;
142.                   end
143.               else
144.                   if lin1_x1 < lin2_x1 || lin1_x2 < lin2_x1
145.                       direccionx = 1;
146.                   elseif lin1_x1 > lin2_x1 || lin1_x2 > lin2_x1
147.                       direccionx = -1;
148.                   end
```

```
149.                if lin1_y1 < lin2_y1 || lin1_y2 < lin2_y1
150.                    direcciony = 1;
151.                elseif lin1_y1 > lin2_y1 || lin1_y2 > lin2_y1
152.                    direcciony = -1;
153.                end
154.            end
155.        end
156.
157.        function [puntos_interseccion, plotHandle] = calcular_interseccion_multiple(obj, robot)
158.            %inicializar una lista para almacenar los puntos de intersección
159.            puntos_interseccion = [];
160.            plotHandle = [];
161.            hold on;
162.
163.            %limpiar las líneas del haz existentes antes de dibujar nuevas
164.            if ~isempty(obj.haz_lines)
165.                delete(obj.haz_lines);
166.                obj.haz_lines = [];
167.            end
168.
169.            %iterar sobre los ángulos dentro del rango angular
170.            for angulo_actual = obj.angulo_inicial:obj.resolucion:obj.angulo_final
171.                obj.angulo = angulo_actual;
172.                %calcular la dirección del láser
173.                angulo_ajustado = mod(225 + angulo_actual, 360); %ajustar el ángulo según la orientación del sensor
174.                obj.direccion_x = cosd(angulo_ajustado + rad2deg(robot.coordenadas_finales(3)));
175.                obj.direccion_y = sind(angulo_ajustado + rad2deg(robot.coordenadas_finales(3)));
176.                %calcular la intersección del haz láser con las líneas del entorno
177.                [obj.distancia, obj.punto, plotHandles] = obj.interseccion(robot);
178.                plotHandle = [plotHandle, plotHandles];
179.                puntos_interseccion    =    [puntos_interseccion;    angulo_ajustado,    obj.punto(1),    obj.punto(2),
     obj.distancia];
180.
181.                % Dibujar el haz del sensor láser
182.                   % x_data = [obj.posicion_sensor_abs(1), obj.posicion_sensor_abs(1) + 20 * cosd(angulo_ajustado +
     rad2deg(robot.coordenadas_iniciales(3)))];
183.                   % y_data = [obj.posicion_sensor_abs(2), obj.posicion_sensor_abs(2) + 20 * sind(angulo_ajustado +
     rad2deg(robot.coordenadas_iniciales(3)))];
184.                   % haz_line = plot(x_data, y_data, 'b-', 'LineWidth', 0.1); % Línea azul fina
185.                   % obj.haz_lines = [obj.haz_lines, haz_line]; % Almacenar el handle de la línea del haz
186.            end
187.        end
188.
189.        function [distancia, punto, plotHandles] = interseccion(obj, robot)
190.            %extraer las coordenadas del sensor láser desde la matriz de transformación homogénea
191.            sensor =  mh_sensor(obj.matriz_sensor_laser);
192.
193.            %inicializar la distancia y el punto de intersección como infinito
194.            distancia = Inf;
195.            punto = [Inf; Inf];
196.            plotHandles = [];
197.
198.            %iterar sobre las líneas del entorno
199.            for i = 1:size(robot.entorno, 1)
200.                %extraer las características de la línea
201.                px1 = robot.entorno(i, 1); %coordenada X del inicio de la línea
202.                py1 = robot.entorno(i, 2); %coordenada Y del inicio de la línea
203.                px2 = robot.entorno(i, 4); %coordenada X del final de la línea
204.                py2 = robot.entorno(i, 5); %coordenada Y del final de la línea
205.
```

```
206.                    %calcular t para obtener el punto de intersección
207.                    if (px2 - px1) * obj.direccion_x >= 0 || (py2 - py1) * obj.direccion_y >= 0
208.                        t = ((px1 - sensor(1, 4)) * (py2 - py1) - (py1 - sensor(2, 4)) * (px2 - px1)) / (obj.direccion_x *
    (py2 - py1) - obj.direccion_y * (px2 - px1));
209.                        if t >= 0
210.                            %calcular las coordenadas del punto de intersección
211.                            punto_interseccion = sensor(1:2, 4) + t * [obj.direccion_x; obj.direccion_y];
212.                            %verificar si el punto de intersección está dentro del segmento de la línea
213.                            if punto_interseccion(1) >= min(px1, px2) && punto_interseccion(1) <= max(px1, px2) &&
    punto_interseccion(2) >= min(py1, py2) && punto_interseccion(2) <= max(py1, py2)
214.                                %calcular la distancia desde el sensor al punto de intersección
215.                                distancia_temp = norm(punto_interseccion - sensor(1:2, 4));
216.                                if distancia_temp < distancia
217.                                    if distancia_temp > obj.valor_maximo
218.                                        distancia=Inf;
219.                                        punto(1)=Inf;
220.                                        punto(2)=Inf;
221.                                    else
222.                                    distancia = distancia_temp;
223.                                    punto = punto_interseccion;
224.                                    end
225.                                    % hold on;
226.                                    %plotHandles = [plotHandles, plot(punto(1), punto(2), 'ro', 'MarkerSize', 5)]; %punto
    rojo
227.                                    % hold off;
228.                                end
229.                            end
230.                        end
231.                    end
232.                end
233.            end
234.
235.
236.            function [lin1, lin2, lin3, lin4] = dibujar_laser(~)
237.                lin1 = line('xdata', [-0.1;0.1]', 'ydata', [0.1;0.1]', 'zdata', [0;0]', 'color', [0 1 0]);
238.                lin2 = line('xdata', [0.1;0.1]', 'ydata', [0.1;-0.1]', 'zdata', [0;0]', 'color', [1 0 0]);
239.                lin3 = line('xdata', [-0.1;0.1]', 'ydata', [-0.1;-0.1]', 'zdata', [0;0]', 'color', [0 1 0]);
240.                lin4 = line('xdata', [-0.1;-0.1]', 'ydata', [-0.1;0.1]', 'zdata', [0;0]', 'color', [0 1 0]);
241.            end
242.
243.            function [coordenada_x, coordenada_y, posicion_abs, posicion_rel, direccion_x, direccion_y] =
    actualizar_laser(obj, robot)
244.                [coordenada_x, coordenada_y] = obj.calculo_origen(robot);
245.                [posicion_abs, posicion_rel] = obj.calcular_posicion_absoluta(robot);
246.                [direccion_x, direccion_y] = obj.direccion_laser();
247.            end
248.
249.
250.
251.        end
252.    end
```

9.6.3. Main

```
1.    %fichero de pruebas
2.        clf
```

```
3.        axis([-10 10 -10 10])
4.        axis equal
5.        hold on
6.        robot=crear_robot();
7.
8.        %robot=mover_robot(robot, [1, 1, pi/2]);
9.        %robot=simulacion_giro_robot(robot, 0.5, 1);
10.       %robot=girar_robot(robot, 90);
11.
12.       %robot=puntos_nuevos(robot, 8);
13.
14.       %robot=simulacion_objetivo(robot);
15.       %robot=simulacion_evitar_obstaculos(robot);
16.       %robot=simulacion_seguir_pared(robot);
17.       %robot=simulacion_seguir_pasillo(robot);
18.       %robot=simulacion_atravesar_puerta(robot);
19.       %robot=simulacion_aparcar(robot);
20.
21.       %robot = piloto(robot);
```

9.6.4. Algoritmo de Dijkstra y relacionados

- **Algoritmo de Dijkstra**

```
1.    function [Camino, Coste]=Dijkstra(matriz_costes,nodo_inicial,nodo_final)
2.
3.    N=size(matriz_costes,1);
4.    cerrados=  nodo_inicial;
5.    abiertos=[];
6.    costes=[];
7.    caminos=[];
8.
9.    % Inicializar abiertos, caminos, coste:
10.   j=1;
11.   for i=1:N
12.         if i~=nodo_inicial
13.                    abiertos=[abiertos;i];
14.                caminos=[caminos;nodo_inicial,i];
15.            costes=[costes;matriz_costes(nodo_inicial,i)];
16.            if i==nodo_final
17.               i_final=j;
18.            end
19.            j=j+1;
20.          else
21.            caminos=[caminos;nodo_inicial,-1];
22.            costes(i)=0;
23.         end
24.   end
25.
26.      %Bucle:
27.
28.      indice=1;
29.      menor=nodo_inicial;
30.      caminos;
31.      costes;
32.
```

```
33.      while indice<=N && menor~=nodo_final
34.
35.      % Extraer el nodo de menor coste de abiertos
36.
37.      M=length(abiertos);
38.      coste_min=Inf;
39.
40.      for i=1:M
41.         if costes(abiertos(i))<coste_min;
42.             menor=abiertos(i);
43.            coste_min=costes(abiertos(i));
44.         end
45.      end
46.     menor;
47.     % Extraer "Menor" de "Abiertos":
48.
49.      temp_abiertos=[];
50.         for i=1:M
51.                 if abiertos(i)~=menor
52.                  temp_abiertos=[temp_abiertos,abiertos(i)];
53.            end
54.         end
55.         abiertos=temp_abiertos;
56.         cerrados=[cerrados;menor];
57.
58.
59.         % Comparar costes y poner al d a
60.
61.         caminos=[caminos -ones(N,1 )];
62.
63.         for i=1:M-1
64.            n=abiertos(i);
65.            if costes(n)>coste_min+matriz_costes(menor,n)
66.               costes(n)=coste_min+matriz_costes(menor,n);
67.               k=1;
68.               aux=[];
69.
70.               while caminos(menor,k)~=-1
71.                  aux(k)=caminos(menor,k);
72.                  k=k+1;
73.               end
74.               aux=[aux n];
75.               %a=length(aux);
76.               caminos(n,1:k)=aux;
77.               b=size(caminos,2);
78.               caminos(n,k+1:b)=-ones(1,size(k+1-b,2));
79.            end
80.         end
81.         indice=indice+1;
82.         caminos;
83.         costes;
84.     end
85.
86.
87.      % Eliminar los -1's:
88.
89.     Camino_aux=caminos( nodo_final,:);
90.
91.     i=1;
```

```
92.        while Camino_aux(i)~=-1
93.            Camino(i)=Camino_aux(i);
94.            i=i+1;
95.        end
96.
97.        Camino;
98.        Coste=costes(nodo_final);
```

- ## Dibuja_nodos

```
1.      function [a, b, c, d, e, f, g, h, i] = Dibuja_nodos
2.      a=text(50,15,'1');
3.          set(a, 'color','g');
4.      i=text(50,50,'9');
5.          set(i,'color','m');
6.      h=text(10,41,'8');
7.          set(h,'color',[0.5 0.5 0]);
8.      b=text(17,4,'2');
9.      c=text(14,4,'3');
10.     d=text(17,36,'4');
11.     e=text(13,36,'5');
12.     f=text(55,39,'6');
13.     g=text(55,42,'7');
14.     end
```

- ## m_tipos

```
1. matriz_tipos=  [0  1  Inf  Inf  Inf  Inf  Inf  Inf  Inf;

                1  0   2   Inf  Inf  Inf  Inf  Inf  Inf;

             Inf   2   0   Inf   3   Inf  Inf   3   Inf;

             Inf  Inf  Inf  0    2    1   Inf  Inf  Inf;

             Inf  Inf  3    0   Inf  Inf   5   Inf  Inf;

             Inf  Inf  Inf  1   Inf   0    2  Inf   Inf;

             Inf  Inf  Inf Inf  Inf   2    0   1    1;

             Inf  Inf   5   Inf  3       Inf   1   0    6;

             Inf  Inf  Inf  Inf Inf  Inf   1   6    0];
```

- ## m_costes

```
1.  matriz_costes=  [0  34.7851  Inf  Inf   Inf  Inf  Inf  Inf Inf;
```

```
          34.7851   0      4   Inf   Inf  Inf  Inf  Inf Inf;

          Inf    4       0   Inf   32   Inf  Inf  38.1182  Inf;

          Inf   Inf     Inf  0     4   38.0526 Inf  Inf  Inf;

          Inf   Inf     32   4      0   Inf     Inf  6.7082 Inf;

          Inf   Inf     Inf   38.0526  Inf  0  4  Inf  Inf;

         Inf   Inf     Inf  Inf  Inf  4   0  45   8.6023;

         Inf   Inf  38.1182 Inf  6.7082  Inf  45  0 40.7922;

          Inf    Inf  Inf    Inf   Inf  Inf  8.6023 40.7922  0];
```

9.6.5. Ficheros de dibujo y líneas

- **Dibuagv**

```
1.      % Traslada o gira un objeto según la matriz homogénea dada.
2.
3.      function dibuagv(agv,xold,xnew)
4.
5.      Mold=XaMH(xold);
6.      Mnew=XaMH(xnew);
7.      M = Mnew / Mold;
8.
9.      dibuobj(agv,M);
10.
11.     end
```

- **Dibuent**

```
1.      function dibuent(entorno,color)
2.
3.      N=size(entorno,1);
4.
5.          for i=1:N
6.             hold on
7.             line('xdata',[entorno(i,1);entorno(i,4)],'ydata',[entorno(i,2);entorno(i,5)],'color',color);
8.
9.          end
10.         hold off
11.     end
```

- **Dibulin**

```
1.      %modifica una línea según la matriz de transformación dada.
```

```
2.
3.      function dibulin(lin,matriz)
4.
5.      [P1, P2]=extrae(lin);
6.      P3=matriz*P1;
7.      P4=matriz*P2;
8.      modlin(lin,P3,P4);
9.
10.     end
```

- **Dibuobj**

```
1.      % Traslada o gira un objeto según la matriz homogénea dada.
2.
3.      function dibuobj(obj,matriz)
4.
5.      N=length(obj);
6.
7.      for i=1:N
8.
9.      dibulin(obj(i),matriz);
10.
11.     end
```

- **Extrae**

```
1.      % Extrae las coordenadas del punto inicial y punto final de una linea.
2.
3.      function [P1, P2]=extrae(linea)
4.
5.      X=get(linea,'xdata');
6.      Y=get(linea,'ydata');
7.      Z=get(linea,'zdata');
8.
9.      P1=[eye(3) [X(1);Y(1);Z(1)];0 0 0 1];
10.     P2=[eye(3) [X(2);Y(2);Z(2)];0 0 0 1];
11.
12.     end
```

9.6.6. Ficheros de matrices y coordenadas

- **mh_sensor**

```
1.      function matriz_sensor = mh_sensor(matriz_sensor_laser)
2.          % Extraer los parámetros de la pose del sensor
3.          x = matriz_sensor_laser(1);
4.          y = matriz_sensor_laser(2);
5.          theta = matriz_sensor_laser(3);
6.
7.          % Crear la matriz de traslación
8.          matriz_traslacion = despl(x, y, 0);
9.
10.         % Crear la matriz de rotación
11.         matriz_rotacion = rotaz(theta);
```

```
12.
13.         % Combinar las matrices de traslación y rotación
14.         matriz_sensor = matriz_traslacion * matriz_rotacion;
15.     end
```

- **MHaX**

```
1.   % Crea la matriz homogénea a partir de la posición (estado) y orientación del robot.

