Universidad Miguel Hernández de Elche

MÁSTER UNIVERSITARIO EN ROBÓTICA

“Creación de mapas de navegación de estructuras reticulares mediante un robot híbrido trepador en entornos de simulación”

Trabajo de Fin de Máster

Curso académico

2021/2022

Autor: Francisco José Soler Mora

Tutor/es: Óscar Reinoso García

Resumen

Las estructuras reticulares de naturaleza metálica son uno de los elementos más presentes en construcciones artificiales realizadas por el hombre, como son los esqueletos de edificios, puentes y torres eléctricas o de comunicaciones. Dichas estructuras requieren tareas de inspección y mantenimiento las cuales suelen realizarse a grandes alturas o en ambientes peligrosos para los operarios que las realizan. Una solución para reducir riesgos humanos es el uso de robots trepadores para desempeñar estas tareas.

El presente trabajo se centra en desarrollar uno de los primeros aspectos requeridos en una aplicación de navegación en robótica móvil, la creación de un mapa. Para cumplir con este objetivo, en primer lugar se ha desarrollado un entorno de simulación del cual obtener nubes de puntos del entorno simulado. En segundo lugar, se ha desarrollado una aplicación para la construcción del mapa de navegación partiendo de las nubes de puntos registradas en simulación. Todo ello se ha desarrollado bajo la plataforma *Robot Operating System* (ROS).

Agradecimientos

En primer lugar, agradecer a mi tutor Óscar Reinoso , por ofrecerme la oportunidad de realizar prácticas en su departamento, así como brindarme flexibilidad y orientarme para realizar este trabajo. Agradecer a todos los integrantes del grupo ARVC su ayuda e implicación cuando la he necesitado y en especial a mis compañeros de laboratorio.

Por último, agradecer a mi pareja y toda mi familia su apoyo y su ánimo durante todo el proceso del máster y el desarrollo de este TFM.

Índice

Índice de figuras

Índice de tablas

Capítulo 1

1. Introducción

Hoy día, la robótica es objeto de aplicación a cualquier tarea que se pueda imaginar, desde aplicaciones domésticas como cocinar, hasta aplicaciones de exploración de otros planetas. El avance de la tecnología, el desarrollo de microprocesadores cada vez más pequeños con mayor potencia y menor consumo, la aparición de nuevos sensores, reducción del coste de los mismos o la impresión 3D, son algunos de los principales motivos por los cuales la aplicación de la robótica a cualquier tarea se ha visto aumentada enormemente.

Una de las aplicaciones más beneficiosas de la robótica, es la reducción de riesgos para el ser humano, como puede ser su aplicación en cirugía, mejorando la precisión del cirujano y la probabilidad de éxito de la operación, o en tareas de riesgo para el ser humano como sofocación de incendios o inspección y mantenimiento de zonas peligrosas.

1.1. HyReCRo: *Hybrid Redundant Climbing Robot*

El "HyReCRo" es un robot híbrido de 10 GDL diseñado por el grupo de investigación en Automatización, Robótica y Visión por Computador (ARVC) de la Universidad Miguel Hernández de Elche (UMH) [1]. Este robot se diseñó para sustituir a operarios en la realización tareas peligrosas de inspección y mantenimiento en estructuras reticulares metálicas.

Se trata de un robot bípedo con una arquitectura híbrida serie-paralela. Su construcción está basada en módulos paralelos conectados entre sí. Cada módulo paralelo está formado por dos actuadores prismáticos fijados a una plataforma en un extremo y a cuerpo central con movimiento restringido mediante una guía lineal. Esta configuración, mostrada en la Figura 1.1.a, otorga a cada módulo paralelo 2 GDL permitiendo controlar la distancia y orientación de la base respecto al elemento unido a la guía lineal o viceversa.

Cada pata se forma mediante dos módulos paralelos contrapuestos, lo que otorga a cada pata la capacidad de modificar la distancia entre sus plataformas y la orientación de cada una de ellas, en total, cada pata dispone de 4 GDL. En la Figura 1.1.b donde se muestra un modelo CAD 3D del robot, se pueden distinguir para cada pata (A o B) los elementos mencionados, donde el cuerpo central o core de los módulos paralelos se señala mediante una "C", y las plataformas mediante una "P".

Por último, cada pata se une a un mismo elemento denominado "cadera" o "hip" en inglés, mediante una articulación rotativa. Estas articulaciones rotativas, otorgan 1 GDL adicional cada, dotando al HyReCRo de un total de 10 GDL, por lo que se considera un robot redundante.

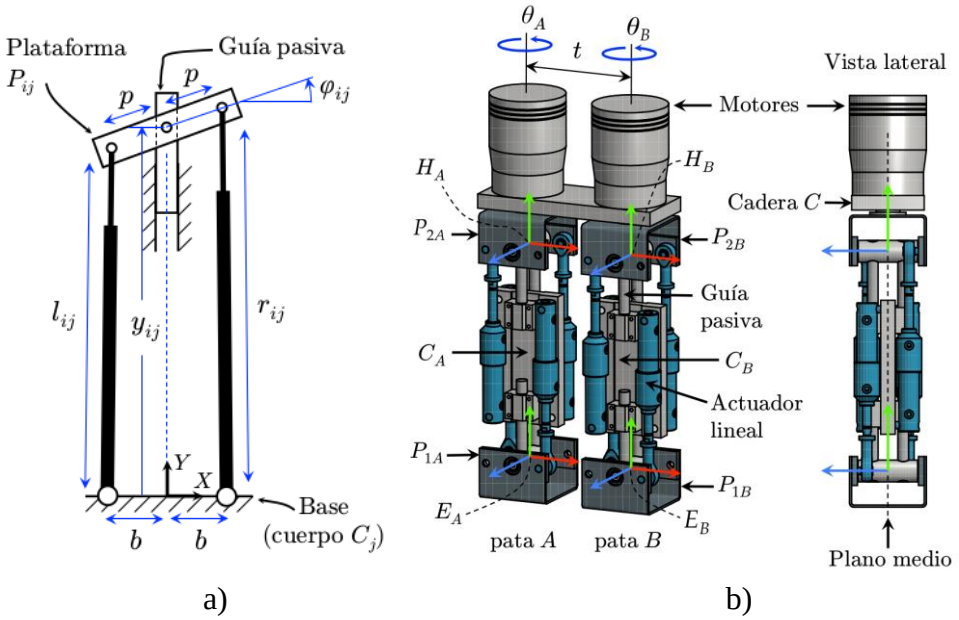


Figura 1.1. Módulo paralelo y modelo CAD del HyReCRo

El movimiento del robot se consigue adhiriendo la plataforma inferior de una pata a la estructura metálica mediante un sistema de imanes permanentes conmutables dejando libre el resto del robot. Durante el movimiento del robot, sólo una de las patas estará fija a la estructura.

1.2. Motivación

Las estructuras reticulares metálicas están muy presentes en la sociedad actual, las podemos encontrar en multitud de construcciones artificiales, como torres de alta tensión, aeropuertos, refinerías, puentes y demás construcciones. Este trabajo nace con el propósito de dotar al robot HyReCRo de la capacidad de navegar a través de este tipo de estructuras, pudiendo así reducir el riesgo al que se exponen los operarios que realizan tareas de inspección y mantenimiento en ellas. Dotar al robot de dicha capacidad se puede dividir en varias subtareas:

- Sensorización
- Mapeado
- Planificación de trayectorias
- Evasión de obstáculos
- Sistema de control

Este trabajo pretende cumplir con las tareas de sensorización y mapeado, para en un futuro dotar al robot de la capacidad de navegar de forma autónoma. Dado que la construcción de un prototipo, suele ser un proceso largo y de un elevado coste económico, en este proyecto se propone implementar una simulación como herramienta para reducir el tiempo de desarrollo, disponer de facilidad para realizar experimentos, capturar de datos de diferentes tipos de estructuras y poner a prueba distintos tipos de algoritmos de mapeado sin riesgo para el prototipo.

1.3. Objetivos

El objetivo que persigue este trabajo es construir un mapa 3D de una estructura reticular partiendo nubes de puntos. Para cumplir con este objetivo, el trabajo principalmente dos aspectos:

- **Simulación:** gran parte de este trabajo se centra implementar una simulación, en la que se incluya tanto la estructura, como el robot, su movimiento y sensores capaces de generar datos de nubes de puntos a partir de las cuales construir el mapa.
- **Construcción del mapa:** el segundo aspecto a afrontar, consiste en aplicar algoritmos para registrar nubes de puntos entre sí, permitiendo así construir un mapa tridimensional de nubes de puntos a medida que el robot se mueve por la estructura.

1.4. Materiales

En este apartado se muestra la lista de las principales herramientas y dispositivos empleados:

- **Ordenador:** el desarrollo de este trabajo se ha realizado en un ordenador con las siguientes características:
  - Sistema Operativo: Ubuntu 20.04 LTS
  - Procesador: Intel i7-10700 2.9Ghz (8C/16T)
  - Memoria RAM: 32 GB

- **Ouster OS-1-128:** sensor LiDAR de clase 1, con 128 canales verticales y resolución seleccionable entre 512, 1024 y 2048 puntos para el canal horizontal. Con una precisión de (±0.7 - 5cm) permite crear nubes de puntos muy densas de gran resolución y sin ser dañino para las personas a su alrededor. Será uno de los sensores empleados en la simulación del robot para obtener nubes de puntos con las que crear el mapa.
- **Intel RealSense d435i:** cámara RGB-D de tipo estereoscópica con una resolución de profundidad de 1280x720 y un rango de (0.3 - 3m). Se emplea en la simulación del robot para obtener nubes de puntos con las que crear el mapa.
- **HyReCRo** [1]**:** robot híbrido de 10 GDL diseñado para navegar a través de estructuras reticulares. Gracias a su diseño híbrido es capaz de alcanzar un gran espacio de trabajo además de poder soportar cargas de cierto peso.
- ***Robot Operating System* (ROS):** plataforma de trabajo de código abierto orientada a robótica. Otorga gran facilidad a la hora de desarrollar y compartir código. Este trabajo se desarrolla totalmente en ROS, más concretamente en su distribución ROS Noetic.

- **Gazebo Simulator:** simulador totalmente integrado con ROS bajo el que se integrará el HyReCRo y los sensores arriba indicados. Se emplea la versión 11.9 de Gazebo.
- **Gazebo API** [2]: interfaz de programación en C++ que permite desarrollar nuevos comportamientos o funciones específicas (plugins) dentro del simulador Gazebo.
- ***Point Cloud Library* (PCL)** [3]**:** librería en C++ de código abierto disponible en todas las plataformas. Dispone de multitud de algoritmo y facilidades para el tratamiento, filtrado, extracción de características y visualización de nubes de puntos y creación del mapa.

1.5. Estructura de la memoria

El presente Trabajo fin de Máster se organiza en los siguientes capítulos:

- **Capítulo 2:** este capítulo propone un estado del arte sobre robots trepadores y los métodos de navegación que emplean. Pretende contextualizar al lector sobre distintos tipos de robots trepadores y sus diferencias con el HyReCRo. Además, se muestran los métodos empleados en la actualidad para construcción de mapas y navegación en estructuras reticulares.
- **Capítulo 3:** una de las secciones de mayor peso de este trabajo, en la que se describen los principales aspectos del simulador empleado, la inclusión del HyReCRo, Ouster-OS1 y RealSense d435 en este último, además de describir el desarrollo de un plugin para permitir controlar el HyReCRo desde ROS.
- **Capítulo 4:** el segundo gran bloque del trabajo, donde se describe en primer lugar la distribución de las herramientas desarrolladas en ROS durante este trabajo. En segundo lugar, se hace una breve descripción de la estructura de la librería PCL. Por último, se describe el proceso de mapeado empleado y se realiza una comparación entre los resultados obtenidos por distintos algoritmos.
- **Capítulo 5:** este capítulo cierra el presente trabajo mediante unas conclusiones tras analizar y comentar los resultados obtenidos. Además, se tratan aspectos mejorables y posibles líneas de investigación y trabajos futuros.

Capítulo 2

2. Estado del Arte

En este capítulo se pone en contexto el HyReCRo como robot trepador de estructuras reticulares y se comentan aspectos de interés sobre la construcción de mapas con nubes de puntos.

2.1. Robots trepadores

El propósito general de este tipo de robos es la realización de tareas de inspección y mantenimiento a grandes alturas, o en entornos peligrosos donde un operario humano estaría expuesto a grandes riesgos para su salud, como caídas desde gran altura, electrocuciones o exposición a sustancias o gases perjudiciales [4].

Desde hace más de dos décadas ha habido numerosas investigaciones sobre este tipo de robots [5], sin embargo, no existe un gran número de robots comerciales hoy en día y estos suelen aparecer en aplicaciones muy específicas [4].

Se puede establecer una clasificación de este tipo de robots en función del tipo de locomoción y el método de adhesión al entorno que emplean [6].

2.1.1. Clasificación según su sistema de movimiento

Los principales métodos de locomoción que se pueden encontrar en la literatura son los siguientes:

- **Caminantes**: se trata de robots cuyo método de movimiento se basa en la disposición de brazos o piernas, de modo que su movimiento se consigue intercalando la fijación y el movimiento de sus patas. Dependiendo del entorno al que se deseen aplicar, se pueden encontrar en la literatura robots bípedos, cuadrúpedos y hexápodos principalmente [6]. Su gran número de patas les otorga una gran maniobrabilidad y GDL que les permite navegar en entornos complicados. A su vez, esto último provoca que su control sea complicado, que tengan un elevado peso y una velocidad de movimiento reducida, lo que no los convierte en los más aptos para superficies de gran tamaño (> 1000m^2) (Figura 2.1).
- **Con ruedas**: este tipo de robots emplean ruedas para navegar por los entornos, lo que les permite un movimiento continuo y rápido a diferencia de los anteriores. Su falta de maniobrabilidad a la hora de evitar obstáculos, o adaptarse al entorno, se compensa por su sencillez en el control y construcción mecánica. Este tipo de robots está más orientado a paredes o superficies planas de gran tamaño (Figura 2.2).

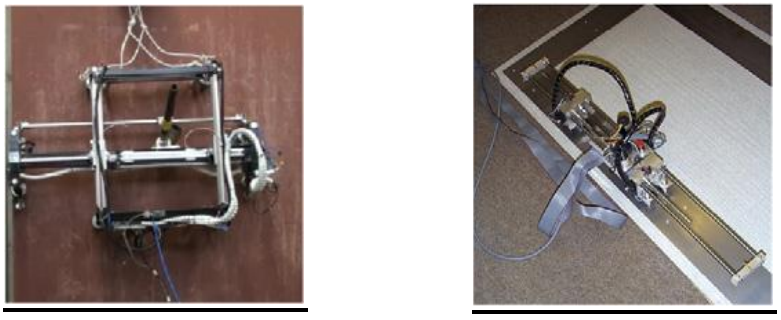
a) TREMO b) REST c) Winspectbot

Figura 2.1. Robots trepadores caminantes [6].

a) b) c)

Figura 2.2. Robots trepadores con ruedas.

