Universidad de Málaga

Escuela de Ingenierías Industriales

Ingeniería de Sistemas y Automática

Trabajo Fin de Grado

Desarrollo del Gemelo Digital de un Brazo Robótico de Cuatro Articulaciones con ROS 2

Ingeniería Electrónica, Robótica y Mecatrónica

Autor: Pascual González Redondo

Tutor: Antonio José Muñoz Ramírez
Cotutor: Juan Manuel Gandarias Palacios

21 de febrero de 2025

Declaración de Originalidad del Trabajo Fin de Grado

D./Dña. Pascual González Redondo
DNI/Pasaporte: 49214282-D. Correo electrónico: grpascu@uma.es
Titulación: Ingeniería Electrónica, Robótica y Mecatrónica
Título del Proyecto/Trabajo: Desarrollo del Gemelo Digital de un Brazo Robótico de Cuatro Articulaciones con ROS 2

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.

Asimismo, 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 21 de febrero de 2025

Fdo.: Pascual González Redondo

Resumen

Desarrollo del Gemelo Digital de un Brazo Robótico de Cuatro Articulaciones con ROS 2

Autor: Pascual González Redondo

Tutor: Antonio José Muñoz Ramírez

Cotutor: Juan Manuel Gandarias Palacios

Departamento: Ingeniería de Sistemas y Automática

Titulación: Ingeniería Electrónica, Robótica y Mecatrónica

Palabras clave Robótica Manipuladores, ROS 2, Digital, Simulación, RVIZ, MoveIt, Gazebo

A lo largo de esta memoria se abordará el concepto de Gemelo Digital aplicado a un manipulador robótico. En primera instancia se propone la posibilidad de realizar un control para un brazo manipulador articulado, el cual cuenta con cuatro articulaciones rotacionales y diversos eslabones que conectan el brazo y lo hacen más funcional. Este robot manipulador ha sido desarrollado en el marco del Roborescue, el equipo de robots de rescate de la Universidad de Málaga.

Debido a que se trata de un robot de rescate, el control en este caso se propone como algo crítico y que debe ser completamente funcional, depurable y mejorable constantemente. El uso de ROS 2 en este caso, supone una mejora con respecto a sus antecesores. Este *software* dedicado a la integración y coordinación de diversos sistemas de robótica necesarios para la correcta construcción de interfaces propias, control y simulación supone un gran avance.

Para realizar el control, se propone partir de su Gemelo Digital. Este concepto hace referencia a una copia digital del manipulador, la cual debe ser capaz de coordinar todos los sistemas que toman importancia para su control, simular las variables y comandarlas con el objetivo de su correcto funcionamiento, así como la creación de una visualización 3D del mismo que pueda acceder a las transmitidas por el robot. El uso de este concepto permite poder depurar los métodos utilizados sin necesidad de contar con el sistema físico.

Para este trabajo se aborda la construcción del modelo 3D en simulación, creando los sistemas capaces de describir su configuración y simular el control. Para ello, se hace uso de herramientas y *software* dedicadas para el control en robótica, ROS 2. Dicho *software* cuenta con lenguajes particularizados para la descripción y construcción del modelo (URDF), herramientas potentes para visualizarlo y simularlo (RVIZ y Gazebo), paquetes dedicados al control (ros_control) y herramientas para la planificación de trayectorias (MoveIt).

Todo ello supone una extensa investigación del *software* con el que se trabajará en este marco, ampliando la posibilidad de coordinar e integrar todos los sistemas dedicados al correcto funcionamiento del mismo.

Dedico este trabajo a mi familia,
que me ha apoyado desde el momento
en el que entré en la carrera.

Agradecimientos

Primero, doy gracias a mi familia
que ha aguantado mis quejas y me ha dado
gran soporte durante esta etapa.

También doy gracias a mis amigos y amigas,
que han sido fundamentales durante esta etapa
y en contadas ocasiones me han dado alas
y un gran soporte.

Por último, doy gracias a mis tutores,
que han tenido gran paciencia conmigo
y empatizado con mi situación personal.
A mi tutor por confiar en mi todos estos años
y darme la oportunidad de mejorar y
aprender cosas nuevas.
A mi cotutor por darme herramientas, por
no perder la esperanza conmigo, darme
energía y calma y por toda la ayuda aportada.

Acrónimos y Notación Matemática

ROS	Robot Operating System
URDF	Unified Robot Description Format
RVIZ	ROS Visualization
DDS	Data Distribution Service
IA	Inteligencia Artificial
OMPL	Open Motion Planning Library

Índice

Página

Índice de Figuras xvii

Índice de Códigos xix

1 Introducción 1
1.1. Motivación . . . 1
1.2. Antecedentes . . . 3
1.2.1. Simulación . . . 4
1.2.2. Control . . . 4
1.3. Objetivos . . . 7
1.4. Justificación del Uso de ROS 2 como Herramienta de Desarrollo . . . 7
1.5. Contenido de la Memoria . . . 8

2 Descripción de la Estructura del Robot 9
2.1. Descomponer un robot . . . 10
2.1.1. Cadenas Cinemáticas . . . 10
2.1.2. Componentes de una cadena cinemática . . . 10
2.1.3. Tipos de Cadenas Cinemáticas . . . 11
2.1.4. Descomposición de Brazo Robótico del Roborescue . . . 13
2.2. Descripción URDF . . . 18
2.2.1. Elementos de URDF . . . 18
2.2.2. Consideraciones y claves para la descripción URDF . . . 20
2.3. Descripción URDF del Brazo Robótico del Roborescue . . . 22
2.3.1. Entorno de ROS 2 . . . 23
2.4. Directorios relacionados con la Descripción URDF . . . 23
2.4.1. Directorio: description . . . 24

3 Visualización del Robot 33
3.1. Herramienta de visualización: RVIZ . . . 34

3.1.1. Ejecución y configuración en RVIZ 34
3.1.2. Publicación de *topics* para la visualización en RVIZ 37
3.2. Resultados de visualización del Robot 38
3.2.1. Rotación de *Arm* 1 y *Arm* 3 39
3.2.2. Rotación de *Arm* 2 y *Arm* 3 39
3.2.3. Rotación de todas las articulaciones 40
3.3. Directorios relacionados con la Visualización 40
3.3.1. Directorio: *worlds* 40
3.3.2. Directorio de ejecución: launch 41

4 Simulación del Robot 43
4.1. Directorios relacionados con la Simulación 43
4.1.1. Directorio: description 44
4.1.2. Directorio de ejecución: launch 46
4.2. Resultados de simulación del Robot 49

5 Control de Posición del Robot 51
5.1. Control de Posición mediante ROS 2 Control y Gazebo 52
5.1.1. ROS 2 Control 52
5.2. Directorios relacionados con el Control de Posición 52
5.2.1. Directorio: description 53
5.2.2. Directorio de ejecución: launch 54
5.2.3. Directorio de ejecución: config 55
5.3. Resultados del Control de Posición 56
5.3.1. Robot en posición inicial 57
5.3.2. Robot en Pose 1 57
5.3.3. Robot en Pose 2 58
5.3.4. Robot en Pose 3 59

6 Planificación de Trayectorias del Robot 61
6.1. Control de Trayectorias mediante MoveIt 62
6.2. Directorios relacionados con el Control de Trayectorias 62
6.2.1. Directorio: description 62
6.2.2. Directorio de ejecución: launch 63
6.2.3. Directorio de configuración: config 67
6.3. Uso de MoveItSetupAssistant 67
6.3.1. Lanzamiento de MoveItSetupAssistant 68
6.3.2. Carga de la descripción URDF 68
6.3.3. Self-Collitions 68

6.3.4. Planning Groups 69
6.3.5. Robot Poses 70
6.3.6. Controllers 70
6.3.7. Generate Package 71
6.4. Resultados de la Planificación de Trayectorias mediante MoveIt 71
6.4.1. Vista Inicial 72
6.4.2. Establecer una Pose Final 73
6.4.3. Trayectoria hacia la Pose Final 73
6.4.4. Trayectoria Finalizada 74
6.5. Consideraciones 75
6.5.1. Sincronización de Relojes 75
6.5.2. Colisiones y Posiciones Erróneas 75
6.5.3. Restablecimiento y reinicio de la Pose 76

7 Mejora de la Visualización: Introducción de Mallas **77**
7.1. Adición de los Meshes a la Descripción URDF 78
7.2. Resultados de la Visualización del Robot con Meshes 78
7.2.1. Pose 1 79
7.2.2. Pose 2 80

8 Conclusiones **81**

Bibliografía **85**

Índice de Figuras

Figura **Página**

1.1. Esquema de funcionamiento de un Gemelo Digital 2

2.1. Esquemático de la cadena cinemática de tipo Grafo o Bucle Cerrado 12
2.2. Esquemático de la Cadena Cinemática Abierta 12
2.3. Esquemático de la Cadena Cinemática Abierta (Árbol) 13
2.4. Brazo Robótico del Roborescue 13
2.5. Base 14
2.6. Articulación 1 + Eslabón Asociado 14
2.7. Articulación 2 + Eslabón Asociado 15
2.8. Eslabón 1 16
2.9. Articulación 3 + Eslabón Asociado 16
2.10. Eslabón 2 17
2.11. Articulación 4 + Eslabón Asociado 18
2.12. Esquemático del Árbol del Sistema 31

3.1. Entorno RVIZ 34
3.2. Botón Add en RVIZ 35
3.3. RobotModel como *plugin* de RVIZ 35
3.4. Topic de RobotModel en RVIZ 36
3.5. Visualización del Robot en RVIZ sin publicaciones al topic 36
3.6. joint_state_publisher_gui 37
3.7. Visualización del Robot en RVIZ 38
3.8. Rotación de *Arm* 1 y *Arm* 3 39
3.9. Rotación de *Arm* 2 y *Arm* 3 39
3.10. Rotación de todas las articulaciones 40

4.1. Robot simulado en Gazebo al inicio 49
4.2. Robot simulazo en Gazebo pasados unos segundos 49

5.1. Robot en posición inicial simulado con Gazebo y Control de Posición 57

5.2. Robot en Pose 1 simulado con Gazebo y Control de Posición 57
5.3. Plot del control: Pose 1 57
5.4. Visualización del contenido del topic joint_states: Pose 1 58
5.5. Robot en Pose 2 simulado con Gazebo y Control de Posición 58
5.6. Plot del control: Pose 2 58
5.7. Visualización del contenido del topic joint_states: Pose 2 59
5.8. Robot en Pose 3 simulado con Gazebo y Control de Posición 59
5.9. Plot del control: Pose 3 59
5.10. Visualización del contenido del topic joint_states: Pose 3 60

6.1. MoveItSetupAssistant - Home Page 68
6.2. MoveItSetupAssistant - Robot Poses 70
6.3. Plugin visual en RVIZ de MotionPlanning 72
6.4. MotionPlanning - Home 72
6.5. MotionPlanning - Establecer una Pose Final 73
6.6. MotionPlanning - Trayectoria hacia Pose Final 73
6.7. MotionPlanning - Trayectoria Finalizada 74
6.8. Representación de la Trayectoria mediante Gráfica de las Posiciones Articulares 74
6.9. MotionPlanning - Colisión 75

7.1. Visualización del Robot Real 79
7.2. Visualización del Robot Real - Pose 1 79
7.3. Visualización del Robot Real - Pose 2 80

Índice de Códigos

2.1. Descripción del robot en URDF . . . 19
2.2. Definición de links en URDF . . . 19
2.3. Definición de joints en URDF . . . 20
2.4. robot.urdf.xacro . . . 24
2.5. inertial_macros.xacro . . . 25
2.6. geometry_values.xacro . . . 26
2.7. urdf_colours.xacro . . . 27
2.8. urdf_robot_description.xacro . . . 27
3.1. urdf.launch.py . . . 41
4.1. robot.sim.xacro . . . 44
4.2. robot_gazebo.xacro . . . 45
4.3. gazebo.launch.py . . . 46
4.4. sim.launch.py . . . 47
5.1. robot.control.xacro . . . 53
5.2. control.launch.py . . . 55
5.3. joints.yaml . . . 55
6.1. new_robot.urdf.xacro . . . 63
6.2. sim_control.launch.py . . . 64
6.3. arm_controllers.yaml . . . 67
6.4. ros2_controllers.yaml . . . 70
7.1. realrobot.urdf.xacro . . . 78

CAPÍTULO 1

Introducción

Contenido

1.1. Motivación 1
1.2. Antecedentes 3
1.2.1. Simulación 4
1.2.2. Control 4
1.3. Objetivos 7
1.4. Justificación del Uso de ROS 2 como Herramienta de Desarrollo 7
1.5. Contenido de la Memoria 8

El mundo de la robótica presenta herramientas variadas para la construcción de manipuladores robóticos. El marco de este trabajo se sitúa en el *Roborescue* [1], equipo de robots de rescate terrestres de la Universidad de Málaga. Este equipo pretende participar en la RoboCup Rescue [1], una competición internacional enfocada en la robótica de rescate, donde distintos países aportan prototipos de robots de rescate variados con la intención de superar las pruebas acordadas por la organización.

1.1. Motivación

La robótica de rescate presenta una gran diversidad de robots, desde drones a robots anfibios y robots terrestres. Dentro de este último grupo, se presentan propuestas distintas dependiendo del

[1] La información relativa al equipo se encuentra en su repositorio de GitHub

enfoque que quieran aportar. En caso de misiones de reconocimiento, la construcción se centra en una mejora de la movilidad del robot, aportándole la capacidad de moverse por terrenos áridos y críticos. También la integración de cámaras y sensores potentes que permitan el escaneo del terreno para dotar al robot de una mejor estrategia a la hora de sumergirse en misiones con cierta complicación. Para misiones de rescate, además de las características anteriores, se hace uso de los manipuladores robóticos. Estos permiten una mejora en este ámbito, dotando al robot de la capacidad de acceder a áreas peligrosas o inaccesibles para los humanos, manipular objetos frágiles, aportar ciertas ayudas en el momento del rescate y una mayor autonomía.

El robot terrestre del equipo de la Universidad de Málaga, tiene acoplado un manipulador robótico. Para la construcción del mismo, los departamentos de control, mecánica, eléctrica y electrónica trabajan conjuntamente, aportando las soluciones necesarias para su uso. El estudio mecánico del brazo ha contemplado la posibilidad de crearlo con cuatro articulaciones rotacionales, manejadas mediante servomotores. En cuanto a lo que respecta la parte del *software*, se plantea el uso del más comúnmente usado en robótica actualmente, ROS 2 [2]. Este permite integrar todos los sistemas necesarios para la creación del manipulador, como es el control, sensorización, monitorización...

Este trabajo se enfoca más concretamente en el estudio del control del manipulador, el cual se realiza mediante herramientas de simulación con la creación de un Gemelo Digital [3]. Este concepto fue un pilar en la industria 4.0 y es altamente usado en la industria 5.0 [4], dada la importancia de una mejora continua de la eficiencia, sostenibilidad y resiliencia en los procesos industriales. Se define como la réplica virtual de un producto o sistema que se actualiza continuamente con datos provenientes de su contraparte física y de su entorno. El esquema aportado a continuación muestra de manera visual el funcionamiento completo del mismo.

El objeto de estudio de este trabajo abarca desde la creación del modelo de simulación hasta el envío de variables de control. Si bien es cierto que la conexión con su interfaz física no es posible por el momento, se desarrollan las herramientas por las cuales se hará un control del brazo en simulación, obteniendo como salida las variables necesarias para que el sistema real pueda realizar movimientos precisos. Un esquema del funcionamiento del Gemelos puede verse representado en la Figura 1.1.

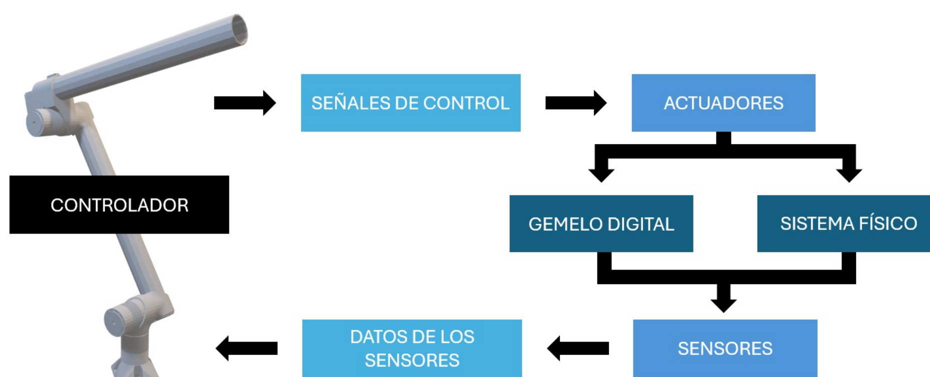

Figura 1.1: Esquema de funcionamiento de un Gemelo Digital

Las ventajas que el uso de la simulación aporta son diversas. Esta aporta la capacidad de visualizar de manera remota el estado del robot (en caso de realizar la interfaz que conecta el modelo físico con el modelo en simulación). También permite realizar pruebas concretas en un entorno seguro mediante la creación de escenarios y condiciones específicas de forma previa a su implementación. Esto aporta la ventaja de mejorar el sistema sin necesidad de contar con el sistema físico real. Al realizar dichas pruebas, se pueden aislar aquellos errores o mejoras que influencien el desempeño de una tarea concreta.

1.2. Antecedentes

El trabajo se ha realizado en el marco del proyecto K-Project [5] del equipo de robots de rescate de la UMA: Roborescue. La robótica de rescate es un área de la robótica multidisciplinaria, que busca desarrollar sistemas robóticos capaces de operar en distintas situaciones de emergencia, dando paso a diferentes terrenos donde la intervención de los mismos puede ser necesaria. [6]

Su relevancia comenzó en la década de los 90, impulsada por eventos catastróficos como los terremotos de Kobe (1995) y Turquía (1999). Además, el ataque al *World Trade Center* en 2001, hizo que la adopción de robots para las misiones de rescate ganase importancia, evitando la intervención humana directa en la exploración de escombros. Desde ese momento, el avance en tecnología, Inteligencia Artificial, sensórica, diseño mecánico y comunicación inalámbrica, han hecho posible el desarrollo de robots de rescate aún más avanzados. [7]

Existen robots de rescate enfocados para distintos ámbitos. Los robots de rescate terrestres están diseñados para desplazarse sobre el suelo mediante ruedas, orugas o patas articuladas, adaptándose a diversos terrenos y obstáculos. [8] Por otro lado, se crean robots aéreos o drones, capaces de explorar superficies en un tiempo más reducido. Se trata de vehículos no tripulados en su mayoría, que cuentan con sensores y cámaras para sobrevolar áreas afectadas. Sus objetivos más comunes son el reconocimiento y mapeo, la detección de víctimas y comunicación. [9] Finalmente, los robots submarinos tienen la capacidad de operar bajo el agua. Estos resultan útiles en desastres naturales como inundaciones o para la ayuda en accidentes marítimos. [10]

En el marco actual, Roborescue UMA se centra en la creación de robots de rescate terrestres, que representan un papel crucial dentro de este mundo, debido a su capacidad para moverse por entornos estructuralmente complejos y críticos. Estos robots deben tener la capacidad de acceder a terrenos inestables, espacios reducidos, además de superar obstáculos.

En este caso, se ha optado por el desarrollo de un robot de rescate terrestre con un brazo robótico acoplado, lo que le permite manipular y retirar ciertos obstáculos, interaccionar con el entorno y una mayor versatilidad de aplicaciones.

1.2.1. Simulación

La visualización en ROS 2 permite, como se dijo anteriormente, depurar el modelo de descripción URDF y todo lo relacionado con la geometría del mismo. Sin embargo, carece de físicas.

La simulación de robots mediante software es un paso esencial en el desarrollo de sistemas robóticos. Antes de hacer uso del *hardware* real, conviene probar las físicas en simulación.

Simular un robot no solo ahorra costos y tiempo, sino que permite trabajar en entornos controlados cuyas condiciones son ajustables a las especificaciones del usuario. Existe la posibilidad de replicar escenarios específicos, como se requiere para Roborescue, cuyo objetivo es participar en la RoboCup [1]. Este concurso impone ciertas condiciones y pruebas de distinto tipo. A pesar de que el equipo cuenta con un taller, requeriría de varias personas para montar las pruebas y, cuando se quisiese cambiar de prueba, habría que desmontar y montar todo de nuevo. Mediante un *software* simulado donde las físicas estén correctamente establecidas y los escenarios estén construidos a medida, resultará más fácil hacer distintas pruebas sin necesidad de demasiado esfuerzo.

La ventaja que esto conlleva es que solo hay que construirlo una vez. Cuando la simulación funciona, la adición o sustracción de objetos en el escenario o el cambio del mismo resulta mucho más sencillo, así como cambiar el propio robot.

Por ejemplo, en el caso de programación para robots tipo ABB, antes de implantar la programación en el robot, conviene hacer uso del software RoboStudio, que permite establecer los caminos y trayectorias que, posteriormente, se le implantarán al robot físicamente. En caso de ROS 2, una de las herramientas que existe es Gazebo.

1.2.1.1. Herramientas de Simulación en ROS 2

A la hora de realizar simulaciones en tiempo real que incluyan físicas dentro de ROS 2, existen algunas herramientas avanzadas de gran utilidad que pueden ser usadas en este ámbito.

Gazebo es una potente herramienta de simulación avanzada que permite crear y probar modelos robóticos en entornos virtuales realistas, añadiendo las dinámicas físicas del mismo, como la gravedad, fricción y colisiones. Además, es una herramienta de código abierto, por lo que se ha convertido en uno de los simuladores más importantes en el ámbito de la robótica. También permite simular sensores como cámaras, LIDAR, sensores de proximidad o GPS, entre otros, o crear escenarios personalizados acordes a las necesidades de cada uno.

También existen otros simuladores como Isaac Sim [11], que permite realizar simulación avanzada con gráficos de alta fidelidad y un soporte de aprendizaje profundo, o PyBullet [12], que es un simulador ligero enfocado a la dinámica y el control.

1.2.2. Control

Dentro de la robótica, una de las partes más importantes es el control. Si se centra el foco de atención en los brazos robóticos se dice que el control es una disciplina fundamental dentro de

este mundo. Su uso permite diseñar, implementar y optimizar estrategias concretas para darle una funcionalidad específica de manera precisa y eficiente.

Estos sistemas son ampliamente utilizados en diversas áreas, y más aún en la época actual, donde la industria 5.0. Ésta representa la próxima evolución en la fabricación y producción industrial, caracterizada por la integración de tecnologías avanzadas con un enfoque centrado en las personas, la sostenibilidad, mejora continua y personalización de las tareas mucho mayor que en las etapas tecnológicas anteriores. La robótica desempeña un papel fundamental. Durante la industria 4.0 se genera una mejora de las tareas buscando la automatización, la primera inclusión de las IAs y las tecnologías IoT. Sin embargo, el lado humano queda apartado en esta industria. Durante esta nueva etapa, se busca tomar la colaboración entre los humanos y las máquinas, consiguiendo que trabajos que conllevan un gasto tanto mental como físico a las personas, se faciliten mediante el uso de la robótica y las IAs. De este modo surgen los robots colaborativos (Cobots). El control de los mismos es sumamente importante.

Los brazos robóticos desempeñan un papel crucial en la nueva industria, dando flexibilidad a las tareas que desempeñan. El control de un brazo robot implica gestionar varios aspectos técnicos y matemáticos como:

- **Cinemática**: Se conoce como el estudio de la relación entre las articulaciones y la posición del extremo del robot (efecto final), o dicho de otra forma, es el estudio de los movimientos del robot analizando posición, velocidad y aceleración.
    - **Cinemática Directa**: Consiste en determinar la posición del extremo a partir de la posición de cada articulación.
    - **Cinemática Inversa**: Consiste en el cálculo de la posición de cada una de las articulaciones para alcanzar una posición final deseada.
- **Dinámica**: Relaciona el movimiento del robot con las fuerzas implicadas en el mismo. Este modelo establece las relaciones matemáticas entre las coordenadas articulares, fuerzas y pares aplicados y los parámetros del robot (masas, inercias, geometrías, etc).

