



Máster en ingeniería de sistemas y control

Sistema láser autónomo para creación y navegación en mapas de interior

Autor

Manuel Rafael Navarro Fuentes

Director

Luis de la Torre Cubillo

Curso académico: 2021/2022

Convocatoria: Septiembre 2022



Máster en ingeniería de sistemas y control

Sistema láser autónomo para creación y navegación en mapas de interior

Proyecto específico propuesto por el alumno

Autor

Manuel Rafael Navarro Fuentes

Director

Luis de la Torre Cubillo

Resumen

En robótica, un problema conocido es el problema de SLAM (simultaneous localization and mapping), que consiste en ser capaz de, mediante el uso de sensores, localizar un robot en un mapa a la vez que sea crea dicho mapa. Este problema hay muchas formas de resolverlo (mediante cámaras, mediante sensores láser, mediante grafos, etc.), pero para nuestro caso, este problema se resolvió en un proyecto anterior (un trabajo de fin de grado realizado por el autor de este proyecto) mediante un sensor láser conocido como LiDAR(light detection and ranging) ya que es muy económico. Dicho sensor realiza mediciones en 360º constantemente y devuelve la distancia de cada medición.

En el proyecto anterior, se implementó una interfaz gráfica para una mayor facilidad de uso y para la potencial mejora futura, con la adición de simulaciones y mediciones extraídas de un robot real. En este proyecto, se creará un entorno de simulación con controles manuales y automáticos para manejar un robot y obtener mediciones del LiDAR en un mapa proporcionado por el usuario.

Estos datos se utilizarán para generar los mapas con SLAM y gracias al mecanismo de manejo automático del robot (o incluso únicamente el manual), se podrán generar mapas de forma sencilla para solventar distintos problemas como: creación automática de mapas de interior, comprobación de distancias reales en comparación con los planos, transporte de materiales en entornos de interior, sistemas móviles de sensores, etc.

En este documento se detallarán distintos aspectos técnicos del desarrollo como la implementación del entorno de simulación, la creación de las interfaces gráficas necesarias y el algoritmo de búsqueda de caminos.

Lista de palabras clave

LiDAR, ROS2, Qt, láser, búsqueda de caminos, entorno de simulación, mapas, SLAM, bajo coste, control manual, robótica, GUI, Astar, A*, enrejado de celdas, descomposición en celdas.

Índice

Índice de tablas	VII
Índice de figuras	X
Índice de algoritmos	XI
1. Introducción	3
1.1. Organización de la memoria	4
1.2. Definición del problema	4
1.2.1. Definición del problema real	5
1.2.2. Definición del problema técnico	5
1.3. Objetivos, restricciones y recursos	10
1.3.1. Objetivos	10
1.3.2. Restricciones	11
1.3.3. Recursos	12
2. Estado del arte	15
2.1. Búsqueda de caminos y métodos de resolución	15
2.2. Descomposición del mapa en celdas (grid)	16
2.3. Simuladores actuales	17
2.4. Sistemas similares al que se pretende construir	18
3. Diseño del sistema	23
3.1. Estructura de ficheros y datos	23
3.1.1. Clase MainWindow	26
3.1.2. Clase Parameters	29
3.1.3. Clases Simulation, cell y node	31
3.1.4. Clase Sim_env	36
3.1.5. Clase JoyPad	38
3.2. Diseño de los algoritmos (búsqueda de caminos y entorno de simulación)	40
3.2.1. Búsqueda de caminos	40
3.2.2. Entorno de simulación	45
3.3. Diseño de la interfaz	47
3.3.1. Modificaciones a la ventana principal	47
3.3.2. Modificaciones a la ventana parámetros	48
3.3.3. Ventana ‘Real map’	49
4. Pruebas	51
4.1. Pruebas de los algoritmos	51
4.2. Discusión de resultados	56
4.2.1. SLAM	56
4.2.2. A-star	56

5. Conclusiones	59
5.1. Conclusiones sobre los objetivos	59
5.2. Conclusiones sobre las pruebas	59
5.3. Futuras mejoras	60
6. Agradecimientos	61
Bibliografía	61

Índice de tablas

1.1. Distribución temporal del proyecto	9
2.1. Métricas de éxito para distintos algoritmos de un robot agrícola en cuanto a cultivos destruidos y desviación del camino óptimo se refiere [29]	21
3.1. Definición de la clase.	25
3.2. Especificación de la clase.	25
3.3. Definición de la clase mainWindow (Nuevas aportaciones).	26
3.4. Definición de la clase mainWindow (Proyecto previo).	26
3.5. Especificación de la clase mainWindow (Nuevas aportaciones).	27
3.6. Especificación de la clase mainWindow (Proyecto previo).	28
3.7. Definición de la clase Parameters (Nuevas aportaciones).	29
3.8. Definición de la clase Parameters (Proyecto previo).	29
3.9. Especificación de la clase Parameters (Nuevas aportaciones).	29
3.10. Especificación de la clase Parameters (Proyecto previo).	30
3.11. Definición de la clase cell.	31
3.12. Especificación de la clase cell.	31
3.13. Definición de la clase node.	32
3.14. Especificación de la clase node.	32
3.15. Definición de la clase Simulation (Nuevas aportaciones).	33
3.16. Definición de la clase Simulation (Proyecto previo).	33
3.17. Especificación de la clase Simulation (Nuevas aportaciones).	34
3.18. Especificación de la clase Simulation (Proyecto previo).	35
3.19. Definición de la clase Sim_env.	36
3.20. Especificación de la clase Sim_env.	37
3.21. Definición de la clase JoyPad.	38
3.22. Especificación de la clase JoyPad.	39
4.1. Resultados métricos para el algoritmo de SLAM fuerza bruta.	53
4.2. Resultados métricos para el algoritmo A*.	54

Índice de figuras

1.1.	RPLIDAR A1M8 [7]	13
1.2.	Mapa realizado con las medidas del LiDAR [7]	14
2.1.	Obtención de un camino mediante un método basado en grid [11]	16
2.2.	Obtención de un camino mediante un método basado en muestreo[12]. Se utiliza un grafo para conectar los puntos origen y final mediante el mapa de caminos.	16
2.3.	Mapa generado por descomposición vertical.	17
2.4.	Mapa generado por enrejado de celdas	17
2.5.	Mapa de líneas generado por una descomposición mediante campo de potencial.	17
2.6.	Turtlebot3.	19
2.7.	SLAM generado por turtlebot3.	19
2.8.	Camino obtenido mediante nav2 en un mapa simulado de turtlebot3.	19
2.9.	Caminos recorridos por un robot usando un algoritmo de evasión de obstáculos con un LiDAR (parte 1) [28]	20
2.10.	Caminos recorridos por un robot usando un algoritmo de evasión de obstáculos con un LiDAR (parte 2) [28]	20
2.11.	Camino seguido por un robot acuático usando tecnología LiDAR para evasión de obstáculos.[30]	20
2.12.	Camino obtenido mediante A* en un mapa predefinido.	21
3.1.	Jerarquía global de ficheros	23
3.2.	Mapa genérico.	41
3.3.	Enrejado de celdas para un mapa genérico.	42
3.4.	Modificaciones a la ventana principal.	47
3.5.	Modificaciones a la ventana parámetros.	48
3.6.	Ventana ‘Real map’.	49
4.1.	Primer mapa diseñado (mapa 1, genérico).	51
4.2.	Segundo mapa diseñado (mapa 2, laberinto).	52
4.3.	Tercer mapa diseñado (mapa 3, alta complejidad).	52
4.4.	Camino obtenido para el mapa 1 (genérico) mediante la heurística de distancia euclíadiana (parámetros 1/e).	54
4.5.	Camino obtenido para el mapa 1 (genérico) mediante la heurística de distancia manhattan (parámetros 1/m).	54
4.6.	Camino obtenido para el mapa 2 (laberinto) mediante la heurística de distancia euclíadiana (parámetros 2/e).	55

4.7. Camino obtenido para el mapa 2 (laberinto) mediante la heurística de distancia manhattan (parámetros 2/m).	55
4.8. Camino obtenido para el mapa 3 (alta complejidad) mediante la heurística de distancia euclíadiana (parámetros 3/e).	55
4.9. Camino obtenido para el mapa 3 (alta complejidad) mediante la heurística de distancia manhattan (parámetros 3/m).	55

Índice de algoritmos

1. find_path(target_x, target_y, robot_x, robot_y, width, height, safety_distance) 44
2. calculate_readings(laser_range, sys_x, sys_y, sys_angle) 46

Acrónimos

LiDAR: Light detection and ranging. Es un tipo de sensor que utiliza la tecnología láser para obtener medidas de distancia del entorno y los obstáculos continuamente en 360º.

SLAM: Simultaneous localization and mapping. Es un concepto que se refiere al problema existente en robótica para localizar a un robot o sistema en un mapa, a la vez que se crea dicho mapa.

Qt: Es un entorno de creación de aplicaciones gráficas orientado a objetos con distintos lenguajes disponibles como c++ o python.

ROS2: Robot operating system v2. El sistema operativo robot es un OS que se utiliza para simplificar el enlace entre distintos componentes software que se ejecutan en robots u ordenadores.

OS: operating system. Un sistema operativo se encarga, a grandes rasgos, de planificar las tareas y los recursos del ordenador.

A*: A estrella, algoritmo de optimización de soluciones mediante heurísticas.

PDS: Product Design Specification es una técnica que consiste en realizar una serie de preguntas para responderlas y así obtener los detalles técnicos del problema.

nav2: Navigation 2 es un sistema de navegación autónoma basado en ROS2.

MORSE: Modular OpenRobots Simulation Engine es un sistema de simulación de robots desarrollado por Laboratoire d'Analyse et d'Architecture des Systèmes ‘ at the University of Toulouse

Capítulo 1

Introducción

Como ya se ha comentado en el resumen, uno de los grandes problemas de la robótica es el SLAM[1, 2, 3], y este problema se resolvió en un proyecto anterior utilizando únicamente un sensor LiDAR (realizado por el autor de este proyecto [4]).

En dicho proyecto anterior, el problema de SLAM se resolvió mediante el uso de un único LiDAR (dada su economicidad [5]) y un mapa de ocupación. El mapa se actualizaba con las medidas obtenidas del LiDAR en función de unos parámetros y esto permite guardar información o .^aprender"de todas las medidas que se han ido obteniendo. Este mecanismo permite obtener un sistema robusto y preciso, que evita guardar información de ruido en un momento dado y permite al sistema identificarse de forma relativamente sencilla a la vez que crear el mapa (en definitiva, realizar SLAM).

El sistema, al hacer uso de un único LiDAR, debía conectarse a un ordenador, y ejecutar un software como interfaz gráfica para crear los mapas. Ese sistema se diseño con 3 funcionalidades: crear mapas reales utilizando el LiDAR, guardar las lecturas del LiDAR a lo largo de una creación de un mapa, y simular la creación de un mapa mediante las medidas previamente guardadas (estas dos últimas permiten experimentar con distintos parámetros utilizando siempre el mismo caso de prueba).

Es un hecho que cada vez más el mundo se está automatizando, y el gran inconveniente del anterior proyecto es que una persona tendría que llevar el LiDAR manualmente y moverse por el entorno de interior para crear el mapa. Por este motivo, una posible solución sería implementar el LiDAR en un robot con capacidad autónoma para facilitar enormemente la tarea de la creación del mapa, o incluso utilizarse para otros fines, como se comentaba también en el resumen.

Ya que implementar un robot desde 0 para realizar todo este tipo de actividades y contar con una aplicación gráfica intuitiva y fácil de usar conllevaría una gran inversión de tiempo, una opción más viable sería implementar un entorno de simulación. Se simularía un robot equipado con un LiDAR, al que podrían enviársele órdenes como el movimiento o la obtención de medidas. Estas medidas serían obtenidas también de forma simulada en un mapa real, pero acorde a los valores que un LiDAR real proporcionaría.

Por lo tanto, el objetivo general de este proyecto, es desarrollar un entorno de simulación con una interfaz gráfica que permita un uso sencillo de un robot genérico equipado con un LiDAR, y a su vez, desarrollar algún algoritmo de búsqueda de caminos que nos permita dirigir al robot a cualquier punto del mapa evitando obstáculos en tiempo real.

1.1. Organización de la memoria

La presente documentación tratará de desglosar las especificaciones necesarias para llevar a cabo el desarrollo e implementación del sistema.

La documentación ha sido dividida como sigue:

- Definición del proyecto: se describirán las características (capítulo 2) y objetivos del proyecto (capítulo 4 sección 4.1), así como se aportará una solución concordante a dichos objetivos.
- Análisis del sistema: se especificarán las distintas partes que compondrán el sistema y se detallarán estas de forma que el sistema quede completamente definido (capítulo 6).
- Pruebas: se detallarán las pruebas que se realizarán sobre el sistema y los resultados obtenidos cuando dicho sistema sea implementado (capítulo 7).
- Conclusiones: se resumirá el trabajo realizado comentando los puntos más importantes y algunos de los problemas encontrados (capítulo 8 secciones 8.1 y 8.2).
- Mejoras y ampliaciones: se comentarán una serie de posibles mejoras y ampliaciones que han surgido a lo largo del desarrollo del proyecto, pero no están incluidos en los objetivos del proyecto, y por motivos técnicos o temporales no se han implementado (capítulo 8 sección 8.3).
- Bibliografía.

1.2. Definición del problema

En este capítulo se tratará de identificar y describir con detalle el problema al que nos enfrentamos. Para ello, se enfocará dicho problema desde dos perspectivas:

- Problema real: se hace referencia al problema general, que podría ser descrito por el usuario al que preocupa dicho problema.
- Problema técnico: se define el problema desde el punto de vista de la ingeniería, transformando el problema real en detalles técnicos que serán útiles para dar una solución a dicho problema.

1.2.1. Definición del problema real

Actualmente, y cada día más, surge la necesidad de disponer de sistemas robots con capacidad autónoma para distintos propósitos (disponer de información del entorno, transporte de materiales, sistemas de sensores móviles, etc.).

Por estos motivos, se plantea un problema de obtención de datos del entorno (SLAM) y de búsqueda de caminos. SLAM fue resuelto en el proyecto anterior, pero, por simplicidad de lectura, se comentará en qué consiste este problema.

Este problema consiste en que un sistema debe ser capaz de localizarse a sí mismo dentro de un mapa que el propio sistema está creando con los datos obtenidos del entorno mediante sensores. El problema viene dado por la falta de sensores que nos digan exactamente dónde se encuentra un sistema dentro del mapa que está creando.

En este trabajo nos centraremos en resolver el problema de la búsqueda de caminos donde, a su vez, existen muy diversas soluciones a este problema, como se comentará en los antecedentes. En nuestro caso, crearemos un entorno de simulación en el que utilizaremos un robot genérico equipado únicamente un sensor LiDAR para SLAM, y la búsqueda de caminos se basará en utilizar estos datos para crear un camino entre dos puntos (inicio y objetivo), y para evitar obstáculos en tiempo real gracias a las mediciones láser.

Se pretende diseñar también un software que proporcione una sencilla interfaz gráfica para uso tanto doméstico como industrial. Este software será fácilmente trasladable a un entorno real, fuera de la simulación.

1.2.2. Definición del problema técnico

Una vez definido el problema real, se tratará de convertir dicho problema en una serie de especificaciones técnicas que nos ayuden a afrontar el problema y obtener una solución desde el punto de vista de la ingeniería.

Para este propósito, se utilizará una técnica muy extendida en la gestión de proyectos que es la técnica PDS (Product Design Specification). Esta técnica consiste en realizar una serie de preguntas para responderlas y así obtener los detalles técnicos del problema.

Funcionamiento

En este apartado se explicará la funcionalidad del sistema que se va a diseñar, expresando de forma técnica las características del problema real.

En primer lugar, como en el apartado anterior, se ha de aclarar que el problema de SLAM se resolvió en el proyecto que se usará como base, pero, por simplicidad de lectura, se explicará también el problema que supone SLAM. La resolución de SLAM mediante únicamente un sensor LiDAR consiste en que el sistema que se diseñe deberá procesar las mediciones láser realizadas por el LiDAR para realizar un mapa en tiempo real. Simultáneamente, con la misma información de las medidas y el mapa, el sistema deberá ser capaz de localizarse a sí mismo utilizando algún algoritmo que le permita contrastar

la información del mapa anterior con las mediciones actuales y volver a actualizar el mapa.

Para este trabajo, por un lado, se desarrollará un entorno de simulación que, dado un mapa proporcionado por el usuario, será capaz de recrear las mediciones de un sensor LiDAR y proporcionarlas. También dispondrá de un robot genérico que será capaz de moverse según los parámetros de movimiento que reciba.

Por otro lado, se creará un sistema de búsqueda de caminos, basado en algún algoritmo de búsqueda de caminos, y con la información del mapa proporcionada por el SLAM que ya ha sido diseñado. Se deberá por lo tanto, a su vez, elegir algún método de descomposición del mapa de ocupación proporcionado por SLAM.