- **Deslizantes:** este tipo de robots obtiene su capacidad de movimiento gracias a fijar y deslizar o rotar sus elementos. Su funcionamiento es similar a los caminantes, debido a que también realizan el movimiento paso a paso. Por el contrario que los robots caminantes, este tipo de robots presenta un control bastante más sencillo, disponiendo también de un menor número de GDL.



a) b)

Figura 2.3. Robots trepadores deslizantes [6].

- **Cableados:** esta última clasificación de robots, se da en los casos en que el movimiento del robot se debe a que está sujeto a un cable o una guía, de modo que el movimiento del robot queda restringido por los cables o guías a los que esté

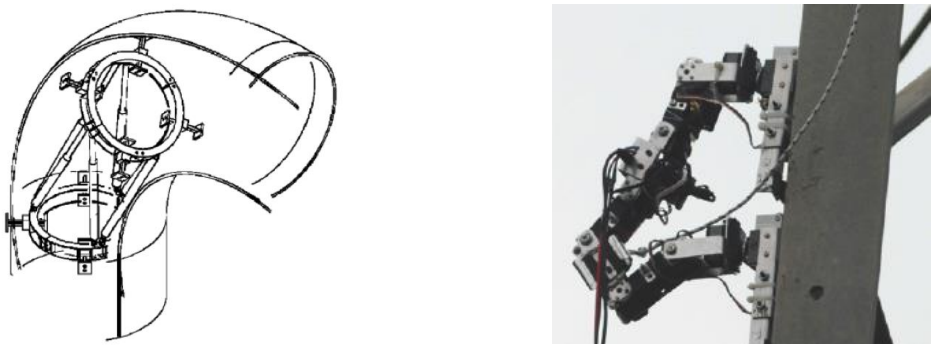
conectado. Este tipo de robots se emplean en fachadas de edificios donde la caída del robot podría provocar daños a terceros (edificios en ciudad). La ventaja de este tipo de robots es que al estar conectados a un cable o guía el riesgo de caída es mínimo.

Tal y como se indica en [4], el tipo de robots trepadores más extendido son los robots caminantes, seguido de los robots con ruedas.

Dentro de esta primera clasificación, el HyReCRo se identifica como un robot caminante, más concretamente bípedo. Para caracterizar más al HyReCRo, se puede realizar una clasificación adicional dentro de este tipo de robots en función de la arquitectura que empleen en sus cadenas cinemáticas. La arquitectura cinemática de estos robots puede ser:

- **Serie**: se trata de robots cuyas cadenas cinemáticas son todas abiertas, es decir, de tipo serie. Este hecho les otorga un gran número de GDL y maniobrabilidad en estructuras complicadas además de un amplio espacio de trabajo. Un ejemplo de este tipo de robots en un robot manipulador industrial, cuya base y efector final se van adhiriendo al entorno intercaladamente para realizar un movimiento paso a paso.
- **Paralela**: robots formados por cadenas cinemáticas cerradas. Esta arquitectura les otorga una gran capacidad de carga y gran estabilidad. Ejemplos de este tipo de robot se basan en la plataforma Stewart, logrando su movimiento paso a paso, fijando y liberando cada uno de sus extremos. Basarse en la plataforma Stewart otorga un gran número de GDL, pero su espacio de trabajo es muy reducido.
- **Híbrida**: este tipo de robots combinan ambas arquitecturas anteriores con el objetivo de dotar al robot de las ventajas de ambas, un amplio espacio de trabajo y una gran capacidad de carga y estabilidad.

El HyReCRo trata de explotar este último concepto, siendo un robot híbrido que combina ambos tipos de estructuras, lo que le otorga un amplio espacio de trabajo y gran capacidad de carga.



a) Robot basado en plataforma Stewart b) Basado en robot de tipo serie.

Figura 2.4. Ejemplo de robots trepadores paralelos y serie.

2.1.2. Clasificación según su sistema de adhesión

La literatura clasifica además los robots trepadores según su sistema de adherencia a las superficies. La clasificación se realiza en cinco grandes grupos:

- **Magnética**: emplea imanes permanentes o electroimanes para sujetar el robot a superficies metálicas. El uso de imanes permanentes supone una alta eficiencia energética, algo necesario para la autonomía del robot. Estos imanes pueden estar en contacto o no con la superficie del entorno. Su uso se limita a entornos ferromagnéticos.
- **Neumática**: emplea sistemas o neumáticos para adherirse a las superficies. Es el sistema de agarre más extendido y se emplea normalmente cuando no se puede aplicar métodos de adhesión magnéticos. El sistema neumático puede ser pasivo o activo, algo que depende de los requerimientos del entorno pero en cuanto a términos de eficiencia energética un sistema pasivo será mejor. No tiene la limitación de entorno de los sistemas de adhesión magnética.
- **Mecánica**: utiliza métodos de agarre mecánicos como pinzas o garras. Su principal ventaja se encuentra en el consumo de energía y seguridad, puesto que a pesar de que se pierda la alimentación, las pinzas se mantienen cerradas y el robot adherido al entorno.
- **Electroestática**: basan su funcionamiento en generar fuerzas electroestáticas o de Van der Waals entre la superficie del entorno y el robot. Este tipo de adhesión aún está en vías de desarrollo y no se encuentra muy extendida.
- **Química**: este tipo de métodos de adhesión es poco frecuente. Se basa en el uso de compuestos adhesivos, como pegamentos. Han de ir añadiendo y eliminando estos compuestos para avanzar por la superficie lo que provoca que no se empleen apenas.

El HyReCRo emplea un sistema de adhesión magnética propuesto en [7]. Este sistema emplea imanes permanentes, lo que proporciona un consumo energético reducido con una fuerza de adhesión capaz de soportar hasta 33 kg por pata.

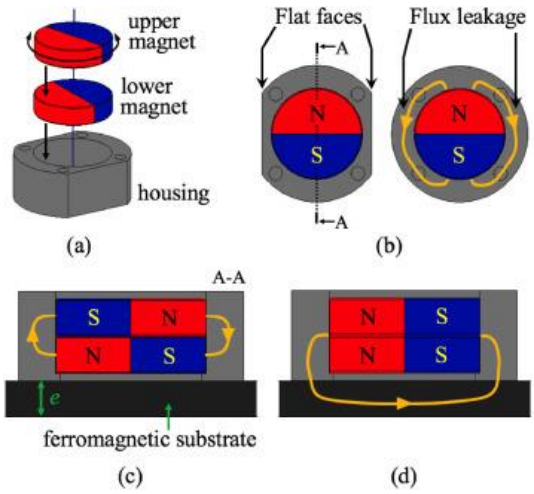


Figura 2.5. Sistema de sujeción diseñado para el HyReCRo [7].

2.2. Mapas de Navegación con Nubes de Puntos

En esta sección se realiza un estado del arte sobre las metodologías empleadas hoy en día para la creación de mapas a partir de nubes de puntos. Las nubes de puntos suelen ser archivos pesados y que requieren de un cierto coste computacional para alinearlas y construir el mapa.

Figura 2.6. Ejemplo de mapa construido a partir de nubes de puntos.

Como se observa en la Figura 2.6, un mapa de navegación construido a partir de nubes de puntos, dispone de muchísima información, por lo que su tratamiento será algo pesado. Gracias al avance de algoritmos y la tecnología, como sensores y microprocesadores, hoy día es posible trabajar con nubes de puntos en tiempo real. En los siguientes apartados se comentarán aspectos de creación de mapas con LiDAR y con sensores de profundidad (cámaras RGB-D).

La construcción de un mapa mediante nubes de puntos, se basa en obtener una transformación (traslación y rotación) entre cada captura de nubes de puntos. Lo anterior se podría entender como conocer la posición y orientación desde la que se captura cada nube de puntos. Si se conoce la posición y orientación de cada toma respecto de un mismo sistema de referencia, se pueden integrar todas las nubes de puntos para formar un único mapa.

Para conocer la posición y orientación de cada nube de puntos respecto de un mismo sistema de referencia existen diferentes técnicas, pero las más usadas se basan en técnicas de emparejamiento de nubes de puntos usando algoritmos basados en *Iterative Closest Point (***ICP***)* o ICP en combinación con extracción de características de las nubes de puntos.

2.2.1. Iterative Closest Point (ICP)

El algoritmo denominado ICP fue propuesto originalmente en [8], realiza un proceso iterativo en el cual se intenta minimizar la distancia entre pares de puntos correspondientes. Existen distintos tipos de medidas de distancia, que proporcionan resultados diferentes que dependiendo del entorno, ofrecen mejores o peores resultados. Los principales tipos de distancias que existen son, punto-punto, punto-plano, plano-plano.

A continuación se desarrolla cada una de estas distancias en mayor medida.

Punto-punto

Como su nombre indica, este tipo de distancia emplea distancia entre puntos, de modo que para cada par de puntos correspondientes se minimiza la distancia euclídea entre ellos. Seguidamente se pasa al siguiente par de puntos y se repite el proceso, hasta conseguir que la distancia entre los puntos sea mínima. En la Figura 2.7 se ejemplifica gráficamente el funcionamiento del ICP con distancia punto a punto [9].

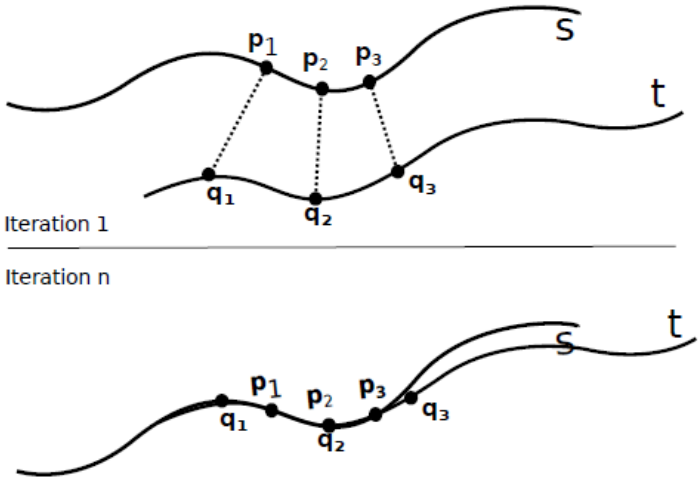


Figura 2.7. ICP basado en distancia punto a punto.

Punto-plano

Esta medida se sitúa en la recta perpendicular a la superficie normal a un punto, y su punto correspondiente. Esto se ejemplifica mejor en la Figura 2.8. Esta segunda distancia es más robusta frente a falsas correspondencias que la anterior (punto-punto) [9].

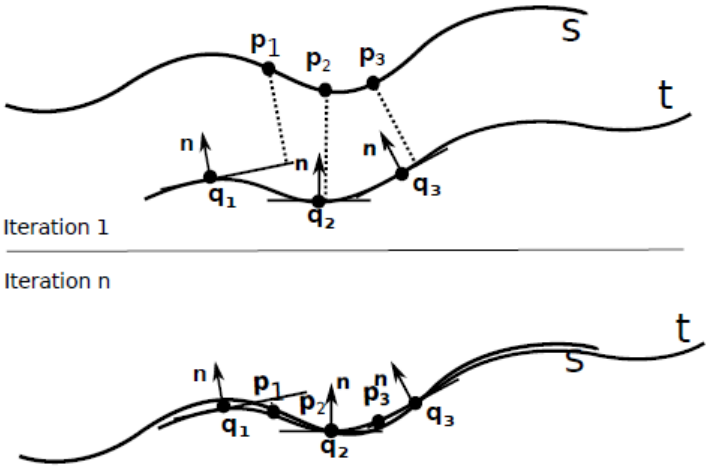


Figura 2.8. ICP basado en distancia punto plano.

Plano-plano

La distancia plano a plano, o una aproximación de esta, fue introducida por Segal *et al.* [10]. En [10] se presenta una variante del ICP, denominada Generalized-ICP en la que se combina tanto la distancia punto-plano como la distancia punto-plano. Este algoritmo tiene en cuenta la superficie plana de ambas nubes de puntos y es más robusto frente a falsas correspondencias que cualquiera de los algoritmos anteriores anteriores. Por otro lado, su tiempo de cómputo también es más elevado que los anteriores. En la Figura 2.9 se observan las diferencias entre la alineación realizada por ICP-Punto-plano y el Generalized-ICP. Se puede observar como la alineación es mucho mejor con el Generalized-ICP.

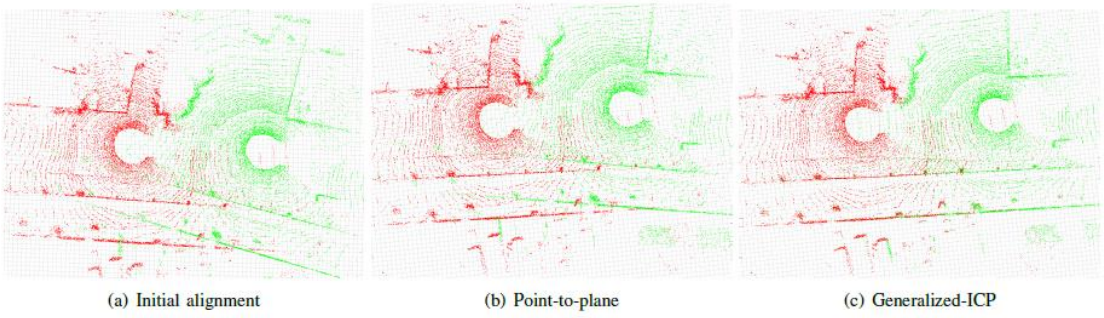


Figura 2.9. Comparación de algoritmos de ICP [10].

Cualquiera de los algoritmos anteriores basados en ICP, necesitan de una transformación inicial o que la traslación y rotación entre las nubes de puntos sea pequeña. Para solventar este problema, aparecen en la literatura métodos para extraer características de las nubes de puntos que permitan establecer una transformación inicial.

2.2.2. Extracción de características

En la literatura existen varios artículos en los que se implementan métodos para extraer características de las nubes de puntos para obtener una estimación inicial que proporcionar a los algoritmos de ICP.

Existen diversas propuestas para ello, pero la gran mayoría coincide en extraer características de las superficies planas. A partir de las superficies planas hacer una correlación entre planos para obtener una transformación inicial y posteriormente un ICP, ya sea punto-punto, punto-plano o plano-plano para afinar más la transformación. Dentro de estos métodos se encuentran los artículos [11], [12] o [13].

Capítulo 3

3. Simulación

Hoy en día, disponer de una herramienta de simulación en robótica se ha convertido en algo indispensable. Gracias a este tipo de herramientas se puede comprobar el desarrollo realizado y realizar multitud de pruebas de forma rápida y sencilla. Existen multitud de software de simulación actualmente, como pueden ser CoppeliaSim, o GazeboSim, dos simuladores con posibilidad de integrarse con ROS y de libre acceso (CoppeliaSim en su versión "Edu").

