

UNIVERSIDAD DEL VALLE DE GUATEMALA
Facultad de Ingeniería



**Integración de una computadora central en el Rover UVG
para la ejecución de ROS**

Trabajo de graduación presentado por Diego Gerardo Mencos Caal para
optar al grado académico de Licenciado en Ingeniería Mecatrónica

Guatemala,

2022

UNIVERSIDAD DEL VALLE DE GUATEMALA
Facultad de Ingeniería



**Integración de una computadora central en el Rover UVG
para la ejecución de ROS**

Trabajo de graduación presentado por Diego Gerardo Mencos Caal para
optar al grado académico de Licenciado en Ingeniería Mecatrónica

Guatemala,

2022

Vo.Bo.:

(f) _____
Dr. Luis Alberto Rivera Estrada

Tribunal Examinador:

(f) _____
Dr. Luis Alberto Rivera Estrada

(f) _____
(f) _____

Fecha de aprobación: Guatemala,

Prefacio

La idea del presente trabajo que lleva como título “Integración de una computadora central en el Rover UVG para la ejecución de ROS” es establecer las bases para llegar a desarrollar proyectos con ROS cada vez más complejos dentro del entorno de la robótica. Debido a que es una herramienta realmente poderosa e innovadora. Sentando las bases para futuras experimentaciones con el Rover UVG y otros robots dentro de la Universidad del Valle de Guatemala.

Índice

Prefacio	v
Lista de figuras	x
Lista de cuadros	xi
Resumen	xiii
Abstract	xv
1. Introducción	1
2. Antecedentes	3
2.1. Ejemplos de sistemas robóticos basados en ROS	3
2.1.1. Ryerson University Rover	3
2.1.2. Robot móvil autónomo basado en ROS	4
2.2. Proyectos de robótica en la Universidad del Valle de Guatemala	5
2.2.1. Robot Explorador Modular	5
2.2.2. Simulaciones de Robótica de Enjambre con ROS	5
3. Justificación	7
4. Objetivos	9
4.1. Objetivo general	9
4.2. Objetivos específicos	9
5. Alcance	11
6. Marco teórico	13
6.1. Robotic Operating System (ROS)	13
6.2. Gazebo	14
6.3. Rviz	15
6.4. Lenguaje de programación XML	15
6.5. Formato unificado de descripción para robots (URDF)	16

6.6.	Formato de descripción para simulación (SDF)	16
6.7.	Sensores comúnmente utilizados para robots móviles	17
6.7.1.	IMU	17
6.7.2.	GPS	17
6.7.3.	Módulos DWM1001	17
6.7.4.	LIDAR	18
7.	Entorno de Robot Operating System (ROS)	19
7.1.	Instalación de ROS 2	19
7.2.	Creación de paquetes y nodos	20
8.	Simulación por medio de Gazebo	23
8.1.	Descripción del robot	23
8.1.1.	Modelo URDF	23
8.1.2.	Modelo SDF	26
8.2.	Odometría y localización del robot	27
8.3.	Control punto a punto	29
8.3.1.	Controlador PID	29
8.3.2.	Pruebas de control punto a punto en el mundo virtual	29
8.4.	Simultaneous Localization And Mapping (SLAM)	31
8.4.1.	Configuración del sensor LIDAR en simulación	31
8.4.2.	Configuraciones necesarias para SLAM	33
8.4.3.	Pruebas de SLAM	34
9.	Implementación Física	37
9.1.	Selección de la computadora central adecuada	37
9.2.	Módulos de percepción y actuación dentro del Rover UVG	38
9.3.	Odometría y localización del robot físico	39
9.3.1.	Odometría con encoders	39
9.3.2.	Localización con sensores DWM1001 y Optitrack	39
9.3.3.	Pruebas de odometría y localización	39
9.4.	Control punto a punto con el robot físico	40
9.4.1.	Comandos de velocidad para el Rover UVG	40
9.4.2.	Pruebas de implementación del controlador PID	41
10.	Conclusiones	43
11.	Recomendaciones	45
12.	Bibliografía	47
13.	Anexos	49
13.1.	Repositorios de Github	49

Lista de figuras

1.	Evaluación de desempeño para el robot en la URC [2].	4
2.	El robot móvil equipado [3].	4
3.	Vista superior del Rover UVG [1].	5
4.	Diagrama de comunicación entre nodos de ROS [6].	14
5.	Entorno gráfico de Gazebo [7].	15
6.	Pololu MiniIMU-9 v2 [13].	17
7.	Módulos DWM1001 [15].	18
8.	SLAM mediante un sensor LIDAR 3D [16].	18
9.	Dimensiones del chasis del Rover UVG en centímetros.	24
10.	Dimensiones de las orugas del Rover UVG en centímetros.	24
11.	Chasis del Rover UVG.	25
12.	Orugas del Rover UVG.	25
13.	URDF del Rover UVG visualizado en RViz.	25
14.	Visualización del URDF del Rover UVG en un mundo en Gazebo.	26
15.	Plugin <i>rqt_robot_steering</i>	27
16.	Visualización de todas las <i>topics</i> disponibles.	28
17.	Información publicada sobre la <i>topic /odom</i>	28
18.	Modelo en el mundo	29
19.	Resultados para la pose deseada #1	30
20.	Visualización del URDF del Rover UVG en un mundo en Gazebo.	32
21.	Visualización del robot rodeado de distintos objetos.	33
22.	Visualización de los objetos detectados por el sensor LIDAR.	33
23.	Visualización del robot rodeado de disntos objetos para probar el algoritmo de SLAM.	34
24.	Visualización del mapa construido al momento de correr el archivo launch. . .	35
25.	Visualización del robot en movimiento en conjunto con el <i>plugin</i> de <i>rqt_robot_steering</i>	35
26.	Visualización del mapa construido luego de la navegación del robot.	36
27.	Raspberry Pi 4 [21].	38
28.	Diagrama de sensores/actuadores del Rover UVG.	39

29.	Plataforma del Laboratorio del Robótica.	40
30.	Rover UVG sobre la plataforma para pruebas de control punto a punto. . . .	41

Lista de cuadros

2.	Características de máquina virtual	20
3.	Pose deseada #1	30
4.	Porcentaje de error #1	30
5.	Porcentaje de error con 5 corridas	31
6.	Porcentaje de error para cada coordenada	31
7.	Características del sensor LIDAR	32
8.	Diferencias Raspberry Pi 3 y 4	38
9.	Porcentaje de error con 5 corridas para la implementación física	42
10.	Porcentaje de error para cada coordenada	42

Resumen

En la actualidad *Robot Operating System* (ROS) es una de las principales herramientas que se utiliza para el desarrollo de robots en toda clase de aplicaciones. Se caracteriza por ser versátil y de código abierto, por lo que se ha considerado a ROS una herramienta sumamente útil para resolver problemas de fabricación, optimización, investigación, recorrido, entre muchos más. Cuenta con una gran cantidad de librerías y recursos que facilitan al desarrollador crear programas para tareas sencillas hasta tareas mucho más complejas.

En la siguiente tesis se presenta un propuesta de integración de ROS al Rover UVG, de manera que se pueda dotar de autonomía y sea capaz de cumplir con múltiples tareas. ROS provee herramientas muy útiles de simulación, como lo son Gazebo y Rviz, con las cuales se realizaron simulaciones realistas del Rover UVG dentro de un mundo virtual. Dentro de estas simulaciones se realizaron pruebas de integración de los distintos módulos externos que posee el robot junto a programas generados en ROS para realizar experimentos de control punto a punto.

De igual forma se evaluó cuál es el modelo óptimo de Raspberry Pi para ser utilizado como computadora central del Rover UVG y poder comunicarse con los módulos externos. Con dicha computadora se realizaron las pruebas físicas para validar el funcionamiento de los controladores generados previamente en la etapa de simulación.

Abstract

Currently *Robot Operating System* (ROS) is one of the main tools used for the development of robots in all kinds of applications. It is characterized by being versatile and open source, this is why it has been considered an extremely useful tool to solve manufacturing, optimization, research, among others. ROS has a large number of libraries and resources that make it easy for the developer to create programs for simple tasks to much more complex and interesting.

In the following thesis, a proposal for the integration of ROS to the Rover UVG is presented, so it can be provided with autonomy and be capable of fulfilling multiple tasks. ROS provides very useful simulation tools, such as Gazebo and Rviz, so that a realistic model is obtained and all the performance tests can be carried out correctly. In the same way, it will be evaluated which is the optimal model of Raspberry Pi or any other computer to be used as the central computer of the Rover UVG and to be able to communicate with the external modules.

CAPÍTULO 1

Introducción

Robot Operating System (ROS) proporciona funcionalidad para la abstracción de hardware, controladores para múltiples dispositivos, comunicación entre procesos dentro de varias máquinas, herramientas para simulación, visualización, entre otros. La característica clave de ROS es cómo se ejecuta el software y cómo se comunica, debido a que permite diseñar software complejo sin saber cómo funciona cierto hardware. Es por ello que gran número de robots que se fabrican hoy en día, utiliza este sistema operativo para su funcionamiento.

En el caso de la Universidad del Valle de Guatemala, se cuenta con el proyecto del robot explorador modular que ha tenido sus últimas modificaciones hasta el año 2021 como se muestra en [1]. Aunque hasta el momento, no se ha logrado que el robot logre moverse con facilidad ni tampoco pueda realizar más de una función. Para lograr que el Rover UVG pueda ampliar sus aplicaciones, se cuenta con distintos módulos externos, entre ellos se encuentran *encoders* en las orugas del robot, módulos DMW1001, sistema de captura de movimiento Optitrack, LIDAR y cámaras. Con dichos módulos se planteó la propuesta de integración junto a ROS, de forma que el robot tenga sensores de localización, visión y mapeo.

Para validar la correcta integración de los módulos externos se realizaron simulaciones realistas en Gazebo. Estas simulaciones abarcaron distintas clases de experimentos dentro de un mundo virtual, como control punto a punto, SLAM, entre otros. Posteriormente se realizó la selección de la computadora central adecuada que permitiera la integración de todos los módulos externos junto a ROS dentro del robot físico. Para poder validar la implementación física se realizaron pruebas simples dentro de una plataforma que nos permitió verificar la integración de los módulos y la funcionalidad de los controladores realizados en la etapa de simulación.

CAPÍTULO 2

Antecedentes

2.1. Ejemplos de sistemas robóticos basados en ROS

La integración de *Robot Operating System* (ROS) en toda clase de robots cada vez es más común, ya sea manipuladores seriales, robots móviles con ruedas, drones, animatrónicos, entre muchos más. En donde se utiliza dicho sistema operativo para controlar robots con aplicaciones sencillas hasta mucho más complejas y dinámicas. A continuación se presentan distintos proyectos que integran ROS, así como el proyecto del Rover UVG con el cual se estará trabajando.

2.1.1. Ryerson University Rover

En [2] Daniel Snider de la universidad de Ryerson en Toronto Canadá describió la implementación de ROS en un Rover para la University Rover Challenge (URC) del año 2017. Se trabajó en una arquitectura para ROS de manera que existiesen cinco sistemas principales dentro del proyecto, los cuales abarcan el sistema autónomo del robot, el sistema de manejo, sistema de visualización, sistema de odometría y por último el sistema de posición. Cada uno de estos sistemas fue implementado como un paquete independiente en la arquitectura de ROS de forma que se pudiera organizar y estructurar de mejor manera cada una de las partes del robot. En [2] se describe con detalle lo que se integra en cada paquete, la forma de comunicación entre los módulos, el lenguaje de programación utilizado, así como todos los sensores y actuadores empleados en cada sistema. Se utilizó una Raspberry Pi 3 para cargar el sistema operativo la cual se comunica a su vez con otros microcontroladores (como Arduino) que se encargan de poder leer los sensores, controlar los motores, entre otras diversas tareas. En la Figura 1 se observa la evaluación del Rover por medio de su desempeño realizando distintas tareas.



Figura 1: Evaluación de desempeño para el robot en la URC [2].

