



UNIVERSIDAD POLITÉCNICA DE MADRID

ESCUELA TÉCNICA SUPERIOR DE INGENIERÍA Y
DISEÑO INDUSTRIAL

Grado en Ingeniería Electrónica y Automática Industrial

TRABAJO FIN DE GRADO

GEMELO DIGITAL Y SIMULACIÓN DINÁMICA DE UN ORGANISMO ROBÓTICO ESCALADOR

Daniel Barroso Saugar

Tutor: Miguel Hernando
Gutierrez

Departamento: Ingeniería
eléctrica, electrónica automática
y física aplicada

Madrid, Febrero, 2025



POLITÉCNICA

UNIVERSIDAD POLITÉCNICA DE MADRID

ESCUELA TÉCNICA SUPERIOR DE INGENIERÍA Y
DISEÑO INDUSTRIAL

Grado en Ingeniería Electrónica y Automática Industrial

TRABAJO FIN DE GRADO

GEMELO DIGITAL Y SIMULACIÓN
DINÁMICA DE UN ORGANISMO
ROBÓTICO ESCALADOR

Firma Autor

Firma Tutor

Copyright ©año. Nombre del alumno

Esta obra está licenciada bajo la licencia Creative Commons

Atribución-NoComercial-SinDerivadas 3.0 Unported (CC BY-NC-ND 3.0). Para ver una copia de esta licencia, visite <http://creativecommons.org/licenses/by-nc-nd/3.0/deed.es> o envíe una carta a Creative Commons, 444 Castro Street, Suite 900, Mountain View, California, 94041, EE.UU. Todas las opiniones aquí expresadas son del autor, y no reflejan necesariamente las opiniones de la Universidad Politécnica de Madrid.

Título: Gemelo digital y simulación dinámica de un organismo robótico escalador
Autor: Daniel Barroso Saugar
Tutor: Miguel Hernando Gutierrez

EL TRIBUNAL

Presidente:

Vocal:

Secretario:

Realizado el acto de defensa y lectura del Trabajo Fin de Grado el día de de ... en, en la Escuela Técnica Superior de Ingeniería y Diseño Industrial de la Universidad Politécnica de Madrid, acuerda otorgarle la CALIFICACIÓN de:

VOCAL

SECRETARIO

PRESIDENTE

Agradecimientos

Me gustaría expresar mi más sincero agradecimiento a mi familia, tanto mis padres como mi hermano no han dejado de apoyarme en todo momento durante el viaje que ha sido mi paso por la universidad. En este periodo he pasado por momentos mejores y peores, momentos de más estrés y de menos, pero ellos siempre han estado junto a mi para lo que necesitase. Gracias por motivarme a esforzarme más incluso cuando las ganas no eran suficiente combustible.

Durante este periodo he tenido la oportunidad de conocer a gran cantidad de personas increíbles. Quiero agradecer sobre todo a los amigos de la universidad, en especial al grupo de "*Los Rajaos*", que han hecho que este proceso se disfrute más y hemos aprendiendo unos de otros, pudiendo compenetrarnos tanto en estudio como en diversos proyectos.

Por último, y no por ello menos importante, quiero dar las gracias a Miguel Hernando por darme la oportunidad de participar en un proyecto tan interesante. A lo largo de mi estancia en la universidad he adquirido gran cantidad de conocimientos de los cuales algunos no he tenido la oportunidad de poner en práctica, pero gracias a este proyecto he podido hacerlo con varios de ellos a la par que adquirir muchos nuevos.

En resumen, gracias a todos aquellos que me han apoyado y me que han confiado en mí durante esta etapa de mi vida.

Resumen

Este trabajo de fin de grado, consiste en la creación del gemelo digital de ROMERIN, un robot modular diseñado específicamente para la inspección de infraestructuras civiles. Este robot combina características avanzadas de escalada y modularidad, empleando un sistema de ventosas para poder adherirse a superficies. Este robot busca ofrecer una solución más eficiente y segura frente a los métodos tradicionales, que a menudo implican riesgos significativos para los operarios.

El gemelo digital de ROMERIN se ha desarrollado usando el motor de simulación MuJoCo (**MUlti-JOints dynamics with COntact**), que permite replicar con gran precisión la estructura física, las dinámicas de movimiento y los sistemas de control del robot. Gracias a esta simulación, se pueden realizar pruebas y ajustes en un entorno controlado antes de hacer dichas pruebas en el mundo físico. Además, se ha implementado una interfaz gráfica de usuario (GUI) utilizando el framework Qt, que permite la interacción del usuario con el modelo digital y permite controlar y simular el robot de forma intuitiva.

El objetivo principal del proyecto ha sido crear una simulación realista que reproduzca el comportamiento del robot en distintos escenarios. Para ello, fue necesario diseñar las patas modulares del robot, modelar su sistema de ventosas y desarrollar un sistema de control para el robot en la simulación.

En el desarrollo del proyecto, se utilizaron herramientas como MuJoCo para la simulación física, C++ para la programación del modelo y el control del sistema, Python para automatizar tareas como la generación de configuraciones específicas, y XML para definir la estructura del robot y sus características físicas. El conjunto de estas herramientas ha permitido crear un modelo detallado que incluye elementos visuales, sensores, actuadores y cálculos físicos realistas.

Este trabajo, más allá de validar la capacidad de ROMERIN para operar en ciertas condiciones, también establece una base sólida para futuras investigaciones. Entre las posibles líneas de desarrollo están la mejora del código para permitir a la simulación andar, la colaboración del gemelo digital con el robot físico para realizar pruebas híbridas, y la adaptación del sistema a diferentes plataformas y entornos operativos.

Palabras clave: Robot, ROMERIN, Gemelo digital, MuJoCo, Interfaz, Control.

Abstract

This bachelor's degree final project focuses on the creation of the digital twin for ROMERIN, a modular robot specifically designed for the inspection of civil infrastructures. This robot combines advanced climbing capabilities and modularity, using a suction cup system to adhere to surfaces. It aims to provide a safer and more efficient alternative to traditional methods, which often involve significant risks for operators.

The digital twin of ROMERIN has been developed using the MuJoCo simulation engine (**M**ULTi-**J**Oints dynamics with **C**Onact), which enables highly accurate replication of the robot's physical structure, movement dynamics, and control systems. This simulation allows for testing and adjustments in a controlled environment before conducting trials in the physical world. Additionally, a graphical user interface (GUI) was implemented using the Qt framework, enabling user interaction with the digital model and providing an intuitive way to control and simulate the robot.

The main objective of the project has been to create a realistic simulation that reproduces the robot's behavior in various scenarios. This required designing the robot's modular legs, modeling its suction cup system, and developing a control system for the robot within the simulation.

In the course of the project, tools such as MuJoCo were used for physical simulation, C++ for programming the model and system control, Python for automating tasks like generating specific configurations, and XML for defining the robot's structure and physical characteristics. This combination of tools facilitated the creation of a detailed model that includes visual elements, sensors, actuators, and realistic physical calculations.

Beyond validating ROMERIN's capability to operate under certain conditions, this work establishes a solid foundation for future research. Potential areas of development include improving the code to enable walking simulations, integrating the digital twin with the physical robot for hybrid testing, and adapting the system to different platforms and operating environments.

Keywords: Robot, ROMERIN, Digital twin, MuJoCo, Interface, Control.

Índice general

| | |
|------------------------------------------------------------------------------|-------------|
| Agradecimientos | IX |
| Resumen | xii |
| Abstract | xiii |
| Índice | xvii |
| 1. Marco y objetivos del proyecto | 1 |
| 1.1. Marco del proyecto | 1 |
| 1.2. Objetivos del proyecto | 1 |
| 1.3. Estructura del documento | 2 |
| 2. Materiales | 3 |
| 2.1. Introducción | 3 |
| 2.2. Visual Studio Code | 3 |
| 2.3. XML | 4 |
| 2.4. C++ | 4 |
| 2.5. MuJoCo | 5 |
| 2.6. CMake | 5 |
| 2.7. Python | 6 |
| 2.8. QT | 7 |
| 2.9. LaTex | 7 |
| 2.10. TeXstudio | 8 |
| 2.11. GitLab | 9 |
| 3. Estado del arte | 11 |
| 3.1. Introducción | 11 |
| 3.2. Simuladores de robots | 11 |
| 3.2.1. Por qué son esenciales las simulaciones en la robótica [17] | 11 |
| 3.3. Simuladores más populares en investigación | 12 |
| 3.3.1. Webots | 12 |
| 3.3.2. CoppeliaSim | 12 |
| 3.3.3. Gazebo | 13 |
| 3.3.4. MuJoCo | 14 |
| 3.3.5. Isaac SIM | 16 |
| 3.3.6. Comparación entre los simuladores | 17 |

| | |
|----------------------------------------------------------------|-----------|
| 4. ROMERIN | 19 |
| 4.1. Introducción | 19 |
| 4.2. Objetivos del proyecto ROMERIN | 19 |
| 4.3. Estructura | 21 |
| 4.3.1. Cuerpo | 21 |
| 4.3.2. Patas modulares | 22 |
| 4.3.3. Sistema de succión | 23 |
| 4.4. Control y locomoción | 24 |
| 4.5. Actuadores | 25 |
| 4.5.1. Características de los Actuadores | 25 |
| 4.5.2. Aplicaciones de los Actuadores | 26 |
| 5. Modelo XML de MuJoCo | 27 |
| 5.1. MuJoCo en XML | 27 |
| 5.2. Estructura del modelo | 32 |
| 5.2.1. Robot.xml | 33 |
| 5.2.2. Cuerpo.xml | 33 |
| 5.2.3. Pata.xml | 33 |
| 6. Implementación del sistema de mensajes | 35 |
| 6.1. Función executeMessage | 35 |
| 6.2. Mensajes | 36 |
| 6.3. Virtualización de mensajes | 38 |
| 6.3.1. Estructura de los mensajes | 39 |
| 6.4. Comunicación por sockets | 40 |
| 7. Gemelo digital e interfaz | 43 |
| 7.1. Introducción | 43 |
| 7.2. Estructura del proyecto | 43 |
| 7.3. Interfaz gráfica | 47 |
| 7.3.1. Explicación de la introducción de mandatos | 48 |
| 7.4. Interacciones de la interfaz con el modelo | 53 |
| 8. Resultados | 55 |
| 8.1. Simulador de ROMERIN | 55 |
| 8.2. Comparación de los resultados con el robot real | 58 |
| 9. Conclusiones y futuros desarrollos | 59 |
| 9.1. Introducción | 59 |
| 9.2. Desarrollos futuros | 60 |
| 9.2.1. Resolución de errores y propuestas de mejora | 60 |
| 9.2.2. Ajuste para poder trabajar en Windows | 60 |
| 9.2.3. Dotar a la simulación de ROMERIN para andar | 61 |
| 9.2.4. Colaboración de simulación con robot real | 61 |
| Bibliografía | 63 |

| | |
|--------------------------------------------|-----------|
| A. Anexo | 65 |
| A.1. Guía de instalación ROMERIN | 65 |
| A.1.1. Instalación de interfaz | 65 |
| A.1.2. Instalación de simulación | 67 |

Índice de figuras

| | | |
|-------|----------------------------------------------------------------|----|
| 2.1. | Logotipo de Visual Studio Code | 4 |
| 2.2. | Logotipo de Lenguaje de marcado existente (XML) | 4 |
| 2.3. | Logotipo lenguaje C++ | 5 |
| 2.4. | Logotipo MuJoCo | 5 |
| 2.5. | Logotipo CMake | 6 |
| 2.6. | Logotipo Python | 6 |
| 2.7. | Logotipo QT | 7 |
| 2.8. | Logotipo LaTeX | 8 |
| 2.9. | Logotipo TeXstudio | 8 |
| 2.10. | Logotipo GitLab | 9 |
| 4.1. | Robot ROMERIN [23] | 19 |
| 4.2. | Modelado 3D del cuerpo de ROMERIN | 21 |
| 4.3. | Comparación entre brazo humano y pata del robot [23] | 22 |
| 4.4. | Grados de libertad de ROMERIN [23] | 22 |
| 4.5. | Fuerza de succión de la pata[1] | 24 |
| 5.1. | Ejemplo de acceso a información de actuadores | 31 |
| 5.2. | Diseño 3D de una pata | 34 |
| 6.1. | Lazo de control de los motores | 38 |
| 7.1. | Esquema de ficheros de la Interfaz | 44 |
| 7.2. | Parte 1 del esquema del repositorio | 44 |
| 7.3. | Parte 2 del esquema del repositorio | 45 |
| 7.4. | Parte 3 del esquema del repositorio | 45 |
| 7.5. | Parte 4 del esquema del repositorio | 46 |
| 7.6. | Parte 5 del esquema del repositorio | 46 |
| 7.7. | Interfaz de usuario (GUI) de ROMERIN | 48 |
| 7.8. | Menú principal desconectado | 49 |
| 7.9. | Menú principal conectado | 49 |
| 7.10. | Menú de interacción con motores | 50 |
| 7.11. | Sección de interacción con cada motor | 50 |
| 7.12. | Simulación de Romerín en estado estático | 52 |
| 7.13. | Menú de interacción con motores en uso | 52 |
| 7.14. | ROMERIN con una pata elevada | 53 |
| 7.15. | Ventosa en uso | 53 |
| 8.1. | Uso de librerías de Qt en la programación del socket | 56 |

| | |
|--------------------------------------------------------|----|
| 8.2. Bucles de los cuatro módulos | 56 |
| 8.3. Función "sendRegularMessages" | 57 |
| 8.4. Gráfica comparativa de resultados | 58 |
| 9.1. Logo de Windows y Linux | 61 |
| A.1. Configuración de proyecto en Qt Creator | 67 |

Índice de tablas

| | |
|-----------------------------------------------------------|----|
| 3.1. Tabla comparativa de simuladores de robots | 17 |
| 6.1. Listado de mensajes del proyecto ROMERIN | 37 |

Capítulo 1

Marco y objetivos del proyecto

1.1. Marco del proyecto

Este trabajo de fin de grado se desarrolla dentro del proyecto **"ROMERIN"** (RObot Modular EscaladoR para INspección de infraestructuras), que consiste en el desarrollo de un robot modular capaz de escalar superficies para inspeccionar estructuras. El proyecto pertenece al Plan Estatal de Investigación del Ministerio de Economía, Industria y Competitividad.

El robot combina las ventajas de los vehículos aéreos no tripulados (UAVs) y los robots terrestres, pudiendo adaptarse en función de la tarea a realizar. Para la adhesión a la superficie, usa un sistema de succión por ventosa en cada una de sus cuatro patas, en las que se hace vacío por medio de una turbina. Más adelante se profundiza más en este robot y su funcionamiento. [23]

El trabajo descrito en este documento surge de la necesidad de hacer el gemelo digital del robot **"ROMERIN"** a través del motor de simulación de MuJoCo (MULTi-JOints dynamics with COn tact en inglés), siendo un modelo que simule la estructura del robot, pesos y lazos de control de manera realista. Este gemelo digital podrá ayudar en el futuro diversas funciones como la mejora de diseño, optimización, mantenimiento o análisis.

1.2. Objetivos del proyecto

El objetivo de este Trabajo de Fin de Grado (*TFG*) es la generación del gemelo digital del robot ROMERIN para simularlo de forma realista. Además, el programa debe contar con una interfaz hecha en QT para poder comunicarse con el gemelo digital y poder hacer la simulación deseada.

Para poder completar estos objetivos principales ya mencionados, el trabajo se compone de otras tareas secundarias que, tras realizar su conjunto, llegan al final esperado. Estas tareas son:

- Aprendizaje del robot romerín, y del sistema de simulación MuJoCo
- Modelado dinámico del robot. Pruebas de movimiento con enlace directo (XML/C++)
- Modelado de la interfaz de control del gemelo digital. (C++)
- Desarrollo de algoritmos básicos de movimiento del robot mediante el control directo por medio de un programa en Python/C++ y usando la interfaz común al gemelo digital y el sistema real.