Por otro lado, existen diversos métodos de control, que son enfoques para controlar los brazos robóticos según la tarea que desempeñarán y los requisitos del sistema global. Los métodos más conocidos son:

- **Control de Posición**: Consiste en asegurar una posición concreta para cada articulación mediante comandos de posición (coordenadas angulares o lineales)
- **Control de Velocidad**: Regula la velocidad de las articulaciones para garantizar movimientos suaves mediante comandos de velocidad.

- **Control de Fuerza**: Trabaja con la fuerza o torque que se le aplica a los actuadores asegurando un objetivo deseado. Este control es importante en robots que interactúan físicamente con su entorno.
- **Control de Trayectorias**: Garantiza que el robot siga una trayectoria determinada en el espacio, estableciendo restricciones de tiempo y velocidad.

Existen otros tipos de control en robótica, y también diferentes herramientas matemáticas, aunque estos son los más comunes. A día de hoy, los cálculos matemáticos se siguen realizando y son muy importantes. Sin embargo, al uso, existen potentes herramientas que hacen más fácil el uso de los modelos matemáticos de control. El grado en el que se realizan los cálculos depende de las herramientas y bibliotecas utilizadas. Dentro de ROS 2, herramientas como Gazebo y bibliotecas como **Pinocchio** automatizan gran parte de estos procesos, reduciendo la necesidad de cálculos manuales. A pesar de esto, es esencial comprender los principios básicos para hacer su configuración y personalización de forma correcta.

1.2.2.1. Herramientas clave para el Control en ROS 2

Cuando se pretende hacer un control robótico y su simulación en ROS 2, es muy útil conocer las herramientas que se están utilizando actualmente.

Los cálculos cinemáticos, que se realizan de forma manual, pueden facilitarse mediante el uso de bibliotecas específicas, las cuales pueden usarse en ROS 2. Se nombrará en este caso las bibliotecas *Pinocchio*[2] y *KDL*[3]. El uso de estas permite calcular trayectorias y resolver cinemática directa e inversa de forma totalmente automática. En el caso de Pinocchio, su uso se extiende a modelos avanzados, ya que permite realizar cálculos de cinemática directa e inversa para el uso de robots complejos con múltiples eslabones y grados de libertad, cinemática diferencial (jacobianos) y dinámica. En el caso de KDL, su uso hace sencillo el cálculo de transformaciones, cinemática directa e inversa para manipuladores.

Finalmente, el cálculo del control puede ser la parte más complicada a la hora de crear estos sistemas. ROS 2 proporciona herramientas fundamentales para la implementación de controladores y la gestión de movimientos de actuadores y robots. Las más conocidas son:

- **ROS 2 Control** [13]: Es un paquete con distintos módulos que permite implementar controladores de posición, velocidad y fuerza mediante la integración de hardware real o simuladores como Gazebo.
- **MoveIt** [14]: Es la herramienta más usada a la hora de hacer un control de manipuladores. Su uso incluye bibliotecas como KDL y Pinnochio para facilitar los cálculos anteriores e implementa un control de trayectorias resolviendo las cinemáticas de forma automática.

[2] Puede encontrarse información relacionada con el paquete de Pinocchio en la sección de ROS Index - Pinocchio
[3] Puede encontrarse información relacionada con el paquete de KDL en la documentación de ROS Wiki - KDL

1.3. Objetivos

En cuanto a los objetivos planteados para este trabajo, se aborda el concepto de Gemelo Digital. La intención del mismo es crear el modelo en simulación del manipulador robótico de cuatro articulaciones del Roborescue, así como su respectivo control. De esta forma se abordan las primeras partes en la creación del Gemelo Digital final. Este trabajo sirve como una base sólida que poder continuar para, finalmente, conectar el sistema físico real con la simulación creada.

Se tienen como objetivos el desarrollo del modelo descriptivo a nivel cinemático y dinámico. Dicho modelo se representará en tres dimensiones mediante herramientas visuales de ROS 2. También, su inclusión en simulación, donde se desarrollarán los procesos implicados en el control articular del robot y la planificación de trayectorias aprovechando las herramientas que este *software* ofrece.

Para poder realizar todo lo expuesto de la forma más completa posible, se ha profundizado en el mundo de la robótica con ROS 2, se ha experimentado con distintos métodos de simulación y control para poder conseguir un resultado realista, flexible y lo más sencillo posible. También se ha llevado a cabo un extenso aprendizaje sobre ROS mediante el seguimiento de la teoría aportada por la propia organización y repositorios relacionados.

1.4. Justificación del Uso de ROS 2 como Herramienta de Desarrollo

En la actualidad, se utilizan diversas herramientas para el desarrollo de *frameworks* para robots [15]. Sin embargo, a la hora de trabajar con sistemas en tiempo real o para integrar múltiples sensores y componentes *hardware* y *software* de distintas naturalezas, existen algunas herramientas con mejores características que otras y más potentes, cuya función está enfocada a este mundo.

Este es el caso de ROS 2, el cual está diseñado para trabajar con sistemas distribuidos, permitiendo la división de un sistema en diversos nodos. ROS permite trabajar con sistemas diferentes intercomunicados mediante suscripción y publicación de mensajes, ofreciendo así una arquitectura modular que facilita la integración de componentes tales como sensores, controladores, algoritmos, etc.

Por otro lado, el soporte que ofrece en tiempo real es el adecuado para aplicaciones críticas en robótica, especialmente con el paso de ROS a ROS 2 [16]. Además de las múltiples librerías que se pueden utilizar para este tipo de aplicaciones, y de la facilidad en cuanto a recursos, que el uso libre de este *software* ofrece.

Es por ello que este trabajo se ha decidido hacer mediante este *software*. ROS 2 incluye una serie de herramientas aptas para el propósito buscado. Para realizar la descripción de las dinámicas y cinemáticas del robot se genera una descripción URDF [17] que, a pesar de no ser específico, ROS 2 hace uso de este formato de forma eficaz. Para la visualización del modelo descrito, se utiliza la herramienta RVIZ2 [18] proporcionada por ROS 2, la cual es compatible con sistemas en tiempo real.

Por otro lado, el uso de Gazebo [19], propio de ROS 2, ha permitido la simulación del mismo. A esto se le añaden las interfaces *software* y *hardware* proporcionadas por ros-control [20].

Por tanto, el interés de este trabajo reside en que se forma una base sólida por la cual obtener el Gemelo Digital completo del brazo robot. Haciendo uso de ROS 2 es posible ejecutar el *workspace* completo y transportarlo de un lugar a otro simplemente instalando las librerías adecuadas. También puede servir como una guía práctica por la cual aprender y customizar el brazo robot de forma sencilla. También permite reutilizar código, nodos y topics para el robot real.

1.5. Contenido de la Memoria

La memoria se dividirá en diversas partes. Cada una de ellas está enfocada a una fase del proyecto distinta y se explicará de tal modo que pueda ser aprovechada por otras aplicaciones similares. Las fases son las siguientes:

- **Descripción Mecánica**: Lo primero, es realizar el modelado en descripción mediante URDF y su visualización con RVIZ2.
- **Visualización de la configuración articular:** Seguidamente, se tiene como objetivo la correcta visualización de la configuración articular de cada una de las cuatro articulaciones, dando pie a un movimiento limitado impuesto en la construcción del URDF.
- **Simulación dinámica:** Por otro lado, se pretende simular el comportamiento dinámico del robot haciendo uso de un simulador incluido en ROS 2 conocido como Gazebo.
- **Control de posición:** Con el objetivo de hacer un control que haga este trabajo lo más cercano a un Gemelo Digital, se configuran y crean los archivos necesarios para la implementación de un control articular del robot, simulado en Gazebo.
- **Planificación de trayectorias:** Mediante el uso de la herramienta MoveIt, y un conjunto de programas de ejecución, se realiza el control de trayectorias del robot en simulación visualizado el RVIZ y Gazebo.
- **Mejora de la visualización:** Para hacer la visualización más cercana a la realidad y poder partir desde ese punto para hacer el Gemelo Digital completo, se introduce el modelo 3D creado para este brazo robótico mediante mallas, de tal forma que se visualiza de una forma lo más realista posible.

Capítulo 2

Descripción de la Estructura del Robot

Contenido

2.1. Descomponer un robot **10**
2.1.1. Cadenas Cinemáticas 10
2.1.2. Componentes de una cadena cinemática 10
2.1.3. Tipos de Cadenas Cinemáticas 11
2.1.4. Descomposición de Brazo Robótico del Roborescue 13
2.2. Descripción URDF **18**
2.2.1. Elementos de URDF 18
2.2.2. Consideraciones y claves para la descripción URDF 20
2.3. Descripción URDF del Brazo Robótico del Roborescue **22**
2.3.1. Entorno de ROS 2 23
2.4. Directorios relacionados con la Descripción URDF **23**
2.4.1. Directorio: description 24

Este capítulo presenta los pasos a seguir para realizar el proceso de modelado y descripción estructural de un robot. En este caso, dicho modelado se dará mediante la descripción URDF, un archivo en formato xml utilizado en ROS. Este tipo de modelado es esencial para conectar el diseño del *hardware* con las herramientas de *software*, permitiendo la simulación, control y análisis del robot en entornos virtuales.

2.1. Descomponer un robot

Antes de empezar a describir el modelo de un robot, debe descomponerse en las distintas partes que este contenga. Una vez claro, se comienza a definir el modelo del robot en formato URDF.

2.1.1. Cadenas Cinemáticas

La forma de definir la estructura de un robot rígido es mediante *cadenas cinemáticas*[1]. Una cadena cinemática es una representación jerárquica de las relaciones cinemáticas entre los diferentes componentes de un sistema mecánico poliarticulado. Se utiliza en robótica para modelar el movimiento relativo de eslabones conectados por articulaciones.

Cada nodo de la cadena representa una articulación, mientras que los bordes representan restricciones o relaciones que rigen su movimiento. Además, ayudan en el diseño y control de brazos y robots móviles, lo que permite un movimiento preciso y la ejecución de tareas.

La construcción de una cadena cinemática implica definir los componentes del sistema, establecer relaciones entre ellos y determinar los grados de libertad para cada articulación. Este proceso suele comenzar con un diseño conceptual, seguido de la creación de un modelo matemático que describe las ecuaciones cinemáticas que rigen el movimiento del sistema.

2.1.2. Componentes de una cadena cinemática

Una cadena cinemática está compuesta por los elementos que describen la estructura física y las relaciones de movimiento entre las partes de un sistema robótico. Principalmente, dichos modelos se componen de nodos y aristas, que representan los eslabones y articulaciones, respectivamente.

2.1.2.1. Eslabones (Links)

Los eslabones son las partes rígidas de un robot que conectan con las articulaciones. Tienen dimensiones geométricas (longitud o masa, entre otras), se describen en relación a un sistema de referencia y no poseen grados de libertad propios, sino que se mueven según las articulaciones conectadas.

2.1.2.2. Articulaciones (Joints)

Las articulaciones conectan los eslabones y permiten movimiento relativo entre ellos. Estas determinan los grados de libertad articulares (DoF) del sistema.

En ROS se describen varios tipos de articulaciones según el tipo de movimiento que realizan:

[1] Cierta información relativa a las cadenas cinemátcias puede encontrarse en la web de The University of Science & Technology

- **Revolute (Rotacional):** Permiten la rotación alrededor un eje fijo y tiene un rango limitado especificado mediante límites superior e inferior. Se utiliza para modelar brazos robóticos, codos o rodillas.

- **Continuous (Continua):** Permiten la rotación infinita alrededor de un eje fijo. Es similar al tipo *revolute* pero carece de límites angulares. Se utiliza para el modelado de ruedas, rotores o giros continuos.

- **Prismatic (Prismática):** Permiten el deslizamiento a lo largo de un eje. Su rango está limitado mediante límites superior e inferior, que indican la distancia máxima y mínima de desplazamiento. Se utiliza para modelar pistones o rieles.

- **Fixed (Fija):** No Permiten el movimiento relativo entre los eslabones conectados a ella. Al no permitir ningún tipo de movimiento no es realmente una articulación, pero a la hora de definir un modelo es útil para asociar un sistema de referencia a un punto concreto (como en el caso de sensores, cámaras...).

- **Floating (Flotante):** Permiten el movimiento en todas direcciones, es decir, Permiten seis grados de libertad articular (DoF): tres rotaciones y tres traslaciones. Se utiliza para simular robots sin restricciones iniciales.

- **Planar (Planar):** Permiten el movimiento en un plano bidimensional: dos traslaciones y una rotación. Sus restricciones limitan al plano definido. Es útil a la hora de definir robots que se mueven en planos 2D, como es el caso de los robots móviles.

2.1.3. Tipos de Cadenas Cinemáticas

Dentro de las cadenas cinemáticas, se pueden encontrar distintos tipos según la intención que se tenga a la hora de modelar un sistema. Cada tipo define cómo están conectados los elementos (*links* y *joints*) y cómo se transmiten los movimientos dentro del mismo.

2.1.3.1. Cadena Cinemática Cerrada

Se trata de una cadena que permite conexiones y ciclos cerrados en la cadena, lo que induce redundancia en el control y una mayor estabilidad estructural.

Al contener bucles cinemáticos, se generan múltiples rutas entre nodos. Son utilizados en robots donde las conexiones múltiples son necesarias para mejorar la precisión y la capacidad de carga. El esquema de su representación se puede ver en la Figura 2.1.

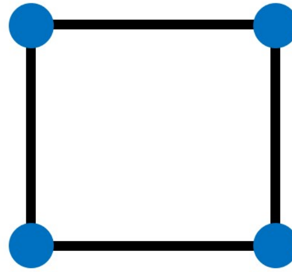

Figura 2.1: Esquemático de la cadena cinemática de tipo Grafo o Bucle Cerrado

2.1.3.2. Cadena Cinemática Abierta

Una cadena cinemática abierta conecta un nodo inicial (base) con uno o varios nodos finales mediante una secuencia de articulaciones. Un esquema habitual para un brazo robótico es el representado en la Figura 2.2.

Es la representación más típica para brazos robóticos.

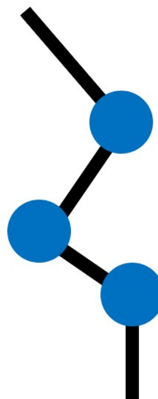

Figura 2.2: Esquemático de la Cadena Cinemática Abierta

Por otro lado, la representación de la cadena en forma de árbol implica la ramificación del nodo raíz en varios caminos. Cada eslabón puede contener múltiples hijos, pero un solo padre. No contiene ciclos cerrados en su configuración.

Es muy utilizado en robótica para robots con movimientos unidireccionales o sin ciclos cinemáticos. Un esquema representativo puede verse en la Figura 2.3

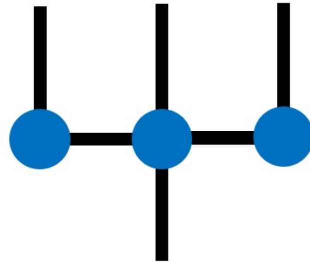

Figura 2.3: Esquemático de la Cadena Cinemática Abierta (Árbol)

2.1.4. Descomposición de Brazo Robótico del Roborescue

Una vez definidos los elementos y tipos de cadenas cinemáticas generales de robótica, es hora de particularizar al caso práctico que engloba este trabajo, el brazo robótico del Roborescue.

En este caso, se trata de un brazo robótico con una cadena cinemática abierta. Su aspecto se muestra en la Figura 2.4.

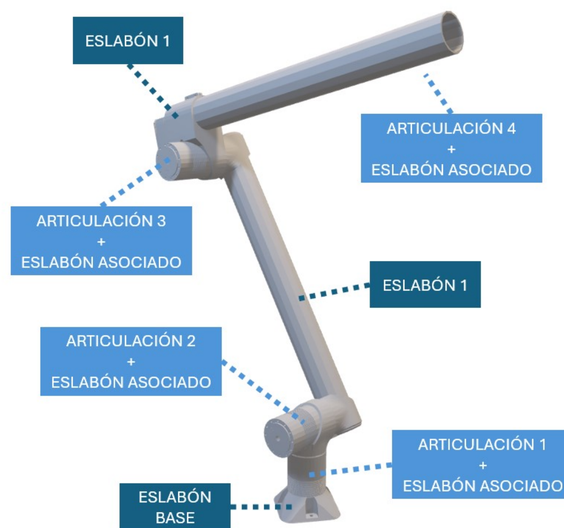

Figura 2.4: Brazo Robótico del Roborescue

Para el modelado de este brazo se debe tener en cuenta su movimiento, tipos de articulaciones y medidas.

2.1.4.1. Eslabón Base

El eslabón base se presenta en la Figura 2.5.

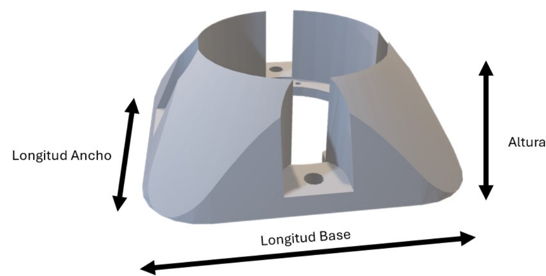

Figura 2.5: Base

Como se observa, se trata de una base de tipo piramidal con un corte en la parte superior. Para modelarla de forma sencilla, se puede ver como un prisma con base cuadrada y altura menor a las medidas de la base. Tomando medidas en SolidWorks del modelo construido, se pueden obtener los siguientes valores:

- **Longitud Base:** 400 mm
- **Longitud Ancho:** 400 mm
- **Altura:** 120 mm

2.1.4.2. Articulación 1 + Eslabón Asociado

Lo siguiente será definir la primera articulación. En este caso se trata de un motor cilíndrico que rota libremente sobre el eje z. Además, lleva asociado un eslabón dado por el estudio de la mecánica del brazo. La Figura 2.6 muestra dicho conjunto.

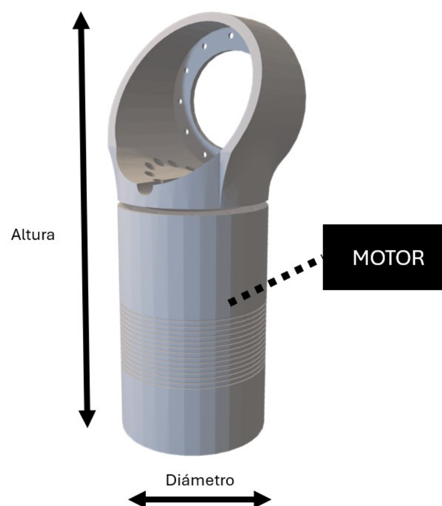

Figura 2.6: Articulación 1 + Eslabón Asociado

Como se puede modelar como un cilindro, se toman los valores midiendo la altura y el radio del cilindro:

- **Altura:** 200 mm
- **Radio:** 50 mm

2.1.4.3. Articulación 2 + Eslabón Asociado

Seguidamente se observa acoplada a la primera articulación, otro motor cilíndrico que daría el giro en el eje y relativo del nuevo sistema de coordenadas, definiendo así esta segunda articulación. A este se le añade un eslabón con forma cilíndrica que extiende su longitud en el eje de rotación y permite el acople de la siguiente parte (tal y como se observa en la Figura 2.7.

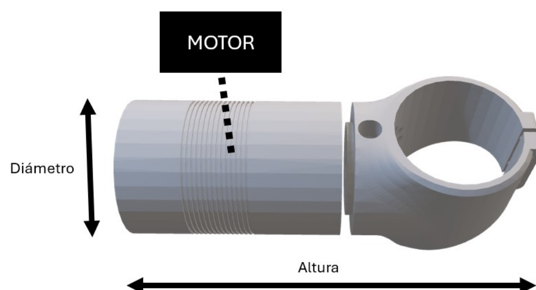

Figura 2.7: Articulación 2 + Eslabón Asociado

Al igual que en el caso anterior, este conjunto se puede modelar como un cilindro simple:

- **Altura:** 200 mm
- **Radio:** 50 mm

2.1.4.4. Eslabón 1

Siguiendo la misma dinámica que en los casos anteriores, se observa un cilindro que servirá para trasladar en el eje z el sistema de la siguiente articulación, haciendo el papel de eslabón de unión. Este cilindro no cuenta con ningún movimiento, por lo que se considera como una articulación fija (mostrado en la Figura 2.8).

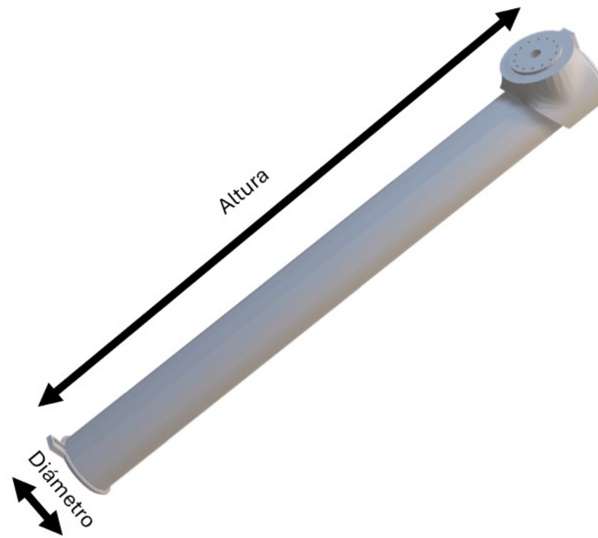

Figura 2.8: Eslabón 1

El cilindro toma las siguientes medidas:

- **Altura:** 800 mm
- **Radio:** 50 mm

2.1.4.5. Articulación 3 + Eslabón Asociado

Acoplado al eslabón anterior, se inserta otro motor que permite el movimiento en el eje y relativo, pero rotado 180º de la anterior articulación, tal y como se observa en la Figura 2.9.

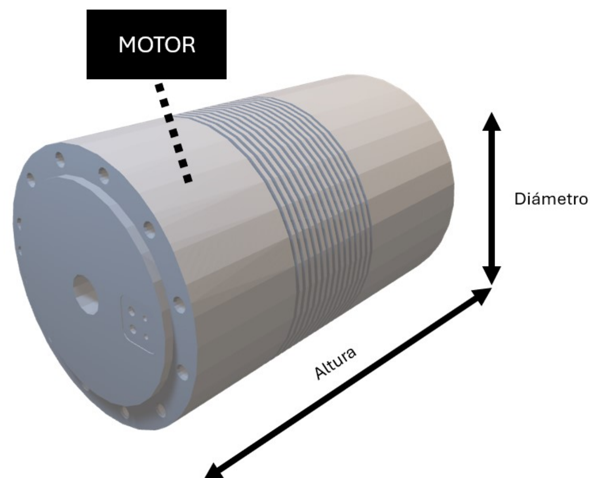

Figura 2.9: Articulación 3 + Eslabón Asociado

Tomando las medidas geométricas del motor, se obtienen los siguientes valores:

- **Altura:** 200 mm
- **Radio:** 50 mm

2.1.4.6. Eslabón 2

Llegando al final de esta cadena cinemática, se traslada y rota la siguiente articulación mediante un acople mecánico. Al modelar este eslabón, se añade el motor para facilitar la posterior integración en el sistema. Dicho conjunto se muestra en la Figura 2.10.

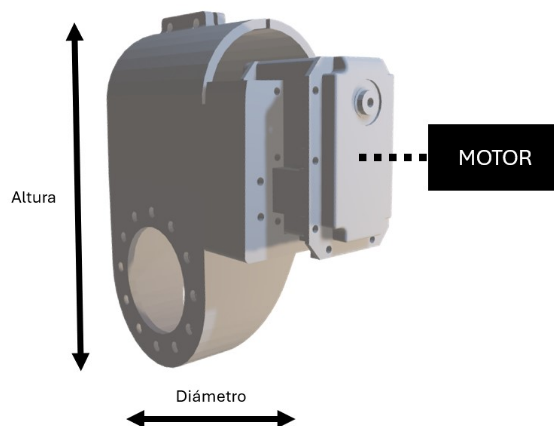

Figura 2.10: Eslabón 2

Modelándolo de nuevo como un cilindro, se obtienen las siguientes medidas:

- **Altura:** 100 mm
- **Radio:** 50 mm

2.1.4.7. Articulación 4 + Eslabón Asociado

La última articulación del robot, se trata de un motor que rota también alrededor de su eje. Se inserta un eslabón en el eje del motor para ampliar su longitud. El eslabón que se acopla a dicha articulación se en la Figura 2.11. A pesar de que este conjunto tiene una articulación, el modelado se hace de esta forma para la descripción del modelo en control.