2.1.2. Robot móvil autónomo basado en ROS

En [3] Jun Takamatsu expone la integración de ROS sobre un robot móvil autónomo en donde utilizan un LIDAR 2D y camaras RGB-D como se muestra en la Figura 2. Para dicha implementación utilizaron gran parte de los paquetes que vienen integrados dentro de ROS, en los cuales solo realizaron cambios mínimos de los parámetros que vienen por defecto para poder integrarlo a su robot. Su principal contribución fue obtener dos configuraciones del sistema de ROS para la navegación y autonomía del robot en trayectorias específicas. Para la primera configuración hicieron uso del LIDAR 2D junto con una Raspberry Pi 3, y para la segunda configuracion hicieron uso del LIDAR como la camara RGB-D para poder trabajar en conjunto con Intel NUC. Para ambas configuraciones lograron que el robot pudiera evadir obstáculos en su camino, así como parar en obstáculos inevitables.

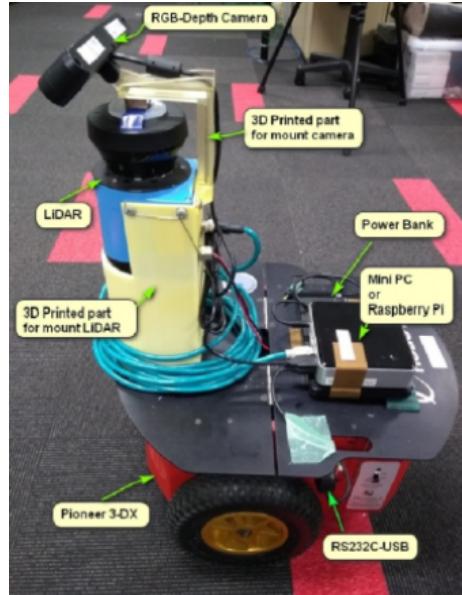


Figura 2: El robot móvil equipado [3].

2.2. Proyectos de robótica en la Universidad del Valle de Guatemala

2.2.1. Robot Explorador Modular

Hasta el año 2021 Hector Sagastume y Javier Archila realizaron las últimas modificaciones para el robot explorador modular. En [1] Hector Sagastume logró realizar una estructura capaz de adaptarse al sistema ya existente brindando protección a los componentes, reducción de tamaño, reducción de peso y optimización de espacio. De igual manera implementó motores con controlador que brindan mayores revoluciones por minuto sin comprometer la tracción del robot explorador, manteniendo una capacidad todo terreno. Sagastume también implementó cinco sensores ultrasónicos dentro de la estructura de forma que fuesen capaces de informar al usuario y al sistema de los cambios de proximidad y ambiente en su entorno. En [4] Javier Archila implementó el autopiloto PixHawk en conjunto con una Raspberry Pi 3 para manejo de telemetría a través de Internet y manejo remoto. Si bien en [1] y [4] fueron capaces de reestructurar dicho robot explorador y dotarlo de capacidades mecánicas y eléctricas para su funcionamiento, ambos concuerdan que se debe seguir trabajando y desarrollando el robot de manera que se puedan implementar nuevos módulos logrando incrementar las aplicaciones del robot.



Figura 3: Vista superior del Rover UVG [1].

2.2.2. Simulaciones de Robótica de Enjambre con ROS

En [5] Marcos Izeppi trabajó en la implementación de un algoritmo capaz de navegar a través de terrenos desconocidos y siguiendo un comportamiento brindado por el algoritmo de optimización de enjambre causando que el robot converja hacia el mínimo local de la función de costo. Determinó la viabilidad de la utilización de ROS para simulaciones de algoritmos de búsqueda de trayectorias como lo es el algoritmo de Dijkstra y D*. Para ello utilizó Gazebo, entorno de simulación de robots que viene en conjunto con ROS. De igual forma Izeppi expone comandos básicos y componentes principales de ROS para poder comprender el funcionamiento y partes esenciales que conforman cualquier proyecto en dicho sistema operativo. Pese a que en [5] se describe información útil y necesaria para el conocimiento y

utilización de ROS, el objetivo principal fue únicamente utilizar ROS como herramienta de simulación para comprobar la funcionalidad de dichos algoritmos, con lo cual no se aprovecha al máximo todos los recursos que el sistema operativo nos puede ofrecer así como un gran número de paquetes que pueden mejorar la eficiencia del proyecto y aumentar su alcance.

CAPÍTULO 3

Justificación

Hasta el momento, solamente se ha logrado controlar el Rover UVG mediante control remoto lo cual limita de gran manera sus aplicaciones. Por ello, mediante la implementación de un sistema operativo capaz de controlar y manejar al robot de una manera adecuada, se puede ampliar el alcance del Rover para que sea apto para cumplir con distintas tareas más complejas por sí solo.

Por ello se considera a ROS como la herramienta óptima para su implementación dentro del Rover UVG. ROS se caracteriza por ser una plataforma de código abierto y versátil. Este sistema operativo cuenta con un gran número de paquetes adecuados para distintos tipos de sensores/actuadores e incluso para la programación de tareas simples del robot. De igual manera, permite la creación de paquetes propios para la realización de tareas específicas así como la comunicación con sistemas o equipos externos.

Los robots exploradores modulares que se fabrican en distintas partes del mundo son utilizados para aplicaciones que abarcan desde el reconocimiento, rescate y exploración hasta aplicaciones militares y espaciales. La integración de ROS al Rover UVG permitiría poder cumplir con alguna o múltiples de estas aplicaciones de manera que se tenga un prototipo bastante completo como base para futuras investigaciones y proyectos.

CAPÍTULO 4

Objetivos

4.1. Objetivo general

Integrar una computadora central al Rover UVG que sea capaz de ejecutar ROS, con lo cual se pueda dotar de autonomía al robot y permita ampliar sus aplicaciones.

4.2. Objetivos específicos

- Seleccionar una computadora central adecuada para ejecutar ROS y que sea integrable al Rover.
- Crear una imagen estandarizada del sistema operativo.
- Establecer los requerimientos de comunicación para los módulos de percepción y actuación del Rover e integrarlos para su funcionamiento en conjunto.
- Realizar simulaciones realistas del Rover para validar el funcionamiento de los módulos externos y su correcta integración con ROS.

CAPÍTULO 5

Alcance

Entre las principales metas del proyecto del Rover UVG se encuentra ampliar sus aplicaciones hasta volverlo un robot completamente autónomo. Para ello se cuentan con distintos módulos externos como lo son *encoders*, módulos DMW1001, sistema de captura de movimiento Optitrack, LIDAR y cámaras. Estos módulos son integrables a una computadora central que ejecutara ROS, de manera que el robot tenga sensores y actuadores adecuados para realizar distintas tareas.

Este trabajo de graduación se enfocara en el desarrollo de los programas en ROS para la correcta integración de cada uno de los módulos externos dentro de dicho sistema operativo. Cada uno de estos módulos proveera información del entorno del robot, la cual podrá ser procesada y utilizada para realizar experimentos. También se desarrollaran programas basados en controladores para realizar control punto a punto dentro de un entorno controlado. Todas estas pruebas se realizan bajo un entorno simulado dentro de Gazebo, y posteriormente en la implementación física gobernada por la computadora central seleccionada.

CAPÍTULO 6

Marco teórico

6.1. Robotic Operating System (ROS)

Robot Operating System (ROS) es una colección de marcos flexibles para la escritura de software de robots. Se caracteriza por ser de código abierto y proveer conexiones entre procesos de múltiples dispositivos. Cuenta con un conjunto amplio de librerías y convenciones que facilitan al desarrollador la tarea de crear un comportamiento complejo y robusto en una amplia variedad de plataformas robóticas. Las principales bibliotecas de ROS han sido desarrolladas en su totalidad para ser utilizadas con el sistema operativo Ubuntu, con base en Linux y de forma experimental (con fallas y detalles por perfeccionar) funciona con IOS y Windows [6].

El ecosistema de ROS se compone de decenas de miles de usuarios en todo el mundo, que trabajan en pequeños proyectos de aficionados hasta sistemas de automatización industrial de gran escala. ROS fue diseñado para ser sistema operativo modular, de forma que los usuarios puedan seleccionar y elegir qué partes son útiles y qué partes se prefiere implementar. La comunidad de usuarios de ROS definen una infraestructura común para proporcionar un punto de integración que ofrece el acceso a los controladores de hardware, las capacidades de robots genéricos, herramientas de desarrollo, librerías externas útiles, y más [7].

Los paquetes en ROS son el elemento principal de su arquitectura. Estos pueden contener los procesos de ROS en tiempo de ejecución (nodos), conjunto de datos, archivos de configuración o cualquier elemento que sea necesario para su funcionamiento. Cada paquete puede ofrecer una determinada función de manera que el software se pueda construir y reutilizar fácilmente. De igual forma, se pueden crear paquetes propios para el desarrollo de tareas específicas dentro de cualquier robot o proyecto donde se trabaje. Un nodo es un proceso que realiza cálculos, y se puede comunicar con otros nodos mediante la transmisión de *topics*, *RPC services* y el *Parameter Server*. Generalmente, cada sistema de control que conforma un robot se describe por medio de paquetes, y estos a su vez por medio de nodos con tareas específicas los cuales se comunican entre sí para lograr los principales objetivos del robot en desarrollo [8]. En la Figura 4 se observan los distintos tipos de comunicación

que se pueden utilizar entre nodos y las características que poseen los nodos.

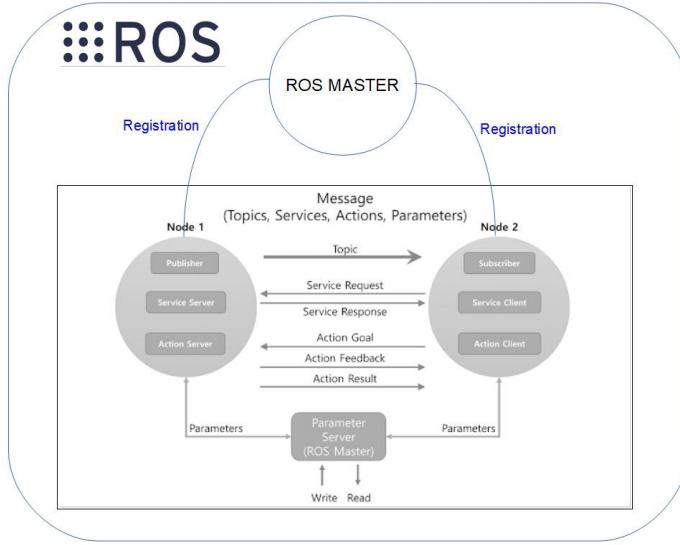


Figura 4: Diagrama de comunicación entre nodos de ROS [6].

6.2. Gazebo

Gazebo es un simulador 3D multi-robot con propiedades dinámicas y cinemáticas completas. Ofrece la posibilidad de simular con precisión y eficiencia para diversidad de robots, objetos y sensores en ambientes complejos interiores y exteriores. Gazebo genera la realimentación realista de sensores, como las interacciones entre los objetos físicamente plausibles, incluida una simulación precisa de la física de cuerpo rígido. Es una herramienta gratuita de simulación, realmente útil para prueba y simulación de algoritmos elaborados, de manera que brinda la posibilidad de visualizar aciertos y fallas del robot en desarrollo [9].

La arquitectura de Gazebo está basada en el motor de simulación ODE (Open Dynamics Engine), creado por Russell Smith, aunque su desarrollo final se atribuye a Andrew Howard y Nate Koenig, por sus investigaciones en la Universidad del Sur de California [10]. Gazebo permite de forma directa, crear mundos y objetos idóneos para insertar el modelo robótico que se esté desarrollando, con el fin de visibilizar la manera en la que el mismo interactúa en un ambiente determinado, de manera que se pueda verificar el funcionamiento de sensores, aptitudes de movilidad e incluso de superación de obstáculos. La integración entre ROS y Gazebo es proporcionada por un conjunto de *plugins* de Gazebo que apoyan muchos robots y sensores existentes. Debido a que los *plugins* presentan la misma interfaz de mensaje que el resto del ecosistema ROS, se pueden escribir nodos ROS que sean compatibles con la simulación, los datos registrados y el hardware de los robots [7].

