UNIVERSIDAD MIGUEL HERNÁNDEZ DE ELCHE

ESCUELA POLITÉCNICA SUPERIOR DE ELCHE

GRADO EN INGENIERÍA DE TECNOLOGÍAS DE TELECOMUNICACIÓN

"DISEÑO E IMPLEMENTACIÓN DE UN SISTEMA DE NAVEGACIÓN AUTÓNOMA PARA ROBOTS MÓVILES EN EL EXTERIOR"

TRABAJO FIN DE GRADO

Junio - 2024

AUTOR: Mario Saura Llópez

DIRECTOR/ES: David Valiente García

Antonio Santo López

AGRADECIMIENTOS

En primer lugar me gustaría agredecer a mis directores David Valiente García y Antonio Santo López por su ayuda en la realización de este trabajo. En especial a Antonio que me ha proporcionado su total implicación y dedicación durante todo el proceso. No cabe duda de que sin su ayuda no hubiera sido posible.

Además, me gustaría expresar mi agradecimiento al grupo ARVC donde he tenido la oportunidad de trabajar en un ambiente excepcional. Mis compañeros de trabajo me han brindado su apoyo constante y han hecho que me sienta como en casa.

Agradecer todo el apoyo recibido por parte de mis amigos y amigas dentro y fuera de la universidad. Ellos han sido una parte fundamental en mi vida.

A mis padres por darme la oportunidad de estudiar y crecer como persona.

Y en último lugar, a mi pareja, quien ha sido mi pilar y motivación en cada paso del camino.

Gracias.

RESUMEN

En el presente documento se desarrolla un sistema de navegación completamente autónomo para ser empleado en robots móviles en entornos exteriores. El sistema se ha implementado en un robot móvil Husky A200 de Clearpath Robotics, el cual presenta cuatro ruedas en configuración diferencial y está equipado con todo tipo de sensores para el reconocimiento del entorno.

En primer lugar se desarrollan dos sistemas de control que proporcionan las salidas de velocidad adecuadas para el robot. Se implementa un controlador PD y un controlador basado en la teoría de Lyapunov. Ambos controladores son comparados en Simulink, lo cual ha permitido seleccionar el controlador basado en Lyapunov debido a su mejor comportamiento y rendimiento.

El siguiente paso comprendió la integración del control en el entorno de ROS que es el sistema operativo que se emplea en el robot y es ampliamente utilizado en la robótica. En este contexto, se programó un Nodo en Python para generar las velocidades del robot a partir de las lecturas del GPS-RTK. Además, se ha desarrollado una planificación global con el algoritmo Dijkstra a partir de un punto destino señalado en el entorno Rviz.

Tras validar el sistema de control en el entorno simulado de Gazebo, se procedió a la integración en el robot real, realizando pruebas en un entorno exterior.

En la última parte del proyecto, se presenta un diseño para un sistema de evasión de obstáculos que permita al robot navegar de forma autónoma en entornos desconocidos. En este apartado se hace uso de redes neuronales para evaluar la transitabilidad del terreno y se implementa una planificación local con el algoritmo RRT.

Palabras clave: Sistema de navegación autónomo, robots móviles, Husky A200, ROS, Simulink, Rviz, Gazebo, Python, Lyapunov, control PID, Djisktra, RRT, path-planning, redes neuronales.

ÍNDICE

AGRADECIMIENTOS II

RESUMEN IV

ÍNDICE DE TABLAS XI

ÍNDICE DE FIGURAS XIV

1. INTRODUCCIÓN 1

1.1. Planteamiento del problema 1

1.2. Motivación del proyecto 2

1.3. Objetivos 2

1.4. Estructura del documento 3

2. ESTADO DEL ARTE 5

2.1. *Mapping* 5

2.2. Localización 6

2.3. Navegación 8

2.3.1. *Path planning* 8

2.3.2. Seguimiento de trayectorias 8

2.3.3. Evasión de obstáculos 10

3. MARCO TEÓRICO **11**

3.1. Tipos de sistemas de control 11

3.2. Sistemas de control empleados 12

3.2.1. Control PID 12

3.2.1.1. Acción proporcional 13

3.2.1.2. Acción integral 14

3.2.1.3. Acción derivativa 15

3.2.2. Control basado en la teoría de Lyapunov 15

3.3. GPS-UTM 16

4. HERRAMIENTAS UTILIZADAS **18**

4.1. Plataforma robótica Husky 18

4.2. Sensores 19

4.2.1. Antena GNSS-RTK 19

4.2.2. LiDAR 21

4.2.3. Cámaras 23

4.2.3.1. Garmin VIRB 360 23

4.2.3.2. Imaging Source DFK 33UX264 24

4.2.4. IMU 24

4.3. ROS 25

4.3.1. Gazebo 27

4.3.2. Rviz 28

4.4. Matlab 28

4.4.1. Simulink 29

5. CONTROL DEL SEGUIMIENTO DE TRAYECTORIAS 30

5.1. Modelo matemático 30

5.1.1. Planteamiento del problema 30

5.1.2. Diseño PD 31

5.1.3. Diseño Lyapunov 32

5.2. Modelo en Simulink 35

5.2.1. *Tuning* del control PD 37

5.2.2. *Tuning* del control de Lyapunov 38

5.2.3. Comparación de los controladores en distintos caminos 41

6. INTEGRACIÓN EN ROS 46

6.1. Nodo de control 47

6.1.1. Evaluación experimental 48

6.2. *Path Planning* 50

6.2.1. Rviz Satellite 50

6.2.2. Dijkstra sobre OverpassTurbo 51

6.2.3. *Clicked Point* 52

6.3. Simulación 53

7. INTEGRACIÓN EN ROBOT REAL **54**

7.1. Procedimiento 54

7.2. Funcionalidades adicionales 55

7.3. Experimento y resultados 55

8. EVASIÓN DE OBSTÁCULOS **59**

8.1. Evaluación de la transitabilidad mediante una red neuronal 60

8.2. *Path Planning* local basado en algoritmo RRT 62

8.3. Unificación en un sistema de control completo 65

8.4. Modelado mundo Gazebo 67

8.5. Resultados 68

9. CONCLUSIONES Y LÍNEAS FUTURAS **71**

9.1. Conclusiones 71

9.2. Líneas futuras 73

BIBLIOGRAFÍA **74**

ANEXOS **81**

A. Anexo I 81

B. Anexo II 90

ÍNDICE DE TABLAS

3.1. Efectos independientes de cada ganancia. 15

4.1. Especificaciones ClearPath Husky A200. 19

4.2. Especificaciones técnicas de las antenas GNSS TS100. 20

4.3. Especificaciones técnicas del LiDAR OS1-128. 22

4.4. Especificaciones técnicas Garmin VIRB 360. 23

4.5. Especificaciones técnicas Imaging Source DFK 33UX264. 24

4.6. Especificaciones técnicas UM7 de Redshiftlabs. 24

5.1. Radio de las ruedas y ancho del robot 35

5.2. Valores de ganancia PID. 37

5.3. Valores de ganancia para Lyapunov. 39

5.4. Tabla de errores XTE_{medio} (m). 44

ÍNDICE DE FIGURAS

2.1. Tareas en la navegación autónoma. 5

3.1. Diagrama de bloques de un control PID [43]. 13

3.2. Efectos de la acción proporcional [44]. 14

3.3. Efectos de la acción integral [44]. 14

3.4. Efectos de la acción derivativa [44]. 15

3.5. Sistema de coordenadas universal transversal de Mercator [46]. 17

4.1. Husky Clearpath A200. 18

4.2. Dimensiones del Husky A200. 18

4.3. Harxon TS100. 20

4.4. Funcionamiento de un LiDAR 3D del fabricante SICK [47]. 21

4.5. LiDAR OS1-128. 22

4.6. Garmin VIRB 360. 23

4.7. Imaging Source DFK 33UX264. 23

4.8. IMU UM7 de Redshiftlabs. 24

4.9. Ejemplo de comunicación mediante tópicos. 25

4.10. Ejemplo de comunicación mediante servicios. 26

5.1. Posición actual y objetivo del robot [49]. 30

5.2. Variación de v para una ganancia y error d fijo. 33

5.3. Variación ω para ganancias fijas. 33

5.4. Differential Drive Kinematic Model. 35

5.5. Diagrama de bloques para obtener el error de posición. 36

5.6. Diagrama de bloques para obtener el *waypoint*. 36

5.7. Sistema de control PD 37

5.8. Trayectoria de la prueba PD. 38

5.9. Velocidades prueba PD. 38

5.10. Evolución de los errores. 38

5.11. Modelo de bloques para la unidad de control no lineal. 39

5.12. Sistema completo de seguimiento de *waypoints* con control Lyapunov. 39

5.13. Resultados para un punto objetivo. 40

5.14. Resultados para una trayectoria elíptica. 42

5.15. Resultados para una trayectoria sinusoidal. 43

5.16. Resultados para trayectoria con giros de 90^{o}. 44

6.1. Modelo del robot en Gazebo 46

6.2. Trayectoria sinusoidal Rviz. 49

6.3. Evolución del error. 49

6.4. Salidas de control para Rviz. 49

6.5. Vista del campus UMH en Rviz 50

6.6. Ejemplo grafo Dijkstra [52]. 51

6.7. Diagrama de flujo del código. 52

6.8. Trayectoria seguida en la navegación entre edificios. 53

7.1. Trayectoria exterior representada sobre imagen por satélite. 56

7.2. Plataforma Husky con los sensores instalados. 56

7.3. Ruta recorrida durante el experimento. 57

7.4. Evolución del error. 57

7.5. Salidas de control para Rviz. 57

8.1. Ilustración de la evasión de obstáculos. 59

8.2. Ilustración del análisis convolucional. 60

8.3. Visualización en Rviz de la transitabilidad mediante red neuronal. . . 61

8.4. Ejemplo de grafo RRT [54]. 62

8.5. Problemas encontrados en la elección de épsilon. 64

8.6. Diagrama de flujo del sistema de control. 65

8.7. Entorno simulado en Gazebo. 67

8.8. Camino a seguir por el robot. 67

8.9. Resultados obtenidos en el punto de partida. 68

8.10. Resultados obtenidos en el 6º *waypoint*. 69

8.11. Resultados obtenidos en el 4º *waypoint*. 70

1. INTRODUCCIÓN

Los robots han evolucionado de ser meras muestras electromecánicas en laboratorios a convertirse en herramientas de trabajo indispensables en diversos sectores de la economía, como la industria, la medicina, la agricultura y la logística, entre otros.

La robótica móvil, en particular, representa un desafío fundamental en la búsqueda de la autonomía de los robots. Un robot se considera autónomo cuando es capaz de determinar las acciones necesarias para llevar a cabo una tarea. Para lograrlo necesita un sistema de percepción que le proporcione información del entorno y una unidad de control que coordine todos los subsistemas que conforman al robot.

1.1. Planteamiento del problema

La navegación autónoma es una tarea ardua y compleja pues abarca varios subproblemas fundamentales:

En primer lugar, se encuentra la elección del tipo de movilidad que va a tener el robot para desplazarse. Desde ruedas convencionales hasta hélices, la elección del sistema de locomoción depende de las características del entorno en el que se va a mover el robot.

En segundo lugar, la percepción del entorno es un aspecto crítico para la navegación autónoma. Los sensores son los ojos y oídos de un robot, permitiéndole capturar información del entorno y tomar decisiones en función de esta información. Es necesario contar con sensores que proporcionen información precisa y confiable del entorno.

En entornos desconocidos esta información se debe analizar para identificar los obstáculos presentes y buscar aquellas zonas por donde el robot pueda transitar de forma segura.

Una vez analizado el entorno, es necesario planificar una trayectoria que permita al robot alcanzar su objetivo de manera eficiente y segura, teniendo en cuenta las

características del entorno y las capacidades del robot.

En último lugar actúa un sistema de control que genere las órdenes de control adecuadas para que el robot siga la trayectoria planificada de forma precisa. Esta debe de tener en cuenta las dinámicas y características físicas del robot.

Todos estos aspectos constituyen un campo de investigación en sí mismo. La capacidad de un robot para moverse de manera eficiente y segura en un entorno desconocido depende de la integración y el desempeño de estos componentes.

1.2. Motivación del proyecto

El presente proyecto de trabajo fin de grado busca proporcionar una base sólida para futuras investigaciones y aplicaciones prácticas en robótica móvil. Su propósito es ofrecer al personal investigador de la Universidad Miguel Hernández de Elche (UMH) una herramienta que facilite la tarea de navegación. De esta forma, se busca que la interacción humana se limite únicamente a definir una posición objetiva.

Para ello, se utilizarán los robots móviles disponibles en el laboratorio de Automática, Robótica y Visión por Computadora (ARVC) de la UMH. En particular, se empleará la plataforma móvil Husky A200 con sensores integrados.

La localización se abordará mediante técnicas de georreferenciación, utilizando el sistema de posicionamiento RTK-GPS, que proporciona una localización precisa del robot en entornos exteriores. Además, como herramienta para la percepción del entorno, se propone el uso de un sensor LiDAR tridimensional.

1.3. Objetivos

El objetivo principal de este proyecto consiste en desarrollar y validar un sistema de navegación que permita a un robot móvil desplazarse de manera autónoma en entornos complejos. Para alcanzar este objetivo, se plantean los siguientes objetivos específicos, que serán revisados nuevamente al final del proyecto:

- Realizar un estudio de los sistemas de control existentes que puedan generar las velocidades de salida necesarias para el robot, y seleccionar el más adecuado para el seguimiento de trayectorias.
- Programar la unidad de control en el entorno de ROS para obtener los datos de los sensores a partir del tópico correspondiente, asegurando una comunicación eficiente y confiable.
- Evaluar y mejorar el desempeño del sistema de control en un entorno simulado, utilizando herramientas de simulación y validación para optimizar su funcionamiento. Posteriormente, implementar el sistema en el robot y realizar pruebas en un entorno real.
- Desarrollar una interfaz de control intuitiva y amigable para el usuario, con el objetivo de facilitar la interacción con el robot. Se busca que el usuario pueda definir una posición objetivo y el robot se desplace de forma autónoma hasta alcanzarla.
- Implementar un sistema de evasión de obstáculos que permita al robot evitar colisiones y navegar de manera segura en entornos desconocidos. Esto implicará el uso de algoritmos y técnicas de percepción del entorno para identificar obstáculos y planificar trayectorias seguras.

1.4. Estructura del documento

A continuación y para facilitar la lectura del documento, se detalla el contenido de cada capítulo:

- **En el capítulo 1. Introducción:** se realiza una introducción al proyecto.
- **En el capítulo 2. Estado del arte:** se presenta un análisis del estado actual de la robótica móvil.
- **En el capítulo 3. Marco teórico:** se explica la base teórica de los conceptos aplicados en este proyecto.

- **En el capítulo 4. Herramientas utilizadas:** se describe la plataforma utilizada, incluyendo los sensores y la estructura de software.
- **En el capítulo 5. Control del seguimiento de trayectorias:** se detalla el desarrollo teórico de los controladores implementados. Además, se realiza un modelo del sistema en el entorno de Simulink para obtener las ganancias óptimas y seleccionar el sistema de control más adecuado para el robot móvil.
- **En el capítulo 6. Integración en ROS:** se describe la integración del controlador en ROS, el proceso de planificación de trayectorias y la simulación correspondiente.
- **En el capítulo 7. Integración en el robot real:** se describe la integración del sistema de control en el robot real y se presentan los resultados obtenidos.
- **En el capítulo 8. Evasión de obstáculos:** se aborda el proceso de evasión de obstáculos y se detallan los resultados obtenidos.
- **En el capítulo 9. Conclusiones y líneas futuras:** se presentan las conclusiones obtenidas a lo largo del proyecto.
- Al final del documento se incluye una lista de referencias **bibliográficas** utilizadas al igual que el **código** desarrollado en el proyecto.

2. ESTADO DEL ARTE

En las últimas dos décadas hemos visto avances significativos en el ámbito de la robótica móvil y su utilidad en una amplia gama de aplicaciones. En especial, las mejoras en la navegación autónoma han permitido el uso de robots en entornos complejos y dinámicos que antes eran inalcanzables o de peligro para el ser humano.

Para ser capaz de navegar de forma independiente en este tipo de entornos, un robot debe llevar a cabo tres tareas esenciales; crear un mapa, localizarse y obtener la ruta óptima, ver Figura 2.1. A continuación, se describe el enfoque adoptado en las últimas décadas para abordar estas tareas.

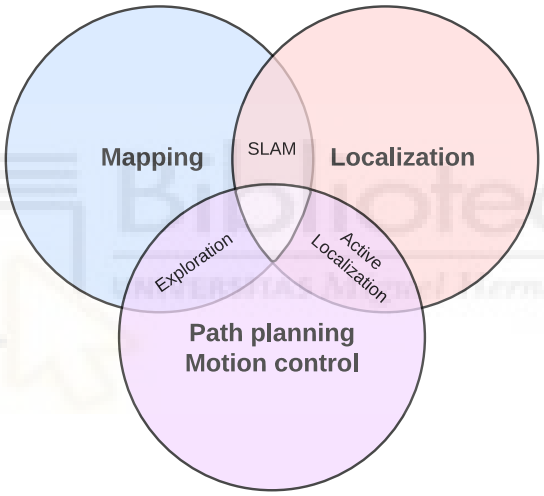


Figura 2.1: Tareas en la navegación autónoma.

2.1. *Mapping*

El *mapping* aborda el problema de obtener una representación espacial del entorno que rodea al robot mediante los sistemas de percepción que este posee. Desde sus inicios [1] este problema se ha abordado desde muchas perspectivas y con variedad de sensores. Aunque ciertos autores hayan utilizado sensores de ultrasonidos por su simplicidad [2] o, debido a su versatilidad, sensores de ondas electromagnéticas de radio [3], sin duda para la realización de esta tarea lo que predomina a día de hoy es la tecnología láser [4].

Mediante la emisión de rayos láser en distintas direcciones el sensor LiDAR (*Light Detection and Ranging*) es capaz de obtener un nube de puntos con gran precisión. Tras la trasformación consecutiva de estas nubes, se forma un mapa del entorno [5].

Dentro de este ámbito se encuentran diversos enfoques para su implementación, principalmente diferenciados por la cantidad de dimensiones que se tienen en cuenta. En primer lugar, en los mapas 2D un objeto estará representado por una ubicación espacial XY, obviando el eje Z. Este enfoque minimalista está presente en una gran cantidad de obras en las que se necesita únicamente esquivar obstáculos [6].

Por otro lado, utilizamos mapas 3D cuando necesitamos una completa representación del entorno. Las aplicaciones son diversas, desde [7] que utiliza un sensor LiDAR 3D para la navegación autónoma de un UAV (*Unmanned Air Vehicles*) hasta la robótica agrícola de precisión [8].

Como el hecho de agregar una nueva dimensión supone mayor coste computacional y aumento en la cantidad de datos, surgen nuevos métodos de segmentación conocidos como mapas 2.5D donde el eje Z se discretiza en niveles [9], facilitando su implementación en tareas de tiempo real.

2.2. Localización

Es esencial que un robot móvil pueda identificar su posición en el espacio para lograr una navegación completamente autónoma. La conciencia de su propia ubicación, junto con la de otros objetos o características relevantes en su entorno, constituyen la base fundamental para las operaciones de navegación de nivel superior. La precisión en la localización es crítica, ya que cualquier error en esta puede afectar la exactitud de las acciones subsiguientes.

Sus inicios los encontramos en AGVs (*Automated Guided Vehicle*) limitados a recorridos fijos [10]. Buscando mayor flexibilidad en la navegación; Leonard y Durrant-Whyte [11] emplearon “balizas geométricas” como zonas fácilmente reconocibles para el robot. Más adelante, Betke y Gurvits [12] implementaron marcas de referencia que permiten al robot calcular su propia posición y orientación.

Posteriormente vimos la aparición del *dead reckoning* [13], técnica que utiliza los sensores propioceptivos para el cálculo de la orientación y posición. Pero la acumulación de error con el tiempo hace inviable su aplicación como única fuente de información.

Por otro lado, durante las últimas dos décadas ha sido ampliamente estudiado el uso de sensores exteroceptivos en la tarea de localización. En primer lugar, tenemos el uso del SONAR (*Sound Navigation And Ranging*) mediante la recepción de ecos [14] y con una mayor precisión encontramos el LiDAR [15] y los sensores infrarrojos [16]. En segundo lugar tenemos el uso de sensores visuales para reconocer el entorno los cuales se emplean, por ejemplo, en la técnica VO (*Visual Odometry*) [17], la cual se presenta como una amplia gama de posibilidades en el ámbito de la localización [18].

En contrapartida, una tradicional forma de localizarse que ha sido ampliamente utilizada es el GPS (*Global Position Systen*) [19]. Desarrollado por el departamento de defensa estadounidense en 1973, este sistema es capaz de proveer información de la posición en cualquier punto de la superficie terrestre, y su mayor potencial de uso radica en aplicaciones que trabajan en entornos amplios de exteriores [20].

No obstante, el GPS sufre de algunas limitaciones como la generación de múltiples caminos [21] y la imposibilidad de uso en interior o en ambientes urbanos densos donde la señal es bloqueada. Por ello, usualmente vamos a encontrar el GPS junto con otros sensores con objetivo de solucionar estos inconvenientes o aliviar las desventajas que traen los propios sensores.

Con el objetivo de mejorar la precisión del GPS Zumberge *et al.* [22] proponen el método PPP (*Precise Point Positioning*) capaz de obtener una precisión de posicionamiento de centímetros con un único receptor GPS de doble frecuencia.

Sin embargo, este servicio actualmente no puede proporcionar rápidamente una alta precisión de posicionamiento. Este hecho promovió el desarrollo de RTK (*Real-Time Kinematic*) capaz de otorgar la posición precisa mediante correcciones en tiempo real respecto a una estación base [23]. Esta técnica permite el uso de GPS en aplicaciones de tiempo crítico como es el control de un robot móvil [24].

2.3. Navegación

2.3.1. *Path planning*

El primer paso en la navegación autónoma es encontrar un camino óptimo entre el punto de partida y el destino. Para resolver esta tarea conocida como *path planning* varios algoritmos han sido propuestos.

En primer lugar tenemos Dijkstra [25], el cual representa una búsqueda mediante nodos con pesos asignados en función de la distancia. Posteriormente surgieron versiones de este algoritmo con menor coste computacional [26]. En segundo lugar encontramos A-star [27] y su versión dinámica D-star, basado en la heurística para encontrar el camino más corto. Además D-star logra mayor eficiencia adaptando el camino al entorno en tiempo real. En tercer lugar tenemos RRT [28], donde a partir de un nodo inicial se realiza una expansión aleatoria de nodos hasta encontrar el punto objetivo. Con un tiempo de búsqueda reducido, esta técnica es útil para entornos con obstáculos complejos.

En [29] podemos encontrar un resumen con las ventajas y desventajas de cada uno de ellos.

2.3.2. Seguimiento de trayectorias

Una vez conocida la ruta que queremos seguir en una primera instancia, se debe aplicar una estrategia de control con la capacidad de generar una velocidad y dirección adecuada, proporcionando un seguimiento de trayectoria suave con un mínimo error. Muchas han sido las contribuciones al estado del arte de algoritmos de control robótico, principalmente categorizadas en modelos Geométricos, Cinemáticos y Dinámicos.

El modelo geométrico se caracteriza por ser el más simple, puesto que se basa únicamente en las dimensiones del vehículo y su posición. En esta categoría encontramos *Follow the Carrot*, el cual trabaja con la orientación actual del robot y la del *target*

point [30]. *Pure Pursuit*, con la misma filosofía traza un arco entre puntos para suavizar el control de seguimiento de trayectoria [31]. *Vector Pursuit* añade la búsqueda vectorial considerando la orientación y curvatura correcta otorgando mayor robustez [32].

Por otra parte, los modelos Cinemáticos tiene en cuenta el movimiento del vehículo con respecto a su posición, velocidad y aceleración pero no las fuerzas involucradas. Además Kanayama *et al.* [33] proponen una regla de control para encontrar velocidades lineales y rotacionales objetivo razonables en vehículos no holónomos.

Por último, los algoritmos Dinámicos adquieren mayor complejidad al tener en cuenta las fuerzas que van a estar involucradas en el movimiento. Al tener en consideración las dinámicas del vehículo, obtiene mejor rendimiento que las categorías anteriores, especialmente en altas velocidades [34].

Por otro lado tenemos controladores específicos que aunque no se ajustan a las categorías anteriores, son ampliamente usados.

- **PID:** como veremos en la Sección 3.2.1 este control clásico se basa en la retroalimentación del error y mediante el ajuste de tres términos (proporcional, integral y derivativo) logra un control preciso. Su simple estructura y el hecho de que no necesite conocer el modelo matemático del sistema lo han convertido en uno de los más utilizados [35].
- ***Fuzzy logic*:** se basa en la lógica difusa para tomar decisiones y ajustar las acciones del sistema por lo que es capaz de lidiar con muchas imprecisiones e incertidumbres. Es por ello que se ha vuelto popular en los vehículos autónomos [36].
- ***Sliding mode control (SMC)*:** genera una superficie de deslizamiento donde permanezca el sistema. Es ampliamente usado en sistemas no lineales ya que posee gran robustez ante perturbaciones y es capaz de converger rápidamente al camino [37].
- ***Model Predictive Control* (MPC):** es un potente controlador de bucle cerrado que utiliza el modelo matemático del sistema para predecir su situación

en el futuro dentro de un rango de horizontes. Puede descomponerse en cuatro categorías; lineal (LMPC), no lineal (NMPC), MPC robusto y MPC adatativo [38].

- ***Immersion and invariance*:** propuesto por Astolfi *et al* en 2003 [39] este método ofrece adaptabilidad y robustez. Se basa en las nociones de inmersión del sistema y *manifold* de invariancia, y no requiere el conocimiento de una función de Lyapunov.

