Proyecto Fin de Carrera

Ingeniería electrónica, robótica y mecatrónica

Algoritmos de evitación de obstáculos usando el robot móvil ROSBOT 2.0

Autor: David Pérez Peralta

Tutor: José Ramón Domínguez Frejo

Tutora externa: Ana Belén Cordobés Menguiano

Dpto. de Ingeniería de Sistemas y Automática
Escuela Técnica Superior de Ingeniería
Universidad de Sevilla

Sevilla, 2021

Proyecto Fin de Carrera
Ingeniería electrónica, robótica y mecatrónica

Algoritmos de evitación de obstáculos usando el robot móvil ROSBOT 2.0

Autor:
David Pérez Peralta

Tutor:
José Ramón Domínguez Frejo
Profesor titular

Tutora externa:
Ana Belén Cordobés Menguiano

Dpto. de Ingeniería de Sistemas y Automática
Escuela Técnica Superior de Ingeniería
Universidad de Sevilla

Sevilla, 2021

Autor: David Pérez Peralta

Tutor: José Ramón Domínguez Frejo

Tutora externa: Ana Belén Cordobés Menguiano

El tribunal nombrado para juzgar el Proyecto arriba indicado, compuesto por los siguientes miembros:

Presidente:

Vocales:

Secretario:

Acuerdan otorgarle la calificación de:

Sevilla, 2021

El Secretario del Tribunal

AGRADECIMIENTOS

Llegué a Sevilla hace cuatro años, sin conocer a nadie y con la ilusión de empezar una nueva etapa en mi vida y sin darme cuenta ha pasado ya todo ese tiempo y me encuentro dando el último paso antes de acabar esta preciosa etapa. No tengo muchas palabras para describir lo que siento, pero sobre todo es alegría y felicidad por lo bonito que ha sido todo.

Gracias a mis compañeros de la carrera, que empezaron siendo unos desconocidos con los que resolver dudas para exámenes y han acabado siendo compañeros de vida. Sin duda, no estaría ahora escribiendo esto si no fuera por esos eternos días estudiando juntos, por todo lo que me habéis ayudado, las dudas infinitas que me habéis resuelto, por los grandes proyectos que hemos hecho juntos y por supuesto por la diversión necesaria tras realizar un buen examen. Estoy súper orgulloso del grupo que hemos formado y de todos vosotros.

Gracias a mis amigos, sois un pilar fundamental de mi vida. Gracias por aguantarme cuando estaba insoportable y por entenderme siempre, por sacarme una sonrisa en los peores momentos y por las escapadas y planes improvisados que hemos hecho. Porque un día conoces a una persona y no sabes lo importante que será el día de mañana, eso es lo bonito de la amistad. En especial a Pablo, Alberto y Zema, por haber sido mi luz en Sevillla y por haberme regalado tantos momentos a vuestro lado. Os habéis convertido en hermanos. Como a los 4 ases les gusta viajar, para el año que viene propongo Milán y los Alpes suizos, ¿qué os parece?

Gracias a mi familia, por apoyarme y decirme que el examen será fácil, aunque no entendáis nada de la materia. Esos sencillos gestos de amor incondicional me han dado la fuerza que necesitaba en muchas ocasiones.

Gracias a mi hermano, porque, aunque siempre te rías de mi se que en el fondo me apoyas sin importar nada. Gracias por estar ahí para hablar cuando lo necesitaba y por todas las coñas y risas que nunca van a faltar entre nosotros. Eres la persona más importante de mi vida, estoy muy orgulloso de la gran persona en la que te estás convirtiendo y quiero seguir creciendo y aprendiendo contigo.

Gracias a mis padres, nunca os podré agradecer lo suficiente ni mostrar lo importante que sois para mí. Gracias por confiar en mí siempre, cuando ni yo lo hacía, por creer en mi cuando yo no creía, sin yo saberlo me estabais dando la fuerza que necesitaba para seguir. Por todas esas llamadas que empezaba enfadado y acababa sonriendo como un niño pequeño con un juguete nuevo, y es que solo veros la cara me ilumina el día. Ojalá algún día pueda ser como vosotros, pero con llegar a la mitad será más que suficiente.

Porque la vida es un juego y hemos venido a jugar.

RESUMEN

La navegación autónoma de un robot es un problema complicado que involucra muchas tareas individuales que deben funcionar en conjunto para conseguir un objetivo común. Actualmente, es un problema de gran interés mundial debido al potencial que tiene para facilitar las labores en casi cualquier ámbito de trabajo. De las tareas necesarias para la navegación autónoma, quizás la evitación de obstáculos sea la más complicada de resolver. El mundo real está en constante cambio, y para que un robot pueda considerarse autónomo debe ser capaz de adaptarse a esos cambios y llegar al punto destino sin colisionar.

En este proyecto se abordará precisamente la parte de evitación de obstáculos, separada en dos secciones. Ambas se han desarrollado utilizando ROS y se han probado en el robot de Husarion ROSBOT 2.0. La primera parte consiste en desarrollar un método de evitación de colisiones, que evite que el robot colisione con cualquier objeto, parándolo si esto va a suceder. La idea es que el programa se ejecute en segundo plano mientras se realizan pruebas en otros aspectos con el robot, y que si este va a chocar actúe el programa desarrollado y lo evite. Así se podrían realizar pruebas con el robot con mayor seguridad.

La segunda parte consiste en implementar y comparar varios algoritmos de evitación de obstáculos en el robot. Los algoritmos seleccionados son Lazy Theta Star (LZS) y el de Campos Potenciales Artificiales (APF). Primero se desarrollará un simulador en C++ que permita probar los algoritmos en un mapa previamente configurado, con el fin de depurarlos y comprobar su correcto funcionamiento. Después se realizarán otras pruebas en el robot ROSBOT 2.0, analizando la eficiencia y el funcionamiento de los métodos en situaciones reales.

ABSTRACT

The problem of autonomous navigation in a robot is a complex one that requires lots of individual tasks to work together to achieve a common goal. Nowadays, it is a problem of great global interest due to the potential that is has to ease labor in almost every work environment. Obstacle avoidance may be the hardest of the tasks involved in autonomous navigation. The real world is constantly changing, and to be able to call a robot autonomous it must be able to adapt to those changes and get to the goal point.

In this project the obstacle avoidance problem is precisely the one that will be tackled, separated in two sections. Both have been developed using ROS and have been tested in the Husarion ROSBOT 2.0 robot. The first part consists in the development of a method that ensures that the robot never collides with an object, stopping it if this is going to happen. The idea is to run the program in the background while tests are run in other aspects of the robot, and if it is going to collide, the program will act and stop it from doing so. Thus, tests could be carried out in the robot with greater safety.

The second part consists in the implementation and comparison of several obstacle avoidance algorithms in the robot. The selected algorithms are Lazy Theta Star (LZS) and Artificial Potential Field (APF). First, a simulator in C++ will be developed to test the different algorithms in a previously loaded map, with the objective to debug and check they have a correct behavior. Then other tests will be run in the ROSBOT 2.0 robot, analyzing the efficiency and correct behavior of both methods in real situations.

ÍNDICE

ÍNDICE DE FIGURAS

ÍNDICE DE GRÁFICAS

1 INTRODUCCIÓN

Desde el primer uso de la palabra 'robot' en 1920 [1], su utilización no ha hecho más que aumentar, sobre todo en los últimos años. Se han desarrollado robots de múltiples formas, tamaños y funciones, que facilitan enormemente la vida de las personas. El objeto de estudio de este trabajo serán los robots móviles, concretamente el aspecto de movimiento autónomo de estos por entornos tanto conocidos como desconocidos.

El mundo en el que vivimos está en constante cambio, por eso cuando se indica a un robot móvil que se desplace de un punto a otro destino, hay infinidad de posibilidades para llegar a él, pero el objetivo es conseguirlo de la forma más eficiente posible.

Existen numerosos algoritmos que permiten a un robot desplazarse evitando obstáculos con un mapa conocido del entorno. Sin embargo, cuando alguno de estos obstáculos cambia de posición o aparecen otros nuevos, si no se actualiza la trayectoria a seguir, el robot podría colisionar con alguno de ellos. Estos cambios del entorno no son tan raros como puede parecer, por lo que surge la necesidad de desarrollar algoritmos de evitación de obstáculos que sean robustos, fiables y efectivos. Por eso, en las últimas décadas ha habido un creciente interés en el desarrollo de estos algoritmos.

Este proyecto tratará del desarrollo de dos partes. Por un lado, se desarrollará un método de evitación de colisiones que utilice los sensores del robot y lo pare cuando este vaya a colisionar con algún obstáculo. La idea de esta parte es que sea un módulo que pueda ejecutarse en segundo plano mientras se prueban otros algoritmos o tareas con el robot. Como las otras tareas estarán en desarrollo, podrían fallar y conducir al robot directamente hacia un objeto, posiblemente dañándolo tanto física como eléctricamente.

Por otro lado, se investigará en el ámbito de planificadores locales (métodos de evitación de obstáculos) y se desarrollarán varios algoritmos para ello. Posteriormente, se probarán en diferentes situaciones y se compararán con diferentes métricas, pudiendo analizar así las ventajas o desventajas de cada uno.

Para poder entender y probar mejor los algoritmos a desarrollar, antes de probarlos en el robot directamente se realizarán pruebas en un mapa simulado para comprobar el correcto funcionamiento de estos. Así se aprovechará mejor el tiempo y se podrá dedicar más a realizar distintas pruebas.

De esta forma, los objetivos del proyecto serán los siguientes:

- Aprender el funcionamiento básico del robot ROSBOT 2.0 y cómo enviarle comandos de movimiento.
- Aprender el funcionamiento básico de ROS.
- Investigar distintos métodos de evitación de obstáculos y entender los puntos fuertes y las limitaciones de cada uno.
- Realizar pequeñas pruebas en el robot con publicadores, suscriptores y archivos LAUNCH para familiarizarse con el entorno de trabajo futuro.
- Desarrollar un método que evite que el robot se choque con cualquier obstáculo. Debe ser robusto y funcionar tanto para objetos estáticos como móviles.
- Investigar y entender el funcionamiento de los planificadores globales y locales y su aplicación en la navegación autónoma.
- Implementar dos algoritmos de evitación de obstáculos en C++ y probarlos tanto en simulación como en el robot de Husarion ROSBOT 2.0.
- Comparar y analizar el funcionamiento de ambos algoritmos en diversas situaciones.

Por tanto, la memoria quedará dividida en varios capítulos que tratarán los puntos anteriores (salvo el tercer punto, ya que se trata de pruebas menores que no tienen que ver con el contexto del proyecto necesariamente).

2 ESTADO DEL ARTE

2.1 Navegación en robots móviles

La navegación de un robot móvil por un entorno es una tarea compleja que debe tratarse correctamente. Las tareas que la componen [2] son:

- Percepción del mundo.
- Planificación de la ruta.
- Generación del camino.
- Seguimiento del camino.

Con el uso de los sensores del vehículo se creará un mapa del entorno. Utilizando ese mapa, se generarán un número de objetivos que el robot deberá seguir. Por último, se efectúa dicho desplazamiento a cada punto del camino encontrado, usando un control adecuado de los actuadores del robot.

Existen diferentes tipos de navegación que siguen los pasos indicados anteriormente: navegación deliberativa y reactiva.

La navegación deliberativa se basa únicamente en el mapa obtenido con los sensores para realizar todos los cálculos [3]. Esto incluye la percepción del mundo a través de los sensores, la localización del robot en el propio mapa y la planificación y seguimiento de una trayectoria libre de obstáculos. Por otro lado, la navegación reactiva se basa en cada instante en la información proporcionada por los sensores para decidir qué comportamiento seguir [3].

Normalmente se usa una combinación de ambos métodos para dotar al vehículo de una estrategia más completa y robusta de movimiento. Así, la navegación deliberativa correspondería con un planificador global, que utiliza la información en un momento dado de los sensores (el mapa actual) para generar una trayectoria al punto destino libre de obstáculos. Y la navegación reactiva correspondería con un planificador local que se encargará de evitar obstáculos que no corresponden con los del mapa, bien porque han cambiado de posición o porque antes directamente no estaban.

2.2 Localización y mapeado simultáneo

El SLAM, o localización y mapeado simultáneo ("Simultaneous Localization and Mapping" en inglés), es una técnica muy extendida en robótica que permite a un robot crear un mapa de su entorno y situarse dentro de ese mapa a la vez. Se trata de una técnica complicada, pues engloba dos problemas complejos: la creación de un mapa fiable del entorno y una buena estimación de la posición del robot en ese mapa. Además, ambas partes están estrechamente relacionadas, por lo que un error en una de ellas provoca un error en la otra.

SLAM es válido tanto para interiores como para exteriores e incluso debajo del agua o el espacio con los sensores adecuados. El mapa obtenido puede estar representado de diversas formas. Puede ser 2D o 3D, basado en balizas, en celdas o en una nube de puntos [4].

A través de las acciones de control del robot y de las observaciones de los sensores, se pretende obtener el mapa y estimar la posición del robot en él.

For definition of SLAM problem we use given values (1,2) and expected values (3,4):

1. Robot control

$$u_{\{1:t\}} = \{ u_1, u_2, u_3, \ldots, u_t \} \quad (1)$$

2. Observations

$$z_{\{1:t\}} = \{ z_1, z_2, z_3, \ldots, z_t \} \quad (2)$$

3. Environment map

$$m \quad (3)$$

4. Robot trajectory

$$x_{\{1:t\}} = \{ x_1, x_2, x_3, \ldots, x_t \} \quad (4)$$

Figura 2-1. Definición del problema SLAM. Obtenida de [5].

Para resolver los problemas que plantea el SLAM, el método más extendido actualmente es el de EKF SLAM, que emplea un filtro extendido de Kalman [6], un algoritmo que estima valores desconocidos de variables a partir de mediciones tomadas en un periodo de tiempo.

En ROS existen paquetes que integran varios algoritmos de SLAM, como *HectorSLAM* [7], que implementa SLAM en 2D que no utiliza odometría, por lo que es válido para robots aéreos, o *gmapping* [8], que utiliza un láser para generar el SLAM. *GMapping* utiliza el filtro probabilístico de partículas Rao-Blackwellized, en el que cada partícula tiene asociado un mapa y se le aplica un peso en relación a la probabilidad de que ese mapa sea correcto. Este método es el más utilizado actualmente y es el que se usará en el desarrollo de este proyecto para obtener un mapa del entorno con el LIDAR que tiene el ROSBOT 2.0.

2.3 Métodos de evitación de obstáculos

Existen numerosos métodos de evitación de obstáculos, cada uno con sus puntos débiles y fuertes. A continuación, se describirán brevemente algunos de los más usados en el ámbito de la robótica para buscar caminos evitando obstáculos estáticos sobre un mapa conocido.

A*

Este algoritmo añade una función heurística al conocido algoritmo de Dijkstra. Este algoritmo calcula un camino entre dos celdas de forma iterada. En cada iteración, examina las celdas vecinas a la celda padre. Esto se repite hasta que se encuentre la celda objetivo [9].

A* añade una cola de prioridad al algoritmo de Dijkstra, dando mayor prioridad a los nodos más cercanos al objetivo, reduciendo el número de celdas que hay que examinar.

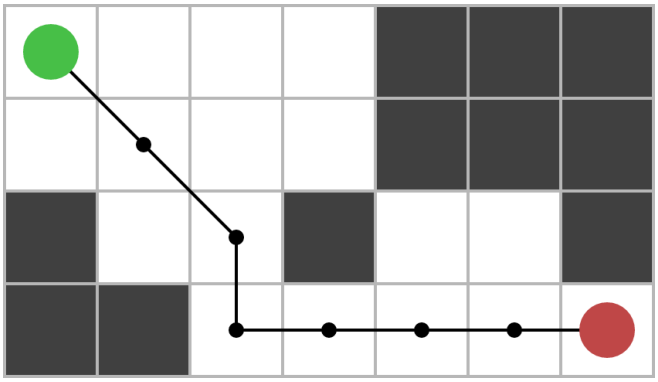


Figura 2-2. Camino resultante del algoritmo A*.
Obtenido de [10].

Campos Potenciales Artificiales (APF, "Artifitial Potential Field")

Este planificador es ampliamente usado en robótica. Consiste en definir un valor de un campo potencial a cada celda. El punto objetivo genera un potencial de atracción y los obstáculos uno de repulsión. La suma de ambos resulta en el potencial total. Este método está basado en los campos generados en la naturaleza, especialmente los electromagnéticos [9].

Fórmulas del potencial de atracción y de repulsión. Obtenido de [9].

$$U_{atr} = \frac{1}{2} \times k_a \times pdist(pose, goal)$$

$$U_{rep} = \frac{k_r}{pdist(pose, obst_i)}$$

El potencial de atracción es proporcional a la distancia al objetivo, y el de repulsión inversamente proporcional al obstáculo en cuestión. Una vez se ha calculado el campo potencial para cada celda, el camino a seguir será moverse siempre a la celda vecina con el menor potencial, pues la celda destino tendrá potencial 0. Como no tiene sentido que un obstáculo provoque un desplazamiento en el robot a una gran distancia, se establece un radio de influencia para los obstáculos. Si la distancia del robot al obstáculo es menor a ese radio, se utiliza la fórmula anterior para calcular el potencial de repulsión, pero si la distancia es mayor se usará un potencial de repulsión nulo, de forma que solo afecte el de atracción del punto objetivo.

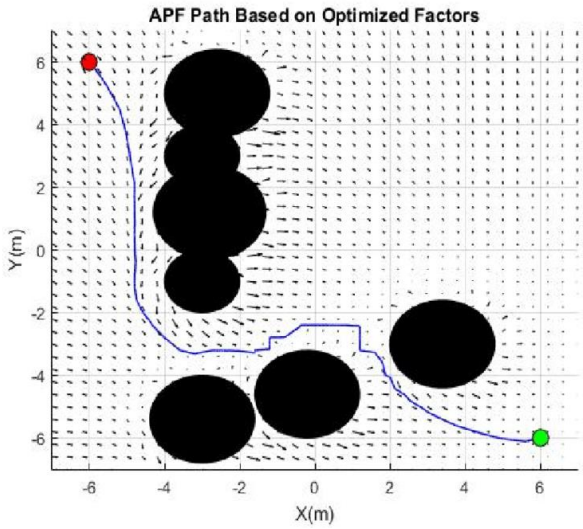


Figura 2-3. Resultado de un camino obtenido con el algoritmo APF. Obtenida de [11].

Dynamic Window Approach (DWA)

Este método considera las limitaciones dinámicas y cinemáticas del robot en cuestión. Consiste en considerar diversas trayectorias de arco con distinta velocidad lineal y angular. Después, se selecciona una trayectoria hasta el siguiente punto con las velocidades adecuadas para que no se choque por el camino.

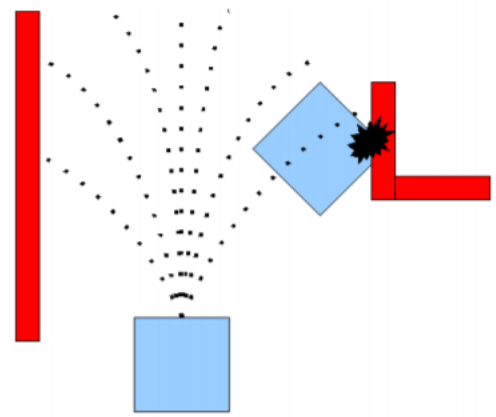


Figura 2-4. Representación del algoritmo DWA. Obtenida de [12].

Rapid-Exploring Random Trees* (RRT)

Este algoritmo genera un árbol con nodos y ramas conectados entre sí. Los nodos se generan aleatoriamente y el algoritmo para cuando un nodo está suficientemente cerca del punto objetivo. Existen diversas modificaciones que generan los nodos en el entorno del punto objetivo, mejorando la eficiencia y encontrando un camino más rápidamente.

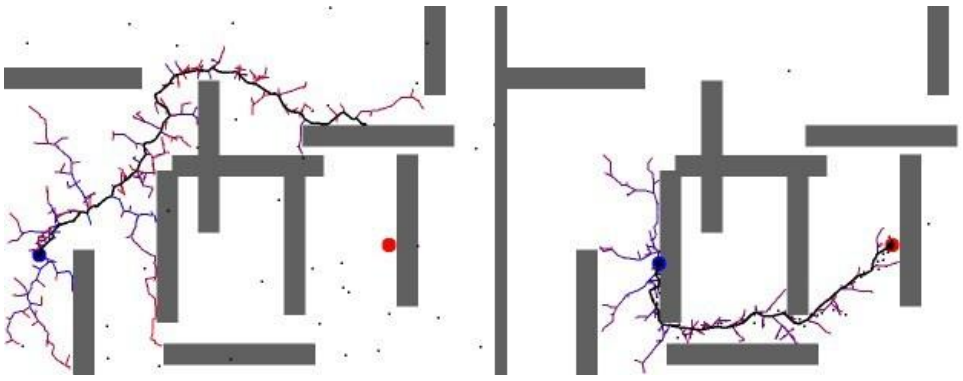


Figura 2-5. Representación de un camino encontrado con el método RRT. Obtenida de [13].

Lazy Theta *

Este algoritmo está basado en Theta*, que a su vez está basado en A*. Al igual que Theta*, busca caminos cuyos puntos pueden estar unidos por cualquier ángulo, eliminando esa restricción de A*. Al poder seleccionar como siguiente punto del camino cualquier otro del mapa, se consigue gran eficiencia en cuanto a distancia recorrida se refiere.

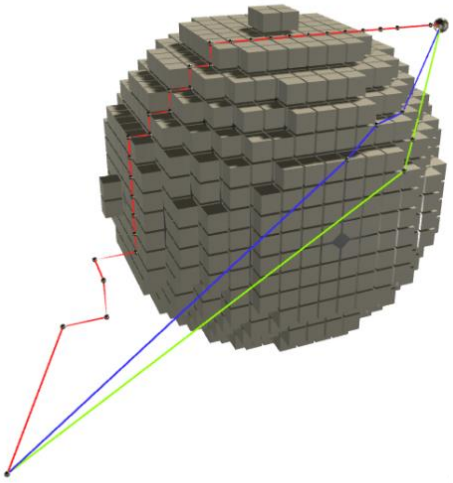


Figura 2-6. Representación de un camino en distintos algoritmos: A* en rojo, Theta* en azul y Lazy Theta* en verde. Obtenida de [14].

2.4 Métricas para comparar algoritmos

Cada algoritmo tiene sus puntos fuertes y débiles, por eso es necesario establecer alguna forma de compararlos efectivamente, unas métricas adecuadas. Se usarán las siguientes métricas:

- Tiempo empleado en el cálculo del camino (para pruebas en simulación).
- Longitud del camino resultante (para pruebas en simulación).
- Distancia real recorrida por el robot (para pruebas con el robot).
- Tiempo empleado por el robot en llegar al objetivo (para pruebas con el robot).

3 ROS (ROBOT OPERATING SYSTEM)

3.1 Introducción a ROS

ROS (Robot Operating System) es un *framework* que permite desarrollar software para robots. Actualmente, es la plataforma más usada en el mundo para desarrollar aplicaciones robóticas.

Contiene grandes conjuntos de herramientas y librerías con el objetivo de simplificar la tarea de crear comportamientos complejos y robustos en robots de diferentes plataformas [15]. Está pensado para trabajar de forma colaborativa, permitiendo desglosar un gran proyecto en diferentes secciones más pequeñas, dedicando un grupo de trabajo a cada una, para luego integrarlo todo con ROS. De esta forma, un grupo de expertos podría dedicarse a desarrollar un sistema para generar mapas del entorno mientras que otro grupo podría encargarse de la navegación por dicho mapa.

ROS permite desarrollar archivos en C++ y en Python. Para este proyecto se usará únicamente C++, pues al ser un lenguaje compilado, frente al interpretado Python, lo hace mucho más rápido y eficiente a la hora de ejecutarse. Por el contrario, a la hora de probar y depurar código, es un trabajo más pesado y lento, pues un cambio mínimo obligará a compilar de nuevo y enlazar todos los archivos.

Una de las dificultades de este proyecto es precisamente el tener que desarrollar todo el código propio utilizando C++, pues es un lenguaje de programación que no he utilizado nunca personalmente. Por tanto, para poder entender documentación, código de otras personas o archivos ya implementados en el firmware del robot tendré que realizar un esfuerzo extra.

3.2 Arquitectura de trabajo

El funcionamiento de ROS está basado en una arquitectura de grafos. De esta manera, la información se gestiona mediante la comunicación entre nodos de mensajes provenientes de distintas fuentes, como sensores, motores, actuadores, etc.

Los elementos principales de la arquitectura son:

Paquetes

Son la unidad básica de organización en ROS. Un paquete es un conjunto de archivos (ejecutables, librerías, tipos de mensajes, etc.) que efectúa una tarea concreta. Tiene gran portabilidad, pues para utilizar un paquete en otro proyecto basta con copiar la carpeta que lo contiene, instalar dependencias (si las requiere) y compilar (si es necesario).

Nodos

Un nodo es un ejecutable dentro de un paquete. La comunicación entre nodos es posible mediante el uso de tópicos. Un nodo puede suscribirse a un tópico y recibir la información que este transmite. También puede publicar en un tópico, de forma que esa información será recibida por todos los nodos que estén suscritos a él. Un nodo puede ser a la vez publicador y suscriptor de diferentes tópicos.

Mensajes

Los nodos utilizan mensajes para comunicarse entre sí. Para que dos nodos compartan información, deben comunicarse utilizando el mismo tipo de mensaje. ROS incluye tipos de mensajes básicos como int, double o bool, pero también permite crear mensajes creando estructuras o vectores, de forma que se pueden personalizar para cada tarea. Por ejemplo, puede haber un mensaje que describa un punto en 3D o uno que describa un mapa creado mediante SLAM.

Tópicos

Los tópicos son el medio por el cual se transmiten los mensajes. Cada tópico solo puede transmitir mensajes de un tipo, por lo que todos los mensajes publicados o recibidos a través de él serán de ese tipo.

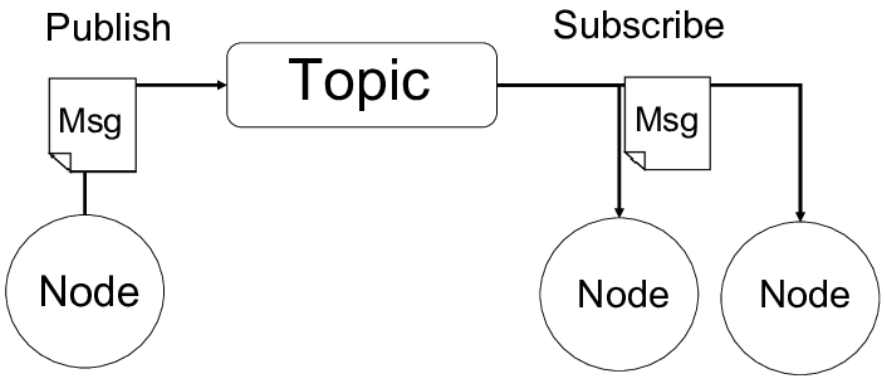


Figura 3-1. Esquema de comunicación entre nodos mediante mensajes y tópicos.
Obtenida de [16].

Servicios

Los tópicos no son apropiados para una comunicación de tipo petición-respuesta, los servicios resuelven este problema. Un servicio puede contener dos tipos de mensajes, uno para la petición y otro para la respuesta. Un nodo actuará como cliente, enviando la petición, mientras que otro será el servidor, proporcionando la respuesta.

Servidor de Parámetros

Los parámetros se almacenan en una base de datos y pueden ser accedidos por los nodos en el momento de ejecución. Son una forma útil de cambiar el funcionamiento de un nodo en tiempo real.

Maestro

El maestro (*rosmaster*) es una unidad por encima de todos los elementos explicados hasta ahora. Se encarga de registrar los nodos, tópicos y mensajes y de la gestión de estos y la comunicación entre ellos.

Manifiesto

Es un fichero (package.xml) que contiene información importante del paquete, como el nombre, versión, licencias, autores o dependencias de otros paquetes.

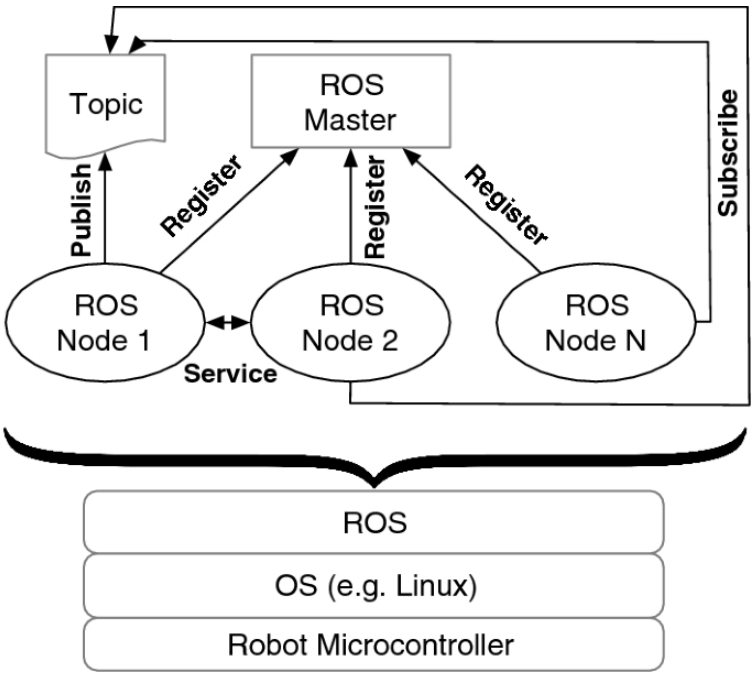


Figura 3-2. Esquema de funcionamiento del maestro y su comunicación con el resto de los elementos. Obtenida de [17].

ROS se creó pensando en un uso conjunto entre diferentes equipos, y hacerlo es tan sencillo como establecer una vía de comunicación entre el maestro de esos equipos, para así poder acceder a toda su estructura interna.

Por último, otra de las grandes ventajas de ROS es la gran comunidad y los amplios recursos que tiene, pudiendo acceder a foros y repositorios oficiales para consultar dudas o descargar paquetes.

4 HUSARION ROSBOT 2.0

Todo el desarrollo de este trabajo se realizará con el robot comercial Husarion ROSbot 2.0, de la empresa Husarion. Como se indica en su propia web, ROSbot es una plataforma de código abierto y autónoma basada en ROS [18]. Aunque tiene un modelo para su simulación en Gazebo, todas las pruebas se realizarán en el robot físico.

Figura 4-1. Imagen de ROSbot 2.0. Obtenida de [18].

El robot viene preparado con todo lo necesario para empezar a trabajar con él. Tiene integrado un ordenador con el sistema operativo Ubuntu y cuenta con los siguientes sensores:

- IMU: equipada con acelerómetro y giroscopio.
- 4 sensores infrarrojos para medir distancias (dos delante y dos detrás).
- Cámara RGBD.
- LIDAR.

El movimiento del robot se consigue controlando cada rueda individualmente, por lo que permite movimientos precisos como rotaciones y giros de forma sencilla. Al igual que muchos robots que basados en ROS, el movimiento se consigue mediante el topic "/cmd_vel", con un mensaje de tipo "geometry_msgs/Twist". Esto hace que se puedan usar multitud de paquetes para probar diferentes aspectos en él.

Por defecto vienen instalados numerosos paquetes de ROS. Sin embargo, el de mayor utilidad es "rosbot_ekf". Este paquete inicializa el firmware de la máquina. Tiene ya implementado un filtro de Kalman extendido para combinar los datos recibidos de los *encoders* y del módulo IMU para estimar mejor la posición y orientación. También genera los *topics* principales para acceder a los datos de los sensores de

distancia, de movimiento del robot o de la cámara.

Para activar el LIDAR es necesario ejecutar el nodo “rplidar_ros”. Esto será necesario para llevar a cabo el SLAM, a partir del cual se construirá un mapa del entorno.

4.1 Entorno de trabajo

Antes de empezar a realizar pruebas con el robot, se ha de preparar correctamente el entorno de trabajo para poder usar el robot de forma eficiente.

El robot, al tener un ordenador completo integrado, puede ser conectado mediante un cable HDMI a una pantalla y desde ahí desarrollar el código necesario y ver archivos, entre otras tareas. Sin embargo, a la hora de realizar pruebas, cuando este se encuentre desplazándose por el suelo, no se podrán ver los cambios en tiempo real ni ejecutar comandos (como publicar a un topic, por ejemplo).

Existen soluciones para cada problema mencionado. Primero, para poder acceder a los archivos del robot mediante un ordenador remoto, basta con establecer una conexión *ssh* mediante ambos equipos. De esta manera, podremos ejecutar nodos y publicar en topics en cualquier momento, aunque el robot se esté moviendo.

Por otro lado, para poder ver los datos que va obteniendo el robot en tiempo real, se utilizará la herramienta "rviz", muy conocida entre la comunidad ROS. Este programa permite ver distintos topics en tiempo real. Algunos, como la generación de mapas, la planificación de trayectorias o los datos del LIDAR son muy útiles para visualizar aquí.

Gracias a que ROS se ha desarrollado pensando en un entorno de trabajo conformado por varios ordenadores y máquinas, esto se puede lograr de manera sencilla. Existe un parámetro llamado "ROS_MASTER_URI", que es una URI que apunta al proceso que controla todo el comportamiento de la sesión. Si en una terminal en el ordenador remoto establecemos este parámetro para que "apunte" al robot, ya estarán conectados.

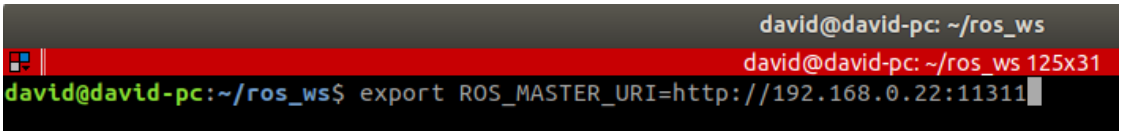


Figura 4-2. Comando export ROS_MASTER_URI

Aun así, esto no será necesario para poder ver los datos publicados por el robot en tiempo real, pues en mi caso sucedía que mi equipo no podía resolver el hostname del robot en su IP. Para solucionar eso, hubo que añadir una entrada al archivo que se encarga precisamente de esto: "etc/hosts".

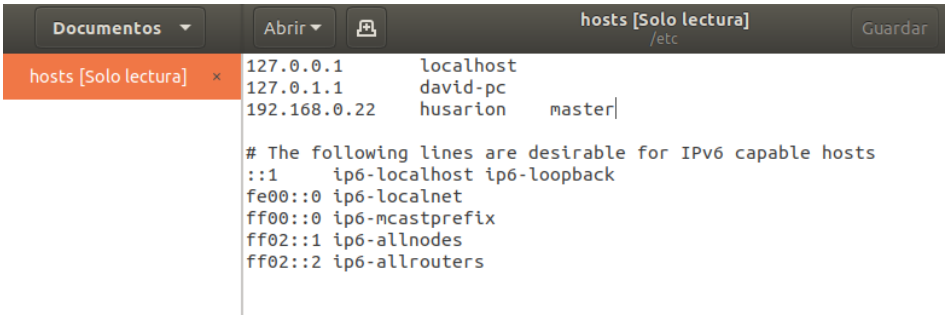


Figura 4-3. Nueva entrada del archivo etc/hosts

Tras realizar esto, la conexión entre el robot y el ordenador remoto estará lista, y se podrán ejecutar tanto nodos como comandos como ver los datos publicados del robot desde el ordenador remoto.

5 DESARROLLO

5.1 Evitación de colisiones

5.1.1 Evitación de colisiones publicando en el topic "/cmd_vel"

La primera parte del proyecto consiste en desarrollar un paquete que se asegure de que el robot nunca se choque. En cualquier momento en el que esté en movimiento y vaya a colisionar con algún objeto, el paquete deberá parar el robot para que no se choque y así evitar posibles daños en él. Esto será muy útil sobre todo a la hora de realizar pruebas en un futuro, pues independientemente de las órdenes que se le den, se tendrá la seguridad de que no se va a chocar. Para lograr este objetivo se hará uso tanto de los sensores de distancia infrarrojo como del LIDAR.

La idea del paquete es que pueda ser usado para otros robots (basados en ROS obviamente), que se ejecute cuando se quieran realizar pruebas y que esté corriendo en segundo plano. Por eso, debe ser lo más genérico posible y fácil de usar. El nombre que se le ha dado al paquete es "no_collision", y dentro se ha creado un archivo en C++ llamado "script".

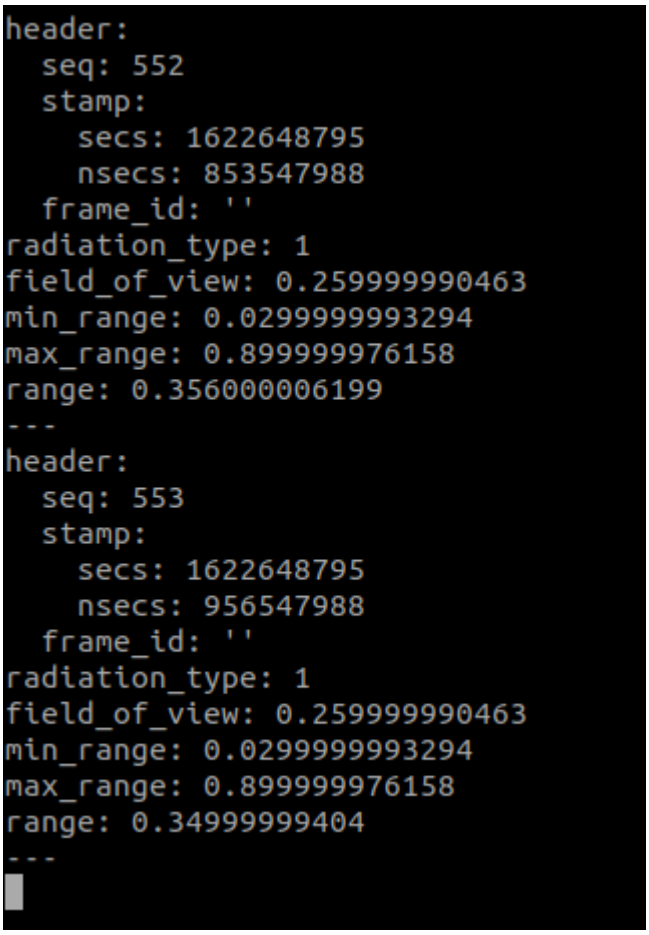


Figura 5-1. Datos del topic /range /fl

Al principio la idea del nodo era que se suscribiera al topic /cmd_vel, que es donde se publican los comandos de velocidad que mueven el robot. El tipo de mensaje de este topic es geometry_msgs/Twist, con campos para la velocidad lineal y angular. También se suscribirá a los topics de los sensores de distancia, con mensaje sensor_msgs/range (/range/fl es el topic correspondiente al sensor delantero izquierdo, por ejemplo). Este tipo de mensaje contiene varios datos importantes acerca del sensor, como el ángulo de visión o la distancia mínima y máxima para que una lectura sea válida [19]. En este caso, la distancia mínima son 3 cm y la máxima 90 cm, por lo que no hay problema para utilizar los datos para el paquete, pues no será necesario usarlos para distancias mayores, aunque los sensores están capacitados para medir distancias de hasta 2 metros [20].

Para parar el robot se sigue la siguiente metodología. Se establece un parámetro llamado "min_dist", por defecto configurado a 25 cm. A partir de este valor se calcula otro parámetro llamado 'correctedMinDist', que es el resultado de multiplicar 'min_dist' por la velocidad lineal del robot y dividirlo entre 0.3.

$$correctedMinDist = minDist * \frac{velLin}{0.3}$$

Este parámetro saturará inferiormente a 15cm, que es el radio máximo del robot. El parámetro será proporcional a la velocidad lineal del robot. Si el robot se está moviendo hacia delante y la distancia medida por alguno de los sensores delanteros es menor que el parámetro por el factor, se publica al topic /cmd_vel un mensaje con todas las velocidades a cero, parando al robot. Por la inercia que lleva, avanzará un poco antes de parar por completo, por eso es importante configurar el factor correctamente. Lo mismo ocurre cuando se mueve hacia atrás y alguno de los sensores traseros mide una distancia menor al parámetro por el factor.

Para probar este método, se controla el robot con el paquete "teleop_twist_keyboard", que lee la pulsación de ciertas teclas y las convierte en mensajes de tipo "geometry_msgs/Twist", que se publican en el topic "/cmd_vel" y mueven el robot.

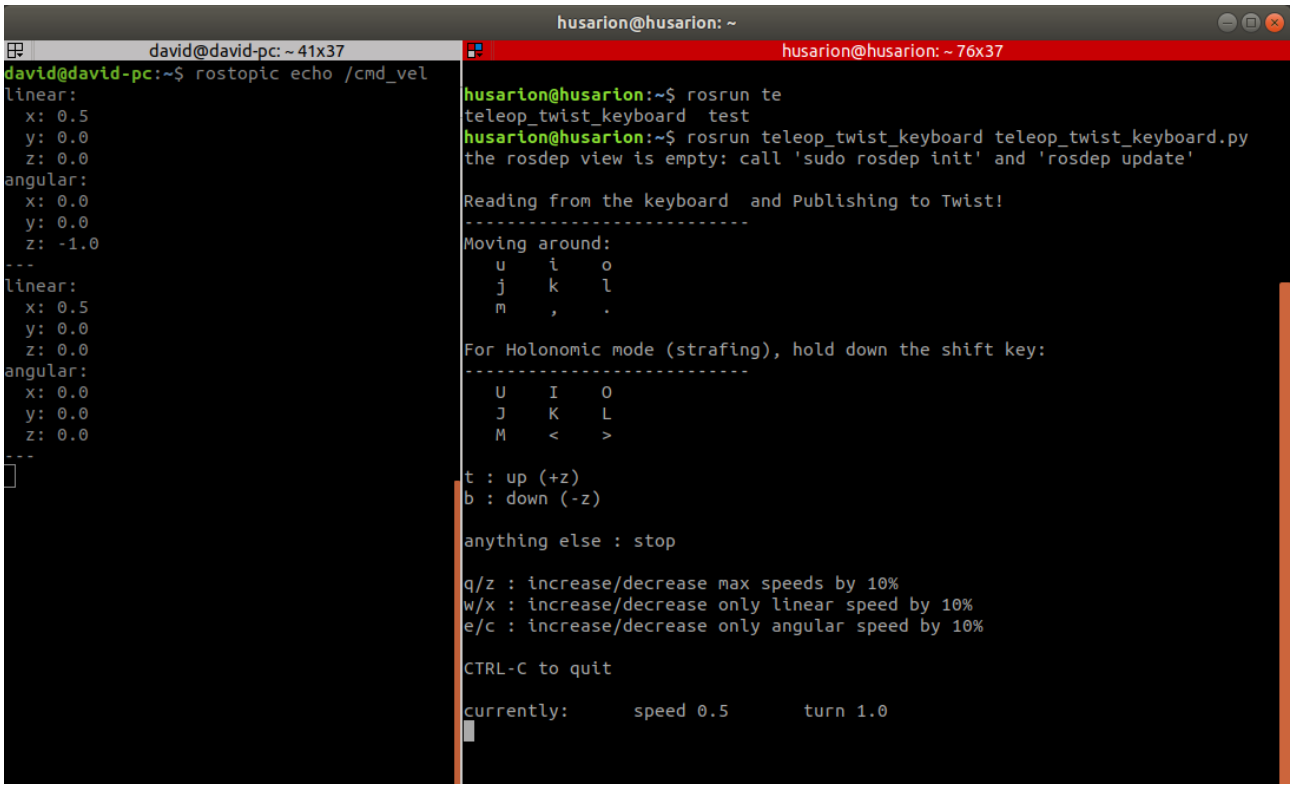


Figura 5-2. A la izquierda: datos del topic /cmd_vel, a la derecha el nodo para controlar manualmente a distancia.

Tras realizar varias pruebas, el método ha funcionado correctamente y cuando se indica al robot que se desplace hacia una pared, el nodo provoca que se pare. Si una vez se ha parado, se le vuelve a indicar que avance, este no se mueve. Pero esto no sucede siempre, si se mandan muchas órdenes para avanzar, al final una de ellas acaba burlando el nodo y provoca que el robot avance y se choque.

Esto sucede porque ocurre un conflicto entre el paquete "teleop_twist_keyboard", que publica un mensaje para que el robot se mueva, y el propio, que publica en el mismo topic pero esta vez un mensaje indicando que se pare. Cuando el robot está parado a una distancia menor a "min_dist" y se pulsa una tecla para que se mueva, se publica en el topic "/cmd_vel" un mensaje para que avance. A la vez, esta información llega al nodo del paquete "no_collision", que procesa la orden y como determina que no debe moverse porque sino chocaría, publica otra orden en ese mismo topic para que no se mueva.

Para evitar este conflicto, se hará uso del paquete "move_base" y de acciones de ROS.

5.1.2 Evitación de colisiones con “move_base” y acciones de ROS

El enfoque de esta solución es totalmente distinto a la anterior. El robot se moverá tras indicarle un punto de destino, y este trazará una trayectoria para llegar hasta él, en vez de moverlo manualmente con el paquete “teleop_twist_keyboard”. La forma de pararlo será cancelando el destino, pudiendo reanudarlo posteriormente si ya no hay peligro de colisión (el objeto era móvil y ya no está en el camino o se ha eliminado).

Conseguir todo esto no es posible con la metodología anterior, por eso se ha decidido usar el paquete “move_base” de ROS, que integra todos los elementos necesarios para el movimiento del robot a un punto destino. El paquete contiene un planificador global para el seguimiento de trayectorias y uno local para la evitación de obstáculos. Además, recoge datos de sensores para crear un mapa de su entorno y usar SLAM (Simultaneous Localization and Mapping, Localización y Mapeado Simultáneo). Con ello realiza dos mapas de costes que serán usados para evaluar los puntos que deben contener las trayectorias local y global. La localización del robot en su entorno lo realiza con el algoritmo probabilístico “amcl”. El esquema básico de funcionamiento del nodo es el siguiente.

