UNIVERSIDAD PENÍNSULA DE SANTA ELENA

FACSITEL

INGENIERÍA EN ELECTRÓNICA Y AUTOMATIZACIÓN

TRABAJO DE INTEGRACIÓN CURRICULAR

"DESARROLLO DE UN SISTEMA DE CONTROL PARA PLANIFICACIÓN DE TRAYECTORIA Y GENERACIÓN DE ENTORNO SOBRE UN ROBOT MÓVIL EN PROCESOS DE EXPLORACIÓN Y SUPERVISIÓN"

Bryan Eduardo Alarcon Bazurto

Dirigido por:
Ing. Mario Alomoto Tomalá, Mgtr.

La Libertad - 2025

DEDICATORIA

Este trabajo de titulación se lo dedico con toda gratitud y amor a Dios, por darme la fortaleza para perseverar y la sabiduría para guiar mi mente y corazón, impidiéndome rendirme hasta culminar con éxito este sueño profesional.

A mis padres, Santo Alarcon y Angélica Bazurto por ser los pilares fundamentales en mi vida, por el amor incondicional, los sacrificios incalculables y el apoyo inquebrantable que me ofrecieron en cada etapa. Su ejemplo de esfuerzo, su fe en mí y sus palabras de aliento fueron la motivación constante para alcanzar esta meta.

A mis hermanos, por su apoyo moral, por darme consejos y estar siempre presente en todo momento.

A mi abuela, Cruz Reyes aunque ya no está a mi lado, sé que desde el cielo me acompañó en este largo camino y se siente muy orgullosa de todo lo que he logrado.

A mi abuelo, Fulton Bazurto quien me brindó su apoyo incondicional desde los inicios de esta travesía académica y continúa siendo, hoy más que nunca, un pilar fundamental en mi vida.

A mi amada novia, Julissa Orrala por estar presente y ser parte de este proceso, por brindarme su amor y palabras de aliento cuando más lo necesitaba, por jamás hacer que me diera por vencido. Te amo.

A mis amistades, Roger, Fanny y Leslie por brindarme desde el principio su compañía en los momentos claves y hacer que esté proceso académico sea una experiencia inolvidable y llevadera.

Bryan Eduardo Alarcon Bazurto

AGRADECIMIENTO

La culminación de este trabajo de titulación representa no solo un logro académico, sino el resultado del apoyo, la guía y la colaboración de muchas personas e institución a las que deseo expresar mi más sincero agradecimiento.

A mi tutor, Ing. Mario Alomoto, Mgtr., y especialista Ing. Junior Figueroa Mgtr. por su invaluable paciencia, su rigor académico y su dedicación. Sus conocimientos y constante motivación fueron cruciales para la culminación de este trabajo de titulación.

A mis amados padres, Santo Alarcon y Angélica Bazurto, por ser mi inspiración y el soporte emocional y económico en todo momento. Su amor y apoyo incondicional fue el motor que me impulsó a no desistir.

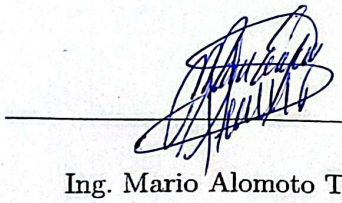
A mis hermanos, novia Ing, Julissa Orrala y familiares cercanos, por su comprensión, sus palabras de aliento y por compartir las exigencias de este proceso. A mis colegas, Ing. Roger Carbo y Andrés Pinargote por brindarme su apoyo y conocimientos en los últimos momentos de la culminación de este trabajo, sin su ayuda no podría ser posible.

Finalmente, a la Universidad Estatal Península de Santa Elena , por brindarme el espacio y la formación integral necesaria para mi desarrollo profesional.

Bryan Eduardo Alarcon Bazurto

APROBACIÓN DEL TUTOR

En mi calidad de tutor del trabajo de titulación denominado: **"Desarrollo de un sistema de control para planificación de trayectoria y generación de entorno sobre un robot móvil en procesos de exploración y supervisión**, elaborado por el estudiantes: **Bryan Eduardo Alarcon Bazurto**, de la carrera de **Ingeniería en Electrónica y Automatización** de la Universidad Estatal Península de Santa Elena, declaro que luego de haber orientado, estudiado y revisado, lo apruebo en todas sus partes y autorizo al estudiante que inicie los trámites legales correspondientes.



Ing. Mario Alomoto Tomalá, Mgtr.
Tutor

TRIBUNAL DE GRADO

Ing. Ronald Humberto Rovira Jurado, Ph.D.
DIRECTOR DE CARRERA

Ing. Luis Enrique Chuquimarca Jiménez, Mgtr.
DOCENTE GUÍA UIC

Ing. Junior Figueroa Olmedo, Mgtr.
DOCENTE ESPECIALISTA

Ing. Mario Alomoto Tomalá, Mgtr.
TUTOR

Ing. Corina Gonzabay De La A, Mgtr.
SECRETARIA DEL TRIBUNAL

Resumen

Este proyecto desarrolló un sistema integral de navegación autónoma para el robot móvil Transbot-SE, implementando algoritmos de SLAM y planificación de trayectorias. La investigación logró crear una plataforma robótica capaz de realizar exploración autónoma en entornos cerrados mediante la integración de sensores LIDAR y cámara PTZ, procesando datos en tiempo real para generar mapas del entorno mientras ejecuta rutas predefinidas. El sistema implementa una plataforma dual que opera en modo manual y automático, permitiendo tanto el control directo por parte del operador como la navegación autónoma completa. El núcleo del desarrollo incluye algoritmos de procesamiento de nubes de puntos LIDAR, técnicas de corrección de deriva mediante cierres de ciclo, y mecanismos de planificación de trayectorias con evasión de obstáculos. Las pruebas experimentales demostraron un desempeño satisfactorio con precisión de mapeo aceptable en modo automático, validando la efectividad de la solución implementada para aplicaciones de exploración y supervisión en entornos controlados.

Palabras clave: Robótica móvil, SLAM, navegación autónoma, LIDAR, planificación de trayectorias, procesamiento de nubes de puntos.

Abstract

This project developed a comprehensive autonomous navigation system for the Transbot-SE mobile robot, implementing Simultaneous Localization and Mapping (SLAM) algorithms and trajectory planning techniques. The research successfully created a robotic platform capable of autonomous exploration in indoor environments through the integration of LIDAR sensors and PTZ camera, processing real-time data to generate environmental maps while executing predefined routes.

The system features a dual-mode architecture operating in both manual and automatic configurations, enabling either direct operator control or complete autonomous navigation. The core development includes LIDAR point cloud processing algorithms, drift correction techniques through loop closure mechanisms, and trajectory planning systems with obstacle avoidance capabilities. Experimental testing demonstrated satisfactory performance with acceptable mapping accuracy in automatic mode, validating the effectiveness of the implemented solution for exploration and monitoring applications in controlled environments.

Keywords: Mobile robotics, SLAM, autonomous navigation, LIDAR, trajectory planning, point cloud processing.

Índice

Índice de figuras

Índice de tablas

1. Introducción

Hoy en la actualidad, la implementación de nuevas tecnologías orientada al ámbito de la automatización ha impulsado el uso de los robots móviles en tareas de mucha complejidad como en estudios ambientales y en tareas de monitoreo o visualización de forma autónoma. La implementación de estos sistemas ha demostrado ser herramientas muy útiles en tareas que sean complicadas para una persona. Por el riesgo o la impotencia que estas tareas demandan, como en inspecciones en el ámbito industrial, monitoreo en ambientes extensos o exploración en ambientes con un limitado espacio de intervención. Como resultado a esta necesidad, se han desarrollado soluciones de detección y control eficientes que permiten a los robots móviles desplazarse de una forma segura y autónoma, gracias a sus características de diseño y método de locomoción que cada uno tiene en diferentes ambientes.

Como resultado de este trabajo es implementar un sistema de control a un robot móvil que permita planificar su trayectoria y recrear el entorno en tiempo real. Para este sistema de control se utiliza un sensor LiDAR que obtendrá datos del medio, los cuales serán procesados por algoritmos para la recreación del entorno. Adicional constará con una cámara que servirá para transmitir imagen del medio en tiempo real. Con la implementación de este sistema de control el robot podrá moverse de forma eficiente, siguiendo una ruta definida y evitando obstáculos que se presenten al momento de realizar la operación.

1.1. Justificación

Hoy en día, los robots móviles desempeñan un papel fundamental en múltiples aplicaciones gracias a su capacidad de realizar tareas que requieren movilidad, percepción del entorno y autonomía en la toma de decisiones. En entornos cerrados y controlados, como laboratorios, almacenes, fábricas o invernaderos, la incorporación de sistemas de navegación autónomos permite optimizar recursos, mejorar la precisión de los procesos y garantizar condiciones de seguridad, reduciendo así la exposición de los operadores humanos

a riesgos innecesarios [1].

En América Latina, la inversión en sistemas robóticos autónomos es reducida, debido a la escasez de recursos, tecnología e infraestructura. Como consecuencia, se genera una dependencia tecnológica respecto a los países desarrollados. En Ecuador, a pesar de su riqueza en recursos mineros y agrícolas, no se han implementado tecnologías de robótica autónoma, a pesar de los múltiples beneficios que ofrecen estos sistemas. Dado que la tendencia global se orienta hacia su desarrollo, la falta de adopción de estas tecnologías podría generar consecuencias negativas para el país [2].

Los sistemas de monitoreo y exploración no solo deben desplazarse de manera autónoma, sino también generar representaciones precisas del entorno para la toma de decisiones. Tecnologías como sensores LiDAR, cámaras y sistemas de ultrasonido han permitido avances significativos en el mapeo, facilitando la construcción de mapas locales y globales que mejoran la capacidad de percepción del robot. Sin embargo, la mayoría de las soluciones propuestas en trabajos previos se enfocan en entornos abiertos o simulaciones, dejando un vacío importante en la investigación aplicada a espacios cerrados y controlados [3].

1.2. Panorama actual

La robótica actual ha trascendido el ámbito de la fabricación para consolidarse como un pilar tecnológico en varias industrias. Según la Federación Internacional de Robótica (2023), el número de robots industriales en servicio en todo el mundo alcanzó los 3,9 millones de unidades en 2022, con una tasa de crecimiento anual del 12 %, lo que refleja una adopción acelerada, especialmente en las industrias automotriz, electrónica y logística. [4]. Este crecimiento sostenido que promedia el 13 % anual desde 2019 responde a la necesidad de incrementar la productividad y ejecutar tareas en entornos peligrosos para los trabajadores humanos [5].

Los robots móviles autónomos (AMR) han surgido como una categoría de gran relevancia por su capacidad para desplazarse en entornos dinámicos sin

intervención humana. A diferencia de los robots industriales convencionales, los AMR pueden adaptarse a condiciones cambiantes mediante la integración de tecnologías de percepción como sensores LiDAR, cámaras y sistemas de fusión de datos, lo que posibilita su aplicación en tareas de exploración, mapeo 3D y monitoreo autónomo. Este avance ha sido impulsado principalmente por el desarrollo de algoritmos de SLAM (Localización y Mapeo Simultáneos), fundamentales para mejorar la precisión y autonomía de estos sistemas [6].

En latinoamérica, países como Ecuador enfrentan el desafío de reducir la insuficiencia tecnológica en la robótica autónoma. Si bien el uso de herramientas tecnológicas para procesos riesgosos aún no ha alcanzado un desarrollo significativo, se han registrado iniciativas relevantes como la desarrollada durante la pandemia de COVID-19. Bajo estas condiciones, la empresa ATIR S.A. implementó en Guayaquil (2021) robots móviles equipados con visión artificial y ROS (Robot Operating System) para desinfección automatizada de ambientes cerrados, validando la eficacia de estas plataformas en entornos críticos y demostrando la capacidad local para desarrollar soluciones robóticas adaptadas a necesidades específicas [7].

1.3. Objetivos

1.3.1. Objetivo general

Desarrollar un sistema de control para la planificación de trayectoria y la generación de entorno en un robot móvil, con el fin de optimizar su desempeño en tareas de exploración y supervisión en entornos dinámicos.

1.3.2. Objetivos específicos

- Estudiar el funcionamiento y utilidad del sensor LIDAR para la adquisición de nubes de puntos para la reconstrucción del entorno.
- Procesar los datos de la nube de puntos para reconstruir un modelo del entorno.

- Implementar un sistema de navegación autónoma basado en la integración de sensores LIDAR y cámara, permitiendo la detección y el mapeo del entorno en tiempo real.
- Desarrollar un algoritmo de planificación de trayectoria para el desplazamiento del robot móvil, evitando obstáculos y garantizando una exploración eficiente del área asignada.
- Evaluar el desempeño del sistema de control mediante pruebas experimentales en distintos escenarios, analizando la precisión de la navegación y la calidad de la generación de entorno.

1.4. Fundamentos teóricos

Las siguientes secciones profundizarán en temas fundamentales que respaldan el diseño y operación de sistemas autónomos, como la robótica móvil, los lenguajes de programación y otros elementos clave de la automatización. El propósito es comprender las nuevas innovaciones de la tecnología que han servido de impulso en estas áreas de desarrollo y asi destacando la evolución, impactos que estas han tenido y sus usos prácticos.

Cada información es esencial para poder entender como es el funcionamiento de los robots móviles, de tal manera que se abarcan temas esenciales como la cinemática diferencial y la incorporación de sensores. Se explicarán su funcionamiento, los diversos entornos donde operan y la importancia que estos tienen en los diversos ámbitos como lo es en las industrias, la seguridad, en lo académico y el monitoreo del ambiente.

1.4.1. Robótica móvil

La robótica combina la informática y la ingeniería para desarrollar dispositivos capaces de realizar tareas en lugar de los seres humanos, principalmente en entornos industriales. Un robot se define como una máquina programable y ajustable que puede ejecutar tareas complejas, tomar decisiones y actuar en función de su entorno, gracias a la incorporación de diversos

sensores que le permiten adaptarse a múltiples funciones. A lo largo del tiempo, los avances en la ciencia y la tecnología han transformado la percepción de los robots: lo que antes era considerado ciencia ficción o parte de relatos futuristas, hoy se ha convertido en una herramienta de ingeniería altamente sofisticada y funcional [8] .

La robótica móvil puede definirse de manera sencilla como el campo que desarrolla sistemas robóticos capaces de desplazarse en distintos entornos y ejecutar tareas complejas, ya sea de forma autónoma o bajo el control de un operador humano. Estos robots están equipados con sensores y actuadores, lo que les permite percibir su entorno, interpretarlo y modificarlo según sea necesario, adaptándose así a diversas condiciones y desafíos del medio donde operan [9].

1.4.2. Elementos de un robot móvil

Un robot móvil está compuesto por diversos componentes que trabajan en conjunto para permitirle desplazarse, percibir su entorno e interactuar con él de forma eficaz (ver en la Fig. 1). Cada uno de estos elementos cumple una función específica dentro del sistema, y su correcta integración es fundamental para que el robot pueda desempeñar adecuadamente las tareas para las que fue diseñado. Entre los componentes principales se encuentran:

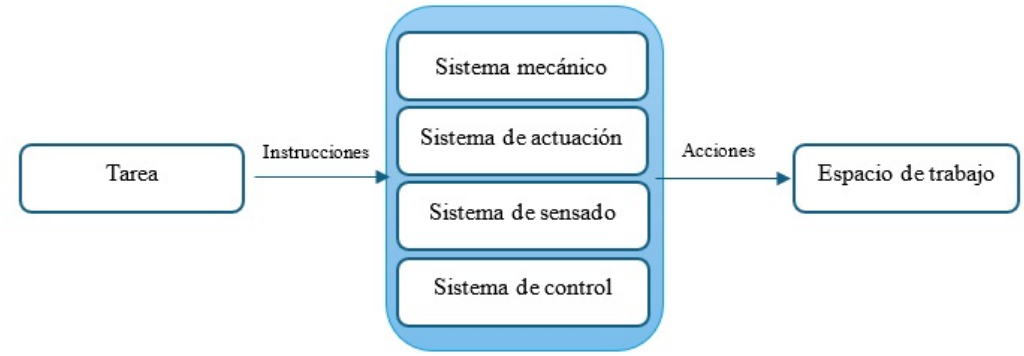


Figura 1: Componentes de un robot móvil [Fuente: Autor].

1.4.2.1. Sistema mecánico de un robot móvil

El soporte estructural de un robot es muy importante por lo que debe ser construido de una forma muy detallada y pensada para garantizar su perfecta maniobrabilidad que este tendrá, además de su estabilidad en las diversas tareas y correcta funcionalidad en diversos ambientes. Se debe tomar en cuenta el material con el que estará construido además de la correcta ubicación y distribución de los elementos tanto electrónicos y mecánicos. Asimismo, considerar el correcto sistema de movilidad, suspensión y la dirección que serán las responsables de que el robot se mueva de un lugar a otro con exactitud en distintos tipos de superficies. [10].

1.4.2.2. Sistema de actuación de un robot móvi

En robótica, los actuadores son componentes mecánicos que convierten la energía en movimiento, actuando como motores que permiten a los robots moverse y realizar tareas específicas. Sin ellos, el robot no sería más que una estructura inerte hecha de metal, plástico u otros materiales. Estos elementos dan vida al sistema: algunos permiten el movimiento, otros activan las articulaciones o activan efectos finales como agarrar o levantar objetos.[11].

1.4.2.3. Sistema de sensado de un robot móvi

Los sensores en robots móviles desempeñan un papel crucial al permitir la interacción con el entorno, la toma de decisiones en tiempo real y la ejecución de tareas de forma autónoma. Para que estos robots operen con plena autonomía en entornos dinámicos y no estructurados, como fábricas, almacenes o centros logísticos, es esencial contar con sistemas de posicionamiento robustos que permitan un reconocimiento ambiental instantáneo. Esto garantiza una navegación eficiente, una adecuada orientación y una planificación de trayectorias basada en la ubicación precisa del robot [12].

1.4.2.4. Sistema de control de un robot móvi

El sistema de control constituye el núcleo funcional de un robot móvil, siendo el encargado de procesar la información proveniente de los sensores, ejecutar algoritmos de navegación, planificar trayectorias y controlar los actuadores.

El controlador, como dispositivo electrónico central, realiza los cálculos necesarios para que el robot ejecute las funciones para las que fue diseñado. Este recibe señales del sistema de sensores, las interpreta y determina las acciones que deben realizarse en los sistemas de actuación, enviando las señales de control correspondientes. [13].

La elección de la placa de control adecuado depende de factores como la potencia de procesamiento, el consumo de energía, la compatibilidad de los sensores, la facilidad de programación y el costo. Por ello, en el desarrollo de robots móviles se utilizan diversas opciones, las cuales se eligen según las necesidades específicas de cada proyecto. Los más utilizados son Arduino, ESP32, Raspberry Pi y NVIDIA Jetson, cuyas principales características se describen en la siguiente tabla 1:

Tabla 1: Comparativa de placas de control para robots móviles.

Placa de control	**Procesador**	**Conectividad**	**Lenguajes compatibles**	**Ventajas**	**Limitaciones**
Arduino Uno	ATmega328P (8 bits)	Sin Wi-Fi ni Bluetooth (requiere módulos externos)	C / C++ (Arduino IDE)	Fácil de usar, muy estable, bajo consumo	Limitada capacidad de procesamiento y memoria
ESP32	Tensilica Xtensa LX6 32 bits	Con Wi-Fi, Bluetooth	C / C++ / MicroPython	Económico, buena velocidad, ideal para control inalámbrico y proyectos IoT	No apto para procesamiento de imágenes o tareas pesadas
Raspberry Pi 5	Broadcom BCM2712 (Quad-core Cortex-A76)	Con Wi-Fi, Bluetooth, Ethernet, PCIe	Python / C / C++ / Node.js	Alto rendimiento, fácil integración con sensores y cámaras, programación desde Windows	Requiere buena refrigeración y fuente estable
NVIDIA Jetson Nano	ARM A57 + GPU 128 CUDA cores (Quad-core)	Con Wi-Fi (externo), Ethernet	Python / C++ / CUDA	Potente para IA y visión artificial	Elevado consumo y costo para proyectos básicos

En este proyecto se eligió la Raspberry Pi 5 como unidad de control principal por ofrecer un equilibrio óptimo entre rendimiento, conectividad y sencillez de programación. Su procesador Broadcom BCM2712 de cuatro

núcleos a 2.4 GHz y su memoria RAM de hasta 8 GB permiten desarrollar y ejecutar aplicaciones de control, procesar datos de sensores y comunicarse con periféricos en tiempo real.

Si bien la Raspberry Pi 5 esta planteada para utilizar Linux también tiene plataformas de control que se integra en el sistema operativo de Windows, como las mas comunes tenemos Python, RealVNC y Visual Studio Code. Las cuales son las principales que se utilizan para la programación, monitoreo remoto y para la supervisión de robots. Debido a su fácil configuración, facilita la compatibilidad con elementos electrónicos externos, debido a esto se considera una alternativa muy conveniente y eficiente para controlar los robots móviles.

1.4.3. Clasificación de los robots móviles según su medio de movilización

El creciente interés en los robots móviles ha impulsado el desarrollo de diversas configuraciones estructurales, permitiendo clasificarlos de manera general según el entorno en el que operan (ver en la Fig. 2).

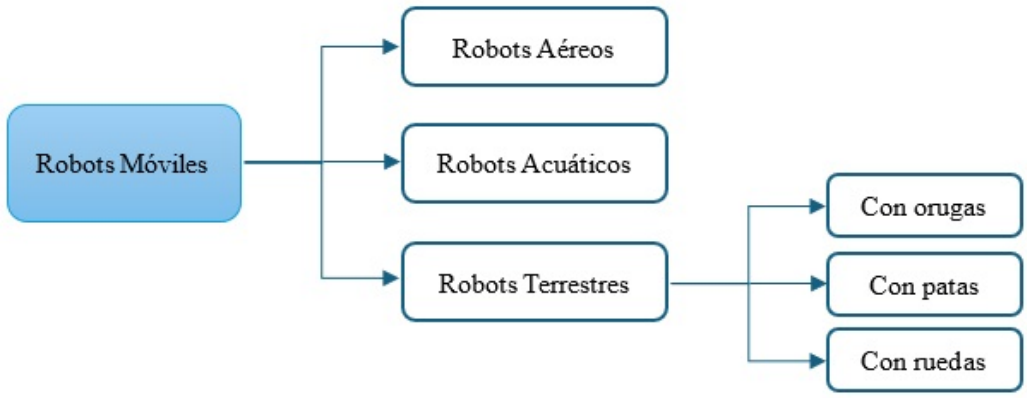


Figura 2: Clasificación de los robots móviles en basada en su método de movimiento [Fuente: Autor].

1.4.3.1. Robots aéreos

Los avances en robótica aérea han posibilitado la creación de drones, también conocidos como vehículos aéreos no tripulados (VANT) (ver en la Fig. 3), los cuales se han vuelto cada vez más comunes en la investigación científica y el ámbito comercial. Su creciente popularidad los ha posicionado como una de las principales tendencias en desarrollo, siendo objeto de intensos estudios y mejoras.

El uso de estos drones ha aumentado considerablemente en actividades que implican riesgos o dificultades para aeronaves tripuladas, como la localización de incendios, la detección de derrames de petróleo en océanos, el monitoreo del tráfico y la inspección de redes eléctricas. Uno de los principales desafíos de estos vehículos radica en garantizar que puedan realizar maniobras complejas de manera autónoma, adaptándose a los cambios en su entorno sin la intervención de un operador [14].

Figura 3: Robot aéreo [15].

1.4.3.2. Robots acuáticos

La robótica acuática surge de la necesidad humana de realizar tareas específicas en entornos submarinos donde la intervención directa resulta difícil, peligrosa o incluso imposible debido a las condiciones extremas (ver en la Fig. 4). Estos sistemas automatizados han sido diseñados para ejecutar actividades que requieren un alto grado de precisión y seguridad, reduciendo los

riesgos para el personal involucrado. Entre sus múltiples aplicaciones destacan la inspección y mantenimiento de depósitos de agua destinados al consumo humano, el monitoreo de acuarios y piscinas para el cultivo de especies acuáticas, y la evaluación de la contaminación en los fondos marinos, lo que permite un análisis preciso del estado ambiental de esos ecosistemas [16].

Un ejemplo típico de su uso es la inspección en tuberías submarinas, que pueden fallar o agrietarse y tener fugas cuando se exponen a condiciones adversas. Estos accidentes no sólo afectan a la integridad de las infraestructuras, sino que también tienen un grave impacto en el medio ambiente, perjudicando gravemente a la flora y fauna marina. Por este motivo, la robótica acuática se ha convertido en una herramienta esencial para monitorear, mantener y preservar ecosistemas acuáticos y estructuras submarinas críticas [16].

Figura 4: Robot acuático [17].

1.4.3.3. Robots terrestres

Una forma muy interesante de clasificar a los robots terrestres está relacionada con su movilidad. Según su desempeño, capacidad de movimiento y toma de decisiones, se pueden distinguir de varios tipos:

Robots con Orugas: En terrenos irregulares, muy resbaladizos o con numerosos obstáculos, los robots que utilizan orugas resultan ser una alternativa más adecuada que aquellos que se desplazan con ruedas (ver en la

Fig. 5). Esto se debe a que las orugas proporcionan una mayor tracción, lo que mejora la estabilidad del robot y reduce el riesgo de deslizamiento. A pesar de su capacidad para desempeñarse mejor en condiciones exigentes, estos sistemas conservan una estructura mecánica relativamente simple y un control similar al de los robots con ruedas, lo que los convierte en una solución eficiente, robusta y práctica para entornos difíciles [13].

Figura 5: Robot móvil con orugas [18].

Robots con Patas: La robótica ha encontrado una gran fuente de inspiración en el comportamiento y las habilidades de los seres vivos, ya que la naturaleza ha desarrollado soluciones de movilidad impresionantes. En seres como humanos, insectos y otros animales, la capacidad de moverse mediante extremidades es fundamental para interactuar con su entorno. Este tipo de movimiento, que varía según la especie, proporciona una gran flexibilidad y versatilidad en la locomoción. En la robótica móvil, se han adoptado conceptos similares para diseñar robots que imiten estas capacidades (ver en la Fig. 6), utilizando configuraciones de extremidades que les permiten realizar maniobras mucho más complejas y precisas que las alcanzadas con ruedas u orugas [9].

Figura 6: Robot móvil con patas [19].

Robots con ruedas: Los robots móviles con ruedas son el tipo más común de robots móviles y se utilizan frecuentemente para el transporte de materiales, mercancías o personas (ver en la Fig. 7). Las ruedas les proporcionan una movilidad altamente eficiente, ya que existe una excelente relación entre la distancia recorrida y la energía consumida. Sin embargo, su funcionamiento suele estar limitado a entornos con superficies planas o ligeramente irregulares, lo que restringe su uso en terrenos más accidentados [20].

Figura 7: Robot móvil con rueda [21].

1.4.4. Tipos de ruedas en los robots móviles

En el ámbito de la robótica móvil, las ruedas son fundamentales para el desplazamiento y la agilidad de los robots. Hay diversos estilos de ruedas, cada uno diseñado con particularidades que se ajustan a varios entornos, usos y requisitos de movilidad. A continuación, se presenta una clasificación de los tipos más comunes.

1.4.4.1. Ruedas convencionales o estándar

Este tipo de ruedas puede entenderse como neumáticos comunes o convencionales. Se clasifican en tres tipos principales: ruedas fijas, ruedas orientables centradas y ruedas orientables descentradas, también conocidas como tipo castor [13].

Ruedas fijas: Las ruedas estáticas en robots móviles son un tipo de rueda sin capacidad direccional, fijada de forma rígida al chasis del robot (ver en la Fig. 8). Solo permiten el movimiento hacia adelante o hacia atrás, girando únicamente sobre su propio eje con una velocidad de rotación $\dot{\phi}$, mientras su orientación β respecto al chasis permanece constante [22].

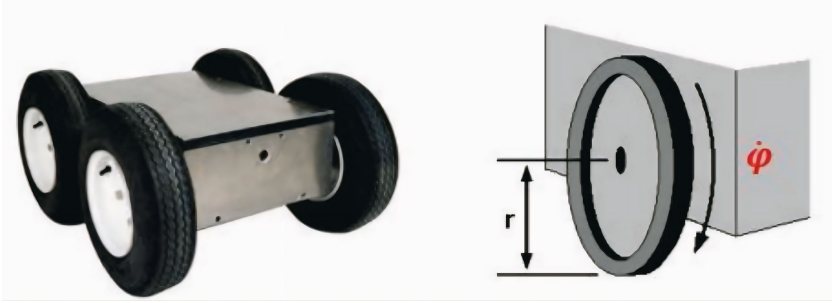


Figura 8: Robot móvil con ruedas fijas [13].

Ruedas orientables centradas: Las ruedas orientables centradas son componentes fundamentales en robots móviles que requieren alta maniobrabilidad. Estas ruedas permiten que el robot se desplace en cualquier dirección sin alterar su orientación, facilitando movimientos laterales, diagonales y giros sobre su propio eje (ver en la Fig. 9) [23].

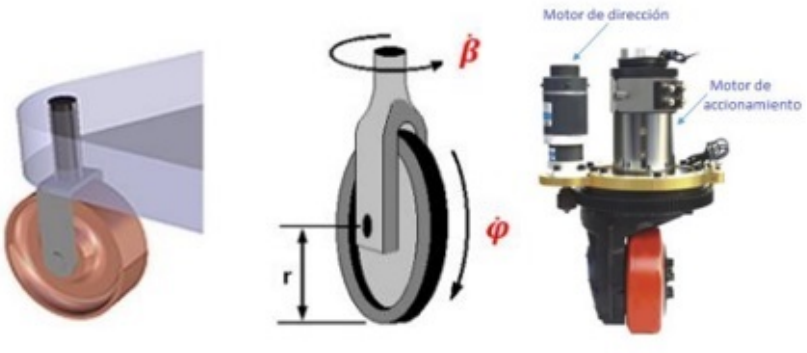


Figura 9: Rueda orientable centrada [13].

Ruedas Orientables Descentradas o Castor: Las ruedas giratorias descentradas, también conocidas como ruedas tipo castor (ver en la Fig. 10), son componentes pasivos ampliamente utilizados en robots móviles para brindar soporte y estabilidad adicional. Se caracterizan por tener un eje de rotación vertical desalineado respecto al eje de la rueda, lo que les permite girar libremente en cualquier dirección y adaptarse fácilmente a los cambios de orientación del robot [24].

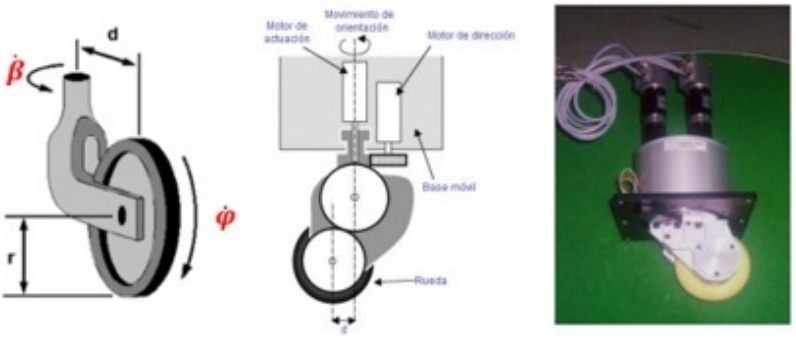


Figura 10: Rueda orientable descentrada o castor activa [13].

1.4.4.2. Ruedas especiales u omnidireccionales

Las ruedas omnidireccionales son elementos clave en la robótica móvil, ya que permiten a los robots desplazarse en cualquier dirección sin modificar su orientación. Están diseñadas con rodillos o cilindros colocados en ángulos específicos alrededor de su circunferencia, lo que facilita movimientos laterales,

diagonales y giros sobre su propio eje. Existen tres tipos principales de estas ruedas especializadas: universales, mecanum y esféricas [25].

Rueda universal : Las ruedas omnidireccionales universales son componentes esenciales en la robótica móvil, ya que permiten el desplazamiento en cualquier dirección sin necesidad de modificar la orientación del robot (ver en la Fig. 11). Su principal característica es el movimiento holonómico, lo que les permite moverse libremente en todas las direcciones dentro de un plano, ofreciendo así una maniobrabilidad excepcional en espacios reducidos [26].

Figura 11: Diseños de ruedas universales [13].

Rueda mecanum o sueca : Las ruedas mecanum están compuestas por rodillos montados en un ángulo, generalmente de 45°, a lo largo de su circunferencia, lo que permite al robot moverse en múltiples direcciones, incluyendo desplazamientos laterales y diagonales, mediante el control individual de la velocidad y dirección de cada rueda (ver en la Fig. 12). Por otro lado, las ruedas suecas, también conocidas como ruedas omnidireccionales, tienen rodillos dispuestos perpendicularmente al plano de la rueda, facilitando el movimiento lateral. Un diseño destacado que utiliza este tipo de ruedas es el "Kiwi Drive", que emplea tres ruedas omnidireccionales distribuidas equitativamente a 120°, ofreciendo así una movilidad holonómica completa [27].

Figura 12: Rueda omnidireccional tipo mecanum o sueca [13].

Rueda de bola o esférica : Estas ruedas no presentan restricciones directas al movimiento, lo que les permite desplazarse en múltiples direcciones, al igual que las ruedas tipo castor, universales y mecanum (ver en la Fig. 13). En este tipo de configuración, el eje de giro puede orientarse en cualquier dirección arbitraria. Para alcanzar este nivel de libertad, se emplea un anillo activo impulsado por un motor y una caja de engranajes, que transfiere la energía a una esfera mediante rodillos y fricción, permitiendo que esta gire de forma inmediata en cualquier dirección [13].

Figura 13: Arquitectura de rueda esférica motorizada por Kumagai y Ochiai [13].

1.4.5. Configuración de los robots móviles según la cantidad de ruedas

La mayoría de los robots móviles con ruedas están diseñados con características específicas que los hacen aptos para tareas concretas. Estas tareas definen desde el inicio su estructura, incluyendo el tipo de ruedas, el sistema de propulsión, el mecanismo de dirección y la configuración física del robot. Por lo general, estos sistemas se reparten a lo largo de los ejes de las ruedas según las necesidades de velocidad, maniobrabilidad y adaptabilidad del terreno.A continuación, se presentan las principales características de diseño de distintos tipos de robots móviles, organizados según la cantidad de ruedas que utilizan [13].

1.4.5.1. Robots con una rueda

Los robots móviles de una sola rueda, conocidos como robots monociclo, representan una categoría avanzada dentro de la robótica móvil (ver en la Fig. 14). Se caracterizan por su diseño compacto y por contar con un único punto de contacto con el suelo, lo que les otorga un radio de giro igual a cero y una gran flexibilidad de movimiento. Estas cualidades los hacen especialmente útiles para el desplazamiento libre y el transporte de cargas en espacios reducidos. Sin embargo, uno de los principales desafíos en su diseño y operación es mantener el equilibrio lateral [28].

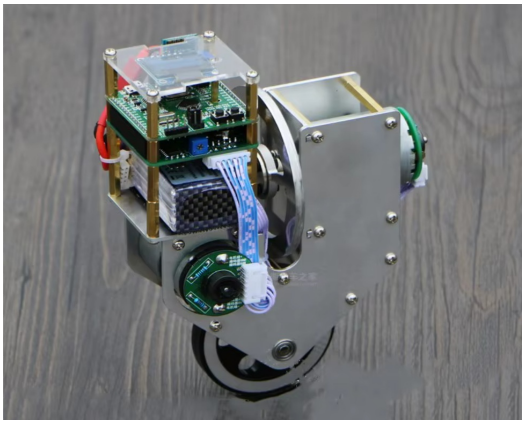


Figura 14: Robots monociclos autoequilibrados con control automático PID [13].

1.4.5.2. Robots con dos ruedas

En términos generales, existen dos tipos de robots móviles con dos ruedas: los de tipo bicicleta y los de tipo péndulo invertido.

Robot tipo bicicleta : En el caso del robot tipo bicicleta, la configuración habitual consiste en una rueda delantera orientable que guía el movimiento y una rueda trasera fija que proporciona la tracción (ver la Fig. 15). Este diseño requiere sistemas de control avanzados para mantener el equilibrio y ejecutar maniobras precisas. su aplicación es limitada debido a que no puede mantenerse en equilibrio por sí solo al detenerse [29].

Figura 15: Ejemplo de un robot tipo bicicleta [13].

Robot tipo péndulo invertido : Los robots de dos ruedas que operan bajo el principio del péndulo invertido (ver en la Fig. 16), categorizados bajo la sigla TWIP, correspondiente a Two-Wheeled Inverted Pendulum, son sistemas dinámicamente inestables que requieren un control activo para mantenerse en equilibrio. Estos robots suelen tener una estructura vertical montada sobre dos ruedas paralelas impulsadas por motores eléctricos, y utilizan sensores como giroscopios y acelerómetros para medir la inclinación y la velocidad angular, lo que les permite ajustar continuamente su postura y estabilidad [30].

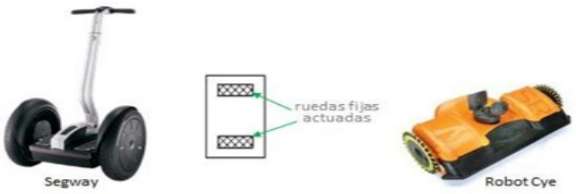


Figura 16: Ejemplos de robots tipo péndulo invertido [13].

1.4.5.3. Robots con tres ruedas

Los robots de tres ruedas son una de las configuraciones más comunes en la robótica móvil debido a su estabilidad estática y a la sencillez de su estructura. Existen múltiples variantes de diseño, las cuales dependen principalmente del tipo de ruedas seleccionadas para su implementación (ver en la Fig. 17) [13].

A continuación, se muestran varios ejemplos populares:

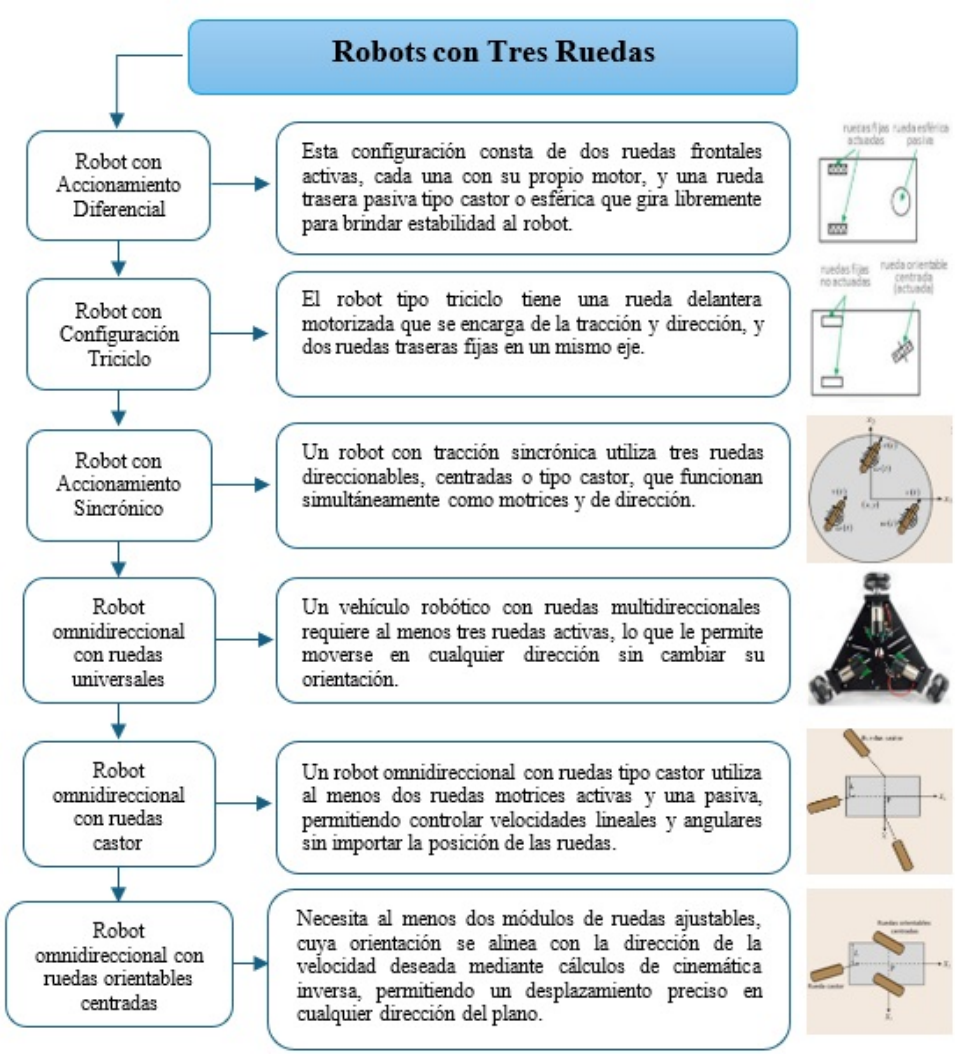


Figura 17: Ejemplos de robots con tres ruedas [Fuente: Autor].

