Universidad de Alcalá
Escuela Politécnica Superior

Máster Universitario en Ingeniería Industrial

Trabajo Fin de Máster

Estudio e implementación del robot AmigoBot en los entornos de programación MatLab-ROS y Robotics-Academy

Autor: Vladislav Kravchenko

Tutores: Manuel Ocaña Miguel y Jose María Cañas Plaza

2022

UNIVERSIDAD DE ALCALÁ

ESCUELA POLITÉCNICA SUPERIOR

Máster Universitario en Ingeniería Industrial

Trabajo Fin de Máster

Estudio e implementación del robot AmigoBot en los entornos de programación MatLab-ROS y Robotics-Academy

Autor: Vladislav Kravchenko

Directores: Manuel Ocaña Miguel y Jose María Cañas Plaza

Tribunal:

Presidente: María Elena López Guillén

Vocal 1º: Noelia Hernández Parra

Vocal 2º: Manuel Ocaña Miguel

Calificación: .....

Fecha: .....

A mi familia, amigos y profesores...

"Estudiar sin deseo arruina la memoria y no retiene nada de lo que absorbe."
Leonardo da Vinci

Agradecimientos

El agradecimiento es la parte principal del hombre de bien

Francisco Gómez de Quevedo Villegas y Santibáñez Cevallos

Este proyecto debe empezar con unas palabras de plena gratitud a todas y cada una de las personas que me han acompañado durante todos estos largos años de estudios en la Universidad de Alcalá.

En primer lugar gratitud a mis padres y familiares más cercanos, que con mucho esfuerzo, compresión, apoyo y empatía han sabido guiarme hasta la finalización de una etapa más en mi vida.

Agradecimientos a mis amigos tanto viejos como nuevos, adquiridos bajo las arcas del conocimiento, que sin ellos, no habrían sido tan entretenidos todos estos años. Les deseo a mis amigos que aún no han acabado sus estudios, un camino sencillo (que no siempre lo es) hasta la conclusión de estos.

Reconocimiento a todos los profesores que han estado formándome durante todos estos largos años de aprendizaje continuo. La gran mayoría de ellos, grandes profesionales de su rama, siempre atentos y comprensivos. Siempre han tenido la ilusión para repetir, una y otra vez, ese conocimiento, que, a veces, cuesta entender a la primera vez.

Y, por último, pero no menos importante, agradecimientos a mis tutores de este trabajo, Manuel Ocaña Miguel y Jose María Cañas Plaza, por su continuo apoyo durante la realización de este TFM.

Resumen

En este Trabajo Fin de Máster (TFM) se aborda, en una primera instancia, la implementación de una serie de algoritmos para aplicaciones robóticas (tales como SLAM, VFH, AMCL, PRM) en el entorno de MatLab y Robotics Operating System (ROS) [1].

A continuación, se presenta un nuevo entorno para aprendizaje robótico denominado Robotics-Academy. En esta plataforma se llevará a cabo la implementación de algunas aplicaciones robóticas que se han mencionado anteriormente, dejando estas aplicaciones a disposición de la comunidad Open Source con licencia GPL [2]. Por último, se realiza una comparación de ambos entornos de programación robótica.

Palabras clave: Robotics-Academy, Python, MatLab, ROS.

Abstract

In this Final Master's Thesis (TFM) is carried out the implementation of a series of algorithms for robotic applications (such as SLAM, VFH, AMCL, PRM), in the first instance. This implementation is carried out in the environment of MatLab and ROS [1].

Next, a new environment for robotic learning called Robotics-Academy is presented. In this platform, the implementation of some robotic applications that have been mentioned above will be carried out, leaving these applications at the disposal of the Open Source community with a GPL license [2]. Finally, a comparison between both platforms is addressed.

Keywords: Robotics-Academy, Python, MatLab, ROS.

Índice general

Índice de figuras

Índice de tablas

Índice de listados de código fuente

Lista de acrónimos

AMCL	Adaptative (KLD-Sampling) Monte Carlo Localization.
ARIA	ActiveMedia Robotics Interface for Application.
CAD	Computer-Aided Design.
CARMEN	Carnegie Mellon Robot Navigation Toolkit.
CVE	Common Vulnerabilities and Exposures.
EOL	End-of-Life.
ERSP	Evolution Robotics Software Platform.
ESO	Educación Secundaria Obligatoria.
GPL	General Public License.
GUI	Graphical User Interface.
HAL	Hardware Abstraction Layer.
HTML	HyperText Markup Language.
JSON	JavaScript Object Notation.
KL	Kullback-Leibler.
LIDAR	Light Detection and Ranging.
LTS	Long Term Support.
LUT	LookUp Table.
MCL	Monte Carlo Localization.
MIT	Massachusetts Institute of Technology.
MRDS	Microsoft Robotics Developer Studio.
PRM	Probabilistic RoadMap Method.
RADI	Robotics-Academy Docker Image.
RAI	Robotics and Artificial Intelligence.
ROS	Robotics Operating System.
SCARA	Selective Compliant Assembly Robot Arm.

SLAM	Simultaneous Localization and Mapping.
SO	Sistema Operativo.
STDR	Simple Two Dimensional Simulator.
TFM	Trabajo Fin de Máster.
UAH	Universidad de Alcalá.
UMH	Universidad Miguel Hernández.
URJC	Universidad Rey Juan Carlos.
URL	Uniform Resource Locator.
USARSim	Unified System for Automation and Robot Simulation.
VFF	Virtual Force Field.
VFH	Vector Field Histogram.
XML	Extensible Markup Language.

Capítulo 1

Introducción

"Es mejor equivocarse siguiendo tu propio camino que tener razón siguiendo el camino de otro."
Fiódor Mijáilovich Dostoyevski

1.1 Historia de la robótica y educación en robótica

La historia de la robótica dio inicio hace muchos años, pero no es hasta el siglo XX donde se desarrollaron los robots modernos. Estos robots modernos cumplían unos objetivos específicos útiles para el desarrollo de la actividad industrial. El primer robot industrial se instaló en una cadena de montaje de General Motors en el año 1961. Este robot se llamaba "Unimate".

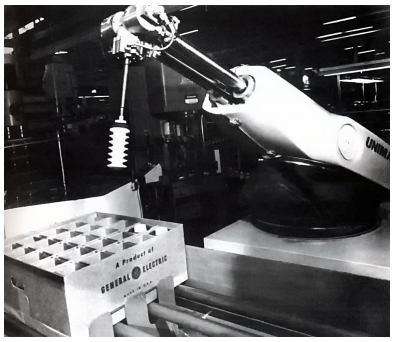


Figura 1.1: Primer robot industrial Unimate

Con el paso del tiempo, los robots se han lanzado cada vez más al público masivo. Ya no son aplicaciones puramente industriales sino que, hoy en día, solventan necesidades de las personas en su vida cotidiana. Ejemplo de esto son las aspiradoras robóticas que resolvieron una necesidad doméstica. También, en la última década, se han desarrollado e implementado los asistentes de conducción autónoma que vienen a resolver la necesidad de automatizar el transporte de personas y de mercancías.

Pero todos estos avances no podrían haberse conseguido sin la formación de profesionales en este sector. La robótica es un campo multidisciplinar donde convergen muchas tecnologías, tales como: la electrónica, mecánica, informática, telecomunicaciones, etc.

Hoy en día, la formación de profesionales en este sector comienza desde una edad temprana de los estudiantes como es en la Educación Secundaria Obligatoria (ESO). Posteriormente, esta formación continua de manera más específica en las universidades mediante grados y postgrados dedicados. Un ejemplo

de esta tendencia educacional desde secundaria es la Comunidad de Madrid, que con el Decreto 48/2015 introdujo la asignatura *Tecnología, programación y robótica* en el currículum oficial de la ESO.

Así, en la enseñanza robótica durante la educación secundaria, se parte de la enseñanza básica del funcionamiento de sensores, actuadores y la programación. Para esto se utilizan plataformas como los robots LEGO, línea de *Lego Mindstorms* (RCX, NXT, EV3, EV4). Se enseñan lenguajes de programación sencillos para niños, tales como RCX-code, Scratch o Blocky [3][4].

En la enseñanza universitaria se imparte la robótica en grados y postgrados como *Electrónica Industrial y Automática*, *Ingeniería Electrónica, Robótica y Mecatrónica*, *Ingeniería Robótica*, *Ingeniería Industrial*, etc., de tal forma que ya se llega a impartir con mayor énfasis en el software, la inteligencia artificial y la visión computacional. De igual forma se imparten estas disciplinas de la robótica en el ámbito internacional, como son las universidades Stanford, Massachusetts Institute of Technology (MIT), Carnegie Mellon University, etc.

El objetivo de todo este proceso educativo es formar profesionales en el sector de la robótica, pero al ser tan multidisciplinar, se puede enfocar desde muchas perspectivas. Uno de los enfoques es el aumento de la formación mediante plataformas de desarrollo de aplicaciones robóticas.

Uno de estos entornos académicos es *Robotics-Academy* (*http://jderobot.github.io/RoboticsAcademy/*), que pone énfasis en el software para robots como portador principal de su inteligencia. Se centra en que los estudiantes programen los algoritmos de percepción, planificación y control habituales en los robots. Esta visión se ha utilizado con éxito en las asignaturas *Robótica* del grado de Ingeniería Telemática y *Visión en robótica* del Máster de Visión Artificial impartidas en la Universidad Rey Juan Carlos (URJC).

Por otro lado, uno de los entornos docentes más empleados es MatLab, con su propio lenguaje. Por ejemplo, en la Universidad Miguel Hernández (UMH) de Elche, se ofrece la toolbox de MatLab ARTE (*A Robotics Toolbox for Education* [5] para visualización 3D y programación de brazos robóticos con lenguaje industrial. También, Aliane [6] utiliza MatLab y Simulink, ofreciendo a los estudiantes prácticas sobre cálculo de trayectorias y control de un manipulador *Selective Compliant Assembly Robot Arm (SCARA).* La Universidad de Zaragoza junto con la *Technical University of Iasi* de Rumanía, desarrollaron una toolbox denominada *Robot Motion Toolbox* [7], permitiendo así la planificación, navegación y control de un robot móvil. Corke desarrolló una toolbox denominada *Robotics Toolbox* [8] que permite realizar la programación de robots móviles, brazos robóticos y robots con visión.

En la Universidad de Alcalá (UAH), en el Máster de Ingeniería Industrial con la especialidad de Robótica y Percepción, en la asignatura Robótica Móvil, se utiliza el entorno de MatLab junto con ***ROS Toolbox***. Aquí se utiliza un robot móvil para probar en él diferentes tipos de aplicaciones tanto de mapeado (SLAM), localización (AMCL), evitación de obstáculos (VFH) y planificación (PRM). Estas aplicaciones se parametrizan de diferente manera obteniendo una visión de los diferentes resultados que se pueden obtener.

En este trabajo, se abordarán dos plataformas para aprendizaje robótico: MatLab con ROS Toolbox y Robotics-Academy. Se desarrollarán una serie de ejercicios en las dos plataformas para obtener, por un lado, la implementación de estos ejercicios para su utilización por el entorno académico, y por otro lado, una comparación de los dos diferentes entornos de enseñanza robótica.

1.2 Plataformas o entornos de desarrollo

Las plataformas o entornos de desarrollo son herramientas que se ubican entre el *framework*[1] y la aplicación de control desarrollada por el usuario. Los *framework* poseen diversos repositorios en los que se guardan diferentes drivers, herramientas, bibliotecas y software reutilizable de los diferentes fabricantes para su fácil utilización. Con este framework ya no se tienen que preocupar de desarrollar la comunicación entre los sensores y actuadores, solamente tienen que utilizar estos recursos existentes y centrarse en el desarrollo de la aplicación. Uno de los *framework* conocidos es ROS, pero en la misma categoría ha habido otros:

- Microsoft Robotics Developer Studio (MRDS) [9].
- Carnegie Mellon Robot Navigation Toolkit (CARMEN) [10].
- Evolution Robotics Software Platform (ERSP) [11].
- ORCA.
- Beesoft.
- Robotics and Artificial Intelligence (RAI).
- ActiveMedia Robotics Interface for Application (ARIA).

Las plataformas de desarrollo facilitan el acceso a los diferentes drivers del sistema gracias a la capa de abstracción de hardware (HAL). Con esta capa adicional, se facilita al usuario la adquisición y depuración de los datos sensoriales que proporciona el hardware del robot. El usuario únicamente tendrá que hacer uso de estos datos en su aplicación. Además, estas plataformas también proporcionan una interfaz gráfica atractiva y sencilla de utilizar. Algunas de las plataformas de ejemplo son:

- RobUALab [12].
- Mobile Robot Interactive Tool [13].
- Robots Formation Control Platform [14].
- Tekkotsu [15].
- Entorno docente con simulador Webots [16].
- TRS [17].

Se ha comentado anteriormente que estas plataformas se ubican por encima de un *framework* que maneja los drivers del sistema. En nuestro caso, este *framework* va a ser ROS en las dos plataformas de aprendizaje que se van a ver durante el desarrollo de este trabajo fin de máster.

[1] Conjunto estandarizado de conceptos, prácticas y criterios para enfocar un tipo de problemática particular que sirve como referencia, para enfrentar y resolver nuevos problemas de índole similar. Wikipedia

1.2.1 Introducción a ROS

En los apartados anteriores ya se ha hablado de ROS pero, ¿qué es esta herramienta? ROS es un *framework* para la escritura de software de robots. Este *framework* posee todas las herramientas, bibliotecas y convenciones necesarias para simplificar la labor de la creación de aplicaciones robóticas complejas y robustas.

Figura 1.2: Ilustración del logotipo de ROS

La historia de ROS empieza a mediados de los 2000 con los esfuerzos de la Universidad de Stanford por crear prototipos internos de sistemas de software flexibles y dinámicos destinados al uso de la robótica. Fue en 2007, cuando Willow Garage proporcionó recursos significativos para extender estos conceptos y crear implementaciones bien probadas. Junto con Willow Garage multitud de investigadores contribuyeron con su tiempo y experiencia para desarrollar tanto las ideas centrales de ROS como sus paquetes de software fundamentales. Gradualmente, con el paso del tiempo, se ha convertido en una plataforma ampliamente utilizada en la comunidad de investigación robótica que trabaja en proyectos que van desde proyectos de pasatiempos de mesa hasta grandes sistemas de automatización industrial. En 2012 se lanzó el proyecto de código abierto ROS-Industrial, que desarrolla las grandes capacidades de ROS para la automatización industrial.

ROS se basa en un sistema de grafos en el que intervienen muchos conceptos diferentes que es necesarios saber. En primer lugar, siempre existe un nodo principal llamado ***roscore*** o ***máster***. Este nodo principal se encarga de la coordinación del resto de nodos que están perfectamente distribuidos, permitiendo su procesamiento en múltiples núcleos, GPUs y clusteres. En ROS se trabaja mediante publicación o subscripción de flujo de datos que van desde imágenes, pasando por estéreo, láser, control hasta actuadores, etc. Otros conceptos que es necesario saber de ROS son:

- **Nodos**: programas ejecutables. Estos nodos facilitan el diseño modular gracias a que cada nodo se compila, ejecuta y maneja individualmente. Se escriben utilizando una librería cliente de ROS (*roscpp* en C++ o *rospy* en Python). Los nodos pueden suscribirse o publicar en *topics* y pueden proporcionar o utilizar *Servicios*.
- **Topics**: Canales de comunicación utilizados por los nodos para comunicarse entre ellos. Cada *topic* transporta un único tipo de *Mensaje*.

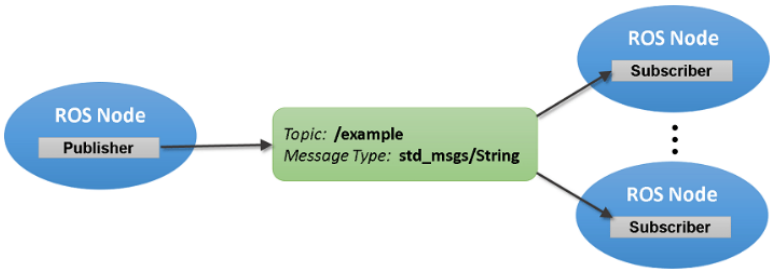


Figura 1.3: Relación entre diferentes conceptos de ROS.

- **Mensajes**: Tipo estructurado de datos para la comunicación entre nodos. En cada *topic*, como

ya se ha comentado, solo se puede transmitir un tipo de mensaje (láser, odometría, etc.). Pueden definirse nuevos mensajes propios.

- **Servicios**: Modelo de comunicación síncrona entre nodos de tipo cliente/servidor. Se utilizan habitualmente para eventos especiales o que necesiten confirmación. Se puede obtener información de los *Servicios* mediante el comando *rosservice*.
- **Rosbag**: es un conjunto de herramientas que permiten almacenar y reproducir datos de una ejecución del sistema.
- **tf**: *topic* y mensaje especial de ROS que sirve para relacionar distintos marcos de coordenadas. Por regla general, siempre está presente en el sistema.

A modo de resumen, en el sistema de ROS cada programa en ejecución se considera como un *nodo*. Existe un nodo *máster* que coordina la ejecución del sistema. La comunicación entre nodos se realiza mediante *topics*, si es una comunicación continua, o, mediante *servicios*, si es una comunicación llamada-respuesta. El topic *tf* es un topic especial encargado de relacionar los distintos marcos de coordenadas del sistema.

1.2.2 Introducción a Robotics-Academy

Robotics-Academy[18] es un entorno docente de robótica universitaria. Este entorno tiene una orientación muy práctica en cuanto al aprendizaje de la programación de la inteligencia de los robots. La plataforma posee una gran colección de ejercicios muy variados que abarcan muchas de las aplicaciones robóticas que han surgido recientemente: drones, robots aspiradores, coches autónomos, asistente de aparcamiento, control de robots móviles, etc.

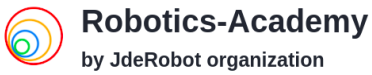


Figura 1.4: Logo de la plataforma Robotics-Academy

Toda esta colección de ejercicios son de código abierto lo que proporciona la posibilidad de compartir, modificar y estudiar el código fuente, además de colaborar entre usuarios.

Robotics-Academy[2] es un entorno multiplataforma pudiendo implementarse en sistemas operativos tales como Linux, Windows y MacOS. La plataforma hace uso del simulador Gazebo y el lenguaje principal con el que se programa es el lenguaje interpretado y multiplataforma Python [19].

1.2.3 Introducción a MatLab - ROS

MatLab [20] es una herramienta para la computación numérica que tiene un uso generalizado por los diferentes campos de la ingeniería, matemáticas, física, etc. Este software, existente para una gran cantidad de plataformas, ofrece una gran facilidad y modularidad en la programación.

Por otro lado, ROS, como se ha visto en la sección 1.2.1, es una plataforma, open source, que proporciona un conjunto de herramientas para el desarrollo de la programación de software para robótica. Este conjunto de herramientas facilita un mecanismo de comunicación entre programas tanto en un mismo ordenador como en varios.

[2] http://jderobot.github.io/RoboticsAcademy/

Ahora, la unión de estas dos potentes herramientas concede un amplio abanico de posibilidades. Esta unión se ha conseguido gracias a que MatLab ofrece la posibilidad de integrar las así denominadas *Toolbox*. Hasta la versión R2019a en MatLab se ofrecía la ***Robotics System Toolbox***, a partir de esta versión esta *Toolbox* pasó a denominarse ***ROS Toolbox***. Esta *Toolbox* facilita una interfaz que conecta MatLab y Simulink con ROS, permitiendo la creación de una red de nodos ROS.

ROS Toolbox proporciona funciones de MatLab y bloques de Simulink para conectar con redes ROS, accediendo a los mensajes generados.

1.3 Introducción al robot AmigoBot

Durante el desarrollo de este trabajo se va trabajar con un robot llamado AmigoBot. Este es un robot móvil con dos ruedas traseras que son las ruedas tractoras y otra rueda que sirve de soporte. El robot posee un sistema de tracción diferencial, es capaz de desarrollar una velocidad lineal de 0.75 metros por segundo (m/s) y una velocidad angular de 0.75 radián por segundo (rad/s).

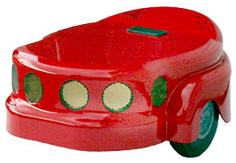


Figura 1.5: Ilustración del robot AmigoBot.

Los sensores que posee este robot son los siguientes:

- Un sensor láser RPLIDAR-A2 que tiene las siguientes características:
  - Número de rayos: 400.
  - Máxima distancia medible: 8.0 metros.
  - Mínima distancia medible: 0.15 metros.
  - Ángulo: 360 grados
- Ocho sensores de ultrasonidos (sonar) con las siguientes características:
  - Máxima distancia medible: 5.0 metros.
  - Mínima distancia medible: 0.1 metros.
  - Ángulo medible: 15 grados.

Lleva incorporado un microcontrolador con un sistema operativo propio llamado ARCOS, que es el encargado de procesar la información de los sensores del sonar y de la odometría. Este microcontrolador, además, gestiona la comunicación con un procesador externo a través de un puerto serie. Este procesador externo es una tarjeta Raspberry PI con el sistema operativo Linux instalado, en el que está incorporado el *framework* ROS. En la Figura 1.6 se puede ver el diagrama de funcionamiento del robot con sus bloques y el sistema software que interviene en su funcionamiento.

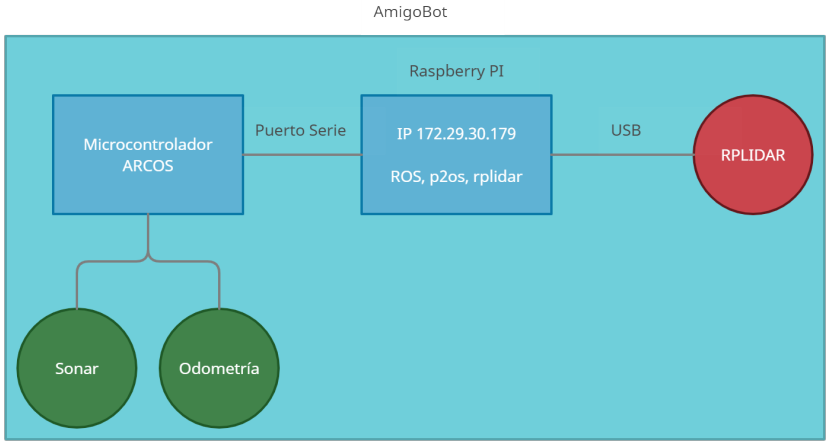


Figura 1.6: Diagrama de funcionamiento del robot AmigoBot.

En la tarjeta Raspberry PI también viene instalado el driver *p2os* que gestiona la comunicación con el AmigoBot. Además, viene instalado otro driver denominado *rplidar* que se utiliza para la lectura de los datos del láser RPLIDAR que viene instalado en el robot. Esta tarjeta viene configurada para tener una IP fija determinada (indicada en la propia carcasa del robot) para que a través de la red WiFi se pueda conectar un ordenador externo.

1.4 Simuladores

Los **simuladores** son herramientas que ofrecen un entorno virtual en el que se emulan las observaciones de los sensores y los efectos de las órdenes a los actuadores. Son herramientas bastante económicas o prácticamente gratuitas en comparación con la adquisición de robots reales.

Gracias a los simuladores es relativamente fácil obtener diferentes tipos de datos del robot, como las lecturas de la posición del robot, lecturas de los sensores, etc. Además, en comparación con los robots reales, los simuladores permiten la repetitividad de los experimentos o prácticas, debido a que en un entorno simulado no cambian las condiciones del entorno, tales como el estado del hardware del robot, condiciones ambientales, iluminación, etc.

Otras ventajas de los simuladores son:

- Posibilidad de diseñar diferentes modelos, probarlos y optimizarlos.
- Conocer el comportamiento de los modelos antes de su puesta en marcha.
- Anticipar errores que se pueden producir en robots reales.

Por estos motivos el uso de los simuladores en docencia de robótica móvil [21] es muy ventajoso para los estudiantes brindando herramientas muy potentes.

Durante el desarrollo de este trabajo se utilizará principalmente el simulador Simple Two Dimensional Simulator (STDR) y se mencionará la existencia del simulador Gazebo en la plataforma Robotics-Academy.

El simulador STDR siendo un simulador muy simple, ofrece la posibilidad de visualizar el comportamiento del robot en dos dimensiones. Este simulador implementa un arquitectura distribuida basada en

el modelo de diseño de software cliente - servidor, permitiendo que cada nodo pueda ser ejecutado en una máquina diferente y comunicarse mediante el uso de las interfaces ROS. Proporciona una interfaz gráfica de usuario (GUI) simple y atractiva que se puede ver en la figura 1.7.

Mediante el uso de este simulador se pretende simular uno o más robots de manera simplificada y minimizar las acciones necesarias que un usuario deba ejecutar para iniciar la experiencia en este simulador. Por otra parte, STDR puede funcionar de dos maneras diferentes: con o sin GUI. El GUI no es estrictamente necesario para que funcione el simulador, pudiendo ejecutar sus funciones mediante la línea de comandos de Ubuntu.

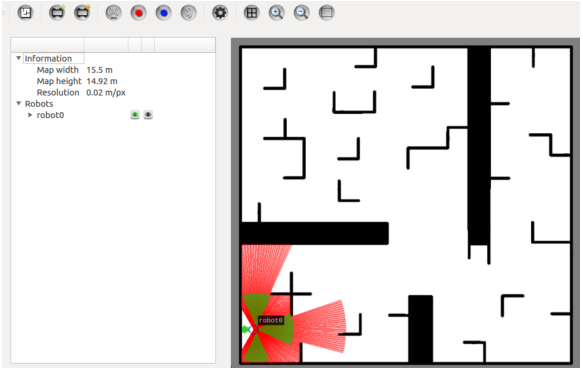


Figura 1.7: Interfaz gráfica de usuario (GUI) del simulador STDR.

Por otro lado, en Robotics-Academy, se trabaja con el simulador Gazebo [22], que es capaz de simular uno o varios robots en un entorno 3D, presentando una elevada capacidad de interacción dinámica entre objetos.

Este simulador permite realizar diseños de robots de forma personalizada, crear mundos virtuales usando herramientas Computer-Aided Design (CAD) e importar modelos ya creados. También permite simular de forma precisa y eficiente grupos de robots en entornos internos y externos complejos, con un motor de físicas consistente y gráficos de alta calidad.

Al igual que el simulador STDR, Gazebo tiene la posibilidad de sincronizarlo con ROS para publicar información de los sensores en sus nodos, así como implementar una lógica y un control que dé ordenes a los robots. También posee un GUI potente y atractivo que se puede ver en la Figura 1.8, con mayores posibilidades que el simulador STDR.

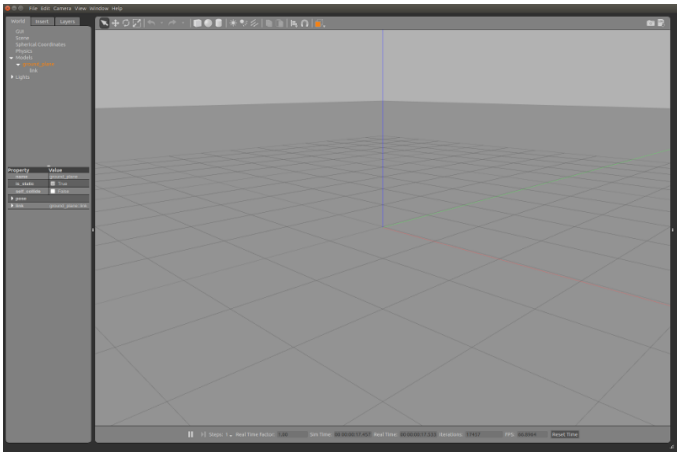


Figura 1.8: Interfaz gráfica de usuario (GUI) del simulador Gazebo.

Otros simuladores que se pueden encontrar y son ampliamente utilizados en simulación robótica son

los siguientes:

- Webots [23][24].
- Unified System for Automation and Robot Simulation (USARSim) [25].
- Player/Stage [26][27].

1.5 Motivación y Objetivos

Este trabajo surge de la necesidad de renovación de la forma en la que se imparte hoy en día la asignatura Robótica Móvil de la UAH. Hasta ahora, la forma de enseñar a los alumnos de la asignatura era mediante la plataforma de aprendizaje MatLab-ROS. La estructura en la que se basa esta plataforma se ha ido quedando obsoleta ya que por un lado se utiliza la versión de Ubuntu 16.04 Long Term Support (LTS) que ya ha llegado al final de su ventana de soporte de 5 años (End-of-Life (EOL)) el día 30 de abril de 2021.

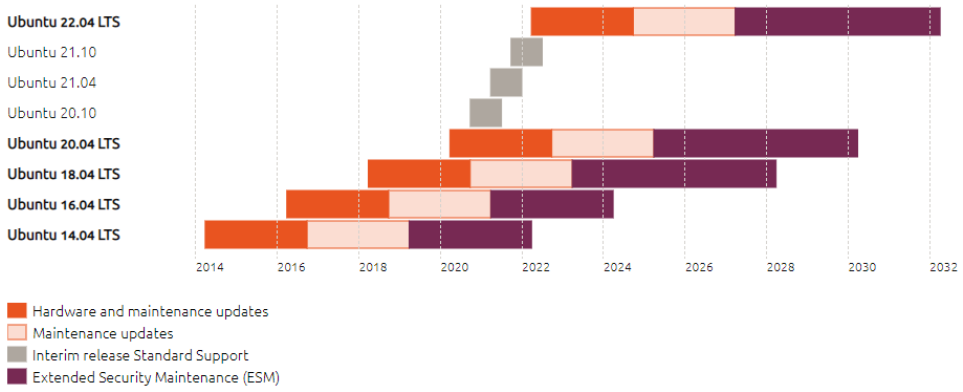


Figura 1.9: Gráfico temporal de soporte a largo plazo de las diferentes versión de Ubuntu.

Para ejecutar esta versión de Ubuntu se utiliza una máquina virtual, en la que ya vienen instalados todos los recursos necesarios para empezar a trabajar con los ejercicios que se proponen en la asignatura de Robótica Móvil. Se debe tener en cuenta también la desventaja que esto provoca debido al recorte de recursos del sistema que supone la utilización de una máquina virtual.

Por otro lado, la versión de MatLab con la que se ha ido trabajando también se ha ido quedando obsoleta, ya que como se ha comentado en la sección 1.2.3, la versión de la *Toolbox Robotics System Toolbox* ha sido renovada desde la versión R2019a de MatLab, con lo que algunas funcionalidades han sido cambiadas o renovadas, lo que podría provocar incompatibilidades en el funcionamiento. Por este motivo se sigue trabajando con versiones de MatLab anteriores a esta versión.

Ubuntu es la plataforma principal para el *framework* ROS desde sus inicios. Cada lanzamiento de una versión de ROS es soportada por una versión LTS de Ubuntu. Debido a que Ubuntu 16.04 (Xenial Xerus) ha llegado al final de su vida útil (EOL), la versión de ROS correspondiente también lo ha hecho (Kinetic). Esto significa el fin de las actualizaciones de seguridad y las correcciones de vulnerabilidades y exposiciones comunes (Common Vulnerabilities and Exposures (CVE)). Todas estas herramientas obsoletas podrían provocar diferentes tipos de riesgos a los usuarios.

Viendo este panorama, se planteó este trabajo que propone una migración de una plataforma de aprendizaje robótico a otra. En esta nueva plataforma, que será Robotics-Academy, se utilizará el nave-

gador web para realizar los diferentes ejercicios propuestos. Se dejará de utilizar las herramientas tales como la máquina virtual y MatLab, y, por otro lado, se renovará la versión de ROS utilizada.

Se tendrá en la plataforma de Robotics-Academy un entorno en la página web en la que confluirán tanto la simulación del sistema como el control y la programación del robot AmigoBot. Se pasará de tener un gran número de aplicaciones ejecutándose para un único fin, a tener todo el sistema recogido en una simple y atractiva página web.

El objetivo fundamental de este proyecto es la utilización académica del robot AmigoBot en los entornos de programación MatLab-ROS y Robotics-Academy. Se realizará un estudio detallado de estos entornos y se abordará la puesta en marcha del robot en los simuladores de estas dos plataformas de desarrollo.

Para llevar a cabo esta tarea en el entorno Robotics-Academy, se debe realizar el desarrollo de los drivers para la conexión del robot real AmigoBot. Estos drivers se pondrán a disposición de la comunidad open source con licencia General Public License (GPL) [2]. Una vez configurada la plataforma para el trabajo con el robot AmigoBot, se desarrollarán unos ejercicios docentes prácticos para demostrar el correcto funcionamiento del trabajo realizado.

Por último, se realizará una comparativa de los dos entornos de desarrollo, buscando las ventajas y desventajas de cada uno. Se indicarán las características más destacadas de cada uno de los entornos y para finalizar se recapitularán las conclusiones alcanzadas con las ideas más relevantes que se han obtenido a lo largo del desarrollo de este trabajo fin de máster.

Capítulo 2

Filosofía de trabajo académico

"El pensamiento no es otra cosa que un simple soplo. Pero un soplo que hace estremecer al mundo."
Victor Marie Hugo

En este capítulo se introducirá al lector en el ámbito de la filosofía de enseñanza tanto de la UAH como de la URJC, en sus respectivas plataformas de enseñanza de robótica.

2.1 Docencia de robótica en la Universidad de Alcalá

Se empieza, en primer lugar, a explicar la filosofía de la UAH, qué es lo que supone trabajar en la plataforma que combina herramientas tan potentes como son MatLab y ROS. En la UAH, en la asignatura de Robótica Móvil, hoy en día, la forma elegida de enseñar robótica móvil consiste en conocer y saber trabajar con diferentes sistemas operativos y herramientas software.

En la asignatura se recomienda la instalación de una máquina virtual en la que se instalará una imagen del Sistema Operativo (SO) Ubuntu 16.04 (como se ha visto en el capítulo 1.5) proporcionada por el profesorado de la asignatura, en la que ya se han preocupado de instalar todo lo necesario para el correcto funcionamiento de todas las herramientas que se verán en la asignatura. Una vez instalada la maquina virtual junto con la imagen del SO Ubuntu, el profesorado recomienda desactivar las actualizaciones del sistema. Esto se realiza para tener el sistema de la forma original sin tener posibles problemas de incompatibilidades que se puedan producir a la hora de actualizar.

Por lo general, se recomienda trabajar con la máquina virtual sobre el SO anfitrión que puede ser tanto Windows, como Linux y MacOS.

En la imagen del SO Ubuntu que proporciona la asignatura, viene preinstalado el *framework* ROS (explicado en la sección 1.2.1). Así, para empezar a trabajar el alumno únicamente tendrá que realizar la configuración del espacio de trabajo catkin (se verá en la sección 3.1). Un espacio de trabajo de catkin es una carpeta donde se modifican, crean e instalan paquetes de catkin. ¿Pero que es **catkin**? **catkin** es el sistema de construcción oficial de ROS y el sucesor del sistema de construcción original de ROS, *rosbuild*. Catkin combina macros CMake y scripts de Python para proporcionar alguna funcionalidad además del flujo de trabajo normal de CMake. Catkin fue diseñado para ser más convencional que *rosbuild*, lo que permite una mejor distribución de paquetes, un mejor soporte de compilación cruzada y una mejor portabilidad. El flujo de trabajo de catkin es muy similar al de CMake, pero agrega soporte para la infraestructura automática de "encontrar paquetes" y la construcción de múltiples proyectos dependientes al mismo tiempo.