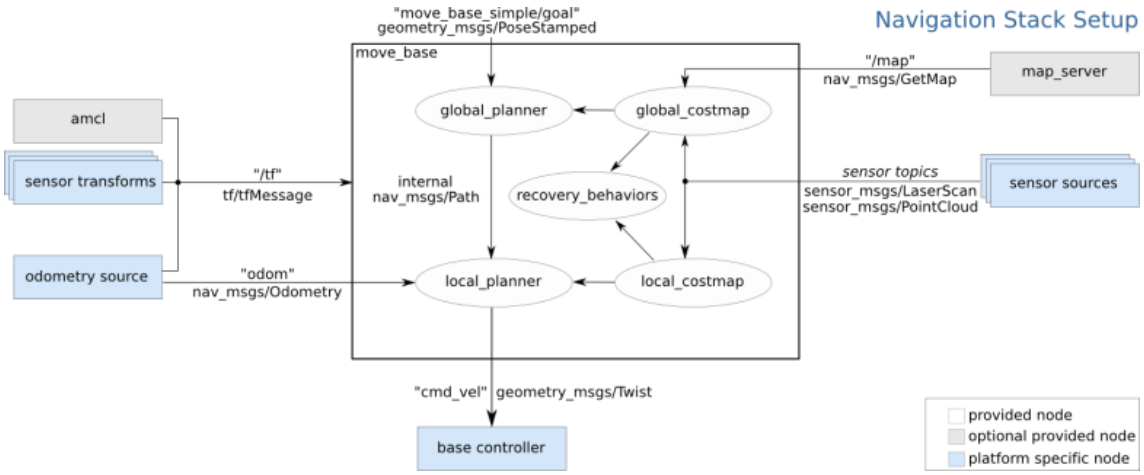


Figura 5-3. Esquema de funcionamiento del nodo “move_base”. Obtenida de [21].

El nodo tiene también comportamientos de recuperación, que usará cuando se encuentre atascado y no pueda alcanzar el objetivo. Estos consisten en limpiar obstáculos del mapa en la zona cercana al robot mediante rotaciones. Si aun así no consigue llegar al objetivo se procede a abortar y el robot se para.

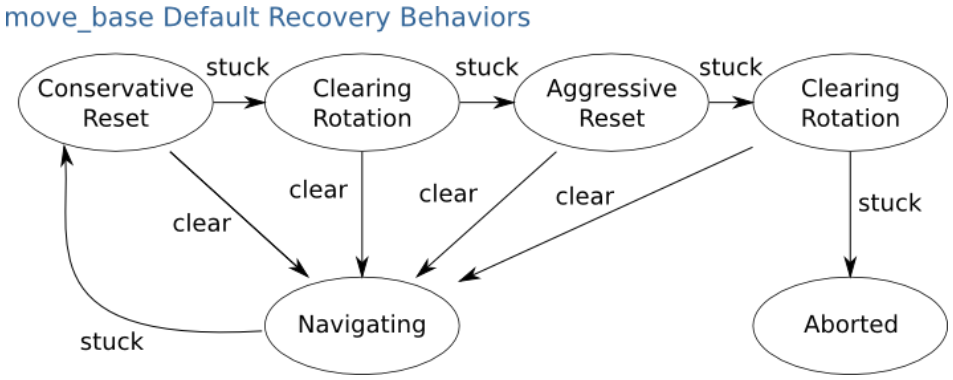


Figura 5-4. Comportamientos de recuperación del nodo “move_base”. Obtenida de [21].

El paquete está pensado para trabajar con acciones de ROS, que permiten establecer tareas de larga duración y obtener un seguimiento de su estado en tiempo real, con la posibilidad de cancelarlas. Esto se implementa en el paquete "actionlib". Las acciones funcionan creando un servidor que ejecuta el objetivo y un cliente que envía las peticiones. En el caso del paquete "move_base", mediante el topic "/move_base/goal" se pueden enviar los puntos de destino al robot, y posteriormente podrán cancelarse si se requiere.

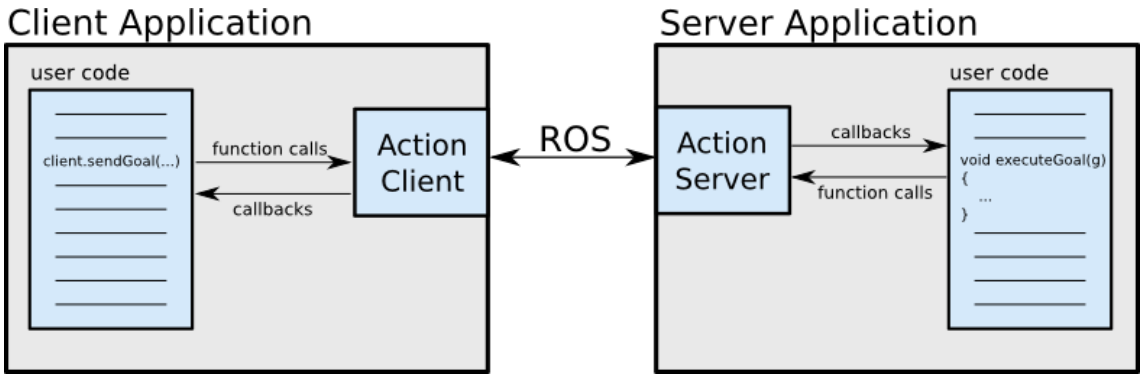


Figura 5-5. Esquema de funcionamiento del cliente y servidor de la acción. Obtenida de [22].

Ahora que se sabe cómo funcionan los paquetes, es necesario implementarlos correctamente. Para ello hay que configurar varios parámetros y se han desarrollado cinco archivos ".launch" con la siguiente información:

- Parámetros comunes del mapa de costes
- Parámetros del mapa de costes local
- Parámetros del mapa de costes global
- Parámetros del planificador de trayectorias
- Archivo que lanza el nodo "move_base" junto con "gmapping", para realizar SLAM.

Con el nodo "move_base" correctamente configurado, el robot evitará chocarse mientras se mueve e intentará realizar trayectorias libres de obstáculos. Esto lo consigue gracias a los datos obtenidos por el LIDAR, el problema es que está situado a unos 15 cm por encima del suelo, y como realiza las lecturas en un área paralela al mismo, si hay algún objeto pequeño delante del robot pegado al suelo, como un lápiz o un zapato, el robot no lo detectará y colisionará con él, provocando errores en odometría al chocar con las ruedas o incluso una parada o daños peores como el volteo del robot.

Por eso se ha procedido a usar los sensores de distancia, que sí detectarían esos objetos, tanto delante como detrás del robot. Además, detectan superficies transparentes como el cristal, algo que con la tecnología LIDAR no ocurre.

La implementación de la lógica de parada se ha desarrollado en el archivo "script.cpp", modificando el contenido anterior. Esta vez, cuando el robot se está desplazando hacia delante y los sensores detectan un objeto, en vez detenerlo forzadamente, se cancela el objetivo actual, pero antes se guarda en una variable interna.

De este modo, si el objeto se puede quitar, cuando el robot detecta que ya no hay obstáculo, publicará como objetivo el anterior y realizará la trayectoria correctamente.

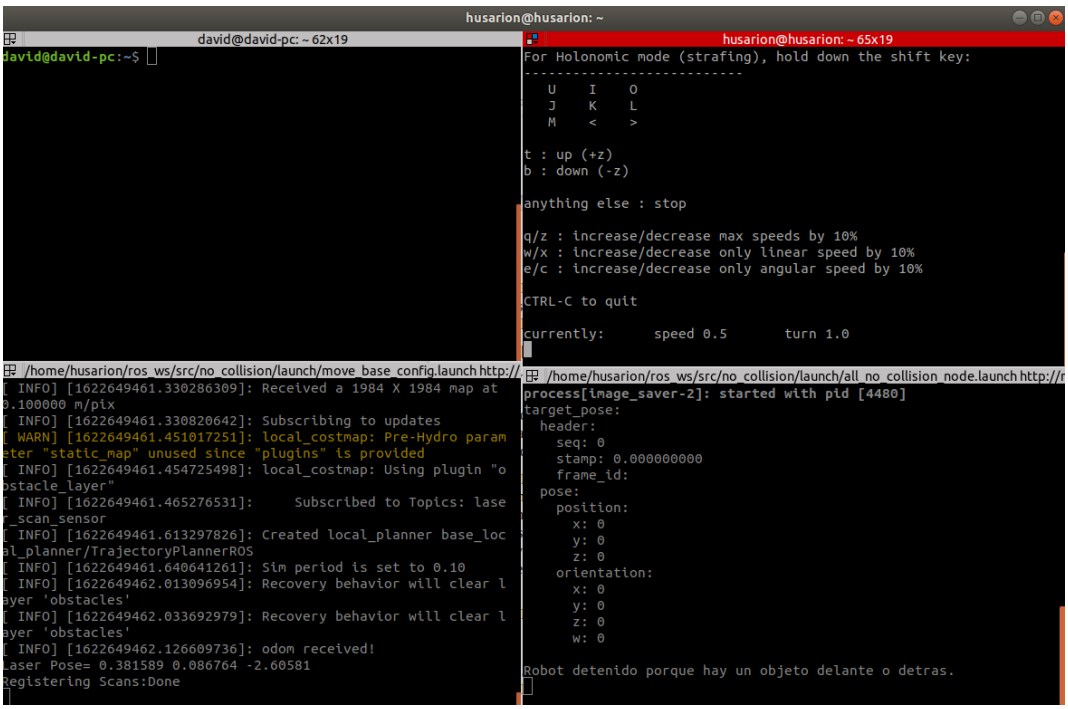


Figura 5-6. Abajo a la izquierda está el nodo “move_base”, abajo a la derecha el propio, con el aviso de detención, y arriba el nodo de control remoto manual.

Figura 5-7. Foto obtenida en la prueba de la captura anterior, controlando el robot manualmente para que se choque con una pared.

Esta sería la prueba más básica que muestra un correcto funcionamiento del método propuesto. Se han realizado más pruebas con situaciones más complejas, cuyos resultados se muestran en el capítulo 6.

5.2 Evitación de obstáculos

La segunda parte del proyecto consiste en desarrollar varios algoritmos de evitación de obstáculos, implementarlos en el robot y comprobar su funcionamiento en diferentes situaciones, realizando comparaciones adecuadas.

Dado que el movimiento del robot está basado en el paquete 'move_base', para que este se mueva hay que definir un punto objetivo mediante el topic "/move_base_simple/goal", que recibe un mensaje de tipo "geometry_msgs/PoseStamped". Este objetivo sería recibido por el planificador global, que trazaría una trayectoria inicial. Esta sería recibida por el planificador local, que se encarga de recorrerla evitando obstáculos y modificándola si es necesario, a la vez que manda los comando de movimiento al robot para que el control se haga correctamente.

En este caso se modificará el planificador local, usando el propio desarrollado, dejando el global por defecto. Ahora surge el problema de cómo hacer que el paquete 'move_base' pueda usar nuestro planificador local y los pasos necesarios para que esto suceda.

La respuesta está en el desarrollo de un plugin para el paquete, usando la API que este proporciona [23]. Para ello, primero hay que crear una clase que se adhiera a la interfaz 'nav_core::BaseLocalPlanner'. Se crea su correspondiente archivo de cabecera, que contendrá las variables y el prototipo de las funciones necesarias.

En el archivo .cpp se implementará la clase. Las funciones mínimas que debe contener son [17]:

- **initialize**(std::string name, tf::TransformListener *tf, costmap_2d::Costmap2DROS *costmap_ros)
- **setPlan**(const std::vector< geometry_msgs::PoseStamped > &plan)
- **computeVelocityCommands** (geometry_msgs::Twist &cmd_vel)
- **isGoalReached**()

Describamos brevemente cada una de ellas.

'Initialize' simplemente inicializa el planificador, asignando valores iniciales a las variables que lo necesiten. Esta función se llama solo una vez al ejecutar el nodo de 'move_base'.

La función 'setPlan' actualiza la trayectoria que el planificador local seguirá, y también se ejecutará una sola vez. Típicamente se selecciona el plan global recibido (&plan) como camino a seguir.

La función 'computeVelocityCommands' se ejecuta periódicamente y se encarga de realizar el control de movimiento del vehículo. La variable 'cmd_vel' que recibe será el mensaje que 'move_base' publicará automáticamente en el topic '/cmd_vel', que hará que el robot se mueva. Para calcular estos comandos de movimiento lineal y angular, se usará un control proporcional sencillo, pero a la vez efectivo.

La velocidad lineal será proporcional a la distancia al objetivo. Por otro lado, la velocidad angular también será proporcional al ángulo desde la posición actual hasta la posición deseada, teniendo en cuenta varios factores. Si el error angular (ángulo objetivo menos ángulo actual) es mayor a 30 grados, el robot tendrá velocidad lineal nula y se limitará a girar hacia el lado de menor recorrido. Cuando alcance un error menor de 50 grados, empezará a moverse linealmente con muy poca velocidad (0,05 m/s) mientras sigue girando, hasta que el error sea menor de 15 grados. En ese momento, ya se efectúa el control proporcional en velocidad lineal, pero saturando en 0.5 m/s.

Las ecuaciones de control se muestran en las siguientes líneas.

$$si\ errorAngular > 30º:$$

$$velLineal = 0$$
$$velAngular = 0.3 * errorAngular$$

$$si\ errorAngular > 15º:$$

$$velLineal = 0.05$$

$$velAngular = 0.5 * abs(errorAngular)$$

$$en\ otros\ casos\ (errorAngular < 15º):$$

$$velLineal = dist, \qquad (saturando\ a\ 0.5\ como\ margen\ superior)$$
$$velAngular = 0.75 * errorAngular$$

Estas decisiones de control ayudarán a establecer un movimiento suave y sin aceleraciones bruscas tanto lineales como angulares, que son las que provocan más errores, sobre todo en cuanto a la odometría.

Por último, la función 'isGoalReached'. Esta simplemente devuelve 'true' o 'false' si se ha llegado al punto destino o no, que en este caso será el último punto del vector de puntos recibidos en la trayectoria global. Esta función también se ejecuta periódicamente.

Una vez se ha escrito la clase, hay que registrar y definir el plugin. Lo primero se realiza dentro de la clase mediante el macro 'PLUGINLIB_EXPORT_CLASS'. Después, se crea un archivo XML que contiene información descriptiva del plugin, como el nombre o la interfaz a la que se adhiere. También hay que registrar el plugin en el sistema de paquetes de ROS, y esto se consigue modificando el manifiesto (archivo 'package.xml') del paquete que contiene el plugin.

```
<library path="lib/libapf_local_planner">
    <class name ="apf_local_planner/APFLocalPlanner" type ="apf_local_planner::APFLocalPlanner"
    base_class_type= "nav_core::BaseLocalPlanner">
        <description>
            Plugin for BaseLocalPlanner that implements the Artifitial Potential Field (APF)
            path planning method.
        </description>
    </class>
</library>
```

Figura 5-8. Archivo XML que registra el plugin del planificador de campos potenciales

El plugin ya está listo para usar, tan solo hace falta añadir una línea al archivo .launch que ejecuta el nodo 'move_base' para que use el planificador local propio.

5.2.1 Campos potenciales artificiales

El primer algoritmo de evitación de obstáculos que se desarrollará será el de campos potenciales artificiales (APF, de 'Aritifitial Potential Field'). Este algoritmo es bien conocido en el mundo de la robótica móvil y es ampliamente usado por su gran sencillez y poco coste computacional.

5.2.1.1 Descripción del algoritmo

El algoritmo APF está basado en el concepto de campos potenciales. Principalmente en el campo electromagnético, encontrado en la naturaleza. El robot se considera una partícula eléctricamente cargada, los obstáculos generan un potencial de repulsión y el punto objetivo un potencial de atracción. El potencial total será la suma de estos dos. El robot se moverá en todo momento hacia el punto con menor campo potencial, acabando en el punto objetivo, que tiene potencial nulo.

Para este trabajo, se dispondrá de un mapa en 2D compuesto por celdas. Cada celda estará a '0' si se considera espacio libre y a '1' si se considera un obstáculo. Para aplicar APF a un espacio discretizado como este, basta con calcular primero el campo potencial total para cada celda del mapa y después desplazar el robot a la celda vecina a la actual que tenga menor potencial hasta llegar al destino.

Por tanto, la fuerza total F de cada celda será la suma del potencial de atracción generado por la celda objetivo más la suma de los potenciales de todos los obstáculos [24].

$$\vec{F}(x,y) = \overrightarrow{F_G}(x,y) + \sum_i \vec{F}obs_i\,(x,y)$$

$$\vec{F}(x,y) = -\vec{\nabla}V$$

La fuerza resultante será un vector de dos componentes. Su gradiente es:

$$\vec{\nabla}V = \left[\frac{\partial V}{\partial x} \quad \frac{\partial V}{\partial y}\right]^T$$

$$\vec{\nabla}V = [Fx \quad Fy]^T$$

El valor de cada componente es el siguiente.

$$F_x(x,y) = -\frac{\partial V(x,y)}{\partial x}$$

$$F_y(x,y) = -\frac{\partial V(x,y)}{\partial y}$$

Se supone ahora que la celda objetivo tenga coordenadas (x1, y1) y que el robot está en la celda (x, y). El potencial atractivo se define como una función lineal y proporcional a la distancia Euclídea de cada celda a esa celda destino.

El potencial es:

$$V(r) = k*r$$

$$r = [(x - x_1)^2 + (y - y_1)^2]^{1/2}$$

Este potencial se aplica a todas las celdas, siendo mayor cuanto mayor sea la distancia a la celda destino, y siendo nulo en esta celda. La constante *k* sería equivalente al valor de la carga de la partícula destino. Este

valor en el código se ha establecido como 40.

Para los obstáculos, el potencial generado será de repulsión, por lo que debe ser inversamente proporcional a la distancia al obstáculo. La contante *k* es equivalente a la carga de los obstáculos. Este valor en el código se ha establecido como 120.

$$V(r) = k/r$$

Si se aplica esta fórmula, el potencial de repulsión afectará a todas las celdas del mapa. Sin embargo, esto no tiene sentido, pues si el obstáculo se encuentra muy lejos del robot, no es lógico que su movimiento se vea afectado por tal obstáculo. Se define entonces un radio de influencia de obstáculos. Si la distancia de una celda a la celda del obstáculo es menor a este radio, se aplica la fórmula anterior, pero si es mayor el potencial será nulo. La fórmula del potencial queda modificada.

$$V(r) = k * \frac{radioInfluencia - r}{r}$$

Con esto ya estaría finalizado el algoritmo APF. Pero si se deja así, surgen varios problemas y situaciones que no puede resolver. El principal problema es la aparición de los llamados 'mínimos locales'. Esto ocurre cuando el potencial total es nulo, que debería ocurrir solo para la celda objetivo, porque sino la trayectoria generada acabaría en esta celda con potencial nulo en lugar de en la celda objetivo. Hay diversas situaciones que pueden provocar un mínimo local, las más típicas se muestran a continuación.

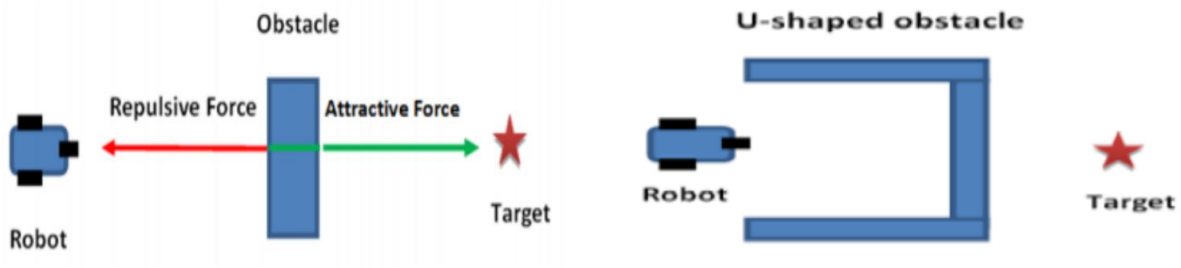


Figura 5-9. Representaciones de situaciones con mínimo local. Obtenida de [24].

Existen diversas soluciones para solventar el problema de mínimos locales, pero en este trabajo se ha optado por desarrollar una solución propia. Posteriormente, cuando se realicen diversas pruebas, tanto en simulación como en el robot ROSBOT 2.0, se comprobará la efectividad de tal método.

El método desarrollado es iterativo y consiste en poder bloquear cada celda del mapa. Al principio, todas las celdas se encuentran desbloqueadas. En cada iteración se calcula el potencial de cada celda y el camino resultante. Si el punto final del camino no corresponde con el objetivo, significa que se ha caído en un mínimo local. Se bloquea la celda final del camino, por lo que ya no se tendrá en cuenta como posible punto del camino. Es como si esa celda ya no existiera. Tras bloquear una celda se vuelve a buscar un camino desde la celda inicial a la destino.

El bucle se puede parar por dos motivos. Primero, si la celda final del camino corresponde a la celda destino, lo que indica que se ha encontrado un camino válido. Segundo, si las ocho celdas vecinas de la celda inicial están bloqueadas. Si esto ocurre, no se habrá podido evitar correctamente el mínimo local.

Esta modificación no aumenta demasiado el coste computacional del algoritmo, pues lo más costoso es calcular el potencial de cada celda, y esa operación solo se realiza una vez al principio. Lo que calcula en cada iteración es el camino siguiendo la dirección de mínimo potencial.

Figura 5-10. Representación de celdas bloqueadas tras no encontrar un camino válido.

La figura se ha obtenido en el simulador desarrollado, cuyo funcionamiento se explica con detalle en el próximo apartado.

Otro problema que surge con APF es que, si el punto objetivo se encuentra muy cerca de un obstáculo, no se podrá llegar hasta él porque el potencial de repulsión será mayor que el de atracción. Para solventar esto, se ha realizado la siguiente modificación: cuando el objetivo está muy cerca de un obstáculo no se tiene en cuenta el potencial repulsivo. De esta forma, el robot podrá llegar hasta el objetivo.

5.2.1.2 Código desarrollado

Con el fin de ahorrar tiempo en depuración y poder realizar numerosas pruebas en distintos escenarios con rapidez, se ha desarrollado un pequeño simulador en C++ que implementa el algoritmo en cuestión.

Tiene un mapa de 30x30 celdas (al igual que el mapa local que genera el robot) almacenado en una matriz. En este caso la resolución será de 1m/celda, a diferencia del robot que es de 0.1m/celda. Pero esto no importa, pues no afecta al tiempo de computación, simplemente a la longitud del camino obtenida.

Las celdas a 0 serán espacio libre, las que estén a 1 serán obstáculos, el número 2 es la celda inicial y el 3 la destino. Estos valores se deben cambiar en el código, por lo que luego habrá que compilar para ver los cambios.

```
//map to test
int map[sizeX][sizeY] = {
  // 0  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29
    {0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0},  //0
    {0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0},  //1
    {0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 2, 0, 0},  //2
    {0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0},  //3
    {0, 0, 0, 1, 1, 1, 1, 1, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0},  //4
    {0, 0, 0, 1, 1, 1, 1, 1, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 1, 0, 0, 0, 0},  //5
    {0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 1, 1, 1, 0, 0},  //6
    {0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 1, 1, 1, 0, 0},  //7
    {0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 1, 1, 1, 0, 0},  //8
    {0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 1, 0, 0, 0, 0},  //9
    {0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0},  //10
    {0, 0, 0, 0, 0, 1, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0},  //11
    {0, 0, 0, 0, 0, 1, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0},  //12
    {0, 0, 0, 0, 0, 1, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0},  //13
    {0, 0, 0, 0, 0, 1, 1, 1, 0, 0, 0, 0, 0, 0, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0, 0, 0},  //14
    {0, 0, 0, 0, 0, 1, 1, 1, 0, 0, 0, 0, 0, 0, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0, 0, 0},  //15
    {0, 0, 0, 0, 0, 1, 1, 1, 0, 0, 0, 0, 0, 0, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0, 0, 0},  //16
    {0, 0, 0, 0, 0, 1, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0},  //17
    {0, 0, 0, 0, 0, 1, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0},  //18
    {0, 0, 0, 0, 0, 1, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0},  //19
    {0, 0, 0, 0, 0, 1, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0},  //20
    {0, 0, 0, 0, 0, 1, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0},  //21
    {0, 0, 0, 0, 0, 1, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0},  //22
    {0, 0, 0, 0, 0, 1, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0},  //23
    {0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0},  //24
    {0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0},  //25
    {0, 3, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0},  //26
    {0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0},  //27
    {0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0},  //28
    {0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0}}; //29
```

Figura 5-11. Matriz de enteros que almacena el mapa y los puntos de origen (arriba a la derecha) y destino (abajo a la izquierda) deseados.

El primer mapa elegido para realizar pruebas es el de la imagen superior, que contiene obstáculos de distintas formas, así como en diferentes orientaciones.

El archivo muestra información útil en el terminal en que se ejecuta. Si ha sido posible encontrar un camino, muestra el mapa con dicho camino junto con su longitud y el tiempo en microsegundos que llevó encontrarlo. Si no se ha encontrado un camino válido, se muestra el mapa con las celdas bloqueadas y el tiempo transcurrido buscando el camino.

A la hora de mostrar el mapa con los resultados, la simbología de 1 y 0 sigue igual, pero el punto inicial se muestra con una ‘X’, el punto final con una ‘Y’ y los puntos del camino encontrado con el signo ‘+‘. Esto es porque estos símbolos son más fáciles de distinguir en el terminal que simplemente números. Para buscar celdas adyacentes se utiliza un criterio de 8 conectividad.

Figura 5-12. Ejemplo de resultado tras ejecutar el simulador con el algoritmo APF.

Otro objetivo principal de este simulador es también poder comparar con otros algoritmos fácilmente, como en el caso de Lazy Theta*, aislando el uso de potencia computacional, pues toda se centrará en el algoritmo y no dependerá de ROS o plugins o sensores externos. Así, los datos obtenidos aquí serán fiables a la hora de comparar los algoritmos entre sí.

Por otro lado, el algoritmo se ha implementado en ROS creando un paquete llamado ‘apf_local_planner’ que contiene dos archivos principales.

- apf_local_planner.h
- apf_local_planner.cpp

El archivo de cabecera contiene las definiciones de variables y funciones que se usarán en el archivo CPP. En este se define el *plugin* del paquete ‘nav_core’ que se creará y se implementa el algoritmo.

5.2.2 Lazy Theta*

5.2.2.1 Descripción del algoritmo

El segundo algoritmo que se ha desarrollado de evitación de obstáculos es el llamado 'Lazy Theta*'. Esta técnica de planificación está basada en el Theta*, que a su vez está basada en el ya conocido algoritmo A* que implementa algunas mejoras en ciertos casos concretos.

Como se sabe, el algoritmo A* utiliza regiones mapeadas para encontrar posibles caminos entre puntos. El problema es que esta técnica encuentra caminos a través de las líneas trazadas por los ejes de las regiones del mapa, restricción que hace que no encuentre el camino válido más corto. Es por este motivo por lo que se dice que el algoritmo Theta*, que soluciona este problema, es un "any-angle path planning" o "planificador para cualquier ángulo" [25].

Si hablamos de la implementación de este algoritmo, en realidad el cambio en el ángulo del camino se traduce en la eliminación de una restricción que existe en el A*. Y es que el algoritmo Theta* permite seleccionar como siguiente punto a cualquier vértice del mapa, mientras que el A* obliga a que este siguiente punto sea un vecino del punto actual donde se encuentre el robot.

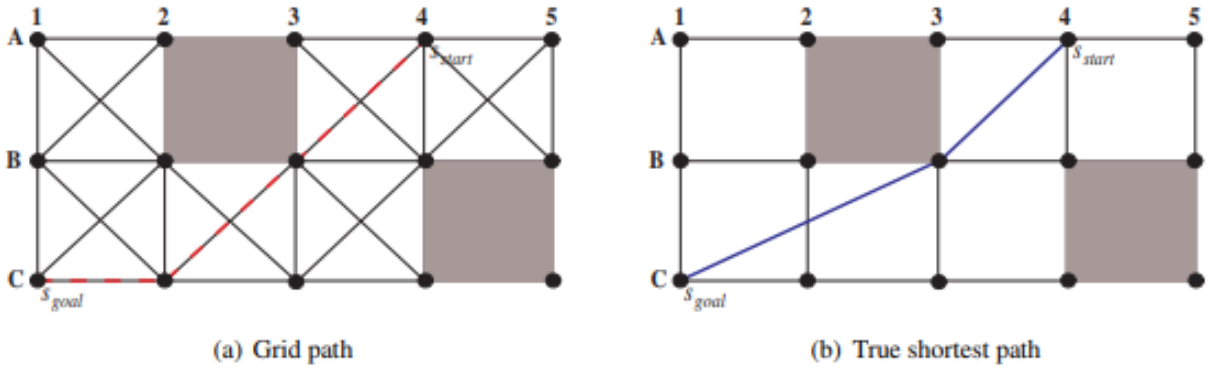


Figura 5-13. Comparación de camino con ángulo restringido (izquierda) y camino en cualquier ángulo (derecha). Obtenida de [25].

Sin embargo, el algoritmo Theta* (así como muchas de sus variantes) también presenta diversos inconvenientes. Para empezar, la estrella de su nombre no hace alusión a la optimización de la implementación o del resultado obtenido, sino a su similitud al algoritmo A*. Además, el coste computacional y tiempo de cálculo de los caminos depende de la variante, pero si comentamos brevemente el Basic Theta*, estos factores crecen linealmente con la complejidad del mapa, esto es, el número de celdas que dicho mapa contenga.

Si hablamos ahora de la variante del algoritmo Theta* que nos ocupa, el Lazy Theta*, este tiene la gran ventaja de ahorrar coste computacional, tal y como se puede intuir a juzgar por su nombre. La idea principal es comprobar menos rutas posibles que en el algoritmo tradicional, de forma que el tiempo de computación bajará notablemente. En concreto, se trata de revisar tan solo una línea de visión por cada vértice dado por las regiones presentes en el mapa en cuestión.

Lo más sorprendente de esta variante es que no se incrementa considerablemente la longitud del camino encontrado, por lo que la mejora de la gestión de recursos se realiza casi sin coste alguno. Teniendo en cuenta todos estos factores, es bastante evidente que la relación ventajas/inconvenientes del algoritmo Lazy Theta* es bastante positiva, al menos, si se compara con las otras variantes de este método y teniendo en cuenta el ámbito y espacio de trabajo necesarios en este proyecto.

Se pasará ahora a una explicación en profundidad del algoritmo Lazy Theta*. Primero se mostrará el desarrollo de este en pseudocódigo.

Algorithm 3 Lazy Theta*

```
1:  procedure MAIN()
2:      open := closed := φ;
3:      g(s_start) := 0;
4:      parent(s_start) := s_start;
5:      open.Insert(s_start, g(s_start) + h(s_start));
6:      while open ≠ φ do
7:          s := open.Pop();
8:          SetVertex(s);
9:          if s = s_goal then return "path found";
10:         closed := closed ∪ {s};
11:         for each s' ∈ nghbr(s) do
12:             if s' ∉ closed then
13:                 if s' ∉ open then
14:                     g(s') := ∞;
15:                     parent(s') := NULL;
16:                 UpdateVertex(s, s');
        return "no path found";
17:
18: procedure UPDATEVERTEX(S,S')
19:     g_old := g(s');
20:     ComputeCost(s, s');
21:     if g(s') < g_old then
22:         if s' ∈ open then
23:             open.Remove(s');
24:         open.Insert(s', g(s') + h(s'));
25:
26: procedure COMPUTECOST(S,S')
27:     /* Path 2 */
28:     if g(parent(s)) + c(parent(s), s') < g(s') then
29:         parent(s') := parent(s);
30:         g(s') := g(parent(s)) + c(parent(s), s');
31:
32: procedure SETVERTEX(S)
33:     if !lineOfSight(parent(s), s) then
34:         /* Path 1 */
35:         parent(s) := argmin_{s'∈nghbr(s)∩closed}(g(s') + c(s', s));
36:         g(s) := min_{s'∈nghbr(s)∩closed}(g(s') + c(s', s));
```

Figura 5-14. Pseudocódigo del algoritmo Lazy Theta*. Las partes en rojo son las diferencias con el algoritmo Theta*, pero se ignorará el código de colores. Obtenido de [14].

Al basarse indirectamente Lazy Theta* en A*, también se crearán una lista abierta y otra cerrada, en las que se irán añadiendo las celdas que componen el mapa del entorno. En la abierta irán las celdas que se han seleccionado para analizar y en la cerrada las que ya se han analizado.

El inicio es igual, pues se comienza con ambas listas vacías. La 'g' de cada celda es la distancia que hay que recorrer desde la celda inicial hasta esa celda, por lo que tiene en cuenta cada celda padre por la que hay que pasar hasta llegar a ella. La 'h' es la distancia Euclídea hasta la celda destino. La función 'f = g+h' será la función a minimizar a la hora de buscar el mejor camino posible.

La celda padre de otra celda es la que hay que visitar antes que ella. Una vez ha finalizado la búsqueda del camino, se mirará la celda padre de la final y se añadirá al camino, y así sucesivamente con cada celda padre hasta llegar a la inicial. Tras invertir el camino para que el orden sea correcto se tiene la trayectoria final a seguir, libre de obstáculos.

Primero se añade la celda inicial a la lista abierta, y se selecciona como celda padre ella misma. Mientras queden celdas en la lista abierta significa que aún podría encontrarse un camino, pero si esta lista queda vacía significa que ya se han analizado todas las celdas posibles y no se ha encontrado un camino válido, por lo que el algoritmo acaba.

Cada iteración consiste en los siguientes pasos. Primero, se mueve la celda con menor 'f' de la lista abierta a la lista cerrada. La función 'setVertex' supone que hay visibilidad entre esa celda y su padre. Si no la hay, se cogerá la celda vecina que ya esté en la lista cerrada que tenga menor 'f' y se adjudicará como nuevo padre, a la vez que se actualiza su 'g'. Aquí radica la mejora en tiempo de computación del algoritmo, pues supone a priori que ya hay visibilidad entre una celda y su celda padre, por lo que si es así no tiene que asignar valores pues ya se hizo antes.

Como algoritmo de visibilidad se usará el algoritmo de Bresenham, que recibe dos puntos en 2D y devuelve todas las celdas por las que pasa el segmento que los une [26]. De esta manera, si ninguno de esos puntos es un obstáculo, habrá visibilidad entre ambos puntos, en caso contrario no. La potencia de este algoritmo reside en que no utiliza operaciones con punto flotante, solo sumas, restas y multiplicaciones de enteros, lo que hace eficiente computacionalmente.

Cuando acaba la función 'setVertex', se comprueba si la celda actual es la objetivo, en cuyo caso se ha encontrado un camino y se procede a calcularlo con lo explicado anteriormente, viendo las celdas padre. Si no se ha llegado aún, se calculan los vecinos de la celda en cuestión y se analizan, definiendo como celda padre la celda padre de la actual (aquí se realiza la suposición de que hay visibilidad entre ellas) y la g como la g de la celda padre más la distancia de esta a la celda vecina en cuestión.

Si la celda vecina no estaba en la lista abierta ni en la cerrada, se añade a la abierta. Si estaba en alguna de las listas (solo puede estar en una de ellas), se comprueba si la f actual es menor que la que tiene y si es así se actualizan los valores de g y de su celda padre.

Así quedaría implementado el método LZS completamente. Este método obtiene buenos resultados en cuanto a distancia del camino se refiere, pero a costa de aumentar el coste computacional. Tras la explicación detallada de ambos algoritmos se puede ver que el coste computacional de LZS es mayor que el de APF.

5.2.2.2 Código desarrollado

Al igual que en caso del APF, para LZS (Lazy Theta Star) también se ha desarrollado un simulador siguiendo el mismo esquema. El mapa lo recibe y se cambia de la misma forma y la información que se muestra en el terminal es la misma (salvo que ahora los puntos del camino se muestran con el signo '-' en vez de con el '+'), tan solo cambia la forma de buscar un camino entre los dos puntos dados.

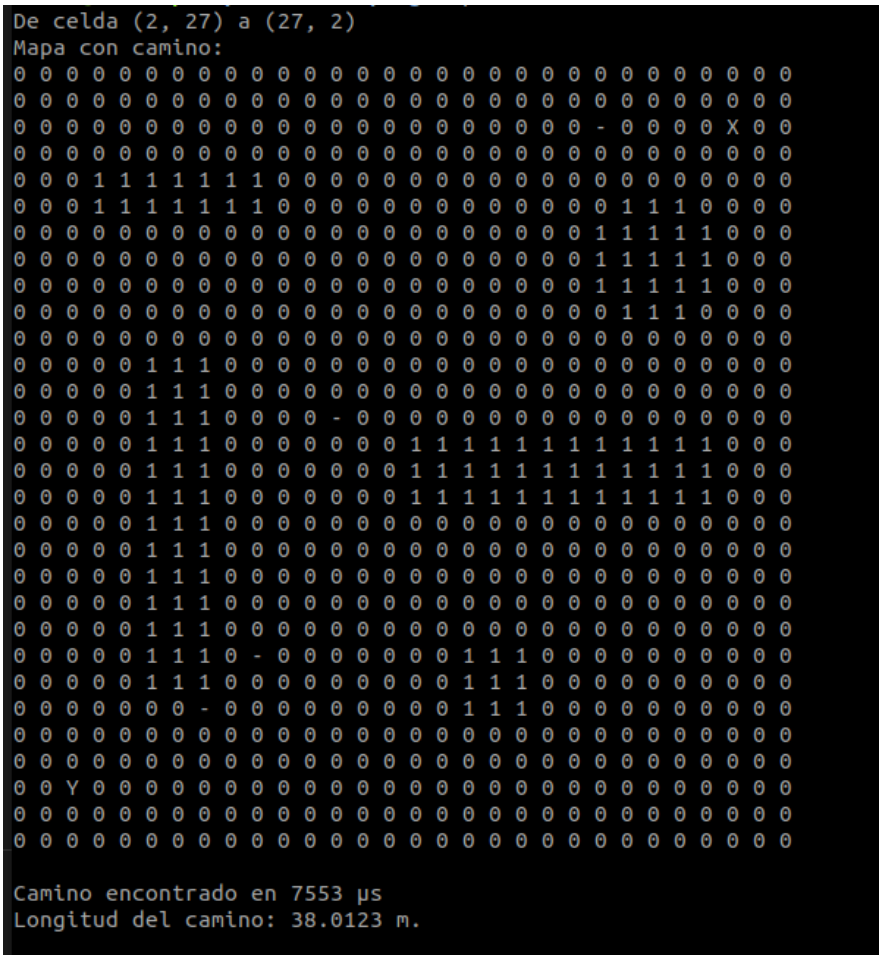


Figura 5-15. Ejemplo de resultado tras ejecutar el simulador con el algoritmo LZS.

En el apartado siguiente se realizarán diversas pruebas y comparaciones de resultados en este simulador con ambos algoritmos.

Al igual que con APF, el algoritmo LZS se ha implementado en ROS creando un paquete, ahora llamado 'lazy_theta_star_local_planner' que contiene dos archivos principales.

- lazy_theta_star _local_planner.h
- lazy_theta_star _local_planner.cpp

El archivo de cabecera contiene las definiciones de variables y funciones que se usarán en el archivo CPP. En este se define el *plugin* del paquete 'nav_core' que se creará y se implementa el algoritmo.

6 PRUEBAS DE EVITACIÓN DE COLISIONES

A continuación, se muestran capturas de pantalla y fotos de varias pruebas realizadas con el método de evitación de colisiones desarrollado en el capítulo 5.1.

En la primera prueba se envió como destino un punto situado a 1 m hacia delante en la orientación del robot, por lo que solo tendrá que avanzar en línea recta para alcanzarlo. En RVIZ de muestra así el camino seguido:

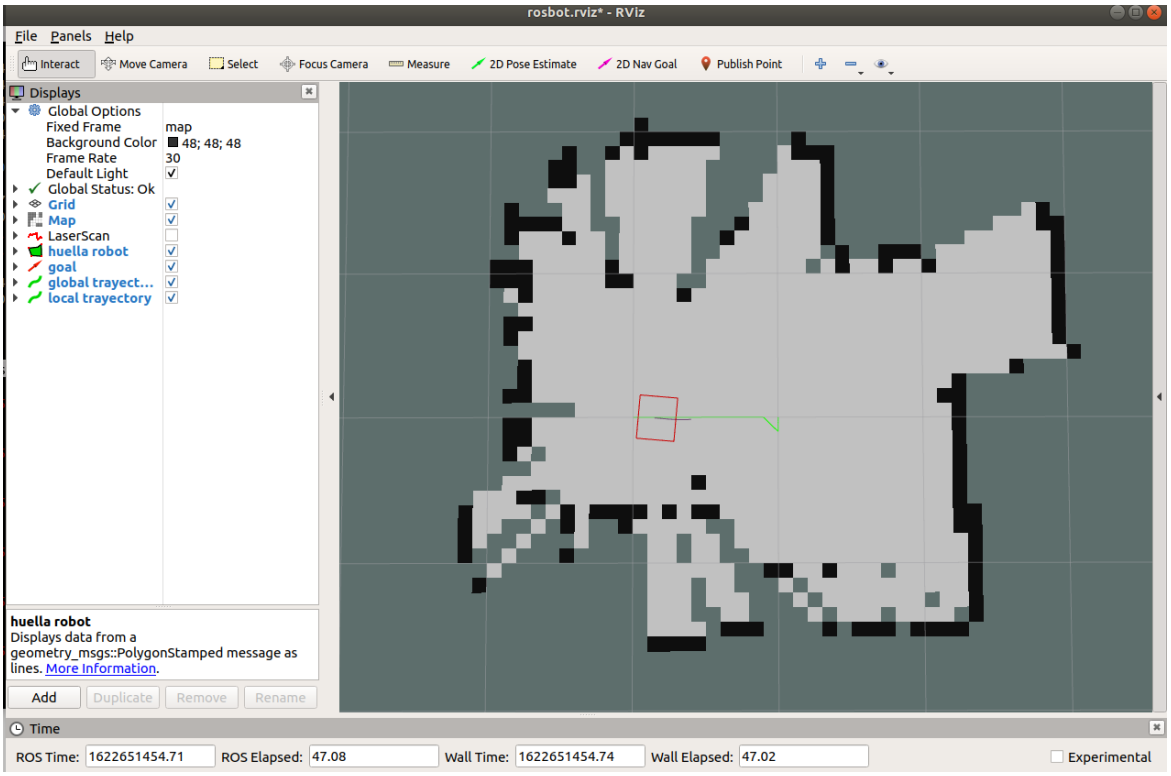


Figura 6-1. Posición inicial, justo al iniciar el desplazamiento. En verde se ve el plan global.

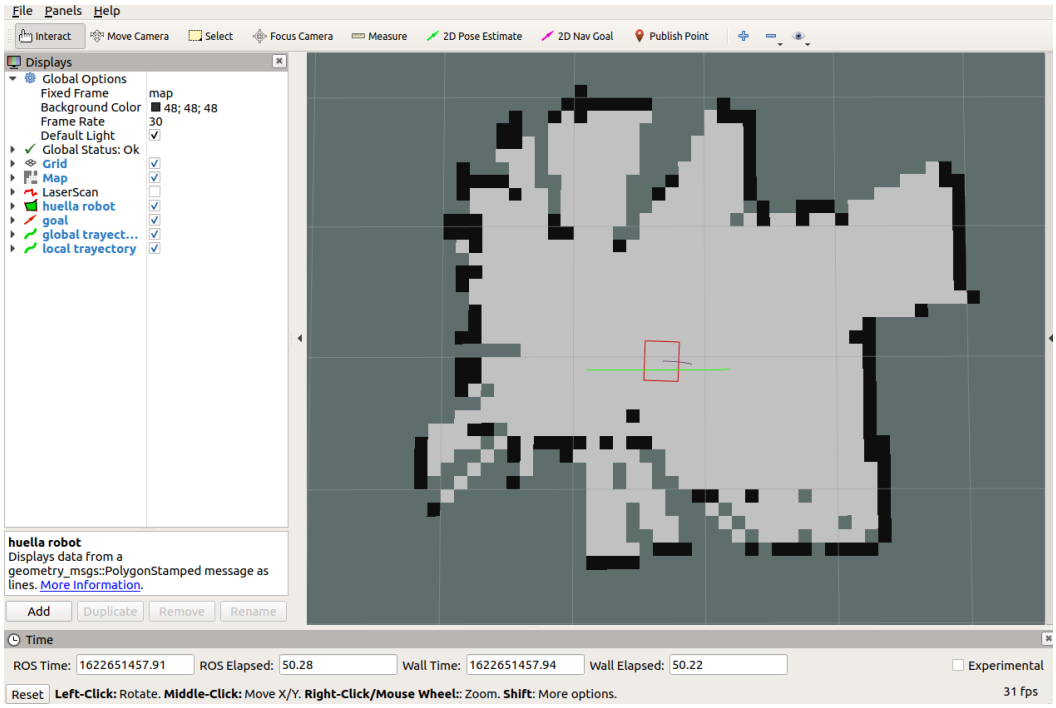


Figura 6-3. Posición del robot al pararse por haber encontrado un obstáculo delante. En el mapa no aparece ningún obstáculo pues el LIDAR no lo detecta, pero los sensores infrarrojos sí.