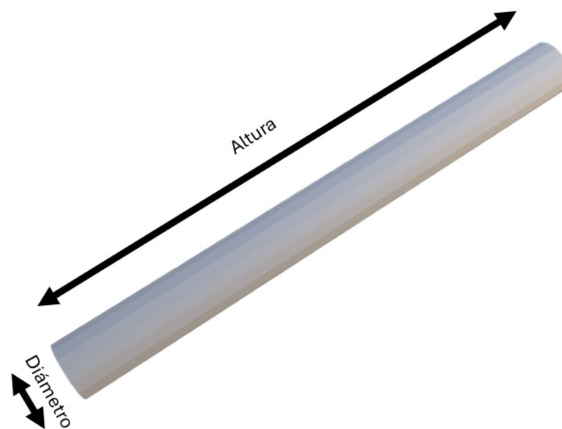

Figura 2.11: Articulación 4 + Eslabón Asociado

De nuevo, se toman las medidas totales para este conjunto:

- **Altura:** 1000 mm
- **Radio:** 50 mm

2.2. Descripción URDF

La descripción URDF (Unified Robot Description Format) es una especificación XML que permite describir la cadena y propiedades de un robot, de modo que, mediante elementos XML, se pueden definir algunas características del mismo, tales como:

- Aspecto Visual
- Cinemática y Dinámica
- Colisiones

Dicho formato es utilizado en el ecosistema de ROS para la simulación y el control de robots en entornos virtuales.

2.2.1. Elementos de URDF

Dentro de la descripción URDF, existen diversos elementos y atributos a tener en cuenta para definir el modelo de un robot.

2.2.1.1. robot

Lo primero al construir el modelo de un robot en descripción URDF es definir el nombre del robot. Este es el elemento raíz del archivo URDF. Se define el nombre y se encapsulan los demás elementos.

Código 2.1: Descripción del robot en URDF

```
1 <robot name="robot_name">
2   ...
3 </robot>
```

2.2.1.2. *link*

El siguiente elemento que se introduce en las descripciones URDF son los eslabones de la cadena cinemática a completar. Los eslabones o nodos se nombran como *link*.

Cada uno de los eslabones debe tener un nombre único. Dentro de los *link* se definen atributos tales como:

- **visual**: Mediante este elemento se describe la representación visual de cada eslabón. Se definen varios parámetros dentro del mismo:
  - ***geometry***: Define la forma que tendrá el eslabón (*cube*, *sphere*, *cylinder* o *mesh*)
  - **material**: Define el color o la textura que tendrá el eslabón.
- ***collision***: Mediante este elemento se describen las colisiones posibles enlazadas al robot. En este caso se podría decir que las colisiones las dan las propias geometrías, dado que son elementos sólidos.
- ***inertial***: Mediante este elemento se describen las inercias asociadas a cada elemento del robot según sus características geométricas. Para ello se deben conocer ciertas propiedades de los elementos tales como la masa, el centro de gravedad o el tensor de inercia.

Código 2.2: Definición de links en URDF

```
1 <link name="link_name">
2   <visual>
3     <geometry>
4       <geometry_element size="l h w" />
5     </geometry>
6     <material name="color_name">
7       <color rgba="r g b a" />
8     </material>
9   </visual>
10  <collision>
11    <geometry>
12      <geometry_element size="l h w" />
13    </geometry>
14  </collision>
```

```
15   <inertial>
16     <mass value="mass_value" />
17     <origin xyz="x y z" rpy="roll pitch yaw" />
18     <inertia
19       ixx="ixx_value" ixy="ixy_value" ixz="ixz_value"
20       iyy="iyy_value" iyz="iyz_value" izz="izz_value" />
21   </inertial>
22 </link>
```

2.2.1.3. *joint*

El elemento *joint* se introduce en las descripciones URDF definiendo las conexiones entre dos *link*. A cada *link* le corresponde un *joint* asociado de forma que se va haciendo de forma modular.

Hay varios subelementos que forman las *joint*:

- ***type***: Como se comentó anteriormente, las articulaciones se clasifican según el tipo de movimiento que realizan (*revolute*, *continuous*, *prismatic*, *fixed*, *floating* o *planar*).
- ***parent***: Define el *link* del que proviene.
- ***child***: Define el *link* que le precede. Suele nombrarse con el mismo nombre que se le da al *joint* definido.
- ***origin***: Se define la posición y orientación relativas entre los *link*.
- ***axis***: Define los límites de movimiento para los casos de articulaciones tipo *revolute* y *prismatic*.
- ***dynamics***: Se definen las propiedades dinámicas tales como la fricción.

Código 2.3: Definición de joints en URDF

```
1 <joint name="joint_name" type="joint_type">
2   <parent link="parent_link_name" />
3   <child link="child_link_name" />
4   <origin xyz="x y z" rpy="roll pitch yaw" />
5   <axis xyz="x_axis y_axis z_axis" />
6   <limit lower="lower_limit" upper="upper_limit" effort="effort_limit"
        velocity="velocity_limit" />
7   <dynamics damping="damping_value" friction="friction_value" />
8 </joint>
```

2.2.2. Consideraciones y claves para la descripción URDF

Cuando se hace una descripción en URDF, hay ciertas consideraciones que se toman en cuenta para poder hacer todo de forma rápida y efectiva, sin tener la necesidad de visualizar y retocar el modelo innumerables veces.

A continuación se nombrarán y explicarán dichas consideraciones que harán su descripción mucho más sencilla y eficaz.

2.2.2.1. xacro

Xacro (XML Macros) es una extensión del formato XML que permite la generación de descripciones más compactas y legibles de archivos XML para robots, especialmente en el contexto de URDF.

Esta extensión permite al usuario definir varios archivos de tipo XML en lugar de un archivo XML exageradamente largo, es decir, se crean archivos para las diferentes partes del código. De esta forma, se define un archivo para cada uno de los elementos globales del código, facilitando la interacción con dichos elementos y proporcionando sencillez al código y accesibilidad a la hora de realizar cambios en él.

Haciendo uso de xacro se puede plantear la creación de herramientas que permitan la generación de gemelos digitales de forma genérica, es decir, dando como entrada el número de articulaciones y eslabones, geometrías de los mismos y algunos elementos más se generará el archivo URDF completo para dichas entradas.

Algunos de los elementos que se pueden definir en un archivo xacro de forma individual son:

- **Medidas**: Puede resultar útil establecer las medidas de la geometría de los elementos en un archivo individual, dando la posibilidad de realizar cambios a posteriori.

- **Inercias**: Para evitar el cálculo reiterado de ciertos valores, se pueden establecer las fórmulas del cálculo de inercias para distintos elementos geométricos. De este modo, se hacen llamadas a las funciones de cálculo cuando sea necesario con tan solo los valores geométricos de los elementos nombrados.

- **Colores**: A la hora de generar un archivo URDF, se pueden establecer distintos colores que permitan la diferenciación entre los elementos, ya sea separar articulaciones de elementos fijos u otras utilidades que pueden ser de ayuda cuando se requiere de una visualización.

Simplemente separando los elementos nombrados del código principal, se obtienen archivos más ordenados y legibles. También se puede usar para incluir todos los códigos en un mismo archivo haciendo uso de funciones como xacro:include.

Como conclusión, el uso de xacro es esencial para la generación de archivos URDF, permitiendo la definición de plantillas que podrán ser utilizadas varias veces en diferentes partes del código, reduciendo la redundancia y el esfuerzo manual al definir elementos similares. También da pie a definir parámetros personalizables y expresiones matemáticas, permitiendo la creación de descripciones parametrizadas y flexibles.

2.2.2.2. Definición de orígenes

Una de las partes más críticas, cuando se crea una descripción URDF, es la definición de los orígenes de los eslabones y articulaciones. La definición de orígenes es fundamental para colocar y orientar correctamente estos elementos.

El elemento *origin* define la posición y orientación de los elementos en el espacio tridimensional:

- **Posición(xyz):**
  - Especifica la ubicación en coordenadas cartesianas (x, y, z) en metros
  - Define el origen de coordenadas del componente en relación con la referencia de su elemento superior
- **Orientación(rpy):**
  - Representa la rotación del componente en términos de *roll* (rotación alrededor del eje x), *pitch* (rotación alrededor del eje y) y *yaw* (rotación alrededor del eje z), expresado en radianes.

Debido a la naturaleza de los elementos, se debe hacer la diferenciación entre el origen de los eslabones y las articulaciones. De esta forma, la descripción de dichos orígenes resultará más sencilla una vez se tenga claro.

- **Origen de los *Joints*:** Cuando se define el origen en un *joint* (articulación), se está especificando el punto en el cual dos *links* (eslabones) se conectan. La orientación y posición de la relación que el eslabón hijo tiene con el padre.

  El origen de la articulación marca la base desde la cual comenzará el siguiente eslabón. Es sumamente importante colocarlo correctamente para que los elementos estén correctamente posicionados y orientados. Además, en este punto se indicará también (en caso de tener limitaciones) el eje director que permite el giro de la articulación

- **Origen de los *Links*:** En el caso de los eslabones, el origen está generalmente relacionado con el centro de gravedad del eslabón. Dicho centro de gravedad está marcado con la geometría del mismo. Es por ello que es crucial conocer dicha geometría a la hora de definir un archivo URDF.

  Si se da el caso en el que el punto de conexión no coincide con el centro de gravedad del eslabón (por ejemplo si se quiere posicionar el eslabón en el extremo del mismo), será necesario ajustar la posición modificando el atributo correspondiente.

Conocidas dichas consideraciones, la creación y descripción del modelo del robot en URDF resultará más sencilla y efectiva.

2.3. Descripción URDF del Brazo Robótico del Roborescue

En cuanto a la descripción del caso particular del brazo robótico del Roborescue en URDF, se ha optado por crear un paquete propio para este apartado.

2.3.1. Entorno de ROS 2

Se crea un *workspace* completo donde se añadirán todos los paquetes realizados durante este trabajo. Un *workspace* es un directorio estructurado que actúa como entorno de trabajo para desarrollar, compilar y gestionar paquetes de ROS 2, o para entenderlo de mejor forma, se trata de una carpeta raíz en la cual se sitúan todos los paquetes.

Un paquete de ROS 2 es la unidad básica de organización y distribución de código. Este directorio contiene archivos relacionados con funcionalidades específicas como nodo, mensajes, servicios, parámetros, configuraciones... Los paquetes son modulares y se diseñan para que actúen de forma independiente unos de otros o de forma interdependientes.

Dentro de un *workspace* típico de ROS 2, se sitúa el directorio src, en el que se crearán todos los paquetes. Una vez compilados los paquetes, aparecerán otros directorios dentro del *workspace* que contienen archivos instalados (binarios y bibliotecas), archivos temporales y archivos de registro: *install*, *build* y *log* respectivamente.

A la hora de crear un paquete para un *workspace* concreto, debe situarse dentro del directorio /worskpace_name/src. Una vez ahí, se crea el paquete mediante el siguiente comando:

```
ros2 pkg create -build-type ament_cmake package_name
```

Mediante ese comando específico de ROS 2, se crea un paquete que contiene ciertos directorios y archivos necesarios para la compilación y el funcionamiento de este. Los archivos que se crean son:

- **CMakeList.txt:** Es un archivo de texto que contiene las instrucciones necesarias para que el compilador conozca como se debe compilar e instalar el paquete

- **package.xml:** Contiene la información de metadatos del paquete: nombre, versión, dependencias, etc. Es obligatorio para cualquier paquete de ROS 2

También se crean dos carpetas genéricas, que son *src* e *install*. En la carpeta *src* es común situar archivos de código fuente en C++ (.cpp). En la carpeta *install* se sitúan las cabeceras de los archivos de código fuente (.hpp). No siempre se van a utilizar, pero a la hora de crear nodos de ROS 2 serán necesarias.

Por otro lado, dentro de un paquete, se pueden encontrar otros directorios como *launch* (archivos de lanzamiento para ejecución de nodos) o *config* (archivos de configuración) entre otros.

En concreto, para este apartado, se crea el paquete urdf_pkg.

2.4. Directorios relacionados con la Descripción URDF

El paquete urdf_pkg tiene en su interior las carpetas necesarias para la descripción URDF. En esta ocasión se crean los siguientes directorios:

- **description**

- **launch**
- **worlds**

El único directorio que tiene relación con este apartado es el directorio de **description**:

2.4.1. Directorio: description

Este directorio es el que contiene toda la descripción URDF. Para estructurarlo de mejor forma, se hace uso de **xacro**, dividiendo la descripción en cinco archivos.

2.4.1.1. robot.urdf.xacro

Este archivo es el global. Se usa para crear el robot, darle nombre e incluir al resto de paquetes. De esta forma, resulta más sencilla la adición o eliminación de los otros archivos. Su contenido es el siguiente:

Código 2.4: robot.urdf.xacro

```
1 <?xml version="1.0"?>
2 <robot xmlns:xacro="http://www.ros.org/wiki/xacro" name="robot">
3
4     <xacro:include filename="inertial_macros.xacro" />
5     <xacro:include filename="geometry_values.xacro" />
6     <xacro:include filename="urdf_colours.xacro" />
7     <xacro:include filename="urdf_robot_description.xacro" />
8
9 </robot>
```

2.4.1.2. inertial_macros.xacro

Dentro de este archivo se realizan las operaciones matemáticas para la obtención de las inercias de un prisma y de un cilindro.

Para el cálculo de los momentos de inercia del prisma, se describen las siguientes fórmulas matemáticas:

$$I_{xx} = \frac{1}{12}m(h^2 + w^2), \tag{2.1}$$

$$I_{yy} = \frac{1}{12}m(l^2 + h^2), \tag{2.2}$$

$$I_{zz} = \frac{1}{12}m(l^2 + w^2), \tag{2.3}$$

$$I_{xy} = I_{xz} = I_{yz} = 0. \tag{2.4}$$

Donde:

- m: Masa del prisma

- l: Longitud de la Base
- w: Anchura de la Base
- h: Altura

Para el cálculo de los momentos de inercia del cilindro, se describen las siguientes fórmulas matemáticas:

$$I_{xx} = I_{yy} = \frac{1}{12} m(3r^2 + l^2), \tag{2.5}$$

$$I_{zz} = \frac{1}{2} mr^2, \tag{2.6}$$

$$I_{xy} = I_{xz} = I_{yz} = 0. \tag{2.7}$$

Donde:

- I_{xx} y I_{yy}: Son los momentos de inercia respecto a los ejes x e y.
- I_{zz}: Es el momento de inercia respecto al eje z (eje longitudinal del cilindro).
- I_{xy}, I_{xz}, I_{yz}: Son los productos de inercia, que son cero debido a la simetría del cilindro.

En caso de tener otras geometrías dentro de la descripción URDF, también se calcularían en este archivo. La codificación es la siguiente:

Código 2.5: inertial_macros.xacro

```
1  <?xml version="1.0"?>
2  <robot xmlns:xacro="http://www.ros.org/wiki/xacro" >
3
4      <xacro:macro name="inertial_box" params="mass length width height *
           origin">
5          <inertial>
6              <xacro:insert_block name="origin"/>
7              <mass value="${mass}" />
8              <inertia ixx="${(1/12) * mass * (height*height + width*width)}"
9                      iyy="${(1/12) * mass * (length*length + height*height)}"
10                     izz="${(1/12) * mass * (length*length + width*width)}"
11                     ixy="0.0" ixz="0.0" iyz="0.0" />
12         </inertial>
13     </xacro:macro>
14
15     <xacro:macro name="inertial_cylinder" params="mass length radius *origin
           ">
16         <inertial>
17             <xacro:insert_block name="origin"/>
18             <mass value="${mass}" />
19             <inertia ixx="${(1/12) * mass * (3*radius*radius + length*length
                   )}" ixy="0.0" ixz="0.0"
20                     iyy="${(1/12) * mass * (3*radius*radius + length*length)
                         }" iyz="0.0"
```

```
21                      izz="${(1/2) * mass * (radius*radius)}" />
22              </inertial>
23      </xacro:macro>
24
25  </robot>
```

2.4.1.3. geometry_values.xacro

Como se nombró anteriormente, resulta útil generalizar los archivos de descripción URDF para dar mayor flexibilidad a la hora de cambiar la medida de alguna de las articulaciones. Para ello, se crea este archivo, el cual cuenta con todas las medidas correspondientes a las geometrías del robot.

Código 2.6: geometry_values.xacro

```
1  <?xml version="1.0"?>
2  <robot xmlns:xacro="http://www.ros.org/wiki/xacro" >
3
4      <xacro:property name="base_length" value="0.4" />
5      <xacro:property name="width_length" value="0.4" />
6      <xacro:property name="height_box" value="0.12" />
7
8      <xacro:property name="arm1_length" value="0.2" />
9      <xacro:property name="arm1_radius" value="0.05" />
10
11     <xacro:property name="arm2_length" value="0.2" />
12     <xacro:property name="arm2_radius" value="0.05" />
13
14     <xacro:property name="cylinder1_length" value="0.8" />
15     <xacro:property name="cylinder1_radius" value="0.05" />
16
17     <xacro:property name="arm3_length" value="0.2" />
18     <xacro:property name="arm3_radius" value="0.05" />
19
20     <xacro:property name="cylinder2_length" value="0.1" />
21     <xacro:property name="cylinder2_radius" value="0.05" />
22
23     <xacro:property name="arm4_length" value="1" />
24     <xacro:property name="arm4_radius" value="0.05" />
25
26 </robot>
```

2.4.1.4. urdf_colours.xacro

En este archivo se escriben los materiales que serán usados dentro de la descripción URDF. En este caso se ha optado por crear dos materiales, que corresponden al color blanco y rojo. El blanco se da para las articulaciones fijas y el rojo para las que tienen algún eje de rotación. Además, se añaden como propiedades de xacro para, en caso de querer cambiar el color de algún tipo, tan solo se tenga que hacer una vez desde este archivo.

En caso de ser necesarios más colores, vale con añadirlos a este documento y asignarlos donde corresponda dentro del archivo de descripción URDF.

La codificación es la siguiente:

Código 2.7: urdf_colours.xacro

```
<?xml version="1.0"?>
<robot xmlns:xacro="http://www.ros.org/wiki/xacro" >

    <material name="white">
        <color rgba="1 1 1 1"/>
    </material>

    <material name="red">
        <color rgba="1 0 0 1"/>
    </material>

    <xacro:property name="fixed_link_colour" value="white" />
    <xacro:property name="revolute_link_colour" value="red" />

</robot>
```

2.4.1.5. urdf_robot_description.xacro

Por último, se crea el archivo *main* de la descripción URDF. Dentro de este archivo se declaran las articulaciones (*joint*) y eslabones (*link*). También se definen los ejes de coordenadas de cada uno, se establecen los colores y tipos de articulación en cada caso, como se explicó en apartados anteriores. El código creado para este caso particular es el mostrado a continuación:

Código 2.8: urdf_robot_description.xacro

```
<?xml version="1.0"?>
<robot xmlns:xacro="http://www.ros.org/wiki/xacro" >

    <!-- This first link called "world" is empty -->
    <link name="world"></link>

    <joint name="base_joint" type="fixed">
        <origin xyz="0 0 0" rpy="0 0 0"/>
        <parent link="world"/>
        <child link="base_link"/>
    </joint>

    <link name="base_link">
        <visual>
            <origin xyz="0 0 ${height_box/2}" rpy="0 0 0"/>
            <geometry>
                <box size="${base_length} ${width_length} ${height_box}" />
            </geometry>
            <material name="${fixed_link_colour}" />
        </visual>
        <collision>
            <origin xyz="0 0 ${height_box/2}" rpy="0 0 0"/>
            <geometry>
                <box size="${base_length} ${width_length} ${height_box}" />
```

```
25              </geometry>
26          </collision>
27          <xacro:inertial_box mass="1.0" length="${base_length}" width="${
                width_length}" height="${height_box}">
28              <origin xyz="0 0 ${height_box/2}" rpy="0 0 0"/>
29          </xacro:inertial_box>
30      </link>
31
32      <joint name="arm1_joint" type="revolute">
33          <origin xyz="0 0 ${height_box}" rpy="0 0 0"/>
34          <parent link="base_link"/>
35          <child link="arm1_link"/>
36          <axis xyz="0 0 1"/>
37          <limit lower="${-2*pi}" upper="${2*pi}" velocity="100" effort="100"
                />
38          <dynamics damping="10.0" friction="10.0"/>
39      </joint>
40
41      <link name="arm1_link">
42          <visual>
43              <origin xyz="0 0 ${arm1_length/2}" rpy="0 0 0"/>
44              <geometry>
45                  <cylinder length="${arm1_length}" radius="${arm1_radius}" />
46              </geometry>
47              <material name="${revolute_link_colour}"/>
48          </visual>
49          <collision>
50              <origin xyz="0 0 ${arm1_length/2}" rpy="0 0 0"/>
51              <geometry>
52                  <cylinder length="${arm1_length}" radius="${arm1_radius}" />
53              </geometry>
54          </collision>
55          <xacro:inertial_cylinder mass="1.0" length="${arm1_length}" radius="
                ${arm1_radius}">
56              <origin xyz="0 0 ${arm1_length/2}" rpy="0 0 0"/>
57          </xacro:inertial_cylinder>
58      </link>
59
60      <joint name="arm2_joint" type="revolute">
61          <origin xyz="0 0 ${arm1_length}" rpy="${-pi/2} 0 0"/>
62          <parent link="arm1_link"/>
63          <child link="arm2_link"/>
64          <axis xyz="0 0 1"/>
65          <limit lower="${-2*pi}" upper="${2*pi}" velocity="100" effort="100"
                />
66          <dynamics damping="10.0" friction="10.0"/>
67      </joint>
68
69      <link name="arm2_link">
70          <visual>
71              <origin xyz="0 0 ${arm2_length/2}" rpy="0 0 0"/>
72              <geometry>
73                  <cylinder length="${arm2_length}" radius="${arm2_radius}" />
74              </geometry>
75              <material name="${revolute_link_colour}"/>
```

```
76          </visual>
77          <collision>
78              <origin xyz="0 0 ${arm2_length/2}" rpy="0 0 0"/>
79              <geometry>
80                  <cylinder length="${arm2_length}" radius="${arm2_radius}" />
81              </geometry>
82          </collision>
83          <xacro:inertial_cylinder mass="1.0" length="${arm2_length}" radius="
                ${arm2_radius}">
84              <origin xyz="0 0 ${arm2_length/2}" rpy="0 0 0"/>
85          </xacro:inertial_cylinder>
86      </link>
87
88      <joint name="cylinder1_joint" type="fixed">
89          <origin xyz="0 0 ${arm2_length}" rpy="${pi/2} 0 0"/>
90          <parent link="arm2_link"/>
91          <child link="cylinder1_link"/>
92      </joint>
93
94      <link name="cylinder1_link">
95          <visual>
96              <origin xyz="0 0 ${cylinder1_length/2}" rpy="0 0 0"/>
97              <geometry>
98                  <cylinder length="${cylinder1_length}" radius="${
                        cylinder1_radius}" />
99              </geometry>
100             <material name="${fixed_link_colour}" />
101         </visual>
102         <collision>
103             <origin xyz="0 0 ${cylinder1_length/2}" rpy="0 0 0"/>
104             <geometry>
105                 <cylinder length="${cylinder1_length}" radius="${
                        cylinder1_radius}" />
106             </geometry>
107         </collision>
108         <xacro:inertial_cylinder mass="1.0" length="${cylinder1_length}"
                radius="${cylinder1_radius}">
109             <origin xyz="0 0 ${cylinder1_length/2}" rpy="0 0 0"/>
110         </xacro:inertial_cylinder>
111     </link>
112
113     <joint name="arm3_joint" type="revolute">
114         <origin xyz="0 0 ${cylinder1_length}" rpy="${pi/2} 0 0"/>
115         <parent link="cylinder1_link"/>
116         <child link="arm3_link"/>
117         <axis xyz="0 0 1"/>
118         <limit lower="${-2*pi}" upper="${2*pi}" velocity="100" effort="100"
                />
119         <dynamics damping="10.0" friction="10.0"/>
120     </joint>
121
122     <link name="arm3_link">
123         <visual>
124             <origin xyz="0 0 ${arm3_length/2}" rpy="0 0 0"/>
125             <geometry>
```

```
126                <cylinder length="${arm3_length}" radius="${arm3_radius}" />
127            </geometry>
128            <material name="${revolute_link_colour}"/>
129        </visual>
130        <collision>
131            <origin xyz="0 0 ${arm3_length/2}" rpy="0 0 0"/>
132            <geometry>
133                <cylinder length="${arm3_length}" radius="${arm3_radius}" />
134            </geometry>
135        </collision>
136        <xacro:inertial_cylinder mass="1.0" length="${arm3_length}" radius="
               ${arm3_radius}">
137            <origin xyz="0 0 ${arm3_length/2}" rpy="0 0 0"/>
138        </xacro:inertial_cylinder>
139    </link>
140
141    <joint name="cylinder2_joint" type="fixed">
142        <origin xyz="0 0 ${arm3_length}" rpy="0 ${-pi/2} 0"/>
143        <parent link="arm3_link"/>
144        <child link="cylinder2_link"/>
145    </joint>
146
147    <link name="cylinder2_link">
148        <visual>
149            <origin xyz="0 0 ${cylinder2_length/2}" rpy="0 0 0"/>
150            <geometry>
151                <cylinder length="${cylinder2_length}" radius="${
                       cylinder2_radius}" />
152            </geometry>
153            <material name="${fixed_link_colour}"/>
154        </visual>
155        <collision>
156            <origin xyz="0 0 ${cylinder2_length/2}" rpy="0 0 0"/>
157            <geometry>
158                <cylinder length="${cylinder2_length}" radius="${
                       cylinder2_radius}" />
159            </geometry>
160        </collision>
161        <xacro:inertial_cylinder mass="1.0" length="${cylinder2_length}"
               radius="${cylinder2_radius}">
162            <origin xyz="0 0 ${cylinder2_length/2}" rpy="0 0 0"/>
163        </xacro:inertial_cylinder>
164    </link>
165
166    <joint name="arm4_joint" type="revolute">
167        <origin xyz="0 0 ${cylinder2_length}" rpy="${-pi/2} 0 0"/>
168        <parent link="cylinder2_link"/>
169        <child link="arm4_link"/>
170        <axis xyz="0 0 1"/>
171        <limit lower="${-2*pi}" upper="${2*pi}" velocity="100" effort="100"
               />
172        <dynamics damping="10.0" friction="10.0"/>
173    </joint>
174
175    <link name="arm4_link">
```

```
176         <visual>
177             <origin xyz="0 0 ${arm4_length/2}" rpy="0 0 0"/>
178             <geometry>
179                 <cylinder length="${arm4_length}" radius="${arm4_radius}" />
180             </geometry>
181             <material name="${revolute_link_colour}"/>
182         </visual>
183         <collision>
184             <origin xyz="0 0 ${arm4_length/2}" rpy="0 0 0"/>
185             <geometry>
186                 <cylinder length="${arm4_length}" radius="${arm4_radius}" />
187             </geometry>
188         </collision>
189         <xacro:inertial_cylinder mass="1.0" length="${arm4_length}" radius="
                 ${arm4_radius}">
190             <origin xyz="0 0 ${arm4_length/2}" rpy="0 0 0"/>
191         </xacro:inertial_cylinder>
192     </link>
193
194 </robot>
```