Una de las herramientas con las que se trabajará (y se compilará con catkin) en gran medida es el simulador STDR. Este simulador ya se ha introducido en el apartado 1.4, por lo que aquí queda decir el motivo de su utilización: este simulador se utiliza en la asignatura por su gran facilidad de uso, su sencillez gráfica y computacional y por su sencillez a la hora de introducir nuevas configuraciones de robots con sus respectivos sensores. La parte negativa de este simulador es que su desarrollo se abandonó en la versión *Kinetic Kame* de ROS (en el momento del desarrollo de este documento la última versión de ROS es *Noetic Ninjemys*, tres versiones más adelante).

Además, por ser un simulador sencillo no presenta la posibilidad de introducir ruido a los sistemas del robot. La simulación se realiza de forma ideal sin presentar los inevitables ruidos que siempre están presentes en los sistemas reales. Con vistas a esta situación, el profesorado de la asignatura ha desarrollado y proporciona a los alumnos el paquete *aux_files*. Este paquete es el encargado de la introducción de ruido en los sistemas del robot. Se explicará en mayor detalle su instalación y configuración en el apartado 3.1. De este modo se consigue la aproximación del simulador a una situación más parecida a la presente en los sistemas reales.

En la asignatura de Robótica Móvil otra de las herramientas software más utilizadas es MatLab. Esta herramienta es muy conocida mundialmente por sus características computacionales en muchísimos ámbitos de la ingeniería, y otras ciencias. Como ya se ha comentado (sección 1.2.3) MatLab posee una toolbox que una vez instalada ofrece la posibilidad de trabajar con muchos algoritmos de robótica móvil y concede la capacidad de trabajar con ROS en MatLab, utilizar sus herramientas, realizar la conexión con el Máster y la conexión con los topics de ROS, permite saber el árbol de transformadas, etc. En esta toolbox vienen implementados muchos algoritmos que se enseñan en la asignatura Robótica Móvil de la UAH.

En general, en la plataforma de aprendizaje de la UAH, se utilizan muchos lenguajes diferentes de programación. Por ejemplo, se utiliza C++, el lenguaje de programación de MatLab y el lenguaje Extensible Markup Language (XML). Por lo que se necesita un cierto nivel de conocimiento de programación para abordar el aprendizaje directo de las aplicaciones robóticas.

El esquema de la configuración del sistema que sigue la UAH para enseñar robótica móvil es el que se puede ver en la siguiente Figura 2.1.

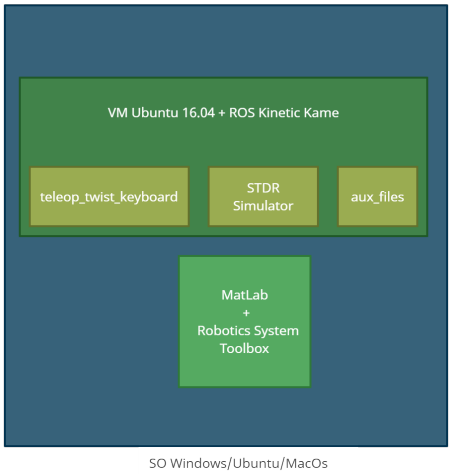


Figura 2.1: Esquema de configuración del sistema MatLab-ROS.

En esta figura 2.1 se puede ver un paquete adicional que no se ha mencionado aún. Este es el paquete

teleop_twist_keyboard encargado de teleoperar el robot AmigoBot tanto por el entorno simulado como en el entorno real.

2.2 Docencia de robótica en la Universidad Rey Juan Carlos

Por el otro lado, se tiene la filosofía de trabajo de la URJC. En esta universidad se plantearon el desarrollo de la plataforma Robotics-Academy en la que se aborda el aprendizaje robótico de forma directa con ejercicios específicos, en los que los alumnos deberán programar la solución en el lenguaje de programación Python.

Como se ha introducido en la sección 1.2.2, la plataforma posee una gran colección de ejercicios muy variados. Todos los ejercicios proporcionados están alojados de forma *open-source* en el portal GitHub[28], que es un sistema de gestión con el que se pueden administrar las diferentes versiones de los proyectos. La página en cuestión es RoboticsAcademy (*https://github.com/JdeRobot/RoboticsAcademy*).

Cada uno de los ejercicios presenta en sí una forma independiente de trabajo, poseen su propia teoría que sirve al alumno para abordar el ejercicio. Además, cada ejercicio tiene sus propios vídeos ilustrativos de funcionamiento del ejercicio una vez solucionada su programación. De esta forma, el alumno sabe cómo debe funcionar el ejercicio una vez haya realizado la solución.

Para que el usuario sea capaz de trabajar con los ejercicios de esta plataforma, la organización Robotics-Academy pone a disposición del usuario un contenedor Docker llamado **RADI**. De esta forma, el usuario puede trabajar con los ejercicios sin tener que instalar prácticamente nada. ¿Cómo es posible esto? Esto es posible gracias a que en este contenedor Docker los desarrolladores ya se han preocupado de instalar todas dependencias necesarias para que la ejecución de los diferentes ejercicios se realice de forma correcta. Además, gracias a que Docker es una tecnología multiplataforma, los ejercicios se pueden ejecutar en cualquier sistema operativo, incluidos Windows, MacOS y Linux.

En esta imagen Docker viene instalado el SO Ubuntu y, además, también viene preinstalado el *framework* ROS. Por lo que en este sentido las dos plataformas, tanto MatLab-ROS como Robotics-Academy, son parecidas.

Además, la plataforma Robotics-Academy posee un foro de discusión en el que los usuarios pueden compartir sus experiencias con los ejercicios, resolver todo tipo de dudas relacionadas o comentar cualquier tipo de problema o bug que pueda existir en cualquiera de los ejercicios o en la plataforma en general. Este foro se puede encontrar en la siguiente Uniform Resource Locator (URL): *https://forum.jderobot.org/*.

Esta plataforma ha optado por la utilización del simulador Gazebo (introducido en el apartado 1.4) para la simulación de los diferentes ejercicios de la plataforma. Un simulador bastante más completo en comparación con el simulador STDR.

Para la ejecución de los ejercicios robóticos la plataforma proporciona una página web en el browser para cada ejercicio. De este modo, tanto la programación de la solución como la visualización de esa solución se realiza desde un mismo sitio: el browser. De tal forma, una ventaja de esta plataforma es la forma sencilla para el alumno de ponerse a trabajar en un determinado ejercicio sin ejecutar demasiados comandos en la terminal. Todas las herramientas se inicializan en segundo plano sin que el alumno tenga que realizar ninguna configuración.

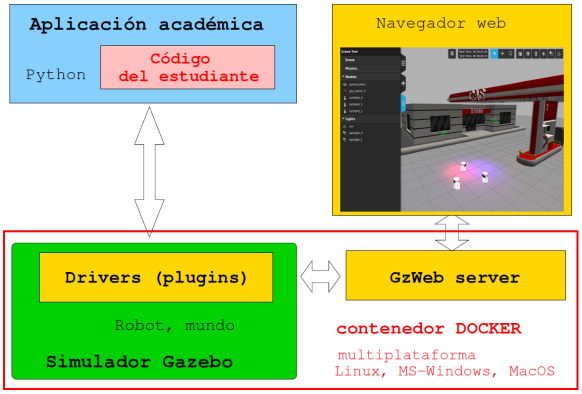


Figura 2.2: Esquema de configuración del sistema Robotics-Academy.

2.3 Conclusión

Volviendo a pensar en lo que supone trabajar en cada una de las plataformas se puede decir lo siguiente:

- La plataforma de UAH supone la integración de muchas herramientas para un único fin común: la solución de una determinada aplicación robótica. En esta plataforma no se hace hincapié en el estudio de los diferentes lenguajes de programación (aunque se utilizan muchos diferentes) sino en la utilización de diferentes herramientas y su coordinación. Pero el objetivo principal sigue siendo la enseñanza de diferentes técnicas robóticas. La mayoría de los algoritmos para estas técnicas robóticas vienen desarrollados a falta de completar una serie de líneas de código para que el algoritmo empiece a trabajar. De esta forma el alumno se centra en dos cosas: en primer lugar, la comprensión de los algoritmos robóticos y su funcionamiento, y, en segundo lugar la comprensión de la infraestructura implicada.

- La plataforma Robotics-Academy supone la simplificación en cuanto a la utilización de diferentes programas y se centra también en enseñar al alumno los algoritmos y aplicaciones robóticas. En comparación con la plataforma de la UAH, en la plataforma Robotics-Academy se debe tener un conocimiento básico o avanzado del lenguaje Python, debido a que si el alumno aborda un ejercicio, este se le presenta completamente en blanco. El alumno es el que programa desde cero la solución al ejercicio en lenguaje Python.

Capítulo 3

Desarrollo e Implementación en MatLab-ROS

"Si se estudia un problema con orden y método, no hay dificultad alguna en resolverlo (Hercule Poirot)."
Agatha Christie

En esta sección se va a mostrar el proceso que se realiza para la implementación del control del robot tanto real como simulado en el entorno MatLab-ROS. Se abordarán una serie de prácticas clásicas de robótica móvil para las cuales se propondrá una solución concreta. Para trabajar con este entorno de desarrollo se debe realizar la puesta en marcha del sistema, la cuál viene descrita en la sección 3.1.

En una de las prácticas se aborda la capacidad de navegación del robot, lo cual es un aspecto esencial de un robot móvil. Los sistemas de navegación de los robots se centran en que el robot se mueva por el entorno sin colisionar con los obstáculos que detecte gracias a los sistemas sensoriales que tenga incorporados. En la Figura 3.1 se representan los bloques de los que se compone un sistema de navegación.

Se empieza con un mapa del entorno y un punto destino. Con esto se realiza la planificación de la trayectoria, la cual crea una secuencia de objetivos que deben ser alcanzados por el robot. Esta secuencia se obtiene con el mapa del entorno, el punto destino y algún tipo de algoritmo de planificación. El seguidor de trayectorias genera los comandos de direccionamiento y velocidad que actuarán sobre los motores del robot móvil. Utilizando los sensores internos del robot junto con técnicas odométricas, se realiza una estimación de la posición del robot. Esta posición realimentará al seguidor de trayectorias.

Después de la navegación se aborda el mapeado que se resolverá mediante dos técnicas diferentes: mediante mapeado con posiciones conocidas y mediante la técnica de *Simultaneous Localization and Mapping (SLAM)*. Una vez mapeado el entorno, se abordará la localización del robot en ese mapa obtenido anteriormente. La localización del robot se resolverá mediante el algoritmo *Adaptative (KLD-Sampling) Monte Carlo Localization (AMCL)*.

Por último, en cuanto a la planificación de la navegación tanto local como global, se abordará mediante los algoritmos *Vector Field Histogram (VFH)* y *Probabilistic RoadMap Method (PRM)*, respectivamente.

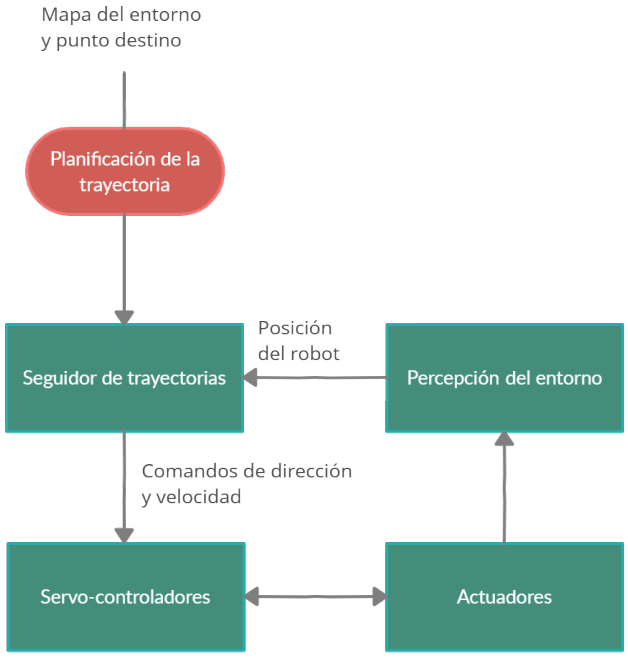


Figura 3.1: Diagrama de un sistema de navegación básico para un robot móvil.

3.1 Introducción al desarrollo de aplicaciones robóticas en el entorno MatLab-ROS

Para poder trabajar en este entorno académico y realizar las prácticas que se van a proponer más adelante, en primer lugar se debe realizar el proceso de configuración del sistema para la utilización de ROS. Hay dos formas de configurar el sistema para trabajar con ROS. La primera consiste en la instalación de una máquina virtual en la que ya se encuentra una preinstalación de ROS Kinetic y los paquetes más importantes a utilizar en los ejercicios. La segunda opción es realizar la instalación de ROS Kinetic sobre Ubuntu 16.04 en modo nativo (véase anexo A.1). En este trabajo se ha elegido la primera opción, por lo que se ha instalado el software de virtualización *VirtualBox* y en él se ha cargado la imagen del sistema operativo.

El siguiente paso es realizar la configuración del entorno para poder generar código. Es necesaria la creación de un espacio de trabajo donde se guardarán y compilarán los archivos de los diferentes paquetes creados. Para proceder con la configuración del entorno se debe abrir la terminal de Ubuntu y ejecutar los siguientes comandos:

Listing 3.1: Configuración del espacio de trabajo

```
1 // Se crea el directorio en la carpeta personal del usuario
2 mkdir -p ~/catkin_ws/src
3 cd ~/catkin_ws/src/
4 // Se inicia el espacio de trabajo catkin
5 catkin_init_workspace
6 cd ~/catkin_ws
7 catkin_make
```

Se debe añadir el espacio de trabajo al *path* por defecto, por lo que se edita el archivo /.bashrc de Ubuntu, encargado de ejecutarse cada vez que el usuario ejecuta el programa bash.

Listing 3.2: Añadir el espacio de trabajo al path por defecto

```
sudo gedit ~/.bashrc
source ~/catkin_ws/devel/setup.bash
export ROS_PACKAGE_PATH=$ROS_PACKAGE_PATH:~/catkin_ws/
```

Una vez configurado el espacio de trabajo se procede a la instalación del simulador STDR. Para instalarlo se debe descargar el código fuente de la siguiente página de GitHub (*https://github.com/stdr-simulator-ros-pkg/stdr_simulator*). Una vez descargado el archivo se debe descomprimir en el espacio de trabajo anteriormente creado (catkin_ws/src):

Listing 3.3: Construcción del espacio de trabajo catkin

```
cd ~/catkin_ws
catkin_make
```

Este último comando puede manifestar la falta de algunas dependencias, tales como: *map_server* y *qt-4*. Para instalar estas dependencias se deben ejecutar los siguientes comandos:

Listing 3.4: Instalación de dependencias

```
sudo apt-get install ros-melodic-map-server
sudo apt-get install qt4-default
```

Con esto, se debe ejecutar de nuevo los comandos del Listado 3.5. Para asegurarnos que el proceso de instalación del simulador ha sido satisfactorio, se ejecuta el siguiente comando en la terminal de Ubuntu:

Listing 3.5: Comando de ejecución del simulador STDR

```
roslaunch stdr_launchers server_with_map_and_gui_plus_robot.launch
```

De esta forma, si se ha instalado todo correctamente, deberá de salir una ventana del simulador de la siguiente forma vista en la Figura 3.2:

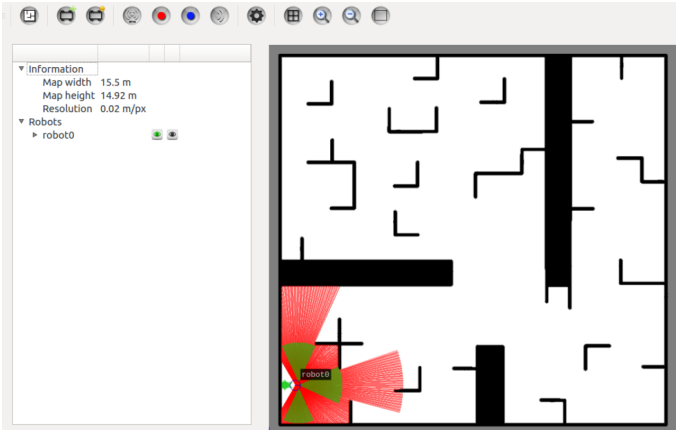


Figura 3.2: Ventana de la primera ejecución del simulador STDR.

Así, de este modo, ya tenemos tanto el entorno ROS instalado como el simulador STDR con el que se va a trabajar para realizar las diferentes simulaciones del robot AmigoBot. El siguiente paso para la configuración del sistema es la introducción de ruido sensorial al simulador.

El simulador es un entorno totalmente ideal, en el que no existen errores de odometría, deslizamientos de las ruedas del robot, errores en las medidas de los sensores, etc. Todo esto hace que el simulador devuelva la posición absoluta del robot respecto al mapa sin ningún tipo de error. Por lo tanto, si se quiere abordar la simulación lo más cercana a la realidad se deben introducir una serie de ruidos que se encarguen de modificar el valor de la odometría para que sea realista, añadiendo un ruido acumulativo.

Esto se realizará mediante el paquete *aux_files* que ha sido creado por el profesorado de la asignatura Robótica Móvil. Este paquete configura el error de odometría en el simulador para que este sea no nulo. A continuación, en el Listado 3.6 se tiene el código fuente del fichero *aux_files.launch*, en el cual se pueden modificar los parámetros de los errores del AmigoBot en el simulador. Estos parámetros son *error_x*, *error_y* y *error_a*.

Listing 3.6: Código fuente del ficher aux_files.launch

```
<launch>
  <node pkg="aux_files" type="local_odom" name="local_odom" >
  <!--remap from="tf" to="tf_new"/> -->
  <param name="error_x" value="0.0"/>
  <param name="error_y" value="0.0"/>
  <param name="error_a" value="0.0"/>
  </node>

  <node name="tf_repub" pkg="aux_files" type="tf_repub">
  <rosparam param="polling_frequency">100.0</rosparam>
        <rosparam param="frame_pairs">
        - source_frame: robot0_laser_1
          target_frame: robot0
        - source_frame: robot0_sonar_0
          target_frame: robot0
        - source_frame: robot0_sonar_1
          target_frame: robot0
        - source_frame: robot0_sonar_2
          target_frame: robot0
        - source_frame: robot0_sonar_3
          target_frame: robot0
        - source_frame: robot0_sonar_4
          target_frame: robot0
        - source_frame: robot0_sonar_5
          target_frame: robot0
        - source_frame: robot0_sonar_6
          target_frame: robot0
        - source_frame: robot0_sonar_7
          target_frame: robot0
        </rosparam>
  <remap from="tf" to="tf_sim" />
  <remap from="tf_changes" to="tf" />
  </node>
</launch>
```

Como se puede ver, en este fichero se crean dos nodos: el primero es */local_odom* que localiza al robot en la posición inicial (0,0) y le asigna un error acumulativo por defecto. El segundo nodo que se crea

es ***/tf_repub*** que recoge la información de los marcos de coordenadas del láser y de los sonar respecto al robot. En las figuras 3.3 y 3.4 se puede ver el antes y el después de ejecutar el paquete *aux_files*. Se puede notar a simple vista que aparecen los nodos que se han mencionado anteriormente.

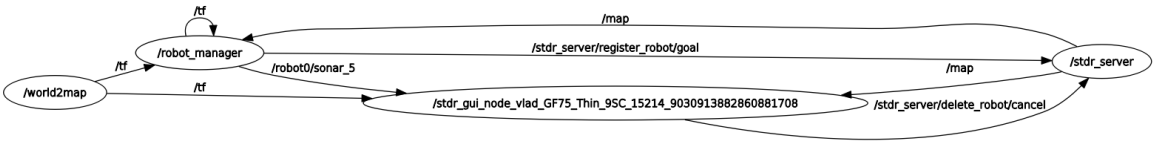


Figura 3.3: Árbol de transformadas antes de ejecutar el paquete *aux_files*.

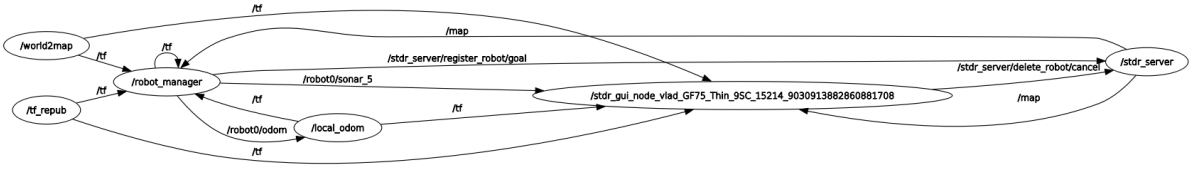


Figura 3.4: Árbol de transformadas después de ejecutar el paquete *aux_files*.

Este paquete *aux_files* publicará el resultado de las operaciones en los topics de ROS: */robot0/local_odom* y */tf*. Serán estos topics a los que se tendrá que suscribir MatLab para la realización de las diferentes ejercicios. Además, se deben realizar una serie de modificaciones en el simulador, debido a que este también publica en el topic */tf*, por lo que se siguen los siguientes pasos:

1. Se realiza una copia del archivo *server_with_map_and_gui_plus_robot.launch*, renombrándolo *server_with_map_and_gui_plus_robot_and_noise.launch*. En este nuevo archivo se editan los nodos del siguiente modo ***<remap from="tf"to="tf_sim/>***.

2. Se editan los archivos incluidos en *server_with_map_and_gui_plus_robot_and_noise.launch*, tales como *robot_manager.launch* y *stdr_gui.launch*. Se deben renombrar y editar los nodos de igual forma.

Una vez editado el entorno del simulador, se debe instalar el paquete aux_files, descomprimiéndolo en el directorio /src del espacio de trabajo y recompilando con catkin_make.

Para ejecutar este paquete se debe llamar al siguiente comando:

Listing 3.7: Comando para ejecutar el paquete aux_files

```
1 roslaunch aux_files aux_files.launch
```

Una vez que se tiene instalado ROS y todas las herramientas relacionadas, se debe instalar MatLab con la *Robotics System Toolbox*. Si se está ejecutando MatLab y ROS en máquina distintas ha de configurarse también la red de tal manera que exista conexión entre ellas. Para ello, todas las máquinas deben estar en la misma red, algo similar a lo que viene ilustrado en la figura 3.5.

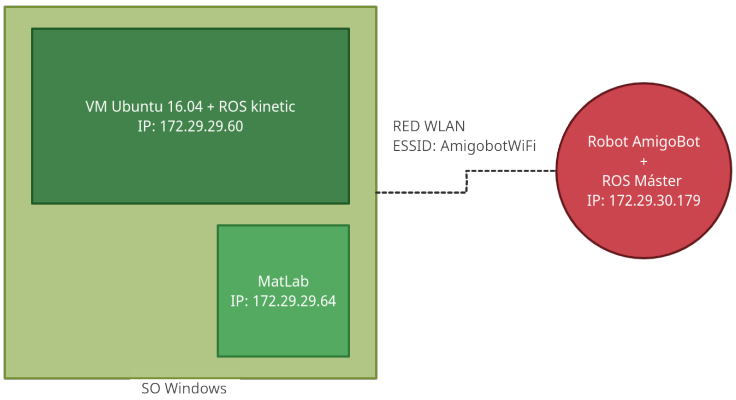


Figura 3.5: Ejemplo de red ROS con diferentes sistemas.

En esta Figura 3.5, MatLab se está ejecutando en el SO Windows en nativo, mientras que ROS se está ejecutando en una máquina virtual con Ubuntu. Todo esto se conecta con el robot AmigoBot mediante la misma red WiFi (en este caso, mediante la red WiFi del laboratorio "AmigoBotWiFi"). Las IPs utilizadas en esta Figura, son a modo de ejemplo para indicar que todas las máquinas están en la misma red.

Para que exista conexión entre las diferentes máquinas se deben configurar una serie de variables de entorno con la dirección IP correspondiente a cada máquina. Las variables de entorno que se deben configurar en Ubuntu en *.bashrc* son las siguientes:

Listing 3.8: Configuración de variables de entorno de red ROS en Ubuntu

```
1 export ROS_MASTER_URI=http://IP_ROSMASTER_MACHINE:11311
2 export ROS_IP=IP_LOCAL_MACHINE
```

La variable **IP_ROSMASTER_MACHINE** es la dirección IP de la máquina que ejecuta el Máster y solo debe haber un Máster por red. La variable **IP_LOCAL_MACHINE** es la dirección de la propia máquina.

Para el caso de MatLab, la conexión es similar, cambiando un poco la forma de la instanciación:

Listing 3.9: Configuración de variables de entorno de red ROS en MatLab

```
1 setenv('ROS_MASTER_URI','http://IP_ROSMASTER_MACHINE:11311')
2 setenv('ROS_IP','IP_LOCAL_MACHINE')
```

Ahora ya se sabe la forma de configurar todo el sistema para proceder con la realización de las diferentes prácticas.

Para poner en marcha toda la plataforma correctamente se debe tener en cuenta si se va a trabajar con el simulador STDR o con el robot real. Para trabajar con el simulador se deben realizar los siguientes pasos:

1. En primer lugar, se debe lanzar el simulador STDR dentro de la máquina virtual. La máquina virtual actuará como Máster del sistema.

2. En segundo lugar, continuando en la máquina virtual, si se quieren aplicar ruidos a la odometría, se debe lanzar el archivo *aux_files.launch* visto anteriormente.

3. A continuación se ejecuta el programa MatLab en la máquina anfitrión.
4. Ahora, si se quiere teleoperar el robot, se debe lanzar el paquete *teleop_twist_keyboard* (ver anexo A.2) dentro de la máquina virtual.
5. Una vez se están ejecutando todas las herramientas anteriores, en el programa MatLab se deben lanzar los scripts encargados de establecer la conexión entre las dos máquinas. Estos scripts realizan la conexión con el Máster, y, además, realizan la declaración de los nodos *subscribers* y los *publishers* que utiliza el simulador para comunicarse.

Por otro lado, si se quiere trabajar con el robot real, se deben llevar a cabo estos otros pasos:

1. En primer lugar, se debe encender el robot real. Al inicializarse el sistema del robot, en este se va a iniciar el Máster, al cual posteriormente se tendrá que conectar desde MatLab.
2. El siguiente paso es lanzar el programa MatLab.
3. A continuación, para teleoperar el robot real, se lanza el nodo *teleop_twist_keyboard*.
4. Desde MatLab se realiza la conexión con el robot real AmigoBot, y, se realiza la declaración de los nodos del robot real.

3.2 Práctica de Mapeado y Localización

En esta sección, se centra en el desarrollo de la práctica en la que se van a implementar diferentes métodos de mapeado y localización con el robot AmigoBot, tanto real como simulado. En primer lugar, se abordará el mapeado puro con posiciones conocidas. A continuación, se implementará la localización y mapeado simultáneos con la técnica **SLAM**. Por último, se trabajará con la técnica **AMCL** para obtener la localización estimada del robot.

3.2.1 Mapeado con posiciones conocidas

Con este método el mapeado será realizado a partir de las medidas del sensor láser y conocida la odometría, monitorizando en todo momento el entorno alrededor del robot y registrando la información en un mapa de ocupación. Este es un método poco real debido a que en la realidad la posición del robot no es conocida de modo preciso.

3.2.1.1 Implementación en MatLab

El script utilizado para implementar este método es ***"MappingWithKnownPoses.mlx"***, en el cual se genera el objeto del mapa, utilizando la clase ***"occupancyMap"***. Esta clase permite crear un mapa de rejilla de ocupación en 2 dimensiones, indicando como argumentos los siguientes parámetros del mapa: ancho del mapa, alto del mapa y resolución del mapa.

Algunos de los parámetros más interesantes y que se van a utilizar de esta clase *"occupancyMap"* son:

- ***"FreeThreshold / OccupiedThreshold"***: definen a partir de qué valor las celdas se consideran vacías u ocupadas.
- ***"GridLocationInWorld"***: indica las coordenadas en el mundo de la esquina inferior izquierda del mapa de rejilla

- ***"LocalOriginInWorld"***: indica la localización del origen del marco local de coordenadas en coordenadas globales del mundo.

El objeto creado a partir de esta clase tiene multitud de funciones, algunas de ellas son:

- ***checkOccupancy***: comprueba si el valor de una localización es libre, ocupado o desconocido.
- ***insertRay***: permite insertar los haces del láser en el mapa.
- ***local2world***: convierte coordenadas locales a coordenadas globales del mundo.

En primer lugar, dentro del script ***"MappingWithKnownPoses.mlx"*** se crea el objeto del mapa, con dimensiones de 500 píxeles por 500 píxeles. A continuación se define su origen, que se encuentra situado en la esquina inferior izquierda, como la coordenada (0, 0) del mundo. La definición del origen se realiza escribiendo en el parámetro ***GridLocationInWorld*** explicado anteriormente.

Listing 3.10: Configuración del mapa mediante la clase *occupancyMap* y su parámetro *GridLocationInWorld*

```
map = occupancyMap(25, 25, 20);
map.GridLocationInWorld = [0,0];
```

Este script posee un bucle principal, en el cual se irá construyendo el mapa del entorno a medida que el robot se mueva por el mismo. El movimiento del robot se realizará con la teleoperación, explicada en mayor detalle en el Anexo A.2. Para realizar la mencionada reconstrucción, lo primero que se debe hacer es obtener el último valor del láser que se ha proporcionado en el topic correspondiente:

Listing 3.11: Lectura del último mensaje del láser

```
msg_laser = laser.LatestMessage;
```

La lectura del láser debe ser referenciada a la posición del robot en el momento de la lectura, para lo cual se obtiene la transformada *"pose"* entre el marco de coordenadas del láser (source) y la odometría (target):

Listing 3.12: Transformación entre el marco de coordenadas del láser y la odometría

```
pose=(tftree,'odom','robot_0_laser_1',msg_laser.Header.Stamp,Timeout,2);
```

A continuación, se debe obtener la posición del robot [x y yaw] gracias a la función de MatLab *quat2eul*, que convierte el cuaternion proporcionado por el topic.

Listing 3.13: Código para extraer la posición del cuaternion

```
position=[pose.Transform.Translation.X, pose.Transform.Translation.Y];
    orientation=quat2eul([pose.Transform.Rotation.W, ...
        pose.Transform.Rotation.X, ...
        pose.Transform.Rotation.Y, ...
        pose.Transform.Rotation.Z], 'ZYX');
    robotPose = [position, orientation(1)];
```

Se extraen los rangos y los ángulos del mensajes del láser:

Listing 3.14: Código de extracción de los rangos y los ángulos del mensaje del láser

```
ranges = msg_laser.Ranges;
angles = msg_laser.AngleMin:msg_laser.AngleIncrement:msg_laser.AngleMax;
ranges(isinf(ranges)) = 7.8;
```

Por último, se insertan las medidas del láser en el mapa utilizando la función *insertRay*.

Listing 3.15: Medidas del láser en el mapa con *insertRay*

```
insertRay(map, robotPose ,ranges, angles, 8);
```

Una vez preparado el script, se debe establecer la conexión entre MatLab y ROS. Para ello, se crean los scripts **conectar.m** y **ini_simulador.m**. En el script conectar.m se debe indicar la dirección IP de la máquina que ejecuta el ROS Máster, y, por otro lado, también se debe indicar la dirección IP de la propia máquina del usuario. Esto se realiza para que los topics creados por cada una de las máquinas sean accesibles por todos en la red ROS.

Listing 3.16: Script de MatLab conectar.m

```
setenv('ROS_MASTER_URI','http://IP_ROS_MASTER_MACHINE:11311')
setenv('ROS_IP','IP_LOCAL_MACHINE')
rosinit
```

Listing 3.17: Script de MatLab ini_simulador.m

```
% Subscriber a la odometria
odom = rossubscriber('/robot0/local_odom');
% Subscriber al laser
laser = rossubscriber('/robot0/laser_1',rostype.sensor_msgs_LaserScan);
% Subsriber a los sonares
sonar_0 = rossubscriber('robot0/sonar_0',rostype.sensor_msgs_Range);
sonar_1 = rossubscriber('robot0/sonar_1',rostype.sensor_msgs_Range);
sonar_2 = rossubscriber('robot0/sonar_2',rostype.sensor_msgs_Range);
sonar_3 = rossubscriber('robot0/sonar_3',rostype.sensor_msgs_Range);
sonar_4 = rossubscriber('robot0/sonar_4',rostype.sensor_msgs_Range);
sonar_5 = rossubscriber('robot0/sonar_5',rostype.sensor_msgs_Range);
sonar_6 = rossubscriber('robot0/sonar_6',rostype.sensor_msgs_Range);
sonar_7 = rossubscriber('robot0/sonar_7',rostype.sensor_msgs_Range);
% Publisher de la velocidad
pub = rospublisher('/robot0/cmd_vel', 'geometry_msgs/Twist');
msg = rosmessage(pub);
r = robotics.Rate(10);
pause(1);
```

Estos dos scripts también se van a utilizar posteriormente en los siguientes ejercicios que se van a desarrollar en esta sección.

3.2.1.2 Resultados con el robot simulado

Una vez establecida la conexión entre MatLab y el nodo máster de ROS, se aborda la simulación tanto para el modelo sin ruido como para el modelo con ruido. La introducción de ruido en el sistema se explica en la sección 3.1.

Para el modelo sin ruido, el mapa se genera de una manera aproximada a la realidad, obteniendo mayor calidad del mapa. El resultado se puede ver en la Figura 3.6. En esta simulación se utilizó el mapa *simple_rooms*, visto en la Figura 3.7.

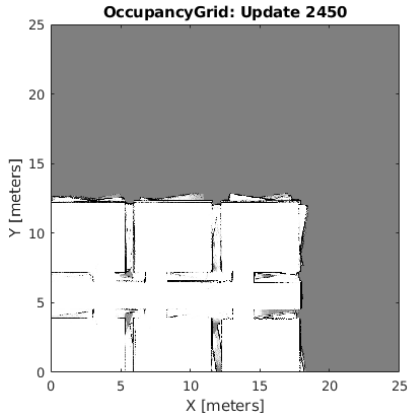


Figura 3.6: Mapa obtenido con la técnica "Mapeado con posiciones conocidas"

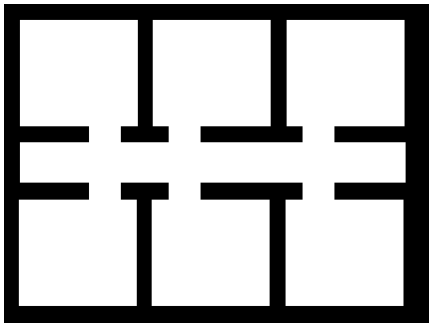


Figura 3.7: Mapa de referencia para la técnica "Mapeado con posiciones conocidas"

Sin embargo, aún trabajando sin ruido puede darse el caso en el que el robot se quede atascado en una pared o esquina del mapa. Esto genera que la odometría siga avanzando como si el robot se estuviera moviendo por el mapa, pero la lectura del láser es siempre la misma en esa posición. Esta circunstancia hace que el mapa se genere de forma incorrecta, distorsionada.

Al introducir ciertos valores de ruido en la odometría con ayuda del archivo aux_files, se puede ver cómo el mapa no se genera correctamente. En este caso, el error que se ha introducido fue de: **x = 0.1, y = 0.1, a = 0.1**. A pesar de ser valores bajos de error, en la siguiente Figura 3.8 puede observarse cómo afecta a la generación del mapa:

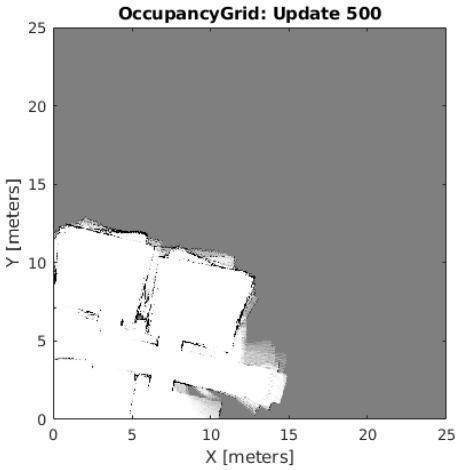


Figura 3.8: Mapa generado con ruido x = 0.1, y = 0.1, a = 0.1.