2.3.3. Evasión de obstáculos

Una vez obtenida la trayectoria global que se quiere seguir, esta debe de ser modificada en tiempo real para evadir los obstáculos que aparezcan en el mismo. De forma que tendremos un planificador local que ante la detección de un obstáculo genere un camino alternativo que tras superar la dificultad se reincorpore a la ruta global.

En este contexto, la evaluación de la transitabilidad se presenta como un componente fundamental de la navegación autónoma.

Un enfoque clásico es utilizar los sensores incorporados para detectar objetos próximos y trabajar en función de la distancia a la que se sitúen, por ejemplo mediante sensores de proximidad [40] o LiDAR 2D [6].

No obstante, los avances de la tecnología nos permiten el uso de métodos más complejos para evaluar qué parte del terreno es navegable. En concreto, se está estudiando el uso del deep-learning como una potente solución al análisis de datos no estructurados como es una *point cloud* [41].

3. MARCO TEÓRICO

3.1. Tipos de sistemas de control

A continuación se muestra una clasificación de los distintos sistemas de control.

- **Según si existe retroalimentación o no**
  - **Lazo abierto:** este método se encarga de proveer instrucciones al equipo con el objetivo de obtener un comportamiento a la salida predeterminado.
  - **Lazo cerrado:** en contraste, en estos sistemas la señal de control se genera teniendo en cuenta la salida de este. De forma que se producen ajustes si la salida no es la deseada.
- **En función de su causalidad**
  - **Causales:** la salida depende únicamente de los valores presentes y/o pasados de la entrada, habiendo una relación de causalidad entre las salidas y las entradas del sistema.
  - **No causales:** cuando no se cumple esta propiedad y la salida depende de valores futuros.
- **En función de sus entradas**
  - **SISO (*Single-Input, Single-Outpt*):** una entrada y una salida.
  - **SIMO (*Single-Input, Multiple-Outpt*):** una entrada y múltiples salidas.
  - **MISO (*Multiple-Input, Single-Outpt*):** múltiples entradas y una única salida.
  - **MIMO (*Multiple-Input, Multiple-Outpt*):** múltiples entradas y múltiples salidas.

- **Según las ecuaciones que definen el sistema**
  - **Lineal:** cuando el sistema verifica el principio de superposición y la salida es proporcional a la entrada.
  - **No lineal:** las ecuaciones de control no son lineales entonces la relación entrada-salida no es proporcional.

3.2. Sistemas de control empleados

En este apartado se presentan los dos esquemas de control empleados en el trabajo. Ambos se clasifican como **causales** y con la presencia de retroalimentación. En cuanto al PID se trata de un controlador **lineal** cuya salida es predecible. Además es de tipo **SISO** por lo que será necesario de dos controladores PID para controlar nuestro sistema, el cual posee 2 entradas y 2 salidas. Por otro lado, el sistema de control basado en la función de Lyapunov es de tipo **no lineal y MIMO**. Entonces con un bloque controlaremos el sistema completo.

3.2.1. Control PID

Desde su origen en 1910 con la invención de Elmer Sperry y tras la publicación de los métodos de ajuste desarrollados por Ziegler y Nichols en 1942 [42], que simplificaron el proceso de sintonización del controlador PID, la popularidad del control PID ha aumentado hasta convertirse en el tipo de controlador más usado en la industria.

Como su nombre sugiere y como podemos ver en la Figura 3.1 el PID está formado por tres acciones de control.

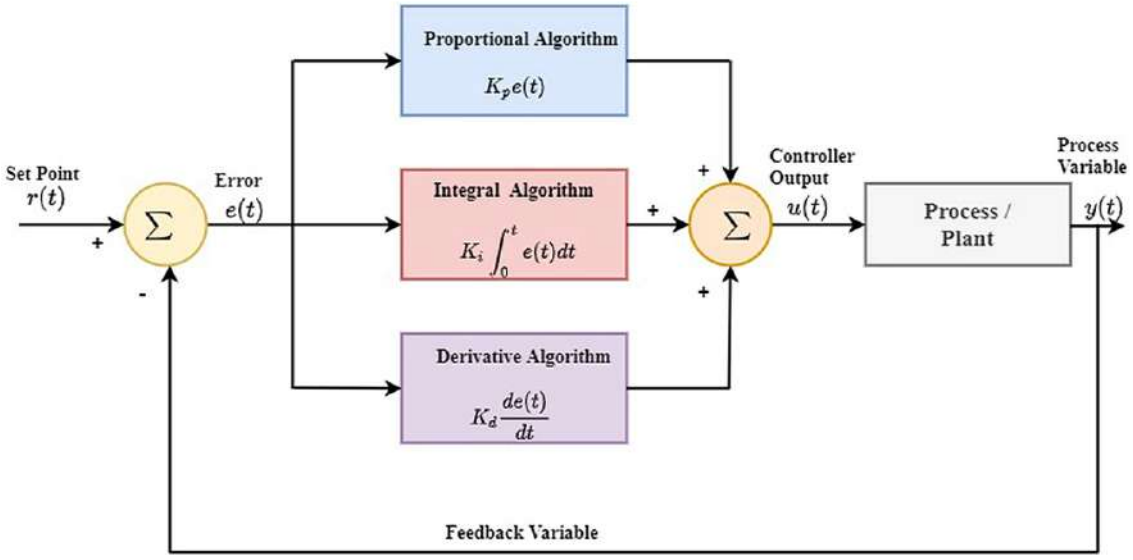


Figura 3.1: Diagrama de bloques de un control PID [43].

De las distintas configuraciones que existen, en este trabajo se ha utilizado un diseño en paralelo como el que se muestra en el diagrama de la Figura 3.1. La función de transferencia de este viene dada por la Ecuación (3.2) donde cada acción trabaja en términos separados y su efecto combinado viene expresado por la suma:

$$g(t) = K_P \cdot e\,(t) + K_I \cdot \int_0^t e\,(t) + K_D \cdot \frac{de\,(t)}{dt} \tag{3.1}$$

o en el dominio de la frecuencia:

$$G(s) = K_P + K_I \cdot \frac{1}{s} + K_D \cdot s \tag{3.2}$$

donde K_P, K_I, y K_D son las ganancias proporcional, integral y derivativa respectivamente.

3.2.1.1 Acción proporcional

De forma intuitiva, este término es directamente proporcional al error actual, de forma que cuanto mayor es la desviación actual, mayor será la corrección que se

aplique. Al aumentar K_P logramos mejorar la precisión y hacer un sistema más rápido, pero a costa de inestabilizarlo. Además, dado que siempre requiere un error distinto de cero para generar su salida, no puede anular el error en estado estacionario por sí solo. Ambas características se muestran en la Figura 3.2.

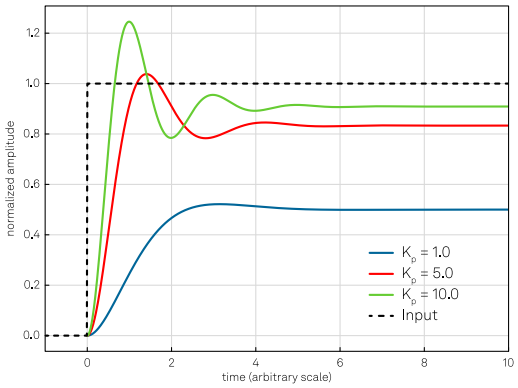


Figura 3.2: Efectos de la acción proporcional [44].

3.2.1.2 Acción integral

Este término examina el error a lo largo del tiempo. Ante un error constante en el tiempo adquirirá un mayor peso resultando en una corrección mayor. Con ello se logra compensar el error estacionario, lo cual no era posible con una acción proporcional, ver Figura 3.3 . Sin embargo, debido a su efecto acumulativo si el error persiste a lo largo del tiempo, el sistema puede volverse inestable, produciéndose oscilaciones alrededor del punto objetivo.

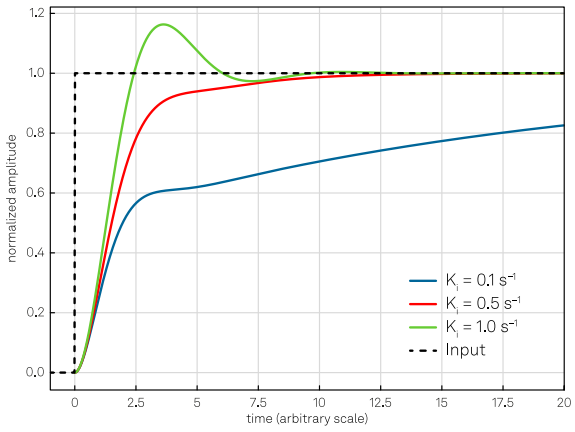


Figura 3.3: Efectos de la acción integral [44].

3.2.1.3 Acción derivativa

Mientras que el término proporcional trabaja con el error actual y el término integral con los valores pasados, la acción derivativa predice los futuros valores de error. Para ello se basa en la tasa de cambio de este, o lo que es lo mismo, en su derivada. Este efecto anticipado mejora la respuesta del régimen transitable, tal y como se comprueba en la Figura 3.4 aumentando la estabilidad siempre y cuando esté bien calibrada dado que es muy sensible a la señal de entrada.

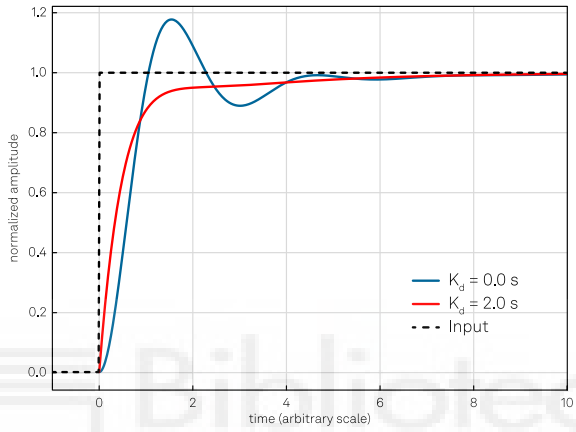


Figura 3.4: Efectos de la acción derivativa [44].

Por último, en la Tabla 3.1 podemos encontrar un resumen de los efectos independientes de cada ganancia.

Acción de control	Tiempo de respuesta	Sobreoscilación	Tiempo de establecimiento	Error estacionario	Estabilidad
Kp aumenta	Disminuye	Aumenta	Cambia poco	Disminuye	Empeora
Ki aumenta	Disminuye	Aumenta	Aumenta	Eliminado	Empeora
Kd aumenta	Cambia poco	Disminuye	Disminuye	Sin cambios	Mejora

Tabla 3.1: Efectos independientes de cada ganancia.

3.2.2. Control basado en la teoría de Lyapunov

Generar un control a partir de la teória de Lyapunov consiste en evaluar y asegurar la estabilidad asintótica del bucle de control. Existen dos principales vertientes para lograrlo: el método de linealización y el método directo.

El método directo tiene en cuenta las propiedades físicas del sistema. Su funcionamiento se basa en el principio de que la energía total del sistema (V) se disipa continuamente hasta llegar a un punto de equilibrio.

En primer lugar, se selecciona un candidato a función de Lyapunov $\dot{V}(x)$. Este candidato responde representa la energía cinética y potencial del sistema y debe de cumplir tres propiedades [45]:

1. $\dot{V}(x)$ es continua y tiene derivadas continuas.
2. Su punto de origen da como resultado un punto fijo estable $\dot{V}(0) = 0$.
3. $\dot{V}(x)$ siempre es positiva para todo $x \neq 0$.

En segundo lugar, se realiza la derivada de $\dot{V}(x)$ y se seleccionan las ecuaciones de control de forma que la derivada sea menor que 0. Esto nos garantiza la estabilidad asintótica de nuestro sistema aunque se trate de una acción de control no lineal.

En la Sección 5.1.3 veremos el desarrollo de estos pasos aplicados a las características físicas del robot empleado.

3.3. GPS-UTM

El receptor GPS utilizado provee la información de la posición en términos de latitud y longitud siguiendo un sistema de coordenadas geográfico WGS 84 (*World Geodetic System 1984*).

En él encontramos la tierra segmentada horizontalmente con los paralelos, siendo el ecuador la línea central que actúa como referencia. Además, perpendicular al ecuador tenemos los meridianos que convergen en los polos.

Para muchas aplicaciones este formato en grados es adecuado. Sin embargo, en el momento en el que hay que realizar cálculos usando las coordenadas de posición es preferible el uso del sistema de coordenadas UTM (*Universal Transversal de Mercator*).

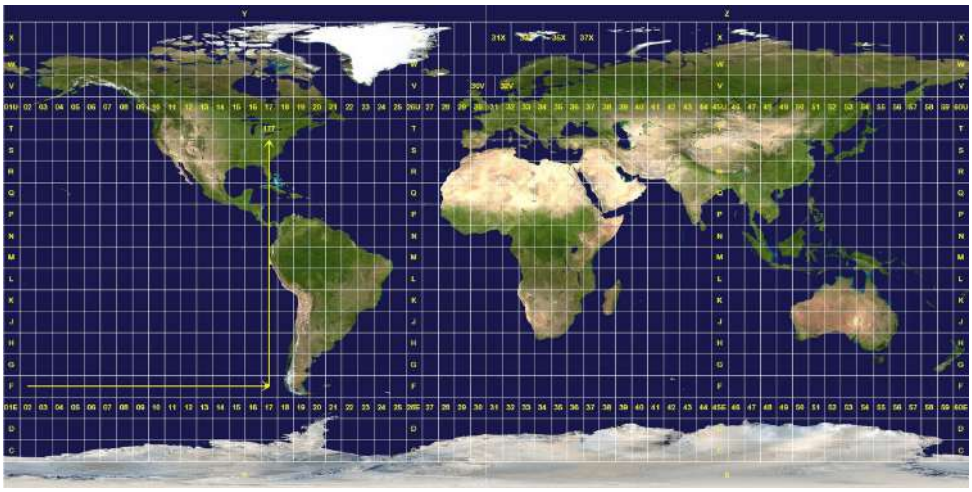


Figura 3.5: Sistema de coordenadas universal transversal de Mercator [46].

El sistema UTM realiza una proyección cilíndrica de la Tierra para poder representarla en una superficie plana. Es una proyección compuesta donde la esfera se representa como un conjunto de cuadrantes. Para ello, la Tierra se divide en 60 husos de 6^{o} de longitud y 20 zonas de 8^{o} de latitud. Los husos se identifican por números entre el 1 y el 60, mientras que las zonas lo hacen mediante letras, que van desde la C hasta la X, como se muestra en la Figura 3.5.

De esta forma, una posición en la Tierra viene dada por el número de zona UTM y el designador de hemisferio, así como por el par de coordenadas XY siguiendo el criterio de orientación ENU (East North Up).

Es posible la transformación de un sistema que trabaja con grados a una referencia UTM en metros. En primer lugar se identifica el cuadrante, su número asociado en función de la longitud y la letra a partir de la latitud.

Una vez obtenido el cuadrante, ya se puede calcular la distancia a la que nos encontramos respecto a su esquina inferior izquierda.

4. HERRAMIENTAS UTILIZADAS

4.1. Plataforma robótica Husky

La base robótica empleada para realizar las pruebas experimentales y validar los resultados del sistema de navegación es el Clearpath HUSKY A200 de la empresa Clearpath Robotics. Este ha sido utilizado en numerosas investigaciones y cuenta con el sistema operativo Ubuntu, así como un conjunto de librerías y herramientas de software que permiten el desarrollo de una amplia variedad de aplicaciones robóticas basadas en ROS, convirtiéndolo en un punto de partida ideal para nuevas investigaciones en el campo de la robótica.

Figura 4.1: Husky Clearpath A200.

Como vemos en la Figura 4.1 se trata de una plataforma de tamaño medio con una construcción robusta para poder realizar tareas en el exterior. Las especificaciones técnicas junto con las dimensiones del robot mencionado, pueden ser consultadas en la Tabla 4.1 y la Figura 4.2 respectivamente.

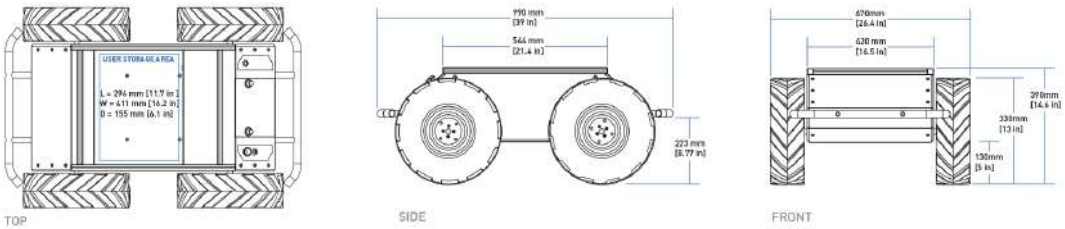


Figura 4.2: Dimensiones del Husky A200.

Masa	Kg	73,30
Longitud	m	0,99
Ancho	m	0,67
Alto	m	0,39
Ancho de vía	m	0,55
Momento de inercia alrededor del eje z	$kg\ m^2$	2,16
Velocidad lineal máxima	m/s	1,00
Velocidad angular máxima	rad/s	2,00

Tabla 4.1: Especificaciones ClearPath Husky A200.

Sus cuatro ruedas están montadas en una configuración cinemática diferencial clásica, donde la pareja de ruedas derecha e izquierda están unidas mecánicamente y accionadas por un motor eléctrico para cada pareja. Entonces, la cinemática del robot viene dada por la ecuación general para robots móviles no holónomos de tipo diferencial (4.1).

$$\dot{q} = \begin{bmatrix} \dot{x} \\ \dot{y} \\ \dot{\theta} \end{bmatrix} = \begin{bmatrix} cos(\theta) & 0 \\ sin(\theta) & 0 \\ 0 & 1 \end{bmatrix} \begin{bmatrix} v \\ w \end{bmatrix} \tag{4.1}$$

4.2. Sensores

La plataforma Husky ha sido diseñada para poder cambiar o mejorar sus características fácilmente. En esta sección se describen los distintos sensores que se han montado en ella.

4.2.1. Antena GNSS-RTK

Para obtener la posición actual del Husky, utilizaremos una configuración GNSS-RTK. Como se introdujo en el Capítulo 2, este servicio nos proporciona un posicionamiento preciso en tiempo real. Para lograr esto, se requieren dos receptores. En primer lugar, la antena base se encuentra en una posición conocida para corregir el error en tiempo real y aumentar la precisión. En segundo lugar, tenemos un receptor colocado en el robot que sirve como antena móvil. Para ambas antenas, se ha utilizado el modelo Harxon Smart Antenna TS100 que se muestra en la Figura 4.3.

Figura 4.3: Harxon TS100.

Su módulo GNSS multifrecuencial rastrea cualquier satélite visible con tecnología avanzada que resiste el problema de múltiples caminos. La antena rover obtiene información periódica de la base hasta una distancia de aproximadamente 1.5 km sin obstrucciones en el terreno.

En este trabajo se va a emplear el método RTK, no obstante estas antenas ofrecen dos tipos más de datos en función de la comunicación que haya entre ambas:

- *Standard Positioning Service* (SPS): es el modo de funcionamiento básico para cualquier antena GNSS el cual obtiene la posición mediante la estimación geométrica de al menos 4 satélites.
- *Differential Global Positioning System* (DGPS): mediante una comparación constante con la base, este método corrige el error de las mediciones GNSS obteniendo mayor precisión que SPS.

En la Tabla 4.2 se muestra la precisión de la posición horizontal en función del método de posicionamiento utilizado, así como el resto de especificaciones técnicas.

Señal recibida	GPS; GLONASS; BDS; GALILEO
Frecuencia máxima de posición	10 Hz
Precisión de posición horizontal (RMS)	SPS 1.5 m; DGPS 0.4 m; RTK 1 cm
Puertos de comunicación	Serial; Bluetooth
Puertos de comunicación	3G/4G; radio

Tabla 4.2: Especificaciones técnicas de las antenas GNSS TS100.

4.2.2. LiDAR

El LiDAR es un sistema de medición entre el sensor y los distintos objetos presentes en el entorno. Para ello consta de un transmisor que emite un haz de luz, un detector que recibe el mismo haz tras ser reflejado en cualquier objeto y un procesador que calcula el tiempo de vuelo (TOF).

Al utilizar rayos de luz, típicamente luz focalizada láser, obtiene mayor precisión, resolución y mediciones con más rapidez que sus hermanos RADAR y SONAR. Además, el uso de láser hace del LiDAR un sensor invariante a las condiciones de luminosidad por lo que se puede emplear en un mayor abanico de situaciones.

El sensor LiDAR se puede encontrar en tres variantes según sus dimensiones. En primer lugar, está la versión más simple que consiste en un sensor unidimensional utilizado para medir distancias puntuales. En segundo lugar, moviendo el haz de medición o rotando en un plano, se obtienen mediciones en una superficie. Por último, los escáneres multiposición permiten obtener una representación tridimensional, como se muestra en la Figura 4.4.

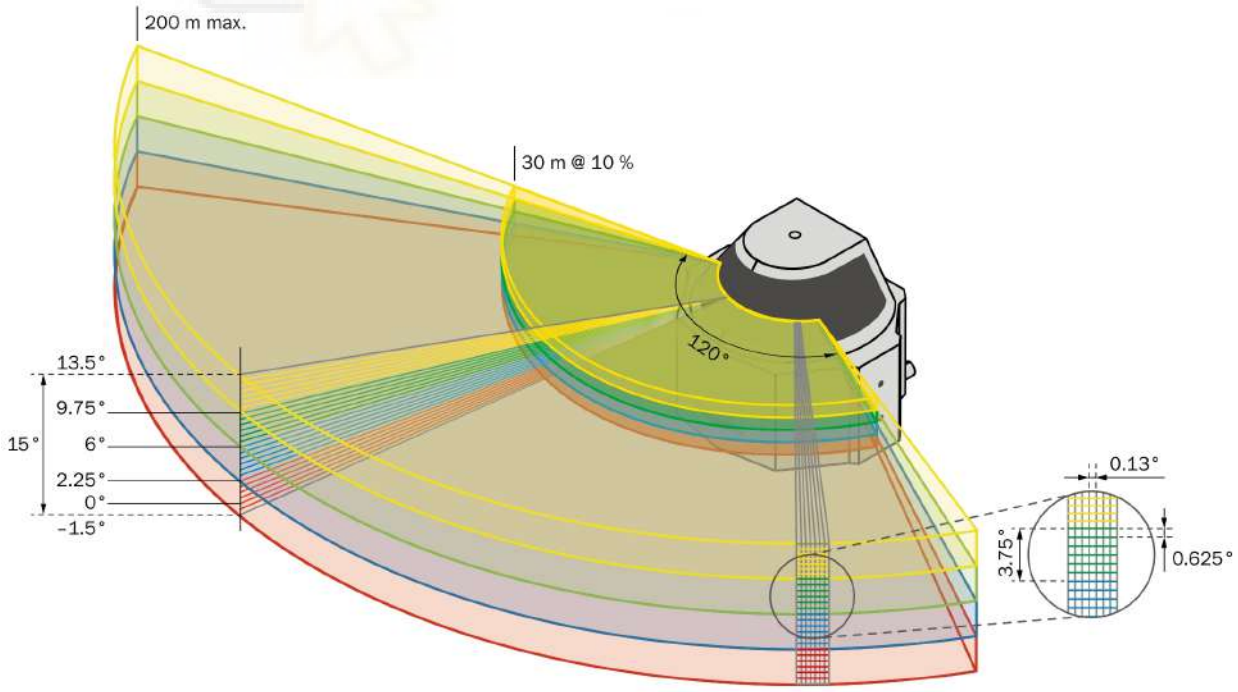


Figura 4.4: Funcionamiento de un LiDAR 3D del fabricante SICK [47].

Esta variante se puede conseguir con distintas técnicas, las más avanzadas emplean varios emisores y receptores para explorar simultáneamente líneas a varios ángulos. Es importante resaltar que la resolución de los sensores LiDAR 3D puede variar tanto en el número de planos (vertical) como en el número de puntos (horizontal), esto dependerá del modelo específico y de la configuración utilizada durante su operación. Esta variabilidad en la resolución puede tener un impacto significativo en la aplicación para la que se utilice el LiDAR.

Figura 4.5: LiDAR OS1-128.

En nuestro caso, para el desarrollo de este proyecto se ha empleado un LiDAR 3D OS1-128 desarrollado por la empresa Ouster, Inc, ver Figura 4.5. El nombre hace referencia a que el sensor cuenta con 128 canales en el eje vertical equispaciados cada 0,18 º lo que produce hasta 2.621.440 puntos por segundo. Esta impresionante resolución, combinada con las características detalladas en la Tabla 4.3, lo convierte en una herramienta de gran utilidad en el campo de la robótica de precisión.

Rango	120 m
Resolución vertical	128 canales
Resolución horizontal	512, 1024 ó 2048
Campo de visión vertical	45 º
Frecuencia de rotación	10 ó 20 Hz

Tabla 4.3: Especificaciones técnicas del LiDAR OS1-128.

4.2.3. Cámaras

Por otro lado, aunque no se vaya a hacer uso en este trabajo, la plataforma robótica empleada también cuenta con dos cámaras. Con el objetivo de obtener una visión de la escena mayor que las cámaras convencionales, se ha optado por cámaras omnidireccionales. Por un lado, contamos con una cámara Garmin con una lente *fisheye*, Figura 4.6, y por otro lado, una de tipo catadióptrica de la empresa Imaging Source, Inc, Figura 4.7. Estas cámaras permiten capturar imágenes desde diferentes perspectivas otorgando al robot mayor versatilidad.

Figura 4.6: Garmin VIRB 360.

Figura 4.7: Imaging Source DFK 33UX264.