Como se puede observar, todas las articulaciones (*joint*) tienen un eslabón agregado (*link*). En el *joint* se declaran los ejes de coordenadas relativos al anterior, haciendo uso de la geometría de este. En cuanto al *link*, dado que se dan de forma visual, se declaran con respecto a la geometría actual. De esta manera, se crea una descripción URDF genérica, dando la posibilidad de cambiar cualquier valor dentro del archivo de geometrías sin alterar la visualización final. El esquema orientativo del árbol final se representa en la Figura 2.12.

Figura 2.12: Esquemático del Árbol del Sistema

Explicada la descripción URDF, se pretende visualizar el robot construido.

CAPÍTULO 3

Visualización del Robot

Contenido

3.1. Herramienta de visualización: RVIZ **34**
3.1.1. Ejecución y configuración en RVIZ 34
3.1.2. Publicación de *topics* para la visualización en RVIZ 37
3.2. Resultados de visualización del Robot **38**
3.2.1. Rotación de *Arm* 1 y *Arm* 3 39
3.2.2. Rotación de *Arm* 2 y *Arm* 3 39
3.2.3. Rotación de todas las articulaciones 40
3.3. Directorios relacionados con la Visualización **40**
3.3.1. Directorio: *worlds* 40
3.3.2. Directorio de ejecución: launch 41

Una vez hecha la descripción del robot en URDF, se pretende visualizar dicha información. La etapa de visualización resulta crucial en el proceso de diseño y desarrollo del modelado del brazo robótico. Esta permite observar los fallos cometidos durante el modelado del mismo y ver cómo se comporta el modelo en un entorno virtual como RVIZ. Es esencial conocer y verificar visualmente la representación del robot para comprobar que esta sea precisa y consistente.

Mediante este apartado, se darán las herramientas para poder confirmar lo descrito en el apartado anterior, ayudando a verificar que las conexiones de *links* y *joints* se han realizado correctamente, que estos estén alineados de forma precisa. También es útil para depurar el diseño estructural en general, ayudando a identificar problemas geométricos.

Antes de pasar a simuladores más complejos, la visualización se considera una etapa de preparación, estableciendo una base sólida y resolviendo los problemas que podrían hacer la siguiente etapa más compleja y tediosa.

Como se ha nombrado, se trabajará en el entorno virtual de RVIZ.

3.1. Herramienta de visualización: RVIZ

Antes de explicar el resto de los directorios contenidos dentro del paquete urdf_pkg, se pretende dar un contexto acerca de la visualización.

En ROS 2 existe una herramienta que permite visualizar entornos en 3D. Dicha herramienta se conoce como RVIZ. Esta fue creada con la intención de representar datos relacionados con un robot en tiempo real, como su estado, entorno y sensores. También se utiliza para depurar, monitorizar y probar robots en un entorno de simulación o real. A la hora de crear un gemelo digital, RVIZ es una potente herramienta.

Para poder visualizar el robot creado en descripción URDF mediante RVIZ, hay varios caminos.

3.1.1. Ejecución y configuración en RVIZ

La primera opción es mediante la ejecución manual del programa RVIZ a través del comando:

```
ros2 run rviz2 rviz2
```

Se abre el entorno de RVIZ, el cual se muestra en la Figura 3.1

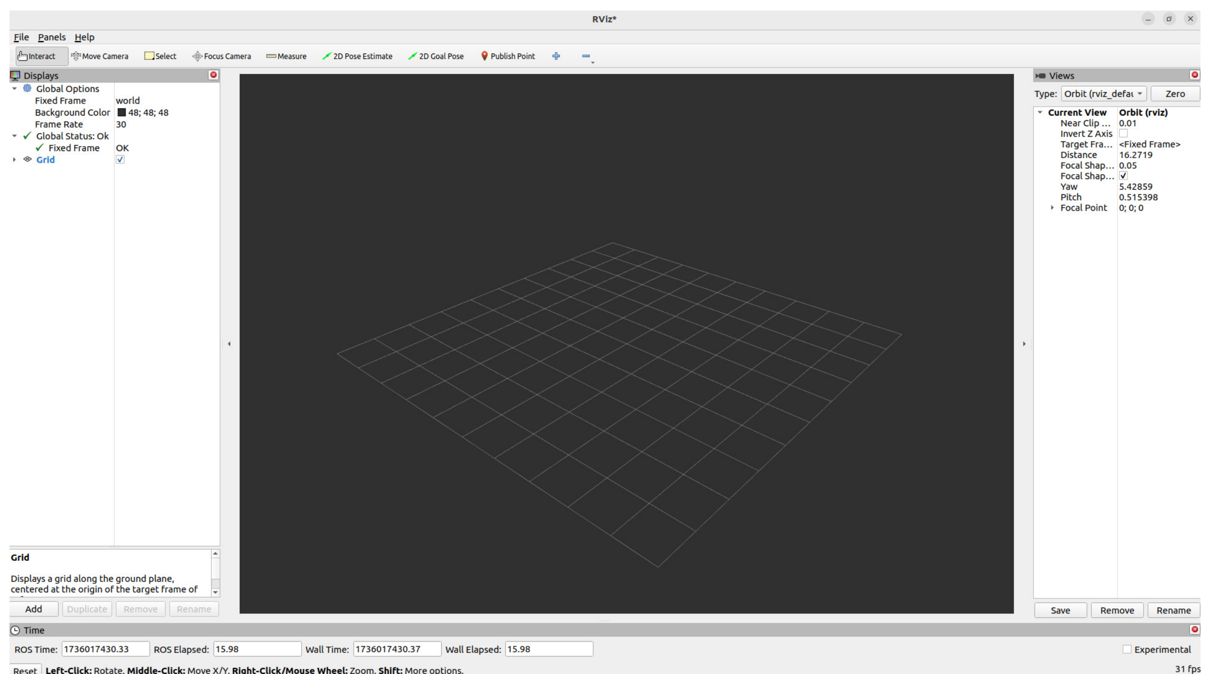

Figura 3.1: Entorno RVIZ

Dentro de este entorno, para el caso particular, se deben añadir tanto el modelo del robot como las transformaciones. De esta forma se visualizarán de forma práctica:

3.1.1.1. RobotModel

Para añadir el modelo del robot se tiene que pulsar en *Add* (Figura 3.2).

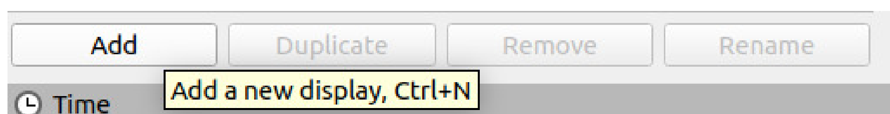

Figura 3.2: Botón Add en RVIZ

A continuación, dentro de la carpeta de rviz_default_plugins se encuentra el *RobotModel* (Figura 3.3).

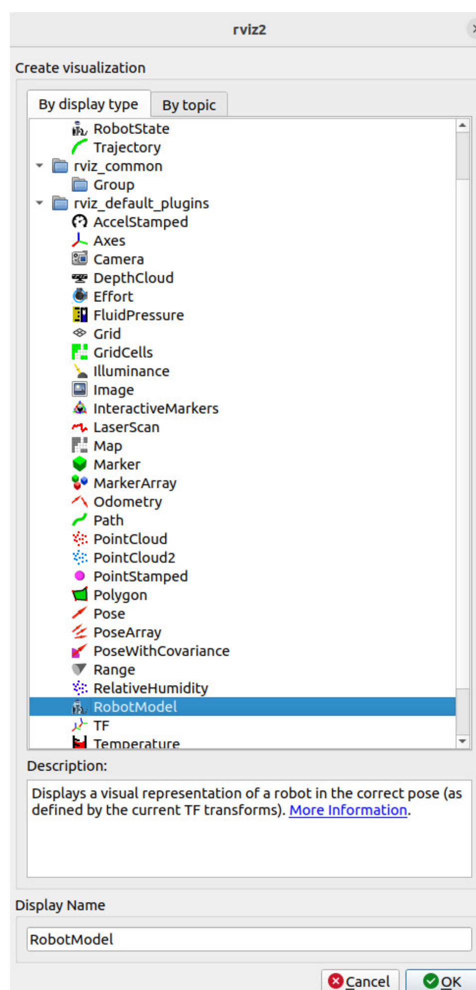

Figura 3.3: RobotModel como *plugin* de RVIZ

Una vez añadido, se configura el *topic* correspondiente, escribiendo robot_description, como se muestra en la Figura 3.4.

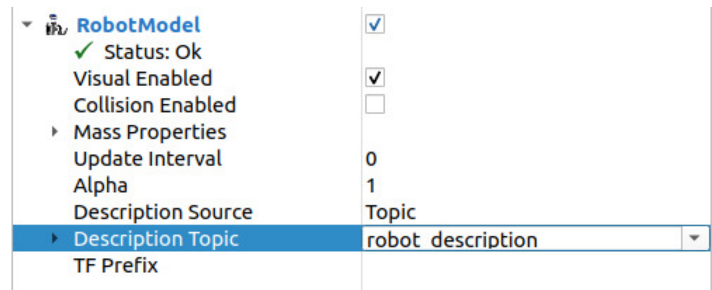

Figura 3.4: Topic de RobotModel en RVIZ

Finalmente, se obtiene la Figura 3.5 referente al robot con sus geometrías caídas sobre el eje (0,0,0). Esto se debe a que aún no hay nada haciendo publicaciones en el *topic*.

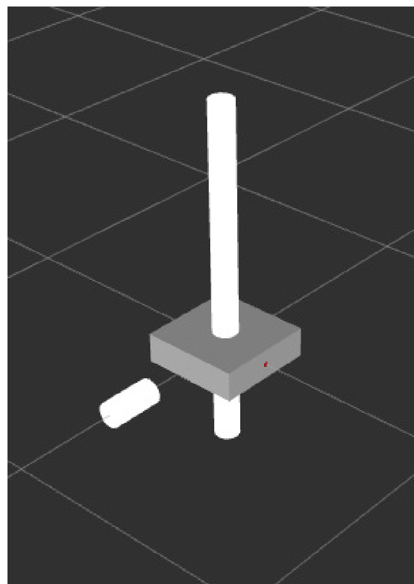

Figura 3.5: Visualización del Robot en RVIZ sin publicaciones al topic

3.1.1.2. Transformaciones (TF)

Es necesario añadir las transformaciones, que permiten visualizar los ejes de coordenadas definidos para cada *joint*. Es crucial que estos ejes se hayan definido correctamente, ya que, de no ser así, el robot se visualizará de forma imprecisa.

La visualización de estos ejes es el equivalente a la construcción de la primera etapa cuando se calcula una cinemática directa, aunque la ventaja que el entorno de ROS 2 ofrece es que no será necesario calcular dicha cinemática.

Se añade de la misma manera que *RobotModel*, aunque esta vez no será necesario añadir ninguna configuración extra.

3.1.2. Publicación de *topics* para la visualización en RVIZ

A la hora de darle valores a las articulaciones, hay dos métodos prácticos que son rápidos y eficaces para visualizar el estado de las mismas:

3.1.2.1. Publicación Manual del *Topic* joint_states

El estado de las articulaciones se puede publicar de forma manual sobre el *topic* `/joint_states`, que es de tipo `sensor_msgs/msg/JointState`. Se puede encontrar la información de este *topic* y cómo configurar cada una de estas articulaciones en la página oficial de ROS.

3.1.2.2. Uso de la interfaz gráfica joint_state_publisher_gui

Es posible modificar el valor de rotación de las articulaciones mediante una interfaz gráfica, lanzando el comando:

```
ros2 run joint_state_publisher_gui joint_state_publisher_gui
```

Este comando ejecuta una interfaz gráfica para interactuar de forma manual con las posiciones de las articulaciones. Se visualizan los *slicers* correspondientes a cada una de las partes del robot. Cada *slicer* tiene la posibilidad de interactuar hasta donde se definieron los límites, mostrado en la Figura 3.6.

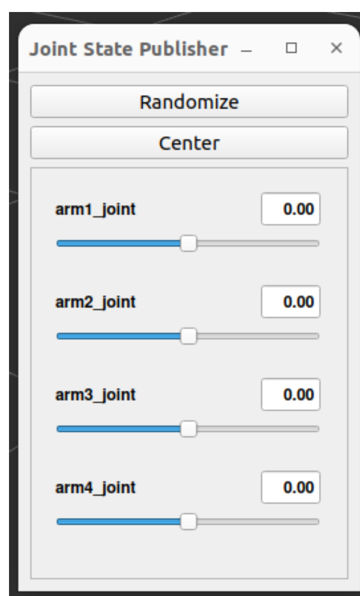

Figura 3.6: joint_state_publisher_gui

Esta herramienta resulta muy útil para la obtención de configuraciones articulares que pueden ser necesarias cuando se definen trayectorias.

3.2. Resultados de visualización del Robot

El resultado final de visualización al ejecutar el comando de joint_state_publisher_gui queda mostrado en la Figura 3.7.

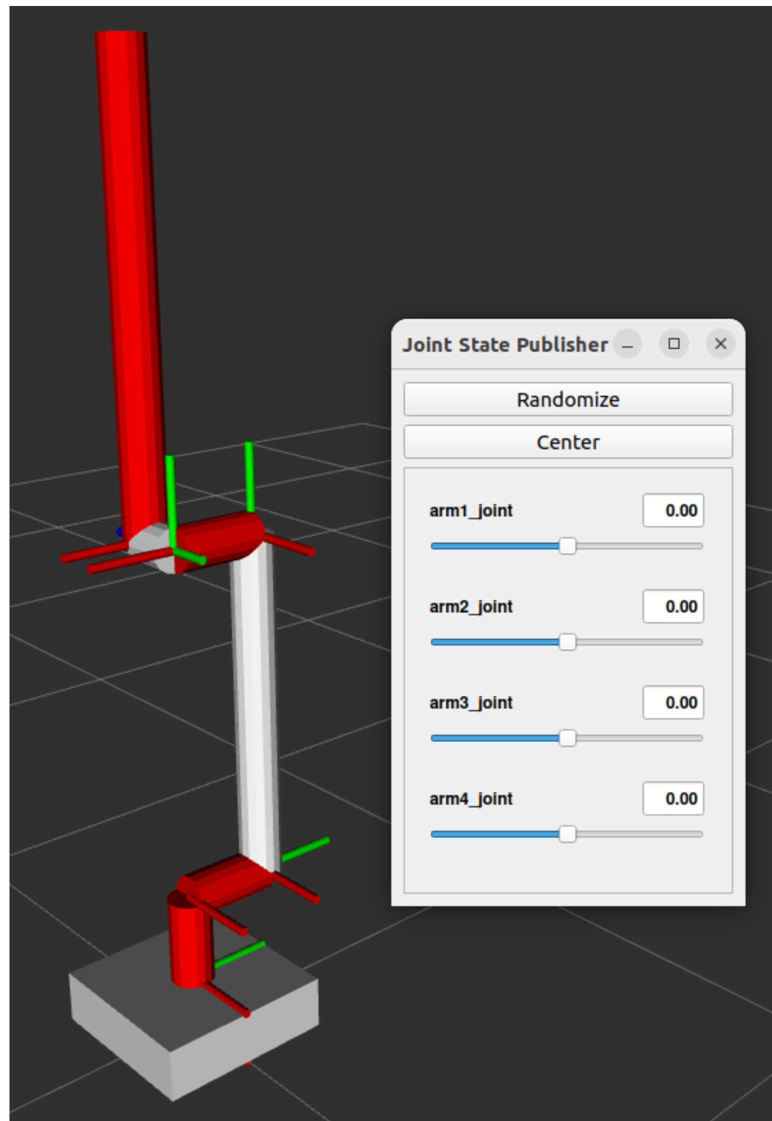

Figura 3.7: Visualización del Robot en RVIZ

Se puede observar la configuración inicial escogida. Se elige una configuración vertical, dado que es la más estándar y permite alcanzar y realizar movimientos de forma más sencilla.

Seguidamente, se observan las distintas poses dependiendo de la configuración del grado al que se encuentra cada articulación (Figuras 3.8, 3.9 y 3.10)

3.2.1. Rotación de *Arm* 1 y *Arm* 3

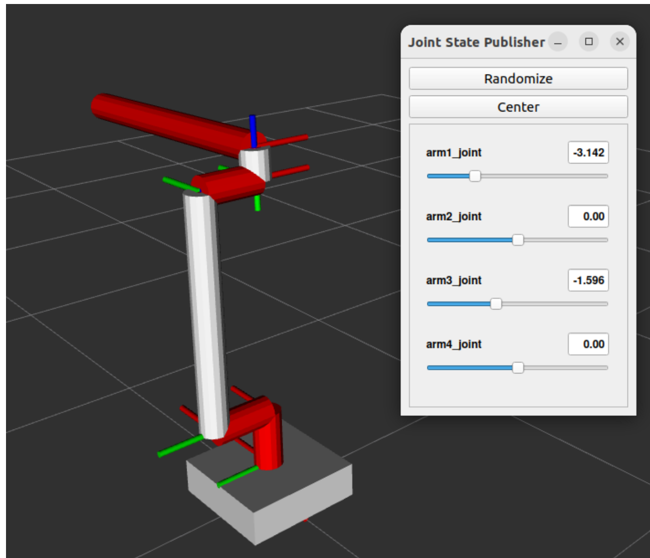

Figura 3.8: Rotación de *Arm* 1 y *Arm* 3

3.2.2. Rotación de *Arm* 2 y *Arm* 3

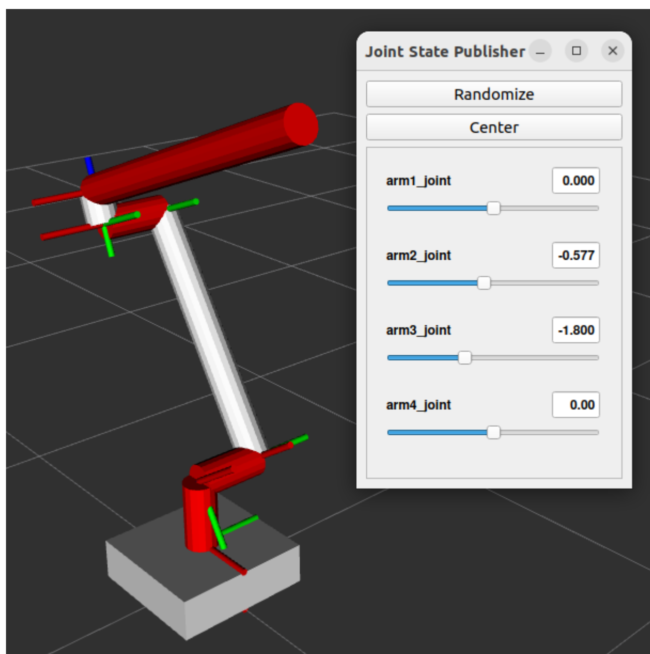

Figura 3.9: Rotación de *Arm* 2 y *Arm* 3

3.2.3. Rotación de todas las articulaciones

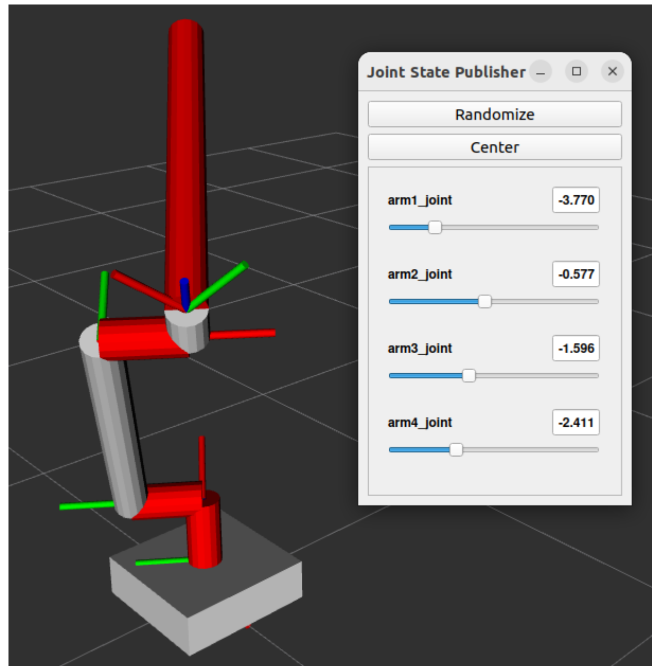

Figura 3.10: Rotación de todas las articulaciones

De esta forma, se puede visualizar la descripción URDF del robot. Esto permite hacer la validación del modelo visual, ayudando en la detección de errores geométricos, desalineación entre eslabones y articulaciones o escalas incorrectas. También permite verificar las relaciones cinemáticas, hacer simulaciones sin necesidad de *hardware*, integrar sensores y controladores, preparar la simulación ante simuladores más complejos o hacer ajustes iterativos del modelo.

