UNIVERSIDAD DE MÁLAGA
ESCUELA DE INGENIERÍAS INDUSTRIALES
DEPARTAMENTO DE INGENIERÍA DE SISTEMAS Y AUTOMÁTICA

GRADO EN INGENIERÍA ELECTRÓNICA, ROBÓTICA Y MECATRÓNICA

Curso 2023/2024

Trabajo Fin de Grado

NAVEGACIÓN LOCAL CON VEHÍCULOS TERRESTRES EMPLEANDO MAPAS PROBABILÍSTICOS 3D

(Reactive navigation for ground vehicles using 3D probabilistic maps)

AUTOR: Violeta Prieto Miedes
TUTOR: Dr. D. Ricardo Vázquez Martín
COTUTOR: D. Dahui Lin Yang

Málaga, Enero de 2024

Resumen

Este Trabajo Fin de Grado (TFG) aborda el problema de navegación local con percepción tridimensional del entorno, empleando métodos de representación interna que alivian la carga computacional de mapas de nubes de puntos.

Centrándose en la navegación en entornos terrestres, este trabajo enfrenta desafíos únicos como la variabilidad de los entornos y obstáculos estáticos y dinámicos. La introducción de mapas probabilísticos 3D representa un avance significativo sobre los métodos tradicionales de mapeo 2D, ofreciendo una representación más rica y detallada del entorno que mejora la planificación de rutas y la evitación de obstáculos.

Se desarrollará un sistema de navegación local autónoma para vehículos terrestres utilizando mapas probabilísticos 3D, con el fin de permitir una navegación segura y eficiente en entornos desconocidos y dinámicos. Esto implicará integrar el Robot Operating System (ROS) para la gestión del vehículo, crear y utilizar mapas tridimensionales, implementar un algoritmo de navegación reactiva para la evitación de obstáculos, y utilizar tecnologías como Gazebo, Husky, Octomap y Octrees. Además, se realizarán pruebas en diferentes entornos para validar la robustez y precisión del sistema.

Abstract

This Final Degree Project (TFG) addresses the problem of local navigation with three-dimensional perception of the environment, using internal representation methods that alleviate the computational load of point cloud maps.

Focusing on navigation in terrestrial environments, this work faces unique challenges such as the variability of environments and static and dynamic obstacles. The introduction of 3D probabilistic maps represents a significant advancement over traditional 2D mapping methods, offering a richer and more detailed representation of the environment that improves route planning and obstacle avoidance.

An autonomous local navigation system for terrestrial vehicles will be developed using 3D probabilistic maps, in order to enable safe and efficient navigation in unknown and dynamic environments. This will involve integrating the Robot Operating System (ROS) for vehicle management, creating and using three-dimensional maps, implementing a reactive navigation algorithm for obstacle avoidance, and using technologies like Gazebo, Husky, Octomap, and Octrees. In addition, tests will be conducted in different environments to validate the robustness and accuracy of the system.

Siglas y Acrónimos

#
3D: 3 dimensiones.

A
ACML: Adaptive Monte Carlo Localization
AI Lab: Artificial Intelligence Laboratory

C
CNN: Convolutional Neural Network

F
FOV: Field Of View

H
HAL: Hardware Abstraction Layer

L
LiDAR: Light Detection and Ranging o Laser Imaging Detection and Ranging

P
PRM: Probabilistic Roadmap

R
RGB: Red, Green, Blue
RGB-D: Red, Green, Blue - Depth
ROS: Robot Operating System
ROSCon: Robot Operating System Conference
RRT: Rapidly Exploring Random Trees
RSF: Robotic Software Framework

S
SLAM: Simultaneous Localization and Mapping

U
UGV: Unmanned Ground Vehicle

Indice general

Indice de Figuras

Capítulo 1

Introducción

En el contexto general, la robótica móvil ha experimentado un crecimiento exponencial en las últimas décadas, impulsada por avances en inteligencia artificial y procesamiento de datos. Este campo ha evolucionado desde simples sistemas de navegación basados en sensores hasta complejas plataformas autónomas capaces de interactuar con su entorno de manera inteligente e independiente.

Desde sus inicios en la década de 1960, la robótica móvil ha avanzado significativamente. Los primeros robots eran principalmente teleoperados, mientras que los desarrollos modernos han llevado a sistemas altamente autónomos. En la navegación autónoma, se ha presenciado un cambio de los sistemas de navegación basados en reglas simples a complejas plataformas capaces de aprender y adaptarse a su entorno [1].

La navegación autónoma de vehículos ha sido un área de investigación intensiva, con aplicaciones que van desde vehículos autónomos en carretera hasta drones aéreos. En particular, este trabajo, se centrará en la navegación en entornos terrestres. Este tipo de navegación presenta desafíos únicos debido a la variabilidad y la complejidad de los entornos, que pueden incluir obstáculos estáticos y dinámicos, variaciones en el terreno y condiciones cambiantes de iluminación y clima. Un sistema de navegación terrestre autónomo debe considerar todos estos factores.

En cuanto al uso de mapas probabilísticos 3D, suponen un avance significativo sobre los enfoques de mapeo 2D tradicionales. Proporcionan una representación más rica y detallada del entorno, lo que se traduce en una mejor planificación de ruta y evasión de obstáculos. El uso de este tipo de mapas probabilísticos 3D ha demostrado mejorar la precisión y la robustez en la navegación autónoma.

1.1 Justificación y motivación

Este proyecto se justifica por varias razones clave:

- Avance Tecnológico en Robótica Móvil: La robótica móvil está en la vanguardia de la tecnología y la innovación. Con este proyecto, se busca contribuir al desarrollo de sistemas de navegación más avanzados y eficientes, especialmente en entornos tridimensionales y dinámicos.

- Aplicaciones Prácticas o con Potencial Comercial: La navegación autónoma de vehículos terrestres tiene un enorme potencial en numerosas aplicaciones, desde logística y transporte hasta exploración y rescate.

- Contribución a la Seguridad y Eficiencia: Mejorar la capacidad de los vehículos para navegar de manera autónoma y segura en entornos desconocidos puede reducir los riesgos asociados con la intervención humana y aumentar la eficiencia operativa.

- Innovación en Mapeo y Percepción del Entorno: El uso de mapas probabilísticos 3D y sistemas como OctoMap y Octrees representa un paso adelante en la precisión y la eficacia con la que los vehículos autónomos pueden percibir y entender su entorno.

La motivación para emprender este proyecto surgió de varios factores. El primero la pasión por la Robótica y la tecnología, especialmente en cómo los sistemas autónomos pueden transformar el mundo, desde agilizar tareas de la vida cotidiana hasta manejar grandes toneladas en una fábrica o, en el ámbito médico, el salvamento de personas. Por otro lado, este Trabajo Fin de Grado, ha supuesto un gran desafío técnico y de aprendizaje. Resulta de gran atractivo la posibilidad de desarrollar soluciones innovadoras que puedan superar todos los desafíos propuestos.

La robótica móvil, posee un gran potencial para generar un impacto positivo en la sociedad, ya sea a través de mejoras en la seguridad, la eficiencia, salud laboral de algunos puestos de trabajo o incluso reducción del impacto ambiental de los sistemas de transporte.

Es una gran motivación la oportunidad de contribuir al cuerpo de conocimiento en este área, así como abrir caminos para futuras investigaciones.

1.2 Objetivos

El objetivo principal de este Trabajo Fin de Grado es desarrollar un sistema de navegación local autónoma para vehículos terrestres utilizando mapas probabilísticos 3D. Este sistema deberá permitir al vehículo moverse de manera segura y eficiente en entornos desconocidos y dinámicos, utilizando tecnologías avanzadas de mapeo y sensores.

De forma más específica, se pretende:

- La integración de ROS para la gestión del vehículo: Utilizar el Robot Operating System (ROS) para gestionar las operaciones del vehículo. ROS proporcionará las herramientas necesarias para el control del vehículo, la integración de sensores, la comunicación entre sistemas y la implementación de algoritmos de navegación.
- La creación y utilización de mapas tridimensionales: Implementar técnicas para la creación de mapas 3D del entorno utilizando sensores y algoritmos avanzados. Estos mapas deberán actualizar la información del entorno en tiempo real para permitir una navegación precisa y adaptativa.
- La implementación de un algoritmo de navegación reactiva y evitación de obstáculos: Desarrollar un sistema de navegación reactiva que permita al vehículo evitar obstáculos en tiempo real, adaptándose a cambios en el entorno y asegurando un movimiento eficiente y seguro.
- Uso de Gazebo, Husky, OctoMap y Octrees: El proyecto se desarrollará en el entorno de simulación Gazebo, utilizando el robot Husky como plataforma de prueba. Se emplearán tecnologías como OctoMap y Octrees para la gestión eficiente de los datos espaciales y la representación del entorno.
- La validación y pruebas en diferentes entornos: Realizar una serie de pruebas y validaciones en diferentes escenarios y condiciones para asegurar la robustez y precisión del sistema de navegación.

1.3 Estructura de la memoria

El contenido de este trabajo se estructura en varios apartados y secciones que permitirán una compresión clara y ordenada del contenido.

- **Capítulo 1:** Introducción.

  En este primer capítulo, se ha realizado una introducción al trabajo realizado, exponiendo los antecedentes, la justificación y la motivación que ha dado lugar a llevar a cabo esta investigación, así como los objetivos que se pretenden cumplir con este trabajo.

- **Capítulo 2:** Estado del arte.

  Se aborda la parte teórica relacionada tanto con la Navegación Local como con la Representación 3D del Entorno. Este capítulo se encuentra dividido en dos partes, en la primera se define el concepto de navegación local y se exploran diferentes métodos y algoritmos utilizados para abordarla. Se proporciona una visión detallada de las estrategias empleadas en la navegación local de vehículos robóticos. En la segunda parte se profundiza en la representación tridimensional del entorno. Se discute el uso de nubes de puntos 3D y se presta especial atención al método Octree, ampliamente empleado en este trabajo, para la representación eficiente de entornos tridimensionales.

- **Capítulo 3**: Entorno de Trabajo.

  Este capítulo se dedica a la descripción del entorno de trabajo utilizado en la investigación. Se introduce desde el Sistema Operativo Robótico (ROS) y se explica su relevancia en el proyecto, destacando su papel en la integración y control de sistemas robóticos, así como su funcionamiento. Se presentan y describen las herramientas de visualización y simulación utilizadas, incluyendo Gazebo, una plataforma de simulación 3D, y RViz, una herramienta de visualización en tiempo real. Se introduce la librería OctoMap, utilizada para la generación de mapas 3D, y se detalla el robot móvil Husky junto con la cámara RealSense implementada en el proyecto.

- **Capítulo 4:** Fundamentos teóricos.

  En este capítulo se exponen y desarrollan aquellos fundamentos teóricos que sustenta el trabajo y la creación del algoritmo. En él se explica el cálculo matemático del área del campo de visión del robot, la creación de los vectores de datos que se observarán en el mapa para la detección de obstáculos, así como se incluye el modelo de control de velocidades del robot

- **Capítulo 5:** Navegación Local empleando Octree.

  Se profundiza en la creación de mapas Octrees, su funcionamiento y su uso en conjunto con la librería OctoMap. Además, se describe de manera detallada el algoritmo utilizado para la evasión de obstáculos, incluyendo su funcionamiento y programación. Este capítulo pone en práctica las bases teóricas para la navegación segura y eficiente del robot en entornos tridimensionales.

- **Capítulo 6**: Validación Experimental.

  Se procede a la validación experimental de todo el trabajo realizado. Aquí se explica la metodología empleada en los diferentes experimentos y se presentan simulaciones que demuestran el correcto funcionamiento de los algoritmos y sistemas implementados. Este capítulo es crucial para evaluar la eficacia de la navegación y la evasión de obstáculos en entornos reales y simulados.

- **Capítulo 7**: Conclusiones y líneas de trabajo futuras

  Por último, se exponen las conclusiones obtenidas una vez finalizado el trabajo, así como se formulan posibles futuras implementaciones

Esta es una visión general de cómo se organiza el trabajo y qué temas se abordan en cada parte. Cada capítulo de este trabajo contribuye a una comprensión integral del sistema de navegación autónoma. El Capítulo 2 establece la base teórica, mientras que el Capítulo 3 describe las herramientas y tecnologías utilizadas, el Capítulo 4 explica los fundamentos teóricos, esencial para entender la implementación práctica en el Capítulo 5. Finalmente, el Capítulo 6 demuestra la aplicabilidad del sistema en escenarios del mundo real. Para finalizar y concluir el trabajo, el Capítulo 7 ofrece las conclusiones y las futuras líneas de investigación.

Capítulo 2

Estado del arte

2.1 Navegación Local

2.1.1 Definición de navegación local

La navegación local, también conocida como navegación reactiva, se trata de una estrategia de control utilizada en la robótica móvil para permitir que un robot evite obstáculos y se mueva de manera segura en su entorno inmediato. Esta técnica se centra en la toma de decisiones en tiempo de ejecución o tiempo real, basadas en la información sensorial actual. Permite al robot móvil responder de manera inmediata a los estímulos del entorno.

No depende de una representación completa y precisa del entorno, no planifica previamente la trayectoria a seguir, reacciona con el entorno con el fin de sortear las distintas barreras o impedimentos que puedan estar en su camino. El robot decide su próxima acción en función de su percepción inmediata del entorno. Este tipo de navegación es adecuada para entornos cambiantes y dinámicos. Es el nivel inferior a la planificación global, encargada de planificar el camino general desde el punto de inicio hasta el punto destino.

La navegación local se basa en algoritmos de control reactivos, estos algoritmos toman decisiones inmediatas en función de la información sensorial disponible. Es un tipo de navegación que depende en gran medida de los datos aportados por los distintos sensores en tiempo real. Sensores como cámaras, LiDar, sensores de ultrasonidos o inerciales, proporcionarán información acerca de la posición, la velocidad y la dirección de obstáculos en un entorno cercano al robot.

Una de las principales ventajas de este enfoque es su capacidad para operar en entornos impredecibles y sobre terrenos no estructurados, lo que representa un gran desafío para los sistemas de navegación más tradicionales. Además, la navegación reactiva reduce la necesidad de cómputo intensivo y mapeo detallado, lo que permite a los robots ser más ágiles y eficientes en su desplazamiento.

Sin embargo, esta técnica también enfrenta desafíos, como la limitación en la anticipación de eventos futuros y la dificultad para navegar en entornos extremadamente congestionados o caóticos. Además, la dependencia de la percepción sensorial en tiempo real significa que cualquier fallo o limitación en los sensores puede afectar significativamente el rendimiento del robot.

Históricamente, la navegación reactiva ha evolucionado desde los primeros experimentos en robótica móvil, donde el énfasis estaba en la interacción simple con el entorno, hasta sistemas más sofisticados que integran la inteligencia artificial y el aprendizaje automático para mejorar la adaptabilidad y la toma de decisiones.

Mirando hacia el futuro, se espera que la navegación local se beneficie enormemente de los avances en inteligencia artificial, permitiendo a los robots no solo reaccionar a su entorno sino también anticipar posibles cambios y adaptarse de manera proactiva. Esto podría abrir nuevas posibilidades en campos como la exploración espacial y en el desarrollo de ciudades inteligentes.

La autonomía de un robot móvil se basa en gran parte en su sistema de navegación. La implantación de un sistema de navegación local o reactiva permite que los robots móviles se adapten y eviten colisiones mientras se desplazan de manera segura y eficiente.

En cuanto a las aplicaciones de este tipo de navegación, es ampliamente utilizada en lo referente a vehículos autónomos, robots de limpieza, drones o robots de servicio. En el campo de la robótica industrial ha supuesto un gran avance en el ámbito de la logística y la distribución para almacenes automatizados, así como para tareas de inspección y mantenimiento de áreas de difícil acceso para los humanos.

2.1.2 Técnicas para evasión de obstáculos en robótica móvil

Para abordar este problema de localización existen técnicas y algoritmos muy diversos, en los que se aplican diferentes enfoques y métodos para que los robots naveguen de manera segura en entornos donde hay obstáculos. Todas estas técnicas y algoritmos tienen en común que se centran en tomar decisiones y acciones que eviten colisiones con elementos que impidan pasar o avanzar en la trayectoria del robot.

Entre las distintas técnicas y algoritmos a implementar en la navegación local caben destacar las descritas a continuación. Se ha llevado esta selección basada en las diapositivas de Dr. Ricardo Vázquez de la asignatura Ampliación de Robótica [2]:

- **Algoritmo del bicho (Bug):** Una estrategia de contorno que permite que el robot rodee el obstáculo dirigiéndose de nuevo al punto de meta desde el punto más cercano (Bug I) o volviendo a la trayectoria inicial tan pronto como la vuelva a encontrar tras rodear el obstáculo (Bug II). Aunque esta estrategia es simple y efectiva en muchos casos, puede no ser eficiente en entornos complejos donde los obstáculos son numerosos o están dispuestos de manera complicada. En la figura 1 se muestra una representación de este algoritmo. [3]

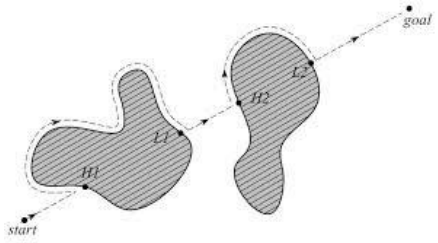

Figura 1:Algoritmo del bicho (Bug). [3]

- **Grafos de visibilidad:** Representa el entorno con nodos y aristas que conectan puntos de visibilidad directa, lo que permite planificar rutas seguras a través de áreas con obstáculos [4]. Puede suponer una gran complejidad dependiendo de la geometría del entorno. Es una técnica bastante utilizada también en la planificación global. Esta técnica es más efectiva en entornos estáticos, donde la disposición de los obstáculos no cambia, ya que cualquier cambio requiere una nueva generación del grafo. En la figura 2 se puede observar el funcionamiento de este algoritmo.

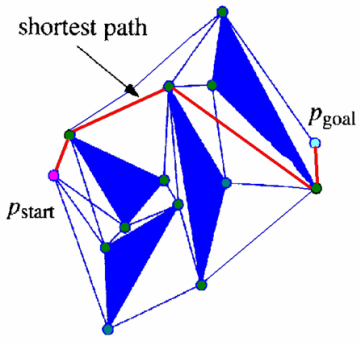

Figura 2: Grafos de visibilidad. [2]

- **Diagramas de Voronoi:** Divide el espacio en regiones basadas en la proximidad a puntos de referencia u obstáculos. Permite evitar los obstáculos manteniendo al robot lo más alejado posible de los mismos [5]. Esto supone una mala optimización del camino y puede provocar problemas o fallos con sensores de corto alcance. Aunque esta técnica es menos eficiente en términos de la longitud del camino, puede ser muy útil en entornos donde la seguridad y la distancia de los obstáculos son críticas, como en operaciones de rescate o en entornos peligrosos. La figura 3 es un ejemplo de estos Diagramas de Voronoi.

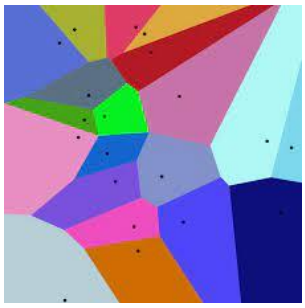

Figura 3: Diagramas de Voronoi. [23]

- **Campos potenciales:** Se aborda el robot como una entidad influenciada por un campo potencial artificial. En este escenario, el robot se desplaza siguiendo una dirección determinada por el gradiente del campo. El campo ejerce una fuerza atractiva hacia el punto de destino, al tiempo que los distintos obstáculos generan una fuerza repulsiva que alejan al robot, evitando así colisiones y guiando al robot hacia su destino de forma segura. Puede sufrir el problema de los mínimos locales, donde el robot queda atrapado en un punto que no es el destino debido a la configuración de las fuerzas repulsivas y atractivas. La figura 4 muestra un ejemplo de una simulación de navegación evitando obstáculos haciendo uso de campos potenciales.

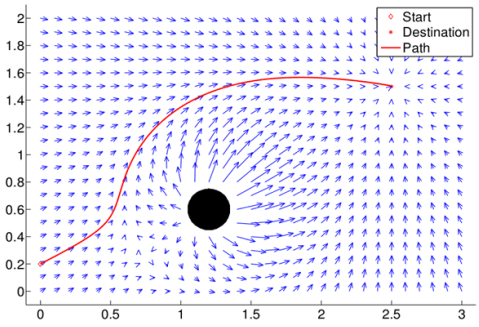

Figura 4: Campos Potenciales. [2]

- **Redes Neuronales y Aprendizaje Profundo:** Las redes neuronales artificiales, particularmente aquellas en el campo del aprendizaje profundo, son modelos computacionales inspirados en la estructura neuronal del cerebro. En la robótica móvil, estas redes pueden procesar una gran cantidad de datos sensoriales para aprender y tomar decisiones. Los robots pueden ser entrenados, mediante técnicas como el aprendizaje supervisado o reforzado, para reconocer y reaccionar ante diversos tipos de obstáculos. Por ejemplo, un robot autónomo podría utilizar una red neuronal convolucional (CNN) para procesar imágenes y determinar la mejor manera de evitar obstáculos. Esta técnica es particularmente útil en entornos dinámicos y no estructurados, donde los obstáculos y condiciones cambian constantemente. Las redes neuronales pueden adaptarse y responder en tiempo real a situaciones no previstas durante la programación inicial. El principal desafío es la necesidad de grandes conjuntos de datos para el entrenamiento y el alto costo computacional. Además, los resultados pueden ser difíciles de interpretar o predecir, ya que las decisiones de la red no siempre son transparentes.

- **Método de restricción de obstáculos (ORM):** Se basa en la identificación y clasificación de obstáculos para luego la planificar la ruta. El ORM funciona mediante la creación de un mapa del entorno que incluye la ubicación y las dimensiones de los obstáculos detectados. Luego, el sistema calcula una trayectoria que minimiza la probabilidad de colisión. Este cálculo suele realizarse teniendo en cuenta no solo la posición actual del vehículo y los obstáculos, sino también su velocidad y dirección de movimiento. Tal y como cuenta Javier Minguez en su Conferencia "Intelligen Robots and Systems" [6]. El ORM funciona identificando subobjetivos potenciales que se encuentran entre los obstáculos o en el borde de un obstáculo. Estos subobjetivos se seleccionan en función de si pueden ser alcanzados desde la ubicación actual del robot. La selección se realiza mediante un algoritmo que verifica la existencia de un camino que conecte dos ubicaciones, considerando el tamaño y la forma del robot y los obstáculos en el entorno.

- **Planificación Basada en Muestreo:** La planificación basada en muestreo involucra generar puntos o nodos aleatorios en el espacio de configuración (que representa todas las posibles posiciones y orientaciones del robot) y luego conectar estos puntos de manera que se forme un camino viable. Dentro de este método podemos hablar tanto del RRT (Rapidly-exploring Random Tree) como del PRM (Probabilistic Roadmap):

    - **RRT (Rapidly-exploring Random Tree):** Esta técnica construye un árbol extendiéndose rápidamente desde el punto de inicio y explorando el espacio de configuración de manera aleatoria. Es eficiente en encontrar un camino factible en espacios de alta dimensión y puede adaptarse a cambios en tiempo real. La figura 5 es un ejemplo gráfico de cómo funciona este algoritmo.

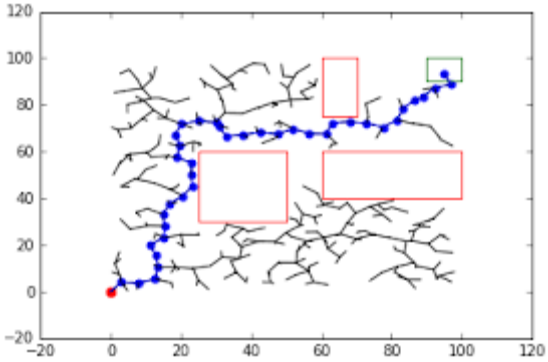

Figura 5: Algoritmo RRT. [24]

- **PRM (Probabilistic Roadmap)**: En esta técnica, se genera un mapa de nodos aleatorios conectados por posibles caminos. Luego, se utiliza un algoritmo de búsqueda para encontrar el camino más corto desde el punto de inicio hasta el destino. Es especialmente útil en entornos donde el mapeo previo es posible. La figura 6 es un ejemplo de este algoritmo.

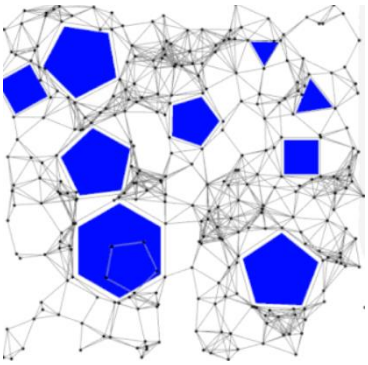
Figura 6: Algoritmo PRM. [25]

La principal ventaja es la capacidad de estas técnicas para manejar entornos de alta complejidad y dimensiones. Sin embargo, pueden ser computacionalmente intensivas y no siempre garantizan la ruta más corta o eficiente.

Para más información acerca de los métodos expuestos se puede consultar la Tesis doctoral de D.A López sobre nuevas aportaciones en algoritmos de planificación para la ejecución de maniobras en robots autónomos no holónomos [5]

En este trabajo se basará en el algoritmo RRT (Rapidly-exploring Random Tree) para crear un nuevo programa para la navegación autónoma local de un vehículo terrestre.

2.1.3 Elección del algoritmo RRT como base.

La elección de este algoritmo como base para elaborar un método de navegación local del robot móvil presente en este trabajo es una decisión estratégica basada en varias ventajas clave que supone este algoritmo. Algunas de las razones más significativas por las que se ha optado por basar el algoritmo en el algoritmo RRT son:

- Exploración Eficiente de Espacios Grandes: El RRT es excepcionalmente bueno en la exploración rápida y eficiente de espacios de escenarios grandes y desconocidos. Esto lo hace ideal para la navegación local en entornos complejos o poco estructurados, donde es crucial identificar rutas viables en poco tiempo.

- Manejo de Alta Dimensionalidad: El algoritmo RRT puede manejar espacios de configuración de alta dimensionalidad. Esto es especialmente relevante en robótica móvil, donde el robot puede necesitar tener en cuenta múltiples grados de libertad y restricciones al planificar su trayectoria. Este punto es importante ya que se va a simular en un robot con unas características físicas y de movimiento muy específicas.

- Solución de Trayectorias Complejas: El RRT es capaz de encontrar soluciones en situaciones donde otros algoritmos de planificación de trayectorias podrían quedar atrapados en mínimos locales o no encontrar una solución. Su naturaleza aleatoria ayuda a superar estos obstáculos. Esto es fundamental para evitar lo máximo posible los problemas de trampas y mínimos locales.

- Balance entre Aleatoriedad y Determinismo: El RRT utiliza tanto componentes aleatorios como deterministas, lo que ayuda a explorar el espacio de una manera más completa que los métodos puramente deterministas o aleatorios. El algoritmo cuenta con una parte aleatoria en la que se lanzan puntos al azar y una parte de cálculo en la que se elige el punto más conveniente para llegar al punto destino

- Integración con Sistemas de Sensores: El algoritmo RRT puede integrarse efectivamente con sistemas de sensores en tiempo real, permitiendo que el robot ajuste su trayectoria en respuesta a la información sensorial actualizada. En este caso ha sido implementado junto con la información y los datos captados por una cámara.

- Viabilidad en Tiempo Real: Aunque el RRT puede ser computacionalmente intensivo, sus variantes más modernas han sido optimizadas para la planificación en tiempo real, lo cual es un requisito esencial en la navegación local de robots móviles.

Se considera este algoritmo una opción robusta y versátil para basar el trabajo y obtener los objetivos deseados.

2.2 Representación 3D del entorno

2.2.1 Nubes de puntos 3D

Una nube de puntos 3D es un conjunto de puntos de datos en un espacio tridimensional. Cada punto representa una posición en el espacio físico, y juntos, estos puntos forman una representación detallada de un entorno. Es un sistema de representación de una superficie a través de un conjunto de vértices (X, Y, Z) en un sistema tridimensional [7]. Todos estos puntos contienen la información necesaria para generar un modelo virtual.

Las nubes de puntos se generan generalmente utilizando sensores como LiDAR, cámaras estéreo o RGB-D. Estos dispositivos capturan información del entorno y la convierten en un conjunto de puntos que representan superficies y objetos.

En su forma más simple, una nube de puntos solo contiene información de posición de la superficie que ha sido capturada con el sensor. En representaciones extendidas, puede incluir información de las componentes del vector normal o el color del objeto capturado.

El procesamiento de nubes de puntos implica varios pasos como la limpieza de datos (eliminación de puntos atípicos), la alineación (en caso de múltiples capturas) y, a veces, la reducción de la densidad de puntos para facilitar el manejo de los datos.
Manejar nubes de puntos puede ser computacionalmente intensivo, especialmente cuando los datos son densos o el entorno es grande. Los mapas de nubes de puntos ocupan muchos recursos de computación, cosa que los hace mapas difíciles de mantener y dificultan el obtener información útil de manera eficiente. Esto es una de las razones por las que para este proyecto se usan estructuras como los Octrees (apartado 2.2.2)

La precisión de los datos depende de la calidad del sensor y las condiciones ambientales durante la captura.

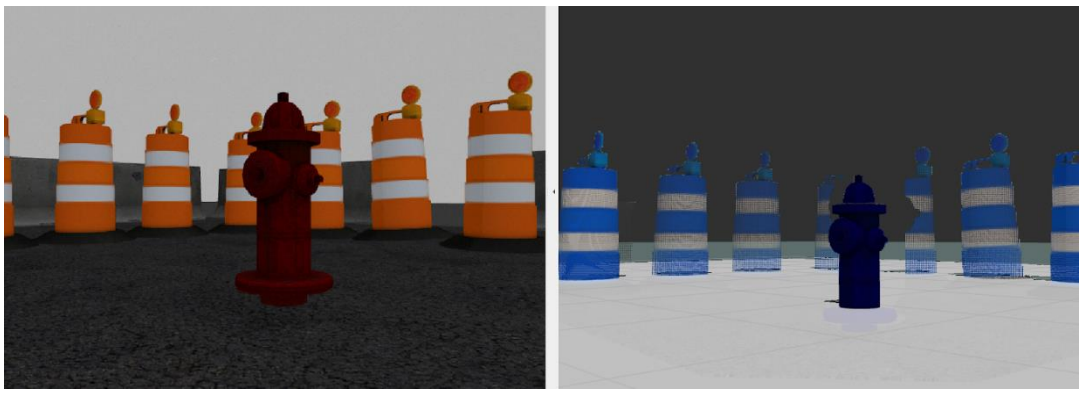

Figura 7: Comparativa imagen real con imagen generada con Nube de puntos.

La imagen presentada en la Figura 7 ilustra de manera efectiva la comparativa entre una imagen real (izquierda) y su correspondiente representación generada a partir de una nube de puntos (derecha) . Este tipo de visualización permite apreciar no solo la fidelidad de la representación tridimensional, sino también cómo las técnicas de captura y procesamiento de datos influyen en la precisión y realismo del modelo generado.

