Universidad Zaragoza
1542

Trabajo Fin de Grado

Evaluación de Estrategias de Exploración Robótica Autónoma

Evaluation of Autonomous Robotic Exploration Strategies

Autor

Ana Rubio Cotelo

Directores

Julio Alberto Placed Perales

José Ángel Castellanos Gómez

Escuela de Ingeniería y Arquitectura

2022

Resumen

El estudio de estrategias de exploración robótica autónoma es un tema actual y de gran interés en el ámbito de la robótica. Estos estudios están posibilitando la exploración de entornos desconocidos mediante robots que también desconocen su ubicación exacta. Esto es conocido como el problema de localización y mapeo simultáneos activos (SLAM Activo), cuyo objetivo principal es adquirir de la forma más eficaz y precisa posible un mapa de dicho entorno desconocido.

En este Trabajo Fin de Grado se ha estudiado el problema de exploración robótica autónoma en entornos desconocidos mediante la implementación y evaluación de un algoritmo de SLAM Activo en el entorno de programación de MATLAB. Para ello, se han abordado las tres etapas del SLAM Activo, haciendo hincapié en la comparación de los distintos criterios empleados en el proceso de toma de decisiones. Inicialmente se han identificado las posibles zonas a explorar por el robot. De esas zonas se ha seleccionado la de mayor interés para la exploración robótica aplicando una función de utilidad. Finalmente, se ejecuta la acción de seguimiento de trayectoria mediante un algoritmo de planificación y seguimiento de trayectoria hacia la frontera seleccionada. Para la evaluación de este algoritmo, ha sido necesaria la creación de tres entornos distintos en el simulador de Gazebo sobre los que se han analizado los resultados obtenidos tras la exploración de cada uno de ellos.

Se ha implementado un algoritmo capaz de ejecutar SLAM Activo, y se ha obtenido como conclusión que la función de utilidad basada en la mezcla de la entropía y el coste es la óptima para el proceso de exploración de un entorno desconocido en el menor tiempo posible.

Abstract

The study of autonomous robotic exploration strategies is currently a topic of great interest in the field of robotics. These studies are making exploration of unknown environments possible under uncertainty. This problem is known as Active Simultaneous Localization and Mapping (Active SLAM), which main objective is to acquire the most effective and precise map posible in an unknown environment.

In this bachelor thesis, autonomous robotic exploration of unknown environments has been studied through the implementation and evaluation of an Active SLAM algorithm in MATLAB. In order to achieve this, the three stages of Active SLAM have been addressed emphasizing the comparison of the different criteria used in the decision-making process. Initially, the possible areas to be explored by the robot have been identified. Then, the most useful for robotic exploration has been selected, according to some utility function. Finally, the optimal action has been executed, using path planning and tracking algorithms towards the selected area to explore. In order to evaluate the proposed algorithm, the creation of three different environments in Gazebo simulator was required, in which the exploration method has been analyzed.

To conclude, an Active SLAM complete algorithm has been implemented and later studied, finding that cost-utility functions based on the combination of entropy and distance are optimal for the process of exploring an unknown environment in the shortest possible time.

ÍNDICE

1. Introducción

En este capítulo se expone el contexto, el alcance, los objetivos, la metodología utilizada y la estructura de este Trabajo de Fin de Grado.

1.1. Motivación, alcance y objetivos

Cuando se utilizan robots autónomos existen situaciones en las que se parte de un entorno que ya es conocido por lo que el robot ya conoce el mapa de este, un ejemplo de este caso se puede ver en los robots aspiradoras que limpian siempre el mismo espacio. En otras ocasiones el robot debe moverse en un entorno desconocido, pero conoce su ubicación exacta, por ejemplo, mediante un sistema de posicionamiento global (GPS).

Además de estas dos situaciones, existe un caso que presenta mayor complejidad en el cual tanto el entorno como la ubicación del robot en él son desconocidas. A primera vista parece un dilema bastante complicado de solucionar, pues para la creación de un mapa se necesita ser capaces de situar al robot en él y a la vez para ser capaces de situar a un robot en un entorno es al menos necesario disponer de una representación de ese entorno (un mapa).

El objetivo de este Trabajo Fin de Grado consiste en el estudio y la comprensión de este problema tan importante en el ámbito de la robótica, ya que para solucionarlo se debe de realizar la compleja tarea de localizar constantemente al robot a la vez que se explora el entorno desconocido y se va generando un mapa de este, estas informaciones se deben de ir actualizando constantemente. La solución al problema de mapeo y localización simultánea, el SLAM, del inglés *Simultaneous Localization And Mapping*, o en español, Localización y Mapeo Simultáneos es en lo que se basa este proyecto que a la vez se refuerza con la aplicación del SLAM Activo cuya finalidad es hacer funcionar al algoritmo de SLAM de una forma más eficiente y así solucionar el problema de exploración robótica. Para ello, se dará gran importancia a la comparación de los distintos criterios empleados en el proceso de toma de decisiones que forma parte de la implementación del SLAM Activo con el fin de conseguir implementar un algoritmo a través del entorno de programación MATLAB que logre la consecución de un proceso de exploración robótica autónoma satisfactorio en un entorno desconocido.

El alcance de este proyecto se encuentra dentro de los objetivos mencionados anteriormente, quedando fuera de este tanto la implementación del algoritmo de SLAM, como la creación de la simulación del robot en Gazebo o la implementación de algoritmos de planificación de trayectorias.

El contexto de este trabajo se enmarca dentro del grupo de Robótica, Percepción y Tiempo Real del DIIS (Departamento de Informática e Ingeniería de Sistemas) y del I3A (Instituto Universitario de Investigación en Ingeniería de Aragón). La realización

de este trabajo se apoya en la utilización del *Robotics System Toolbox* de MATLAB, para la implementación del algoritmo de SLAM Activo basado en grafos en tres entornos distintos para poder abordar los problemas que se han planteado.

1.2. Estructura

La memoria de este trabajo se ha estructurado en 6 capítulos:

- En el capítulo 1 se contextualiza el proyecto a realizar, se exponen los objetivos, el alcance y la estructura de la memoria.

- En el capítulo 2 se definen los conceptos de SLAM y SLAM Activo, haciendo hincapié en los pasos en los que se divide el SLAM Activo.

- En el capítulo 3 se tratan las herramientas empleadas. Primero, se introduce el entorno de programación del Sistema Operativo Robótico (ROS), a continuación se describe el simulador utilizado, Gazebo, y finalmente se muestra la *Toolbox* de MATLAB utilizada, *Robotics System Toolbox*.

- En el capítulo 4 se desarrolla la implementación de los algoritmos con la finalidad de ejecutar SLAM Activo en este proyecto. Explicando en detalle cada uno de los pasos del SLAM Activo y el porqué de las decisiones tomadas.

- En el capítulo 5 se muestran los resultados obtenidos tras la ejecución del algoritmo de SLAM Activo.

- Para finalizar, en el capítulo 6, se exponen las conclusiones obtenidas tras la realización de este Trabajo Fin de Grado.

Adicionalmente, se incluyen en el trabajo los siguientes anexos:

- En el Anexo I se muestra el algoritmo creado cuya función se basa en la detección de las fronteras a partir de un mapa de ocupación.

- En el Anexo II se muestra el algoritmo creado para el cálculo de la trayectoria *(path)* mediante la utilización de los planificadores de trayectorias.

- En el Anexo III se muestra el algoritmo creado para el cálculo de la frontera de mayor interés a estudiar.

- En el Anexo IV se muestra el código del algoritmo de exploración robótica.

2. Antecedentes

En este capítulo se explicarán los conceptos básicos que forman parte de la exploración robótica autónoma: el SLAM y el SLAM Activo.

La finalidad del SLAM y la base sobre la que se apoya este trabajo radica en el análisis de la siguiente cuestión: si se coloca a un robot en una localización y en un ambiente desconocidos, ¿podría crear un mapa de dicho entorno y a la vez localizarse en él?

2.1. SLAM

La localización y el mapeo simultáneo o SLAM es el proceso por el cual un robot construye un mapa de su entorno a la vez que se ubica a él mismo en dicho entorno. Esta tarea es un problema bastante importante en el ámbito de la robótica debido a que tanto la ubicación exacta del robot como el mapa de entorno son variables dependientes, haciendo que sea complejo estimarlas por separado. A pesar de esta dificultad, en la actualidad se están realizando nuevos estudios sobre este tema con los que esta estimación es posible de realizar sin conocer tanto la localización como el entorno. Aunque es un tema que sigue necesitando estudio, pues los mapas creados mediante el mapeo y la localización simultánea no son tan exactos como los mapas creados a partir de otros datos iniciales.

La resolución del problema de SLAM consiste en estimar la trayectoria del robot y el mapa de entorno a medida que el robot se mueve en él. El problema de SLAM se describe mediante herramientas probabilísticas debido a que las mediciones de entorno no son exactas, esto se refleja de forma visual en la Figura 2.1 ya que pueden presentar, por ejemplo, ruido en las observaciones o mala asociación de datos.

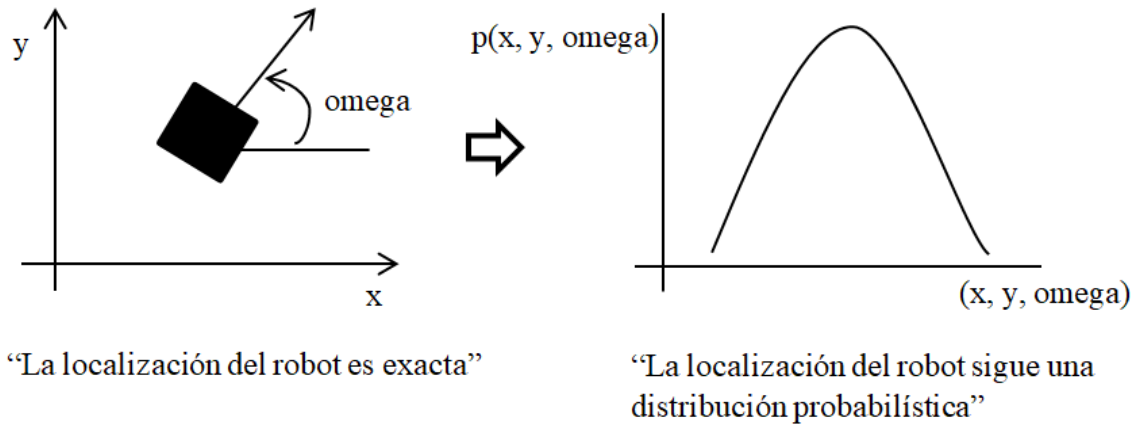

Figura 2.1 *Explicación visual del enfoque probabilístico*

En este trabajo, se abordará el problema de SLAM a través del denominado SLAM basado en grafos (*graph* SLAM), el cual requiere la realización de dos tareas, la primera consiste en la asociación de los datos en bruto obtenidos de los sensores, y la posterior creación de un grafo. La segunda tarea consiste en la optimización del grafo, esto es, la determinación de la configuración de los nodos más probable a partir de los arcos que forman el grafo.

Para la realización de la primera tarea del SLAM basado en grafos es necesaria la construcción de un grafo compuesto por nodos y arcos. En él, cada nodo, representa la pose (posición y orientación) del robot o la posición de un punto de referencia del mapa. Mientras que un arco entre dos nodos codifica una medición de un sensor que crea una restricción entre las poses conectadas. Estas restricciones pueden ser tanto de odometría (poses consecutivas del robot), de cerrado de bucle o de observación. Un arco de cerrado de bucle se produce cuando se crea una restricción entre dos nodos no consecutivos y representa la reobservación de una zona conocida [1].

A continuación, en la Figura 2.2, se muestra el proceso de construcción del grafo mediante SLAM basado en grafos. Una forma de representar el grafo es mediante esta matriz y a partir de ella se puede construir el grafo.

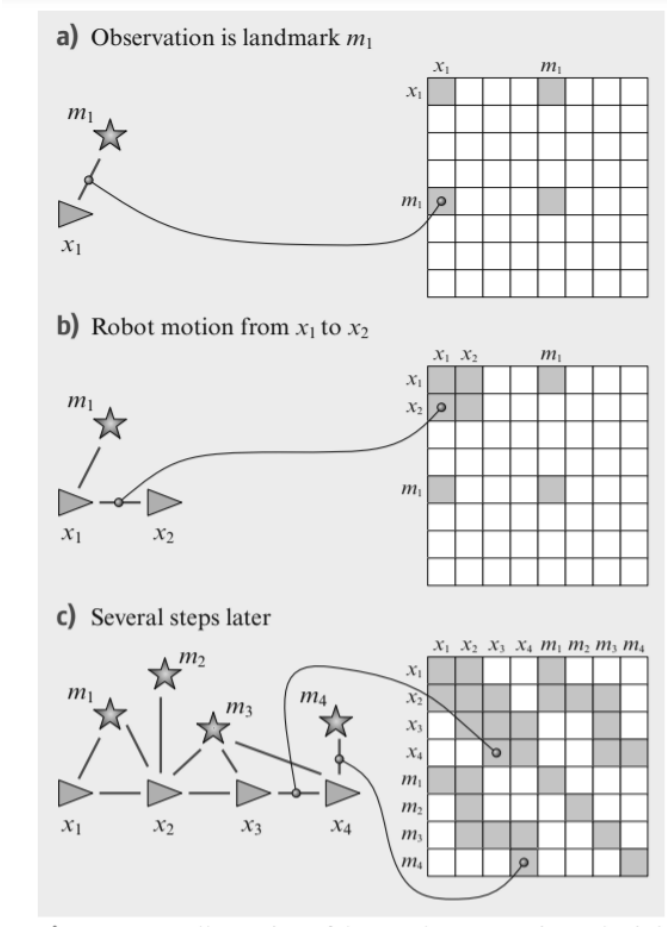

Figura 2.2 *Figura que muestra la construcción del grafo. El diagrama de la izquierda muestra el grafo y el de la derecha muestra las restricciones en forma de matriz donde landmark son los puntos de referencia [2]*

En la diagonal de la matriz se muestran las posiciones del robot y los puntos de referencia, la suma de la cantidad de estos valores indicará la dimensión de la matriz. Siendo, por ejemplo, en el caso c) la dimensión de esta matriz de 8x8 al haber cuatro posiciones del robot y cuatro puntos de referencia. Las variables *x* representadas como triángulos se corresponden con las poses del robot y las variables *m* representadas como estrellas corresponden con puntos de referencia

Los elementos que se encuentran fuera de la diagonal representan los arcos. En el caso a) se ve de forma muy clara esta relación, ya que entre la pose x_1 y el punto de referencia m_1 existe una restricción, esta se representará en la matriz ocupando la celdas (m_1, x_1) y (x_1, m_1).

A continuación, en la Figura 2.3, se muestra un grafo completo. En él se pueden ver los nodos que representan las poses de robot (triángulos) y los puntos de referencia del mapa (estrellas). Los arcos se muestran representados por líneas rectas continuas y punteadas. Los arcos que unen posiciones del robot representan restricciones relativas de movimiento (odometría), mientras que los arcos que unen posiciones del robot con puntos de referencia representan restricciones relativas de observación. En esta figura, se puede apreciar que, a medida que se van estimando las poses del robot con los puntos de referencia dados, se van produciendo discordancias entre la posición verdadera y la posición estimada.

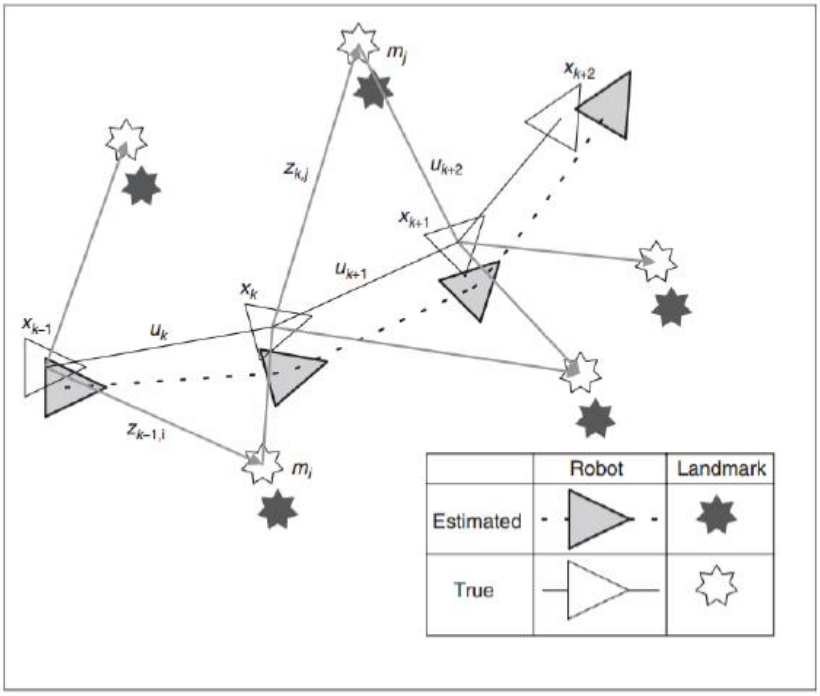

Figura 2.3 *Grafo del problema de SLAM, donde los triángulos representan las poses del robot, las estrellas los puntos de referencia y las líneas rectas o punteadas las restricciones.* [3]

Estas discordancias influyen en cómo de preciso es el modelo de entorno. Se necesitan aplicar ciertas restricciones para que el algoritmo SLAM basado en grafos sea satisfactorio. Para ello, se generan restricciones relativas que relacionan cada posición del robot con la posición anterior y con las restricciones espaciales entre las poses que resultan de la observación z_t o de las medidas de odometría u_t que están codificadas en los arcos de los nodos. Estas restricciones se generan cada vez que el robot ve un punto de referencia y el SLAM basado en grafos recoge cada una de estas restricciones.

Una vez construido el grafo, se busca encontrar la configuración de las poses del robot que mejor satisfaga las restricciones. La resolución al problema de SLAM basado en grafos desde un enfoque probabilístico consiste en estimar la siguiente trayectoria del robot y el mapa de entorno *m* con las medidas y la posición inicial como se muestra en la siguiente ecuación [1]:

$$p(x_{1:T}, m | z_{1:T}, u_{1:T}, x_0) \quad (1)$$

Donde $x_{1:T}$ es la trayectoria del robot descrita como una secuencia aleatoria de variables, $u_{1:T}$ un conjunto de medidas de odometría y $z_{1:T}$ las percepciones del entorno. La posición inicial x_0 define la posición del mapa y puede ser elegida de forma arbitraria, es la utilizada para posicionar el centro de referencia y a partir de ella comenzar a realizar las estimaciones.

