



UNIVERSIDAD
DE MÁLAGA



TRABAJO FIN DE GRADO

DESARROLLO DE UN SIMULADOR EN GAZEBO PARA EL ROBOT MÓVIL PANTER

DEVELOPMENT OF A GAZEBO SIMULATOR FOR THE PANTER MOBILE ROBOT

GRADO EN INGENIERÍA ELECTRÓNICA, ROBÓTICA Y MECATRÓNICA

ESCUELA DE INGENIERÍAS INDUSTRIALES

UNIVERSIDAD DE MÁLAGA

Autor: Jorge Avila Orero

Tutor: Javier Serón Barba

Área de conocimiento: Ingeniería de Sistemas y Automática

Cotutor: David Padial Allue

Área de conocimiento: Ingeniería de Sistemas y Automática

Málaga, julio de 2025



**DECLARACIÓN DE ORIGINALIDAD DEL
PROYECTO/TRABAJO FIN DE GRADO**

D./ Dña.:

DNI/Pasaporte: Correo electrónico:

Titulación:

Título del Proyecto/Trabajo:

.....
.....

DECLARA BAJO SU RESPONSABILIDAD

Ser autor/a del texto entregado y que no ha sido presentado con anterioridad, ni total ni parcialmente, para superar materias previamente cursadas en esta u otras titulaciones de la Universidad de Málaga o cualquier otra institución de educación superior u otro tipo de fin.

Así mismo, declara no haber trasgredido ninguna norma universitaria con respecto al plagio ni a las leyes establecidas que protegen la propiedad intelectual, así como que las fuentes utilizadas han sido citadas adecuadamente.

En Málaga, a de de 20.....

Fdo.:

TÍTULO

Resumen del proyecto

Los simuladores representan una herramienta esencial en los proyectos de robótica, ya que permiten realizar numerosas pruebas sin necesidad de que el robot se encuentre físicamente operativo. Entre las principales ventajas del uso de un simulador, destaca la posibilidad de obtener datos sobre el comportamiento del vehículo en distintos escenarios, como terrenos no estructurados. Asimismo, permiten la incorporación de sensores virtuales para registrar la respuesta de los actuadores ante diversas señales de entrada (inputs), así como verificar el correcto funcionamiento del software antes de su implementación en el vehículo real.

El vehículo tiene grandes dimensiones, y realizar pruebas es complejo, ya que conlleva mucho coste en tiempo, y la necesidad de estar presencialmente en el lugar de trabajo. El simulador ofrece la gran ventaja de utilizar el gemelo digital de vehículo, sin de estar presente y nos permite ubicarlo en cualquier escenario.

El proyecto inicia con un diseño proporcionado en SolidWoks, que necesita que su longitud sea disminuida en 40cm para cumplir con las dimensiones actuales del vehículo. El vehículo ha sido recortado y, a su vez se han eliminando elementos no esenciales en la simulación, consiguiendo que esta sea más ligera.

El simulador para el vehículo Panter se ha realizado en la plataforma de simulación robótica Gazebo, utilizando la versión Ignition Fortress.

Para la descripción del robot en Gazebo se ha utilizado URDF, que ofrece mayor interconectividad con el middleware de robótica seleccionado, y a su vez es compatible con Gazebo.

El simulador robótico es programable a través del middleware de robótica ROS 2 Humble. El programa desarrollado permite el control mediante teclado de la dirección del vehículo, utilizando direccionamiento de Ackermann, y el desplazamiento de las ruedas se realiza introduciendo inputs de esfuerzo (torque) sobre la unión de la mangueta y la rueda, simulando así el funcionamiento de la dirección y del motor.

A su vez, se ha introducido un sensor de fuerza y toque en las ruedas, comprobando el torque real que tenemos en distintos escenarios, y que nos permite comprobar el funcionamiento de Panter.

Palabras clave: palabra1, palabra2, palabra3.

Índice

1. Introducción	1
1.1. Antecedentes	1
1.2. Objetivos y tareas	2
1.3. Motivación	3
1.4. Contenido de la memoria	4
2. Estado del Arte	5
2.1. Robótica móvil	5
2.2. Simulador de vehículos	7
2.3. Posibles simuladores	7
2.4. Middleware robótica	9
2.5. Posibles Middlewares	9
2.6. Descripción del robot: URDF vs SDF	10
3. Desarrollo del simulador	13
3.1. Entorno de trabajo y herramientas	13
3.1.1. ROS2 Humble	14
3.1.2. Gazebo	15
3.1.3. Compatibilidad ROS2 y Gazebo	17
3.1.4. SolidWorks	18
3.1.5. Visual Studio Code	18
3.1.6. Git y GitHub	19
3.1.7. Términos	19

3.2.	Modelo SolidWorks	21
3.2.1.	Exportar de SolidWorks a URDF	23
3.3.	Jerarquía del proyecto	25
3.3.1.	control_pkg	25
3.3.2.	description_pkg	26
3.3.3.	bringup_pkg	27
3.4.	Descripción Modelo Robot Panter	28
3.4.1.	Estructura	28
3.4.2.	Características para el control	30
3.5.	Programación en ROS	31
3.5.1.	Estructura	32
3.5.1.1.	Cabezera (.hpp)	32
3.5.1.2.	Cuerpo (.cpp)	32
3.5.2.	Control por velocidad	34
3.5.3.	Control por esfuerzo	35
3.6.	CMakeLists.txt	37
3.7.	package.xml	38
3.8.	Lanzadores (.launch.py)	38
3.8.1.	description_pkg/sim_urdf.launch.py	39
3.8.2.	control_pkg/ros.launch.py	39
3.8.3.	bringup_pkg/proyect.launch.py	40
3.9.	RViz	40
3.10.	Ficheros de configuración	41
3.10.1.	ros_gz_bridge.yaml	41
3.10.2.	control_config.yaml	42
3.11.	Físicas	43
3.11.1.	Motor de físicas Gazebo	43
3.11.2.	Colisiones	43
3.11.3.	Masas	44
3.11.4.	Inercias	47

3.11.4.1. Ajuste inercia Chasis	49
3.11.5. Rozamiento	52
3.11.6. Suspensiones	53
3.11.7. Reductora	57
3.12. Plugins	57
3.12.1. Plugin Direccionamiento de Ackermann	57
3.12.2. Plugin sensor force-torque	59
3.12.3. Framework ros2_control	59
3.13. Escenario Simulación	60
4. Resultados	63
4.1. Escenario por defecto	63
4.2. Escenario personalizado	64
5. Conclusiones y líneas futuras	67
5.1. Conclusiones	67
5.2. Líneas futuras de trabajo	67
Bibliografía	69
1. Aplicación del Simulador	71
1.1. Instalación del Simulador de Panter	71
1.2. Uso del Simulador	75

Índice de Tablas

1.	Comandos del teclado para controlar el robot	33
2.	Masas de los componentes de Panter.	45
3.	Distribución de masas por rueda (en kg).	46
4.	Masas de los elementos asociados al chasis de Panter.	46
5.	Masas de los componentes de Panter.	47

Índice de Ilustraciones

1.	Vehículo Panter de Molveco	1
2.	Reconstrucción Panter	2
3.	Campos en robótica	4
1.	Direccionamiento Diferencial	5
2.	Direccionamiento con ruedas Mecanum	6
3.	Direccionamiento de Ackermann	6
4.	Simulador Gazebo	7
5.	Simulador Webots	8
6.	Simulador CoppeliaSim	8
7.	Simulador CARLA	8
1.	Esquema funcionamiento ROS	14
2.	ROS 2 Humble	14
3.	Gazebo Classic	15
4.	Ignition Gazebo	16
5.	Gazebo Sim	16
6.	Gazebo Ignition Fortress	17
7.	Tabla compatibilidad ROS-Gazebo [3]	18
8.	Logo SolidWorks	18
9.	Logo Visual Studio Code	18
10.	Logo Git	19
11.	Repositorio proyecto simulador en GitHub [4]	19
12.	Panter tras extrusión	22

13.	Panter reconstruido	23
14.	Interfaz herramienta SW2URDF	24
15.	Configuración árbol de entidades, sw2urdf exporter [6]	25
16.	Paquetes proyecto	25
17.	Estructura paquete control_pkg	26
18.	Estructura paquete description_pkg	27
19.	Estructura paquete bringup_pkg	28
20.	Parent y child en un joint	29
21.	Descripción Panter, con eje de los Joints activos	30
22.	Panter en RViz	41
23.	Flujo ros_gz_bridge	41
24.	Estructura de ros_gz_bridge.yaml [12]	42
25.	Representación colisiones de Panter en Gazebo	44
26.	Desmontaje para pesaje	44
27.	Pesaje rueda	45
28.	Representación inercias Panter	49
29.	Sistema de referencia Panter	50
30.	Esquema de ajuste inercia Chasis	51
31.	Inercia Chasis ajustada	52
32.	Enter Caption	53
33.	Esquema conjunto de suspensión	54
34.	Conjunto de suspensión	54
35.	Planteamiento 1 joints	55
36.	Planteamiento 3 joints	56
37.	Malla escenario en Blender	60
38.	Escenario en Blender	61
39.	Enter Caption	62
40.	Enter Caption	62
1.	Enter Caption	65
1.	Enter Caption	74

2.	Play Gazebo	76
3.	Enter Caption	76
4.	Controladores iniciados correctamente	77

Capítulo 1

Introducción

1.1. Antecedentes

Este Trabajo de Fin de Grado se enmarca en el proyecto Panter de la Universidad de Málaga, que persigue el desarrollo de un vehículo eléctrico de rescate a partir del chasis del Panter 4×4 de la marca Movelco. Los componentes del vehículo serán modificados con el objetivo de mejorar el rendimiento y adaptarlo a misiones de emergencia en entornos no estructurados.

El vehículo es el mostrado en la Figura 1, que pertenece al Departamento de Ingeniería de Sistemas y Automática de la Universidad de Málaga.



Figura 1. Vehículo Panter de Molveco

Para el desarrollo de este proyecto se toma como referencia el trabajo anterior realizado

por el departamento: 'Modelado 3d de un vehículo eléctrico para misiones de rescate en entornos no estructurados' [1].

El vehículo actualmente ha sufrido múltiples modificaciones; la Figura 2 representa su estado actual.



Figura 2. Reconstrucción Panter

1.2. Objetivos y tareas

El objetivo principal de este proyecto es crear un simulador para el robot móvil Panter. Este simulador se basa en la plataforma de simulación Gazebo Fortress y se ha implementado la interoperabilidad con el middleware ROS 2 Humble, lo que permite gestionar tanto el control como la obtención de datos del robot Panter.

Para conducir Panter se han definido dos modos de operación:

- Control mediante teclado con referencias de velocidad.
- Control mediante teclado con referencias de torque aplicado a cada rueda.

Como objetivo secundario, se ha añadido la lectura del torque en cada una de las ruedas, de modo que sea posible analizar cómo se comporta 'Panter' en diferentes escenarios.

A continuación se describen las principales tareas realizadas:

1. Investigar, seleccionar y configurar ROS 2 y Gazebo, asegurando su correcta compatibilidad.
2. Aprendizaje de SolidWorks para la modificación de modelado 3D.

3. Investigación y desarrollo de descripción robot en URDF frente SDF.
4. Investigación de estructura y jerarquía necesaria para el desarrollo del proyecto.
5. Diseñar el entorno de simulación en Gazebo, incluyendo el world y la configuración inicial.
6. Programar la comunicación entre ROS 2 y Gazebo, implementando los nodos necesarios para el control y la lectura de datos.
7. Crear los ficheros de lanzamiento de los programas.
8. Verificar y experimentar el correcto funcionamiento del simulador en distintos escenarios de prueba.
9. Elaborar la documentación detallada del proyecto.

1.3. Motivación

La motivación de este proyecto nace de mi deseo de involucrarme en una experiencia de robótica real al finalizar el Grado en Ingeniería Electrónica, Robótica y Mecatrónica, con Mención en Robótica y Automatización. A través de este trabajo, he tenido la oportunidad de explorar un área totalmente nueva para mí: la creación de un simulador desde cero. Esto complementa los conocimientos que ya adquirí durante la carrera en montaje y programación de robots, y me ha permitido profundizar en las principales fases de desarrollo de un proyecto robótico, definidas en la Figura 3.

En este proyecto se emplean las herramientas más extendidas en la actualidad para la simulación de robots y el desarrollo de middleware robótico, lo que añade un valor profesional significativo.

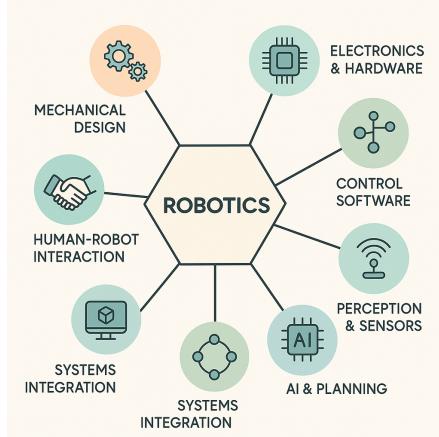


Figura 3. Campos en robótica

1.4. Contenido de la memoria

La memoria contiene la siguiente estructura:

Capítulo 1. Introducción. En este capítulo se abordan los antecedentes, las tareas y objetivos, la motivación y el contenido de la memoria.

Capítulo 2. Estado del Arte. Se lleva a cabo un estudio preliminar de los temas vinculados al proyecto y una revisión de la literatura más reciente.

Capítulo 3. Desarrollo del Simulador. Expone el contenido del proyecto, cómo se ha llevado a cabo, su estructura y jerarquía.

Capítulo 4. Resultados.

Capítulo 5. Conclusiones.

Capítulo 2

Estado del Arte

2.1. Robótica móvil

La robótica móvil abarca el diseño, la construcción y el control de máquinas autónomas o semiautónomas que se desplazan en entornos dinámicos. Estos robots integran avanzados sistemas de percepción y navegación como LIDAR, cámaras y sensores iniciales que les permiten localizarse con precisión, trazar rutas óptimas y adaptarse en tiempo real a obstáculos o imprevistos.

Los robots móviles pueden adoptar diversas arquitecturas y plataformas según la aplicación y el entorno en que operen. Entre los tipos más comunes se encuentran:

- **Ruedas diferenciales:** dos ruedas motrices y ruedas locas de apoyo; muy usados en investigación y educación. (Por ejemplo: TurtleBot, Roomba).



Figura 1. Direccionamiento Diferencial

- **Omnidireccionales** (ruedas Mecanum o esferas): facilitan desplazamientos laterales y giros en el lugar; óptimos en espacios reducidos. (Por ejemplo: MiR100, Clearpath

Ridgeback).

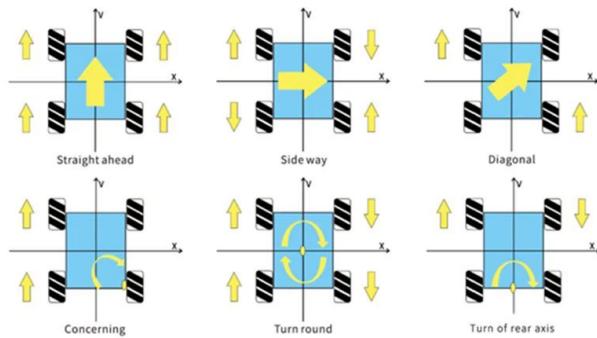


Figura 2. Direccionamiento con ruedas Mecanum

- **Ackermann:** imitan la geometría de dirección de un coche, con ruedas delanteras giratorias y traseras fijas; ideales para pruebas de conducción y mapeo en entornos reales. (Por ejemplo: vehículo común).

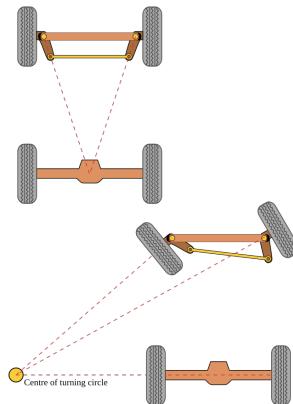


Figura 3. Direccionamiento de Ackermann

- **Robots con orugas:** alta tracción y estabilidad en terrenos irregulares o blandos.
- **Robots con patas** (cuádrupedos, bípedos, hexápodos): imitan la locomoción de un animal. Superan obstáculos complejos y desniveles.
- **Drones multirrotor (UAVs):** vehículos aéreos para inspección y cartografía.
- **Vehículos submarinos (AUV/ROV):** sumergibles autónomos o guiados para explotación y salvamento.

2.2. Simulador de vehículos

Los simuladores de vehículos robóticos son entornos virtuales que imitan el comportamiento físico, sensorial y de control de robots móviles, desde pequeñas plataformas diferenciales hasta vehículos autónomos complejos. Su principal ventaja es permitir el desarrollo y la validación de algoritmos de navegación, percepción y actuación sin recurrir a costosos prototipos reales, reduciendo riesgos y acelerando ciclos de diseño.

2.3. Posibles simuladores

Las plataformas más destacadas para la simulación de vehículos robóticos incluyen:

- **Gazebo / Ignition Gazebo:** muy integrado con ROS (1 y 2), ofrece múltiples motores de física (ODE, Bullet, DART) y un ecosistema de plugins para sensores (LIDAR, cámaras, IMU). Ideal para prototipos que requieran interoperabilidad con nodos reales de ROS.