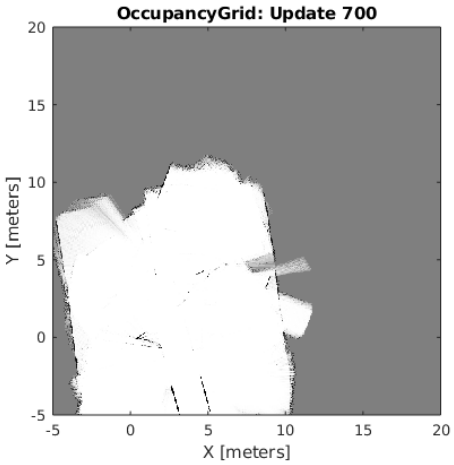


Figura 3.9: Mapa generado con ruido x = 0.5, y = 0.5, a = 0.5.

Si se incrementa el error de odometría por ejemplo a: **x = 0.5, y = 0.5, a = 0.5**, el resultado obtenido en la Figura 3.9 muestra que el mapa generado es prácticamente indistinguible. Al introducir un error en la odometría, si este es muy grande, dado a que es acumulativo, como puede observarse en las figuras anteriores 3.8 y 3.9, se vuelve inviable la generación del mapa del entorno.

3.2.1.3 Resultados con el robot real

Para realizar las pruebas de mapeado con el robot real, en primer lugar se deben modificar los scripts de MatLab *conectar.m* e *ini_simulador.m*. En el script *conectar.m* se debe establecer ROS_MASTER_URI con la IP del robot, ya que el ROS Máster se ejecuta dentro del sistema del robot. En el script *ini_simulador.m*, se cambian los topics del robot simulado por los topics creados por el robot real. Para conservar ambos scripts, lo que se hace es renombrar uno de ellos en *ini_real.m*. A continuación se presentan los resultados obtenidos con el robot real, obteniendo el mapa del laboratorio de la Escuela Politécnica de la Universidad de Alcalá en la figura 3.10.

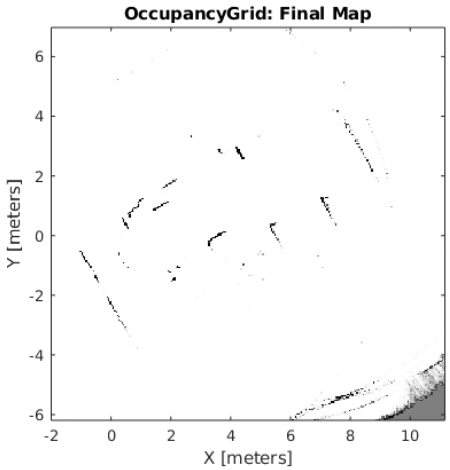


Figura 3.10: Mapa generado del laboratorio de la Escuela Politécnica de la Universidad de Alcalá mediante la técnica "Mapeado con posiciones conocidas".

En esta Figura 3.10, se puede observar el mapeado de una parte de los pasillos ubicados entre las mesas del laboratorio destinadas al alumnado. Aunque se puede observar que la calidad del mapa generado no es muy buena. Esto se debe al error de la odometría real del robot, la altura de toma de datos del sensor láser y la interacción con los diferentes elementos (reflexión especular, etc.).

3.2.2 Localización y mapeado simultáneos: SLAM

En esta sección se va a implementar la técnica SLAM que también está basada en los resultados que se obtienen de las medidas del láser. Esta técnica utiliza la optimización del grafo de posiciones, es decir, permite la corrección tanto del mapa como la trayectoria del robot a través de un cierre de lazo. De este modo, compensa los errores de la odometría en la creación del mapa y la reconstrucción de la trayectoria.

3.2.2.1 Base teórica

La técnica de SLAM es una técnica en la que se realiza el mapeado del entorno y al mismo tiempo se estima la posición del robot en ese mismo entorno. Por lo tanto, consiste en calcular una estimación de la ubicación del robot, x_t, y un mapa del entorno, m_t a partir de una serie de observaciones, o_t, durante un tiempo discreto con un paso de muestreo, t. Todos los valores enumerados son probabilísticos. El objetivo de esta técnica es obtener $P(m_t, x_t|o_{1:t})$.

La aplicación del teorema de Bayes es la base para actualizar constantemente la ubicación a posteriori,

teniendo en cuenta el mapa y la función de transición $P(x_t|x_{t-1})$:

$$P(m_t, x_t|o_{1:t}) = \sum_{m_{t-1}} P(o_t|x_t, m_t) \sum_{x_{t-1}} P(x_t|x_{t-1}) P(x_{t-1}|m_t, o_{1:t-1}) \tag{3.1}$$

Esta técnica se describe de manera más detallada en el artículo, consistente en dos partes, (Simultaneous Localization and Mapping: Part I [29] y Simultaneous Localization and Mapping: Part II [30]) escrito por *Hugh Durrant-Whyte* y *Tim Bailey*. En la primera parte de ese tutorial, se introduce una breve historia de los primeros desarrollos en SLAM. Se continua con la sección de formulación en forma bayesiana y se explica la evolución del proceso SLAM. También se describen las dos soluciones computacionales clave para el problema de SLAM: mediante el uso del filtro de Kalman Extendido (EKF-SLAM) y mediante el uso de filtros de partículas Rao-Blackwellized (FastSLAM). En la segunda parte del tutorial, se describen otras soluciones recientes al problema de SLAM.

El EKF-SLAM es la aplicación del filtro de Kalman extendido al problema de SLAM. Esta aproximación estima la posición del robot y la localización de los objetos en el entorno. Se asume que las correspondencias son conocidas. EKF-SLAM tiene problemas de convergencia a la hora de tener fuertes no linealidades. Un mapa con n marcas produce unas matrices de estado y de covarianzas de dimensiones $(3 + 2n)$. Un ejemplo de de estas matrices de estas dimensiones se puede ver en la siguiente figura 3.11.

$$\begin{pmatrix} x \\ y \\ \theta \\ m_{1,x} \\ m_{1,y} \\ \vdots \\ m_{n,x} \\ m_{n,y} \end{pmatrix} \begin{pmatrix} \sigma_{xx} & \sigma_{xy} & \sigma_{x\theta} & \sigma_{xm_{1,x}} & \sigma_{xm_{1,y}} & \cdots & \sigma_{xm_{n,x}} & \sigma_{xm_{n,y}} \\ \sigma_{yx} & \sigma_{yy} & \sigma_{y\theta} & \sigma_{ym_{1,x}} & \sigma_{ym_{1,y}} & \cdots & \sigma_{m_{n,x}} & \sigma_{m_{n,y}} \\ \sigma_{\theta x} & \sigma_{\theta y} & \sigma_{\theta\theta} & \sigma_{\theta m_{1,x}} & \sigma_{\theta m_{1,y}} & \cdots & \sigma_{\theta m_{n,x}} & \sigma_{\theta m_{n,y}} \\ \sigma_{m_{1,x}x} & \sigma_{m_{1,x}y} & \sigma_{\theta} & \sigma_{m_{1,x}m_{1,x}} & \sigma_{m_{1,x}m_{1,y}} & \cdots & \sigma_{m_{1,x}m_{n,x}} & \sigma_{m_{1,x}m_{n,y}} \\ \sigma_{m_{1,y}x} & \sigma_{m_{1,y}y} & \sigma_{\theta} & \sigma_{m_{1,y}m_{1,x}} & \sigma_{m_{1,y}m_{1,y}} & \cdots & \sigma_{m_{1,y}m_{n,x}} & \sigma_{m_{1,y}m_{n,y}} \\ \vdots & \vdots & \vdots & \vdots & \vdots & \ddots & \vdots & \vdots \\ \sigma_{m_{n,x}x} & \sigma_{m_{n,x}y} & \sigma_{\theta} & \sigma_{m_{n,x}m_{1,x}} & \sigma_{m_{n,x}m_{1,y}} & \cdots & \sigma_{m_{n,x}m_{n,x}} & \sigma_{m_{n,x}m_{n,y}} \\ \sigma_{m_{n,y}x} & \sigma_{m_{n,y}y} & \sigma_{\theta} & \sigma_{m_{n,y}m_{1,x}} & \sigma_{m_{n,y}m_{1,y}} & \cdots & \sigma_{m_{n,y}m_{n,x}} & \sigma_{m_{n,y}m_{n,y}} \end{pmatrix}$$

Figura 3.11: Matriz de estado y matriz de covarianzas con dimensiones de (3+2n).

Por otro lado, FastSLAM fue el primero en representar el modelo como no lineal y la distribución de la posición como no Gaussiana. En FastSLAM un mapa grande se considera como un conjunto de submapas locales, lo que permite eliminar la dependencia de los puntos de referencia entre sí y, por lo tanto, reducir significativamente el tiempo para volver a calcular la evolución del estado del sistema. El coste computacional es proporcional al número de partículas empleado y al número de objetos en el mapa.

En esta sección se trabajará, más en concreto, con el algoritmo de SLAM implementado en MatLab en la ***Robotics System Toolbox***. Este algoritmo se llama lidarSLAM que utiliza los escaneos del Light Detection and Ranging (LIDAR) para mapear el entorno, realizando diferentes correlaciones con medidas anteriores para realizar un grafo posicional subyacente de la trayectoria del robot. Este grafo posicional contiene nodos conectados que representan las posiciones relativas del robot. El algoritmo llega a corregir la estimación de la posición del robot en el grafo cuando detecta un cierre de bucle. Un cierre de bucle significa que el robot detecta que ya había pasado por este entorno anteriormente.

En este algoritmo se asume que los datos provienen de un robot que navega por un entorno y obtiene las medidas del LIDAR de forma incremental a lo largo de toda su trayectoria. De este modo, las medidas son comparadas con las medidas más recientes para identificar posiciones relativas, agregándose al grafo de forma incremental.

Se debe tener en cuenta que, en cualquier sistema, cuando se trabaja con la técnica SLAM, el entorno

y los sensores del robot afectan al rendimiento y la calidad de la correlación de datos. Se deben ajustar correctamente los parámetros del sistema para un entorno en concreto.

3.2.2.2 Implementación en MatLab

En este apartado se trabaja con el script *OnlineSLAM.mlx*, realizando las adaptaciones necesarias para poder ejecutarlo tanto con el simulador STDR como con el robot real.

En este script viene implementada la clase *robotics.LidarSLAM*, la cuál es la encargada de realizar la localización y el mapeo simultáneos a través de la entrada del sensor láser. Esta clase realiza la correlación de los diferentes escaneos mediante la coincidencia de análisis. Además, busca el cierre de bucles para superponer regiones asignadas previamente.

Esta clase presenta una serie de propiedades que se describirán brevemente a continuación:

- ***PoseGraph***: Grafo de posición subyacente que conecta las diferentes medidas de los escaneos láser. La agregación de los diferentes escaneos a la clase LidarSLAM actualiza este grafo de posición.
- ***MapResolution***: Resolución de la cuadrícula de ocupación, especificada en celdas por metro. Por defecto, 20 celdas por metro.
- ***MaxLidarRange***: Rango máximo del sensor LIDAR, especificado en metros. Por defecto, 8 metros.
- ***OptimizationFcn***: Función de optimización del grafo de posición. Por defecto, se establece la función *optimizePoseGraph*.
- ***LoopClosureThreshold***: Umbral para aceptar los cierre de lazo. Un umbral alto corresponde a una mejor coincidencia. Por defecto, se establece un valor de 100.
- ***LoopClosureSearchRadius***: Radio de búsqueda para la detección de cierre de lazo. El aumento de este radio afecta al rendimiento al aumentar el tiempo de búsqueda. Se debe ajustar en función del entorno. Por defecto, se establece un valor de 8 metros.
- ***LoopClosureMaxAttemps***: Número de intentos de encontrar cierres de lazo. Si se aumenta el número de intentos se afecta al rendimiento al aumentar el tiempo de búsqueda. Por defecto, se establece un valor de 1.
- ***LoopClosureAutoRollback***: Permite la reversión de cierres de lazo. Se especifica como *true* o *false*. Si la propiedad está en *true*, la propiedad rechaza el cierre de lazo si detecta un cambio brusco en el error residual. Por defecto *true*.
- ***OptimizationInterval***: Número de cierres de lazo aceptados para realizar la optimización. Por defecto, realiza la optimización cada vez que se cierra un lazo. Por defecto, se establece un valor de 1.
- ***MovementThreshold***: mínimo cambio en la posición para procesar las medidas. Por defecto, se establece un vector de [0 0].
- ***ScanRegistrationMethod***: Método de registro del escaneo.

De todas estas propiedades, se van a modificar cuatro de ellas en el script *OnlineSLAM.mlx*. Las demás propiedades se establecerán con los valores por defecto. Estas propiedades son:

- ***MapResolution***: Se establecerá la resolución del mapa acorde a la resolución del mapa en el simulador STDR.
- ***MaxLidarRange***: Se establecerá un valor acorde al rango máximo del LIDAR. Se recomienda establecer un valor ligeramente menor a este rango máximo debido a que las lecturas suelen ser menos precisas entorno al rango máximo.
- ***LoopClosureThreshold*** y ***LoopClosureSearchRadius***: Se establecerán unos valores obtenidos de forma empírica.

Una vez creado este objeto y establecidos los valores de las propiedades, el script entra en un bucle en el que se añaden las medidas del láser al objeto. Estas medidas son comparadas con las medidas que se han tomado anteriormente para establecer cierres de lazo y optimizar el grafo de posición. Una vez se ha realizado el recorrido del mapa y se han añadido todas las medidas del láser al objeto, se crea el mapa mediante la función *buildMap*. A esta función se le debe pasar como parámetros las propias medidas del láser, las posiciones, la resolución del mapa y el rango máximo del LIDAR.

3.2.2.3 Resultados con el robot simulado

Una vez establecidos todos los parámetros del algoritmo en MatLab, se procede a su ejecución en el entorno simulado con el simulador STDR. Los resultados que se obtienen se pueden ver en las siguientes figuras, donde la Figura 3.12 es la obtenida mediante la técnica SLAM y la Figura 3.13 es el mapa utilizado como entorno del robot en el simulador.

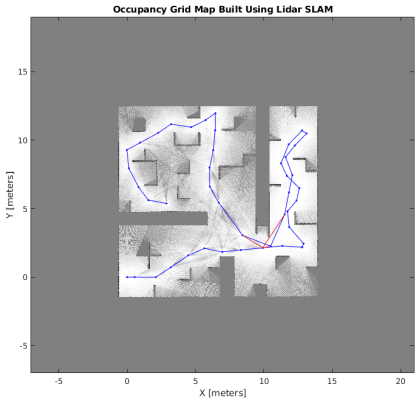


Figura 3.12: Mapa generado con la técnica SLAM

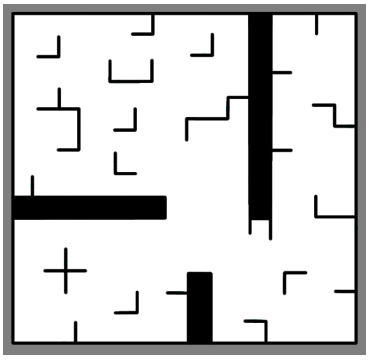


Figura 3.13: Mapa utilizado como entorno del robot simulado

Se puede ver que el algoritmo de SLAM haciendo el mapeado del entorno, recrear de forma muy precisa el mapa. En comparación con la técnica de mapeado con posiciones conocidas, se obtienen mapas más fieles y robustos. En el resultado de la Figura 3.12 se pueden ver zonas grises que indican que son zonas no observadas, lo que significa que el robot no ha pasado por esa zona del mapa y no tiene suficientes datos. Por el contrario, las zonas blancas del mapa indican alta probabilidad de no estar ocupadas. En el mapa de referencia se puede observar que posee zonas negras, las cuales indican alta probabilidad de estar ocupadas. En el mapa generado se puede observar mediante la línea azul el grafo posicional, los nodos conectados que representan las posiciones relativas del robot. Se intuye que las zonas blancas se distribuyen de mayor forma alrededor de las posiciones relativas por donde ha pasado el robot, lo que es lógico. Hay mayor flujo de datos sobre estas zonas. Las líneas rojas sobre el mapa generado indican los cierres de bucle identificados en el conjunto de datos.

3.2.2.4 Resultados con el robot real

Una vez realizadas las pruebas con el robot simulado, se ha pasado a probar la técnica SLAM directamente sobre el robot real. Para ello se ha realizado la conexión con el robot real.

Para la comprobación del funcionamiento de la técnica SLAM sobre el robot real, se ha procedido a mapear el entorno del propio laboratorio en el que se estaban realizando las pruebas. Los resultados de este mapeado con la técnica SLAM se puede ver en la Figura 3.14

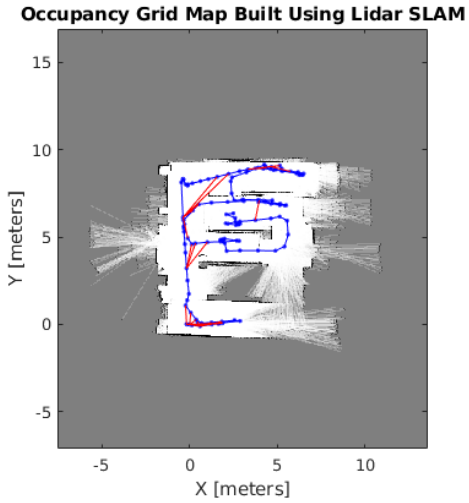


Figura 3.14: Mapa generado del laboratorio de la Escuela Politécnica de la Universidad de Alcalá mediante la técnica "SLAM".

Se puede observar que se ha realizado un mapeado bastante preciso de las paredes interiores del laboratorio. Además se puede intuir que la puerta del laboratorio estaba abierta debido a que hay líneas blancas que indican espacio libre de obstáculos justo en la zona de la puerta (pared izquierda sobre el mapa generado). De peor forma se han mapeado las zonas de las filas de los alumnos. Se pueden llegar a intuir estas filas pero pueden llevar a confusión, debido a que hay zonas conectadas (zonas blancas), mientras que en la realidad entre fila y fila no hay espacios libres. Las líneas azules indican el grafo posicional, el cual también puede producir confusión, debido a que hay trayectorias que no es posible realizar porque están obstaculizadas por las mesas de laboratorio.

Se puede llegar a distinguir que hay mayor número de líneas rojas que indican cierres de bucle identificados en el conjunto de datos. Estos cierres de bucles parecen estar bien cerrados porque se corresponden bastante bien con la realidad del aula del laboratorio.

Se puede ver claramente que el fondo del laboratorio no ha sido mapeado de forma correcta. Esto es debido a que es bastante complicado manipular el robot por las filas de las mesas del laboratorio teniendo que esquivar sillas, alumnos y sus mochilas correspondientes. Por este motivo solo se han realizado trayectorias cortas por estos pasillos obteniendo pocos datos de estos para un buen mapeado, en comparación con la pared de entrada al laboratorio.

3.2.3 Localización con AMCL

En esta sección se introducirá la técnica de localización mediante el algoritmo MCL. Se abordará la resolución de la localización del robot mediante la técnica AMCL (*Adaptive Monte Carlo Localization*). Además, se realizará una implementación en MatLab de esta técnica y se probará el resultado tanto con el robot simulado como con el robot real.

3.2.3.1 Base teórica

El algoritmo MCL es una técnica mediante la cual los robots se localizan mediante un filtro de partículas. Dado un mapa del entorno, el algoritmo estima la posición y orientación del robot mientras se mueve y percibe el entorno. El algoritmo utiliza un filtro de partículas en el que cada partícula representa un estado posible del robot.

De este modo, como se describe en el artículo ***Monte Carlo Localization for Mobile Robots*** [31], el objetivo de este algoritmo es calcular recursivamente en cada paso de tiempo, k, el conjunto de partículas $S_k = \{s_k^i; i = 1...N\}$ que se extraen de la densidad a posteriori $P(x_k|Z^k)$ del estado actual condicionado a todas las mediciones. El algoritmo procede en dos fases:

- **Fase de predicción**: En esta fase se parte de un conjunto de partículas calculadas en la iteración anterior y se aplica el modelo de movimiento a cada partícula de la población. Haciendo esto se obtiene un nuevo conjunto aún sin incorporar las medidas de los sensores.
- **Fase de actualización**: En esta fase se toman en cuenta las medidas y el peso de cada partícula (a mayor peso, mayor probabilidad de que sea la posición real del robot). Entonces, se realiza un remuestreo, en el cuál se seleccionan las muestras con mayor peso, obteniéndose un nuevo conjunto de partículas.

Estas dos fases son repetidas recursivamente. Para el tiempo $k = 0$ se generan las partículas dependiendo de si se quiere realizar una localización global o una localización local.

Para una localización global, la densidad de probabilidad inicial puede ser una densidad uniforme distribuida por todas las posiciones posibles del entorno. Por el contrario, en una localización local, en la que se da la posición aproximada del robot, la densidad de partículas es centrada alrededor de esa posición.

Por otro lado, en esta sección se va a presentar y se va a trabajar con la técnica AMCL [32]. Esta técnica es una variante del algoritmo MCL y ajusta de forma dinámica el número de partículas basándose en el método de la divergencia de Kullback-Leibler (KL) [33], garantizando que la distribución de partículas converja con la verdadera distribución del estado del robot en función de todas las mediciones de movimiento y sensores anteriores con alta probabilidad.

Como se sabe y como se verá en este documento, el algoritmo MCL tiene una desventaja y es que la carga computacional de una actualización del algoritmo es lineal al número de partículas necesarias para la estimación. Por lo tanto, esta nueva técnica se ha desarrollado, como muchas otras, para hacer un uso más efectivo de las partículas disponibles, permitiendo así conjuntos de partículas de tamaño razonable.

AMCL es un enfoque para aumentar la eficiencia de los filtros de partículas adaptando el número de muestras a la incertidumbre del estado subyacente. Este enfoque puede ser muy beneficioso, por ejemplo, en una localización global, debido a la alta incertidumbre al comienzo se necesitan un gran número de partículas para representar con precisión su creencia (ver Figura 3.15).

Figura 3.15: Mapa durante una localización global mediante AMCL. Comienzo del algoritmo.

En el momento en el que el robot sabe la ubicación estimada en la que se encuentra, solo una pequeña cantidad de muestras es suficiente para rastrear con precisión su posición (ver Figura 3.16).

Figura 3.16: Mapa durante una localización global mediante AMCL. El robot conoce su ubicación aproximada.

Por lo tanto, este enfoque adapta el número de muestras durante el proceso de localización, eligiendo conjuntos de muestras grandes durante la localización global y conjuntos de muestras pequeños para el seguimiento de posición.

A continuación, en la tabla 3.1 se presenta un ejemplo del procedimiento del algoritmo MCL para un robot en un pasillo de una dimensión con tres puertas. El robot posee un sensor que devuelve verdadero o falso dependiendo de si hay una puerta o no.

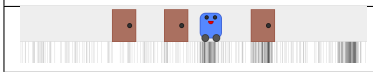
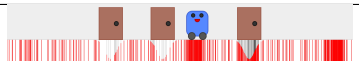
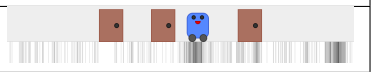
t = 0		
		
El algoritmo se inicializa con una distribución uniforme de partículas. El robot considera que es igualmente probable que esté en cualquier lugar del pasillo.	El robot detecta la puerta. Asigna un peso a cada una de las partículas. Las partículas que pueden dar esos datos del sensor, adquieren mayor peso.	Remuestreo: el robot genera un conjunto nuevo de partículas, la mayoría de las cuales se generan alrededor de las partículas anteriores con mayor peso. Ahora cree que está en una de las tres puertas.
t = 1		
		
Se actualiza el movimiento. El robot se mueve a la derecha. Todas las partículas también se mueven hacia la derecha y se aplica algo de ruido gaussiano. El robot está ubicado físicamente entre la segunda y la tercera puerta.	Se realiza una medida con el sensor. El robot no detecta ninguna puerta. Asigna un peso a cada una de las partículas. Las partículas que pueden dar esos datos del sensor, adquieren mayor peso.	El robot vuelve a generar un conjunto de nuevas partículas, la mayoría de las cuales se generan alrededor de las partículas anteriores con mayor peso. El robot ahora cree que está en uno de los dos lugares.
t = 2		
		
Ahora el robot se mueve a la izquierda. Todas las partículas también se mueven a la izquierda, aplicando ruido gaussiano. El robot se encuentra físicamente ante la segunda puerta.	Se realiza una nueva medida con el sensor. El robot detecta la puerta. Asigna un peso a cada una de las partículas. Las partículas que pueden dar esos datos del sensor, adquieren mayor peso.	El robot vuelve a generar un conjunto de nuevas partículas, la mayoría de las cuales se generan alrededor de las partículas anteriores con mayor peso. El robot se ha localizado correctamente.

Tabla 3.1: Ejemplo de algoritmo MCL

3.2.3.2 Implementación en MatLab

Para trabajar con este método se dispone en la toolbox de MatLab de un script llamado ***AMCL_Localization.mlx***, que al igual que en las técnicas presentadas anteriormente, se tiene un esqueleto del programa y parte del código ya está implementada. El alumnado debe completar el código para el correcto funcionamiento de la localización con AMCL.

Para trabajar con esta técnica en el entorno de MatLab, este posee una clase llamada ***robotics.MonteCarloLocalization***. Esta clase utiliza los datos proporcionados por el sensor láser y la odometría para obtener la localización aproximada del robot sobre un mapa conocido a priori.

Para la localización del robot, este método utiliza un filtro de partículas para estimar la posición del robot sobre el mapa. Cada partícula representa un posible estado del robot. Con el tiempo, todas las partículas convergen alrededor de una sola ubicación a medida que el robot se mueve y detecta diferentes características del entorno.

Esta clase presenta diferentes propiedades, que son:

- ***IntialPose***: Posición inicial del robot que se utiliza para iniciar la localización. Es un parámetro que se especifica como vector de tres elementos: posición (x, y) y la orientación (theta). Iniciar el algoritmo indicando este parámetro, permite reducir el número máximo de partículas. El valor por defecto es [0 0 0].

- ***InitialCovariance***: Indica la covarianza de la distribución gaussiana para la posición inicial. Es una matriz diagonal que da una estimación de la incertidumbre del parámetro ***IntialPose***. El valor por defecto de esta propiedad es diag([1 1 1]).

- ***GlobalLocalization***: Es un flag que puede valer *true* o *false*, que indica al algoritmo si debe utilizar la localización global o la localización local. Si el parámetro se establece en *false*, el algoritmo inicia las partículas teniendo en cuenta las propiedades ***IntialPose*** y ***InitialCovariance***. Un valor *true* en esta propiedad distribuye uniformemente las partículas por todo el mapa e ignora las propiedades anteriores. La localización global requiere un mayor número de partículas para cubrir la totalidad del mapa. El valor por defecto es *false*.

- ***ParticleLimits***: Vector de dos elementos que indica el mínimo y máximo número de partículas. El valor por defecto para esta propiedad es [500 5000].

- ***SensorModel***: Campo de probabilidad del modelo de sensor. Por defecto utiliza el objeto *likelihoodFieldSensorModel*, que crea un objeto de modelo de sensor de campo de probabilidad para sensores de rango. En este objeto se indican, por ejemplo, la posición del sensor relativa a las coordenadas locales del robot.

- ***MotionModel***: Campo de probabilidad del modelo de movimiento de odometría. El valor predeterminado utiliza el objeto *odometryMotionModel* que crea un modelo de movimiento de odometría para vehículos de accionamiento diferencial. En este objeto se puede indicar el ruido gaussiano para el movimiento del vehículo.

- ***UpdateThreshold***: Cambio mínimo en alguna de las coordenadas [x y theta], necesario para desencadenar una actualización de las partículas. Esta propiedad se especifica como un vector de tres elementos y su valor predeterminado es [0.2 0.2 0.2]. La localización actualiza las partículas si se cumple el cambio mínimo en cualquiera de las coordenadas.

- ***ResamplingInterval***: Número de actualizaciones del filtro de partículas entre dos muestras. El valor por defecto es 1.
- ***UseLidarScan***: Es un objeto que puede tomar como valores *true* o *false* e indica si se utiliza el objeto *LIDARScan* como entrada de escaneado. Por defecto se establece esta propiedad en *false*.

En el script ***AMCL_Locatlization.mlx***, lo primero que se hace es cargar un mapa, que puede ser uno de los mapas generados por las técnicas de mapeado anteriormente descritas. A continuación, el movimiento del robot se modela con el objeto explicado anteriormente ***MotionModel***. Este objeto posee una propiedad que representa la incertidumbre en el movimiento lineal y rotacional del robot. El aumento de esta propiedad permite una mayor propagación de las partículas utilizando medidas de odometría. Se establecen los valores de esta propiedad a los valores que vienen por defecto: [0.2 0.2 0.2 0.2]. Cada uno de estos valores del vector representan diferentes tipos de errores en el movimiento, siendo estos respectivamente: Error rotacional debido a movimiento rotacional, Error rotacional debido a movimiento traslacional, Error traslacional debido a movimiento traslacional, Error traslacional debido a movimiento rotacional.

Se sigue con la configuración del modelo del sensor. Para ello se definen los parámetros para el objeto ***SensorModel*** de acuerdo a los parámetros reales del sensor que posee el robot real. Para conseguir mejores resultados es necesario modelar el sensor del robot lo más parecido posible al sensor real. Para hacerlo, el objeto ***SensorModel*** posee una serie de propiedades que se van a establecer. En primer lugar, se define la propiedad ***SensorLimits*** con los valores [0 8]. Es un vector de dos elementos, el primero indica el mínimo rango del sensor y el segundo elemento indica el máximo rango del sensor. Estos valores son introducidos en metros. En segundo lugar, se define el mapa en la propiedad ***Map***. Este mapa representa el entorno del robot como una rejilla con valores binarios. Un valor *true(1)* representa un obstáculo, por el contrario, un valor *false(0)*, indica un espacio libre. El mapa que se le proporciona es el mapa que se ha cargado anteriormente. Por último se deben establecer los parámetros correspondientes a la propiedad ***SensorPose***, que indica la posición del sensor relativa al sistema de coordenadas del robot. Es un vector de tres elementos [x y theta]. Para obtener estos valores, se debe realizar una transformación del marco de coordenadas del láser al marco de coordenadas base del robot. Esta transformación devuelve un cuaternión que se transforma en ángulos de Euler. Así se obtienen los tres parámetros que se pasan a esta propiedad del objeto ***SensorModel***.

El siguiente paso es la definición de la propia clase ***robotics.MonteCarloLocalization***. Una vez creada se establecen los valores para las propiedades y objetos explicados anteriormente. Después de este proceso, se entra en un bucle en el que se van actualizando en cada iteración los datos del sensor láser y los datos de la odometría del robot. Estos datos se introducen como parámetro a la clase AMCL creada, y que, tras procesarlos, devuelve la posición estimada del robot y la covarianza estimada de esta. También, en cada iteración del bucle se actualiza la visualización de la posición estimada del robot, la posición de las partículas y las lecturas del láser.

3.2.3.3 Resultados con el robot simulado

El primer paso para probar la técnica AMCL es ejecutarla en simulación con el simulador STDR. El mapa utilizado para probar la localización del robot en simulación es el presentado en la Figura 3.12 que se había generado con la técnica SLAM a partir del mapa de la figura 3.13.

Como se sabe, esta técnica se puede ejecutar tanto para la localización local como para localización global. La localización local se realiza cuando se conoce la posición inicial del robot, o se conoce de forma aproximada. Esta localización local consiste en realizar un seguimiento de dicha posición que compense

los errores de odometría mediante el uso de observaciones del entorno. Por otro lado, en la localización global la posición inicial del robot es desconocida.

Conocido esto, se intenta la localización local del robot, dando una posición aproximada de su posición. Los resultados obtenidos se pueden observar en las Figuras 3.17, 3.18 y 3.19:

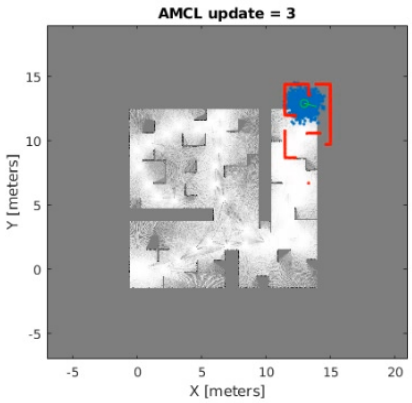


Figura 3.17: Inicialización de la localización local con el robot simulado

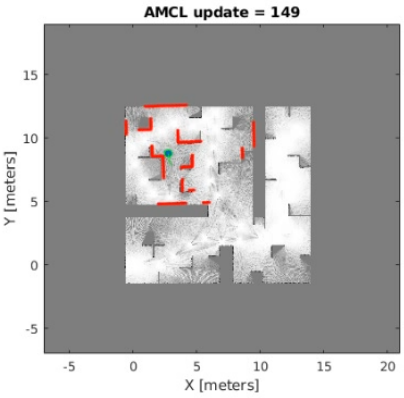


Figura 3.18: Resultado de la localización local con el robot simulado

Como se puede observar, en la localización local, todas las partículas del algoritmo se dispersan alrededor de la ubicación aproximada indicada del robot. Con el paso del tiempo, el robot se mueve por el entorno detectando diferentes características que le ayudan a localizarse. Se puede decir que el robot se ha localizado correctamente en la iteración 149 del algoritmo. Todas las partículas del algoritmo han convergido en un único punto. También, como se puede ver en la Figura 3.19, con el paso del tiempo y siguiendo con su movimiento, el robot sigue estimando de forma correcta la posición real del robot. Es importante mencionar que esta localización local se están utilizando un rango de partículas entre 500 y 5000 partículas.

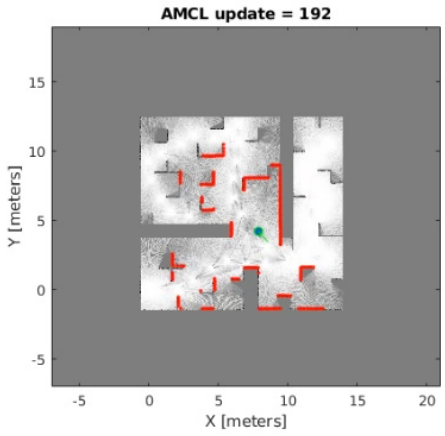


Figura 3.19: Localización local con el robot simulado

Una vez probada esta técnica con la localización local de forma satisfactoria, se continua con la localización global. En el script se debe indicar en la propiedad ***GlobalLocalization*** un valor de *true*. Jugando con los valores del número de partículas de la propiedad ***ParticleLimits***, se extrae que para la localización global el número de partículas debe ser mayor que en la localización local. Un rango entre 1000 y 10000 partículas. Esto se puede explicar sabiendo que en la localización global, las partículas se

deben distribuir por la totalidad del mapa del entorno del robot, debido a que la posición inicial del robot es desconocida. Así, debe haber un número de partículas elevado en comparación con la localización local donde las partículas se distribuyen alrededor de la posición inicial aproximada del robot.