2.   function X=MHaX(M)

3.   X(1)=M(1,4);
4.   X(2)=M(2,4);
5.   X(3)=atan2(M(2,1),M(1,1));

6.   end
```

- **XaMH**

```
1.      % Crea la matriz homogénea a partir de la posición (estado) y orientación del robot.
2.       function matriz=XaMH(X)
3.      matriz= despl(X(1),X(2),0)*rotaz(X(3));
4.      end
```

9.6.7. Entornos

- **Entorno con una sola pared**

```
1.      function ent_pared = entorno_pared()
2.
3.      ent_pared=[
4.                  5, -5, 0, 5, 10, 0, 0, 0
5.              ];
6.      end
```

- **Entorno objetivo**

```
1.      % Define el entorno del objetivo
2.      function ent0 = entorno_objetivo()
3.
4.      %                          xin     yin     zin     xfin          yfin         zfin  ángulo objeto
5.
6.      ent0=[       0                        0                        0                        0
                     0                        0                        0       0];
7.      end
```

- **Entorno evitación de obstáculos**

```
1.   function ent_evita_obstaculos = entorno_evita_obstaculos()
2.   ent_evita_obstaculos = [ -15  10  0  -5   10  0  pi/2;
3.   -5   10  0  -5   15  0  pi;
4.   -5   15  0  -15  15  0  pi/2;
5.   -15  15  0  -15  10  0  0;
6.   5    15  0  15   15  0  pi/2;
7.   15   15  0  15   20  0  0;
8.   15   20  0  10   20  0  3*pi/2;
9.   5    20  0  5    15  0  0;
10.  10   20  0  10   25  0  pi;
11.  10   25  0  5    25  0  3*pi/2;
12.  5    25  0  5    20  0  0;
13.  0    30  0  10   30  0  pi/2;
14.  10   30  0  10   35  0  pi;
15.  10   35  0  0    35  0  3*pi/2;
16.  0    35  0  0    30  0  0 ];

17.  end
```

- **Entorno seguir pared**

```
1.   function ent_pared = entorno_seguir_pared()
2.   ent_pared=[
3.   0, 0, 0, 0, 40, 0, 0;
4.   20, 0, 0, 20, 40, 0, 0;
5.   ];
6.   end
```

- **Entorno seguir un pasillo**

```
1.   function ent_pasillo = entorno_pasillo()
2.
3.   ent_pasillo=[
4.           -5    0    0    -5    30   0    0;
5.           -5    30   0    20    30   0    3*pi/2;
6.           -15   0    0    -15   40   0    pi;
7.           -15   40   0    20    40   0    pi/2;
8.           20    30   0    20    40   0    pi/2];
9.   end
```

- **Entorno atravesar puertas**

```
1.   % Define el entorno en forma de líneas.
2.   % Define un recinto para atravesar puerta.
3.   function ent = entorno_puerta()
4.
5.   ent=[
6.           -15  25  0  -1.5  25  0  pi/2;
7.           7    -1.5 25  0  -1.5  26  0  pi;
8.           8    -1.5 26  0  -16  26  0  3*pi/2;
9.           9    -16  0  0  -16  26  0  0;
10.          10   5   25  0   1.5  25  0  pi/2;
```

```
11.           11   1.5 25  0   1.5  26  0  0;
12.           12   1.5 26  0   6    26  0  3*pi/2;
13.           13   6    0  0   6    26  0  0;
14.           14   -15  0  0   -15  25  0  0;
15.           15   -15  0  0   5    0   0  0;
16.           16   5    0  0   5    25  0  0;
17.       ];
18.   end
```

- ## Entorno aparcar

```
1.    function ent = entorno_aparcar()
2.
3.    ent=[
4.        -15     0       0       -15     15      0; %pared izquierda
5.        -15     15      0       0       15      0; %pared frontal
6.         0     0       0        0       15      0  %pared derecha
7.        ];
8.
9.    end
```

- ## Entrono final