1.4.5.4. Robots con cuatro ruedas

Los modelos mencionados anteriormente pueden ser ampliados a configuraciones de vehículos de cuatro ruedas con el fin de mejorar tanto la estabilidad como la tracción sobre el terreno (ver en la Fig. 18). Alternativamente, se pueden incorporar ruedas activas, lo que requiere gestionar adecuadamente su movimiento mediante la resolución de la cinemática inversa. Además, los robots de cuatro ruedas necesitan un sistema de suspensión que garantice el contacto constante de todas las ruedas con el suelo, especialmente en superficies irregulares, evitando así que alguna de ellas pierda adherencia [13].

A continuación, se muestran algunos tipos de robots con cuatro ruedas.

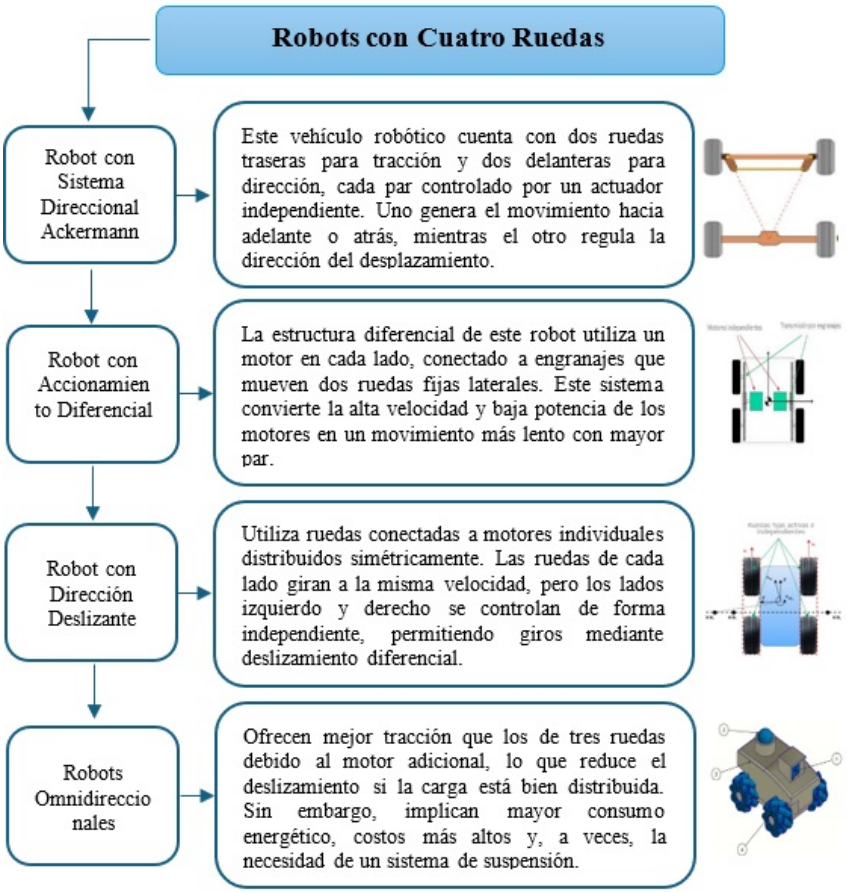


Figura 18: Ejemplos de robots con cuatro ruedas [Fuente: Autor].

1.4.6. Cinemática de los robots móviles con rudas

La cinemática es una rama de la mecánica que se enfoca en el estudio del movimiento de los cuerpos, sin considerar su masa, momento de inercia ni las fuerzas o torques que lo producen. En el contexto de la robótica móvil, la cinemática cumple un papel fundamental al analizar cómo varían la posición, la velocidad y la aceleración de un robot durante su desplazamiento, sin tomar en cuenta los factores dinámicos que intervienen en dicho movimiento [13].

1.4.7. Cinemática directa e inversa de los robots móviles con ruedas

La cinemática directa y la cinemática inversa conforman las metodologías que son fundamentales para estudiar el desplazamiento de los robots, particularmente para entender como es la relación de la posición y la orientación final del robot.

Cinemática directa : Permite calcular la posición y orientación de un robot que interactúa con su entorno dentro de un sistema de coordenadas de referencia global. Se basa en valores conocidos de la configuración interna del robot, como los ángulos de las articulaciones o desplazamientos lineales, y los parámetros geométricos de sus segmentos. Se parte de la información interna del sistema para predecir la ubicación de un punto específico del robot, normalmente su extremo o centro [31].

En el caso de un robot móvil con ruedas, al considerar características como el radio de las ruedas, su posición respecto al chasis, la distancia entre ellas y la ubicación de un punto de referencia del robot (usualmente el centro geométrico), así como las velocidades angulares de rotación de las ruedas $\dot{\varphi}_i$ que representan la rapidez del giro de cada rueda y las variaciones de orientación de las mismas $\dot{\beta}_i$ que indican el cambio en el ángulo de orientación de cada rueda, es posible calcular la velocidad cartesiana del robot $(\dot{x}, \dot{y}, \dot{\theta})$ en el sistema de coordenadas global (ver en la Fig. 19) [31].

Con esta información, no solo se pueden determinar las velocidades linea-

les y angulares del robot, sino también estimar su posición en el plano global a lo largo del tiempo.

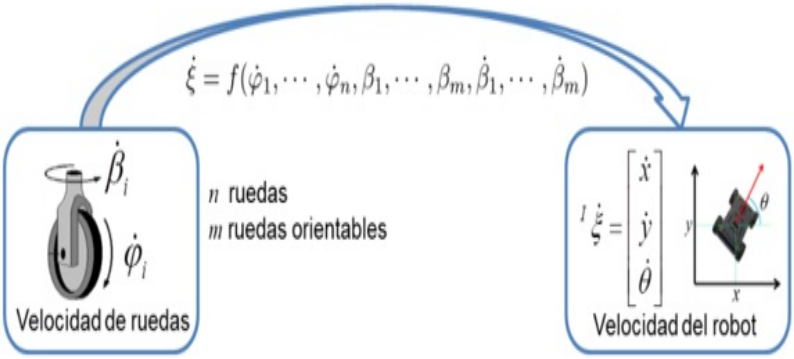


Figura 19: Representación de la cinemática directa [13].

Cinemática inversa : Consiste en determinar qué valores deben adoptar las articulaciones o mecanismos del robot para que este alcance una posición y orientación deseadas dentro del entorno. A diferencia de la cinemática directa, este planteamiento es más complejo, ya que una misma configuración final puede alcanzarse por múltiples combinaciones de movimientos internos (soluciones múltiples) o, en algunos casos, no existir solución alguna debido a las limitaciones geométricas del robot o restricciones impuestas por el entorno [31].

En el caso de los robots móviles con ruedas, la cinemática inversa facilita el cálculo de las velocidades de rotación $\dot{\varphi}_i$ que representan la rapidez del giro de cada rueda, las velocidades de dirección $\dot{\beta}_i$ que indican la variación de orientación de cada rueda, y las orientaciones necesarias de las ruedas β_i, para que el robot pueda desplazarse con una velocidad cartesiana deseada $(\dot{x}, \dot{y}, \dot{\theta})$ en el plano (ver en la Fig. 20). Al aplicar este análisis, se determina cómo deben actuar individualmente las ruedas para generar el movimiento requerido en el chasis del robot [31].

Esto es crucial en sistemas de control y navegación, donde se necesita que

el robot siga trayectorias específicas o se desplace con precisión hacia ciertos puntos en su entorno.

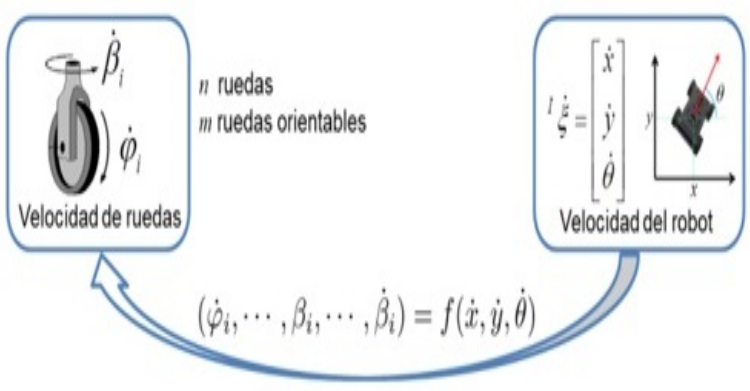


Figura 20: Representación de la cinemática inversa [13].

1.4.8. Modelo cinemático del robot diferencial

El modelo cinemático de un robot diferencial describe el desplazamiento de un robot móvil equipado con dos ruedas motrices laterales, una a la izquierda y otra a la derecha, que pueden ser controladas de forma independiente. En muchos casos, este tipo de configuración incluye además una o más ruedas pasivas para mantener el equilibrio. Su uso es común en robótica móvil debido a su simplicidad, fácil implementación y control eficiente [32].

Las variables indispensables de este sistema son:

- v: velocidad lineal en el punto central de las ruedas
- ω: velocidad angular del robot con relación al eje vertical
- R: radio de las ruedas del robot.
- L: distancia que existe entre las ruedas.
- ω_r, ω_l: velocidades angulares de la rueda derecha e izquierda, respectivamente.

Correspondencia cinemática entre las velocidades angulares de las ruedas y la velocidad lineal del robot

$$v = \frac{R}{2}(\omega_r + \omega_l) \tag{6}$$

$$\omega = \frac{R}{L}(\omega_r - \omega_l) \tag{7}$$

Modelo cinemático en el plano:

$$\dot{x} = v\cos(\theta) \tag{8}$$

$$\dot{y} = v\sin(\theta) \tag{9}$$

$$\dot{\theta} = \omega \tag{10}$$

En el cual $(x,\ y)$ representa la posición del robot en el plano y θ su orientación con respecto al eje x. Este conjunto de ecuaciones permite describir cómo se desplaza el robot en función del control ejercido sobre sus ruedas, siendo fundamentales para tareas como la navegación, la planificación de trayectorias y la simulación de movimiento [32].

Para el desarrollo del modelo cinemático, se establecen las siguientes premisas::

- El centro de masa del sistema se localiza en el centro geométrico del robot.
- El sistema de referencia se encuentra ubicado entre las dos ruedas motrices la cual está alineado con el eje de rotación.
- La rueda de apoyo, la cual permite el movimiento en múltiples direcciones (omnidireccional), no impone restricciones al desplazamiento en el robot.
- La velocidad angular de la rueda derecha se muestra como $\dot{\phi}_d$, mientras que la rueda izquierda se representa de forma $\dot{\phi}_i$.

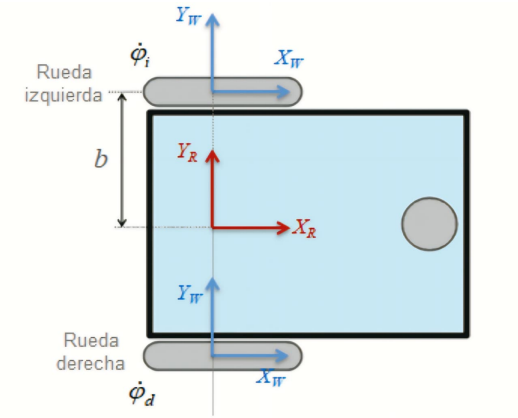


Figura 21: Geometría de un robot con accionamiento direccional [13]..

1.4.9. Representación de la posición y orientación

Al diseñar un robot móvil con ruedas, es importante determinar la relación entre el movimiento del chasis y sus ruedas con respecto al sistema de referencia global, lo que permite determinar la pose del robot, es decir, su posición y orientación en el plano. En este análisis, el robot se considera como un cuerpo rígido que se mueve en un plano horizontal.. El chasis del robot presenta tres grados de libertad en dicho plano: dos asociados a su posición lineal (x, y) y uno a su orientación angular θ en torno al eje vertical (eje z), perpendicular al plano [13].

Para describir la postura del robot en el plano, se establece una relación entre dos sistemas de referencia: uno global o inercial, denotado como $\{I\}$, y uno local vinculado al propio robot, denotado como $\{R\}$ (ver en la Fig. 22). El sistema global está definido por los ejes X_I y Y_I, que forman una base inercial arbitraria sobre el plano. En tanto, el sistema local $\{R\}$ tiene como origen un punto R, generalmente el centro de masa del robot, y está definido por los ejes móviles X_R y Y_R. Esta configuración permite describir con precisión la posición del robot respecto al entorno, facilitando así el desarrollo de modelos de navegación y control [13].

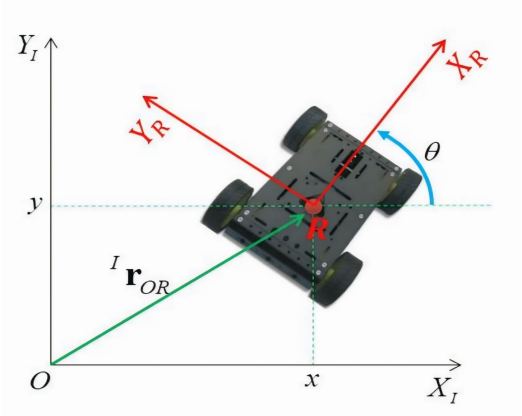


Figura 22: Modelado de la configuración espacial del robot mediante sistemas de coordenadas global y local[13].

La ubicación de un robot movil referente al sistema global se describe mediante las coordenadas x e y, a la vez que la diferencia angular que existe entre los sistemas de referencia global y local se representa por medio de θ. La ubicacion que tiene el robot se expresa como un vector compuesto por estos tres elementos, señalando su condición de relación con el sistema de referencia global, el cual esta representado a través de la ecuación (1).

$$ {}^I\boldsymbol{\xi} = \begin{bmatrix} x \\ y \\ \theta \end{bmatrix} \tag{1} $$

Para descubrir el desplazamiento del robot desde una perspectiva local a una global, es necesario establecer una relación entre los movimientos a lo largo de los ejes de ambos sistemas de referencia. Esta conversión depende de la postura actual del robot y se realiza mediante una matriz de rotación ortogonal alrededor del eje Z representada a continuación:

$$ {}^R\mathbf{R}_I(\theta) = \begin{bmatrix} \cos\theta & \sin\theta & 0 \\ -\sin\theta & \cos\theta & 0 \\ 0 & 0 & 1 \end{bmatrix} \tag{2} $$

Dicha matriz permite mapear el movimiento del sistema local de referencia global $\{X_I, Y_I\}$ al movimiento en términos del sistema de referencia local

$\{X_R, Y_R\}$, esta operación se denota como:

$$^{R}\xi = {}^{R}R_I \, {}^{I}\xi \tag{3}$$

A partir de esta misma matriz, también es posible calcular el desplazamiento en el sistema global en función del movimiento en el sistema local.

$$^{I}\xi = {}^{I}R_R \, {}^{R}\xi = \left({}^{R}R_I\right)^{-1} \, {}^{R}\xi \tag{4}$$

Como parte de la estrategia, primero se determina el efecto de cada rueda del robot en su sistema de referencia local , $^{R}\vec{\xi}$, luego, se emplea la matriz de rotación $^{I}R_R$. Aunque obtener la inversa de una matriz puede ser complejo, sin embargo, La conversión entre sistemas de referencia se simplifica considerablemente en este caso, ya que consiste en transformar el vector $^{R}\vec{\xi}$ del sistema local al global $^{I}\vec{\xi}$, aprovechando las propiedades de la matriz de rotación ortogonal $^{I}R_R = (R_I)^{-1} = (R_I)^{T}$:

$$^{I}R_R(\theta) = \begin{bmatrix} \cos\theta & -\sin\theta & 0 \\ \sin\theta & \cos\theta & 0 \\ 0 & 0 & 1 \end{bmatrix} \tag{5}$$

1.4.10. Estimación de la postura mediante odometría

La estimación computacional de la postura relativa (posición y orientación) de un robot móvil en un instante determinado se realiza integrando el modelo cinemático respecto al sistema de referencia global, técnica conocida como odometría. Este método de navegación, empleado tradicionalmente en barcos, aviones y automóviles, ha sido adoptado también en robots móviles. La odometría se basa en el uso de datos sensoriales para estimar la ubicación y dirección de un objeto en movimiento sobre un plano. Para ello, se utilizan sensores en las ruedas, como los codificadores, y sensores de orientación angular, como los giroscopios [33].

La determinación de la postura del robot móvil, a partir de variables de control como (x, y, θ), se realiza utilizando las siguientes relaciones obtenidas en la ecuación de cinemática directa:

$$x(t) = \int_0^t v(t)\cos(\theta(t))\,dt$$
$$y(t) = \int_0^t v(t)\sin(\theta(t))\,dt$$
$$\theta(t) = \int_0^t \omega(t)\,dt$$

Podría resultar beneficioso desarrollar un modelo de tiempo discreto a partir de estas ecuaciones, ya que sería útil para simulaciones computacionales y otras aplicaciones en dicho dominio. Dado que las ecuaciones presentan una naturaleza no lineal, no es adecuado emplear las técnicas convencionales utilizadas para transformar sistemas lineales de tiempo continuo a tiempo discreto. En su lugar, una alternativa viable es el uso del método de integración de Euler, el cual proporciona una estimación de la integración basada en una aproximación de primer orden de la serie de Taylor [33].

1.4.11. Planificación de trayectoria

Dada una ubicación inicial del robot, el objetivo es calcular cómo desplazarlo gradualmente hasta alcanzar una ubicación final, asegurando que no colisione con los obstáculos del entorno. Esta problemática abarca tanto a robots móviles como a vehículos y robots con articulaciones, donde la posición representa de forma única el estado del robot en su entorno. El algoritmo recibirá como entradas la posición inicial y final del robot, junto con una descripción geométrica que puede ampliarse a aspectos cinemáticos, dinámicos y a la representación de los obstáculos. Como salida, se espera obtener una trayectoria detallada compuesta por una serie de transformaciones secuenciales que el robot debe seguir para trasladarse de forma segura desde su punto de origen hasta el destino final [34].

1.4.12. Algoritmos de localización

Los algoritmos de localización son fundamentales para que un robot pueda determinar con precisión su posición y orientación dentro de un entorno,

utilizando sensores como LiDAR, cámaras, IMU y odometría. Esta capacidad es especialmente crítica cuando el robot debe desplazarse en espacios dinámicos o desconocidos, donde no se puede depender de información previa [35].

1.4.13. Sistemas de comunicación

El proceso de comunicación se define como el mecanismo que posibilita el flujo de información a través de diferentes localizaciones., involucrando distintos elementos que permiten que el mensaje llegue correctamente desde el emisor hasta el receptor. Para que este proceso sea verdaderamente efectivo, es fundamental que el destinatario comprenda el mensaje de forma clara y sin dificultades, tal como fue intencionado por quien lo emite [36].

1.4.14. Elementos de un sistema de comunicación

En el proceso de comunicación intervienen varios elementos esenciales que permiten que el mensaje fluya de manera clara y efectiva entre el emisor y el receptor, estos elementos se mencionan en la tabla 2:

Tabla 2: Elementos de un sistema de comunicación.

Emisor	Es la persona, sistema o entidad que origina el mensaje. Es quien transmitie la información a otros.
Mensaje	Es el contenido o conjunto de datos que el emisor desea comunicar.
Referente	Se refiere al tema o asunto del que trata el mensaje.
Canal	Constituye el medio físico o lógico que posibilita la transmisión del mensaje, estableciendo el enlace de comunicación entre los elementos emisores y receptores..
Receptor	El receptor representa el nodo destino que adquiere el mensaje desde el medio de transmisión y ejecuta su procesamiento semántico.
Código	Es el conjunto de signos, símbolos o reglas compartidas entre el emisor y el receptor, que permiten codificar y decodificar el mensaje.
Contexto	Es el entorno o situación en la que se produce la comunicación.

1.4.15. Comunicación mediante tecnologías inalámbricas

La tecnología inalámbrica permite que los dispositivos se conecten a una red sin la necesidad de utilizar cables físicos. Esta conectividad se logra a

través de puntos de acceso configurados para emitir y recibir datos mediante ondas electromagnéticas, lo que facilita la transmisión de información de manera eficiente y flexible. Se pueden clasificar en dos grandes grupos:

1.4.15.1. Redes de largo alcance

Están diseñadas para conectar dispositivos que se encuentran en distintas ubicaciones geográficas, permitiendo la comunicación a grandes distancias. Son comunes en sistemas de telecomunicaciones, redes móviles o enlaces satelitales.

1.4.15.2. Redes de corto alcance

Estas redes permiten la transferencia de datos entre dispositivos que se encuentran próximos entre sí. Son ampliamente utilizadas en entornos domésticos, oficinas y sistemas personales, como el Wi-Fi, Bluetooth o NFC [37].

A continuación, se presentan varias de estas tecnologías, sus rasgos distintivos y las ventajas que ofrecen en diversas aplicaciones.

Tecnología de Radio Frecuencia (RFID)

La tecnología de identificación por radiofrecuencia, conocida como RFID (por sus siglas en inglés: Radio Frequency Identification), es un sistema avanzado que permite la identificación y seguimiento de productos mediante el uso de ondas de radio. Su funcionamiento se basa en la comunicación entre un lector y un microchip integrado en una etiqueta o dispositivo, sin necesidad de contacto visual directo, lo que representa una clara ventaja frente a los tradicionales códigos de barras [38].

Tecnología Bluetooth

La tecnología inalámbrica Bluetooth es un sistema de transmisión de datos de corto alcance que permite la conexión entre dispositivos electrónicos sin la necesidad de cables. Su principal función es facilitar la comunicación de datos entre dispositivos digitales, como computadoras, teléfonos móviles, cámaras digitales, auriculares y otros equipos. El alcance efectivo de Bluetooth es de aproximadamente 10 metros, lo que lo convierte en una opción ideal para

conexiones cercanas, como la transferencia de archivos entre dos dispositivos o la conexión de periféricos a un ordenador [39].

Tecnología Zigbbe

Zigbee es un protocolo de comunicación que se utiliza para conectar dispositivos inteligentes en el hogar, como lo son bombillas, enchufes y cerraduras inteligentes. Este sistema permite que estos dispositivos se comuniquen entre sí en una red local, facilitando su gestión y automatización. Una ventaja es que puedes utilizarlo como un mando a distancia, como el mando a distancia IKEA Tradfri, para controlar estos dispositivos sin un mando tradicional [40].

Tecnología Wifi

Wi-Fi es una tecnología de telecomunicaciones que permite la conexión inalámbrica entre dispositivos electrónicos, como computadoras, teléfonos y otros equipos, facilitando el intercambio de datos o la conexión a una red de Internet. Utiliza ondas de radio para transmitir información entre los dispositivos, permitiendo la creación de redes locales (LAN) sin la necesidad de cables. El rango de cobertura de Wi-Fi suele ser de hasta 100 metros, aunque esta distancia puede variar dependiendo de factores como el entorno y la cantidad de interferencias [41].

1.4.16. Lenguajes de programación para robot móviles

Los lenguajes de programación son herramientas fundamentales que permiten a los desarrolladores comunicarse con las computadoras mediante instrucciones precisas [42].

A continuación, se presenta un resumen de algunos de los lenguajes más utilizados, destacando sus principales características y aplicaciones:

1.4.16.1. JavaScript

Es un lenguaje interpretado y orientado a objetos, ampliamente utilizado en el desarrollo web para crear sitios dinámicos e interactivos. Su compatibilidad con todos los navegadores modernos, junto con su facilidad de aprendizaje

y versatilidad, lo convierten en una herramienta clave tanto en el desarrollo frontend como backend [43].

1.4.16.2. Java

Lenguaje orientado a objetos diseñado para ser portátil y ejecutarse en múltiples plataformas gracias a la Máquina Virtual de Java (JVM). Es comúnmente utilizado en aplicaciones empresariales, desarrollo móvil (especialmente Android) y sistemas embebidos [43].

1.4.16.3. C

Lenguaje de programación de propósito general reconocido por su eficiencia y control a bajo nivel.Este lenguaje encuentra su principal aplicación en la creación de sistemas operativos, software de bajo nivel y programas que exigen máximo rendimiento y control hardware.[43].

1.4.16.4. C++

Extensión del lenguaje C que incorpora características de programación orientada a objetos. Este lenguaje es adecuado para la construcción de motores de videojuegos, software de procesamiento gráfico y sistemas embebidos que requieren respuesta determinista en tiempo real. [43].

1.4.16.5. Python

Lenguaje de alto nivel, interpretado y de propósito general, conocido por su sintaxis clara y legible. Se aplica en diversos campos como desarrollo web, análisis de datos, inteligencia artificial y automatización. Su amplia biblioteca estándar y comunidad activa favorecen el desarrollo rápido de aplicaciones compleja [43].

1.5. Marco contextual

En los últimos años, los avances en la robótica móvil han revolucionado diversos sectores industriales, científicos y sociales, al facilitar la automatización de tareas en entornos complejos, peligrosos o de difícil acceso. En particular, los robots móviles autónomos han asumido un papel crucial en labores de exploración y monitoreo, gracias a su capacidad de desplazarse por su entorno, recopilar información en tiempo real y operar con mínima intervención humana.

Con la creación de los sofisticados sistemas de control para estos robots se requiere el uso de métodos que ayuden a la planificación de optimas trayectorias siendo estas seguras y efectivas, además de recrear representaciones del entorno con la ayuda de sensores como el LiDAR y cámaras. Estas capacidades son muy útiles para desarrollar la navegación de forma autónoma evitando obstáculos del medio y así adaptarse a situaciones cambiantes que se presenten. Este tipo de innovación esta siendo cada ves mas requeridas en los sectores industriales, agricultura y de seguridad.

Esta investigación se desarrolla físicamente en entornos cerrados y controlados dentro de la Universidad Estatal Península de Santa Elena (UPSE), específicamente en laboratorios y espacios interiores donde las condiciones de iluminación, temperatura y disposición del espacio son estables. Este escenario controlado permite validar el sistema en condiciones reales pero optimizadas, eliminando variables externas como condiciones climáticas extremas, iluminación solar variable o terrenos irregulares que podrían afectar el desempeño inicial del sistema de percepción y navegación.

El desarrollo de este proyecto beneficiará principalmente a tres grupos clave. Para la comunidad académica de la UPSE, representa un fortalecimiento de las capacidades de investigación en robótica móvil y sienta las bases para proyectos futuros, estableciendo un punto de referencia en el desarrollo tecnológico institucional. Para la industria local, demuestra el potencial de soluciones robóticas autónomas de bajo costo aplicables a sectores como el

agroindustrial, logística y monitoreo, ofreciendo un caso de estudio relevante para las necesidades reales de la región. Asimismo, para los estudiantes de ingeniería, constituye un caso de estudio integral para el aprendizaje de sistemas integrados, percepción robótica y navegación autónoma, enriqueciendo su formación profesional con tecnologías de innovadoras.

En el ámbito académico y de investigación, el diseño de sistemas autónomos de navegación inteligente se ha consolidado como un campo prioritario dentro de la robótica y la automatización. Asimismo, la integración de sensores avanzados y el procesamiento de datos en tiempo real impulsa el desarrollo de soluciones más robustas, eficaces y aplicables a necesidades reales, contribuyendo así al desarrollo de este campo desde el ámbito universitario ecuatoriano.

2. Métodos y diseño experimental

2.1. Métodos

La elaboración de este documento se basa en una investigación aplicada, el cual busca resolver problemas reales y desarrollar una nueva tecnología a partir del uso de tecnologías existentes. Para ello, se ejecutarán evaluaciones de factibilidad técnica y operacional para definir las estrategias de implementación de robots en misiones de navegación automatizada y vigilancia del entorno.

Además, se trata de un estudio de campo, por lo que se recogerán datos mediante pruebas prácticas combinadas con un estudio documental que permita revisar información clave sobre el movimiento del robot. Esto incluirá la consulta de artículos académicos, disertaciones y textos especializados para comprender mejor los sistemas autónomos de control de navegación.

Sin embargo, este trabajo se clasifica como una investigación de forma experimental, debido que se llevaran acabo varias pruebas para evaluar que el funcionamiento del sistema de control sea adecuado y pueda satisfacer los requerimientos establecidos. Este estudio proporcionara las bases esenciales

para la creación del sistema de control que esta enfocado en la planificación de rutas y desarrollo de mapas del entorno. Este proyecto se organizará en sucesivas etapas las cuales estarán descritas a continuación generando un enfoque de forma completa y organizada.

Fase 1. Revisión Bibliográfica.
En un primer momento se realizará un estudio detallado de los sistemas de control de la planificación de trayectorias y la recreación del entorno. Esto incluirá la consulta de artículos académicos, revistas especializadas, tesis y fuentes en línea que ayuden en el desarrollo de un sistema de planificación de trayectorias y control de generación ambiental y las diversas funciones que desempeñan.

Fase 2. Selección del tipo de Robot Móvil.
En esta etapa se seleccionará un robot terrestre que cumpla con los requerimientos técnicos del proyecto. Se tomarán en cuenta aspectos como su capacidad de movimiento, duración de batería, sensores, actuadores, tamaño y costo.

Fase 3. Selección de Componentes Electrónicos y Eléctricos adicionales.
En este apartado se definirán los dispositivos que formarán parte del sistema, como sensores, actuadores y baterías. La selección se hará considerando factores clave como resistencia, peso, durabilidad y compatibilidad con el resto de los componentes, asegurando así un funcionamiento óptimo del robot en sus tareas de exploración y supervisión.

Fase 4. Desarrollo del Algoritmo de Control y Planificación de Trayectorias.
Una vez definidos los componentes, se programará el sistema de navegación del robot. Se implementarán algoritmos de planificación de trayectorias y técnicas como SLAM para que el robot pueda moverse de forma autónoma dentro de un entorno específico de un punto A, a un punto B, evitando obstáculos y creando mapas en tiempo real.

Fase 5. Pruebas y Ajustes del Sistema.
Finalmente, se realizarán experimentos en diferentes escenarios para probar el rendimiento del robot. Probará si puedes identificar correctamente los obstáculos, generar mapas con un bajo margen de error y seguir rutas planificadas. En esta etapa pueden surgir problemas, por lo que se realizarán los ajustes necesarios para optimizar su funcionamiento y asegurar su correcto funcionamiento en tareas de investigación y seguimiento. Con este enfoque, el proyecto conseguirá que el robot sea capaz de moverse de forma autónoma, reconocer su entorno y alcanzar los objetivos marcados.

2.1.1. Descripción del proyecto

El desarrollo de sistemas autónomos ha avanzado significativamente en los últimos años, impulsado por la creciente necesidad de soluciones inteligentes para la exploración y supervisión en entornos industriales, urbanos y naturales. En este trabajo se propone el desarrollo de un sistema de control para la planificación de trayectoria y la generación de entornos en un robot móvil, con el objetivo de optimizar su autonomía y eficiencia en tareas de exploración y supervisión.

El sistema propuesto sigue un diseño modular ilustrada en la Fig. 23, donde una estación de control (computador con interfaz VNC) se comunica mediante Wi-Fi con la unidad de procesamiento (Raspberry Pi 5). Esta última ejecuta los algoritmos centrales de SLAM (Localización y Mapeo Simultáneo), planificación de trayectorias y generación de mapas en tiempo real, mientras gestiona los sensores (LiDAR y cámara) y actuadores (motores) del robot móvil.

El desarrollo del sistema dará lugar a que el robot móvil se desplace de forma independiente evitando obstáculos y siguiendo una trayectoria que se ha definido previamente con la ayuda de la información que proporcionaran los sensores.

Para esto se utilizarán algoritmos de recreación de entorno y navegación los cuales dependerán del sensor LiDAR, este facilitara los datos para recons-

truir en tiempo real el entorno donde se está ejecutando la tarea. Con estas pruebas se evaluará el rendimiento del sistema en diversas situaciones para verificar así la capacidad de adaptación, exactitud y eficiencia que tiene el robot ante actividades de investigación y monitoreo.

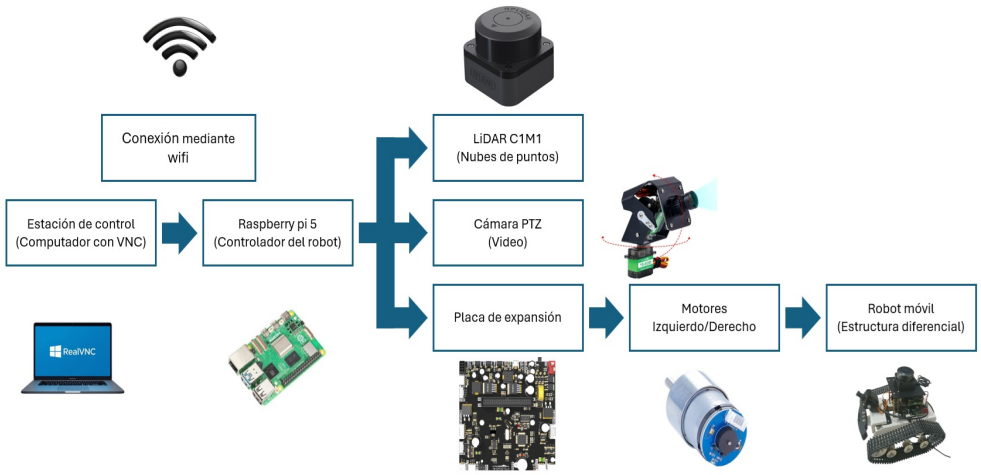


Figura 23: Diagrama general del sistema [Fuente: Autor].

2.2. Componentes de la propuesta

En esta sección se describen los componentes físicos y lógicos utilizados para la implementación del sistema de control, detallando sus características técnicas y funcionalidades específicas dentro de la arquitectura del proyecto. El despliegue comprende tanto elementos de hardware como de software.

2.2.1. Componentes físicos

La plataforma hardware constituye la base física del sistema autónomo. A continuación se describen los elementos electrónicos seleccionados para el desarrollo del robot móvil.

2.2.1.1. Robot móvil Transbot-SE

El Transbot-SE es un robot móvil con locomoción basada en orugas (ver en la Fig. 24). La configuración hardware incluye un brazo robótico de 3 GDL y un sistema visual PTZ de 2 GDL, proporcionando capacidades de navegación estable en múltiples tipos de terrenos y escenarios operativos.

Gracias a su configuración, el robot resulta adecuado para aplicaciones que requieren alta maniobrabilidad y resistencia, siendo utilizado principalmente en investigación y educación, aunque también puede adaptarse a entornos industriales o de exploración complejos [44].

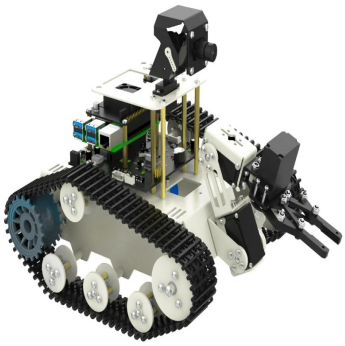


Figura 24: Robot móvil Transbot-SE [44].

La compatibilidad con la placa Raspberry Pi 5 lo convierte en una plataforma ideal para el desarrollo de soluciones avanzadas en inteligencia artificial y robótica autónoma [44]. En este proyecto, el Transbot-SE sirve como plataforma física para la implementación del sistema de navegación autónoma, alojando los sensores, actuadores y unidad de procesamiento.

La Tabla 3 resume las principales características técnicas del robot:

Tabla 3: Características técnicas del Transbot-SE [44].

Parámetro	**Especificación**
Control Principal	Raspberry Pi 5
Sistema operativo	Raspberry Pi OS
Lenguaje de programación	Python / C
Duración de batería	3 horas
Esquema de energía	12.6V, 4.400 mAh
Comunicación	Wi-Fi (LAN/AP)
Interfaz de control	APP móvil, PC
Peso	2.1 kg

2.2.1.2. Sensor LiDAR RPLIDAR C1M1

El RPLIDAR C1M1 representa un sistema de escaneo láser 2D de última generación que proporciona cobertura angular completa de 360°. Su arquitectura permite adquirir hasta 5000 muestras por segundo mediante un mecanismo de rotación de alta velocidad. Incorpora un sistema de transmisión inalámbrica y datos que garantiza operación continua y estable en misiones de larga duración.[45].

Figura 25: Sensor LiDAR RPLIDAR C1M1 [45].

El RPLIDAR C1 tiene una distancia de medición de hasta 12 metros de radio y un rango ciego bajo de solo 0,05 metros. Realiza fácilmente el escaneo y la medición de objetos a distintas distancias y permite la evitación de obstáculos. El RPLIDAR C1 no solo ofrece un rendimiento potente, sino que también presenta un diseño compacto y ágil (ver en la Fig. 25). Es pequeño, con bajos niveles de ruido y vibración, lo que facilita su integración

en diversas aplicaciones [45].

Las características técnicas se evidencian en la tabla 4.

Tabla 4: Especificaciones del sensor RPLIDAR C1M1 [45].

Rango de distancia	Objeto blanco: 0.05-12 metros (con reflexión <70 %) Objeto negro: 0.05-6 metros (con reflexión <10 %)
Tasa de muestreo	5 kHz
Frecuencia de escaneo	8–12 Hz, 10 Hz @ típico
Resolución angular	0.72° @ típico
Planitud del campo de escaneo	0°–1.5° (personalizable)
Interfaz de comunicación	TTL UART
Velocidad de comunicación	460800 bps
Precisión	±30 mm
Resolución	15 mm
Grado de protección	IP54
Límite de luz ambiental	40,000 lux

2.2.1.3. Placa de expansión Transbot

La placa de expansión del Transbot es un componente adicional diseñado para ampliar las capacidades del robot, ya que proporciona puertos y conexiones extra que permiten la integración de diversos sensores, actuadores y módulos (ver en la Fig. 26) Gracias a ello, es posible conectar y gestionar dispositivos externos como cámaras, sensores ultrasónicos, motores adicionales y servomotores, lo que otorga una mayor flexibilidad y adaptabilidad en el desarrollo de proyectos avanzados en robótica [46].

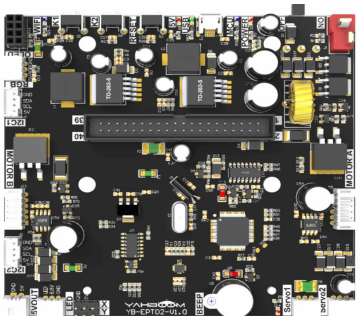


Figura 26: Placa de expansión Transbot [46].

Este módulo incorpora interfaces de comunicación como I2C, UART, SPI y GPIO, que facilitan su integración con plataformas de desarrollo como Raspberry Pi o Jetson Nano. Además, es compatible con sistemas de control y programación como ROS (Robot Operating System), lo que posibilita la ejecución de funciones complejas, incluyendo navegación autónoma, manipulación de objetos y procesamiento de imágenes [46].

A continuación, se presenta la tabla 5 que resume sus principales especificaciones técnicas, destacando las características más relevantes de esta placa de expansión.

Tabla 5: Características técnicas de la placa de expansión Transbot [46].

Potencia de salida	5V DC
Chip de comunicación	Chip serie CH340C
Fuente de alimentación	12V DC
Interfaz de comunicación	Micro USB a USB

2.2.1.4. Raspberry Pi 5

La Raspberry Pi 5 es un ordenador de placa única compacto y asequible, diseñado para facilitar el aprendizaje de informática y programación (ver en la Fig. 27). Funciona como un equipo completo, permitiendo navegar por la web, reproducir vídeos, redactar documentos, desarrollar software y jugar. Asimismo, esta brinda la posibilidad de programar por medio de los pines GPIO los cuales son compatibles con la comunicación en serie que maneja,

las cuales son SPI y I2C. estas cualidades que tiene la convierten en la opción mas convenientes para realizar proyectos dentro del ámbito de la electrónica y la robótica las cuales necesitan varias conexiones con actuadores y sensores. Adicional, resulta muy útil en el albito donde se necesite el procesamiento de imágenes y videos, además de complejos cálculos matemáticos.

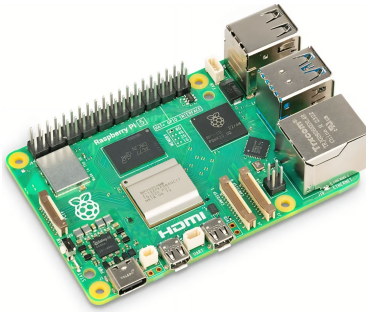


Figura 27: Rasberry Pi 5 [47].

En la tabla 6 se muestran las especificaciones de la Raspberry Pi 5, destacando sus componentes principales.

Tabla 6: Características de la unidad de procesamiento Raspberry Pi 5 [47].

Componente	**Especificación**
Plataforma	Raspberry Pi 5 - 4GB
CPU	Quad-core ARM Cortex-A76 64-bit
RAM	4GB LPDDR4X-4267 SDRAM
Comunicaciones	Wi-Fi dual-band 802.11ac, Bluetooth 5.0
Salidas de video	Dos puertos micro-HDMI
Interfaz USB	Cuatro puertos (2× USB 3.0, 2× USB 2.0)
Fuente de poder	USB-C 5.1V/5A (25.5W)
E/S digitales	Conector GPIO 40-pines (UART, I2C, SPI)

2.2.1.5. Motores con codificadores

Los motores de corriente continua con codificadores integrados son ampliamente utilizados en robótica debido a su capacidad para proporcionar información precisa sobre la posición y la velocidad del eje (ver en la Fig. 28). Esta funcionalidad posibilita la implementación de esquemas de control por retroalimentación, donde la velocidad y dirección del motor se regulan de forma adaptativa según la información proporcionada por el sistema. Esta capacidad permite establecer sistemas de control por retroalimentación de velocidad, donde la RPM del motor se regula de forma adaptativa según los pulsos proporcionados por el encoder rotativo.[48].