2.2.2 Octree

Un Octree se trata de una estructura de datos jerárquica que divide el espacio en octales o cubos tridimensionales o voxels. En el contexto de nubes de puntos 3D, un Octree permite organizar eficientemente los datos espaciales, facilitando operaciones como la búsqueda y el procesamiento de puntos específicos. La figura 8 muestra un árbol octal en el que se puede ver gráficamente esta subdivisión de espacios mencionada.

La denominación "octree" se compone de "oct" (octante) "tree" (árbol). Un octree o árbol octal es una estructura de datos jerárquica, en forma de árbol, donde cada nodo se divide en 8 cubos tridimensionales de menor tamaño o "hijos" [8]. Esta estructura se emplea principalmente para subdividir un espacio tridimensional de manera recursiva en ocho octantes. Inicialmente se parte de un solo octante y este se va dividiendo sucesivamente hasta que se alcanza un nivel deseado de detalle o se cumple algún criterio de división predefinido. Cada octante del árbol incluye información sobre el contenido de este segmento en el espacio.

Como se ha mencionado antes, los octantes se organizan en una estructura jerárquica similar a un árbol, donde cada octante tiene un padre y puede tener cero o más hijos. Esta jerarquía permite una representación eficiente de los datos, ya que los niveles más bajos representan los detalles y los niveles superiores las regiones generales.

Los Octrees son útiles para simplificar y comprimir la representación del entorno. Al subdividir el espacio en cubos más pequeños, permiten un enfoque más manejable y eficiente en la gestión de datos tridimensionales.
En la navegación autónoma, los Octrees ayudan a optimizar la detección de colisiones y la planificación de trayectorias, ya que permiten un cálculo más rápido de la ruta evitando áreas densas o complicadas.

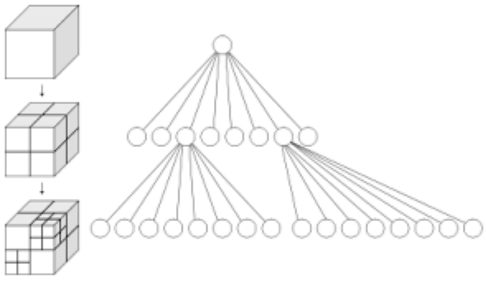

Figura 8: Estructura jerárquica de datos Octree. [8]

Políticas de ocupación.

En el ámbito de la navegación autónoma, para planear y establecer rutas seguras evitando obstáculos es necesario establecer decisiones sobre el estado de los nodos (es decir, si un espacio está ocupado o libre). Esta decisión se basa en políticas de actualización de datos. Estas políticas determinan cómo se actualizan las probabilidades en los nodos del Octree en función de las observaciones y las mediciones recopiladas por los sensores del robot. Estas son algunas de las políticas comúnmente utilizadas para decidir el estado de un nodo en OctoMap:

- Política de ocupación simple: En esta política, se actualiza un nodo del Octree a "ocupado" si se observa alguna evidencia de ocupación en ese volumen de espacio. Esto significa que, si el sensor detecta cualquier obstáculo o estructura en el espacio, el nodo se marca como ocupado con alta probabilidad. Si no se detecta nada, se marca como libre. Esta política es adecuada para sensores que proporcionan mediciones binarias (ocupado/libre).

- Modelo probabilístico de ocupación: En lugar de usar una asignación binaria de estados, se utilizan probabilidades para reflejar la incertidumbre en las observaciones. Un nodo en el Octree se actualiza utilizando un modelo probabilístico, como el modelo Log odds o el modelo Bayesiano. Este enfoque permite una representación más precisa de la incertidumbre y permite la fusión de información de múltiples fuentes de sensores.

- Fusión de información: En entornos robóticos del mundo real, se suelen utilizar múltiples sensores para obtener una visión más completa del entorno. Las políticas de actualización pueden combinar información de diferentes sensores de manera que se refleje adecuadamente la incertidumbre y la correlación entre las mediciones. La fusión de información puede involucrar modelos probabilísticos y estadísticos avanzados.

- Actualización basada en el historial: En algunas aplicaciones, es importante tener en cuenta el historial de observaciones y cambios en el entorno. Los nodos pueden actualizarse teniendo en cuenta las observaciones pasadas, lo que puede ayudar a mejorar la coherencia en el mapa 3D y a lidiar con la dinámica del entorno.

- Adaptación dinámica de resolución: En ciertas situaciones, es posible que se desee ajustar la resolución del Octree en función de la distancia o la relevancia de las mediciones. Esto permite una representación más eficiente y precisa del entorno al asignar más resolución donde sea necesario y reducirla en áreas menos importantes.

En general, las políticas de actualización en OctoMap se diseñan para reflejar con precisión la incertidumbre en las mediciones y para proporcionar una representación coherente y útil del entorno tridimensional para la planificación y la navegación de robots.

Modelado de áreas no mapeadas.

En el contexto de la navegación autónoma, un robot sólo puede calcular rutas libres de colisiones y obstáculos para aquellas áreas que han sido escaneadas y cubiertas por mediciones de sus sensores, confirmándose así que están libres.

El modelado de áreas no mapeadas es esencial en la navegación autónoma y el mapeo 3D. Estas áreas representan un riesgo potencial ya que carecen de información sobre su estado. Los robots deben planificar rutas seguras teniendo en cuenta tanto las áreas mapeadas como las no mapeadas. La gestión de la incertidumbre en áreas no mapeadas es crítica para la seguridad y el rendimiento del robot en entornos desconocidos.

Eficiencia y resolución.

El mapa desempeña un papel fundamental en cualquier sistema autónomo. ya que se utiliza tanto en la planificación como en la ejecución de acciones. Por consiguiente, es totalmente necesario que el mapa sea altamente eficiente, no solo en términos de tiempos de acceso sino también en lo que respecta al consumo de memoria. Desde una perspectiva práctica, el consumo de memoria a menudo se convierte un uno de los principales obstáculos en los sistemas de mapeo tridimensional. Por tanto, resulta crucial que el modelo de mapa sea compacto en términos de uso de memoria, lo que permitirá mapear entornos extensos, mantener el modelo en la memoria principal de un robot y facilitar su transferencia eficaz entre múltiples robots.

La resolución del mapa es un concepto altamente relacionado con la eficiencia y el tamaño de la memoria. La resolución en el contexto de Octrees y OctoMap se refiere al tamaño de los cubos o voxels en los que se divide el espacio tridimensional para almacenar información. Una resolución más alta significa que los cubos son más pequeños y, por tanto, la representación es más detallada, cosa que requiere más memoria y espacio de procesamiento. Por el contrario, una resolución más baja, significa cubos más grandes y una representación menos detallada pero más eficiente en términos de memoria.

Es necesario encontrar un equilibrio, ya que una resolución muy alta permitirá detectar obstáculos con mayor precisión, pero aumentará la complejidad computacional de los algoritmos.

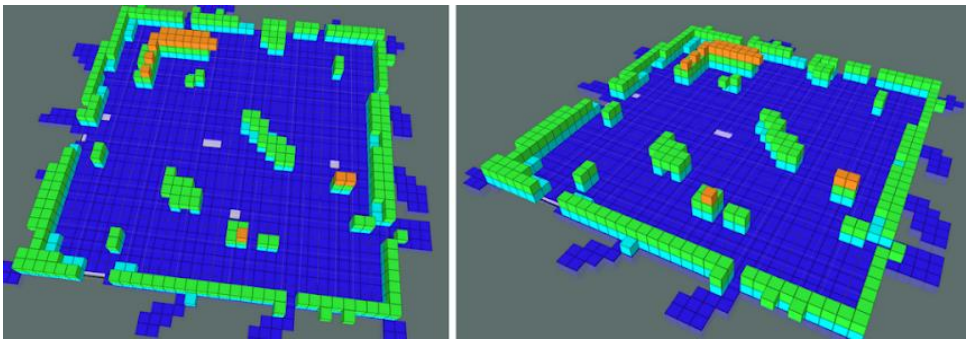

Figura 9: Creación de Mapa de Octrees con una resolución baja.

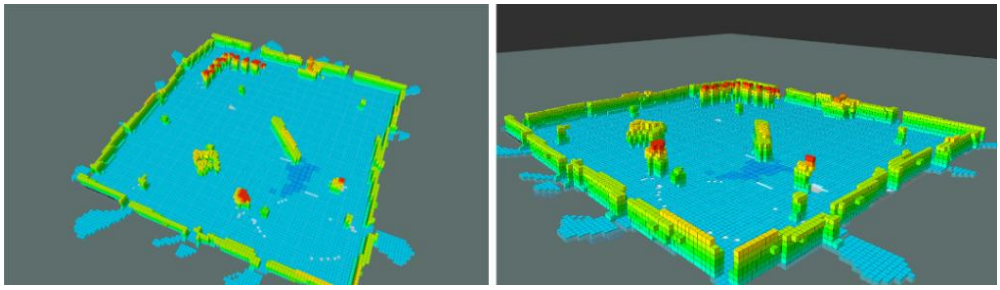

Figura 10: Creación de Mapa de Octrees con una resolución alta.

En las figuras 9 y 10 se puede observar un mismo mapa de Octrees con dos resoluciones distintas, destaca a simple vista como el mapa con una resolución más baja es más impreciso y no tan detallado.

Si se desea consultar más información sobre los Octrees y funcionamiento y aplicabilidad se puede consultar el siguiente apartado de la bibliografía [9].

Capítulo 3

Entorno de Trabajo

3.1 Sistema Operativo ROS

3.1.1 Introducción a ROS

El nombre ROS son las siglas de Ros Operating System (Sistema Operativo de Robots, en español). ROS es un conjunto de herramientas y bibliotecas de código abierto que se utiliza ampliamente en el campo y estudio de la robótica para el desarrollo y control de robots. A pesar de su nombre, ROS no se trata de un sistema operativo en sentido tradicional, es un entorno de desarrollo en robótica (RSF) que se ejecuta sobre un sistema operativo convencional, como Linux. Incluye comandos, un sistema de archivos con estructura jerárquica, subprogramas, bibliotecas y todo lo que cabe esperar de un sistema operativo sin serlo.

En la actualidad, ROS desempeña un papel fundamental como infraestructura de desarrollo, despliegue y ejecución de sistemas robóticos. Cuenta con una extensa federación de repositorios que ofrecen paquetes de software para una gran variedad de aplicaciones. Es posible encontrar más información de interés sobre ROS en su página web [10]. Para la elaboración de este Trabajo Fin de Grado la versión de ROS utilizada es ROS Noetic [11] y se usará sobre Ubuntu 20.04 [12].

3.1.2 Contexto histórico de ROS

ROS fue creado en 2007 en el Laboratorio de Inteligencia Artificial de Stanford (AI Lab) en Silicon Valley, California (EEUU). Este equipo, liderado por Andrew NG continuó con la colaboración fundamental de la empresa Willow Garage, fundada por Scott Hassan y Keenan Wyrobek. Willow Garage se dedicó a la investigación en robótica y proporcionó un entorno propicio para el desarrollo y la evolución de ROS.

Desde su inicio en 2007, ROS ha experimentado un crecimiento constante y significativo. En la actualidad y desde 2012 con periodicidad anual se realiza un congreso dedicado a ROS, congreso conocido como ROSCon, en el cual se exponen los avances más importantes de este mundo.

Hasta la fecha, existen más de 15 versiones entre ROS1 y ROS2, categorizadas cada una con un nombre cuya inicial va en orden alfabético ascendente con cada cambio de versión

3.1.3 Ecosistema (Estructura y Organización)

ROS se ha diseñado con una filosofía de máxima modularidad en todos sus aspectos, tanto para su estructura como para su organización. Esta modularidad facilita la implementación en robots así como la gestión del software en general.

Estructura

A rasgos generales, ROS es un sistema de *nodos* que comparte información en tiempo real a través de una red controlada por el *nodo Máster*. Cada nodo representa una tarea, programa o proceso específico del robot que se ejecuta de forma paralela. La información se comparte mediante *topics*, utilizando *mensajes* para la comunicación entre uno o varios nodos o mediante *servicios*. Existirán diferentes nodos que publican o se subscriben a esos topics, por otro lado, el uso de los servicios permite a un nodo pedir información a otro en un momento determinado y no de forma constante como en el proceso de subscripción.
La forma más simple de entender ROS es mediante el uso de grafos como el que aparece a continuación en la figura 11. En esta imagen se puede ver visualmente el funcionamiento y arquitectura de ROS.

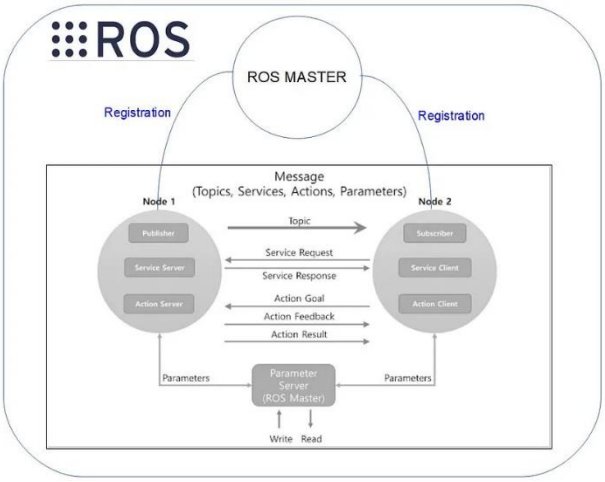

Figura 11: Arquitectura de ROS. [26]

A continuación, se explica de forma más detallada cada uno de los componentes de esta estructura

- Nodo Máster: Actúa como servidor central, permitiendo que los nodos se encuentren entre sí y puedan comunicarse. Este incluye el denominado *parameter server*, el cual se puede definir como un diccionario con nombres y valores accesible a toda la red de ROS.
- Nodos: Representan las instancias ejecutables, es decir, los procesos. Los nodos pueden combinarse y comunicarse utilizando los mecanismos previamente mencionados, los topics, services y parameter server

- Topics: Son canales de comunicación asíncrona utilizados para el intercambio de mensajes entre nodos.
- Mensajes: Encargados de transmitir la información entre los nodos a través d elos topics o services. Pueden contener una gran variedad de tipos de datos, desde numeros enteros, decimales, booleanos a estructuras de datos más complejas
- Services: Son utilizados para el intercambio de mensajes entre nodos, pero en este caso de forma síncrona, cuando un nodo lo llama o envía una solicitud

ROS se encarga de establecer y administrar la infraestructura de comunicación completa, que abarca desde la emisión de mensajes hasta la ejecución de operaciones asíncronas, incluyendo la capacidad de almacenar mensajes en un historial revisable y reproducible.

Organización

ROS trabaja sobre un espacio de trabajo o *Workspace* el cual contiene todo lo necesario para poder ejecutar los distintos paquetes. Hablamos de stack o pila a un conjunto de paquetes de una misma temática concreta.

- **Workspace:** Es el directorio principal en el que se organiza y desarrolla el proyecto. Un workspace de ROS proporciona un entorno aislado y estructurado para trabajar con múltiples paquetes de ROS. Consta de 3 directorios principales:

  '*src*', directorio en el que se almacenan todos los paquetes que componen el proyecto

  - '*build*', directorio que se utiliza para compilar los paquetes en el workspace

  - '*devel*', contiene las dependencias necesarias para que ROS pueda encontrar tus paquetes y bibliotecas durante la ejecución.

  El propósito principal del workspace es proporcionar un entorno limpio y aislado para cada proyecto, lo que facilita la gestión de dependencias y versiones, evitando conflictos entre ellas.

  Para compilar los paquetes en tu workspace se hace uso del comando '*catkin_make*' esto compila todos los paquetes en *'src/'* y genera ejecutables, bibliotecas y otros archivos necesarios en *'build/'* y *'devel/'*

- **Paquete:** Es la unidad básica de organización en ROS. Contiene rodos los archivos, recursos y código fuente relacionados con una parte específica de tu proyecto o sistema robótico. Cada paquete está diseñado para realizar una función o proporcionar una característica específica en el sistema. Un paquete de ros incluye al menos los siguientes elementos:
  - Un archivo package.xml: Este archivo contiene información sobre el paquete, como su nombre, versión, descripción y las dependencias que necesita para funcionar correctamente.
  - Un archivo CMakeLists.txt: Este archivo es utilizado por CMake, la herramienta de construcción utilizada en ROS, para configurar cómo se compila y se enlaza el código del paquete.
  - Un directorio src: Aquí es donde se coloca el código fuente específico del paquete, incluidos los nodos ROS y las bibliotecas.

- Posiblemente otros directorios como launch para archivos de lanzamiento, config para archivos de configuración, urdf para descripciones de modelos, etc.

Los paquetes de ROS se utilizan para dividir y modularizar el desarrollo del robot o sistema robótico en componentes más manejables y reutilizables. Pueden depender de otros paquetes.

3.1.4 Versatilidad

ROS es un marco de desarrollo robótico que destaca por su versatilidad y capacidad para abordar una amplia gama de aplicaciones en robótica y sistemas robóticos.

ROS ofrece una solución integral y holística al desarrollo de robots. Cuenta con una serie de características que evidencian su versatilidad:

- Soporte Multiplataforma: Es compatible con varias distribuciones de Linux y Windows, esto lo hace muy accesible.
- Arquitectura altamente Modular, permite crear paquetes independientes que luego podrán ser integrados en un sistema más grande. Esto se traduce en la implementación de la Capa de Abstracción de Hardware (HAL), la cual permite a los desarrolladores interactuar con diferentes tipos de hardware de manera consistente. Esto significa que los desarrolladores pueden escribir código que funciones con varios robots y dispositivos sin preocuparse por los detalles específicos de cada hardware subyacente. Facilita la aplicación de algoritmos (como SLAM, Algoritmos de planificación de trayectorias, Filtro de Kalman…) en cualquier tipo de sistema robótico.
- Aplicabilidad: ROS es adecuado tanto para aplicaciones de robótica móvil como de robótica industrial, se puede utilizar en robots terrestres, submarinos, aéreos o incluso brazos robóticos.
- Comunidad y ecosistema: ROS cuenta con una comunidad activa de desarrolladores y una amplia base de usuarios. Esto se traduce en una constante evolución y una gran cantidad de información y recursos

Cuando se habla de ROS se habla de un marco de desarrollo robótico altamente versátil.

3.2 Entorno de simulación y Visualización

3.2.1 Gazebo

Gazebo es un simulador de código abierto altamente popular utilizado en la robótica, especialmente en conjunción con ROS (Robot Operating System). Este simulador se destaca por su integración efectiva con ROS, proporcionando un entorno de simulación 3D avanzado que permite a los desarrolladores probar, depurar y desarrollar algoritmos y controladores de robots de manera eficiente y realista [13].

Una de las características más notables de Gazebo es su capacidad para recrear un entorno de simulación tridimensional con una física muy realista. Esto es crucial para modelar con precisión robots, sensores, y el entorno circundante. La herramienta facilita la simulación de escenarios complejos y realistas, lo que es esencial para probar el comportamiento de los robots bajo una amplia gama de condiciones, desde interacciones cotidianas hasta situaciones extremas.

Es capaz de simular la dinámica de cuerpos rígidos, lo que significa que en él se pueden modelar la física de los robots y cómo interactúan con el mundo que les rodea. Esto incluye la simulación de colisiones, gravedad, fricción e inercias. Tiene en cuenta los parámetros físicos del robot y sus restricciones holonómicas, consiguiendo así una simulación lo más cercana a la realidad posible.

Gazebo permite simular una amplia variedad de escenarios, tanto interiores como exteriores. Esto se logra a través de una representación fiel de las condiciones físicas, la textura y la iluminación de estos entornos, lo que permite a los desarrolladores obtener una comprensión más profunda de cómo los robots operarán en diferentes contextos.

Además, Gazebo no solo permite realizar una variedad de simulaciones, sino que también facilita la creación de modelos precisos de robots y sensores. Esto incluye la representación detallada de la geometría, la cinemática y las características físicas de los objetos dentro del entorno simulado. Estos modelos pueden ser importados fácilmente y utilizados directamente en ROS, simplificando significativamente el proceso de desarrollo y prueba.

Su integración con ROS es muy versátil. Se puede controlar y supervisar los robots utilizando nodos, así como recibir y enviar datos mediante topics y servicios, todo ello en tiempo real.

Gazebo constituye una herramienta esencial en el desarrollo del robot y de la navegación por el espacio. Permitirá probar de forma realista todo lo trabajado en este proyecto.

3.2.2 RViz

Se trata de una herramienta de visualización 3D poderosa y versátil que se utiliza comúnmente en ROS para visualizar datos y estado en tiempo real en entornos robóticos. RViz se utiliza para comprender la percepción del robot, verificar el funcionamiento de los algoritmos de planificación y navegación, y depurar problemas en la interacción con el entorno [14].

RViz permite combinar en una misma pantalla distintos modelos de robots y datos de distintos sensores. Genera una visualización tanto de elementos en 2D como en 3D. Una característica clave de RViz es su capacidad para mostrar en tiempo real información publicada en diversos topics de ROS. Pueden ser representados al mismo tiempo datos como la odometría, los mapas en 2 y 3 dimensiones, la vista de la cámara o los datos recogidos por un sensor láser. Estos distintos tipos de datos permiten mostrar cómo los robots perciben y navegan en su entorno.

RViz cuenta con una interfaz gráfica de usuario o GUI intuitiva que permite ir agregando y configurando todas estas visualizaciones de datos antes mencionados de manera sencilla. Se pueden personalizar las visualizaciones según las necesidades específicas. Además, cuenta con una configuración muy flexible y personalizable de los datos en cuanto a objetos, escalas y colores.

RViz es una herramienta de depuración esencial para los problemas de navegación y control de robots. A través de esta herramienta, se puede verificar la correspondencia entre la percepción sensorial y la planificación de la navegación del robot, identificar obstáculos y valuar la idoneidad de las trayectorias planificadas, todo ello, como se ha mencionado anteriormente, en tiempo real.

En resumen, RViz es una herramienta multifuncional que facilita la visualización y comprensión de datos complejos en el ámbito de la robótica, mejorando la interacción entre el usuario y los sistemas robóticos.

3.3 Octomap

OctoMap es un framework de mapeo 3D probabilístico eficiente basado en Octrees o árbol de octales. Es una librería de código abierto muy utilizada en el campo de la robótica y de la visión por computador. Se caracteriza por su capacidad para representar eficientemente el espacio tridimensional, utilizando para ello un enfoque de mapeo de cuadrícula de ocupación 3D. La librería proporciona estructuras de datos y algoritmos de mapeo escritos en C++, lo que facilita su integración en sistemas de robótica y aplicaciones de visión por computador. Se trata de una herramienta especialmente útil en entornos donde la comprensión detallada del espacio tridimensional es crucial, como en la navegación autónoma, la manipulación robótica y la planificación de trayectorias en entornos complejos [15].

OctoMap jugará un papel fundamental en el desarrollo de este proyecto, se usará para la generación de un mapa en tres dimensiones que permitirá al robot móvil detectar objetos y poder posteriormente corregir su posición. La precisión de OctoMap en la representación del entorno permite que los robots móviles tomen decisiones de navegación informadas y seguras, incluso en entornos desconocidos o dinámicos.

La capacidad de OctoMap para manejar datos de gran volumen y alta resolución lo hace idóneo para aplicaciones en las que la percepción detallada y la precisión son fundamentales. Su naturaleza probabilística ayuda en la gestión de la incertidumbre y en la estimación de la ocupación del espacio, lo cual es esencial para evitar colisiones y para la planificación de rutas seguras.

Se trata de una herramienta muy flexible y escalable. Dado que OctoMap puede manejar diferentes resoluciones de mapeo, se adapta bien a una variedad de escenarios de uso, desde espacios interiores pequeños hasta entornos exteriores más extensos. La estructura de Octree facilita la escalabilidad del mapeo, permitiendo representar grandes áreas sin comprometer excesivamente los recursos computacionales.

OctoMap se integra eficientemente con una variedad de sensores, incluyendo LiDAR, cámaras estéreo y otros sistemas de percepción visual. La compatibilidad con el Robot Operating System (ROS) facilita su uso en una amplia gama de plataformas robóticas y sistemas de simulación.

A pesar de su eficiencia, el manejo de grandes volúmenes de datos en OctoMap puede presentar desafíos computacionales. La optimización de parámetros como la resolución del Octree y la gestión de datos en tiempo real son aspectos clave. La actualización continua del mapa y la gestión de la información dinámica requieren un equilibrio entre precisión y rendimiento.

3.4 Robot móviles y cámara

3.4.1 Husky

Husky es una plataforma de desarrollo robótico, se trata de UGV ("Unmanned Ground Vehicle" o en español, "Vehículo Terrestre no Tripulado"). Fue desarrollado por la empresa Clearpath Robotics [16].

Destaca por su gran versatilidad gracias a su considerable capacidad de carga útil, lo que le permite ser personalizado para satisfacer una amplia gama de necesidades, tanto en el ámbito industrial como en la investigación y desarrollo en robótica. El robot husky se integra de manera impecable con la plataforma de desarrollo ROS, lo que potencia aún más su versatilidad y utilidad en diversos contextos y aplicaciones. Su sólida construcción hace que cuente con una plataforma robusta y resistente, así como su tren de transmisión de alto par le permite moverse en una gran variedad de terrenos, desde pavimentos y entornos de interior hasta terrenos irregulares y de exterior. Se puede apreciar cómo es físicamente este robot en la figura 12 a continuación

Figura 12: Robot Husky. [16]

Historia

Como se ha mencionado antes, Husky fue creado por Clearpath Robotics, una empresa con sede en Kitchene, Ontario, Canadá, que se especializa en el desarrollo de soluciones de robótica y automatización.

Clearpcth Robotic fue fundada en 2009 por un grupo de estudiantes de la Universidad de Waterloo en Canadá. Se fundó como una nueva empresa enfocada en la fabricación de robots móviles para aplicaciones de investigación y desarrollo en robótica.
La primera versión del robot Husky se lanzó en 2011 y, desde entonces, ha experimentado múltiples iteraciones y mejoras. Fue creado para abordar la creciente demanda de una plataforma robótica móvil versátil que pudiera ser utilizada en gran variedad de aplicaciones.

Características

El robot Husky, se trata de una plataforma de desarrollo robótico de tamaño mediano. Cuenta con unas dimensiones de aproximadamente 1 metro de largo, 0'7 metros de ancho y 0'4 de alto. Su plataforma superior, sobre la que es posible colocar e instalar los distintos dispositivos es de 1200 cm^2. En la figura 13 se puede observar más detalladamente todas estas medidas

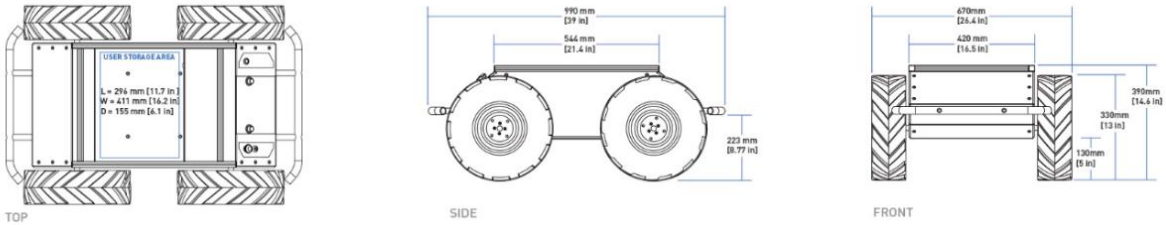

Figura 13: Medidas y Planos del robot Husky. [16]

Cuenta con 4 ruedas de unos 3 metros de diámetro cada una, ruedas que por su composición y forma le permite atravesar prácticamente todo tipo de terreno. Cuenta con Drivetrain 4x4 o, en otras palabras, cuenta con tracción en las cuatro ruedas.

El peso del Husky es de 50 kilogramos y la carga útil máxima soportada por este es de 75 kilogramos.

Su sistema de transmisión y propulsión ha sido diseñado con previsión para proporcionar una velocidad máxima de 1m/s (3'6km/h). Sus características físicas le permiten subir pendientes de hasta 45º (pendientes del 100%) y soportar inclinaciones laterales de hasta 30º (pendientes del 58%).

En lo referente a sus baterías y sistemas de potencia, utiliza baterías de ácido de plomo sellado, con una capacidad de 24 voltios y 20 Amperios hora. Ofrece una autonomía de 8 horas en modo espera y de 3 horas en uso nominal, con un tiempo de recarga de 4 horas. Los usuarios tienen acceso a potencia de 192 watios con posibilidad de ser ampliable hasta 480 watios.

Husky ofrece diversos modos de control, incluyendo el control directo de voltaje, el control sobre la velocidad de las ruedas o sobre la velocidad cinemática. Proporciona retroalimentación sobre la tensión de la batería, las corrientes del motor, la odometría de las ruedas y las salidas del sistema de control.

La comunicación se realiza vía RS232 a 115200 baudios, y utiliza encoders de cuadratura con una resolución de 78000 pulsos por metro.

Para consultar más especificaciones se puede consultar la ficha técnica en el apartado de la bibliografía [16].

Versatilidad

La versatilidad y las múltiples aplicaciones de este robot móvil, combinadas con su capacidad de integración con ROS lo convierte en una elección sólida para la realización de este proyecto basado en la navegación terrestre autónoma y la creación de mapas mediante OctoMap.

La capacidad de carga útil configurable del Husky permite ser equipado con los sensores y cámaras necesarios para el proyecto. Además, gracias a su diseño que le permite desplazarse en una diversidad de entornos, tanto interiores como exteriores, junto con su estructura robusta, ofrece la versatilidad necesaria para llevar a cabo diversas pruebas en

distintos contextos y condiciones sin restricciones específicas, lo que favorece la realización de situaciones más realistas, un aspecto crucial en la navegación autónoma.

La inclusión de sensores integrados para los sistemas de odometría en el Husky simplifica y facilita la localización del propio robot, un elemento de gran relevancia en la navegación autónoma. Además, cuenta con una capacidad de cómputo suficiente para realizar las tareas de procesamiento de datos en tiempo real, esto resulta esencial para la ejecución de tareas de navegación local o reactiva, así como en la generación de mapas.

3.4.2 Cámara RealSense de Intel®

La cámara RealSense es una tecnología desarrollada por Intel que combina hardware y software para proporcionar capacidades de percepción 3D a dispositivos electrónicos. Estas cámaras están diseñadas no solo para capturar imágenes, sino para entender y procesar la profundidad del espacio que las rodea. Para este trabajo la cámara utilizada será la Intel® RealSense™ Depth Camera D415.

Esta cámara cuenta con tecnología de detección de imágenes RGB-D (Rojo, Verde, Azul - Profundidad), que combina imágenes en color con datos de profundidad. Esto permite no solo capturar la apariencia (color y textura) de los objetos, sino también su distancia y forma tridimensional. El modelo D415 cuenta con dos sensores de profundidad y un sensor RGB.

La serie D400 de Intel RealSense usa la visión estereoscópica para calcular la profundidad, esto permite medir la profundidad al utilizar el principio de la estereopsis, proceso por el cual el cerebro humano percibe la profundidad al comparar las imágenes capturadas desde dos puntos de vista ligeramente diferentes.

