UNIVERSIDAD MIGUEL HERNÁNDEZ DE ELCHE

ESCUELA POLITÉCNICA SUPERIOR DE ELCHE

GRADO EN INGENIERÍA ELECTRÓNICA Y AUTOMÁTICA INDUSTRIAL

"NAVEGACIÓN AUTÓNOMA DE UN ROBOT HUSKY EQUIPADO CON UN SENSOR LIDAR EN ENTORNOS SEMIESTRUCTURADOS "

TRABAJO FIN DE GRADO

Enero - 2026

AUTOR: David Mateo Rodríguez de Liébana

DIRECTOR/ES: Arturo Gil Aparicio

ÍNDICE

ÍNDICE DE FIGURAS

ÍNDICE DE TABLAS

1. INTRODUCCIÓN

1.1 ROBÓTICA MÓVIL

En la actualidad, la robótica móvil se ha consolidado como una de las ramas más dinámicas y prometedoras de la tecnología, gracias a su capacidad para combinar autonomía y sistemas de percepción avanzados. Desde robots de servicio en hospitales [1] y almacenes [2] hasta vehículos autónomos en la industria del transporte [3], los avances en este campo están cambiando la manera en que interactuamos con el entorno y ejecutamos tareas. La integración de sensores y algoritmos de navegación permite que estos sistemas se desplacen con gran precisión y puedan realizar las tareas en entornos no estructurados.

Sin embargo, a pesar de su progreso, la robótica móvil todavía presenta importantes desafíos cuando opera fuera de entornos estructurados. En un entorno estructurado se conocen exactamente las posiciones de los obstáculos ya que es un entorno estable predefinido; por ello es posible determinar con antelación el camino que puede seguir el robot. En cambio, en entornos no estructurados, donde el entorno es cambiante y no está controlado, el uso de la robótica móvil aún sigue siendo un desafío. En la Figura 1, se puede apreciar la diferencia entre un entorno estructurado predefinido y un entorno no estructurado cambiante.

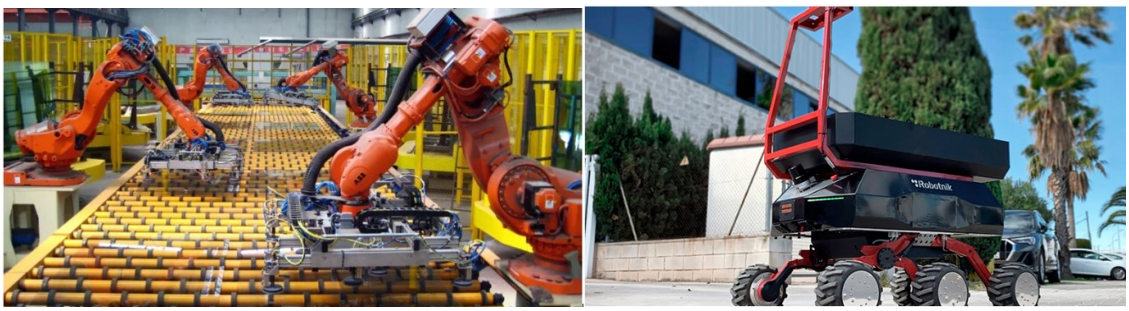

Figura 1. Comparación ilustrativa entre un entorno estructurado (fábrica) y no estructurado (entorno natural o urbano).

A pesar de los avances alcanzados en la robótica móvil, el principal desafío radica en su funcionamiento dentro de entornos no estructurados, donde las condiciones cambian de forma impredecible y no es posible conocer de antemano la ubicación de obstáculos ni las rutas óptimas de desplazamiento. Esta falta de estabilidad y control limita la autonomía de los robots y dificulta su aplicación en escenarios reales como espacios urbanos, entornos naturales o áreas de emergencia. Por lo tanto, el problema que se aborda consiste en desarrollar soluciones de navegación y percepción que permitan a los robots móviles operar de manera eficiente y segura en entornos no estructurados.

La robótica es una rama de la ciencia y la ingeniería que combina conocimientos de distintas ramas como la mecánica, la electrónica, la informática o la automatización, con el fin de diseñar, construir, programar y utilizar robots. Su objetivo principal es crear sistemas capaces de realizar tareas de manera autónoma o semiautónoma, ya sea en entornos industriales, científicos o domésticos. Por tanto, un robot no es una cosa concreta ya que pueden variar mucho dependiendo de su función.

Según su forma, los robots se pueden dividir en dos tipos principales: los de base fija, dentro de los cuales están los robots tipo serie y tipo paralelo, y los robots móviles.

Los robots tipo serie poseen una única cadena cinemática, su estructura mecánica está formada por una cadena de eslabones conectados uno tras otro, similar a un brazo humano como se muestra en la parte izquierda de la Figura 2. Cada articulación depende de la anterior, y los movimientos se transmiten a lo largo de toda la cadena hasta llegar al extremo, generalmente un efector final como una pinza. Este tipo de robots ofrece gran flexibilidad y un amplio rango de trabajo.

Los robots tipo paralelo, en cambio, como se observa en la parte derecha de la Figura 2, cuentan con varias cadenas cinemáticas conectadas en paralelo que mueven un mismo efector final. A diferencia de los robots en serie, en estos cada articulación trabaja de forma simultánea para mover el extremo. Son más rápidos y precisos en tareas repetitivas, aunque su rango de movimiento es más limitado.

Figura 2. Ejemplo de robot tipo serie (izquierda) vs robot tipo paralelo (derecha).

Además de esos dos tipos de robots, existen los robots móviles que son los utilizados en este estudio. Los robots móviles son aquellos diseñados para desplazarse de un lugar a otro en el espacio, sin estar fijos a una base. A diferencia de los robots de base fija, estos poseen sistemas de locomoción que les permite moverse y navegar de forma controlada pudiendo adaptarse a los cambios que pueda haber en el entorno. La función principal de los robots móviles no es solo manipular objetos, sino también transportarse e interactuar con el entorno, por ello se usan tanto en aplicaciones donde es necesario el movimiento de forma autónoma.

Para poder realizar las tareas correctamente, los robots móviles deben tener varios componentes esenciales:

- Sistema de locomoción, que permite el desplazamiento. Dependiendo del entorno, el robot puede usar ruedas, orugas (como los empleados en vehículos militares con orugas) o patas.
- Sensores, encargados de recopilar la información sobre el entorno. Entre los más usados están las cámaras, los sensores basados en ultrasonidos, los sensores infrarrojos como el LiDAR y los sistemas GNSS como el GPS.
- Sistemas de control o procesamiento, que permiten tomar decisiones en tiempo real a partir de la información proporcionada por los sensores,

como un ordenador o incluso el uso de inteligencia artificial para planificar rutas [4].

Esas son las partes fundamentales de los robots móviles, pero dentro de este tipo existe una gran variedad. Entre los que se encuentran los robots humanoides, que simulan la forma y algunos comportamientos de las personas y son utilizados, por ejemplo, para atención al cliente [5]; los robots caminantes, que imitan el desplazamiento de animales o personas bien a 2 patas o a 4 y son usados, por ejemplo, para tareas de rescate en zonas peligrosas tras terremotos o derrumbes [6]; e incluso los vehículos autónomos [3].

1.2 SENSORES

Como se ha mencionado anteriormente, los sensores son una parte fundamental de los robots móviles, ya que estos necesitan información del entorno por el que se desplazan. Los sensores más usados para conocer lo que rodea al robot son los basados en ultrasonidos (como el sonar), las cámaras, los que usan GNSS como el GPS y el LiDAR.

1.2.1 SENSORES BASADOS EN ULTRASONIDOS

Los sensores basados en ultrasonidos son dispositivos que emplean ondas sonoras de alta frecuencia, es decir, ondas cuya frecuencia está por encima del límite audible. Un sensor de ultrasonidos tiene un emisor y un receptor: el emisor genera las ondas ultrasónicas mientras que el receptor recibe las ondas reflejadas tras chocar con el objeto.

Al recibir la onda que se ha reflejado en el objeto, es posible determinar la distancia al objeto, ya que esta se obtiene multiplicando la velocidad del sonido por el tiempo transcurrido entre la emisión y la recepción dividido por 2 (debido a que la onda realiza un trayecto de ida y vuelta). De este modo se puede cuantificar la cercanía de un obstáculo, en la Figura 3 se ve ilustrado cómo funciona este tipo de sensor.

Este tipo de sensores resultan útiles porque al funcionar por emisión de ondas, puede funcionar en ambientes con poca iluminación y son relativamente baratos y resistentes. Sin embargo, uno de los mayores inconvenientes es que las distancias medidas pueden verse afectadas por superficies irregulares, que provocan múltiples reflexiones y pueden hacer que la distancia calculada sea mayor que la real.

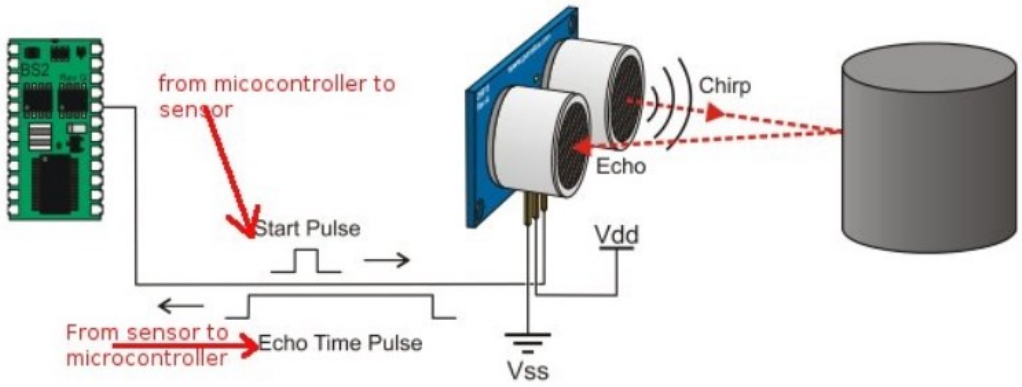

Figura 3. Esquema de funcionamiento de un sensor de ultrasonidos.

1.2.2 CÁMARAS

Otro tipo de sensores muy usados en la robótica móvil son las cámaras. Las cámaras son sensores ópticos que capturan la información visual del entorno. A diferencia de otros sensores que proporcionan únicamente distancias o posiciones de obstáculos, las cámaras ofrecen una gran cantidad de datos en forma de imágenes, incluso en 3D, que permite analizar todo lo que se encuentra alrededor del robot.

El funcionamiento de la cámara se basa en un sensor fotosensible, generalmente CCD o CMOS, compuesto por una matriz de fotodiodos. Cuando estos fotodiodos reciben la luz reflejada por los objetos del entorno, generan una señal eléctrica que indica la cantidad de luz captada. A partir de estas señales es posible obtener una imagen en blanco y negro. Para generar una imagen de color hay diferentes métodos, en la Figura 4 se presenta el método más común, el uso de filtros de Bayer, que permiten obtener la intensidad de cada color primario RGB (rojo, verde y azul) en cada fotodiodo.

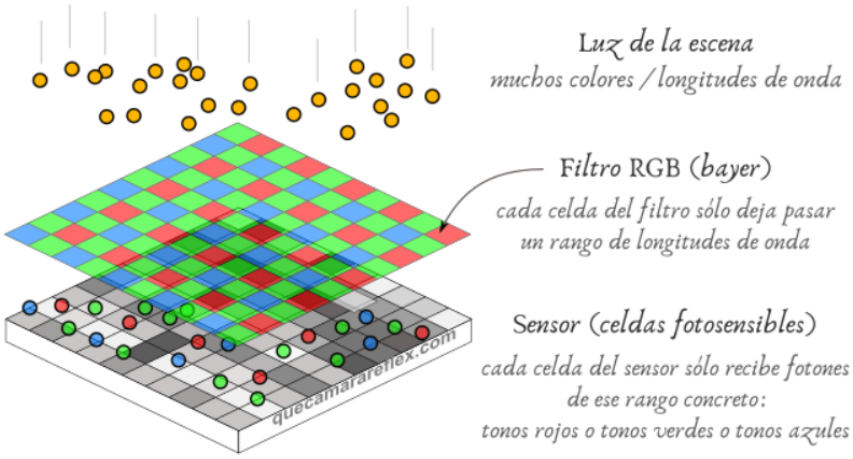

Figura 4. Diagrama del funcionamiento del filtro Bayer en una cámara.

Una vez obtenida la imagen, ya sea en color o blanco y negro, se usan técnicas de visión por computador para analizarla, por ejemplo, para detectar bordes, reconocer objetos o segmentar regiones. De esta forma el robot puede extraer información relevante sobre el entorno para determinar por dónde moverse.

El uso de cámaras presenta varias ventajas, como la gran cantidad de información que proporcionan sobre el ambiente y la posibilidad de detectar y clasificar objetos. Sin embargo, también presenta limitaciones, ya que su rendimiento depende de las condiciones de iluminación y requieren capacidad computacional para poder procesar las imágenes en tiempo real.

1.2.3 GNSS

Además de los sensores mencionados anteriormente, también es muy común el uso de sistemas GNSS (Sistema Global de Navegación por Satélite), entre los cuales el más conocido es el GPS. GNSS es el término genérico para referirse a todos los sistemas que usan satélites para determinar la posición, velocidad y tiempo en cualquier lugar del mundo.

La Figura 5 representa gráficamente el funcionamiento de estos sistemas que se basan en la trilateración. Conociendo la distancia entre el receptor y al menos cuatro satélites, es posible calcular la posición exacta del objeto. Para calcular correctamente las distancias, los satélites mientras están en órbita van emitiendo

su posición y hora exacta. El robot, equipado con un receptor GNSS, recibe estas señales y, midiendo el tiempo de llegada, calcula la distancia a cada satélite. Cuantas más señales de satélites se reciban, más precisos serán los resultados.

Debido a su gran exactitud, el GNSS es muy usado en la robótica móvil. No obstante, presenta inconvenientes, como su alto coste, la vulnerabilidad a interferencias y la pérdida de señal en entornos como túneles o zonas con grandes obstáculos.

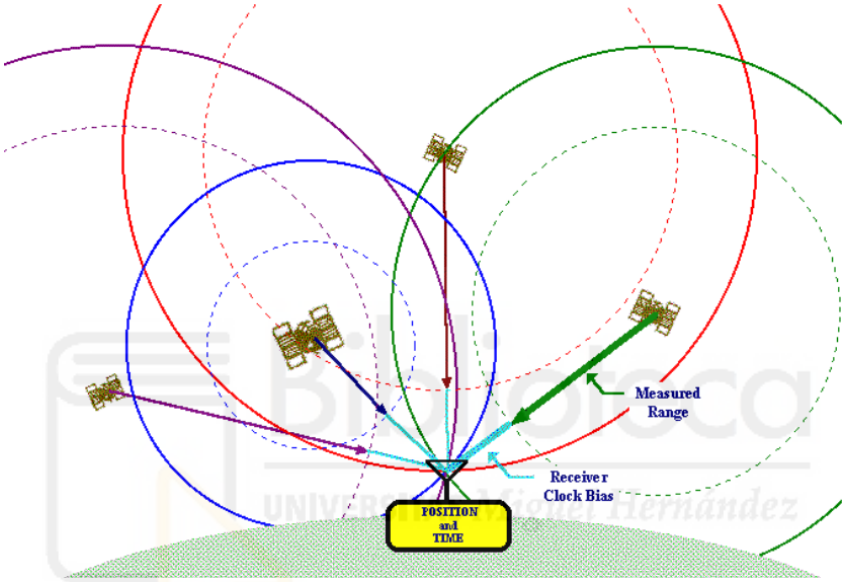

Figura 5. Representación del proceso trilateración con 4 satélites.

1.2.4 LIDAR

Tras analizar distintos métodos para obtener información del entorno, el siguiente sensor relevante es el LiDAR. Este dispositivo utiliza luz láser para generar una nube de puntos 3D con la que es posible mapear el espacio circundante. En la Figura 6 se muestra un ejemplo de un muestreo LiDAR completo, donde se representan todos los puntos detectados por el sensor en una única adquisición.

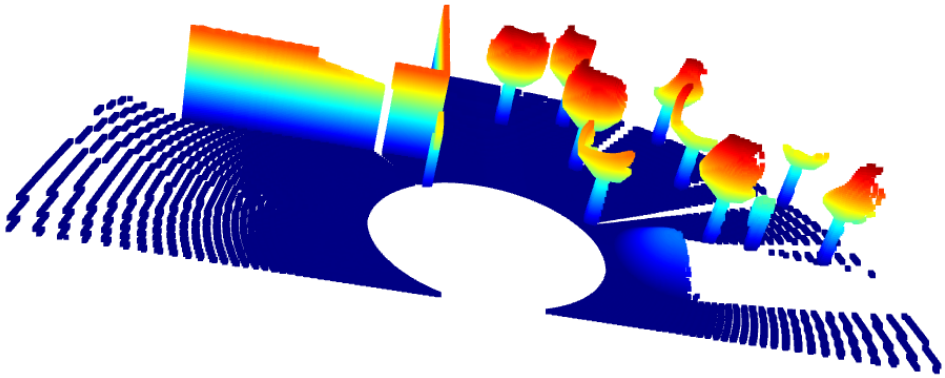

Figura 6. Nube de puntos obtenida a partir de un muestreo completo del sensor LiDAR.

Para crear la nube de puntos, el LiDAR calcula la distancia a los objetos midiendo el tiempo de vuelo del pulso láser, de forma similar a los sensores ultrasonidos, pero con la ventaja de que la velocidad de la permite obtener medidas mucho más precisas. Además de la distancia, el LiDAR registra la dirección del pulso emitido y recibido, lo que permite conocer los ángulos horizontal (azimut) y vertical (elevación). Con estos datos, la posición tridimensional de cada punto puede obtenerse conociendo la distancia (r) y los ángulos (θ, φ) mediante:

$$x = r \cdot \cos(\theta) \cdot \cos(\varphi)$$
$$y = r \cdot \sin(\theta) \cdot \cos(\varphi)$$
$$z = r \cdot \sin(\varphi)$$

El sistema LiDAR integra los siguientes componentes principales:

- Emisor láser, encargado de generar los pulsos.
- Receptor fotodetector, que capta el retorno del haz.
- Sistema de barrido, que dirige el láser en distintas direcciones, mediante: espejos giratorios; espejos MEMS, que al aplicarle una corriente gira en uno o dos ejes desviando así el haz láser; o sistemas flash que iluminan todo el campo de visión).
- Reloj de alta precisión, responsable de medir con exactitud el tiempo de vuelo.
- Procesador interno, que calcula la nube de puntos a partir de las distancias y los ángulos medidos.

Su uso está ampliamente extendido en robótica móvil, desde vehículos autónomos, donde se utiliza para detectar objetos y planificar trayectorias, hasta aplicaciones de topografía y cartografía, donde permite generar modelos 3D del terreno o de entornos urbanos [7] como se puede apreciar en la Figura 7. El LiDAR también resulta especialmente útil para resolver problemas de SLAM, en los que no se conoce ni la posición en la que se encuentra el robot ni el mapa del entorno. Sin embargo, como cualquier sensor, tiene sus limitaciones: condiciones atmosféricas adversas como la lluvia o el polvo pueden reducir su rango y aumentar del ruido, y las superficies especulares pueden reflejar el haz fuera del receptor, impidiendo su detección.

Tras evaluar los distintos tipos de sensores, se decidió utilizar el LiDAR por ser el que mejor se adaptaba a las necesidades.

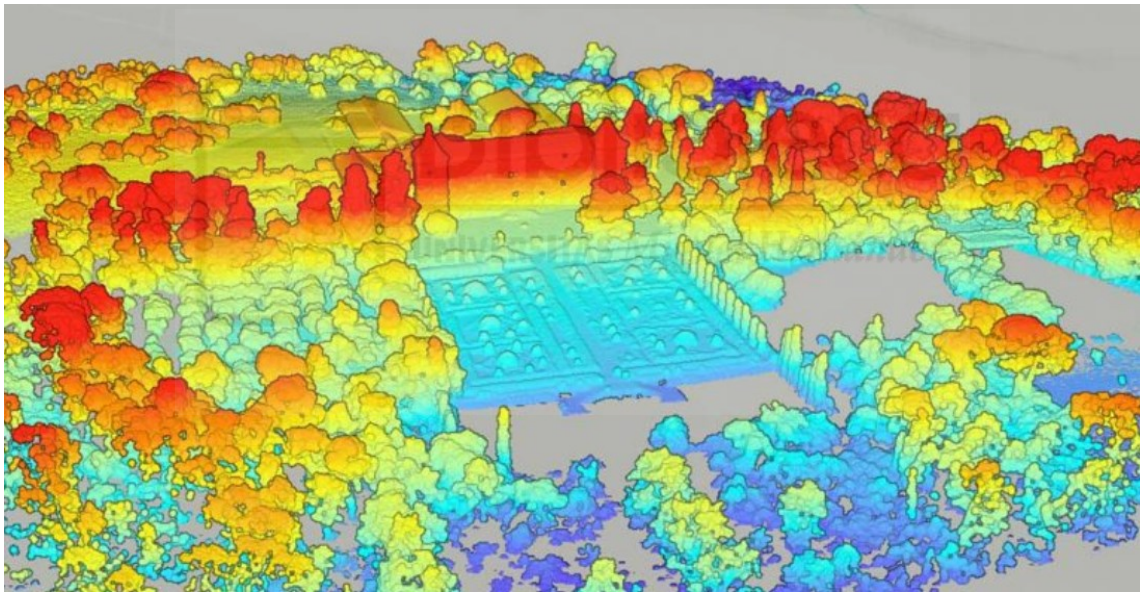

Figura 7. Ejemplo de nube de puntos 3D generada con LiDAR.

1.2.5 ODOMETRÍA

Además de conocer el entorno, es necesario obtener información sobre el movimiento del robot. Para ello se utiliza la odometría, una técnica en la que se estima la pose (posición y orientación) del robot o vehículo a partir de las mediciones de su propio desplazamiento.

En la Figura 8 se ilustra el principio de funcionamiento: se mide cuánto ha girado el motor o la rueda durante un tiempo y, a partir de ese ángulo, se calcula la distancia lineal recorrida. Con estos valores se actualiza la pose del robot y el proceso se repite de forma continua. Dentro de esta técnica los sensores más comunes que se usan son los encoders rotatorios que tienen ranuras o patrones para contar pulsos, los tacómetros para medir la velocidad angular o los sensores de ángulo para medir directamente cuánto gira.

No obstante, las estimaciones obtenidas mediante odometría no son totalmente exactas. Factores como el deslizamiento de las ruedas, el ruido de los sensores o las irregularidades del terreno provocan que el error se acumule con el tiempo, reduciendo la fiabilidad de la odometría si se utiliza de manera aislada.

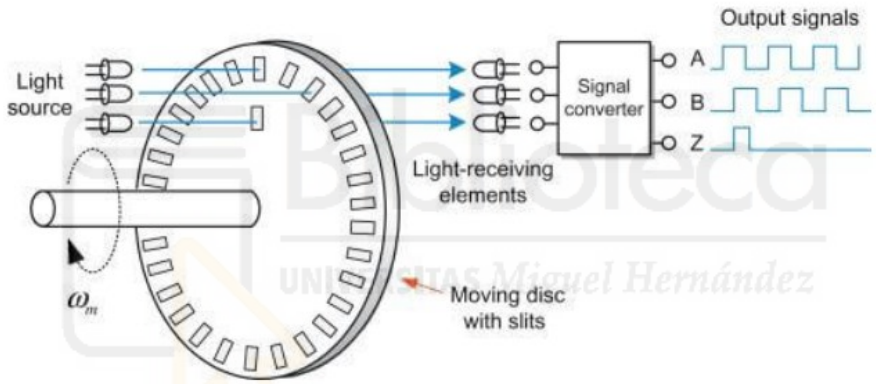

Figura 8. Encoder rotatorio y su lectura de pulsos.

1.3 OBJETIVOS

En el presente apartado se expondrán los objetivos del trabajo, distinguiendo entre objetivos técnicos, orientados a los aspectos prácticos y de desarrollo, y objetivos de aprendizaje, enfocados en las competencias y conocimientos que se buscan desarrollar.

1.3.1 OBJETIVOS TÉCNICOS

El objetivo principal del presente estudio es, a través de simulación, evaluar la capacidad de un robot móvil para desplazarse por un circuito predefinido en el cual hay obstáculos que deberá esquivar. Para cumplir este objetivo, el robot

debe ser capaz de desarrollar tres capacidades fundamentales: seguimiento de trayectorias, detección de obstáculos en un entorno local y replanificación de la trayectoria para esquivarlos de forma eficiente.

En una primera etapa, se evalúa la capacidad del robot de desplazarse a lo largo de la ruta definida en condiciones ideales, sin obstáculos. En una segunda etapa, se introducen obstáculos en distintos puntos del circuito, para analizar si se detectan correctamente y se identifican en tiempo real entre obstáculos o no para conocer que puntos son transitables. Tras confirmarse la correcta identificación de las zonas transitables, se estudia la efectividad del algoritmo de replanificación de forma offline, utilizando diferentes nubes de puntos que contienen los obstáculos detectados anteriormente.

Posteriormente, se evalúa el desempeño de todo el sistema conjunto, verificando que el robot pueda modificar la trayectoria cuando sea necesario y seguir el recorrido sin colisiones. Finalmente, se realizan pruebas en distintos escenarios para comparar si el robot consigue completar el circuito con diferente cantidad y tipo de obstáculos. Este enfoque permite validar la viabilidad de la navegación autónoma en un entorno conocido simulado.