Ambos programas son buenas opciones a la hora de simular entornos bajo un desarrollo en ROS, tanto el uno como el otro incorporan multitud de sensores y robots que se pueden integrar de forma sencilla para empezar a simular el entorno deseado. Los factores positivos y negativos de cada uno de ellos son los siguientes.

- **Gazebo**:
  - Sencilla integración con ROS
  - Amplia y extensa comunidad
  - Desarrollo de plugins complejo
  - API de referencia sólo en C++
- **CoppeliaSim**:
  - Desarrollo de plugins sencillo
  - API de referencia en C++, Lua y Python
  - Compleja integración con ROS

Para este trabajo se ha decidido emplear Gazebo, debido a que su integración con ROS viene por defecto, siendo muy sencillo comunicarse entre ROS y Gazebo. Por otro lado, a pesar de que la comunicación entre ROS y CoppeliaSim es algo más complicada, el acceso a la API de Coppelia y desarrollos personalizados en ella es más sencillo e intuitivo, estado esta última disponible en lenguajes C++, Lua y Python.

3.1. Gazebo

Gazebo es una plataforma de simulación 3D que permite al usuario una recreación realista de un entorno donde comprobar, analizar y estudiar el desarrollo de tareas realizadas por sistemas robóticos.

Debido a que el objetivo final de este trabajo es construir mapas de navegación a partir de nubes de puntos, la simulación de físicas en el entorno de simulación es algo innecesario. Desactivar las físicas simplifica y agiliza la simulación del HyReCRo, evitando problemas en su movimiento y en la adhesión a las superficies simulando magnetismo.

El presente capítulo divide el desarrollo de una simulación en dos aspectos, la descripción de los elementos físicos y el desarrollo de plugins.

3.1.1. Descripción de elementos

Un entorno de simulación en Gazebo, comúnmente se conforma de dos elementos, un mundo y un modelo. El mundo, representa todo el entorno de simulación y su función es la de contener todos los elementos que no sean el modelo del robot. Por otro lado, el modelo del robot se inserta en el mundo y contiene todos los componentes que forman el robot incluidos sus sensores.

La descripción de cada uno de los elementos mencionados ya sea el mundo o el modelo del robot, se realiza siguiendo un formato específico en lenguaje XML. Además de lo anterior, Gazebo dispone de herramientas para la construcción del mundo, o del robot de forma gráfica, sin necesidad de escribir la descripción de estos elementos manualmente.

Independientemente de como se realice la descripción de los elementos, gráfica (sólo SDF) o manualmente, el resultado debería ser el mismo, un archivo en formato XACRO/URDF o SDF. Por un lado, los formatos XACRO [14] y URDF [15] son específicos de ROS y su diferencia recae en que el formato XACRO es un formato previo a URDF, creado para poder construir sentencias condicionales, macros que poder instanciar repetidas veces, parametrizar valores, etc. La forma más común de proceder es desarrollar la descripción en XACRO para más tarde emplear el conversor automático incorporado en ROS que permite convertir un archivo XACRO a URDF. Por otro lado, el formato SDF [16] mencionado, es un formato específico de Gazebo cuya descripción de los elementos es muy similar al formato URDF, pero más completo al ser un formato específico del propio simulador.

Comentarios y aspectos a tener en cuenta:

- Las uniones, sean del tipo que sean y en cualquiera de los formatos, requieren de un elemento padre y un elemento hijo, los cuales se pueden entender como maestro y esclavo. Esto se traduce en que el elemento esclavo, cuando estén las físicas desactivadas, se moverá siempre con respecto al elemento maestro y no viceversa.
- A la hora de insertar un modelo URDF en Gazebo, este último realiza una conversión interna a formato SDF.
- La descripción de uniones fijas (`<joint type= “fixed”>`) en formato URDF provoca que durante la conversión a SDF, se genere un único elemento en Gazebo formado por los elementos que forman la unión en URDF. Esto provoca que en ocasiones no se pueda obtener la posición de un elemento específico del modelo al estar fusionado con otros elementos.
- Se puede evitar que Gazebo fusione dos elementos unidos mediante una unión fija mediante la siguiente instrucción:

```
<gazebo reference='JOINT_NAME'>
  <preserveFixedJoint>true</preserveFixedJoint>
</gazebo>
```

- La instrucción anterior no se encuentra documentada en [15].
- Para que la instrucción anterior funcione ambos elementos (`<link>`) han de tener descripción sobre su elemento `<intertial>` a pesar de que sus valores sean cero. En caso de no disponer del campo `<intertial>` la conversión sufrirá un conflicto y no se generará correctamente el modelo en Gazebo.
- En las Figura 3.1 y Figura 3.2, se muestra el modelo generado en Gazebo a partir del mismo URDF, con la diferencia de que en la Figura 3.2 se ha indicado la instrucción `<preserveFixedJoint>`.

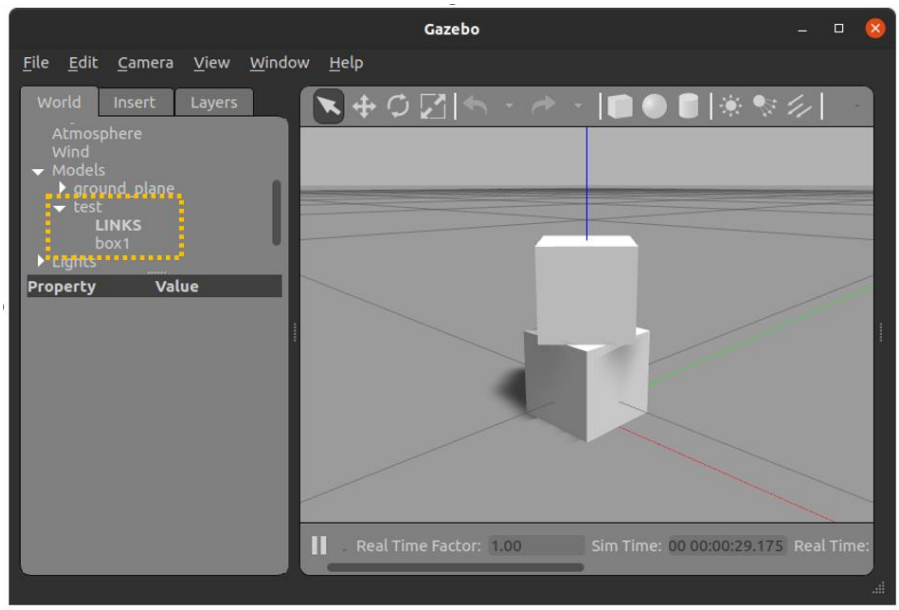


Figura 3.1. Modelo de ejemplo sin mantener la unión fija

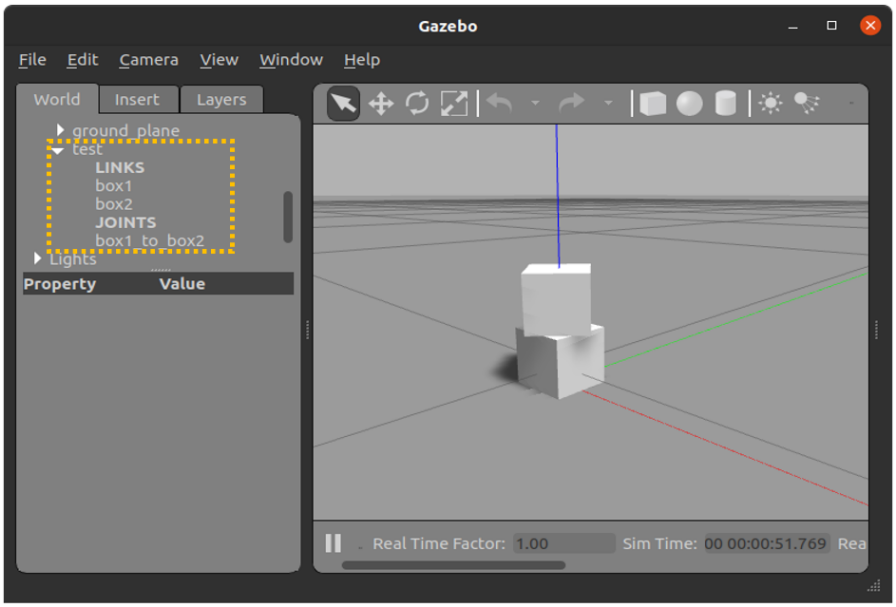


Figura 3.2. Modelo de ejemplo manteniendo la unión fija

Una descripción de los elementos empleados en la simulación se realiza en las siguientes secciones de este capítulo.

3.1.2. Plugins

El segundo aspecto de importancia para este trabajo son los plugins. Un plugin en Gazebo es un fragmento de código previamente compilado, que tiene comunicación con el entorno de simulación y se ejecuta durante el transcurso de esta. Los plugins de Gazebo se desarrollan a través de su interfaz de programación de aplicaciones (API) en lenguaje C++ [2].

Los plugins permiten al desarrollador controlar prácticamente cualquier aspecto de Gazebo. Se recomienda su uso cuando se quiere alterar o desarrollar algún aspecto de la simulación, como puede ser el movimiento de un robot, inserción de modelos durante la simulación, creación dinámica de nuevos modelos, publicación y subscripción de mensajes en ROS, etc.

Los plugins se pueden dividir en diferentes clases, atendiendo al tipo de objeto al que estén orientados, esta clasificación también nace de las clases que proporciona Gazebo en un API para facilitar el desarrollo de plugins. Como norma general, los plugin en Gazebo se desarrollan como una clase heredad de alguna de las siguientes:

- ***WorldPlugin*:** este tipo de plugin está orientado a funcionar dentro de un elemento de tipo "world". Desde él se puede acceder a todos los modelos disponibles en el entorno y acceder a sus propiedades.
- ***ModelPlugin*:** este tipo de plugin se adhiere a objetos de tipo "model" como puede ser un robot, un coche u objetos a los que se desee dar un comportamiento específico.
- ***SensorPlugin*:** este tipo de plugin está enfocado a usarse en la inclusión y modelado de sensores, y su funcionamiento. Proporciona facilidades para obtener distancia entre objetos, adición de ruido, características visuales, etc.
- ***SystemPlugin*:** clase que se carga con el inicio de Gazebo y ofrece acceso a archivos o creación de plugins.
- ***VisualPlugin*:** da acceso a características visuales del programa.
- ***GUIPlugin*:** otorga acceso a características y funciones disponibles en la interfaz gráfica de Gazebo o teclado.

La clasificación anterior sólo identifica el plugin, pero a pesar de ello se puede emplear la Gazebo API para acceder a prácticamente cualquier aspecto deseado. Por poner un ejemplo, independientemente de si se trata de un WorldPlugin o un ModelPlugin, se puede obtener el objeto de tipo World que representa la simulación.

La API de Gazebo se divide en varios módulos orientados según el tipo de objeto o las funciones que desempeña. Tras la realización de este trabajo, los módulos que se consideran más importantes son:

- **Physics:** este módulo es el más importante, permite acceder a la mayoría de aspectos relevantes. Dentro de lo que se consideran aspectos relevantes se encuentra acceder a modelos del entorno (robots o cualquier objeto del entorno), acceso a las articulaciones del modelo y sus propiedades, acceso a cada uno de los elementos que forman un modelo (posición relativa, absoluta)
- **Events:** este módulo también se considera de relevancia, pues permite detectar eventos, como la creación o eliminación de un modelo en el entorno, detectar un nuevo paso en la simulación o detectar la puesta en pausa de esta última.

En este trabajo se desarrolla un plugin del tipo "*WorldPlugin*" empleando principalmente los módulos "*Physics*" y "*Events*" para controlar el cambio de pierna del HyReCRo.

3.2. HyReCRo

La primera tarea a la hora de realizar este trabajo fue la descripción del robot HyReCRo. El HyReCRo es un robot que contiene módulos paralelos, hecho que supone una dificultad para poder emplear los formatos de descripción indicados hasta ahora (XACRO/URDF) puesto que no permiten describir este tipo de módulos (orientado a arquitecturas paralelas y robótica móvil).

Puesto que no se van a emplear físicas durante la simulación, la fuerza y robustez por las que se emplean las cadenas cinemáticas paralelas en la realidad pierden su utilidad en simulación, siendo posible interpretar un "equivalente serie" del HyReCRo. En dicho "equivalente serie" se sustituye cada par de cadenas cinemáticas que forman una pata por una arquitectura de tipo RPR (rotación-prismática-rotación). Al sustituir cada pata por esta arquitectura, el "equivalente serie" del HyReCRo se convierte en un robot con arquitectura RPR-RR-RPR, de lo que se puede deducir que dispone de 8 GDL. A pesar de esta diferencia con el robot original, el movimiento de su "equivalente serie" es suficientemente fiel al original para el propósito de este trabajo.

Debido a la configuración de robot bípedo del HyReCRo, es necesario contar con dos archivos, dos XACRO/URDF o SDF, que describan el robot en sus dos configuraciones, con la pata A fija y con la pata B fija. La necesidad de dos archivos se debe a que el formato XACRO/URDF o SDF no contemplan robots bípedos, y como se ha comentado con anterioridad, en una articulación rotativa, el elemento hijo siempre se mueve respecto al padre, de modo que se necesitan dos archivos, cada uno para describir cuando se encuentra cada una de las patas fijas.

En el paquete "hyrecro" [17] se pueden encontrar las carpetas "urdf" y "sdf" las cuales contienen las descripciones del HyReCRo en los formatos indicados por el nombre de la carpeta, y en sus dos configuraciones, pata A fija y pata B fija. Además, en la carpeta "sdf" se pueden encontrar la descripción del robot sin sensores, o con sensores.

Para este trabajo se han empleado ambos tipos de formatos, por comodidad y ser un formato más completo, se han usado las descripciones en formato SDF para el entorno

de Gazebo, y sus equivalentes en XACRO/URDF para ROS. La función de las descripciones en XACRO/URDF es la de indicar a ROS las transformaciones entre los distintos sistemas de referencia de los elementos del robot, además de proporcionar una interfaz gráfica con deslizaderas (Figura 3.3) que permiten modificar las posiciones articulares del robot.