Una vez que se capturan las imágenes desde los diferentes sensores, el software de la cámara RealSense compara las imágenes para encontrar puntos de correspondencia entre ellas. Esto implica buscar características comunes (esquinas, bordes u otros detalles visuales) que sean identificables en ambas imágenes.

Con los puntos de correspondencia identificados, la cámara RealSense utiliza la información de la diferencia de ángulo entre las imágenes para calcular la distancia entre la cámara y los objetos en el campo de visión. Este proceso se conoce como triangulación y se basa en principios geométricos para determinar la profundidad.

A parte de la triangulación, la cámara crea un mapa de profundidad que representa la distancia de los objetos en la escena con respecto a la cámara. Este mapa de profundidad se utiliza para dar una representación visual de la escena en 3D, con los objetos más cercanos apareciendo más claros y los objetos más distantes más oscuros.

Este tipo de cámaras ha supuesto un gran avance en el mundo de los robots y la visión computarizada. Son cámaras ligeras, de bajo consumo y fáciles de implementar que aportan gran versatilidad a este proyecto y facilitarán la creación de mapas tridimensionales. Se puede ver cómo son físicamente este tipo de cámaras en la figura 14.

El módelo Intel RealSense Depth Camera D415 cuenta con un campo de visión horizontal de aproximadamente 64º y vertical de 43º. El rango ideal de visión se encuentra

entre los 0,5m y los 3m. Esta cámara, para este trabajo, se ha configurado para proporcionar imágenes de 640 x 480 pixeles, pero según las características del fabricante la resolución se puede aumentar hasta los 1920 x 1080 pixeles.

Se distingue por su tasa de actualización de 30Hz, lo que implica que tiene la capacidad de capturar y procesar hasta 30 imágenes por segundo. Esta característica es especialmente valiosa para aplicaciones que operan en tiempo real como es el caso actual. Se pueden consultar más especificaciones técnicas en la página web de Intel [17] así como en su hoja de especificaciones [18].

Figura 14: Cámara RealSense de Intel. [17]

Capítulo 4

Fundamentos Teóricos

Este apartado pretende presentar y desarrollar todos los fundamentos teóricos que sustentan el algoritmo creado para la navegación local del vehículo. Se explicarán aquellos conceptos claves y modelos matemáticos usados en los cálculos que se realizan durante la simulación.

4.1 Muestreo de puntos aleatorios

4.1.1 Distribución uniforme discreta

Para el muestreo de puntos aleatorios que se convertirán en los posibles puntos de estudio para determinar el movimiento en nuestro algoritmo se ha seguido una distribución uniforme discreta.

En teoría de probabilidad y estadística, la distribución uniforme discreta es una distribución de probabilidad discreta simétrica que surge en espacios de probabilidad equiprobables, es decir, en situaciones donde de *n* resultados diferentes, todos tienen la misma probabilidad de ocurrir [19]. Es una distribución muy sencilla que asigna probabilidades iguales a un conjunto finito de puntos del espacio (figura 15).

Si se denomina X a una variable aleatoria discreta dentro del conjunto

$$\{x_1, x_2, \dots, x_n\}$$

y tiene una distribución uniforma discreta entonces se escribirá como

$$X \sim Uniforme\{x_1, x_2, \dots, x_n\}$$

y su función de probabilidad será

$$P[X = x] = \frac{1}{n} \qquad para\ x = x_1, x_2, \dots, x_n$$

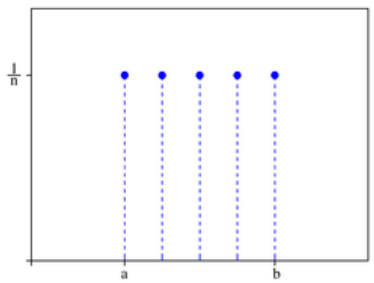

Figura 15: Distribución uniforme discreta. [19]

4.1.2 Área sujeta a muestreo

En la generación de estos puntos aleatorios es necesario establecer un área o rango sobre el cuál generar estos puntos. Este área será un espacio que rodee al robot y esté acorde con sus características físicas.

Esta área se ha definido en forma de donut o toroide plano. Existe un radio exterior y un radio interior que definirán esta área. Estos valores deberán entrar en concordancia tanto con la geometría del robot como con el alcance de la cámara de visión de obstáculos. En este caso, para el radio interior se ha elegido el valor 0.7m, dado que el robot mide aproximadamente 1m. Para el radio exterior se ha asignado el valor de 3m, ya que, según las especificaciones del elemento de visión, el valor óptimo del rango de profundidad se encuentra entre los 0.5m y lo 5m

Por otro lado, en la definición de este área de muestreo es necesario tener en cuenta el ángulo del campo de visión (FOV) de la cámara. Con el valor del ángulo de visión horizontal de la cámara se genera una superficie que corresponderá con un segmento del área con forma de toroide o donut mencionada antes. En este caso, se debe tener en cuenta que el ángulo de visión de la cámara utilizada en el proyecto es de unos 60º.

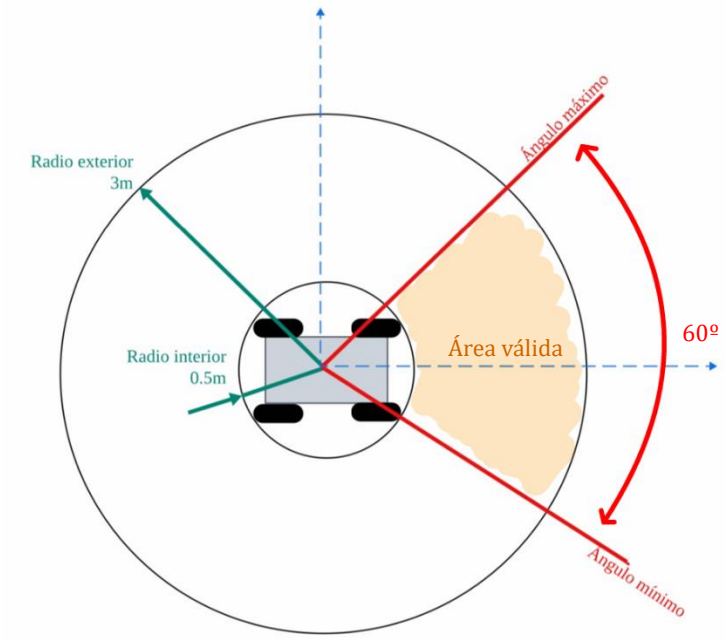

Figura 16: Representación del cálculo de área válida para la generación de puntos aleatorios.

Para poder calcular matemáticamente un punto dentro de esta área se necesitará hacer uso de algunas herramientas matemáticas como es la trigonometría.

En primer lugar, se asignarán los valores de ángulo máximo y ángulo mínimo, sumando y restando al valor de la orientación del robot en ese momento la mitad del valor en ángulos del campo de visión de la cámara, en este caso 30º o π/3. Se puede visualizar gráficamente en la figura 16.

$$\text{min\_theta} = \text{orientacion} - (\pi/3)$$

$$\text{max \_theta} = \text{orientacion} + (\pi/3)$$

Posteriormente se calcularán dos valores que corresponderán al ángulo y a la distancia de ese punto. Aquí es donde se introducirá la distribución aleatoria.

$$\text{theta} = \min\ \_\text{theta} + (\text{numero\_aleatorio}) * (\max\ \_\text{theta} - \min\ \_\text{theta})$$
$$\text{distance} = \text{sqrt}((\text{numero\_aleatorio}) * (\text{radio\_ext} - \text{radio\_int})) + \text{radio\_int}$$

En el caso del ángulo se multiplica un numero aleatorio entre la diferencia de ángulos para que se encuentre dentro de un valor de ángulo permitido y se suma el valor mínimo para asegurarnos que siempre se encuentra dentro de los márgenes.

Para el cálculo de la distancia se multiplica la raíz cuadrada del número aleatorio por la diferencia de radios (el ancho de la circunferencia) y se le suma el valor del radio interior para asegurar que el valor se encuentra dentro de los márgenes.

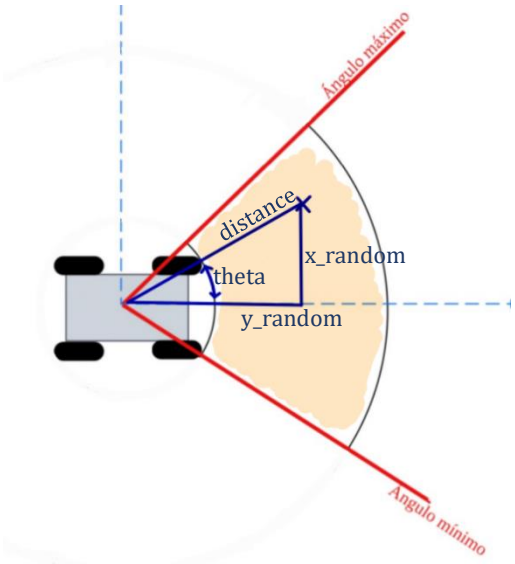

Figura 17: Cálculo del punto aleatorio dentro del área permitida.

Tal y como se observa en la figura 17, una vez obtenidos un valor de distancia y ángulo aleatorios dentro del área permitida basta con aplicar las fórmulas básicas de trigonometría para obtener el punto deseado

$$x\_random = posicion\_actual(x) + distance * \cos(theta)$$

$$y\_random = posicion\_actual(y) + distance * \sin(theta)$$

Tras este procedimiento, se obtienen las coordenadas (x,y) de un punto dentro del campo de visión del robot.

4.2 Criterios de validez de un punto

De todos los puntos aleatorios creados en el área descrita en el apartado anterior, no todos serán posible puntos a los que el robot pueda acceder de forma directa. Algunos de ellos se pueden encontrar ocupados, que hay un obstáculo intermedio o que no hay suficiente espacio para que el robot pueda posicionarse sobre este.

Para determinar cualquiera de los puntos generados aleatoriamente como válido, es necesario que cumpla con unos requisitos. En este apartado se describirán los modelos matemáticos usados para la verificación de un punto como posible o no.

4.2.1 Generación de estructuras datos para la verificación de un punto.

De forma simple, tomando el robot como un objeto puntual bastaría con solo identificar si ese punto se encuentra o no ocupado por un objeto del mapa y si la línea recta que los une también se encuentra sin obstáculos. Sin embargo, con el fin de proporcionar un modelo mucho más cercano a la realidad y que permita unas simulaciones más fiables es necesario crear vectores de puntos que representen ese punto y esa recta a verificar.

Circunferencia:

Se ha creado un vector de puntos que forman un círculo de un radio determinado alrededor del punto aleatorio seleccionado, con el fin de no solo verificar que ese se encuentre sin obstáculo, sino que los puntos de su alrededor inmediato sobre los que se posicionará el robot también lo estén.

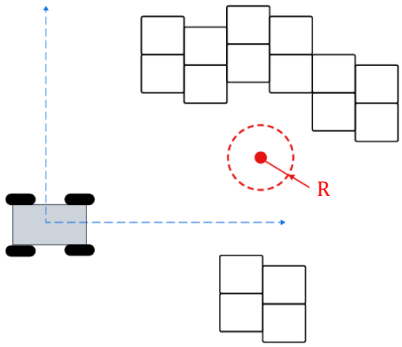

Figura 18: Circunferencia de puntos alrededor del punto a evaluar.

Para calcular estos puntos que conforman la circunferencia únicamente es necesario calcular el ángulo en función del número de puntos que se desea que tenga la circunferencia, dividiendo 2π entre ese número de puntos

$$angulo\_calculo = 2\pi * iteracion \,/\, numero\_puntos\_deseados$$

A medida que se van realizando iteraciones el valor de este ángulo va aumentando a partes iguales hasta completar la circunferencia.

Cuando se ha calculado el valor de este ángulo, se aplican razones trigonométricas para calcular los distintos puntos.

$$punto\_circunferencia(x) = posicio\_actual(x) + R * \cos\,(angulo\_calculo)$$
$$punto\_circunferencia(y) = posicio\_actual(y) + R * \text{sen}\,(angulo\_calculo)$$

Rectas que conforman un pasillo:

Dado que el robot tiene unas dimensiones específicas y no se trata de un objeto puntual, no basta con verificar la línea recta que une la posición actual del robot con el punto destino. Será necesario crear un pasillo con un ancho simular al del robot para verificar que, efectivamente, no se va a encontrar ningún obstáculo con el que pueda colisionar

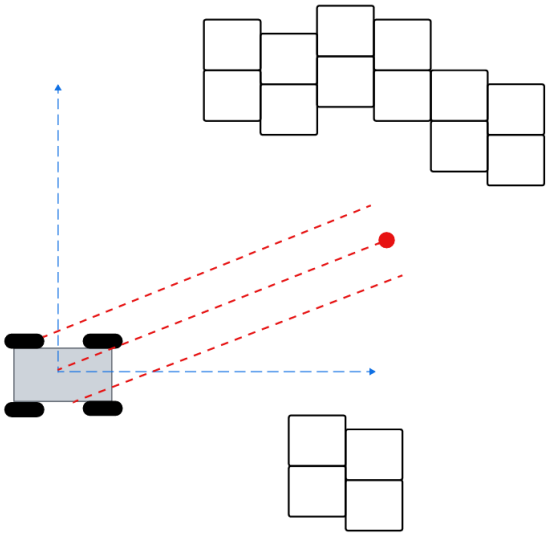

Figura 19: Rectas de puntos que conectan el robot con el punto a evaluar

Para obtener el valor de los puntos que forman estas rectas, en primer lugar, será necesario dividir la distancia de estas rectas entre la distancia que se desea que haya entre punto y punto de la recta, para así obtener el valor incremental con el que se obtendrá cada punto.

$$distancia\_calculo = distancia\_total * iteracion \,/\, incremento\_distancia$$

Finalmente, mediante una sencilla ecuación que le va sumando a la posición actual del robot los distintos valores incrementales de distancia que se van calculando, obtenemos la primera recta central. Para el cálculo de las rectas paralelas laterales bastará con trasladar en el eje x el valor del ancho que se desee.

Recta central:

$$ptoRecta(x) = ptoInicio(x) + distancia\_calculo * (ptoFinal(x) - ptoInicio(x))$$

$$ptoRecta(y) = ptoInicio(y) + distancia\_calculo * (ptoFinal(y) - ptoInicio(y))$$

Rectas laterales:

$$ptoRecta(x) = ptoInicio(x) + distancia\_calculo * \big(ptoFinal(x) - ptoInicio(x)\big) \pm 0.5$$

$$ptoRecta(y) = ptoInicio(y) + distancia\_calculo * (ptoFinal(y) - ptoInicio(y))$$

4.3 Selección del punto más próximo al punto final

Una vez realizado el proceso de creación de puntos aleatorios dentro del área establecida y de haber aplicado los criterios para considerar un punto como válido se habrá obtenido un conjunto de puntos dispuestos de forma aleatoria, a los que el robot podrá moverse.

Para la selección del punto más idóneo para alcanzar el destino final entre todos estos puntos posibles calculados se hace según el criterio de distancia. Aquel punto de entre los posibles cuya distancia al punto final sea la menor, será el punto destino intermedio al que se moverá el robot.

En la figura 20 se puede ver un ejemplo de un muestreo de puntos entra los que ya se han elegido los puntos válidos y los puntos que se descartan y posteriormente se ha calculado la distancia al punto final.

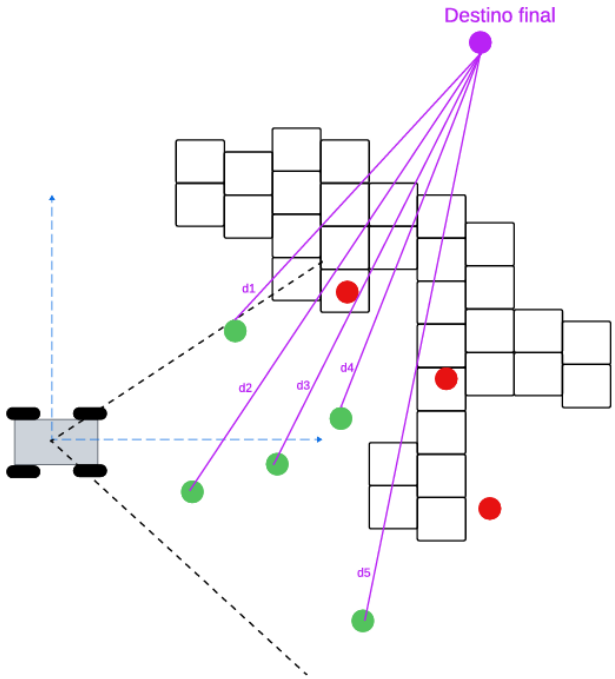

Figura 20: Ejemplo cálculo de distancias al punto final desde los posibles puntos aleatorios.

Para obtener estas distancias $(d_1, d_2, \dots, d_n)$ que aparecen en la figura 20, se ha hecho uso de la fórmula de la distancia euclídea entre dos puntos, aplicada a cada uno de ellos. La distancia euclídea entre dos puntos $P_1 = (x_1, y_1)$ y $P_2 = (x_2, y_2)$ se define como:

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

Una vez realizado este cálculo para todos los posibles puntos se obtendrá un vector que contenga el valor de todas estas distancias. Para elegir el punto de destino intermedio, se compararán todos estos valores y se elegirá el punto cuyo valor de la distancia sea el mínimo.

4.4 Cálculos realizados para comandar la velocidad

Una vez seleccionado el punto de destino intermedio al que se desea ir, es necesario comandar la velocidad para que el robot efectúe el movimiento. El movimiento se efectúa en dos partes: primero el comando de giro y en segundo lugar el comando de avance.

El comando de giro sirve para posicionar el robot orientado hacia el punto que se desea alcanzar, y el comando de avance hará que el robot se desplace en línea recta hacia ese punto.

Para ambos momentos del movimiento se ha escogido un control proporcional para comandar las velocidades angular y lineal. En el siguiente diagrama, figura 21, se muestra el control proporcional en bucle cerrado que realizará el robot en su movimiento.

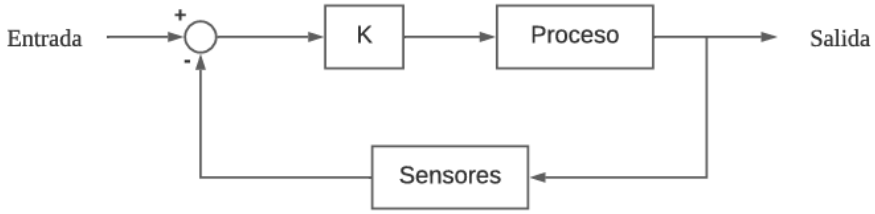

Figura 21: Diagrama de control proporcional en bucle cerrado.

Comando de giro

Para determinar el giro del robot es necesario conocer tanto la orientación actual a la que se encuentra el robot como la orientación a la que se desea llegar.

Mediante la diferencia de estos dos valores de orientación se obtendrá el ángulo que debe girar el robot para posicionarse en la dirección del punto destino. Esto se puede ver de forma gráfica en la figura 21.

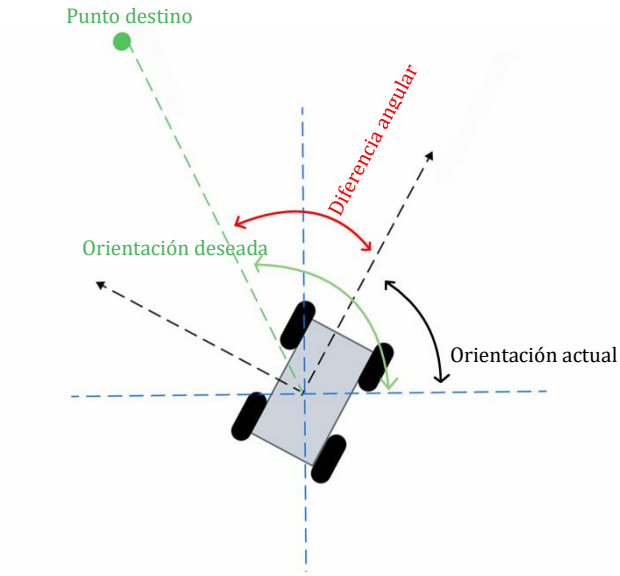

Figura 22: Cálculo de la orientación para comandar la velocidad de giro.

Conforme el robot va efectuando el giro se va calculando el valor de la diferencia angular. Este será el valor proporcional que se introducirá en el control para el movimiento

del giro en z. Este valor de diferencia angular nos proporciona el sentido de giro, ya que, si es positiva, el robot girará en sentido antihorario, y si esta diferencia angular es negativa girará en sentido horario.

El control de la velocidad angular, como se ha mencionado, es un control proporcional a la diferencia angular, para que gire más rápido cuanto mayor sea esta diferencia y reduzca su velocidad de giro cuando este valor sea menor, es decir, esté más próximo a la orientación deseada.

$$Velocidad.angular.z = 0.5 * diferencia\_angular$$

El valor 0.5 se ha escogido arbitrariamente, se cambiará en la fase de experimentación por el valor con el que mejor resultados se obtengan.

Comando de avance

Una vez el robot está completamente orientado es necesario realizar el movimiento de avance hacia el punto destino. Para conocer el comando de velocidad con el que se debe comandar el robot se hará uso de la distancia del robot hasta el punto que se desea alcanzar.

Con el cálculo de este valor se sabrá qué distancia debe avanzar el robot hasta alcanzar su destino. La distancia al punto de destino se calcula como la distancia euclídea entre la posición actual y la posición a la que se desea llegar, esta distancia euclídea se deduce a partir del teorema de Pitágoras. La figura 22 muestra un ejemplo de este cálculo.

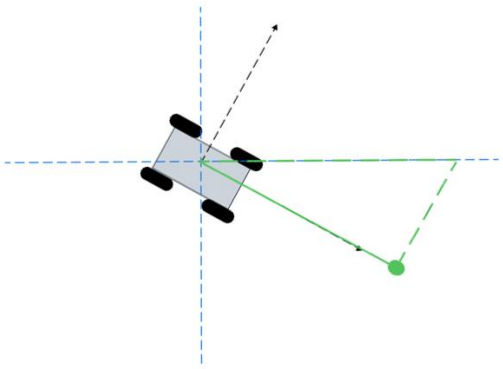

Figura 23: Cálculo de la distancia para comandar la velocidad de avance.

Conforme el robot va avanzando, se va calculando el valor de la distancia. Este será el valor proporcional que se introducirá en el control para el movimiento lineal en el eje x. Este valor será siempre positivo puesto que previamente el robot se ha orientado hacia el punto destino y por tanto, el movimiento debe ser siempre hacia delante.

El control de la velocidad lineal, como se ha mencionado, es un control proporcional a la distancia, para que el robot avance más rápido cuanto mayor sea la distancia y reduzca su velocidad cuando este valor sea menor, es decir, esté más próximo al punto deseado.

$$Velocidad.linear.x = 0.7 * distancia$$

El valor 0.7 se ha escogido arbitrariamente, se cambiará en la fase de experimentación por el valor con el que mejor resultados se obtengan.

Capítulo 5

Navegación Local empleando Octrees

5.1 Presentación del entorno de simulación

Para la realización de las pruebas y simulaciones detalladas en este documento, es crucial seleccionar un entorno de simulación adecuado. Como se mencionó previamente en la sección 4 del capítulo 3, se ha elegido el robot Husky y la cámara RealSense para este proyecto. Para integrar estos elementos, se ha optado por un entorno de simulación en Gazebo, descargado de un repositorio específico en GitHub.

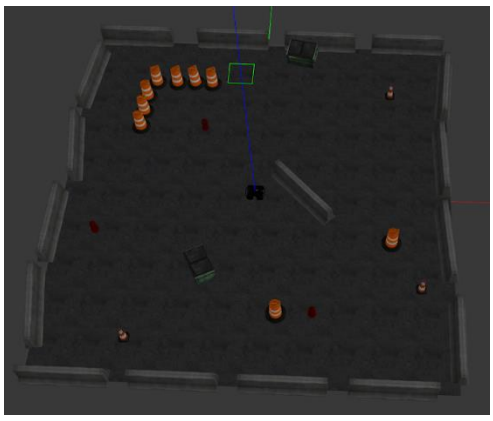

Figura 24: Escenario de Gazebo para la simulación.

Este entorno se ha adquirido del repositorio de GitHub llamado "Husky" [20], mantenido por Tinker Twins, Este nombre es la marca personal bajo la cual se conocen a los hermanos gemelos Chinmay y Tanmay Samak, conocidos por su especial interés en el campo de la robótica y los sistemas autónomos, especializándose en el campo de los vehículos. Este dúo ha participado en varios proyectos importantes, abarcando desde sistemas microelectromecánicos hasta asentamientos en órbita espacial. Su trayectoria incluye una amplia gama de herramientas y tecnologías, fortaleciendo su base científica y habilidades técnicas [21]. La figura 24 es una imagen captada del escenario de Gazebo proporcionado por el repositorio y sobre el que se ejecutarán las distintas simulaciones

El repositorio "Husky" es una colección de paquetes de ROS diseñados específicamente para el robot Husky. Incluye archivos esenciales para ejecutar y simular diversos problemas de localización, navegación y teleoperación, centrados en aspectos fundamentales de la robótica.

Dentro de las capacidades de este entorno de simulación se incluyen:

- **Keyboard Teleoperation:** Permite el control del robot mediante un teclado.
- **Map-Less Navigation:** Navegación sin necesidad de un mapa preestablecido, donde el robot emplea sus sensores para moverse y evitar obstáculos en tiempo real.
- **Simultaneous Localization and Mapping (SLAM):** El robot genera un mapa del entorno mientras se localiza dentro de este, crucial para operar en entornos desconocidos.
- **Adaptive Monte Carlo Localization (AMCL):** Método de localización en mapas conocidos, utilizando un filtro de partículas para estimar la posición y orientación del robot.
- **Map-Based Navigation:** Navegación empleando un mapa predefinido, lo que permite al robot planificar rutas y navegar eficientemente.

Instalando y configurando este repositorio, se obtienen todos los paquetes y nodos necesarios para iniciar y ejecutar cada uno de estos métodos y funciones.

El entorno físico simulado consiste en un área cuadrada de 10 metros por lado, delimitada por barreras dispuestas de manera discontinua. Dentro de este espacio, se distribuyen varios obstáculos que facilitan la demostración del funcionamiento de los distintos nodos que componen el repositorio. Algunos de estos objetos u obstáculos son contenedores, barreras y distintos tipos de conos y balizas. En la figura 25 se puede apreciar el entorno de simulación con su correspondiente rejilla y la disposición de los obstáculos en el espacio.

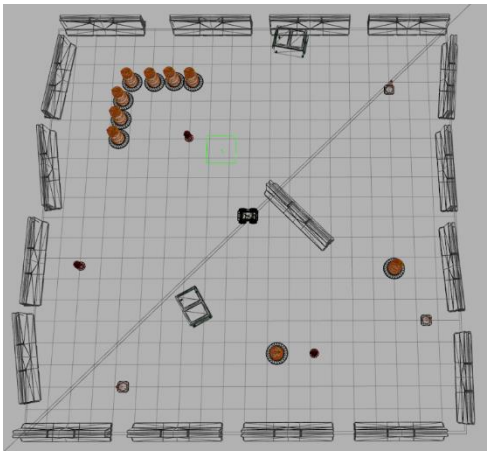

Figura 25: Disposición de obstáculos en el entorno de simulación en Gazebo.

La disposición de algunos elementos de este escenario serán modificados en diferentes etapas del trabajo para verificar la eficacia del algoritmo implementado. Gracias a la versatilidad del entorno de simulación Gazebo, es factible implementar y recrear una gran variedad de escenarios de simulación diferentes.

5.2 Modificaciones en el entorno de simulación

Para lograr una adaptación óptima del entorno de simulación proporcionado por el repositorio “Husky”, mencionado y descrito anteriormente en la primera sección de este capítulo, se han implementado algunas modificaciones y mejoras. Estas modificaciones se han llevado a cabo con el objetivo específico de ajustar el entorno a las necesidades y requisitos particulares del proyecto.

Estas modificaciones no han implicado alteraciones radicales o disruptivas en la estructura o funcionalidad fundamental del entorno original. Se han enfocado en realizar mejoras incrementales y ajustes. Todos estos cambios se han diseñado cuidadosamente para facilitar y potenciar el desempeño del algoritmo creado para la evasión de obstáculos, asegurando su funcionamiento eficaz dentro del marco de trabajo establecido. A continuación, se detallarán estos cambios.

Cambio en el tamaño del área de colisión del robot Husky

En primer lugar, durante los primeros ensayos y pruebas se observó que el área de colisión del robot era mucho mayor al área del robot en sí, es decir, la zona definida para detectar colisiones y obstáculos cercanos excedía significativamente las dimensiones físicas reales del robot. En la figura 26 se puede apreciar el tamaño del robot frente al tamaño del área definida para detectar las colisiones.

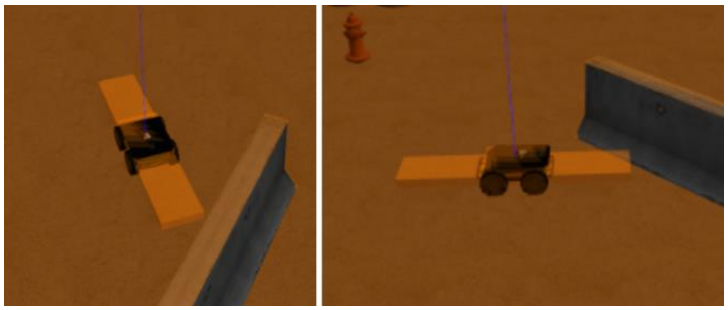

Figura 26: Área de colisión del robot Husky antes de la modificación.

Esta discrepancia en la configuración del entorno provocaba respuestas erróneas del sistema como es la detección de colisiones inexistentes en determinados momentos.

Para abordar este problema, se decidió ajustar el área de detección de colisiones de forma que fuese lo más cercano posible y representara con precisión el tamaño real del robot. En la figura 27 se puede apreciar el significativo cambio del tamaño de esta área mencionado frente al anterior (representado en la figura 26).

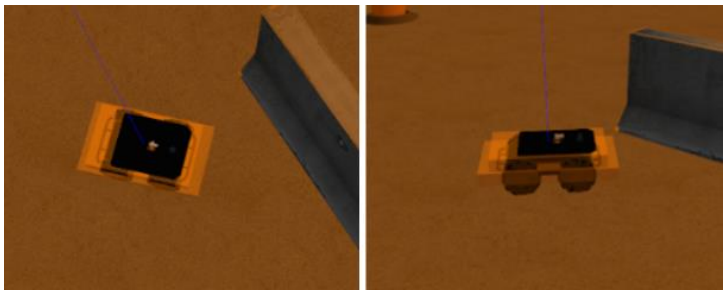

Figura 27: Área de colisión del robot Husky después de la modificación.

Este cambio no solo mejoró la precisión de la simulación, sino que también contribuyó a un comportamiento más realista del robot al navegar en su entorno. Con una representación más ajustada y precisa del tamaño físico del robot es posible que se realicen maniobras de movimiento más ajustadas y eficientes, optimizando así su capacidad para moverse en el entorno

El archivo .urdf modificado que contiene la descripción del entrono con estas modificaciones se puede encontrar completo en el Anexo I apartado 10 . Este archivo es fundamental para la definición del entorno de simulación, ya que representa como es el modelo del robot y del entorno.

