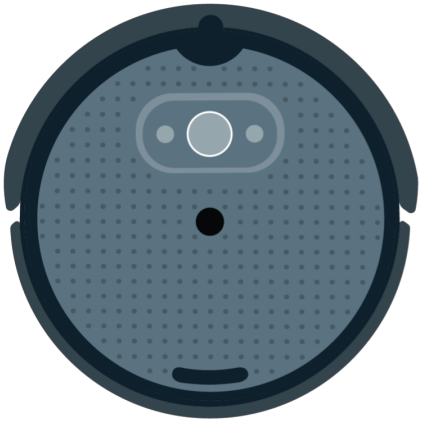
Trabajo Fin de Grado
Ingeniería en Tecnologías Industriales



Integración de métodos de percepción, control y localización para navegación autónoma en plataforma Create 3

Autor: Antonio López García

Tutor: José María Maestre Torreblanca

Dpto. Ingeniería de Sistemas y Automática
Escuela Técnica Superior de Ingeniería
Universidad de Sevilla

Sevilla, 2025

Trabajo Fin de Grado
Ingeniería en Tecnologías Industriales

Integración de métodos de percepción, control y localización para navegación autónoma en plataforma Create 3

Autor:
Antonio López García

Tutor:
José María Maestre Torreblanca
Catedrático de Universidad

Dpto. Ingeniería de Sistemas y Automática
Escuela Técnica Superior de Ingeniería
Universidad de Sevilla

Sevilla, 2025

Trabajo Fin de Grado: Integración de métodos de percepción, control y localización para navegación autónoma en plataforma Create 3

Autor: Antonio López García
Tutor: José María Maestre Torreblanca

El tribunal nombrado para juzgar el trabajo arriba indicado, compuesto por los siguientes profesores:

Presidente:

Vocal/es:

Secretario:

acuerdan otorgarle la calificación de:

El Secretario del Tribunal

Fecha:

Agradecimientos

Con la finalización de este proyecto cierro una de las etapas más importantes de mi vida, en la cual he aprendido y crecido muchísimo como persona. Sin embargo, esto no hubiese sido posible sin el apoyo incondicional de mi entorno, formado por las personas más importantes de mi vida.

En primer lugar, quiero hacer una especial mención a mi familia, mi padre, mi madre y mi hermana. Gracias por vuestra comprensión y cariño durante todos estos años, por la confianza depositada en mí, y sobre todo por escucharme y apoyarme siempre en los momentos difíciles. También quiero agradecer a los demás miembros de mi familia. Especialmente a mis dos abuelas, quienes no han podido estar hoy aquí para verme, pero nunca dudaron que lo conseguiría.

En segundo lugar, dar las gracias a mis amigos, a los de toda la vida, sois una parte esencial de mi vida y todavía nos queda mucho camino por recorrer. Mencionar también a los amigos que he ido haciendo en esta etapa universitaria y que han hecho de ella una experiencia increíble.

Por último, agradecer a mi tutor su amabilidad y apoyo.

Antonio López García
Sevilla, 2025

Resumen

Este Trabajo Fin de Grado presenta el desarrollo de un sistema experimental de navegación autónoma en la plataforma iRobot Create 3, empleando el sistema de localización Marvelmind, que proporciona posicionamiento de alta precisión en entornos interiores. El proyecto combina técnicas de control, percepción y localización, con especial atención a la validación de la odometría interna frente a referencias externas. El sistema de navegación integra sensores infrarrojos para la detección de obstáculos, maniobras de evasión y mecanismos de seguridad. La estimación de posición se optimiza mediante un filtro de Kalman bidimensional, que mejora la estabilidad y precisión de las mediciones. Asimismo, se ha explorado la incorporación de visión artificial a través de YOLOv8n sobre una cámara IP, estableciendo una base para futuras ampliaciones en percepción visual, aunque limitada por el hardware disponible. Todo el desarrollo se implementó en Python, utilizando librerías específicas para comunicación con el robot, procesamiento de datos y visualización en tiempo real. El trabajo constituye una plataforma experimental que integra control, localización y percepción, orientada a la mejora de la autonomía en robots móviles de interior.

Abstract

This Bachelor's Thesis presents the development of an experimental autonomous navigation system on the iRobot Create 3 platform, using the Marvelmind system to provide high-precision indoor positioning. The project integrates control, perception, and localization techniques, with a strong focus on validating the robot's internal odometry against external reference measurements. The navigation system incorporates infrared sensors for obstacle detection, evasive maneuvers, and safety mechanisms. Position estimation is enhanced with a two-dimensional Kalman filter, which improves the stability and accuracy of ultrasonic measurements. Additionally, the project explored the integration of computer vision through YOLOv8n and an IP camera, laying the groundwork for future extensions in visual perception, though constrained by hardware limitations. The entire system was implemented in Python, relying on specialized libraries for robot communication, data processing, and real-time visualization. The outcome is an experimental platform that combines control, localization, and perception, aimed at advancing indoor mobile robot autonomy.

Índice

Nomenclatura

Introducción

IPS	Indoor Positioning Systems.
SDK	Software Development Kit.

Marco teórico y tecnológico

LIDAR	Light Detection and Ranging.
YOLO	You Only Look Once.
BLE	Bluetooth Low Energy.
RFID	Radio Frequency Identification.
RSSI	Received Signal Strength Indicator.
RTSP	Real Time Streaming Protocol.

Conclusiones

SLAM	Simultaneous Localization and Mapping.

Notación

d	Distancia al objeto calculada mediante el tiempo de vuelo ultrasónico: $$d = \frac{t_{\text{vuelo}} \cdot v_{\text{sonido}}}{2}$$
e	Fórmula euclidiana para el cálculo de error entre odometría interna y Marvelmind(ultrasonidos): $$e = \sqrt{(x_{\text{Roomba}} - x_{\text{Marvelmind}})^2 + (y_{\text{Roomba}} - y_{\text{Marvelmind}})^2}$$
E	Fórmula euclidiana para el cálculo de error entre posición real y Marvelmind(ultrasonidos): $$E = \sqrt{(x_{\text{real}} - x_{\text{mm}})^2 + (y_{\text{real}} - y_{\text{mm}})^2}$$
D	Fórmula euclidiana para el cálculo de distancia entre el punto inicial y el punto objetivo: $$D = \sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2}$$
θ_{objetivo}	Fórmula para el cálculo del ángulo del robot hacia el destino. $$\theta_{\text{objetivo}} = \text{atan2}(y_{\text{dest}} - y_{\text{actual}},\, x_{\text{dest}} - x_{\text{actual}})$$
$\Delta\theta$	Fórmula para el cálculo de la diferencia mínima entre los dos ángulos en grados. $$\Delta\theta = \big((\theta_2 - \theta_1 + 180) \text{ mód } 360\big) - 180$$

Índice de Figuras

Índice de Tablas

1 Introducción

1.1 Contexto y motivación

En las últimas décadas, la robótica móvil ha experimentado un avance vertiginoso, consolidándose como una disciplina clave dentro de la automatización inteligente. Su impacto se extiende a numerosos sectores, desde la industria hasta los servicios y la investigación, donde los robots móviles permiten optimizar procesos, incrementar la seguridad y explorar nuevas formas de interacción hombre-máquina.

Un aspecto fundamental en este campo es la localización precisa. Disponer de un posicionamiento fiable permite a los robots ejecutar tareas de forma autónoma, segura y eficiente. Aplicaciones como la logística en almacenes, el guiado de visitantes en museos, la asistencia en hospitales o la inspección en entornos industriales dependen directamente de la capacidad de un robot para conocer su posición y moverse de manera controlada en espacios interiores.

Sin embargo, lograr este objetivo no está exento de desafíos. Los sistemas de odometría, aunque sencillos de implementar, sufren una notable acumulación de error a medida que el robot se desplaza, debido a factores físicos como el deslizamiento de ruedas o las irregularidades del terreno. Por otra parte, los sistemas de localización absoluta en interiores (IPS), basados en tecnologías como ultrasonidos, radiofrecuencia o visión artificial, proporcionan medidas externas que corrigen esos errores, aunque también presentan limitaciones propias.

La combinación de ambas fuentes de información mediante técnicas de fusión sensorial es, por tanto, esencial para mejorar la robustez y precisión del posicionamiento. En este contexto, la plataforma iRobot Create 3 se presenta como una base idónea para la experimentación, al ofrecer telemetría y odometría en tiempo real a través de su SDK. Como referencia externa, en este trabajo se emplea el sistema Marvelmind, que utiliza triangulación acústica para ofrecer coordenadas de alta precisión en entornos interiores, lo que permite validar y mejorar la información de odometría.

Figura 1.1 Balizas Marvelmind.

El punto clave de este proyecto radica en el estudio, desarrollo e implementación de un sistema experimental que combine percepción, control y localización, con el fin de dotar a la plataforma Create 3 de capacidades avanzadas de navegación autónoma en entornos interiores.

1.2 Objetivo

El objetivo de este proyecto es diseñar, implementar y validar un sistema experimental de navegación autónoma en interiores sobre la plataforma iRobot Create 3, integrando diferentes técnicas de control, localización y percepción.

El trabajo no se limita a la evaluación comparativa de sensores, sino que persigue la construcción de un sistema de navegación autónoma capaz de desplazarse en un entorno interior evitando obstáculos y garantizando la seguridad en su movimiento. En este contexto, el filtro de Kalman bidimensional desempeña un papel central, ya que permite fusionar las lecturas del sistema Marvelmind con las estimaciones de la odometría para obtener un posicionamiento más estable y preciso.

Como extensión experimental, se ha explorado la integración de visión artificial mediante el uso de YOLOv8n y una cámara IP, lo que abre la puerta a incorporar capacidades avanzadas de percepción visual en versiones futuras. En conjunto, el proyecto busca consolidar una plataforma modular que combine localización, control y percepción, sentando las bases para la mejora de la autonomía de robots móviles en entornos interiores.

Por ello, se pretenden integrar 4 bloques claramente diferenciados, en los cuales se divide este documento:

1. Control manual y validación de odometría, desarrollando herramientas para el pilotaje remoto del robot y la evaluación de la precisión de sus sensores internos.
2. Estimación del error de posicionamiento, comparando la odometría del robot con las medidas proporcionadas por el sistema Marvelmind.
3. Navegación autónoma, incorporando sensores infrarrojos, estrategias de evasión de obstáculos y un filtro de Kalman bidimensional que mejora la precisión y estabilidad de la localización.
4. Exploración de visión artificial, mediante la integración experimental de YOLOv8n y una cámara IP, como primer paso hacia la percepción visual.

El resultado buscado es un sistema modular y extensible que combine posicionamiento, control y percepción, sirviendo como base para futuras mejoras en autonomía y capacidades de los robots móviles en entornos interiores.

2 Marco teórico y tecnológico

2.1 Fundamentos de la robótica móvil

Un robot móvil autónomo representa uno de los mayores desafíos de la ingeniería moderna, ya que debe ser capaz de ejecutar tareas complejas sin intervención humana directa. Para lograr esta autonomía, el sistema robótico debe interpretar continuamente su entorno y adaptar su comportamiento en función de la información percibida, lo que requiere la integración coherente de múltiples subsistemas especializados.

2.1.1 Localización

La localización constituye el pilar fundamental de cualquier sistema de navegación autónoma, ya que consiste en determinar con precisión la posición y orientación actual del robot respecto a un sistema de coordenadas conocido. Sin esta información básica, resulta imposible planificar movimientos coherentes o ejecutar tareas que requieran conocimiento espacial.

Los métodos de localización pueden clasificarse en tres categorías principales según su principio de funcionamiento. La odometría representa el enfoque más básico, basándose en la información proporcionada por los encoders de las ruedas para estimar el desplazamiento relativo del robot. Este método, aunque sencillo de implementar y de respuesta inmediata, presenta la limitación inherente de la acumulación progresiva de error. Fenómenos como el deslizamiento de las ruedas, las irregularidades en la superficie de rodadura, o las imprecisiones en los parámetros del modelo cinemático contribuyen a que la estimación de posición se degrade gradualmente con el tiempo y la distancia recorrida.

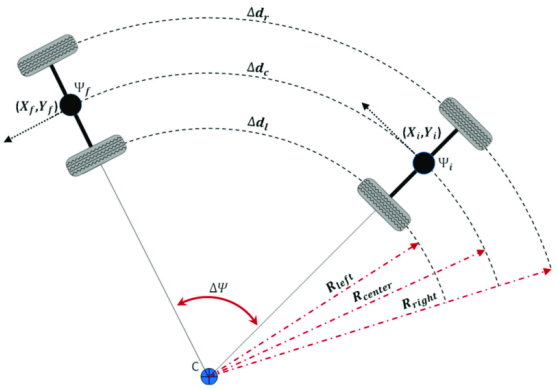


Figura 2.1 Modelo de odometría por ruedas para robots con tracción diferencial.

Por esta razón, la odometría suele complementarse con sistemas de localización absoluta que proporcionan referencias externas independientes de la historia de movimiento del robot. Estos sistemas, englobados dentro de la categoría de Indoor Positioning Systems (IPS), permiten obtener la posición absoluta en entornos cerrados donde el GPS tradicional no resulta operativo debido a la atenuación de señal.

La combinación óptima se logra mediante técnicas de fusión sensorial que integran ambas fuentes de información. Esta aproximación permite aprovechar las ventajas de cada método: la alta frecuencia de

actualización y continuidad de la odometría, junto con la precisión absoluta y la ausencia de deriva acumulativa de los sistemas externos. El filtro de Kalman, que será descrito en detalle más adelante, representa una de las técnicas más efectivas para realizar esta fusión de manera estadísticamente óptima.

2.1.2 Percepción del entorno

La capacidad de percepción permite al robot detectar, identificar y caracterizar los elementos presentes en su entorno operativo. Esta información resulta crucial para la navegación segura, especialmente en entornos dinámicos donde pueden aparecer obstáculos no previstos en el mapa inicial.

Los sensores de proximidad constituyen la primera línea de defensa contra colisiones. Los sensores infrarrojos, utilizados en este proyecto, emiten pulsos de luz infrarroja y miden la intensidad de la reflexión para determinar la presencia de objetos cercanos. Aunque su alcance es limitado y pueden verse afectados por las condiciones de iluminación, ofrecen una respuesta rápida y son especialmente efectivos para la detección de obstáculos a corta distancia. Los sensores ultrasónicos operan bajo un principio similar pero utilizando ondas acústicas, lo que los hace más robustos ante variaciones lumínicas, aunque más sensibles a las condiciones atmosféricas y a la geometría de las superficies.

Figura 2.2 Sensor LIDAR.

Los sistemas LIDAR (Light Detection and Ranging) representan una tecnología más avanzada que proporciona mediciones precisas de distancia en múltiples direcciones simultáneamente. Mediante el uso de láser y la medición de tiempo de vuelo, estos sensores pueden generar mapas bidimensionales o tridimensionales del entorno con alta resolución espacial. Sin embargo, su mayor complejidad y coste los hace más apropiados para aplicaciones que requieren mapeo detallado del entorno.

Los sensores de contacto, como los bumpers integrados en la plataforma Create 3, actúan como último recurso de seguridad. Cuando el robot entra en contacto físico con un obstáculo, estos sensores detectan la presión mecánica y pueden activar rutinas de emergencia para evitar daños.

La visión por computador representa el paradigma más avanzado de percepción, utilizando cámaras y algoritmos de procesamiento de imágenes para extraer información semántica del entorno. En este proyecto se ha explorado experimentalmente el uso de YOLOv8n (You Only Look Once), una arquitectura de red neuronal optimizada para la detección de objetos en tiempo real. Aunque las limitaciones de hardware impidieron su integración completa en el sistema final, esta aproximación establece las bases para futuras expansiones en capacidades de percepción visual.

2.1.3 Planificación de la trayectoria

La planificación de trayectorias aborda el problema de determinar una ruta óptima desde la posición actual del robot hasta un punto objetivo, considerando las restricciones impuestas por los obstáculos conocidos y las limitaciones cinemáticas del vehículo.

La planificación global utiliza mapas completos del entorno para calcular trayectorias óptimas antes de iniciar el movimiento. Algoritmos como A*, RRT (Rapidly-exploring Random Trees) o campos de potencial artificial pueden generar rutas que minimicen criterios como la distancia total, el tiempo de recorrido o el consumo energético. Sin embargo, esta aproximación requiere conocimiento previo completo del entorno y puede no ser adecuada para espacios dinámicos.

En contraste, la planificación local o reactiva adapta la trayectoria en tiempo real basándose en la información proporcionada por los sensores. Este enfoque, implementado en el presente proyecto, resulta más robusto ante cambios inesperados en el entorno, aunque puede llevar a soluciones subóptimas desde el punto de vista global. La estrategia reactiva implementada combina navegación hacia el objetivo con maniobras evasivas automáticas cuando se detectan obstáculos, permitiendo al robot adaptarse dinámicamente a condiciones cambiantes.

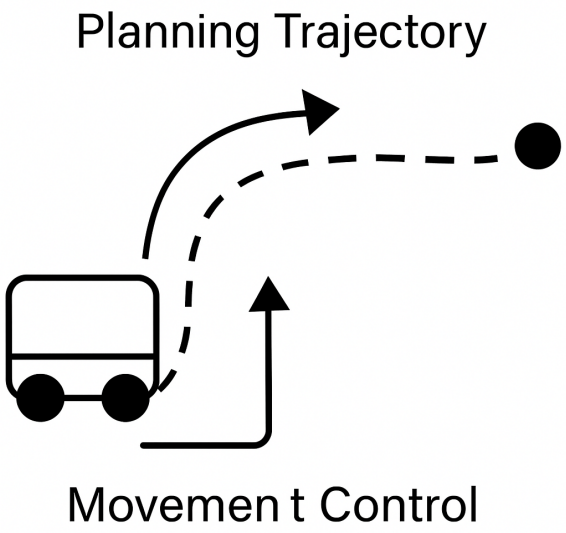


Figura 2.3 Planificación de Trayectoria y Control.

2.1.4 Control de movimiento

Los algoritmos de control traducen las decisiones de planificación en comandos específicos para los actuadores del robot. En robots con tracción diferencial, como el Create 3, el control se basa en la asignación independiente de velocidades a las ruedas izquierda y derecha.

El control diferencial permite generar tanto movimientos rectilíneos como rotaciones puras mediante la combinación apropiada de velocidades. Cuando ambas ruedas giran a la misma velocidad, el robot avanza en línea recta. Las diferencias en velocidad generan trayectorias curvas, mientras que velocidades de igual magnitud pero signo opuesto producen rotaciones sobre el centro geométrico del robot.

El control proporcional del ángulo de dirección implementa una estrategia de realimentación que ajusta continuamente la orientación del robot en función del error angular respecto al rumbo deseado. La ganancia proporcional determina la agresividad de las correcciones: valores altos proporcionan respuesta rápida pero pueden generar inestabilidad, mientras que valores bajos resultan en comportamiento suave pero respuesta lenta.

Los mecanismos de corrección post-evasiva, desarrollados específicamente para este proyecto, permiten al robot reorientarse automáticamente hacia el objetivo después de completar maniobras de esquiva. Esta funcionalidad resulta esencial para mantener el progreso hacia la meta en entornos con obstáculos múltiples.

2.2 Plataformas de hardware utilizadas

El desarrollo del sistema experimental se ha llevado a cabo sobre una arquitectura hardware que integra múltiples componentes especializados, cada uno desempeñando funciones específicas en los subsistemas de localización, percepción o control del robot.

2.2.1 Robot móvil: iRobot Create 3

El iRobot Create 3 constituye la plataforma base del proyecto, representando un sistema robótico educativo desarrollado por iRobot que combina la robustez mecánica de la serie Roomba con capacidades de programación avanzadas específicamente diseñadas para la investigación en robótica móvil.

Figura 2.4 Vista inferior Create 3. En esta imagen se observan claramente algunas de las partes esenciales para la odometría: las dos ruedas motrices con encoders integrados, así como sensores de corriente (current sensors) conectados a ellas.

La arquitectura mecánica del Create 3 se basa en un sistema de tracción diferencial que proporciona alta maniobrabilidad en espacios reducidos. Las dos ruedas motrices, equipadas con encoders de alta resolución, permiten tanto el avance rectilíneo como la rotación sobre el eje vertical del robot. Esta configuración resulta especialmente adecuada para navegación en interiores, donde se requiere precisión en los movimientos y capacidad de giro en espacios confinados.

Figura 2.5 Vista general Create 3 y base de carga. Se representa la parte frontal de la roomba, donde se encuentran los 7 sensores infrarrojos y los bumpers.

El sistema sensorial integrado incluye siete sensores infrarrojos de proximidad dispuestos en configuración semicircular en la parte frontal del robot. Esta disposición proporciona cobertura angular amplia para la detección temprana de obstáculos, permitiendo al sistema implementar estrategias de evasión antes de que se produzca el contacto físico. Los sensores de colisión (bumpers) actúan como sistema de seguridad secundario, detectando contactos mecánicos cuando las medidas de proximidad resultan insuficientes.

Figura 2.6 Vista interna Create 3.

La instrumentación interna incluye giroscopio y acelerómetro de tres ejes, aunque estos sensores no fueron utilizados directamente en el presente proyecto. La odometría interna se calcula mediante los encoders de rueda, que proporcionan mediciones de alta frecuencia sobre el desplazamiento angular de cada rueda motriz. Estos datos se procesan mediante algoritmos cinemáticos integrados para estimar la posición y orientación del robot en tiempo real.

La conectividad del sistema se realiza principalmente a través de Bluetooth Low Energy (BLE), que ofrece un balance adecuado entre ancho de banda y consumo energético para aplicaciones de control en tiempo real. La interfaz Wi-Fi, aunque disponible, no fue utilizada en este proyecto para simplificar la arquitectura de comunicaciones y evitar interferencias potenciales con otros sistemas de red.

La programación del Create 3 se realiza a través del SDK oficial irobot_edu_sdk, que proporciona una interfaz Python de alto nivel para el control del robot. Este SDK abstrae los detalles de bajo nivel de la comunicación y ofrece funciones directas para el control de movimiento, lectura de sensores y gestión del estado del sistema.

2.2.2 Sistema de localización absoluta: Marvelmind

La necesidad de disponer de referencias de posición independientes de la odometría interna motivó la selección del sistema Marvelmind como solución de localización absoluta en interiores. Este sistema comercial utiliza tecnología ultrasónica para proporcionar coordenadas de alta precisión en entornos donde el GPS convencional no es operativo.

2.2.2.1 Indoor Positioning Systems (IPS)

Los sistemas de posicionamiento en interiores han experimentado un desarrollo significativo en las últimas décadas, impulsados por las limitaciones del GPS en entornos cerrados y la creciente demanda de aplicaciones de localización en espacios interiores. Estos sistemas deben enfrentar desafíos únicos como la atenuación de señal, las reflexiones múltiples, y la interferencia electromagnética presente en entornos construidos.

Las tecnologías disponibles para IPS presentan características diferenciadas en términos de precisión, coste, complejidad de implementación y robustez ante condiciones ambientales variables. Los sistemas basados en identificación por radiofrecuencia (RFID) ofrecen soluciones de bajo coste pero con precisión limitada, adecuadas para aplicaciones de seguimiento aproximado. Las técnicas basadas en triangulación por intensidad de señal recibida (RSSI) utilizando Wi-Fi o Bluetooth Low Energy proporcionan mayor precisión a coste moderado, aunque su rendimiento puede verse afectado por obstáculos metálicos y variaciones en la propagación de radiofrecuencia.

Tabla 2.1 Comparativa de tecnologías de localización.

Tecnología	Principio de funcionamiento	Precisión típica	Coste relativo	Complejidad	Condiciones ideales
RFID	Identificación por radiofrecuencia	Baja (1–5m)	Bajo	Baja	Entorno libre de obstáculos metálicos
Wi-Fi / Bluetooth BLE	Triangulación por RSSI	Media (2–5m)	Bajo	Media	Cobertura homogénea
Visión artificial	Procesamiento de imágenes	Alta (cm)	Alto	Alta	Buena iluminación, sin obstrucciones
Ultrasonidos (US)	Tiempo de vuelo y triangulación	Muy alta (1–2cm)	Medio	Media	Entorno controlado, sin rebotes acústicos

Los sistemas de visión artificial pueden alcanzar precisiones extremas mediante el procesamiento de marcadores visuales o características naturales del entorno, pero requieren condiciones de iluminación controladas y línea de visión directa entre cámaras y objetivos. Su complejidad computacional es considerablemente mayor, lo que puede limitar su aplicación en sistemas con recursos limitados.

Los sistemas ultrasónicos, categoría a la que pertenece Marvelmind, representan un compromiso óptimo entre precisión, coste y robustez para aplicaciones de robótica móvil en interiores. La alta velocidad de propagación del sonido en aire permite mediciones de tiempo de vuelo con resolución temporal elevada, traduciéndose en precisiones del orden de centímetros bajo condiciones apropiadas.

2.2.2.2 Fundamentos físicos de la localización por ultrasonidos

Los sistemas de localización ultrasónica operan mediante la medición precisa del tiempo transcurrido entre la emisión de un pulso acústico y la recepción de su reflexión o la recepción directa en un receptor remoto. Este principio, conocido como tiempo de vuelo (Time of Flight), permite calcular distancias con alta precisión utilizando la ecuación fundamental:

$$d = \frac{t_{\text{vuelo}} \cdot v_{\text{sonido}}}{2} \tag{2.1}$$

donde $v_{\text{sonido}} \approx 343\,\text{m/s}$ representa la velocidad de propagación del sonido en aire a temperatura ambiente (20°C), y t_{vuelo} corresponde al intervalo temporal medido entre emisión y recepción del pulso ultrasónico.

La triangulación de múltiples mediciones de distancia desde posiciones conocidas (anchors o balizas fijas) permite determinar la posición bidimensional o tridimensional del objeto móvil (hedgehog o tag). Para localización bidimensional se requieren al menos tres anchors con posiciones conocidas, mientras que la localización tridimensional necesita un mínimo de cuatro referencias.

El sistema Marvelmind implementa una arquitectura distribuida donde múltiples anchors se sincronizan temporalmente para realizar mediciones simultáneas hacia el tag móvil. Esta sincronización resulta crítica para la precisión del sistema, ya que errores de sincronización de microsegundos se traducen directamente en errores de posición de centímetros.

2.2.2.3 Factores que afectan la propagación ultrasónica

La precisión de los sistemas ultrasónicos está influenciada por diversos factores ambientales y geométricos que deben considerarse durante el diseño e implementación del sistema de localización.

Los rebotes acústicos constituyen una fuente significativa de error, especialmente en entornos con superficies duras y paralelas que pueden generar ecos múltiples. Estos ecos retardados pueden ser interpretados erróneamente como señales directas, introduciendo errores sistemáticos en las mediciones de distancia. La geometría del entorno de operación debe diseñarse minimizando estas reflexiones mediante el uso de materiales absorbentes acústicos o mediante la disposición estratégica de las superficies reflectantes.

Los obstáculos físicos entre anchors y tags pueden bloquear completamente la propagación ultrasónica, ya que las ondas acústicas de alta frecuencia presentan capacidad limitada de difracción alrededor de objetos sólidos. Esta característica requiere mantener líneas de visión acústica despejadas entre todos los elementos del sistema, lo que puede limitar la flexibilidad de implementación en entornos complejos.

Las variaciones ambientales, particularmente temperatura y humedad, afectan la velocidad de propagación del sonido y, consecuentemente, la precisión de las mediciones de distancia. La velocidad del sonido varía aproximadamente 0.6 m/s por cada grado Celsius de variación térmica, lo que representa un error potencial del 0.17 por ciento por grado en las mediciones de distancia. Los sistemas avanzados incorporan compensación térmica automática mediante sensores de temperatura distribuidos.

La disposición geométrica de los anchors influye significativamente en la precisión de localización, siendo preferibles configuraciones que maximicen los ángulos entre las líneas de visión desde el tag hacia diferentes anchors. Configuraciones colineales o con ángulos agudos resultan en mayor amplificación de errores de medición y deben evitarse.

2.2.3 Cámara de visión: TP-Link Tapo C200

La exploración de capacidades de visión artificial motivó la incorporación experimental de una cámara IP comercial de bajo coste. La TP-Link Tapo C200 representa una solución orientada originalmente a aplicaciones de videovigilancia, pero cuyas características técnicas la hacen adecuada para experimentación en visión por computador.

Figura 2.7 Cámara TP-Link Tapo C200.

La resolución máxima de 1080p (Full HD) proporciona calidad de imagen suficiente para algoritmos de detección de objetos, mientras que la transmisión mediante protocolo RTSP (Real Time Streaming Protocol) permite la integración directa con bibliotecas de procesamiento de imágenes como OpenCV. La conectividad de red local facilita el acceso programático a la secuencia de vídeo sin requerir hardware especializado adicional.

La integración experimental con YOLOv8n (You Only Look Once version 8 nano) demostró la viabilidad técnica de incorporar detección de objetos en tiempo real al sistema de navegación. YOLOv8n representa una versión optimizada de la arquitectura YOLO específicamente diseñada para aplicaciones con recursos computacionales limitados, manteniendo capacidades de detección competitivas con requisitos de procesamiento reducidos.

Aunque las limitaciones de rendimiento del hardware disponible impidieron la integración completa de esta funcionalidad en el sistema final de navegación, la implementación experimental establece la arquitectura básica para futuras expansiones en percepción visual. La capacidad demostrada de detectar y clasificar objetos en tiempo real abre posibilidades para navegación semántica y comportamientos adaptativos más sofisticados.

2.2.4 Ordenador host

La coordinación de todos los subsistemas del proyecto requirió un ordenador host que actuara como nodo central de procesamiento y control. Un portátil Asus TUF Gaming F17 FX706LI_FX706LI proporcionó la plataforma computacional necesaria, ejecutando Windows 10 como sistema operativo base.

Figura 2.8 Asus TUF Gaming F17 FX706LI_FX706LI.

La arquitectura de comunicaciones implementada integra múltiples interfaces: conectividad Bluetooth para el control del Create 3, comunicación serie USB para el enlace con el módem Marvelmind a través del puerto COM3, y conectividad de red local para el acceso RTSP a la cámara IP. Esta configuración multimodal requirió la implementación de arquitecturas de software asíncronas capaces de gestionar múltiples flujos de datos simultáneos sin interferencias.

El entorno de desarrollo se basó en Python 3.10, seleccionado por su amplio ecosistema de bibliotecas científicas y su capacidad para integrar de manera eficiente comunicaciones, procesamiento numérico y visualización en tiempo real. Las bibliotecas principales incluyen irobot_edu_sdk para el control del robot, marvelmind.py para la comunicación con el sistema de localización, opencv-python para procesamiento de imágenes, ultralytics para la implementación de YOLO, y matplotlib para visualización de datos y trayectorias.

La gestión asíncrona de procesos mediante asyncio permitió la ejecución concurrente de múltiples tareas: control del robot, adquisición de datos de localización, procesamiento de sensores, y actualización de visualizaciones. Esta arquitectura resulta esencial para mantener la respuesta en tiempo real del sistema de navegación, evitando bloqueos que podrían comprometer la seguridad o eficiencia del robot.

2.3 Configuración del entorno experimental

El diseño experimental requirió la configuración de un entorno controlado que permitiera la validación sistemática de los algoritmos de localización y navegación desarrollados. Se seleccionó un espacio interior de aproximadamente 2 × 3 metros, delimitado por paredes y con superficie de baldosas cerámicas. Esta superficie proporcionó condiciones favorables de fricción para la odometría del robot, minimizando el deslizamiento de las ruedas y reduciendo una fuente significativa de error en las estimaciones de posición.

La temperatura ambiente se mantuvo relativamente estable entre 24-26°C durante la realización de los experimentos. Este control resulta importante para el sistema Marvelmind, ya que las variaciones térmicas afectan directamente la velocidad de propagación del sonido y, consecuentemente, la precisión de las mediciones ultrasónicas. La estabilidad térmica contribuyó a minimizar errores sistemáticos en la localización absoluta.

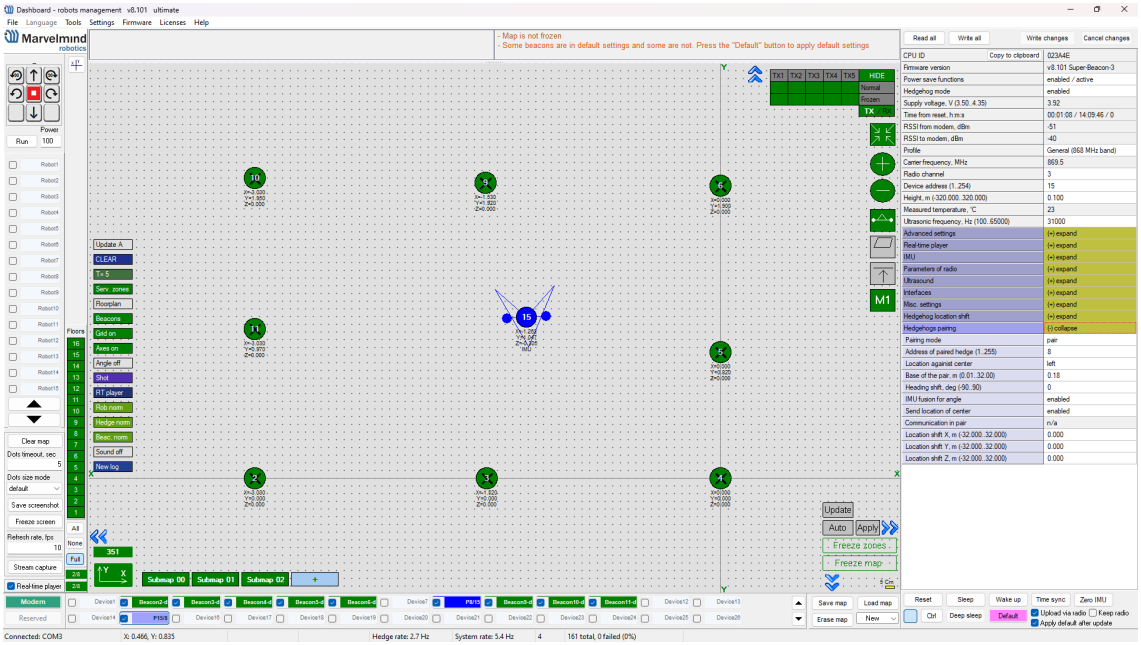


Figura 2.9 Configuración Dashboard.

La implementación del sistema de localización ultrasónica requirió una fase previa de instalación y calibración que determinó en gran medida la precisión final del sistema. Se desplegaron ocho balizas fijas (anchors) en las esquinas y puntos estratégicos del perímetro del área experimental, configurándolas para operación en modo bidimensional.

Cada anchor fue asignado con un identificador único mediante el software Marvelmind Dashboard, configurándose como "stationary beacon" para indicar su posición fija. La calibración geométrica se realizó mediante medición manual de las distancias entre anchors utilizando herramientas de medición convencionales, introduciendo estas referencias en el sistema para optimizar los algoritmos de triangulación.

Dos balizas móviles (hedgehogs) se instalaron sobre la plataforma Create 3 en configuración de "Hedgehogs pairing". Esta disposición permite al sistema Marvelmind no solo estimar la posición del robot, sino también su orientación mediante el cálculo del ángulo entre los dos tags móviles. Esta funcionalidad de estimación de orientación resultó crucial para los algoritmos de navegación autónoma, proporcionando una referencia angular independiente de la odometría interna del robot.

La configuración del sistema se optimizó para maximizar la frecuencia de actualización manteniendo la precisión. La comunicación con el ordenador host se estableció mediante puerto serie USB virtual, utilizando el protocolo propietario de Marvelmind para la transmisión de coordenadas y metadatos de calidad de señal.

3 Descripción funcional del sistema

3.1 Primera Parte (Sistemas de control manual)

Esta sección está dedicada a describir las partes esenciales de los distintos códigos que se han diseñado en este proyecto. Como punto de partida, se comenzará abordando el código orientado al control manual y obtención de odometría interna del robot (roomba_pos.py).

3.1.1 Control manual + obtención de odometía (roomba_pos.py)

En primer lugar, se creyó conveniente que la forma de conexión más óptima sería mediante el lenguaje de programación Python a través de Bluetooth. La página web oficial de iRobot ofrece un sistema de sincronización y conexión directo sin necesidad de programas adicionales. Sin embargo, esta opción fue rechazada debido a las limitaciones funcionales avanzadas que presentaba esta posibilidad.

En segundo lugar, la roomba puede ser controlada por cable directamente, pudiendo así prescindir del Bluetooth. Pese a ello, se escogió la alternativa de conexión Bluetooth, la cual es más elegante a efectos estéticos y más funcional, ya que el cable limita la autonomía del robot. En ensayos posteriores, se realizan recorridos autónomos para los cuales el cable hubiese supuesto un impedimento.

Es muy importante recalcar que durante la realización del proyecto, se decidió inicialmente que el control manual del Create 3 se realizase mediante las entradas por teclado del ordenador host.

Sin embargo, el control era bastante limitado debido a la poca precisión y sensibilidad que estas ofrecen, la roomba presentaba un comportamiento "errático". Por ello, se reestructuró la idea inicial, se añadió al equipo un **mando PS4**.