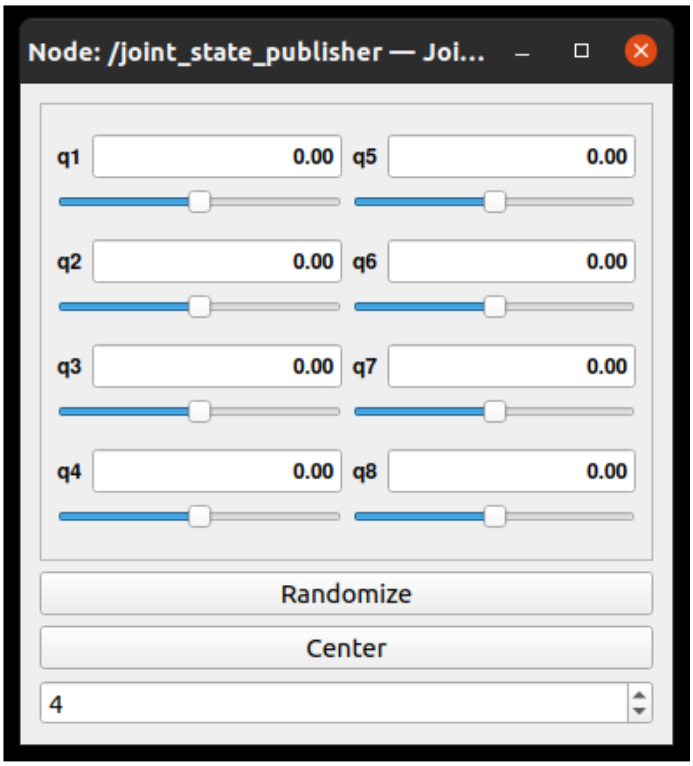


Figura 3.3. Interfaz gráfica para mover las articulaciones del HyReCRo

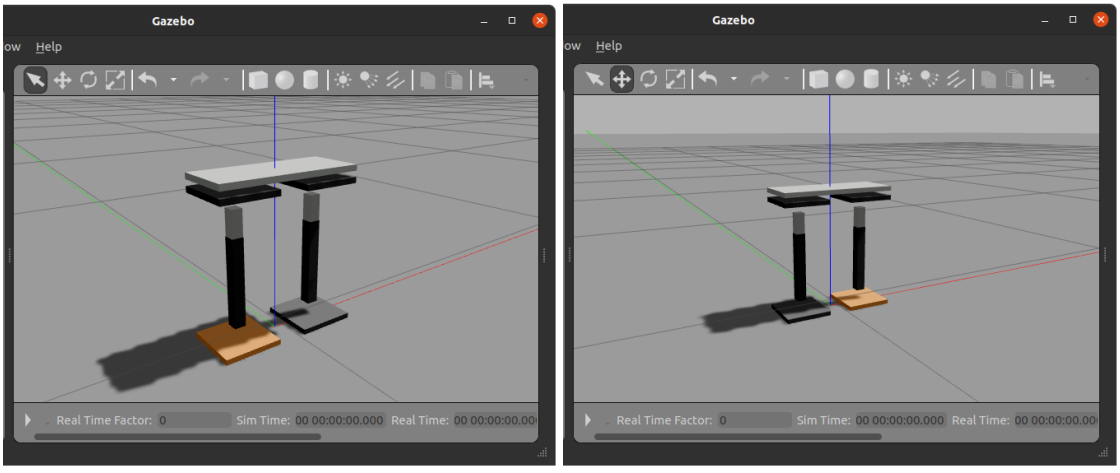


Figura 3.4. Representación gráfica de las configuraciones del HyReCRo en Gazebo

La Figura 3.4 representa gráficamente en Gazebo la descripción del robot en sus dos configuraciones, donde el color naranja en la pata indica que está fija.

3.3. Ouster OS-1

Figura 3.5. Sensor LiDAR Ouster OS1-128

Tanto la descripción del sensor Ouster OS1 como el plugin de Gazebo correspondiente para su correcto funcionamiento se han obtenido del paquete desarrollado por Wil Selby en [18]. En este paquete se puede encontrar la descripción de un sensor OS1 de 64 canales, al cual se le han configurado sus parámetros para que coincidan con los del OS1 de 128 canales. En la Figura 3.5 se observa el sensor real.

El archivo de descripción que modela el OS1 de 128 canales se encuentra disponible dentro de la carpeta "urdf" del paquete "hyrecro" en [17]. En la Figura 3.6 se muestra el sensor sobre el robot. Por reducir coste de computación gráfica se ha sustituido el modelo 3D del OS1-128 por un cilindro de las mismas dimensiones. Además, se muestra en la FIGURA un ejemplo de los datos que proporciona el sensor simulado desde RViz (herramienta de visualización en ROS).

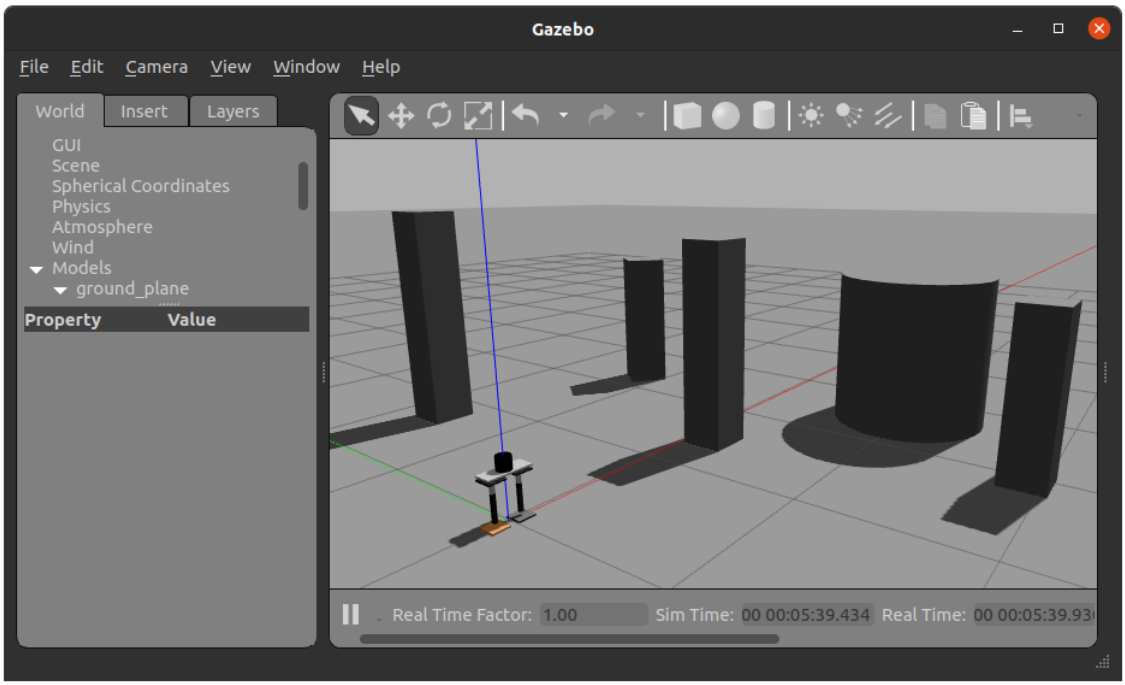


Figura 3.6. Representación gráfica del OS1-128 en sobre el HyReCRo en Gazebo

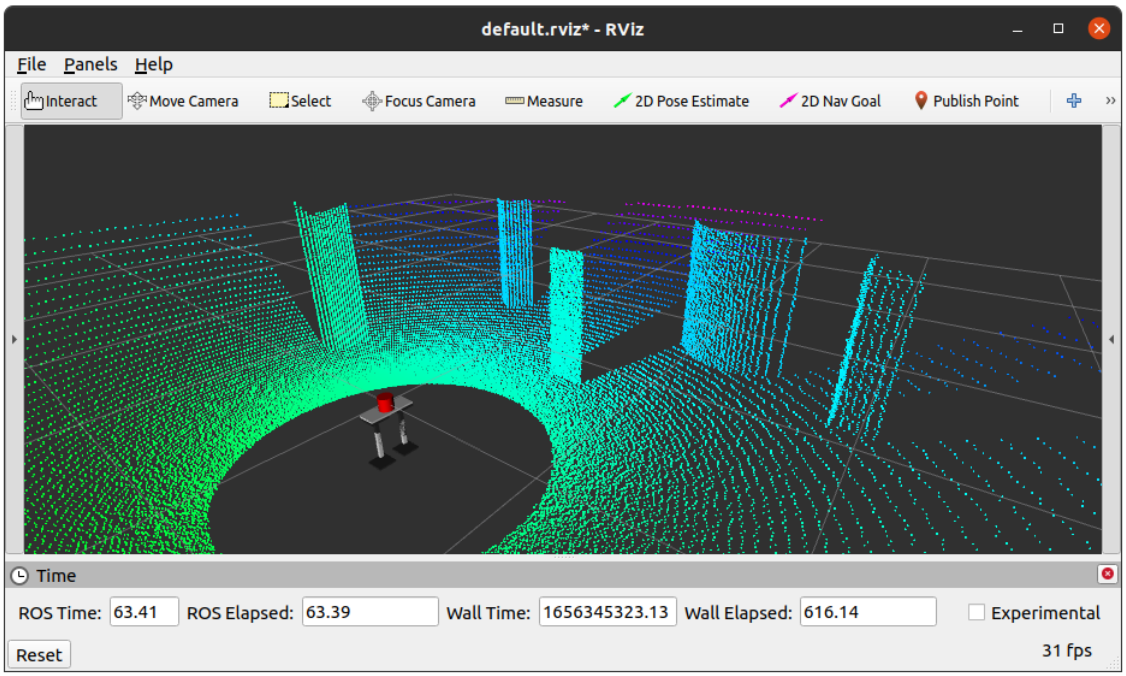


Figura 3.7. Ejemplo de nube de puntos generada por el OS1-128 simulado en RViz

3.4. RealSense d435i

Figura 3.8. Cámara RGB-D RealSense d435i

La cámara RGB-D RealSense d435i mostrada en la Figura 3.8, se ha simulado a partir de los paquetes disponibles en [19] y [20]. La simulación de este sensor es menos realista de lo que cabría esperar, puesto que a pesar de disponer el sensor de parámetros para establecer sus valores de ruido, estos no se traducen a la hora de generar la información, proporcionando unos datos sin ruido.

Puesto que se dispone del sensor en la realidad, tras varias pruebas con éste y basándose en los datos que proporciona el fabricante, se establece que las nubes de puntos tienen un ruido aceptable entre los 0.3 y los 3 metros. Con estos valores, se modifican los parámetros del sensor a simular para que genere datos entre 0.3 y 3 metros, para maximizar la similitud con el sensor real.

En este trabajo sólo se consideran los valores de profundidad que proporciona la cámara (nubes de puntos), sin tener en cuenta los valores de color del píxel, ni IMU, a

futuro se considerará si es necesario o no añadir la información de color o IMU para mejorar la navegación del entorno.

En la Figura 3.9 y se observa una imagen de la RealSense d435i sobre el HyReCRo y un ejemplo de los datos simulados sobre RViz respectivamente.

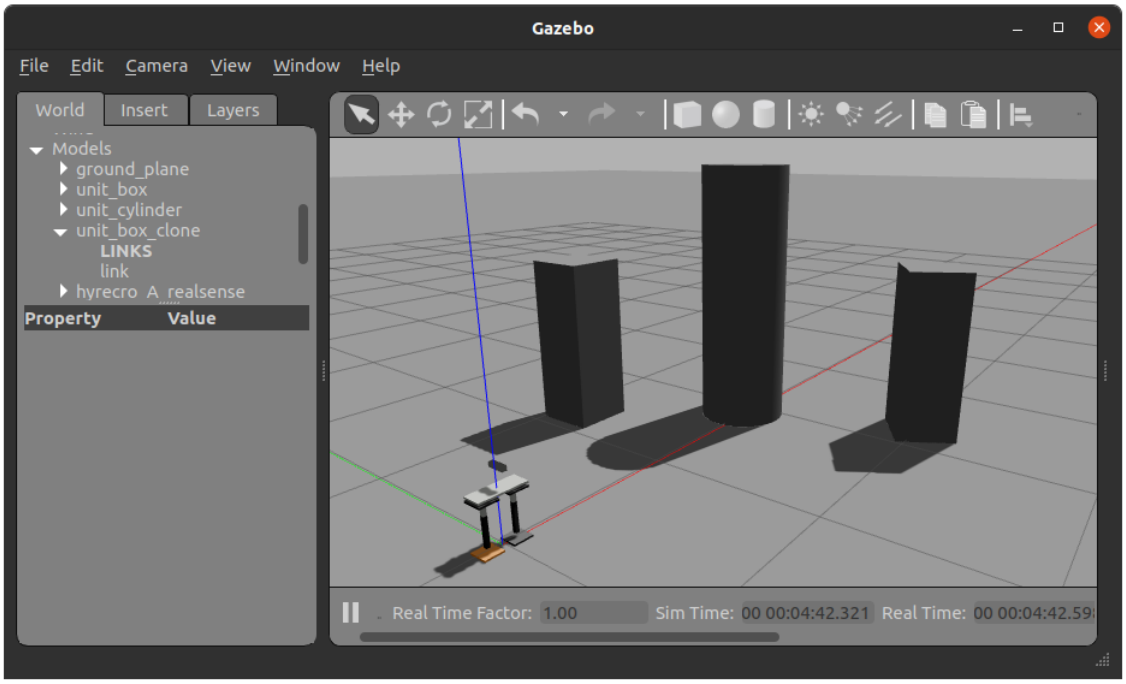


Figura 3.9. Representación gráfica de la RealSense d435i sobre el HyReCRo

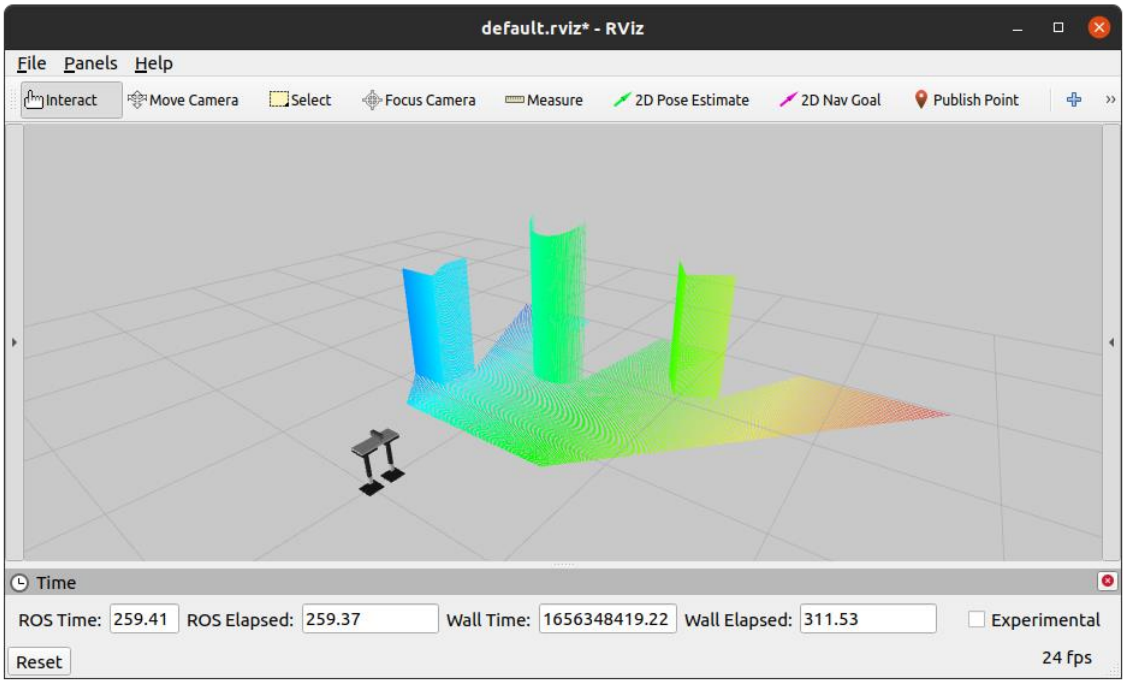


Figura 3.10. Ejemplo de nube de puntos generada por la RealSense d435i simulada

3.5. Estructura Reticular

La estructura reticular que se emplea en este trabajo se desarrolla desde el mismo entorno de Gazebo, y será la que constituya el mundo ("world") en Gazebo. El modelo de la estructura reticular se ha desarrollado desde la herramienta que incorpora Gazebo denominada "Model Editor" mediante formas geométricas simples. Una representación gráfica de la estructura en cuestión se puede observar en la Figura 3.11.