El resultado de la localización global se puede apreciar en las siguiente Figuras 3.20 y 3.21:

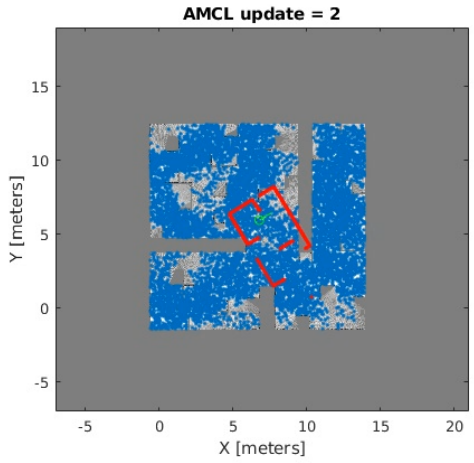


Figura 3.20: Inicialización de la localización global con el robot simulado

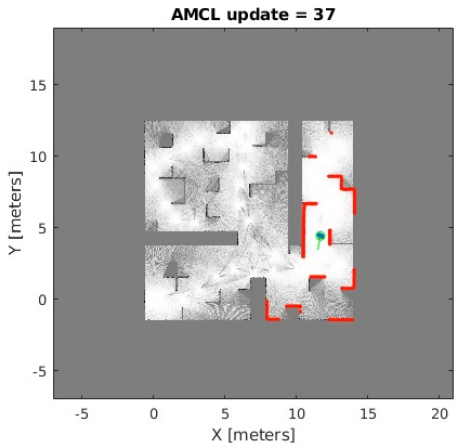


Figura 3.21: Resultado de la localización global con el robot simulado

Se comprueba que con esta técnica tanto para la localización local como para la localización global del robot simulado, el algoritmo implementado realiza de forma correcta la localización del robot.

3.2.3.4 Resultados con el robot real

Para poder realizar las pruebas sobre el robot real, se debe tener en cuenta que hay que modificar una serie de parámetros en el script que son: indicar los topics del robot real, se deben habilitar los motores del robot real, y se debe cambiar el mapa por el mapa generado del laboratorio 3.14.

Al igual que en la simulación, se empieza a probar esta técnica con la localización local. Se debe tener en cuenta que el robot real siempre inicializa su posición en la coordenada (0, 0) con orientación de 0 radianes. Esto, lo que quiere decir que en la localización local, se debe llevar el robot real a una posición aproximada del mapa cercana al (0, 0). De esta forma, haciendo todo el proceso, se consigue el resultado satisfactorio de la localización local del robot real en el entorno del laboratorio que se puede ver en la figura 3.22.

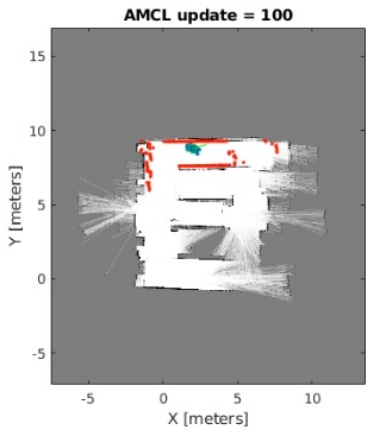


Figura 3.22: Localización local con el robot real

En la localización global se ha conseguido el mismo resultado correcto de la localización del robot real que se presenta en la figura anterior. Lo único que se ha modificado es el número de partículas, por un valor mayor que en la localización local del robot real.

3.3 Práctica de Navegación local y global

En esta sección se abordarán los conceptos y las técnicas de navegación tanto local como global. En primer lugar, se abordará el planificador local VFH. A continuación, teniendo un planificador local seguro, se abordará el planificador global PRM.

3.3.1 Planificación local con VFH

La evitación de obstáculos es uno de los principales problemas a resolver para las aplicaciones robóticas. Cualquier robot móvil posee algún tipo de técnica para prevenir las colisiones con los obstáculos del entorno. Estas técnicas van desde las más primitivas que al detectar un obstáculo detienen el robot por completo hasta sofisticados algoritmos que permiten al robot evitar los obstáculos en su movimiento. Estos últimos son mucho más complejos ya que permiten detectar el obstáculo y, además, realizar una cuantificación de las dimensiones de ese obstáculo. En esta sección se aborda uno de estos algoritmos llamado VFH.

3.3.1.1 Base teórica

El algoritmo de VFH es un algoritmo propuesto por *Johann Borenstein* y *Yoram Koren* en 1991 en la revista *IEEE Transactions on Robotics and Automation* en el artículo llamado ***The vector field histogram-fast obstacle avoidance for mobile robots*** [34]. El algoritmo propone la planificación local del movimiento en tiempo real utilizando una representación estadística del entorno del robot. Esta representación es denominada cuadrícula de histograma. VFH tiene en cuenta la dinámica y la forma de robot para devolver comandos de dirección específicos a este, produciendo rutas óptimas y evitando las colisiones con obstáculos desconocidos. Este algoritmo es uno de los más populares y de los que más se utilizan en robótica, a la par que el enfoque de la ventana dinámica (*The Dynamic Window Approach to Collision Avoidance*) [35].

El algoritmo de VFH se basa en trabajos previos conocidos como Virtual Force Field (VFF) [36]. VFF tenía un problema principal: la reducción drástica de datos que ocurre cuando las fuerzas individuales de repulsión de la cuadrícula de histograma son añadidas para calcular el vector de fuerza resultante F_r. Cientos de puntos son reducidos en un paso en solo dos elementos: dirección y magnitud de F_r. Por consecuencia, se pierde información detallada sobre la distribución local de obstáculos.

Por este motivo, se ha desarrollado el algoritmo de VFH que realiza una reducción de datos en dos pasos. En primer lugar, se construye una cuadrícula de histograma cartesiano bidimensional que es continuamente actualizada en tiempo real con medidas de los sensores del robot (tales como sonar o láser). El número de celdas de esta cuadrícula influirá en la computación y decisiones del algoritmo. En general, cuanto mayor número de celdas (mayor resolución) tenga el histograma más preciso será el algoritmo, pero en contrapartida, mayor será el coste computacional. La resolución que se escoja debe tener en cuenta que este algoritmo debe proporcionar una respuesta en tiempo real. Cada celda de este histograma posee un valor que será actualizado por la información recibida a través de los sensores y aplicando una función de distribución probabilística. Solo se actualizan las celdas que están a una determinada distancia del robot. Un histograma cartesiano bidimensional se puede ver en la Figura 3.23.

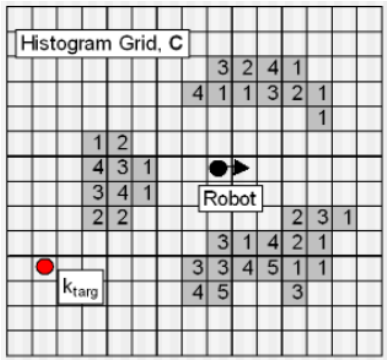


Figura 3.23: Histograma cartesiano bidimensional.

En el siguiente paso un histograma polar unidimensional es construido alrededor del robot en un momento dado. Este histograma comprende n sectores angulares de ancho α. Cada sector posee un valor h que representa la densidad de obstáculos polares (*polar obstacle density*) en la dirección que corresponde a ese sector. Para calcular el vector *polar obstacle density* se define el tamaño de la ventana activa, que establece las celdas que se deben considerar del histograma cartesiano para el cálculo del vector. Un ejemplo de este diagrama se puede ver en la Figura 3.24.

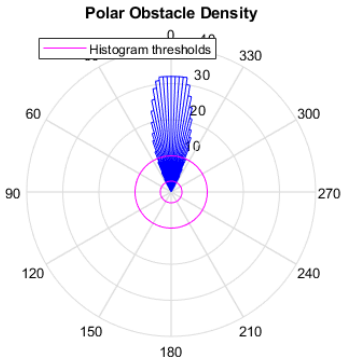


Figura 3.24: Histograma polar unidimensional.

Posteriormente, se establece un umbral con el cual se obtienen los sectores válidos y no válidos. Al unir varios sectores válidos se forma un valle candidato. El algoritmo deberá escoger el mejor valle candidato (el que se aproxime en la dirección del destino). Para escoger el mejor valle, el algoritmo distingue dos tipos de valles:

- ***Wide Valley***: número de sectores consecutivos válidos es igual o superior al parámetro del algoritmo S_{max}.
- ***Narrow Valley***: número de sectores consecutivos válidos es inferior al parámetro del algoritmo S_{max}.

Ahora, la dirección que tome el robot se calcula de forma diferente para cada tipo de valle.

- Para ***Wide Valley***: Se denomina K_n como el borde del valle más cercano al sector que está en la dirección de destino. El borde más alejado se establece como K_f y es igual a $K_f = K_n + S_{max}$. Por lo tanto, la dirección del robot se calculará mediante la fórmula 3.2

$$\theta = \frac{(K_n + K_f)}{2} \tag{3.2}$$

- Para ***Narrow Valley***: El borde más alejado K_f del valle es más pequeño que S_{max}. De este modo, se define K_n el primer sector libre del valle más próximo al sector que está en la dirección de destino y K_f como el último sector libre. El cálculo de la dirección se realiza con la misma fórmula 3.2.

Una vez calculada la dirección se calcula la velocidad del robot. Para ello, sea H_c el valor del vector *polar obstacle density* en la dirección calculada. La reducción de la velocidad se calculará mediante la fórmula 3.3.

$$V' = V_{max} \cdot (1 - \frac{H_{cc}}{H_m}) \tag{3.3}$$

En la fórmula 3.3 $H_{cc} = min(H_c, H_m)$ siendo H_m una constante empíricamente determinada que causa una suficiente reducción de la velocidad del robot.

La velocidad final del robot a aplicar será calculada mediante la fórmula 3.4. En la que Ω es la velocidad actual de giro del robot y Ω_{max} es la velocidad máxima de giro que puede desarrollar el robot.

$$V = V' \cdot (1 - \frac{\Omega}{\Omega_{max}}) \tag{3.4}$$

Una vez calculadas tanto la dirección como la velocidad sólo quedaría aplicar los comandos al robot.

En trabajos posteriores se desarrollaron mejoras como, por ejemplo, *VFH+* [37] que realiza una simplificación de la dinámica del robot: el robot sólo se mueve en rectas o arcos. Por otro lado, los obstáculos que bloquean una dirección dada también bloquean todas las posibles trayectorias a través de esa dirección. En el trabajo de *VFH** [38] se verifica el comando de dirección para minimizar el costo y las funciones heurísticas, demostrando que se pueden abordar situaciones problemáticas que en las versiones anteriores no se pueden manejar.

3.3.1.2 Implementación en MatLab

La *Robotics System Toolbox* de MatLab proporciona el algoritmo VFH a través de la clase denominada ***robotics.VectorFieldHistogram*** que posee un objeto llamado ***controllerVFH***. Este objeto requiere cuatro parámetros de entrada para el robot que son: ***RobotRadius***, ***SafetyDistance***, ***MinTurningRadius*** y ***DistanceLimits***. Además de estos parámetros que pertenecen a las propiedades del objeto, existen otras que se van a introducir a continuación:

- ***NumAngularSectors***: Número de sectores angulares en el histograma del campo vectorial. Define el número de ubicaciones utilizadas para crear los histogramas. Es un valor que una vez definido no se puede cambiar cuando el objeto se ha inicializado. Por defecto tiene un valor de 180.

- ***DistanceLimits***: Especifica el rango de distancias que se desea tener en cuenta para evitar obstáculos. Este límite se utiliza para ignorar las lecturas del sensor que se cruzan con partes del robot, inexactitudes del sensor a distancias cortas o el propio ruido del sensor. Se puede utilizar estos límites para no considerar todos los obstáculos en el rango completo del sensor. Es un vector de dos elementos, con valores especificados en metros y su valor por defecto se establece en [0.05 2].

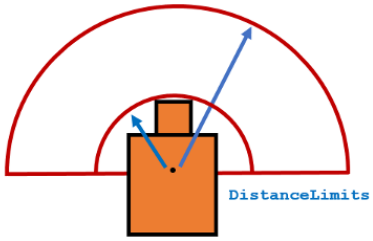


Figura 3.25: Ilustración de la propiedad ***DistanceLimits***

- ***RobotRadius***: Radio del robot. Especifica el círculo más pequeño que puede circunscribir el robot. Se utiliza para calcular la dirección sin obstáculos. Es un valor que se indica en metros y su valor predeterminado es 0.1.

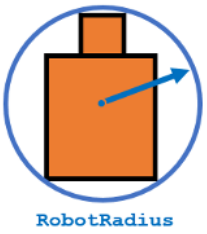


Figura 3.26: Ilustración de la propiedad ***RobotRadius***

- ***SafetyDistance***: Distancia de seguridad del robot. Esta es una distancia adicional que se añade a la propiedad ***RobotRadius*** e indica un área de seguridad alrededor del robot. Es un valor que se indica en metros y su valor por defecto es 0.1.

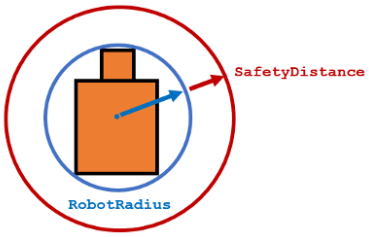


Figura 3.27: Ilustración de la propiedad ***SafetyDistance***

- ***MinTurningRadius***: Radio de giro mínimo para el robot que viaja a una velocidad deseada. Valor escalar por defecto 0.1.

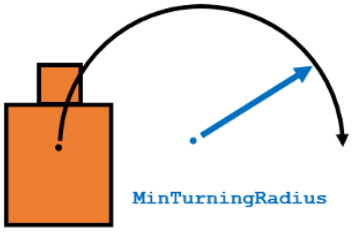


Figura 3.28: Ilustración de la propiedad ***MinTurningRadius***

- ***TargetDirectionWeight***: Peso de la función de coste para la dirección objetivo. Para seguir la dirección objetivo se debe establecer un valor superior a la suma de las propiedades ***CurrentDirectionWeight*** y ***PreviousDirectionWeight***. Para omitir el peso de la función de coste para la dirección objetivo se debe establecer un valor de 0. Valor por defecto 5.

- ***CurrentDirectionWeight***: Peso de la función de coste para la dirección actual. Valores altos en esta propiedad producen rutas eficientes. Para omitir esta propiedad se debe establecer un valor de 0. La propiedad tiene un valor por defecto de 2.

- ***PreviousDirectionWeight***: Peso de la función de coste para la dirección previa. Valores altos de esta propiedad producen rutas más suaves. Para omitir esta propiedad se debe establecer un valor de 0. La propiedad tiene un valor por defecto de 2.

- ***HistogramThreshold***: Umbrales para el cálculo del histograma binario. Los espacios ocupados son representados por zonas con valores de densidad mayores que el umbral máximo. Los valores menores al umbral mínimo son representados como espacios libres. Los valores que están dentro del rango se establecen en los valores del histograma previo, siendo por defecto espacio libre. El valor por defecto de este umbral es [3 10].

- ***UseLidarScan***: Propiedad que indica si se utiliza el objeto ***LIDARScan*** o no. Un valor *true* indica que se utiliza este objeto. Un valor *false* indica que no se utiliza el objeto ***LIDARScan***. El valor por defecto es *false*.

Una vez se conocen todas estas propiedades, el algoritmo selecciona varias direcciones en función del espacio libre y las posibles direcciones de movimiento. La función de coste, con los pesos de la dirección previa, actual y objetivo, calcula el coste de las diferentes direcciones. Por lo tanto, este algoritmo devuelve la dirección en la que existe un espacio libre de obstáculos con el coste mínimo. Una vez, que se conoce la dirección con el coste mínimo, se pueden enviar los comandos de movimiento al robot para que circule en esa dirección.

3.3.1.3 Resultados con el robot simulado

Para probar este algoritmo en el entorno de MatLab se ha creado el código que resuelve el problema de la planificación local con VFH desde cero. Con este algoritmo se ha utilizado el mapa que se había generado con la técnica SLAM en la sección 3.2.2.4, que se puede ver en la Figura 3.14.

Se han realizado una serie de retoques a este mapa con la herramienta GIMP [39], tales como: incorporación de paredes más gruesas, agregación de obstáculos en los pasillos para probar este algoritmo en el simulador, y, por último, se ha realizado una limpieza general del mapa. El resultado de estas operaciones se puede ver en la siguiente Figura 3.29.

Este mapa se ha generado para su utilización también con el simulador STDR. Para ello, se crea un archivo llamado ***"mi_mapa.yaml"***, en el que se deben indicar: nombre del archivo del mapa, resolución del mapa, el origen de coordenadas, el umbral de zonas ocupadas, el umbral de zonas libres, y si las zonas negras/blancas del mapa deben ser invertidas.

Configurado el entorno, el algoritmo que se ha implementado en MatLab realiza la conexión con el robot simulado, a continuación inicializa el objeto ***controllerVFH***, en el que posteriormente se indicarán todos los parámetros que se han visto anteriormente. Después de todo esto, el código entra en un bucle en el que se leen los datos tanto de la odometría del robot como de las medidas del sensor láser. Se llama al objeto VFH, al que se le pasan como argumento tanto los datos del láser como la dirección

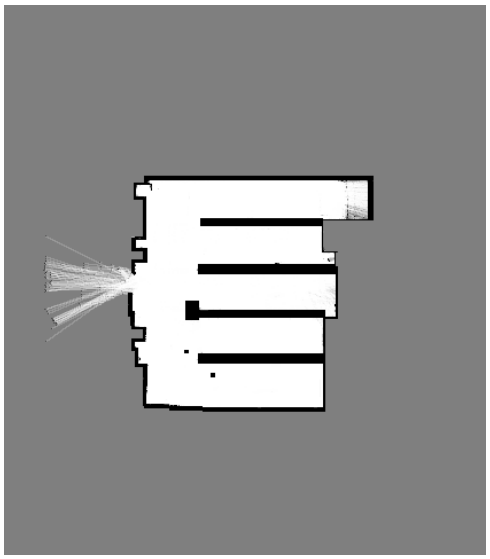


Figura 3.29: Mapa retocado del laboratorio de la Escuela Politécnica de la Universidad de Alcalá.

objetivo; lo que nos devuelve es la dirección con espacio libre con el menor coste posible. Conocida la dirección en la que se va a mover el robot, se establecen las velocidades lineal y angular. La velocidad lineal se establece constante, a 0.2 m/s si la dirección es válida y la velocidad angular se calcula con el método *exampleHelperComputeAngularVelocity*. Si la dirección no es válida la velocidad lineal del robot se establece nula y la velocidad angular en 0.5 rad/s. Esta velocidad angular no se establece nula para que el algoritmo pueda encontrar una nueva dirección al girar el robot sobre el mismo punto. Por último, se guardan las medidas de la odometría en un array para posteriormente realizar una visualización del recorrido del robot sobre el mapa.

Para probar cómo afectan todas las propiedades de este objeto sobre el algoritmo, se van a ajustar diferentes valores para cada una de las propiedades.

1. Variable ***targetDir***: Se configura esta variable a 0, dejando todos los demás parámetros del VFH a los valores por defecto. Esta variable, especificada en radianes, indica la dirección objetivo del robot. Se ha comprobado que el robot evita el obstáculo satisfactoriamente en la Figura 3.30.

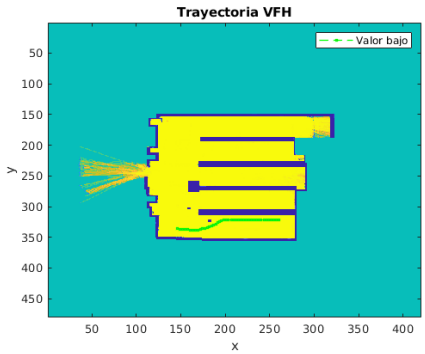


Figura 3.30: Recorrido del robot con la variable targetDir a 0.

2. Propiedad ***NumAngularSectors***: Para esta propiedad se indicaron valores bajos, medios y altos, dejando todos los demás parámetros por defecto. El resultado que se obtuvo se puede ver en la Figura 3.31, que nos indica que tanto con valores bajos, medios y altos de esta propiedad el robot llega a esquivar correctamente el obstáculo.

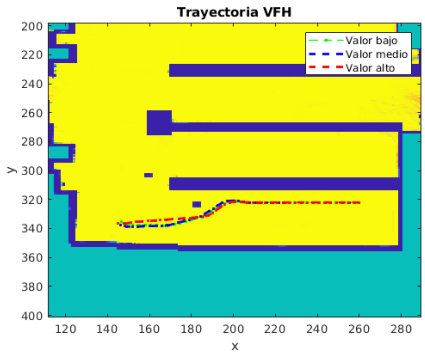


Figura 3.31: Recorrido del robot ajustando la propiedad NumAngularSectors.

3. Propiedad ***DistanceLimits***: Para esta propiedad también se indicaron valores bajos, medios y altos, dejando todos los demás parámetros por defecto, incluido ***NumAngularSectors***. El resultado que se obtuvo se puede ver en la Figura 3.32. Para valores altos, el algoritmo tiene en cuenta obstáculos que están muy alejados del robot, que como se puede observar, provoca que el robot no esquive correctamente el obstáculo. Algo similar pasa con valores bajos para esta propiedad, pero ahora los obstáculos no se tienen en cuenta hasta que el robot esté muy cerca de ellos, llegando a no tener posibilidades de esquivar el obstáculo a tiempo. Por el contrario, con valores medios de esta propiedad el robot evita el obstáculo sin problemas.

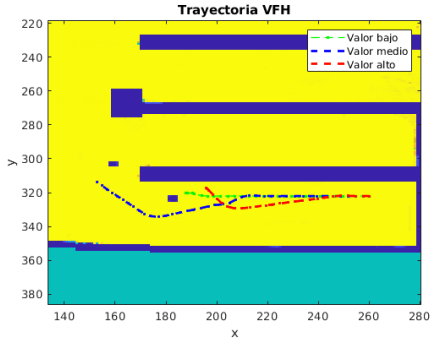


Figura 3.32: Recorrido del robot ajustando la propiedad DistanceLimits a valores bajos, medios y altos.

4. Propiedad ***RobotRadius***: En esta propiedad se indica el valor real del radio del robot, que es 0.15 metros. Si se indican valores más pequeños, el algoritmo podrá obtener direcciones por zonas estrechas en las que piensa que podrá pasar el robot, que en realidad no es así aumentando de este modo la probabilidad de que haya colisión con el obstáculo. Para valores más grande del radio real del robot, el algoritmo visualiza un robot más grande de lo que es en realidad realizando movimientos de evitación de obstáculos antes de lo necesario. El ajuste de los diferentes valores se puede ver en la siguiente Figura 3.33.

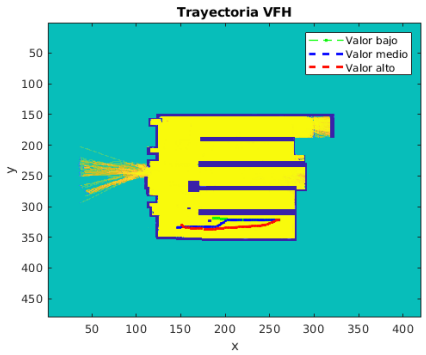


Figura 3.33: Recorrido del robot ajustando la propiedad RobotRadius a valores bajos, medios y altos.

5. Propiedad ***SafetyDistance***: esta propiedad actúa de forma análoga a la propiedad anterior. Aumenta o disminuye el espacio de seguridad circundante al radio del robot que debe estar libre de obstáculos. Los resultados se pueden ver en la Figura 3.34.

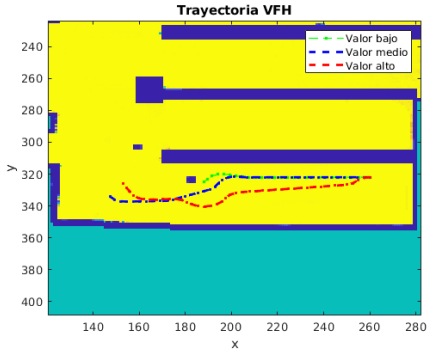


Figura 3.34: Recorrido del robot ajustando la propiedad SafetyDistance a valores bajos, medios y altos.

6. Propiedad ***MinTurningRadius***: Conociendo el significado de esta propiedad, se ha comprobado que valores muy altos hacen que el algoritmo funcione de forma indeseada, ya que indican el radio mínimo que debe girar el robot con la velocidad actual. Esto puede provocar que no haya radio de giro suficiente para el robot. Valores bajos funcionan muy bien, llegando a esquivar correctamente el obstáculo. En la Figura 3.35, se observa que para los tres valores insertados, el robot evita el obstáculo.

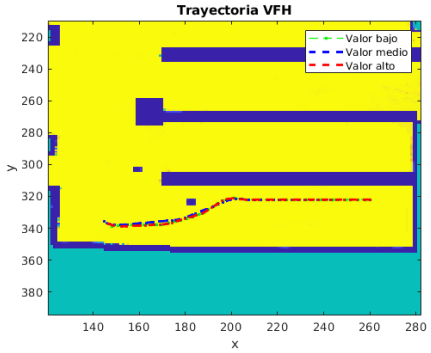


Figura 3.35: Recorrido del robot ajustando la propiedad MinTurningRadius a valores bajos, medios y altos.

7. Propiedad ***TargetDirectionWeight***: Para poder seguir la dirección objetivo, esta propiedad debe tener un valor superior a la suma de los pesos de la dirección actual y la dirección previa. Teniendo esta limitación en cuenta, se ajustan los valores a bajos, medios y altos. La Figura 3.36 refleja el resultado obtenido, en el que los tres valores esquivan el obstáculo. También se puede ver que para valores bajos, el robot simulado se quedó parado, sin motivo alguno. Este comportamiento no volvió a repetir en simulaciones posteriores.

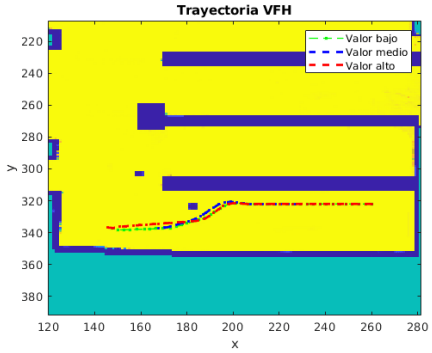


Figura 3.36: Recorrido del robot ajustando la propiedad TargetDirectionWeight a valores bajos, medios y altos.

8. Propiedad ***CurrentDirectionWeight***: Peso de la función de coste para mover el robot en la misma orientación de la dirección actual. Valores altos de esta propiedad generan trayectorias optimizadas. Aún así, en la Figura 3.37 se puede ver que tanto para valores bajos, medios y altos el robot evita el obstáculo.

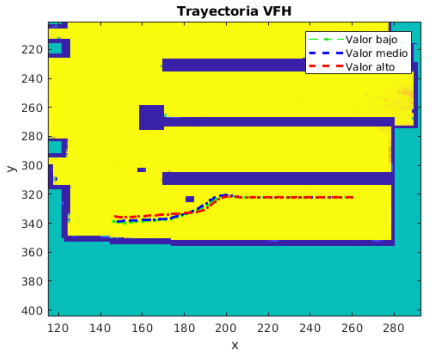


Figura 3.37: Recorrido del robot ajustando la propiedad CurrentDirectionWeight a valores bajos, medios y altos.

9. Propiedad ***PreviousDirectionWeight***: Altos valores para esta propiedad producen trayectorias suaves. En la Figura 3.38, por el contrario, se puede ver que para un valor alto establecido de 10, el robot colisiona con el obstáculo. Esto se podría interpretar como que el algoritmo quisiera suavizar demasiado el recorrido quedando atrapado en el obstáculo.

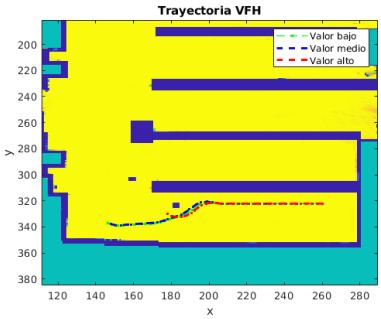


Figura 3.38: Recorrido del robot ajustando la propiedad PreviousDirectionWeight a valores bajos, medios y altos.

10. Propiedad ***HistrogramThresholds***: Umbrales que utiliza el algoritmo para computar el histograma binario a partir de la densidad polar de obstáculos. Valores de densidad polar de obstáculos superior que el umbral superior se representan como espacios ocupados (1), mientras que valores inferiores al umbral inferior son representados como espacios libres (0). Los valores que quedan entre medias son configurados a los valores en el anterior histograma binario, donde por defecto son espacios libres (0). Se probaron diferentes rangos, obteniendo el resultado del recorrido que se ve en la Figura 3.39.

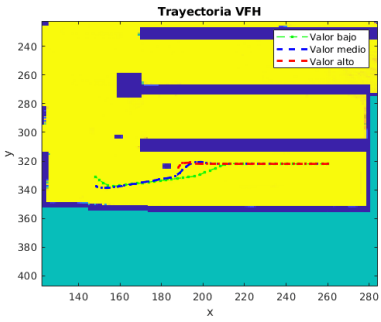


Figura 3.39: Recorrido del robot ajustando la propiedad HistrogramThresholds a valores bajos, medios y altos.

Finalmente, los valores óptimos que se han establecido en las propiedades del objeto ***controllerVFH***, se pueden ver resumidos en la Tabla 3.2.

Propiedad	Valor óptimo
targetDir	0
NumAngularSectors	270
DistanceLimits	[0.25, 3]
RobotRadius	0.15
SafetyDistance	0.20
MinTurningRadius	0.15
TargetDirectionWeight	5
CurrentDirectionWeight	2
PreviousDirectionWeight	2
HistogramThresholds	[3, 10]

Tabla 3.2: Valores óptimos para las propiedades del objeto ***controllerVFH***

Con estos valores óptimos se realizó otro recorrido en simulación, que se visualiza en la Figura 3.40, obteniendo un resultado satisfactorio, consiguiendo que el robot evite el obstáculo.

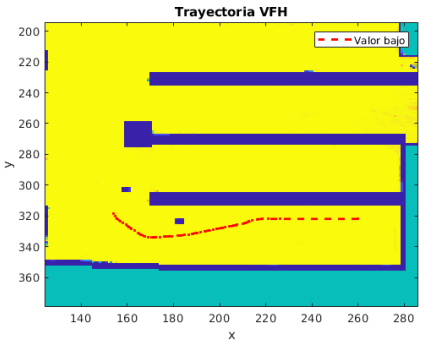


Figura 3.40: Recorrido del robot simulado con los valores óptimos de VFH

3.3.1.4 Resultados con el robot real

Una vez comprobado que en simulación el algoritmo funciona correctamente, se han procedido a realizar las pruebas con el robot real. Los parámetros utilizados han sido los mismos que los que se han utilizado en simulación, presentados en la Tabla 3.2. El resultado se puede ver a continuación en la Figura 3.41.

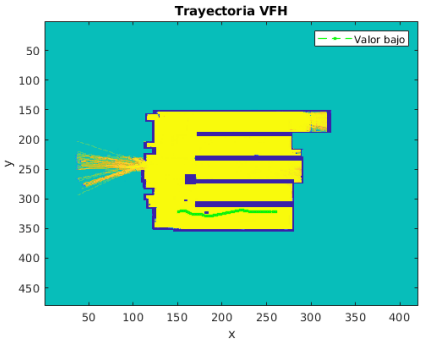


Figura 3.41: Recorrido del robot real con los valores óptimos de VFH

Se ha puesto un obstáculo en la misma posición, aproximadamente, que se indica en el mapa, viendo que el robot lo esquiva satisfactoriamente. Para probar este algoritmo con el robot real, lo único que se debe hacer es, como siempre, cambiar los nombres de los topics ROS a los que se debe suscribir/publicar el algoritmo.

3.3.1.5 Algoritmo de planificación local VFH con PurePursuit

El algoritmo PurePursuit [40] es un algoritmo de seguimiento de rutas que calcula la velocidad angular que mueve el robot desde la posición actual hasta un punto localizado en frente del robot. La velocidad lineal se puede cambiar en cualquier punto debido a que se considera constante. Este algoritmo puede modelarse como una persecución constante de un punto que está delante del robot. Este controlador permite ir obteniendo sobre la marcha la dirección objetivo para llegar al siguiente punto objetivo en la ruta que se ha establecido previamente por el usuario. Combinando este controlador que devuelve la dirección objetivo y el algoritmo de VFH que determina si es capaz de ir en esa dirección sin colisionar con algún obstáculo.

Se podría pensar que se ha obtenido un planificador global al unir estos dos algoritmos, pero esto no es así. El sistema sigue siendo un planificador local al que sobre la marcha se le van indicando las direcciones objetivo. Dado que la ruta global no se calcula en ningún momento, no podría considerarse que se ha conseguido un planificador global, a pesar de que el robot sea capaz de alcanzar el punto objetivo.

Para entender el controlador PurePursuit se debe comprender su marco de coordenadas de referencia. A este controlador se le pasa como entrada la posición [x y] de un punto de la ruta, que es utilizada para calcular el comando de la velocidad. Por otro lado, la posición del robot es una entrada con la posición [x y] y la orientación positiva θ del robot en radianes partiendo desde el eje x. Los valores positivos de x e y se encuentran en el primer cuadrante, como se puede ver en la Figura 3.42.

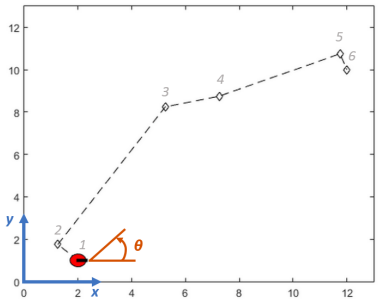


Figura 3.42: Coordenadas de referencia del controlador PurePursuit

Una propiedad importante de este algoritmo es *LookAheadDistance* que indica al robot la distancia de frente en la que tiene que buscar desde la posición actual para calcular los comandos de velocidad angular. Un valor pequeño de esta propiedad hará que el robot se mueva rápidamente hacia la ruta. Se podría elegir un valor más grande para esta propiedad con el fin de reducir las oscilaciones a lo largo del camino. Sin embargo, esto podría resultar en curvaturas más grandes cerca de las esquinas entre dos puntos de la ruta.

Las limitaciones de este controlador que se pueden observar son: no puede seguir el camino directo entre los puntos de la ruta establecida y, por otro lado, el algoritmo no estabiliza el robot en un punto. Se debe indicar un umbral de distancia para que indique al robot que ha llegado al objetivo.

Los resultados obtenidos para el robot simulado y el robot real se puede observar en las Figuras 3.43 y 3.44.

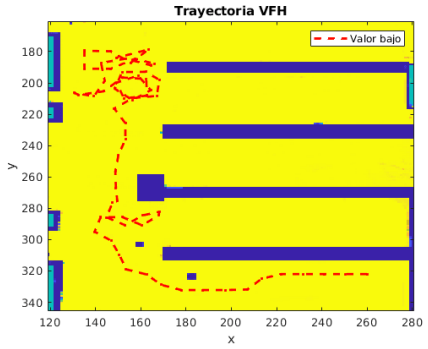


Figura 3.43: Resultado de PurePursuit con VFH en robot simulado

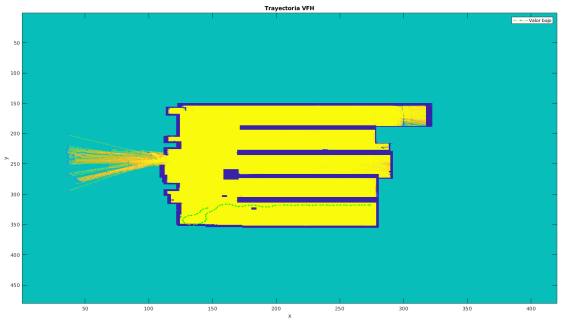


Figura 3.44: Resultado de PurePursuit con VFH en robot real

En la Figura 3.44 se observa cómo el robot sufre una desviación de la trayectoria en la parte final, pero finalmente consigue redireccionar la ruta hacia el punto objetivo. También se observa cómo en una zona

la trayectoria del robot se superpone a la pared inferior del mapa, lo cual puede deberse a las diferencias entre el mapa generado y el mapa real por el que navega el robot.