1.3. Estructura del documento

Para facilitar la lectura de este documento, se detalla en esta sección la estructura del proyecto:

- **Capítulo 1:** breve introducción de lo que es el proyecto y motivaciones que han dado pie a la necesidad del mismo. También incluye los objetivos previstos.
- **Capítulo 2:** aplicaciones, lenguajes y herramientas utilizadas para realizar este proyecto, así como una explicación de las mismas.
- **Capítulo 3:** estado del arte, exposición de simuladores de robots y una comparativa entre los mismos.
- **Capítulo 4:** Descripción detallada del robot ROMERIN y sus características y capacidades.
- **Capítulo 5:** Descripción detallada de los documentos XML usados en la simulación del gemelo digital de ROMERIN.
- **Capítulo 6:** Explicación de la comunicación entre interfaz y simulación, así como de los posibles mensajes que hay.
- **Capítulo 7:** Explicación general del conjunto de programas y el uso de la interfaz.
- **Capítulo 8:** Resultados de simulación y detalles importantes del proceso para obtener el resultado.
- **Capítulo 9:** Valoración de los resultados obtenidos y de posibles líneas de trabajos futuras.
- **Anexo:** Guías de instalación de ambos programas y las herramientas necesarias para trabajar con ellos.

Capítulo 2

Materiales

2.1. Introducción

En este capítulo se muestran todas las herramientas utilizadas a lo largo de el desarrollo del proyecto. Se incluye una breve explicación para cada una, así como algunas de las aplicaciones en las que han participado.

2.2. Visual Studio Code

Visual Studio Code, también conocido como VSCode, es un editor de código que destaca por varias razones, entre ellas están versatilidad para usar diferentes lenguajes y extensiones, su capacidad para resaltar la sintaxis y detectar errores o su capacidad para auto completar comandos, variables, etc.

Al usar VSCode, automáticamente queda resaltado el código que estemos escribiendo, de tal forma que se pueda distinguir en todo momento y con la mayor rapidez posible qué es cada elemento.

Por lo general, todos los editores de código, incluido VSCode, cuentan con métodos de detección de errores, ya sean causados por fallos en la sintaxis empleada o por otro motivo.

En cuanto al auto completado de código, esta es una ayuda muy útil que proporciona esta herramienta ya que aporta posibles sugerencias o correcciones en tiempo real durante la escritura de código.

En general, este editor de código es muy útil no solo para personas que se inicien en la programación, sino también para profesionales de este sector, ya que permite adaptar la herramienta a nuestro gusto, permitiendo añadir no solo todos los lenguajes que se deseen junto con un amplio número de extensiones. [24]

Para este proyecto, se ha empleado para hacer todo el código empleado en la simulación, no en la interfaz, para esto último se ha empleado otra herramienta que se comentará más adelante.



Figura 2.1: Logotipo de Visual Studio Code

2.3. XML

El Lenguaje de marcado existente (o XML de sus siglas en inglés), permite al usuario almacenar y definir datos, de tal forma que se habilita un intercambio de información entre computación, sitios web y otros entornos similares. Gracias a la estructura y normas predefinidas, hace que sea fácil la transmisión de estos archivos XML a través de cualquier red ya que con estas mismas reglas, es con lo que el que recibe el contenido, puede leer la información.

A diferencia de otros lenguajes, éste no puede realizar cálculos ni tareas de computación por sí mismo, sino que requiere de la cooperación con otros lenguajes de programación para poder tratar los datos que contiene.

Para facilitar el tratado de datos a aquellos lenguajes de programación con los que coopere, XML proporciona símbolos de marcado para que el proceso sea más eficiente. [19]

En el caso concreto de este proyecto, se ha usado este lenguaje para definir el espacio físico de la simulación, incluyendo las patas del robot, el cuerpo del mismo, sus colisiones y físicas (como peso y gravedad) y la réplica del entorno donde se encuentra el robot real en el laboratorio.



Figura 2.2: Logotipo de Lenguaje de marcado existente (XML)

2.4. C++

C++ es un lenguaje de programación orientado a objetos, compilado y multi-paradigma, es un lenguaje imperativo, esto quiere decir que en cada momento de compilación se conoce el estado del programa, así como el valor de sus variables.

EN 1980 se desarrolló C++ como una extensión orientada a C, y tomó este nombre ya que en este último lenguaje existe un operador ++, indicando que es una extensión.

Debido a esto, un código en C puede compilarse en C++. Esto en sus inicios fue una gran ventaja, pero según ha pasado el tiempo, se ha ido convirtiendo más en una debilidad, ya que para mantener esta característica, ha tenido que conservar graves inconvenientes del lenguaje original. [12]

Para este proyecto, C++ se ha empleado en toda la programación relacionada con el simulador MuJoCo y el tratado de información de los documentos XML, además de todo el código de funciones y cálculos junto a interacciones con la interfaz. Se puede decir sin lugar a dudas que ha sido la herramienta más usada.



Figura 2.3: Logotipo lenguaje C++

2.5. MujoCo

Es el motor de simulación escogido para este proyecto, por ello se habla en detalle de éste en la sección 3.3.4, pero cabe recalcar que se ha usado para hacer todo el apartado simulado de ROMERIN, combinándolo con el resto de herramientas para hacerlo funcionar correctamente. [21]

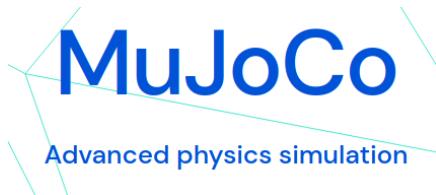


Figura 2.4: Logotipo MujoCo

2.6. CMake

CMake es una herramienta de código abierto ampliamente utilizada para el desarrollo software que define y gestiona cómo se compila un programa. Su principal función es la generación de archivos de construcción (Makefiles) y archivos ejecutables. Además, es una herramienta multiplataforma, lo que permite utilizarla en una amplia variedad de sistemas operativos como Windows, macOS y Linux, garantizando la portabilidad de los proyectos desarrollados con ella.[20]

Visual Studio ofrece compatibilidad nativa con CMake, lo que facilita la edición, compilación y depuración de proyectos basados en CMake tanto en sistemas Windows como en el Subsistema de Windows para Linux (WSL) o entornos remotos, todo desde una única instancia de la aplicación. Esta integración permite a Visual Studio utilizar directamente los archivos de configuración de CMake, como CMakeLists.txt, para proporcionar funciones avanzadas como IntelliSense y exploración de código. Además, el propio Visual Studio se encarga de invocar cmake.exe para llevar a cabo las tareas de configuración y compilación del proyecto. [20]

Para poder obtener los archivos ejecutables de este proyecto, se ha usado CMake, facilitando así su compilación del mismo, ya que en VSCode, algunas librerías concatenadas de MuJoCo daban problemas, pero mediante CMake, se han podido incluir todas.



Figura 2.5: Logotipo CMake

2.7. Python

Python es un lenguaje de programación de alto nivel y orientado a objetos, su uso principal es para el desarrollo web y de aplicaciones. Además, debido a su tipificación, suele usarse en el campo de desarrollo rápido de aplicaciones (RAD).

Una de las características que hacen a este lenguaje de programación tan atractivo, es su simplicidad ya que cuenta con una sintaxis propia que se centra en la legibilidad, esto hace que sea de rápido y fácil aprendizaje.

Otra característica destacable es que puede usar módulos y paquetes, esto quiere decir que algunas partes de los códigos utilizados en un proyecto, tienen mayor facilidad para ser reutilizados en nuevos proyectos. [18]

En este proyecto, para evitar hacer cambios en cada documento XML de las patas, se ha hecho un documento general de patas, y mediante un script hecho en Python, se crean o actualizan los documentos específicos con los nombres concretos.

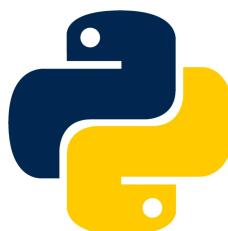


Figura 2.6: Logotipo Python

2.8. QT

Qt es un framework de desarrollo utilizado para crear interfaces gráficas y aplicaciones interactivas que puedan funcionar en múltiples plataformas. Este enfoque hace que el mismo código pueda compilarse en Windows, macOS, Linux, Android o iOS, siendo muy versátil, cualidad muy valiosa en el ámbito del desarrollo de software actual.

Aunque está basado en C++, Qt también incorpora QML, un lenguaje declarativo que simplifica la creación de interfaces dinámicas y su integración y colaboración con otros lenguajes de programación. Esta flexibilidad se ve reforzada por una amplia colección de bibliotecas, APIs y herramientas, siendo también apoyada por una comunidad activa que colabora en su constante mejora.

Además, Qt ofrece un modelo de obtención de licencias flexible, con opciones comerciales y de código abierto, lo que permite adaptarse a distintos perfiles de proyecto y necesidades empresariales. En conjunto, estas características convierten a Qt en una opción sólida y confiable para el desarrollo de aplicaciones profesionales con interfaces de alta calidad. [9]

Este framework se ha usado para hacer la interfaz de usuario(GUI) que permite la interacción con la simulación mediante botones, comunicándose con ella a través de sockets.



Figura 2.7: Logotipo QT

2.9. LaTex

LaTeX es un sistema de composición tipográfica muy utilizado en ámbitos académicos y científicos que permite crear documentos de alta calidad. A diferencia de los editores de texto convencionales, no se basa en el modo “WYSIWYG” (What You See Is What You Get), sino en un lenguaje declarativo que dicta unas normas para establecer la estructura del documento, de esta forma, lo que vemos no va a ser exactamente lo que es el documento final. Esto permite concentrarse en la organización lógica del texto, mientras el programa se encarga de crear una presentación profesional.

Gracias a su naturaleza de software libre y a al hecho de ser multiplataforma, LaTeX está disponible en distintos sistemas operativos y cuenta con una amplia comunidad que contribuye al desarrollo de nuevos paquetes, plantillas y herramientas. Esto facilita la adaptación del documento a las necesidades específicas de cada proyecto.

Este sistema de composición tipográfica, se ha consolidado como un estándar en numerosos campos del conocimiento, ya que ofrece control detallado sobre la apariencia final del documento, gestionando con soltura el contenido como si de fórmulas matemáticas se tratase, referencias bibliográficas, y garantizando siempre una alta calidad en el resultado final.[25]

Este documento se ha hecho gracias a LaTeX y mediante la plantilla proporcionada por la ETSIDI. [14]



Figura 2.8: Logotipo LaTeX

2.10. TeXstudio

TeXstudio es un editor de texto de código abierto cuya función principal reside en facilitar al usuario la creación de documentos LaTeX. Incluye herramientas que hacen más cómoda la escritura, por ejemplo el autocompletado de comandos, el resaltado de sintaxis o el corrector. También permite la inserción de imágenes, fórmulas y creación de tablas, permitiendo así crear documentos complejos. Además, cuenta con un visor de PDF, permitiendo ver en todo momento el resultado final.

En sus inicios, surgió de una bifurcación de Texmaker, pero a día de hoy ha evolucionado hasta convertirse en un software independiente. Está disponible en múltiples plataformas como Windows, macOS o Linux. Este editor se distribuye bajo la licencia GPL v2, lo que garantiza que sea accesible para todos. [26]

TeXstudio ha sido usado en este proyecto para poder redactar el documento actual mediante el lenguaje LaTeX.

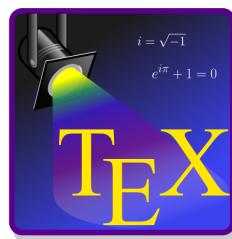


Figura 2.9: Logotipo TeXstudio

2.11. GitLab

GitLab es una plataforma web de código abierto que sirve para facilitar la gestión de todas las etapas del desarrollo de un proyecto de programación de cualquier tipo. Utiliza el sistema Git, lo que permite almacenar y gestionar los proyectos en repositorios remotos. La ventaja principal de GitLab, es que todas las etapas del desarrollo quedan centralizadas en una sola herramienta, permitiendo colaboraciones entre distintos equipos.

Algunas características a destacar de esta plataforma son que tiene integración continua (CI) y despliegue continuo (CD), esto permite la automatización de ciertas fases como la construcción, pruebas y despliegue de aplicaciones. Además, ofrece opciones a la hora de colaborar con otros usuarios como la de inserción de comentarios en el código o el control de tareas. Esto convierte a GitLab en una solución que cubre desde la planificación hasta la entrega del software.

A parte de estas funcionalidades, también incluye otras relacionadas con la seguridad como el control de acceso pudiendo dar más o menos permisos a cada miembro del equipo, desde solo visualizar hasta también poder editar el contenido del proyecto. [22]

Esta herramienta es la que se ha usado para mantener el control de versiones y poder almacenar por seguridad en la nube el proyecto.



Figura 2.10: Logotipo GitLab

Capítulo 3

Estado del arte

3.1. Introducción

Un gemelo digital es una representación virtual de un sistema, hecho para que se comporte y actúe como el sistema real y con la mayor exactitud posible. A partir de datos reales como longitudes o fuerzas, se realiza una simulación que cumpla con lo mencionado anteriormente.

Para poder hacer un gemelo digital, primero hay que hacer un estudio del objeto o sistema del que se va a hacer. En este punto es en el que se toman todas las medidas necesarias para hacer la copia digital.

Tras tener la copia digital, el modelo se puede utilizar para hacer diversas simulaciones, las cuales tienen usos como la prevención de accidentes que pueda sufrir el sistema, ya sea por la falta de fuerza, picos de tensión, o golpes.

3.2. Simuladores de robots

En esta sección se habla sobre la importancia de las simulaciones en el ámbito de la robótica y se habla sobre diversos simuladores, así como de comparativas entre los mismos.

3.2.1. Por qué son esenciales las simulaciones en la robótica [17]

La simulación en robótica emplea gemelos digitales para interactuar de forma dinámica con modelos robóticos en entornos virtuales. Estos sistemas sirven para que haya menos errores y para aumentar la eficiencia en la ingeniería de automatización convencional.

En el campo de la robótica, los sistemas simulados son fundamentales ya que permiten hacer comprobaciones y experimentos que nos permiten ver no solo el resultado final del comportamiento del modelo, sino que también nos permite hacer un estudio del funcionamiento y rendimiento de cada parte que lo forma sin necesidad de gastar tantos recursos como requeriría hacer esto de manera física.

Los simuladores en robótica han evolucionado enormemente, adaptándose a los diversos tipos de robot. Estos robots suelen operar en entornos donde las tareas pueden cambiar con frecuencia, y en el caso de entornos industriales, es común que colaboren con trabajadores en fábricas. Las herramientas de simulación no solo permiten tomar decisiones en tiempo real, como ajustar la velocidad de un movimiento o programar acciones específicas, sino también anticiparse a posibles imprevistos y analizar cómo reacciona el sistema ante ellos.

Para este proyecto, se han valorado varios simuladores, estudiando sus características.

3.3. Simuladores más populares en investigación

3.3.1. Webots

Es un paquete de simulación de robots profesional de código abierto, desarrollado por Cyberbotics. Puede trabajar en Windows, Linux y MacOS. Este paquete ofrece al usuario la posibilidad de creación de entornos en tres dimensiones, tienen sus propias propiedades físicas como la masa, movimiento de articulaciones, coeficientes de fricción, etc. Para los cálculos físicos, utiliza el motor de ODE (Open Dynamics Engine).

Webots trabaja con diversos lenguajes de programación, entre los cuales se encuentran: C, C++, Java, Python, ROS y MATLAB. A pesar de esto, una de las ventajas que tiene, es que si no conocemos ninguno de estos lenguajes, podemos programar los robots e-puck y Hemisson, que únicamente hay que programarlos en un lenguaje de programación gráfica simple llamado BotStudio.