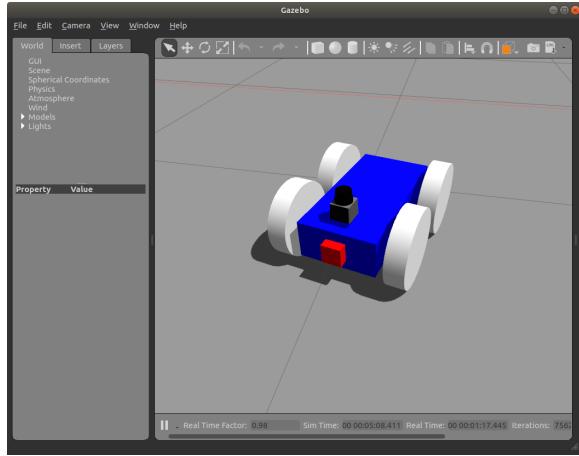


Figura 5: Entorno gráfico de Gazebo [7].

6.3. Rviz

Rviz es la herramienta de visualización 3D con la que dispone ROS. Esta herramienta proporciona la representación de robots, datos de sensores como escaneos láser, nubes de puntos tridimensionales e imágenes de cámaras dentro del robot, entre otras. Rviz tiene funciones estrictas de visualización por lo que a diferencia de Gazebo, no provee esquemas de simulación de fricción, gravedad y otros, ni la construcción de escenarios de interacción. La forma en que funciona Rviz es simplemente leyendo e interpretando los diferentes datos contenidos en mensajes de ROS. Por tanto, es necesario tener un generador externo de estos mensajes, como puede ser un robot físico o simulado [11].

6.4. Lenguaje de programación XML

Las siglas XML son la abreviación de “Extensible Markup Language”, lo cual se trata de un lenguaje utilizado para estructurar la información en cualquier documento que contenga texto como lo son archivos de configuración de una aplicación específica, base de datos, hojas de cálculo, entre otras. La razón de su popularidad se debe al hecho de ser un estándar abierto y además libre, creado por W3C, el consorcio *World Wide Web*, en colaboración con un equipo de trabajo que incluye representantes de las compañías de software más importantes. El lenguaje XML fue creado en 1996 y desde ese momento su utilización tuvo un crecimiento sostenido [7].

La tecnología XML mantiene la información estructurada de la forma más abstracta y reutilizable posible. Por ello, un documento XML está formado las siguientes partes:

- **Prólogo:** Los documentos XML pueden empezar con unas líneas que describen la versión XML, el tipo de documento y otras cosas.
- **Cuerpo:** El cuerpo tiene que contener sólo un elemento raíz, característica obligatoria

para que el documento esté bien formado. Sin embargo es necesaria la adquisición de datos para su buen funcionamiento.

- **Elementos:** Los elementos XML pueden tener contenido (más elementos, caracteres o ambos) o bien ser elementos vacíos.
- **Atributos:** Los elementos pueden tener atributos, que son una manera de incorporar características o propiedades a los elementos de un documento. Deben ir entre comillas.
- **Entidades predefinidas:** Entidades para representar caracteres especiales para que no sean interpretados como marcado en el procesador XML.
- **Comentarios:** Son aclaraciones que pueden ser escritas para informar al lector, y son ignorados por el procesado.

6.5. Formato unificado de descripción para robots (URDF)

United Robotics Description Format (URDF) es un formato de lenguaje utilizado para la descripción de robots en el marco de gramática XML. Es una herramienta útil para el modelado de un robot con lo cual se pueden realizar pruebas de simulación y análisis. Para la creación del modelo no solamente se definen enlaces y uniones del robot, también es necesario conocer cierta información física de los componentes básicos, como los atributos de masa, inercia, colores y tipos de articulaciones, ya sea una articulación giratoria o una articulación traslacional [7].

6.6. Formato de descripción para simulación (SDF)

Simulation Description Format (SDF) es un formato XML que describe objetos y entornos para simuladores, visualización y control de robots. Fue desarrollado originalmente como parte del simulador de robots Gazebo. Se puede describir con precisión todos los aspectos de un robot usando dicho formato, ya sea que el robot solamente posea un chasis simple con ruedas o hasta un robot mucho más detallado como un humanoide. Además de los atributos cinemáticos y dinámicos, se pueden definir sensores, propiedades de superficie, texturas, fricción de juntas y muchas más propiedades para un robot. Estas características permiten usar el formato SDF para simulación, visualización, planificación de movimiento y control de robots. La simulación requiere entornos ricos y complejos en los que existen modelos e interactúan. Por ello el formato SDF también proporciona los medios para definir una amplia variedad de entornos, en donde se pueden incluir varias luces en un entorno, el tipo de terreno del ambiente, entre otras características [12].

6.7. Sensores comúnmente utilizados para robots móviles

6.7.1. IMU

Una *Inertial Measurement Unit* (IMU) es un dispositivo electrónico que utiliza una combinación de acelerómetros y giroscopios para medir la aceleración, la velocidad angular y la orientación del sensor. La rotación y orientación 3D puede ser obtenida por medio de ángulos de Euler o cuaterniones según la IMU que se tenga. Tanto la velocidad angular como velocidad lineal se obtiene por medio de vectores tridimensionales (x , y y z) con respectivos valores de velocidad angular o lineal que dependen de los ejes x , y y z [13].

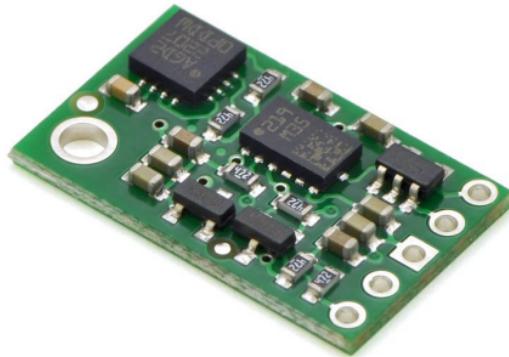


Figura 6: Pololu MiniIMU-9 v2 [13].

6.7.2. GPS

Otro de los sensores más utilizados son los *Global Positioning System* (GPS), el cual es un sistema de navegación por satélite compuesto por una red de 24 satélites colocados en órbita por el Departamento de Defensa de Estados Unidos. Su principal función es establecer la posición en coordenadas de latitud y longitud en cualquier lugar de la Tierra. Para ello necesita tener cobertura de por lo menos tres satélites, con ello puede triangular la posición de los satélites captados y nos presenta los datos de longitud, latitud y altitud calculados [14].

6.7.3. Módulos DWM1001

Un módulo DWM1001 es un transceptor inalámbrico monochip basado en técnicas UWB, que permite a los ingenieros desarrollar sistemas de localización en tiempo real (RTLS) con una precisión de 10 cm en interiores y exteriores. Compatible con IEEE802.15.4-2011, el CI soporta una transferencia de datos de hasta 6,8 Mbps. Además, los módulos DWM1001 integran el CI DW1000, una antena y componentes de gestión de potencia y sincronización para simplificar la integración del diseño con una amplia variedad de microcontroladores [15].



Figura 7: Módulos DWM1001 [15].

6.7.4. LIDAR

En la actualidad, la mayoría de robots móviles autónomos se caracterizan por poseer sensores LIDAR para crear entornos 3D. Un *Light Detection and Ranging* (LIDAR) es un dispositivo que permite determinar la distancia desde un emisor láser a un objeto o superficie utilizando un láser pulsado. La distancia a los distintos objetos se determina midiendo el tiempo de retraso entre la emisión del pulso y su detección a través de la señal reflejada. Un dispositivo LIDAR es uno de los sensores principales para la navegación SLAM. *Simultaneous Localization And Mapping* (SLAM) se trata de una técnica de navegación que permite a un robot o vehículo, construir un mapa del entorno y al mismo tiempo navegar por él, gracias a la información que captan sus sensores en tiempo real [16].

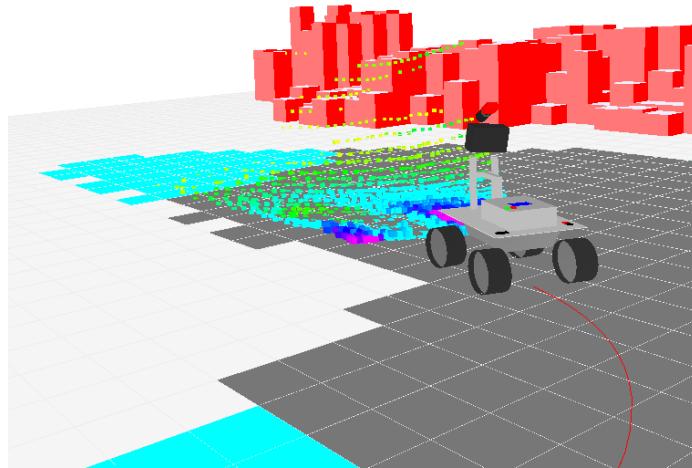


Figura 8: SLAM mediante un sensor LIDAR 3D [16].

CAPÍTULO 7

Entorno de Robot Operating System (ROS)

Previo al desarrollo de los programas en ROS para control del Rover UVG, se explicará acerca de los pasos para la instalación de dicho sistema operativo dentro de una máquina virtual. También se profundiza en los conceptos fundamentales que tiene este sistema operativo y la metodología a seguir para el desarrollo de programas básicos en ROS. De igual forma se mencionan algunas prácticas comunes y eficientes que ayudan a obtener los mejores resultados al momento de realizar distintos proyectos en ROS.

7.1. Instalación de ROS 2

Para la implementación del sistema operativo del Rover UVG se decidió utilizar la versión de ROS 2 en lugar de su primera versión (ROS). Esto debido a que la segunda versión nos ofrece mejoras significativas como:

1. **Lenguajes:** ROS trabaja con Python 2.7 y C++ 11, en donde ROS 2 da un salto importante permitiendo crear código en Python 3.5 y en versiones superiores de C++ como la 14 y 17 [17].
2. **Centralizado vs. Distribuido:** ROS funciona mediante un modelo centralizado, donde se requiere de un *Master* para poder registrar todos los nodos, servicios y topics en ejecución. Dicho *Master* permite a los nodos localizarse unos a otros. En cambio en ROS 2 se prescinde de este de manera que sean los propios nodos los que avisen al resto de su aparición o marcha de la red [17].
3. **TCPROS vs. DDS:** ROS hace uso de TCPROS el cual se basa en TCP/IP para el envío de mensajes y comunicación con servicios. En ROS 2 se decidió utilizar *Data*

Distribution Service (DDS) el cuál es un *framework* de arquitectura publicador/subscriptor que aporta gran flexibilidad y eficiencia al permitir comunicación directa entre nodos, algo realmente necesario para aplicaciones en tiempo real [17].

4. **Migración:** La última versión de ROS con soporte oficial es “Noetic”, que se lanzó en 2020, para la cual seguirán dando soporte hasta 2025. Después de eso, no habrá nuevas versiones de ROS con soporte oficial, por lo que la mayoría de proyectos tendrán que migrar a ROS 2 [17].

Previo a la instalación de ROS 2, se descargó una máquina virtual que soportara Linux de manera que se pudiera realizar todo el desarrollo dentro de dicho sistema operativo. Esto debido a que si bien, ROS 2 supone estar disponible para Windows 10, el desarrollo de los proyectos de ROS es mucho más fácil para Linux y muchas librerías de ROS solo tienen soporte para dicho sistema operativo. Se descargó Oracle VM VirtualBox para la creación de dicha máquina virtual para la cual se seleccionaron las características que se presentan en el Cuadro 2. Dichas características son los requisitos que debe tener la máquina para poder ejecutar ROS 2 de forma fluida, ya que se realizarán simulaciones y visualizaciones en tiempo real.

Sistema Operativo	Ubuntu (64-bit)
Memoria RAM	8192 MB
Procesadores	2
Almacenamiento	30 GB

Cuadro 2: Tabla de especificaciones. Características para la creación de la máquina virtual en Oracle VM VirtualBox

Luego de instalar la máquina virtual, con las respectivas especificaciones señaladas previamente, se procedió a realizar la instalación de ROS 2 en dicha máquina. Para esta instalación fue necesario conocer las distribuciones existentes, en donde actualmente las distribuciones con soporte son “Foxy”, “Humble” y “Eloquent” para ROS 2. Se decidió instalar la versión “Foxy” debido a que es la única distribución que es *Long-term support* (LTS) hasta el momento. Para realizar la instalación se utilizó la guía existente en la documentación de ROS 2 para “Foxy” que se puede encontrar en [18].