Figura 28: Motores 520 con codificador. [48].

A continuación, se presenta en la tabla 7 las características técnicas que tienen los motores con codificadores, destacando así sus atributos más relevantes.

Tabla 7: Características técnicas de los motores 310 [48].

Modelo	JGA27-310R20
Voltaje nominal	7.4 V
Corriente nominal	$\leq$ 0.65 A
Potencia nominal	4.8 W
Torque nominal	0.4 kg·cm
Tipo de codificador	Codificador incremental Hall de fase AB
Peso aproximado	70 g

2.2.1.6. Cámara Ptz 2DOF

Una cámara PTZ 2DOF (Pan-Tilt-Zoom) dispone de dos grados de libertad que le permiten rotar entre 0 y 180° en los ejes horizontal y vertical. Incorpora una cámara USB de alta definición sin bloque, con un amplio campo de visión de 120°, una tasa de 120 PFS y una resolución de 2 megapíxeles (ver en la Fig. 29). Su diseño compacto facilita su integración en espacios reducidos y permite una visualización flexible y dinámica de áreas amplias sin necesidad de ajustes manuales. Estas capacidades la convierten en una solución óptima para sistemas de vigilancia y plataformas robóticas, dado que puede reorientar autónomamente su campo visual para realizar seguimiento continuo de objetivos móviles.[47].

Figura 29: Cámara Ptz 2DOF [47].

En la tabla 8 se caracterizan los parámetros técnicos del la cámara PTZ, indicando sus principales componentes y capacidades.

Tabla 8: Configuración técnica del sistema de visión PTZ de 2 GDL [47].

Dispositivo	Cámara PTZ 2 GDL (Versión Básica)
Movimientos articulados	2 ejes de rotación independientes
Conexión de datos	Interfaz USB 2.0
Campo visual	Cobertura angular de 180°
Resolución de imagen	Sensor de 2 megapíxeles (1800×1800)
Plataformas soportadas	Integrable con Raspberry Pi, Jetson y sistemas embebidos

2.2.1.7. Batería de litio

Las baterías de Li-ion constituyen sistemas de almacenamiento energético recargable extensamente implementados en dispositivos electrónicos y plataformas robóticas, destacando por su elevada densidad de energía, tamaño reducido y prolongado ciclo de vida operativo.(ver en la Fig. 30). Ofrecen ventajas significativas frente a otras tecnologías, como las baterías de plomo-ácido o níquel-cadmio, gracias a su mayor capacidad de almacenamiento, lo que permite una mayor autonomía operativa. Además, presentan una baja tasa de autodescarga, lo que les permite conservar su carga durante largos períodos sin uso [47].

Figura 30: Batería de Litio 7.4V 2000mAh [47].

A continuación, se evidencian en la tabla 9 las especificaciones técnicas que tiene la batería, destacando sus elementos más relevantes.

Tabla 9: Características de la batería de litio de 7.4 V y 2000 mAh [47].

Voltaje nominal	7.4V
Voltaje de suministro	5.6V 8.4V
Capacidad de la batería	2000mAh
Corriente nominal	15A (7.5C)
Peso de la batería	aproximadamente 115g
Tamaño de la batería	67*37*22mm

2.2.1.8. Sensor ultrasónico HC-SR04

El HC-SR04 es un sensor de distancia que funciona bajo el principio de ecolocalización ultrasónica. Su operación consiste en emitir un tren de pulsos de sonido a una frecuencia de 40 kHz, que es inaudible para el oído humano, y posteriormente detectar el eco reflejado por un objeto. El sensor HC-SR04 opera enviando una breve señal de 10µs a su pin Trigger, lo que desencadena la emisión de un pulso ultrasónico de 40kHz. Su pin Echo se activa entonces y permanece en alto hasta recibir el eco, midiendo así el tiempo de vuelo de la onda (ver en la Fig. 31) [49].

Figura 31: Sensor ultrasónico HC-SR04 [49].

A continuación en la tabla 10, se presentan las principales características del sensor:

Tabla 10: Características del sensor HC-SR04 31) [49]

Rango de medición efectivo	2 cm a 400 cm
Precisión	Resolución de 3 mm
Ángulo de medición	Menor a 15 grados
Tensión de operación	5V DC
Compatibilidad	Arduino, Raspberry Pi y otros microcontroladores

2.2.2. Componentes lógicos

A continuación, se detallan los programas utilizados para la codificación de la parte física de esta investigación. Estos recursos son fundamentales

para el desarrollo y el correcto funcionamiento del sistema, ya que facilitan la organización y el control de las tareas realizadas.

2.2.2.1. Raspberry Pi Imager

Raspberry Pi Imager es una aplicación oficial desarrollada por la Fundación Raspberry Pi que facilita la grabación de una imagen del sistema operativo en una tarjeta SD o unidad USB destinada a una Raspberry Pi. Esta herramienta automatiza el proceso de descarga, formateo e instalación del sistema operativo, simplificando considerablemente la configuración inicial. Con ella, es posible instalar diversos sistemas compatibles con la Raspberry Pi 5. Para este proyecto, se empleó una versión del sistema operativo Raspberry Pi OS que incluye las bibliotecas y requisitos esenciales para el funcionamiento del sistema (ver en la Fig. 32) [50].

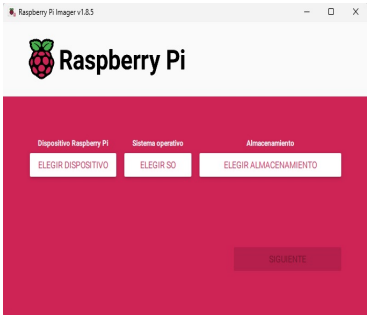


Figura 32: Entorno Raspberry Pi Imager [Fuente: Autor].

2.2.2.2. Advanced Ip Scanner

Esta herramienta de escaneo de red se configura como una solución versátil y de alto rendimiento, específicamente diseñada para optimizar los procesos de descubrimiento y gestión de dispositivos en red. Este software permite a los usuarios examinar, monitorear y obtener información detallada sobre la configuración de su red (ver en la Fig. 38). Resulta fundamental para este proyecto, ya que al conectar la Raspberry Pi 5 a la red, el módem le asigna una dirección IP única necesaria para habilitar el control remoto. Para utilizar el programa, basta con hacer clic en el ícono de IP ubicado en la parte superior,

que realizará un escaneo y listará todos los dispositivos conectados a la red [51].

Figura 33: Entorno de Advanced IP Scanner [51].

2.2.2.3. Vnc Viewer

VNC Viewer es una herramienta que permite el acceso y control remoto de otro dispositivo mediante el protocolo VNC (Computación en Red Virtual). A través de esta aplicación, es posible visualizar el escritorio y operar el equipo remoto como si se estuviera físicamente presente, utilizando cualquier dispositivo con acceso a Internet (ver en la Fig. 34). En este proyecto, se accede a la Raspberry Pi 5 ingresando la dirección IP previamente identificada mediante un programa de escaneo. Al introducir dicha IP en el navegador y presionar Enter, se despliega automáticamente la pantalla principal de la Raspberry Pi, lo que permite realizar las configuraciones iniciales del dispositivo [52].

Figura 34: Entorno de Vnc Viewer [Fuente: Autor].

2.2.2.4. Python

En este proyecto se emplea Python como lenguaje de programación de alto nivel e interpretado, python fue seleccionado para este desarrollo basándose en su arquitectura de codificación intuitiva y fácil comprensión, extenso ecosistema de librerías científicas, y amplia adopción en aplicaciones de robótica e inteligencia artificial. Para el desarrollo de este proyecto se utilizó el editor Thonny, una herramienta ligera optimizada para programación en sistemas embebidos [43].

Las principales bibliotecas de Python utilizadas en la implementación del sistema de control del robot se organizan según su funcionalidad de la siguiente manera:

- **Interfaz gráfica:**

  PyQt5 es una biblioteca de Python que sirve como enlace (binding) integral para el framework Qt, permitiendo el desarrollo de interfaces gráficas de usuario (GUI) multiplataforma. PyQt5 se destaca por su arquitectura integral de widgets predefinidos y contenedores avanzados para la construcción de interfaces, junto con módulos especializados que amplían su funcionalidad a áreas como bases de datos, gráficos e integración web [53].

- **Cómputo matemático:**

  numpy: Biblioteca fundamental para computación numérica que proporciona soporte para arrays multidimensionales y operaciones matemáticas avanzadas. En este proyecto, numpy se utiliza para el procesamiento eficiente de nubes de puntos del LiDAR, transformaciones geométricas para el sistema de navegación, y cálculos matriciales requeridos por los algoritmos de SLAM y planificación de trayectorias [54].

  SciPy: (Scientific Python) es una biblioteca fundamental para el cómputo científico y técnico dentro de Python. Construida sobre la

base de NumPy y su estructura de arreglos multidimensionales, SciPy proporciona un amplio conjunto de módulos especializados para tareas avanzadas de cálculo numérico. Entre sus componentes principales se incluyen rutinas optimizadas para cálculo integral y ecuaciones diferenciales, álgebra lineal, procesamiento de señales, estadística avanzada y optimización matemática [55].

Math: Esta librería constituye un módulo estándar que proporciona acceso a las funciones matemáticas definidas por el estándar del lenguaje C, ofreciendo capacidades computacionales avanzadas para operaciones científicas y de ingeniería. Esta biblioteca incluye una amplia gama de funciones trascendentes, entre las que destacan las funciones trigonométricas (seno, coseno, tangente y sus inversas), funciones hiperbólicas, logaritmos en diferentes bases y funciones exponenciales [56].

- **Visualización de datos:**

  matplotlib: Biblioteca para visualización de datos y generación de gráficos. Aunque no participa en el funcionamiento en tiempo real del robot, es esencial para tareas de depuración y análisis, permitiendo visualizar trayectorias ejecutadas, mapas generados por el algoritmo de SLAM, y datos crudos de sensores para validar el correcto funcionamiento del sistema [57].

- **Procesamiento de visión:**

  OpenCV: es conocida en Python como módulo cv2, esta biblioteca de código abierto es especializada en visión por computadora y procesamiento de imágenes en tiempo real.

- **Comunicación con hardware:**

  pyserial: Biblioteca que gestiona la comunicación mediante puertos seriales (UART, RS-232) de manera consistente across diferentes sistemas operativos. Proporciona una API orientada a objetos para configurar

parámetros de comunicación como velocidad en baudios, bits de parada, paridad y control de flujo. En este trabajo, pyserial facilita la comunicación con periféricos embebidos y sensores que utilizan interfaces seriales para la transmisión de datos [58].

2.3. Diseño Experimental

Este desarrollo del sistema de control se trabajo sobre el robot móvil Transbot SE, la misma que tuvo modificaciones estructurales para así lograr adaptar los nuevos componentes que se usarían en este proyecto. Este robot movil fue elegido debido a su capacidad robusta de movilidad, además de su estructurado material. Aspectos los cuales permiten moverse con mucha eficacia en diversos ambientes. La estructura de este sistema incluye los siguientes elementos esenciales

Se empleó el uso de la Raspberry Pi 5 como unidad de procesamiento central sobre la cual se llevan acabo los algoritmos de navegación, planificación de las rutas y el manejo de los datos de los sensores. En la parte de recreación del espacio que lo rodea, se incluyó el sensor LiDAR, este dispositivo recrea el entorno mediante los escaneos láser, el cual permite esquivar los obstáculos presentes en la operación. Además, este sistema cuenta con una cámara la cual ofrece una retroalimentación visual del entorno, aunque no participa directamente en la toma de decisiones del algoritmo, sirve como apoyo para el operador.

Los datos que son recolectados por el sensor LiDAR son procesados en la unidad de control principal, donde se llevan acabo los algoritmos de control que permiten la movilidad del robot de manera independiente, evitando así choques que comprometan la estructura y los elementos que lo conforman, llevando a cabo así con precisión la trayectoria predefinida en su totalidad.

En las siguientes secciones se detallan los parámetros técnicos que definen la arquitectura del sistema, estructurados en componentes de hardware y software para permitir un análisis minucioso de cada componente

2.3.1. Diseño e implementación del Hardware

El hardware forma la base física que define las capacidades de procesamiento, detección y movilidad del sistema robótico. Para garantizar un rendimiento adecuado en tareas de investigación autónomas, se desarrolló una configuración de hardware específica que integra los componentes de adquisición de datos, unidad de procesamiento y sistema de actuación.

Como se muestra en el diagrama esquemático de la Figura 35, la arquitectura hardware sigue una distribución espacial optimizada para las funciones de percepción y movilidad:

- **Unidad de procesamiento central:** En el núcleo del sistema se encuentra la Raspberry Pi 5, equipada con un ventilador para disipación térmica que garantiza estabilidad operacional durante la ejecución prolongada de los algoritmos de navegación y SLAM.
- **Sistema de detección:** En la parte superior del robot se ubica el sensor LiDAR RPLidar C1M1, el cual nos ofrece escaneos láser de 360° que facilitan la identificación de obstáculos que se encuentran presentes en todas las direcciones y la creación de mapas instantáneamente.
- **Visión artificial complementaria:** En la parte delantera del robot se ubica la cámara PTZ 2DOF, especialmente posicionada para la captura del video en tiempo real del entorno designado y proporcionar una retroalimentación visual para el operador, aunque no interviene directamente en la toma de decisiones del sistema.
- **Sistema de movilidad:** En la base inferior se localizan los motores 520 con encoders, responsables del desplazamiento del robot mediante el control de movimiento diferencial.
- **Gestión de energía:** la batería de litio de 12 V está ubicada estratégicamente para equilibrar el centro de gravedad del robot y al mismo tiempo proporciona energía estable a todos los componentes del sistema.

Figura 35: Esquema del robot Transbot [Fuente: Autor].

En la Figura 36 se ilustran todos los elementos mencionados, posicionados correspondientes dentro del robot. Esta visualización permite observar claramente la organización física de los componentes, ayudando en el entendimiento de su configuración y relaciones de conectividad.

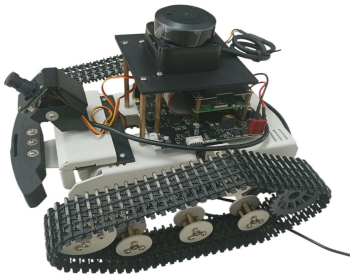


Figura 36: Robot móvil Transbot-SE [Fuente: Autor].

Esta configuración mejorada posibilita que el robot conserve un diseño compacto para incrementar su estabilidad, al mismo tiempo que amplia el rango de visión del LiDAR y garantiza una adecuada repartición del peso para una movilidad eficaz.

2.3.1.1. Conexiones de la placa de expansión

La placa de expansión Transbot SE incorpora un microcontrolador monolítico responsable de gestionar los diversos componentes de la tarjeta, posibilitando que la Raspberry Pi establezca comunicación efectiva con los los dispositivos externos del sistema. Entre estos se incluyen los motores, los servomotores que controlan el movimiento de la cámara PTZ, así como diversos puertos y pines de expansión que facilitan la conexión de sensores y actuadores adicionales sin necesidad de adaptadores externos.

La tarjeta de expansión esta diseñada principalmente para ampliar y optimizar las capacidades de la unidad de control principal, la conexión que existe entre ambas placas se lleva acabo por medio de un cable plano. Esta conexión garantiza una comunicación eficiente en el intercambio de información, además de suministrar energía a los módulos. Gracias a esta interfaz se lleva a cabo una transmisión de señales de forma continua y fluida, generando un control inmediato del robot en las actividades que conllevan la planificación de rutas y el desarrollo del entorno.

En la siguiente figura se muestra la disposición general de los elementos conectados (ver en la Fig. 37), y posteriormente se presenta la tabla **??** con las conexiones entre los diferentes pines de la placa de expansión y la Raspberry Pi, destacando sus funciones principales.

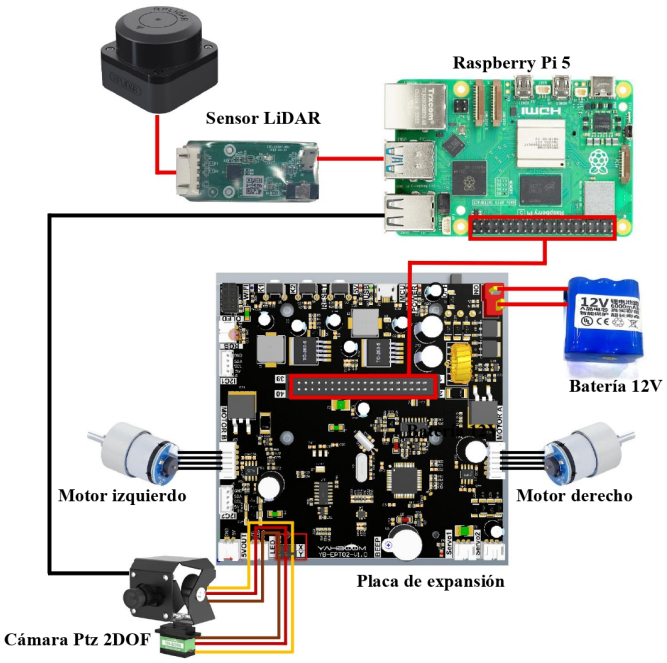


Figura 37: Conexión de los componentes a la placa de expansión [Fuente: Autor].

Tabla 11: Configuración de interconexión de periféricos en la placa de expansión.

Dispositivo	**Interface de conexión**	**Propósito funcional**
Raspberry Pi	Bus GPIO (flat cable)	Intercambio de datos y distribución de potencia eléctrica
Actuador izquierdo	Canal Motor A	Regulación de velocidad y sentido de giro
Actuador derecho	Canal Motor B	Gobierno de velocidad y orientación
Unidad de visión PTZ	Puerto USB	Captura de video y control de orientación
Sensor LIDAR C1M1	Interface USB	Escaneo láser ambiental para cartografía y detección de obstáculos
Sistema de energía	Entrada de potencia	Alimentación eléctrica del conjunto

2.3.2. Diseño e implementación del Software

El software constituye un componente fundamental que otorga la funcionalidad correcta al hardware, al posibilitar la ejecución de tareas e interacciones específicas entre los distintos módulos del sistema.

A través de los sistemas de monitoreo en tiempo real, el programa gestiona el manejo de los datos, sincroniza la interacción entre los dispositivos y decide los movimientos en el momento, dando así una operación del sistema coordinada y precisa.

2.3.2.1. Librería Transbot

Este conjunto de herramientas se ha creado para administrar el hardware del robot Transbot-se, ofreciendo los recursos necesarios para manejar el desplazamiento, lo que facilita una conexión eficaz y continua entre el software y el hardware del sistema.

Para la instalación de la librería Transbot, indispensable para el correcto funcionamiento del robot, es necesario crear previamente un entorno virtual en el que se almacenarán todas las dependencias requeridas. Una vez configurado el entorno, se descarga el archivo correspondiente desde la página oficial de Yahboom, se descomprime y se procede con su instalación.

2.3.2.2. Librería RPLidarExpress

La librería del sensor LiDAR RPLidar C1M1-R2 permite su integración en aplicaciones de robótica, navegación y mapeo en tiempo real mediante comunicación serial UART (3.3V-TTL). El SDK oficial desarrollado por Slamtec facilita la conexión del sensor a una computadora para la visualización gráfica de los puntos de escaneo y la adquisición de datos crudos, los cuales pueden emplearse en aplicaciones personalizadas dentro de entornos como ROS u otras plataformas.

La información se envía mediante una conexión UART a una tasa común

de 460800 baudios y con el formato 8N1, asegurando una transferencia de datos sólida y efectiva.

2.3.3. Implementación del sistema de control

La creación del sistema de gestión para el robot móvil Transbot-SE implica diseñar una base que posibilite la organización de rutas y la creación del entorno instantáneamente. Este sistema garantiza que el robot funcione de forma independiente y segura, ofreciendo dos modalidades de funcionamiento para diversas situaciones de exploración y vigilancia.

2.3.3.1. Diagrama de flujo del sistema de control

El sistema de control permite al robot adquirir los datos del entorno mediante el sensor LiDAR, estos datos son procesados para generar el mapa en tiempo real y planificar así la trayectoria. El funcionamiento detallado de este proceso puede observarse en el siguiente diagrama de flujo (ver en la Fig. 38).

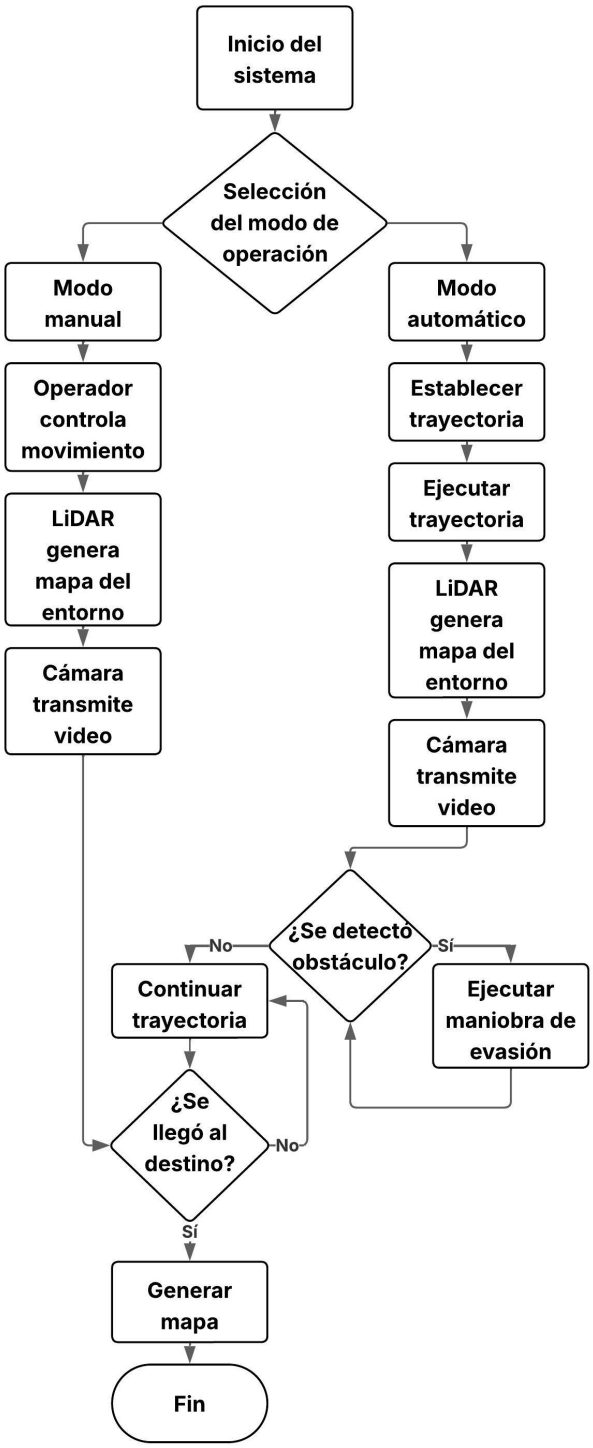


Figura 38: Diagrama del sistema de control [Fuente: Autor].

El sistema da inicio solicitando al usuario seleccionar entre las dos modalidades de operación: manual o automático. Durante la operación manual, el operador ejerce control completo sobre la locomoción del robot, dirigiendo su desplazamiento mientras el sistema adquiere los datos del entorno mediante el sensor LiDAR para su posterior procesamiento y generación del mapa. Mientras tanto, la cámara transmite video durante la operación.

En el modo automático, el robot funciona sin intervención humana: inicialmente se determina un camino a seguir, después se lleva a cabo el recorrido programado mientras el LiDAR recolecta información para crear el mapa y la cámara envía video. Durante la navegación independiente, si el robot identifica un obstáculo, realiza una maniobra de esquiva y prosigue hacia su objetivo.

El sistema mantiene constantemente la adquisición de datos y la transmisión de video en ambos modos de operación, garantizando una supervisión completa del proceso de exploración.

2.3.3.2. Plataforma de interacción Usuario - Sistema

Para el desarrollo de la plataforma de interacción, se empleó la biblioteca de Python Tkinter, especializada en la creación de interfaces gráficas. Esta herramienta nos ofrece los elementos necesarios para realizar ventanas, botones, campos de texto, menús desplegables y etiquetas, permitiendo crear entornos visuales intuitivos.

El sistema cuenta con un entorno visual principal donde se puede elegir entre dos modalidades de funcionamiento: manual y automático (ver en la Fig. 39). La elección depende de los requerimientos del operador según las necesidades específicas de cada tarea.

Figura 39: Menú principal del sistema de control [Fuente: Autor].

Modo manual: Al seleccionar el modo manual, el usuario accede a una interfaz dividida en seis módulos o secciones (ver en la Fig. 40) las cuales se detallan a continuación:

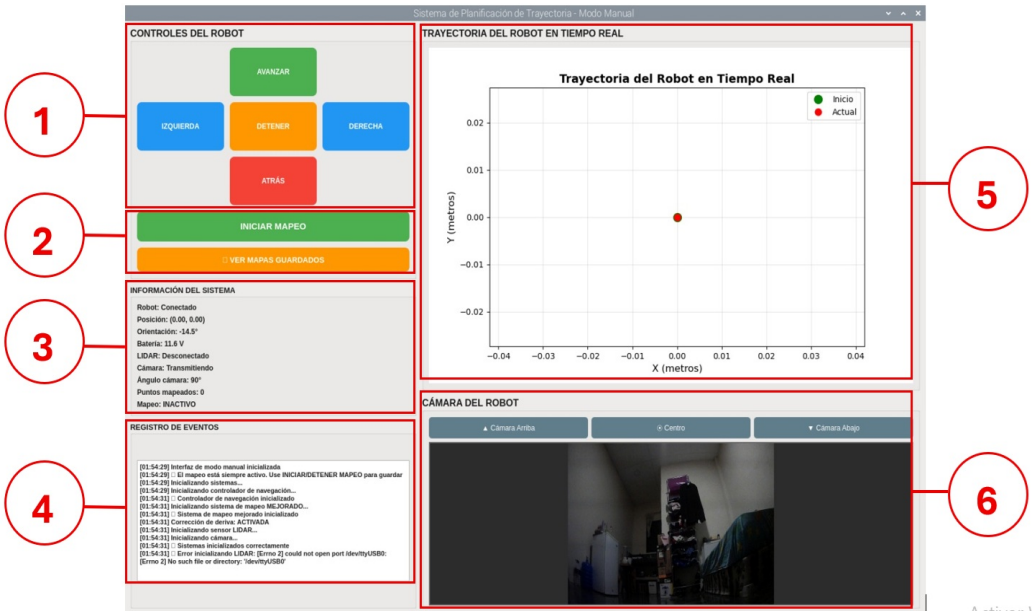


Figura 40: Modo manual del sistema de control [Fuente: Autor].

1. Control de Movimiento: Contiene los comandos direccionales del robot: avance progresivo, retroceder, girar tanto a la izquierda como a la derecha y detener la marcha del robot. Cada acción se ejecuta por medio de

botones que envían instrucciones en tiempo real al sistema de actuación.

2. Gestión de Mapas: Esta sección da la orden de activación del sensor LIDAR para llevar acabo el mapeo y la consulta de los mapas guardados.

Además, cuenta con dos funciones principales:

- **Iniciar Mapeo:** Se da inicio al funcionamiento del sensor LiDAR para comenzar a adquirir el mapa del entorno. Al finalizar la operación del robot, se detiene el sensor LiDAR y como siguiente el mapeo, almacenando el mapa..
- **Ver Mapas Guardados:** Esta opción permite visualizar los mapas que han sido generados anteriormente.

3. Información del Sistema: Presenta las métricas operativas del sistema en tiempo real:

- Estado de conexión del robot
- Posición actual (coordenadas X, Y) y orientación del robot
- Nivel de carga de la batería en porcentaje
- Estado de los sensores: indica si el sensor LIDAR está activado o desactivado y la cámara está transmitiendo o se encuentra inactiva
- Ángulo actual de la cámara PTZ
- se muestra el número de puntos mapeados y un indicador del estado del proceso de mapeo

4. Registro de Eventos: Funciona como un historial temporal de las acciones del robot.

5. Trayectoria del robot en tiempo real: Se visualiza gráficamente el desplazamiento del robot, actualizándose dinámicamente según los movimientos ejecutados por el operador a través del panel de control.

6. Cámara del robot: Se muestra la transmisión del video en tiempo real capturado por la cámara PTZ, con la capacidad de ajustar su orientación mediante controles de movimiento vertical (arriba, centro, abajo).

Modo automático: Al seleccionar el modo automático, el operador accede a una interfaz especializada para planificación y ejecución autónoma de trayectoria (ver en la Fig. 41), organizada en seis módulos principales:

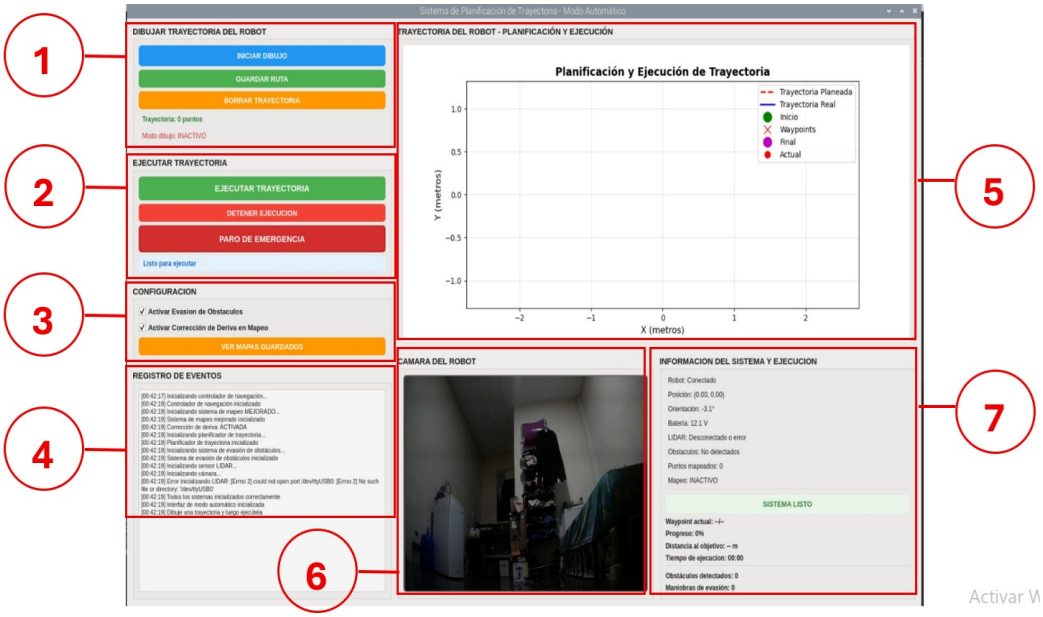


Figura 41: Modo automático del sistema de control [Fuente: Autor].

1. Dibujar Trayectoria del robot: Incorpora tres funciones esenciales del sistema como lo es:

- **Iniciar Dibujo:** Activa el modo para dibujar la ruta sobre el lienzo.
- **Guardar Ruta:** Este botón almacena la trayectoria que fue diseñada por el operador para su posterior ejecución.
- **Borrar Trayectoria:** Elimina el diseño actual del lienzo, permitiendo reiniciar la planificación.

Incluye además indicadores del número de puntos registrados y estado de activación del sistema.

2. Ejecutar Trayectoria: Gestiona la operación autónoma del robot mediante los siguientes controles:

- **Ejecutar Trayectoria:** Inicia la navegación autónoma de la trayectoria establecida.
- **Detener Ejecución:** Pausa la ejecución de la trayectoria.
- **Paro de Emergencia:** Detiene inmediatamente todas las operaciones que se esten llevando a cabo.

Incluye indicador de estado "Listo para ejecutar".

3. Panel de Configuración: Opciones avanzadas para personalizar el comportamiento autónomo:

- **Evasíón de Obstáculos:** Activa/desactiva el sistema de detección y evasión de obstáculos del sistema.
- **Corrección de Deriva:** Esta opción hace que los errores que se han acumulado durante el mapeo sean corregidos.
- **Ver Mapas Guardados:** Permite acceder al apartado del historial, donde los mapas han sido almacenados en ejecuciones anteriores.

4. Registro de Eventos: esta sección funciona como un historial temporal, donde se registran todas las acciones que ha ejecutado el sistema.

5. Planificación y Ejecución de Trayectoria: Facilita la representación de la trayectoria a través de un plano cartesiano con mediciones en metros. El sistema aplica un formato de etiquetado en orden para realizar el monitoreo. Con los posteriores parámetros:

- **Punto Verde:** Establece el punto inicial de la trayectoria, primer clic.
- **Punto Morado:** Se inserta al dar un sgundo clic - define el punto de destino a donde va a llegar el robot.

- **Puntos Intermedios:** son los puntos donde se une la anterior trayectoria con una nueva
- **Indicador de Posición:** El punto rojo muestra la ubicación actual del robot en el plano

La trayectoria planificada se representa con líneas segmentadas de color rojo en el plano, y durante la ejecución aparece una línea azul continua que representa la trayectoria real que a realizado el robot.

6. Cámara del robot: Permite visualizar en tiempo real del entorno mediante la cámara del robot, mantenida en una posición fija durante el modo automático.

7. Información del sistema y ejecución: Provee información del sistema en tiempo real:

- Estado de conexión y posición actual (X, Y, orientación)
- Nivel de batería y estado del LIDAR
- Detección de obstáculos y conteo de evitaciones
- Progreso de trayectoria y distancia al objetivo
- Tiempo de ejecución y puntos mapeados
- Waypoints actuales y estado general del sistema

2.3.4. Lógica del sistema de Mapeo y Reconstrucción del entorno

El sistema de mapeo implementado constituye el sentido de la vista del robot, permitiéndole comprender y registrar el entorno de manera autónoma durante su desplazamiento. Este proceso se encarga de transformar las mediciones crudas del sensor LiDAR en una representación espacial del entorno. El proceso de SLAM mediante LIDAR puede entenderse conceptualmente como un ciclo de percepción y corrección (ver en la Fig. 42). El robot escanea su entorno, procesa estos datos para construir un mapa trazando sus

elementos principales y, de manera crucial, corrige los errores acumulados en su trayectoria antes de almacenar el mapa definitivo. A continuación, se detallan las etapas técnicas específicas de este proceso.

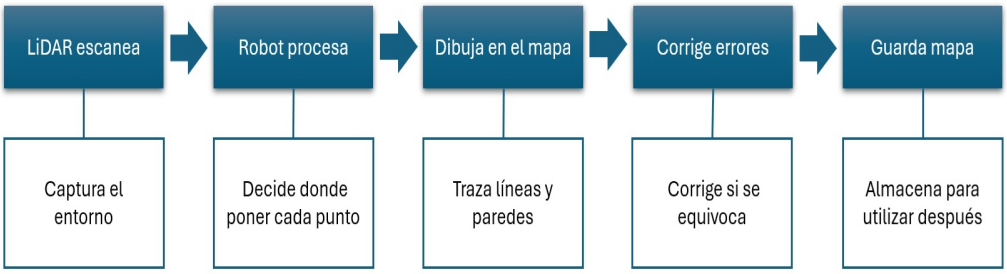


Figura 42: Diagrama general del sistema de mapeo [Fuente: Autor].

2.3.4.1. Algoritmo de Construcción del Mapa

La generación de mapas inicia con la captura continua de datos por parte del sensor LIDAR, que actúa como los ojos del sistema al escanear el entorno y medir distancias hacia todos los objetos que se encuentran a su alrededor. Cuando se lleva acabo un escaneo, se genera alrededor de 1000 y 5000 datos que son evaluados de forma severa quitando lecturas las cuales son demasiadas cercanas a una medida de menos 10 centímetros o en su caso demasiadas distantes a mas de 8 metros.

Los puntos válidos son procesados individualmente para determinar su posición exacta en el espacio, aplicando una rotación de 230° específica de la configuración mecánica del robot Transbot antes de convertirlos a coordenadas globales utilizando la ubicación y orientación actual del robot como referencia.

El núcleo del algoritmo reside en el procesamiento de cada punto LIDAR mediante el método de trazado de rayos. Para cada medición válida, el sistema convierte las coordenadas polares (θ, d) a coordenadas cartesianas relativas al robot (x_{sensor}, y_{sensor}), y luego las transforma a coordenadas

globales (x_{global}, y_{global}) considerando la pose actual del robot.

Como siguiente, se aplica el algoritmo de Bresenham para llevar acabo el trazo una línea virtual, que va desde la posición del robot (x_0, y_0) hasta el punto detectado (x_1, y_1). Todas las celdas que forman parte de esta trayectoria son marcadas como espacios libres mediante decrementos probabilísticos (FREE_INC), mientras que la celda final donde impacta el láser se registra como obstáculo mediante incrementos (OCCUPIED_INC).

Este procedimiento es llevado acabo de manera continua para cada uno de los escaneos, incorporando datos nuevos cada vez que el robot realiza un desplazamiento realizando así un refinamiento gradual del mapa.

2.3.4.2. Algoritmo de corrección y refinamiento

Para llevar a cabo la correcta recreación del entorno por un largo trayecto que ha realizado el robot en las tareas, se incorpora un mecanismo que ajusta la compensación de la deriva acumulada. El modulo que detecta el cierre de ciclo revisa de manera constante las diversas coincidencias entre los nuevos escaneos y los escaneos anteriores. Guardando así un registro de los últimos cien escaneos completados. Cuando el robot pasa o regresa a un área antes mapeada, este sistema lo que realiza es una comparación de los nuevos datos con los anteriores. Si se llega a detectar una coincidencia que supere el setenta porciento de similitud entre escaneos, se reconoce automáticamente un cierre de ciclo y empieza el procedimiento de ajuste de los datos.

Este mecanismo de refinamiento opera sobre un grafo de poses donde cada nodo representa una posición histórica del robot y las aristas encapsulan las relaciones de transformación entre ellas. Al detectar un cierre de ciclo, el sistema genera restricciones de corrección que se propagan a través de todo el grafo de poses, realineando las secciones afectadas del mapa y compensando los errores acumulativos de odometría. Este proceso asegura que el mapa mantenga su consistencia geométrica incluso durante sesiones extendidas de mapeo, enderezando paredes torcidas y corrigiendo distorsiones progresivas.

2.3.5. Lógica de planificación de trayectoria

Este sistema implementado usa una técnica de planificación mixta que incorpora enfoques globales y locales para realizar la navegación autónoma. Esta configuración junta las rutas predefinidas con una respuesta rápida a cambios en el entorno. El método dual incluye un módulo de planificación global el cual determina la mejor ruta con el uso de waypoint, además de un sistema de local de control que ayuda a la modificación de la ejecución del robot en tiempo real de acuerdo con las condiciones cambiantes. Con esta combinación se garantiza la capacidad del robot a adaptarse a situaciones inesperadas durante las ejecuciones de las tareas.

Ahora se describen los elementos del sistema que ayudan a su eficiente funcionamiento.

2.3.5.1. Algoritmo de generación de trayectorias mediante interpolación

La creación de trayectorias utiliza funciones de interpolación cúbica para producir caminos suaves que conectan los puntos de referencia establecidos por el operador. El procedimiento inicia con el cálculo de distancias acumulativas entre waypoints consecutivos, estableciendo un parámetro de progresión a lo largo de la ruta. Sobre este parámetro se construyen dos funciones de interpolación independientes para las coordenadas cartesianas X e Y. Esta metodología garantiza continuidad en las derivadas de la trayectoria, traduciéndose en transiciones suaves entre segmentos.

2.3.5.2. Algoritmo de mecanismo de seguimiento con punto de referencia

El seguimiento de trayectoria implementa el principio de "persecución pura"(Pure Pursuit), calculando continuamente un punto virtual localizado a distancia constante sobre la ruta planificada. La orientación del robot se ajusta permanentemente hacia este punto de referencia, generando un

comportamiento de seguimiento estable y progresivo. La distancia de anticipación, configurada en treinta centímetros, opera como factor de suavizado: valores mayores producen trayectorias más progresivas con menor precisión instantánea, mientras valores menores mejoran la exactitud con movimientos más bruscos.

2.3.5.3. Algoritmo de modulación adaptativa de velocidad

El sistema incorpora un esquema de control de velocidad inteligente que modula dinámicamente la rapidez de desplazamiento según características geométricas de la trayectoria. En segmentos rectos, el robot opera a su velocidad máxima configurada en treinta centímetros por segundo, mientras en curvas pronunciadas reduce gradualmente su velocidad para mantener la estabilidad. Esta adaptación se calcula en tiempo real mediante evaluación continua de curvatura entre puntos sucesivos de la ruta interpolada. Para el waypoint final, el sistema aplica tolerancias de posición más estrictas de cinco centímetros y reduce la velocidad mínima a cuatro centímetros por segundo, asegurando así que el robot llegue al destino establecido.

2.3.5.4. Algoritmo de estrategia de navegación secuencial

La ejecución de trayectorias sigue una metodología secuencial donde el robot se desplaza hacia waypoints individuales de forma consecutiva. El sistema distingue entre waypoints intermedios y punto destino final, implementando estrategias de navegación diferenciadas. Para cada waypoint, el cálculo de control considera dos variables de error principales: error de posición (distancia lineal al objetivo) y error de orientación (discrepancia angular entre dirección actual y dirección requerida). El controlador emplea una ganancia proporcional de 2.0 para correcciones angulares, proporcionando equilibrio entre capacidad de respuesta y estabilidad operativa. Cada ciclo de control se ejecuta a 10 Hz, asegurando capacidad de reacción ante variaciones ambientales.