Este paquete de simulación provee al usuario de modelos de robots ya establecidos, pero si no desea usar ninguno de estos, puede crear sus propios modelos o añadir objetos especiales de los entornos simulados. Los modelos pueden ser exportados o importados en formato diversos formatos: URDF, STL, OBJ, o 3D.

Este paquete de simulación destaca por su facilidad de uso y su amplia librería de robots preconfigurados. Además, es compatible con ROS Y ROS 2. Por todo esto, entra dentro de la lista de mejores simuladores de 2024. [13]

3.3.2. CoppeliaSim

Este simulador avanzado de robótica es muy utilizado en investigación, educación, desarrollo de aplicaciones para la robótica. Esto es debido a su gran flexibilidad y capacidad para hacer simulaciones más complejas y que suelen ser necesarias para el campo de la robótica.

CoppeliaSim permite diseñar y probar sistemas en un entorno virtual. Se centra en un enfoque modular en el que cada elemento puede ser configurado y soporta multitud de motores físicos para hacer su simulación, com por ejemplo: Bullet, ODE, Vortex, y Newton. Dentro de estas opciones, el usuario puede usar un simulador

físico u otro en función de la necesidad específica del proyecto.

Este simulador permite una creación sencilla del sistema mediante una interfaz gráfica muy intuitiva, facilitando así la creación de escenas complejas.

Para trabajar con CoppeliaSim, podemos hacerlo a través de diferentes lenguajes de programación, ya sea Lua (lenguaje integrado), Python, C++, Java o MATLAB.

Como otros simuladores, incluye modelos predefinidos de robots industriales, pero se pueden desarrollar nuevos modelos. Además, es compatible con otras herramientas externas como ROS o ROS2.

Algunas de las características que hacen a CoppeliaSim entre dentro de la lista de mejores simuladores de 2024, es que es sencillo de usar, aporta simulaciones realistas, y tiene una versión gratuita. [10]

3.3.3. Gazebo

El Sistema Operativo de Robots (ROS en inglés) es un marco de trabajo de código abierto que proporciona bibliotecas y herramientas diseñadas para fomentar la innovación en el ámbito de la robótica. Desde su lanzamiento en 2007, ROS se ha convertido en una de las bases más utilizadas para el desarrollo de sistemas robóticos, tanto en investigación como en aplicaciones industriales. Su activa comunidad colabora constantemente para mejorar el sistema, permitiendo la integración de proyectos y aplicaciones complementarias en un ecosistema robusto y colaborativo.[16]

ROS está diseñado principalmente para programarse en C++ y Python, lenguajes ampliamente usados y accesibles para la mayoría de desarrolladores. Su versión actual, ROS 2, es compatible con sistemas operativos como Linux, Windows y macOS, además de otras plataformas embebidas. Esto posibilita el desarrollo fluido de soluciones autónomas, interfaces de usuario y gestión back-end.[15]

Uno de los mayores aliados de ROS en el campo de la simulación robótica es Gazebo, un simulador de robots de código abierto que permite a los desarrolladores trabajar en entornos complejos con un alto grado derealismo. Gazebo utiliza motores físicos como ODE, Bullet, Simbody y DART para simular fenómenos como colisiones, fricción y fuerzas gravitatorias. Además, su arquitectura modular basada en plugins facilita la personalización y extensión de funcionalidades según las necesidades específicas de cada proyecto.

Gazebo también destaca por ofrecer una gran cantidad de modelos de sensores y la capacidad de simular ruido, lo que lo convierte en una herramienta ideal para probar algoritmos de control, navegación y percepción antes de implementarlos en hardware real. Su integración con ROS mejora el flujo de trabajo desde la simulación hasta el despliegue, garantizando eficiencia y confiabilidad en el desarrollo robótico.

La colaboración entre ROS y Gazebo ha establecido un estándar en el ámbito de la robótica, combinando la potencia de ROS para el desarrollo de software con

las capacidades avanzadas de simulación de Gazebo. Esto permite diseñar, probar y optimizar robots en escenarios virtuales detallados antes de llevarlos al mundo real, consolidándose como herramientas esenciales tanto para la investigación como para aplicaciones industriales avanzadas. [11]

3.3.4. MujoCo

MujoCo es un motor de simulación gratuito de código abierto, su función es ayudar en la investigación y desarrollo de la robótica, biomecánica y animación, así como en otras áreas en las que sea necesarias simulaciones rápidas y precisas.

Este motor de simulación ofrece a la hora de simular, una combinación de velocidad, poder de simulación muy exacta. Es el primer simulador completo diseñado desde cero con el propósito de estar basado en la optimización de modelos.

MujoCo trabaja con ciertas técnicas computacionales como el control óptimo, identificación de sistemas o el diseño de mecanismos automatizados. Aplica estas técnicas a sistemas dinámicos complejos, también tiene otras aplicaciones como la prueba y validación de esquemas de control antes de su implementación en las máquinas físicas, visualización de sistemas interactivos, animación, o videojuegos.

El nombre de MujoCo viene del inglés Multi-Joints dynamics with Contact. Su función es la investigación y avance en la robótica, biomecánica, animación y machine learning. En sus inicios, estaba desarrollada por la empresa Roboti LLC, pero en octubre de 2021 fue adquirido por DeepMind, y en 2022 se hizo de código abierto. El código de MujoCo se encuentra en su repositorio de GitHub.

Este motor de simulación es una librería de C/C++ compatible con Python y otros lenguajes hecha para los desarrolladores. Está preparado para maximizar el rendimiento y operar en estructuras de datos en bajo nivel mediante la construcción de los objetos mediante documentos MJCF que describen la escena, estos archivos están escritos en lenguaje XML para que sean humanamente legibles. Esta librería incluye visualización interactiva con una GUI (Interfaz Gráfica de Usuario nativa) nativa, mediante OpenGL. También incluye una gran cantidad de funciones que facilitan los cálculos físicos como reacciones con gravedad, choques, etc, para lo cual, usa su propio motor físico.[21]

- Por qué MujoCo en vez de Gazebo:

Desde el comienzo del desarrollo del proyecto, se ha buscado usar las herramientas que aporten mayor funcionalidad al mismo. Este proyecto parte de otro anterior en el que se usó Gazebo en conjunto con ROS 2 como simulador para **ROMERIN**, pero hay ciertas características que han hecho de MujoCo una mejor opción de cara a hacer el gemelo digital.

Cuando se plantea simular un robot escalador de cuatro patas que emplea ventosas para adherirse a superficies verticales o inclinadas, se requiere tanto

una gran precisión como una gran consistencia en la aplicación de las leyes físicas. Las razones por las que se seleccionó este motor de simulación son:

- **Modelado de contacto y fricción:** MuJoCo ofrece un modelo de colisión y fricción muy preciso, perfecto para simular ventosas que se adhieren a la pared, esto es porque su motor está optimizado para múltiples contactos simultáneos y para calcular fuerzas de forma continua y suave. Por otra parte, Gazebo, aunque admite diferentes motores de física (como se ha comentado anteriormente), suele requerir ajustes para representar la fricción y el contacto con exactitud en algunos escenarios. Este desafío puede conducir a inestabilidades o imprecisiones al simular la succión de las ventosas, un factor crítico en este proyecto, ya que es el elemento principal que habilita el desplazamiento en paredes verticales.
- **Desempeño y velocidad de cálculo:** MuJoCo destaca por su eficiencia y estabilidad en cálculos dinámicos, esto lo hace muy popular para entornos de aprendizaje que requieren simulaciones rápidas. Al manejar robots escaladores de alta complejidad, la rapidez y la suavidad de MuJoCo resultan beneficiosas para ello. Gazebo, aunque es muy utilizado en robótica (sobre todo integrado con ROS y ROS 2), puede llegar a presentar cuellos de botella en simulaciones complejas. Esto hace que, para obtener un rendimiento similar, esto hace que sea necesario un control exhaustivo de errores, funciones, variables y procesos que, aun así, no aseguran que se consiga el mismo rendimiento.
- **Control detallado de la dinámica:** MuJoCo tiene herramientas muy robustas de análisis y optimización de la dinámica, facilitando descripciones matemáticas profundas de cada parte de la simulación. Además, la capacidad analítica de las fuerzas que soporta resulta valiosa en caso de necesitar estrategias de control avanzadas, como las de un robot escalador. Mientras tanto, Gazebo tiene una plataforma flexible a través de diversos "plugins" y bibliotecas de física, las cuales no siempre proporcionan la exactitud que se requiere para modelar con precisión la adherencia por succión.
- **Facilidad de configuración y realismo:** Esta es la característica principal que se ha tenido en cuenta para seleccionar este simulador, ya que MuJoCo permite definir de forma relativamente sencilla y detallada las propiedades de los materiales (coeficientes de fricción, elasticidad, amortiguación, límite de fuerzas, etc.) y controlar la estabilidad numérica limitando valores (como en la fuerza de los motores usados en este proyecto), lo que minimiza la necesidad de iteraciones de ensayos para alcanzar los valores de simulación deseados. Por otra parte, Gazebo necesita múltiples iteraciones de prueba y error para ajustar adecuadamente los parámetros de fricción y contacto para conseguir realismo, esto supone un gran coste temporal en algunos proyectos que no disponen de mucho tiempo. En el caso particular de este tipo de robots, esa optimización adicional puede traducirse en tiempos prolongados de configuración y resultados menos exactos en comparación a la precisión y el rendimiento que ofrece MuJoCo.

3.3.5. Isaac SIM

Isaac Sim es una plataforma de simulación enfocada a la roboótica que ha sido desarrollada por NVIDIA, combina el motor de física PhysX con capacidades avanzadas de renderizado en tiempo real. Su objetivo principal es ayudar en la investigación, desarrollo y validación de algoritmos en el ámbito de la robótica siendo apoyado por la inteligencia artificial. Al estar integrado con otras herramientas y tecnologías de NVIDIA (como RTX o CUDA), Isaac Sim hace posible simulaciones masivas.[6]

Además de las capacidades de simulación mencionadas, Isaac Sim tiene herramientas específicas para la generación de datos y de sensores virtuales (cámaras RGB, LiDAR, etc.). Esto permite a los desarrolladores replicar escenarios realistas para el entrenamiento y la validación de algoritmos de percepción. También es compatible con ROS/ROS2, lo que facilita la comunicación entre el software de control de los robots y la simulación.[7]

Su uso se extiende a ámbitos muy diversos, desde proyectos académicos que necesitan probar algoritmos de control complejos hasta aplicaciones industriales donde es importante validar robots móviles o brazos robóticos en entornos complejos. Isaac Sim agiliza la fase de pruebas de prototipos, disminuye la dependencia de hardware y reduce riesgos, lo que lo convierte en una herramienta de gran importancia para la innovación en robótica y automatización.[3]

3.3.6. Comparación entre los simuladores

Estos cuatro simuladores (Webots, CoppeliaSim, Gazebo y MujoCo) permiten recrear un sistema real de forma virtual mediante códigos programados desde cero o implementando librerías ya proporcionadas por los mismos. A continuación se muestra una comparación entre ellos, aportando alguna característica más detallada para mayor detalle.

| | Webots | CoppeliaSim | Gazebo | MujoCo | Isaac Sim |
|---------------------|-----------------------------------------------------|-------------------------------------------------------------------------|-------------------------------------------------------------------|-----------------------------------------------|----------------------------------------------------------------------|
| Enfoque | Investigación y educación | Hacer una simulación versátil y modular para investigación y desarrollo | Simulaciones avanzadas colaborando con ROS | Hacer simulaciones físicas precisas y rápidas | Simulación robótica avanzada con IA y gran rendimiento |
| Licencia | Código abierto | De pago para uso comercial pero gratis para uso educativo | Código abierto | Código abierto | Gratis para usuario individual pero de pago si es para uso comercial |
| Lenguajes | Python, Java MATLAB y C++ | Python, MATLAB, C++ Y Lua | Python, C++, ROS Y ROS2 y plugins propios | Python y C++ | Python |
| Motor físico | Basado en ODE y educación | ODE, Bullet, Vortex y Newton | ODE, Bullet, Simbody y DART | Motor Propio del simulador | NVIDIA PhysX, con simulación basada en GPU |
| Realismo | Bueno para proyectos educativos | Depende del motor elegido pero muy bueno | Depende del motor elegido pero excelente | Superior ya que se diseñó para esto | Excelente, gracias a NVIDIA PhysX |
| Casos de uso | Dibulgación educativa e iniciaciones en la robótica | Prototipado rápido y multi-robot | Control de robots, simulación de sistemas completos y navegación | Biomecánica y control óptimo | Entrenamiento de IA y simulación de robots colaborativos |
| Simulación | De alta calidad e intuitiva | De alta calidad y personalizada | Menos detallada pero funcional | Más básica pero orientada al estudio de datos | Gráficos de alta calidad con Ray Tracing |
| Dificultad | Baja, de fácil acceso para principiantes | Media, necesita de conocimientos básicos de simulación | Alta ya que necesita familiaridad con ROS y sistemas distribuidos | Alta, está orientado a usuarios avanzados | Alta, necesita experiencia en IA y en programación avanzada |

Tabla 3.1: Tabla comparativa de simuladores de robots

[8], [2], [4], [5], [6]

Teniendo en cuenta todas estas características, se ha escogido MujoCo por su precisión a la hora de hacer simulaciones físicas, su velocidad de simulación que permite realizar grandes simulaciones (debido a su gran optimización), la sencillez a la hora de describir sus modelos en XML y por su adaptación para este proyecto, ya que se emplea principalmente para aplicaciones de biomecánica.

Capítulo 4

ROMERIN

4.1. Introducción

Este proyecto, como se ha comentado anteriormente en este documento, consiste en hacer un gemelo digital del robot ROMERIN. Este es un robot cuya funcionalidad es la del mantenimiento de infraestructuras civiles, ya que en ocasiones los métodos tradicionales pueden suponer peligros o ciertos riesgos significativos para los operarios que desempeñen esta función. El robot ROMERIN combina modularidad con una capacidad de escalada de tal forma que proporciona una solución eficiente y segura que permita operar en condiciones adversas.

