Memoria Práctica 2: Construcción de mapas

Jordi Florit Ensenyat
Josep Gabriel Fornes Reynes
Jorge Elías García

Universidad Politécnica de Madrid
Asignatura: Robots Autónomos

11 de diciembre de 2025

Resumen

Este documento detalla el proceso de implementación y evaluación de un sistema de construcción de mapas de ocupación (*occupancy grids*) en el entorno de simulación CoppeliaSim [1]. Se presentan las decisiones de diseño tomadas, los experimentos y los resultados obtenidos. Se define también el controlador codificado, así como los diferentes algoritmos para la construcción del mapa, como Slam y otras implementaciones propias. Finalmente, se extraen conclusiones sobre el comportamiento del sistema y posibles líneas de trabajo futuro.

1. Introducción

A diferencia de la práctica anterior, donde el foco residía en la navegación puramente reactiva para la evasión de obstáculos, este trabajo introduce el problema del mapeo probabilístico y requiere una estrategia de exploración mucho más robusta que garantice la cobertura del escenario sin que el robot quede atrapado en mínimos locales.

Para abordar estos desafíos, se ha realizado una evolución significativa en la arquitectura del sistema, migrando hacia un esquema distribuido basado en **ROS 2 (Robot Operating System)** [2]. Esta decisión de diseño permite desacoplar la adquisición de datos, el control y el procesamiento del mapa en nodos independientes que se comunican mediante paso de mensajes, acercando la simulación a un despliegue robótico real.

El desarrollo del proyecto se ha articulado en tres ejes principales:

1. **Exploración Robusta y Reactiva:** Se ha diseñado un nuevo nodo de control (`ExploradorAutonomo`) que sustituye la lógica difusa anterior por un algoritmo basado en la lectura de un escáner láser (LiDAR). Para superar las limitaciones detectadas en la práctica 1, este controlador implementa una visión segmentada en cinco sectores para detectar mejor las esquinas y, crucialmente, incorpora mecanismos de recuperación ("maniobras de pánico" o marcha atrás) que permiten al robot desencajarse físicamente si la distancia a los obstáculos se vuelve crítica.
2. **Mapeo Probabilístico (Log-Odds):** Se ha implementado desde cero un algoritmo de construcción de mapas de ocupación utilizando el método de *Log-Odds* (logaritmo de la razón de probabilidades). Este enfoque, ejecutado en el nodo `MapeadorProbabilistico`, permite actualizar la creencia sobre el estado de cada celda (libre u ocupada) de forma aditiva y computacionalmente eficiente. Para la actualización de las celdas atravesadas por el haz del láser, se ha integrado una versión vectorizada del algoritmo de trazado de líneas de Bresenham.
3. **Validación y Comparativa:** Además de la implementación propia, se ha configurado e integrado el paquete estándar de navegación `slam_toolbox` de ROS 2. Esto ha permitido realizar una comparativa técnica entre los mapas generados por nuestro algoritmo ad-hoc ("No SLAM") y una solución de SLAM (*Simultaneous Localization and Mapping*) completa, analizando las diferencias en la calidad de la reconstrucción y la gestión de la deriva odométrica.

A lo largo de esta memoria se detallarán la fundamentación matemática del algoritmo de Log-Odds utilizado, la arquitectura de nodos implementada en ROS 2 y el análisis de los resultados experimentales obtenidos tanto en el escenario simple como en entornos modificados con obstáculos complejos.

2. Arquitectura del Sistema

El desarrollo de esta práctica se sustenta en una arquitectura distribuida y modular. A continuación, se describen los fundamentos del sistema operativo robótico utilizado, las herramientas específicas empleadas y su integración con el entorno de simulación.

2.1. Fundamentos de ROS 2

ROS 2 (Robot Operating System) no es un sistema operativo en el sentido tradicional, sino un *middleware* de código abierto que proporciona servicios estándar para la robótica, como la abstracción de hardware, el control de dispositivos de bajo nivel y el paso de mensajes entre procesos.