Entorno

Como entorno se entiende el conjunto de aspectos que rodean y condicionan al problema tanto desde el punto de vista software como hardware. A continuación se detallan una serie de características del entorno:

- Entorno de uso del software: este software está destinado a ser utilizado en general para crear mapas en tiempo real de zonas de interior (como pueden ser entornos industriales o domésticos) así como utilizar la navegación del robot en el propio mapa. El sistema diseñado en el proyecto anterior era capaz de realizar SLAM, por lo que en este trabajo se pretende partir de esa útil base para implementar el resto. El software se utilizará en sistemas linux dado que ofrece una mayor compatibilidad con librerías para el desarrollo de este tipo de proyectos.
- Entorno hardware: si el sistema se implementase en hardware, sería necesario disponer de un pequeño robot equipado con un LiDAR, algún dispositivo de comunicación y algún sistema de movimiento, así como baterías. Se comenzó realizando esta parte del desarrollo, pero debido a problemas técnicos y temporales, se acabó paralizando. En el desarrollo de esta parte, se diseñó un módulo software para la conexión de dispositivos de bluetooth de baja energía en Qt y se conectó con una placa arduino. El sistema permitía mover unas ruedas omnidireccionales en un robot simple mediante un joystick.
- Entorno de desarrollo: el sistema se diseñará en Qt creator[6], ya que la interfaz será desarrollada en este sistema, que permite también un sencillo desarrollo de c++ y otros.
- La parte de la interfaz de usuario se realizará también en Qt Creator, ya que es una herramienta muy potente para diseñar interfaces de usuario tanto en sistemas linux como windows. En nuestro caso, como ya se ha comentado, se creará en un sistema linux. Esta herramienta permite grandes opciones de depuración y una gran facilidad de uso para crear y modificar elementos gráficos.

Vida esperada

La estimación de la vida esperada para un proyecto software es compleja, ya que, como se sabe, el ámbito informático sufre constantes cambios y avances día a día. Esta aplicación contará con un entorno de desarrollo fácilmente modificable y ampliable, por lo que puede servir de base para otros proyectos o sistemas. También contará con un algoritmo de búsqueda de caminos y una útil interfaz gráfica de usuario, por lo que una estimación aproximada podría ser de 5 a 8 años de vida.

Ciclo de mantenimiento

En este caso, ya que la aplicación cuenta con una amplia gama de servicios, el mantenimiento se podría dividir en 3 apartados:

- Perfeccionamiento: consistirá en realizar mejoras al software para buscar una mayor eficiencia y preparar el sistema para cambios futuros, hacerlo más tolerante a los cambios en lugar de diseñar un sistema cerrado. Esto le aportará un aumento en la vida esperada.
- Adaptación: consiste en realizar una serie de modificaciones referentes a las necesidades que se planteen en los usuarios.
- Corrección: como en la mayoría de sistemas informáticos, existe la posibilidad de que el sistema contenga algún fallo. En ese caso, se tomarán las medidas necesarias para su corrección.

Competencia

Como se comentará en los antecedentes, existen muchos sistemas que proporcionan mapas 2D utilizando tecnología LiDAR, pero ninguno proporciona un entorno de simulación con un uso tan sencillo y una aplicación gráfica con las distintas funcionalidades.

Aspecto externo

Para la presentación de este proyecto, se crearán una serie de documentos (así como se dispondrá del código) y se podrán encontrar en un repositorio de github para tener un fácil acceso.

La interfaz de usuario será amigable y tendrá unas facilidades de uso como pueden ser los botones naturales y la asociación de botones por funcionalidades. A su vez, se creará dicha interfaz en inglés para optar a una mayor acogida por la comunidad internacional.

Buenas prácticas

Para el desarrollo del código del proyecto se utilizarán una serie de buenas prácticas que se llevan a cabo de forma general en los lenguajes orientados a objetos y en programación:

- Utilizar nombres de variables con nombres explicativos y que quede claro, sin ser muy extensos, cuál es la función de la variable.
- Tabular y en general ordenar el código de forma que sea comprensible y queden claras las diferencias entre partes.
- Relacionado con lo anterior, dividir el código en clases a la hora de un lenguaje orientado a objetos, permitiendo una mayor claridad y entendimiento del código.
- Documentar de forma adecuada el código, así como realizar comentarios cuando se estime oportuno, dejando clara la funcionalidad de todas las variables, funciones, clases, etc.

Calidad y fiabilidad

La calidad y fiabilidad de los proyectos de informática cumplen un papel fundamental debido a la integración de estos en la sociedad. En nuestro caso, esta calidad y fiabilidad se obtendrá gracias a dos medidas:

La primera, para obtener un software de calidad, es necesario recurrir a técnicas estándar como las comentadas en el apartado anterior. Por otro lado, es necesario a su vez realizar un buen diseño del software, que recae en la calidad del proyectista. También es importante, si se diseña un interfaz gráfica, consultar los detalles de diseño con un profesional, o en su defecto, con una persona perteneciente al campo del diseño gráfico.

Para obtener fiabilidad, se llevarán a cabo una serie de pruebas que pretenderán no ser exhaustivas (ya que los casos de prueba serían muy numerosos) pero sí probar gran parte del software.

Programa de tareas

Este trabajo de fin de máster se organizará en las siguientes fases:

- Investigación: en esta fase se incluyen las tareas referentes a la investigación del problema, antecedentes (sistemas similares, posibles soluciones del estado del arte, etc.), la investigación y/o familiarización con las herramientas que se van a utilizar, así como las que se podrían utilizar (opencv, Qt, c++, ROS, etc.) para realizar una comparación entre estas y decidir qué herramientas serán las más adecuadas al problema.

- Análisis y especificación de requisitos: cuando el problema esté descrito, será el momento de pasar a la fase de análisis y especificación de requisitos, donde se especificará de forma más técnica las necesidades de nuestro problema y se podrá contrastar, consecuentemente, el resultado final del proyecto con los requisitos que se han definido.
- Diseño: cuando el problema haya quedado completamente analizado y especificado, se pasará a diseñar una solución que reúna las características que han sido especificadas para dar con la solución óptima al problema.
- Implementación: tras la definición, se implementará dicha definición en un sistema real.
- Evaluación y pruebas: el sistema real se someterá a unas pruebas previamente definidas que contrastarán el resultado del proyecto con las especificaciones y los requisitos definidos en las fases anteriores.
- Documentación: la documentación será una actividad paralela a todo el proyecto, que se realizará a medida que el desarrollo del mismo avance, y con ello sea necesario documentar el progreso.

Si bien es cierto que estas son las fases en las que se organizará el proyecto, bien es sabido que pocas veces la realidad es exactamente como hemos imaginado el proyecto, y generalmente hay que hacer nuevas investigaciones o especificaciones de requisitos conforme el desarrollo avanza. Es decir, estas fases no son fijas en el tiempo, sino que puede haber variaciones debido a problemas dinámicos que pueden aparecer. Sin embargo, el proyectista se compromete a realizar un estudio en profundidad para generar los menores cambios posibles en el futuro.

En la tabla 1.1 se puede ver la distribución temporal del proyecto organizada por fases.

Meses / actividad	Investigación	Análisis	Diseño	Implementación	Evaluación y pruebas	Documentación	Suma
Abril	35	10				15	60
Mayo		10	30			15	55
Junio			25	20		15	60
Julio				50		15	65
Agosto					45	15	60
Horas	35	20	55	70	45	75	300

Tabla 1.1: Distribución temporal del proyecto

Pruebas

Las pruebas que se realizarán serán detalladas en el capítulo de pruebas. De forma general, se puede decir que las pruebas se realizarán tras la implementación del software o durante el desarrollo del mismo y serán de dos tipos:

- Caja blanca: este tipo de pruebas se realizan teniendo en cuenta la estructura interna del software para comprobar si se cumplen todas las especificaciones lógicas. Estas

pruebas se realizarán durante el desarrollo del software para ir comprobando el progreso del proyecto.

- Caja negra: este tipo de pruebas se realizarán para comprobar el funcionamiento final del software completo. Estas pruebas se realizarán al finalizar el software para determinar si cumple las especificaciones y requisitos descritos en la memoria.

Seguridad

Este apartado, aunque forma parte de la especificación de la PDS, no es realmente importante en el proyecto que nos atañe ya que no será necesaria ninguna información personal del usuario. Por lo tanto, el único problema de seguridad que podría haber, sería una mala intencionalidad por parte del proyectista (extracción de datos del usuario sin su consentimiento, como podría ser, por ejemplo, su localización), y dado que el código del software se podrá revisar, se podrá ver que no existe tal intencionalidad.

1.3. Objetivos, restricciones y recursos

1.3.1. Objetivos

En líneas generales, el objetivo de este proyecto es diseñar un entorno de simulación que permita el uso de un robot genérico y la obtención de los valores del LiDAR en un mapa proporcionado por el usuario. También el diseño de una descomposición del mapa y un algoritmo de búsqueda de caminos.

Para ello, es necesario que se cumplan los siguientes objetivos:

- El entorno de simulación debe ser capaz de proporcionar mediciones tipo LiDAR con el mismo tipo de datos y distribución, en el mapa proporcionado por el usuario.
- El entorno de simulación deberá contar con un sistema de manejo manual del robot.
- El entorno de simulación deberá contar con una api para enviar datos de movimiento sobre el robot (para poder ser usado por el algoritmo de búsqueda de caminos).
- El sistema deberá contar con un algoritmo de búsqueda de caminos.
- El sistema deberá ser capaz de modificar ciertos parámetros del algoritmo de búsqueda de caminos, así como de la simulación, para obtener distintos resultados comparables.
- El sistema debe ser capaz de obtener distintas métricas de error de los algoritmos y mostrarlas.
- El sistema probará distintos métodos de comunicación para obtener el más adecuado.

1.3.2. Restricciones

Las restricciones presentes en el ámbito del diseño se dividirán en dos tipos (factores dato y factores estratégicos) y no se podrán modificar durante el desarrollo

Factores dato

Son factores dato aquellas restricciones que vienen definidas por la naturaleza o el ámbito del problema, o por el cliente que desee el sistema, y no existen posibilidades a la hora de elegir. Dichas restricciones serían las siguientes para nuestro proyecto:

- Dado que en el proyecto anterior se usó un LiDAR RPLIDAR A1M8, el entorno de simulación simulará las mediciones para este tipo de LiDAR.
- El sistema deberá cumplir los objetivos previamente especificados (Capítulo 1, apartado 1.3.1: Objetivos).
- El sistema será destinado como mínimo a usuarios que utilicen el sistema operativo ubuntu 20.04, siendo posible optar a mayor compatibilidad.
- El desarrollo debe tener una duración aproximada de 300h.
- Las librerías utilizadas serán de código abierto, para permitir un menor coste y una mayor compatibilidad con versiones futuras, así como mayor adaptación y demás ventajas que supone utilizar código abierto.
- Los recursos utilizados serán de los que dispongan tanto el director del trabajo de fin de máster como el proyectista.
- El sistema contará con un control de errores que informe al usuario de la causa del problema ocurrido.

Factores estratégicos

Los factores estratégicos son aquellas restricciones que se plantean a la hora de diseñar y desarrollar el proyecto, pero que entre ellas existen una serie de posibilidades. Son dichas restricciones para nuestro proyecto las siguientes:

- Como lenguaje de programación solo hay dos opciones (C y C++) dado que el software de desarrollo del LiDAR está escrito en C. Se usará C++ debido a su facilidad de integración con distintas herramientas como ROS, opencv y el propio software de desarrollo del LiDAR.
- Como entorno programación se utilizará QtCreator por razones ya descritas (facilidad de depuración, de obtención de datos, etc.).
- Como sistema operativo se usará linux, concretamente la distribución de ubuntu, dada la familiaridad del proyectista con dicho sistema y la facilidad de instalación de las librerías y frameworks necesarios. Además, es un sistema operativo gratuito.

- Para la elaboración de la documentación se ha optado por utilizar overleaf, un editor de LaTeX gratuito en línea que permite control de versiones y almacenamiento en la nube. Para la presentación, se utilizará la herramienta gratuita de google presentaciones.
- Como entorno de desarrollo para la interfaz de usuario se utilizará Qt creator, un potente framework de desarrollo para aplicaciones gráficas, que cuenta con muchas ayudas de depuración, así como señalizaciones y ayudas a la hora de codificar para facilitar el desempeño y conseguir un menor número de errores sintácticos y por lo tanto un menor tiempo empleado. Qt además podría utilizarse para desarrollar aplicaciones para windows y es de código abierto y gratuito.

1.3.3. Recursos

Para el desarrollo del proyecto se emplearán los siguientes recursos. Estos recursos pueden estar sujetos a cambios, que se notificarán en la memoria del proyecto:

Recursos software

Los siguientes recursos software se utilizarán para el desarrollo, pruebas y documentación del proyecto:

- **Sistema operativo:** Ubuntu LTS 18.04, ubuntu LTS 20.04.
- **Librerías y frameworks:** opencv, Qt, RPLIDAR sdk, levmarq.
- **Lenguaje de programación:** C++.
- **Entornos de desarrollo:** Visual Studio Code, Qt creator.
- **Editor de texto:** para la documentación se utilizará el editor de texto en línea overleaf y dicha documentación se creará mediante L^AT_EX.

Recursos humanos

Para el desarrollo de este proyecto se requerirá la participación del proyectista Manuel Rafael Navarro Fuentes, el cual se encargará del diseño, implementación y, en general, desarrollo del proyecto y del director del proyecto Luis de la Torre Cubillo, que se encargará de la supervisión del avance del proyecto y aportará ideas y correcciones al mismo.

Recursos hardware

Para desarrollar el programa que genere el entorno de simulación y realice SLAM, el algoritmo de búsqueda de caminos, y la interfaz gráfica de usuario para utilizarlo, se hará uso de un portátil personal con las siguientes características:

- **Procesador:** AMD RYZEN 9 4900h (8 core, 3.3GHz, 4MB L2 Cache)
- **Memoria RAM:** 16GB DDR4

- **Sistema de almacenamiento:** 1TB SSD

Para el sistema de mapeo se modelará la salida que proporcionaría un LiDAR: RPLIDAR A1M8 de slamtec[7]. Este LiDAR, como se puede ver en la página web, dispone de un rango de detección máximo aproximado de 12m, y mínimo de unos 15cm, lo cuál para entornos de interior normalmente es más que suficiente. Su capacidad de procesamiento es de 8000 muestras por segundo y se puede variar según las necesidades del usuario.

El LiDAR en cuestión es el siguiente:



Figura 1.1: RPLIDAR A1M8 [7]

Cada muestra medida con el LiDAR contiene 4 datos:

- Angle_z_q14: ángulo desde el que se tomó la medida en representación z.
- Dist_mm_q2: distancia en milímetros desde el LiDAR hasta un obstáculo.
- Quality: calidad de la medida.
- Flag: syncbit.

Estas medidas permiten representar puntos en un mapa como se puede ver en la siguiente imagen:

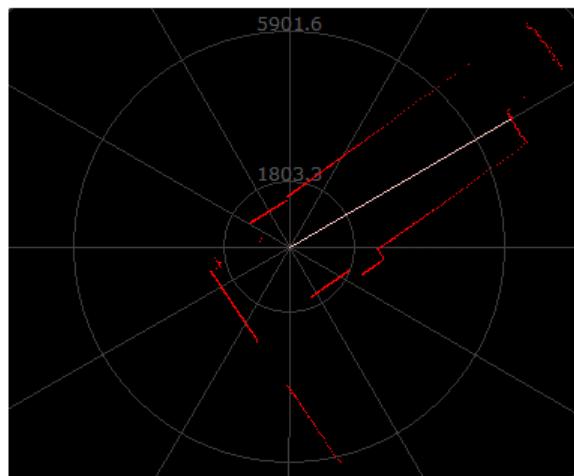


Figura 1.2: Mapa realizado con las medidas del LiDAR [7]

Capítulo 2

Estado del arte

Este capítulo se dividirá en cuatro apartados: primero se explicará el problema de la búsqueda de caminos en robótica y qué métodos de resolución o técnicas se pueden aplicar. Después, se comentarán distintas aproximaciones a la descomposición de mapas (parcialmente necesaria para la búsqueda de caminos). Seguidamente, se comentarán los principales simulares actuales y por qué nuestro problema no se encuentra dentro del marco de uso de estos simuladores. Por último, se expondrán ejemplos de sistemas similares al que se pretende construir y se comentarán diferencias y similitudes con el sistema que se pretende construir.

2.1. Búsqueda de caminos y métodos de resolución

La búsqueda de caminos es un problema a la orden del día en robótica, que ya se ha resuelto de muchas formas distintas, pero ninguna de ellas es completamente eficaz para los distintos tipos de problemas que se puedan plantear.

El problema consiste en, dado un robot, que este sea capaz de navegar hacia un punto determinado dentro de un mapa sin ningún otro tipo de interacción con el usuario. Para realizar este tipo de tarea, el robot podría simplemente realizar movimientos aleatorios hasta llegar a su destino pero esto, evidentemente, no es óptimo ni realista. Con el fin de obtener un resultado realista y factible, es necesario planificar una ruta libre de obstáculos e interferencias.