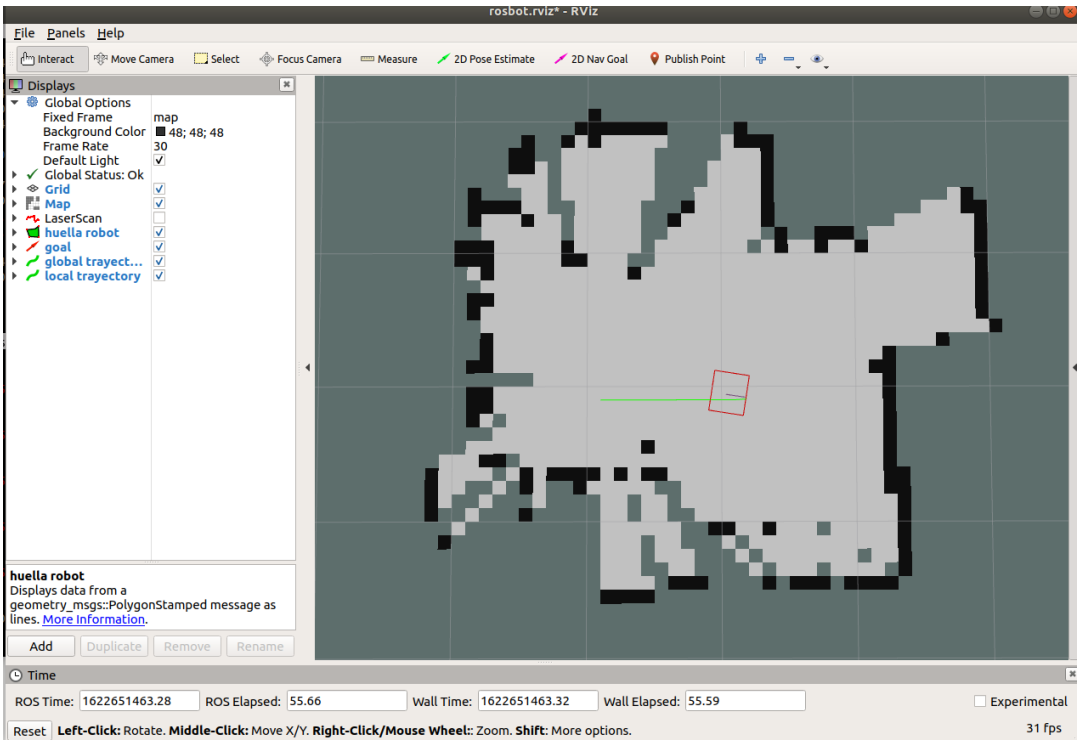


Figura 6-2. Posición final del robot, alcanzando correctamente el objetivo anterior tras haberse eliminado el obstáculo.

A continuación, se muestran varias fotos del robot y el obstáculo (un zapato) de la misma prueba.

Figura 6-4. Posición inicial del robot y el obstáculo.

Figura 6-5. El robot avanzando hacia el objetivo de 1m.

Figura 6-7. Robot detenido al estar demasiado cerca del obstáculo. Se cancela el objetivo y se para.

Figura 6-6. Se elimina el obstáculo manualmente y se deja el camino al destino libre.

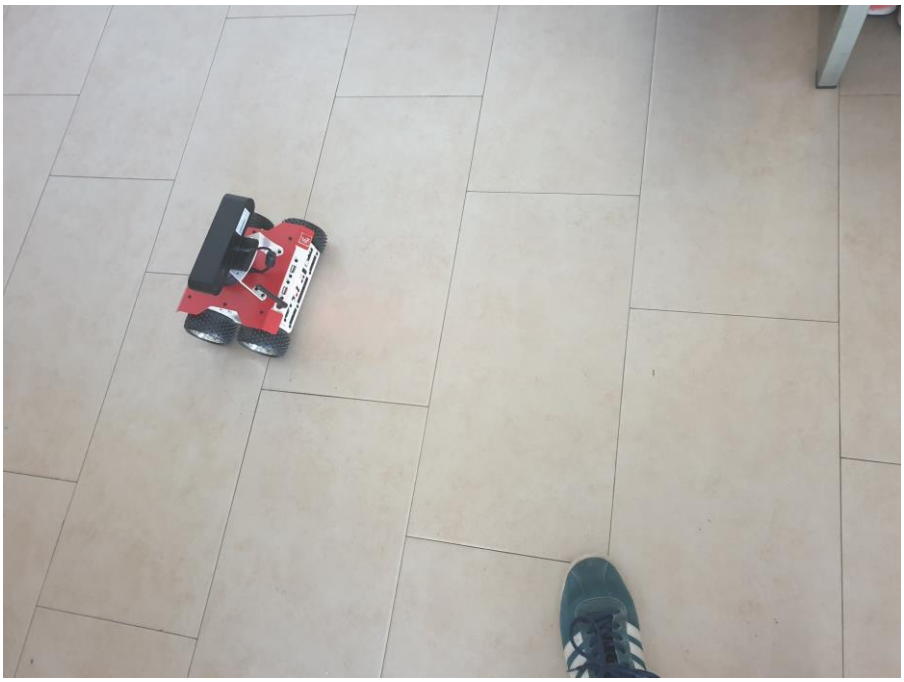


Figura 6-8. El robot retoma el objetivo original y lo alcanza correctamente.

En este enlace se puede ver un vídeo de la prueba realizada:

Vídeo obstáculo estático

Se realizó otra prueba, esta vez con un obstáculo móvil que aparece por un corto período de tiempo delante del robot, y este evitó la colisión como se esperaba. En enlace para ver el vídeo es el siguiente:

Vídeo obstáculo móvil

Se realizó otra prueba, similar a la primera con un obstáculo estático también. Sin embargo, cómo se aprecia en el vídeo, al quitar el obstáculo manualmente, el robot realiza una rotación del 360 grados antes de volver a su objetivo. Esto sucede porque al quitar el obstáculo acerco demasiado la mano al robot, por lo que el LIDAR la detecta y, como está en la parte delantera, como el objetivo, no puede llegar a él porque considera que hay un obstáculo. Entonces procede a realizar el primer comportamiento de recuperación, que consiste en hacer ese giro de 360 grados con el fin de eliminar obstáculos móviles en un área cercana. Al terminar la rotación ya no detecta ningún obstáculo delante y procede a ir al objetivo.

Vídeo obstáculo estático 2 (comportamiento de recuperación)

7 PRUEBAS Y COMPARACIONES DE ALGORITMOS DE EVITACIÓN DE OBSTÁCULOS

Una vez se han desarrollado ambos algoritmos, se procede a comprobar su efectividad mediante diversas pruebas en distintas situaciones.

7.1 Pruebas en simulación

A continuación, se mostrarán y se analizarán los resultados obtenidos en el simulador desarrollado con ambos métodos de evitación de obstáculos. Con el fin de que las comparaciones sean más fiables, se ha establecido el mismo radio de inflación de obstáculos en LZS que de esfera de influencia en APF, fijado en 2.5m. Eso correspondería a 25cm si el mapa tuviera la resolución del que aporta el robot, y esos 25cm son suficientes para que este no colisione con las paredes, pues mide 24cm de ancho (sería suficiente con un radio de protección de 13cm pero se da mayor para tener un margen de seguridad).

7.1.1 Prueba 1. Camino largo

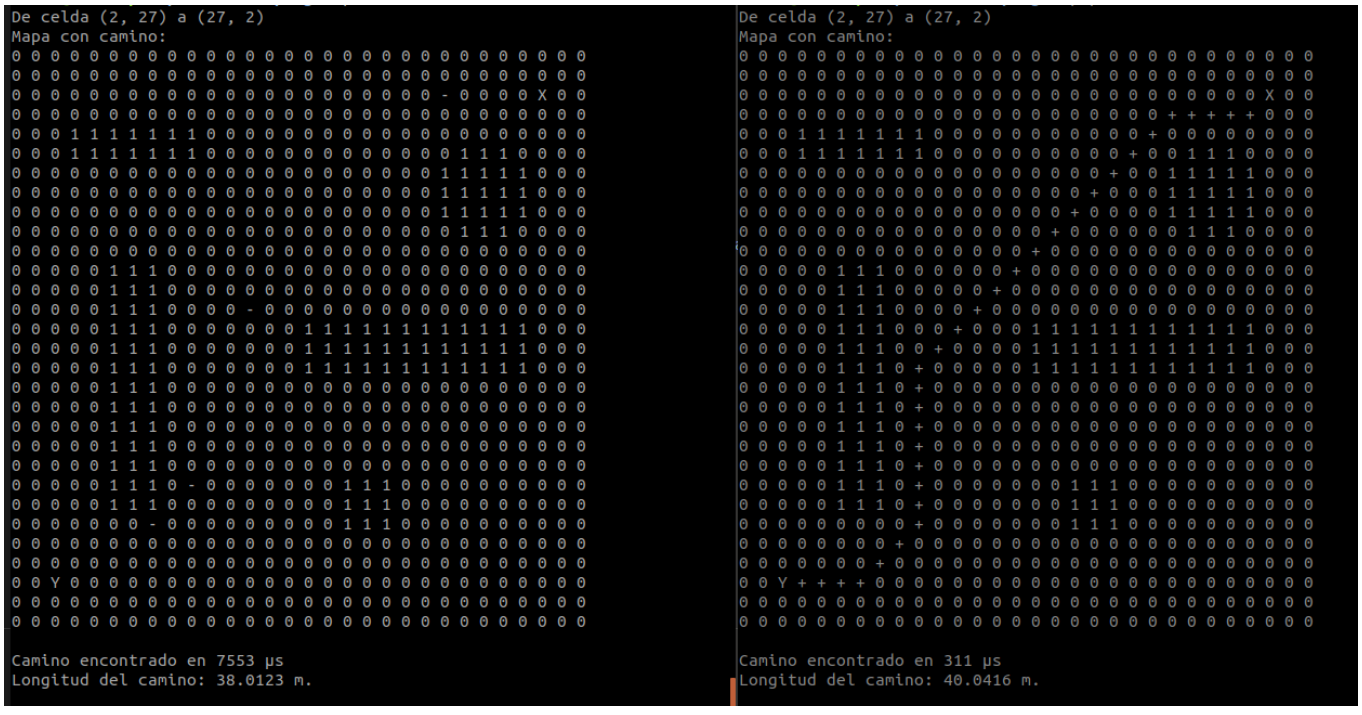


Figura 7-1. Prueba en simulación de un camino de gran longitud.

Esta primera prueba corresponde a un camino largo en el mapa disponible, pues el robot debe ir de una esquina a la contraria. En este caso ambos métodos han encontrado un camino válido.

Se observa cómo LZS tarda bastante más en encontrar el camino, ¡casi 25 veces más!, aunque es más

corto, con 2.02m menos de recorrido. En cuanto a la evitación de obstáculos, ambos métodos los evitan correctamente, aunque en el último punto del camino de LZS el robot pasaría muy cerca de un obstáculo. Esto se debe a que la implementación del método no es perfecta, se ha intentado corregir ese fallo, pero solo ocurre a veces y en la parte inferior de los obstáculos. En principio no debería ocurrir esto, pues el mapa inflado tiene un radio de 2.5m, que debería incluir esas celdas.

En este caso sería preferible usar el método APF, pues encuentra el camino en menor tiempo.

7.1.2 Prueba 2. Mínimo local

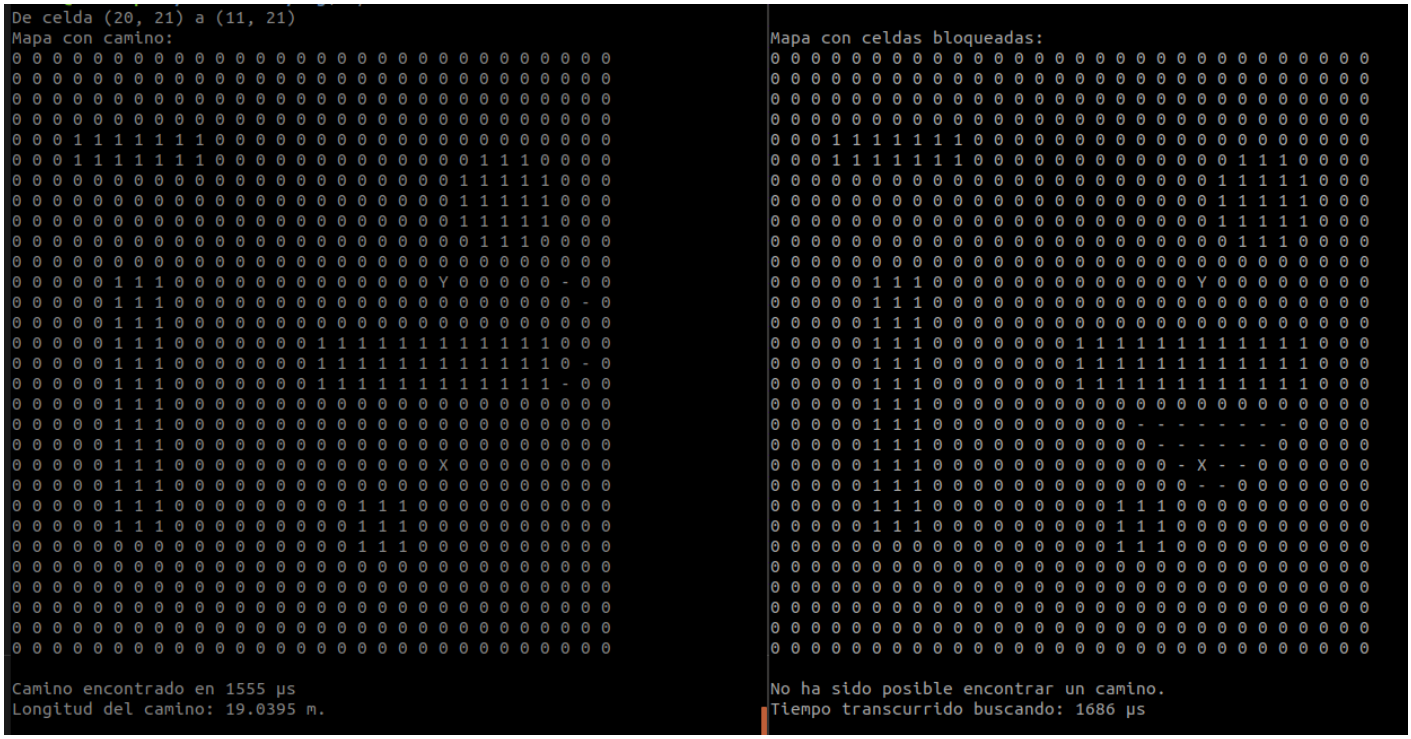


Figura 7-2. Prueba en simulación de un camino con un mínimo local.

Como ya se ha comentado, el principal problema del algoritmo de campos potenciales son los mínimos locales que se encuentran cuando el potencial de atracción del objetivo es el mismo que el de repulsión de los obstáculos. Para solventar esto se propuso una solución propia, aunque no siempre funciona. En este caso, entre el punto inicial y objetivo hay un obstáculo horizontal grande, lo que causa ese mínimo local.

Se observa cómo LZS encuentra un camino válido en 1.5ms aproximadamente, siendo este casi el mismo tiempo que APF tarda en fallar. Se ven con el símbolo '-' las celdas bloqueadas en APF, y cómo esa búsqueda se expande hasta el ángulo de 45º respecto del punto inicial en dirección al destino.

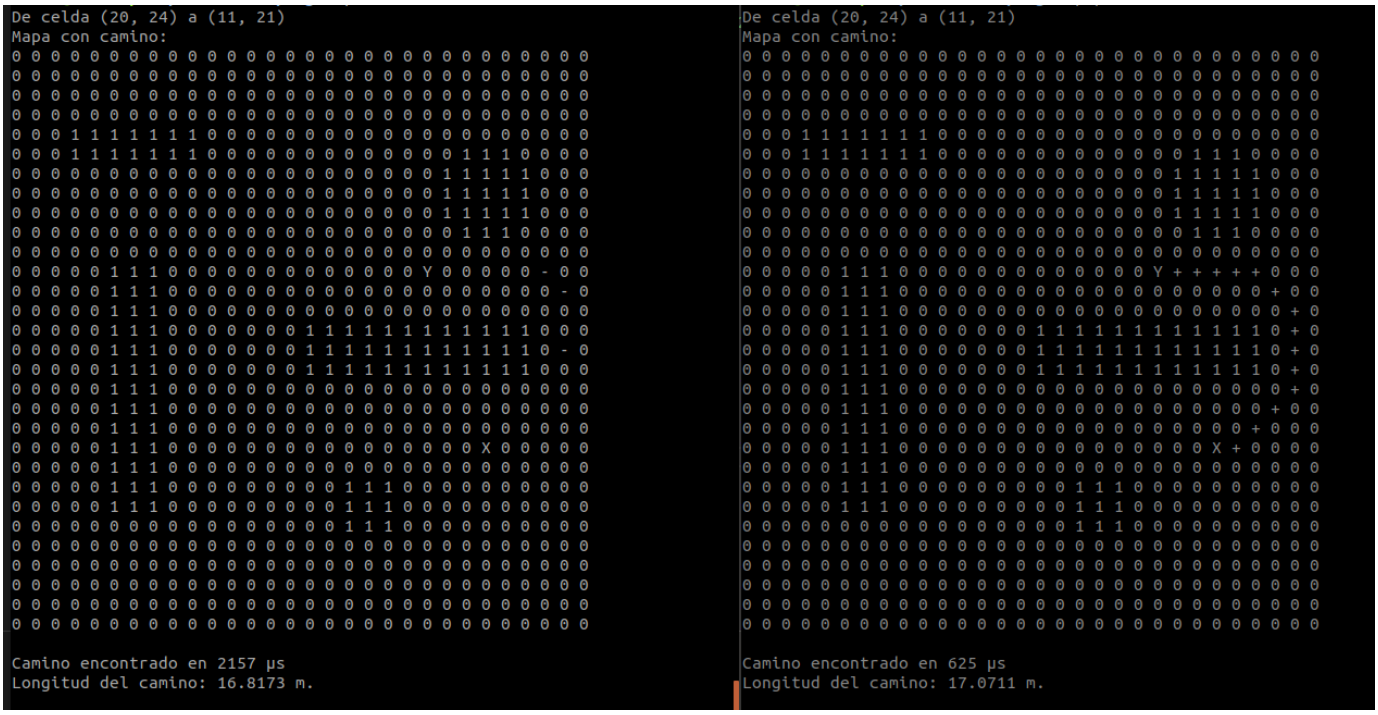


Figura 7-3. Prueba en simulación de un camino con mínimo local resuelto.

Ahora se ha desplazado hacia la derecha 3 celdas el punto inicial, y ya sí encuentra un camino el algoritmo APF. Esto es porque ya en los 45º hacia la derecha se acaba el obstáculo y, por tanto, lo bordea y acaba llegando al destino.

La longitud de los caminos encontrados es prácticamente la misma, con una diferencia de 0.2m apenas, pero el tiempo de computación de LZS sigue siendo mayor, casi 3.5 veces mayor. Ahora sí que ambos métodos encuentran un camino que mantiene siempre una distancia de seguridad con los obstáculos.

7.1.3 Prueba 3. Camino libre de obstáculos en línea recta

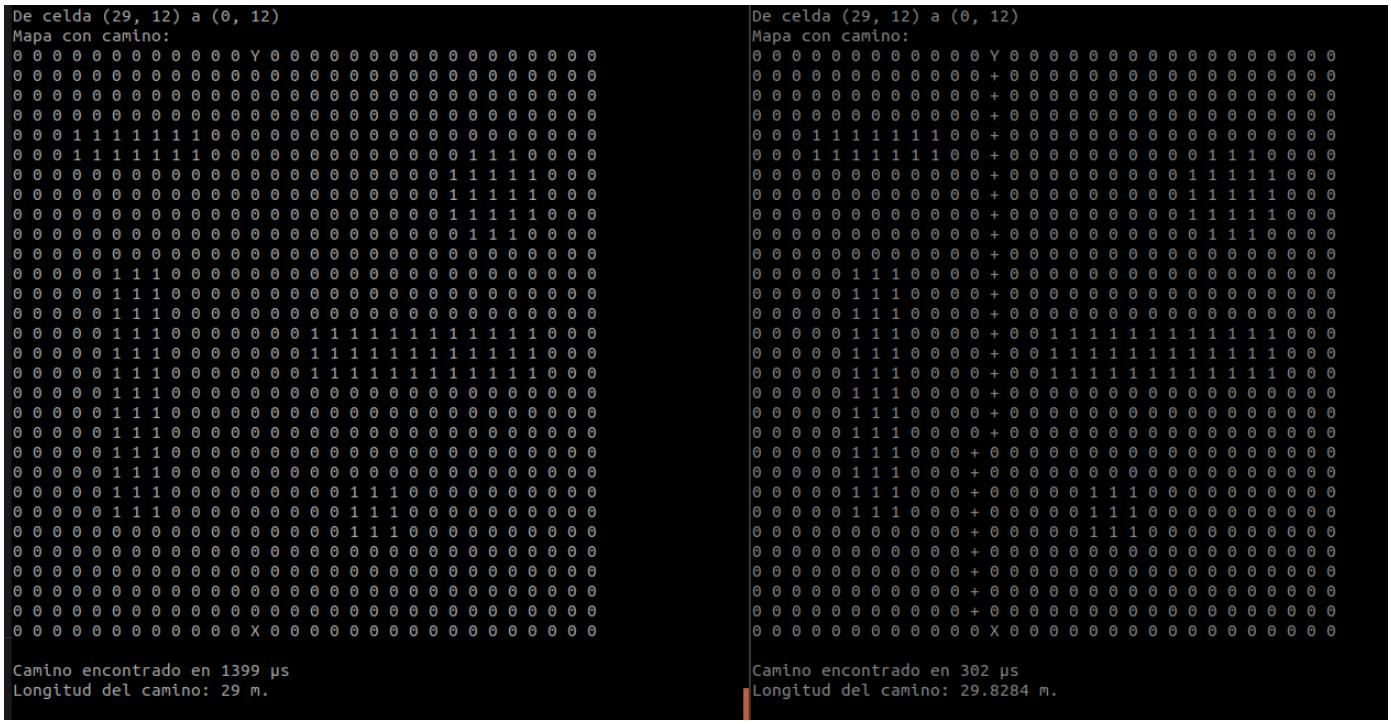


Figura 7-4. Prueba en simulación de un camino en línea recta libre de obstáculos.

En esta prueba se ha querido comparar cuánto tardan los métodos en encontrar un simple camino en línea recta libre de obstáculos. La solución de LZS tiene longitud mínima, 29m, mientras que en APF se desplaza ligeramente hacia la izquierda en lugar de moverse directamente en línea recta. Esto es debido al potencial de repulsión del pequeño obstáculo circular de la derecha.

Sin embargo, si atendemos al tiempo de computación, nuevamente LZS tarda mucho más, del orden de 4.5 veces más.

7.1.4 Prueba 4. Punto destino muy cercano a un obstáculo

En la siguiente prueba, se ha establecido el punto destino muy cerca de un obstáculo. Estos son los resultados.

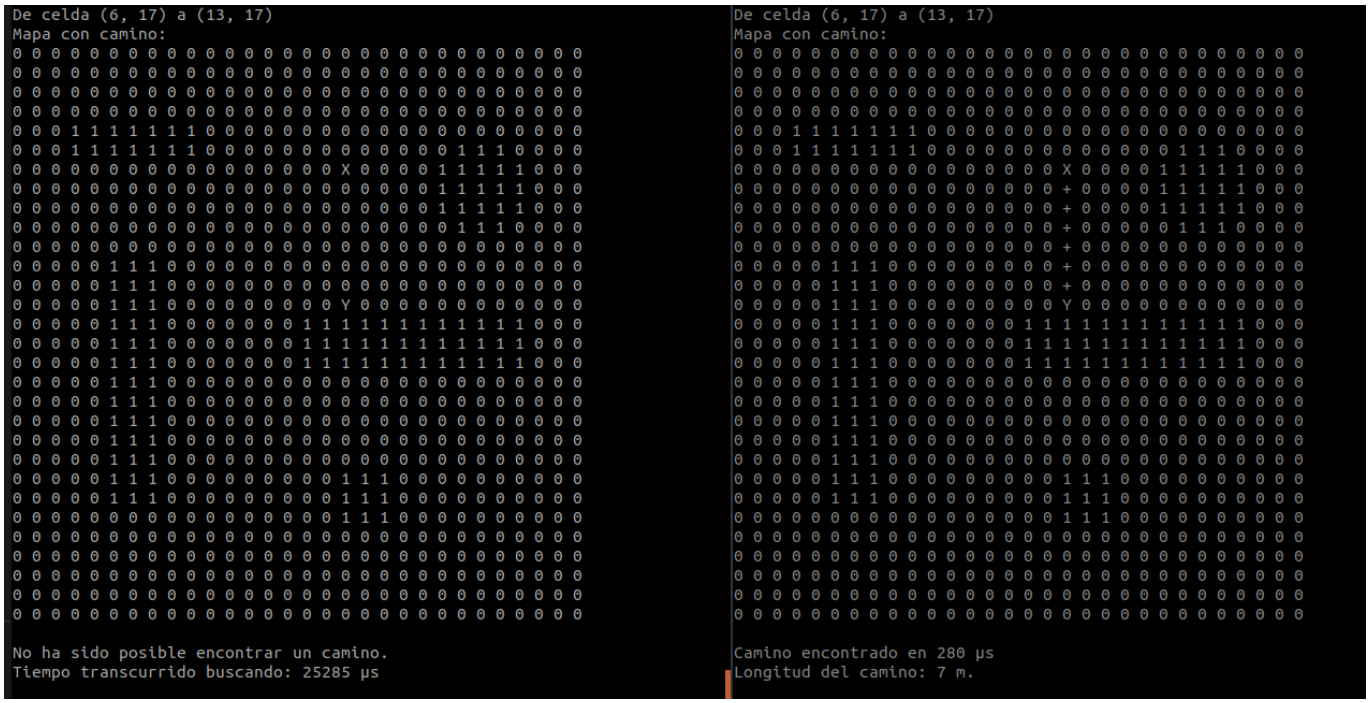


Figura 7-5. Prueba en simulación con el punto destino muy cerca de un obstáculo.

En este caso es LZS el método que no encuentra un camino válido. Esto es porque al inflar el mapa original, el destino quedaría envuelto en celdas de obstáculos, por lo que sería inaccesible. Sin embargo, en APF sí que se encuentra un camino válido, pues se realizó esa modificación del algoritmo original para que funcionara en estos casos. Aunque el camino sea correcto en teoría, quizá el robot acabaría tan cerca del obstáculo que podría colisionar con él. Para que esto funcionara correctamente habría que ajustar muy bien el parámetro de radio de influencia de obstáculos en APF y el de tolerancia objetivo en el control de movimiento.

En cuanto al tiempo de computación, se aprecia cómo LZS tarda más de 25ms en determinar que no hay un camino válido, un espacio de tiempo considerable en un bucle de control en tiempo real.

7.1.5 Prueba 5. Camino de longitud media

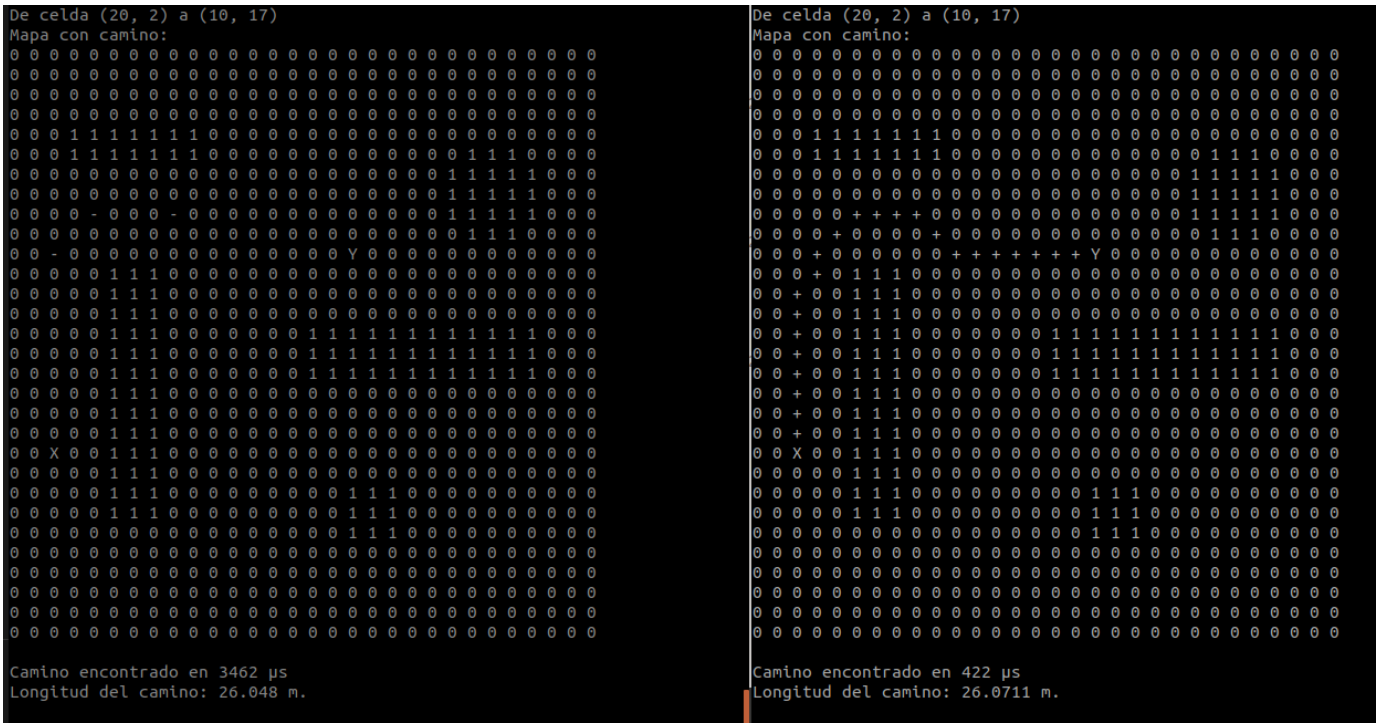


Figura 7-6. Prueba en simulación de un camino de longitud media.

Para esta prueba se ha establecido un camino de longitud media, partiendo de un borde del mapa y queriendo llegar a un punto centrado de este. Ambos métodos encuentran un camino válido y de casi la misma longitud, siendo el de LZS apenas 0.3m más corto. En cuanto al tiempo de simulación, de nuevo LZS tarda bastante más en encontrarlo.

7.1.6 Prueba 6. Mínimo local 2

Ahora se desplaza el punto destino hasta acabar detrás del obstáculo vertical.

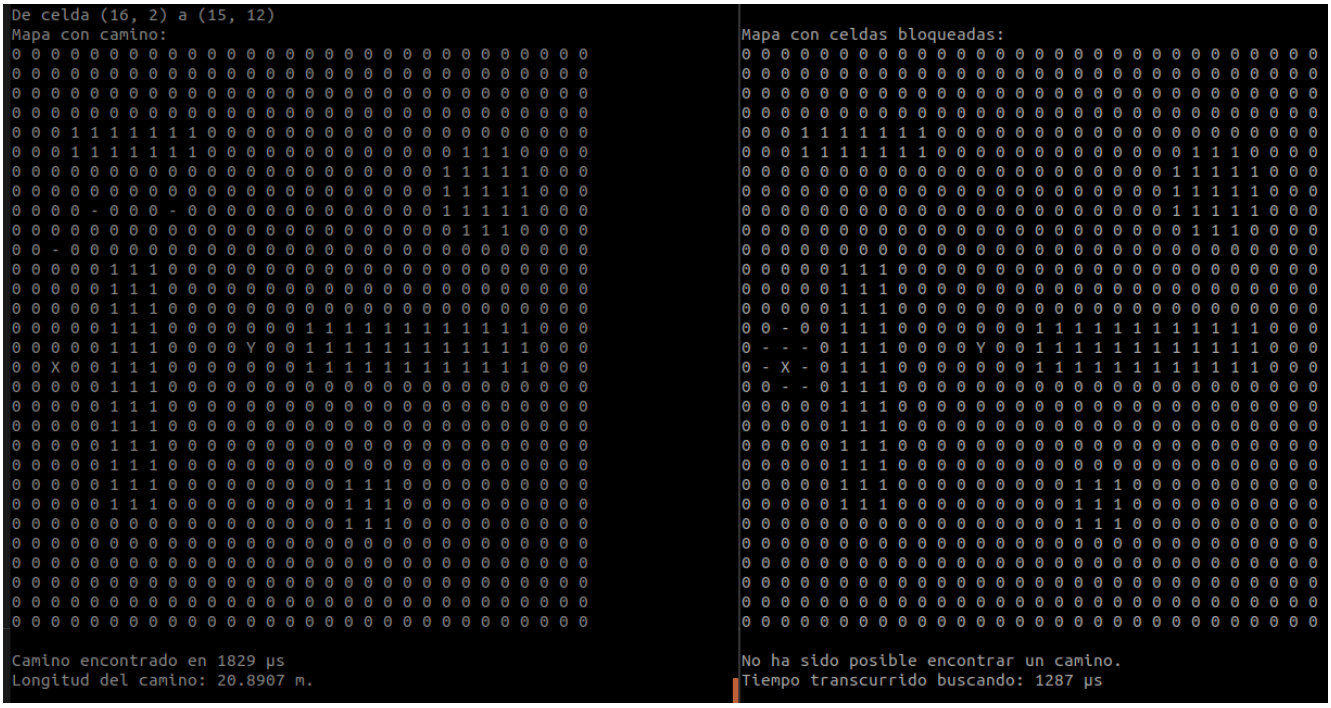


Figura 7-7. Prueba en simulación de un camino de con mínimo local.

Se vuelve a dar un caso de mínimo local con un obstáculo grande, en este caso vertical. Al igual que antes, LZS encuentra un camino, pero APF no, al estar el punto destino directamente detrás del obstáculo.

7.1.7 Prueba 7. Camino largo 2

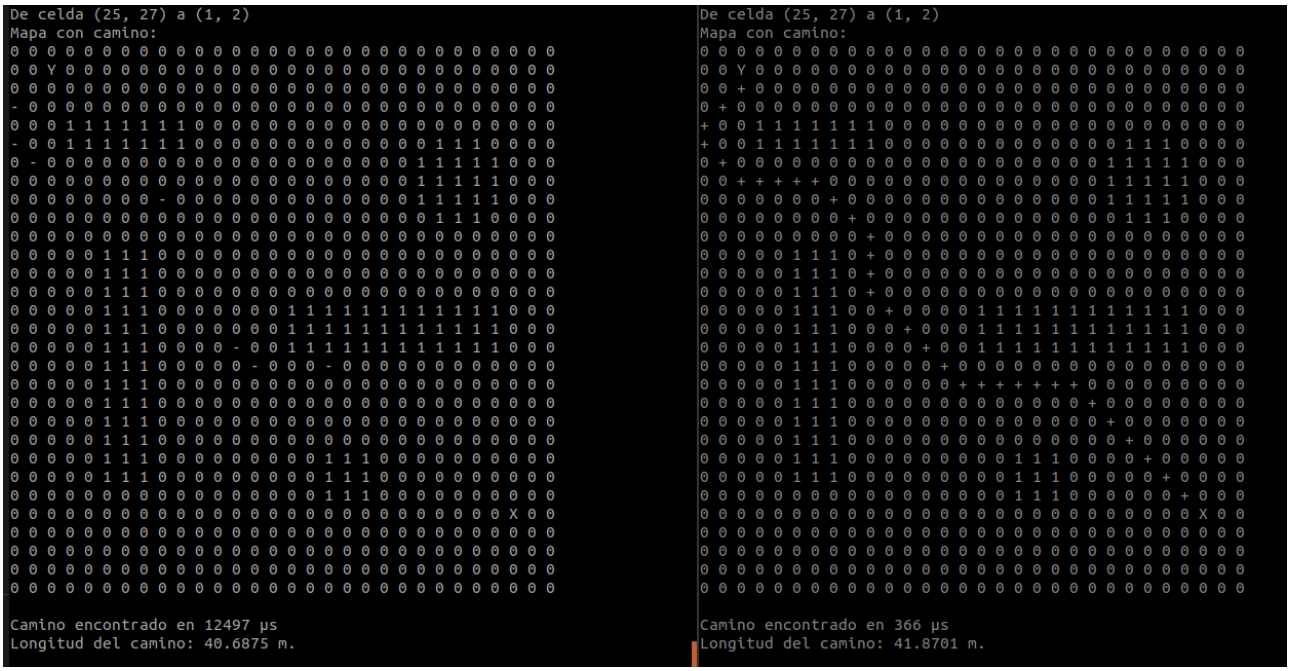


Figura 7-8. Prueba en simulación de un camino largo.

Se vuelven a dar puntos de inicio y destino en esquinas contrarias del mapa. Ambos métodos encuentran caminos válidos, de nuevo LZS con un mayor tiempo de computación, pero con ligeramente menor longitud.

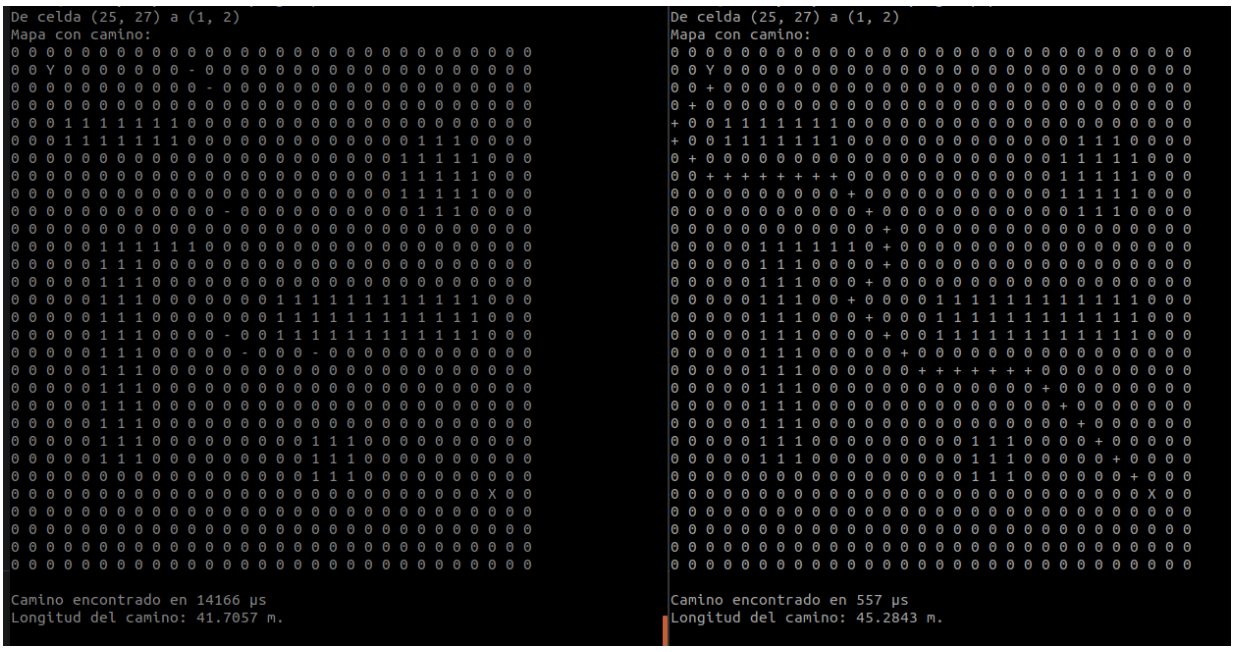


Figura 7-9. Se añade una fila pequeña de obstáculos que desvía el camino encontrado.

Se modifica el mapa y se añade una pequeña fila de obstáculos en la parte central. Esto obliga a desviar la trayectoria, pero ambos métodos nuevamente encuentran una válida. Sin embargo, ahora la diferencia en distancia es mucho mayor, pasando de 1.2m antes a 3.6m ahora, aproximadamente. Aun así, este cambio introduce un mínimo local en APF en la esquina de la fila nueva con el obstáculo vertical de su izquierda, pero el algoritmo lo evita correctamente.

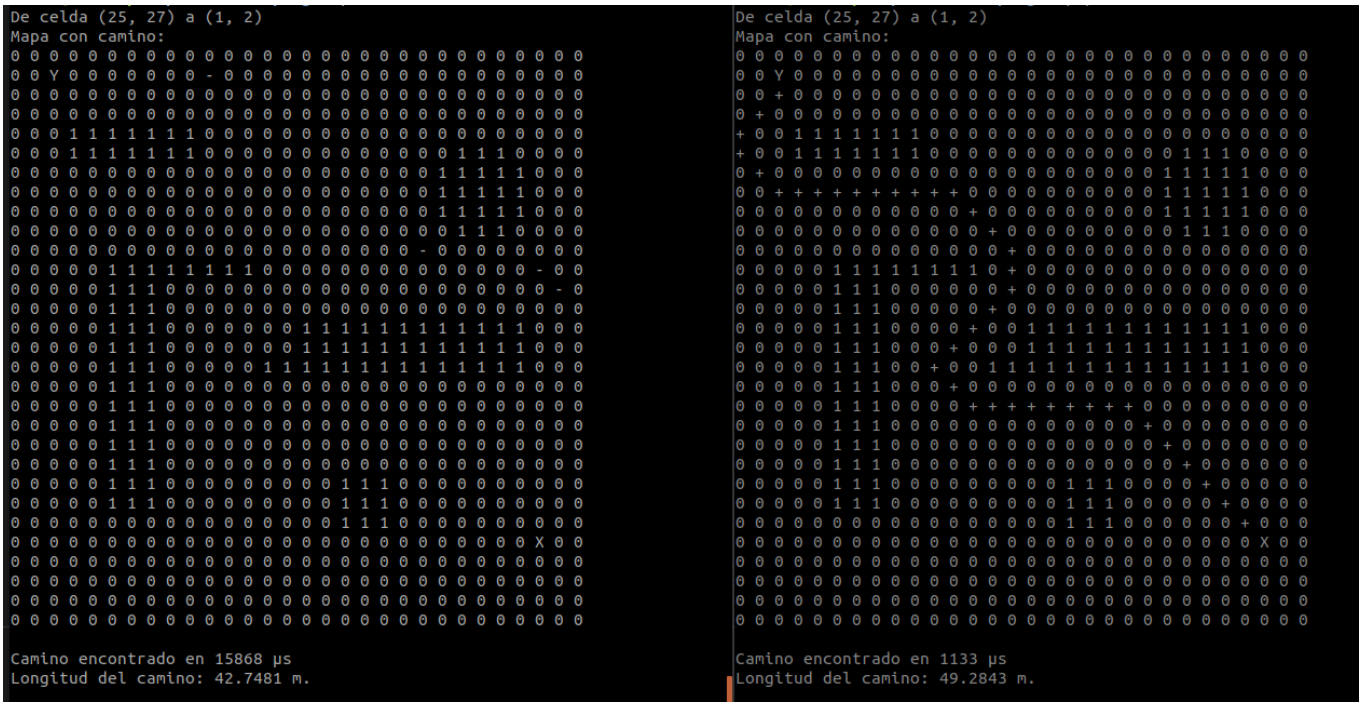


Figura 7-10. Se añade otra fila pequeña de obstáculos.

Se añade una nueva fila pequeña de obstáculos debajo de la anterior que obliga a APF a hacer un recorrido tipo ‘zig-zag’ para sortear los obstáculos. Por otro lado, LZS encuentra ahora otro camino más rápido por el lado derecho del mapa, evitando la longitud extra impuesta por esos nuevos obstáculos. Así, la diferencia de distancias es ahora de 6.5m, bastante notable, además de que el camino hallado por LZS es más seguro en cuanto a que no pasa por el medio de tantos obstáculos.

Aun así, no se puede dejar de lado el tiempo de computación, pues LZS tarda 15 veces más en encontrar el camino, una diferencia de tiempo que puede ser crítica según la rapidez que se requiera en el lazo de control.

7.1.8 Prueba 8. Destino inaccesible

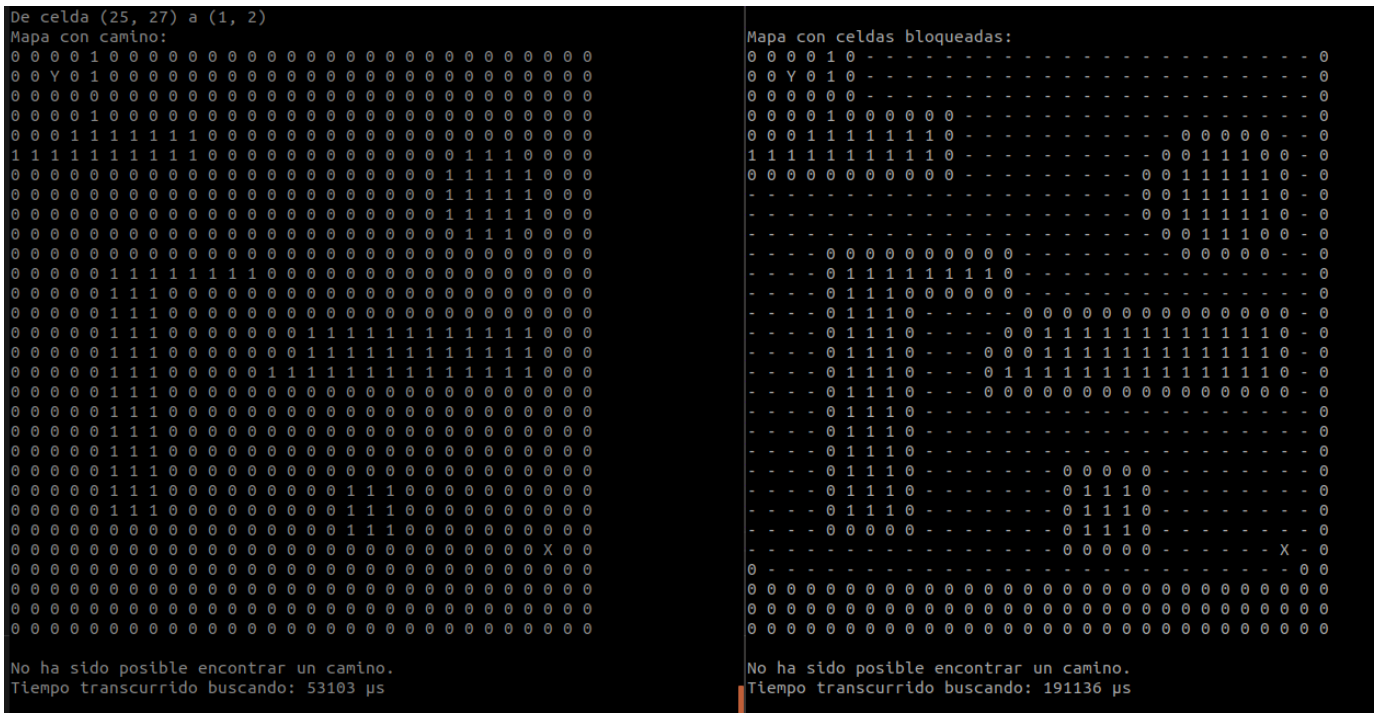


Figura 7-11. Prueba en simulación de un destino inaccesible.