Este nuevo enfoque permitió una notable mejoría en la sensibilidad y controlabilidad. Además, esta iniciativa aporta una estética mucho más dinámica y divertida que la anterior, más atractiva e interesante para el espectador.

Una vez definido este notable cambio, se describe el funcionamiento del código:

El programa inicia estableciendo la conexión con el robot a través del SDK oficial de iRobot, y verifica la disponibilidad del mando mediante la librería pygame. En caso de no detectarse ningún dispositivo, el sistema se cierra automáticamente para evitar errores posteriores. Una vez que el mando es reconocido, se activa una rutina de inicialización en la que el robot emite una secuencia acústica que confirma que la conexión se ha establecido correctamente.

Durante la ejecución, se mantiene un bucle principal en el que se leen continuamente los valores de los ejes y botones del mando. Estos valores se convierten en velocidades diferenciales de las ruedas, aplicando modos adicionales como turbo, marcha atrás o drift, según el botón pulsado. Además, se implementaron mecanismos de seguridad: un freno de emergencia que detiene al robot de forma inmediata y una parada total, que detiene el programa por completo.

El comportamiento del sistema puede resumirse mediante el siguiente pseudocódigo:

```
inicializar_bluetooth()
detectar_mando_PS4()

si no hay_mando:
    cerrar_programa()

emitir_tonos_iniciales()

mientras programa_activo:
    leer_joystick()
    calcular_velocidades_ruedas()
    si freno_emergencia:
        detener_robot()
    si parada_total:
        cerrar_programa()
    enviar_velocidades_al_robot()
```

Código 3.1 Control principal de la roomba con el mando.

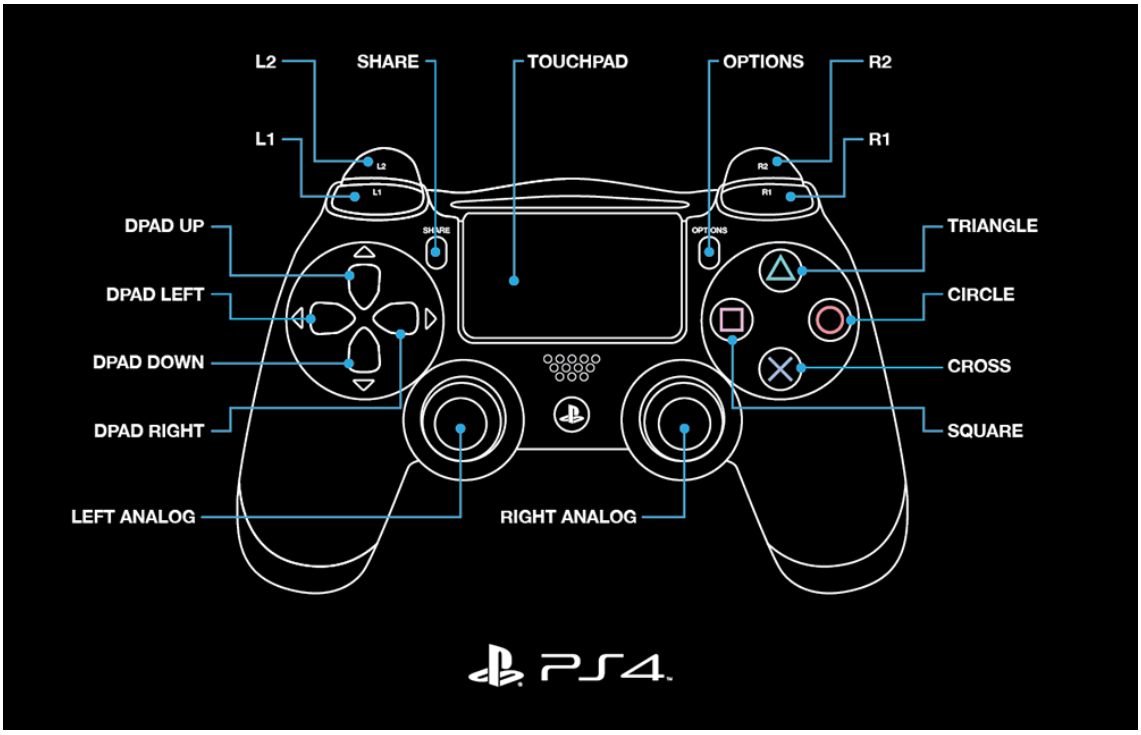


Figura 3.1 Estructura mando PS4.

En la imagen anterior, se muestra de manera visual la estructura del mando PS4. Una vez definidas y comentadas las conversiones de entrada del mando de velocidades, se le asocia a cada control una funcionalidad concreta. A continuación, se muestran en la siguiente tabla:

Tabla 3.1 Controles y funciones del Create 3.

Botón	Función
Joystick L	Dirección (volante)
R2 (gatillo)	Acelerador
L2 (gatillo)	Freno
Círculo	Turbo
X	Freno de emergencia
Cuadrado	Modo marcha atrás
Triángulo	Claxon
R1	Giro 180°
L1	Drift
OPTIONS	Parada de emergencia

Es importante tener en cuenta algunas especificaciones al respecto.

- Freno de emergencia: una vez deje de estar pulsado, el movimiento podrá reanudarse de manera completamente habitual.
- Parada de emergencia: al pulsarlo, se detiene completamente el robot y se cierra el programa. No podrá volver a reanudarse el movimiento, el sistema se debe reiniciarse.
- Modo marcha atrás: se debe volver a pulsar el botón cuadrado para revertir la operación.

La plataforma iRobot Create 3 proporciona de manera nativa estimaciones de posición y orientación a través de sus encoders de rueda. Estos datos, que constituyen la odometría interna, se comunican mediante el SDK oficial del robot en tiempo real. El acceso a esta información resulta esencial para cualquier sistema de navegación, ya que permite reconstruir la trayectoria recorrida sin necesidad de sensores externos.

En este proyecto, se desarrolló un módulo dedicado a la adquisición periódica de dichos datos. La rutina establece en primer lugar la conexión con el robot, confirmando que el canal de comunicación está disponible y operativo. A continuación, se inicia un bucle de ejecución que consulta la posición actual estimada por la odometría y la almacena en un archivo compartido en formato JSON. Este archivo actúa como repositorio común, al que pueden acceder otros procesos del sistema, como la estimación de error frente a Marvelmind o la fusión de datos mediante el filtro de Kalman.

El módulo incluye además mecanismos de control para evitar bloqueos en caso de pérdida de conexión. Si durante la ejecución no se reciben datos en un tiempo determinado, el sistema genera un mensaje de alerta y se detiene de forma segura. Esta medida garantiza la fiabilidad del registro y evita la propagación de errores en las etapas posteriores de navegación.

La lógica de funcionamiento puede representarse mediante el siguiente pseudocódigo y diagrama de flujo:

```
inicializar_conexion_create3()

si no hay_conexion:
    cerrar_programa()

mientras programa_activo:
    leer_datos_odometria()
    guardar_datos_JSON()
    si error_comunicacion:
        mostrar_alerta()
        cerrar_programa()
```

Código 3.2 Lectura y guardado periódico de la posición.

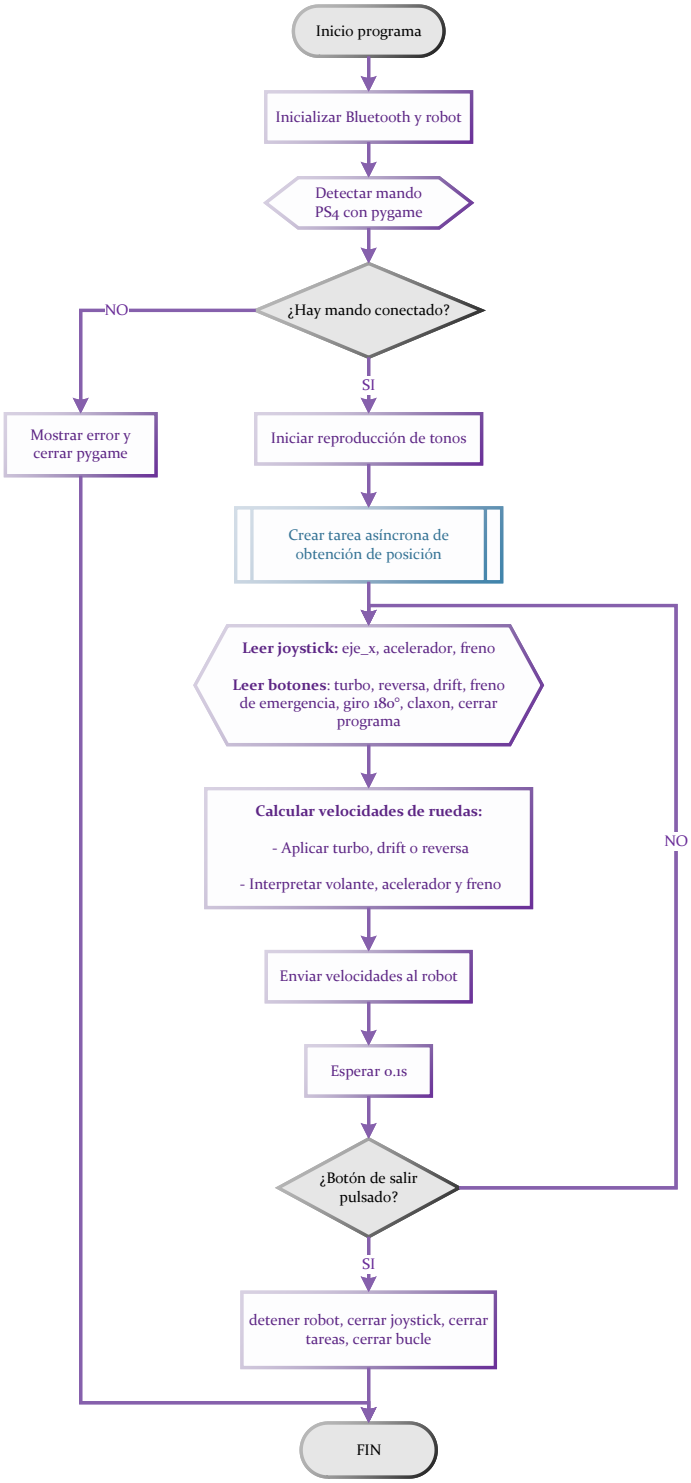


Figura 3.2 Flujo detallado del control manual con mando PS4 (roomba_pos.py).

3.1.2 Adquisición de datos de Marvelmind (marvelmind_process.py)

El sistema Marvelmind se emplea como referencia de localización absoluta en interiores. Su función dentro del proyecto es proporcionar coordenadas del robot en un marco fijo del entorno, complementando la odometría de la iRobot Create 3 y sirviendo como entrada para la posterior fusión con el filtro de Kalman. La comunicación se establece desde el ordenador host hacia el módem Marvelmind, y se accede al flujo de datos a través de un puerto serie configurado con los parámetros recomendados por el fabricante. Antes de iniciar la captura, se verifica la disponibilidad del dispositivo y se comprueba que el puerto corresponde efectivamente al módem detectado en el sistema.

Una vez abierta la comunicación, el módulo mantiene un bucle de adquisición continuo que lee los paquetes publicados por el sistema. Cada paquete contiene la posición estimada del "tag" asociado al robot y metadatos de sincronización que permiten evaluar la actualidad y la integridad de la lectura. Tras la lectura, se realiza un procesamiento estructurado para extraer las coordenadas y la marca temporal, y se aplica una validación básica de integridad que descarta tramas incompletas o con campos fuera de rango. En caso de no recibir datos durante un intervalo prefijado, el módulo emite una alerta y procede a cerrar de forma segura la conexión para evitar que el resto del sistema trabaje con información obsoleta.

Con el fin de asegurar la coherencia con el resto de componentes, las coordenadas se normalizan antes de su almacenamiento. Esta normalización incluye la conversión de unidades al formato que utiliza el proyecto, la alineación de ejes con el marco global del entorno, y la aplicación de un desplazamiento de origen si fuese necesario para que la referencia externa coincida con la empleada por la odometría. De esta manera, tanto la posición absoluta de Marvelmind como la estimación relativa de la Create 3 quedan en el mismo sistema de referencia, lo que facilita su comparación directa y la fusión posterior.

Los datos validados se escriben en un archivo compartido en formato JSON, común a los distintos procesos del sistema. Este archivo expone de forma accesible la última posición conocida, su marca temporal y un indicador de validez. Otros módulos, como el cálculo del error o el filtro de Kalman, consumen este mismo repositorio, lo que simplifica la sincronización entre procesos y evita acoplamientos innecesarios. Si durante la ejecución se detecta un cambio de puerto o una desconexión física del módem, el módulo intenta una reconexión controlada; si no es posible, finaliza de manera ordenada para no propagar estados inconsistentes.

La lógica principal puede sintetizarse en el siguiente pseudocódigo:

```
abrir_puerto_serie_marvelmind()

si no puerto_disponible:
    registrar_alerta()
    cerrar_programa()

mientras programa_activo:
    trama = leer_trama_serie()
    si trama_invalida:
        continuar

    datos = procesar_trama(trama)
    si datos_fuera_de_rango:
        continuar

    coords = normalizar_coordenadas(datos)
    escribir_JSON(coords, timestamp=ahora(), valido=true)

    si timeout_sin_datos:
        registrar_alerta()
        cerrar_conexion()
        cerrar_programa()
```

Código 3.3 Funcionamiento de adquisición de datos.

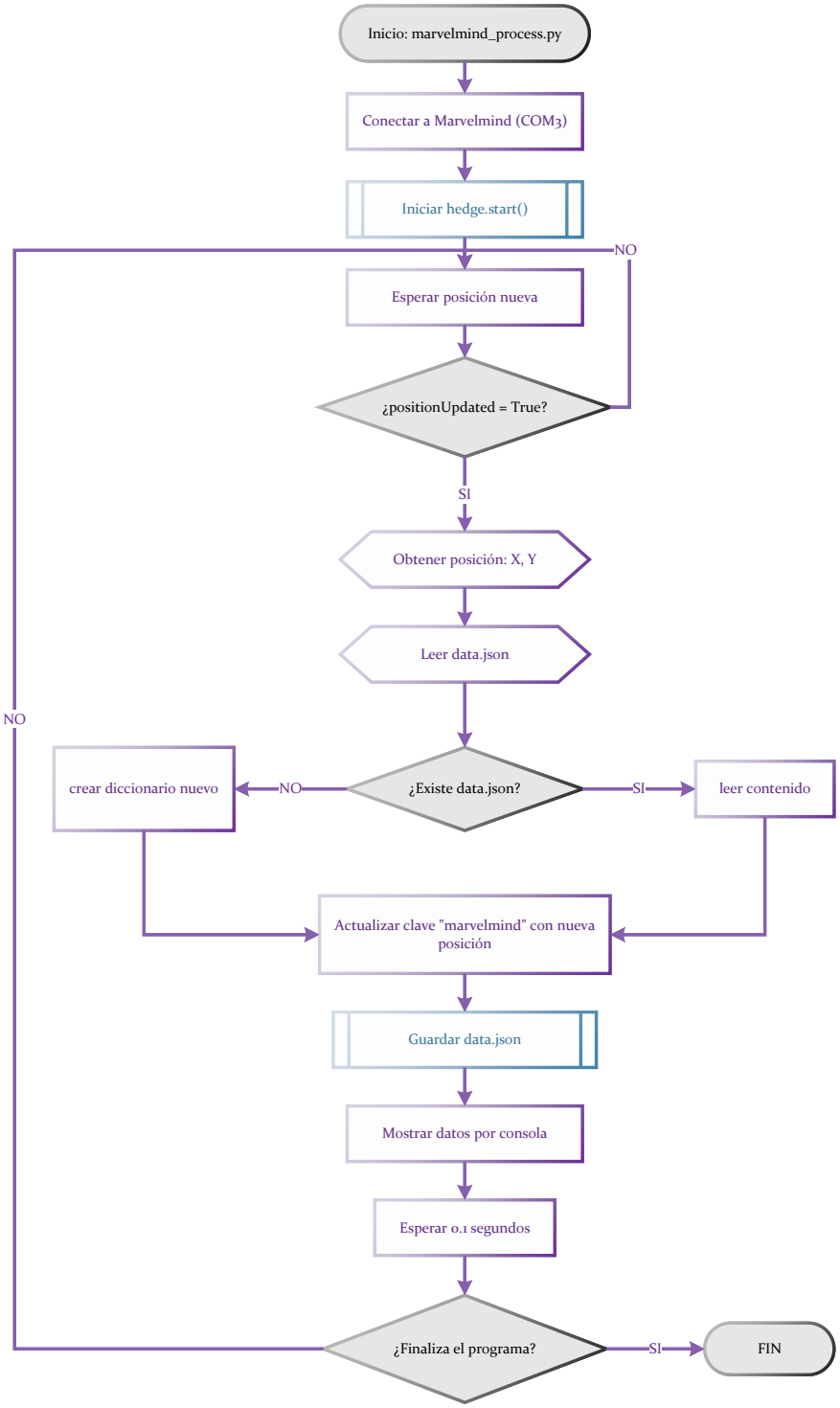


Figura 3.3 Flujo detallado de adquisición de datos de Marvelmind (marvelmind_process.py).

3.1.3 Cálculo de error (coordinador.py)

Para evaluar la fiabilidad de la estimación de posición del robot, el sistema implementa un módulo de comparación entre la odometría (posición relativa calculada por el propio robot a partir del movimiento de las ruedas) y la posición absoluta proporcionada por Marvelmind. El objetivo de este bloque es cuantificar la discrepancia entre ambas fuentes de información y disponer de un indicador de error que pueda utilizarse tanto en la validación experimental como en la integración con el filtro de Kalman.

El diagrama de flujo correspondiente refleja cómo, en cada iteración, se leen las posiciones procedentes de las dos fuentes. La odometría ofrece la trayectoria relativa del robot, pero está sujeta a acumulación de errores y deriva con el tiempo. Marvelmind, por su parte, proporciona una posición absoluta en el entorno, aunque con cierto nivel de ruido en sus mediciones.

Una vez obtenidas las posiciones (x_roomba, y_roomba) y (x_marvelmind, y_marvelmind), se calcula la diferencia entre ellas, que se interpreta como el error de odometría respecto a la referencia externa. Este error se expresa normalmente como la distancia euclidiana entre ambos puntos:

$$e = \sqrt{(x_{\text{Roomba}} - x_{\text{Marvelmind}})^2 + (y_{\text{Roomba}} - y_{\text{Marvelmind}})^2} \tag{3.1}$$

Además, se registran los valores de error a lo largo del tiempo, lo que permite analizar la evolución de la deriva de la odometría y la estabilidad de Marvelmind. Estos resultados son útiles tanto para la justificación de la necesidad de un algoritmo de fusión (Kalman) como para valorar la calidad del sistema en distintos entornos de prueba.

El bloque incluye mecanismos de validación: si en un instante no se dispone de datos válidos de Marvelmind, el error no se calcula y se marca la muestra como inválida. De este modo se evita introducir falsos errores derivados de lecturas incompletas o fallidas.

En conjunto, este módulo constituye una herramienta de diagnóstico esencial, ya que hace visible la diferencia entre un sistema de localización relativa y uno absoluto, y aporta una métrica objetiva para evaluar el rendimiento del robot en condiciones reales.

La lógica principal del coordinador puede sintetizarse en el siguiente pseudocódigo:

```
leer_posicion_odometria()
leer_posicion_marvelmind()

si posicion_marvelmind_valida:
    error_x = x_marvelmind - x_odometria
    error_y = y_marvelmind - y_odometria
    error = sqrt(error_x^2 + error_y^2)

    registrar_error(error)
sino:
    marcar_muestra_invalida()
```

Código 3.4 Lógica del coordinador.

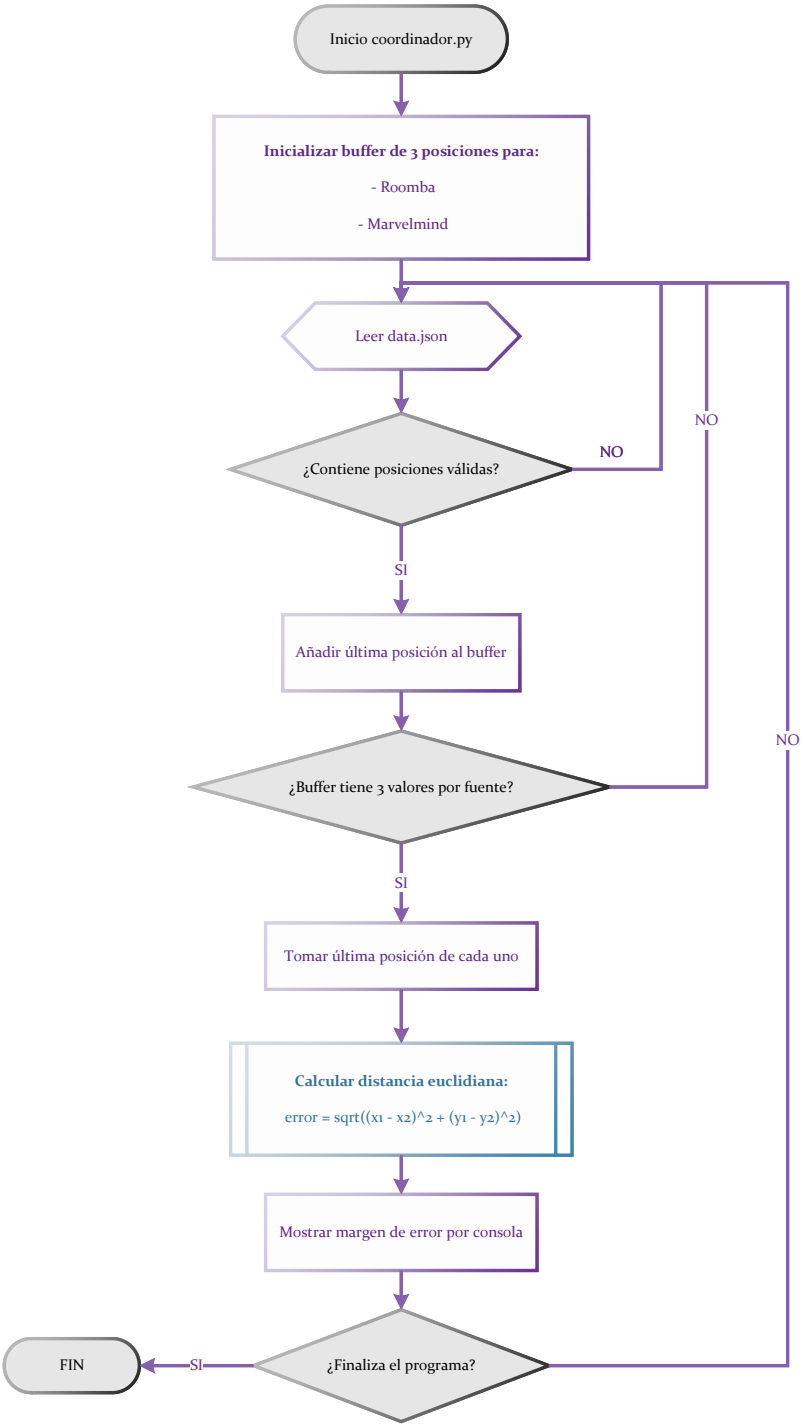


Figura 3.4 Flujo detallado del cálculo de error (coordinador.py).

3.1.4 Núcleo de control del sistema (launcher.py)

El Launcher constituye el punto de entrada del sistema desarrollado y su finalidad principal es coordinar la ejecución de los distintos módulos que conforman el proyecto. Dado que cada bloque funcional se implementa como un proceso independiente, resulta imprescindible disponer de un mecanismo que permita iniciarlos, monitorizarlos y, en caso necesario, detenerlos de manera controlada.

El diseño del Launcher responde a la necesidad de garantizar la consistencia del sistema completo. De esta forma, se evitan ejecuciones parciales que podrían generar errores (por ejemplo, intentar calcular error sin tener datos de Marvelmind, o inicializar navegación sin disponer de odometría). Además, el Launcher centraliza la gestión de logs, facilitando la depuración y la trazabilidad de los ensayos.

En su funcionamiento, el Launcher inicializa los procesos de manera ordenada, comprobando que cada uno ha arrancado correctamente antes de lanzar el siguiente. En caso de error en la inicialización, se notifica al usuario y se detiene el resto de la secuencia.

Durante la ejecución, el Launcher mantiene un bucle de supervisión que monitoriza la actividad de los procesos hijos. Si alguno finaliza de forma inesperada, el sistema emite una alerta y ofrece al usuario la posibilidad de reiniciarlo o cerrar todos los módulos de forma segura. Esta monitorización es especialmente relevante en un proyecto experimental, donde pueden producirse desconexiones o fallos de comunicación que de otro modo dejarían al sistema en un estado inconsistente.

Cuando el usuario decide finalizar la sesión, el Launcher envía una orden de terminación a todos los procesos activos, asegurando que los recursos compartidos (archivos JSON, puertos de comunicación, etc.) se liberen correctamente. De esta forma, se evita que queden procesos en segundo plano interfiriendo con ejecuciones posteriores.

La lógica general del Launcher puede resumirse mediante el siguiente pseudocódigo:

```
procesos = inicializar_procesos

si algun_proceso_falla:
    mostrar_alerta()
    detener_todos_procesos()
    cerrar_launcher()

mientras launcher_activo:
    monitorizar_procesos(procesos)
    si algun_proceso_termina_inesperadamente:
        notificar_usuario()
        opcion = leer_respuesta()
        si opcion == reiniciar:
            reiniciar_proceso()
        si opcion == cerrar:
            detener_todos_procesos()
            cerrar_launcher()

al terminar:
    detener_todos_procesos()
    liberar_recursos()
    cerrar_launcher()
```

Código 3.5 Ejecución del Launcher.

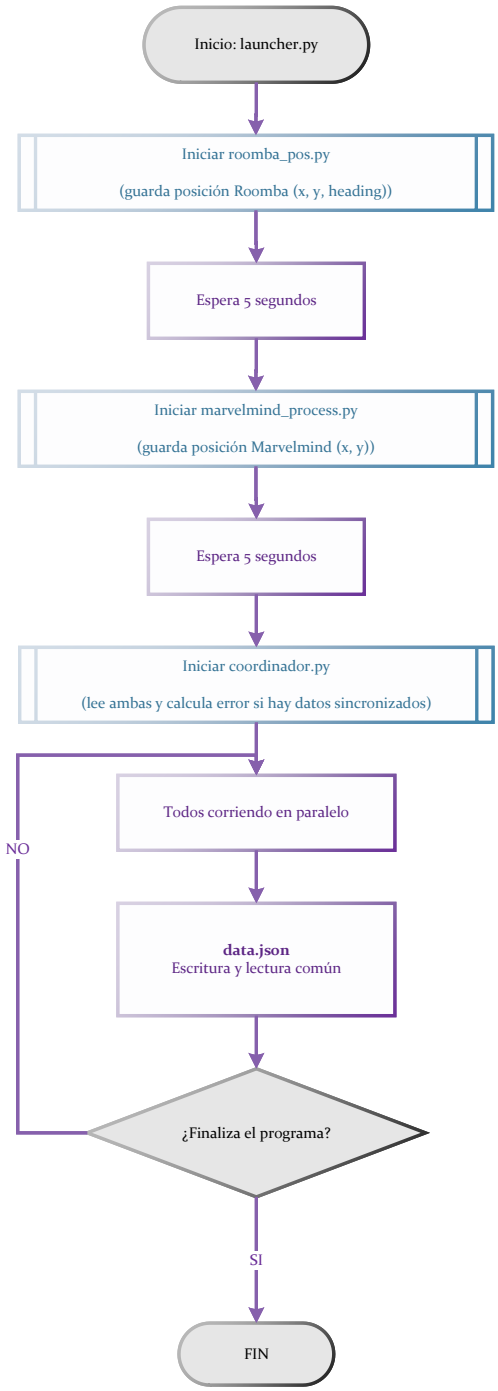


Figura 3.5 Flujo del núcleo de control del sistema (launcher.py).

3.2 Segunda Parte (Estimación del error de posición Marvelmind)

Si bien el sistema Marvelmind se considera más preciso que la odometría interna del Create 3, resulta necesario cuantificar su exactitud real dentro del entorno de pruebas utilizado. Para ello, se diseñó un procedimiento experimental orientado a medir la discrepancia entre las coordenadas proporcionadas por Marvelmind y la posición real en el espacio, medida de forma manual.

El entorno de trabajo se delimitó mediante las balizas fijas de Marvelmind, abarcando un rectángulo de 2 x 3 metros. Sobre esa superficie se seleccionaron catorce puntos de referencia distribuidos homogéneamente, que se midieron con un metro convencional y se marcaron físicamente en el suelo. Dichas mediciones manuales se consideran prácticamente exactas y, por tanto, se emplearon como referencia real.

3.2.1 Medición de error punto a punto (medicion_error_mm2.py)

El procedimiento de medida fue el siguiente: el usuario colocaba la baliza móvil (hedgehog) en uno de los puntos marcados en el suelo, introducía por pantalla la posición real de dicho punto y, a continuación, el programa adquiría automáticamente diez lecturas de Marvelmind en esa posición. De estas muestras se calculaba un promedio, con el fin de reducir el efecto del ruido y de posibles lecturas erráticas.

Finalmente, se determinaba el error en cada punto aplicando la fórmula de la distancia euclidiana entre la posición real introducida por el usuario (x_real,y_real) y la posición media obtenida de Marvelmind (x_mm,y_mm):

$$E = \sqrt{(x_{\text{real}} - x_{\text{mm}})^2 + (y_{\text{real}} - y_{\text{mm}})^2} \tag{3.2}$$

Este proceso se repetía de manera sistemática en cada uno de los puntos de referencia. Los resultados se almacenaban en un archivo CSV, lo que permitía disponer de un registro completo de la exactitud de Marvelmind en distintas partes del entorno y analizar posibles particularidades, como zonas con mayor dispersión o áreas más fiables.

Con ello se obtiene una visión objetiva de la precisión de Marvelmind, información esencial para valorar su uso como referencia principal y para justificar la posterior implementación del filtro de Kalman. Tras obtener el error, se guardan los datos de las distintas posiciones y error en CSV.

El programa está diseñado para realizar este proceso de manera repetitiva, es decir, tomando más puntos anclados para poder contar con múltiples referencias en todo el entorno y conocer las particularidades de este. Una vez finalizado el proceso de muestreo, el programa finaliza y genera diversos archivos, los cuales se tratarán más adelante.

La lógica general puede resumirse mediante el siguiente pseudocódigo:

```
para cada punto_referencia en lista_puntos:

    colocar_baliza_en(punto_referencia)
    posicion_real = leer_input_usuario()

    muestras = []
    repetir 10 veces:
        muestras.agregar(leer_posicion_marvelmind())

    promedio_mm = calcular_promedio(muestras)

    error = distancia_euclidiana(posicion_real, promedio_mm)

    guardar_en_CSV(posicion_real, promedio_mm, error)
```

Código 3.6 Lógica de la medición del error.

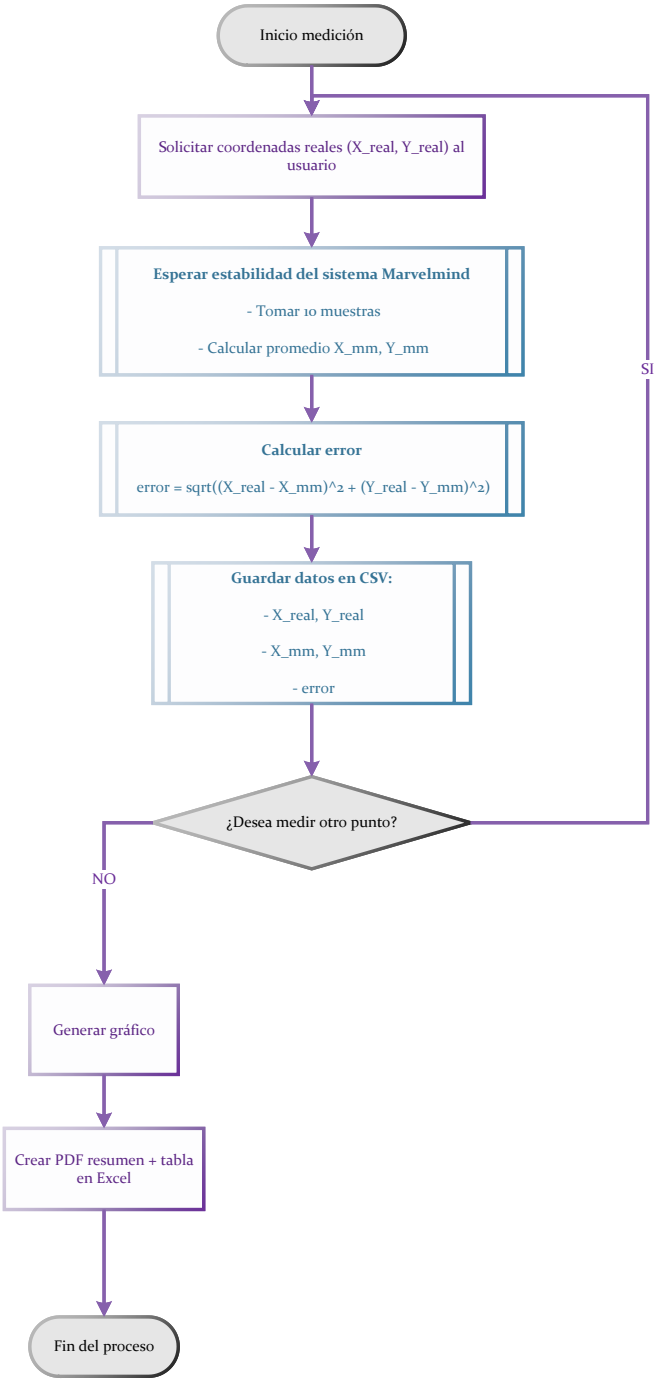


Figura 3.6 Flujo detallado de medición de error punto a punto (medicion_error_mm2.py).

3.3 Tercera Parte (Navegación autónoma)

En los sistemas de localización basados en sensores es habitual que las mediciones de posición presenten ruido y pequeñas oscilaciones, lo que repercute en la estabilidad del control. En este proyecto, aunque Marvelmind proporciona posiciones absolutas de buena calidad, las lecturas puntuales pueden mostrar ligeras variaciones que, si se utilizan directamente, podrían traducirse en maniobras erráticas del robot. Para mitigar este efecto se optó por implementar un filtro de Kalman bidimensional, concebido como un estimador que combina la predicción interna del sistema con las mediciones externas, ofreciendo una trayectoria más estable y confiable.

3.3.1 Principio del flitro de Kalman

El principio del filtro de Kalman se basa en un proceso recursivo que alterna entre predicción y corrección. Durante la fase de predicción se calcula la nueva posición del robot a partir de la posición anterior, siguiendo un modelo de movimiento simplificado que asume un desplazamiento progresivo y sin cambios bruscos de velocidad. Posteriormente, en la fase de actualización, la estimación se corrige mediante la lectura actual de Marvelmind, de manera que el algoritmo pondera la confianza relativa entre lo que predice el modelo y lo que indican los sensores. Esta combinación otorga al sistema la capacidad de reducir el impacto del ruido de las medidas sin perder sensibilidad frente a variaciones reales en la trayectoria.

3.3.2 Implementación práctica en el proyecto

La implementación práctica del filtro en el proyecto se realizó en Python, trabajando con entradas bidimensionales compuestas por las coordenadas x e y aportadas por Marvelmind. El intervalo de muestreo se fijó en 0,2 segundos, correspondiente a la frecuencia de actualización del sistema, y el modelo de movimiento se formuló bajo el supuesto de velocidad constante entre iteraciones, lo que resulta adecuado para el desplazamiento suave del robot Create 3. La ganancia del filtro de Kalman se ajustó de manera empírica tras diversas pruebas, con el propósito de equilibrar la suavidad de la trayectoria estimada y la capacidad de reacción del sistema cuando se producen cambios significativos de posición.

Una vez integrado en el bucle de navegación autónoma, el filtro pasó a ser el encargado de proporcionar la posición de referencia para el resto de módulos, entre ellos el cálculo de error, la detección de estancamiento o las maniobras de evasión. De esta forma, las decisiones de control dejaron de depender directamente de las medidas individuales de Marvelmind y se basaron en una estimación más coherente y estable, lo que incrementó notablemente la fiabilidad del sistema durante los ensayos experimentales.

La lógica general puede resumirse mediante el siguiente pseudocódigo:

```
inicializar_estado(x0, y0)
inicializar_covarianza()

mientras navegacion_activa:
    # Predicción
    x_pred = x_est
    y_pred = y_est
    P_pred = P + Q

    # Medida Marvelmind
    (x_med, y_med) = leer_marvelmind()

    # Actualización
    K = P_pred / (P_pred + R)
    x_est = x_pred + K * (x_med - x_pred)
    y_est = y_pred + K * (y_med - y_pred)
    P = (1 - K) * P_pred

    usar_posicion(x_est, y_est)
```

Código 3.7 Implementación del filtro de Kalman.