La planificación de una ruta se puede llevar a cabo haciendo uso de distintos tipos de algoritmos, pero los dos grandes tipos de algoritmos con los que se suele resolver son: algoritmos basados en grid [8] y algoritmos basados en muestreo [9]. Los primeros se basan en encontrar el camino óptimo dado un mapa descompuesto en celdas (o grid), aunque necesitan una gran cantidad de memoria para almacenar todas las celdas e información sobre estas. Los segundos se basan en generar nuevos nodos en un espacio de estados e intentar conectarlos entre sí, por lo que si el problema es de una gran dimensionalidad, este tipo de algoritmos puede ser una buena solución. [10]

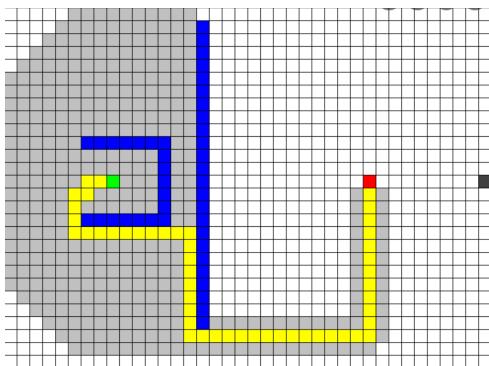


Figura 2.1: Obtención de un camino mediante un método basado en grid [11]

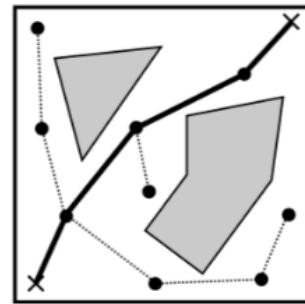


Figura 2.2: Obtención de un camino mediante un método basado en muestreo[12]. Se utiliza un grafo para conectar los puntos origen y final mediante el mapa de caminos.

Para nuestro problema, ya que los mapas que se generarán serán relativamente pequeños (mapas de interior), no tendríamos que preocuparnos de la dimensionalidad, por lo que podemos perfectamente usar un algoritmo basado en grid. Uno de los algoritmos más comunes para este tipo de problemas es el algoritmo A*, ya que permite obtener una buena solución en un tiempo reducido. Este algoritmo se explicará con más detalle en el capítulo 3, sección 3.2.

2.2. Descomposición del mapa en celdas (grid)

En el apartado anterior se especifica que el algoritmo utilizado para este proyecto se basará en celdas, pero también hay distintos métodos para descomponer el mapa en celdas. A grandes rasgos, una descomposición del mapa en celdas, permite que un robot posicionado en una celda transicione a otra adyacente sin chocar con ningún obstáculo, siempre que la celda adyacente se encuentre disponible (es decir, no sea un obstáculo en sí misma). A continuación se comentan distintos tipos de descomposición de mapas en celdas [13]:

- **Descomposición exacta:** este tipo de descomposición se basa en descomponer el espacio libre en una colección de regiones cuya unión es exactamente el espacio libre. Uno de los tipos más comunes de esta descomposición es la llamada "vertical", representada en la figura 2.3.
- **Descomposición aproximada:** en este caso, el espacio libre no se representa de forma exacta, sino que habrá celdas ocupadas en las que solo una parte de estas esté ocupada. Una de las descomposiciones más común de este tipo es el enrejado en celdas, como se representa en la figura 2.4.
- **Campos de potencial:** este método se basa en considerar al robot como una partícula bajo la influencia de un campo de potencial artificial, cuyas variaciones reflejan la "estructura" del espacio libre. En la figura 2.5 se puede ver el diagrama de líneas generado por un campo de potencial.

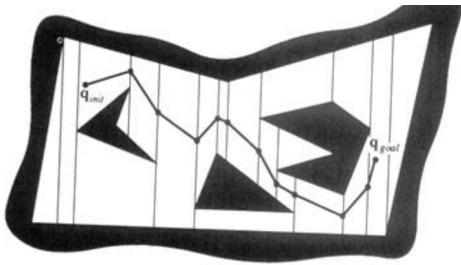


Figura 2.3: Mapa generado por descomposición vertical.

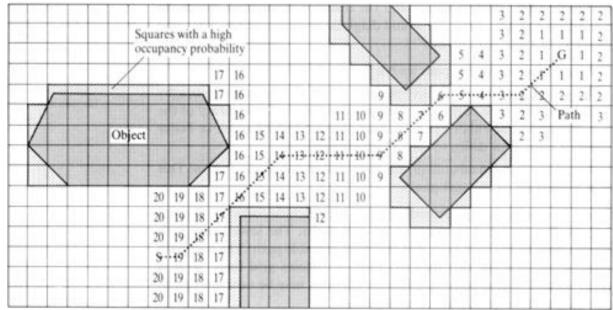


Figura 2.4: Mapa generado por enrejado de celdas

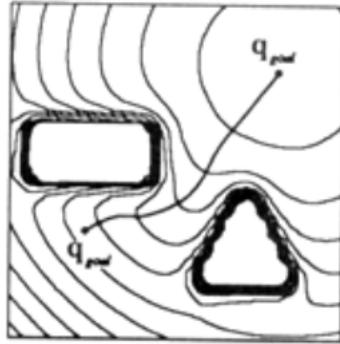


Figura 2.5: Mapa de líneas generado por una descomposición mediante campo de potencial.

Como se puede intuir mediante las figuras, una de las implementaciones más sencillas y robustas es el enrejado en celdas, al que además se puede aplicar el algoritmo A*. En este caso, cada nodo que el algoritmo intenta optimizar sería una celda, y la distancia entre nodos sería de 1, ya que el coste de moverte de una celda a otra siempre es el mismo al ser celdas idénticas. Por estos motivos, se ha decidido utilizar este tipo de descomposición para este proyecto.

Por lo tanto, la búsqueda de caminos se podrá realizar gracias a la información generada mediante SLAM del proyecto anterior, descomponiendo el mapa obtenido en celdas, y obteniendo el camino óptimo mediante algoritmo A*.

2.3. Simuladores actuales

En la actualidad, existen infinidad de simuladores robots, pero nos basaremos en algunos artículos que se han encargado de estudiar y resumir las características, ventajas y desventajas de muchos de estos simuladores.

Un simulador con varias funcionalidades y ampliamente utilizado en investigación es UWsim [14, 15], desarrollado por el grupo IRS en España. Este simulador dispone del motor físico Bullet [16] y pone a disposición del usuario distintos tipos de robots con brazos o partes rígidas que pueden interactuar con el entorno. Sin embargo, este simulador

carence de la funcionalidad de sensor láser tipo LiDAR.

MORSE [14, 17], un simulador desarrollado en Francia y basado en el motor de juegos Blender [18]. Este simulador dispone de una funcionalidad similar a UWsim, pero dispone también, entre otros añadidos, de un sensor láser tipo LiDAR. Sin embargo, la curva de aprendizaje de blender es tan escarpada que llevaría mucho tiempo a cualquier persona aprender a usar blender antes de poder realizar ningún tipo de simulación.

Entre los grandes simuladores, tenemos también a Gazebo [14, 19], el cuál ha sido extensamente usado en investigación, ya que es de código abierto y está fuertemente ligado a ROS. Este simulador tiene muchas más opciones en cuanto a físicas y sensores/actuadores, ya que permite implementar nuestros propios sensores mediante una API, y dispone de varios motores físicos en los que puede ser compilado. Dispone también de un sensor láser tipo LiDAR. El principal problema de este simulador, a pesar de todas sus ventajas, es el uso de ROS, ya que no es sencillo conectarlo con un entorno de desarrollo gráfico como Qt. Otra de las desventajas es, como con MORSE, que tiene cierta curva de aprendizaje.

Motores gráficos dedicados a juegos como unreal engine [20] o unity [21] pueden ser una buena opción dado su alto nivel de personalización, pero cuentan con la desventaja de no implementar actualmente simulaciones de ningún tipo de robots [14, 22].

Por último, Simbad es un simulador similar al que se pretende construir, donde no existe un motor físico como tal, sino simplemente una comprobación de colisión con objetos en un mundo 2D. Sin embargo, este simulador no permite utilizar sensores láser y no está abierto a personalización, por lo que no es de ninguna utilidad para este proyecto [22].

En definitiva, existen algunos simuladores muy complejos que permiten realizar las tareas que se buscan, pero la curva de aprendizaje para comenzar a usar estos simuladores suele ser bastante escarpada. Esto influye, por tanto, en el tipo de usuarios que pueden usar el software de simulación, siendo muy complejo para un usuario novicio o ajeno al campo de la robótica usar este tipo de software.

Por otro lado, los simuladores simples, generalmente carecen de las funcionalidades que se buscan, especialmente la generación de las mediciones láser del LiDAR.

2.4. Sistemas similares al que se pretende construir

En robótica, y más últimamente, es relativamente sencillo encontrar robots con capacidad autónoma, pero estos no están diseñados generalmente para un usuario final, sino para investigación o educación. Otro de los problemas que presentan este tipo de robots es la simulación, ya que no son accesibles para todo el mundo y no disponen de un entorno gráfico sencillo para un usuario final.

Dentro de estos sistemas actuales, se encuentran dos ampliamente usados en educación e investigación[23, 24]: turtlebot [25] y nav2 [26].

El turtlebot es un robot didáctico equipado con un LiDAR, en adición a otros muchos

sensores, capaz de realizar SLAM y está dotado de cierta autonomía. En las figuras 2.6 y 2.7, se puede ver al turtlebot y un mapa generado mediante su SLAM.

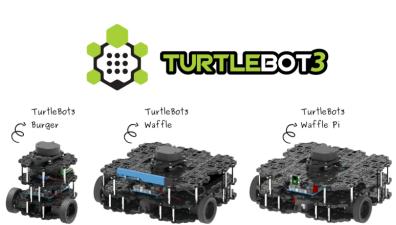


Figura 2.6: Turtlebot3.

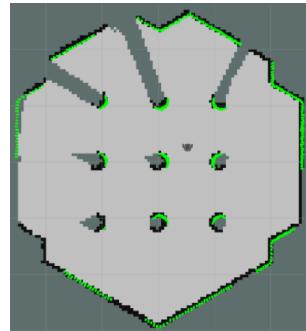


Figura 2.7: SLAM generado por turtlebot3.

Nav2 es un sistema de navegación autónoma para robots que utiliza ros2[27] y pretende ser un estándar para la navegación segura en robots autónomos. Además de la navegación autónoma, dispone de otras características relacionadas con la navegación. En la figura 2.8 se puede ver un ejemplo de un camino obtenido mediante nav2.

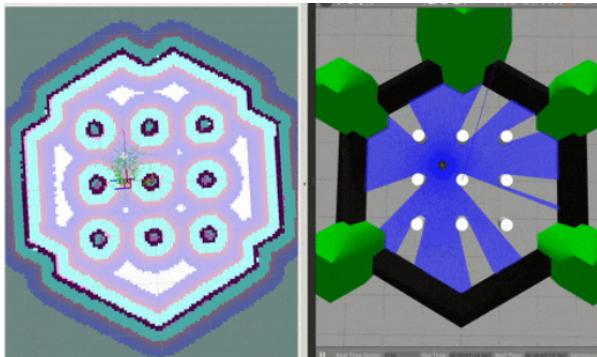


Figura 2.8: Camino obtenido mediante nav2 en un mapa simulado de turtlebot3.

Podemos encontrar otro gran grupo de sistemas publicados en revistas científicas, pero la mayoría de las veces estos únicamente cuentan con una de las funcionalidades deseadas para nuestro sistema, ya que implementar todas ellas requiere una gran cantidad de tiempo y esfuerzo.

Hay muchos ejemplos de cómo utilizar un LiDAR para la evasión de obstáculos, ya que se obtienen unas medidas bastante fiables y a una gran velocidad. [28, 29].

En el artículo publicado por Huixu Dong et al.[28] podemos ver cómo utilizando únicamente un LiDAR, se puede evitar de forma robusta la colisión con obstáculos. En este caso, se realizan una serie de experimentos y se puede comprobar que el robot nunca choca (figuras 2.9 y 2.10), por lo que es posible utilizar únicamente un LiDAR para la evasión de obstáculos. Sin embargo, en su caso no crear un método SLAM y no explican qué tipo de búsqueda de camino están usando, por lo que la información útil para nuestro proyecto sería conocer la capacidad de evasión de obstáculos que nos permite un LiDAR.

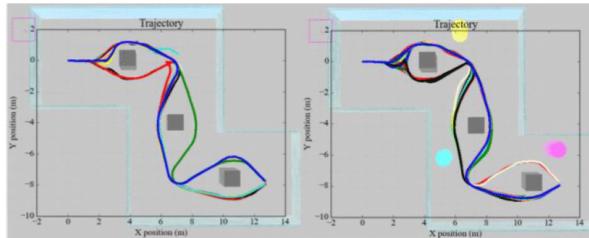


Figura 2.9: Caminos recorridos por un robot usando un algoritmo de evasión de obstáculos con un LiDAR (parte 1) [28]

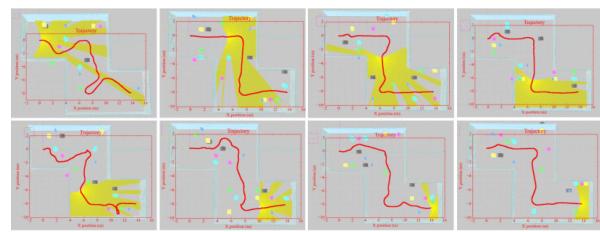


Figura 2.10: Caminos recorridos por un robot usando un algoritmo de evasión de obstáculos con un LiDAR (parte 2) [28]

La tecnología LiDAR puede ser usada también para evasión de obstáculos en robots acuáticos, donde el único cambio a realizar sería el control del robot [30]. Como se puede apreciar en la figura 2.11, el sistema consigue evadir obstáculos de forma eficaz, y únicamente se modificaría el control del robot.

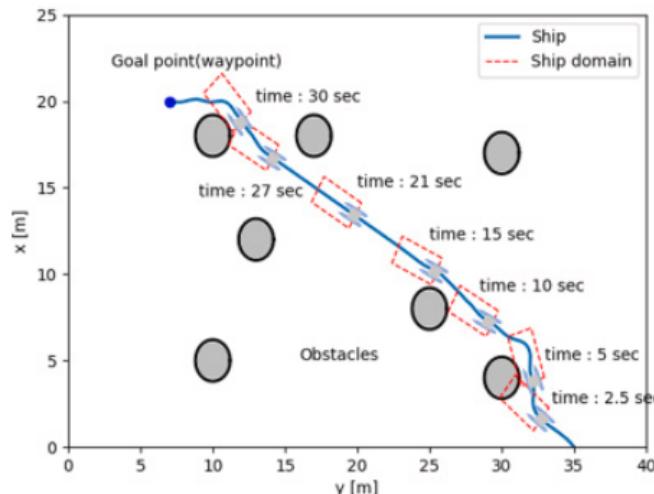


Figura 2.11: Camino seguido por un robot acuático usando tecnología LiDAR para evasión de obstáculos.[30]

También existen implementaciones de robots agrícolas que usan la tecnología LiDAR para evitar chocar con los cultivos y destrozarlos, usando un algoritmo para diferenciar cultivos de malas hierbas [29]. En la tabla 2.1 se puede ver cómo usan esta tecnología para conseguir que el robot no destruya ningún cultivo y se desvíe lo mínimo posible del camino óptimo.

Algorithms	RUBY filtered	RUBY only	OPAL filtered	OPAL only	PEARL filtered	PEARL only	RANSAC filtered	RANSAC only
Success Rate	100%	90%	70%	60%	80%	0%	30%	100%
Success rate without crushing plants	100%	100%	70%	50%	80%	0%	0%	100%
Average error	2.33	7.20	3.98	6.26	1.41	NaN	10.82	4.51
Maximal error	3.39	10.94	7.39	8.67	3.17	NaN	13.44	5.59
Minimal error	1.28	3.49	1.59	2.23	1.04	NaN	6.65	2.73

Tabla 2.1: Métricas de éxito para distintos algoritmos de un robot agrícola en cuanto a cultivos destruidos y desviación del camino óptimo se refiere [29]

Sin embargo, este tipo de sistemas se alejan en cierta medida del sistema que queremos construir, ya que no se pretende encontrar ningún tipo de patrón o reconocimiento de obstáculos. En definitiva, aunque se aleje de nuestra meta, este sistema representa el potencial del LiDAR para ser utilizado como único sistema en la detección y evasión de obstáculos.

Hay también muchos ejemplos de sistemas simples que utilizan LiDAR y muchos otros de búsqueda de caminos, pero no han sido sometidos a ningún tipo de revisión. Además, la mayoría de veces, estos sistemas carecen de muchas funcionalidades, por lo que, aunque pueden ser útiles como punto de partida, no son muy fiables en cuanto a rigor científico.

El siguiente es un ejemplo de sistema que combina LiDAR y búsqueda de caminos pero carente de SLAM. Este sistema usa un mapa predefinido y un LiDAR para evitar obstáculos, a la vez que calcula el camino mediante el algoritmo A*. En la figura 2.12 se puede ver un camino elegido por el algoritmo A* [31].