En la Figura 3.43 se puede observar una ruta mayor indicando puntos en los pasillos del mapa. Se puede observar que el robot realiza el recorrido correctamente pero al alcanzar el punto objetivo, no se detiene sino que realiza recorridos alrededor de ese punto lo que indica que no se ha establecido correctamente el umbral para decir al robot que ha llegado a su destino.

3.3.2 Planificación global con PRM

El algoritmo de planificación PRM fue presentado en el artículo llamado ***Probabilistic Roadmaps for Path Planning in High-Dimensional Configuration Spaces*** [41]. Este artículo fue publicado en 1996 en la revista *IEEE Transactions on robotics and automation* por ***Lydia E. Kavraki***, ***Petr Svestka***, ***Jean-Claude Latombe*** y ***Mark H. Overmars***. La idea principal del algoritmo PRM consiste en obtener muestras aleatorias del espacio de configuración del robot. Estas muestras se prueban para ver si están en el espacio libre de obstáculos y una vez comprobado esto, se utiliza un planificador local para intentar conectar las diferentes configuraciones cercanas.

Este método se desarrolla en dos fases: una fase de aprendizaje y otra fase de consulta. En la fase de aprendizaje, se construye el mapa de ruta probabilística y se almacena como un grafo cuyos nodos corresponden a configuraciones libres de colisiones. Los bordes corresponden a caminos factibles entre las diferentes configuraciones. En la fase de consulta, las configuraciones de inicio y objetivo son conectadas al grafo, y, la ruta se obtiene mediante una consulta de la ruta más corta mediante el algoritmo de Dijkstra. El algoritmo de Dijkstra consiste en ir explorando todos los caminos más cortos que parten de un vértice origen y que llevan a todos los demás vértices.

3.3.2.1 Implementación en MatLab

La *Robotics System Toolbox* de MatLab proporciona el algoritmo PRM a través del objeto ***mobileRobotPRM***. Este objeto requiere dos argumentos de entrada que son: ***map*** y ***numnodes***.

- ***map***: Representación del mapa mediante un objeto *binaryOccupancyMap*. Este objeto es una cuadrícula con valores binarios en los que se indican zonas con obstáculos mediante *true* (1) y zonas libres mediante *false* (0).
- ***numnodes***: Número máximo de nodos en la ruta, especificado como escalar. Al aumentar este valor aumenta la complejidad y el tiempo de cálculo para este planificador global.

Por otro lado, las propiedades de este objeto se recogen a continuación:

- ***ConnectionDistance***: Indica la distancia máxima entre dos nodos conectados entre sí. Se especifica como un par separado por comas que consiste en 'ConnectionDistance' y un escalar en metros. Esta propiedad controla si los nodos se conectan en función de la distancia que los separa. Los nodos se conectan solamente si no hay obstáculos directos en la ruta. Decrementando el valor de esta propiedad, el número de conexiones también baja lo que afecta además en una bajada de la complejidad y el tiempo de cómputo.
- ***Map***: esta propiedad representa lo mismo que el argumento de entrada.
- ***NumNodes***: Al igual que la propiedad anterior, esta propiedad ya se ha explicado en los argumentos de entrada del objeto ***mobileRobotPRM***.

Este objeto también tiene tres funciones asociadas que tienen la siguiente funcionalidad:

- ***findpath***: Esta función recibe como argumentos el objeto PRM, la posición inicial y la posición objetivo. Con estos valores encuentra una ruta sin obstáculos entre la posición inicial y la posición objetivo. Es un objeto de ruta que contiene una red de puntos conectados.
- ***show***: Esta función recibe como argumento el objeto PRM y con ello, muestra el mapa junto con la ruta. Si no existe ninguna ruta se llama a la función ***update***.
- ***update***: Esta función recibe como parámetro el objeto PRM. Si se llama una primera vez, crea una ruta creando el objeto *mobileRobotPRM*. En llamadas posteriores, recrea la ruta haciendo un remuestreo del mapa. Esta función utiliza las propiedades ***Map***, ***NumNodes*** y ***ConnectionDistance*** especificadas en el objeto PRM, que le sirven para crear la ruta.

Especificando y utilizando todas estas propiedades y funciones se puede llegar a obtener soluciones tales como se indican en las siguientes figuras 3.45 y 3.46. Se puede observar como en la Figura 3.45 hay un número menor de nodos, lo que desemboca en menor complejidad y menor tiempo de cómputo en comparación con la Figura 3.46. Pero la ventaja de la configuración de la Figura 3.46 es que el robot tiene más opciones para encontrar una ruta, prácticamente, hasta cualquier posición del mapa. En contraposición, la configuración de la Figura 3.45 visualiza zonas del mapa vacías de nodos, lo que dificultaría la construcción de la ruta hasta esas posiciones objetivo.

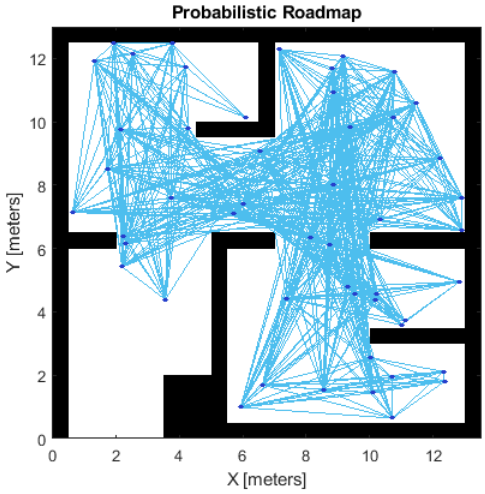


Figura 3.45: Hoja de ruta del planificador global PRM con bajo número de nodos

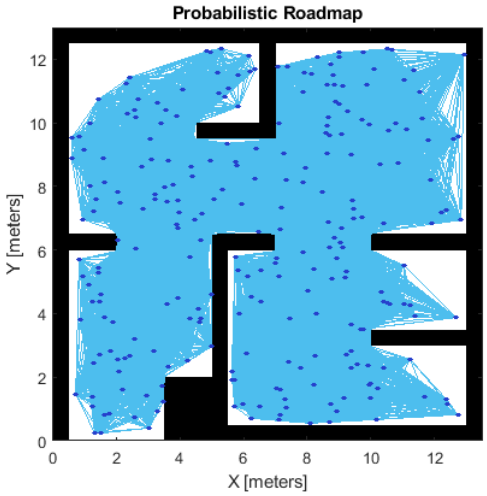


Figura 3.46: Hoja de ruta del planificador global PRM con alto número de nodos

La implementación de este algoritmo en MatLab se realiza desde cero, partiendo de la documentación que se proporciona en la página oficial de MathWorks para el planificador global PRM. Como siempre, en primer lugar se realiza la programación de la conexión con los topics de ROS. A continuación, el mapa que se va a utilizar en la simulación del entorno, se transforma en un objeto *binaryOccupancyMap*. Posteriormente, este mapa es pasado como argumento al objeto *mobileRobotPRM* junto con un número inicial de nodos.

Para construir la ruta se debe indicar la posición actual del robot y la posición objetivo que se quiere alcanzar. Además, se especifica ***ConnectionDistance*** en una primera instancia este valor a 1 metro. Después de realizar todos estos pasos, se llama a la función ***findpath***, que será la encargada de encontrar la ruta hasta el punto objetivo. Si, por el contrario esta función no encuentra una ruta, se ha implementado

un algoritmo que aumenta el número de nodos hasta que esta función encuentre una ruta. Este trozo de algoritmo se puede ver en el siguiente Listado 3.18:

Listing 3.18: Código para aumentar el número de nodos hasta encontrar una ruta

```
while isempty(path)
    prm.NumNodes = prm.NumNodes + 10;
    update(prm);
    path = findpath(prm, startLocation, endLocation);
end
```

Por último, una vez que el algoritmo ha encontrado una ruta, se entra en un bucle *while*, el cual se ejecuta hasta llegar a la posición objetivo con una distancia máxima que puede estar alejado el robot del punto objetivo.

Cabe mencionar que para controlar el robot se hace uso del controlador PurePursuit que se había explicado en la sección 3.3.1.5. Para realizar el control del robot se debe calcular la posición actual del robot (x, y, orientación). Estos parámetros se pasan a la función *step* junto con el objeto del controlador PurePursuit. Esta función nos devuelve la velocidad lineal y la velocidad angular. Estas velocidades se envían al robot, con lo que se mueve. Por último se calcula la distancia que existe entre la posición actual del robot y la posición objetivo. Si se cumple la distancia máxima se sale del bucle *while* y se para el robot.

3.3.2.2 Resultados con el robot simulado

Se han realizado diferentes pruebas para ver el comportamiento del algoritmo. Se han indicado diferentes valores a la propiedad ***ConnectionDistance***. Los resultados se pueden ver en las siguientes figuras 3.48, 3.47 y 3.49.

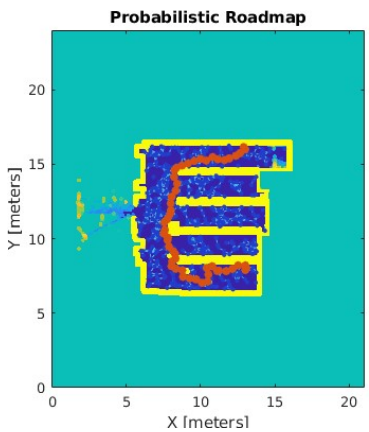


Figura 3.47: ***ConnectionDistance*** con un valor de 0.5 m.

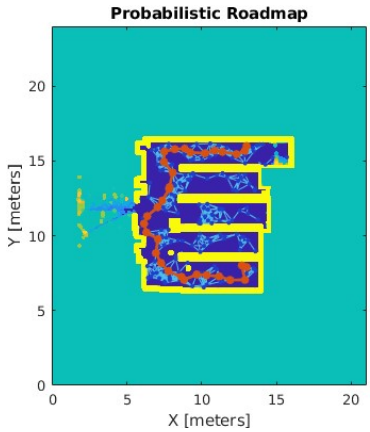


Figura 3.48: ***ConnectionDistance*** con un valor de 1 m.

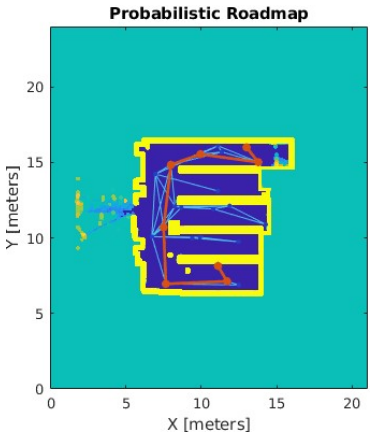


Figura 3.49: ***ConnectionDistance*** con un valor de 5 m.

Así, se puede ver que con un valor alto para esta propiedad la ruta que se construye utiliza muchos menos nodos resultando en rutas abruptas y poco óptimas. Por el contrario, indicando un valor muy pequeño, resulta en una ruta más suave, pero con un coste de tiempo de cómputo mucho mayor. Por lo que se ha visto haciendo estas pruebas, un valor óptimo para esta propiedad es 1 metro, debido a que el tiempo de cómputo no es muy elevado y las rutas que se llegan a construir son más que aceptables.

Viendo estas figuras, cabe mencionar que se puede apreciar que se han aumentado las zonas ocupadas del mapa, dando así un mayor margen de seguridad a la hora de navegar al robot. Esto se ha realizado con la función *inflate*.

3.3.2.3 Resultados con el robot real

Los parámetros obtenidos mediante simulación son válidos para trabajar con el robot real. Pero se ha observado que las rutas generadas con el algoritmo configurado en simulación (dando un valor de NumNodes igual a 10 e incrementándolo iterativamente hasta obtener una ruta válida) en algunos casos eran demasiado bruscas para el robot real.

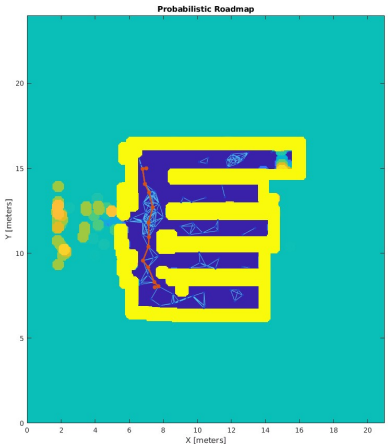


Figura 3.50: Planificador global PRM con robot real.

Capítulo 4

Desarrollo e Implementación en Robotics-Academy

"Nunca consideres el estudio como una obligación, sino como una oportunidad para penetrar en el bello y maravilloso mundo del saber."
Albert Einstein

En este capítulo se proporciona una descripción del diseño de la plataforma Robotics-Academy y su instalación. A continuación, se describe el proceso que se ha seguido para implementar las diferentes funcionalidades que se tenían en la plataforma MatLab-ROS tales como, simulador, teleoperación, etc. Por último, se describen las plantillas programadas para los ejercicios de mapeado y localización con el robot AmigoBot sobre esta plataforma con la intención de compararlo con las equivalentes con MatLab-ROS explicadas en el capítulo 3.

4.1 Instalación

La instalación de la plataforma Robotics-Academy se ha simplificado en gran medida respecto a las versiones anteriores gracias a RADI, que ya tiene dentro Ubuntu 20.04, ROS-Noetic y Gazebo preinstalados. Los usuarios de esta nueva versión sólo deben seguir tres pasos:

1. Instalar Docker en sus equipos. El proceso de instalación se puede encontrar en la siguiente página web: *https://docs.docker.com/get-docker/*.

2. Extraer la distribución actual del RADI. Esto se realiza con el siguiente comando en la terminal.

Listing 4.1: Comando para extraer la distribución actual de RADI

```
1 docker pull jderobot/robotics-academy:latest
```

3. Ejecutar el RADI utilizando las instrucciones de ejecución específicas de cada ejercicio de la plataforma.

De esta sencilla forma, se puede empezar a trabajar con esta plataforma de enseñanza robótica.

4.2 Diseño de los ejercicios

En esta sección se describe tanto la estructura general de Robotics-Academy como la estructura base que tienen los ejercicios proporcionados por esta plataforma. También, se realiza una breve explicación de cómo ejecutar los diferentes ejercicios dentro de esta plataforma.

La mayoría de los ejercicios que se plantean en esta plataforma contienen una estructura mediante una plantilla Web. La ejecución de los ejercicios con plantillas web es la última versión y la forma preferida para ejecutar los ejercicios. Estas plantillas proporcionan una página web en el navegador con una interfaz gráfica sencilla, atractiva y cómoda para programar las soluciones de estos ejercicios.

Los ejercicios poseen una parte principal que es el HAL. Este HAL realiza la conexión con el entorno ROS para leer y escribir datos de los sensores y actuadores. Para realizar la conexión, hace uso de una serie de archivos escritos en lenguaje Python y utilizando la librería *rospy*. Esta librería proporciona una forma rápida de acceder a los Topics, Servicios y Parámetros de ROS. Una posible estructura del HAL se puede ver en la Figura 4.1:

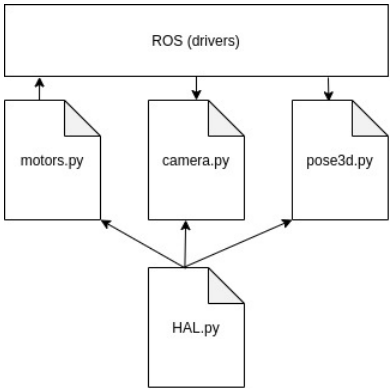


Figura 4.1: Conexión entre el HAL y los Topics de ROS.

La estructura de las plantillas web tiene una estructura similar con dos archivos principales que son: ***exercice.py*** y ***exercice.html*** que interactúan entre sí en ejecución mediante *WebSockets*, uno ejecutando dentro del RADI y el otro ejecutándose desde el navegador web. El archivo ***exercice.py*** se encarga de establecer la conexión con ***exercice.html*** mediante el protocolo de red *WebSocket*. Gracias a este protocolo se intercambian datos entre estos dos archivos de forma bidireccional al mismo tiempo, proporcionando una gran ventaja: un acceso más rápido a los datos.

Una vez establecida la conexión, en *exercice.py* vienen definidas las funciones que se podrán utilizar por el usuario en el navegador; funciones tales como la lectura de la odometría, lectura de los diversos sensores, escritura de las velocidades del robot, etc.

El archivo ***exercice.html*** se encarga de visualizar la página web acorde al diseño de cada ejercicio. También, recoge los eventos que ocurren dentro del navegador y se encarga de transferirlos mediante la comunicación establecida.

A continuación, en la Figura 4.2 se puede ver la estructura general de las plantillas web.

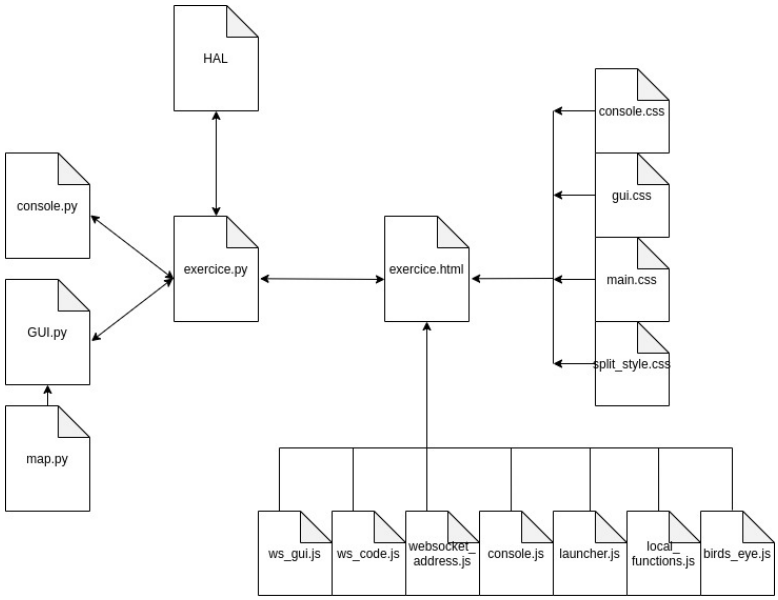


Figura 4.2: Estructura de las plantillas Web.

Los archivos vistos en la figura 4.2, se introducirán brevemente a continuación:

- El archivo ***ws_gui.js*** es el encargado de procesar los comandos relacionados con el apartado de la visualización del GUI.
- El archivo ***ws_code.js*** se encarga de enviar los comandos desde *exercice.html*. Estos comandos pueden ser cargar código desde un archivo, parar la ejecución del código, reiniciar la simulación, etc.
- El archivo ***websocket_address.js*** contiene la dirección IP de la máquina en la que se ejecuta el archivo *exercice.py*, es decir del contenedor RADI en ejecución.
- El archivo ***birds_eye.js*** contiene el código necesario para actualizar la simulación del ejercicio según los datos recibidos desde el archivo *ws_gui.js*.
- Los archivos ***console.css***, ***gui.css***, ***main.css*** y ***split_style.css*** sirven para organizar la presentación y el aspecto del documento HyperText Markup Language (HTML) (la página web del ejercicio). Estos archivos ubican en la página todos los elementos presentes en el sitio que ha elegido el desarrollador. Con la ayuda de todos estos archivos se consigue el diseño de la página web que se puede ver en la figura 4.3.

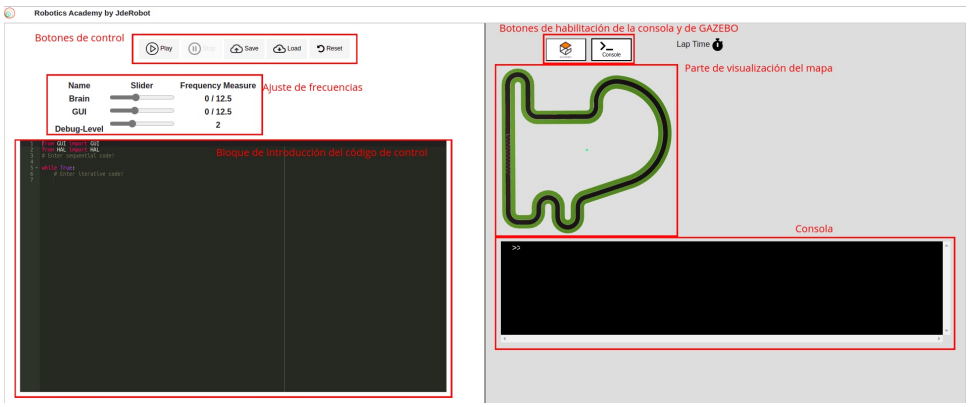


Figura 4.3: Estructura de la página web Robotics-Academy 3.1

En esta página web existen diferentes áreas bien diferenciadas:

- Un primer área con botones de control, desde los cuáles se puede ejecutar el código fuente del usuario con el botón *Play*. A su vez, también se puede parar la ejecución del código con el botón *Stop*. Además, existe la posibilidad de guardar el código de control desarrollado por el usuario con el botón *Save*. Una vez guardado el archivo, se puede cargar a la página web ese mismo archivo con el botón *Load*.
- Un segundo área en el que se realiza el control de las frecuencias de actualización tanto del cerebro de la aplicación como del GUI.
- El tercer área es el bloque principal de la programación del control del robot. Aquí es donde el usuario tiene que introducir el código fuente correspondiente para la correcta resolución del control del robot.
- En la parte derecha de la página web, en primer lugar se tienen los botones de habilitación, tanto de la consola como del visor del simulador Gazebo. Cada vez que se pulse uno de estos botones, se abrirá el área correspondiente al botón pulsado. Por ejemplo, en la figura 4.3, se ha pulsado el botón de la consola habilitando su vista, mientras que, el botón del simulador Gazebo se mantuvo sin pulsar. Si se volviese a pulsar sobre el botón de la consola se deshabilitaría su vista.
- Debajo de los botones se presenta el área de la visualización del simulador de la aplicación. Al realizar el código de control se podrá ver la simulación del robot en este área.
- Por último, debajo del área del mapa, se podrán ver tanto la consola como el simulador Gazebo u otras vistas, dependiendo de si se han habilitado o no.

Para proceder con la ejecución de uno de los ejercicios de la plataforma se deben seguir los siguientes pasos:

1. En primer lugar, se debe iniciar un nuevo contenedor docker de la imagen de RADI y mantenerla ejecutando en segundo plano.

Listing 4.2: Comando para inicializar el RADI

```
1 docker run --rm -it -p 8000:8000 -p 2303:2303 -p 1905:1905 -p 8765:8765 -p 6080:6080 -p
      1108:1108 jderobot/robotics-academy:3.1.4 ./start.sh
```

2. En el equipo local se debe acceder a la dirección 127.0.0.1:8000 en el browser y se elige el ejercicio deseado.
3. Se debe esperar a que el botón Conectar se vuelva verde y muestra *Connected*. Entonces, se debe pulsar el botón *Launch* y esperar hasta que aparezca una alerta con el mensaje *Connection Established*.
4. Una vez establecida la conexión, se puede utilizar el ejercicio.

4.2.1 Actualización a nueva versión Robotics-Academy 3.2

La plataforma ahora luce una nueva interfaz del browser. Se han organizado de mejor manera los botones de control, los ajustes de frecuencia y los botones específicos de los ejercicios, tales como la consola, el simulador Gazebo, el botón de mapeado, etc. La nueva interfaz se puede visualizar en la figura 4.4.

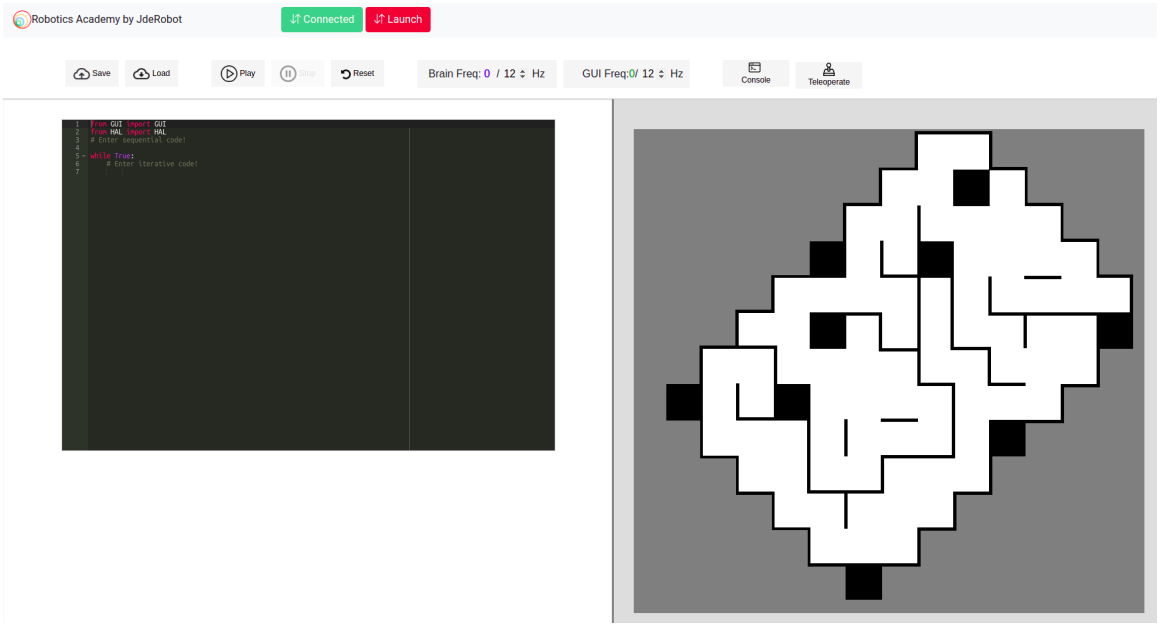


Figura 4.4: Nueva interfaz web de la plataforma Robotics-Academy 3.2

En esta figura, se puede ver que se han añadido dos botones en la parte superior de la ventana:

- **Connected**
- **Launch**

El primero de ellos, no es un botón en sí, sino que es más bien un panel informativo. Se encarga de informar al usuario de la conexión entre el ejercicio y el **manager.py**. Este archivo se explicará en mayor detalle a continuación en la sección 4.2.2.

El segundo, **Launch**, sí que es un botón, y, una vez se haya pulsado, se encarga de establecer al conexión con la estructura propia del ejercicio en cuestión.

En cuanto a las partes restantes de la interfaz web, la plataforma ha mantenido la estructura que se venía viendo en la versión anterior. La introducción de código del usuario en la parte izquierda de la página mientras que en la parte derecha se mantiene las herramientas de simulación, consola, etc.

4.2.2 Conexión y estructura

En esta nueva versión los ejercicios de Robotics-Academy se sirven al browser a través de un servidor de ***Django***[42]. Django es un entorno de aplicaciones web gratuito y de código abierto escrito en Python. Un entorno web es un conjunto de componentes que te ayudan a desarrollar sitios web más fácil y rápidamente. A continuación se puede ver la Figura 4.5 en la que se muestra el servidor Django con las entradas de los diferentes ejercicios existentes de Robotics-Academy.

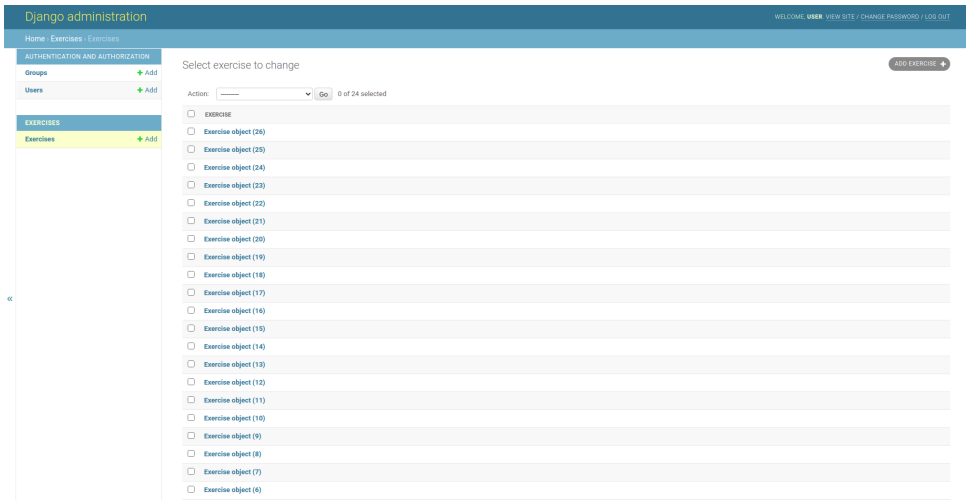


Figura 4.5: Servidor Django de los ejercicios de Robotics-Academy.

Estas entradas, posteriormente, se visualizarán por el usuario de la siguiente manera, vista en la Figura 4.6.

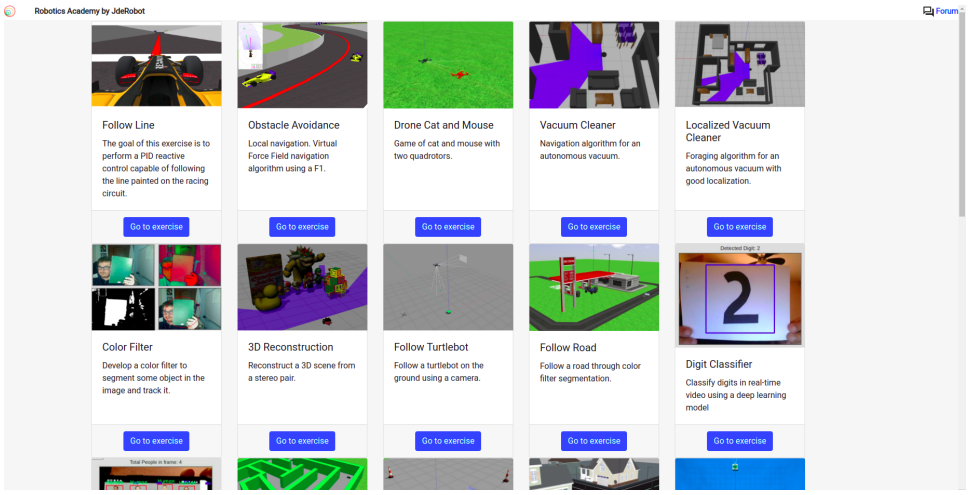


Figura 4.6: Visualización de los ejercicios en el servidor Django.

El proceso que se sigue de conexión de los ejercicios en esta reciente versión de la plataforma es el siguiente:

1. Se realiza una primera conexión del WebSocket del archivo ***manager.py*** con el ***exercise.html*** del ejercicio en cuestión elegido en el servidor de Django.

2. Una vez establecida la conexión mediante el botón ***Launch*** el archivo manager.py recibe el mensaje 'open', y, entonces, comienza a lanzar el ejercicio.

3. Este manager.py utiliza otro archivo llamado ***instructions.json***, en el cuál busca obtener la ruta del script que inicializa el simulador STDR. También, busca la ruta del archivo ***exercise.py*** y lo ejecuta.

4. Al ejecutarse el archivo ***exercise.py*** manda ejecutar el archivo ***gui.py***. Ambos archivos crean un WebSocket para comunicarse con el exercise.html.

5. Una vez los servidores de WebSocket están creados, exercise.py y gui.py notifican al manager.py que continúe con la inicialización del ejercicio. Esta notificación se realiza mediante otros dos archivos ***ws_code.log*** y ***ws_gui.log***.

6. Después de esto, el manager.py envía un mensaje al navegador web para que establezca las conexiones con ambos WebSocket, tanto de exercise.py como gui.py.

7. El ejercicio se considera inicializado y la página debería de mostrarse funcional.

Por lo tanto, los nuevos archivos principales introducidos en la estructura de la plataforma son manager.py e instructions.json. El archivo manager.py se encarga de distinguir que ejercicio se desea ejecutar en cada sesión. Una vez sabe que ejercicio es, lanza todo lo necesario relacionado que ese ejercicio. Aquí es donde el archivo instructions.json le ayuda, ya que indica las rutas del sistema donde debe mirar para proceder con el lanzamiento de las herramientas correspondientes. El archivo manager.py también se encarga de configurar todo lo necesario para la consola de la interfaz web. Por último, se hace cargo del procesamiento del código que introduce el usuario.

Se han mencionado anteriormente dos archivos que son ws_code.log y ws_gui.log. Como se ha dicho son creados por exercise.py y gui.py, respectivamente. El archivo ws_code.log en su interior tiene el siguiente mensaje:

websocket_code=ready

Mientras que el archivo ws_gui.log tiene este otro mensaje:

websocket_gui=ready

El manager.py se encarga de leer estos archivos una vez son creados y proseguir con al ejecución de los ejercicios. Sin estos mensajes la conexión con los ejercicios no se podría realizar. Una vez se inicia de nuevo el manager.py, una de las primeras tareas que realiza es borrar estos archivos para una nueva ejecución de los ejercicios.

4.3 Desarrollo

En esta sección se describe el proceso de implementación en la página web del simulador, de la teleoperación, etc. que se han empleado en los dos ejercicios creados. También se describe el proceso de configuración que se ha llevado para trabajar tanto con el robot simulado como con el robot real dentro de la plataforma. Dentro del navegador, en vez del simulador Gazebo se va a trabajar con el simulador STDR, que se ha visto en el capítulo 3.

4.3.1 Desarrollo del HAL

Como se ha visto, se va a trabajar con el simulador STDR. Este simulador crea unos topics ROS por los cuales se transmite la información de las medidas de los sensores y se permite la escritura en los actuadores. Por lo tanto, es necesario conseguir que el HAL de la plataforma Robotics-Academy, se conecte correctamente a estos topics de ROS para interactuar con el robot simulado.

Para el robot AmigoBot el HAL va a tener cuatro archivos Python que van a establecer la conexión con los topics ROS creados por el simulador STDR. Para ver los topics creados para el robot AmigoBot,

se inicializa el simulador con el correspondiente archivo *.launch, escribiendo en el terminal dentro del directorio que contiene el archivo:

Listing 4.3: Ejecución del archivo amigobot.launch

```
roslaunch amigobot.launch
```

A continuación, se ejecuta el comando de ROS para visualizar los topics existentes en la red ROS, incluyendo tanto los *"Publishers"* como los *"Subscribers"*:

Listing 4.4: Comando para visualizar topics de ROS

```
rostopic list
```

La salida que ofrece este comando se puede ver en la figura 4.7. Los topics asociados a los sensores y actuadores del robot AmigoBot son los remarcados en rojo.

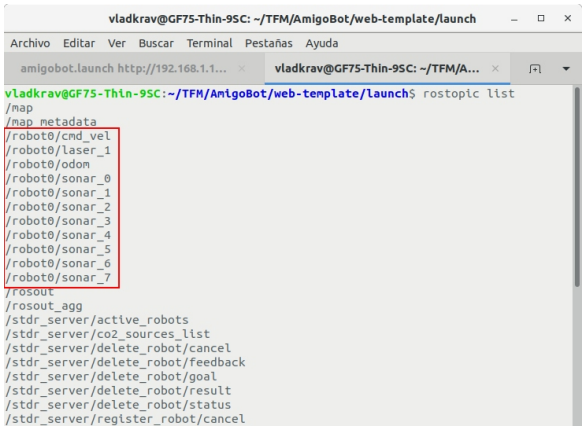


Figura 4.7: Topics creados por el simulador STDR para sensores y actuadores.

Ahora que se saben los topics a los que tenemos que suscribirnos o publicar, se puede continuar con la configuración del HAL. La plataforma Robotics-Academy proporciona código que se heredará para llevar a cabo esta configuración. Por ejemplo, se podrán reutilizar los siguientes archivos:

- *motors.py*: publica en el topic asociado a las velocidades del robot.
- *laser.py*: se suscribe a los topics asociados a los sensores laser.
- *pose3d.py*: se suscribe a los topics que proporcionan la posición del robot.