1.3.2 OBJETIVOS DE APRENDIZAJE

El presente estudio tiene como finalidad no solo evaluar la capacidad del robot móvil en simulación, sino también consolidar y ampliar los conocimientos en diversas áreas relacionadas con la robótica y la programación. Entre los objetivos de aprendizaje destacan:

- Programación en Python: desarrollo de scripts para el control del robot, con énfasis en la eficiencia del código y su claridad.
- Simulación con CoppeliaSim: diseño y configuración de escenarios, así como programación de comportamientos autónomos dentro del entorno virtual para permitir la experimentación segura y reproducible.
- Planificación de trayectorias: comprensión y aplicación del algoritmo RRT (Rapidly-Exploring Random Tree), incluyendo sus fundamentos, ventajas, limitaciones y aplicaciones a la navegación de robots móviles.

- Robótica y navegación de vehículos autónomos: adquisición de competencias teóricas y prácticas en detección de obstáculos, replanificación de trayectorias y seguimiento de rutas, fundamentales para el desarrollo de sistemas autónomos complejos en entornos simulados.

2. MARCO TEÓRICO

En este capítulo se ofrece una visión general de las técnicas y métodos empleados en el ámbito de la robótica móvil para abordar problemas de transitabilidad y planificación de trayectorias. En el apartado 2.1 se revisan algunos trabajos relacionados en el área de conocimiento de interés que sirven como base y referencia. Seguidamente, en el apartado 2.2 se presenta una técnica de detección de obstáculos mediante sensores LiDAR. Finalmente, en el apartado 2.3 se exponen los principales métodos de planificación de trayectorias en mapas, los cuales son esenciales para la navegación en robótica móvil.

2.1 TRABAJOS RELACIONADOS

La planificación de trayectorias y la evasión de obstáculos constituyen dos de los problemas principales en la navegación autónoma de robots móviles. A lo largo de las últimas décadas se han propuesto numerosos enfoques que abarcan desde el uso de métodos clásicos basados en grafos, como Dijkstra o A*, hasta algoritmos de muestreo como RRT, que se analizará en el apartado 2.3.

Paralelamente, la integración de sensores como LiDAR o cámaras de profundidad ha permitido una representación más precisa del entorno, facilitando la detección de obstáculos y la generación de trayectorias seguras. Estudios recientes muestran que, si bien los métodos actuales ofrecen soluciones efectivas en entornos estáticos o parcialmente conocidos, todavía presentan limitaciones cuando se enfrentan a escenarios dinámicos y con restricciones de tiempo real. A continuación, se revisan algunos trabajos relevantes para situar la contribución de este TFG en contexto.

El primer estudio, “Research on unmanned vehicle obstacle avoidance technology based on LIDAR and depth camera fusión” [8], analiza el uso combinado de LIDAR y cámaras de profundidad para mejorar la detección de obstáculos en vehículos no tripulados. La propuesta se basa en fusionar los datos de ambos sensores para aprovechar la precisión del LiDAR y la información visual de la cámara. Una de las principales ventajas es que se logra

una mayor fiabilidad en entornos con iluminación variable o condiciones climáticas cambiantes, donde un solo sensor podría fallar. Además, permite generar mapas tridimensionales más detallados, útiles para la navegación autónoma. Entre las desventajas, se destacan los altos costos de implementación y la gran capacidad de cómputo que se necesita para procesar la información en tiempo real. En conclusión, la tecnología de fusión de LiDAR y cámara es prometedora para aumentar la seguridad y precisión en la conducción autónoma, pero aún se enfrenta a retos técnicos y económicos para su aplicación en muchos ámbitos.

El segundo artículo, “Gradient Field-Based Dynamic Window Approach for Collision Avoidance in Complex Environments” [9], propone una mejora del método “Dynamic Window Approach” (DWA) mediante el uso de campos de gradiente para guiar el movimiento de un robot en entornos complejos. Mientras que el objetivo atrae y los obstáculos repelen, produciendo vectores/pendientes locales. El DWA es un planificador local usado en robótica móvil que genera velocidades seguras y factibles teniendo en cuenta la dinámica del robot, su objetivo es elegir la velocidad lineal y angular que lleve al robot hacia la meta sin colisiones cumpliendo los límites físicos. Al combinar la planificación local con la capacidad de respuesta dinámica del robot permite ajustar finamente las trayectorias y evitar que quede atrapado en mínimos locales. La ventaja principal es que el sistema puede adaptarse rápidamente a cambios en el entorno, manteniendo rutas seguras y eficientes. Como desventajas, se señala la necesidad de ajustar parámetros cuidadosamente, lo que puede dificultar su implementación práctica. Además, puede demandar mayor procesamiento en entornos densamente poblados. En conclusión, este enfoque mejora la capacidad de evasión en situaciones dinámicas, pero requiere optimización para garantizar su aplicabilidad en escenarios reales complejos.

Por último, el trabajo “Off the Beaten Track: Laterally Weighted Motion Planning for Local Obstacle Avoidance” [10] introduce un método de planificación de movimiento que pondera lateralmente las trayectorias, con el objetivo de evitar obstáculos de forma más natural y fluida. A diferencia de los enfoques tradicionales, esta técnica prioriza desplazamientos laterales suaves para seguir el camino o girar alrededor de obstáculos, incluso si eso significa desviarse del

camino al objetivo. Entre sus ventajas, se destaca que genera rutas más suaves, mejora la comodidad del movimiento y puede ser especialmente útil en vehículos o robots que comparten espacio con humanos. Su principal desventaja es la mayor complejidad algorítmica y la posible ineficiencia en espacios muy reducidos donde el movimiento lateral es limitado. En conclusión, este enfoque aporta una perspectiva innovadora para la evasión local, con gran potencial en espacios compartidos, aunque requiere ajustes para situaciones muy congestionadas.

En conjunto, estos tres trabajos analizados muestran estrategias diversas en el campo de la navegación autónoma. Mientras que la fusión sensorial busca mejorar la percepción del entorno para alimentar algoritmos más robustos, los métodos basados en ventanas dinámicas y los que incorporan funciones de coste lateral exploran nuevas formas de optimizar la planificación local y la interacción con obstáculos. Sin embargo, cada propuesta enfrenta limitaciones propias, ya sea en términos de coste computacional, dificultad de parametrización como el estudio que usa DWA o restricciones de aplicabilidad en determinados escenarios.

En este contexto, el presente trabajo propone una aproximación complementaria que combina el uso de la diferencia de normales aplicada a datos de LiDAR para la detección de obstáculos y con el algoritmo RRT para la planificación de trayectorias. Esta integración permite contar con una percepción robusta y precisa del entorno al mismo tiempo que se generan rutas viables y eficientes en mapas complejos. De esta forma, la propuesta busca aportar un equilibrio entre la fiabilidad en la detección y flexibilidad en la planificación. Además, esta propuesta puede implicar una menor carga computacional en comparación con los trabajos mencionados, lo que sugiere un menor coste de procesamiento para escenarios similares.

2.2 DETECCIÓN DE OBSTÁCULOS CON LIDAR

En el ámbito de la detección de obstáculos usando LiDAR se han desarrollado diversos métodos para analizar y segmentar nubes de puntos tridimensionales.

Entre estos métodos, para el presente trabajo se ha optado por emplear la diferencia de normales (Difference of Normals, DoN), debido a su capacidad para detectar discontinuidades geométricas de forma eficiente y con un coste computacionales moderado. En este apartado se describe el funcionamiento de este método y se presentan ejemplos de investigaciones en las que se ha aplicado.

La técnica de diferencia de normales es una herramienta utilizada en el análisis de nubes de puntos para la detección de bordes, aristas y cambios bruscos de geometría. Su principio se basa en calcular la normal en cada punto en dos escalas distintas: una con un radio pequeño, que captura los detalles locales, y otra con un radio mayor, que representa la forma global de la superficie. La diferencia entre ambas normales proporciona una medida del cambio de orientación local de la superficie. Si la variación es significativa, se interpreta como la presencia de un borde, arista o zona con alta curvatura; en cambio, si la diferencia es pequeña, el punto se considera perteneciente a una región lisa. Con el fin de ilustrar de forma intuitiva este concepto, en la Figura 9 se muestran, para cada punto de la nube de puntos 3D, las normales estimadas utilizando los dos radios de vecindad.

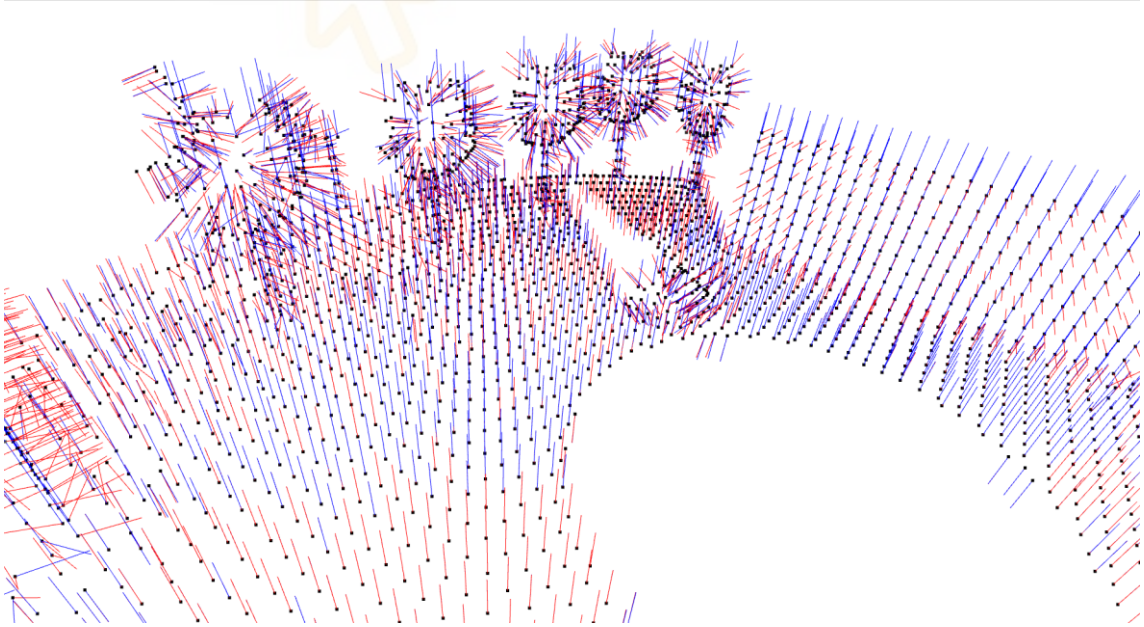

Figura 9. Normales calculadas con ambos radios en cada punto de la nube.

Como se puede observar en la figura anterior, en superficies planas ambas normales presentan direcciones muy similares, ya que la geometría local y global coincide. Por el contrario, en zonas de borde la normal calculada con un radio pequeño se adapta a la geometría local, mientras que la normal calculada con un radio mayor refleja una superficie suavizada, provocando una discrepancia significativa entre ambas. Esta diferencia es la que permite identificar bordes y obstáculos de forma robusta, constituyendo la base del método DoN empleado en este trabajo. En la Figura 10 se muestra un ejemplo del resultado de la detección de bordes mediante el operador DoN aplicado a una nube de puntos LiDAR.

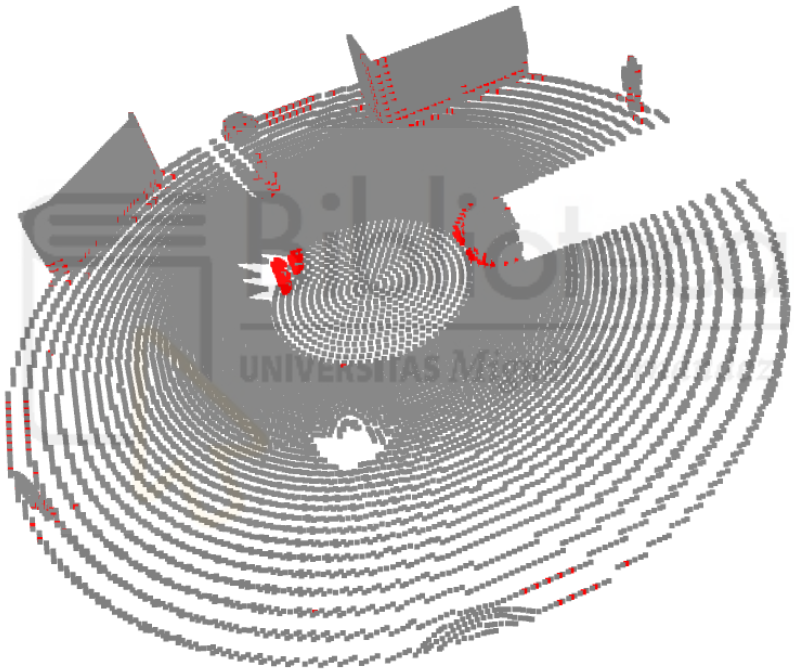

Figura 10. Detección de bordes (en rojo) en una nube de puntos LiDAR mediante DoN.

El umbral para clasificar estas diferencias puede variar según la resolución de la nube de puntos y la aplicación concreta. Este procedimiento no solo permite detectar discontinuidades de manera eficiente, sino que también facilita la segmentación de objetos en escenarios complejos. Además, es especialmente valioso en aplicaciones donde la nube de puntos es muy densa, ya que ayuda a reducir la información resaltando únicamente las zonas más relevantes. En definitiva, la diferencia de normales constituye un método robusto y flexible para

mejorar la interpretación de datos tridimensionales, aportando eficiencia en áreas como la robótica o la visión por computador.

El primer trabajo en el que se introduce formalmente la técnica de diferencia de normales es el artículo de Ioannou et al. (2012), titulado “Difference of Normals as a Multi-Scale Operator in Unorganized Point Clouds” [11], publicado en 2012. En este estudio se plantea por primera vez el uso de DoN como un operador multiescalar capaz de resaltar bordes y discontinuidades en nubes de puntos no organizadas, como las obtenidas mediante sensores LiDAR. En el trabajo se demuestra que al calcular las normales en diferentes escalas espaciales y compararlas, es posible detectar objetos de distintos tamaños en entornos urbanos, como peatones, vehículos o farolas. Este trabajo marcó un punto de partida, ya que estableció las bases de la técnica, mostrando además que se podía integrar en procesos de segmentación y clasificación de escenas tridimensionales complejas.

Posteriormente, la técnica ha sido aplicada en el campo de la robótica móvil. Un ejemplo es el trabajo de Edison Velasco, titulado “Estimación de la pose y control de Robots Móviles en Entornos no Estructurados basados en LiDAR” [12]. En esta investigación, se aborda el problema de la navegación autónoma en escenarios reales, donde las condiciones del terreno y la presencia de obstáculos imprevisibles plantean grandes desafíos. El autor utiliza un sensor LiDAR para captar el entorno y aplica el operador DoN con el fin de detectar bordes y discontinuidades en la nube de puntos generada, diferenciando con precisión entre superficies transitables y obstáculos.

Los resultados que se obtienen muestran que la técnica de la diferencia de normales ofrece un buen equilibrio entre precisión y eficiencia computacional, aspecto crítico para la operación en tiempo real de un robot móvil. Se evidenció que, mediante la correcta elección de los parámetros, el sistema identifica irregularidades en el terreno como bordillos y objetos de diferentes tamaños con un bajo tiempo de procesamiento. Esto contribuye a una mejor estimación de la pose, al disponer de un mapa más detallado y segmentado del entorno. Además, en el trabajo se demuestra que el uso de DoN en conjunto con estrategias de control adaptativo permite al robot mantener trayectorias más estables y

seguras, incluso en superficies con pendientes. En definitiva, el estudio confirma la aplicabilidad de la diferencia de normales en contextos de navegación autónoma, validando su utilidad como herramienta para aumentar la autonomía y robustez de sistemas de robótica móvil en entornos no estructurados.

Asimismo, se han explorado aplicaciones en contextos muy distintos, como la navegación submarina en el trabajo “Bathymetry-based SLAM with Difference of Normals Point-Cloud Subsampling and Probabilistic ICP Registration” [13]. En esta investigación, los autores emplean datos batimétricos, mediciones de la profundidad del agua en el fondo de océanos o lagos, para la localización y el mapeo simultáneo (SLAM) en entornos submarinos, donde la calidad de las nubes de puntos suele ser muy irregular debido a las condiciones del medio. La técnica de DoN se utiliza como mecanismo de submuestreo adaptativo, permitiendo reducir la densidad de puntos en zonas planas y conservar detalles relevantes en áreas con mayor variación geométrica. Esto mejora la eficiencia del proceso de registro al disminuir la carga computacional sin perder precisión. Combinada con un algoritmo ICP probabilístico, empleado para alinear dos nubes de puntos mediante la búsqueda de una matriz de transformación que minimiza la distancia entre ellas. La metodología logra resultados robustos en la construcción de mapas submarinos, demostrando que el DoN puede extenderse más allá de la robótica terrestre, siendo útil en aplicaciones donde la detección de bordes y discontinuidades es esencial para la navegación.

Como se observa en los distintos estudios mencionados, la técnica de diferencia de normales puede aplicarse en contextos muy variados, pero en todos los casos su rendimiento depende de los parámetros elegidos. A continuación, se muestra la fórmula usada para cada punto p, que calcula la DoN como un vector, donde n̂ representa la normal estimada del punto p para los radios de vecindad r_1 y r_2:

$$\Delta_{\hat{\mathbf{n}}}(\mathbf{p}, r_1, r_2) = \frac{\hat{\mathbf{n}}(\mathbf{p}, r_1) - \hat{\mathbf{n}}(\mathbf{p}, r_2)}{2}$$

En este caso, al ser un vector indica orientación de las superficies y bordes cuando existe una mayor diferencia en un eje que en otro. En cambio, si no importa la orientación del borde, sino únicamente si hay borde o no, se puede

normalizar el vector que indica DoN para obtener únicamente la magnitud de la variación en lugar de la dirección. Para el cálculo de DoN, como se ha mencionado, se deben calcular las normales para dos valores distintos: los parámetros r_1 y r_2. El primer radio corresponde al radio pequeño y controla el nivel de detalle local: valores reducidos permiten detectar bordes finos, aunque con mayor sensibilidad al ruido, mientras que valores grandes suavizan la información y pueden ocultar detalles relevantes. Por su parte, el segundo radio actúa como radio grande y describe la forma global de la superficie; radios pequeños limitan la detección de discontinuidades amplias, mientras que radios grandes destacan únicamente los bordes más marcados. La relación entre ambos también es crítica: si la diferencia entre r_1 y r_2 es demasiado pequeña, las normales resultan similares y apenas se resaltan bordes; si la diferencia es excesiva, aumenta la detección de falsos positivos. Finalmente, el umbral determina qué puntos se consideran bordes. Como se observa en la Figura 11, la magnitud de los radios afecta a detectar unos bordes u otros. Valores bajos incrementan la sensibilidad, pero pueden generar ruido, mientras que valores altos reducen la detección a cambios significativos. Además, el umbral está relacionado con la diferencia de tamaños de los radios: cuanto mayor sea la diferencia entre r_1 y r_2, más elevado debe ser el umbral para evitar un exceso de falsos positivos.

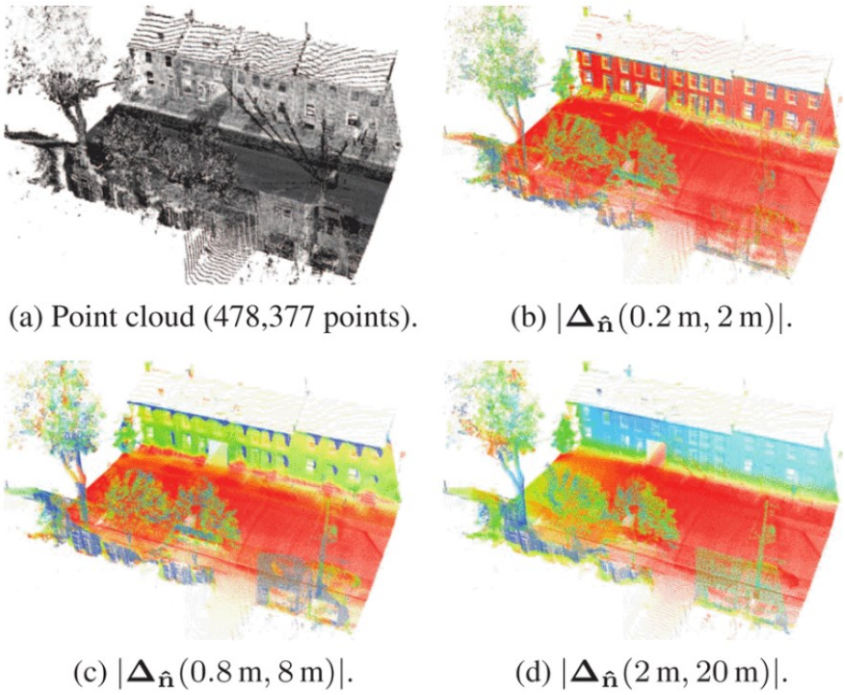

Figura 11. Representación visual del cálculo de DoN con diferentes valores de radios.

2.3 PLANIFICACIÓN DE TRAYECTORIAS EN MAPAS

La planificación de trayectorias en mapas constituye un área fundamental en robótica y en sistemas de navegación autónoma, ya que permite determinar rutas óptimas o factibles evitando obstáculos. Existen diversos métodos que abordan este problema desde diferentes enfoques, entre los más representativos se encuentran los algoritmos Dijkstra, A* y RRT. A continuación, se presentará una descripción de cada uno de estos métodos.

2.3.1 DIJKSTRA

El algoritmo de Dijkstra [14] es un método clásico de búsqueda de caminos mínimos en grafos ponderados. Su objetivo es encontrar la ruta más corta desde un nodo de inicio hasta todos los demás nodos del mapa. Funciona expandiendo de manera iterativa el nodo con el costo acumulado más bajo, actualizando las distancias mínimas conocidas hacia los vecinos, y marcando aquellos ya visitados. Este proceso continúa hasta que se alcanza el nodo destino o todos los nodos han sido explorados, garantizando la solución de la ruta más corta.

Sin embargo, su principal limitación es el alto costo computacional en grafos grandes, ya que explora exhaustivamente muchas rutas antes de llegar a la meta. Aun así, es ampliamente utilizado en aplicaciones como la planificación de rutas en redes de transporte, sistemas de navegación GPS y análisis de redes de comunicación, donde la exactitud del resultado es prioritaria frente al tiempo de cómputo.

2.3.2 A*

Por otro lado, el algoritmo A* [15] es una evolución del método de Dijkstra que mejora su eficiencia añadiendo una función heurística que estima la distancia restante hasta la meta. De este modo, en cada paso selecciona el nodo que tiene el menor valor de la suma entre el costo recorrido (*g*) y la heurística estimada (*h*), lo que se conoce como $f(n) = g(n) + h(n)$. Esto permite priorizar la expansión de los nodos que parecen más prometedores, reduciendo el número de estados explorados y acelerando la búsqueda.

Una de sus ventajas más importantes es que si la heurística es admisible (no sobreestima el costo real), A* garantiza encontrar la ruta óptima, pero con mayor eficiencia que Dijkstra. Por ello, es ampliamente utilizado en aplicaciones de navegación de robots móviles, cálculo de rutas en videojuegos y sistemas que requieren tomar decisiones en tiempo real. A* logra un equilibrio muy útil entre precisión y eficiencia, lo que lo convierte en uno de los métodos más populares en la planificación de trayectorias.

2.3.3 RRT

El último algoritmo de planificación de trayectorias que se va a explicar es el Rapidly-exploring Random Tree (RRT) [16]. Este algoritmo es un planificador basado en muestreo diseñado para construir rápidamente un árbol de configuraciones alcanzables en el espacio libre. A continuación, se muestra en la Figura 12 su pseudocódigo del método para un espacio de configuración general *C*, partiendo desde una configuración inicial q_{init} que pertenezca al espacio *C* y con una cantidad máxima de *K* iteraciones para poder encontrar el camino:

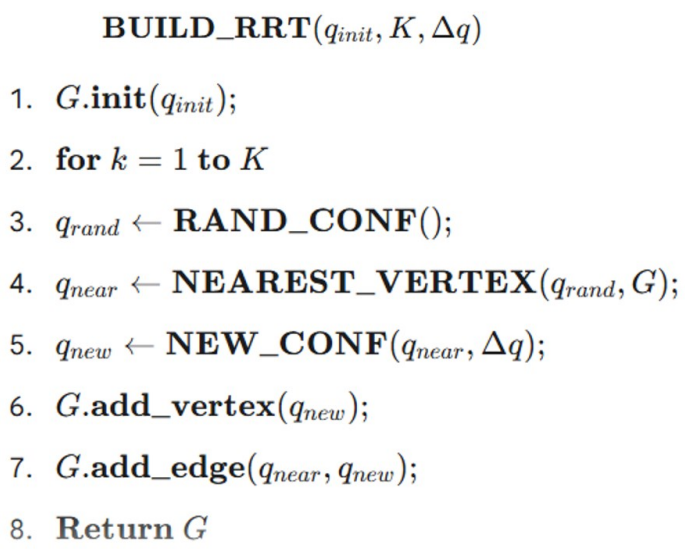

Figura 12. Pseudocódigo RRT.

Como se puede ver en la figura anterior, en primer lugar, se guarda la configuración inicial, q_{init}, en el árbol *G* que representa todos los vértices y líneas que pertenecen al RRT creado. Luego, en cada iteración se busca una posición aleatoria q_{rand} que no necesariamente ha de pertenecer explícitamente al conjunto de configuraciones admisibles del espacio *C*. Posteriormente, se identifica el nodo q_{near} del árbol *G* cuyo valor es el más próximo a q_{rand} conforme a una métrica previamente definida. Tras esto, se ejecuta una operación de propagación que intenta avanzar desde el nodo q_{near} hasta q_{rand} con una distancia de Δq, consiguiendo así crear un nuevo nodo, q_{new}. Antes de incorporar este nuevo nodo al árbol se verifica la validez de la trayectoria entre q_{near} y q_{new} comprobando si hay colisiones en este segmento, en caso de ser válido se añade el nodo y el segmento al árbol. El algoritmo termina cuando un nodo llega hasta el punto deseado y así se ha completado el camino, o en su defecto, tras realizar *K* iteraciones.