2.3.5.5. Algoritmo de integración con cecanismos de evasión

El proceso de planificación de la trayectoria opera en una relación estrecha con el sistema que evita los obstáculos. Mientras el robot realiza la tarea de seguir la trayectoria establecida, analiza contantemente los datos del sensor LiDAR para así identificar obstáculos que se presenten en el momento de la trayectoria. El sistema de evasión se activa cuando se detecte un obstáculo, reajustando la ruta inicial establecida hasta que el obstáculo haya sido superado, una vez llevado acabo el robot sigue su trayectoria hasta finalizarla. Con este enfoque se refuerza la planificación global, ofreciendo así una solidez ante entornos cambiantes.

2.3.5.6. Algoritmo de gestión de estados y transiciones

El sistema implementa una máquina de estados bien definida que gestiona las transiciones entre los diferentes modos operativos. Los estados principales incluyen (DIBUJANDO) para la definición de la trayectoria, (GUARDADO) cuando la ruta está almacenada y lista para ejecución, (EJECUTANDO) durante la navegación activa, (PAUSADO) para interrupciones temporales, (FINALIZADO) al completar la misión, y (EMERGENCIA) para situaciones críticas (ver en la Fig. 43).

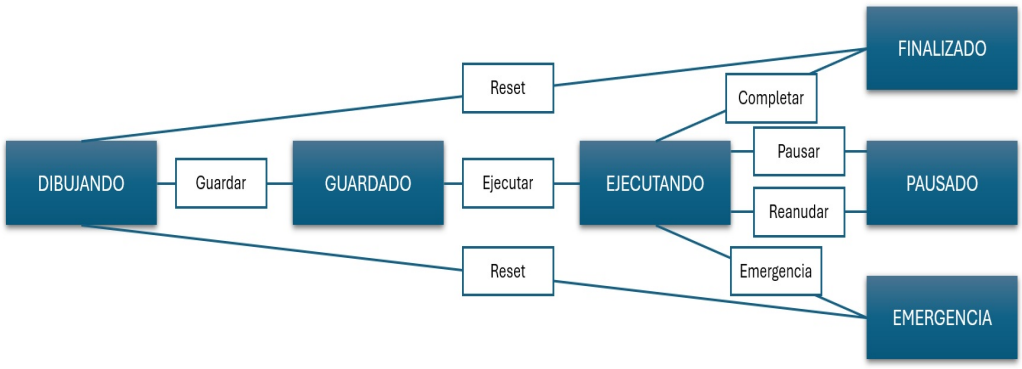


Figura 43: Diagrama de transición de estados del sistema de planificación [Fuente: Autor].

Cada transición de estado activa comportamientos específicos y actualiza la interfaz de usuario correspondientemente. El estado de emergencia tiene la máxima prioridad, deteniendo inmediatamente todos los movimientos del robot cuando se activa, ya sea por detección de obstáculo crítico o por intervención del usuario.

2.3.5.7. Algoritmo de validación de viabilidad y seguridad

Previo a la ejecución de trayectorias, el sistema realiza validación de viabilidad para verificar factibilidad física de la ruta propuesta. Esta validación incluye análisis de curvatura máxima, rechazando trayectorias que requieran giros que vayan más allá de las capacidades físicas del chasis del robot. La evaluación de la curvatura se enfoca en el método del circulo de curvatura, analizando así un conjunto de tres puntos seguidos para calcular los radios de los giros necesarios. Además, el sistema de algoritmo toma en cuenta las dimensiones reales del robot asegurando limites durante la planificación, teniendo como resultado evitar choques con los objetos del entorno.

2.3.5.8. Métricas y optimización de desempeño

Este sistema incluye los indicadores para evaluar el rendimiento de la navegación del robot. Los cuales abarcan el tiempo que toma la operación, además del porcentaje de los puntos alcanzados, el número de obstáculos que se presentaron y la distancia que se recorrido en su totalidad.

3. Resultados

3.1. Evolución del Sistema de Mapeo

En esta parte se presenta la evolución experimental del subsistema de reconstrucción de entorno, obtenida en el laboratorio de electrónica dentro de la UPSE. Los resultados demuestran la mejora progresiva en el algoritmo de procesamiento de datos del sensor LIDAR capturando desde una superposición crítica de puntos hasta la generación de un mapa global fijo y coherente.

3.1.1. Prueba Inicial: Superposición Crítica de Puntos LIDAR

Como resultado del primer intento de mapeo se presenta una superposición masiva y crítica de los puntos captados por el sensor LiDAR (ver en la Fig. 44). Aunque los datos fueron adquiridos correctamente, se acumulan de manera caótica, sin formar una representación clara de las paredes ni de los obstáculos. La trayectoria del robot, representada en línea roja, es visible; sin embargo, el mapa resultante carece de definición.

Este resultado fue fundamental para evidenciar el cumplimiento del primer objetivo específico, relacionado con el estudio del sensor LiDAR, lo cual permitió evidenciar de inmediato la necesidad de implementar algoritmos de registro y filtrado de la nube de puntos, correspondiente al segundo objetivo específico.

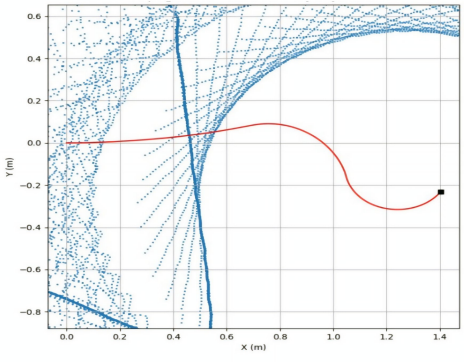


Figura 44: Mapa global XY inicial que muestra los puntos LIDAR (azules) y la trayectoria predefinida del robot (línea roja) [Fuente: Autor].

3.1.2. Segunda iteración: Persistencia de Problemas de Registro

A continuación se presenta un intento de corrección en la adquisición del mapa, sin embargo, generó un aumento en la superposición de los puntos (ver en la Fig. 45), lo que evidencia un error en el registro de la posición y la orientación del robot durante cada medición. Como resultado, el mapa obtenido se encuentra lejos de representar de manera útil el entorno.

A pesar de no haber sido una iteración exitosa, esta fase resultó fundamental en el proceso de desarrollo, por lo que permitió descartar determinados enfoques en el procesamiento de datos y resaltó la necesidad de implementar una fusión entre sensores y odometría más robusta.

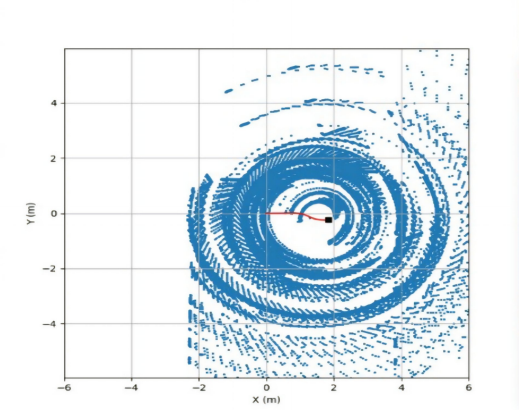


Figura 45: Mapa global de una segunda prueba, donde se observa una densidad extrema de puntos en el eje Y para un rango amplio del eje X.) [Fuente: Autor].

3.1.3. Tercera iteración: Avance con mapa en tiempo real y prueba en trayectoria recta

Con esta iteración se muestra un avance significativo en el desarrollo. En este caso se visualiza la trayectoria del robot con claridad desde el punto de inicio hasta el punto final (ver en la Fig. 46). Sin embargo, se observa que el mapa generado se desplaza junto con el robot, formando una estela en lugar de una representación estática. Esto indica que el sistema actualiza el mapa en tiempo real, pero aún no logra consolidarlo en una sola estructura global.

Este resultado indica un avance significativo al mostrar el funcionamiento preliminar de la unión entre los sensores y el sistema de navegación. Este comportamiento, en el que el mapa se mueve junto al desplazamiento del robot, resalta la necesidad de mejorar el algoritmo de SLAM para obtener un mapa estable y consistente del entorno.

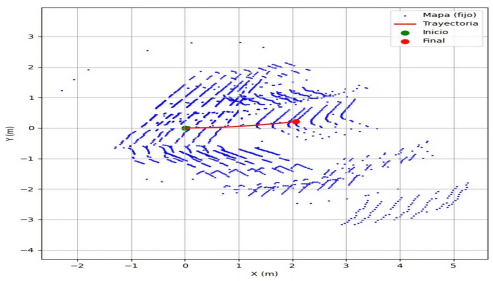


Figura 46: Visualización del mapa en tiempo real junto con la trayectoria recorrida por el robot. [Fuente: Autor].

3.1.4. Cuarta iteración: Generación de un Mapa Global Fijo y Estable

El mapa obtenido en esta etapa evidencia un avance significativo del sistema durante una trayectoria recta (ver en la Fig. 47), el mapa se consolida de una mejor forma mostrando paredes paralelas claramente definidas. Esto demuestra la escalabilidad del sistema, Esto se debe que en un recorrido completo genera un mapa coherente y estable del espacio explorado. Los puntos LiDAR se alinean correctamente, formando estructuras compactas que corresponden a los límites físicos del laboratorio. Estos resultados presentados demuestran un evidente cumplimiento del segundo objetivo específico, que trata del análisis de los datos para así lograra la recreación del entorno.

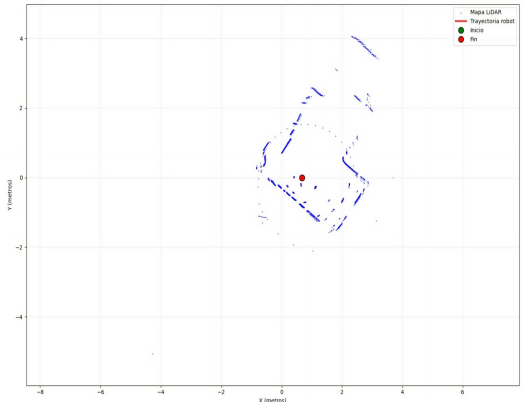


Figura 47: Mapa global fijo resultante de una trayectoria en línea recta. [Fuente: Autor].

3.1.5. Quinta iteración: Mapeo adaptativo con compensación de deriva en tiempo real

Como resultado de las constantes mejoras del algoritmo, se obtuvo un esquema ocupacional, donde se emplea escalas logarítmicas diferenciales entre las zonas libres y los obstáculos (ver en la Fig. 48). Para las áreas ocupadas se utilizan incrementos positivos, en cuanto a las áreas libres opera mediante decrementos, estableciendo así umbrales dinámicos que conforme a la exploración se van actualizando. Este sistema utiliza un sistema de reconocimiento de recurrencias espaciales, que permite que el robot identifique de forma autónoma cuando este retorne a ubicaciones donde ya estuvo con anterioridad. Si no existe una coincidencia considerable en un 75 % con la posición actual y los datos históricos, se inicia el proceso de corrección de los errores de posición que han sido acumulados en trayecto de la ruta. Con la ayuda de esta corrección en tiempo real se eliminan las deformaciones típicas de los ecosistemas de mapeo.

Gracias a la notable mejora en la recreación del entorno, podemos comenzar a probar el funcionamiento del robot. Este avance nos permite contar con una representación más fiel del espacio que está siendo cartografiado.

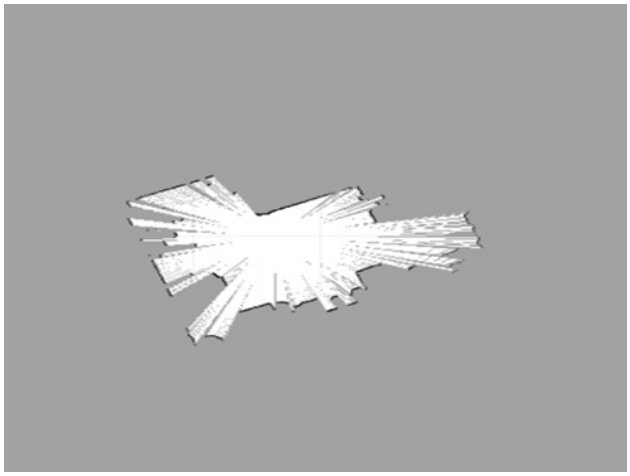


Figura 48: Mapeo adaptativo con compensación de deriva en tiempo real . [Fuente: Autor].

3.2. Pruebas en entornos cerrados: Evaluación de planificación de trayectorias y precisión de mapeoo

El sistema robótico fue sometido a evaluaciones en ambientes controlados y delimitados, seleccionados estratégicamente para eliminar variables externas que pudieran comprometer el desempeño del sistema (ver en la Fig. 49). Esta configuración experimental permitió evitar factores ambientales críticos como variaciones extremas de iluminación, condiciones meteorológicas adversas e irregularidades existentes en el terreno, garantizando así una valoración objetiva de las capacidades de la plataforma.

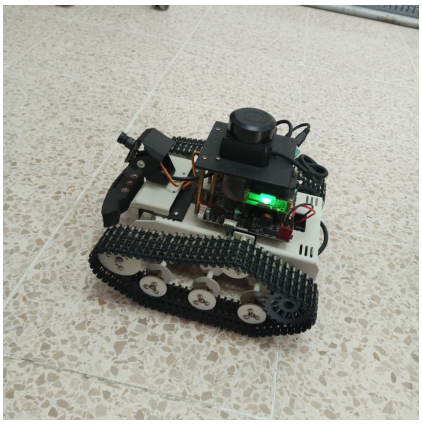


Figura 49: Robot en el entorno seleccionado . [Fuente: Autor].

La arquitectura hardware implementada incorpora sistemas sensoriales fundamentales para la percepción ambiental: una cámara PTZ de dos grados de libertad que proporciona retroalimentación visual en tiempo real del entorno seleccionado, y un sensor LiDAR que ejecuta barridos láser omnidireccionales para la captura de datos puntuales (ver en la Fig. 50). Esta información es procesada computacionalmente para reconstruir una representación del entorno, estableciendo los cimientos para las funciones de navegación autónoma y cartografía digital.

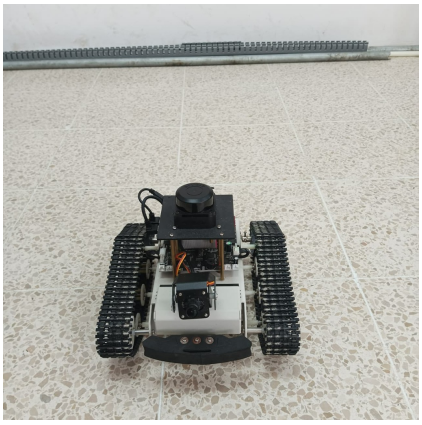


Figura 50: Arquitectura del hardware implementada. [Fuente: Autor].

El procedimiento de validación del sistema se realizo en dos modos de operación, modo manual, donde el operador supervisa todos los movimientos del robot a través de la interfaz y mediante la observación en ese momento. En el modo automático, el sistema realiza de manera independiente la realización de las rutas planificadas. Este enfoque facilita la evaluación del sistema tanto en la respuesta del robot y la funcionalidad del sistema.

3.2.1. Prueba en Modo Manual

En la configuración de operación manual, el sistema otorga control completo al operador sobre la cinemática del robot mediante una interfaz gráfica intuitiva. El usuario dirige los desplazamientos utilizando los comandos direccionales disponibles: movimiento hacia adelante, atrás, rotación hacia la izquiera y derecha, manteniendo en todo momento la supervisión directa del comportamiento robótico.

Se presenta a continuación el espacio de pruebas donde se va a evaluar el desempeño del sistema de recreación de mapas del robot (ver en la Fig. 51).

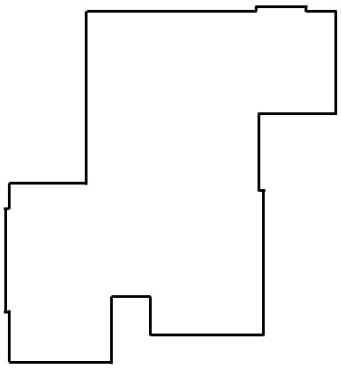


Figura 51: Ilustración del área seleccionada para la evaluación del desempeño del robot en modo manual. [Fuente: Autor].

Durante la ejecución del sistema manual, la interfaz genera en tiempo real un registro gráfico de la trayectoria ejecutada por el operador durante la prueba (ver en la Fig. 52), representando visualmente la secuencia completa de movimientos realizados. Esta funcionalidad permite documentar con precisión las rutas seguidas y establecer correlaciones directas entre los comandos del operador y el comportamiento cinemático que da como resultado el robot.

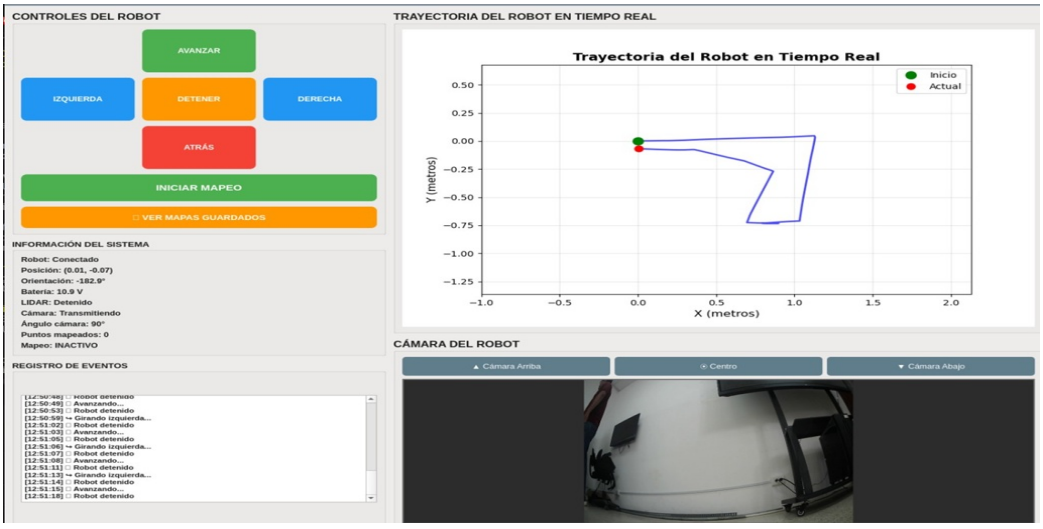


Figura 52: Trayectoria realizada por el operador en modo automático. [Fuente: Autor].

De forma paralela al control manual, el sensor LiDAR mantiene una operación continua, capturando datos ambientales que son procesados para construir progresivamente una representación del espacio seleccionado. Este mapeo obtenido en esta prueba (ver en la Fig. 53) demuestra la capacidad del sistema para realizar adquisición de datos sensoriales independientemente del modo operativo seleccionado.

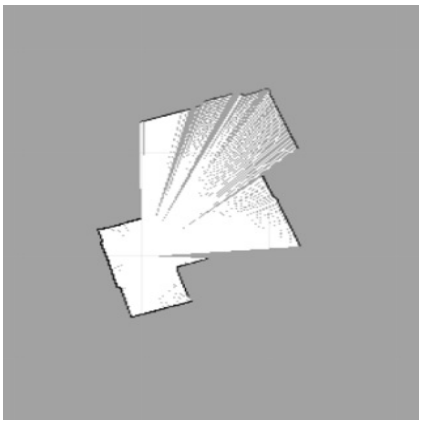


Figura 53: Resultado del mapeo obtenido en la primer prueba. [Fuente: Autor].

Como producto de la prueba se obtuvo una representación cartográfica del entorno, bajo el control del movimiento del robot decididas por el operador. Las métricas de efectividad de esta prueba se evidencian en la tabla 12:

Tabla 12: Métricas de desempeño durante operación manual

Tiempo de ejecución (min)	2
Precisión del mapeo (%)	70
Número de puntos obtenidos	4697
Error de trayectoria (%)	3
Distancia entre inicio y final(m)	0.05
Distancia total recorrida(m)	1.8

3.2.2. Prueba en Modo Automático

En la configuración de operación autónoma, el sistema ejecuta trayectorias predefinidas mediante algoritmos de planificación de rutas, sin intervención del operador humano. La plataforma utiliza los datos del sensor LiDAR para navegar de forma independiente mientras genera simultáneamente el mapa del entorno y registra la trayectoria ejecutada en tiempo real.

A continuación, se presenta el espacio de prueba para el modo automático, donde se va a evaluar el desempeño del sistema de recreación del entorno(ver en la Fig. 54).

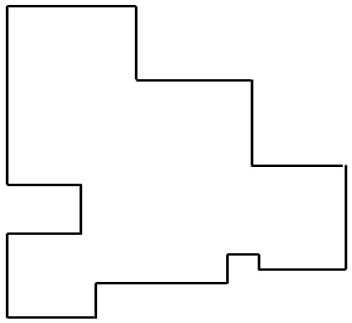


Figura 54: Ilustración del área seleccionada para la evaluación del desempeño del robot en modo automáticp. [Fuente: Autor].

En el modo automático se planifica y ejecuta la trayectoria basándose en la información sensorial disponible, evitando obstáculos mediante el algoritmo de evasión implementado. Durante la navegación, la interfaz gráfica proporciona visualización simultánea de la ruta planificada y la trayectoria real ejecutada por el robot (ver en la Fig. 55). Facilitando el monitoreo continuo del sistema de control en cada momento.

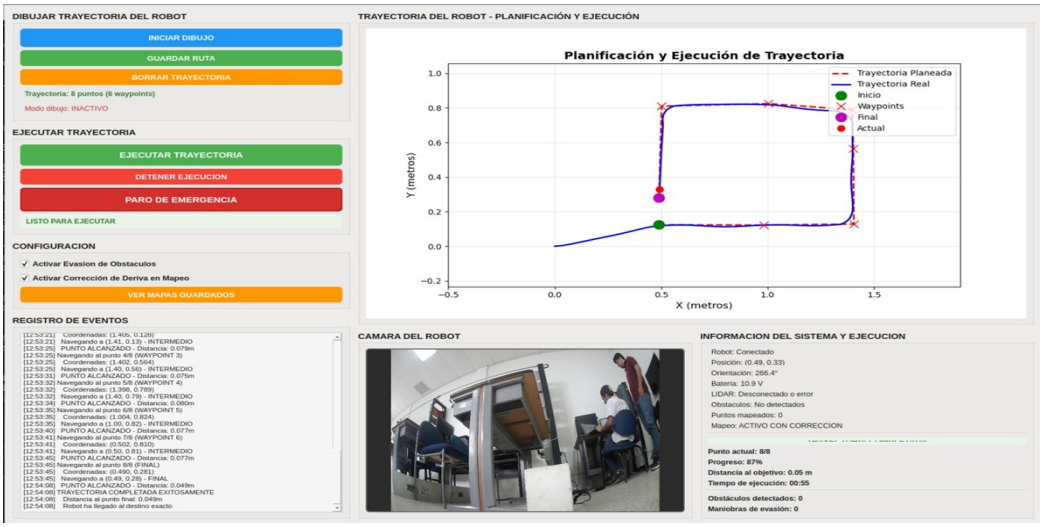


Figura 55: Trayectoria definida para evaluar el desempeño del robot en modo automático. [Fuente: Autor].

Como resultado del modo automático, se produjo una representación cartográfica del entorno bastante aceptable, considerando la trayectoria previamente planificada en el algoritmo de navegación por el operador (ver en la Fig. 56). Teniendo en cuenta la reconstrucción ambiental resultante demuestra una relación bastante alta comparada con la configuración real del espacio (ver en la Fig. 54).

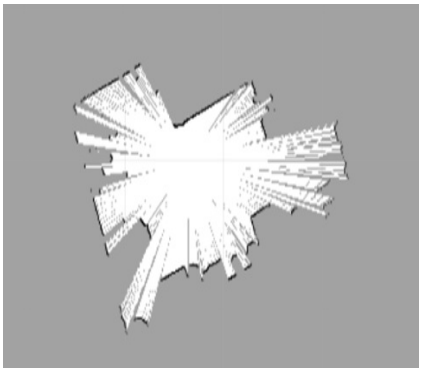


Figura 56: Resultado del mapeo obtenido en la primer prueba. [Fuente: Autor].

A continuación se presentan en la tabla 13 las métricas resultantes de

esta operación en el modo automático.

Tabla 13: Métricas de desempeño durante operación automática

Tiempo de ejecución (min)	2
Precisión del mapeo (%)	80
Número de puntos obtenidos	4697
Waypoint recorridos	8 puntos 6 Waypoint
Error de trayectoria (%)	2
Distancia entre inicio y final(m)	0.05
Distancia total recorrida(m)	0.05

4. Conclusiones

Se implementó exitosamente un sistema de adquisición de datos LIDAR mediante el desarrollo de un driver personalizado que interpreta el protocolo de comunicación del sensor RPLIDAR. El estudio detallado del protocolo permitió decodificar correctamente los comandos de entrada y salida, así como la estructura de los paquetes de datos en modo Express. Esta implementación aprovecha la capacidad del sensor para generar paquetes de 84 bytes, optimizando la densidad de puntos por revolución y facilitando la reconstrucción detallada del entorno mediante nubes de puntos de alta resolución.

Se realizo el desarrollo de un flujo de trabajo para el manejo de nubes de puntos que incorpora técnicas de SLAM, abarcando funciones como cierre de bucles y emparejamiento de escaneos. Las cuales ajustan de manera automática la desviación odométrica, brindando así la estimación de la ubicación del robot de forma precisa y de las paredes y obstáculos. La implementación incluye ajustes para la orientación entre el sensor LiDAR y la estructura del robot, reproduciendo mapas de forma coherentes con la realidad.

Se desarrolló e implementó un algoritmo de planificación de trayectoria

que integra navegación global con evasión local de obstáculos. El sistema mantiene al robot en su ruta planificada mientras monitorea continuamente el entorno, desviándose temporalmente cuando detecta obstáculos y reincorporándose automáticamente a la trayectoria original una vez superado el impedimento. Esta estrategia garantiza una exploración eficiente del área asignada mientras prioriza la seguridad del movimiento del robot.

Se implementó un sistema de navegación autónoma que fusiona datos del sensor LIDAR y la cámara visual, permitiendo la detección y mapeo simultáneo del entorno en tiempo real. La integración multisensorial proporciona redundancia en la apreciación del entorno mejorando la robustez del sistema ante condiciones variables y facilitando la reconstrucción precisa de características del entorno para la planificación de movimientos.

El sistema diseñado dio como resultado un rendimiento adecuado mientras se llevaron acabo las pruebas experimentales, demostrando el correcto funcionamiento del sistema combinando mapeo, seguimiento de trayectoria y la evasión de los obstáculos. La evaluación en entornos diferentes confirmo la correcta operatividad del sistema, evidenciando habilidades del sistema en la navegación y la recreación de mapas del entorno, logrando una eficacia esperada.

5. Recomendaciones

Se sugiere analizar detalladamente los protocolos de comunicación que corresponde a cada modelo de sensor LiDAR. Es crucial tener en cuenta la estructura de los datos enviados, además de comprender los valores hexadecimales de las cabeceras, y aplicar algoritmos para la correcta decodificación de los datos obtenidos para su posterior implementación.

Se recomienda realizar un análisis detallado de los fundamentos matemáticos de cada algoritmo de procesamiento implementado, ya que estos se basan en cálculos y aproximaciones numéricas específicas. Es fundamental estudiar detalladamente todos los parámetros de configuración, considerando que los

valores estándar aplicados en este trabajo están optimizados para el rango de sensibilidad del sensor actual.

Para futuras mejoras se recomienda la correcta gestión de recursos que manejen de forma eficiente la capacidad de procesamiento de datos entre la visión computacional, el mapeo y la planificación de trayectoria. Optimizando así el uso de la RAM del controlador y asegurando su funcionamiento aun en entornos complicados o de mayor demanda.

Se recomienda perfeccionar los algoritmos de mapeo para así aumentar el rendimiento del sistema, utilizando técnicas de cierre mas efectivas y el correcto ajuste de los métodos de asociación con escaneos.

Bibliografía

Referencias

[1] A. Molina-Leal, A. Gómez-Espinosa, J. A. Escobedo Cabello, E. Cuán-Urquizo y S. R. Cruz-Ramírez, 'Trajectory Planning for a Mobile Robot in a Dynamic Environment Using an LSTM Neural Network', Applied Sciences, vol. 11, no. 22, p. 10689, 2021. Disponible en: `https://www.mdpi.com/2076-3417/11/22/10689`

[2] I. A. G. Ulloa and W. J. N. Guerrero, "Desarrollo de un robot autónomo con sistema de mapeo y detección de obstáculos mediante sensor LiDAR," Tesis de grado, Universidad Politécnica Salesiana, Cuenca, Ecuador, 2024. Disponible en: `http://dspace.ups.edu.ec/handle/123456789/27937`

[3] D. Rodríguez Nieto, 'Título del Trabajo Fin de Máster', Universidad Politécnica de Madrid, Madrid, España, 2023. Disponible en: `https://oa.upm.es/74897/1/TFM_DANIEL_RODRIGUEZ_NIETO_2.pdf`. Accedido: 2025-09-01.

[4] International Federation of Robotics, 'World Robotics Report 2023: 3.9 Million Industrial Robots Operating Worldwide', 2023. Disponible en:

https://ifr.org/ifr-press-releases/news/robot-investments-doubled-worldwide

[5] V. Villani, F. Pini, F. Leali, and C. Secchi, 'Survey on human–robot collaboration in industrial settings: Safety, intuitive interfaces and applications', Mechatronics, vol. 55, pp. 248–266, 2018. Disponible en: https://doi.org/10.1016/j.mechatronics.2018.02.009

[6] M. L. Córdova, 'Control de movimiento y planificación de un robot esférico', Instituto Nacional de Astrofísica, Óptica y Electrónica, 2018. Disponible en: http://inaoe.repositorioinstitucional.mx/jspui/handle/1009/2032

[7] C. Pérez Villalva, A. Zambrano Rodríguez, and M. Miranda Ramos, 'Desarrollo de un prototipo robótico para desinfección de ambientes cerrados, contaminados por COVID-19 utilizando tecnología ROS y Visión Artificial', SERIE, vol. 14, no. 9, pp. 121–138, Aug. 2021. Disponible en: http://www.dspace.espol.edu.ec/handle/123456789/52783

[8] R. M. Tapia García, 'Robótica móvil'. Dec-2017. Disponible en: http://repositorio.ugto.mx/handle/20.500.12059/4898

[9] A. E. Gil Pérez, 'Robótica móvil: Qué es y sus aplicaciones'. Jul-2022. Disponible en: https://openwebinars.net/blog/robotica-movil-que-es-y-sus-aplicaciones/

[10] A. S. Navarro and M. Pucllas Palacios, 'Diseño de un robot móvil de dos hileras para trasplante de plántulas de arroz en sembríos de la Costa Norte Peruana', 2020. [Online]. Disponible en: https://hdl.handle.net/20.500.14138/3580

[11] 'Actuadores para robótica', 2025. [Online]. Available: https://roboticoss.com/actuadores-para-robotica/. Disponible en: https://roboticoss.com/actuadores-para-robotica/

[12] Robotnik, 'Tipos de sensores en robótica móvil', Robotnik.eu, 13 Abril 2023. [Online]. Disponible en: https://robotnik.eu/es/tipos-de-sensores-en-robotica-movil/

[13] J. R. Figueroa Olmedo, W. M. Montalvo López, y M. M. Bayas Sampedro, *Cinemática y Dinámica de Robots Móviles con Ruedas*, 1ra ed. Quito, Ecuador: CILADI, 2023. [En línea]. Disponible: https://ciladi.org/wp-content/uploads/Libro-Robots-VF3.pdf

[14] F. Guevara Soriano, A. A. Reyes Montero y A. Sánchez López, «Seguimiento autónomo de personas con un robot,» Research in Computing Science, 2017. [En línea]. Disponible en: https://rcs.cic.ipn.mx/2017_135/Seguimiento%20autonomo%20de%20personas%20con%20un%20robot%20aereo%20no%20tripulado.pdf

[15] C. Parejo, «La movilidad mediante robots aéreos: Han llegado los drones,» Revista La Comuna, 30 Enero 2024. [En línea]. Disponible en: https://www.revistalacomuna.com/

[16] F. R. Cabrera Aguayo, «Diseño y desarrollo de robots submarinos bioinspirados,» Tesis doctoral, E.T.S. de Ingenieros Industriales, Universidad Politécnica de Madrid, Madrid, España, Diciembre 2021. [En línea]. Disponible en: https://oa.upm.es/69989/

[17] EcoInventos, «El robot submarino que puede ayudar a 'reforestar' los arrecifes de coral del mundo,» EcoInventos, 3 Enero 2020. [En línea]. Diponible en: https://ecoinventos.com/larvalbot/

[18] D. Electronics, «Robot todo terreno UGV01,» [En línea]. Disponible en: https://www.dynamoelectronics.com/tienda/robot-todo-terreno-ugv01/

[19] R. Álvarez, «Este sorprendente robot cuadrúpedo es autónomo, mantiene el equilibrio y hasta sabe pedir un ascensor,» Xataka, 26 Diciembre 2017. [En línea]. Disponible en: https://www.xataka.com/robotica-e-ia/este-sorprendente-robot-cuadrupedo-es-autonomo-mantiene-el-equilibrio-y-hasta-saber-pedir-un-ascensor

[20] A. Apaza, «Diseño y construcción de un robot móvil para exploración de ambientes interiores,» Tesis de Licenciatura, Escuela Profesional de Ingeniería Electrónica, Universidad Católica de Santa María, Arequipa, Perú, 2020. [En línea]. Disponible en: https://repositorio.ucsm.edu.pe/handle/20.500.12920/10779

[21] C. Sánchez, «Starship da un paso más,» Logística - CdeComunicacion.es, 5 Noviembre 2018. [En línea]. Disponible en: https://logistica.cdecomunicacion.es/noticias/sectoriales/29230/starship-da-un-paso-mas-sus-robots-ya-son-repartidores-autonomos-y-entregan-a-domicilio-en-reino-unido

[22] R. Siegwart y I. R. Nourbakhsh, «Introduction to Autonomous Mobile Robots,» 2004. [En línea]. Disponible en: https://www.ucg.ac.me/skladiste/blog_13268/objava_56689/fajlovi/Introduction%20to%20Autonomous%20Mobile%20Robots%20book.pdf

[23] R. Cao , J. Gu, C. Yu y R. André, «OmniWheg: An omnidirectional wheel-leg transformable robot,» 2022. [En línea]. Disponible en: https://arxiv.org/abs/2203.02118

[24] G. Calandin y L. Ignacio, «Modelado cinemático y control de robots móviles con ruedas,» 2008. [En línea]. Disponible en: https://riunet.upv.es/handle/10251/1840

[25] A. H. Husam A. Neama, «A Comparative Review of Omnidirectional Wheel Types for Mobile Robotics,» 2023. [En línea]. Disponible en: https://www.researchgate.net/publication/372941239_A_Comparative_Review_of_Omnidirectional_Wheel_Types_for_Mobile_Robotics

[26] A. G. Cabrera Castro y J. P. Reinozo Chacón, "Diseño y construcción de un robot móvil omnidireccional basado en ROS para la enseñanza de conceptos matemáticos a estudiantes de bachillerato," Tesis de grado, Universidad Politécnica Salesiana, Cuenca, Ecuador, 2022. [En línea]. Disponible: http://dspace.ups.edu.ec/handle/123456789/24330

[27] V. J. Gonzalez Villela, A. J. A, Sánchez Balpuesta y A. Súares Arriaga, «Cinemática de una plataforma móvil omnidireccional con llantas Mecanum en configuración AB,» Septiembre 2015. [En línea]. Disponible en: https://www.researchgate.net/publication/342945027_Cinematica_de_una_plataforma_movil_omnidireccional_con_llantas_Mecanum_en_configuracion_AB

[28] M. E. HERRERA CORDERO, «Desarrollo de un robot rueda con movimiento en un solo plano,» 2020. [En línea]. Disponible en: http://www.mecamex.net/tesis/2020/Tesis_Mario.pdf

[29] D. Bravo y C. F. Rengifo, «Estudio de la Dinámica y Control de una Bicicleta Robótica,» 2020. [En línea]. Disponible en: https://www.researchgate.net/publication/338879389_Estudio_de_la_Dinamica_y_Control_de_una_Bicicleta_Robotica

[30] O. Gutiérrez Frías, «Estabilización del Péndulo Invertido Sobre Dos Ruedas mediante el método de Lyapunov,» 2013. [En línea]. Disponible en: https://www.researchgate.net/publication/257684403_Estabilizacion_del_Pendulo_Invertido_Sobre_Dos_Ruedas_mediante_el_metodo_de_Lyapunov

[31] U. D. Bosco, «Modelo Cinemático,» Fundamentos de Robótica, Guía 4, 2020. [En línea]. Disponible en: https://www.udb.edu.sv/udb_files/recursos_guias/electronica-ingenieria/fundamentos-de-robotica/2020/i/guia-4.pdf

[32] D. M. A. R. C. T. S. Y. Vlad Ovidiu Mihalca, «Simulating Simple Tasks for a Kinematic Model of Differential-drive Mobile Robots,» 2023. [En línea]. Disponible en: https://www.researchgate.net/publication/372274271_Simulating_Simple_Tasks_for_a_Kinematic_Model_of_Differential-drive_Mobile_Robots

[33] S. J. Chapman, «Electric Machinery Fundamentals,» McGraw-Hill Education, 2011. [En línea]. Disponible en: https://books.google.es/books?id=4of6AQAAQBAJ

[34] M. Hernando, J. M. Sanz y D. Olayo, «Planificación de trayectorias,» Noviembre 2021. [En línea]. Disponible en: https://blogs.upm.es/miguelhernando/wp-content/uploads/sites/734/2022/11/GuiadoYNavegacion_Planificacion.pdf

[35] C. A. Velásquez Hernández, J. J. Chávez Chávez, y E. Córdoba Nieto, "Implementación de sistema de navegación autónomo en robot móvil experimental para reconstrucción y exploración de entornos desconocidos," *Revista EIA*, vol. 12, no. 24, pp. 45-58, dic. 2015. [En línea]. Disponible: https://revistas.eia.edu.co/index.php/mem/article/view/817

[36] F. Sing, «Componentes de un sistema de comunicación,» Creando TODO a partir de NADA, 4 Marzo 2014. [En línea]. Disponible en: https://fernandosing.wordpress.com/2014/03/04/componentes-de-un-sistema-de-comunicacion/

[37] Ikusi, «Qué son las redes inalámbricas y cuáles son las características que deben tener,» [En línea]. Disponible en: https://www.ikusi.com/mx/blog/que-son-las-redes-inalambricas/

[38] Tecnipesa, «Tecnología RFID: ¿Qué ventajas tiene?,» [En línea]. Disponible en: https://www.tecnipesa.com/blog/69-tecnologia-rfid-que-ventajas-tiene

[39] S. Corporation, «¿Qué es la tecnología inalámbrica BLUETOOTH?,» Guía de Ayuda, 2012. [En línea]. Disponible en: https://helpguide.sony.net/gbmig/44506666/v1/es/contents/02/01/01/01.html

[40] Homey, «¿Qué es Zigbee? La tecnología de red eléctrica inteligente más popular del mundo,» 2023. [En línea]. Disponible en: https://homey.app/es-es/wiki/que-es-zigbee/

[41] Concepto, «Wifi - Concepto, características, origen, usos y tipos,» [En línea]. Disponible en: https://concepto.de/wifi/

[42] U. México, «Lenguajes de programación, qué son, cuántos hay y cómo funcionan,» 27 Noviembre 2024. [En línea]. Diponible en: https://mexico.unir.net/noticias/ingenieria/lenguajes-programacion/

[43] U. O. Uruguay, «Los 10 lenguajes de programación más usados actualmente,» [En línea]. Disponible en: https://fi.ort.edu.uy/blog/los-10-lenguajes-de-programacion-mas-usados-actualmente

[44] Shopify, "Yahboom,"2024. [En línea]. Disponible en: https://category.yahboom.net/collections/robotics/products/transbot-se?variant=45278072144188

[45] YouYeetoo Corporation, «RPLIDAR C1M1-R2,» [En línea]. Disponible en: https://wiki.youyeetoo.com/en/Lidar/C1M1-R2.

[46] Yahboom, "Transbot Expansion Board,"2025. [En línea]. Disponible en: https://category.yahboom.net/products/transbot-expansion-board.

[47] Yahboom, "MicroROS-Pi5," [En línea]. Disponible en: http://www.yahboom.net/study/MicroROS-Pi5

[48] Yahboom, "Motorreductor de CC 520 con codificador, 205 RPM, 333 RPM y 550 RPM," 2025. [En línea]. Disponible en: https://category.yahboom.net/es/products/md520.

[49] AGelectronica, «Datasheet Sensor Ultrasónico HC-SR04,» AGelectronica, [En línea]. Disponible en: https://agelectronica.lat/pdfs/textos/U/ULTRASONIC-HC-SR04.PDF

[50] Raspberry Pi Foundation, "Raspberry Pi Imager," *Raspberry Pi*, [En línea]. Disponible en: https://www.raspberrypi.com/software/.