Su descripción se encuentra disponible en formato SDF en la carpeta "worlds" del paquete "hyrecro". En esta carpeta se encuentran dos versiones de la misma:

- ***reticular_GUI_plugin.world****:* describe la estructura e incorpora el plugin para controlar al HyReCRo mediante la interfaz gráfica.
- ***reticular_target_plugin.world:*** describe la estructura e incorpora el plugin para controlar al HyReCRo estableciendo posiciones articulares objetivo.

Ambos archivos son exactamente iguales salvo por sus últimas líneas, en las que se inserta el plugin correspondiente. Una descripción de estos plugins se presenta en la sección siguiente de este mismo capítulo.

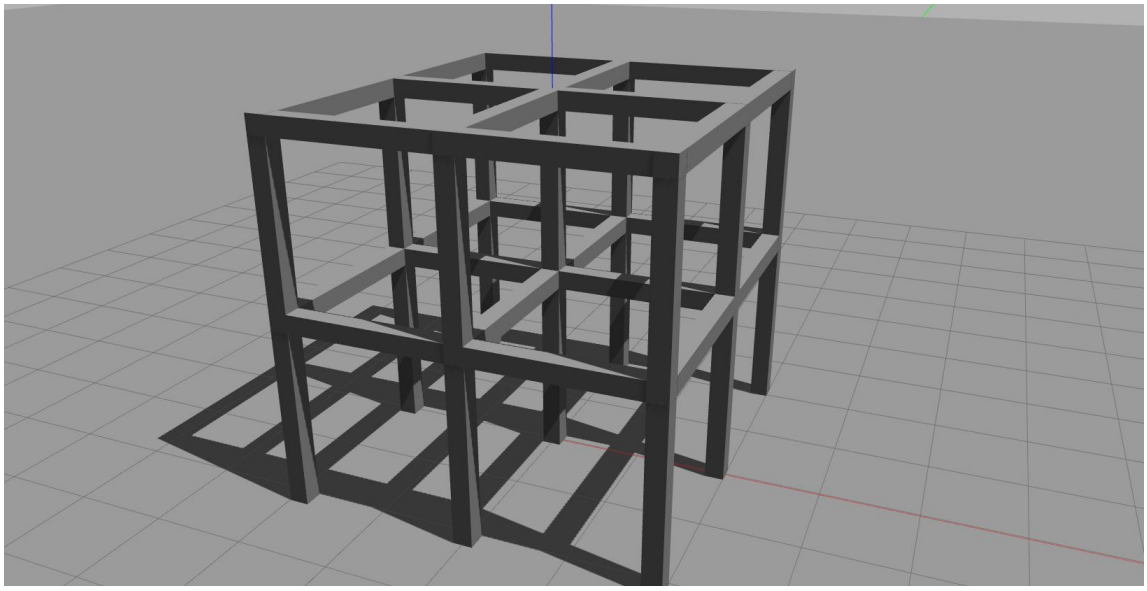


Figura 3.11. Estructura reticular simulada en Gazebo

3.6. HyReCRo plugins

El funcionamiento bípedo del robot es un aspecto que se escapa al simulador Gazebo, y por lo tanto es necesario desarrollar una nueva funcionalidad mediante un plugin que permita realizar la tarea del cambio de pata en el HyReCRo. En este apartado se detallan los principales aspectos de los plugins desarrollados para el HyReCRo.

El código fuente de estos plugins se encuentra en el paquete "hyrecro_plugins" disponible en [17].

Se han desarrollado dos plugins con sus respectivos objetivos:

- ***gazebo_ros_hyrecro_GUI.cpp:*** este plugin tiene por objetivo mover las articulaciones a partir de la interfaz gráfica mostrada en la Figura 3.3.
- ***gazebo_ros_hyrecro_targets.cpp:*** este plugin cumple la función de establecer las posiciones articulares gracias a un tópico de ROS además de publicar la posición actual de cada articulación.

Ambos plugins comparten muchos aspectos del código fuente. Su diferencia radica principalmente en el tópico al que se suscriben para actualizar las posiciones articulares.

Tras intentar varios métodos para implementar el cambio de pata mediante un plugin, como acceder al modelo e invertir sus articulaciones, se opta por una opción más sencilla en la que se reduce el número de problemas con respecto al resto de opciones que se probaron. Esta opción consiste en emular dicho cambio de pata eliminando e insertando un nuevo modelo del robot en la configuración requerida. Para cumplir con este funcionamiento, se requieren de dos modelos del robot, el "hyrecro_A_serial" y el "hyrecro_B_serial" disponibles en la carpeta "/sdf" del paquete "hyrecro" [17]. Cada uno de ellos está descrito de modo que una de sus patas (A o B) sea considerada la base fija del robot.

A priori, puede parecer necesario desarrollar un "ModelPlugin" puesto que nuestro objetivo es dar una nueva funcionalidad a un modelo, pero esta opción no es viable, ya que si el método para simular el cambio de pata se basa en eliminar un modelo e insertar otro, el plugin se eliminaría con él. Este problema se resuelve desarrollado un "WorldPlugin", cuya única diferencia con un "ModelPlugin" es que se adhiere al "world" (la estructura reticular mencionada en la sección anterior) y este último es constante durante toda la simulación.

Los plugins realizan 2 funciones principales, mover el modelo del HyReCRo a partir de mensajes de ROS y efectuar un cambio de pata cuando se indique. Para realizar el cambio de para, se dispone de un parámetro (en el "*parameter_server*" de ROS) denominado ***"hyrecro/fixed_leg"*** el cual almacena la configuración deseada del HyReCRo. Este parámetro acepta los caracteres "A" o "B" (por defecto se encuentra a "A"). Cuando se pasa de un valor "A" a un valor "B" o viceversa, los plugins se encargan de almacenar las posiciones articulares actuales del HyReCRo, eliminar el modelo viejo e insertar el nuevo modelo en la posición requerida y con los valores articulares necesarios para que no haya un cambio aparente en la simulación del robot. Gracias a este parámetro, desde cualquier terminal de Ubuntu o nodo de ROS se puede efectuar el cambio de pata de forma sencilla. Los tópicos a los que se suscribe cada plugin son ***"/joint_states"*** para el plugin "*gazebo_ros_hyrecro_GUI* " y ***"/joints_tmp_target"*** para el *"gazebo_ros_hyrecro_targets"*.

Además, el plugin *"gazebo_ros_hyrecro_targets"* publica continuamente la posición de cada articulación bajo el tópico ***"/joints_position"***.

Para simular trayectorias de movimiento del HyReCRo, se ha desarrollado un nodo denominado ***"move_simulator.py"*** para funcionar en conjunto con el plugin *"gazebo_ros_hyrecro_targets"*. Este nodo se encarga de leer las posiciones articulares

objetivo de un archivo ".csv". A su vez, se suscribe al tópico *"/joints_position"* para conocer la posición exacta de las articulaciones en cada momento. Dadas las posiciones actuales y las posiciones objetivo, se podría establecer un controlador para ir estableciendo las posiciones articulares que permiten llegar al objetivo de forma continua y suave. Dado que esto último no es el objetivo de este trabajo, y requeriría un controlador para cada articulación, se ha optado por una solución más sencilla.

La solución por la que se ha optado se basa en obtener la diferencia entre ambas posiciones articulares y dividir esta diferencia en un número elevado (100) para obtener un movimiento suave. Para simular el movimiento, se va sumando el resultado del cociente anterior a la posición actual de las articulaciones tantas veces como el valor del divisor. El valor objetivo en cada instante se publica en el tópico *"/joints_tmp_target"* de modo que el plugin es capaz de establecer la posición articular indicada en todo momento.

Comentarios y aspectos a tener en cuenta sobre el "hyrecro_plugin":

- Intentar acceder a un elemento que no existe en la simulación provoca un error.
- Es necesario identificar cuando se inserta un nuevo elemento en Gazebo para saber cuándo se puede acceder a dicho elemento. Para ello se emplea el evento `gazebo::event::Events::ConnectAddEntity()`.

3.7. Ejemplo de funcionamiento

En esta sección se muestra el procedimiento paso a paso a seguir para poner en marcha la simulación.

Inicio de la simulación

Para iniciar la simulación y controlar el HyReCRo mediante la interfaz gráfica ejecutar en una terminal de Ubuntu lo siguiente:

- `$ roslaunch hyrecro gazebo_hyrecro_GUI.launch`

En caso de querer simular el movimiento a lo largo de una trayectoria ejecutar de igual modo en una terminal de Ubuntu:

- `$ roslaunch hyrecro gazebo_hyrecro_targets.launch`

Cada uno de estos *launchfiles* se encarga de cargar el HyReCRo con su correspondiente plugin.

En la Figura 3.12 se muestra el resultado tras lanzar la simulación con el objetivo de mover las articulaciones del HyReCRo mediante la interfaz gráfica. En ella se han establecido unas posiciones articulares para corroborar que el movimiento del robot es acorde con las posiciones articulares establecidas.

Para simular el movimiento del robot a lo largo de una trayectoria, es necesario iniciar la simulación con "*gazebo_hyrecro_targets.launch"* y posteriormente lanzar el nodo "*move_simulator"* para ir publicando las posiciones articulares requeridas.

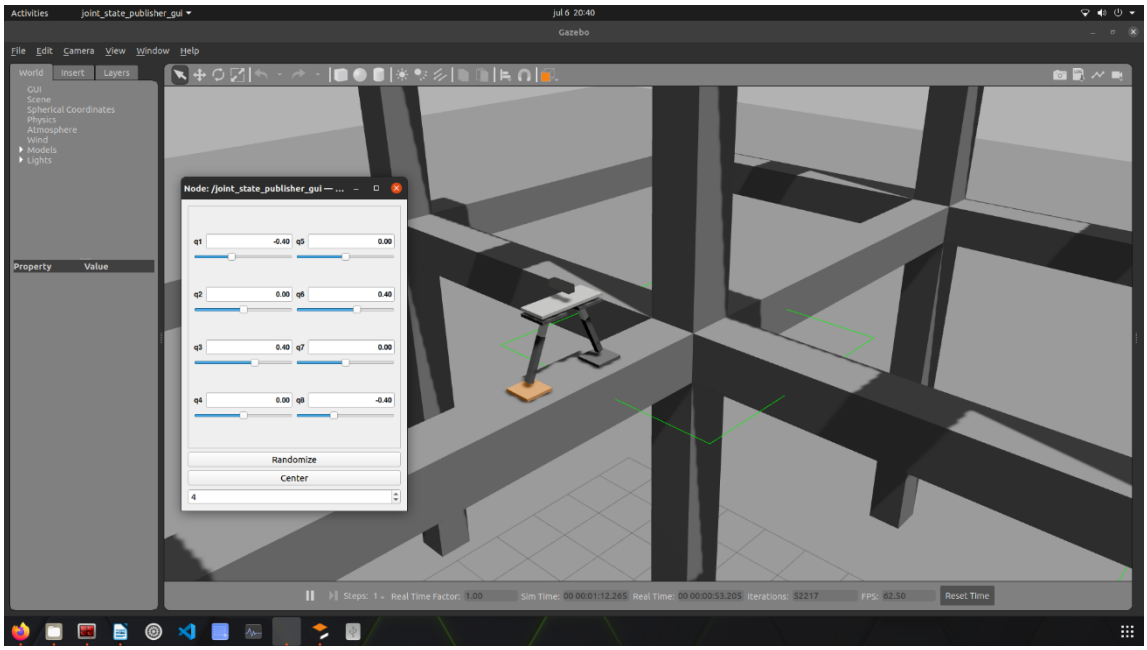


Figura 3.12. Inicio de la simulación

Cambio de pata

Para realizar un cambio de pata, es necesario indicar la pata que se desea fijar en el parámetro "hyrecro/fixed_leg". Para ello se emplea el siguiente comando:

- `$ rosparam set /hyrecro/fixed_leg B`

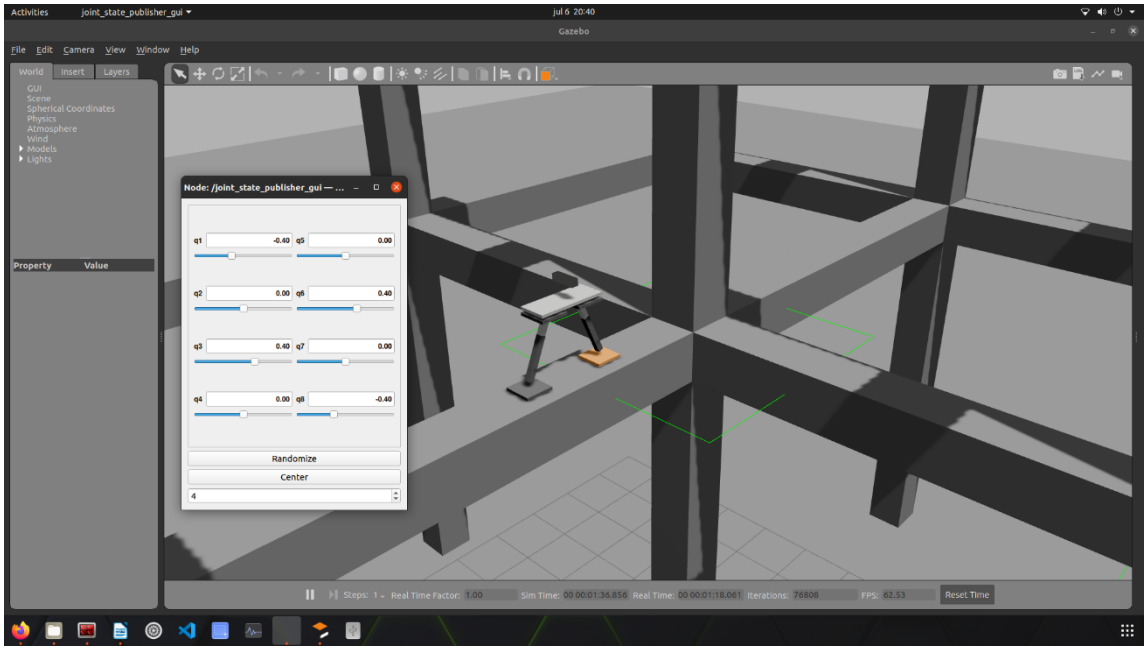


Figura 3.13. Ejemplo cambio de pata

Simular movimiento

Se puede simular el movimiento del robot de forma manual usando las deslizaderas de la Figura 3.3 o estableciendo una serie de posiciones articulares objetivo usando el nodo *"move_simulator.py"*. Este nodo lee de forma secuencial las posiciones articulares establecidas en un archivo ".csv" que se pasará como argumento a la hora de iniciar el nodo. Este archivo ha de estar ubicado en la carpeta "trajectories" del paquete "hyrecro":