RRT es un algoritmo probabilísticamente completo: si existe la solución y se deja al algoritmo muestrear indefinidamente (tantas iteraciones como sean necesarias), la probabilidad de encontrar una solución tiende a 1. No obstante, la primera trayectoria hallada por RRT suele no ser óptima ni suave; por ello se aplican técnicas para mejorar la calidad como la optimización local. Para obtener la solución óptima existe la variante RRT*, que incorpora técnicas de

reconfiguración del árbol y garantiza optimalidad asintótica. Además de RRT*, existen varias variantes como el RRT-Connect, que usa dos árboles (uno desde el inicio y otro desde la meta) que van creciendo y tratan de conectarse, acelerando de esta manera la búsqueda. Sin embargo, estas versiones son más complejas que el RRT básico, del cual se está hablando.

La eficiencia y robustez del RRT dependen críticamente de variables que se eligen al implementarlo: el tamaño del paso, Δq, o la métrica de distancia empleada para determinar el nodo más cercano pueden ser determinantes. RRT puede tener dificultades en pasajes estrechos en el espacio C, donde el muestreo uniforme raramente genera configuraciones útiles y, en caso de que el camino tenga que pasar por ahí para llegar al objetivo, podría necesitar una gran cantidad de iteraciones. En aplicaciones reales, RRT suele usarse como planificador global inicial debido a su simplicidad y su capacidad para explorar grandes espacios; después se combina con técnicas para refinamiento como la optimización. En este estudio, se ha optado por utilizar el RRT básico, dado que no se requiere obtener una solución óptima, y su rapidez y simplicidad resultan adecuadas para los objetivos planteados. Finalmente, se presenta la Figura 13 para ilustrar cómo crece el árbol en un mapa con obstáculos para encontrar el camino entre dos puntos:

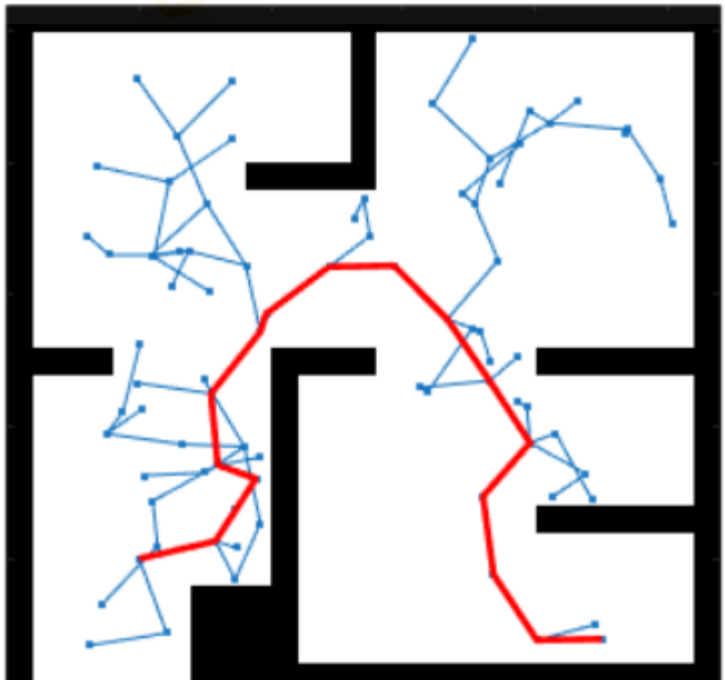

Figura 13. Esquema conceptual del RRT en un mapa con obstáculos.

3. HERRAMIENTAS UTILIZADAS

En este capítulo se presenta un panorama general de las herramientas empleadas en el desarrollo del proyecto, destacando tanto los entornos de simulación utilizados como los recursos específicos que permiten la implementación y validación de las funciones propuestas.

3.1 HERRAMIENTAS DE SIMULACIÓN

La simulación constituye una fase fundamental en el desarrollo, validación y verificación de sistemas robóticos ya que permite evaluar algoritmos y comportamientos sin incurrir en los costes o riesgos asociados al uso directo en plataformas físicas. En el ámbito de la robótica existen múltiples entornos de simulación, como MuJoCo [17], Webots [18], MORSE [19] o Unity Robotics [20]. Entre las herramientas más extendidas y relevantes se encuentran ROS/Gazebo y CoppeliaSim.

ROS/Gazebo [21] constituye una de las combinaciones más empleadas en robótica, especialmente en entornos académicos e industriales, debido a su integración directa con el ecosistema Robot Operating System (ROS). Gazebo ofrece un motor de simulación física robusto, modelos detallados de robots y sensores, y una infraestructura que permite reproducir escenarios complejos con un alto grado de realismo como se observa en la Figura 14. Su uso típico incluye:

- Pruebas de algoritmos de navegación y control de robots móviles y manipuladores.
- Simulación de sensores avanzados, como LiDAR o cámaras RGB-D, para validar percepción y fusión de datos.
- Transferencia de controladores desde la simulación al hardware real, gracias a su estrecha integración con ROS.

Entre sus principales ventajas se encuentran su escalabilidad, la amplia disponibilidad de recursos comunitarios y la precisión física. No obstante, también presenta limitaciones como una curva de aprendizaje relativamente

elevada y un alto consumo computacional, factores que pueden reducir su conveniencia para entornos educativos básicos o prototipado rápido.

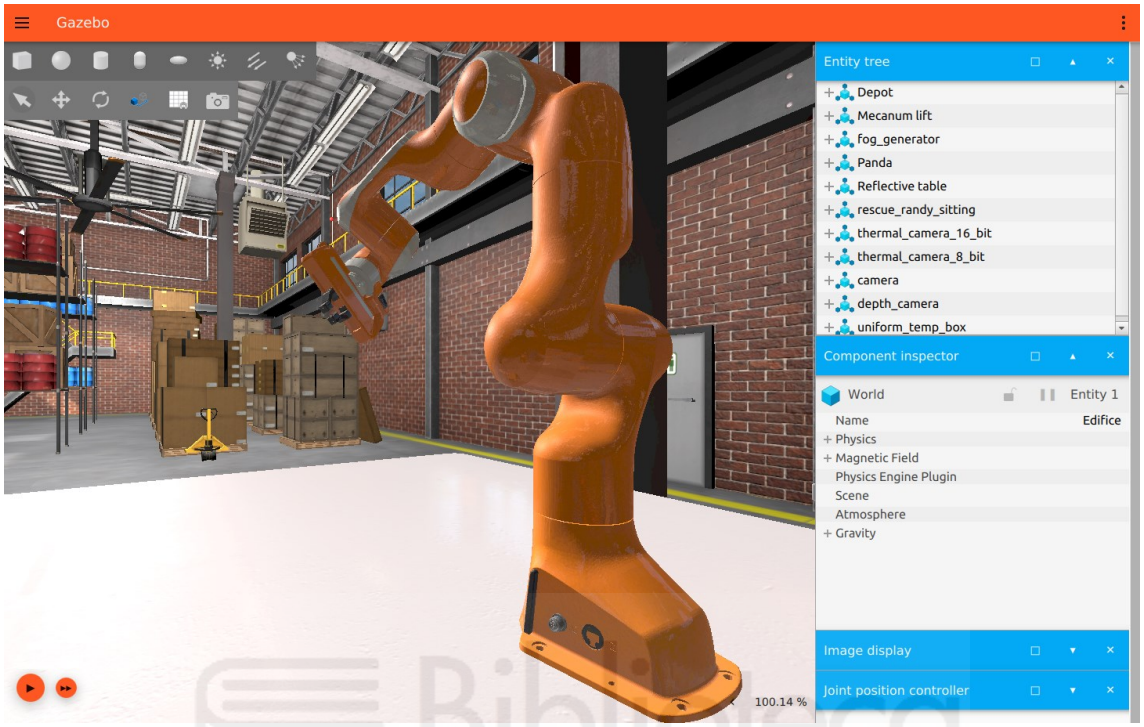

Figura 14. Ejemplo entorno de ROS/Gazebo.

CoppeliaSim [22] (anteriormente V-REP), por su parte, es otro simulador ampliamente utilizado en investigación y docencia. Se caracteriza por su enfoque versátil y su arquitectura flexible, ofreciendo herramientas intuitivas para configurar robots y entornos, así como mecanismos para ejecutar scripts y conectar algoritmos externos mediante diversas APIs compatibles con múltiples lenguajes y sistemas. Aunque su realismo físico puede ser menor que el de Gazebo y su rendimiento puede disminuir en escenas muy grandes, CoppeliaSim resulta muy útil para prototipado rápido, simulaciones educativas y experimentales. A continuación, se muestra la Figura 15 en la que se puede observar la interfaz de CoppeliaSim. En la parte izquierda se encuentra el panel de selección de modelos mediante el cual se agregan elementos a la simulación, mientras que a su derecha se visualiza la estructura jerárquica de la escena que refleja la arquitectura modular de la escena:

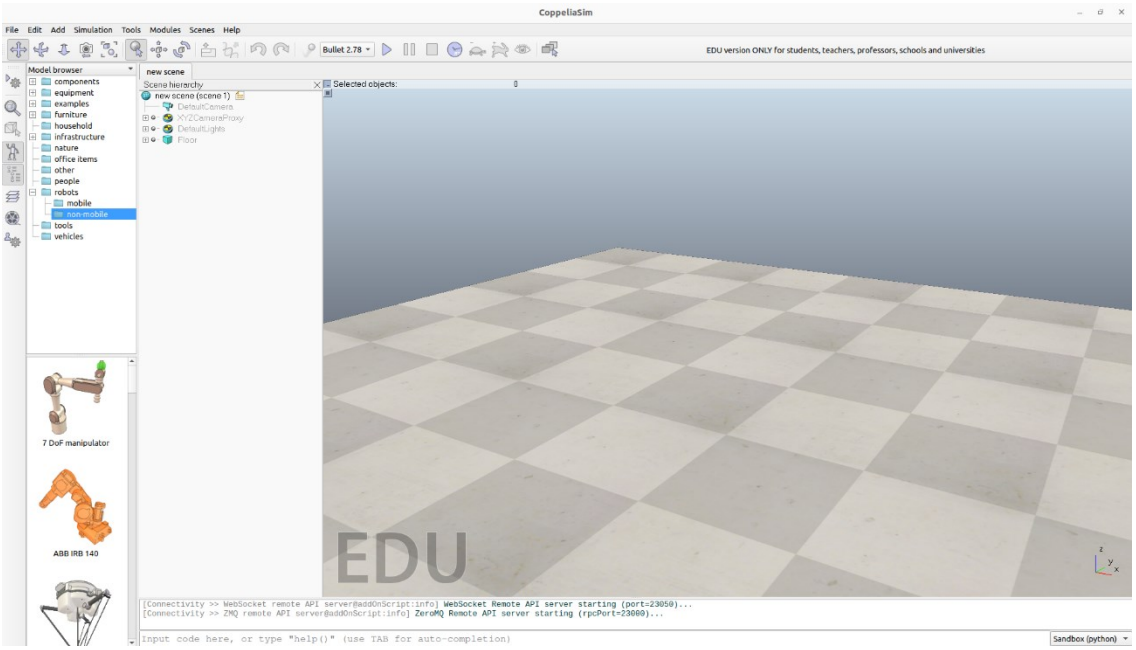

Figura 15. Interfaz de CoppeliaSim.

En resumen, mientras Gazebo se centra en el realismo físico y la integración profunda con ROS, CoppeliaSim ofrece flexibilidad de uso y prototipado rápido, lo que lo hace especialmente adecuado para este proyecto.

3.2 COPPELIASIM

En este apartado se profundizará más en las características CoppeliaSim y se explicarán las razones por las que se ha preferido usar este simulador frente a Gazebo.

CoppeliaSim es un entorno de simulación modular y altamente configurable, diseñado para permitir la creación, prueba y validación de robots, sensores y actuadores en escenarios virtuales de manera flexible. Su arquitectura soporta múltiples lenguajes de programación, motores de física configurables y la ejecución de scripts internos o externos, lo que facilita el desarrollo de algoritmos de control, planificación y percepción en tiempo real. Entre sus principales características destacan:

- Arquitectura modular y flexible: Integración sencilla de robots, sensores y objetos dinámicos y estáticos.
- Compatibilidad con múltiples lenguajes: Python, C/C++, Java, Lua y MATLAB, lo que permite integrar algoritmos externos de forma directa.
- Motores de física configurables: Bullet, ODE o Vortex, ajustando el equilibrio entre realismo y rendimiento según las necesidades del proyecto.
- Simulación de sensores y actuadores: Cámaras, LiDAR, GPS, IMU, motores, servos, brazos robóticos, entre otros. En la Figura 16 se puede observar un brazo robótico usando una cámara en Coppelia, en este caso la cámara está situada en el efector final del brazo robótico y es de color azul.
- Interacción mediante scripts y APIs externas: Permite ejecutar y controlar la simulación desde programas externos, favoreciendo pruebas de algoritmos complejos.
- Entorno gráfico avanzado e intuitivo: Facilita el modelado, posicionamiento y modificación de robots y objetos, así como la depuración y validación visual de comportamientos.
- Prototipado rápido y educativo: Ideal para proyectos experimentales y entornos de enseñanza, donde la rapidez y flexibilidad son prioritarias.

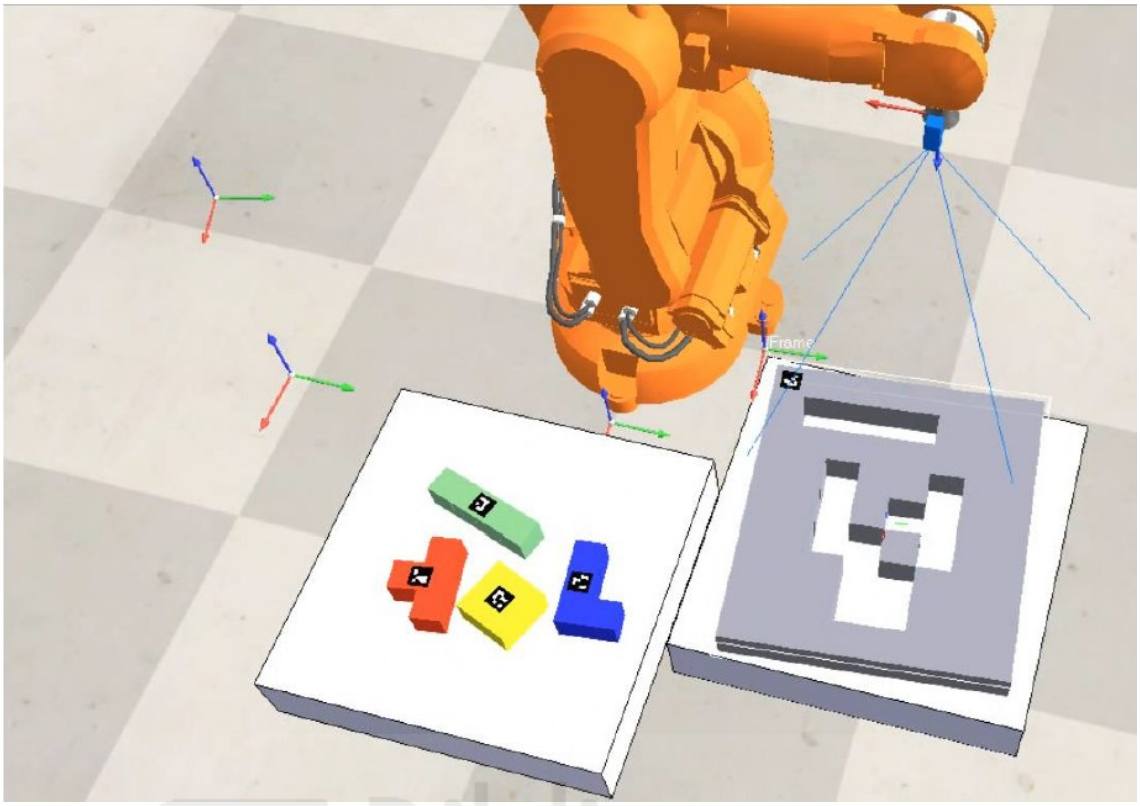

Figura 16. Ejemplo de simulación de un robot manipulador en CoppeliaSim.

Gracias a estas características, CoppeliaSim resulta particularmente adecuado para este proyecto. La principal razón para elegir este simulador para este estudio es que su entorno intuitivo y flexible permite desarrollar y probar algoritmos de manera rápida, sin depender de un ecosistema específico como ROS, el cual tiene una curva de aprendizaje muy elevada. Además, se han considerado otras razones relevantes para su elección:

- Gran flexibilidad de integración, gracias a la posibilidad de conectar algoritmos externos mediante distintos lenguajes de programación y APIs.
- Adecuación al nivel de realismo requerido: el proyecto no requiere de un realismo físico extremo, por lo que CoppeliaSim cumple los objetivos de manera eficiente.
- Validación de comportamientos y algoritmos de manera directa, optimizando la experimentación y corrección de errores.

A continuación, se muestra la Tabla I en la que se hace una comparación entre ROS/Gazebo y CoppeliaSim para los rasgos más significativos:

Característica	ROS/Gazebo	CoppeliaSim
Realismo físico	Alto, adecuado para simulaciones avanzadas.	Moderado, suficiente para prototipado y educación.
Integración con ROS	Nativa y profunda, con paquetes oficiales	Posible mediante APIs, no nativa.
Curva de aprendizaje	Elevada; requiere conocimientos de ROS.	Moderada; entorno más intuitivo.
Prototipado rápido	Menos adecuado por complejidad y configuración.	Muy adecuado; rápido y flexible.
Tamaño de escenas	Soporta entornos grandes y complejos.	Puede reducir su rendimiento en escenas muy grandes.

Tabla I. Comparación entre ROS/Gazebo y CoppeliaSim.

En conclusión, CoppeliaSim combina flexibilidad, facilidad de uso y compatibilidad con múltiples lenguajes, consolidándose como la opción más adecuada para este proyecto.

3.3 PYARTE

En este apartado se presenta pyARTE [23], una librería desarrollada en la UMH por A. Gil en Python destinada a la enseñanza, experimentación y aplicación de métodos de robótica mediante el lenguaje de programación Python. Su propósito principal es proporcionar una herramienta accesible y didáctica para la modelización, simulación y control de robots, principalmente para manipuladores, pero también para algunos móviles. De esta forma permite a estudiantes e investigadores trabajar con conceptos fundamentales de la robótica, como la cinemática, la dinámica y la planificación de movimientos, sin la necesidad de plataformas propietarias o entornos complejos. Por consiguiente, pyARTE se integra como un recurso versátil dentro del ecosistema Python, facilitando tanto el aprendizaje como el desarrollo de prototipos y experimentos. Entre sus principales características destacan:

- Modelado de robots mediante parámetros DH: La librería permite definir manipuladores utilizando parámetros de Denavit-Hartenberg estándar o modificados, facilitando la obtención de matrices homogéneas y la representación de su cadena cinemática.
- Cálculo de cinemática directa e inversa: Incluye funciones para determinar la posición y orientación del efector final a partir de los valores articulares, así como métodos numéricos o analíticos para resolver la cinemática inversa cuando la arquitectura del robot lo permite.
- Generación y análisis de trayectorias: pyARTE permite planificar trayectorias en el espacio articular o cartesiano, incluyendo interpolaciones lineales o polinomiales, lo que facilita la evaluación de movimientos controlados y su posterior transferencia a un simulador.
- Enfoque didáctico y accesible: Su sintaxis clara y su documentación orientada a ejemplos la convierten en una herramienta adecuada para estudiantes y proyectos académicos, permitiendo una curva de aprendizaje reducida respecto a estructuras más complejas.
- Integración con el ecosistema Python: Gracias a su compatibilidad con bibliotecas como NumPy o Matplotlib, pyARTE permite visualizar resultados, realizar análisis numéricos y complementar algoritmos de control con técnicas avanzadas de optimización o aprendizaje automático.
- Facilidad de integración con CoppeliaSim: pyARTE puede complementarse de forma directa con el entorno de simulación, permitiendo enviar consignas de movimiento y recibir estados del robot de manera sencilla.

Gracias a estas características, pyARTE es una herramienta adecuada para este proyecto, ya que permite implementar y verificar funciones de cinemática y planificación sin la necesidad de entornos más complejos como ROS.

Además, la comunicación entre CoppeliaSim y pyARTE, mediante la librería ZeroMQ (zmq), permite el intercambio eficiente de datos en tiempo real entre ambos entornos. La conexión y gestión de dicha comunicación se implementa en el archivo *simulation.py*, que actúa como puente entre el simulador y los algoritmos desarrollados en Python. De este modo, su integración directa con Python facilita la conexión con scripts externos utilizados en CoppeliaSim,

permitiendo combinar la precisión matemática de pyARTE con la visualización y validación práctica del simulador. Este enfoque híbrido garantiza un flujo de trabajo eficiente, flexible y apropiado para los objetivos planteados en este estudio.

4. SOFTWARE

El software desarrollado constituye la base lógica del sistema y está diseñado siguiendo una estructura modular. A lo largo de esta sección se describe cómo se organiza el código, el papel de cada uno de los scripts que lo componen y la forma en que estos interactúan entre sí. También se incluye una representación gráfica del funcionamiento general del programa mediante un diagrama de flujo, así como un análisis de las funciones más relevantes desde el punto de vista de su utilidad y rendimiento dentro del conjunto de la aplicación.

4.1 ESTRUCTURA DEL SOFTWARE

El software implementado en este proyecto se organiza en tres programas independientes, cada uno encargado de una función específica dentro del sistema y estructurados según un flujo jerárquico de control, en el que un script principal coordina la ejecución de un módulo subordinado, a su vez, invoca funcionalidades adicionales en función de las necesidades del sistema. Esta organización modular permite mantener el código más claro y estructurado, facilita su mantenimiento y mejora la escalabilidad del sistema, al separar de manera lógica las distintas responsabilidades.

A continuación, se describen los scripts desarrollados y se resume la función de cada uno, así como las clases que los componen.

4.1.1 HUSKY_GYMKHANA.PY

El script *husky_gymkhana.py* constituye el módulo principal del sistema y tiene como objetivo coordinar la simulación completa del robot dentro del entorno de trabajo. Su estructura permite integrar y gestionar los distintos componentes del sistema manteniendo un flujo jerárquico de control.

En primer lugar, se inicializa la simulación mediante la creación de una instancia de la clase ‘Simulation’ definida en *simulation.py*, que, como se ha descrito en el apartado anterior, se encarga de establecer la comunicación con el simulador

CoppeliaSim. A continuación, se crean y se configuran los distintos elementos del sistema robótico como el propio robot (‘HuskyRobot’) o el sensor LiDAR simulado (‘Ouster’). Esta fase garantiza que todos los elementos necesarios para la percepción, planificación y control del movimiento se encuentren correctamente disponibles dentro del entorno simulado.

Posteriormente, se define la secuencia ordenada de movimientos que el robot debe ejecutar para completar el recorrido de la yincana. Cada movimiento se especifica mediante una posición final y un eje principal de desplazamiento, lo que permite planificar la trayectoria de forma progresiva y controlada. Tras la ejecución de cada segmento, se actualiza la información relativa a los bordes u obstáculos detectados, de manera que esta información pueda reutilizarse en los siguientes tramos del recorrido.

Finalmente, una vez completada la secuencia de movimientos, el script detiene la simulación mediante la llamada al método ‘simulation.stop()’, liberando los recursos y finalizando la ejecución del sistema.

En conjunto, *husky_gymkhana.py* permite integrar todos los diferentes componentes del sistema (robot, sensores y módulos de planificación), coordinando su funcionamiento y asegurando que el robot siga correctamente la ruta definida a lo largo del recorrido, al mismo tiempo que facilita la supervisión del comportamiento del sistema durante la simulación.

4.1.2 MOVEMENT.PY

El script *Movement.py* implementa la clase ‘PathPlanner’, que constituye el módulo encargado del control del movimiento del robot dentro del entorno de simulación. Su objetivo principal es calcular y ejecutar los desplazamientos necesarios para que el robot siga la trayectoria definida, ya sea mediante movimientos en línea recta o trayectorias generadas con el algoritmo RRT en presencia de obstáculos.

La clase ‘PathPlanner’ almacena la información esencial para el control del movimiento, incluyendo la posición objetivo, el eje principal de desplazamiento y la posición y eje del movimiento anterior, lo que permite determinar con precisión

el estado inicial del robot antes de cada tramo. Adicionalmente, mantiene referencias a los objetos simulados (robot, sensores y entorno) y una estructura de datos destinada a almacenar los puntos de borde detectados durante la navegación. Asimismo, incorpora parámetros relacionados con el sentido de avance (marcha adelante o atrás) y con la identificación del último tramo del recorrido.

En primer lugar, el script genera puntos intermedios de referencia (waypoints), que permiten que el robot avance de forma progresiva y controlada a lo largo del eje principal de movimiento. A continuación, se calculan las velocidades lineales y angulares necesarias para el control del robot. Dependiendo de la naturaleza del segmento a recorrer, trayectoria recta o trayectoria generada mediante RRT, se ajustan dinámicamente los valores de velocidad para corregir el error angular y mantener el desplazamiento hacia el objetivo.

Además, el sistema reduce la velocidad cuando el robot se aproxima al objetivo final o cuando se detectan situaciones de riesgo, como inclinaciones excesivas o posibles colisiones, garantizando así un movimiento seguro y estable.

Todo este proceso se coordina a través del método ‘move_to_target’, que gestiona de forma integrada la verificación del progreso, el cálculo de trayectorias y el envío de comandos de velocidad al robot. Este método también contempla la posibilidad de saltar waypoints en caso de que un punto no pueda ser alcanzado por la presencia de un obstáculo o la planificación falle en un punto concreto, lo que permite mantener la continuidad del movimiento sin interrumpir la simulación.