3.3. Directorios relacionados con la Visualización

Finalmente, se crean dos directorios dentro del paquete urdf_pkg. Dichos directorios están relacionados con la visualización obtenida en RVIZ, o servirán para acceder a esta herramienta de una forma más dinámica.

3.3.1. Directorio: *worlds*

El entorno configurado con los parámetros del robot y las transformaciones se puede guardar como un archivo con la extensión .rviz. Dicho archivo se denomina como un *world*.

Este directorio almacena los archivos de configuración RVIZ (.rviz) en caso de que se requiera la ejecución del entorno de visualización sin necesidad de un *launch*.

3.3.2. Directorio de ejecución: launch

También se puede crear un archivo *launch* que permita la publicación de ciertos *topics*. RVIZ se suscribirá a dichos *topics* y representará en 3D el modelo del robot, así como su movimiento.

Dentro del paquete urdf_pkg se creará un directorio launch donde se almacenarán los ejecutables de este tipo. Los archivos *launch* son *script* basados en Python utilizados para definir y ejecutar múltiples nodos, configuraciones y parámetros con tan solo un comando. En este caso, se programa el archivo urdf.launch.py y su ejecución se hace de la siguiente forma:

```
ros2 launch urdf_pkg urdf.launch.py
```

Este archivo en concreto contiene lo siguiente:

- **Procesamiento del archivo de descripción URDF**: Se declara el nombre del paquete donde se encuentra el archivo que se va a utilizar, en este caso la descripción URDF, y se procesa generando una cadena del contenido.
- **Nodo robot_state_publisher**: Se configura el nodo que publica la descripción URDF para poder tratar la información desde otros nodos.
- **Nodo RVIZ**: Se configura el nodo que abre y lanza el ejecutable de RVIZ.

Mediante un comando final, se encapsulan los nodos a ejecutar y se lanzan. El código completo de este archivo se declara a continuación:

Código 3.1: urdf.launch.py

```
1  import os
2  from ament_index_python.packages import get_package_share_directory
3  from launch import LaunchDescription
4  from launch_ros.actions import Node
5  import xacro
6
7
8  def generate_launch_description():
9
10     # Specify the name of the package and path to xacro file within the
           package
11     pkg_name = 'urdf_pkg'
12     file_subpath = 'description/robot.urdf.xacro'
13
14     # Use xacro to process the file
15     xacro_file = os.path.join(get_package_share_directory(pkg_name),
           file_subpath)
16     robot_description_raw = xacro.process_file(xacro_file).toxml()
17
18     # Configure the robot_state_publisher node
19     node_robot_state_publisher = Node(
20         package='robot_state_publisher',
```

```
21          executable='robot_state_publisher',
22          output='screen',
23          parameters=[{'robot_description': robot_description_raw}]
24      )
25
26      # Path to RViz configuration file
27      rviz_config_file = os.path.join(get_package_share_directory(pkg_name), "
            worlds", "view_robot.rviz")
28
29      # Configure the RViz node
30      node_rviz = Node(
31          package='rviz2',
32          executable='rviz2',
33          name='rviz2',
34          output='screen',
35          arguments=['-d', rviz_config_file]
36      )
37
38      # Run both nodes
39      return LaunchDescription([
40          node_robot_state_publisher,
41          node_rviz
42      ])
```

Cuando este archivo se ejecuta, se observa lo mostrado en la Figura 3.5. Una vez publicada una pose para el robot, se vería como en Figura 3.7.

CAPÍTULO 4

Simulación del Robot

Contenido

4.1. Directorios relacionados con la Simulación **43**
4.1.1. Directorio: description 44
4.1.2. Directorio de ejecución: launch 46
4.2. Resultados de simulación del Robot **49**

Para llevar a cabo la simulación en Gazebo, es necesario instalar la herramienta desde la página oficial. Esto permite abrirlo como si se tratase de un programa independiente, al contrario de RVIZ, que se ejecuta directamente desde ROS 2. Esto se debe a que RVIZ está diseñada como un nodo de ROS 2, mientras que Gazebo es un simulador independiente que trabaja con ROS.

Al igual que se explicó en el apartado de **Descripción URDF**, es conveniente crear un paquete específico para llevar a cabo las operaciones de simulación. También se hace uso del paquete ya creado **urdf_pkg**, de tal forma que cualquier cambio en dicha descripción y visualización se refleje de forma automática sobre la simulación.

4.1. Directorios relacionados con la Simulación

Para comenzar la simulación, se crea un nuevo paquete llamado sim_pkg. Éste tendrá una estructura similar al anterior. Los directorios incluidos dentro de este paquete son los siguientes:

- config

- description
- include
- launch
- src

Los directorios src e *include*, en este caso, están vacíos. Es decir, no es necesario crear ningún nodo adicional para la simulación. Al menos en este caso particular, ya que, como se muestra en próximos apartados, si se requiere de un control complejo, sí será necesario hacer las correspondientes librerías y archivos de ejecución.

Para el apartado de simulación se hará uso de dos directorios: *description* y *launch*.

4.1.1. Directorio: description

En este directorio se añaden nuevos archivos tipo **.xacro** que serán necesarios para simular el robot en Gazebo. Para la parte de simulación son necesarios los siguientes:

4.1.1.1. robot.sim.xacro

Este archivo es el global. Se usa para crear el robot con la parte de simulación añadida, darle nombre y para incluir al resto de paquetes. De esta forma, resulta más sencilla la adición o eliminación de los otros archivos. Es exactamente igual que el archivo **robot.urdf.xacro** pero se añade lo específico de Gazebo. Su contenido es el siguiente:

Código 4.1: robot.sim.xacro

```
1  <?xml version="1.0"?>
2  <robot xmlns:xacro="http://www.ros.org/wiki/xacro" name="robot">
3
4      <xacro:include filename="$(find urdf_pkg)/description/inertial_macros.
           xacro" />
5      <xacro:include filename="$(find urdf_pkg)/description/geometry_values.
           xacro" />
6      <xacro:include filename="$(find urdf_pkg)/description/urdf_colours.xacro
           " />
7      <xacro:include filename="$(find urdf_pkg)/description/
           urdf_robot_description.xacro" />
8      <xacro:include filename="$(find sim_pkg)/description/robot_gazebo.xacro"
            />
9
10 </robot>
```

4.1.1.2. robot_gazebo.xacro

En este archivo, se declaran las propiedades de cada articulación para que se vean iguales que en la visualización. Se declaran los colores de Gazebo y se añade el **plugin** necesario para la publicación de las articulaciones en Gazebo: **gazebo_ros_joint_state_publisher**

Código 4.2: robot_gazebo.xacro

```xml
<?xml version="1.0"?>
<robot xmlns:xacro="http://www.ros.org/wiki/xacro">

    <gazebo reference="base_link">
        <material>Gazebo/White</material>
    </gazebo>

    <gazebo reference="arm1_link">
        <material>Gazebo/Red</material>
    </gazebo>

    <gazebo reference="arm2_link">
        <material>Gazebo/Red</material>
    </gazebo>

    <gazebo reference="cylinder1_link">
        <material>Gazebo/White</material>
    </gazebo>

    <gazebo reference="arm3_link">
        <material>Gazebo/Red</material>
    </gazebo>

    <gazebo reference="cylinder2_link">
        <material>Gazebo/White</material>
    </gazebo>

    <gazebo reference="arm4_link">
        <material>Gazebo/Red</material>
    </gazebo>

    <gazebo>
        <plugin name="gazebo_ros_joint_state_publisher"
            filename="libgazebo_ros_joint_state_publisher.so">
            <update_rate>20</update_rate>
            <joint_name>base_joint</joint_name>
            <joint_name>arm1_joint</joint_name>
            <joint_name>arm2_joint</joint_name>
            <joint_name>arm3_joint</joint_name>
            <joint_name>cylinder1_joint</joint_name>
            <joint_name>cylinder2_joint</joint_name>
            <joint_name>arm4_joint</joint_name>
        </plugin>
    </gazebo>

</robot>
```

Añadidas las descripciones en formato xml necesarias para la ejecución de Gazebo de forma correcta, se crea el archivo launch que configura el entorno y lanza los nodos correspondientes.

4.1.2. Directorio de ejecución: launch

Se crea el directorio **launch** al igual que se hizo en el paquete **urdf_pkg**. Este directorio contiene todos los ejecutables relacionados con la simulación. Serán necesarios tres archivos de ejecución para lanzar Gazebo. A pesar de que se podrían unir en un solo archivo, se ha decidido hacer de esta forma por temas de organización y dado que dos de ellos son más genéricos y pueden ser usados por otros ejecutables. Los archivos son:

4.1.2.1. description.launch.py

Este archivo crea formatos para el uso de funciones repetitivas dentro de un archivo de ejecución. También declara el contenido del nodo **robot_state_publisher_node**.

4.1.2.2. gazebo.launch.py

Mediante este ejecutable se gestiona el lanzamiento de Gazebo, se configura su entorno creando el mundo y se hace el spawn del robot. Esto se refiere a cargar el modelo del robot en el entorno simulado. También se añade la ejecución del archivo anterior para poder formatear como se ha descrito. El archivo se muestra a continuación:

Código 4.3: gazebo.launch.py

```
1  from launch import LaunchDescription
2  from launch.actions import DeclareLaunchArgument
3  from launch.actions import IncludeLaunchDescription
4  from launch.substitutions import LaunchConfiguration, PathJoinSubstitution
5  from launch_ros.actions import Node
6  from launch_ros.substitutions import FindPackageShare
7
8
9  def generate_launch_description():
10
11     gui_arg = DeclareLaunchArgument(
12         name='gui',
13         default_value='true',
14     )
15     package_arg = DeclareLaunchArgument('urdf_package',
16                                         description='The package where the
                                               robot description is located',
17                                         default_value='urdf_pkg')
18     model_arg = DeclareLaunchArgument('urdf_package_path',
19                                       description='The path to the robot
                                             description relative to the
                                             package root',
20                                       default_value='description/robot.urdf.
                                             xacro')
```

```
21
22      empty_world_launch = IncludeLaunchDescription(
23          PathJoinSubstitution([FindPackageShare('gazebo_ros'), 'launch', '
                gazebo.launch.py']),
24          launch_arguments={
25              'gui': LaunchConfiguration('gui'),
26              'pause': 'true',
27          }.items(),
28      )
29
30      description_launch_py = IncludeLaunchDescription(
31          PathJoinSubstitution([FindPackageShare('sim_pkg'), 'launch', '
                description.launch.py']),
32          launch_arguments={
33              'urdf_package': LaunchConfiguration('urdf_package'),
34              'urdf_package_path': LaunchConfiguration('urdf_package_path')}.
                    items()
35      )
36
37      urdf_spawner_node = Node(
38          package='gazebo_ros',
39          executable='spawn_entity.py',
40          name='urdf_spawner',
41          arguments=['-topic', '/robot_description', '-entity', 'robot', '-
                unpause'],
42          output='screen',
43      )
44
45      return LaunchDescription([
46          gui_arg,
47          package_arg,
48          model_arg,
49          empty_world_launch,
50          description_launch_py,
51          urdf_spawner_node,
52      ])
```

4.1.2.3. sim.launch.py

Por último, se crea el archivo de ejecución que incluye la configuración y lanzamiento de RVIZ y la configuración y lanzamiento del archivo anterior de ejecución de Gazebo, todo ello mediante el formato dado en el Código 3.1.

Código 4.4: sim.launch.py

```
1 from launch import LaunchDescription
2 from launch.actions import DeclareLaunchArgument
3 from launch.actions import IncludeLaunchDescription
4 from launch.substitutions import LaunchConfiguration, PathJoinSubstitution
5 from launch_ros.actions import Node
6 from launch_ros.substitutions import FindPackageShare
7
8 def generate_launch_description():
```

```
9
10   # Specify the name of the package and path to xacro file within the
         package
11   package_arg = DeclareLaunchArgument('urdf_package',
12                                       description='The package where the
                                             robot description is located',
13                                       default_value='sim_pkg')
14   model_arg = DeclareLaunchArgument('urdf_package_path',
15                                     description='The path to the robot
                                           description relative to the
                                           package root',
16                                     default_value='description/robot.sim.
                                           xacro')
17
18   # Path to RViz configuration file
19   rvizconfig_arg = DeclareLaunchArgument(
20       name='rvizconfig',
21       default_value=PathJoinSubstitution([FindPackageShare('urdf_pkg'), '
             worlds', 'view_robot.rviz']),
22   )
23
24   # Include the Gazebo launch file, provided by the gazebo_ros package
25   gazebo_launch = IncludeLaunchDescription(
26       PathJoinSubstitution([FindPackageShare('sim_pkg'), 'launch', 'gazebo
             .launch.py']),
27       launch_arguments={
28           'urdf_package': LaunchConfiguration('urdf_package'),
29           'urdf_package_path': LaunchConfiguration('urdf_package_path')
30       }.items(),
31   )
32
33   # Configure the RViz node
34   rviz_node = Node(
35       package='rviz2',
36       executable='rviz2',
37       output='screen',
38       arguments=['-d', LaunchConfiguration('rvizconfig')],
39   )
40
41   # Run all nodes
42   return LaunchDescription([
43       package_arg,
44       model_arg,
45       rvizconfig_arg,
46       gazebo_launch,
47       rviz_node,
48   ])
```

Mediante la ejecución de este último, se abrirá Gazebo y RVIZ para obtener la simulación y visualización de la misma en tiempo real.

4.2. Resultados de simulación del Robot

Una vez se lanza el launch: **ros2 launch sim_pkg sim.launch.py** se visualiza el robot en simulación:

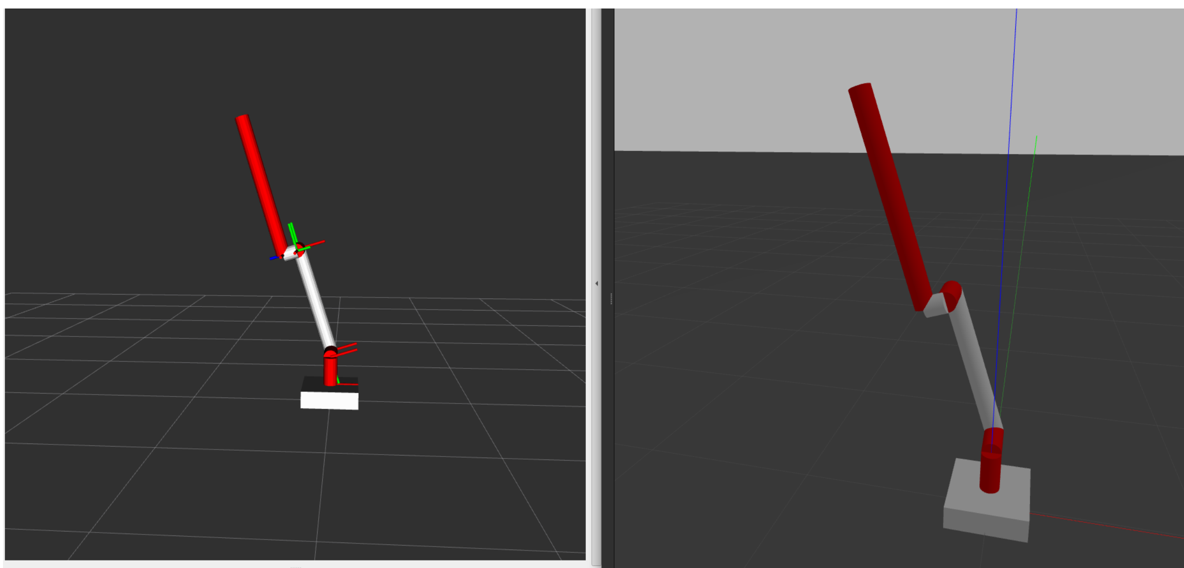

Figura 4.1: Robot simulado en Gazebo al inicio

Sin embargo, al cabo de unos segundos ocurre lo siguiente:

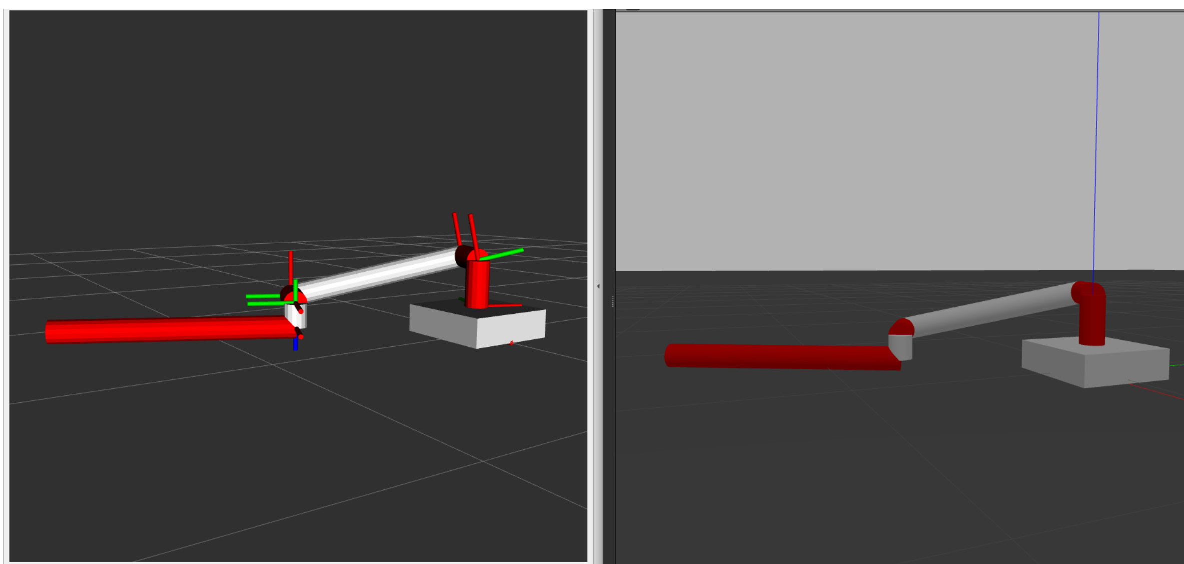

Figura 4.2: Robot simulazo en Gazebo pasados unos segundos

El robot se simula haciendo uso de los valores de masa, y la gravedad tiene influencia sobre él. Esto hace que, tras un tiempo, el robot caiga. Esto se debe a la falta de controladores en el robot. Ninguna de las articulaciones está siendo simulada con un control de ningún tipo. Se puede entender como

si los motores estuviesen completamente apagados, o como si no hubiera motores, ni fricción, ni *dumping*, ni ningún tipo de variable que impida que el robot caiga. Para evitar que esto ocurra, debe hacerse algún tipo de control que permita publicar valores sobre las articulaciones.

CAPÍTULO 5

Control de Posición del Robot

Contenido

5.1. Control de Posición mediante ROS 2 Control y Gazebo **52**
5.1.1. ROS 2 Control 52
5.2. Directorios relacionados con el Control de Posición **52**
5.2.1. Directorio: description 53
5.2.2. Directorio de ejecución: launch 54
5.2.3. Directorio de ejecución: config 55
5.3. Resultados del Control de Posición **56**
5.3.1. Robot en posición inicial 57
5.3.2. Robot en Pose 1 57
5.3.3. Robot en Pose 2 58
5.3.4. Robot en Pose 3 59

Como se pudo ver en el apartado anterior, la simulación mediante Gazebo permite visualizar el robot con las físicas consideradas y dinámicas propias del robot. Sin embargo, el robot carece de controladores que permitan una estabilización del mismo y un uso práctico. Incluyendo la herramienta de ROS 2 Control es posible realizar el control del robot.

5.1. Control de Posición mediante ROS 2 Control y Gazebo

5.1.1. ROS 2 Control

ROS 2 Control es un framework para el control en tiempo real de robots haciendo uso de ROS 2. Es una adaptación de los paquetes incluidos en ros_control de ROS, con el objetivo de simplificar la integración de nuevo hardware y superar las limitaciones del software anterior.

5.1.1.1. Repositorios Principales de ROS 2 Control

Existen una serie de repositorios que serán necesarios para la implementación de los sistemas de control incluidos en este framework:

- **ros2_control**: Interfaces y componentes principales.
- **ros2_controllers**: Controladores comunes.
- **control_toolbox**: Implementación de atributos caracteristicos de la teoría de control (como los PID).
- **realtime_tools**: Herramientas para el soporte en tiempo real (buffers y publicadores).
- **control_msgs**: Tratamiento de los datos para controladores.
- **kinematics_interface**: Interfaz para frameworks cinemáticos en C++.

Todos los paquetes mostrados han sido utilizados para la implementación del sistema creado. Además, existen otros paquetes adicionales que pueden facilitar el aprendizaje (como el de ros2_control_demos) o que cuentan con documentos de planificación y diseño del proyecto (roadmap).

5.2. Directorios relacionados con el Control de Posición

El uso de la herramienta Gazebo es igual que en el apartado anterior y no es necesario realizar ninguna modificación dentro de este para realizar el control de posición. Para poder operar sobre el robot y realizar dicho control y su correspondiente simulación en tiempo real, se retoma la explicación de los directorios contenidos dentro del paquete sim_pkg.

Anteriormente, se comentó lo contenido dentro de los directorios *description* y *launch*, aunque a estos se les añaden los parámetros y archivos necesarios para su control. También se configura el archivo de configuración de parámetros dentro del directorio *config*.

5.2.1. Directorio: description

En este directorio, están todas las descripciones URDF del robot. Hasta ahora no se ha introducido nada relacionado con los repositorios de ROS 2 Control, pero en este momento se añaden ciertas modificaciones. Se crea un nuevo archivo .xacro para añadir las modificaciones pertinentes.

5.2.1.1. robot.control.xacro

En este archivo se añaden modificaciones para el control del robot.

- **Definición del sistema *hardware***: Se define un sistema *hardware* con el nombre de **GazeboSystem** y un *plugin* que permite a Gazebo interpretar comandos de control y publicar estados en las articulaciones en tiempo real. Dicho *plugin* es el de gazebo_ros2_control/GazeboSystem.
- **Definición de las interfaces de comandos y estados del control**: Para cada articulación que se desea controlar, se definen interfaces de comandos y estados para controlar y obtener información de la posición y velocidad de las mismas.
- **Plugin de Gazebo**: Define un *plugin* para Gazebo que conecta esta herramienta con ros2_control y especifica la ruta en la cual se encuentran los parámetros de los controladores.

El código interno se muestra a continuación:

Código 5.1: robot.control.xacro

```
1  <?xml version="1.0"?>
2  <robot xmlns:xacro="http://www.ros.org/wiki/xacro" name="robot">
3
4      <!-- Inclusiones de otros archivos Xacro -->
5      <xacro:include filename="$(find urdf_pkg)/description/inertial_macros.
           xacro" />
6      <xacro:include filename="$(find urdf_pkg)/description/geometry_values.
           xacro" />
7      <xacro:include filename="$(find urdf_pkg)/description/urdf_colours.xacro
           " />
8      <xacro:include filename="$(find urdf_pkg)/description/
           urdf_robot_description.xacro" />
9      <xacro:include filename="$(find sim_pkg)/description/robot_gazebo.xacro"
            />
10
11     <!-- Control del robot -->
12     <ros2_control name="GazeboSystem" type="system">
13         <hardware>
14             <plugin>gazebo_ros2_control/GazeboSystem</plugin>
15         </hardware>
16
17         <!-- Definici n de articulaciones -->
18         <joint name="arm1_joint">
19             <command_interface name="position" />
20             <command_interface name="velocity" />
21             <state_interface name="position"/>
```

```
22            <state_interface name="velocity"/>
23        </joint>
24        <joint name="arm2_joint">
25            <command_interface name="position" />
26            <command_interface name="velocity" />
27            <state_interface name="position"/>
28            <state_interface name="velocity"/>
29        </joint>
30        <joint name="arm3_joint">
31            <command_interface name="position" />
32            <command_interface name="velocity" />
33            <state_interface name="position"/>
34            <state_interface name="velocity"/>
35        </joint>
36        <joint name="arm4_joint">
37            <command_interface name="position" />
38            <command_interface name="velocity" />
39            <state_interface name="position"/>
40            <state_interface name="velocity"/>
41        </joint>
42    </ros2_control>
43
44    <!-- Plugin de Gazebo -->
45    <gazebo>
46        <plugin filename="libgazebo_ros2_control.so" name="
              gazebo_ros2_control">
47            <parameters>$(find sim_pkg)/config/joints.yaml</parameters>
48        </plugin>
49    </gazebo>
50
51 </robot>
```

5.2.2. Directorio de ejecución: launch

En este directorio se añade el archivo correspondiente a la ejecución del control en Gazebo. No serán necesarios más archivos.