Con vistas a la estructura de estos archivos, se crea ***sonar.py*** que se suscribirá a los topics de los sensores sonar. Esta conexión en Python se realiza gracias a la librería *rospy* y haciendo uso del comando que proporciona:

Listing 4.5: Conexión mediante la librería *rospy*

```
self.sub = rospy.Subscriber('topic_name','message_class','queue_size')
```

Ahora que se tienen todos los archivos necesarios para establecer la conexión con el robot y los nombres de los topics del simulador STDR, el HAL quedaría de la siguiente manera:

Listing 4.6: Código del HAL para la conexión con el robot AmigoBot

```
self.motors = PublisherMotors("/robot0/cmd_vel", 0.75, 0.75)
self.pose3d = ListenerPose3d("/robot0/odom")
self.sonar_0 = ListenerSonar("/robot0/sonar_0")
self.sonar_1 = ListenerSonar("/robot0/sonar_1")
self.sonar_2 = ListenerSonar("/robot0/sonar_2")
self.sonar_3 = ListenerSonar("/robot0/sonar_3")
self.sonar_4 = ListenerSonar("/robot0/sonar_4")
self.sonar_5 = ListenerSonar("/robot0/sonar_5")
self.sonar_6 = ListenerSonar("/robot0/sonar_6")
self.sonar_7 = ListenerSonar("/robot0/sonar_7")
self.laser = ListenerLaser("/robot0/laser_1")
```

De este modo, el HAL para el robot AmigoBot ya estaría configurado y listo para proporcionar a la aplicación del usuario la información correspondiente.

4.3.2 Desarrollo de exercice.py

En el archivo *exercice.py* se han programado muchas funcionalidades que se ejecutarán en el momento en el que el usuario ejecute este archivo. Desde ese momento, se encarga de crear la comunicación vía *WebSocket* con la página web del ejercicio en el navegador. Así, una vez establecida la conexión, es capaz de recibir el código del usuario para su análisis y posterior ejecución. El código del usuario se analiza para ser dividido en código secuencial y código iterativo. El secuencial se ejecuta una única vez, mientras que, el código iterativo se ejecuta de forma cíclica.

Además, *exercice.py* es capaz de establecer las diferentes frecuencias de ejecución según las preferencias del usuario. Las diferentes frecuencias que se pueden establecer son la frecuencia de ejecución del cerebro del robot que ejecuta el código fuente del usuario, y, por otro lado, la frecuencia del GUI. Este archivo también se encarga de procesar los comandos de los botones del navegador (*Play*, *Stop*, *Save*, *Load*, *Reset*) que el usuario pulse.

Por último, este archivo genera los módulos *"HAL"* y *"GUI"* y los añade a los módulos Python disponibles de la aplicación. Estos módulos proporcionan las funciones específicas del ejercicio, de las cuales el usuario puede hacer uso y el intérprete del código podrá ejecutar.

De este modo, a continuación, se debe realizar la configuración del archivo *exercice.py* para que el usuario de la página web pueda hacer uso de las características que se han implementado en el HAL. Así, se podrá leer los datos de las medidas de los sensores y escribir en los actuadores del robot desde el entorno web.

En primer lugar, se define el módulo del *HAL*:

Listing 4.7: Definición del HAL en el archivo *exercice.py*

```
hal_module = imp.new_module("HAL")
hal_module.HAL = imp.new_module("HAL")
hal_module.HAL.motors = imp.new_module("motors")
```

A continuación, se asignan a estos módulos las funciones creadas:

Listing 4.8: Asignación al HAL de las funciones específicas creadas

```
hal_module.HAL.getPose3d = self.hal.pose3d.getPose3d
hal_module.HAL.motors.sendV = self.hal.motors.sendV
hal_module.HAL.motors.sendW = self.hal.motors.sendW
hal_module.HAL.getLaserData = self.hal.laser.getLaserData
hal_module.HAL.getSonarData_0 = self.hal.sonar_0.getSonarData
hal_module.HAL.getSonarData_1 = self.hal.sonar_1.getSonarData
hal_module.HAL.getSonarData_2 = self.hal.sonar_2.getSonarData
hal_module.HAL.getSonarData_3 = self.hal.sonar_3.getSonarData
hal_module.HAL.getSonarData_4 = self.hal.sonar_4.getSonarData
hal_module.HAL.getSonarData_5 = self.hal.sonar_5.getSonarData
hal_module.HAL.getSonarData_6 = self.hal.sonar_6.getSonarData
hal_module.HAL.getSonarData_7 = self.hal.sonar_7.getSonarData
```

Estas funciones se describen en mayor detalle a continuación:

- **HAL.getPose3d().x**: función que sirve para obtener la posición en x del robot.
- **HAL.getPose3d().y**: función que sirve para obtener la posición en y del robot.
- **HAL.getPose3d().yaw**: función que sirve para obtener la orientación del robot.
- **HAL.motors.sendW()**: función que sirve para establecer la velocidad angular del robot.
- **HAL.motors.sendV()**: función que sirve para establecer la velocidad lineal del robot.
- **HAL.getLaserData()**: función para obtener los datos del sensor LIDAR del robot.
- **HAL.getSonarData_0()**: función para obtener los datos del sonar 1.
- **HAL.getSonarData_1()**: función para obtener los datos del sonar 2.
- **HAL.getSonarData_2()**: función para obtener los datos del sonar 3.
- **HAL.getSonarData_3()**: función para obtener los datos del sonar 4.
- **HAL.getSonarData_4()**: función para obtener los datos del sonar 5.
- **HAL.getSonarData_5()**: función para obtener los datos del sonar 6.
- **HAL.getSonarData_6()**: función para obtener los datos del sonar 7.
- **HAL.getSonarData_7()**: función para obtener los datos del sonar 8.

Para el módulo del *"GUI"*, se procede de la misma forma y se le asigna la función de actualizar el apartado de la interfaz gráfica de la página web.

Listing 4.9: Definición del GUI en el archivo *exercice.py* y asignación de función

```
gui_module = imp.new_module("GUI")
gui_module.GUI = imp.new_module("GUI")
gui_module.GUI.update = self.gui.update_gui
```

- **GUI.update**: función que sirve para actualizar el apartado gráfico del browser.

Con este procedimiento ya se habría configurado el archivo *exercice.py* para el primer ejercicio que se va a desarrollar en este proyecto. Para el segundo ejercicio se procederá de la misma forma.

4.3.3 Desarrollo de la interfaz gráfica

El entorno web de la plataforma Robotics-Academy pretende conseguir en un mismo lugar tanto la programación del control del robot como la simulación de ese mismo control.

Por consiguiente, una parte de control ya ha sido desarrollada en los apartados anteriores. El robot AmigoBot dentro de la plataforma Robotics-Academy ya podría realizar una serie de movimientos y mediciones cumpliendo los objetivos propuestos por el usuario de la aplicación. Sin embargo, por ahora no se visualizarían las acciones del robot en el simulador de la página web. En este apartado se abordará este objetivo: conseguir la correcta visualización de la simulación del robot en la página web.

El simulador STDR proporciona una serie de mapas monocromáticos por los que se puede mover el robot. Uno de estos mapas (Figura 4.8) se va a utilizar y se cargará en la página web con el archivo ***gui.css***.

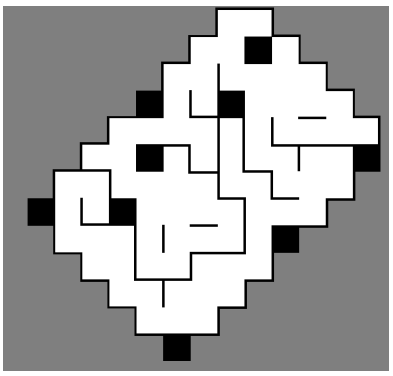


Figura 4.8: Mapa utilizado en la configuración de la interfaz gráfica.

Dentro del archivo *gui.css*, que es un archivo heredado de la plataforma, se modifica la siguiente parte del código, en la que se indica la ubicación de la imagen utilizada como mapa, y, las dimensiones de la imagen. Además se indica el margen que debe haber con el área ocupada por encima de este mapa.

Listing 4.10: Ubicación del mapa y dimensiones en el archivo *gui.css*

```
16 #birds-eye{
17     margin-top: 15px;
18     background: url("./img/robocup.png");
19     background-size: cover;
20     height: 729px;
21     width: 769px;
22 }
```

Una vez cargado el mapa, se tiene que conseguir la representación del robot dentro de este mapa, con la visualización de las medidas de los sonares y las del láser. La representación en la página web se realiza con la etiqueta HTML *<canvas>*, con la que se permite realizar gráficos o animaciones, sobre la marcha, a través de secuencias de comandos (normalmente en JavaScript). Esta etiqueta es solo un contenedor de gráficos, se debe utilizar un script para dibujar realmente los gráficos. Por ello, a continuación, se crea un script con el lenguaje JavaScript que será el encargado de dibujar las medidas sensoriales y el propio robot sobre el espacio dedicado a ello.

El elemento canvas es un espacio de coordenadas en el que el origen de coordenadas (coordenada (0,0)) está ubicado en la esquina superior izquierda. De este modo, todos los elementos que se representen sobre este canvas están posicionados de forma relativa a este punto. Para conseguir la correcta y fiel

representación, hay que realizar una transformación de coordenadas locales del robot a coordenadas globales del sistema. El sistema de coordenadas último (coordenadas globales) en el que se trabajará es el del elemento canvas. El proceso de transformación de sistemas de coordenadas que se sigue es el siguiente:

1. Se parte del sistema local de coordenadas del robot, en el cual se calculan las coordenadas de las medidas de los sensores.

2. El sistema de coordenadas locales anterior se transforma en el sistema de coordenadas globales del sistema de referencia del simulador STDR. En el simulador STDR el origen de coordenadas se sitúa en la esquina inferior izquierda. En teoría, este sistema de referencia es el global si se trabaja directamente con el simulador STDR, pero como se quiere representar el robot en el entorno canvas, se debe realizar un cambio más del sistema de coordenadas.

3. Se transforma el sistema de referencia global del simulador STDR al sistema de referencia del elemento canvas cuyo origen de coordenadas se sitúa en la esquina superior izquierda.

Para conseguir las coordenadas locales al robot de las medidas de sus sensores, se hace uso de la trigonometría básica, en la que utilizando las distancias medidas y los ángulos correspondientes se obtienen las coordenadas deseadas. Por otro lado, los cambios de un sistema de referencia local a otro sistema de referencia global se realizan con la ayuda de las matrices de traslación y rotación que se pueden ver en la ecuación 4.1.

$$\begin{bmatrix} x_g \\ y_g \end{bmatrix} = \begin{bmatrix} x \\ y \end{bmatrix} + \begin{bmatrix} cos\theta & -sen\theta \\ sen\theta & cos\theta \end{bmatrix} * \begin{bmatrix} x_r \\ y_r \end{bmatrix} \tag{4.1}$$

Con esta ecuación matricial se llegaría al sistema de coordenadas del simulador STDR. Para llegar al sistema de referencia del entorno canvas se debe realizar una simple permutación de coordenadas, al tener los sistemas en el mismo lado pero en diferentes esquinas.

Para una mejor visualización del proceso, se presenta la siguiente explicación junto con la Figura 4.9. En esta figura, en la primera representación se puede observar un punto P, que puede ser la medida que ha realizado un sensor sonar, por ejemplo. Con la distancia a ese punto P desde el origen de las coordenadas locales (0, 0), que es el propio robot, se obtienen x_{rP} e y_{rP}. El siguiente paso, es transformar a coordenadas globales y es aquí donde se hace uso de la ecuación matricial 4.1. Se debe saber la posición del robot en el sistema de coordenadas global (x, y) y su orientación respecto a ese sistema, θ, (en este caso 0^o), con lo que se obtendría la posición del punto P en el sistema de referencia global.

En la misma figura, en la segunda representación se mide un punto P, pero ahora el robot tiene un cierto ángulo respecto al sistema de coordenadas global.

Una vez explicado el proceso teórico que se sigue para la representación del robot y sus diferentes medidas sensoriales dentro de la página web, se procede con la programación de este proceso en Python. Esta programación se realiza en el archivo ***map.py*** y con la ayuda de ***GUI.py*** se hace llegar la información generada a los archivos JavaScript, los cuales procesarán esa información y, mediante la etiqueta HTML *<canvas>*(implementado en el archivo ***birds-aye.js***), se dibujarán en la página web las medidas del sensor láser, las medidas del sensor sonar, el movimiento del robot y la representación del mismo robot. Un ejemplo de código implementado con la etiqueta HTML *<canvas>*para la representación de las medidas láser se presenta a continuación en el Listado 4.11:

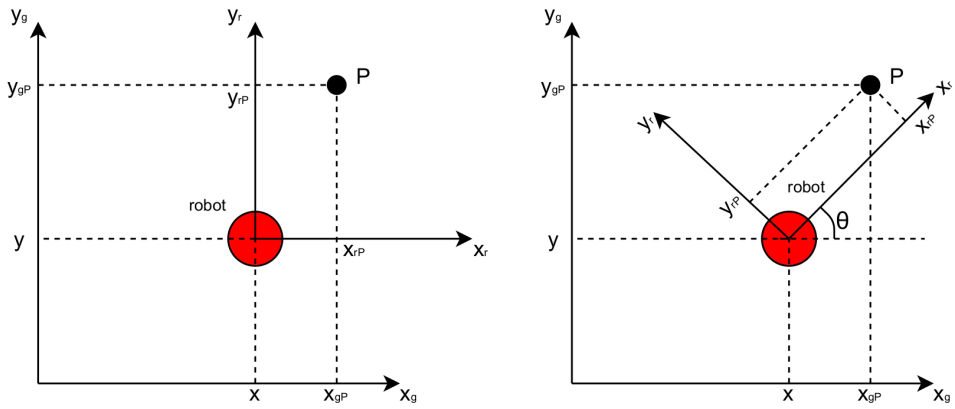


Figura 4.9: Representación del sistemas global y sistema local del robot

Listing 4.11: Ejemplo de representación de las medidas del láser con la etiqueta HTML <canvas>

```
function drawLaser(dataLaser, laser_global){
  for(let d of dataLaser){
    ctx.beginPath();
    ctx.strokeStyle = "#FF2D00";
    ctx.moveTo(laser_global[0], laser_global[1]);
    ctx.lineTo(d[0], d[1]);
    ctx.stroke();
    ctx.closePath();
  }
}
```

Con este proceso de desarrollo del simulador del robot AmigoBot dentro de la página web de Robotics-Academy, se obtiene el siguiente resultado visto en la Figura 4.11. La Figura 4.10 representa la visualización del robot AmigoBot directamente en el simulador STDR. De este modo, se puede realizar una rápida comparación entre los dos simuladores, viendo que, se han conseguido unos resultados realmente parecidos a los que tienen en el simulador original.

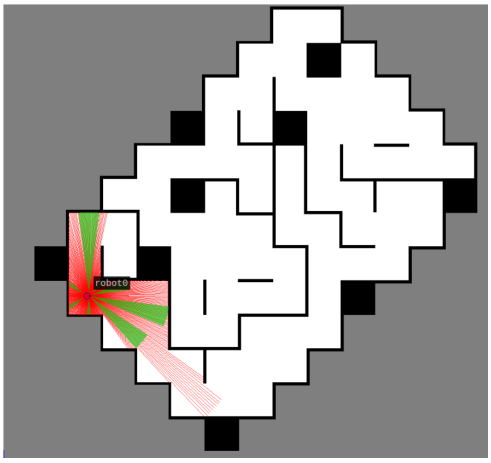


Figura 4.10: AmigoBot representado en el simulador STDR

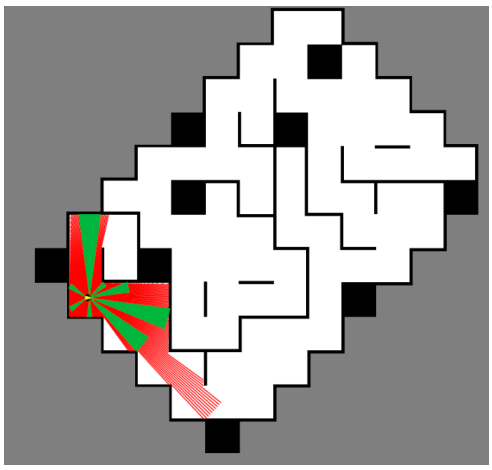


Figura 4.11: AmigoBot representado en la página web

4.3.4 Desarrollo de los archivos de configuración

Una vez implementado el sistema del robot simulado en la página web, se ha decidido realizar algunas pruebas con el robot real. Una de las diferencias que se pueden encontrar entre el robot simulado y el robot real, es que el robot simulado se puede ubicar en cualquier posición del mapa cambiando las coordenadas de inicialización del robot en su archivo de configuración que es fácilmente accesible y editable. Con el robot real no pasa lo mismo. Este siempre se inicializa en la posición (0, 0), y su archivo de configuración no es tan fácilmente accesible. Por lo tanto, si el usuario quiere que el robot se inicialice en la misma posición que el robot simulado dentro de la página web, se vería en la situación en la que no podría hacerlo fácilmente.

Con vistas a este problema, se han añadido los archivos de configuración. Para que este usuario pueda indicar la localización y orientación iniciales del robot, se ha decidido crear dos plantillas de configuración. Estos archivos de configuración sirven para que el usuario pueda indicar la localización y orientación iniciales del robot real sobre el mapa de la página web de Robotics-Academy. Aprovechando este archivo, se han añadido en este otros parámetros adicionales tales como, la posición y orientación de los sensores respecto del sistema de referencia local del robot, los topics de los sensores y actuadores del robot.

Se han creado dos archivos de configuración para diferenciar los parámetros individuales tanto para el robot simulado como para el robot real. Estos archivos de configuración se han creado con el formato JavaScript Object Notation (JSON) y se han denominado **config_sim.json** y **config_real.json**. JSON es un formato que almacena información de forma estructurada y se utiliza principalmente para transferir información de forma muy simple. A continuación, en el Listado 4.12, se puede ver el contenido de uno de estos archivos de configuración.

Para transferir información desde esos archivos de configuración hacia el lenguaje Python, se ha creado el archivo **parse_configuration.py**. Así se obtienen los parámetros básicos para la puesta en marcha del sistema. Además, con este método se pueden relacionar dentro del código ya desarrollado los parámetros que se han indicado dentro de estos archivos de configuración. Un ejemplo de este procedimiento es el propio HAL, debido a que en los archivos de configuración se indican los topics necesarios para el sistema. De este modo, el HAL leería estos topics directamente desde estos archivos de configuración, quedando una forma más genérica respecto al desarrollo realizado durante los primeros pasos. La nueva estructura del HAL, como ejemplo de esta relación de parámetros desde los archivos de configuración, se puede ver a continuación en el Listado 4.13.

Listing 4.13: Estructura renovada del HAL

```
self.config = Config()
self.motors = PublisherMotors(self.config.topic_motors, self.max_velV, self.max_velW)
self.pose3d = ListenerPose3d(self.config.topic_pose)
self.sonar_0 = ListenerSonar(self.config.topic_sonar_0)
self.sonar_1 = ListenerSonar(self.config.topic_sonar_1)
self.sonar_2 = ListenerSonar(self.config.topic_sonar_2)
self.sonar_3 = ListenerSonar(self.config.topic_sonar_3)
self.sonar_4 = ListenerSonar(self.config.topic_sonar_4)
self.sonar_5 = ListenerSonar(self.config.topic_sonar_5)
self.sonar_6 = ListenerSonar(self.config.topic_sonar_6)
self.sonar_7 = ListenerSonar(self.config.topic_sonar_7)
self.laser = ListenerLaser(self.config.topic_laser)
```

Ya se tiene la estructura necesaria para establecer una posición determinada, la que desee el usuario, al robot. Las transformaciones necesarias para llevar al robot a la posición (x, y) y rotación indicadas por el usuario, se realizan mediante nueva matriz de traslación y rotación, haciendo uso de la ecuación 4.1.

Listing 4.12: Archivo de configuración **config_sim.json**

```
{
    "CONFIGURATION": {
        "POS_X" : 4,
        "POS_Y" : 8,
        "ORIENTATION" : 0,
        "MAX_VEL_V" : 0.75,
        "MAX_VEL_W" : 0.75,
        "TOPIC_POSE" : "/robot0/odom",
        "TOPIC_MOTOR" : "/robot0/cmd_vel",
        "TOPIC_LASER" : "/robot0/laser_1",
        "TOPIC_SONAR0": "/robot0/sonar_0",
        "TOPIC_SONAR1": "/robot0/sonar_1",
        "TOPIC_SONAR2": "/robot0/sonar_2",
        "TOPIC_SONAR3": "/robot0/sonar_3",
        "TOPIC_SONAR4": "/robot0/sonar_4",
        "TOPIC_SONAR5": "/robot0/sonar_5",
        "TOPIC_SONAR6": "/robot0/sonar_6",
        "TOPIC_SONAR7": "/robot0/sonar_7",
        "SONAR_0": {
            "POS_X": 0.076,
            "POS_Y": 0.1,
            "ORIENTATION" : 1.5708
        },
        "SONAR_1":{
            "POS_X": 0.125,
            "POS_Y": 0.075,
            "ORIENTATION" : 0.715585
        },
        "SONAR_2":{
            "POS_X": 0.15,
            "POS_Y": 0.03,
            "ORIENTATION" : 0.261799
        },"SONAR_3":{
            "POS_X": 0.15,
            "POS_Y": -0.03,
            "ORIENTATION" : -0.261799
        },
        "SONAR_4":{
            "POS_X": 0.125,
            "POS_Y": -0.075,
            "ORIENTATION" : -0.715585
        },
        "SONAR_5":{
            "POS_X": 0.076,
            "POS_Y": -0.1,
            "ORIENTATION" : -1.5708
        },
        "SONAR_6":{
            "POS_X": -0.14,
            "POS_Y": -0.058,
            "ORIENTATION" : -2.53073
        },
        "SONAR_7":{
            "POS_X": -0.14,
            "POS_Y": 0.058,
            "ORIENTATION" : 2.53073
        },
        "LASER":{
            "POS_X": 0.09,
            "POS_Y": 0.0,
            "ORIENTATION" : 0.0
        }
    }
}
```

4.3.5 Desarrollo de la teleoperación del robot

Al igual que se realiza en el entorno de MatLab-ROS, se desea teleoperar al robot en la plataforma de Robotics-Academy mediante teclado, para que el usuario pueda libremente mover el robot por el mapa. El concepto general que se ha propuesto para lograr esta meta es el siguiente: El usuario mediante un switch, puede indicar en cualquier momento si quiere realizar la teleoperación del robot. Si el usuario ha puesto el switch en *ON*, al realizar la pulsación de una serie de teclas determinadas, se le mandan diferentes comandos al robot (avanza, gira a la derecha, gira a la izquierda, para, etc.). Por el contrario, si el usuario desactiva este switch poniéndolo a *OFF*, se deja de poder realizar el control mediante la teleoperación.

Siguiendo este concepto general, se ha implementado en primer lugar el switch para la activación o desactivación de la teleoperación gracias a un *checkbox*, que es un elemento HTML de entrada. Utilizando su atributo *checked* (booleano que puede valer *true* o *false*) se consigue capturar la entrada del usuario sobre el switch en la página web. Para representar el switch en la página web se ha utilizado un archivo CSS, en el que se ha introducido el código para visualizar el siguiente aspecto del switch, visto en las figuras 4.12 y 4.13.

Figura 4.12: Switch OFF

Figura 4.13: Switch en ON

Ahora que se tiene implementado el switch, cada vez que el usuario lo active/desactive, se mandará un mensaje a la plantilla Python del ejercicio (*exercice.py*) indicando el estado de este switch. Cuando esté activo, el cerebro no leerá/ejecutará los comandos que el usuario mande desde su código Python para el control del robot. Esta exclusión se realiza para no producir interferencias en la ejecución si está activada la teleoperación y el usuario ejecute comandos de movimiento del robot desde su teclado. Si la teleoperación está activada, se debe controlar el robot mediante el teclado, por el contrario, se controla desde el programa escrito del usuario.

Para controlar el robot mediante la teleoperación, se deben establecer una serie de teclas para el control del robot. Las teclas elegidas son las siguientes:

- Tecla 'w': comando para ir hacia delante.
- Tecla 's': comando para parar.
- Tecla 'a': comando para girar a la izquierda.
- Tecla 'd': comando para girar a la derecha.
- Tecla 'q': comando para incrementar la velocidad lineal un 10 %.
- Tecla 'z': comando para decrementar la velocidad lineal un 10 %.
- Tecla 'e': comando para incrementar la velocidad angular un 10 %.
- Tecla 'c': comando para decrementar la velocidad angular un 10 %.

En la página web, mediante el evento *onkeydown*, se realiza la captura de las teclas pulsadas por el usuario. Si la tecla coincide con alguna de las teclas elegidas, se envía esa tecla hacia el cerebro, el cuál procesa el comando a mandar al robot para que realice alguna acción concreta de las descritas.

Además, se ha implementado en la página web una ayuda para el usuario, en la que se indican las teclas que se deben pulsar para realizar el control del robot. Para ello se ha optado por una ventana *pop-up* que se visualice únicamente cuando el usuario mueva el ratón sobre el switch implementado anteriormente. Esto se ha conseguido realizar mediante el método *toggle* que posee la propiedad *classList* de JavaScript. Cada vez que el usuario entra en la zona del switch y sale de la zona, se realiza un toggle para ver o dejar de ver el mensaje *pop-up*. Una vez implementada esta funcionalidad su aspecto en la página web se puede ver en la siguiente Figura 4.14.

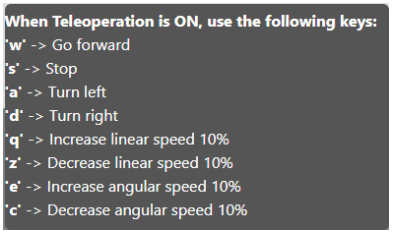


Figura 4.14: Mensaje pop-up de ayuda.

De este modo, el usuario de la plataforma ya puede realizar la teleoperación del robot AmigoBot.

4.4 Ejercicio mapeado con posiciones conocidas

En esta sección se describe el desarrollo realizado del ejercicio dedicado al mapeado mediante posiciones conocidas, a semejanza del ejercicio de mapeado realizado en el entorno MatLab-ROS. Una vez se ha trabajado con este tipo de mapeado en el entorno de MatLab-ROS, se procede a realizar una funcionalidad muy parecida en la plataforma de Robotics-Academy. Al igual que se ha explicado en la sección 3.2.1, este método consiste en realizar un mapeado del entorno gracias a la obtención de las medidas del LIDAR y conociendo también la posición del robot, mediante su odometría.

Gracias a que se tiene el entorno configurado, se procede directamente a implementar las nuevas características que va a necesitar este ejercicio. Por ello se parte de un primer concepto de lo que tendría que tener la página web para que el usuario pueda realizar este mapeado con posiciones conocidas:

- La página web debe tener el mapa del entorno sobre el que se moverá el robot, mediante la teleoperación.
- Se dispondrá de un espacio adicional en el cuál se visualizará el resultado del mapeado.
- El resultado del mapeado tiene que visualizarse en tiempo real.
- La visualización del mapeado o, lo que es lo mismo, la activación o desactivación del mapeado, se realizará mediante un botón.

En primer lugar, se procede a crear ese espacio gráfico adicional en el que se dibujará posteriormente el resultado que se irá obteniendo del mapeado. Para ello, en el archivo *gui.css* se crea un espacio en blanco de dimensiones 500 píxeles por 500 píxeles, para obtener un mapa de hasta 25x25 metros, con una resolución de 20 celdas por metro.

A continuación, se introduce el código necesario para visualizar este espacio del mapa en la página web HTML. Esto se realiza con la etiqueta de HTML *<canvas>* que se ha introducido en la sección 4.3.3. Por ello, a continuación se crea un script con el lenguaje JavaScript que será el encargado de dibujar el mapeado sobre este espacio dedicado al mapa.

El script se crea en el archivo denominado *mapping.js*. En este script, la tarea principal que se realiza, es la de dibujar sobre el mapa los puntos obtenidos de los impactos del sensor LIDAR al recorrer el robot el entorno. Al realizar el mapeado, pueden existir impactos que no se corresponden a la realidad del entorno. Por este motivo, se implementa la siguiente funcionalidad: el mapa generado tiene unas dimensiones determinadas en píxeles, por lo que se podría considerar una rejilla en la que cada píxel representa un punto en el que puede impactar el sensor LIDAR. Con cada impacto en un punto determinado del sensor LIDAR se va incrementando un contador, con un valor de 0 a 255, donde el valor de 255 representa que no ha habido impactos (un espacio en blanco, espacio libre de obstáculos), mientras que un valor de 0 representa un espacio en el que hubo muchos impactos (por lo que se podría considerar un espacio ocupado).

También, dentro de este script se considera que si hay un impacto por detrás de un píxel que se consideraba ocupado (en negro) se borra y se pone en blanco, reiniciando el contador. Esto se realiza para estar más seguros del resultado que obtendrá al terminar el mapeado.

Por último, la configuración del elemento canvas es tal que cada vez que es llamado, se reinicia el dibujo existente quedando el espacio de dibujo en blanco, con cada iteración del código. Lo que se quiere implementar en este ejercicio es un mapeado continuo en el que se podrá ir viendo en cada paso como se está realizando el mapeado del entorno. Por ello, se debe conseguir guardar de alguna forma el dibujo del mapa que se ha generado hasta el instante actual de ejecución. Para conseguirlo, se hace uso de de los métodos pertenecientes a la API de canvas 2D: ***getImageData()*** y ***putImageData()***.

El método *getImageData()* devuelve un objeto *ImageData* que representa los datos de píxeles subyacentes para el área del lienzo denotada por el rectángulo que comienza en (x,y) y tiene un ancho y alto determinados. Con este método lo que se consigue es guardar el dibujo del lienzo para posteriormente restaurarlo igual que estaba. Para restaurar el dibujo se hace uso del método *putImageData()*, que dibuja los datos del objeto *ImageData* sobre el lienzo.

Ahora que ya se realiza el dibujo del mapeado, se procede a implementar el botón que activará o desactivará esta funcionalidad en la página web.

Este botón tendrá una apariencia similar a la implementada en otros ejercicios de la plataforma Robotics-Academy. La creación de este botón se realiza en el archivo *main.css*, y, a continuación, se introduce en el archivo *exercice.html*. Aquí se le asigna una imagen que se va a utilizar para su visualización. También, en cada pulsación de este botón saltará la ejecución de una función implementada en el archivo *ws_gui.js* encargada de visualizar o no (dependiendo del estado del botón) el mapeado del ejercicio. El botón implementado tiene el siguiente formato visto en la Figura 4.15.

Figura 4.15: Botón para habilitar o deshabilitar el mapeado

4.4.1 Resultados obtenidos

En este momento, el ejercicio de mapeado con posiciones conocidas tendría implementadas todas las funciones que se habían planteado en un principio. Los resultados que se obtienen con esta implementación

son los que se pueden ver en la siguiente Figura 4.17:

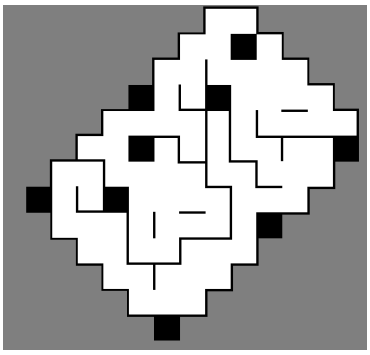


Figura 4.16: Mapa del que se realiza el mapeado con posiciones conocidas

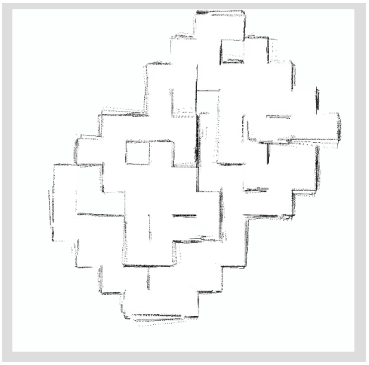


Figura 4.17: Resultado del mapeado mediante posiciones conocidas

4.5 Ejercicio Monte Carlo Localization.

En esta sección se abordará el desarrollo, al igual que se hizo en el entorno MatLab-ROS, del ejercicio orientado a la auto-localización del robot mediante la técnica Monte Carlo Localization. Se explicará el desarrollo de la visualización específica necesaria para este ejercicio. A continuación se abordará la creación de una plantilla funcional de este algoritmo para su utilización por el alumnado.

4.5.1 Implementación de la visualización

Para trabajar con esta técnica se deben preparar unas funciones específicas para poder representar tanto la localización de las partículas en el entorno como la posición estimada del robot. Las dos funciones que se van a crear son:

- ***showParticles(particles)***.
- ***showEstimatedPose(pose)***.

Estas funciones se crean en el archivo ***gui.py*** y, una vez llamadas, transfieren la información necesaria al archivo JavaScript encargado de realizar la visualización de la posición de las partículas y la posición estimada del robot, así cómo, la orientación de estos.

Este archivo escrito en JavaScript es el que ya se había visto anteriormente denominado ***birds_eye.js***. En él se desarrolla la visualización, creando una nueva función: ***drawAMCL***. Esta función recibe como argumentos dos listas que deben contener la posición en x, la posición en y y la orientación, θ. La posición (x,y) se debe expresar en píxeles. En esta función ***drawAMCL***, gracias a las matrices de rotación y traslación, se dibujan las partículas y la posición estimada del robot en las posiciones del mapa correctas.

Para una primera representación las partículas se dibujan círculos con origen en una coordenada (x, y) con un radio determinado. En la posición estimada del robot también se realiza la misma operación pero con un radio del círculo mayor para que se pueda distinguir su situación. Para visualizar la orientación sobre estos círculos se ha decidido dibujar formas de "*ChupaChups*", esto es, se dibuja una línea recta desde el centro del círculo hasta una posición fuera del círculo, a una distancia que se pueda distinguir claramente la orientación. El resultado es el que se puede visualizar en la Figura 4.18.

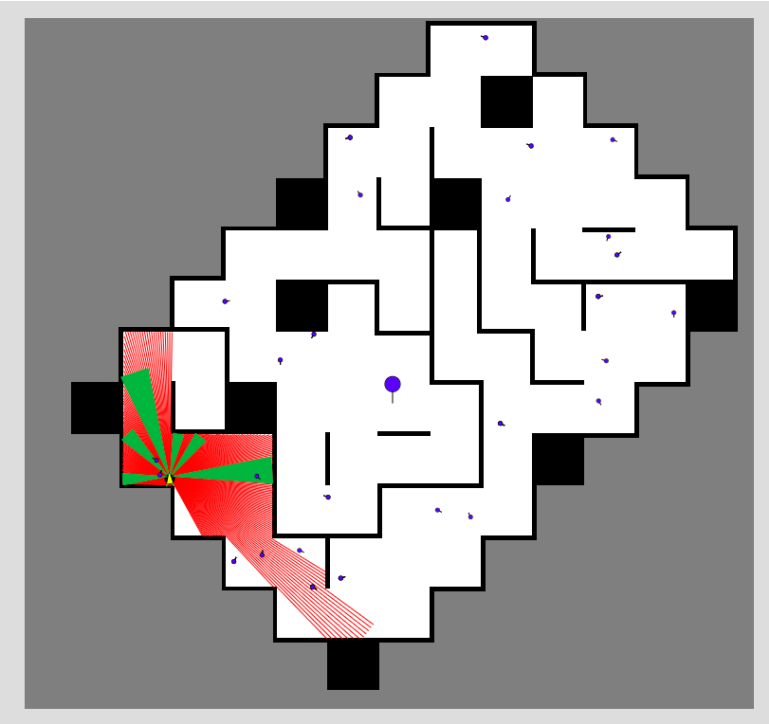


Figura 4.18: Mapa RoboCup con la visualización de las partículas y la posición estimada del robot.