3.3.3 Navegación autónoma K6 (navegacion_K6.py)

El núcleo de este proyecto se materializa en el sistema de navegación autónoma desarrollado para el robot iRobot Create 3. Este sistema se diseñó siguiendo una arquitectura modular que integra de manera coherente los distintos subsistemas de percepción, control y localización. Gracias a esta integración, el robot es capaz de desplazarse hacia coordenadas objetivo introducidas por el usuario, adaptando continuamente su trayectoria en función de la información obtenida del entorno.

El desarrollo de la versión final del programa, denominada navegación K6, es el resultado de múltiples ensayos y refinamientos realizados durante el proyecto. Esta versión engloba los objetivos fundamentales planteados inicialmente: establecer una comunicación fiable con el robot mediante el SDK oficial en Python y la interfaz Bluetooth, incorporar un sistema de localización absoluta en interiores basado en Marvelmind, integrar un filtro de Kalman 2D para suavizar las mediciones y, sobre esa base, implementar un módulo de navegación autónoma completo. Dicho módulo abarca desde el cálculo del ángulo hacia el objetivo y el ajuste diferencial de las ruedas, hasta la supervisión de obstáculos a través de los sensores infrarrojos, la detección de situaciones de riesgo y la ejecución de maniobras evasivas reactivas. Finalmente, el sistema se complementa con la capacidad de visualizar la trayectoria en tiempo real y de registrar los datos en formato gráfico para su análisis posterior.

La estructura del código refleja esta organización modular. Tras las importaciones necesarias de bibliotecas matemáticas, gráficas y de comunicación, se inicializan los componentes de hardware, entre ellos la baliza Marvelmind y la conexión Bluetooth con el Create 3. A continuación, se definen las constantes y variables globales que configuran el comportamiento del sistema, como las tolerancias de posición y orientación, los umbrales de detección de obstáculos o las estructuras de almacenamiento de la trayectoria.

Un bloque específico se reserva para la gestión de señales del sistema, lo que permite interrumpir la ejecución de manera controlada. Junto a ello, se incluyen diversas funciones auxiliares de navegación, destinadas a calcular ángulos, diferencias angulares o distancias euclídeas, todas ellas esenciales para la toma de decisiones en el bucle principal de control.

Las funciones auxiliares de navegación pueden representarse de la forma:

Distancia euclídea, usada en calcular_distancia. Es la distancia entre el punto inicial y el punto objetivo:

$$D = \sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2}$$

Ángulo hacia el objetivo, usado en calcular_angulo_objetivo. Ángulo del robot hacia el destino:

$$\theta_{\text{objetivo}} = \operatorname{atan2}(y_{\text{dest}} - y_{\text{actual}},\, x_{\text{dest}} - x_{\text{actual}})$$

Diferencia angular mínima, usada en diferencia_angular. Diferencia mínima entre los dos ángulos en grados:

$$\Delta\theta = \big((\theta_2 - \theta_1 + 180) \bmod 360\big) - 180$$

El programa cuenta con rutinas dedicadas a la finalización segura de la ejecución. En este proceso no solo se detiene el movimiento del robot y la adquisición de datos, sino que también se genera un registro gráfico de la trayectoria recorrida y un archivo CSV con los datos experimentales. Esta doble salida, visual y numérica, proporciona un soporte fundamental para el análisis del comportamiento del sistema en distintos ensayos.

La función principal articula todo el flujo de navegación. Al iniciar el programa, el usuario introduce las coordenadas de destino, que pasan a ser la referencia del sistema. Seguidamente se inicializan las variables de control y comienza el bucle de navegación, dentro del cual se repite de manera cíclica la lectura de datos de Marvelmind, la aplicación del filtro de Kalman y la actualización de la posición estimada. Con esta información se evalúa la distancia al destino y se decide el tipo de acción a ejecutar: avanzar hacia el objetivo, reorientarse, modificar la velocidad, activar la maniobra evasiva o detenerse en caso de alcanzar la meta.

El sistema se complementa con mecanismos de representación gráfica en tiempo real, que permiten observar durante la ejecución la trayectoria seguida por el robot en relación con el destino fijado. De este modo, la navegación K6 se configura como una solución integral que combina adquisición de datos, filtrado, control reactivo, seguridad y registro experimental.

La lógica general de estos bloques puede resumirse mediante el siguiente pseudocódigo:

```
inicializar_robot()
inicializar_marvelmind()
inicializar_kalman()

leer_objetivo_usuario()

mientras no alcanzado_objetivo:
    posicion = leer_marvelmind()
    posicion_filtrada = kalman(posicion)

    si obstaculo:
        maniobra_evasion()
    sino si estancado:
        maniobra_recuperacion()
    sino:
        mover_hacia_objetivo(posicion_filtrada)

detener_robot()
guardar_trayectoria()
```

Código 3.8 Lógica de bloques iniciales del sistema de navegación autónoma.

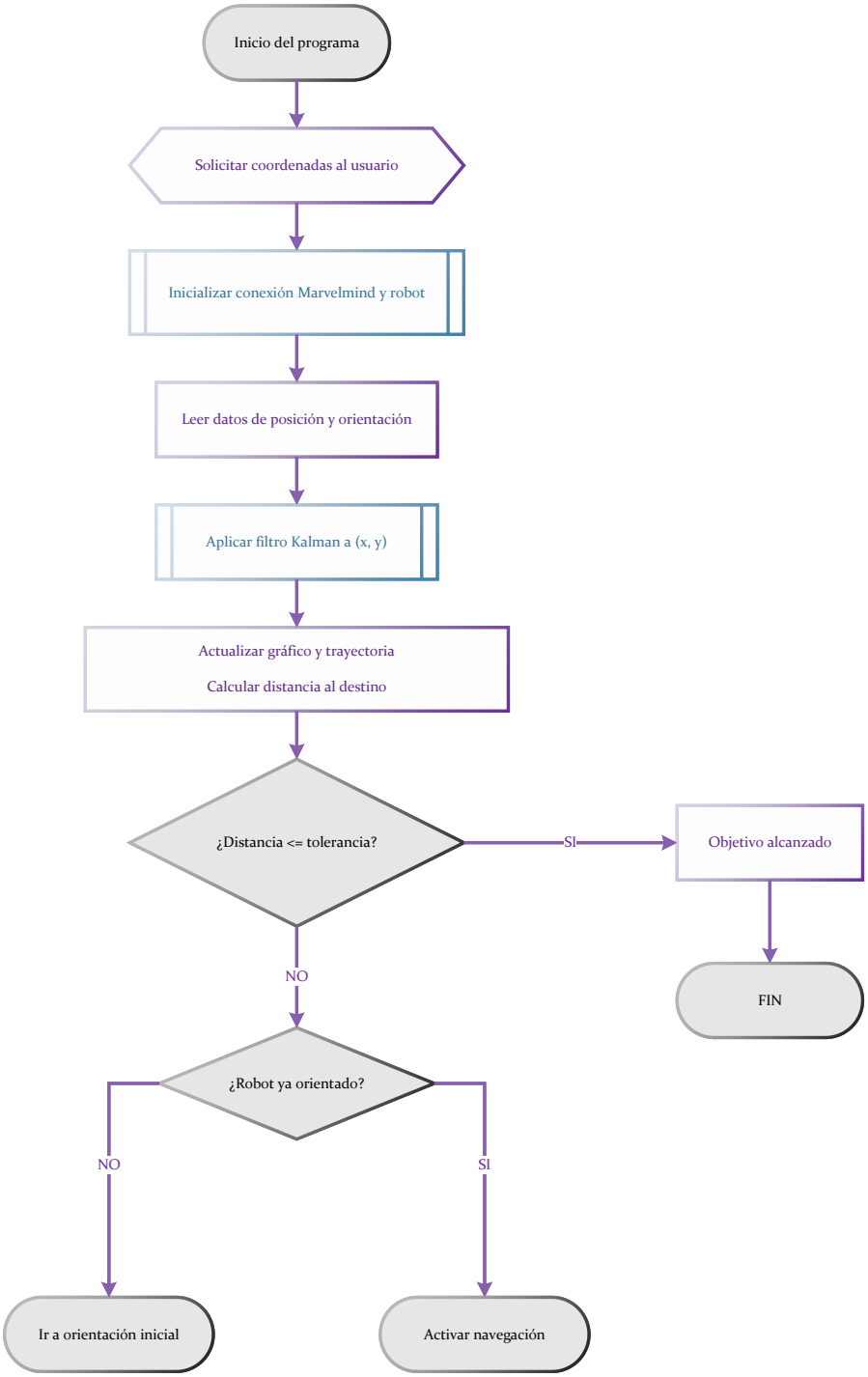


Figura 3.7 Flujo general del sistema (navegacion_K6.py).

3.3.3.1 Orientación inicial antes de moverse

Una vez conocida la distancia y el ángulo hacia el objetivo, el sistema debe garantizar que el robot esté correctamente alineado antes de comenzar a avanzar. El diagrama de flujo correspondiente muestra la lógica de esta fase: primero se evalúa si la diferencia angular $|\Delta\theta|$ excede un umbral de tolerancia. Si el robot ya se encuentra orientado dentro de los márgenes definidos, el sistema habilita el modo de avance. En caso contrario, se activa la rutina de giro correctivo.

Durante la orientación, el robot permanece detenido en el eje lineal y aplica únicamente velocidad angular. El sentido del giro depende del signo de $|\Delta\theta|$: positivo para giro antihorario y negativo para horario. El sistema supervisa continuamente la reducción de este error angular hasta que se sitúe por debajo de la tolerancia. Para evitar movimientos bruscos, la velocidad angular puede modularse proporcionalmente a la magnitud de $|\Delta\theta|$, logrando una aproximación más suave a la orientación deseada.

El diagrama incluye también comprobaciones de seguridad: si durante el proceso se detecta un obstáculo cercano mediante sensores infrarrojos, la rutina se interrumpe y se deriva a la estrategia de evasión. Asimismo, si la orientación no se logra en un tiempo máximo prefijado, se emite una alerta para evitar bloqueos indefinidos.

La lógica general de este bloque puede resumirse mediante el siguiente pseudocódigo:

```
delta_theta = calcular_delta_theta(posicion, objetivo)

si abs(delta_theta) <= theta_tol:
    habilitar_avance()
sino:
    mientras abs(delta_theta) > theta_tol y tiempo < limite:
        if delta_theta > 0:
            girar_antihorario(velocidad_proporcional(delta_theta))
        else:
            girar_horario(velocidad_proporcional(delta_theta))

        delta_theta = recalcular_delta_theta()
        if obstaculo_detectado:
            activar_evasion()
            salir()

    habilitar_avance()
```

Código 3.9 Lógica de la orientación inicial.

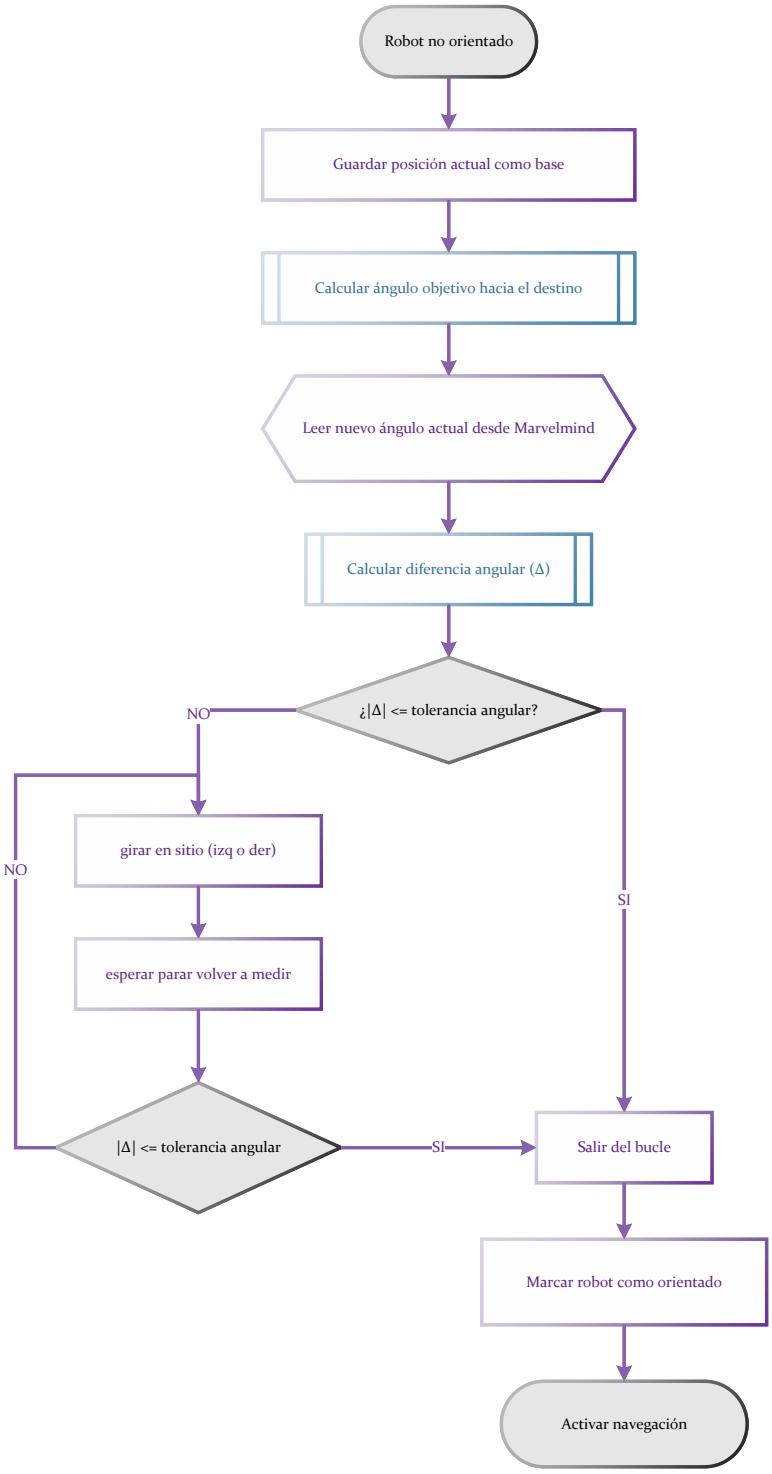


Figura 3.8 Flujo orientación inicial (navegacion_K6.py).

3.3.3.2 Re-cálculo de ángulo hacia el destino

En el transcurso de la navegación autónoma no basta con calcular una sola vez la dirección hacia el objetivo. El robot está en movimiento, y tanto su posición como su orientación cambian en cada instante. Por ello, el sistema debe recalcular continuamente el ángulo hacia el destino en cada iteración del bucle de control.

Este diagrama de flujo recoge precisamente esa lógica: tras cada actualización de la posición actual del robot, se vuelve a calcular la dirección relativa al objetivo mediante la función arcotangente (atan2). A partir de este valor se obtiene la nueva diferencia angular $|\Delta\theta|$, que indica cuánto debe corregir el robot su rumbo para mantener la trayectoria adecuada.

El recalculo periódico cumple varios propósitos. En primer lugar, evita que el robot se desvíe progresivamente debido a errores acumulados en la odometría o a correcciones parciales del sistema Marvelmind. En segundo lugar, permite reaccionar en tiempo real a maniobras evasivas: después de esquivar un obstáculo, el robot debe volver a orientarse hacia el destino, lo que exige un nuevo cálculo de $|\Delta\theta|$ Además, el recalculo constante proporciona robustez frente a las incertidumbres del entorno, ya que asegura que en todo momento se trabaja con la información más reciente disponible.

El sistema aplica el mismo criterio de normalización angular que en la fase de orientación inicial, garantizando que $|\Delta\theta|$ permanezca en el rango [-180º,180º]. De esta manera, las decisiones de giro siempre se realizan por la ruta más corta y evitan oscilaciones indeseadas.

Este recalculo está integrado en el bucle de navegación principal, de modo que cada vez que el robot actualiza su posición también se actualiza el ángulo al destino. El resultado es un control dinámico que ajusta el rumbo de manera continua, permitiendo al robot aproximarse al objetivo de forma progresiva incluso en escenarios donde se ve obligado a realizar múltiples correcciones.

La lógica general de este bloque puede resumirse mediante el siguiente pseudocódigo:

```
mientras modo_autonomo:
    posicion = leer_posicion_actual()
    objetivo = obtener_coordenadas_objetivo()

    theta_obj = atan2(objetivo.y - posicion.y, objetivo.x - posicion.x)
    delta_theta = normalizar_angulo(theta_obj - posicion.theta)

    actualizar_estado(delta_theta)

    si abs(delta_theta) > theta_tol:
        activar_rutina_orientacion()
```

Código 3.10 Lógica de re-cálculo de orientación.

3.3.3.3 Lectura de sensores infrarrojos

Los sensores infrarrojos (IR) integrados en la plataforma iRobot Create 3 constituyen la primera línea de defensa del sistema frente a colisiones. Su función es detectar la presencia de obstáculos en la trayectoria inmediata del robot, proporcionando una medida binaria (detección/no detección) o, en algunos casos, un valor proporcional a la proximidad.

El diagrama de flujo correspondiente representa el proceso de adquisición y validación de lecturas de estos sensores. El ciclo comienza con la consulta periódica de los sensores frontales del robot, integrados en el hardware de la Create 3. Cada lectura se compara con un umbral predefinido: si la señal de infrarrojos indica que la distancia al obstáculo es menor que este valor, el sistema genera un evento de “obstáculo detectado”.

Estas lecturas se realizan de manera continua dentro del bucle de navegación, de forma que el robot evalúa su entorno en cada iteración antes de ejecutar movimientos de avance. El umbral de detección se establece experimentalmente: un valor demasiado bajo puede hacer que el robot no reaccione a tiempo, mientras que un valor demasiado alto podría generar falsas alarmas y limitar la fluidez de la navegación.

El sistema contempla además la posibilidad de lecturas erráticas (ruido o detección momentánea). Para ello, se aplica una regla de confirmación temporal: solo si la condición de obstáculo persiste durante varias lecturas consecutivas se considera válida y activa la rutina de evasión. Esta lógica añade robustez y evita que pequeñas fluctuaciones interrumpan innecesariamente la marcha del robot.

La disposición y numeración de los sensores es la siguiente:

Tabla 3.2 Correspondencia de índices de sensores IR con su posición física en el robot.

Índice	Posición física
0	Lateral izquierdo trasero
1	Lateral izquierdo medio
2	Frontal izquierdo
3	Frontal central
4	Frontal derecho
5	Lateral derecho medio
6	Lateral derecho trasero

En el caso de que se confirme la presencia de un obstáculo, el flujo se desvía hacia la rutina de evasión y reorientación, interrumpiendo inmediatamente el avance. Si no se detecta obstáculo, el sistema retorna al bloque principal de navegación, permitiendo que el robot continúe su trayectoria hacia el objetivo.

La lógica general de este bloque puede resumirse mediante el siguiente pseudocódigo:

```
mientras modo_autonomo:
    lectura_IR = leer_sensores_IR()

    si lectura_IR < UMBRAL:
        contador_obstaculo += 1
    sino:
        contador_obstaculo = 0

    si contador_obstaculo >= N_lecturas_confirmacion:
        generar_evento("obstaculo_detectado")
        activar_rutina_evasion()
    sino:
        continuar_avance()
```

Código 3.11 Lógica de sensores infrarrojos.

3.3.3.4 Detección de obstáculos críticos y cambios repentinos

Además de la lectura rutinaria de los sensores infrarrojos, el sistema incorpora una capa adicional de seguridad destinada a identificar obstáculos críticos y cambios repentinos en el entorno. Este bloque responde a la necesidad de actuar con rapidez cuando la proximidad de un objeto supone un riesgo inmediato de colisión, o cuando el entorno cambia súbitamente (por ejemplo, la aparición inesperada de una persona en la trayectoria del robot).

El diagrama de flujo correspondiente refleja cómo el robot evalúa continuamente no solo la presencia de obstáculos, sino también la velocidad con la que estos aparecen en el rango de detección. Para ello, se comparan lecturas consecutivas de los sensores: si la distancia registrada disminuye bruscamente en un intervalo muy corto de tiempo, el sistema interpreta la situación como un cambio repentino y activa de inmediato los mecanismos de seguridad.

Un obstáculo crítico se define como aquel cuya distancia medida es inferior a un umbral mínimo absoluto (por ejemplo, unos pocos centímetros frente al robot). En este caso, no se activa una maniobra de evasión progresiva, sino un freno de emergencia que detiene el movimiento en seco. Esta reacción inmediata es imprescindible para proteger tanto al robot como a su entorno en situaciones de riesgo.

El módulo contempla además la posibilidad de lecturas falsas o transitorias. Para discriminar entre ruido y situaciones reales, se aplican filtros de validación, como la confirmación de varias lecturas consecutivas o la

comparación con un buffer histórico de medidas. Sin embargo, en el caso de obstáculos críticos se prioriza la seguridad: ante la duda, el sistema prefiere detener al robot y revaluar la situación antes de continuar.

De esta manera, la detección de obstáculos críticos y cambios repentinos actúa como una capa superior de protección, que se ejecuta en paralelo al resto de rutinas de navegación. Su integración asegura que el robot no dependa únicamente de la evasión planificada, sino que sea capaz de reaccionar de forma instantánea a imprevistos en su entorno.

La lógica general de este bloque puede resumirse mediante el siguiente pseudocódigo:

```
lectura_actual = leer_sensores_IR()
diferencia = lectura_anterior - lectura_actual

# Detección de obstáculo crítico
si lectura_actual < UMBRAL_CRITICO:
    detener_robot()
    generar_evento("freno_emergencia")

# Detección de cambio repentino
si diferencia > DELTA_UMBRAL:
    detener_robot()
    generar_evento("cambio_repentino_detectado")

guardar_lectura(lectura_actual)
```

Código 3.12 Lógica de detección de obstáculos críticos y cambios repentinos.

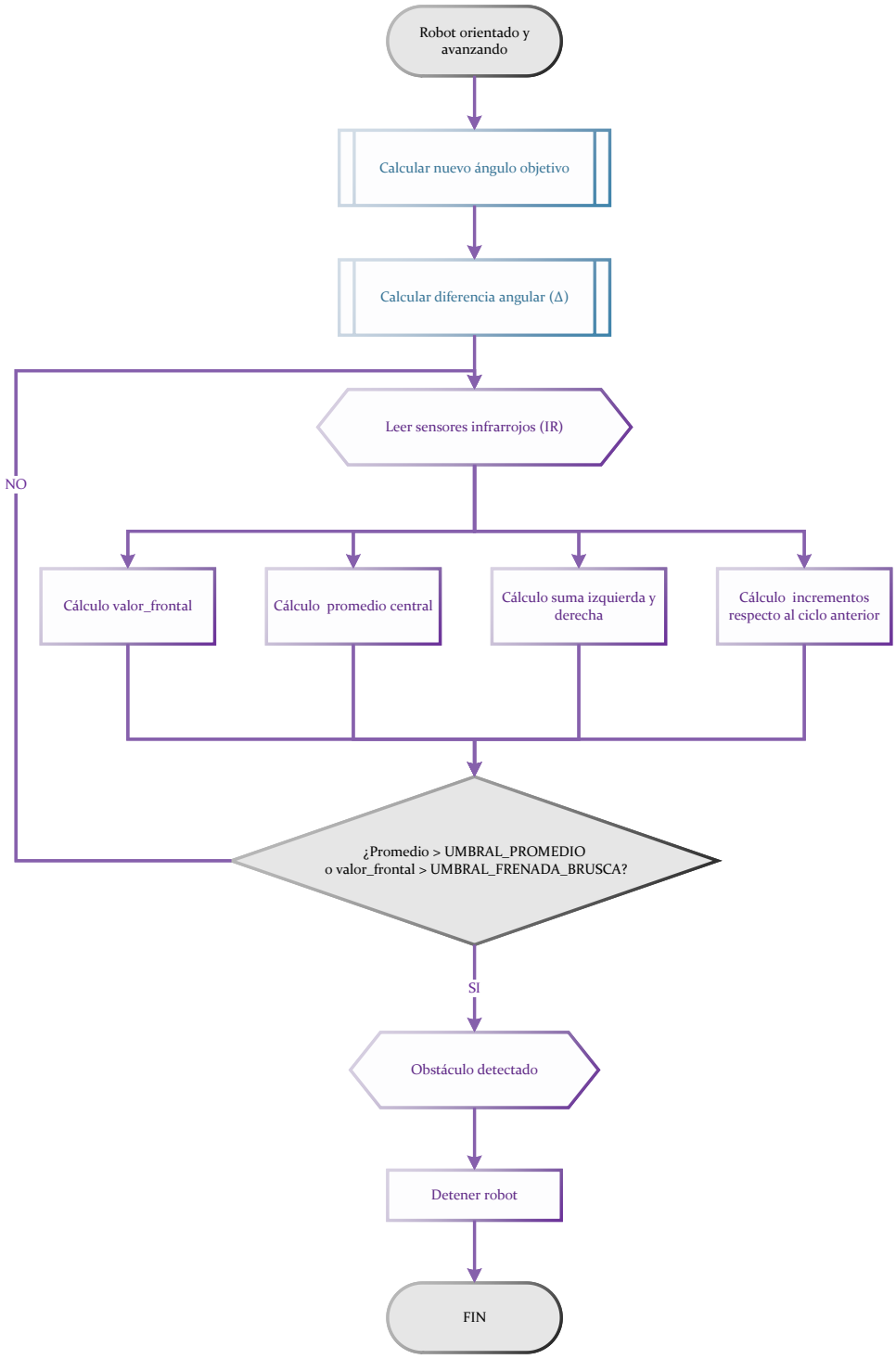


Figura 3.9 Flujo control reactivo de navegación (navegacion_K6.py).

3.3.3.5 Cálculo de velocidad según proximidad y peligro

En la navegación autónoma no basta con decidir de manera binaria entre avanzar o detenerse; resulta imprescindible que la velocidad del robot se ajuste de forma progresiva en función de la proximidad a los obstáculos y del nivel de riesgo que estos representen. El bloque correspondiente del diagrama de flujo refleja cómo el sistema determina una consigna de velocidad lineal y angular que garantice tanto la seguridad del desplazamiento como la fluidez en la ejecución de la trayectoria.

El procedimiento parte de una velocidad nominal, entendida como el valor máximo de avance que el robot adopta en condiciones totalmente seguras. A medida que los sensores infrarrojos registran la presencia de objetos cercanos, la velocidad se va reduciendo gradualmente. Esta disminución es proporcional a la distancia medida, de modo que, cuanto más próximo se encuentre un obstáculo, menor será la velocidad permitida, evitando así aproximaciones bruscas o colisiones inesperadas.

El sistema también contempla la existencia de situaciones intermedias de peligro. En aquellos casos en los que los sensores identifican valores que no justifican una parada inmediata, pero sí advierten de un riesgo potencial, la reducción de velocidad se intensifica. Si las mediciones alcanzan valores cercanos a los umbrales de seguridad establecidos, la consigna puede incluso anularse por completo, deteniendo el avance del robot hasta que las condiciones vuelvan a ser seguras.

Por último, la velocidad se ajusta de manera específica cuando el robot se aproxima a su destino. Reducir progresivamente la consigna al acercarse al punto objetivo permite mejorar la precisión en la llegada y evita que se produzca un sobrepaso de la meta.

El resultado de este enfoque es un sistema de control que no aplica una velocidad fija, sino que genera una consigna adaptativa y sensible a las circunstancias dinámicas del entorno. Gracias a ello, el movimiento se vuelve más estable, se reduce el riesgo de colisión y se incrementa la precisión en la consecución de la trayectoria planificada.

La lógica general de este bloque puede resumirse mediante la siguiente tabla y pseudocódigo :

Tabla 3.3 Velocidad del robot según proximidad y nivel de peligro detectado.

Estado	Condición	Velocidad (unidad)
Zona libre	$valor\_frontal \leqslant 30$	20
Precaución	$30 < valor\_frontal \leqslant 40$ o incremento moderado	15
Precaución/Alerta	$40 < valor\_frontal \leqslant 50$ o incremento alto	10
Alerta/Emergencia	$valor\_frontal > 50$ o incremento muy alto	5
Emergencia	$valor\_frontal > 1500$	0 (Parada inmediata)

```
velocidad = VELOCIDAD_NOMINAL

distancia = leer_sensores_infrarrojos()

si distancia < UMBRAL_CRITICO:
    velocidad = 0
sino si distancia < UMBRAL_PELIGRO:
    velocidad = VELOCIDAD_BAJA
sino si distancia < UMBRAL_PRECAUCION:
    velocidad = VELOCIDAD_MEDIA
sino:
    velocidad = VELOCIDAD_NOMINAL

si cerca_del_objetivo():
    velocidad = reducir_velocidad(velocidad)

enviar_velocidad(velocidad)
```

Código 3.13 Lógica de velocidad según proximidad y peligro.

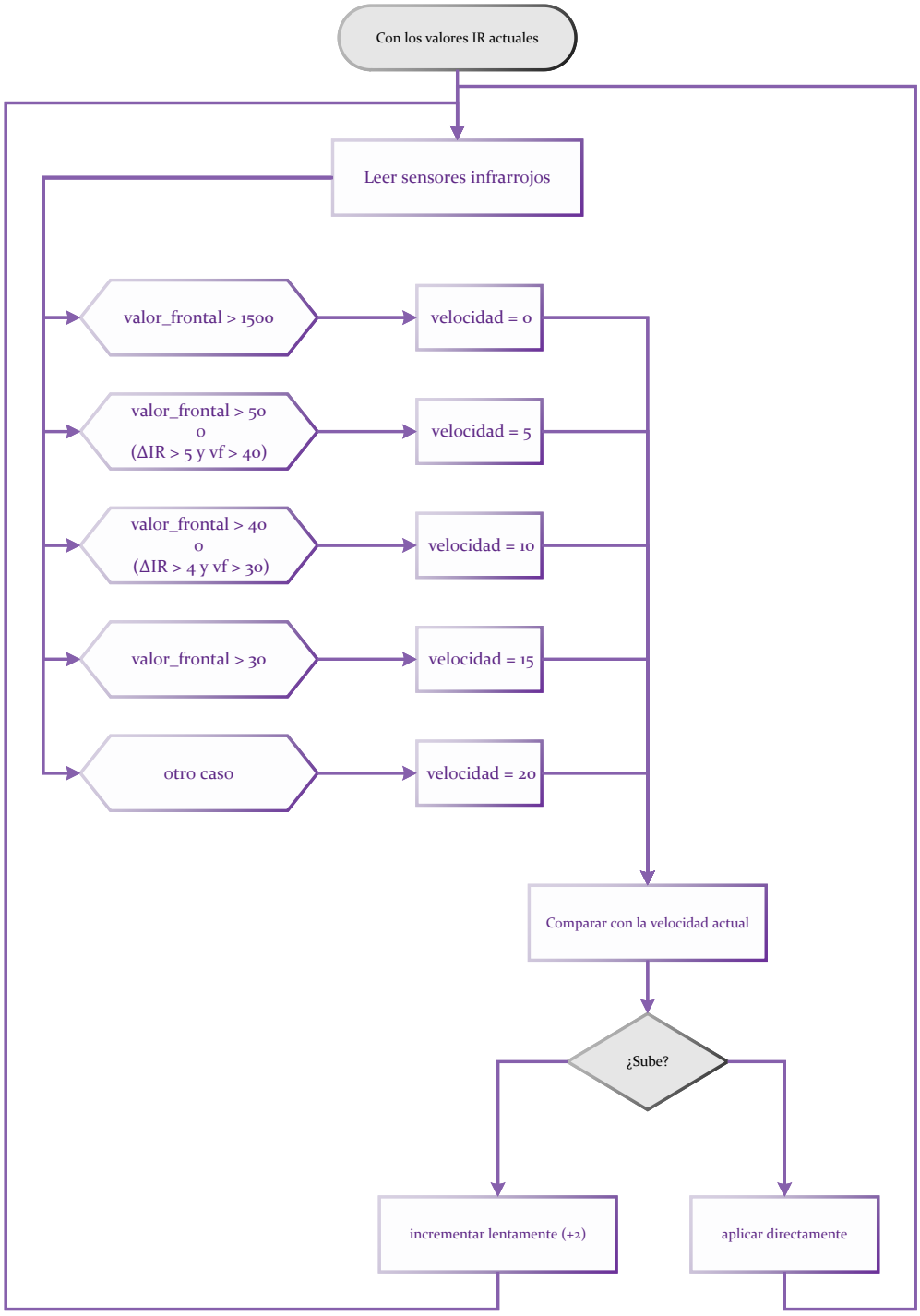


Figura 3.10 Flujo gestión de velocidad adaptativa (navegacion_K6.py).

3.3.3.6 Detección y activación de maniobra evasiva

El bloque de detección y activación de maniobra evasiva constituye la transición entre la fase de navegación normal y la fase de evasión activa. Su finalidad es identificar situaciones en las que la proximidad de un obstáculo hace inviable continuar la marcha directa, y activar de inmediato una maniobra que permita liberar la trayectoria del robot.

El proceso comienza con la evaluación de las lecturas de los sensores infrarrojos de la Create 3. El sistema compara estas lecturas con los umbrales de seguridad: cuando la distancia detectada cae por debajo del valor configurado, se genera un evento de "obstáculo en trayectoria". El diagrama de flujo refleja este punto de decisión: si no existe peligro, el robot continúa su avance normal; si se confirma el obstáculo, se activa la lógica de evasión.

Una vez activada, la maniobra se inicia siempre con un freno inmediato, deteniendo cualquier movimiento lineal en curso para evitar colisión. A continuación, se ordena un retroceso breve que permite al robot ganar margen respecto al obstáculo. Este retroceso tiene una duración limitada y controlada, con el fin de que el robot no se aleje demasiado del objetivo ni interfiera con otras partes del entorno.

La siguiente fase es la reorientación angular: el sistema calcula un ángulo de evasión, que normalmente corresponde a un giro lateral predefinido (por ejemplo, ±90°) o a una estrategia probabilística que introduce cierta variabilidad en el giro para evitar que el robot se quede atrapado en bucles de colisión repetitiva. Tras completar el giro, el robot retoma el modo de avance, ahora en una nueva dirección, desde la cual vuelve a recalcular su ángulo hacia el objetivo y evalúa la conveniencia de reemprender la marcha directa.

El bloque contempla además situaciones de fallo: si durante la maniobra el robot detecta un obstáculo crítico en la dirección de retroceso o durante el giro, el sistema puede reiniciar el proceso con una orientación alternativa. Esta capacidad de adaptación evita que la evasión quede bloqueada en entornos especialmente estrechos.

En conjunto, este diagrama muestra cómo el robot no se limita a frenar ante un obstáculo, sino que dispone de una estrategia activa de evasión, coordinada y segura, que le permite continuar su navegación autónoma en entornos dinámicos y con obstáculos imprevistos.

La lógica general de este bloque puede resumirse mediante el siguiente pseudocódigo:

```
si obstaculo_detectado:
    detener_robot()
    retroceder(tiempo_corto)

    angulo_evasion = calcular_angulo_evasion()
    girar(angulo_evasion)

    if obstaculo_en_rumbo:
        angulo_evasion = angulo_evasion_alternativo()
        girar(angulo_evasion)

    reanudar_avance()
```

Código 3.14 Lógica de detección y activación de maniobra evasiva.