[51] Famatech, "Advanced IP Scanner," *Advanced IP Scanner*, [En línea]. Disponible en: https://www.advanced-ip-scanner.com/

[52] RealVNC, "VNC Viewer," *RealVNC*, [En línea]. Disponible en: https://www.realvnc.com/en/connect/download/viewer/

[53] Unipython, «PyQt5: Interfaces gráficas con Python,» Unipython, [En línea]. Disponible en: https://unipython.com/pyqt5-interfaces-graficas-con-python/

[54] NumPy, «What is NumPy?,» [En línea]. Disponible en: https://numpy.org/doc/2.3/user/whatisnumpy.html.

[55] Instituto de Astrofísica de Canarias, «SciPy: Scientific Python,» IAC Research, [En línea]. Disponible en: https://research.iac.es/sieinvens/python-course/scipy.html

[56] Interactive Chaos, «La librería Math de Python,» Interactive Chaos, [En línea]. Disponible en: https://interactivechaos.com/es/manual/tutorial-de-python/la-libreria-math

[57] Matplotlib, «Matplotlib: visualización con Python», [En línea]. Disponible en: https://matplotlib.org/.

[58] PySerial, «PySerial Documentation,» [En línea]. Disponible en: https://pyserial.readthedocs.io/en/latest/pyserial.html.

Anexos

Anexo A

Instalación del sistema operativo a la Raspberry pi 5

Para instalar el sistema operativo en la Raspberry Pi 5, es necesario comenzar instalando la aplicación Raspberry Pi Imager en el computador. Luego de ejecutarla con privilegios de administrador, se procede a elegir el dispositivo donde se llevará a cabo la instalación.

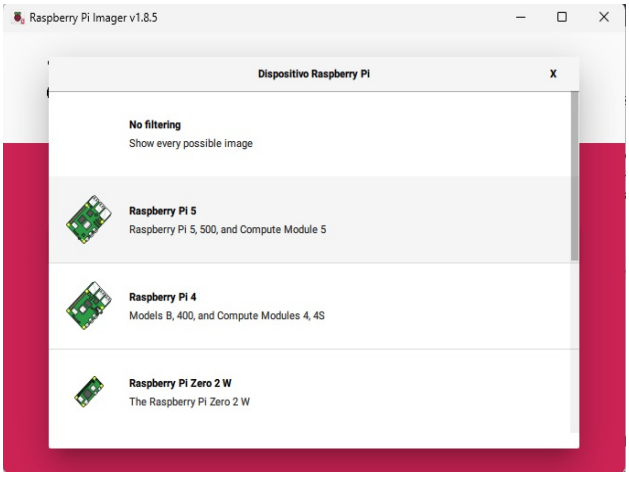


Figura 57: Selección del dispositivo Raspberry PI 5

Después de seleccionar el dispositivo, se procede a escoger el sistema operativo. El fabricante sugiere instalar una versión de 64 bits en la Raspberry Pi 5; sin embargo, esta decisión puede ajustarse según los programas necesarios y la capacidad de almacenamiento de la tarjeta microSD utilizada en el proyecto. Para este caso, se optó por instalar el sistema operativo recomendado por el fabricante.

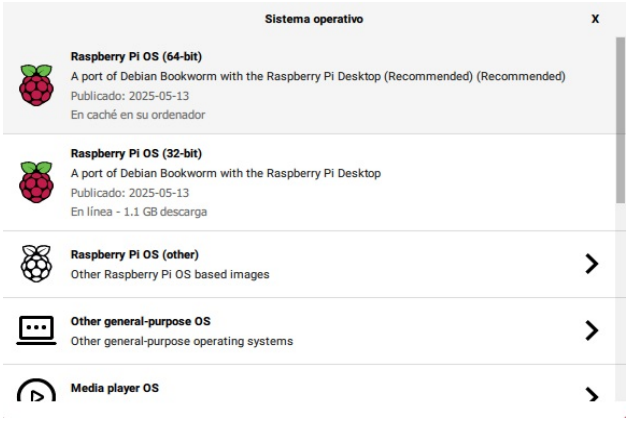


Figura 58: Selección del sistema operativo Raspberry PI Os

A continuación, se inserta la tarjeta microSD en la computadora utilizando su adaptador, permitiendo que la herramienta Raspberry Pi Imager la reconozca y la muestre en su interfaz. Este paso también sirve para verificar que la tarjeta tenga una capacidad adecuada, que debe estar entre 16 GB y 64 GB. Una vez completadas estas acciones, se procede a cargar el sistema operativo en la tarjeta microSD. Durante este proceso, se muestra una barra de progreso acompañada del mensaje “Escribiendo”, que indica el avance de la instalación. Al finalizar, se retira la tarjeta de la computadora para insertarla en la ranura microSD de la Raspberry Pi 5.

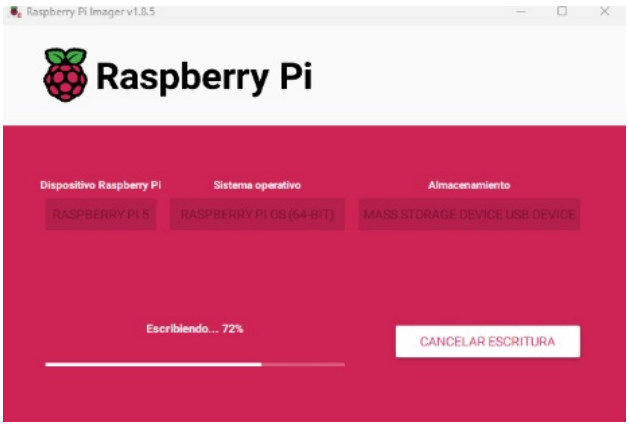


Figura 59: Selección del sistInstalación del sistema operativo Raspberry PI Os

Anexo B

Visualización de la pantalla de la Raspberry mediante RealVnc

Para acceder a la Raspberry Pi 5, se utiliza la dirección IP previamente obtenida a través del programa IP Scanner, el cual analiza la red local y detecta los dispositivos conectados, incluyendo la Raspberry Pi. Una vez identificada su dirección IP, se establece una conexión remota mediante el protocolo VNC Viewer, lo que permite controlar y gestionar la Raspberry Pi de forma sencilla, sin necesidad de conectar periféricos directamente al dispositivo.

Figura 60: Conexión entre RealVnc y la Raspberry Pi

Antes de acceder a la Raspberry Pi a través del programa RealVNC, es necesario ingresar el nombre de usuario y la contraseña previamente configurados en el dispositivo.

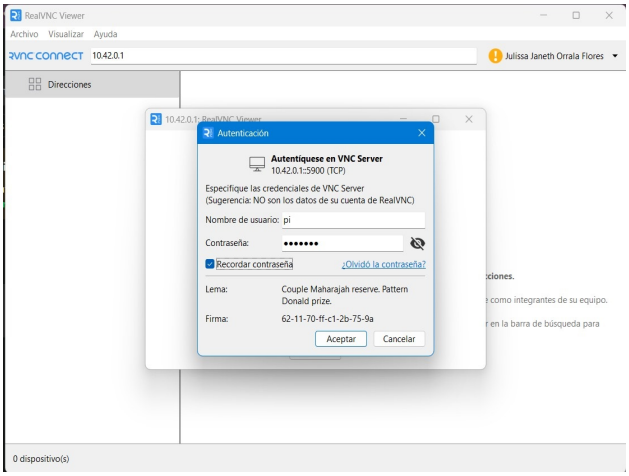


Figura 61: Acceso remoto a la Raspberry Pi

Una vez que la Raspberry Pi está conectada a través del programa RealVNC, aparecerá una ventana de verificación. En este paso, basta con aceptar para acceder a la pantalla principal del dispositivo.

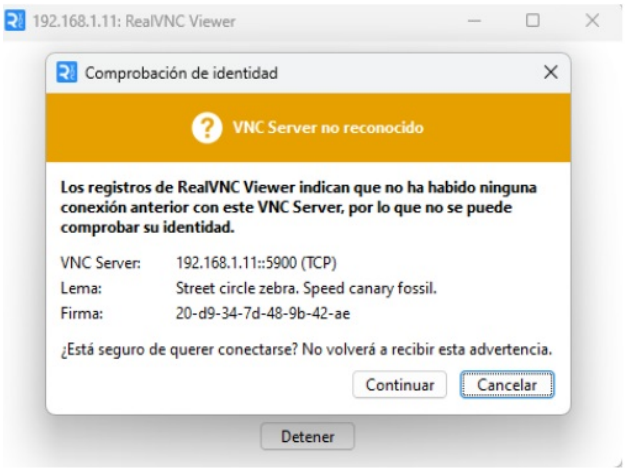


Figura 62: Comprobación de identidad para poder acceder a la Raspberry P

Después de completar los pasos anteriores, se podrá acceder al entorno inicial de la Raspberry Pi y proceder con su configuración para su uso en el presente proyecto de investigación.

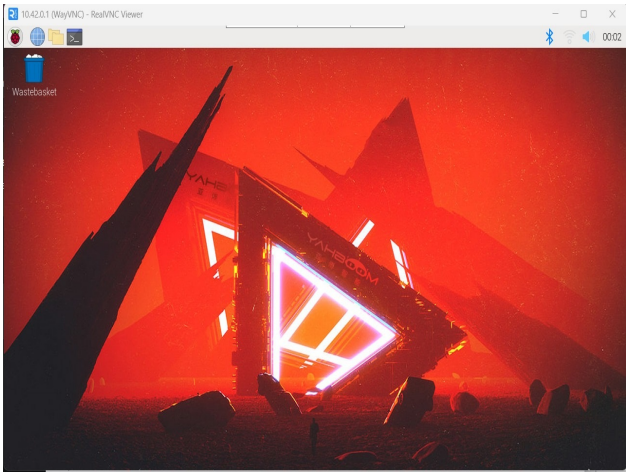


Figura 63: Pantalla principal de la Raspberry Pi

Anexo C

Diagrama del flujo de dependencias de los archivos

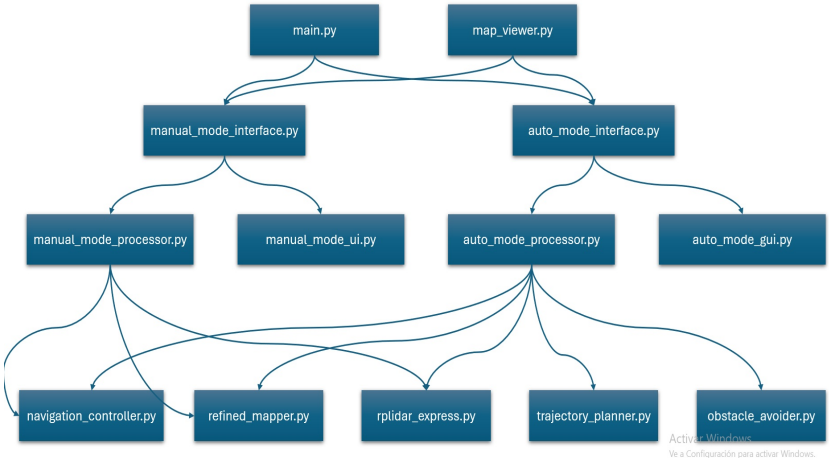


Figura 64: Diagrama de flujo de dependencias

Anexo D

Código de programación del sistema de control en Python

main.py

```python
# main.py (MODIFICADO)
#!/usr/bin/env python3
# main.py
# Sistema principal - Selección de Modo

import sys
import os
from PyQt5.QtWidgets import (QApplication, QMainWindow, QVBoxLayout, QHBoxLayout,
                             QPushButton, QWidget, QLabel, QMessageBox)
from PyQt5.QtCore import Qt
from PyQt5.QtGui import QFont

class ModeSelectionWindow(QMainWindow):
    def __init__(self):
        super().__init__()
        self.init_ui()

    def init_ui(self):
        self.setWindowTitle("Sistema de Planificación de Trayectoria")
        self.setGeometry(300, 200, 600, 400)

        # Widget central
        central_widget = QWidget()
        self.setCentralWidget(central_widget)

        # Layout principal
        layout = QVBoxLayout(central_widget)
        layout.setAlignment(Qt.AlignCenter)

        # Título
        title = QLabel("SISTEMA DE PLANIFICACIÓN DE TRAYECTORIA\nY GENERACIÓN DE ENTORNO")
        title.setAlignment(Qt.AlignCenter)
        title.setFont(QFont("Arial", 16, QFont.Bold))
        layout.addWidget(title)

        # Espaciador
        layout.addSpacing(50)

        # Subtítulo
        subtitle = QLabel("SELECCIÓN DE MODO DE OPERACIÓN")
        subtitle.setAlignment(Qt.AlignCenter)
        subtitle.setFont(QFont("Arial", 12, QFont.Bold))
        layout.addWidget(subtitle)

        layout.addSpacing(30)

        # Botones de modo
        btn_layout = QHBoxLayout()

        # Botón Modo Manual
        self.btn_manual = QPushButton("MODO MANUAL")
        self.btn_manual.setMinimumSize(200, 80)
        self.btn_manual.setFont(QFont("Arial", 12, QFont.Bold))
        self.btn_manual.setStyleSheet("""
            QPushButton {
                background-color: #4CAF50;
                color: white;
                border: none;
                border-radius: 10px;
            }
            QPushButton:hover {
                background-color: #45a049;
            }
        """)

        # Botón Modo Automático
        self.btn_auto = QPushButton("MODO AUTOMÁTICO")
        self.btn_auto.setMinimumSize(200, 80)
        self.btn_auto.setFont(QFont("Arial", 12, QFont.Bold))
        self.btn_auto.setStyleSheet("""
            QPushButton {
                background-color: #2196F3;
                color: white;
                border: none;
                border-radius: 10px;
            }
            QPushButton:hover {
                background-color: #1976D2;
            }
        """)

        btn_layout.addWidget(self.btn_manual)
        btn_layout.addSpacing(20)
        btn_layout.addWidget(self.btn_auto)

        layout.addLayout(btn_layout)
        layout.addSpacing(50)

        # Conectar señales
        self.btn_manual.clicked.connect(self.open_manual_mode)
        self.btn_auto.clicked.connect(self.open_auto_mode)

    def open_manual_mode(self):
        try:
            from manual_mode_interface import ManualModeInterface
            self.manual_window = ManualModeInterface()
            self.manual_window.show()
            self.hide()
        except Exception as e:
            QMessageBox.critical(self, "Error", f"No se pudo iniciar el modo manual:\n{str(e)}")

    def open_auto_mode(self):
        try:
            from auto_mode_interface import AutoModeInterface
            self.auto_window = AutoModeInterface()
            self.auto_window.show()
            self.hide()
        except Exception as e:
            QMessageBox.critical(self, "Error", f"No se pudo iniciar el modo automático:\n{str(e)}")

def main():
    app = QApplication(sys.argv)
    app.setStyle('Fusion')

    window = ModeSelectionWindow()
    window.show()

    sys.exit(app.exec_())

if __name__ == "__main__":
    main()
```

map_viewer.py

```python
#!/usr/bin/env python3
# map_viewer.py
# Visor de mapas guardados - Versión mejorada

import os
import glob
import numpy as np
import matplotlib.pyplot as plt
from PyQt5.QtWidgets import (QApplication, QMainWindow, QVBoxLayout, QHBoxLayout, QPushButton, QWidget, QLabel, QListWidget, QGroupBox, QMessageBox, QScrollArea)
from PyQt5.QtCore import Qt
from PyQt5.QtGui import QFont, QPixmap

class MapViewer(QMainWindow):
    def __init__(self):
        super().__init__()
        self.current_map_index = 0
        self.map_files = []
        self.init_ui()
        self.load_maps()

    def init_ui(self):
        self.setWindowTitle("Visor de Mapas - Sistema de Mapeo")
        self.setGeometry(200, 100, 1200, 800)

        central_widget = QWidget()
        self.setCentralWidget(central_widget)

        main_layout = QHBoxLayout(central_widget)

        # Panel izquierdo - Lista de mapas (más estrecho)
        left_panel = QVBoxLayout()
        left_panel.setContentsMargins(5, 5, 5, 5)

        list_group = QGroupBox("Mapas Guardados")
        list_group.setFont(QFont("Arial", 11, QFont.Bold))
        list_group.setMaximumWidth(250) # Más estrecho

        list_layout = QVBoxLayout()

        self.map_list = QListWidget()
        self.map_list.itemClicked.connect(self.on_map_selected)
        list_layout.addWidget(self.map_list)

        # Botones de control
        btn_layout = QVBoxLayout()

        self.btn_prev = QPushButton("Anterior")
        self.btn_next = QPushButton("Siguiente")
        self.btn_refresh = QPushButton("Actualizar")
        self.btn_delete = QPushButton("Eliminar")

        self.btn_prev.clicked.connect(self.previous_map)
        self.btn_next.clicked.connect(self.next_map)
        self.btn_refresh.clicked.connect(self.load_maps)
        self.btn_delete.clicked.connect(self.delete_map)

        for btn in [self.btn_prev, self.btn_next, self.btn_refresh, self.btn_delete]:
            btn.setMinimumHeight(35)
            btn.setFont(QFont("Arial", 9))
            btn.setStyleSheet("""
                QPushButton {
                    background-color: #2196F3;
                    color: white;
                    border: none;
                    border-radius: 5px;
                }
                QPushButton:hover {
                    background-color: #1976D2;
                }
            """)

        btn_layout.addWidget(self.btn_prev)
        btn_layout.addWidget(self.btn_next)
        btn_layout.addWidget(self.btn_refresh)
        btn_layout.addWidget(self.btn_delete)

        list_layout.addLayout(btn_layout)
        list_group.setLayout(list_layout)
        left_panel.addWidget(list_group)

        # Panel derecho - SOLO Visualización del mapa (más espacio)
        right_panel = QVBoxLayout()

        # Visualización del mapa - Ocupa todo el espacio
        map_group = QGroupBox("Visualización del Mapa")
        map_group.setFont(QFont("Arial", 12, QFont.Bold))

        map_layout = QVBoxLayout()
        self.lbl_map_image = QLabel()
        self.lbl_map_image.setAlignment(Qt.AlignCenter)
        self.lbl_map_image.setMinimumSize(800, 700) # Más grande
        self.lbl_map_image.setStyleSheet("""
            QLabel {
                background-color: #2C2C2C;
                border: 2px solid #666;
            }
        """)
        map_layout.addWidget(self.lbl_map_image)
        map_group.setLayout(map_layout)
        right_panel.addWidget(map_group)

        # Agregar paneles
        main_layout.addLayout(left_panel, 0) # Panel izquierdo más estrecho
        main_layout.addLayout(right_panel, 1) # Panel derecho ocupa más espacio

    def load_maps(self):
        """Cargar lista de mapas guardados"""
        self.map_list.clear()
        self.map_files = []

        # Buscar archivos de mapa
        map_patterns = [
            "manual_map_*.png",
            "corrected_map_*.png",
            "refined_map_*.png",
            "auto_map_*.png"
        ]

        for pattern in map_patterns:
            map_files = glob.glob(pattern)
            for map_file in map_files:
                if map_file not in self.map_files:
                    self.map_files.append(map_file)

        # Ordenar por fecha (más reciente primero)
        self.map_files.sort(reverse=True)

        # Agregar a la lista
        for map_file in self.map_files:
            filename = os.path.basename(map_file)
            # Extraer timestamp del nombre
            if 'manual_map_' in filename:
                timestamp = filename.replace('manual_map_', '').replace('.png', '')
                display_name = f"Manual - {timestamp}"
            elif 'corrected_map_' in filename:
                timestamp = filename.replace('corrected_map_', '').replace('.png', '')
                display_name = f"Corregido - {timestamp}"
            elif 'refined_map_' in filename:
                timestamp = filename.replace('refined_map_', '').replace('.png', '')
                display_name = f"Refinado - {timestamp}"
            elif 'auto_map_' in filename:
                timestamp = filename.replace('auto_map_', '').replace('.png', '')
                display_name = f"Auto - {timestamp}"
            else:
```

```
            display_name = filename

        self.map_list.addItem(display_name)

    if self.map_files:
        self.map_list.setCurrentRow(0)
        self.display_map(0)

def display_map(self, index):
    """Mostrar mapa seleccionado"""
    if 0 <= index < len(self.map_files):
        self.current_map_index = index
        map_file = self.map_files[index]

        try:
            # Cargar y mostrar imagen
            pixmap = QPixmap(map_file)
            if not pixmap.isNull():
                # Usar todo el espacio disponible
                scaled_pixmap = pixmap.scaled(
                    self.lbl_map_image.width() - 10,
                    self.lbl_map_image.height() - 10,
                    Qt.KeepAspectRatio,
                    Qt.SmoothTransformation
                )
                self.lbl_map_image.setPixmap(
                    scaled_pixmap)
            else:
                self.lbl_map_image.setText("Error
                    cargando imagen")

        except Exception as e:
            self.lbl_map_image.setText(f"Error: {str(e)}"
                )

def on_map_selected(self, item):
    """Manejar selección de mapa de la lista"""
    index = self.map_list.currentRow()
    self.display_map(index)

def previous_map(self):
    """Mapa anterior"""
    if self.map_files:
        new_index = (self.current_map_index - 1) % len(
            self.map_files)
        self.display_map(new_index)
        self.map_list.setCurrentRow(new_index)

def next_map(self):
    """Siguiente mapa"""
    if self.map_files:
        new_index = (self.current_map_index + 1) % len(
            self.map_files)
        self.display_map(new_index)
        self.map_list.setCurrentRow(new_index)


def delete_map(self):
    """Eliminar mapa actual"""
    if not self.map_files:
        return

    current_file = self.map_files[self.current_map_index
        ]

    reply = QMessageBox.question(
        self,
        "Confirmar eliminación",
        f"¿Está seguro de que desea eliminar el mapa?\n{
            os.path.basename(current_file)}",
        QMessageBox.Yes | QMessageBox.No,
        QMessageBox.No
    )

    if reply == QMessageBox.Yes:
        try:
            # Eliminar archivos relacionados
            base_name = current_file.replace('.png', '')
            related_files = [
                current_file,
                base_name + '_logodds.npy',
                base_name + '_trajectory.npy'
            ]

            for file in related_files:
                if os.path.exists(file):
                    os.remove(file)

            # Recargar lista
            self.load_maps()

            QMessageBox.information(self, "Éxito", "Mapa
                eliminado correctamente")

        except Exception as e:
            QMessageBox.critical(self, "Error", f"No se
                pudo eliminar el mapa: {str(e)}")

def show_map_viewer():
    """Función para mostrar el visor de mapas"""
    viewer = MapViewer()
    viewer.show()
    return viewer

if __name__ == "__main__":
    app = QApplication([])
    viewer = MapViewer()
    viewer.show()
    app.exec_()
```

manual_mode_interface.py

```
#!/usr/bin/env python3
# manual_mode_interface.py
# Interfaz de Modo Manual - Con manejo robusto de errores
    OPTIMIZADO SIN LIDAR

import sys
import time
import threading
import math
import cv2
from PyQt5.QtWidgets import QApplication, QMainWindow,
    QMessageBox
from PyQt5.QtCore import QTimer, QThread, pyqtSignal,
    pyqtSlot
from PyQt5.QtGui import QImage, QPixmap
import numpy as np

from rplidar_express import RPLidarExpress
from navigation_controller import NavigationController
from refined_mapper import RefinedMapper
from map_viewer import show_map_viewer

from manual_mode_ui import ManualModeUI
from manual_mode_processor import ManualModeProcessor

class ManualModeInterface(QMainWindow):
    def __init__(self):
        super().__init__()
        self.processor = ManualModeProcessor()
        self.ui = ManualModeUI()
        self.setCentralWidget(self.ui)
        self.setWindowTitle("Sistema de Planificación de
            Trayectoria - Modo Manual")
        self.setGeometry(100, 50, 1400, 900)
```

```
        # Variables para optimización
        self.map_viewer = None # Mantener referencia al visor de mapas

        self.connect_signals()
        self.init_systems()

    def connect_signals(self):
        """Conectar señales entre UI y Processor - SIN SEÑAL LIDAR"""
        # Controles del robot
        self.ui.btn_forward.pressed.connect(lambda: self.processor.move_robot('forward'))
        self.ui.btn_backward.pressed.connect(lambda: self.processor.move_robot('backward'))
        self.ui.btn_left.pressed.connect(lambda: self.processor.move_robot('left'))
        self.ui.btn_right.pressed.connect(lambda: self.processor.move_robot('right'))
        self.ui.btn_stop.clicked.connect(self.processor.stop_robot)
        self.ui.btn_generate_map.clicked.connect(self.toggle_mapping)
        self.ui.btn_view_maps.clicked.connect(self.show_map_viewer)

        # Controles de cámara
        self.ui.btn_camera_up.clicked.connect(lambda: self.processor.move_camera('up'))
        self.ui.btn_camera_down.clicked.connect(lambda: self.processor.move_camera('down'))
        self.ui.btn_camera_center.clicked.connect(lambda: self.processor.move_camera('center'))

        # Señales del procesador a la UI - SIN SEÑAL LIDAR
        self.processor.status_signal.connect(self.ui.update_status)
        self.processor.log_signal.connect(self.ui.log_message)
        self.processor.trajectory_signal.connect(self.ui.update_trajectory)
        # SEÑAL LIDAR ELIMINADA - No se necesita para visualización
        self.processor.camera_frame_signal.connect(self.ui.update_camera_frame)
        self.processor.camera_error_signal.connect(self.ui.handle_camera_error)

    def init_systems(self):
        """Inicializar sistemas reales del robot"""
        try:
            self.ui.log_message("Inicializando sistemas...")
            self.processor.initialize_systems()
            self.ui.log_message("Sistemas inicializados correctamente")

        except Exception as e:
            self.ui.log_message(f"Error inicializando sistemas: {str(e)}")
            QMessageBox.critical(self, "Error de Inicialización",
                                 f"No se pudieron inicializar todos los sistemas:\n{str(e)}")

    def toggle_mapping(self):
        """Activar/desactivar guardado de mapa"""
        self.processor.toggle_mapping()

        # Actualizar UI según estado
        if self.processor.is_mapping:
            self.ui.btn_generate_map.setText("DETENER MAPEO")
            self.ui.btn_generate_map.setStyleSheet("""
QPushButton {
    background-color: #f44336;
    color: white;
    border: none;
    border-radius: 8px;
}
QPushButton:hover {
    background-color: #d32f2f;
}
""")
            self.ui.lbl_mapping_status.setText("Mapeo: ACTIVO (Guardando)")
        else:
            self.ui.btn_generate_map.setText("INICIAR MAPEO")
            self.ui.btn_generate_map.setStyleSheet("""
QPushButton {
    background-color: #4CAF50;
    color: white;
    border: none;
    border-radius: 8px;
}
QPushButton:hover {
    background-color: #45a049;
}
""")
            self.ui.lbl_mapping_status.setText("Mapeo: INACTIVO")

    def show_map_viewer(self):
        """Mostrar visor de mapas - VERSIÓN CORREGIDA"""
        try:
            # Reutilizar ventana existente en lugar de crear nueva
            if self.map_viewer is None or not self.map_viewer.isVisible():
                from map_viewer import MapViewer
                self.map_viewer = MapViewer()
                self.map_viewer.show()
            else:
                self.map_viewer.raise_() # Traer al frente si ya está abierto
                self.map_viewer.activateWindow()

            self.ui.log_message("Visor de mapas abierto")
        except Exception as e:
            self.ui.log_message(f"Error abriendo visor de mapas: {str(e)}")
            QMessageBox.critical(self, "Error", f"No se pudo abrir el visor de mapas:\n{str(e)}")

    def closeEvent(self, event):
        """Manejar cierre de la aplicación"""
        self.ui.log_message("Cerrando aplicación...")
        self.processor.cleanup()

        # Cerrar visor de mapas si está abierto
        if self.map_viewer and self.map_viewer.isVisible():
            self.map_viewer.close()

        self.ui.log_message("Sistema cerrado correctamente")
        event.accept()

def main():
    app = QApplication(sys.argv)
    app.setStyle('Fusion')

    try:
        interface = ManualModeInterface()
        interface.setWindowTitle("Sistema de Planificación de Trayectoria - Modo Manual")
        interface.setGeometry(100, 50, 1400, 900)
        interface.show()
        sys.exit(app.exec_())
    except Exception as e:
        print(f"Error fatal: {e}")
        QMessageBox.critical(None, "Error Fatal", f"El programa encontró un error:\n{str(e)}")
        sys.exit(1)

if __name__ == "__main__":
    main()
```

auto_mode_interface.py

```python
#!/usr/bin/env python3
# auto_mode_interface.py
# Módulo principal - Interfaz de Modo Automático OPTIMIZADO

import sys
import time
import math
import threading
import numpy as np
import traceback
from PyQt5.QtWidgets import QApplication, QMainWindow, QMessageBox
from PyQt5.QtCore import QTimer, pyqtSlot

# Módulos del sistema
from rplidar_express import RPLidarExpress
from navigation_controller import NavigationController
from refined_mapper import RefinedMapper
from map_viewer import show_map_viewer

# Nuevos módulos específicos del automático
from trajectory_planner import TrajectoryPlanner
from obstacle_avoider import ObstacleAvoider

# Módulos propios
from auto_mode_gui import AutoModeGUI
from auto_mode_processor import AutoModeProcessor

class AutoModeInterface(QMainWindow):
    def __init__(self):
        super().__init__()

        # Inicializar GUI
        self.gui = AutoModeGUI()
        self.setCentralWidget(self.gui)
        self.setWindowTitle("Sistema de Planificación de Trayectoria - Modo Automático")
        self.setGeometry(50, 30, 1600, 900)

        # Inicializar procesador
        self.processor = AutoModeProcessor()

        # Variables para optimización
        self.map_viewer = None # Mantener referencia al visor de mapas

        # Conectar señales entre GUI y procesador
        self._connect_signals()

        # Inicializar sistemas
        self.init_systems()

        self.gui.log_message("Interfaz de modo automático inicializada")
        self.gui.log_message("Dibuje una trayectoria y luego ejecútela")

    def _connect_signals(self):
        """Conectar señales entre la GUI y el procesador"""
        # Señales de GUI a procesador
        self.gui.btn_start_drawing.clicked.connect(self.processor.toggle_drawing_mode)
        self.gui.btn_save_route.clicked.connect(self.processor.save_route)
        self.gui.btn_clear_trajectory.clicked.connect(self.processor.clear_trajectory)
        self.gui.btn_execute.clicked.connect(self.start_execution)
        self.gui.btn_stop_execution.clicked.connect(self.stop_execution)
        self.gui.btn_emergency_stop.clicked.connect(self.processor.emergency_stop_handler)
        self.gui.btn_view_maps.clicked.connect(self.show_map_viewer)

        # Señales de procesador a GUI
        self.processor.trajectory_updated.connect(self.gui.update_trajectory_display)
        self.processor.log_message_signal.connect(self.gui.log_message)
        self.processor.execution_state_changed.connect(self.update_execution_state)
        self.processor.obstacle_detected_signal.connect(self.update_obstacle_status)
        self.processor.drawing_status_changed.connect(self.gui.update_drawing_status)
        self.processor.execution_buttons_changed.connect(self.gui.update_execution_buttons)

        # Conectar eventos de gráficas
        self.gui.trajectory_canvas.mpl_connect('button_press_event', self.processor.on_trajectory_click)
        self.gui.trajectory_canvas.mpl_connect('motion_notify_event', self.processor.on_trajectory_drag)

    def init_systems(self):
        """Inicializar sistemas reales del robot"""
        try:
            self.gui.log_message("Inicializando controlador de navegación...")
            self.processor.nav_controller = NavigationController()
            self.gui.lbl_robot_status.setText("Robot: Conectado")
            self.gui.log_message("Controlador de navegación inicializado")

            self.gui.log_message("Inicializando sistema de mapeo MEJORADO...")
            self.processor.mapper = RefinedMapper(map_size=10.0, resolution=0.025)
            # Habilitar corrección de deriva por defecto
            self.processor.mapper.enable_correction(True)
            self.gui.log_message("Sistema de mapeo mejorado inicializado")
            self.gui.log_message("Corrección de deriva: ACTIVADA")

            # SISTEMAS ESPECÍFICOS DEL AUTOMÁTICO
            self.gui.log_message("Inicializando planificador de trayectoria...")
            self.processor.trajectory_planner = TrajectoryPlanner()
            self.gui.log_message("Planificador de trayectoria inicializado")

            self.gui.log_message("Inicializando sistema de evasión de obstáculos...")
            self.processor.obstacle_avoider = ObstacleAvoider()
            self.gui.log_message("Sistema de evasión de obstáculos inicializado")

            self.gui.log_message("Inicializando sensor LIDAR...")
            self.processor.init_lidar(self.gui.log_message)

            self.gui.log_message("Inicializando cámara...")
            self.processor.init_camera()

            # Conectar señales de cámara
            if hasattr(self.processor, 'camera_thread'):
                self.processor.camera_thread.frame_signal.connect(self.gui.update_camera_frame)
                self.processor.camera_thread.error_signal.connect(self.gui.handle_camera_error)

            # Timer para actualizaciones
            self.timer = QTimer()
            self.timer.timeout.connect(self.update_display)
            self.timer.start(100) # 10 Hz

            self.gui.log_message("Todos los sistemas inicializados correctamente")

        except Exception as e:
            self.gui.log_message(f"Error inicializando
```

```
                sistemas: {str(e)}")
            QMessageBox.critical(self, "Error de
                Inicialización",
                f"No se pudieron inicializar
                    todos los sistemas:\n{str
                    (e)}")

    def start_execution(self):
        """Iniciar ejecución de la trayectoria"""
        if not self.processor.has_saved_route:
            self.gui.log_message("ERROR: No hay ruta
                guardada para ejecutar")
            self.gui.log_message("Por favor, dibuje una
                trayectoria y guárdela primero")
            return

        if self.processor.is_executing:
            self.gui.log_message("Ejecución ya en curso")
            return

        # Iniciar sesión de mapeo en la pose actual
        if self.processor.nav_controller and self.processor.
            mapper:
            initial_pose = self.processor.nav_controller.
                get_current_pose()
            self.processor.mapper.start_mapping_session(
                initial_pose)
            self.gui.log_message(f"Sesión de mapeo iniciada
                en: ({initial_pose[0]:.2f}, {initial_pose
                [1]:.2f})")

        self.processor.start_execution(self.gui.log_message)

        self.gui.lbl_execution_state.setText("EJECUTANDO
            TRAYECTORIA")
        self.gui.lbl_execution_state.setStyleSheet("color:
            F57C00; background-color: #FFF3E0; padding: 8
            px; border-radius: 4px;")
        self.gui.lbl_mapping_status.setText("Mapeo: ACTIVO
            CON CORRECCIÓN")

    def stop_execution(self):
        """Detener ejecución de la trayectoria"""
        self.processor.stop_execution()

        self.gui.lbl_execution_state.setText("SISTEMA LISTO"
            )
        self.gui.lbl_execution_state.setStyleSheet("color:
            #2E7D32; background-color: #E8F5E8; padding: 8
            px; border-radius: 4px;")
        self.gui.lbl_mapping_status.setText("Mapeo: INACTIVO
            ")

    @pyqtSlot(str, str)
    def update_execution_state(self, state, style_sheet):
        """Actualizar estado de ejecución en la GUI"""
        self.gui.lbl_execution_state.setText(state)
        self.gui.lbl_execution_state.setStyleSheet(
            style_sheet)

    @pyqtSlot(bool)
    def update_obstacle_status(self, detected):
        """Actualizar estado de obstáculos en la GUI"""
        if detected:
            self.gui.lbl_obstacle_status.setText("Obstáculos
                : DETECTADOS - EVADIENDO")
            self.gui.lbl_obstacle_status.setStyleSheet("
                color: red; font-weight: bold;")
        else:
            self.gui.lbl_obstacle_status.setText("Obstáculos
                : No detectados")
            self.gui.lbl_obstacle_status.setStyleSheet("
                color: green;")

    def update_display(self):
        """Actualizar la interfaz con datos reales - VERSIÓN
            OPTIMIZADA SIN LIDAR"""
        try:
            # Actualizar información del robot
            if self.processor.nav_controller:
                x, y, yaw = self.processor.nav_controller.
                    get_current_pose()
                battery = self.processor.nav_controller.
                    get_battery_voltage()

                self.gui.lbl_position.setText(f"Posición: ({x
                    :.2f}, {y:.2f})")
                self.gui.lbl_orientation.setText(f"Orientació
                    n: {math.degrees(yaw):.1f}°")
                self.gui.lbl_battery.setText(f"Batería: {
                    battery:.1f} V")

                # Actualizar mapeo si está activo
                if self.processor.is_mapping and self.
                    processor.mapper:
                    points_mapped = self.processor.mapper.
                        total_points_processed
                    self.gui.lbl_points_mapped.setText(f"
                        Puntos mapeados: {points_mapped}")

                    # Actualizar estado de mapeo
                    if hasattr(self.processor.mapper, '
                        reference_pose') and self.processor
                        .mapper.reference_pose is not None:
                        self.gui.lbl_mapping_status.setText("
                            Mapeo: ACTIVO CON CORRECCIÓN")
                    else:
                        self.gui.lbl_mapping_status.setText("
                            Mapeo: ACTIVO")

            # Actualizar información LIDAR - SOLO ESTADO, NO
                GRÁFICA
            if (hasattr(self.processor, 'lidar') and
                self.processor.lidar and
                hasattr(self.processor.lidar, 'scanning') and
                self.processor.lidar.scanning):
                self.gui.lbl_lidar.setText("LIDAR: Escaneando
                    activo")
            else:
                self.gui.lbl_lidar.setText("LIDAR:
                    Desconectado o error")

            # ELIMINADA COMPLETAMENTE LA ACTUALIZACIÓN DE GRÁ
                FICA LIDAR

            # Actualizar tiempo de ejecución
            if self.processor.is_executing:
                elapsed_time = time.time() - self.processor.
                    execution_start_time
                minutes = int(elapsed_time // 60)
                seconds = int(elapsed_time % 60)
                self.gui.lbl_execution_time.setText(f"Tiempo
                    de ejecución: {minutes:02d}:{seconds
                    :02d}")

            # Actualizar estadísticas
            self.gui.lbl_obstacles_detected.setText(f"Obstá
                culos detectados: {self.processor.
                obstacles_detected_count}")
            self.gui.lbl_avoidance_maneuvers.setText(f"
                Maniobras de evasión: {self.processor.
                avoidance_maneuvers_count}")

            # Actualizar información de ejecución
            if self.processor.is_executing and self.
                processor.saved_trajectory:
                total_waypoints = len(self.processor.
                    saved_trajectory)
                current_index = self.processor.
                    current_waypoint_index

                self.gui.lbl_current_waypoint.setText(f"Punto
                    actual: {current_index + 1}/{
                    total_waypoints}")

                progress = int((current_index /
                    total_waypoints) * 100) if
                    total_waypoints > 0 else 0
                self.gui.lbl_progress.setText(f"Progreso: {
                    progress}%")

                # Calcular distancia al objetivo actual
                if current_index < total_waypoints:
                    target_x, target_y = self.processor.
                        saved_trajectory[current_index]
                    if self.processor.nav_controller:
```

```
                    current_x, current_y, _ = self.processor.nav_controller.get_current_pose()
                    distance = math.hypot(target_x - current_x, target_y - current_y)
                    self.gui.lbl_distance_to_target.setText(f"Distancia al objetivo: {distance:.2f} m")

            except Exception as e:
                print(f"Error actualizando display: {e}")

    def show_map_viewer(self):
        """Mostrar visor de mapas - VERSIÓN CORREGIDA"""
        try:
            # Reutilizar ventana existente en lugar de crear nueva
            if self.map_viewer is None or not self.map_viewer.isVisible():
                from map_viewer import MapViewer
                self.map_viewer = MapViewer()
                self.map_viewer.show()
            else:
                self.map_viewer.raise_() # Traer al frente si ya está abierto
                self.map_viewer.activateWindow()

            self.gui.log_message("Visor de mapas abierto")
        except Exception as e:
            self.gui.log_message(f"Error abriendo visor de mapas: {str(e)}")
            QMessageBox.critical(self, "Error", f"No se pudo abrir el visor de mapas:\n{str(e)}")

    def closeEvent(self, event):
        """Manejar cierre de la aplicación"""
        self.gui.log_message("Cerrando aplicación...")

        # Detener timers
        if hasattr(self, 'timer'):
            self.timer.stop()

        # Detener todos los procesos
        self.processor.cleanup()

        # Cerrar visor de mapas si está abierto
        if self.map_viewer and self.map_viewer.isVisible():
            self.map_viewer.close()

        self.gui.log_message("Sistema cerrado correctamente")
        event.accept()

def main():
    app = QApplication(sys.argv)
    app.setStyle('Fusion')

    try:
        window = AutoModeInterface()
        window.show()
        sys.exit(app.exec_())
    except Exception as e:
        print(f"Error fatal: {e}")
        QMessageBox.critical(None, "Error Fatal", f"El programa encontró un error:\n{str(e)}")
        sys.exit(1)

if __name__ == "__main__":
    main()
```