La arquitectura de ROS se basa en un grafo de computación donde el procesamiento se realiza en nodos (*nodes*) que pueden recibir, enviar y multiplexar mensajes de sensores, control, estado, planificación y actuadores. La comunicación principal se realiza mediante un modelo de publicación/suscripción a través de canales denominados tópicos (*topics*), lo que permite desacoplar la captación de datos de su procesamiento.

Para esta práctica, la elección de ROS 2 frente a la API remota directa (ZMQ) se justifica por la necesidad de modularidad y escalabilidad. Permite ejecutar el controlador de navegación y el algoritmo de mapeo como procesos independientes y concurrentes, facilitando la depuración y acercando la simulación a un despliegue en un robot físico real.

2.2. Herramientas y Paquetes Utilizados

La implementación se ha realizado en **Python** utilizando la librería cliente oficial **rclpy**. Esta librería proporciona la API estándar y permite a los desarrolladores crear nodos de ROS y utilizar funcionalidades básicas del sistema robótico (como la comunicación entre procesos mediante tópicos y servicios) de forma idiomática en Python Los principales componentes del ecosistema ROS y librerías externas utilizados son:

- **Tipos de Mensajes Estándar:** Se han empleado los mensajes definidos por el consorcio ROS para garantizar la interoperabilidad entre nodos propios y paquetes de terceros:
  - `sensor_msgs/msg/LaserScan`: Para la transmisión de datos del LiDAR (rangos y ángulos).
  - `nav_msgs/msg/Odometry`: Para la estimación de la pose del robot.
  - `geometry_msgs/msg/Twist`: Para el envío de comandos de velocidad.
  - `nav_msgs/msg/OccupancyGrid`: Estándar para la representación de mapas de ocupación, se ha usado en concreto para crear un visualizador del SLAM.
- **SLAM Toolbox [3]:** Se ha integrado este paquete estándar de navegación para generar una verdad terreno (*ground truth*) comparativa y validar el desempeño de nuestro algoritmo de mapeo probabilístico propio, viene integrado con el propio ROS2.
- **Visualización y Análisis:**
  - **RViz2:** Visualizador 3D nativo de ROS 2. Se ha utilizado para la depuración de las transformadas (TF) y la visualización estándar del mapa generado por `slam_toolbox`.
  - **Matplotlib [4]:** Se ha implementado un visualizador personalizado en Python. Este script demuestra la flexibilidad del sistema al renderizar la matriz de ocupación en tiempo real, permitiendo comparar visualmente la salida del algoritmo Log-Odds propio con la solución estándar.

2.3. Integración con CoppeliaSim

La integración entre el entorno de simulación CoppeliaSim y el ecosistema ROS 2 se articula mediante el plugin oficial `simROS2`. A diferencia de la práctica anterior, donde el control se ejercía desde un cliente externo, en esta arquitectura el robot Pioneer P3DX opera como un nodo ROS nativo. Esta capacidad se consigue mediante un *child script* embebido en lenguaje Lua, diseñado para ejecutar tres funciones críticas en cada ciclo de simulación:

1. **Publicación de Sensores (`/scan`):** El script lee los sensores de proximidad y empaqueta las lecturas simulando un escáner láser Hokuyo. Los datos se transmiten como mensajes de tipo `sensor_msgs/msg/LaserScan`, cubriendo un campo de visión de 240°.

2. **Odometría y Transformadas (`/odom` y `tf`):** El simulador actúa como fuente de *Ground Truth*. Calcula la posición y orientación absolutas del robot en la escena y las publica tanto en el tópico de odometría como en el árbol de transformadas (TF).

3. **Suscripción a Comandos (`/cmd_vel`):** El nodo escucha los comandos de velocidad lineal y angular y, mediante un modelo cinemático diferencial inverso, calcula y aplica las velocidades de rotación correspondientes a las ruedas motrices izquierda y derecha.

Para satisfacer los requisitos específicos de cada parte de la práctica, se han desarrollado dos versiones de este script de integración:

- **Configuración para Mapeo Propio ("No SLAM"):** En este enfoque, el script prioriza la entrega directa de datos para nuestro algoritmo de *Log-Odds*. La odometría perfecta del simulador se utiliza directamente para situar los rayos del láser en la matriz de ocupación, asumiendo que no existe error de localización.