El mapa *m* puede ser parametrizado como un conjunto de puntos de referencia ubicados especialmente mediante representaciones densas, como cuadrículas de ocupación, mapas de superficie o mediante mediciones de sensores sin procesar. Esto depende de los sensores utilizados, de las características del entorno, de los algoritmos de estimación, etc. El mapa, independientemente del tipo de representación, se define por las mediciones y las ubicaciones donde se han adquirido estas mediciones.

Este Trabajo Fin de Grado se va a centrar en el grafo de poses o *pose-graph*, el cual solo contiene como nodos las poses del robot, no tiene en cuenta los puntos de referencia para su realización ya que al utilizar un mapa de ocupación, los puntos de referencia ya se encuentran contenidos en él.

La optimización del grafo consiste en resolver un problema de minimización de errores cuya ecuación es:

$$x^* = \operatorname{argmin} F(x) \quad (2)$$

Donde F(x) es la función de coste que representa los errores en la estimación, esta función depende de las observaciones.

$$F(x) = \sum_{(i,j)\in C} e_{ij}^T \Omega_{ij} e_{ij} \tag{3}$$

Donde la variable e_{ij} es el error de la estimación y la variable Ω_{ij} representa la calidad de la observación. Cuanto mayor error haya o mayor sea el error en la observación menos fiable será el arco.

El problema de minimización de errores trata de encontrar una configuración de nodos que minimice el error introducido por el ruido en las restricciones con la finalidad de la obtención de un mapa lo más realista posible. En la Figura 2.4 se muestra un esquema que refleja el error debido a las discordancias entre la pose real y la estimada.

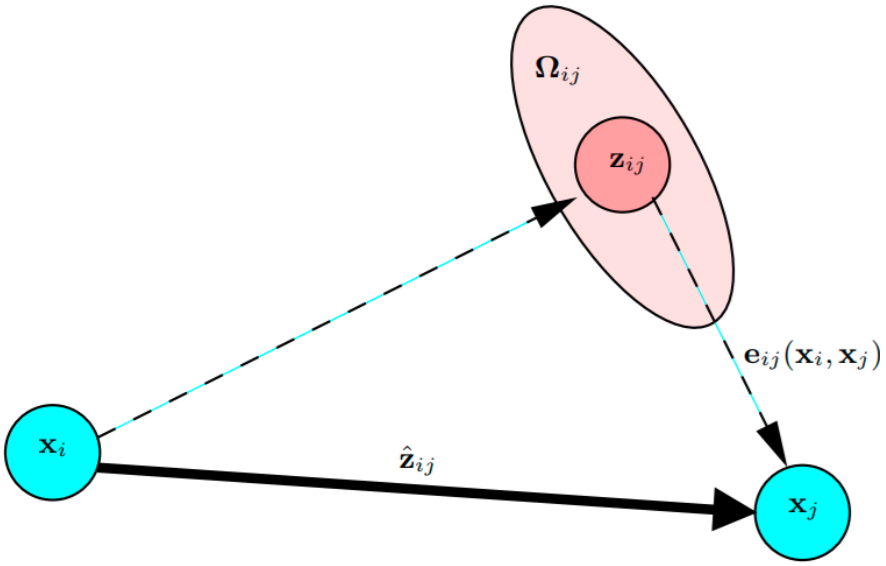

Figura 2.4 *Grafo que muestra la función error* [1]

Debido a esta discordancia, se ha de tener en cuenta la variable error, estudiando el desplazamiento entre la medida real (z_{ij}) y la estimada $(\hat{z}_{ij})$, esta diferencia, se calcula mediante la función error:

$$e_{ij}(x_i, x_j) = z_{ij} - \hat{z}_{ij}(x_i, x_j) \tag{4}$$

2.2.SLAM Activo

El SLAM Activo se define como el paradigma de controlar un robot que está realizando SLAM con la finalidad de reducir la incertidumbre de su localización y de la representación del mapa, dando lugar a representaciones más precisas del entorno. Puede ser definido como un problema de control de movimiento de un robot realizando SLAM de tal forma que maximice la exactitud de la representación del mapa y de su localización [4] [5]. En la Figura 2.5 se muestra una gráfica que representa los tres problemas fundamentales de la robótica móvil: localización, control de movimiento y mapeo.

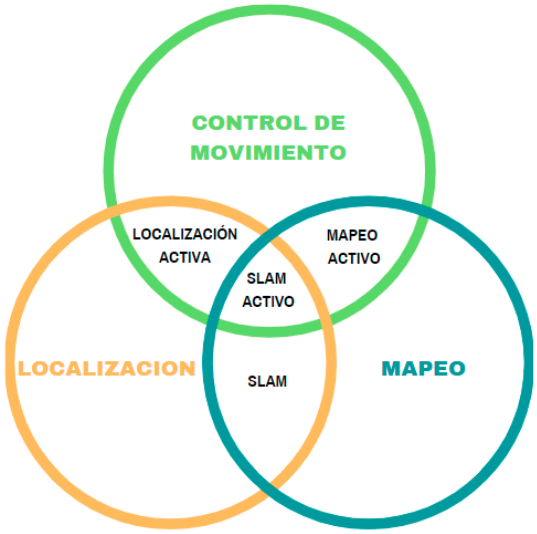

Figura 2.5 *Diagrama de los tres problemas fundamentales en robots móviles*

El SLAM Activo se puede dividir en tres pasos: en primer lugar, el robot identifica posibles localizaciones para explorar, a continuación se realiza el cálculo de la utilidad de las opciones disponibles y selección de la que sea de mayor utilidad para el robot, para finalmente guiar el robot hacia la opción de mayor interés escogida.

El primer paso del SLAM Activo comienza con la identificación y la elección de distintos puntos de interés que ayuden al robot a solucionar el dilema de exploración-explotación, explorar nuevas localizaciones (exploración) a la vez que maximiza la información esperada que obtiene del entorno (explotación).

Las localizaciones de los puntos de interés en el mapa son seleccionadas en base a la información de sus alrededores, usando técnicas como exploraciones basadas en fronteras donde los robots buscan áreas conocidas en el mapa que sean adyacentes a un espacio sin explorar.

Una vez se ha identificado y seleccionado los puntos de interés en la estimación actual del mapa, el siguiente paso es la evaluación de su utilidad.

El robot calcula la utilidad de elegir cualquiera de las posiciones disponibles seleccionando la de mayor utilidad. El proceso de evaluar la recompensa de una acción particular a un punto de interés se realiza a través de una función de utilidad (U) que se encarga de evaluar el efecto esperado de las futuras acciones del robot. Se basa en cuantificar el error en la estimación de la posición del robot y el error en la estimación de la posición del mapa. La razón fundamental de una función de utilidad es ayudar a lograr los objetivos de SLAM Activo, es decir, resolver el dilema exploración-explotación.

La función de utilidad puede estar basada en dos teorías, la teoría de la información que da lugar a métricas como la entropía o la Teoría del Diseño Óptimo de Experimentos (TOED). Una función de utilidad basada en la teoría de la información es la basada en la entropía, en ella, se considerará un valor elevado de entropía cuanto mayor sea el número de celdas desconocidas que se encuentran en el mapa de ocupación del entorno a estudiar. La calidad del mapa aumentará a medida que se conozca más información sobre él, interesando elegir la frontera que más reduzca la entropía. También puede estar basada en la distancia a cada una de las fronteras, en esta función se evalúa el coste de ir hacia cada una de las fronteras, siendo así elegida la frontera que más cerca se encuentre de la posición actual del robot.

La mezcla de la función basada en la entropía y en el coste permite al robot considerar el costo relacionado con la exploración como el tiempo asignado a la misma. La complejidad de esta función se basa en la consecución de una relación entre ellas que se encargue de hacer que su escala sea comparable [5] [6].

El último paso trata de la selección y ejecución de la acción o conjunto de acciones óptimo. El proceso de seleccionar la mejor opción es un proceso de optimización en el cual el robot necesita maximizar la función de utilidad dada sujeta a las acciones disponibles. La finalidad de este paso es el guiado del robot hacia la frontera de mayor interés escogida mediante un seguimiento de trayectorias planificadas. Para la planificación de la trayectoria, se utilizan algoritmos de planificación como el PRM (*Probabilistic Roadmap*) o el RRT (R*apidly Exploring Random Trees*) y el seguimiento de esta misma se realiza mediante controladores.

Uno de los enfoques más utilizados para finalizar la ejecución del algoritmo (conocido como criterio de parada o *stopping criteria*) es el de establecer un valor máximo de tiempo al algoritmo. Otro de los enfoques utilizados habitualmente se basa en verificar si se ha producido algún incremento en la información del entorno. En caso de ser estos incrementos constantes durante un cierto periodo de tiempo se daría por terminado el algoritmo de SLAM [5].

3. Herramientas

En este capítulo se tratan las herramientas utilizadas a lo largo de este proyecto. Se exponen las herramientas a través de las cuales se elaboran los algoritmos necesarios para llevar a cabo este proyecto: ROS (*Robot Operating System)* [7] y *Robotics System Toolbox* de MATLAB [8] y la plataforma de simulación utilizada, Gazebo.

3.1. Robot Operating System (ROS)

ROS es un sistema operativo de programación robótica, se trata de un *framework* para el desarrollo de software para robots, programado en C++ y Phyton. Contiene una colección de herramientas y bibliotecas que tienen como objetivo simplificar la tarea de crear un comportamiento robótico complejo y robusto. El acceso al sistema ROS ha sido posible a través de la máquina virtual VMware, basada en Ubuntu Linux, la cual está preconfigurada para admitir los ejemplos en ROS, esta máquina virtual contiene ROS (ROS Melodic) y Gazebo (versión 9.0.0).

ROS tiene dos partes básicas: la parte del sistema operativo y *ros-pkg*, paquetes aportados por la comunidad de usuarios que implementan funcionalidades tales como SLAM, planificación, percepción, simulación, etc. Los procesos de ROS se representan como nodos en una estructura gráfica donde el procesamiento se realiza en los nodos, que pueden recibir, mandar y multiplexar mensajes de sensores, control, estados, planificadores y actuadores [7] [9].

A continuación se describirán los conceptos básicos de ROS que son más relevantes para este proyecto [10]:

- *Master*: Es el gestor de ROS y permite que los nodos se localicen entre ellos para que puedan comunicarse mediante *topics* y servicios. Cada nodo se dirige primero al *Master* para crear y buscar el *topic* al que quieren publicar o subscribirse.

- Nodos: Son procesos que realizan cálculos, un sistema de control de robot generalmente comprende muchos nodos. Por ejemplo, un nodo controla los motores de las ruedas, un nodo realiza la localización, un nodo realiza la planificación de la ruta, etc.

- *Parametrer Server*: Forma parte del *Master* y permite que los datos se almacenen por clave en una ubicación central

- *Topics:* Es el *bus* mediante el que un nodo envía un mensaje publicándolo en un *topic* determinado, un nodo se suscribirá al *topic* correspondiente al tipo de datos que esté interesado en conocer y si quiere generar datos entonces el nodo deberá publicar en el *topic* respectivo. Cada *topic* transporta solamente un único tipo de mensaje. En la Figura 3.1 se puede ver un modelo de trasmisión de información.

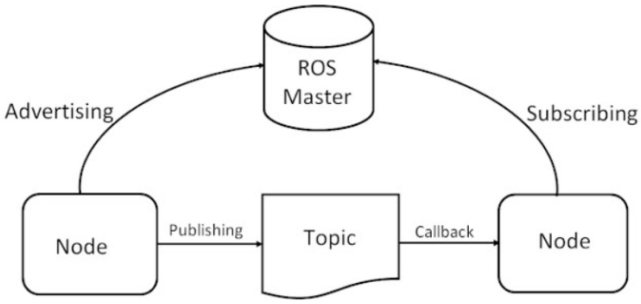

Figura 3.1 *Modelo de comunicación entre nodos* [11]

- Servicios: Se encargan del intercambio de mensajes síncrono entre nodos de tipo cliente/ servidor. Trabajan con parejas de nodos, unos que envían un mensaje de pregunta y otros que recibe la respuesta cuando esta tiene lugar.

3.2. Gazebo

En este apartado se mostrará la creación de varios entornos de simulación y más adelante la implementación de los algoritmos para conseguir que el robot se sincronice con ROS. Para la realización de este proyecto, se han creado 3 entornos, que se mostrarán en el subapartado 3.2.1.

3.2.1. Creación de entornos de simulación en Gazebo

Gazebo es un simulador de entornos 3D que permite evaluar el comportamiento de un robot en un mundo virtual. Este simulador permite diseñar robots de forma personalizada, crear mundos virtuales o importar modelos ya creados [12].

Para este trabajo se ha utilizado un modelo de robot ya existente: turtlebot3 Burger [13], el cual se muestra en las Figuras 3.2 y 3.3.

Figura 3.2 *Modelo de robot turtlebot3 Burger* [13]

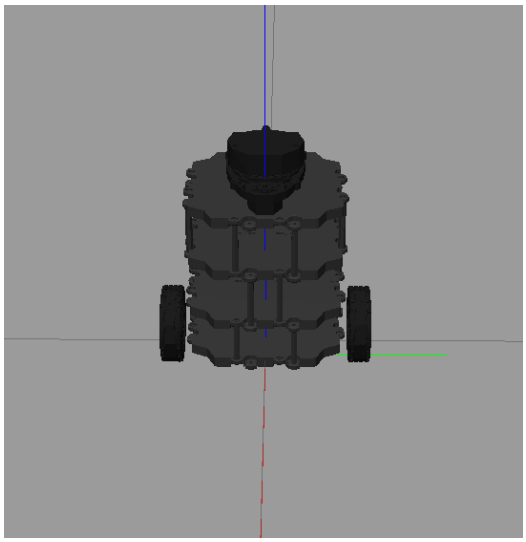

Figura 3.3 *Modelo turtlebot3 visto en Gazebo*

El robot *turtlebot3* modelo Burger utilizado en este trabajo presenta las siguientes características: una velocidad linear máxima de valor igual a 0.16 m/s, una velocidad angular máxima igual a 0.55 rad/s, un radio que engloba las ruedas del robot de 0.165 m. El sensor láser incorporado en este robot es el LDS-01, capaz de recopilar información alrededor del robot gracias a que es de 360º con una distancia de detección de valores entre 0.120 m y 3.500 m.

Para la creación y edición de los entornos de simulación se ha utilizado Gazebo a través de la máquina virtual VMware. Los entornos creados son de dimensiones que varían entre 30 y 40 m^2, ya que debido a las reducidas velocidades máximas que presenta el robot *turtlebot3*, se han creado entornos para los que se puedan obtener resultados de exploración para un tiempo de exploración no muy elevado. El primer entorno realizado y sobre el que en el apartado 3.3 se realizarán las correspondientes pruebas se muestra en la Figura 3.4.

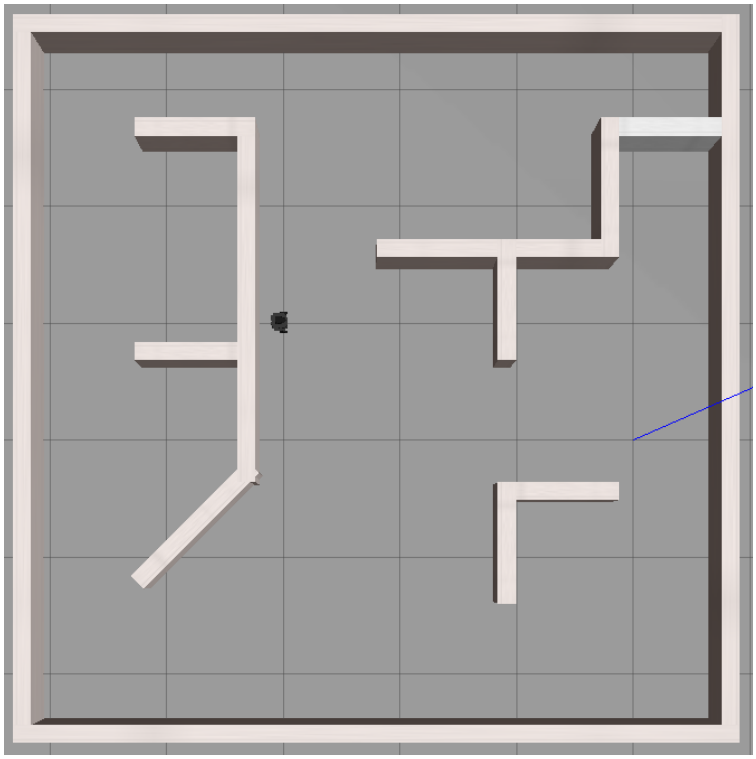

Figura 3.4 *Interfaz de Gazebo mostrando el Entorno 1*

Los escenarios 2 y 3 se muestran en las Figuras 3.5 y 3.6.