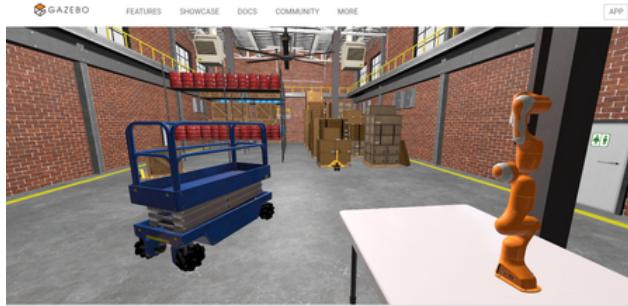


Figura 4. Simulador Gazebo

- **Webots:** incluye un editor gráfico 3D, control en Python, C/C++ y MATLAB, y ejemplos preconfigurados para plataformas diferenciales y Ackermann. Su licencia Apache 2.0 facilita su uso académico y en proyectos de investigación.



Figura 5. Simulador Webots

- **CoppeliaSim (V-REP)**: permite scripting distribuido en Lua, Python o C/C++, soporta motores de física avanzados (MuJoCo, Bullet, ODE, Vortex, Newton) y ofrece herramientas para ajustar trayectorias y parámetros dinámicos.

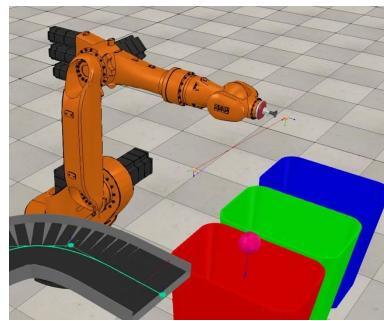


Figura 6. Simulador CoppeliaSim

- **CARLA**: construido sobre el motor de videojuegos Unreal Engine, proporciona entornos urbanos con semáforos, peatones y tráfico controlado por IA.

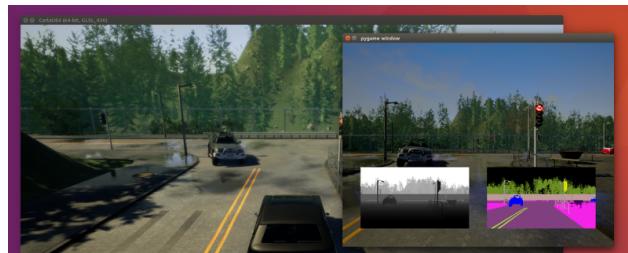


Figura 7. Simulador CARLA

2.4. Middleware robótica

Un middleware de robótica es un software encargado de simplificar la complejidad de la comunicación, coordinación y gestión de los distintos componentes de un sistema robótico.

En lugar de que cada nodo, sensor o actuador implemente su propio mecanismo de transmisión de datos y sincronización, el middleware ofrece servicios comunes (paso de mensajes, sincronización de relojes, descubrimiento de nodos, gestión de parámetros) que facilitan el desarrollo modular, escalable y distribuido de aplicaciones.

Se podría comparar un middleware de robótica con un “sistema operativo” específico para robots: proporciona infraestructuras de comunicación (publish/subscribe, request/response, transferencia de ficheros o parámetros), mecanismos de serialización y deserialización de datos, y herramientas de monitorización y diagnóstico.

Un middleware de robótica no solo acelera el desarrollo de software para robots, sino que aporta robustez, escalabilidad y modularidad a arquitecturas que, de otro modo, serían difíciles de mantener y evolucionar.

En la actualidad, contar con un middleware sólido ya no es una opción, sino un requisito casi imprescindible para cualquier proyecto robótico.

2.5. Posibles Middlewares

- **ROS 2** se posiciona hoy como la referencia indiscutible en robótica general: combina un ecosistema muy maduro con las ventajas del estándar DDS (QoS, descubrimiento, escalabilidad). Es la opción recomendada para la mayoría de proyectos académicos e industriales que requieran flexibilidad, compatibilidad y soporte comunitario.
- **YARP** sigue siendo una opción sólida en el ámbito de los humanoides gracias a su flexibilidad a la hora de conectar puertos en tiempo de ejecución y su eficiencia en entornos de procesamiento visual.
- **LCM** destaca en aplicaciones embebidas donde la latencia debe minimizarse al máximo, como robots de competición o sistemas de control ultra-precisos.

- **Orcos RTT** es bueno para desarrollos que demandan bucles de control en tiempo real, especialmente en robots manipuladores o prótesis.
- **ROS-Industrial** resulta la alternativa más consolidada para introducir ROS en entornos de automatización industrial, aprovechando la colección de drivers de PLC y protocolos de campo.
- Los middlewares emergentes como **Micro-ROS** o **Locus** son cada vez más relevantes en IoT y microcontroladores, abriendo la puerta a integrar pequeños dispositivos con arquitecturas ROS 2.

2.6. Descripción del robot: URDF vs SDF

En el contexto de la simulación de robots, conviene analizar dos formatos de descripción que suelen emplearse: URDF y SDF. Cada uno presenta fortalezas y limitaciones que impactan directamente en la interoperabilidad y en la fidelidad de la simulación.

- **SDF (Simulation Description Format)**

- **Características principales**

- Es el formato nativo de Gazebo e Ignition Fortress, basado en XML, pensado para describir no solo la cinemática de un robot, sino también todos los elementos que componen un “mundo” de simulación: terrenos, luces, obstáculos y propiedades físicas.
 - Permite definir enlaces (**<link>**) y articulaciones (**<joint>**) con geometrías tanto para visualización como para colisión, especificando parámetros avanzados de física (coeficientes de fricción estática y dinámica, amortiguamiento, restitución, etc.).
 - Incluye de forma nativa la posibilidad de agregar sensores (cámaras, LiDAR, IMU), así como plugins personalizados de control o de percepción, y bloques **<world>** completos para configurar el entorno.

- **Ventajas**

- Ofrece un modelado físico más detallado y realista: se pueden ajustar con precisión los parámetros de inercia, fricción y motores de física.
- Facilita la inclusión y configuración de múltiples sensores sin necesidad de recurrir a archivos externos o plugins adicionales.
- Agrupa en un mismo archivo la descripción del robot y del entorno, simplificando el mantenimiento de escenarios complejos.

– **Limitaciones**

- No es interpretado de forma nativa por ROS 2: para integrar un modelo SDF en un nodo de ROS 2 es necesario usar herramientas como `ros_ign_bridge` o paquetes específicos (`gz_ros2_control`), lo que añade complejidad en la configuración inicial.
- No existe un equivalente a XACRO en SDF, de modo que insertar macro-parámetros o repetir estructuras (por ejemplo, múltiples ruedas idénticas) puede resultar más pesado.
- La curva de aprendizaje es más pronunciada: aprender todas las etiquetas y jerarquías de SDF (especialmente en sus versiones más recientes) requiere más tiempo que en URDF.

• **URDF (Unified Robot Description Format)**

– **Características principales**

- Diseñado para integrarse de forma nativa con todo el ecosistema ROS (ROS 1 y ROS 2), prioriza la definición de la cinemática y la geometría visual/colisión de enlaces y articulaciones.
- Su sintaxis basada en XML, junto a la herramienta XACRO, facilita parametrizar dimensiones y reutilizar fragmentos (por ejemplo, definir una plantilla para cada rueda o sensor).
- En URDF se describe la masa, el centro de gravedad y el tensor de inercia de cada enlace, pero no se incluyen por defecto parámetros avanzados de fricción o restitución.

- Ventajas

- Integración directa con nodos y paquetes de ROS 2.
- Más simple y legible para describir la estructura cinemática pura de un robot: fácil de depurar y editar cuando solo se trabajan aspectos geométricos o de articulaciones.
- Gracias al paquete `gazebo_ros`, Gazebo convierte automáticamente un URDF en SDF al iniciar la simulación, de modo que el usuario puede seguir trabajando sobre URDF y aprovechar parte de las ventajas de SDF sin esfuerzo extra.

- Limitaciones

- URDF no incluye nativamente bloques para especificar sensores complejos, ni detalles avanzados de física. Para ello, es necesario insertar en el URDF etiquetas `<gazebo>` con fragmentos de SDF embebidos (por ejemplo, para indicar fricción o usar un plugin de Gazebo).
- Cuando se requiere modelar un entorno completo (terreno, iluminación, obstáculos estáticos), el URDF por sí mismo resulta insuficiente: el usuario debe combinarlo con un archivo SDF de mundo o incluir fragmentos `<gazebo>` que, a la larga, ensucian la descripción cinemática.
- No existe un mecanismo como los plugins SDF para definir transmisiones de motor, motores PID o sensores, pero puede recurrirse a extensiones en bloques `<gazebo>`.

Capítulo 3

Desarrollo del simulador

3.1. Entorno de trabajo y herramientas

En esta sección se describe el entorno de trabajo utilizado en el proyecto, así como las herramientas y conceptos básicos que se emplean. A lo largo de la memoria se profundizará en cada uno de ellos.

Se han utilizado los siguientes entornos para el desarrollo del proyecto:

- Sistema operativo Ubuntu 22.04.5 LTS.
- Gazebo Ignition Fortress para la simulación .
- El middleware robótico empleado es ROS2, versión Humble.
- Para el modelado 3D se emplea SolidWorks.
- Para la descripción del robot en Gazebo, URDF (Unified Robot Description Format).
- Para la descripción del entorno en Gazebo, SDF (Simulation Description Format).
- Visual Studio Code como entorno de programación.
- Git y GitHub, para el control de versiones del código y documentación del proyecto.

3.1.1. ROS2 Humble

ROS 2, desarrollado por Open Robotics, actúa como plataforma unificada para el software y el hardware del robot, funcionando a modo de sistema operativo. Como evolución de ROS 1, adopta DDS (Data Distribution Service) como capa de transporte y un modelo de comunicación distribuida basado en topics (publicador-suscriptor) y servicios (cliente-servidor), lo que garantiza una entrega de mensajes fiable, configuraciones de calidad de servicio y cifrado integrado. Cada funcionalidad (navegación, mapeo, localización, simulación, etc.) se implementa como un paquete reutilizable, lo que agiliza tanto el desarrollo como el despliegue de soluciones en plataformas robóticas variadas.

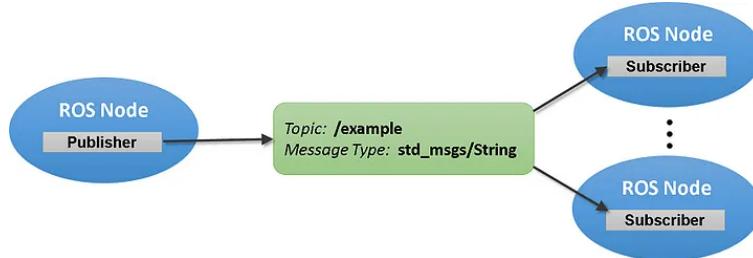


Figura 1. Esquema funcionamiento ROS

Para este proyecto se utiliza la distribución LTS ROS 2 Humble Hawksbill, lanzada en mayo de 2022 y con soporte garantizado hasta mayo de 2027. Humble Hawksbill está optimizada para ejecutarse de forma nativa en Ubuntu 22.04 Jammy Jellyfish, el sistema operativo utilizado, lo que asegura máxima compatibilidad, estabilidad y acceso a las últimas mejoras del ecosistema ROS 2.



Figura 2. ROS 2 Humble

3.1.2. Gazebo

Desarrollado por Open Robotics, es el simulador robótico más utilizado y extendido. Al pertenecer a la misma organización responsable de ROS, la integración nativa con este framework lo convierte en la mejor opción para el desarrollo y prueba de sistemas robóticos.

Gazebo ha evolucionado a lo largo de tres grandes familias de versiones.

La primera, Gazebo Classic, nació en 2002 y abarca las versiones numeradas del 1 al 11; incorporó de serie el motor ODE, añadió más tarde Bullet y DART, y estableció la integración nativa con ROS 1 y ROS 2.

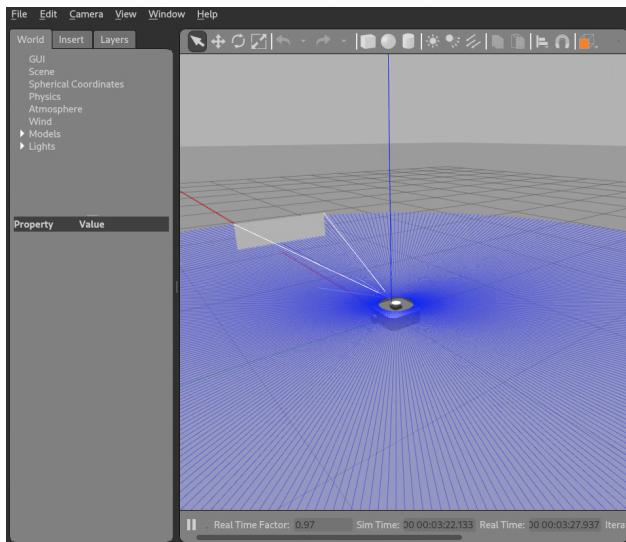


Figura 3. Gazebo Classic

En 2019 surgió Ignition Gazebo, que desmontó el monolito de Classic en componentes independientes de física, renderizado y sensores para mejorar la escalabilidad y el mantenimiento; sus principales lanzamientos son Acropolis, Blueprint, Citadel, Dome, Edifice y Fortress.

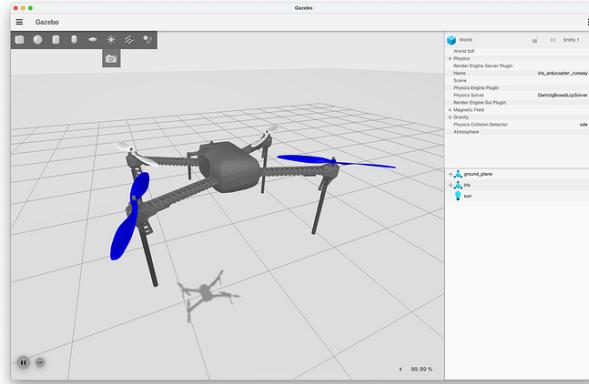


Figura 4. Ignition Gazebo

Finalmente, en 2023, Ignition Gazebo se rebautizó como Gazebo Sim, conservando la arquitectura modular de Ignition, adoptando un versionado semántico y garantizando la compatibilidad binaria durante todo su ciclo de vida; entre sus versiones más relevantes figuran Garden y Harmonic.

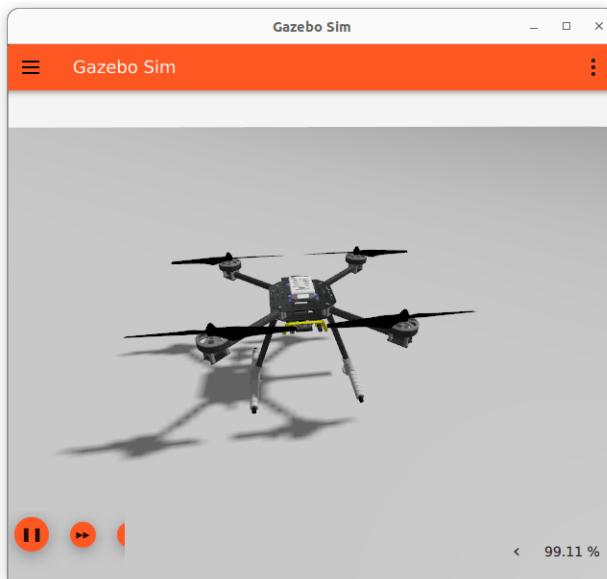


Figura 5. Gazebo Sim

Esta trayectoria muestra cómo Gazebo ha pasado de un simulador monolítico a una colección de bibliotecas independientes y, por último, a una plataforma unificada, sencilla de instalar y mantener.

Versión Gazebo en el proyecto

La versión de Gazebo utilizada es Gazebo Fortress [2], lanzado en julio de 2023, e inicia la línea Ignition Gazebo con una arquitectura completamente modular. Cada elemento de simulación: sensores, motores de física, renderizado, funciona como un componente independiente que puede desplegarse y actualizarse sin interferir en el resto, lo que acelera el desarrollo y permite escalar simulaciones complejas.

Para la física, ofrece múltiples motores (Bullet, DART, Simbody, TPE) que pueden seleccionarse en tiempo de ejecución según las necesidades de precisión o velocidad.



Figura 6. Gazebo Ignition Fortress

3.1.3. Compatibilidad ROS2 y Gazebo

Para garantizar un entorno de desarrollo robusto y coherente, es esencial escoger un middleware y un simulador que ofrezcan integración nativa, instalación sencilla y soporte a largo plazo. Una conexión fluida entre ROS 2 y Gazebo facilita el intercambio de datos entre nodos de software y componentes de simulación, acelera la configuración de escenarios virtuales y reduce el riesgo de incompatibilidades durante el desarrollo. Además, al basarse en distribuciones con soporte a largo plazo, se asegura estabilidad en el ciclo de vida del proyecto y simplifica la gestión de actualizaciones y dependencias.