```
1.    function ent_final=entorno_final()
2.    ent_final=[   0    0    0   15   0    0   -pi/2;
3.            15   0    0   15   3    0    0;
4.            15   3    0   16   3    0   -pi/2;
5.            16   3    0   16   0    0    pi;
6.            16   0    0   60   0    0   -pi/2;
7.            60   0    0   60   19.5 0    0;
8.            60   19.5 0   16   19.5 0    pi/2;
9.            16   19.5 0   16   5    0    pi;
10.           16   5    0   15   5    0    pi/2;
11.           15   5    0   15   35   0    0;
12.           15   35   0   16   35   0    3*pi/2;
13.           16   35   0   16   20.5 0    pi;
14.           16   20.5 0   60   20.5 0   -pi/2;
15.           60   20.5 0   60   40   0    0;
16.
17.           60   40   0   56   40   0    pi/2;
18.           56   40   0   56   41   0    pi;
19.           56   41   0   60   41   0   -pi/2;
20.
21.           60   41   0   60   60   0    0;
22.           60   60   0   0    60   0    pi/2;
23.           0    60   0   0    0    0    pi;
24.
25.           54   40   0   16   40   0    pi/2;
26.           16   40   0   16   37   0    pi;
27.           16   37   0   15   37   0    pi/2;
28.           15   37   0   15   41   0    0;
29.           15   41   0   54   41   0   -pi/2;
30.           54   41   0   54   40   0    0];
31.   end
```

2. Planos

UNIVERSITAT POLITÈCNICA DE VALÈNCIA

Escuela Técnica Superior de Ingeniería Aeroespacial y Diseño Industrial

Planos

Implementación de una arquitectura funcional para robots móviles en Matlab

Grado en Ingeniería Electrónica Industrial y Automática

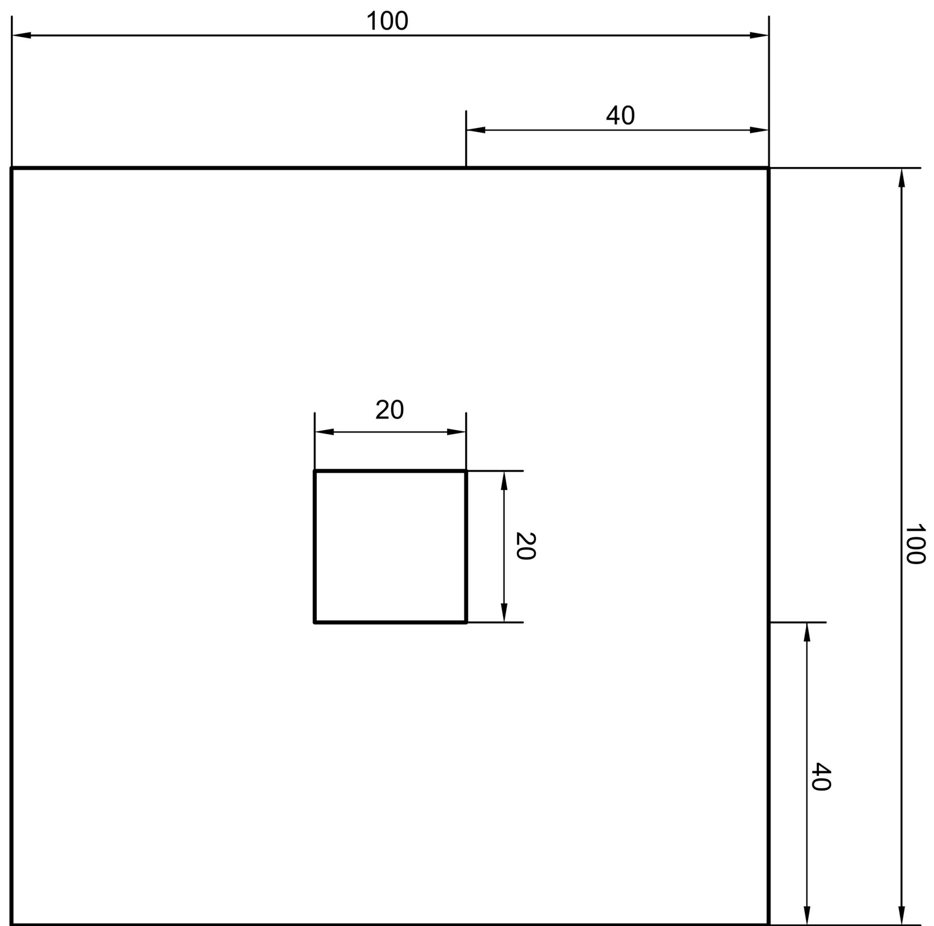
AUTORA: Peñarrubia Zamora, Laura

TUTOR: Zotovic Stanisic, Ranko

CURSO ACADÉMICO: 2023/2024

ÍNDICE DE PLANOS

1. Robot y sensor láser
2. Diagrama de flujo de la función intersección de la clase sensor láser
3. Diagrama de flujo de la función calcular intersección múltiple de la clase sensor láser
4. Diagrama de flujo de la función mover robot de la clase robot
5. Diagrama de flujo de la función simulación giro robot de la clase robot
6. Diagrama de flujo de la función girar robot de la clase robot
7. Diagrama de flujo de la función objetivo de la clase robot
8. Diagrama de flujo de la función vector objetivo de la clase robot
9. Diagrama de flujo de la función evita obstáculos de la clase robot
10. Diagrama de flujo de la función orientación pared de la clase robot
11. Diagrama de flujo de la función seguir pared de la clase robot
12. Diagrama de flujo de la función orientación pasillo de la clase robot
13. Diagrama de flujo de la función seguir pasillo de la clase robot
14. Diagrama de flujo de la función infinito de la clase robot
15. Diagrama de flujo de la función atravesar puertas de la clase robot
16. Diagrama de flujo de la función vector aparcamiento de la clase robot
17. Diagrama de flujo de la función aparcar de la clase robot
18. Diagrama de flujo de la función Dijkstra
19. Diagrama de flujo de la función piloto de la clase robot
20. Diagrama de flujo de la función simulación objetivo de la clase robot
21. Diagrama de flujo de la función simulación evita obstáculos de la clase robot
22. Diagrama de flujo de la función simulación seguir pared de la clase robot
23. Diagrama de flujo de la función simulación seguir pasillo de la clase robot
24. Diagrama de flujo de la función simulación atravesar puertas de la clase robot
25. Diagrama de flujo de la función simulación aparcar de la clase robot
26. Diagrama de flujo de la función simulación piloto de la clase robot



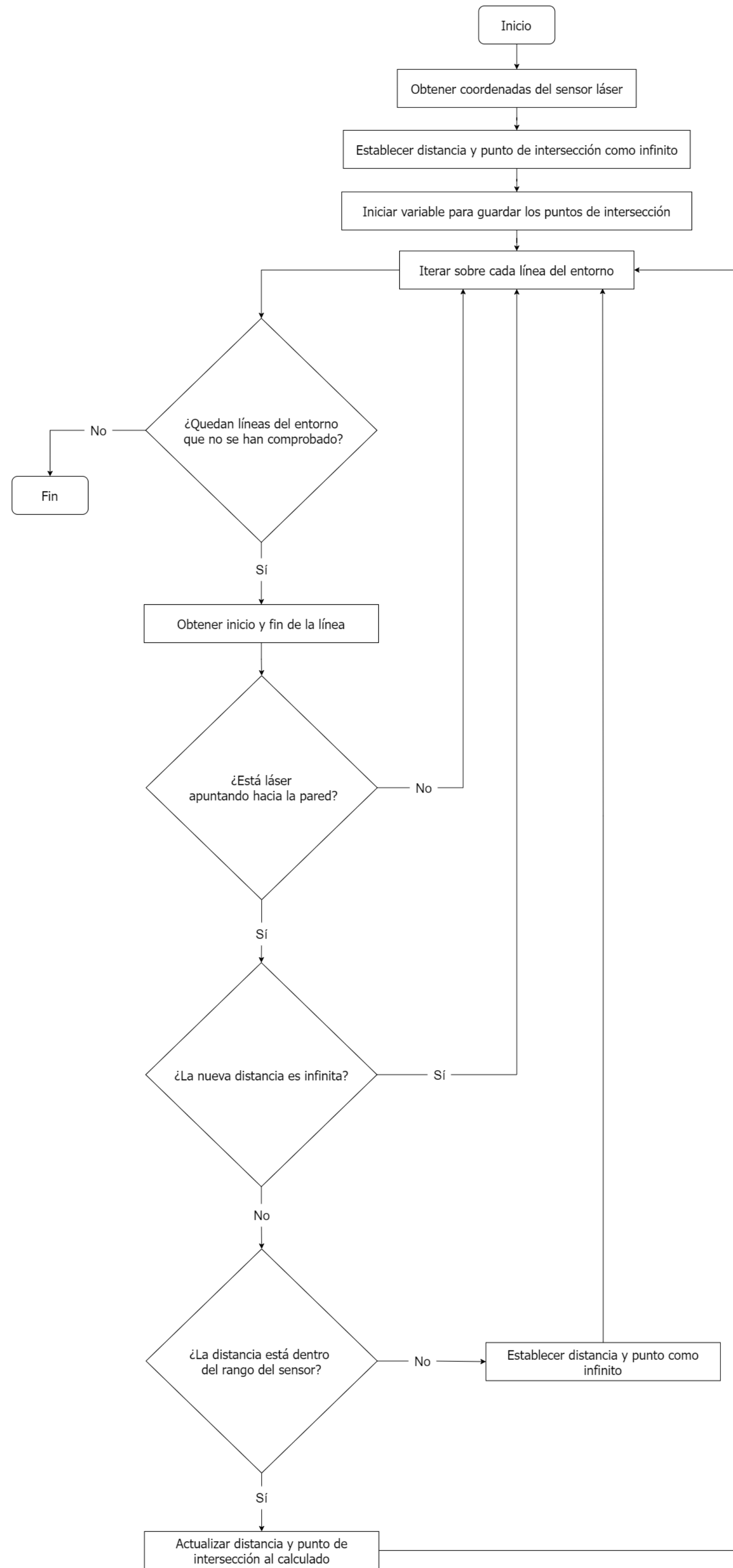
	Fecha	Apellidos, Nombre	Firmas
Diseñado	Jun 2024	Peñarrubia Zamora, Laura	
Dibujado	Jun 2024	Peñarrubia Zamora, Laura	
Comprobado	Jun 2024	Peñarrubia Zamora, Laura	

ESCUELA TÉCNICA SUPERIOR DE INGENIERÍA AEROESCPACIAL Y DISEÑO INDUSTRIAL

Título del proyecto: Implementación de una arquitectura funcional para robots móviles en MATLAB.

Autor: Peñarrubia Zamora, Laura

Escala	Nombre del plano	Número del plano
1:10	Robot y sensor láser	1/26



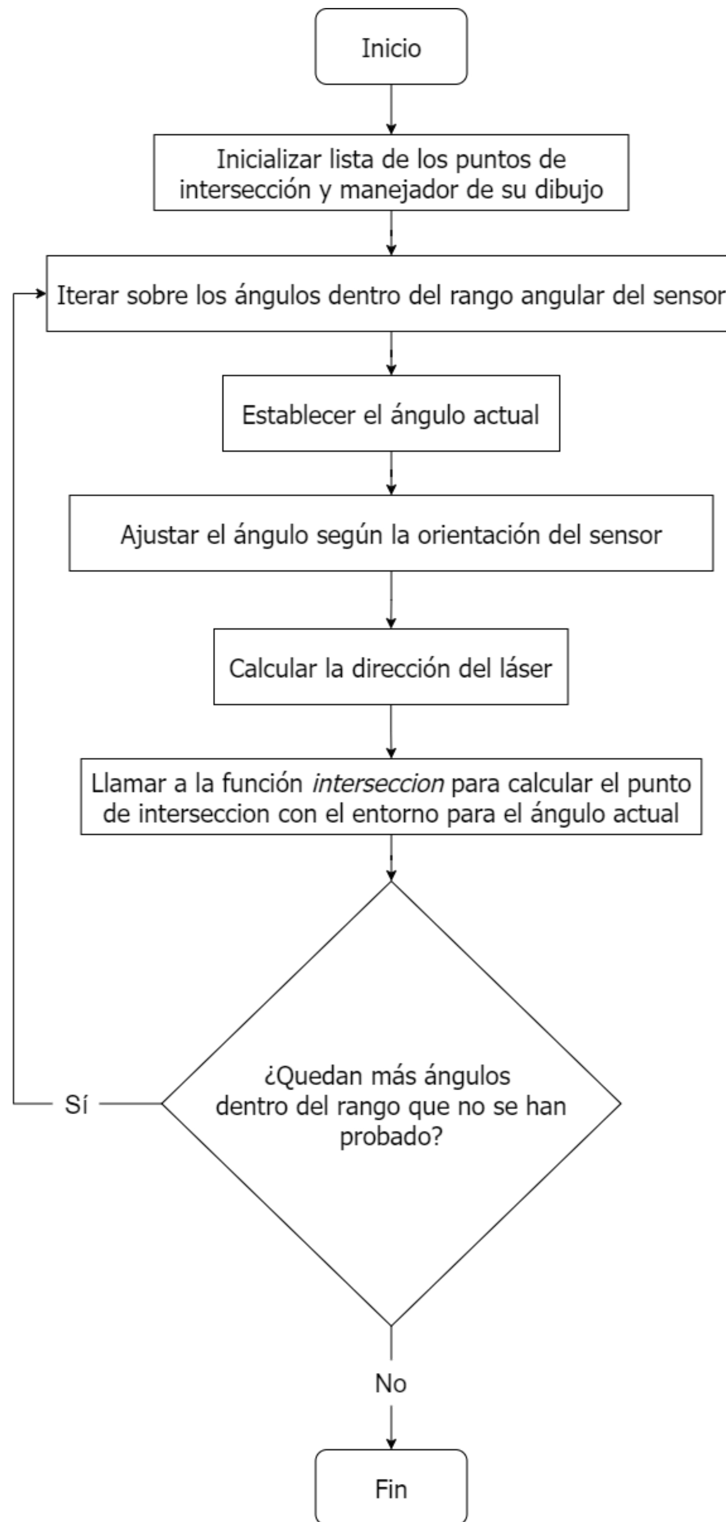
	Fecha	Apellidos, Nombre	Firmas
Diseñado	Jun 2024	Peñarrubia Zamora, Laura	
Dibujado	Jun 2024	Peñarrubia Zamora, Laura	
Comprobado	Jun 2024	Peñarrubia Zamora, Laura	

ESCUELA TÉCNICA SUPERIOR DE INGENIERÍA AEROESCPACIAL Y DISEÑO INDUSTRIAL

Título del proyecto: Implementación de una arquitectura funcional para robots móviles en MATLAB.

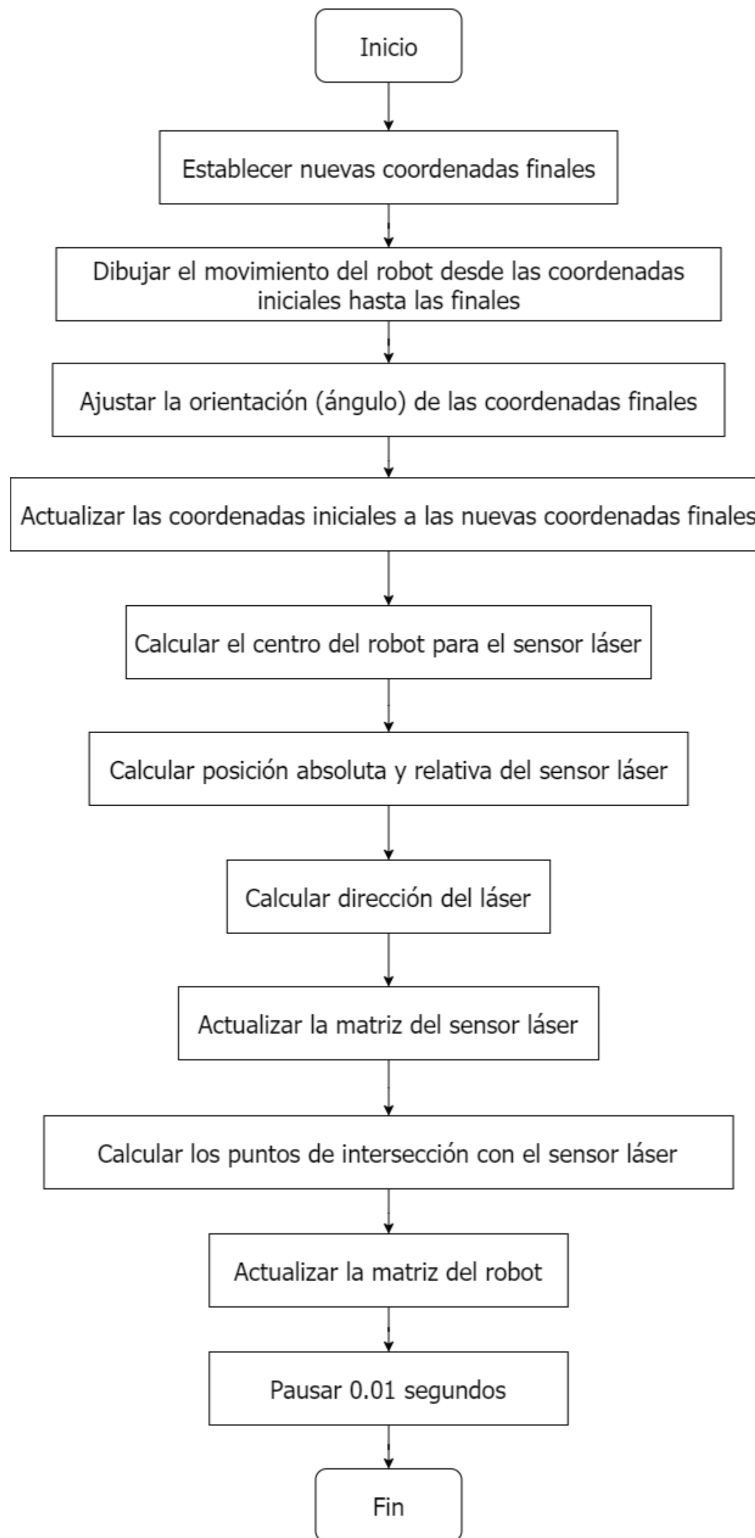
Autor: Peñarrubia Zamora, Laura

Escala	Nombre del plano	Número del plano
	Diagrama de flujo de la función intersección de la clase sensor láser	2/26



	Fecha	Apellidos, Nombre	Firmas	ESCUELA TÉCNICA SUPERIOR DE INGENIERÍA AEROESCPACIAL Y DISEÑO INDUSTRIAL
Diseñado	Jun 2024	Peñarrubia Zamora, Laura		Título del proyecto: Implementación de una arquitectura funcional para robots móviles en MATLAB.
Dibujado	Jun 2024	Peñarrubia Zamora, Laura		
Comprobado	Jun 2024	Peñarrubia Zamora, Laura		Autor: Peñarrubia Zamora, Laura

Escala	Nombre del plano	Número del plano
	Diagrama de flujo de la función calcular intersección múltiple de la clase sensor láser	3/26



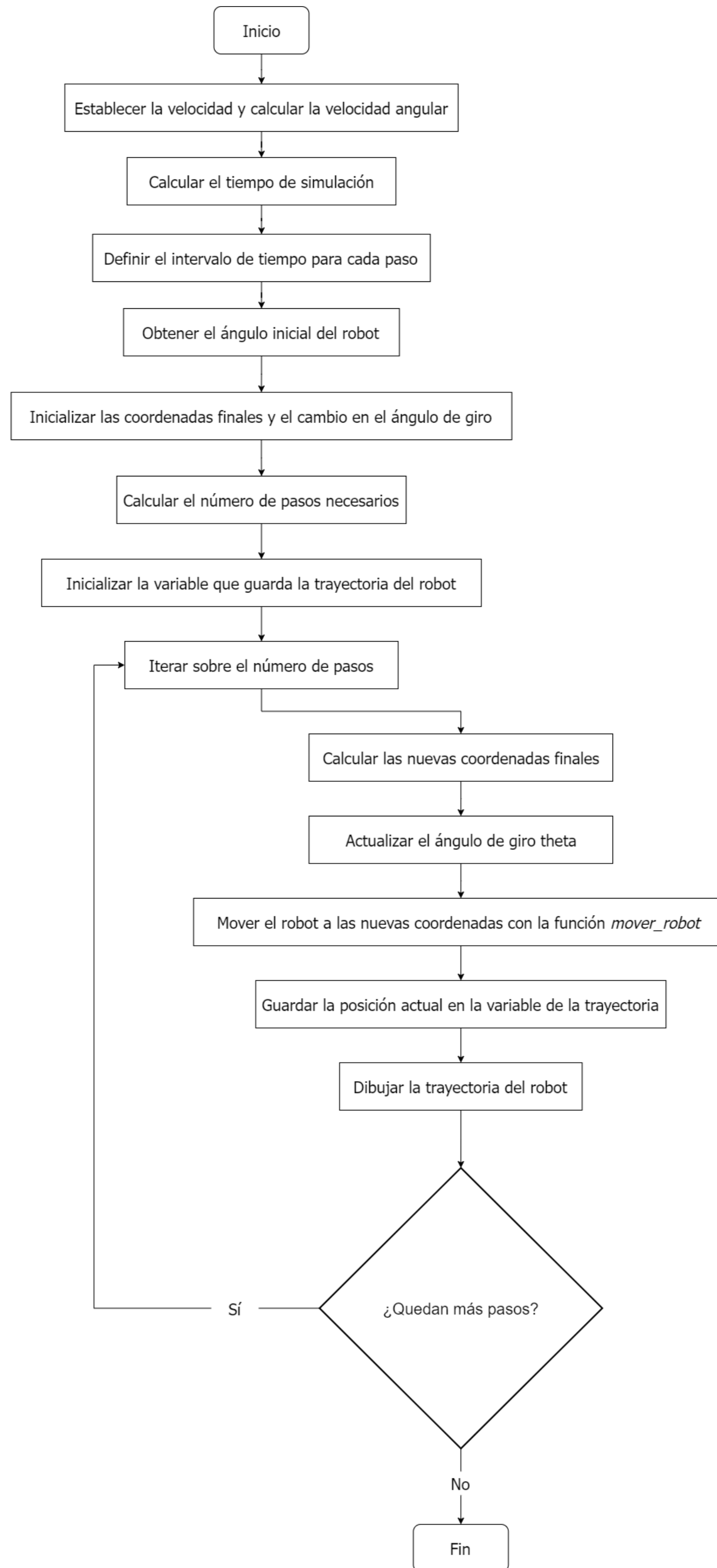
	Fecha	Apellidos, Nombre	Firmas
Diseñado	Jun 2024	Peñarrubia Zamora, Laura	
Dibujado	Jun 2024	Peñarrubia Zamora, Laura	
Comprobado	Jun 2024	Peñarrubia Zamora, Laura	