5.2.2.1. control.launch.py

En este *script* de ejecución se introducen los mismos parámetros que se introdujeron en el Código 4.4, aunque se le añade la configuración de los nodos de los controladores. Hay dos controladores:

- **joint_state_broadcaster**: Este controlador publica los estados de las articulaciones en el topic `/joint_states`.

- **joint_controller**: Este controlador envía comandos a las articulaciones en los términos configurados. En este caso envía posiciones está configurado para enviar posiciones, como se verá más adelante.

La adición de estos controladores al código es la siguiente.

Código 5.2: control.launch.py

```
# Configure controller nodes
load_joint_state_controller = ExecuteProcess(
    cmd=['ros2', 'control', 'load_controller', '--set-state', 'active',
         'joint_state_broadcaster'],
    output='screen'
)

load_joint_controller = ExecuteProcess(
    cmd=['ros2', 'control', 'load_controller', '--set-state', 'active',
         'joint_controller'],
    output='screen'
)
```

También se debe tener en cuenta que ambos se deben añadir al **return**

5.2.3. Directorio de ejecución: config

El directorio *config* se utiliza en ROS 2 para almacenar archivos de configuración que definen parámetros, ajustes o configuraciones específicas que los nodos o sistemas utilizan durante su ejecución. Estos archivos ayudan a personalizar y parametrizar el comportamiento de los nodos sin necesidad de modificar el código fuente, permitiendo así una mayor flexibilidad y personalización del código.

En este caso, tan solo es necesario un archivo de configuración.

5.2.3.1. joints.yaml

El formato .yaml es un formato de serialización de datos muy utilizado en diversas aplicaciones, dado que es fácil de leer y escribir. Se creó para que fuese fácilmente entendible por los humanos.

En concreto, este archivo ha sido creado para definir los controladores:

- **controller_manager**: El es responsable de administrar los controladores y especificar la frecuencia de actualización de dichos controladores, así como indicar el uso de tiempo (si se utiliza tiempo simulado o real).
- **joint_state_broadcaster**: Queda definido el controlador anteriormente nombrado.
- **joint_controller**: Se define y, además, se establecen sus parámetros, como son las articulaciones y la interfaz de control que se va a usar (en cuyo caso sería la posición).

Internamente, este archivo contiene el siguiente código:

Código 5.3: joints.yaml

```
controller_manager:
  ros__parameters:
    update_rate: 100
    use_sim_time: true
```

```
5
6       joint_state_broadcaster:
7         type: joint_state_broadcaster/JointStateBroadcaster
8
9       joint_controller:
10        type: position_controllers/JointGroupPositionController
11
12  joint_controller:
13    ros__parameters:
14      joints:
15      - arm1_joint
16      - arm2_joint
17      - arm3_joint
18      - arm4_joint
19      interface_name: position
```

5.3. Resultados del Control de Posición

Para llevar a cabo la correcta ejecución del archivo *launch*, se deben haber compilado primero todos los paquetes incluidos en ROS 2 Control. Una vez hecho eso, se ejecuta el *launch* con el comando:

```
ros2 launch sim_pkg control.launch.py
```

Para controlar y visualizar el estado de las articulaciones se hace uso de dos comandos:

- Muestra el estado de las articulaciones publicadas en el *topic*.

```
ros2 topic echo joint_states
```

- Publica la posición establecida en radianes dentro del array data:

```
ros2 topic pub /joint_controller/commands std_msgs/msg/Float64MultiArray
"data: [arm1_position, arm2_position, arm3_position, arm4_position]"
```

Los resultados se muestran a continuación:

5.3.1. Robot en posición inicial

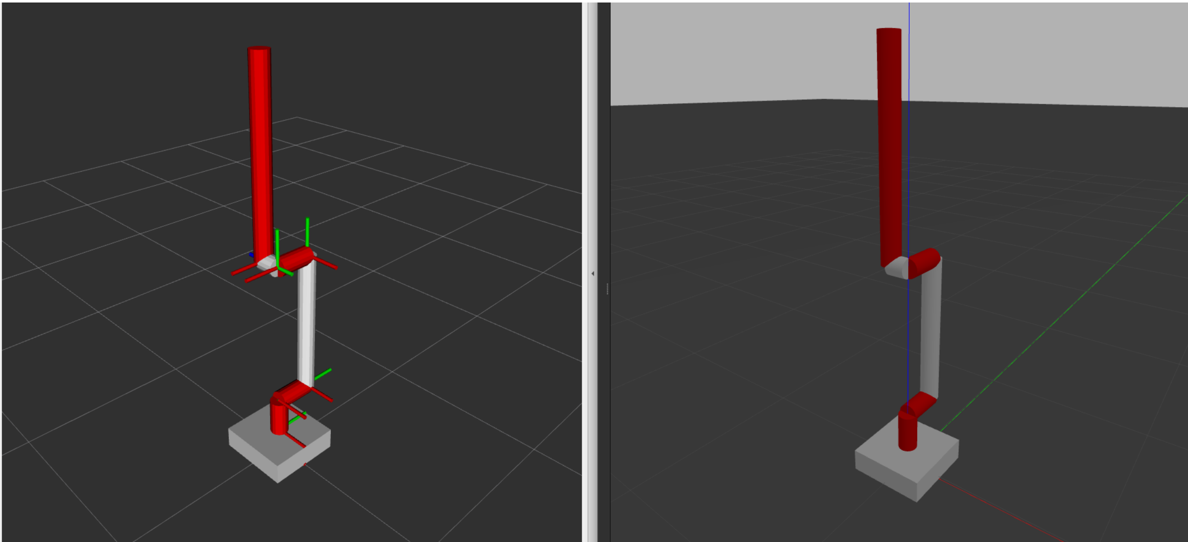

Figura 5.1: Robot en posición inicial simulado con Gazebo y Control de Posición

5.3.2. Robot en Pose 1

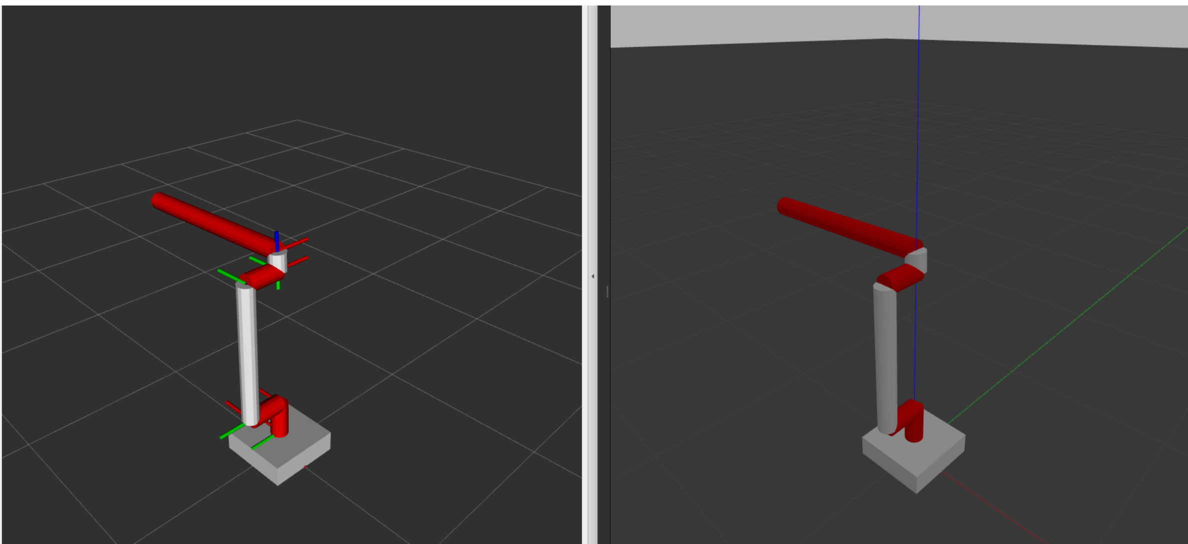

Figura 5.2: Robot en Pose 1 simulado con Gazebo y Control de Posición

```
paskuich@paskuich-GV62-7RE:~/control_ws$ ros2 topic pub /joint_controller/commands std_msgs/msg/Float64
MultiArray "data: [-3.142, 0.0, -1.596, 0.0]"
publisher: beginning loop
publishing #1: std_msgs.msg.Float64MultiArray(layout=std_msgs.msg.MultiArrayLayout(dim=[], data_offset=
0), data=[-3.142, 0.0, -1.596, 0.0])
```

Figura 5.3: Plot del control: Pose 1

```
header:
  stamp:
    sec: 2107
    nanosec: 520000000
  frame_id: ''
name:
- arm1_joint
- arm2_joint
- arm3_joint
- arm4_joint
position:
- -3.142000967324743
- -4.5339154620016586e-05
- -1.5959975231214414
- -5.1698513346742914e-05
```

Figura 5.4: Visualización del contenido del topic joint_states: Pose 1

5.3.3. Robot en Pose 2

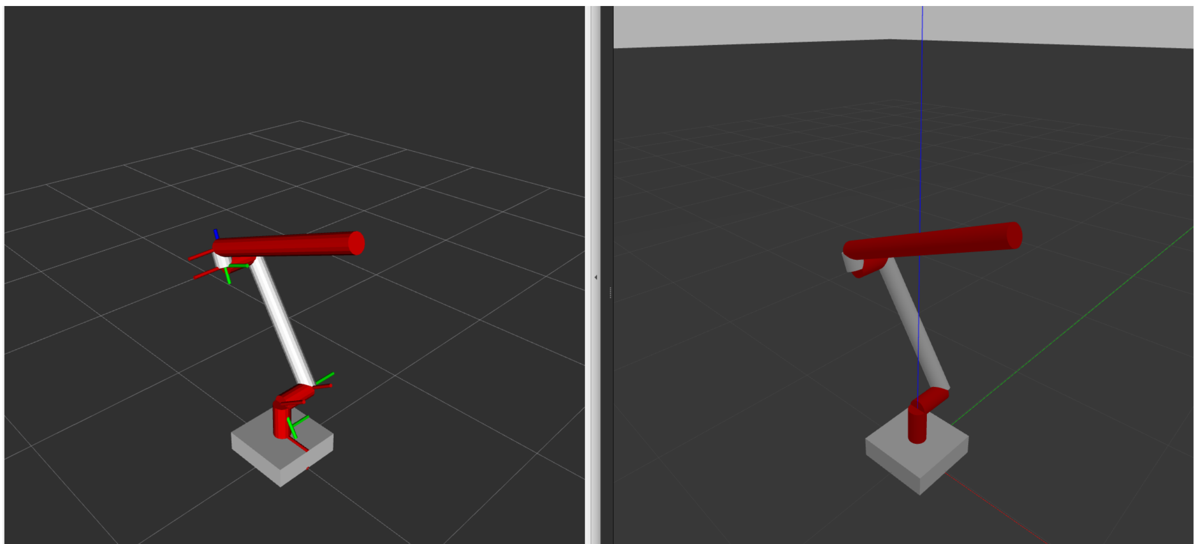

Figura 5.5: Robot en Pose 2 simulado con Gazebo y Control de Posición

```
paskuich@paskuich-GV62-7RE:~/control_ws$ ros2 topic pub /joint_controller/commands std_msgs/msg/Float64
MultiArray "data: [0.0, -0.577, -1.8, 0.0]"
publisher: beginning loop
publishing #1: std_msgs.msg.Float64MultiArray(layout=std_msgs.msg.MultiArrayLayout(dim=[], data_offset=
0), data=[0.0, -0.577, -1.8, 0.0])
```

Figura 5.6: Plot del control: Pose 2

```
header:
  stamp:
    sec: 2209
    nanosec: 410000000
  frame_id: ''
name:
- arm1_joint
- arm2_joint
- arm3_joint
- arm4_joint
position:
- 6.595397727515717e-06
- -0.5770014099590433
- -1.7999966490078383
- -6.011929978289032e-05
```

Figura 5.7: Visualización del contenido del topic joint_states: Pose 2

5.3.4. Robot en Pose 3

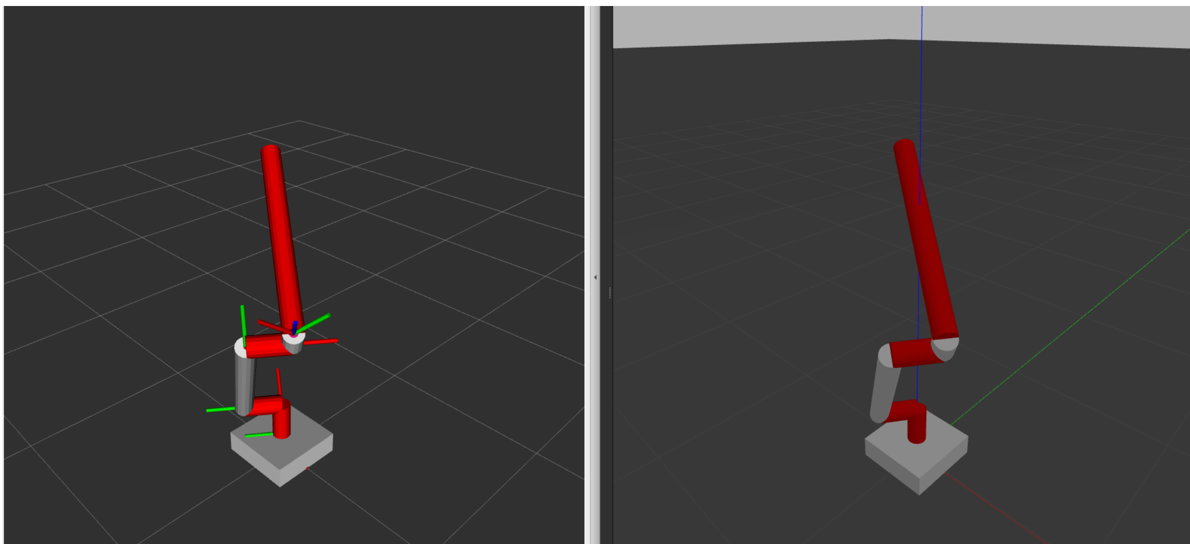

Figura 5.8: Robot en Pose 3 simulado con Gazebo y Control de Posición

```
paskuich@paskuich-GV62-7RE:~/control_ws$ ros2 topic pub /joint_controller/commands std_msgs/msg/Float64
MultiArray "data: [-3.770, -0.577, -1.596, -2.411]"
publisher: beginning loop
publishing #1: std_msgs.msg.Float64MultiArray(layout=std_msgs.msg.MultiArrayLayout(dim=[], data_offset=
0), data=[-3.77, -0.577, -1.596, -2.411])
```

Figura 5.9: Plot del control: Pose 3

```
header:
  stamp:
    sec: 2243
    nanosec: 430000000
  frame_id: ''
name:
- arm1_joint
- arm2_joint
- arm3_joint
- arm4_joint
position:
- -3.7699935015285218
- -0.5770027427807114
- -1.5959969784474466
- -2.4109997824427993
```


Figura 5.10: Visualización del contenido del topic joint_states: Pose 3

CAPÍTULO 6

Planificación de Trayectorias del Robot

Contenido

6.1. Control de Trayectorias mediante MoveIt **62**
6.2. Directorios relacionados con el Control de Trayectorias **62**
6.2.1. Directorio: description 62
6.2.2. Directorio de ejecución: launch 63
6.2.3. Directorio de configuración: config 67
6.3. Uso de MoveItSetupAssistant **67**
6.3.1. Lanzamiento de MoveItSetupAssistant 68
6.3.2. Carga de la descripción URDF 68
6.3.3. Self-Collitions 68
6.3.4. Planning Groups 69
6.3.5. Robot Poses 70
6.3.6. Controllers 70
6.3.7. Generate Package 71
6.4. Resultados de la Planificación de Trayectorias mediante MoveIt **71**
6.4.1. Vista Inicial 72
6.4.2. Establecer una Pose Final 73
6.4.3. Trayectoria hacia la Pose Final 73
6.4.4. Trayectoria Finalizada 74
6.5. Consideraciones **75**
6.5.1. Sincronización de Relojes 75

6.5.2. Colisiones y Posiciones Erróneas 75
6.5.3. Restablecimiento y reinicio de la Pose 76

Una vez conseguido el control de posición mediante el uso de ROS 2, se investigarán las herramientas de simulación capaces de realizar una planificación de trayectorias. Esta planificación es una técnica usada para garantizar que un robot siga una trayectoria concreta en el espacio a lo largo del tiempo. No solo se define una secuencia de posiciones en el espacio, sino también la velocidad y la aceleración con las que deben alcanzarse, proporcionando una referencia de tiempo precisa.

Es un tipo de control fundamental en aplicaciones donde se considera crítico un movimiento preciso, fluido y seguro.

6.1. Control de Trayectorias mediante MoveIt

MoveIt es una herramienta avanzada en ROS 2 para la manipulación y planificación de movimiento en robots. Su funcionalidad incluye la generación, planificación y ejecución de trayectorias, siendo específicamente útil en el diseño de controladores para manipuladores robóticos. Utiliza conceptos de cinemática, dinámica y algoritmos de planificación para garantizar que los robots sigan trayectorias precisas y eficientes.

En ROS, MoveIt se estableció como una herramienta robusta, altamente optimizada y con funcionalidad completa para la planificación y el control de movimientos de robots, incluyendo cinemáticas y dinámicas, y generación de trayectorias. Sin embargo, en ROS 2, aunque se ha portado parcialmente como MoveIt 2, aún carece de la misma estabilidad y optimización que su predecesor, lo que puede requerir modificaciones adicionales en el código, así como ajustes específicos para asegurar un funcionamiento adecuado en aplicaciones avanzadas de control y manipulación robótica.

6.2. Directorios relacionados con el Control de Trayectorias

Primero, se crea un nuevo paquete que se llamará sim_pkg_moveit_config. Para la creación de este paquete, se reciclarán algunos archivos del paquete sim_pkg.

6.2.1. Directorio: description

Se hará uso del Código 5.1 que se vio en el apartado de control, aunque se añadirán nuevas configuraciones.

6.2.1.1. new_robot.control.xacro

Los cambios añadidos al archivo son:

- **Plugin Hardware: mock_components/GenericSystem**: Este *plugin* es parecido al que se vio anteriormente. Se usa para la creación de sistemas *hardware* simulados que imitan el comportamiento del *harware* real.

- **Sustitución del controlador: joint_controller**: Se sustituye el controlador joint_controller (que estaba destinado a un control de posición) por el controlador arm_controller (destinado al control de trayectorias).

Este cambio se puede ver reflejado en el código mediante la inclusión de las siguientes líneas:

Código 6.1: new_robot.urdf.xacro

```
1  <!-- Esta parte anade el plugin mock_components-->
2  <hardware>
3      <plugin>mock_components/GenericSystem</plugin>
4  </hardware>
5
6  <!-- Esta parte sustituye a la busqueda del archivo joint.yaml -->
7  <gazebo>
8      <plugin filename="libgazebo_ros2_control.so" name="gazebo_ros2_control">
9          <parameters>$(find sim_pkg_moveit_config)/config/arm_controllers.
               yaml</parameters>
10     </plugin>
11 </gazebo>
```

6.2.2. Directorio de ejecución: launch

En este directorio se almacenarán los ejecutables creados por MoveItSetupAssistant para el correcto funcionamiento de la herramienta, además del fichero de ejecución **control.launch.py** con las modificaciones pertinentes para el control de trayectorias mediante MoveIt:

6.2.2.1. sim_control.launch.py

Se crea un nuevo archivo *launch* que tomará el nombre de sim_control.launch.py. Este archivo contiene todas las declaraciones de parámetros y nodos que tenía el ejecutable mostrado en el Código 5.2. Además, se añaden las líneas de código características de MoveIt.

- **Modificaciones para la extracción de datos del nuevo URDF**: Se tienen que modificar las inclusiones del nuevo archivo URDF, el cual se encuentra en el directorio *description* de este nuevo paquete.

- **Configuración de MoveIt**: Se usa MoveItConfigsBuilder para cargar y configurar la nueva descripción URDF, los controladores de MoveIt, el *pipeline* de planificación y el nodo publicador de toda la descripción y semántica del robot.

- **Lanzamiento de Nodos**:

  - **move_group**: Es un nodo propio de MoveIt que gestiona la planificación de trayectorias y ejecución de movimientos. Debe habilitarse la opción de use_sim_time para la sincronización de los relojes de los nodos. Sin esta opción, MoveIt no funciona.
  - **rviz**: Se debe configurar el nodo de ejecución de RVIZ para que añada la nueva descripción del robot con las propiedades de MoveIt.
  - **static_transform_publisher**: Este nodo publica la estática entre los *frames* de *world* y *base_link*. Es importante ya que define la base del robot en el entorno simulado.
  - **ros2_control_node**: Lanza el gestor de controladores del control_manager haciendo uso del archivo de controladores generado por MoveItSetupAssistant, que tiene el nombre de ros2_controllers.yaml
  - **Nodos de controladores**: Finalmente, se lanza el nodo de *joint_state_broadcaster* (visto en el apartado anterior) y el nodo que controla las trayectorias del robot (arm_controller).

- **Simulación de Gazebo**: En este caso, la ejecución de Gazebo es exactamente igual al del *launch* del apartado anterior, ya que se han modificado sus argumentos de entrada.