Como último paso, se procedió a realizar la instalación de “colcon”, una de las herramientas para compilar con la cual trabaja ROS 2. El procedimiento para realizar la instalación de dicho compilador y de otras herramientas adicionales que se utilizaron, se encuentran detallados dentro del Capítulo 13.1 de la sección de anexos. De igual forma, en dicho anexo se encuentra a detalle cada uno de los pasos para la creación de paquetes y nodos que se explica en el capítulo posterior.

7.2. Creación de paquetes y nodos

Para el desarrollo de proyectos en ROS 2 es recomendado realizar un *Workspace* para cada uno de los proyectos que se desean realizar. Es por ello que en la máquina virtual se creó

una carpeta llamada *Workspaces* que contendrá todos los proyectos creados para ROS 2. Una práctica común es crear una carpeta con el nombre deseado seguido de “_ws” para indicar que es un espacio de trabajo para ROS 2. Dentro de la carpeta del *workspace* se deben crear los paquetes deseados, para lo cual es necesario abrir una terminal en dicha carpeta, o bien, abrir una terminal y cambiar el directorio hasta que nos encontremos dentro del respectivo *workspace*. Luego se procede a crear una carpeta con el nombre de *src*, abreviatura de la palabra *source* en inglés, dentro de la cual se deberán crear todos los paquetes deseados. El número de paquetes que se pueden crear dentro de un *workspace* es ilimitado, dicha cantidad depende directamente del proyecto en que se trabaje y la organización que se desee tener.

Posteriormente se procede a crear el paquete con el lenguaje que deseemos, ya sea Python o C++, siempre dentro de la carpeta *src*. Luego de haber creado el paquete, se puede notar que se ha creado una carpeta con el nombre especificado. Para los paquetes creados en Python se podrá observar que ingresando dentro de dichas carpetas, se generaron 3 carpetas más. La primera carpeta, con el mismo nombre de nuestro paquete, sera donde tendremos que almacenar todos los programas que se desarrollen para el paquete en específico. De igual forma, se generaron 3 archivos independientes los cuales se describen a continuación:

- *package.xml*: contiene los metadatos del paquete, como lo son el nombre del paquete, distribución de ROS 2, información del autor, entre otros. [19]
- *setup.py*: contiene instrucciones de como instalar el paquete. [19]
- *setup.cfg*: el cual es un paquete de Python que ayuda a poder administrar todos los ejecutables que se almacenen en el paquete creado. [19]

Para los paquetes creados en C++ se creará una carpeta con el nombre especificado y dentro de ella se podrán encontrar dos carpetas más. La primera es la carpeta de *include* en donde se deberán almacenar todos los archivos de cabecera o los *header files* en inglés, los cuales son de suma importancia para lenguajes como C y C++ en este caso. La segunda carpeta creada será *src* en donde se deberán colocar todos los programas desarrollados para este paquete específico. Adicionalmente, se crearán los siguientes archivos independientes:

- *package.xml*: contiene los metadatos del paquete, como lo son el nombre del paquete, distribución de ROS 2, información del autor, entre otros. [19]
- *Cmakelist.txt*: contiene un conjunto de directivas e instrucciones que describen los archivos de origen y los destinos del proyecto. [19]

Seguido de la creación de los paquetes, se deben crear los nodos los respectivos nodos. La función de estos nodos depende directamente del proyecto en que se trabaje. De la misma manera que para los paquetes, estos nodos pueden ser desarrollados en C++ o en Python, pero se recomienda desarrollar estos nodos en el mismo lenguaje en que se crearon los paquetes.

Una vez desarrollado el/los paquetes respectivos, junto a la cantidad de nodos que se deseen, se debe compilar el *workspace* para verificar que no exista ningún error dentro de

los nodos o dentro de los paquetes. Una vez que se termine de compilar podremos observar que se generaron 3 carpetas más dentro de nuestra carpeta principal. Cada una de estas carpetas se describe a continuación:

- *build*: Contiene archivos intermedios generados cuando el compilador verifica la sintaxis de los programas.
- *install*: Carpeta donde se instalan los paquetes, y a su vez cada paquete tendrá un directorio distinto.
- *log*: Contiene credenciales de seguridad y autenticación que se genera cada vez que se compila uno o varios paquetes.

CAPÍTULO 8

Simulación por medio de Gazebo

Previo a realizar las pruebas físicas del robot junto a los programas generados en ROS, se realizaran simulaciones para comprobar y validar, dentro de un mundo virtual, el correcto funcionamiento de la integración de todos los sensores y actuadores en conjunto con los controladores programados. Para ello se definirán las propiedades físicas del robot para luego usar dicho modelo en Gazebo y observar su comportamiento.

8.1. Descripción del robot

8.1.1. Modelo URDF

Para la simulación del Rover UVG en Gazebo es necesario definir todos sus cuerpos rígidos, junto a su inercia y características visuales. Para la creación del URDF del Rover UVG, se definieron cuerpos rígidos muy simples pero que pudieran representar las características importantes del modelo físico. Para ello se realizaron mediciones tanto de la base del Rover UVG como de las orugas, como se muestra en las Figuras 9 y 10 respectivamente.

A continuación se presentan los eslabones creados para cada cuerpo rígido del robot:

- *base_link*: representa el chasis del Rover UVG, en donde su propiedad visual esta constituido por un *mesh* que se puede observar en la Figura 11, diseñado con sus respectivas medidas. Su propiedad de inercia y propiedades de colisión se definieron mediante una caja de 0.671 x 0.645 x 0.152 (base x ancho x altura) metros.
- *drivewhl_l_link*: representa la oruga del lado izquierdo del Rover UVG, en donde su propiedad visual esta constituido por un *mesh* que se puede observar en la Figura

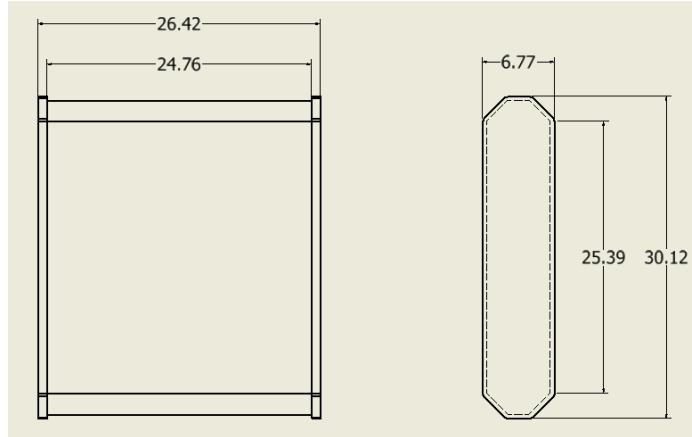


Figura 9: Dimensiones del chasis del Rover UVG en centímetros.

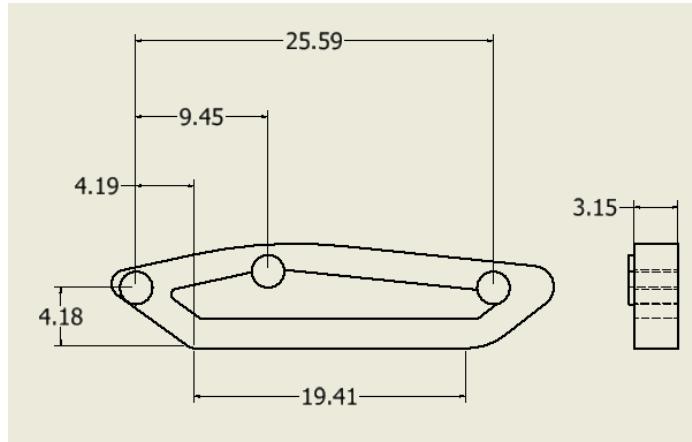


Figura 10: Dimensiones de las orugas del Rover UVG en centímetros.

12, diseñado con sus respectivas medidas. Su propiedad de inercia y propiedades de colisión se definieron mediante una cilindros de radio 0.14 m y 0.08 m de ancho. El radio de estos cilindros se determino de manera que la altura del chasis en el modelo URDF sea la misma que tiene el robot físico.

- *drivewhl_r_link*: representa la oruga del lado derecbo del Rover UVG, todas sus propiedades son iguales a las de *drivewhl_l_link*.
- *gps_link* y *imu_link*: Se definió un eslabón para simular un sensor gps y una IMU respectivamente. Estos eslabones no se definieron mediante ningún elemento físico solamente se unieron por medio de juntas a *base_link* para su uso posterior.
- *lidar_link*: este eslabon representa el sensor LIDAR que se encuentra en la parte superior del Rover UVG. Tanto su propiedad visual como de colisión se definieron con respecto a las medidas del sensor físico. Se utilizó un cilindro de radio 0.0508 m y 0.18 m de alto para dicho cuerpo rígido.

Para comprobar que el modelo URDF del robot creado realmente describa al modelo físico del Rover UVG, se puede utilizar RViz como se muestra en la Figura 13.

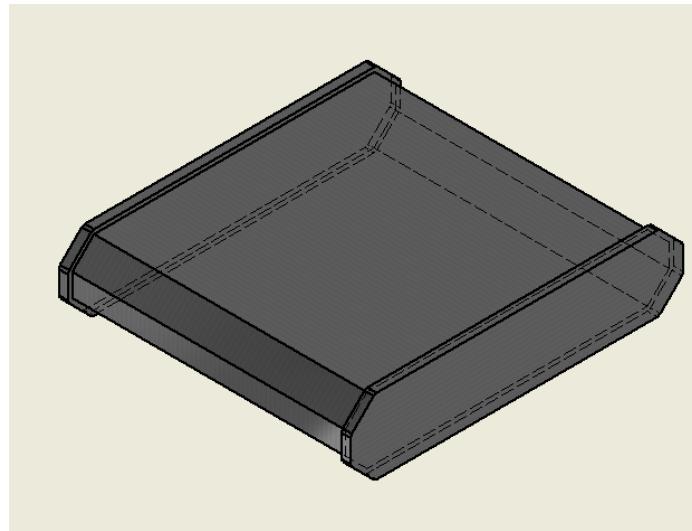


Figura 11: Chasis del Rover UVG.

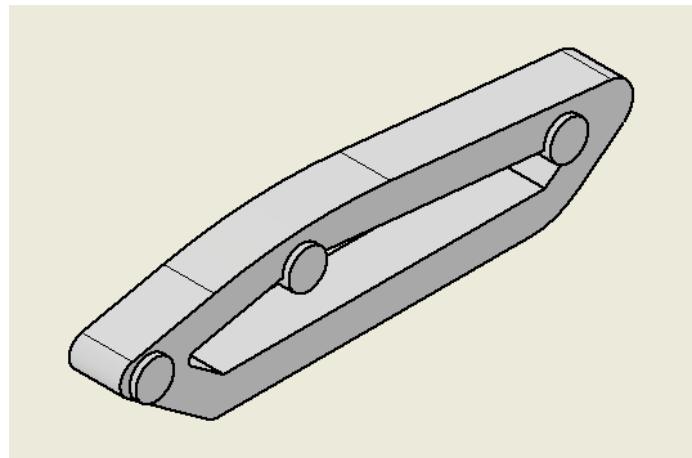


Figura 12: Orugas del Rover UVG.

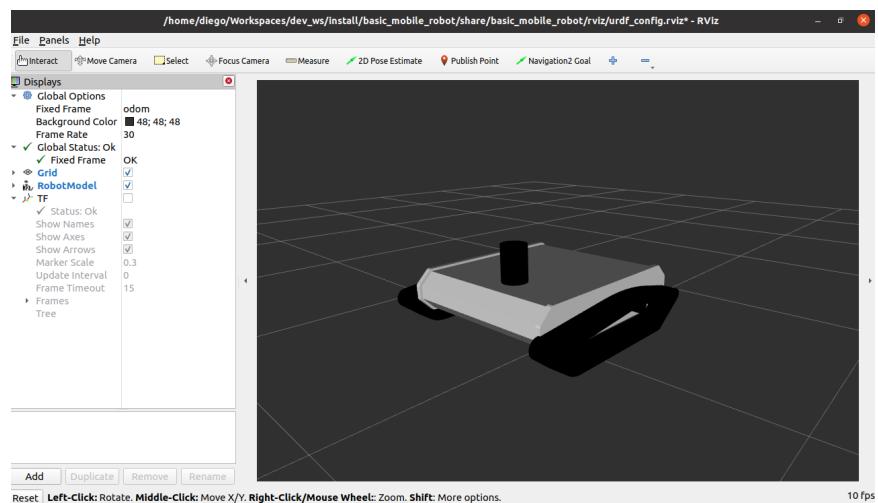


Figura 13: URDF del Rover UVG visualizado en RViz.