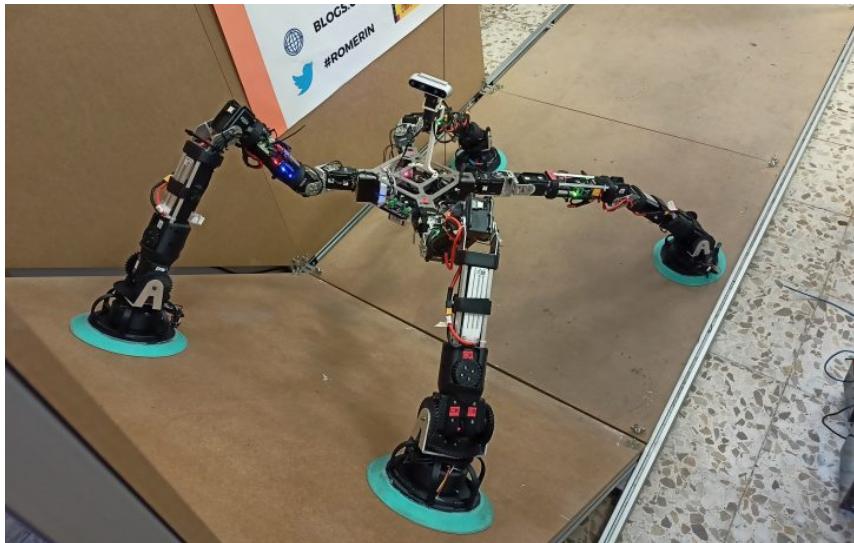


Figura 4.1: Robot ROMERIN [23]

4.2. Objetivos del proyecto ROMERIN

El objetivo principal del proyecto, es construir un robot que logre una plena modularidad en sus aspectos mecánicos, energéticos y de control. Para ello se pretende lograr:

- **Diseñar patas modulares completamente autónomas:** Esto quiere decir que cada pata es considerada como un robot independiente. Este diseño modular aumenta la flexibilidad del sistema, permitiendo a cada pata moverse y gestionar sus propios recursos independientemente del número de patas utilizadas en el organismo robótico.
- **Incorporar un método de adhesión a superficies robusto:** Para ello, cada módulo (pata) consta de su propio sistema de succión, basado en una ventosa que se adhiere a la superficie por medio de una turbina que genera vacío (hay una por ventosa). El uso de turbinas, en lugar de bombas convencionales mejora la capacidad de adhesión en superficies irregulares o con fisuras ya que desplaza grandes cantidades de aire. Este enfoque también minimiza la pérdida de presión generalizada en el robot cuando una ventosa encuentra fugas de aire, garantizando una mayor fiabilidad en la adhesión.
- **Crear una arquitectura de control:** Dado que se compone de cuatro módulos individuales, es necesario conseguir un sistema de control capaz de gestionar tanto la información proveniente de cada uno, como la que se les envía, para conseguir que el conjunto funcione como uno solo. Para conseguirlo, la arquitectura MoCLORA (Modular Climbing-and-Legged Robotic Organism Architecture) ha sido desarrollada para adaptarse a la variabilidad del número de patas y a las diferentes configuraciones del robot. Esta arquitectura incluye capas de control de bajo y alto nivel para coordinar tanto los movimientos del cuerpo como la locomoción del robot en superficies complejas.
- **Garantizar eficiencia:** Tanto energéticamente como en la distribución óptima de carga. Esto unido a un buen método de adhesión a la superficie, garantiza una gran seguridad para la integridad de la máquina(evitando caídas o un mal funcionamiento). Además, se ha implementado un método novedoso de compensación de gravedad, que permite optimizar la fuerza que realiza cada actuador y mejorar la eficiencia del sistema, reduciendo en gran medida los errores de posición que se pueden dar durante la locomoción.
- **Desarrollar herramientas de simulación:** Aquí es donde entra este trabajo de fin de grado. Es necesario obtener herramientas de simulación para comprobar todo aquello que se ha mencionado en el punto anterior. Para garantizar cierta seguridad en el proyecto, antes de su puesta en marcha física, es necesario primero obtener datos teóricos lo más similares a la realidad posible. De esta manera se llegan a evitar numerosos riesgos para la integridad de la máquina, como precipitarse desde una superficie en la que esté adherida. El entorno de simulación incluye opciones de prueba rápida de los algoritmos.

4.3. Estructura

ROMERIN está compuesto por un cuerpo principal y cuatro patas modulares con sus ventosas propias, diseñadas específicamente para la inspección de infraestructuras complejas y evitar posibles riesgos a trabajadores humanos. Además, cuenta con dos patas adicionales que pueden utilizarse como repuestos en caso de mal funcionamiento, ofreciendo flexibilidad operativa en caso de fallos mecánicos. Esta configuración modular permite una rápida reconfiguración y mantenimiento en campo, lo que mejora significativamente la operatividad del sistema.

4.3.1. Cuerpo

El cuerpo del robot ROMERIN actúa como un nexo central para las patas modulares, asegurando una distribución uniforme del peso y proporcionando un centro de control robusto para todos los componentes del robot (los módulos, sensores, cámara, etc). Está equipado con una unidad de control, que incluye un ordenador de alto rendimiento encargado de procesar los datos provenientes de los módulos y gestionar las comunicaciones internas y externas.

Entre los componentes más destacados del cuerpo se encuentran los sistemas de comunicación inalámbrica y las interfaces de sensores para monitorización en tiempo real, esto hace que el operador pueda obtener datos precisos sobre el estado del robot, su posición y el entorno que le rodea mientras está en uso. Además, el cuerpo aloja una batería centralizada de alta capacidad que no solo suministra energía a todas las patas, sino que también asegura una gestión eficiente de la energía, optimizando la autonomía del robot para que pueda realizar operaciones prolongadas.

El diseño del cuerpo también incluye puntos de fijación reforzados que garantizan una conexión fuerte y segura con las patas modulares, incluso en condiciones adversas, como en superficies inclinadas, verticales o bajo fuerzas gravitatorias.



Figura 4.2: Modelado 3D del cuerpo de ROMERIN

4.3.2. Patas modulares

Cada pata modular ha sido diseñada como una unidad autónoma que combina eficiencia, flexibilidad y robustez.

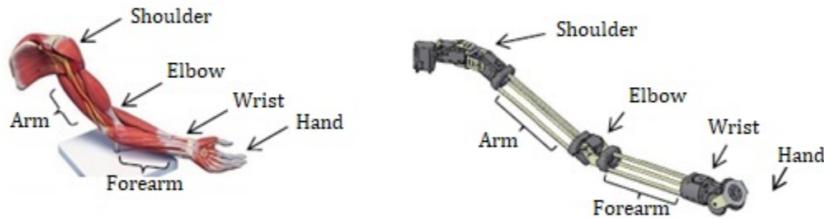


Figura 4.3: Comparación entre brazo humano y pata del robot [23]

Estas patas cuentan con seis motores de alta precisión que proporcionan seis grados de libertad (DoF), lo que les permite realizar movimientos complejos y adaptarse a geometrías variables del terreno. Esta configuración facilita el movimiento en superficies como paredes inclinadas, verticales y techos, en terrenos rugosos o irregulares, así como superficies de diversos materiales. Además, cada pata dispone de sensores que recogen el ángulo de cada articulación, la velocidad con la que se mueven.

Cada pata está formada por 6 piezas principales que pesan 212, 360, 535, 205, 125 y 287 gramos respectivamente y con un peso total de 1724.

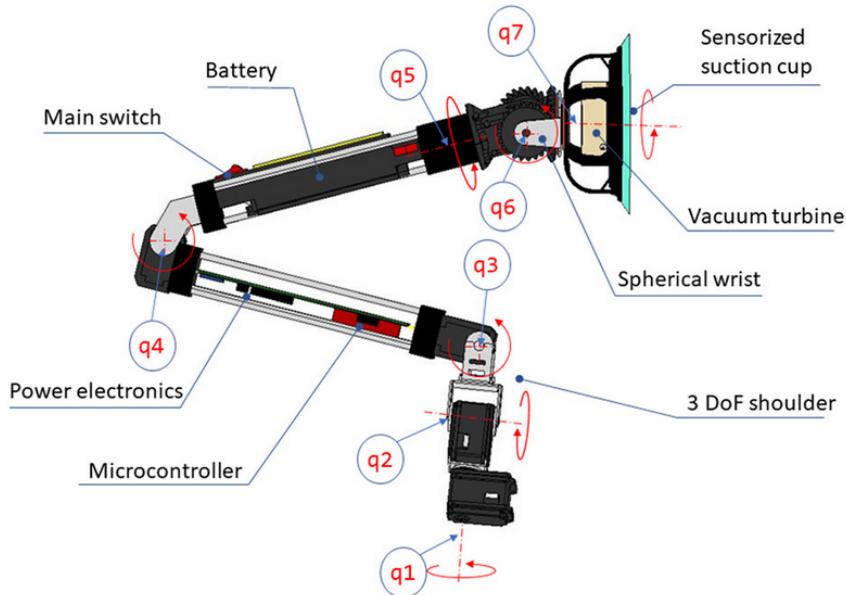


Figura 4.4: Grados de libertad de ROMERIN [23]

Como se ve en la imagen anterior, hay un motor que actualmente no existe, es el representado por q1, esto es debido a que tras varias pruebas de diseño se decidió eliminar de manera que, como se ha comentado, actualmente cada módulo dispone

de 6 grados de libertad.

El diseño de las patas incluye materiales ligeros y resistentes, lo que reduce la carga generada por la estructura del robot mientras garantiza su durabilidad y resistencia a golpes o estrés. Inicialmente, cada pata contaba con su propia batería, lo que le permitía operar de forma independiente y distribuir energía entre otros módulos en caso necesario. Sin embargo, tras diversas pruebas de campo, se optó por un sistema centralizado de energía que reduce el peso total y mejora la autonomía operativa.

Gestión inteligente de recursos: Las patas están equipadas con sensores que monitorizan su estado en tiempo real, incluyendo la posición de cada articulación, el torque de los motores y la adherencia de las ventosas. Estos datos se envían al cuerpo central para prevenir fallos durante las operaciones y poder controlar a ROMERIN.

4.3.3. Sistema de succión

El sistema de succión de ROMERIN se basa en una solución innovadora que utiliza turbinas independientes para cada ventosa, en lugar de bombas de vacío convencionales. Esta elección de diseño tiene varias ventajas: mueve un flujo constante de aire que mejora la adherencia en superficies irregulares y evita que una fuga comprometa la estabilidad del sistema.

- **Ventosas de alto rendimiento:** Cada ventosa ha sido diseñada para adherirse a materiales muy distintos, incluyendo metal, vidrio y hormigón, siempre que cumplan con requisitos básicos como una superficie que sea lo suficientemente plana y con una textura compatible. Estas ventosas también son altamente resistentes a condiciones como humedad, polvo y cambios de temperatura, lo que las hace ideales para inspecciones en exteriores y ambientes industriales(lugares donde hay gran presencia de polvo en superficies y en suspensión).
- **Tolerancia a fallos:** Si una pata pierde contacto con la superficie, el sistema de succión garantiza que el resto de las patas mantengan la adhesión, evitando caídas o daños al robot. Además, hay un sistema de monitorización de presión, esto permite detectar fugas y ajustar automáticamente el flujo de aire para mantener la succión indicada de manera constante.
- **Flexibilidad operacional:** Este sistema de succión también permite al robot adaptarse a diferentes grados de inclinación, desde superficies horizontales hasta techos.

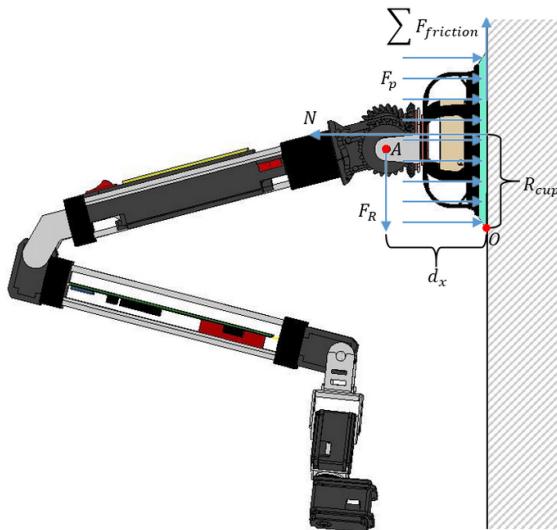


Figura 4.5: Fuerza de succión de la pata[1]

4.4. Control y locomoción

Lo más complicado de este proyecto es conseguir un control que permita un desplazamiento no solo horizontal, sino también vertical y diagonal, compenetrando simultáneamente la gestión del movimiento de los cuatro módulos acoplados con la información que brindan los tres sensores de posición que hay en cada ventosa, gracias a los cuales el sistema de control detecta a qué distancia se encuentra la superficie. Este control es especialmente complicado debido a las limitaciones de los robots escaladores y modulares.

Para conseguir esto, es necesario tener en cuenta que:

- **Tiene que haber una compensación de la gravedad:** Debe calcular las fuerzas de par necesarias para compensar los efectos de la gravedad y las interacciones que tiene con el entorno. Además, debe tener esto un bajo coste computacional para lograr un mayor rendimiento. La propuesta de este proyecto utiliza un método basado en problemas de distribución de fuerzas (*Force Distribution Problem*, FDP), que calcula las fuerzas de reacción en los puntos de contacto sin necesidad de sensores de fuerza/torque, reduciendo así el coste del sistema.
- **Hay que calcular las trayectorias:** Para cada módulo es necesario calcular el giro que deben hacer cada uno de sus motores de tal forma que su conjunto dote a la pata del movimiento requerido. Para calcular estas trayectorias hay que determinar el mejor punto de apoyo posible para entrar en contacto con la superficie y así prevenir colisiones y maximizar la distancia recorrida en los casos que sea necesario. Este proceso incluye un planificador de trayectorias y puntos de apoyo. Dichos planificadores aseguran la estabilidad del robot mientras optimizan la energía consumida por los actuadores y mejoran la seguridad

en la adhesión de las ventosas.

- **Planificación de la marcha:** La determinación de qué pata debe moverse en cada momento es muy importante ya que se debe mantener el equilibrio y la eficiencia energética. En este proyecto, se ha usado un controlador de marcha bioinspirado que elige la pata más que debe iniciar el siguiente movimiento en función de su estabilidad y distribución de carga.

El control implementado permite un intercambio fluido entre el robot físico y su gemelo digital, facilitando la validación y mejora del sistema mediante simulaciones rápidas y precisas.

A la hora de gestionar los motores del robot, deben hacerse ciertas conversiones de ángulos. Esto se debe a que cada motor funciona con ángulos relativos al motor anterior (sin contar con el primero de ellos), mientras que en la interfaz, estos ángulos no se muestran de esta forma, por ello son necesarias dos funciones encargadas de realizar dichas conversiones, se llaman " $m2q$ " y " $q2m$ ", la primera se encarga de traducir al programa de control los ángulos que le indica la interfaz y la segunda hace la función inversa, indicándole a la interfaz qué ángulo mostrar. Además, es necesario hacer otro tipo de conversiones más ya que los dos últimos motores tienen movimientos dependientes el uno del otro, de tal forma que si uno se mueve, el otro lo hará también.

4.5. Actuadores

Los actuadores han tenido gran importancia en el diseño y funcionamiento del robot para lograr que un organismo modular tenga la capacidad de desplazarse y escalar superficies. Estos elementos son responsables de generar el movimiento necesario para coordinar las patas modulares de manera simultánea y asegurar el desplazamiento eficiente en superficies complicadas, incluyendo terrenos horizontales, paredes verticales e incluso techos.

4.5.1. Características de los Actuadores

Los actuadores empleados en ROMERIN son motores eléctricos de corriente continua (DC) diseñados para ser óptimos en cuestión de potencia, precisión y eficiencia energética. Algunas de sus características principales incluyen:

- **Alto par motor:** Los actuadores están diseñados para manejar cargas significativas y superar fuerzas gravitatorias en configuraciones de escalada. Esto es esencial para mantener la estabilidad del robot en superficies inclinadas o verticales.
- **Bajo peso:** Dado que ROMERIN se basa en un diseño modular, cada pata debe ser autónoma y ligera para no comprometer la movilidad del robot. Los actuadores han sido seleccionados para minimizar el peso total del sistema.

- **Control de precisión:** Gracias a los "encoders", los actuadores permiten un control preciso de posición y velocidad, lo que es fundamental para la coordinación de las diversas patas del robot.
- **Mecánica:** Los actuadores están situados en cada módulo de la pata, permitiendo que cada articulación tenga un rango concreto de movimiento:
 - Link 2: $[-110^\circ, 110^\circ]$
 - Link 3: $[-110^\circ, 110^\circ]$
 - Link 4: $[-35^\circ, 170^\circ]$
 - Link 5: $[-180^\circ, 180^\circ]$
 - Link 6: $[-90^\circ, 90^\circ]$
 - Link 7: $[-180^\circ, 180^\circ]$

Estos rangos aseguran que las patas puedan alcanzar posiciones seguras que garanticen la adhesión y el equilibrio del robot durante el movimiento.