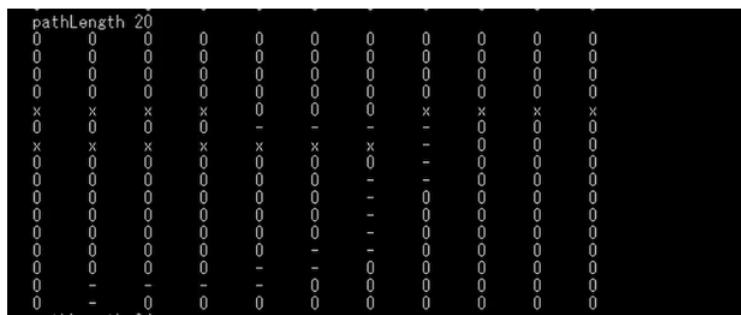


Figura 2.12: Camino obtenido mediante A* en un mapa predefinido.

Como se puede observar, no es tarea sencilla encontrar un sistema que contenga los puntos clave de nuestro proyecto (integración de SLAM, búsqueda de caminos y simulación del robot, facilidad de uso para el usuario final, capacidad de mejora, etc.), por lo que, tras la realización de este proyecto, se espera contar con un sistema decentemente completo, apto para el uso tanto de usuarios expertos como novicios, y allanar el camino de futuras investigaciones.

Capítulo 3

Diseño del sistema

En este capítulo se presentará la estructura y el funcionamiento del software que se va a diseñar. Se detallarán los ficheros necesarios, las distintas clases existentes, los algoritmos que se han creado y las interfaces que utilizará el software.

3.1. Estructura de ficheros y datos

En esta sección se explicará la jerarquía de ficheros y las clases pertenecientes a cada archivo.

La jerarquía global de ficheros se puede observar en la figura 3.1.

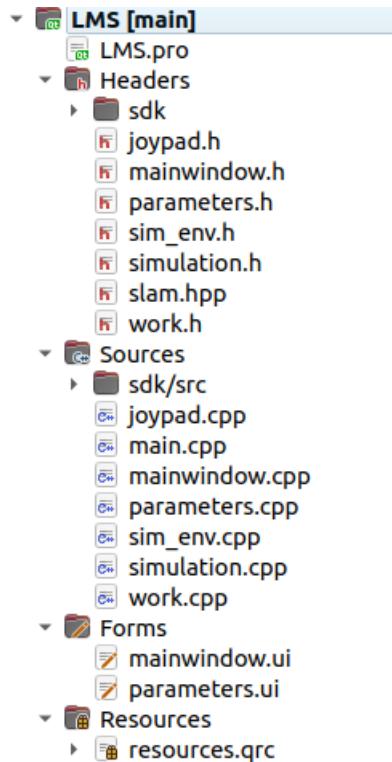


Figura 3.1: Jerarquía global de ficheros

Como se puede observar, hay 5 tipos de ficheros:

- .pro: este fichero es único y define una serie de directrices para que la herramienta disponible en Qt, qmake, cree un makefile para compilar todos los archivos de forma correcta.
- Headers: estos ficheros contendrán las definiciones de las distintas clases que se usarán y algunos de los métodos de cada clase.
- Sources: estos ficheros contendrán el código principal de las clases.
- Forms: estos ficheros serán los encargados de definir la interfaz gráfica de las ventanas necesarias para nuestra aplicación.
- Resources: este fichero contiene información sobre los recursos que usará el software. En este caso, únicamente el ícono de la aplicación.

Dentro de estos ficheros, los más importantes en este proyecto son los pertenecientes a las clases main_window, simulation y sim_env, donde residen las funcionalidades principales.

La clase main_window se encargará de controlar las conexiones entre las demás clases y también de mantener la interfaz gráfica de usuario y actualizar los valores pertinentes cuando el usuario realiza alguna acción.

La clase sim_env se encargará de mantener el entorno de simulación del robot. Le llegarán las acciones a través de una API y realizará las acciones pertinentes.

La clase simulation se encargará de realizar SLAM y de enviar los datos de movimiento al entorno de simulación acorde a los movimientos introducidos por el usuario, así como realizar el algoritmo de búsqueda de caminos cuando el usuario desee llevar el robot de un punto a otro de forma autónoma.

A continuación, se explicará detalladamente cada fichero y las funcionalidades que implementa, así como las clases. Se mostrarán las nuevas aportaciones a la clase y seguidamente, la clase tal y como se encontraba en el proyecto anterior. Se detallarán los ficheros header y sus conexiones con los demás ficheros. Para detallar las clases, se usarán las plantillas de las tablas 3.1 y 3.2.

En dichas tablas se pueden apreciar unos elementos que pueden parecer inusuales, los slots y las signals. Ambos pertenecen a un tipo de método de Qt.

En Qt, el par signal-slot se puede interpretar como un paradigma de manejo de interrupciones, donde una clase (o, más concretamente, una instancia de una clase), envía una signal, dando lugar a una interrupción en otra instancia de una clase, que ejecutará el slot asociado a esa signal. De forma general, este tipo de conexiones se pueden interpretar como un sistema de interrupciones sleep-wake_up.

Este par signal-slot se debe conectar al principio de la ejecución de la ventana principal de Qt, para que, durante toda la ejecución, las clases mantengan sus signal-slot correctamente conectados. Se desaconseja la reconexión de signal-slots en tiempo de ejecución ya que puede derivar en comportamientos anómalos del software, a menos que se tenga completo control del software en cuestión.

Nombre de la clase
- Atributos
- Métodos
- Slots
- Signals

Tabla 3.1: Definición de la clase.

Nombre	
Descripción	
Atributos	
Métodos	
Slots	
Signals	

Tabla 3.2: Especificación de la clase.

3.1.1. Clase MainWindow

El primer fichero que nos encontramos es mainwindow.h. En este fichero se define la clase mainWindow, la cuál se encargará de administrar la mayor parte de la interfaz gráfica y la funcionalidad del programa. Los ficheros mainwindow.cpp y mainwindow.ui forman parte de la especificación de la clase mainWindow. El fichero cpp implementa el código principal de las funciones, mientras que el fichero ui implementa la interfaz gráfica de la ventana principal (mainWindow).

MainWindow
- sim_env: Sim_env
- Sin nuevos métodos.
- mousePressEvent()
- path_not_found()
- on_b_end_sim_clicked()
- Sin nuevas signals.

Tabla 3.3: Definición de la clase mainWindow (Nuevas aportaciones).

mainWindow
- param: parameters
- pool: QThreadPool
- work: Work
- simulation: Simulation
- Q1,Q2: Qpoint
- points, count: int
- mainWindow()
- on_b_new_map_clicked()
- on_b_save_map_clicked()
- on_b_save_sim_clicked()
- on_b_restart_clicked()
- on_b_load_clicked()
- on_b_start_stop_clicked()
- btnaction()
- on_b_distance_clicked()
- on_map_button_clicked()
- on_text_spdTextChanged(const QString &arg1)
- print_img(const QImage &img)
- sim_finished()
- error_port()
- Sin signals.

Tabla 3.4: Definición de la clase mainWindow (Proyecto previo).

Nombre	MainWindow
Descripción	Esta ventana se encargará de proporcionar la funcionalidad principal del programa, iniciar los estados necesarios y manejar las ventanas.
Atributos	<ul style="list-style-type: none"> - sim_env: este atributo se encargará de iniciar y mantener el entorno de simulación del robot real.
Métodos	Sin nuevos métodos
Slots	<ul style="list-style-type: none"> - mousePressEvent(): se encarga de recoger el click derecho para la navegación en el mapa. - path_not_found(): cuando el algoritmo de conducción autónoma no encuentra un camino, este método informará al usuario mediante una ventana emergente. - on_b_end_sim_clicked(): cuando la simulación termina, este método reinicia los valores necesarios e informa al usuario.
Signals	Sin nuevas signals.

Tabla 3.5: Especificación de la clase mainWindow (Nuevas aportaciones).

Nombre	mainWindow
Descripción	Esta ventana se encargará de proporcionar la funcionalidad principal del programa (botones para iniciar y guardar mapas, cargar y guardar simulaciones, abrir la ventana de ajustar parámetros, calcular distancias y cambiar la velocidad de simulación).
Atributos	<ul style="list-style-type: none"> - param: parameters -> clase que proporcionará la ventana para modificar los parámetros. - pool: QThreadPool -> variable que permitirá crear varios hilos de ejecución. - work: Work -> clase que permitirá crear mapas reales. - simulation: Simulation -> clase que permitirá manejar simulaciones. - Q1,Q2: QPoint -> puntos que pulsará el usuario en el mapa para calcular distancias. - points: int -> número de puntos que ha clicado el usuario en el mapa (máximo 2). - count: int -> número de medidas cargados desde el fichero de simulación.
Métodos	<ul style="list-style-type: none"> - mainWindow(): constructor, inicializa las clases, points y count.
Slots	<ul style="list-style-type: none"> - on_b_new_map_clicked(): este slot lanza un hilo para empezar la creación de un nuevo mapa - on_b_save_map_clicked(): este slot guarda la imagen del mapa que se está creando y para la creación. - on_b_save_sim_clicked(): este slot guarda los datos en un fichero de simulación. - on_b_restart_clicked(): este slot resetea la aplicación. - on_b_load_clicked(): este slot carga una simulación. - on_b_start_stop_clicked(): este slot comienza/para una simulación. - btnaction(): este slot abre la ventana de parámetros para modificarlos. - on_b_distance_clicked(): este slot calcula la distancia entre puntos y la muestra. - on_map_button_clicked(): este slot muestra la información de un punto clicado por el usuario en el mapa y la guarda. - on_text_spdTextChanged(QString): este slot cambia la velocidad de simulación. - print_img(QImage): este slot muestra la imagen del mapa en la pantalla. - sim_finished(): este slot muestra al usuario que la simulación ha acabado - error_port(): este slot muestra un mensaje de error al no poder conectar con el puerto especificado.
Signals	No cuenta con ninguna señal.

Tabla 3.6: Especificación de la clase mainWindow (Proyecto previo).

3.1.2. Clase Parameters

Parameters
- safety_distance_ : int
- perc_occupancy_ : int
- heuristic_ : char
- Métodos set y get para los nuevos parámetros.
- Sin nuevos slots
- Sin nuevas signals

Tabla 3.7: Definición de la clase Parameters (Nuevas aportaciones).

parameters
- ui : Ui::parameters
- width_, height_, baudrate_, range_x_, range_y_, range_angle_ : int
- angle_var_, x_var_, y_var_, w_ : double
- p_ : slam::position
- port_ : char*
- levmarq_, ignore_human_ : bool
- parameters()
- clear()
- Métodos get y set .
- void on_b_accept_clicked();
- void on_algorithm_currentIndexChanged(int index);
- void on_b_cancel_clicked();
- Sin signals.

Tabla 3.8: Definición de la clase Parameters (Proyecto previo).

Nombre	Parameters
Descripción	Esta clase se encarga de almacenar los parámetros para el correcto funcionamiento del programa.
Atributos	<ul style="list-style-type: none"> - safety_distance_: distancia de seguridad para el movimiento del robot que definirá el tamaño de celda del enrejado de celdas. - perc_occupancy_: porcentaje de ocupación para que una celda del enrejado de celdas se considere ocupada. - heuristic_: heurística que usará el algoritmo A*.
Métodos	<ul style="list-style-type: none"> - Métodos set y get para los nuevos parámetros.
Slots	Sin nuevos slots
Signals	Sin nuevas signals

Tabla 3.9: Especificación de la clase Parameters (Nuevas aportaciones).

Nombre	parameters
Descripción	Esta clase permitirá al usuario modificar los parámetros de forma gráfica y a su antojo, pudiendo dejar todos como predeterminados.
Atributos	<ul style="list-style-type: none"> - <code>ui</code>: variable que proporciona la interfaz gráfica. - <code>width_</code>: ancho del mapa. - <code>height_</code>: alto del mapa. - <code>baudrate_</code>: baudrate de la conexión con el LiDAR. - <code>range_x_</code>: rango de X en el que se estimará la posición. - <code>range_y_</code>: rango de Y en el que se estimará la posición. - <code>range_angle_</code>: rango del ángulo en el que se estimará la posición. - <code>angle_var_</code>: variación del ángulo con la que se harán las distintas estimaciones. - <code>x_var_</code>: variación de X con la que se harán las distintas estimaciones. - <code>y_var_</code>: variación de Y con la que se harán las distintas estimaciones. - <code>w_</code>: peso del valor anterior del pixel para obtener el valor nuevo de ocupación de dicho pixel. - <code>p_</code>: punto inicial del sistema en el mapa. - <code>port_</code>: puerto al que está conectado el LiDAR. - <code>levmarq_</code>: variable que definirá si se usa el algoritmo Levenberg Marquardt o fuerza bruta. - <code>ignore_human_</code>: variable que definirá si el ángulo en el que estaría el humano debe ser ignorado o no.
Métodos	<ul style="list-style-type: none"> - <code>parameters()</code>: constructor que inicializa los parámetros a unos valores predeterminados fijados por el programador. - <code>clear()</code>: restablece los parámetros a sus valores predeterminados y limpia memoria. - Métodos get y set: para visualizar y modificar los parámetros respectivamente.
Slots	<ul style="list-style-type: none"> - <code>on_b_accept_clicked()</code>: establece los valores de los parámetros a los valores especificados por el usuario si estos son correctos. - <code>on_algorithm_currentIndexChanged(int index)</code>: establece el algoritmo que se va a utilizar junto con sus parámetros necesarios. - <code>on_b_cancel_clicked()</code>: cancela la edición, no se tienen en cuenta los valores modificados. - <code>on_ignore_human_stateChanged(int arg1)</code>: slot que establece el parámetro de ignorar el humano dependiendo de la elección del usuario.
Signals	Sin signals

Tabla 3.10: Especificación de la clase Parameters (Proyecto previo).

3.1.3. Clases Simulation, cell y node

cell
- free : bool
- unknown : bool
- center : cv::Point
- x_min : int
- x_max : int
- y_min : int
- y_max : int
- operator = ()
- Sin slots
- Sin signals

Tabla 3.11: Definición de la clase cell.

Nombre	cell
Descripción	Esta clase se encargará de almacenar información sobre las celdas en las que se descompondrá el mapa.
Atributos	<ul style="list-style-type: none"> - free: almacena si la celda está libre o no. - unknown: almacena si la celda es desconocida. - center: almacena la posición del centro de la celda. - x_min: valor mínimo de x que alcanza la celda. - x_max: valor máximo de x que alcanza la celda. - y_min: valor mínimo de y que alcanza la celda. - y_max: valor máximo de y que alcanza la celda.
Métodos	Sin métodos
Slots	Sin slots
Signals	Sin signals

Tabla 3.12: Especificación de la clase cell.

node
- pos : cv::Point
- parent_pos : cv::Point
- on_hold : bool
- distance_walked : int
- heuristic : int
- total_dist : int
- heap : char
- operator = ()
- node()
- Sin slots
- Sin signals

Tabla 3.13: Definición de la clase node.

Nombre	node
Descripción	Esta clase almacena información sobre los nodos que se irán visitando en el algoritmo A*.
Atributos	<ul style="list-style-type: none"> - pos: posición del nodo (número de celda) - parent_pos: posición del nodo padre - on_hold: almacena si se encuentra activo o explorado. - distance_walked: valor de distancia desde el origen hasta el nodo. - heuristic: valor de distancia estimada desde el nodo hasta el punto objetivo. - total_dist: suma de la distancia andada y la heurística. - heap: almacena en qué heap se encuentra el nodo.
Métodos	<ul style="list-style-type: none"> - operator = (): operador de asignación. - node(): constructor.
Slots	Sin slots
Signals	Sin signals

Tabla 3.14: Especificación de la clase node.

Simulation
<ul style="list-style-type: none"> - <code>live_</code>: bool - <code>readings_ready_</code>: bool - <code>navigating_</code>: bool - <code>moving_</code>: bool - <code>safety_distance_</code>: int - <code>perc_occupancy_</code>: int - <code>started_new_path_</code>: bool - <code>heuristic_</code>: char - <code>cells_m_</code>: array of cell - <code>navigation_points_</code>: std::vector<cv::Point> - <code>target_p_</code>: cv::Point
<ul style="list-style-type: none"> - <code>create_cell_map()</code> - <code>canTravel()</code> - <code>compute_cell_occupancy()</code> - <code>find_path()</code> - <code>get_neighbors()</code> - <code>reconstruct_path()</code> - Métodos set y get para los nuevos atributos
<ul style="list-style-type: none"> - <code>update_readings()</code> - <code>moved_robot()</code> - <code>finished_navigation()</code>
<ul style="list-style-type: none"> - <code>wait_for_readings()</code> - <code>move_robot()</code> - <code>path_not_found()</code>

Tabla 3.15: Definición de la clase Simulation (Nuevas aportaciones).

Simulation
<ul style="list-style-type: none"> - <code>running_</code>: bool - <code>started_</code>: bool - <code>finished_</code>: bool - <code>loaded_</code>: bool - <code>levmarq_</code>: bool - <code>ignore human_</code>: bool - <code>start_</code>: int - <code>P_</code>: slam::Position - <code>width_,height_,count_,range_x_,range_y_,range_angle_</code>: int - <code>x_var_,y_var_,angle_var_,w_,speed_</code>: double - <code>mat_</code>: cv::Mat - <code>readings_[]: rplidar_response_measurement_node_hq_t</code>
<ul style="list-style-type: none"> - <code>init()</code> - <code>run()</code> - <code>clear()</code> - Métodos get y set.
- Sin slots.
<ul style="list-style-type: none"> - <code>print_img(QImage)</code> - <code>sim_finished()</code>

Tabla 3.16: Definición de la clase Simulation (Proyecto previo).