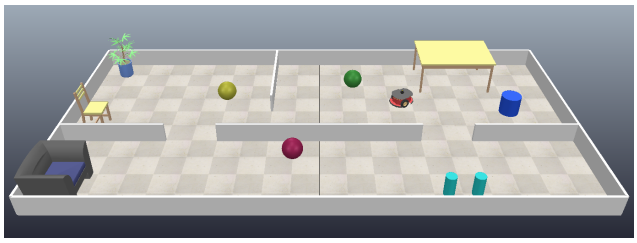

Figura 1: Escenario de pruebas en el que el robot se desplaza y construye el mapa de ocupación.

- **Configuración para SLAM ("SLAM Toolbox"):** Para la integración con el paquete `slam_toolbox`, se requirió un refinamiento matemático en la publicación del láser. Los nodos de SLAM estándar son estrictos con la consistencia temporal y geométrica de los mensajes. Por ello, el script modificado calcula los incrementos angulares (`angle_increment`) con precisión de punto flotante basándose en el número exacto de rayos recibidos, evitando errores de consistencia que provocarían el rechazo de los paquetes por parte del stack de navegación.

Además, para ambas configuraciones anteriores se ha utilizado el mismo escenario con el fin de disponer de una base de comparación común. La escena empleada se muestra en la Figura 1, donde se incluyen distintos obstáculos que incrementan la dificultad del robot a la hora de construir el mapa de ocupación. Estos obstáculos no solo permiten comprobar si el controlador evita correctamente los objetos, sino que también facilitan la verificación de que el robot los incorpora adecuadamente en el mapa de ocupación.

2.4. Diagrama de Arquitectura

La Figura 2 muestra un esquema simple, detallando el flujo de datos entre el simulador y los nodos de control y mapeo.

3. Estrategia de Navegación y Exploración

A continuación, se presentan las estrategias de navegación y exploración empleadas para la construcción del mapa de ocupación.

3.1. Sistema Controlador

En primer lugar, se ha implementado el sistema controlador, encargado del movimiento del robot por la escena, evitando colisiones y situaciones de bloqueo. De este modo, se garantiza la recogida de información suficiente y bien distribuida por todo el escenario, condición necesaria para que el mapa de ocupación resultante se ajuste adecuadamente a la estructura real del entorno.

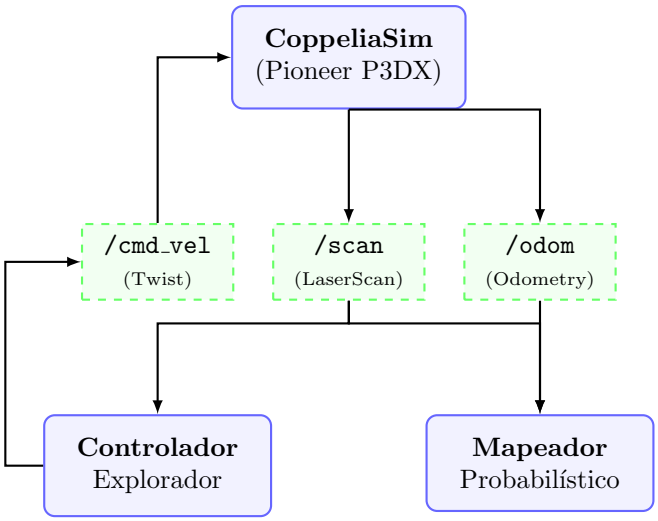

Figura 2: Diagrama de arquitectura de nodos y tópicos en ROS 2.