En esta siguiente prueba, se pretende comprobar cuánto tardan los métodos en detectar que el destino es inaccesible, estando este lo más lejos posible del punto inicial. El inicio está abajo a la derecha y el destino arriba a la izquierda.

Aunque en el mapa se ve cómo hay un hueco (‘0’) por el que se podría llegar al destino, el robot no cabe por ese hueco por lo que si intentara pasar por él chocaría inevitablemente. En este sentido ambos métodos funcionan bien, pues detectan que por ahí no se puede pasar.

Mirando el tiempo que ha tardado cada algoritmo en detectar que no hay un camino válido, LZS tardó 53ms, mientras que APF 191ms. Este tiempo es excesivamente alto y no sería admisible en apenas ninguna situación real, pues la frecuencia de control resultaría demasiado baja.

Este elevado tiempo (191ms) se debe a que el algoritmo ha ido bloqueando de forma recursiva prácticamente todas las celdas del mapa (marcadas con signo ‘-‘). En este caso obviamente sería preferible el método LZS.

7.1.9 Prueba 9. Mapa de habitaciones

En las siguientes pruebas se cambia de mapa a uno con una estructura de habitaciones unidas por un pasillo. Hay tres habitaciones en la parte superior, tres en la inferior y un pasillo horizontal que une todas en el medio.

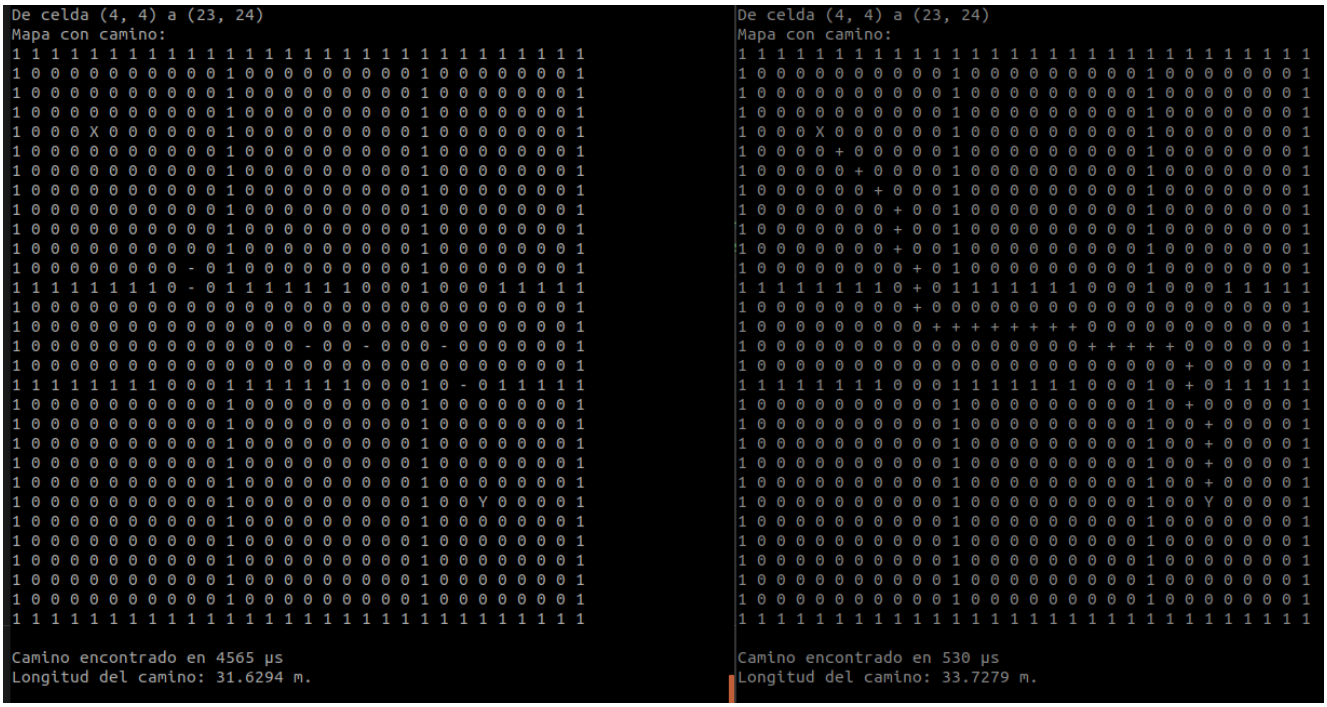


Figura 7-12. Prueba en simulación con mapa de habitaciones.

La primera prueba va desde la habitación arriba a la izquierda hasta la de abajo a la derecha. Ambos métodos encuentran un camino válido con las diferencias usuales: LZS tarda más pero su camino es de menor longitud. En este caso habría un mínimo local en APF al acercarse a la habitación del medio de abajo, sin embargo, lo evita correctamente sin llegar a entrar en esa habitación.

7.1.10 Prueba 10. Mapa de habitaciones con mínimo local

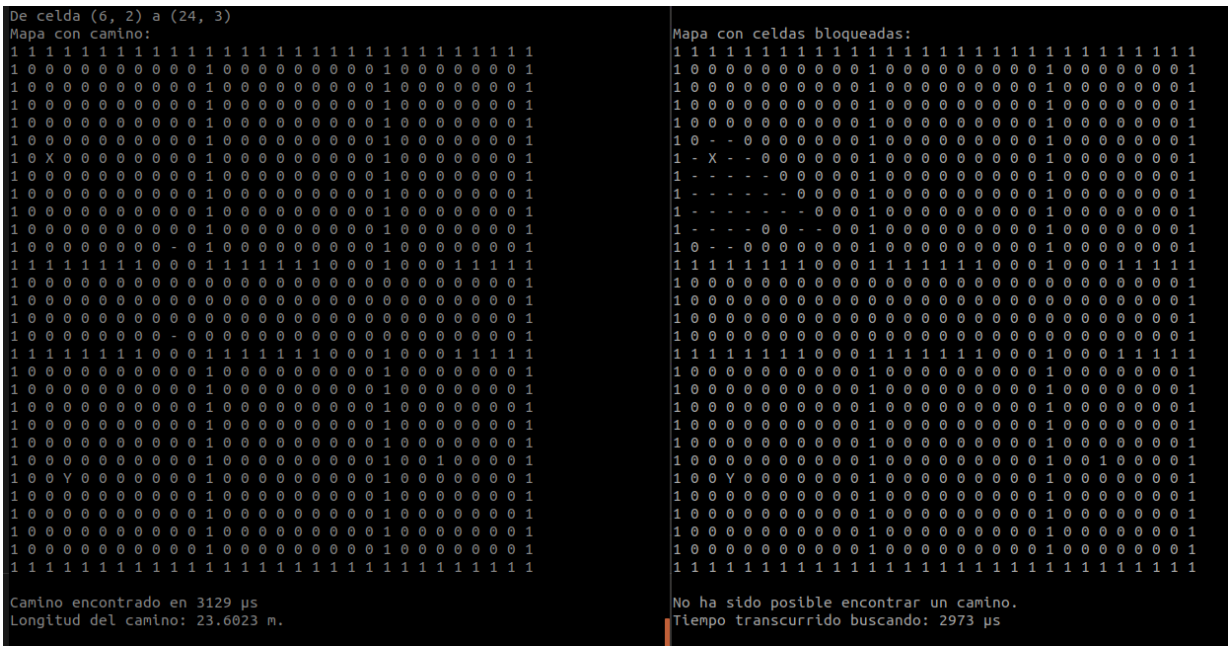


Figura 7-13. Prueba en simulación con mapa de habitaciones con mínimo local.

En este caso, el mínimo local en APF no se consigue evitar, y no se encuentra un camino válido. Atendiendo a las celdas bloqueadas se observa cómo casi se consigue salir del mínimo local, pues si el punto de inicio estuviera un poco desplazado a la derecha sería suficiente.

En cuanto a LZS, el camino encontrado consiste tan solo en dos puntos, resultando bastante eficiente en cuanto a longitud se refiere.

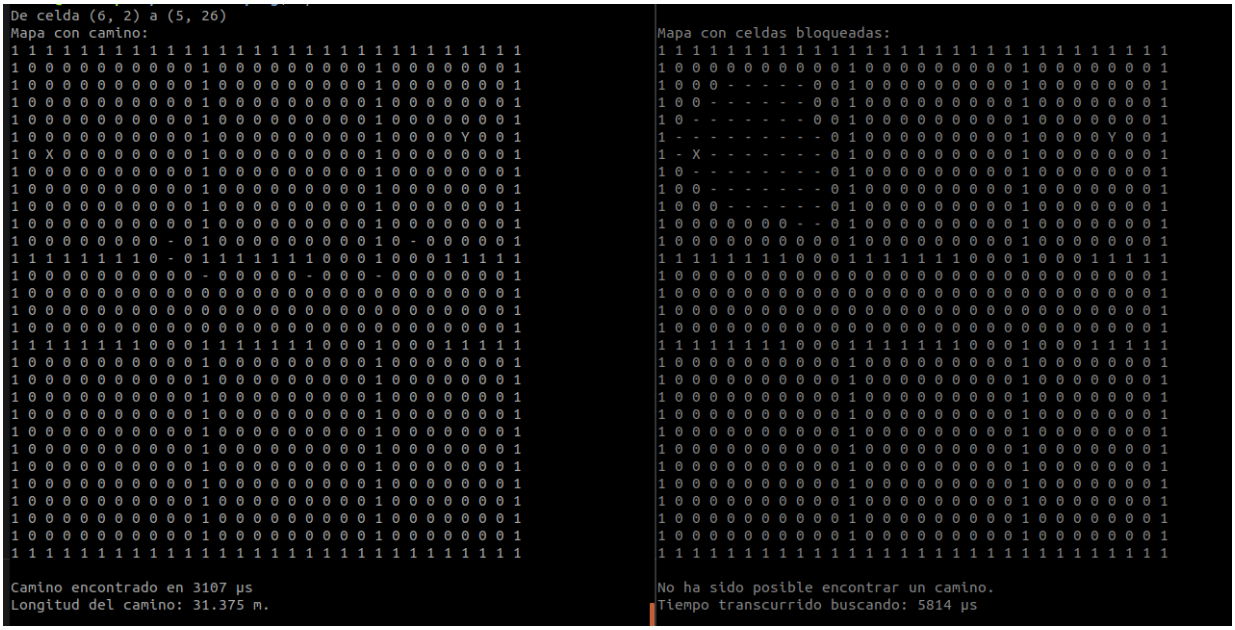


Figura 7-14. Otra prueba en simulación con mapa de habitaciones con mínimo local.

Se realiza otra prueba con mínimo local, pero ahora la habitación objetivo es la de arriba a la derecha. Como la pared de obstáculos es muy grande, no se consigue evitar el mínimo local y se tardarían casi 6ms en conocerlo.

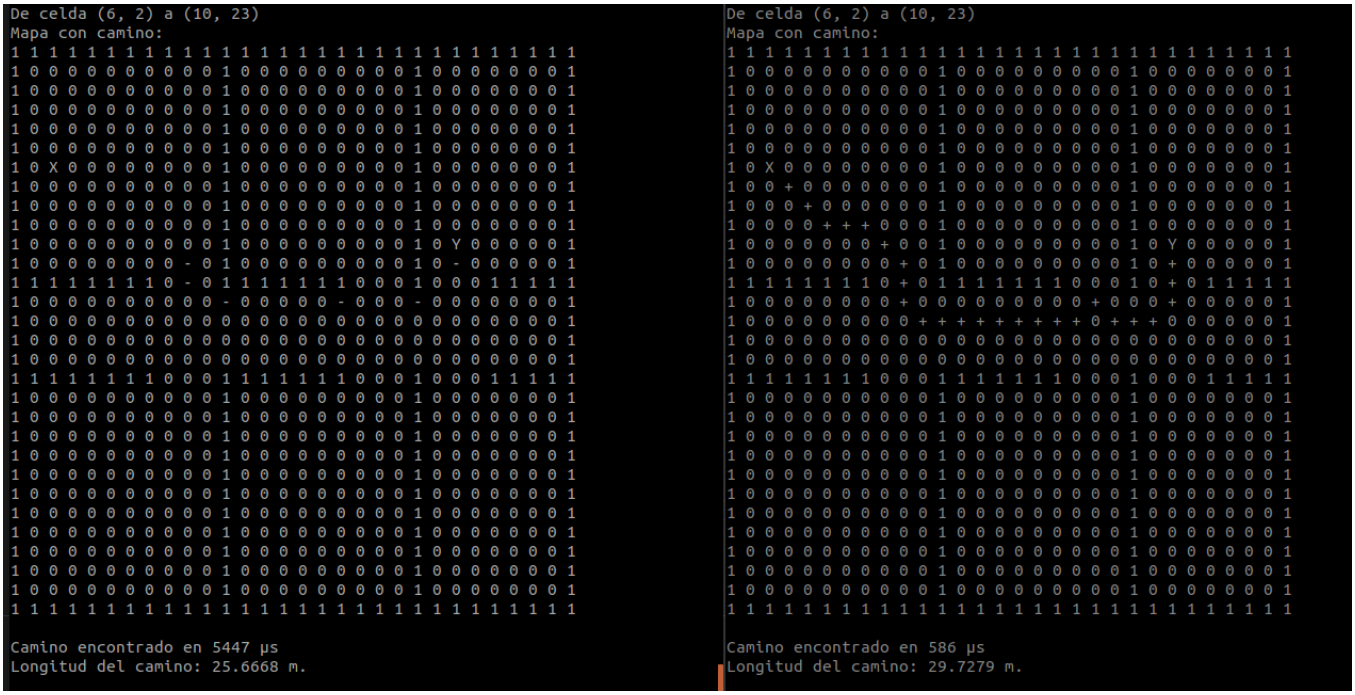
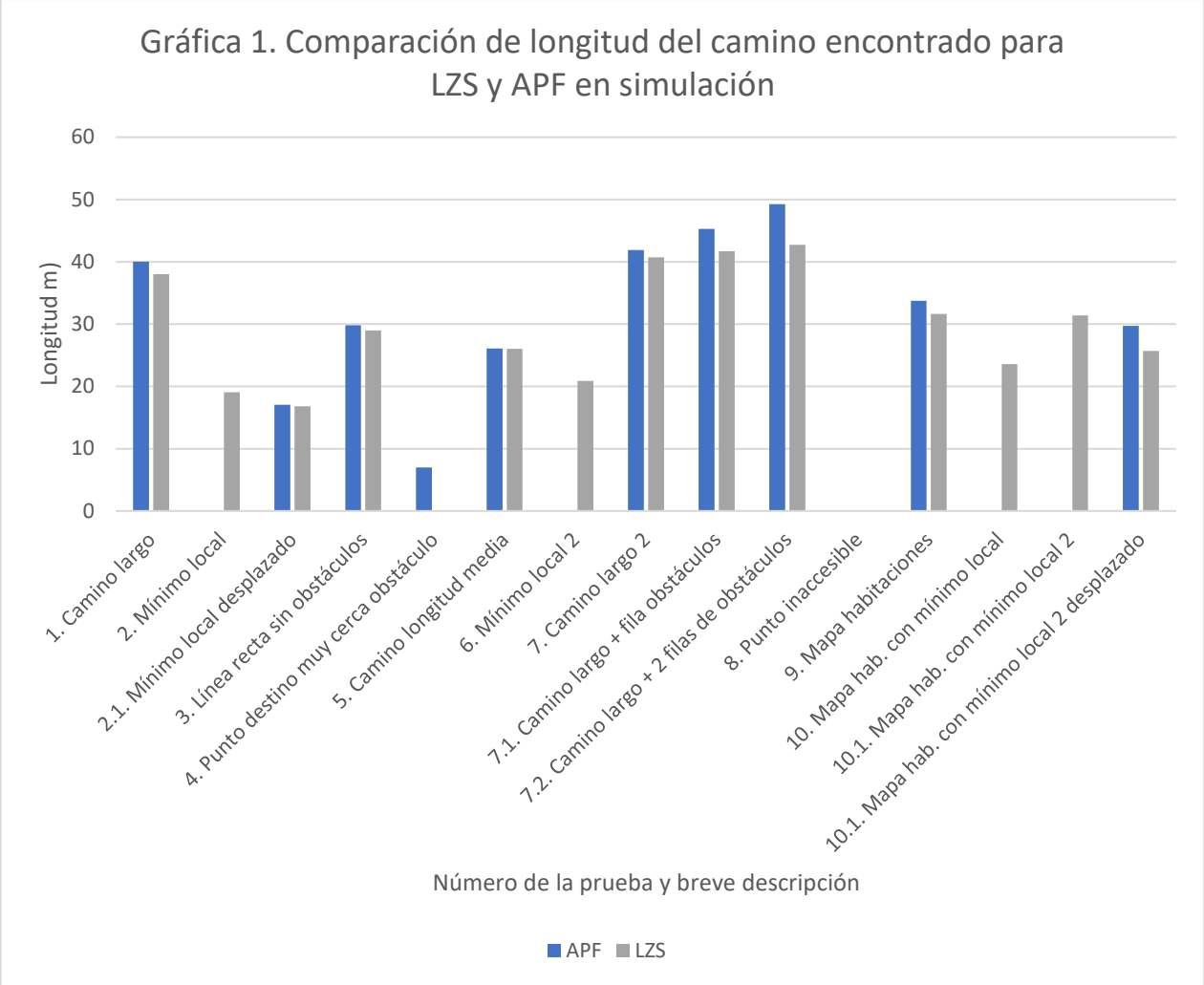


Figura 7-15. Prueba en simulación con mapa de habitaciones con mínimo local evitado.

Se comprueba que, si el punto destino se desplaza ligeramente hacia abajo, el mínimo local sigue existiendo, pero ahora sí que se evita correctamente, hallando una trayectoria válida. En cuanto a longitud del recorrido, la diferencia es aproximadamente de 4.0m menor para LZS, pero tarda casi 10 veces más en encontrar el camino.

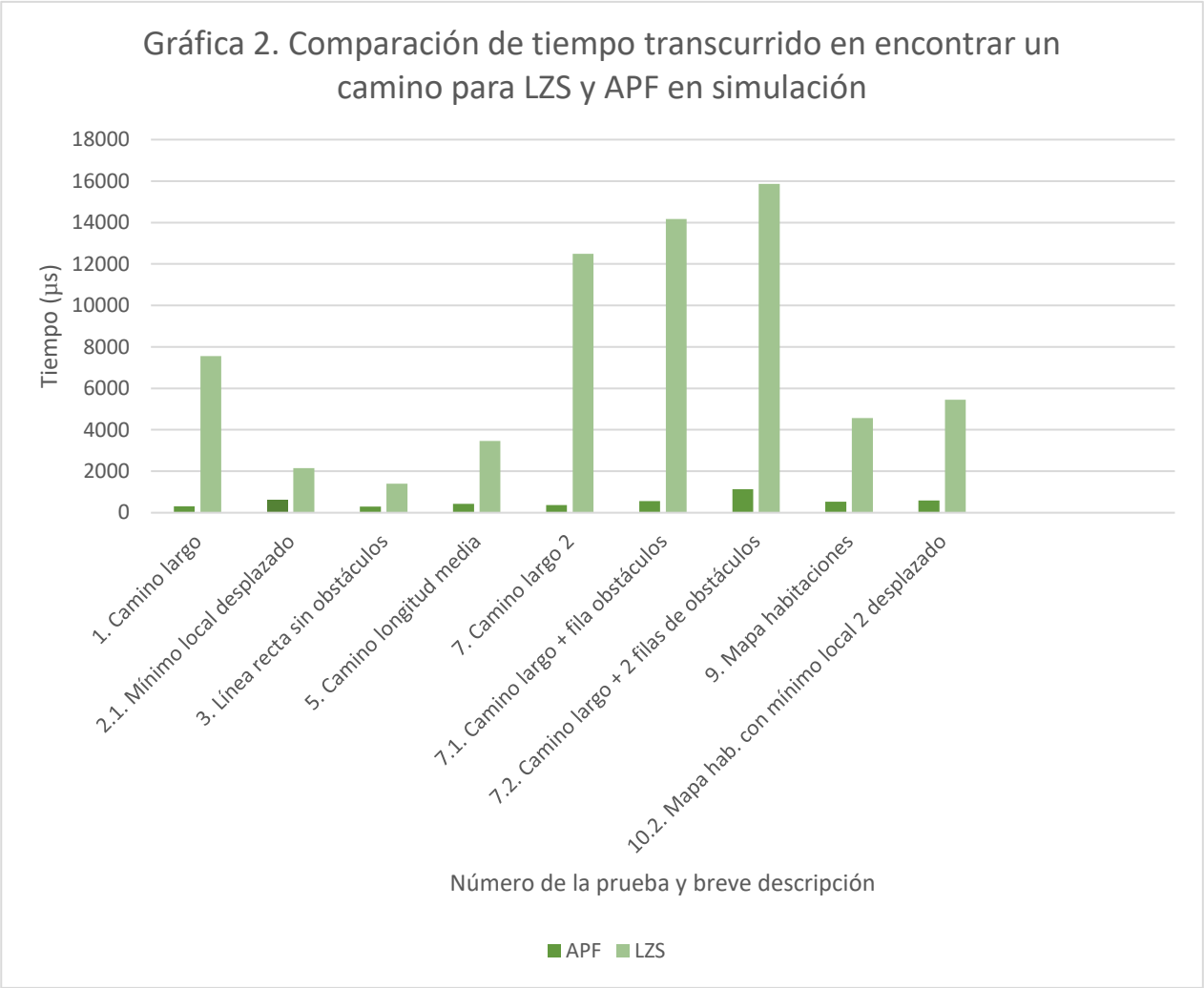
7.2 Gráficas de pruebas en simulación

Ya se han presentado los resultados obtenidos en diversas pruebas en el simulador, pero con el fin de visualizar mejor la información obtenida se obtendrán algunas gráficas.



En las pruebas en las que se encontró camino para ambos métodos, la diferencia media de longitud de camino es de 2.29m, siendo menor siempre en LZS. Expresado en porcentaje, el camino de LZS es de media un 5.91% más corto que el de APF.

Las barras que no aparecen indican que para esa prueba ese método no pudo encontrar un camino válido.



En este gráfico tan solo se muestran las pruebas en las que ambos métodos encontraron un camino. Se observa claramente cómo APF es mucho más rápido que LZS. De media, APF es 3000µs más rápido, o un 1367%, o sea que es más de 13 veces más rápido. Esto es una gran ventaja de APF respecto a LZS, pues al tener menor carga computacional dedicada al algoritmo en sí, se puede dedicar a otras partes del sistema, por ejemplo, a realizar un control de movimiento más preciso u obtener datos de mayor calidad en los sensores.

Aun así, el tiempo de computación de LZS no es para nada exagerado, siendo de casi 16ms en el peor de los casos. Esta marca de tiempo permitiría un bucle de control con suficientes hercios para llevar a cabo un buen método de evitación de obstáculos en tiempo real sin problema.

7.3 Pruebas en ROSBOT 2.0

Tras realizar distintas pruebas en simulación y comprobar que el funcionamiento básico de los algoritmos desarrollados es correcto, se procede a realizar diversas pruebas en el robot ROSBOT 2.0.

Para ello, se han creado dos plugins para el paquete 'move_base': uno para el algoritmo de campos potenciales artificiales (APF) y otro para el Lazy Theta Star (LZS). Ambos tienen el mismo control de movimiento, por lo que diferencia radica en el método que utilizan para calcular la trayectoria a seguir, obviamente.

Para cada prueba se han tomado dos medidas con el fin de comparar ambos métodos:

- Tiempo empleado en llegar al destino (en segundos)
- Distancia recorrida por el robot (en metros)

Para que los métodos se puedan comparar mejor, se han establecido el parámetro de distancia de inflado de obstáculos de LZS y el radio de influencia de los obstáculos de APF al mismo valor: 2.8. Esto corresponde a 28cm, pues la resolución del mapa es de 0.1m por celda. Esta distancia incluye el radio máximo del robot más unos 10cm de margen, para que este no pase demasiado cerca de los obstáculos, además de que los movimientos no son exactos.

En cada prueba se ha hecho una grabación en vídeo para que se pueda ver mejor el movimiento que realiza el robot. Esta se puede ver en los enlaces que aparecen al final de la sección de la prueba o en el anexo correspondiente.

Además, se han hecho capturas de pantalla en RVIZ, donde se ve el mapa, el robot y su orientación.

7.3.1 Prueba 1. Avanzar en línea recta sin obstáculos

La primera prueba que se ha realizado es muy sencilla, simplemente consiste en mover el robot en línea recta hacia delante una distancia de 1m, sin obstáculos de por medio. Esta es la situación inicial:

Figura 7-16. Situación inicial de la prueba 1.

A continuación, se mostrarán capturas de pantalla del recorrido del robot en RVIZ, tanto para APF como para LZS. El rectángulo verde es la huella del robot, la flecha verde la orientación que tiene, las líneas verde claro son el plan global (no usado en ninguno de los métodos), los cuadrados negros son obstáculos en el mapa, gris claro es camino libre y gris oscuro es zona desconocida.

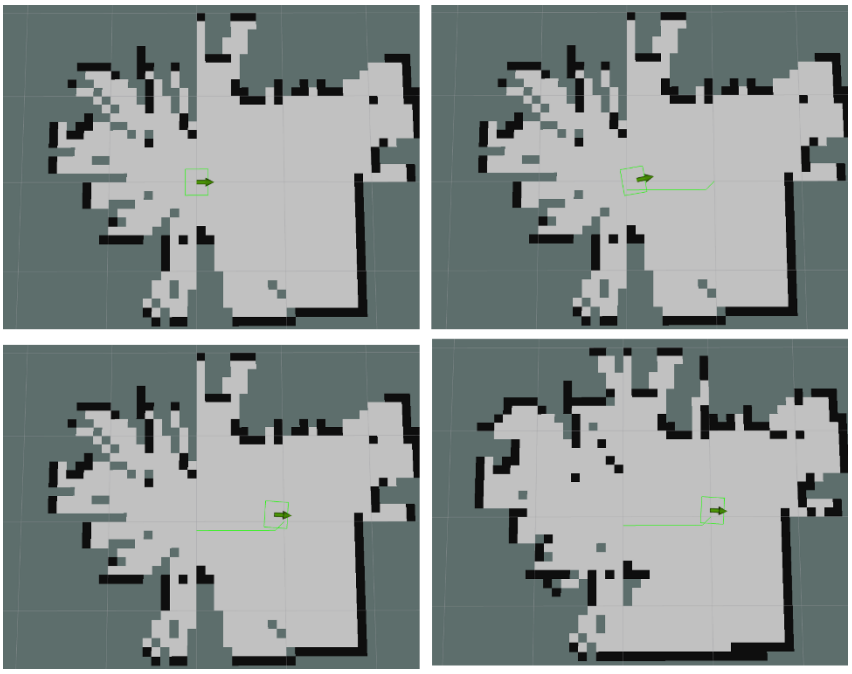


Figura 7-17. Recorrido de la prueba 1 con APF.

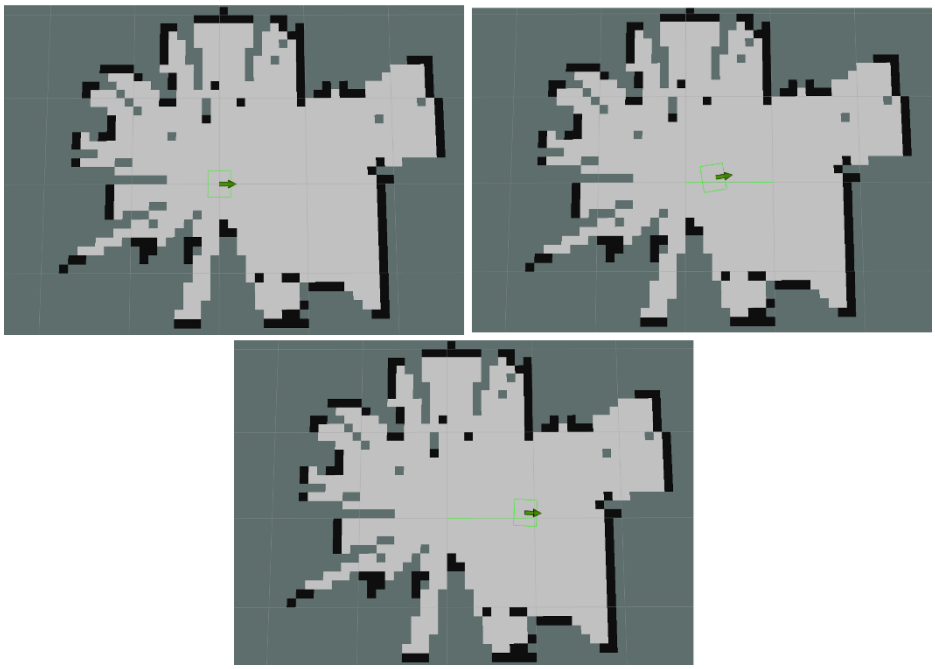


Figura 7-18. Recorrido de la prueba 1 con LZS.

Las capturas no aportan demasiada información en este caso debido a la gran sencillez de la prueba. En cuanto a datos, APF tardó 5.98s en llegar al destino, mientras que LZS lo hizo en 5.39s. La longitud recorrida con APF fue de 1.00m y con LZS de 0.88m. En ambas métricas LZS supera a APF, lo cual indica que, para escenarios sencillos con mucho espacio libre de obstáculos, a priori LZS parece una mejor opción.

La diferencia de las métricas se debe también al control usado, pues difiere ligeramente en ambos casos. En LZS, como los puntos del camino son mínimos para llegar al destino, se siguen todos uno a uno a la hora de desplazar el robot. Sin embargo, en APF los puntos del camino suceden celda a celda hasta llegar al destino. Por esto, como el control de movimiento es proporcional, para evitar que el robot acelere y frene cada vez que llegue a un punto, en vez de seguir todos, se sigue uno de cada 3 puntos de la trayectoria. Se evitan algunos de esos acelerones indeseados, pero no todos, por eso tarda más en llegar al punto destino. Este efecto se ve también en los vídeos siguientes.

Vídeo de prueba 1 con APF

Vídeo de prueba 1 con LZS

7.3.2 Prueba 2. Avanzar en línea recta con un obstáculo en medio

Esta segunda prueba es como la anterior, pero ahora se sitúa un obstáculo entre el robot y el punto destino, que estará a 1m hacia delante del punto inicial, como antes.

Figura 7-19. Situación inicial de la prueba 2.

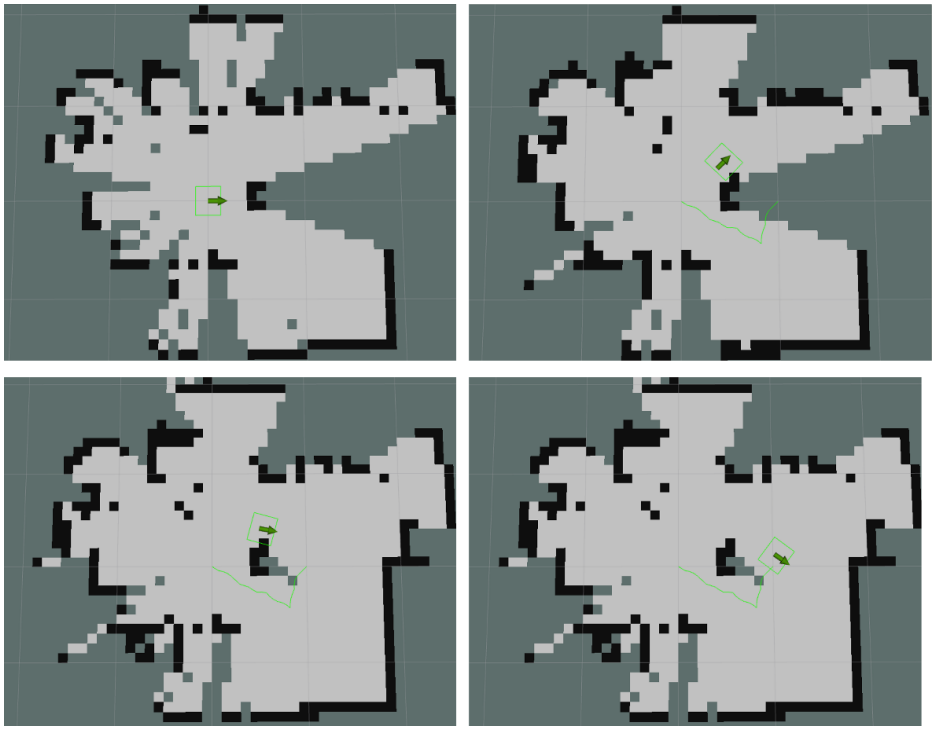


Figura 7-20. Recorrido de prueba 2 con APF.

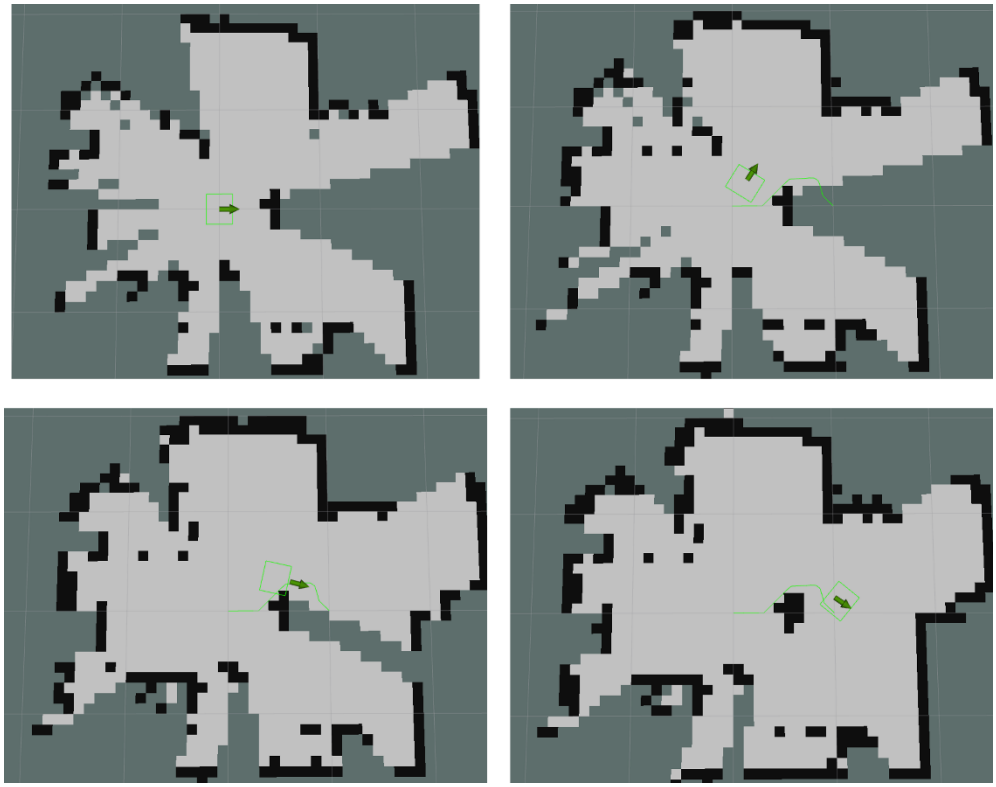


Figura 7-21. Recorrido de la prueba 2 con LZS.

Ambos métodos evitan el obstáculo de forma correcta, y por el mismo lado. Pero vuelve a haber diferencias en cuanto a los resultados numéricos. En este caso, APF tardó 15.28s y recorrió 1.34m, mientras que LZS tardó 18.09s con una longitud recorrida de 1.37m.

Las distancias recorridas apenas difieren, pero APF lo hizo casi 3s más rápido. Esto es porque LZS evita el obstáculo con un margen mayor, lo que es positivo porque así es más segura la trayectoria seguida pero quizá en este caso tampoco era necesario que fuera tan grande. Por eso, APF supera a LZS en esta prueba.

Vídeo de prueba 2 con APF

Vídeo de prueba 2 LZS

7.3.3 Prueba 3. Volver al punto inicial después de prueba 2.

Esta prueba consiste en volver al punto inicial después de haber terminado la prueba 2. Así, el punto destino estará en el sentido contrario al que apunta el robot, por lo que deberá hacer un giro brusco antes de empezar el movimiento.

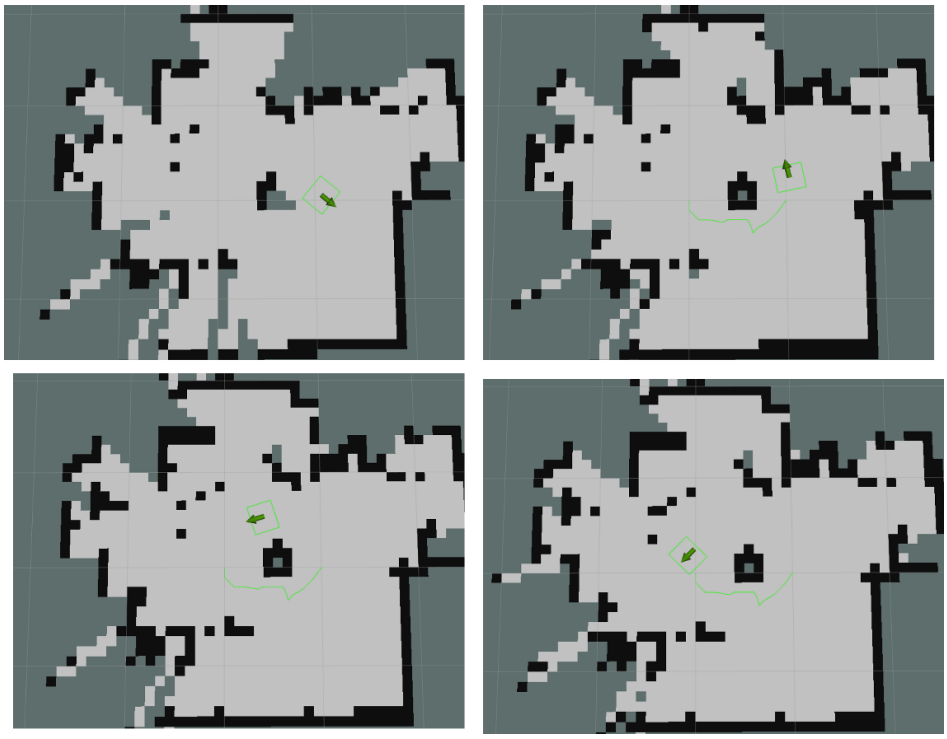


Figura 7-23. Recorrido de la prueba 3 con APF.

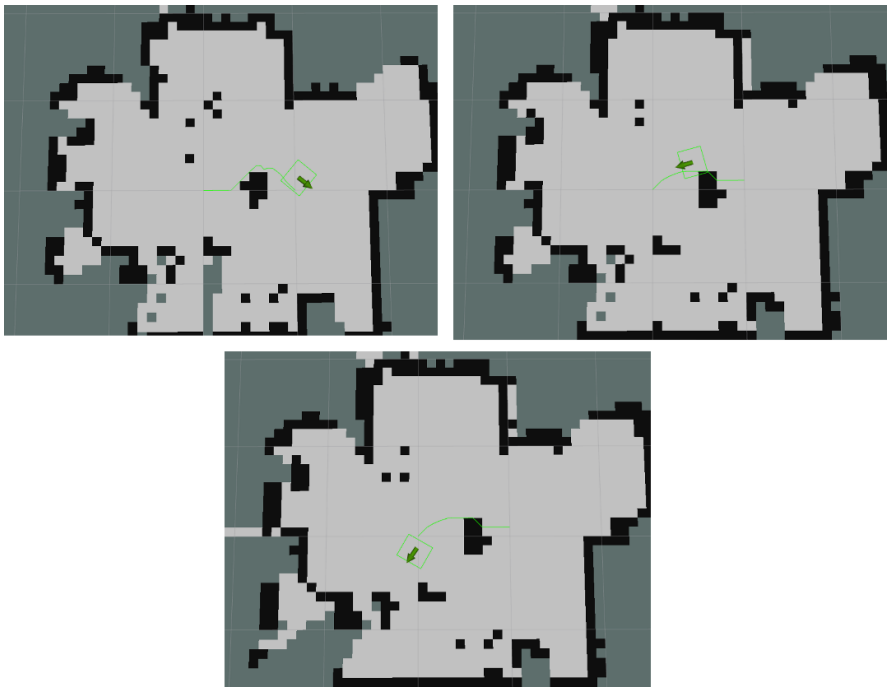


Figura 7-22. Recorrido de la prueba 3 con LZS.

Los algoritmos no tienen problema en resolver esta situación sencilla. Ambos vuelven por el mismo lado que fueron, lo que tiene sentido, pues si no ha cambiado el mapa, ese sigue siendo el camino más corto que encuentra cada método para volver al punto inicial anterior.

APF tarda 24.19s y recorre 1.46m, mientras que LZS tarda 20.69s con 1.30m recorridos. Ambas marcas temporales son superiores a las de la prueba anterior, precisamente por ese giro inicial que hace el robot. La distancia recorrida con LZS es menor, pero es porque el robot pasa muy cerca del obstáculo y colisiona ligeramente con él, por lo que realmente la prueba no es superada con éxito. Eso sucede porque cuando el robot va al primer punto de la trayectoria encontrada, hace el giro en sentido horario, mientras que APF lo hace en antihorario. Al ser horario, cuando empieza a moverse tiene una orientación más directa al objeto que APF, lo que provoca la colisión posterior. Si hubiera girado en sentido antihorario, seguramente habría sorteado el objeto sin problema.

Vídeo de prueba 3 con APF

Vídeo de prueba 3 con LZS

7.3.4 Prueba 4. Camino con dos obstáculos.

Ahora se añade otro obstáculo en el suelo y el punto destino vuelve a estar a un metro delante del robot.

Figura 7-25. Situación inicial de la prueba 4.

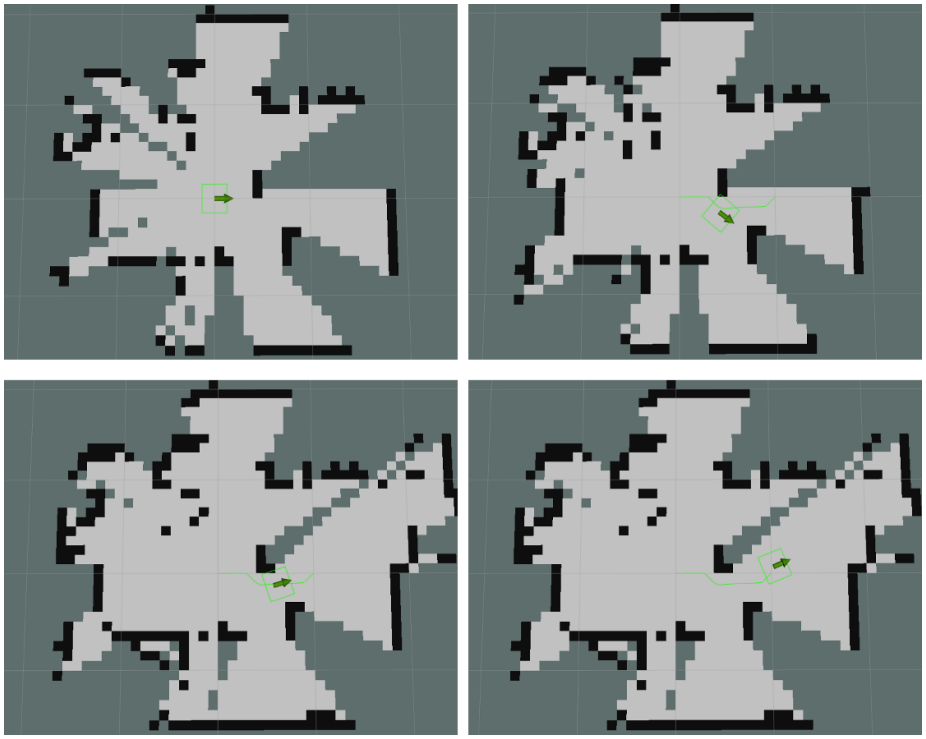


Figura 7-24. Recorrido de la prueba 4 con APF.

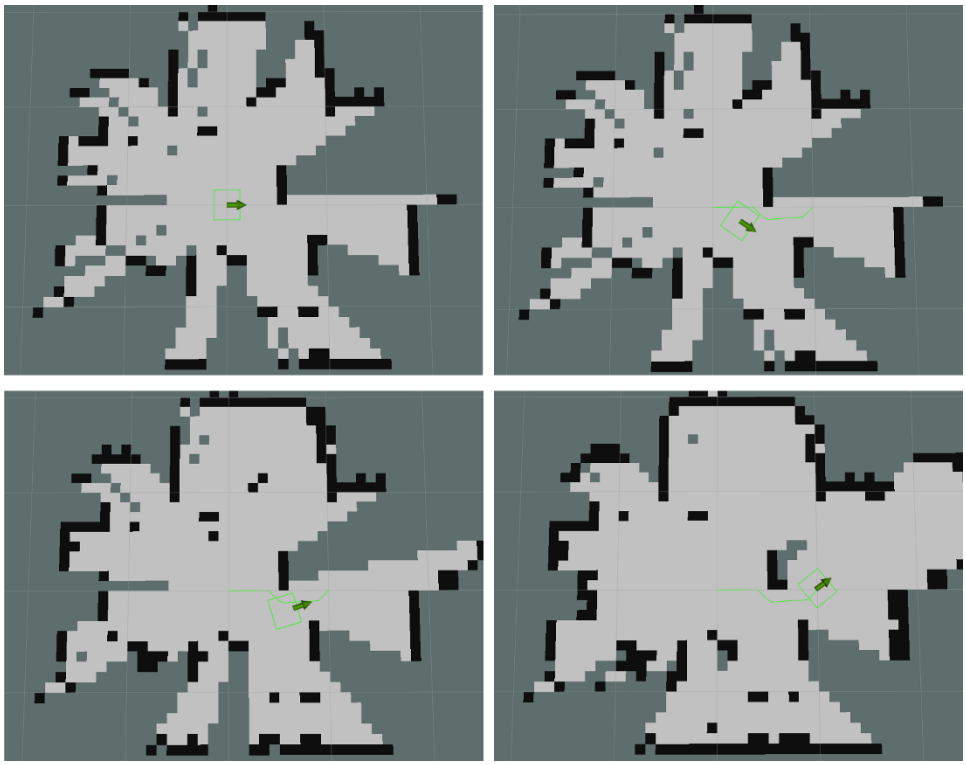


Figura 7-26. Recorrido de la prueba 4 con LZS.