ESCUELA TÉCNICA SUPERIOR DE INGENIERÍA AEROESCPACIAL Y DISEÑO INDUSTRIAL

Título del proyecto: Implementación de una arquitectura funcional para robots móviles en MATLAB.

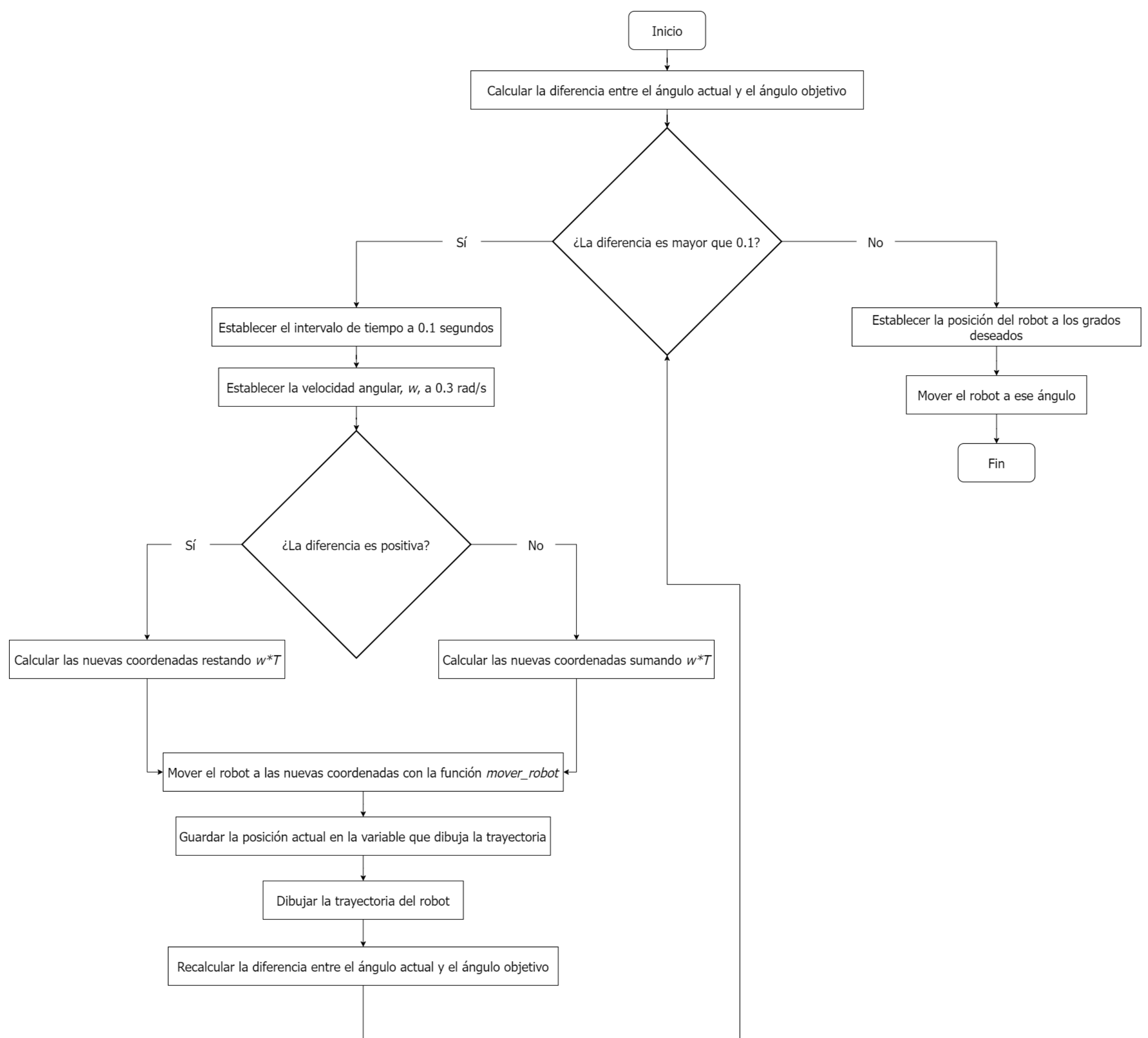
Autor: Peñarrubia Zamora, Laura

Escala	Nombre del plano	Número del plano
	Diagrama de flujo de la función mover robot de la clase robot	4/26



	Fecha	Apellidos, Nombre	Firmas	ESCUELA TÉCNICA SUPERIOR DE INGENIERÍA AEROESCPACIAL Y DISEÑO INDUSTRIAL
Diseñado	Jun 2024	Peñarrubia Zamora, Laura	[firma]	Título del proyecto: Implementación de una arquitectura funcional para robots móviles en MATLAB.
Dibujado	Jun 2024	Peñarrubia Zamora, Laura	[firma]	
Comprobado	Jun 2024	Peñarrubia Zamora, Laura	[firma]	Autor: Peñarrubia Zamora, Laura

Escala	Nombre del plano	Número del plano
	Diagrama de flujo de la función simulación giro robot de la clase robot	5/26



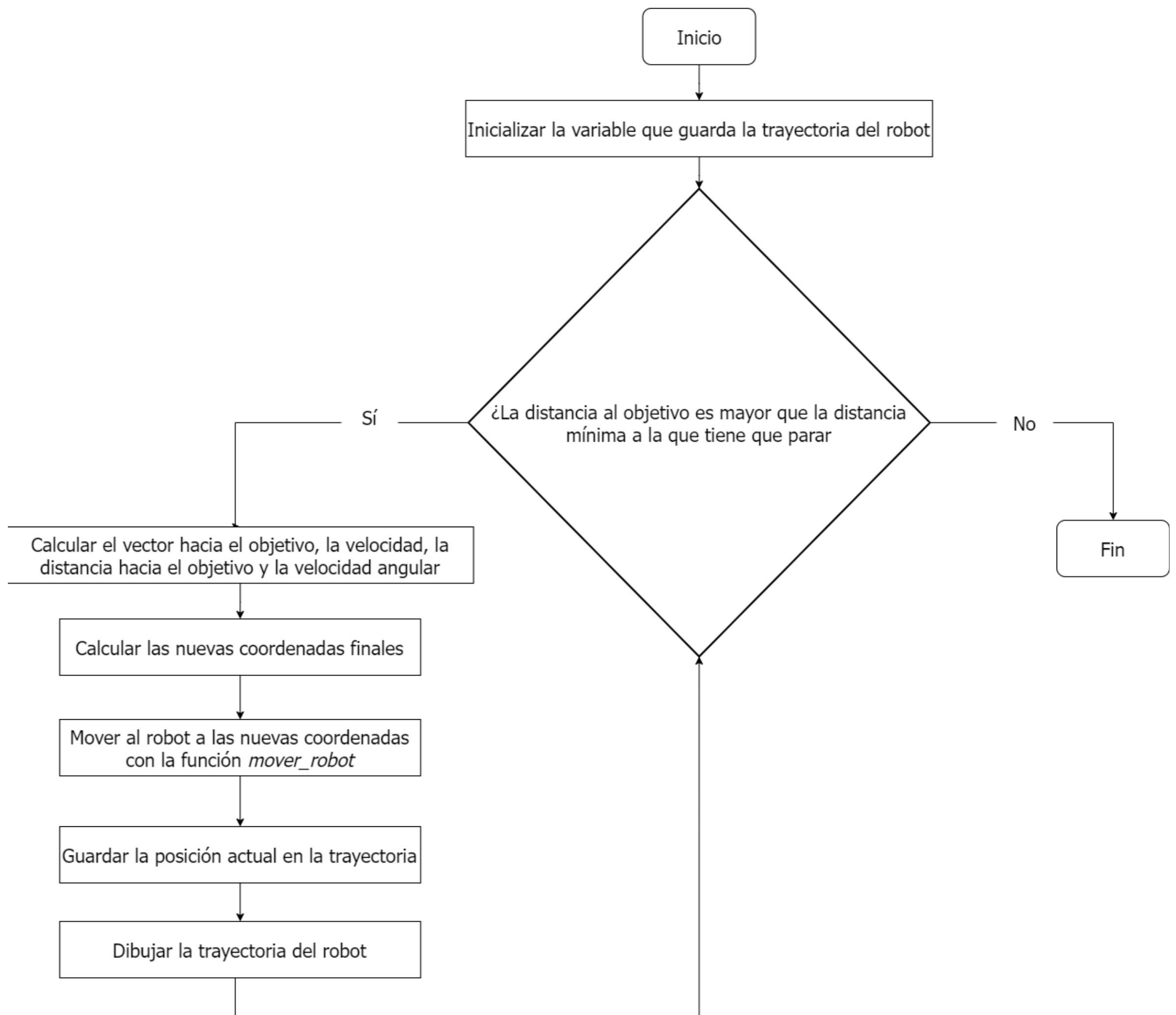
	Fecha	Apellidos, Nombre	Firmas
Diseñado	Jun 2024	Peñarrubia Zamora, Laura	
Dibujado	Jun 2024	Peñarrubia Zamora, Laura	
Comprobado	Jun 2024	Peñarrubia Zamora, Laura	

ESCUELA TÉCNICA SUPERIOR DE INGENIERÍA AEROESCPACIAL Y DISEÑO INDUSTRIAL

Título del proyecto: Implementación de una arquitectura funcional para robots móviles en MATLAB.

Autor: Peñarrubia Zamora, Laura

Escala	Nombre del plano	Número del plano
	Diagrama de flujo de la función girar robot de la clase robot	6/26



	Fecha	Apellidos, Nombre	Firmas
Diseñado	Jun 2024	Peñarrubia Zamora, Laura	
Dibujado	Jun 2024	Peñarrubia Zamora, Laura	
Comprobado	Jun 2024	Peñarrubia Zamora, Laura	

ESCUELA TÉCNICA SUPERIOR DE INGENIERÍA AEROESCPACIAL Y DISEÑO INDUSTRIAL

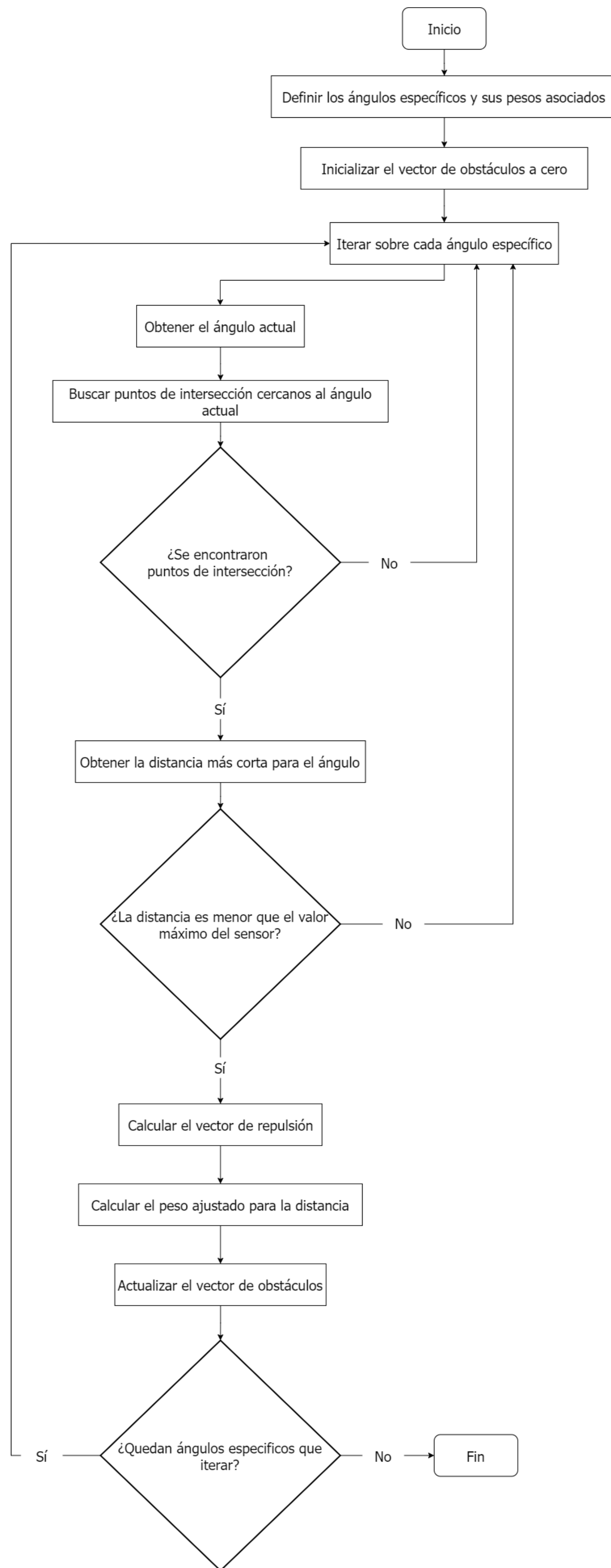
Título del proyecto: Implementación de una arquitectura funcional para robots móviles en MATLAB.

Autor: Peñarrubia Zamora, Laura

Escala	Nombre del plano	Número del plano
	Diagrama de flujo de la función objetivo de la clase robot	7/26

	Fecha	Apellidos, Nombre	Firmas	ESCUELA TÉCNICA SUPERIOR DE INGENIERÍA AEROESCPACIAL Y DISEÑO INDUSTRIAL
Diseñado	Jun 2024	Peñarrubia Zamora, Laura		Título del proyecto: Implementación de una arquitectura funcional para robots móviles en MATLAB.
Dibujado	Jun 2024	Peñarrubia Zamora, Laura		
Comprobado	Jun 2024	Peñarrubia Zamora, Laura		Autor: Peñarrubia Zamora, Laura

Escala	Nombre del plano	Número del plano
	Diagrama de flujo de la función vector objetivo de la clase robot	8/26



	Fecha	Apellidos, Nombre	Firmas
Diseñado	Jun 2024	Peñarrubia Zamora, Laura	
Dibujado	Jun 2024	Peñarrubia Zamora, Laura	
Comprobado	Jun 2024	Peñarrubia Zamora, Laura	

ESCUELA TÉCNICA SUPERIOR DE INGENIERÍA AEROESCPACIAL Y DISEÑO INDUSTRIAL

Título del proyecto: Implementación de una arquitectura funcional para robots móviles en MATLAB.

Autor: Peñarrubia Zamora, Laura

Escala	Nombre del plano	Número del plano
	Diagrama de flujo de la función evita obstáculos de la clase robot	9/26

Inicio

Buscar puntos de intersección para el ángulo y el ángulo perpendicular

Obtener las distancias del ángulo perpendicular(distancia b) y el ángulo lateral (distancia a)

¿La *distancia a* es infinita?

Sí

Asignar *distancia a* a cero

No

¿La *distancia b* es infinita?

Sí

Asignar *distancia a* a cero

No

Calcular la diferencia entre *distancia a* y *distancia b*

Calcular la diferencia entre *distancia b* y la distancia de referencia

¿La diferencia entre la *distancia b* y la de referencia es menor que 0.1?

Sí

No

Asignar el vector resultante a 0, sigue recto

Asignar el vector resultante basado en la diferencia de distancia entre ángulos

¿El robot está orientado verticalmente?

No

Sí

¿Los sensores son los de la izquierda?

Sí

No

¿La coordenada X medida por el ángulo perpendicular es negativa?

Sí

No

Calcular la coordenada X hacia donde se tiene que mover el robot sumando la distancia de referencia a la medida del ángulo

Calcular la coordenada X hacia donde se tiene que mover el robot restando la distancia de referencia a la medida del ángulo

Mover el robot a la nueva posición

Girar el robot 90º con la funcion *girar_robot*

Fin

¿La *distancia b* es menor a la de referencia?

No

Sí

¿La *distancia b* es menor a la de referencia?

No

Sí

No

¿La *distancia a* es cero?

Sí

Sí

¿La *distancia a* es cero?

No

Sí

¿La *distancia a* es cero?

No

¿La *distancia a* es cero?

No

Sí

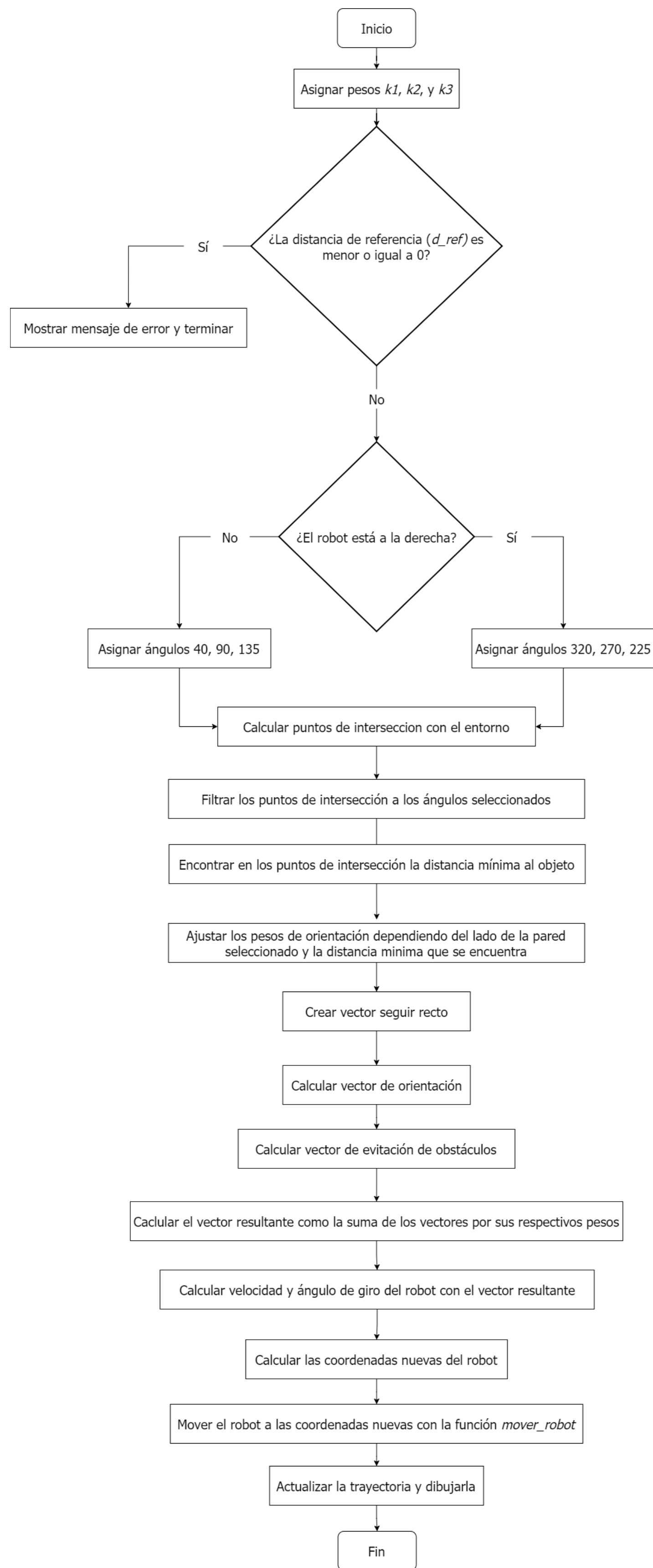
Asignar vector a [0, diferencia entre distancias]

Asignar vector a [0, -diferencia entre distancias]

Fin

	Fecha	Apellidos, Nombre	Firmas	ESCUELA TÉCNICA SUPERIOR DE INGENIERÍA AEROESCPACIAL Y DISEÑO INDUSTRIAL
Diseñado	Jun 2024	Peñarrubia Zamora, Laura	[firma]	Título del proyecto: Implementación de una arquitectura funcional para robots móviles en MATLAB.
Dibujado	Jun 2024	Peñarrubia Zamora, Laura	[firma]	
Comprobado	Jun 2024	Peñarrubia Zamora, Laura	[firma]	Autor: Peñarrubia Zamora, Laura

Escala	Nombre del plano	Número del plano
	Diagrama de flujo de la función orientación pared de la clase robot	10/26



	Fecha	Apellidos, Nombre	Firmas
Diseñado	Jun 2024	Peñarrubia Zamora, Laura	
Dibujado	Jun 2024	Peñarrubia Zamora, Laura	
Comprobado	Jun 2024	Peñarrubia Zamora, Laura	

ESCUELA TÉCNICA SUPERIOR DE INGENIERÍA AEROESCPACIAL Y DISEÑO INDUSTRIAL

Título del proyecto: Implementación de una arquitectura funcional para robots móviles en MATLAB.