En conjunto, *Movement.py* constituye el vínculo entre la planificación de trayectorias y la ejecución física del movimiento en el entorno simulado, asegurando que el robot siga la ruta definida de manera precisa y segura.

4.1.3 EDGE_RRT_PLANNER.PY

El script *edge_rrt_planner.py* integra dos bloques funcionales principales: la detección de bordes en nubes de puntos tridimensionales y la planificación de trayectorias mediante el algoritmo RRT. Este módulo proporciona al sistema la

capacidad de percibir el entorno y de generar rutas seguras en presencia de obstáculos.

En primer lugar, se define la clase ‘EdgeDetector’, responsable de procesar la información proporcionada por el sensor LiDAR. Esta clase aplica el método diferencia de normales (DoN) con el objetivo de detectar variaciones bruscas en la geometría que se puedan corresponder con obstáculos. A partir de esta información, se filtran los puntos relevantes según criterios de distancia mínima y máxima, y se identifican aquellas zonas que pueden representar un riesgo de colisión al robot. Como resultado, el sistema clasifica la nube de puntos en dos conjuntos: puntos considerados obstáculos y puntos considerados espacio libre, que serán utilizados posteriormente en el proceso de planificación.

El bloque de planificación de trayectorias se apoya en dos clases. En primer lugar, la clase ‘Node’, que representa los nodos del árbol RRT y almacena las coordenadas bidimensionales de cada nodo junto con el índice del nodo padre, lo que permite reconstruir la trayectoria una vez alcanzado el objetivo.

En segundo lugar, la clase ‘RRT’ implementa el algoritmo de planificación de rutas. Este algoritmo construye de forma incremental un árbol de nodos a partir de la posición inicial, generando muestras aleatorias en el espacio libre y conectándolas con los nodos existentes siempre que no se detecten colisiones. Para mejorar la eficiencia computacional, se utiliza una estructura de datos tipo KDTree, que permite realizar búsquedas rápidas tanto en el conjunto de puntos libres como en los obstáculos detectados. Sin embargo, cuando el número de obstáculos es reducido (inferior a un umbral determinado), se prescinde de esta estructura debido al sobrecoste computacional que supone su construcción.

Adicionalmente, el algoritmo incorpora mecanismos para mejorar la robustez, como la corrección automática del objetivo en caso de que este se encuentre en una zona no visible, así como la generación de un objetivo alternativo cuando la posición inicial se detecta en colisión. Asimismo, se implementa una verificación discreta de colisiones a lo largo de los segmentos de trayectoria entre nodos, garantizando que las rutas generadas sean seguras.

Finalmente, el módulo permite la visualización gráfica del árbol generado y de la trayectoria final, facilitando el análisis del comportamiento del algoritmo y la validación de los resultados obtenidos durante la simulación.

En conjunto, *edge_rrt_planner.py* proporciona las capacidades de percepción del entorno y planificación de rutas, actuando como nexo entre los datos sensoriales proporcionados por el LiDAR y el control de movimiento del robot.

Esta estructura jerárquica de ejecución garantiza que la información fluya de manera ordenada desde el script principal hacia los módulos subordinados, manteniendo una clara separación de responsabilidades y facilitando futuras ampliaciones del sistema, como la incorporación de nuevos algoritmos de planificación o integración de sensores adicionales.

4.2 DIAGRAMA DE FLUJO

A continuación, en la Figura 17 se presenta el diagrama de flujo correspondiente al funcionamiento general del software desarrollado. En él se muestra de forma estructurada cómo interactúan los diferentes módulos del sistema, así como la secuencia de procesos que permiten al robot percibir el entorno, planificar rutas y ejecutar los movimientos necesarios durante la yincana.

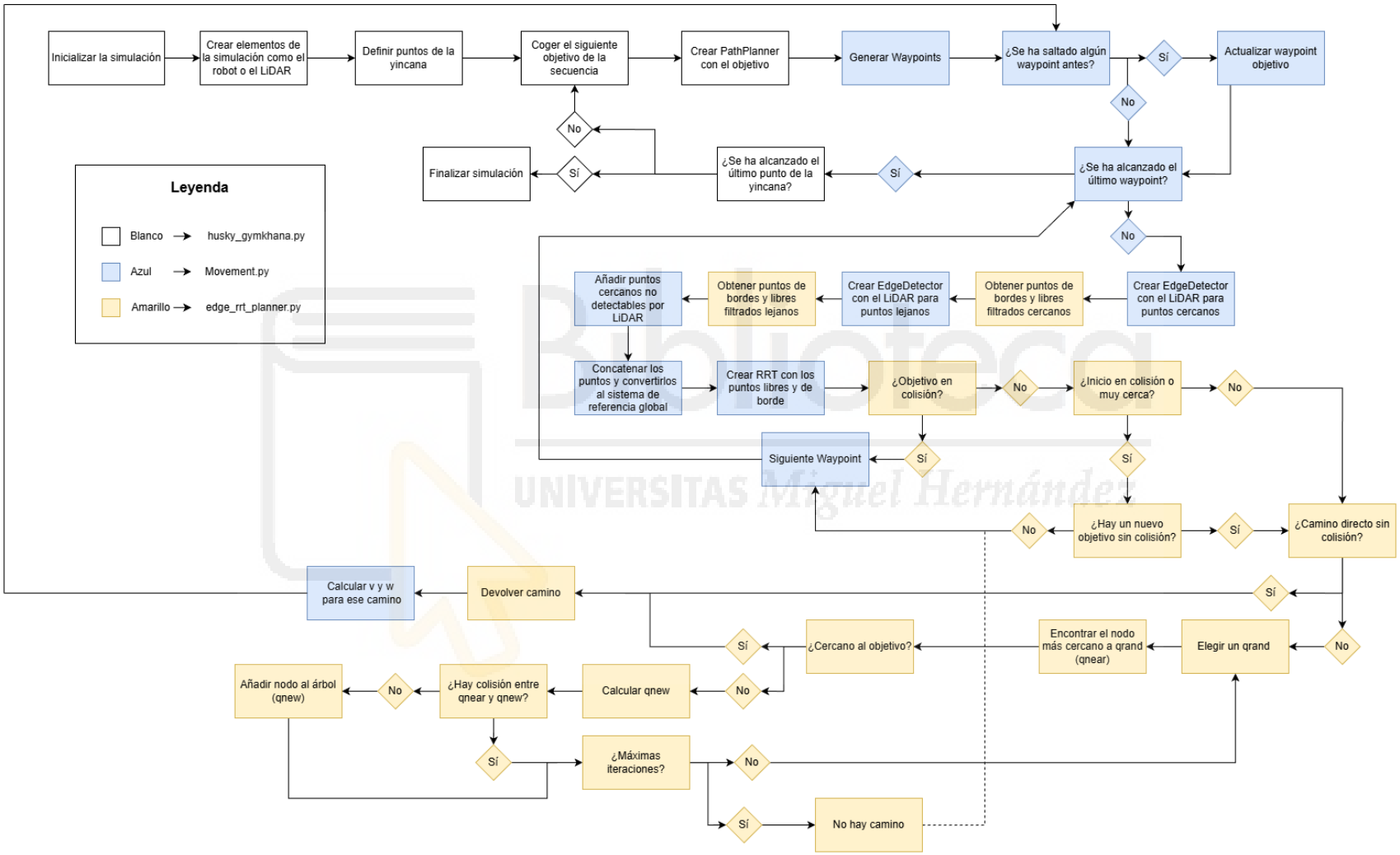

Figura 17. Diagrama de flujo del software.

El diagrama de flujo presentado refleja de manera estructurada el funcionamiento completo del sistema desarrollado, mostrando la relación jerárquica entre los tres módulos del software (*husky_gymkhana.py*, *Movement.py* y *edge_rrt_planner.py*). En la parte superior se representa el proceso de inicialización, donde el sistema configura el entorno de simulación, carga el robot y define los puntos que componen la yincana. A partir de estos datos, el script principal selecciona el siguiente objetivo y en el módulo de control (*Movement.py*) se generan los waypoints intermedios necesarios para avanzar hacia él.

Posteriormente, el flujo se ramifica hacia el módulo de planificación, donde se procesa la información proporcionada por el sensor LiDAR. Este módulo lleva a cabo la detección de bordes y la obtención de puntos libres, tanto cercanos como lejanos, para caracterizar el entorno. La detección de bordes se realiza en dos fases, ya que la primera los parámetros están ajustados para encontrar bordes cercanos y, en la segunda para detectar los lejanos. Con dichos puntos se ejecuta el algoritmo RRT, que comprueba si existe un camino libre de colisiones hacia el objetivo. En caso de detectar colisión o imposibilidad de avanzar directamente, se ajustan los criterios de búsqueda y se generan nuevas propuestas de nodos hasta encontrar una ruta valida o determinar que no existe.

Una vez que el planificador devuelve el camino válido, el módulo de control calcula las velocidades lineales y angulares necesarias para que el robot siga la trayectoria definida. El sistema evalúa continuamente si el robot ha alcanzado el waypoint actual o si es necesario actualizar el waypoint objetivo a uno anterior. Este ciclo se repite de forma iterativa hasta completar la secuencia completa de puntos de la yincana.

En conjunto, el diagrama ilustra con claridad cómo cada módulo del software coopera para llevar a cabo las tareas de percepción, planificación y movimiento, y permite comprender de forma visual la estructura jerárquica y el flujo de información dentro del sistema.

4.3 FUNCIONES DE INTERÉS

Una vez expuesta la organización global del software y el flujo de interacción entre los distintos módulos del sistema, en este apartado se desarrollan de manera detallada aquellas funciones que resultan de especial importancia para el desarrollo del proyecto.

```python
# Calculates angular velocity and adjusts linear speed based on angular error and movement direction.
def calculate_turning_velocity(self, angular_error, v):
    abs_error = abs(angular_error)

    if abs_error < 1:
        w = 0
    elif abs_error > 15:
        if abs_error > 30:
            w = np.clip(angular_error / 5, -1.5, 1.5)
        else:
            w = np.clip(angular_error / 5, -1, 1)
        v = min(0.5, v)
    else:
        w = np.clip(angular_error / 10, -0.5, 0.5)

    # Reverse direction if moving backwards
    if self.direction == 'backward':
        v = -v

    return v, w
```

Figura 18. Función 'calculate_turning_velocity'.

La Figura 18 ilustra el código de la función 'calculate_turning_velocity' del script *Movement.py*, la cual se utiliza para calcular tanto la velocidad angular (ω) como la velocidad lineal (v) necesarias en función del error angular del robot respecto a su objetivo.

Además, la función reduce su velocidad lineal cuando el error angular es mayor a 15º, limitando su valor máximo para favorecer una corrección estable del rumbo. En los casos en que el error angular es superior a 30º, la corrección angular se incrementa progresivamente, permitiendo una reacción más rápida ante desviaciones importantes. Como se puede apreciar en la figura, la velocidad angular es nula cuando el error angular es inferior a 1º, con el objetivo de evitar oscilaciones continuas que podrían provocar inestabilidades en el movimiento. Este umbral permite asumir pequeñas desviaciones angulares como aceptables.

La transición entre los distintos rangos de corrección angular se realiza de forma gradual, evitando sobrecorrecciones ante errores pequeños y favoreciendo una corrección rápida cuando el error es elevado.

Por último, si el movimiento se realiza en sentido inverso (‘backward’), la función invierte el signo de la velocidad lineal, manteniendo la coherencia del control independientemente del sentido de avance.

```python
# Generates a collision-free path using RRT. Returns: path_nodes: list of 2D path points,
# near_edges: detected edge points, start_collision: bool flag indicating collision at start
def path_points(self, goal_point, current_point):
    # Extract edge and free points
    near_detector = EdgeDetector(self.lidar, voxel=0.12, small_radius=0.2, large_radius=0.3, min_distance=0, max_distance=4, threshold=0.045, margin=0.25)
    near_edges, near_free = near_detector.get_edges()
    far_detector = EdgeDetector(self.lidar, voxel=0.25, small_radius=0.4, large_radius=1, min_distance=4, max_distance=10, threshold=0.15, margin=0.25)
    far_edges, far_free = far_detector.get_edges()

    # Add synthetic free points (rings of points) around the robot at ground level
    min_z = min(near_free[:, 2])
    rings = [np.array([[0, 0, min_z]])]
    for r in np.arange(0.2, 2.5, 0.2):
        num_points = int(np.ceil(2 * np.pi * r / 0.2))
        thetas = np.linspace(0, 2 * np.pi, num_points, endpoint=False)
        xs = r * np.cos(thetas)
        ys = r * np.sin(thetas)
        zs = np.full_like(xs, min_z)
        ring_points = np.column_stack((xs, ys, zs))
        rings.append(ring_points)
    inside_points = np.vstack(rings)

    # Combine all edge and free points; far_free is excluded since it's already in near_free
    all_edges = np.concatenate((near_edges, far_edges), axis=0)
    all_free = np.concatenate((inside_points, near_free), axis=0)

    # Optional visualization of the combined point cloud (commented out)
    # all_points = np.concatenate((all_free, all_edges), axis=0)
    # colors_edges = np.tile([1.0, 0.0, 0.0], (all_edges.shape[0], 1))
    # colors_free = np.tile([0.5, 0.5, 0.5], (all_free.shape[0], 1))
    # lidar.from_points(all_points)
    # colors = np.concatenate((colors_free, colors_edges), axis=0)
    # lidar.choose_colors(colors)
    # lidar.draw_pointcloud()

    # Filter out edges that are above the robot sensor (keep only those below 0.2m)
    all_edges = all_edges[all_edges[:, 2] <= 0.2]
    near_edges = near_edges[near_edges[:, 2] <= 0.2]

    # Shift free and edge points relative to the robot's current position
    free = all_free[:, :2] + current_point
    current_edges = all_edges[:, :2] + current_point

    # Combine current and previously known edges
    edges = np.concatenate((current_edges, self.edges_before), axis=0)

    # Run RRT (Rapidly-exploring Random Tree) path planning
    rrt_instance = RRT(current_point, goal_point, free, edges)
    path_nodes, start_collision = rrt_instance.run()

    # Draw the RRT path if the goal corresponds to the final gymkhana point ([0, 0.5]) and drawing is enabled
    if np.array_equal(goal_point, np.array([0, 0.5])) and self.should_draw_path :
        rrt_instance.draw(path_nodes)

    return path_nodes, near_edges[:, :2] + current_point, start_collision
```

Figura 19. Función ‘path_points’.

En la Figura 19 se muestra la función ‘path_points’ del script *Movement.py*, cuyo objetivo es obtener un camino libre de colisiones hacia el siguiente waypoint, empleando el algoritmo RRT cuando sea necesario.

Tal y como se ha descrito en el apartado anterior, la función calcula los puntos de borde y los puntos libres en dos fases, una para el entorno cercano y otra para el entorno lejano, utilizando distintos parámetros del detector de bordes.

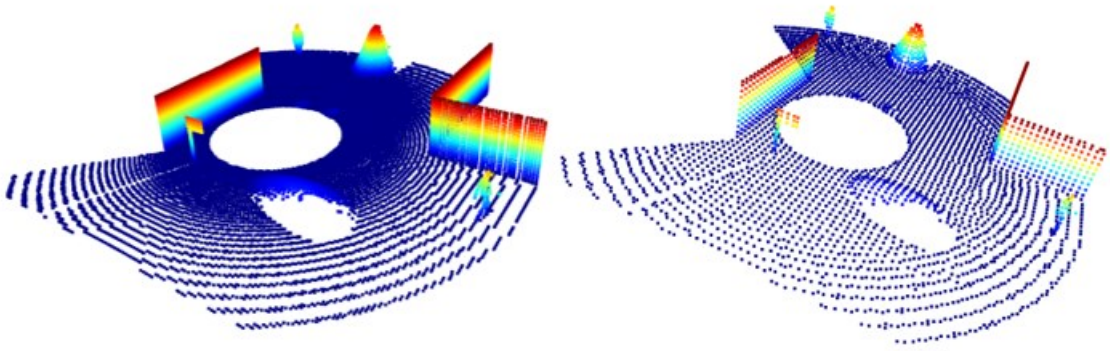

Figura 20. Nube de puntos LiDAR tras aplicar un voxelizado de alta resolución (0.12 m) a la izquierda y de baja resolución (0.25 m) a la derecha.

Como se observa en la Figura 20, la diferencia en el parámetro de voxelizado es fundamental para este proceso. Mientras que el valor de 0,12m genera una nube de puntos densa que permite identificar obstáculos cercanos, el valor de 0,25m simplifica la geometría del entorno lejano. Esta doble resolución permite detectar correctamente los obstáculos tanto cercanos como lejanos.

A continuación, se generan puntos libres artificiales en forma de anillos concéntricos alrededor del robot, con un espaciado de 0,2 m y hasta una distancia máxima de 2,5 m, con el fin de compensar las limitaciones del campo de visión del LiDAR. Este sensor presenta una apertura horizontal de 90º, por lo que existen zonas próximas al robot que no son directamente observables. Se asume que dichos puntos son transitables, ya que cualquier obstáculo presente habría sido detectado en iteraciones anteriores.

Posteriormente, se combinan los puntos libres y los puntos de obstáculo detectados, y se filtran los puntos de obstáculo cuya altura es superior a 0,2 m respecto al plano del sensor, ya que no representan un riesgo de colisión directa para el robot.

Los puntos se expresan en el sistema de referencia global de la simulación, y se descarta la componente vertical (eje Z), dado que el robot se desplaza en un

plano bidimensional. Aunque el entorno pueda presentar desniveles, se asume que cualquier punto clasificado como obstáculo no debe ser atravesado independientemente de su altura.

Finalmente, se ejecuta el algoritmo RRT utilizando los puntos libres y de obstáculo obtenidos, y si el objetivo corresponde al punto final de la yincana y está habilitada la visualización, se muestra por pantalla el árbol RRT y la trayectoria resultante junto con el mapa completo que recoge la posición de los obstáculos.

```python
# Filter points likely to be in collision based on a bounding box
# around the vehicle and vertical clearance constraints.
def filter_collision_indices(self, data):
    if self.min_distance > 2:
        return np.array([], dtype=int)

    # Vehicle dimensions and sensor height
    size_x, size_y, size_z = 0.67, 0.99, 1.25
    clearance = 0.13
    lidar_height = 1.1325

    x, y, z = np.abs(data[:, 0]), np.abs(data[:, 1]), data[:, 2]
    mask = ((x - self.margin < size_x / 2) &
            (y - self.margin < size_y / 2) &
            (clearance < (lidar_height + z)) &
            ((lidar_height + z) < size_z))
    return np.where(mask)[0]
```

Figura 21. Función 'filter_collision_indices'.

En la Figura 21 se presenta la función 'filter_collision_indices' del script *edge_rrt_planner.py*. Esta función se encarga de detectar puntos potencialmente peligrosos cercanos al robot, por lo que no se aplica cuando se analizan puntos lejanos ('self.min_distance > 2').

Para identificar dichos puntos, se define un volumen tridimensional equivalente a las dimensiones del robot, al que se le añade un margen de seguridad. Se aplica una máscara sobre la nube de puntos para seleccionar únicamente aquellos que se encuentran dentro de este volumen y que podrían provocar una colisión.

Además, se descartan los puntos situados por debajo de la base del robot, evitando así clasificar como obstáculos el suelo o pequeños desniveles que no afectan a la navegación.

```python
# Adjusts the goal if the start position it is in collision.
# Returns a valid nearby point if a collision-free path can be established.
def start_collision(self, obstacles):
    candidates = self.find_candidates(self.start, obstacles)

    if candidates is None or len(candidates) == 0:
        print("[Start Collision] No nearby candidate points found.")
        return None  # No valid candidates found near the start

    self.start_check_collision = True   # Lower collision threshold during checking

    # Filter candidates where a collision-free path from the start exists
    valid_candidates = []
    for i in range(len(candidates)):
        if not self.check_path(candidates[i], self.start, obstacles):
            valid_candidates.append(candidates[i])

    self.start_check_collision = False

    if len(valid_candidates) == 0:
        print("[Start Collision] All nearby candidates result in collision.")
        return None  # No valid path to any candidate

    valid_candidates = np.array(valid_candidates)

    # Select candidate closest to the original goal
    closest_index = np.argmin(np.linalg.norm(valid_candidates - self.goal, axis=1))
    return valid_candidates[closest_index]
```

Figura 22. Función 'start_collision'.

La Figura 22 ilustra la función 'start_collision' del script *edge_rrt_planner.py*. Esta función resulta fundamental cuando la posición inicial del robot se encuentra demasiado próxima a un obstáculo, situación en la que el algoritmo RRT puede no ser capaz de generar un camino válido.

En estos casos, la función busca puntos cercanos libres de colisión desde los cuales sea posible establecer un desplazamiento directo, permitiendo al robot alejarse de la zona conflictiva. Por ello, se evalúan candidatos cercanos y se selecciona aquel que permite una trayectoria libre de colisiones.

Si no se encuentra ningún punto valido, la función devuelve un valor nulo, lo que indica que el robot no dispone de una salida viable desde su posición actual y, por tanto, se considera que ha quedado bloqueado.

```python
# Generate a random configuration in the space
def rand_conf(self, near_rand):
    if near_rand:
        # Bias sampling near the goal
        return np.array([
            np.random.uniform(self.goal[0] - self.goal_threshold, self.goal[0] + self.goal_threshold),
            np.random.uniform(self.goal[1] - self.goal_threshold, self.goal[1] + self.goal_threshold)
        ], dtype=np.float32)
    else:
        # Uniform sampling within the space bounds
        return np.array([random.uniform(*self.x_bounds), random.uniform(*self.y_bounds)], dtype=np.float32)
```

Figura 23. Función 'rand_conf'.

Finalmente, se presenta la función 'rand_conf' del script *edge_rrt_planner.py*. Esta función se encarga de generar configuraciones aleatorias ('q_rand') para la expansión del árbol RRT como se observa en la Figura 23.

Cuando el robot se encuentra lejos del objetivo, las muestras se generan de forma uniforme dentro de los límites del espacio libre. Sin embargo, cuando el árbol se aproxima al objetivo, se activa un sesgo que concentra las muestras dentro de una región cercana al punto objetivo, aumentando así la probabilidad de conectar con él y reduciendo el tiempo de planificación.

5. RESULTADOS

En este capítulo se presentan y analizan los resultados obtenidos tras la aplicación de los métodos de planificación y control desarrollados a lo largo de este trabajo. El objetivo principal de este apartado es evaluar el comportamiento del sistema en distintos escenarios, analizando su capacidad para generar trayectorias válidas, seguirlas correctamente y operar bajo condiciones compatibles con un funcionamiento en tiempo real.

Para ello, se han definido varios entornos de prueba con distintos niveles de complejidad, que permiten estudiar de forma progresiva el rendimiento del sistema, desde situaciones ideales sin obstáculos hasta escenarios altamente restrictivos. En cada uno de estos entornos se han realizado diferentes pruebas orientadas a evaluar tanto el algoritmo de planificación como el de seguimiento de trayectorias. Asimismo, se analizan las limitaciones observadas y se discute la viabilidad del enfoque propuesto para su aplicación en un robot móvil real.

5.1 ENTORNOS DE PRUEBA

En este apartado se describen los entornos de prueba utilizados para la evaluación del sistema desarrollado. Estos entornos han sido diseñados con el objetivo de analizar el comportamiento de los algoritmos de planificación y seguimiento bajo distintos niveles de complejidad, permitiendo una evaluación progresiva del rendimiento del sistema.

Con el fin de estudiar de manera sistemática la influencia del entorno en los resultados obtenidos, se han definido tres escenarios diferenciados.

Cada uno de ellos presenta un nivel de dificultad creciente, lo que permite evaluar desde el funcionamiento básico del sistema hasta su comportamiento en situaciones altamente restrictivas.

5.1.1 ENTORNO 1: ENTORNO SIN OBSTÁCULOS

El primer entorno corresponde a un escenario ideal sin presencia de obstáculos, en el que el robot dispone de un espacio completamente libre para desplazarse. Este entorno se utiliza como caso base para validar el correcto funcionamiento del sistema, centrándose principalmente en el seguimiento de trayectorias y en la ejecución de movimientos suaves y estables.

Al no existir restricciones externas, este escenario permite analizar el comportamiento del controlador y del algoritmo de seguimiento de puntos de manera aislada, sin necesidad de activar el planificador de rutas ni el detector de obstáculos. De este modo, sirve como referencia para la comparación con entornos más complejos. A continuación, en las Figuras 24 a 26 se muestra la configuración del entorno 1 desde diferentes puntos de vista.

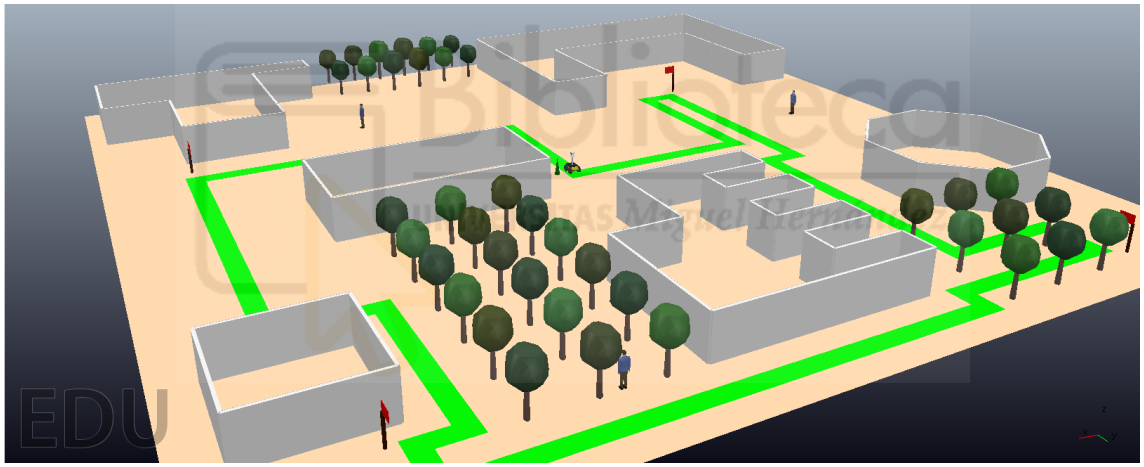

Figura 24. Vista general del entorno 1.