- `$ rosrun hyrecro move_simulator target_1.csv`

El archivo ".csv" ha de tener una forma específica que se muestra con un ejemplo en la Figura 3.14. Además, para indicar un cambio de pata, se establece una "C" en los campos referentes a cada posición articular.

```
q1,q2,q3,q4,q5,q6,q7,q8
C,C,C,C,C,C,C,C
0.0000,0.0000,0.0000,0.0000,3.1416,0.0000,0.0000,0.0000
C,C,C,C,C,C,C,C
0.1000,0.0000,-0.4737,3.1416,3.1416,-0.8046,0.0000,-0.3968
C,C,C,C,C,C,C,C
0.0000,0.0000,0.0000,1.5733,0.0000,0.0000,0.0000,0.0000
C,C,C,C,C,C,C,C
0.7536,0.0000,-0.102,3.1416,0.0000,0.0000,0.0000,-0.9010
C,C,C,C,C,C,C,C
0.0000,0.0000,0.0000,0.0000,-1.5710,0.0000,0.0000,0.0000
C,C,C,C,C,C,C,C
0.0000,0.0000,0.0000,0.0000,0.0000,0.0000,0.0000,0.0000
C,C,C,C,C,C,C,C
0.3543,0.0000,0.8447,0.0000,0.0000,0.4102,0.0000,-0.0440
C,C,C,C,C,C,C,C
0.0000,0.0000,0.0000,3.1416,0.0000,0.0000,0.0000,0.0000
C,C,C,C,C,C,C,C
0.0000,0.0000,0.0000,3.1416,3.1416,0.0000,0.0000,0.0000
```

Figura 3.14. Ejemplo de CSV para almacenar posiciones articulares objetivo

Capítulo 4

4. Construcción de Mapas

En este capítulo, se relatan los procedimientos y algoritmos empleados durante el desarrollo de este trabajo para la construcción de mapas. Este trabajo nace como primera etapa del mapeado y navegación del robot HyReCRo mediante nubes de puntos. En esta primera aproximación al mapeado, se emplearán los métodos clásicos de registration de nubes de puntos, es decir, algoritmos basados en ICP.

Las siguientes subsecciones detallan los aspectos cubiertos durante el desarrollo del trabajo.

4.1. *Point Cloud Library* (PCL)

La PCL se trata de una librería de código abierto que se ha hecho más y más grande a lo largo de los años. Presentada por primera vez en 2011 ([21]) ha ido creciendo gracias a la comunidad y cuenta con multitud de algoritmos de filtrado, registration o segmentación disponibles en la literatura. El formato de nubes de puntos para trabajar con esta librería es *".pcd"*.

Dispone de diferentes módulos que le permiten abarcar todas las tareas posibles a la hora de trabajar con nubes de puntos. En la Figura 4.1 se muestran los principales módulos de los que dispone, cada uno con sus clases y funciones correspondientes.

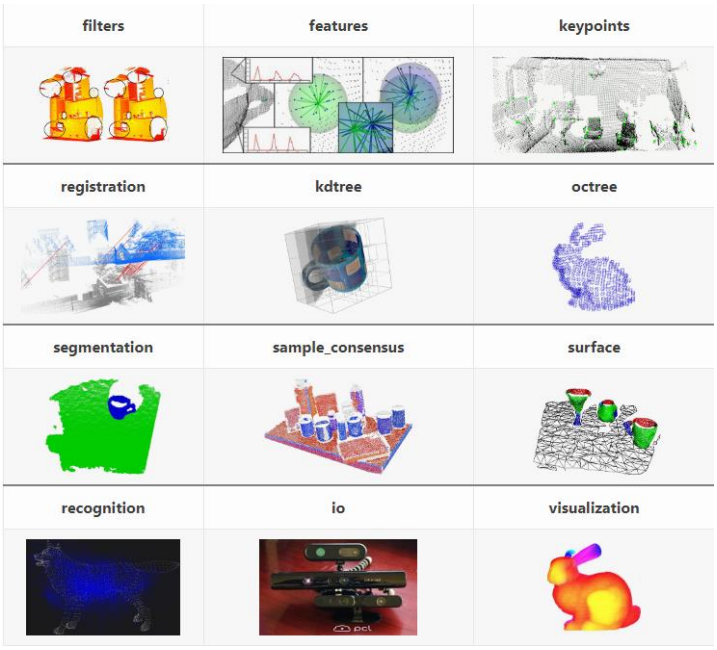


Figura 4.1. Módulos disponibles en la PCL.

Como introducción a la librería PCL, destacar que emplea un tipo de objeto general para almacenar nubes de puntos *pcl::PointCloud.* A este objeto, se le asocia un tipo de punto que define la información contenida en cada punto, ya sea solo su posición (*pcl::PointXYZ*), su posición y color (*pcl::PointXYZRGB*) o puntos con su normal asociada (*pcl::PointNormal*). Se recomienda emplear *typedef* en c++ para definir una nube de puntos junto con el tipo de punto que emplea, un ejemplo para nubes de puntos con información únicamente de la posición de los puntos se muestra a continuación:

```
typedef pcl::PointXYZ PointT;
typedef pcl::PointCloud<PointT> PointCloud;
```

Para continuar con la introducción a la librería PCL, se comentan los módulos más empleados para este trabajo.

Filters

Dentro de este módulo, que contiene clases y funciones para realizar un filtrado de las nubes de puntos, se han empleado principalmente dos filtros:

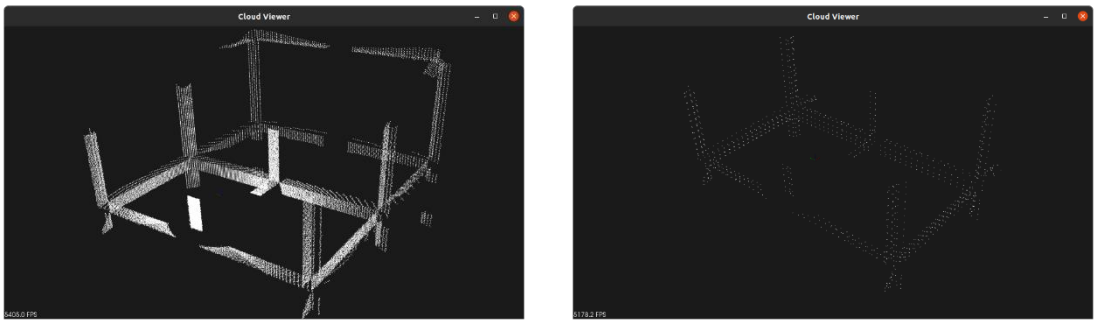
- ***pcl::PassThrought****:* se trata de un filtro paso-banda que se queda con los puntos dentro de unos límites establecidos para un eje indicado.
- ***pcl::VoxelGrid:*** este otro filtro permite reducir el número de puntos de nuestra nube. Su funcionamiento consiste en establecer un prisma cuadrangular recto, para el cual todos los puntos que se encuentren en su interior se sustituyen por un único punto.

Un ejemplo de uso de estos filtros se muestra a continuación:

```
// Filter Height
pcl::PassThrough<pcl::PointXYZ> pass;
pass.setInputCloud(inCloud);
pass.setFilterFieldName("z");
pass.setFilterLimits(-1, 1);
pass.filter(*inCloud);
// Voxel Filter
pcl::VoxelGrid<pcl::PointXYZ> sor;
sor.setInputCloud(inCloud);
sor.setLeafSize(0.1, 0.1, 0.1);
sor.filter(*inCloud);
```

En la Figura 4.2 se observa una nube de puntos antes y después de aplicarle estos filtros. Para el filtro *passthrought*, se ha elegido que solo se mantengan los datos dentro de un cubo de ± (2, 2, 1) metros para los ejes (x, y, z). Tras este filtro, se ha aplicado un *voxelgrid* de (0.1, 0.1, 0.1) metros.

Como se puede observar, la cantidad de puntos es mucho menor (41.588-1.052), lo que supone una reducción del coste computacional y mejora para los algoritmos de ICP.



a) Sin filtrar b) Filtrada

Figura 4.2. Ejemplos de filtrado en nubes de puntos.

Registration

Este módulo contiene numerosos algoritmos para solventar el problema de registration. De este módulo en concreto se ha optado por emplear los algoritmos clásicos para resolver este problema, ICP punto-punto, ICP punto-planto, Generalized-ICP.

- ***pcl::IterativeClosestPoint****:* esta clase de la librería PCL define un tipo de objeto que contiene las funciones necesarias para implementar un ICP punto a punto. Requiere de una nube de partida (*"source"*) y una nube objetivo (*"target"*). Este algoritmo buscará una transformación para alinear la nube *source* con la nube *target.*
- ***pcl::IterativeClosestPointWithNormals:*** esta clase es un caso especial de la anterior. Esta clase las funciones necesarias para aplicar un ICP basado en distancias de punto-plano. De igual modo que la anterior requiere de un *source* y un *target*, pero en este caso, deberán ser nubes de puntos que contengan información de sus normales.
- ***pcl::GeneralizedIterativeClosestPoint:*** clase con el mismo funcionamiento que las anteriores pero que emplea el algoritmo propuesto en [10] para aplicar un ICP con distancias plano-plano.

A la hora de declarar objetos de este tipo, es recomendable también indicar el tipo de punto que emplean tanto el *source* como el *target.* Esto se ejemplifica en la siguiente línea:

```
pcl::IterativeClosestPoint<PointXYZ, PointXYZ> icp;
```

Visualization

La librería PCL contiene su propio módulo para poder visualizar nubes de puntos. Dentro de este módulo encontramos varias clases que permiten mostrar las nubes de puntos por pantalla, las más destacables son:

- ***pcl::visualization::CloudViewer:*** esta clase es la más sencilla para visualizar las nubes de puntos, pero tan sólo es capaz de representar la posición

de los puntos, sin tener en cuenta sus normales, color, etc.

- ***pcl::visualization::PCLVisualizer:*** esta clase es la más completa para visualizar nubes de puntos, permite entre otras opciones representar el color de los puntos, representar sus normales o modificar su tamaño y forma.

En la Figura 4.3 se muestra un ejemplo del uso de la clase *pcl::PCLVisualizer* para mostrar en colores distintos los planos detectados en la nube de puntos.

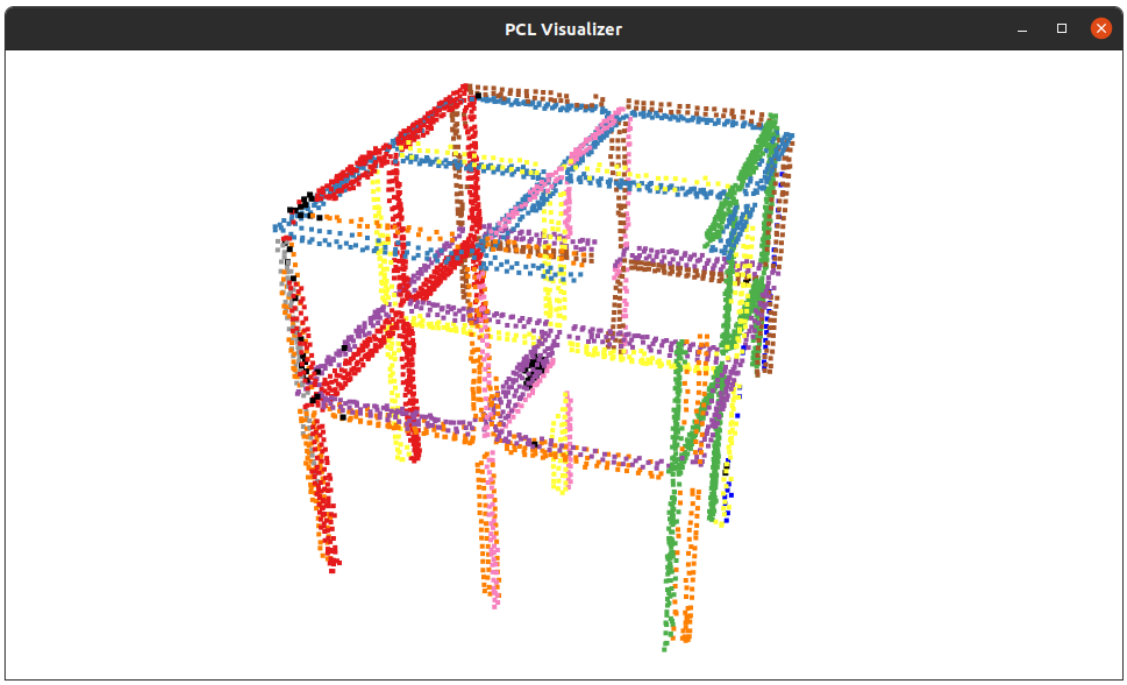


Figura 4.3. Ejemplos de PCLVisualizer.

4.2. Paquete en ROS

Para una mayor organización, la construcción de mapas y programas desarrollados para el tratamiento de nubes de puntos se ha localizado en un paquete específico de ROS. Este paquete se denomina **"hyrecro_pcl"** y se encuentra disponible en [17].

Dentro de este paquete, se encuentran un conjunto de nodos desarrollados en C++ dado que la API de la PCL se encuentra disponible en este lenguaje. Este paquete contiene tres aspectos importantes que se desarrollan a continuación:

4.2.1. Nodos

El paquete hyrecro_pcl contiene una serie de nodos que se detallan a continuación:

offline_mapping.cpp

Este nodo es el encargado de la construcción del mapa a partir de un conjunto de nubes de puntos. Se encarga de leer los archivos correspondientes a las nubes de puntos e

ir aplicando el algoritmo indicado para obtener una transformación entre la nube de puntos actual y la anterior. Además publica en ROS un tópico con el mapa.

Este nodo acepta tres parámetros en su respectivo orden:

- **Ruta:** ruta completa hasta la carpeta donde se encuentren las nubes de puntos en ".pcd". (ej. "*/home/arvc/hyrecro_ws/src/hyrecro_pcl/dataSets/set0/Filtered"*)
- **ICP_method:** mediante este parámetro se puede establecer el algoritmo de ICP que se desea usar de entre los tres implementados hasta el momento ("punto_punto"; "punto_plano"; "plano_plano").
- **Numero de archivos:** requiere que se indique el número de nubes de puntos disponibles en la carpeta.