Autor: Peñarrubia Zamora, Laura

Escala	Nombre del plano	Número del plano
	Diagrama de flujo de la función seguir pared de la clase robot	11/26

Inicio

Obtener *punto a*, *punto b*, *punto d* y *punto e*

Calcular la diferencia entre las distancias del *punto_b* y el *punto_e* (ángulos perpendiculares)

¿La diferencia es menor a 0.08?

Sí

¿El robot está totalmente horizontal o vertical?

Sí

Seguir recto

Fin

No

Girar el robot 90 grados

Calcular los puntos de intersección con el entorno y filtrar puntos

Obtener *punto_b* y *punto_e* (ángulos perpendiculares) de los puntos de intersección

¿Las coordenadas Y de los ángulos perpendiculares son iguales?

Sí

¿Las coordenadas X son negativas?

Sí

¿La coordenada X del *punto_b* es mayor a la coordenada X *punto_e*?

Sí

Calcular la coordenada X como -(*punto_b* - *punto_e*)

Fin

No

¿La coordenada X del *punto_b* es menor a la coordenada X *punto_e*?

Sí

Calcular la coordenada X como -(*punto_e* - *punto_b*)

Fin

No

Asignar la coordenada X a la coordenada X del *punto_e*

Fin

No

¿La coordenada X del *punto_b* es mayor a la coordenada X *punto_e*?

Sí

¿La coordenada X del *punto_e* es cero?

Sí

Calcular la coordenada X como *punto_b*/2

Fin

No

Calcular la coordenada X como (*punto_b* - *punto_e*)

Fin

No

¿La coordenada X del *punto_b* es menor a la coordenada X *punto_e*?

No

Asignar la coordenada X a la coordenada X del *punto_e*

Fin

Sí

¿La coordenada X del *punto_b* es cero?

Sí

Calcular la coordenada X como *punto_e*/2

Fin

No

Calcular la coordenada X como (*punto_e* - *punto_b*)

Fin

No

¿Las coordenadas X son iguales?

No

¿Las coordenadas X son negativas?

Sí

¿La coordenada X del *punto_b* es mayor a la coordenada X *punto_e*?

Sí

Calcular la coordenada X como -(*punto_b* - *punto_e*)

Fin

No

¿La coordenada X del *punto_b* es menor a la coordenada X *punto_e*?

Sí

Calcular la coordenada X como -(*punto_e* - *punto_b*)

Fin

No

Asignar la coordenada X como a la coordenada X del *punto_e*

Fin

No

¿La coordenada X del *punto_b* es mayor a la coordenada X *punto_e*?

Sí

Calcular la coordenada X como (*punto_b* - *punto_e*)

Fin

No

¿La coordenada X del *punto_b* es menor a la coordenada X *punto_e*?

Sí

Calcular la coordenada X como (*punto_e* - *punto_b*)

Fin

No

Asignar la coordenada X como a la coordenada X del *punto_e*

Fin

No

Asignar coordenada X a cualquiera de las dos coordenadas X de los puntos

Mover el robot a la nueva posición con *mover_robot*

Asignar constante de velocidad a 30 para que vaya más rápido

Asignar vector orientación a [0,0] (ir recto)

Fin

No

¿La distancia del *punto_b* es menor a la distancia del *punto_e*?

Sí

Se comprueba si las distancias del *punto_e* y el *punto_d* son infinitas y se les asigna un 0 en caso de que lo sean

Calcular la diferencia como (*punto_d* - *punto_e*)

¿La distancia del *punto_d* es 0?

Sí

Calcular el vector como [0, -diferencia]

No

Calcular el vector como [0, diferencia]

No

¿La distancia del *punto_b* es mayor a la distancia del *punto_e*?

No

Fin

Sí

Se comprueba si las distancias del *punto_a* y el *punto_b* son infinitas y se les asigna un 0 en caso de que lo sean

Calcular la diferencia como (*punto_a* - *punto_b*)

¿La distancia del *punto_a* es 0?

Sí

Calcular el vector como [0, diferencia]

No

Calcular el vector como [0, -diferencia]

	Fecha	Apellidos, Nombre	Firmas
Diseñado	Jun 2024	Peñarrubia Zamora, Laura	
Dibujado	Jun 2024	Peñarrubia Zamora, Laura	
Comprobado	Jun 2024	Peñarrubia Zamora, Laura	

ESCUELA TÉCNICA SUPERIOR DE INGENIERÍA AEROESCPACIAL Y DISEÑO INDUSTRIAL

Título del proyecto: Implementación de una arquitectura funcional para robots móviles en MATLAB.

Autor: Peñarrubia Zamora, Laura

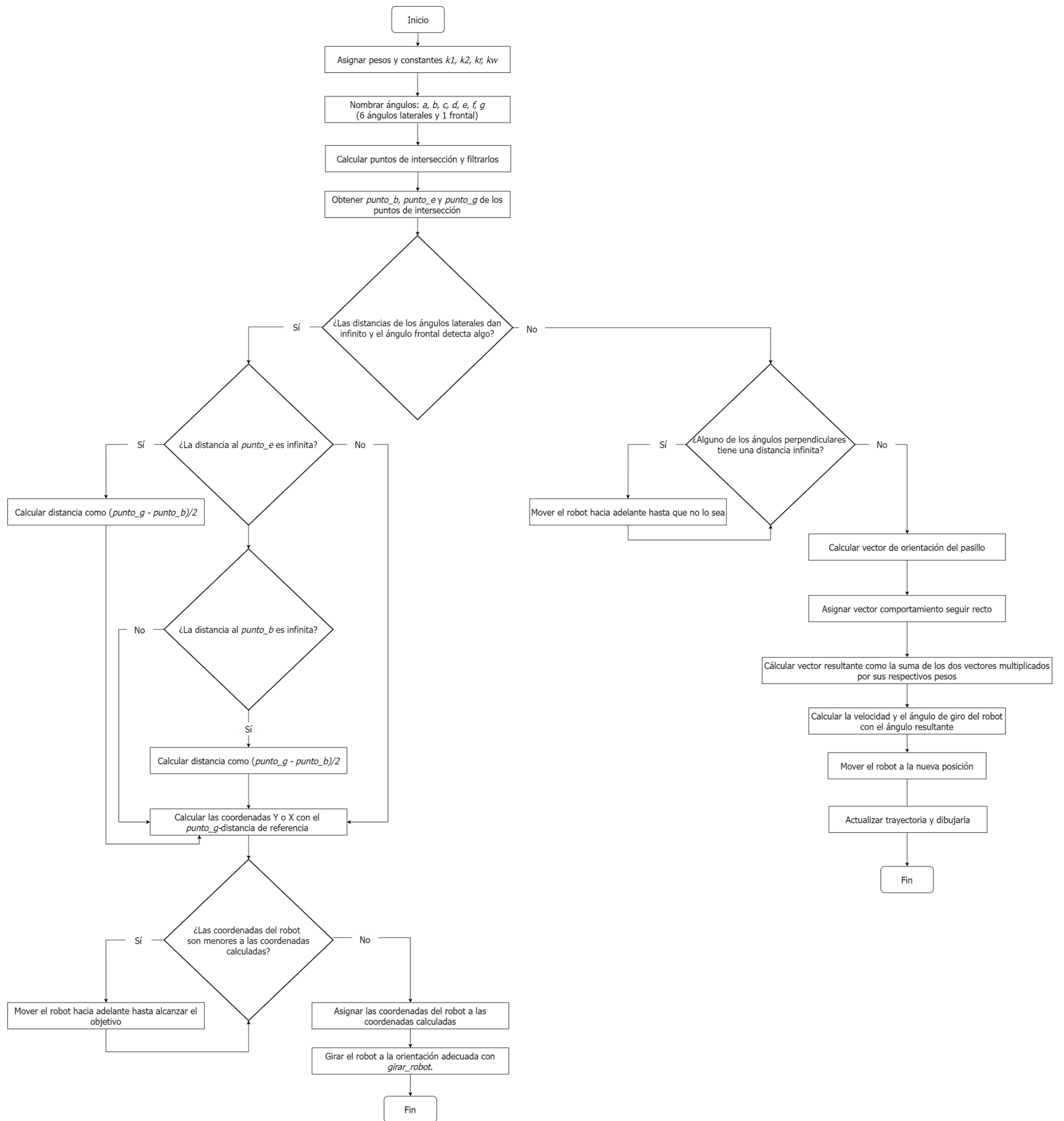
Escala

Nombre del plano

Diagrama de flujo de la función orientación pasillo de la clase robot

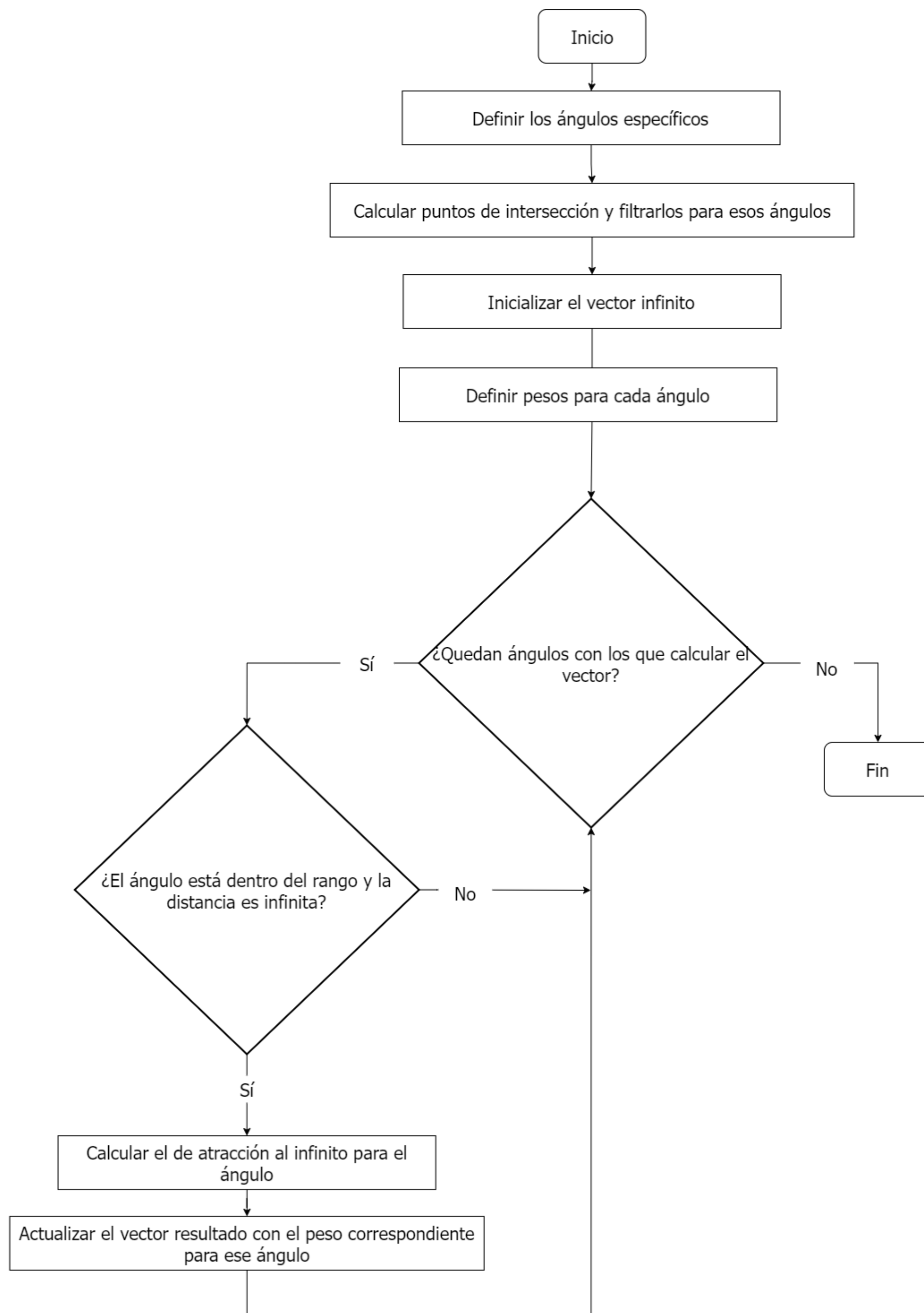
Número del plano

12/26



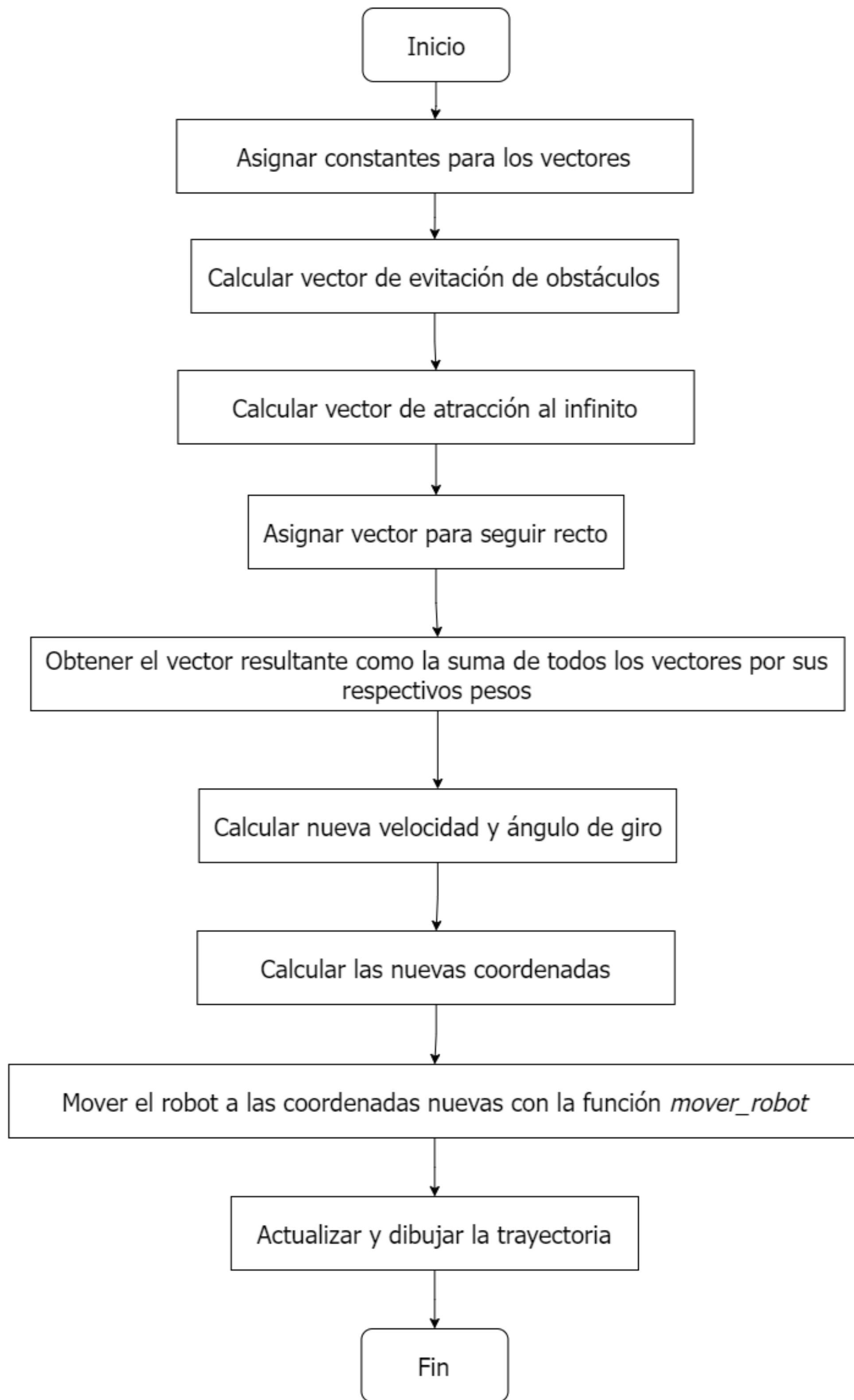
	Fecha	Apellidos, Nombre	Firmas	ESCUELA TÉCNICA SUPERIOR DE INGENIERÍA AEROESCPACIAL Y DISEÑO INDUSTRIAL
Diseñado	Jun 2024	Peñarrubia Zamora, Laura		Título del proyecto: Implementación de una arquitectura funcional para robots móviles en MATLAB.
Dibujado	Jun 2024	Peñarrubia Zamora, Laura		
Comprobado	Jun 2024	Peñarrubia Zamora, Laura		Autor: Peñarrubia Zamora, Laura

Escala	Nombre del plano	Número del plano
	Diagrama de flujo de la función seguir pasillo de la clase robot	13/26



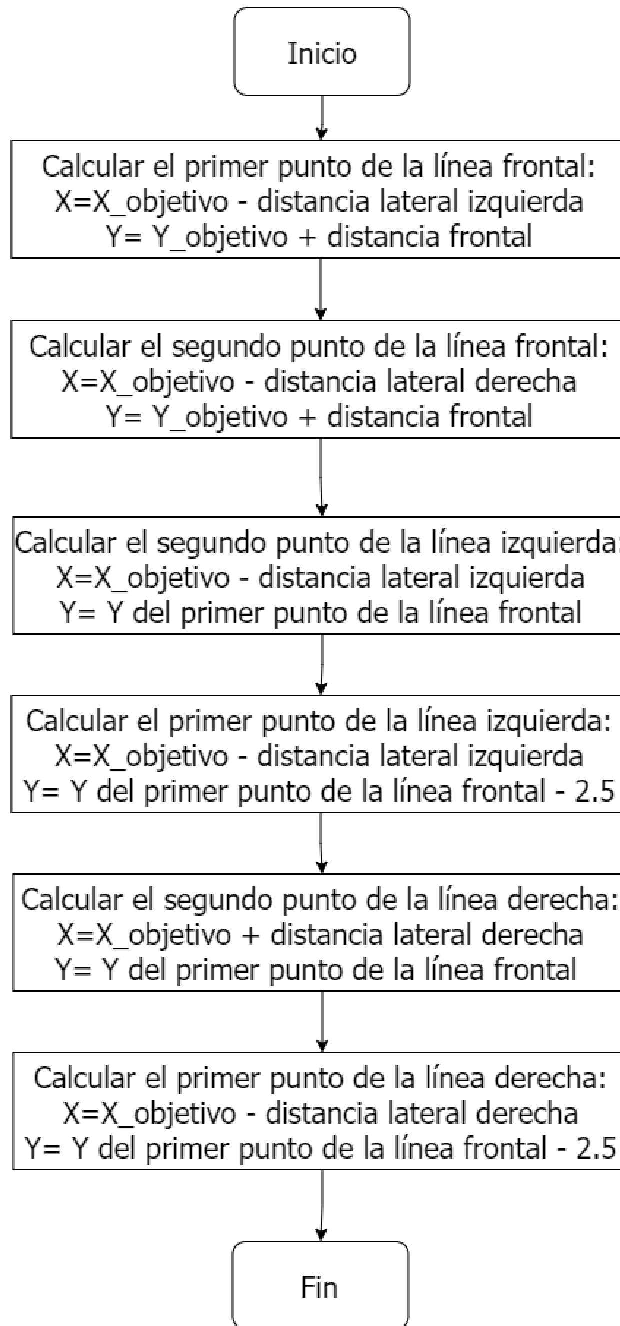
	Fecha	Apellidos, Nombre	Firmas	ESCUELA TÉCNICA SUPERIOR DE INGENIERÍA AEROESCPACIAL Y DISEÑO INDUSTRIAL
Diseñado	Jun 2024	Peñarrubia Zamora, Laura	[firma]	Título del proyecto: Implementación de una arquitectura funcional para robots móviles en MATLAB.
Dibujado	Jun 2024	Peñarrubia Zamora, Laura	[firma]	
Comprobado	Jun 2024	Peñarrubia Zamora, Laura	[firma]	Autor: Peñarrubia Zamora, Laura

Escala	Nombre del plano	Número del plano
	Diagrama de flujo de la función infinito de la clase robot	14/26



	Fecha	Apellidos, Nombre	Firmas	ESCUELA TÉCNICA SUPERIOR DE INGENIERÍA AEROESCPACIAL Y DISEÑO INDUSTRIAL
Diseñado	Jun 2024	Peñarrubia Zamora, Laura		Título del proyecto: Implementación de una arquitectura funcional para robots móviles en MATLAB.
Dibujado	Jun 2024	Peñarrubia Zamora, Laura		