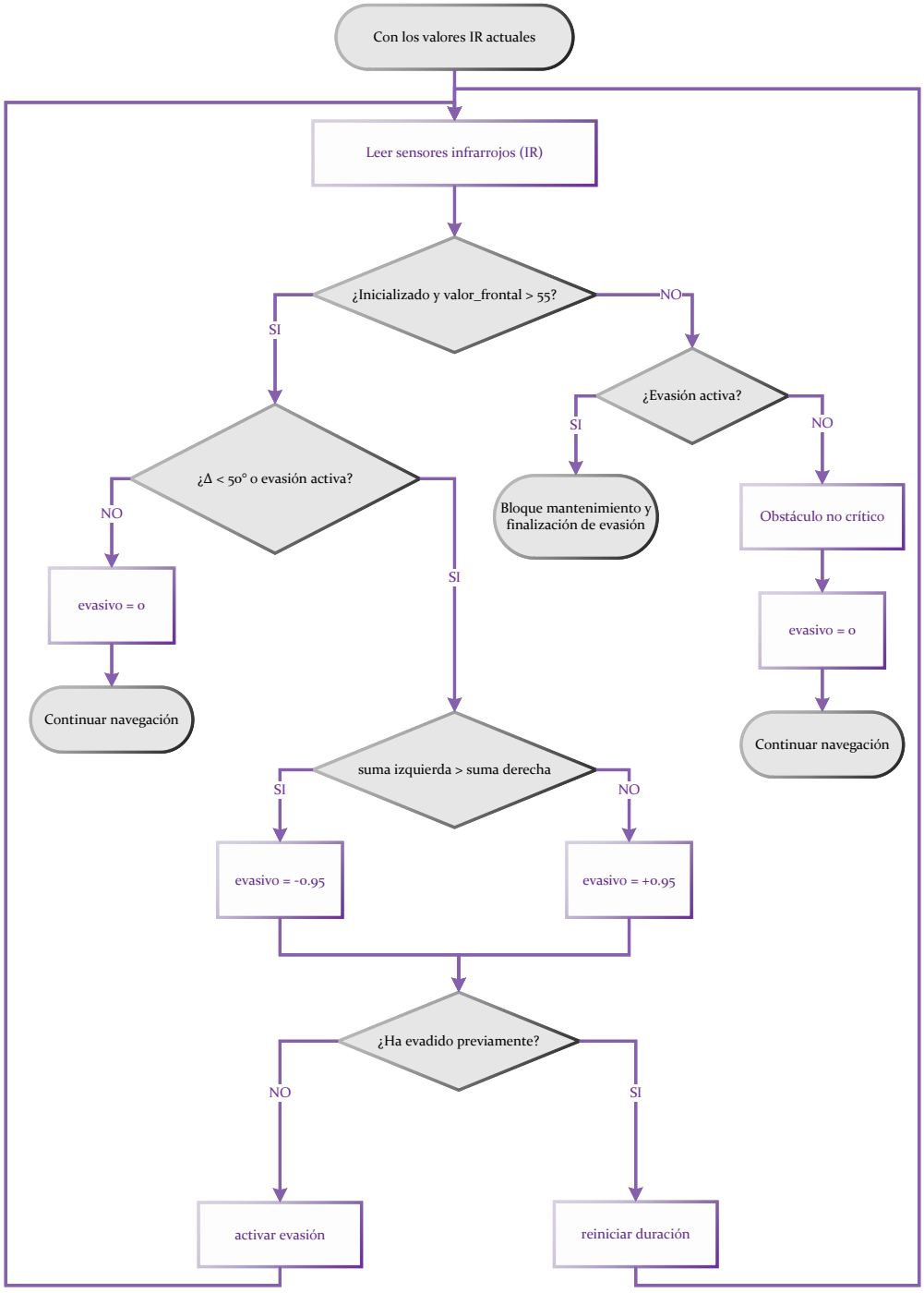


Figura 3.11 Flujo activación de evasión lateral (navegacion_K6.py).

3.3.3.7 Búsqueda de nueva orientación al objetivo

Una vez finalizada la maniobra evasiva, el robot se encuentra en una posición distinta a la inicialmente prevista y con una orientación modificada. Para garantizar que la navegación sigue siendo efectiva, resulta imprescindible llevar a cabo una búsqueda de nueva orientación al objetivo. Este bloque asegura que, tras la evasión, el robot no continúe desplazándose en una dirección arbitraria, sino que reajuste su rumbo hacia el destino.

El diagrama de flujo recoge cómo, tras concluir el retroceso y el giro evasivo, el sistema calcula nuevamente las coordenadas relativas entre la posición actual del robot y el objetivo. Se obtiene el nuevo ángulo hacia la meta y la diferencia con la orientación actual del robot. Si esta diferencia supera el umbral de tolerancia definido, se activa una rutina de reorientación similar a la descrita en la fase inicial de navegación, aunque en este caso partiendo de un contexto posterior a una evasión.

El algoritmo contempla la posibilidad de que el robot se encuentre en un entorno complejo, con obstáculos persistentes o en pasillos estrechos. En esos casos, puede ser necesario realizar varias iteraciones de búsqueda: el robot gira progresivamente, recalcula la diferencia angular y solo cuando encuentra una trayectoria libre hacia el objetivo, procede al avance. El diagrama ilustra estas bifurcaciones como ciclos de reintento hasta que la condición de orientación adecuada se cumple.

Una característica destacada de este bloque es que integra tanto la geometría hacia el objetivo como la información de sensores: no basta con que esté dentro del umbral, también se requiere que en la nueva dirección no haya obstáculos detectados. De esta manera, el robot asegura que la trayectoria de avance hacia la meta es viable y segura.

En definitiva, este bloque dota al sistema de una capacidad de recuperación inteligente tras las evasiones: no se limita a esquivar y avanzar, sino que se reposiciona activamente hacia el destino, manteniendo la coherencia con el plan general de navegación.

La lógica general de este bloque puede resumirse mediante el siguiente pseudocódigo:

```
posicion = leer_posicion_actual()
objetivo = obtener_coordenadas_objetivo()

theta_obj = atan2(objetivo.y - posicion.y, objetivo.x - posicion.x)
delta_theta = normalizar_angulo(theta_obj - posicion.theta)

si abs(delta_theta) > theta_tol:
    mientras abs(delta_theta) > theta_tol:
        girar(sentido(delta_theta), velocidad_proporcional(delta_theta))
        delta_theta = recalcular_delta_theta()

        if obstaculo_en_rumbo:
            girar(angulo_alternativo)
            recalcular_delta_theta()

habilitar_avance()
```

Código 3.15 Lógica de búsqueda de nueva orientación al objetivo.

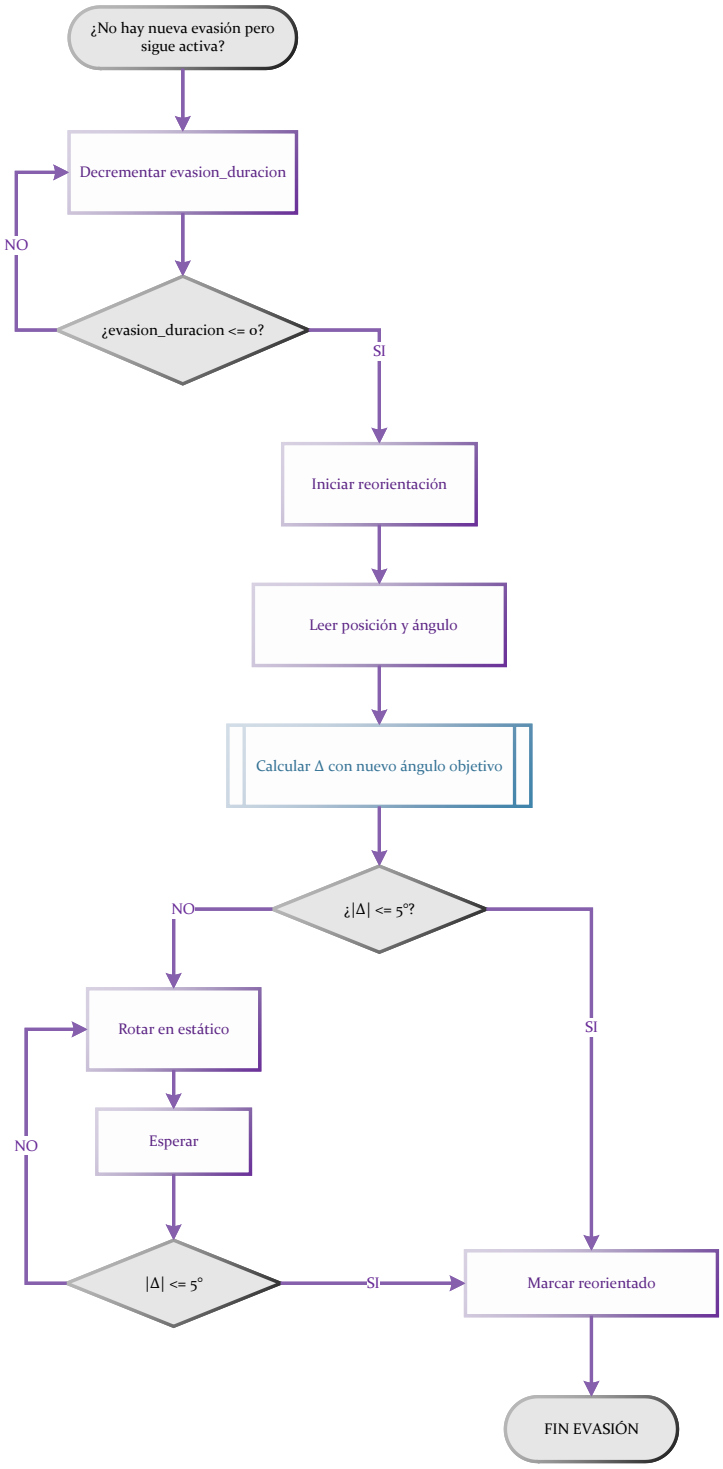


Figura 3.12 Flujo mantenimiento y finalización de evasión (navegacion_K6.py).

3.3.3.8 Detección de estancamiento

El sistema de navegación autónoma contempla la posibilidad de que el robot quede estancado durante su recorrido. Esto sucede cuando, pese a que se envían órdenes de movimiento, la posición del robot apenas varía o lo hace de forma inconsistente con respecto a la consigna. Las causas más comunes son el contacto prolongado con un obstáculo, el atrapamiento en un espacio estrecho o el deslizamiento de las ruedas en superficies de baja fricción.

El diagrama de flujo correspondiente muestra cómo se supervisa de forma continua el progreso del robot. En cada iteración del bucle de control, se compara la posición actual con la anterior. Si en un intervalo de tiempo el desplazamiento acumulado es inferior a un umbral mínimo, mientras el robot debería estar avanzando, se interpreta que existe una situación de estancamiento.

Para evitar falsas detecciones provocadas por el ruido de los sensores o por pausas momentáneas, esta condición debe cumplirse de manera consecutiva durante varias iteraciones. Solo entonces se activa el estado de "robot estancado", lo que desencadena una maniobra de recuperación. Dicha maniobra consiste en detener inmediatamente al robot, retroceder durante un breve intervalo de tiempo y realizar un giro que lo aleje de la situación de bloqueo. Una vez ejecutada esta acción, el sistema vuelve a calcular la orientación hacia el objetivo y reanuda la navegación normal.

Este mecanismo dota al sistema de mayor robustez y autonomía, al permitirle responder por sí mismo a situaciones en las que de otro modo quedaría detenido indefinidamente.

La lógica general de este bloque puede resumirse mediante el siguiente pseudocódigo:

```
1
2  si v_cmd > 0:
3      desplazamiento = posicion_actual - posicion_anterior
4
5      si desplazamiento < UMBRAL_MIN:
6          contador_estancado += 1
7      sino:
8          contador_estancado = 0
9
10     si contador_estancado > N_iteraciones:
11         detener_robot()
12         retroceder(tiempo_corto)
13         girar(angulo_evasion)
14         reanudar_avance()
```

Código 3.16 Lógica de detección de estancamiento.

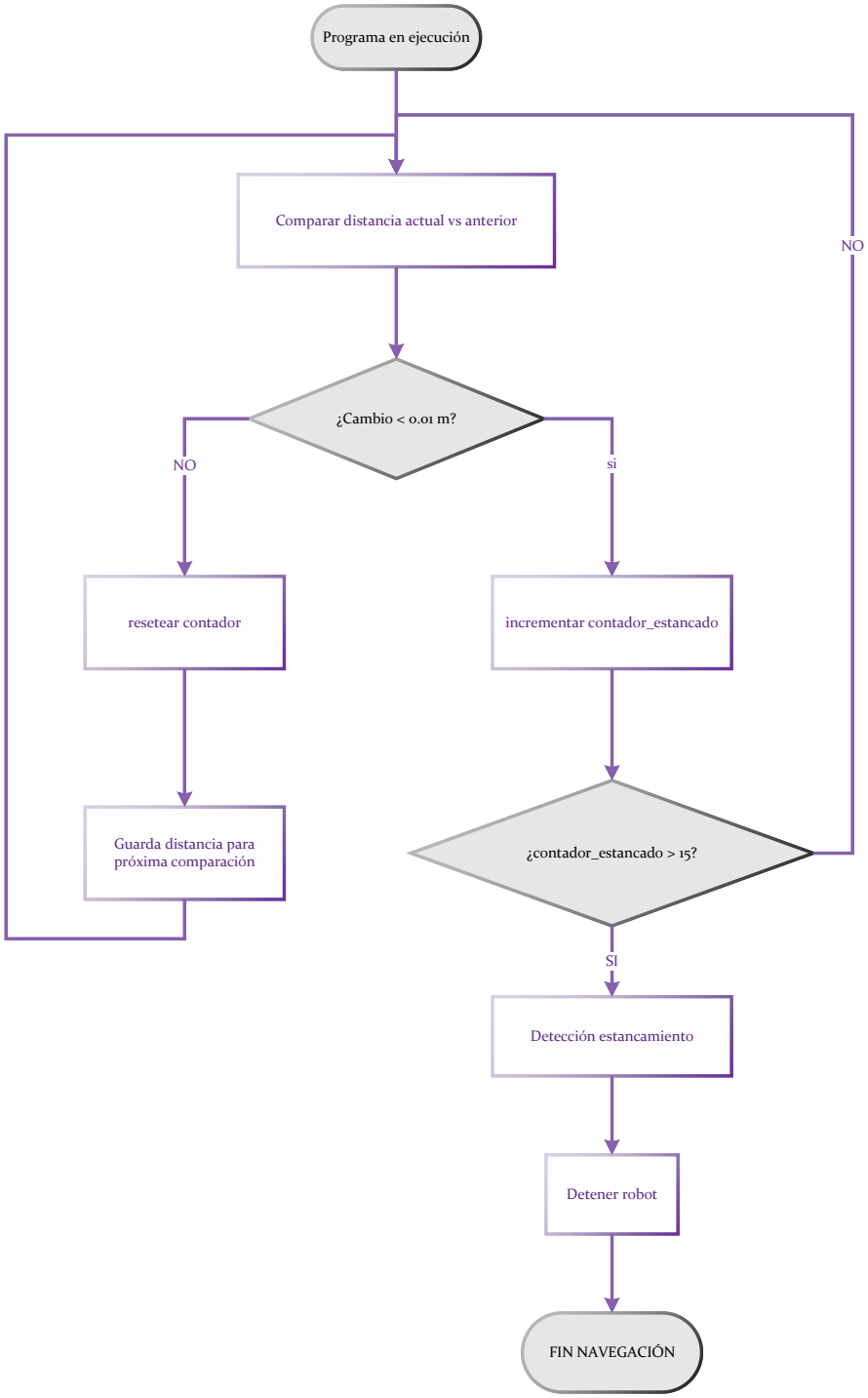


Figura 3.13 Flujo detección de estancamiento (navegacion_K6.py).

3.4 Cuarta Parte (Integración de visión artificial)

El objetivo de esta fase fue dotar al robot de una nueva capacidad de percepción visual mediante la integración del modelo de detección de objetos YOLO (You Only Look Once) en la arquitectura de navegación ya desarrollada. Para ello, se empleó una cámara TP-Link Tapo C200 como fuente de vídeo en tiempo real a través de un flujo RTSP. La intención era complementar los datos de los sensores infrarrojos y el sistema de posicionamiento Marvelmind con información visual, de forma que el robot pudiera no solo navegar hacia coordenadas concretas, sino también identificar y reaccionar ante objetos presentes en el entorno.

3.4.1 Integración de YOLO con el sistema de navegación (navegacion_TAPO.py)

La implementación de esta integración partió directamente de la estructura del sistema de navegación K6, la cual se mantuvo casi sin modificaciones. La única diferencia fue la incorporación de un bloque adicional de visión artificial que funcionaba en paralelo al bucle principal de navegación. Este bloque se ejecutaba en un hilo independiente encargado de la adquisición del vídeo a través de OpenCV y del posterior procesado mediante el modelo YOLOv8 en su versión más ligera (nano). La elección de esta variante se debió a su menor carga computacional en comparación con otros modelos de la familia, lo que en principio debía facilitar la ejecución en tiempo real sin comprometer el rendimiento general.

Dentro de este hilo, el sistema establecía la conexión con la cámara Tapo utilizando las credenciales de acceso y construía la URL RTSP necesaria para abrir el flujo de vídeo. Una vez confirmada la conexión, el programa comenzaba a capturar frames de manera continua. Para reducir la carga sobre el procesador, se optó por procesar únicamente uno de cada varios frames, lo que permitía ejecutar la detección de objetos de forma periódica en lugar de hacerlo sobre cada imagen recibida. Sobre los frames seleccionados, YOLO ejecutaba el proceso de detección, generando una lista de las clases de objetos identificados. Estas detecciones se actualizaban solo cuando se producía un cambio respecto a la lista previa, lo que evitaba redundancias innecesarias. Además, se generaban anotaciones visuales, cajas delimitadoras y etiquetas que se proyectaban sobre los frames, los cuales eran mostrados en una ventana interactiva para supervisar el funcionamiento del sistema.

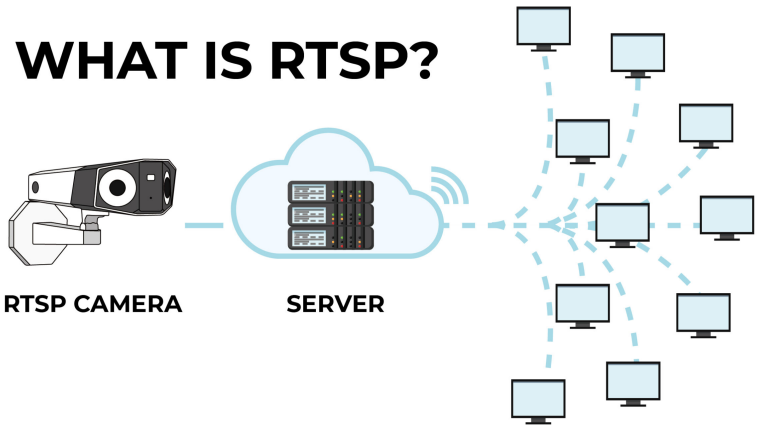


Figura 3.14 Arquitectura RTSP.

La ejecución de este hilo paralelo permitió comprobar en la práctica la viabilidad de integrar detección visual en el proceso de navegación. Sin embargo, a medida que se avanzaba en las pruebas, comenzaron a evidenciarse diversas problemáticas. En primer lugar, la activación del modelo de detección produjo interferencias perceptibles en el rendimiento del robot. Aunque el sistema de navegación funcionaba con normalidad cuando la cámara permanecía inactiva, la activación del módulo de visión generó comportamientos erráticos, especialmente en fases críticas como la alineación inicial y los ajustes finos de rumbo.

A esta dificultad se sumó un uso intensivo de los recursos de la CPU. Mientras que el sistema sin visión artificial mantenía un consumo en torno al ocho por ciento, la activación de YOLO elevó la carga hasta

valores cercanos al cincuenta por ciento. Aunque esta cifra seguía siendo asumible en términos absolutos, en la práctica comprometía la capacidad del robot para mantener un control estable y reactivo. Se detectaron también problemas de asincronía, pues a pesar de ejecutarse en un hilo independiente, el elevado consumo de recursos afectaba al bucle principal de control, introduciendo retardos en la lectura de sensores y en la ejecución de maniobras.

Para mitigar estos inconvenientes se exploraron diversas estrategias. Una de las primeras fue la reducción de la frecuencia de procesamiento, pasando de analizar uno de cada cinco frames a uno de cada treinta. Esto consiguió aliviar de manera notable el uso de la CPU, que descendió hasta valores cercanos al diez por ciento. No obstante, los problemas de estabilidad y reacción del robot persistieron, lo que indicaba que la causa no residía únicamente en la frecuencia de análisis. También se probó desactivar el renderizado de anotaciones visuales, ya que el dibujado de cajas y etiquetas sobre cada frame añadía una carga adicional al procesamiento gráfico. Con esta modificación, se mejoró la fluidez de la detección, aunque no se logró eliminar del todo la interferencia sobre el bucle de navegación.

Pese a estas medidas, la integración de YOLO en paralelo con la navegación no alcanzó un nivel aceptable de estabilidad. El sistema seguía mostrando pérdida de precisión en la orientación y retrasos en la respuesta, lo cual resultaba incompatible con los objetivos del proyecto, cuyo propósito principal era garantizar una navegación fiable y precisa en entornos delimitados.

En vista de estos resultados, se decidió suspender temporalmente la integración de visión artificial, considerándola una fase experimental que, aunque aportó información valiosa, no resultó viable en las condiciones actuales. Se concluyó que el modelo YOLO, incluso en su versión más ligera, introducía una carga excesiva para el sistema, comprometiendo la navegación en tiempo real del Create 3.

La lógica general de este bloque puede resumirse mediante el siguiente pseudocódigo:

```
iniciar hilo_vision():
    conectar_camara_RTSP(usuario, contraseña, ip)

    si no se logra_conexion:
        mostrar("Error al conectar cámara")
        terminar_hilo()

    cargar_modelo_YOLO("yolov8n")

    contador = 0
    N = 30  # procesar solo 1 de cada N frames

    mientras camara_abierta:
        frame = capturar_frame()

        si frame no válido:
            mostrar("Error en captura")
            continuar

        si contador % N == 0:
            detecciones = YOLO.detectar(frame)
            objetos = extraer_clases(detecciones)
            actualizar_lista(objetos)

        mostrar_frame(frame, con_anotaciones)
        contador += 1

        si tecla_presionada('q'):
            salir_bucle()

    cerrar_camara()
    cerrar_ventanas()
```

Código 3.17 Lógica de integración de YOLO con el sistema de navegación.

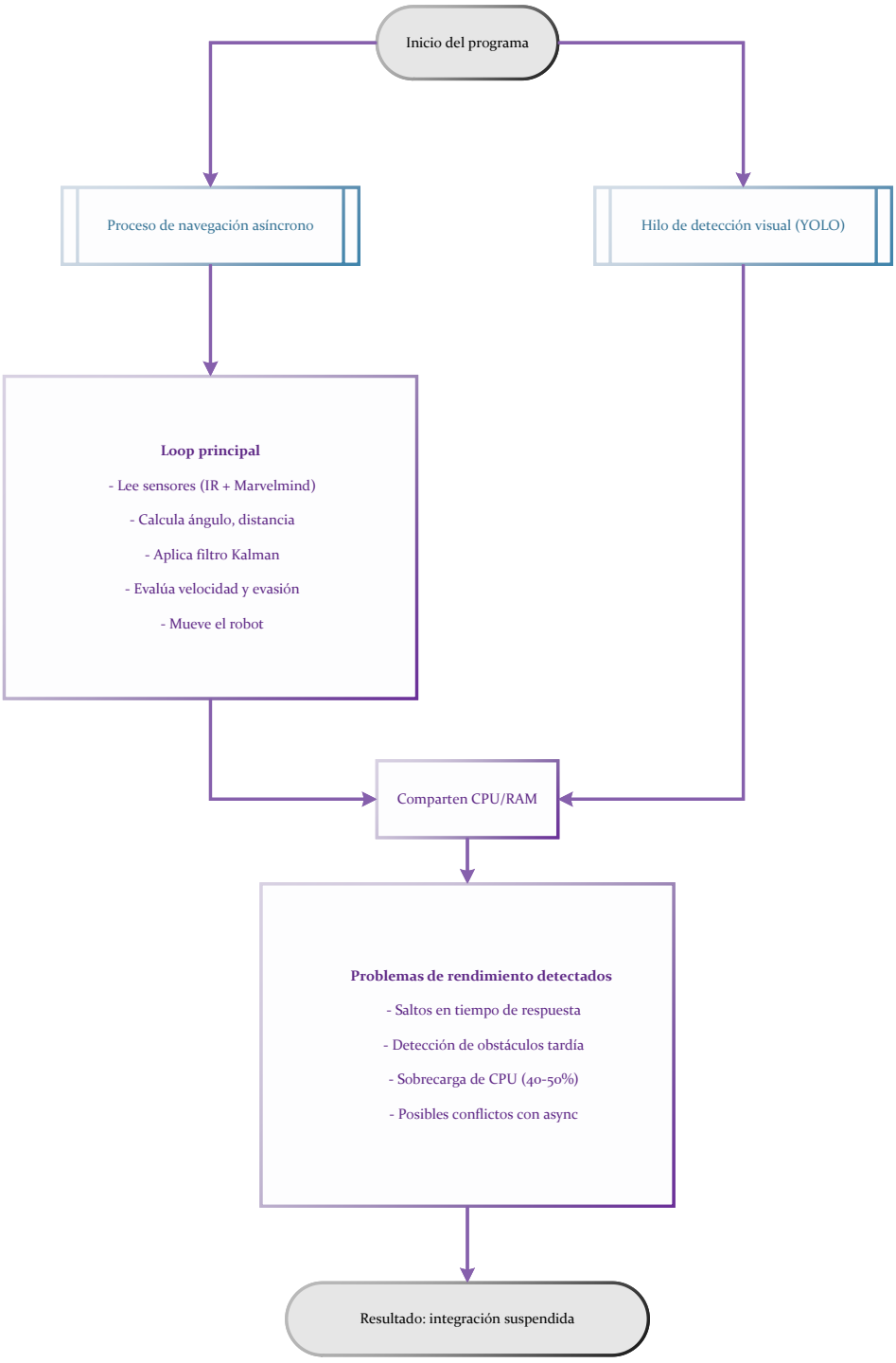


Figura 3.15 Flujo arquitectura paralela de navegación + YOLO (navegacion_TAPO.py).

4 Resultados experimentales

4.1 Primera Parte (Sistemas de control manual)

El objetivo principal de esta fase del proyecto es demostrar el correcto funcionamiento de los códigos informáticos previamente explicados con detalle. Las demostraciones se realizarán mediante vídeos explicativos realizados en el entorno de trabajo y sus correspondientes gráficos, los cuales detallan el comportamiento y funcionamiento de los programas en función del tiempo.

4.1.1 Visualización del control manual del Create 3

En primer lugar, se visualizará el funcionamiento manual del Create 3 con el mando de Playstation 4. Este ensayo se realiza libremente, sin dependencia del sistema de localización externo Marvelmind, el cual se verá posteriormente.

Por ello, la única finalidad de este video es explicar visualmente la relación entre los controles del mando y el comportamiento del robot.

Ver video en YouTube

Además, se incluye un código QR para facilitar el acceso:

Figura 4.1 Código QR odometría interna.

4.1.2 Visualización de la obtención de la odometría interna del robot

En segundo lugar, se extrae la odometría interna del Create 3. Se comienza ejecutando el programa y obteniendo la posición del robot en metros (coordenadas X e Y) y su orientación en grados.

Es importante destacar que en este vídeo las balizas Marvelmind no tienen ningún impacto aunque se explique su funcionamiento y estén conectadas. Su conexión y montaje se debe al siguiente ensayo el cual coordina este sistema con el de la odometría interna.

Como se ha podido apreciar, **el programa aporta medidas muy dispares en cuanto a la posición**; entre lecturas alcanzan diferencias del orden de la decena en metros, lo cual es completamente erróneo. Ya que, se aprecia en el ensayo que el robot apenas se ha desplazado unos centímetros.

Por ello, se decidió descartar en este proyecto cualquier valor o medida relacionado con la odometría interna. El sistema de localización adaptado fue el Marvelmind, de localización externa. La precisión de este sistema auxiliar ha sido probada y demostrada en ensayos posteriores.

Otro punto a detallar es un ruido constante que puede apreciarse durante todo el audio de este vídeo y los siguientes. Este problema se debe a las frecuencias de ultrasonidos que generan las balizas, las cuales perturban la grabación del audio.

Ver video en YouTube

Además, se incluye un código QR para facilitar el acceso:

Figura 4.2 Código QR control manual.

En las siguientes páginas, se adjuntan los gráficos de la ejecución del programa:

1. Análisis de Orientación del Robot

La gráfica de orientación del robot revela un comportamiento altamente dinámico que refleja la naturaleza del control manual implementado mediante el mando de PS4. Durante los primeros 7 segundos de operación, el robot mantiene una orientación relativamente estable en torno a los 100-120 grados, indicando una fase inicial.

A partir del segundo 8, se inicia un período de alta variabilidad angular que caracteriza el resto de la sesión. Los cambios de orientación más dramáticos se observan en múltiples intervalos temporales, con transiciones abruptas que alcanzan variaciones de hasta 180 grados en períodos muy cortos. Estas transiciones extremas, particularmente visibles alrededor de los segundos 9, 11, 14, 21, 27, 33 y 38, muestran la ejecución de maniobras de giro completo implementadas en el código mediante el botón R1 (giro de 180 grados instantáneo).

El patrón de variación angular muestra una alternancia característica entre valores cercanos a 0-50 grados y valores en el rango de 300-360 grados, lo que indica que el robot frecuentemente atraviesa la discontinuidad angular del sistema de coordenadas polares (transición entre 359° y 0°). Esta alternancia es particularmente evidente en los segundos 31-39, donde se observan oscilaciones regulares entre estas dos regiones angulares. La alta frecuencia de cambios direccionales refleja la respuesta inmediata del sistema de control a las entradas del mando, confirmando la efectividad del algoritmo de interpretación de controles que convierte los movimientos del joystick en comandos de velocidad diferencial para las ruedas.

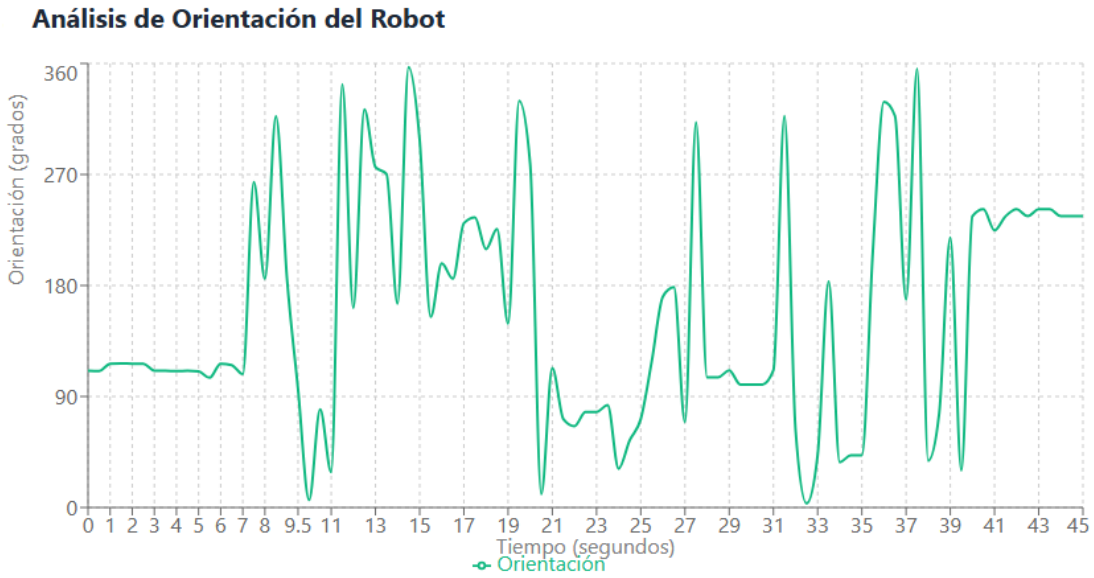


Figura 4.3 Análisis de Orientación del Robot (Odometría interna).

2. Análisis de Velocidad

El análisis de velocidad lineal muestra un comportamiento predominantemente moderado con picos específicos de alta intensidad que revelan las características del control manual implementado. Durante la mayor parte de la sesión, la velocidad se mantiene en rangos bajos a moderados, reflejando un control cauteloso y deliberado por parte del operador.

Los picos de velocidad más significativos se presentan en momentos específicos: alrededor del segundo 22, y posteriormente en los segundos 27 y 41-43. Estos picos extraordinarios sugieren la activación del modo turbo implementado mediante el botón círculo del mando PS4, que incrementa la velocidad máxima de 20 a 40 unidades de velocidad base.

La distribución temporal de las velocidades altas coincide temporalmente con algunos de los cambios de orientación más dramáticos observados en la gráfica anterior, indicando que el operador combinó maniobras de alta velocidad con cambios direccionales significativos, posiblemente explorando las capacidades máximas del sistema de control. Los períodos de velocidad cercana a cero, especialmente visibles en los segundos 1-7 y en varios intervalos intermedios, corresponden a fases de parada controlada.

Es muy importante recalcar que las velocidades en metros por segundo aportadas por la odometría son completamente erróneas, en el video del ensayo puede observarse con claridad que la velocidad es infinitamente menor. Estas lecturas completamente alejadas de la realidad son las responsables de que se optara por no hacer uso de la odometría interna del robot y se implementara un sistema de localización externo.

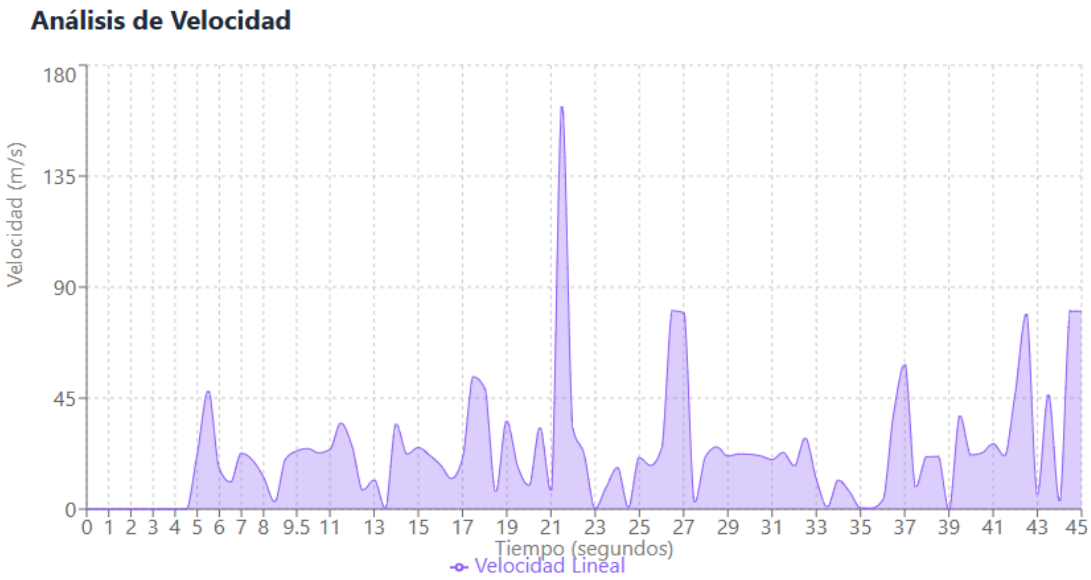


Figura 4.4 Análisis de Velocidad (Odometría interna).

3. Distancia Total Recorrida

La evolución de la distancia total acumulada presenta un patrón de crecimiento monotónico que refleja la actividad continua del robot durante toda la sesión de control manual. El comportamiento muestra tres fases distintivas claramente diferenciadas en la pendiente de acumulación de distancia.

La fase inicial (segundos 0-10) exhibe un crecimiento moderado y relativamente lineal, alcanzando aproximadamente 200 metros acumulados. Esta fase corresponde al período de orientación inicial observado en las gráficas anteriores, donde el operador se familiariza con el sistema y ejecuta movimientos exploratorios de baja intensidad.

La fase intermedia (segundos 10-30) muestra un incremento significativo en la pendiente, evidenciando un período de mayor actividad locomotriz. Durante este intervalo, la distancia acumulada se incrementa de 200 a aproximadamente 600 metros, indicando una tasa de desplazamiento promedio considerablemente superior. Esta fase coincide temporalmente con los picos de velocidad y los cambios de orientación más dinámicos observados anteriormente.

La fase final (segundos 30-45) presenta la pendiente más pronunciada de toda la sesión, con un crecimiento acelerado hasta alcanzar más de 1000 metros de distancia total acumulada. Este comportamiento muestra que el operador había adquirido suficiente destreza en el manejo del sistema para ejecutar maniobras más complejas y mantener el robot en movimiento continuo con mayor confianza.

Como se puntualizó anteriormente, las medidas en metros son erróneas. Sin embargo, a pesar de que el orden de magnitud no sea el real, se pueden estudiar similitudes del comportamiento entre gráficas dejando a un lado el sistema de referencia.

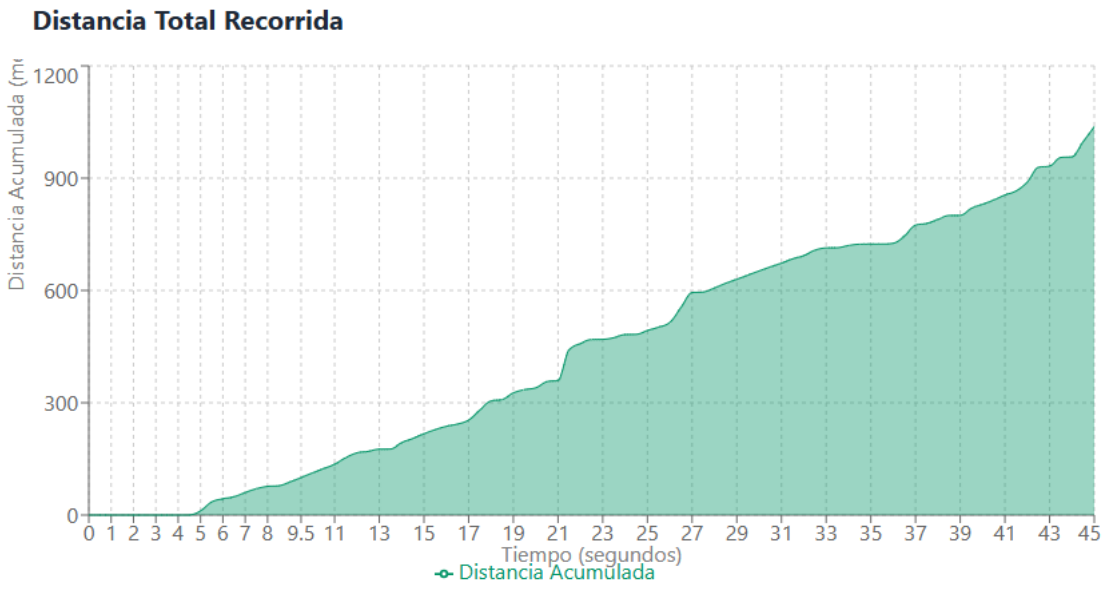


Figura 4.5 Distancia Total Recorrida (Odometría interna).