auto_mode_processor.py

```
#!/usr/bin/env python3
# auto_mode_processor.py
# Procesamiento y lógica del modo automático - CON MAPEO REAL OPTIMIZADO SIN VISUALIZACIÓN LIDAR

import time
import math
import threading
import traceback
import os
from PyQt5.QtCore import QThread, pyqtSignal, QObject
import numpy as np

# Módulos del sistema
from rplidar_express import RPLidarExpress
from navigation_controller import NavigationController
from refined_mapper import RefinedMapper
from trajectory_planner import TrajectoryPlanner
from obstacle_avoider import ObstacleAvoider

class CameraThread(QThread):
    """Hilo para captura de cámara en tiempo real"""
    frame_signal = pyqtSignal(np.ndarray)
    error_signal = pyqtSignal(str)

    def __init__(self, camera_index=0):
        super().__init__()
        self.camera_index = camera_index
        self.running = False
        self.cap = None

    def run(self):
        """Ejecutar captura de cámara"""
        try:
            import cv2
            print(f"Intentando abrir cámara con índice: {self.camera_index}")
            self.cap = cv2.VideoCapture(self.camera_index)

            if not self.cap.isOpened():
                for i in range(5):
                    self.cap = cv2.VideoCapture(i)
                    if self.cap.isOpened():
                        print(f"Cámara encontrada en índice: {i}")
                        break
                    self.cap.release()

            if not self.cap or not self.cap.isOpened():
                self.error_signal.emit("No se pudo abrir ninguna cámara")
                return

            self.cap.set(cv2.CAP_PROP_FRAME_WIDTH, 640)
            self.cap.set(cv2.CAP_PROP_FRAME_HEIGHT, 480)
            self.cap.set(cv2.CAP_PROP_FPS, 30)

            self.running = True
            print("Cámara inicializada correctamente")

            while self.running:
                ret, frame = self.cap.read()
                if ret:
                    self.frame_signal.emit(frame)
                else:
                    print("Error leyendo frame de cámara")
                    break
                time.sleep(0.033)

        except Exception as e:
            error_msg = f"Error en cámara: {str(e)}"
            print(error_msg)
```

```
            self.error_signal.emit(error_msg)
        finally:
            if self.cap:
                self.cap.release()
                print("Cámara liberada")

    def stop(self):
        """Detener captura de cámara"""
        self.running = False
        if self.isRunning():
            self.wait(2000)

class AutoModeProcessor(QObject):
    """Procesador principal para el modo automático - VERSIÓN OPTIMIZADA SIN VISUALIZACIÓN LIDAR"""

    # Señales
    trajectory_updated = pyqtSignal(list, list)
    log_message_signal = pyqtSignal(str)
    execution_state_changed = pyqtSignal(str, str)
    obstacle_detected_signal = pyqtSignal(bool)
    drawing_status_changed = pyqtSignal(bool, int)
    execution_buttons_changed = pyqtSignal(bool, bool)

    def __init__(self):
        super().__init__()

        # Sistemas
        self.nav_controller = None
        self.lidar = None
        self.mapper = None
        self.trajectory_planner = None
        self.obstacle_avoider = None

        # Estados
        self.is_mapping = False
        self.is_executing = False
        self.is_drawing = False
        self.has_saved_route = False
        self.obstacle_detected = False
        self.emergency_stop = False

        # Datos de trayectoria
        self.planned_trajectory = [] # Trayectoria dibujada
        self.saved_trajectory = [] # Trayectoria guardada para ejecución
        self.actual_trajectory = [] # Trayectoria real ejecutada
        self.current_waypoint_index = 0
        self.lidar_points = [] # Solo para detección de obstáculos, NO para visualización

        # Hilos
        self.lidar_thread = None
        self.lidar_running = False
        self.execution_thread = None
        self.camera_thread = None

        # Variables para estadísticas
        self.execution_start_time = 0
        self.obstacles_detected_count = 0
        self.avoidance_maneuvers_count = 0

        # Configuración de rendimiento
        self.last_mapping_log_time = 0
        self.mapping_log_interval = 5.0 # Segundos entre logs de mapeo

    def toggle_drawing_mode(self):
        """Activar/desactivar modo de dibujo"""
        self.is_drawing = not self.is_drawing

        if self.is_drawing:
            # Si ya hay una ruta guardada, limpiar para nueva ruta
            if self.has_saved_route:
                self.clear_trajectory()
            self.log_message_signal.emit("Modo dibujo ACTIVADO - Haga clic en el mapa para dibujar la trayectoria")
            self.log_message_signal.emit("Haga clic para establecer: 1) Punto inicial, 2) Waypoints intermedios, 3) Punto final")
        else:
            self.log_message_signal.emit("Modo dibujo DESACTIVADO")
            if len(self.planned_trajectory) >= 2:
                self.log_message_signal.emit(f"Trayectoria dibujada con {len(self.planned_trajectory)} puntos. Use 'GUARDAR RUTA' para confirmar.")

        # Actualizar estado en la GUI
        self.drawing_status_changed.emit(self.is_drawing, len(self.planned_trajectory))
        self.execution_buttons_changed.emit(self.has_saved_route, self.is_executing)

    def save_route(self):
        """Guardar la ruta dibujada para ejecución"""
        if len(self.planned_trajectory) < 2:
            self.log_message_signal.emit("ERROR: Se necesitan al menos 2 puntos (inicio y final) para guardar la ruta")
            return False

        self.saved_trajectory = self.planned_trajectory.copy()
        self.has_saved_route = True

        self.log_message_signal.emit(f"RUTA GUARDADA EXITOSAMENTE")
        self.log_message_signal.emit(f"- Punto inicial: ({self.saved_trajectory[0][0]:.2f}, {self.saved_trajectory[0][1]:.2f})")
        self.log_message_signal.emit(f"- Punto final: ({self.saved_trajectory[-1][0]:.2f}, {self.saved_trajectory[-1][1]:.2f})")
        self.log_message_signal.emit(f"- Waypoints intermedios: {len(self.saved_trajectory) - 2}")

        # Actualizar botones
        self.execution_buttons_changed.emit(self.has_saved_route, self.is_executing)

        return True

    def on_trajectory_click(self, event):
        """Manejar clic en la gráfica de trayectoria"""
        if self.is_drawing and event.inaxes:
            x, y = event.xdata, event.ydata
            if x is not None and y is not None:
                # Agregar punto a la trayectoria
                self.planned_trajectory.append((x, y))

                # Actualizar visualización
                self.trajectory_updated.emit(self.planned_trajectory, self.actual_trajectory)

                # Determinar tipo de punto
                point_count = len(self.planned_trajectory)
                if point_count == 1:
                    point_type = "PUNTO INICIAL"
                elif point_count >= 2:
                    point_type = "PUNTO FINAL" if point_count == 2 else f"WAYPOINT {point_count-1}"

                self.log_message_signal.emit(f"{point_type} agregado: ({x:.2f}, {y:.2f})")

                # Actualizar estado de dibujo
                self.drawing_status_changed.emit(self.is_drawing, point_count)

                if point_count >= 2:
                    self.execution_buttons_changed.emit(False, self.is_executing)

    def on_trajectory_drag(self, event):
        """Manejar arrastre en la gráfica de trayectoria - Deshabilitado para control preciso"""
        pass # Deshabilitamos el arrastre para tener control preciso de los puntos
```

```python

    def clear_trajectory(self):
        """Borrar␣trayectoria␣actual␣completamente"""
        self.planned_trajectory.clear()
        self.saved_trajectory.clear()
        self.actual_trajectory.clear()
        self.has_saved_route = False
        self.current_waypoint_index = 0

        # Actualizar visualización
        self.trajectory_updated.emit(self.planned_trajectory, self.actual_trajectory)

        # Actualizar estados
        self.drawing_status_changed.emit(self.is_drawing, 0)
        self.execution_buttons_changed.emit(False, self.is_executing)

        self.log_message_signal.emit("TRAYECTORIA␣BORRADA␣COMPLETAMENTE")
        self.log_message_signal.emit("Lienzo␣listo␣para␣nueva␣trayectoria")

    def emergency_stop_handler(self):
        """Manejar␣paro␣de␣emergencia"""
        try:
            self.emergency_stop = True
            self.is_executing = False
            self.is_drawing = False
            self.is_mapping = False

            # Detener robot inmediatamente
            if self.nav_controller and self.nav_controller.bot:
                self.nav_controller.bot.set_car_motion(0, 0)

            # Actualizar estado
            self.execution_state_changed.emit("PARO␣DE␣EMERGENCIA",
                                              "color:␣#D32F2F;␣background-color:␣#FFEBEE;␣padding:␣8px;␣border-radius:␣4px;")

            # Actualizar botones
            self.execution_buttons_changed.emit(self.has_saved_route, False)
            self.drawing_status_changed.emit(False, len(self.planned_trajectory))

            self.log_message_signal.emit("PARO␣DE␣EMERGENCIA␣ACTIVADO␣-␣Sistema␣detenido")
            self.log_message_signal.emit("Todos␣los␣procesos␣han␣sido␣interrumpidos")

            # Reiniciar contadores
            self.emergency_stop = False
            self.obstacles_detected_count = 0
            self.avoidance_maneuvers_count = 0

        except Exception as e:
            self.log_message_signal.emit(f"Error␣en␣paro␣de␣emergencia:␣{str(e)}")

    def init_lidar(self, log_callback):
        """Inicializar␣LIDAR␣-␣VERSIÓN␣OPTIMIZADA␣SIN␣VISUALIZACIÓN"""
        def lidar_processing_thread():
            try:
                # ELIMINADO os.nice(5) - No se necesita ajuste de prioridad
                self.lidar = RPLidarExpress(port='/dev/ttyUSB0', baudrate=460800)
                if self.lidar.start_express_scan():
                    scan_gen = self.lidar.iter_packets()
                    self.lidar_running = True
                    log_callback("LIDAR␣inicializado␣correctamente")

                    while self.lidar_running and self.lidar.scanning:
                        try:
                            # CAPTURA RÁPIDA solo para mapeo y detección de obstáculos
                            all_points = []
                            start_time = time.time()
                            packets_collected = 0
                            max_packets = 128 # Reducido para mayor responsividad
                            timeout_seconds = 0.03 # Timeout más corto

                            while (packets_collected < max_packets and
                                   (time.time() - start_time) < timeout_seconds and
                                   self.lidar_running):
                                try:
                                    points = next(scan_gen, None)
                                    if points:
                                        all_points.extend(points)
                                        packets_collected += 1
                                except StopIteration:
                                    scan_gen = self.lidar.iter_packets()
                                    break
                                except Exception as e:
                                    break

                            if all_points:
                                # ACTUALIZACIÓN SOLO PARA DATOS OPERATIVOS, NO VISUALIZACIÓN
                                self.lidar_points = all_points

                                # MAPEO en hilo separado si está activo
                                if self.is_mapping and self.nav_controller and self.mapper:
                                    # Usar threading para no bloquear
                                    mapping_thread = threading.Thread(
                                        target=self.process_mapping_data,
                                        args=(all_points,),
                                        daemon=True
                                    )
                                    mapping_thread.start()

                                # DETECCIÓN DE OBSTÁCULOS en hilo separado
                                if self.is_executing and self.obstacle_avoider:
                                    obstacle_thread = threading.Thread(
                                        target=self.process_obstacle_detection,
                                        args=(all_points,),
                                        daemon=True
                                    )
                                    obstacle_thread.start()

                            time.sleep(0.01) # Pausa mínima para mejor respuesta

                        except Exception as e:
                            if self.lidar_running:
                                log_callback(f"Error␣en␣hilo␣LIDAR:␣{str(e)}")
                            time.sleep(0.05)

            except Exception as e:
                log_callback(f"Error␣inicializando␣LIDAR:␣{str(e)}")

        self.lidar_thread = threading.Thread(target=
```

```
            lidar_processing_thread, daemon=True)
        self.lidar_thread.start()

    def process_mapping_data(self, lidar_points):
        """Procesar␣datos␣de␣mapeo␣en␣hilo␣separado"""
        try:
            if self.is_mapping and self.nav_controller and
                self.mapper:
                robot_x, robot_y, robot_yaw = self.
                    nav_controller.get_current_pose()
                points_processed = self.mapper.update_map(
                    robot_x, robot_y, robot_yaw,
                    lidar_points)

                # Log periódico para no saturar la interfaz
                current_time = time.time()
                if (points_processed > 0 and
                    current_time - self.last_mapping_log_time
                        > self.mapping_log_interval):
                    self.log_message_signal.emit(f"␣Puntos␣
                        mapeados:␣{self.mapper.
                        total_points_processed}")
                    self.last_mapping_log_time = current_time

        except Exception as e:
            print(f"Error␣en␣procesamiento␣de␣mapeo:␣{e}")

    def process_obstacle_detection(self, lidar_points):
        """Procesar␣detección␣de␣obstáculos␣en␣hilo␣separado
            """
        try:
            if self.is_executing and self.obstacle_avoider
                and self.nav_controller:
                current_pose = self.nav_controller.
                    get_current_pose()
                obstacle_detected = self.obstacle_avoider.
                    check_obstacles(lidar_points,
                    current_pose)

                if obstacle_detected and not self.
                    obstacle_detected:
                    self.obstacle_detected = True
                    self.obstacles_detected_count += 1
                    self.obstacle_detected_signal.emit(True)
                    self.log_message_signal.emit("␣OBSTÁCULO␣
                        DETECTADO␣-␣Iniciando␣evasión")

                elif not obstacle_detected and self.
                    obstacle_detected:
                    self.obstacle_detected = False
                    self.obstacle_detected_signal.emit(False)
                    self.log_message_signal.emit("␣OBSTÁCULO␣
                        SUPERADO␣-␣Volviendo␣a␣trayectoria"
                        )

        except Exception as e:
            print(f"Error␣en␣detección␣de␣obstáculos:␣{e}")

    def init_camera(self):
        """Inicializar␣cámara␣en␣hilo␣separado"""
        try:
            self.camera_thread = CameraThread(camera_index
                =0)
            self.camera_thread.start()
        except Exception as e:
            self.log_message_signal.emit(f"Error␣iniciando␣c
                ámara:␣{str(e)}")

    def start_execution(self, log_callback):
        """Iniciar␣ejecución␣de␣la␣trayectoria␣con␣mapeo␣
            real"""
        if not self.has_saved_route or not self.
            saved_trajectory:
            log_callback("ERROR:␣No␣hay␣ruta␣guardada␣para␣
                ejecutar")
            return

        if self.is_executing:
            log_callback("Ejecución␣ya␣en␣curso")
            return

        self.is_executing = True
        self.is_mapping = True # INICIAR MAPEO AUTOMÁ
            TICAMENTE
        self.obstacle_detected = False
        self.current_waypoint_index = 0
        self.actual_trajectory.clear()

        # Iniciar temporizador y contadores
        self.execution_start_time = time.time()
        self.obstacles_detected_count = 0
        self.avoidance_maneuvers_count = 0

        # INICIAR SESIÓN DE MAPEO REAL
        if self.nav_controller and self.mapper:
            initial_pose = self.nav_controller.
                get_current_pose()
            self.mapper.start_mapping_session(initial_pose)
            log_callback(f"Sesión␣de␣mapeo␣iniciada␣en:␣({
                initial_pose[0]:.2f},␣{initial_pose[1]:.2f
                })")
            log_callback(f"Corrección␣de␣deriva:␣ACTIVADA")

        # Actualizar botones
        self.execution_buttons_changed.emit(self.
            has_saved_route, True)

        log_callback("INICIANDO␣EJECUCIÓN␣DE␣TRAYECTORIA␣CON
            ␣MAPEO␣REAL")
        log_callback(f"Puntos␣a␣seguir:␣{len(self.
            saved_trajectory)}")
        log_callback(f"Desde:␣({self.saved_trajectory
            [0][0]:.2f},␣{self.saved_trajectory[0][1]:.2f
            })")
        log_callback(f"Hasta:␣({self.saved_trajectory
            [-1][0]:.2f},␣{self.saved_trajectory[-1][1]:.2
            f})")
        log_callback("MAPEO␣AUTOMÁTICO␣ACTIVADO␣CON␣CORRECCI
            ÓN␣DE␣DERIVA")

        # Iniciar hilo de ejecución
        self.execution_thread = threading.Thread(target=self
            .execute_trajectory, daemon=True)
        self.execution_thread.start()

    def stop_execution(self):
        """Detener␣ejecución␣de␣la␣trayectoria"""
        self.is_executing = False
        self.is_mapping = False # DETENER MAPEO

        # Guardar mapa automáticamente al detener
        self.save_map_after_execution()

        # Detener robot
        if self.nav_controller and self.nav_controller.bot:
            self.nav_controller.bot.set_car_motion(0, 0)

        # Actualizar botones
        self.execution_buttons_changed.emit(self.
            has_saved_route, False)

        self.log_message_signal.emit("EJECUCIÓN␣DETENIDA")
        self.log_message_signal.emit("MAPEO␣DETENIDO␣-␣Mapa␣
            guardado␣automáticamente")

    def execute_trajectory(self):
        """Ejecutar␣la␣trayectoria␣-␣VERSIÓN␣CORREGIDA␣CON␣
            MAPEO␣REAL"""
        try:
            waypoints = self.saved_trajectory
            total_waypoints = len(waypoints)

            self.log_message_signal.emit(f"Iniciando␣ejecuci
                ón␣con␣{total_waypoints}␣puntos")

            for i, (target_x, target_y) in enumerate(
                waypoints):
                if not self.is_executing or self.
                    emergency_stop:
                    self.log_message_signal.emit("Ejecución␣
                        interrumpida␣por␣usuario")
                    break

                self.current_waypoint_index = i

                # Determinar tipo de punto
```

```
            if i == 0:
                point_type = "INICIO"
                is_final = False
            elif i == total_waypoints - 1:
                point_type = "FINAL"
                is_final = True
            else:
                point_type = f"WAYPOINT {i}"
                is_final = False

            self.log_message_signal.emit(f"Navegando a punto {i+1}/{total_waypoints} ({point_type})")
            self.log_message_signal.emit(f"   Coordenadas: ({target_x:.3f}, {target_y:.3f})")

            # Navegar al waypoint actual CON INFORMACIÓN DE PUNTO FINAL
            success = self.navigate_to_waypoint(target_x, target_y, is_final_point=is_final)

            if not success and self.is_executing:
                self.log_message_signal.emit(f"ERROR navegando al punto {i+1}")

                # Detener ejecución si falla un punto crítico
                if i == total_waypoints - 1: # Si es el punto final
                    self.log_message_signal.emit("NO SE PUDO ALCANZAR EL PUNTO FINAL")
                break

            # Pequeña pausa entre puntos (excepto después del final)
            if not is_final:
                time.sleep(0.2)

        # VERIFICACIÓN FINAL MEJORADA
        if self.is_executing and not self.emergency_stop:
            # Verificar si realmente llegamos al punto final
            if waypoints:
                final_x, final_y = waypoints[-1]
                current_x, current_y, _ = self.nav_controller.get_current_pose()
                final_distance = math.hypot(final_x - current_x, final_y - current_y)

                if final_distance <= 0.1: # 10cm de tolerancia
                    self.log_message_signal.emit("TRAYECTORIA COMPLETADA EXITOSAMENTE")
                    self.log_message_signal.emit(f"   Distancia al punto final: {final_distance:.3f}m")
                    self.log_message_signal.emit("   Robot ha llegado al destino exacto")
                else:
                    self.log_message_signal.emit("Trayectoria completada con imprecisiones")
                    self.log_message_signal.emit(f"   Distancia al punto final: {final_distance:.3f}m")

            # Guardar mapa
            self.save_map_after_execution()

            self.execution_state_changed.emit("TRAYECTORIA COMPLETADA", "color: #2E7D32; background-color: #E8F5E8; padding: 8px; border-radius: 4px;")

    except Exception as e:
        error_msg = f"ERROR CRÍTICO en ejecución: {str(e)}\n{traceback.format_exc()}"
        self.log_message_signal.emit(error_msg)
        print("="*50)
        print("ERROR CRÍTICO:")
        print(error_msg)
        print("="*50)

        # Asegurar parada en caso de error
        self.is_executing = False
        self.is_mapping = False
        if self.nav_controller and self.nav_controller.bot:
            self.nav_controller.bot.set_car_motion(0, 0)
    finally:
        # GARANTIZAR DETENCIÓN COMPLETA
        self.is_executing = False
        self.is_mapping = False
        self.execution_buttons_changed.emit(self.has_saved_route, False)

        # DETENER ROBOT POR ÚLTIMA VEZ
        if self.nav_controller and self.nav_controller.bot:
            self.nav_controller.bot.set_car_motion(0, 0)

def save_map_after_execution(self):
    """Guardar mapa después de la ejecución usando el sistema real"""
    try:
        if self.mapper and self.mapper.total_points_processed > 0:
            timestamp = time.strftime("%Y%m%d_%H%M%S")
            success = self.mapper.save_data(f"auto_map_{timestamp}")
            if success:
                self.log_message_signal.emit(f"MAPA GUARDADO: auto_map_{timestamp}")
                self.log_message_signal.emit(f"   Puntos procesados: {self.mapper.total_points_processed}")
                self.log_message_signal.emit(f"   Trayectoria guardada: {len(self.mapper.trajectory)} puntos")
            else:
                self.log_message_signal.emit("ERROR guardando mapa")
    except Exception as e:
        self.log_message_signal.emit(f"ERROR guardando mapa: {str(e)}")

def navigate_to_waypoint(self, target_x, target_y, is_final_point=False):
    """Navegar a un waypoint específico - VERSIÓN CORREGIDA CON MAPEO REAL"""
    try:
        # TOLERANCIAS MEJORADAS
        if is_final_point:
            pos_tolerance = 0.05 # 5cm para punto final
            min_speed = 0.04 # Velocidad mínima más baja para precisión
        else:
            pos_tolerance = 0.08 # 8cm para waypoints intermedios
            min_speed = 0.04

        max_attempts = 300 # Aumentado para dar más tiempo
        attempt = 0
        distance = float('inf') # Inicializar variable

        self.log_message_signal.emit(f"   Navegando a ({target_x:.2f}, {target_y:.2f}) - {'FINAL' if is_final_point else 'INTERMEDIO'}")

        consecutive_close_points = 0 # Contador para puntos muy cercanos

        while self.is_executing and not self.emergency_stop and attempt < max_attempts:
            # Obtener pose actual
            try:
```

```
            current_x, current_y, current_yaw = self.
                nav_controller.get_current_pose()
        except Exception as e:
            self.log_message_signal.emit(f"␣␣ERROR␣
                obteniendo␣pose:␣{str(e)}")
            time.sleep(0.1)
            attempt += 1
            continue

        # Guardar trayectoria actual
        self.actual_trajectory.append((current_x,
            current_y, current_yaw))
        self.trajectory_updated.emit(self.
            saved_trajectory, self.
            actual_trajectory)

        # Calcular distancia al objetivo
        prev_distance = distance
        distance = math.hypot(target_x - current_x,
            target_y - current_y)

        # VERIFICACIÓN MEJORADA DE LLEGADA
        if distance < pos_tolerance:
            self.log_message_signal.emit(f"␣␣PUNTO␣
                ALCANZADO␣-␣Distancia:␣{distance:.3
                f}m")

            # DETENER ROBOT INMEDIATAMENTE
            if self.nav_controller and self.
                nav_controller.bot:
                self.nav_controller.bot.set_car_motion
                    (0, 0)

            time.sleep(0.5) # Pequeña pausa para
                estabilización
            return True

        # DETECCIÓN DE ESTANCAMIENTO - si no nos
            estamos acercando
        if abs(distance - prev_distance) < 0.01 and
            distance < 0.03:
            consecutive_close_points += 1
            if consecutive_close_points > 10: # Si
                está estancado cerca del objetivo
                self.log_message_signal.emit(f"
                    Estancamiento␣detectado,␣
                    forzando␣llegada␣(dist:␣{
                    distance:.3f}m)")
                if self.nav_controller and self.
                    nav_controller.bot:
                    self.nav_controller.bot.
                        set_car_motion(0, 0)
                return True
        else:
            consecutive_close_points = 0

        # CÁLCULO DE CONTROL MEJORADO
        target_angle = math.atan2(target_y -
            current_y, target_x - current_x)
        angle_error = target_angle - current_yaw

        # Normalizar ángulo (-pi a pi)
        while angle_error > math.pi:
            angle_error -= 2 * math.pi
        while angle_error < -math.pi:
            angle_error += 2 * math.pi

        # CONTROL DE VELOCIDAD ADAPTATIVO
        base_speed = 0.3 if not is_final_point else
            0.04

        # Reducir velocidad cuando estamos cerca del
            objetivo
        speed_reduction = min(1.5, distance / 0.5) #
            Reducir progresivamente
        base_speed *= speed_reduction

        # Velocidad angular proporcional al error
        angular_speed = 2.0 * angle_error # Control P

        # LIMITES MÁS CONSERVADORES
        angular_speed = max(-0.3, min(0.3,
            angular_speed))
        actual_speed = max(min_speed, base_speed * (1
            - min(1.0, abs(angle_error) / (math.
            pi/3))))

        # Reducir velocidad extra para punto final
        if is_final_point and distance < 0.2:
            actual_speed *= 0.5

        # Evasión de obstáculos (si está activada)
        if self.obstacle_detected and self.
            obstacle_avoider:
            actual_speed, angular_speed = self.
                obstacle_avoider.avoid_obstacle(
                self.lidar_points, (current_x,
                    current_y, current_yaw),
                (target_x, target_y), actual_speed,
                    angular_speed
            )
            self.avoidance_maneuvers_count += 1

        # MOVER ROBOT
        if self.nav_controller and self.
            nav_controller.bot:
            try:
                self.nav_controller.bot.set_car_motion
                    (actual_speed, angular_speed)
            except Exception as e:
                self.log_message_signal.emit(f"␣␣ERROR
                    ␣moviendo␣robot:␣{str(e)}")
                time.sleep(0.1)
                attempt += 1
                continue

        attempt += 1
        time.sleep(0.1) # Control loop a 10Hz

    # MANEJO DE TIMEOUT
    if attempt >= max_attempts:
        self.log_message_signal.emit(f"␣␣TIMEOUT␣-␣No
            ␣se␣pudo␣alcanzar␣el␣punto")
        self.log_message_signal.emit(f"␣␣␣␣␣Intentos:
            ␣{attempt},␣Distancia␣final:␣{distance
            :.3f}m")

        # Detener robot en caso de timeout
        if self.nav_controller and self.
            nav_controller.bot:
            self.nav_controller.bot.set_car_motion(0,
                0)

    return attempt < max_attempts

except Exception as e:
    error_msg = f"ERROR␣en␣navegación:␣{str(e)}\n{
        traceback.format_exc()}"
    self.log_message_signal.emit(error_msg)
    print("ERROR␣en␣navigate_to_waypoint:")
    print(error_msg)

    # Asegurar que el robot se detenga en caso de
        error
    if self.nav_controller and self.nav_controller.
        bot:
        self.nav_controller.bot.set_car_motion(0, 0)

    return False

def cleanup(self):
    """Limpiar␣recursos"""
    # Detener ejecución
    self.is_executing = False
    self.is_mapping = False
    self.emergency_stop = True

    # Guardar mapa final si hay datos
    if self.mapper and self.mapper.
        total_points_processed > 0:
        try:
            timestamp = time.strftime("%Y%m%d_%H%M%S")
            self.mapper.save_data(f"final_map_{timestamp}
                ")
            self.log_message_signal.emit(f"␣Mapa␣final␣
                guardado:␣final_map_{timestamp}")
```

```
            except Exception as e:
                self.log_message_signal.emit(f"␣Error␣guardando␣mapa␣final:␣{str(e)}")

        # Detener LIDAR
        self.lidar_running = False
        if self.lidar:
            try:
                self.lidar.stop_scan()
                self.lidar.close()
            except Exception as e:
                print(f"Error␣cerrando␣LIDAR:␣{e}")

        # Detener cámara
        if self.camera_thread and self.camera_thread.isRunning():
            self.camera_thread.stop()

        # Detener robot
        if self.nav_controller and self.nav_controller.bot:
            try:
                self.nav_controller.bot.set_car_motion(0, 0)
            except Exception as e:
                print(f"Error␣deteniendo␣robot:␣{e}")
```

manual_mode_ui.py

```
#!/usr/bin/env python3
# manual_mode_ui.py
# Diseño de la interfaz de usuario para modo manual - SIN VISUALIZACIÓN LIDAR

import time
import cv2
from PyQt5.QtWidgets import (QVBoxLayout, QHBoxLayout, QPushButton, QWidget,
                             QLabel, QGroupBox, QGridLayout, QTextEdit)
from PyQt5.QtCore import Qt, pyqtSignal
from PyQt5.QtGui import QFont, QPixmap, QImage
import matplotlib.pyplot as plt
from matplotlib.backends.backend_qt5agg import FigureCanvasQTAgg as FigureCanvas
from matplotlib.figure import Figure
import numpy as np

class ManualModeUI(QWidget):
    # Señales para comunicación con el procesador
    camera_frame_signal = pyqtSignal(np.ndarray)
    camera_error_signal = pyqtSignal(str)

    def __init__(self):
        super().__init__()
        self.trajectory_x = []
        self.trajectory_y = []
        self.init_ui()
        self.init_plots()

    def init_ui(self):
        main_layout = QHBoxLayout(self)
        main_layout.setSpacing(10)
        main_layout.setContentsMargins(10, 10, 10, 10)

        # ===== PANEL IZQUIERDO - CONTROLES =====
        left_panel = QVBoxLayout()
        left_panel.setSpacing(10)

        # CONTROLES DEL ROBOT
        controls_group = QGroupBox("CONTROLES␣DEL␣ROBOT")
        controls_group.setFont(QFont("Arial", 12, QFont.Bold))
        controls_group.setMinimumWidth(500)

        controls_layout = QGridLayout()
        controls_layout.setSpacing(10)

        self.btn_forward = self.create_control_button("AVANZAR", "#4CAF50")
        self.btn_left = self.create_control_button("IZQUIERDA", "#2196F3")
        self.btn_right = self.create_control_button("DERECHA", "#2196F3")
        self.btn_backward = self.create_control_button("ATRÁS", "#f44336")
        self.btn_stop = self.create_control_button("DETENER", "#FF9800")

        controls_layout.addWidget(self.btn_forward, 0, 1)
        controls_layout.addWidget(self.btn_left, 1, 0)
        controls_layout.addWidget(self.btn_stop, 1, 1)
        controls_layout.addWidget(self.btn_right, 1, 2)
        controls_layout.addWidget(self.btn_backward, 2, 1)

        self.btn_generate_map = QPushButton("INICIAR␣MAPEO")
        self.btn_generate_map.setMinimumHeight(50)
        self.btn_generate_map.setFont(QFont("Arial", 11, QFont.Bold))
        self.btn_generate_map.setStyleSheet("""
␣␣␣␣␣␣␣␣␣␣␣␣QPushButton␣{
␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣background-color:␣#4CAF50;
␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣color:␣white;
␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣border:␣none;
␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣border-radius:␣8px;
␣␣␣␣␣␣␣␣␣␣␣␣}
␣␣␣␣␣␣␣␣␣␣␣␣QPushButton:hover␣{
␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣background-color:␣#45a049;
␣␣␣␣␣␣␣␣␣␣␣␣}
␣␣␣␣␣␣␣␣""")
        controls_layout.addWidget(self.btn_generate_map, 3, 0, 1, 3)

        # Botón para ver mapas
        self.btn_view_maps = QPushButton("␣VER␣MAPAS␣GUARDADOS")
        self.btn_view_maps.setMinimumHeight(40)
        self.btn_view_maps.setFont(QFont("Arial", 10, QFont.Bold))
        self.btn_view_maps.setStyleSheet("""
␣␣␣␣␣␣␣␣␣␣␣␣QPushButton␣{
␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣background-color:␣#FF9800;
␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣color:␣white;
␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣border:␣none;
␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣border-radius:␣8px;
␣␣␣␣␣␣␣␣␣␣␣␣}
␣␣␣␣␣␣␣␣␣␣␣␣QPushButton:hover␣{
␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣background-color:␣#F57C00;
␣␣␣␣␣␣␣␣␣␣␣␣}
␣␣␣␣␣␣␣␣""")
        controls_layout.addWidget(self.btn_view_maps, 4, 0, 1, 3)

        controls_group.setLayout(controls_layout)
        left_panel.addWidget(controls_group)

        # INFORMACIÓN DEL SISTEMA
        info_group = QGroupBox("INFORMACIÓN␣DEL␣SISTEMA")
        info_group.setFont(QFont("Arial", 10, QFont.Bold))

        info_layout = QVBoxLayout()

        self.lbl_robot_status = QLabel("Robot:␣Conectando...")
        self.lbl_position = QLabel("Posición:␣(0.00,␣0.00)")
        self.lbl_orientation = QLabel("Orientación:␣0.00°")
        self.lbl_battery = QLabel("Batería:␣--␣V")
```

```
        self.lbl_lidar = QLabel("LIDAR: Conectando...")
        self.lbl_camera_status = QLabel("Cámara:
              Inicializando...")
        self.lbl_camera_angle = QLabel("Ángulo cámara: 90°")
        self.lbl_points = QLabel("Puntos mapeados: 0")
        self.lbl_mapping_status = QLabel("Mapeo: INACTIVO")

        for lbl in [self.lbl_robot_status, self.lbl_position
              , self.lbl_orientation,
                    self.lbl_battery, self.lbl_lidar, self.
                          lbl_camera_status,
                    self.lbl_camera_angle, self.lbl_points,
                          self.lbl_mapping_status]:
            lbl.setFont(QFont("Arial", 9))
            info_layout.addWidget(lbl)

        info_group.setLayout(info_layout)
        left_panel.addWidget(info_group)

        # REGISTRO DE EVENTOS
        log_group = QGroupBox("REGISTRO DE EVENTOS")
        log_group.setFont(QFont("Arial", 10, QFont.Bold))

        log_layout = QVBoxLayout()
        self.txt_log = QTextEdit()
        self.txt_log.setMaximumHeight(200)
        self.txt_log.setReadOnly(True)
        self.txt_log.setFont(QFont("Arial", 8))
        log_layout.addWidget(self.txt_log)

        log_group.setLayout(log_layout)
        left_panel.addWidget(log_group)

        # ===== PANEL DERECHO - GRÁFICAS =====
        right_panel = QVBoxLayout()
        right_panel.setSpacing(10)

        # TRAYECTORIA DEL ROBOT (OCUPANDO TODO EL ESPACIO
              LIBERADO POR LIDAR)
        traj_group = QGroupBox("TRAYECTORIA DEL ROBOT EN
              TIEMPO REAL")
        traj_group.setFont(QFont("Arial", 12, QFont.Bold))
        traj_group.setMinimumHeight(600) # Más alto para
              ocupar el espacio del LIDAR eliminado

        traj_layout = QVBoxLayout()
        self.trajectory_figure = Figure(figsize=(12, 6)) #
              Figura más grande
        self.trajectory_canvas = FigureCanvas(self.
              trajectory_figure)
        traj_layout.addWidget(self.trajectory_canvas)
        traj_group.setLayout(traj_layout)

        right_panel.addWidget(traj_group)

        # CÁMARA DEL ROBOT CON CONTROLES
        camera_group = QGroupBox("CÁMARA DEL ROBOT")
        camera_group.setFont(QFont("Arial", 12, QFont.Bold))
        camera_group.setMinimumHeight(350)

        camera_layout = QVBoxLayout()

        # Controles de la cámara
        camera_controls_layout = QHBoxLayout()

        self.btn_camera_up = QPushButton("Cámara Arriba")
        self.btn_camera_down = QPushButton("Cámara Abajo")
        self.btn_camera_center = QPushButton("Centro")

        for btn in [self.btn_camera_up, self.btn_camera_down
              , self.btn_camera_center]:
            btn.setMinimumHeight(35)
            btn.setFont(QFont("Arial", 9))
            btn.setStyleSheet("""
        QPushButton {
            background-color: #607D8B;
            color: white;
            border: none;
            border-radius: 5px;
        }
        QPushButton:hover {
            background-color: #455A64;
        }
    """)

        camera_controls_layout.addWidget(self.btn_camera_up)
        camera_controls_layout.addWidget(self.
              btn_camera_center)
        camera_controls_layout.addWidget(self.
              btn_camera_down)

        camera_layout.addLayout(camera_controls_layout)

        # Video de la cámara
        self.lbl_camera = QLabel("Cámara no disponible")
        self.lbl_camera.setAlignment(Qt.AlignCenter)
        self.lbl_camera.setMinimumSize(480, 270)
        self.lbl_camera.setStyleSheet("""
        QLabel {
            background-color: #2C2C2C;
            color: #CCCCCC;
            border: 2px solid #666;
            font-size: 14px;
        }
    """)
        camera_layout.addWidget(self.lbl_camera)

        camera_group.setLayout(camera_layout)

        right_panel.addWidget(camera_group)

        # ENSAMBLAR INTERFAZ
        main_layout.addLayout(left_panel, 1)
        main_layout.addLayout(right_panel, 2)

        self.log_message("Interfaz de modo manual
              inicializada")
        self.log_message("El mapeo está siempre activo. Use
              INICIAR/DETENER MAPEO para guardar")

    def create_control_button(self, text, color):
        button = QPushButton(text)
        button.setMinimumSize(120, 80)
        button.setFont(QFont("Arial", 10, QFont.Bold))
        button.setStyleSheet(f"""
        QPushButton {{
            background-color: {color};
            color: white;
            border: none;
            border-radius: 8px;
            font-size: 12px;
        }}
        QPushButton:hover {{
            background-color: {self.darken_color(color)
              };
        }}
    """)
        return button

    def darken_color(self, color, amount=20):
        if color.startswith('#'):
            color = color[1:]
        r = max(0, int(color[0:2], 16) - amount)
        g = max(0, int(color[2:4], 16) - amount)
        b = max(0, int(color[4:6], 16) - amount)
        return f"#{r:02x}{g:02x}{b:02x}"

    def init_plots(self):
        """Inicializar gráficas - SIN GRÁFICA LIDAR"""
        # Gráfica de trayectoria - más grande y detallada
        self.traj_ax = self.trajectory_figure.add_subplot
              (111)
        self.traj_ax.set_xlabel('X (metros)', fontsize=12)
        self.traj_ax.set_ylabel('Y (metros)', fontsize=12)
        self.traj_ax.grid(True, alpha=0.3)
        self.traj_ax.axis('equal')
        self.traj_line, = self.traj_ax.plot([], [], 'b-',
              linewidth=2, alpha=0.7)
        self.traj_start = self.traj_ax.plot([], [], 'go',
              markersize=10, label='Inicio')[0]
        self.traj_current = self.traj_ax.plot([], [], 'ro',
              markersize=8, label='Actual')[0]
        self.traj_ax.legend(fontsize=10)
        self.traj_ax.set_title('Trayectoria del Robot en
              Tiempo Real', fontsize=14, fontweight='bold')
```

```
        self.trajectory_canvas.draw()

    def update_status(self, status_data):
        """Actualizar información de estado del sistema"""
        try:
            self.lbl_robot_status.setText(f"Robot: {
                status_data.get('robot_status', '
                Desconocido')}")
            self.lbl_position.setText(f"Posición: ({
                status_data.get('x', 0):.2f}, {status_data
                .get('y', 0):.2f})")
            self.lbl_orientation.setText(f"Orientación: {
                status_data.get('yaw_deg', 0):.1f}°")
            self.lbl_battery.setText(f"Batería: {status_data
                .get('battery', 0):.1f} V")
            self.lbl_lidar.setText(f"LIDAR: {status_data.get
                ('lidar_status', 'Desconocido')}")
            self.lbl_camera_status.setText(f"Cámara: {
                status_data.get('camera_status', '
                Desconocido')}")
            self.lbl_camera_angle.setText(f"Ángulo cámara: {
                status_data.get('camera_angle', 90)}°")
            self.lbl_points.setText(f"Puntos mapeados: {
                status_data.get('points_mapped', 0)}")
            self.lbl_mapping_status.setText(f"Mapeo: {
                status_data.get('mapping_status', '
                INACTIVO')}")
        except Exception as e:
            print(f"Error actualizando estado: {e}")

    def update_trajectory(self, trajectory_data):
        """Actualizar gráfica de trayectoria"""
        try:
            x = trajectory_data.get('x', 0)
            y = trajectory_data.get('y', 0)

            # Actualizar trayectoria
            if len(self.trajectory_x) == 0 or abs(self.
                trajectory_x[-1] - x) > 0.01 or abs(self.
                trajectory_y[-1] - y) > 0.01:
                self.trajectory_x.append(x)
                self.trajectory_y.append(y)

                if len(self.trajectory_x) > 1000:
                    self.trajectory_x = self.trajectory_x
                        [-500:]
                    self.trajectory_y = self.trajectory_y
                        [-500:]

            # Actualizar gráfica de trayectoria
            self.traj_line.set_data(self.trajectory_x, self.
                trajectory_y)
            if len(self.trajectory_x) > 0:
                self.traj_start.set_data([self.trajectory_x
                    [0]], [self.trajectory_y[0]])
                self.traj_current.set_data([x], [y])

                margin = 1.0
                if len(self.trajectory_x) > 1:
                    x_min, x_max = min(self.trajectory_x),
                        max(self.trajectory_x)
                    y_min, y_max = min(self.trajectory_y),
                        max(self.trajectory_y)
                    self.traj_ax.set_xlim(x_min - margin,
                        x_max + margin)
                    self.traj_ax.set_ylim(y_min - margin,
                        y_max + margin)

            self.trajectory_canvas.draw_idle()

        except Exception as e:
            print(f"Error actualizando trayectoria: {e}")

    def update_lidar(self, lidar_data):
        """Función eliminada - No se necesita procesamiento
            LIDAR para visualización"""
        # COMPLETAMENTE VACÍA - Solo se mantiene por
            compatibilidad con señales existentes
        # No se realiza ningún procesamiento para visualizaci
            ón LIDAR
        pass

    def update_camera_frame(self, frame):
        """Actualizar frame de cámara en la interfaz"""
        try:
            # Convertir BGR a RGB
            frame_rgb = cv2.cvtColor(frame, cv2.
                COLOR_BGR2RGB)

            # Redimensionar si es necesario
            h, w, ch = frame_rgb.shape
            bytes_per_line = ch * w

            # Convertir a QImage
            qt_image = QImage(frame_rgb.data, w, h,
                bytes_per_line, QImage.Format_RGB888)
            pixmap = QPixmap.fromImage(qt_image)

            # Escalar para ajustar al label
            self.lbl_camera.setPixmap(pixmap.scaled(
                self.lbl_camera.width(),
                self.lbl_camera.height(),
                Qt.KeepAspectRatio,
                Qt.SmoothTransformation
            ))

            self.lbl_camera_status.setText("Cámara:
                Transmitiendo")

        except Exception as e:
            self.log_message(f"Error procesando frame de cá
                mara: {str(e)}")

    def handle_camera_error(self, error_message):
        """Manejar errores de cámara"""
        self.log_message(f"{error_message}")
        self.lbl_camera_status.setText("Cámara: Error")
        self.lbl_camera.setText("CÁMARA NO DISPONIBLE")

    def log_message(self, message):
        """Agregar mensaje al log"""
        timestamp = time.strftime("%H:%M:%S")
        log_entry = f"[{timestamp}] {message}"
        self.txt_log.append(log_entry)
        # Auto-scroll
        scrollbar = self.txt_log.verticalScrollBar()
        scrollbar.setValue(scrollbar.maximum())
```