Comprobado	Jun 2024	Peñarrubia Zamora, Laura		Autor: Peñarrubia Zamora, Laura

Escala	Nombre del plano	Número del plano
	Diagrama de flujo de la función atravesar puertas de la clase robot	15/26



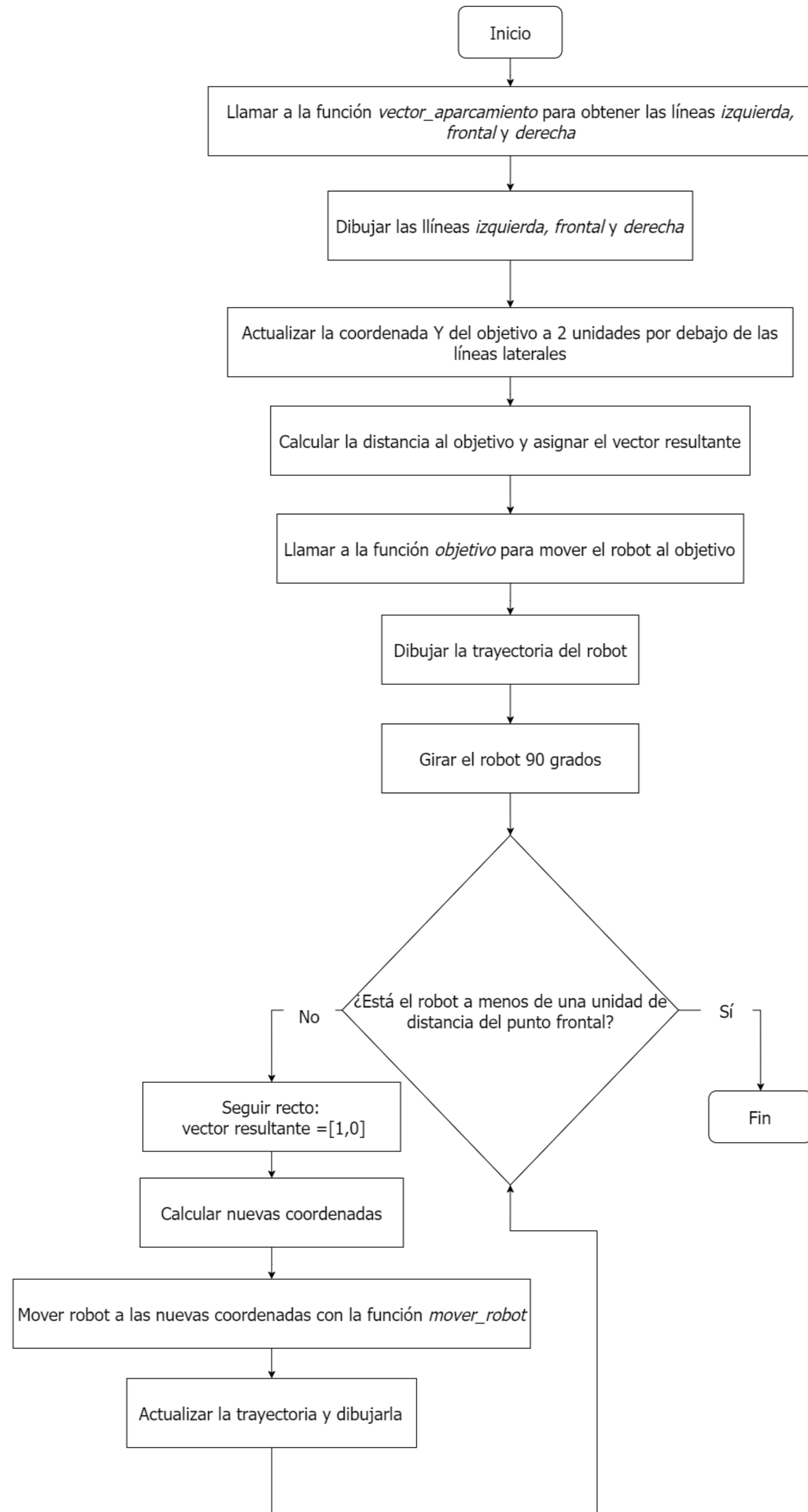
	Fecha	Apellidos, Nombre	Firmas
Diseñado	Jun 2024	Peñarrubia Zamora, Laura	
Dibujado	Jun 2024	Peñarrubia Zamora, Laura	
Comprobado	Jun 2024	Peñarrubia Zamora, Laura	

ESCUELA TÉCNICA SUPERIOR DE INGENIERÍA AEROESCPACIAL Y DISEÑO INDUSTRIAL

Título del proyecto: Implementación de una arquitectura funcional para robots móviles en MATLAB.

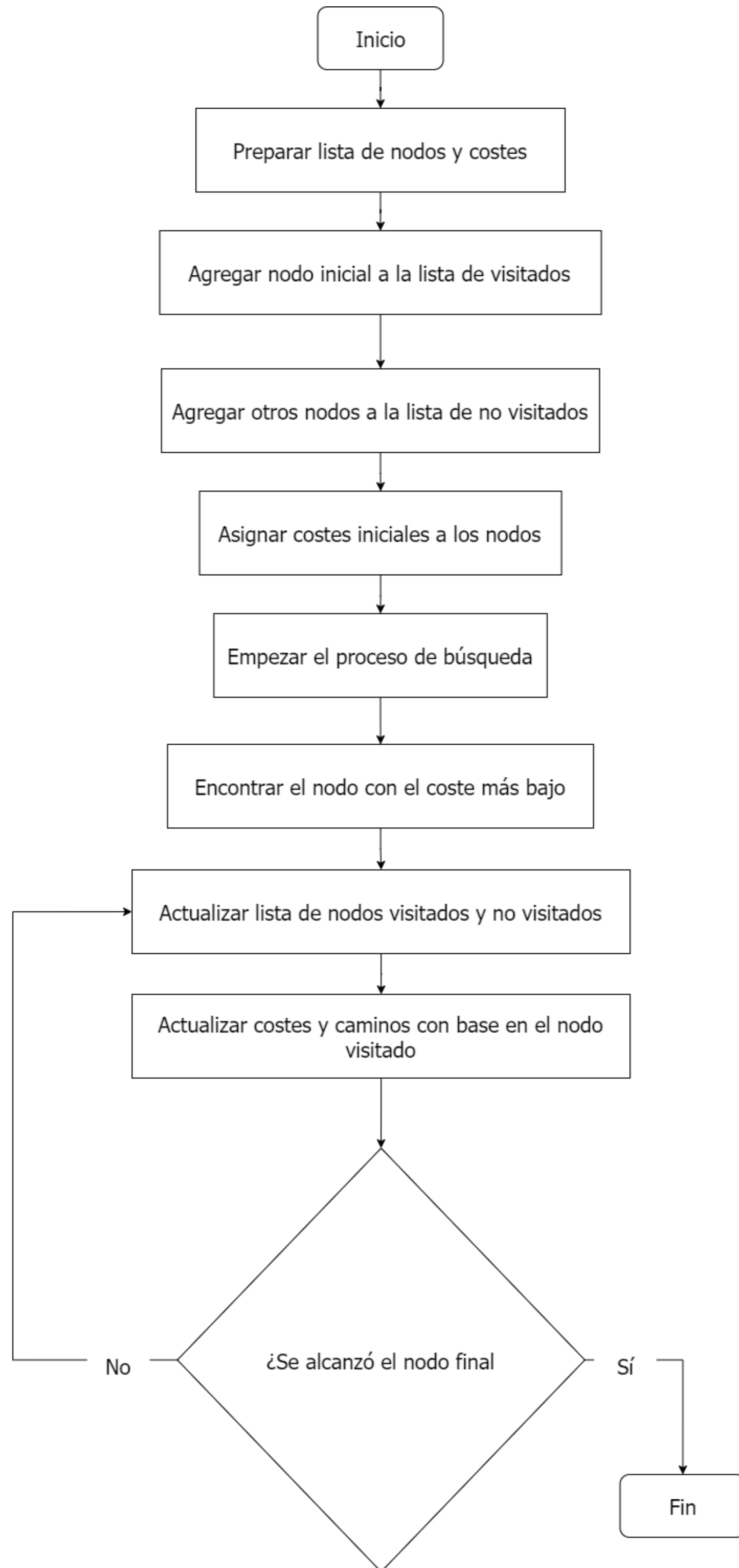
Autor: Peñarrubia Zamora, Laura

Escala	Nombre del plano	Número del plano
	Diagrama de flujo de la función vector aparcamiento de la clase robot	16/26



	Fecha	Apellidos, Nombre	Firmas	ESCUELA TÉCNICA SUPERIOR DE INGENIERÍA AEROESCPACIAL Y DISEÑO INDUSTRIAL
Diseñado	Jun 2024	Peñarrubia Zamora, Laura		Título del proyecto: Implementación de una arquitectura funcional para robots móviles en MATLAB.
Dibujado	Jun 2024	Peñarrubia Zamora, Laura		
Comprobado	Jun 2024	Peñarrubia Zamora, Laura		Autor: Peñarrubia Zamora, Laura

Escala	Nombre del plano	Número del plano
	Diagrama de flujo de la función aparcar de la clase robot	17/26



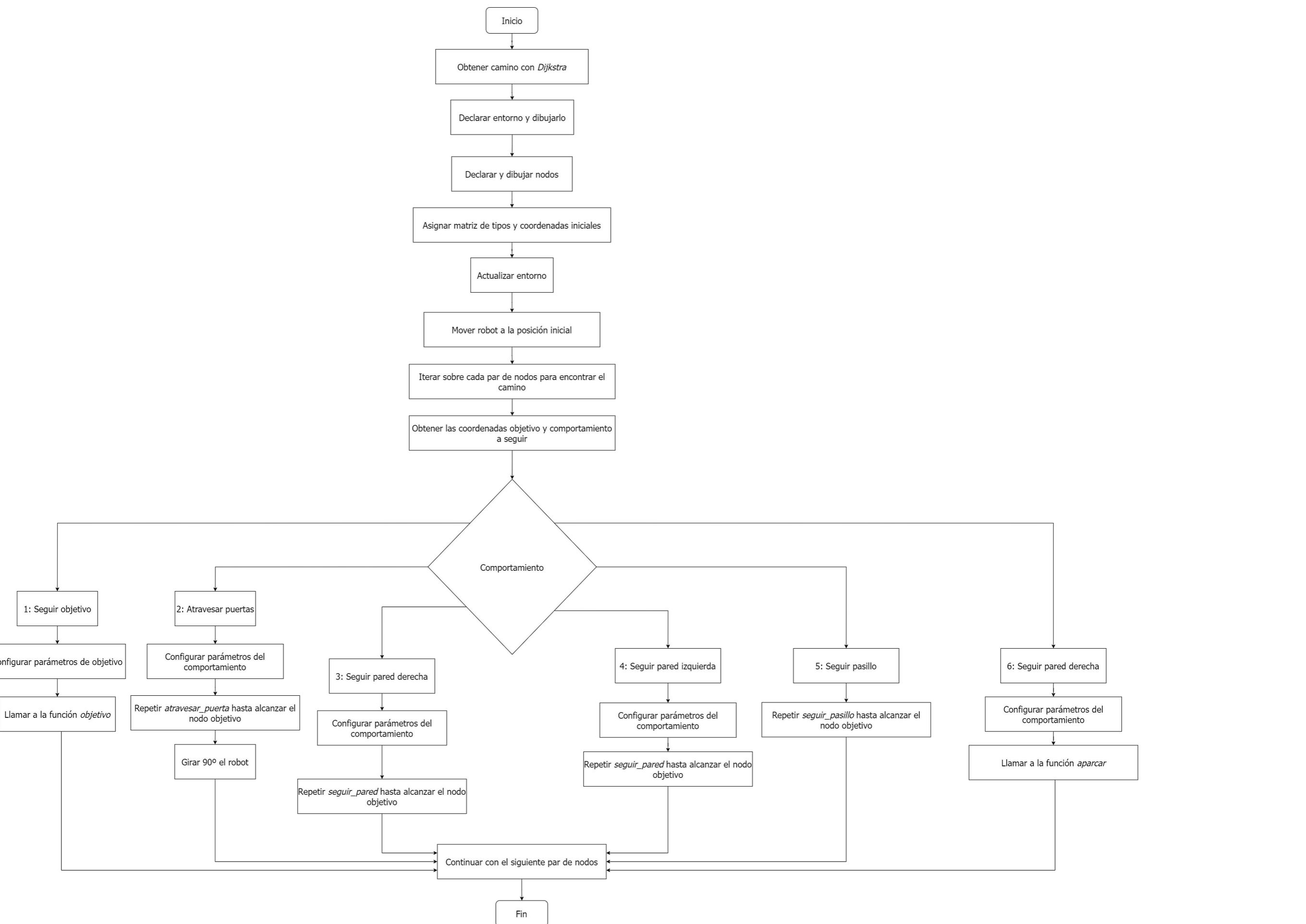
	Fecha	Apellidos, Nombre	Firmas
Diseñado	Jun 2024	Peñarrubia Zamora, Laura	[firma]
Dibujado	Jun 2024	Peñarrubia Zamora, Laura	[firma]
Comprobado	Jun 2024	Peñarrubia Zamora, Laura	[firma]

ESCUELA TÉCNICA SUPERIOR DE INGENIERÍA AEROESCPACIAL Y DISEÑO INDUSTRIAL

Título del proyecto: Implementación de una arquitectura funcional para robots móviles en MATLAB.

Autor: Peñarrubia Zamora, Laura

Escala	Nombre del plano	Número del plano
	Diagrama de flujo de la función Dijkstra	18/26



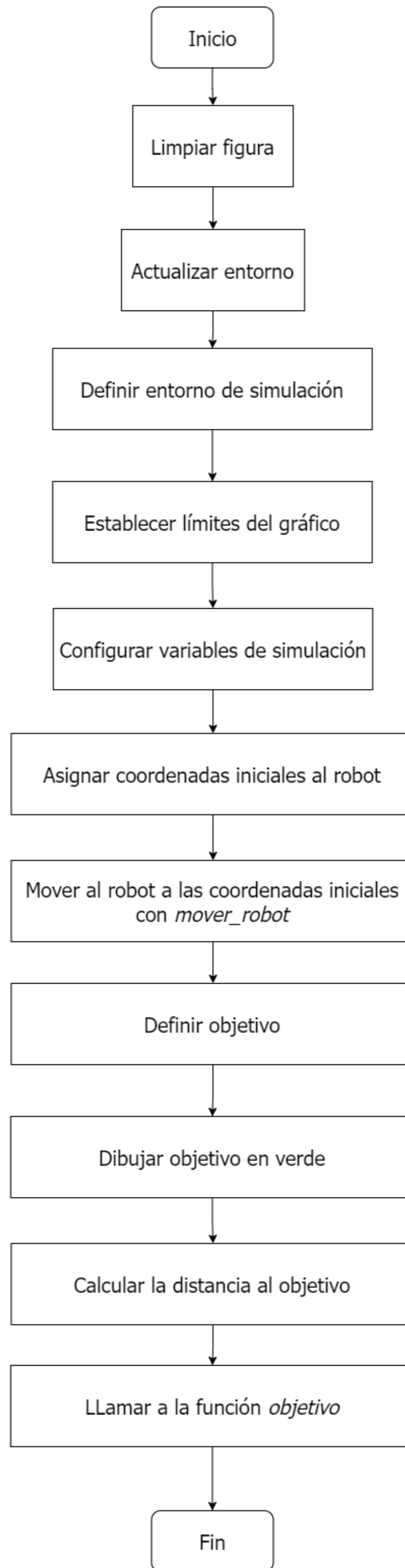
	Fecha	Apellidos, Nombre	Firmas
Diseñado	Jun 2024	Peñarrubia Zamora, Laura	
Dibujado	Jun 2024	Peñarrubia Zamora, Laura	
Comprobado	Jun 2024	Peñarrubia Zamora, Laura	

ESCUELA TÉCNICA SUPERIOR DE INGENIERÍA AEROESCPACIAL Y DISEÑO INDUSTRIAL

Título del proyecto: Implementación de una arquitectura funcional para robots móviles en MATLAB.

Autor: Peñarrubia Zamora, Laura

Escala	Nombre del plano	Número del plano
	Diagrama de flujo de la función piloto de la clase robot	19/26



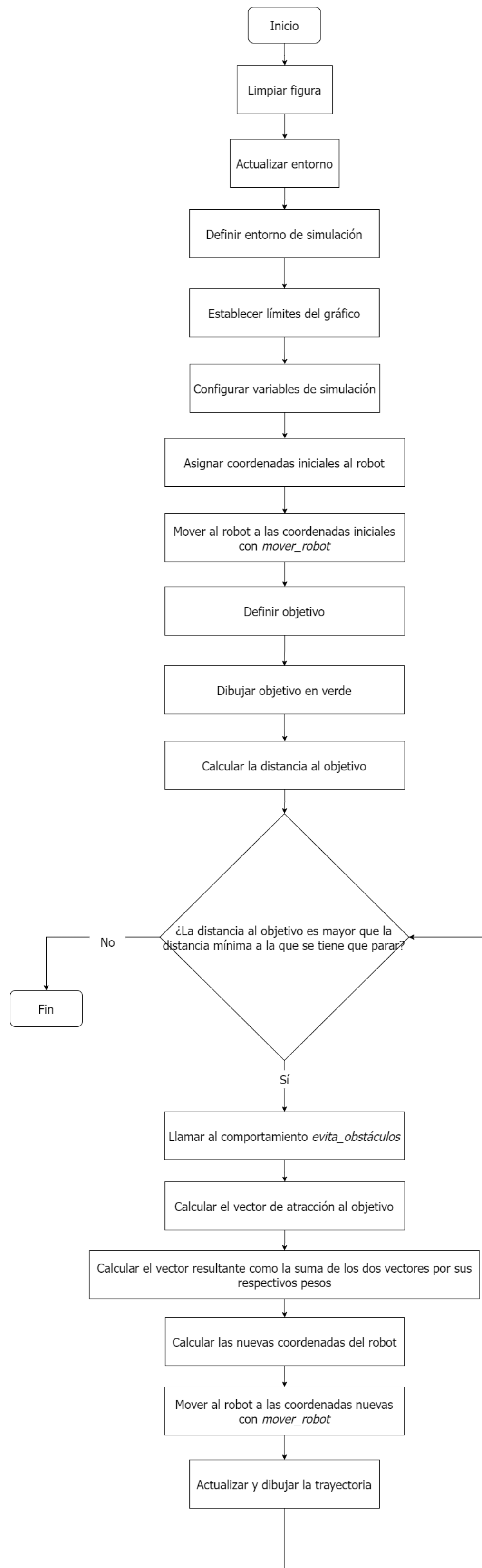
	Fecha	Apellidos, Nombre	Firmas
Diseñado	Jun 2024	Peñarrubia Zamora, Laura	
Dibujado	Jun 2024	Peñarrubia Zamora, Laura	
Comprobado	Jun 2024	Peñarrubia Zamora, Laura	

ESCUELA TÉCNICA SUPERIOR DE INGENIERÍA AEROESCPACIAL Y DISEÑO INDUSTRIAL

Título del proyecto: Implementación de una arquitectura funcional para robots móviles en MATLAB.

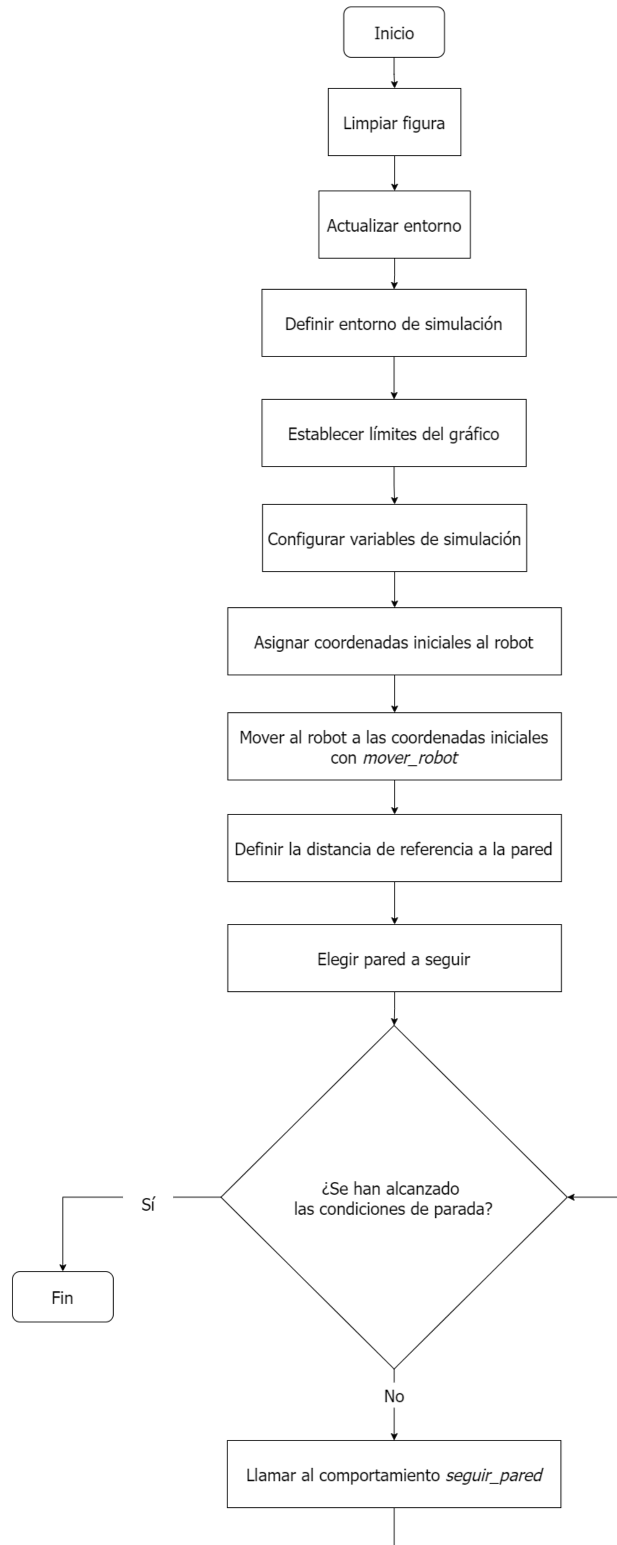
Autor: Peñarrubia Zamora, Laura

Escala	Nombre del plano	Número del plano
	Diagrama de flujo de la función simulación objetivo	20/26



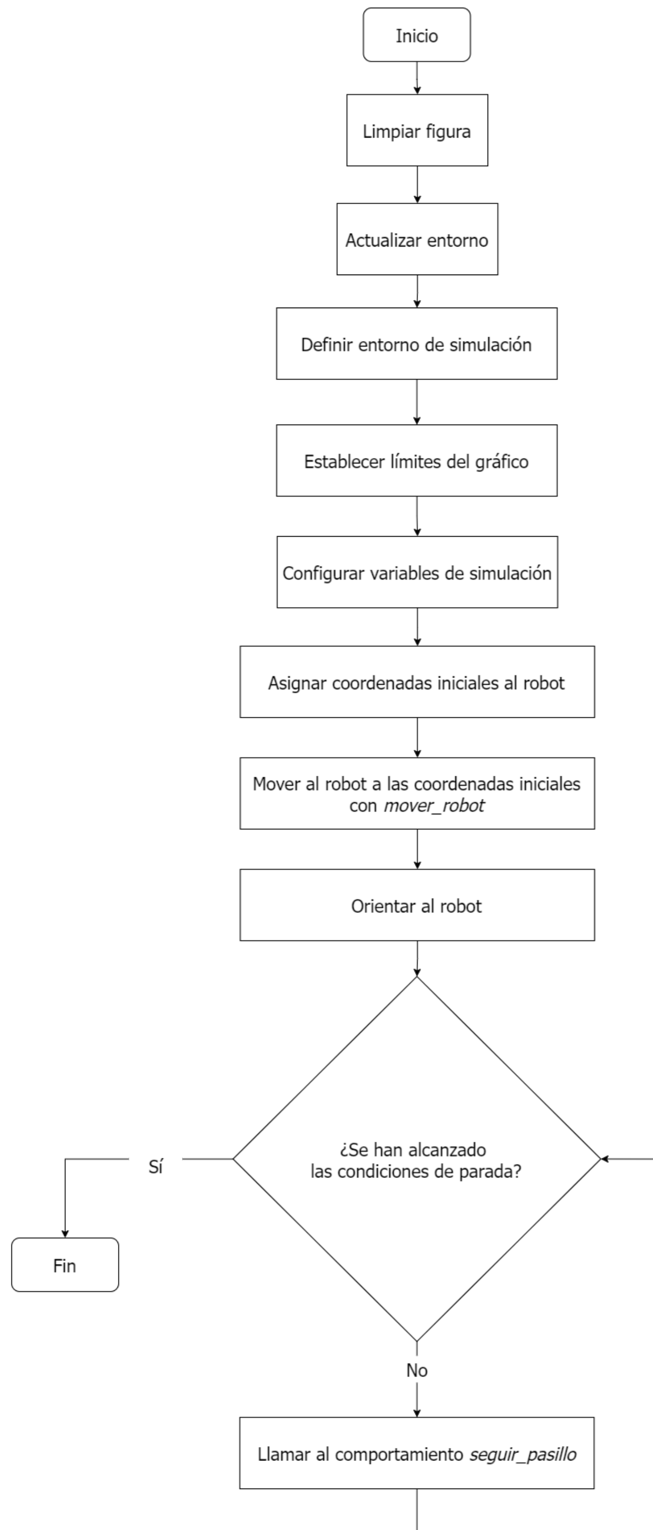
	Fecha	Apellidos, Nombre	Firmas	ESCUELA TÉCNICA SUPERIOR DE INGENIERÍA AEROESCPACIAL Y DISEÑO INDUSTRIAL
Diseñado	Jun 2024	Peñarrubia Zamora, Laura		Título del proyecto: Implementación de una arquitectura funcional para robots móviles en MATLAB.
Dibujado	Jun 2024	Peñarrubia Zamora, Laura		
Comprobado	Jun 2024	Peñarrubia Zamora, Laura		Autor: Peñarrubia Zamora, Laura

Escala	Nombre del plano	Número del plano
	Diagrama de flujo de la función simulación evita obstáculos	21/26



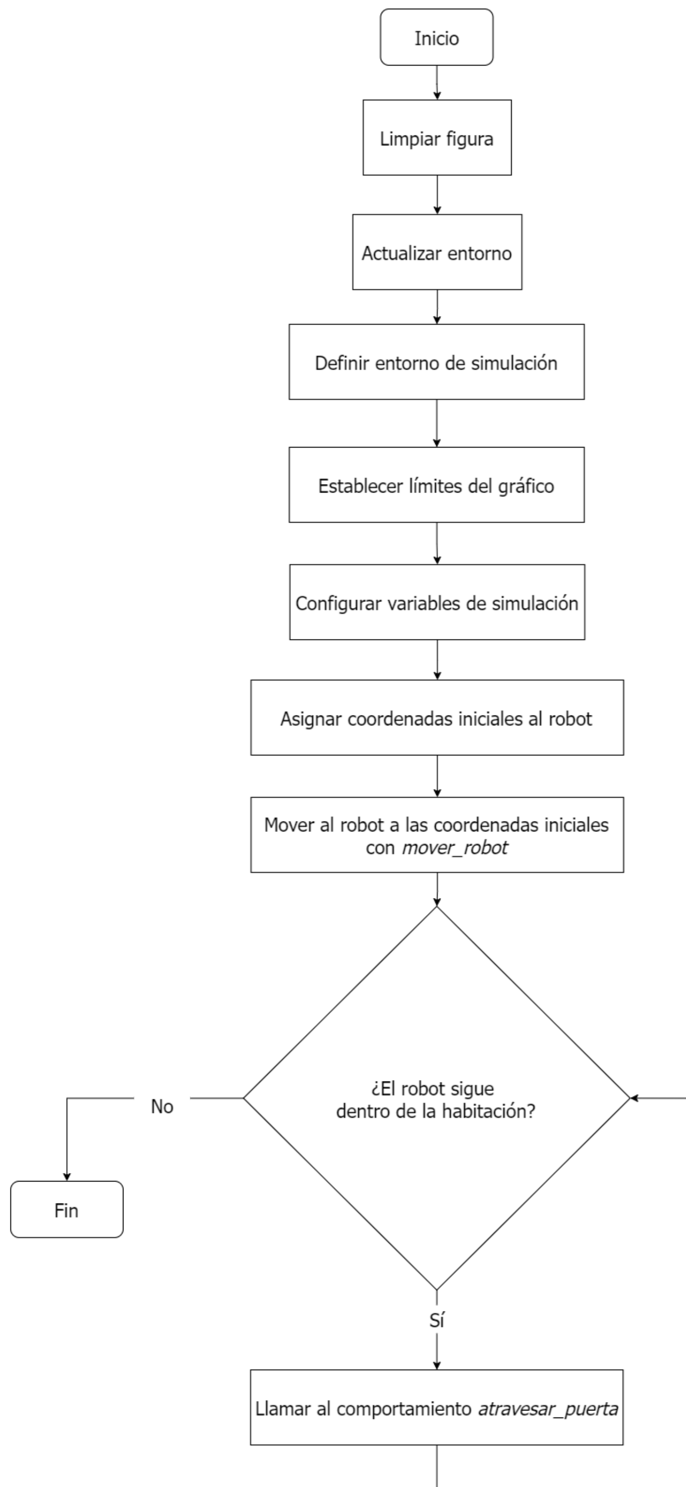
	Fecha	Apellidos, Nombre	Firmas	ESCUELA TÉCNICA SUPERIOR DE INGENIERÍA AEROESCPACIAL Y DISEÑO INDUSTRIAL
Diseñado	Jun 2024	Peñarrubia Zamora, Laura		Título del proyecto: Implementación de una arquitectura funcional para robots móviles en MATLAB.
Dibujado	Jun 2024	Peñarrubia Zamora, Laura		
Comprobado	Jun 2024	Peñarrubia Zamora, Laura		Autor: Peñarrubia Zamora, Laura

Escala	Nombre del plano	Número del plano
	Diagrama de flujo de la función simulación seguir pared	22/26



	Fecha	Apellidos, Nombre	Firmas	ESCUELA TÉCNICA SUPERIOR DE INGENIERÍA AEROESCPACIAL Y DISEÑO INDUSTRIAL
Diseñado	Jun 2024	Peñarrubia Zamora, Laura		Título del proyecto: Implementación de una arquitectura funcional para robots móviles en MATLAB.
Dibujado	Jun 2024	Peñarrubia Zamora, Laura		
Comprobado	Jun 2024	Peñarrubia Zamora, Laura		Autor: Peñarrubia Zamora, Laura

Escala	Nombre del plano	Número del plano
	Diagrama de flujo de la función simulación seguir pasillo	23/26



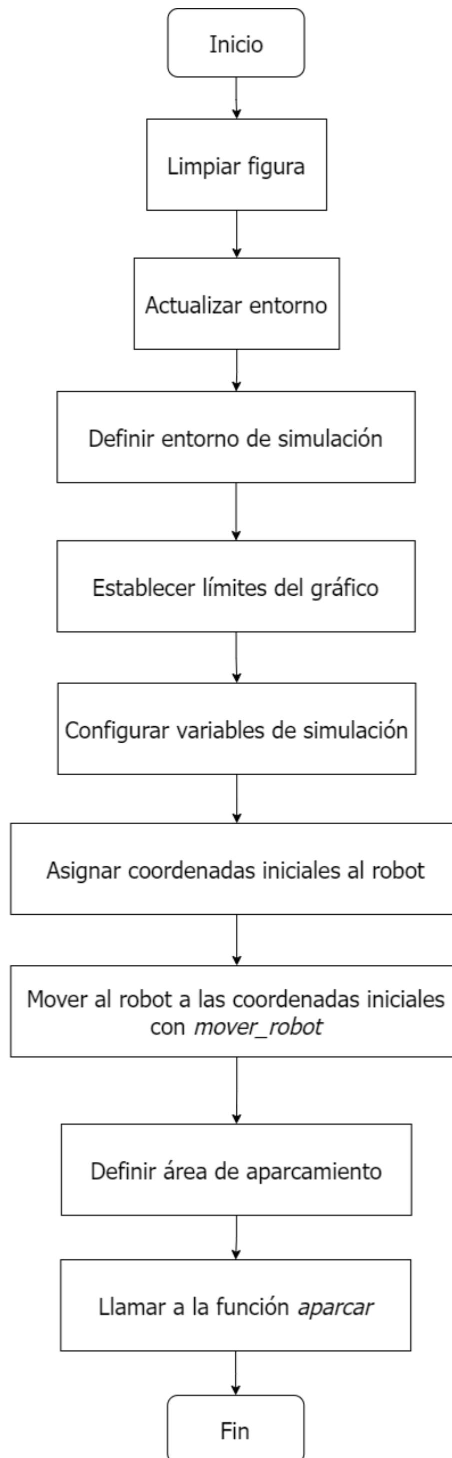
	Fecha	Apellidos, Nombre	Firmas
Diseñado	Jun 2024	Peñarrubia Zamora, Laura	Laura
Dibujado	Jun 2024	Peñarrubia Zamora, Laura	Laura
Comprobado	Jun 2024	Peñarrubia Zamora, Laura	Laura

ESCUELA TÉCNICA SUPERIOR DE INGENIERÍA AEROESCPACIAL Y DISEÑO INDUSTRIAL

Título del proyecto: Implementación de una arquitectura funcional para robots móviles en MATLAB.

Autor: Peñarrubia Zamora, Laura

Escala	Nombre del plano	Número del plano
	Diagrama de flujo de la función simulación atravesar puertas	24/26



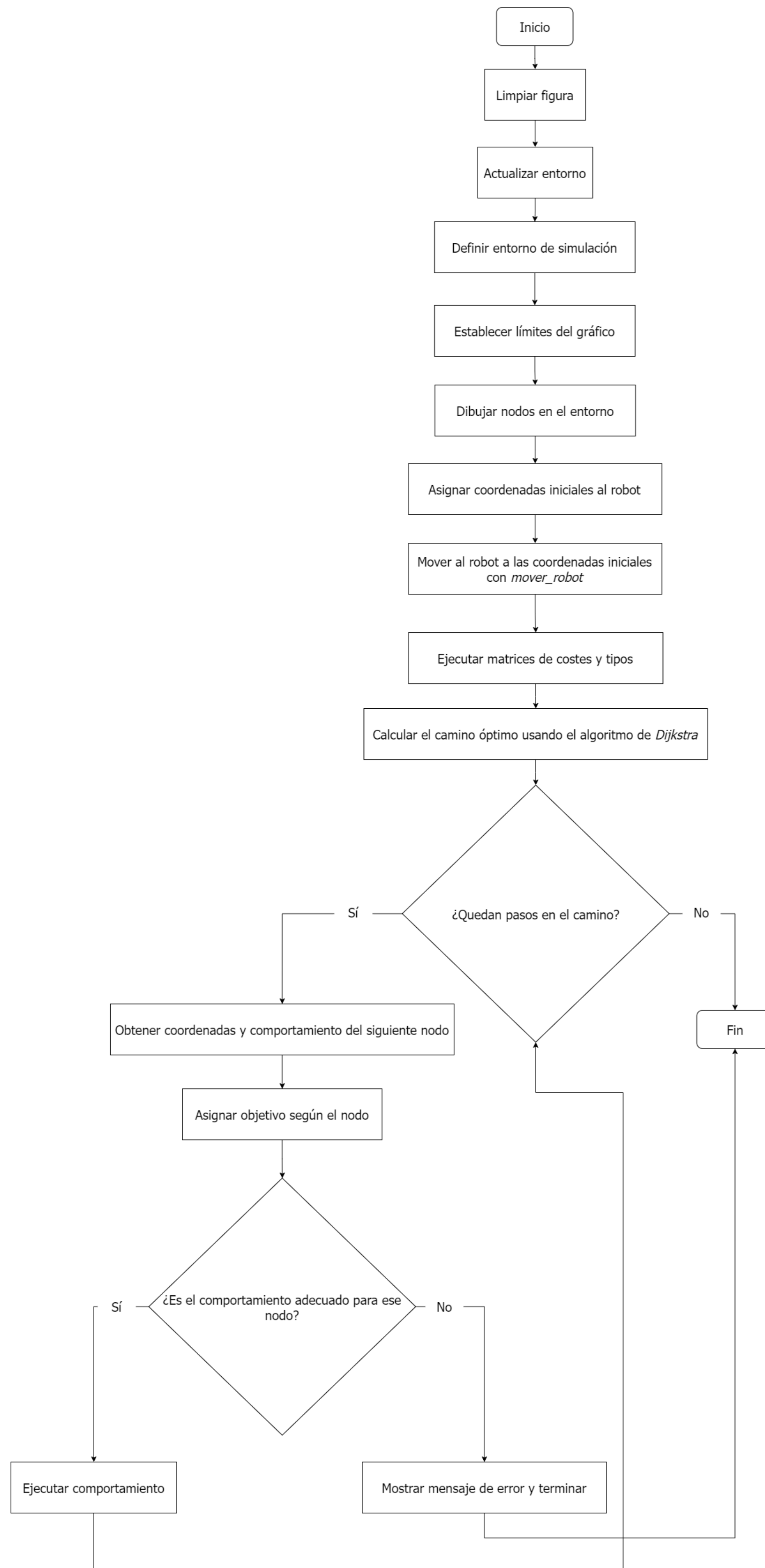
	Fecha	Apellidos, Nombre	Firmas
Diseñado	Jun 2024	Peñarrubia Zamora, Laura	
Dibujado	Jun 2024	Peñarrubia Zamora, Laura	
Comprobado	Jun 2024	Peñarrubia Zamora, Laura	

ESCUELA TÉCNICA SUPERIOR DE INGENIERÍA AEROESCPACIAL Y DISEÑO INDUSTRIAL

Título del proyecto: Implementación de una arquitectura funcional para robots móviles en MATLAB.

Autor: Peñarrubia Zamora, Laura