3.1.1. Descripción Sistema Controlador Implementado

El controlador se ejecuta de forma reactiva a partir de las lecturas del sensor láser. En cada iteración, primero **limpia los datos** del escáner (sustituyendo valores extremadamente pequeños o grandes por una distancia máxima razonable para evitar medidas erróneas) y, a continuación, **divide el haz láser en cinco sectores angulares**: derecha, frontal-derecha, frontal, frontal-izquierda e izquierda. De cada uno de estos sectores se extrae la **distancia mínima medida**, obteniendo así una representación compacta del espacio libre alrededor del robot que se utilizará como entrada para la toma de decisiones de movimiento. Con esta información, el controlador distingue entre dos escenarios: **presencia de obstáculo en la zona frontal y camino libre.** Esta decisión se toma comparando las distancias mínimas de los sectores frontal y fronto-laterales con un umbral de seguridad: si alguna de ellas es inferior a dicho umbral, se considera que hay un obstáculo delante.

En primer lugar, en caso de **presencia de un obstáculo**, el controlador distingue entre una situación **muy crítica** y otra **menos severa**.

- En el **caso crítico** (distancia global por debajo de un umbral reducido o tiempo de bloqueo excesivo), el robot ejecuta una maniobra de **marcha atrás**, avanzando ligeramente en sentido contrario y sin giro para alejarse de la zona congestionada.
- En caso de una **situación no tan extrema**, el comportamiento es más conservador: el robot **se detiene** y comienza a **girar sobre sí mismo** en

la dirección con mayor espacio libre, con el objetivo de encontrar una nueva orientación desde la que continuar avanzando y rodear el obstáculo sin necesidad de retroceder.

En segundo lugar, cuando el **camino frontal está libre**, el controlador entra en un modo de **exploración guiada** en el que el robot avanza mientras corrige su trayectoria para mantenerse aproximadamente centrado en pasillos o zonas estrechas.

- Si el robot se encuentra **demasiado cerca de la pared izquierda** (distancia lateral izquierda por debajo de un umbral), avanza ligeramente más despacio y aplica un **pequeño giro hacia la derecha**, separándose de esa pared.
- Si, por el contrario, está **demasiado cerca de la pared derecha**, el comportamiento es simétrico: reduce suavemente la velocidad y aplica un **giro hacia la izquierda** para recentrarse.
- Cuando existe **espacio suficiente a ambos lados**, el controlador ordena un **avance recto** a velocidad de crucero, favoreciendo una exploración eficiente del entorno sin correcciones adicionales.

3.2. Comparación con el controlador de la práctica anterior

A diferencia de la práctica anterior, en esta **no se ha implementado el mismo controlador basado en lógica difusa**. Esto se debe a que, al aplicarlo en escenas más complejas que las de la práctica anterior, el robot no era capaz de desplazarse correctamente por toda la zona, quedando atrapado con frecuencia en esquinas o pasillos estrechos. Una posible causa es que dicho controlador toma sus decisiones únicamente a partir de las lecturas instantáneas de los sensores, y no incorpora estados internos ni mecanismos específicos de desatasco, por lo que tiende a repetir patrones locales en configuraciones complicadas. En cambio, el **nuevo controlador** reactivo, basado en el láser y con variables de estado para fijar la dirección de giro y detectar bloqueos prolongados, proporciona **trayectorias más estables y permite una exploración mucho más completa del entorno**. Asimismo, los mecanismos explícitos de desatasco, como la maniobra de **marcha atrás** en situaciones críticas, han resultado clave para evitar bloqueos recurrentes y mejorar la cobertura global del mapa.

4. Construcción del Mapa de Ocupación (*Occupancy Grid Mapping*)

Se han utilizado dos mapeadores de ocupación distintos. Por un lado, se ha implementado un ***occupancy grid* propio** que representa cada celda mediante el método de ***log-odds***: en lugar de almacenar directamente la probabilidad de estar ocupada, cada celda guarda el valor $l = \log\left(\frac{p}{1-p}\right)$, donde p es la probabilidad de ocupación. De este modo, celdas con $l > 0$ indican mayor evidencia de estar ocupadas ($p > 0.5$) y con $l < 0$ mayor evidencia de estar libres ($p < 0.5$), mientras que $l = 0$ corresponde al estado desconocido ($p = 0.5$). Las nuevas observaciones del LiDAR se integran sumando incrementos positivos o negativos de log-odds, lo que implementa de forma eficiente la actualización bayesiana de la probabilidad de ocupación de cada celda. Por otro lado, se ha empleado el **módulo `slam_toolbox` de ROS**, el cual genera también un mapa de ocupación probabilístico a partir de los escaneos y, además, estima la pose del robot en dicho mapa, implementando un algoritmo de **SLAM 2D**.