- **Capacidad de torque:** Los actuadores son capaces de generar el torque necesario para desplazarse.
- **Eficiencia energética:** Cada actuador está diseñado para trabajar de forma eficiente aprovechando la batería, lo cual permite mantener la autonomía del robot.

4.5.2. Aplicaciones de los Actuadores

Los actuadores del robot ROMERIN tienen múltiples aplicaciones dentro del sistema, destacando las siguientes:

- **Locomoción:** Controlan los movimientos de las patas para realizar patrones de marcha bioinspirados, permitiendo al robot adaptarse a diferentes entornos y condiciones.
- **Compensación gravitatoria:** Mediante un método eficiente de compensación de gravedad, los actuadores garantizan que las patas soporten las fuerzas necesarias para mantener la posición estable del robot mientras escala.
- **Generación de succión:** Los actuadores trabajan junto a las ventosas para aplicar la fuerza necesaria que permita al robot adherirse a las superficies.
- **Planificación de trayectorias:** Gracias a su precisión y capacidad de respuesta, los actuadores permiten ejecutar trayectorias que mejoran la distribución de la carga entre los módulos.

Capítulo 5

Modelo XML de MujoCo

En este capítulo se va a describir y explicar todo el contenido de los documentos XML encargados de la descripción gráfica y física de la simulación. Además, se comentará la estructura que se ha seguido en estos documentos para facilitar la comprensión de los mismos.

5.1. MujoCo en XML

Los motores de simulación, como ya se ha mencionado, necesitan de una descripción gráfica y física para saber cómo es el sistema que deben simular en el lenguaje correspondiente. En el caso de MujoCo, usa el formato XML para hacer esta descripción. En este apartado se va a describir los comandos usados para realizar la simulación de ROMERIN.

MujoCo puede unir varios documentos distintos para simular todo un sistema, pero en el principal, todo el código debe estar comprendido en:

```
<mujoco model="Nombre_del_modelo">
  ...
</mujoco>
```

Dentro de este modelo, al comienzo del apartado ”mujoco model=...”, se puede indicar el compilador, por ejemplo el *eulerseq*=”XYZ”, pero hay que tener en cuenta que esto no es necesario ya que si no se le indica, toma uno por defecto. Tras esto también se pueden crear clases dentro del apartado ”default” como por ejemplo:

```
<compiler eulerseq="XYZ"/>
<default>
  <default class="nombre_de_la_clase"/>
</default>
```

Para poder incluir ficheros externos como imágenes, texturas o materiales para ciertos elementos del sistema simulado, existe el apartado de "assets", donde se clasifican sus nombres y la dirección donde se aloja el documento externo.

Dentro de los ficheros que se pueden añadir en este apartado, se encuentran los "meshes", que son diseños en tres dimensiones que pueden usarse únicamente como elementos visuales o también como elementos de colisión. Un ejemplo de este apartado es:

```
<asset>
  <texture name="grid" type="2d" builtin="checker" rgb1="0.1 0.2 0.3"
    rgb2="0.2 0.3 0.4" width="512" height="512"/>
  <material name="grid" class="unused" texture="grid" texrepeat="1 1"
    texuniform="true" reflectance="0.2"/>
  <mesh name="link1CGEnv" class="unused"
    file="../Pata/meshes/link1CGEnv.stl" scale="0.001 0.001 0.001"/>
  ...
</asset>
```

Estos apartados son los previos antes de la descripción gráfica y física que se ha mencionado anteriormente, esto se incluye en el siguiente, el cuerpo conjunto o "worldbody". En el cuerpo conjunto es donde se incorporan los diferentes "bodys" o cuerpos individuales que van a componer al sistema. Cada uno de ellos puede contener a uno o varios más, para indicar esto y que sea más visual, se suelen aplicar tabulaciones para que se vea más fácilmente, pero basta con que se encuentren entre la apertura y el cierre del "body" padre para quedar anidados. Teniendo un cuerpo padre y uno o más hijos, se unen mediante "joints" o articulaciones entre ambos. Estas articulaciones se escriben dentro del cuerpo padre y se unen a éste mismo con los que se encuentren en el nivel inmediatamente inferior. A estas articulaciones ("joints") hay que indicarles en qué posición se encuentran, la orientación (ángulos de Euler) y con respecto a qué eje giran (si lo hace dependiendo del tipo de articulación), así como los límites de giro.

```
<asset>
  <body name="nombre" pos="0 0 0" euler="0 -0 0">
    <geom name="visual" type="mesh" contype="0" conaffinity="0" group="0"
      pos="0 0 0" euler="-90 -0 0" mesh="link1CGSimple" material="Marron"/>
    <geom type="capsule" size="0.0313 0.0001" friction="1 0.005 0.00015"
      contype="1" conaffinity="1" pos="0 0 0" euler="0 -0 0" rgba="1 1 1 0"/>
    <joint name="Q1" type="hinge" pos="0 0 0" axis="-1 0 0" limited="true"
      stiffness="0.2" range="-1 1" damping="1" frictionloss="0.5"/>
    <body name="link2" pos="-0 -0.001 0.0596" euler="-0.046 0 0.086">
      ...
    <\body>
  <\body>
</asset>
```

Atributos de los comandos del ejemplo mostrado:

- **Geom:** Define una geometría, proporcionando tanto las propiedades visuales como las de colisión del cuerpo:
 - **name:** Nombre de la geometría.
 - **type:** Tipo de geometría.
 - **mesh:** Es el modelo externo incluido en los "assets".
 - **size:** Es el vector que define las dimensiones de la geometría.
 - **friction:** Coeficiente de fricción del elemento, cuyas tres componentes equivalen a la tangencial, rotacional y rodadura respectivamente.
 - **contype:** Declara los contactos que puede tener el elemento.
 - **conaffinity:** Establece las interacciones que la geometría puede tener con otras.
 - **material:** Es el material visual que se ha declarado previamente en los "assets".
 - **rgba:** Le otorga valores y apariencia a la geometría, indicando las mezclas del rojo, verde, azul y si tiene apariencia o no respectivamente.
- **Joint:** Elemento que vincula dos o más cuerpos a un cuerpo padre.
 - **name:** Nombre de la articulación.
 - **type:** Tipo de articulación (en concreto en el ejemplo es "hinge", lo que indica que solo tiene un eje de giro).
 - **pos:** la posición donde se encuentra la articulación entre dos cuerpos.
 - **axis:** Indica el eje de giro x, y o z respectivamente.
 - **limited:** Si este atributo es "true", quiere decir que el movimiento de esta articulación está limitado, mientras que si es "false", no.
 - **stiffness:** Es la rigidez de la articulación.
 - **range:** El rango de movimiento que se le ha dado.

- **damping:** Es el coeficiente de amortiguación que se le aplica al movimiento.
- **frictionloss:** Pérdida de energía debido al giro de la articulación (causada por la fricción).

Dentro de los "bodys" se incluyen los "geoms", que son las formas que componen el sistema, estas formas pueden ser tanto unas predeterminadas por MuJoCo como cubos o esferas, como los meshes mencionados anteriormente.

```
<geom name="visual" type="mesh" contype="0" conaffinity="0" group="0"
pos="0 0 0" euler="-90 -0 0" mesh="link1CGSimple" material="Marron"/>
```

Este "geom", como se puede apreciar contiene un "mesh" y un material, mientras que el siguiente ejemplo es una forma tridimensional predeterminada por MuJoCo, una cápsula.

```
<geom type="capsule" size="0.0372 0.06" friction="1 0.005 0.0001"
contype="1" conaffinity="1" pos="2.168e-19 -0.015 0" euler="-90 0 0"
rgba="1 1 1 0"/>
```

Para determinar si un "geom" pertenece al grupo de elementos visuales o al de elementos de colisión, están los atributos "contype" y "conaffinity". Ambos atributos son máscaras de bits que si indican "0" o al menos uno de ellos lo indica, el elemento no tendrá colisiones. Si su valor es 1 o más y coincide con los de otra geometría, ambas podrán chocar. En resumen, son dos atributos que van de la mano para declarar si hay o no colisiones

Además, se pueden definir sitios concretos del espacio o "sites" que pueden ser utilizados por articulaciones o actuadores, pero estos últimos se van a explicar más en detalle posteriormente en este capítulo.

```
<site name="force_torque_Q3" pos="0 -0.1271 0.00113" euler="90 -0 0"/>
```

Estos son los elementos que se incluyen en el "worldbody", el siguiente grupo de elementos es el de actuadores. Estos pueden ser de diversos tipos, en el caso de este proyecto se han usado únicamente motores y ventosas. A este tipo de elementos se les introduce el "site" o "body" donde se desea que hagan efecto (en el caso de los motores, en "sites" colocados en articulaciones).

```
<actuator>
<motor name="motor_Q_THOR_1" joint="Q_THOR_2" ctrlrange="-7.3 7.3"
ctrllimited="true"/>
<adhesion name="Ventosa_THOR" body="Base_THOR" ctrlrange="0 117.72"
```

```
    gain="117.72"/>
<\actuator>
```

Explicación de los atributos mostrados en el ejemplo anterior:

- **motor:** define un actuador de tipo motor que aplica cierta fuerza torque.
 - **name:** Nombre del motor.
 - **joint:** Articulación en la que hace efecto la fuerza torque que aplica este motor.
 - **ctrlrange:** Define el rango de los valores de fuerza que puede aplicar el actuador.
 - **ctrllimited:** Si vale "true" la fuerza queda limitada al límite establecido, si vale "false", no.
- **adhesion:** define un actuador que aplica una fuerza de succión a la superficie que entra en contacto con él.
 - **name:** Nombre del succionador.
 - **body:** Cuerpo al que se le aplica esta característica (la succión).
 - **ctrlrange:** Como en el motor, el rango de fuerza que puede hacer.
 - **gain:** La fuerza de succión que hace el actuador.

Es importante tener en cuenta que al crear la sección de actuadores, el modelo que crea MuJoCo aparece un vector de punteros almacenado en "mjData" llamado "ctrl" en el que cada dirección del vector accede a un actuador distinto. El orden con el que se almacenan los actuadores en este vector es en orden de lectura, de tal forma que en el ejemplo mostrado, el motor tendría la dirección cero y la ventosa la dirección uno, si se hubiera añadido otro tras ésta, la dirección tres y así sucesivamente. Esta información es muy importante ya que en el código de C++, ésta es la forma que hay para acceder a esta información y trabajar con ella. Un ejemplo de cómo se accede a la información de los actuadores en el código de C++ donde "num" equivale al motor con el que se quiere trabajar es:

```
modelData->ctrl[num] = torqueForce; // Se da la fuerza al motor
```

Figura 5.1: Ejemplo de acceso a información de actuadores

Por último, la sección de sensores ofrece numerosas opciones, pero en este proyecto se han usado sensores de fuerza, presión y velocidad.

```
<sensor>
  <force name="force_force_torque_Q_THOR_2" noise="0"
    site="force_torque_Q_THOR_2"/>
  <torque name="torque_force_torque_Q_THOR_2" noise="0"
    site="force_torque_Q_THOR_2"/>
  <touch name="pressure_sensor_FRIGG" site="baseSuccionador_FRIGG"/>
<\sensor>
```

Explicación de los sensores y sus atributos:

- **force:** define un sensor que detecta la fuerza lineal aplicada a un cuerpo en un punto específico.
 - **name:** Nombre del sensor.
 - **noise:** Define el nivel de ruido (que en la simulación consiste en una variación aleatoria de valores).
 - **site:** Define el punto concreto donde se quiere medir la fuerza.
- **torque:** Define el sensor que mide la fuerza torque aplicada en un punto concreto, en este caso en las articulaciones.
 - **name:** Nombre del sensor.
 - **noise:** Define el nivel de ruido (que en la simulación consiste en una variación aleatoria de valores).
 - **site:** Como en el sensor anterior, pero en este caso el site debe ser una articulación concreta.

Una vez se han comprendido los comandos utilizados en el proyecto, se va a explicar la estructura del modelo.

5.2. Estructura del modelo

El proyecto se divide en tres documentos: "Robot.xml", "Cuerpo.xml" y "Pata.xml". "Robot.xml" es el documento principal que importa "Cuerpo.xml" y los documentos especificados de "Pata.xml" ("Thor.xml", "Loki.xml", ...). Este documento es el que se llama en el programa de C++ para iniciar la simulación y del

que se extrae la información con la que se va a trabajar.

5.2.1. Robot.xml

Como se ha mencionado ya, éste es el documento principal, contiene configuraciones para la simulación total del robot, actuadores, sensores y todos los "meshes", materiales y texturas que necesitan los elementos del sistema.

En el "worldbody" es donde se incluyen los elementos externos de cuerpo y patas. Estas están todas en el mismo nivel de tabulación pero en el inmediatamente posterior al del cuerpo para mayor claridad visual. Al estar contenidas las patas por el "body" del cuerpo y sin darles rango de movimiento, da el resultado como si quedasen soldadas en su punto de contacto con el mismo.

5.2.2. Cuerpo.xml

Este es el documento de menor tamaño, contiene tres elementos visuales y tres de colisión superpuestos con los anteriores que tienen su visibilidad desactivada. Estos tres elementos pertenecen al cuerpo principal, a una cámara en la parte superior y al soporte que une este último con el primero.

Una imagen de la simulación de este fichero es 4.2.

5.2.3. Pata.xml

Este documento es genérico y sirve de plantilla para, mediante un script en Python, especificarlo cambiando los nombres para sus cuatro patas ya que todas deben ser idénticas y estar formadas de los mismos elementos (THOR, ODIN, LOKI y FRIGG).

Su estructura es notablemente más compleja que los documentos anteriores, conteniendo no solo las piezas de la pata sino también los succionadores.

Contiene numerosos "bodys" uno dentro de otro y unidos por "joints", de las cuales la primera tiene el fin de unir cada pata al cuerpo. Dentro de cada uno, aparecen los "geoms" visuales y de colisión correspondientes a cada pata, pero también se indican "sites" en las articulaciones para que "Robot.xml" pueda saber en qué punto debe incluir un actuador.

En cada "body" aparece un elemento que define las coordenadas iniciales, donde se incluye la posición relativa a la pieza donde se aplica y también se le indica la masa, ambos son atributos de gran importancia para los cálculos físicos de MuJoCo.

```
<inertial pos="0 0 0" mass="1" fullinertia="1 1 1 0 0 0"/>
```

En el código mostrado, el atributo "fullinertia" representa a la matriz de inercia:

$$\mathbf{I} = \begin{bmatrix} I_{xx} & I_{xy} & I_{xz} \\ I_{xy} & I_{yy} & I_{yz} \\ I_{xz} & I_{yz} & I_{zz} \end{bmatrix}$$

$$\mathbf{I} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

Esta matriz significa que el cuerpo al que pertenece tiene la misma resistencia al movimiento en los ejes X, Y y Z, indicando que los ejes principales de inercia están alineados con los locales del cuerpo.

Por último, queda el succionador, que contiene los "geoms" que lo definen, pero en su parte inferior contiene un cilindro de muy poca altura que sirve a modo de base, en ella es donde se aplica el actuador de succión definido en el documento principal. Junto a estos elementos, el succionador incluye sensores de distancia cuya función es la de ver cuánto tiene que descender verticalmente la pata para entrar en contacto con el suelo, esto se usarán en un futuro para hacer que la simulación de ROMERIN pueda andar.