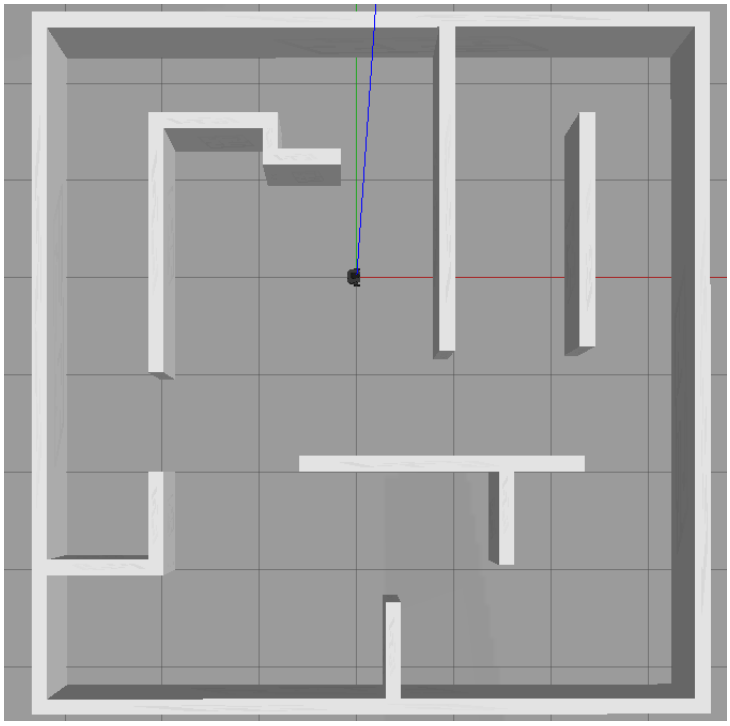

Figura 3.5 *Interfaz de Gazebo mostrando el Entorno2*

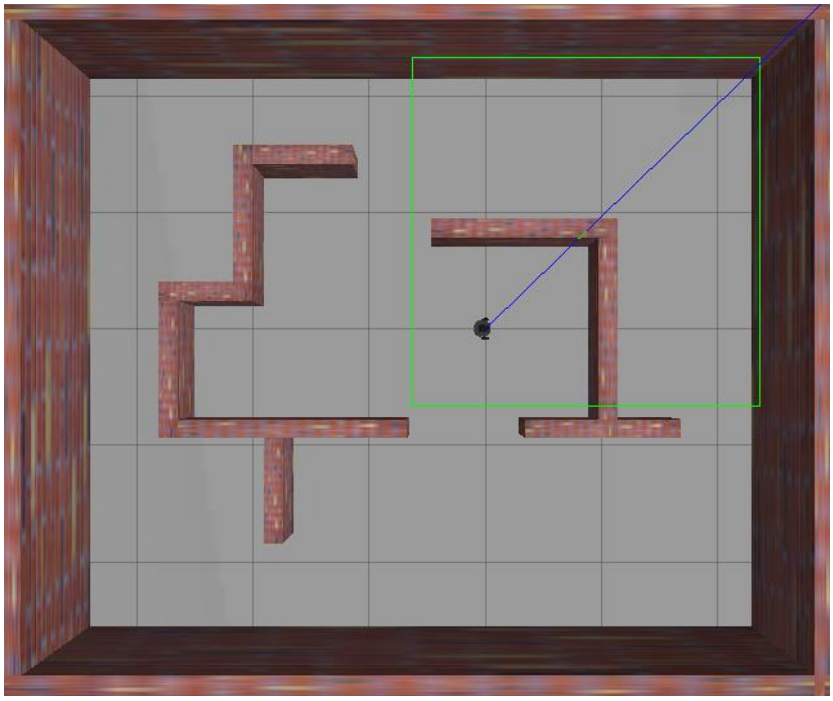

Figura 3.6 *Interfaz de Gazebo mostrando el Entorno 3*

3.3.Robotics System Toolbox de MATLAB

MATLAB es la herramienta mediante la cual se simula el control del robot. *Robotics System Toolbox* de MATLAB proporciona una interfaz que conecta MATLAB y Simulink con ROS, lo que permite compilar y ejecutar el código de robots que operen bajo ROS y conectar MATLAB a Gazebo. En la Figura 3.7 se muestra un esquema de las conexiones comentadas anteriormente.

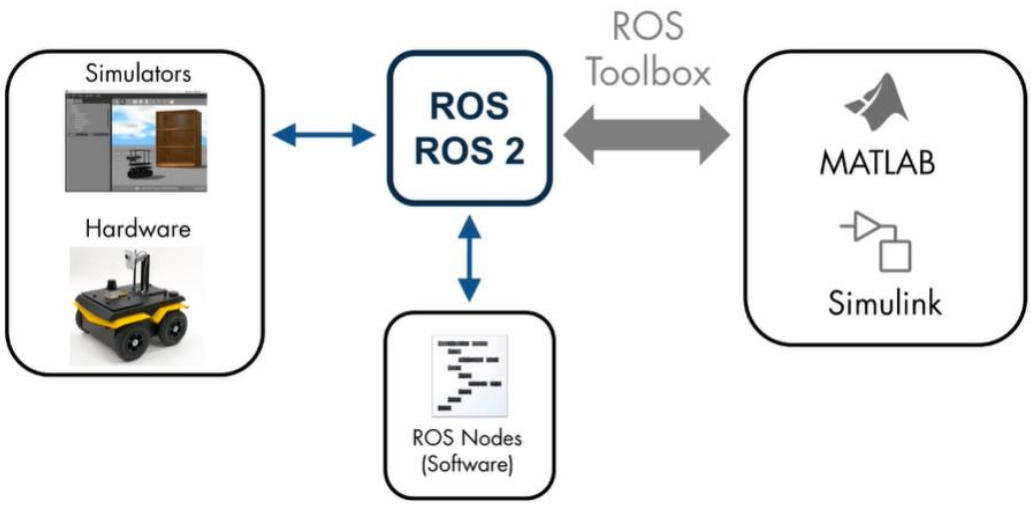

Figura 3.7 *Esquema de las conexiones entre ROS, simuladores y hardware y MATLAB y Simulink* [14]

3.3.1. Conectividad MALTAB-ROS-Gazebo

En este subapartado se explicará cómo realizar la conectividad entre ROS y MATLAB. Se trabajará con dos máquinas, en una se ejecutará MATLAB y en la otra se ejecutará ROS. Debe de existir una conectividad completa entre cada una de las máquinas a través de la red para que los *topics* de cada una de ellas sean accesibles a las demás máquinas. Para conseguir esta conectividad a través de la red se deben de configurar dos variables de entorno: *ROS_MASTER_URI*, que da la información del nodo principal y *ROS_IP*, que especifica la dirección del nodo global. En la Figura 3.8 se muestra de manera más visual la conectividad entre MATLAB y ROS.

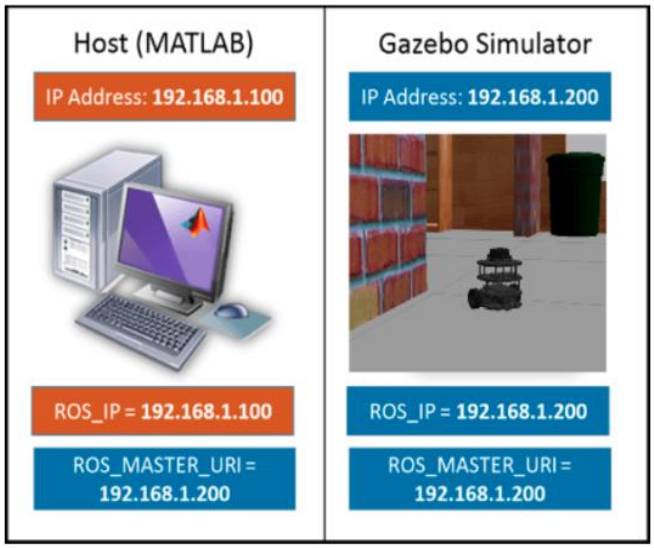

Figura 3.8 *Conexión MATLAB-ROS* [15]

Tras este paso, se procede a realizar la conexión en MATLAB, a través de la creación de un archivo que realice la conexión con Gazebo para inicializar ROS y conectarlo al robot:

```
ipaddress = "http://192.168.178.132:11311";
rosinit(ipaddress)
```

Para ver la disponibilidad de los *topics* de ROS se ejecuta el siguiente código:

```
rostopic list
```

En caso de que al ejecutar este comando, no se pudiesen ver todos los *topics* sería una señal de que la red no ha sido configurada correctamente. Para la realización de este proyecto interesará analizar 3 *topics* en concreto: *scan*, *cmd_vel y odom.*

- ***Scan*:** Mediante este *topic* se obtiene acceso a los sensores del láser.
  Tipo: sensor_msg/ LaserScan
  Publishers: * /gazebo
  Subscribers: * /matlab_global_node
  Con esta información se indica que el escáner láser es publicado por Gazebo y que se subscribe desde MATLAB.

- ***Cdm_vel**:* Mediante este *topic* se controla la velocidad del robot
  Tipo: geometry_msg/Twist
  *Publishers** /matlab_global_node
  Subscribers: * /gazebo
  Gazebo está subscrito al *topic /cdm_vel* y se deberá de publicar en MATLAB.

- ***Odom*:** Mediante este *topic* se conoce la localización del robot
  Tipo: nav_msgs/Odometry
  Publishers:* /gazebo
  Subscribers: * /matlab_global_node
  El *topic* de odometría es publicado por Gazebo y se subscribe desde MATLAB.

Además de conocer los 3 *topics* mencionados anteriormente, también es necesario conocer las funciones más utilizadas de la *Robotics System Toolbox* como:

- *Rosinit*: Es la instrucción encargada de iniciar el sistema ROS.

- *Rosshutdown:* Se encarga de cerrar el sistema ROS.

- *Rosmsg:* Obtiene información sobre los mensajes y los tipos de mensajes de ROS.

- *Rostopic*: Obtiene información sobre los *topics* en la red de ROS. Muestra todos los *topics* bajo el *Master*.

- *Rossubscriber:* Se encarga de crear un ROS *subscriber* para mensajes recibidos en la red de ROS.

- *Rospublisher:* Se encarga de publicar un *topic.*

3.3.2. Algoritmos e implementación de graph-SLAM

En este subapartado se muestran los algoritmos de MATLAB en los que se utilizarán los *topics* y las funciones mencionadas anteriormente.

El primer paso consiste en realizar la conexión con Gazebo. La conexión MATLAB-ROS se ha explicado en el apartado 3.3.1. El siguiente paso consiste en conectarse con Gazebo a los *topics* del láser, la odometría y la velocidad para estudiar el

funcionamiento del escáner láser a la vez que la función de mover el robot y la de conocer su posición. Para ello, se crea un algoritmo cuya función sea hacer que el robot gire sobre su eje a la vez que muestra su entorno.

Con ese fin, se publica en MATLAB el *topic cmd_vel* junto al mensaje que contiene la información de esta publicación. Los comandos de velocidad enviados al robot serán lineales y angulares, pues el robot consta de dos ruedas las cuales son las encargadas de realizar su función móvil. La velocidad lineal ejecuta la función de avance por lo que estos valores serán dados en el eje X y la velocidad angular, es la encargada de realizar el giro del robot y posicionar a este en la dirección de avance deseada, por lo que estos valores serán dados en el eje Z.

Como en el algoritmo creado solo interesa girar el robot sobre su eje, solamente se envía al *topic*, mediante la orden *send*, el mensaje de que el robot gire con una velocidad angular de 1 radián por segundo.

```
robotCmd = rospublisher("/cmd_vel") ;
velMsg = rosmessage(robotCmd);
velMsg.Angular.Z=1;
send(robotCmd,velMsg)
```

Una vez realizada esta acción se procede a mostrar los datos obtenidos por el robot durante el escaneo a través del sensor láser.

```
lidarSub = rossubscriber("/scan");
scanMsg = receive(lidarSub);
tic
while toc < 2*pi
  scanMsg = receive(lidarSub);
end
velMsg.Angular.Z = 0;
send(robotCmd,velMsg)
```

El resultado obtenido se muestra en las Figuras 3.9 y 3.10. En la Figura 3.10 se muestra la información recogida por el robot del entorno tras realizar un giro de 360º en el punto (0,0), pudiendo apreciarse el alcance del sensor láser del robot.

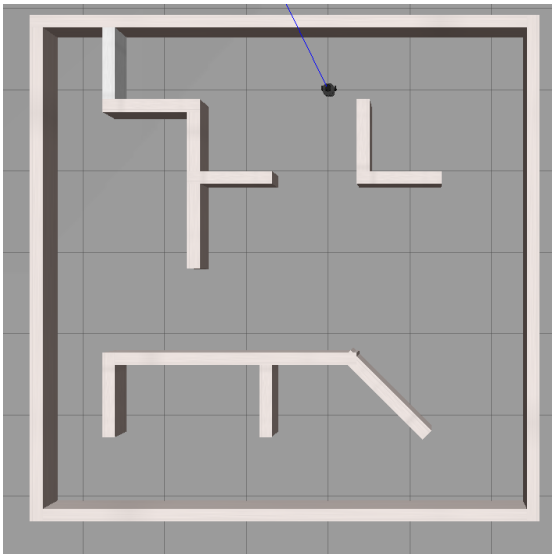

Figura 3.9 *Robot en el Entorno 1 mostrado en el simulador de Gazebo*

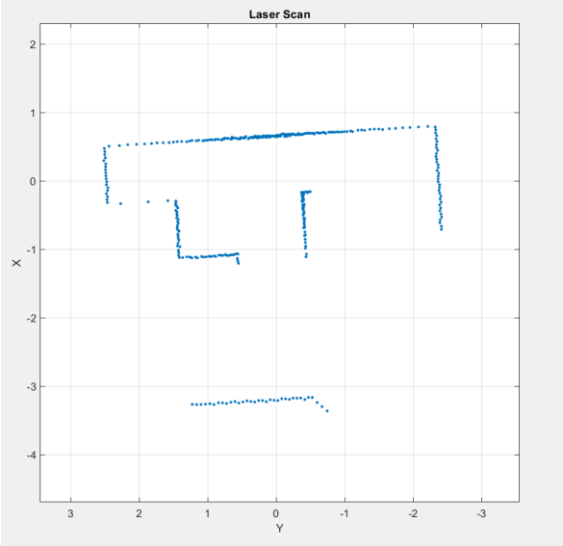

Figura 3.10 *Información del Entorno 1 obtenida a través del sensor láser del robot*

Una vez creados los entornos de simulación, se procede a utilizar un algoritmo de *graph* SLAM de la *Robotics System Toolbox* de MALTAB con la finalidad de conseguir conectar el robot al algoritmo SLAM y que dicho robot sea capaz de recorrer el entorno seleccionado evitando obstáculos a la vez que va creando un mapa del entorno.

El primer paso consiste en suscribir y publicar los *topics* necesarios, se publica el *topic* correspondiente a la velocidad y se subscriben los *topics* correspondientes al escáner láser y a la odometría. Con el objetivo de realizar la evasión de obstáculos comentada, se ha empleado el algoritmo VFH (*Vector Field Histograms*) el cual permite al robot evitar obstáculos durante su recorrido a lo largo del entorno, para ello, se especifican las propiedades del algoritmo de acuerdo a las dimensiones del robot y se establece la dirección objetivo *"targetDir"* a 0 para que el robot avance solamente con velocidad lineal, es decir en línea recta. Y se configura *rateControl*, mediante esta variable se puede controlar la velocidad a la que opera el bucle [16].

El siguiente paso a realizar es crear el objeto *LidarSLAM*. La finalidad de la clase *LidarSLAM* es la realización de la localización y el mapeo simultáneos para las entradas del sensor de escaneo láser. La propiedad p*ose-graph* del objeto *LidarSLAM* guarda la cantidad de nodos, arcos y arcos de cerrado de bucle, así como la ID de cada uno de estos obtenidos del escáner láser. También busca cierres de bucle, donde los escaneos se superponen a regiones previamente mapeadas, y optimiza las poses de los nodos en el grafo de poses, p*ose-graph* [17]. En la siguiente línea de código se muestra la creación del objeto *LidarSLAM*.

```
slamAlg=lidarSLAM(mapResolution,maxLidarRange);
```

Para aplicar esta función, primero se especifica el rango máximo del láser, el cual debe ser menor que el rango máximo, ya que cuanto más cerca del rango máximo se encuentre este valor, menos precisas serán las lecturas del láser. Es por eso que el valor escogido es 8. La resolución del mapa se establece a 20 celdas por metro, lo que ofrece una precisión de 5 cm. El umbral de cierre de lazo, *slamAlg.LoopClosureThreshold*, interesa que sea un número elevado, ya que a mayor valor, mejor es el rechazo de falsos positivos a la hora de identificar lazos cerrados, el valor dado a esta variable por lo tanto es 200. Un radio de búsqueda de lazos cerrados que presente un valor elevado, dará lugar a un rango de estudio de cierres de bucle más amplio, por lo tanto un valor elevado de la variable *slamAlg.LoopClosureSearchRadius*, incrementará el tiempo de búsqueda. Es por ello por lo que el valor seleccionado para esta variable es de 3 metros, pues se considera un rango de búsqueda suficiente para la localización de arcos de cerrado de bucle.

En el siguiente código se muestra el bucle realizado durante 600 segundos para navegar con el robot sobre el entorno virtual. La posición del robot se actualizará en el bucle a partir de los puntos de la trayectoria y los escaneos que se obtienen del robot mientras este navega por el entorno. Los cierres de bucle se detectan automáticamente mientras el robot se mueve y la optimización del *pose-graph* se realiza siempre que se detecta un cierre de bucle. Y mediante la utilización del controlador VFH se controlarán las velocidades tanto angular como lineal enviadas al robot, de tal forma que cuando el escáner láser del robot detecte un obstáculo, este controlador le mandará al robot la información de que no se mueva con velocidad lineal y solamente gire hasta encontrar una dirección en la que no se detecte ningún obstáculo.

```
tic
while toc<600
  scanMsg = receive(lidarSub);
  range = double(lidarSub.LatestMessage.Ranges);
  angles = double(scanMsg.readScanAngles);
  scan = lidarScan(range,angles);
  steerDir = vfh(scan, targetDir);
  if ~isnan(steerDir)
    desiredV = 0.2;
    w=exampleHelperComputeAngularVelocity(steerDir,1);
  else
    desiredV = 0.0;
    w = 0.2;
  end
```

```
  velMsg.Linear.X = desiredV;
  velMsg.Angular.Z = w;
  velPub.send(velMsg);
  range(range==Inf)= maxLidarRange+2
  scans = lidarScan(range,angles);
  [isScanAccepted,loopClosureInfo,optimizationInfo]
  = addScan(slamAlg,scans);
  waitfor(controlRate);
end
velMsg.Linear.X=0;
velMsg.Angular.Z = 0;
send(robotCmd,velMsg)
```

Una vez ejecutado este algoritmo, con la finalidad de obtener un mapa que nos de la información del recorrido del robot y de los escáneres obtenidos del robot, se crea un *OccupancyMap*, es decir, un mapa de cuadrícula de ocupación en 2D, en el que cada celda de la cuadrícula de ocupación tiene un valor que representa la probabilidad de ocupación de esa celda, siendo los valores cercanos a 1 los que representan una alta probabilidad de que esa celda se encuentre ocupada por un obstáculo mientras que valores cercanos a 0 indican una alta probabilidad de que la celda se encuentre libre de obstáculos [18].

```
[scans,optimizedPoses]  = scansAndPoses(slamAlg);
map = buildMap(scans,optimizedPoses,mapResolution,
      maxLidarRange);
```

Los resultados se muestran en las Figuras 3.11 a 3.15 partiendo del Entorno 1.