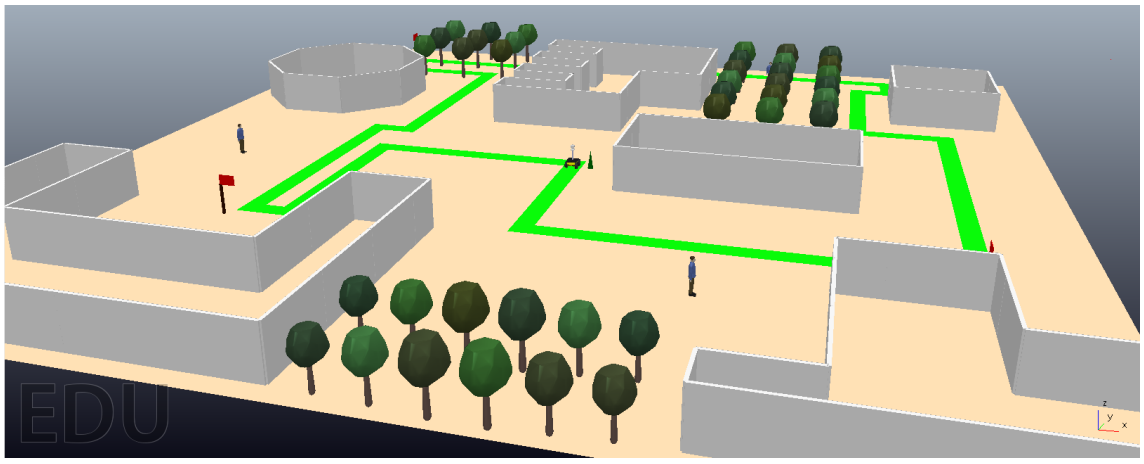

Figura 25. Vista alternativa del entorno 1.

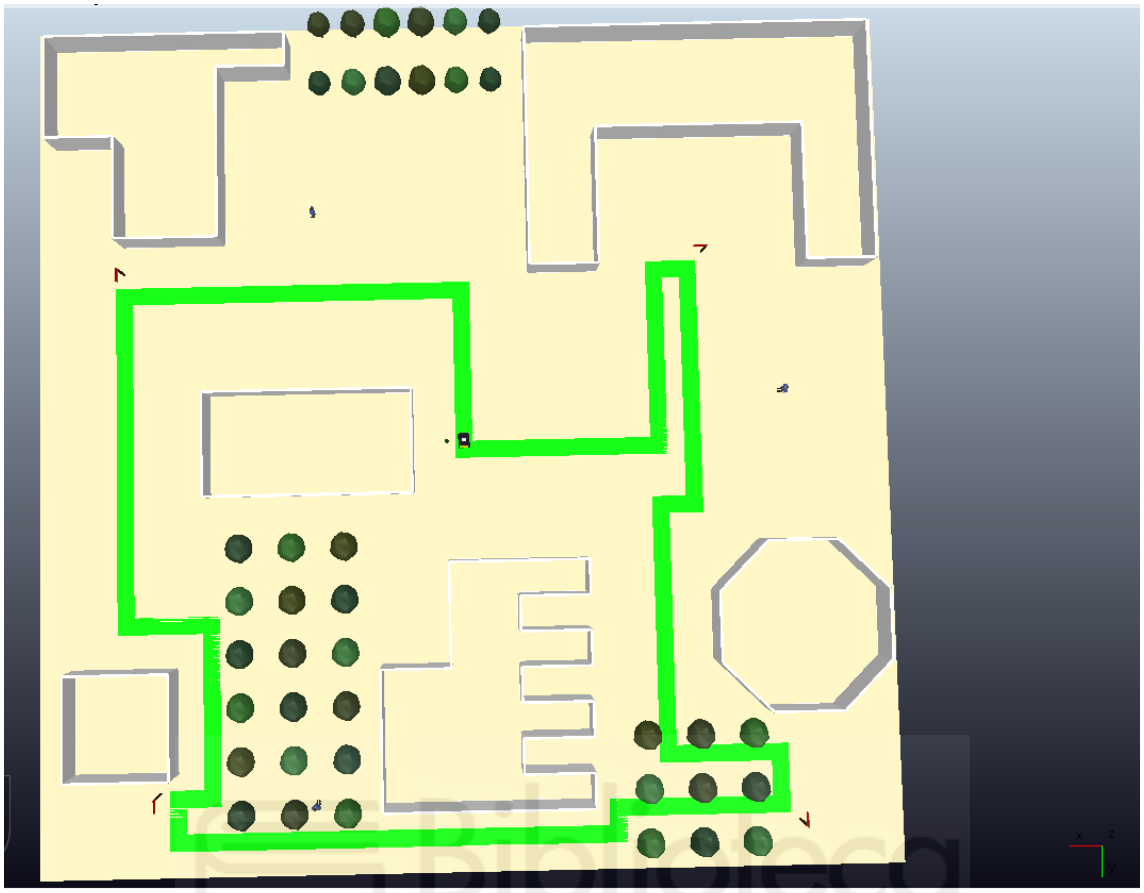

Figura 26. Vista superior (2D) del entorno 1.

Como se puede apreciar en las figuras, en este entorno el robot debe moverse por el camino marcado en verde sin tener que esquivar ningún obstáculo. Además, el recorrido definido es de tipo circular, de modo que el punto de inicio coincide con el punto final.

En el entorno, los objetos representados en gris corresponden a edificios sin techo, ya que al robot únicamente le afectan las paredes verticales. Esta simplificación permite reducir la información innecesaria en la simulación y mejorar la eficiencia del sistema.

Asimismo, en las Figuras 24 y 25 se puede ver el robot Husky utilizado en la simulación, junto al cual se encuentra un pequeño cono verde que indica la posición inicial del robot. Además, se han añadido cuatro banderas rojas distribuidas a lo largo del circuito que actúan como puntos de referencia visuales para indicar el recorrido que debe seguir el robot.

5.1.2 ENTORNO 2: ENTORNO CON OBSTÁCULOS DE DIFICULTAD MEDIA

El segundo entorno introduce un conjunto de obstáculos distribuidos de forma que el robot debe realizar maniobras de planificación y evasión para alcanzar el objetivo. La disposición de los obstáculos genera trayectorias con cambios de dirección moderados, sin llegar a configuraciones extremadamente restrictivas. Este entorno parte de la misma configuración que el entorno 1, pero incorpora obstáculos adicionales en el camino que debe seguir el robot.

Este escenario permite evaluar la capacidad del sistema para generar trayectorias viables en presencia de obstáculos, así como la calidad del seguimiento de dichas trayectorias, analizando posibles desviaciones, incrementos en el tiempo de ejecución y limitaciones derivadas de la planificación. En las Figuras 27 a 30 se muestra el entorno desde diferentes puntos de vista para mayor detalle.

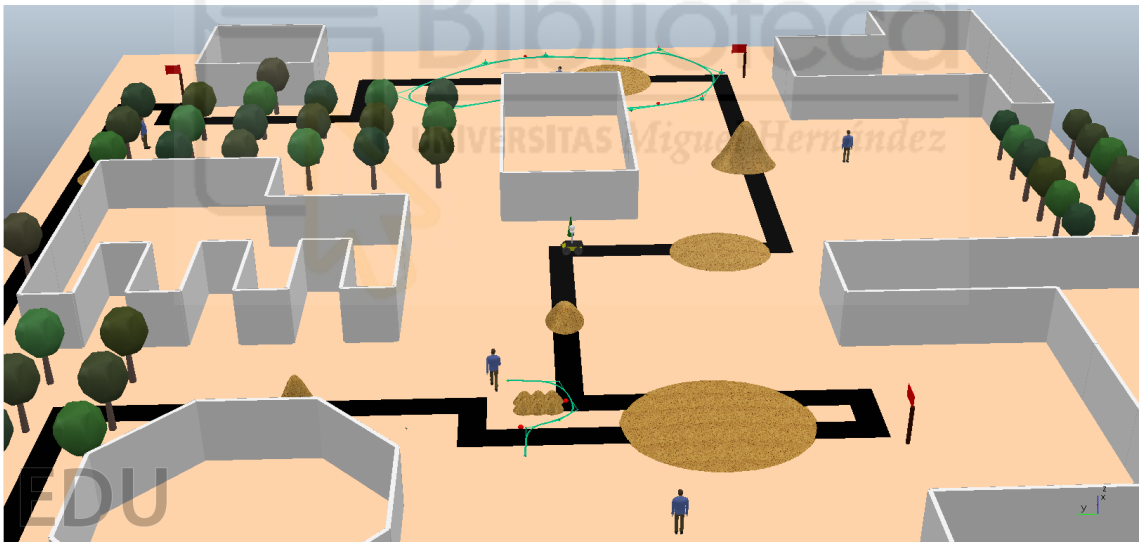

Figura 27. Vista general del entorno 2.

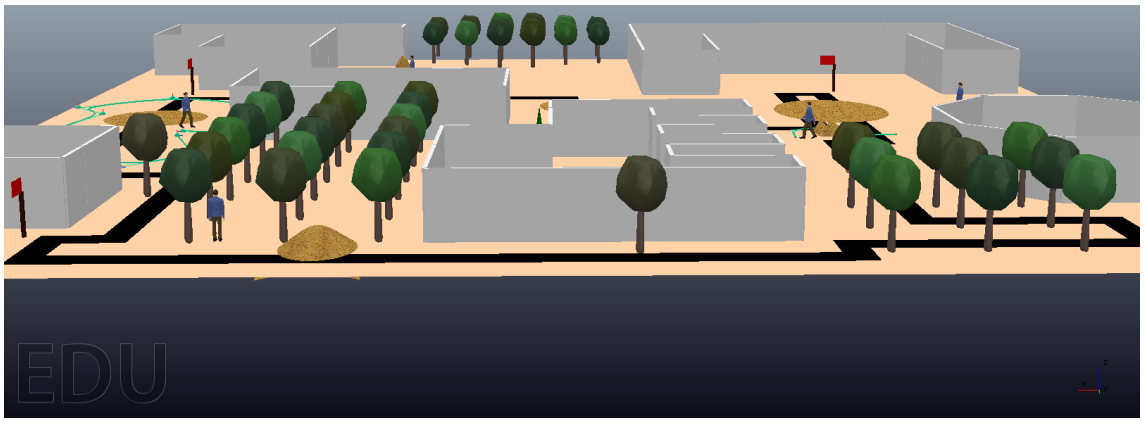

Figura 28. Vista alternativa del entorno 2.

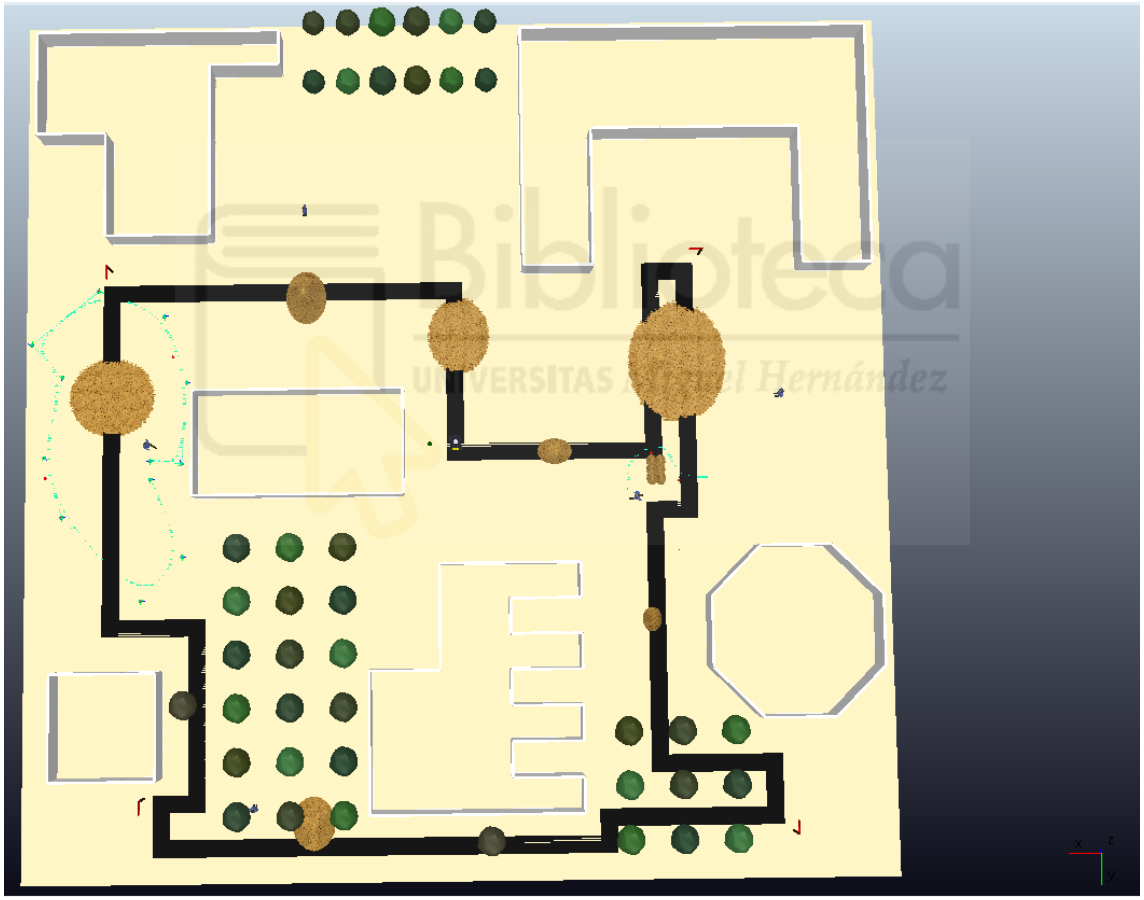

Figura 29. Vista superior (2D) del entorno 2.

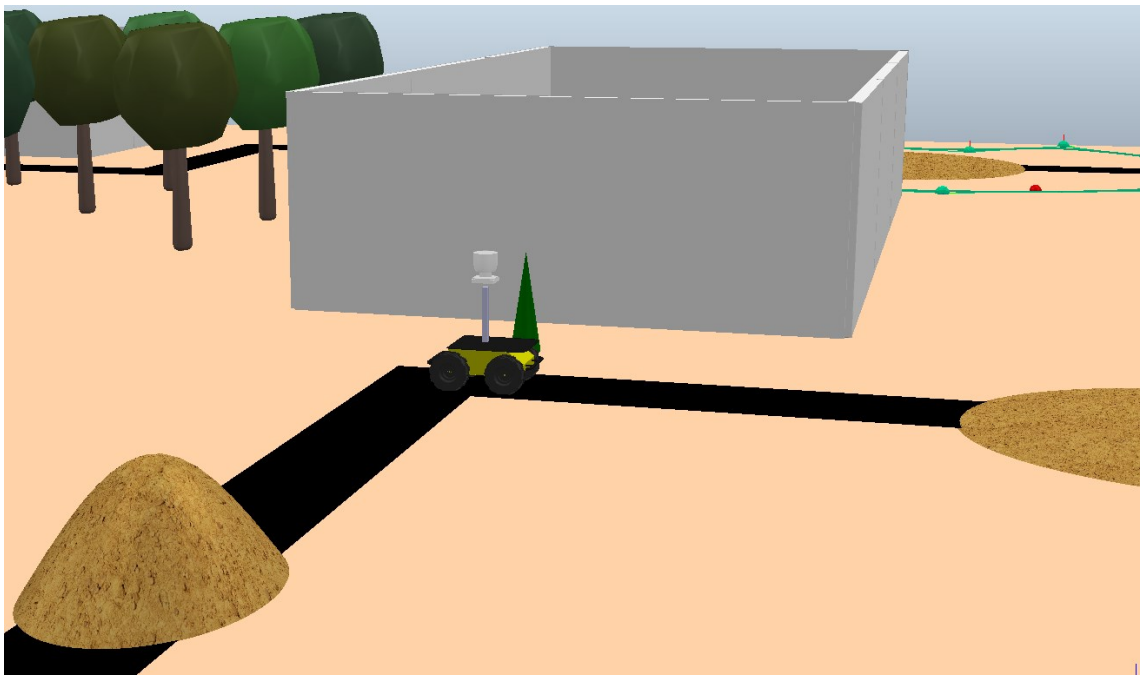

Figura 30. Vista del robot Husky en el entorno 2.

Como se puede observar en las Figuras 27 a 29, el entorno 2 es similar al entorno 1, aunque presenta varias diferencias relevantes. En primer lugar, el camino por el que debe moverse aparece en negro, simulando una carretera. Además, se han añadido distintos tipos de obstáculos sobre el recorrido.

Entre estos obstáculos se incluyen baches, representados en color marrón claro, así como dos personas en movimiento, que siguen trayectorias prefijadas. Las líneas de color azul claro indican el recorrido seguido por cada una de estas personas. Adicionalmente, hay otras 2 personas estáticas que no interfieren en el recorrido del robot como en el entorno anterior.

Asimismo, se han añadido dos árboles que actúan como obstáculos adicionales, visibles en la parte inferior de la Figura 29. Aunque en el entorno anterior existían zonas de arboleda, estas no afectaban al movimiento del robot.

Por último, en la Figura 30 se muestra el robot Husky en la posición de inicial del entorno, con el sensor LiDAR montado en la parte superior.

5.1.3 ENTORNO 3: ENTORNO CON OBSTÁCULOS DE DIFICULTAD MÁXIMA

El tercer entorno presenta una configuración altamente restrictiva, con una mayor densidad de obstáculos y zonas de paso más estrechas. Este escenario ha sido diseñado para poner a prueba los límites del sistema, forzando posibles fallos tanto en la fase de planificación como en el seguimiento de trayectorias.

En este entorno pueden darse situaciones en las que el algoritmo no sea capaz de encontrar una solución válida dentro de los parámetros establecidos, lo que permite identificar de forma clara las limitaciones del enfoque propuesto y analizar su comportamiento en condiciones extremas. En las Figuras 31 a 35 se muestra la configuración del entorno desde diferentes puntos de vista.

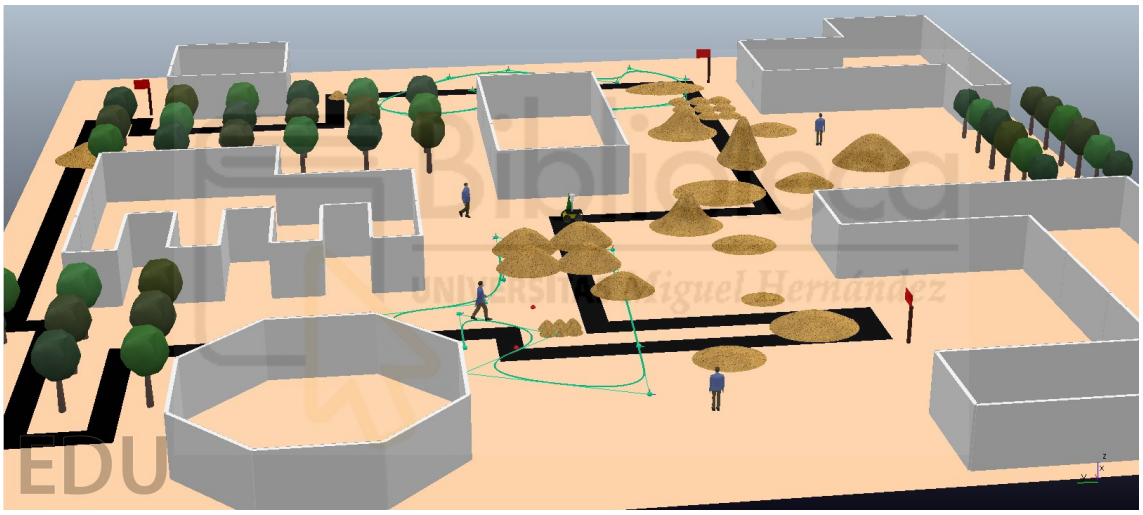

Figura 31. Vista general en perspectiva del entorno 3

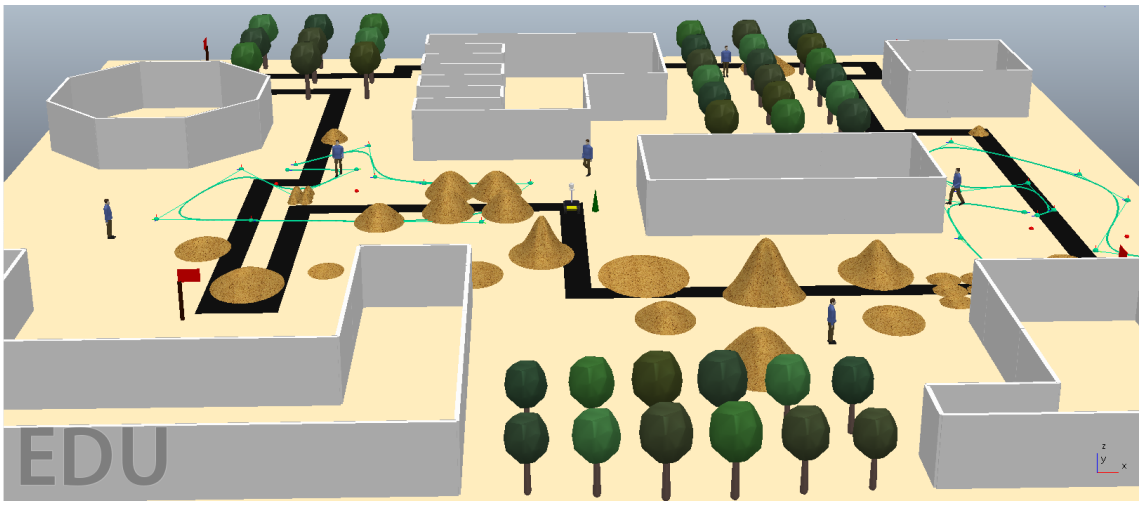

Figura 32. Vista frontal del entorno 3.

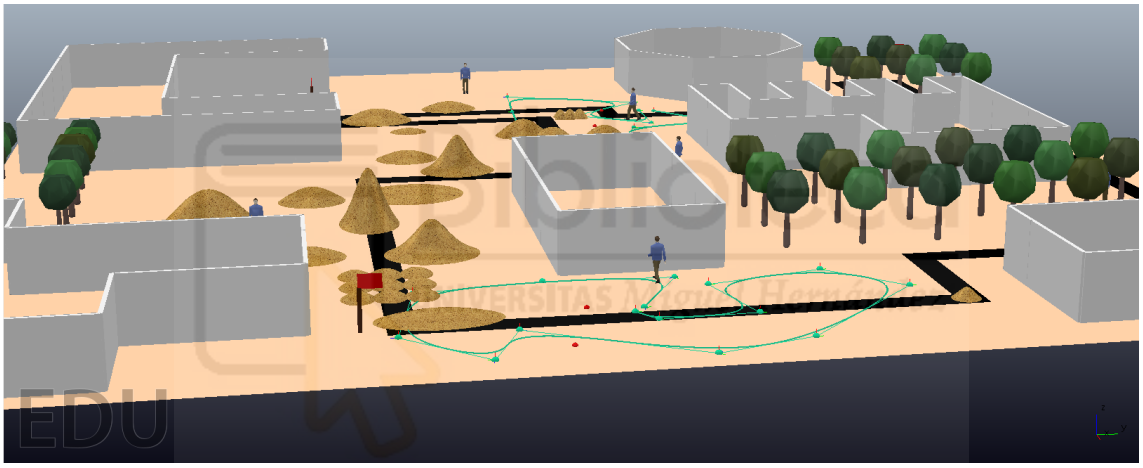

Figura 33. Vista lateral del entorno 3.

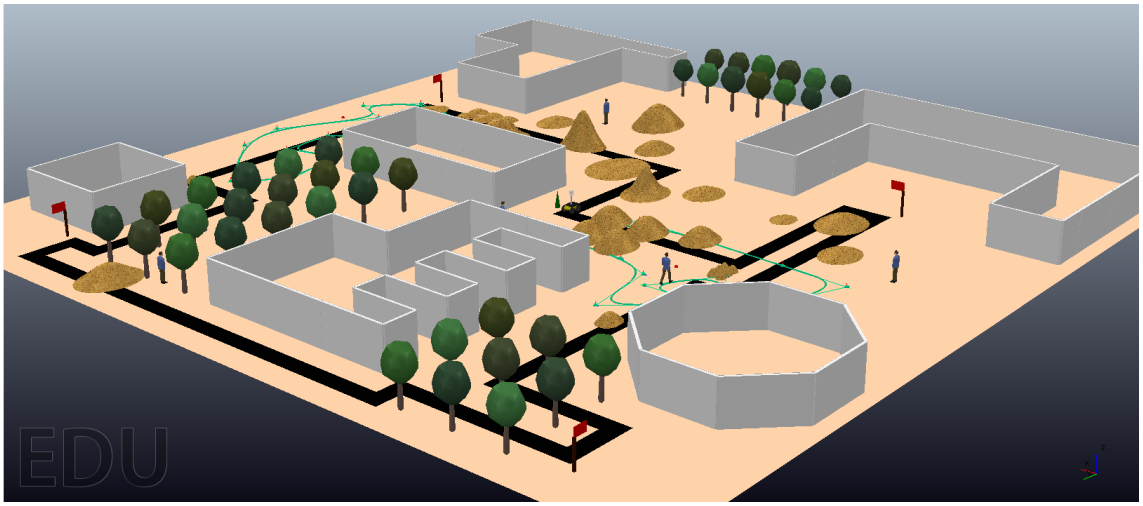

Figura 34. Vista isométrica posterior del entorno 3.