4. Distribución de Velocidades

El histograma de distribución de velocidades proporciona una perspectiva estadística detallada del comportamiento locomotriz durante la sesión de control manual. La distribución presenta una característica fuertemente sesgada hacia las velocidades bajas, con una frecuencia máxima en el rango de velocidades más bajas (8.0-8.5 m/s).

La mayor concentración de datos se encuentra en los rangos de velocidad entre 8.0 y 22.0 m/s, abarcando aproximadamente el 70 por ciento de todas las mediciones. Esta concentración en velocidades bajas a moderadas indica un estilo de control predominantemente cauteloso, consistente con un operador que prioriza el control preciso sobre la velocidad máxima.

Los rangos de velocidad media (22.0-37.0 m/s) muestran una frecuencia considerablemente reducida pero aún significativa, con 2-4 ocurrencias por rango. Estos valores intermedios sugieren períodos de transición entre maniobras o fases de exploración de las capacidades del sistema.

Particularmente notable es la presencia de valores extremos aislados en los rangos superiores (83.5-84.0 m/s, 136.0-136.5 m/s, y 163.0-163.5 m/s), que aparecen como eventos únicos o muy raros. Estos picos extremos corresponden a los momentos de activación del modo turbo observados en la gráfica temporal de velocidad, confirmando que representan eventos excepcionales.

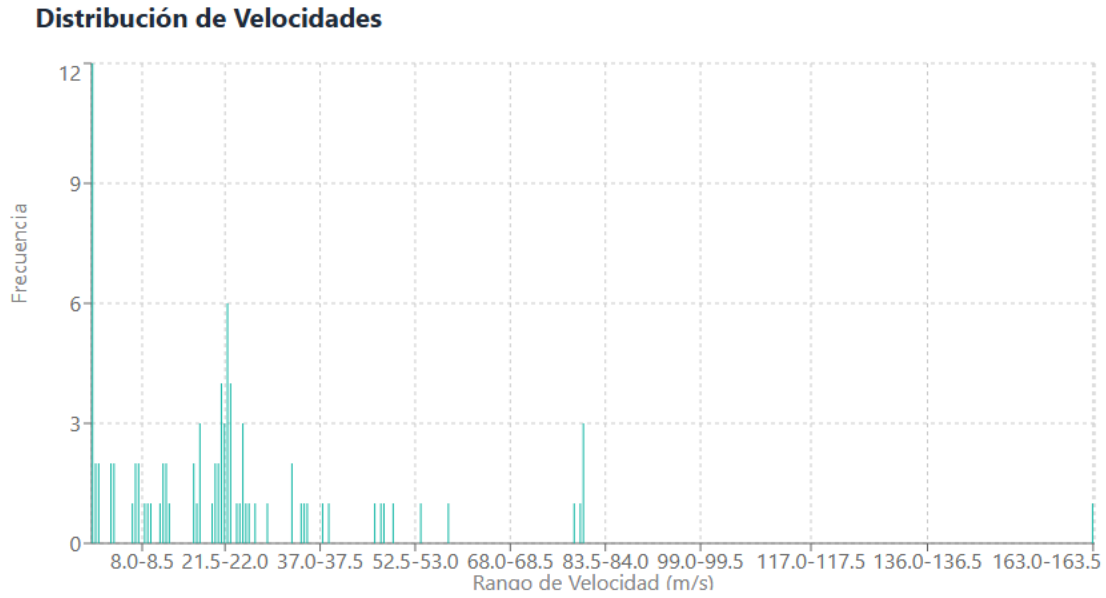


Figura 4.6 Distribución de Velocidades (Odometría interna) .

5. Evolución de la Posición en el Tiempo

La gráfica de evolución posicional muestra trayectorias complejas y aparentemente independientes para las coordenadas X e Y, revelando la naturaleza exploratoria del control manual ejecutado. La coordenada X (línea azul) presenta un patrón de variación más amplio, oscilando entre valores de aproximadamente -70 y +110 metros, mientras que la coordenada Y (línea roja) muestra variaciones más contenidas en el rango de -55 a +70 metros.

Durante los primeros 8 segundos, ambas coordenadas muestran variaciones relativamente menores, consistentes con la fase de orientación inicial identificada en análisis previos. La coordenada X permanece en valores negativos moderados (-10 a -20), mientras que Y se mantiene cerca de cero, sugiriendo movimientos exploratorios limitados en las proximidades del punto de origen.

El período más dinámico se extiende desde el segundo 8 hasta el 35, caracterizado por excursiones significativas en ambas dimensiones. La coordenada X alcanza su valor mínimo (aproximadamente -70 metros) alrededor del segundo 12, seguido por una transición progresiva hacia valores positivos que culminan en un máximo de +110 metros cerca del segundo 35. Esta evolución sugiere un desplazamiento neto considerable en la dirección este durante esta fase.

Simultáneamente, la coordenada Y ejecuta un patrón más errático pero igualmente significativo, con un máximo prominente de +70 metros alrededor del segundo 23, seguido por variaciones que indican cambios frecuentes en la dirección norte-sur. La aparente independencia entre las evoluciones de X e Y confirma que el operador exploró el espacio de manera bidimensional, utilizando efectivamente las capacidades de movimiento omnidireccional del sistema.

Figura 4.7 Evolución de la Posición en el Tiempo (Odometría interna).

6. Trayectoria Completa del Robot

La representación espacial de la trayectoria completa revela un patrón de exploración altamente complejo que abarca un área rectangular.

La trayectoria no sigue un patrón geométrico regular, sino que presenta características de exploración libre y aparentemente aleatoria. Se pueden identificar varias agrupaciones de puntos que sugieren áreas de mayor actividad o regiones donde el operador realizó maniobras repetitivas o exploró con mayor detalle. Una agrupación notable se localiza en la región central-izquierda del gráfico, donde se observa una alta densidad de puntos que sugiere actividad prolongada o maniobras complejas en esa zona específica. Otra concentración significativa aparece en la región superior, distribuida a lo largo de diferentes valores de X, indicando exploración longitudinal en esa franja.

La distribución espacial confirma que el sistema de control manual permitió al operador acceder efectivamente a todo el espacio de navegación disponible, sin evidencia de restricciones espaciales o zonas inaccesibles. La variabilidad en la densidad de puntos refleja las diferentes intensidades de exploración en distintas regiones.

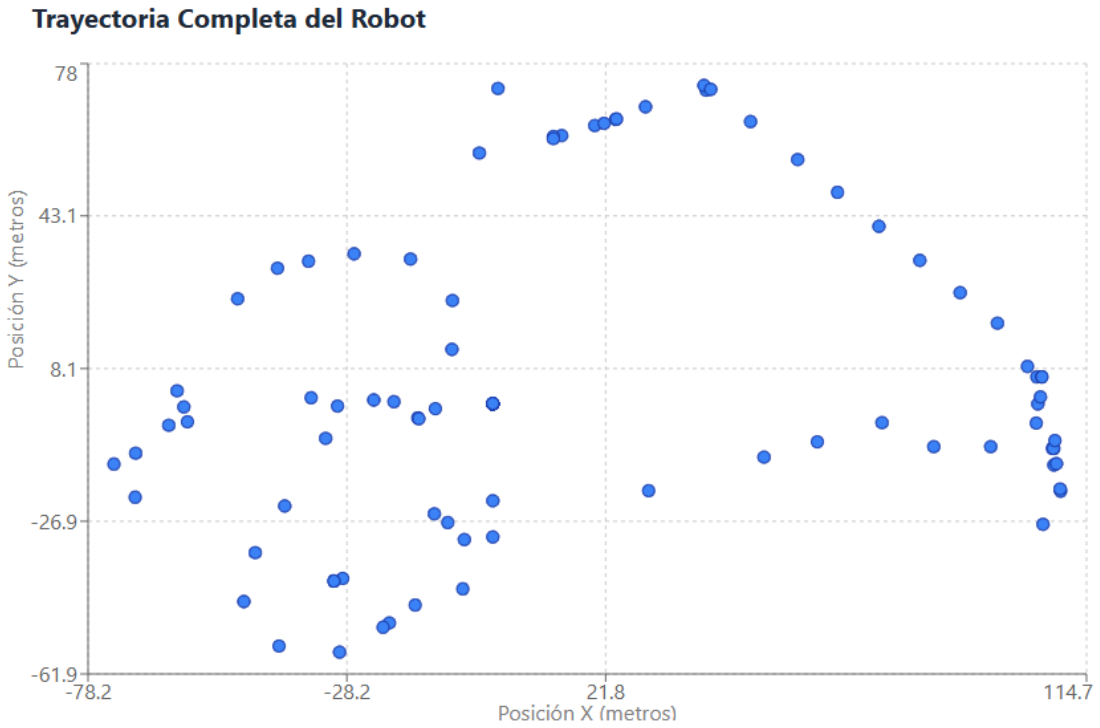


Figura 4.8 Trayectoria Completa del Robot (Odometría interna).

7. Velocidad Angular

La gráfica de velocidad angular presenta un comportamiento extremadamente dinámico que refleja la respuesta inmediata del robot a los comandos de orientación del mando PS4. Los valores de velocidad angular oscilan en un rango amplio, desde prácticamente 0°/s hasta picos máximos que alcanzan 360°/s, indicando la capacidad del sistema para ejecutar giros tanto graduales como abruptos según las demandas del operador.

Los primeros 7 segundos muestran velocidades angulares relativamente bajas (0-50°/s), consistentes con la fase de familiarización inicial identificada en análisis previos. Este comportamiento sugiere movimientos de orientación cautelosos mientras el operador se adapta a la respuesta del sistema de control.

A partir del segundo 8, se inicia un período de alta variabilidad angular caracterizado por múltiples picos de velocidad extrema. Los picos más significativos alcanzan valores de 360°/s en los segundos 11, 14, 21, 27, 37 y 39, correspondiendo a las maniobras de giro instantáneo de 180° implementadas mediante el botón R1 del mando. Estos eventos aparecen como pulsos discretos de duración muy breve, confirmando la naturaleza instantánea de estas maniobras programadas. Los períodos entre estos picos extremos muestran velocidades angulares variables en rangos moderados (50-200°/s), indicando giros continuos controlados mediante el joystick analógico. La variabilidad en estos rangos intermedios refleja los ajustes dinámicos de orientación realizados por el operador durante la navegación libre.

Particularmente notable es la alternancia entre períodos de alta actividad angular y fases de relativa estabilidad, sugiriendo un patrón de control que combina maniobras direccionales decisivas con períodos de movimiento directo. Este comportamiento es característico de un control manual exploratorio donde el operador alterna entre cambios de rumbo significativos y navegación directa en la dirección seleccionada.

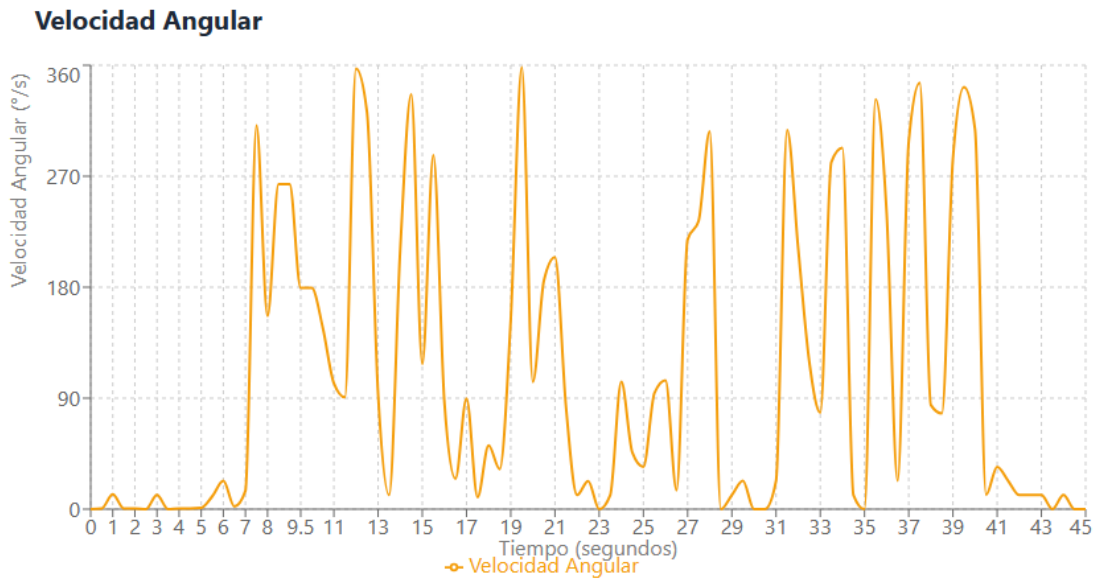


Figura 4.9 Velocidad Angular (Odometría interna).

4.1.3 Visualización del funcionamiento del launcher

En tercer lugar, se añaden al entorno de trabajo las balizas Marvelmind; su disposición y funcionamiento fueron detallados en la sección "Inicialización y configuración del proyecto" (aunque estuviesen añadidas en el anterior ensayo, su impacto se refleja en este).

Como se detalló anteriormente, el launcher coordina 3 subsistemas de manera conjunta:

- roomba_pos.py (odometría)
- marvelmind_process.py (posición absoluta)
- coordinador.py (cálculo del error)

De esta manera, se obtiene el margen de error en cada lectura de posición.

Ver video en YouTube

Además, se incluye un código QR para facilitar el acceso:

Figura 4.10 Código QR funcionamiento launcher.

En las siguientes páginas, se adjuntan los gráficos de la ejecución del programa:

1. Evolución del error de localización

El primer gráfico presenta una visualización del error acumulado de localización a lo largo del tiempo durante el ensayo de comparación en tiempo real entre el sistema de odometría de la Roomba Create 3 y el sistema de posicionamiento Marvelmind. La representación temporal abarca un período de aproximadamente 15 segundos, desde el segundo 15 hasta el segundo 29.5, tiempo durante el cual ambos sistemas estuvieron operando simultáneamente. La curva del error muestra una evolución característica que puede dividirse en tres fases distintas.

Durante la fase inicial (segundos 15-18), el error se mantiene oscilando entre 0 y 40 metros, este comportamiento inicial ya indica una deriva significativa en la precisión del sistema de odometría de la Roomba Create 3.

La fase intermedia (segundos 18-23.5) revela un incremento progresivo y pronunciado del error, alcanzando su pico máximo de aproximadamente 75 metros en el segundo 23.5. Esta escalada del error indica una divergencia significativa entre las estimaciones de posición proporcionadas por la odometría de la Roomba y las mediciones del sistema Marvelmind. El crecimiento sostenido del error durante esta fase sugiere la presencia de errores sistemáticos acumulativos, posiblemente relacionados con derivas odométricas, pérdidas de tracción, o interferencias en las mediciones ultrasónicas.

La tercera fase (segundos 23.5-29.5) muestra una estabilización del error en un nivel elevado, manteniéndose aproximadamente entre 55-58 metros. Esta meseta en el error indica que, aunque la discrepancia entre ambos sistemas se mantiene constante en términos absolutos, no continúa creciendo exponencialmente.

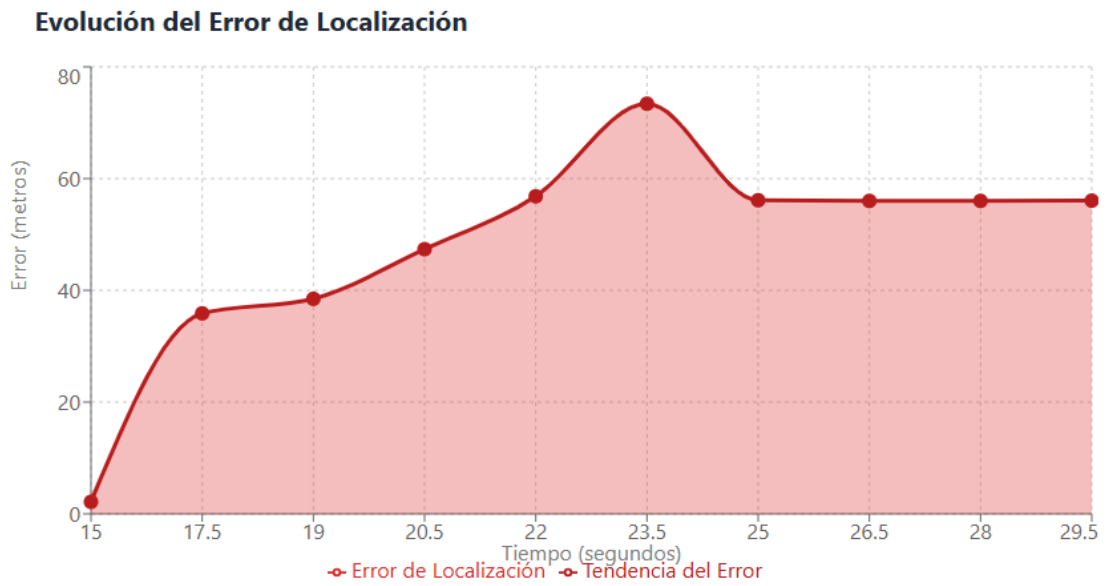


Figura 4.11 Evolución del error de localización (Launcher).

2. Comparación de Coordenadas X - Roomba vs Marvelmind

El segundo gráfico ilustra la evolución temporal de las coordenadas X registradas por ambos sistemas de posicionamiento durante el ensayo. La representación revela comportamientos marcadamente diferentes entre las mediciones de la Roomba (línea roja) y las mediciones de Marvelmind (línea azul).

El sistema Marvelmind (línea azul) presenta una notable estabilidad a lo largo de todo el período de medición, manteniendo valores de coordenada X prácticamente constantes cerca de 0 metros. Esta estabilidad sugiere que el robot se ha desplazado distancias en el orden de los centímetros, como puede observarse en el vídeo del ensayo.

Por el contrario, el sistema de odometría interna de la Roomba (línea roja) muestra un comportamiento completamente diferente y problemático. Durante los primeros segundos del ensayo (hasta aproximadamente el segundo 18.5), la coordenada X se mantiene estable alrededor de 0 metros, creando una falsa impresión de concordancia con las mediciones de Marvelmind. Sin embargo, a partir del segundo 18.5, se produce un cambio abrupto y drástico en las mediciones que revela la naturaleza crítica de las discrepancias entre sistemas. El salto más significativo ocurre entre los segundos 18.5 y 22, donde la coordenada X de la odometría experimenta una transición repentina desde 0 metros hasta aproximadamente -37 metros. Esta discontinuidad muestra un evento específico durante el ensayo que causó una reestimación súbita de la posición por parte del sistema de odometría, posiblemente relacionado con un deslizamiento significativo de las ruedas o un giro brusco.

Tras esta transición abrupta, las mediciones de la odometría interna se estabilizan en el nuevo valor de aproximadamente -37 metros, manteniéndose relativamente constantes durante el resto del ensayo. Al final del período de medición (segundo 29.5), se observa un ligero incremento hacia -35 metros, sugiriendo un pequeño movimiento en dirección positiva del eje X.

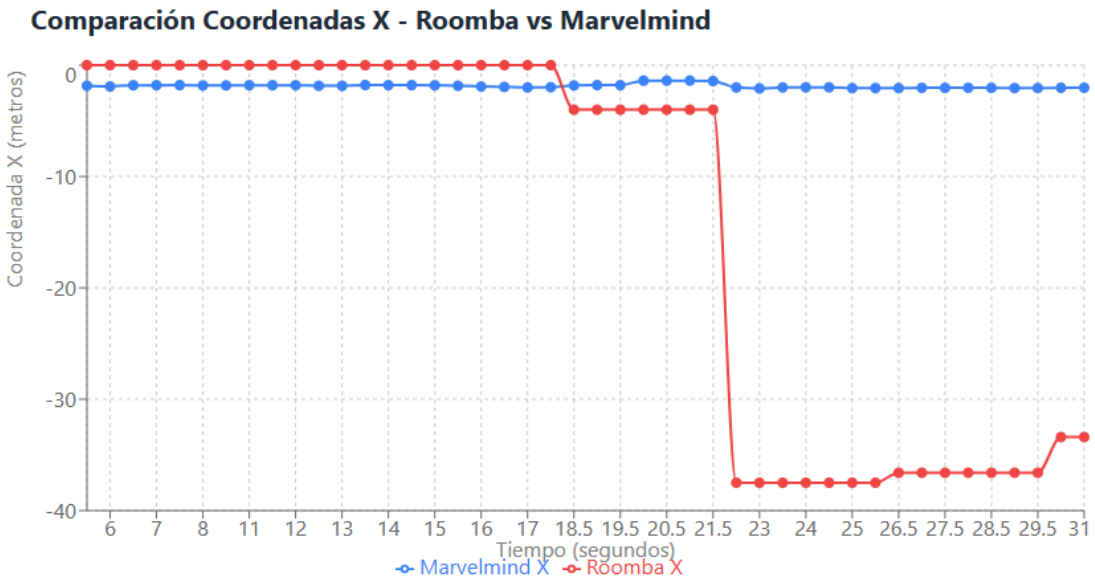


Figura 4.12 Comparación de Coordenadas X - Roomba vs Marvelmind (Launcher).

3. Comparación de Coordenadas Y - Roomba vs Marvelmind

El tercer gráfico presenta la evolución de las coordenadas Y de ambos sistemas, revelando patrones de comportamiento que complementan y confirman las observaciones realizadas en el análisis de las coordenadas X. Al igual que en el eje X, se observa una divergencia fundamental entre las mediciones de ambos sistemas.

El sistema Marvelmind (línea verde) mantiene una estabilidad excepcional en el eje Y, con valores que oscilan mínimamente alrededor de 0 metros durante todo el período de ensayo. Esta consistencia refuerza la interpretación de que, desde la perspectiva del sistema de ultrasonidos, estos movimientos se mantuvieron dentro de la resolución y precisión del sistema Marvelmind.

El sistema de odometría de la Roomba (línea naranja) exhibe un patrón temporal más complejo y dinámico. Durante la fase inicial del ensayo, las coordenadas Y presentan cierta variabilidad, con un incremento repentino que alcanza aproximadamente 20 metros en el segundo 18.5. Este incremento inicial se debe a la acumulación de errores durante la fase de navegación. Entre los segundos 18.5 y 21.5, se produce un período de estabilización relativa alrededor de los 20 metros, seguido de una transición descendente que lleva las coordenadas Y a aproximadamente -5 metros en el segundo 22. Esta transición sugiere un movimiento significativo del robot en dirección negativa del eje Y, o alternativamente, una corrección.

La fase final del ensayo (segundos 25-29.5) muestra una nueva transición descendente más pronunciada, donde las coordenadas Y de la odometría alcanzan valores de aproximadamente -25 metros al final del período de medición. Este descenso progresivo podría indicar un movimiento continuado del robot o la acumulación sostenida de errores en dirección negativa del eje Y.

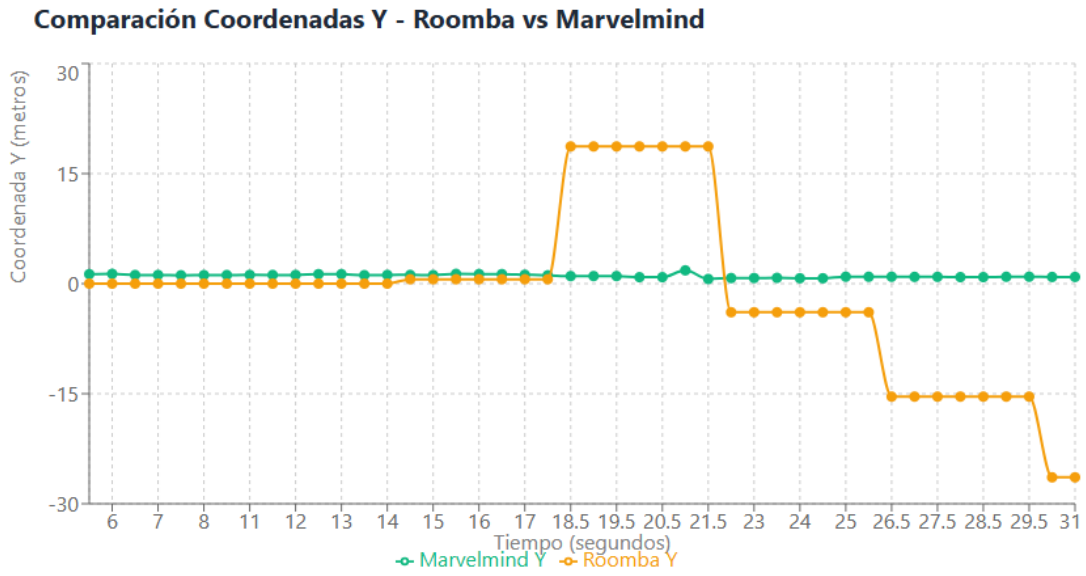


Figura 4.13 Comparación de Coordenadas Y - Roomba vs Marvelmind (Launcher).

4. Distribución del Error de Localización

El cuarto gráfico presenta un histograma que cuantifica la distribución de frecuencias de los errores de localización registrados durante el ensayo completo. Esta representación estadística proporciona una perspectiva complementaria sobre la magnitud y distribución de las discrepancias entre ambos sistemas de posicionamiento.

El análisis de la distribución revela una concentración significativa de errores en los rangos más elevados, con la mayor frecuencia de observaciones registrada en el rango de 50-60 metros. Esta concentración en valores altos de error confirma las observaciones realizadas en los gráficos temporales, donde se identificó una divergencia sustancial entre ambos sistemas durante la mayor parte del ensayo.

El rango de 40-50 metros presenta la segunda mayor frecuencia, mientras que los rangos de menor error (0-10 metros y 10-40 metros) muestran frecuencias mínimas. Esta distribución sesgada hacia errores elevados indica que las condiciones de concordancia entre ambos sistemas fueron limitadas y se concentraron principalmente en momentos específicos del ensayo, probablemente durante las fases iniciales de estabilización. La ausencia de una distribución normal o gaussiana en los errores sugiere que las discrepancias entre sistemas no siguen un patrón de error aleatorio, sino que están dominadas por errores sistemáticos o eventos específicos que causaron divergencias pronunciadas entre las mediciones.

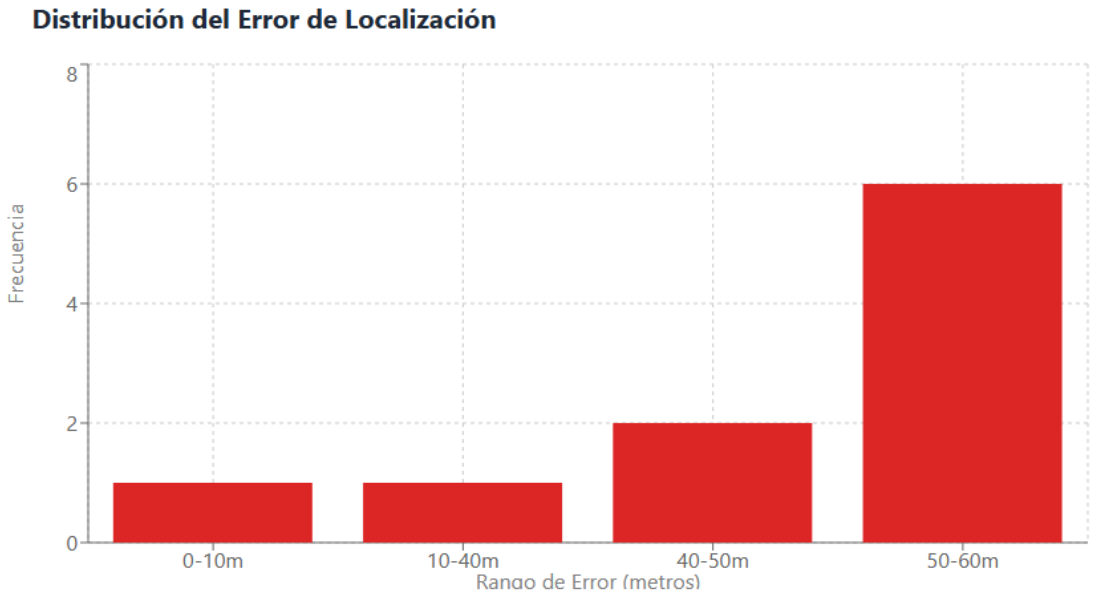


Figura 4.14 Distribución del Error de Localización (Launcher).

5. Trayectorias en el Plano XY

El quinto gráfico presenta una visualización bidimensional de las trayectorias estimadas por ambos sistemas en el plano cartesiano XY, proporcionando una perspectiva espacial integral de las discrepancias observadas entre las mediciones. La trayectoria registrada por el sistema Marvelmind (puntos azules) se concentra en una región muy limitada del espacio, con todas las posiciones agrupadas cerca del origen del sistema de coordenadas (0,0 metros). Esta concentración espacial confirma las observaciones realizadas en los gráficos temporales individuales, donde ambas coordenadas X e Y del sistema Marvelmind mostraron estabilidad excepcional.

En contraste dramático, la trayectoria de odometría interna de la Roomba (puntos rojos) revela un patrón de distribución espacial completamente diferente. Las posiciones se extienden a lo largo de un rango mucho más amplio, con coordenadas que abarcan desde aproximadamente -50 metros hasta 0 metros en X, y desde -27 metros hasta +20 metros en Y. Esta dispersión espacial amplia indica que el sistema registró movimientos significativos del robot a lo largo de toda el área de ensayo. La distribución de los puntos rojos no sigue un patrón de trayectoria continua típico, sino que presenta agrupaciones en diferentes regiones del plano XY. Esta discontinuidad espacial refleja las transiciones abruptas observadas en los gráficos temporales y sugiere la presencia de saltos discretos en las estimaciones de posición de la odometría.

La separación espacial extrema entre ambos conjuntos de trayectorias (azul y roja) visualiza de manera clara la magnitud de las discrepancias entre sistemas, confirmando que los errores de localización observados en los análisis temporales corresponden a diferencias fundamentales en la interpretación del movimiento del robot por parte de cada sistema de posicionamiento.

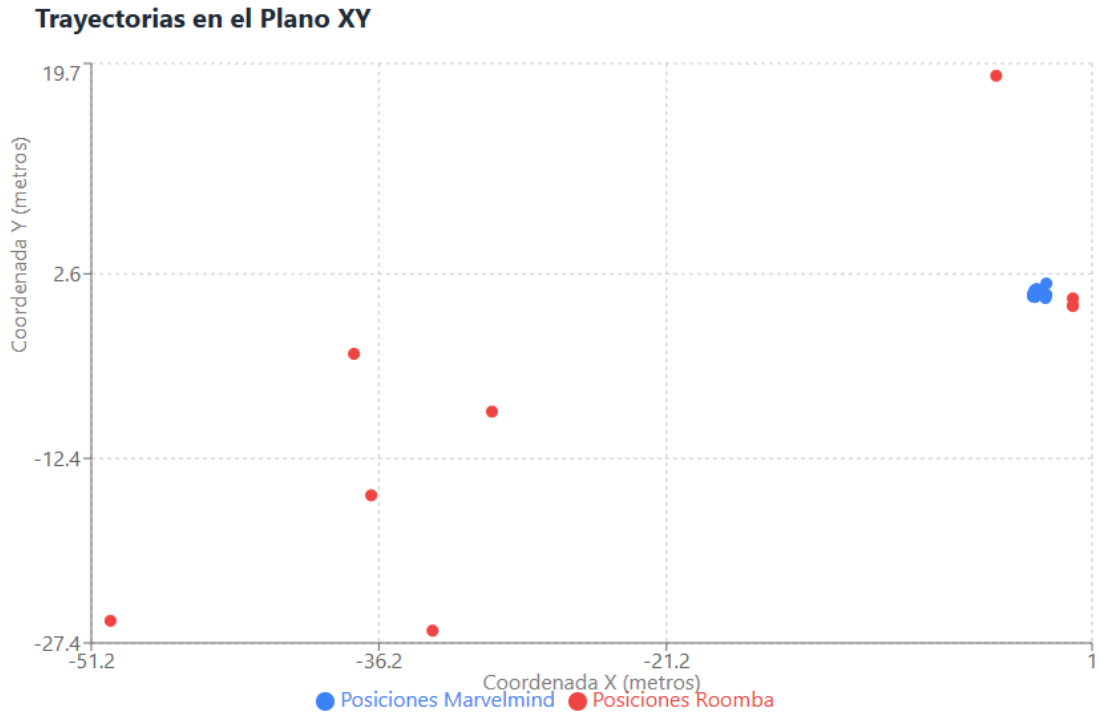


Figura 4.15 Trayectorias en el Plano XY (Launcher).

4.2 Segunda Parte (Estimación del error de posición Marvelmind)

En secciones anteriores, se detallaron los pasos a seguir durante la realización de este ensayo. Por ello, una vez sabido y comprendido el funcionamiento, tras finalizar el proceso y medir el último punto de los 14 especificados, se cierra el programa. Al finalizar, genera de manera automática los siguientes documentos:

Un **archivo excel** con la siguiente forma:

Tabla 4.1 Datos experimentales reales, Marvelmind y error entre ambos.

Punto	X_{real}	Y_{real}	X_{mm}	Y_{mm}	Error_m
1	−0.25	0.25	−0.3586	0.4094	0.192879
2	0.00	0.40	−0.0814	0.6008	0.216672
3	0.00	1.80	0.0544	1.6816	0.130299
4	−0.50	1.60	−0.4260	1.4760	0.144402
5	−0.80	1.00	−0.8679	1.0358	0.076760
6	−1.25	0.00	−1.3358	0.0947	0.127788
7	−1.50	0.25	−1.5916	0.4114	0.185582
8	−1.50	1.50	−1.4390	1.3970	0.119708
9	−2.00	1.00	−1.9397	1.0991	0.116004
10	−2.60	0.75	−2.5054	0.6864	0.113992
11	−3.03	0.50	−2.9353	0.3640	0.165723
12	−3.03	1.25	−2.9143	1.1384	0.160752
13	−2.75	1.75	−2.6956	1.6749	0.092733
14	−2.25	1.92	−2.1935	1.8098	0.123840

Este ensayo es de vital importancia, en los ensayos anteriores, se descartó cualquier tipo de dato relacionado con la odometría interna del robot debido a su considerable desviación de la realidad.

Por lo tanto, era indispensable también verificar la precisión del sistema Marvelmind con respecto a la realidad, el cual arroja resultados notablemente más precisos que la odometría, motivo por el cual se eligió como sistema de referencia de localización durante todo el proyecto.

Es importante recordar que en el sistema de ejes cartesianos escogidos en este proyecto, las coordenadas X tienden al valor negativo por cuestiones de facilidad en cuanto a su posterior representación. Esto se debe al posicionamiento de las balizas en el entorno real de trabajo.

Un **archivo pdf** con el siguiente contenido:

1. Error de Posicionamiento por Punto Medido

El primer gráfico presenta un diagrama de dispersión que muestra la distribución espacial de los errores de posicionamiento del sistema Marvelmind integrado en la Roomba Create 3. En este análisis, cada punto representa una medición realizada en coordenadas específicas del área de ensayo, donde el eje X abarca un rango de -3.0 a 0.0 metros y el eje Y se extiende desde -0.5 hasta 2.0 metros.

La representación utiliza una escala de colores tipo "plasma" que codifica visualmente la magnitud del error de posicionamiento, donde los tonos más oscuros (azul-púrpura) indican errores menores, aproximadamente entre 0.08 y 0.10 metros, mientras que los colores más claros y cálidos (amarillo-naranja) señalan errores superiores, alcanzando valores cercanos a los 0.22 metros. La distribución de los puntos de medición revela una cobertura sistemática del área experimental. Esta disposición sugiere un patrón de muestreo planificado que busca caracterizar el comportamiento del sistema en diferentes regiones del espacio de trabajo.

El análisis espacial del error revela heterogeneidad en el rendimiento del sistema, con algunas zonas mostrando mayor precisión que otras. Particularmente notable es la presencia de puntos con errores elevados en las coordenadas extremas del área de medición. Este fenómeno puede deberse a factores como el eco y reverberación, ya que existen diversos objetos y zonas que pueden potenciar a un comportamiento errático de las ondas ultrasónicas. De hecho, las balizas más próximas a la zona de menor precisión, se encuentran en la zona más abierta de la habitación donde se ubica el entorno de trabajo.