Nombre	Simulation
Descripción	Esta clase se encarga de ejecutar simulaciones de SLAM, ya sea utilizando ficheros de simulación o el entorno de simulación sobre un mapa aportado por el usuario.
Atributos	<ul style="list-style-type: none"> - live_: almacena qué tipo de simulación se está ejecutando. - readings_ready_: almacena si las lecturas del LiDAR están listas - navigating_: almacena si el robot está navegando hacia un punto. - moving_: almacena si se ha mandado la orden de movimiento al entorno de simulación. - safety_distance_: almacena la distancia de seguridad con la que se crearán las celdas del enrejado. - perc_occupancy_: almacena el porcentaje de ocupación para tener en cuenta si una celda está ocupada o no. Este porcentaje está basado en el porcentaje de píxeles ocupados dentro de una celda. - started_new_path_: almacena si se comenzó un nuevo camino mientras se navegaba hacia otro punto. - heuristic_: almacena la heurística a usar en A* - cells_m_: array con todas las celdas del mapa y su información. - navigation_points_: vector con el camino que tomará el robot para llegar de un punto P_ a un punto target_p_. - target_p_: punto de destino de navegación.
Métodos	<ul style="list-style-type: none"> - create_cell_map(): inicializa el array de celdas para la descomposición del mapa en un enrejado de celdas. - canTravel(): Método que proporciona información de hacia dónde puede viajar el robot en base a las celdas ocupadas. - compute_cell_occupancy(): calcula las celdas ocupadas y libres en base al mapa de ocupación generado por SLAM. - find_path(): encuentra el camino hacia un punto target_p_ utilizando el algoritmo A*. - get_neighbors(): obtiene las celdas vecinas de una celda. - reconstruct_path(): reconstruye el camino en base a la lista de prioridad creada por el algoritmo A*. - Métodos set y get para los nuevos atributos
Slots	<ul style="list-style-type: none"> - update_readings(): recibe las lecturas del LiDAR y actualiza los valores del atributo readings_. - moved_robot(): actualiza la variable moving_ a falso y almacena las posiciones a las que se ha movido el robot para computar el error al final de la simulación. - finished_navigation(): actualiza la variable navigating_ a false.
Signals	<ul style="list-style-type: none"> - wait_for_readings(): pide y espera a que el entorno de simulación envíe los valores del LiDAR. - move_robot(): envía la orden de mover el robot al entorno de simulación. - path_not_found(): envía información a la ventana principal cuando no se encuentra camino con el algoritmo de conducción autónoma.

Tabla 3.17: Especificación de la clase Simulation (Nuevas aportaciones).

Nombre	Simulation
Descripción	Esta clase implementa todos los datos y funciones necesarias para simular un mapa con lecturas del LiDAR extraídas de un fichero de simulación.
Atributos	<ul style="list-style-type: none"> - <code>running_</code>: estado de simulación: corriendo. - <code>started_</code>: estado de simulación: empezada. - <code>finished_</code>: estado de simulación: finalizada. - <code>loaded_</code>: estado de simulación: cargada. - <code>speed_</code>: velocidad de simulación. - <code>readings_[]</code>: lecturas del LiDAR almacenadas en memoria, leídas de un fichero de simulación. - <code>mat_</code>: imagen que se enviará a la ventana principal para imprimirla. - <code>start_</code>: variable que indica si se ha procesado el primer frame. - <code>width_</code>: ancho del mapa. - <code>height_</code>: alto del mapa. - <code>range_x_</code>: rango de X en el que se estimará la posición. - <code>range_y_</code>: rango de Y en el que se estimará la posición. - <code>range_angle_</code>: rango del ángulo en el que se estimará la posición. - <code>angle_var_</code>: variación del ángulo con la que se harán las distintas estimaciones. - <code>x_var_</code>: variación de X con la que se harán las distintas estimaciones. - <code>y_var_</code>: variación de Y con la que se harán las distintas estimaciones. - <code>w_</code>: peso del valor anterior del pixel para obtener el valor nuevo de ocupación de dicho pixel. - <code>p_</code>: punto inicial del sistema en el mapa. - <code>levmarq_</code>: variable que definirá si se usa el algoritmo Levenberg Marquardt o fuerza bruta. - <code>ignore_human_</code>: variable que definirá si el ángulo en el que estaría el humano debe ser ignorado o no.
Métodos	<ul style="list-style-type: none"> - <code>init()</code>: inicializa los parámetros a unos valores predeterminados y todos sus estados a false. - <code>run()</code>: comienza la simulación y se ejecuta todo el código necesario para realizar SLAM simulado. - <code>clear()</code>: libera memoria y restablece los parámetros predeterminados.
	<ul style="list-style-type: none"> - Métodos get y set: para visualizar y modificar los parámetros respectivamente.
Slots	Sin slots.
Signals	<ul style="list-style-type: none"> - <code>print_img(QImage)</code>: envía una imagen del mapa a la ventana principal para que la imprima en pantalla. - <code>sim_finished()</code>: avisa a la ventana principal que la simulación ha terminado para que esta avise al usuario.

Tabla 3.18: Especificación de la clase Simulation (Proyecto previo).

3.1.4. Clase Sim_env

Sim_env
<ul style="list-style-type: none"> - P_: slam::Position - readings_: array of rplidar_response_measurement_node_hq_t - map_file_: QString - finished_: bool - started_: bool - running_: bool - speed_: double - num_readings_: int - laser_range_: int - x_speed_: float - y_speed_: float - map_: cv::Mat
<ul style="list-style-type: none"> - draw_system() - init() - run() - set_parameters() - clear() - line() - setValues() - Métodos set y get
<ul style="list-style-type: none"> - create_readings() - update_x_speed() - update_y_speed() - move_robot()
<ul style="list-style-type: none"> - emit_readings() - moved()

Tabla 3.19: Definición de la clase Sim_env.

Nombre	Sim_env
Descripción	Se encargará de mantener un entorno de simulación en el que un robot genérico pueda moverse y generar mediciones de un LiDAR.
Atributos	<ul style="list-style-type: none"> - P_: posición del robot en todo momento. - readings_: lecturas del LiDAR que el entorno generará. - num_readings_: variable que almacena el número de lecturas del LiDAR. - laser_range_: variable que almacena el rango máximo del LiDAR. - map_file_: nombre del archivo del que se ha leído el mapa. - finished_: variable que almacena si se ha finalizado la simulación - started_: variable que almacena si ha comenzado la simulación - running_: variable que almacena si la simulación se está ejecutando o parada. - speed_: variable que almacena la velocidad de simulación. - x_speed_: variable que almacena la velocidad en x cuando se está usando el joystick - y_speed_: variable que almacena la velocidad en y cuando se está usando el joystick - map_: variable que almacena el mapa real y las mediciones del LiDAR que se han realizado. Se mostrará en pantalla.
Métodos	<ul style="list-style-type: none"> - draw_system(): dibuja el robot en el mapa. - init(): inicializa la clase y sus atributos. - run(): mantiene el entorno de simulación ejecutando y muestra el mapa por pantalla continuamente. - set_parameters(): ajusta la posición, velocidad y el nombre del mapa del entorno. - clear(): reinicia los atributos a los valores por defecto. - line(): crea una línea con los colores deseados entre dos puntos del mapa. - setValues(): atribuye los colores deseados a un pixel del mapa. - Métodos set y get: métodos para obtener y configurar los atributos.
Slots	<ul style="list-style-type: none"> - create_readings(): crea las lecturas del LiDAR dado un rango del láser, la posición del sistema, un número de lecturas, y un mapa del entorno. - update_x_speed(): actualiza la velocidad en x del robot. - update_y_speed(): actualiza la velocidad en y del robot. - move_robot(): mueve al robot en función de los valores de x e y proporcionados.
Signals	<ul style="list-style-type: none"> - emit_readings(): envía los valores del LiDAR que ha generado. - moved(): envía una señal cuando el robot se ha movido.

Tabla 3.20: Especificación de la clase Sim_env.

3.1.5. Clase JoyPad

JoyPad
<ul style="list-style-type: none">- m_x: float- m_y: float- m_returnAnimation: array of QParallelAnimationGroup- m_xAnimation: array of QPropertyAnimation- m_yAnimation: array of QPropertyAnimation- m_bounds: QRectF- m_knopBounds: QRectF- m_lastPos: QPoint- knopPressed: bool- m_alignment: Qt::Alignment
<ul style="list-style-type: none">- JoyPad()- x()- y()- resizeEvent()- paintEvent()- mousePressEvent()- mouseReleaseEvent()- mouseMoveEvent()
<ul style="list-style-type: none">- setX()- setY()- removeXAnimation()- addXAnimation()- removeYAnimation()- addYAnimation()- setAlignment()
<ul style="list-style-type: none">- xChanged()- yChanged()

Tabla 3.21: Definición de la clase JoyPad.

Nombre	JoyPad
Descripción	Esta clase permite implementar un joypad en una aplicación Qt.
Atributos	<ul style="list-style-type: none"> - m_x: variable que almacena el valor de x en el joypad. - m_y: variable que almacena el valor de y en el joypad. - m_returnAnimation: animación al soltar el joystick. - m_xAnimation: animación al soltar el joystick. - m_yAnimation: animación al soltar el joystick. - m_bounds: Tamaño del joypad - m_knopBounds: Tamaño del joypad - m_lastPos: Valores anteriores de x e y. - knopPressed: variable que almacena si el joystick está pulsado. - m_alignment: ajustar la alineación en caso de que el widget no sea cuadrático.
Métodos	<ul style="list-style-type: none"> - JoyPad(): constructor. - x(): devuelve el valor de x. - y(): devuelve el valor de y. - resizeEvent(): cambiar tamaño del widget. - paintEvent(): dibujar el widget en pantalla. - mousePressEvent(): guardar la posición del joystick. - mouseReleaseEvent(): asignar a x e y, 0. - mouseMoveEvent(): calcular x e y con respecto a la posición del ratón.
Slots	<ul style="list-style-type: none"> - setX(): asignar un valor a x. - setY(): asignar un valor a y. - removeXAnimation(): omitir la animación en x al soltar el ratón. - addXAnimation(): añadir la animación en x al soltar el ratón. - removeYAnimation(): omitir la animación en y al soltar el ratón. - addYAnimation(): añadir la animación en y al soltar el ratón. - setAlignment(): asignar un valor al alineamiento.
Signals	<ul style="list-style-type: none"> - xChanged(): proporciona el valor de x cuando esta cambia. - yChanged(): proporciona el valor de y cuando esta cambia.

Tabla 3.22: Especificación de la clase JoyPad.

3.2. Diseño de los algoritmos (búsqueda de caminos y entorno de simulación)

Ya que la funcionalidad principal del sistema es dotar a un robot de capacidad autónoma en un mapa creado con SLAM, deberemos diseñar unos algoritmos que permitan la búsqueda de caminos y, a su vez, un entorno de simulación que permita modelar el comportamiento de un robot y el sensor LiDAR.

Para ello, se dividirá el apartado en dos subapartados:

- **búsqueda de caminos:** Como se comentó de forma general en los antecedentes, se ha elegido descomponer el mapa en un enrejado de celdas, por lo que este algoritmo comprobará las distintas celdas disponibles y creará un camino que se irá recalculando en cada paso del robot, para evitar así la colisión con obstáculos. El algoritmo elegido es A* dados su bajo coste computacional y su relación estrecha con el enrejado de celdas.
- **Entorno de simulación:** se creará un entorno de simulación que modelará tanto el robot como la obtención de medidas del sensor láser LiDAR y dispondrá de una API para enviar ciertas órdenes al robot, como obtener los datos del sensor o moverlo de la forma deseada.

3.2.1. Búsqueda de caminos

Descomposición del mapa en celdas

Para entender cómo se ha creado este algoritmo de búsqueda de caminos, primero es necesario entender cómo se descompondrá el mapa en celdas.

De forma general, como se comentaba en los antecedentes, el mapa estará dividido en celdas de igual tamaño que podrán aparecer como libres u ocupadas. Esta ocupación viene dada por el mapa de ocupación obtenido mediante SLAM, el cuál se actualiza con cada lectura de datos del LiDAR y asigna a cada pixel del mapa un valor entre 0 y 255, siendo este valor una medida de ocupación. Este valor de ocupación se moverá en unos rangos que considerarán un pixel libre, ocupado o parcialmente ocupado. Este último valor se utiliza para los pixeles que no tienen un valor de ocupación demasiado alto, pero pueden estar cerca de otros obstáculos, o que en algún momento se consideró como ocupado[4].

Las celdas comprenderán un número determinado de pixeles y se contará como ocupada si existe un porcentaje de pixeles ocupado. Para asegurar la correcta evasión de obstáculos, se ha utilizado un enfoque conservador y se considerará una celda como ocupada incluso cuando los pixeles sean parcialmente ocupados. El porcentaje se podrá modificar mediante los parámetros de la aplicación, y en los capítulos de pruebas y conclusiones se comprobará cuál funciona mejor para nuestro caso y se usará como predeterminado.

Para ilustrar el enrejado de celdas, en la figura 3.2 aparece un mapa genérico, y en la figura 3.3 se muestra descompuesto en el enrejado de celdas:

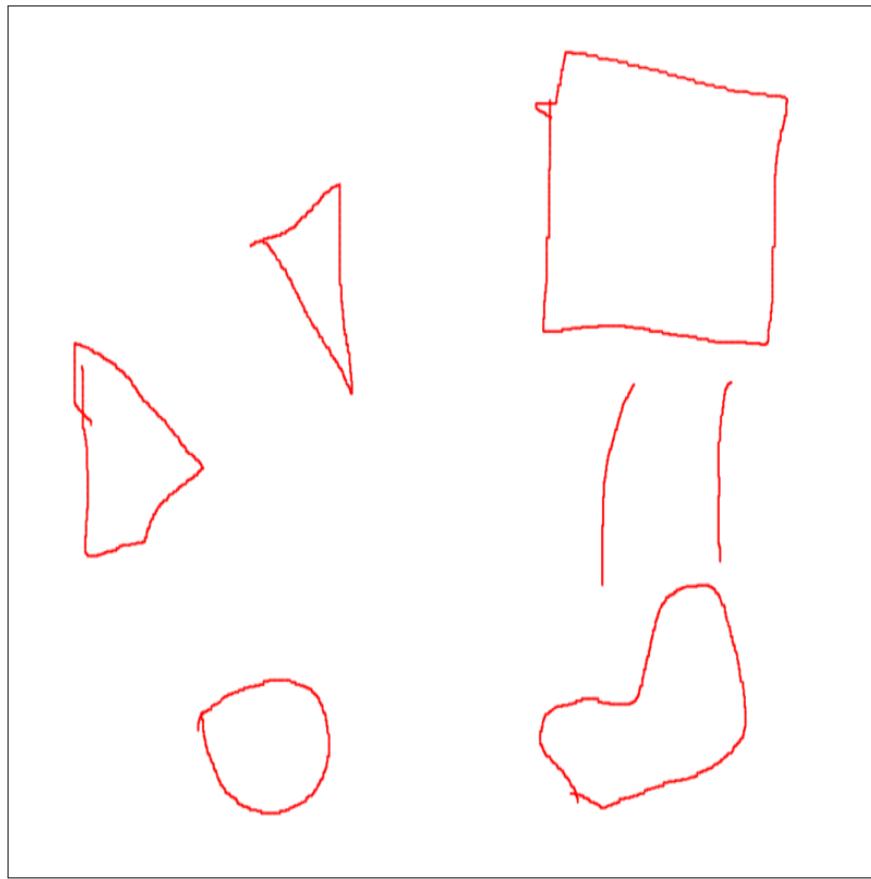


Figura 3.2: Mapa genérico.