8.1.2. Modelo SDF

Para poder utilizar el modelo del robot previamente definido en Gazebo, se tiene que trasladar a formato SDF, debido a que el formato URDF no reúne todas las características que necesita Gazebo para su simulación. Es por ello que se realizó un modelo SDF para la visualización del Rover UVG en Gazebo, y se utilizó el modelo URDF para la interacción con RViz. Para el modelo SDF se definieron los mismos eslabones y juntas que en el URDF. Adicionalmente, se agregaron dos *plugins* en el modelo SDF. El primero es el *basic_mobile_bot_diff_drive* que se encuentra en la librería *libgazebo_ros_diff_drive.so*, el cual realiza el cálculo de odometría del robot a partir de un modelo cinemático de un robot móvil diferencial. Para ello fue necesario especificarle el diámetro de las ruedas, la separación entre cada una, y las velocidades mínimas y máximas del robot. También se le indica el nombre de las *topics* de las cuales recibirá la información de velocidad y a cuál publicará la información de odometría. El segundo *plugin* que se utilizó fue el *basic_mobile_bot_joint_state* de la librería *libgazebo_ros_joint_state_publisher*, el cual se encarga de publicar la transformación de posición y rotación de todas las juntas del robot.

Para la simulación de los sensores se utilizan dos *plugins* más, uno para la IMU y el otro para el sensor GPS. El primero es *basic_mobile_bot_imu* que se encuentra en la librería *libgazebo_ros_imu_sensor.so* en el cual se le indica el marco de referencia para la IMU y el nombre de la *topic* en donde publicará toda la información. El segundo *plugin* es *basic_mobile_bot_gps* de la librería *libgazebo_ros_gps_sensor.so*, al cual solamente es necesario indicarle el nombre de la *topic* a la cual publicará información. Para comprobar que el modelo SDF del robot creado realmente describa al modelo físico del Rover UVG, se abre Gazebo y se inserta el modelo previamente SDF creado como se muestra en la Figura 14.

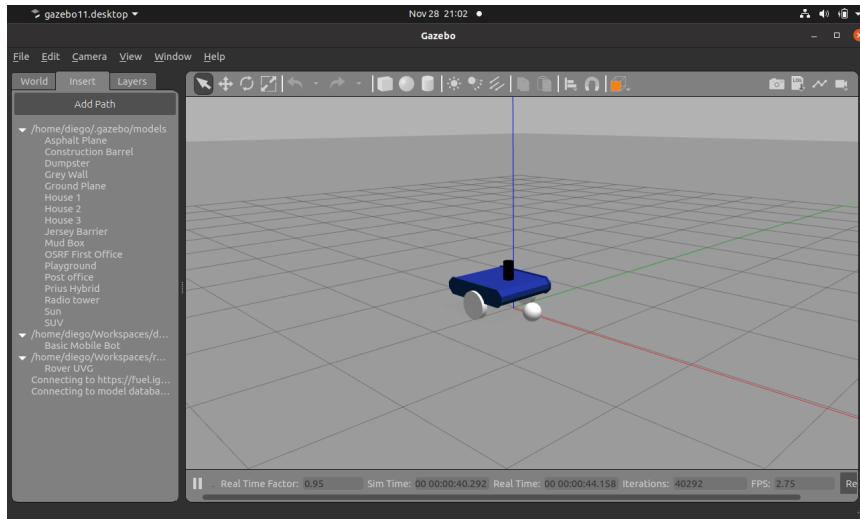


Figura 14: Visualización del URDF del Rover UVG en un mundo en Gazebo.

Si se presiona el botón de *play* en la simulación, se podrán ver el conjunto de *lasers* que aparecen alrededor del robot como se muestra en la Figura 14. De igual forma se la pestaña de *World* se podrán observar todos los eslabones que conforman al robot y sus respectivas transformaciones.

8.2. Odometría y localización del robot

Con el Rover UVG físico, los sensores que se tienen para obtener la odometría y localización del robot son los encoders de las orugas, los módulos DWM1001 y un sistema de captura de movimiento OptiTrack. Aunque estos serán los sensores para poder obtener la odometría del robot, en Gazebo no se cuenta con ninguno de estos para su uso, por lo que se utilizó un sensor GPS y una IMU para poder simular dichos valores dentro del mundo virtual. Por ello fue que se crearon dichos eslabones tanto en el modelo URDF como en el SDF del robot. Para las pruebas de funcionamiento, se utilizó el *plugin* de *rqt_robot_steering* con la cual se pueden escribir *Twist messages* a *topics* específicas. Este tipo de mensajes son útiles para describir velocidades con su componente lineal y angular, como se observa en la Figura 15.

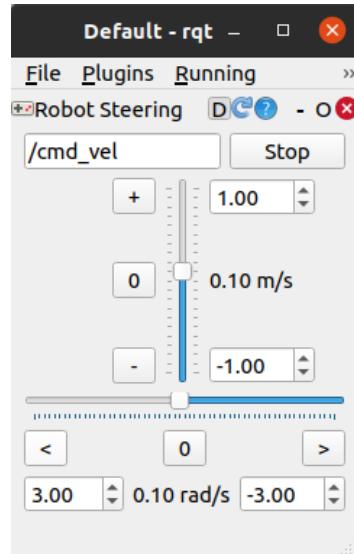


Figura 15: Plugin *rqt_robot_steering*.

Para validar que el robot sí pueda interpretar estos comandos de velocidad, se coloca el modelo SDF del robot creado en un mundo de Gazebo, como se realizó previamente. Se especifica la *topic* en el *Robot Steering* en la cual se escribirán los comandos de velocidad, en este caso debe ser la misma que se especificó en el modelo SDF del robot. Luego de esto se puede observar como el robot va moviéndose acorde a como se muevan los *sliders* de cada tipo de velocidad (lineal/angular). Abriendo una terminal y listando las *topics* disponibles, se observará la *topic* */odom* que se definió en el modelo SDF, como se muestra en la Figura 16.

```

hhj@hhj-laptop:~$ source /opt/ros/eloquent/setup.bash
hhj@hhj-laptop:~$ ros2 topic list
/parameter_events
/rosout
/turtle1/cmd_vel
/turtle1/color_sensor
/turtle1/pose
hhj@hhj-laptop:~$ 

```

Figura 16: Visualización de todas las *topics* disponibles.

Imprimiendo la información dentro de dicha *topic* podremos encontrar la posición x, y y z del robot y el cuaternión de orientación, como se muestra en la Figura 17.

```

focalfossa@focalFossa-VirtualBox:~/dev_ws/src/basic_mobile_robot/launch$ ros2 topic echo /odom
{
  "header": {
    "stamp": {
      "sec": 36,
      "nanosec": 161000000
    },
    "frame_id": "odom",
    "child_frame_id": "base_footprint"
  },
  "pose": {
    "position": {
      "x": 0.0019506930625214215,
      "y": 7.734287265682983e-09,
      "z": 0.0
    },
    "orientation": {
      "x": 0.0,
      "y": 0.0,
      "z": 0.002220191765877774,
      "w": 0.9999975353712242
    }
  },
  "covariance": [
    - 0.0009599680117201153,
    - 1.53045061117739e-18,
    - 0.0,
    - 0.0,
    - 0.0,
    - 5.333804829875089e-23,
    - 1.530450607315127e-18,
    - 0.0009599680117206679,
    - 0.0,
    - 0.0,
    - 0.0,
    - 1.6794267091434833e-20,
    - 0.0,
    - 0.0,
    - 4.995840263369695e-07,
    - 5.136999544273652e-20,
    - 6.7415765483229716e-18,
    - 0.0
  ]
}

```

Figura 17: Información publicada sobre la *topic* */odom*.

Con ello se valida que el robot efectivamente se mueva acorde a los valores de velocidad lineal y angular, y si se puede determinar la posición y orientación del robot en el mundo en Gazebo.

8.3. Control punto a punto

8.3.1. Controlador PID

De manera que se puede controlar la velocidad del robot y conociendo su ubicación dentro del mundo en Gazebo, se puede realizar control punto a punto de forma que solamente se le especifique la ubicación a la que deseemos que llegue y el robot sea capaz de moverse desde su punto de inicio hasta el punto deseado. Para ello se implementó un controlador proporcional integral y derivativo (PID) con acercamiento exponencial debido a que su estructura permite variar de forma fácil las constantes hasta encontrar el arreglo óptimo que nos entrega los mejores resultados. De igual forma, este controlador es independiente del modelo del robot, ya que no requiere conocer características físicas del robot además del diámetro y separación de sus ruedas.

Para ello se desarrolló el nodo llamado *simple_controller* dentro del paquete de *controllers* que se encuentra en el repositorio mencionado en el Capítulo 13.1. Dicho nodo se encarga de suscribirse a la *topic* de */odom* la cual contiene la localización del robot dentro del mundo virtual. Mediante la implementación del controlador PID con acercamiento exponencial se establece la velocidad lineal y angular del robot, la cual se publica en la */cmd_vel*, y se actualiza conforme el robot se va acercando a la coordenada deseada.

8.3.2. Pruebas de control punto a punto en el mundo virtual

Para poder verificar la funcionalidad del control punto a punto con el controlador previamente definido, se colocó el modelo generado del robot dentro del mundo que se muestra en la Figura 18.

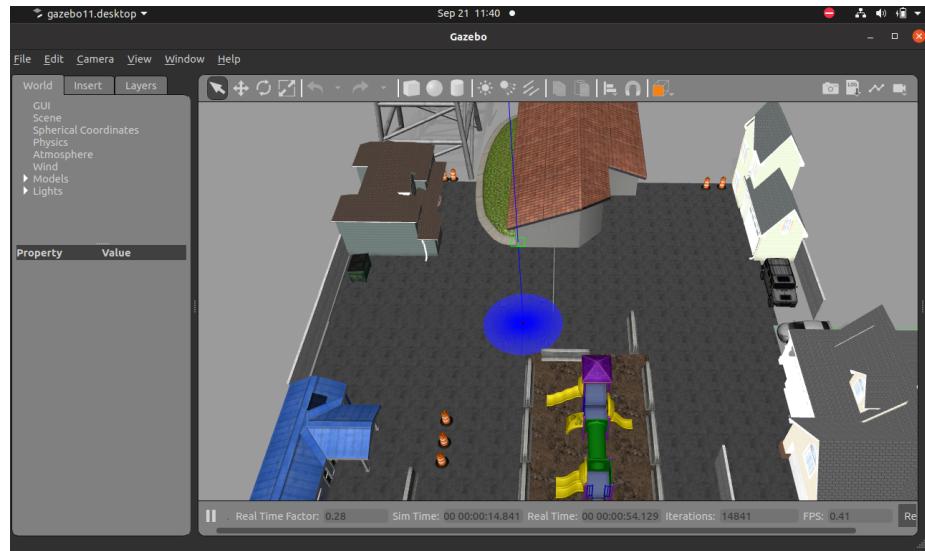


Figura 18: Modelo en el mundo

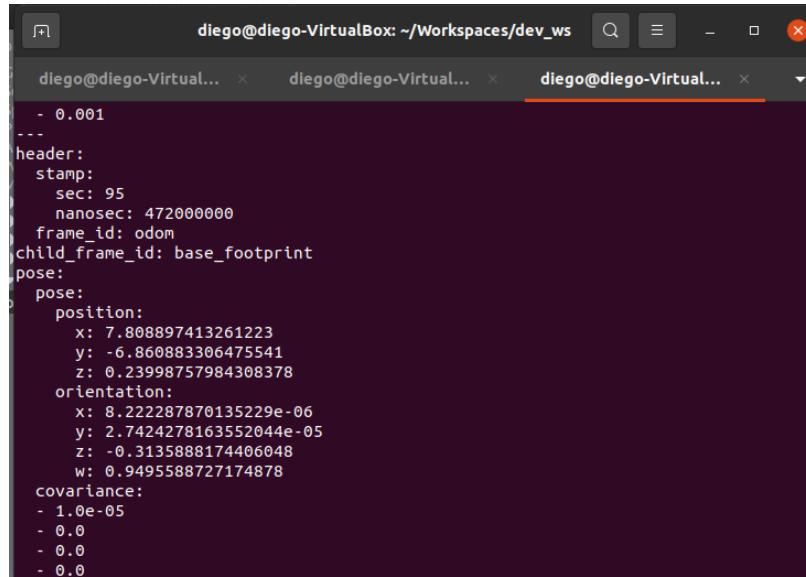
Luego se procedió a correr el controlador PID generado previamente en el cual se esta-

bleció la pose deseada como:

Coordenada	Valor (m)
x	8.0
y	-7.0

Cuadro 3: Pose deseada #1. Coordenadas de la pose deseada #1

Luego de que el robot convergiera a una posición en el mundo, se verificó en que posición se encontraba el robot como se observa en la Figura 27.



```
diego@diego-VirtualBox: ~/Workspaces/dev_ws
```

```

- 0.001
---
header:
  stamp:
    sec: 95
    nanosec: 472000000
  frame_id: odom
child_frame_id: base_footprint
pose:
  pose:
    position:
      x: 7.808897413261223
      y: -6.860883306475541
      z: 0.23998757984308378
    orientation:
      x: 8.222287870135229e-06
      y: 2.7424278163552044e-05
      z: -0.3135888174406048
      w: 0.9495588727174878
covariance:
- 1.0e-05
- 0.0
- 0.0
- 0.0

```

Figura 19: Resultados para la pose deseada #1

Con esta información se pudo obtener los resultados presentados en el Cuadro 4.