Una diferencia clave entre ambos enfoques es que el mapeador propio asume que la **posición del robot** (x, y, θ) **en el mundo es conocida** en cada instante (proporcionada directamente por el simulador), de modo que sólo se encarga de integrar las medidas del LiDAR en el mapa. En cambio, `slam_toolbox` realiza **simultáneamente la estimación de la trayectoria del robot y la construcción del mapa,** resolviendo de forma conjunta el problema de localización y mapeo característico del **SLAM**. Esta diferencia entre ambos mapeadores supondrá distinciones claras en los resultados obtenidos, como se podrá apreciar a lo largo del documento.

4.1. Mapeador Propio

El mapeador propio representa el entorno como una **rejilla 2D** formada por pequeñas celdas cuadradas de tamaño fijo. Cada celda corresponde a una porción del espacio del mundo (por ejemplo, unos pocos centímetros cuadrados) y almacena la creencia de si esa porción está **ocupada, libre o sigue siendo desconocida**, en lugar de guardar directamente la probabilidad p. La relación entre ambos valores viene dada por

$$l = \log\left(\frac{p}{1-p}\right), \qquad p = \frac{1}{1+e^{-l}}.$$

Al inicio, todas las celdas se consideran desconocidas ($p = 0.5$), lo que equivale a $l = 0$. A medida que el robot recibe escaneos del LiDAR y conoce su pose (x, y, θ), cada rayo del sensor se proyecta sobre la rejilla: las celdas que el rayo atraviesa antes de llegar a la medida se interpretan como **evidencia de espacio libre** y se actualizan sumando un **incremento negativo fijo de log-odds** (lo que desplaza p hacia valores menores de ocupación), mientras que la celda donde el rayo impacta en un obstáculo se interpreta como **evidencia de ocupación** y se actualiza con un **incremento positivo de log-odds** (desplazando p hacia valores superiores). En el caso de rayos que alcanzan el rango máximo del sensor (sin impacto claro), se actualiza

únicamente el espacio libre a lo largo del rayo. Sin embargo, para evitar borrar obstáculos bien consolidados por lecturas anteriores, el mapeador solo aplica la actualización de "libre" en aquellas celdas cuyo valor de log-odds **no sea ya excesivamente alto** (es decir, celdas que siguen siendo dudosas). Este proceso se repite de forma acumulativa para todos los escaneos, de modo que las **celdas atravesadas reiteradamente por rayos tienden a consolidarse como libres**, y las que reciben **impactos consistentes tienden a consolidarse como ocupadas**, saturando los valores de log-odds en un rango acotado para evitar problemas numéricos. Así, el mapa se va construyendo de **manera incremental**, integrando de forma bayesiana y compacta toda la información proporcionada por el sensor a lo largo del recorrido del robot.

El mapa de ocupación se mantiene internamente en una matriz bidimensional (array de `NumPy` [5]) que se actualiza tras cada escaneo del LiDAR. Tras cada actualización, esta matriz se convierte en probabilidades de ocupación y se almacena, de modo que un script de visualización puede cargar dicho array y representarlo, asignando a cada celda un nivel de gris en función de su probabilidad de estar ocupada. Así, a medida que el robot explora y el array se actualiza, la imagen se va refinando y el mapa emerge progresivamente en pantalla.

4.1.1. Resultados

Para obtener un resultado más representativo, se ha optado por realizar un vídeo del proceso de construcción del mapa de ocupación, en el que se observa el movimiento del robot en la escena y la generación en tiempo real de dicho mapa. Para ver el vídeo de la simulación, consulte el siguiente enlace: **ver vídeo de la simulación con mapeador propio**.