Una vez implementadas estas funciones se procede a dejarlas habilitadas para su uso por el usuario en el navegador. Para esto se añaden a los módulos "*GUI*" las siguientes dos funciones que el usuario puede llamar de la siguiente forma:

- **GUI.showParticles()**: esta función debe recibir como argumento una lista con cinco arrays. Los tres primeros índices deben ser: posición x de la partícula, posición y de la partícula, orientación θ de la partícula. El cuarto índice es el peso de la partícula actual, mientras que el quinto índice es el peso de la partícula anterior. Esta función representa las partículas sobre el mapa cargado en el navegador web.
- **GUI.showEstimatedPose()**: esta función debe recibir como argumento una lista con tres índices. El primero debe ser la posición en x estimada del robot. En segundo lugar, la posición y estimada del robot. Y, por último, la orientación θ estimada del robot. Esta función se encarga de representar la posición estimada del robot sobre el mapa del navegador.

En la función **GUI.showParticles()** se ha añadido un rango de colores que indica visualmente el peso de las diferentes partículas. Es un rango dividido en 5 partes:

1. Peso <0.2. Si el peso es menor que una probabilidad de 0.2, entonces se representa esa partícula en blanco.
2. 0.2 $\leq$ Peso <0.4. Si el peso está en este rango de probabilidad, entonces se representa en amarillo.
3. 0.4 $\leq$ Peso <0.6. Se representa en naranja.
4. 0.6 $\leq$ Peso <0.8. Se representa en rojo.
5. 0.8 $\leq$ Peso $\leq$ 1. Se representa en verde.

Así el usuario ya sería capaz de representar tanto la posición estimada del robot como la posición de las partículas sobre el entorno que son necesarias para la realización de este ejercicio.

4.5.2 Plantilla del algoritmo, solución de referencia

El siguiente paso es desarrollar una plantilla funcional del algoritmo MCL, para que el usuario tenga en su mano la solución. Posteriormente, cuando el ejercicio se ponga en funcionamiento en la plataforma Robotics-Academy, esta plantilla funcional se proporcionará de tal forma que el usuario tenga que rellenar unas partes determinadas del código para su correcto funcionamiento. Así, abarcará el aprendizaje de este algoritmo de auto-localización del robot.

4.5.2.1 Etapa de Inicialización

En primer lugar, en esta plantilla se trabaja con el mapa del entorno. De este mapa se obtienen las líneas que forman las paredes. Estas líneas se obtienen de forma manual sabiendo las coordenadas de dos puntos de la línea. Así, se obtiene que el mapa RoboCup es un mapa formado por 70 líneas que se han obtenido al calcular 140 puntos del mapa. Estos puntos se presentan en la siguiente tabla 4.1.

Tabla 4.1: Conjunto de puntos que forman las líneas de las paredes del mapa Robocup

Línea 1	[104,330,1][104,497,1]	**Línea 2**	[104,497,1][157,497,1]	**Línea 3**	[157,497,1][157,550,1]
Línea 4	[157,550,1][211,550,1]	**Línea 5**	[211,550,1][211,604,1]	**Línea 6**	[211,604,1][265,604,1]
Línea 7	[265,604,1][265,659,1]	**Línea 8**	[265,659,1][431,659,1]	**Línea 9**	[431,659,1][431,604,1]
Línea 10	[431,604,1][484,604,1]	**Línea 11**	[484,604,1][484,547,1]	**Línea 12**	[484,547,1][538,547,1]
Línea 13	[538,547,1][538,439,1]	**Línea 14**	[538,439,1][647,439,1]	**Línea 15**	[647,439,1][647,385,1]
Línea 16	[647,385,1][700,385,1]	**Línea 17**	[700,385,1][700,283,1]	**Línea 18**	[747,277,1][535,277,1]
Línea 19	[591,328,1][591,281,1]	**Línea 20**	[537,223,1][537,281,1]	**Línea 21**	[590,224,1][644,224,1]
Línea 22	[747,277,1][747,224,1]	**Línea 23**	[747,224,1][698,224,1]	**Línea 24**	[698,224,1][698,171,1]
Línea 25	[698,171,1][644,171,1]	**Línea 26**	[644,171,1][644,117,1]	**Línea 27**	[644,117,1][590,116,1]
Línea 28	[590,116,1][590,62,1]	**Línea 29**	[590,62,1][482,62,1]	**Línea 30**	[482,62,1][482,116,1]
Línea 31	[482,116,1][536,116,1]	**Línea 32**	[536,116,1][536,8,1]	**Línea 33**	[536,8,1][428,8,1]
Línea 34	[428,8,1][428,62,1]	**Línea 35**	[428,62,1][374,62,1]	**Línea 36**	[374,62,1][374,116,1]
Línea 37	[374,116,1][320,116,1]	**Línea 38**	[320,116,1][320,224,1]	**Línea 39**	[320,224,1][212,224,1]
Línea 40	[212,224,1][212,278,1]	**Línea 41**	[212,278,1][158,278,1]	**Línea 42**	[158,278,1][158,332,1]
Línea 43	[104,332,1][212,332,1]	**Línea 44**	[212,332,1][212,440,1]	**Línea 45**	[158,440,1][266,440,1]
Línea 46	[158,440,1][158,384,1]	**Línea 47**	[266,385,1][266,549,1]	**Línea 48**	[266,385,1][212,385,1]
Línea 49	[266,549,1][374,549,1]	**Línea 50**	[320,603,1][320,549,1]	**Línea 51**	[374,549,1][374,495,1]
Línea 52	[374,495,1][482,495,1]	**Línea 53**	[482,495,1][482,387,1]	**Línea 54**	[482,387,1][431,387,1]
Línea 55	[431,387,1][431,116,1]	**Línea 56**	[431,333,1][370,333,1]	**Línea 57**	[370,333,1][370,279,1]
Línea 58	[370,279,1][262,279,1]	**Línea 59**	[262,279,1][262,333,1]	**Línea 60**	[262,333,1][316,333,1]
Línea 61	[316,333,1][316,279,1]	**Línea 62**	[482,222,1][374,222,1]	**Línea 63**	[374,222,1][374,168,1]
Línea 64	[482,170,1][482,334,1]	**Línea 65**	[482,170,1][428,170,1]	**Línea 66**	[482,334,1][536,334,1]
Línea 67	[536,334,1][536,388,1]	**Línea 68**	[536,388,1][590,388,1]	**Línea 69**	[320,493,1][320,439,1]
Línea 70	[373,437,1][428,437,1]				

En el algoritmo se crean las tablas LookUp Table (LUT) que son unas estructuras de datos (normalmente vectores), que se usan para sustituir unas rutinas de computación mediante una simple indexación de los vectores. Así, el algoritmo consigue ahorrar tiempo de procesamiento, porque sacar un valor de la memoria es mucho más rápido que hacer un cálculo. De este modo tenemos tablas LUT para los puntos

de las líneas y las propias líneas que se obtienen al hacer el producto vectorial de los puntos (se explicará más en detalle a continuación).

El siguiente paso es la generación de las partículas aleatorias alrededor de la posición del robot, así, siendo una localización local la que se va a llevar a cabo en esta plantilla. Se generan tantas partículas como se indiquen en la variable *num_particles*. Todo este proceso se realiza en la parte del código secuencial. En la parte del código iterativo el algoritmo se ejecuta si se cumple que el robot haya recorrido una distancia determinada sobre el mapa. Si se cumple esta condición se pasa al cálculo del algoritmo propio MCL.

4.5.2.2 Etapa de Observación

Se continúa con la etapa de observación. En esta etapa para cada partícula se calcula el vector estimado de distancias del haz del láser hasta los obstáculos en forma de paredes del mapa. Ahora es necesario obtener el inicio y el final del haz en forma de dos puntos expresados en coordenadas cartesianas (x,y). El primer punto es la localización de la partícula. El segundo punto es el impacto del haz estimado que parte desde esa partícula e impacta en una pared del mapa. Para obtener este punto de impacto se hace uso de la geometría proyectiva planar 2D. Si tenemos una línea infinita formada por dos puntos P_1 y P_2, ecuación 4.2. Estos puntos corresponderán al haz del láser:

$$\begin{aligned} &P_1(x_1, y_1, h_1) \;\; \textit{siendo } h_1 = 1 \\ &P_2(x_2, y_2, h_2) \;\; \textit{siendo } h_2 = 1 \end{aligned} \tag{4.2}$$

Siendo la ecuación de la recta la que viene representada en la ecuación 4.3:

$$a \cdot x + b \cdot y + c \cdot h = 0 \tag{4.3}$$

Haciendo el producto vectorial de estos dos puntos P_1 y P_2 se pueden obtener los valores de a, b y c de la línea, L_1, que forma el haz láser:

$$\begin{aligned} &L_1 = P_1 \times P_2 \\ &L_1(a, b, c) \end{aligned} \tag{4.4}$$

Así, se calculan todas las líneas existentes en el mapa, tanto de las paredes del mapa como de los haces del láser estimados. Por otra parte, la intersección de dos líneas se puede expresar como el producto vectorial, ecuación 4.5.

$$S(x, y, h) = L_1 \times L_2 \tag{4.5}$$

El resultado obtenido se debe normalizar para obtener el punto de intersección, S, de las dos líneas, pared del mapa y haz láser.

$$S(\frac{x}{h}, \frac{y}{h}, 1) \tag{4.6}$$

Siendo el punto S el punto de intersección entre la línea infinita que forma el rayo láser y la línea infinita que forma el segmento de pared del mapa. Se deben calcular todos los puntos de intersección entre todos los haces del láser estimado con todas las paredes del mapa. Estando formado el segmento de la pared del mapa mediante sus puntos extremos Q_1 y Q_2, entonces S cae en el interior del segmento de la pared del mapa si se satisface uno de los siguientes casos:

1. En el caso en el que las coordenadas x sean diferentes en los puntos Q_1 y Q_2, el punto S de la intersección pertenece al segmento si cumple las siguientes condiciones, siendo P_1 y P_2 los puntos del segmento que forma el haz del láser:

   1. $min(Q_{1x}, Q_{2x}) <= S_x <= max(Q_{1x}, Q_{2x})$
   2. $min(P_{1x}, P_{2x}) <= S_x <= max(P_{1x}, P_{2x})$ y $min(P_{1y}, P_{2y}) <= S_y <= max(P_{1y}, P_{2y})$

   (4.7)

2. En el caso en el que las coordenadas x sean iguales en los puntos Q_1 y Q_2 (segmento vertical), el punto S de la intersección pertenece al segmento si cumple las siguientes condiciones, siendo P_1 y P_2 los puntos del segmento que forma el haz del láser:

   1. $min(Q_{1y}, Q_{2y}) <= S_y <= max(Q_{1y}, Q_{2y})$
   2. $min(P_{1x}, P_{2x}) <= S_x <= max(P_{1x}, P_{2x})$ y $min(P_{1y}, P_{2y}) <= S_y <= max(P_{1y}, P_{2y})$

   (4.8)

Estas dos últimas condiciones 4.7 y 4.8 gráficamente representan lo que está representado en la Figura 4.19.

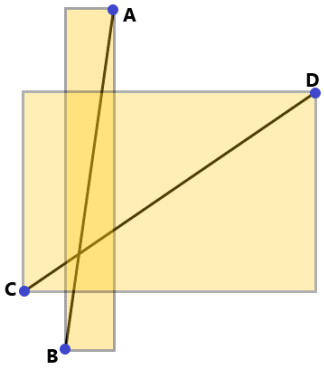


Figura 4.19: Intersección de dos segmentos de línea.

La Figura 4.19 muestra que los segmentos AB y CD se cruzan si su punto de intersección se encuentra dentro del rectángulo medio más oscuro, es decir, el área en el espacio que ambos ocupan. En otras palabras, si el punto de intersección es (x, y), entonces, x debe ser menor que el valor más pequeño del lado derecho y mayor que el valor más pequeño del lado izquierdo. De forma similar se comprueba para la coordenada y. Se deben pasar las cuatro pruebas (dos para la coordenada x y dos para la coordenada y) antes de concluir que se cruzan los segmentos.

Una vez implementada esta solución, se vio que el algoritmo funcionaba bien, calculando todas las líneas y los puntos de intersección, pero al estar calculando el punto de intersección de todos los rayos láser de las partículas estimadas con todas las paredes del mapa (70), resultaba en un algoritmo que consumía

demasiados recursos para una operación fluida que desembocara en la auto-localización del robot. Por este motivo se decidió implementar una comprobación adicional que redujera la carga computacional del algoritmo.

En un primer momento, la comprobación adicional que se decidió implementar fue hacer uso de una librería adicional llamada **Shapely**, la cual contiene un método *object.intersects(other)* que devuelve un valor de *True* si el objeto intersecta de alguna manera con *other*. Esta forma también funcionaba y reducía el tiempo de computación del algoritmo, pero no era lo suficiente para una operación fluida.

Viendo este panorama, se encontró otra forma de determinar si dos segmentos de línea se cruzan, y, se optó por realizar todas las operaciones de comprobación a mano, mediante multiplicaciones, sumas y restas. Así, la teoría dice que la forma robusta de determinar la intersección de dos segmentos formados por puntos finales (A, B) y (C,D) es verificar el signo de los cuatro determinantes expuestos en la ecuación 4.9:

$$\begin{vmatrix} A_x - C_x & B_x - C_x \\ A_y - C_y & B_y - C_y \end{vmatrix} \begin{vmatrix} A_x - D_x & B_x - D_x \\ A_y - D_y & B_y - D_y \end{vmatrix}$$
$$\begin{vmatrix} C_x - A_x & D_x - A_x \\ C_y - A_y & D_y - A_y \end{vmatrix} \begin{vmatrix} C_x - B_x & D_x - B_x \\ C_y - B_y & D_y - B_y \end{vmatrix} \tag{4.9}$$

Cada determinante está calculando el producto vectorial de los puntos extremos de los vectores de dos segmentos de línea. Para comprobar que realmente intersectan los segmentos se debe cumplir que el resultado de los determinantes multiplicados entre sí sean menores o iguales que cero. Esto se expresa en la siguiente ecuación 4.10. Siendo Det_0 el determinante arriba a la izquierda, Det_1, el determinante arriba a la derecha, Det_2, el determinante abajo a la izquierda y, por último, Det_3, el determinante abajo a la derecha:

$$\begin{aligned} Det_0 \cdot Det_1 <= 0 \\ Det_2 \cdot Det_3 <= 0 \end{aligned} \tag{4.10}$$

Para comprobar esto se tiene un simulador de esta situación en la siguiente página web: desmos (*https://www.desmos.com/calculator/0wr2rfkjbk?lang=es*)

Todo este proceso resultó en una reducción significativa de la carga computacional del algoritmo de la etapa de observación. A continuación se presenta una tabla comparativa, 4.2, entre los tiempos de ejecución de cada uno de los códigos, tanto sin optimizar como optimizando de las dos formas vistas. Estos tiempos se han obtenido con el algoritmo ajustado para crear 50 partículas.

Algoritmo	Sin optimizar	Shapely	Optimización a mano
Tiempo (s)	0.687986850739	0.393244028091	0.0595829486847

Tabla 4.2: Comparación de tiempos de cómputo de la etapa de observación.

De esta forma se ha conseguido reducir significativamente el tiempo de ejecución del algoritmo. Ese tiempo optimizado puede utilizarse para la generación de un número de partículas mayor o, como se verá más adelante, tener en cuenta un número mayor de haces del láser.

De esta forma, ahora, ya se saben los dos puntos, el de partida y el de impacto del haz, por lo que se rellenan cada uno de los vectores estimados. Posteriormente, cada uno de estos vectores estimados se comparará con el vector real de haces obtenido de los sensores del robot.

En el vector real se devuelven 400 haces del láser, lo que es un número demasiado elevado lo que supondría una elevada carga computacional. Por ello se disminuye esta cantidad de información a un número menor, por ejemplo, de 10 haces del láser, para que el cálculo sea mucho más ligero. Estos 10 haces se cogen de forma equiespaciada y el resultado que se obtiene se puede ver en la siguiente Figura 4.20.

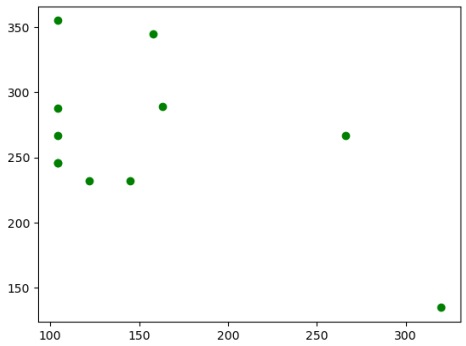


Figura 4.20: Representación de los impactos de los 10 haces equiespaciados.

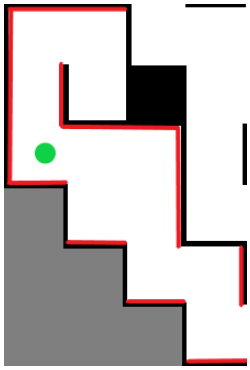


Figura 4.21: Posibles paredes de impacto de los haces del láser del robot real.

La comparación de las figuras 4.20 y 4.21 nos muestra que con 10 haces del láser, estos serían los datos que tendríamos para comprar con los haces del láser estimado de cada partícula. En la figura de la derecha, en verde se representa la posición del robot desde la que se han obtenido los 10 haces del láser. Se puede intuir la posición que está representando la Figura 4.20 mirando la Figura 4.21, aunque siendo el espacio del mapa tan similar podría haber confusión.

Para comparar el vector estimado de las partículas con el vector real del robot, se deben saber las distancias desde el robot hasta los obstáculos. Para obtener esta distancia se hace uso de la **distancia Manhattan**. Este método nos dice cuál es la distancia entre dos puntos en una cuadrícula de líneas rectas y cuadros rodeados por líneas rectas. Este método expresado matemáticamente nos dice que si tomamos dos puntos p y q en una cuadrícula con coordenadas $p = (p_1, p_2)$ y $q = (q_1, q_2)$, la distancia Manhattan entre dichos puntos es la suma de los valores absolutos de las diferencias entre las coordenadas de estos puntos. De tal forma que sería como en la ecuación 4.11:

$$d(p, q) = |q_1 - p_1| + |q_2 - p_2| \tag{4.11}$$

En la Figura 4.22 se puede ver una cuadrícula con dos puntos unidos por diferentes líneas. La línea verde indica la distancia euclídea mientras las líneas roja, amarilla y azul tienen la misma longitud de ruta, pero son varias maneras de unir ambos puntos con un camino mínimo siguiendo las líneas rectas.

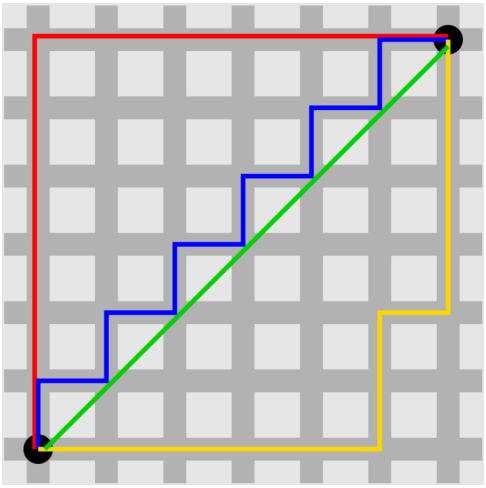


Figura 4.22: Ilustración de la distancia Manhattan.

Sabiendo los puntos de partida y los puntos de impactos de los haces láser es muy fácil obtener este valor de la distancia.

Una vez se saben las distancias, se realiza la diferencia entre los valores obtenidos de los haces estimados y los haces del láser realmente observado. Ese valor de la diferencia se almacena en otro vector llamado *diferencias*, que indicará la mayor o menor concordancia de cada partícula con las lecturas del robot real.

Más adelante, en este vector *diferencias*, se realiza la suma de sus elementos. El resultado de esta suma se utilizará para obtener el peso de cada partícula. El peso se obtiene con el siguiente modelo probabilístico de observación, expuesto en la siguiente expresión 4.12:

$$Prob\ (posicion/observacion\ laser(t)) = e^{-(\sum_{i=0}^{n} diferencias[i])} \tag{4.12}$$

Con este modelo probabilístico se consigue que las partículas tengan un modelo gradual de probabilidad de observación.

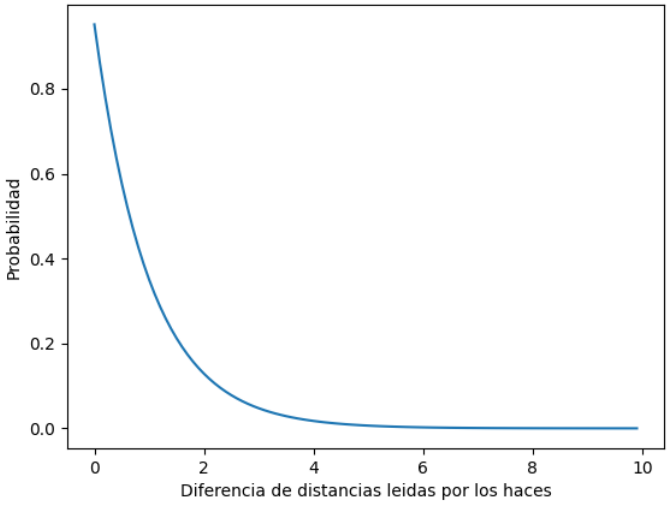


Figura 4.23: Gráfico Probabilidad - Diferencia de valores medidos en los haces de las partículas y el robot real.

El exponente en esta expresión es la suma de los valores del vector. Cuanto mayor sea el valor de la suma, menor será el peso de esa partícula, queriendo decir que la diferencia entre los valores leídos de los haces estimados del láser desde la posición de esa partícula y los haces del vector láser realmente observado es grande. Cuanto menor sea el valor de la suma de ese vector mayor será el peso de esa partícula.

En la Figura 4.23 se expone lo descrito anteriormente. En este gráfico se puede observar que en el eje x se representa la diferencia entre los valores leídos de los haces estimados de una partícula y los haces del robot realmente observados. Se puede observar que, cuanto mayor es la diferencia de distancias leídas por los haces, menor es la probabilidad, y, viceversa.

Pero para obtener el peso de las partículas, con la ecuación 4.12, lo que se obtiene es la probabilidad no normalizada de las partículas. Se debe normalizar la distribución. La normalización consiste en realizar la suma de todos los valores y posteriormente dividir cada probabilidad entre ese valor de la suma. Así, se consigue el peso de las partículas y que todas ellas juntas sumen el valor de 1.

4.5.2.3 Etapa de movimiento

En la siguiente etapa se aplica el modelo de movimiento a las partículas. Se calcula la distancia recorrida en valores de x e y y el giro realizado por el robot. Posteriormente estos valores se aplican a las coordenadas x e y de las partículas con sus correspondientes orientaciones, θ.

En esta etapa, se coge el 80-90 % de las partículas que tengan un peso mayor que un determinado umbral. Estas partículas son las más probables y serán elegidas para moverse en esta etapa. También se cogen el 20-10 % de las partículas restantes, aunque tengan poca probabilidad. La forma de elección de las partículas es de una forma random por lo que es muy probable que se seleccione una misma partícula más de una vez. También, se les añade un pequeño ruido aleatorio.

Por último, se estima la posición del robot. Para ello, se multiplican las coordenadas x, y y orientación, θ, de cada partícula por su peso.

4.5.2.4 Etapa de remuestreo

Esta etapa se realiza en el momento en el que el algoritmo ha realizado una cantidad de 10 iteraciones. En ese momento, el algoritmo entra en una etapa de evaluación de las partículas existentes. Mira las partículas que poseen el mayor peso, y mediante una elección aleatoria de las partículas selecciona una y compara su peso con un umbral determinado. Esto se realiza para conseguir la generación de nuevas partículas alrededor de las partículas que mayor peso poseen. Si es mayor que el umbral se le asigna de forma random unas coordenadas nuevas x, y y θ.

También, se coge un pequeño porcentaje de partículas y se remuestrean en zonas con poca probabilidad. Ese porcentaje aleatorio es el mecanismo que tiene el algoritmo para explorar nuevas zonas del mapa. Si las partículas con mayor peso están perdidas, entonces estas partículas pueden encontrar por azar alguna zona prometedora donde sea posible que se encuentre el robot.

4.5.3 Resultados obtenidos

Con la plantilla ya funcional, se procedió a la realización de pruebas de localización obteniendo los siguientes resultados. En el primer paso se generan las partículas alrededor de la posición inicial aproximada del robot sobre el mapa (ver Figura 4.24).

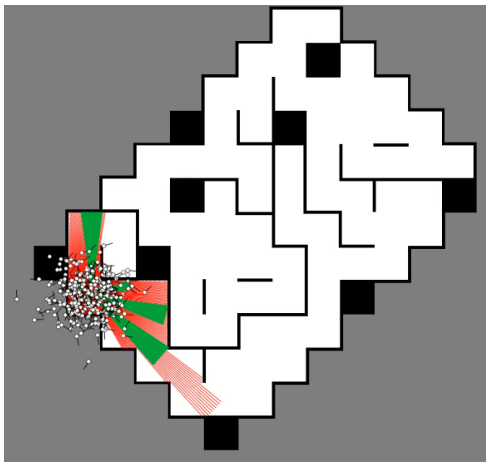


Figura 4.24: Generación de partículas inicial del algoritmo MCL en Robotics-Academy.

Con el paso de las iteraciones del algoritmo, se puede observar cómo las partículas se mueven junto con el robot, dando como resultado que la posición estimada del robot (representada en el mapa con el círculo azul) se encuentre entorno a la posición real del robot ver figuras a continuación: 4.25, 4.26, 4.27.

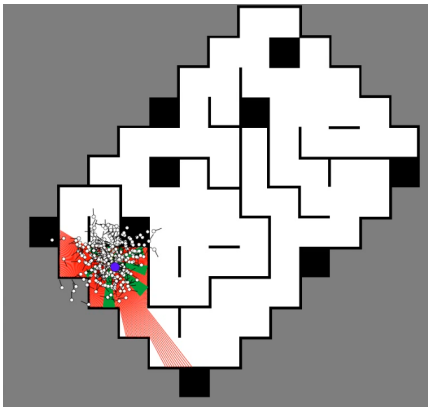


Figura 4.25: Evolución de partículas del algoritmo MCL en Robotics-Academy (1).

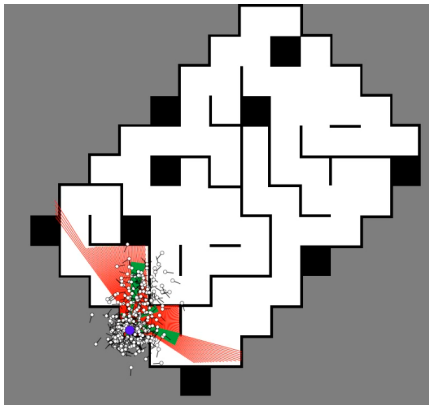


Figura 4.26: Evolución de partículas del algoritmo MCL en Robotics-Academy (2).

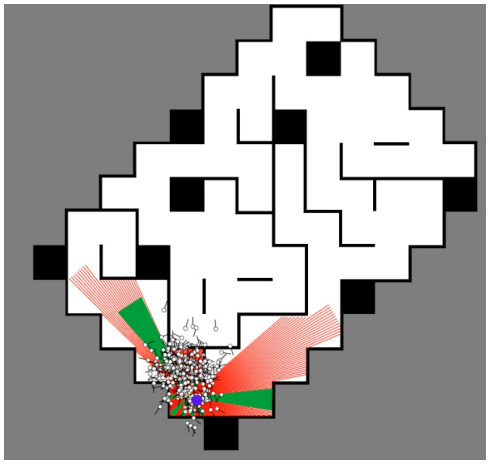


Figura 4.27: Evolución de partículas del algoritmo MCL en Robotics-Academy (3).

Para visualizar mejor los resultados obtenidos se ha grabado un vídeo demostrativo que se ha subido a la página oficial de Robotics-Academy (https://www.youtube.com/channel/UCgmUgpircYAv_QhLQziHJOQ) en YouTube.

Capítulo 5

Conclusiones y líneas futuras

"El único hombre que nunca se equivoca es el que nunca hace nada ."
Johann Wolfgang von Goethe

En este capítulo, en primer lugar se empieza con la comparación de los resultados obtenidos en las dos plataformas. Posteriormente se exponen las conclusiones obtenidas y se proponen futuras líneas de trabajo para este proyecto.

5.1 Comparación de plataformas

En esta sección se abordarán los resultados que se han obtenido en los ejercicios trabajados y que permiten comparar las dos plataformas de enseñanza robótica. Se compararán los ejercicios que han sido realizados en la plataforma MatLab-ROS con los que han sido desarrollados en la plataforma Robotics-Academy.

5.1.1 Práctica de mapeado con posiciones conocidas

El mapeado con posiciones conocidas es el primer ejercicio que se puso en marcha en la plataforma Robotics-Academy y los resultados que se han obtenido se pueden ver en la Figura 5.1. La comparación del mismo mapa obtenido en la plataforma MatLab-ROS es la que se puede ver en la Figura 5.2. El entorno que se estaba mapeando con estas dos plataformas es el que se puede ver en la Figura 5.3.

Como se puede observar en las dos plataformas se obtiene un mapa distinguible y relativamente similar al entorno original. En las dos plataformas se abordó este mapeado sin la introducción de ruido en los sensores de la odometría. Aún así, se puede observar cómo existe una diferencia significativa en la plataforma MatLab-ROS en la que se ha añadido un espacio adicional en gris entre las paredes de la zona central del mapa (remarcada con un círculo rojo) y, además, la parte derecha del mapa se puede apreciar una ligera rotación respecto al entorno original. Estos errores pueden ser debido a algún desajuste de las lecturas de la odometría del robot.

En la plataforma Robotics-Academy, no ocurre esta distorsión del mapa, pero se puede notar un menor contraste de líneas negras, que afecta a la percepción del mapa.

En las dos plataformas existen zonas del mapa en las que no se ha realizado un mapeado correcto de la zona y aparecen zonas en las que no existen paredes cuando sí que deberían existir según el mapa original.

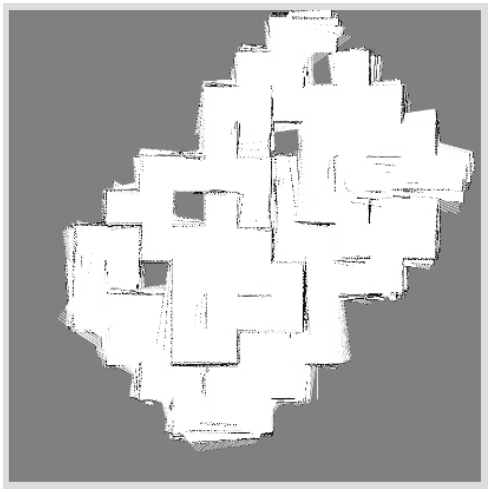


Figura 5.1: Mapa RoboCup obtenido en Robotics-Academy.

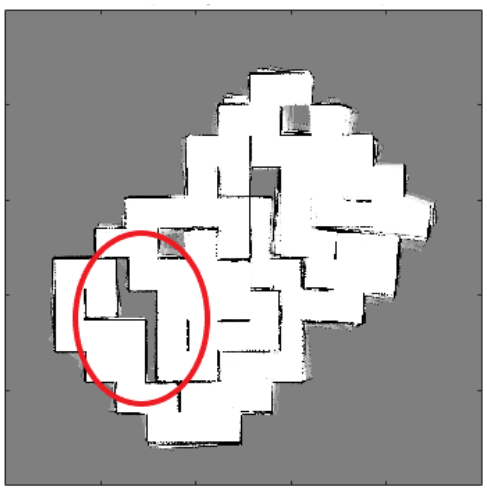


Figura 5.2: Mapa RoboCup obtenido en MatLab-ROS.

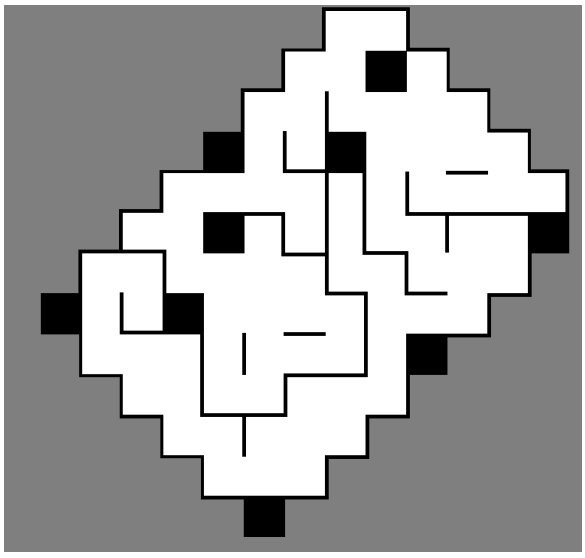


Figura 5.3: Entorno mapeado RoboCup

Para llegar al resultado que se puede ver en la Figura 5.2 en la plataforma MatLab-ROS es necesario que el alumno añada una serie de líneas en el código a una plantilla ya existente para resolver este ejercicio. Mientras que en la plataforma Robotics-Academy el ejercicio se ha diseñado de tal forma que el alumno teleopere el robot desde el navegador y mientras el robot se mueve, en paralelo se realice el mapeado del entorno simulado, con lo que no sería necesario programar ningún código, sino analizar el funcionamiento del mismo. Hablando del movimiento del robot, en la plataforma Robotics-Academy, la teleoperación se realiza desde el propio navegador mientras que en MatLab-ROS, la teleoperación se realiza ejecutando el nodo *teleop_twist_keyboard.py*, es decir, se añade una herramienta más para llegar a trabajar con el ejercicio.

5.1.2 Comparativa de la práctica de auto-localización

En este TFM se ha abordado el problema de la auto-localización en las dos plataformas con dos tipos de algoritmos: MCL y AMCL. Siendo el último la evolución del primero.

El ejercicio de la auto-localización mediante la técnica AMCL se ha abordado en el capítulo 3.2.3

en la plataforma MatLab-ROS, consiguiendo que el robot se localice tanto mediante la localización local como mediante la localización global. En ambos casos se parte de un código fuente existente y privativo, una plantilla, que tiene muchos parámetros para modular su funcionamiento. Para la localización local del robot mediante la técnica AMCL se han utilizado entre 500 y 5000 partículas. Mientras que para la localización global se han utilizado unos valores más altos entre 1000 y 10000 partículas. En los dos casos se obtuvo el resultado esperado: el algoritmo localizaba la ubicación del robot satisfactoriamente.

Por otro lado, se ha abordado el ejercicio de la auto-localización mediante la técnica MCL en el capítulo 4.5 en la plataforma Robotics-Academy. En esta plataforma se han creado las herramientas necesarias para que se pueda operar con la técnica de la localización: visualización de las partículas y su orientación, visualización de la posición estimada del robot, visualización del peso de las partículas mediante rangos de colores.

Además, se ha programado desde cero el algoritmo MCL para que el usuario pueda tener una plantilla inicial con la que trabajar este ejercicio. El algoritmo que se realizado es para una localización local del robot. Debido a la limitación computacional que existe se pueden utilizar un máximo de 500 partículas para una ejecución fluida del algoritmo. Como se ha visto en el capítulo 4.5, el algoritmo sigue de forma satisfactoria la ubicación del robot. Esto también se puede ver en el vídeo proporcionado en el canal de YouTube de Robotics-Academy (Anexo C.2).

Para comparar los resultados obtenidos se presentan las figuras 5.4 para MatLab-ROS y 5.5 para Robotics-Academy.