Cambio en la posición de la cámara RealSense

Durante el estudio y observación detenida de los topics por los que se publican los distintos datos recogidos del entorno por lo sensores del robot, se percibió un problema relacionado con la cámara Intel RealSense, mediante la cual se obtiene la información de la nube de puntos que posteriormente se transformará en los Octrees de OctoMap. El problema identificado radica en que, debido a la posición inicial de la cámara en el robot Husky, las imágenes capturadas no eran adecuadas. A continuación, en las figuras 28 y 29 se puede observar la posición original de la cámara sobre el robot y la imagen que esta proporcionaba dada esa posición.

Figura 28: Posición original de la cámara RealSense sobre el robot Husky.

Figura 29: Imagen proporcionada por la cámara RealSense en su posición original.

Como se puede apreciar, la calidad de la imagen capturada no era satisfactoria, ya que se veía parcialmente obstruida por la base del robot y estaba colocada detrás de otros sensores y componentes del robot, lo que interfería con su campo de visión óptimo.

Para solucionar este problema, se realizaron ajustes en varios archivos del repositorio, modificando la posición de la cámara RealSense. Las Figuras 30 y 31 muestran tanto el cambio de posición de la cámara en el robot como la mejora en la calidad de la imagen obtenida gracias a esta nueva ubicación. Con estos ajustes, se logró una visión más clara y sin obstrucciones, mejorando significativamente la capacidad del robot para interpretar su entorno.

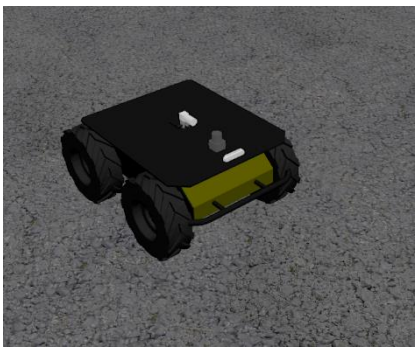

Figura 30: Posición modificada de la cámara RealSense sobre el robot Husky.

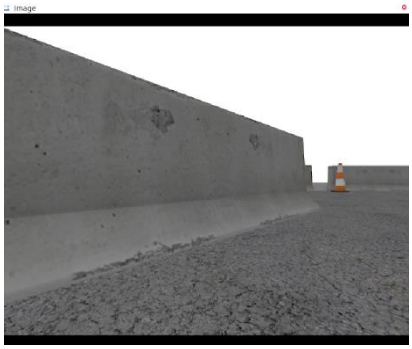

Figura 31: Imagen proporcionada por cámara RealSense en posición modificada.

5.3 Creación de mapas a partir de Octrees

La técnica de creación de mapas utilizando Octrees representa un avance significativo en los campos de computación gráfica y procesamiento de imágenes, especialmente para la representación eficaz del espacio tridimensional (3D). Los Octrees, como se ha mencionado en puntos anteriores, son fundamentales para gestionar y visualizar espacios complejos, objetos detallados y grandes conjuntos de datos, optimizando el uso de recursos y mejorando el rendimiento.

En el contexto de este Trabajo de Fin de Grado, el primer desafío a afrontar fue la generación de un primer mapa empleando estructuras jerárquicas de Octrees, utilizando la biblioteca OctoMap. Este paso inicial fue crucial, ya que implicaba no solo aprender a manejar la biblioteca OctoMap, sino también adquirir habilidades para procesar y analizar los datos proporcionados por los Octrees.

Tras la fimiliarización con el entorno de simulación de Gazebo, el siguiente paso fue instalar y configurar la biblioteca OctoMap. Esta herramienta, permite representar el espacio del entorno de simulación en Gazebo a través de Octrees. La utilización de OctoMap es crucial para una representación precisa del entorno 3D en el que se desarrollarán las simulaciones.

Para crear el primer mapa basado en Octrees se aborda el problema desde dos enfoques diferentes, logrando resultados satisfactorios únicamente con uno de ellos. A continuación, se describen ambos métodos y se explica por qué uno fue efectivo y el otro no cumplió con las expectativas. El primer enfoque, que inicialmente condujo a un error, impulsó a buscar alternativas y a desarrollar un segundo método que resultó más exitoso.

El análisis de estos dos enfoques no solo permitió identificar la solución efectiva, sino que también proporcionó una comprensión más profunda de los desafíos y las posibles soluciones en la creación de mapas 3D usando Octrees, lo que ha contribuido significativamente al aprendizaje del autor en este campo.

5.3.1 Localización y Mapeado simultáneo (SLAM)

Inicialmente, se opta por implementar la técnica de Localización y Mapeo Simultáneo (SLAM) para construir un mapa de un entorno desconocido, mientras se localizaba al robot dentro de él. Paralelamente, se ejecutaba el mapeo utilizando la biblioteca OctoMap, que emplea estructuras de Octree para una representación tridimensional.

El método SLAM [22] es fundamental para robots autónomos que operan en entornos no mapeados previamente. Este método integra varios sensores para calcular la posición del robot y generar un mapa simultáneamente. Al combinarlo con OctoMap, obtenemos una visión 3D del entorno, superando la limitación bidimensional del SLAM tradicional.

Sin embargo, este enfoque presentó desafíos significativos debido al alto coste computacional. Se realizaron múltiples pruebas para obtener mapas precisos tanto en 3D (con Octrees) como en 2D (con SLAM), pero los resultados fueron insatisfactorios. Las

limitaciones del procesador del ordenador utilizado para la simulación resultaron en mapas imprecisos y poco fiables para guiar al robot en un algoritmo de evasión de obstáculos.

El mapa basado en Octrees demostró ser inestable, con constantes cambios y la generación inadecuada de nuevos Octrees a medida que el robot avanzaba o giraba. A continuación, en las siguientes figuras (figuras 32 y 33) se presentan ejemplos de intentos fallidos de generación de mapas con este método. En ambas figuras se puede ver un mapa que no se ajusta al entorno real de simulación.

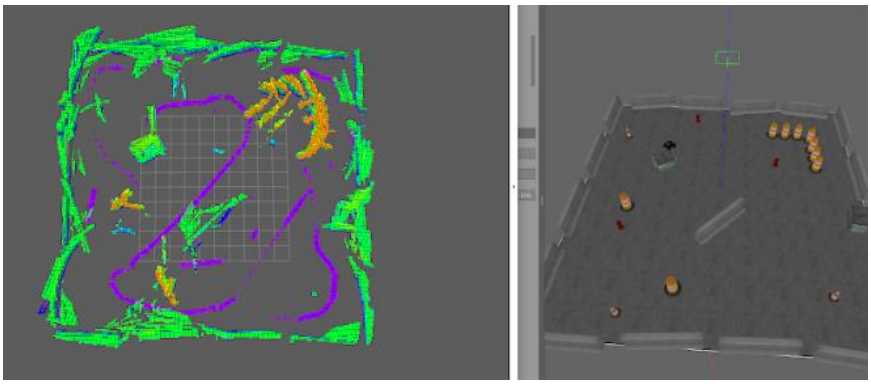

Figura 32: Creación de mapa con OctoMap mediante método SLAM.

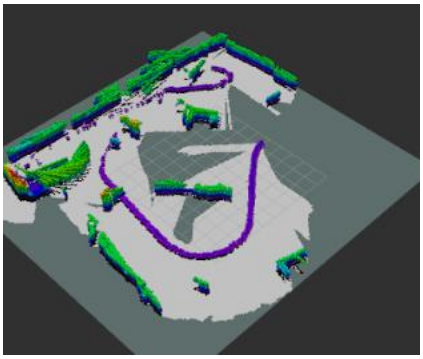

Figura 33: Creación de mapa con OctoMap mediante método SLAM (II).

Antes de explorar una alternativa para generar un mapa más estable y consistente, que sirva de base para un futuro algoritmo de evasión de obstáculos, se investigaron otras posibles causas de la inexactitud en la generación del mapa. Una de estas revisiones incluyó verificar la orientación y posición de la cámara para asegurar su alineación con el movimiento del robot Husky. Esto implicó un exhaustivo análisis de las transformadas y su correlación con los comandos enviados al robot. Finalmente se concluyó que esta no era la causa raíz del problema y se dio paso a la investigación de otros posibles métodos de ejecución.

5.3.2 Localización de Monte Carlo Adaptativa (AMCL)

En la siguiente fase o línea de esta investigación, con el fin de reducir el coste computacional y con ello obtener mejores resultados, más precisos y con menos ruido de la representación tridimensional del entorno simulado basado en Octrees, se plantea el uso del algoritmo o método AMCL [22], algoritmo utilizado en robótica para la localización de un robot en un mapa previamente conocido. La implementación de este método también estaba incluida en el repositorio.

El AMCL utiliza un método de filtro de partículas para seguir múltiples hipótesis de la posición del robot, lo que permite la corrección de errores y la adaptación a cambios en el entorno. AMCL puede manejar incertidumbres y es comúnmente usado en navegación autónoma para determinar la posición del robot con respecto a un mapa, lo cual es crucial para tareas como la planificación de trayectorias y la navegación.

El uso de este método de localización daría solución al problema expuesto anteriormente en la sección 3.1 del capítulo 5, ya que al ejecutarse sobre un mapa ya conocido, el coste computacional se reduce notablemente, haciendo más factible su procesado para mi ordenador.

Tras varias pruebas, se consiguió la generación del mapa tridimensional basado en Octrees estable y muy aproximada a la realidad. Con este método, finalmente se pudo construir un mapa sólido para el desarrollo del algoritmo de evasión de obstáculos. A continuación, se añaden algunas imágenes (figuras 34 y 35) que muestran este mapa estable y fiable.

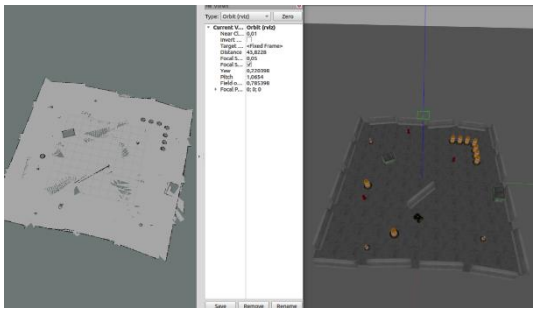

Figura 34: Mapa 2D creado usando método ACML.

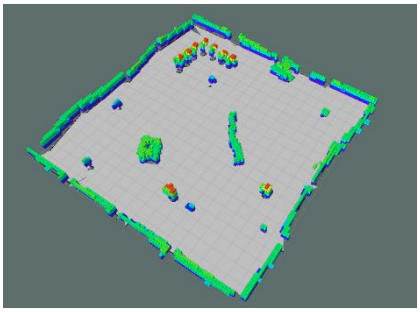

Figura 35: Mapa 3D OctoMap creado usando método ACML (II).

Con la creación de este mapa 3D estable basado en la biblioteca OctoMap, se concluye la primera fase del trabajo. Tras esta primera parte se obtuvo los conocimientos suficientes para el tratamiento y procesamiento de los Octrees y del mapa que posteriormente se utilizarán en el algoritmo de navegación del robot para evitar obstáculos.

5.4 Descripción del algoritmo empleado

Para desarrollar un método que permita la navegación autónoma y la evasión de obstáculos del robot Husky en el entorno de Gazebo, tal y como se menciona en el capítulo 2, se ha escogido como base para la programación el algoritmo RRT (Rapidly-exploring Random Tree).

El objetivo del código que se ha desarrollado es implementar un método basado en RRT de planificación de navegación reactiva en el robot Husky a través de ROS. Un resumen del funcionamiento del algoritmo implementado es el siguiente: el código genera en momentos específicos un número determinado de puntos aleatorios dentro de un campo definido, basándose en la posición actual del robot. Evalúa la viabilidad de alcanzar cada uno de estos puntos, descartando aquellos que estén ocupados o sean inaccesibles. Entre todos los puntos seleccionados elige el más cercano a la posición final que se quiere alcanzar. Una vez seleccionado ese punto de destino intermedio que permita acercarnos a ese punto final, se comanda la velocidad del robot, generando primero el giro del robot orientándolo primero hacia ese punto y luego avanzando hacia él. El código presta atención en todo momento al mapa y a los obstáculos de alrededor, teniéndolos en cuenta en cada movimiento para evitar colisiones.

Este algoritmo se puede ilustrar gráficamente mediante un diagrama de flujo, que se presenta a continuación en la figura 36.

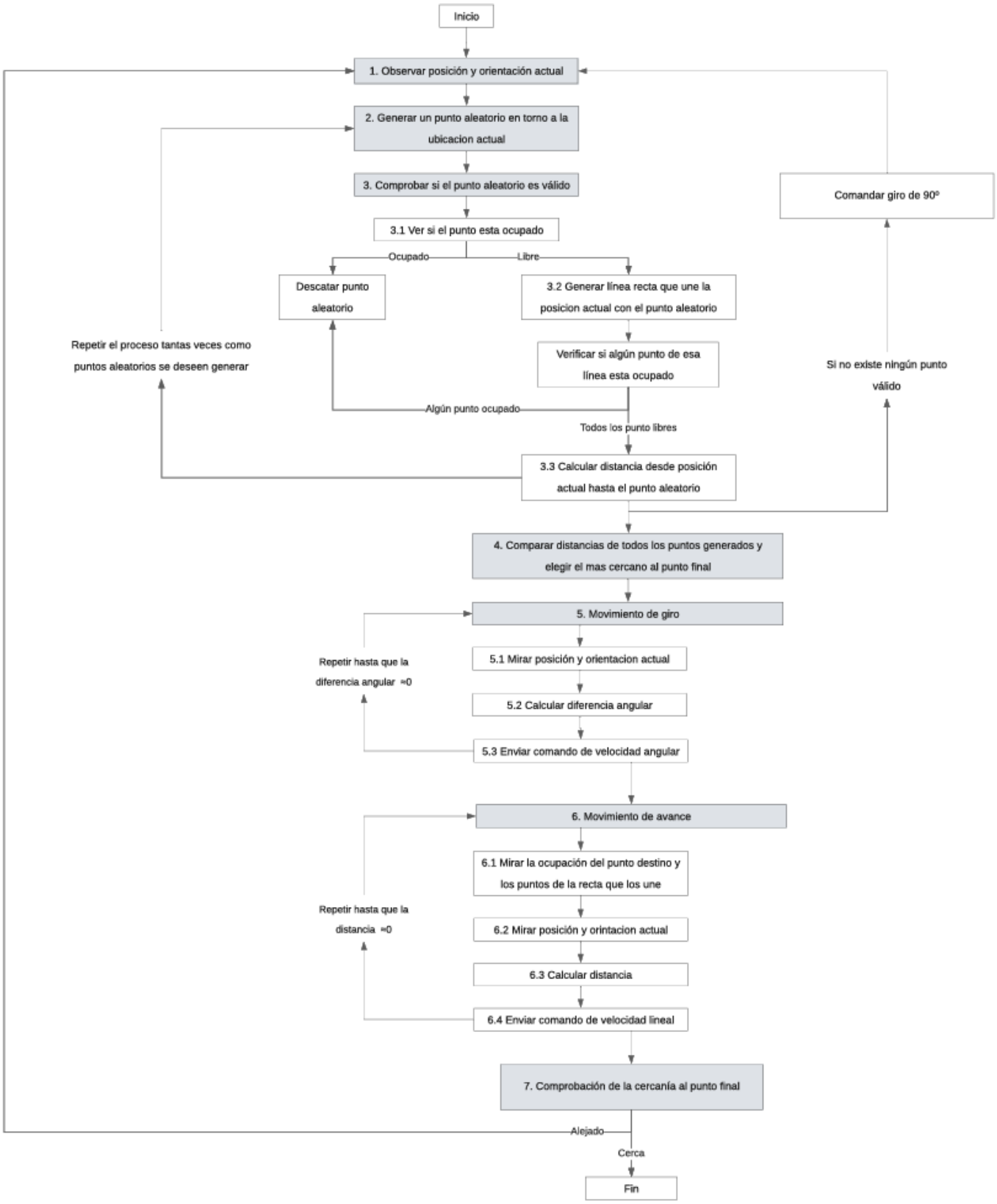

Figura 36: Diagrama de flujo del código creado para implementar el método RRT.

Una vez explicado en líneas generales el procedimiento que sigue el código se profundizará en cada una de sus partes.

1. **Observar la posición y la orientación actual.** Será necesario obtener este dato para situar al robot en un punto del mapa y generar o simular lo que sería su campo de visión o un área cercana donde posteriormente ir generando los distintos puntos aleatorios. En caso de no conocer este dato, los punto serían generados en cualquier parte del mapa, puntos que podrían no estar dentro del campo de visión del robot y por tanto no tener información o tener información no fiable acerca de ellos. Esto provocaría unos cálculos y una navegación poco fiable y segura.

   Para obtener esta posición y esta orientación se hará uso de la función "set_puntoOdom()" dentro de odometria.cpp (Anexo I apartado 3.2). Esta función procesa el mensaje de odometría enviado por el topic de ROS y devuelve dos vectores, uno con la posición (x, y, z) del robot y otro con el cuaternio de orientación.
   Además, se hace uso de funciones como "VectorToPoint3d" y "CuaternionToEulerAngles" dentro de "utilities.cpp" (Anexo I punto 5.2) para transformar el vector posición a un tipo punto tridimensional y el cuaternio a un ángulo de Euler dado en radianes respectivamente.

   En el caso de no recibir un valor de odometría y no poder obtener la posición el código no se seguirá ejecutando, expresando un mensaje de error indicando que no ha recibido este valor. El código se quedará a la espera de este valor

2. **Generar un punto aleatorio en torno a la posición del robot:** Para generar este punto aleatorio se hace uso de la función "generateRandomPoint" dentro de utilities.cpp (Anexo I punto 5.2). Esta función, haciendo uso de la posición actual de robot y de su orientación, así como de unos parametros tanto de radio máximo y radio interior calcula un punto aleatorio haciendo uso de una distribución uniforme discreta. Se usa la función rand() la cual genera números pseudoaleatorios que siguen este tipo de distribución. Es posible utilizar otro tipo de distribuciones o formas aleatorias de generar los puntos, al igual que es posible variar el número de puntos generado. Con un mayor número de puntos generados, existirá una mayor posibilidad de encontrar el punto más óptimo o cercano al punto final, cosa que también supondrá un gran gasto de cálculos y coste computacional. A menor número de puntos menos posibilidades existirá de encontrar este punto, dando lugar así a una compilación más ligera y menos gasto de computación. Aun así, con independencia del número de puntos que se elija, siempre la función se quedará con el mejor de ellos.

   La función "generateRandomPoint" genera un valor de ángulo y un valor de distancia, ambos dentro de los márgenes de campo de visión establecidos. Con ese valor de distancia y de ángulo, mediante cálculos trigonométricos obtiene un valor de x y de y. En este caso, el valor de z se mantiene constante en un offset o valor predeterminado de 0,4 metros. Este valor es elegido dadas las especificaciones del robot, la altura de este y la posición de la cámara.

   Con este punto se establece el inicio de uno de los bucles principales de ejecución dentro del algoritmo. Se volverá al inicio de este tantas veces como puntos aleatorios se quieran generar y validar.

3. **Comprobar si el punto aleatorio es válido:** Para decir que el punto generado aleatoriamente es un válido es necesario tener en cuenta dos factores. En primer lugar, que el punto en sí esté libre de obstáculos. En segundo lugar, que se pueda ir hacia él en línea recta y no haya ningún obstáculo en esta recta. Es necesario que se verifiquen ambos factores dado que, tanto como si el punto está ocupado, como si está libre pero no se puede acceder, se debe descartar. Estos factores se validarán de forma secuencial, ya que si el punto está ocupado se descarta directamente y no es necesario comprobar si se puede llegar a él en línea recta. En caso de que el punto no se encuentre ocupado será cuando se pase a verificar este segundo punto.

   3.1 Comprobar la ocupación: Observar si en esa posición existe o se detecta un obstáculo. Para ver si el punto está ocupado se hará uso de la función "obstáculo cerca" dentro de "detecta obstaculo.cpp" (Anexo I apartado 4.2). Esta función solo necesita el valor de ese punto como punto tipo 3D (tipo de dato con el que trabaja la librería OctoMap). Esta función busca el Octree del nodo correspondiente a esa posición en OctoMap y mira la información que contiene. En el caso de que contenga información el punto estará ocupado, en el caso de que esté vacío el punto se determinara como libre. Aquí es donde entra en juego las distintas políticas de ocupación de las que se habla en el apartado 2.2.2 de este documento. Existen diferentes formas de tratar estos datos provenientes de los Octrees y a raíz de ellos se puede cambiar la forma de la toma de decisión con respecto a la ocupación del punto y con ello la forma de comandar al robot. En este caso se ha considerado que la política de ocupación libre es óptima para este proceso puesto que se trata de una forma simple que permite simplificar cálculos y acortar procesos y tiempo de computación, cosa muy relevante en la navegación reactiva.
   Finalmente, esta función devuelve un valor booleano indicando la ocupación del punto. En función de este valor se procede al siguiente paso o se descarta el punto directamente
   De igual forma que se evalúa el punto aleatorio elegido se evaluará una circunferencia de puntos dispuestos alrededor de el en una distancia cercana, ya que el robot no es un objeto puntual, sino que hay que tener en cuenta sus dimensiones a la hora de consultar la ocupación de un punto. Para realizar esto se hace uso de la función "createCircunferencePoint3d" dentro de utilities.cpp (Anexo I apartado 5.2) que, a partir de un punto central, un radio y un número de puntos devuelve un vector con las posicione de los puntos que forman la circunferencia. Una vez realizado esto se vuelve a usar la función "obtáculo_cerca"
   Mencionada antes, para cada uno de los puntos

   3.2 Generar una línea recta entre el punto actual en el que se encuentra el robot y el punto aleatorio y generado. Si el punto no se encuentra ocupado, es necesario verificar si la línea recta que separa la posición del robot de este punto, se encuentra libre de obstáculos. Para ello se hace uso de varias funciones. En primer lugar, se generan 3 rectas o vectores de puntos tipo 3d intermedios mediante la función "createVectorIntermediatePoints3d" dentro de utilities.cpp (Anexo I apartado 5.2). Esta función solo necesita un punto inicial, un punto final y un parámetro que indique la distancia de separación de cada punto intermedio de las rectas creadas. La elección de este valor deberá elegirse teniendo en cuenta el coste

computacional ya que a menor valor de este parámetro más puntos serán generados y por tanto más puntos será necesario almacenar y evaluar. Para la elección óptima de este parámetro debe tenerse en cuenta esto junto con la resolución de los Octrees del mapa. Un valor de este parámetro que nos asegure que se van a detectar todos los obstáculos u Octrees ocupados intermedios debe de ser menor de la mitad del tamaño de un Octree para no perder información relevante.
Estos puntos intermedios de las rectas se guardarán en un vector de tamaño variable para, posteriormente, mediante un bucle de ejecución, se observe uno a uno la ocupación de cada uno de ellos. De igual forma que ocurría en el apartado anterior, la razón por la que se hace uso de tres rectas y no de una sola es para tener en cuenta las dimensiones del robot. De nuevo, se usará la misma función que en punto anterior (punto 3.1 del proceso) "obstaculo_cerca" siguiendo el mismo proceso ya descrito.
Si tan solo uno de los puntos intermedios de la recta se califica como ocupado, este punto aleatorio elegido se descarta ya que se entiende que no es posible llegar a él en línea recta.

Tras estos dos procesos, si el punto es calificado como válido se calcula la distancia desde la posición actual del robot hasta este punto mediante la función "calculateDistance" dentro de utilities.cpp (Anexo I apartado 5.2). Esta función simplemente calcula la distancia euclídea entre dos puntos.

En este punto se finaliza el bucle de ejecución que se inicia en el punto 2. Este bucle se repetirá tantas veces como puntos aleatorios se determine que hay que generar.
Finalmente, tras finalizar este bucle o proceso se obtendrá como resultado un vector que contiene todos los puntos aleatorios generados que han sido determinados válidos y además un vector con las distancias calculadas de cada uno de estos al punto final.

4. **Comparar las distancias de cada punto generado**. De estos vectores que se acaban de crear en el punto 3 de este proceso, se escoge la distancia de menor valor y el punto que le corresponde, es decir, de todos los puntos aleatorios posibles, se elige aquel que más nos permita acercarnos al punto final. Este punto intermedio lo denominaremos punto destino. Se hace uso de la función "minimumValueIndex" dentro de utilities.cpp (Anexo I apartado 5.2) la cual evalúa punto a punto el vector de las distancias para quedarse con el menor valor.

5. Movimiento de giro

5.1 Observar la posición y orientación actual del robot. De igual forma y utilizando las mismas funciones que las mencionadas en el punto 1, se obtiene la posición actual del robot. Esto es importante realizarlo en cada ejecución del bucle de movimiento para poder situar al robot en el espacio conforme se va moviendo.

5.2 Calcular la diferencia angular. Teniendo en cuenta la orientación a la que se encuentra el robot y la orientación final que se desea, se calcula la diferencia entre ambas. Esta diferencia irá aproximándose a cero conforme la orientación del robot se vaya acercando a la deseada. En este punto se ha tenido especial cuidado con la forma de representar el valor de los ángulos. Se ha elegido una representación en radianes entre -π y π para que los giros se realicen siempre de la forma más optima y girar hacia el lado de menor ángulo. Esta diferencia angular también proporcionará el sentido de giro en función del signo de esta.

5.3 Enviar el comando de velocidad angular. Una vez calculada esta diferencia angular, mediante la función "motionControlAngle" dentro de motion.cpp (Anexo I apartado 5.2) se transforma este valor en un formato de mensaje apto para ser enviado por el topic de ROS correspondiente comandando así la velocidad angular del vehículo. Para comandar esta velocidad angular se ha propuesto un control proporcional, por tanto, el valor de esta velocidad de giro será mayor cuanto mayor sea la diferencia de ángulo. De esta forma, cuando se esté acercando al ángulo deseado se reducirá la velocidad. Por otro lado, dadas las condiciones no holonómicas del robot Husky, no es posible comandar la velocidad solo en efectos de velocidad angular. Para que el robot pueda efectuar el giro se añade una velocidad linear de valor muy reducido. Cabe mencionar que es posible provocar un giro del robot completamente sobre sí mismo haciendo que cada par de ruedas laterales giren en sentidos distintos, pero esta opción es descartada, entre otros motivos, porque este tipo de movimiento provoca que el valor de error de la odometría del robot crezca demasiado, siendo esto perjudicial para la navegación.

Se dará por concluido y se dejará de ejecutar este bucle o proceso de giro cuando el valor de la diferencia angular se encuentre dentro de la tolerancia establecida y sea prácticamente igual a cero.

6. Movimiento de avance.

6.1 Observar la ocupación del punto de destino, así como la ocupación de los puntos de la recta que los une. Es importante añadir esta verificación antes de avanzar, puesto que tras el giro, o con forme se va avanzando, es probable que se hayan detectado nuevos obstáculo los cuales se encontraran fuera del campo de visión del robot. Esto es debido a las condiciones y especificaciones de la cámara, dependerá tanto del rango de profundidad de visión, como del campo de visión.
Este punto no es necesario realizarlo en el comando de giro ya que el robot gira sobre sí mismo y se tiene la certeza de que no va a colisionar. Sin embargo, en el avance, si es posible que aparezcan nuevos objetos que puedan interferir en el movimiento. Esto además favorecerá y permitirá la navegación del robot en escenarios dinámicos.

6.2 Observar la posición y orientación actual del robot. De la misma forma que, tanto en el apartado anterior, como el punto 1 antes mencionado, se obtiene la posición actual del robot. Se vuelve a hacer uso de las mismas funciones y el mismo método para ello. De igual forma que ocurre en el comando de giro, es importante conocer la posición del robot en cada ciclo de ejecución.

6.3 Calcular la distancia al punto destino. Haciendo uso de la función "calculateDistance" dentro de utilities.cpp (Anexo I apartado 5.2) mencionada en el punto 3, se va calculando la distancia restante al punto de destino conforme el robot va avanzando. A medida que el robot se vaya acercando al punto de destino este valor de distancia euclídea irá disminuyendo.

6.4 Enviar el comando de velocidad lineal. De igual forma que ocurre para la velocidad angular, se ha creado la función "motionControlDistance" de motion.cpp (Anexo I apartado 6.2) para transformar este valor de distancia en un mensaje apto para enviar por el topic correspondiente de ROS usado para comandar la velocidad. Así mismo, también se ha propuesto un control proporcional, en función de la distancia, para que vaya más rápido cuando esté más lejos del punto y reduzca la velocidad cuando se acerque En este caso no se ha tenido en cuenta el signo de la distancia puesto que esta siempre será positiva, ya que previamente se ha orientado el robot y el avance se debe realizar siempre hacia adelante.

Bajo un criterio similar al del movimiento de giro, se dará por finalizado el proceso de avance y se dejará de ejecutar este bucle cuando el valor de la distancia se encuentre dentro del valor de tolerancia establecido muy cercano a 0.

Tras estos movimientos, se mandará un comando de parar el movimiento del robot Husky mediante la función "motionControlSTOP" dentro de motio.cpp (Anexo I apartado 6.2) para comandar con valor 0 tanto el movimeinto de giro como el de avance. Parando así el robot hasta que se elija un nuevo punto al que ir.

Tras el desarrollo de estos dos puntos en los que se explica cómo se determina y se comanda el movimiento, es necesario hacer algunas aclaraciones.

En primer lugar, la elección de este método de movimiento, en el que en primer lugar se gira y luego se avanza ha sido elegida con el fin de hacer el movimiento más preciso y seguro. En el caso elegir otro tipo de control para el movimiento, estos desplazamientos no se harían de forma recta (a no ser que el robot se encontrase perfectamente orientado al punto) sino que tendrían un radio de curvatura. Dado que el algoritmo que se ha elegido para simular es el RRT, que los movimientos se efectuasen de forma curva no era compatible con la elección de caminos de este método, ya que en él las conexiones entre puntos se hacen en línea recta. Destacar que es posible modificar esto y crear una variante del algoritmo, para conseguir un movimiento que se vea visualmente más fluido, pero los cálculos y verificaciones de obstáculos intermedios deberían calcularse a partir de modelos matemáticos mucho más complejos que estimen la curvatura y el camino que seguirá el robot a partir del comando de velocidades angulares y lineales. El uso de la línea recta es más rápido y menos costoso computacionalmente. Esto favorece la rapidez, un requisito necesario en la navegación local y reactiva.

Por otro lado, también es posible cambiar este tipo de control proporcional implementado por otros más complejos que incluyan algún tipo de controlador integral o derivativo; o incluso por uno más simple en el que los valores de velocidad sean siempre constantes.

7. **Comprobación de la cercanía al punto final:** Una vez se ha terminado de comandar el movimiento es necesario verificar la cercanía de este al punto final. Si la distancia se encuentra dentro de los márgenes establecidos se dará por finalizado el procedimiento, quedándose a la espera de recibir un nuevo punto final al que desplazarse.