	GZ Fortress (LTS)	GZ Harmonic (LTS)	Gz Ionic
ROS 2 Rolling	✗	⚡	✓
ROS 2 Kilted	✗	⚡	✓
ROS 2 Jazzy (LTS)	✗	✓	✗
ROS 2 Humble (LTS)	✓	⚡	✗
ROS 1 Noetic (LTS)	⚡	✗	✗

- ✓ - Recommended combination
- ✗ - Incompatible / not possible.
- ⚡ - Possible, *but use with caution*. These combinations of ROS and Gazebo can be made to work together, but some effort is required.

Figura 7. Tabla compatibilidad ROS-Gazebo [3]

Tanto Humble como Fortress son distribuciones LTS (Long Term Support), con soporte garantizado hasta mayo de 2027 para ROS 2 Humble y hasta septiembre de 2026 para Gazebo Fortress. La documentación oficial recomienda emparejar siempre las versiones LTS de ROS y Gazebo para asegurar estabilidad a largo plazo en desarrollo y despliegue.

3.1.4. SolidWorks

Es el software de diseño asistido por computadora (CAD) 3D utilizado para la creación del modelo de Panter. Este modelo ha sido utilizado para modificar los elementos del robot y finalmente, obtener las mallas (meshes) que permiten la visualización de Panter en Gazebo.



Figura 8. Logo SolidWorks

3.1.5. Visual Studio Code

Visual Studio es un editor de código fuente. Se trata de un entorno de desarrollo ligero y compatible con múltiples sistemas operativos, cuyo potencial se amplía mediante extensiones. Soporta una gran variedad de lenguajes.

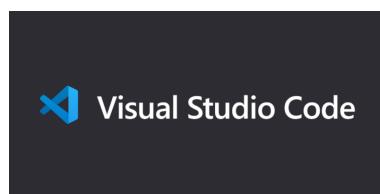


Figura 9. Logo Visual Studio Code

3.1.6. Git y GitHub

Git es un sistema de control de versiones distribuido que registra el historial de un proyecto, permite trabajar de forma paralela mediante ramas y fusiona cambios sin comprometer la estabilidad del código.



Figura 10. Logo Git

GitHub es la plataforma en línea que aloja repositorios Git. Esta colaboración promueve un flujo de trabajo estructurado y colaborativo en proyectos.

A screenshot of a GitHub repository page for 'panter_ws'. The top navigation bar shows 'Code' is selected. The main content area displays a list of commits from user 'jorge' with messages like 'Reorden de los launcher' and dates like '2 weeks ago'. On the right side, there are sections for 'About' (no description), 'Releases' (no releases), 'Packages' (no packages), and 'Languages' (not applicable). The bottom of the page shows a 'Simulador Panter en Gazebo' section.

Figura 11. Repositorio proyecto simulador en GitHub [4]

3.1.7. Términos

A continuación se explican brevemente algunos de los términos empleados durante el proyecto:

- **Formato XML** (eXtensible Markup Language): es un lenguaje de marcado que emplea etiquetas personalizadas para representar datos jerárquicos de forma legible por humanos y máquinas. En robótica se utiliza para estructurar la descripción de componentes (enlaces, articulaciones, inercia, colisiones, visuales) de manera modular y extensible.

- **Python:** es el lenguaje preferido en ROS 2 para automatizar la generación y manipulación de archivos de descripción (URDF/XACRO), lanzar nodos y gestionar configuraciones. Los launch files de ROS 2 suelen escribirse en Python, facilitando la inclusión de parámetros y la composición de nodos en un solo fichero.
- **URDF:** es el formato de archivo XML que se utiliza para describir la geometría y la estructura de un robot en ROS. Son los ficheros con extensión .urdf.
- **SDF:** es un estándar XML diseñado para describir mundos y modelos en simulación de robots, especialmente con Gazebo.
- Archivos con extensión **.STL**: fichero de malla 3D en formato STL (STereoLithography), ampliamente usado para representar la geometría superficial de piezas CAD.
- Archivos con extensión **.yaml**: son ficheros de texto estructurado, legibles por humanos, muy usados en robótica y en ROS2 para representar datos de configuración y parámetros, utilizados para la configuración de software y aplicaciones.
- Archivos con extensión **.rviz**: son los archivos de configuración de la herramienta RViz. Permite cargar una configuración de interfaz ya desarrollada.
- Archivo con extensión **.world**: archivos que representan el entorno modificable en Gazebo utilizando lenguaje XML.
- Ficheros con extensión **.hpp**: son los archivos C++ de cabezera, donde se declaran las interfaces del código. Aquí se definen las clases, métodos públicos y privados, atributos y constructores, sin incluir la implementación de cada función.
- Ficheros con extensión **.cpp**: Archivos C++ que contienen la implementación del código. Define los métodos y funciones, configura publicadores, suscriptores, timers y callbacks. Aquí se encuentra toda la lógica de ejecución del nodo.
- Archivo con extensión **.launch.py**: scripts en Python que utilizan la biblioteca de lanzamiento de ROS 2 para orquestar el arranque de la aplicación. Cada archivo define una 'LaunchDescription' en la que se declaran nodos, procesos, parámetros y argumentos

de entrada, y se pueden incluir otros archivos de lanzamiento. Mediante estos scripts se inicializan de forma modular y reproducible herramientas como Gazebo, ROS 2 y RViz 2, asegurando un despliegue ordenado de todos los nodos desarrollados, tal como se describe en esta memoria.

- Archivo con extensión **.xacro**: es un lenguaje macro XML. Permite estructurar el código en archivos XML más cortos y legibles utilizando macros. Son utilizados para dividir los modelos URDF del robot.
- **Terminal**: también conocida como consola de texto. Está incorporada en el sistema operativo, permite comunicarnos a través del teclado. Será utilizada como interfaz de la línea de comandos y del texto para el usuario. Para abrir una terminal de texto en Ubuntu: ctrl + alt + t.

3.2. Modelo SolidWorks

El proyecto parte de un modelado en SolidWorks del vehículo Panter, al que hay que realizarle una serie de modificaciones y correcciones, para que se adecúe a cómo será el robot Panter real cuando finalice su desarrollo, adaptado para que la simulación sea fluida.

Aunque el modelo proporcionado por el departamento de Ingeniería de Sistemas y Automática incluye prácticamente todas las piezas del robot, algunos componentes resultan innecesarios en simulación al no aportar valor físico. Al eliminarlos, aligeramos la carga computacional y mejoramos notablemente el rendimiento.

El modelo de Panter utilizado para la simulación está compuesto por los siguientes elementos:

- **P-CH**: chasis del robot.
- **P-SD01**: amortiguadores delanteros (derecho e izquierdo).
- **P-ST01**: amortiguadores traseros (derecho e izquierdo).
- **P-SD02**: brazos superiores de suspensión delantera (derecho e izquierdo).

- **P-ST02:** brazos superiores de suspensión trasera (derecho e izquierdo).
- **P-SD03:** brazos inferiores de suspensión delantera (derecho e izquierdo).
- **P-ST03:** brazos inferiores de suspensión trasera (derecho e izquierdo).
- **P-ED:** ruedas delanteras (derecha e izquierda).
- **P-ET:** ruedas traseras (derecha e izquierda).

Una vez que el número de elementos de robot ha sido aligerado, necesitamos recortar la longitud del robot en 40 cm, con el objetivo de que tenga las mismas proporciones que el robot real.

Para ello, el método por el que se ha realizado es extruir 40 cm todas las barras horizontales de la base de panter, y posteriormente, desplazar, utilizando relaciones de posición, todos los elementos a su posición correcta.

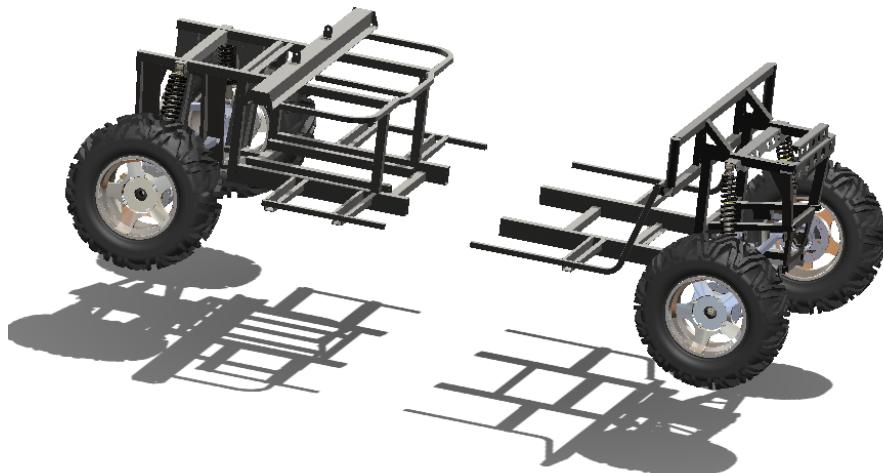


Figura 12. Panter tras extrusión



Figura 13. Panter reconstruido

3.2.1. Exportar de SolidWorks a URDF

Como se comentó en la introducción, URDF (Unified Robot Description Format) es el estándar XML que ROS utiliza para describir la geometría y la estructura de un robot.

Tras modelar el robot en SolidWorks, instalé el complemento 'plugin' SolidWorks to URDF Exporter [5], que aprovecha el árbol jerárquico del ensamblaje para:

- Definir semi-automáticamente los links y joints.
- Generar los archivos .stl de cada componente.
- Exportar la descripción al formato URDF.

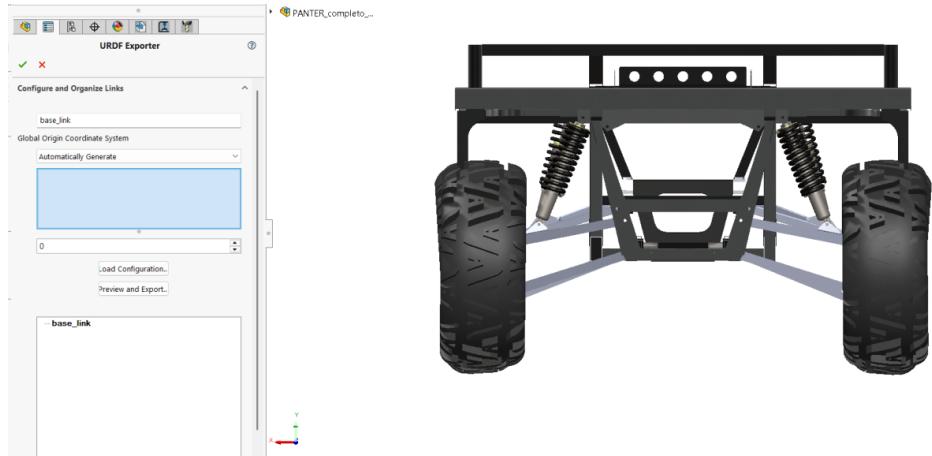


Figura 14. Interfaz herramienta SW2URDF

Sin embargo, al hacer pruebas con la exportación semi-automática, surgieron varios inconvenientes, por lo que llevé a cabo estas correcciones:

- Ejes de rotación mal definidos: el plugin no alineaba correctamente los ejes de los joints. Para solventarlo, creé los sistemas de coordenadas locales en cada pieza de SolidWorks, orientando los ejes según el movimiento deseado.
- Sistema de coordenadas global: al intentar fijarlo manualmente, el 'plugin' mostraba errores. Finalmente dejé que el 'plugin' lo generara de forma automática.
- Parámetros físicos y visuales incorrectos: valores de masa, inercia, tipo de 'joint' y colores no eran adecuados. Los recalcule y ajusté externamente antes de incorporarlos al URDF.

A pesar de estas dificultades, el 'plugin' ha sido de gran utilidad porque:

1. Proporciona una base URDF inicial, reduciendo el trabajo manual.
2. Ofrece los modelos por separado en formato .stl.
3. Facilita la configuración de posiciones relativas, sistemas de coordenadas y joints en el ensamblaje.

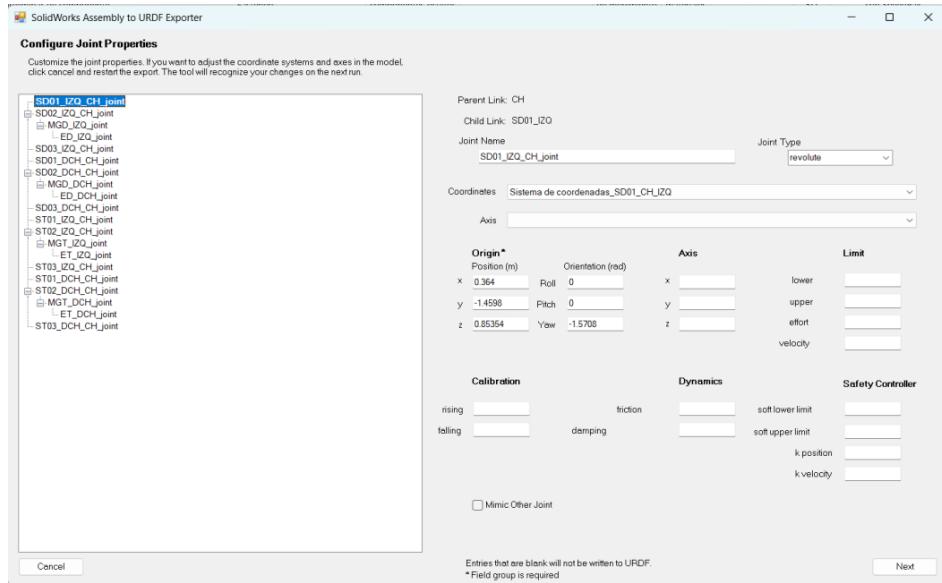


Figura 15. Configuración árbol de entidades, sw2urdf exporter [6]

3.3. Jerarquía del proyecto

El proyecto se apoya en la plantilla oficial de ROS/Gazebo [7] presentada en la conferencia 'ROSCon' [8].

El espacio de trabajo de este proyecto, denominado panter_ws, se organiza en tres paquetes con funciones claras y diferenciadas:

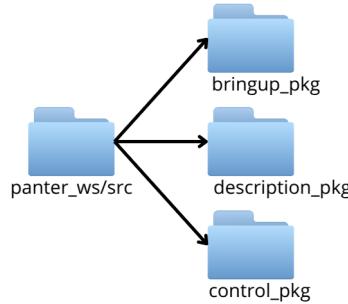


Figura 16. Paquetes proyecto

3.3.1. control_pkg

El paquete control_pkg agrupa toda la lógica de conducción de Panter en ROS 2. Su estructura contiene:

- **config/**: contiene el archivo .yaml de configuración de los controladores del frame_work 'ros2_control', explicado en el Apartado ??.
- **include/**: archivos de cabecera que definen las interfaces de los controladores.
- **launch/**: scripts en Python que automatizan la puesta en marcha de los nodos y la configuración inicial de ROS 2 al iniciar el sistema de control.
- **src/**: implementaciones en C++ de los algoritmos de control por velocidad y de torque de las ruedas.

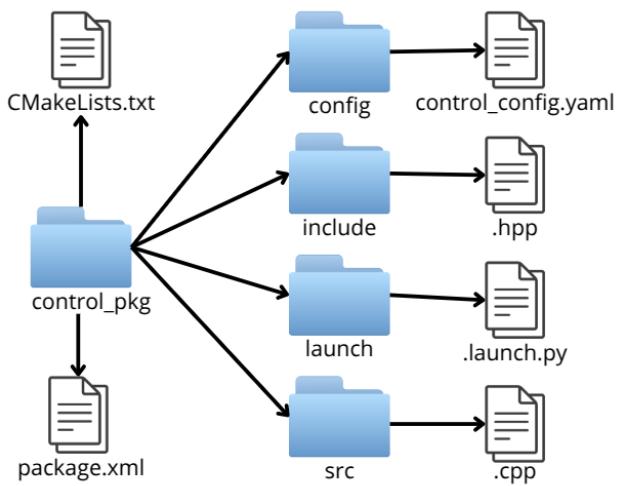


Figura 17. Estructura paquete control_pkg

3.3.2. description_pkg

Estructurado en subdirectorios para separar cada tipo de recurso:

- **config/**: contiene el archivo ros_gz_bridge.yaml, que define el mapeo de topics entre Gazebo Ignition y ROS 2.
- **launch/**: incluye el script sim_urdf.launch.py, responsable de iniciar Gazebo con el modelo URDF y cargar parámetros adicionales de simulación.
- **meshes/**: contiene los modelos CAD exportados en formato .stl para cada pieza de Panter (chasis, ruedas, suspensiones, etc.).

- **rviz/**: el fichero 'panter.rviz' establece la configuración de RViz 2 necesaria para visualizar el robot y su estado en tiempo real.
- **urdf/**: contiene los archivos .urdf que describen la geometría, la masa y la cinemática del vehículo.
- **worlds/**: escenarios de Gazebo diseñados para probar Panter en diferentes entornos y condiciones de terreno.
- **models/**: almacena los archivos COLLADA (.dae) que definen la geometría y las texturas del escenario de simulación personalizado, permitiendo recrear fielmente el entorno de pruebas.