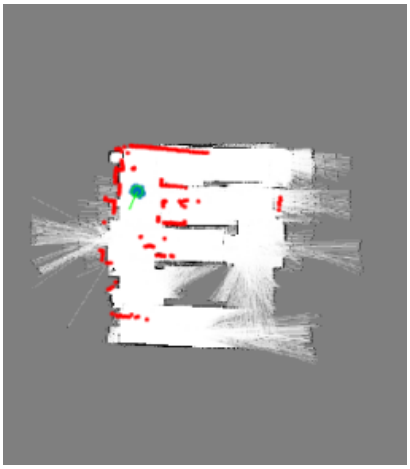


Figura 5.4: Localización del robot AmigoBot en el mapa del aula con la plataforma MatLab-ROS.

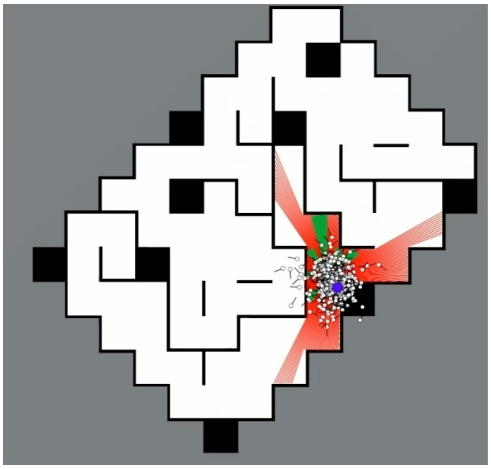


Figura 5.5: Localización del robot AmigoBot en el mapa RoboCup con la plataforma Robotics-Academy.

De estas figuras se extrae que en la localización en la plataforma MatLab-ROS, las partículas convergen en un punto, y se mantienen juntas una vez el algoritmo haya localizado el robot. En la plataforma Robotics-Academy, las partículas se encuentran en una nube alrededor de la posición real de robot, indicando con mayor grado de libertad la posición estimada del robot. Esto se ha realizado para tener un margen de exploración de nuevas zonas que puedan ser mejores para la localización del robot. Además esto ayuda también a la problemática de la localización en el mapa de Robotics-Academy, debido a su gran simetría en muchas de las zonas. Esto provoca la dualidad en las medidas de los sensores, porque dos zonas diferentes pueden dar lugar a medidas semejantes.

También, en la plataforma de MatLab-ROS, los haces del láser se indican mediante impactos en las paredes, mientras que en la plataforma Robotics-Academy se ha programado para que se impriman los

haces del láser según se ven en el simulador STDR. Además, también se ven las lecturas del sonar aunque no se utilicen sus lecturas en este ejercicio.

Centrando la atención en las partículas, una de las diferencias claras es que en la plataforma Robotics-Academy se puede ver fácilmente la orientación de cada una de las partículas gracias a su forma de "*Chupa-Chups*", mientras que en la plataforma de MatLab-ROS no es posible saber la orientación de las partículas. En la plataforma MatLab-ROS tampoco se puede distinguir el peso de cada una de las partículas, ya que todas ellas son representadas mediante el mismo color azul. En la plataforma Robotics-Academy cada una de las partículas tiene su propio color dependiendo del rango de probabilidad en el que se encuentre. Así se puede distinguir qué partículas tienen mayor peso debido a sus mejores lecturas de los sensores (posición semejante a la posición del robot real).

De este modo, en la plataforma MatLab-ROS se tiene un algoritmo AMCL 100 % funcional, pero de código cerrado propiedad de la compañía privada **The MathWorks, Inc.**. Con él se puede trabajar tanto localización global como local. Por otro lado, ahora se tiene la plataforma Robotics-Academy que ofrece un algoritmo de MCL, de código abierto, en el que cada usuario puede contribuir para mejorar la solución propuesta del algoritmo utilizado como plantilla. En esta plataforma, por ahora, sólo se puede utilizar la localización local del robot. El número de partículas que se pueden utilizar en la plataforma Robotics-Academy es mucho menor que las que se pueden utilizar en MatLab-ROS, pero aún teniendo menor número de partículas se puede llegar a solucionar el problema de la localización. El algoritmo MCL, no puede resolver el problema del secuestro (*kidnapping*) del robot. Si la posición cambia de forma discontinua, la localización fallará.

La desventaja que se puede ver en ambos algoritmos es que se debe tener conocimiento previo del mapa en el que se quiere realizar la localización del robot.

5.1.3 Comparativa general

Una comparación general puede residir en la comparación del tiempo dedicado al desarrollo de las soluciones en ambas plataformas. Por ejemplo, en la plataforma MatLab-ROS el tiempo dedicado estimado a cada ejercicio es de aproximadamente 3 semanas. Teniendo en cuenta que en la plataforma se trabajan cinco ejercicios en un tiempo determinado de un cuatrimestre. En la plataforma de Robotics-Academy, se han desarrollado dos ejercicios a lo largo de la duración del TFM. Aproximadamente un año y medio. Evidentemente el tiempo dedicado a cada uno de los ejercicios es diferente debido a la dificultad variante asociada. En Robotics-Academy se ha tardado más en el desarrollo de los ejercicios debido a que se ha tenido que crear la infraestructura de los dos ejercicios y se ha programado la solución de referencia completamente desde cero, sin partir de código existente previo.

Por otro lado, se puede introducir otro factor de comparación: las limitaciones en cada una de las plataformas educativas. Por un lado, en MatLab-ROS una de las limitaciones que se encuentra es que se puede trabajar con las funciones proporcionadas por MatLab y obtener unos resultados, pero no es posible analizar el código que obtiene esos resultados. En Robotics-Academy existe una limitación computacional mayor que en la plataforma MatLab-ROS. En Robotics-Academy no se puede cambiar de mapa del entorno en un ejercicio desarrollado, mientras que en la plataforma MatLab-ROS existe la posibilidad de trabajar con diferentes mapas. En MatLab-ROS como se ha podido ver, en cada uno de los ejercicios es posible configurar multitud de parámetros, mientras que en Robotics-Academy no es posible esa diversidad.

En ambas plataformas, la capacidad de diseñar ejercicios depende mucho de los conocimientos que posea el desarrollador. En Robotics-Academy es necesario tener pleno conocimiento de la estructura de

la plataforma para llegar a desarrollar un ejercicio. En MatLab-ROS, los ejercicios realmente ya están diseñados, pero existe una multitud de posibilidades de trabajar con cada uno de ellos. Es muy fácil coger cualquier plantilla y modificarla según tus necesidades y características del sistema.

La calidad y la comodidad es un factor importante a la hora de la elección con qué plataforma trabajar. Por ejemplo, en MatLab-ROS, desde mi punto de vista, no es cómodo trabajar debido a lo que se puede ver en la Figura 5.6.

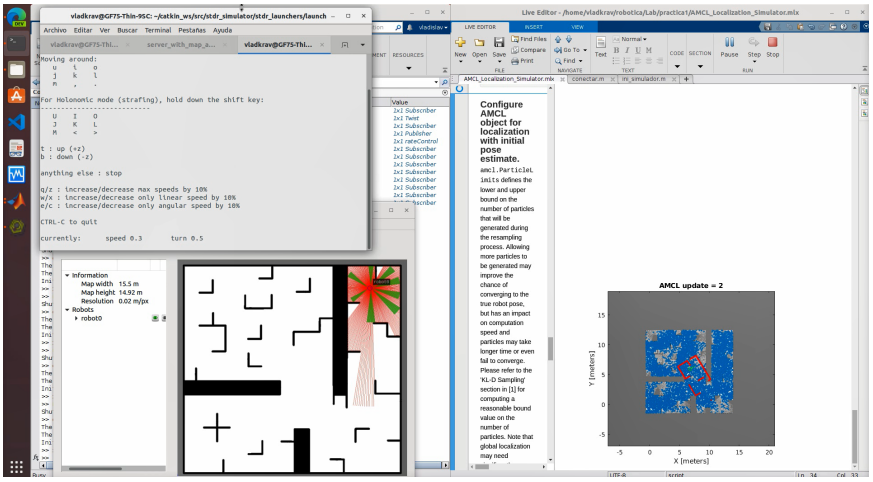


Figura 5.6: Escritorio del sistema operativo con MatLab-ROS

En esta Figura 5.6, podemos ver cómo de sobrecargado se encuentra el escritorio del alumno a la hora de ejecutar un ejercicio en la plataforma MatLab-ROS. Por un lado, se tiene el terminal desde el cual, se ejecuta MatLab en una ventana, en otra ventana se ejecuta ROS y en otra ventana se realiza la teleoperación del robot AmigoBot. A continuación se tiene la ventana del simulador STDR para visualizar el movimiento que se realiza mediante la teleoperación desde el terminal. Además, se deben tener abiertas las ventanas de MatLab para ver el proceso de ejecución del ejercicio programado. De este modo, en primer lugar el alumno debe emplear un determinado tiempo en organizar todas las ventanas, para que no molesten unas a otras. Se puede concluir que o bien el alumno tiene dos o más monitores o el trabajo de la práctica se vuelve incómodo.

En mi opinión, como alumno de las dos plataformas, es más cómodo trabajar con Robotics-Academy que evita toda esta aglomeración de ventanas y se tiene una ventana del navegador para la programación del ejercicio, espacio para el simulador, etc.

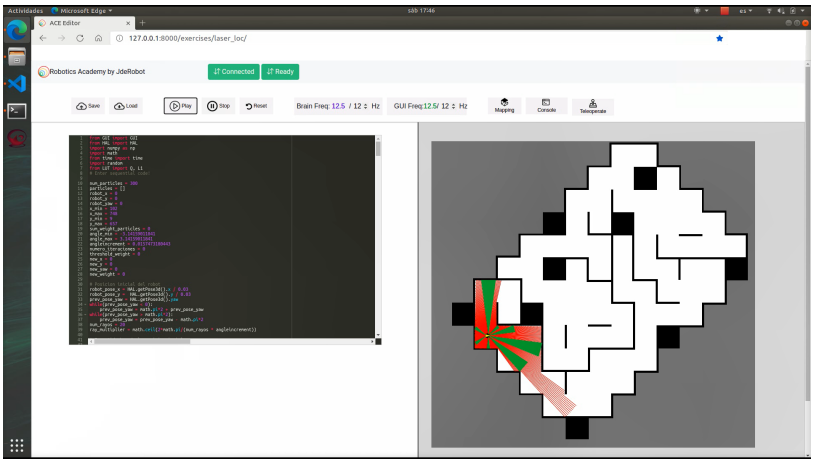


Figura 5.7: Escritorio del sistema operativo con Robotics-Academy

5.2 Conclusiones

A lo largo del desarrollo de este TFM se han visto dos diferentes plataformas de aprendizaje robótico: MatLab-ROS y Robotics-Academy.

En primer lugar, se han utilizado diferentes técnicas para trabajar con robots móviles tales como "mapeado con posiciones conocidas", SLAM, VFH, AMCL, PRM. Todas estas técnicas se han investigado y estudiado bajo la plataforma MatLab-ROS mediante la toolbox *Robotics System* y el simulador STDR. Se han analizado detalladamente los parámetros que intervienen en cada uno de los algoritmos, se ha programado la solución a estos algoritmos, se han obtenido diferentes resultados para el robot AmigoBot tanto para el robot simulado como para el robot real. Finalizado el estudio de estas técnicas bajo esta plataforma, se consiguieron los conocimientos necesarios para proceder con la realización de la segunda parte de este proyecto en la plataforma Robotics-Academy.

La segunda parte del proyecto consistía en la implementación de dos ejercicios en la plataforma Robotics-Academy. Los ejercicios que se han integrado en esta plataforma de aprendizaje han sido "mapeado con posiciones conocidas" y el algoritmo MCL. Para ambos se ha programado toda la infraestructura necesaria, la plantilla de cada ejercicio, su página web, etc. y se ha programado una solución de referencia desde cero.

Con el ejercicio "mapeado con posiciones conocidas" implementado en Robotics-Academy se probó su funcionamiento, obteniendo unos resultados bastante similares a los obtenidos en la plataforma MatLab-ROS. Uno de los inconvenientes que se puede observar a primera vista es que en MatLab-ROS es posible introducir ruido a los parámetros del robot, mientras que en Robotics-Academy, no es posible. Por otro lado, como ventajas se puede decir que se ha conseguido integrar en un solo lugar todas las herramientas necesarias para el ejercicio, cuando en la plataforma MatLab-ROS era necesario tener abiertas gran cantidad de ventanas para trabajar completamente.

Desarrollar e integrar el ejercicio MCL en la plataforma Robotics-Academy supuso un gran reto debido a que fue necesario el estudio detallado de este algoritmo, y, aparte fue necesaria su máxima optimización para que no supusiera una carga computacional demasiado elevada para su procesamiento en esta plataforma. Finalmente, se consiguió su integración obteniendo con la solución de referencia unos resultados satisfactorios en la auto-localización del robot.

Durante este proyecto se han estudiado prácticamente desde cero diferentes tipos de lenguajes de programación (Python, JavaScript, HTML, CSS) y herramientas como Docker. Supuso un gran esfuerzo el estudio en paralelo de los lenguajes de programación utilizados por la plataforma Robotics-Academy y su utilización para el desarrollo de las funcionalidades implementadas y sobre todo de los algoritmos. Se han manejado en este TFM muchas tecnologías heterogéneas: tecnologías web (servidor Django, *WebSocket*, *frontend*...), tecnologías robóticas (ROS, STDR, algoritmos de mapeado de auto-localización), tecnologías DevOpc (contenedores Docker...), todas ellas necesarias para la plataforma Robotics-Academy.

Los objetivos planteados en la sección 1.5, durante el primer capítulo, se pueden revisitar y analizar en qué medida se han alcanzado. Uno de los objetivos era el estudio de las dos plataformas de aprendizaje MatLab-Ros y Robotics-Academy. Durante los capítulos 3 y 4 se ha profundizado en el estudio e investigación de estas plataformas, para desarrollar los ejercicios que se han visto. Por lo tanto, se puede decir que este objetivo ha sido completado satisfactoriamente.

Otro objetivo era la correcta implementación del robot AmigoBot en la plataforma Robotics-Academy, lo que se ha conseguido utilizando y desarrollando los diferentes drivers para los motores, láser y sonar, utilizando correctamente los topics proporcionados por el robot, tanto simulado como real. Así, cualquier desarrollador en esta plataforma podrá hacer uso de los archivos y seguir trabajando con el robot

AmigoBot, por ejemplo en otros ejercicios.

Por último, se propuso el desarrollo de una serie de algoritmos en la plataforma Robotics-Academy que sirvieran de soluciones de referencia para estos dos ejercicios. En párrafos anteriores se ha comentado que se ha conseguido programándolos completamente desde cero.

5.3 Líneas futuras

Este proyecto ha abarcado una parte de los ejercicios estudiados en la plataforma MatLab-ROS. Una posible línea futura de trabajo e investigación sería la implementación de los algoritmos de localización y mapeado simultáneos, navegación local y global, tales como SLAM, VFH, PRM, dentro de la plataforma de aprendizaje Robotics-Academy.

Otra posible línea de trabajo es la continua mejora y optimización de los algoritmos, es decir las soluciones de referencia, en los ejercicios ya creados e implementados en la nueva plataforma. Un buen ejemplo de ello, puede ser la plantilla puesta a disposición del usuario del algoritmo MCL. El código siempre puede ser mejorado y optimizado. Además, como se ha comentado en la conclusión, en la plataforma MatLab-ROS es posible la introducción de ruido en el sistema del robot simulado, que no es posible en la plataforma Robotics-Academy. De este modo, una de las posibilidades podría ser el desarrollo de esta característica.

Además, también se podría implementar la posibilidad de elección del mapa del entorno por el usuario. Esto es posible en la plataforma de MatLab-ROS, cambiando los parámetros en ROS y en MatLab, pero, en Robotics-Academy habría que entrar en los archivos del núcleo del ejercicio, lo que, para el usuario final no es muy práctico.

Por otro lado, como se comentó en la introducción, en la asignatura de Robótica Móvil (202003) de la Universidad de Alcalá, se trabaja con la plataforma MatLab-ROS, que por los sistemas utilizados empieza a considerarse obsoleta. Por lo tanto, sería interesante la introducción de la plataforma de aprendizaje Robotics-Academy en las prácticas de laboratorio de esta asignatura. Así los alumnos podrían realizar la comparación de estas dos plataformas, con todas las ventajas que supone trabajar con la plataforma Robotics-Academy (ultima versión de sistema operativo, multiplataforma, etc.).

También, como línea futura de trabajo, podría ser interesante trabajar en la mejora y renovación de Robotics-Academy, en su continua actualización de todas sus partes (Docker, SO, etc.).

Se ha elaborado una breve encuesta a los alumnos de la asignatura de Robótica Móvil de la UAH del año 2021/2022 que van a cursar esta asignatura y tendrán la suficiente experiencia para obtener una comparación y sacar algunas conclusiones sobre las dos plataformas. También se realizará esta encuesta al profesorado de la asignatura y a los profesores relacionados con la robótica. A los encuestados se les realizará una pequeña encuesta mediante Google Forms con las siguientes preguntas, algunas de ellas se pueden visualizar en la Figura 5.8:

1. ¿Cual ha sido el nivel de dificultad de la plataforma MatLab-ROS?
2. ¿Cual ha sido el nivel de dificultad de la plataforma Robotics-Academy?
3. En el ejercicio de Mapeado con posiciones conocidas ¿Qué le parecen los resultados obtenidos en la plataforma MatLab-ROS?
4. En el ejercicio de Mapeado con posiciones conocidas ¿Qué le parecen los resultados obtenidos en la plataforma Robotics-Academy?

5. En el ejercicio de auto localización ¿Qué le parecen los resultados obtenidos en la plataforma MatLab-ROS?

6. En el ejercicio de auto localización ¿Qué le parecen los resultados obtenidos en la plataforma Robotics-Academy?

7. ¿Qué plataforma tiene mejor documentación?

8. ¿Que plataforma le parece más didáctica?

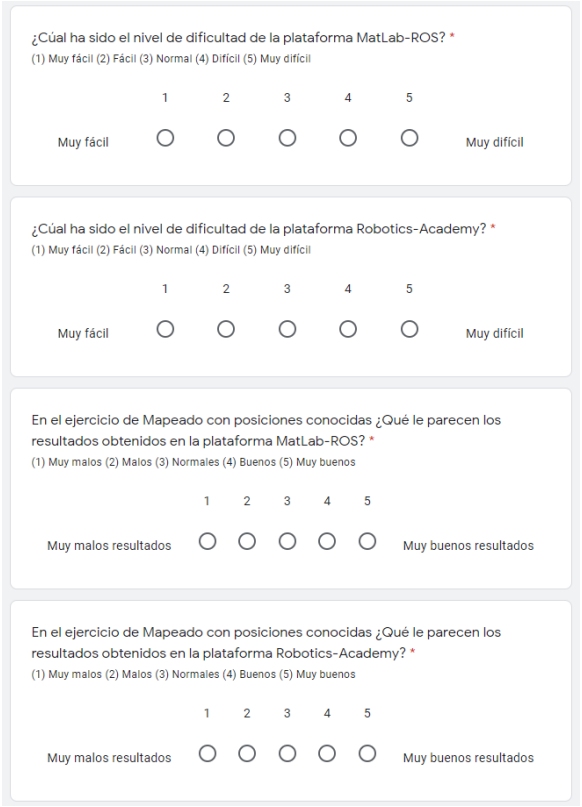


Figura 5.8: Encuesta desarrollada en Google Forms.

El enlace para rellenar esta encuesta se puede encontrar en la siguiente URL: Google Forms (*https://docs.google.com/forms/d/e/1FAIpQLSc0Fy976natqtHbymodt_bDeLYqC0sLZpKfQt0RPY3gkVjBew/viewform*)

En este documento no se exponen los resultados debido a que estos se obtendrán, como línea futura, después de la presentación de este trabajo. Serán útiles para conocer los resultados obtenidos en este trabajo, útiles como retroalimentación para la asignatura de Robótica Móvil y serán buenos para la plataforma Robotics-Academy. Además los resultados de la encuesta nos ayudarán en la edición de una publicación en una revista.

Bibliografía

[1] Stanford Artificial Intelligence Laboratory et al. Robotic operating system. [Online]. Available: https://www.ros.org

[2] "Gnu general public license, version 3," http://www.gnu.org/licenses/gpl.html, June 2007, last retrieved 2020-01-01.

[3] E. M. Jimenez Jojoa, E. C. Bravo, and E. B. Bacca Cortes, "Tool for experimenting with concepts of mobile robotics as applied to children's education," *IEEE Transactions on Education*, vol. 53, no. 1, pp. 88–95, Feb 2010.

[4] F. Cerezo and F. Sastrón, "Laboratorios virtuales y docencia de la automática en la formación tecnológica de base de alumnos preuniversitarios," *Revista Iberoamericana de Automática e Informática industrial*, vol. 12, no. 4, pp. 419–431, 2015. [Online]. Available: https://polipapers.upv.es/index.php/RIAI/article/view/9343

[5] A. Gil, O. Reinoso, J. M. Marin, L. Paya, and J. Ruiz, "Development and deployment of a new robotics toolbox for education," *Computer Applications in Engineering Education*, vol. 23, no. 3, pp. 443–454, 2015. [Online]. Available: https://onlinelibrary.wiley.com/doi/abs/10.1002/cae.21615

[6] N. Aliane, "Teaching fundamentals of robotics to computer scientists," *Computer Applications in Engineering Education*, vol. 19, no. 3, pp. 615–620, 2011. [Online]. Available: https://onlinelibrary.wiley.com/doi/abs/10.1002/cae.20342

[7] R. Gonzalez, C. Mahulea, and M. Kloetzer, "A matlab-based interactive simulator for mobile robotics," in *2015 IEEE International Conference on Automation Science and Engineering (CASE)*, 2015, pp. 310–315.

[8] P. Corke, "A robotics toolbox for matlab," *IEEE Robotics Automation Magazine*, vol. 3, no. 1, pp. 24–32, 1996.

[9] B. Gates, "A robot in every home," *Scientific American*, vol. 296, no. 1, pp. 58–65, 2007.

[10] M. Montemerlo, N. Roy, S. Thrun, D. Hähnel, C. Stachniss, and J. Glover, "CARMEN – the carnegie mellon robot navigation toolkit," http://carmen.sourceforge.net, 2002.

[11] M. Munich, J. Ostrowski, and P. Pirjanian, "Ersp: a software platform and architecture for the service robotics industry," in *2005 IEEE/RSJ International Conference on Intelligent Robots and Systems*, 2005, pp. 460–467.

[12] C. A. Jara, F. A. Candelas, J. Pomares, and F. Torres, "Java software platform for the development of advanced robotic virtual laboratories," *Computer Applications in Engineering Education*, vol. 21, no. S1, pp. E14–E30, 2013. [Online]. Available: https://onlinelibrary.wiley.com/doi/abs/10.1002/cae.20542

[13] J. Guzmán, M. Berenguel, F. Rodríguez, and S. Dormido, "An interactive tool for mobile robot motion planning," *Robotics and Autonomous Systems*, vol. 56, no. 5, pp. 396–409, 2008. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0921889007001443

[14] E. Fabregas, G. Farias, S. Dormido-Canto, M. Guinaldo, J. Sánchez, and S. D. Bencomo, "Platform for teaching mobile robotics," *Journal of Intelligent & Robotic Systems*, vol. 81, no. 1, pp. 131–143, 2016.

[15] D. S. Touretzky, "Robotics for computer scientists: what's the big idea?" *Computer Science Education*, vol. 23, no. 4, pp. 349–367, 2013.

[16] L. Guyot, N. Heiniger, O. Michel, and F. Rohrer, "Teaching robotics with an open curriculum based on the e-puck robot, simulations and competitions," in *Proceedings of the 2nd International Conference on Robotics in Education. Vienna, Austria*, 2011.

[17] R. Detry, P. Corke, and M. Freese, "Trs: An open-source recipe for teaching/learning robotics with a simulator," 2014.

[18] J. M. Cañas, E. Perdices, L. García-Pérez, and J. Fernández-Conde, "A ros-based open tool for intelligent robotics education," *Applied Sciences*, vol. 10, no. 21, 2020. [Online]. Available: https://www.mdpi.com/2076-3417/10/21/7419

[19] G. Van Rossum and F. L. Drake Jr, *Python reference manual*. Centrum voor Wiskunde en Informatica Amsterdam, 1995.

[20] MATLAB, *9.9.0.1592791 (R2020b)*. Natick, Massachusetts: The MathWorks Inc., 2020.

[21] J. M. C. Plaza, M. Á. C. Quevedo, and V. Matellán, "Uso de simuladores en docencia de robótica móvil," *Revista Iberoamericana de Tecnologías del Aprendizaje*, 2009.

[22] N. Koenig and A. Howard, "Design and use paradigms for gazebo, an open-source multi-robot simulator," in *2004 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)(IEEE Cat. No. 04CH37566)*, vol. 3. IEEE, 2004, pp. 2149–2154.

[23] Webots, "http://www.cyberbotics.com," open-source Mobile Robot Simulation Software. [Online]. Available: http://www.cyberbotics.com

[24] O. Michel, "Webots: Professional mobile robot simulation," *Journal of Advanced Robotics Systems*, vol. 1, no. 1, pp. 39–42, 2004. [Online]. Available: http://www.ars-journal.com/International-Journal-of-Advanced-Robotic-Systems/Volume-1/39-42.pdf

[25] S. Carpin, M. Lewis, J. Wang, S. Balakirsky, and C. Scrapper, "Usarsim: a robot simulator for research and education," in *Proceedings 2007 IEEE International Conference on Robotics and Automation*, 2007, pp. 1400–1405.

[26] B. Gerkey, R. T. Vaughan, A. Howard *et al.*, "The player/stage project: Tools for multi-robot and distributed sensor systems," in *Proceedings of the 11th international conference on advanced robotics*, vol. 1. Citeseer, 2003, pp. 317–323.

[27] R. Vaughan, "Stage: A multiple robot simulator. i nstitute for robotics and intelligent systems technical report iris-00-393," *University of Southern California*, 2000.

[28] github, "Github," 2020. [Online]. Available: https://github.com/

[29] H. Durrant-Whyte and T. Bailey, "Simultaneous localization and mapping: part i," *IEEE Robotics Automation Magazine*, vol. 13, no. 2, pp. 99–110, June 2006.

[30] T. Bailey and H. Durrant-Whyte, "Simultaneous localization and mapping (slam): part ii," *IEEE Robotics Automation Magazine*, vol. 13, no. 3, pp. 108–117, Sep. 2006.

[31] F. Dellaert, D. Fox, W. Burgard, and S. Thrun, "Monte carlo localization for mobile robots," in *Proceedings 1999 IEEE International Conference on Robotics and Automation (Cat. No.99CH36288C)*, vol. 2, May 1999, pp. 1322–1328 vol.2.

[32] D. Fox, "Kld-sampling: Adaptive particle filters and mobile robot localization," *Advances in Neural Information Processing Systems (NIPS)*, vol. 14, no. 1, pp. 26–32, 2001.

[33] S. Kullback and R. A. Leibler, "On information and sufficiency," *The annals of mathematical statistics*, vol. 22, no. 1, pp. 79–86, 1951.

[34] J. Borenstein and Y. Koren, "The vector field histogram-fast obstacle avoidance for mobile robots," *IEEE Transactions on Robotics and Automation*, vol. 7, no. 3, pp. 278–288, June 1991.

[35] D. Fox, W. Burgard, and S. Thrun, "The dynamic window approach to collision avoidance," *IEEE Robotics Automation Magazine*, vol. 4, no. 1, pp. 23–33, March 1997.

[36] J. Borenstein and Y. Koren, "Real-time obstacle avoidance for fast mobile robots," *IEEE Transactions on Systems, Man, and Cybernetics*, vol. 19, no. 5, pp. 1179–1187, Sep. 1989.

[37] I. Ulrich and J. Borenstein, "Vfh+: Reliable obstacle avoidance for fast mobile robots," in *Proceedings. 1998 IEEE international conference on robotics and automation (Cat. No. 98CH36146)*, vol. 2. IEEE, 1998, pp. 1572–1577.

[38] ——, "Vfh/sup*: Local obstacle avoidance with look-ahead verification," in *Proceedings 2000 ICRA. Millennium Conference. IEEE International Conference on Robotics and Automation. Symposia Proceedings (Cat. No. 00CH37065)*, vol. 3. IEEE, 2000, pp. 2505–2511.

[39] The GIMP Development Team. Gimp. [Online]. Available: https://www.gimp.org

[40] R. C. Coulter, "Implementation of the pure pursuit path tracking algorithm," Carnegie-Mellon UNIV Pittsburgh PA Robotics INST, Tech. Rep., 1992.

[41] L. E. Kavraki, P. Svestka, J.-C. Latombe, and M. H. Overmars, "Probabilistic roadmaps for path planning in high-dimensional configuration spaces," *IEEE transactions on Robotics and Automation*, vol. 12, no. 4, pp. 566–580, 1996.

[42] Django Software Foundation. Django. [Online]. Available: https://djangoproject.com

[43] "Información sobre gnu/linux en wikipedia," http://es.wikipedia.org/wiki/GNU/Linux [Último acceso 1/noviembre/2013].

[44] L. Lamport, *LaTeX: A Document Preparation System, 2nd edition.* Addison Wesley Professional, 1994.

[45] D. Flanagan, *JavaScript : the definitive guide.* Beijing; Sebastopol, CA: O'Reilly, 2011. [Online]. Available: http://www.amazon.com/JavaScript-Definitive-Guide-Activate-Guides/dp/0596805527/ref=sr_1_6?s=books&ie=UTF8&qid=1319803043&sr=1-6

[46] J. Duckett, *HTML & CSS: design and build websites.* Wiley Indianapolis, IN, 2011, vol. 15.

[47] K. Patel, "Incremental journey for world wide web: introduced with web 1.0 to recent web 5.0–a survey paper," *International Journal of Advanced Research in Computer Science and Software Engineering*, vol. 3, no. 10, 2013.

[48] D. Merkel, "Docker: lightweight linux containers for consistent development and deployment," *Linux journal*, vol. 2014, no. 239, p. 2, 2014.

Apéndice A

Manual de usuario

"El arte de escribir consiste en decir mucho con pocas palabras."
Antón Chéjov

A.1 Instalación de Robotic Operating System (ROS)

Para poner en marcha el entorno de MatLab y ROS, en primer lugar se debe instalar el framework ROS para el sistema operativo Ubuntu 18.04 que se está usando. Posteriormente se procederá con la configuración del espacio de trabajo.

Para la instalación de ROS se deben seguir los siguientes pasos explicados en la página oficial de ROS para la versión de Ubuntu correspondiente. En nuestro caso la versión es: *melodic*:

1. Se configura el sistema operativo para aceptar software de la organización packages.ros.org.

Listing A.1: Comando para aceptar software de la organización *packages.ros.org*

```
sudo sh -c 'echo "deb http://packages.ros.org/ros/ubuntu$(lsb_release -sc)
main" > /etc/apt/sources.list.d/ros-latest.list\'
```

2. Se añaden las claves:

Listing A.2: Comando para añadir las claves de *packages.ros.org*

```
curl -s https://raw.githubusercontent.com/ros/rosdistro/master/ros.asc |
sudo apt-key add -
```

3. Para llevar a cabo la instalación, nos aseguramos de tener todos los paquetes actualizados:

Listing A.3: Comando para comprobar la existencia de actualizaciones para el sistema

```
sudo apt update
```

4. Se procede a la instalación completa de ROS, junto con rqt, rviz, etc.

Listing A.4: Comando para instalar ROS

```
sudo apt install ros-melodic-desktop-full
```

5. Para añadir automáticamente las variables del entorno de ROS a una sesión bash cada vez que se lanza un shell se debe ejecutar el siguiente comando:

Listing A.5: Comando para añadir las variables del entorno de ROS

```
echo "source /opt/ros/melodic/setup.bash" >> ~/.bashrc
source ~/.bashrc
```

6. También, se instalan diferentes dependencias que son herramientas de uso frecuente con ROS:

Listing A.6: Instalación de diferentes dependencias

```
sudo apt install python3-rosdep python3-rosinstall
python3-rosinstall-generator python3-wstool build-essential
```

7. Para finalizar con la instalación de ROS, antes de poder utilizar las herramientas de ROS, se necesita instalar e inicializar *rosdep*. Este paquete permite instalar fácilmente las dependencias del sistema para el origen que se desea compilar y es necesario para ejecutar algunos componentes principales en ROS.

Listing A.7: Instalación del paquete *rosdep*

```
sudo apt install python3-rosdep
sudo rosdep init
rosdep update
```

De esta manera ya se tendría instalado ROS en el sistema operativo Ubuntu 18.04 con el que se ha desarrollado este TFM. Esta instalación sirve tanto para el entorno MatLab-ROS como para el entorno de Robotics-Academy.

A.2 Configuración de la teleoperación del robot

En el entorno de MatLab-ROS, para mover el robot por el mapa se ha utilizado el nodo de ROS *teleop_twist_keyboard.py*. Este nodo se encuentra dentro del paquete *teleop_twist_keyboard*. Para hacer funcionar este nodo, se debe ejecutar la siguiente instrucción en la terminal de Ubuntu:

Listing A.8: Comando para trabajar con el nodo *teleop_twist_keyboard.py*

```
rosrun teleop_twist_keyboard teleop_twist_keyboard.py cmd_vel:=/robot0/cmd_vel_speed:=0.3
    _turn:=0.5
```

Con esta instrucción se realiza una redirección de topics cambiando el topic en el que se escribe por defecto (*cmd_vel*) por el del robot simulado (*/robot0/cmd_vel*). Además, se modifican los parámetros de la velocidad lineal(*_speed*) y la velocidad angular (*_turn*). Estos parámetros establecen la velocidad

máxima que se puede comandar al robot. Como se puede ver, se han establecido 0.3 m/s para la velocidad lineal y 0.5 rad/s de velocidad angular.

El robot real también se puede teleoperar con el mismo nodo, estableciendo el topic correspondiente al robot real, y, de igual modo, estableciendo las velocidades máximas tanto lineal como angular.

Apéndice B

Herramientas y recursos

Las herramientas hardware que han sido necesarias para desarrollar este proyecto son las siguientes:

- Un ordenador compatible.
- Robot AmigoBot.

Por otro lado, las herramientas software que se han necesitado para abordar este proyecto son las siguientes:

- Sistema operativo GNU/Linux [43]
- Procesador de textos LaTeX [44]
- ROS [1]
- Servicio web para el control de versiones GitHub [28]
- Lenguajes Python, JavaScript, CSS, HTML [19] [45] [46] [47]
- MatLab y "ROS Toolbox" [20]
- Docker [48]

Otros recursos necesarios para la elaboración del proyecto han sido:

- Herramientas y cables necesarios para la conexión del robot.

Apéndice C

Enlaces y Vídeos

En esta sección se presentan los enlaces a las páginas web de los ejercicios que se han implementado en la plataforma Robotics-Academy. También, se presentan los enlaces a los vídeos demostrativos que se han realizado sobre estos ejercicios. Los enlaces de los vídeos apuntan a la página web de YouTube, al canal oficial de la plataforma Robotics-Academy (https://www.youtube.com/channel/UCgmUgpircYAv_QhLQziHJOQ).

C.1 Ejercicio de mapeado con posiciones conocidas

- Página web: http://jderobot.github.io/RoboticsAcademy/exercises/MobileRobots/laser_mapping
- Vídeo: https://www.youtube.com/watch?v=obHhJ-_Y96c

C.2 Ejercicio auto-localización con MCL

- Página web: Aún no disponible.
- Vídeo: https://www.youtube.com/watch?v=y7rBPpV2NdI

Universidad de Alcalá
Escuela Politécnica Superior

ESCUELA POLITECNICA
SUPERIOR

Universidad
de Alcalá