Escala	Nombre del plano	Número del plano
	Diagrama de flujo de la función simulación aparcar	25/26



	Fecha	Apellidos, Nombre	Firmas	ESCUELA TÉCNICA SUPERIOR DE INGENIERÍA AEROESCPACIAL Y DISEÑO INDUSTRIAL
Diseñado	Jun 2024	Peñarrubia Zamora, Laura		Título del proyecto: Implementación de una arquitectura funcional para robots móviles en MATLAB.
Dibujado	Jun 2024	Peñarrubia Zamora, Laura		
Comprobado	Jun 2024	Peñarrubia Zamora, Laura		Autor: Peñarrubia Zamora, Laura

Escala	Nombre del plano	Número del plano
	Diagrama de flujo de la función simulación piloto	26/26

3. Pliego de condiciones

UNIVERSITAT POLITÈCNICA DE VALÈNCIA

Escuela Técnica Superior de Ingeniería Aeroespacial y Diseño Industrial

Pliego de condiciones

Implementación de una arquitectura funcional para robots móviles en Matlab

Grado en Ingeniería Electrónica Industrial y Automática

AUTORA: Peñarrubia Zamora, Laura

TUTOR: Zotovic Stanisic, Ranko

CURSO ACADÉMICO: 2023/2024

ÍNDICE

1. Objeto **2**
2. Condiciones de los materiales **2**
2.1. Descripción 2
2.2. Control de calidad 2
3. Condiciones de la ejecución **3**
3.1. Descripción 3
3.2. Control de calidad 3
4. Pruebas de servicio **3**

1. Objeto

Esta especificación se refiere a las condiciones necesarias para el diseño e implementación de una arquitectura para robots móviles en el software MATLAB. El proyecto únicamente contempla el ámbito de desarrollo en el programa elegido. Queda excluido todo lo que se refiera a la implementación física de esta arquitectura en un entorno real.

2. Condiciones de los materiales

2.1. Descripción

Los sistemas de software requeridos para la elaboración de este proyecto son los siguientes:

- **Licencia de MATLAB**

  Esta licencia es necesaria para desarrollar, simular y probar la arquitectura del robot móvil dentro del entorno de MATLAB.

- **Licencia de Microsoft Office 365**

  Esta licencia es requerida para la documentación y presentación de los resultados del proyecto, así como para la elaboración de informes y demás documentos relacionados.

Por otra parte, el material físico necesario para crear la arquitectura es el siguiente:

- **Ordenador**

  Para ejecutar el software requerido, como MATLAB y Office.

2.2. Control de calidad

El control de calidad se realiza para asegurar que los materiales utilizados para el proyecto cumplen con los requisitos necesarios para ejecutar la arquitectura. Los procedimientos del control de calidad incluyen:

- **Verificación del software**

  Asegurar que las licencias de MATLAB y Microsoft Office 365 están correctamente instaladas, actualizadas y funcionando según lo esperado. Se realizarán pruebas de funcionamiento iniciales y periódicas para detectar y corregir cualquier problema técnico.

- **Verificación del hardware (ordenador)**

  Se verificará que el ordenador en uso puede ejecutar eficientemente MATLAB y Microsoft Office 365. Esto puede incluir pruebas de rendimiento y la verificación de que el software no presenta problemas de lentitud o fallos recurrentes debido al equipo informático.

3. Condiciones de la ejecución

3.1. Descripción

La ejecución del proyecto se llevará a cabo en un entorno de desarrollo basado en MATLAB. El objetivo principal es diseñar y simular una arquitectura para robots móviles, asegurando que todas las funcionalidades requeridas sean implementadas y verificadas dentro del software. El proceso incluirá la creación de funciones, entornos de simulación y la documentación de todos los procedimientos y resultados obtenidos.

3.2. Control de calidad

La verificación del proyecto se realizará a través de pruebas y simulaciones en MATLAB, asegurando que la arquitectura diseñada cumpla con los requisitos especificados. Se utilizarán herramientas de depuración y verificación de código para comprobar el correcto funcionamiento de los ficheros.

4. Pruebas de servicio

Las pruebas de servicio se llevarán a cabo para asegurar que la arquitectura del robot móvil cumple con los requisitos y funciona según lo esperado en el entorno de simulación de MATLAB. Estas pruebas incluyen, los siguientes aspectos:

- **Pruebas de funcionalidad:** Verificar que todas las funciones y ficheros implementados operan correctamente en diferentes escenarios de simulación.

- **Pruebas de rendimiento:** Evaluar el funcionamiento del sistema en términos de tiempo de respuesta, uso de recursos y eficiencia durante la simulación.

- **Pruebas de regresión:** Realizar pruebas continuas para garantizar que las nuevas modificaciones, como la adición de nuevas funciones que simulen

comportamientos, no afecten negativamente a las funciones previamente implementadas.

Los resultados de estas pruebas serán documentados para identificar posibles mejoras y garantizar que la arquitectura final funcione adecuadamente.

4. Presupuesto

UNIVERSITAT POLITÈCNICA DE VALÈNCIA

Escuela Técnica Superior de Ingeniería Aeroespacial y Diseño Industrial

Presupuesto

Implementación de una arquitectura funcional para robots móviles en Matlab

Grado en Ingeniería Electrónica Industrial y Automática

AUTORA: Peñarrubia Zamora, Laura

TUTOR: Zotovic Stanisic, Ranko

CURSO ACADÉMICO: 2023/2024

ÍNDICE DE CONTENIDO

1. Introducción 3
2. Presupuesto por naturaleza 3
2.1. Cuadro de precios elementales 3
2.2. Cuadro de precios descompuestos 4
2.3. Estado de mediciones 4