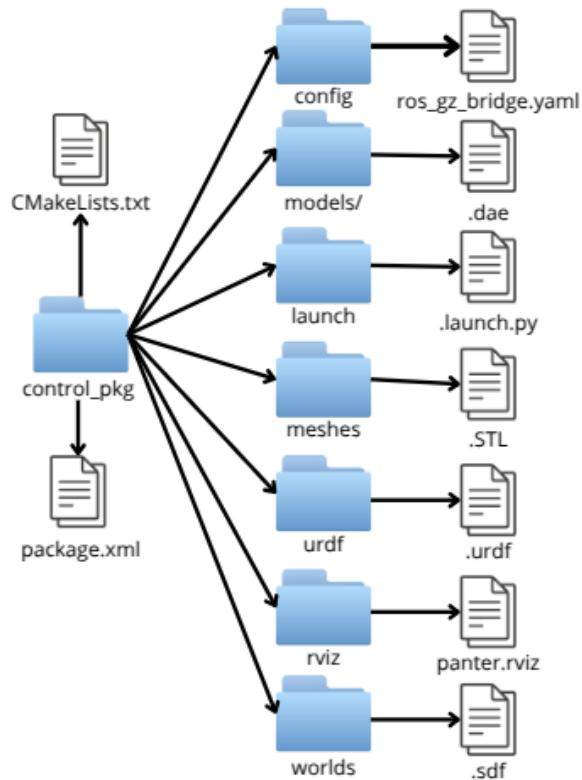


Figura 18. Estructura paquete description_pkg

3.3.3. bringup_pkg

Incluye dos lanzadores principales:

- 'proyectoVel.launch.py', para el control por velocidad.
- 'proyectoTorque.launch.py', para el control por torque.

Ambos scripts de lanzamiento inician de forma secuencial todos los launchers y nodos de '**description_pkg**' y '**control_pkg**', proporcionando un punto de entrada único y coherente para la simulación completa.

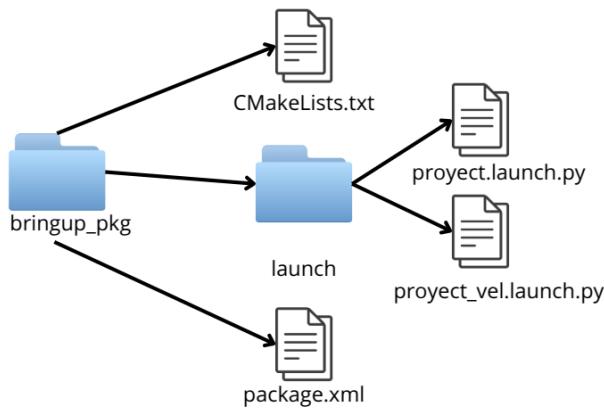


Figura 19. Estructura paquete bringup_pkg

Cada paquete incluye sus propios archivos CMakeLists.txt y package.xml, cuya configuración y contenidos se detallan en los Apartados 3.6 y 3.7.

3.4. Descripción Modelo Robot Panter

Para la simulación del vehículo móvil Panter se definen dos variantes de la descripción URDF, adaptadas a distintos modos de control: panter.urdf y panter_vel.urdf.

3.4.1. Estructura

Ambos archivos, 'panter.urdf' y 'panter_vel.urdf', comparten la definición de los mismos componentes del robot, pero difieren en los plugins y bloques adicionales que especifican su modo de desplazamiento.

- **Links:** cada parte rígida del robot (chasis, amortiguadores, brazos, manguetas, ruedas) se describe con un bloque <link> que incluye otros bloques:

- **Inercial**: con la masa, centro de masa y el tensor de inercia para esa pieza.
- **Visual**: referencia al archivo mesh (.stl) que se utilizará para el renderizado, así como el material o color aplicados.
- **Collision**: describe la geometría que Gazebo emplea para calcular las colisiones de ese link con el entorno y con otros objetos.
- **Joints**: cada articulación se define con un bloque <joint>, indicando:
 - Tipo de articulación que se establece entre dos links.
 - **Parent link**: el enlace rígido “padre”, que actúa como soporte fijo para el movimiento.
 - **Child link**: el enlace rígido “hijo”, que se desplaza (rota o traslada) en relación con el parent.
 - Límites de effort, velocidad y posición, que definen hasta dónde y con qué fuerza/- velocidad puede moverse esa articulación.
 - El eje sobre el que se produce el giro o la traslación.
 - La posición del joint en el espacio, es decir, las coordenadas relativas a alguno de sus links.

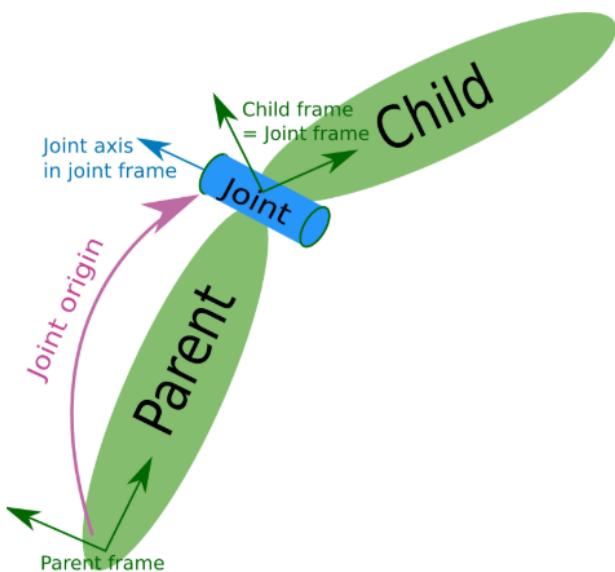


Figura 20. Parent y child en un joint

- Se ha añadido un sensor de esfuerzo y par en la articulación motriz de cada rueda, empleando un plugin nativo de Gazebo. Este plugin se configura dentro del URDF mediante una etiqueta <gazebo>, lo que permite capturar, durante la simulación, los valores de fuerza y torque aplicados a las articulaciones.
- A través de etiquetas <gazebo> en el URDF, se han definido los parámetros de fricción de las ruedas de Panter, incluyendo los coeficientes de fricción estática y dinámica. De este modo, el modelo reproduce comportamientos más realistas al interactuar con el terreno.

La Figura 21 ilustra el modelo URDF de Panter en el simulador Gazebo, con la visualización de los ejes de sus articulaciones activa.

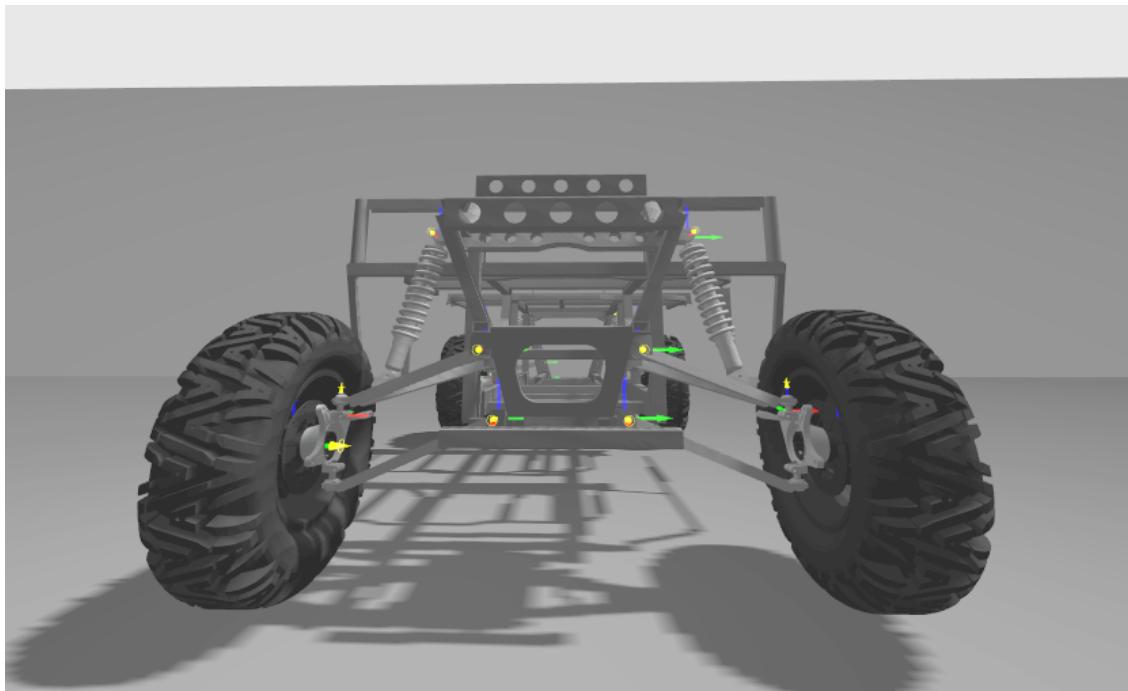


Figura 21. Descripción Panter, con eje de los Joints activos

3.4.2. Características para el control

Luego, cada descripción se ha desarrollado por la necesidad de incluir una forma distinta de control sobre Panter:

panter.urdf

En el archivo 'panter.urdf', el movimiento del robot se controla enviando comandos de torque a las ruedas a través de ROS 2. Para ello se utiliza el framework **ros2_control**, un plugin de ROS 2 que permite regular actuadores en modo esfuerzo (torque), posición o velocidad; este framework se detalla en el Apartado 3.12.3.

Con ros2_control he configurado dos controladores principales:

- **JointEffortController** para aplicar torque directamente a las ruedas motrices, garantizando un control fino del par en cada articulación.
- **JointPositionController** (interfaz de posición) para determinar el ángulo de las ruedas delanteras, implementando la cinemática de Ackermann y asegurando giros precisos y realistas.

panter_vel.urdf

En la descripción panter_vel.urdf, el movimiento del robot se controla enviando consignas de velocidad lineal y angular a través de ROS 2 por el topic 'cmd_vel' con mensajes de tipo 'geometry_msgs/Twist.msg' [9]. Para implementar este control se utiliza el plugin de direccionamiento Ackermann de Gazebo, que se encarga de traducir esas consignas en el movimiento del vehículo. Este plugin se describe en detalle en el apartado 3.12.1.

3.5. Programación en ROS

Para el control de Panter en ROS se han desarrollado dos nodos teleoperados por teclado, que difieren en la forma de generar el movimiento:

- **controlVelocidad**: captura comandos de velocidad lineal y angular desde el teclado y los publica en el topic '/cmd_vel' para desplazar el robot.
- **keyboardcontrolTorque**: traduce las pulsaciones de teclado en comandos de esfuerzo (torque) para cada rueda, permitiendo un control directo del par aplicado.

3.5.1. Estructura

En C++, y especialmente en un proyecto ROS 2, es buena práctica separar la interfaz (cabezera) de la implementación (cuerpo):

3.5.1.1. Cabezera (.hpp)

Se encargan de definir la interfaz de nuestras clases y funciones, sin incluir su lógica interna.

En ellos se encuentra:

- Definición de clases: atributos (públicos y privados) y sus métodos.
- Declaración de constantes, tipos, clases/funciones, etc.
- Inclusiones de otros headers necesarios para referenciar tipos (por ejemplo, incluir <rclcpp.hpp> para nodos de ROS 2).

Cualquier otro módulo que necesite crear o manipular un objeto declarado en esa cabecera, sólo necesita incluir el .hpp para conocer su API pública.

3.5.1.2. Cuerpo (.cpp)

Este fichero implementa íntegramente el nodo de teleoperación de Panter. Su funcionamiento se compone de bloques bien diferenciados:

Configuración inicial en el constructor

Al instanciar el nodo, se crean los publicadores para actuar sobre el movimiento, y suscripciones a los sensores de fuerza/torque de cada rueda, de modo que cada lectura entrante se almacena en variables internas. Finalmente, se programa un temporizador que cada tres segundos informa en consola de los valores actuales de fuerza y torque en todas las ruedas.

Callbacks de sensores

Cada vez que llega un mensaje de sensor, el callback correspondiente copia sus componentes de fuerza y torque en la estructura de datos asociada.

Bucle de teleoperación por teclado

El nodo arranca un hilo dedicado exclusivamente a leer pulsaciones desde la terminal en modo “raw”, lo que evita tener que pulsar “Enter” tras cada tecla. Dentro de este bucle, cada carácter presionado se interpreta según un conjunto predefinido de controles: avanzar, retroceder, girar en ambas direcciones y detenerse, recogidas en la Tabla 1. Para cada acción, el bucle prepara los datos correspondientes de torque o velocidad según el tipo de control, y los envía inmediatamente a los publicadores correspondientes. Pulsar la barra espaciadora restaura la terminal a su modo original sin interrumpir el resto de la aplicación.

Tecla	Acción
w	Avanzar recto
d	Avanzar derecha
a	Avanzar izquierda
s	Retroceder recto
c	Retroceder derecha
z	Retroceder izquierda
x	Detener el robot

Tabla 1. Comandos del teclado para controlar el robot

Función principal ‘main’

La función principal arranca ROS 2 y crea la instancia del nodo. A continuación, lanza el lector de teclado en un hilo apartado para que pueda ejecutarse simultáneamente con la gestión de mensajes y temporizadores de ROS 2. El hilo principal se bloquea en la atención de callbacks hasta que se solicita el fin de la teleoperación. Tras ello, se espera a que el hilo de teclado finalice, se apaga ROS 2 y el programa devuelve éxito al sistema operativo.

1. Se inicializa con la función **‘init()’**: inicializa las estructuras internas de ROS 2, procesa argumentos de la línea de comandos y deja todo listo para crear nodos, publishers, subscribers y servicios.
2. Crear instancia del nodo con la función **‘make_shared’**: construye un objeto de la clase definida en el nodo y devuelve un shared_ptr al mismo.

3. Lanza el hilo para la lectura de teclado con la función '**keyboard_thread**': crea un nuevo hilo (thread) separado del hilo principal que llamará a la función asignada, usando el objeto apuntado por node. Es decir, en paralelo con el resto de la lógica de ROS 2, este hilo ejecutará el bucle de lectura de teclado.
4. **rclcpp::spin(node)** es una llamada bloqueante que se mantiene escuchando callbacks, al ser bloqueante, si no utilizase otro hilo para la lectura por teclado, se bloquearía el ciclo de ROS 2 y no recibirías callbacks.
5. La función **keyboard_thread.join()**: bloquea el hilo principal hasta que el keyboard_thread termine su ejecución.
6. La función '**shutdown**': libera los recursos internos de ROS 2, detiene la comunicación, y marca a todos los publishers/subscribers/servicios como inactivos. A partir de este punto, el nodo ya no podrá publicar ni suscribirse.
7. Con '**return 0**' termina el programa devolviendo 0 al sistema operativo, indicando que todo fue correcto.

Por lo tanto, el nodo es el objeto que se crea en tiempo de ejecución (a través de `std::make_shared()`). El '`.cpp`' es la estructura que indica a qué lógica concreta corresponde ese nodo y contiene el '`main`' para arrancarlo.

3.5.2. Control por velocidad

- Se define una clase **ControlVelocidad** que hereda de **rclcpp::Node** con nombre “**controlVelocidad**”.
- En el constructor es donde se crean los publishers, los subscribers y cualquier otra inicialización que se defina:
 - Se crea un *publisher* de tipo `geometry_msgs::msg::Twist` en el tópico `/cmd_vel`.
 - Se crean cuatro *subscriptions* a mensajes `geometry_msgs::msg::Wrench`, una para cada sensor de fuerza/torque ubicado en las cuatro ruedas del robot:
 - Sensor rueda delantera izquierda (ED_IZQ).

- Sensor rueda delantera derecha (ED_DCH).
- Sensor rueda trasera izquierda (ETIZQ).
- Sensor rueda trasera derecha (ETDCH).
- Cada callback de los sensores almacena internamente en una variable de tipo `geometry_msgs::msg::Wrench` la última lectura de fuerza y torque recibida.
- Se crea un 'timer' que, cada 3 segundos, imprime por consola las cuatro lecturas actuales (fuerza y torque de cada rueda).
- En el método `manual_drive_panter()`:
 - Se cambia la terminal a modo "raw" para leer una tecla en bruto.
 - Se lee un carácter de la entrada estándar.
 - Según el carácter leído, se construye un mensaje `Twist` ajustando `linear.x` y `angular.z`, y se publica inmediatamente en `/cmd_vel`.
 - Si se detecta la tecla espacio, se restaura la terminal al modo normal y se abandona la lectura en crudo.
 - Cualquier otra pulsación no válida se ignora mostrando un mensaje informativo.
- En la función `main`:
 - Se inicializa ROS 2 y se crea la instancia de `ControlVelocidad`.
 - Se lanza en un hilo separado el método `manual_drive_panter()`, para no bloquear el bucle principal de ROS.
 - Se llama a `rclcpp::spin(node)` para procesar las *subscriptions* y el *timer* de forma continua.
 - Al finalizar, se espera a que el hilo de teclado termine y se apaga ROS 2.

3.5.3. Control por esfuerzo

- Se define una clase `KeyboardcontrolTorque` que hereda de `rclcpp::Node` con nombre "keyboardcontrolTorque".