Coordenada	Valor Real (m)	Valor en Simulación (m)	Porcentaje de error (%)
x	8.0	7.81	2.38 %
y	-7.0	-6.86	2.00 %

Cuadro 4: Porcentaje de error para prueba #1. Porcentaje de error con respecto a la pose deseada #1

Por último, se realizaron 5 corridas más con distintas poses deseadas para que se pudiera obtener un porcentaje de error más preciso. En el Cuadro 5 se observa el resumen de dichos resultados.

Corrida	Coordenada	Valor Real (m)	Valor en Simulación (m)	Porcentaje de error (%)
1	x	-12.5	12.31	1.52
	y	12.5	12.61	0.88
2	x	3.0	3.12	4.0
	y	4.5	4.39	2.44
3	x	5.0	5.10	2.0
	y	5.0	5.45	9.0
4	x	10.0	9.81	1.9
	y	-1.0	-0.95	5.0
5	x	-8.6	-9.1	5.81
	y	-4.0	-3.85	3.75

Cuadro 5: Porcentaje de error para 5 distintas poses. Porcentaje de error para 5 poses deseadas distintas.

Con ello se obtuvo el promedio de los porcentajes de error tanto para la coordenada en *x*, como en *y* como se muestra en el Cuadro 6.

Coordenada	Porcentaje de error (%)
x	2.93
y	3.84

Cuadro 6: Porcentaje de error para cada coordenada después de las 6 corridas.

8.4. Simultaneous Localization And Mapping (SLAM)

Un sensor LIDAR en conjunto con un sistema de localización y el URDF del robot nos permite generar un mapa del entorno del mismo al momento que este lo recorre, incluso cuando se desconoce por completo este entorno. En esta sección se presentan las configuraciones que se tuvieron que realizar dentro de la simulación para poder obtener el mapa de cualquier entorno virtual en el cual se coloque el robot.

8.4.1. Configuración del sensor LIDAR en simulación

Uno de los sensores con los que cuenta el Rover UVG es un LIDAR 2D de la marca Hokuyo. Para poder validar su integración y conocer cómo funcionan este tipo de sensores dentro de ROS, se realizaron pruebas agregando un sensor LIDAR simulado dentro del robot en Gazebo. Para ello se modificó el modelo SDF del robot de manera que se pudiera utilizar una plantilla de un sensor LIDAR que tuviera las mismas especificaciones que el sensor real, las cuales se muestran en el Cuadro 7.

Característica	Descripción
Marca	Hokuyo
Modelo	URG-04LX-UG01
Número de mediciones	1081
Rango de escaneo	0.06 a 4 metros
Ángulo de escaneo	270°
Resolución angular	0.25°

Cuadro 7: Características del sensor LIDAR 2D Hokuyo

Como se observó en el capítulo del URDF del robot, ya se encuentra un eslabón que representa al sensor LIDAR. Es por ello que sólo se necesita incluir un *plugin* que pueda simular la información que proveen este tipo de sensores dentro de la simulación en Gazebo. Para ello se utiliza el *plugin scan* de la librería *libgazebo_ros_ray_sensor.so* dentro del modelo SDF del robot y se especifican todas las características presentadas en el Cuadro 7 así como el marco de referencia, y el nombre de la *topic* en donde publica la respectiva información. Si se vuelve a insertar el modelo SDF del robot se podrá observar un conjunto de rayos láser alrededor del sensor LIDAR como se muestra en la Figura 20. Estos rayos láser se van moviendo a medida que se mueve el robot y se puede observar que la información en la topic */scan* se va actualizando acorde a los obstáculos encontrados. Dicha información es presentada como un vector que contiene la distancia de cada obstáculo con respecto al

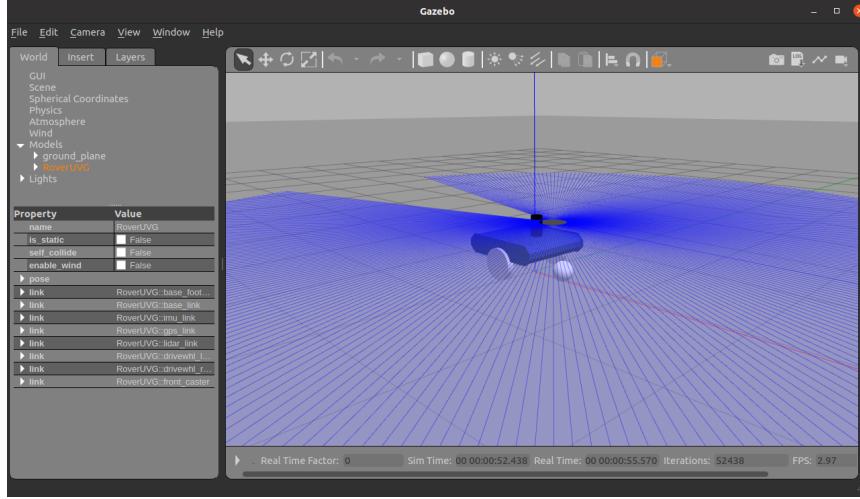


Figura 20: Visualización del URDF del Rover UVG en un mundo en Gazebo.

Para poder visualizar la información filtrada del sensor LIDAR se puede utilizar RVIZ. En donde agregando una ventana para ver mensajes tipo *LaserScan* y escribiendo el nombre de la *topic* (*/scan*) se puede visualizar los objetos que se encuentran dentro del rango de sensor LIDAR. Para ello se insertaron varios objetos cerca del robot como se muestra en la Figura 21, en distintas posiciones y con distintas geometrías para poder observar como serían detectados por el sensor LIDAR. En la Figura 22 se puede observar una ventana de RVIZ que nos muestra la nube de puntos detectada por el sensor LIDAR. Comparando la nube de puntos mostrada en la Figura 22 con la posición de los obstáculos mostrados en la

Figura 21, se puede comprobar que el sensor esta detectando correctamente cada objeto en su rango.

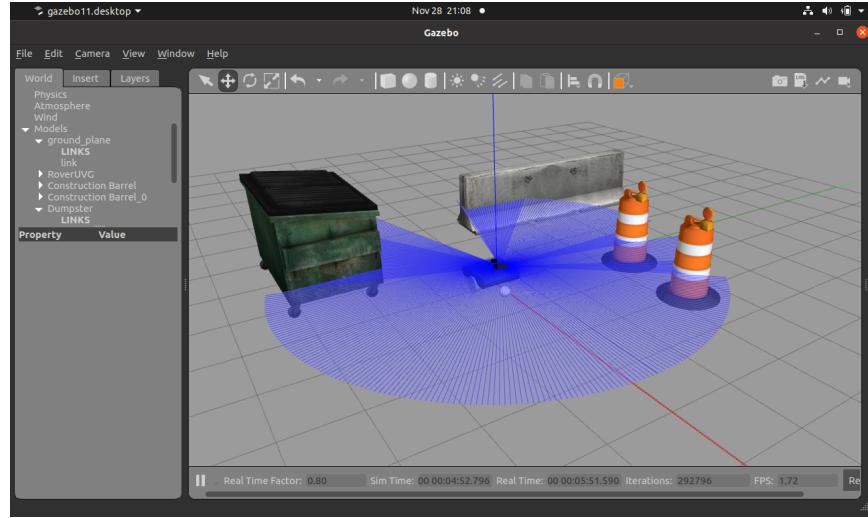


Figura 21: Visualización del robot rodeado de distintos objetos.

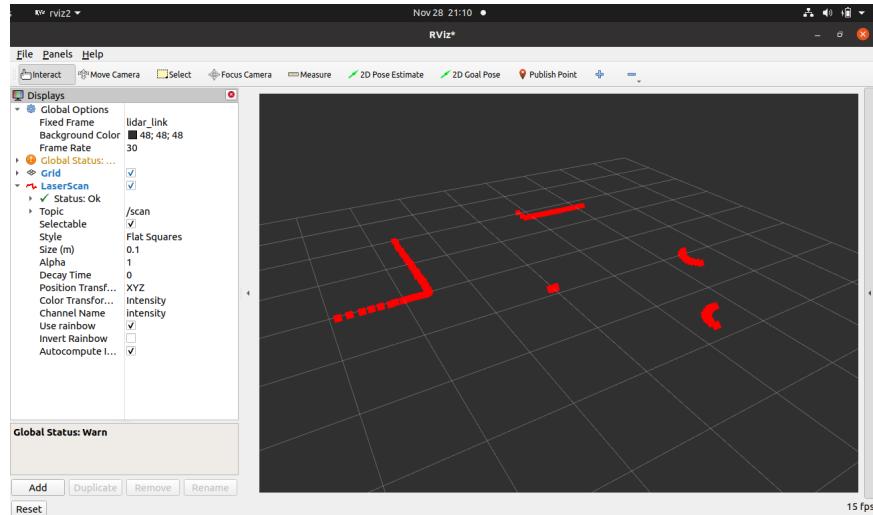


Figura 22: Visualización de los objetos detectados por el sensor LIDAR.

8.4.2. Configuraciones necesarias para SLAM

Uno de los paquetes disponibles en ROS 2 para la implementación de SLAM es el llamado *Slam Toolbox*. Este paquete es el más utilizado dentro del entorno de ROS 2 tanto para su ejecución en robots físicos como simulados. La guía de instalación de dicho paquete, así como de otras herramientas necesarias para realizar el mapeo se encuentran en repositorio mostrado en el Capítulo 13.1.

Para poder correr el algoritmo de SLAM por medio de dicho paquete es necesario:

1. Publicar información provista por un sensor LIDAR en la topic `/scan`.
2. Publicar la posición y orientación del robot en la topic `/odom`.
3. Tener el modelo URDF del robot definido con claridad.
4. Actualizar la posición y orientación del marco de referencia de la base del robot con respecto al marco de referencia de odometría.

Para el caso del robot simulado, el requisito 1 y 2 se definen dentro del modelo SDF del robot como se definió en secciones anteriores. Para el requisito 3, implica que el eslabón del LIDAR en el modelo URDF tenga asociado un marco de referencia con el mismo nombre del marco de referencia al cual se encuentra conectado la información del sensor LIDAR. En el caso del sensor LIDAR simulado, este parámetro también se configura en el modelo SDF. Por último para el requisito 4, la manera más sencilla de hacerlo es mediante el paquete llamado *Robot Localization*, el cual se encarga de tomar la información provista de la topic `/odom` e ir actualizando la posición y orientación del marco de referencia de la base del robot con respecto a marco de referencia de odometría.

8.4.3. Pruebas de SLAM

Para realizar las pruebas de SLAM se inserto el modelo SDF del robot en Gazebo así como distintos objetos dentro de un mundo virtual en Gazebo, como se muestra en la Figura 23.