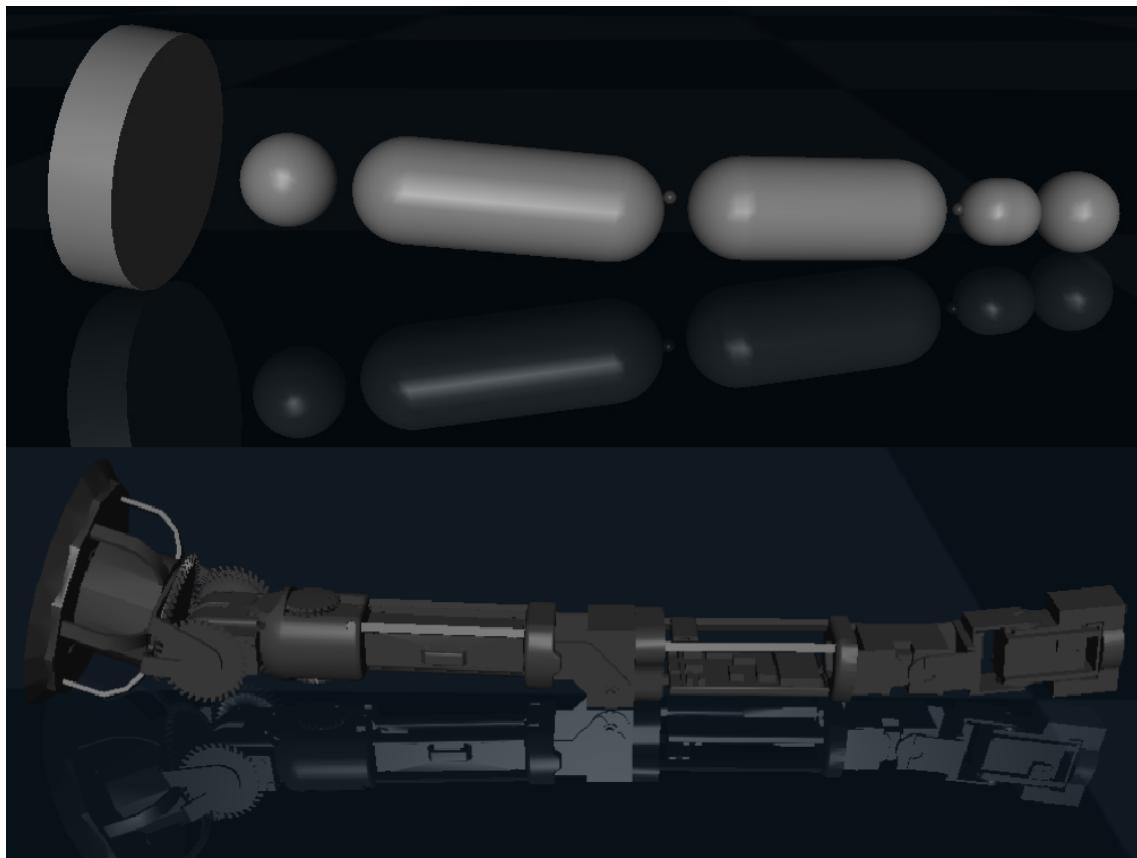


Figura 5.2: Diseño 3D de una pata

Capítulo 6

Implementación del sistema de mensajes

Como se ha comentado anteriormente, este proyecto se compone de dos programas, la interfaz gráfica de usuario (GUI) y la simulación, que interactúan entre ellos por medio de mensajes a través de sockets como el medio por el que pasa el flujo de información.

Estos mensajes están definidos desde el principio del código, pero es solo una función la que gestiona el tipo de mensaje que se manda o recibe, esta función se llama "executeMessage" y en la simulación se encuentra en la clase "ModuleSimulator".

6.1. Función executeMessage

En la interfaz, tiene la función de mandar un mensaje concreto a la simulación según el botón que se haya pulsado, sin embargo en la simulación funciona de manera complementaria a esta, pudiendo mandar y recibir un mensaje y clasificándolo dependiendo de lo que recibe.

"executeMessage" tiene como entrada un puntero de tipo "RomerinMsg", éste es el mensaje que recibe de la interfaz o que manda a la misma y el que hay que clasificar, todo para indicarle al resto del programa qué hay que hacer en la simulación. Una vez se clasifica el mensaje con un "switch case", se deriva la información a las funciones correspondientes.

Antes de explicar los posibles mensajes que intervienen en la función, se va a explicar la composición de los mensajes (su estructura). El mensaje consiste en un paquete de información en forma de "array" de variables, estas variables también pueden ser otros "arrays". En las dos primeras posiciones se encuentran los "headers" 1 y 2 respectivamente, que equivalen al valor hexadecimal "0xaa" (170 en decimal). Tras ellos se encuentra el tamaño del mensaje, que engloba el conjunto del id y la información enviada, y tras estos datos van los comandos que se mandan en el flujo de información.

Para agregar la información de los comandos que recibe "executeMessage" o que extrae de ellos esta misma función, se emplean funciones como "setTurbine", "setGoalAngle", "setVelocity" o "reboot", que se encargan de asignar los valores de las variables correspondientes en función del mensaje. Además, aquellos comandos que van acompañados de valores numéricos como ángulos o velocidades, requieren de una función intermedia que interpreta la información recibida, extrae y devuelve el valor de tipo float y esto lo que se le introduce como entrada a las funciones mencionadas.

6.2. Mensajes

Como ya se ha comentado, el programa sabe qué hacer gracias a los mensajes que manda la interfaz, dentro de ellos hay información como el identificador del módulo, del motor que se desea mover, o valores numéricos que indican la fuerza que se va a aplicar a un actuador, pero lo más importante es qué tipo de mensaje se recibe. Se va a proceder a explicar los tipos de mensajes que intervienen en la función.

| Mensaje | Valor hex | Valor dec | Descripción |
|--------------------------|-----------|-----------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| ROM_TORQUE_ON | 0x03 | 3 | Este mensaje consiste en una variable booleana llamada “torque” que, en función del identificador que recibe, le indica a una articulación si puede moverse o no, es decir, activa la potencia del motor. Cada objeto motor contiene su variable “torque” individual, haciendo que pueda ir indicándose uno a uno el que se desea activar. |
| ROM_TORQUE_OFF | 0x04 | 4 | Al igual que hay un mensaje que le indica a los motores cuándo pueden activarse, existe este otro que desactiva la potencia del mismo, también recibe un booleano que se le asigna a la variable “torque” del motor que indica el identificador que se ha enviado. |
| ROM_SUCTION_CUP | 0x06 | 6 | Este mensaje es el encargado de gestionar las ventosas que hay en ROMERIN, indicando la fuerza de succión que han de hacer cada una. Dentro de la información que contiene, están el tipo de mensaje y el valor de la fuerza que se ha indicado, que varía entre 0 y 100 siendo este último el mayor que puede tomar. |
| ROM_GOAL_ANG | 0x07 | 7 | Para poder mover los motores, es necesario indicar el ángulo al que se desea que lleguen, para esto está este mensaje. Es el encargado de darle el valor “goal_angle” que indica la referencia de dónde debe moverse cada motor que, mediante un lazo de control compuesto por un PID de posición con un PD de velocidad en cascada, calcula la fuerza que debe hacer el motor para llegar a ese ángulo. |
| ROM_VELOCITY_PROFILE | 0x22 | 34 | Este mensaje establece la velocidad objetivo que el lazo de control usa para calcular la fuerza que debe hacer. Este mensaje puede tomar valores entre 0 y 100, siendo estos porcentajes de la velocidad máxima que los motores pueden tomar. |
| ROM_REBOOT_MOTOR | 0x08 | 8 | En el caso en que se quiera reiniciar un motor a sus valores iniciales, se emplea este mensaje. |
| ROM_GET_FIXED_MOTOR_INFO | 0x09 | 9 | Se encarga de extraer información de los límites articulares de ROMERIN. |
| ROM_GET_NAME | 0x11 | 17 | Cada módulo contiene un nombre propio (THOR, ODIN, LOKI Y FRIGG), este mensaje es el encargado de obtener el nombre a la variable “MODULE_NAME” que contiene individualmente cada módulo. |
| ROM_GET_MOTOR_INFO | 0x15 | 21 | Este mensaje contiene la información de los motores en el estado actual que extrae de la simulación, esto lo usa la interfaz para mostrar por pantalla este estado y que el usuario sea capaz de verlo numéricamente. |
| ROM_GET_CONFIG | 0x16 | 22 | Solicita a la simulación la configuración actual. |
| ROM_CONFIG | 0x17 | 23 | Envía un test al ordenador para establecer una configuración de variables como “NAME”, “WIFI_SSID”, “IP”, “GATEWAY” o “MASK”. |
| ROM_CONFIG_V2 | 0x31 | 49 | Tiene la misma función que “ROM_CONFIG” pero extendida a la segunda versión del proyecto, incluyendo datos como dimensiones o límites. |
| ROM_GET_CONFIG_V2 | 0x30 | 48 | Solicita la información extendida que proporciona “ROM_CONFIG_V2”. |
| ROM_TEST_A_MESSAGE | 0x20 | 32 | Este mensaje se usa para “debuggear”, indicando que se debe imprimir por pantalla la frase “ATTACH ON - TEST A”. |
| ROM_GOAL_CURRENT | 0x24 | 36 | Esta orden no se emplea actualmente en la simulación de MuJoCo (aunque en un futuro puede usarse), pero sí en el control real de ROMERIN, indicando si hay que hacer un cambio en la intensidad de corriente que se le da a los actuadores. |
| ROM_CONTROL_MODE | 0x25 | 37 | Este mensaje tiene la función de indicar el modo de operación en el que se están trabajando los motores, lo indica la variable “control_mode”. Este modo puede ser el modo “posición” o el “posición basado en corriente”. Los dos mueven el actuador al ángulo que le indica el usuario, pero se diferencian en que el segundo limita también la corriente, consiguiendo una mayor precisión en el movimiento. |

Tabla 6.1: Listado de mensajes del proyecto ROMERIN

Para que se entienda mejor la finalidad del mensaje "ROM_VELOCITY_PROFILE", se incluye una imagen que representa el esquema del lazo de control real de los motores Dynamixel empleados en el robot y que se han simulado:

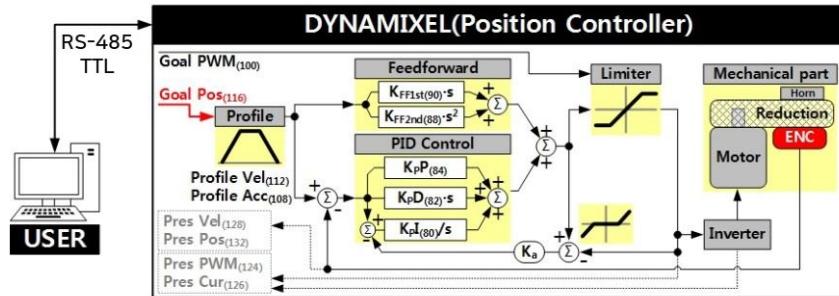


Figura 6.1: Lazo de control de los motores

Como se observa en el esquema, el lazo está compuesto por un limitador, un inversor, un generador de perfiles de velocidad, un bloque PID junto al control "feedforward" y el "encoder", que proporciona la posición actual del giro del motor cerrando así el lazo. Debido a estos componentes del lazo, la velocidad debe quedar limitada (se suele hacer por seguridad y para lograr mayor estabilidad al sistema evitando vibraciones), esto afecta a la fuerza total de torque que ejerce el motor y es lo que genera la necesidad de trabajar con un perfil de velocidades y por ello, usar este mensaje.

6.3. Virtualización de mensajes

En la clase ModuleSimulator es donde se hace la virtualización de los mensajes. Se simula la comunicación con el hardware de ROMERIN sin necesidad de interactuar con el robot real. Recibe y procesa los mensajes antes de enviar la información a los motores mediante la función "executeMessage()" explicada anteriormente, generando respuestas simuladas.

El proceso inicia con la función "sendMessage()", que en vez de enviar mensajes reales, llama a "sendVirtualMessage()", encargada de gestionar la virtualización del mensaje. Si hay conexión con la interfaz, se usa "romerinMsg_simulate()" para generar una versión virtual del mensaje y luego enviarla mediante la función "writeDatagram()" dentro del objeto "ip_port", que usa el "socket" UDP para intercambiar la información con la interfaz. Esto permite enviar mensajes dentro de la simulación sin necesidad de usar hardware real. "executeMessage()" gestiona la información, y ejecuta las acciones mencionadas anteriormente. En la interfaz, simplemente se imprime información en la consola con "BT_DEBUG_PRINT()".

Resumiendo, la virtualización de mensajes en la clase ModuleSimulator permite probar el sistema sin necesidad de hardware real, facilitando el desarrollo, la depuración y la validación del código en un entorno seguro y controlado.

6.3.1. Estructura de los mensajes

A la hora de comunicarse simulación e interfaz, dentro de "ModuleSimulator" (clase que pertenece al primero de ellos), se producen cuatro llamadas a "sendMessage()", mandando entre todas tres tipos de mensajes, un texto, el estado del robot, y toda la información actual de los motores de ese módulo. Para facilitar la comprensión de esta sección del código y debido a la importancia de la misma, se va a explicar la estructura de cada una de estas opciones:

- **Información de los motores:** Esta información se manda de forma regular ya que es la entrada de una llamada a "sendMessage()" que hay dentro de la función "sendRegularMessages()" (Se va a explicar más adelante, pero por el momento es suficiente saber que se le llama cada 10 milisegundos y manda mensajes en cada llamada a modo de control). Para mandar estos datos se hace un bucle de seis iteraciones (una por cada motor) y se invoca en cada una de ellas a "sendMessage()", a la que se le pone como entrada la función "motor_info_message", que tiene como objetivo crear un mensaje de tipo "RomerinMsg" encapsulando toda la información relevante de un motor en un formato compacto y eficiente. Este mensaje sigue una estructura binaria específica que asegura una transmisión rápida y clara.

El mensaje empieza con un encabezado ("HEADER") de dos bytes que valen lo mismo, 0xAA en hexadecimal. Este encabezado actúa como un identificador estándar para señalar el inicio del mensaje. Tras él, un byte indica el tamaño total del mensaje ("size"), seguido por un identificador ("ROM_MOTOR_INFO" = 0x01), que especifica que la información pertenece a un motor. Después, en el campo "info", se almacenan los distintos parámetros del motor, organizados de forma secuencial para facilitar la lectura y escritura de datos.

La sección de "info" incluye datos clave de cada motor: su ID ("m_id"), estado ("status"), modo de operación ("operatingMode"), posición ("position"), velocidad ("velocity"), intensidad de corriente ("intensity"), temperatura ("temperature") y voltaje ("voltage"). Cada uno de estos parámetros se guarda en la clase "MotorInfo". Para escribir estos valores en el mensaje, se utilizan funciones auxiliares como "romerin_writeUChar" y "romerin_writeFloat", que aseguran que los datos se almacenen correctamente en formato binario. Al final, la variable "size" se ajusta para reflejar el número total de bytes del mensaje, que en este caso es de 22, asegurando que la interfaz pueda interpretar y procesar la información fácilmente. La estructura final queda:

```
[HEADER (0xAA 0xAA)] [SIZE (22)] [ID (0x01)] [m_id] [status] [operatingMode] [position] [velocity] [intensity] [temperature] [voltage]
```

- **Estado del robot:** Esta información también se manda de forma regular ya que se hace invocando a "sendMessage()" dentro de "sendRegularMessages()", pero en este caso se hace una única iteración y se le introduce como entrada la función "state_message", que genera un mensaje de tipo "RomerinMsg" con

el identificador "ROM_STATE" (0x18 en hexadecimal). Este mensaje contiene información sobre la conexión y el estado de energía del sistema. Primero, empaqueta cuatro booleanos en un solo byte ("aux"), donde cada bit indica si el sistema está conectado a Bluetooth, WiFi o si detecta alimentación. Luego, se añade el "cicle_time", que representa el tiempo de ciclo del sistema en un único byte. Ambos valores se escriben en la sección "info" del mensaje, y el tamaño total del mensaje se calcula sumando la longitud de los datos a medida que se van añadiendo.