4.2.3.1 Garmin VIRB 360

El tipo de lentes *fisheye* se caracteriza por tener una distancia focal corta y un amplio campo de visión, en este caso de 360º. Sin embargo, esto hace que se produzca una distorsión a medida que los objetos se alejan del centro de la imagen, donde las líneas rectas pasan a convertirse en curvas. Por lo tanto, es posible que haya que realizar un post procesado de las imágenes. Respecto a las características individuales de esta cámara tenemos:

Resolución	2496x2496 (2 archivos)
Campo de visión (FoV)	201,8º por lente
Velocidad de captura de imágenes	30 FPS
Peso	133 gramos

Tabla 4.4: Especificaciones técnicas Garmin VIRB 360.

4.2.3.2 Imaging Source DFK 33UX264

Este tipo de cámaras obtiene un mayor campo de visión gracias a la combinación de varios elementos refractores y superficies reflectantes (espejos) en su interior. Con ello se logra obtener grandes distancias focales en un diseño compacto, permitiendo fotografías desde un tipo macro hasta largas distancias. Respecto a las características particulares de esta cámara tenemos:

Resolución	1440x1080
Rango dinámico	8/12 bit
Velocidad de captura de imágenes	1 FPS (configurable)
Peso	65 gramos

Tabla 4.5: Especificaciones técnicas Imaging Source DFK 33UX264.

4.2.4. IMU

Con el último sensor que cuenta este robot móvil es una unidad de medición inercial (IMU) para medir su aceleración y velocidad, tanto lineal como angular, y orientación. Para ello hace uso de giróscopio, magnetómetro y acelerómetros de tres ejes combinados mediante un Filtro de Kalman extendido.

Figura 4.8: IMU UM7 de Redshiftlabs.

El modelo elegido es el UM7 de Redshiftlabs cuya imagen se muestra en la Figura 4.8 y sus características son las siguientes:

Frecuencia de la estimación del EKF	500 Hz
Precisión (pitch and roll) estática	$\pm 1^{o}$
Precisión (pitch and roll) en movimiento	$\pm 3^{o}$
Precisión (yaw) estática	$\pm 3^{o}$
Precisión (yaw) en movimiento	$\pm 5^{o}$

Tabla 4.6: Especificaciones técnicas UM7 de Redshiftlabs.

4.3. ROS

ROS (*Robot Operating System*) es un *middleware* presentado en el año 2009 [48] con el objetivo de actuar como marco de trabajo en el desarrollo e investigación de tecnologías relacionadas con la robótica. Este meta-sistema de código abierto, proporciona una capa de comunicaciones estructurada entre el software y el hardware aportando una mayor abstracción de los códigos. Su popularidad se refleja en la abundancia de aplicaciones y funcionalidades desarrolladas y respaldadas por una amplia comunidad.

Su funcionamiento reside en la comunicación entre **Nodos**, los cuales son ejecutables que realizan una tarea específica dentro del sistema, como el control de un sensor, la planificación de movimiento o el procesamiento de datos. Gracias a su enfoque modular, es posible trabajar de forma independiente con cada Nodo a la vez que existe una interconexión gestionada por un Nodo principal (**ROS Master**). La comunicación entre Nodos se realiza mediante la publicación de **Mensajes** en **Tópicos** o subscripción a estos. ROS permite programar un Nodo en distintos lenguajes como Python o *C++* y la posterior comunicación entre Nodos de distintos lenguajes. Esto es posible debido a que los mensajes que generan cada uno de ellos son estructuras de datos predefinidas en función del tópico al que vaya destinado.

Como podemos apreciar en la Figura 4.9, los Tópicos actúan como canales de comunicación unidireccionales donde los Nodos intercambian mensajes bajo un paradigma de publicación/suscripción.

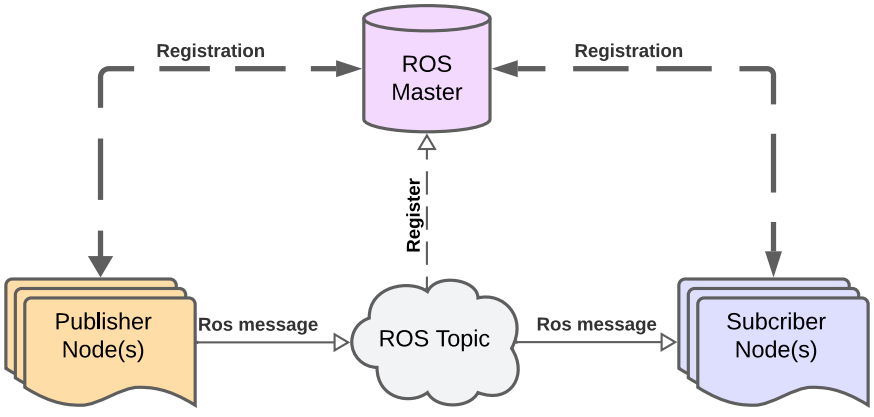


Figura 4.9: Ejemplo de comunicación mediante tópicos.

Este modelo es adecuado para intercambiar información entre muchos Nodos, pero en caso de querer realizar interacciones de solicitud/respuesta entre dos Nodos, ROS nos propone el uso de **Servicios**. Donde un Nodo denominado cliente realiza una petición a otro Nodo llamado servidor. Una vez realizada la acción el servidor le devuelve una respuesta.

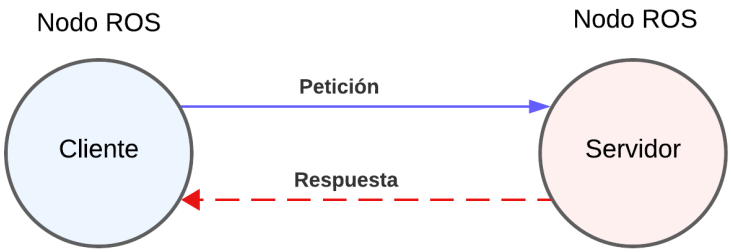


Figura 4.10: Ejemplo de comunicación mediante servicios.

Una vez visto como ROS trabaja en el nivel conocido como ROS *Computation Graph Level* podemos pasar analizar la organización de programas utilizada en ROS *Filesystem Level* que al igual que en un sistema operativo convencional, los archivos se encuentran organizados de cierta manera:

- **Paquetes:** son los componentes básicos de ROS pues contienen la programación de los Nodos creados y cualquier código necesario para su ejecución como puede ser: librerías, archivos de configuración, archivos de lanzamiento... Entre ellos destacan:
  - *CMakeLists.txt*: este es un *script* de *CMake* el cual proporciona instrucciones sobre cómo construir el paquete dentro del entorno ROS.
  - *Package.xml*: contiene los meta datos relacionados con el paquete por lo que es útil para describir y administrar sus propiedades.
- **Meta paquetes:** son usados para formar una agrupación de paquetes con la misma temática.
- **Repositorios:** son colecciones de paquetes que comparten un sistema de control de versiones (VCS) como *Git*.

- **Tipos de mensaje:** define la estructura de los mensajes utilizados en las comunicaciones de ROS.
- **Tipos de servicio:** define la estructura de datos petición/respuesta de un servicio.

Otra de las fortalezas de ROS son las potentes herramientas que nos ofrece para la visualización, análisis y registro de datos de un sistema. En concreto, ROS permite integrar programas de simulación, los cuales se han vuelto indispensables en el mundo de la robótica. El hecho de probar un nuevo algoritmo sobre hardware real puede resultar costoso y consumir mucho tiempo. Un entorno simulado con precisión siempre actúa como primera barrera ante problemas de diseño aportándonos seguridad y una rápida implementación.

Dentro de la gran variedad de software de simulación que existe actualmente, en este proyecto se han empleado los dos más populares Gazebo y Rviz.

4.3.1. Gazebo

Gazebo es una plataforma de simulación 3D que ofrece la posibilidad de realizar simulaciones de todo tipo de robots en entornos complejos. Con ello se puede estudiar el comportamiento del modelo en una recreación realista de cualquier tipo de entorno que se desarrolle.

El entorno que el software genera está conformado por dos elementos; el *mundo* y el *modelo*. El mundo comprende el entorno completo de simulación y su propósito es alojar todos los elementos que no constituyan el diseño del robot. En contraste, el modelo se introduce en el mundo y abarca todos los elementos que conforman el robot; chasis, ruedas, sensores... De esta forma se puede tanto probar un modelo robótico en varios mundos, como testear distintos robots en un mismo entorno.

Ambos elementos interactúan entre sí; el chasis puede colisionar con obstáculos, se pueden desplazar o rotar los elementos creados, los sensores como cámara y LiDAR pueden hacer mediciones sobre objetos del entorno y proporcionar los datos

generados en los mismos Tópicos de ROS que lo harían los sensores reales.

Además existe la opción de añadir *plugins* que actúen sobre una zona específica del mundo, modificando así el comportamiento del modelo. Entre ellos podemos encontrar entornos acuáticos, gravedad simulada en las tres dimensiones o entornos urbanos.

4.3.2. Rviz

Por otro lado, tenemos Rviz el cual es otro entorno de visualización de datos 3D. En concreto, con esta herramienta podemos obtener de forma visual los datos que los sensores han medido y publicado en sus correspondientes tópicos. La manera en la que se muestran los datos es muy intuitiva y personalizable, el usuario puede elegir en cada momento qué mensajes quiere mostrar y de qué forma.

Para hacer la interfaz más completa, al igual que en Gazebo, podemos extender las funcionalidades básicas de Rviz mediante *plugins*, como veremos en la Sección 6.2.1.

Tanto Rviz como Gazebo leen los archivos URDF para generar una representación tridimensional del robot. Estos archivos tienen un formato de lenguaje XML y en ellos se describe al detalle cada eslabón y articulación que conforman el robot, así como los sensores que incorpora.

4.4. Matlab

Matlab es un software matemático muy conocido y usado en el ámbito científico por sus inmensas capacidades. En él se combina un entorno de escritorio junto con un lenguaje de programación propio (lenguaje M). Matlab parte de la tarea básica de representación de datos pero su utilidad se abre a tareas de procesamiento de señal, *machine learning* o robótica y sistemas de control como es nuestro caso. Para ello se apoya en dos herramientas adicionales; GUIDE (editor de interfaces de usuario - GUI) y Simulink.

4.4.1. Simulink

Simulink es una entorno gráfico que incorpora Matlab para el desarrollo de simulaciones en sistemas dinámicos. Para ello hace uso de un diseño basado en modelos, donde el sistema se forma por la interconexión entre bloques.

Estos bloques contemplan todas las funcionalidades que el usuario pueda necesitar. Comenzando por funciones matemáticas y herramientas gráficas para la visualización de variables. Sus funciones se pueden ampliar mediante el uso de librerías específicas de la temática en la que se trabaje. Por ejemplo, en el ámbito de la robótica contamos con elementos que simulan el comportamiento cinemático de un robot así como los sensores que este posea y un entorno en el que moverse.

Así pues, Simulink permite al usuario construir tanto sistemas lineales como no lineales y evaluar su rendimiento en el dominio del tiempo (ya sea discreto o continuo) o en función del dominio de Laplace. Además, al ser una herramienta de Matlab, tiene disponibles las funcionalidades de este mientras se ejecuta una simulación.

5. CONTROL DEL SEGUIMIENTO DE TRAYECTORIAS

5.1. Modelo matemático

En esta sección se definen las reglas de control a partir del problema que supone el seguimiento de trayectorias para un robot diferencial no holónomo.

5.1.1. Planteamiento del problema

Considerando el escenario planteado en la Figura 5.1 donde el robot se encuentra en una posición arbitraria (x, y, θ) respecto al origen y a una distancia no nula de la posición deseada (x_d, y_d, δ).

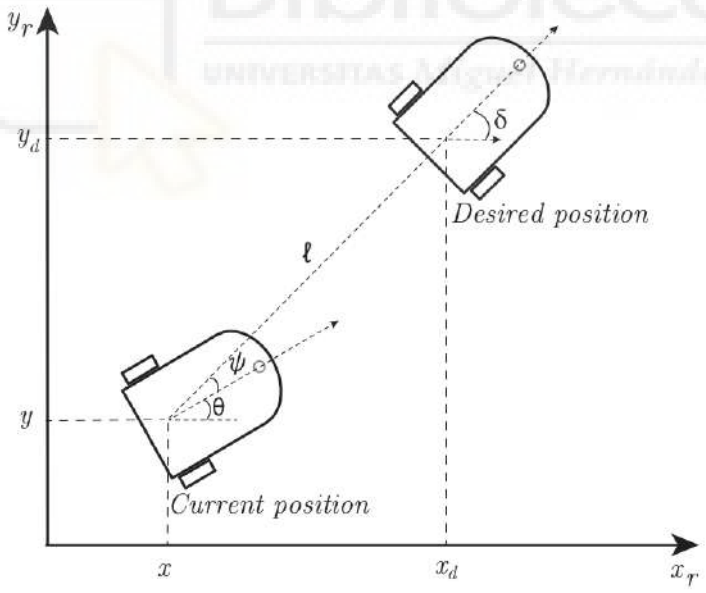


Figura 5.1: Posición actual y objetivo del robot [49].

Se define la diferencia entre ambas posiciones mediante dos valores. El error lineal d, obtenido como la distancia euclídea entre las coordenadas XY de ambas posiciones en la Ecuación (5.1). El error angular ψ, la diferencia que existe en las orientaciones del robot en el punto de partida y el destino. Estos parámetros se definen mediante

las siguientes ecuaciones matemáticas:

$$d = \sqrt{(x_d - x)^2 + (y_d - y)^2} \tag{5.1}$$

$$\delta = \arctan(\frac{y_d - y}{x_d - x}) \tag{5.2}$$

$$\psi = \delta - \theta \tag{5.3}$$

La función principal del controlador es generar acciones de control de tal manera que los errores asociados tiendan a cero a medida que transcurre el tiempo. Estas acciones de control se expresan en términos de velocidad lineal v y velocidad angular w. En otras palabras, el controlador diseñado debe establecer una relación entre los errores de entrada y las órdenes de salida de manera que se logre una transición suave y precisa entre posiciones.

5.1.2. Diseño PD

Se ha optado por utilizar la variante PD, en la cual se omite la parte integral para evitar una complejidad excesiva. Como se ha mencionado en la Sección 3.2.1.2, este término puede provocar oscilaciones en el sistema, lo cual es probable en esta tarea donde los datos de entrada son posiciones GPS que pueden contener ruido.

Por otra parte, la relación en el sistema de control es MIMO (multi-input, multi-output) pero como hemos visto en la Sección 3.2.1 el controlador PID se clasifica como SISO (single-input, single-output). Para remediar este problema se hará uso de dos controladores PD independientes, uno para regular la acción sobre la velocidad lineal y otro para la angular. Por tanto el sistema queda gobernado por las siguientes ecuaciones independientes:

$$\begin{cases} v = k_{pv}d + k_{dv}\dot{d} \\ w = k_{pw}\psi + k_{dw}\dot{\psi} \end{cases}$$

Donde k_{pv}, k_{dv}, k_{pw} y k_{dw} son los valores de ganancia asociados a cada término y son de valor fijo positivo.

5.1.3. Diseño Lyapunov

Un segundo controlador va a ser diseñado partiendo de una función de Lyapunov. La función candidata de Lyapunov que ha sido empleada para el diseño es la representada por la Ecuación (5.4).

$$V = V_1 + V_2 = \frac{d^2}{2} + \frac{\psi^2}{2} \tag{5.4}$$

Como podemos apreciar, cumple los tres requisitos impuestos mencionados anteriormente; $V > 0$, $V = 0$ solo cuando se anula por completo el error y V es continua para todo valor de d, ψ. Siguiendo el método presentado en la Sección 3.2.2 realizamos su derivada, resultando en:

$$\dot{V} = d\dot{d} + \psi\dot{\psi} \tag{5.5}$$

Una vez realiza la derivada empleamos la relación entre la variación del error y las velocidades de la Ecuación (5.6) para expresar $\dot{V}$ en función de las salidas v, w.

$$\begin{bmatrix} \dot{d} \\ \dot{\psi} \\ \dot{\delta} \end{bmatrix} = \begin{bmatrix} -\cos(\psi) & 0 \\ \sin(\psi)/l & -1 \\ \sin(\psi)/l & 0 \end{bmatrix} \begin{bmatrix} v \\ \omega \end{bmatrix} \tag{5.6}$$

Sustituyendo en (5.5):

$$\dot{V} = d\left[-v\cos(\psi)\right] + \psi\left(\frac{v\sin(\psi)}{d} - \omega\right) \tag{5.7}$$

Por último, seleccionamos los valores de v y w para que se verifique la cuarta condición $\dot{V} < 0$:

$$\begin{cases} v = k_d d\cos(\psi) \\ \omega = k_d\cos(\psi)\sin(\psi) + k_\psi\psi \end{cases} \tag{5.8}$$

Donde k_d y k_ψ son ganancias cuyo valor será fijo y positivo. Entonces, si sustituimos estas expresiones en la Ecuación (5.7) y agrupamos términos, $\dot{V}$ toma la forma:

$$\dot{V} = -k_d d^2\cos^2(\psi) - k_\psi\psi^2 \tag{5.9}$$

Se puede observar cómo la derivada de la función, Ecuación (5.9), siempre es negativa, cumpliendo así la cuarta y última condición. Esto garantiza un control asintóticamente estable en el seguimiento de trayectorias.

Antes de su implementación, podemos realizar un análisis previo de las formas que tendrán las acciones de control en función del error. Para ello, realizamos un barrido en el error angular en el rango $[-\pi, \pi]$ fijando los valores de ganancia y error en la distancia.

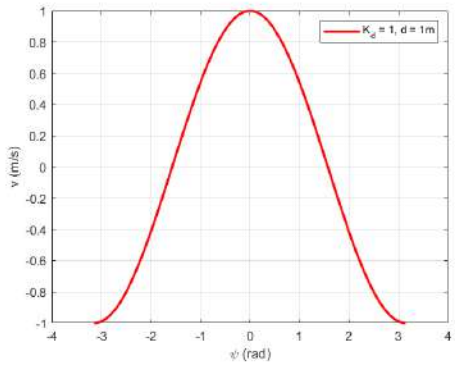


Figura 5.2: Variación de v para una ganancia y error d fijo.

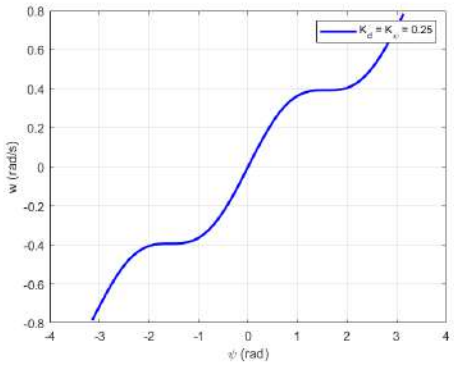


Figura 5.3: Variación ω para ganancias fijas.

En la Figura 5.2 se muestra cómo la señal de velocidad lineal está influenciada tanto por el error lineal como por el error angular. Cuando hay una gran diferencia entre la orientación actual y la orientación objetivo, se aplica una restricción a la velocidad lineal del vehículo. Esto permite realizar giros pronunciados de manera más suave y en un espacio más reducido.

En cuanto a la velocidad angular, los valores describen un comportamiento lineal en el rango central, un comportamiento constante en el siguiente tramo y un aumento progresivo en casos de gran error. Este comportamiento es más interesante que una acción puramente proporcional a la entrada.

5.2. Modelo en Simulink

En este apartado se realizará un modelado del robot en el entorno de Simulink con el objetivo de obtener los valores de ganancia de cada controlador. Es de gran importancia una correcta selección de estos parámetros para garantizar un seguimiento de trayectorias suave a la par que preciso. Para su elección, se analizará la variación en las velocidades, posiciones y errores durante la ejecución del control en distintos caminos.

Para realizar el modelado adecuado del robot, es fundamental comprender las condiciones físicas del sistema, especialmente las dimensiones del propio robot. Las medidas relevantes en el proceso de modelado de un robot diferencial son la distancia entre las ruedas, que denominaremos "L", y el radio de las ruedas "R". Los valores de estas dimensiones para el robot Husky se encuentran en la Tabla 5.1.

R	165 mm
L	535 mm

Tabla 5.1: Radio de las ruedas y ancho del robot

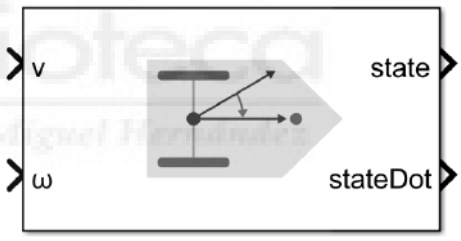


Figura 5.4: Differential Drive Kinematic Model.

Para esta tarea emplearemos el bloque `Differential Drive Kinematic Model` de la librería *Robotics System Toolbox*. Este bloque simula las dinámicas de un vehículo con tracción diferencial de forma que proporciona la posición y orientación actuales a partir de las velocidades lineal y angular.

Por otro lado, se ha diseñado un bloque algebraico para obtener los errores en posición, Figura 5.5. Este posee como entrada la posición deseada y está retroalimentado con la posición actual del robot proporcionada por el modelo diferencial. A partir de estos datos, se obtienen los errores implementando en Simulink las Ecuaciones (5.1) a (5.3).

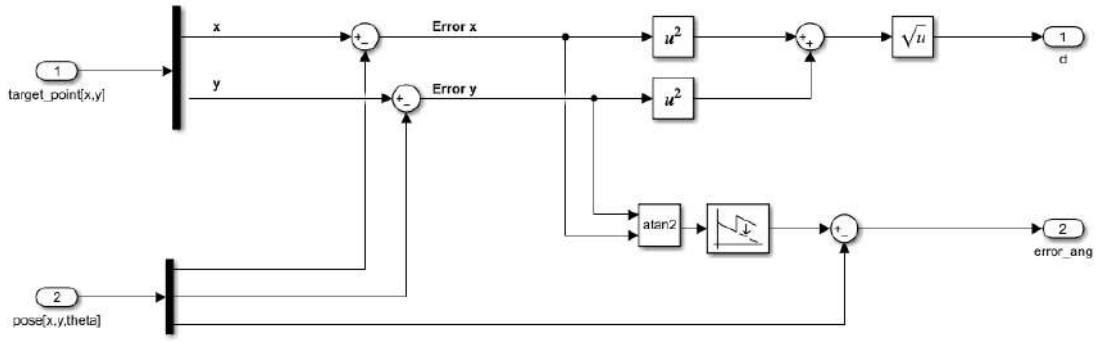


Figura 5.5: Diagrama de bloques para obtener el error de posición.

Los caminos están formados por un conjunto de puntos intermedios conocidos como *waypoints*. Desde el Workspace de Matlab le proporcionamos el camino a seguir (*Path*) en forma de matriz de coordenadas. Para proporcionar el punto XY objetivo, se ha diseñado el bloque `Next_waypoint`, Figura 5.6. Una vez se dé por alcanzado el *waypoint* actual, el bloque proporciona a su salida la siguiente referencia. Además, cuando se alcance el último *waypoint* de la lista se parará la simulación con el objeto STOP.

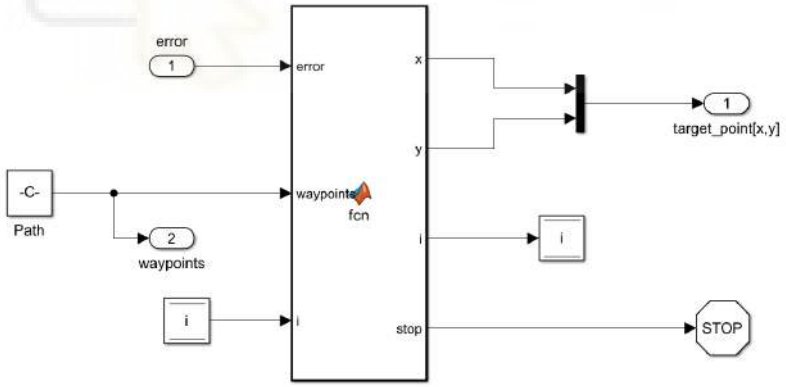


Figura 5.6: Diagrama de bloques para obtener el *waypoint*.

Por último, para la representación visual del robot siguiendo la trayectoria se ha usado el bloque `Robot visualizer`. Este nos permite visualizar el robot en movimiento y los *waypoints* a alcanzar.

Una vez tenemos el entorno de simulación listo se realiza el proceso de "*tuning*". Este proceso consiste en encontrar los valores óptimos de los parámetros del controlador

para minimizar el error y asegurar que la variable de control se alinee con los cambios dinámicos del sistema.

5.2.1. *Tuning* del control PD

El sistema completo de Simulink toma la siguiente forma para el caso del controlador PD.

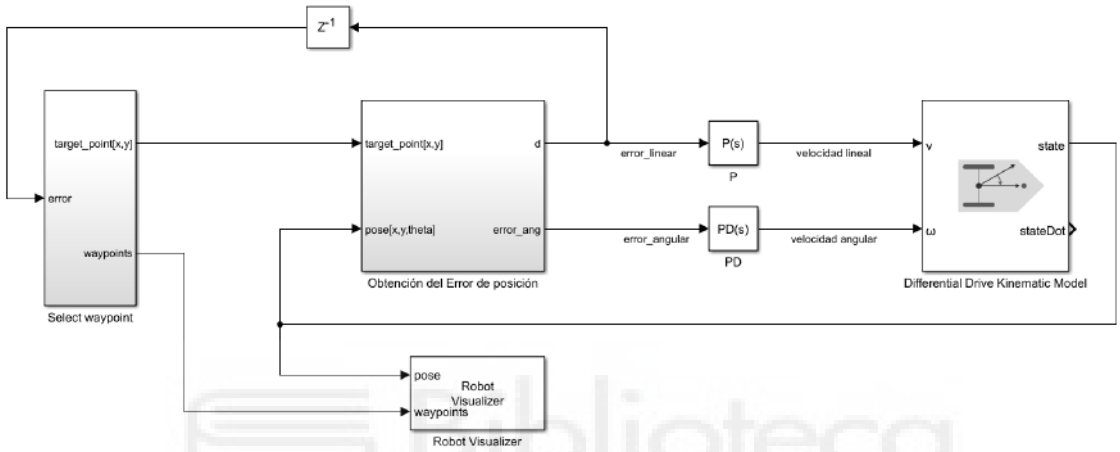


Figura 5.7: Sistema de control PD

Tras un análisis del comportamiento del robot sobre distintas trayectorias y haciendo uso de la herramienta PID tuner, se ha llegado a la conclusión que los parámetros más indicados para este sistema son los presentes en la Tabla 5.2.