Todas las funciones y códigos mencionados en esta explicación se encuentran con sus respectivos comentarios y cabeceras en el Anexo I para su consulta. En este caso, y para la realización del proyecto, no se ha hecho uso de código o de funciones externas. Dado que son funciones para usos específicos y particulares de este trabajo, he decidido crearlas yo. Las distintas funciones son totalmente adaptables a otros contextos y trabajos. Para la consulta de los fundamentos teóricos o de los cálculos realizados en algunas funciones consultar el capítulo 3.

Será en el apartado siguiente en el que se explique más detalladamente la estructura y la conexión de este algoritmo RRT en ROS y con el entorno simulado en Gazebo.

5.5 Integración del código en ROS y el entorno de simulación de Gazebo

Este proyecto toma como punto de partida el repositorio "Husky" de Tinker Twins, el cual, tal y como se describe en el apartado 2 de este capítulo, simula el robot Husky en unas condiciones de entorno específicas. Es relevante mencionar que a este repositorio se le ha realizado una serie de modificaciones para adaptarlo a las necesidades y requerimientos de este proyecto, todos estos cambios se encuentran reflejados en el apartado 3 de este capítulo.

Para lograr el objetivo y conseguir la navegación autónoma de este vehículo mediante la observación, uso y procesado de mapas tridimensionales probabilísticos como son los Octrees de la librería OctoMap es necesario implantar y ejecutar el código que he elaborado y que se desarrolla y explica detenidamente en la sección anterior (sección 4). Para ello es necesario que quede todo completamente integrado en ROS y en los distintos nodos que ejecuta el repositorio.

La siguiente imagen, figura 37, presenta una visión global de la integración del programa en C++ creado por mí, en el sistema operativo ROS. La imagen muestra el diagrama de arquitectura del nodo de ROS creado. Esta es solo una parte del esquema de simulación global del entorno, la parte que representa la integración del código creado para este proyecto que simula el algoritmo RRT en el resto del entorno ROS.

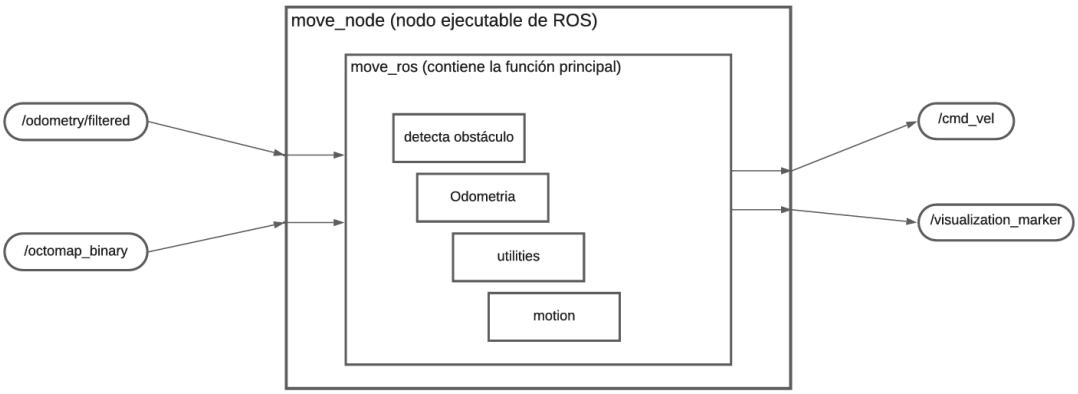

Figura 37: Arquitectura nodo ROS para algoritmo RRT.

El código principal se encuentra en el archivo move_ros.cpp, en él se encuentra el programa que ejecutará el algoritmo basado en RRT. En este programa principal se van llamando e instanciando las distintas funciones de otros archivos y programas.

A su vez, este programa principal está instanciada dentro del archivo move_node.cpp, el cual contiene la configuración que lo convierte en nodo ejecutable para ROS, dado que contiene la función "main".

Tener separado el archivo ejecutable del resto de código favorece la versatilidad de este y del proyecto, ya que este código será ejecutable por sí solo siempre y cuando tenga las entradas sensoriales correspondientes, que en este caso provienen de topics y se publican en otros topics. Se trata de una arquitectura modulable.

En la parte izquierda se pueden ver los dos topics que hacen la labor de proporcionar la información de entrada de los sensores. En este caso, se proporciona la información de la odometría del robot mediante el topic "/odometry/filtered" y la información de los objetos captados por la cámara en forma de Octree mediante el topic "/octomap_binary". Por otro lado, en el lado derecho aparecen los topic que envían los valores de salida a distintos actuadores, en este caso, los comandos de velocidad se estarán publicando por el topic "/cmd_vel" y los valores de los distintos puntos y grafos a representar calculados por el algoritmo mediante el topic "/visualization_marker.".

Una vez se ha integrado el código perfectamente como nodo ejecutable en ROS es posible lanzar la simulación.

La siguiente imagen, figura 38, muestra una visión o esquema global de todos los nodos que se ejecutan durante la simulación de este proyecto:

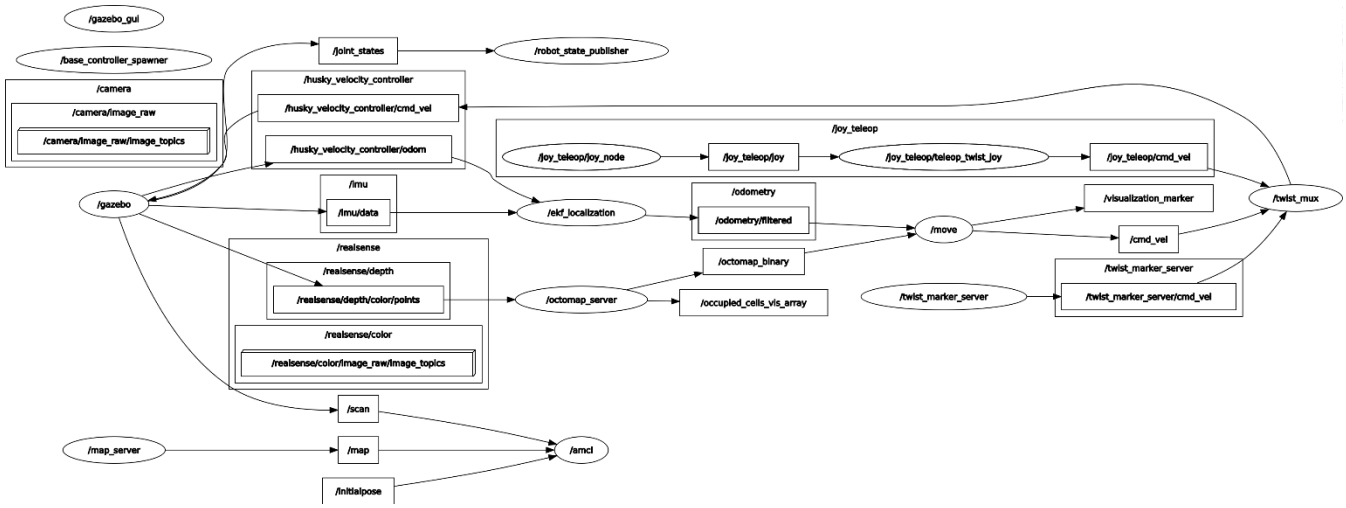

Figura 38: Esquema global de nodos de ROS durante la simulación.

Empezando por la parte derecha de la imagen, al inicio del gráfico, aparecen los nodos "/gazebo" y "/gazebo_gui", estos son los nodos del simulador 3D que proporciona la capacidad de simular el robot Husky en un determinado entorno virtual y el que representa la interfaz gráfica del usuario respectivamente.
El nodo "/gazebo" interactúa con otros nodos que controlan aspectos específicos del robot o proveen información sensorial.

El nodo "/base_controller_spawner" es un nodo específico utilizado en ROS para gestionar el lanzamiento de controladores para una base robótica. Este nodo facilitará la activación y desactivación de los controladores que permiten al robot interactuar con su entorno, por ejemplo, controlando su movimiento. Es importante destacar que este tipo de configuración permite una gran flexibilidad en el desarrollo y operación de robots, permitiendo a los desarrolladores adaptar el comportamiento del robot a una amplia gama de tareas y entornos.

Otro de los nodos que aparece en la parte derecha del esquema es el nodo "/map_server" y será el nodo encargado de proporcional el mapa al resto, esto será crucial para la navegación y la localización.

El nodo "/acml" hace uso de la información proporcionada por el nodo "/map_server" y nodos derivados de él así como de otros valores para estimar la posición del robot dentro del mapa.

El nodo "/robot_state_publisher" mantiene la información sobre el estado actual de las partes y articulaciones del robot, información proveniente del nodo "/joint_states" el cual parte del entorno de simulación gazebo y de las partes del robot.

Mediante una serie de transformaciones (nodo "/tf"), esta información se traslada a los nodos "/octomap_server" y "/odometry", nodos cruciales en el nuestro programa creado para el algoritmo RRT. Estos dos nodos enviarán la información necesaria al nodo "/move" el cuál se encargará de comandar la velocidad y representar los puntos en el entorno gráfico Rviz.

Este esquema también incluye otros nodos como "/joy_teleop "que permite, si fuese necesario, el control del robot mediante un joystick o por teclado.

Finalmente, todos los comandos de velocidad provenientes de todos los posibles nodos que pueden comandar el movimiento del robot se comunican con "/twist_mux", siendo este un nodo multiplexor de mensajes tipo Twist que tiene como propósito recibir los comandos de velocidad de las distintas fuentes y decidir cuál de estos debe de ser enviado al sistema de control del robot representado por el nodo "/husky_velocity_controller". Este nodo, a la vez de recibir y ejecutar los comandos de movimiento del robot proporciona la información necesaria al nodo "/ekf_localization" el cual ejecuta el filtro extendido de Kalman para estimar la pose del robot. Tal y como se observa en el esquema de nodos el EKF combina los datos de las predicciones de movimiento y las mediciones junto con las incertidumbres correspondientes de ambas combinandola para producir una estimación actualizada y precisa de la foto del robot.

Capítulo 6

Validación Experimental

6.1 Metodología de experimentos

Para llevar a cabo los distintos experimentos y simulaciones se ejecutará el método de navegación local implementado ante distintas distribuciones de obstáculos y simulaciones, con el objetivo de demostrar la validez, versatilidad y adaptabilidad del código ante distintas escenas y circunstancias.

En este capítulo se pretende demostrar el correcto funcionamiento de todas las bases teóricas y todo el trabajo realizado para la elaboración de este proyecto.

El objetivo principal el conseguir simular el correcto funcionamiento del algoritmo RRT de evasión de obstáculos empleando para ello en mapas probabilísticos 3D, en este caso Octrees. Durante la realización de esta fase experimental o de simulación se hará uso del software detallado con anterioridad en este documento en el capítulo 3. Será necesario del uso de Gazebo, Rviz y ROS, así como la librería OctoMap y los distintos nodos del repositorio "Husky".

Se expondrán todos los resultados obtenidos de las simulaciones, abarcando tanto los datos finales como los procesos seguidos Para determinar la validez de estos resultados, se considerarán varios factores clave: la eficacia en la evasión de obstáculos sin causar colisiones, la llegada exitosa al punto de destino seleccionado, y el comportamiento apropiado de las velocidades angular y lineal, asegurando que sean extrapolables a un robot físico real.

Entre las limitaciones encontradas, es relevante mencionar la capacidad reducida del ordenador utilizado para crear, depurar y ejecutar estos nodos. En algunas ocasiones, al intentar procesar cálculos intensivos o manejar un volumen alto de datos, el sistema no opera con la fluidez deseada, resultando en respuestas lentas y a veces inexactas.

El propósito es obtener resultados que no solo validen, sino también corroboren el alto nivel de funcionalidad y fiabilidad de todo el trabajo realizado hasta la fecha.

6.2 Navegación hacia un punto de destino lejano

En primer lugar, se llevará a cabo una simulación donde el robot comienza desde un punto inicial y debe dirigirse hacia un punto de destino preestablecido, situado a una distancia específica. Esta simulación se efectúa utilizando configuraciones que se asemejan estrechamente a las condiciones físicas reales del robot.

En esta simulación en particular, se ha ajustado el campo de visión de la cámara para que abarque aproximadamente 80 grados, alineándose con las especificaciones técnicas de la cámara real utilizada. Además, se ha configurado la resolución de los Octrees para que estos posean un tamaño mínimo de 0.6 metros,

El punto de partida del robot se ha fijado en la coordenada (0,0) metors, mientras que el punto de destino se establece en la coordenada (8,8) metros. Esta elección de puntos permite evaluar la eficiencia del algoritmo en la navegación y evasión de obstáculos en un espacio definido y controlado. En la figura 39 se ilustran los resultados gráficos obtenidos de la simulación mediante la herramienta RViz.

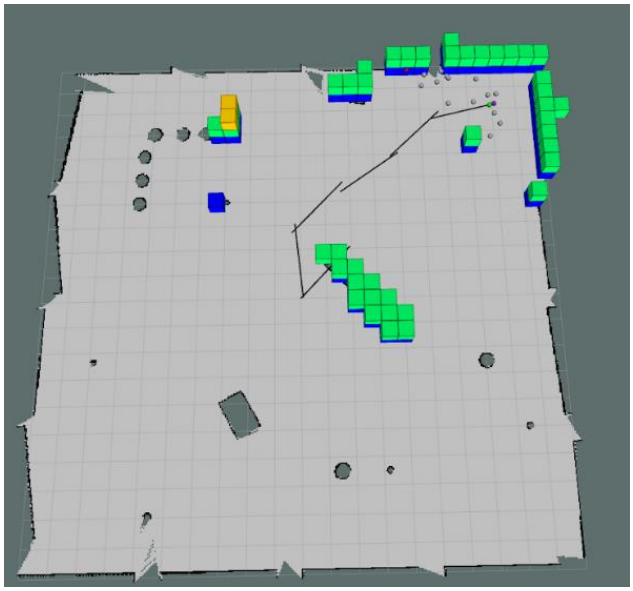

Figura 39: Simulación de navegación hacia un punto de destino alejado mediante algoritmo RRT.

Estos resultados proporcionan información visual que evidencia el correcto comportamiento del robot, así como su capacidad para navegar eficientemente hacia el destino. La interpretación de estos resultados es crucial para ajustar y optimizar el algoritmo y la configuración del robot para aplicaciones en entornos más complejos y variables.

Las líneas rectas color negro son las distintas rectas de cálculo que se han ido creando a partir de la selección del punto aleatorio más cercano al punto final. En este caso concreto, se puede ver como en un primer lugar, con el robot en la posición (0,0) metros el primer punto aleatorio escogido para el cuál se calcula la recta no continua, esto es, debido a que tras orientarse el robot debidamente hacia esa posición, aparecen nuevos obstáculos intermedios y es necesario calcular un nuevo punto. En las siguientes imágenes contenidas en la figura 40 es posible apreciarlo visualmente.

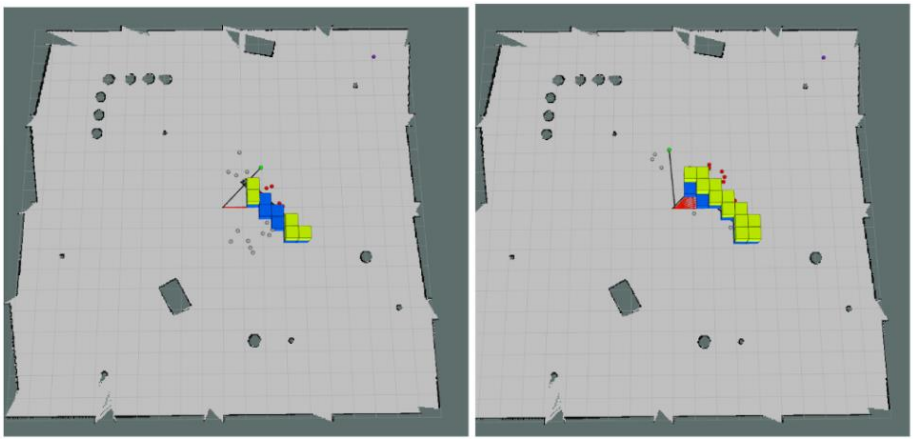

Figura 40: Primeros pasos del algoritmo para un punto lejano.

En la imagen de la derecha se puede ver como en un primer momento se escoge entre los puntos aleatorios generados (color gris) el punto más cercano al punto final (color verde), descartando aquellos que se encuentran ocupados o que no se puede acceder a ellos en línea recta (rojo). En la imagen de la izquierda se puede ver que, tras realizar el giro, se percibe un obstáculo intermedio entre el robot y dicho punto, por tanto se vuelven a generar puntos aleatorios y escoge un nuevo punto destino.

El punto final aparece en la esquina superior derecha del mapa, en el punto (8,8) (color morado oscuro).

En la gráfica que se muestra en la figura 41, se puede observar cómo es el movimiento del robot en función de sus velocidades angular y lineal. En los momentos en los que está avanzando el valor de giro es constante 0 y en los momentos en los que está girando, el avance tiene un valor constante.

En este caso si se ha comandado al robot con un control proporcional para el valor de la velocidad lineal, pero para la angular se ha simplificado añadiendo únicamente el valor constante 0,3. Se ha tomado esta decisión dado que, a la hora de simular, la parte final del giro se realizaba de forma muy lenta, no siendo esto óptimo para un algoritmo de navegación reactiva. Este comportamiento se puede ver gráficamente en la figura 42.

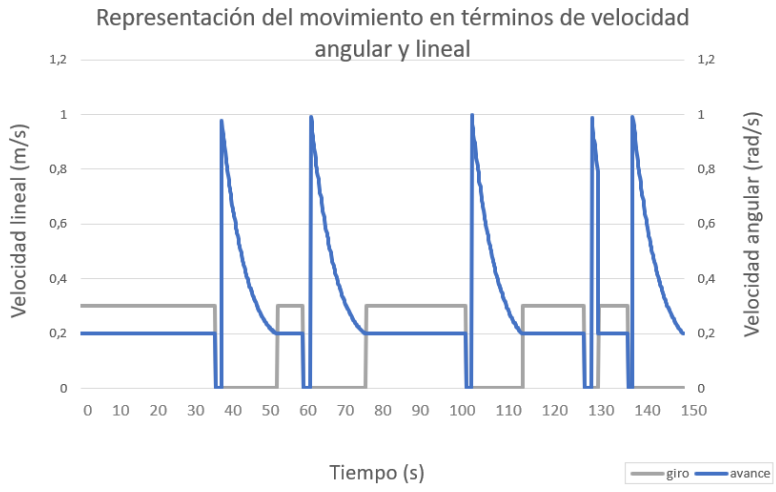

Figura 41: Gráfica velocidades angular y lineal algoritmo RRT para un punto lejano.

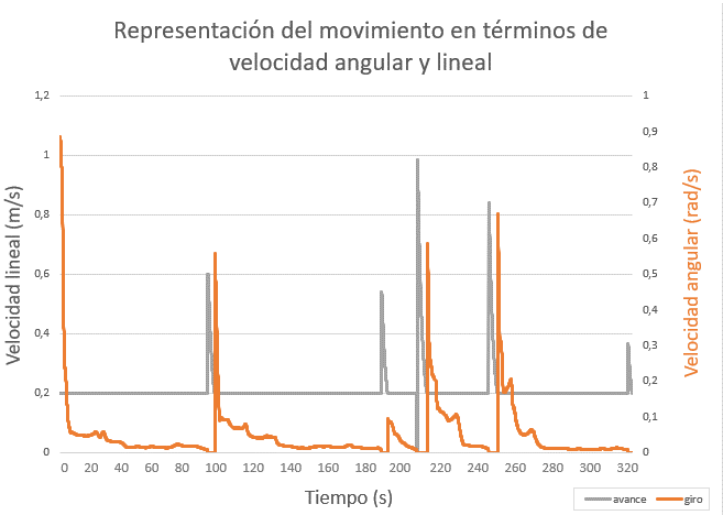

Figura 42: Gráfica velocidades angular y lineal algoritmo RRT (II).

Los giros son una parte muy crítica de la navegación ya que si el robot gira de forma muy brusca en repetidas ocasiones tiende a aumentar su error de odometría, dando lugar así a incoherencias con el programa. Con el modelo de control proporcional para la velocidad angular, los giros se hacen muy inestables ya que conforme se va haciendo 0 la diferencia angular, la velocidad de giro es más lenta costándole mucho más tiempo es llegar a la tolerancia establecida.

6.3 Navegación en distintos escenarios de dificultad

En esta sección se expondrán una serie de simulaciones en distintos escenarios que se han considerado de interés, ya sea por su complejidad o porque permiten demostrar el funcionamiento del algoritmo expuesto a condiciones muy estrictas. Las simulaciones están diseñadas para probar y destacar la capacidad del algoritmo para adaptarse y reaccionar adecuadamente en diferentes situaciones. Tres de estos escenarios a representar son:

- **Esquivar una pared frontal:** Este escenario pone a prueba la habilidad del algoritmo para detectar y evitar un obstáculo grande y directo, como una pared. Es crucial para evaluar la eficiencia del sistema de sensores del robot y su capacidad para tomar decisiones de navegación rápidas y precisas en presencia de obstáculos inminentes.
- **Atravesar un pasillo estrecho:** Este escenario desafía al algoritmo a maniobrar a través de un espacio confinado y limitado. Es una prueba significativa para la precisión y la sensibilidad del sistema, ya que requiere una navegación cuidadosa y precisa para evitar colisiones en un ambiente restringido.
- **Escapar de una trampa local:** Aquí, el algoritmo se enfrenta a la tarea de encontrar una ruta de escape en una situación donde las opciones de movimiento son limitadas, simulando una vía sin salida en el entorno. Este escenario es fundamental para demostrar la habilidad del algoritmo para resolver problemas complejos.

6.3.1 Pared frontal

En este caso de simulación, el robot inicia con una pared puesta justo en frente de su campo de visión. El robot se encuentra totalmente perpendicular a la pared y el punto destino ha sido seleccionado detrás de esta. Deberá rodear la para poder llegar al destino final.

La figura 43 muestra la situación inicial del escenario de simulación, se puede observar en el entorno de siempre, el robot Husky con la mencionada pared a esquivar.

Figura 43: Simulación Pared Frontal Gazebo.

En la figura 44 se ilustran los resultados gráficos obtenidos de la simulación mediante la herramienta RViz. En la imagen se observan algunos pasos de ejecución del algoritmo.

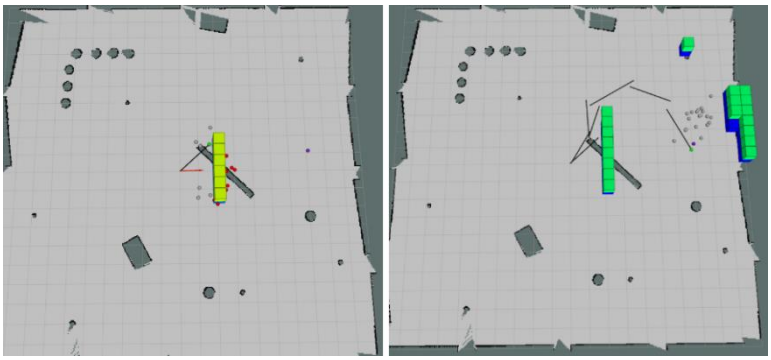

Figura 44: Simulación Pared Frontal Rviz.

En los pasos iniciales, cuando el robot se encuentra más próximo a la pared se puede ver como se escogen hasta 2 puntos destino que son descartados. Finalmente se encuentran los puntos necesarios para bordear esta pared.

Para este caso concreto, cuanto mayor sea el campo visual de la cámara mejores resultados se pueden llegar a obtener. Si el campo visual es muy reducido pueden surgir problemas a la hora de encontrar un punto válido intermedio al que aproximarse. Para este caso concreto, el algoritmo ha sido simulado para una cámara de 120º de visión.

6.3.2 Pasillo estrecho

En este caso de simulación, el robot inicia con dos paredes en sus laterales simulando un pasillo. El punto destino ha sido seleccionado al final de este pasillo, deberá poder avanzar por el pasillo sin colisionar con las paredes laterales.

La figura 45 muestra la situación inicial del escenario de simulación, se puede observar en el entorno de siempre, el robot Husky con el pasillo a atravesar.

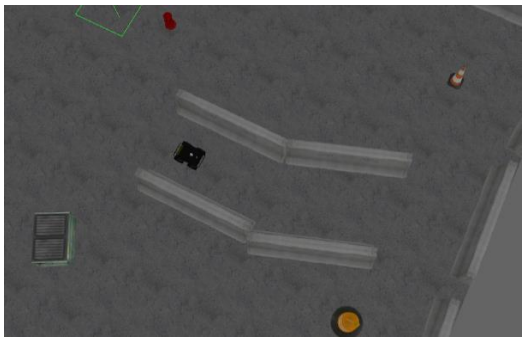

Figura 45: Simulación Pasillo Estrecho Gazebo.

La figura 46 presenta los resultados gráficos de la simulación, visualizados mediante la herramienta RViz. Esta imagen muestra la etapa inicial y la etapa final del algoritmo, ilustrando así cómo se desarrolla el proceso en distintos momentos.

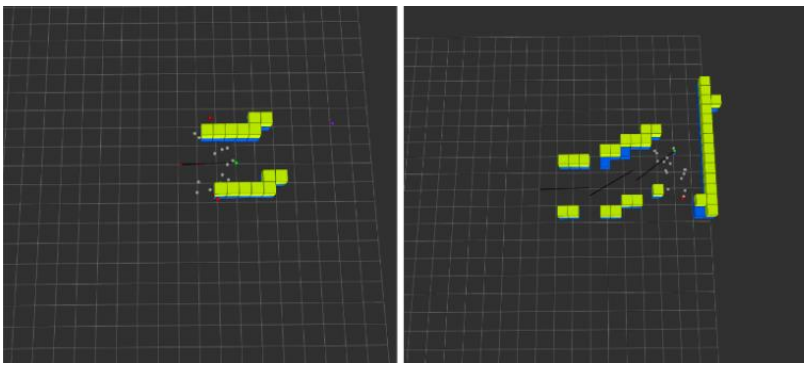

Figura 46: Simulación Pasillo Estrecho Rviz.

Para esta simulación, a diferencia del reto, se ha usado el método SLAM para mapear y localizar el robot debido a los cambios significativos que producía el pasillo nuevo creado para esta simulación en el mapa ya realizado en el repositorio para usar en el método AMCL.

Se puede ver como el algoritmo funciona correctamente para esta situación, los puntos que quedan fuera del pasillo o sobre las paredes de este son descartados y se elige el punto que más se aproxima al punto final que se encuentra al final del pasillo.

A diferencia de la situación anterior, la Pared Frontal, para conseguir atravesar un pasillo con éxito no es necesario tener una cámara con un campo de visión muy amplio. En este caso el aspecto de relevancia está en la distancia del área donde se lanzan los puntos, dado que si el pasillo es muy estrecho y dicha distancia muy grande habrá más posibilidad de que gran parte de los puntos aleatorios se encuentren fuera del pasillo y sean descartados.

6.3.3 Trampa local

En este caso de simulación, se introducirá el punto destino el punto (-8,8), con el fin de observar como el robot es capaz de escapar de esa trampa local que forma los conos color naranja del escenario (parte superior izquierda de la figura 47). La figura 47 muestra la situación inicial del escenario de simulación.

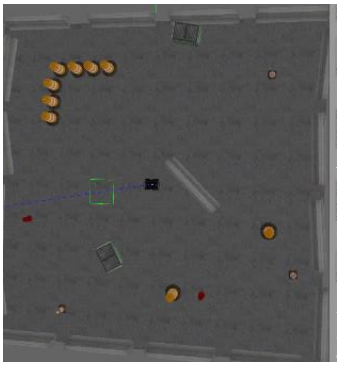

Figura 47*:* Simulación Trampa Local Gazebo.

En este caso de simulación, el robot debe bordear dos paredes que forman un ángulo recto para llegar al punto de destino que se encuentra detrás de estos obstáculos.

En esta simulación nos encontramos ante un problema de mínimos locales en la navegación autónoma. Este problema es un desafío significativo para los algoritmos de búsqueda y optimización utilizados en la planificación de rutas y toma de decisiones en los comandos de navegación.

Este problema, en el contexto de la navegación autónoma surge de una solución que parece óptima o suficientemente buena dentro de una región limitada del espacio de búsqueda, pero resulta no serlo si en vez de observarse esta pequeña región del espacio, se observa el espacio en su totalidad. En otra palabras, mientras que robot piensa que se está acercando al punto objetivo, es posible que no sea así y que se esté adentrando en una ruta sin salida o en una ruta en la que es mucho más probable que le cueste mas salir y llegar al destino. El hecho de que el robot llegue a un punto sin salida puede provocar comportamientos oscilatorios repetitivos lo que conlleva no alcanzar el punto de destino.

Actualmente, el algoritmo creado, la mayoría de las veces es capaz de solventar este problema. Lo hace mediante un pequeño giro de 90º en el caso de que no encuentre ningún punto válido al que moverse.

A continuación, se muestra la simulación de este escenario con imágenes.

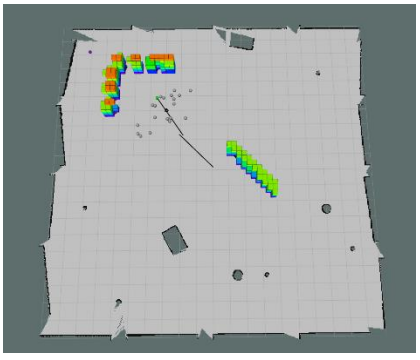

Figura 48: Aproximación a la trampa local, simulación en RViz.

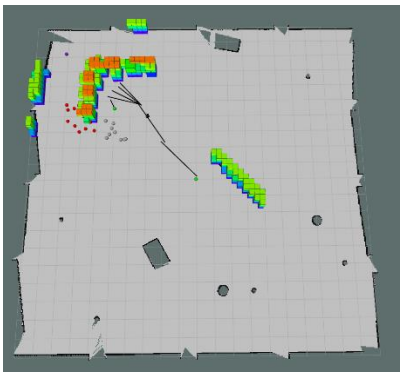

Figura 49: Simulación RViz durante trampa local.

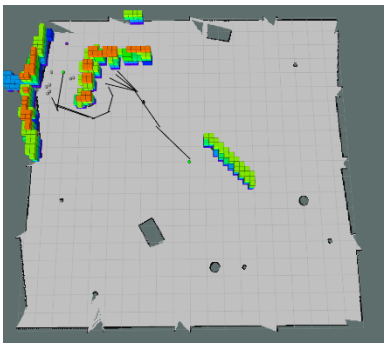

Figura 50: Simulación RViz despues de trampa local.

En la figura 48 se puede ver como el robot se aproxima en línea recta al punto final situado en el (-8,8) metros. Al no encontrarse ningún obstáculo la trayectoria es prácticamente recta. Esto hace que acabe aproximándose cada vez más al ángulo recto formado por los conos naranjas.

En la figura 49 se puede ver como hace varios intentos de aproximarse a varios posibles puntos, pero finalmente, tras no encontrar ninguno valido realiza un giro y retoma el algoritmo implementado. Tras el giro vuelve a lanzar puntos aleatorios y a escoger el más cercano.