Como con la información de los motores, el mensaje sigue la estructura de la clase "RomerinMsg", que comienza con un encabezado de dos bytes que valen 0xAA, seguido de un byte que indica el tamaño total, el identificador del mensaje (0x18), y los datos en "info". El tamaño total resultante es de 6 bytes y su estructura queda:

[HEADER (0xAA 0xAA)] [SIZE (6)] [ID (0x18)] [aux] [cicle_time]

- **Texto:** En este caso, la información no se manda de forma cíclica ya que se encuentra fuera de "sendRegularMessages()". Para mandar este tipo de información, la entrada de "sendMessage()" es la función "text_message()", que también crea un mensaje del tipo "RomerinMsg". Comienza con su encabezado de dos bytes de valor 0xAA, seguido por la sección "size" que indica la longitud total del mensaje, incluyendo el identificador ("id") y la información ("info"). El identificador toma el valor de "ROM_TEXT" (0x05 en hexadecimal), identificando el mensaje como uno de texto, y la información contiene la cadena proporcionada por el usuario, que se mete en el buffer con "romerin_writeString()", la cual incluye un último carácter vacío, indicando el fin del mensaje. El tamaño total del mensaje es de 6 bytes y la estructura queda:

[HEADER (0xAA 0xAA)] [SIZE (N)] [ID (0x05)] [text] [0x00]

6.4. Comunicación por sockets

Como ya se ha mencionado, los mensajes pasan a través de "sendVirtualMessage()" para su virtualización y envío. Para esto último se emplean los sockets. Estos son métodos de comunicación y transmisión de datos entre procesos en la misma máquina usando la metodología de "cliente-servidor". Para transmitir estos datos se usan varios protocolos de comunicación, se puede usar el TCP (SOCK_STREAM) o el UDP (SOCK_DGRAM).

En este caso se ha optado por usar el segundo protocolo (el por qué de esta decisión se explica detalladamente en 7.4) y se va a explicar el proceso que realiza el código para lograr dicha comunicación:

En cuanto se invoca a "sendVirtualMessage()", la función verifica que el mensaje tenga contenido (m.size > 0), de no ser el caso, se sale de la función y no se

envía nada. Tras esto se comprueba también que el socket ("ip_port") y la dirección ("sender") sean válidos. Una vez se ha visto que estas condiciones se cumplen, el programa procede a encapsular del mensaje.

Para el envío, se envuelve el mensaje original dentro de otro con el identificador "ROM_SIMULATION" de valor 0xF0 en hexadecimal (esto se hace en cada mensaje), añadiendo el identificador del módulo ("vid"). Luego, el mensaje encapsulado se envía como un datagrama tipo UDP mediante "writeDatagram()" a la dirección almacenada en "sender" y al puerto 12001.

El mensaje final que se envía tiene la estructura [HEADER (0xAA 0xAA)] [SIZE] [ID (0xF0)] [VID] [m], donde "SIZE" es "m.size" + 3, "VID" es el identificador del módulo que envía el mensaje, y [m] es el contenido original. Así, "sendVirtualMessage()" permite la comunicación simulando un entorno donde los módulos pueden intercambiar información mediante sockets UDP.

Capítulo 7

Gemelo digital e interfaz

En esta sección se describen los resultados obtenidos, desde la estructura del proyecto final hasta explicaciones de funcionamiento y uso.

7.1. Introducción

El proyecto finalmente consta de dos programas, uno encargado de la simulación y el otro es la interfaz, encargada de indicar a la simulación qué debe hacer. La GUI (interfaz gráfica de usuario) permite seleccionar mandatos que le indican al simulador cómo debe comportarse, algunos de estos mandatos vienen acompañados de otros datos o valores que complementan la información en caso de que la orden mandada lo requiera. Posteriormente se muestra en detalle cómo funciona la aplicación para que el usuario pueda utilizarla sin necesidad de tener conocimientos de robótica o de programación.

7.2. Estructura del proyecto

Todo el código tanto de simulación e interfaz, ha sido desarrollado en C++, siendo apoyados ambos por librerías del marco de trabajo QT, pero en el caso del primero, también con un script en Python.

A continuación, se muestra un esquema sencillo de la estructura de estos dos programas para facilitar su entendimiento:

- Esquema de la estructura de la interfaz:

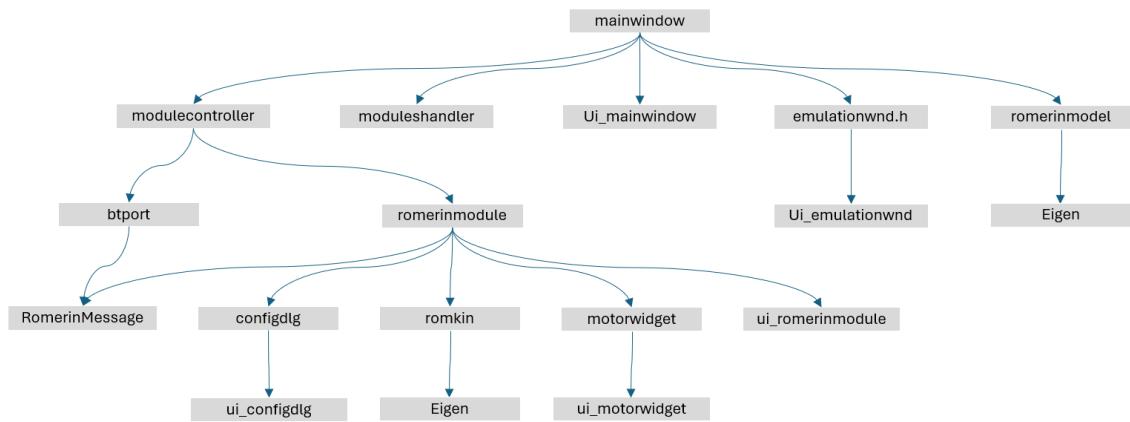


Figura 7.1: Esquema de ficheros de la Interfaz

- Esquema de estructura del repositorio (se ha separado en 5 imágenes para que se pueda entender lo que hay en él):

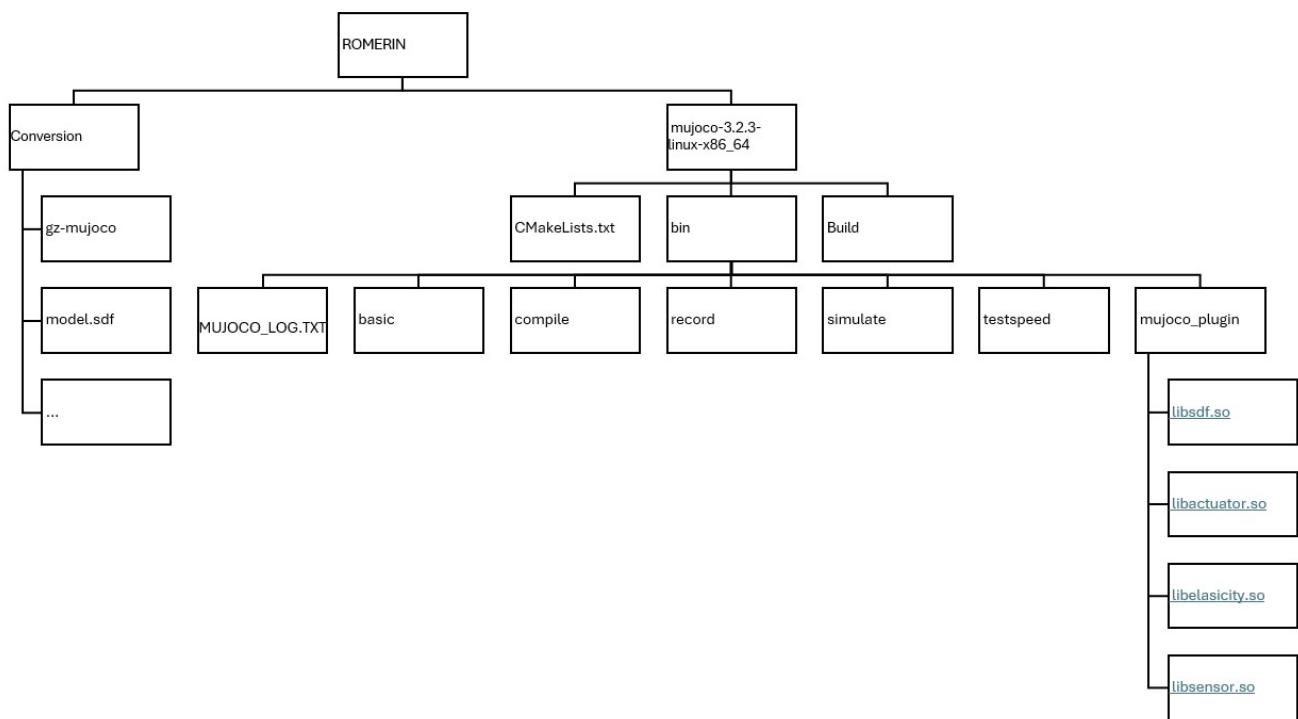


Figura 7.2: Parte 1 del esquema del repositorio

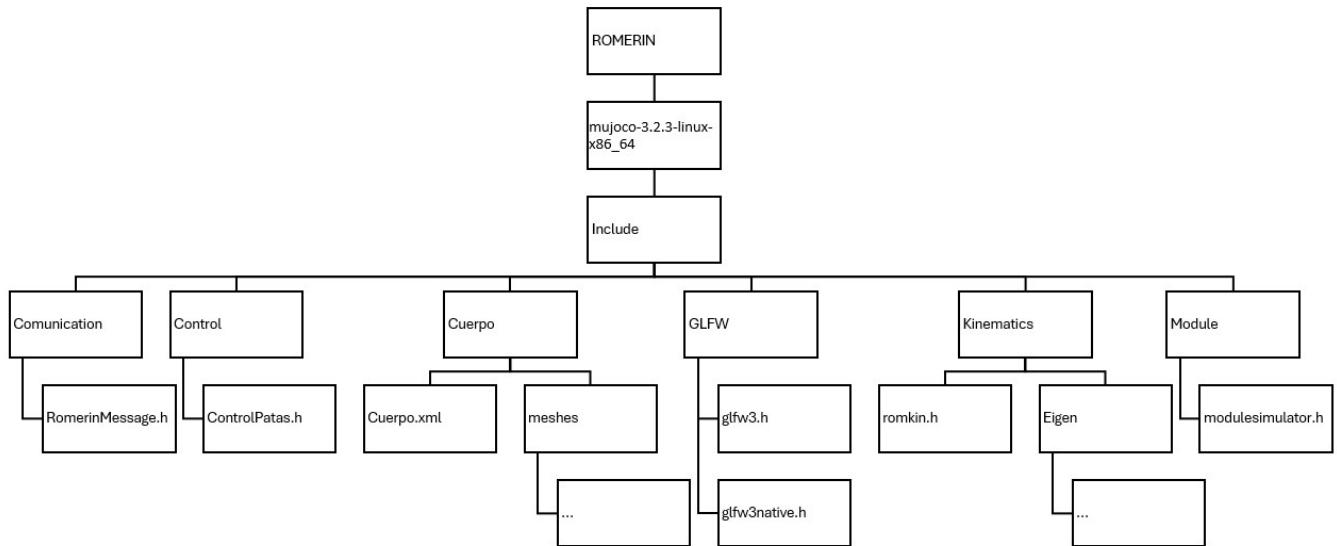


Figura 7.3: Parte 2 del esquema del repositorio

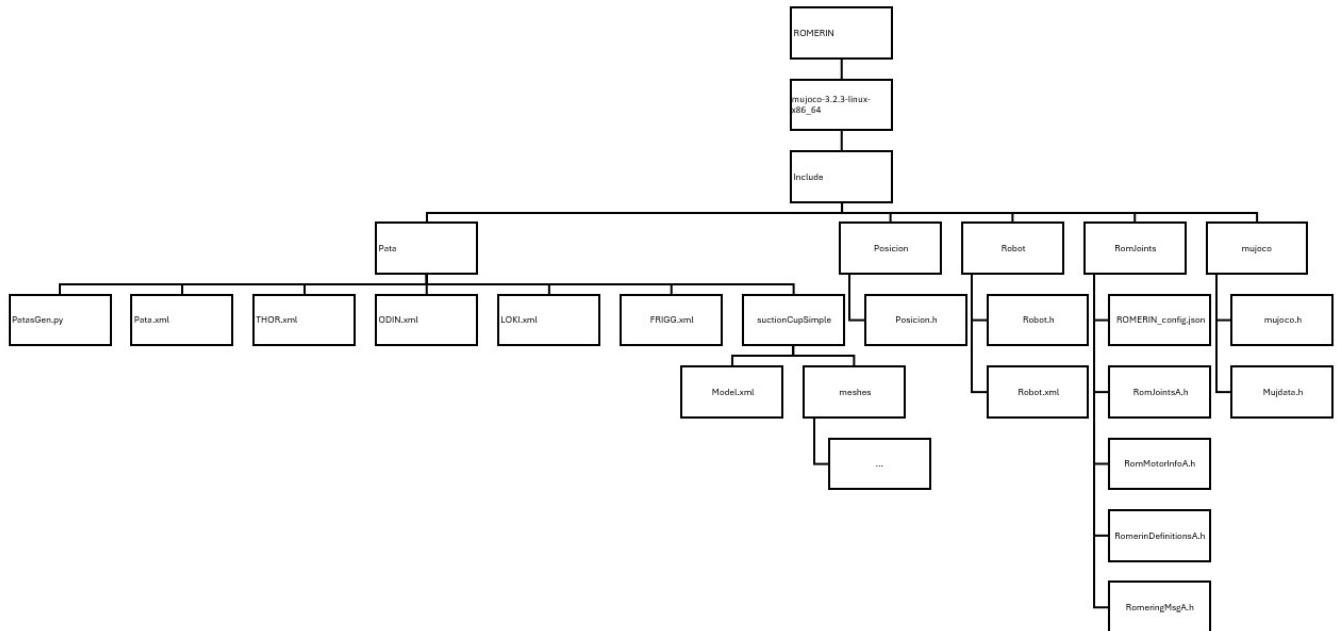


Figura 7.4: Parte 3 del esquema del repositorio

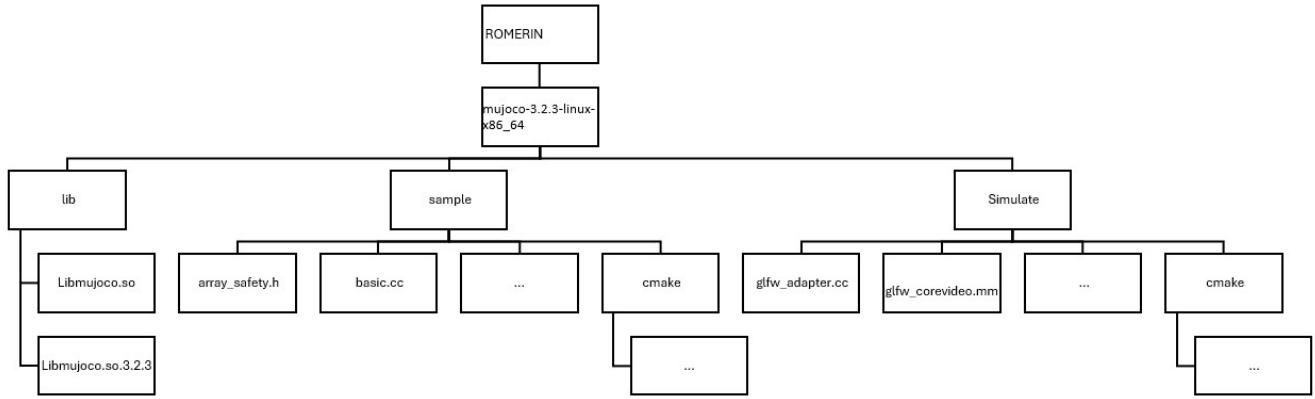


Figura 7.5: Parte 4 del esquema del repositorio

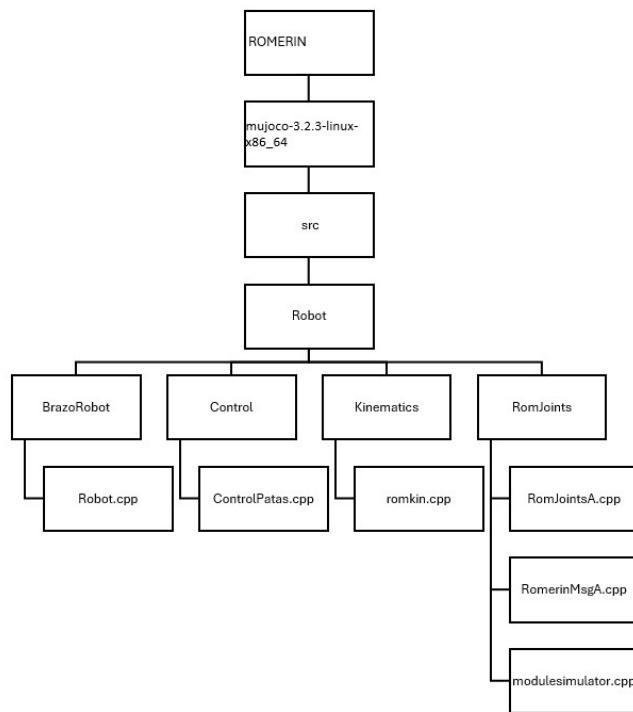


Figura 7.6: Parte 5 del esquema del repositorio

7.3. Interfaz gráfica

La interfaz gráfica de usuario (GUI) ha sido desarrollada por Miguel Hernando Gutierrez y se ha trabajado en paralelo al desarrollo de la misma para poder realizar el resto del proyecto.