En el vídeo se aprecia cómo el robot se desplaza con fluidez por toda la escena, evitando los obstáculos y capturándolos para la construcción del mapa. En la Figura 3, que muestra el mapa final construido por el robot, se observa que se han reconstruido los principales objetos de la escena, como el sillón situado en la parte inferior, las distintas bolas repartidas por la superficie, la silla de la sala superior izquierda (a pesar de que el robot no ha llegado a entrar en ella) o la mesa ubicada en la parte superior derecha.

Figura 3: Mapa de ocupación realizado por el mapeador propio.

4.2. `Slam Toolbox` de ROS

Como segunda aproximación, se ha utilizado una implementación de **SLAM** ya existente en ROS, concretamente el paquete `slam_toolbox`. Este paquete implementa un algoritmo de **SLAM 2D** (Simultaneous Localization and Mapping) con láser, que estima simultáneamente el **mapa del entorno** y la **trayectoria del robot** a partir de las lecturas del LiDAR y de la odometría disponible.

A alto nivel, `slam_toolbox` mantiene una **estimación de la pose** del robot en el mapa (posición y orientación) y un **grafo de poses** que representa la trayectoria recorrida. Cada vez que se recibe un nuevo escaneo láser, el sistema:

- **Asocia** la nueva nube de puntos con el mapa acumulado hasta el momento, ajustando la pose actual del robot mediante técnicas de registro.
- **Actualiza** el grafo de poses añadiendo un nuevo nodo o refinando los existentes.
- **Detecta y corrige cierres de lazo** (*loop closures*) cuando el robot vuelve a pasar por zonas previamente visitadas, optimizando el grafo para reducir el error acumulado.
- **Reconstruye el mapa 2D** de ocupación a partir de la trayectoria optimizada y de todos los escaneos registrados.

Es importante remarcar que, a diferencia de un sistema con *ground truth*, `slam_toolbox` **no conoce la posición real del robot**: únicamente dispone de lecturas ruidosas del LiDAR y de una odometría imperfecta. La localización y el mapa se estiman de forma conjunta, lo que hace que el problema sea más complejo que en el caso del mapeador propio, donde la posición era conocida.

En la práctica se han probado dos configuraciones distintas de `slam_toolbox`, correspondientes a los modos de ejecución síncrono y asíncrono. En primer lugar, se utilizó el lanzamiento síncrono estándar:

```
ros2 launch slam_toolbox
online_sync_launch.py
slam_params_file:=./slam_sync.yaml
```

En esta configuración, el nodo de SLAM procesa los escaneos láser de forma más acoplada al ciclo de recepción de datos, empleando el fichero de parámetros `slam_sync.yaml`. Dicho fichero establece, entre otros aspectos, un alcance efectivo del láser más conservador (`max_laser_range = 5.0`) y umbrales de actualización más agresivos (`minimum_travel_distance = 0.05`, `minimum_travel_heading = 0.05`), lo que se traduce en actualizaciones más frecuentes del mapa ante pequeños movimientos del robot.

A continuación se evaluó el modo de ejecución asíncrono:

```
ros2 launch slam_toolbox
online_async_launch.py
slam_params_file:=./slam_async.yaml
```

En este caso, el fichero `slam_async.yaml` configura un alcance del sensor más acorde con el escenario de trabajo (`max_laser_range = 20.0`), incrementos mínimos de movimiento más exigentes (`minimum_travel_distance = 0.1`, `minimum_travel_heading = 0.1`) y un `transform_timeout` más ajustado. Además, se habilitan explícitamente mecanismos de cierre de lazo (`do_loop_closing = true`) con parámetros específicos para la detección de bucles, de modo que el sistema puede corregir parte de la deriva cuando el robot vuelve a pasar por zonas ya exploradas.