	K_P	K_D
PID_v	0.2	0
PID_ω	0.242	0.15

Tabla 5.2: Valores de ganancia PID.

Como se puede apreciar, se ha optado por una configuración tipo P para el caso de la velocidad lineal, puesto que no es necesario un anticipación del error lineal en instantes futuros. El comportamiento del error lineal es muy predecible.

Una prueba de validación cuyo objetivo es alcanzar el punto [4,4] desde el punto de origen [0,0] y una orientación puramente horizontal se muestra a continuación.

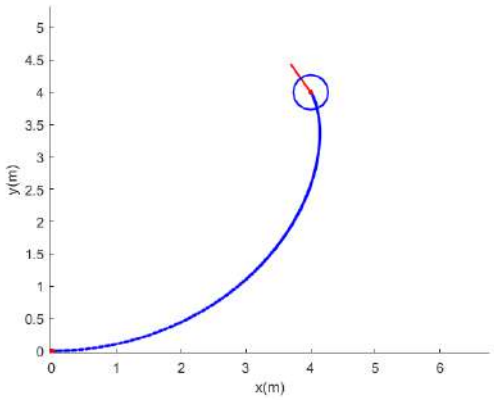


Figura 5.8: Trayectoria de la prueba PD.

En la Figura 5.8 vemos que la trayectoria seguida por el robot ha sido suave y estable. Para un mayor análisis encontramos en las Figuras 5.9 y 5.10 la evolución de las velocidades lineales y angulares adquiridas por el robot durante la trayectoria y los errores de ambas respectivamente.

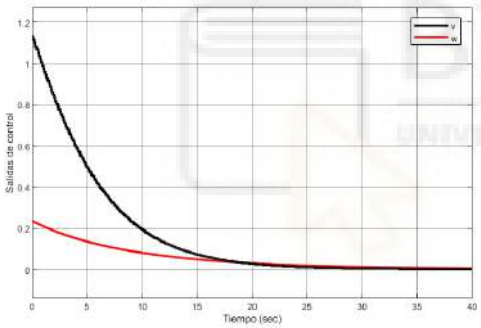


Figura 5.9: Velocidades prueba PD.

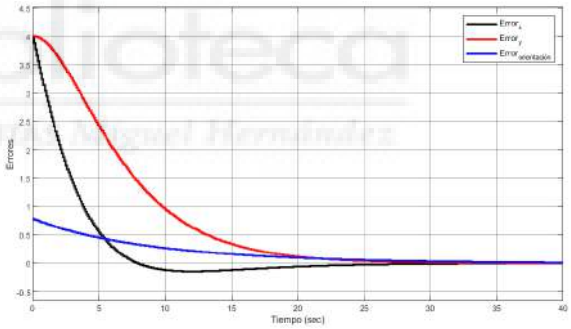


Figura 5.10: Evolución de los errores.

En la Figura 5.10 se puede apreciar cómo los errores tienden a cero y por consiguiente la velocidad del robot se reduce hasta alcanzar el punto. Estos resultados demostraron la funcionalidad del controlador implementado, validando el proceso de optimización.

5.2.2. *Tuning* del control de Lyapunov

En cuando al control no lineal obtenido por Lyapunov no se consta de ningún bloque que contemple sus ecuaciones. Por lo que se ha realizado el subsistema `Controlador Lyapunov`, ver Figura 5.11, siguiendo las ecuaciones presentadas en (5.8).

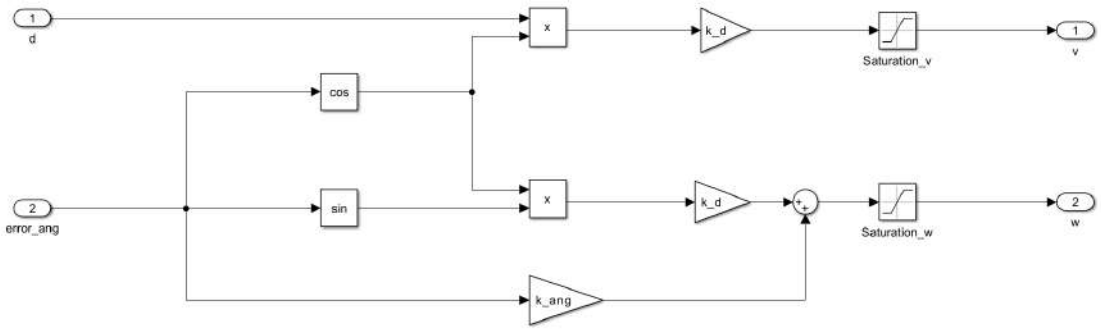


Figura 5.11: Modelo de bloques para la unidad de control no lineal.

Entonces el sistema completo para este esquema quedaría de la siguiente forma:

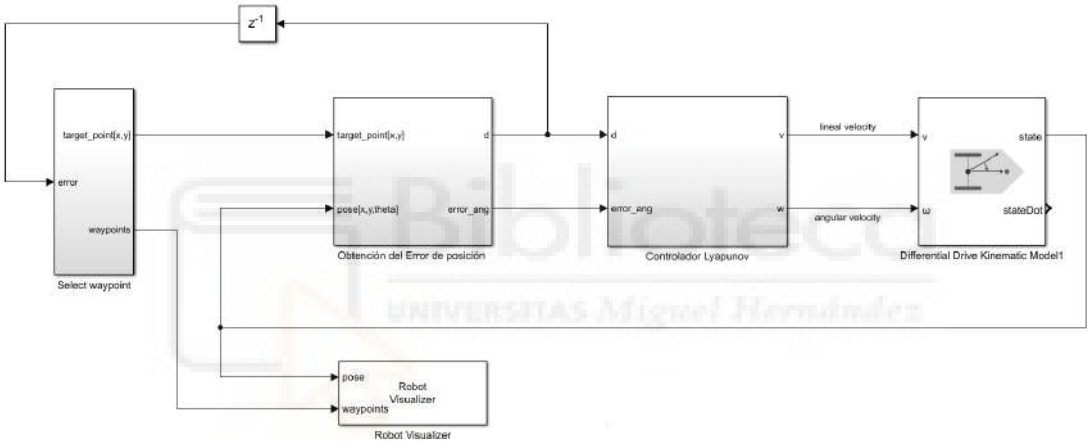


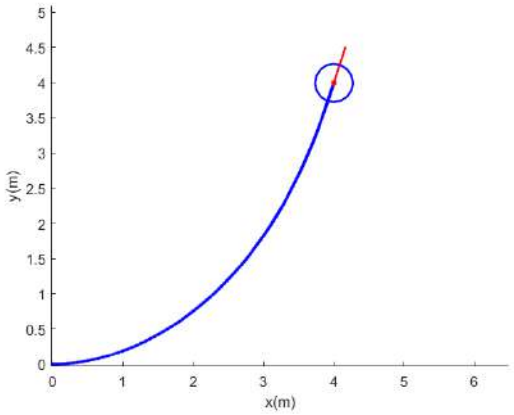
Figura 5.12: Sistema completo de seguimiento de *waypoints* con control Lyapunov.

Tras el análisis del desempeño de este sistema para distintos caminos, se han elegido las ganancias de la Tabla 5.3.

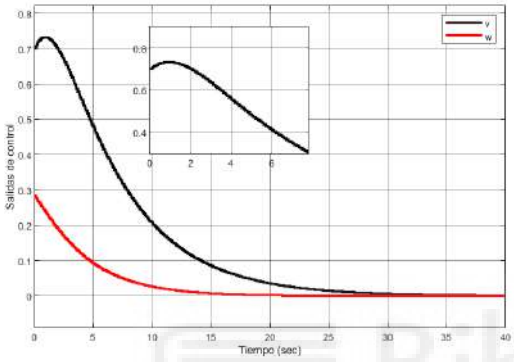
K_d	0.175
K_ψ	0.25

Tabla 5.3: Valores de ganancia para Lyapunov.

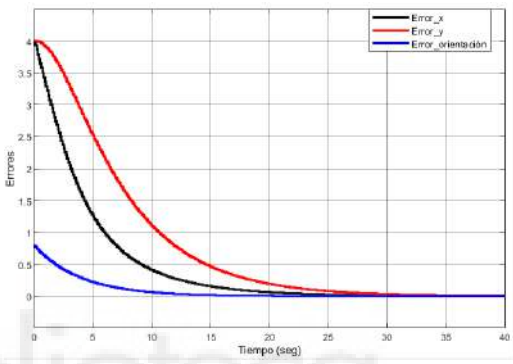
Al igual que con el PD, realizamos una prueba de validación, para la que se obtienen los siguientes resultados:



(a) Trayectoria.



(b) Velocidades.



(c) Evolución del error.

Figura 5.13: Resultados para un punto objetivo.

Si nos fijamos en los valores de velocidad obtenidos, podemos observar el acoplamiento entre la velocidad angular y la velocidad lineal. En los primeros instantes, cuando el error angular es grande, la velocidad lineal se ve limitada por la función coseno, como hemos visto en la Ecuación (5.8). Esto resulta en giros suaves y amortiguados.

Esto supone una gran ventaja frente al controlador tipo PD que como hemos visto en la Figura 5.13(b) al haber un pico tanto en el error de distancia como en el de orientación, ambas velocidades van a ser grandes en un primer momento, dando como resultado giros pronunciados. Esta característica que los diferencia la veremos representada en las distintas trayectorias realizadas.

5.2.3. Comparación de los controladores en distintos caminos

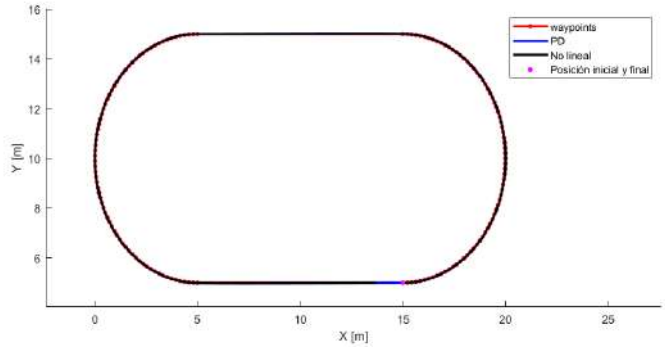
A continuación se va a someter a los dos sistemas de control a distintas trayectorias para evaluar su rendimiento.

- **Precisión**: se dará por alcanzado un *waypoint* cuando el robot se encuentre a una distancia euclídea de 10 cm.
- **Límites**: como hemos visto en la Tabla 4.1 la plataforma Husky tiene unos límites de velocidad de $[-1, 1]$ m/s y $[-2, 2]$ rad/s. Dentro de estos límites se ha comprobado que un buen rango de trabajo para el seguimiento de trayectorias puede ser [0.1, 0.3] m/s y $[-0.35, 0.35]$ rad/s. Para limitar las salidas de control a estos valores se ha utilizado los bloques de saturación de Matlab.
- **Frecuencia**: para la simulación se ha utilizado un periodo de muestreo de 0.1 segundos, lo que equivale a una frecuencia de 10 Hz. Con el objetivo de simular la obtención de datos mediante GPS.
- **Error XTE**: para evaluar el desempeño de los controladores se ha medido el error de desviación transversal (*Cross Track Error*, XTE) que posee el robot durante el recorrido. El trazado que une el *waypoint* previamente superado con el nuevo punto de destino, es el recorrido que nosotros pretendemos seguir. El XTE nos indica en cada momento a qué distancia nos encontramos de ese camino ideal.
- **Valor medio del XTE**: una vez obtenida la evolución temporal del error empleamos su valor medio como índice de rendimiento.

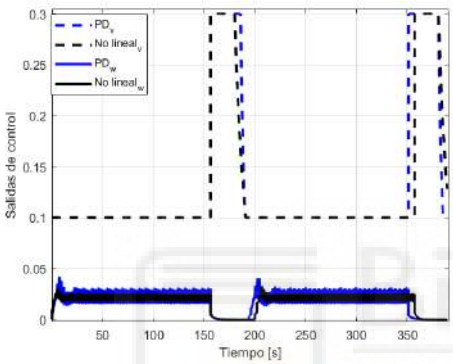
$$\mathrm{XTE}_{\mathrm{medio}} = \frac{1}{n}\sum_{i=1}^{n}\mathrm{XTE}_i \tag{5.10}$$

Donde n es el número total de valores de XTE recolectados durante la simulación y XTE_i es el valor del XTE en el i-ésimo punto de muestreo.

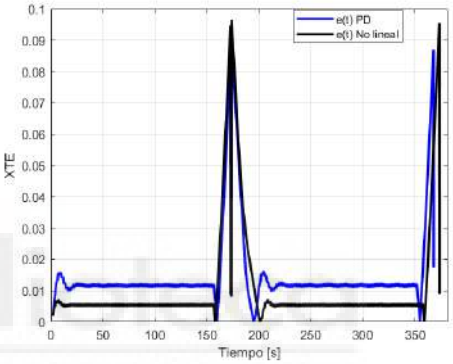
Para comenzar, tenemos una trayectoria elíptica compuesta por dos líneas rectas paralelas unidas por dos arcos en sus extremos.



(a) Trayectoria.



(b) Salidas de control.

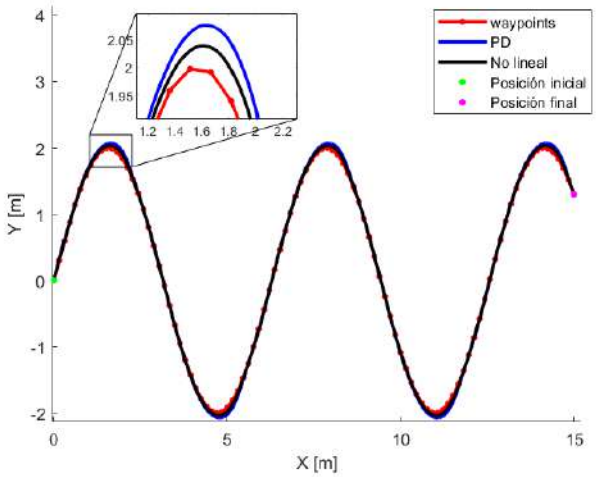


(c) Evolución del error.

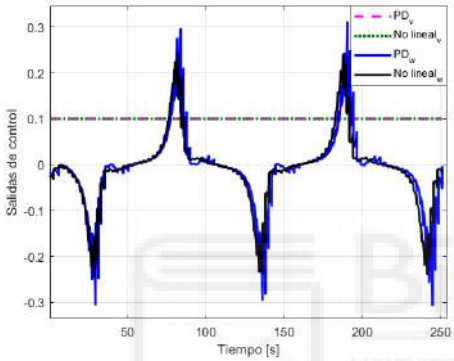
Figura 5.14: Resultados para una trayectoria elíptica.

En la Figura 5.14(a) se puede ver cómo el seguimiento de la ruta por parte de ambos controladores ha sido perfecta. Las curvas se han realizado a una velocidad angular prácticamente constante y una lineal mínima, ver Figura 5.14(b). Una vez llegamos al tramo recto, la velocidad angular se anula y la lineal toma su valor máximo. Cabe mencionar que el controlador no lineal ha obtenido un error menor que el PD, ver Figura 5.14(c).

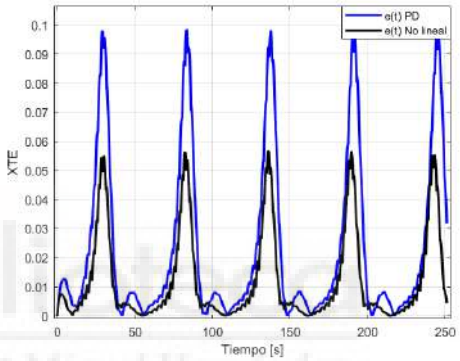
El siguiente caso consiste en un trayecto sinusoidal donde el robot debe de estar constantemente ajustando su orientación.



(a) Trayectoria.



(b) Salidas de control.

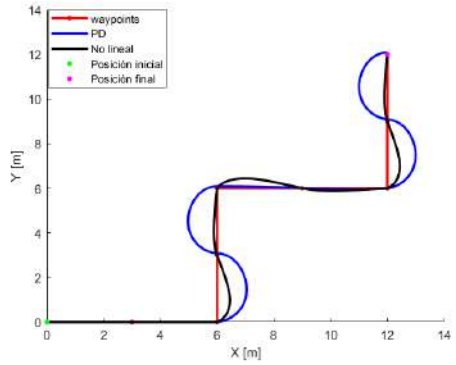


(c) Evolución del error.

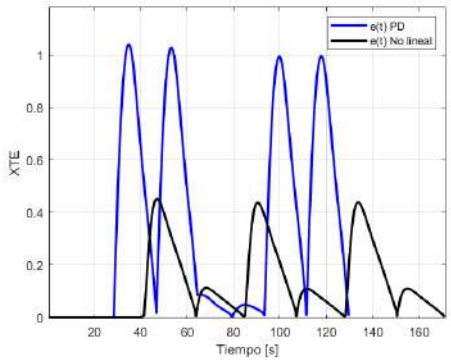
Figura 5.15: Resultados para una trayectoria sinusoidal.

Si observamos la Figura 5.15(a), podemos ver cómo en los tramos donde se requiere un giro más pronunciado, el controlador no lineal toma ventaja sobre el controlador PD. Este hecho se refleja en la Figura 5.15(a), donde se obtiene un error menor en comparación con el controlador PD, que es casi el doble.

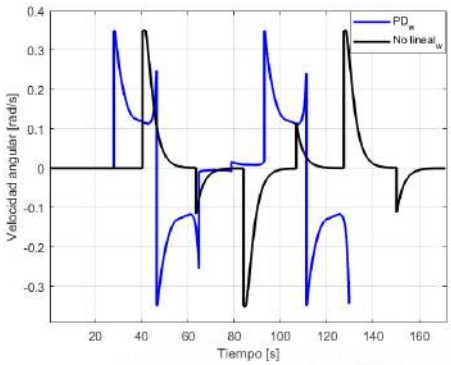
Por último, se ha realizado una prueba más exigente donde el recorrido cuenta con giros de 90^{o} asemejando un posible camino que deba de seguir un robot en la vida real.



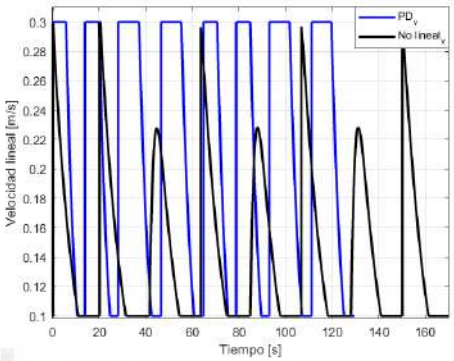
(a) Trayectoria.



(b) Evolución del error.



(c) Velocidad angular.



(d) Velocidad lineal.

Figura 5.16: Resultados para trayectoria con giros de 90º.

Cuando se tiene que realizar el giro de 90º tanto el error en distancia como el angular se encuentran en valores máximos. En el caso del PD esto implica un pico en ambas velocidades lo que resulta en giros muy pronunciados, alejándose hasta 1 metro de la ruta deseada, Figura 5.16(a). En el caso del no lineal al haber un pico en el error angular, se reduce la velocidad del vehículo para obtener giros balanceados, Figura 5.16(d).

Controlador	Elíptica	Sinusoidal	Con giros rectos
PD	0.0143	0.0236	0.3496
NL	0.0104	0.0125	0.1230

Tabla 5.4: Tabla de errores XTE_{medio} (m).

En la Tabla 5.4 se presenta el error promedio obtenido para cada simulación. Esta representa la distancia promedio que ha tenido el robot respecto a una trayectoria

ideal. En la tabla se puede apreciar como el PD genera valores de error más elevados en comparación con el control no lineal, y esta discrepancia se incrementa con el aumento de la dificultad del circuito. Evaluando el desempeño de ambos controladores llegamos a la conclusión de que el control no lineal aportará resultados más satisfactorios en futuras implementaciones. Por lo tanto, las secciones subsiguientes del proyecto se centrarán exclusivamente en este controlador.

6. INTEGRACIÓN EN ROS

En este capítulo se detalla la integración del controlador en el entorno de desarrollo de ROS, teniendo como base los fundamentos teóricos y prácticos explicados previamente.

Para realizar una simulación realista del comportamiento de un robot es necesaria su descripción mediante un archivo de descripción URDF. Este contiene todas las características que describen el robot; la representación visual, el modelo de colisión que presenta y el modelo cinemático y dinámico. Dado que el Husky Clearpath se trata de un robot comercial, el fabricante nos proporciona su archivo URDF.

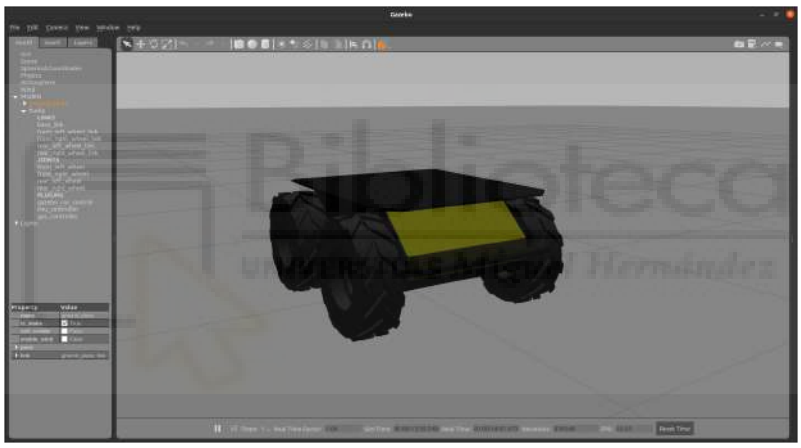


Figura 6.1: Modelo del robot en Gazebo

En la Figura 6.1 se presenta la visualización del Husky dentro del simulador Gazebo. La ejecución de esta representación se ha realizado con un archivo de configuración *launch*. Estos permiten la ejecución de múltiples Nodos de forma directa, especificando los parámetros y argumentos de entrada necesarios. Así simplemente habría que lanzar desde una terminal:

```
$ roslaunch nombre_paquete nombre_archivo.launch
```

6.1. Nodo de control

Como se mencionó en la Sección 4.3, las tareas en ROS se llevan a cabo a través de entidades conocidas como Nodos. En este contexto, hemos desarrollado un Nodo específico en Python para la tarea de control.

Este Nodo de control opera en base a dos entradas principales: la posición actual del robot y la posición objetivo a la que debe llegar. A partir de estas entradas, el Nodo calcula las velocidades necesarias que el robot debe adoptar para alcanzar la posición objetivo.

En términos de ROS, esto se traduce en que el Nodo se suscribe a un tópico que proporciona los datos de posición y, a su vez, publica las velocidades calculadas en un tópico destinado a las velocidades del robot, ver Anexo A.1.

- `/navsat/fix`: es el tópico donde se publican las coordenadas GPS actuales del robot y **NavSatFix** es el tipo de mensaje que utiliza para la comunicación, del cual nos interesan los siguientes datos:
  - Float64 latitude
  - Float64 longitude
- `/husky_velocity_controller/cmd_vel`: es el tópico empleado en Husky para publicar sus velocidades. El tipo de mensaje que utiliza es **Twist** y se compone principalmente por:
  - Vector3 linear
  - Vector3 angular

Se puede observar que el tópico nos proporciona la posición en grados (latitud, longitud). Para realizar la conversión a coordenadas en metros, se ha utilizado la librería *utm* de Python. Una vez que tenemos los datos de entrada, realizamos los cálculos matemáticos de las Ecuaciones (5.1) a (5.3) en Python.

En lo que respecta a la orientación actual del robot, se podría obtener directamente con el sensor IMU. No obstante, como se ha comentado previamente, este acumula mucho error haciendo inutilizables sus datos a largo plazo. Como solución, se ha optado por calcular la orientación como el ángulo entre la posición anterior y la actual. Para ello empleamos la función *atan2*, al igual que se hace con la orientación destino:

```
1     o_act = math.atan2((y-y_ant),(x-x_ant))
2     o_des = math.atan2((y2[0]-y),(x2[0]-x))
```

Para emplear este método de forma satisfactoria hay que lidiar con un inconveniente. El GPS puede generar varias posiciones en torno a un punto, y en el mejor de los casos, para nuestro sensor es de 1 cm, Tabla 4.2. Para solucionarlo, se calculará la orientación actual del robot una vez se haya desplazado 10 cm.

6.1.1. Evaluación experimental

Con el objetivo de probar el correcto funcionamiento del Nodo de control se ha realizado una simulación en Gazebo y Rviz en un mundo libre de obstáculos.

Para visualizar la ruta deseada y la trayectoria que sigue el robot dentro de Rviz se han creado dos *publishers*:

- `/global_path`: este será el tópico empleado para publicar el camino que se desea seguir. El tipo de mensaje que utiliza es **Path** y se compone principalmente por un array de **PoseStamped** que contiene las coordenadas de la ruta.
- `/husky/path`: este tópico se emplea para publicar la trayectoria que describe el robot y también utiliza mensajes de tipo **Path**.

Así pues, si realizamos una trayectoria sinusoidal al igual que hemos visto en Simulink, el resultado obtenido es el siguiente:

Figura 6.2: Trayectoria sinusoidal Rviz.