Finalmente, en la figura 50, se observa como ha sido capaz de salir del ángulo recto sin salida bordeando el obstáculo hasta llegar al punto final establecido.

Este algoritmo no es siempre capaz de solventar estos problemas y en ocasiones es posible que se planteen situaciones más críticas y no llegue al punto de destino.

Para evitar esto siempre se pueden hacer mejoras tanto en el hardware (sensores más potentes) como el en software (capacidad de almacenar rutas o una planificación global muy potente).

Capítulo 7

Conclusiones y líneas de trabajo futuras

Tras la realización de este Trabajo de Fin de grado y la comprobación y puesta en marcha de su simulación y funcionamiento, se puede afirmar que se han cumplido gran parte de los objetivos de forma satisfactoria.

Se ha logrado crear un algoritmo que permita la navegación autónoma de un robot Husky obteniendo datos de los sensores necesarios para la construcción de mapas probabilísticos tridimensionales.

Este proyecto ha representado un considerable desafío, tanto en términos técnicos como conceptuales. La inmersión en el mundo del mapeo 3D basado en Octrees me ha abierto un amplio abanico de posibilidades y aplicaciones. El aprendizaje obtenido al superar los obstáculos encontrados durante el desarrollo del proyecto ha sido inmensamente enriquecedor, ampliando significativamente mi comprensión y habilidades en áreas críticas de la robótica y la navegación autónoma.

La integración de tecnologías avanzadas de sensores y la aplicación de técnicas de mapeo y navegación autónoma han demostrado ser áreas prometedoras para explorar más a fondo. Este proyecto ha demostrado que, mediante la combinación adecuada de hardware y software, se pueden superar desafíos significativos en la robótica, abriendo la puerta a nuevas soluciones para problemas complejos en entornos dinámicos y cambiantes.

En conclusión, este Trabajo de Fin de Grado no solo ha cumplido con sus objetivos iniciales, sino que también ha contribuido a mi desarrollo personal y profesional. Ha proporcionado una plataforma sólida para futuras investigaciones y ha destacado la importancia de la adaptabilidad y el aprendizaje continuo en el campo de la robótica y la tecnología.

7.1 Líneas de trabajo futuras

Las líneas de trabajo futuro que surgen de este Trabajo de Fin de Grado abren un campo amplio y diverso de posibilidades para la continuación de la investigación y el desarrollo en áreas relacionadas. A continuación, se detallan varias direcciones potenciales para futuras investigaciones y aplicaciones:

- Realización de simulaciones en otro tipo de escenarios más complejos: Dadas las limitadas características del ordenador donde se ha creado y simulado el proyecto queda pendiente su prueba en un ordenador más potente y para unas circunstancias más críticas.

- Mejora y Optimización del Algoritmo Actual: Existe un potencial significativo para perfeccionar el algoritmo de navegación autónoma desarrollado. Esto podría incluir la mejora de la eficiencia en el procesamiento de datos, la optimización de la toma de decisiones en tiempo real y la mejora en la precisión del mapeo 3D.

- Exploración de nuevos sensores que puedan suponer una mejora en la visión del robot: Investigar la integración de nuevos tipos de sensores y sistemas de percepción avanzados, así como es estudio del uso simultáneo de dos o mas tipos de sistemas de captación de imágenes

- Aplicación y pruebas del algoritmo en otro tipo de robots y otro tipo de entornos y condiciones: Probar y adaptar el algoritmo desarrollado para diferentes tipos de robots, incluyendo drones y vehículos autónomos, así como su aplicación en diversos entornos, como urbanos, industriales o naturales.

Bibliografía

[1] Robotnik. (n.d.). Historia de los robots y la robótica. Robotnik Automation SLL. Acceso el 02 de enero de 2024, de https://robotnik.eu/es/historia-de-los-robots-y-la-robotica/

[2] Vázquez Martín, R. Navegación Autónoma: evitación de obstáculos y planificación de caminos. Grado en Ingeniería en Electrónica, Robótica y Mecatrónica, Ampliación de Robótica. Departamento de Ingeniería de Sistemas y Automática, Universidad de [nombre de la universidad].

[3] Romero Marras, J.J. (2016, 9 de abril). Algoritmo DistBug: Teoría. Acceso el 09 de enero de 2024, de https://jjromeromarras.wordpress.com/2016/04/09/algoritmo-distbug-teoria/

[4] Yandún, A., & Sotomayor, N. (2012). Planeación y seguimiento de trayectorias para un robot móvil. Escuela Politécnica Nacional, Quito – Ecuador.

[5] López García, D. A. (2012). Nuevas aportaciones en algoritmos de planificación para la ejecución de maniobras en robots autónomos no holónomos (Tesis doctoral). Universidad de Huelva, Departamento de Ingeniería Electrónica, de Sistemas Informáticos y Automática. ISBN: 978-84-15147-78-7. D.L.: H 53- 2012.

[6] Minguez, J. The Obstacle-Restriction Method (ORM) for robot obstacle avoidance in difficult environments. IEEE/RSJ Int. Conf. Intell. Robot. Syst. Edmonton, Alta, Canada, 2005, 3706–3712

[7] Inloc Robotics. (n.d.). Nube de puntos. Acceso el 20 de diciembre de 2023, de https://inlocrobotics.com/es/nube-de-puntos/

[8] Wikipedia. (abril 2023). Árbol octal. En Wikipedia. Acceso el 20 de diciembre de 2023, de https://es.wikipedia.org/wiki/%C3%81rbol_octal

[9] Melero Rus, F. J (2008). BP-Octree: Una estructura Jerárquica de volúmenes envolventes (Tesis doctoral). Universidad de Granada, Departamento de Lenguajes y Sistemas Informáticos. ISBN: 978-691-8356-4. D.L: GR 2828-2008.

[10] Open Robotics. (n.d.). Robot Operating System (ROS). Acceso el 15 de diciembre de 2023, de https://www.ros.org/

[11] Open Robotics. (n.d.). ROS Noetic Ninjemys. En ROS Wiki. Recuperado el 03 de enero de 2024, de https://wiki.ros.org/noetic

[12] Ubuntu. (2020). Ubuntu 20.04 LTS (Focal Fossa). Acceso el 15 de diciembre de 2023, de https://releases.ubuntu.com/focal/

[13] Open Robotics. (n.d.). Gazebo Simulator Acceso el 16 de diciembre de 2023, de https://gazebosim.org/home

[14] Open Robotics. (n.d.). RViz. En ROS Wiki. Acceso el 19 de diciembre de 2023, de https://wiki.ros.org/rviz

[15] Hornung, A., Wurm, K. M., Bennewitz, M., Stachniss, C., & Burgard, W. (n.d.). OctoMap: An Efficient Probabilistic 3D Mapping Framework Based on Octrees. Acceso el 07 de enero de 2024, de https://octomap.github.io/

[16] Clearpath Robotics. (n.d.). Husky Unmanned Ground Vehicle Robot. Acceso el 07 de enero de 2024, de https://clearpathrobotics.com/husky-unmanned-ground-vehicle-robot/

[17] Intel. (n.d.). Depth Camera D415. Acceso el 11 de enero de 2024, de https://www.intelrealsense.com/depth-camera-d415/

[18] Intel Corporation. (2023, septiembre). Product Family D400 Series Datasheet. Intel® RealSense™ Technology. Número de documento: 337029-017.

[19] Wikipedia. (septiembre 2023). Distribución uniforme discreta. En Wikipedia. Acceso el 10 de enero de 2024, de https://es.wikipedia.org/wiki/Distribuci%C3%B3n_uniforme_discreta

[20] Tinker Twins. (Mayo,2023). Husky. GitHub. Acceso el 29 de diciembre de 2023, de https://github.com/Tinker-Twins/Husky

[21] Tinker Twins. (2024, enero). Tinker Twins [Perfil de usuario]. GitHub. Acceso el 12 de enero de 2024, de https://github.com/Tinker-Twins

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

[23] Wikipedia. (2023, Diciembre, 27). Polígonos de Thiessen. En Wikipedia. Accedido el de enero de 2024, de https://es.wikipedia.org/wiki/Pol%C3%ADgonos_de_Thiessen

[24] Arzamendia López, M. E. (2018). Reactive evolutionary path planning for autonomous surface vehicles in lake environments [Disertación doctoral]. Consejo Nacional de Ciencia y Tecnología (CONACYT), Paraguay. Acceso el 12 de enero de 2024, https://repositorio.conacyt.gov.py/xmlui/handle/20.500.14066/3540?locale-attribute=en

[25] Wikipedia. (2024, enero, 06). Probabilistic roadmap. En Wikipedia. Acceso el 12 de enero de 2024, de https://en.wikipedia.org/wiki/Probabilistic_roadmap

[26] Cruz. L. (Julio 2019) ROS (Robot Operating System): Fundamentos. Medium. Acceso el 13 de enero de 2024, de https://medium.com/@robtech.impaciente/ros-robot-operating-system-fundamentos-e92478c26e02

Anexo I

Código utilizado

1. Move_node

Move_node.cpp

```
#include "evita_obstaculos/move_ros.hpp"

int main(int argc, char** argv)
{
    ros::init(argc, argv, "move");
    ros::NodeHandle n;

    move_ros programa_navegacion(n);
    programa_navegacion.run();

  return 0;
}
```

2. Move_ros

2.1 Move_ros.hpp

```
#include "ros/ros.h"
#include "geometry_msgs/Twist.h"
#include "octomap_msgs/Octomap.h"
#include "octomap_msgs/conversions.h"
#include "octomap/OcTree.h"
#include "octomap/AbstractOcTree.h"
#include "octomap/AbstractOccupancyOcTree.h"
#include "geometry_msgs/PointStamped.h"
#include "nav_msgs/Odometry.h"
#include "fstream"
#include "evita_obstaculos/detecta_obstaculo.hpp"
#include "evita_obstaculos/odometria.hpp"
#include "evita_obstaculos/utilities.hpp"
#include "evita_obstaculos/motion.hpp"
#include "evita_obstaculos/visualization.hpp"

class move_ros
{
private:

    detecta_obstaculo detecta_obstaculo_p;
    odometria odometria_p;
    utilities utilities_p;
    motion motion_p;
    visualization visualization_p;

    ros::NodeHandle n_;  //Interactuar con el sistema de ROS

    ros::Subscriber octomap_sub;
    ros::Subscriber odometry_sub;
    ros::Publisher vel_pub;
    ros::Publisher visualization_pub;

    octomap::AbstractOcTree *tree;  //Puntero a una estructura AbstractOcTree para
almacenar en estructura arbol
    octomap::OcTree *Octree; //Almacena el mapa de cuadrícula de ocupación 3D en un
OcTree

    nav_msgs::Odometry::ConstPtr odometry;

    std::vector<double> pos_actual;
    std::vector<double> orientacion_actual_q;
    double orientacion_actual_E;
    double orientacion_prueba;
```

```
double distancia_final;
char nueva_posicion_final;
char continuar;
bool bool_odom;
bool octomap_recibido;
double distancia_punto_destino;
double distancia_anterior_punto_destino;
double angulo_punto_destino;
double dif_angular;
char cont;
int interaciones_ejecucion_totales;
double radio_circunferencia;
double numPuntos_circunferecnia;
bool evaluar_recta;
bool punto_nuevo;
bool no_punto_valido;


double radio_ext;
double radio_int;
double num_puntos;
double distancia_calculo;
double tolerancia_angular;
double tolerancia_distancia;
double tolerancia_distancia_final;
octomap::point3d PuntoFIN3d;
double pos_fin_x;
double pos_fin_y;
double pos_fin_z;

std::vector<octomap::point3d> Vector_puntos_aleatorios;
std::vector<double> Vector_distancias_pto_final;
std::vector<octomap::point3d> vector_intermediatePoints;
std::vector<octomap::point3d> vector_circunferencePoints;

octomap::point3d pos_actual3d;
std::vector<double> RandomPoint;
octomap::point3d RandomPoint3d;
bool occupation;
bool ocupacion_punto_destino;
bool inter_occupation;
bool circunference_occupation;
double distancia_pto_final;
int indice_min_distancia;
octomap::point3d pos_destino3d;
geometry_msgs::Twist Comando_velocidad;
bool no_avanzar;
```

```
    visualization_msgs::Marker borrar_puntos;
    visualization_msgs::Marker punto_visualizacion;
    visualization_msgs::MarkerArray array_visualizacion;
    visualization_msgs::Marker linea_visualizacion;

     void octomapCallback(const octomap_msgs::Octomap::ConstPtr& octomap_msg);
     void odometryCallback(nav_msgs::Odometry::ConstPtr odometry_);

public:

    move_ros(ros::NodeHandle n);
    ~ move_ros();

    void run();

};
```

2.2 Move_ros.cpp

```
#include "evita_obstaculos/move_ros.hpp"

move_ros::move_ros(ros::NodeHandle n): n_(n)
{
    // INICIALIZACIONES
    pos_actual = {0, 0, 0};
    orientacion_actual_q = {0, 0, 0, 0};
    orientacion_actual_E = 0;
    orientacion_prueba = 0;
    nueva_posicion_final = {};
    bool_odom = false;
    octomap_recibido = false;
    distancia_punto_destino={};
    distancia_anterior_punto_destino=1000; //inicializar con valor alto
    angulo_punto_destino={};
    dif_angular=1;
    no_avanzar=0;
    evaluar_recta=0;
    interaciones_ejecucion_totales=0;
    punto_nuevo=0;
    no_punto_valido=0;

    // PARÁMETROS
    radio_ext = 3;      //para la generación de puntos aleatorios, distancia máxima
desde el centro del robot
    radio_int = 0.7;    //para la generación de puntos aleatorios, radio interior
central que obviamos por ser el robot (físicamente)
    num_puntos = 20;    //numero de puntos aleatorios que creamos
    distancia_calculo=0.25;  //intervalo entre un punto y otro de la recta generada
entre punto origen y punto final
    tolerancia_angular=0.01;  //valor de aproximación al angulo destino
    tolerancia_distancia=0.5;  // valor de aproximación a la distancia destino
    tolerancia_distancia_final=0.7;  // valor de aproximación a la distancia destino
    radio_circunferencia=0.5;
    numPuntos_circunferecnia=10;
    PuntoFIN3d = {0, 0, 0};  //Punto de destino

    // SUBSCRIBERS AND PUBLISHERS
    octomap_sub = n_.subscribe("/octomap_binary", 1, &move_ros::octomapCallback,
this);
    odometry_sub= n_.subscribe("/odometry/filtered", 1, &move_ros::odometryCallback,
this);
    vel_pub = n_.advertise<geometry_msgs::Twist>("/cmd_vel", 10);
    visualization_pub =
n_.advertise<visualization_msgs::Marker>("/visualization_marker", 1);
}
```

```
void move_ros::octomapCallback(const octomap_msgs::Octomap::ConstPtr& octomap_msg)
{
    octomap_recibido = true;
    tree = octomap_msgs::msgToMap(*octomap_msg); //Convierta una representación de
octomap en un nuevo octree(probabilidades completas o binario). Necesitará liberar la
memoria. Devuelve NULL en caso de error. Es del tipo AbstractOcTree
    Octree = dynamic_cast<octomap::OcTree*>(tree);  //Convertir a Octree

    detecta_obstaculo_p.set_mapa(Octree);
}

void move_ros::odometryCallback(nav_msgs::Odometry::ConstPtr odometry_)
{
    odometry = odometry_;
}

void move_ros::run ()
{
    ros::Rate rate(10);

    while(ros::ok())
    {
        ros::spinOnce();
        rate.sleep();

        std::cout << "INICIO"<< std::endl;
        interaciones_ejecucion_totales = interaciones_ejecucion_totales+1;

        inicio:
        // introducir punto destino
        if(PuntoFIN3d.x()==0 && PuntoFIN3d.y()==0 && PuntoFIN3d.z()==0)
        {
            std::cout << "Introducir punto destino: " << std::endl;
            std::cout << "Introduce posición x: ";
            std::cin >> pos_fin_x;
            std::cout <<std::endl;
            std::cout << "Introduce posición y: ";
            std::cin >> pos_fin_y;
            std::cout <<std::endl;
            pos_fin_z= 0.4;
            PuntoFIN3d = {pos_fin_x, pos_fin_y, pos_fin_z};  //Punto de destino
            punto_nuevo=1;
        }
        if (punto_nuevo==1)
        {
            do
                {
                    std::cout << "P: " << std::endl;
                    ros::spinOnce();
```

```
                    bool_odom =
odometria_p.set_puntoOdom(odometry,pos_actual,orientacion_actual_q);
                    pos_actual3d= utilities_p.VectorToPoint3d(pos_actual);
                    orientacion_actual_E =
utilities_p.CuaternionToEulerAngles(orientacion_actual_q);
                    distancia_punto_destino=
utilities_p.calculateDistance(pos_actual3d, PuntoFIN3d);
                    angulo_punto_destino =
utilities_p.calculateAngle(pos_actual3d,PuntoFIN3d);
                    dif_angular = angulo_punto_destino - orientacion_actual_E;
                    while(dif_angular>M_PI) { dif_angular = dif_angular - 2*M_PI;}
                    while(dif_angular<-M_PI) { dif_angular = dif_angular + 2*M_PI;}
                    Comando_velocidad =
motion_p.motionControlAngle(distancia_punto_destino,dif_angular);
                    vel_pub.publish(Comando_velocidad);
                    ros::Duration(0.05).sleep();
                }while (dif_angular < -tolerancia_angular || dif_angular >
tolerancia_angular);
                punto_nuevo=0;
                no_punto_valido=0;
        }

        //visualización punto destino
        punto_visualizacion= visualization_p.PointColor(PuntoFIN3d,1000, 0.54, 0.17,
0.86);
        visualization_pub.publish(punto_visualizacion);

// 1. COGER EL PUNTO DE LA UBICACIÓN ACTUAL DEL ROBOT
        ros::spinOnce();
        bool_odom =
odometria_p.set_puntoOdom(odometry,pos_actual,orientacion_actual_q);
        pos_actual3d= utilities_p.VectorToPoint3d(pos_actual);
        orientacion_actual_E=
utilities_p.CuaternionToEulerAngles(orientacion_actual_q);

        if (bool_odom==true )
        {
// 2. GENERAR PUNTOS (tipo octomap::point 3d) ALEATORIOS DENTRO DE UN RADIO
DETERMINADO
            no_avanzar = 0;
            Vector_puntos_aleatorios.clear();
            Vector_distancias_pto_final.clear();
            vector_circunferencePoints.clear();
            Vector_puntos_aleatorios.resize(num_puntos);
            Vector_distancias_pto_final.resize(num_puntos);
            vector_circunferencePoints.resize(numPuntos_circunferecnia);

            for (int iter=0; iter<num_puntos; iter++)
            {
```

```
                //GENERAR PUNTO ALEATORIO
                std::cout << "Posición actual:  "<< pos_actual[0] << " " <<
pos_actual[1] << " " << pos_actual[2] << " " << std::endl;
                std::cout << "Orientación actual: "<< orientacion_actual_E <<
std::endl;
                RandomPoint =
utilities_p.generateRandomPoint(pos_actual,orientacion_actual_E,radio_ext,radio_int);
                RandomPoint3d = utilities_p.VectorToPoint3d(RandomPoint);
                std::cout << "Punto aleatorio3d:  "<< RandomPoint3d.x() << " " <<
RandomPoint3d.y() << " " << RandomPoint3d.z() << "   ";

// 3. VER SI EL PUNTO ALEATORIO ES VÁLIDO
                //VER SI ESTE PUNTO ESTA OCUPADO
                occupation = detecta_obstaculo_p.obstaculo_cerca(RandomPoint3d);
                std::cout << "Ocupación del punto aleatorio:  "<< occupation <<
std::endl;

                //SI NO ESTÁ OCUPADO VER SI HAY ALGÚN PUNTO OCUPADO EN LA
CIRCUNFERENCIA DE ALREDEDOR
                // SI ESTÁ OCUPADO VACIAR
                if (occupation==false)
                {
                    vector_circunferencePoints.clear();
                    vector_circunferencePoints =
utilities_p.createCircunferencePoints3d(RandomPoint3d,radio_circunferencia,numPuntos_
circunferecnia);

                    for (int v1=0; v1<vector_circunferencePoints.size(); v1++)
                    {
                        //std::cout << "t4" << std::endl;
                        circunference_occupation =
detecta_obstaculo_p.obstaculo_cerca(vector_circunferencePoints[v1]);
                        if (circunference_occupation==true)
                        {
                            std::cout << "Punto ocupado en la circunferencia cercana"
<< std::endl;

                            //visualizacion de puntos (rojo)
                            punto_visualizacion=
visualization_p.PointColor(RandomPoint3d,iter,1,0,0);
                            visualization_pub.publish(punto_visualizacion);

                            Vector_puntos_aleatorios[iter] = {};
                            evaluar_recta=0;
                            break;
                        }
                        else
                        {
                            evaluar_recta=1;
```

```
                        }
                    }
                    if (evaluar_recta==1)
                    {
                        //SI NO ESTÁ OCUPADO VER SI HAY ALGÚN PUNTO OCUPADO EN LA
LÍNEA RECTA QUE LOS UNE
                        // SI ESTÁ OCUPADO VACIAR
                        //RESETEAMOS Y CREAMOS EL VECTOR DE PUNTOS INTERMEDIOS
                        vector_intermediatePoints.clear();
                        vector_intermediatePoints =
utilities_p.createVectorIntermediatePoints3d(pos_actual3d,RandomPoint3d,distancia_cal
culo);

                        // MIRAR CADA PUNTO DE ESE VECTOR PARA. SI OCUPADO VACIAMOS Y
SALIMOS DEL BUCLE SINO METEMOS EL PUNTO EN EL VECTOR
                        for (int v=0; v<vector_intermediatePoints.size(); v++)
                        {
                            inter_occupation =
detecta_obstaculo_p.obstaculo_cerca(vector_intermediatePoints[v]);
                            if (inter_occupation==true)
                            {
                                //visualizacion de puntos (rojo)
                                punto_visualizacion=
visualization_p.PointColor(RandomPoint3d,iter,1,0,0);
                                visualization_pub.publish(punto_visualizacion);

                                Vector_puntos_aleatorios[iter] = {};
                                break;
                            }
                            else
                            {
                                //visualizacion de puntos (celeste)
                                punto_visualizacion=
visualization_p.PointColor(RandomPoint3d,iter,0.70,0.70,0.70);
                                visualization_pub.publish(punto_visualizacion);

                                Vector_puntos_aleatorios[iter] = RandomPoint3d;
                            }
                        }
                    }
                }
                else
                {
                    Vector_puntos_aleatorios[iter] = {};
                    Vector_distancias_pto_final[iter] = {};

                    //visualizacion de puntos (cambio a rojo)
                    punto_visualizacion=
visualization_p.PointColor(RandomPoint3d,iter,1,0,0);
```

```
                    visualization_pub.publish(punto_visualizacion);
                }

// 4. SI EL PUNTO ALEATORIO ES VÁLIDO CALCULAR LA DISTANCIA DESDE EL PUNTO ALEATORIO
AL FINAL
                // SI EL PUNTO ALEATORIO ES VÁLIDO (NO OCUPADO Y LINEA QUE LO UNE NO
OCUPADA) CALCULAR DISTANCIA
                // SI NO, PONER 0
                if (Vector_puntos_aleatorios[iter](0) != 0 &&
Vector_puntos_aleatorios[iter](1) != 0)
                {
                    distancia_pto_final =
utilities_p.calculateDistance(RandomPoint3d,PuntoFIN3d);
                    Vector_distancias_pto_final[iter] = distancia_pto_final;
                }
                else
                {
                    Vector_distancias_pto_final[iter] = {};
                }
            ros::Duration(0.0001).sleep();
            }
            std::cout << "vector creado "<< std::endl;
                        for (int t5=0;  t5<Vector_puntos_aleatorios.size(); t5++)
                                {
                                    std::cout << "vector puntos aleatorios
resultante  " << t5 << " ";
                                    std::cout << "(" <<
Vector_puntos_aleatorios[t5].x() << ", " << Vector_puntos_aleatorios[t5].y() << ", "
<< Vector_puntos_aleatorios[t5].z() << ")   " << Vector_distancias_pto_final[t5] <<
std::endl;
                                }
            std::cout << "  "<< std::endl;

// 5. COMPARAR DISTANCIAS Y ELEGIR LA MÁS CERCANA AL PUNTO FINAL
            indice_min_distancia =
utilities_p.minimumValueIndex(Vector_distancias_pto_final);
            std::cout << "indice distancia mínima "<< indice_min_distancia <<
std::endl;

            //visualizacion de puntos (verde)
            punto_visualizacion=
visualization_p.PointColor(Vector_puntos_aleatorios[indice_min_distancia],indice_min_
distancia,0,1,0);
            visualization_pub.publish(punto_visualizacion);

            linea_visualizacion =
visualization_p.createLineMarker(pos_actual3d,Vector_puntos_aleatorios[indice_min_dis
tancia],interaciones_ejecucion_totales);
            visualization_pub.publish(linea_visualizacion);
```

```
            if (indice_min_distancia == -1)
            {
                std::cout << "NO HAY NINGÚN PUNTO VÁLIDO" << std::endl;
                bool_odom =
odometria_p.set_puntoOdom(odometry,pos_actual,orientacion_actual_q);
                orientacion_actual_E =
utilities_p.CuaternionToEulerAngles(orientacion_actual_q);
                orientacion_prueba = orientacion_actual_E + 1.5;
                do
                {
                    ros::spinOnce();
                    bool_odom =
odometria_p.set_puntoOdom(odometry,pos_actual,orientacion_actual_q);
                    orientacion_actual_E =
utilities_p.CuaternionToEulerAngles(orientacion_actual_q);
                    dif_angular = orientacion_prueba - orientacion_actual_E;
                    while(dif_angular>M_PI) { dif_angular = dif_angular - 2*M_PI;}
                    while(dif_angular<-M_PI) { dif_angular = dif_angular + 2*M_PI;}
                    Comando_velocidad =
motion_p.motionControlAngle(distancia_punto_destino,dif_angular);
                    vel_pub.publish(Comando_velocidad);
                    ros::Duration(0.05).sleep();
                }while (dif_angular < -tolerancia_angular || dif_angular >
tolerancia_angular);
                punto_nuevo=0;
                no_punto_valido=0;
                no_punto_valido=1;
                //PuntoFIN3d = {0, 0, 0};
                goto inicio;
            }

// 6. COMANDAR LA VELOCIDAD
            //CALCULO PREVIO DE DISTANCIA Y ÁNGULO AL PUNTO DESTINO (RandomPoint3d
elegido)
            pos_destino3d = Vector_puntos_aleatorios[indice_min_distancia];
            std::cout << "(" << pos_destino3d.x() << ", " << pos_destino3d.y() << ",
" << pos_destino3d.z() << ")" << std::endl;

            bool_odom =
odometria_p.set_puntoOdom(odometry,pos_actual,orientacion_actual_q);
            pos_actual3d= utilities_p.VectorToPoint3d(pos_actual);
            orientacion_actual_E =
utilities_p.CuaternionToEulerAngles(orientacion_actual_q);

            distancia_punto_destino = utilities_p.calculateDistance(pos_actual3d,
pos_destino3d);
            angulo_punto_destino =
utilities_p.calculateAngle(pos_actual3d,pos_destino3d);
```

```
            dif_angular = angulo_punto_destino - orientacion_actual_E;
            while(dif_angular>M_PI) { dif_angular = dif_angular - 2*M_PI;}
            while(dif_angular<-M_PI) { dif_angular = dif_angular + 2*M_PI;}
            std::cout << "distancia  " << distancia_punto_destino << "  angulo  " <<
angulo_punto_destino << " dif angular:  " << dif_angular << std::endl;
            std:: cout << " orient angular " << orientacion_actual_E << std::endl;
            std:: cout << " dif_angular" << dif_angular << std::endl;

            // PRIMERO COMANDO EL GIRO
            while (dif_angular < -tolerancia_angular || dif_angular >
tolerancia_angular)
            {
                std::cout << "dif_angular " << dif_angular << std::endl;
                //LEER DEL TOPIC Y CONVERTIR EL CUATERNIO A EULER
                ros::spinOnce();
                std::cout << "MOVIMIENTO ANGULO";
                std::cout << "(" << pos_destino3d.x() << ", " << pos_destino3d.y() <<
", " << pos_destino3d.z() << ")" << std::endl;
                bool_odom =
odometria_p.set_puntoOdom(odometry,pos_actual,orientacion_actual_q);
                pos_actual3d= utilities_p.VectorToPoint3d(pos_actual);
                orientacion_actual_E =
utilities_p.CuaternionToEulerAngles(orientacion_actual_q);
                std::cout << "Orientacion actual Euler: " << orientacion_actual_E <<
std::endl;

                // CALCULAR LA DISTANCIA Y EL ÁNGULO QUE DEBE GIRAR
                distancia_punto_destino= utilities_p.calculateDistance(pos_actual3d,
pos_destino3d);
                angulo_punto_destino =
utilities_p.calculateAngle(pos_actual3d,pos_destino3d);
                std::cout << "distancia  " << distancia_punto_destino << "  angulo  "
<< angulo_punto_destino << std::endl;

                // VER LA DIFERENCIA DEL ANGULO ACTUAL Y ANGULO FINAL
                dif_angular = angulo_punto_destino - orientacion_actual_E;
                while(dif_angular>M_PI) { dif_angular = dif_angular - 2*M_PI;}
                while(dif_angular<-M_PI) { dif_angular = dif_angular + 2*M_PI;}
                std::cout << "dif_angular" << dif_angular << std::endl;


                Comando_velocidad =
motion_p.motionControlAngle(distancia_punto_destino,dif_angular);
                std::cout << "Mensaje Twist " << Comando_velocidad.linear.x << "  "
<< Comando_velocidad.angular.z << std::endl;

                std::ofstream archivo_csv;
                archivo_csv.open("/home/violeta/catkin_ws/velocidad.csv",
std::ofstream::out | std::ofstream::app);
```

```
                archivo_csv << " giro: "<< Comando_velocidad.angular.z << std::endl;

                // COMANDAR LA VELOCIDAD ANGULAR
                vel_pub.publish(Comando_velocidad);

                //visualization_pub.publish(punto_visualizacion);
                ros::Duration(0.05).sleep();
            }

            std::cout << "Distancia punto destino: " << distancia_punto_destino <<
"  no_avanzar: " <<  no_avanzar <<std::endl;
            distancia_anterior_punto_destino=1000; //inicializar con un valor alto

            // DESPUES COMANDO DE AVANCE
            while (distancia_punto_destino>tolerancia_distancia && no_avanzar==0)
            {
                // UNA VEZ QUE HEMOS GIRADO VERIFICAR SI EL PUNTO SIGUE LIBRE Y SI LA
RECTA QUE LOS UNE TAMBIÉN
                occupation = detecta_obstaculo_p.obstaculo_cerca(pos_destino3d);
                std::cout << "Ocupación del punto destino:  "<< occupation <<
std::endl;

                // SI NO ESTÁ OCUPADO VER SI HAY ALGÚN PUNTO OCUPADO EN LA LÍNEA
RECTA QUE LOS UNE
                // SI ESTÁ OCUPADO ACTIVAMOS "NO AVANZAR"
                if (occupation==false)
                {
                    //RESETEAMOS Y CREAMOS LA CIRCUNFERENCIA DE ALREDEDOR DEL PUNTO
                    vector_circunferencePoints.clear();
                    vector_circunferencePoints =
utilities_p.createCircunferencePoints3d(RandomPoint3d,radio_circunferencia,numPuntos_
circunferecnia);

                    for (int v1=0; v1<vector_circunferencePoints.size(); v1++)
                    {
                        std::cout << v1 << std::endl;
                        inter_occupation =
detecta_obstaculo_p.obstaculo_cerca(vector_circunferencePoints[v1]);
                        if (inter_occupation==true)
                        {
                            no_avanzar = 1;
                            std::cout << "Hay obstáculo no avanzar circunf" <<
std::endl;
                            goto stop;
                        }
                    }
                    if (no_avanzar!=1)
                    {
                        //RESETEAMOS Y CREAMOS EL VECTOR DE PUNTOS INTERMEDIOS
```

```
                        vector_intermediatePoints.clear();
                        vector_intermediatePoints =
utilities_p.createVectorIntermediatePoints3d(pos_actual3d,pos_destino3d,distancia_cal
culo);

                        // MIRAR CADA PUNTO DE ESE VECTOR PARA. SI OCUPADO ACTIVAMOS
"NO AVANZAR".
                        for (int v=0; v<vector_intermediatePoints.size(); v++)
                        {
                            inter_occupation =
detecta_obstaculo_p.obstaculo_cerca(vector_intermediatePoints[v]);
                            if (inter_occupation==true)
                            {
                                no_avanzar = 1;
                                std::cout << "Hay obstáculo no avanzar" << std::endl;
                                goto stop;
                            }
                        }
                    }
                }
                else
                {
                    std::cout << "Hay obstáculo no avanzar" << std::endl;
                    no_avanzar = 1;
                    goto stop;
                }

                //LEER DEL TOPIC Y CONVERTIR EL PUNTO A PUNTO3D (OCTOMAP)
                ros::spinOnce();
                std::cout << "MOVIMIENTO DISTANCIA";
                std::cout << "(" << pos_destino3d.x() << ", " << pos_destino3d.y() <<
", " << pos_destino3d.z() << ")" << std::endl;
                bool_odom =
odometria_p.set_puntoOdom(odometry,pos_actual,orientacion_actual_q);
                pos_actual3d= utilities_p.VectorToPoint3d(pos_actual);
                orientacion_actual_E =
utilities_p.CuaternionToEulerAngles(orientacion_actual_q);
                std::cout << "Posición actual:  "<< pos_actual3d(0) << " " <<
pos_actual3d(1) << " " << pos_actual3d(2) << " " << std::endl;

                // CALCULAR LA DISTANCIA Y EL ÁNGULO QUE DEBE GIRAR
                distancia_punto_destino = utilities_p.calculateDistance(pos_actual3d,
pos_destino3d);
                angulo_punto_destino =
utilities_p.calculateAngle(pos_actual3d,pos_destino3d);
                dif_angular = angulo_punto_destino - orientacion_actual_E;
                std::cout << "dif_angular" << dif_angular << std::endl;
                std::cout << "distancia  " << distancia_punto_destino << "  angulo  "
<< angulo_punto_destino << std::endl;
```

```
                // COMPARAR CON LA DISTANCIA ANTERIOR PARA VERIFICAR QUE NO SE ESTÁ
ALEJANDO
                if (distancia_punto_destino > distancia_anterior_punto_destino+0.001)
                {
                    std::cout << "SE ALEJAAAA" << std::endl;
                    goto stop;
                }

                distancia_anterior_punto_destino= distancia_punto_destino;

                Comando_velocidad =
motion_p.motionControlDistance(distancia_punto_destino,dif_angular);
                std::cout << "Mensaje Twist " << Comando_velocidad.linear.x << "  "
<< Comando_velocidad.angular.z << std::endl;

                // exportación de datos al archivo csv
                std::ofstream archivo_csv;
                archivo_csv.open("/home/violeta/catkin_ws/velocidad.csv",
std::ofstream::out | std::ofstream::app);
                archivo_csv << " avance: "<< Comando_velocidad.linear.x<< std::endl;

                // COMANDAR LA VELOCIDAD LINEAL
                vel_pub.publish(Comando_velocidad);
                ros::Duration(0.05).sleep();
                //std::cout << "wait " << std::endl;
            }

            stop:
            //delete points
            borrar_puntos = visualization_p.deleteAllPoints();
            //visualization_pub.publish(borrar_puntos);

            Comando_velocidad = motion_p.motionControlSTOP();

            std::ofstream archivo_csv;
            archivo_csv.open("/home/violeta/catkin_ws/velocidad.csv",
std::ofstream::out | std::ofstream::app);
            archivo_csv << " STOP "<< std::endl;

            vel_pub.publish(Comando_velocidad);
            std::cout << "STOP " << std::endl;
            std::cout << " ------------------------------------------------ "<<
std::endl;
            //ros::Duration(20.0).sleep();

            distancia_final = utilities_p.calculateDistance(pos_actual3d,PuntoFIN3d);
            std::cout << "Disttancia hasta llegar al punto destino: "<<
distancia_final << std::endl;
```

```
            if (distancia_final<tolerancia_distancia_final)
            {
                std::cout << "Destino final alcanzado " << std::endl;
                nueva_posicion_final = {};
                std::cout << "elección " << nueva_posicion_final << std::endl;

                while (nueva_posicion_final!= 's' && nueva_posicion_final!= 'n')
                {
                    std::cout << "¿Desea introducir otro punto destino? [s/n] " <<
std::endl;
                    std::cin >> nueva_posicion_final;
                    std::cout << "elección " << nueva_posicion_final << std::endl;
                }
                if (nueva_posicion_final == 's')
                {
                    PuntoFIN3d = {0, 0, 0};  //Punto de destino
                }
                else if (nueva_posicion_final == 'n')
                {
                    break;
                }
            }
        }
        else {std::cout << "Odometry es un puntero nulo." << std::endl;}
    }
}


move_ros::~move_ros(){}
```