- En el constructor:
 - Se crea un *publisher* de tipo `std_msgs::msg::Float64MultiArray` en el tópico `/effort_controller/commands` para enviar torques a las cuatro ruedas.
 - Se crea un *publisher* de tipo `std_msgs::msg::Float64MultiArray` en el tópico `/steering_controller/commands` para enviar los ángulos de giro de las ruedas delanteras.
 - Se crean cuatro *subscriptions* a mensajes `geometry_msgs::msg::Wrench`, una para cada sensor de fuerza/torque en las cuatro ruedas, y cada callback almacena la última lectura en una variable interna.
 - Se crea un *timer* que, cada 3 segundos, imprime por consola las lecturas actuales de fuerza y torque de las cuatro ruedas.
- En el método `keyboard_loop()`:
 - Se cambia la terminal a modo “raw” y se entra en un bucle que se mantiene mientras ROS 2 está activo.
 - Se lee un carácter de la entrada estándar.
 - Según el carácter leído:
 - Se forma un mensaje `Float64MultiArray` con los valores de torque a aplicar en las cuatro ruedas.
 - Se forma un mensaje `Float64MultiArray` con los ángulos de giro para las dos ruedas delanteras.
 - Ambos mensajes se publican inmediatamente en sus respectivos tópicos (`/effort_controller/commands` y `/steering_controller/commands`).
 - Si se detecta la tecla espacio, se restaura la terminal al modo normal y se sale del bucle.
 - Cualquier pulsación no válida se ignora mostrando un mensaje informativo.
- En la función `main`:

- Se inicializa ROS 2 y se crea la instancia de `KeyboardcontrolTorque`.
- Se lanza en un hilo separado el método `keyboard_loop()`, de modo que la lectura de teclado y la publicación de torques/giro no bloqueen el procesamiento de sensores.
- Se llama a `rclcpp::spin(node)` para procesar continuamente las *subscriptions* y el *timer*.
- Al finalizar, se espera a que el hilo de teclado termine y se apaga ROS 2.

3.6. CMakeLists.txt

Para garantizar una configuración de compilación sólida y escalable, cada paquete de `panter_ws` sigue la siguiente estructura en su `CMakeLists.txt`:

1. Declaración inicial:

Se define la versión mínima de CMake y se nombra el paquete, indicando que se programará en C++.

```
cmake_minimum_required(VERSION 3.8)
project(control_pkg)
```

2. Localización de dependencias:

Con `find_package()` se cargan las librerías esenciales de ROS 2, de modo que los directorios y las bibliotecas estén disponibles durante la compilación.

```
find_package(ament_cmake REQUIRED)
find_package(rclcpp REQUIRED)
find_package(std_msgs REQUIRED)
```

3. Definición de nodos:

Cada nodo de control se compila como un ejecutable independiente y se vincula con sus dependencias.

```
add_executable(controlVelocidad src/controlVelocidad.cpp)
ament_target_dependencies(controlVelocidad rclcpp std_msgs geometry_msgs)
```

4. Instalación y exportación:

Determinan qué elementos se instalarán en el espacio de destino y generan la información para que otros paquetes los localicen e importen.

```
install(TARGETS
        controlVelocidad
        DESTINATION lib/${PROJECT_NAME}
)
ament_package()
```

3.7. package.xml

El archivo `package.xml` define la configuración y las dependencias de cada paquete en ROS 2:

- **Metadatos del paquete:** incluye nombre, versión, descripción, correo del responsable y licencia.
- **Herramienta de compilación:** especifica `ament_cmake` como herramienta de compilación.
- **Dependencias de ejecución:** enumera los paquetes necesarios en tiempo de ejecución (por ejemplo, `rclcpp` y `geometry_msgs`).

3.8. Lanzadores (.launch.py)

Los ficheros de lanzamiento en ROS 2 son scripts en Python que, mediante la librería 'launch', arrancan de forma coordinada varios nodos, procesos y configuraciones. En este proyecto se definen tres lanzadores principales:

3.8.1. description_pkg/sim_urdf.launch.py

Inicia la simulación de Gazebo con el modelo URDF de Panter, publica sus transformadas y habilita el puente con ROS 2. En el caso del control por velocidad, el lanzador es **simVel.launch.py**.

Estructura

1. Declaración del archivo urdf y archivo sdf que pertenece al entorno de simulación (world): **urdf_file** y **world_file**.
2. Nodo **robot_state_publisher** : genera el parámetro **/robot_description**, que es el parámetro de ROS donde se almacena la descripción completa del robot en formato URDF.
3. Nodo **launch_gazebo**: ejecuta el simulador Gazebo Fortress, incluyendo el ‘world’ desarrollado.
4. Nodo **spawn_panter**: se encarga de que el modelo Panter aparezca en el simulador en unas coordenadas establecidas.

3.8.2. control_pkg/ros.launch.py

Inicia el nodo de control teleoperado por torque y carga sus parámetros de funcionamiento. En el caso del control por velocidad, el lanzador es **ros_vel.launch.py**.

Estructura

1. Declaración del paquete control_pkg, y de los archivos de configuración de los controladores y del archivo urdf: **control_pkg**, **controller_config** y **urdf_file**, respectivamente.
2. Nodo **control_node**, se encarga de iniciar el controlador ros2_control.
3. Nodo **joint_state_publisher**: lee la descripción de las articulaciones en el URDF y publica de forma periódica mensajes sensor_msgs/JointState con la posición de cada articulación, aportando los datos necesarios para que **robot_state_publisher** genere los frames TF correspondientes y para que RViz2 represente en tiempo real el movimiento articulado del robot según el estado de sus articulaciones [10].

4. Nodo **effort_spawner**: arranca el controlador de torque de las articulaciones para gestionar el giro de las ruedas.
5. Nodo **steering_spawner**: pone en marcha el controlador de posición responsable de orientar las ruedas delanteras según la cinemática Ackermann.
6. Nodo **ros_control**: se encarga de ejecutar el control de Panter a través de teclado, desarrollado en ROS2.

3.8.3. bringup_pkg/proyect.launch.py

Ofrece un punto único de entrada que incluya los launchers anteriores, simplificando el lanzamiento completo. En el caso del control por velocidad, el lanzador completo es **project_vel.launch.py**.

Estructura

1. Nodo **gazebo_launch**: ejecuta el lanzador **sim_urdf.launch.py**.
2. Nodo **control_launch**: ejecuta el lanzador **ros.launch.py**.
3. Nodo **bridge_node**: inicia el fichero con el puente de comunicación entre ROS2 y Gazebo, **ros_gz_bridge.yaml**.
4. Nodo de lanzamiento de **RViz2**, programa de ROS para la visualización del robot.

3.9. RViz

RViz2 (ROS Visualization 2) es la herramienta de visualización en tiempo real de ROS 2. Permite representar de manera interactiva y configurable los datos que fluyen por los distintos topics de un robot: como estados de articulaciones y modelos URDF, mapas de navegación, trayectorias planificadas y sensores LIDAR o cámaras.

He creado un archivo de configuración personalizado para RViz2 (panter.rviz) que, utilizando el topic **robot_description**, carga automáticamente el modelo de Panter y ajusta todos los parámetros de visualización necesarios.

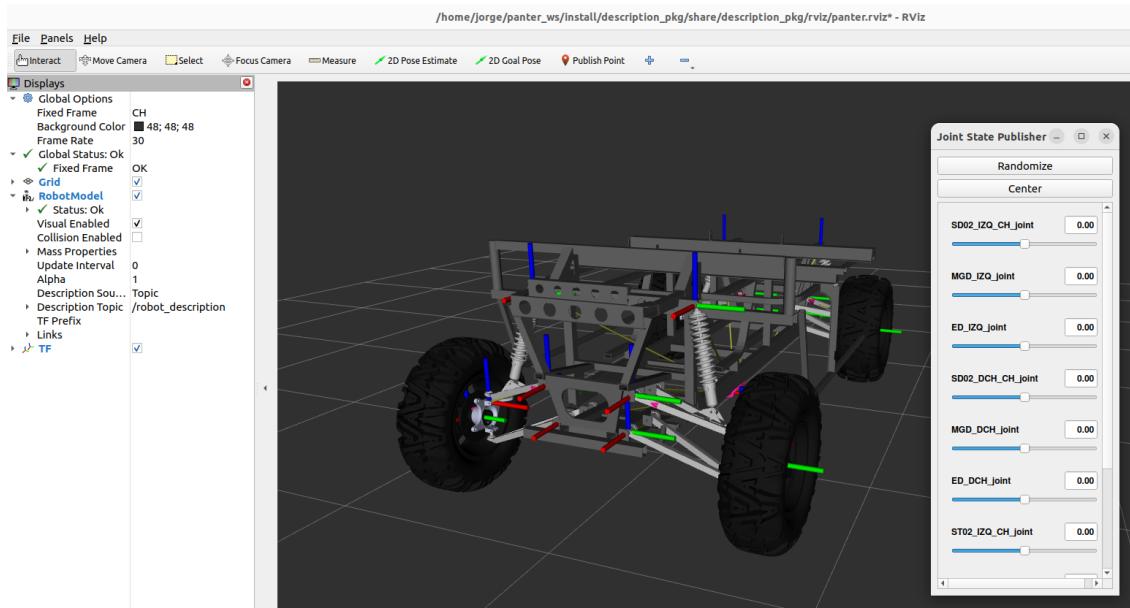


Figura 22. Panter en RViz

Mediante el nodo 'joint_state_publisher', es posible manipular en tiempo real la posición de cada articulación mediante controles deslizantes. Esta funcionalidad interactiva fue clave al desarrollar el URDF de Panter, ya que facilitó la comprobación visual de que todos los links y joints estaban correctamente definidos y conectados.

3.10. Ficheros de configuración

3.10.1. ros_gz_bridge.yaml

En un entorno de simulación con ROS 2 e Ignition Gazebo, el archivo ros_gz_bridge.yaml actúa como un “puente de configuración” que le indica al nodo encargado del bridge qué canales de comunicación debe enlazar y cómo debe traducir los mensajes entre ROS y Gazebo.

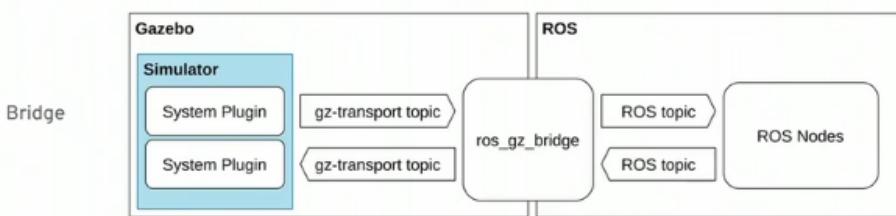


Figura 23. Flujo ros_gz_bridge

Para cada una de estas líneas de configuración se indica:

- El nombre del 'topic' en ROS y su nombre equivalente en Gazebo.
- El tipo de mensaje que usa ROS y el tipo paralelo en Ignition (por ejemplo, geometry_msgs/msg/Twist ↔ ignition.msgs.Twist [11]).
- La dirección del flujo de datos:
 - ROS→Gazebo para comandos y actuadores.
 - Gazebo→ROS para sensores y estados de simulación.
 - Bidireccional, cuando queremos mantener ambos ROS y Gazebo sincronizados en los dos sentidos.

```
- ros_topic_name: "/cmd_vel"
  gz_topic_name: "/cmd_vel"
  ros_type_name: "geometry_msgs/msg/Twist"
  gz_type_name: "ignition.msgs.Twist"
  direction: ROS_TO_GZ
```

Figura 24. Estructura de ros_gz_bridge.yaml [12]

3.10.2. control_config.yaml

El archivo control_config.yaml es la configuración de los controladores en ROS 2 Control: define qué controladores cargar, a qué ritmo se actualizan y sobre qué articulaciones actúan.

En el archivo control_config.yaml se han declarado los siguientes controladores:

- **effort_controller:**

- Tipo: effort_controllers/JointGroupEffortController
 - Función: regula el torque aplicado a las articulaciones encargadas de impulsar las cuatro ruedas (EDIZQ_joint, ED_DCH_joint, ETIZQ_joint y ET_DCH_joint), garantizando la fuerza correcta para el movimiento del vehículo.

- **steering_controller:**

- Tipo: position_controllers/JointGroupPositionController

- Función: controla la posición angular de las articulaciones de las manguetas delanteras (MGD_IQZ_joint y MGD_DCH_joint), definiendo así la orientación de las ruedas y, por tanto, la dirección del vehículo.

3.11. Físicas

3.11.1. Motor de físicas Gazebo

En Gazebo, la simulación de la dinámica del robot (colisiones, fricción, gravedad y restricciones) se lleva a cabo por un “motor de físicas”. Aunque la interfaz de usuario y los formatos de mundo (SDF) son comunes, internamente Gazebo puede emplear diferentes bibliotecas de simulación.

El motor de físicas utilizado en el proyecto es XXXXXX.

3.11.2. Colisiones

La gestión de colisiones en Gazebo se basa en distinguir dos tipos de geometrías para cada enlace del robot: la visual (lo que se ve en pantalla) y la collision (la forma simplificada que usa el motor de físicas para detectar contactos). Esta separación permite optimizar el rendimiento, ya que las mallas de colisión pueden ser prismas, cilindros o cajas mucho más ligeras computacionalmente que los modelos visuales detallados.

Cuando dos geometrías de colisión se solapan, el motor de físicas genera un contacto que incluye un punto de aplicación, la superficie de contacto, para evitar que los objetos penetren uno en otro y para modelar el deslizamiento o bloqueo.

Para definir las colisiones he utilizado figuras geométricas básicas, que hacen más simple y ligera la simulación a nivel computacional.

Las ruedas han sido descritas con cilindros, mientras que el chasis con un prisma rectangular.

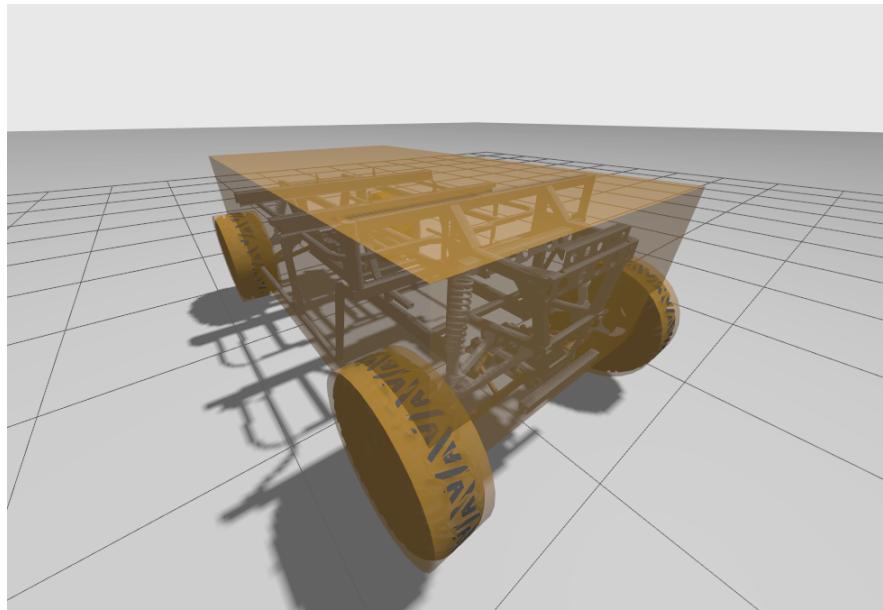


Figura 25. Representación colisiones de Panter en Gazebo

3.11.3. Masas

Conocer la masa de cada elemento del vehículo permite realizar cálculos precisos para las físicas del robot en el simulador.

Para obtener dichas masas, ha sido necesario desmontar del vehículo real para realizar un pesaje de las piezas de forma aislada.



Figura 26. Desmontaje para pesaje



Figura 27. Pesaje rueda

En la Tabla 5 se recogen las masas medidas de todos los componentes principales, tanto del eje delantero como del trasero.

Elemento	Masa (kg)
<i>Eje delantero</i>	
Amortiguador Delantero (SD01)	3.75
Brazo Superior Delantero (SD02)	2.75
Brazo Inferior Delantero (SD03)	2.55
Mangueta Delantera	2.95
Palier Delantero	0.90
Disco Frenos Delantero	3.90
Rueda Delantera (ED)	15.95
Rueda Trasera Completa (ET)	20,75
<i>Eje trasero</i>	
Amortiguador Trasero (ST01)	5.35
Brazo Superior Trasero (ST02)	2.65
Brazo Inferior Trasero (ST03)	2.70
Mangueta Trasera	2.20
Palier Trasero	4.95
Disco Frenos Trasero	4.35
Rueda Trasera	19.95
Rueda Trasera Completa (ET)	25,2

Tabla 2. Masas de los componentes de Panter.

En la descripción URDF, las ruedas (tanto delanteras como traseras) se modelan como conjuntos completos que incluyen la llanta, el disco de freno y el palier correspondiente. En particular, a la rueda trasera se le ha asignado la masa del palier delantero. Son denominadas en la Tabla 5 como 'rueda completa'.

Para determinar la masa del chasis, se ha pesado el vehículo apoyado sobre ambos ejes (delantero y trasero), excluyendo todos los componentes montados sobre el chasis (baterías, controladoras, reductoras planetarias, rodamientos, cadenas y motores PID), cuyos resultados se recogen en la Tabla 3.

	Izquierda (kg)	Derecha (kg)
Delantero	100,55	80,15
Trasero	80,95	81,70
TOTAL		343,35

Tabla 3. Distribución de masas por rueda (en kg).