En los mapas se aprecia cómo al principio, el robot no conoce qué hay en el mapa en el punto destino. Conforme avanza hacia él, descubre el mapa y al estar libre no hay problema en llegar. Ambos métodos evitan los obstáculos correctamente, girando primero hacia la derecha y luego hacia la izquierda.

APF tarda 15.99s y recorre 1.10m, mientras que LZS tarda 15.49s y recorre 1.23m. Al igual que en la prueba anterior, APF recorre menor distancia porque pasa más cerca del primer objeto, quizá demasiado para considerarlo un margen de seguridad suficiente.

Vídeo de prueba 4 con APF

Vídeo de prueba 4 con LZS

7.3.5 Prueba 5. Volver a punto inicial tras mover un obstáculo de sitio

Esta prueba es una continuación de la anterior. Se indica al robot que vuelva al punto inicial, pero antes se mueve un objeto de sitio y se coloca tal que obstruya el camino directo al destino.

Figura 7-27. Situación inicial de la prueba 5.

Por tanto, el robot no podrá seguir la ruta anterior para llegar el destino y deberá buscar otra ruta que sí sea válida.

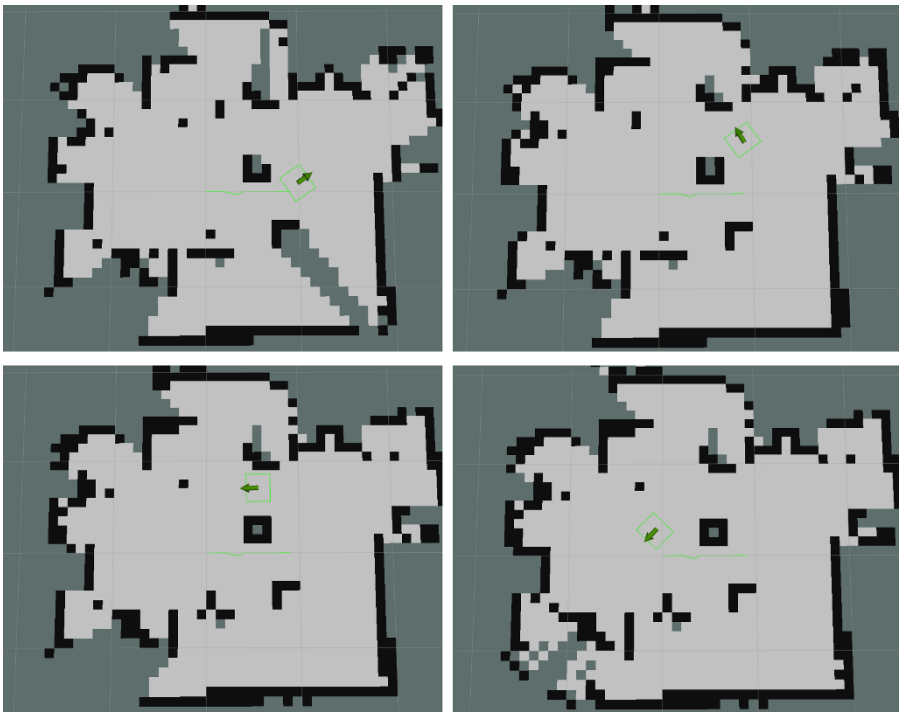


Figura 7-28. Recorrido de la prueba 5 con APF.

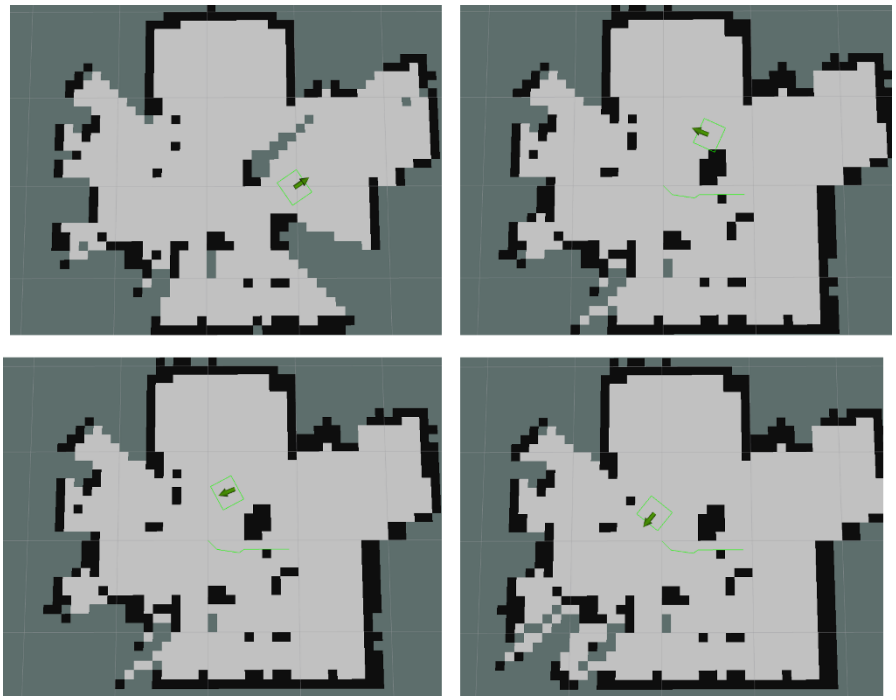


Figura 7-29. Recorrido de la prueba 5 con LZS.

Ambos métodos evitan los obstáculos correctamente, encontrando trayectorias muy similares. En las capturas correspondientes a APF parece que el obstáculo no se ha movido de sitio, pero no es así. Lo que sucede es que el mapa no se ha actualizado correctamente. El que se muestra es el mapa global, mientras que el algoritmo hace uso del mapa local (que sí cambia continuamente) para buscar caminos, por eso la trayectoria sí se realiza bien. El motivo por el que no se actualizó el mapa global al moverse el robot es desconocido.

APF tardó 25.29s y recorrió 1.72m, mientras que LZS tardó 19.29s con una distancia recorrida de 1.46m. En este caso LZS supera a APF en ambas métricas, pues hace el recorrido más rápido y recorriendo una distancia menor, aunque ambos evitaran los obstáculos por el mismo lado.

Con esta prueba se pretendía poner de manifiesto la capacidad de los algoritmos de adaptarse a cambios en el entorno y poder encontrar nuevos caminos para llegar a los puntos objetivo. Estos cambios del entorno deben ocurrir antes de que el robot calcule la trayectoria, pues si a mitad del movimiento se mueve un obstáculo el robot no lo tendría en cuenta y podría colisionar con él. Por tanto, los métodos no están adaptados para obstáculos móviles, solo para estáticos. Aun así, funcionan correctamente para ese tipo de situaciones.

Vídeo de prueba 5 con APF

Vídeo de prueba 5 con LZS

7.3.6 Prueba 6. Mínimo local

En esta prueba se crea una situación con un mínimo local para APF, por tanto, está orientada a este método y ver si consigue evitarlo correctamente. El punto destino está situado a 1m delante de la posición inicial.

Figura 7-31. Situación inicial de la prueba 6.

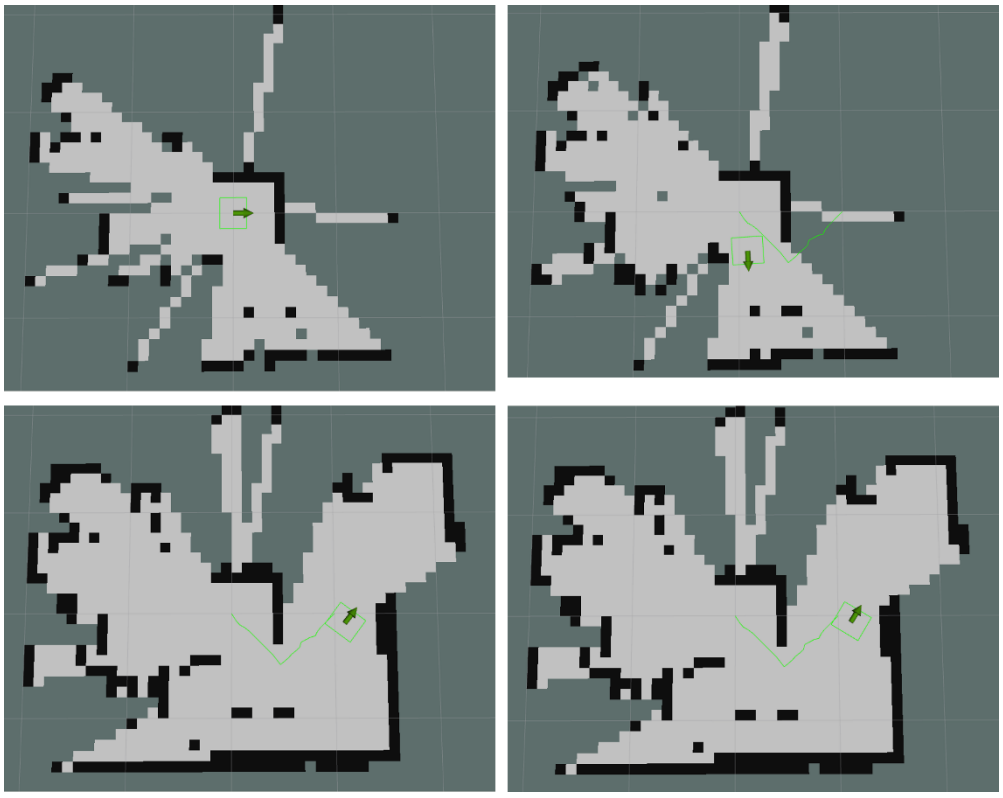


Figura 7-30. Recorrido de la prueba 6 con LZS.

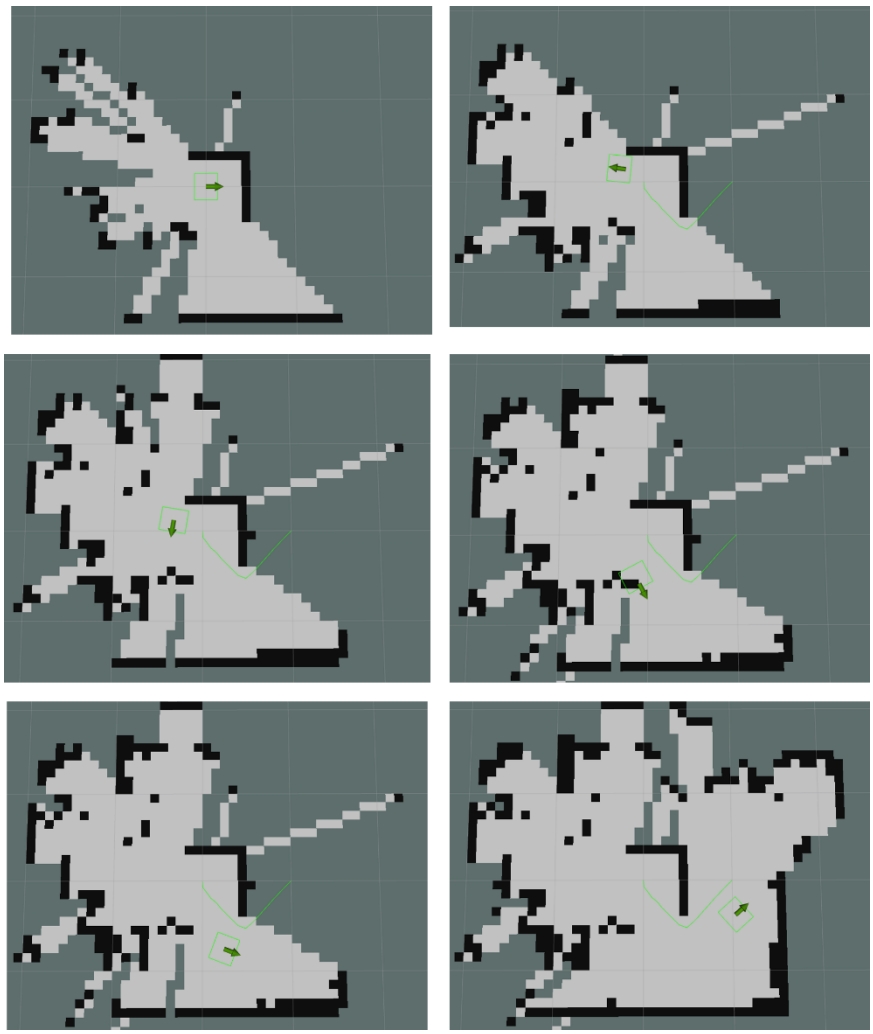


Figura 7-32. Recorrido de la prueba 6 con APF.

Ambos métodos han conseguido llegar correctamente el destino, APF evitando el mínimo local. LZS toma un camino mucho más directo, yendo directamente hacia la derecha y tras pasar los obstáculos girando para llegar al punto objetivo. Por otro lado, APF primero se mueve hacia atrás y luego ya evita los obstáculos por la derecha como LZS. Este desplazamiento raro en sentido contrario al objetivo quizá se deba a que el potencial de repulsión de los obstáculos tiene un valor elevado inicialmente, lo que obliga al robot a moverse hacia atrás.

APF tarda 36.29s y recorre 2.61m, mientras que LZS tarda 24.69s y recorre 1.69m. Obviamente LZS tiene un desempeño mucho mejor que APF, debido principalmente al desplazamiento inicial de APF ya comentado.

En este tipo de situaciones el método LZS es más robusto que APF, pues no existe el concepto de mínimo local y se resuelve como una situación normal de evitación de obstáculos. Sin embargo, APF tiene esa dificultad añadida.

Vídeo de prueba 6 con APF

Vídeo de prueba 6 con LZS

7.3.7 Prueba 7. Mínimo local 2

Esta prueba es similar a la anterior, pero se genera un mínimo local con una solución más complicada de resolver para APF.

Figura 7-33. Situación inicial de la prueba 7.

El punto destino está situado a 1.2m por delante de la posición inicial. Al añadir los obstáculos a ambos lados del robot se crea esa dificultad extra a la hora de evitar el mínimo local.

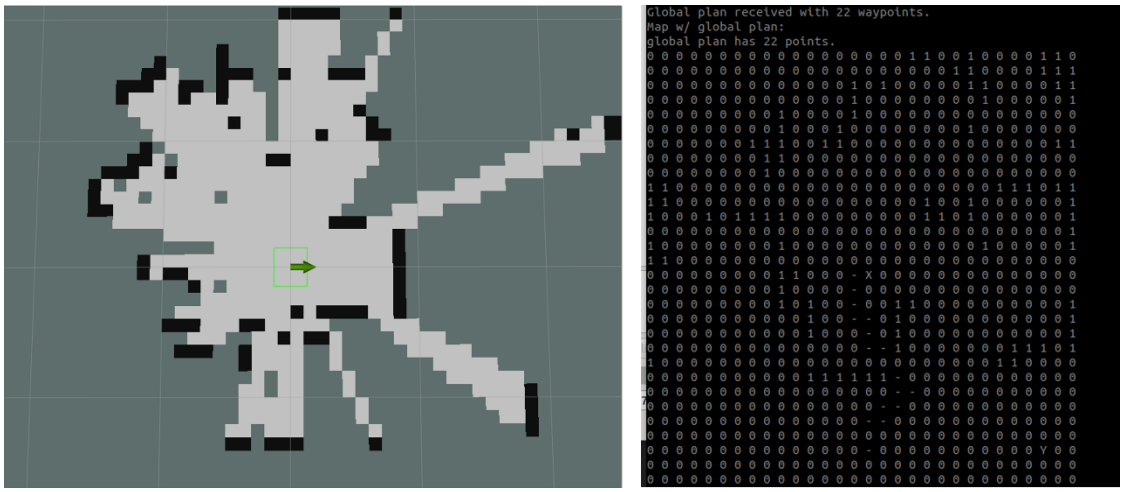


Figura 7-34. Resultado de la prueba 7 con APF.

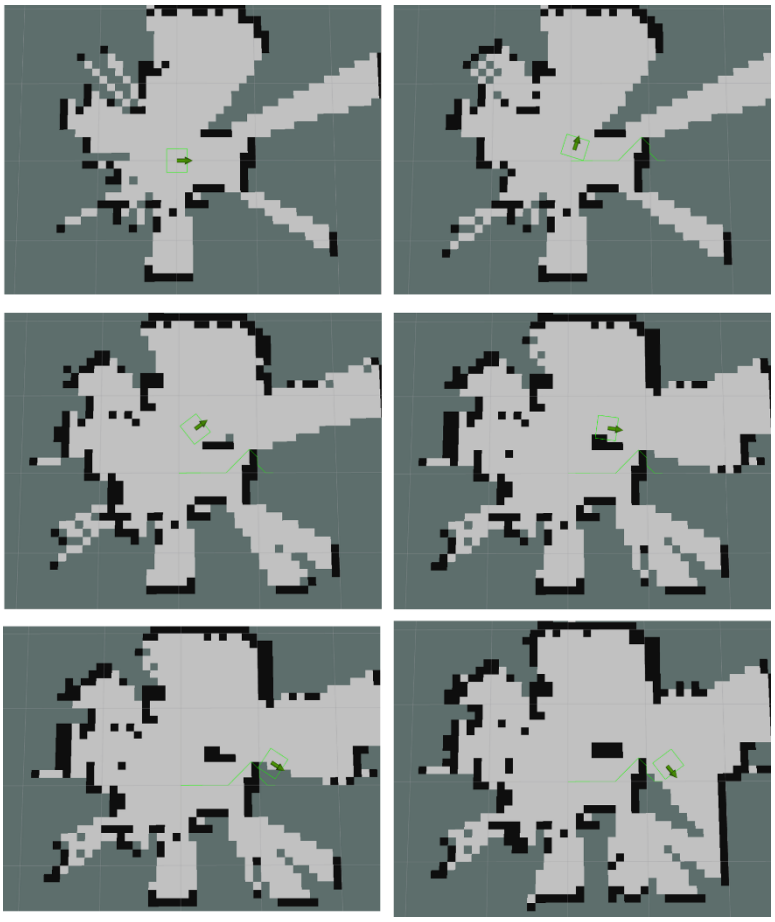


Figura 7-35. Resultado de la prueba 7 con LZS.

En este caso APF no consigue encontrar un camino hasta el punto destino y evitar el mínimo local. En la captura de la derecha se ve la trayectoria global obtenida, aunque se ignora a la hora de utilizar los métodos. Se aprecia cómo esta trayectoria pasa por el hueco entre los obstáculos, pero en la imagen de la situación inicial se comprueba que el robot no cabe por ahí. La trayectoria global no tiene en cuenta el tamaño del robot y precisamente por esto se ignora y no se utiliza.

LZS tarda 25.69s y recorre 2.01m, y APF dedica 751µs a buscar un camino. La solución de LZS a la situación era de esperar, gira a la izquierda y evita la zona entre los obstáculos.

Vídeo de prueba 7 con APF

Vídeo de prueba 7 con LZS

7.3.8 Prueba 8. Desplazamiento por mapa desconocido

En esta prueba se colocan estratégicamente obstáculos en zonas desconocidas del mapa a priori para el robot. Por tanto, cuando el robot calcule la trayectoria hacia el punto destino, al no tener conocimiento de esos obstáculos, colisionará con ellos.

Figura 7-36. Situación inicial de la prueba 8.

El punto objetivo se encuentra a 1.2m delante de la posición inicial. Para llegar hasta él, es de esperar que el robot evite los obstáculos que tiene enfrente por la derecha. El problema está en que la libreta azul del medio no es conocida para el robot a priori, por lo que no la tendrá en cuenta a la hora de buscar una trayectoria, colisionando con ella seguramente cuando se mueva. Esta prueba es de esperar que no funcione, pero aun así se ha realizado para poner de manifiesto más claramente el problema de no tener un mapa totalmente conocido.

Figura 7-37. Resultado de la prueba 8 con APF.

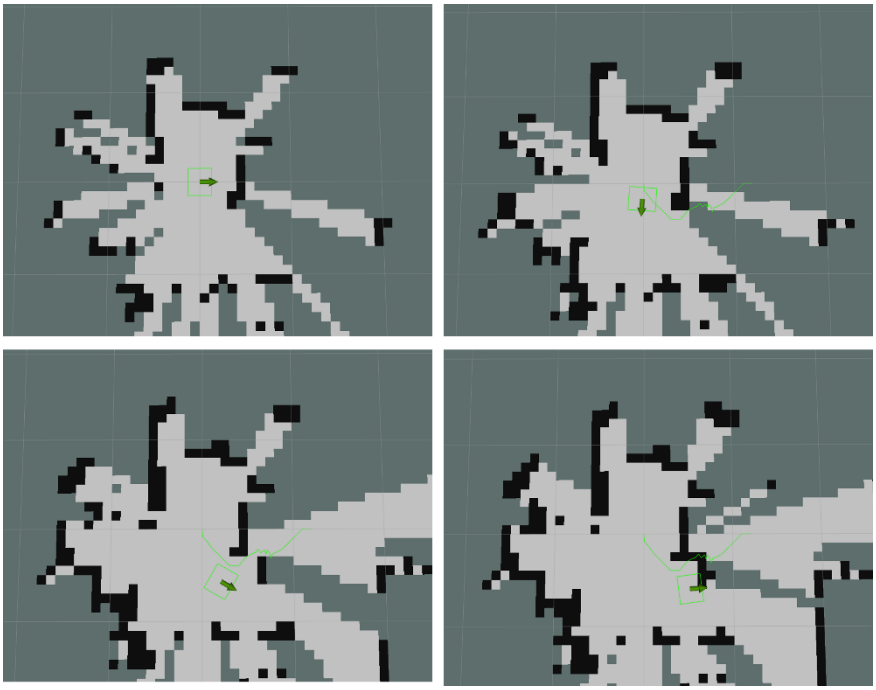


Figura 7-38. Resultado de la prueba 8 con LZS.

El algoritmo APF no ha sido capaz de encontrar un camino para esta prueba, pues se ha formado un mínimo local enfrente del robot. No era la intención en esta prueba, pero una vez más se pone de manifiesto el problema de los mínimos locales para APF.

LZS sí que encuentra un camino válido en principio hasta el objetivo. En la primera captura se puede ver cómo los obstáculos que están enfrente del robot bloquean la visión y no se detecta el obstáculo que está más a la derecha. En las dos últimas capturas el mapa se actualiza y ya sí aparecen los obstáculos, pero como no se tienen en cuenta a la hora de buscar una trayectoria, el robot acaba colisionando con ellos.

Existen diversas soluciones posibles para este problema. Una solución podría ser detectar cuando aparece un nuevo obstáculo en el mapa y volver a calcular en ese instante, por lo que ya sí se tendría en cuenta y se evitaría. Otra solución sería calcular la trayectoria periódicamente. Así se haría uso del mapa actualizado en cada instante, el problema es que utilizarían más recursos que en la anterior.

Lo ideal habría sido usar el detector de colisiones desarrollado anteriormente junto con los algoritmos de evitación de obstáculos, y recalcular la trayectoria cada vez que se detecte uno. Además de recalcularla cuando se detecte un obstáculo, debería hacerse de forma periódica también. Esto no se ha implementado porque en las pruebas se quería ver el funcionamiento individual de los dos algoritmos, pero podría ser una buena solución para resolver este problema.

De todas maneras, en los métodos desarrollados de evitación de obstáculos se supone un mapa conocido, por lo que este problema no existiría. Los métodos se podrían modificar siguiendo una de las soluciones descritas u otra diferente para resolverlo.

Vídeo de prueba 8 con APF

Vídeo de prueba 8 con LZS

7.3.9 Prueba 9. Entorno con varios obstáculos circulares pequeños.

En esta prueba se colocan entre el robot y el punto destino diversos obstáculos circulares pequeños. Se sitúan más próximos entre sí que los obstáculos de pruebas anteriores, por lo que el robot tendrá menos margen para evitarlos.

Figura 7-39. Situación inicial de la prueba 9.

El punto destino está situado a 1.2m delante de la posición inicial. El obstáculo de la derecha se sitúa para que el robot no se mueva hacia la derecha sorteando todos los obstáculos pequeños para llegar al destino.

Con los parámetros de las pruebas anteriores no se encuentra un camino válido en ninguno de los métodos. Para ello, se baja la distancia de inflado de los obstáculos de LZS de 2.8 a 2.0. En APF hay que bajar el radio de influencia de los obstáculos hasta 1.3 para que encuentre un camino válido.

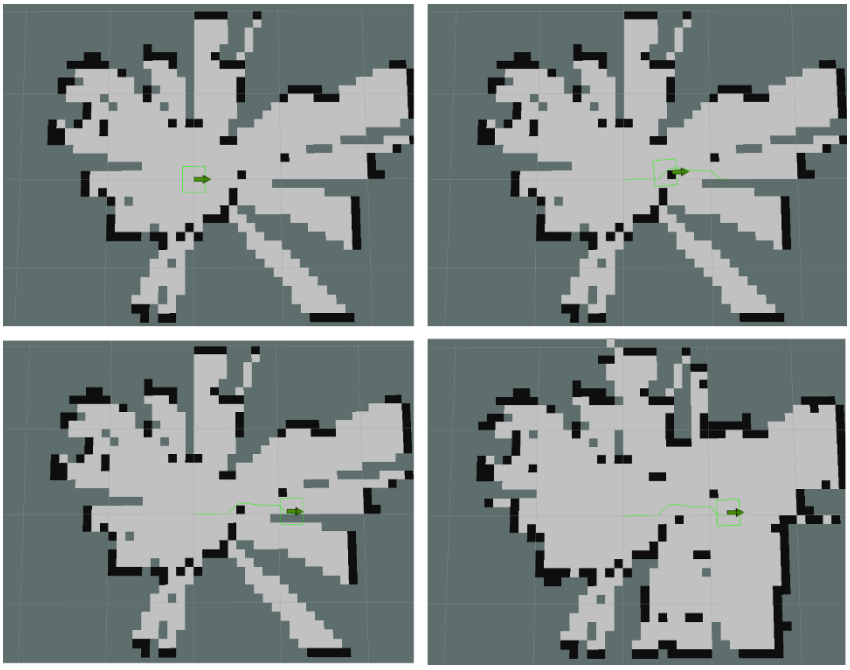


Figura 7-40. Resultado de la prueba 9 con APF.

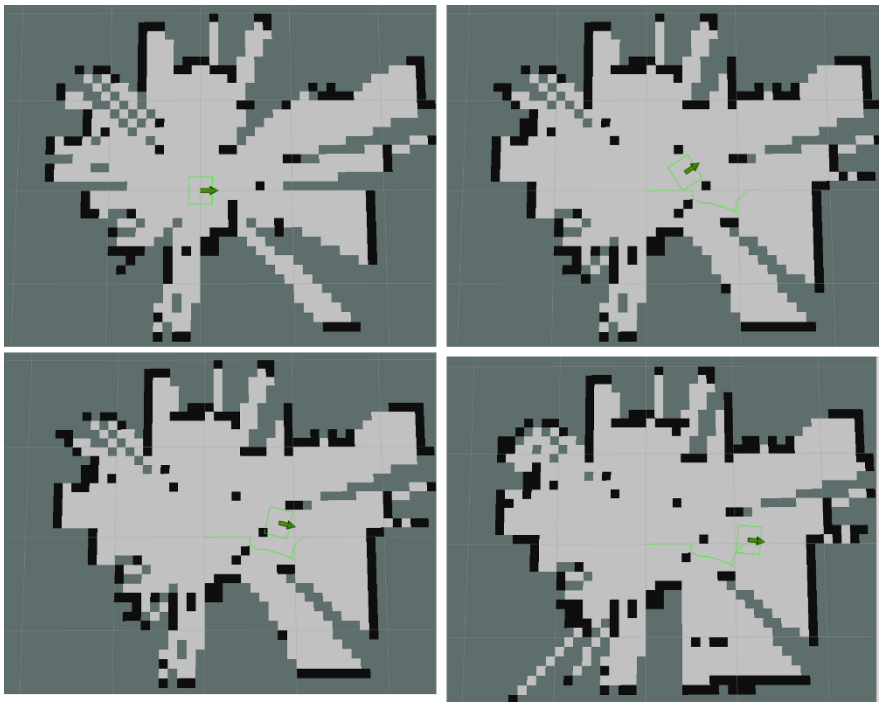


Figura 7-41. Resultado de la prueba 9 con LZS.

APF tarda 7.09s y recorre 1.20m, mientras que LZS tarda 16.69s y recorre 1.32m. APF tiene mejores resultados en este caso para ambas métricas, y esto es parcialmente debido a que su parámetro se modificó a un valor menor (1.3 frente a 2.0 en LZS). Sin embargo, esto no es bueno, pues el robot acaba pasando demasiado cerca de los obstáculos, apenas a uno o dos centímetros. Si bien es cierto que no colisiona, esto no sería aceptable en una situación real.

Por otro lado, LZS obtiene un mejor resultado en cuanto a esto se refiere, ya que su parámetro no se redujo tanto (de 2.8 a 2.0). En conclusión, en un escenario con muchos obstáculos próximos entre ellos es preferible usar el método LZS que el APF.

Vídeo de prueba 9 con APF

Vídeo de prueba 9 con LZS

7.3.10 Prueba 10. Punto destino inalcanzable.

En esta prueba se define un destino inalcanzable, con el fin de comprobar cuánto tiempo dedica cada método a buscar un camino. El punto destino está situado a 1.2m por delante de la posición inicial.

Figura 7-43. Situación inicial de la prueba 10.

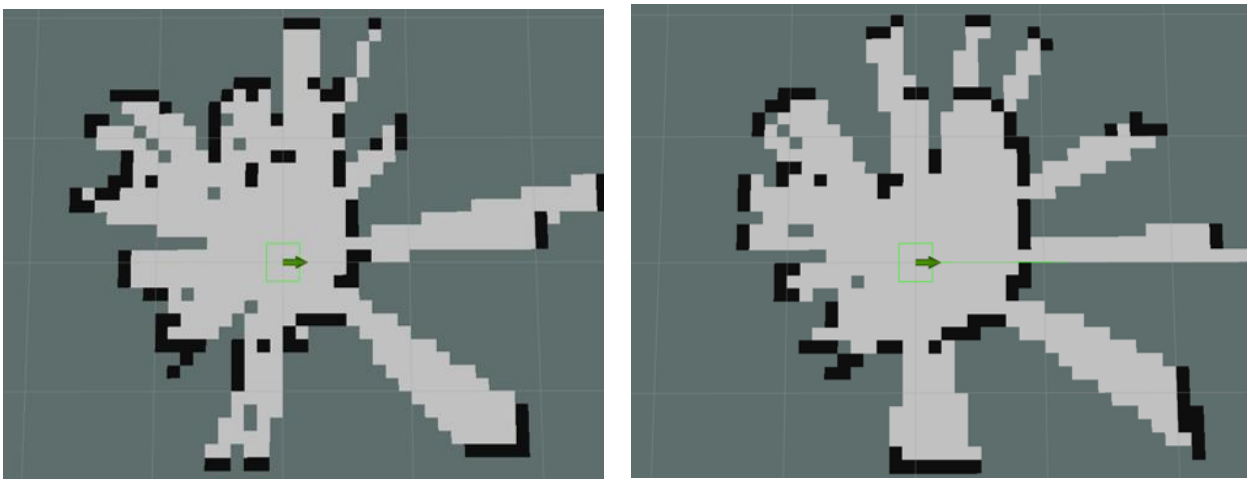


Figura 7-42. A la izquierda el mapa de la prueba 10 con APF, a la derecha con LZS.

Como no se encuentra camino con ningún método, las únicas capturas que se muestran son el mapa inicial obtenido en cada prueba. APF dedicó 1149µs a buscar un camino y LZS 23.101µs. Al igual que en las pruebas de simulación, LZS tarda más en buscar un camino (lo encuentre o no). En este caso del orden de 20 veces más.

Vídeo de prueba 10 con APF

Vídeo de prueba 10 con LZS

7.3.11 Prueba 11. Desplazamiento por mapa desconocido por partes

En esta prueba se muestra de nuevo el problema de no tener un mapa conocido por completo. El punto destino está situado 1.2m hacia la derecha y 0.3m hacia delante del punto inicial.

Figura 7-44. Situación inicial de la prueba 11.

La libreta negra del medio bloquea la visión de las otras dos, por tanto, no se tendrán en cuenta a la hora de buscar una trayectoria válida. Ocurrirá lo mismo que en la prueba 8: el robot colisionará con esos obstáculos desconocidos.

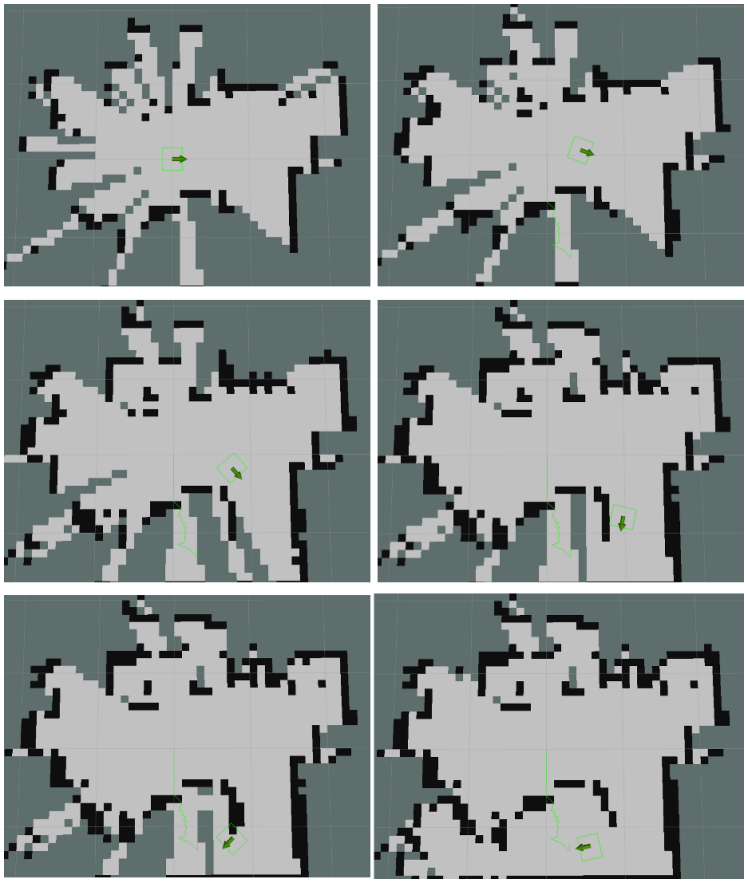


Figura 7-45. Resultado de la prueba 11 con APF.

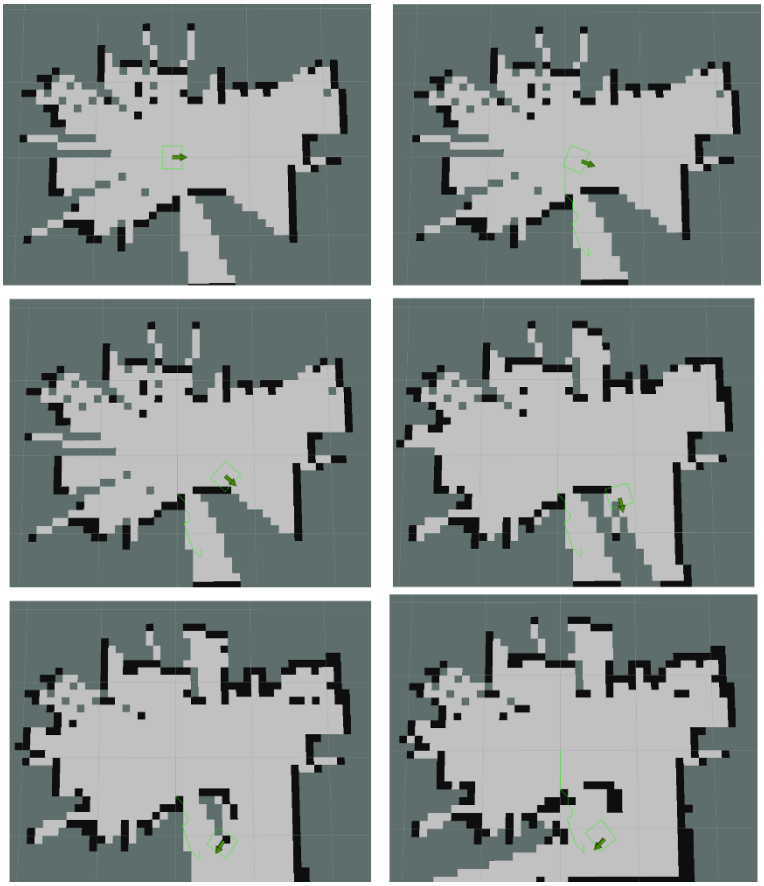


Figura 7-46. Resultado de la prueba 11 con LZS.

Ambos métodos colisionan con los obstáculos como era de esperar. En la primera captura de cada método se aprecia cómo efectivamente esa zona del mapa es desconocida y cuando el robot se desplaza la descubre, pero colisiona igualmente pues este cambio no se tiene en cuenta.

A continuación, se repite la misma prueba, pero en dos fases. Primero, se indica al robot que se desplace a un punto situado a 1.2m delante de la posición inicial (prueba 11.1). Cuando llegue a ese punto, se actualizará el mapa y ya sí se conocerán esos obstáculos que antes no se veían. Luego se manda como punto destino el del principio de la prueba (prueba 11.2) y ya sí se llegará a él correctamente evitando los obstáculos necesarios.

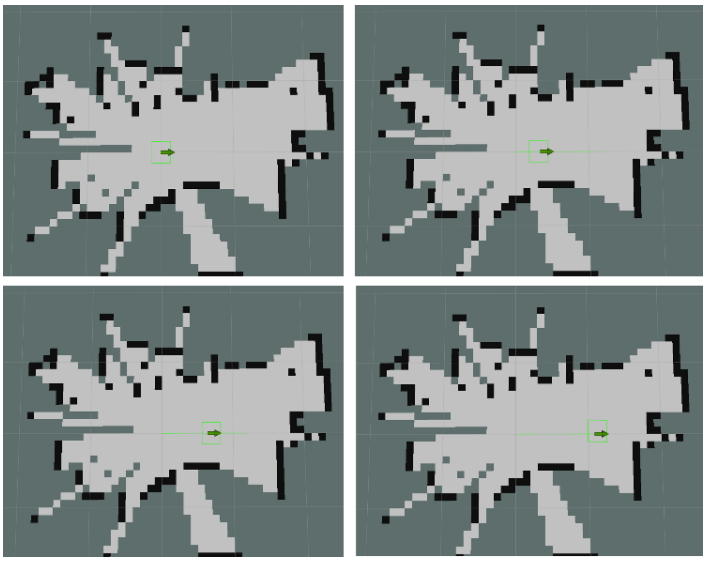


Figura 7-47. Resultado de la prueba 11.1 con APF.

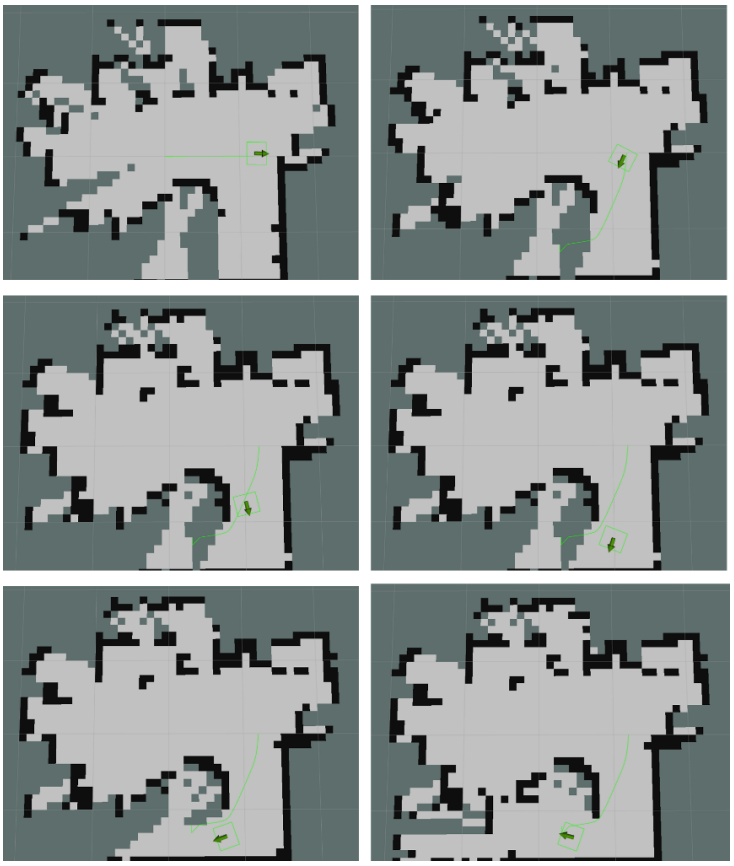


Figura 7-49. Resultado de la prueba 11.2 con APF.

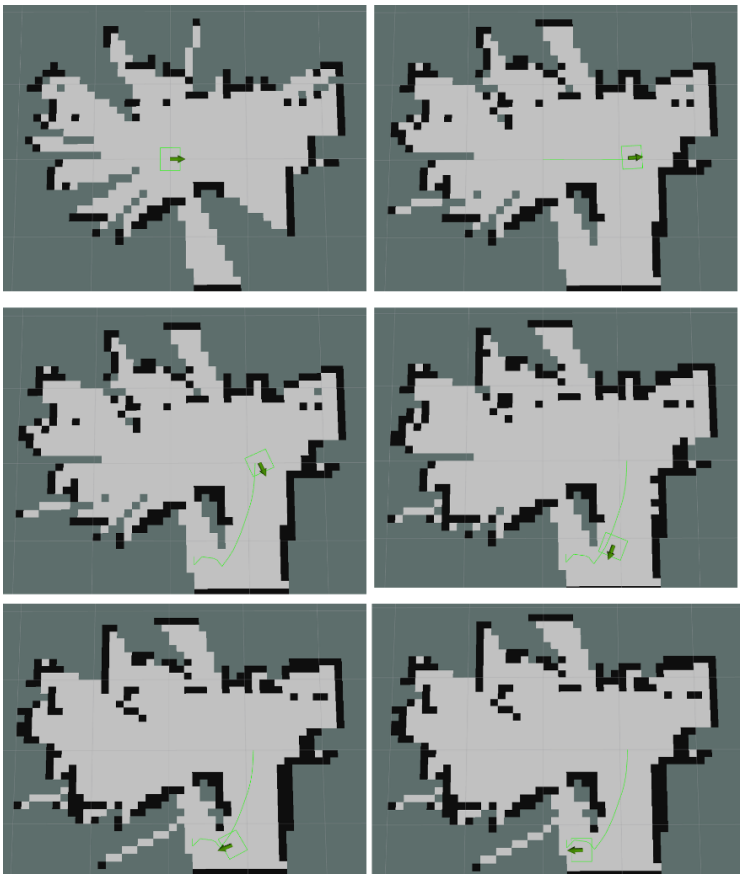


Figura 7-48. Resultado de las pruebas 11.1 y 11.2 con LZS.

Efectivamente, se cumple lo esperado y ahora ambos métodos llegan a su destino correctamente. En la primera parte, APF tarda 6.89s y recorre 1.32m, mientras que LZS tarda 3.09s y recorre 1.38m. En la segunda parte, APF tarda 31.48s y recorre 2.20m, mientras que LZS tarda 17.68s y recorre 2.15m.

La primera corresponde a un simple desplazamiento en línea recta. LZS tarda menos de la mitad de tiempo en llegar que APF, debido a que la trayectoria de APF tiene varios puntos por lo que acelera y frena al pasar por cada uno de ellos. Sin embargo, como la trayectoria de LZS contiene un único punto (el destino) porque hay línea de visión entre origen y destino, no frena y se mueve directamente hacia él.

La segunda parte es la realmente interesante. Nuevamente, LZS tarda prácticamente la mitad de tiempo en llegar al objetivo y lo hace de forma más directa y manteniendo un mayor margen con los obstáculos. Por tanto, este método ha tenido un mejor desempeño en esta prueba.