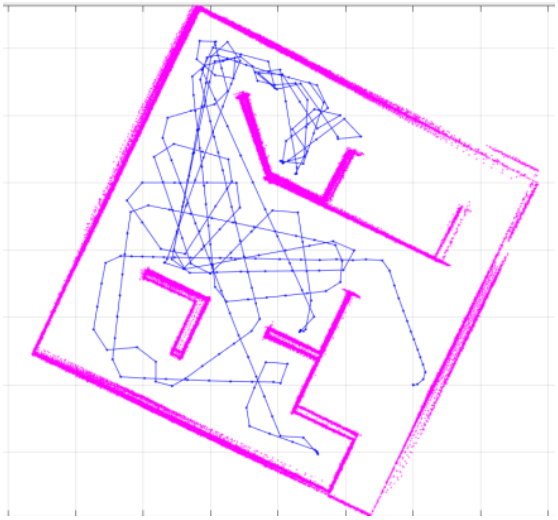

Figura 3.11 *Pose-graph y las mediciones del sensor láser obtenidos del recorrido del robot en el Entorno 1*

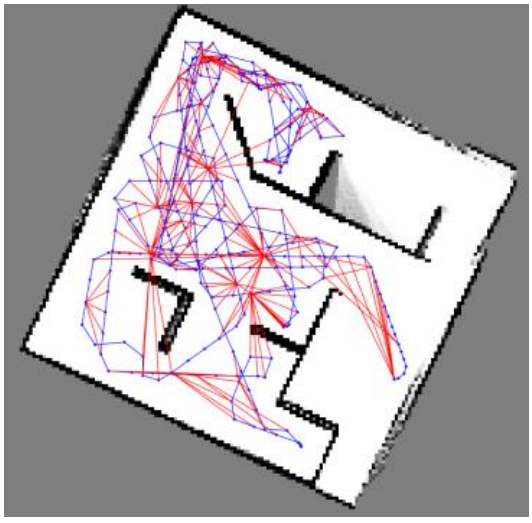

Figura 3.12 *OccupancyMap creado a partir del pose-graph y las mediciones del sensor láser en el Entorno 1*

En la Figura 3.11 se puede ver en azul los nodos y los arcos de odometría guardados en el *pose-graph* durante el recorrido del robot, los nodos son representados por los puntos y los arcos son las líneas rectas que unen los nodos. En rosa se representan las lecturas del sensor láser cada vez que se detecta un obstáculo. En la Figura 3.12 se muestra el *pose-graph* y el mapa. El mapa es un *OccupancyGrid* y el *pose-graph* muestra los arcos de odometria en azul y de cerrado de bucle en rojo. En las Figuras 3.13 y 3.14 se muestra con más detalle un ejemplo de cerrado de bucle que se encuentra en la Figura 3.13 y en la Figura 3.14 se muestra ampliado. Se generarán cerrados de bucle cuando se escaneen regiones mapeadas previamente, pues un arco de cerrado de bucle conecta nodos no consecutivos, estableciendo una restricción entre ellos.

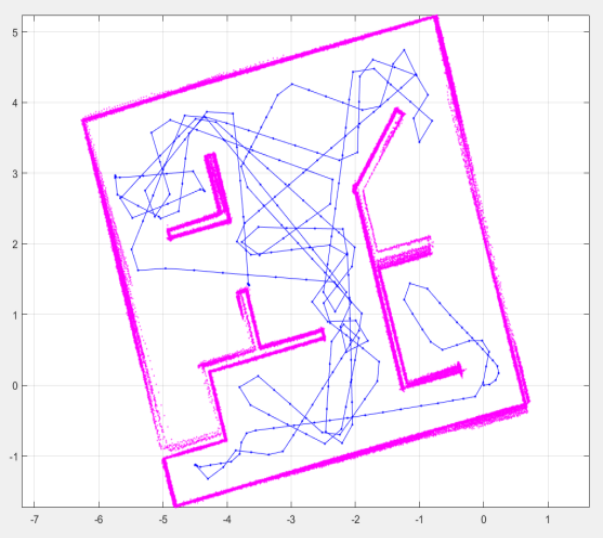

Figura 3.13 *Pose-graph y las mediciones del sensor láser obtenidos del recorrido del robot en el Entorno 1*

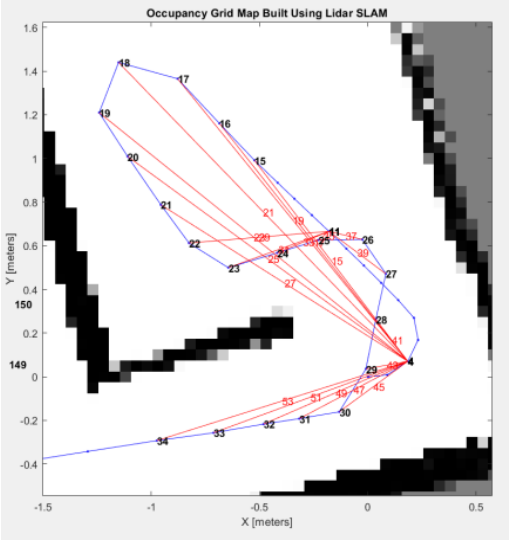

Figura 3.14 *Arco de cerrado de bucle en rojo del pose-graph*

Si un bucle no se cierra correctamente pueden aparecer superposiciones en el mapa dando lugar a una obtención de un mapa erróneo como se puede apreciar en la Figura 3.15.

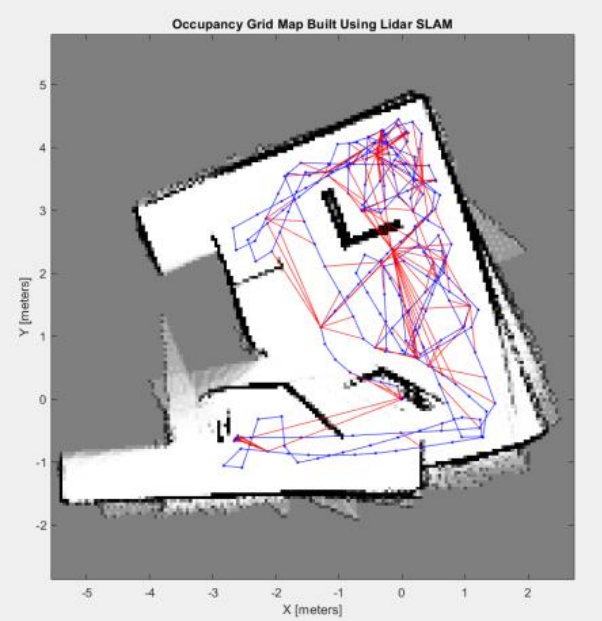

Figura 3.15 *OccupancyMap con error en cierre de bucle*

Esta superposición se debe a que algoritmo de SLAM aplicado ha identificado mal una zona, pues al no saber dónde se encuentra el robot exactamente e ir analizando el entorno a la vez que lo recorre, se puede dar la situación de que dos zonas del entorno que no son la misma tengan un parecido muy similar y el algoritmo de SLAM las identifique como iguales, creando así una restricción sobre dos nodos que el algoritmo ha identificado como cercanos.

Una posible solución para este problema (falso positivo) es el aumento del valor del umbral de cierre de lazo *LoopClosureThreshold*, ya que de esta manera se reduce la posibilidad de identificar un cerrado de bucle incorrecto al exigir mayor parecido a la información de los sensores a la hora de establecer que por esa zona ya se ha pasado. Otra forma de solucionarlo sería disminuyendo el radio de búsqueda de cerrados de bucle *slamAlg.LoopClosureSearchRadius* reduciendo así el rango de búsqueda de arcos de cerrado de bucle.

4. Implementación de algoritmos de SLAM Activo

En este capítulo de la memoria, se abordarán las tres fases de SLAM Activo por separado. Dichas fases son:

- Identificación de posibles localizaciones a explorar.
- Cálculo de la utilidad de las opciones disponibles y selección de la que sea de mayor utilidad para el robot.
- Guiado del robot hacia la opción de mayor interés escogida.

4.1. Detección de fronteras

El objetivo de este apartado consiste en la identificación de las fronteras en un entorno desconocido a partir de un mapa de ocupación. Mediante esta detección de fronteras será posible identificar los puntos de interés a los que el robot podrá dirigirse. Se entiende por frontera aquel punto que se encuentre entre zona conocida y zona desconocida.

Para la realización de este apartado, se ha partido de dos mapas distintos, uno de ellos obtenido de la biblioteca de MATLAB y el otro obtenido a través del escáner láser del robot de la misma forma que en el apartado 3.3.2 se obtenían los *OccupancyMap*. Estos mapas se muestran en las Figuras 4.1 y 4.2.

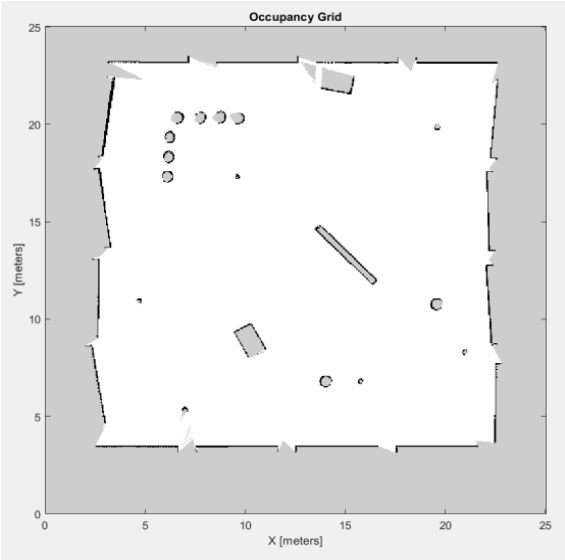

Figura 4.1 *Mapa obtenido de MATLAB*

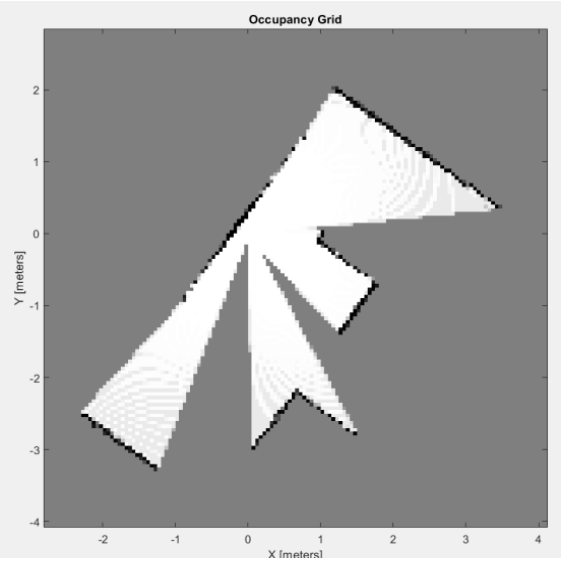

Figura 4.2 *Mapa obtenido del escáner láser del Entorno 1*

Para proceder a la obtención de fronteras se ha construido la matriz de ocupación de ambos mapas mediante la función *occupancyMatrix*. La finalidad de esta función es crear una matriz en la que cada elemento de esta sea una celda del mapa. Los valores respectivos de cada elemento de la matriz indican el estado de esa celda en el mapa:

- La celda estará ocupada si su valor es 1.
- La celda estará libre si su valor es 0.
- La celda será desconocida si su valor es -1.

Una vez obtenida esta matriz, se ha procedido a desarrollar un algoritmo que localice las fronteras, recorriendo secuencialmente la matriz buscando las celdas libres que se encuentren adyacentes a una celda de valor desconocido, ya que este sería el último punto conocido del mapa antes de entrar en un terreno desconocido.

Para cada celda libre se han analizado las 8 celdas que se encuentran a su alrededor en búsqueda de celdas de valor desconocido, es decir, si el valor de alguna de estas celdas es -1. La ubicación de estas celdas de valor desconocido se ha guardado en un nuevo vector, el cual se ha ido actualizando con todas las posiciones de la matriz en las que es desconocida la ocupación de la celda o dicho en otras palabras, la ubicación de las fronteras en dicha matriz. Una vez realizado esto, se ha empleado la función *grid2world* para saber en qué punto del espacio (x, y) se encuentra cada frontera obtenida de la forma (i, j) representando las filas y las columnas de la matriz respectivamente.

El resultado de aplicar el algoritmo de detección de fronteras se muestra en las Figuras 4.3 y 4.4. Las fronteras son representadas con cruces rojas. Como se puede apreciar en estas figuras hay una gran cantidad de fronteras, lo cual complica el proceso de elección de la más favorable. Es por ello, por lo que se busca agrupar fronteras que se encuentren cercanas. Se estudian dos tipos distintos de *cluster* o agrupación: *MeanShiftCluster* y *clusterXYpoints* cuya finalidad es la misma, agrupar las fronteras que se encuentren juntas para reducir y facilitar las futuras decisiones del robot. En este apartado, se estudian ambos métodos de agrupación con la finalidad de analizar cuál de ellos da mejor resultado.

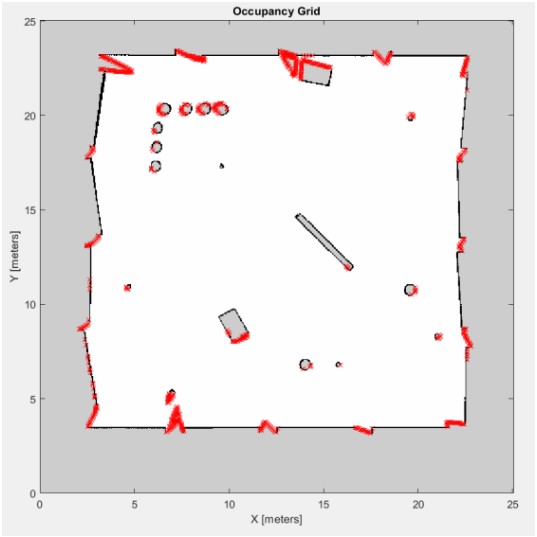

Figura 4.3 *Fronteras en el mapa de MATLAB*

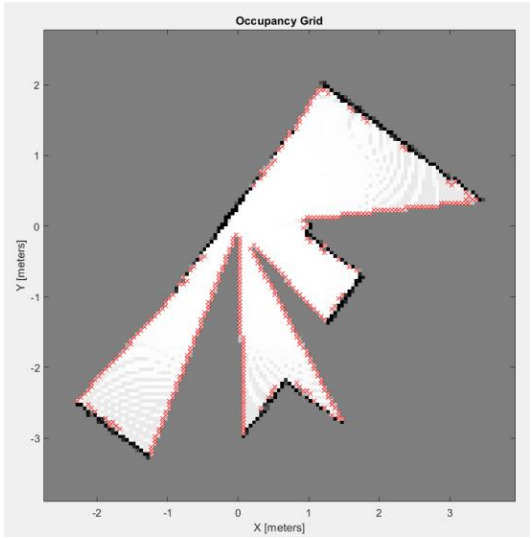

Figura 4.4 *Fronteras en el mapa obtenido del escáner del Entorno 1*

En el proceso de agrupación de fronteras se han eliminado aquellos *clusters* cuya cantidad de fronteras agrupadas fuese inferior a 6, pues se ha considerado que un número inferior a ese es una agrupación de fronteras de poco interés para la exploración. A continuación, se muestran los dos métodos mediante los que se ha realizado la agrupación de fronteras y los distintos resultados obtenidos:

Método 1: *MeanShiftCluster*

El método *MeanShiftCluster* calcula la media de un grupo de puntos que están dentro de una distancia determinada. Dando lugar al siguiente resultado que se muestra en las Figuras 4.5 y 4.6.

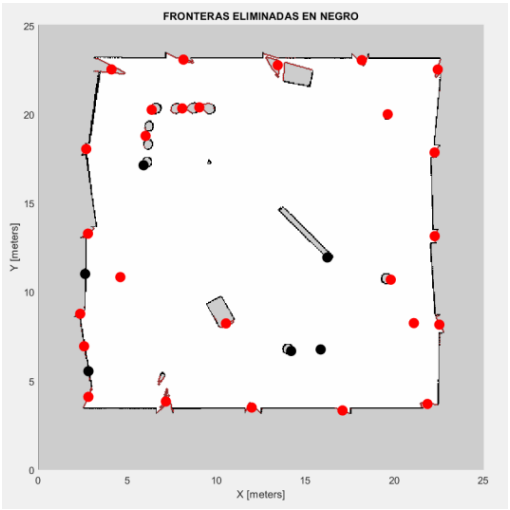

Figura 4.5 *Agrupación de fronteras con el método MeanShiftCluster en el mapa de MATLAB*

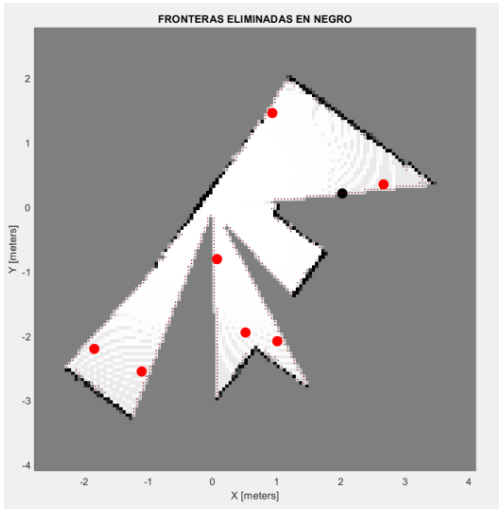

Figura 4.6 *Agrupación de fronteras con el MeanShiftCluster en el mapa del escáner láser del Entorno 1*

En rojo se representan las fronteras de interés ya que agrupan una cantidad superior a 6 fronteras y en negro se muestran las fronteras que son prescindibles y que se han eliminado.

Método 2: *clusterXYpoints*

El método *clusterXYpoints* calcula el centroide de un grupo de puntos que están dentro de una distancia determinada. Se ha obtenido el siguiente resultado.

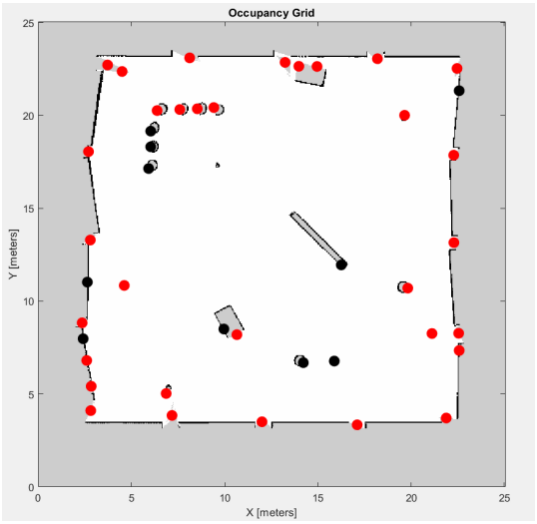

Figura 4.7 *Agrupación de fronteras con método clusterXYpoints en el mapa de MATLAB*

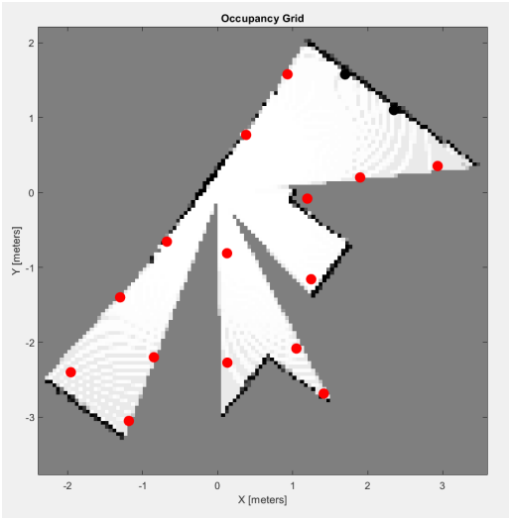

Figura 4.8 *Agrupación de fronteras con método clusterXYpoints en el mapa del escáner láser del Entorno 1*

Como con el método anterior, en rojo se ven las fronteras con una agrupación de fronteras superior a 6 y en negro las fronteras prescindibles y que se han eliminado.