manual_mode_processor.py

```
#!/usr/bin/env python3
# manual_mode_processor.py
# Procesamiento y lógica de negocio para modo manual - CON
    MAPEO REAL SIN VISUALIZACIÓN LIDAR

import sys
import time
import threading
import math
import cv2
from PyQt5.QtCore import QObject, pyqtSignal, QTimer,
    QThread
import numpy as np

from rplidar_express import RPLidarExpress
```

```
from navigation_controller import NavigationController
from refined_mapper import RefinedMapper

class CameraThread(QThread):
    """Hilo para captura de cámara en tiempo real"""
    frame_signal = pyqtSignal(np.ndarray)
    error_signal = pyqtSignal(str)

    def __init__(self, camera_index=0):
        super().__init__()
        self.camera_index = camera_index
        self.running = False
        self.cap = None

    def run(self):
        """Ejecutar captura de cámara"""
        try:
            print(f"Intentando abrir cámara con índice: {self.camera_index}")
            self.cap = cv2.VideoCapture(self.camera_index)

            if not self.cap.isOpened():
                # Intentar con diferentes índices
                for i in range(5):
                    self.cap = cv2.VideoCapture(i)
                    if self.cap.isOpened():
                        print(f"Cámara encontrada en índice: {i}")
                        break
                    self.cap.release()

            if not self.cap or not self.cap.isOpened():
                self.error_signal.emit("No se pudo abrir ninguna cámara")
                return

            # Configurar cámara
            self.cap.set(cv2.CAP_PROP_FRAME_WIDTH, 640)
            self.cap.set(cv2.CAP_PROP_FRAME_HEIGHT, 480)
            self.cap.set(cv2.CAP_PROP_FPS, 30)

            self.running = True
            print("Cámara inicializada correctamente")

            while self.running:
                ret, frame = self.cap.read()
                if ret:
                    self.frame_signal.emit(frame)
                else:
                    print("Error leyendo frame de cámara")
                    break
                time.sleep(0.033) # ~30 FPS

        except Exception as e:
            error_msg = f"Error en cámara: {str(e)}"
            print(error_msg)
            self.error_signal.emit(error_msg)
        finally:
            if self.cap:
                self.cap.release()
                print("Cámara liberada")

    def stop(self):
        """Detener captura de cámara"""
        self.running = False
        if self.isRunning():
            self.wait(2000) # Esperar máximo 2 segundos

class ManualModeProcessor(QObject):
    # Señales para comunicación con la UI - SIN SEÑAL LIDAR PARA VISUALIZACIÓN
    status_signal = pyqtSignal(dict)
    log_signal = pyqtSignal(str)
    trajectory_signal = pyqtSignal(dict)
    camera_frame_signal = pyqtSignal(np.ndarray)
    camera_error_signal = pyqtSignal(str)

    def __init__(self):
        super().__init__()
        self.nav_controller = None
        self.lidar = None
        self.mapper = None
        self.is_mapping = False
        self.scan_gen = None
        self.lidar_thread = None
        self.lidar_running = False
        self.camera_thread = None
        self.camera_angle = 90 # Ángulo inicial de la cámara (0-180)

        self.lidar_points = [] # Solo para mapeo, NO para visualización
        self.update_timer = QTimer()
        self.update_timer.timeout.connect(self.update_systems)

    def initialize_systems(self):
        """Inicializar sistemas reales del robot"""
        try:
            self.log_signal.emit("Inicializando controlador de navegación...")
            self.nav_controller = NavigationController()
            self.log_signal.emit("Controlador de navegación inicializado")

            self.log_signal.emit("Inicializando sistema de mapeo MEJORADO...")
            self.mapper = RefinedMapper(map_size=10.0, resolution=0.025)
            # Habilitar corrección de deriva por defecto
            self.mapper.enable_correction(True)
            self.log_signal.emit("Sistema de mapeo mejorado inicializado")
            self.log_signal.emit("Corrección de deriva: ACTIVADA")

            self.log_signal.emit("Inicializando sensor LIDAR...")
            self.init_lidar()

            self.log_signal.emit("Inicializando cámara...")
            self.init_camera()

            # Iniciar timer de actualizaciones
            self.update_timer.start(100) # 10 Hz

        except Exception as e:
            self.log_signal.emit(f"Error inicializando sistemas: {str(e)}")
            raise

    def init_lidar(self):
        """Inicializar LIDAR en un hilo separado - VERSIÓN OPTIMIZADA SIN VISUALIZACIÓN"""
        def lidar_thread_func():
            try:
                # ELIMINADO os.nice(5) - No se necesita ajuste de prioridad
                self.lidar = RPLidarExpress(port='/dev/ttyUSB0', baudrate=460800)
                if self.lidar.start_express_scan():
                    scan_gen = self.lidar.iter_packets()
                    self.lidar_running = True
                    self.log_signal.emit("LIDAR inicializado correctamente")

                    # Procesar datos LIDAR continuamente - SOLO PARA MAPEO
                    while self.lidar_running and self.lidar.scanning:
                        try:
                            all_points = []
                            start_time = time.time()
                            packets_collected = 0
                            max_packets = 128 # Reducido para mejor responsividad
                            timeout_seconds = 0.03 # Timeout más corto

                            while (packets_collected < max_packets and
                                   (time.time() - start_time) < timeout_seconds and
                                   self.lidar_running):
                                try:
                                    points = next(scan_gen,
```

```
                                None)
                            if points:
                                all_points.extend(points)
                                packets_collected += 1
                        except StopIteration:
                            scan_gen = self.lidar.iter_packets()
                            break
                        except Exception:
                            break

                    if all_points:
                        # ACTUALIZACIÓN SOLO PARA MAPEO, NO VISUALIZACIÓN
                        self.lidar_points = all_points

                        # MAPEO en hilo separado si está activo
                        if self.is_mapping and self.nav_controller and self.mapper:
                            mapping_thread = threading.Thread(
                                target=self.process_mapping_data,
                                args=(all_points,),
                                daemon=True
                            )
                            mapping_thread.start()

                    time.sleep(0.01) # Pausa mínima para mejor respuesta

                except Exception as e:
                    if self.lidar_running:
                        self.log_signal.emit(f"Error en hilo LIDAR: {str(e)}")
                    time.sleep(0.05)

        except Exception as e:
            self.log_signal.emit(f"Error inicializando LIDAR: {str(e)}")

    self.lidar_thread = threading.Thread(target=lidar_thread_func, daemon=True)
    self.lidar_thread.start()

def process_mapping_data(self, lidar_points):
    """Procesar datos de mapeo en hilo separado"""
    try:
        if self.is_mapping and self.nav_controller and self.mapper:
            robot_x, robot_y, robot_yaw = self.nav_controller.get_current_pose()
            points_processed = self.mapper.update_map(robot_x, robot_y, robot_yaw, lidar_points)
            # Log periódico para no saturar
            if points_processed > 0 and self.mapper.total_points_processed % 100 == 0:
                self.log_signal.emit(f"Puntos mapeados: {self.mapper.total_points_processed}")
    except Exception as e:
        print(f"Error en procesamiento de mapeo: {e}")

def init_camera(self):
    """Inicializar cámara en hilo separado"""
    try:
        self.camera_thread = CameraThread(camera_index=0)
        self.camera_thread.frame_signal.connect(self.camera_frame_signal)
        self.camera_thread.error_signal.connect(self.camera_error_signal)
        self.camera_thread.start()
    except Exception as e:
        self.log_signal.emit(f"Error iniciando cámara: {str(e)}")

def move_robot(self, direction):
    """Mover el robot real"""
    if not self.nav_controller or not self.nav_controller.bot:
        self.log_signal.emit("Robot no disponible")
        return

    try:
        speed = 0.15
        angular_speed = 0.3

        if direction == 'forward':
            self.nav_controller.bot.set_car_motion(speed, 0)
            self.log_signal.emit("Avanzando...")
        elif direction == 'backward':
            self.nav_controller.bot.set_car_motion(-speed, 0)
            self.log_signal.emit("Retrocediendo...")
        elif direction == 'left':
            self.nav_controller.bot.set_car_motion(0, angular_speed)
            self.log_signal.emit("Girando izquierda...")
        elif direction == 'right':
            self.nav_controller.bot.set_car_motion(0, -angular_speed)
            self.log_signal.emit("Girando derecha...")

    except Exception as e:
        self.log_signal.emit(f"Error moviendo robot: {str(e)}")

def stop_robot(self):
    """Detener el robot real"""
    try:
        if self.nav_controller and self.nav_controller.bot:
            self.nav_controller.bot.set_car_motion(0, 0)
            self.log_signal.emit("Robot detenido")
    except Exception as e:
        self.log_signal.emit(f"Error deteniendo robot: {str(e)}")

def move_camera(self, direction):
    """Mover la cámara (servomotor 2)"""
    if not self.nav_controller or not self.nav_controller.bot:
        self.log_signal.emit("Robot no disponible para mover cámara")
        return

    try:
        angle_step = 10 # Paso de ángulo en grados

        if direction == 'up':
            self.camera_angle = min(180, self.camera_angle + angle_step)
        elif direction == 'down':
            self.camera_angle = max(0, self.camera_angle - angle_step)
        elif direction == 'center':
            self.camera_angle = 90

        # Mover servomotor 2 (cámara)
        self.nav_controller.bot.set_pwm_servo(2, self.camera_angle)

        self.log_signal.emit(f"Cámara movida a {self.camera_angle}°")

    except Exception as e:
        self.log_signal.emit(f"Error moviendo cámara: {str(e)}")

def toggle_mapping(self):
    """Activar/desactivar guardado de mapa (el mapeo es siempre activo)"""
    self.is_mapping = not self.is_mapping

    if self.is_mapping:
        # INICIAR guardado de mapa - INICIAR SESIÓN DE MAPEO REAL
        if self.nav_controller and self.mapper:
```

```
            initial_pose = self.nav_controller.get_current_pose()
            self.mapper.start_mapping_session(initial_pose)
            self.log_signal.emit("␣Mapeo␣ACTIVADO␣-␣Guardando␣datos...")
            self.log_signal.emit(f"␣␣␣Sesión␣iniciada␣en␣({initial_pose[0]:.2f},{initial_pose[1]:.2f})")
            self.log_signal.emit(f"␣␣␣Corrección␣de␣deriva:␣ACTIVADA")
        else:
            # DETENER guardado y GUARDAR mapa usando el sistema real
            if self.mapper and self.mapper.total_points_processed > 0:
                try:
                    timestamp = time.strftime("%Y%m%d_%H%M%S")
                    success = self.mapper.save_data(f"manual_map_{timestamp}")
                    if success:
                        self.log_signal.emit(f"␣Mapa␣guardado␣manual_map_{timestamp}")
                        self.log_signal.emit(f"␣␣␣Puntos␣procesados:␣{self.mapper.total_points_processed}")
                        self.log_signal.emit(f"␣␣␣Trayectoria␣guardada:␣{len(self.mapper.trajectory)}␣puntos")
                        self.log_signal.emit("␣Use␣'VER␣MAPAS␣GUARDADOS'␣para␣ver␣el␣mapa")
                    else:
                        self.log_signal.emit("␣Error␣guardando␣mapa")
                except Exception as e:
                    self.log_signal.emit(f"␣Error␣guardando␣mapa:␣{str(e)}")

            self.log_signal.emit("␣Mapeo␣DETENIDO␣-␣Mapa␣guardado")

    def update_systems(self):
        """Actualizar␣datos␣del␣sistema␣para␣la␣UI␣-␣SIN␣DATOS␣LIDAR␣PARA␣VISUALIZACIÓN"""
        try:
            status_data = {}
            trajectory_data = {}

            if self.nav_controller:
                # Obtener datos del robot
                x, y, yaw = self.nav_controller.get_current_pose()
                linear_speed, angular_velocity = self.nav_controller.get_motion_data()
                battery = self.nav_controller.get_battery_voltage()

                # Preparar datos de estado
                status_data.update({
                    'robot_status': 'Conectado',
                    'x': x,
                    'y': y,
                    'yaw_deg': math.degrees(yaw),
                    'battery': battery,
                    'camera_angle': self.camera_angle,
                    'points_mapped': self.mapper.total_points_processed if self.mapper else 0,
                    'mapping_status': 'ACTIVO␣(Guardando)' if self.is_mapping else 'INACTIVO'
                })

                # Preparar datos de trayectoria
                trajectory_data.update({
                    'x': x,
                    'y': y
                })

            # Datos LIDAR - SOLO ESTADO, NO DATOS PARA VISUALIZACIÓN
            if self.lidar and self.lidar.scanning:
                status_data['lidar_status'] = 'Escaneando'
            else:
                status_data['lidar_status'] = 'Desconectado'

            # Datos cámara
            status_data['camera_status'] = 'Transmitiendo' if self.camera_thread and self.camera_thread.running else 'Error'

            # ELIMINADA COMPLETAMENTE LA PREPARACIÓN DE DATOS LIDAR PARA VISUALIZACIÓN

            # Emitir señales - SIN SEÑAL LIDAR
            self.status_signal.emit(status_data)
            self.trajectory_signal.emit(trajectory_data)
            # NO SE EMITE SEÑAL LIDAR PARA VISUALIZACIÓN

        except Exception as e:
            print(f"Error␣actualizando␣sistemas:␣{e}")

    def cleanup(self):
        """Limpiar␣recursos"""
        self.log_signal.emit("␣Cerrando␣sistemas...")

        # Detener timer
        self.update_timer.stop()

        # Guardar mapa final si hay datos
        if self.mapper and self.mapper.total_points_processed > 0:
            try:
                timestamp = time.strftime("%Y%m%d_%H%M%S")
                self.mapper.save_data(f"final_manual_map_{timestamp}")
                self.log_signal.emit(f"␣Mapa␣final␣guardado:␣final_manual_map_{timestamp}")
            except Exception as e:
                self.log_signal.emit(f"␣Error␣guardando␣mapa␣final:␣{str(e)}")

        # Detener LIDAR
        self.lidar_running = False
        if self.lidar:
            try:
                self.lidar.stop_scan()
                self.lidar.close()
                print("LIDAR␣cerrado␣correctamente")
            except Exception as e:
                print(f"Error␣cerrando␣LIDAR:␣{e}")

        # Detener cámara
        if self.camera_thread and self.camera_thread.isRunning():
            self.camera_thread.stop()

        # Detener robot
        self.stop_robot()

        self.log_signal.emit("Sistemas␣cerrados␣correctamente")
```

navigation_controller.py

```python
#!/usr/bin/env python3
# navigation_controller.py
# Control de navegación real para el robot Transbot + DEBUG

import math
import time
import numpy as np
import logging

class NavigationController:
    def __init__(self):
        self.x = 0.0
        self.y = 0.0
        self.yaw = 0.0
        self.trajectory = []
        self.last_update_time = time.time()

        try:
            from Transbot_Lib import Transbot
            self.bot = Transbot(debug=False)
            self.bot.create_receive_threading()
            self.bot.set_auto_report_state(True, forever=
                  False)
            self.bot.set_imu_adjust(True, forever=False)
            time.sleep(2)
            logging.debug("NavigationController: Robot
                  Transbot inicializado")
        except Exception as e:
            logging.error(f"NavigationController: Error - {e
                  }")
            self.bot = None

    def update_pose(self, dt=None):
        """Actualizar pose del robot usando datos reales del
              IMU"""
        if not self.bot:
            return 0, 0

        try:
            current_time = time.time()
            if dt is None:
                dt = current_time - self.last_update_time
            self.last_update_time = current_time

            # Obtener datos del giroscopio y movimiento
            gx, gy, gz = self.bot.get_gyroscope_data()
            linear_speed, angular_velocity = self.bot.
                  get_motion_data()

            # Integrar para obtener pose (usando el
                  giroscopio para mejor precisión)
            self.yaw += gz * dt
            self.x += linear_speed * math.cos(self.yaw) * dt
            self.y += linear_speed * math.sin(self.yaw) * dt

            self.trajectory.append((self.x, self.y, self.yaw
                  ))

            logging.debug(f"Pose actualizada: ({self.x:.2f},
                  {self.y:.2f}, {math.degrees(self.yaw):.1f
                  }°)")

            return linear_speed, angular_velocity
        except Exception as e:
            logging.error(f"Error en update_pose: {e}")
            return 0, 0

    def get_current_pose(self):
        """Obtener pose actual del robot"""
        # Actualizar pose antes de devolver
        self.update_pose()
        return (self.x, self.y, self.yaw)

    def get_motion_data(self):
        """Obtener datos de movimiento reales"""
        if not self.bot:
            return 0, 0
        try:
            return self.bot.get_motion_data()
        except Exception as e:
            logging.error(f"Error obteniendo datos de
                  movimiento: {e}")
            return 0, 0

    def get_battery_voltage(self):
        """Obtener voltaje de la batería"""
        if not self.bot:
            return 0.0
        try:
            voltage = self.bot.get_battery_voltage()
            logging.debug(f"Batería: {voltage:.1f}V")
            return voltage
        except Exception as e:
            logging.error(f"Error obteniendo voltaje: {e}")
            return 0.0

    def get_gyroscope_data(self):
        """Obtener datos del giroscopio"""
        if not self.bot:
            return 0, 0, 0
        try:
            return self.bot.get_gyroscope_data()
        except Exception as e:
            logging.error(f"Error obteniendo datos
                  giroscopio: {e}")
            return 0, 0, 0

    def get_trajectory(self):
        """Obtener trayectoria completa"""
        return self.trajectory

    def reset_pose(self):
        """Reiniciar pose a cero"""
        self.x = 0.0
        self.y = 0.0
        self.yaw = 0.0
        self.trajectory = []
        self.last_update_time = time.time()
        logging.debug("Pose reiniciado a cero")
        return True
```

auto_mode_gui.py

```python
#!/usr/bin/env python3
# auto_mode_gui.py
# Diseño de la interfaz gráfica para el modo automático -
      SIN VISUALIZACIÓN LIDAR

from PyQt5.QtWidgets import (QVBoxLayout, QHBoxLayout,
      QPushButton, QWidget,
                             QLabel, QGroupBox, QTextEdit,
                                   QCheckBox, QFrame)
from PyQt5.QtCore import Qt, pyqtSlot
from PyQt5.QtGui import QFont, QPixmap, QImage
import matplotlib.pyplot as plt
from matplotlib.backends.backend_qt5agg import
      FigureCanvasQTAgg as FigureCanvas
from matplotlib.figure import Figure
import numpy as np
import cv2

class AutoModeGUI(QWidget):
    def __init__(self):
        super().__init__()
```

```
        self.init_ui()
        self.init_plots()

    def init_ui(self):
        """Inicializar la interfaz de usuario - SIN ELEMENTOS LIDAR"""
        main_layout = QHBoxLayout(self)
        main_layout.setSpacing(12)
        main_layout.setContentsMargins(12, 12, 12, 12)

        # ===== PANEL IZQUIERDO - CONTROL DE TRAYECTORIA =====
        left_panel = QVBoxLayout()
        left_panel.setSpacing(12)

        # GRUPO: DIBUJAR TRAYECTORIA
        draw_group = QGroupBox("DIBUJAR TRAYECTORIA DEL ROBOT")
        draw_group.setFont(QFont("Arial", 11, QFont.Bold))
        draw_group.setMinimumWidth(220)

        draw_layout = QVBoxLayout()

        self.btn_start_drawing = QPushButton("INICIAR DIBUJO")
        self.btn_start_drawing.setMinimumHeight(35)
        self.btn_start_drawing.setFont(QFont("Arial", 10, QFont.Bold))
        self.btn_start_drawing.setStyleSheet(self.get_button_style("#2196F3", "#1976D2"))

        self.btn_save_route = QPushButton("GUARDAR RUTA")
        self.btn_save_route.setMinimumHeight(30)
        self.btn_save_route.setFont(QFont("Arial", 10, QFont.Bold))
        self.btn_save_route.setStyleSheet(self.get_button_style("#4CAF50", "#45a049"))
        self.btn_save_route.setEnabled(False)

        self.btn_clear_trajectory = QPushButton("BORRAR TRAYECTORIA")
        self.btn_clear_trajectory.setMinimumHeight(30)
        self.btn_clear_trajectory.setFont(QFont("Arial", 10, QFont.Bold))
        self.btn_clear_trajectory.setStyleSheet(self.get_button_style("#FF9800", "#F57C00"))

        draw_layout.addWidget(self.btn_start_drawing)
        draw_layout.addWidget(self.btn_save_route)
        draw_layout.addWidget(self.btn_clear_trajectory)

        # Información de trayectoria dibujada
        self.lbl_trajectory_info = QLabel("Trayectoria: 0 puntos")
        self.lbl_trajectory_info.setFont(QFont("Arial", 9, QFont.Bold))
        self.lbl_trajectory_info.setStyleSheet("color: #2E7D32; padding: 4px;")
        draw_layout.addWidget(self.lbl_trajectory_info)

        # Información de estado de dibujo
        self.lbl_drawing_status = QLabel("Modo dibujo: INACTIVO")
        self.lbl_drawing_status.setFont(QFont("Arial", 9))
        self.lbl_drawing_status.setStyleSheet("color: #D32F2F; padding: 4px;")
        draw_layout.addWidget(self.lbl_drawing_status)

        draw_group.setLayout(draw_layout)
        left_panel.addWidget(draw_group)

        # GRUPO: EJECUTAR TRAYECTORIA
        execute_group = QGroupBox("EJECUTAR TRAYECTORIA")
        execute_group.setFont(QFont("Arial", 11, QFont.Bold))

        execute_layout = QVBoxLayout()

        self.btn_execute = QPushButton("EJECUTAR TRAYECTORIA")
        self.btn_execute.setMinimumHeight(40)
        self.btn_execute.setFont(QFont("Arial", 11, QFont.Bold))
        self.btn_execute.setStyleSheet(self.get_button_style("#4CAF50", "#45a049"))
        self.btn_execute.setEnabled(False)

        self.btn_stop_execution = QPushButton("DETENER EJECUCION")
        self.btn_stop_execution.setMinimumHeight(30)
        self.btn_stop_execution.setFont(QFont("Arial", 10, QFont.Bold))
        self.btn_stop_execution.setStyleSheet(self.get_button_style("#f44336", "#d32f2f"))
        self.btn_stop_execution.setEnabled(False)

        # BOTÓN DE PARO DE EMERGENCIA
        self.btn_emergency_stop = QPushButton("PARO DE EMERGENCIA")
        self.btn_emergency_stop.setMinimumHeight(45)
        self.btn_emergency_stop.setFont(QFont("Arial", 11, QFont.Bold))
        self.btn_emergency_stop.setStyleSheet("""
            QPushButton {
                background-color: #D32F2F;
                color: white;
                border: 2px solid #B71C1C;
                border-radius: 6px;
            }
            QPushButton:hover {
                background-color: #C62828;
                border: 2px solid #9C1C1C;
            }
            QPushButton:pressed {
                background-color: #B71C1C;
            }
        """)

        execute_layout.addWidget(self.btn_execute)
        execute_layout.addWidget(self.btn_stop_execution)
        execute_layout.addWidget(self.btn_emergency_stop)

        # Progreso de ejecución
        self.lbl_execution_info = QLabel("Listo para ejecutar")
        self.lbl_execution_info.setFont(QFont("Arial", 9, QFont.Bold))
        self.lbl_execution_info.setStyleSheet("color: #1565C0; background-color: #E3F2FD; padding: 6px; border-radius: 4px;")
        execute_layout.addWidget(self.lbl_execution_info)

        execute_group.setLayout(execute_layout)
        left_panel.addWidget(execute_group)

        # GRUPO: CONFIGURACIÓN
        config_group = QGroupBox("CONFIGURACION")
        config_group.setFont(QFont("Arial", 11, QFont.Bold))

        config_layout = QVBoxLayout()

        # Checkbox para evasión de obstáculos
        self.cb_obstacle_avoidance = QCheckBox("Activar Evasion de Obstaculos")
        self.cb_obstacle_avoidance.setChecked(True)
        self.cb_obstacle_avoidance.setFont(QFont("Arial", 10))

        # Checkbox para corrección de deriva en mapeo
        self.cb_correction_mapping = QCheckBox("Activar Corrección de Deriva en Mapeo")
        self.cb_correction_mapping.setChecked(True)
        self.cb_correction_mapping.setFont(QFont("Arial", 10))

        # Botón para ver mapas guardados
        self.btn_view_maps = QPushButton("VER MAPAS GUARDADOS")
        self.btn_view_maps.setMinimumHeight(30)
        self.btn_view_maps.setFont(QFont("Arial", 10, QFont.Bold))
        self.btn_view_maps.setStyleSheet(self.get_button_style("#FF9800", "#F57C00"))

        config_layout.addWidget(self.cb_obstacle_avoidance)
        config_layout.addWidget(self.cb_correction_mapping)
```

```
        config_layout.addWidget(self.btn_view_maps)

        config_group.setLayout(config_layout)
        left_panel.addWidget(config_group)

        # GRUPO: REGISTRO DE EVENTOS
        log_group = QGroupBox("REGISTRO DE EVENTOS")
        log_group.setFont(QFont("Arial", 11, QFont.Bold))

        log_layout = QVBoxLayout()
        self.txt_log = QTextEdit()
        self.txt_log.setMinimumHeight(250)
        self.txt_log.setReadOnly(True)
        self.txt_log.setFont(QFont("Arial", 8))
        self.txt_log.setStyleSheet("background-color: #
            F5F5F5; border: 1px solid #CCCCCC;")
        log_layout.addWidget(self.txt_log)

        log_group.setLayout(log_layout)
        left_panel.addWidget(log_group)

        # ===== PANEL DERECHO - VISUALIZACIONES =====
        right_panel = QVBoxLayout()
        right_panel.setSpacing(12)

        # TRAYECTORIA DEL ROBOT (OCUPANDO TODO EL ESPACIO
            SUPERIOR LIBERADO POR LIDAR)
        traj_group = QGroupBox("TRAYECTORIA DEL ROBOT - 
            PLANIFICACIÓN Y EJECUCIÓN")
        traj_group.setFont(QFont("Arial", 11, QFont.Bold))
        traj_group.setMinimumHeight(500) # Más alto para
            ocupar el espacio del LIDAR eliminado

        traj_layout = QVBoxLayout()
        self.trajectory_figure = Figure(figsize=(12, 6)) #
            Figura más ancha
        self.trajectory_canvas = FigureCanvas(self.
            trajectory_figure)
        traj_layout.addWidget(self.trajectory_canvas)
        traj_group.setLayout(traj_layout)

        right_panel.addWidget(traj_group)

        # Fila inferior: CÁMARA + INFORMACIÓN DEL SISTEMA Y
            EJECUCIÓN
        bottom_row = QHBoxLayout()

        # CÁMARA DEL ROBOT
        camera_group = QGroupBox("CAMARA DEL ROBOT")
        camera_group.setFont(QFont("Arial", 11, QFont.Bold))
        camera_group.setMinimumHeight(180)

        camera_layout = QVBoxLayout()
        self.lbl_camera = QLabel("Camara no disponible")
        self.lbl_camera.setAlignment(Qt.AlignCenter)
        self.lbl_camera.setMinimumSize(200, 100)
        self.lbl_camera.setStyleSheet("""
            QLabel {
                background-color: #2C2C2C;
                color: #CCCCCC;
                border: 2px solid #666;
                font-size: 12px;
                border-radius: 6px;
            }
        """)
        camera_layout.addWidget(self.lbl_camera)
        camera_group.setLayout(camera_layout)

        # INFORMACIÓN DEL SISTEMA Y EJECUCIÓN
        info_group = QGroupBox("INFORMACION DEL SISTEMA Y
            EJECUCION")
        info_group.setFont(QFont("Arial", 11, QFont.Bold))
        info_group.setMinimumHeight(180)

        info_layout = QVBoxLayout()

        # Información del sistema
        self.lbl_robot_status = QLabel("Robot: Conectando
            ")
        self.lbl_position = QLabel("Posicion: (0.00, 0.00)")
        self.lbl_orientation = QLabel("Orientacion: 0.00°")
        self.lbl_battery = QLabel("Bateria: --V")
        self.lbl_lidar = QLabel("LIDAR: Conectando...")
        self.lbl_obstacle_status = QLabel("Obstaculos: No
            detectados")
        self.lbl_points_mapped = QLabel("Puntos mapeados: 0"
            )
        self.lbl_mapping_status = QLabel("Mapeo: INACTIVO")

        for lbl in [self.lbl_robot_status, self.lbl_position
            , self.lbl_orientation,
                self.lbl_battery, self.lbl_lidar, self.
                    lbl_obstacle_status,
                self.lbl_points_mapped, self.
                    lbl_mapping_status]:
            lbl.setFont(QFont("Arial", 9))
            lbl.setStyleSheet("padding: 2px;")
            info_layout.addWidget(lbl)

        # Separador
        separator1 = QFrame()
        separator1.setFrameShape(QFrame.HLine)
        separator1.setFrameShadow(QFrame.Sunken)
        info_layout.addWidget(separator1)

        # Información de ejecución
        self.lbl_execution_state = QLabel("SISTEMA LISTO")
        self.lbl_execution_state.setFont(QFont("Arial", 10,
            QFont.Bold))
        self.lbl_execution_state.setStyleSheet("color: #2
            E7D32; background-color: #E8F5E8; padding: 8px
            ; border-radius: 4px;")
        self.lbl_execution_state.setAlignment(Qt.AlignCenter
            )

        self.lbl_current_waypoint = QLabel("Waypoint actual:
             --/--")
        self.lbl_current_waypoint.setFont(QFont("Arial", 9))

        self.lbl_progress = QLabel("Progreso: 0%")
        self.lbl_progress.setFont(QFont("Arial", 9))

        self.lbl_distance_to_target = QLabel("Distancia al
            objetivo: --m")
        self.lbl_distance_to_target.setFont(QFont("Arial",
            9))

        self.lbl_execution_time = QLabel("Tiempo de
            ejecucion: 00:00")
        self.lbl_execution_time.setFont(QFont("Arial", 9))

        # Separador
        separator2 = QFrame()
        separator2.setFrameShape(QFrame.HLine)
        separator2.setFrameShadow(QFrame.Sunken)

        # Estadísticas de obstáculos
        self.lbl_obstacles_detected = QLabel("Obstaculos
            detectados: 0")
        self.lbl_obstacles_detected.setFont(QFont("Arial",
            9))

        self.lbl_avoidance_maneuvers = QLabel("Maniobras de
            evasion: 0")
        self.lbl_avoidance_maneuvers.setFont(QFont("Arial",
            9))

        info_layout.addWidget(self.lbl_execution_state)
        info_layout.addWidget(self.lbl_current_waypoint)
        info_layout.addWidget(self.lbl_progress)
        info_layout.addWidget(self.lbl_distance_to_target)
        info_layout.addWidget(self.lbl_execution_time)
        info_layout.addWidget(separator2)
        info_layout.addWidget(self.lbl_obstacles_detected)
        info_layout.addWidget(self.lbl_avoidance_maneuvers)

        info_group.setLayout(info_layout)

        bottom_row.addWidget(camera_group)
        bottom_row.addWidget(info_group)

        right_panel.addLayout(bottom_row)

        # ENSAMBLAR INTERFAZ
        main_layout.addLayout(left_panel, 1)
        main_layout.addLayout(right_panel, 2)
```

```
    def get_button_style(self, normal_color, hover_color,
        disabled=False):
        """Generar estilo CSS para botones"""
        if disabled:
            return f"""
                QPushButton {{
                    background-color: #cccccc;
                    color: #666666;
                    border: none;
                    border-radius: 6px;
                }}
            """
        else:
            return f"""
                QPushButton {{
                    background-color: {normal_color};
                    color: white;
                    border: none;
                    border-radius: 6px;
                }}
                QPushButton:hover {{
                    background-color: {hover_color};
                }}
            """

    def init_plots(self):
        """Inicializar gráficas - SIN GRÁFICA LIDAR"""
        # Gráfica de trayectoria - más grande y detallada
        self.traj_ax = self.trajectory_figure.add_subplot
            (111)
        self.traj_ax.set_xlabel('X (metros)', fontsize=12)
        self.traj_ax.set_ylabel('Y (metros)', fontsize=12)
        self.traj_ax.grid(True, alpha=0.3)
        self.traj_ax.axis('equal')
        self.traj_ax.set_xlim(-3, 3)
        self.traj_ax.set_ylim(-3, 3)
        self.traj_ax.set_title('Planificación y Ejecución de
            Trayectoria', fontsize=14, fontweight='bold')

        # Elementos de la gráfica
        self.planned_line, = self.traj_ax.plot([], [], 'r--
            , linewidth=2, label='Trayectoria Planeada')
        self.actual_line, = self.traj_ax.plot([], [], 'b-',
            linewidth=2, label='Trayectoria Real')
        self.start_point = self.traj_ax.plot([], [], 'go',
            markersize=12, label='Inicio')[0]
        self.waypoint_markers = self.traj_ax.plot([], [], '
            rx', markersize=10, label='Waypoints')[0]
        self.end_point = self.traj_ax.plot([], [], 'mo',
            markersize=12, label='Final')[0]
        self.current_point = self.traj_ax.plot([], [], 'ro',
             markersize=8, label='Actual')[0]
        self.traj_ax.legend(fontsize=10, loc='upper right')

        self.trajectory_canvas.draw()

    def update_trajectory_display(self, planned_trajectory,
        actual_trajectory, current_pose=None):
        """Actualizar visualización de la trayectoria"""
        try:
            # Limpiar la gráfica
            self.planned_line.set_data([], [])
            self.waypoint_markers.set_data([], [])
            self.start_point.set_data([], [])
            self.end_point.set_data([], [])
            self.current_point.set_data([], [])

            if planned_trajectory:
                x_planned = [p[0] for p in planned_trajectory
                    ]
                y_planned = [p[1] for p in planned_trajectory
                    ]
                self.planned_line.set_data(x_planned,
                    y_planned)

                # Mostrar waypoints (todos los puntos
                    intermedios)
                if len(planned_trajectory) > 2:
                    x_waypoints = x_planned[1:-1] # Excluir
                        inicio y final
                    y_waypoints = y_planned[1:-1]
                    self.waypoint_markers.set_data(
                        x_waypoints, y_waypoints)

                # Mostrar punto inicial (verde)
                if len(planned_trajectory) >= 1:
                    start_x, start_y = planned_trajectory[0]
                    self.start_point.set_data([start_x], [
                        start_y])

                # Mostrar punto final (magenta)
                if len(planned_trajectory) >= 2:
                    end_x, end_y = planned_trajectory[-1]
                    self.end_point.set_data([end_x], [end_y])

                # Actualizar información
                total_points = len(planned_trajectory)
                waypoints_count = max(0, total_points - 2) #
                    Excluyendo inicio y final
                self.lbl_trajectory_info.setText(f"
                    Trayectoria: {total_points} puntos ({
                    waypoints_count} waypoints)")

            if actual_trajectory:
                x_actual = [p[0] for p in actual_trajectory]
                y_actual = [p[1] for p in actual_trajectory]
                self.actual_line.set_data(x_actual, y_actual)

                # Punto actual
                if actual_trajectory:
                    current_x, current_y = actual_trajectory
                        [-1][:2]
                    self.current_point.set_data([current_x],
                        [current_y])

            # Ajustar límites automáticamente si hay datos
            if planned_trajectory or actual_trajectory:
                all_x = []
                all_y = []
                if planned_trajectory:
                    all_x.extend(x_planned)
                    all_y.extend(y_planned)
                if actual_trajectory:
                    all_x.extend(x_actual)
                    all_y.extend(y_actual)

                if all_x and all_y:
                    margin = 0.5
                    x_min, x_max = min(all_x) - margin, max(
                        all_x) + margin
                    y_min, y_max = min(all_y) - margin, max(
                        all_y) + margin
                    self.traj_ax.set_xlim(x_min, x_max)
                    self.traj_ax.set_ylim(y_min, y_max)

            self.trajectory_canvas.draw_idle()

        except Exception as e:
            print(f"Error actualizando trayectoria: {e}")

    def update_drawing_status(self, is_drawing, point_count
        =0):
        """Actualizar estado del modo dibujo"""
        if is_drawing:
            self.lbl_drawing_status.setText(f"Modo dibujo:
                ACTIVO - Puntos: {point_count}")
            self.lbl_drawing_status.setStyleSheet("color: #2
                E7D32; padding: 4px;")
            self.btn_start_drawing.setText("DETENER DIBUJO")
            self.btn_start_drawing.setStyleSheet(self.
                get_button_style("#f44336", "#d32f2f"))
            self.btn_save_route.setEnabled(point_count >= 2)
                 # Mínimo inicio y final
        else:
            self.lbl_drawing_status.setText("Modo dibujo:
                INACTIVO")
            self.lbl_drawing_status.setStyleSheet("color: #
                D32F2F; padding: 4px;")
            self.btn_start_drawing.setText("INICIAR DIBUJO")
            self.btn_start_drawing.setStyleSheet(self.
                get_button_style("#2196F3", "#1976D2"))
            self.btn_save_route.setEnabled(point_count >= 2)

    def update_execution_buttons(self, has_saved_route,
        is_executing):
```

```
        """Actualizar estado de los botones de ejecución"""
        self.btn_execute.setEnabled(has_saved_route and not is_executing)
        self.btn_stop_execution.setEnabled(is_executing)
        self.btn_save_route.setEnabled(not has_saved_route and not is_executing)

        # Actualizar texto informativo
        if is_executing:
            self.lbl_execution_info.setText("EJECUTANDO TRAYECTORIA")
            self.lbl_execution_info.setStyleSheet("color: #F57C00; background-color: #FFF3E0; padding: 6px; border-radius: 4px;")
        elif has_saved_route:
            self.lbl_execution_info.setText("LISTO PARA EJECUTAR")
            self.lbl_execution_info.setStyleSheet("color: #2E7D32; background-color: #E8F5E8; padding: 6px; border-radius: 4px;")
        else:
            self.lbl_execution_info.setText("DIBUJE Y GUARDE UNA TRAYECTORIA")
            self.lbl_execution_info.setStyleSheet("color: #1565C0; background-color: #E3F2FD; padding: 6px; border-radius: 4px;")

    @pyqtSlot(np.ndarray)
    def update_camera_frame(self, frame):
        """Actualizar frame de cámara en la interfaz"""
        try:
            # Convertir BGR a RGB
            frame_rgb = cv2.cvtColor(frame, cv2.COLOR_BGR2RGB)

            # Redimensionar si es necesario
            h, w, ch = frame_rgb.shape
            bytes_per_line = ch * w

            # Convertir a QImage
            qt_image = QImage(frame_rgb.data, w, h, bytes_per_line, QImage.Format_RGB888)
            pixmap = QPixmap.fromImage(qt_image)

            # Escalar para ajustar al label
            self.lbl_camera.setPixmap(pixmap.scaled(
                self.lbl_camera.width(),
                self.lbl_camera.height(),
                Qt.KeepAspectRatio,
                Qt.SmoothTransformation
            ))

        except Exception as e:
            self.log_message(f"Error procesando frame de cámara: {str(e)}")

    @pyqtSlot(str)
    def handle_camera_error(self, error_message):
        """Manejar errores de cámara"""
        self.log_message(f"{error_message}")
        self.lbl_camera.setText("CÁMARA NO DISPONIBLE")

    def log_message(self, message):
        """Agregar mensaje al log"""
        import time
        timestamp = time.strftime("%H:%M:%S")
        log_entry = f"[{timestamp}] {message}"
        self.txt_log.append(log_entry)

        # Auto-scroll al final
        scrollbar = self.txt_log.verticalScrollBar()
        scrollbar.setValue(scrollbar.maximum())
```