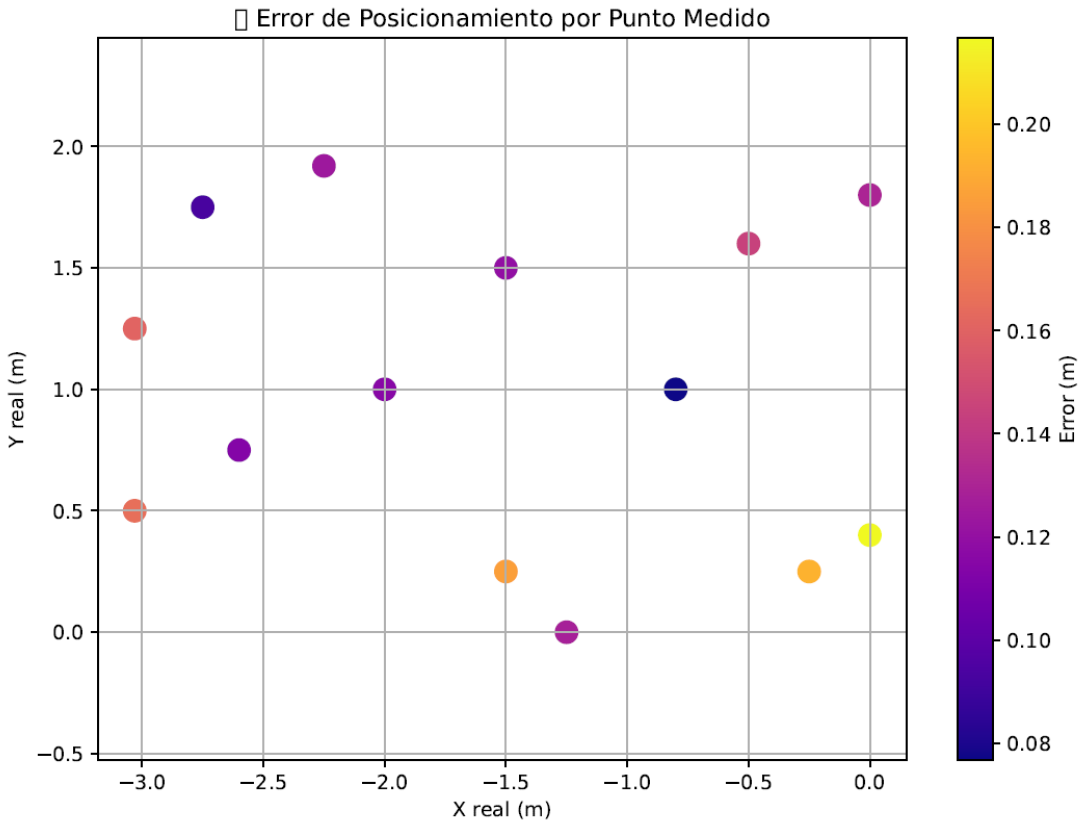


Figura 4.16 Error de posicionamiento por punto medio .

2. Error por Punto Medido

El segundo gráfico presenta una representación en barras que cuantifica individualmente el error de posicionamiento para cada uno de los catorce puntos de medición realizados durante el ensayo. Esta visualización permite identificar de manera directa y cuantitativa el rendimiento del sistema Marvelmind en cada ubicación específica del experimento.

El análisis de los datos revela una variabilidad significativa en el error de posicionamiento a lo largo de las diferentes mediciones. Los valores oscilan entre un mínimo de aproximadamente 0.075 metros (punto 5) y un máximo de 0.22 metros (punto 2), evidenciando una dispersión considerable en la precisión del sistema. Esta variabilidad sugiere que el rendimiento del sistema no es uniforme y depende de factores específicos relacionados con la posición espacial y las condiciones locales de cada punto de medición. Los errores más elevados se concentran en los primeros puntos del ensayo (puntos 1 y 2), con valores de 0.195 y 0.22 metros respectivamente, lo que podría indicar efectos de estabilización inicial del sistema o condiciones particulares en esas ubicaciones específicas. Por el contrario, el punto 5 muestra el mejor rendimiento con un error de tan solo 0.075 metros, estableciendo el límite inferior de precisión alcanzable por el sistema en condiciones óptimas.

La distribución general de errores muestra que la mayoría de las mediciones se sitúan en un rango intermedio entre 0.10 y 0.18 metros, con una tendencia hacia valores más bajos en la segunda mitad del experimento (puntos 8-14), lo que podría sugerir una mejora en las condiciones de medición o en la estabilidad del sistema a medida que progresaba el ensayo.

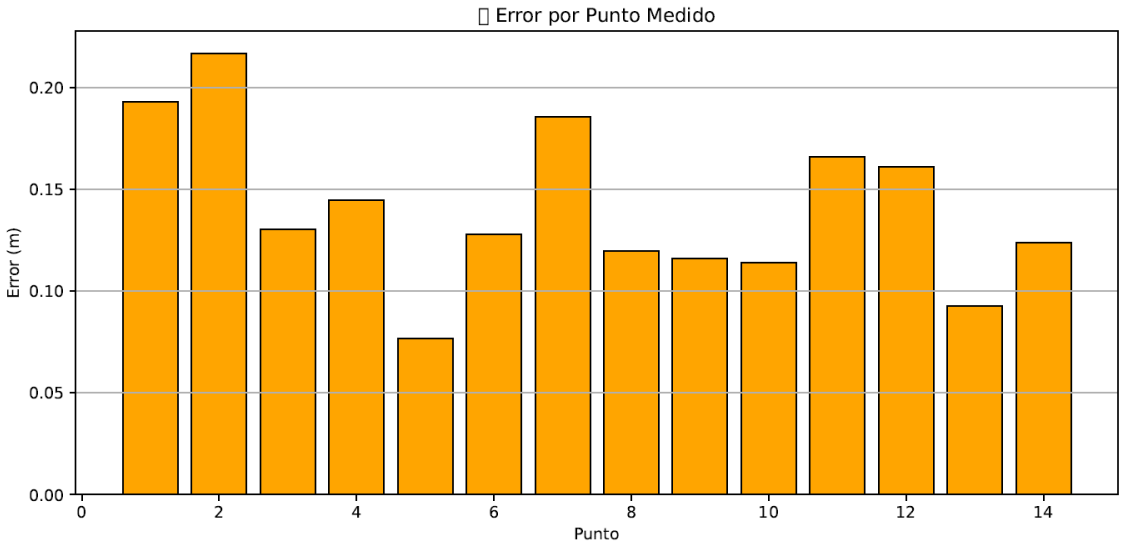


Figura 4.17 Gráfico error por punto medio.

3. Mapa de Calor del Error

El tercer gráfico constituye un mapa de calor bidimensional que emplea estimación de densidad kernel (KDE) ponderada por el error, proporcionando una representación continua y suavizada de la distribución espacial de los errores de posicionamiento. Esta técnica estadística permite visualizar regiones de mayor y menor precisión mediante gradientes de color, donde las intensidades rojas más altas indican zonas con errores de posicionamiento más elevados. El análisis del mapa de calor revela la existencia de múltiples "hotspots" o zonas críticas donde el sistema Marvelmind presenta un rendimiento degradado. Estas regiones de alta intensidad se distribuyen de manera no uniforme a lo largo del área experimental, sugiriendo la presencia de factores ambientales o geométricos que afectan localmente la precisión del sistema de posicionamiento ultrasónico.

Las concentraciones más intensas de error se observan en las coordenadas (-2.2, 1.9), (0.0, 1.8), (-1.5, 0.2), y (0.0, 0.3), formando un patrón irregular que no sigue una distribución geométrica simple. Esta distribución heterogénea sugiere que los errores no están únicamente relacionados con la distancia a los beacons de referencia, sino que pueden verse influenciados por otros factores como reflexiones acústicas, obstáculos, o variaciones en las propiedades acústicas del entorno.

La técnica KDE empleada con un factor de suavizado permite identificar gradientes de transición entre zonas de alta y baja precisión, revelando que el rendimiento del sistema varía de manera continua en el espacio. Las regiones intermedias, representadas por tonalidades más tenues, indican zonas de transición donde el error se sitúa en valores medios.

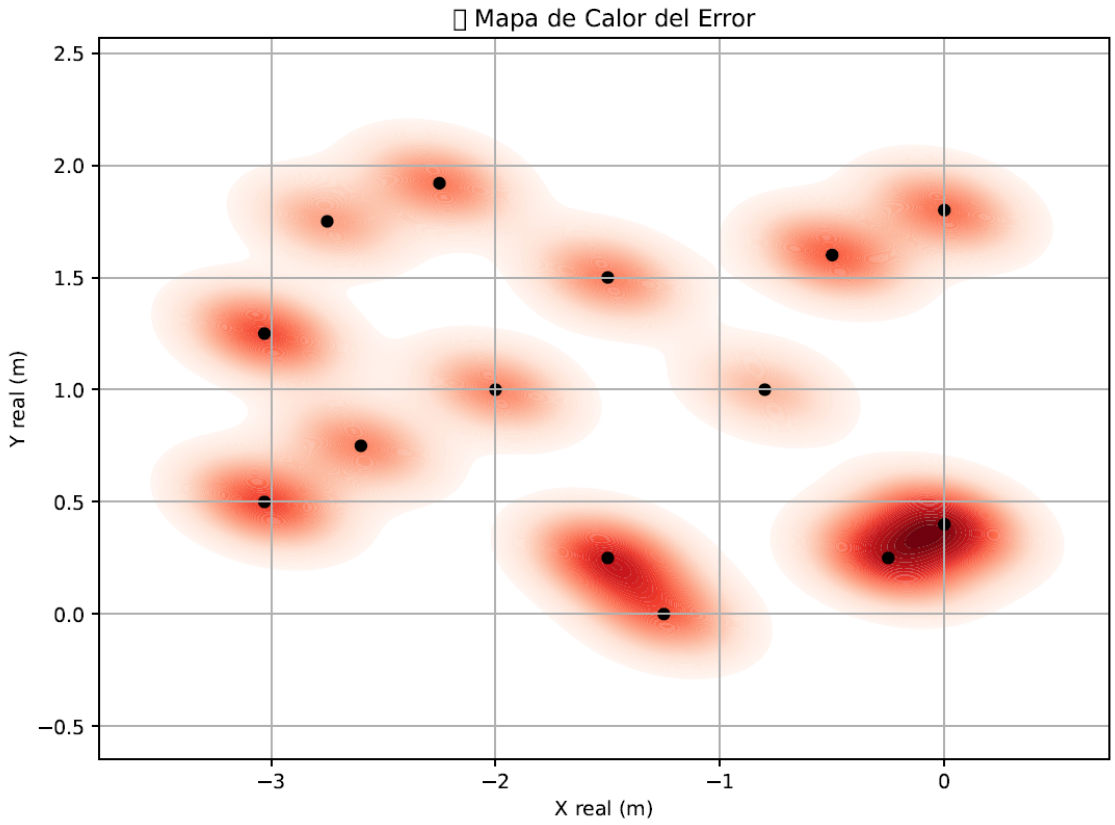


Figura 4.18 Mapa de calor del error.

4.3 Tercera Parte (Navegación autónoma)

El sistema de navegación autónoma es el corazón de este proyecto. El ensayo que se muestra en esta sección es la versión definitiva y funcional. Sin embargo, este es el resultado de múltiples ensayos anteriores y versiones menos desarrolladas y configuradas. Para alcanzar el funcionamiento deseado, se han probado infinidad de parámetros y alternativas mediante ensayo y experimentación adaptadas a la realidad del entorno de trabajo.

El complejo funcionamiento de este sistema fue detallado en el desarrollo de su código informático. Aun así, en el siguiente vídeo, se detalla y se hace énfasis en los principales movimientos más conflictivos y difíciles.

Ver video en YouTube

Además, se incluye un código QR para facilitar el acceso:

Figura 4.19 Código QR odometría interna.

Como paso previo a la visualización de gráficas, a modo de pequeño resumen, se procede a comentar y detallar el procedimiento ejecutado en el ensayo.

En primer lugar, una vez que se ejecuta el programa y se introducen las coordenadas de destino (en este caso (-2, 1.1)), el robot calcula la distancia entre el punto inicial y el definido por el usuario. Acto seguido, antes de comenzar el movimiento, realiza la orientación inicial. El Create 3 rota sucesivamente hasta que la diferencia angular entre la orientación actual y la objetiva sea menor que la tolerancia impuesta, es decir, 30 grados.

En segundo lugar, una vez orientada la roomba, comienza a desplazarse hacia el punto objetivo realizando correcciones de trayectoria, ya que el programa compara constantemente las lecturas de posición y orientación para que nunca se pierda el rumbo fijado. Durante este movimiento, se realizan lecturas de los sensores IR, para de esta manera, en función de la proximidad de diversos obstáculos, reducir preventivamente la velocidad. **Es muy importante observar que las lecturas de los sensores son un tanto inexactas e irregulares; entre lecturas, las reducciones de velocidad varían con prácticamente la misma proximidad al obstáculo.** Esto se debe a los límites impuestos en los parámetros actuales, los cuales se encuentran a merced de estas variaciones puntuales de los sensores. Aún así, en la realidad estos cambios de velocidad son prácticamente imperceptibles debido a los suaves cambios que se producen en las distintas lecturas.

En tercer lugar, el robot se mantiene a velocidad notablemente reducida a medida que se aproxima al obstáculo; se recuerda que en todo momento continúa realizando constantes correcciones. En el momento en el que los sensores frontales superan valores mayores a 55, el robot inicia la maniobra evasiva. Si al finalizar esta maniobra, el valor del sensor continúa siendo elevado, continúa evadiendo hasta que se suavicen las lecturas de los sensores IR y, por tanto, el riesgo haya disminuido.

En último lugar, con el obstáculo completamente superado tras las numerosas evasiones, se realiza la maniobra de reorientación debido a que, tras las maniobras evasivas, el robot ya no se encuentra orientado como se consiguió con éxito en la alineación inicial. Posteriormente, una vez reorientada, la roomba entra en zona libre, sin ningún obstáculo cercano, por lo que se aumenta la velocidad debido a los reducidos valores de los sensores. Como finalización del ensayo, una vez que la posición al punto objetivo es menor a la tolerancia de posición (0.1 metros), se da por finalizada exitosamente la maniobra, ya que se encuentra excesivamente próxima al punto definido. Por ello, se detiene la navegación y de la misma manera el robot.

En las siguientes páginas, se adjuntan los gráficos de la ejecución del programa:

1. Cambios Angulares

La gráfica de cambios angulares revela un comportamiento característico del sistema de navegación implementado. En el primer paso temporal se observa un pico extremadamente pronunciado que alcanza aproximadamente 140 grados, correspondiente a la fase de orientación inicial del robot. Este evento singular representa el momento en que el algoritmo ejecuta la rutina de alineación previa al movimiento, donde el robot debe girar considerablemente para orientarse hacia el objetivo establecido por el usuario.

Tras esta corrección inicial masiva, el comportamiento angular se estabiliza notablemente. Durante los pasos temporales 2 al 58, los cambios angulares se mantienen consistentemente por debajo de los 20 grados, con la mayoría de valores oscilando entre 0 y 10 grados. Esta estabilización indica que el sistema de control proporcional implementado (con constante k=0.6) funciona eficazmente para mantener la trayectoria hacia el objetivo con correcciones angulares menores.

Se pueden identificar algunos picos moderados alrededor de los pasos 25-30, que coinciden temporalmente con las maniobras evasivas detectadas en otras gráficas. Estos incrementos angulares reflejan las correcciones necesarias cuando el robot debe desviarse de su rumbo directo para esquivar obstáculos, seguido por la reorientación hacia el objetivo una vez superado el obstáculo.

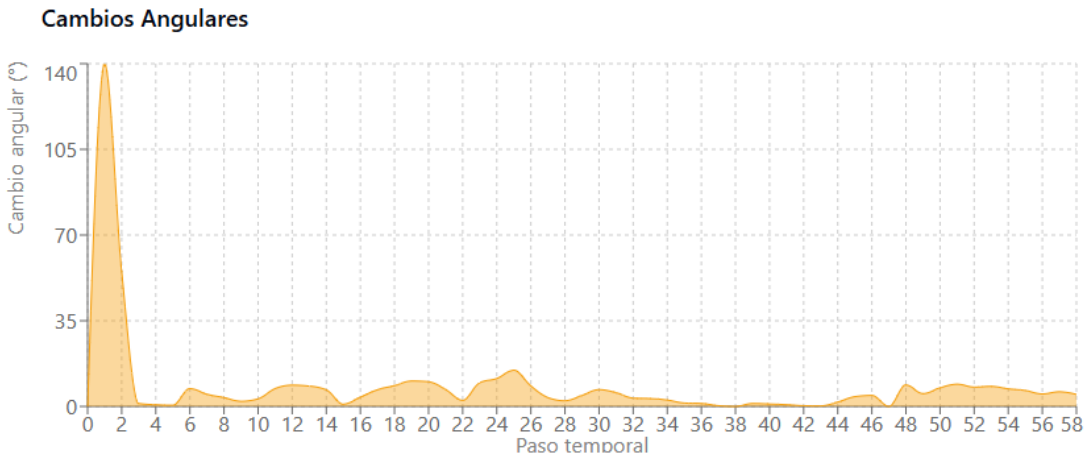


Figura 4.20 Cambios Angulares (Navegación autónoma).

2. Distancia al Origen

La evolución de la distancia al origen muestra un patrón complejo que refleja la trayectoria no lineal seguida por el robot. Inicialmente, la distancia se mantiene relativamente baja (0.65-1.0 metros) durante los primeros 12 pasos temporales, sugiriendo que el robot se desplaza en las proximidades de su punto de partida mientras se orienta y comienza su navegación.

Entre los pasos 13 y 25, se observa un incremento progresivo hasta alcanzar un máximo de aproximadamente 1.95 metros. Este comportamiento indica que el robot se aleja del origen siguiendo una trayectoria que inicialmente lo conduce en dirección opuesta o tangencial a su punto de partida, debido a maniobras evasivas tempranas. Particularmente notable es la disminución pronunciada entre los pasos 25 y 35, donde la distancia desciende de 1.95 a aproximadamente 1.3 metros. Este patrón coincide con el período de mayor actividad evasiva mostrado en otras gráficas, sugiriendo que las maniobras de esquiva han resultado en un acercamiento relativo al origen.

El tramo final (pasos 35-58) muestra un incremento gradual pero sostenido hasta alcanzar valores cercanos a 2.4 metros, indicando que el robot se dirige definitivamente hacia su destino final, alejándose progresivamente de su punto de partida.

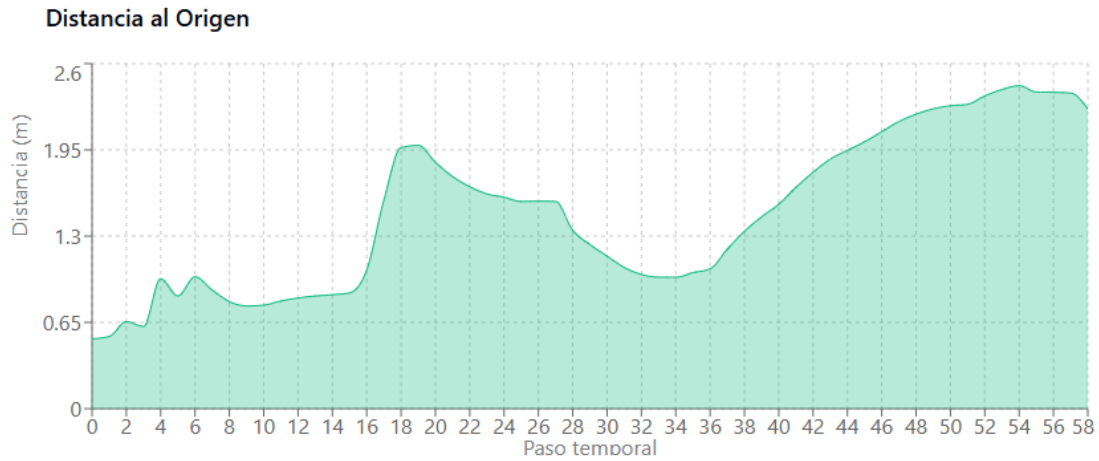


Figura 4.21 Distancia al Origen (Navegación autónoma).

3. Evolución Angular

La gráfica de evolución angular proporciona una visión clara de la orientación del robot a lo largo de toda la misión. El robot inicia con una orientación de aproximadamente 315 grados, experimentando una caída abrupta inicial hasta 180 grados alrededor del paso 2, que corresponde al ajuste de orientación inicial implementado en el algoritmo.

Durante los pasos 3 al 22, el ángulo evoluciona gradualmente desde 120 hasta 220 grados, mostrando un incremento progresivo y relativamente suave. Este comportamiento indica que el robot mantiene una orientación general hacia el sureste-suroeste mientras navega, ajustando su rumbo de manera continua según los cálculos del ángulo objetivo hacia el destino. El período más dinámico se presenta entre los pasos 22 y 30, donde se observan fluctuaciones más pronunciadas con un pico máximo alrededor de 220 grados seguido de un descenso. Estas variaciones corresponden temporalmente con las maniobras evasivas, confirmando que el robot modifica significativamente su orientación durante los esquives de obstáculos.

En la fase final (pasos 30-58), se presenta una tendencia descendente consistente desde 180 grados hasta aproximadamente 40 grados. Esta progresión angular indica que el robot se reorienta gradualmente hacia el noreste, sugiriendo que el objetivo final se encuentra en esa dirección relativa desde la posición del robot en esa fase de la navegación.

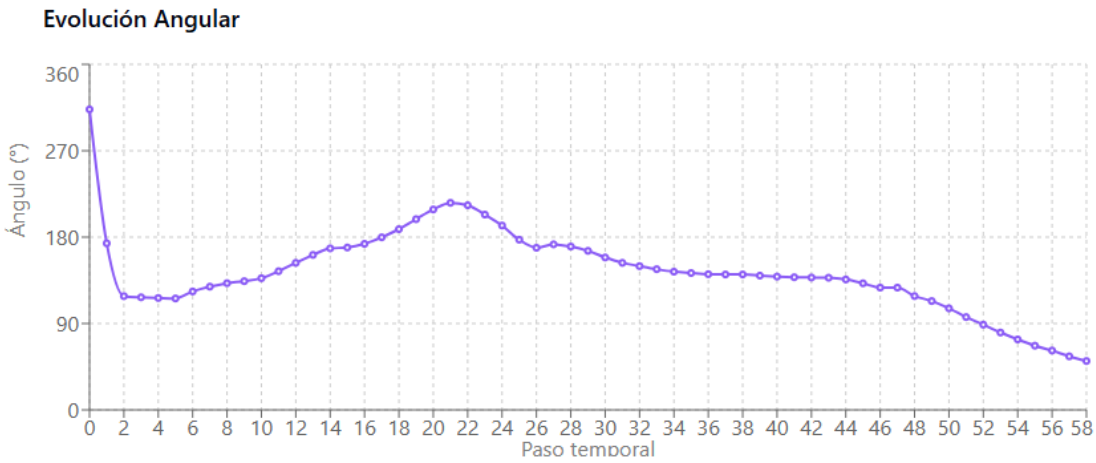


Figura 4.22 Evolución Angular (Navegación autónoma).

4. Detección de Maniobras Evasivas

La representación espacial de las maniobras evasivas muestra 8 puntos discretos distribuidos a lo largo de la trayectoria del robot, proporcionando información valiosa sobre la distribución y naturaleza de los obstáculos encontrados. Los puntos se concentran principalmente en dos regiones del espacio de navegación: una agrupación en el cuadrante superior izquierdo (coordenadas aproximadas X: -1.8 a -1.5, Y: 0.4 a 0.8) y otra en el cuadrante superior derecho (X: -0.8 a -0.4, Y: 0.3 a 0.9).

La distribución espacial sugiere que el robot encontró múltiples obstáculos durante dos fases principales de su navegación. La primera agrupación indica una zona con alta densidad de obstáculos o un obstáculo complejo que requirió múltiples maniobras evasivas consecutivas. La segunda agrupación, más dispersa, sugiere encuentros con obstáculos individuales durante la aproximación final al objetivo.

La ausencia de maniobras evasivas en ciertas regiones del mapa (particularmente en el cuadrante inferior y en las zonas centrales) indica corredores relativamente libres de obstáculos que el robot pudo atravesar sin necesidad de esquives significativos.

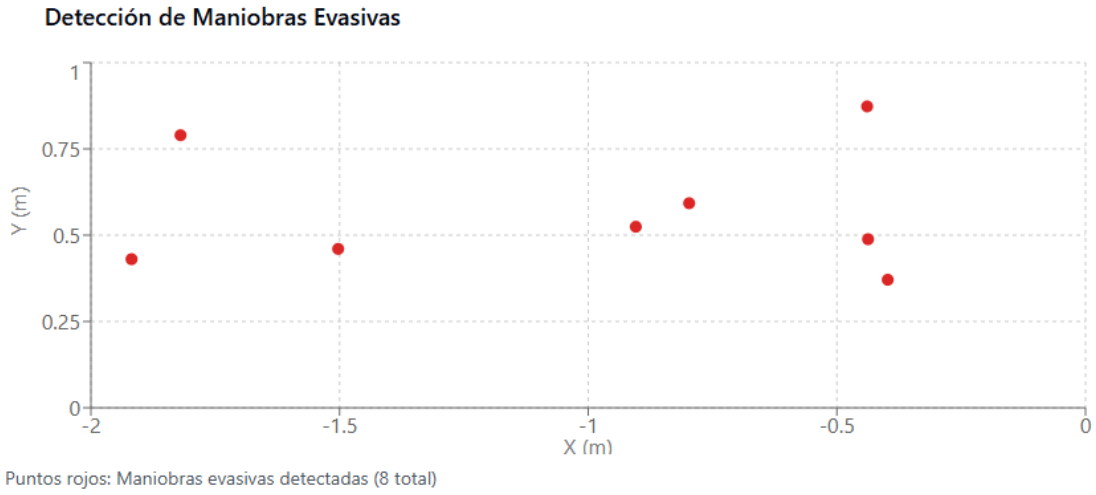


Figura 4.23 Detección de Maniobras Evasivas (Navegación autónoma).

5. Severidad de Maniobras

El análisis de severidad revela patrones temporales específicos en la intensidad de las maniobras evasivas. Los primeros pasos temporales (0-10) muestran los valores más altos de severidad, con picos que alcanzan hasta 140 en el índice de severidad. Este comportamiento inicial elevado refleja tanto la fase de orientación inicial como los primeros encuentros con obstáculos, cuando el sistema aún está ajustando su comportamiento de navegación.

Un segundo período significativo de severidad se presenta alrededor de los pasos 15-25, con valores moderados a altos (20-70 en el índice). Esta fase coincide con la zona de mayor densidad de maniobras evasivas identificada en la gráfica espacial, confirmando que el robot atravesó una región particularmente desafiante en términos de obstáculos.

La fase final de navegación (pasos 25-58) muestra una severidad considerablemente reducida, con valores mayoritariamente por debajo de 20. Esta disminución indica que el robot alcanzó zonas menos congestionadas y que su sistema de navegación había optimizado su comportamiento, requiriendo maniobras menos dramáticas para evitar obstáculos menores.

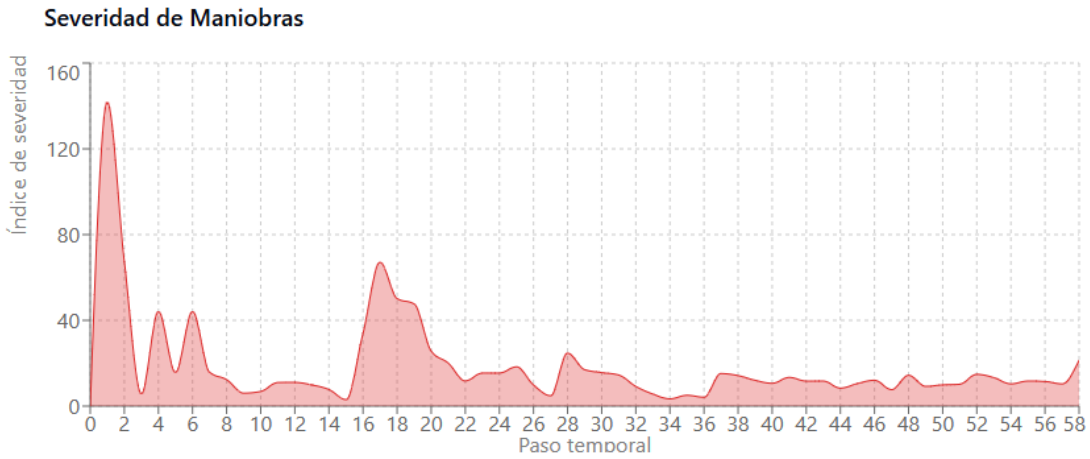


Figura 4.24 Severidad de Maniobras (Navegación autónoma).

6. Trayectoria Completa

La representación de la trayectoria completa revela un patrón de navegación complejo y realista para un entorno con obstáculos. El robot inicia en las coordenadas (-0.380, 0.364) y finaliza en (-2.004, 1.041), describiendo una trayectoria que se extiende aproximadamente 1.6 metros en dirección oeste y 0.7 metros en dirección norte.

La trayectoria no sigue una línea recta, sino que presenta múltiples desviaciones y curvaturas que reflejan las decisiones de navegación del algoritmo. Se pueden identificar claramente tres segmentos principales: una fase inicial con movimientos relativamente directos hacia el oeste, una fase intermedia con mayor tortuosidad y cambios de dirección (correspondiente a la zona de alta actividad evasiva), y una fase final con movimientos más directos hacia el objetivo.

La densidad de puntos varía a lo largo de la trayectoria, siendo mayor en las zonas de maniobras complejas, lo que indica velocidades reducidas durante la navegación en áreas congestionadas. La distribución espacial confirma que el algoritmo logró mantener un progreso general hacia el objetivo mientras gestionaba efectivamente los obstáculos encontrados.

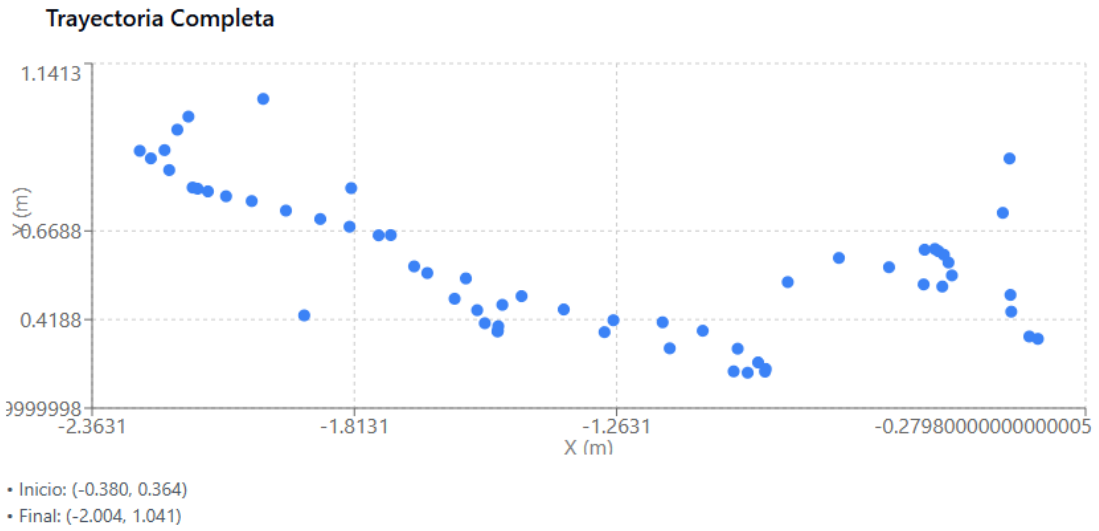


Figura 4.25 Trayectoria Completa (Navegación autónoma).

7. Velocidad Estimada

La gráfica de velocidad estimada (muestreada a 5 Hz) proporciona información detallada sobre la dinámica del movimiento del robot a lo largo de toda la misión. Los primeros pasos temporales muestran velocidades variables entre 0.5 y 2.2 m/s, reflejando la fase de aceleración inicial y los primeros ajustes de velocidad en respuesta a la detección de obstáculos.

El pico más pronunciado se presenta alrededor del paso 17, alcanzando aproximadamente 3.0 m/s, que representa la velocidad máxima registrada durante toda la misión. Este pico coincide temporalmente con una fase de navegación en espacio relativamente libre, antes del período de mayor actividad evasiva. Las fases de mayor actividad evasiva (pasos 15-30) muestran variaciones significativas en la velocidad, con múltiples aceleraciones y desaceleraciones que reflejan el comportamiento adaptativo del algoritmo de control de velocidad implementado. Los valores oscilan entre prácticamente cero (durante maniobras de precisión) y valores moderados de 1.0-1.5 m/s.

La fase final de navegación (pasos 35-58) muestra un comportamiento más estable, con velocidades consistentemente moderadas (0.2-0.8 m/s) que indican una aproximación controlada al objetivo final. Las estadísticas generales revelan una velocidad promedio de 0.523 m/s, una velocidad máxima de 3.010 m/s, y una distancia total recorrida de 6.18 metros, valores que confirman un comportamiento de navegación eficiente considerando la complejidad del entorno y los obstáculos encontrados.

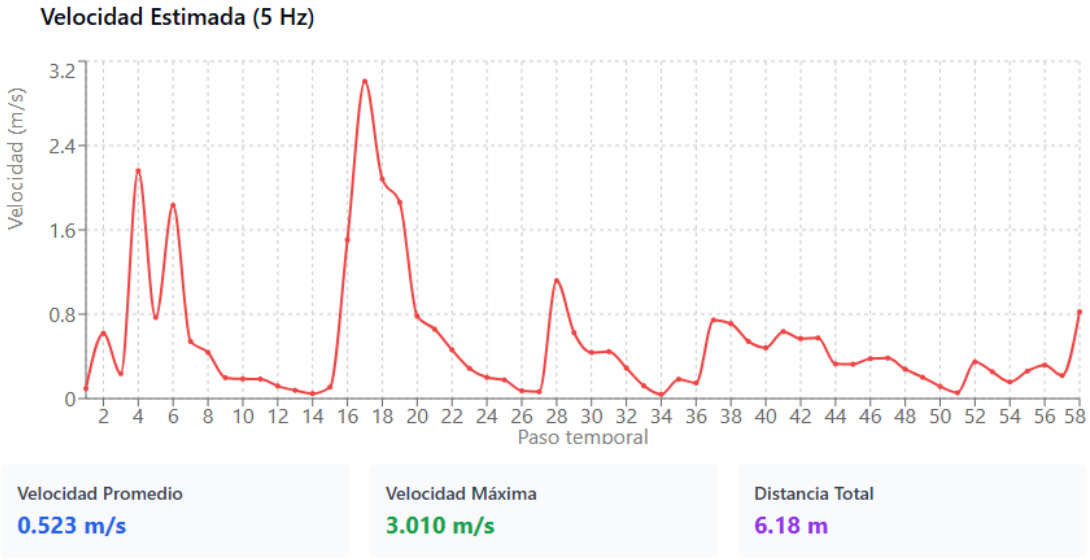


Figura 4.26 Velocidad Estimada (Navegación autónoma).

5 Conclusiones

5.1 Consideraciones finales

El desarrollo realizado en este proyecto ha permitido componer y validar una plataforma experimental completa para la navegación autónoma sobre la base del iRobot Create 3, integrando diversos subsistemas de control, localización y percepción. A lo largo de las distintas fases se ha comprobado empíricamente que, aunque la odometría provee una actualización continua y de alta frecuencia, su fiabilidad es insuficiente para el control preciso que exige la plataforma: las lecturas mostraron desviaciones no tolerables en escenarios reales, lo que condujo a su desestimación como referencia primaria en favor de un sistema de posicionamiento absoluto. Esta decisión está documentada en los resultados donde se compara la odometría frente a la referencia externa, que expone claramente la acumulación rápida de error en los encoders.

Como contrapeso a la odometría, la incorporación del sistema Marvelmind supuso una mejora significativa en la exactitud absoluta de la localización dentro del entorno acotado de trabajo. La caracterización experimental de Marvelmind, realizada mediante un muestreo sistemático en puntos anclados del rectángulo de ensayo (toma de 10 muestras por punto y análisis estadístico), permitió identificar tanto la precisión media del sistema como las zonas del entorno donde la medición se ve afectada por fenómenos físicos (ecos y reverberaciones). Estos ensayos ofrecieron información práctica imprescindible para interpretar las variaciones observadas durante la navegación autónoma y justificar la necesidad de técnicas de filtrado.

Ante la presencia de ruido y oscilaciones puntuales en las lecturas de posición, la implementación de un filtro de Kalman bidimensional demostró ser una solución eficaz para estabilizar la estimación de la posición utilizada por el bucle de control. El filtro actúa combinando la predicción del modelo de movimiento con las medidas externas, reduciendo la sensibilidad del controlador a errores puntuales y mejorando la continuidad de las consignas de velocidad y orientación. Su integración en el bucle de navegación permitió que las decisiones de dirección y las maniobras evasivas se ejecutaran sobre una referencia más coherente, lo que se tradujo en un comportamiento global más robusto frente a perturbaciones en las lecturas.