Se ha marcado con una línea roja la ruta deseada y con una línea verde la trayectoria que sigue el robot. Como se puede apreciar, prácticamente se solapan ambas rutas indicando el buen comportamiento del controlador. Además, podemos calcular el error que posee el robot durante el recorrido al igual que hemos hecho en Simulink:

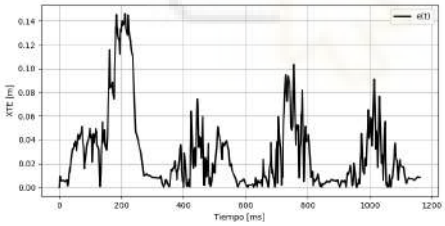


Figura 6.3: Evolución del error.

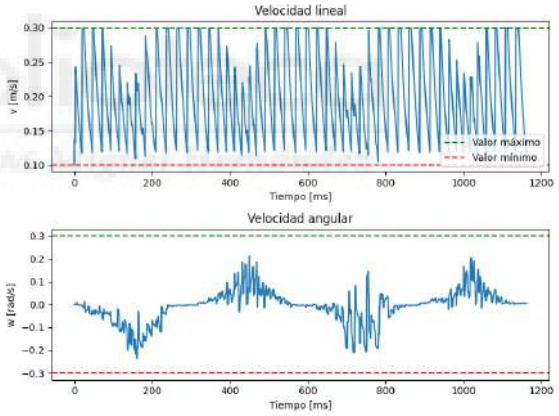


Figura 6.4: Salidas de control para Rviz.

La Figura 6.3 refleja que se ha seguido la ruta deseada con un error mínimo. El valor medio resultante es de 0.028 metros y su valor máximo de 0.146 metros. Por otro lado, en la Figura 6.4 vemos la evolución de las velocidades durante el recorrido dentro del rango establecido. Las líneas verde y roja representan los valores de velocidad máxima y mínima, respectivamente.

6.2. *Path Planning*

Para realizar el ejemplo anterior ha sido necesario ingresar manualmente las coordenadas que forman la ruta a seguir. Este método es muy ineficaz, en especial cuando se desea probar varias trayectorias o realizar modificaciones durante la ejecución.

Como alternativa, se propone utilizar Rviz como interfaz para definir de forma dinámica el camino que se desea seguir. El objetivo es mostrar el robot sobre una vista satélite de la zona para conocer de forma visual donde está situado y entonces seleccionar la posición destino directamente sobre este mapa.

6.2.1. Rviz Satellite

Para lograr mostrar una imagen satelital en Rviz se hará uso del *Plugin rviz_satellite* que se puede encontrar en [50]. Este paquete se encarga de obtener las imágenes satélite de una zona a partir de un objeto URI que le proporcionemos como fuente. En este caso se ha utilizado OpenStreetMap, ya que no tiene restricciones de uso y proporciona datos actualizados. Una vez se obtienen las imágenes se cargan en Rviz mediante una instancia de `AerialMapDisplay`.

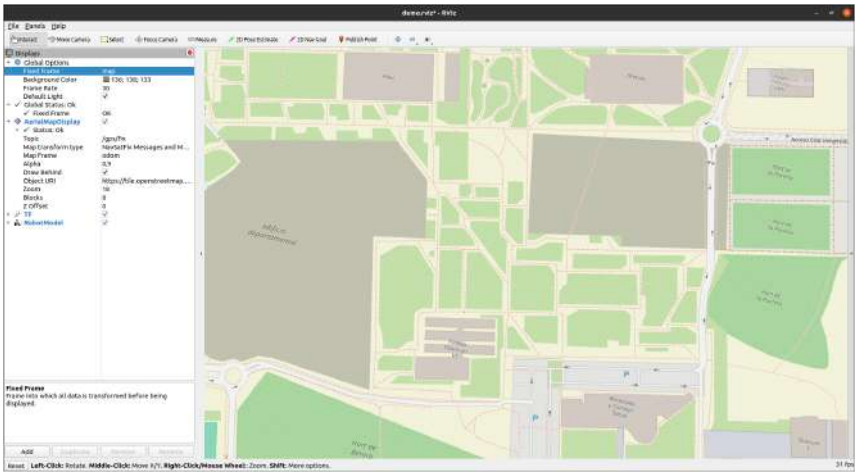


Figura 6.5: Vista del campus UMH en Rviz

En la Figura 6.5 se puede apreciar los resultados tras realizar la configuración oportuna para este paquete:

- `Latitude`: 38.275401 y `Longitude`: -0.686178 (Campus UMH)
- `Topic`: Navsat/fix.
- `Map Transform Type`: Map frame mode
- `Zoom` 18.

6.2.2. Dijkstra sobre OverpassTurbo

En segundo lugar, se hace uso de la librería [51] para obtener el camino óptimo entre dos puntos.

Para su funcionamiento necesita una base de datos de las coordenadas geográficas del terreno, la cual obtendremos de la propia página de *overpass-turbo*. Esta base de datos cuenta con los Nodos pertenecientes a caminos de tipo *highway*.

Con estos datos, la librería emplea un algoritmo Dijkstra para hallar la ruta más corta. La idea es formar un grafo uniendo el vértice origen con el resto de vértices a través del recorrido con menor coste, en nuestro caso, la distancia. Una vez está formado el grafo se sabe cuál es el mejor camino para alcanzar un vértice cualquiera.

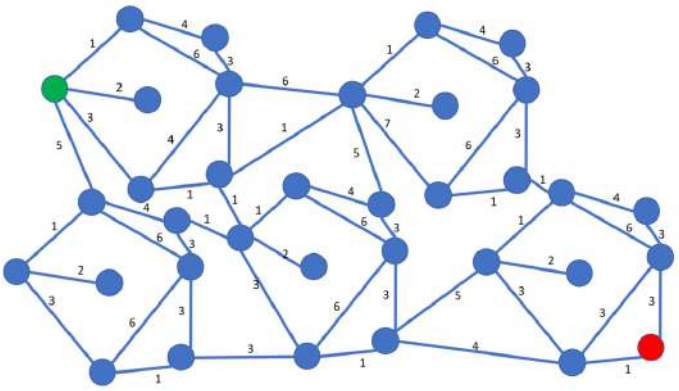


Figura 6.6: Ejemplo grafo Dijkstra [52].

La librería ha sido modificada para actuar como una función de Python. Esta función recibe las coordenadas de origen y destino y devuelve la ruta más corta al destino. Además, se ha utilizado el campus de la UMH como base de datos. Ver Anexo A.2

6.2.3. *Clicked Point*

En último lugar, se crea un Nodo suscriptor al tópico `/clicked_point`, ver Anexo A.3. Este tópico contiene las coordenadas x, y, z del punto seleccionado en Rviz, el cual será nuestra posición objetivo.

Estas coordenadas vienen expresadas en el sistema de referencia de Rviz, mientras que el algoritmo para la obtención de la ruta trabaja con valores de latitud/longitud. Por ese motivo, se ha adaptado la posición del *target point* siguiendo los pasos de la Figura 6.7. Además, al igual que con el Nodo de control, nos suscribimos al tópico `/navsat/fix` para obtener nuestra posición actual en todo momento.

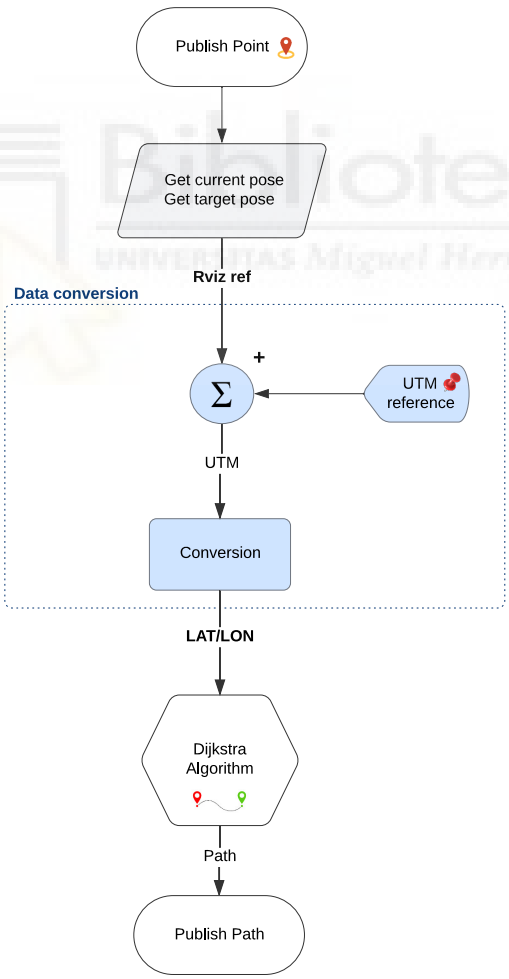


Figura 6.7: Diagrama de flujo del código.

Una vez obtenidas las coordenadas de la posición actual y la posición objetivo en el sistema de referencia correcto, se llama a la función de Dijkstra para obtener el camino más corto. Este camino se publica en `/husky/globalPath` para que el Nodo de control tenga acceso a las coordenadas que forman la ruta y se visualice el mensaje *Path* en Rviz.

6.3. Simulación

De esta forma ya podemos realizar pruebas más interesantes como enviar al robot desde un edificio a otro del campus de la UMH. En este caso, se ha seleccionado el edificio Innova como punto de partida y el edificio Altet como destino, lo que supone 250 metros de recorrido.

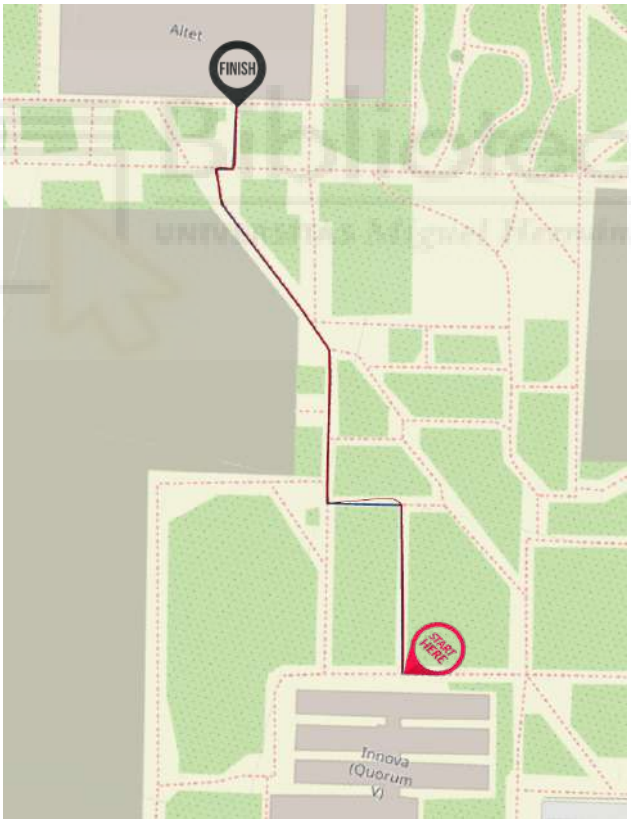


Figura 6.8: Trayectoria seguida en la navegación entre edificios.

En la Figura 6.8 se muestra con color azul la ruta deseada que ha generado el algoritmo Dijkstra y con color rojo el camino realizado por el robot. En ella se refleja que la simulación ha sido satisfactoria, llegando a destino sin problemas.

7. INTEGRACIÓN EN ROBOT REAL

Una vez superadas todas las simulaciones, se procede a las pruebas en el robot real. En este capítulo se describe el proceso empleado para llevar a cabo la navegación en el robot Husky A200 y la prueba realizada en el campus de la UMH.

7.1. Procedimiento

El procedimiento para la puesta en marcha del robot Husky A200 es el siguiente:

1. En primer lugar, se establece una conexión remota entre el ordenador equipado en el robot y el ordenador de control mediante SSH.

2. En segundo lugar, se ejecutan los nodos encargados del funcionamiento de la plataforma robótica.

   $husky_ros_setup

   $husky_launch_base

3. A continuación, se coloca la estación base GPS del robot en una zona de referencia en el exterior. Se ha elegido las coordenadas 38.274883, -0.685636 como punto de referencia debido a que es un lugar despejado y cuenta con una marca en el suelo para asegurar la reproducibilidad.

4. Se realiza el lanzamiento y comprobación de los sensores que posee el robot.

   $husky_launch_sensors

   $husky_check_sensors

5. Se lanza el nodo de navegación y se comprueba que el robot es capaz de moverse de forma autónoma.

   $python3 run_control.py

7.2. Funcionalidades adicionales

Con el objetivo de tener un sistema de navegación más completo, se han añadido algunas funcionalidades extras al sistema principal.

- **Recover route:** son muchos los motivos que pueden causar la necesidad de parar la navegación, es por ello que se ha agregado la opción de *recover_route*. Esta permite al robot retomar la ruta desde el punto más cercano, evitando tener que reiniciar la navegación desde el principio.
- **Distancia de muestreo:** cuando se graba el *rosbag* con la ruta que próximamente seguirá el robot se hace a la frecuencia de actualización del sensor. Esto generará *waypoints* a todo tipo de distancias. Como remedio se ha añadido la posibilidad de muestrear la ruta de forma que haya una distancia mínima entre *waypoints*.
- **Follow reverse:** en ciertos estudios es de interés que el robot repita la ruta en sentido contrario. Para cubrir esta necesidad se ha aportado la opción *follow_reverse*.

Estos añadidos al igual que otras variables de configuración, como las ganancias del controlador, se han unificado en un archivo de configuración. De esta forma se facilita la modificación a futuro de los parámetros del sistema de navegación.

7.3. Experimento y resultados

Una necesidad común en robótica e investigación es tener la capacidad de replicar un experimento, ya sea para validar resultados o estudiar los efectos del cambio de condiciones. Para abordar esta problemática, se ha registrado manualmente una ruta circular en un entorno exterior, con el objetivo de que el robot la siga de forma autónoma. La ruta no tiene obstáculos y tiene una longitud de 173 metros.

Figura 7.1: Trayectoria exterior representada sobre imagen por satélite.

En la Figura 7.1 se muestra la ruta grabada. Cada punto azul representa una coordenada GPS-RTK guardada en el *rosbag*. Como se puede observar, la distancia entre cada *waypoint* es irregular, ya que las coordenadas se guardan a una frecuencia determinada sin tener en cuenta la posición, lo que resulta en áreas con una mayor acumulación de puntos que en otras. Por esta razón, se realiza un muestreo de los puntos para que la distancia entre ellos sea constante.

Tras realizar los pasos descritos en el apartado anterior, se ha lanzado el nodo de navegación con una distancia entre *waypoints* de 1.2 metros y una tolerancia de 0.5 metros. Además, se han utilizado las ganancias mostradas en la Tabla 5.3.

Figura 7.2: Plataforma Husky con los sensores instalados.

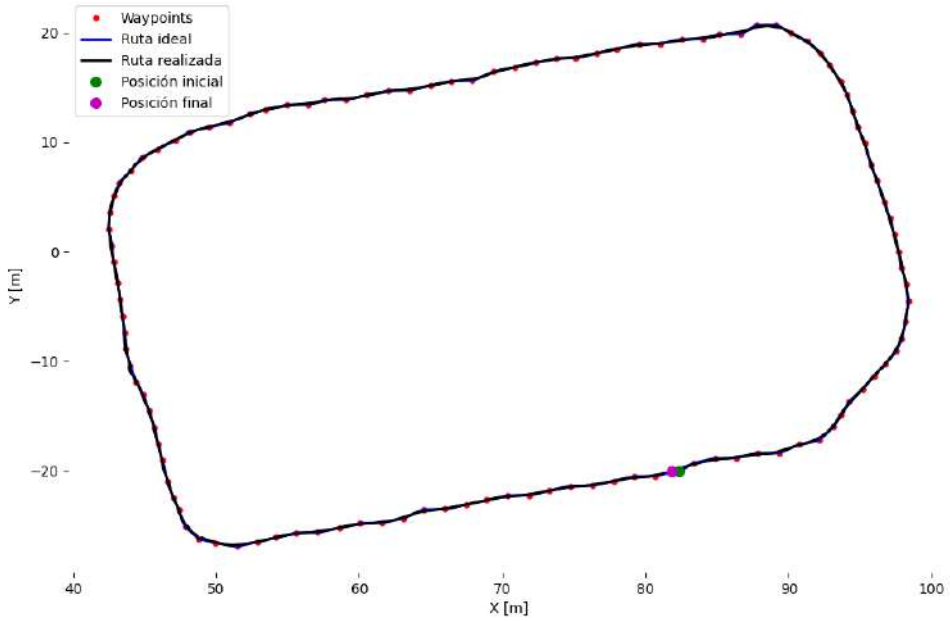


Figura 7.3: Ruta recorrida durante el experimento.

En la Figura 7.3 se muestran los resultados obtenidos. En color rojo se representan las posiciones GPS-RTK tras ser muestreadas, es decir, los *waypoints* que el robot debe alcanzar. La línea azul representa la unión de estos *waypoints* y muestra la ruta ideal, mientras que en negro se muestra el recorrido que ha realizado el robot. Como se puede apreciar, tanto la ruta real como la deseada están prácticamente superpuestas, lo que indica que el experimento se ha llevado a cabo con éxito. En la Figura 7.4 se puede corroborar esto, ya que se muestra cómo el error ha sido mínimo.

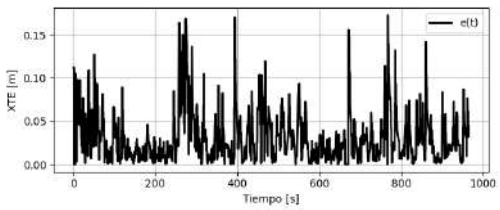


Figura 7.4: Evolución del error.

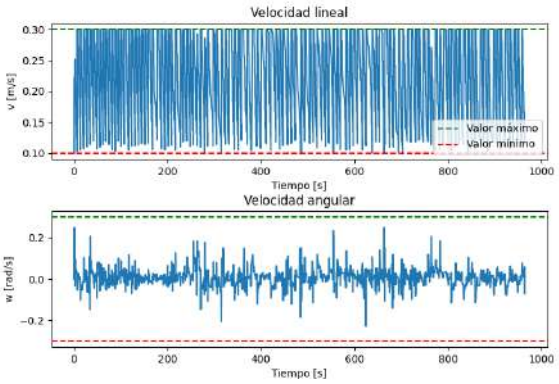


Figura 7.5: Salidas de control para Rviz.

Además de la Figura 7.4, se puede observar que el error máximo de XTE ha sido de 0.173 metros, mientras que el valor medio de error durante la ejecución ha sido de 0.054 metros. Estos valores son muy bajos y en términos prácticos despreciables, lo que indica que el robot ha seguido la ruta de manera precisa. Por otro lado, en la Figura 7.5 se muestran las velocidades del robot en función del tiempo, donde se puede apreciar que el control se encuentra dentro de la zona deseada.

8. EVASIÓN DE OBSTÁCULOS

En esta sección se abordará el problema de evitar obstáculos en el contexto de la navegación autónoma de un robot. Hasta ahora, se ha desarrollado un sistema de control que permite alcanzar una posición destino a partir de un punto de partida siguiendo la trayectoria global generada por el planificador Dijkstra utilizando una hoja de ruta a priori, o un *rosbag*.

Sin embargo, es necesario tener en cuenta los obstáculos presentes en el entorno para evitar colisiones, con el objetivo de obtener una navegación completamente autónoma.

De esta forma, surgen dos nuevas tareas que se deben abordar. En primer lugar, identificar que zonas son seguras para transitar con el robot. En segundo lugar, generar un camino seguro a través de estas zonas.

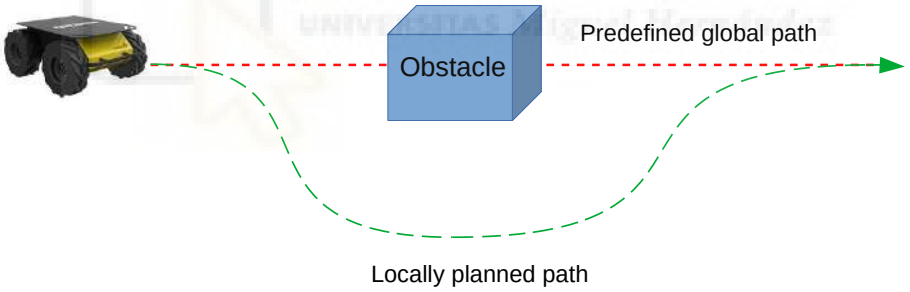


Figura 8.1: Ilustración de la evasión de obstáculos.

Este problema se ilustra en la Figura 8.1 donde un obstáculo se encuentra en el camino global planificado. Entonces, es necesario generar un nuevo camino alternativo que permita al robot evadirlo de manera segura y continuar hacia su destino. Este proceso de generación de un nuevo camino debe realizarse de forma dinámica y en tiempo real, teniendo en cuenta la posición y orientación actual del robot, así como la información proporcionada por los sensores para detectar y localizar los obstáculos en el entorno.

8.1. Evaluación de la transitabilidad mediante una red neuronal

La capacidad de identificar las áreas del terreno por las que el robot puede desplazarse de manera segura es de gran interés en el campo de la robótica móvil, ya que le permitiría operar exitosamente en una gran variedad de aplicaciones.

Para la resolución de este problema es esencial el uso de sensores que permitan obtener información detallada del entorno. En este caso, se ha utilizado un sensor LiDAR, el cual cuenta con una alta precisión y es invariante a las condiciones de iluminación. Este sensor proporciona los datos en forma de nube de puntos 3D donde cada punto está definido con coordenadas cartesianas, $\overrightarrow{p_i} = (x_i, y_i, z_i)$. Además, se define para cada punto una etiqueta, l_i, para indicar si es transitable (1) o no (0) y un vector característico, $\overrightarrow{f_i} \in \mathbb{R}$, con información encapsulada de cada uno.

Para hallar la transitabilidad de los puntos mediante estos datos, se ha empleado una técnica emergente que consiste en el uso de redes neuronales convolucionales. Tras un entrenamiento sobre un conjunto de datos etiquetados $\{(\overrightarrow{p_i}, \overrightarrow{f_i}), l_i\}$, el modelo se encarga de capturar patrones muy precisos en los datos de la nube de puntos, lo que le permite predecir con exactitud el estado de travesabilidad de cada punto.

El modelo utilizado es Te-next [53] desarrollado por Antonio Santo. Sin entrar en detalle, esta red neuronal sigue una arquitectura de codificador-decodificador (encoder-decoder), que permite analizar y segmentar nubes de puntos 3D para determinar las áreas transitables. Sus resultados han demostrado que el método es robusto y efectivo, superando en precisión a otros enfoques documentados.

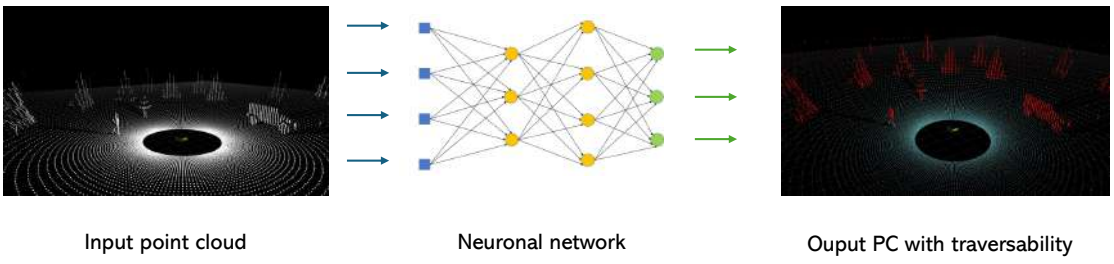


Figura 8.2: Ilustración del análisis convolucional.

En la Figura 8.2 se ilustra el funcionamiento de la red neuronal. Mediante el análisis de la nube de puntos de entrada, la red es capaz de segmentar la nube en áreas transitables y no transitables.

El modelo de red fue implementado utilizando librerías de aprendizaje profundo como Minkowski Engine y Pytorch. Además, se creó un Nodo ROS para proporcionar la evaluación de la transitabilidad como un servicio, ver Anexo B.1. Siguiendo el esquema mostrado en la Figura 4.10, el Nodo es llamado a través de un mensaje de solicitud que contiene la nube de puntos a evaluar. Luego, el servicio procesa la nube de puntos y devuelve la clasificación como respuesta. A continuación, se muestra la estructura del mensaje de solicitud y respuesta creado para la comunicación:

```
1        sensor_msgs/PointCloud2 input
2        ---
3        sensor_msgs/PointCloud2 trav_points
4        sensor_msgs/PointCloud2 no_trav_points
5
```

Con este método, se evalúan únicamente las nubes necesarias para la planificación local, lo que reduce el tiempo de cálculo y mejora la eficiencia del sistema.

Por otro lado, tras analizar la nube de puntos, se le asigna a cada punto un código de color RGB, en función de si es transitable o no. De esta forma, se puede visualizar la nube de puntos en Rviz y comprobar visualmente el análisis realizado por la red como se muestra en la Figura 8.3.

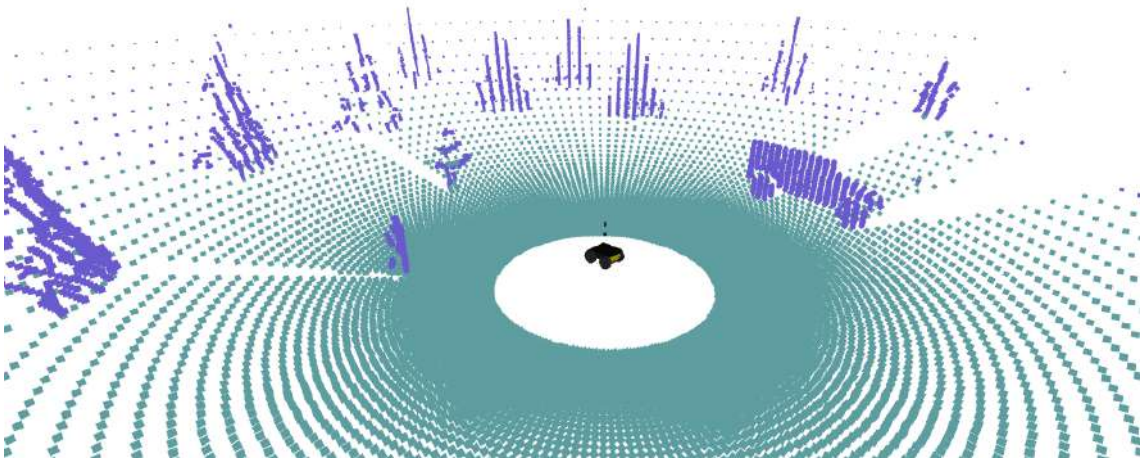


Figura 8.3: Visualización en Rviz de la transitabilidad mediante red neuronal.