Una vez aplicados los *clusters*, se ha de comprobar si en el *occupancyGrid* la celda correspondiente a la agrupación de fronteras es una celda libre, ocupada o desconocida. Interesa que esta celda esté libre, pues es la celda a la que se enviaría al robot en caso de decidir ese destino como frontera óptima. Es por esto por lo que el algoritmo creado evaluará cada una de las fronteras buscando que éstas sean celdas libres y en caso de que no lo sean, se moverá a la celda contigua libre más cercana. En el supuesto de que no hubiese ninguna celda contigua libre esta frontera se descartaría pues no se podría enviar al robot cerca de esta frontera para explorarla.

Finalmente, los resultados obtenidos se muestran en las Figuras 4.9 a 4.12, donde las estrellas rojas indican en el mapa la localización de las fronteras detectadas tras el agrupamiento.

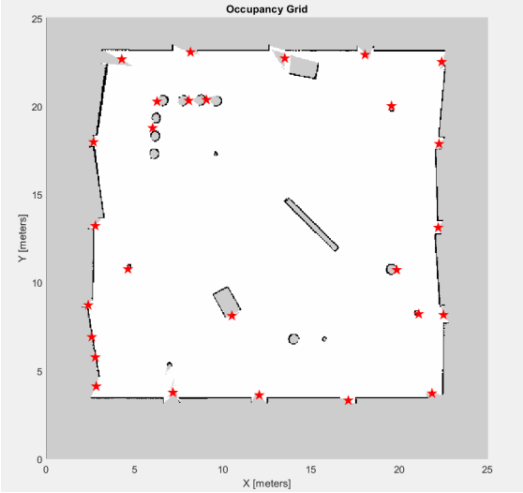

Figura 4.9 *Fronteras finales en el mapa de MATLAB con el método MeanShiftCluster*

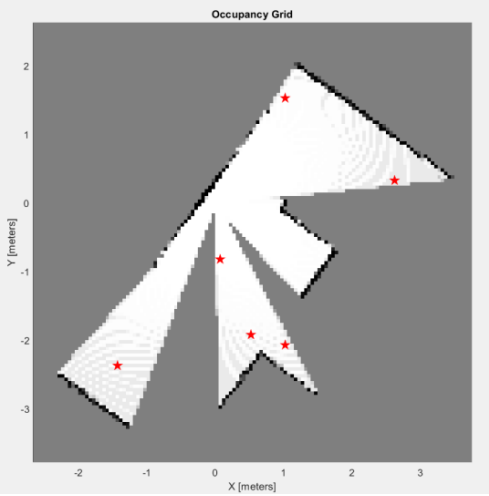

Figura 4.10 *Fronteras finales en el mapa del escáner láser del Entorno 1 con el método MeanShiftCluster*

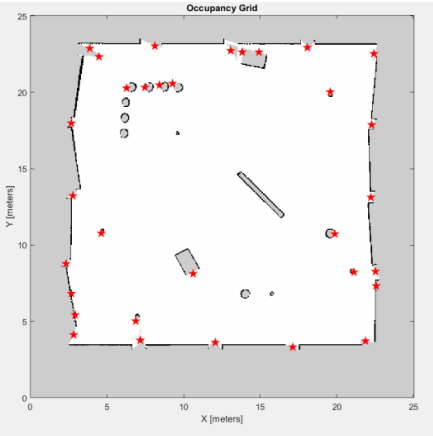

Figura 4.11 *Fronteras finales en el mapa de MATLAB con el método clusterXYpoints*

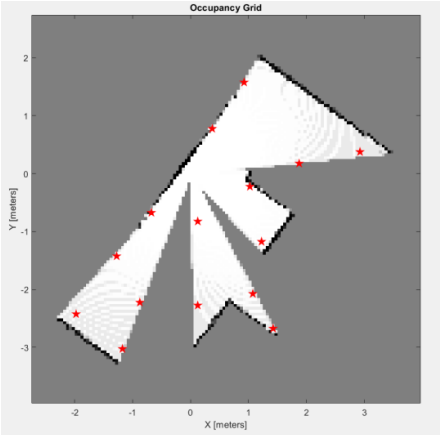

Figura 4.12 *Fronteras finales en el mapa del escáner láser del Entorno 1 con el método clusterXYpoints*

La diferencia encontrada entre ambos métodos consiste en que en el método *clusterXYpoints* se obtiene un mayor número de fronteras, y que estas siempre se encuentran en el mismo punto del mapa, mientras que con el método *MeanShiftCluster* el número de fronteras finales obtenidas es menor, y las fronteras obtenidas no siempre se encuentran en la misma posición para un mismo mapa dado. Ambos métodos poseen características interesantes para el estudio de un entorno desconocido, sin embargo, a partir de este apartado solamente se usará *clusterXY* ya que sus resultados se encuentran siempre en el mismo punto del mapa y el método *MeanShiftCluster* da como resultado un número de fronteras demasiado bajo. En el Anexo I se muestra el algoritmo creado que realiza esta función de detección de fronteras.

4.2. Selección del destino de mayor utilidad

La finalidad de este apartado es, una vez detectadas las fronteras, estudiar para cada una de ellas su utilidad. Se estudiará tanto la cantidad de celdas desconocidas que se encuentran alrededor de dicha frontera como el coste del robot de desplazarse desde su posición hasta el punto en el que se encuentra la frontera a estudiar.

En este apartado se plantean cuatro funciones de utilidad, la primera a mencionar, es la función de utilidad que se basa en el criterio de selección de forma aleatoria de la frontera.

La segunda función de utilidad se basa en el estudio del coste de desplazarse hacia cada una de las fronteras, para lo cual, se ha de calcular el camino a cada una de ellas, siendo la frontera de mayor interés en este caso aquella que presente la menor distancia desde la posición del robot, pues el coste de desplazamiento será menor. Para ello se busca maximizar la función de utilidad mediante la minimización de la distancia euclídea del camino a recorrer, donde n es la cantidad de puntos (*waypoints*) que contiene el camino creado *path* y x e y son las posiciones de dichos puntos.

$$U_C = -\sum_{x,y \in path}^{n} \sqrt{(x_{n+1} - x_n)^2 + (y_{n+1} - y_n)^2} \tag{5}$$

La tercera función de utilidad estudia la cantidad de celdas desconocidas que se encuentren alrededor de la posición de la frontera. Interesará elegir la frontera que mayor información ofrezca, o lo que es lo mismo, la que menor entropía genere ya que a mayor cantidad de celdas desconocidas que haya en la frontera elegida mayor información sobre el entorno será conocida. Se habla en términos de entropía ya que en aproximaciones en las que se usan mapas de ocupación de celdas, *occupancyMaps*, la entropía es aproximadamente igual al número de celdas desconocidas. La fórmula aplicada se muestra a continuación donde para maximizar U_H es necesario minimizar la entropía definida como el sumatorio de celdas desconocidas.

$$U_H = -\sum_{c \epsilon M} p(b(c)) \tag{6}$$

Esta función representa el sumatorio en todas las celdas del mapa de la probabilidad de que la celda *c* esté ocupada. Donde, *b(c)* es una distribución binaria que indica si una celda esta libre u ocupada y *p(b(c))* es la probabilidad de dicha distribución, se obtendrá un valor igual a 0 cuando la celda sea conocida y un valor igual a 1 cuando la celda sea desconocida.

Además de las comentadas previamente, se ha estudiado una función utilidad que combina la información y el coste. El robot estudia ambas opciones a la vez y en función de la distancia (coste) y la información elige la de mayor interés. La fórmula utilizada que relaciona estas variables se muestra a continuación donde la utilidad es representada por la letra U, el coste U_c y la entropía o información U_H.

$$U = U_H / U_c \tag{7}$$

$$U= U_H +0.25/ U_c \tag{8}$$

En este apartado, se muestra en las Figuras 4.13 a 4.16 un mapa de ocupación generado del Entorno 1 en el que se representan las distintas fronteras detectadas (estrellas rojas) y qué frontera se ha escogido (estrella azul) para cada uno de los criterios comentados anteriormente, en todas ellas el robot se encuentra en la posición (0,0) del mapa, representado en él con una estrella de color negro. La Figura 4.13 se corresponde con el criterio aleatorio, la Figura 4.14 con el criterio basado en el coste, la Figura 4.15 con el criterio basado en la entropía y la Figura 4.16 se corresponde con el criterio basado en el coste y en la entropía. Al ser un mapa de ocupación, las zonas blancas del mapa representan las zonas libres, las negras, zonas ocupadas y las grises, zonas desconocidas.

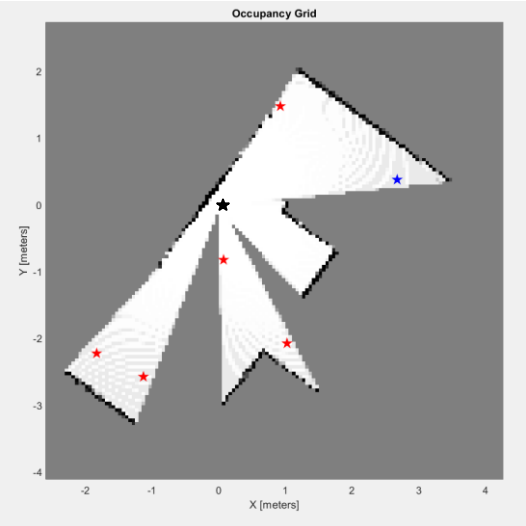

Figura 4.13 *Frontera seleccionada de forma aleatoria*

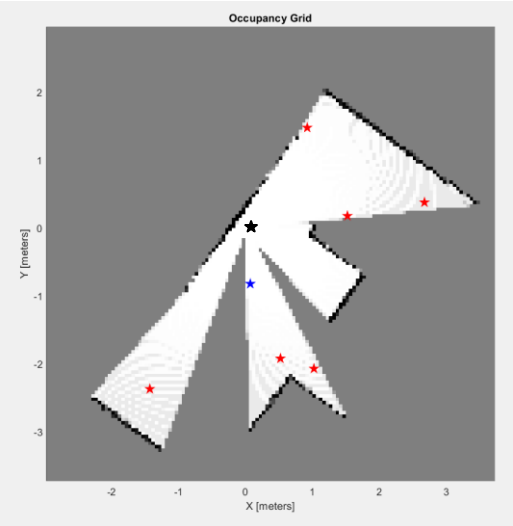

Figura 4.14 *Frontera seleccionada mediante la función de utilidad que se basa en el coste*

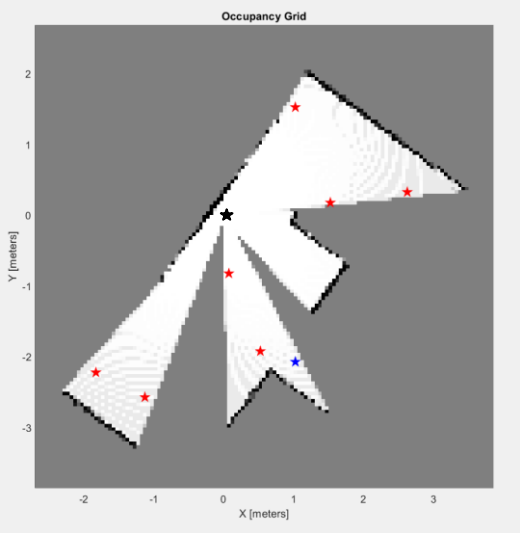

Figura 4.15 *Frontera seleccionada mediante la función de utilidad que se basa en la entropía*

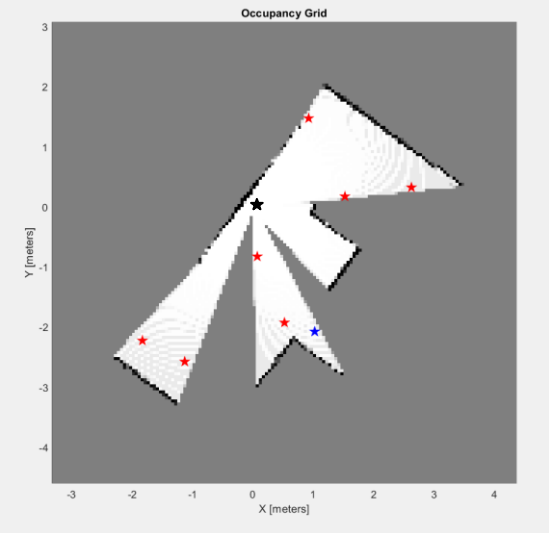

Figura 4.16 *Frontera seleccionada mediante la función de utilidad que combina la entropía y el coste*

De las cuatro funciones de utilidad expuestas, la función de elección de la frontera deseada de forma aleatoria es la de mayor rapidez, pues no debe de realizar ninguna comprobación. Los otros tres criterios utilizados para aplicar la función de utilidad tardan un tiempo muy parecido. Como se puede ver en las Figuras 4.15 y 4.16, ante la información obtenida de este mapa, el resultado de la frontera elegida es el mismo, pues en este caso la frontera con mayor entropía interesa más que cualquier otra frontera con menor entropía pero menor distancia en su recorrido para la función de utilidad basada en la mezcla entre la entropía y el coste. El algoritmo redactado para la selección de la frontera de mayor utilidad en función del criterio aplicado se muestra en el Anexo II.

4.3.Mover al robot hacia el destino seleccionado

La finalidad de este apartado es, una vez conocida la frontera de mayor interés a estudiar, crear un camino que conecte la actual posición del robot con la posición deseada y proceder a guiar al robot a través de ese camino generado a su destino final.

Para crear un camino que conecte la posición en la que se encuentra el robot con la posición final deseada, es necesaria la utilización de algoritmos de planificación de trayectorias cuya función es permitir que un vehículo autónomo encuentre la ruta más corta y libre de obstáculos. Se estudiarán dos algoritmos de planificación, el planificador PRM (*Probabilistic Roadmap*) y el planificador RRT (R*apidly Exploring Random Trees*) con el fin de analizar ambos métodos y ver cuál de ellos da lugar a una implementación del algoritmo de exploración autónoma más eficaz. A partir de un punto de inicio, de un punto final y de un mapa en el que se localizan estos puntos, ambos métodos se encargan de generar un camino (*path*) con posiciones muy cercanas entre sí por las que el robot deberá pasar, habiendo ya realizado una evasión de los obstáculos que aparecen en el mapa.

El planificador PRM construye una hoja de ruta en el espacio libre que hay en un mapa determinado mediante la utilización de nodos muestreados de forma aleatoria en el espacio libre y conectando estos nodos entre sí. El número de nodos PRM que se usan en la construcción de la hoja de ruta dependen de la dimensión y complejidad del mapa, una gran cantidad de nodos crea una hoja de ruta densa y aumenta la probabilidad de encontrar una ruta, aunque el hecho de tener más nodos afecta directamente al tiempo de cálculo de la creación de la hoja de ruta; a más nodos, más tiempo se necesitará [19].

El planificador *RRT* está basado en árboles, crea un árbol de búsqueda de forma incremental a partir de muestras extraídas de un espacio de estados determinado de forma aleatoria. El árbol se extiende por el espacio de búsqueda y conecta el estado inicial (q_{init}) con el estado objetivo. El proceso general de crecimiento de los árboles es el siguiente:

1. El planificador muestrea un estado aleatorio (q_{target}) en el espacio de estados, es decir, en el espacio de configuración libre del mapa de entorno.
2. Busca un estado que ya esté en el árbol de búsqueda y sea el más cercano (q_{near}).
3. Se expande desde el estado cercano hacia el estado aleatorio hasta que alcanza un nuevo estado (q_{new}).
4. Este nuevo estado se agrega al árbol de búsqueda.

En la Figura 4.17 se muestran los estados mencionados anteriormente en un diagrama visual.