Vídeo de prueba 11 con APF (en 1 fase)

Vídeo de prueba 11 con LZS (en 1 fase)

Vídeo de prueba 11.1 con APF

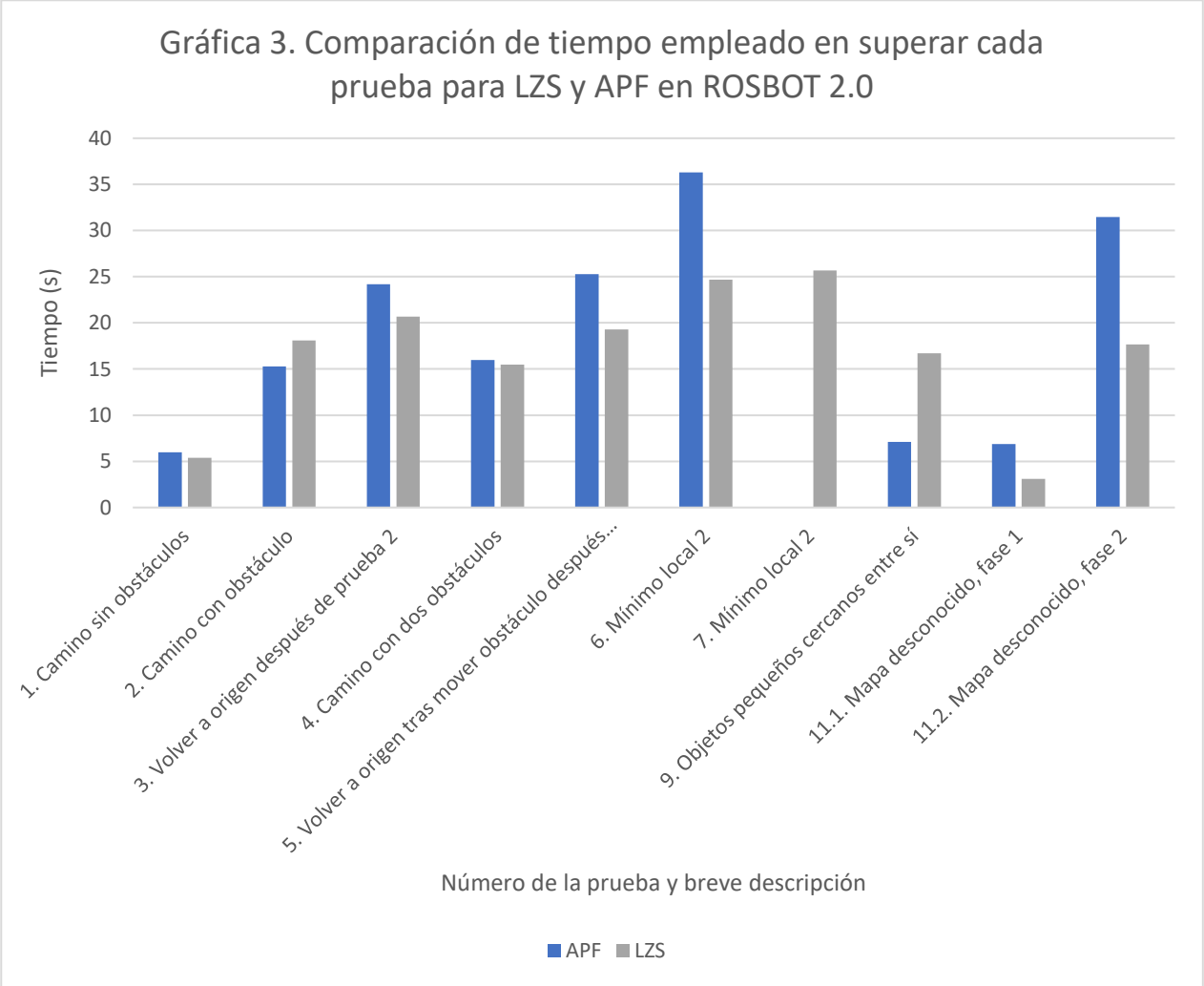
Vídeo de prueba 11.1 con LZS

Vídeo de prueba 11.2 con APF

Vídeo de prueba 11.2 con LZS

7.4 Gráficas de prueba en ROSBOT 2.0

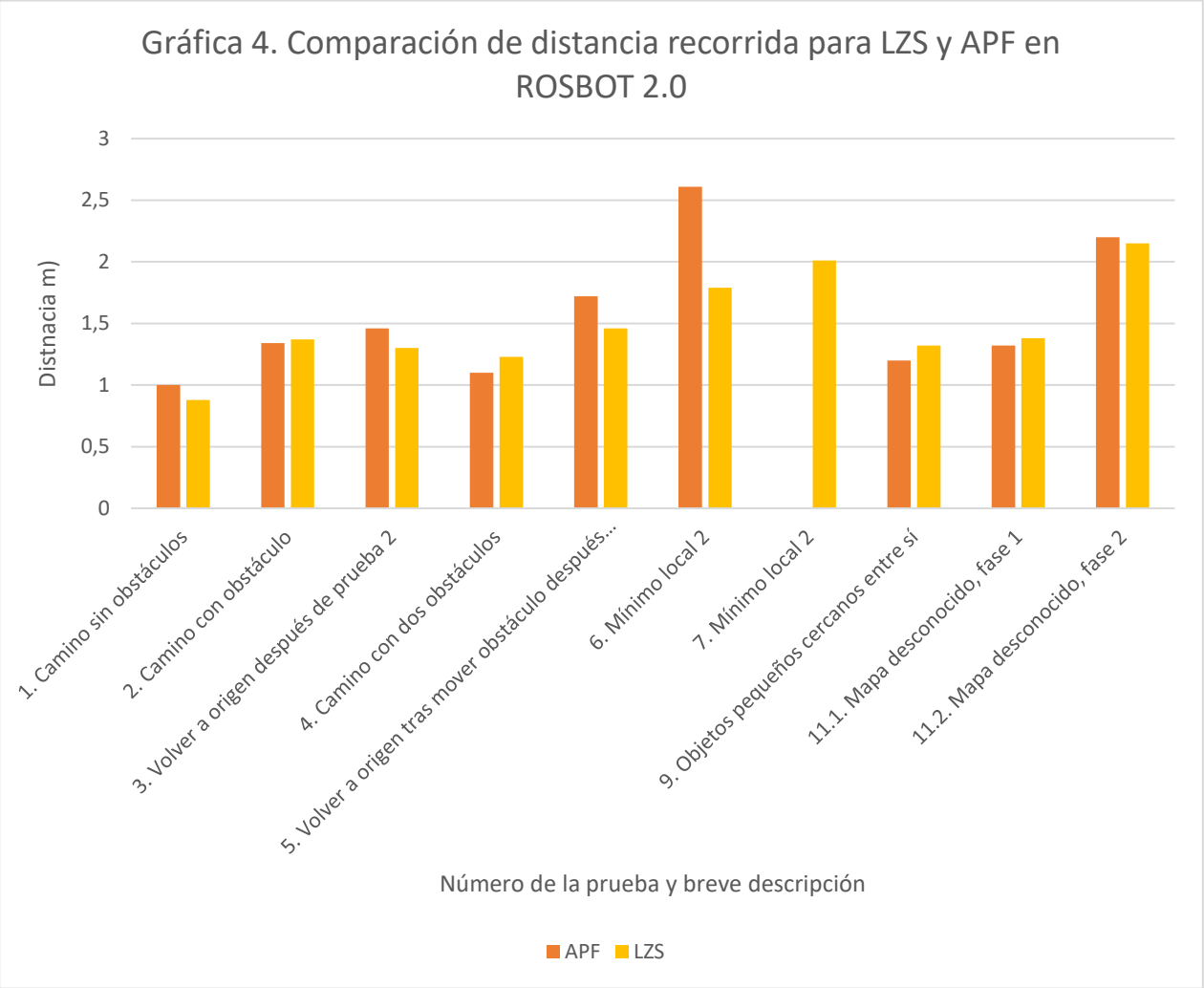
Para visualizar mejor los datos obtenidos en cada prueba, se realizarán varias gráficas.



En la primera gráfica se compara la distancia recorrida por cada método en cada prueba. Se han eliminado las pruebas 8, 10 y 11 porque no sirven los datos obtenidos en ellas bien porque no se llega al objetivo o porque el robot colisiona con algún obstáculo. Además, en la prueba 7 APF no encontró ningún camino, por lo que tampoco se puede comparar.

En general, LZS recorre menos distancia en todas las pruebas menos en la 2 y en la 9, porque APF pasa más cerca de los obstáculos. La gran diferencia de la prueba 9 se debe a que el parámetro de APF se redujo bastante más que para LZS (1.3 frente a 2.0), pero como ya se comentó, provoca que el robot pase demasiado cerca de los obstáculos.

Es destacable también la gran diferencia de tiempo en la última prueba, tanto en la parte recta como en la segunda. De media LZS es 3.04s más rápido que APF, o un 3.14%. Si ignoramos la prueba 9 debido a la diferencia del parámetro entre ambos métodos, LZS es 4.62s más rápido y el porcentaje sube a un 20.47%. Estas cifras son realmente grandes y la diferencia muy notable.



Las diferencias en la distancia recorrida son menores que en el tiempo empleado para llegar al destino, aun así, LZS tiene resultados un poco menores. Destaca la prueba 6, donde APF recorre 82cm más que LZS, debido a ese desplazamiento hacia atrás que hace antes de evitar el mínimo local. La diferencia media de distancia es de 11.89cm y el camino un 4.79% más corto para LZS. Si ignoramos la prueba 9, la diferencia media de distancia recorrida es de 14.88cm, siendo LZS el camino de LZS un 6.65% más corto.

8 CONCLUSIÓN

La navegación autónoma de un robot es un problema muy complejo de resolver, que requiere del trabajo en conjunto de muchas tareas como analizar y obtener información del entorno, buscar trayectorias eficientes que eviten obstáculos o la comunicación con otros dispositivos, entre otras. Afortunadamente, es un problema de creciente interés en todo el mundo, por lo que se está avanzando mucho en él, desarrollando nuevos algoritmos que mejoran los anteriores en todos los aspectos.

Cuando se comenzó a realizar este proyecto apenas se tenían conocimientos de ROS y de C++, lo que suponía un reto extra. Ahora que ya se ha finalizado, se han adquirido numerosos conocimientos en ambas áreas y se ha descubierto el gran potencial que tiene ROS a la hora de trabajar con robots. Es cierto que la curva de aprendizaje es muy lenta y al principio cuesta entender los conceptos básicos y su funcionamiento, pero realmente las posibilidades que ofrece son increíbles.

Por otro lado, trabajar con el robot físico en lugar de con una simulación también ha supuesto un reto, pues aparecen más problemas y aspectos que tratar que en simulación simplemente no existen, como la batería o el hecho de que el robot puede dañarse si se descuida al hacer las pruebas (pues el código puede ser incorrecto y hacer que se mueva sin control y que colisione, por ejemplo).

En cuanto a la primera parte del trabajo, la evitación de colisiones funciona correctamente tanto para obstáculos estáticos como para móviles. Además, el hecho de que se guarde el objetivo previo cuando se para y que se programe automáticamente cuando el obstáculo desaparece del camino es positivo, pues evita tener que hacerlo manualmente, y si el objetivo fue hallado tras pasos previos, evita tener que realizarlos todos de nuevo.

La segunda parte del proyecto es más extensa y ha sido más complicada de implementar que la primera. Tras desarrollar los dos algoritmos de evitación de obstáculos se ha comprobado un funcionamiento básico correcto de ambos, tanto de ‘lazy theta star’ como en ‘campos potenciales artificiales’. En general, LZS es deseable en la mayoría de las situaciones y obtuvo mejores resultados. El principal problema de APF son los mínimos locales, que no siempre se consiguen evitar. Aunque es cierto que existen diversos métodos que resuelven este problema, se prefirió desarrollar uno propio para ver hasta qué punto resolvía el problema.

8.1 Trabajos futuros

Los métodos desarrollados funcionan solo para mapas conocidos completamente y para obstáculos estáticos, aunque bien es cierto que si se mueven antes de buscar una trayectoria sí se evitan correctamente. Se podría realizar una implementación conjunta del detector de colisiones con los algoritmos de evitación de obstáculos para mejorar los métodos y conseguir que el robot evite obstáculos móviles u obstáculos inicialmente desconocidos en el mapa.

También habría que realizar más pruebas para determinar mejor la bondad de los resultados obtenidos. Por ejemplo, no se han probado los algoritmos en espacios muy grandes, aunque en principio no debería haber una gran diferencia en los resultados obtenidos.

Se podrían integrar los métodos desarrollados con un planificador global que sí tenga en cuenta el tamaño del robot. De esta manera el planificador global podría ofrecer una trayectoria inicial que evite obstáculos y el local dedicarse a evitar obstáculos móviles y a recalcular trayectorias si algún camino se ha obstruido.

En cuanto al control de movimiento del robot, se podrían ajustar las constantes proporcionales del control para obtener un movimiento más fluido. También está presente el problema de que en líneas rectas el robot se desplaza acelerando y frenando cada cierta distancia, en lugar de ir directamente hacia el objetivo. Esto sucede para APF, ya que el algoritmo establece un plan con muchos puntos, separados únicamente por una celda en la cuadrícula del mapa. Como el control del robot es proporcional, cuando va a ir al siguiente punto acelera y se va frenando progresivamente hasta que lo alcanza, y cuando selecciona el siguiente punto vuelve acelerar. Esto provoca un movimiento acelerado indeseable en el movimiento. Como trabajo futuro se podría implementar un algoritmo que corrigiera esto, por ejemplo, procesando el plan creado y dejando únicamente los puntos necesarios, eliminando puntos cuando hay visibilidad directa entre el anterior y el siguiente.

REFERENCIAS

[1] Wikipedia, «Wikipedia,» marzo 2021. [En línea]. Available: https://es.wikipedia.org/wiki/Historia_de_los_robots#:~:text=por%20radio%20remotamente.-,A%C3%B1os%201920,checa%20Karel%20%C4%8Capek%20en%201920.&text=%E2%80%8B%20Seg%C3%BAn%20%C4%8Capek%2C%20la%20palabra,conocido%20t%C3%A9rmino%20de%20%22aut%C3%B3mata%22.

[2] V. F. M. Martínez, «Planificación de trayectorias para robots móviles,» 1995.

[3] J. A. Torabuela y A. O. Rodríguez, «Navegación reactiva en entornos estrechos e intricados,» 2014.

[4] W. Burgard, C. Stachniss, K. Arras y M. Bennewitz, «Introduction to Mobile Robotics. SLAM, Simultaneous Localization and Mapping,» 28 junio 2012. [En línea]. Available: http://ais.informatik.uni-freiburg.de/teaching/ss12/robotics/slides/12-slam.pdf.

[5] Ł. Mitka, «SLAM Navigation,» 18 junio 2017. [En línea]. Available: https://husarion.com/tutorials/ros-tutorials/6-slam-navigation/.

[6] H. Durrant-Whyte y T. Bailey, «Simultaneous Localization and Mapping: Part I,» *IEEE Robotics & Automation Magazine,* vol. 13, nº 2, pp. 99-110, 2006.

[7] J. Meyer y S. Kohlbrecher, «Hector SLAM Documentation,» 6 noviembre 2013. [En línea]. Available: http://wiki.ros.org/hector_slam.

[8] C. Stachniss, U. Frese, G. Grisetti y W. Burgard, «OpenSLAM Documentation,» 2 febrero 2014. [En línea]. Available: http://wiki.ros.org/openslam_gmapping.

[9] F. Peralta, M. Arzamendia, D. Gregor, D. G. Reina y S. Toral, «A Comparison of Local Path Planning Techniques of Autonomous Surface Vehicles for Monitoring Applications: The Ypacarai Lake Case-study,» *Sensors*, vol. 20, nº 1488, pp. 1-28, 2020.

[10] GrowingWithTheWeb, «A* Pathfinding Algorithm,» 28 mayo 2016. [En línea]. Available: https://www.growingwiththeweb.com/2012/06/a-pathfinding-algorithm.html.

[11] F. A. Raheem y M. M.Badr, «Development of Modified Path Planning Algorithm Using Artificial Potential Field (APF) Based on PSO for Factors Optimization,» noviembre 2017. [En línea]. Available: https://www.researchgate.net/publication/321597608_Development_of_Modified_Path_Planning_Algorithm_Using_Artificial_Potential_Field_APF_Based_on_PSO_for_Factors_Optimization.

[12] E. Marder-Eppstein, «DWA Documentation,» 28 octubre 2020. [En línea]. Available: http://wiki.ros.org/dwa_local_planner.

[13] A. Salehi, «Rapidly-exploring Random Tree Algorithm,» 28 julio 2015. [En línea]. Available: https://lordhippo.com/professional-work/f180/errt/.

[14] R. H. &. C.-M. Hung, «Pathfinding in 3D Space - A*, Theta*, Lazy Theta* in octree structure,» 8 marzo 2016. [En línea]. Available: https://ascane.github.io/assets/portfolio/pathfinding3d-report.pdf.

[15] ROS, «www.ros.org,» 6 agosto 2018. [En línea]. Available: https://www.ros.org/about-ros/.

[16] Husarion, «Husarion,» 9 2019. [En línea]. Available: https://husarion.com/tutorials/ros-tutorials/7-path-planning/. [Último acceso: 5 2021].

[17] E. Marder-Eppstein, «ROS Documentation,» 27 agosto 2015. [En línea]. Available: http://docs.ros.org/en/hydro/api/nav_core/html/classnav__core_1_1BaseLocalPlanner.html.

[18] Husarion, «Husarion,» 9 2019. [En línea]. Available: https://store.husarion.com/products/rosbot. [Último acceso: 4 2021].

[19] ROS, «ROS Documentation,» 11 2012. [En línea]. Available: http://docs.ros.org/en/melodic/api/sensor_msgs/html/msg/Range.html. [Último acceso: 4 2021].

[20] STMicroelectronics, «ST,» 5 2016. [En línea]. Available: https://www.st.com/resource/en/datasheet/vl53l0x.pdf. [Último acceso: 4 2021].

[21] ROS, «ROS,» 11 2009. [En línea]. Available: http://wiki.ros.org/move_base. [Último acceso: 5 2021].

[22] ROS, «ROS Wiki,» 8 2016. [En línea]. Available: http://wiki.ros.org/actionlib. [Último acceso: 5 2021].

[23] ROS, «ROS.org,» 19 agosto 2015. [En línea]. Available: http://wiki.ros.org/navigation/Tutorials/Writing%20A%20Global%20Path%20Planner%20As%20Plugin%20in%20ROS.

[24] D. C. Shah, «Path Planning for Mobile Robots Using Potential Field Method,» 15 mayo 2018. [En línea]. Available: https://www.researchgate.net/publication/325158411_Path_Planning_for_Mobile_Robots_Using_Potential_Field_Method.

[25] K. Daniel, A. Nash y S. Koenig, «Theta*: Any-Angle Path Planning on Grids,» octubre 2010. [En línea]. Available: https://arxiv.org/ftp/arxiv/papers/1401/1401.3843.pdf.

[26] GeeksforGeeks, «GeeksforGeeks,» 22 marzo 2021. [En línea]. Available: https://www.geeksforgeeks.org/bresenhams-line-generation-algorithm/.

[27] ROS, «ROS Wiki,» 9 2019. [En línea]. Available: https://wiki.ros.org/nav_core. [Último acceso: 5 2021].

[28] ROS, «ROS Wiki,» 9 2019. [En línea]. Available: http://wiki.ros.org/navigation/Tutorials/Writing%20A%20Global%20Path%20Planner%20As%20Plugin%20in%20ROS. [Último acceso: 5 2021].

[29] K. Yamashina, «www.researchgate.com,» agosto 2015. [En línea]. Available: https://www.researchgate.net/figure/Publish-Subscribe-messaging-III-ROS-COMPLIANT-FPGA-COMPONENT-This-section-describes-the_fig1_281395257.

[30] G. Mazzeo y M. Staffa, «Springer,» 24 agosto 2019. [En línea]. Available: https://link.springer.com/article/10.1007/s12369-019-00581-4.

[31] Autor, «Este es el ejemplo de una cita,» *Tesis Doctoral*, vol. 2, nº 13, 2012.

[32] O. Autor, «Otra cita distinta,» *revista*, p. 12, 2001.

Anexo A. Código “no_collision_node.launch”

```
<launch>

   <arg name="use_gazebo" default="false"/>

   <include unless="$(arg use_gazebo)" file="$(find rosbot_ekf)/launch/all.launch"/>
   <include if="$(arg use_gazebo)" file="$(find rosbot_description)/launch/rosbot.launch"/>

   <include file="$(find no_collision)/launch/no_collision_1.launch"/>

</launch>
```

Anexo B. Código “script.cpp”

```
#include <ros/ros.h>
#include <sensor_msgs/Range.h>
#include <geometry_msgs/Twist.h>
#include <std_msgs/Bool.h>
#include <std_srvs/Empty.h>

#include <move_base_msgs/MoveBaseAction.h>
#include <actionlib/client/simple_action_client.h>
#include <geometry_msgs/PoseStamped.h>
using namespace std;

typedef actionlib::SimpleActionClient<move_base_msgs::MoveBaseAction> MoveBaseClient;

ros::Publisher vel_pub;
ros::Publisher photo_pub;
ros::ServiceClient client;
std_srvs::Empty photoSrv;

ros::Publisher goal_pub;
move_base_msgs::MoveBaseGoal goal;
move_base_msgs::MoveBaseGoal prev_goal;

std_msgs::Bool takePhoto;
float dist[] = {0.0, 0.0, 0.0, 0.0}; //FL, FR, RL, RR
geometry_msgs::Twist robotSpeed;

bool goal_cancelledFront = false;
bool goal_cancelledBack = false;

//PARAMETERS
float min_dist;
float correctedMinDist;

void rangeFL_Callback (const sensor_msgs::Range rango) {
  dist[0] = rango.range;
}
void rangeFR_Callback (const sensor_msgs::Range rango) {
  dist[1] = rango.range;
}
void rangeRL_Callback (const sensor_msgs::Range rango) {
  dist[2] = rango.range;
}
```

```
void rangeRR_Callback (const sensor_msgs::Range rango) {
  dist[3] = rango.range;
}

//Si va hacia delante y se va a chocar se para
void vel_Callback (const geometry_msgs::Twist speed) {
  robotSpeed = speed;
}

void output () {
  for (float x : dist) {
    cout << x << " ";
  }
  cout << std::endl;
}

void controlSpeed() {
  correctedMinDist = min_dist * (robotSpeed.linear.x/0.3);
  if ((correctedMinDist < 0.15) (correctedMinDist = 0.15; //límite inferior del parámetro
  if ( (dist[0] < correctedMinDist || dist[1] < correctedMinDist) && ( robotSpeed.linear.x > 0 ) ) { //delante
    if (!goal_cancelledFront) {
      robotSpeed.linear.x = 0;
      robotSpeed.linear.y = 0;
      robotSpeed.linear.z = 0;
      robotSpeed.angular.z = 0;
      // vel_pub.publish(robotSpeed);
      prev_goal = goal;
      goal_cancelledFront = true;
      cout << goal << std::endl;
    }
  } else if (goal_cancelledFront && (dist[0] > correctedMinDist && dist[1] > correctedMinDist)) {
    output();
    goal_cancelledFront = false;
    goal = prev_goal;
    cout << "Camino de delante libre." << std::endl;
  }

  if ( (dist[2] < correctedMinDist || dist[3] < correctedMinDist) && ( robotSpeed.linear.x < 0 ) ) { //atras
    if (!goal_cancelledBack) {
      robotSpeed.linear.x = 0;
      robotSpeed.linear.y = 0;
      robotSpeed.linear.z = 0;
      robotSpeed.angular.z = 0;
      // vel_pub.publish(robotSpeed);
      prev_goal = goal;
      goal_cancelledBack = true;
      cout << goal << std::endl;
    }
  } else if (goal_cancelledBack && !(dist[2] < correctedMinDist || dist[3] < correctedMinDist)) {
    goal_cancelledBack = false;
    goal = prev_goal;
    cout << "Camino de atras libre." << std::endl;
  }

}

void photo_Callback (const std_msgs::Bool takePhoto_C) {
  if (takePhoto_C.data) {
    client.call(photoSrv); //saves the image from the camera to hard drive
    takePhoto.data = false;
    photo_pub.publish(takePhoto);
  }
}

void goal_Callback (const geometry_msgs::PoseStamped goal_C) {
  // goal = goal_C;

  goal.target_pose.pose.position.x = goal_C.pose.position.x;
  goal.target_pose.pose.position.y = goal_C.pose.position.y;
```

```
  goal.target_pose.pose.orientation.w = goal_C.pose.orientation.w;
}

int main(int argc, char **argv)
{
  ros::init(argc, argv, "no_collision");
  ros::NodeHandle n("~");

  MoveBaseClient ac("move_base", true); //spin the thread by default

  //wait for the action server to come up
  while(!ac.waitForServer(ros::Duration(5.0))){
    ROS_INFO("Waiting for the move_base action server to come up");
  }

  ros::Subscriber rangeFL_sub = n.subscribe("/range/fl", 5, rangeFL_Callback);
  ros::Subscriber rangeFR_sub = n.subscribe("/range/fr", 5, rangeFR_Callback);
  ros::Subscriber rangeRL_sub = n.subscribe("/range/rl", 5, rangeRL_Callback);
  ros::Subscriber rangeRR_sub = n.subscribe("/range/rr", 5, rangeRR_Callback);
  ros::Subscriber vel_sub = n.subscribe("/cmd_vel", 1, vel_Callback);
  ros::Subscriber photo_sub = n.subscribe("/take_photo", 1, photo_Callback);

  ros::Subscriber goal_sub = n.subscribe("/move_base_simple/goal", 2, goal_Callback);

  n.param<float>("min_dist", min_dist, 0.25);

  vel_pub = n.advertise<geometry_msgs::Twist>("/cmd_vel", 1);
  photo_pub = n.advertise<std_msgs::Bool>("/take_photo", 1);

  takePhoto.data = false;
  photo_pub.publish(takePhoto);

  client = n.serviceClient<std_srvs::Empty>("/image_saver/save");

  ros::Rate loop_rate(50);


  bool flagCancelled = false;

  while (ros::ok())
  {
    ros::spinOnce();

    //output();
    controlSpeed();
    if ((goal_cancelledFront || goal_cancelledBack) && !flagCancelled) {
      ac.cancelAllGoals();
      flagCancelled = true;
      cout << "Robot detenido porque hay un objeto delante o detras." << std::endl;
    }
    if (!goal_cancelledFront && !goal_cancelledBack && flagCancelled) { //ya no hay obstaculo
      flagCancelled = false;

      goal.target_pose.header.frame_id = "base_link";
      goal.target_pose.header.stamp = ros::Time::now();
      ac.sendGoal(goal);
      cout << "Ya no hay obstáculo, volviendo al objetivo anterior. x: " << goal.target_pose.pose.position.x << ", y: " << goal.target_pose.pose.position.y << ", w:
" << goal.target_pose.pose.orientation.w << std::endl;
    }

    loop_rate.sleep();
  }
}
```

Anexo C. Código “apf_local_planner_plugin.xml”

```
<library path="lib/libapf_local_planner">
<class name ="apf_local_planner/APFLocalPlanner" type ="apf_local_planner::APFLocalPlanner" base_class_type= "nav_core::BaseLocalPlanner">
<description> Plugin for BaseLocalPlanner that implements the Artifitial Potential Field (APF) path planning method.
</description>
</class>
</library>
```

Anexo D. Código “package.xml” del paquete ‘apf_local_planner’

```
<?xml version="1.0"?>
<package format="2">
  <name>apf_local_planner</name>
  <version>1.0.0</version>
  <description>Local_planner package implementing the Artifitial Potential Field path planning method.</description>

  <!-- One maintainer tag required, multiple allowed, one person per tag -->
  <!-- Example:  -->
  <!-- <maintainer email="jane.doe@example.com">Jane Doe</maintainer> -->
  <maintainer email="davidpp00@outlook.com">David</maintainer>


  <!-- One license tag required, multiple allowed, one license per tag -->
  <!-- Commonly used license strings: -->
  <!--   BSD, MIT, Boost Software License, GPLv2, GPLv3, LGPLv2.1, LGPLv3 -->
  <license>TODO</license>


  <!-- Url tags are optional, but multiple are allowed, one per tag -->
  <!-- Optional attribute type can be: website, bugtracker, or repository -->
  <!-- Example: -->
  <!-- <url type="website">http://wiki.ros.org/local_planner</url> -->


  <!-- Author tags are optional, multiple are allowed, one per tag -->
  <!-- Authors do not have to be maintainers, but could be -->
  <!-- Example: -->
   <author email="davidpp00@outlook.com">David</author>


  <!-- The *depend tags are used to specify dependencies -->
  <!-- Dependencies can be catkin packages or system dependencies -->
  <!-- Examples: -->
  <!-- Use depend as a shortcut for packages that are both build and exec dependencies -->
  <!--   <depend>roscpp</depend> -->
  <!--   Note that this is equivalent to the following: -->
  <!--   <build_depend>roscpp</build_depend> -->
  <!--   <exec_depend>roscpp</exec_depend> -->
  <!-- Use build_depend for packages you need at compile time: -->
  <!--   <build_depend>message_generation</build_depend> -->
  <!-- Use build_export_depend for packages you need in order to build against this package: -->
  <!--   <build_export_depend>message_generation</build_export_depend> -->
  <!-- Use buildtool_depend for build tool packages: -->
  <!--   <buildtool_depend>catkin</buildtool_depend> -->
  <!-- Use exec_depend for packages you need at runtime: -->
  <!--   <exec_depend>message_runtime</exec_depend> -->
  <!-- Use test_depend for packages you need only for testing: -->
  <!--   <test_depend>gtest</test_depend> -->
  <!-- Use doc_depend for packages you need only for building documentation: -->
  <!--   <doc_depend>doxygen</doc_depend> -->
  <buildtool_depend>catkin</buildtool_depend>
  <build_depend>nav_core</build_depend>
```

```
  <build_depend>roscpp</build_depend>
  <build_depend>std_msgs</build_depend>

  <build_depend>base_local_planner</build_depend>

  <build_depend>control_toolbox</build_depend>

  <build_depend>cmake_modules</build_depend>

  <build_depend>costmap_2d</build_depend>

  <build_depend>geometry_msgs</build_depend>

  <build_depend>nav_msgs</build_depend>

  <build_depend>pluginlib</build_depend>

  <build_depend>tf</build_depend>

  <build_depend>tf_conversions</build_depend>


  <exec_depend>base_local_planner</exec_depend>

  <exec_depend>control_toolbox</exec_depend>

  <exec_depend>costmap_2d</exec_depend>

  <exec_depend>geometry_msgs</exec_depend>

  <exec_depend>nav_msgs</exec_depend>

  <exec_depend>pluginlib</exec_depend>

  <exec_depend>tf</exec_depend>

  <exec_depend>tf_conversions</exec_depend>

  <build_export_depend>nav_core</build_export_depend>
  <build_export_depend>roscpp</build_export_depend>
  <build_export_depend>std_msgs</build_export_depend>
  <exec_depend>nav_core</exec_depend>
  <exec_depend>roscpp</exec_depend>
  <exec_depend>std_msgs</exec_depend>


  <!-- The export tag contains other, unspecified, tags -->
  <export>
  <nav_core plugin="${prefix}/apf_local_planner_plugin.xml" />
  </export>

</package>
```

Anexo E. Código “apf_local_planner.h”

```
#ifndef APF_LOCAL_PLANNER_H_
#define APF_LOCAL_PLANNER_H_

/** include the libraries you need in your planner here */
/** for local path planner interface */
#include <ros/ros.h>
#include <costmap_2d/costmap_2d_ros.h>
```

```
#include <costmap_2d/costmap_2d.h>

#include <nav_core/base_local_planner.h>
#include <base_local_planner/goal_functions.h>

#include <time.h>

using namespace std;

#include <nav_msgs/Path.h>
#include <nav_msgs/Odometry.h>
#include <nav_msgs/OccupancyGrid.h>
#include <geometry_msgs/PoseStamped.h>
#include <geometry_msgs/PoseWithCovarianceStamped.h>
#include <geometry_msgs/Twist.h>

// transforms
#include <angles/angles.h>
#include <tf/tf.h>
#include <tf/transform_listener.h>

#include <tf2_ros/buffer.h>
// other
#include <array>
#include <vector>
#include <chrono>

using std::string;

#define D2R 0.01745329252 //constant to convert radians to degrees
#define GOAL_TOL 0.10     //distance tolerance to reach goal

#define ANG1 30
#define ANG2 15

//move through the global plan every *this* points, skipping the rest
//this is so that the points for the planner are not very close and useless
#define GLOBAL_WAYPOINT_PATH_STEP 7
#define LOCAL_WAYPOINT_PATH_STEP 3

#define OBS_MASS 120
#define GOAL_MASS 40
#define OBS_RADIUS 2.8 //radio de influencia de cada obstáculo

//número de celdas vecinas que tienen que estar bloqueadas para que se aborte la búsqueda de camino
#define BLOCKED_NEIGHBOUR_CELLS_TO_ABORT_PATH 8

namespace apf_local_planner
{

 struct pos
 {
  double x, y, ang;
 };

 struct cell
 {
  int x, y;
 };

 class APFLocalPlanner : public nav_core::BaseLocalPlanner
 {
 public:
  APFLocalPlanner();
  APFLocalPlanner(std::string name, tf2_ros::Buffer *tf, costmap_2d::Costmap2DROS *costmap_ros);
  /**
   * @brief  Destructor for the wrapper
   */
  ~APFLocalPlanner();
```

```
  /**
  * @brief Initializes the ros wrapper
  * @param name The name to give this instance of the trajectory planner
  * @param tf A pointer to a transform listener
  * @param costmap The cost map to use for assigning costs to trajectories
  */
  void initialize(std::string name, tf2_ros::Buffer *tf, costmap_2d::Costmap2DROS *costmap_ros);

  /**
  * @brief Set the plan that the controller is following; also reset Simple-planner
  * @param orig_global_plan The plan to pass to the controller
  * @return True if the plan was updated successfully, false otherwise
  */
  bool setPlan(const std::vector<geometry_msgs::PoseStamped> &orig_global_plan);

  /**
  * @brief Given the current position, orientation, and velocity of the robot, compute velocity commands to send to the base
  * @param cmd_vel Will be filled with the velocity command to be passed to the robot base
  * @return True if a valid trajectory was found, false otherwise
  */
  bool computeVelocityCommands(geometry_msgs::Twist &cmd_vel);

  /**
  * @brief  Check if the goal pose has been achieved
  * @return True if achieved, false otherwise
  */
  bool isGoalReached();

private:
  ros::Subscriber pose_sub; //pose seubscriber
  ros::Subscriber map_sub; //local map subscriber

  costmap_2d::Costmap2DROS *costmap_ros_; ///<@brief pointer to costmap
  costmap_2d::Costmap2D *costmap_;
  unsigned int sizeX, sizeY; //size of the local map
  double originX, originY, resolution; //parameters of local map

  //POTENTIAL FIELD
  int map[30][30];      //local map
  double delta[30][30]; //gradient for each cell of the map

  tf2_ros::Buffer *tf_; ///<@brief pointer to Transform Listener

  int count, localCount;                        //variables to keep track of which point of each plan the robot is in
  int length, localLength;                       //length in waypoints of global and local plan
  std::vector<geometry_msgs::PoseStamped> globalPlan; // global plan
  std::vector<geometry_msgs::PoseStamped> plan;       //local plan

  geometry_msgs::Twist cmd; // contains the velocity

  bool goal_reached_;
  bool initialized_;
  bool started_;

  double errX, errY, errAng;
  double dist, angOfNextPoint, actAng;

  bool firstTime; //true if it is the first cycle of an execution
  int numCycles; //to calculate average execution time
  double pathLength;          //length of real path followed by robot
  double theoricalPathLength; //length of calculated path
  // double startTime, stopTime;
  std::chrono::_V2::steady_clock::time_point startTime, stopTime; //to measure the time it took the robot to reach the goal

  pos actPose, prevPose; //actual and previous pose
  pos goalPos; //goal position in meters
  bool pathFound; //has a path being found?

  double distGoal; //distance from each cell to the goal cell
```

```
    double distObs; //distance from the robot to each obstacle
    int gx, gy; //map x and y coordinates of goal pose

    bool usingLocalPlan = false;

    std::vector<cell> cellPath, blockedCells;

    cell startCell, goalCell;

    /**
    * @brief Calculates distance and error in position to the next point in the local plan.
    */
    void calcDistAndErr();

    void pose_Callback(const nav_msgs::Odometry::Ptr &poseReceived);

    void map_Callback(const nav_msgs::OccupancyGrid::Ptr &mapReceived);

    void calcPotentialField();

   /**
    * @brief Calculates Euclidean distance between two points.
    */
    double distEuclidean(double xEnd, double yEnd, double xStart, double yStart);

    bool calcPath();

    /**
    * @brief Transforms the path from map cells to world coordinates.
    */
    void transformPath();

    /**
    * @brief Clamps a value between two other values.
    */
    double clamp(double d, double min, double max);

    bool isBlocked(cell c);

    void calcPathPotentialField();

    bool isInPath(int x, int y);
    bool isInVector(cell c, vector<cell> v);

    void printMapWithPath();

    void printMapWithBlockedCells();

    /**
    * @brief Calculates the CMD commands with the implemented control.
    */
    void calculateCMDCommands();

    /**
    * @brief Stops the robot, shows useful information about the path if
    * one was found and resets important variables for the next goal.
    */
    void stopRobotAndShowInfo();

    bool lineOfSight(cell c1, cell c2);

    void showGlobalPlan();
  };
};
#endif
```