Del valor total recogido en la Tabla 3 es preciso descontar la masa correspondiente a las ruedas y a los conjuntos de suspensión.

$$M_{\text{SoloChasis}} = 343,35 \text{ kg} - M_{\text{Ruedas}} - M_{\text{Suspensión}} = 201,65 \text{ Kg} \quad (1)$$

A la masa calculada en la Ecuación 1 debe sumársele el peso de los componentes instalados sobre el chasis.

Elemento	Masa unitaria (kg)	Unidades	Masa total (kg)
Baterías	45	4	180
Controladoras	4,65	4	18,6
Reducadoras planetarias	35	4	140
Motores PID	20,15	4	80,6
Margen (rodamientos, cadenas, ...)	20	4	80
TOTAL			499,2

Tabla 4. Masas de los elementos asociados al chasis de Panter.

$$M_{\text{ChasisCompleto}} = M_{\text{SoloChasis}} + M_{\text{ElementosAdicionales}} = 700,85Kg \quad (2)$$

Masas en la descripción URDF

La masas utilizadas en los elementos de la descripción se encuentran en la Tabla 5.

Elemento	Masa (kg)
Chasis (CH)	700,85
<i>Eje delantero</i>	
Amortiguador Delantero (SD01)	3,75
Brazo Superior Delantero (SD02)	2,75
Brazo Inferior Delantero (SD03)	2,55
Mangueta Delantera (MDG)	2,95
Rueda Delantera Completa (ED)	20,75
<i>Eje trasero</i>	
Amortiguador Trasero (ST01)	5,35
Brazo Superior Trasero (ST02)	2,65
Brazo Inferior Trasero (ST03)	2,70
Mangueta Trasera (MGT)	2,20
Rueda Trasera Completa (ET)	25,20

Tabla 5. Masas de los componentes de Panter.

3.11.4. Inercias

La matriz de inercia representa la distribución de la masa alrededor de los ejes principales de un cuerpo rígido y mide su resistencia a los cambios en la velocidad de giro. Calcularla para cada componente del robot es importante para reproducir un comportamiento dinámico fiel y mantener un control preciso.

Si estos valores se infraestiman o sobreestiman, el modelo dinámico no representa con exactitud la respuesta ante fuerzas y momentos, lo que puede resultar en movimientos erráticos e inestabilidad. Para aligerar el proceso de cálculo y evitar la complejidad de tratar geometrías

complejas, cada pieza del robot (chasis, brazos de suspensión, amortiguador, ruedas, etc.) se representa mediante formas geométricas elementales (cilindros y prismas). Gracias a ello, podemos aplicar directamente fórmulas analíticas para obtener de forma rápida y rigurosa los tres momentos de inercia principales (I_x, I_y, I_z) de cada elemento.

Matriz inercia

El uso del prisma rectangular y del cilindro como aproximaciones geométricas básicas facilita el cálculo de la matriz de inercia [13] para los distintos componentes de Panter. Gracias a esta simplificación, cualquier variación en las dimensiones de las piezas puede incorporarse de forma rápida y directa, permitiendo recalcular la matriz de inercia con mínimo esfuerzo.

Prisma rectangular:

$$\begin{aligned}I_x &= \frac{1}{12}m(b^2 + h^2), \\I_y &= \frac{1}{12}m(l^2 + h^2), \\I_z &= \frac{1}{12}m(b^2 + l^2).\end{aligned}$$

Para el chasis y los brazos de suspensión se ha adoptado la aproximación mediante prismas rectangulares.

Cilindro:

$$\begin{aligned}I_x &= \frac{1}{12}m(3r^2 + h^2), \\I_y &= \frac{1}{2}mr^2, \\I_z &= \frac{1}{12}m(3r^2 + h^2).\end{aligned}$$

Para las ruedas y para los amortiguadores se ha adoptado la aproximación mediante cilindros.

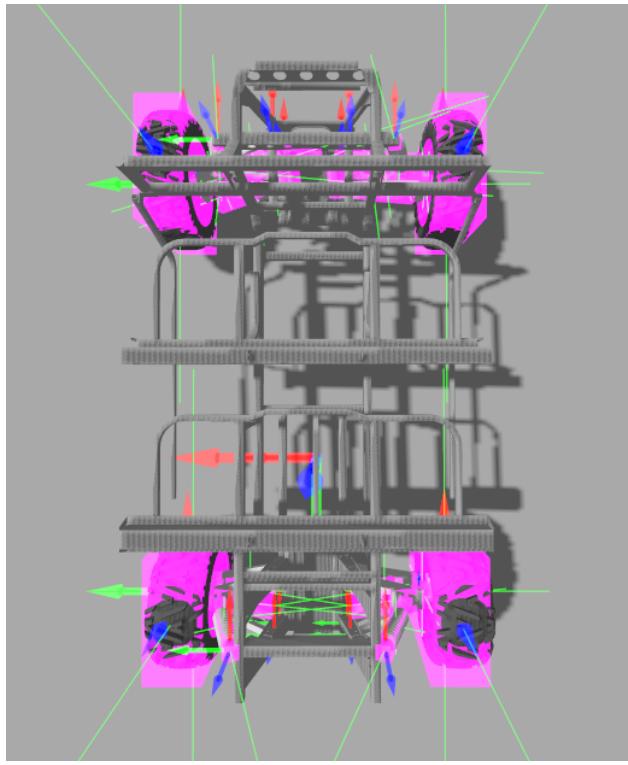


Figura 28. Representación inercias Panter

3.11.4.1. Ajuste inercia Chasis

Para determinar el centro de masas (CoM) del vehículo a partir de las lecturas de peso en cada rueda, se requieren dos elementos fundamentales:

1. Distribución de cargas (lecturas de peso en cada rueda):

Delantero-Izquierdo (DI) = 100,55kg,

Delantero-Derecho (DD) = 80,15kg,

Trasero-Izquierdo (TI) = 80,95kg,

Trasero-Derecho (TD) = 81,70kg,

$$\text{Total} = 100,55 + 80,15 + 80,95 + 81,70 = 343,35\text{kg}.$$

2. Cotas geométricas del vehículo:

- $L = 2.20486\text{m}$: distancia entre el eje delantero y el eje trasero (“wheelbase”).

- $T = 1.36356m$: ancho entre ruedas izquierdas y derechas (“track width”).

Definición de la Convención de Ejes

El sistema de referencia es el mostrado en la Figura 29.

$X > 0$; hacia la izquierda en planta (o a la derecha de frente).

$Y > 0$; hacia atrás (hacia el eje trasero).

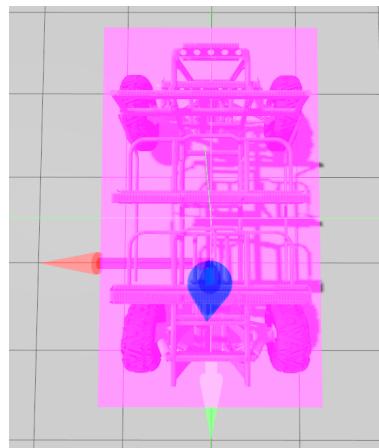


Figura 29. Sistema de referencia Panter

Cálculo del Centro de Masas

El centro de masas $(X_{\text{CoM}}, Y_{\text{CoM}})$ se descompone en dos coordenadas:

X_{CoM} = desplazamiento lateral (izquierda/derecha).

Y_{CoM} = desplazamiento longitudinal (adelante/atrás).

Coordenada longitudinal Y_{CoM}

Para la coordenada longitudinal, tomamos $Y = 0$ en el eje delantero y “hacia atrás” como $Y > 0$. Aplicando la regla de momentos:

$$Y_{\text{CoM}} = \frac{W_{\text{front}} \cdot 0 + W_{\text{rear}} \cdot L}{M_{\text{tot}}} = \frac{162,65 \times 2,20486}{343,35} \approx 1,0449 \text{ m.}$$

$$Y_{\text{CoM}} = +1,0449 \text{ m.}$$

Es decir, el CoM se encuentra 1,0449 m detrás del eje delantero.

Coordenada lateral X_{CoM}

Definimos la línea central como $X = 0$. Las ruedas izquierdas están en $X = +\frac{T}{2}$, y las derechas en $X = -\frac{T}{2}$.

La fórmula del momento lateral es:

$$X_{\text{CoM}} = \frac{W_{\text{left}}\left(+\frac{T}{2}\right) + W_{\text{right}}\left(-\frac{T}{2}\right)}{M_{\text{tot}}} = \frac{(181,50) \frac{1,36356}{2} - (161,85) \frac{1,36356}{2}}{343,35} \approx 0,0390 \text{ m.}$$

$$X_{\text{CoM}} = +0,0390 \text{ m.}$$

Resultado final

Por tanto, en el sistema donde $X > 0$ apunta a la izquierda y $Y > 0$ apunta hacia atrás:

$$(X_{\text{CoM}}, Y_{\text{CoM}}) = (+0,0390 \text{ m}, +1,0449 \text{ m}).$$

- $X_{\text{CoM}} = +0,0390 \text{ m}$: *0,039m hacia la izquierda* (en planta) de la línea central.
- $Y_{\text{CoM}} = +1,0449 \text{ m}$: *1,0449m hacia atrás* (desde el eje delantero).

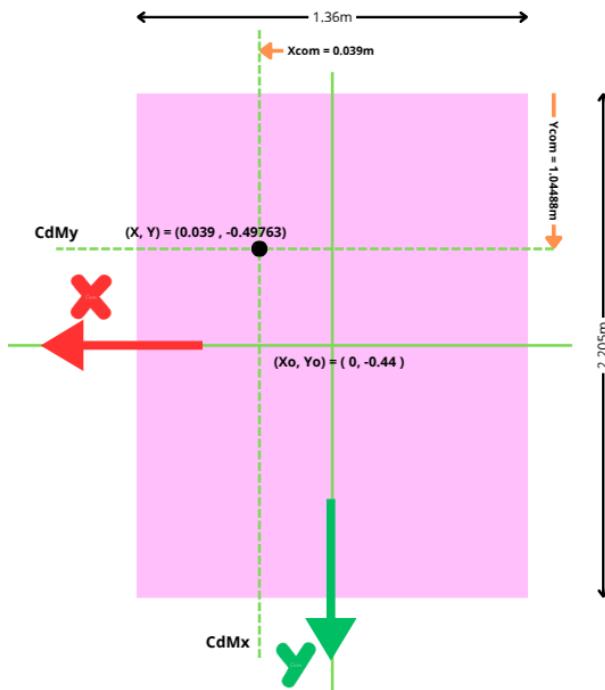


Figura 30. Esquema de ajuste inercia Chasis

Finalmente, en la Figura 31 se observa cómo es la inercia en Panter.

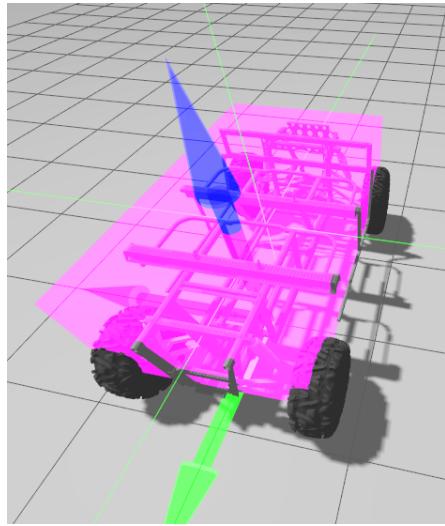


Figura 31. Inercia Chasis ajustada

3.11.5. Rozamiento

En el simulador, el rozamiento entre las ruedas y el terreno se modela a través de los parámetros de “surface” y “friction” que se declaran tanto en las geometrías de colisión de las ruedas como en el plano que representa el suelo. Dado que nuestro escenario es un terreno no estructurado (tierra, campo con cierta rugosidad), he elegido coeficientes de fricción relativamente altos para las ruedas, de modo que puedan transmitir el par de tracción sin deslizarse excesivamente.

En la descripción URDF, ‘m’ y ‘m2’ son los coeficientes de fricción estática y dinámica, respectivamente. Un valor de 1.5 significa que la máxima fuerza tangencial antes de que la rueda empiece a deslizarse es 1.5 veces la normal.

Dirección Ruedas

En Panter, las ruedas delanteras giran de forma completamente independiente y no están conectadas por ningún mecanismo de suspensión o eje común. Cuando una rueda pisa un terreno más elevado o hundido que la otra, cada una adopta un ángulo distinto, lo que puede desestabilizar la trayectoria, como se ilustra en la Figura 32.

Para evitar este descontrol, hemos añadido a las articulaciones de las manguetas delanteras

dos parámetros que introducen resistencia al giro ante perturbaciones:



Figura 32. Enter Caption

Para resolver esta situación es necesario incluir una resistencia al giro físico de las ruedas debido a perturbaciones externas. Para ello he incluido dos parámetros en la definición de las articulaciones de las manguetas delanteras:

- Amortiguamiento (damping): aplica un par proporcional a la velocidad de giro de la rueda, reduciendo oscilaciones y frenando movimientos bruscos.
- Fricción (friction): define un coeficiente de resistencia estática y dinámica, de modo que las ruedas requieren un par mínimo para iniciar el giro y mantienen posición cuando reciben pequeñas fuerzas externas.

Con estos ajustes, las ruedas delanteras responden de manera más estable ante desniveles, mejorando la dirección incluso en terrenos irregulares.

3.11.6. Suspensiones

En primer lugar, no ha sido posible modelar la física en Gazebo del conjunto de suspensión. A continuación, se va a desarrollar la problemática e intentos realizados.

Para el modelado de las suspensiones debemos primero conocer cómo funciona el conjunto de suspensión de Panter.

El sistema de suspensión está compuesto por dos brazos de suspensión, que se encuentran unidos al chasis, y estos a la mancueta, que a su vez está unida a la rueda. Luego, el amortiguador se encuentra entre el chasis y el brazo de suspensión superior.

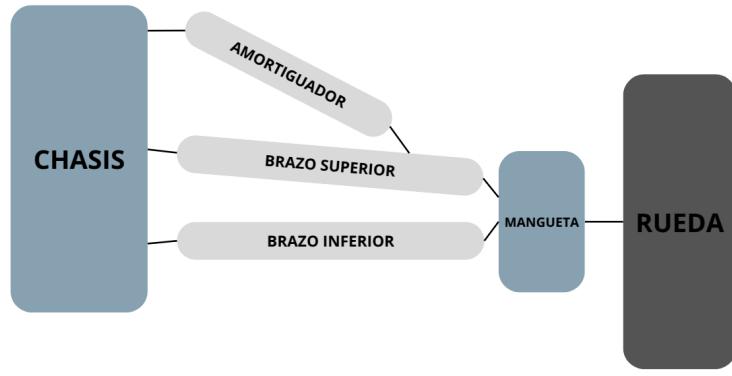


Figura 33. Esquema conjunto de suspensión

Para poder modelar la suspensión en el simulador gazebo, debemos introducir cómo se deben comportar las articulaciones que unen esos elementos en la descripción URDF. La unión de dos links (eslabones) se realiza con una articulación (joint).

Al declarar una articulación, debemos establecer qué link es el parent (parent), y qué link es el hijo (child) del movimiento, es decir, el que realizará el movimiento alrededor del parent.

Entonces, por la propia cinemática que tiene Gazebo, se establece una serie de normas:

- Un link puede ser parent de varios links child.
- Un link solo puede ser child de un solo parent.
- Un link puede ser a la vez parent y child.



Figura 34. Conjunto de suspensión

A partir de estas normas, he tratado de buscar la forma de resolver esta situación.

Planteamiento 1

El primer intento realizado para modelar el conjunto de suspensión es establecer joints que unan todos los elementos, como en la figura XXX, donde la 'P' representa a parent, y la 'C' representa a child. Las circunferencias rojas representan los child que están generando conflicto, ya que tienen varios parents.

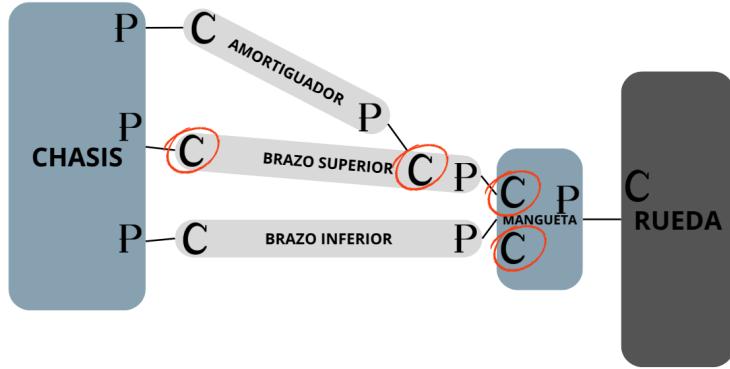


Figura 35. Planteamiento 1 joints

El problema surge en que Gazebo no permite introducir bucles cerrados de cinemática, y en el diseño del conjunto de suspensión, aparecen dos bucles cerrados.

No es posible resolver el problema de esta forma.

Planteamiento 2

El segundo intento realizado es establecer links falsos, con masa e inercia nula, situados entre los joints que entran en conflicto. Esta idea no ha resuelto el problema.