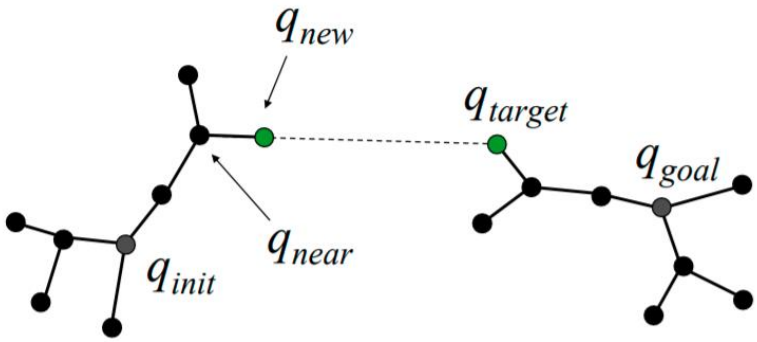

Figura 4.17 *Diagrama visual del planificador RRT [20]*

Planificador RRT

Con el planificador RRT se ha introducido un mapa y los puntos iniciales y finales los cuales se encuentran dentro del mapa, y se ha obtenido un camino. Para la obtención de un camino de la mejor forma posible se ha procedido a variar el valor de la variable *MaxConnectionDistance*, que representa la longitud máxima de movimientos, es un valor escalar que se da en metros. En las Figuras 4.18 y 4.19 se le ha dado un valor de 0.3 m con lo que se puede apreciar una consecución de puntos más cercanos entre los estados del camino creado. En cambio en las Figuras 4.20 y 4.21 al ser mayor la distancia de conexión, 0.8 m, se ve mayor restricción a la hora de crear el camino. El mapa de la Figura 4.18 se ha obtenido a través de un escaneo del sensor láser del Entorno 1 y el mapa de la Figura 4.19 se ha obtenido de la biblioteca de MATLAB *exampleMaps.*

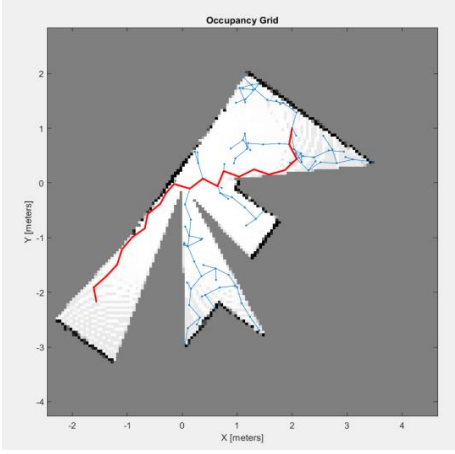

Figura 4.18 *Mapa obtenido del escáner láser del Entorno 1 que muestra el path creado con el planificador RRT con MaxConnectionDistance = 0.3*

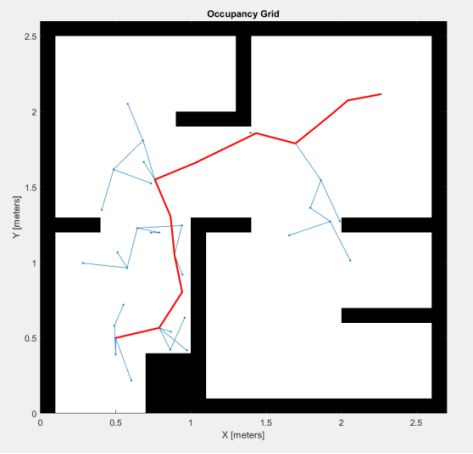

Figura 4.19 *Mapa obtenido de la biblioteca de MATLAB que muestra el path creado con el planificador RRT con MaxConnectionDistance = 0.3*

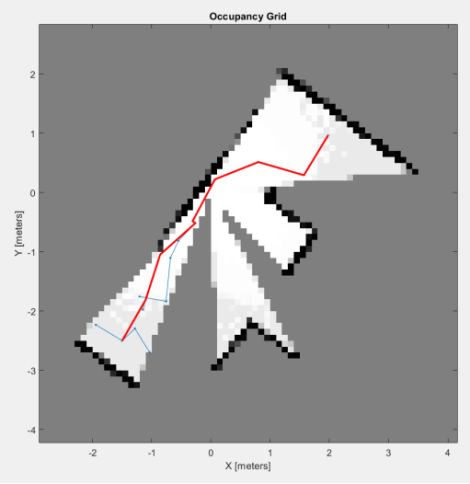

Figura 4.20 *Mapa obtenido del escáner láser del Entorno 1 que muestra el path creado con el planificador RRT con MaxConnectionDistance = 0.8*

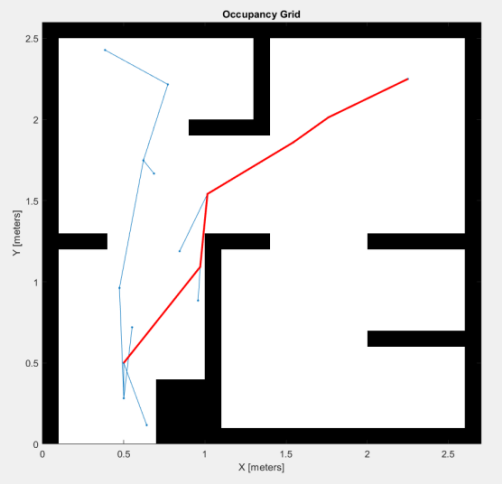

Figura 4.21 *Mapa obtenido de la biblioteca de MATLAB que muestra el path creado con el planificador RRT con MaxConnectionDistance = 0.8*

En las cuatro figuras mostradas, las líneas de color azul representan las ramificaciones del árbol que no llevan al punto deseado, mientras que las líneas de color rojo representan el camino planificado que se enviará al robot. Para un valor bajo de la variable *MaxConnectionDistance* el camino creado tiene muchos *waypoints*, puntos a seguir para completar el camino, lo que hará que el robot tenga que comprobar constantemente si ha llegado al *waypoint* siguiente, lo cual ralentizará el proceso computacional. Es por esto por lo que para el desarrollo de este trabajo se ha decidido usar un valor de *MaxConnectionDistance* igual a 0.8 m. En la Tabla 1 se muestran los demás valores relevantes de las variables utilizadas para este método. Donde *MaxIterations* es el máximo número de iteraciones a realizar, *MaxConnectionDistance* es la máxima distancia que habrá entre los puntos a seguir de la trayectoria creada, *GoalReachedFcn* se encarga de evaluar si se ha alcanzado el destino, *@exampleHelperCheckIfGoal* es la función predeterminada de *GoalReachedFcn* llamada para verificar si el estado de uno de los nodos del RRT se encuentra lo suficientemente cerca del objetivo para dar por completado el cálculo y *ValidationDistance* que es la distancia de validación.

Variables	**Valor**
MaxIterations	4000
MaxConnectionDistance	0.8 m
GoalReachedFcn	*@exampleHelperCheckIfGoal*
ValidationDistance	0.05 m

Tabla 1 *Variables del método RRT*

Planificador PRM

Al igual que con el planificador RRT, con el planificador PRM se ha introducido un mapa y los puntos iniciales y finales los cuales se encuentran dentro del mapa, y se ha obtenido un camino. Para conocer cuáles eran las mejores condiciones que se debían aplicar para conseguir un camino eficaz, se ha variado la cantidad de nodos ya que estos afectan tanto al tiempo de ejecución como a la sencillez del camino creado. En las Figuras 4.22 y 4.23 se muestran los resultados obtenidos tras aplicar el planificador PRM con número de nodos igual a 10. El mapa de la Figura 4.22 es el mapa de ocupación obtenido por el robot del escáner láser del Entorno 1. Para todas las pruebas a realizar con el planificador PRM de este mapa se tratará de encontrar un camino entre el punto inicial (2, 1) y el punto final (-1.7, -2.7). El mapa de la Figura 4.23 se ha obtenido de la biblioteca de MATLAB *exampleMaps*.

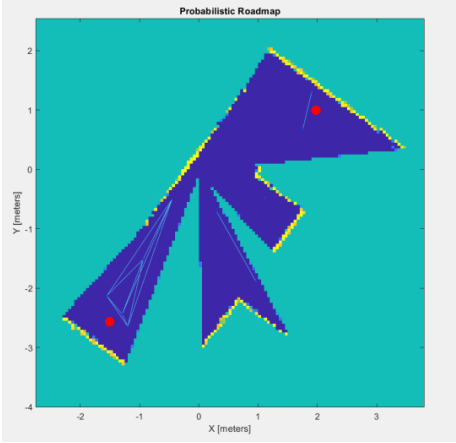

Figura 4.22 *Mapa obtenido del escáner láser del Entorno 1 que muestra el path creado con el planificador PRM con 10 nodos*

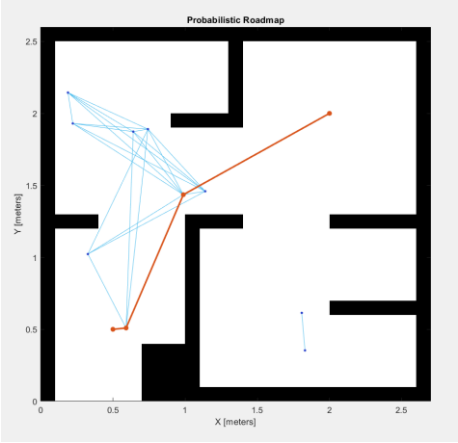

Figura 4.23 *Mapa de MATLAB que muestra el path creado con el planificador PRM con 10 nodos*

A pesar de que en el mapa de la Figura 4.23 sí que es posible crear un *path*, en el mapa de la Figura 4.22 se ve que esa cantidad de nodos es insuficiente para construir un camino entre los dos puntos (en rojo), por lo que se descarta el valor de 10 nodos. En las Figuras 4.24 y 4.25 se pueden ver los mismos mapas pero con más nodos y por lo tanto más conexiones entre ellos.

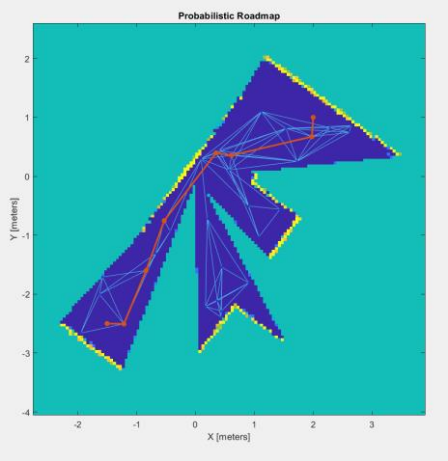

Figura 4.24 *Mapa obtenido del escáner láser del Entorno 1 que muestra el path creado con el planificador PRM con 30 nodos*

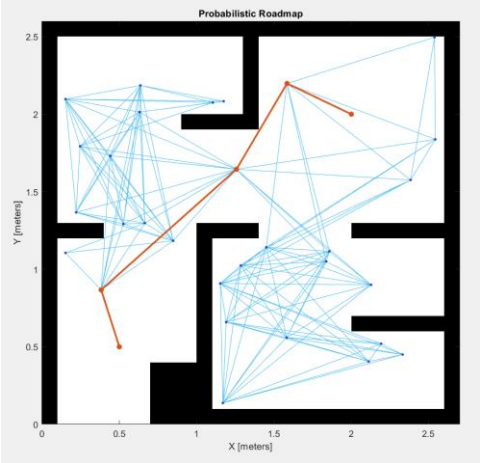

Figura 4.25 *Mapa de MATLAB que muestra el path creado con el planificador PRM con 30 nodos*

Con estos resultados, es posible la creación de un *path* en ambos mapas, por lo que 30 es el número mínimo de nodos aceptable para este tamaño de mapa. Por debajo de esta cantidad existen dificultades para la consecución de un *path*, además al ser un número de nodos no muy elevado, el tiempo de computación es bajo. Tras ver los resultados de aplicar un número de nodos bajo, se procederá a aplicar el planificador con un número más elevado de nodos para estudiar si al aumentar la cantidad de nodos también aumenta significativamente la eficacia del camino creado. En las Figuras 4.26 y 4.27 se muestran los resultados de los caminos generados para los mapas al aumentar el número de nodos a 60 y en las Figuras 4.28 y 4.29 se muestran los resultados de los caminos generados para los mapas al aumentar considerablemente el número de nodos a 250.

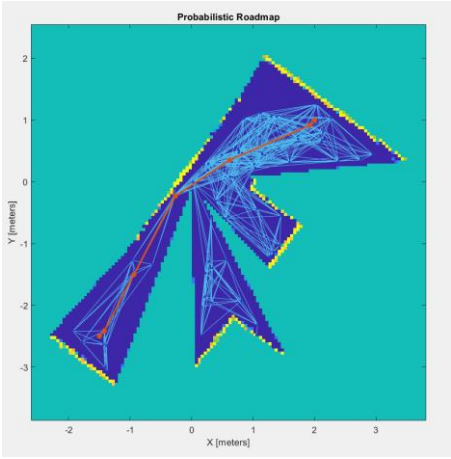

Figura 4.26 *Mapa obtenido del escáner láser del Entorno 1 que muestra el path creado con el planificador PRM con 60 nodos*

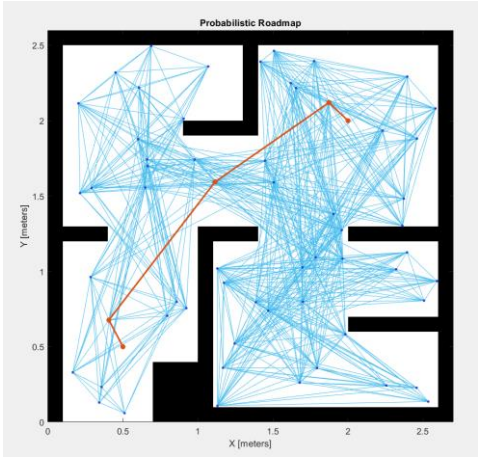

Figura 4.27 *Mapa de MATLAB que muestra el path creado con el planificador PRM con 60 nodos*

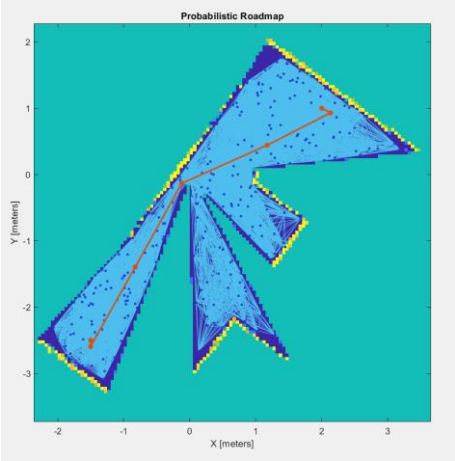

Figura 4.28 *Mapa obtenido del escáner láser del Entorno 1 que muestra el path creado con el planificador PRM con 250 nodos*

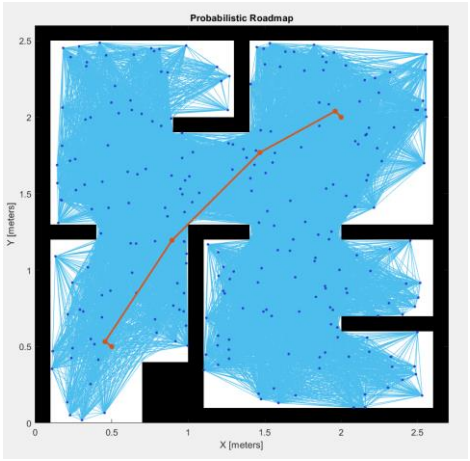

Figura 4.29 *Mapa de MATLAB que muestra el path creado con el planificador PRM con 250 nodos*

Tras estos cálculos se puede ver que para un número pequeño de nodos no se asegura que se pueda crear siempre un camino, sin embargo, a medida que se va aumentando el número de nodos, el camino mejora, pero llega a un punto en el que por más nodos que se añadan a dicho camino ya no mejora. Esto se puede ver en las Figuras 4.26, 4.27, 4.28, 4.29 ya que la diferencia del camino creado con 60 nodos y del camino creado con 250 nodos es casi imperceptible. Para los mapas mostrados en las imágenes anteriores, se escoge el valor de 30 nodos, ya que crea un camino eficaz con pocas iteraciones. Pero durante la realización de este proyecto, en los mapas obtenidos no siempre se encuentra un camino para esa cantidad de número de nodos por lo que interesará que esta cantidad aumente en el caso de que sea incapaz de encontrar un camino hacia la frontera deseada. Y es por ello por lo que se añade a la función que crea el camino la condición de que si para 30 nodos no hay camino que el número de nodos aumente 250 nodos más y así de forma consecutiva hasta encontrar un camino. Este algoritmo se muestra a continuación, donde *path* es el camino creado y *prm.NumNodes* el número de nodos.

```
while isempty(path) &&  prm.NumNodes < 1500
      prm.NumNodes = prm.NumNodes + 250;
      update(prm);
      path = findpath(prm, startLocation, endLocation);
end
```

En la Tabla 2 se muestran los valores iniciales relevantes de las variables para la consecución de una aplicación satisfactoria de este método. Donde *ConnectionDistance* es la distancia máxima que puede haber entre dos nodos conectados y *NumNodes* es la cantidad de nodos que se generarán en el mapa con el fin de obtener un camino.

Variables	Valor
ConnectionDistance	1.5 m
NumNodes	*30*

Tabla 2 *Variables del método PRM*

La implementación del código que se encarga de crear un camino entre dos puntos en función del algoritmo de planificación deseado se muestra en el Anexo III.

El algoritmo de planificación elegido para el desarrollo de este trabajo es el algoritmo de planificación PRM, pues en todos los experimentos realizados, siempre es capaz de encontrar un camino hacia las fronteras deseadas. En cambio, el método RRT no es capaz de encontrar siempre dichos caminos aunque en este método los caminos generados unen de una forma más directa el punto final y el inicial que el método PRM. Para la resolución de este trabajo se prima la consecución de un camino hacia todas las fronteras a estudiar antes que una trayectoria más directa al destino deseado, utilizando en el algoritmo desarrollado el método PRM.

Una vez obtenido el camino para unir los puntos deseados se procede a conectar con Gazebo-ROS para mover el robot secuencialmente por todas estas posiciones. Para ello se emplea el controlador *PurePursuit* [21]. El controlador *PurePursuit* se ha utilizado para conseguir que el robot siga un conjunto de puntos definido por el *path*, el cual será definido en el controlador de la forma *controller*.W*aypoints*, este controlador calcula las velocidades lineales y angulares del robot en función de la posición actual de este (información obtenida de la odometría). La variable *LookAheadDistance* controla la distancia a la que se coloca el punto de búsqueda anticipada. Para este trabajo, se le ha asignado un valor de 0.75 m ya que un valor mayor que este, da lugar a que mientras se ejecuta el algoritmo, el robot trate de intentar acortar el camino y termine colisionando con un obstáculo.

Las máximas velocidades angular y lineal a enviar al robot se establecen en 0.55 rad/s y en 0.16 m/s respectivamente, pues se ha comprobado que la combinación de estas dos velocidades es satisfactoria para este proceso de exploración. Además de establecer valores de velocidad máximos, se ha incluido una condición a estas, pues no interesa que mientras el robot gire para posicionarse se siga moviendo linealmente, ya que esto puede dar lugar a colisiones. Esta condición indica que cuando el robot se mueva con una velocidad angular superior a 0.4 rad/s, la velocidad lineal debe de bajar a 0.025 m/s, así el robot se mueve linealmente a una velocidad muy baja mientras gira hasta conseguir orientarse en la dirección deseada. Esta información se irá enviando a los *topics* de ROS en el *cmd_vel* para que el robot se mueva y se irá actualizando a medida que el robot vaya realizando su recorrido a través del *path.* A continuación se muestra un extracto del código realizado para la consecución de esta acción.

```
while( distanceToGoal > goalRadius )
     odomMsg = receive(odomSub,3);
     quat = [odomMsg.Pose.Pose.Orientation.W
      odomMsg.Pose.Pose.Orientation.X
      odomMsg.Pose.Pose.Orientation.Y eulZYX(1)];
     eulZYX = quat2eul(quat);
     robotCurrentPose= [odomMsg.Pose.Pose.Position.X
      odomMsg.Pose.Pose.Position.Y
      odomMsg.Pose.Pose.Orientation.Z]'
     [v, omega] = controller(robotCurrentPose);
     if omega>0.4
        v=0.025;
     end
     velMsg.Linear.X = v;
     velMsg.Angular.Z= omega;
     velPub.send(velMsg);
     t_scan=0.125;
     tic
     while toc< t_scan
       scanMsg = receive(lidarSub);
       range = double(lidarSub.LatestMessage.Ranges);
       angles = double(scanMsg.readScanAngles);
       range(range==Inf)= maxLidarRange+2;
       scans = lidarScan(range,angles);
       steerDir = vfh(scans, targetDir);
       if isnan(steerDir)
           Colision = true;
           break
       end
       send(robotCmd,velMsg)
       [isScanAccepted,loopClosureInfo,optimizationInfo] =
        addScan(slamAlg,scans);
       [scans,optimizedPoses]  = scansAndPoses(slamAlg);
        waitfor(controlRate);
     end
     if Colision == true
        break
     end
     velMsg.Linear.X = 0;
     velMsg.Angular.Z= 0;
     send(robotCmd,velMsg)
     odomMsg = receive(odomSub,3);
     quat = [odomMsg.Pose.Pose.Orientation.W
     odomMsg.Pose.Pose.Orientation.X
     odomMsg.Pose.Pose.Orientation.Y
     odomMsg.Pose.Pose.Orientation.Z];
     eulZYX = quat2eul(quat);
     robotCurrentPose= [odomMsg.Pose.Pose.Position.X
     odomMsg.Pose.Pose.Position.Y  eulZYX(1)]';
     distanceToGoal = norm(robotCurrentPose(1:2) -
     robotGoal(:));
end
```

Donde *goalRadius* es la máxima distancia aceptada a la que puede estar el robot de su posición final y es por eso que el bucle se ejecutará hasta que la variable *distanceToGoal*, sea menor que *goalRadius* lo cual indicará que el robot ha llegado a su destino final. Durante la ejecución de este código el controlador *PurePursuit* calcula las velocidades que se deben de enviar posteriormente al robot. Una vez llegado el robot a su posición deseada, se envía al robot la información de dejar de moverse y finalizar el proceso.