8.2. *Path Planning* local basado en algoritmo RRT

Se ha utilizado el algoritmo RRT (Rapidly-exploring Random Tree) [28] para generar un camino seguro a través de los puntos transitables identificados por la red neuronal.

El algoritmo RRT se basa en la generación de un árbol de búsqueda aleatorio que se expande desde el punto de inicio hacia el punto destino. En cada iteración, se selecciona un punto aleatorio en el espacio de búsqueda y el árbol se expande hacia ese punto. El algoritmo termina cuando el árbol alcanza el punto destino o se supera un número máximo de iteraciones.

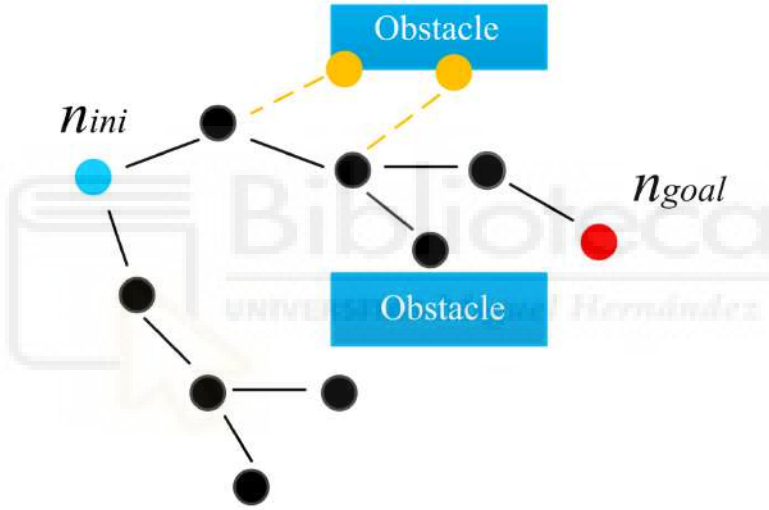


Figura 8.4: Ejemplo de grafo RRT [54].

La Figura 8.4 muestra la expansión del algoritmo RRT desde el nodo inicial (n_{ini}, azul) hasta el nodo objetivo (n_{goal}, rojo) en un espacio con obstáculos (rectángulos azules). Los nodos negros representan puntos explorados y las líneas negras son conexiones válidas. Las líneas punteadas amarillas indican intentos fallidos de conexión debido a obstáculos. Como se puede apreciar el RRT se expande iterativamente, sorteando obstáculos, para encontrar un camino desde n_{ini} hasta n_{goal}.

El objetivo planteado es utilizar esta técnica como planificador local. Después de haber generado el camino global con Dijkstra o manualmente, se ejecutará una planificación local entre *waypoints* para obtener un camino transitable.

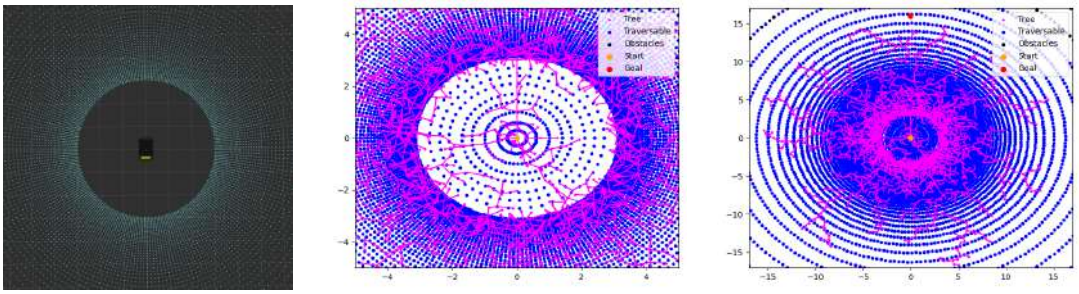
Como punto de partida, se ha utilizado la librería [55], específicamente la versión del RRT que trabaja con nubes de puntos tridimensionales. Algunas de las características con las que cuenta son:

- **Épsilon:** es la longitud de paso máxima. Es interesante un valor relativamente pequeño para una expansión del grafo precisa, lo cual nos lleva a dos problemas. En primer lugar, debido a la posición del LiDAR, los puntos a la altura del suelo no son detectados hasta un rango de 3 m, ver Figura 8.5(a). Este hecho exige tener una épsilon de al menos esa distancia para realizar la búsqueda. Como solución se crean manualmente círculos transitables en ese rango para que el algoritmo alcance la nube de puntos, ver Figura 8.5(b). Por otro lado, la separación entre los puntos detectados por el LiDAR aumenta según la distancia, generando un límite para el árbol de expansión, ver Figura 8.5(c). Entonces, para asegurar su alcance, el valor óptimo de épsilon dependerá de la separación entre *waypoints*. En el caso de tener grabado el trayecto, podemos fijar la distancia entre *waypoints* con las funciones adicionales creadas y de esta forma asegurar una ejecución óptima.

- **Max_nodes:** indica el número máximo de iteraciones que se pueden realizar para encontrar el destino. Un valor mayor resultará en una búsqueda más intensa, pero también aumentará el tiempo de cómputo. Es por ello que se ha optado por un valor de 1000 iteraciones.

- **Robot_radius:** en la búsqueda RRT se considerará las dimensiones del robot. A la hora de planificar el camino se asegurará de que haya una superficie por la que navegar sin obstáculos en ese radio. Aunque el radio del robot sea aproximadamente de 0.5 m, se ha optado por un valor de 1 m para tener un espacio lateral de seguridad de 0.5 m.

- **TrajectorySmoother:** la ruta generada por un algoritmo RRT suele contener irregularidades y giros bruscos. Para mejorar la eficiencia de la navegación y obtener giros más suaves se suaviza el camino obtenido.



(a) Rango LiDAR. (b) Círculos añadidos. (c) Rango de búsqueda.

Figura 8.5: Problemas encontrados en la elección de épsilon.

Además, el algoritmo tiene la peculiaridad de que en cada iteración, después de seleccionar un punto aleatorio y conectarlo al árbol existente, se intenta establecer una conexión directa entre este nuevo punto y el objetivo. Esto resulta en rutas más directas hacia el objetivo y un tiempo de ejecución más rápido en comparación con el RRT básico, lo cual es esencial para la navegación autónoma.

Entonces a partir de esta librería, se ha realizado una adaptación del código para satisfacer las necesidades del proyecto. Específicamente, se ha creado una función de Python que lleve a cabo estas tareas, ver Anexo B.2.

Esta función toma como entrada la posición inicial, la posición final y las nubes de puntos transitables y no transitables, todas ellas en coordenadas UTM con una referencia local. Los parámetros de salida relevantes son los siguientes:

- **Goal_colision:** indica si el destino se encuentra dentro de un obstáculo, lo que significa que es inalcanzable.
- **Goal_reached:** indica si el algoritmo ha generado un camino que conecta el punto de inicio con el punto final. En caso contrario, se devuelve el camino hasta el nodo más cercano.
- **Path:** contiene los puntos que forman el camino generado, el cual ha sido suavizado para facilitar la conducción del robot.

Estos parámetros nos permitirán abordar todos los posibles casos que pueden ocurrir durante la planificación local.

8.3. Unificación en un sistema de control completo

Para implementar un sistema de control completo que permita al robot navegar de manera autónoma evitando obstáculos, se ha unificado la evaluación de la transitabilidad mediante la red neuronal y la planificación local con el algoritmo RRT, junto con el control previamente realizado. La integración de estos componentes se muestra en el diagrama de flujo de la Figura 8.6 y se encuentra en el Anexeo B.3.

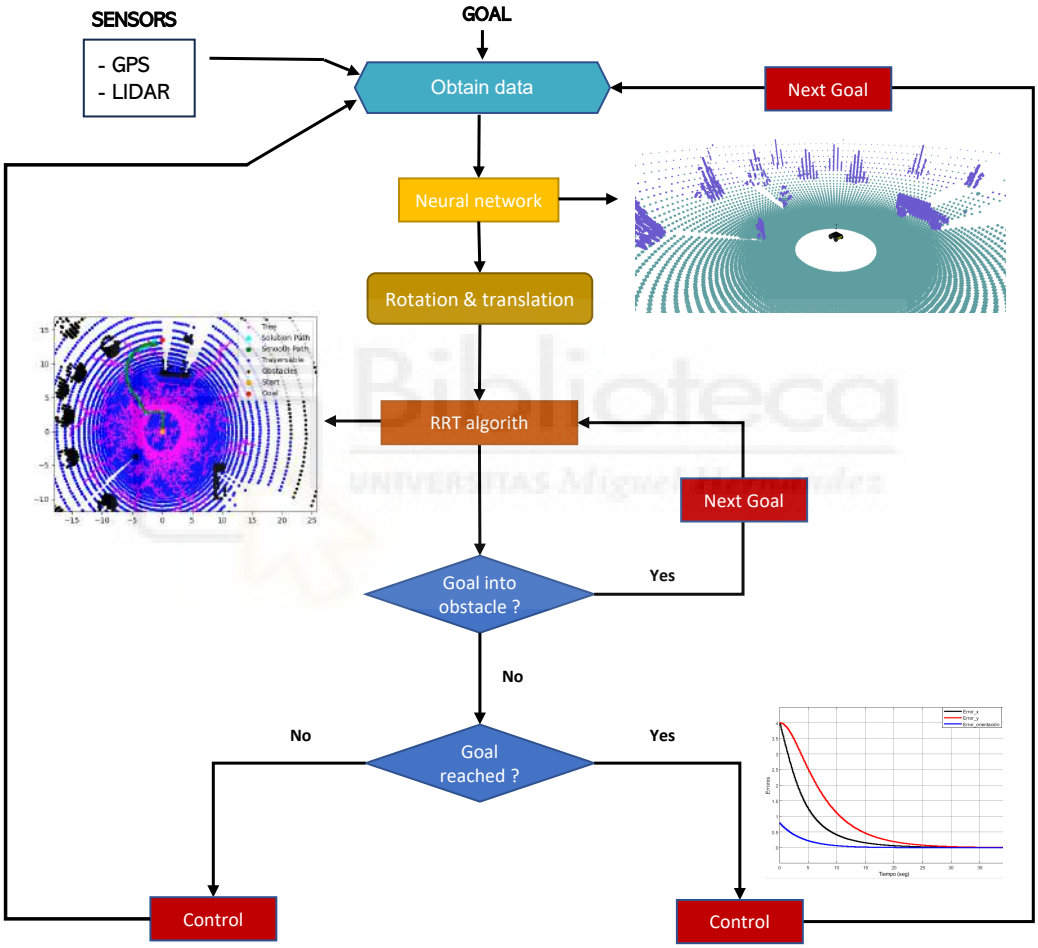


Figura 8.6: Diagrama de flujo del sistema de control.

Supongamos que nos encontramos en una posición i y queremos alcanzar la posición $i+1$. En primer lugar, obtenemos los datos de nuestra posición actual mediante la suscripción a los sensores correspondientes. A continuación, creamos un cliente ROS que solicita al servicio la evaluación de la nube de puntos mediante la red neuronal.

Los puntos clasificados están referenciados a la posición del LiDAR. Por lo tanto, antes de enviarlos al algoritmo RRT, es necesario transformarlos al sistema de referencia global. Esto se logra mediante una transformación que aplica tanto como un desplazamiento a los puntos LiDAR. Utilizando nuestra posición actual *(x, y, z)* y la orientación calculada θ, se calcula la matriz de transformación de la siguiente manera:

$$T = \begin{pmatrix} cos(\theta) & -sin(\theta) & 0 & x \\ sin(\theta) & cos(\theta) & 0 & y \\ 0 & 0 & 1 & -0{,}7 \\ 0 & 0 & 0 & 1 \end{pmatrix} \tag{8.1}$$

Una vez obtenidos los puntos transitables en el sistema de referencia global se ejecuta el algoritmo RRT para generar un camino seguro entre los *waypoints*. Por último, se envía la información al controlador para que el robot siga el camino generado.

Como se puede observar, el grafo contempla los 3 casos posibles que se pueden dar en la planificación local. La solución planteada para cada uno de ellos es la siguiente:

1. **Objetivo alcanzado:** el camino se ha generado de forma exitosa y se envía al controlador para que el robot lo siga.

2. **Objetivo no alcanzado:** en este caso, no se ha generado un camino que conecte el punto de inicio con el punto final. Esto se puede deber entre otras cosas, a que un obstáculo restringe la visión del LiDAR. Entonces, se envía al controlador el camino hasta el nodo más cercano. Una vez alcanzado, se repite el proceso con la esperanza de que desde esa posición se pueda alcanzar el objetivo.

3. **Objetivo inalcanzable:** contempla la posibilidad de que el objetivo se encuentre rodeado de un obstáculo que impida ser alcanzado. En ese caso, se vuelve a ejecutar el algoritmo RRT pero con el siguiente objetivo de la lista.

8.4. Modelado mundo Gazebo

Para evaluar el sistema de evasión de obstáculos se ha creado un entorno simulado en Gazebo, ver Figura 8.7. En él se han representado gran variedad de elementos que puede encontrarse el robot en la realidad, como paredes, árboles, coches, etc.

Figura 8.7: Entorno simulado en Gazebo.

La prueba a superar consiste en completar un camino circular compuesto por doce *waypoints* tal y como se muestra en la Figura 8.8. El robot debe ser capaz de navegar por el entorno evitando los obstáculos presentes en él.

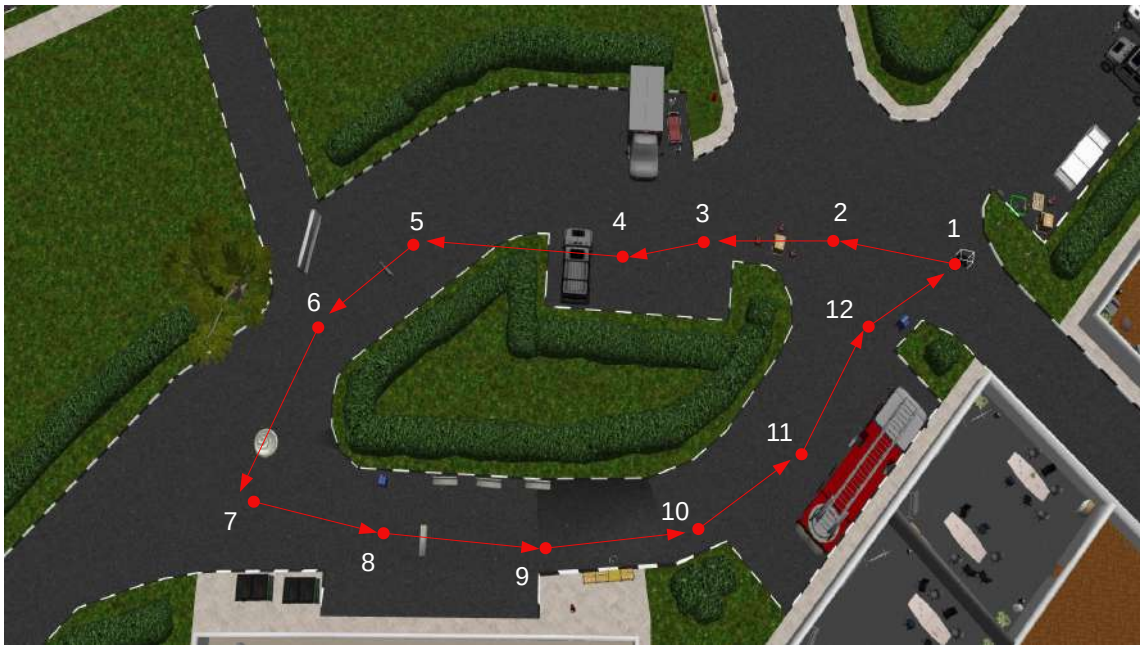
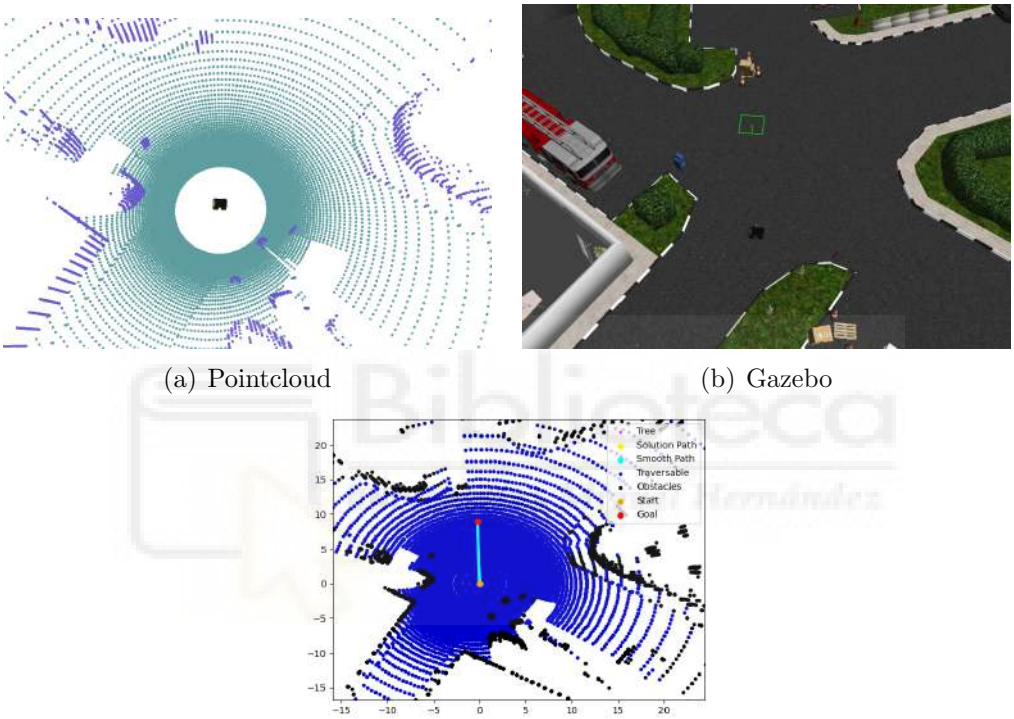


Figura 8.8: Camino a seguir por el robot.

8.5. Resultados

La prueba ha sido completada exitosamente. Durante la prueba, se utilizó un valor de épsilon de 0.4 m, lo que se traduce en un radio de alcance de 15 metros. Además, se estableció el número máximo de iteraciones en 1000 y un margen de 0.5 m para considerar que un *waypoint* ha sido alcanzado. A continuación, se presentan los casos más relevantes observados durante la prueba.



(a) Pointcloud (b) Gazebo

(c) RRT

Figura 8.9: Resultados obtenidos en el punto de partida.

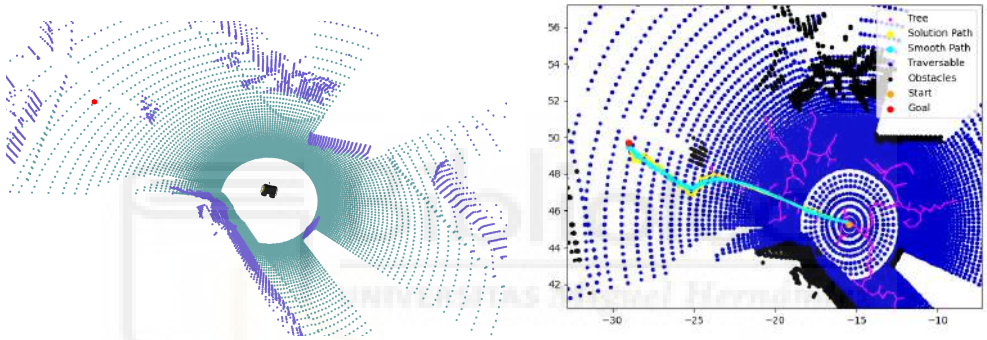
En primer lugar, se exhibe el punto de partida del robot. En la Figura 8.9(a), se observa la nube de puntos generada y la posición actual del robot. Estos elementos nos permiten visualizar la perspectiva del robot durante la planificación local. Los puntos transitables se representan en un color verde grisáceo, mientras que los puntos no transitables se muestran en un color azul violeta. Además, se ha resaltado el objetivo actual como un punto rojo dentro de la nube de puntos.

Al comparar la representación del entorno en el mundo de Gazebo, el cual se muestra en la Figura 8.9(b), se puede apreciar que la nube de puntos coincide con la realidad.

Esto indica que la red neuronal ha evaluado correctamente el entorno.

Por otro lado, en la tercera Figura 8.9(c) se ilustra el camino generado por el algoritmo RRT a partir de la nube de puntos. Este algoritmo cumple su objetivo al obtener de manera rápida el camino hacia el destino. Aunque el algoritmo RRT se expande de forma aleatoria, en casos donde el objetivo se encuentra directamente al alcance, se obtendrá un camino directo hacia el destino. Esto se debe a que en cada iteración se intenta establecer una conexión entre el punto aleatorio y el objetivo.

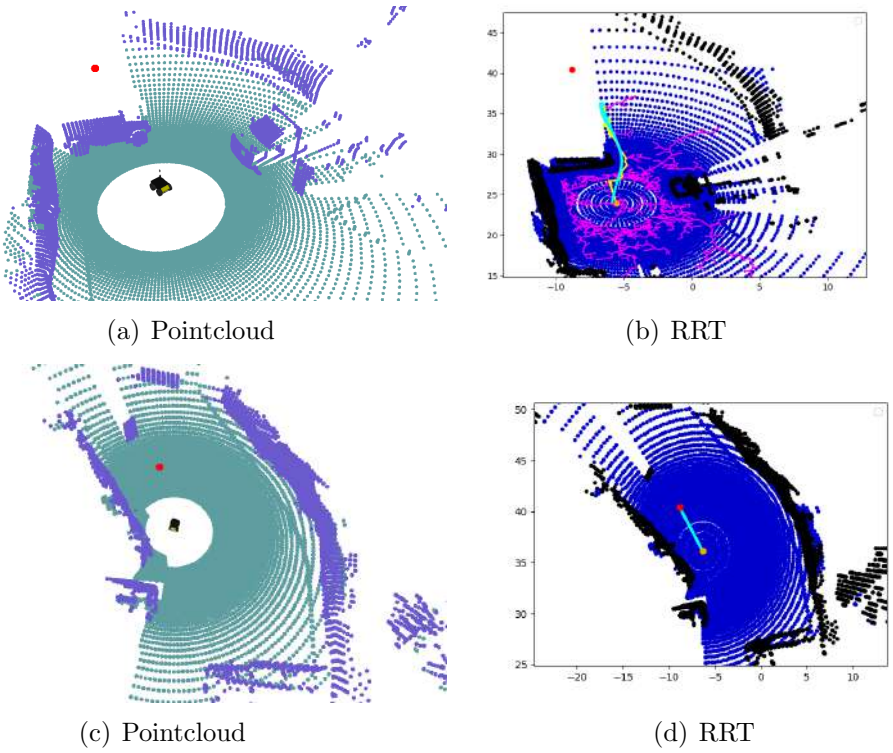
En segundo lugar, se encuentra el caso en el que el LiDAR tenga visión sobre el objetivo, pero la ruta no sea directa debido a la presencia de obstáculos. Esto lo encontramos en el *waypoint* 6:



(a) Pointcloud (b) RRT

Figura 8.10: Resultados obtenidos en el 6º *waypoint*.

En la Figura 8.10(a), se puede observar que el objetivo se encuentra en una zona transitable. Además, en la Figura 8.10(b), se aprecia cómo el algoritmo es capaz de evadir el objeto presente en el camino y alcanzar el objetivo. En último lugar, se encuentra el caso donde el sensor LiDAR no tiene una visión completa de la situación del objetivo, como ocurre en el cuarto *waypoint*.



(a) Pointcloud (b) RRT

(c) Pointcloud (d) RRT

Figura 8.11: Resultados obtenidos en el 4º *waypoint*.

En este caso, el algoritmo RRT se expande hasta el número máximo de iteraciones y devuelve un camino hasta el nodo más cercano, ver Figura 8.11(b). En la Figura 8.11(c) vemos cómo al llegar a esa posición, la visibilidad del LiDAR aumenta permitiendo ahora si alcanzar el destino. Entonces se genera un camino seguro para llegar a él, Figura 8.11(d).

Durante la ejecución de la evaluación y planificación el robot se encuentra parado. El tiempo de ejecución dependerá de la potencia del sistema hardware empleado, de la complejidad de la nube de puntos a evaluar y por último, de la facilidad para encontrar un camino que una los puntos de comienzo y destino. Por ejemplo, este experimento se ha realizado con un ordenador portátil Intel Core i7-1165G7 y se ha obtenido un tiempo medio de 1.3 segundos.

9. CONCLUSIONES Y LÍNEAS FUTURAS

Para concluir este proyecto, en este apartado final se presentarán las conclusiones obtenidas tras su realización. Posteriormente, se discutirán las líneas futuras que podrían desarrollarse a partir de este trabajo.

9.1. Conclusiones

La conclusión más importante es que se ha desarrollado un sistema de navegación autónoma para un robot móvil sobre ruedas en entornos exteriores capaz de operar robustamente tanto en entornos simulados como en entornos reales.