El código del *launch* correspondiente es el que se muestra a continuación:

Código 6.2: sim_control.launch.py

```
1  import os
2  from launch import LaunchDescription
3  from launch.actions import DeclareLaunchArgument
4  from launch.actions import IncludeLaunchDescription
5  from launch.substitutions import LaunchConfiguration, PathJoinSubstitution
6  from launch.actions import ExecuteProcess
7  from launch_ros.actions import Node
8  from launch_ros.substitutions import FindPackageShare
9  from ament_index_python.packages import get_package_share_directory
10 from moveit_configs_utils import MoveItConfigsBuilder
11
12 def generate_launch_description():
13     # Specify the name of the package and path to xacro file within the
           package
14     # Specify the name of the package and path to xacro file within the
           package
15     package_arg = DeclareLaunchArgument('urdf_package',
16                                         description='The package where the
                                                robot description is located',
17                                         default_value='sim_pkg_moveit_config
                                                ')
18     model_arg = DeclareLaunchArgument('urdf_package_path',
19                                       description='The path to the robot
                                              description relative to the
                                              package root',
20                                       default_value='description/new_robot.
                                              control.xacro')
21
```

```
22  # Load the robot configuration
23  moveit_config = (
24      MoveItConfigsBuilder("sim_pkg")
25      .robot_description(file_path="description/new_robot.control.xacro")
26      .trajectory_execution(file_path="config/moveit_controllers.yaml")
27      .planning_scene_monitor(
28          publish_robot_description=True,
            publish_robot_description_semantic=True
29      )
30      .planning_pipelines(
31          pipelines=["ompl"]
32      )
33      .to_moveit_configs()
34  )
35
36  # Start the actual move_group node/action server
37  run_move_group_node = Node(
38      package="moveit_ros_move_group",
39      executable="move_group",
40      output="screen",
41      parameters=[
42          moveit_config.to_dict(),
43          {"use_sim_time": True}
44      ],
45  )
46
47  # Path to RViz configuration file
48  rvizconfig_arg = DeclareLaunchArgument(
49      name='rvizconfig',
50      default_value=PathJoinSubstitution([FindPackageShare('urdf_pkg'), '
        worlds', 'view_robot.rviz']),
51  )
52
53  # Include the Gazebo launch file, provided by the gazebo_ros package
54  gazebo_launch = IncludeLaunchDescription(
55      PathJoinSubstitution([FindPackageShare('sim_pkg'), 'launch', 'gazebo
        .launch.py']),
56      launch_arguments={
57          'urdf_package': LaunchConfiguration('urdf_package'),
58          'urdf_package_path': LaunchConfiguration('urdf_package_path')
59      }.items(),
60  )
61
62  # Configure the RViz node
63  rviz_node = Node(
64      package='rviz2',
65      executable='rviz2',
66      output='screen',
67      arguments=['-d', LaunchConfiguration('rvizconfig')],
68      parameters=[
69          moveit_config.robot_description,
70          moveit_config.robot_description_semantic,
71          moveit_config.robot_description_kinematics,
72          moveit_config.planning_pipelines,
73          moveit_config.joint_limits,
```

```
74          ],
75      )
76
77      static_tf = Node(
78          package="tf2_ros",
79          executable="static_transform_publisher",
80          name="static_transform_publisher",
81          output="log",
82          arguments=["--frame-id", "world", "--child-frame-id", "base_link"],
83      )
84
85      ros2_controllers_path = os.path.join(
86          get_package_share_directory("sim_pkg_moveit_config"),
87          "config",
88          "ros2_controllers.yaml",
89      )
90
91      ros2_control_node = Node(
92          package="controller_manager",
93          executable="ros2_control_node",
94          parameters=[ros2_controllers_path],
95          remappings=[
96              ("/controller_manager/robot_description", "/robot_description"),
97          ],
98          output="both",
99      )
100
101     # Configure controller nodes
102     load_joint_state_controller = ExecuteProcess(
103         cmd=['ros2', 'control', 'load_controller', '--set-state', 'active',
104             'joint_state_broadcaster'],
105         output='screen'
106     )
107
108     load_arm_controller = ExecuteProcess(
109         cmd=['ros2', 'control', 'load_controller', '--set-state', 'active',
110             'arm_controller'],
111         output='screen'
112     )
113
114     # Run all nodes
115     return LaunchDescription([
116         package_arg,
117         model_arg,
118         rvizconfig_arg,
119         static_tf,
120         gazebo_launch,
121         rviz_node,
122         run_move_group_node,
123         ros2_control_node,
124         load_joint_state_controller,
125         load_arm_controller,
126     ])
```

La importancia de la terminación _moveit_config de este paquete es debido a que la función

moveit_config añade esa terminación a la hora de buscar los archivos en el paquete. Si este directorio no la tuviese, el *launch* no funcionaría.

6.2.3. Directorio de configuración: config

Para que todos los controladores funcionen correctamente, es necesario crear los archivos de configuración correspondientes. La mayoría de archivos los da el asistente *MoveItSetupAssistant*. Sin embargo, algunos de ellos deben ser creados a mano.

6.2.3.1. arm_controllers

Al igual que se hizo para el controlador de posición del apartado anterior, debe crearse un archivo de configuración específico para controlar la trayectoria. Este archivo contiene lo siguiente:

Código 6.3: arm_controllers.yaml

```yaml
controller_manager:
  ros__parameters:
    update_rate: 100
    use_sim_time: true

    joint_state_broadcaster:
      type: joint_state_broadcaster/JointStateBroadcaster

    arm_controller:
      type: joint_trajectory_controller/JointTrajectoryController

arm_controller:
  ros__parameters:
    command_interfaces:
      - position
    state_interfaces:
      - position
      - velocity
    joints:
    - arm1_joint
    - arm2_joint
    - arm3_joint
    - arm4_joint
```

Como se observa, este controlador es del tipo joint_trajectory_controller y gestiona mensajes del tipo JointTrajectoryController.

6.3. Uso de MoveItSetupAssistant

Al igual que MoveIt en ROS 1, para ROS 2 esta herramienta cuenta con una interfaz gráfica diseñada para facilitar las configuraciones iniciales del robot con el objetivo de integrarlas en aplicaciones de planificación de caminos, control y simulación. Su función es la de generar los archivos y configura-

ciones personalizadas para que MoveIt funcione a partir de un robot definido con una descripción URDF.

Los pasos a seguir para obtener dichos archivos se muestran a lo largo de esta sección.

6.3.1. Lanzamiento de MoveItSetupAssistant

Una vez que se instala la herramienta MoveIt, se puede ejecutar el comando específico que abrirá esta interfaz:

```
ros2 launch moveit_setup_assistant setup_assistant.launch.py.
```

La interfaz se muestra en la Figura 6.1.

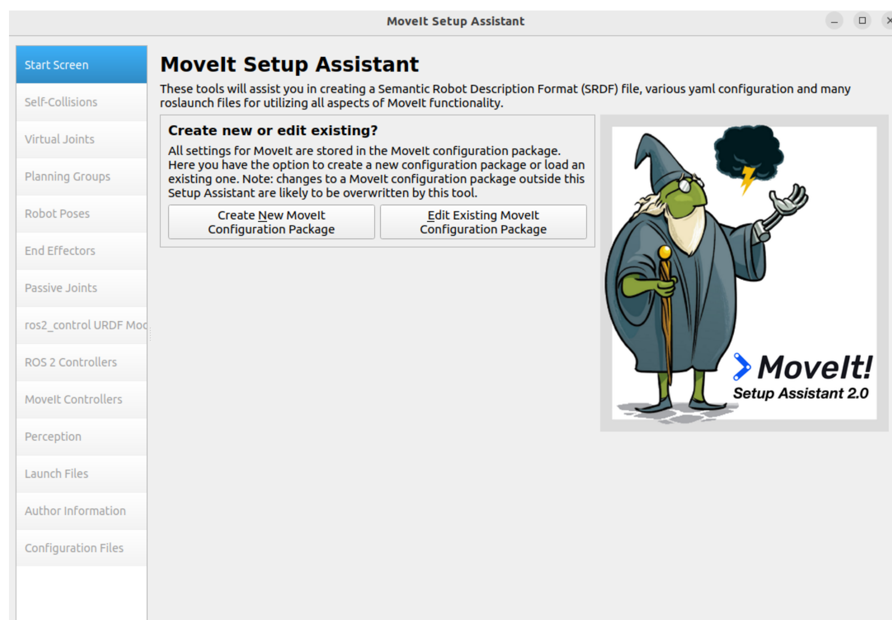

Figura 6.1: MoveItSetupAssistant - Home Page

6.3.2. Carga de la descripción URDF

Para acceder al archivo URDF, se debe pulsar en *Create New MoveIt Configuration Package*. Se habilita la opción de cargar el modelo del robot en descripción URDF. Buscando el archivo y pulsando en el botón *Load Files* se mostrará el robot.

6.3.3. Self-Collitions

Se debe generar una matriz de colisiones, la cual se realizará por medio de las colisiones definidas en el archivo URDF. Es por ello que el Capítulo 2, en la Sección 2.2, se comentaba que las colisiones para este caso particular debían ser las propias geometrías porque se tratan de elementos sólidos.

6.3.4. Planning Groups

Es necesario definir un Planning Group con las características del robot. Dado que se pueden crear varios grupos, podría crearse uno para los efectores finales con la intención de controlar dichos elementos.

Para el caso particular, se rellenan los datos de la siguiente forma:

- **Group Name**: arm
- **Kinematic Solver**: kdl_kinematics_plugin/KDLKinematicsPlugin

Para el resto de dejan los valores predeterminados. Como se observa, la forma de resolver las cinemáticas es mediante KDL, biblioteca para cálculos cinemáticos que se comentó en la Sección 1.2.2.1.

También se indica que la forma de hacer la planificación de caminos es mediante el uso de OMPL [21]. Esta biblioteca de código abierto está diseñada para resolver los problemas de planificación de movimientos. Es decir, permite calcular la trayectoria del robot desde un punto inicial hasta un punto objetivo, haciendo uso de las restricciones del propio robot y del entorno.

Seguidamente, y dado que el robot planteado es de cadena cinemática abierta, se pulsa en el botón de *Add Kin. Chain*. Se abre una ventana en la cual se establece la articulación inicial y final del robot, calculando las transformaciones y cinemáticas de forma automática.

6.3.5. Robot Poses

La interfaz de MoveitSetupAssistant permite establecer distintas poses predeterminadas del robot y guardarlas. Esto puede resultar muy útil a la hora de plantear un proceso cíclico donde sea necesario que el robot alcance distintas poses establecidas, por ejemplo, un Pick&Place.

Además, la configuración de estas poses se realiza por medio de *slicers* similares a los de la herramienta joint_states_publisher_gui, como se muestra en la Figura 6.2.

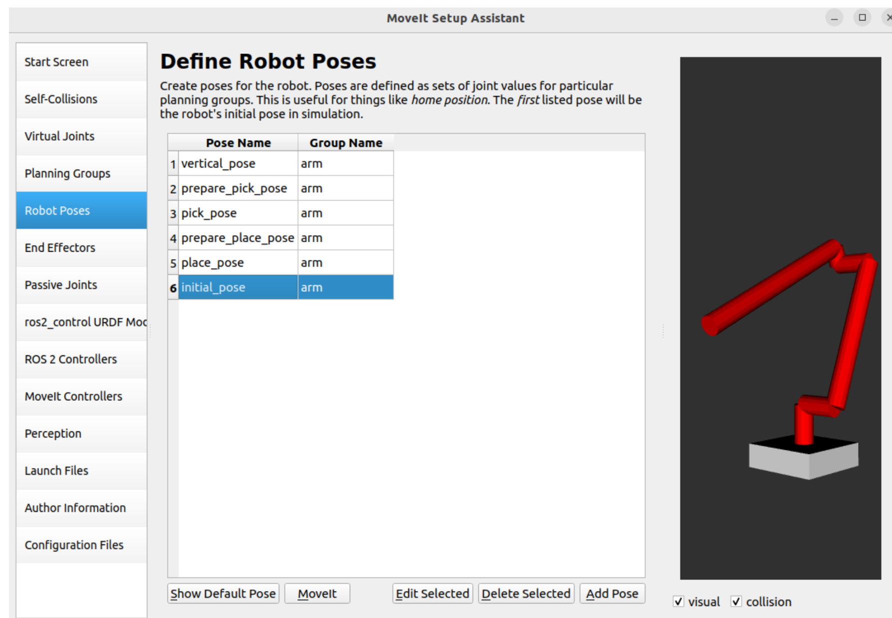

Figura 6.2: MoveItSetupAssistant - Robot Poses

6.3.6. Controllers

En las pestañas de ROS 2 Controllers y MoveIt Controllers se deben generar los controladores necesarios para que el robot funcione correctamente. Para ello se pulsa sobre los botones de *Auto Add* y se añaden de forma automática dichos controladores. Se generan dos archivos de configuración: ros2_controllers.yaml y moveit_controllers.yaml. Para el caso del controlador de ROS 2, el archivo deberá ser modificado ya que no cuenta con el controlador de control de trayectorias:

Código 6.4: ros2_controllers.yaml

```
1  controller_manager:
2    ros__parameters:
3      update_rate: 100  # Hz
4      use_sim_time: true
5
6      joint_trajectory_controller:
7        type: position_controllers/JointTrajectoryController
8
9
10     joint_state_broadcaster:
```

```
11      type: joint_state_broadcaster/JointStateBroadcaster
12
13 joint_trajectory_controller:
14   ros__parameters:
15     joints:
16       - arm1_joint
17       - arm2_joint
18       - arm3_joint
19       - arm4_joint
```

Se ha cambiado a mano el controlador de **joint_trajectory_controller** para que pueda ser del tipo que interesa en este caso.

6.3.7. Generate Package

Para generar el paquete, se introduce la información del autor y se configura la generación del paquete de MoveIt. Para que esto funcione correctamente, se debe crear un directorio con el nombre de sim_pkg (no debe estar dentro del workspace). En el apartado de *Configuration Files*, se pulsa en el botón *Browse* para buscar el directorio creado. Pulsando sobre el botón *Generate Package* se generarán los archivos necesarios de configuración y ejecución.

Lo último que se debe hacer para que MoveIt funcione correctamente, es extraer los archivos del paquete recientemente creado. Éste tendrá en su interior los directorios de *launch* y *config* (con sus archivos correspondientes), ficheros de ROS 2, como CMakeList.txt y package.xml, y el archivo generado .setup_assistant que coordinará todo lo demás.

Finalmente, se introducen los archivos dentro del paquete sim_pkg_moveit_config del *workspace*, teniendo en cuenta que los archivos de configuración van en su directorio correspondiente. También se debe tener en cuenta que ambos paquetes tienen su propio package.xml y CMakeList.txt, por lo que habrá que unificar ambos.

6.4. Resultados de la Planificación de Trayectorias mediante MoveIt

Para visualizar los resultados del control de trayectorias mediante MoveIt, se ejecuta el *launch* del paquete sim_pkg_moveit_config creado anteriormente. Este tiene el nombre de sim_control.launch.py.

Al arrancar, se abre tanto Gazebo como RVIZ. Sin embargo, no se muestra nada de MoveIt. Para ello, dentro de RVIZ, se debe importar el *plugin* de MotionPlanning a través de *Add >moveit_ros_visualization >MotionPlanning*.

Esto introduce la interfaz de MoveIt en RVIZ para la planificación de trayectorias, que se muestra en la Figura 6.3

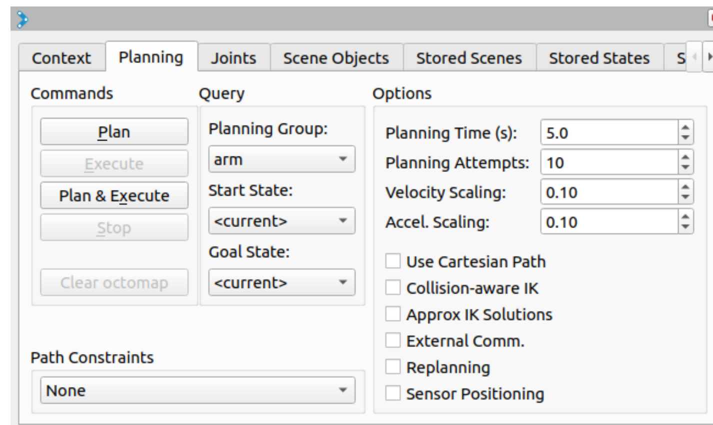

Figura 6.3: Plugin visual en RVIZ de MotionPlanning

Para usarla, basta con posicionar las articulaciones en la pose deseada. Dicha pose se verá de color naranja en RVIZ. Para establecer la pose, se mueven las articulaciones desde la pestaña *Joints* de MotionPlanning. Una vez establecida, basta con pulsar el botón de *Plan & Execute*. Se visualiza, tanto en Gazebo como en RVIZ, cómo el robot va lentamente hasta la pose final, calculando el camino más rápido y sencillo de forma totalmente automática. A continuación se muestra el funcionamiento descrito:

6.4.1. Vista Inicial

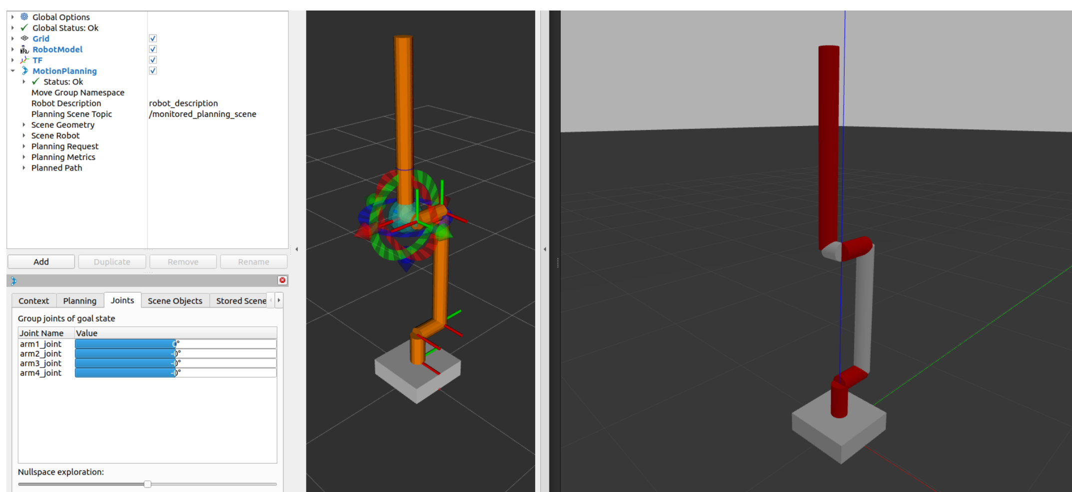

Figura 6.4: MotionPlanning - Home

6.4.2. Establecer una Pose Final

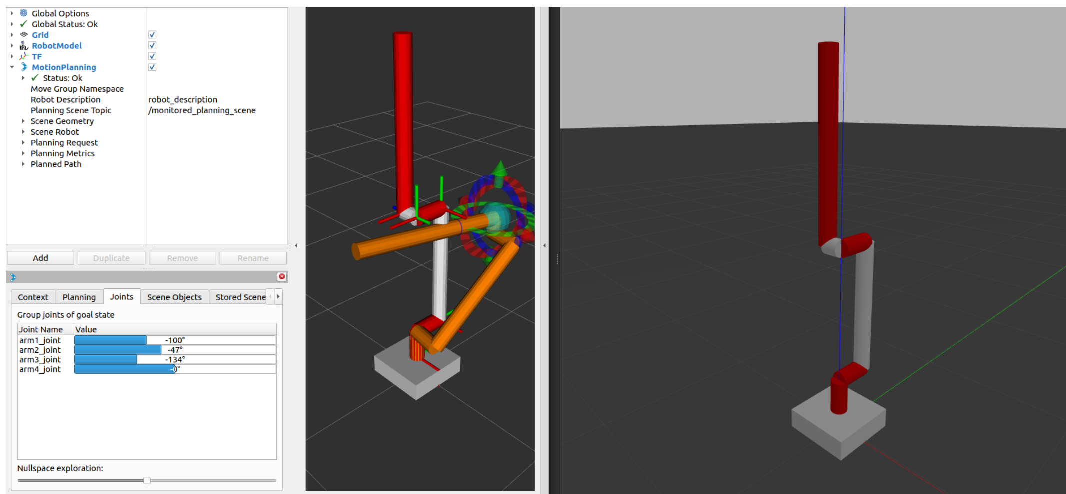

Figura 6.5: MotionPlanning - Establecer una Pose Final

6.4.3. Trayectoria hacia la Pose Final

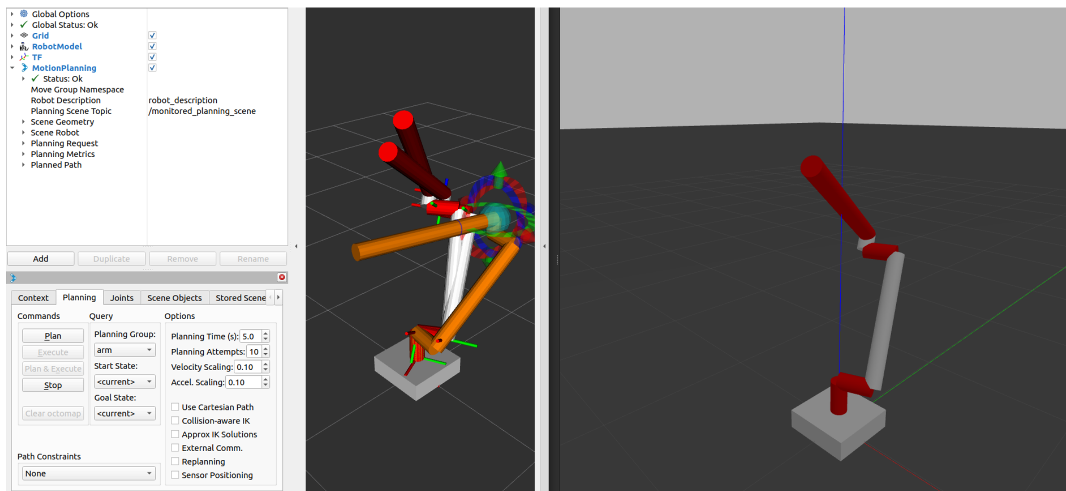

Figura 6.6: MotionPlanning - Trayectoria hacia Pose Final

6.4.4. Trayectoria Finalizada

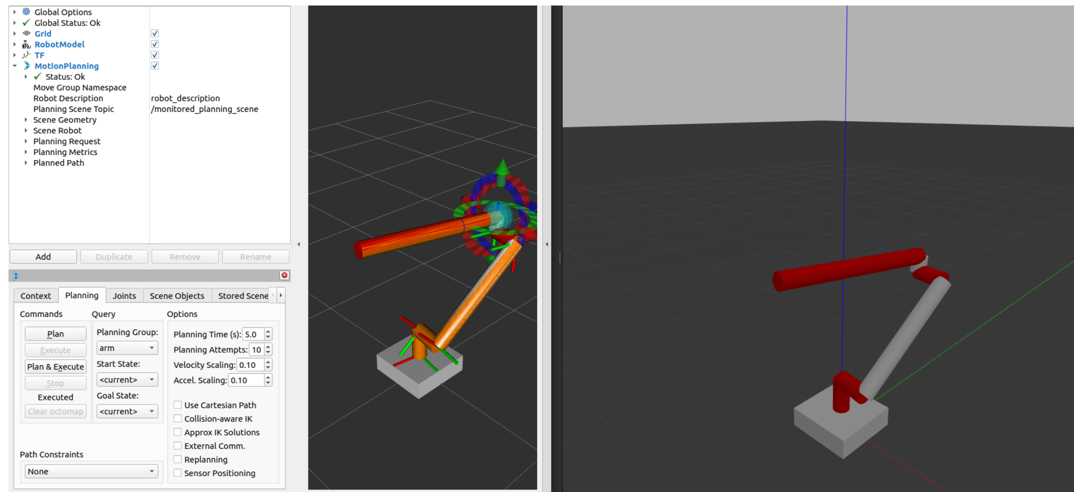

Figura 6.7: MotionPlanning - Trayectoria Finalizada

Para visualizar de forma dinámica el movimimiento de cada articulación, se extrae en un archivo .csv, mediante *PlotJuggler*, la trayectoria del robot. Haciendo uso de la librería de python 3: matplotlib, y programando un archivo ejecutable de python que recoja los archivos .csv y haga el *plot*, se obtiene la siguiente gráfica:

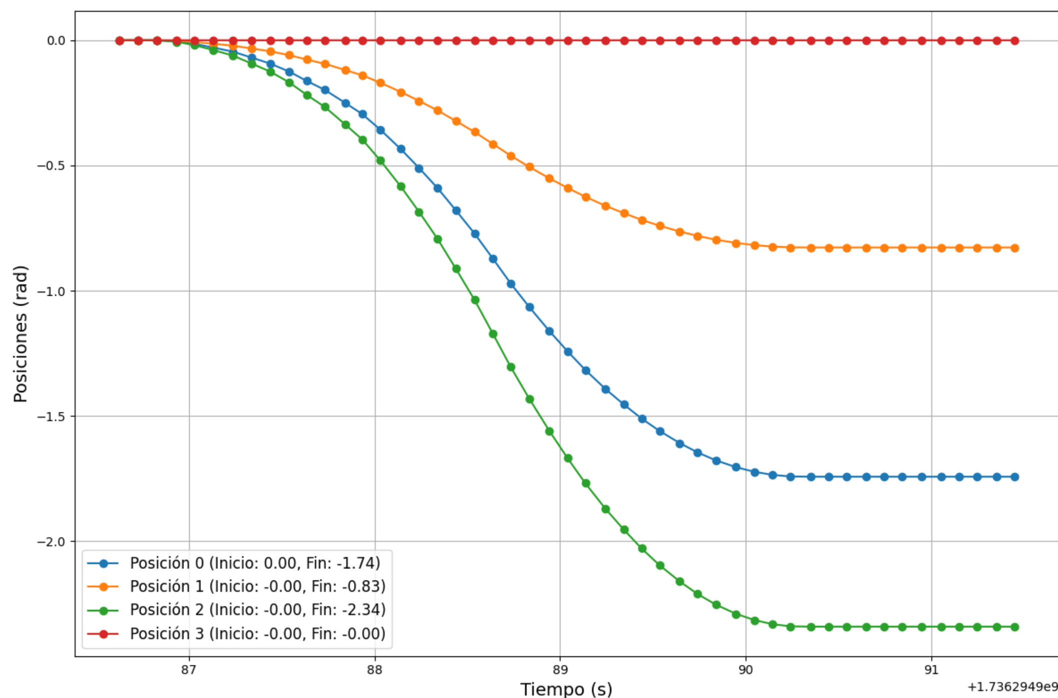

Figura 6.8: Representación de la Trayectoria mediante Gráfica de las Posiciones Articulares

6.5. Consideraciones

Hay ciertas consideraciones que es importante tener en cuenta a la hora de ejecutar este archivo:

6.5.1. Sincronización de Relojes

Para que el *MotionPlaning* se ejecute, todos los relojes de control deben tener el mismo valor y con la opción de tiempo de simulación activada, es decir, todos deben estar perfectamente sincronizados o no se ejecutará el movimiento.