Planteamiento 3

Utilizar articulaciones fijas (fixed joints) en dos puntos de los bucles, para que rompa con el bucle de cinemática. Esta es la idea que propone el tutorial de gazebo, pero llevada a la práctica, la respuesta del sistema sigue siendo que hay links que tienen más de un parent.

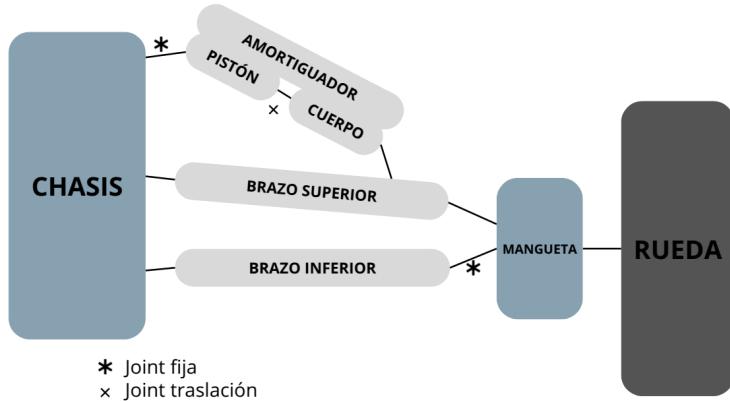


Figura 36. Planteamiento 3 joints

Planteamiento 4

Otro planteamiento para resolver este problema es el uso de mimic joints, cuya cualidad es la de hacer que una articulación realice el mismo movimiento que la otra articulación existente. Podríamos aplicarlo a los dos brazos de suspensión, en el que el inferior esté conectado al chasis y a la mancueta, y el superior no esté conectado a la mancueta e imite el movimiento del inferior. Pero surgen los siguientes problemas:

- La gravedad actúa sobre el vehículo al comenzar la simulación, por lo que el brazo superior cae.
- Los brazos de suspensión no son paralelos, por lo que su movimiento ante el trabajo de suspensión no será igual para ellos.
- Al solo unir la mancueta por un punto, esta no se mantendrá en su orientación correcta al encontrar cualquier perturbación, sino que perderá su verticalidad y por tanto también la de la rueda.

Conclusión conjunto de suspensión

No es posible describir el conjunto de suspensión de Panter, ya que contiene bucles cerrados de cinemática, que no son soportados por el simulador Gazebo.

Possible solución: simplificar el conjunto de suspensión, dejando únicamente un solo brazo de suspensión y eliminando el amortiguador, y a su vez mantener las características de amortiguación que ofrecen estos elementos.

3.11.7. Reductora

La reductora planetaria (o caja de engranajes planetarios) es un elemento mecánico que permite transformar la velocidad y el par de giro de un motor en valores más adecuados para impulsar las ruedas.

Panter utiliza el reductor planetario APEX AE205 con relación de transmisión 15:1 y juego angular menor a 12 arc-min ($J < 12$).

Para el caso en el que el simulador de Panter es controlado por 'Effort' esfuerzo, es necesario incluir el valor de la reductora, para que la simulación sea fiel a la realidad; para ello debe declararse en la descripción URDF.

En la descripción URDF utilizo el bloque `<transmission>`, que le dice a ROS 2 Control cómo enlazar un “actuador” (motor) físico con la articulación del robot, aplicando una relación de reducción mecánica. Por lo tanto, permite declarar el valor de reducción 15:1.

3.12. Plugins

3.12.1. Plugin Direccionamiento de Ackermann

El plugin AckermannSteering de Ignition Gazebo [14] implementa un controlador de dirección tipo Ackermann, pensado para vehículos que cuentan con ruedas delanteras que giran (steering) y ruedas motrices traseras, como es el caso del robot Panter. Es necesario que el modelo tenga al menos un par de ruedas izquierdas y derechas, así como un par de articulaciones (joints) que controlen la dirección de las ruedas delanteras. A continuación se detallan las características principales del plugin:

- **Bloque `<gazebo>`** En la descripción del modelo, se incluye un bloque `<gazebo>` que permite instanciar el plugin asociado al sistema de simulación.
- Definición del plugin:
 - `filename="libignition-gazebo-ackermann-steering-system.so"`: indica la librería que contiene la implementación del controlador Ackermann.

- `<name=“ignition::gazebo::systems::AckermannSteering”`: identifica el sistema dentro de Ignition Gazebo.
- `<left_steering_joint>MGD_IQZ_joint</left_steering_joint>` Articulación que controla la dirección de la rueda delantera izquierda (pivot de giro). El plugin usará esta referencia para aplicar el ángulo apropiado al girar.
- `<right_steering_joint>MGD_DCH_joint</right_steering_joint>`: articulación que controla la dirección de la rueda delantera derecha. Ambas juntas de dirección (`left_steering_joint` y `right_steering_joint`) definen la geometría del tren delantero.
- `<left_joint>ED_IQZ_joint</left_joint> <right_joint>ED_DCH_joint</right_joint>` Identifican las juntas motrices correspondientes a la rueda trasera izquierda y trasera derecha, respectivamente. El plugin las usa para aplicar el torque necesario que genere la velocidad deseada.
- `<left_joint>ET_IQZ_joint</left_joint> <right_joint>ET_DCH_joint</right_joint>` Se repiten los tags de las articulaciones para incluir las dos ruedas motrices adicionales (en este caso, trasera izquierda y trasera derecha). De este modo, el controlador sabe que hay dos pares de ruedas motrices (ED_IQZ/ED_DCH y ET_IQZ/ET_DCH) a las que debe distribuir el torque.
- `<wheel_separation>1.41</wheel_separation>` Distancia (en metros) entre las ruedas izquierda y derecha de cada eje motriz. En este modelo equivale a 1.41 m. Este valor es esencial para calcular correctamente los ángulos Ackermann de cada rueda al girar.
- `<wheel_radius>0.343</wheel_radius>` Radio (en metros) de las ruedas motrices. Con 0.343 m, el plugin convierte la velocidad lineal deseada (v en m/s) en la velocidad angular de las juntas motrices ($\omega = v/0.343$).
- `<min_velocity>-15.0</min_velocity> <max_velocity>15.0</max_velocity>` Rangos de velocidad lineal permisible (en m/s) para el vehículo. Aquí se establece que puede ir desde -15 m/s (marcha atrás) hasta +15 m/s (marcha adelante). El plugin limitará la consigna de velocidad dentro de este intervalo.

- <min_acceleration>-0.5</min_acceleration> <max_acceleration>10</max_acceleration>
Límites de aceleración (en m/s²) que el sistema puede aplicar.
 - <min_acceleration>-0.5</min_acceleration>: desaceleración máxima (frenado) de -0.5 m/s².
 - <max_acceleration>10</max_acceleration>: aceleración máxima de +10 m/s².

Estos valores ayudan a suavizar la transición de velocidad y evitan aceleraciones bruscas que podrían desestabilizar la simulación.

- <topic>cmd_vel</topic> Topic ROS al que el plugin se suscribe para recibir comandos de velocidad (`geometry_msgs/Twist`). A partir de cada mensaje en `cmd_vel`, el sistema calcula los movimientos de las juntas de dirección y las velocidades angulares de las ruedas motrices.

3.12.2. Plugin sensor force-torque

El plugin GazeboRosFTSensor [15] es un plugin que simula un sensor de fuerza y par (force/torque) en Gazebo.

Este plugin se acopla a una articulación (joint) o a un enlace (link) de un modelo en Gazebo y calcula, en cada ciclo de simulación, la fuerza y el par que actúan en dicha junta. A continuación, emite estos valores a través de un mensaje 'WrenchStamped' en un topic ROS. El sensor virtual publica el esfuerzo (fuerza y par) en el sistema de coordenadas del enlace hijo de la articulación, y toma la dirección de la medición desde el enlace hijo hacia el padre de la articulación.

3.12.3. Framework ros2_control

Ros2_control es un framework modular de ROS 2 que unifica el acceso al hardware y la ejecución de controladores, actuando como puente entre los algoritmos de control y los drivers físicos. En este proyecto he utilizado ros2_control para gestionar los actuadores, lo que ha posibilitado:

- Gestionar de forma unificada tanto los controladores de torque (esfuerzo) como los de velocidad para sus ruedas y manguetas, sin cambiar código ni la estructura de nodos.
- Definir en el URDF las transmisiones (<transmission>) y actuadores (hardware_interface) de cada joint (por ejemplo ED_IQZ_joint, MGD_IQZ_joint) de modo que el controller_manager reconozca automáticamente cada hardware interface.
- Carga los plugins de control (Position, Velocity, Effort) en tiempo de ejecución.
- Publicar y suscribirse vía ROS 2 a los topics estándar.

3.13. Escenario Simulación

Para conseguir una simulación creíble, es esencial recrear un entorno que refleje las condiciones exteriores reales en las que operará Panter. Dado que este robot está diseñado para moverse sobre superficies irregulares, he modelado un terreno no estructurado en Blender.

Comencé con un simple plano, lo subdividí en numerosas secciones y ajusté manualmente cada zona para generar una topografía variada; el resultado de esa malla inicial puede verse en la Figura 37.

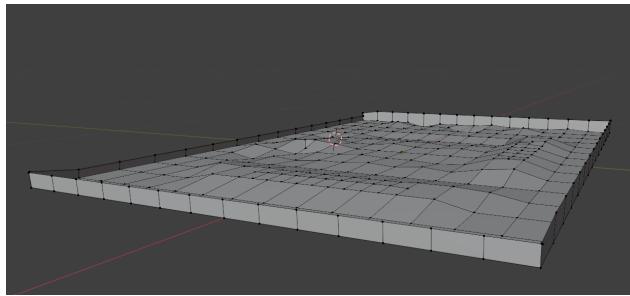


Figura 37. Malla escenario en Blender

A continuación, perfeccioné los relieves y ondulaciones hasta completar el terreno definitivo, que se muestra en la Figura 38.

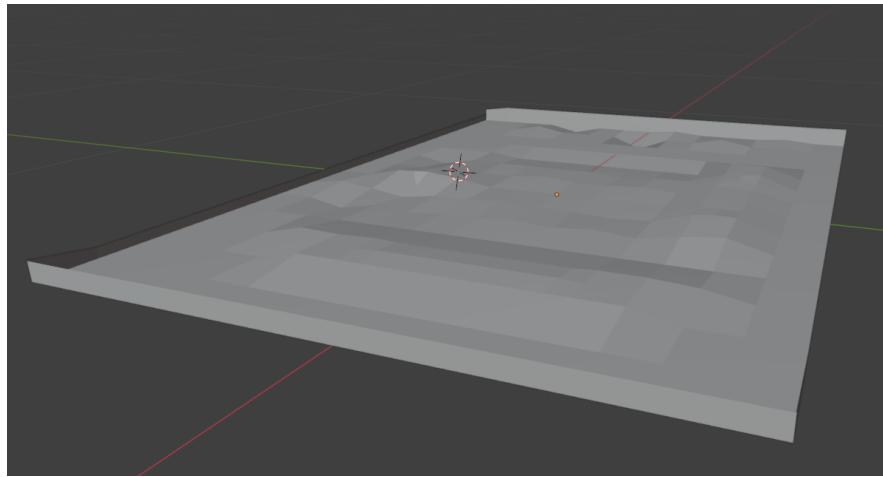


Figura 38. Escenario en Blender

A continuación, en el paquete `description_pkg` se crea la carpeta '`models/Tierra`' con la siguiente organización:

- `model.sdf`: contiene la descripción del escenario en formato SDF, exportada desde Blender y adaptada al mundo de Gazebo (ver Figura 38).
- `model.config`: define el nombre del modelo (“Tierra”), que es el identificador que usará Gazebo al instanciar el escenario desde el archivo `world`.
- `/meshes`: Guarda la malla Collada (.dae) del terreno irregular que generamos en Blender (ver Figura 37).
- `/materials`: incluye las texturas y archivos de material necesarios para dar realismo al terreno dentro del simulador.

A continuación, en el archivo `tierra_world.sdf` se incorpora el modelo del terreno para que quede integrado en la simulación. La Figura 39 muestra a Panter desplazándose sobre el entorno diseñado, mientras que la Figura 40 ofrece una vista global del escenario completo.

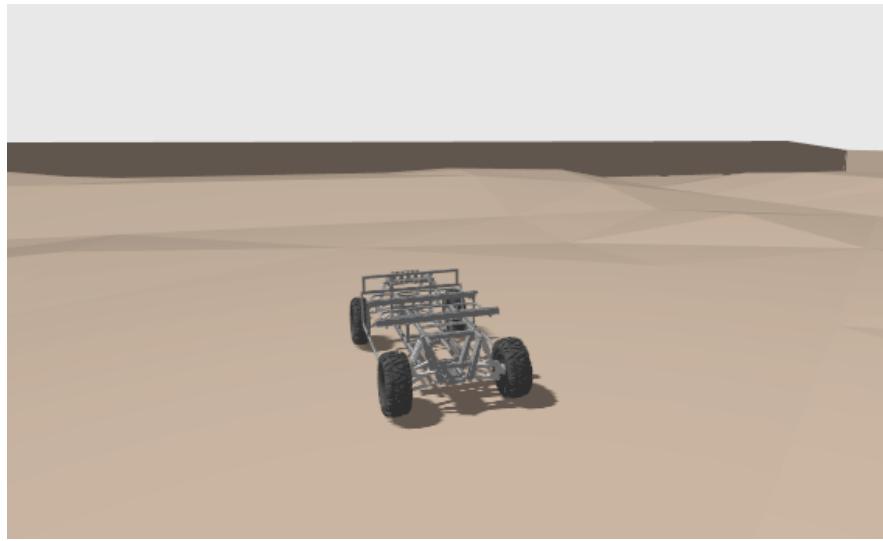


Figura 39. Enter Caption

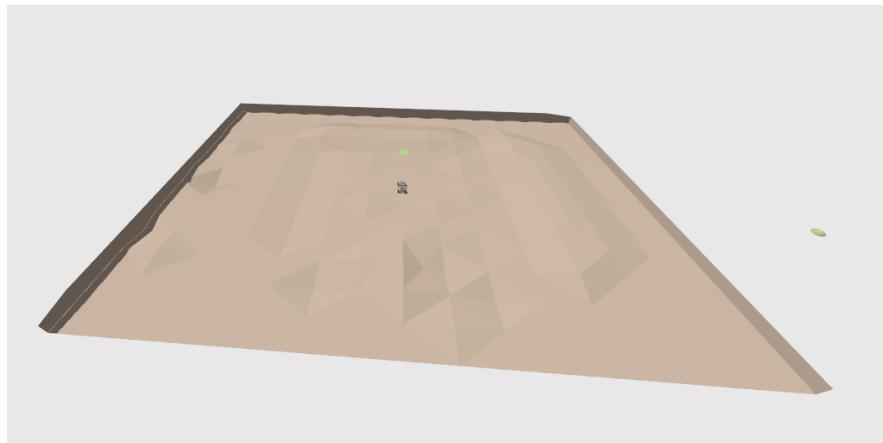


Figura 40. Enter Caption

Capítulo 4

Resultados

He evaluado el rendimiento de Panter en dos entornos de prueba (el escenario por defecto y el escenario personalizado) utilizando ambos modos de conducción: control por velocidad y control por torque.

La simulación se encuentra en un rendimiento gráfico del 75 % al 98 %, con el entorno modelado y Panter en movimiento, y tanto Gazebo como RViz2 en funcionamiento.

4.1. Escenario por defecto

Este entorno, completamente plano (ver Figura ??), carece de irregularidades que puedan alterar su trayectoria, lo que lo convierte en la opción ideal para validar que la lógica de conducción está correctamente implementada.

En este escenario, los sensores force-torque instalados en cada rueda funcionan correctamente, proporcionando datos precisos sobre las fuerzas y los torques que actúan en cada articulación.

Control por Velocidad

Aquí comprobamos que, al enviar consignas de velocidad lineal y angular, Panter sigue la trayectoria esperada sin desviaciones.

Control por Par

Empleando el controlador de esfuerzo, el robot aplica pares constantes en cada rueda y demuestra un seguimiento de la trayectoria igualmente preciso.

4.2. Escenario personalizado

En este caso, Panter se enfrenta a un terreno no estructurado con desniveles y obstáculos de pequeña escala (ver Figura ??). Las irregularidades del suelo introducen perturbaciones que nos permiten evaluar la robustez de los algoritmos de control.

En este escenario, los sensores de fuerza y par no registran ninguna magnitud; probablemente, al tratarse de un terreno definido solo geométricamente sin propiedades dinámicas de contacto, el sistema de simulación no genera fuerzas reales y por ello las lecturas permanecen en cero.

Control por Velocidad

Bajo consignas de velocidad, Panter experimenta ligeras oscilaciones cuando supera zonas elevadas, pero el algoritmo de navegación compensa adecuadamente las variaciones y mantiene la trayectoria global.