Anexo F. Código “apf_local_planner.cpp”

```
#include <pluginlib/class_list_macros.h>
#include "apf_local_planner/apf_local_planner.h"

//register this planner as a BaseLocalPlanner plugin
PLUGINLIB_EXPORT_CLASS(apf_local_planner::APFLocalPlanner, nav_core::BaseLocalPlanner)

using namespace std;

#include <tf2/utils.h>

namespace apf_local_planner
{

 APFLocalPlanner::APFLocalPlanner() : costmap_ros_(NULL), tf_(NULL), initialized_(false) {}

 APFLocalPlanner::APFLocalPlanner(std::string name, tf2_ros::Buffer *tf, costmap_2d::Costmap2DROS *costmap_ros) : costmap_ros_(NULL), tf_(NULL),
initialized_(false)
 {
  initialize(name, tf, costmap_ros);
 }

 APFLocalPlanner::~APFLocalPlanner() {}

 void APFLocalPlanner::initialize(std::string name, tf2_ros::Buffer *tf, costmap_2d::Costmap2DROS *costmap_ros)
 {
  //initialize
  if (!initialized_)
  {
   count = 1;
   localCount = 1;
   firstTime = true;
   numCycles = 0;
   pathLength = 0;
   theoricalPathLength = 0;
   usingLocalPlan = false;

   //inicializo posiciones
   actPose.x = 0;
   actPose.y = 0;
   prevPose.x = 0;
   prevPose.y = 0;
   goalPos.x = 0;
   goalPos.y = 0;

   gx = 0;
   gy = 0;

   ros::NodeHandle n;
   pose_sub = n.subscribe("/odom", 50, &APFLocalPlanner::pose_Callback, this);
   map_sub = n.subscribe("/move_base/local_costmap/costmap", 5, &APFLocalPlanner::map_Callback, this);

   costmap_ros_ = costmap_ros;
   tf_ = tf;

   initialized_ = true;
   ROS_INFO("Artifitial Potential Field planner plugin initialized.");
  }
  else
  {
   ROS_WARN("This planner has already been initialized, doing nothing.");
  }
 }

 bool APFLocalPlanner::setPlan(const std::vector<geometry_msgs::PoseStamped> &orig_global_plan)
 {

  if (!initialized_)
```

```
  {
    ROS_ERROR("This planner has not been initialized, please call initialize() before using this planner");
    return false;
  }

  globalPlan = orig_global_plan;
  plan.clear(); //clear vector to use for APF planning

  length = globalPlan.size();
  cout << "Global plan received with " << length << " waypoints." << endl;

  costmap_ = costmap_ros_->getCostmap();

  sizeX = costmap_->getSizeInCellsX();
  sizeY = costmap_->getSizeInCellsY();
  originX = costmap_->getOriginX();
  originY = costmap_->getOriginY();
  resolution = costmap_->getResolution();

  //inicializo la matriz del gradiente
  for (unsigned int i = 0; i < sizeX; i++)
  {
    for (unsigned int j = 0; j < sizeY; j++)
    {
      delta[i][j] = 0;
    }
  }

  //get local map
  for (unsigned int i = 0; i < sizeX; i++)
  {
    for (unsigned int j = 0; j < sizeY; j++)
    {
      if (costmap_->getCost(i, j) == 0)
      {
        map[i][j] = 0;
      }
      else
      {
        map[i][j] = 1;
      }
    }
  }

  showGlobalPlan();

  //calculate APF to the LAST waypoint in the global plan
  costmap_->worldToMapEnforceBounds(globalPlan[length - 1].pose.position.x, globalPlan[length - 1].pose.position.y, gx, gy);
  calcPathPotentialField();

  goal_reached_ = false;

  return true;
}

bool APFLocalPlanner::computeVelocityCommands(geometry_msgs::Twist &cmd_vel)
{

  if (!initialized_)
  {
    ROS_ERROR("This planner has not been initialized, please call initialize() before using this planner");
    return false;
  }

  if (firstTime)
  {
    startTime = chrono::steady_clock::now();
    firstTime = false;
  }
```

```
if (length != 0)
{
 if (pathFound)
 {
  //move to the global waypoint following all points in local plan
  calcDistAndErr();

  if (dist < GOAL_TOL) //local waypoint reached
  {
   cout << "Local waypoint reached." << endl;

   if (localCount < localLength - 1) //to go to next waypoint in global plan
   {
    // localCount++;
    if ((localLength - 1 - localCount) < LOCAL_WAYPOINT_PATH_STEP + 1)
    {
     localCount = localLength - 1; //selecciono el siguiente punto del plan global al que ir (último punto)
    }
    else
    {
     localCount += LOCAL_WAYPOINT_PATH_STEP;
    }
   }
   else //he llegado al punto del plan global
   {


    if (count < length - 1)
    {
     if ((length - 1 - count) < GLOBAL_WAYPOINT_PATH_STEP + 1)
     {
      count = length - 1; //selecciono el siguiente punto del plan global al que ir (último punto)
      ROS_INFO("going to last global plan point %d", count);
     }
     else
     {
      count += GLOBAL_WAYPOINT_PATH_STEP;
      ROS_INFO("going to global plan point %d", count);
     }
    }
    else
    {
     goal_reached_ = true;
     stopRobotAndShowInfo();
    }
   }
  }
  else //control para ir al waypoint
  {
   calculateCMDCommands();
  }
 }
 else //if a path is not found to the global waypoint: try LZS to final global plan point
 {
  ROS_INFO("Could not find path to global plan point %d", count);

  count = length - 1;

  if (!pathFound) //he encontrado un camino con LZS hasta el último punto del plan global
  {
   goal_reached_ = true; //just to stop execution

   ROS_ERROR("Could not found path to goal...");
   stopRobotAndShowInfo();
  }
  else
  {
   ROS_INFO("Found path to last global waypoint with LZS, following it...");
```

```
      }
    }

    numCycles++;
  }

  // if (length != 0)
  // {
  //   if (pathFound)
  //   {
  //     //move to the global waypoint following all points in local plan
  //     calcDistAndErr(usingLocalPlan); //follow global plan

  //     if (dist < GOAL_TOL) //local waypoint reached
  //     {
  //       if (!usingLocalPlan)
  //       {
  //         cout << "Global waypoint " << count << " reached." << endl;
  //         theoricalPathLength += distEuclidean(globalPlan[count].pose.position.x, globalPlan[count].pose.position.y, globalPlan[count - 1].pose.position.x,
globalPlan[count - 1].pose.position.y);

  //         if (count < length - 1) //to go to next waypoint in global plan
  //         {
  //           //count++;
  //           if ((length - 1 - count) < GLOBAL_WAYPOINT_PATH_STEP + 1)
  //           {
  //             count = length - 1;
  //           }
  //           else
  //           {
  //             count += GLOBAL_WAYPOINT_PATH_STEP;
  //             ROS_INFO("global plan point %d", count);
  //           }

  //           // (gx, gy) is the map cell for next global waypoint
  //           costmap_->worldToMapEnforceBounds(globalPlan[count].pose.position.x, globalPlan[count].pose.position.y, gx, gy);
  //           calcPathPotentialField();
  //         }
  //         else
  //         {
  //           goal_reached_ = true;

  //           stopRobotAndShowInfo();
  //         }
  //       }
  //       else
  //       {
  //         cout << "Local waypoint " << localCount << " reached." << endl;
  //         theoricalPathLength += resolution * distEuclidean(cellPath[localCount].x, cellPath[localCount].y, cellPath[localCount - 1].x, cellPath[localCount -
1].y);

  //         if (localCount < localLength - 1) //to go to next waypoint in local plan
  //         {
  //           localCount++;
  //         }
  //         else //he llegado al final del plan local
  //         {
  //           cout << "Reached last point of the local plan." << endl;
  //           usingLocalPlan = false;
  //         }
  //         // costmap_->worldToMapEnforceBounds(globalPlan[count].pose.position.x, globalPlan[count].pose.position.y, gx, gy);
  //         // calcPathPotentialField();
  //       }
  //     }
  //     else //control para ir al waypoint
  //     {
  //       calculateCMDCommands();
  //     }
  //   }
```

```
    //   else //if a path is not found to the gloabl waypoint: ignore it and try a path to the next one
    //   {
    //     cout << "Could not find APF path to (" << globalPlan[count].pose.position.x << ", " << globalPlan[count].pose.position.y << ") point of global plan." <<
endl;
    //     if (count < length - 1) //to go to next waypoint in global plan
    //     {
    //       if ((length - 1 - count) < GLOBAL_WAYPOINT_PATH_STEP + 1)
    //       {
    //         count = length - 1;
    //       }
    //       else
    //       {
    //         count += GLOBAL_WAYPOINT_PATH_STEP;
    //         ROS_INFO("global plan point %d", count);
    //       }

    //       // (gx, gy) is the map cell for next global waypoint
    //       costmap_->worldToMapEnforceBounds(globalPlan[count].pose.position.x, globalPlan[count].pose.position.y, gx, gy);
    //       calcPathPotentialField();
    //       if (pathFound)
    //       {
    //         usingLocalPlan = true;
    //         cout << "A local plan was found to (" << globalPlan[count].pose.position.x << ", " << globalPlan[count].pose.position.y << ") point of global plan,
with " << localLength << " waypoints." << endl;
    //       }
    //     }
    //     else //goal is unreachable
    //     {
    //       goal_reached_ = true; //just to stop execution

    //       ROS_ERROR("The goal is unreachable...");
    //       stopRobotAndShowInfo();
    //     }
    //   }

    //   numCycles++;
    // }

    //publish data
    cmd_vel = cmd;

    return true;
  }

  bool APFLocalPlanner::isGoalReached()
  {
    if (!initialized_)
    {
      ROS_ERROR("This planner has not been initialized, please call initialize() before using this planner");
      return false;
    }

    return goal_reached_;
  }

  bool APFLocalPlanner::lineOfSight(cell c1, cell c2)
  {
    //BRESENHAM ALGORITHM
    vector<cell> passedCells;
    cell cp;

    int x1 = c1.x;
    int x2 = c2.x;
    int y1 = c1.y;
    int y2 = c2.y;

    bool steep = abs(y2 - y1) > abs(x2 - x1); //si la pendiente de la recta es mayor a 1

    if (steep)
```

```
 {
  swap(x1, y1);
  swap(x2, y2);
 }

 if (x1 > x2)
 {
  swap(x1, x2);
  swap(y1, y2);
 }

 //From this point there are several assumptions:
 // 1) Line is drawn from left to right.
 // 2) x1 < x2 and y1 < y2
 // 3) Slope of the line is between 0 and 1.
 //    We draw a line from lower left to upper
 //    right.

 int deltaY = abs(y2 - y1);
 int deltaX = x2 - x1;
 int error = 0;

 int y = y1;
 int ystep = 1;
 if (y1 > y2)
 {
  ystep = -1;
 }

 for (int x = x1; x < x2 + 1; x++)
 {
  if (steep)
  {
   cp.x = y;
   cp.y = x;
  }
  else
  {
   cp.x = x;
   cp.y = y;
  }
  passedCells.push_back(cp); //cells that the line passes through

  error += deltaY;
  if (2 * error >= deltaX) //hay un cambio en y
  {
   y += ystep;
   error -= deltaX;
  }
 }

 for (auto c : passedCells) //if one of the cells that the line crosses is an obstacle, there is no line of sight
 {
  if (map[c.x][c.y] == 1)
  {
   return false;
  }
 }

 return true;
}

void APFLocalPlanner::stopRobotAndShowInfo()
{
 cmd.linear.x = 0;
 cmd.linear.y = 0; //paro el robot
 cmd.angular.z = 0;

 stopTime = chrono::steady_clock::now();
```

```
 double processingTime = chrono::duration_cast<chrono::milliseconds>(stopTime - startTime).count();

 if (pathFound) //only print useful information if path is found.
 {
  //path duration
  ROS_WARN("Path duration: %.2f seconds.", processingTime / 1000);
  //path length
  ROS_WARN("Calculated path length: %.2f meters.", theoricalPathLength);
  ROS_WARN("Followed path length: %.2f meters.", pathLength);

  //average executation time (for computational cost)
  double execTime = processingTime / numCycles;
  ROS_WARN("Average execution time: %.2f milliseconds/cycle.\n", execTime);
 }

 //Reseteamos las variables de las métricas para poder medir la siguiente prueba.
 numCycles = 0;
 pathLength = 0;
 theoricalPathLength = 0;
 firstTime = true;
 count = 1;
 localCount = 1;
 usingLocalPlan = false;
}

void APFLocalPlanner::calculateCMDCommands()
{
 if (fabs(errAng) > ANG1 * D2R)
 {
  cmd.angular.z = (errAng)*0.3;
  cmd.linear.x = 0.0;
  cmd.linear.y = 0.0;
 }
 else if (fabs(errAng) > ANG2 * D2R)
 {
  cmd.angular.z = (errAng)*0.5;
  cmd.linear.x = 0.05;
  cmd.linear.y = 0.05;
 }
 else
 {
  cmd.linear.x = dist; //límite de velocidad lineal
  if (cmd.linear.x > 0.5)
  {
   cmd.linear.x = 0.5;
  }
  cmd.angular.z = (errAng)*0.75;
 }
}

void APFLocalPlanner::pose_Callback(const nav_msgs::Odometry::Ptr &poseReceived)
{
 prevPose.x = actPose.x;
 prevPose.y = actPose.y;

 actPose.x = poseReceived->pose.pose.position.x;
 actPose.y = poseReceived->pose.pose.position.y;
 actPose.ang = tf2::getYaw(poseReceived->pose.pose.orientation);

 pathLength += sqrt((actPose.x - prevPose.x) * (actPose.x - prevPose.x) + (actPose.y - prevPose.y) * (actPose.y - prevPose.y));
}

//Update Local Map
void APFLocalPlanner::map_Callback(const nav_msgs::OccupancyGrid::Ptr &mapReceived)
{
 std::vector<int8_t> data = mapReceived->data;

 for (unsigned int i = 0; i < sizeX; i++)
 {
```

```
    for (unsigned int j = 0; j < sizeY; j++)
    {
     if (data[i * sizeX + j] == 0)
     {
      map[i][j] = 0;
     }
     else
     {
      map[i][j] = 1;
     }
    }
   }
}

void APFLocalPlanner::calcDistAndErr()
{
 errX = plan[localCount].pose.position.x - actPose.x;
 errY = plan[localCount].pose.position.y - actPose.y;

 actAng = actPose.ang;

 dist = sqrt(errX * errX + errY * errY);
 if (errX == 0 && errY == 0)
 {
  angOfNextPoint = actAng;
 }
 else
 {
  angOfNextPoint = atan2(errY, errX);
 }

 errAng = angOfNextPoint - actAng;

 // make sure that we chose the smallest angle, so that the robot takes the shortest turn
 if (errAng > 180 * D2R)
 {
  errAng -= 360 * D2R;
 }
 if (errAng < -180 * D2R)
 {
  errAng += 360 * D2R;
 }
}

void APFLocalPlanner::calcPotentialField()
{
 //inicializo la matriz del gradiente
 for (unsigned int i = 0; i < sizeX; i++)
 {
  for (unsigned int j = 0; j < sizeY; j++)
  {
   delta[i][j] = 0;
  }
 }

 for (int i = 0; i < sizeX; i++)
 {
  for (int j = 0; j < sizeY; j++)
  {
   //Attraction Force
   distGoal = distEuclidean(gx, gy, i, j); //distance from current cell to goal cell

   delta[i][j] += distGoal * GOAL_MASS; //fuerza mayor cuanto más lejos del destino

   //Repulsive Force
   if (map[i][j] == 1) //si la celda actual es un obstáculo
   {
    int xmin = clamp(i - OBS_RADIUS, 0, sizeX - 1);
    int xmax = clamp(i + OBS_RADIUS, 0, sizeX - 1);
```

```
      int ymin = clamp(j - OBS_RADIUS, 0, sizeY - 1);
      int ymax = clamp(j + OBS_RADIUS, 0, sizeY - 1);

      for (int k = xmin; k <= xmax; k++)
      {
       for (int l = ymin; l <= ymax; l++)
       {
        distObs = distEuclidean(i, j, k, l);

        if (distGoal > distObs) //cuando el objetivo está muy cerca de un obstáculo no se tiene en cuenta el potencial repulsivo
        {
         if (distObs < OBS_RADIUS)
         {
          if (distObs == 0)
          {
           delta[k][l] += OBS_MASS * (OBS_RADIUS - distObs); //quitamos la división porque saldría infinito
          }
          else
          {
           delta[k][l] += OBS_MASS * ((OBS_RADIUS - distObs) / distObs);
          }
         }
        }
       }
      }
     }
    }
   }
}

double APFLocalPlanner::distEuclidean(double xEnd, double yEnd, double xStart, double yStart)
{
 return sqrt((xEnd - xStart) * (xEnd - xStart) + (yEnd - yStart) * (yEnd - yStart));
}

bool APFLocalPlanner::calcPath()
{
 cellPath.clear();
 blockedCells.clear();

 cell startCell = {(int)(sizeX / 2), (int)(sizeY / 2)}; //15, 15 (centro del mapa)
 cell currCell = startCell;
 cell nextCell = startCell;
 cell searchCell = startCell;

 cellPath.push_back(currCell); //añado celda inicial como parte del camino

 cell goalCell = {gx, gy};

 int blockedNeighbourCells = 0;

 while ((currCell.x != goalCell.x) || (currCell.y != goalCell.y))
 {
  int xmin = clamp(currCell.x - 1, 0, sizeX - 1);
  int xmax = clamp(currCell.x + 1, 0, sizeX - 1);
  int ymin = clamp(currCell.y - 1, 0, sizeY - 1);
  int ymax = clamp(currCell.y + 1, 0, sizeY - 1);

  int minForce = 1000000;
  blockedNeighbourCells = 0;

  for (int k = xmin; k <= xmax; k++) //compruebo qué celda vecina tiene menor fuerza resultante
  {
   for (int l = ymin; l <= ymax; l++)
   {
    searchCell = {k, l};

    if (isBlocked(searchCell))
    {
```

```
        blockedNeighbourCells++;
       }
       else
       {
        if (delta[k][l] < minForce)
        {
         minForce = delta[k][l];
         nextCell.x = k;
         nextCell.y = l;
        }
       }
      }
     }

     //si todas las celdas vecinas están bloqueadas, abortar, pues no se ha encontrado ningún camino valído
     if (blockedNeighbourCells == BLOCKED_NEIGHBOUR_CELLS_TO_ABORT_PATH)
     {
      return false; //A path could not be found.
     }

     if (nextCell.x == currCell.x && nextCell.y == currCell.y)
     {
      if (!isBlocked(currCell))
       {

        blockedCells.push_back(currCell);

        currCell = startCell; //we begin again looking for a path
        nextCell = startCell;

        cellPath.clear();
        cellPath.push_back(startCell);
       }
     }
     else
     {
      currCell = nextCell;
      cellPath.push_back(currCell); //añado la nueva celda al camino
     }
    }
    return true;
}

void APFLocalPlanner::transformPath()
{
 plan.clear();  //clear plan
 double wx, wy; //world x & y coordinates
 geometry_msgs::PoseStamped worldPose;

 for (auto c : cellPath)
 {
  costmap_->mapToWorld(c.x, c.y, wx, wy);
  worldPose.pose.position.x = wx;
  worldPose.pose.position.y = wy;
  plan.push_back(worldPose);
 }
}

double APFLocalPlanner::clamp(double d, double min, double max)
{
 const double t = d < min ? min : d;
 return t > max ? max : t;
}

bool APFLocalPlanner::isBlocked(cell c)
{
 for (auto bc : blockedCells)
 {
  if (bc.x == c.x && bc.y == c.y)
```

```
    {
      return true;
    }
  }
  return false;
}

void APFLocalPlanner::calcPathPotentialField()
{
  double startTimePathROS = ros::Time::now().toSec();
  calcPotentialField();
  pathFound = calcPath();
  transformPath(); //if a path is not found it does nothing, so (plan) if empty.
  localLength = plan.size();
  double endTimePathROS = ros::Time::now().toSec();
  double pathCalcTimeROS = (endTimePathROS - startTimePathROS) * 1000000; //time elapsed in milliseconds

  if (pathFound)
  {
    printMapWithPath();
    cout << "Local path found in " << (int)pathCalcTimeROS << " µs with " << localLength << " waypoints." << endl;

    localCount = 1;
    if (localLength == 1) //si solo hay 1 punto en el camino
    {
      localCount = 0;
    }
  }
  else
  {
    // printMapWithBlockedCells();
    ROS_ERROR("A local path could not be found.");
    cout << "Spent " << (int)pathCalcTimeROS << " µs looking for one." << endl;
  }
}

void APFLocalPlanner::printMapWithPath()
{
  cout << "Mapa con camino: " << endl;
  for (int i = 0; i < sizeX; i++)
  {
    for (int j = 0; j < sizeY; j++)
    {
      if (i == gx && j == gy) //goal point
      {
        cout << "Y ";
      }
      else if (i == (int)sizeX / 2 && j == (int)sizeY / 2) //start point
      {
        cout << "X ";
      }
      else if (isInPath(i, j))
      {
        cout << "- ";
      }
      else if (map[i][j] == 0)
      {
        cout << "0 ";
      }
      else
      {
        cout << "1 ";
      }
    }
    cout << endl;
  }
  cout << endl;
}
```

```
bool APFLocalPlanner::isInPath(int x, int y)
{
 for (auto c : cellPath)
 {
  if (x == c.x && y == c.y)
  {
   return true;
  }
 }
 return false;
}

void APFLocalPlanner::printMapWithBlockedCells()
{
 cout << "Mapa con celdas bloqueadas: " << endl;
 for (int i = 0; i < sizeX; i++)
 {
  for (int j = 0; j < sizeY; j++)
  {
   if (i == gx && j == gy)
   {
    cout << "Y ";
   }
   else if (i == (int)sizeX / 2 && j == (int)sizeY / 2) //start point
   {
    cout << "X ";
   }
   else if (isBlocked({i, j}))
   {
    cout << "- ";
   }
   else if (map[i][j] == 0)
   {
    cout << "0 ";
   }
   else
   {
    cout << "1 ";
   }
  }
  cout << endl;
 }
 cout << endl;
}

bool APFLocalPlanner::isInVector(cell c, vector<cell> v)
{
 for (auto cv : v)
 {
  if (c.x == cv.x && c.y == cv.y)
  {
   return true;
  }
 }
 return false;
}

void APFLocalPlanner::showGlobalPlan()
{

 vector<cell> globalPoints;
 cell gc;
 for (auto p : globalPlan)
 {
  costmap_->worldToMapEnforceBounds(p.pose.position.x, p.pose.position.y, gc.x, gc.y);
  globalPoints.push_back(gc);
 }
 cout << "Map w/ global plan: " << endl;
 cout << "global plan has " << globalPoints.size() << " points." << endl;
```

```
  for (int i = 0; i < sizeX; i++)
  {
   for (int j = 0; j < sizeY; j++)
   {
    gc.x = i;
    gc.y = j;

    if (i == globalPoints.back().x && j == globalPoints.back().x) //goal point
    {
     cout << "Y ";
    }
    else if (i == (int)sizeX / 2 && j == (int)sizeY / 2) //start point
    {
     cout << "X ";
    }
    else if (isInVector(gc, globalPoints))
    {
     cout << "- ";
    }
    else if (map[i][j] == 0)
    {
     cout << "0 ";
    }
    else
    {
     cout << "1 ";
    }
   }
   cout << endl;
  }
  cout << "fin mapa global " << endl;
  cout << endl;
 }
};
```

Anexo G. Código “lazy_theta_star_local_planner_plugin.xml”

```
<library path="lib/liblazy_theta_star_local_planner">
<class name ="lazy_theta_star_local_planner/LZSLocalPlanner" type ="lazy_theta_star_local_planner::LZSLocalPlanner" base_class_type=
"nav_core::BaseLocalPlanner">
<description> Plugin for BaseLocalPlanner that implements the Lazy Theta Star (LZS) path planning method.
</description>
</class>
</library>
```

Anexo H. Código “package.xml” del paquete ‘lazy_theta_star_local_planner’

```
<?xml version="1.0"?>
<package format="2">
 <name>lazy_theta_star_local_planner</name>
 <version>1.0.0</version>
 <description>Local_planner package implementing the Lazy Theta Star path planning method.</description>

 <!-- One maintainer tag required, multiple allowed, one person per tag -->
 <!-- Example:  -->
 <!-- <maintainer email="jane.doe@example.com">Jane Doe</maintainer> -->
 <maintainer email="davidpp00@outlook.com">David</maintainer>
```

```
<!-- One license tag required, multiple allowed, one license per tag -->
<!-- Commonly used license strings: -->
<!--   BSD, MIT, Boost Software License, GPLv2, GPLv3, LGPLv2.1, LGPLv3 -->
<license>TODO</license>


<!-- Url tags are optional, but multiple are allowed, one per tag -->
<!-- Optional attribute type can be: website, bugtracker, or repository -->
<!-- Example: -->
<!-- <url type="website">http://wiki.ros.org/local_planner</url> -->


<!-- Author tags are optional, multiple are allowed, one per tag -->
<!-- Authors do not have to be maintainers, but could be -->
<!-- Example: -->
 <author email="davidpp00@outlook.com">David</author>


<!-- The *depend tags are used to specify dependencies -->
<!-- Dependencies can be catkin packages or system dependencies -->
<!-- Examples: -->
<!-- Use depend as a shortcut for packages that are both build and exec dependencies -->
<!--   <depend>roscpp</depend> -->
<!--   Note that this is equivalent to the following: -->
<!--   <build_depend>roscpp</build_depend> -->
<!--   <exec_depend>roscpp</exec_depend> -->
<!-- Use build_depend for packages you need at compile time: -->
<!--   <build_depend>message_generation</build_depend> -->
<!-- Use build_export_depend for packages you need in order to build against this package: -->
<!--   <build_export_depend>message_generation</build_export_depend> -->
<!-- Use buildtool_depend for build tool packages: -->
<!--   <buildtool_depend>catkin</buildtool_depend> -->
<!-- Use exec_depend for packages you need at runtime: -->
<!--   <exec_depend>message_runtime</exec_depend> -->
<!-- Use test_depend for packages you need only for testing: -->
<!--   <test_depend>gtest</test_depend> -->
<!-- Use doc_depend for packages you need only for building documentation: -->
<!--   <doc_depend>doxygen</doc_depend> -->
<buildtool_depend>catkin</buildtool_depend>
<build_depend>nav_core</build_depend>
<build_depend>roscpp</build_depend>
<build_depend>std_msgs</build_depend>

<build_depend>base_local_planner</build_depend>

<build_depend>control_toolbox</build_depend>

<build_depend>cmake_modules</build_depend>

<build_depend>costmap_2d</build_depend>

<build_depend>geometry_msgs</build_depend>

<build_depend>nav_msgs</build_depend>

<build_depend>pluginlib</build_depend>

<build_depend>tf</build_depend>

<build_depend>tf_conversions</build_depend>


<exec_depend>base_local_planner</exec_depend>

<exec_depend>control_toolbox</exec_depend>

<exec_depend>costmap_2d</exec_depend>
```

```
  <exec_depend>geometry_msgs</exec_depend>

  <exec_depend>nav_msgs</exec_depend>

  <exec_depend>pluginlib</exec_depend>

  <exec_depend>tf</exec_depend>

  <exec_depend>tf_conversions</exec_depend>

  <build_export_depend>nav_core</build_export_depend>
  <build_export_depend>roscpp</build_export_depend>
  <build_export_depend>std_msgs</build_export_depend>
  <exec_depend>nav_core</exec_depend>
  <exec_depend>roscpp</exec_depend>
  <exec_depend>std_msgs</exec_depend>


  <!-- The export tag contains other, unspecified, tags -->
  <export>
  <nav_core plugin="${prefix}/lazy_theta_star_local_planner_plugin.xml" />
  </export>

</package>
```

Anexo I. Código “lazy_theta_star_local_planner.h”

```
#ifndef LZS_LOCAL_PLANNER_H_
#define LZS_LOCAL_PLANNER_H_

/** include the libraries you need in your planner here */
/** for local path planner interface */
#include <ros/ros.h>
#include <costmap_2d/costmap_2d_ros.h>
#include <costmap_2d/costmap_2d.h>

#include <nav_core/base_local_planner.h>
#include <base_local_planner/goal_functions.h>

#include <time.h>

using namespace std;

#include <nav_msgs/Path.h>
#include <nav_msgs/Odometry.h>
#include <nav_msgs/OccupancyGrid.h>
#include <geometry_msgs/PoseStamped.h>
#include <geometry_msgs/PoseWithCovarianceStamped.h>
#include <geometry_msgs/Twist.h>

// transforms
#include <angles/angles.h>
#include <tf/tf.h>
#include <tf/transform_listener.h>

#include <tf2_ros/buffer.h>

// other
#include <array>
#include <vector>
```

```
#include <chrono>

using std::string;

#define D2R 0.01745329252 //constant to convert radians to degrees
#define GOAL_TOL 0.10     //distance tolerance to reach goal in meters

#define ANG1 30 //si el error angular es mayor a este valor, el robot solo gira
#define ANG2 15 //si el error angular es mayor a este valor, el robot gira y se desplaza linealmente lentamente

//move through the global plan every *this* points, skipping the rest
//this is so that the points for the planner are not very close and useless
#define GLOBAL_WAYPOINT_PATH_STEP 5

//inflating radius for each obstacle. Equals map resolution * value, in this case 20cm (0.1*2)
//must be enough so that the robot does not collide with nearby obstacles
#define OBS_INFLATING_RADIUS 2.8

namespace lazy_theta_star_local_planner
{

 struct pos
 {
  double x, y, ang;
 };

 struct cell
 {
  int x, y;

  // default + parameterized constructor
  cell(int x = 0, int y = 0)
     : x(x), y(y) {}

  cell &operator=(const cell &a)
  {
   x = a.x;
   y = a.y;
   return *this;
  }

  cell operator+(const cell &a) const
  {
   return cell(a.x + x, a.y + y);
  }

  cell operator-(const cell &a) const
  {
   return cell(x - a.x, y - a.y);
  }

  cell &operator+=(const cell &a)
  {
   x += a.x;
   y += a.y;
   return *this;
  }

  cell &operator-=(const cell &a)
  {
   x -= a.x;
   y -= a.y;
   return *this;
  }

  bool operator==(const cell &a) const
  {
   return (x == a.x && y == a.y);
  }
```

```
};

struct node
{
 cell c;
 cell parentCell;
 double hn;
 double gn;
};

class LZSLocalPlanner : public nav_core::BaseLocalPlanner
{
public:
 LZSLocalPlanner();
 LZSLocalPlanner(std::string name, tf2_ros::Buffer *tf, costmap_2d::Costmap2DROS *costmap_ros);
 /**
  * @brief  Destructor for the wrapper
  */
 ~LZSLocalPlanner();
 /**
 * @brief Initializes the ros wrapper
 * @param name The name to give this instance of the trajectory planner
 * @param tf A pointer to a transform listener
 * @param costmap The cost map to use for assigning costs to trajectories
 */
 void initialize(std::string name, tf2_ros::Buffer *tf, costmap_2d::Costmap2DROS *costmap_ros);

 /**
 * @brief Set the plan that the controller is following; also reset Simple-planner
 * @param orig_global_plan The plan to pass to the controller
 * @return True if the plan was updated successfully, false otherwise
 */
 bool setPlan(const std::vector<geometry_msgs::PoseStamped> &orig_global_plan);

 /**
 * @brief Given the current position, orientation, and velocity of the robot, compute velocity commands to send to the base
 * @param cmd_vel Will be filled with the velocity command to be passed to the robot base
 * @return True if a valid trajectory was found, false otherwise
 */
 bool computeVelocityCommands(geometry_msgs::Twist &cmd_vel);

 /**
 * @brief  Check if the goal pose has been achieved
 * @return True if achieved, false otherwise
 */
 bool isGoalReached();

private:
 ros::Subscriber pose_sub; //pose seubscriber
 ros::Subscriber map_sub;  //local map subscriber

 costmap_2d::Costmap2DROS *costmap_ros_; ///<@brief pointer to costmap
 costmap_2d::Costmap2D *costmap_;
 unsigned int sizeX, sizeY;          //size of the local map
 double originX, originY, resolution; //parameters of local map

 //POTENTIAL FIELD
 int map[30][30];        //local map
 int inflatedMap[30][30]; //local map with obstacles inflated, so that robot does not collide with them

 tf2_ros::Buffer *tf_; ///<@brief pointer to Transform Listener

 int count, localCount;                      //variables to keep track of which point of each plan the robot is in
 int length, localLength;                    //length in waypoints of global and local plan
 std::vector<geometry_msgs::PoseStamped> globalPlan; // global plan
 std::vector<geometry_msgs::PoseStamped> plan;       //local plan

 geometry_msgs::Twist cmd; // contains the velocity
```

```
bool goal_reached_;
bool initialized_;
bool started_;

double errX, errY, errAng;
double dist, angOfNextPoint, actAng;

bool firstTime;           //true if it is the first cycle of an execution
int numCycles;            //to calculate average execution time
double pathLength;        //length of real path followed by robot
double theoricalPathLength; //length of calculated path
// double startTime, stopTime;
std::chrono::_V2::steady_clock::time_point startTime, stopTime; //to measure the time it took the robot to reach the goal

pos actPose, prevPose; //actual and previous pose
pos goalPos;          //goal position in meters
bool pathFound;       //has a path being found?

double distObs; //distance from the robot to each obstacle
int gx, gy;     //map x and y coordinates of goal pose

std::vector<cell> cellPath;

std::vector<node> openList;   //open list
std::vector<node> closedList; //closed list

cell startCell, goalCell;

/**
* @brief Calculates distance and error in position to the next point in the local plan.
*/
void calcDistAndErr();

void pose_Callback(const nav_msgs::Odometry::Ptr &poseReceived);

void map_Callback(const nav_msgs::OccupancyGrid::Ptr &mapReceived);

/**
* @brief Calculates Euclidean distance between two points.
*/
double distEuclidean(double xEnd, double yEnd, double xStart, double yStart);

bool calcPath();

/**
* @brief Transforms the path from map cells to world coordinates.
*/
void transformPath();

/**
* @brief Clamps a value between two other values.
*/
double clamp(double d, double min, double max);

bool isInPath(cell c1);

void printMapWithPath();

/**
* @brief Calculates a local plan with Lazy Theta Star method.
* The path is stored in 'cellPath' vector.
* @return True if a path is found, false otherwise.
*/
bool calcPathLazyThetaStar();
bool lineOfSight(cell c1, cell c2);
void setVertex(node n1);
int isInClosedList(cell c);
int isInOpenList(cell c);
std::vector<cell> calcVecinos(cell c);
```

```
    void inflateObstacles();
    void printInflatedMapWithPath();

    bool isInVector(cell c, vector<cell> v);

    /**
    * @brief Calculates a path with the Lazy Theta Star method.
    * It measures the time taken to do so, even if a path could
    * not be found. Also sets the local plan length variable.
    */
    void calcPathLazyThetaStarWhole();

    /**
    * @brief Calculates the CMD commands with the implemented control.
    */
    void calculateCMDCommands();

    /**
    * @brief Stops the robot, shows useful information about the path if
    * one was found and resets important variables for the next goal.
    */
    void stopRobotAndShowInfo();
  };
};
#endif
```

Anexo J. Código “lazy_theta_star_local_planner.cpp”

```
#include <pluginlib/class_list_macros.h>
#include "lazy_theta_star_local_planner/lazy_theta_star_local_planner.h"

//register this planner as a BaseLocalPlanner plugin
PLUGINLIB_EXPORT_CLASS(lazy_theta_star_local_planner::LZSLocalPlanner, nav_core::BaseLocalPlanner)

using namespace std;

#include <tf2/utils.h>

namespace lazy_theta_star_local_planner
{

  LZSLocalPlanner::LZSLocalPlanner() : costmap_ros_(NULL), tf_(NULL), initialized_(false) {}

  LZSLocalPlanner::LZSLocalPlanner(std::string name, tf2_ros::Buffer *tf, costmap_2d::Costmap2DROS *costmap_ros) : costmap_ros_(NULL), tf_(NULL),
initialized_(false)
  {
    initialize(name, tf, costmap_ros);
  }

  LZSLocalPlanner::~LZSLocalPlanner() {}

  void LZSLocalPlanner::initialize(std::string name, tf2_ros::Buffer *tf, costmap_2d::Costmap2DROS *costmap_ros)
  {
    //initialize
    if (!initialized_)
    {
      count = 1;
      localCount = 1;
      firstTime = true;
      numCycles = 0;
      pathLength = 0;
      theoricalPathLength = 0;

      //inicializo posiciones
```

```
  actPose.x = 0;
  actPose.y = 0;
  prevPose.x = 0;
  prevPose.y = 0;
  goalCell.x = 0;
  goalCell.y = 0;

  gx = 0;
  gy = 0;

  ros::NodeHandle n;
  pose_sub = n.subscribe("/odom", 50, &LZSLocalPlanner::pose_Callback, this);
  map_sub = n.subscribe("/move_base/local_costmap/costmap", 5, &LZSLocalPlanner::map_Callback, this);

  costmap_ros_ = costmap_ros;
  tf_ = tf;

  initialized_ = true;
  ROS_INFO("Lazy Theta Star planner plugin initialized.");
 }
 else
 {
  ROS_WARN("This planner has already been initialized, doing nothing.");
 }
}

bool LZSLocalPlanner::setPlan(const std::vector<geometry_msgs::PoseStamped> &orig_global_plan)
{

 if (!initialized_)
 {
  ROS_ERROR("This planner has not been initialized, please call initialize() before using this planner");
  return false;
 }

 globalPlan = orig_global_plan;
 plan.clear(); //clear vector to use for LZS planning

 length = globalPlan.size();
 cout << "Global plan received with " << length << " waypoints." << endl;

 costmap_ = costmap_ros_->getCostmap();

 sizeX = costmap_->getSizeInCellsX();
 sizeY = costmap_->getSizeInCellsY();
 originX = costmap_->getOriginX();
 originY = costmap_->getOriginY();
 resolution = costmap_->getResolution();

 //get local map
 for (unsigned int i = 0; i < sizeX; i++)
 {
  for (unsigned int j = 0; j < sizeY; j++)
  {
   if (costmap_->getCost(i, j) == 0)
   {
    map[i][j] = 0;
   }
   else
   {
    map[i][j] = 1;
   }
  }
 }

 //calculate LZS to the first waypoint in the global plan
 costmap_->worldToMapEnforceBounds(globalPlan[count].pose.position.x, globalPlan[count].pose.position.y, gx, gy);
 startCell = {(int)sizeX / 2, (int)sizeY / 2}; //set start and goal cells
 goalCell = {gx, gy};
```

```
 calcPathLazyThetaStarWhole();

 goal_reached_ = false;

 return true;
}

bool LZSLocalPlanner::computeVelocityCommands(geometry_msgs::Twist &cmd_vel)
{

 if (!initialized_)
 {
  ROS_ERROR("This planner has not been initialized, please call initialize() before using this planner");
  return false;
 }

 if (firstTime)
 {
  startTime = chrono::steady_clock::now();
  firstTime = false;
 }

 if (length != 0)
 {
  if (pathFound)
  {
   //move to the global waypoint following all points in local plan
   calcDistAndErr();

   if (dist < GOAL_TOL) //local waypoint reached
   {
    cout << "Local waypoint reached." << endl;

    if (localCount < localLength - 1) //to go to next waypoint in global plan
    {
     localCount++;

    }
    else //he llegado al punto del plan global
    {
     if (count < length - 1)
     {
      if ((length - 1 - count) < GLOBAL_WAYPOINT_PATH_STEP + 1)
      {
       count = length - 1; //selecciono el siguiente punto del plan global al que ir (último punto)
       ROS_INFO("going to last global plan point %d", count);
      }
      else
      {
       count += GLOBAL_WAYPOINT_PATH_STEP;
       ROS_INFO("going to global plan point %d", count);
      }

      // (gx, gy) is the map cell for next global waypoint
      costmap_->worldToMapEnforceBounds(globalPlan[count].pose.position.x, globalPlan[count].pose.position.y, gx, gy);
      goalCell = {gx, gy};
      calcPathLazyThetaStarWhole();
     }
     else
     {
      goal_reached_ = true;
      stopRobotAndShowInfo();
     }
    }
   }
   else //control para ir al waypoint
   {
    calculateCMDCommands();
```

```
    }
   }
   else //if a path is not found to the global waypoint: try LZS to final global plan point
   {
    ROS_INFO("Could not find path to global plan point %d", count);

    count = length - 1;

    // (gx, gy) is the map cell for next global waypoint
    costmap_->worldToMapEnforceBounds(globalPlan[count].pose.position.x, globalPlan[count].pose.position.y, gx, gy);
    goalCell = {gx, gy};
    calcPathLazyThetaStarWhole();

    if (!pathFound) //he encontrado un camino con LZS hasta el último punto del plan global
    {
     goal_reached_ = true; //just to stop execution

     ROS_ERROR("Could not found path to goal...");
     stopRobotAndShowInfo();
    }
    else
    {
     ROS_INFO("Found path to last global waypoint with LZS, following it...");
    }
   }

   numCycles++;
  }

  //publish data
  cmd_vel = cmd;

  return true;
}

bool LZSLocalPlanner::isGoalReached()
{
  if (!initialized_)
  {
   ROS_ERROR("This planner has not been initialized, please call initialize() before using this planner");
   return false;
  }

  return goal_reached_;
}

void LZSLocalPlanner::stopRobotAndShowInfo()
{
  cmd.linear.x = 0;
  cmd.linear.y = 0; //paro el robot
  cmd.angular.z = 0;

  stopTime = chrono::steady_clock::now();
  double processingTime = chrono::duration_cast<chrono::milliseconds>(stopTime - startTime).count();

  if (pathFound) //only print useful information if path is found.
  {
   //path duration
   ROS_WARN("Path duration: %.2f seconds.", processingTime / 1000);
   //path length
   ROS_WARN("Theorical path length: %.2f meters.", theoricalPathLength);
   ROS_WARN("Followed path length: %.2f meters.", pathLength);

   //average executation time (for computational cost)
   double execTime = processingTime / numCycles;
   ROS_WARN("Average execution time: %.2f milliseconds/cycle.\n", execTime);
  }

  //Reseteamos las variables de las métricas para poder medir la siguiente prueba.
```

```
 numCycles = 0;
 pathLength = 0;
 theoricalPathLength = 0;
 firstTime = true;
 count = 1;
 localCount = 1;
}

void LZSLocalPlanner::calculateCMDCommands()
{
 if (fabs(errAng) > ANG1 * D2R)
 {
  cmd.angular.z = (errAng)*0.3;
  cmd.linear.x = 0.0;
  cmd.linear.y = 0.0;
 }
 else if (fabs(errAng) > ANG2 * D2R)
 {
  cmd.angular.z = (errAng)*0.5;
  cmd.linear.x = 0.05;
  cmd.linear.y = 0.05;
 }
 else
 {
  cmd.linear.x = dist; //límite de velocidad lineal
  if (cmd.linear.x > 0.5)
  {
   cmd.linear.x = 0.5;
  }
  cmd.angular.z = (errAng)*0.75;
 }
}

void LZSLocalPlanner::pose_Callback(const nav_msgs::Odometry::Ptr &poseReceived)
{
 prevPose.x = actPose.x;
 prevPose.y = actPose.y;

 actPose.x = poseReceived->pose.pose.position.x;
 actPose.y = poseReceived->pose.pose.position.y;
 actPose.ang = tf2::getYaw(poseReceived->pose.pose.orientation);

 pathLength += sqrt((actPose.x - prevPose.x) * (actPose.x - prevPose.x) + (actPose.y - prevPose.y) * (actPose.y - prevPose.y));
}

//Update Local Map
void LZSLocalPlanner::map_Callback(const nav_msgs::OccupancyGrid::Ptr &mapReceived)
{
 // std::vector<int8_t> data = mapReceived->data; //innecesario

 for (unsigned int i = 0; i < sizeX; i++)
 {
  for (unsigned int j = 0; j < sizeY; j++)
  {
   // inflatedMap[i][j] = 0; //initialize the value

   if (mapReceived->data[i * sizeX + j] == 0) //was 'data' before
   {
    map[i][j] = 0;
   }
   else
   {
    map[i][j] = 1;

    // //aprovecho el bucle y creo el mapa inflado aquí
    // int xmin = clamp(i - OBS_INFLATING_RADIUS, 0, sizeX - 1);
    // int xmax = clamp(i + OBS_INFLATING_RADIUS, 0, sizeX - 1);
    // int ymin = clamp(j - OBS_INFLATING_RADIUS, 0, sizeY - 1);
    // int ymax = clamp(j + OBS_INFLATING_RADIUS, 0, sizeY - 1);
```

```
      // for (int k = xmin; k < xmax; k++)
      // {
      //   for (int l = ymin; l < ymax; l++)
      //   {
      //     distObs = distEuclidean(i, j, k, l);

      //     if (distObs < OBS_INFLATING_RADIUS)
      //     {
      //       inflatedMap[k][l] = 1;
      //     }
      //   }
      // }
    }
  }
 }
}

void LZSLocalPlanner::calcDistAndErr()
{
 errX = plan[localCount].pose.position.x - actPose.x;
 errY = plan[localCount].pose.position.y - actPose.y;
 actAng = actPose.ang;

 dist = sqrt(errX * errX + errY * errY);
 if (errX == 0 && errY == 0)
 {
  angOfNextPoint = actAng;
 }
 else
 {
  angOfNextPoint = atan2(errY, errX);
 }

 errAng = angOfNextPoint - actAng;

 // make sure that we chose the smallest angle, so that the robot takes the shortest turn
 if (errAng > 180 * D2R)
 {
  errAng -= 360 * D2R;
 }
 if (errAng < -180 * D2R)
 {
  errAng += 360 * D2R;
 }
}

void LZSLocalPlanner::setVertex(node n1) //return true if parent cell changed, false if there was line of sight
{
 bool isLineOfSight = lineOfSight(n1.parentCell, n1.c); //check visibility between the cell and its parent cell

 if (!isLineOfSight)
 {
  std::vector<cell> vecinos = calcVecinos(n1.c);

  int min_gn = 1000000;
  int closedListIndexMinGn = -1;
  int closedPos = 0;

  vector<cell> vecinosClosed;
  for (auto c : vecinos)
  {
   if (isInClosedList(c) != -1)
   {
    vecinosClosed.push_back(c);
   }
  }
  for (auto c : vecinosClosed)
  {
```

```
      closedPos = isInClosedList(c); //posicion del vecino en la cerrada
      if (closedList[closedPos].gn < min_gn)
      {
        min_gn = closedList[closedPos].gn;
        closedListIndexMinGn = closedPos;
      }
    }

    closedList.back().gn = min_gn + distEuclidean(n1.c.x, n1.c.y, closedList[closedPos].c.x, closedList[closedPos].c.y);
    closedList.back().parentCell = closedList[closedListIndexMinGn].c;
  }
}

bool LZSLocalPlanner::isInVector(cell c, vector<cell> v)
{
  for (auto cv : v)
  {
    if (c.x == cv.x && c.y == cv.y)
    {
      return true;
    }
  }
  return false;
}

bool LZSLocalPlanner::calcPathLazyThetaStar()
{
  cellPath.clear();
  openList.clear();
  closedList.clear();

  bool isPathFound = false;
  int openclose = 0;
  int listPos = 0;

  node initialNode;
  initialNode.c = startCell;
  initialNode.parentCell = startCell; //parent of start cell is itself
  initialNode.hn = distEuclidean(gx, gy, startCell.x, startCell.y);
  initialNode.gn = 0;

  openList.push_back(initialNode);

  while (!isPathFound) //while a path is not found...
  {
    closedList.push_back(openList[0]);
    openList.erase(openList.begin());

    setVertex(closedList.back()); //pasamos el último elemento de la lista cerrada

    int lastCellIndex = isInClosedList(closedList.back().parentCell); //parent MUST be in closed list
    double parentGn = closedList[lastCellIndex].gn;

    if (closedList.back().c == goalCell)
    {
      isPathFound = true;
      continue;
    }
    else
    {
      std::vector<cell> vecinos = calcVecinos(closedList.back().c);

      for (auto c : vecinos) //for each adyacent cell
      {
        int inOpen = isInOpenList(c);
        int inClosed = isInClosedList(c);

        if (inOpen == -1 && inClosed == -1) //if cell not in open or closed list, add it to open list
        {
```

```
          node newCell;
          newCell.c = c;
          newCell.parentCell = closedList.back().parentCell;
          newCell.hn = distEuclidean(c.x, c.y, goalCell.x, goalCell.y);
          newCell.gn = parentGn + distEuclidean(c.x, c.y, closedList.back().parentCell.x, closedList.back().parentCell.y); //parentGn

          openList.push_back(newCell);
         }
        else if (inOpen != -1) //cell is in open list already
        {
         double newGn = closedList.back().gn + distEuclidean(closedList.back().parentCell.x, closedList.back().parentCell.y, openList[inOpen].c.x,
openList[inOpen].c.y);

          if (newGn < openList[inOpen].gn)
          {
           openList[inOpen].gn = newGn;
           openList[inOpen].parentCell = closedList.back().parentCell;
          }
        }
        else if (inClosed != -1) //cell is in closed list already
        {
         double newGn = closedList.back().gn + distEuclidean(closedList.back().parentCell.x, closedList.back().parentCell.y, closedList[inClosed].c.x,
closedList[inClosed].c.y);
          if (newGn < closedList[inClosed].gn)
          {
           closedList[inClosed].gn = newGn;
           closedList[inClosed].parentCell = closedList.back().parentCell;
          }
        }
       }
     }

     //Reordenamos la lista abierta según fn = gn + hn
     if (openList.size() == 0) //if there are no more unexplored cells a path cannot be found (maybe goal cell is too close to an obstacle)
     {
      return false;
     }
     else
     {
      for (int i = 0; i < openList.size() - 1; i++)
      {
       for (int j = 0; j < openList.size() - 1; j++)
       {
        double fn = openList[j].hn + openList[j].gn;
        double fn_1 = openList[j + 1].hn + openList[j + 1].gn;

        if (fn > fn_1)
        {
         node aux = openList[j];
         openList[j] = openList[j + 1];
         openList[j + 1] = aux;
        }
       }
      }
     }
    }

    //FOLLOW PARENT CELLS TO CREATE ACTUAL PATH
    cellPath.clear();
    cell currCell = closedList.back().c;

    cellPath.push_back(currCell); //add to the path the last element of the closed list (Goal Cell)

    bool inStartCell = false;

    while (!inStartCell)
    {
     int closedIndex = isInClosedList(currCell);
     if (currCell == startCell)
```

```
    {
      inStartCell = true;
    }
    else
    {
      currCell = closedList[closedIndex].parentCell;
      cellPath.push_back(currCell);
    }
  }

  //Reverse path
  reverse(cellPath.begin(), cellPath.end());

  return true;
}

bool LZSLocalPlanner::lineOfSight(cell c1, cell c2)
{
  //BRESENHAM ALGORITHM
  vector<cell> passedCells;
  cell cp;

  int x1 = c1.x;
  int x2 = c2.x;
  int y1 = c1.y;
  int y2 = c2.y;

  bool steep = abs(y2 - y1) > abs(x2 - x1); //si la pendiente de la recta es mayor a 1

  if (steep)
  {
    swap(x1, y1);
    swap(x2, y2);
  }

  if (x1 > x2)
  {
    swap(x1, x2);
    swap(y1, y2);
  }

  //From this point there are several assumptions:
  // 1) Line is drawn from left to right.
  // 2) x1 < x2 and y1 < y2
  // 3) Slope of the line is between 0 and 1.
  //    We draw a line from lower left to upper
  //    right.

  int deltaY = abs(y2 - y1);
  int deltaX = x2 - x1;
  int error = 0;

  int y = y1;
  int ystep = 1;
  if (y1 > y2)
  {
    ystep = -1;
  }

  for (int x = x1; x < x2 + 1; x++)
  {
    if (steep)
    {
      cp.x = y;
      cp.y = x;
    }
    else
    {
      cp.x = x;
```

```
      cp.y = y;
    }
    passedCells.push_back(cp); //cells that the line passes through

    error += deltaY;
    if (2 * error >= deltaX) //hay un cambio en y
    {
      y += ystep;
      error -= deltaX;
    }
  }

  for (auto c : passedCells) //if one of the cells that the line crosses is an obstacle, there is no line of sight
  {
    if (inflatedMap[c.x][c.y] == 1)
    {
      return false;
    }
  }

  return true;
}

std::vector<cell> LZSLocalPlanner::calcVecinos(cell c)
{
  std::vector<cell> vecinos;
  vecinos.clear();

  int xmin = clamp(c.x - 1, 0, sizeX - 1);
  int xmax = clamp(c.x + 1, 0, sizeX - 1);
  int ymin = clamp(c.y - 1, 0, sizeY - 1);
  int ymax = clamp(c.y + 1, 0, sizeY - 1);

  for (int k = xmin; k <= xmax; k++)
  {
    for (int l = ymin; l <= ymax; l++)
    {
      if (k == c.x && l == c.y) //skip self cell
      {
        continue;
      }
      else
      {
        if (inflatedMap[k][l] == 0)
        {
          vecinos.push_back({k, l});
        }
      }
    }
  }

  return vecinos;
}

int LZSLocalPlanner::isInOpenList(cell c)
{
  for (int i = 0; i < openList.size(); i++) //for each node in the open list
  {
    if (openList[i].c == c)
    {
      return i;
    }
  }
  return -1;
}

int LZSLocalPlanner::isInClosedList(cell c) //returns index of cell in list, or -1 if not in list
{
  for (int i = 0; i < closedList.size(); i++) //for each node in the closed list
```

```
  {
   if (closedList[i].c == c)
   {
    return i;
   }
  }
 return -1;
}

void LZSLocalPlanner::inflateObstacles()
{
 for (int i = 0; i < sizeX; i++)
 {
  for (int j = 0; j < sizeY; j++)
  {
   inflatedMap[i][j] = 0; //initialize the value

   //Inflate obstacles
   if (map[i][j] == 1) //si la celda actual es un obstáculo
   {
    int xmin = clamp(i - OBS_INFLATING_RADIUS, 0, sizeX - 1);
    int xmax = clamp(i + OBS_INFLATING_RADIUS, 0, sizeX - 1);
    int ymin = clamp(j - OBS_INFLATING_RADIUS, 0, sizeY - 1);
    int ymax = clamp(j + OBS_INFLATING_RADIUS, 0, sizeY - 1);

    for (int k = xmin; k < xmax; k++)
    {
     for (int l = ymin; l < ymax; l++)
     {
      distObs = distEuclidean(i, j, k, l);

      if (distObs < OBS_INFLATING_RADIUS)
      {
       inflatedMap[k][l] = 1;
      }
     }
    }
   }
  }
 }

 inflatedMap[startCell.x - 1][startCell.y - 1] = 0;
 inflatedMap[startCell.x - 1][startCell.y] = 0;
 inflatedMap[startCell.x - 1][startCell.y + 1] = 0;

 inflatedMap[startCell.x][startCell.y - 1] = 0;
 inflatedMap[startCell.x][startCell.y] = 0;
 inflatedMap[startCell.x][startCell.y + 1] = 0;

 inflatedMap[startCell.x + 1][startCell.y - 1] = 0;
 inflatedMap[startCell.x + 1][startCell.y] = 0;
 inflatedMap[startCell.x + 1][startCell.y + 1] = 0;
}

void LZSLocalPlanner::printInflatedMapWithPath()
{
 cell c;
 cout << endl
     << "Inflated map: " << endl;
 for (int i = 0; i < sizeX; i++)
 {
  for (int j = 0; j < sizeY; j++)
  {
   c.x = i;
   c.y = j;

   if (i == goalCell.x && j == goalCell.y)
   {
    cout << "Y ";
```

```
      }
      else if (i == startCell.x && j == startCell.y)
      {
       cout << "X ";
      }
      else if (isInPath(c))
      {
       cout << "- ";
      }
      else if (inflatedMap[i][j] == 0)
      {
       cout << "0 ";
      }
      else
      {
       cout << "1 ";
      }
     }
     cout << endl;
   }
   cout << endl;
}

double LZSLocalPlanner::distEuclidean(double xEnd, double yEnd, double xStart, double yStart)
{
  return sqrt((xEnd - xStart) * (xEnd - xStart) + (yEnd - yStart) * (yEnd - yStart));
}

void LZSLocalPlanner::transformPath()
{
  plan.clear();  //clear plan
  double wx, wy; //world x & y coordinates
  geometry_msgs::PoseStamped worldPose;

  for (auto c : cellPath)
  {
    costmap_->mapToWorld(c.x, c.y, wx, wy);
    worldPose.pose.position.x = wx;
    worldPose.pose.position.y = wy;
    plan.push_back(worldPose);
  }
}

double LZSLocalPlanner::clamp(double d, double min, double max)
{
  const double t = d < min ? min : d;
  return t > max ? max : t;
}

void LZSLocalPlanner::printMapWithPath()
{
  cell c;

  cout << "Mapa con camino: " << endl;
  for (int i = 0; i < sizeX; i++)
  {
    for (int j = 0; j < sizeY; j++)
    {
      c.x = i;
      c.y = j;

      if (i == gx && j == gy) //goal point
      {
       cout << "Y ";
      }
      else if (i == (int)sizeX / 2 && j == (int)sizeY / 2) //start point
      {
       cout << "X ";
      }
```

```
      else if (isInPath(c))
      {
        cout << "- ";
      }
      else if (map[i][j] == 0)
      {
        cout << "0 ";
      }
      else
      {
        cout << "1 ";
      }
    }
    cout << endl;
  }
  cout << endl;
}

bool LZSLocalPlanner::isInPath(cell c1)
{
  for (auto c : cellPath)
  {
    if (c == c1)
    {
      return true;
    }
  }
  return false;
}

void LZSLocalPlanner::calcPathLazyThetaStarWhole()
{
  double startTimePathROS = ros::Time::now().toSec();
  inflateObstacles();
  pathFound = calcPathLazyThetaStar();
  transformPath();
  localLength = plan.size();
  double endTimePathROS = ros::Time::now().toSec();
  double pathCalcTimeROS = (endTimePathROS - startTimePathROS) * 1000000; //time elapsed in milliseconds

  if (pathFound)
  {
    printMapWithPath();
    cout << "Local path found in " << (int)pathCalcTimeROS << " µs with " << localLength << " waypoints." << endl;

    localCount = 1;       //segundo punto del vector de camino
    if (localLength == 1) //si solo hay 1 punto en el camino
    {
      localCount = 0; //primer punto del vector de camino
    }
  }
  else
  {
    // printMapWithBlockedCells();
    ROS_ERROR("A local path could not be found.");
    cout << "Spent " << (int)pathCalcTimeROS << " µs looking for one." << endl;
  }
 }

};
```