3. Odometría

3.1 Odometria.hpp

```
#include "vector"
#include "nav_msgs/Odometry.h"
#include "octomap_msgs/Octomap.h"
#include "octomap_msgs/conversions.h"

class odometria
{
private:

    double x_odom;
    double y_odom;
    double z_odom;
    double qx_odom;
    double qy_odom;
    double qz_odom;
    double qw_odom;

    double linear;
    double angular;

public:
    odometria();
    ~ odometria();

    bool set_puntoOdom(nav_msgs::Odometry::ConstPtr odometry, std::vector<double>&
posicion, std::vector<double>& orientacion);
    void set_velOdom(nav_msgs::Odometry::ConstPtr odometry, std::vector<double>&
velocidad);
};
```

3.2 Odometria.cpp

```
#include "evita_obstaculos/odometria.hpp"

odometria::odometria()
{
    x_odom = 0;
    y_odom = 0;
    z_odom = 0;
    qx_odom = 0;
    qy_odom = 0;
    qz_odom = 0;
    qw_odom = 0;
```

```
    linear = 0;
    angular = 0;

}

bool odometria::set_puntoOdom(nav_msgs::Odometry::ConstPtr odometry,
std::vector<double>& posicion, std::vector<double>& orientacion)
{
    if (odometry)
    {
        // Extraer posición del mensaje de la Odometría
        x_odom = odometry->pose.pose.position.x;
        y_odom = odometry->pose.pose.position.y;
        z_odom = odometry->pose.pose.position.z;
        qx_odom = odometry->pose.pose.orientation.x;
        qy_odom = odometry->pose.pose.orientation.y;
        qz_odom = odometry->pose.pose.orientation.z;
        qw_odom = odometry->pose.pose.orientation.w;

        posicion = {x_odom, y_odom, z_odom};
        orientacion = {qx_odom, qy_odom, qz_odom, qw_odom};

       return true;
    }
    else
    {
        return false;
    }
}

void odometria::set_velOdom(nav_msgs::Odometry::ConstPtr odometry,
std::vector<double>& velocidad)
{
    std::cout << "set_velOdom" << std::endl;
    // Extraer velocidad del mensaje de la Odometría
    linear = odometry->twist.twist.linear.x;
    angular= odometry->twist.twist.angular.z;

    velocidad= {linear, angular};
    std::cout << "set_velOdom_2   " << velocidad[1] << "   " << velocidad[0] <<
std::endl;
}

odometria::~odometria(){}
```

4. Detecta obstáculo

4.1 detecta_obstaculo.hpp

```
#include "iostream"
#include "octomap_msgs/Octomap.h"
#include "octomap_msgs/conversions.h"
#include "octomap/OcTree.h"
#include "octomap/AbstractOcTree.h"
#include "octomap/AbstractOccupancyOcTree.h"

class detecta_obstaculo
{
private:
    octomap::AbstractOcTree *tree;  //Puntero a una estructura AbstractOcTree para
almacenar en estructura arbol
    octomap::OcTree *Octree; //Almacena el mapa de cuadrícula de ocupación 3D en un
OcTree

    bool octomap_recibido;

public:
    detecta_obstaculo();
    ~ detecta_obstaculo();

    void set_mapa(octomap::OcTree *Octree_);
    bool obstaculo_cerca(const octomap::point3d location);
    octomap::OcTree* get_map();
};
```

4.2 detecta_obstaculo.cpp

```
#include "evita_obstaculos/detecta_obstaculo.hpp"

detecta_obstaculo::detecta_obstaculo()
{
    octomap_recibido = false;  //si hemos recibido mensaje de octomap
}

void detecta_obstaculo::set_mapa (octomap::OcTree *Octree_)
{
    octomap_recibido = true;
    Octree = Octree_;
}

// VERIFICAR SI HAY UN OBTÁCULO EN UN PUNTO ESPECÍFICO (tipo octomap point3d)
bool detecta_obstaculo::obstaculo_cerca(const octomap::point3d location)
{

    if (!Octree)
    {
    std::cout << "Mapa OctoMap no disponible." << std::endl;
    return false;
    }

    octomap::OcTreeNode* result = Octree->search(location);

    if (result && Octree->isNodeOccupied(result))
    {
        //std::cout << "Hay un obstáculo     " << std::endl;
        return true;
    }
    else
    {
        //std::cout << "No hay obstáculo    " <<std::endl;
        return false;
    }
}


detecta_obstaculo::~detecta_obstaculo(){}

octomap::OcTree* detecta_obstaculo::get_map(){
    return Octree;
}
```

5. Utilities

5.1 utilities.hpp

```
#include "iostream"
#include "cmath"
#include "cstdlib"
#include "ctime"
#include "vector"
#include "octomap_msgs/Octomap.h"
#include "octomap_msgs/conversions.h"
#include "octomap/OcTree.h"

class utilities
{
private:
    std::vector<double> vector;
    octomap::point3d octoPoint3d;

    double min_theta;
    double max_theta;
    double theta;
    double distance;
    double x_centro;
    double y_centro;
    double z_centro;
    std::vector<double> randomPoint;
    double diferenciaX;
    double diferenciaY;
    double diferenciaZ;
    double distancia;
    std::vector<double> intermediatePoint;
    octomap::point3d intermediatePoint3d;
    std::vector<octomap::point3d> intermediateVectorPoints3d;
    octomap::point3d puntoInicio3d;
    octomap::point3d puntoFin3d;
    double distanciaTotal;
    int cantidadPuntos;
    double angulo_circunferencia;
    octomap::point3d punto_circunferencia;
    std::vector<octomap::point3d> vector_puntos_circunferencia;
    std::vector<double> vectorDistances;
    double minimumDistance;
    int minimumIndex;
    octomap::point3d FinalPoint3d;
    octomap::point3d StartPoint3d;
    double deltaX;
    double deltaY;
```

```
    double hipotenusa;
    double angle;
    double Euler_Angle;
    double qx;
    double qy;
    double qz;
    double qw;

public:
    utilities();
    ~ utilities();

    octomap::point3d VectorToPoint3d (std::vector<double> vector);
    std::vector<double> Point3dToVector(octomap::point3d octoPoint3d);
    std::vector<double> generateRandomPoint(std::vector<double> centro, double
orientacion, double radio_ext, double radio_int);
    double calculateDistance(octomap::point3d puntoA, octomap::point3d puntoB);
    std::vector<octomap::point3d> createVectorIntermediatePoints3d(octomap::point3d
puntoInicio3d, octomap::point3d puntoFin3d, double incremento);
    std::vector<octomap::point3d> createCircunferencePoints3d(octomap::point3d
centro, double radio, int numPuntos);
    int minimumValueIndex(std::vector<double> vectorDistances);
    double calculateAngle(octomap::point3d StartPoint3d, octomap::point3d
FinalPoint3d);
    double CuaternionToEulerAngles (std::vector<double> cuaternion_orientation);

};
```

5.2 utilities.cpp

```
#include "evita_obstaculos/utilities.hpp"

utilities::utilities()
{
};
// DE VECTOR A UNTO 3D
octomap::point3d utilities::VectorToPoint3d (std::vector<double> vector){
    octoPoint3d = {vector[0],vector[1],vector[2]};
    return octoPoint3d;
}

// DE PUNTO 3D A VECTOR
std::vector<double> utilities::Point3dToVector(octomap::point3d octoPoint3d){
    vector = {octoPoint3d(0), octoPoint3d(1), octoPoint3d(2)};
    return vector;
}

// GENERAR PUNTO ALEATORIO 2D (con offset z=0.4)
```

```
std::vector<double> utilities::generateRandomPoint(std::vector<double> centro, double
orientacion, double radio_ext, double radio_int){
                        //std::cout << "rand" << rand() << std::endl;
    min_theta = orientacion - (1*M_PI/3);
    max_theta = orientacion + (1*M_PI/3);

    theta = min_theta + (static_cast<double>(rand()) / RAND_MAX) * (max_theta -
min_theta); //modifico para que solo sean los 180º frontales
    distance = sqrt(static_cast<double>(rand()) / RAND_MAX) * (radio_ext-
radio_int)+radio_int;  //modifico para que excluya el circulo unidad central que es
el propio robot

    x_centro = centro[0] + distance * cos(theta);
    y_centro = centro[1] + distance * sin(theta);

    randomPoint = {x_centro,y_centro,0.4};
    return randomPoint;
}

// CALCULAR DISTANCIA
double utilities::calculateDistance(octomap::point3d puntoA, octomap::point3d
puntoB){

    diferenciaX = std::abs(puntoA(0) - puntoB(0));
    diferenciaY = std::abs(puntoA(1)- puntoB(1));
    diferenciaZ = std::abs(puntoA(2) - puntoB(2));

    distancia = sqrt(diferenciaX*diferenciaX + diferenciaY*diferenciaY +
diferenciaZ*diferenciaZ);

    return distancia;
}

// CALCULO DE VECTOR DE PUNTOS QUE FORMA LA RECTA QUE UNE DOS PUNTOS
std::vector<octomap::point3d>
utilities::createVectorIntermediatePoints3d(octomap::point3d puntoInicio3d,
octomap::point3d puntoFin3d, double incremento){

    distanciaTotal = calculateDistance(puntoInicio3d,puntoFin3d);
    cantidadPuntos = static_cast<int>(distanciaTotal/incremento);
    intermediateVectorPoints3d.clear();
    for(int i = 0; i<=cantidadPuntos; i++)
    {
        double t = static_cast<double>(i)/cantidadPuntos;
        intermediatePoint3d(0)= (puntoInicio3d(0)+ t*(puntoFin3d(0)-
puntoInicio3d(0)))+0.5;
        intermediatePoint3d(1)= puntoInicio3d(1)+ t*(puntoFin3d(1)-puntoInicio3d(1));
        intermediateVectorPoints3d.push_back(intermediatePoint3d);
```

```
    }
    for(int i = 0; i<=cantidadPuntos; i++)
    {
        double t = static_cast<double>(i)/cantidadPuntos;
        intermediatePoint3d(0)= puntoInicio3d(0)+ t*(puntoFin3d(0)-puntoInicio3d(0));
        intermediatePoint3d(1)= puntoInicio3d(1)+ t*(puntoFin3d(1)-puntoInicio3d(1));
        intermediateVectorPoints3d.push_back(intermediatePoint3d);

    }
    for(int i = 0; i<=cantidadPuntos; i++)
    {
        double t = static_cast<double>(i)/cantidadPuntos;
        intermediatePoint3d(0)= (puntoInicio3d(0)+ t*(puntoFin3d(0)-
puntoInicio3d(0)))-0.5;
        intermediatePoint3d(1)= puntoInicio3d(1)+ t*(puntoFin3d(1)-puntoInicio3d(1));
        intermediateVectorPoints3d.push_back(intermediatePoint3d);

    }
    return intermediateVectorPoints3d;
}

// CÁLCULO DE CIRCUNFERENCIA ALREDEDOR DEL PUNTO ELEGIDO
std::vector<octomap::point3d> utilities::createCircunferencePoints3d(octomap::point3d
centro, double radio, int numPuntos){
    vector_puntos_circunferencia.clear();
    for (int i = 0; i<numPuntos; i++)
    {
        angulo_circunferencia = 2*M_PI*i/numPuntos;
        punto_circunferencia(0) = centro(0) + radio * cos(angulo_circunferencia);
        punto_circunferencia(1) = centro(1) + radio * sin(angulo_circunferencia);
        vector_puntos_circunferencia.push_back(punto_circunferencia);
    }
    return vector_puntos_circunferencia;
}
// BUSCAR EL VALOR MÍNIMO DE UN VECTOR Y DEVOLVER EL ÍNDICE
int utilities::minimumValueIndex(std::vector<double> vectorDistances){
    if (vectorDistances.empty())
    {
        std::cout << "Vector distancias is empty" << std::endl;
        return -1;
    }
    minimumDistance = -1;
    minimumIndex = -1;

    // iniciar el valor de comparación con el primer valor no nulo del vector
    for (int t=0; t<vectorDistances.size(); ++t)
    {
        if (vectorDistances[t] != 0)
        {
```

```
            minimumDistance = vectorDistances[t];
            minimumIndex = t;
            break;
        }
    }

    // si una vez buscado el primer valor no nulo el valor de mínima distancia sigue
siendo 0 es porque no hay ningun punto válido
    if (minimumDistance == 0)
    {
        return -1;
    }
    else
    {
        for (int i=1; i<vectorDistances.size(); ++i)
        {
            if (vectorDistances[i] < minimumDistance && vectorDistances[i] != 0)
            {
                minimumDistance = vectorDistances[i];
                minimumIndex = i;
            }
        }
        return minimumIndex;
    }
}

// CALCULO DEL ANGULO AL PUNTO FINAL (para comandar velocidad)
double utilities::calculateAngle (octomap::point3d StartPoint3d, octomap::point3d
FinalPoint3d){
    deltaX = (FinalPoint3d(0)-StartPoint3d(0));
    deltaY = (FinalPoint3d(1)-StartPoint3d(1));
    angle = atan2(deltaY,deltaX);

    return angle;
}

// PASAR DE CUATERNIO A ANGULO DE EULER
double utilities::CuaternionToEulerAngles (std::vector<double>
cuaternion_orientation){
    qx = cuaternion_orientation[0];
    qy = cuaternion_orientation[1];
    qz = cuaternion_orientation[2];
    qw = cuaternion_orientation[3];

    Euler_Angle = atan2(2*(qw*qz+qx*qy),1-2*(qy*qy+qz*qz));

    return Euler_Angle;
}
utilities::~utilities(){}
```

6. Motion

6.1 motion.hpp

```
#include "iostream"
#include "geometry_msgs/Twist.h"

class motion
{
private:

    geometry_msgs::Twist twist_control;

public:
    motion();
    ~motion();

    geometry_msgs::Twist motionControlAngle (double distance, double angle);
    geometry_msgs::Twist motionControlDistance (double distance, double angle);
    geometry_msgs::Twist motionControlSTOP ();

};
```

6.2 motion.cpp

```
#include "evita_obstaculos/motion.hpp"

motion::motion()
{
};

geometry_msgs::Twist motion::motionControlAngle (double distance, double angle){
    twist_control.linear.x = 0.02;
    if (angle > 0)
    {
        twist_control.angular.z = 0.3;;
    }
    else
    {
        twist_control.angular.z = -0.3;
    }


    return twist_control;
}

geometry_msgs::Twist motion::motionControlDistance (double distance, double angle){
    twist_control.linear.x = 0.5;
    twist_control.angular.z = 0;

    return twist_control;
}

geometry_msgs::Twist motion::motionControlSTOP ( )
{
    twist_control.linear.x = 0;
    twist_control.angular.z = 0;

    return twist_control;
}

motion::~motion(){}
```

7. Visualization

7.1 visualization.hpp

```
#include "iostream"
#include "visualization_msgs/Marker.h"
#include "visualization_msgs/MarkerArray.h"
#include "octomap_msgs/Octomap.h"
#include "octomap_msgs/conversions.h"
#include "octomap/OcTree.h"

class visualization
{
private:
    visualization_msgs::MarkerArray vector_visualizacion;
    visualization_msgs::Marker punto_visualizacion;

public:
    visualization();
    ~visualization();

    visualization_msgs::Marker PointColor(octomap::point3d point, int iter, float r,
float g, float b);
    visualization_msgs::Marker createLineMarker(octomap::point3d start,
octomap::point3d end,int iteracion);
    visualization_msgs::Marker deleteAllPoints();
};
```

7.2 visualization.cpp

```
#include "evita_obstaculos/visualization.hpp"

visualization::visualization()
{
};

visualization_msgs::Marker visualization::PointColor(octomap::point3d point, int
iter, float r, float g, float b)
{
    visualization_msgs::Marker marker;
    marker.header.frame_id = "map";
    marker.header.stamp = ros::Time::now();
    marker.ns = "punto_aleatorio";
    marker.id = iter;
    marker.type = visualization_msgs::Marker::SPHERE;
    marker.action = visualization_msgs::Marker::ADD;

    marker.pose.position.x = point.x();
    marker.pose.position.y = point.y();
    marker.pose.position.z = point.z();
    marker.pose.orientation.x = 0.0;
    marker.pose.orientation.y = 0.0;
    marker.pose.orientation.z = 0.0;
    marker.pose.orientation.w = 1.0;

    marker.scale.x = 0.2;
    marker.scale.y = 0.2;
    marker.scale.z = 0.2;

    // Establece el nuevo color
    marker.color.r = r;
    marker.color.g = g;
    marker.color.b = b;
    marker.color.a = 1.0; // No transparente

    return marker;
}

visualization_msgs::Marker visualization::createLineMarker(octomap::point3d start,
octomap::point3d end, int iteracion)
{
    visualization_msgs::Marker line_marker;
    line_marker.header.frame_id = "map";
    line_marker.header.stamp = ros::Time::now();
    line_marker.ns = "line";
    line_marker.id = iteracion;
    line_marker.type = visualization_msgs::Marker::LINE_STRIP;
```

```
    line_marker.action = visualization_msgs::Marker::ADD;

    // Escala
    line_marker.scale.x = 0.05;  // Ancho de la línea

    // Color
    line_marker.color.r = 0.0;
    line_marker.color.g = 0.0;
    line_marker.color.b = 0.0;
    line_marker.color.a = 1.0;

    // Puntos de inicio y fin
    geometry_msgs::Point start_point;
    start_point.x = start.x();
    start_point.y = start.y();
    start_point.z = start.z();
    line_marker.points.push_back(start_point);

    geometry_msgs::Point end_point;
    end_point.x = end.x();
    end_point.y = end.y();
    end_point.z = end.z();
    line_marker.points.push_back(end_point);

    return line_marker;
}

visualization_msgs::Marker visualization::deleteAllPoints()
{
    visualization_msgs::Marker marker;
    marker.header.frame_id = "map";
    marker.header.stamp = ros::Time::now();
    marker.ns = "punto_aleatorio";
    marker.action = visualization_msgs::Marker::DELETEALL;

    return marker;
}


visualization::~visualization(){}
```

8. CMakeLists.txt

```
cmake_minimum_required(VERSION 3.0.2)
project(evita_obstaculos)

## Compile as C++11, supported in ROS Kinetic and newer
# add_compile_options(-std=c++11)

## Find catkin macros and libraries
## if COMPONENTS list like find_package(catkin REQUIRED COMPONENTS xyz)
## is used, also find other catkin packages
find_package(octomap REQUIRED)
find_package(catkin REQUIRED COMPONENTS
  roscpp
  rospy
  std_msgs
  octomap_ros
  octomap_msgs
)

## System dependencies are found with CMake's conventions
# find_package(Boost REQUIRED COMPONENTS system)


## Uncomment this if the package has a setup.py. This macro ensures
## modules and global scripts declared therein get installed
## See http://ros.org/doc/api/catkin/html/user_guide/setup_dot_py.html
# catkin_python_setup()

################################################
## Declare ROS messages, services and actions ##
################################################

## To declare and build messages, services or actions from within this
## package, follow these steps:
## * Let MSG_DEP_SET be the set of packages whose message types you use in
##   your messages/services/actions (e.g. std_msgs, actionlib_msgs, ...).
## * In the file package.xml:
##   * add a build_depend tag for "message_generation"
##   * add a build_depend and a exec_depend tag for each package in MSG_DEP_SET
##   * If MSG_DEP_SET isn't empty the following dependency has been pulled in
##     but can be declared for certainty nonetheless:
##     * add a exec_depend tag for "message_runtime"
## * In this file (CMakeLists.txt):
##   * add "message_generation" and every package in MSG_DEP_SET to
##     find_package(catkin REQUIRED COMPONENTS ...)
##   * add "message_runtime" and every package in MSG_DEP_SET to
##     catkin_package(CATKIN_DEPENDS ...)
```

```
##   * uncomment the add_*_files sections below as needed
##     and list every .msg/.srv/.action file to be processed
##   * uncomment the generate_messages entry below
##   * add every package in MSG_DEP_SET to generate_messages(DEPENDENCIES ...)

## Generate messages in the 'msg' folder
# add_message_files(
#   FILES
#   Message1.msg
#   Message2.msg
# )

## Generate services in the 'srv' folder
# add_service_files(
#   FILES
#   Service1.srv
#   Service2.srv
# )

## Generate actions in the 'action' folder
# add_action_files(
#   FILES
#   Action1.action
#   Action2.action
# )

## Generate added messages and services with any dependencies listed here
# generate_messages(
#   DEPENDENCIES
#   std_msgs
# )

################################################
## Declare ROS dynamic reconfigure parameters ##
################################################

## To declare and build dynamic reconfigure parameters within this
## package, follow these steps:
## * In the file package.xml:
##   * add a build_depend and a exec_depend tag for "dynamic_reconfigure"
## * In this file (CMakeLists.txt):
##   * add "dynamic_reconfigure" to
##     find_package(catkin REQUIRED COMPONENTS ...)
##   * uncomment the "generate_dynamic_reconfigure_options" section below
##     and list every .cfg file to be processed

## Generate dynamic reconfigure parameters in the 'cfg' folder
# generate_dynamic_reconfigure_options(
#   cfg/DynReconf1.cfg
```

```
#   cfg/DynReconf2.cfg
# )

####################################
## catkin specific configuration ##
####################################
## The catkin_package macro generates cmake config files for your package
## Declare things to be passed to dependent projects
## INCLUDE_DIRS: uncomment this if your package contains header files
## LIBRARIES: libraries you create in this project that dependent projects also need
## CATKIN_DEPENDS: catkin_packages dependent projects also need
## DEPENDS: system dependencies of this project that dependent projects also need
catkin_package(
#  INCLUDE_DIRS include
#  LIBRARIES evita_obstaculos
  CATKIN_DEPENDS roscpp rospy std_msgs octomap_msgs octomap_ros
#  DEPENDS system_lib
)

###########
## Build ##
###########

## Specify additional locations of header files
## Your package locations should be listed before other locations
include_directories(
  include
  ${PROJECT_SOURCE_DIR}/include
  ${catkin_INCLUDE_DIRS}
  ${OCTOMAP_INCLUDE_DIRS}

)


add_library(mi_libreria_octomap src/move_ros.cpp src/detecta_obstaculo.cpp
src/odometria.cpp src/utilities.cpp src/motion.cpp src/visualization.cpp)
add_dependencies(mi_libreria_octomap ${${PROJECT_NAME}_EXPORTED_TARGETS}
${catkin_EXPORTED_TARGETS})
target_link_libraries(mi_libreria_octomap ${catkin_LIBRARIES})

#add_library(mi_libreria_odometria src/move_ros.cpp src/odometria.cpp)
#add_dependencies(mi_libreria_odometria ${${PROJECT_NAME}_EXPORTED_TARGETS}
${catkin_EXPORTED_TARGETS})
#target_link_libraries(mi_libreria_odometria ${catkin_LIBRARIES})


add_executable(move_node src/move_node.cpp)
add_dependencies(move_node ${${PROJECT_NAME}_EXPORTED_TARGETS}
${catkin_EXPORTED_TARGETS})
```

```
target_link_libraries(move_node ${catkin_LIBRARIES} mi_libreria_octomap )


# ## Declare a C++ library

# ##add_library(${PROJECT_NAME}
# ## src/${PROJECT_NAME}/evita_obstaculos.cpp
# ## )

# ## Add cmake target dependencies of the library
# ## as an example, code may need to be generated before libraries
# ## either from message generation or dynamic reconfigure
# # add_dependencies(${PROJECT_NAME} ${${PROJECT_NAME}_EXPORTED_TARGETS}
${catkin_EXPORTED_TARGETS})
# ## add_dependencies (move )

# ## Declare a C++ executable
# ## With catkin_make all packages are built within a single CMake context
# ## The recommended prefix ensures that target names across packages don't collide
# add_executable(move src/move.cpp)

# ## Rename C++ executable without prefix
# ## The above recommended prefix causes long target names, the following renames the
# ## target back to the shorter version for ease of user use
# ## e.g. "rosrun someones_pkg node" instead of "rosrun someones_pkg
someones_pkg_node"
# # set_target_properties(${PROJECT_NAME}_node PROPERTIES OUTPUT_NAME node PREFIX "")

# ## Add cmake target dependencies of the executable
# ## same as for the library above
#  add_dependencies(${PROJECT_NAME}_node ${${PROJECT_NAME}_EXPORTED_TARGETS}
${catkin_EXPORTED_TARGETS})
#  add_dependencies(move ${${PROJECT_NAME}_EXPORTED_TARGETS}
${catkin_EXPORTED_TARGETS})

# ## Specify libraries to link a library or executable target against
# target_link_libraries(move
#   ${catkin_LIBRARIES}
#   ${OCTOMAP_LIBRARIES}
#    )

# #############
# ## Install ##
# #############

# # all install targets should use catkin DESTINATION variables
# # See http://ros.org/doc/api/catkin/html/adv_user_guide/variables.html

# ## Mark executable scripts (Python etc.) for installation
```

```
# ## in contrast to setup.py, you can choose the destination
# # catkin_install_python(PROGRAMS
# #   scripts/my_python_script
# #   DESTINATION ${CATKIN_PACKAGE_BIN_DESTINATION}
# # )

# ## Mark executables for installation
# ## See
http://docs.ros.org/melodic/api/catkin/html/howto/format1/building_executables.html
# # install(TARGETS ${PROJECT_NAME}_node
# #   RUNTIME DESTINATION ${CATKIN_PACKAGE_BIN_DESTINATION}
# # )

# ## Mark libraries for installation
# ## See
http://docs.ros.org/melodic/api/catkin/html/howto/format1/building_libraries.html
# # install(TARGETS ${PROJECT_NAME}
# #   ARCHIVE DESTINATION ${CATKIN_PACKAGE_LIB_DESTINATION}
# #   LIBRARY DESTINATION ${CATKIN_PACKAGE_LIB_DESTINATION}
# #   RUNTIME DESTINATION ${CATKIN_GLOBAL_BIN_DESTINATION}
# # )

# ## Mark cpp header files for installation
# # install(DIRECTORY include/${PROJECT_NAME}/
# #   DESTINATION ${CATKIN_PACKAGE_INCLUDE_DESTINATION}
# #   FILES_MATCHING PATTERN "*.h"
# #   PATTERN ".svn" EXCLUDE
# # )

# ## Mark other files for installation (e.g. launch and bag files, etc.)
# # install(FILES
# #   # myfile1
# #   # myfile2
# #   DESTINATION ${CATKIN_PACKAGE_SHARE_DESTINATION}
# # )

# #############
# ## Testing ##
# #############

# ## Add gtest based cpp test target and link libraries
# # catkin_add_gtest(${PROJECT_NAME}-test test/test_evita_obstaculos.cpp)
# # if(TARGET ${PROJECT_NAME}-test)
# #   target_link_libraries(${PROJECT_NAME}-test ${PROJECT_NAME})
# # endif()

# ## Add folders to be run by python nosetests
# # catkin_add_nosetests(test)
# */
```