5. Experimentación

Una vez realizados los algoritmos, se procede a realizar las siguientes pruebas en los entornos de Gazebo creados con la finalidad de estudiar las distintas funciones de utilidad comentadas en el apartado 4.2 y ver cuál de ellas es más eficaz para el proceso de exploración de un entorno desconocido. En el Anexo IV se muestra el código completo del algoritmo de exploración utilizado para la realización de los experimentos.

Las distintas funciones a estudiar son las siguientes: la primera se basa en una elección de fronteras de forma totalmente aleatoria, en la segunda función, el robot solamente tiene en cuenta la distancia que hay desde su posición hasta la frontera y seleccionará aquella frontera que se encuentre más cerca, en la tercera, el robot tiene en cuenta la cantidad de celdas desconocidas que va a poder conocer a la hora de acercarse a la nueva frontera y la cuarta función se basa en una mezcla de estas dos últimas, según lo descrito en el apartado 4.2.

Para cada una de estas funciones se mostrará el mapa, la trayectoria, el tiempo que ha tardado en recorrer dicha trayectoria, la distancia y la cantidad de arcos de cerrado de bucle. Se realizará una comparación ordenada por entornos, comenzando por mostrar el Entorno 1 y los mapas generados para cada una de las funciones a estudiar, a continuación se mostrará el Entorno 2 con sus respectivos mapas, para finalmente terminar con los mapas generados del Entorno 3. El algoritmo de exploración se repite hasta que se ha completado el mapa del entorno, es decir, una vez que ya no se detectan más fronteras (condición de parada o *stopping criterion*).

5.1. Entorno 1

En la Figura 5.1 se muestra el Entorno 1 sobre el que se ha aplicado el algoritmo que ejecuta SLAM Activo. En las Figuras 5.2 a 5.4 y en la Tabla 3 se representan los resultados obtenidos focalizando la atención tanto en la trayectoria del robot a lo largo del entorno, como en el tiempo que se ha invertido en recorrerlo, como en la distancia recorrida, a la vez que se muestra la cantidad de arcos de cerrado de bucle para cada función de utilidad analizada.

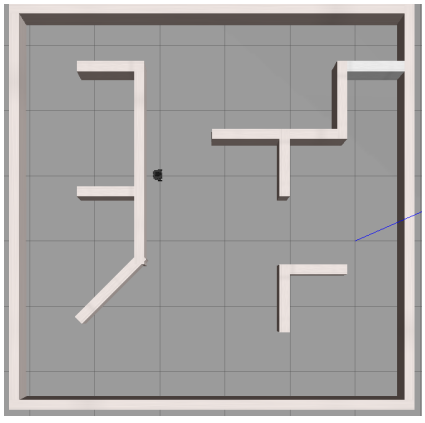

Figura 5.1 *Interfaz de Gazebo mostrando el Entorno 1*

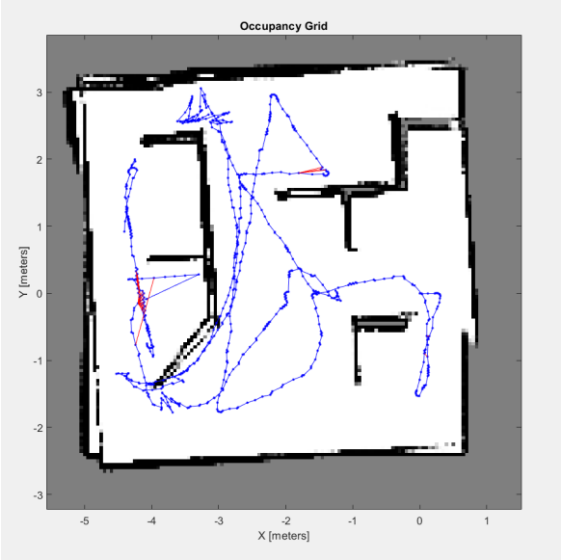

Figura 5.2 *Mapa del Entorno 1 obtenido del escáner láser tras la ejecución de la función de utilidad basada en la elección de frontera de forma aleatoria*

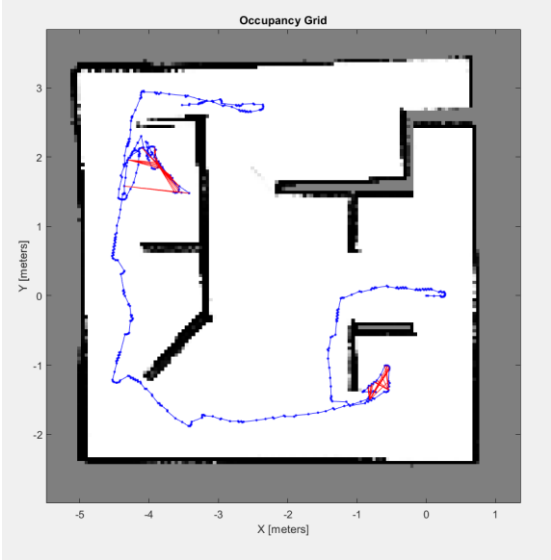

Figura 5.3 *Mapa del Entorno 1 obtenido del escáner láser tras la ejecución de la función de utilidad basada en el coste*

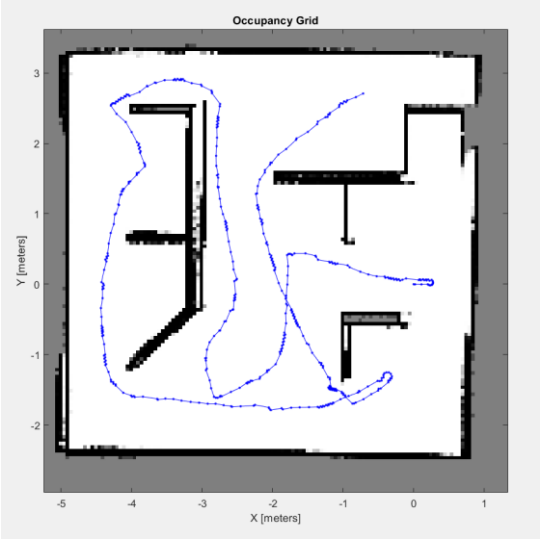

Figura 5.4 *Mapa del Entorno 1 obtenido del escáner láser tras la ejecución de la función de utilidad basada en la entropía*

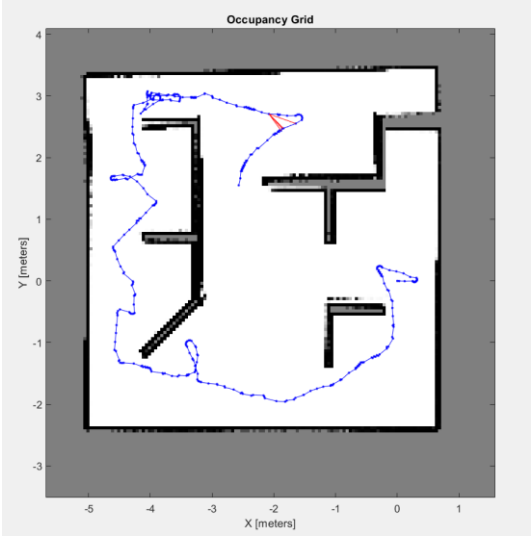

Figura 5.5 *Mapa del Entorno 1 obtenido del escáner láser tras la ejecución de la función de utilidad basada en la combinación de la función de coste y la de entropía*

Función de utilidad	Tiempo total normalizado	Distancia recorrida (metros)	Número de nodos	Número de arcos de cerrado de bucle
Aleatoria	1	52	862	32
Coste	0.41	23.85	462	57
Entropía	0.43	27	494	0
Mezcla de coste y entropía	0.31	21.85	382	4

Tabla 3 *Tabla de resultados del Entorno 1*

En cuanto a la función que elige la frontera de forma aleatoria, se observa que es la que más tiempo ha tardado en recorrer el Entorno 1. Además, observando la distancia recorrida, se puede apreciar que la elección de fronteras para este método no ha sido óptima, pues es la que mayor distancia ha recorrido casi duplicando el valor de la segunda mayor distancia recorrida, dando lugar a una trayectoria errática.

Los tiempos se han normalizado con respecto a la función basada en la aleatoriedad porque es la que más tiempo ha consumido. Tras los resultados obtenidos en la Tabla 3 se observa que en el Entorno 1 la función más eficaz es la función de utilidad basada en la mezcla entre la entropía y el coste dando como resultado un mapa completo en el menor tiempo y la menor distancia recorrida, 21.85 metros. El tiempo y la distancia recorrida resultantes tras aplicar las funciones basadas en coste y en entropía son similares.

La calidad visual del mapa obtenido tras la aplicación de la función de utilidad basada en la mezcla entre la entropía y el coste es alta, pues no presenta tantos errores de asociación de datos como se pueden apreciar en los demás mapas. Además de un tiempo corto de exploración, esta función da como resultado un mapa de buena calidad.

La duración del algoritmo está relacionada directamente con el recorrido realizado y por ende, con el número de nodos, pues como se puede apreciar en la Tabla 3 el menor valor de la cantidad de nodos se encuentra en la función basada en la mezcla de la entropía y el coste y el mayor valor se encuentra en la función de selección aleatoria de frontera ya que como se puede apreciar en la Figura 5.2 es la función que mayor recorrido ha realizado. La función basada en el coste es la que más arcos de cerrado de bucle presenta como se puede ver en la Figura 5.3. Aunque estos cerrados de bucle no son de interés para la obtención de información del entorno, ya que son generados entre nodos que se encuentran cercanos.

5.2.Entorno 2

A continuación, en la Figura 5.6 se muestra el Entorno 2. Los resultados obtenidos en este entorno tras la aplicación del algoritmo se muestran en las Figuras 5.7 a 5.10 y en la Tabla 4.

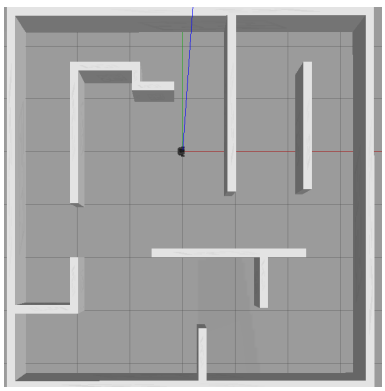

Figura 5.6 *Interfaz de Gazebo mostrando el Entorno 2*

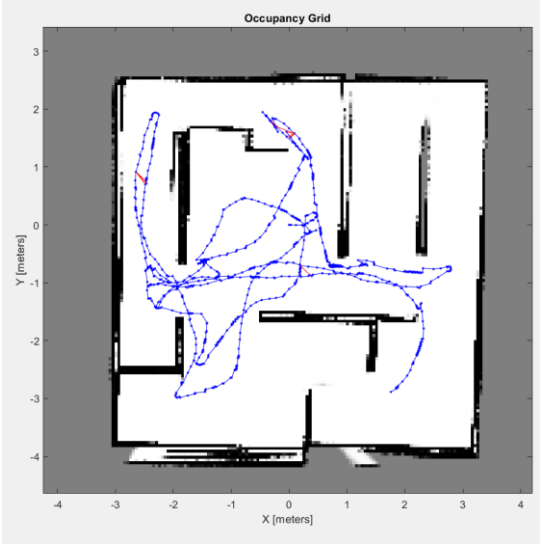

Figura 5.7 *Mapa del Entorno 2 obtenido del escáner láser tras la ejecución de la función de utilidad basada en la elección de frontera de forma aleatoria*

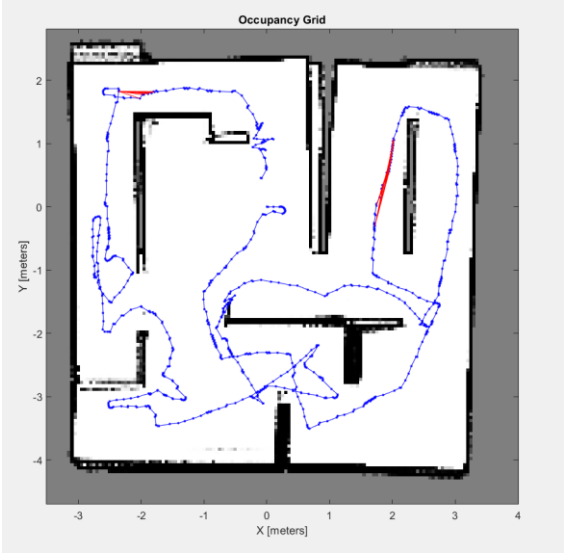

Figura 5.8 *Mapa del Entorno 2 obtenido del escáner láser tras la ejecución de la función de utilidad basada en el coste*

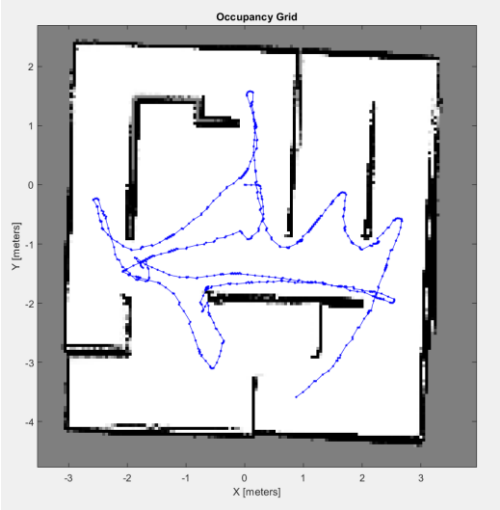

Figura 5.9 *Mapa del Entorno 2 obtenido del escáner láser tras la ejecución de la función de utilidad basada en la entropía*

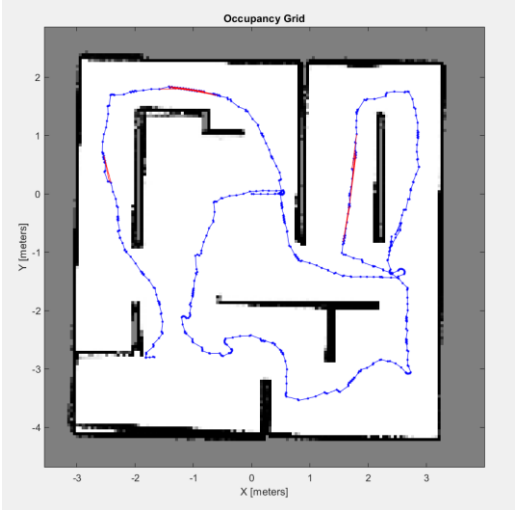

Figura 5.10 *Mapa del Entorno 2 obtenido del escáner láser tras la ejecución de la función de utilidad basada en la combinación de la función de coste y la de entropía*

Función de utilidad	Tiempo total normalizado	Distancia recorrida (metros)	Número de nodos	Número de arcos de cerrado de bucle
Aleatoria	1	55	893	11
Coste	0.72	47	826	37
Entropía	0.51	35	661	0
Mezcla de coste y entropía	0.43	33	606	30

Tabla 4 *Tabla de resultados del Entorno 2*

Tras los resultados obtenidos en la Tabla 4 se ha observado que en el Entorno 2 la función más eficaz vuelve a ser la misma que en el Entorno 1, la función de utilidad basada en la mezcla entre la entropía y el coste, pues a pesar de dar resultados similares a la función basada solamente en la entropía, la distancia recorrida en esta última es ligeramente superior como lo es también el tiempo de exploración. Al ser un entorno que presenta una mayor complejidad que el Entorno 1, la función basada en el coste, no da buenos resultados debido a que hay 3 caminos abiertos y está constantemente encontrando zonas cercanas a ellos a explorar a pesar de que la cantidad de fronteras a estudiar no sea relevante. La función basada en la aleatoriedad presenta una trayectoria muy errática como se puede observar en la Figura 5.7 pues pasa continuamente por zonas ya recorridas previamente que no aportan ninguna nueva información y solo aumentan la distancia recorrida a la vez que tiempo de computación.

5.3.Entorno 3

Finalmente se muestra el Entorno 3 en la Figura 5.11 y sus respectivos resultados en las Figuras 5.12 a 5.17 y en la Tabla 5.