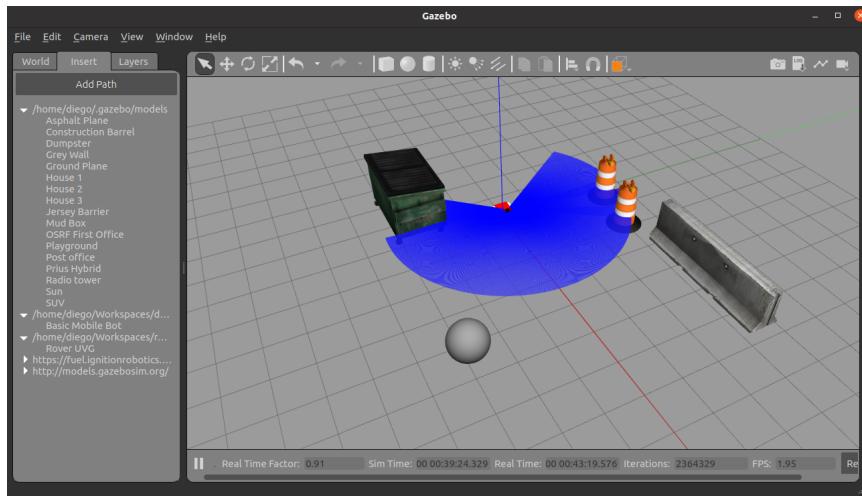


Figura 23: Visualización del robot rodeado de distintos objetos para probar el algoritmo de SLAM.

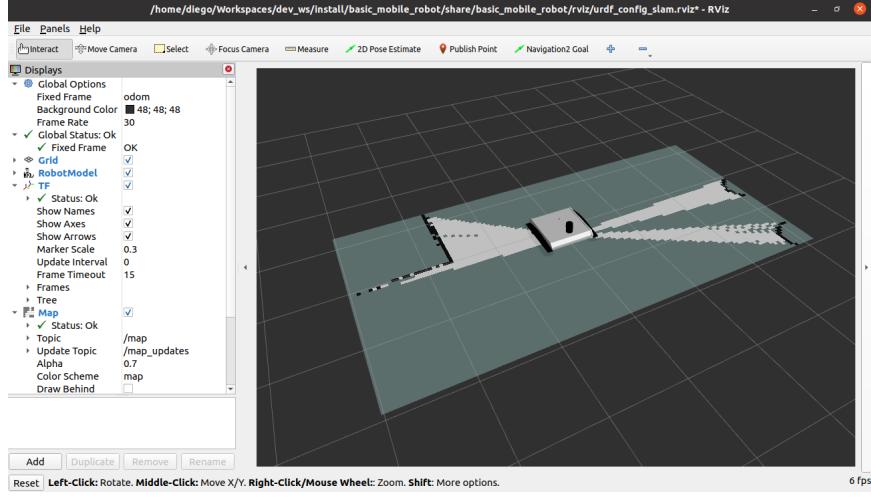


Figura 24: Visualización del mapa construido al momento de correr el archivo launch.

Se procedió a correr el archivo *launch* previamente definido y con ello se obtuvieron los resultados presentados en la Figura 24.

Como se puede observar en la Figura 24 solamente se muestra el mapa correspondiente a la nube de puntos que observa el sensor LIDAR por el momento. Por ello es necesario que el robot recorra la mayoría del terreno para que pudiera ir construyendo el mapa entero. Esto se realizó utilizando el *plugin* de *rqt_robot_steering* para poder navegar el robot a lo largo del terreno, como se muestra en la Figura 25.

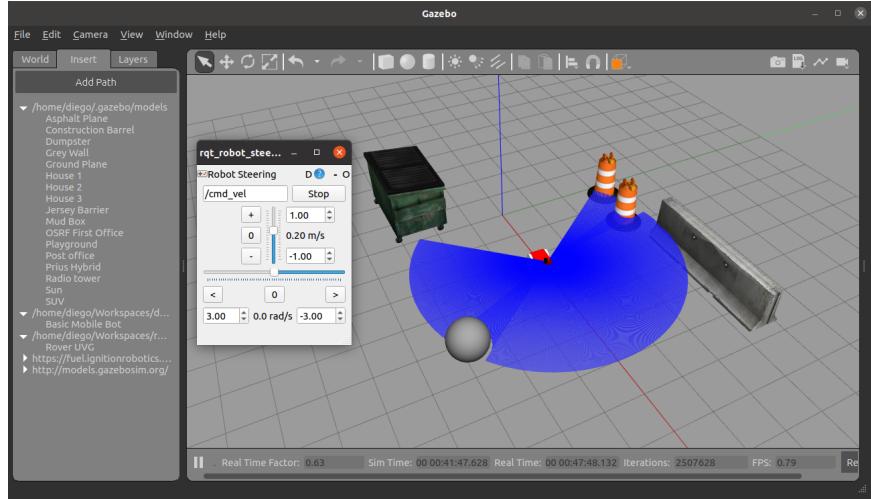


Figura 25: Visualización del robot en movimiento en conjunto con el *plugin* de *rqt_robot_steering*.

Luego de haber recorrido el mapa en la mayoría de sectores donde podría detectar la mayoría de los obstáculos, se obtuvieron los resultados presentados en la Figura 26. Como se puede observar en dicha figura, el mapeo si se hace acorde a los obstáculos presentes en el mundo virtual. Si bien el mapa no se encuentra completo ya que no se navegó el robot por todo el mundo, si se puede verificar que el algoritmo de SLAM funciona correctamente

dentro de la simulación en Gazebo.

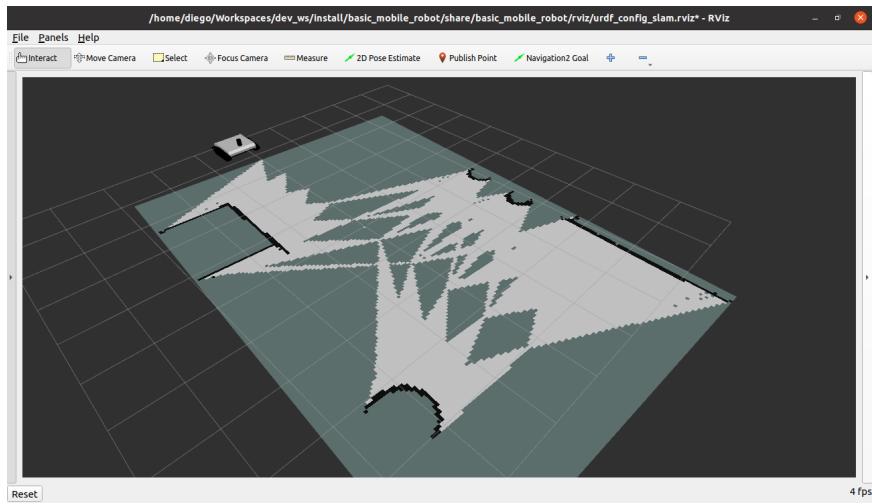


Figura 26: Visualización del mapa construido luego de la navegación del robot.

CAPÍTULO 9

Implementación Física

Luego de comprobar el funcionamiento de los programas de ROS en la simulación, se verificará que los mismos programas sean aceptables y funcionales para la implementación en el robot físico. Para ello se definió cuál es la computadora central óptima para el Rover UVG que será la encargada de ejecutar los programas de ROS 2. También se menciona cuáles son las diferencias existentes para la integración de los sensores y actuadores físicos con respecto a la simulación, y se determina si la integración de todos los módulos realmente cumplen con los objetivos esperados.

9.1. Selección de la computadora central adecuada

Para poder correr ROS 2 de manera eficiente se necesita un sistema operativo Ubuntu Linux (18.04) o superior. Se requiere la versión de 64 bits de Ubuntu y se debe contar con 4 GB de RAM dentro del ordenador [20]. Es por ello que comúnmente se hace uso de Raspberry Pi para poder ejecutar proyectos de ROS para distintas aplicaciones. Estas computadoras se caracterizan por ser ordenadores de tamaño bolsillo y poseer un sistema Linux embebido en su totalidad.

El modelo óptimo de Raspberry Pi para ejecutar proyectos de ROS 2 son las Raspberry Pi 4 con 4 GB de RAM (o superior). Esto debido a que los modelos previos no presentan la cantidad de memoria necesaria, ni tampoco cuentan con el procesador más potente y moderno de la versión 4. En el Cuadro 8 se presenta una tabla más detallada de las principales características de las versiones 3 y 4 de Raspberry Pi.

Característica	Raspberry Pi 4	Raspberry Pi 3
Procesador	Quad Core Cortex A-72 1,5 GHz	Quad Core Cortex A-53 1,4 GHz
Memoria RAM	1, 2, 4, 8 GB LPDDR4	1 GB LPDDR2
USB	2 x USB 2.0, 2 x USB 3.0	4 x USB 2.0
Alimentación	USB Tipo-C	microUSB
HDMI	2 x Micro HDMI	HDMI
Ethernet	Gigabit sin limitaciones	Gigabit hasta 300 Mbps
Wi-Fi	2,4 / 5 GHz	2,4 / 5 GHz
Bluetooth	5.0	4.2

Cuadro 8: Características para Raspberry Pi 3 y 4 [21].



Figura 27: Raspberry Pi 4 [21].

9.2. Módulos de percepción y actuación dentro del Rover UVG

El Rover UVG cuenta con varios sensores y actuadores que tienen la tarea principal de controlar dicho robot en determinados entornos, conocer la ubicación en tiempo real del robot y reconocer el entorno que rodea la misma. Específicamente el Rover UVG cuenta con un sensor propioceptivo como es el caso del sensor de odometría (que funciona mediante la lectura de los *encoders* instalados en las orugas del robot), también con sensores exteroceptivos como lo son los sensores DWM1001 y funciona en conjunto con el sistema de captura de movimiento Optitrack para conocer la ubicación y orientación del robot. También cuenta con un sensor LIDAR 2D que entrega la distancia de los objetos a su alrededor, con una Raspy Cam para poder realizar visión por computadora y finalmente con un brazo robótico que permite realizar distintas tareas dentro de su espacio de trabajo. Claramente el Rover UVG cuenta un motor en cada oruga del mismo y que es controlador por un microcontrolador ESP32 para la lectura de *encoders* y la recepción de los comandos de velocidad que vendrán desde ROS. En la Figura 28 se puede observar un diagrama que ejemplifica y describe cada uno de estos módulos en conjunto con la computadora central.

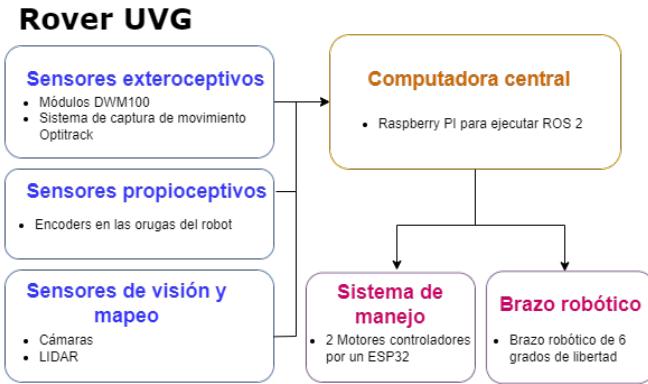


Figura 28: Diagrama de sensores/actuadores del Rover UVG.

9.3. Odometría y localización del robot físico

9.3.1. Odometría con encoders

Lo primero que se realizó fue crear un programa en Python para poder comunicarse con un ESP32 mediante Arduino, debido a que este es el microcontrolador que manipula las orugas del Rover UVG y se encarga de leer e interpreta la información de los *encoders*. Esto se realizó mediante comunicación serial y utilizando la librería JSON, con la cual se pueden enviar cadenas de información en forma de diccionarios. Esta librería permite la comunicación entre varios dispositivos seriales mediante la codificación y decodificación de los mensajes de manera sencilla.

Se procedió a crear un nodo llamado *hardware_arduino_odom* que se encarga de publicar la información de odometría que reciba por medio de la comunicación serial desde Arduino. Esto se realizó mediante la creación de una clase llamada *ArduinoCommunication* en la cual se crearon dos funciones. La primera función denominada *_init_* se encarga de establecer las características nodo, tales como el nombre del nodo (*hardware_arduino_odom*), el nombre de la topic a la cual publica la información (*/odom*), y la frecuencia a la cual publica dicha información (cada 0.5 segundos). La segunda función se encarga de

9.3.2. Localización con sensores DWM1001 y Optitrack

Para los módulos DWM1001 y el sistema de captura de movimiento Optitrack, se solicitó realizar el mismo procedimiento previamente mostrado con lo cual se tendrían tres nodos publicando información de odometría.