El módulo de navegación K6, fruto de múltiples iteraciones, incorporó además estrategias reactivas de evasión, reorientación post-maniobra y mecanismos de seguridad (frenadas y detección de estancamiento). Durante las pruebas, estos subsistemas mostraron buen comportamiento en condiciones controladas; sin embargo, la fase de alineación inicial y las reorientaciones posteriores a evasiones resultaron especialmente sensibles a imprecisiones angulares derivadas del sistema de localización externo, lo que revela una línea clara de trabajo futuro en la mejora de la estimación angular y en la robustez de las rutinas de reorientación.

Por último, la exploración de la percepción visual mediante YOLOv8n integrada sobre streaming RTSP supuso una experiencia reveladora. A nivel conceptual, la adición de visión ofrece capacidades semánticas que podrían mejorar la toma de decisiones; en la práctica, y dadas las limitaciones del hardware disponible, la ejecución concurrente del modelo de detección introdujo una carga de CPU y latencias que terminaron por interferir de forma significativa con el bucle de control del robot. Tras varios ensayos de mitigación (procesar 1 de cada N frames, reducir el renderizado de anotaciones y eliminar esperas bloqueantes), la integración en tiempo real no alcanzó la estabilidad exigida y fue suspendida temporalmente para priorizar la fiabilidad de la navegación. Estos hechos están documentados en las pruebas de rendimiento y en el balance final del experimento.

5.2 Posibles mejoras futuras

De cara a seguir mejorando el sistema, una de las primeras limitaciones que debería abordarse es la estimación del ángulo del robot. Esto afecta directamente a la alineación inicial y a la reorientación después de esquivar obstáculos. Una forma sencilla de mejorar este aspecto sería aprovechar el sensor inercial (IMU) integrado en el robot, ya que sus medidas de giro y aceleración permitirían complementar las posiciones dadas por Marvelmind, ofreciendo así una referencia más estable.

También convendría trabajar sobre la propia instalación del sistema Marvelmind. La disposición de las balizas influye mucho en la calidad de la localización, por lo que revisarlas para que no queden alineadas y procurar que tengan un ángulo de visión amplio sería un buen paso. Además, realizar calibraciones periódicas o incluso compensar automáticamente la temperatura ambiente ayudaría a que las mediciones fueran más estables y menos sensibles a cambios externos.

Desde el punto de vista del software, la experiencia con la integración de visión artificial dejó claro que las tareas que consumen muchos recursos no deben mezclarse directamente con el control del robot en tiempo real. Una posible solución sería dividir los procesos en módulos separados o incluso usar un framework como ROS 2, que permite organizar la navegación, los sensores y el control en nodos independientes que se comunican entre sí. Otra vía sería descargar el procesamiento de visión a hardware especializado, como una GPU o una placa tipo Jetson, que se encargaría de detectar objetos y devolver solo la información necesaria al robot.

En cuanto a los algoritmos, el propio filtro de Kalman podría mejorarse haciendo que se ajuste automáticamente en función de la calidad de la señal recibida. De esta manera, cuando Marvelmind entregue lecturas menos fiables, el sistema confiaría menos en ellas y evitaría errores bruscos. También podrían aplicarse filtros sencillos para eliminar lecturas extrañas que a veces contaminan las estimaciones. Y en el futuro, sería interesante combinar la planificación reactiva actual con una planificación más global basada en mapas, lo que haría que las trayectorias fueran más suaves y eficientes.

Por último, otro punto importante es la validación. Sería recomendable repetir los experimentos en condiciones controladas y registrar más métricas, como errores medios de posición o tiempos de llegada al objetivo, para tener una base sólida de comparación. Esto no solo permitiría evaluar mejor el rendimiento del sistema, sino también generar datos que podrían aprovecharse en fases futuras para entrenar o ajustar nuevos módulos de percepción.

En definitiva, las mejoras a corto plazo deberían centrarse en consolidar la estimación de posición y la robustez del control (fusión con IMU, calibración de balizas y filtrado adaptativo). Una vez alcanzada esa estabilidad, el sistema estaría preparado para integrar percepciones más avanzadas, como mapas mediante SLAM o detección de objetos en tiempo real, que lo llevarían a un nivel mucho más completo.

6 Anexos

6.1 roomba_pos.py

```python
#IMPORTACIONES
import pygame
# Importa la librería para interactuar con el mando de PS4 (joysticks, botones, etc.)
import asyncio
# Permite ejecutar funciones de manera asíncrona (simultáneamente sin bloqueo)
import time
# Utilizado para pausar la ejecución durante un tiempo determinado (como sleep)
import math
# Proporciona funciones matemáticas como trigonometría y conversión de ángulos
import sys
# Permite interactuar con el sistema, por ejemplo, para cerrar el programa
import json
# Módulo para leer y escribir archivos en formato JSON (estructura clave-valor)

from irobot_edu_sdk.backend.bluetooth import Bluetooth
# Importa el backend Bluetooth necesario para comunicarse con la Roomba Create 3
from irobot_edu_sdk.robots import event, Create3
# Importa el modelo de robot Create3 y los decoradores de eventos del SDK
from irobot_edu_sdk.music import Note
# Permite tocar tonos musicales usando el buzzer integrado en la Roomba


#GUARDAR POSICIÓN ROOMBA
def guardar_posicion_roomba(x, y, heading):
# Función encargada de guardar la posición actual de la Roomba en un archivo JSON compartido.Este archivo actúa como medio de comunicación entre diferentes scripts del sistema.
    try:
    # Intenta abrir el archivo existente 'data.json' en modo lectura
        with open("data.json", "r") as f:
            data = json.load(f)
            # Carga el contenido del archivo como diccionario
    except (FileNotFoundError, json.JSONDecodeError):
    # Si el archivo no existe o está mal formado (vacío o con error de sintaxis JSON)
        data = {}
        # Se crea un diccionario vacío como base

    data["roomba"] = {"x": x, "y": y, "heading": heading}
    # Se actualiza el campo "roomba" con los valores de posición y orientación actuales

    with open("data.json", "w") as f:
    # Se vuelve a abrir el archivo 'data.json', esta vez en modo escritura
        json.dump(data, f, indent=4)
        # Se vuelca todo el contenido del diccionario actualizado al archivo, con formato legible


#INICIALIZA EL BLUETOOTH Y EL ROBOT
backend = Bluetooth()
# Se crea una instancia del backend Bluetooth.Es el canal de comunicación entre el ordenador y la Roomba Create 3.
robot = Create3(backend)
# Se crea el objeto principal 'robot', que representa a la Roomba Create 3.
```

(continuación en la siguiente página)

(continuación del código anterior)

```
#VARIABLES GLOBALES
turbo_mode = False
# Variable global que indica si el modo turbo (más velocidad) está activo.
reverse_mode = False
# Variable global para activar o desactivar el modo marcha atrás.
drift_mode = False
# Variable global para activar el modo drift, que modifica la intensidad de giro para maniobras más cerradas.
program_running = True
# Variable de control general que permite iniciar o detener el bucle principal del programa.
position_task = None
# Esta variable almacenará la tarea asincrona que lee la posición de la Roomba.


#INICIALIZACION
@event(robot.when_play)
# Indica que esta función se ejecuta automáticamente al iniciar el programa.
async def inicializacion(robot):
    pygame.init()
    # Inicializa el módulo pygame, necesario para leer el estado del mando PS4.
    pygame.joystick.init()

    if pygame.joystick.get_count() == 0:
    # Se comprueba si hay al menos un mando de PS4 conectado.
        print(" No se detectó ningún mando de PS4.")
        pygame.quit()
        # Si no se detecta ningún mando, se notifica al usuario y se cierra correctamente el entorno pygame.
        exit()
    else:
        print(" Mando detectado:", pygame.joystick.Joystick(0).get_name())
        # Si se detecta al menos un mando, se imprime su nombre por pantalla como confirmación.

    await robot.play_tone(Note.E4, Note.EIGHTH)
    # Reproduce un tono de encendido usando el buzzer de la Roomba como señal auditiva de que está lista.
    await robot.play_tone(Note.G4, Note.QUARTER)
    await robot.play_tone(Note.C5, Note.HALF)


#LECTURA Y GUARDADO PERIÓDICO DE LA POSICIÓN
async def get_roomba_position():
# Su objetivo es leer continuamente la posición estimada del robot y guardarla.
    while program_running:
    # El bucle se ejecuta mientras el programa esté activo.
        try:
        # Solicita al robot su posición actual (estructura con x, y, heading)
            pos = await robot.get_position()
            x, y, heading_rad = pos.x, pos.y, pos.heading
            # Extrae las coordenadas x, y (en metros) y la orientación (en radianes).
            heading_deg = math.degrees(heading_rad) % 360
            # Convierte el ángulo de orientación de radianes a grados y lo normaliza en el rango [0, 360).

            print(f"Posición Roomba: X={x:.2f}m, Y={y:.2f}m, Heading={heading_deg:.2f}°")
            # Muestra la posición actual por consola con formato legible (2 decimales).
            guardar_posicion_roomba(x, y, heading_deg)
            # Guarda la posición actual de la Roomba en el archivo 'data.json'.

        except Exception as e:
        # Si hay un error al obtener la posición (por ejemplo, si se pierde la conexión), se captura y se
            muestra.
            print(f" Error al obtener posición: {e}")

        await asyncio.sleep(0.5)
        # Espera 0.5 segundos antes de volver a hacer una nueva lectura de posición.


#CONVERSIÓN DE ENTRADAS DEL MANDO DE VELOCIDADES
def interpretar_controles(volante: float, acelerador: float, freno: float) -> tuple[float, float]:
# Convierte los controles del mando PS4 en velocidades de las ruedas
    global turbo_mode, reverse_mode, drift_mode
    # Se usan las variables globales para ajustar el comportamiento del robot según los modos activos

    MAX_VELOCIDAD = 40 if turbo_mode else 20
    # Define la velocidad máxima según si el modo turbo está activado o no
    GIRO_INTENSIDAD_MOVIMIENTO = 1.5 if drift_mode else 0.8
    # Modifica la intensidad del giro cuando el robot se está desplazando (modo drift)
    GIRO_INTENSIDAD_NORMAL = 0.6
    # Define la intensidad del giro cuando el robot está parado (gira sobre su eje)
    DEAD_ZONE = 0.1
    # Define la zona muerta. Si el joystick se mueve menos que este valor, se ignora el movimiento
    VELOCIDAD_GIRO = 10
    # Define una velocidad base para girar sobre el eje sin avance (cuando velocidad_base = 0)
```

(continuación en la siguiente página)

(continuación del código anterior)

```
    if abs(volante) < DEAD_ZONE:
    # Define una velocidad base para girar sobre el eje sin avance (cuando velocidad_base = 0)
        volante = 0
    if abs(acelerador) < DEAD_ZONE:
    # Si el acelerador no se ha movido suficientemente, lo ignoramos
        acelerador = 0
    if abs(freno) < DEAD_ZONE:
    # Si el freno no se ha movido suficientemente, lo ignoramos
        freno = 0

    if reverse_mode:
    # Si está activado el modo marcha atrás, intercambiamos el acelerador y el freno
        acelerador, freno = freno, acelerador

    velocidad_base = (acelerador - freno) * MAX_VELOCIDAD
    # Calculamos la velocidad base en función del acelerador y el freno
    velocidad_base = max(-MAX_VELOCIDAD, min(MAX_VELOCIDAD, velocidad_base))
    # Asegura que la velocidad no supere los límites permitidos

    factor_giro = volante * (GIRO_INTENSIDAD_MOVIMIENTO if velocidad_base != 0 else GIRO_INTENSIDAD_NORMAL)
    # Calcula un factor de giro que se aplica sobre la velocidad base.Si el robot está en movimiento, se usa la
          intensidad de giro de movimiento; si está parado, la normal.

    if velocidad_base == 0 and volante != 0:
    # Si el robot está parado pero hay giro, realiza un giro sobre su eje.
        velocidad_left = VELOCIDAD_GIRO * volante
        velocidad_right = -VELOCIDAD_GIRO * volante
    else:
        velocidad_left = velocidad_base * (1 + factor_giro)
        # Si el robot se está moviendo, aplicamos el giro ajustando las velocidades de ambas ruedas
        velocidad_right = velocidad_base * (1 - factor_giro)

    return velocidad_left, velocidad_right


#CONTROL PRINCIPAL DE LA ROOMBA CON EL MANDO
@event(robot.when_play)
async def manual_control(robot):
# Controla la Roomba en tiempo real usando el mando PS4, y lanza en paralelo la lectura de posición.
    global turbo_mode, reverse_mode, drift_mode, program_running, position_task
    # Declaramos como globales las variables que pueden cambiar desde esta función
    joystick = pygame.joystick.Joystick(0)
    # Se obtiene el primer joystick disponible (el mando de PS4)
    joystick.init()

    position_task = asyncio.create_task(get_roomba_position())
    # Se lanza la tarea asincrona que lee la posición de la Roomba de forma periódica

    try:
        while program_running:
        # Bucle principal de control, se ejecuta continuamente mientras el programa esté activo
            pygame.event.pump()
            # Actualiza el estado del mando. Sin esto, no se leerían los eventos nuevos

            eje_x = joystick.get_axis(0)
            # Lee el eje horizontal del joystick izquierdo (volante)
            acelerador = (joystick.get_axis(5) + 1) / 2
            # Lee el gatillo derecho (R2) acelerador. Se escala de [-1,1] a [0,1]
            freno = (joystick.get_axis(4) + 1) / 2
            # Lee el gatillo izquierdo (L2) freno. También se escala a [0,1]

            turbo_mode = joystick.get_button(1)
            # Botón circulo (Botón 1): activa el modo turbo (más velocidad)
            drift_mode = joystick.get_button(9)
            # Botón L1 (Botón 9): activa el modo drift (giros más cerrados)

            if joystick.get_button(0):
            # Botón X (Botón 0): freno de emergencia. Detiene el robot completamente.
                print(" Freno de emergencia activado!")
                await robot.set_wheel_speeds(0, 0)
                await asyncio.sleep(0.5)

            if joystick.get_button(2):
            # Botón cuadrado (Botón 2): alterna el modo marcha atrás
                reverse_mode = not reverse_mode
                print(f" Modo marcha atrás:", "Activado" if reverse_mode else "Desactivado")
                await asyncio.sleep(0.3)
                # Pequeña pausa para evitar múltiples lecturas por rebote
```

(continuación en la siguiente página)

(continuación del código anterior)

```
            if joystick.get_button(3):
            # Botón triángulo (Botón 3): claxon. Emite un tono agudo.
                print(" Claxon!")
                await robot.play_tone(Note.G5, Note.QUARTER)

            if joystick.get_button(10):
            # Botón R1 (Botón 10): giro instantáneo de 180 grados sobre el eje
                print(" Giro 180° activado!")
                await robot.set_wheel_speeds(30, -30)
                # Una rueda hacia delante, la otra hacia atrás
                await asyncio.sleep(0.7)
                await robot.set_wheel_speeds(0, 0)
                # Detener al terminar el giro

            if joystick.get_button(9):
            # Botón L1 (reutilizado): alterna el modo turbo también desde aquí
                turbo_mode = not turbo_mode
                print(f" Turbo: {'Activado' if turbo_mode else 'Desactivado'}")
                await asyncio.sleep(0.3)

            if joystick.get_button(6):
            # Botón OPTIONS (Botón 6): detener completamente el programa y cerrar
                print(" Deteniendo la Roomba y cerrando el programa")
                program_running = False
                await robot.set_wheel_speeds(0, 0)
                # Detiene el robot
                await asyncio.sleep(1)

                if position_task:
                # Cancela la tarea de lectura de posición si estaba activa
                    position_task.cancel()
                    try:
                        await position_task
                    except asyncio.CancelledError:
                        pass

                pygame.quit()
                # Cierra el entorno pygame correctamente

                loop = asyncio.get_running_loop()
                # Cierra todas las tareas pendientes de asyncio para evitar errores de cierre
                for task in asyncio.all_tasks(loop):
                    task.cancel()
                    try:
                        await task
                    except asyncio.CancelledError:
                        pass
                loop.stop()
                print(" Programa cerrado con éxito.")
                return
                # Sale completamente de la función

            velocidad_izquierda, velocidad_derecha = interpretar_controles(eje_x, acelerador, freno)
            # Si no se ha pulsado ninguna tecla especial, interpreta los controles normales
            await robot.set_wheel_speeds(velocidad_izquierda, velocidad_derecha)
            # Aplica las velocidades a las ruedas de la Roomba
            await asyncio.sleep(0.1)
            # Espera un poco antes de la siguiente lectura, para evitar sobrecarga

    except asyncio.CancelledError:
    # En caso de que la tarea sea cancelada por el sistema (por ejemplo al cerrar), se ignora la excepción
        pass

#LANZADOR PRINCIPAL DEL PROGRAMA
try:
# Intenta ejecutar el método 'play' del objeto robot, que inicia todos los eventos decorados con @event
    robot.play()
except asyncio.exceptions.CancelledError:
# Si el programa se cancela correctamente desde dentro (por ejemplo, pulsando OPTIONS), se captura la excepción
      y no se hace nada más. Es un cierre limpio esperado.
    print("Programa finalizado correctamente sin errores.")
except Exception as e:
# Captura cualquier otra excepción inesperada que pueda haber ocurrido durante la ejecución
    if "Event loop stopped before Future completed" not in str(e):
    # Si se trata de un error diferente, lo imprime para que el usuario lo sepa
        print(f" Se detectó un error inesperado: {e}")
```

Código 6.1 roomba_pos.py.

6.2 marvelmind_process.py

```python
#IMPORTACIONES
import json
# Importa el módulo 'json' para leer y escribir en el archivo de datos compartido 'data.json'
import time
# Importa 'time' para poder pausar la ejecución temporalmente dentro de un bucle

from example import MarvelmindHedge
# Este objeto se encarga de gestionar la conexión y lectura de datos desde el sistema Marvelmind


#GUARDAR POSICION MARVELMIND
def guardar_posicion_marvelmind(x, y):
# Define una función para guardar la posición recibida del sistema Marvelmind en el archivo 'data.json'
    print(f"[DEBUG] Intentando guardar en data.json: X={x}, Y={y}")
    # Imprime por consola los datos que se intentarán guardar (modo depuración)

    try:
    # Intenta abrir el archivo de datos para leer su contenido actual
        with open("data.json", "r") as f:
            data = json.load(f)
    except (FileNotFoundError, json.JSONDecodeError):
    # Si el archivo no existe o tiene un error de formato, crea un diccionario vacío
        data = {}

    data["marvelmind"] = {"x": x, "y": y}
    # Actualiza el diccionario con la nueva posición recibida del sistema Marvelmind

    with open("data.json", "w") as f:
        json.dump(data, f, indent=4)
        # Escribe el diccionario actualizado en el archivo 'data.json'

    print(f"[DEBUG] Guardado en data.json: X={x}, Y={y}")
    # Imprime confirmación de que los datos se guardaron correctamente


#INICIALIZACION DEL SISTEMA MARVELMIND
hedge = MarvelmindHedge(tty="COM3", adr=None, debug=False)
# Gestiona la lectura de posiciones desde la baliza móvil a través del puerto serie COM3.
hedge.start()
# Internamente, empieza a escuchar por el puerto serie y a recibir datos en segundo plano.
print("[INFO] Marvelmind iniciado y conectado en COM3")
# Imprime por consola que el sistema Marvelmind ha sido correctamente iniciado y está conectado.

try:
    while True:
        if hedge.positionUpdated:
        # Verifica si hay una nueva posición disponible (indicador interno del objeto 'hedge')
            pos = hedge.position()
            # Obtiene la posición actual desde el buffer del sistema Marvelmind
            x, y = pos[1], pos[2]
            # Extrae únicamente las coordenadas X e Y (en metros) de la posición

            print(f"[DEBUG] Datos recibidos de Marvelmind: X={x}, Y={y}")
            # Imprime en consola la posición recién recibida, útil para depuración

            guardar_posicion_marvelmind(x, y)
            # Llama a la función que guarda la posición en el archivo data.json

            print(f"[MARVELMIND] X: {x:.3f} m, Y: {y:.3f} m")
            # Imprime la posición en un formato más amigable para el usuario final

        time.sleep(0.1)
        # Espera 0.1 segundos antes de la siguiente iteración para evitar sobrecargar el sistema


#MANEJO DE INTERRUPCION DEL TECLADO
except KeyboardInterrupt:
# Si el usuario pulsa Ctrl+C, se lanza una excepción KeyboardInterrupt
    print("[INFO] Marvelmind detenido.")
    hedge.stop()
    # Solicita al objeto Marvelmind que detenga su hilo de ejecución y libere el puerto serie
```

Código 6.2 marvelmind_process.py.

6.3 coordinador.py

```python
#IMPORTACIONES
import json
# Importa el módulo 'json' para leer el archivo data.json, donde están las posiciones de Roomba y Marvelmind
import time
# Importa 'time' para hacer pausas entre iteraciones del bucle


#CALCULO ERROR
def calculate_error(pos1, pos2):
# Define una función que calcula la distancia euclidiana entre dos puntos en el plano XY. Esta función se usará para comparar la posición de la Roomba (odometría) con la de Marvelmind (ultrasonidos).
    if pos1 and pos2:
    # Si ambas posiciones están definidas (no son None), se realiza el cálculo
        return ((pos1["x"] - pos2["x"])**2 + (pos1["y"] - pos2["y"])**2)**0.5
    return None
    # Si alguna de las posiciones es None, se devuelve None para indicar que no se puede calcular error


#MENSAJES DE INICIO Y PREPARACIÓN DEL BUFFER
print("[INFO] Coordinador iniciado")
# Mensaje informativo al iniciar el script
print("[INFO] Esperando que ambas posiciones se sincronicen antes de calcular el margen de error")
# Mensaje que indica que el programa está esperando datos de ambas fuentes
buffer = {"roomba": [], "marvelmind": []}
# Inicializa un diccionario con dos listas vacías para almacenar las últimas posiciones de Roomba y Marvelmind


#BUCLE PRINCIPAL DE SINCRONIZACIÓN Y CÁLCULO DE ERROR
while True:
    try:
    # Intenta abrir y leer el archivo 'data.json', donde otros scripts almacenan las posiciones
        with open("data.json", "r") as f:
            data = json.load(f)
            roomba_pos = data.get("roomba")
            # Extrae la posición de la Roomba y de Marvelmind si existen
            marvelmind_pos = data.get("marvelmind")

        if roomba_pos:
        # Si hay una nueva posición de la Roomba, se añade al buffer correspondiente
            buffer["roomba"].append(roomba_pos)
        if marvelmind_pos:
        # Igual con la posición de Marvelmind
            buffer["marvelmind"].append(marvelmind_pos)

        if len(buffer["roomba"]) > 3:
        # Se mantiene en el buffer solo las últimas 3 posiciones de cada sistema para evitar acumulación
            buffer["roomba"].pop(0)
        if len(buffer["marvelmind"]) > 3:
            buffer["marvelmind"].pop(0)

        if len(buffer["roomba"]) == 3 and len(buffer["marvelmind"]) == 3:
        # Solo se realiza el cálculo si ambos sistemas tienen al menos 3 posiciones recientes
            roomba_avg = buffer["roomba"][-1]
            # Se toma la posición más reciente de cada fuente
            marvelmind_avg = buffer["marvelmind"][-1]
            error = calculate_error(roomba_avg, marvelmind_avg)
            # Se calcula el error (distancia entre ambas posiciones)
            print(f"[INFO] Margen de error: {error:.4f} metros")
            # Se muestra el resultado en consola con 4 decimales

        else:
        # Si aún no hay suficientes datos, se avisa al usuario
            print("[INFO] No hay suficientes datos sincronizados. Esperando...")

        time.sleep(1)
        # Se espera 1 segundo antes de volver a iterar

    except (FileNotFoundError, json.JSONDecodeError):
    # Si el archivo no existe o está mal formado (por ejemplo, se está escribiendo en él mientras se lee)
        print("[INFO] Archivo de datos no encontrado o corrupto. Esperando...")
        time.sleep(1)

    except KeyboardInterrupt:
    # Si el usuario interrumpe el programa con Ctrl+C, se captura la excepción y se cierra el programa ordenadamente
        print("\n[INFO] Coordinador detenido.")
        break
```

Código 6.3 coordinador.py.

6.4 launcher.py

```python
#IMPORTACIONES
import subprocess
# Importa el módulo 'subprocess' para poder lanzar scripts externos como procesos independientes
import time
# Importa el módulo 'time' para poder introducir pausas entre lanzamientos de procesos

print("[INFO] Iniciando todos los procesos...")


#LANZAMIENTO DEL PROCESO ROOMBA
roomba_proc = subprocess.Popen(["python", "roomba_pos.py"])
# Inicia el script 'roomba_pos.py' como un proceso independiente usando el intérprete de Python
time.sleep(5)
# Espera 5 segundos para asegurarse de que el script de la Roomba se ha inicializado correctamente


#LANZAMIENTO DEL PROCESO MARVELMIND
marvelmind_proc = subprocess.Popen(["python", "marvelmind_process.py"])
# Inicia el script 'marvelmind_process.py' como otro proceso independiente
time.sleep(5)
# Espera otros 5 segundos para permitir que el sistema Marvelmind se conecte y comience a emitir datos


#LANZAMIENTO DEL COORDINADOR
coordinador_proc = subprocess.Popen(["python", "coordinador.py"])
# Inicia el script 'coordinador.py', que calcula la diferencia entre ambas posiciones


#MANTENIMIENTO Y CIERRE CONTROLADO
try:
# Espera a que cualquiera de los procesos termine. Esto bloquea la ejecución principal.
    roomba_proc.wait()
    marvelmind_proc.wait()
    coordinador_proc.wait()
except KeyboardInterrupt:
# Si el usuario interrumpe con Ctrl+C (KeyboardInterrupt), se entra en este bloque
    print("\n[INFO] Cerrando todos los procesos...")
    roomba_proc.terminate()
    # Termina el proceso de la Roomba
    marvelmind_proc.terminate()
    # Termina el proceso de Marvelmind
    coordinador_proc.terminate()
    # Termina el proceso del Coordinador
    print("[INFO] Todos los procesos han sido detenidos correctamente.")
    # Confirmación final
```

Código 6.4 launcher.py.

6.5 medicion_error_mm2.py

```python
#IMPORTACIONES
import math
# Necesario para calcular la distancia euclidiana (error)
import time
# Control de pausas y sincronización
import csv
# Para guardar datos en formato CSV
import pandas as pd
# Procesamiento de tablas y exportación a Excel
import matplotlib.pyplot as plt
# Para generar gráficos
import seaborn as sns
# Para generar mapas de calor y gráficos estilizados

from datetime import datetime
# Para nombrar archivos con fecha y hora
from marvelmind import MarvelmindHedge
# Interfaz para acceder al sistema Marvelmind
from matplotlib.backends.backend_pdf import PdfPages
# Exportación de gráficos a PDF
import warnings
# Para suprimir advertencias no críticas

warnings.filterwarnings("ignore")
# Oculta advertencias durante ejecución (por ejemplo: matplotlib, seaborn)


#NOMBRES DE ARCHIVO DINÁMICO CON TIMESTAMP
now = datetime.now().strftime("%Y-%m-%d_%H-%M")
# Crea un timestamp para etiquetar los archivos de salida con la fecha y hora actual
csv_filename = f"errores_{now}.csv"
# Nombres para los distintos formatos de salida
excel_filename = f"tabla_errores_{now}.xlsx"
pdf_filename = f"reporte_errores_{now}.pdf"


#INICIALIZACIÓN DEL SISTEMA MARVELMIND
def init_hedge():
# Función para iniciar el sistema Marvelmind
    hedge = MarvelmindHedge(tty="COM3", adr=None, debug=False)
    hedge.start()
    print("Marvelmind iniciado. Esperando posición estable...")
    return hedge


#LECTURA ESTABLE DE POSICIÓN ESTIMADA
def get_stable_position(hedge, samples=10):
# Captura múltiples muestras para obtener una posición media y reducir el ruido
    posiciones = []
    while len(posiciones) < samples:
        try:
            if hedge.positionUpdated:
                pos = hedge.position()
                x, y = pos[1], pos[2]
                posiciones.append((x, y))
                print(f"Recibido: X={x:.3f}, Y={y:.3f}")
            time.sleep(0.2)
        except Exception:
            continue
            # Ignora errores puntuales del sensor

    x_avg = sum(p[0] for p in posiciones) / samples
    # Calcula el promedio de X y de Y a partir de las muestras recogidas
    y_avg = sum(p[1] for p in posiciones) / samples
    return x_avg, y_avg


#CÁLCULO DEL ERROR
def calcular_error(x_real, y_real, x_mm, y_mm):
# Cálculo de la distancia euclidiana entre la posición real y la estimada
    return math.sqrt((x_real - x_mm)**2 + (y_real - y_mm)**2)
```

(continuación en la siguiente página)

(continuación del código anterior)

```
#GUARDADO EN CSV
def guardar_en_csv(punto_id, x_real, y_real, x_mm, y_mm, error):
# Añade una nueva fila con los datos de cada punto medido al archivo CSV
    with open(csv_filename, "a", newline="") as f:
        writer = csv.writer(f)
        writer.writerow([punto_id, x_real, y_real, x_mm, y_mm, error])

with open(csv_filename, "w", newline="") as f:
# Crea el archivo CSV desde cero con cabecera al inicio del programa
    writer = csv.writer(f)
    writer.writerow(["Punto", "X_real", "Y_real", "X_mm", "Y_mm", "Error_m"])

#GENERACIÓN PDF Y EXCEL CON LOS RESULTADOS
def generar_reportes():
# Esta función genera tres tipos de gráficos y exporta los resultados a un PDF y a un Excel
    df = pd.read_csv(csv_filename)
    # Carga los datos del archivo CSV
    df.to_excel(excel_filename, index=False)
    # Exporta a Excel

    with PdfPages(pdf_filename) as pdf:

        # Gráfico de dispersión con escala de color según el error
        plt.figure(figsize=(8, 6))
        scatter = plt.scatter(df['X_real'], df['Y_real'], c=df['Error_m'], cmap='plasma', s=120)
        plt.colorbar(scatter, label='Error (m)')
        plt.title('Error de Posicionamiento por Punto Medido')
        plt.xlabel('X real (m)')
        plt.ylabel('Y real (m)')
        plt.grid(True)
        plt.axis('equal')
        plt.tight_layout()
        pdf.savefig()
        plt.close()

        # Mapa de calor de densidad (KDE) ponderado por el error
        plt.figure(figsize=(8, 6))
        sns.kdeplot(x=df['X_real'], y=df['Y_real'], weights=df['Error_m'], cmap="Reds", fill=True, bw_adjust
            =0.3, levels=100)
        plt.scatter(df['X_real'], df['Y_real'], c='black', s=30)
        plt.title('Mapa de Calor del Error')
        plt.xlabel('X real (m)')
        plt.ylabel('Y real (m)')
        plt.grid(True)
        plt.axis('equal')
        plt.tight_layout()
        pdf.savefig()
        plt.close()

        # Gráfico de barras: error por punto
        plt.figure(figsize=(10, 5))
        plt.bar(df['Punto'], df['Error_m'], color='orange', edgecolor='black')
        plt.title('Error por Punto Medido')
        plt.xlabel('Punto')
        plt.ylabel('Error (m)')
        plt.grid(axis='y')
        plt.tight_layout()
        pdf.savefig()
        plt.close()


#BUCLE PRINCIPAL DE MEDICIÓN
hedge = init_hedge()
# Inicializa el sistema Marvelmind y lanza el hilo que empieza a leer datos por el puerto COM3
punto = 1
# Variable que representa el número del punto actual a medir. Se incrementará manualmente por el usuario.

try:
    while True:
        print(f"\n Punto {punto}")
        # Imprime por consola el identificador del punto actual
        x_real = float(input("Introduce X real (m): ").replace(",", "."))
        # Solicita al usuario que introduzca la coordenada X real del punto (en metros)
        y_real = float(input("Introduce Y real (m): ").replace(",", "."))

        print("Esperando posición estable...")
        x_mm, y_mm = get_stable_position(hedge)
        # Llama a la función que toma varias muestras del sistema Marvelmind y devuelve el promedio
        print(f"Marvelmind estable: X={x_mm:.3f}, Y={y_mm:.3f}")
        # Informa que se ha alcanzado una posición estimada suficientemente estable
```

(continuación en la siguiente página)

(continuación del código anterior)

```
         error = calcular_error(x_real, y_real, x_mm, y_mm)
        # Calcula el error de posicionamiento entre la coordenada real y la medida por Marvelmind
        print(f"Error: {error:.3f} metros")
        # Muestra el valor del error por consola con 3 cifras decimales

        guardar_en_csv(punto, x_real, y_real, x_mm, y_mm, error)
        # Guarda los datos del punto medido y el error en el archivo CSV correspondiente
        punto += 1
        # Incrementa el número de punto para la siguiente medición


#FINALIZACIÓN DEL PROGRAMA
except KeyboardInterrupt:
# Si el usuario pulsa Ctrl+C durante el bucle, se lanza esta excepción
    print("\n Medición detenida por el usuario.")
    hedge.stop()
    # Detiene correctamente el hilo de lectura de Marvelmind y libera el puerto serie
    generar_reportes()
    # Genera automáticamente los reportes en formato CSV, Excel y PDF con los datos recogidos
    print(f" Reportes generados:\n - CSV: {csv_filename}\n - Excel: {excel_filename}\n - PDF: {pdf_filename}")
    # Muestra al usuario los nombres de los archivos generados con los resultados de la medición
```

Código 6.5 medicion_error_mm2.py.

6.6 kalman2d.py

```
import numpy as np

class Kalman2D:
    def __init__(self, dt=0.2, process_variance=1e-2, measurement_variance=1e-1):
        self.dt = dt
        self.x = np.zeros((4, 1))
        self.F = np.array([[1, 0, dt, 0],
                           [0, 1, 0, dt],
                           [0, 0, 1, 0],
                           [0, 0, 0, 1]])
        self.H = np.array([[1, 0, 0, 0],
                           [0, 1, 0, 0]])
        self.P = np.eye(4)
        self.Q = process_variance * np.eye(4)
        self.R = measurement_variance * np.eye(2)

    def update(self, z):
        z = np.reshape(z, (2, 1))
        self.x = np.dot(self.F, self.x)
        self.P = np.dot(self.F, np.dot(self.P, self.F.T)) + self.Q
        y = z - np.dot(self.H, self.x)
        S = np.dot(self.H, np.dot(self.P, self.H.T)) + self.R
        K = np.dot(self.P, np.dot(self.H.T, np.linalg.inv(S)))
        self.x = self.x + np.dot(K, y)
        I = np.eye(self.F.shape[0])
        self.P = np.dot((I - np.dot(K, self.H)), self.P)
        return self.x[0, 0], self.x[1, 0]
```

Código 6.6 kalman2d.py.