Tras comparar experimentalmente ambas configuraciones, se observó que la versión **síncrona** tendía a generar mapas más ruidosos, con mayor presencia de paredes dobles y pequeños artefactos debidos a la combinación del ruido de la simulación y las actualizaciones muy frecuentes del mapa. El modo **asíncrono**, con parámetros más restrictivos en la actualización y un alcance de láser mayor, produjo **mapas de ocupación más estables y coherentes con la geometría real del escenario.** Por este motivo, todos los resultados presentados en el apartado de experimentos y en la comparativa final se han obtenido utilizando la configuración asíncrona (`online_async_launch.py` con `slam_async.yaml`).

4.2.1. Resultados

Al igual que en los resultados anteriores, se proporciona un vídeo sobre el proceso de construcción del mapa de ocupación con el mapeador SLAM, utilizando la herramienta RViz2 para visualizar el proceso en tiempo real. Para poder visualizar el vídeo de la simulación, consulte el siguiente enlace: **ver vídeo de la simulación con el mapeador SLAM**.

En la Figura 4 se muestra el mapa de ocupación generado por el mapeador SLAM. En comparación con el resultado anterior, la diferencia principal es que esta construcción es menos fiable, ya que la representación de los objetos y de la estructura de la escena no es exacta (aparición de paredes dobles, objetos espurios, artefactos, etc.). A pesar de que en este caso no se emplea odometría para que el robot obtenga una estimación precisa de su posición, el sistema ha sido capaz de reconstruir el escenario de manera razonable, acercándose más a una simulación en condiciones reales.

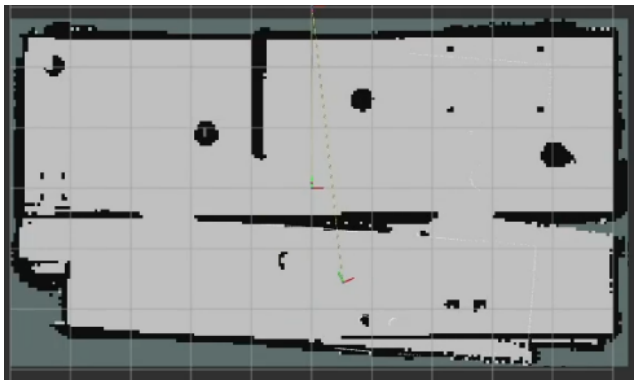

Figura 4: Mapa de ocupación realizado por el mapeador SLAM.

4.3. Comparación entre algoritmos

A partir de los resultados obtenidos con ambos mapeadores, se pueden extraer varias diferencias relevantes. En primer lugar, el **mapeador propio genera un mapa de ocupación mucho más limpio y definido** (las paredes aparecen continuas, los objetos están mejor delimitados y apenas se observan artefactos). Esto es coherente con el hecho de que el algoritmo dispone en todo momento de la pose exacta del robot proporcionada por el simulador, por lo que únicamente debe integrar las lecturas del LiDAR en la rejilla de ocupación sin tener en cuenta incertidumbres de localización.

Por el contrario, el mapa generado por *SLAM Toolbox* presenta ciertas **inconsistencias visibles**, como paredes dobles o la aparición de "fantasmas" en zonas donde el robot ha pasado varias veces. Estos efectos se deben a que el algoritmo de SLAM **estima de forma conjunta la trayectoria del robot y el mapa** a partir de medidas ruidosas del láser y de una odometría imperfecta. Cuando la deriva acumulada es significativa o el cierre de lazo no resulta suficientemente informativo, el mapa resultante refleja esa incertidumbre.

Sin embargo, esta aparente desventaja del mapeador SLAM debe interpretarse con cautela. Aunque el mapa sea menos preciso que el obtenido con odometría perfecta, el enfoque de SLAM es mucho más realista desde el punto de vista robótico, ya que en un robot físico no se dispone de la pose verdadera, y cualquier sistema de mapeo debe enfrentarse a errores de detecciones y de movimiento. En ese sentido, el resultado de *SLAM Toolbox* demuestra que, incluso sin conocer la pose exacta del robot, es posible reconstruir de manera

razonable la geometría global del escenario.