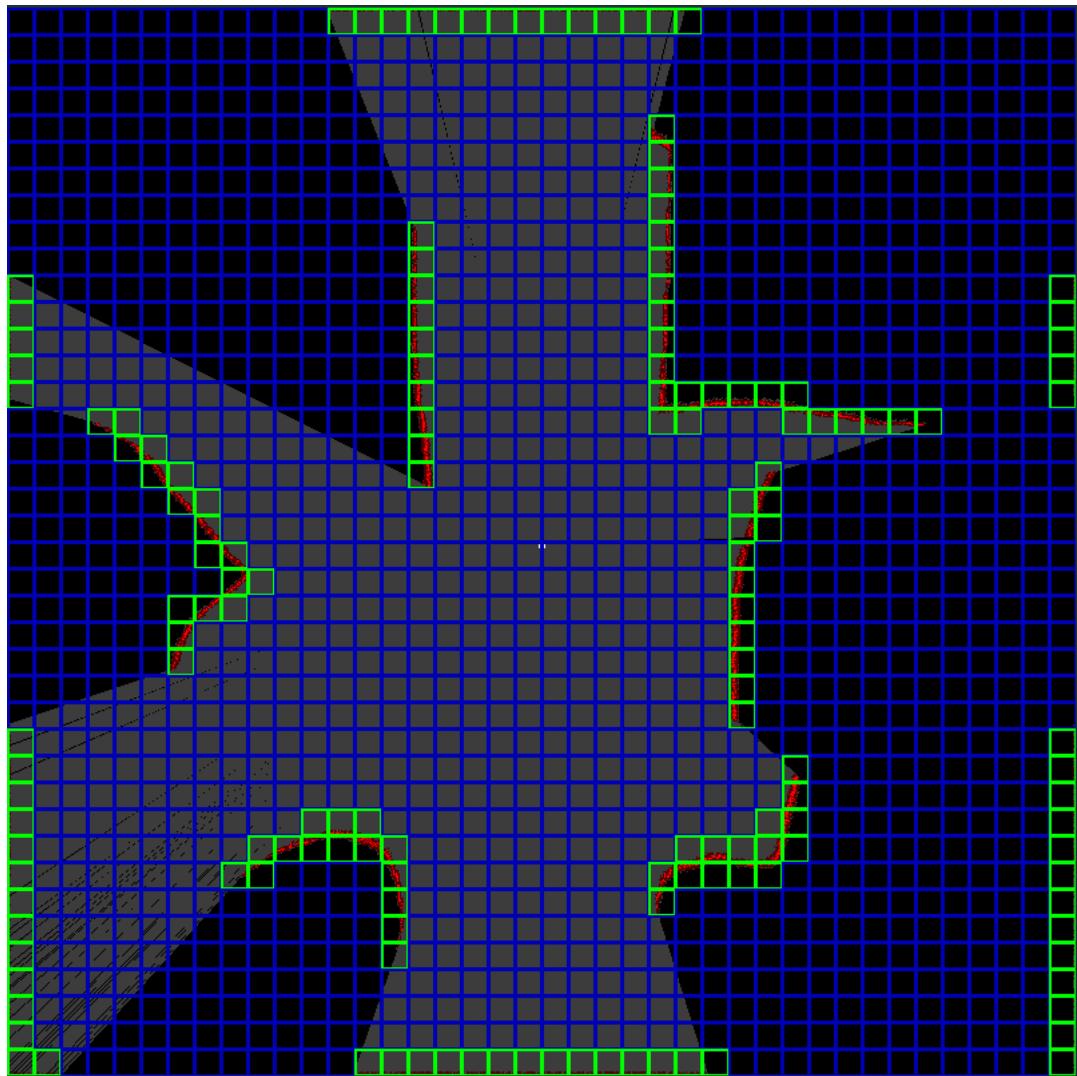


Figura 3.3: Enrejado de celdas para un mapa genérico.

En azul se pueden distinguir las celdas libres y en verde las ocupadas. Dentro de las celdas verdes (ocupadas) se pueden ver los obstáculos del mapa creado por SLAM en color rojo y dentro de las celdas azules (libres) se puede distinguir el espacio libre en color gris. Los bordes del mapa se consideran como ocupados ya que, al ser un mapa de interior, el robot no podrá salir de ahí. En este caso, los bordes del mapa que el robot aún no ha descubierto mediante los rayos del LiDAR, se consideran como desconocidos.

Las celdas con el fondo negro son desconocidas, pero se tomarán como libres hasta que el robot las descubra. De esta forma, se puede navegar hacia una celda desconocida y conforme el robot se vaya acercando se pueden dar dos casos: a) hay un obstáculo obstruyendo el camino: el algoritmo recalculará el camino b) la celda está rodeada de obstáculos: el algoritmo determinará que no hay camino posible y el robot se detendrá.

Algoritmo A*

El algoritmo A* es un algoritmo de búsqueda en grafos de tipo heurístico. Se basa en dos informaciones, heurística y real, para determinar el camino de menor coste desde un nodo x a un nodo y. En general, el algoritmo trata de optimizar la función $f(n) = g(n) + h(n)$, donde g es el coste del camino recorrido desde el nodo x hasta un nodo intermedio, y h es el valor heurístico desde ese nodo intermedio a un nodo final y. En el nodo inicial, el valor de la función será puramente heurístico, y en el nodo final, el valor será puramente el coste real.

En nuestro caso, aplicaremos este algoritmo a un enrejado de celdas, lo que significa que las celdas serán los nodos y el coste de moverse de una celda a otra será de 1. Utilizaremos dos heurísticas que compararemos en los capítulos 6 y 7, las cuales son distancia manhattan y distancia euclidiana.

La primera representa la distancia en celdas que recorrería el robot para llegar de un punto origen (o) a otro destino (d), y se obtiene mediante la fórmula: $valor = abs(o.x - d.x) + abs(o.y - d.y)$.

La segunda calcula la distancia en línea recta desde un punto origen (o) a un punto destino (d) y se calcula mediante la fórmula: $valor = \sqrt{(o.x - d.x)^2 + (o.y - d.y)^2}$.

Este algoritmo hace uso de algunas funciones simples como calcular los vecinos o ver si puede moverse a una celda concreta, pero se han evitado por simplicidad. Aún así, estas funciones se incluirán en el código fuente por lo que se puede consultar fácilmente. El algoritmo base se puede encontrar en [\[Astar\]](#), aunque se le han realizado unas modificaciones considerables.

El algoritmo A* en cuestión, definido en pseudocódigo, quedaría tal que:

Algorithm 1 find_path(target_x, target_y, robot_x, robot_y, width, height, safety_distance)

```

curr_dist ← 0
curr_node.x ← system_x
curr_node.y ← system_y
curr_node.heap ← “open”
curr_node.distance_walked ← curr_dist
curr_node.heuristic ← heuristic(curr_node.x, curr_node.y, target_x, target_y)
curr_node.total_dist ← curr_node.distance_walked + curr_node.heuristic
hold_nodes ← array of nodes
hold_nodes.add(curr_node)
node_track ← 2D array of nodes of size width x height
node_track[curr_node.x][curr_node.y] ← curr_node
neighbors ← array of nodes of size 4
while path_not_found and hold_nodes is not empty do
    aux_dist ← (width/(safety_distance)) + height/(safety_distance)))
    aux_it ← hold_nodes.begin()
    aux_node ← NULL
    for i from 0 to hold_nodes.length() with step 1 do
        Compute best node and set as curr_node
    end for
    Delete the picked node from the hold_nodes list
    Get neighbors
    for Each neighbor do
        if is target point then
            reconstruct_path()
        end if
        neighbor.distance_walked ← curr_dist + 1
        neighbor.heuristic ← heuristic(neighbor.x, neighbor.y, target_x, target_y)
        neighbor.total_dist ← curr_node.distance_walked + curr_node.heuristic
        Add the node to the track list and to the open list of nodes only if there is no
        other node with the same position with a shorter path already.
    end for
    Close node (add it to closed list of nodes)
end while
Return path not found

```

3.2.2. Entorno de simulación

Como ya se ha explicado, el entorno de simulación se puede dividir en dos partes: la simulación de un robot y el modelado del LiDAR.

Simulación simple de un robot

Para la simulación simple de un robot, se implementará de tal forma que este dispondrá de una posición y en un mapa, y el propio mapa proporcionado por el usuario. Esto servirá para definir si el robot podrá moverse hacia cierto punto y también para obtener las mediciones del LiDAR, que se explicarán en el siguiente punto. El robot podrá ser movido en cualquier momento, y las mediciones podrán ser obtenidas también en cualquier momento, mientras el entorno de simulación se encuentre activo. Se ha supuesto un robot con capacidad de movimiento omnidireccional (el robot no necesita girar para moverse a los lados, ya que las ruedas permitirán este tipo de movimiento)

Se proporciona un método para mover el robot con un incremento (o decremento) en sus componentes x e y. Este incremento solo podrá efectuarse si se encuentra dentro de los límites del mapa y si no hay ningún obstáculo impidiendo el movimiento. Esto, en un robot real, se traduciría en un cálculo trigonométrico para determinar el movimiento de las ruedas, pero se ha evitado este paso, utilizando un robot genérico.

Modelado del LiDAR

Para modelar el LiDAR, se ha estudiado de cerca el funcionamiento de un LiDAR real (especificado en el capítulo 4, sección 4.3.3). Mediante este estudio, se ha podido determinar que el rango de mediciones en cada iteración ronda las 530-630 mediciones. Para modelar este comportamiento, se ha creado un valor aleatorio que determinará cuántas mediciones se crearán en ese rango.

El rango máximo del láser, según la documentación del LiDAR, sería de 12m, pero el rango efectivo que se ha llegado a alcanzar en la realidad es de unos 8m. Por lo tanto, el rango que se usará para cada rayo láser será de 8m como máximo.

Con estos dos comportamiento, el LiDAR obtendría de 530 a 630 medidas de unos 8m como máximo. Cada medida tendrá asignado un ángulo, una distancia, una calidad y una bandera, como se especifica también en el capítulo 4, sección 4.3.3. La calidad y la bandera se mantendrán estáticas, mientras que el ángulo y la distancia variará con cada rayo.

Para determinar la distancia, primero se lanzará un rayo con un ángulo determinado y una distancia de 8m. Para hacer esto, obtenemos las coordenadas x e y del punto final del rayo mediante trigonometría, y utilizamos un iterador de openCV para recorrer una línea en el mapa (este iterador utiliza el algoritmo de bresenham). Si se encuentra con un obstáculo en el mapa mientras recorre la línea, utiliza las coordenadas x e y del

obstáculo para calcular de forma reversa la distancia a la que se encuentra del robot, y se introducirá cierto valor de ruido (positivo o negativo) para obtener unas medidas más cercanas a un entorno real, donde el láser siempre sufre cierto ruido. Si no se encuentra con ningún obstáculo, la distancia sería de 8m+-ruido. Estas medidas se ponen finalmente a disposición del usuario para utilizarlas, como si se hubiesen obtenido de un láser real, con el mismo tipo de dato.

El algoritmo de creación de las medidas, en pseudocódigo, sería:

Algorithm 2 calculate_readings(laser_range, sys_x, sys_y, sys_angle)

```

num_readings ← random number between 530 and 630
readings ← Array of lidar readings type of size num_readings
for i from 0 to num_readings with step 1 do
    readings[i].angle = (360.0/num_readings) * i * (1_lshift_14)/90.f
    readings[i].quality = 255
    readings[i].flag = 2
    dist ← laser_range
    max_x_ray ← (cos(((360.0/num_readings) * i + sys_angle) * PI/180.0) *
laser_range) + sys_x
    max_y_ray ← (sin(((360.0/num_readings) * i + sys_angle) * PI/180.0) *
laser_range) + sys_y

```

Iterate through a line of the map with bresenham algorithm until hitting an object, calculate distance to abject as $dist = \sqrt{(sys_x - obj_x)^2 + (sys_y - obj_y)^2}$ and add/substract a small noise.

```
    readings[i].dist = dist
```

end for

Publish readings

3.3. Diseño de la interfaz

3.3.1. Modificaciones a la ventana principal

La ventana principal sufrió unas modificaciones mínimas. El botón empezar/pausar simulación ahora ocupa la mitad de espacio, y se ha implementado un botón finalizar simulación. Esto es debido a que la simulación ahora puede no estar ligada a un fichero de simulación, sino a una simulación en tiempo real que necesita ser finalizada.

Se ha implementado también un label encima de estos dos botones para mostrar qué tipo de simulación se va a ejecutar (leída de un fichero o en tiempo real aportando un mapa).

El botón empezar/pausar simulación ahora tiene una funcionalidad adicional. Cuando el modo de simulación es "live", el programa mostrará una ventana de elección de archivo para cargar un mapa real y comenzará una simulación en tiempo real.

Por último, se ha implementado también un joystick para controlar al robot y un cartel de información sobre cómo controlarlo.

Todos estos cambios se pueden apreciar en la figura 3.4:

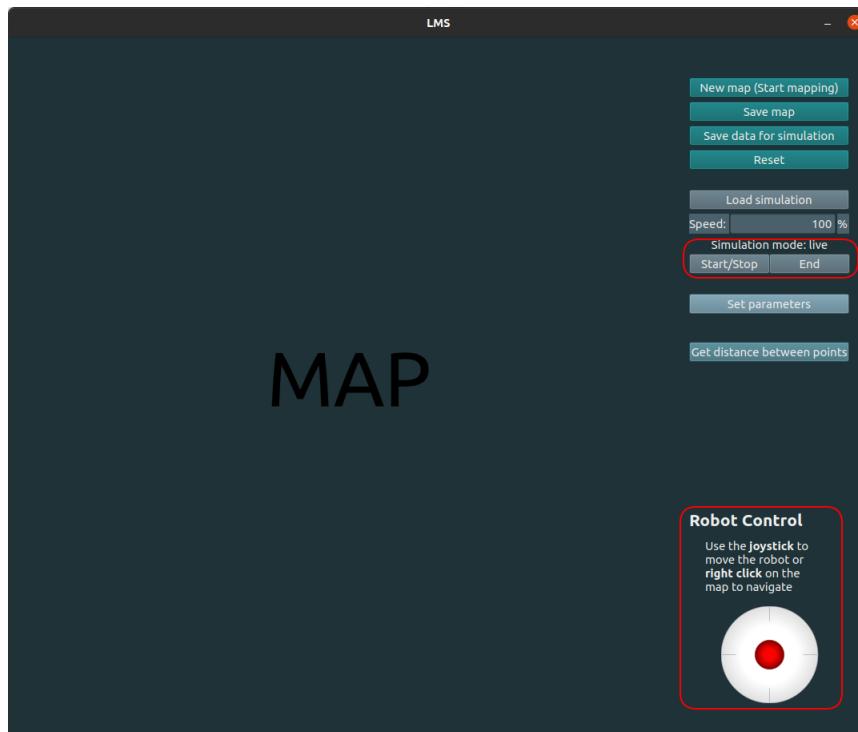


Figura 3.4: Modificaciones a la ventana principal.

3.3.2. Modificaciones a la ventana parámetros

A esta ventana se le han añadido 3 campos para el algoritmo de búsqueda de caminos. Estos campos son: distancia de seguridad (para el enrejado de celdas), porcentaje de ocupación para que una celda se considere ocupada y heurística a utilizar en el algoritmo A*. Se pueden apreciar dichos cambios en la figura 3.5:

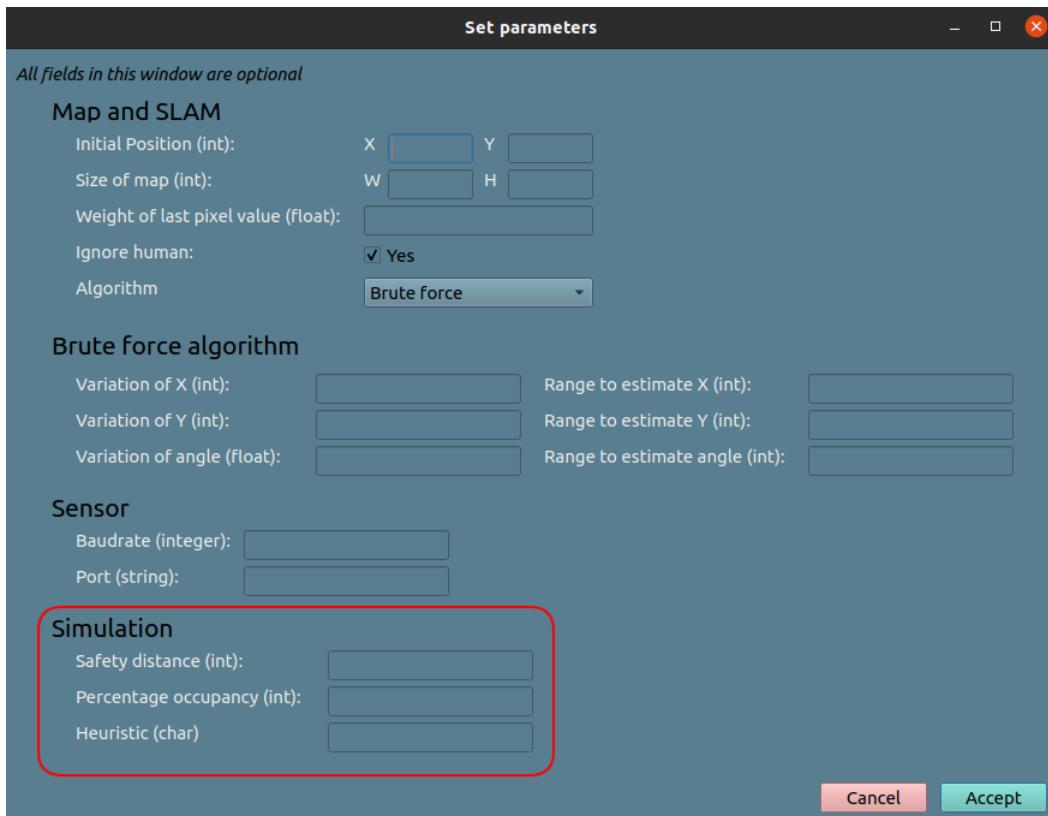


Figura 3.5: Modificaciones a la ventana parámetros.