En primer lugar, se han diseñado y aplicado dos sistemas de control de seguimiento de trayectoria. El objetivo ha sido estudiar cuál de ellos logra un mejor rendimiento y se adapta mejor a las necesidades del proyecto. Estos controladores tienen características diferentes: el control PD es lineal y SISO, mientras que el control basado en la teoría de Lyapunov es no lineal y MIMO.

Para llevar a cabo el estudio, ambos controladores se han integrado en el entorno gráfico de Simulink. En este entorno, se obtuvieron los valores óptimos de ganancia sin verse afectados por factores externos. Posteriormente, se realizó una comparación entre ambos sistemas en trayectorias con diferentes características. Después de evaluar su rendimiento utilizando el *Cross Track Error* (XTE), se puede afirmar que el control de Lyapunov logra mejores resultados en términos de seguimiento de trayectoria y robustez. Destaca especialmente en trayectorias con curvas pronunciadas y cambios bruscos de dirección, donde obtuvo un error medio de 0.123 m, mientras que el PD se queda atrás con 0.349 m. La dependencia de la velocidad lineal en el ángulo de dirección ha sido un factor diferenciador para lograr adaptabilidad sobre el control PD.

Este estudio proporciona una base sólida para el resto de la investigación, ya que se ha demostrado que el control de Lyapunov es el más adecuado para ser implementado en el robot.

A continuación, se ha llevado a cabo la migración del sistema de control al entorno de ROS con el propósito de implementar el sistema de control en un robot físico. Este cambio de paradigma implica la programación de un sistema basado en nodos y tópicos que facilitan la comunicación entre los distintos componentes del robot. Esta transición hacia un entorno más avanzado y complejo es fundamental para lograr la integración exitosa del sistema de control en un entorno real.

En este sentido, se ha desarrollado un nodo de control que utiliza la información de los sensores del robot para calcular las velocidades lineales y angulares necesarias para seguir la trayectoria deseada. Con el fin de mitigar la dependencia de la IMU, que tiende a acumular errores, se obtiene la orientación del robot a partir del GPS. Siguiendo el mismo principio que la orientación objetivo, se calcula la orientación actual utilizando las posiciones actual y anterior con una distancia mínima de 10 cm para evitar errores causados por el ruido del GPS.

Por otro lado, para completar el objetivo de obtener un sistema autónomo con la mínima interacción humana, se ha empleado Rviz como interfaz de manejo que permita al usuario indicar el punto de destino. Para formar un entorno visual completo, se ha añadido la visualización del robot, los *waypoints*, la ruta deseada y la ruta realizada. Además, se ha aplicado un mapa satelital de la zona como fondo para así tener constancia de la ubicación del robot en todo momento.

Para formar el camino a partir de la posición actual del robot y el punto de destino seleccionado, se ha aplicado una planificación de trayectorias global. En este caso, se ha empleado el algoritmo Dijkstra que busca el camino más corto.

Entonces se ha procedido a la validación del sistema en entornos reales. Empleando el campus de la UMH como escenario, se ha comprobado que el robot es capaz de seguir la trayectoria deseada y llegar al punto de destino sin problemas.

En este momento del proyecto se presentó la necesidad de añadir al robot la capacidad de evadir obstáculos que haya presentes en el camino. Para ello, se ha implementado un modelo de red neuronal existente, capaz de evaluar la transitabilidad del terreno a partir de un sensor LiDAR. A partir de esta información, se realiza una planificación local que asegure la navegación segura del robot entre *waypoints*.

En definitiva, el nivel de consecución de los objetivos planteados en este proyecto ha sido muy alto. Pese a haber podido realizarse implementaciones con mayor nivel de eficiencia y precisión, los resultados son robustos y satisfactorios.

9.2. Líneas futuras

Tal y como se ha mencionado anteriormente, en este trabajo se ha desarrollado en su plenitud un sistema de navegación autónoma para robots móviles en entornos exteriores. Sin embargo, existen una serie de mejoras y ampliaciones que se pueden llevar a cabo para aumentar la robustez y eficiencia del sistema. Algunas posibles líneas futuras de investigación podrían incluir:

- Este proyecto se ha basado en el uso del GPS como medio de localización del robot. Sin embargo, el GPS puede presentar limitaciones en entornos urbanos o interiores, donde la señal puede ser débil o estar bloqueada. Por tanto, sería interesante explorar técnicas como la fusión de sensores (LiDAR, IMU, cámaras, etc.) para mejorar la precisión y fiabilidad de la localización del robot en diferentes entornos.
- Investigar y desarrollar algoritmos de planificación de trayectorias más eficientes y adaptativos, que tengan en cuenta la dinámica del robot y las características del entorno.
- Pese a que el uso de Rviz como interfaz de manejo ha sido un gran avance, se podría seguir trabajando en la creación de una interfaz propia que cuente con toda la información necesaria para el usuario.

Estas son solo algunas ideas para futuras investigaciones en el campo de la navegación autónoma. El objetivo final es seguir mejorando y avanzando en el desarrollo de sistemas autónomos más robustos y eficientes para su aplicación en diferentes escenarios y entornos.

BIBLIOGRAFÍA

[1] H. P. Moravec, «The Stanford cart and the CMU rover,» *Proceedings of the IEEE*, vol. 71, n.º 7, págs. 872-884, 1983.

[2] A. Elfes, «A sonar-based mapping and navigation system,» en *Proceedings. 1986 IEEE International Conference on Robotics and Automation*, vol. 3, 1986, págs. 1151-1156. DOI: 10.1109/ROBOT.1986.1087534.

[3] S. Dogru y L. Marques, «Using Radar for Grid Based Indoor Mapping,» en *2019 IEEE International Conference on Autonomous Robot Systems and Competitions (ICARSC)*, 2019, págs. 1-6. DOI: 10.1109/ICARSC.2019.8733614.

[4] T. Talaviya, D. Shah, N. Patel, H. Yagnik y M. Shah, «Implementation of artificial intelligence in agriculture for optimisation of irrigation and application of pesticides and herbicides,» *Artificial Intelligence in Agriculture*, vol. 4, págs. 58-73, 2020.

[5] J. Shan y C. K. Toth, *Topographic laser ranging and scanning: principles and processing*. CRC press, 2018.

[6] Y. Peng, D. Qu, Y. Zhong, S. Xie, J. Luo y J. Gu, «The obstacle detection and obstacle avoidance algorithm based on 2-D lidar,» en *2015 IEEE International Conference on Information and Automation*, 2015, págs. 1648-1653. DOI: 10.1109/ICInfA.2015.7279550.

[7] G. A. Kumar, A. K. Patil, R. Patil, S. S. Park e Y. H. Chai, «A LiDAR and IMU integrated indoor navigation system for UAVs and its application in real-time pipeline classification,» *Sensors*, vol. 17, n.º 6, pág. 1268, 2017.

[8] U. Weiss y P. Biber, «Plant detection and mapping for agricultural robots using a 3D LIDAR sensor,» *Robotics and autonomous systems*, vol. 59, n.º 5, págs. 265-273, 2011.

[9] S. Yang, S. Yang y X. Yi, «An Efficient Spatial Representation for Path Planning of Ground Robots in 3D Environments,» *IEEE Access*, vol. 6, págs. 41 539-41 550, 2018. DOI: 10.1109/ACCESS.2018.2858809.

[10] G.-C. Vosniakos y A. Mamalis, «Automated guided vehicle system design for FMS applications,» *International Journal of Machine Tools and Manufacture*, vol. 30, n.º 1, págs. 85-97, 1990.

[11] J. Leonard y H. Durrant-Whyte, «Mobile robot localization by tracking geometric beacons,» *IEEE Transactions on Robotics and Automation*, vol. 7, n.º 3, págs. 376-382, jun. de 1991, ISSN: 1042296X. DOI: 10.1109/70.88147. dirección: http://ieeexplore.ieee.org/document/88147/ (visitado 26-06-2023).

[12] M. Betke y L. Gurvits, «Mobile robot localization using landmarks,» *IEEE Transactions on Robotics and Automation*, vol. 13, n.º 2, págs. 251-263, 1997. DOI: 10.1109/70.563647.

[13] A. S. Etienne, J. Berlie, J. Georgakopoulos y R. Maurer, «Role of dead reckoning in navigation.,» 1998.

[14] O. Wijk y H. I. Christensen, «Localization and navigation of a mobile robot using natural point landmarks extracted from sonar data,» *Robotics and Autonomous Systems*, vol. 31, n.º 1-2, págs. 31-42, 2000.

[15] R. Triebel y W. Burgard, «Improving simultaneous mapping and localization in 3d using global constraints,» en *aaai*, 2005, págs. 1330-1335.

[16] C.-H. Chen y K.-T. Song, «Complete coverage motion control of a cleaning robot using infrared sensors,» en *IEEE International Conference on Mechatronics, 2005. ICM'05.*, IEEE, 2005, págs. 543-548.

[17] F. Fraundorfer y D. Scaramuzza, «Visual odometry: Part ii: Matching, robustness, optimization, and applications,» *IEEE Robotics & Automation Magazine*, vol. 19, n.º 2, págs. 78-90, 2012.

[18] H. P. Moravec, *Obstacle avoidance and navigation in the real world by a seeing robot rover*. Stanford University, 1980.

[19] P. K. Enge, «The global positioning system: Signals, measurements, and performance,» *International Journal of Wireless Information Networks*, vol. 1, págs. 83-105, 1994.

[20] G. Xu e Y. Xu, *GPS*. Springer, 2007.

[21] T. Kos, I. Markezic y J. Pokrajcic, «Effects of multipath reception on GPS positioning performance,» en *Proceedings ELMAR-2010*, IEEE, 2010, págs. 399-402.

[22] J. Zumberge, M. Heflin, D. Jefferson, M. Watkins y F. H. Webb, «Precise point positioning for the efficient and robust analysis of GPS data from large networks,» *Journal of geophysical research: solid earth*, vol. 102, n.º B3, págs. 5005-5017, 1997.

[23] C. Rizos y S. Han, «Reference station network based RTK systems-concepts and progress,» *Wuhan University Journal of Natural Sciences*, vol. 8, págs. 566-574, 2003.

[24] H.-J. Woo, B.-J. Yoon, B.-G. Cho y J.-H. Kim, «Research into navigation Algorithm for unmanned ground vehicle using Real Time Kinemtatic (RTK)-GPS,» en *2009 ICCAS-SICE*, 2009, págs. 2425-2428.

[25] E. W. Dijkstra, «A note on two problems in connexion with graphs,» en *Edsger Wybe Dijkstra: His Life, Work, and Legacy*, 2022, págs. 287-290.

[26] W. Shu-Xi, «The improved dijkstra's shortest path algorithm and its application,» *Procedia Engineering*, vol. 29, págs. 1186-1190, 2012.

[27] P. E. Hart, N. J. Nilsson y B. Raphael, «A formal basis for the heuristic determination of minimum cost paths,» *IEEE transactions on Systems Science and Cybernetics*, vol. 4, n.º 2, págs. 100-107, 1968.

[28] S. LaValle, «Rapidly-exploring random trees: A new tool for path planning,» *Research Report 9811*, 1998.

[29] D. Delling, P. Sanders, D. Schultes y D. Wagner, «Engineering route planning algorithms,» en *Algorithmics of large and complex networks: design, analysis, and simulation*, Springer, 2009, págs. 117-139.

[30] M. J. Barton, «Controller development and implementation for path planning and following in an autonomous urban vehicle,» *Undergraduate thesis, University of Sydney*, 2001.

[31] R. C. Coulter et al., *Implementation of the pure pursuit path tracking algorithm*. Carnegie Mellon University, The Robotics Institute, 1992.

[32] J. Wit, C. D. Crane III y D. Armstrong, «Autonomous ground vehicle path tracking,» *Journal of Robotic Systems*, vol. 21, n.º 8, págs. 439-449, 2004.

[33] Y. Kanayama, Y. Kimura, F. Miyazaki y T. Noguchi, «A stable tracking control method for an autonomous mobile robot,» en *Proceedings., IEEE International Conference on Robotics and Automation*, IEEE, 1990, págs. 384-389.

[34] J. Kong, M. Pfeiffer, G. Schildbach y F. Borrelli, «Kinematic and dynamic vehicle models for autonomous driving control design,» en *2015 IEEE Intelligent Vehicles Symposium (IV)*, 2015, págs. 1094-1099. DOI: 10.1109/IVS.2015.7225830.

[35] R. Marino, S. Scalzi y M. Netto, «Nested PID steering control for lane keeping in autonomous vehicles,» *Control Engineering Practice*, vol. 19, n.º 12, págs. 1459-1467, 2011, ISSN: 0967-0661. DOI: https://doi.org/10.1016/j.conengprac.2011.08.005. dirección: https://www.sciencedirect.com/science/article/pii/S0967066111001808.

[36] X. Wang, M. Fu, H. Ma e Y. Yang, «Lateral control of autonomous vehicles based on fuzzy logic,» *Control Engineering Practice*, vol. 34, págs. 1-17, 2015, ISSN: 0967-0661. DOI: https://doi.org/10.1016/j.conengprac.2014.09.015. dirección: https://www.sciencedirect.com/science/article/pii/S0967066114002342.

[37] D. Chwa, «Sliding-mode tracking control of nonholonomic wheeled mobile robots in polar coordinates,» *IEEE Transactions on Control Systems Technology*, vol. 12, n.º 4, págs. 637-644, 2004. DOI: 10.1109/TCST.2004.824953.

[38] C. E. Garcia, D. M. Prett y M. Morari, «Model predictive control: Theory and practice—A survey,» *Automatica*, vol. 25, n.º 3, págs. 335-348, 1989.

[39] A. Astolfi y R. Ortega, «Immersion and invariance: A new tool for stabilization and adaptive control of nonlinear systems,» *IEEE Transactions on Automatic control*, vol. 48, n.º 4, págs. 590-606, 2003.

[40] J. Borenstein e Y. Koren, «Obstacle avoidance with ultrasonic sensors,» *IEEE Journal on Robotics and Automation*, vol. 4, n.º 2, págs. 213-218, 1988. DOI: 10.1109/56.2085.

[41] A. Santo López, A. Gil, D. Valiente, M. Ballesta y A. Peidro, «Computing the Traversability of the Environment by Means of Sparse Convolutional 3D Neural Networks,» en *Proceedings of the 20th International Conference on Informatics in Control, Automation and Robotics Volume 1*, INSTICC-Institute for Systems y Technologies of Information, Control and ..., 2023.

[42] J. G. Ziegler y N. B. Nichols, «Optimum settings for automatic controllers,» *Transactions of the American society of mechanical engineers*, vol. 64, n.º 8, págs. 759-765, 1942.

[43] R. P. Borase, D. Maghade, S. Sondkar y S. Pawar, «A review of PID control, tuning methods and applications,» *International Journal of Dynamics and Control*, vol. 9, págs. 818-827, 2021.

[44] Z. Instruments, *Principles of PID Controllers*, jul. de 2023. dirección: https://www.zhinst.com/sites/default/files/documents/2023-08/zi_whitepaper_principles_of_pid_controllers.pdf.

[45] A. Behal, W. Dixon, D. M. Dawson y B. Xian, *Lyapunov-based control of robotic systems*. CRC Press, 2009.

[46] *Sistema de coordenadas universal transversal de Mercator - Wikipedia, la enciclopedia libre — es.wikipedia.org*, https://es.wikipedia.org/wiki/Sistema_de_coordenadas_universal_transversal_de_Mercator, [Accessed 27-03-2024].

[47] H. Weber, *Sick ag whitepaper*, feb. de 2024. dirección: https://cdn.sickcn.com/media/docs/2/12/212/whitepaper_lidar_es_im0080212.pdf.

[48] M. Quigley, K. Conley, B. Gerkey, J. Faust, T. Foote, J. Leibs, R. Wheeler, A. Y. Ng et al., «ROS: an open-source Robot Operating System,» en *ICRA workshop on open source software*, Kobe, Japan, vol. 3, 2009, pág. 5.

[49] M. Elsayed, A. Hammad, A. Hafez y H. Mansour, «Real time trajectory tracking controller based on lyapunov function for mobile robot,» *Int. J. Comput. Appl*, vol. 168, n.º 11, págs. 1-6, 2017.

[50] «Plugin rviz_satellite.» (), dirección: `https://github.com/nobleo/rviz_satellite`.

[51] A. Priyam, *Find and plot your optimal path using plotly and NetworkX in python*, ago. de 2020. dirección: `https://towardsdatascience.com/find-and-plot-your-optimal-path-using-plotly-and-networkx-in-python-17e75387b873`.

[52] V. Giraud, *Shortest path: Dijkstra algorithm*, 2020. dirección: `https://datascience.lc/shortest-path-dijkstra-algorithm/`.

[53] A. Santo, *Red neuronal Te-next*, 2024. dirección: `https://github.com/ARVCUMH?tab=repositories`.

[54] H. Zhang, Y. Wang, J. Zheng y J. Yu, «Path planning of industrial robot based on improved RRT algorithm in complex environments,» *IEEE Access*, vol. 6, págs. 53 296-53 306, 2018.

[55] A. Gil, *RRTPlanner*, 2024. dirección: `https://github.com/4rtur1t0/RRTPlanner`.

ANEXOS

A. Anexo I

En este primer anexo se incluyen los códigos de ROS desarrollados para el seguimiento de trayectorias libres de obstáculos.

En primer lugar, se muestra la unidad de control la cual se encarga de generar las salidas de control necesarias para seguir una trayectoria predefinida y libre de obstáculos. Obtiene los datos de posición del robot a través del tópico `/navsat/fix` y publica el resultado del control en el tópico `/husky_velocity_controller/cmd_vel`. Además, publica la trayectoria realizada el tópico `/husky/path`.

```
import rospy
import utm
from geometry_msgs.msg import Twist,PoseStamped, Vector3
from sensor_msgs.msg import NavSatFix
from nav_msgs.msg import Path
import math
import yaml
import numpy as np
import pandas as pd

try:
    with open(r'config/config.yaml') as file:
        config = yaml.load(file, Loader=yaml.FullLoader)
except:
    print("YAML loading error!...")

#Referencia
x_ref,y_ref, _, _ = utm.from_latlon(config['latitud_ref'],config['
        ↪ longitud_ref'])
i,alpha_prev,d_prev = 0 , 0.0 , 0.0
x_ant, y_ant, v_ant = -0.01 , -0.01 ,0
o_act = 0
x2, y2 = [], []

def sampler_gps(route=config['csv_rosbag'], dist_muestreo=config['
        ↪ dist_muestreo']):

    df = pd.read_csv(route, dtype = 'str')
    latitude = df['latitude'].to_numpy(dtype=np.float32)
    longitude = df['longitude'].to_numpy(dtype=np.float32)
    idx, data = [],[]
    dist_acum = 0

    for i, (lat, long) in enumerate(zip(latitude, longitude)):

```

```
34          if i == 0:
35              xo,yo,_,_ = utm.from_latlon(lat, long)      # devuelve
            ↪ easting, northing
36              xo,yo = xo - x_ref , yo - y_ref             #Me hago mi
            ↪ sistema local
37
38          else:
39              x,y,_,_ = utm.from_latlon(lat, long)
40              x,y  = x - x_ref , y - y_ref                #Me hago mi
            ↪ sistema local
41
42              dist_relativas = np.linalg.norm( np.asarray([xo,yo]) -
            ↪ np.asarray([x, y]))  # distancia euclidea de posiciones
            ↪  relativas
43              dist_acum += dist_relativas
44
45              if dist_relativas >= dist_muestreo:
46                  idx.append(i)
47                  dist_acum = 0
48                  data.append((x, y))
49                  xo, yo = x, y  # actualizas tu siguiente medida
50
51      return np.array(data)
52
53  def leer_coordenadas():
54      global x2,y2
55
56      data = sampler_gps()
57      x2 = data[:,0]
58      y2 = data[:,1]
59
60      if config['follow_reverse']:
61          x2 = np.flip(x2)
62          y2 = np.flip(y2)
63      if config['recover_route']:
64          #Obtengo coordenadas actuales:
65          x_i,y_i, _, _ = utm.from_latlon(pose.latitud, pose.longitud
            ↪ )
66          x,y  = x_i - x_ref , y_i - y_ref
67          #Miro diferencias con la ruta a seguir:
68          dif_x = x2 - x
69          dif_y = y2 - y
70          d = np.linalg.norm(np.array([dif_x, dif_y]), axis=0)
71          i = np.argmin(d)
72          #Elimino coordenadas anteriores
73          x2 = x2[i:]
74          y2 = y2[i:]
75
76
77  class get_pose_current():
78      def __init__(self):
79          self.latitud  = 0.0
80          self.longitud = 0.0
81          self.message_received = False
82
83      def listener(self):
84          self.sub=rospy.Subscriber('/gnss/fix' , NavSatFix, self.
            ↪ callback_pos) #CAMBIAR
```

```
85
86      def callback_pos(self, data):
87          self.latitud = data.latitude
88          self.longitud = data.longitude
89          self.message_received = True
90
91  def callback_path(path_msg):
92      #Recibo un nuevo camino a seguir --> actualizo x2,y2
93      global x2,y2
94      x2, y2= [],[]
95
96      for pose_stamped in path_msg.poses:
97          x2.append(pose_stamped.pose.position.x)
98          y2.append(pose_stamped.pose.position.y)
99
100 def talker(x, z):
101     pub.publish(Twist(linear=Vector3(x=x), angular=Vector3(z=z)))
102
103 def publish_path(x,y):
104
105     # Construye el mensaje de tipo Path
106     path_msg.header.stamp = rospy.Time.now()
107     path_msg.header.frame_id = "map"
108
109     pose = PoseStamped()
110     pose.header.stamp = rospy.Time.now()
111     pose.header.frame_id = "map"
112     pose.pose.position.x = x
113     pose.pose.position.y = y
114     pose.pose.position.z = 0
115     path_msg.poses.append(pose)
116     path_publisher.publish(path_msg)
117
118 def Control():
119     global x_ant,y_ant,alpha_prev,d_prev,v_ant,i,x2,y2 ,o_act
120
121     #SISTEMA LOCAL
122     x_i,y_i, _, _ = utm.from_latlon(pose.latitud, pose.longitud)
123     x,y  = x_i - x_ref , y_i - y_ref
124     publish_path(x,y)
125
126     #DISTANCIA DESPLAZADA EN CADA ITERACION
127     d_desplazado = math.hypot(x - x_ant, y - y_ant)
128     o_des = math.atan2((y2[0]-y),(x2[0]-x))
129
130     if d_desplazado > 0.1:
131         o_act = math.atan2((y-y_ant),(x-x_ant))
132         #SI no me he desplazado, mi orientacion actual no la voy a
            ↪ calcular bien --> me quedo con la que tenia
133
134     #ERROR ANGULAR
135     o_act , o_des = (o_act + 2*math.pi)%(2*math.pi) , (o_des + 2*
            ↪ math.pi)%(2*math.pi)  #[-pi,pi] --> [0,2pi]
136     alpha = o_des - o_act
137     alpha = ((alpha+math.pi) % (2*math.pi) ) - math.pi   #[0,2pi]
            ↪ --> [-pi,pi]
138     #E_ori en rango [-pi,pi] corrijo posibles saltos de 2pi
139
```

```
140     #ERROR LINEAL
141     d = math.hypot(x2[0] - x, y2[0] - y)
142 
143     #CONTROL
144     v = config['kd']*(d)*math.cos(alpha)
145     w = config['kd']*math.cos(alpha)*math.sin(alpha) + config['kang
            ↪ ']*alpha
146 
147     #LIMITO RANGO
148     v = np.clip(v,config['vmin'],config['vmax'])
149     w = np.clip(w,config['wmin'],config['wmax'])
150 
151     #PUBLICO
152     talker(v,w)
153 
154     #GUARDO ERRORES ANTERIORES
155     alpha_prev, d_prev, v_ant = alpha , d , v
156     if d_desplazado > 0.1 :
157         x_ant,y_ant = x,y
158 
159     #PUNTO ALCANZADO ??
160     if  d < config['precision'] :          #Elimino la primera
            ↪ coordenada objetivo y desplazo a la izq
161         print('\nPUNTO x =',x,'    y = ',y, 'ALCANZADO\n')
162         x2 = x2[1:]
163         y2 = y2[1:]
164 
165 
166 if __name__ == '__main__':
167 
168     rospy.init_node('control_system',anonymous=True)
169 
170     #Config
171     r = rospy.Rate(10)
172     pub = rospy.Publisher('/husky_velocity_controller/cmd_vel',
            ↪ Twist , queue_size=10)
173     path_publisher = rospy.Publisher('/husky/path', Path,
            ↪ queue_size=10)
174     path_pub = rospy.Publisher('/husky/global_path', Path,
            ↪ queue_size=10)
175 
176     pose = get_pose_current()
177     pose.listener()
178     path_msg = Path()
179     path_global = Path()
180     time_ref = rospy.Time.now().to_sec()
181 
182     print("Esperando gps...")
183     while not pose.message_received:
184         pass
185 
186     leer_coordenadas()
187     while not rospy.is_shutdown():
188         Control()
189     r.sleep()
```