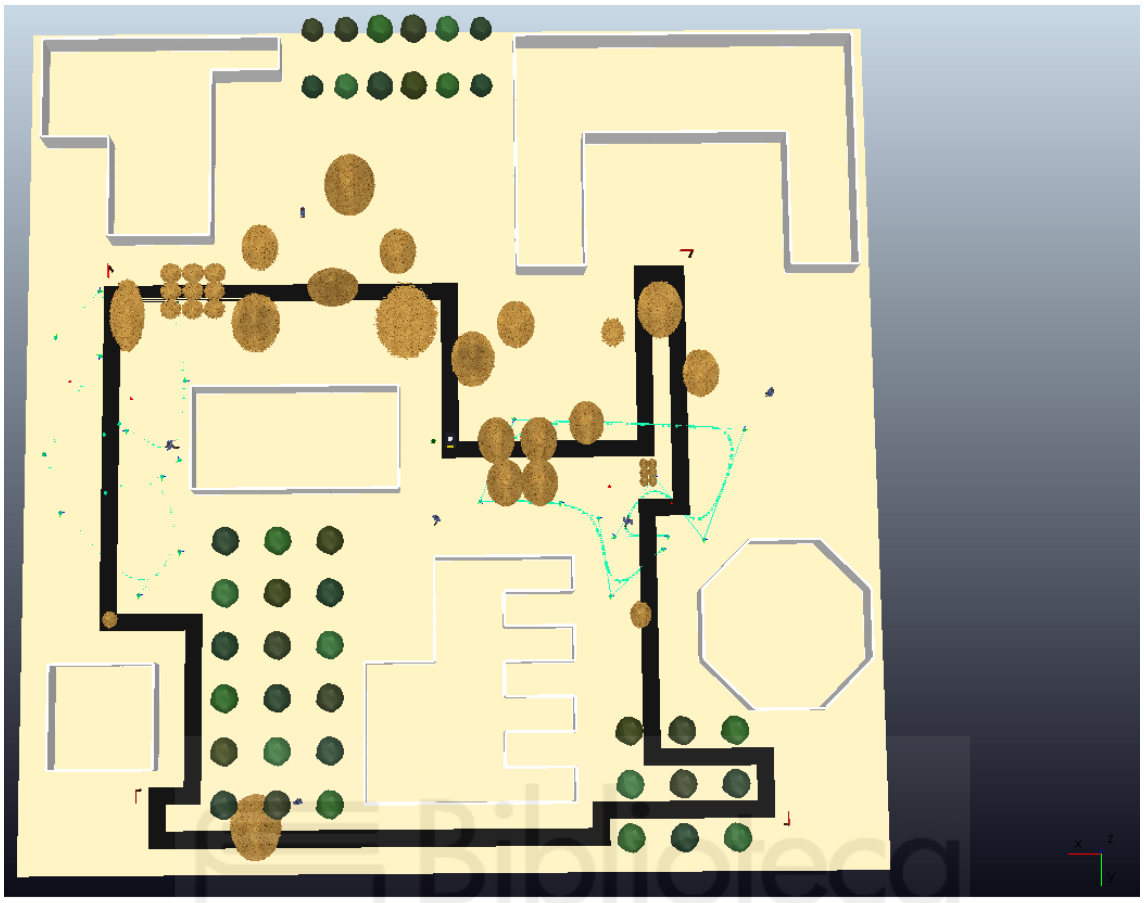

Figura 35. Vista superior (2D) del entorno 3.

Como se ilustra en las Figuras 31 a 35, el tercer entorno mantiene una estructura similar al entorno 2, incorporando baches y dos personas en movimiento que cruzan por el camino que debe seguir el robot. No obstante, la principal diferencia respecto al entorno anterior es la mayor densidad de baches en determinadas zonas, lo que incrementa significativamente la complejidad del escenario.

Con el objetivo de facilitar la comparación entre los tres escenarios definidos, en la Figura 36 se muestran los entornos desde un punto de vista similar.

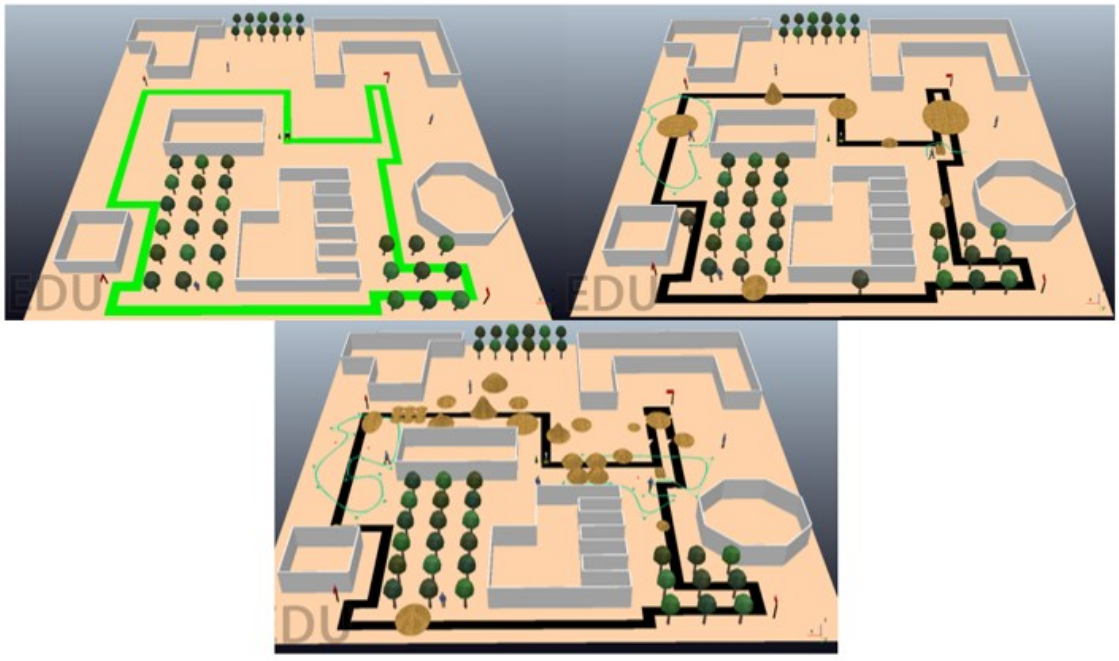

Figura 36. Comparación visual de los entornos de prueba: entorno 1 (superior izquierda), entorno 2 (superior derecha) y entorno 3 (parte inferior).

En conjunto, los tres entornos definidos permiten evaluar de manera progresiva el rendimiento del sistema desarrollado. El entorno sin obstáculos sirve como referencia para validar el funcionamiento básico, mientras que los entornos con dificultad media y máxima introducen restricciones crecientes que permiten analizar la capacidad de planificación, la robustez del seguimiento y las limitaciones del sistema.

5.2 SEGUIMIENTO DE TRAYECTORIAS

En este apartado se analiza el comportamiento del sistema durante el seguimiento del recorrido dentro del entorno de simulación. El objetivo principal es evaluar la capacidad del robot para seguir correctamente un camino previamente definido, centrándose en el desempeño del controlador y en la coherencia del movimiento generado, independientemente del método empleado para la generación de la trayectoria.

En primer lugar, se realizaron pruebas en el entorno 1, descrito en el apartado anterior, ya que la ausencia de obstáculos permite analizar el seguimiento de

trayectorias en condiciones ideales. Este entorno se utiliza como caso base para verificar que el sistema es capaz de seguir correctamente una trayectoria sin la influencia de factores externos, evaluando así el comportamiento del controlador de forma aislada.

Con el objetivo de mejorar la estabilidad del movimiento, se introdujo una reducción de la velocidad del robot antes de alcanzar tramos curvos. De este modo, se evita que la inercia provoque desviaciones significativas respecto a la trayectoria deseada, especialmente en giros de mayor curvatura. Este ajuste permite obtener un seguimiento más preciso y un movimiento más suave durante la ejecución del recorrido.

Una vez validado el correcto funcionamiento del sistema en el entorno sin obstáculos, el mismo esquema de control se aplicó a escenarios más complejos, como el entorno 2. En este caso, la ejecución de la trayectoria se realiza en presencia de obstáculos, lo que permite evaluar el comportamiento del sistema en condiciones más cercanas a un entorno realista dentro de la simulación.

El análisis del seguimiento se ha llevado a cabo considerando distintos tipos de trayectorias, con el fin de estudiar el comportamiento del sistema ante diferentes niveles de complejidad. En concreto, se han evaluado tramos rectilíneos, tramos curvos y trayectorias generadas mediante el algoritmo RRT para la evasión de obstáculos.

Adicionalmente, se ha llevado a cabo un primer ensayo orientado a comparar el desempeño del seguimiento de trayectorias utilizando dos algoritmos de control diferentes. El objetivo de este ensayo es analizar las diferencias en la capacidad de seguimiento, estabilidad y suavidad del movimiento generado por cada algoritmo frente a un mismo circuito. Para ello, ambos métodos se evaluaron en el escenario 1, ya que la ausencia de obstáculos permite analizar el comportamiento del algoritmo de seguimiento de forma directa y sin interferencias externas. En la Tabla II se presentan los resultados obtenidos para ambos algoritmos, incluyendo el error medio en todo el circuito y el tiempo total de ejecución.

Algoritmos	Error medio	Tiempo de ejecución
Algoritmo 1	5,87 mm	56:28
Algoritmo 2	55,65 mm	1:14:42

Tabla II. Comparación de algoritmos de seguimiento de trayectorias.

Tal como se observa en la tabla anterior, el algoritmo que presenta un mejor desempeño en el seguimiento de la trayectoria es el primero, el cual ha sido utilizado en el desarrollo de este trabajo. Sin embargo, el segundo algoritmo, aunque es muy parecido al primero, presenta claras diferencias en su funcionamiento. La principal diferencia radica en que emplea dos funciones diferenciadas dentro del algoritmo: una para el seguimiento en tramos rectos y otra para el seguimiento en tramos curvos.

En los tramos rectos, su comportamiento es similar al del primer algoritmo, ya que corrige continuamente la posición y orientación mediante giros suaves cuando este no se encuentra centrado en el camino. Sin embargo, en los tramos curvos el robot gira de forma constante a una velocidad baja (0,25 m/s) hasta conseguir alinearse correctamente con la dirección que debe seguir en dicho tramo. Una vez que el error es reducido, el algoritmo cambia a la función correspondiente a los tramos rectos.

Otra diferencia relevante es que, en lugar de utilizar puntos intermedios o waypoints a lo largo de los tramos, solo se le indica el punto de inicio y final de cada segmento, a partir de los cuales calcula cómo debe moverse para seguir el camino definido.

Este análisis comparativo permite justificar la selección del algoritmo de seguimiento empleado en el resto del estudio y sirve como base para el análisis detallado del rendimiento del sistema en función del tipo de trayectoria.

Para cuantificar el rendimiento del seguimiento, se ha comparado la trayectoria planificada con la trayectoria ejecutada por el robot durante la simulación, obtenida a partir de su posición a lo largo del tiempo. El análisis se basa en el cálculo del error medio de seguimiento para cada tipo de trayectoria. Con el fin

de obtener una evaluación representativa sin introducir redundancias, el error se ha calculado sobre dos tramos del recorrido para cada categoría, permitiendo así una comparación clara entre los distintos tipos de trayectoria analizados.

En la Figura 37 se muestran los valores del error medio de seguimiento obtenidos para cada tipo de trayectoria, lo que permite observar de manera directa la influencia de la complejidad del recorrido en la precisión del seguimiento. Además, en la Figura 38 se muestran dos capturas en las que se compara la trayectoria planificada con la trayectoria ejecutada. En ella se muestra una situación sin presencia de obstáculos y otra en la que el robot debe esquivar obstáculos durante la ejecución del recorrido.

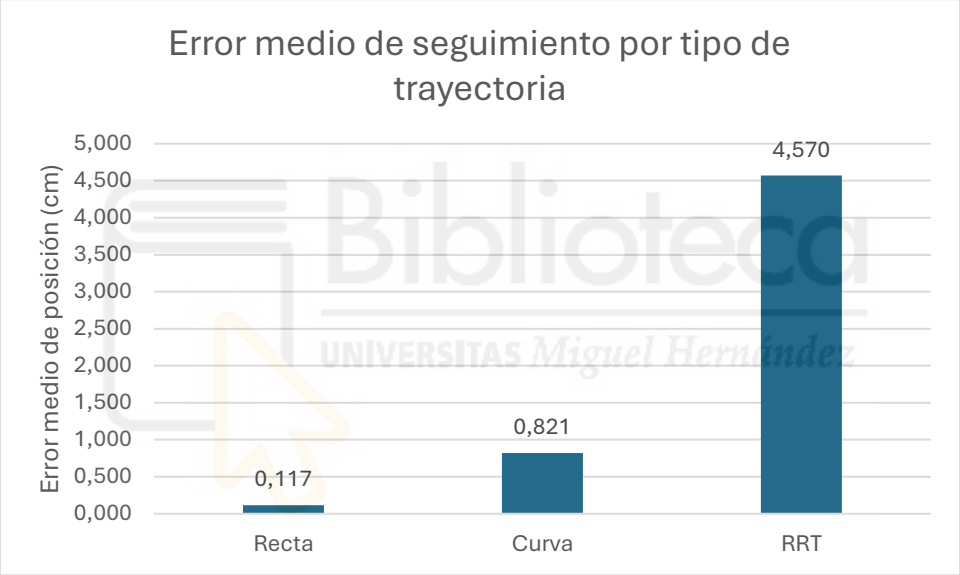

Figura 37. Error medio de seguimiento en función del tipo de trayectoria.

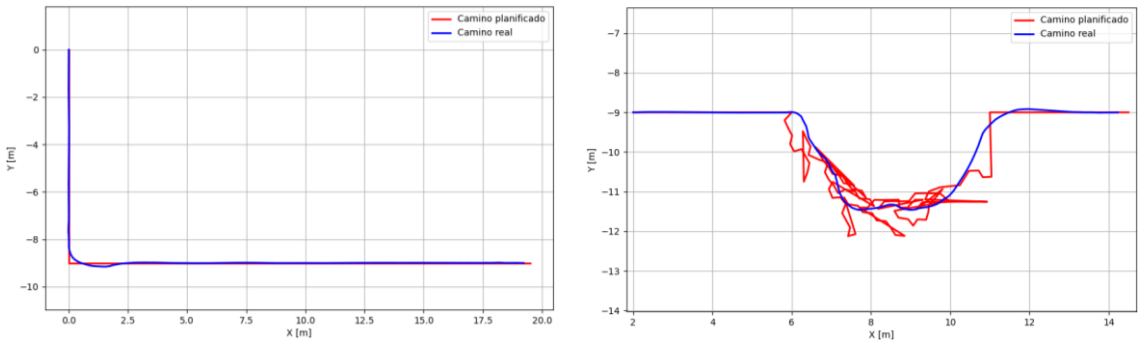

Figura 38. Comparación entre la trayectoria ejecutada y planificada en una zona sin obstáculos (izquierda) y en una zona con obstáculos (derecha).

Como se observa en las figuras, el error de seguimiento es mayor en las trayectorias generadas mediante RRT, lo cual es coherente con la mayor complejidad de este tipo de recorridos, caracterizados por la presencia de múltiples cambios bruscos de dirección. En contraste, el error obtenido en los tramos rectilíneos es mínimo, mientras que en las trayectorias curvas, aunque el error es mayor que en las rectas, se mantiene dentro de límites aceptables para un correcto seguimiento del recorrido.

Cabe destacar que los resultados presentados corresponden a un entorno de simulación, por lo que los valores de error son inferiores a los que cabría esperar en un sistema real, al no considerarse efectos como el ruido sensorial o las imprecisiones propias de los actuadores. No obstante, el análisis realizado permite evaluar de forma aislada el comportamiento del controlador y del sistema de seguimiento, centrándose en su desempeño intrínseco.

Aunque el error asociado a las trayectorias generadas mediante RRT es el más elevado de los casos analizados, su impacto en el movimiento del robot resulta limitado, ya que el sistema ha sido configurado para reducir la velocidad de avance en este tipo de trayectorias, favoreciendo así la estabilidad y el seguimiento en recorridos con elevada complejidad geométrica.

5.3 RRT

En este apartado se analizan los resultados obtenidos del algoritmo de planificación de trayectorias basado en RRT (Rapidly-exploring Random Tree). El objetivo principal es evaluar la capacidad del planificador para generar trayectorias viables en entornos con obstáculos, así como estudiar su comportamiento ante distintas configuraciones espaciales y niveles de complejidad.

A diferencia del apartado 5.2, centrado en el seguimiento de trayectorias ya definidas, el presente análisis se enfoca exclusivamente en la fase de planificación. Por tanto, se evalúa la capacidad del algoritmo para encontrar un camino libre de colisiones entre el punto inicial y el objetivo, sin considerar

aspectos relacionados con la ejecución de la trayectoria ni con el comportamiento dinámico del robot durante el movimiento.

El algoritmo RRT se ha empleado en los entornos descritos en el apartado 5.1, tanto en escenarios sin obstáculos como en entornos con diferentes grados de complejidad. En todos los casos, el planificador se ejecuta para determinar la trayectoria que debe seguir el robot. Cuando no existen obstáculos entre el origen y el destino, el algoritmo genera directamente una trayectoria rectilínea. En cambio, cuando es necesario evitar elementos del entorno, el RRT explora el espacio de estados mediante la construcción de un árbol de nodos, conectando progresivamente el estado inicial con la posición objetivo y evitando las regiones ocupadas.

Con el fin de analizar de manera sistemática el comportamiento del planificador, se han considerado distintas situaciones representativas: escenarios sin obstáculos, configuraciones con obstáculos aislados y entornos altamente restrictivos. Este enfoque permite evaluar de forma progresiva el rendimiento del algoritmo y observar cómo la complejidad del entorno influye en su capacidad para encontrar una solución válida.

En ausencia de obstáculos, el planificador genera una trayectoria rectilínea entre el punto inicial y el objetivo, como se muestra en la Figura 39. Este caso sirve como referencia para verificar el correcto funcionamiento del algoritmo en condiciones ideales.

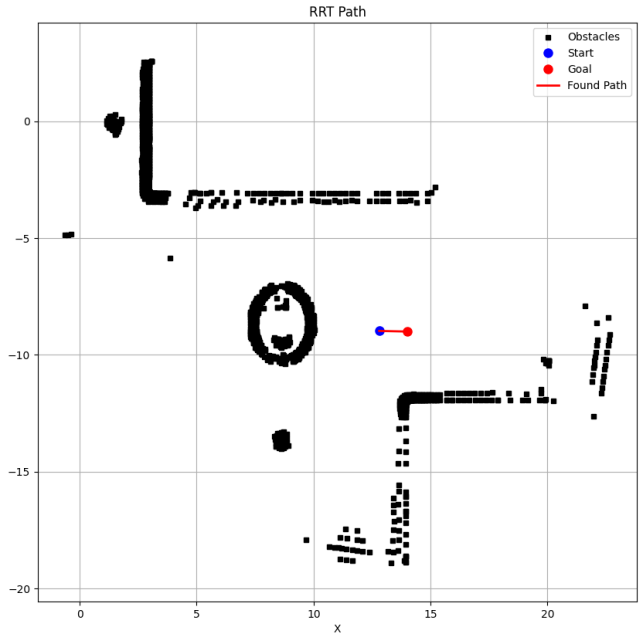

Figura 39. Resultado de la planificación en ausencia de obstáculos.

Cuando existen obstáculos en el entorno, el algoritmo RRT debe explorar el espacio para hallar una solución. En situaciones de complejidad baja o media, el planificador es capaz de generar trayectorias completas que conectan correctamente el origen con el destino. Estas trayectorias, aunque no necesariamente óptimas, resultan viables desde el punto de vista de la planificación y presentan múltiples cambios de dirección, característicos de los métodos basados en muestreo aleatorio. Las Figuras 40 y 41 ilustran ejemplos de planificación con obstáculos en los que se alcanza el objetivo con éxito.

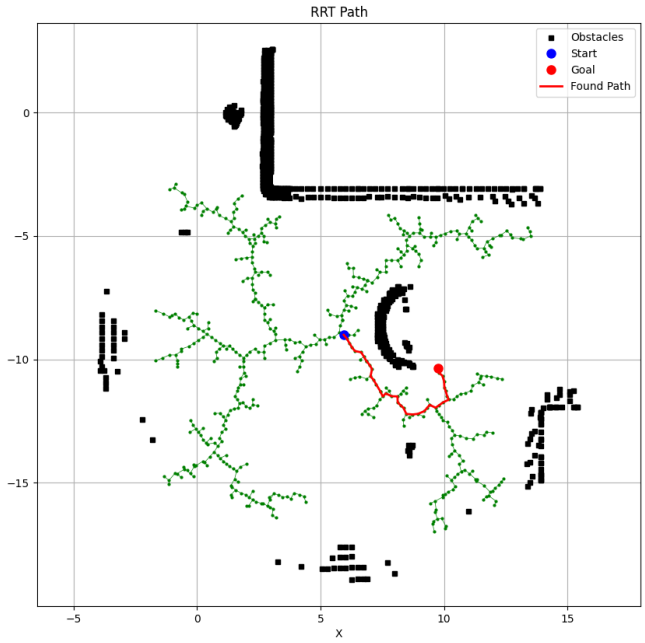

Figura 40. Expansión del árbol RRT y trayectoria generada ante la ausencia de visibilidad directa del objetivo.

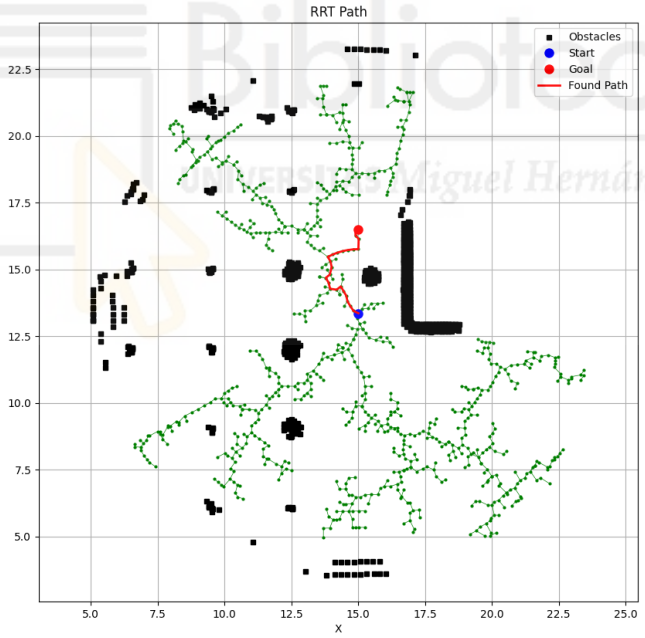

Figura 41. Planificación exitosa en un entorno con múltiples obstáculos estáticos.

Como se observa en las figuras, el planificador consigue encontrar un camino válido en ambos casos, aunque las situaciones representadas difieren. En la Figura 41 se muestra un escenario en el que algoritmo debe esquivar un obstáculo intermedio dentro de un entorno con múltiples obstáculos, resolviendo

correctamente la planificación. En cambio, en la Figura 40 el obstáculo bloquea la visibilidad directa del objetivo. En esta situación, el algoritmo selecciona un objetivo intermedio visible y lo más cercano posible al objetivo final, permitiendo avanzar progresivamente hacia la meta. En este caso concreto, el objetivo final se encontraba alineado con el robot en el eje X, en la posición aproximada (-9.5, 10).

Sin embargo, en escenarios con mayor densidad de obstáculos o con zonas de paso muy estrechas, se han observado situaciones en las que el algoritmo no logra encontrar un camino válido dentro de los parámetros establecidos. Este comportamiento se debe, en parte, a limitaciones inherentes del método RRT, cuya exploración se basa en muestreo aleatorio, así como a restricciones geométricas del entorno. Las Figuras 42 y 43 muestran ejemplos representativos de este tipo de situaciones.

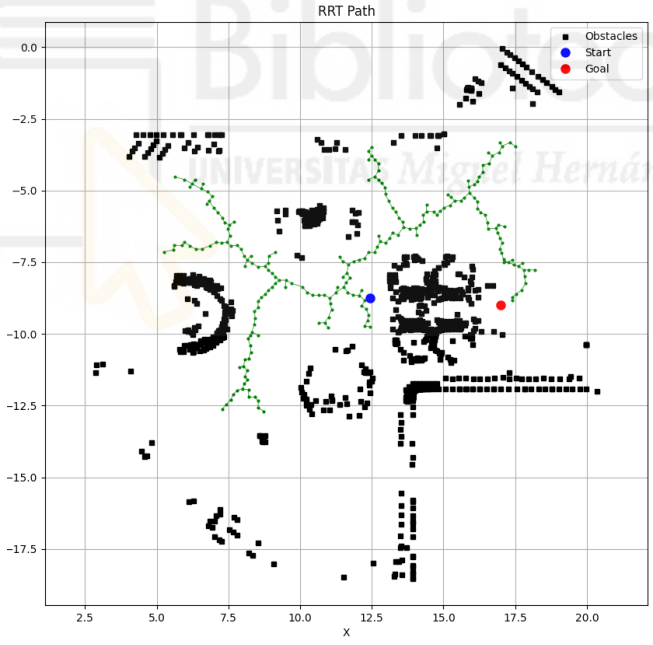

Figura 42. Planificación sin solución: objetivo situado dentro de una región no transitable.