3.3.3. Ventana ‘Real map’

Esta ventana es una ventana muy sencilla, únicamente mostrará el mapa proporcionado por el usuario, junto con el robot y las mediciones del LiDAR que este obtiene en tiempo real. Se puede apreciar en la figura 3.6:

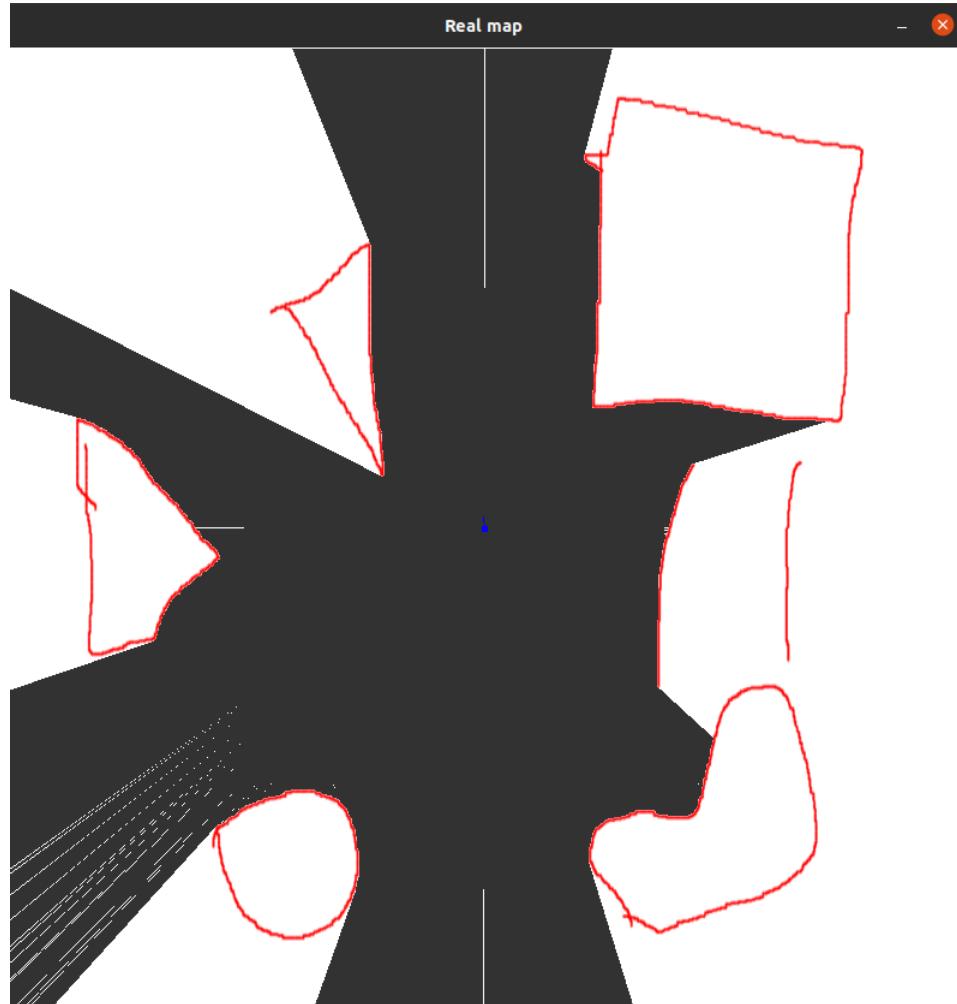


Figura 3.6: Ventana ‘Real map’.

Capítulo 4

Pruebas

En este capítulo se procederá a realizar una serie de pruebas para obtener distintas métricas de los algoritmos que se han diseñado para poder comparar los distintos posibles parámetros que se pueden aplicar a los algoritmos y también comprobar el correcto funcionamiento de estos.

4.1. Pruebas de los algoritmos

Para las pruebas, se han diseñado 3 mapas, el primero de ellos es un mapa simple con algunos objetos dispuestos de forma psuedoaleatoria, y se puede apreciar en la figura 4.1.

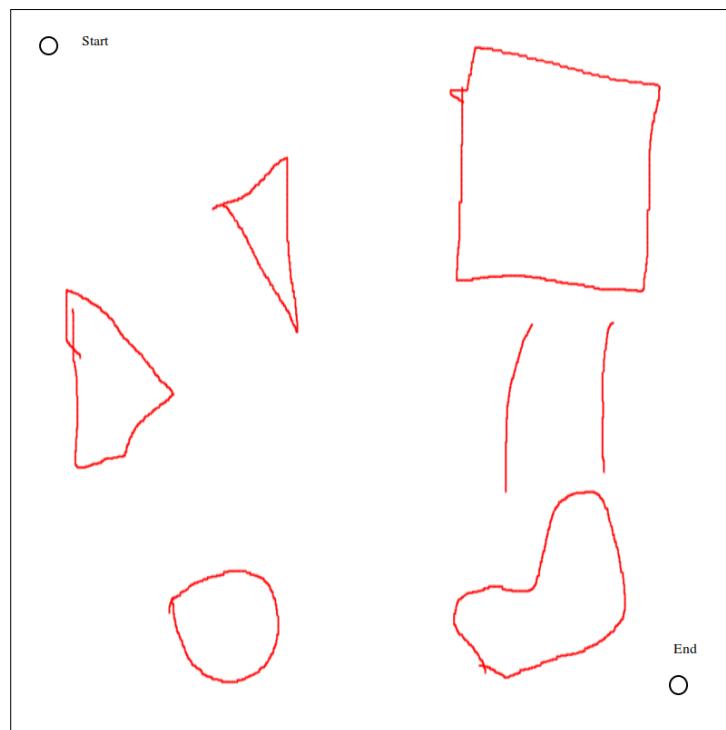


Figura 4.1: Primer mapa diseñado (mapa 1, genérico).

El segundo tipo de mapa que se diseñó es un laberinto relativamente simple con una salida y varias paredes obstaculizando el camino recto. Se puede apreciar en la figura 4.2. Se dispone de un vídeo para ilustrar el funcionamiento del sistema en este mapa: <https://youtu.be/XwuEWhTFIaQ>.

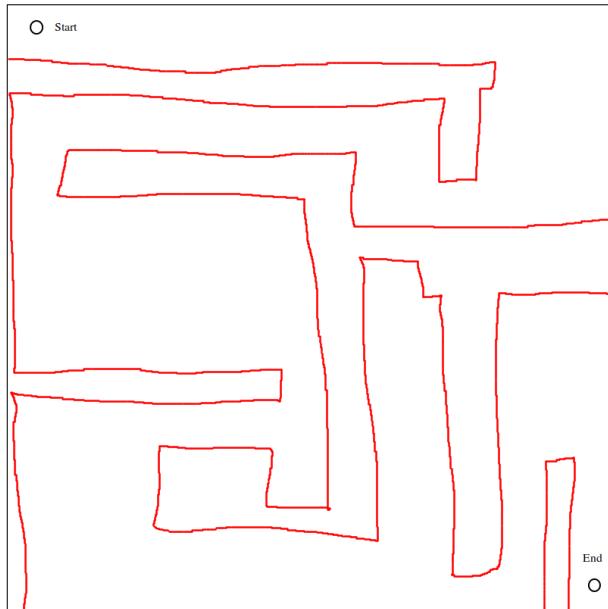


Figura 4.2: Segundo mapa diseñado (mapa 2, laberinto).

El último mapa que se diseñó es un mapa con distintos y abundantes tipos de obstáculos para complicar las decisiones tomadas por el algoritmo. Se puede apreciar en la figura 4.3.

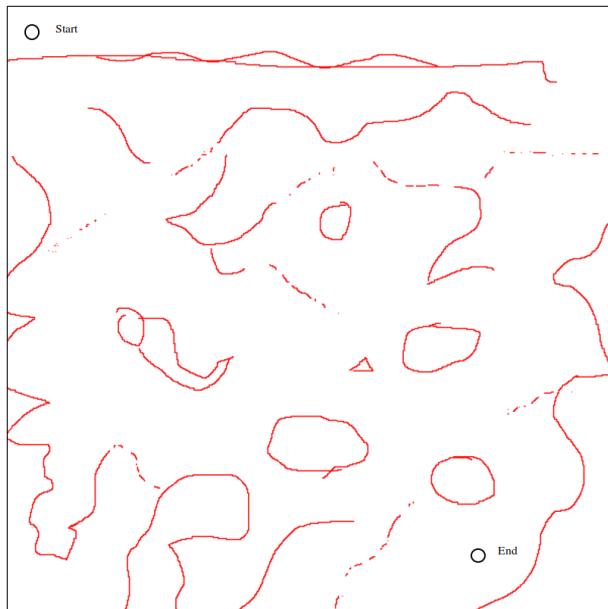


Figura 4.3: Tercer mapa diseñado (mapa 3, alta complejidad).

Utilizando estos mapas, se han llevado a cabo 2 tipos de experimentos:

- **Experimentación con SLAM:** ahora que se dispone de un entorno de simulación, se pueden realizar pruebas con el algoritmo de SLAM y obtener un valor de error para la estimación de la posición real del sistema, experimento que no se pudo realizar en el proyecto anterior dada la falta de información sobre el sistema real. Para la experimentación, se variarán los valores de los parámetros de SLAM y se calculará el error y el tiempo de ejecución. La representación de los parámetros será “variación de x / variación de y / variación del ángulo / rango para estimar x / rango para estimar y / rango para estimar el ángulo”.

Estos parámetros hacen referencia a las distintas combinaciones de posición y de rango en el que el sistema podría estar situado después de realizar una medición del LiDAR.

Si se incrementa la variación de las componentes de la posición, la estimación de la posición será más imprecisa, ya que se traduce en una comprobación de menos posiciones, pero también en un decremento de la carga computacional al disminuir la cantidad de comprobaciones.

Si se incrementa el rango de estimación, se incrementará la precisión de la estimación de la posición, ya que se comprobará un rango más amplio en el mapa en el que puede estar el sistema, pero se incrementará la carga computacional al aumentar el número de comprobaciones.

En la tabla 4.1 se muestran los valores métricos obtenidos para los parámetros mencionados:

Parameters	Algorithm run time (ms)	Mean relative error
1 / 1 / 2 / 40 / 40 / 100	~3000	0.10%
1 / 1 / 2 / 30 / 30 / 80	~1400	0.20%
1 / 1 / 2 / 20 / 20 / 60	~450	0.20%
2 / 2 / 3 / 40 / 40 / 100	~600	0.30%
2 / 2 / 3 / 30 / 30 / 80	~230	0.38%
2 / 2 / 3 / 20 / 20 / 60	~80	0.18%
2 / 2 / 4 / 40 / 40 / 100	~400	0.32%
2 / 2 / 4 / 30 / 30 / 80	~180	0.41%
2 / 2 / 4 / 20 / 20 / 60	~60	0.21%
3 / 3 / 4 / 40 / 40 / 100	~200	0.45%
3 / 3 / 4 / 30 / 30 / 80	~70	0.24%
3 / 3 / 4 / 20 / 20 / 60	~20	0.50%

Tabla 4.1: Resultados métricos para el algoritmo de SLAM fuerza bruta.

- **Experimentación con A***: para el algoritmo de A*, se probarán los distintos mapas (1, 2 o 3) con las dos heurísticas que se han implementado ('m' para manhattan y 'e' para euclidiana), y se podrá ver el camino que ambas heurísticas obtienen, el tiempo de ejecución, el coste del camino elegido, el coste del camino óptimo y el número de colisiones del robot a lo largo del camino.

En la tabla 4.2 se muestran los valores métricos obtenidos, y en las figuras 4.4 a 4.9, se muestran los caminos obtenidos por el algoritmo A* para los distintos mapas, y el mapa que se ha ido generando. En las figuras, el camino elegido aparecerá en color gris, los obstáculos en rojo, las celdas ocupadas en verde, las celdas libres en azul y el espacio desconocido en negro.

Parameters	Algorithm run time (ms)	Optimal path size (cells)	Path size (cells)	Path size (cm)	Collisions
1 / m	0.16	72	72	1441	0
1 / e	1.62	72	72	1441	0
2 / m	0.17	185	191	3821	0
2 / e	1.87	185	187	3741	0
3 / m	0.04	92	96	1921	0
3 / e	0.72	92	97	1941	0

Tabla 4.2: Resultados métricos para el algoritmo A*

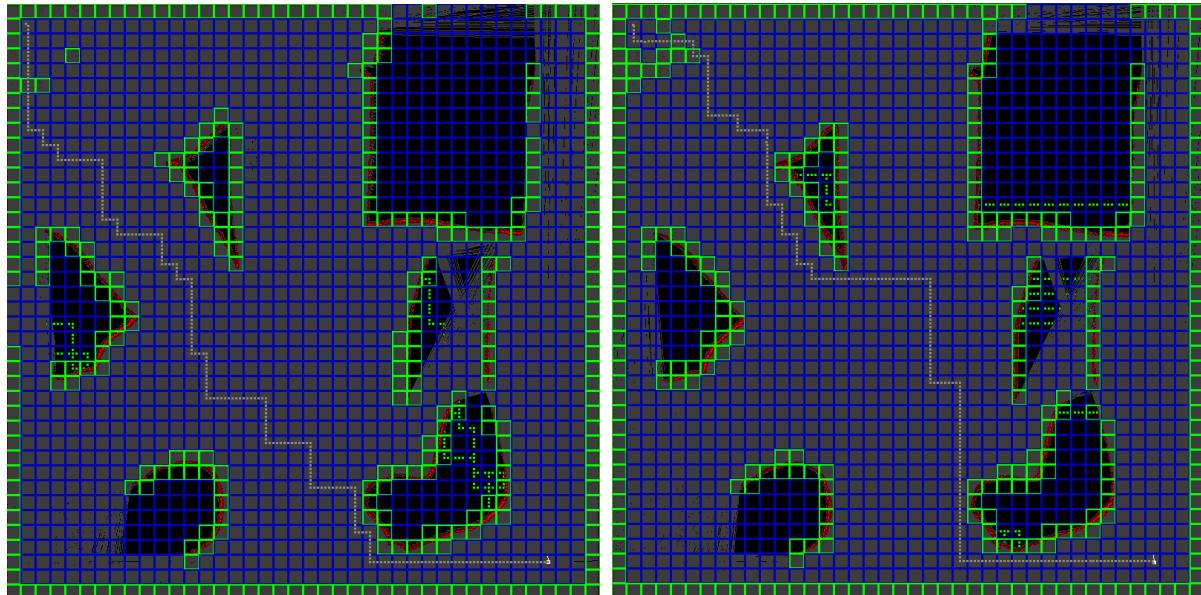


Figura 4.4: Camino obtenido para el mapa 1
Figura 4.5: Camino obtenido para el mapa 1
(genérico) mediante la heurística de distancia
(genérico) mediante la heurística de distancia
euclíadiana (parámetros 1/e). euclíadiana (parámetros 1/m).

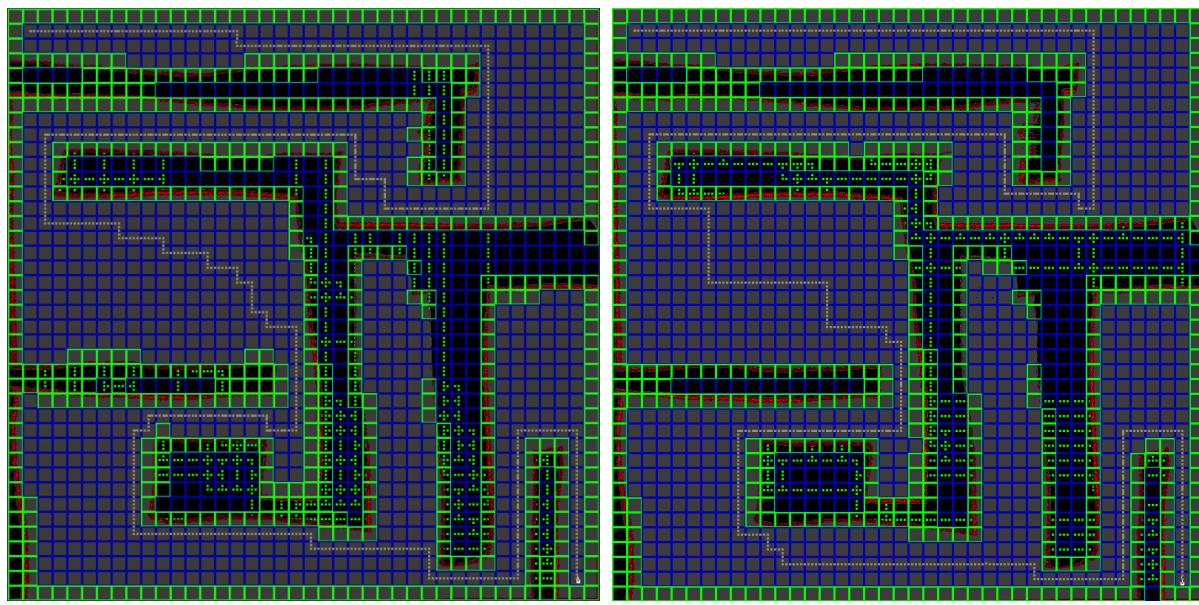


Figura 4.6: Camino obtenido para el mapa 2
 Figura 4.7: Camino obtenido para el mapa 2
 (laberinto) mediante la heurística de distan-
 (laberinto) mediante la heurística de distan-
 cia euclidiana (parámetros 2/e).

cia manhattan (parámetros 2/m).