Para el correcto funcionamiento del nodo, los archivos .pcd han de tener una nomenclatura concreta, han de tener la forma "data#.pcd" donde "#" será un entero el cual indica el orden en el que se han tomado las nubes de puntos.

El nodo realiza la construcción del mapa realizando transformaciones relativas entre dos nubes de puntos consecutivas e integrando estas transformaciones en una transformada global que indica la rotación y traslación de cada nube de puntos respecto de la toma inicial.

El nodo contiene un bucle en el que se van leyendo las nubes de puntos de dos en dos. Se indica la primera nube de puntos como *target* y la segunda como *source* de modo que las nubes de puntos se transforman a la nube anterior. Tras obtener la información de ambas nubes de puntos se les aplica un filtrado tanto de *PassThrought* como *VoxelGrid*. En un futuro se añadirán argumentos para poder modificar los parámetros de estos filtros sin necesidad de compilar el código.

Aplicar el algoritmo de ICP devuelve una matriz de transformación homogénea (MTH) que transforma el *source* al sistema de referencia del *target*. La MTH que devuelve el algoritmo ICP es una transformada relativa, para obtener la transformada global basta con ir multiplicando la MTH relativa por una MTH global que pasa del sistema del *target* a la base. En la primera iteración del bucle la MTH global es igual a la identidad. La actualización de la MTH global se realiza mediante las siguientes líneas de código.

```
// TRANSFORMATION
    globalMth = globalMth * tmpMth;
```

Una vez disponible la transformada entre la nube de puntos *source* y la nube inicial, se aplica esta transformada a la nube de puntos *source* y se añade a la nube de puntos que contiene el mapa. Para evitar puntos duplicados, tras añadir la nube de puntos al mapa se realiza un *VoxelGrid* para eliminar puntos duplicados.

A medida que se va construyendo el mapa, el mismo se va publicando en el tópico **"/map"** de ROS, de este modo se puede observar desde RViz como se va construyendo el mapa.

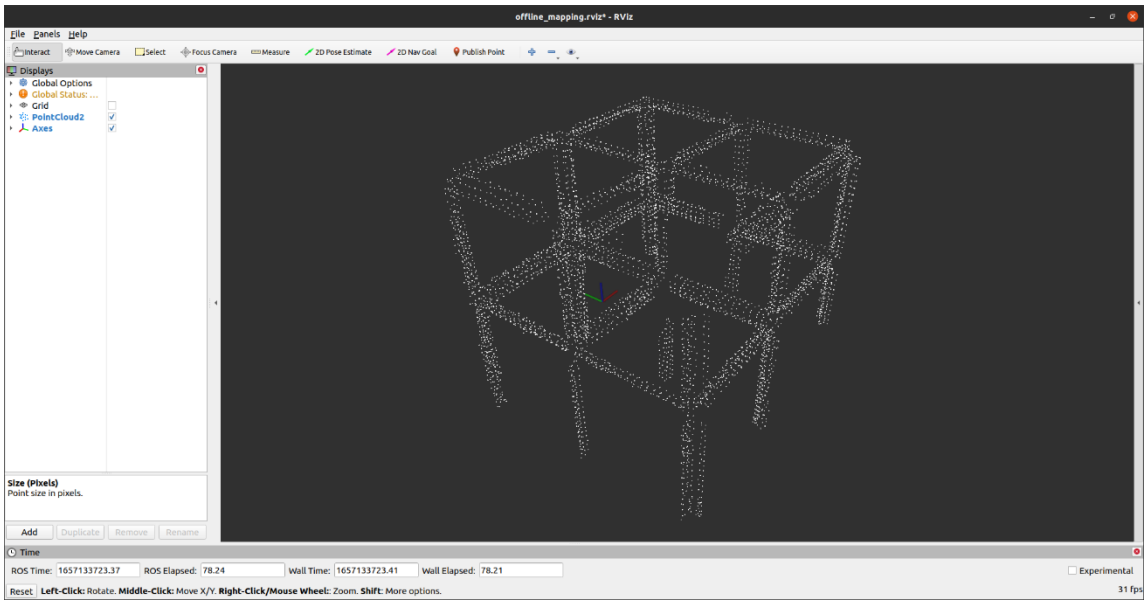


Figura 4.4. Mapa construido con offline_mapping.

show_planes.cpp

Este nodo lee un archivo ".pcd" que se introduce como argumento, y muestra por pantalla la nube de puntos con la segmentación de sus planos en distintos colores. Un ejemplo se muestra en la Figura 4.5. Los puntos en negro son puntos que no se han asociado a ningún plano en concreto.

Este nodo acepta tres parámetros de entrada:

- **Online:** parámetro booleano para decidir si la nube de puntos se ha de leer de un tópico o de un archivo, un valor verdadero indica lectura de un tópico en tiempo real. La segmentación de los planos en tiempo real aun no esta totalmente operativa.
- **Tópico:** indica el tópico donde se publica la nube de puntos a la que aplicar la segmentación en caso de estar activado el modo online.
- **Archivo:** en caso de estar desactivado el modo online, indica la ruta completa al archivo .pcd que se desea segmentar.

save_PCD.cpp

Este nodo permite almacenar de forma manual las nubes de puntos que se van generando en el formato necesario para el nodo *offline_mapping*. Requiere dos parámetros:

- **Tópico:** nombre del tópico donde se está publicando la nube de puntos.
- **Ruta:** ruta completa a la carpeta donde se desean almacenar los archivos .pcd.

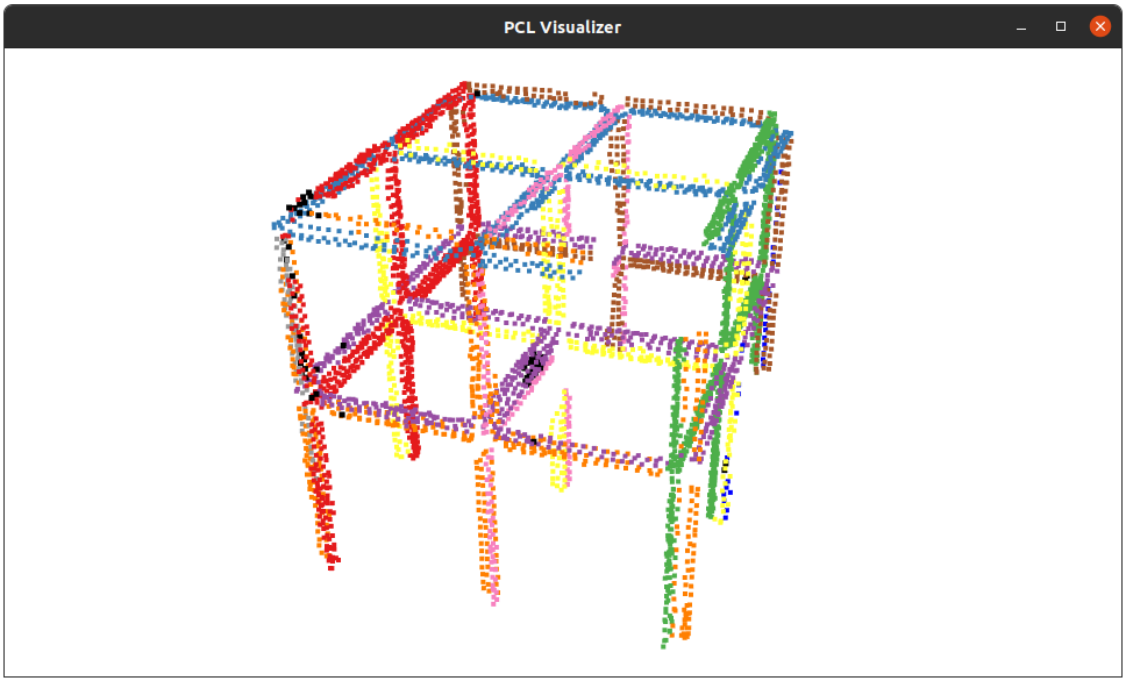


Figura 4.5. Segmentación de planos con show_planes.

Existen otros nodos que ya se están desarrollando pero contienen ciertos errores de por lo que aún no están completamente operativos y no se han incluido como parte de la memoria. A medida que esté operativos se irán actualizando en el repositorio disponible en [17].

- ***online_mapping:*** construcción del mapa en línea a partir de nubes de puntos publicadas en ROS.
- ***filter_pcd:*** nodo que permite aplicar filtros a una determinada nube de puntos almacenada en un archivo ".pcd" con propósitos de seleccionar el mejor filtrado.
- ***save_clouds:*** nodo para guardar las nubes de puntos publicadas en ROS de forma automática en archivos ".pcd" para su posterior tratamiento.

4.2.2. Launchfiles

Para una mayor facilidad de uso y evitar la necesidad de compilar el código cada vez que se desee realizar una prueba con una modificación, cada nodo acepta una serie de argumentos que le permiten modificar su funcionamiento. Cada nodo tiene un asociado un *launchfile* de ROS desde el cual se pueden indicar sus argumentos de forma sencilla.

Por lo tanto se disponen de los *launchfiles:*

- ***offline_mapping.launch***
- ***show_planes.launch***

- ***save_PCD.launch***

Un aspecto adicional del *offline_mapping.launch* es que lanza a su vez RViz desde donde se puede observar cómo se va construyendo el mapa.

4.3. Resultados

En este apartado se evalúan los resultados obtenidos con los tres algoritmos de ICP que se han usado. Para evaluar los algoritmos se ha realizado un experimento en el que se realiza una trayectoria con el HyReCRo (Figura 4.6) y se obtienen los datos simulados del Ouster OS-1.

De esta trayectoria se han captura 75 nubes de puntos con una diferencia de pose entre captura y captura reducida. Cuanto menor sea el movimiento entre captura y captura de las nubes de puntos mejor será el funcionamiento de los algoritmos de ICP.

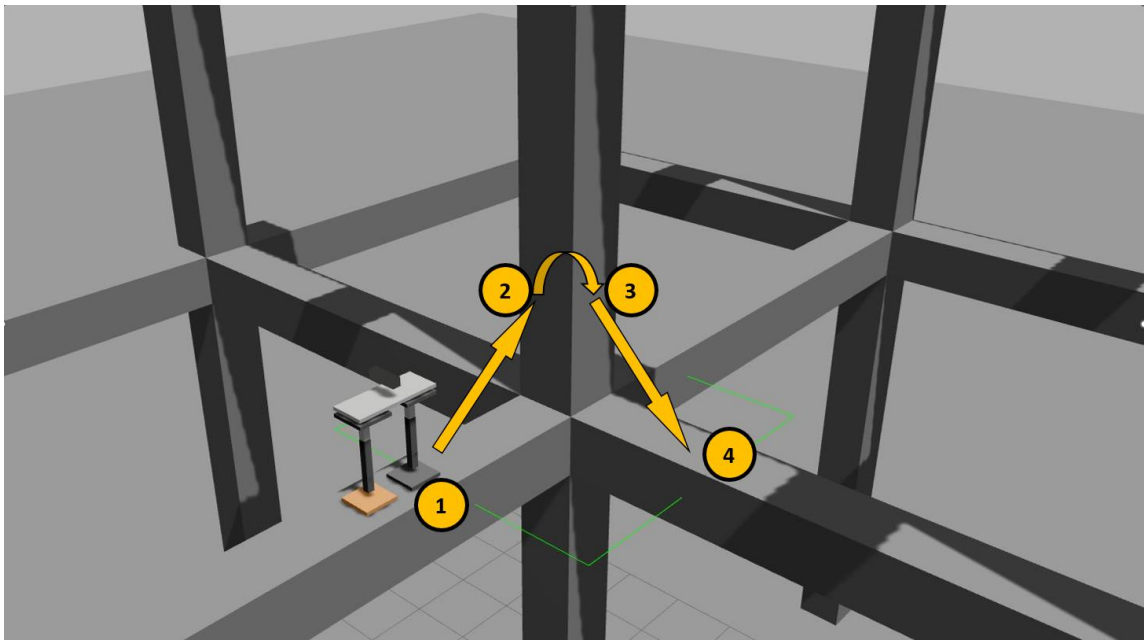


Figura 4.6. Trayectoria realizada en el experimento.

La estructura de la que se han realizado los mapas se encuentra disponible en la Figura 3.11 y los mapas generados para cada uno de los algoritmos usados, ICP punto-punto, ICP punto-plano, ICP plano-plano, se pueden observar en la Figura 4.7, Figura 4.8 y Figura 4.9 respectivamente.

De forma visual se puede observar, como el mapa generado es de mejor calidad en el Generalized-ICP (plano-plano)(Figura 4.9) que en el resto de los algoritmos. Adicionalmente el ICP punto-plano (Figura 4.8) también ofrece mejores resultados que el ICP con distancia punto-punto (Figura 4.7).

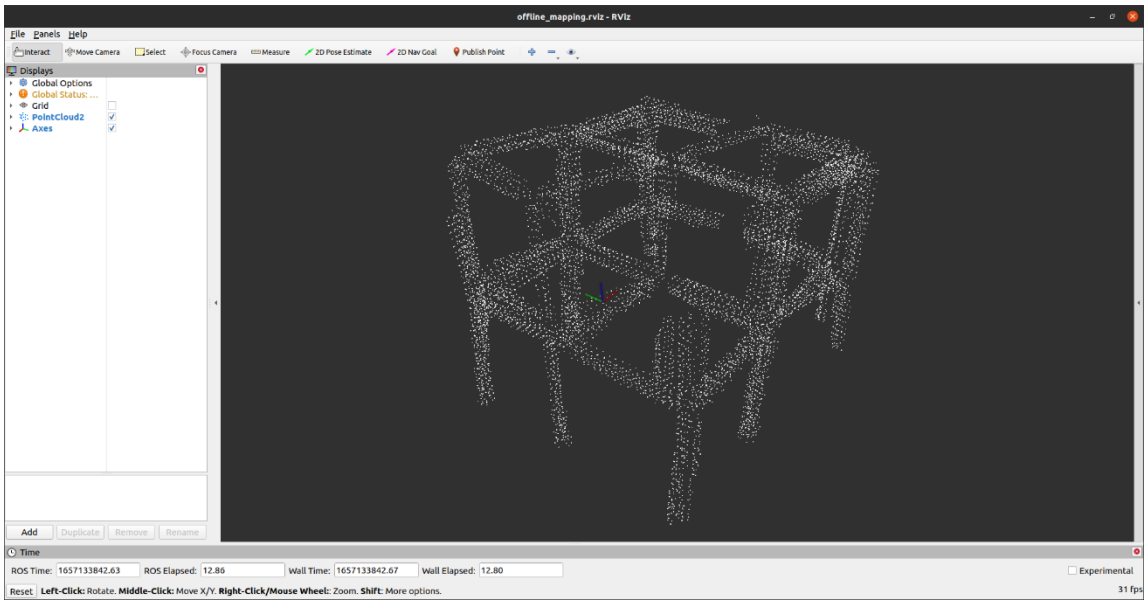


Figura 4.7. Mapa construido con ICP punto-punto.