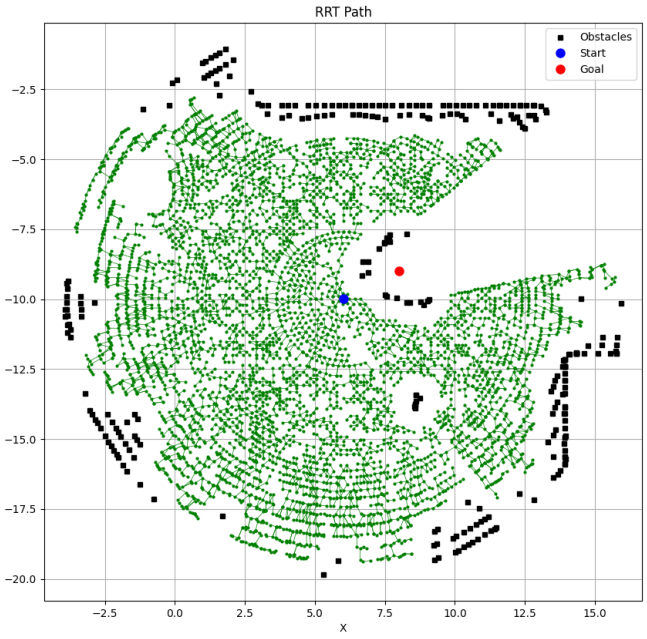

Figura 43. Planificación sin solución: objetivo situado en el interior de un obstáculo.

En estos casos, el objetivo se encuentra situado en el interior de un obstáculo. Sin embargo, dado que la representación del entorno considera únicamente los bordes de los obstáculos como puntos de colisión, el objetivo no es identificado explícitamente como no transitable. A pesar de ello, el algoritmo no logra establecer una conexión válida hasta el objetivo, ya que no existe un camino libre de colisiones que permita acceder a su posición desde el espacio navegable. Este comportamiento pone de manifiesto una limitación derivada del modelo de representación del entorno empleado en la planificación. En la Figura 42, por ejemplo, la trayectoria se queda a menos de un metro del objetivo sin llegar a completarse, ya que los puntos cercanos son considerados no válidos debido a la proximidad a obstáculos.

Asimismo, se han identificado situaciones en las que el planificador no encuentra solución porque el objetivo se encuentra demasiado próximo a un obstáculo o directamente dentro de una región no transitable. Tal como se muestra en la Figura 44, el sistema considera como no transitable cualquier punto situado a menos de 0,75 m de un obstáculo del entorno, teniendo en cuenta el tamaño del robot y un margen adicional de seguridad de 0,25 m. Esta restricción se introduce para evitar que el robot se acerque excesivamente a obstáculos y pueda quedar

atascado durante la ejecución. Además, como se ha observado en el apartado 5.2, el seguimiento de trayectorias generadas mediante RRT presenta errores del orden de unos centímetros, lo que podría provocar la invasión de zonas no transitables si no se aplicara dicho margen de seguridad.

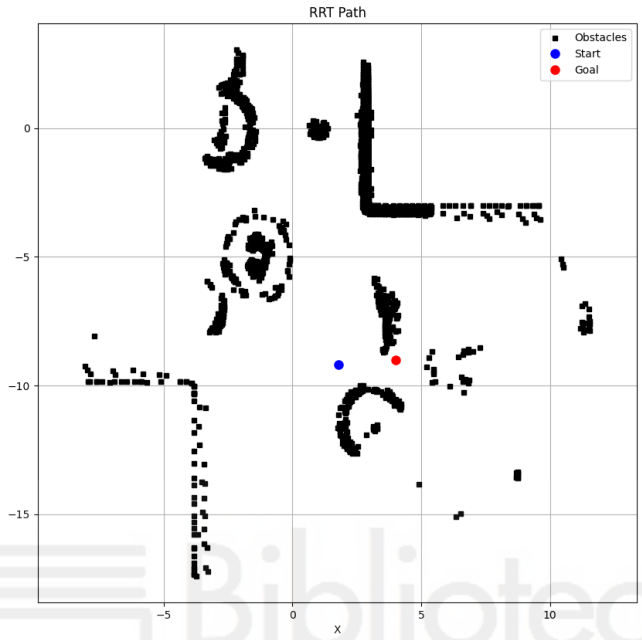

Figura 44. Planificación sin solución: objetivo situado en zona no transitable debido al margen de seguridad.

Por último, también se han observado casos en los que, aun encontrándose tanto el robot como el objetivo en zonas transitables, no se logra encontrar una trayectoria válida debido a la elevada densidad de obstáculos. En estas situaciones, como se ilustra en la Figura 45, no existe un paso viable, ni siquiera estrecho, que permita conectar el origen con el destino.

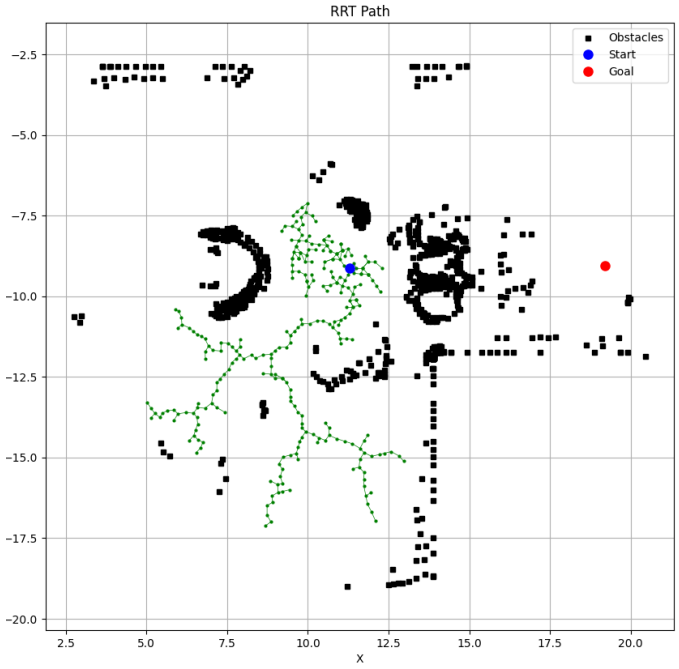

Figura 45. Planificación sin solución ante una densidad crítica de obstáculos sin paso viable.

La aparición de casos sin solución no se considera un fallo del sistema, sino un resultado relevante que permite identificar los límites del enfoque propuesto. Estos escenarios ponen de manifiesto la dificultad de la planificación en entornos altamente restrictivos y justifican la necesidad de analizar conjuntamente la planificación y el control para garantizar un comportamiento robusto del sistema global.

En conjunto, los resultados obtenidos indican que el algoritmo RRT es adecuado para la generación de trayectorias en entornos de complejidad baja y media, permitiendo al robot planificar recorridos viables en la mayoría de los casos analizados. No obstante, su rendimiento disminuye en escenarios extremadamente complejos, donde la probabilidad de encontrar una solución válida se reduce de forma significativa. Cabe destacar, además, que algunas de las situaciones sin solución detectadas en esta fase son parcialmente mitigadas en la etapa de control del sistema, mediante la lógica implementada en la función ‘move_to_target’ del archivo *Movement.py*.

5.4 PRUEBAS REALIZADAS

En este apartado se analiza el comportamiento global del sistema dentro del entorno de simulación. A diferencia de los apartados anteriores, centrados de forma separada en el seguimiento de trayectorias (apartado 5.2) y en la planificación (apartado 5.3), el presente análisis integra ambos procesos y evalúa el funcionamiento del sistema de navegación de manera conjunta.

El objetivo principal de este apartado es estudiar cómo el robot ejecuta en simulación las trayectorias generadas por el planificador, considerando aspectos relevantes como el tiempo necesario para alcanzar el objetivo, la aparición de colisiones, la evolución de la velocidad y la viabilidad para operar en tiempo real.

Con el objetivo de ilustrar el funcionamiento completo del sistema de navegación, en la Figura 46 se muestra de forma detallada el proceso desde la percepción del entorno hasta generación de la trayectoria final.

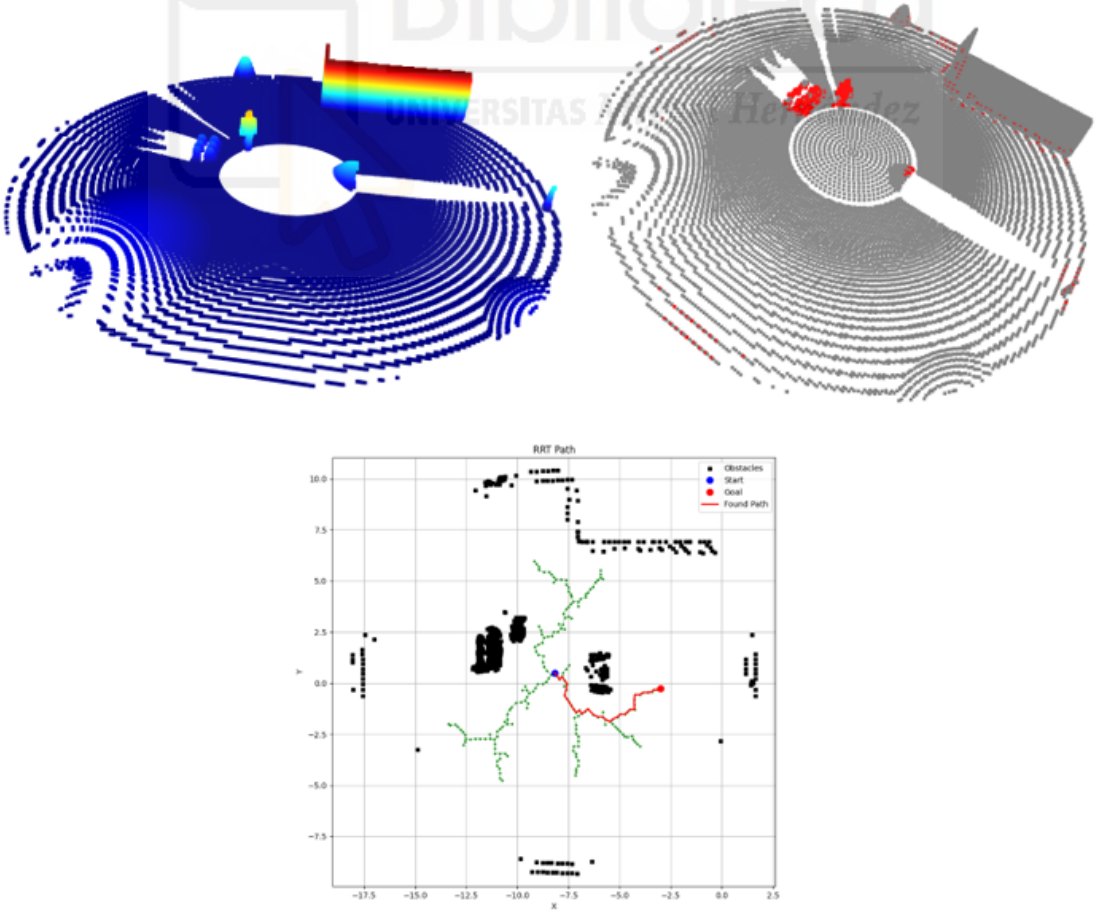

Figura 46. Ejemplo del funcionamiento del algoritmo RRT paso a paso.

En la figura anterior se visualizan las distintas etapas del sistema desarrollado, incluyendo la adquisición de la nube de puntos mediante el sensor LiDAR, la detección de zonas no transitables a través del cálculo de diferencia de normales (DoN) y la planificación del camino mediante el algoritmo RRT. En la parte superior a la izquierda se muestra el primer paso del proceso, correspondiente a la captura de la nube de puntos por el LiDAR, donde pueden apreciarse distintos obstáculos cercanos, como un edificio, una persona y varios baches. El círculo interior vacío se corresponde con la posición del robot, así como con aquellas zonas del suelo que no pueden ser detectadas debido a las limitaciones del campo de visión del sensor, tal como se ha descrito anteriormente.

A partir de esta información, el cálculo de DoN permite identificar los puntos del entorno considerados no transitables para el robot, como se puede apreciar en la parte superior derecha de la figura. Por último, en la parte inferior se muestra el resultado del planificador RRT, junto con la ruta generada. Cabe destacar que los obstáculos representados en esta última imagen no coinciden con los detectados en la etapa de diferencias de normales, ya que no se trata de la primera adquisición del LiDAR durante la simulación. El sistema almacena la información de los puntos identificados anteriormente como obstáculos, construyendo de forma progresiva el mapa del entorno.

Una vez descrito el funcionamiento global del sistema, se ha evaluado su comportamiento sobre el robot Husky en el simulador CoppeliaSim, utilizando los entornos definidos en el apartado 5.1, abarcando situaciones con baja, media y alta complejidad. En el entorno 1, correspondiente a un escenario de baja complejidad, el robot es capaz de completar el recorrido sin colisiones, manteniendo un comportamiento estable durante toda la ejecución. Las Figuras 47 y 48 muestran capturas representativas de este entorno, en las que se visualiza simultáneamente el mapa con la trayectoria planificada y la posición del robot durante la ejecución, lo que permite relacionar visualmente la planificación con el comportamiento dinámico del sistema.

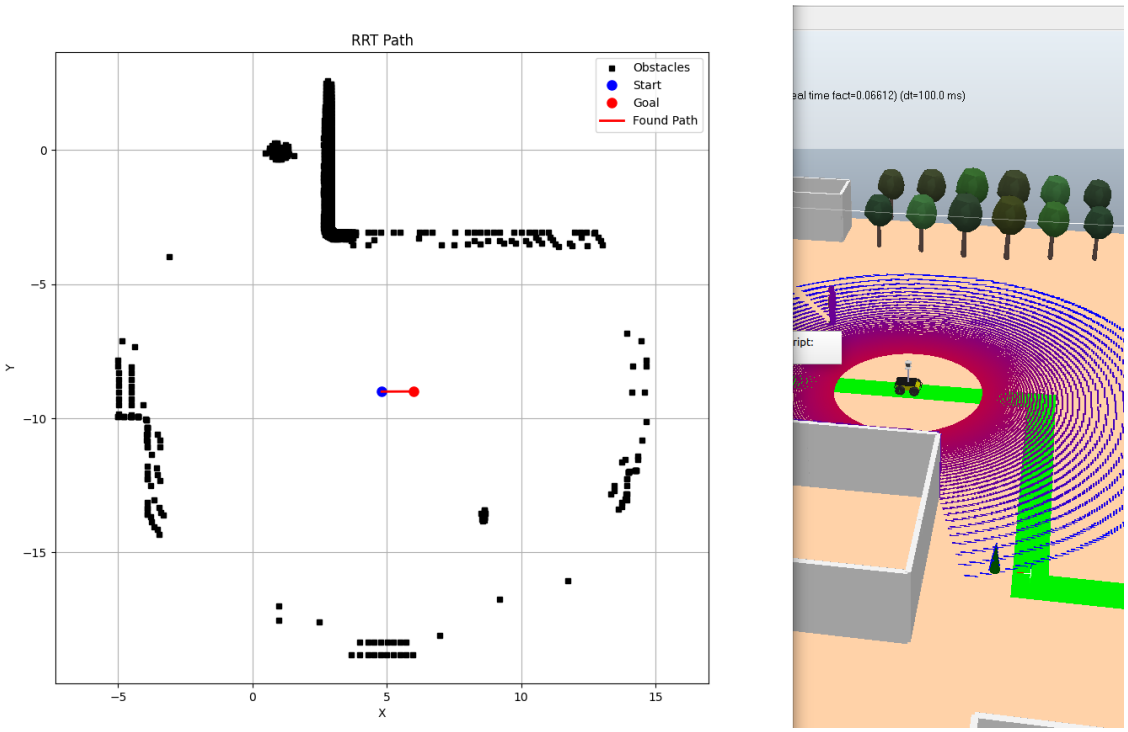

Figura 47. Ejecución del sistema con planificación rectilínea en el segundo tramo del entorno 1.

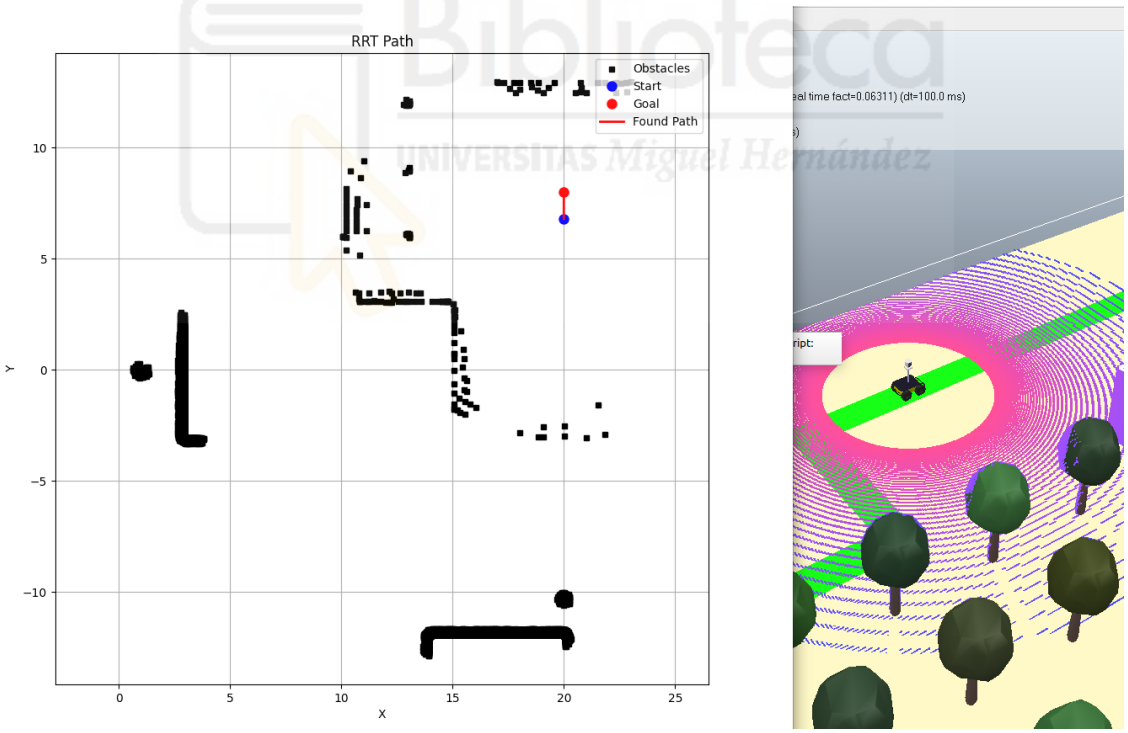

Figura 48. Ejecución del sistema con planificación rectilínea en el tercer tramo del entorno 1.

De la observación de las figuras se desprende que en este primer entorno todas las trayectorias planificadas son segmentos rectilíneos, ya que no existen

obstáculos que interfieran en el camino marcado en verde que debe seguir el robot para completar el circuito.

En el entorno 2, de complejidad media, donde aparecen obstáculos tanto estáticos como dinámicos, se observa una clara adaptación del comportamiento del robot. En este escenario, el sistema debe generar trayectorias más complejas para esquivar los obstáculos presentes, por lo que no todas las trayectorias son rectilíneas, a diferencia de lo observado en el entorno 1. A pesar de ello, el robot consigue completar el circuito sin colisiones, demostrando la capacidad del sistema para operar en entornos más realistas. Las Figuras 49 y 50 muestran capturas representativas de la ejecución en este entorno.

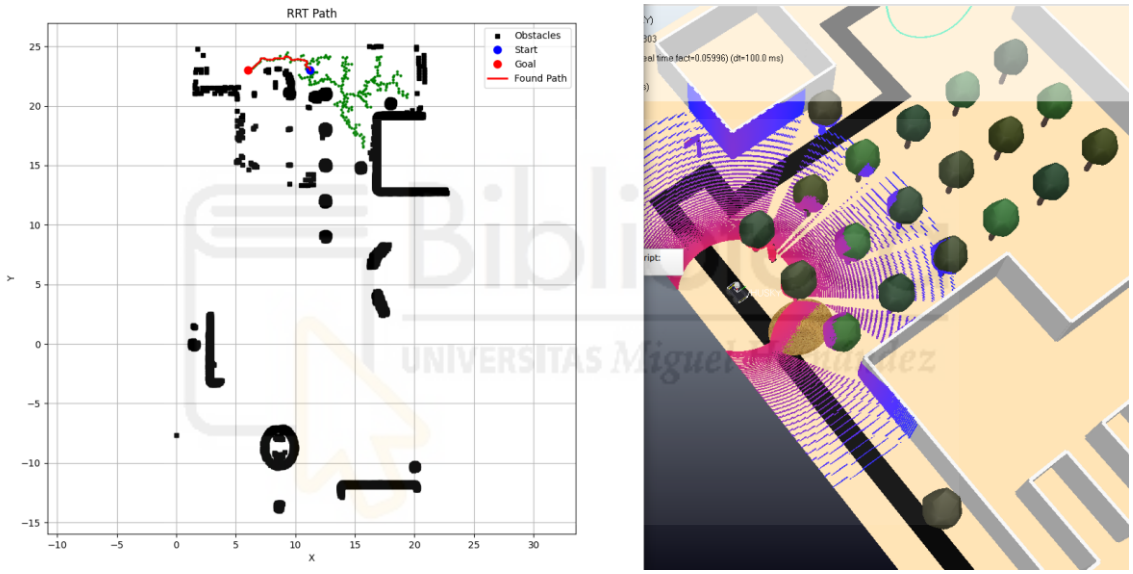

Figura 49. Ejecución del sistema con replanificación local mediante RRT para la evasión de un bache en el entorno 2.

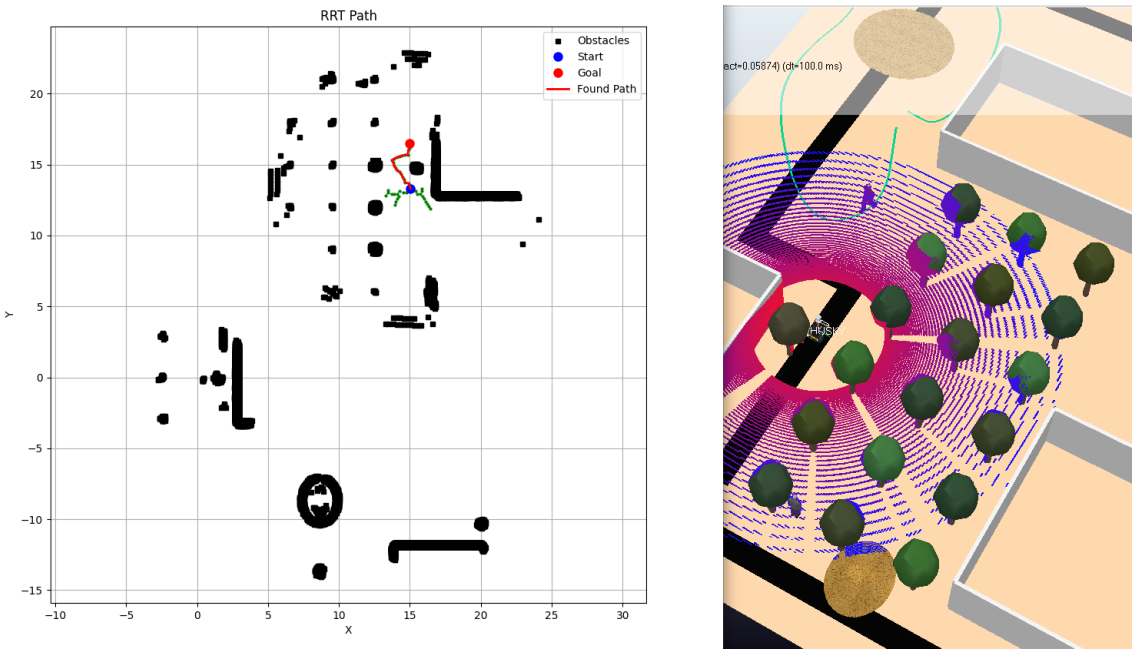

Figura 50. Ejecución del sistema con replanificación local mediante RRT para la evasión de un árbol en el entorno 2.

Como se aprecia en las figuras anteriores, el sistema consigue encontrar trayectorias viables que permiten continuar el recorrido. En particular, en la Figura 51 se muestra la planificación correspondiente a una situación cercana a la finalización del circuito en el entorno 2.

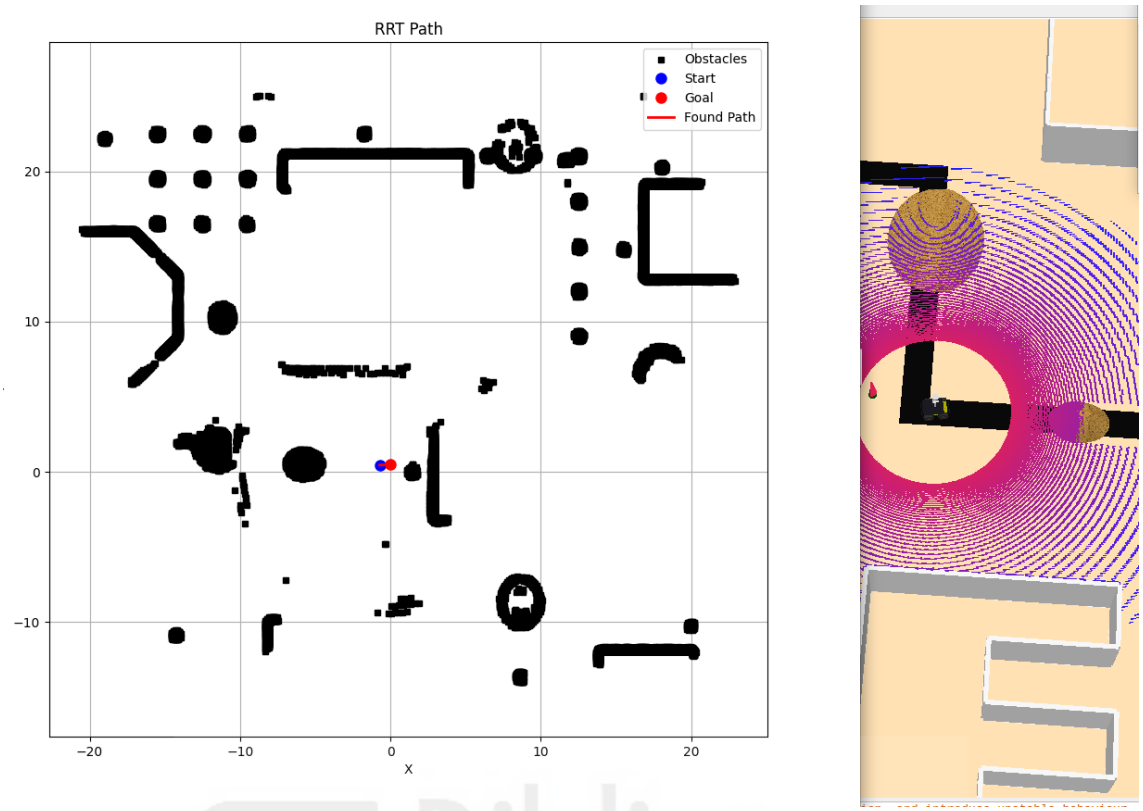

Figura 51. Mapa de ocupación global y planificación final durante la ejecución en el entorno 2.