2.4. Valoración 5
3. Desglose y justificación del presupuesto 5

ÍNDICE DE TABLAS

Tabla 1. Cuadro de precios elementales. 3
Tabla 2. Cuadro de precios descompuestos 4
Tabla 3. Estado de mediciones 4
Tabla 4. Valoración. 5

1. Introducción

El presente documento detalla el presupuesto necesario para el desarrollo de un sistema de robots móviles utilizando MATLAB. Incluye un desglose de los costos por naturaleza, Además, se proporciona una justificación detallada de cada uno de los componentes incluidos en el presupuesto.

2. Presupuesto por naturaleza

El presupuesto por naturaleza incluye los precios elementales de los recursos requeridos, el estado de las mediciones y una valoración global. Este enfoque permite una clara identificación de cada componente del presupuesto, facilitando su análisis y justificación.

2.1. Cuadro de precios elementales

En esta sección se presenta un cuadro con los precios de los elementos básicos necesarios para el desarrollo del proyecto, incluyendo las licencias de software, el equipo informático y la mano de obra.

1. Cuadro de precios elementales						
Software						
Ref.	**Ud.**	**Descripción**	**Precio(€)**			
m1	Ud.	Licencia de MATLAB	900			
m2	Ud.	Licencia de Microsoft Office 365	69			
Hardware						
Ref.	**Ud.**	**Descripción**	**Precio(€)**	**Vida útil (años)**	**h/año**	**€/h**
m3	Ud.	Ordenador	900	5	1760	0,10
Mano de obra						
Ref.	**Ud.**	**Descripción**	**Precio(€)**			
h1	h.	Ingenieria electrónica	20			

Tabla 1. Cuadro de precios elementales.

Fuente. Propia.

2.2. Cuadro de precios descompuestos

En esta sección se muestran los diferentes objetos que confeccionan el presupuesto general del proyecto, en este caso, hay 3 grupos:

- d1: la programación de la arquitectura en MATLAB.
- d2: el manual de usuario de la arquitectura
- d3: la arquitectura con el manual de usuario incluido

2. Cuadro de precios descompuestos					
Ref.	**Ud.**	**Descripción**	**Precio(€)**	**Cantidad**	**Total**
d1	Ud.	Programación de la arquitectura funcional de un robot móvil en MATLAB.			
m1	Ud.	Licencia de MATLAB	900	1	900
m3	h.	Ordenador	0,10	300	30,68
h1	h.	Ingeniera electrónica	15	300	4500
				Precio e.m.	**5430,68**
Ref.	**Ud.**	**Descripción**	**Precio(€)**	**Cantidad**	**Total**
d2	Ud.	Creación del manual de usuario para su posterior uso.			
m1	Ud.	Licencia de MATLAB	900	1	
m2	Ud.	Licencia de Microsoft Office 365	69	1	
m3	h.	Ordenador	0,1	2	
h1	h.	Ingeniera electrónica	20	2	
				Precio e.m.	**1009,2**
Ref.	**Ud.**	**Descripción**	**Precio(€)**	**Cantidad**	**Total**
d3	Ud.	Arquitectura funcional para un robot móvil en MATLAB con manual de usuario incluido.			
d1	Ud.	Programación de la arquitectura funcional de un robot móvil en MATLAB	5430,68	1	5430,68
d2	Ud.	Creación del manual de usuario para su posterior uso	1009,2	1	1009,20
				Total	**6439,88**

Tabla 2. Cuadro de precios descompuestos

Fuente. Propia.

2.3. Estado de mediciones

En este apartado se muestra el resultado final de este proyecto.

3. Estado de mediciones			
Ref.	**Ud.**	**Descripción**	**Total**
d3	Ud.	Arquitectura funcional para un robot móvil en MATLAB con manual de usuario incluido	1

Tabla 3. Estado de mediciones

Fuente. Propia.

2.4. Valoración

Por último, en esta sección se muestra el precio final del proyecto con los costes indirectos asociados a él (electricidad, agua, instalaciones, etc.) y el IVA.

4. Valoración					
Ref.	**Ud.**	**Descripción**	**Precio**	**Cantidad**	**Total**
d3	Ud.	Arquitectura funcional para un robot móvil en MATLAB con manual de usuario incluido	6439,88	1	6439,88
				Total con costes directos (€)	6439,88
		Costes indirectos		15%	965,98
				Total con costes indirectos(€)	7405,86
		IVA		21%	1555,23
				Coste final(€):	8961,10

Tabla 4. Valoración.

Fuente. Propia.

3. Desglose y justificación del presupuesto

El coste total del proyecto asciende a 8961,10€ incluidos gastos y el IVA. En este apartado se justifica los diferentes precios de este presupuesto.

Primero, en el cuadro de precios elementales, se ha establecido el precio del ordenador en 900€, que es la media de precio de un ordenador de gama media-alta para poder ejecutar MATLAB. Estos ordenadores suelen tener una vida útil en torno a los 5 años, tiempo que se ha incluido para calcular el precio del ordenador en horas. Además, respecto al ordenador, su tiempo de uso al año suele ser de 1760h, contando las 40 horas laborales a la semana e incluyendo un mes de vacaciones. Por otra parte, el precio de las licencias se ha extraído de su propia página web y las dos tienen una vigencia de un año. Por último, la mano de obra de la ingeniera electrónica se ha establecido en 20€ la hora, que es la media de lo que cobra un ingeniero junior en España.

En el cuadro de precios descompuestos se ha establecido 300 horas para la creación de la arquitectura. Esta medida se toma partiendo de la base de que este trabajo de fin de grado tiene estipuladas 360 horas de trabajo.

Por último, en la valoración se tienen en cuenta los costes indirectos asociados al proyecto, como el agua, la luz, las instalaciones, etc. El valor se establece en un 15% sobre el total de los costes directos. Para finalizar, al precio obtenido con los costes indirectos se le suma el 21% de IVA según la legislación española, resultando en un total de 8961.10€.