navigation_controller.py

```
#!/usr/bin/env python3
# navigation_controller.py
# Control de navegación real para el robot Transbot + DEBUG

import math
import time
import numpy as np
import logging

class NavigationController:
    def __init__(self):
        self.x = 0.0
        self.y = 0.0
        self.yaw = 0.0
        self.trajectory = []
        self.last_update_time = time.time()

        try:
            from Transbot_Lib import Transbot
            self.bot = Transbot(debug=False)
            self.bot.create_receive_threading()
            self.bot.set_auto_report_state(True, forever=False)
            self.bot.set_imu_adjust(True, forever=False)
            time.sleep(2)
            logging.debug("NavigationController: Robot Transbot inicializado")
        except Exception as e:
            logging.error(f"NavigationController: Error - {e}")
            self.bot = None

    def update_pose(self, dt=None):
        """Actualizar pose del robot usando datos reales del IMU"""
        if not self.bot:
            return 0, 0

        try:
            current_time = time.time()
            if dt is None:
                dt = current_time - self.last_update_time
            self.last_update_time = current_time

            # Obtener datos del giroscopio y movimiento
            gx, gy, gz = self.bot.get_gyroscope_data()
            linear_speed, angular_velocity = self.bot.get_motion_data()

            # Integrar para obtener pose (usando el giroscopio para mejor precisión)
            self.yaw += gz * dt
            self.x += linear_speed * math.cos(self.yaw) * dt
            self.y += linear_speed * math.sin(self.yaw) * dt

            self.trajectory.append((self.x, self.y, self.yaw))

            logging.debug(f"Pose actualizada: ({self.x:.2f}, {self.y:.2f}, {math.degrees(self.yaw):.1f}°)")

            return linear_speed, angular_velocity
        except Exception as e:
            logging.error(f"Error en update_pose: {e}")
            return 0, 0

    def get_current_pose(self):
        """Obtener pose actual del robot"""
        # Actualizar pose antes de devolver
```

```
        self.update_pose()
        return (self.x, self.y, self.yaw)

    def get_motion_data(self):
        """Obtener datos de movimiento reales"""
        if not self.bot:
            return 0, 0
        try:
            return self.bot.get_motion_data()
        except Exception as e:
            logging.error(f"Error obteniendo datos de
                movimiento: {e}")
            return 0, 0

    def get_battery_voltage(self):
        """Obtener voltaje de la batería"""
        if not self.bot:
            return 0.0
        try:
            voltage = self.bot.get_battery_voltage()
            logging.debug(f"Batería: {voltage:.1f}V")
            return voltage
        except Exception as e:
            logging.error(f"Error obteniendo voltaje: {e}")
            return 0.0

    def get_gyroscope_data(self):
        """Obtener datos del giroscopio"""
        if not self.bot:
            return 0, 0, 0
        try:
            return self.bot.get_gyroscope_data()
        except Exception as e:
            logging.error(f"Error obteniendo datos
                giroscopio: {e}")
            return 0, 0, 0

    def get_trajectory(self):
        """Obtener trayectoria completa"""
        return self.trajectory

    def reset_pose(self):
        """Reiniciar pose a cero"""
        self.x = 0.0
        self.y = 0.0
        self.yaw = 0.0
        self.trajectory = []
        self.last_update_time = time.time()
        logging.debug("Pose reiniciado a cero")
        return True
```

refined_mapper.py

```
#!/usr/bin/env python3
# refined_mapper.py
# Sistema de mapeo SLAM avanzado con Loop Closure y Pose
    Graph Optimization

import math
import time
import numpy as np
import matplotlib.pyplot as plt
from collections import deque
import logging

# Configurar logging
logging.basicConfig(level=logging.INFO, format='%(asctime)s
    - %(name)s - %(levelname)s - %(message)s')

class PoseGraph:
    def __init__(self):
        self.poses = [] # Lista de poses [x, y, theta]
        self.edges = [] # Lista de constraints [from_idx,
            to_idx, dx, dy, dtheta]
        self.loop_closure_edges = [] # Constraints de loop
            closure

    def add_pose(self, pose):
        """Agregar una nueva pose al grafo"""
        self.poses.append(pose)

        # Agregar edge odométrico si hay más de una pose
        if len(self.poses) > 1:
            prev_pose = self.poses[-2]
            dx = pose[0] - prev_pose[0]
            dy = pose[1] - prev_pose[1]
            dtheta = pose[2] - prev_pose[2]

            # Normalizar ángulo
            while dtheta > math.pi:
                dtheta -= 2 * math.pi
            while dtheta < -math.pi:
                dtheta += 2 * math.pi

            self.edges.append((len(self.poses)-2, len(self.
                poses)-1, dx, dy, dtheta))

    def add_loop_closure(self, from_idx, to_idx,
        relative_pose):
        """Agregar un constraint de loop closure"""
        self.loop_closure_edges.append((from_idx, to_idx,
            relative_pose[0],
                                        relative_pose[1],
                                            relative_pose[2]))
        logging.info(f"Loop closure agregado: {from_idx} -> {
            to_idx}")

    def optimize(self):
        """Optimización simplificada del grafo de poses"""
        if len(self.loop_closure_edges) == 0:
            return self.poses

        logging.info(f"Optimizando grafo con {len(self.
            loop_closure_edges)} loop closures")

        # Aplicar correcciones basadas en loop closures
        optimized_poses = self.poses.copy()

        for lc in self.loop_closure_edges:
            from_idx, to_idx, dx, dy, dtheta = lc

            if from_idx >= len(optimized_poses) or to_idx >=
                len(optimized_poses):
                continue

            # Calcular corrección
            current_from = optimized_poses[from_idx]
            current_to = optimized_poses[to_idx]

            # Error actual
            error_x = current_to[0] - current_from[0] - dx
            error_y = current_to[1] - current_from[1] - dy
            error_theta = current_to[2] - current_from[2] -
                dtheta

            # Normalizar error angular
            while error_theta > math.pi:
                error_theta -= 2 * math.pi
            while error_theta < -math.pi:
                error_theta += 2 * math.pi

            # Aplicar corrección distribuida
            correction_strength = 0.3 # Factor de suavizado

            # Corregir poses entre from_idx y to_idx
```

```
            for i in range(from_idx + 1, to_idx):
                if i < len(optimized_poses):
                    factor = (i - from_idx) / (to_idx - from_idx)
                    optimized_poses[i] = (
                        optimized_poses[i][0] - error_x * factor * correction_strength,
                        optimized_poses[i][1] - error_y * factor * correction_strength,
                        optimized_poses[i][2] - error_theta * factor * correction_strength
                    )

        self.poses = optimized_poses
        return optimized_poses

class RefinedMapper:
    def __init__(self, map_size=12.0, resolution=0.02):
        self.map_size = map_size
        self.resolution = resolution
        self.grid_size = int(map_size / resolution)
        self.center = self.grid_size // 2

        # Grid de probabilidades
        self.logodds = np.zeros((self.grid_size, self.grid_size), dtype=np.float32)

        # Parámetros de ocupación (agresivos para mapas definidos)
        self.OCCUPIED_INC = 12
        self.FREE_INC = -4
        self.LOGODDS_MIN = -200
        self.LOGODDS_MAX = 200

        self.FREE_THRESHOLD = -60
        self.OCCUPIED_THRESHOLD = 60

        # ESTRUCTURAS SLAM AVANZADAS
        self.pose_graph = PoseGraph()
        self.scan_history = deque(maxlen=200) # Historial de escaneos
        self.submaps = []
        self.loop_closures = []

        # Búferes de datos
        self.points_buffer = deque(maxlen=15000)
        self.trajectory = [] # Trayectoria optimizada
        self.raw_trajectory = deque(maxlen=3000) # Trayectoria odométrica
        self.total_points_processed = 0

        # Estados del sistema
        self.reference_pose = None
        self.last_scan = None
        self.last_optimization_time = 0
        self.optimization_interval = 10.0 # Segundos entre optimizaciones
        self.scan_matching_enabled = True
        self.loop_closure_enabled = True

        # Estadísticas
        self.stats = {
            'total_points': 0,
            'loop_closures_detected': 0,
            'optimizations_performed': 0,
            'scan_matches_refined': 0
        }

        logging.info(f"SLAM Avanzado inicializado - Mapa: {map_size}x{map_size}m, Res: {resolution}m")

    def start_mapping_session(self, initial_pose):
        """Iniciar nueva sesión de mapeo"""
        self.reference_pose = initial_pose
        self.scan_history.clear()
        self.submaps.clear()
        self.loop_closures.clear()
        self.trajectory = [initial_pose]
        self.raw_trajectory.clear()
        self.points_buffer.clear()
        self.total_points_processed = 0
        self.pose_graph = PoseGraph()
        self.pose_graph.add_pose(initial_pose)

        self.stats = {
            'total_points': 0,
            'loop_closures_detected': 0,
            'optimizations_performed': 0,
            'scan_matches_refined': 0
        }

        logging.info(f"Sesión de mapeo iniciada en: ({initial_pose[0]:.2f}, {initial_pose[1]:.2f})")

    def world_to_grid(self, x, y):
        """Conversión de coordenadas mundiales a grid"""
        gx = int(self.center - x / self.resolution)
        gy = int(self.center + y / self.resolution)
        return max(0, min(self.grid_size-1, gx)), max(0, min(self.grid_size-1, gy))

    def bresenham_line(self, x0, y0, x1, y1):
        """Algoritmo de Bresenham para trazado de rayos"""
        x0, y0, x1, y1 = int(x0), int(y0), int(x1), int(y1)
        points = []
        dx = abs(x1 - x0)
        dy = abs(y1 - y0)
        x, y = x0, y0
        sx = -1 if x0 > x1 else 1
        sy = -1 if y0 > y1 else 1

        if dx > dy:
            err = dx / 2.0
            while x != x1:
                points.append((x, y))
                err -= dy
                if err < 0:
                    y += sy
                    err += dx
                x += sx
        else:
            err = dy / 2.0
            while y != y1:
                points.append((x, y))
                err -= dx
                if err < 0:
                    x += sx
                    err += dy
                y += sy

        points.append((x, y))
        return points

    # LOOP CLOSURE DETECTION
    def detect_loop_closure(self, current_pose, current_scan):
        """Detectar si el robot ha vuelto a una posición ya visitada"""
        if len(self.scan_history) < 30:
            return None

        current_time = time.time()
        best_match_idx = None
        best_similarity = 0.0
        min_similarity_threshold = 0.75 # Umbral de similitud

        # Buscar en el historial (excluyendo escaneos recientes)
        search_start = max(0, len(self.scan_history) - 100)
        search_end = len(self.scan_history) - 15 # Excluir últimos 15 escaneos

        for i in range(search_start, search_end):
            historic_pose, historic_scan = self.scan_history[i]

            # Verificar proximidad espacial primero (más rápido)
            distance = math.sqrt((current_pose[0] - historic_pose[0])**2 +
                                 (current_pose[1] - historic_pose[1])**2)

            if distance < 1.5: # Radio de búsqueda de 1.5 metros
```

```
            # Calcular similitud de escaneos LIDAR
            similarity = self.calculate_scan_similarity(
                current_scan, historic_scan)

            if similarity > min_similarity_threshold and
                similarity > best_similarity:
                best_similarity = similarity
                best_match_idx = i

    if best_match_idx is not None:
        self.stats['loop_closures_detected'] += 1
        logging.info(f"LOOP CLOSURE detectado!
            Similaridad: {best_similarity:.3f}")

    return best_match_idx

def calculate_scan_similarity(self, scan1, scan2):
    """Calcular similitud entre dos escaneos LIDAR"""
    if len(scan1) == 0 or len(scan2) == 0:
        return 0.0

    # Tomar muestras equiespaciadas de ambos escaneos
        para comparar
    sample_size = min(72, len(scan1), len(scan2)) # 72
        muestras (5 grados)
    step1 = max(1, len(scan1) // sample_size)
    step2 = max(1, len(scan2) // sample_size)

    matches = 0
    total_comparisons = 0

    for i in range(0, min(len(scan1), len(scan2)), max(
        step1, step2)):
        if i >= len(scan1) or i >= len(scan2):
            break

        angle1, dist1 = scan1[i]
        angle2, dist2 = scan2[i]

        # Comparar distancias en ángulos similares
        if abs(angle1 - angle2) < math.radians(10): # 10
            grados de tolerancia
            if abs(dist1 - dist2) < 200: # 20cm de
                tolerancia
                matches += 1
            total_comparisons += 1

    return matches / total_comparisons if
        total_comparisons > 0 else 0.0

# SCAN MATCHING
def refine_pose_with_scan_matching(self, raw_pose,
    lidar_points):
    """Refinar la pose odométrica usando alineación con
        el mapa existente"""
    if not self.scan_matching_enabled or len(self.
        points_buffer) < 500:
        return raw_pose

    best_pose = raw_pose
    best_score = self.calculate_scan_alignment_score(
        raw_pose, lidar_points)

    # Búsqueda local en una rejilla 3x3x3
    position_step = 0.03 # 3cm
    angle_step = math.radians(1.5) # 1.5 grados

    for dx in [-position_step, 0, position_step]:
        for dy in [-position_step, 0, position_step]:
            for dtheta in [-angle_step, 0, angle_step]:
                test_pose = (
                    raw_pose[0] + dx,
                    raw_pose[1] + dy,
                    raw_pose[2] + dtheta
                )

                score = self.
                    calculate_scan_alignment_score(
                    test_pose, lidar_points)
                if score > best_score:
                    best_score = score
                    best_pose = test_pose

    if best_pose != raw_pose:
        self.stats['scan_matches_refined'] += 1

    return best_pose

def calculate_scan_alignment_score(self, pose,
    lidar_points):
    """Calcular score de alineación entre escaneo y mapa
        """
    if len(lidar_points) == 0:
        return 0.0

    score = 0.0
    valid_points = 0

    for angle_rad, dist_mm in lidar_points:
        # Filtrar puntos extremos
        if dist_mm < 100 or dist_mm > 8000: # 10cm a 8m
            continue

        # Aplicar rotación 230° del Transbot
        angle_rotated = angle_rad + math.radians(230)

        # Calcular coordenada global del punto
        dist_m = dist_mm / 1000.0
        global_angle = pose[2] + angle_rotated
        x_global = pose[0] + math.cos(global_angle) *
            dist_m
        y_global = pose[1] + math.sin(global_angle) *
            dist_m

        # Verificar coincidencia con el mapa
        grid_x, grid_y = self.world_to_grid(x_global,
            y_global)
        if 0 <= grid_x < self.grid_size and 0 <= grid_y
            < self.grid_size:
            occupancy = self.logodds[grid_y, grid_x]

            if occupancy > self.OCCUPIED_THRESHOLD:
                score += 2.0 # Excelente match - punto en
                    área ocupada
            elif occupancy > 0:
                score += 1.0 # Buen match - área
                    probablemente ocupada
            elif occupancy < self.FREE_THRESHOLD:
                score -= 1.0 # Mal match - punto en área
                    libre

            valid_points += 1

    return score / valid_points if valid_points > 0 else
        0.0

# POSE GRAPH OPTIMIZATION
def optimize_pose_graph(self):
    """Ejecutar optimización del grafo de poses si es
        necesario"""
    current_time = time.time()

    # Verificar si es tiempo de optimizar
    if (current_time - self.last_optimization_time <
        self.optimization_interval or
        len(self.loop_closures) == 0):
        return

    logging.info(f"Iniciando optimización del grafo de
        poses...")

    # Optimizar el grafo
    optimized_poses = self.pose_graph.optimize()

    if len(optimized_poses) == len(self.trajectory):
        self.trajectory = optimized_poses
        self.stats['optimizations_performed'] += 1
        logging.info(f"Optimización completada. Poses
            optimizadas: {len(optimized_poses)}")
    else:
        logging.warning("Error en optimización - tamañ
            os no coinciden")

    self.last_optimization_time = current_time

# SUBMAP MANAGEMENT
```

```
    def create_submap(self, center_pose):
        """Crear un submapa local alrededor de la pose actual"""
        submap = {
            'center_pose': center_pose,
            'radius': 2.0,
            'points': [],
            'timestamp': time.time(),
            'pose_index': len(self.trajectory) - 1
        }

        # Colectar puntos dentro del radio
        for point in self.points_buffer:
            dist = math.sqrt((point[0] - center_pose[0])**2 +
                             (point[1] - center_pose[1])**2)
            if dist <= submap['radius']:
                submap['points'].append(point)

        self.submaps.append(submap)
        return submap

    def find_matching_submap(self, current_pose):
        """Encontrar submapa que coincida con la pose actual"""
        if len(self.submaps) == 0:
            return None

        best_submap_idx = None
        min_distance = float('inf')

        for i, submap in enumerate(self.submaps):
            distance = math.sqrt(
                (current_pose[0] - submap['center_pose'][0])**2 +
                (current_pose[1] - submap['center_pose'][1])**2
            )

            if distance < min_distance and distance < submap['radius']:
                min_distance = distance
                best_submap_idx = i

        return best_submap_idx

    # MAIN MAPPING LOGIC
    def update_map(self, robot_x, robot_y, robot_yaw, lidar_points):
        """Actualización principal del mapa con todas las mejoras SLAM"""
        current_pose = (robot_x, robot_y, robot_yaw)
        current_time = time.time()

        # Inicialización si es la primera vez
        if self.reference_pose is None:
            self.start_mapping_session(current_pose)
            self.raw_trajectory.append(current_pose)
            return 0

        # 1. REFINAR POSE CON SCAN MATCHING
        refined_pose = current_pose
        if self.scan_matching_enabled and len(lidar_points) > 50:
            refined_pose = self.refine_pose_with_scan_matching(
                current_pose, lidar_points)

        # 2. DETECTAR LOOP CLOSURE
        loop_match_idx = None
        if self.loop_closure_enabled and len(self.scan_history) > 30:
            loop_match_idx = self.detect_loop_closure(
                refined_pose, lidar_points)

            if loop_match_idx is not None:
                historic_pose = self.scan_history[loop_match_idx][0]

                # Calcular transformación relativa
                dx = refined_pose[0] - historic_pose[0]
                dy = refined_pose[1] - historic_pose[1]
                dtheta = refined_pose[2] - historic_pose[2]

                # Normalizar ángulo
                while dtheta > math.pi:
                    dtheta -= 2 * math.pi
                while dtheta < -math.pi:
                    dtheta += 2 * math.pi

                # Agregar loop closure al grafo
                current_idx = len(self.trajectory)
                self.pose_graph.add_loop_closure(
                    loop_match_idx, current_idx, (dx, dy, dtheta))
                self.loop_closures.append((loop_match_idx,
                    current_idx, (dx, dy, dtheta)))

        # 3. ACTUALIZAR ESTRUCTURAS DE DATOS
        self.raw_trajectory.append(current_pose)
        self.trajectory.append(refined_pose)
        self.scan_history.append((refined_pose, lidar_points))
        self.pose_graph.add_pose(refined_pose)

        # 4. OPTIMIZAR GRAFO SI ES NECESARIO
        self.optimize_pose_graph()

        # 5. ACTUALIZAR MAPA DE OCUPACIÓN
        points_processed = self.update_occupancy_grid(
            refined_pose, lidar_points)

        # 6. MANEJO DE SUBMAPAS
        if len(self.trajectory) % 100 == 0: # Crear submapa cada 100 poses
            self.create_submap(refined_pose)

        # Log periódico de estadísticas
        if len(self.trajectory) % 50 == 0:
            self.log_statistics()

        return points_processed

    def update_occupancy_grid(self, pose, lidar_points):
        """Actualizar el grid de ocupación con los puntos LIDAR"""
        robot_x, robot_y, robot_yaw = pose
        points_processed = 0

        for angle_rad, dist_mm in lidar_points:
            # Validar distancia
            if dist_mm < 50 or dist_mm > 8000: # 5cm a 8m
                continue

            # Aplicar rotación 230° del Transbot
            rotation_angle = math.radians(230)
            angle_rad_rotated = angle_rad + rotation_angle

            # Convertir a coordenadas del sensor
            dist_m = dist_mm / 1000.0
            x_sensor = math.cos(angle_rad_rotated) * dist_m
            y_sensor = math.sin(angle_rad_rotated) * dist_m

            # Transformar a coordenadas globales
            cos_yaw, sin_yaw = math.cos(robot_yaw), math.sin(robot_yaw)
            x_global = cos_yaw * x_sensor - sin_yaw * y_sensor + robot_x
            y_global = sin_yaw * x_sensor + cos_yaw * y_sensor + robot_y

            # Convertir a coordenadas de grid
            x0g, y0g = self.world_to_grid(robot_x, robot_y)
            x1g, y1g = self.world_to_grid(x_global, y_global)

            # Trazar rayo y actualizar celdas
            line_cells = self.bresenham_line(x0g, y0g, x1g, y1g)
            if len(line_cells) < 2:
                continue

            # Actualizar celdas libres (excluyendo la última)
            for cx, cy in line_cells[:-1]:
                if 0 <= cx < self.grid_size and 0 <= cy < self.grid_size:
```

```
                # Solo actualizar si no está fuertemente ocupado
                if self.logodds[cy, cx] < self.OCCUPIED_THRESHOLD:
                    new_value = self.logodds[cy, cx] + self.FREE_INC
                    self.logodds[cy, cx] = max(self.LOGODDS_MIN, min(self.LOGODDS_MAX, new_value))

        # Actualizar celda ocupada (última celda)
        lx, ly = line_cells[-1]
        if 0 <= lx < self.grid_size and 0 <= ly < self.grid_size:
            new_value = self.logodds[ly, lx] + self.OCCUPIED_INC
            self.logodds[ly, lx] = max(self.LOGODDS_MIN, min(self.LOGODDS_MAX, new_value))

        # Guardar punto para visualización
        self.points_buffer.append((x_global, y_global))
        points_processed += 1

    # Actualizar estadísticas
    self.total_points_processed += points_processed
    self.stats['total_points'] = self.total_points_processed

    return points_processed

def get_occupancy_image(self):
    """Generar imagen de ocupación del mapa"""
    img = np.full((self.grid_size, self.grid_size), 127, dtype=np.uint8)

    # Aplicar umbrales
    img[self.logodds < self.FREE_THRESHOLD] = 255 # Libre - blanco
    img[self.logodds < -20] = 200 # Probablemente libre
    img[self.logodds == 0] = 127 # Desconocido - gris
    img[self.logodds > 20] = 60 # Probablemente ocupado
    img[self.logodds > self.OCCUPIED_THRESHOLD] = 0 # Ocupado - negro

    return img

def save_data(self, filename_prefix):
    """Guardar datos del mapa"""
    try:
        # Guardar datos numéricos
        np.save(f'{filename_prefix}_logodds.npy', self.logodds)

        if self.trajectory:
            trajectory_array = np.array(self.trajectory)
            np.save(f'{filename_prefix}_trajectory.npy', trajectory_array)

        # Guardar imagen del mapa
        img = self.get_occupancy_image()

        fig, ax = plt.subplots(figsize=(12, 10))
        ax.imshow(img, cmap='gray', extent=self.get_map_extent())
        ax.set_xlabel('X (metros)')
        ax.set_ylabel('Y (metros)')
        ax.grid(True, alpha=0.3)
        ax.set_title(f'Mapa SLAM - {filename_prefix}\n'
                     f'Puntos: {self.total_points_processed},'
                     f'Loop Closures: {self.stats["loop_closures_detected"]}')

        plt.tight_layout()
        plt.savefig(f'{filename_prefix}_map.png', dpi=300, bbox_inches='tight')
        plt.close()

        # Log de estadísticas finales
        logging.info(f"Mapa guardado: {filename_prefix}")
        logging.info(f"   - Puntos procesados: {self.total_points_processed}")
        logging.info(f"   - Loop closures: {self.stats['loop_closures_detected']}")
        logging.info(f"   - Optimizaciones: {self.stats['optimizations_performed']}")
        logging.info(f"   - Poses refinadas: {self.stats['scan_matches_refined']}")

        return True

    except Exception as e:
        logging.error(f"Error guardando mapa: {e}")
        return False

def get_map_extent(self):
    """Obtener extensión del mapa en metros"""
    half_size = self.map_size / 2
    return [-half_size, half_size, -half_size, half_size]

def reset_map(self):
    """Reiniciar completamente el mapa"""
    self.logodds = np.zeros((self.grid_size, self.grid_size), dtype=np.float32)
    self.reference_pose = None
    self.scan_history.clear()
    self.submaps.clear()
    self.loop_closures.clear()
    self.trajectory.clear()
    self.raw_trajectory.clear()
    self.points_buffer.clear()
    self.total_points_processed = 0
    self.pose_graph = PoseGraph()

    self.stats = {
        'total_points': 0,
        'loop_closures_detected': 0,
        'optimizations_performed': 0,
        'scan_matches_refined': 0
    }

    logging.info("Mapa reiniciado completamente")

def enable_correction(self, enabled):
    """Habilitar/deshabilitar correcciones (para compatibilidad)"""
    self.scan_matching_enabled = enabled
    self.loop_closure_enabled = enabled
    status = "HABILITADAS" if enabled else "DESHABILITADAS"
    logging.info(f"Correcciones SLAM {status}")

def log_statistics(self):
    """Log de estadísticas del sistema"""
    logging.info(
        f"SLAM Stats: "
        f"Puntos: {self.stats['total_points']} | "
        f"Loops: {self.stats['loop_closures_detected']} | "
        f"Opts: {self.stats['optimizations_performed']} | "
        f"Refinements: {self.stats['scan_matches_refined']}"
    )

def get_statistics(self):
    """Obtener estadísticas del mapeo"""
    return self.stats.copy()
```

rplidar_express.py

```python
#!/usr/bin/env python3
# rplidar_express.py
# Driver real para RPLidar - Con debug detallado

import serial
import time
import numpy as np
import logging

class RPLidarExpress:
    def __init__(self, port='/dev/ttyUSB1', baudrate=460800):
        try:
            self.serial = serial.Serial(port, baudrate, timeout=1)
            self.scanning = False
            logging.debug("RPLidarExpress inicializado correctamente")
        except Exception as e:
            logging.error(f"Error inicializando RPLidar: {e}")
            raise

    def send_command(self, cmd, payload=None):
        try:
            if payload is None:
                packet = bytes([0xA5, cmd])
                checksum = 0xA5 ^ cmd
            else:
                payload_size = len(payload)
                packet = bytes([0xA5, cmd, payload_size]) + payload
                checksum = 0xA5 ^ cmd ^ payload_size
                for b in payload:
                    checksum ^= b
                packet += bytes([checksum])
            self.serial.write(packet)
            time.sleep(0.01)
        except Exception as e:
            logging.error(f"Error enviando comando: {e}")

    def read_descriptor(self):
        start_time = time.time()
        while time.time() - start_time < 5:
            if self.serial.read(1) == b'\xA5':
                if self.serial.read(1) == b'\x5A':
                    descriptor = self.serial.read(5)
                    return descriptor
        logging.error("Timeout leyendo descriptor")
        return None

    def start_express_scan(self):
        logging.debug("Iniciando escaneo express...")
        self.send_command(0x82, b'\x00\x00\x00\x00\x00')
        descriptor = self.read_descriptor()
        if descriptor == b'\x54\x00\x00\x40\x85':
            self.scanning = True
            logging.debug("Escaneo express iniciado correctamente")
            return True
        else:
            logging.error(f"Descriptor invalido: {descriptor}")
            return False

    def stop_scan(self):
        if self.scanning:
            logging.debug("Deteniendo escaneo...")
            self.send_command(0x25)
            self.scanning = False
            time.sleep(0.5)
            logging.debug("Escaneo detenido")
        return True

    def decode_express_packet(self, packet, angle_prev):
        if len(packet) != 84:
            #logging.warning(f"Paquete de tamaño incorrecto: {len(packet)} bytes")
            return [], angle_prev

        if packet[0] >> 4 != 0xA or (packet[1] >> 4) != 0x5:
            #logging.warning("Cabecera de paquete inválida")
            return [], angle_prev

        angle_q6 = (packet[2] | (packet[3] << 8)) & 0x7FFF
        angle_start = (angle_q6 / 64.0) % 360.0

        cabins = packet[4:84]
        points = []

        for i in range(0, len(cabins), 2):
            byte1 = cabins[i]
            byte2 = cabins[i+1]

            dist = ((byte2 & 0x3F) << 8) | byte1
            dist_mm = dist
            if dist_mm == 0:
                continue

            index = i // 2
            angle_diff = (angle_start - angle_prev + 360) % 360
            angle = angle_prev + angle_diff * (index / 40.0)
            angle_rad = np.radians(angle)
            points.append((angle_rad, dist_mm))

        return points, angle_start

    def iter_packets(self):
        angle_prev = 0.0
        packet_count = 0
        error_count = 0
        max_errors = 10

        while self.scanning and error_count < max_errors:
            try:
                packet = self.serial.read(84)
                if len(packet) == 84:
                    points, angle_prev = self.decode_express_packet(packet, angle_prev)
                    packet_count += 1
                    if packet_count % 50 == 0:
                        logging.debug(f"Paquetes LIDAR procesados: {packet_count}")
                    yield points
                    error_count = 0 # Reset error count on success
                else:
                    logging.warning(f"Paquete incompleto ({len(packet)} bytes)")
                    error_count += 1
                    time.sleep(0.01)
            except Exception as e:
                logging.error(f"Error procesando paquete LIDAR: {e}")
                error_count += 1
                time.sleep(0.1)

        if error_count >= max_errors:
            logging.error("Demasiados errores en LIDAR, deteniendo escaneo")
            self.stop_scan()

    def close(self):
        if self.scanning:
            self.stop_scan()
        self.serial.close()
        logging.debug("LIDAR cerrado correctamente")
```

trajectory_planner.py

```python
#!/usr/bin/env python3
# trajectory_planner.py
# Planificador de trayectorias suaves

import numpy as np
import math
from scipy.interpolate import CubicSpline
import matplotlib.pyplot as plt

class TrajectoryPlanner:
    def __init__(self):
        self.waypoints = []
        self.smooth_trajectory = []
        self.max_velocity = 0.2 # m/s
        self.max_angular_velocity = 0.5 # rad/s
        self.lookahead_distance = 0.3 # metros

    def add_waypoint(self, x, y):
        """Agregar un punto de control"""
        self.waypoints.append((x, y))

    def clear_waypoints(self):
        """Limpiar todos los puntos de control"""
        self.waypoints.clear()
        self.smooth_trajectory.clear()

    def generate_smooth_trajectory(self, num_points=100):
        """Generar trayectoria suave usando splines cúbicos"""
        if len(self.waypoints) < 2:
            return []

        # Separar coordenadas
        x_coords = [p[0] for p in self.waypoints]
        y_coords = [p[1] for p in self.waypoints]

        # Crear parámetro de distancia acumulada
        distances = [0]
        for i in range(1, len(self.waypoints)):
            dx = x_coords[i] - x_coords[i-1]
            dy = y_coords[i] - y_coords[i-1]
            distances.append(distances[-1] + math.hypot(dx, dy))

        # Crear splines cúbicos
        try:
            cs_x = CubicSpline(distances, x_coords)
            cs_y = CubicSpline(distances, y_coords)

            # Generar puntos suavizados
            t_smooth = np.linspace(0, distances[-1], num_points)
            x_smooth = cs_x(t_smooth)
            y_smooth = cs_y(t_smooth)

            self.smooth_trajectory = list(zip(x_smooth, y_smooth))
            return self.smooth_trajectory

        except Exception as e:
            print(f"Error generando trayectoria suave: {e}")
            # Fallback: usar waypoints originales
            self.smooth_trajectory = self.waypoints.copy()
            return self.smooth_trajectory

    def get_lookahead_point(self, current_pos, lookahead_dist=None):
        """Obtener punto de lookahead para control Pure Pursuit"""
        if lookahead_dist is None:
            lookahead_dist = self.lookahead_distance

        if not self.smooth_trajectory:
            return None

        current_x, current_y = current_pos

        # Encontrar el punto más cercano en la trayectoria
        closest_idx = 0
        min_dist = float('inf')

        for i, (x, y) in enumerate(self.smooth_trajectory):
            dist = math.hypot(x - current_x, y - current_y)
            if dist < min_dist:
                min_dist = dist
                closest_idx = i

        # Buscar punto a la distancia de lookahead
        for i in range(closest_idx, len(self.smooth_trajectory)):
            x, y = self.smooth_trajectory[i]
            dist = math.hypot(x - current_x, y - current_y)
            if dist >= lookahead_dist:
                return (x, y), i

        # Si no se encuentra, usar el último punto
        return self.smooth_trajectory[-1], len(self.smooth_trajectory) - 1

def calculate_curvature(self, point1, point2, point3):
    """Calcular curvatura entre tres puntos"""
    x1, y1 = point1
    x2, y2 = point2
    x3, y3 = point3

    # Vectores
    dx1 = x2 - x1
    dy1 = y2 - y1
    dx2 = x3 - x2
    dy2 = y3 - y2

    # Producto cruz
    cross = dx1 * dy2 - dy1 * dx2

    # Normas
    norm1 = math.hypot(dx1, dy1)
    norm2 = math.hypot(dx2, dy2)

    if norm1 * norm2 == 0:
        return 0

    # Curvatura
    curvature = 2 * cross / (norm1 * norm2)
    return curvature

def get_velocity_profile(self):
    """Generar perfil de velocidad basado en curvatura"""
    if len(self.smooth_trajectory) < 3:
        return [self.max_velocity] * len(self.smooth_trajectory)

    velocities = []

    for i in range(len(self.smooth_trajectory)):
        if i == 0 or i == len(self.smooth_trajectory) - 1:
            velocities.append(0) # Detenerse al inicio y final
        else:
            # Calcular curvatura
            curvature = abs(self.calculate_curvature(
                self.smooth_trajectory[i-1],
                self.smooth_trajectory[i],
                self.smooth_trajectory[i+1]
            ))

            # Reducir velocidad en curvas cerradas
            speed = self.max_velocity * (1 - min(1.0, curvature * 2))
            velocities.append(max(0.05, speed)) # Velocidad mínima

    return velocities

def is_trajectory_feasible(self, robot_width=0.3, clearance=0.2):
    """Verificar si la trayectoria es factible (simulación simple)"""
    # En una implementación real, aquí se verificarían colisiones
    # con el mapa existente o con obstáculos conocidos
    if len(self.smooth_trajectory) < 2:
        return False, "Trayectoria demasiado corta"

    # Verificar curvaturas muy cerradas
```

```
        max_curvature = 0
        for i in range(1, len(self.smooth_trajectory) - 1):
            curvature = abs(self.calculate_curvature(
                self.smooth_trajectory[i-1],
                self.smooth_trajectory[i],
                self.smooth_trajectory[i+1]
            ))
            max_curvature = max(max_curvature, curvature)

        if max_curvature > 2.0: # Curvatura muy cerrada
            return False, f"Curvatura demasiado cerrada: {
                max_curvature:.2f}"

        return True, "Trayectoria factible"

    def visualize_trajectory(self):
        """Visualizar trayectoria planeada"""
        if not self.smooth_trajectory:
            return

        x_coords = [p[0] for p in self.smooth_trajectory]
        y_coords = [p[1] for p in self.smooth_trajectory]

        plt.figure(figsize=(10, 8))

        # Trayectoria suave
        plt.plot(x_coords, y_coords, 'b-', linewidth=2,
                 label='Trayectoria Suave')

        # Waypoints originales
        if self.waypoints:
            x_orig = [p[0] for p in self.waypoints]
            y_orig = [p[1] for p in self.waypoints]
            plt.plot(x_orig, y_orig, 'ro--', linewidth=1,
                     label='Waypoints')

        plt.grid(True, alpha=0.3)
        plt.axis('equal')
        plt.xlabel('X (metros)')
        plt.ylabel('Y (metros)')
        plt.title('Trayectoria Planificada')
        plt.legend()
        plt.show()
```

obstacle_avoider.py

```
#!/usr/bin/env python3
# obstacle_avoider.py
# Sistema de evasión de obstáculos usando LIDAR

import math
import numpy as np
from collections import deque

class ObstacleAvoider:
    def __init__(self):
        self.safety_distance = 0.3 # metros
        self.critical_distance = 0.15 # metros
        self.avoidance_distance = 0.5 # metros
        self.obstacle_history = deque(maxlen=10)
        self.avoidance_active = False
        self.avoidance_direction = 0 # -1: izquierda, 1:
            derecha

    def check_obstacles(self, lidar_points, robot_pose):
        """Verificar obstáculos en la trayectoria"""
        if not lidar_points:
            return False

        x, y, yaw = robot_pose
        obstacles_in_path = []

        for angle, distance in lidar_points:
            if distance < self.avoidance_distance:
                # Convertir a coordenadas cartesianas
                    relativas
                rel_x = distance * math.cos(angle + yaw)
                rel_y = distance * math.sin(angle + yaw)

                # Verificar si está en el camino frontal
                if abs(rel_y) < 0.2 and rel_x > 0: # En
                    frente del robot
                    obstacles_in_path.append((rel_x, rel_y,
                        distance))

                # Obstáculos críticos (muy cerca)
                if distance < self.critical_distance:
                    return True

        # Promediar obstáculos detectados
        if obstacles_in_path:
            avg_distance = np.mean([d for _, _, d in
                obstacles_in_path])
            if avg_distance < self.safety_distance:
                self.obstacle_history.append(True)
            return True

        self.obstacle_history.append(False)

        # Considerar obstáculo si hay varios positivos en el
            historial
        return sum(self.obstacle_history) >= 5

    def avoid_obstacle(self, lidar_points, robot_pose,
        target, current_speed, current_angular):
        """Calcular maniobra de evasión de obstáculo"""
        x, y, yaw = robot_pose
        target_x, target_y = target

        # Analizar sectores LIDAR
        left_obstacles = []
        right_obstacles = []
        front_obstacles = []

        for angle, distance in lidar_points:
            if distance < self.avoidance_distance:
                rel_angle = angle # Ya está en coordenadas
                    del robot

                # Sector frontal (-30° a 30°)
                if abs(rel_angle) < math.radians(30):
                    front_obstacles.append(distance)
                # Sector izquierdo (30° a 90°)
                elif math.radians(30) <= rel_angle <= math.
                    radians(90):
                    left_obstacles.append(distance)
                # Sector derecho (-90° a -30°)
                elif math.radians(-90) <= rel_angle <= math.
                    radians(-30):
                    right_obstacles.append(distance)

        # Calcular distancias mínimas
        min_front = min(front_obstacles) if front_obstacles
            else float('inf')
        min_left = min(left_obstacles) if left_obstacles
            else float('inf')
        min_right = min(right_obstacles) if right_obstacles
            else float('inf')

        # Determinar dirección de evasión
        if not self.avoidance_active:
            # Elegir dirección con más espacio
            if min_left > min_right:
                self.avoidance_direction = -1 # Girar
                    izquierda
```

```
            else:
                self.avoidance_direction = 1 # Girar derecha
            self.avoidance_active = True

        # Calcular velocidades de evasión
        if min_front < self.critical_distance:
            # Obstáculo crítico - retroceder y girar
            new_speed = -0.1
            new_angular = 0.5 * self.avoidance_direction
        elif min_front < self.safety_distance:
            # Obstáculo cercano - reducir velocidad y girar
            speed_factor = min_front / self.safety_distance
            new_speed = current_speed * speed_factor * 0.5
            new_angular = 0.8 * self.avoidance_direction
        else:
            # Mantener evasión pero con menos agresividad
            new_speed = current_speed * 0.8
            new_angular = 0.3 * self.avoidance_direction

        # Verificar si podemos volver a la trayectoria
        if self.can_return_to_path(robot_pose, target,
              lidar_points):
            self.avoidance_active = False
            # Calcular ángulo hacia el objetivo
            target_angle = math.atan2(target_y - y, target_x
                  - x)
            angle_error = target_angle - yaw
            new_angular = 2.0 * angle_error
            new_speed = current_speed

        return new_speed, new_angular

    def can_return_to_path(self, robot_pose, target,
          lidar_points):
        """Verificar si es seguro volver a la trayectoria
              original"""
        x, y, yaw = robot_pose
        target_x, target_y = target

        # Calcular ángulo hacia el objetivo
        target_angle = math.atan2(target_y - y, target_x - x
              )

        # Verificar obstáculos en dirección al objetivo
        for angle, distance in lidar_points:
            if distance < self.safety_distance:
                # Verificar si el obstáculo está en dirección
                      al objetivo
                angle_diff = abs(angle - target_angle)
                if angle_diff < math.radians(45): # Dentro de
                      45° del objetivo
                    return False

        return True

    def get_obstacle_map(self, lidar_points, robot_pose):
        """Generar mapa local de obstáculos"""
        x, y, yaw = robot_pose
        obstacle_points = []

        for angle, distance in lidar_points:
            if distance < self.avoidance_distance:
                # Convertir a coordenadas globales
                global_x = x + distance * math.cos(angle +
                      yaw)
                global_y = y + distance * math.sin(angle +
                      yaw)
                obstacle_points.append((global_x, global_y))

        return obstacle_points
```