9. package.xml

```
<?xml version="1.0"?>
<package format="2">
  <name>evita_obstaculos</name>
  <version>0.0.0</version>
  <description>The evita_obstaculos package</description>

  <!-- One maintainer tag required, multiple allowed, one person per tag -->
  <!-- Example:  -->
  <!-- <maintainer email="jane.doe@example.com">Jane Doe</maintainer> -->
  <maintainer email="violeta@todo.todo">violeta</maintainer>


  <!-- One license tag required, multiple allowed, one license per tag -->
  <!-- Commonly used license strings: -->
  <!--   BSD, MIT, Boost Software License, GPLv2, GPLv3, LGPLv2.1, LGPLv3 -->
  <license>TODO</license>


  <!-- Url tags are optional, but multiple are allowed, one per tag -->
  <!-- Optional attribute type can be: website, bugtracker, or repository -->
  <!-- Example: -->
  <!-- <url type="website">http://wiki.ros.org/evita_obstaculos</url> -->


  <!-- Author tags are optional, multiple are allowed, one per tag -->
  <!-- Authors do not have to be maintainers, but could be -->
  <!-- Example: -->
  <!-- <author email="jane.doe@example.com">Jane Doe</author> -->


  <!-- The *depend tags are used to specify dependencies -->
  <!-- Dependencies can be catkin packages or system dependencies -->
  <!-- Examples: -->
  <!-- Use depend as a shortcut for packages that are both build and exec
dependencies -->
  <!--   <depend>roscpp</depend> -->
  <!--   Note that this is equivalent to the following: -->
  <!--   <build_depend>roscpp</build_depend> -->
  <!--   <exec_depend>roscpp</exec_depend> -->
  <!-- Use build_depend for packages you need at compile time: -->
  <!--   <build_depend>message_generation</build_depend> -->
  <!-- Use build_export_depend for packages you need in order to build against this
package: -->
  <!--   <build_export_depend>message_generation</build_export_depend> -->
  <!-- Use buildtool_depend for build tool packages: -->
  <!--   <buildtool_depend>catkin</buildtool_depend> -->
  <!-- Use exec_depend for packages you need at runtime: -->
```

```
  <!--   <exec_depend>message_runtime</exec_depend> -->
  <!-- Use test_depend for packages you need only for testing: -->
  <!--   <test_depend>gtest</test_depend> -->
  <!-- Use doc_depend for packages you need only for building documentation: -->
  <!--   <doc_depend>doxygen</doc_depend> -->
  <buildtool_depend>catkin</buildtool_depend>

  <build_depend>roscpp</build_depend>
  <build_depend>rospy</build_depend>
  <build_depend>std_msgs</build_depend>
  <build_depend>octomap_msgs</build_depend>
  <build_depend>octomap_ros</build_depend>


  <build_export_depend>roscpp</build_export_depend>
  <build_export_depend>rospy</build_export_depend>
  <build_export_depend>std_msgs</build_export_depend>
  <build_export_depend>octomap_msgs</build_export_depend>
  <build_export_depend>octomap_ros</build_export_depend>


  <exec_depend>roscpp</exec_depend>
  <exec_depend>rospy</exec_depend>
  <exec_depend>std_msgs</exec_depend>
  <exec_depend>octomap_msgs</exec_depend>
  <exec_depend>octomap_ros</exec_depend>

  <!-- The export tag contains other, unspecified, tags -->

  <export>
    <!-- Other tools can request additional information be placed here -->

  </export>
</package>
```

10. Husky.urdf

```
<?xml version="1.0"?>
<!--
Software License Agreement (BSD)

\file      husky.urdf.xacro
\authors   Paul Bovbel <pbovbel@clearpathrobotics.com>, Devon Ash
<dash@clearpathrobotics.com>
\copyright Copyright (c) 2015, Clearpath Robotics, Inc., All rights reserved.

Redistribution and use in source and binary forms, with or without modification, are
permitted provided that
the following conditions are met:
 * Redistributions of source code must retain the above copyright notice, this list
of conditions and the
   following disclaimer.
 * Redistributions in binary form must reproduce the above copyright notice, this
list of conditions and the
   following disclaimer in the documentation and/or other materials provided with the
distribution.
 * Neither the name of Clearpath Robotics nor the names of its contributors may be
used to endorse or promote
   products derived from this software without specific prior written permission.

THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS" AND ANY
EXPRESS OR IMPLIED WAR-
RANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND
FITNESS FOR A PARTICULAR
PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT HOLDER OR CONTRIBUTORS BE
LIABLE FOR ANY DIRECT, IN-
DIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT
LIMITED TO, PROCUREMENT
OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS
INTERRUPTION) HOWEVER CAUSED AND
ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING
NEGLIGENCE OR OTHERWISE)
ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE
POSSIBILITY OF SUCH DAMAGE.
-->
<robot name="husky" xmlns:xacro="http://ros.org/wiki/xacro">
  <!-- IMU Link -->
  <xacro:arg name="imu_xyz"     default="$(optenv HUSKY_IMU_XYZ 0.19 0 0.149)"/>
  <xacro:arg name="imu_rpy"     default="$(optenv HUSKY_IMU_RPY 0 -1.5708 3.1416)"/>
  <xacro:arg name="imu_parent"  default="$(optenv HUSKY_IMU_PARENT base_link)"/>

  <!-- LMS1XX Laser Primary and Secondary -->
```

```
  <xacro:arg name="laser_enabled"           default="$(optenv HUSKY_LMS1XX_ENABLED
0)" />
  <xacro:arg name="laser_topic"             default="$(optenv HUSKY_LMS1XX_TOPIC
front/scan)"/>
  <xacro:arg name="laser_prefix"            default="$(optenv HUSKY_LMS1XX_PREFIX
front)"/>
  <xacro:arg name="laser_parent"            default="$(optenv HUSKY_LMS1XX_PARENT
top_plate_link)"/>
  <xacro:arg name="laser_xyz"               default="$(optenv HUSKY_LMS1XX_XYZ 0.2206
0.0 0.00635)" />
  <xacro:arg name="laser_rpy"               default="$(optenv HUSKY_LMS1XX_RPY 0.0
0.0 0.0)" />

  <xacro:arg name="laser_secondary_enabled" default="$(optenv
HUSKY_LMS1XX_SECONDARY_ENABLED 0)" />
  <xacro:arg name="laser_secondary_topic"   default="$(optenv
HUSKY_LMS1XX_SECONDARY_TOPIC rear/scan)"/>
  <xacro:arg name="laser_secondary_prefix"  default="$(optenv
HUSKY_LMS1XX_SECONDARY_PREFIX rear)"/>
  <xacro:arg name="laser_secondary_parent"  default="$(optenv
HUSKY_LMS1XX_SECONDARY_PARENT top_plate_link)"/>
  <xacro:arg name="laser_secondary_xyz"     default="$(optenv
HUSKY_LMS1XX_SECONDARY_XYZ -0.2206 0.0 0.00635)" />
  <xacro:arg name="laser_secondary_rpy"     default="$(optenv
HUSKY_LMS1XX_SECONDARY_RPY 0.0 0.0 3.14159)" />

  <!-- UST10 Laser Primary  and Secondary -->
  <xacro:arg name="laser_ust10_front_enabled" default="$(optenv HUSKY_UST10_ENABLED
1)" />
  <xacro:arg name="laser_ust10_front_topic"   default="$(optenv HUSKY_UST10_TOPIC
/scan)" />
  <xacro:arg name="laser_ust10_front_prefix"  default="$(optenv HUSKY_UST10_PREFIX
front)" />
  <xacro:arg name="laser_ust10_front_parent"  default="$(optenv HUSKY_UST10_PARENT
top_plate_link)" />
  <xacro:arg name="laser_ust10_front_xyz"     default="$(optenv HUSKY_UST10_XYZ
0.2206 0.0 0.00635)" />
  <xacro:arg name="laser_ust10_front_rpy"     default="$(optenv HUSKY_UST10_RPY 0 0
0)" />

  <xacro:arg name="laser_ust10_rear_enabled"  default="$(optenv
HUSKY_UST10_SECONDARY_ENABLED 0)" />
  <xacro:arg name="laser_ust10_rear_topic"    default="$(optenv
HUSKY_UST10_SECONDARY_TOPIC rear/scan)" />
  <xacro:arg name="laser_ust10_rear_prefix"   default="$(optenv
HUSKY_UST10_SECONDARY_PREFIX rear)" />
  <xacro:arg name="laser_ust10_rear_parent"   default="$(optenv
HUSKY_UST10_SECONDARY_PARENT top_plate_link)" />
```

```
  <xacro:arg name="laser_ust10_rear_xyz"      default="$(optenv
HUSKY_UST10_SECONDARY_XYZ -0.2206 0.0 0.00635)" />
  <xacro:arg name="laser_ust10_rear_rpy"      default="$(optenv
HUSKY_UST10_SECONDARY_RPY 0 0 3.14159)" />

  <!-- Velodyne LiDAR Primary and Secondary -->
  <xacro:arg name="laser_3d_enabled"              default="$(optenv
HUSKY_LASER_3D_ENABLED 0)" />
  <xacro:arg name="laser_3d_topic"                default="$(optenv
HUSKY_LASER_3D_TOPIC points)"/>
  <xacro:arg name="laser_3d_tower"                default="$(optenv
HUSKY_LASER_3D_TOWER 1)"/>
  <xacro:arg name="laser_3d_prefix"               default="$(optenv
HUSKY_LASER_3D_PREFIX )"/>
  <xacro:arg name="laser_3d_parent"               default="$(optenv
HUSKY_LASER_3D_PARENT top_plate_link)"/>
  <xacro:arg name="laser_3d_xyz"                  default="$(optenv
HUSKY_LASER_3D_XYZ 0 0 0)" />
  <xacro:arg name="laser_3d_rpy"                  default="$(optenv
HUSKY_LASER_3D_RPY 0 0 0)" />

  <xacro:arg name="laser_3d_secondary_enabled"    default="$(optenv
HUSKY_LASER_3D_SECONDARY_ENABLED 0)" />
  <xacro:arg name="laser_3d_secondary_topic"      default="$(optenv
HUSKY_LASER_3D_SECONDARY_TOPIC secondary_points)"/>
  <xacro:arg name="laser_3d_secondary_tower"      default="$(optenv
HUSKY_LASER_3D_SECONDARY_TOWER 1)"/>
  <xacro:arg name="laser_3d_secondary_prefix"     default="$(optenv
HUSKY_LASER_3D_SECONDARY_PREFIX secondary_)"/>
  <xacro:arg name="laser_3d_secondary_parent"     default="$(optenv
HUSKY_LASER_3D_SECONDARY_PARENT top_plate_link)"/>
  <xacro:arg name="laser_3d_secondary_xyz"        default="$(optenv
HUSKY_LASER_3D_SECONDARY_XYZ 0 0 0)" />
  <xacro:arg name="laser_3d_secondary_rpy"        default="$(optenv
HUSKY_LASER_3D_SECONDARY_RPY 0 0 -3.14159)" />

  <!-- RealSense Camera Primary and Secondary -->
  <xacro:arg name="realsense_enabled"             default="$(optenv
HUSKY_REALSENSE_ENABLED 1)" />
  <xacro:arg name="realsense_topic"               default="$(optenv
HUSKY_REALSENSE_TOPIC realsense)" />
  <xacro:arg name="realsense_prefix"              default="$(optenv
HUSKY_REALSENSE_PREFIX camera)" />
  <xacro:arg name="realsense_parent"              default="$(optenv
HUSKY_REALSENSE_PARENT top_plate_link)" />
  <xacro:arg name="realsense_xyz"                 default="$(optenv
HUSKY_REALSENSE_XYZ 0 0 0)" />
  <xacro:arg name="realsense_rpy"                 default="$(optenv
HUSKY_REALSENSE_RPY 0 0 0)" />
```

```
  <xacro:arg name="realsense_secondary_enabled"   default="$(optenv
HUSKY_REALSENSE_SECONDARY_ENABLED 0)" />
  <xacro:arg name="realsense_secondary_topic"     default="$(optenv
HUSKY_REALSENSE_SECONDARY_TOPIC realsense_secondary)" />
  <xacro:arg name="realsense_secondary_prefix"    default="$(optenv
HUSKY_REALSENSE_SECONDARY_PREFIX camera_secondary)" />
  <xacro:arg name="realsense_secondary_parent"    default="$(optenv
HUSKY_REALSENSE_SECONDARY_PARENT top_plate_link)" />
  <xacro:arg name="realsense_secondary_xyz"       default="$(optenv
HUSKY_REALSENSE_SECONDARY_XYZ 0 0 0)" />
  <xacro:arg name="realsense_secondary_rpy"       default="$(optenv
HUSKY_REALSENSE_SECONDARY_RPY 0 0 0)" />

  <!-- BlackflyS Camera Primary and Secondary -->
  <xacro:arg name="blackfly_enabled"                  default="$(optenv
HUSKY_BLACKFLY 1)"/>
  <xacro:arg name="blackfly_mount_enabled"            default="$(optenv
HUSKY_BLACKFLY_MOUNT_ENABLED 1)"/>
  <xacro:arg name="blackfly_mount_angle"              default="$(optenv
HUSKY_BLACKFLY_MOUNT_ANGLE 0)"/>
  <xacro:arg name="blackfly_prefix"                   default="$(optenv
HUSKY_BLACKFLY_PREFIX blackfly)"/>
  <xacro:arg name="blackfly_parent"                   default="$(optenv
HUSKY_BLACKFLY_PARENT top_plate_link)"/>
  <xacro:arg name="blackfly_xyz"                      default="$(optenv
HUSKY_BLACKFLY_XYZ 0 0 0)"/>
  <xacro:arg name="blackfly_rpy"                      default="$(optenv
HUSKY_BLACKFLY_RPY 0 0 0)"/>

  <xacro:arg name="blackfly_secondary_enabled"        default="$(optenv
HUSKY_BLACKFLY_SECONDARY 0)"/>
  <xacro:arg name="blackfly_secondary_mount_enabled"  default="$(optenv
HUSKY_BLACKFLY_SECONDARY_MOUNT_ENABLED 1)"/>
  <xacro:arg name="blackfly_secondary_mount_angle"    default="$(optenv
HUSKY_BLACKFLY_SECONDARY_MOUNT_ANGLE 0)"/>
  <xacro:arg name="blackfly_secondary_prefix"         default="$(optenv
HUSKY_BLACKFLY_SECONDARY_PREFIX blackfly_secondary)"/>
  <xacro:arg name="blackfly_secondary_parent"         default="$(optenv
HUSKY_BLACKFLY_SECONDARY_PARENT top_plate_link)"/>
  <xacro:arg name="blackfly_secondary_xyz"            default="$(optenv
HUSKY_BLACKFLY_SECONDARY_XYZ 0 0 0)"/>
  <xacro:arg name="blackfly_secondary_rpy"            default="$(optenv
HUSKY_BLACKFLY_SECONDARY_RPY 0 0 0)"/>

  <!-- Bumper Extension -->
  <xacro:property name="husky_front_bumper_extend"  value="$(optenv
HUSKY_FRONT_BUMPER_EXTEND 1)" />
```

```
  <xacro:property name="husky_rear_bumper_extend"   value="$(optenv
HUSKY_REAR_BUMPER_EXTEND 1)" />

  <!-- Height of the sensor arch in mm.  Must be either 510 or 300 -->
  <xacro:arg name="sensor_arch"         default="$(optenv HUSKY_SENSOR_ARCH 0)" />
  <xacro:arg name="sensor_arch_height"  default="$(optenv HUSKY_SENSOR_ARCH_HEIGHT
510)" />
  <xacro:arg name="sensor_arch_xyz"     default="$(optenv HUSKY_SENSOR_ARCH_OFFSET 0
0 0)"/>
  <xacro:arg name="sensor_arch_rpy"     default="$(optenv HUSKY_SENSOR_ARCH_RPY 0 0
0)"/>

  <!-- Extras -->
  <xacro:arg name="robot_namespace" default="$(optenv ROBOT_NAMESPACE /)" />
  <xacro:arg name="urdf_extras"     default="$(optenv HUSKY_URDF_EXTRAS empty.urdf)"
/>
  <xacro:arg name="cpr_urdf_extras" default="$(optenv CPR_URDF_EXTRAS empty.urdf)" />

  <!-- Included URDF/XACRO Files -->
  <xacro:include filename="$(find
husky_description)/urdf/accessories/hokuyo_ust10.urdf.xacro" />
  <xacro:include filename="$(find
husky_description)/urdf/accessories/intel_realsense.urdf.xacro"/>
  <xacro:include filename="$(find
husky_description)/urdf/accessories/flir_blackfly_mount.urdf.xacro"/>
  <xacro:include filename="$(find
husky_description)/urdf/accessories/sensor_arch.urdf.xacro"/>
  <xacro:include filename="$(find
husky_description)/urdf/accessories/sick_lms1xx_mount.urdf.xacro"/>
  <xacro:include filename="$(find
husky_description)/urdf/accessories/vlp16_mount.urdf.xacro"/>
  <xacro:include filename="$(find husky_description)/urdf/decorations.urdf.xacro" />
  <xacro:include filename="$(find husky_description)/urdf/wheel.urdf.xacro" />

  <xacro:property name="M_PI" value="3.14159"/>

  <!-- Base Size -->
  <xacro:property name="base_x_size" value="0.98740000" />
  <xacro:property name="base_y_size" value="0.57090000" />
  <xacro:property name="base_z_size" value="0.24750000" />

  <!-- Wheel Mounting Positions -->
  <xacro:property name="wheelbase" value="0.5120" />
  <xacro:property name="track" value="0.5708" />
  <xacro:property name="wheel_vertical_offset" value="0.03282" />

  <!-- Wheel Properties -->
  <xacro:property name="wheel_length" value="0.1143" />
  <xacro:property name="wheel_radius" value="0.1651" />
```

```
  <!-- Base link is the center of the robot's bottom plate -->
  <link name="base_link">
    <visual>
      <origin xyz="0 0 0" rpy="0 0 0" />
      <geometry>
        <mesh filename="package://husky_description/meshes/base_link.dae" />
      </geometry>
    </visual>
    <collision>
      <origin xyz="${( (husky_front_bumper_extend - husky_rear_bumper_extend ) /
2.0)} 0 ${base_z_size/4}" rpy="0 0 0" />
      <geometry>
        <box size="${ base_x_size + 0.1 } ${base_y_size + 0.1} ${base_z_size/2}"/>
      </geometry>
    </collision>
    <collision>
      <origin xyz="0 0 ${base_z_size*3/4-0.01}" rpy="0 0 0" />
      <geometry>
        <box size="${base_x_size+0.1} ${base_y_size+0.1} ${base_z_size/2}"/>
      </geometry>
    </collision>
  </link>

  <!-- Base footprint is on the ground under the robot -->
  <link name="base_footprint"/>

  <joint name="base_footprint_joint" type="fixed">
    <origin xyz="0 0 ${wheel_vertical_offset - wheel_radius}" rpy="0 0 0" />
    <parent link="base_link" />
    <child link="base_footprint" />
  </joint>

  <!-- Inertial link stores the robot's inertial information -->
  <link name="inertial_link">
    <inertial>
      <mass value="46.034" />
      <origin xyz="-0.00065 -0.085 0.062" />
      <inertia ixx="0.6022" ixy="-0.02364" ixz="-0.1197" iyy="1.7386" iyz="-0.001544"
izz="2.0296" />
    </inertial>
  </link>

  <joint name="inertial_joint" type="fixed">
    <origin xyz="0 0 0" rpy="0 0 0" />
    <parent link="base_link" />
    <child link="inertial_link" />
  </joint>
```

```
  <!-- Husky wheel macros -->
  <xacro:husky_wheel wheel_prefix="front_left">
    <origin xyz="${wheelbase/2} ${track/2} ${wheel_vertical_offset}" rpy="0 0 0" />
  </xacro:husky_wheel>
  <xacro:husky_wheel wheel_prefix="front_right">
    <origin xyz="${wheelbase/2} ${-track/2} ${wheel_vertical_offset}" rpy="0 0 0" />
  </xacro:husky_wheel>
  <xacro:husky_wheel wheel_prefix="rear_left">
    <origin xyz="${-wheelbase/2} ${track/2} ${wheel_vertical_offset}" rpy="0 0 0" />
  </xacro:husky_wheel>
  <xacro:husky_wheel wheel_prefix="rear_right">
    <origin xyz="${-wheelbase/2} ${-track/2} ${wheel_vertical_offset}" rpy="0 0 0" />
  </xacro:husky_wheel>

  <!-- Husky Decorations -->
  <xacro:husky_decorate />

  <!--
    Add the main sensor arch if the user has specifically enabled it, or if a sensor
    requires it for mounting
  -->
  <xacro:if value="$(arg sensor_arch)">
    <xacro:sensor_arch prefix="" parent="top_plate_link" size="$(arg
sensor_arch_height)">
      <origin xyz="$(arg sensor_arch_xyz)" rpy="$(arg sensor_arch_rpy)"/>
    </xacro:sensor_arch>
  </xacro:if>

  <!--
      IMU Link: Standard location to add an IMU (i.e. UM7 or Microstrain)
  -->
  <link name="imu_link"/>
  <joint name="imu_joint" type="fixed">
    <origin xyz="$(arg imu_xyz)" rpy="$(arg imu_rpy)" />
    <parent link="$(arg imu_parent)" />
    <child link="imu_link" />
  </joint>
  <gazebo reference="imu_link">
  </gazebo>

  <!--
    SICK LMS1XX Priamry and Secondary Laser Scans
  -->
  <xacro:if value="$(arg laser_enabled)">
    <xacro:sick_lms1xx_mount prefix="$(arg laser_prefix)"/>
    <xacro:sick_lms1xx frame="$(arg laser_prefix)_laser" topic="$(arg laser_topic)"
robot_namespace="$(arg robot_namespace)"/>

    <joint name="$(arg laser_prefix)_laser_mount_joint" type="fixed">
```

```
      <origin xyz="$(arg laser_xyz)" rpy="$(arg laser_rpy)" />
      <parent link="$(arg laser_parent)" />
      <child link="$(arg laser_prefix)_laser_mount" />
    </joint>
  </xacro:if>

  <xacro:if value="$(arg laser_secondary_enabled)">
    <xacro:sick_lms1xx_mount prefix="$(arg laser_secondary_prefix)"/>
    <xacro:sick_lms1xx frame="$(arg laser_secondary_prefix)_laser" topic="$(arg
laser_secondary_topic)" robot_namespace="$(arg robot_namespace)"/>

    <joint name="$(arg laser_secondary_prefix)_laser_mount_joint" type="fixed">
      <origin xyz="$(arg laser_secondary_xyz)" rpy="$(arg laser_secondary_rpy)" />
      <parent link="$(arg laser_secondary_parent)" />
      <child link="$(arg laser_secondary_prefix)_laser_mount" />
    </joint>
  </xacro:if>

  <!--
    Hokuyo UST10 Primary and Secondary Laser Scans
  -->
  <xacro:if value="$(arg laser_ust10_front_enabled)">
    <xacro:hokuyo_ust10_mount topic="$(arg laser_ust10_front_topic)" prefix="$(arg
laser_ust10_front_prefix)" parent_link="$(arg laser_ust10_front_parent)">
        <origin xyz="$(arg laser_ust10_front_xyz)" rpy="$(arg laser_ust10_front_rpy)"
/>
      </xacro:hokuyo_ust10_mount>
  </xacro:if>

  <xacro:if value="$(arg laser_ust10_rear_enabled)">
    <xacro:hokuyo_ust10_mount topic="$(arg laser_ust10_rear_topic)" prefix="$(arg
laser_ust10_rear_prefix)" parent_link="$(arg laser_ust10_rear_parent)">
        <origin xyz="$(arg laser_ust10_rear_xyz)" rpy="$(arg laser_ust10_rear_rpy)"
/>
      </xacro:hokuyo_ust10_mount>
  </xacro:if>

  <!-- Intel Realsense Primary and Secondary -->
  <xacro:if value="$(arg realsense_enabled)">
    <link name="$(arg realsense_prefix)_realsense_mountpoint"/>
    <joint name="$(arg realsense_prefix)_realsense_mountpoint_joint" type="fixed">
      <origin xyz="$(arg realsense_xyz)" rpy="$(arg realsense_rpy)" />
      <parent link="$(arg realsense_parent)"/>
      <child link="$(arg realsense_prefix)_realsense_mountpoint" />
    </joint>
    <xacro:intel_realsense_mount prefix="$(arg realsense_prefix)" topic="$(arg
realsense_topic)" parent_link="$(arg realsense_prefix)_realsense_mountpoint"/>
  </xacro:if>
```

```
  <xacro:if value="$(arg realsense_secondary_enabled)">
    <link name="$(arg realsense_secondary_prefix)_realsense_mountpoint"/>
    <joint name="$(arg realsense_secondary_prefix)_realsense_mountpoint_joint"
type="fixed">
      <origin xyz="$(arg realsense_secondary_xyz)" rpy="$(arg
realsense_secondary_rpy)" />
      <parent link="$(arg realsense_secondary_parent)"/>
      <child link="$(arg realsense_secondary_prefix)_realsense_mountpoint" />
    </joint>
    <xacro:intel_realsense_mount prefix="$(arg realsense_secondary_prefix)"
topic="$(arg realsense_secondary_topic)" parent_link="$(arg
realsense_secondary_prefix)_realsense_mountpoint"/>
  </xacro:if>

  <!-- BlackflyS Camera Primary and Secondary -->
  <xacro:if value="$(arg blackfly_enabled)">
    <xacro:flir_blackfly_mount prefix="$(arg blackfly_prefix)"
                               parent="$(arg blackfly_parent)"
                               mount_enabled="$(arg blackfly_mount_enabled)"
                               mount_angle="$(arg blackfly_mount_angle)">
      <origin xyz="$(arg blackfly_xyz)" rpy="$(arg blackfly_rpy)"/>
    </xacro:flir_blackfly_mount>
  </xacro:if>

  <xacro:if value="$(arg blackfly_secondary_enabled)">
    <xacro:flir_blackfly_mount prefix="$(arg blackfly_secondary_prefix)"
                               parent="$(arg blackfly_secondary_parent)"
                               mount_enabled="$(arg
blackfly_secondary_mount_enabled)"
                               mount_angle="$(arg blackfly_secondary_mount_angle)">
      <origin xyz="$(arg blackfly_secondary_xyz)" rpy="$(arg
blackfly_secondary_rpy)"/>
    </xacro:flir_blackfly_mount>
  </xacro:if>

  <!--
    Velodyne 3D LiDAR Primary and Secondary
  -->
  <xacro:if value="$(arg laser_3d_enabled)">
    <xacro:if value="$(arg laser_3d_tower)">
      <xacro:vlp16_mount prefix="$(arg laser_3d_prefix)" parent_link="$(arg
laser_3d_parent)" topic="$(arg laser_3d_topic)">
        <origin xyz="$(arg laser_3d_xyz)" rpy="$(arg laser_3d_rpy)" />
      </xacro:vlp16_mount>
    </xacro:if>
    <xacro:unless value="$(arg laser_3d_tower)">
      <xacro:VLP-16 parent="$(arg laser_3d_parent)" topic="$(arg laser_3d_topic)"
name="$(arg laser_3d_prefix)velodyne">
        <origin xyz="$(arg laser_3d_xyz)" rpy="$(arg laser_3d_rpy)" />
```

```
      </xacro:VLP-16>
    </xacro:unless>
  </xacro:if>

  <gazebo>
    <plugin name="gazebo_ros_control" filename="libgazebo_ros_control.so">
      <robotNamespace>$(arg robot_namespace)</robotNamespace>
      <legacyModeNS>true</legacyModeNS>
    </plugin>
  </gazebo>

  <gazebo>
    <plugin name="imu_controller" filename="libhector_gazebo_ros_imu.so">
      <robotNamespace>$(arg robot_namespace)</robotNamespace>
      <updateRate>50.0</updateRate>
      <bodyName>base_link</bodyName>
      <topicName>imu/data</topicName>
      <accelDrift>0.005 0.005 0.005</accelDrift>
      <accelGaussianNoise>0.005 0.005 0.005</accelGaussianNoise>
      <rateDrift>0.005 0.005 0.005 </rateDrift>
      <rateGaussianNoise>0.005 0.005 0.005 </rateGaussianNoise>
      <headingDrift>0.005</headingDrift>
      <headingGaussianNoise>0.005</headingGaussianNoise>
    </plugin>
  </gazebo>

  <gazebo>
    <plugin name="gps_controller" filename="libhector_gazebo_ros_gps.so">
      <robotNamespace>$(arg robot_namespace)</robotNamespace>
      <updateRate>40</updateRate>
      <bodyName>base_link</bodyName>
      <frameId>base_link</frameId>
      <topicName>navsat/fix</topicName>
      <velocityTopicName>navsat/vel</velocityTopicName>
      <referenceLatitude>49.9</referenceLatitude>
      <referenceLongitude>8.9</referenceLongitude>
      <referenceHeading>0</referenceHeading>
      <referenceAltitude>0</referenceAltitude>
      <drift>0.0001 0.0001 0.0001</drift>
    </plugin>
  </gazebo>

  <!-- Optional custom includes. -->
  <xacro:include filename="$(arg urdf_extras)" />

  <!-- Optional for Clearpath internal softwares -->
  <xacro:include filename="$(arg cpr_urdf_extras)" />

</robot>
```