9.3.3. Pruebas de odometría y localización

Para la realización de todas las pruebas con el Rover UVG, se utilizó la plataforma del Laboratorio de Robótica ubicado en el CIT-116 de la Universidad del Valle de Guatemala.

Dicha plataforma, que se muestra en la Figura 29, esta junto al sistema de captura de movimiento Optitrack y a su vez, los módulos DMW1001 estan instalados en los paralelos de las cámaras. Esto nos permite realizar las pruebas con todos los módulos dentro de dicha plataforma.



Figura 29: Plataforma del Laboratorio del Robótica.

9.4. Control punto a punto con el robot físico

9.4.1. Comandos de velocidad para el Rover UVG

Para poder controlar la velocidad del robot físico, es necesario establecer comunicación de ROS hacia Arduino. Esto de igual forma, se realizó mediante comunicación serial, en donde se envían cadenas en formato JSON que codifica la velocidad para cada una de las orugas del robot. Por ello, fue necesario modificar el nodo *hardware_arduino_odom* para que fuera posible recibir comandos de velocidad por medio de mensajes establecidos por ROS llamados *Twist*. Este tipo de mensajes se utiliza para enviar/recibir comandos de velocidad dividio en su componente angular y lineal, algo muy común cuando se trabaja con robots móviles diferenciales. Para ello fue necesario importar *Twist* de la librería *geometry_msgs.msg* con lo cual fue posible definir variables que recibieran dicho información de la velocidad y posteriormente fuese enviada por medio de comunicación serial a Arduino. A continuación se presentan las modificaciones hechas al código visto en la sección anterior, con el cual se define la suscripción a la *topic* que contendrá la información de velocidad y se envía la información por el puerto serial.

Se puede observar ahora el nodo *hardware_arduino_odom* se suscribe a una *topic* (*/cmd_vel*) en la cual constantemente se está publicando la velocidad a la que se desea que se mueva el robot. Posteriormente en la función *subscribe_message* se crea un diccionario que almacena la información recibida y la envia a Arduino.

9.4.2. Pruebas de implementación del controlador PID

Las pruebas de control punto a punto se realizaron de igual forma sobre la plataforma del Laboratorio de Robótica y utilizando la información de localización que provee el sistema de captura de movimiento Optitrack, debido a que este sistema nos entrega tanto posición como orientación (a diferencia de los módulos DWM1001) y es mucho más preciso que la información que proveen los encoders. Para ello se realizó un archivo *launch* llamado *rover_uvg_control_pap.launch.py* que corre el nodo de localización del Optitrack y RVIZ para poder visualizar el robot. Luego lo que se realiza es correr el nodo de *simple_controller*, el cual es el controlador PID para el control punto a punto (el mismo utilizado para la simulación) y se establece la coordenadas deseada. Para todas las corridas se estableció las coordenadas (0,0) y se orientó al Rover UVG de manera que tuviera que dar la vuelta completa hasta entre perpendicular al objetivo, como se muestra en la Figura 30.



Figura 30: Rover UVG sobre la plataforma para pruebas de control punto a punto.

En el repositorio presentado en el Capítulo 13.1 se presenta un video que muestra distintas corridas de pruebas de control punto a punto, en donde se cambia la posición y orientación inicial del Rover UVG para validar que desde cualquier punto inicial pueda converger al punto deseado. Para todas estas corridas se obtuvo la posición final con lo cual se pudo obtener un estimado del porcentaje de error como se muestra en el Cuadro 9.

Corrida	Coordenada	Valor deseado (m)	Resultado (m)	Porcentaje de error (%)
1	x	0.0	0.095	5.26
	y	0.0	0.081	7.49
2	x	0.0	0.05	4.76
	y	0.0	0.10	9.09
3	x	0.0	0.07	6.54
	y	0.0	0.06	5.60
4	x	1.0	1.01	0.99
	y	1.0	1.09	8.25
5	x	1.0	1.08	7.40
	y	1.0	1.11	9.91

Cuadro 9: Porcentaje de error para 5 corridas en la implementación física. Porcentaje de error para 5 corridas.

Con ello se obtuvo el promedio de los porcentajes de error tanto para la coordenada en x , como en y para la implementación física del controlador punto a punto como se muestra en el Cuadro 10.

Coordenada	Porcentaje de error (%)
x	4.99
y	8.06

Cuadro 10: Porcentaje de error para cada coordenada después de las 5 corridas en la implementación física.

CAPÍTULO 10

Conclusiones

- El entorno de ROS 2 permite el desarrollo de proyectos complejos de una manera muy sencilla y eficiente, debido a que cuenta con gran cantidad de paquetes que se encargan de ayudar a la construcción y simulación de un robot móvil, en este caso particular. En donde la tarea principal para el desarrollador es conocer las cualidades que tendrá el robot en desarrollo para utilizar los recursos disponibles en ROS que ayuden a lograr los objetivos del mismo.
- El controlador PID con acercamiento exponencial genera resultados muy satisfactorios al momento de realizar los experimentos de control punto a punto dentro de la simulación en Gazebo, en donde se obtuvieron en promedio porcentajes de error inferiores a un 4% con respecto a los valores reales.
- Las pruebas realizadas en la simulación son realmente útiles para poder probar la funcionalidad de algoritmos desarrollados para la implementación física, como se puede observar en los resultados del control punto a punto con el Rover UVG en donde los porcentajes de error no subieron más que un 5.15% con respecto a los valores simulados, aún teniendo problemas con el sistema mecánico del robot.
- Gracias a la arquitectura de ROS es muy sencilla la integración de múltiples módulos externos, debido a que se puede trabajar cada uno de estos de manera independiente y sin depender de ningún proceso central. Esto ayuda a agilizar las pruebas que se realizan con cada módulo y las fallas que puedan darse sobre un módulo no interrumpen el funcionamiento de los demás.
- El modelo óptimo de computadora central para el Rover UVG es la Raspberry Pi de 4 GB de memoria RAM, debido a que permite acceder a todas las funcionalidades que nos provee ROS 2.

CAPÍTULO 11

Recomendaciones

- Se recomienda realizar pruebas dentro de la simulación sobre un entorno que sea lo más parecido al territorio donde se harán las pruebas físicas del robot, de manera que se pueda obtener resultados muy similares.
- Se recomienda investigar sobre sensores/actuadores típicos que se utilicen dentro de robots comandados con ROS, debido a que esto puede ayudar a facilitar y a ampliar las capacidades del Rover UVG.
- Se recomienda tener entornos de prueba amplios para el robot físico, debido a que el sistema mecánico del robot no permite dar vueltas cerradas, por lo que las pruebas de desplazamiento y rotación se ven afectadas si se tiene un espacio de prueba reducido.
- Se recomienda tener entornos de prueba amplios para el robot físico, debido a que el sistema mecánico del robot no permite dar vueltas cerradas, por lo que las pruebas de desplazamiento y rotación se ven afectadas si se tiene un espacio de prueba reducido.
- Es sumamente importante ir actualizando las capacidades de la computadora central a medida que se vayan utilizando módulos más complejos debido a que si bien el uso de un Raspberry Pi 4, ayuda a cumplir los objetivos presentados en este trabajo de graduación, los algoritmos como SLAM, visión por computadora, entre otros, saturan en gran manera el nivel de procesamiento de la misma.
- Para proyectos que se realicen con robots móviles, es de suma importancia que la eficiencia del sistema mecánico y el manejo del robot sea muy alto para poder obtener los mejores resultados con los controladores u algoritmos que se realicen para dicho robot.

CAPÍTULO 12

Bibliografía

- [1] H. Sagastume, “Diseño Mecánico, Selección de Motores e Implementación de Sensores para un Robot Explorador Modular,” Tesis de licenciatura, Universidad Del Valle de Guatemala, 2021.
- [2] D. Snider, M. Mirvish, M. Barcis y V. A. Tezer, “University Rover Challenge: Tutorials and Team Survey,” en *Robot Operating System (ROS)*, Springer, 2019, págs. 315-370.
- [3] S. Gatesichapakorn, J. Takamatsu y M. Ruchanurucks, “ROS based autonomous mobile robot navigation using 2D LiDAR and RGB-D camera,” en *2019 First international symposium on instrumentation, control, artificial intelligence, and robotics (ICA-SYMP)*, IEEE, 2019, págs. 151-154.
- [4] J. Archila, “Diseño e implementación de capacidades automáticas de navegación para un Robot Explorador Modular,” Tesis de licenciatura, Universidad Del Valle de Guatemala, 2021.
- [5] M. Izeppi, “Aplicación de Herramientas de Aprendizaje Reforzado y Aprendizaje Profundo en Simulaciones de Robótica de Enjambre con Restricciones Físicas,” Tesis de licenciatura, Universidad Del Valle de Guatemala, 2022.
- [6] M. Quigley, K. Conley, B. Gerkey y col., “ROS: an open-source Robot Operating System,” en *ICRA workshop on open source software*, Kobe, Japan, vol. 3, 2009, pág. 5.
- [7] R. Merino López, “Creación de modelo URDF del robot manfred,” Tesis de mtría., Universidad Carlos III de Madrid, 2014.
- [8] Open Robotics, *Nodes*, <http://wiki.ros.org/es/Nodes>, 2021.
- [9] Open Source, *Gazebo*, <https://gazebosim.org/home>, 2021.
- [10] N. Koenig y A. Howard, “Design and use paradigms for gazebo, an open-source multi-robot simulator,” en *2004 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)(IEEE Cat. No. 04CH37566)*, IEEE, vol. 3, 2004, págs. 2149-2154.
- [11] Open Source, *Rviz*, <http://wiki.ros.org/rviz>, 2021.
- [12] M. Sokolov, I. Afanasyev, R. Lavrenov, A. Sagitov, L. Sabirova y E. Magid, “Modelling a crawler-type UGV for urban search and rescue in Gazebo environment,” en *Artificial Life and Robotics (ICAROB 2017), International Conference on*, 2017, págs. 360-362.

- [13] D. Ferrer Calatayud, “Adquisición de datos IMU con un sistema embebido,” Tesis doct., Universitat Politècnica de València, 2015.
- [14] *Sistema de posicionamiento global GPS*, <https://www.azimutmarine.es/sistema-posicionamiento-gps>, 2022.
- [15] M. Camara, *Productos de banda ultra ancha*, <https://www.comunicacionesinalambricashoy.com/productos-de-banda-ultra-ancha/>, 2020.
- [16] L. Martínez, *Navegación SLAM: robots que construyen mapas*, <https://clem.es/noticia/navegacion-slam-robots-que-construyen-mapas-514>, 2020.
- [17] T. Dirk, *Changes between ROS 1 and ROS 2*, <http://design.ros2.org/articles/changes.html>, sep. de 2015.
- [18] *Ubuntu (Debian)*, <https://docs.ros.org/en/foxy/Installation/Ubuntu-Install-Debians.html>, 2022.
- [19] S. Macenski, T. Foote, B. Gerkey, C. Lalancette y W. Woodall, “Robot Operating System 2: Design, architecture, and uses in the wild,” *Science Robotics*, vol. 7, n.º 66, eabm6074, 2022. DOI: 10.1126/scirobotics.abm6074. dirección: <https://www.science.org/doi/abs/10.1126/scirobotics.abm6074>.
- [20] *What is ROS*, <https://ubuntu.com/robotics/what-is-ros>, 2022.
- [21] A. Huertos, *Raspberry Pi 4 vs Raspberry Pi 3*, <https://computerhoy.com/noticias/tecnologia/raspberry-pi-4-vs-raspberry-pi-3-mejora-nuevo-modelo-443801>, 2019.

CAPÍTULO 13

Anexos

13.1. Repositorios de Github

A continuación se presenta el link al repositorio de Github en donde se presenta todo el proyecto de simulación presentado en el capítulo 8: https://github.com/men18300/ros2_roveruvg_ws.git

También se encuentra el link al repositorio de Github en donde se presenta todo el proyecto para el Rover UVG físico presentado en el capítulo 9: https://github.com/men18300/ros2_roveruvg_ws.git