En cuanto al coste de implementación, el mapeador propio resulta más sencillo de configurar, pues sólo requiere ajustar los parámetros de la rejilla (resolución, tamaño) y los incrementos de log-odds asociados a celdas libres u ocupadas. *SLAM Toolbox*, en cambio, incorpora numerosos parámetros (modelos de ruido, umbrales de cierre de lazo, frecuencias de actualización, etc.) que requieren un proceso de ajuste más cuidadoso para obtener mapas de buena calidad.

5. Conclusiones y Trabajo Futuro

En esta práctica se ha diseñado e implementado un sistema completo de construcción de mapas de ocupación en un entorno de simulación basado en CoppeliaSim y ROS 2. La arquitectura propuesta desacopla el controlador de navegación, el mapeador probabilístico y las herramientas de visualización, acercando el funcionamiento del sistema al de una plataforma robótica real.

Desde el punto de vista de la navegación, el nuevo controlador reactivo basado en el sensor láser ha demostrado ser significativamente más robusto que el controlador difuso utilizado en la práctica anterior. La segmentación del haz en sectores, junto con los mecanismos explícitos de desatasco (marcha atrás y giros de recuperación), ha permitido al robot explorar el escenario de forma más completa, reduciendo las situaciones de bloqueo en pasillos estrechos o esquinas.

En cuanto al mapeo, la implementación propia basada en log-odds ha permitido construir mapas de ocupación precisos e interpretables. El uso de la odometría perfecta del simulador elimina la incertidumbre de localización y facilita el análisis del algoritmo de actualización bayesiana de las celdas. Por otro lado, la integración de *SLAM Toolbox* ha hecho posible comparar esta solución ad-hoc con un algoritmo de SLAM 2D completo, observando cómo la falta de pose verdadera introduce artefactos en el mapa pero, al mismo tiempo, hace el sistema más representativo de un despliegue real.

En conjunto, los experimentos realizados muestran la fuerte dependencia entre la calidad de la exploración y la calidad del mapa: un buen mapeador no compensa un controlador deficiente, y viceversa. La combinación de un controlador robusto con un mapeo probabilístico bien parametrizado resulta esencial para obtener mapas útiles para tareas posteriores de planificación o navegación.

Trabajo futuro

Como posibles líneas de trabajo futuro se plantean las siguientes:

- **Planificación de exploración**: integrar un módulo de planificación (por ejemplo, basado en fronteras de exploración) que seleccione de forma activa las zonas aún desconocidas del mapa, en lugar de depender únicamente de un controlador reactivo.
- **Mejora del mapeador propio**: extender el mapeador de log-odds hacia un esquema de SLAM completo, incorporando la estimación de la trayectoria del robot y técnicas de corrección de deriva (como cierres de lazo simples o ajuste de grafo).
- **Escenarios más complejos y dinámicos**: evaluar el comportamiento del sistema en entornos con múltiples habitaciones, pasillos más estrechos u obstáculos móviles, analizando el impacto de estos factores en la estabilidad del controlador y en la calidad del mapa.

Estas líneas de mejora permitirían consolidar el trabajo realizado en esta práctica y avanzar hacia sistemas de navegación y mapeo más robustos y próximos a aplicaciones reales.

Referencias

[1] "Página web de coppeliasim." `https://www.coppeliarobotics.com/`.

[2] "Documentación de ros2." `https://docs.ros.org/en/humble/`.

[3] S. Macenski, "slam_toolbox: Slam for lifelong mapping and localization in ROS 2." `https://docs.ros.org/en/ros2_packages/humble/api/slam_toolbox/`.

[4] J. D. Hunter, "Matplotlib: A 2d graphics environment," *Computing in Science & Engineering*, vol. 9, no. 3, pp. 90–95, 2007.

[5] C. Harris, K. Millman, S. Walt, R. Gommers, P. Virtanen, D. Cournapeau, E. Wieser, J. Taylor, S. Berg, N. Smith, R. Kern, M. Picus, S. Hoyer, M. Kerkwijk, M. Brett, A. Haldane, J. Río, M. Wiebe, P. Peterson, and T. Oliphant, "Array programming with numpy," *Nature*, vol. 585, pp. 357–362, 09 2020.