Código A.1: Nodo de control

En segundo lugar, se muestra el código que se encarga de la planificación global basada en el algoritmo Dijkstra. Se ha partido de [51] para formar está función que devuelve el camino óptimo a partir de dos coordenadas GPS.

```
import osmnx as ox
import networkx as nx
import plotly.graph_objects as go
import numpy as np
import pandas as pd

def plot_path(lat, long, origin_point, destination_point):
    fig = go.Figure(go.Scattermapbox(
        name="Path",
        mode="lines",
        lon=long,
        lat=lat,
        marker={'size': 10},
        line=dict(width=4.5, color='blue')))

    fig.add_trace(go.Scattermapbox(
        name="Source",
        mode="markers",
        lon=[origin_point[1]],
        lat=[origin_point[0]],
        marker={'size': 12, 'color': "red"}))

    fig.add_trace(go.Scattermapbox(
        name="Destination",
        mode="markers",
        lon=[destination_point[1]],
        lat=[destination_point[0]],
        marker={'size': 12, 'color': 'green'}))

    lat_center = np.mean(lat)
    long_center = np.mean(long)

    fig.update_layout(mapbox_style="open-street-map",
                      mapbox_center_lat=30, mapbox_center_lon=-80
        ↪ )
    fig.update_layout(margin={"r": 0, "t": 0, "l": 0, "b": 0},
                      mapbox={
                          'center': {'lat': lat_center,
                                     'lon': long_center},
                          'zoom': 13})
    fig.show()

#Generar las lineas que forman el camino
def node_list_to_path(G, node_list):
    edge_nodes = list(zip(node_list[:-1], node_list[1:]))
    lines = []
    for u, v in edge_nodes:
        data = min(G.get_edge_data(u, v).values(),
                   key=lambda x: x['length'])
        if 'geometry' in data:
            xs, ys = data['geometry'].xy
            lines.append(list(zip(xs, ys)))
```

```
52          else:
53              x1 = G.nodes[u]['x']
54              y1 = G.nodes[u]['y']
55              x2 = G.nodes[v]['x']
56              y2 = G.nodes[v]['y']
57              line = [(x1, y1), (x2, y2)]
58              lines.append(line)
59      return lines
60
61
62  def Dijkstra(orig_lat,orig_lon,dest_lat,dest_lon,input_osm_file,
        ↪ plot_route,output_csv,clicked_path_folder):
63
64      # Cargo el .OSM file (nodes and edges)
65      filepath=input_osm_file
66      G= ox.graph.graph_from_xml(filepath, bidirectional=True,
        ↪ simplify=False, retain_all=True)
67
68      # Cargo los puntos de origen y destino
69      origin_point=(orig_lat, orig_lon)
70      destination_point=(dest_lat, dest_lon)
71
72      # Obtener el nodo mas cercano a los puntos de origen y destino
73      origin_node=ox.distance.nearest_nodes(G, X=origin_point[1], Y=
        ↪ origin_point[0], return_dist=False)
74      destination_node=ox.distance.nearest_nodes(G, X=
        ↪ destination_point[1], Y=destination_point[0],
        ↪ return_dist=False)
75
76      print('Origin node: ')
77      print(origin_node)
78      print('Destination node: ')
79      print(destination_node)
80
81      # Optimal path
82      route = nx.shortest_path(G, origin_node, destination_node,
        ↪ weight='length')
83
84      # Obtener nodos
85      lines = node_list_to_path(G, route)
86      long2, lat2, data = [], [], []
87
88      for i in range(len(lines)):
89          z = list(lines[i])
90          l1 = list(list(zip(*z))[0])
91          l2 = list(list(zip(*z))[1])
92
93          for j in range(len(l1)):
94              long2.append(l1[j])
95              lat2.append(l2[j])
96              if(j==0):
97                  data.append([l1[0], l2[0]])
98                  if(i==(len(lines)-1)):
99                      data.append([l1[1],l2[1]])
100
101
102     # Plot on the open street map
103     if plot_route:
```

```
104         plot_path(lat2, long2, origin_point, destination_point)
105 
106     #Save the GPs coordinates in a .csv file
107     if output_csv:
108         d = pd.DataFrame(data)
109         file_name = clicked_path_folder+'path.csv'
110         d.to_csv(file_name, header=['latitude', 'longitude'], index
             ↪ =False)
111 
112     return lat2,long2
```

Código A.2: Planificador global basado en algoritmo Dijkstra

En último lugar, se presenta el código empleado para hacer uso de Rviz como herramienta para el usuario. A partir del tópico `/clicked_point` se obtiene la posición seleccionada por el usuario y se calcula la trayectoria óptima a seguir llamando al planificador global.

```
1  import rospy
2  import sys
3  sys.path.append("..")  # Salir del directorio actual
4 
5  from include.path_planning.main import Dijkstra #Dijkstra
6  from geometry_msgs.msg import PointStamped
7  from sensor_msgs.msg import NavSatFix
8  from nav_msgs.msg import Path
9  from geometry_msgs.msg import PoseStamped
10 
11 import yaml
12 import utm
13 #Transformada
14 from tf.transformations import euler_from_quaternion,
          ↪ quaternion_from_euler
15 
16 try:
17     with open(r'../config/config.yaml') as file:
18         config = yaml.load(file, Loader=yaml.FullLoader)
19 except:
20     print("YAML loading error!...")
21 
22 class get_target_position():
23     def __init__(self):
24         self.x=0.0
25         self.y=0.0
26         self.nuevo_dato = 0
27     def listener(self):
28         self.sub=rospy.Subscriber("/clicked_point" , PointStamped,
           ↪ self.callback_pos)
29 
30     def callback_pos(self,data):
31         self.x = data.point.x
32         self.y = data.point.y
33         self.nuevo_dato = 1
34 
```

```
class get_pose_current():
    def __init__(self):
        self.latitud =0.0
        self.longitud=0.0

    def listener(self):
        self.sub=rospy.Subscriber('/navsat/fix' , NavSatFix, self.
         ↪ callback_pos)

    def callback_pos(self, data):
        self.latitud = data.latitude
        self.longitud = data.longitude

path_msg = Path()

if __name__ == '__main__':

    rospy.init_node('clicked_point_subscriber',anonymous=True)

    #Config
    pose = get_pose_current()
    pose.listener()
    target_position = get_target_position()
    target_position.listener()
    r = rospy.Rate(10)

    #Referencia
    x_ref,y_ref, Num, Letra = utm.from_latlon(config['latitud_ref'
         ↪ ],config['longitud_ref'])

    while not rospy.is_shutdown():
        x = target_position.x
        y = target_position.y


        if( target_position.nuevo_dato == 1):
            global nuevo_dato
            target_position.nuevo_dato = 0

            utm_x = x_ref + target_position.x
            utm_y = y_ref + target_position.y

            lat_des,lon_des = utm.to_latlon(utm_x, utm_y ,Num,Letra
         ↪ )
            print("lat,lon origen:",pose.latitud,pose.longitud)
            print("lat,lon destino:",lat_des, lon_des)

            #OBTENGO CAMINO
            lat,lon = Dijkstra(pose.latitud , pose.longitud,lat_des
         ↪ ,lon_des,config['input_osm_file'],config['plot_route'],
         ↪ config['output_csv'],config['clicked_path_folder'])
            print ("lat:",lat,"lon:",lon)
            x = []
            y = []
            i=0
            global_path = Path()

            for i in range(len(lat)):
```

```
88
89                  pointx,pointy,Num2,Letra2 = utm.from_latlon(lat[i],
        ↪ lon[i])
90                  pointx,pointy = pointx - x_ref , pointy - y_ref
91                  x.append(float(pointx))
92                  y.append(float(pointy))
93
94                  tmp_pose = PoseStamped()
95                  tmp_pose.pose.position.x = float(pointx)
96                  tmp_pose.pose.position.y = float(pointy)
97                  global_path.poses.append(tmp_pose)
98
99                  i = i+1
100
101             global_path.header.frame_id = "map"
102             pubPath = rospy.Publisher('/husky/globalPath', Path,
        ↪ queue_size=10, latch=True)
103             pubPath.publish(global_path)
104
105         r.sleep()
```

Código A.3: Uso de Rviz como interfaz para seleccionar el punto de destino

B. Anexo II

En este segundo anexo se incluyen los códigos de ROS desarrollados para el seguimiento de trayectorias teniendo en cuenta los obstáculos presentes en el camino.

En primer lugar, se muestra el servicio de ROS que se encarga de analizar la nube de puntos obtenida por el sensor LIDAR y clasificar los puntos en transversables y no transversables mediante una red neuronal convolucional.

```
import torch
import torch.nn as nn
from torch.optim import SGD
import MinkowskiEngine as ME
import numpy as np
import open3d as o3d
from TeNeXt.minkUnet_custom import TeNeXtA
import time
import rospy
from sensor_msgs.msg import PointCloud2
import sensor_msgs.point_cloud2 as pc2
from std_msgs.msg import Header
from sensor_msgs.msg import PointField
import struct

from control_system.srv import Traversability,
    ↪ TraversabilityResponse
import struct

class TraversabilityService:
    def __init__(self):
        self.device = torch.device('cpu')
        self.root = "TeNeXt/BestModel9_th_0.4862739620804787
    ↪ voxel_size0.2_0.9197958031905511.pth"
        self.model = TeNeXtA(1, 1).to(self.device)
        self.model.load_state_dict(torch.load(self.root,
    ↪ map_location=torch.device('cpu')))
        self.criterion = nn.BCELoss()
        self.optimizer = SGD(self.model.parameters(), lr=1e-1)
        self.threshold = 0.4862739620804787
        self.voxel_size = 0.2

        self.pub = rospy.Publisher("trav_analysis", PointCloud2
    ↪ , queue_size=2)

    def handle_traversability(self, req):
        start = time.time()
        field_names = [field.name for field in req.input.fields
    ↪ ]
        points = list(pc2.read_points(req.input, skip_nans=True
    ↪ , field_names=field_names))

        if len(points) == 0:
            rospy.logwarn("Received an empty cloud")
```

```
39                 return TraversabilityResponse()
40             else:
41                 pcd_array = np.asarray(points)
42                 pointcloud = o3d.geometry.PointCloud()
43                 pointcloud.points = o3d.utility.Vector3dVector(
        ↪ pcd_array[:, 0:3])
44                 coords_orig = np.asarray(pointcloud.points)
45                 coords = ME.utils.batched_coordinates([coords_orig
        ↪ / self.voxel_size], dtype=torch.float32)
46                 self.features = np.ones((coords_orig.shape[0], 1))
47
48                 # Inference
49                 test_in_field = ME.TensorField(torch.from_numpy(
        ↪ self.features).to(dtype=torch.float32),
50                                                coordinates=coords,
51                                                quantization_mode=ME
        ↪ .SparseTensorQuantizationMode.UNWEIGHTED_AVERAGE,
52                                                minkowski_algorithm=
        ↪ ME.MinkowskiAlgorithm.SPEED_OPTIMIZED,
53                                                device=self.device)
54                 test_output = self.model(test_in_field.sparse())
55                 logit = test_output.slice(test_in_field)
56                 pred_raw = logit.F.detach().cpu().numpy()
57                 pred = np.where(pred_raw > self.threshold, 1, 0)
58
59                 points = self.visualize_each_cloud(pred,
        ↪ coords_orig)
60                 header = Header()
61                 header.stamp = rospy.Time.now()
62                 header.frame_id = "base_link"
63                 fields = [PointField('x', 0, PointField.FLOAT32, 1)
        ↪ ,
64                           PointField('y', 4, PointField.FLOAT32, 1)
        ↪ ,
65                           PointField('z', 8, PointField.FLOAT32, 1)
        ↪ ,
66                           PointField('rgb', 16, PointField.UINT32,
        ↪ 1)]
67
68                 # Built cloud and publish it
69                 self.pc2_trav=pc2.create_cloud(header,fields,points
        ↪ )
70                 self.pub.publish(self.pc2_trav)
71                 # Separate traversable and non-traversable points
72                 transversable_points = [pt for pt, pred_label in
        ↪ zip(points, pred) if pred_label == 1]
73                 no_transversable_points = [pt for pt, pred_label in
        ↪  zip(points, pred) if pred_label == 0]
74
75                 # Create PointCloud2 messages
76                 transversable_points_msg = pc2.create_cloud(header,
        ↪  fields, transversable_points)
77                 no_transversable_points_msg = pc2.create_cloud(
        ↪ header, fields, no_transversable_points)
78
79                 stop = time.time()
80                 duration = stop - start
81                 rospy.loginfo(f"Processing duration: {duration}")
```

```
82
83                  return TraversabilityResponse(trav_points=
        ↪ transversable_points_msg,
84                                                no_trav_points=
        ↪ no_transversable_points_msg)
85
86          def visualize_each_cloud(self, pred, coords):
87              points = []
88              for k, i in enumerate(pred):
89                  if i == 1:
90                      r, g, b, a = 95, 158, 160, 255
91                  else:
92                      r, g, b, a = 106, 90, 205, 255
93                  rgb = struct.unpack('I', struct.pack('BBBB', b, g,
        ↪ r, a))[0]
94                  pt = [coords[k, 0], coords[k, 1], coords[k, 2], rgb
        ↪ ]
95                  points.append(pt)
96              return points
97
98      if __name__ == '__main__':
99          rospy.init_node('traversability_service')
100         service = TraversabilityService()
101         rospy.Service('traversability_analysis', Traversability,
        ↪ service.handle_traversability)
102         rospy.loginfo('Traversability service ready.')
103         rospy.spin()
```

Código B.1: Servicio de ROS: análisis de la nube de puntos mediante redes neuronales

En segundo lugar, se muestra el código que se encarga de local basado en el algoritmo RRT. Este se encarga de generar una ruta entre waypoints teniendo en cuenta los obstáculos presentes en el camino.

```
1       import matplotlib.pyplot as plt
2       from include.RTTPlanner.rrtplanner.rrtplannerPC import
            ↪ RRTPlannerPC
3       from include.RTTPlanner.rrtplanner.trajectorysmoother import
            ↪ TrajectorySmoother
4       import rospy
5       from sensor_msgs.msg import PointCloud2
6       import time
7
8       class points_traversable:
9           def __init__(self):
10              self.points = None
11              self.message_received = False
12
13          def callback(self, data):
14              self.points = data
15              self.message_received = True
16
17          def listener(self):
18              self.sub=rospy.Subscriber('/trav_points', PointCloud2,
            ↪ self.callback)
```

```
19
20      class points_obstacles:
21          def __init__(self):
22              self.points = None
23              self.sub = None
24              self.message_received = False
25
26          def callback(self, data):
27              self.points = data
28              self.message_received = True
29
30          def listener(self):
31              self.sub=rospy.Subscriber('/no_trav_points', PointCloud
            ↪ 2, self.callback)
32
33
34      def find_path(start,goal,tranversable,obstacles):
35          start_time = time.time()
36          planner = RRTPlannerPC(start=start,
37                                 goal=goal,
38                                 pc_traversable=tranversable,
39                                 pc_obstacles=obstacles,
40                                 epsilon=0.4,
41                                 robot_radius=1,
42                                 max_nodes=1000)
43
44
45          if planner.goal_colision():
46              print("Goal collision detected. Aborting path planning.
            ↪ ")
47              return None, None, False, True
48          else:
49              tree = planner.build_rrt_connect_to_goal()
50              planner.print_info()
51              path_found = planner.get_solution_path(tree)
52              smoother = TrajectorySmoother(s=1)
53              path_smooth = smoother.smooth3D(path_found) #Error m >
            ↪ k must hold --> revisar ruta path_found
54
55              stop = time.time()
56              duration = stop - start_time
57
58              #Plot the tree and solution
59              tree.plot(label = 'Tree')
60              tree.plot_path(path=path_found, color='yellow', label='
            ↪ Solution Path', linewidth=2)
61              tree.plot_path(path=path_smooth, color='cyan', label='
            ↪ Smooth Path', linewidth=3)
62
63              #Plot obstacles, start and goal
64              planner.plot()
65              plt.legend(loc='upper right')
66              # plt.legend([])
67              # update plots
68              plt.show(block=True)
69
70              print('FINISHED')
71              rospy.loginfo(f"Processing duration: {duration}")
```

```
72
73              x3 = path_smooth[:,0]
74              y3 = path_smooth[:,1]
75
76              return x3,y3,planner.goal_reached, False
```

Código B.2: Planificador local basado en algoritmo RRT

En tercer y último lugar, se muestra el código final con todos los aspectos mostrados previamente integrados. Primeramente, se obtiene la nube de puntos de la posición de partida. Esta se analiza mediante el servicio de ROS previamente creado. Posteriormente, se planifica una ruta hasta el próximo waypoint con el planificador local. Finalmente, se controla el robot para que siga la trayectoria planificada evitando los obstáculos presentes en el camino.

```
1  import rospy
2  import utm
3  from geometry_msgs.msg import Twist,PoseStamped, Vector3
4  from sensor_msgs.msg import NavSatFix
5  from nav_msgs.msg import Path
6  import math
7  import yaml
8  import numpy as np
9  from RRT import find_path as rrt
10
11 try:
12     with open(r'config/config.yaml') as file:
13         config = yaml.load(file, Loader=yaml.FullLoader)
14 except:
15     print("YAML loading error!...")
16
17 import open3d as o3d
18 from sensor_msgs.msg import PointCloud2
19 import sensor_msgs.point_cloud2 as pc2
20 from  control_system.srv import Traversability,
          ↪ TraversabilityRequest
21
22 p_alcanzado = False
23 field_names = "['x', 'y', 'z']"
24
25 #REFERENCIAS
26 x_ref,y_ref, _, _ = utm.from_latlon(config['latitud_ref'],config['
          ↪ longitud_ref'])
27 i,alpha_prev,d_prev = 0 , 0.0 , 0.0
28 x_ant, y_ant, v_ant = -0.01 , -0.01 ,0
29 o_act = 0
30
31 class get_pose_current():
32     def __init__(self):
33         self.latitud  = 0.0
34         self.longitud = 0.0
35         self.message_received = False
```

```
36
37      def listener(self):
38          self.sub=rospy.Subscriber('/navsat/fix' , NavSatFix, self.
             ↪ callback_pos)
39
40      def callback_pos(self, data):
41          self.latitud = data.latitude
42          self.longitud = data.longitude
43          self.message_received = True
44
45  class Point_Cloud:
46      def __init__(self):
47          self.points = None
48          self.message_received = False
49
50      def callback(self, data):
51          self.message_received = True
52          self.points = data
53
54      def listener(self):
55          self.sub=rospy.Subscriber('/os1_cloud_node/points',
             ↪ PointCloud2, self.callback)
56
57  def call_traversability_service(cloud):
58      print("Llamando al servicio de análisis de puntos...")
59      rospy.wait_for_service('traversability_analysis')
60      try:
61          traversability_service = rospy.ServiceProxy('
             ↪ traversability_analysis', Traversability)
62          req = TraversabilityRequest(input=cloud)
63          resp = traversability_service(req)
64          return resp.trav_points, resp.no_trav_points
65      except rospy.ServiceException as e:
66          rospy.logerr(f"Service call failed: {e}")
67
68  def my_position():
69      x_i,y_i, _, _ = utm.from_latlon(pose.latitud, pose.longitud)
70      x,y  = x_i - x_ref , y_i - y_ref   #Me hago mi sistema local
71      z = 0
72      return x,y,z
73
74  def points():
75      # Llamar al servicio de análisis de puntos
76      trav, no_trav = call_traversability_service(cloud.points)
77
78      trav = list(pc2.read_points(trav, skip_nans=True, field_names=
             ↪ field_names))
79      trav_points = np.asarray(trav)
80
81      no_trav = list(pc2.read_points(no_trav, skip_nans=True,
             ↪ field_names=field_names))
82      no_trav_points = np.asarray(no_trav)
83
84      #Convertir a pc3
85      pc_trav = o3d.geometry.PointCloud()
86      pc_obs = o3d.geometry.PointCloud()
87
88      pc_trav.points = o3d.utility.Vector3dVector(np.asarray(
```

```
            ↪ trav_points))
89      pc_obs.points = o3d.utility.Vector3dVector(np.asarray(
            ↪ no_trav_points))
90
91      # Crear la matriz de transformación
92      cos_theta = np.cos(o_act)
93      sin_theta = np.sin(o_act)
94
95      # Matriz de transformación 4x4
96      T = np.eye(4)
97      T[:3, :3] = np.array([[cos_theta, -sin_theta, 0],
98                            [sin_theta, cos_theta, 0],
99                            [0, 0, 1]])
100
101     T[:3, 3] = np.asarray(my_position())
102     T[2, 3] = -0.7 #Altura del LIDAR respecto al suelo
103
104     #Aplicar la transformación a las nubes de puntos
105     pc_trav.transform(T)
106     pc_obs.transform(T)
107
108     #Convertir de nuevo a numpy array
109     tranversable = np.asarray(pc_trav.points)
110     obstacles = np.asarray(pc_obs.points)
111
112     return tranversable, obstacles
113
114 def global_path(x2,y2):
115     # Construye el mensaje de tipo Path
116     path_global.header.stamp = rospy.Time.now()
117     path_global.header.frame_id = "odom"
118
119     pose = PoseStamped()
120     pose.header.stamp = rospy.Time.now()
121     pose.header.frame_id = "odom"
122     pose.pose.position.x = x2
123     pose.pose.position.y = y2
124     pose.pose.position.z = 0
125     path_global.poses.append(pose)
126     path_pub.publish(path_global)
127
128 def talker(x, z):
129     pub.publish(Twist(linear=Vector3(x=x), angular=Vector3(z=z)))
130
131 def publish_path(x,y):
132     # Construye el mensaje de tipo Path
133     path_msg.header.stamp = rospy.Time.now()
134     path_msg.header.frame_id = "odom"
135
136     pose = PoseStamped()
137     pose.header.stamp = rospy.Time.now()
138     pose.header.frame_id = "odom"
139     pose.pose.position.x = x
140     pose.pose.position.y = y
141     pose.pose.position.z = 0
142     path_msg.poses.append(pose)
143     path_publisher.publish(path_msg)
144
```

```
145 def control(x2,y2):
146     global x_ant,y_ant,alpha_prev,d_prev,v_ant,i,o_act,p_alcanzado
147     x,y,z = my_position()
148     publish_path(x,y)
149 
150     #DISTANCIA
151     d_desplazado = math.hypot(x - x_ant, y - y_ant)
152     d_destino = math.hypot(x2[0] - x, y2[0] - y)
153     o_des = math.atan2((y2[0]-y),(x2[0]-x))
154 
155     #SI no me he desplazado, mi orientacion actual no la voy a
            ↪ calcular bien --> me quedo con la que tenia
156     if d_desplazado > 0.1 :
157         o_act = math.atan2((y-y_ant),(x-x_ant))
158 
159     #ERROR ANGULAR
160     o_act , o_des = (o_act + 2*math.pi)%(2*math.pi) , (o_des + 2*
            ↪ math.pi)%(2*math.pi)  #[-pi,pi] --> [0,2pi]
161     alpha = o_des - o_act
162     alpha = ((alpha+math.pi) % (2*math.pi) ) - math.pi   #[0,2pi]
            ↪ --> [-pi,pi]
163 
164     #ERROR LINEAL
165     d = math.hypot(x2[0] - x, y2[0] - y)
166 
167     #CONTROL
168     v = config['kd']*(d)*math.cos(alpha)
169     w = config['kd']*math.cos(alpha)*math.sin(alpha) + config['kang
            ↪ ']*alpha
170 
171     #LIMITO RANGO
172     v = np.clip(v,config['vmin'],config['vmax'])
173     w = np.clip(w,config['wmin'],config['wmax'])
174 
175     #PUBLICO
176     talker(v,w)
177 
178     #ME GUARDO ERRORES ANTERIORES
179     alpha_prev, d_prev, v_ant = alpha , d , v
180     if d_desplazado > 0.1 :
181         x_ant,y_ant = x,y
182 
183     # HE LLEGADO ??
184     if  d_destino < config['precision'] :
185         print('\nPUNTO ALCANZADO\n')
186         if(len(x2) == 1):
187             p_alcanzado = True
188         else:
189             x2 = x2[1:]
190             y2 = y2[1:]
191             global_path(x2[0],y2[0])
192             print("x3:",x2[0],"y3",y2[0])
193 
194     return x2,y2
195 
196 
197 
198 
```

```
199 if __name__ == '__main__':
200
201     rospy.init_node('control',anonymous=True)
202     r = rospy.Rate(10)
203
204     pub = rospy.Publisher('/husky_velocity_controller/cmd_vel',
            ↪ Twist , queue_size=10)
205     path_publisher = rospy.Publisher('/husky/path', Path,
            ↪ queue_size=10)
206     path_pub = rospy.Publisher('/global_path', Path, queue_size=10)
207
208     path_msg = Path()
209     path_global = Path()
210     pose = get_pose_current()
211     pose.listener()
212
213     cloud = Point_Cloud()
214     cloud.listener()
215
216     print("Esperando gps...")
217     while not pose.message_received:
218         pass
219
220     print("Esperando puntos...")
221     while(not cloud.message_received or cloud.points is None):
222         pass
223
224     while not rospy.is_shutdown():
225         start = np.asarray(my_position())
226         goal = [x2[0],y2[0],0]
227
228         tranversable,obstacles = points()
229
230         print("MY POSITION:",start, "GOAL:",goal)
231         x3,y3,reached,colision = rrt(start,goal,tranversable,
             ↪ obstacles)
232
233         if colision or x3 is None:
234             x2,y2 = x2[1:],y2[1:]  #Me salto este objetivo
235             print("Cambio de objetivo...")
236
237         elif not reached:
238             print("Objetivo no se alcanza")
239             x3,y3 = x3[:-3],y3[:-3]
240             #Elimino los dos ultimos puntos para q no se acerque
             ↪ mucho a la zona no visible
241
242             while not p_alcanzado:
243                 x3,y3 = control(x3,y3)
244
245             p_alcanzado = False
246             start = np.asarray(my_position())
247             goal = [x2[0],y2[0],0]  #No cambio de objetivo
248
249             tranversable,obstacles = points()
250             x3,y3,reached,colision = rrt(start,goal,tranversable,
             ↪ obstacles)
251
```

```
252             while not p_alcanzado:
253                 x3,y3 = control(x3,y3)
254 
255             p_alcanzado = False
256             x2,y2 = x2[1:],y2[1:]
257             print("x2:",x2[0],"y2",y2[0])
258 
259         else:
260             while not p_alcanzado:
261                 x3,y3 = control(x3,y3)
262 
263             p_alcanzado = False
264             x2,y2 = x2[1:],y2[1:]
265             print("x2:",x2[0],"y2",y2[0])
266 
267     r.sleep()
```

Código B.3: Sistema de navegación autónoma