Anexo K. Código “simulador_APF.cpp”

```
// other
#include <array>
#include <vector>
#include <math.h>
#include <utility>
#include <string.h>
#include <iostream>
#include <algorithm>
#include <chrono>

using namespace std;

#define D2R 0.01745329252 //constant to convert radians to degrees
#define GOAL_TOL 0.2      //distance tolerance to reach goal

#define OBS_MASS 100
#define GOAL_MASS 40
#define OBS_RADIUS 2.5 //radio de influencia de cada obstáculo

struct pos
{
  double x, y, ang;
};

struct cell
{
  int x, y;
};

const unsigned int sizeX = 30;
const unsigned int sizeY = 30;

//POTENTIAL FIELD
// int map[30][30] = {{}};

//map to test
int map[sizeX][sizeY] = {
    // 0  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29
    // {0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0},  //0
    // {0, 0, 3, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0},  //1
    // {0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0},  //2
    // {0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0},  //3
    // {0, 0, 0, 1, 1, 1, 1, 1, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0},  //4
    // {1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 1, 0, 0, 0, 0, 0},  //5
    // {0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 1, 1, 1, 0, 0, 0},  //6
    // {0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 1, 1, 1, 0, 0, 0},  //7
    // {0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 1, 1, 1, 0, 0, 0},  //8
    // {0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 1, 0, 0, 0, 0, 0},  //9
    // {0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0},  //10
    // {0, 0, 0, 0, 0, 1, 1, 1, 1, 1, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0},  //11
    // {0, 0, 0, 0, 0, 1, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0},  //12
    // {0, 0, 0, 0, 0, 1, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0},  //13
    // {0, 0, 0, 0, 0, 1, 1, 1, 0, 0, 0, 0, 0, 0, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0, 0, 0, 0, 0},  //14
    // {0, 0, 0, 0, 0, 1, 1, 1, 0, 0, 0, 0, 0, 0, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0, 0, 0, 0, 0},  //15
    // {0, 0, 0, 0, 0, 1, 1, 1, 0, 0, 0, 0, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0, 0, 0, 0, 0},  //16
    // {0, 0, 0, 0, 0, 1, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0},  //17
    // {0, 0, 0, 0, 0, 1, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0},  //18
    // {0, 0, 0, 0, 0, 1, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0},  //19
    // {0, 0, 0, 0, 0, 1, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0},  //20
    // {0, 0, 0, 0, 0, 1, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0},  //21
    // {0, 0, 0, 0, 0, 1, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0},  //22
    // {0, 0, 0, 0, 0, 1, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0},  //23
    // {0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0},  //24
    // {0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 2, 0, 0, 0},  //25
    // {0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0},  //26
```

```
    // {0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0},  //27
    // {0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0},  //28
    // {0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0}}; //29

    {1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1},  //1
    {1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 1},  //1
    {1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 1},  //1
    {1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 1},  //1
    {1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 1},  //1
    {1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 1},  //1
    {1, 0, 2, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 1},  //1
    {1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 1},  //1
    {1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 1},  //1
    {1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 1},  //1
    {1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 3, 0, 0, 0, 0, 0, 1},  //1
    {1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 1},  //1
    {1, 1, 1, 1, 1, 1, 1, 1, 0, 0, 0, 1, 1, 1, 1, 1, 1, 1, 0, 0, 0, 1, 0, 0, 0, 1, 1, 1, 1, 1},  //1
    {1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1},  //3
    {1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1},  //3
    {1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1},  //3
    {1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1},  //3
    {1, 1, 1, 1, 1, 1, 1, 1, 0, 0, 0, 1, 1, 1, 1, 1, 1, 1, 0, 0, 0, 1, 0, 0, 0, 1, 1, 1, 1, 1},  //1
    {1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 1},  //3
    {1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 1},  //3
    {1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 1},  //3
    {1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 1},  //3
    {1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 1},  //3
    {1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 1},  //3
    {1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 1},  //3
    {1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 1},  //3
    {1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 1},  //3
    {1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 1},  //27
    {1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 1},  //28
    {1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1}}; //29

// 0 0 1 1 0 1 0 0 0 0 0 0 0 Y 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
// 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 0 0 0 0 0 0 0 0
// 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 1 0 0 0 0 1 0 0 0 0 0 0 0 0 0
// 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 1 1 0 0 0 0 0 0 0 0 0 0 0 0
// 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 1 0 0 0 0 0 0 0 0 0 0 0 0
// 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
// 0 0 1 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 1 1 1 1 0 0 0
// 0 0 1 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 1 1 1 1 1 0 0 0
// 0 0 0 0 0 0 0 0 1 1 1 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 1 0 0 0
// 0 0 0 0 0 0 0 0 1 1 1 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 1 0 0 1
// 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 1 1 0 0
// 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 1 1 0 0
// 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0
// 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 1 0 0
// 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0
// 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 0
// 0 1 0 0 0 1 1 1 1 0 1 1 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 0
// 1 1 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 1 1 0
// 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 0 0 0 0 0 0 0 0 0 0 0 0 1 0
// 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 0 0 0 0 0 0 0 0 0 0 0 0 1 0
// 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 1 1 0 0 0 0 0 0 0 0 0 0 1 1
// 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 1 1 0 0 0 0 0 0 0 0 1 1 1 1
// 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 1 1 1 1 1 1 1 1 1 1 0 0
// 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 1 1 0 0 0 0 0 0 0 0 0
// 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
// 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
// 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
// 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
// 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
// 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0

pos startPos, goalPos;
// pos startPos = {13, 9};
// pos goalPos = {2, 9};
```

```
double deltaX[sizeX][sizeY]; //x gradient for each cell

double distGoal;
double distObs;

std::vector<cell> cellPath;
std::vector<cell> blockedCells;

cell goalCell, startCell;

double pathLength = 0;

void calcPotentialField();

double distEuclidean(double xEnd, double yEnd, double xStart, double yStart);

bool calcPath();

void printMap();

void printMapWithPath();

void printPath();

bool isInPath(int x, int y);

bool isBlocked(cell c);

void printForce();

void printMapWithBlockedCells();

void checkStartAndGoalCells();

double clamp(double d, double min, double max);

int main(int argc, char **argv)
{
 checkStartAndGoalCells();
 goalCell = {(int)goalPos.x, (int)goalPos.y};
 startCell = {(int)startPos.x, (int)startPos.y};

 printf("De celda (%d, %d) a (%d, %d)\n", startCell.x, startCell.y, goalCell.x, goalCell.y);

 auto startTime = chrono::steady_clock::now();
 calcPotentialField();
 bool pathCalculated = calcPath();
 auto endTime = chrono::steady_clock::now(); //std::chrono::_V2::steady_clock::time_point

 auto processingTime = chrono::duration_cast<chrono::microseconds>(endTime - startTime).count();

 printMapWithPath();

 // if (pathCalculated)
 //   cout << "Found path in " << processingTime << " µs" << endl;
 // else
 //   cout << "A path could not be found." << endl
 //        << "Time elapsed: " << processingTime << " µs" << endl;

 if (pathCalculated)
 {
  cout << "Camino encontrado en " << processingTime << " µs" << endl;

  for (int i = 0; i < cellPath.size() - 1; i++)
  {
   pathLength += distEuclidean(cellPath[i].x, cellPath[i].y, cellPath[i + 1].x, cellPath[i + 1].y);
  }
  cout << "Longitud del camino: " << pathLength << " m. " << endl;
```

```
  }
  else
  {
   printMapWithBlockedCells();

   cout << "No ha sido posible encontrar un camino." << endl;
   cout << "Tiempo transcurrido buscando: " << processingTime << " µs" << endl;
  }

  cout << endl
      << endl;
}

void calcPotentialField()
{
  int gx, gy; //map x and y coordinates of goal pose
  gx = goalPos.x;
  gy = goalPos.y;

  for (int i = 0; i < sizeX; i++)
  {
   for (int j = 0; j < sizeY; j++)
   {
    //Attraction Force
    distGoal = distEuclidean(gx, gy, i, j); //distance from current cell to goal cell

    deltaX[i][j] += distGoal * GOAL_MASS; //fuerza mayor cuanto más lejos del destino

    //Repulsive Force
    if (map[i][j] == 1) //si la celda actual es un obstáculo
    {
     int xmin = clamp(i - OBS_RADIUS, 0, sizeX);
     int xmax = clamp(i + OBS_RADIUS, 0, sizeX);
     int ymin = clamp(j - OBS_RADIUS, 0, sizeY);
     int ymax = clamp(j + OBS_RADIUS, 0, sizeY);

     for (int k = xmin; k < xmax; k++)
     {
      for (int l = ymin; l < ymax; l++)
      {
       distObs = distEuclidean(i, j, k, l);

       if (distGoal > distObs)
       {
        if (distObs < OBS_RADIUS)
        {
         if (distObs == 0)
         {
          deltaX[k][l] += OBS_MASS * (OBS_RADIUS - distObs);
         }
         else
         {
          deltaX[k][l] += OBS_MASS * ((OBS_RADIUS - distObs) / distObs);
         }
        }
       }
      }
     }
    }
   }
  }
}

double distEuclidean(double xEnd, double yEnd, double xStart, double yStart)
{
  return std::sqrt((xEnd - xStart) * (xEnd - xStart) + (yEnd - yStart) * (yEnd - yStart));
}

bool calcPath()
```

```
{
 cellPath.clear();
 blockedCells.clear();

 cell startCell = {(int)startPos.x, (int)startPos.y}; //15, 15 (centro del mapa)
 cell currCell = startCell;
 cell nextCell = startCell;
 cell searchCell = startCell;

 cellPath.push_back(currCell); //añado celda inicial como parte del camino

 cell goalCell = {(int)goalPos.x, (int)goalPos.y};

 int blockedNeighbourCells = 0;

 while ((currCell.x != goalCell.x) || (currCell.y != goalCell.y))
 {
  int xmin = clamp(currCell.x - 1, 0, sizeX - 1);
  int xmax = clamp(currCell.x + 1, 0, sizeX - 1);
  int ymin = clamp(currCell.y - 1, 0, sizeY - 1);
  int ymax = clamp(currCell.y + 1, 0, sizeY - 1);

  int minForce = 1000000;
  blockedNeighbourCells = 0;

  for (int k = xmin; k <= xmax; k++) //compruebo qué celda vecina tiene menor fuerza resultante
  {
   for (int l = ymin; l <= ymax; l++)
   {
    searchCell = {k, l};

    if (isBlocked(searchCell))
    {
     blockedNeighbourCells++;
    }
    else
    {
     // if (deltaX[k][l] < minForce && !isBlocked(searchCell))
     if (deltaX[k][l] < minForce)
     {

      minForce = deltaX[k][l];
      nextCell.x = k;
      nextCell.y = l;
      // printf("Next cell is (%d, %d) \n", nextCell.y, nextCell.x);
     }
    }
   }
  }
  if (blockedNeighbourCells == 8)
  {
   // cout << "A path could not be found." << endl;
   printMapWithBlockedCells();
   return false;
  }

  if (nextCell.x == currCell.x && nextCell.y == currCell.y)
  {
   if (!isBlocked(currCell))
   {

    blockedCells.push_back(currCell);
    // printf("Blocked (%d, %d)\n", currCell.y, currCell.x);

    currCell = startCell; //we begin again looking for a path
    nextCell = startCell;

    cellPath.clear();
    cellPath.push_back(startCell);
```

```
      }
    }
    else
    {
      currCell = nextCell;
      cellPath.push_back(currCell); //añado la nueva celda al camino
      // printf("Current cell is (%d, %d) \n", currCell.y, currCell.x);
    }
  }
  return true;
}

bool isBlocked(cell c)
{
  for (auto bc : blockedCells)
  {
    if (bc.x == c.x && bc.y == c.y)
    {
      return true;
    }
  }
  return false;
}

void printMap()
{
  cout << "Mapa: " << endl;
  for (int i = 0; i < sizeX; i++)
  {
    for (int j = 0; j < sizeY; j++)
    {
      if (i == goalPos.x && j == goalPos.y)
      {
        cout << "Y ";
      }
      else if (i == startPos.x && j == startPos.y)
      {
        cout << "X ";
      }
      else if (map[i][j] == 0)
      {
        cout << "0 ";
      }
      else
      {
        cout << "1 ";
      }
    }
    cout << endl;
  }
  cout << endl;
}

void printMapWithPath()
{
  cout << "Mapa con camino: " << endl;
  for (int i = 0; i < sizeX; i++)
  {
    for (int j = 0; j < sizeY; j++)
    {
      if (i == goalPos.x && j == goalPos.y)
      {
        cout << "Y ";
      }
      else if (i == startPos.x && j == startPos.y)
      {
        cout << "X ";
      }
      else if (isInPath(i, j))
```

```
      {
        cout << "+ ";
      }
      else if (map[i][j] == 0)
      {
        cout << "0 ";
      }
      else
      {
        cout << "1 ";
      }
    }
    cout << endl;
  }
  cout << endl;
}

bool isInPath(int x, int y)
{
  for (auto c : cellPath)
  {
    if (x == c.x && y == c.y)
    {
      return true;
    }
  }
  return false;
}

void printPath()
{
  cout << "Camino obtenido, longitud: " << cellPath.size() << " celdas. " << endl;

  for (auto c : cellPath)
  {
    cout << "(" << c.x << ", " << c.y << ")" << endl;
  }
  cout << endl;
}

void printForce()
{
  cout << "Fuerzas: " << endl;
  for (int i = 0; i < sizeX; i++)
  {
    for (int j = 0; j < sizeY; j++)
    {
      printf("%.2f ", deltaX[i][j]);
    }
    cout << endl;
  }
  cout << endl;
}

double clamp(double d, double min, double max) //clamps a value between two values
{
  const double t = d < min ? min : d;
  return t > max ? max : t;
}

void printMapWithBlockedCells()
{
  cout << "Mapa con celdas bloqueadas: " << endl;
  for (int i = 0; i < sizeX; i++)
  {
    for (int j = 0; j < sizeY; j++)
    {
      if (i == goalPos.x && j == goalPos.y)
      {
```

```
      cout << "Y ";
    }
    else if (i == startPos.x && j == startPos.y)
    {
      cout << "X ";
    }
    else if (isBlocked({i, j}))
    {
      cout << "- ";
    }
    else if (map[i][j] == 0)
    {
      cout << "0 ";
    }
    else
    {
      cout << "1 ";
    }
   }
   cout << endl;
  }
  cout << endl;
}

void checkStartAndGoalCells()
{
  for (int i = 0; i < sizeX; i++)
  {
    for (int j = 0; j < sizeY; j++)
    {
      if (map[i][j] == 2)
      {
        startPos.x = i;
        startPos.y = j;
      }
      else if (map[i][j] == 3)
      {
        goalPos.x = i;
        goalPos.y = j;
      }
    }
  }
}
```

Anexo L. Código “simulador_LZS.cpp”

```
// other
#include <array>
#include <vector>
#include <math.h>
#include <utility>
#include <string.h>
#include <iostream>
#include <algorithm>
#include <chrono>

using namespace std;

#define D2R 0.01745329252 //constant to convert radians to degrees
#define GOAL_TOL 0.1      //distance tolerance to reach goal

//inflating radius for each obstacle. Equals map resolution * value, in this case 20cm (0.1*2)
```

```
//must be enough so that the robot does not collide with nearby obstacles
#define OBS_INFLATING_RADIUS 2.5

struct pos
{
 double x, y, ang;
};

struct cell
{
 int x, y;

 // default + parameterized constructor
 cell(int x = 0, int y = 0)
    : x(x), y(y) {}

 cell &operator=(const cell &a)
 {
  x = a.x;
  y = a.y;
  return *this;
 }

 cell operator+(const cell &a) const
 {
  return cell(a.x + x, a.y + y);
 }

 cell operator-(const cell &a) const
 {
  return cell(x - a.x, y - a.y);
 }

 cell &operator+=(const cell &a)
 {
  x += a.x;
  y += a.y;
  return *this;
 }

 cell &operator-=(const cell &a)
 {
  x -= a.x;
  y -= a.y;
  return *this;
 }

 bool operator==(const cell &a) const
 {
  return (x == a.x && y == a.y);
 }
};

struct node
{
 cell c;
 cell parentCell;
 double hn;
 double gn;
};

const unsigned int sizeX = 30;
const unsigned int sizeY = 30;

//POTENTIAL FIELD

//map to test
int map[sizeX][sizeY] = {
   // 0  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29
```

```
    // {0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0},  //0
    // {0, 0, 3, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0},  //1
    // {0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0},  //2
    // {0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0},  //3
    // {0, 0, 0, 1, 1, 1, 1, 1, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0},  //4
    // {1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 1, 0, 0, 0, 0},  //5
    // {0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 1, 1, 1, 0, 0, 0},  //6
    // {0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 1, 1, 1, 0, 0, 0},  //7
    // {0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 1, 1, 1, 0, 0, 0},  //8
    // {0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 1, 0, 0, 0, 0},  //9
    // {0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0},  //10
    // {0, 0, 0, 0, 0, 1, 1, 1, 1, 1, 1, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0},  //11
    // {0, 0, 0, 0, 0, 1, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0},  //12
    // {0, 0, 0, 0, 0, 1, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0},  //13
    // {0, 0, 0, 0, 0, 1, 1, 1, 0, 0, 0, 0, 0, 0, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0, 0, 0},  //14
    // {0, 0, 0, 0, 0, 1, 1, 1, 0, 0, 0, 0, 0, 0, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0, 0, 0},  //15
    // {0, 0, 0, 0, 0, 1, 1, 1, 0, 0, 0, 0, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0, 0, 0},  //16
    // {0, 0, 0, 0, 0, 1, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0},  //17
    // {0, 0, 0, 0, 0, 1, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0},  //18
    // {0, 0, 0, 0, 0, 1, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0},  //19
    // {0, 0, 0, 0, 0, 1, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0},  //20
    // {0, 0, 0, 0, 0, 1, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0},  //21
    // {0, 0, 0, 0, 0, 1, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0},  //22
    // {0, 0, 0, 0, 0, 1, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0},  //23
    // {0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0},  //24
    // {0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 2, 0, 0},  //25
    // {0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0},  //26
    // {0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0},  //27
    // {0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0},  //28
    // {0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0}}; //29

     {1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1},  //1
     {1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 1},  //1
     {1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 1},  //1
     {1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 1},  //1
     {1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 1},  //1
     {1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 1},  //1
     {1, 0, 2, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 1},  //1
     {1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 1},  //1
     {1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 1},  //1
     {1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 1},  //1
     {1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 3, 0, 0, 0, 0, 0, 1},  //1
     {1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 1},  //1
     {1, 1, 1, 1, 1, 1, 1, 1, 0, 0, 0, 1, 1, 1, 1, 1, 1, 1, 0, 0, 0, 1, 0, 0, 0, 1, 1, 1, 1, 1},  //1
     {1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1},  //3
     {1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1},  //3
     {1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1},  //3
     {1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1},  //3
     {1, 1, 1, 1, 1, 1, 1, 1, 0, 0, 0, 1, 1, 1, 1, 1, 1, 1, 0, 0, 0, 1, 0, 0, 0, 1, 1, 1, 1, 1},  //1
     {1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 1},  //3
     {1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 1},  //3
     {1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 1},  //3
     {1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 1},  //3
     {1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 1},  //3
     {1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 1},  //3
     {1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 1},  //3
     {1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 1},  //3
     {1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 1},  //3
     {1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 1},  //27
     {1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 1},  //28
     {1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1}}; //29

int inflatedMap[sizeX][sizeY];

pos startPos, goalPos;
cell goalCell, startCell;

double distObs;
```

```
std::vector<cell> cellPath;

std::vector<node> openList;
std::vector<node> closedList;

bool pathCalculated = false;

double pathLength = 0;

double distEuclidean(double xEnd, double yEnd, double xStart, double yStart);
void checkStartAndGoalCells();
double clamp(double d, double min, double max);

void printPath();
void printMapWithPath();
bool isInPath(cell c1);

bool calcPathLazyThetaStar();         //OK
bool lineOfSight(cell c1, cell c2);   //OK
void setVertex(node n1);              //OK
int isInClosedList(cell c);           //OK
int isInOpenList(cell c);             //OK
std::vector<cell> calcVecinos(cell c); //OK
void inflateObstacles();              //OK
void printInflatedMapWithPath();      //OK

int main(int argc, char **argv)
{
 checkStartAndGoalCells();
 goalCell = {(int)goalPos.x, (int)goalPos.y};
 startCell = {(int)startPos.x, (int)startPos.y};

 printf("De celda (%d, %d) a (%d, %d)\n", startCell.x, startCell.y, goalCell.x, goalCell.y);

 auto startTime = chrono::steady_clock::now();

 //ENLARGE OBSTACLES SO THAT ROBOT DOESN'T COLLIDE
 inflateObstacles();
 pathCalculated = calcPathLazyThetaStar();
 // bool pathCalculated = false;

 auto endTime = chrono::steady_clock::now();
 auto processingTime = chrono::duration_cast<chrono::microseconds>(endTime - startTime).count();

 // printInflatedMapWithPath();
 printMapWithPath();
 // printPath();

 if (pathCalculated)
 {
  cout << "Camino encontrado en " << processingTime << " µs" << endl;

  for (int i = 0; i < cellPath.size() - 1; i++)
  {
   pathLength += distEuclidean(cellPath[i].x, cellPath[i].y, cellPath[i + 1].x, cellPath[i + 1].y);
  }
  cout << "Longitud del camino: " << pathLength << " m. " << endl;
 }
 else
 {
  cout << "No ha sido posible encontrar un camino." << endl;
  cout << "Tiempo transcurrido buscando: " << processingTime << " µs" << endl;
 }

 cout << endl
     << endl;
}

double distEuclidean(double xEnd, double yEnd, double xStart, double yStart)
```

```
{
 return sqrt((xEnd - xStart) * (xEnd - xStart) + (yEnd - yStart) * (yEnd - yStart));
}

void checkStartAndGoalCells()
{
 for (int i = 0; i < sizeX; i++)
 {
  for (int j = 0; j < sizeY; j++)
  {
   if (map[i][j] == 2)
   {
    startPos.x = i;
    startPos.y = j;
   }
   else if (map[i][j] == 3)
   {
    goalPos.x = i;
    goalPos.y = j;
   }
  }
 }
}

void setVertex(node n1) //return true if parent cell changed, false if there was line of sight
{
 bool isLineOfSight = lineOfSight(n1.parentCell, n1.c); //check visibility between the cell and its parent cell

 if (!isLineOfSight)
 {
  // printf("No line of sight between (%d, %d) and (%d, %d).\n", n1.parentCell.x, n1.parentCell.y, n1.c.x, n1.c.y);
  std::vector<cell> vecinos = calcVecinos(n1.c);

  int min_gn = 1000000;
  int closedListIndexMinGn = -1;
  int closedPos = 0;

  // for (auto c : vecinos) //por cada celda vecina compruebo cuáles están en la lista cerrada
  // {
  //   closedPos = isInClosedList(c); //index or -1 if not in list
  //   if (closedPos != -1)          //cell c is in closed list
  //   {
  //     if (closedList[closedPos].gn < min_gn)
  //     {
  //       min_gn = closedList[closedPos].gn;
  //       closedListIndex = closedPos;
  //     }
  //   }
  // }
  vector<cell> vecinosClosed;
  for (auto c : vecinos)
  {
   if (isInClosedList(c) != -1)
   {
    vecinosClosed.push_back(c);
   }
  }
  for (auto c : vecinosClosed)
  {
   closedPos = isInClosedList(c); //posicion del vecino en la cerrada
   if (closedList[closedPos].gn < min_gn)
   {
    min_gn = closedList[closedPos].gn;
    closedListIndexMinGn = closedPos;
   }
  }

  closedList.back().gn = min_gn + distEuclidean(n1.c.x, n1.c.y, closedList[closedPos].c.x, closedList[closedPos].c.y);
  // closedList.back().gn = min_gn + 1;
```

```
    // closedList.back().gn = min_gn + distEuclidean(n1.c.x, n1.c.y, closedList[closedPos].c.x, closedList[closedPos].c.y);
    closedList.back().parentCell = closedList[closedListIndexMinGn].c;

    // printf("Parent of (%d, %d) has been set to (%d, %d).\n", closedList.back().c.x, closedList.back().c.y, closedList.back().parentCell.x,
closedList.back().parentCell.y);
    // cout << "Changed some parent, closed list: " << endl;
    // for (auto c : closedList)
    // {
    //   printf("(%d, %d) with parent (%d, %d)\n", c.c.x, c.c.y, c.parentCell.x, c.parentCell.y);
    // }
   }
   else
   {
    // printf("Line of sight between (%d, %d) and (%d, %d)!\n", n1.parentCell.x, n1.parentCell.y, n1.c.x, n1.c.y);
   }
}

bool calcPathLazyThetaStar()
{
  cellPath.clear();
  openList.clear();
  closedList.clear();

  bool isPathFound = false;
  int openclose = 0;
  int listPos = 0;

  node initialNode;
  initialNode.c = {(int)startPos.x, (int)startPos.y};
  initialNode.parentCell = {(int)startPos.x, (int)startPos.y}; //parent of start cell is itself
  initialNode.hn = distEuclidean(goalPos.x, goalPos.y, startPos.x, startPos.y);
  initialNode.gn = 0;

  openList.push_back(initialNode);

  while (!isPathFound) //while a path is not found...
  {
   closedList.push_back(openList[0]);
   openList.erase(openList.begin());
   // cout << endl
   //      << "openlist length is " << openList.size() << endl;
   // cout << "closedlist length is " << closedList.size() << endl;
   // printf("Added (%d, %d) to closed list.\n", closedList.back().c.x, closedList.back().c.y);

   setVertex(closedList.back()); //pasamos el último elemento de la lista cerrada

   int lastCellIndex = isInClosedList(closedList.back().parentCell); //parent MUST be in closed list
   // printf("Parent of (%d, %d) is in index %d in closed list.\n", closedList.back().c.x, closedList.back().c.y, lastCellIndex);
   double parentGn = closedList[lastCellIndex].gn;
   // cout << "parent gn is: " << parentGn << endl;

   if (closedList.back().c == goalCell) //PATH FOUND!
   // if (isInClosedList(goalCell) != -1)
   {
    isPathFound = true;
    // cout << "Path found!" << endl;
    continue;
   }
   else
   {
    std::vector<cell> vecinos = calcVecinos(closedList.back().c);

    for (auto c : vecinos) //for each adyacent cell
    {
     int inOpen = isInOpenList(c);
     int inClosed = isInClosedList(c);
     // printf("openList index: %d, closed list index: %d\n", inOpen, inClosed);

     if (inOpen == -1 && inClosed == -1) //if cell not in open or closed list, add it to open list
```

```
      {
       // printf("Vecino (%d, %d) ni en open ni en closed\n", c.x, c.y);
       node newCell;
       newCell.c = c;
       newCell.parentCell = closedList.back().parentCell;
       newCell.hn = distEuclidean(c.x, c.y, goalCell.x, goalCell.y);
       newCell.gn = parentGn + distEuclidean(c.x, c.y, closedList.back().parentCell.x, closedList.back().parentCell.y); //parentGn

       openList.push_back(newCell);
      }
      else if (inOpen != -1) //cell is in open list already
      {
       // printf("Vecino (%d, %d) en open\n", c.x, c.y);
       double newGn = closedList.back().gn + distEuclidean(closedList.back().parentCell.x, closedList.back().parentCell.y, openList[inOpen].c.x,
openList[inOpen].c.y); // + 1
       // OLD OLD double newGn = closedList.back().gn + distEuclidean(closedList.back().c.x, closedList.back().c.y, openList[inOpen].c.x,
openList[inOpen].c.y); // + 1

       // double newGn = closedList.back().gn + 1;
       if (newGn < openList[inOpen].gn)
       {
        openList[inOpen].gn = newGn;
        openList[inOpen].parentCell = closedList.back().parentCell;
       }
      }
      else if (inClosed != -1) //cell is in closed list already
      {
       // printf("Vecino (%d, %d) en closed\n", c.x, c.y);
       double newGn = closedList.back().gn + distEuclidean(closedList.back().parentCell.x, closedList.back().parentCell.y, closedList[inClosed].c.x,
closedList[inClosed].c.y); // +1
       // double newGn = closedList.back().gn + distEuclidean(closedList.back().c.x, closedList.back().c.y, openList[inOpen].c.x, openList[inOpen].c.y); // + 1
       // double newGn = closedList.back().gn + 1;
       if (newGn < closedList[inClosed].gn)
       {
        closedList[inClosed].gn = newGn;
        closedList[inClosed].parentCell = closedList.back().parentCell;

        //move cell to end of closed list
        // cout << "-- closedlist length is " << closedList.size() << endl;
        // node lastC = closedList[inClosed];              //      <<<<<<<<<<<<<<<       ------ <<<<<<<<<<
        // closedList.erase(closedList.begin() + inClosed - 1); //not sure if this works  //                    IT WAS UNCOMMENTED
        // closedList.push_back(lastC);                    //      <<<<<<<<<<<<<<<       ------ <<<<<<<<<<
        // cout << "--same? closedlist length is " << closedList.size() << endl;
       }
      }
     }
    }

    //Reordenamos la lista abierta según fn = gn + hn
    if (openList.size() == 0)
    {
     // cout << "Lista abierta vacia....." << endl;
     // isPathFound = true; //to end loop
     // cout << "A path could not be found..." << endl;
     return false;
    }
    else
    {
     for (int i = 0; i < openList.size() - 1; i++)
     {
      for (int j = 0; j < openList.size() - 1; j++)
      {
       double fn = openList[j].hn + openList[j].gn;
       double fn_1 = openList[j + 1].hn + openList[j + 1].gn;

       if (fn > fn_1)
       {
        node aux = openList[j];
        openList[j] = openList[j + 1];
```

```
          openList[j + 1] = aux;
        }
       }
      }
     }
    }

    //FOLLOW PARENT CELLS TO CREATE ACTUAL PATH

    cellPath.clear();
    cell currCell = closedList.back().c;

    cellPath.push_back(currCell); //add to the path the last element of the closed list (Goal Cell)

    bool inStartCell = false;

    while (!inStartCell)
    {
     int closedIndex = isInClosedList(currCell);
     // if (closedIndex == -1)
     // {
     //   // a path has been found but the first cell in path is not in closed list, dont know why ?
     //   break;
     // }
     if (currCell == startCell)
     {
      inStartCell = true;
     }
     else
     {
      currCell = closedList[closedIndex].parentCell;
      cellPath.push_back(currCell);
     }
     // cout << "waiting here..." << endl;
    }

    //Reverse path
    reverse(cellPath.begin(), cellPath.end());

    return true;
}

bool lineOfSight(cell c1, cell c2) //iterative method to know if there is visibility between two given cells
{
    // cell dir = c2 - c1;  //direction vector from cell c1 to cell c2 (we treat the cell as an int vector here)
    // pos vPos;           //vector de posición de
    // vPos.x = c1.x + 0.5; //we add 0.5 to correct point position vs square cell representation
    // vPos.y = c1.y + 0.5;

    // double distLS = distEuclidean(c1.x, c1.y, c2.x, c2.y);
    // double step = 0.05;
    // unsigned int nIter = distLS / step;

    // for (size_t i = 0; i < nIter; i++)
    // {
    //   vPos.x += (float)dir.x * step;
    //   vPos.y += (float)dir.y * step;
    //   cell vPos2Cell = {(int)vPos.x, (int)vPos.y};

    //   if (inflatedMap[vPos2Cell.x][vPos2Cell.y] == 1) //if current cell position is an obstacle      //use map for normal local map with no inflation
    //     return false;

    //   if (vPos2Cell == c2) //si he llegado a la celda c2 es que sí hay visibilidad
    //     break;
    // }

    // return true;

    //BRESENHAM ALGORITHM
```

```
vector<cell> passedCells;
cell cp;

int x1 = c1.x;
int x2 = c2.x;
int y1 = c1.y;
int y2 = c2.y;

bool steep = abs(y2 - y1) > abs(x2 - x1); //si la pendiente de la recta es mayor a 1

if (steep)
{
 swap(x1, y1);
 swap(x2, y2);
}

if (x1 > x2)
{
 swap(x1, x2);
 swap(y1, y2);
}

// From this point there are several assumptions:
// 1) Line is drawn from left to right.
// 2) x1 < x2 and y1 < y2
// 3) Slope of the line is between 0 and 1.
//    We draw a line from lower left to upper
//    right.

int deltaY = abs(y2 - y1);
int deltaX = x2 - x1;
int error = 0;

int y = y1;
int ystep = 1;
if (y1 > y2)
{
 ystep = -1;
}

for (int x = x1; x < x2 + 1; x++)
{
 if (steep)
 {
  cp.x = y;
  cp.y = x;
 }
 else
 {
  cp.x = x;
  cp.y = y;
 }
 passedCells.push_back(cp);

 error += deltaY;
 if (2 * error >= deltaX) //hay un cambio en y
 {
  y += ystep;
  error -= deltaX;
 }
}

for (auto c : passedCells)
{
 if (inflatedMap[c.x][c.y] == 1)
 {
  return false;
 }
}
```

```
 return true;
}

std::vector<cell> calcVecinos(cell c)
{
 std::vector<cell> vecinos;
 vecinos.clear();

 int xmin = clamp(c.x - 1, 0, sizeX - 1);
 int xmax = clamp(c.x + 1, 0, sizeX - 1);
 int ymin = clamp(c.y - 1, 0, sizeY - 1);
 int ymax = clamp(c.y + 1, 0, sizeY - 1);

 for (int k = xmin; k <= xmax; k++)
 {
  for (int l = ymin; l <= ymax; l++)
  {
   if (k == c.x && l == c.y) //skip self cell
   {
    continue;
   }
   else
   {
    if (inflatedMap[k][l] == 0 || inflatedMap[k][l] == 3)
    {
     vecinos.push_back({k, l});
    }
   }
  }
 }

 return vecinos;
}

double clamp(double d, double min, double max) //clamps a value between two values
{
 const double t = d < min ? min : d;
 return t > max ? max : t;
}

int isInClosedList(cell c) //returns index of cell in list, or -1 if not in list
{
 for (int i = 0; i < closedList.size(); i++) //for each node in the closed list
 {
  if (closedList[i].c == c)
  {
   return i;
  }
 }
 return -1;
}

int isInOpenList(cell c)
{
 for (int i = 0; i < openList.size(); i++) //for each node in the open list
 {
  if (openList[i].c == c)
  {
   return i;
  }
 }
 return -1;
}

void printPath()
{
 if (pathCalculated)
 {
```

```
    cout << "Path: " << endl;
    for (auto c : cellPath)
    {
      cout << "(" << c.x << ", " << c.y << ")" << endl;
    }
    cout << endl;
  }
}

void printMapWithPath()
{
  cell c;
  cout << "Mapa con camino: " << endl;
  for (int i = 0; i < sizeX; i++)
  {
    for (int j = 0; j < sizeY; j++)
    {
      c.x = i;
      c.y = j;

      if (i == goalPos.x && j == goalPos.y)
      {
        cout << "Y ";
      }
      else if (i == startPos.x && j == startPos.y)
      {
        cout << "X ";
      }
      else if (isInPath(c))
      {
        cout << "- ";
      }
      else if (map[i][j] == 0)
      {
        cout << "0 ";
      }
      else
      {
        cout << "1 ";
      }
    }
    cout << endl;
  }
  cout << endl;
}

bool isInPath(cell c1)
{
  for (auto c : cellPath)
  {
    if (c == c1)
    {
      return true;
    }
  }
  return false;
}

void inflateObstacles()
{
  for (int i = 0; i < sizeX; i++)
  {
    for (int j = 0; j < sizeY; j++)
    {
      inflatedMap[i][j] = 0; //initialize the value

      //Inflate obstacles
      if (map[i][j] == 1) //si la celda actual es un obstáculo
      {
```

```
      int xmin = clamp(i - (double)OBS_INFLATING_RADIUS, 0, sizeX - 1);
      int xmax = clamp(i + (double)OBS_INFLATING_RADIUS, 0, sizeX - 1);
      int ymin = clamp(j - (double)OBS_INFLATING_RADIUS, 0, sizeY - 1);
      int ymax = clamp(j + (double)OBS_INFLATING_RADIUS, 0, sizeY - 1);

      for (int k = xmin; k < xmax; k++)
      {
       for (int l = ymin; l < ymax; l++)
       {
        distObs = distEuclidean(i, j, k, l);

        if (distObs < (double)OBS_INFLATING_RADIUS)
        {
         inflatedMap[k][l] = 1;
        }
       }
      }
     }
    }
  }
}

void printInflatedMapWithPath()
{
  cell c;
  cout << endl
       << "Inflated map: " << endl;
  for (int i = 0; i < sizeX; i++)
  {
    for (int j = 0; j < sizeY; j++)
    {
     c.x = i;
     c.y = j;

     if (i == goalPos.x && j == goalPos.y)
     {
      cout << "Y ";
     }
     else if (i == startPos.x && j == startPos.y)
     {
      cout << "X ";
     }
     else if (isInPath(c))
     {
      cout << "- ";
     }
     else if (inflatedMap[i][j] == 0)
     {
      cout << "0 ";
     }
     else
     {
      cout << "1 ";
     }
    }
   cout << endl;
  }
  cout << endl;
}
```

Anexo M. Código “move_base_config.launch”

```
<launch>

 <arg name="rosbot_pro" default="false" />
 <arg name="use_gazebo" default="false" />

 <!-- Gazebo -->
 <group if="$(arg use_gazebo)">
  <include file="$(find rosbot_gazebo)/launch/maze_world.launch" />
  <include file="$(find rosbot_description)/launch/rosbot_gazebo.launch"/>
  <param name="use_sim_time" value="true" />
 </group>

 <!-- ROSbot 2.0 -->
 <group unless="$(arg use_gazebo)">
  <include file="$(find rosbot_ekf)/launch/all.launch">
   <arg name="rosbot_pro" value="$(arg rosbot_pro)" />
  </include>

  <include if="$(arg rosbot_pro)" file="$(find rplidar_ros)/launch/rplidar_a3.launch" />
  <include unless="$(arg rosbot_pro)" file="$(find rplidar_ros)/launch/rplidar.launch" />
 </group>

 <node unless="$(arg use_gazebo)" pkg="tf" type="static_transform_publisher" name="laser_broadcaster" args="0 0 0 3.14 0 0 base_link laser 100" />

 <node pkg="gmapping" type="slam_gmapping" name="gmapping">
  <param name="base_frame" value="base_link"/>
  <param name="odom_frame" value="odom" />
  <param name="delta" value="0.1" />
 </node>

 <node pkg="move_base" type="move_base" name="move_base" output="screen">
  <param name="controller_frequency" value="10.0"/>
<param name="base_local_planner" value="apf_local_planner/APFLocalPlanner" />
  <rosparam file="$(find no_collision)/config/costmap_common_params.yaml" command="load" ns="global_costmap" />
  <rosparam file="$(find no_collision)/config/costmap_common_params.yaml" command="load" ns="local_costmap" />
  <rosparam file="$(find no_collision)/config/costmap_local_params.yaml" command="load" />
  <rosparam file="$(find no_collision)/config/costmap_global_params.yaml" command="load" />
  <rosparam file="$(find no_collision)/config/trajectory_planner_params.yaml" command="load" />
 </node>

</launch>
```

Anexo N. Código “costmap_common_params.yaml”

```
#distancia a la que una medida se considera obstáculo
obstacle_range: 6.0 #0.3
#distancia hasta la que se va descubriendo el mapa
raytrace_range: 8.5 #3.0
footprint: [[0.12, 0.14], [0.12, -0.14], [-0.12, -0.14], [-0.12, 0.14]]

#robot_radius: ir_of_robot
#margen de seguridad de los obstáculos
#inflation_radius: 0.16

map_topic: /map
subscribe_to_updates: true

global_frame: map
robot_base_frame: base_link
always_send_full_costmap: true
```

```
static_layer:
   map_topic: /map
   subscribe_to_updates: true
plugins:
   - {name: inflation_layer,       type: "costmap_2d::InflationLayer"}
   - {name: obstacle_layer,        type: "costmap_2d::ObstacleLayer"}
obstacle_layer:
   observation_sources: laser_scan_sensor
   laser_scan_sensor: {sensor_frame: laser, data_type: LaserScan, topic: scan, marking: true, clearing: true, min_obstacle_height: 0.0, max_obstacle_height: 5.0,
obstacle_range: 6.0, raytrace_range: 8.5}
```

Anexo O. Código “costmap_global_params.yaml”

```
global_costmap:
  update_frequency: 2.5
  publish_frequency: 2.5
  transform_tolerance: 0.5
  width: 15
  height: 15
  origin_x: -7.5
  origin_y: -7.5
  rolling_window: true
  inflation_radius: 2.5
  resolution: 0.1
  plugins:
   - {name: static_layer,       type: "costmap_2d::StaticLayer"}
```

Anexo P. Código “costmap_global_params.yaml”

```
local_costmap:
  update_frequency: 5
  publish_frequency: 5
  transform_tolerance: 0.25
  static_map: false
  rolling_window: true
  width: 3
  height: 3
  origin_x: -1.5
  origin_y: -1.5
  resolution: 0.1
  inflation_radius: 0.6
  plugins:
     - {name: obstacle_layer,       type: "costmap_2d::VoxelLayer"}
```

Anexo Q. Codigo “trayectory_planner_params.yaml”

```
TrajectoryPlannerROS:
  max_vel_x: 0.2
  min_vel_x: 0.1
  max_vel_theta: 0.35
  min_vel_theta: -0.35
  min_in_place_vel_theta: 0.25

  acc_lim_theta: 0.25
  acc_lim_x: 2.5
```

```
acc_lim_Y: 2.5

holonomic_robot: false

meter_scoring: true

xy_goal_tolerance: 0.15
yaw_goal_tolerance: 0.25
```

Anexo R. Enlaces de vídeos de las pruebas realizadas

Parte de evitación de colisiones.

Vídeo obstáculo estático

Vídeo obstáculo móvil

Vídeo obstáculo estático 2 (comportamiento de recuperación)

Parte de evitación de obstáculos.

Vídeo de prueba 1 con APF

Vídeo de prueba 1 con LZS

Vídeo de prueba 2 con APF

Vídeo de prueba 2 LZS

Vídeo de prueba 3 con APF

Vídeo de prueba 3 con LZS

Vídeo de prueba 4 con APF

Vídeo de prueba 4 con LZS

Vídeo de prueba 5 con APF

Vídeo de prueba 5 con LZS

Vídeo de prueba 6 con APF

Vídeo de prueba 6 con LZS

Vídeo de prueba 7 con APF

Vídeo de prueba 7 con LZS

Vídeo de prueba 8 con APF

Vídeo de prueba 8 con LZS

Vídeo de prueba 9 con APF

Vídeo de prueba 9 con LZS

Vídeo de prueba 10 con APF

Vídeo de prueba 10 con LZS

Vídeo de prueba 11 con APF (en 1 fase)

Vídeo de prueba 11 con LZS (en 1 fase)

Vídeo de prueba 11.1 con APF

Vídeo de prueba 11.1 con LZS

Vídeo de prueba 11.2 con APF

Vídeo de prueba 11.2 con LZS