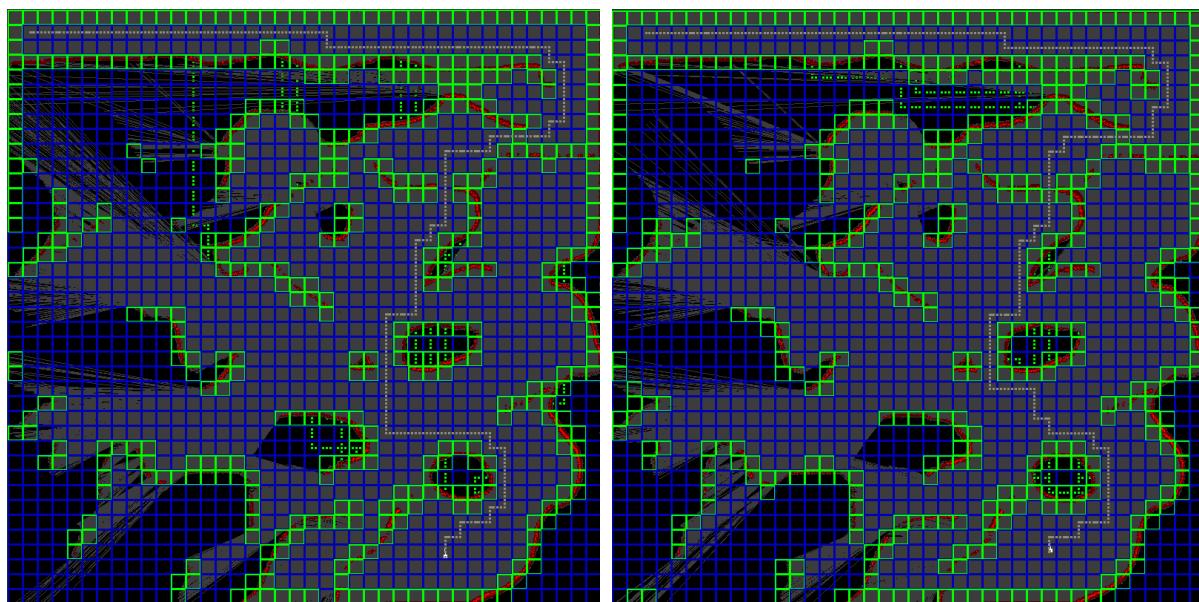


Figura 4.8: Camino obtenido para el mapa 3
 Figura 4.9: Camino obtenido para el mapa 3
 (alta complejidad) mediante la heurística de
 (alta complejidad) mediante la heurística de
 distancia euclidiana (parámetros 3/e). distancia manhattan (parámetros 3/m).

4.2. Discusión de resultados

4.2.1. SLAM

Con respecto al algoritmo de SLAM, se puede ver que, de forma general, el error obtenido es bastante bajo, ya que supone menos de 1 % en la estimación de la posición. Sin embargo, ya que es el error medio, cuando este se acerca al 0.40 %, se puede ver algún cambio brusco en la posición del sistema y, por tanto, algún fallo en el cálculo de los objetos.

Con un error de en torno al 0.2 %, la posición del sistema es prácticamente indiscernible de la posición real, sin cambios bruscos ni errores aparentes en la detección de los obstáculos.

Si bien es cierto que con la primera combinación se obtiene un error del 0.1 %, no es nada realista aplicar esta configuración, ya que el tiempo de cómputo se torna en los 3000ms (3s por cada frame o actualización de valores), lo que lo hace inviable en un entorno real.

Como mejores resultados, tenemos los parámetros que actualmente son los predeterminados 2/2/3/20/20/60. Estos parámetros permiten actualizar el mapa 12 veces por segundo, lo que resulta en una actualización muy fluida y permite que el sistema haya realizado pequeños cambios en la posición, mejorando así la estimación en sí misma. Otro resultado interesante es 2/2/4/20/20/60, aunque la precisión disminuye ligeramente y el rango del ángulo sería también menor. Por esta razón, podemos considerar los anteriores como los mejores parámetros.

4.2.2. A-star

Con respecto al algoritmo A*, se puede ver que los valores de los caminos son prácticamente idénticos excepto en el mapa del laberinto, en el que la heurística de manhattan obtiene un camino algo peor. Los tiempos de cómputo de la heurística manhattan son, relativamente, mucho menores, ya que no necesita realizar la potencia ni la raíz cuadrada pero, de forma realista, el algoritmo completo toma un tiempo que ronda el milisegundo, con lo cuál no tiene realmente sentido comparar sus tiempos de ejecución en función de la heurística escogida.

Fijándonos ahora en los mapas, se puede ver que la heurística euclíadiana, sobre todo en el primer mapa, prefiere escoger caminos diagonales, ya que en esto se basa el cálculo de la distancia euclíadiana, por lo que es un resultado esperado. La heurística de manhattan por su parte toma caminos más rectos.

Se puede observar que, de forma general, el algoritmo funciona bastante bien, eligiendo un camino esperado con una distancia muy aceptable. Un pequeño problema que aparece en el mapa de alta complejidad es que, cuando el sistema se aleja de un obstáculo, el ruido en las lecturas puede provocar que a veces, una casilla que se había marcado como libre, aparezca como ocupada (cuando tiene obstáculos muy cerca), y por tanto el robot haya tomado un camino que ahora aparece como ocupado.

Se podría decir que la heurística euclíadiana podría ser la preferida, ya que obtiene un camino más similar a lo que podríamos imaginar si pensásemos en una solución de forma lógica, y además para caminos largos, parece obtener unos resultados algo mejores.

Capítulo 5

Conclusiones

En este capítulo se detallarán las conclusiones a las que se ha llegado, los resultados obtenidos con el trabajo final y se relacionarán los objetivos que definimos al principio con los que hemos conseguido. También se comentarán las pruebas y se hablará de las mejoras que se podrían implementar en un futuro, si se continuase con este trabajo.

5.1. Conclusiones sobre los objetivos

Una vez finalizado el trabajo, se puede afirmar que se han alcanzado la mayoría de los objetivos propuestos, a excepción de uno de ellos. El único objetivo que no se ha alcanzado, ha sido la comprobación de distintas tecnologías de comunicación, ya que no se disponía de un robot real con el que probarlas.

Pero, con respecto a los demás objetivos, se dispone de un entorno de simulación que permite obtener mediciones de un LiDAR con su mismo tipo de datos y distribución desde un mapa proporcionado por el usuario, se puede manejar al robot en dicho entorno con un control manual o automático. El usuario tiene libertad para implementar sus propios algoritmos utilizando la API del entorno de simulación, que a su vez permite una modificación de los parámetros mediante la aplicación gráfica principal. Por último, se puede extraer distintas métricas de error y tiempos de ejecución del programa mediante impresión en consola.

La aplicación gráfica principal funciona correctamente con el entorno de simulación y permite al usuario un uso sencillo e intuitivo, mostrando el robot, el camino escogido, suficiente información autocontenido en la interfaz, y el mapa real por el que circula el robot.

5.2. Conclusiones sobre las pruebas

Como se comentaba en el capítulo 4, donde se presentaron las pruebas realizadas, el algoritmo de SLAM con los parámetros predeterminados es capaz de realizar unas 12 actualizaciones del mapa por segundo, por lo que es un número bastante bueno para no perder la posición del sistema, minimizando los cambios bruscos de posición de este. Con estos parámetros, se obtiene un 0.2 % de error medio cuadrático, que se traduce en una simulación fluida y en la que los pequeños errores en la estimación de la posición son escasamente perceptibles.

Con respecto a la búsqueda de caminos, el algoritmo A* funciona a una alta velocidad, siendo así posible implementarlo en cualquier tipo de sistema. Esto es debido a que en un mapa con un número relativamente pequeño de casillas, y una heurística similar a la distancia real, el número de comprobaciones es muy limitado. Aún en el peor caso, cuando no hay camino y se necesita comprobar más nodos, el tiempo de ejecución es muy reducido. En definitiva, el tiempo de ejecución del algoritmo A* es irrelevante, siendo en todos los casos menor a 2ms.

Los resultados obtenidos son más que satisfactorios, el sistema ha conseguido evitar todos los obstáculos en el camino (0 colisiones) recalculando su ruta constantemente y llegar desde un punto origen a un punto destino. Según la tabla 4.2, podemos ver que, además, los caminos son muy cercanos al camino óptimo. Teniendo en cuenta el peor de los casos (usando la peor heurística), obtenemos un 0 % de camino adicional para el caso del primer mapa, un 3.24 % para el segundo mapa (1.08 % en el mejor caso) y un 5.43 % para el tercero (4.35 % en el mejor caso).

Estos pequeños errores son también, a veces, debidos a cierto ruido en las medidas del LiDAR y a la aproximación conservadora que se ha seguido. En ciertos casos el sistema detectaría un obstáculo debido a este ruido y lo evitaría, pero con el coste de moverse a otra casilla y empeorar el camino a la casilla final.

5.3. Futuras mejoras

Como principal y más clara futura mejora, sería implementar este sistema en un robot real. Las bases están ya implementadas, y las mediciones que usa la simulación son exactamente las mismas que las de un LiDAR, por lo que el único problema sería elegir/diseñar el robot real, y conectarlo con la aplicación.

Otras posibles mejoras serían con respecto al entorno de simulación, ya que es un entorno limitado al alcance de este trabajo. Entre estas mejoras están:

- Añadir un entorno para crear mapas directamente desde la aplicación y simularlos a continuación.
- Guardar datos de la simulación de un entorno mientras se ejecuta, como se guardan cuando se usa un LiDAR real.
- Aportar más información sobre el enrejado de celdas o las decisiones tomadas por el algoritmo A*, a modo educativo/ilustrativo.
- Mejorar la simulación del robot, incluyendo otro tipo de robots como robots de 2 ruedas.

Capítulo 6

Agradecimientos

Me gustaría agradecer al director de este proyecto por su colaboración y consejos, y a todas las personas que me han apoyado y ayudado en esta compleja etapa de estudios y trabajo durante una pandemia.

Bibliografía

- [1] Nico Lang y Jan Wegner. “Country-wide high-resolution vegetation height mapping with Sentinel-2”. En: *Remote Sensing of Environment* 233 (nov. de 2019), pág. 111347. DOI: 10.1016/j.rse.2019.111347.
- [2] D. Maturana y S. Scherer. “3D Convolutional Neural Networks for landing zone detection from LiDAR”. En: *2015 IEEE International Conference on Robotics and Automation (ICRA)*. 2015, págs. 3471-3478. DOI: 10.1109/ICRA.2015.7139679.
- [3] Chen Xiong, Qiangsheng Li y Xinzheng Lu. “Automated regional seismic damage assessment of buildings using an unmanned aerial vehicle and a convolutional neural network”. En: *Automation in Construction* 109 (nov. de 2019). DOI: 10.1016/j.autcon.2019.102994.
- [4] Manuel Rafael Navarro Fuentes. “Laser mapping software”. En: *Catalogo Mezquita UCO* (jun. de 2020).
- [5] aerial insight. *Lidar vs fotogrametría: ¿qué tecnología es mejor?* <https://www.aerial-insights.co/blog/lidar-vs-fotogrametria/>. Accessed on 2020-01-31.
- [6] *Web oficial de Qt.* <https://www.qt.io/download>.
- [7] *Web oficial del LiDAR RPLIDAR A1M8.* <https://www.slamtec.com/en/Lidar/A11>.
- [8] Peter E. Hart, Nils J. Nilsson y Bertram Raphael. “A Formal Basis for the Heuristic Determination of Minimum Cost Paths”. En: *IEEE Transactions on Systems Science and Cybernetics* 4.2 (1968), págs. 100-107. DOI: 10.1109/TSSC.1968.300136.
- [9] L.E. Kavraki, M.N. Kolountzakis y J.-C. Latombe. “Analysis of probabilistic roadmaps for path planning”. En: *IEEE Transactions on Robotics and Automation* 14.1 (1998), págs. 166-171. DOI: 10.1109/70.660866.
- [10] MathWorks. *Design, simulate, and deploy path planning algorithms.* <https://www.mathworks.com/discovery/path-planning.html>. Accessed on 2022-08-07.
- [11] daywalkr. *maze-generator.* <https://github.com/topics/maze-generator?l=java&o=desc&s=updated>. Accessed on 2022-08-07.
- [12] Roland Siegwart, Margarita Chli, Juan Nieto y Nick Lawrance. *Autonomous Mobile Robots.* https://ethz.ch/content/dam/ethz/special-interest/mavt/robotics-n-intelligent-systems/asl-dam/documents/lectures/autonomous_mobile_robots/spring-2018/Planning_II.pdf. Accessed on 2022-08-07.
- [13] Anónimo. *Tema 4 planificación.* Temario de la asignatura robots autónomos, curso 2020-2021. Accessed on 2022-08-07.

- [14] Jorge Fuentes-Pacheco, Jose Ascencio y J. Rendon-Mancha. “Visual Simultaneous Localization and Mapping: A Survey”. En: *Artificial Intelligence Review* 43 (nov. de 2015). DOI: 10.1007/s10462-012-9365-8.
- [15] IRS. *UWsim oficial webpage*. <http://www.irs.uji.es/uwsim/>. Accessed on 2022-09-05.
- [16] Game Physics Simulation. *Bullet oficial webpage*. <https://pybullet.org/wordpress/index.php/forum-2/>. Accessed on 2022-09-05.
- [17] Laboratoire d’Analyse et d’Architecture des Systèmes ‘ at the University of Toulouse. *MORSE oficial webpage*. <https://morse-simulator.github.io/>. Accessed on 2022-09-05.
- [18] Fundación Blender. *Blender oficial webpage*. <https://www.blender.org/>. Accessed on 2022-09-05.
- [19] Fundación Blender. *Gazebo oficial webpage*. <https://gazebosim.org/home>. Accessed on 2022-09-05.
- [20] Epic Games. *Unreal engine oficial webpage*. <https://www.unrealengine.com/en-US/>. Accessed on 2022-09-05.
- [21] Unity Technologies. *Unity oficial webpage*. <https://unity.com/es>. Accessed on 2022-09-05.
- [22] Jeff Craighead, Robin Murphy, Jenny Burke y Brian Goldiez. “A Survey of Commercial Open Source Unmanned Vehicle Simulators”. En: *Proceedings 2007 IEEE International Conference on Robotics and Automation*. 2007, págs. 852-857. DOI: 10.1109/ROBOT.2007.363092.
- [23] Robin Amsters y Peter Slaets. “Turtlebot 3 as a Robotics Education Platform”. En: *Robotics in Education*. Ed. por Munir Merdan, Wilfried Lepuschitz, Gottfried Koppensteiner, Richard Balogh y David Obdržálek. Cham: Springer International Publishing, 2020, págs. 170-181. ISBN: 978-3-030-26945-6.
- [24] Tomas Horelican. “Utilizability of Navigation2/ROS2 in Highly Automated and Distributed Multi-Robotic Systems for Industrial Facilities”. En: *IFAC-PapersOnLine* 55.4 (2022). 17th IFAC Conference on Programmable Devices and Embedded Systems PDES 2022 — Sarajevo, Bosnia and Herzegovina, 17-19 May 2022, págs. 109-114. ISSN: 2405-8963. DOI: <https://doi.org/10.1016/j.ifacol.2022.06.018>. URL: <https://www.sciencedirect.com/science/article/pii/S2405896322003330>.
- [25] turtlebot. *Turtlebot*. <https://www.turtlebot.com/>. Accessed on 2022-08-07.
- [26] nav2. *Nav2*. <https://navigation.ros.org/>. Accessed on 2022-08-0.
- [27] <https://docs.ros.org/en/humble/>. Web oficial de ROS.
- [28] Huixu Dong, Ching-Yen Weng, Chuangqiang Guo, Haoyong Yu e I-Ming Chen. “Real-Time Avoidance Strategy of Dynamic Obstacles via Half Model-Free Detection and Tracking With 2D Lidar for Mobile Robots”. En: *IEEE/ASME Transactions on Mechatronics* 26.4 (2021), págs. 2215-2225. DOI: 10.1109/TMECH.2020.3034982.

- [29] Flavio B.P. Malavazi, Remy Guyonneau, Jean-Baptiste Fasquel, Sébastien Lagrange y Franck Mercier. “LiDAR-only based navigation algorithm for an autonomous agricultural robot”. En: *Computers and Electronics in Agriculture* 154 (2018), págs. 71-79. ISSN: 0168-1699. DOI: <https://doi.org/10.1016/j.compag.2018.08.034>. URL: <https://www.sciencedirect.com/science/article/pii/S0168169918302679>.
- [30] Ji-Soo Kim, Dong-Hun Lee, Dae-Woong Kim, Hyeri Park, Kwang-Jun Paik y Sanghyun Kim. “A numerical and experimental study on the obstacle collision avoidance system using a 2D LiDAR sensor for an autonomous surface vehicle”. En: *Ocean Engineering* 257 (2022), pág. 111508. ISSN: 0029-8018. DOI: <https://doi.org/10.1016/j.oceaneng.2022.111508>. URL: <https://www.sciencedirect.com/science/article/pii/S0029801822008794>.
- [31] Scott Manhart. *Autonomous Path Planning with LIDAR and Motion Capture*. <https://www.hackster.io/manhart3/autonomous-path-planning-with-lidar-and-motion-capture-6ff36f#overview>. Accessed on 2022-08-05.