6.5.2. Colisiones y Posiciones Erróneas

Debido a las geometrías del robot y colisiones establecidas, MoveIt detecta ciertas posiciones y las muestra como una colisión a la hora de planificar caminos:

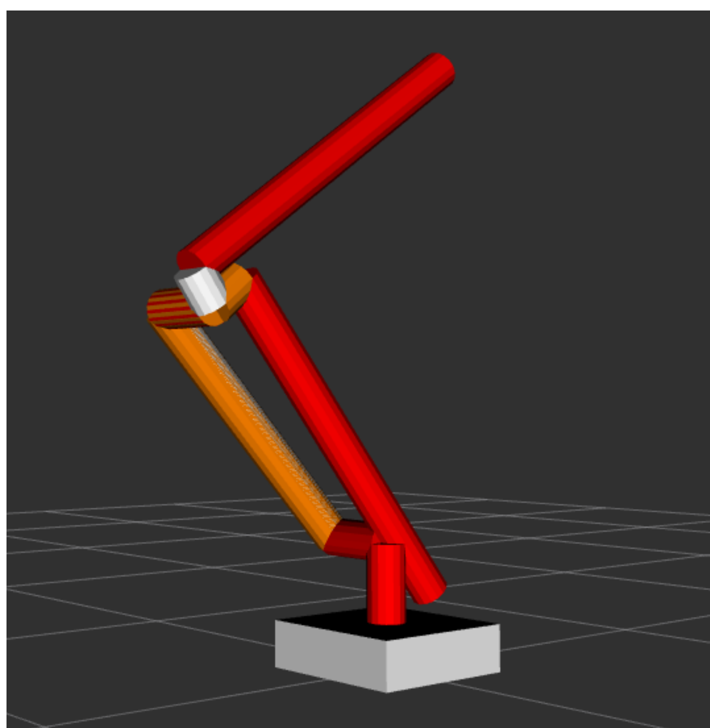

Figura 6.9: MotionPlanning - Colisión

Como se observa, MoveIt muestra en rojo el estado de colisión entre las articulaciones que se muestran en ese color. Esto provoca problemas en la simulación ya que se contempla que la posición definida no es alcanzable y los controladores se paran. Esto también ocurre si se traspasa el suelo. Es lógico que esto pase para la simulación, debido a las dinámicas propias añadidas. En caso de ser un sistema real se obtendría un mensaje de error en el que se indicase que el robot no puede alcanzar dicha posición y sus motores pararían.

6.5.3. Restablecimiento y reinicio de la Pose

La pose del robot se queda guardada en Gazebo y puede ser tedioso si se queda en una posición inalcanzable, ya que este no será posible moverlo una vez establecida esa pose, debido al apagado de los controladores. Para restablecer la pose, Gazebo eliminará el robot del mundo mediante el siguiente comando:

```
ros2 service call /delete_entity gazebo_msgs/srv/DeleteEntity "name: 'robot'"
```

De esta forma puede volver a ejecutarse el *launch* y se vuelve a mostrar el robot en posición inicial.

CAPÍTULO 7

Mejora de la Visualización: Introducción de Mallas

Contenido

7.1. Adición de los Meshes a la Descripción URDF **78**

7.2. Resultados de la Visualización del Robot con Meshes **78**

7.2.1. Pose 1 79

7.2.2. Pose 2 80

Una malla (o *mesh*) es una representación tridimensional de un objeto físico o virtual que utiliza una red de vértices conectados por aristas para formar caras y polígonos. Una estructura geométrica que representa la forma de un objeto en el espacio tridimensional. En el desarrollo de un gemelo digital, estos *meshes* se usan para dar forma a las articulaciones, haciendo que sean similares al robot físico real.

En el Capítulo 2, Sección 2.1, se descompone el robot en distintas partes, mostrando el modelado 3D de cada una de ellas.

En este capítulo se pretende hacer uso de esos *meshes* para que la visualización del robot sea mucho más realista.

7.1. Adición de los Meshes a la Descripción URDF

Para completar la visualización y añadir los *meshes* correspondientes en cada articulación, se hace uso de SolidWorks [22]. Este programa de diseño y simulación 3D tiene la opción de hacer una exportación URDF, que da como resultado un archivo de descripción URDF y los *meshes* correspondientes a cada una de las partes. Haciendo uso de ese *plugin*, se configura cada una de las partes del modelo 3D y se exporta.

Para poder ejecutar la visualización en RVIZ, se crea un nuevo paquete mesh_urdf_pkg. Dentro de este, se utilizan los mismos directorios que se utilizaron en el paquete urdf_pkg, añadiendo el directorio de *meshes* con los archivos en formato 3D (.stl) correspondientes.

El archivo .xacro de descripción, cambia la geometría de cada articulación sustituyéndola por el archivo 3D. A continuación se muestra un ejemplo de este cambio:

Código 7.1: realrobot.urdf.xacro

```
1 <geometry>
2     <mesh filename="package://mesh_urdf_pkg/meshes/base_link.STL"/>
3 </geometry>
```

Además, dentro de la descripción URDF, están calculadas, directamente desde SolidWorks, las inercias correspondientes a cada estructura según la geometría y el peso. Si cierto es que esto es una ventaja, en SolidWorks se hace complicada la adición de materiales y pesos, por lo que lo propio sería hacer uso de OnShape [23], una plataforma CAD online que permite calcular las inercias importando el archivo .stl y estableciendo el material y peso.

El resto del paquete es exactamente igual que su predecesor. Se realizan ciertas modificaciones en el archivo URDF, dado que el diseño en SolidWorks tiene ciertos parámetros que no han sido configurados para esta tarea (como los ejes de coordenadas, que están creados en el plano de perfil y deben ajustarse en la descripción), y se ejecuta el *launch*.

7.2. Resultados de la Visualización del Robot con Meshes

Los resultados de la adición de los *meshes* a la descripción URDF se muestran en RVIZ:

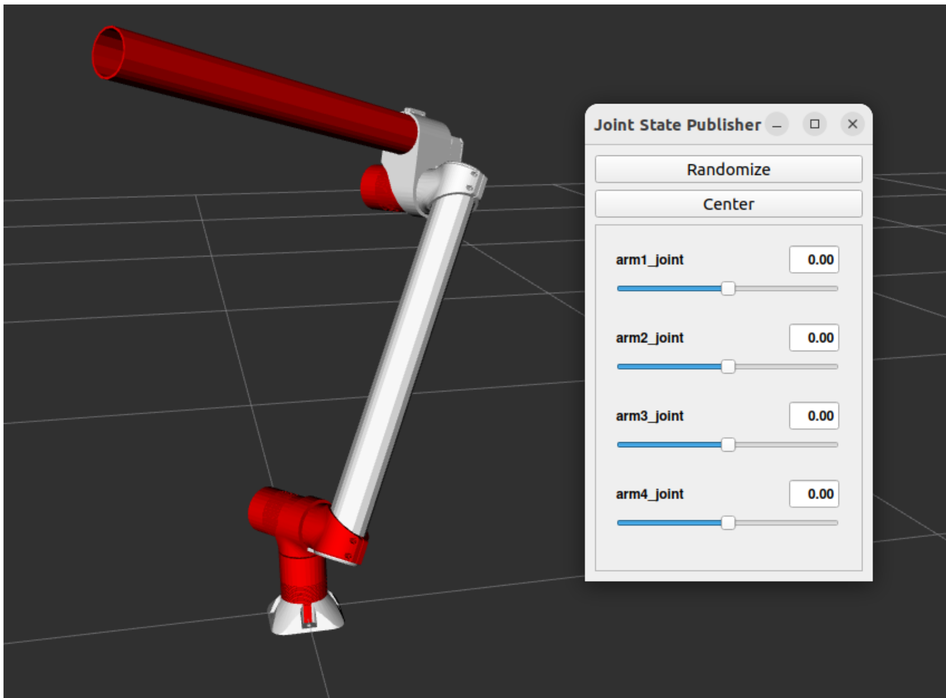

Figura 7.1: Visualización del Robot Real

Como se observa, pueden crearse distintas poses mediante la GUI de Joint State Publisher. De esta forma se obtienen diversos resultados para el robot visualizado:

7.2.1. Pose 1

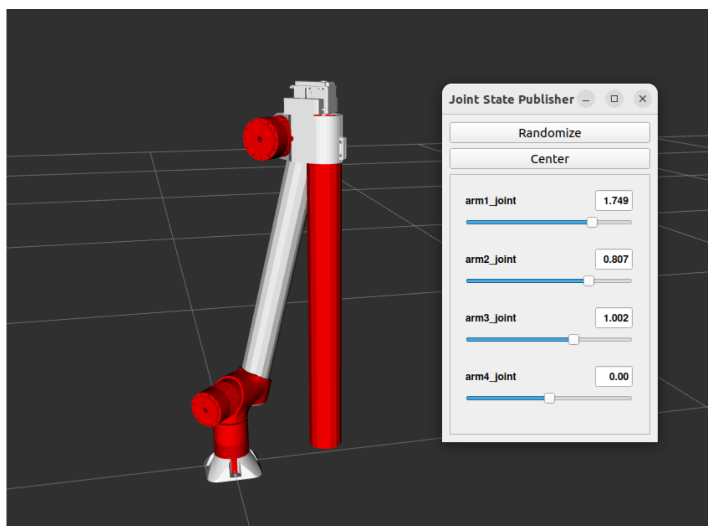

Figura 7.2: Visualización del Robot Real - Pose 1

7.2.2. Pose 2

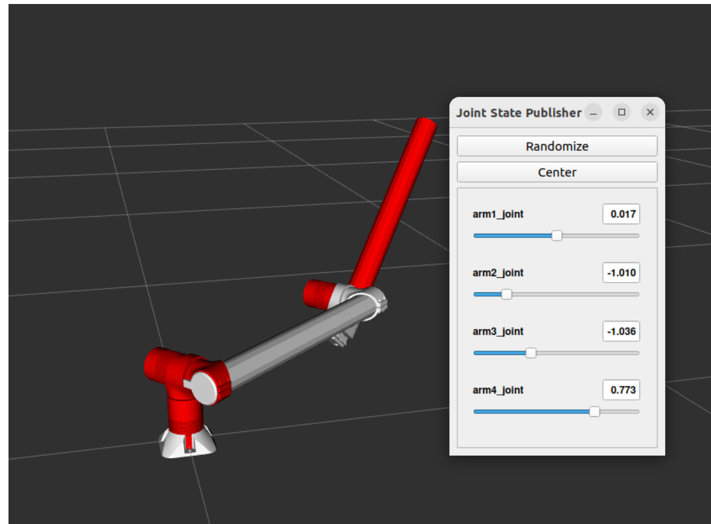

Figura 7.3: Visualización del Robot Real - Pose 2

Las limitaciones que la inclusión de los *meshes* provocan, se comentan en la conclusión (Capítulo 8.

CAPÍTULO 8

Conclusiones

En este último capítulo se analiza la calidad de las soluciones aportadas para el planteamiento de todo el trabajo, así como la cumplimentación de los objetivos planteados en la sección 1.3. También se darán posibles mejoras que añadir en un futuro, además de los planteamientos necesarios para continuar con este proyecto.

Las contribuciones aportadas mediante este trabajo son diversas. Para que el robot pueda realizar movimientos, es necesario tener un control de posición. En un principio, este control estaba planteado mediante Matlab. Dicha herramienta es altamente potente pero limitada. La integración de nuevos módulos o la ampliación de un sistema puede resultar tedioso para los usuarios. Sin embargo, ROS 2 está diseñado específicamente para trabajar con sistemas modulares donde cada nodo realiza una función específica, permitiendo ampliar el sistema o modificarlo con mayor facilidad y flexibilidad. Además, la curva de aprendizaje para nuevos usuarios del grupo Roborescue en cuanto a la integración de lenguajes de programación, resultará más sencilla en el caso de ROS 2. Este es compatible con múltiples lenguajes y permite integrar bibliotecas personalizadas y especializadas según el objetivo a conseguir.

A lo largo del trabajo se plantearon diferentes objetivos. El primero de ellos era realizar la correcta descripción URDF. Se ha conseguido satisfactoriamente, aportando una gran flexibilidad al sistema. El uso de URDF para el modelado de robots tiene grandes ventajas en comparación con otros enfoques, como descripciones en formatos propietarios, lenguajes de programación o herramientas específicas. Este formato es de alta estandarización y compatibilidad, con una estructura simple y clara. Además, se puede realizar su portabilidad con baja carga de recursos ya que se trata de un archivo en formato de texto.

Por otro lado, se pretendía hacer uso de herramientas de simulación. RVIZ ha resultado ser un software muy completo para este objetivo. Este permite, no solo visualizar el brazo robótico, sino incluir herramientas que aportan recursos a la hora de realizar la implementación del control. También ha sido de gran utilidad para la corrección y verificación de la inclusión de la geometría y la dinámica, así como la detección temprana de errores en el modelado.

Seguidamente, se plantea el uso de la simulación, acercando este proyecto más aún a la idea de gemelo digital. La utilización de Gazebo como una útil herramienta de simulación ha facilitado que esto sea una realidad, permitiendo así la adición de un control por posición simulado, además de la correspondiente planificación de trayectorias.

El estudio del control mediante ROS 2, ha permitido la publicación de mensajes concretos para hacer uso de ellos a la hora de mover el robot. Este añadido es necesario para la posterior creación del Gemelo Digital. El control de posición mediante el *framework* de ros_control es extenso y contiene algunas dificultades. Sin embargo, el uso de paquetes modulares que este incluye permite una accesibilidad y aprendizaje sencillo de los mismos. Las configuraciones y el uso de cada paquete requiere de paciencia para poder comprenderlos más profundamente. A pesar de dicho aprendizaje que conlleva una dificultad añadida, no es necesario indagar y profundizar excesivamente en ellos. Tan solo con entender su uso correctamente, se pueden desarrollar herramientas útiles para realizar el control de posición.

Por otro lado, la herramienta MoveIt ha sido de gran ayuda en el desarrollo de la planificación de trayectorias, dando resultados de movimiento suave y firme. Además, permite alejarse un mínimo del uso continuado de código, dando lugar a una interfaz gráfica más apta para el usuario. Esta herramienta también requiere de un aprendizaje directo, pero intentar acelerar este proceso conlleva a errores de código y acciones innecesarias en cuanto a programación. A pesar de todo, llegar a resultados finales de forma exitosa.

Finalmente, se hace la inclusión de las mallas de los archivos 3D. Lamentablemente, no se ha podido utilizar este modelo para simulación debido a la naturaleza y complejidad de los archivos 3D proporcionados. Gazebo no es capaz de procesar ciertos archivos de complejidad elevada. Esto se debe a la cantidad de aristas y puntos que contienen dichos modelos. En un futuro se estudiará el modelado de los mismos con la intención de reducir la geometría y poder procesar estos archivos en simulación.

En cuanto a la utilización de ROS para este trabajo, se llegan a diversas conclusiones. La utilización de este *software* como una herramienta para el modelado, simulación y control ha resultado muy útil. La información aportada por ROS 2, además de la inmensa comunidad de creadores y programadores que trabajan con herramientas de este calibre, hace sencillo el aprendizaje del *software*. Si bien es cierto que requiere de paciencia y cuidado a la hora de crear paquetes y configuraciones, el uso de módulos para cada parte permite aislar los errores y aprender sobre ellos para futuros proyectos.

En cuanto a su uso en el Roborescue UMA, la utilización de ROS 2 es muy adecuada. Cada uno de los departamentos del equipo trabaja en un ámbito concreto. Globalizando el uso del *software*

adecuado, permitirá la integración todos los sistemas. Además, también será de gran utilidad para la posterior creación de Gemelo Digital del manipulador robótico con el que cuenta el robot del equipo. Haciendo uso de este concepto, da la posibilidad de hacer pruebas aisladas en simulación, sin necesidad del robot real. El soporte en tiempo real también es una característica a favor del uso de este *software*, dado que para el ámbito de los robots de rescate puede ser un factor crítico.

Otro de los puntos que serán necesarios a desarrollar en el futuro, es la inclusión de un efector final y el desarrollo del mismo. Siguiendo los pasos aportados durante todo este estudio, se hará más sencilla su creación, pues este se puede extrapolar a diferentes ámbitos.

Resumiendo, a pesar de que este trabajo no está enfocado en la creación del Gemelo Digital completo, se han desarrollado las herramientas necesarias para poder obtenerlo con mayor facilidad. El estudio ha recorrido desde las etapas iniciales del proyecto, donde se modela mediante descripción la geometría del robot. Posteriormente, un aprendizaje y uso de herramientas de visualización y simulación necesarias para obtener el estado del robot de forma digital. Posteriormente, se han creado los sistemas y controladores capaces de comandar los movimientos al robot, realizando el control de posición y una planificación de trayectorias suaves y en la que se tiene en cuenta las físicas del robot. Este estudio en su completitud, que servirá como guía a futuros integrantes del equipo, para la creación del apartado software del control. Añadiendo una interfaz que trate los mensajes enviados y los recibidos por el propio robot y un estudio del *hardware* del mismo, dará lugar al Gemelo Digital completo. La utilización de ROS 2 y la flexibilidad aportada al código, permitirán modelar y cambiar aquello que sea necesario, además de poder realizar el aprendizaje desde unos primeros pasos hasta un nivel más elevado dentro de este ámbito.

Bibliografía

[1] H. Kitano, S. Tadokoro, I. Noda, H. Matsubara, T. Takahashi, A. Shinjou, and S. Shimada, "Robocup rescue: Search and rescue in large-scale disasters as a domain for autonomous agents research," in *IEEE SMC'99 Conference Proceedings. 1999 IEEE International Conference on Systems, Man, and Cybernetics (Cat. No. 99CH37028)*, vol. 6. IEEE, 1999, pp. 739–743.

[2] F. P. Audonnet, A. Hamilton, and G. Aragon-Camarasa, "A systematic comparison of simulation software for robotic arm manipulation using ros2," in *2022 22nd International Conference on Control, Automation and Systems (ICCAS)*. IEEE, 2022, pp. 755–762.

[3] Y. Jiang, S. Yin, K. Li, H. Luo, and O. Kaynak, "Industrial applications of digital twins," *Philosophical Transactions of the Royal Society A*, vol. 379, no. 2207, p. 20200360, 2021.

[4] A. V. Travez Tipan and C. M. Villafuerte Garzon, "Industria 5.0, revisión del pasado y futuro de la producción y la industria," *Ciencia Latina Revista Científica Multidisciplinar*, vol. 7, no. 1, pp. 1059–1070, ene. 2023. [Online]. Available: https://www.ciencialatina.org/index.php/cienciala/article/view/4457

[5] L. by UMA-TECH, "K-project," Sitio web oficial, University of Malaga, UMA-TECH, April 2016, plataforma para gestión de proyectos tecnológicos innovadores. [Online]. Available: https://www.link.uma.es/k-project/

[6] Ó. J. Cubillos Clavijo, "Robots de rescate humano," 2015. [Online]. Available: https://repositorio.ecci.edu.co/handle/001/1854?locale-attribute=es

[7] G. Bermudez, K. S. Novoa, and W. Infante, "La robótica en actividades de búsqueda y rescate urbano. origen, actualidad y perspectivas," *Tecnura*, vol. 8, no. 15, pp. 97–108, jul. 2004. [Online]. Available: https://revistas.udistrital.edu.co/index.php/Tecnura/article/view/6165

[8] R. Adrian and G. García, "Prototipo virtual de un robot móvil multi-terreno para aplicaciones de búsqueda y rescate," *ResearchGate, October*, 2016.

[9] E. Y. Moreno Martínez, "El uso de drones en operaciones de búsqueda y rescate en colombia: una revisión sistemática." 2023.

[10] M. Alcaraz-Carrazco, J. Gudiño-Lau, Ó. Issac-Zamora, S. M. Charre-Ibarra, J. A. Alcalá-Rodríguez, D. Vélez-Díaz *et al.*, "Robot submarino: estado del arte y diseño," *XIKUA Boletín Científico de la Escuela Superior de Tlahuelilpan*, vol. 10, no. 19, pp. 10–16, 2022.

[11] G. G. Gielen, H. C. Walscharts, and W. M. Sansen, "Isaac: A symbolic simulator for analog integrated circuits," *IEEE Journal of solid-state circuits*, vol. 24, no. 6, pp. 1587–1597, 1989.

[12] C. Mower, T. Stouraitis, J. Moura, C. Rauch, L. Yan, N. Z. Behabadi, M. Gienger, T. Vercauteren, C. Bergeles, and S. Vijayakumar, "Ros-pybullet interface: A framework for reliable contact simulation and human-robot interaction," in *Conference on Robot Learning*. PMLR, 2023, pp. 1411–1423.

[13] R. C. Maintainers, "Welcome to the ros2_control documentation," Sitio web oficial, ROS 2, 2023, documentación oficial sobre ROS Control. [Online]. Available: https://control.ros.org/rolling/index.html

[14] P. Robotics, "Moveit 2 documentation," Sitio web oficial, ROS 2, 2019, documentación oficial sobre MoveIt2. [Online]. Available: https://moveit.picknik.ai/main/index.html

[15] E. Tsardoulias and P. Mitkas, "Robotic frameworks, architectures and middleware comparison," *arXiv preprint arXiv:1711.06842*, 2017.

[16] J. Kay and A. R. Tsouroukdissian, "Real-time control in ros and ros 2.0," *ROSCon15*, 2015.

[17] D. Tola and P. Corke, "Understanding urdf: A dataset and analysis," *IEEE Robotics and Automation Letters*, 2024.

[18] H. R. Kam, S.-H. Lee, T. Park, and C.-H. Kim, "Rviz: a toolkit for real domain data visualization," *Telecommunication Systems*, vol. 60, pp. 337–345, 2015.

[19] N. Koenig and A. Howard, "Design and use paradigms for gazebo, an open-source multi-robot simulator," in *2004 IEEE/RSJ international conference on intelligent robots and systems (IROS)(IEEE Cat. No. 04CH37566)*, vol. 3. Ieee, 2004, pp. 2149–2154.

[20] S. Chitta, E. Marder-Eppstein, W. Meeussen, V. Pradeep, A. R. Tsouroukdissian, J. Bohren, D. Coleman, B. Magyar, G. Raiola, M. Lüdtke *et al.*, "ros_control: A generic and simple control framework for ros," *Journal of Open Source Software*, vol. 2, no. 20, pp. 456–456, 2017.

[21] I. A. Sucan, M. Moll, and L. E. Kavraki, "The open motion planning library," *IEEE Robotics & Automation Magazine*, vol. 19, no. 4, pp. 72–82, 2012.

[22] M. Feder, A. Giusti, and R. Vidoni, "An approach for automatic generation of the urdf file of modular robots from modules designed using solidworks," *Procedia Computer Science*, vol. 200, pp. 858–864, 2022.

[23] F. Cano and M. Martínez-Ripoll, “On shape,” *Journal of Molecular Structure: THEOCHEM*, vol. 258, no. 1-2, pp. 139–158, 1992.