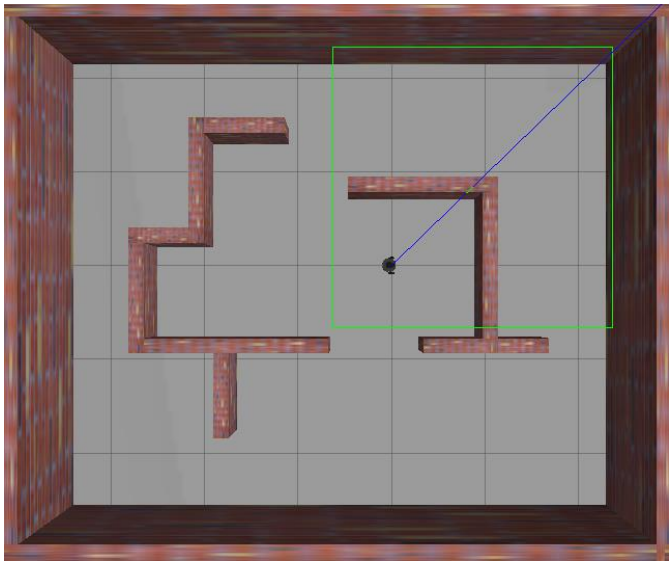

Figura 5.11 *Interfaz de Gazebo mostrando el entorno 3*

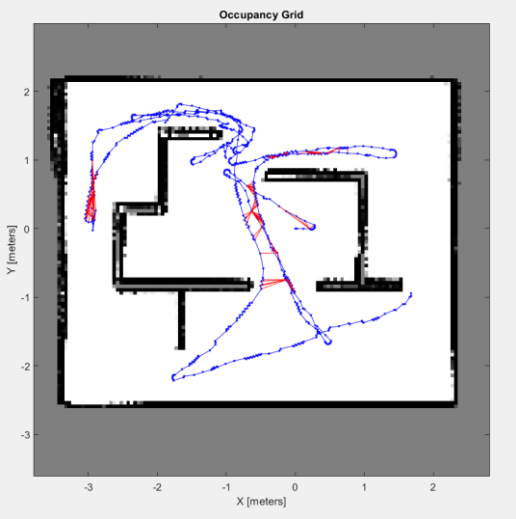

Figura 5.12 *Mapa del Entorno 3 obtenido del escáner láser tras la ejecución de la función de utilidad basada en la elección de frontera de forma aleatoria*

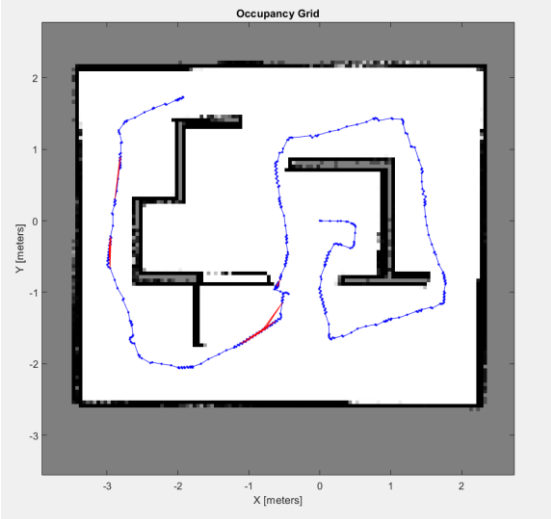

Figura 5.13 *Mapa del Entorno 3 obtenido del escáner láser tras la ejecución de la función de utilidad basada en la distancia*

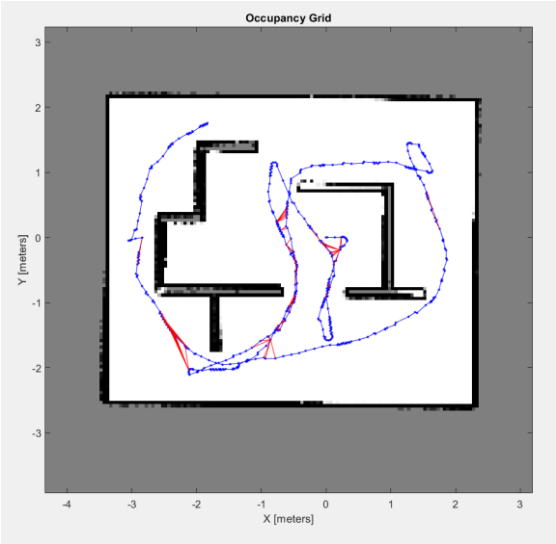

Figura 5.14 *Mapa del Entorno 3 obtenido del escáner láser tras la ejecución de la función de utilidad basada en la entropía*

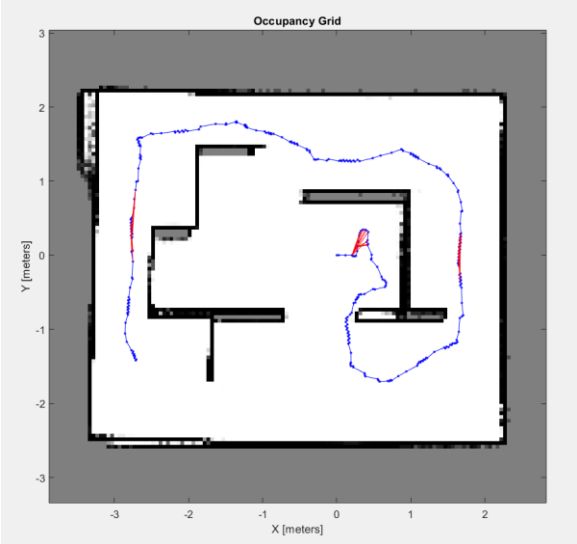

Figura 5.15 *Mapa del Entorno 3 obtenido del escáner láser tras la ejecución de la función de utilidad basada en la combinación de la función de coste y la de entropía*

Función de utilidad	Tiempo total normalizado	Número de nodos	Distancia recorrida (metros)	Número de arcos de cerrado de bucle
Aleatoria	1	763	39	128
Coste	0.41	428	20	45
Entropía	0.48	507	29	83
Mezcla de coste y entropía	0.26	310	15.4	33

Tabla 5 *Tabla de resultados del Entorno 3*

Analizando los resultados obtenidos y reflejados en la Tabla 5 y en los mapas obtenidos, se observa que en el Entorno 3, la función de utilidad basada en la mezcla entre la entropía y el coste ha conseguido crear un mapa completo del entorno recorriendo solamente 15.4 metros y en un periodo de tiempo muy bajo en comparación con las demás funciones.

En cuanto a las funciones basadas en el coste y en la entropía ambas tienen una duración de la ejecución similar, en este caso, como se puede apreciar en las Figuras 5.13 y 5.14 la diferencia se encuentra en la trayectoria seguida, pues cuando el criterio de elección está basado en el coste, la trayectoria sigue de forma ordenada la exploración del mapa sin pasar por una misma zona más de una vez. En cambio cuando el criterio de selección se basa en la entropía se puede apreciar una trayectoria menos ordenada, pues el robot pasa en más de una ocasión por la zona central incrementando la cantidad de arcos de cerrados de bucle. Como se puede observar en la Figura 5.14, la calidad de dicho mapa creado con la función basada en la entropía es superior a la de los demás mapas, pues al aumentar la cantidad de cerrados de bucle, la calidad visual del mapa obtenido mejora, reduciendo así la probabilidad de que el mapa generado presente errores de asociación de datos.

Estos cerrados de bucle son de gran interés pues son generados al pasar por caminos distintos que aun estando dichos caminos alejados en el grafo, se han unido al ser reconocido por el algoritmo de SLAM que ya se había pasado por esa zona con anterioridad.

El mapa que presenta mayor calidad visual es el obtenido tras la aplicación de la función basada en la entropía, pero como se puede apreciar en la Tabla 5, el tiempo empleado en realizar la exploración del entorno es muy elevado. En la mitad del tiempo empleado para la exploración, se obtiene una aproximación bastante buena del entorno mediante la aplicación de la función de utilidad basada en la entropía y el coste.

En cambio, la función basada en la aleatoriedad da unos resultados muy poco satisfactorios, pues como se puede observar en la Tabla 5, el tiempo empleado en la exploración del entorno es el más alto. También se puede apreciar en la Figura 5.12 que la trayectoria no sigue ningún orden concreto, dando lugar a un recorrido que pasa en más de una ocasión por una misma zona, pero mientras que en la función basada en la entropía, la finalidad era conocer mayor cantidad de celdas desconocidas, en esta función de aleatoriedad lo hace sin criterio de selección alguno, dando lugar a una elevada cantidad de nodos que va directamente relacionados con el tiempo empleado en recorrerlos siguiendo la trayectoria.

5.4. Conclusiones

Tras los experimentos realizados se ha observado que cuando el robot recorre un entorno desconocido por primera vez, no es capaz de cerrar ningún bucle y es por eso

por lo que en los mapas mostrados en el capítulo 5 solo se presentan estos cierres de bucle cuando el robot vuelve a pasar por una zona que ya ha pasado y el robot la reconoce. El algoritmo de SLAM aplicado en este trabajo, al no saber exactamente donde se encuentra el robot, puede dar lugar a un aumento de incertidumbre y a problemas a la hora de la creación del mapa. Esto se puede observar en los mapas obtenidos, pues las líneas de las zonas ocupadas no son uniformes, lo cual puede dar lugar durante el estudio del entorno a la aparición de zonas desconocidas a estudiar donde el robot ya ha pasado con anterioridad y ha establecido que esa zona estaba ocupada.

También se ha comprobado que es muy importante a la hora de aplicar SLAM Activo una buena elección de la función de utilidad a ejecutar, pues tras la comparación de los resultados de las Tablas 3, 4 y 5 de los distintos entornos se puede ver la diferencia tanto en tiempo de exploración como en el recorrido realizado que hace el elegir una función de utilidad óptima. Donde se ve que en efecto, la función que se basa en la elección aleatoria de fronteras proporciona los tiempos más altos de exploración junto a una mayor distancia en metros recorrida y a una baja calidad del mapa generado.

En la función basada en el coste, se identifican fronteras muy cercanas por lo que el robot invierte gran parte del tiempo de exploración en estar constantemente calculando la frontera siguiente a estudiar. Tras la aplicación de esta función, también se ha observado una alta cantidad de arcos de cerrado de bucle, aunque, como se puede observar tanto en las Figuras 5.3, 5.8 como 5.13 al ser estos cerrados de bucle generados entre nodos que se encuentran relativamente consecutivos, no resultan de gran interés.

También se ha observado que para entornos como el 2, que presentan cierta complejidad, esta función no da buenos resultados, pues continuamente encuentra fronteras muy cercanas a él y necesita bastante tiempo en salir de la zona en la que se encuentra para ir a explorar una zona nueva más lejana. En cambio para entornos como el 1 y el 3 se obtienen muy buenos resultados al no presentar zonas de la complejidad del Entorno 2.

En cambio, con la función basada en la entropía, se seleccionan fronteras más lejanas, esto se traduce en un mayor tiempo de exploración del entorno y tiempos más espaciados entre el cálculo de una frontera y la siguiente. Al seleccionarse zonas con gran cantidad de celdas desconocidas, la probabilidad de volver a pasar por zonas conocidas y cerrar bucles es muy baja.

Cabe destacar los óptimos resultados obtenidos con la función basada en la mezcla de la entropía y el coste tras los resultados vistos en los tres entornos. Se observan tiempos de exploración reducidos en comparación con los resultados obtenidos con las demás funciones de utilidad y se observa también una menor distancia recorrida. Es por ello por lo que se concluye que esta función de utilidad es la más óptima cuando se quiere explorar un entorno en el menor tiempo posible.

6. Conclusiones

En este Trabajo Fin de Grado se ha estudiado el problema de exploración robótica autónoma. En concreto para el caso en el que el objetivo principal de dicha tarea de exploración es adquirir de la forma más eficaz y precisa posible un mapa de un entorno desconocido sin conocer la posición del robot en dicho entorno, esto es conocido como localización y mapeo simultáneos activos (SLAM Activo). Para ello el robot debe obtener su localización en dicho entorno a la vez que lo recorre y construye el mapa de este entorno.

Este trabajo se centra en la parte del SLAM Activo que estudia la selección de la mejor frontera a estudiar, es decir la siguiente acción del robot a realizar, mediante el análisis de las posibles funciones de utilidad con la finalidad de seleccionar aquella que facilite el recorrido del entorno de la manera más eficaz posible. Para la realización de este trabajo, ha sido necesario adquirir conocimientos sobre la teoría del SLAM, del SLAM Activo y sobre los grafos de pose y su creación. A la vez que se ha visto necesario aprender a trabajar en el entorno de programación de MATLAB y la utilización de la *Robotics System Toolbox* de MATLAB.

Mediante la programación de un algoritmo de SLAM Activo en MATLAB se ha conseguido realizar al completo la ejecución de la tarea de exploración robótica de un entorno desconocido sin conocer la ubicación del robot en él.

Para la implementación de este algoritmo en este Trabajo Fin de Grado, se han creado tres entornos virtuales de simulación en Gazebo siendo necesario adquirir conocimientos sobre la máquina virtual VMware mediante la cual se ha podido acceder al simulador Gazebo. Para la creación de dichos entornos, en un principio, se planteó la idea de crear entornos de grandes dimensiones, pero debido al bajo valor de las velocidades máximas del robot usado en este trabajo y al tiempo que empleaba en recorrerlos, se decidió el uso de entornos de dimensiones más adecuadas para el robot. Los entornos creados presentan zonas a las que el robot puede acceder mediante distintos caminos, a la vez que tiene zonas sin salida, lo cual presenta un desafío y es de gran interés estudiar a la hora de aplicar el algoritmo de exploración robótica.

Una vez creados los entornos, ha sido necesario familiarizarse con los conceptos básicos de ROS y adquirir conocimientos sobre cómo realizar la conexión de ROS y Gazebo con el entorno de MATLAB. Tras esta conexión y estos conocimientos adquiridos, ha sido posible la obtención de un mapa de ocupación a partir del escáner láser del robot del entorno de Gazebo.

A partir de este mapa de ocupación se ha procedido a detectar las fronteras que se encuentran en dicho mapa, es por ello por lo que se ha creado e implementado una función cuya finalidad es la detección de las fronteras y la agrupación de estas. Para

optimizar esta función, se han analizado dos métodos de agrupación de puntos, seleccionando el método cuyo algoritmo presenta repetibilidad y una ligera mayor cantidad de fronteras resultantes, pues se ha comprobado que a medida que se conoce más información del entorno, la cantidad de fronteras detectadas disminuye, y con el método no seleccionado, se ha dado el caso de que no ha sido capaz de seleccionar ninguna frontera estando el mapa del entorno aún sin acabar.

Una vez obtenidas las fronteras, se ha implementado un código que ha estudiado cuatro criterios de selección de las fronteras, con la finalidad de optimizar el proceso de SLAM Activo.

Para dar por terminada la creación del algoritmo de SLAM Activo, se han estudiado e implementado dos métodos de planificación de rutas para analizar cuál era el de mayor utilidad. Se han observado las ventajas y las desventajas de cada uno de ellos y se ha terminado seleccionando el método que mayor fiabilidad ha dado a la hora de encontrar un camino hacia la frontera deseada. Para la consecución de la navegación del robot a través de la ruta creada se han utilizado los controladores *PurePursuit* para seguir la ruta y VFH cuya finalidad ha sido evitar los obstáculos a lo largo del recorrido del robot.

Durante el seguimiento de ruta se han observado ciertos problemas, pues el controlador VFH detectaba los obstáculos cuando estos se encontraban enfrente de él, pero debido a la forma del robot cuyas ruedas se encuentran fuera del radio del cuerpo de este, si al girar en una esquina del entorno una de estas se quedaba atascada, el robot no era capaz de reconocer el obstáculo, y se ha tenido que esperar a que con las medidas de velocidades lineales y angulares dadas por el controlador *PurePursuit* se consiguiese desatascar la rueda o detectar finalmente el obstáculo para proceder a retroceder y a buscar un nuevo camino hacia la frontera de mayor interés. También se han dado complicaciones a lo largo del desarrollo del código que ejecuta el seguimiento de la trayectoria debido a la lenta conexión entre MATLAB y ROS y a la búsqueda de la relación que mejor se ajustase al recorrido del entorno entre la velocidad lineal, la angular y la distancia de *LookAhead.*

Finalmente, tras la ejecución completa del algoritmo de SLAM Activo, se ha realizado un análisis de las diferencias de utilizar cada uno de los distintos criterios a la hora de definir la función de utilidad. Tras los resultados obtenidos de la parte de la experimentación, se ha concluido que la función de utilidad cuyo criterio de selección se ha basado en la combinación entre la entropía y el coste, recorre el entorno desconocido en un tiempo y una distancia mínimos comparados con datos de los tiempos y distancias de las funciones basadas en los demás criterios.

Como trabajo futuro se plantea la experimentación en un entorno con la versión real del robot *turtlebot3*.

Bibliografía

[1] Grisetti, G., Kümmerle, R., Stachniss, C., & Burgard, W. (2010). A tutorial on graph-based SLAM. IEEE Intelligent Transportation Systems Magazine, 2(4), 31-43.

[2] Koubâa, A. (Ed.). (2017). Robot Operating System (ROS) (Vol. 1, pp. 112-156). Cham: Springer.

[3] Marios Xanthidis, C. Stachniss, P. Allen, C. Fermuller Paul Furgale, Margarita Chli, Marco Hutter, Martin Rufli, Davide Scaramuzza, Roland Siegwart. SLAM Tutorial.

[4] Placed, J. A., & Castellanos, J. A. (2020). A Deep Reinforcement Learning Approach for Active SLAM. Applied Sciences, 10(23), 8386.

[5] Lindado, H. D. C. (2014). Active slam: utility functions and applications (Doctoral dissertation, Universidad de Zaragoza).

[6] Stachniss, C. (2009). Robotic mapping and exploration (Vol. 55). Springer.

[7] "ROS". http://www.ros.org/

[8] "Matlab: Robotics System Toolbox". http://mathworks.com/help/robotics/index.html

[9] "Matlab: Robot Operating System (ROS) Support from ROS Toolbox". https://es.mathworks.com/hardware-support/robot-operating-system.html

[10] "ROS Introduction". http://wiki.ros.org/

[11] "ROS topics: ROS tutorial". https://robinrobotic.blogspot.com/2019/07/ros topics-ros-tutorial.html

[12] "Gazebo". http://gazebosim.org/

[13] "ROS Components, TurtleBot3 Burger". https://www.roscomponents.com/es/robots-moviles/214-turtlebot3-burger.html#/cursos-no/turtlebot_3_burger_modelo-burger_intl_

[14] "ROS Toolbox". https://es.mathworks.com/products/ros.html

[15] "Get Started with Gazebo and a Simulated TurtleBot". https://es.mathworks.com/help/ros/ug/get-started-with-gazebo-and-a-simulated-turtlebot.html

[16] "Obstacle Avoidance with TurtleBot and VFH". https://es.mathworks.com/help/nav/ug/obstacle-avoidance-with-turtlebot-and-vfh.html

[17] "LidarSLAM". https://es.mathworks.com/help/nav/ref/lidarslam.html

[18] "occupancyMap" https://es.mathworks.com/help/nav/ref/occupancymap.html

[19] "Planificación de rutas en entornos de diferente complejidad". https://es.mathworks.com/help/robotics/examples/path-planning-in-environments-of-difference-complexity.html

[20] Choset, H. (2015). Robotic motion planning: RRT's.

[21] "Controlador de persecución pura". https://es.mathworks.com/help/robotics/ug/pure-pursuit-controller.html

Tabla de Figuras

Lista de tablas