En dicha figura se observa el mapa con los puntos identificados por el sistema como no transitables. Las líneas representan las paredes de los edificios presentes en el entorno, mientras que los círculos pequeños corresponden a árboles, que suelen aparecer agrupados, y a las banderas utilizadas como referencias visuales del recorrido. Asimismo, las demás zonas no transitables pertenecen a los baches y a las personas. Todas las áreas por las que se desplazan las personas en movimiento son detectadas como obstáculo, ya que el sistema almacena la información de sus posiciones a lo largo del tiempo.

Por el contrario, en el entorno 3, caracterizado por una configuración altamente restrictiva, se han identificado situaciones en las que el robot no consigue pasar, por lo que no consigue completar el circuito. Estos casos pueden deberse a la imposibilidad de encontrar una trayectoria válida durante la fase de planificación o a que el robot se aproxima en exceso a zonas no transitables, lo que puede provocar atascos o incluso colisiones. En estas situaciones, el sistema replanifica para evitar impactos, pero no siempre logra alcanzar el objetivo. En

las Figuras 52 y 53 se muestran ejemplos representativos de este tipo de comportamientos.

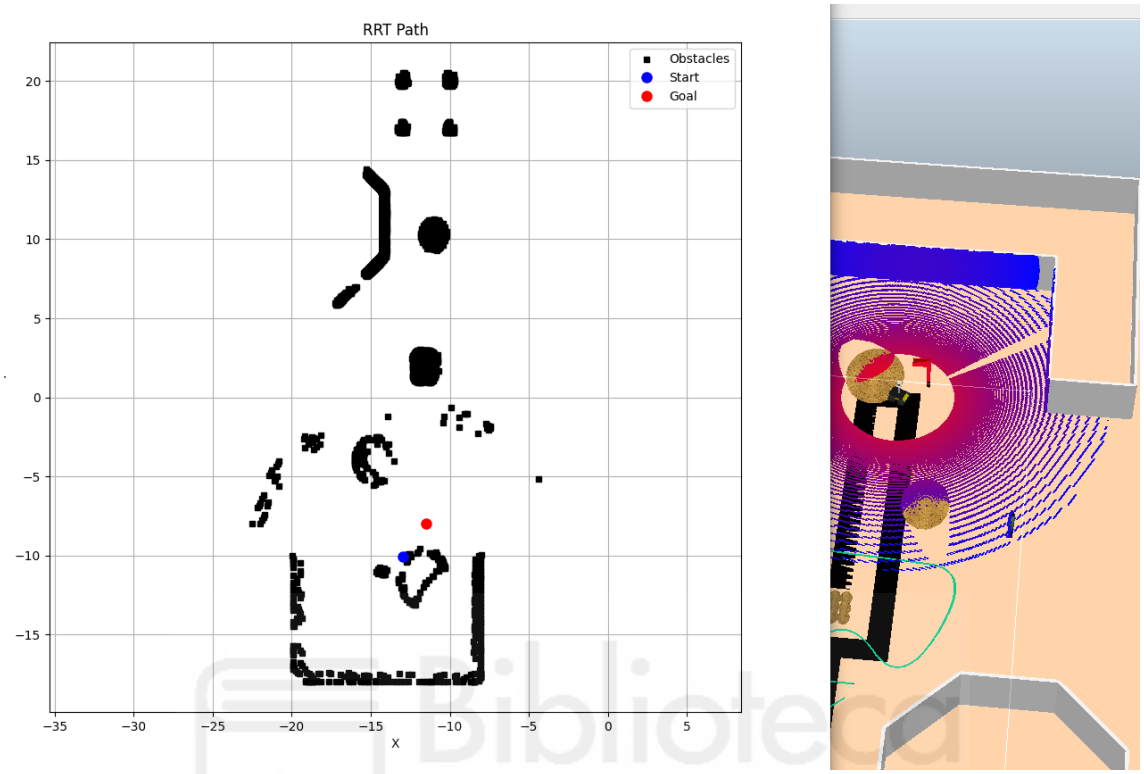

Figura 52. Ejemplo de bloqueo del robot durante la ejecución en el entorno 3.

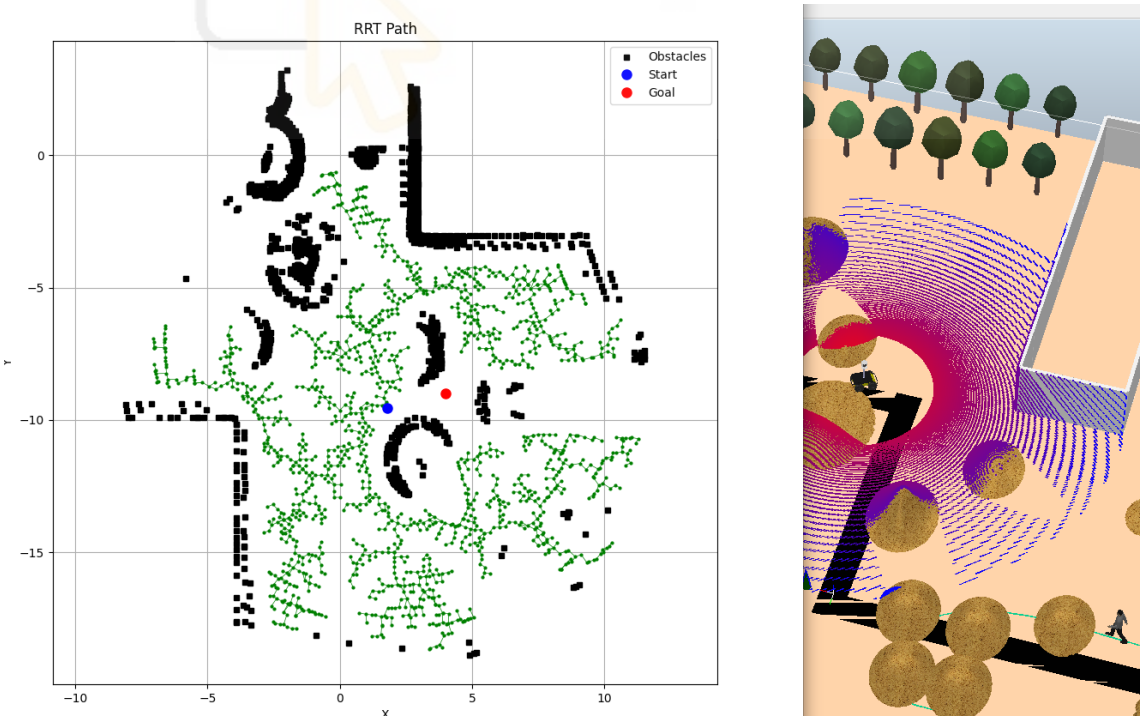

Figura 53. Ejecución del sistema sin trayectoria válida durante la planificación en el entorno 3.

En la Figura 52 se observa cómo el robot no logra esquivar un obstáculo y queda atascado en una zona concreta del entorno. Por otro lado, en la Figura 53 el robot se encuentra en una zona transitable, al igual que el objetivo, pero no consigue hallar una trayectoria libre de colisiones que permita continuar el recorrido. La Figura 54 representa la misma situación tras un nuevo cálculo del objetivo, en el que inicialmente se obtiene una solución; sin embargo, poco después el robot vuelve a quedarse bloqueado debido a la elevada densidad de baches presentes en la zona.

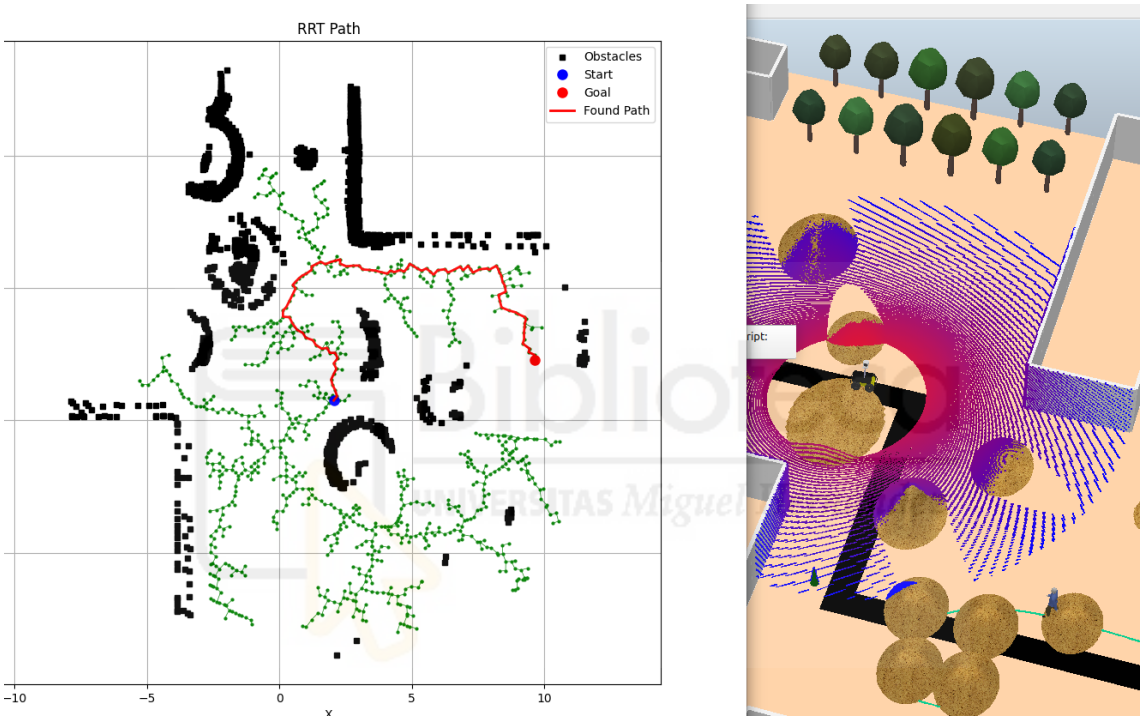

Figura 54. Ejecución del sistema en el entorno 3: planificación RRT y trayectoria generada en una zona altamente restrictiva.

Como se desprende de los resultados obtenidos, el sistema funciona correctamente en simulación en los entornos de complejidad baja y media, logrando completar los recorridos sin colisiones. En cambio, no ha sido posible completar el circuito del entorno 3, debido a la aparición de múltiples situaciones de bloqueo y a la colisión puntual con un obstáculo móvil, concretamente una persona.

Una vez comprobado que el sistema es capaz de completar el recorrido en los entornos de complejidad baja y media, se ha realizado un ensayo orientado a

analizar la influencia del número máximo de iteraciones del algoritmo RRT en el tiempo necesario para completar el circuito. Este análisis se ha llevado a cabo exclusivamente en el entorno 2, puesto que se trata del escenario de mayor complejidad en el que el sistema consigue completar el recorrido de forma consistente. En la Tabla III se recogen los tiempos necesarios para completar el circuito en función del número máximo de iteraciones permitido al algoritmo RRT.

Número de iteraciones	Circuito completado	Tiempo de ejecución
500	No	—
1000	Sí	1:20:17
3000	Sí	1:18:19
5000	Sí	1:17:2
10000	Sí	1:16:11

Tabla III. Tiempo necesario para completar el circuito en el entorno 2 en función del número de iteraciones del RRT.

Para la obtención de los resultados mostrados en la tabla anterior, cada configuración se ha ejecutado en dos ocasiones, calculándose posteriormente el valor medio. Esta medida se ha adoptado debido al carácter aleatorio del algoritmo RRT, que puede generar trayectorias distintas en cada ejecución, afectando tanto a la planificación como al tiempo total del recorrido.

Como puede observarse en la tabla, a medida que aumenta el número de iteraciones, el sistema logra completar el circuito con trayectorias progresivamente más directas, lo que se traduce en una ligera reducción del tiempo total de ejecución. Con un número reducido de iteraciones, el planificador puede no encontrar una trayectoria válida en determinadas situaciones, lo que obliga al sistema a avanzar hacia el siguiente objetivo intermedio (waypoint) o, en algunos casos, impide la finalización del recorrido.

No obstante, el incremento del número de iteraciones presenta rendimientos decrecientes: a partir de cierto umbral, el aumento en el número de muestras no conlleva mejoras significativas en el comportamiento global del sistema.

Además, en este caso, el tiempo empleado en el cálculo del RRT, incluso para valores elevados de iteraciones, no tiene un impacto apreciable en el tiempo total de ejecución en el entorno de simulación, ya que el factor limitante principal en este caso es la velocidad del simulador.

En un sistema real, sin embargo, un número elevado de iteraciones podría afectar de forma más significativa a la capacidad de operación en tiempo real. Por este motivo, en el resto de pruebas realizadas se ha fijado el número máximo de iteraciones en 3000, valor que ofrece un compromiso adecuado entre la robustez de la planificación y el coste computacional. Aumentar este parámetro no permite completar el circuito en el entorno 3, ni produce mejoras relevantes en los resultados obtenidos, mientras que una reducción del mismo incrementa la probabilidad de que el sistema no logre generar una trayectoria válida en determinadas situaciones.

Este análisis pone de manifiesto la necesidad de ajustar adecuadamente el número de iteraciones del RRT, de modo que se garantice un compromiso adecuado entre la robustez de la planificación y la viabilidad del sistema para operar en tiempo real.

Asimismo, el comportamiento del sistema es coherente con las velocidades típicas de robots móviles reales, que suelen operar en un rango aproximado de 0,3 a 1 m/s. En la simulación, el robot se desplaza a 1 m/s en zonas sin obstáculos, reduciendo su velocidad hasta 0,25 m/s en tramos con elevada curvatura o durante la evasión de obstáculos. Además, el sistema reduce su velocidad antes de alcanzar una curva, lo que permite realizar el giro de forma más estable. Estas velocidades se encuentran dentro de los rangos habituales en aplicaciones reales. Por otro lado, el sistema procesa información del LiDAR cada 0,1 s, lo que equivale a una frecuencia de 10 Hz, también acorde con sensores reales.

Para garantizar un funcionamiento correcto del sistema, es fundamental ajustar adecuadamente determinados parámetros. En primer lugar, el tamaño de los radios y los umbrales empleados en el cálculo de diferencias de normales, ya que de ellos depende la correcta identificación de zonas transitables. Asimismo,

el umbral utilizado en el algoritmo RRT para determinar la viabilidad de una conexión entre nodos resulta crítico: valores excesivamente restrictivos pueden impedir encontrar soluciones válidas, mientras que umbrales demasiado permisivos pueden generar trayectorias no seguras. Finalmente, el control de la velocidad resulta esencial para evitar giros excesivamente lentos o rápidos y para garantizar una deceleración adecuada antes de curvas pronunciadas.

Finalmente, en la Tabla IV se recogen los tiempos necesarios para completar el circuito en los entornos 1 y 2. En dicha tabla se incluyen el tiempo de ejecución, el tiempo de simulación y el factor de tiempo real, definido como la relación entre el tiempo simulado y el tiempo real de cómputo.

Entorno	Tiempo de ejecución	Tiempo de simulación	Factor de tiempo real
Entorno 1	57:32	4:14	0,0736
Entorno 2	1:18:17	6:04	0,0775

Tabla IV. Tiempos de ejecución en simulación y factor de tiempo real para los entornos 1 y 2.

Los datos presentados en la tabla muestran que el factor de tiempo real es muy similar en ambos entornos, lo que indica que la complejidad de la planificación no afecta de forma significativa al rendimiento temporal del sistema. El bajo factor de tiempo real se debe principalmente al uso de pasos de simulación de 0,1 s, necesarios para obtener información del LiDAR con la frecuencia deseada, así como a la velocidad de ejecución del simulador en el equipo utilizado. A pesar de ello, los resultados indican que el sistema podría operar en tiempo real en un entorno real.

En términos de viabilidad en tiempo real, los resultados obtenidos en simulación muestran que el sistema es capaz de operar dentro de los márgenes temporales necesarios para un robot móvil real, permitiendo una navegación reactiva, incluso en presencia de obstáculos.

En conjunto, el análisis realizado confirma que el sistema desarrollado presenta un comportamiento robusto y coherente en simulación, siendo capaz de planificar y ejecutar trayectorias de manera segura en entornos de complejidad

baja y media. Las limitaciones observadas en escenarios extremadamente complejos no se consideran fallos del sistema, sino una consecuencia esperable de las restricciones del entorno y del método de planificación empleado, proporcionando información valiosa sobre los límites operativos del enfoque propuesto.

6. CONCLUSIONES

En este estudio se ha abordado el problema de la navegación autónoma de un robot móvil en entornos semiestructurados mediante el uso de un sensor LiDAR y algoritmos de planificación de trayectorias. A lo largo del desarrollo del proyecto, se han diseñado, implementado y validado los distintos módulos de percepción, planificación y control, con el objetivo de evaluar la viabilidad de la solución propuesta.

En relación con los objetivos planteados, puede afirmarse que se han alcanzado de manera satisfactoria. El sistema desarrollado ha demostrado ser capaz de seguir trayectorias predefinidas en ausencia de obstáculos, detectar obstáculos de forma autónoma a partir de datos de LiDAR y replanificar la trayectoria cuando la ruta se ve comprometida. Asimismo, se ha verificado el correcto funcionamiento conjunto de todos los módulos, permitiendo que el robot complete el recorrido en distintos escenarios con diferentes niveles de complejidad. Desde el punto de vista de los objetivos de aprendizaje, el proyecto ha permitido consolidar conocimientos en robótica móvil, simulación, programación en Python y planificación de trayectorias, cumpliendo así con las competencias previstas para este trabajo.

En cuanto a la validez y aplicabilidad de la solución, la aproximación empleada resulta especialmente adecuada para entornos semiestructurados y estáticos o cuasiestáticos, donde los obstáculos no cambian de posición de forma brusca y el robot dispone de tiempo suficiente para procesar la información sensorial y replanificar la trayectoria. El uso del sensor LiDAR combinado con la técnica de diferencia de normales ha permitido una detección robusta de obstáculos, diferenciando de forma eficaz zonas transitables y no transitables. Por su parte, el algoritmo RRT ha demostrado ser una herramienta flexible para la planificación de trayectorias en mapas complejos, siendo capaz de encontrar rutas viables en presencia de obstáculos de distinta densidad. No obstante, los resultados también evidencian las limitaciones del sistema en escenarios extremadamente congestionados o con pasajes muy estrechos, donde la planificación puede no encontrar una solución válida.

Finalmente, en lo relativo a mejoras y trabajos futuros, se identifican varias líneas de ampliación del proyecto. Una mejora inmediata sería la extensión del sistema a entornos dinámicos. Asimismo, podría evaluarse el uso de variantes más avanzadas del algoritmo RRT, como el RRT* o RRT-Connect, con el objetivo de obtener trayectorias óptimas y suaves. Otra posible línea de trabajo consiste en la fusión sensorial, integrando el LiDAR con cámaras para aumentar la robustez del sistema en condiciones adversas y mejorar la detección de obstáculos.

Como trabajo futuro, podría incorporarse un sistema SLAM, que permita acumular la información del entorno a lo largo del tiempo y reducir la dependencia de información local y facilitar la navegación en entornos no conocidos. Además, una evolución natural del proyecto sería la transferencia del sistema desde el entorno de simulación a un robot real, lo que permitiría analizar el impacto de factores como el ruido sensorial o la incertidumbre en la odometría. Finalmente, en escenarios altamente restrictivos, como los analizados en el entorno de máxima dificultad, podría estudiarse la combinación del RRT con planificadores globales basados en grafos o técnicas híbridas, con el objetivo de aumentar la probabilidad de encontrar soluciones en pasajes estrechos.

En conclusión, este trabajo demuestra que la combinación de percepción basada en LiDAR y planificación de trayectorias mediante RRT constituye una solución viable para la navegación autónoma de robots móviles en entornos semiestructurados, sentando una base sólida para futuras investigaciones y desarrollos en el ámbito de la robótica móvil autónoma.

7. BIBLIOGRAFÍA

[1] Leotronics “Robots de servicios de transporte y logística: entornos interiores con tráfico público” apartado “transporte en hospitales y centros asistenciales, En: *leotronics.eu*. URL: https://leotronics.eu/es/nuestro-blog/robots-de-servicios-de-transporte-y-logistica-entornos-interiores-con-trafico-publico.

[2] Esnova “Robots AMR, claves en la nueva logística industrial”, En: *Esnova.com*. URL: https://esnova.com/es/blog/robots-autonomos-amr.

[3] Isaac Oyeyemi Olayode et al. “Systematic literature review on the applications, impacts, and public perceptions of autonomous vehicles in road transportation system”, En: *Journal of Traffic and Transportation Engineering (English Edition)*. DOI: https://doi.org/10.1016/j.jtte.2023.07.006.

[4] Lixing Liu et al. “Path planning techniques for mobile robots: Review and prospect”, En: *Expert Systems with Applications.* DOI: https://doi.org/10.1016/j.eswa.2023.120254.

[5] Christina Soyoung Song y Young-Kyung Kim “The role of the human-robot interaction in consumers’ acceptance of humanoid retail service robots”, En: *Journal of Business Research.* DOI: https://doi.org/10.1016/j.jbusres.2022.03.087.

[6] Marco Tranzatto et al. “CERBERUS in the DARPA Subterranean Challenge”, En: *Revista Science Robotics*. DOI: https://doi.org/10.1126/scirobotics.abp9742.

[7] Janusz Będkowski y Michał Pełka “Affordable Robotic Mobile Mapping System Based on Lidar with Additional Rotating Planar Reflector”, En: *MDPI*. DOI: https://doi.org/10.3390/s23031551.

[8] Hao Qiu et al. “Research on unmanned vehicle obstacle avoidance technology based on LIDAR and depth camera fusion”, En: *Applied Mathematics and Nonlinear Sciences*. DOI: https://doi.org/10.2478/amns.2023.2.00575.

[9] Ze Zhang et al. “Gradient Field-Based Dynamic Window Approach for Collision Avoidance in Complex Environments”, En: *arXiv*. DOI: https://doi.org/10.48550/arXiv.2504.03260.

[10] Jordy Sehn, Timothy D. Barfoot y Jack Collier “Off the Beaten Track: Laterally Weighted Motion Planning for Local Obstacle Avoidance”, En: *IEEE Transactions on Field Robotics (Volume 1)*. DOI: https://doi.org/10.1109/TFR.2024.3492151.

[11] Yanni Ioannou et al. “Difference of Normals as Multi-scale Operator in Unorganized Point Clouds”, En: *2012 Second International Conference on 3D Imaging, Modeling, Processing, Visualization & Transmission*. DOI: https://doi.org/10.1109/3DIMPVT.2012.12.

[12] Edison Velasco-Sánchez “Estimación de pose y control de Robots Móviles en Entornos no Estructurados basado en LiDAR”, En: *Universidad de Alicante.* URL: https://rua.ua.es/entities/publication/9fa26a2d-b7d1-44d9-b8a9-3262df294ebd

[13] Albert Palomer et al. “Bathymetry-based SLAM with difference of normals point-cloud subsampling and probabilistic ICP registration”, En: *2013 MTS/IEEE OCEANS – Bergen*. DOI: https://doi.org/10.1109/OCEANS-Bergen.2013.6608091

[14] E. W. Dijkstra “A Note on Two Problems in Connexion with Graphs”, En: *Numerische Mathematik volumen 1 páginas 269-271*. DOI: https://doi.org/10.1007/BF01386390

[15] Peter E. Hart et al. “A Formal Basis for the Heuristic Determination of Minimum Cost Paths”, En: *IEEE Transactions on Systems Science and Cybernetics*. DOI: https://doi.org/10.1109/TSSC.1968.300136

[16] J. J. Kuffner and S. M. LaValle “RRT-connect: An efficient approach to single-query path planning” En: *Millennium Conference. IEEE International Conference on Robotics and Automation. Symposia Proceedings páginas 995-1001*. DOI: https://doi.org/10.1109/ROBOT.2000.844730

[17] Documentación MuJoCo, MuJoCo, [Online]. Disponible en: https://mujoco.readthedocs.io/en/stable/overview.html

[18] Guía de uso de Webots, Cyberbotics Ltd., [Online]. Disponible en: https://cyberbotics.com/doc/guide/index

[19] Documentación simulador MORSE, Openrobots, [Online]. Disponible en: https://www.openrobots.org/morse/doc/latest/morse.html

[20] Repositorio de Unity Robotics, Unity Technologies, Repositorio en GitHub, [Online]. Disponible en: https://github.com/Unity-Technologies/Unity-Robotics-Hub

[21] Guía básica de Gazebo, Gazebo, [Online]. Disponible en: https://gazebosim.org/docs/latest/getstarted

[22] Manual uso de Coppelia, Coppelia Robotics, [Online]. Disponible en: https://manual.coppeliarobotics.com

[23] A. Gil, pyARTE [Software], Universidad Miguel Hernández, Repositorio en GitHub. Disponible en: https://github.com/4rtur1t0/pyARTE