6.7 navegacion_K6.py

```python
#IMPORTACIONES
import asyncio
# Para manejo de tareas asincronas
import math
# Funciones matemáticas como sqrt y atan2
import signal
# Captura de señales del sistema (ej. CTRL+C)
import sys
# Acceso a funciones del sistema
import matplotlib.pyplot as plt
# Gráficos de trayectoria

from irobot_edu_sdk.backend.bluetooth import Bluetooth
# Backend Bluetooth para Create3
from irobot_edu_sdk.robots import event, Create3
# Control del robot Create3 con eventos
from kalman2d import Kalman2D
# Filtro de Kalman para suavizar coordenadas
from marvelmind import MarvelmindHedge
# Posicionamiento por ultrasonido Marvelmind


#INICIALIZACIÓN DE HARDWARE
hedge = MarvelmindHedge(tty="COM3", adr=0x4E, debug=False)
# Inicializa sensor Marvelmind por puerto COM3
hedge.start()
# Comienza la lectura de datos del sensor

backend = Bluetooth()
# Prepara conexión Bluetooth
robot = Create3(backend)
# Inicializa robot Create3 usando Bluetooth


#CONSTANTES Y VARIABLES GLOBALES
TOLERANCIA_POS = 0.10
# Distancia mínima al destino para considerar que se ha llegado
TOLERANCIA_ANGULO_INICIAL = 30
# Margen de error angular permitido al comenzar
UMBRAL_PROMEDIO = 400
# Umbral para detección de obstáculos por promedio de sensores
UMBRAL_FRENADA_BRUSCA = 1500
# Valor límite para parada inmediata

programa_activo = True
# Bandera que indica si el programa sigue en ejecución
trayectoria = []
# Lista para guardar la trayectoria recorrida
x_objetivo = None
# Coordenada X del destino
y_objetivo = None
# Coordenada Y del destino

kalman = Kalman2D(dt=0.2)
# Filtro de Kalman con paso de tiempo de 200 ms. Sincroniza el modelo de predicción del filtro de Kalman con la
      frecuencia real del sistema (≈5 Hz).


#FUNCIÓN PARA DETENER EL ROBOT Y GUARDAR DATOS
async def detener_robot():
# Finaliza el programa de forma segura
    global programa_activo, x_objetivo, y_objetivo
    programa_activo = False
    print("\n Deteniendo robot...")
    await robot.set_wheel_speeds(0, 0)
    # Detiene las ruedas
    hedge.stop()
    # Detiene la lectura del sensor Marvelmind
    print(" Navegación finalizada.")
```

(continuación en la siguiente página)

(continuación del código anterior)

```
    if trayectoria:
    # Si hay datos de trayectoria, los guarda
        x_vals, y_vals, _ = zip(*trayectoria)
        plt.title("Trayectoria recorrida")
        plt.xlabel("X (m)")
        plt.ylabel("Y (m)")
        plt.grid(True)
        plt.axis("equal")
        plt.plot(x_vals, y_vals, '-o')
        plt.plot(x_objetivo, y_objetivo, 'rx')
        # Marca el objetivo con una X roja

        from datetime import datetime
        nombre_archivo = f"trayectoria_{datetime.now().strftime('%Y%m%d_%H%M%S')}.png"
        plt.savefig(nombre_archivo)
        # Guarda el gráfico como imagen
        print(f" Gráfica guardada como: {nombre_archivo}")
        plt.show()

        import csv
        timestamp_actual = datetime.now().strftime('%Y%m%d_%H%M%S')
        nombre_csv = f"trayectoria_{timestamp_actual}.csv"

        with open(nombre_csv, mode='w', newline='') as archivo:
            writer = csv.writer(archivo)
            writer.writerow(["Timestamp", "X (m)", "Y (m)", "Ángulo (°)"])
            for (x, y, angulo) in trayectoria:
                writer.writerow([timestamp_actual, x, y, angulo])
                # Guarda datos en CSV

        print(f" Datos guardados en: {nombre_csv}")

    sys.exit(0)
    # Cierra el programa


#GESTIÓN DE SEÑALES (CTRL+C)
def signal_handler(sig, frame):
# Maneja interrupción manual (Ctrl+C)
    asyncio.create_task(detener_robot())
    # Ejecuta la función de detención asincrona

signal.signal(signal.SIGINT, signal_handler)
# Enlaza la señal SIGINT a la función anterior


#FUNCIONES AUXILIARES DE NAVEGACIÓN
def calcular_angulo_objetivo(x_actual, y_actual, x_dest, y_dest):
# Calcula ángulo del robot hacia el destino
    return math.degrees(math.atan2(y_dest - y_actual, x_dest - x_actual)) % 360

def diferencia_angular(a1, a2):
# Calcula la diferencia mínima entre dos ángulos en grados (-180 a 180)
    return ((a2 - a1 + 180) % 360) - 180

def calcular_distancia(x1, y1, x2, y2):
# Distancia Euclídea entre dos puntos (x1,y1) y (x2,y2)
    return math.sqrt((x2 - x1)**2 + (y2 - y1)**2)


#FUNCIÓN PRINCIPAL DEL ROBOT
@event(robot.when_play)
# Evento que se ejecuta cuando se inicia el robot
async def main(robot):
    global programa_activo, x_objetivo, y_objetivo
    main.velocidad_actual = 20
    # Velocidad inicial
    main.angulo_objetivo_congelado = None
    main.reorientado = False
    main.evasion_duracion = 0
    # Temporizador de duración de evasión


#ENTRADA DE COORDENADAS Y CONFIGURACIÓN DE GRÁFICA
    print("\n Inicio del sistema de navegación.")

    try:
        x_objetivo = float(input("Introduce coordenada X destino (en metros): "))
        # Entrada escogida por usuario
        y_objetivo = float(input("Introduce coordenada Y destino (en metros): "))
```

(continuación en la siguiente página)

(continuación del código anterior)

```
    except ValueError:
        print(" Coordenadas inválidas.")
        await detener_robot()
        return

    plt.ion()
    # Activa modo interactivo en matplotlib
    fig, ax = plt.subplots()
    ax.set_title("Trayectoria en tiempo real")
    ax.set_xlabel("X (m)")
    ax.set_ylabel("Y (m)")
    ax.grid(True)


#INICIALIZACIÓN DE VARIABLES DE CONTROL
    orientado = False
    # Bandera: ya se orientó al objetivo?
    inicializado = False
    # Para evitar evasión antes de comenzar
    main.evadiendo = False
    # Indica si está en maniobra evasiva

    distancia_anterior = float('inf')
    # Usado para detectar estancamiento
    contador_estancado = 0
    sensores_ir_anteriores = [0] * 7
    # Lectura previa de sensores IR


#BUCLE PRINCIPAL DE NAVEGACIÓN
    while programa_activo:
        try:
            datos = hedge.position()
            # Obtiene posición y orientación desde Marvelmind
            if not datos or datos[1] == 0 and datos[2] == 0:
                print(" Esperando datos válidos de posicionamiento...")
                await asyncio.sleep(0.2)
                continue

            _, x_med, y_med, _, angulo_actual, *_ = datos
            # Extrae x, y, ángulo crudo
            angulo_actual = angulo_actual / 10
            # Corrige ángulo (viene en deci-grados)

            x, y = kalman.update([x_med, y_med])
            # Aplica filtro Kalman
            trayectoria.append((x, y, angulo_actual))
            # Guarda en trayectoria

            ax.clear()
            ax.set_title("Trayectoria en tiempo real")
            ax.set_xlabel("X (m)")
            ax.set_ylabel("Y (m)")
            ax.grid(True)
            if trayectoria:
                x_vals, y_vals, _ = zip(*trayectoria)
                ax.plot(x_vals, y_vals, '-o', label="Recorrido")
                # Traza recorrido
                ax.plot(x_objetivo, y_objetivo, 'rx', label="Destino")
                # Marca destino
                ax.legend()
            plt.pause(0.01)
            # Actualiza gráfica


            #EVALUACIÓN DE POSICIÓN Y DISTANCIA AL OBJETIVO
            distancia = calcular_distancia(x, y, x_objetivo, y_objetivo)
            # Calcula distancia actual al objetivo

            print(f"\n Posición: ({x:.2f}, {y:.2f})")

            print(f" Distancia al destino: {distancia:.2f} m")

            if distancia <= TOLERANCIA_POS:
            # Si se llegó al destino
                print(" Objetivo alcanzado.")
                await robot.set_wheel_speeds(0, 0)
                break
```

(continuación en la siguiente página)

(continuación del código anterior)

```
            #ORIENTACIÓN INICIAL ANTES DE MOVERSE
            if not orientado:
            # Solo se ejecuta la primera vez, antes de que el robot empiece a moverse
                print(" Alineando orientación antes de avanzar...")

                x_ini, y_ini = x, y
                # Se congela el ángulo objetivo desde la posición actual hasta el destino
                angulo_objetivo = calcular_angulo_objetivo(x_ini, y_ini, x_objetivo, y_objetivo)
                # Calcula ángulo del robot hacia el destino

                for intento in range(100):
                # Intenta como máximo durante 100 ciclos (~10 segundos)
                    datos = hedge.position()
                    # Se vuelve a leer el ángulo actual desde Marvelmind
                    if not datos:
                        continue
                        # Si no hay datos válidos, se salta esta iteración

                    _, _, _, _, angulo_actual_raw, *_ = datos
                    # Extrae el ángulo crudo (en deci-grados)
                    angulo_actual = angulo_actual_raw / 10
                    # Convierte a grados reales
                    diferencia = diferencia_angular(angulo_actual, angulo_objetivo)
                    # Calcular diferencia angular (Δ)

                    print(f" Corrigiendo: actual={angulo_actual:.1f} | objetivo={angulo_objetivo:.1f} | Δ={
                        diferencia:.1f}")

                    if abs(diferencia) <= TOLERANCIA_ANGULO_INICIAL:
                        break
                        # Si ya está suficientemente alineado (por ejemplo, ±30°), se detiene la rotación

                    # Gira en su lugar, sin moverse adelante/atrás
                    direccion = 1 if diferencia > 0 else -1
                    # Decide sentido del giro: horario (derecha) o antihorario (izquierda)
                    await robot.set_wheel_speeds(-15 * direccion, 15 * direccion)
                    # Gira en el lugar a velocidad moderada
                    await asyncio.sleep(0.2)
                    # Espera mientras gira
                    await robot.set_wheel_speeds(0, 0)
                    # Detiene el giro brevemente
                    await asyncio.sleep(0.2)
                    # Permite que el sensor actualice el ángulo

                orientado = True
                # Marca que ya se ha alineado al objetivo
                inicializado = True
                # Ahora se permiten evasiones y cálculos dinámicos de rumbo
                await robot.set_wheel_speeds(0, 0)
                # Asegura que el robot esté completamente detenido
                await asyncio.sleep(0.5)
                # Pequeña pausa antes de comenzar a avanzar
                continue
                # Vuelve al bucle sin ejecutar navegación todavía (paso extra de seguridad)


            #RE-CÁLCULO DE ÁNGULO HACIA EL DESTINO
            angulo_objetivo = calcular_angulo_objetivo(x, y, x_objetivo, y_objetivo)
            # Calcula nuevo ángulo hacia el objetivo
            diferencia = diferencia_angular(angulo_actual, angulo_objetivo)
            # Diferencia con orientación actual
            print(f" Ángulo actual: {angulo_actual:.1f} | Objetivo: {angulo_objetivo:.1f} | Δ: {diferencia:.1f}"
                )


            #LECTURA DE SENSORES INFRARROJOS
            sensores_ir = (await robot.get_ir_proximity()).sensors
            # Lee los 7 sensores IR del frontal del robot
            print(f"Sensores IR: {sensores_ir}")


            # DETECCIÓN DE OBSTÁCULOS CRÍTICOS
            promedio_ir = sum(sensores_ir[1:4]) / 3
            # Promedio de sensores frontales (índices 1, 2, 3)
            if promedio_ir > UMBRAL_PROMEDIO or any(s > UMBRAL_FRENADA_BRUSCA for s in sensores_ir):
                print(" Obstáculo detectado (promedio o pico alto en cualquier sensor). Deteniendo robot.")
                await robot.set_wheel_speeds(0, 0)
                # El robot se detiene por seguridad y finaliza la navegación.
                await detener_robot()
                break
```

(continuación en la siguiente página)

(continuación del código anterior)

```
            #ANÁLISIS TEMPORAL DE OBSTÁCULOS: DETECCIÓN DE CAMBIOS REPENTINOS
            incrementos = [sensores_ir[i] - sensores_ir_anteriores[i] for i in range(1, 6)]
            # Diferencias frente al ciclo anterior
            valores_altos = any(sensores_ir[i] > 60 for i in range(1, 6))
            # Algún sensor en zona de riesgo?
            sensores_ir_anteriores = sensores_ir.copy()
            # Actualiza valores para el próximo ciclo

            izq_total = sensores_ir[0] + sensores_ir[1] + sensores_ir[2]
            # Suma de sensores del lado izquierdo
            der_total = sensores_ir[4] + sensores_ir[5] + sensores_ir[6]
            # Suma de sensores del lado derecho
            valor_frontal = max(sensores_ir)
            # Valor máximo de todos los sensores (para umbrales)


            #CÁLCULO DE VELOCIDAD SEGÚN PROXIMIDAD Y PELIGRO
            evasivo = 0
            # Inicializa el componente de evasión en 0. Solo se modifica si hay peligro real.
            nueva_velocidad = main.velocidad_actual
            # Por defecto, se mantiene la velocidad actual.

            if valor_frontal > 1500:
                nueva_velocidad = 0
                # Obstáculo extremadamente cerca: ordena una parada inmediata
                print(" Objeto demasiado cerca. Parando y finalizando navegación.")
                await robot.set_wheel_speeds(0, 0)
                # Detiene el robot inmediatamente
                await detener_robot()
                # Llama a la rutina de cierre (guarda datos, apaga sensores, etc.)
                break
                # Sale del bucle principal
            elif valor_frontal > 50 or (any(inc > 5 for inc in incrementos) and valor_frontal > 40):
                nueva_velocidad = 5
                #  Alerta: obstáculo muy cerca o acercamiento muy rápido →velocidad mínima

                print(" Alerta: incremento alto y valor relevante. Reduciendo a 5")

            elif valor_frontal > 40 or (any(inc > 4 for inc in incrementos) and valor_frontal > 30):
                nueva_velocidad = 10
                #  Precaución: obstáculo a media distancia o incremento moderado →velocidad media-baja

                print(" Precaución: crecimiento moderado y valor relevante. Reduciendo a 10")

            elif valor_frontal > 30:
                nueva_velocidad = 15
                #  Objeto detectado lejos →pequeña reducción preventiva

                print(" Objeto detectado lejos. Reduciendo a 15")
            else:
                nueva_velocidad = 20
                #  Zona libre →velocidad máxima permitida
                print(" Zona libre: velocidad máxima")


            #SUAVIZACIÓN DE CAMBIOS DE VELOCIDAD
            if nueva_velocidad < main.velocidad_actual:
            # Evita saltos bruscos: si se reduce la velocidad, lo hace de golpe (seguridad); si se aumenta, se
                  hace poco a poco (+2 por ciclo).
                main.velocidad_actual = nueva_velocidad
                # Reducción inmediata

            elif nueva_velocidad > main.velocidad_actual:
                main.velocidad_actual = min(main.velocidad_actual + 2, nueva_velocidad)
                # Aumento progresivo

            velocidad = main.velocidad_actual


            # DETECCIÓN Y ACTIVACIÓN DE MANIOBRA EVASIVA
            if inicializado and valor_frontal > 55:
            # Solo evade si ya ha iniciado navegación y hay obstáculo claro delante
                if abs(diferencia) < 50 or main.evadiendo or main.evasion_duracion > 0:
                # Se considera evasión solo si: 1)El robot no está muy desalineado (Δ < 50°) →es viable girar
                      lateralmente, 2)ya está evadiendo (para continuar la evasión), 3)aún está dentro del
                      periodo de evasión activa (temporizador no agotado)
                    evasivo = -0.95 if izq_total > der_total else +0.95
                    # Determina el lado por donde es más seguro esquivar. Si la suma de sensores izquierdos es
                         mayor →el obstáculo está a la izquierda →gira a la derecha. Viceversa si hay más presi
                         ón a la derecha
```

(continuación en la siguiente página)

(continuación del código anterior)

```
                    if not main.evadiendo:
                    # Si esta es una nueva evasión (no estaba evadiendo)
                        print(" Maniobra evasiva activada:", "derecha" if evasivo > 0 else "izquierda")
                    else:
                        print(" Refrescando duración de evasión por nuevo obstáculo.")

                    main.evasion_duracion = 10
                    # Si ya está evadiendo, se reinicia el temporizador para extender la maniobra. Temporizador
                         de evasión: 10 ciclos (≈2 segundos)
                    main.evadiendo = True
                    # Marca que estamos en modo evasivo
                    main.reorientado = False
                    # Aún no se ha realineado con el objetivo tras la evasión

                else:
                    evasivo = 0
                    # Si no se cumplen las condiciones para evadir, el ajuste evasivo se anula
            else:
                if main.evadiendo:
                # Si no se detecta un nuevo obstáculo pero está evadiendo, se descuenta duración
                    main.evasion_duracion -= 1
                    # Reduce contador en cada ciclo
                    print(f"[DEBUG] evadiendo: {main.evadiendo}, duracion: {main.evasion_duracion}, valor_frontal:
                          {valor_frontal}")

                    if main.evasion_duracion <= 0:
                    # Cuando se acaba el tiempo de evasión, se inicia la reorientación
                        print(" Obstáculo esquivado. Reorientando rumbo final")


                        #BÚSQUEDA DE NUEVA ORIENTACIÓN AL OBJETIVO
                        for _ in range(20):
                        # Bucle de reorientación: intenta hasta 20 ciclos corregir el ángulo al objetivo
                            datos = hedge.position()
                            if not datos:
                                continue
                                # Si no hay datos válidos, se intenta en la próxima vuelta

                            _, x_med, y_med, _, angulo_actual_raw, *_ = datos
                            angulo_actual = angulo_actual_raw / 10
                            # Convierte a grados reales
                            x, y = kalman.update([x_med, y_med])
                            # Filtra posición

                            angulo_objetivo = calcular_angulo_objetivo(x, y, x_objetivo, y_objetivo)
                            # Recalcula ángulo desde posición actual al objetivo
                            diferencia = diferencia_angular(angulo_actual, angulo_objetivo)

                            print(f"[REORIENTACIÓN] Pos=({x:.2f}, {y:.2f}) Áct={angulo_actual:.1f} Obj={
                                 angulo_objetivo:.1f} Δ={diferencia:.1f}")

                            if abs(diferencia) <= 5:
                            # Si la diferencia angular ya es pequeña (≤5°), finaliza reorientación
                                break

                            direccion = 1 if diferencia > 0 else -1
                            # Ejecuta un pequeño giro en el sentido necesario para corregir ángulo
                            await robot.set_wheel_speeds(-10 * direccion, 10 * direccion)
                            await asyncio.sleep(0.2)
                            await robot.set_wheel_speeds(0, 0)
                            await asyncio.sleep(0.1)

                        print(" Rumbo corregido.")
                        main.reorientado = True
                        main.evadiendo = False
                        # Sale del modo evasión
                    evasivo = 0
                    # Anula ajuste evasivo si ya terminó el proceso


            #CÁLCULO DE AJUSTE DE DIRECCIÓN Y VELOCIDADES DIFERENCIALES
            k = 0.6
            # Constante proporcional para el control angular
            ajuste = max(-1, min(1, k * diferencia / 90 + evasivo))
            # Ajuste total = control angular + evasión
            print(f" Ajuste por ángulo: {k * diferencia / 90:.2f} | Evasivo: {evasivo:.2f} | Ajuste total: {
                 ajuste:.2f}")
```

(continuación en la siguiente página)

(continuación del código anterior)

```
            vel_izq = velocidad * (1 - ajuste)
            # Calcula velocidad para rueda izquierda
            vel_der = velocidad * (1 + ajuste)
            # Calcula velocidad para rueda derecha

            await robot.set_wheel_speeds(vel_izq, vel_der)
            # Aplica velocidades calculadas
            await asyncio.sleep(0.05)
            # Pausa breve entre ciclos


            #DETECCIÓN DE ESTANCAMIENTO
            if abs(distancia_anterior - distancia) < 0.01:
            # Si casi no ha avanzado
                contador_estancado += 1
            else:
                contador_estancado = 0
                # Reinicia si detecta avance
            distancia_anterior = distancia
            # Guarda distancia para próxima comparación

            if contador_estancado > 15:
            # Si lleva más de 15 ciclos (≈3 s) estancado
                print(" Estancado. Deteniendo robot.")
                await robot.set_wheel_speeds(0, 0)
                break
                # Finaliza el bucle y detiene el robot


        #MANEJO DE ERRORES DURANTE LA NAVEGACIÓN
        except Exception as e:
            print(f" Error: {e}")
            # Captura y muestra cualquier excepción
            await robot.set_wheel_speeds(0, 0)
            # Detiene el robot por seguridad
            await asyncio.sleep(0.5)
            # Espera medio segundo antes de reintentar

    await detener_robot()
    # Ejecuta rutina final: guarda trayectoria, detiene sensores, etc.


#EJECUCIÓN PRINCIPAL DEL PROGRAMA
if __name__ == '__main__':
    try:
        robot.play()
        # Inicia el evento principal del robot (main)
    except KeyboardInterrupt:
        print("\n CTRL+C detectado. Cerrando...")
        asyncio.run(detener_robot())
        # Cierre limpio si se interrumpe con teclado
    except Exception as e:
        print(f" Error final: {e}")
        # Captura errores al ejecutar el script
```

Código 6.7 navegacion_K6.py.

6.8 navegacion_TAPO.py

```python
import asyncio
import math
import signal
import sys
import matplotlib.pyplot as plt
import cv2
import threading
import os


from irobot_edu_sdk.backend.bluetooth import Bluetooth
from irobot_edu_sdk.robots import event, Create3
from kalman2d import Kalman2D
from marvelmind import MarvelmindHedge
from ultralytics import YOLO

hedge = MarvelmindHedge(tty="COM3", adr=0x4E, debug=False)
hedge.start()

backend = Bluetooth()
robot = Create3(backend)

TOLERANCIA_POS = 0.10
TOLERANCIA_ANGULO_INICIAL = 30
UMBRAL_PROMEDIO = 400
UMBRAL_FRENADA_BRUSCA = 1500

programa_activo = True
trayectoria = []
x_objetivo = None
y_objetivo = None

kalman = Kalman2D(dt=0.2)

async def detener_robot():
    global programa_activo, x_objetivo, y_objetivo
    programa_activo = False
    print("\n Deteniendo robot...")
    await robot.set_wheel_speeds(0, 0)
    hedge.stop()
    print(" Navegación finalizada.")

    if trayectoria:
        x_vals, y_vals, _ = zip(*trayectoria)
        plt.title("Trayectoria recorrida")
        plt.xlabel("X (m)")
        plt.ylabel("Y (m)")
        plt.grid(True)
        plt.axis("equal")
        plt.plot(x_vals, y_vals, '-o')
        plt.plot(x_objetivo, y_objetivo, 'rx')

        from datetime import datetime
        nombre_archivo = f"trayectoria_{datetime.now().strftime('%Y%m%d_%H%M%S')}.png"
        plt.savefig(nombre_archivo)
        print(f" Gráfica guardada como: {nombre_archivo}")
        plt.show()

        import csv
        timestamp_actual = datetime.now().strftime('%Y%m%d_%H%M%S')
        nombre_csv = f"trayectoria_{timestamp_actual}.csv"

        with open(nombre_csv, mode='w', newline='') as archivo:
            writer = csv.writer(archivo)
            writer.writerow(["Timestamp", "X (m)", "Y (m)", "Ángulo (°)"])
            for (x, y, angulo) in trayectoria:
                writer.writerow([timestamp_actual, x, y, angulo])

        print(f" Datos guardados en: {nombre_csv}")

    sys.exit(0)

def signal_handler(sig, frame):
    asyncio.create_task(detener_robot())

signal.signal(signal.SIGINT, signal_handler)
```

(continuación en la siguiente página)

(continuación del código anterior)

```
    def calcular_angulo_objetivo(x_actual, y_actual, x_dest, y_dest):
        return math.degrees(math.atan2(y_dest - y_actual, x_dest - x_actual)) % 360

    def diferencia_angular(a1, a2):
        return ((a2 - a1 + 180) % 360) - 180

    def calcular_distancia(x1, y1, x2, y2):
        return math.sqrt((x2 - x1)**2 + (y2 - y1)**2)


    objetos_detectados = []
    # Lista global para almacenar los objetos detectados por YOLO en tiempo real


    #BLOQUE DE INTEGRACIÓN DE VISIÓN ARTIFICIAL
    def vision_tapo_thread():
        global objetos_detectados
        # Declaramos que vamos a modificar la variable global desde esta función

        modelo = YOLO('yolov8n.pt')
        # Carga el modelo YOLOv8 en su versión más ligera (nano) para detección de objetos

        usuario = "antoniolopez"
        # Credenciales y datos de conexión de la cámara TP-Link Tapo
        contrasena = "roombatfg"
        ip = "192.168.236.29"
        url_rtsp = f"rtsp://{usuario}:{contrasena}@{ip}:554/stream1"
        # Construye la URL RTSP con usuario, contraseña e IP

        os.environ["OPENCV_VIDEOIO_DEBUG"] = "0"
        # Desactiva mensajes de depuración de OpenCV para evitar ruido en consola
        cap = cv2.VideoCapture(url_rtsp)
        # Abre el flujo de vídeo desde la cámara

        if not cap.isOpened():
        # Si no se logra abrir el stream, se muestra error y sale
            print(" No se pudo conectar al stream RTSP.")
            return

        print(" Cámara conectada. Detectando objetos...")
        # Mensaje indicando que la cámara está lista y comenzará detección

        vision_tapo_thread.last_deteccion = []
        # Guarda la última lista de detecciones para detectar cambios
        frame_count = 0
        # Contador de frames procesados
        N = 30
        # Procesar solo 1 de cada N frames para reducir carga de CPU

        while True:
        # Bucle infinito de lectura de frames
            ret, frame = cap.read()
            # Captura un frame del stream
            if not ret:
            # Si no se pudo leer, muestra advertencia y continúa
                print(" No se pudo leer el frame.")
                continue

            if frame_count % N == 0:
            # Procesa el frame solo cada N iteraciones
                resultados = modelo(frame, verbose=False)[0]
                # Ejecuta detección de objetos sobre el frame
                anotado = resultados.plot()
                # Dibuja anotaciones (cajas y etiquetas) sobre el frame

                objetos_detectados = list(set([modelo.names[int(cls)] for cls in resultados.boxes.cls]))
                # Extrae nombres de clases detectadas y elimina duplicados

                if set(objetos_detectados) != set(vision_tapo_thread.last_deteccion):
                # Si la detección actual es distinta a la anterior, actualiza
                    vision_tapo_thread.last_deteccion = objetos_detectados.copy()

            else:
            # Si no se procesa detección, muestra el frame sin anotaciones
                anotado = frame

            frame_count += 1
            # Aumenta el contador de frames
```

(continuación en la siguiente página)

(continuación del código anterior)

```
        cv2.imshow("Tapo + YOLO", anotado)
        # Muestra el frame en una ventana con título
        if cv2.waitKey(1) & 0xFF == ord('q'):
        # Si se presiona 'q', sale del bucle
            break

    cap.release()
    # Libera el stream de vídeo
    cv2.destroyAllWindows()
    # Cierra todas las ventanas abiertas de OpenCV


@event(robot.when_play)
async def main(robot):
    global programa_activo, x_objetivo, y_objetivo
    main.velocidad_actual = 20
    main.angulo_objetivo_congelado = None
    main.reorientado = False
    main.evasion_duracion = 0


    print("\n Inicio del sistema de navegación.")
    print(" Iniciando cámara...")
    threading.Thread(target=vision_tapo_thread, daemon=True).start()
    await asyncio.sleep(0.5)

    try:
        x_objetivo = float(input("Introduce coordenada X destino (en metros): "))
        y_objetivo = float(input("Introduce coordenada Y destino (en metros): "))
    except ValueError:
        print(" Coordenadas inválidas.")
        await detener_robot()
        return

    plt.ion()
    fig, ax = plt.subplots()
    ax.set_title("Trayectoria en tiempo real")
    ax.set_xlabel("X (m)")
    ax.set_ylabel("Y (m)")
    ax.grid(True)

    orientado = False
    inicializado = False
    main.evadiendo = False


    distancia_anterior = float('inf')
    contador_estancado = 0
    sensores_ir_anteriores = [0] * 7

    while programa_activo:
        try:
            datos = hedge.position()
            if not datos or datos[1] == 0 and datos[2] == 0:
                print(" Esperando datos válidos de posicionamiento...")
                await asyncio.sleep(0.2)
                continue

            _, x_med, y_med, _, angulo_actual, *_ = datos
            angulo_actual = angulo_actual / 10

            x, y = kalman.update([x_med, y_med])
            trayectoria.append((x, y, angulo_actual))

            ax.clear()
            ax.set_title("Trayectoria en tiempo real")
            ax.set_xlabel("X (m)")
            ax.set_ylabel("Y (m)")
            ax.grid(True)
            if trayectoria:
                x_vals, y_vals, _ = zip(*trayectoria)
                ax.plot(x_vals, y_vals, '-o', label="Recorrido")
                ax.plot(x_objetivo, y_objetivo, 'rx', label="Destino")
                ax.legend()
            plt.pause(0.01)
```

(continuación en la siguiente página)

(continuación del código anterior)

```
            distancia = calcular_distancia(x, y, x_objetivo, y_objetivo)

            print(f"\n Posición: ({x:.2f}, {y:.2f})")

            print(f" Distancia al destino: {distancia:.2f} m")

            if distancia <= TOLERANCIA_POS:
                print(" Objetivo alcanzado.")
                await robot.set_wheel_speeds(0, 0)
                break


            if not orientado:
                print(" Alineando orientación antes de avanzar...")

                x_ini, y_ini = x, y
                angulo_objetivo = calcular_angulo_objetivo(x_ini, y_ini, x_objetivo, y_objetivo)

                for intento in range(100):
                    datos = hedge.position()
                    if not datos:
                        continue

                    _, _, _, _, angulo_actual_raw, *_ = datos
                    angulo_actual = angulo_actual_raw / 10
                    diferencia = diferencia_angular(angulo_actual, angulo_objetivo)

                    print(f" Corrigiendo: actual={angulo_actual:.1f} | objetivo={angulo_objetivo:.1f} | Δ={
                        diferencia:.1f}")

                    if abs(diferencia) <= TOLERANCIA_ANGULO_INICIAL:
                        break

                    direccion = 1 if diferencia > 0 else -1
                    await robot.set_wheel_speeds(-15 * direccion, 15 * direccion)
                    await asyncio.sleep(0.2)
                    await robot.set_wheel_speeds(0, 0)
                    await asyncio.sleep(0.2)

                orientado = True
                inicializado = True
                await robot.set_wheel_speeds(0, 0)
                await asyncio.sleep(0.5)
                continue


            angulo_objetivo = calcular_angulo_objetivo(x, y, x_objetivo, y_objetivo)
            diferencia = diferencia_angular(angulo_actual, angulo_objetivo)
            print(f" Ángulo actual: {angulo_actual:.1f} | Objetivo: {angulo_objetivo:.1f} | Δ: {diferencia:.1f}"
                )

            sensores_ir = (await robot.get_ir_proximity()).sensors
            print(f"Sensores IR: {sensores_ir}")


            promedio_ir = sum(sensores_ir[1:4]) / 3
            if promedio_ir > UMBRAL_PROMEDIO or any(s > UMBRAL_FRENADA_BRUSCA for s in sensores_ir):
                print(" Obstáculo detectado (promedio o pico alto en cualquier sensor). Deteniendo robot.")
                await robot.set_wheel_speeds(0, 0)
                await detener_robot()
                break

            incrementos = [sensores_ir[i] - sensores_ir_anteriores[i] for i in range(1, 6)]
            valores_altos = any(sensores_ir[i] > 60 for i in range(1, 6))
            sensores_ir_anteriores = sensores_ir.copy()

            izq_total = sensores_ir[0] + sensores_ir[1] + sensores_ir[2]
            der_total = sensores_ir[4] + sensores_ir[5] + sensores_ir[6]
            valor_frontal = max(sensores_ir)


            evasivo = 0
            nueva_velocidad = main.velocidad_actual

            if valor_frontal > 1500:
                nueva_velocidad = 0
                print(" Objeto demasiado cerca. Parando y finalizando navegación.")
                await robot.set_wheel_speeds(0, 0)
                await detener_robot()
                break
```

(continuación en la siguiente página)

(continuación del código anterior)

```
            elif valor_frontal > 50 or (any(inc > 5 for inc in incrementos) and valor_frontal > 40):
                nueva_velocidad = 5
                print(" Alerta: incremento alto y valor relevante. Reduciendo a 5")

            elif valor_frontal > 40 or (any(inc > 4 for inc in incrementos) and valor_frontal > 30):
                nueva_velocidad = 10
                print(" Precaución: crecimiento moderado y valor relevante. Reduciendo a 10")

            elif valor_frontal > 30:
                nueva_velocidad = 15
                print(" Objeto detectado lejos. Reduciendo a 15")
            else:
                nueva_velocidad = 20
                print(" Zona libre: velocidad máxima")


            if nueva_velocidad < main.velocidad_actual:
                main.velocidad_actual = nueva_velocidad

            elif nueva_velocidad > main.velocidad_actual:
                main.velocidad_actual = min(main.velocidad_actual + 2, nueva_velocidad)

            velocidad = main.velocidad_actual


            if inicializado and valor_frontal > 55:
                if abs(diferencia) < 50 or main.evadiendo or main.evasion_duracion > 0:
                    evasivo = -0.95 if izq_total > der_total else +0.95

                    if not main.evadiendo:
                        print(" Maniobra evasiva activada:", "derecha" if evasivo > 0 else "izquierda")
                    else:
                        print(" Refrescando duración de evasión por nuevo obstáculo.")

                    main.evasion_duracion = 10

                    main.evadiendo = True
                    main.reorientado = False
                else:
                    evasivo = 0
            else:
                if main.evadiendo:
                    main.evasion_duracion -= 1
                    print(f"[DEBUG] evadiendo: {main.evadiendo}, duracion: {main.evasion_duracion}, valor_frontal:
                          {valor_frontal}")

                    if main.evasion_duracion <= 0:
                        print(" Obstáculo esquivado. Reorientando rumbo final...")


                        for _ in range(20):
                            datos = hedge.position()
                            if not datos:
                                continue

                            _, x_med, y_med, _, angulo_actual_raw, *_ = datos
                            angulo_actual = angulo_actual_raw / 10
                            x, y = kalman.update([x_med, y_med])

                            angulo_objetivo = calcular_angulo_objetivo(x, y, x_objetivo, y_objetivo)
                            diferencia = diferencia_angular(angulo_actual, angulo_objetivo)

                            print(f"[REORIENTACIÓN] Pos=({x:.2f}, {y:.2f}) Áct={angulo_actual:.1f} Obj={
                                  angulo_objetivo:.1f} Δ={diferencia:.1f}")

                            if abs(diferencia) <= 5:
                                break

                            direccion = 1 if diferencia > 0 else -1
                            await robot.set_wheel_speeds(-10 * direccion, 10 * direccion)
                            await asyncio.sleep(0.2)
                            await robot.set_wheel_speeds(0, 0)
                            await asyncio.sleep(0.1)

                        print(" Rumbo corregido.")
                        main.reorientado = True
                        main.evadiendo = False
                    evasivo = 0
```

(continuación en la siguiente página)

(continuación del código anterior)

```
            k = 0.6
            ajuste = max(-1, min(1, k * diferencia / 90 + evasivo))
            print(f" Ajuste por ángulo: {k * diferencia / 90:.2f} | Evasivo: {evasivo:.2f} | Ajuste total: {
                ajuste:.2f}")

            vel_izq = velocidad * (1 - ajuste)
            vel_der = velocidad * (1 + ajuste)

            await robot.set_wheel_speeds(vel_izq, vel_der)
            await asyncio.sleep(0.05)

            if abs(distancia_anterior - distancia) < 0.01:
                contador_estancado += 1
            else:
                contador_estancado = 0
            distancia_anterior = distancia

            if contador_estancado > 15:
                print(" Estancado. Deteniendo robot.")
                await robot.set_wheel_speeds(0, 0)
                break


        except Exception as e:
            print(f" Error: {e}")
            await robot.set_wheel_speeds(0, 0)
            await asyncio.sleep(0.5)



    await detener_robot()

if __name__ == '__main__':
    try:
        robot.play()
    except KeyboardInterrupt:
        print("\n CTRL+C detectado. Cerrando...")
        asyncio.run(detener_robot())
    except Exception as e:
        print(f" Error final: {e}")
```

Código 6.8 navegacion_TAPO.py.

Bibliografía

[1] Grover, R. A., & Kapoor, R. (2021). *Robotics: Control, Sensing, Vision, and Intelligence*. McGraw-Hill Education.

[2] Siegwart, R., Nourbakhsh, I. R., & Scaramuzza, D. (2011). *Introduction to Autonomous Mobile Robots* (2nd ed.). MIT Press.

[3] Thrun, S., Burgard, W., & Fox, D. (2005). *Probabilistic Robotics*. MIT Press.

[4] Ultralytics. (2024). *YOLOv8 Documentation*. Recuperado de *https://docs.ultralytics.com*

[5] Python Software Foundation. (2024). *Python 3.10 Documentation*. Recuperado de *https://docs.python.org/3/*

[6] Van Rossum, G., & Drake, F. L. (2001). *The Python Language Reference Manual*. Network Theory Ltd.

[7] iRobot. (2023). *iRobot®Create®3 Developer Documentation*. Recuperado de *https://create.irobot.com*

[8] Marvelmind Robotics. (2022). *Marvelmind Indoor Navigation System – User Manual v5.103*. Recuperado de *https://marvelmind.com/downloads/*

[9] Matplotlib Development Team. (2023). *Matplotlib: Visualization with Python*. Recuperado de *https://matplotlib.org*

[10] OpenCV.org. (2023). *OpenCV-Python Tutorials*. Recuperado de *https://docs.opencv.org*

[11] Ultralytics. (2024). *ultralytics Python Package*. Recuperado de *https://pypi.org/project/ultralytics/*

[12] Python Software Foundation. (2024). *asyncio – Asynchronous I/O*. Recuperado de *https://docs.python.org/3/library/asyncio.html*

[13] Python Software Foundation. (2024). *json — JSON encoder and decoder*. Recuperado de *https://docs.python.org/3/library/json.html*

[14] Python Software Foundation. (2024). *csv — CSV File Reading and Writing*. Recuperado de *https://docs.python.org/3/library/csv.html*

[15] Redmon, J., & Farhadi, A. (2018). *YOLOv3: An Incremental Improvement*. arXiv:1804.02767. Recuperado de *https://arxiv.org/abs/1804.02767*

[16] GitHub. (2023). *Create 3 SDK Examples*. Recuperado de *https://github.com/iRobotEducation/create3_examples*