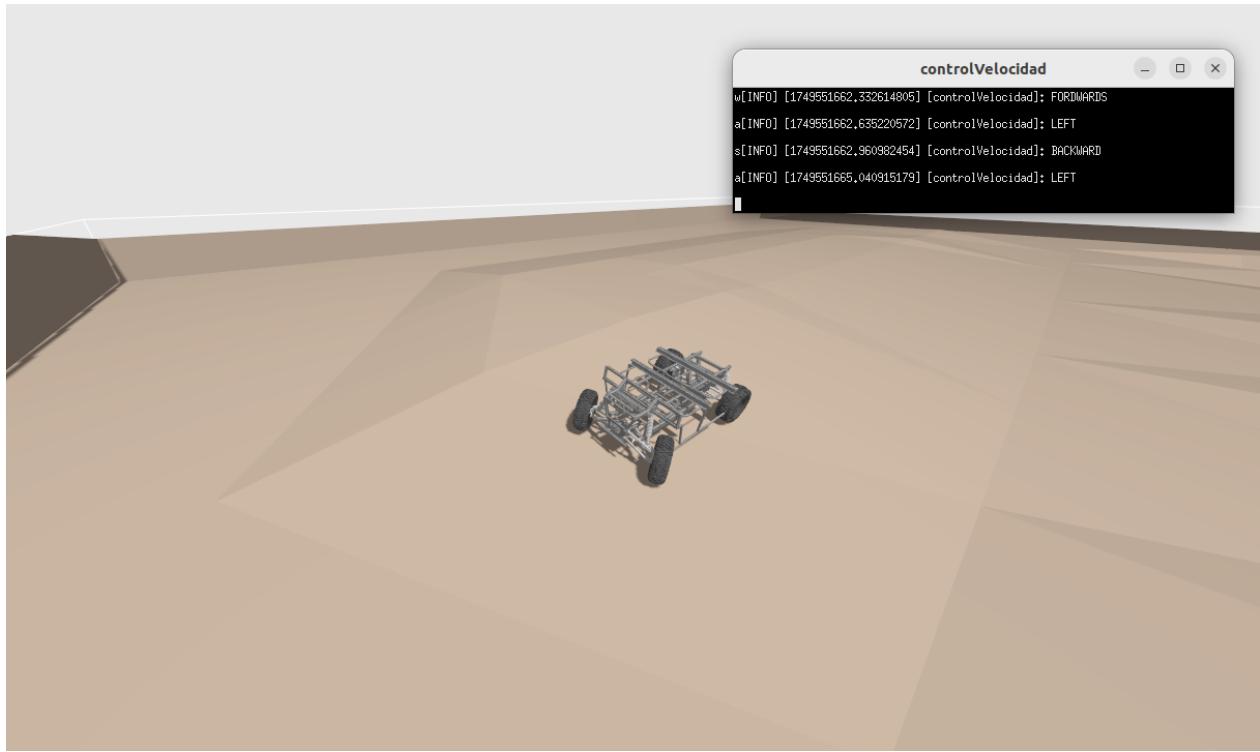


Figura 1. Enter Caption

Control por Par

Capítulo 5

Conclusiones y líneas futuras

5.1. Conclusiones

Se ha cumplido con éxito el objetivo principal de aprender a desarrollar un simulador en Gazebo. Para ello ha sido necesario configurar por completo el entorno de trabajo en Linux e integrar ROS para implementar la lógica de control. Durante el proceso he adquirido experiencia en herramientas fundamentales como Gazebo Ignition Fortress, RViz2, colcon y SolidWorks; me he iniciado en Python y XML, he profundizado en C++ y he empleado YAML para la parametrización y configuración de modelos.

Además, he puesto en práctica el control de versiones (una competencia imprescindible en proyectos reales y trabajo en equipo), lo que ha garantizado la trazabilidad y la calidad del desarrollo.

Gracias a todo ello, he diseñado, implementado y validado un simulador robótico funcional, sobre el que construir futuras mejoras y extensiones del proyecto.

5.2. Líneas futuras de trabajo

Este proyecto abre muchas posibilidades de desarrollo y mejoras. Entre ellas destacaría el modelado de las suspensiones de Panter, creación de escenarios de Gazebo, implementación de sensores, programación de diferentes sistemas de navegación en ROS e implementación del manejo del simulador con las controladoras del robot real.

Bibliografía

- [1] Calero, I. R., “Modelado 3d de un vehículo eléctrico para misiones de rescate en entornos no estructurados,” 2023.
- [2] Robotics, O., “Tutorials – gazebo api,” 2025, <https://gazebosim.org/api/gazebo/6/tutorials.html> (visitado el 2025-06-04).
- [3] Robotics, O., “Ros installation — gazebo documentation,” 2025, https://gazebosim.org/docs/all/ros_installation/ (visitado el 2025-06-04).
- [4] Ávila, J., “panter_ws – simulador de vehículo robot en ros2 y gazebo,” 2025, https://github.com/JorgeAvila102/panter_ws (visitado el 2025-06-04).
- [5] Contributors, R. W., “sw_urdf_exporter,” 2025, https://wiki.ros.org/sw_urdf_exporter (visitado el 2025-06-04).
- [6] arab-meet Contributors, “Export solidworks as urdf – sw2urdf tutorial,” 2025,
- [7] Robotics, O., “Ros-gz project template guide — fortress,” 2025, https://gazebosim.org/docs/fortress/ros_gz_project_template_guide/ (visitado el 2025-06-04).
- [8] y Dharini Dutia, M. C., “Ros 2 and gazebo integration best practices [vimeo],” 2025, <https://vimeo.com/showcase/9954564/video/767127300> (visitado el 2025-06-04).
- [9] Foundation, O. S. R., “geometry_msgs/msg/twist — ros noetic documentation,” 2025, https://docs.ros.org/en/noetic/api/geometry_msgs/html/msg/Twist.html (visitado el 2025-06-04).
- [10] Contributors, R. W., “joint_state_publisher,” 2025, https://wiki.ros.org/joint_state_publisher (visitado el 2025-06-04).

- [11] Contributors, G. S. G., “ros_gz_bridge – bridge communication between ros and gazebo,” 2025, https://github.com/gazebosim/ros_gz/tree/ros2/ros_gz_bridge#bridge-communication-between-ros-and-gazebo (visitado el 2025-06-04).
- [12] Robotics, O., “Ros 2 integration — gazebo fortress,” 2025, https://gazebosim.org/docs/fortress/ros2_integration/ (visitado el 2025-06-04).
- [13] contributors, W., “List of moments of inertia,” 2025, https://en.wikipedia.org/wiki/List_of_moments_of_inertia (visitado el 2025-06-04).
- [14] Robotics, O., “Ackermannsteering class reference,” 2025, https://gazebosim.org/api/gazebo/6/classignition_1_1gazebo_1_1systems_1_1AckermannSteering.html#details (visitado el 2025-06-04).
- [15] Foundation, O. S. R., “Tutorial: Force/torque sensor,” 2025, https://classic.gazebosim.org/tutorials?tut=force_torque_sensor (visitado el 2025-06-04).
- [16] Foundation, O. S. R., “Ros 2 humble - installation guide,” 2025, <https://docs.ros.org/en/humble/Installation/Ubuntu-Install-Debs.html> (visitado el 2025-06-04).
- [17] Robotics, O., “Gazebo fortress - install on ubuntu,” 2025, https://gazebosim.org/docs/fortress/install_ubuntu/ (visitado el 2025-06-04).
- [18] Chacon, S. y Straub, B., “Inicio – sobre el control de versiones: Instalación de git,” 2025,
- [19] Docs, G., “Clonar un repositorio,” 2025, <https://docs.github.com/es/repositories/creating-and-managing-repositories/cloning-a-repository> (visitado el 2025-06-04).

Anexo 1

Aplicación del Simulador

1.1. Instalación del Simulador de Panter

El sistema operativo en el que se ha diseñado el simulador de Panter es Ubuntu 22.04.5, sobre él, hay que hacer las siguientes instalaciones.

- Instalación de ROS 2 Humble de la web oficial [16].

Configurar el entorno de ROS2:

```
$ source /opt/ros/humble/setup.bash
```

- Instalación de Gazebo Fortress de la web oficial [17].

Luego, para tener el proyecto en el dispositivo, es necesario crear un clon del repositorio de GitHub que contiene el proyecto [4]. Para ello, en primer lugar, hay que seguir las indicaciones de instalación de Git para Linux de la web oficial [18]. Luego, abrimos la Terminal de Linux con la siguiente combinación de teclas: ctrl + alt + t. En ella, escribimos los siguientes comandos, que permiten clonar el proyecto de forma local [19].

Creamos la zona de trabajo (WorkSpace)

Carpeta donde queremos alojar el proyecto, al que llamaré 'panter_ws':

```
$ mkdir panter_ws
```

```
$ cd panter_ws
```

Clonamos el repositorio del proyecto

```
$ git clone https://github.com/JorgeAvila102/panter_ws.git
```

Otras instalaciones necesarias

- Visual Studio Code:

```
$ sudo apt install ./code_1.72.0-1664925838_arm64.deb
```

- Xterm:

```
$ sudo apt update  
$ sudo apt-get install xterm
```

Establecer el entorno de trabajo de Visual Studio Code

```
$ cd panter_ws  
$ mkdir .vscode
```

Y podemos añadir en la carpeta .vscode, los siguientes archivos de configuración que facilitarán su uso y compilación:

- **c_cpp_properties.json:**

```
{  
  "configurations": [  
    {  
      "name": "Linux",  
      "includePath": [
```

```

        "${workspaceFolder}/**",
        "/opt/ros/humble/include/**"
    ],
    "defines": [],
    "compilerPath": "/usr/bin/gcc",
    "cStandard": "c17",
    "cppStandard": "gnu++17",
    "intelliSenseMode": "clang-x64"
}

],
"version": 4
}

```

- **tasks.json** permite compilar pulsando las teclas ctrl + Shift + b:

```

{
    "version": "2.0.0",
    "tasks": [
        {
            "type": "shell",
            "command": "colcon",
            "args": [
                "build", "--symlink-install"
            ],
            "group": "build",
            "label": "colcon: build",
            "presentation": {
                "panel": "new",

```

```

        "focus": true,
        "reveal": "silent",
        "clear": true
    } } ] }

```

Modificación en URDF

Para el correcto funcionamiento, debemos realizar el siguiente cambio en el archivo panter.urdf: cambiar el usuario jorge, por el nuevo usuario.

```

1432 <gazebo>
1433   <plugin filename="gz_ros2_control-system" name="gz_ros2_control::GazeboSimROS2ControlPlugin">
1434     <robot_param>robot_description</robot_param>
1435     <robot_param_node>robot_state_publisher</robot_param_node>
1436     <parameters>/home/jorge/panter_ws/src/control_pkg/config/control_config.yaml</parameters>
1437   </plugin>
1438 </gazebo>

```

Figura 1. Enter Caption

Ejecutar este comando en la terminal de Linux.

```
$ sed -i "s|/home/jorge/panter_ws|${HOME}/panter_ws|g"
~/panter_ws/src/control_pkg/urdf/panter.urdf
```

Configurar bash

Copiar todo el siguiente código en una única vez en la terminal y ejecutarlo.

```
$ echo 'source /opt/ros/humble/setup.bash' >> ~/.bashrc

$ echo 'source ${HOME}/panter_ws/install/setup.bash' >> ~/.bashrc

$ echo 'export IGN_GAZEBO_RESOURCE_PATH="${HOME}/panter_ws/install/
description_pkg/share/description_pkg/worlds:${HOME}/panter_ws/
install/description_pkg/share/description_pkg/
models:$IGN_GAZEBO_RESOURCE_PATH"' >> ~/.bashrc
```

```

$ echo 'export GZ_SIM_RESOURCE_PATH="$HOME/panter_ws/install/
description_pkg/share/description_pkg/worlds:$HOME/panter_ws/
install/description_pkg/share/description_pkg/
models:$GZ_SIM_RESOURCE_PATH"' >> ~/.bashrc

$ echo 'export IGN_GAZEBO_SYSTEM_PLUGIN_PATH="$IGN_GAZEBO_
SYSTEM_PLUGIN_PATH:/opt/ros/humble/lib"' >> ~/.bashrc

$ echo 'export IGN_GAZEBO_PHYSICS_ENGINE_PATH="/usr/
lib/x86_64-linux-gnu/ignition/physics"' >> ~/.bashrc

$ echo 'export LD_LIBRARY_PATH="$LD_LIBRARY_PATH:/opt/ros/humble/
lib:$HOME/panter_ws/install/gz_ros2_control/lib"' >> ~/.bashrc

$ source ~/.bashrc

```

1.2. Uso del Simulador

Una vez las herramientas han sido instaladas, y el entorno de trabajo modificado, podemos utilizar el simulador.

Compilación del WorkSpace

- **Opción 1:** Sobre algún archivo del proyecto en Visual Studio Code, presionamos ctrl + Shift + b y posteriormente Enter.
- **Opción 2:** en la terminal:

```

$ cd ~/panter_ws
$ colcon build

```

Lanzamiento del simulador

- Control de velocidad:

```
$ cd ~/panter_ws/src
$ ros2 launch bringup_pkg project_vel.launch.py
```

- Control de torque:

```
$ cd ~/panter_ws/src
$ ros2 launch bringup_pkg project.launch.py
```

Importante en el control de torque: se debe pulsar el play Figura 2 en Gazebo antes de **5 segundos**, en el caso de que no sean pulsados, los controladores dan error (Figura 3) y es necesario volver a lanzarlo.



Figura 2. Play Gazebo

```
[...]
[effort_controller]: Loaded effort_controller
[ruby $(which ign) gazebo-2] [INFO] [1749040699.152884908] [controller_manager]: Configuring controller 'effort_controller'
[ruby $(which ign) gazebo-2] [INFO] [1749040699.154442891] [effort_controller]: configure successful
[INFO] [spawner-10]: process started with pid [10425]
[ruby $(which ign) gazebo-2] [ERROR] [1749040704.156720309] [controller_manager]: Switch controller timed out after 5.000000 seconds!
[spawner-9] [ERROR] [1749040704.159302535] [spawner_effort_controller]: Failed to activate controller : effort_controller
[ruby $(which ign) gazebo-2] [INFO] [1749040704.162447630] [controller_manager]: Loading controller 'steering_controller'
[spawner-10] [INFO] [1749040704.238328940] [spawner_steering_controller]: Loaded steering_controller
[ruby $(which ign) gazebo-2] [INFO] [1749040704.239296881] [controller_manager]: Configuring controller 'steering_controller'
[ruby $(which ign) gazebo-2] [INFO] [1749040704.239815334] [steering_controller]: configure successful
[ERROR] [spawner-9]: process has died [pid 10413, exit code 1, cmd '/opt/ros/humble/lib/controller_manager/spawner effort_controller --ros-args'].
[ruby $(which ign) gazebo-2] [ERROR] [1749040709.241660822] [controller_manager]: Switch controller timed out after 5.000000 seconds!
[spawner-10] [ERROR] [1749040709.242946147] [spawner_steering_controller]: Failed to activate controller : steering_controller
[ERROR] [spawner-10]: process has died [pid 10425, exit code 1, cmd '/opt/ros/humble/lib/controller_manager/spawner steering_controller --ros-args'].
```

Figura 3. Enter Caption

Cuando ha sido iniciado correctamente, podemos ver los mensajes de la Figura 4.

```
[INFO] [spawned-9]: Desired controller update period (0.01 s) is slower than the gazebo simulation period (0 s).
[INFO] [spawned-9]: process started with pid [10890]
[ruby $(which ign) gazebo-2] [INFO] [1749040972.586428622] [controller_manager]: Loading controller 'effort_controller'
[spawned-9] [INFO] [1749040972.603244964] [spawned_effort_controller]: Loaded effort_controller
[ruby $(which ign) gazebo-2] [INFO] [1749040972.604430855] [controller_manager]: Configuring controller 'effort_controller'
[ruby $(which ign) gazebo-2] [INFO] [1749040972.605851568] [effort_controller]: configure successful
[ruby $(which ign) gazebo-2] [INFO] [1749040972.830784771] [effort_controller]: activate successful
[spawned-9] [INFO] [1749040972.832790204] [spawned_effort_controller]: Configured and activated effort_controller
[INFO] [spawned-9]: process has finished cleanly [pid 10890]
[INFO] [spawned-10]: process started with pid [10902]
[ruby $(which ign) gazebo-2] [INFO] [1749040974.300143030] [controller_manager]: Loading controller 'steering_controller'
[spawned-10] [INFO] [1749040974.317552838] [spawned_steering_controller]: Loaded steering_controller
[ruby $(which ign) gazebo-2] [INFO] [1749040974.318506871] [controller_manager]: Configuring controller 'steering_controller'
[ruby $(which ign) gazebo-2] [INFO] [1749040974.319031208] [steering_controller]: configure successful
[ruby $(which ign) gazebo-2] [INFO] [1749040974.337543293] [steering_controller]: activate successful
[spawned-10] [INFO] [1749040974.349240086] [spawned_steering_controller]: Configured and activated steering_controller
[INFO] [spawned-10]: process has finished cleanly [pid 10902]
```

Figura 4. Controladores iniciados correctamente

Comandos de control

Los comandos de teclado para controlar el robot han sido definidos en la tabla 1.

Para ambos controladores, es necesario pulsar las teclas teniendo la terminal de XTerm abierta y pulsada.

Para finalizar la simulación se recomienda pulsar ctrl + c sobre la terminal de Linux.