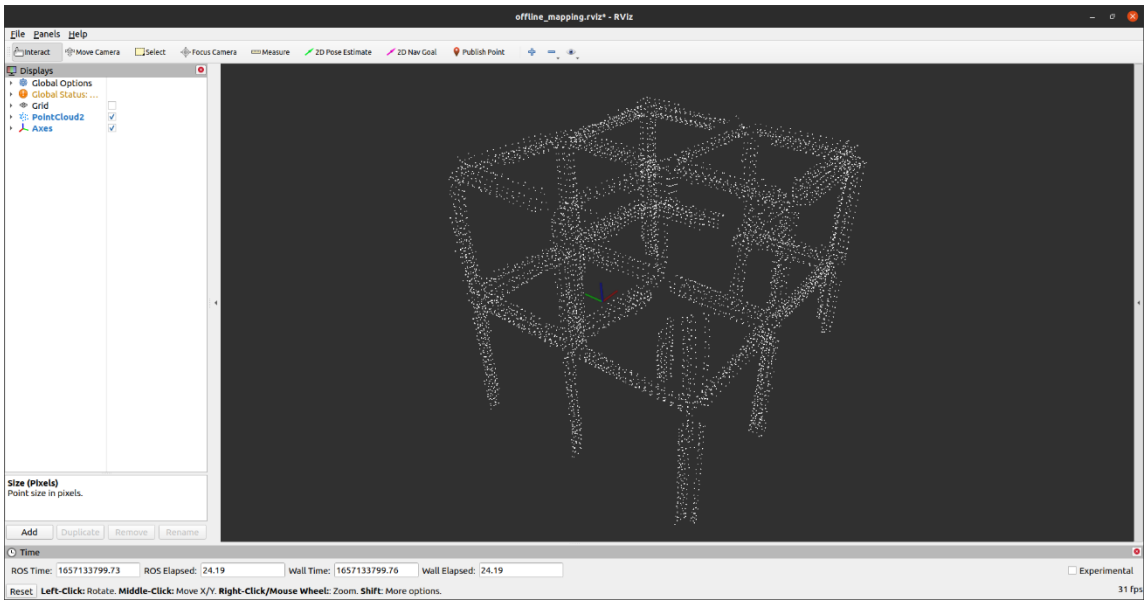


Figura 4.8. Mapa construido con ICP punto-plano.

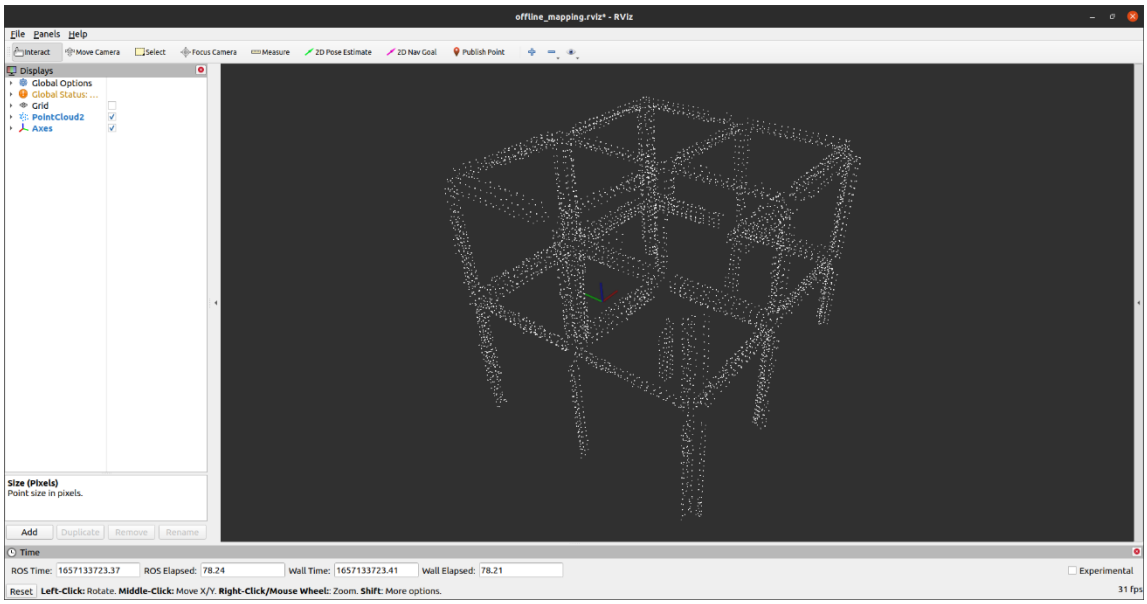


Figura 4.9. Mapa construido con ICP plano-plano.

Los objetos de tipo ICP, devuelven un valor (*Fit Score*) de bondad en la alineación de las nubes de puntos, pero este valor no se corresponde con la realidad. Para obtener una comparación más extensa sobre la bondad de estos algoritmos, se analiza también el tiempo de ejecución de cada uno de ellos, además de una medida de error que consiste en el conocimiento previo de la estructura. De la estructura y el trayecto realizado durante la toma de los datos se conoce que el mapa debería estar compuesto por once planos. Dado que debe estar formada por once planos, aquellos puntos que se encuentren fuera de estos planos son error en la transformada, por lo que se puede tomar como medida del error el número de puntos que no pertenece a ninguno de los once planos.

Tabla 4-1. Resultados de los algoritmos de ICP

Algoritmo	Tiempo requerido (ms)	Fit Score	Medida de error propuesta
ICP punto-punto	8.098	0.00398	2.609
ICP punto-plano	7.091	0.04569	940
ICP plano-plano	45.783	0.00434	37

Los resultados se muestran en la Tabla 4-1, donde se puede observar como el tiempo requerido para el ICP plano-plano es cinco veces mayor que el algoritmo ICP punto-punto. A pesar de ello, la duración del trayecto que realiza el robot se estima que requiere de un tiempo mayor de modo que el tiempo no parece ser un problema a priori.

Por otro lado, también se observa como la medida del error es mucho menor en ICP plano-plano plano, lo que le convierte en el algoritmo con mejores resultados en cuanto a precisión en la reconstrucción. En un futuro será necesario ejecutar más pruebas a estos algoritmos, variando sus parámetros para obtener mejores resultados y simular la ejecución de estos algoritmos en tiempo real para poder decidir cual tiene un mejor funcionamiento.

Capítulo 5

5. Conclusiones y trabajos futuros

El presente capítulo constituye la conclusión y propuestas de trabajos futuros aplicados a este TFM. El objetivo era desarrollar mapas de navegación a partir de sensores de rango, LiDAR o cámaras RGB-D en entornos de simulación previamente a la puesta en marcha del robot. En primer lugar, se ha desarrollado una simulación funcional del robot HyReCRo, en la que se pueden modificar las articulaciones mediante una interfaz gráfica o establecer posiciones articulares objetivo a las que se desplazará el robot. Además, la simulación incluye un método para emular el cambio de pata dentro de Gazebo.

En segundo lugar, se han desarrollado una serie de nodos que permiten aplicar distintos algoritmos de *registration* de nubes de puntos dentro de la plataforma ROS. Con estos algoritmos se ha desarrollado una primera toma de contacto a la construcción de mapas con nubes de puntos con resultados satisfactorios.

5.1. Conclusiones

Algunos comentarios adicionales sobre el desarrollo de la simulación y la construcción de mapas se detallan a continuación.

Sobre la simulación, cabe destacar la falta de información sobre como desarrollar nuevos plugins en Gazebo y su dificultad tanto de desarrollo (C++) como de depuración, debido a que en muchas ocasiones el código compilaba sin problemas y se producían errores durante su ejecución sin ningún tipo de información sobre la procedencia del error. A pesar de ello, la API de Gazebo es muy extensa y ofrece muchas posibilidades que se pueden explotar para mejorar la simulación y la fidelidad de esta. Debido a los errores durante la ejecución de programas y la falta de información sobre el desarrollo de plugins en Gazebo, esta parte del trabajo ha sido la que más tiempo de desarrollo ha acarreado.

Por otro lado, la aplicación de los algoritmos de *registration* y desarrollo de nodos para la construcción de mapas ha sido un proceso más sencillo, a pesar de la necesidad de usar C++ para ello. En este caso existe mucha más información para ello y una serie de tutoriales y guías de gran ayuda. De los algoritmos puestos a prueba, se considera como más efectivo el Generalized-ICP, que intenta minimizar la distancia plano-plano. A pesar de ello, se considera importante realizar más pruebas y modificar parámetros de los algoritmos para intentar mejorar la precisión de los algoritmos y reducir su tiempo. Lamentablemente, el tiempo que se ha dedicado a este apartado se ha visto afectado por el anterior y no se ha podido profundizar y realizar más experimentos con los algoritmos para la construcción de mapas.

Gracias a este trabajo se han obtenido nuevas competencias, para el desarrollo de aplicaciones en simulación en Gazebo y una primera aproximación a la construcción de mapas de forma satisfactoria.

5.2. Trabajos futuros

Existen numerosas posibilidades de trabajos futuros relacionados con este TFM.

En el apartado de simulación:

- Mejorar la fidelidad de la simulación de la cámara RGB-D RealSense d435 añadiendo ruido a la nube de puntos generada en función de la distancia a la cámara.
- Mejorar la fidelidad de la simulación del HyReCRo añadiendo la simulación de las físicas. Posible eliminación de la necesidad de emplear dos modelos del HyReCRo.
- Intentar implementar cadenas cinemáticas en Gazebo, gracias a emplear el formato de descripción SDF en lugar de URDF.

En el apartado de la construcción de mapas:

- Testear y modificar los parámetros de los algoritmos para mejorar su rendimiento.
- Completar el desarrollo de nodos para construcción de mapas en tiempo real.
- Desarrollo de nodos que publique bajo un tópico los coeficientes de los planos detectados en la escena.
- Desarrollo de algoritmos en dos pasos, estimación de una transformación inicial partiendo de la extracción de planos de la escena y una posterior ejecución de ICP.
- Prueba de nuevos algoritmos para *registration* de nubes de puntos.

Bibliografía

[1] A. Peidró, Kinematic Analysis and Design of the HyReCRo Robot: a Serial-Parallel and Redundant Structure-Climbing Robot, 2018.

[2] Gazebo Sim, «Gazebo Namespace Reference,» [En línea]. Available: http://osrf-distributions.s3.amazonaws.com/gazebo/api/11.0.0/namespacegazebo.html.

[3] R. B. Rusu y S. Cousins, «Point Cloud Library,» Mayo 2011. [En línea]. Available: https://pointclouds.org/documentation/.

[4] A. Hajeer, L. Chen y E. Hu, «Review of Classification for Wall Climbing Robots for Industrial Inspection Applications,» de *2020 16th IEEE International Conference on Automation Science and Engineering (CASE)*, 2020.

[5] R. Saltaret, R. Aracil, Ó. Reinoso, J. M. Sabater y M. Almonacid, «Parallel Climbing Robots for Construction, Inspection and Maintenance,» de *16th IAARC/IFAC/IEEE INTERNATIONAL SYMPOSIUM ON AUTOMATION AND ROBOTICS IN CONSTRUCTION (ISARC'99)*, Madrid, 1999, pp. 359-367.

[6] D. Schmidt y K. Berns, «Climbing robots for maintenance and inspections of vertical - A survey of design aspects and technologies,» *Robotics and Autonomous Systems*, vol. 61, nº 12, pp. 1288-1305, 2013.

[7] A. Peidró, M. Tavakoli, J. M. Marín y Ó. Reinoso, «Design of compact switchable magnetic grippers for the HyReCRo structure-climbing robot,» *Mechatronics*, vol. 212, nº 59, p. 199, 2019.

[8] R. B. Rusu, «Semantic 3D Object Maps for Everyday Manipulation,» *KI - K¨unstliche Intelligenz*, vol. 24, nº 4, pp. 345-348, 2010.

[9] B. Bellekens, V. Spruyt, R. Berkvens y M. Weyn, «A Survey of Rigid 3D Pointcloud Registration Algorithms,» de *AMBIENT 2014 : The Fourth International Conference on Ambient Computing, Applications, Services and Technologies*, 2014.

[10] A. Segal, D. Haehnel y S. Thrun, «Generalized-ICP,» de *Proceedings of Robotics: Science and Systems*, Seattle, 2009.

[11] W. S. Grant, R. C. Voorhies y L. Itti, «Finding Planes in LiDAR Point Clouds for Real-Time Registration,» de *2013 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, Tokyo, 2013.

[12] J. Xiao, B. Adler y H. Zhang, «3D Point Cloud Registration Based on Planar Surfaces,» de *2012 IEEE International Conference on Multisensor Fusion and Integration for Intelligent Systems (MFI)*, Hamburg, 2012.

[13] K. Favre, M. Pressigout, E. Marchand y L. Morin, «A Plane-based Approach for Indoor Point Clouds Registration,» de *ICPR 2020 - 25th International Conference on Pattern Recognition*, Italia, 2021.

[14] ROS, «ROS XACRO,» [En línea]. Available: https://github.com/ros/xacro/wiki. [Último acceso: 26 Junio 2022].

[15] ROS, «ROS URDF/XML,» [En línea]. Available: http://wiki.ros.org/urdf/XML. [Último acceso: 26 Junio 2022].

[16] Open Source Robotics Foundation, «SDF Format - Specification,» [En línea]. Available: http://sdformat.org/spec. [Último acceso: 26 junio 2022].

[17] F. J. Soler, «Urwik/hyrecro_simulation,» [En línea]. Available: https://github.com/Urwik/hyrecro_simulation. [Último acceso: 27 Junio 2022].

[18] W. Selby, «GitHub - wilselby/ouster_example,» [En línea]. Available: https://github.com/wilselby/ouster_example. [Último acceso: 26 Junio 2022].

[19] I. Alvarado, «issaiass/realsense2_description,» [En línea]. Available: https://github.com/issaiass/realsense2_description. [Último acceso: 27 Junio 2022].

[20] I. Alvarado, «issaiass/realsense_gazebo_plugin,» [En línea]. Available: https://github.com/issaiass/realsense_gazebo_plugin. [Último acceso: 27 Junio 2022].

[21] R. B. Rusu y S. Cousins, «3D is here: Point Cloud Library (PCL),» de *IEEE International Conference on Robotics and Automation (ICRA)*, Shangai, 2011.

[22] Gazebo Sim, «Tutorial : Plugins 101 - Gazebo,» [En línea]. Available: https://classic.gazebosim.org/tutorials?tut=plugins_hello_world&cat=write_plugin. [Último acceso: 26 Junio 2022].

[23] «Point Cloud Library,» [En línea]. Available: https://pointclouds.org/. [Último acceso: 30 06 2022].