Este programa ha sido desarrollado en el marco de trabajo de QT, que se usa principalmente para este tipo de tareas, esto ha proporcionado grandes ventajas no solo en el apartado visual, sino en la gestión de la información proporcionada por el usuario. Algunas de las ventajas que proporciona QT son:

- **Multiplataforma sin esfuerzo:** Qt permite desarrollar aplicaciones que funcionan de manera nativa en diferentes sistemas operativos como Windows, macOS, Linux, Android e iOS sin necesidad de reescribir el código para cada plataforma. Las aplicaciones se integran visual y funcionalmente con el sistema operativo, lo que garantiza una muy buena experiencia de usuario.
- **Ecosistema robusto y versátil:** Qt no solo es un marco para crear interfaces gráficas, sino que ofrece herramientas avanzadas como QT Creator o Qt Designer, módulos adicionales y sistemas de señales y ranuras. Todo esto permite a Qt crear aplicaciones complejas.
- **Interfaces gráficas modernas y personalizables:** Qt permite crear interfaces gráficas modernas y atractivas utilizando tecnologías como QT Widgets (que es ideal para aplicaciones tradicionales de escritorio), Qt Quick (que se usa para la creación de interfaces dinámicas) y la personalización total que permite cambiar estilos y temas.

Esta interfaz ha sido desarrollada de tal forma que permite al usuario interactuar de forma simple e intuitiva con la simulación sin necesidad de editar el código o de tener conocimientos de programación o robótica.

El proceso que realiza este programa es relativamente similar al de la simulación. Se comunica mediante sockets tipo UDP para mandar mensajes a la simulación y a su vez los recibe y clasifica en función de la información que contenga dicho mensaje.

Un detalle a tener en cuenta, es que este programa es capaz de continuar con el flujo de información con la simulación gracias a que los mensajes de actualización de los valores que se muestran en el menú se hacen cada varias décimas de milisegundos, en un inicio estaba hecho a la velocidad del procesador y esto causaba que se colapsara el socket y por ello el programa entero.

Estos mensajes no sirven únicamente como mandatos para el robot, sino que también tienen la función de indicarle a la interfaz cuándo debe mostrar algunos menús que inicialmente no aparecen, como ejemplo si no hay comunicación, no muestra el

menú individual de cada pata, sino que permanece en el menú inicial.

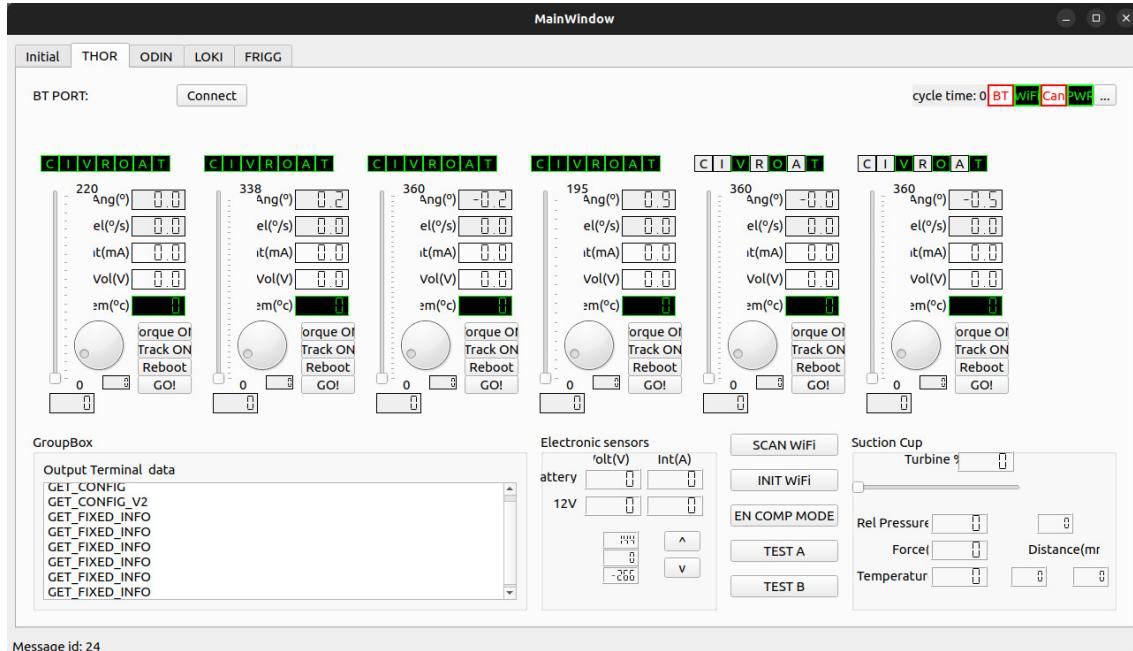


Figura 7.7: Interfaz de usuario (GUI) de ROMERIN

7.3.1. Explicación de la introducción de mandatos

En esta sección se va a describir con detalle el funcionamiento de la Interfaz y todas las opciones posibles de las que dispone.

Al ejecutar la interfaz, lo primero que aparece es el menú principal desconectado. En él aparecen varias opciones, pero de cara a la simulación, la necesaria para seguir con el funcionamiento es el botón "Enable Wifi", que habilita la conexión por un puerto al que se debe conectar el simulador.

Bajo el botón de "Enable Wifi", aparece una pequeña tabla que se completará con la información correspondiente una vez se haya conectado a interfaz con el simulador.

También cuenta con opciones como la de conectarse a otro simulador llamado Apolo, u otras como la de conectarse a ciertas IPs o a la propia del ordenador o la de usar unos joystick para controlar el robot. Estas dos últimas se usan para conectarse al robot real o controlarlo.

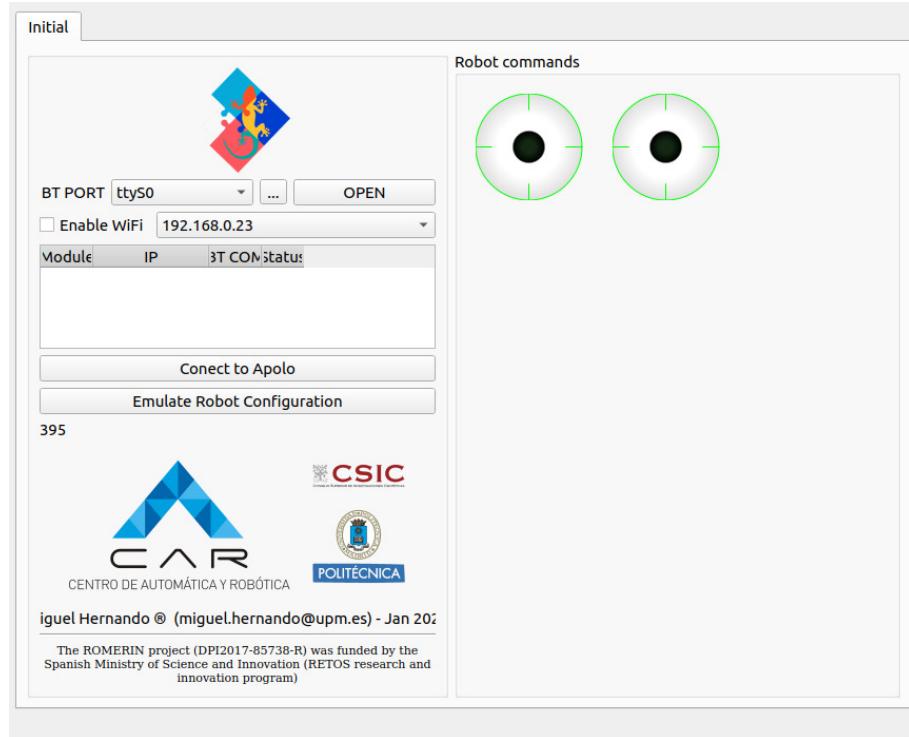


Figura 7.8: Menú principal desconectado

Al hacer click en el botón "Enable WiFi", el programa se conecta a un puerto por medio de un socket tipo UDP y, si encuentra conexión con el simulador, empieza un intercambio de mensajes cuya información ayuda a completar la tabla mencionada anteriormente con datos como el nombre de la pata a la que virtualmente se conecta, la ip de dicha pata o el estado.

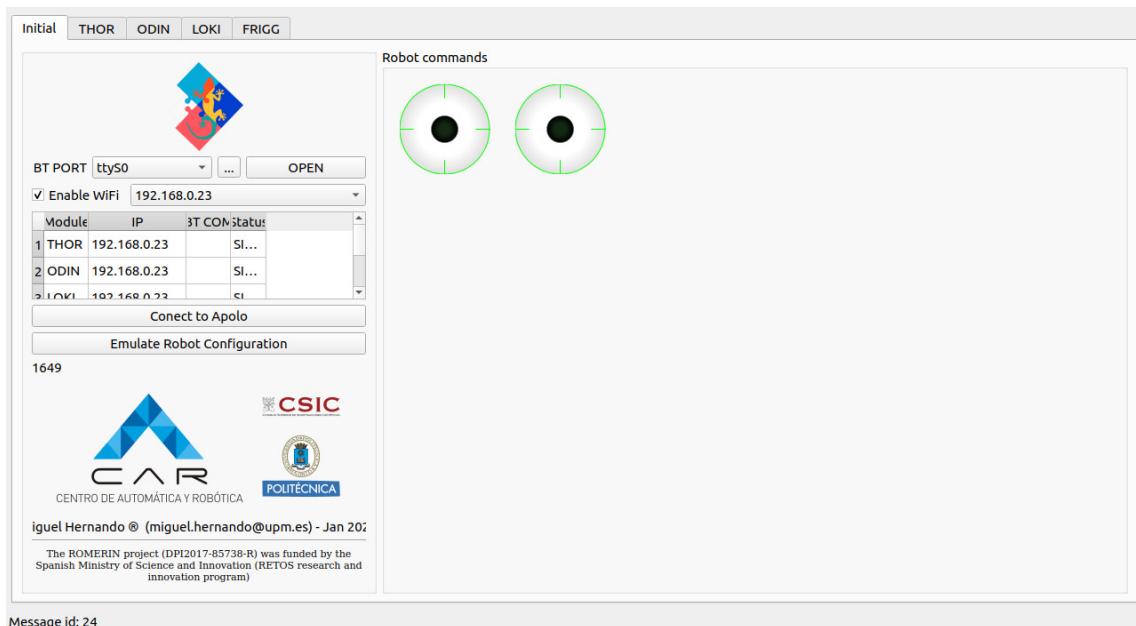


Figura 7.9: Menú principal conectado

Dentro de esta pantalla, se pueden apreciar las secciones correspondientes a cada uno de los 6 motores de la pata, que contienen todo lo necesario para moverlos, activarlos y desactivarlos.

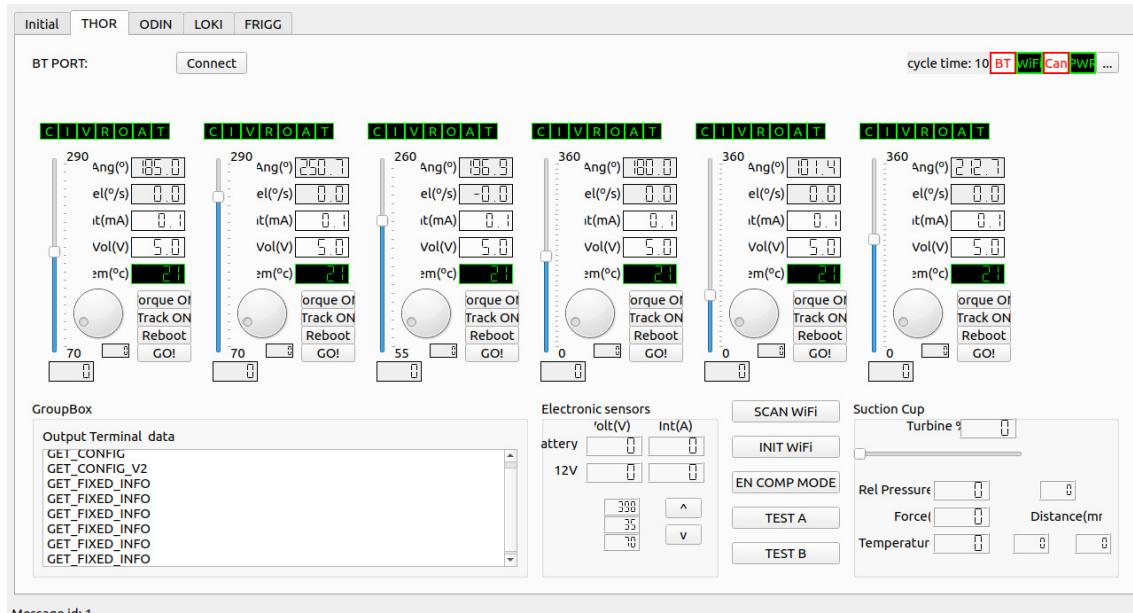


Figura 7.10: Menú de interacción con motores

Junto a este cambio en la pestaña principal, también salen pestañas específicas de cada pata. Estas pestañas son los menús de los actuadores, tanto los motores como las ventosas. En este menú es donde se pueden modificar valores para mover a ROMERIN en la simulación o mandarle cierta orden para que la realice.

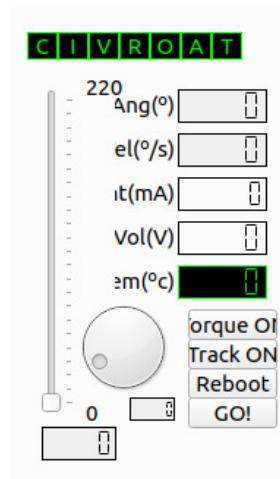


Figura 7.11: Sección de interacción con cada motor

Dentro de este menú aparecen diferentes acciones aplicadas a esta simulación:

- **Botón Torque On:** Este botón habilita el funcionamiento de los motores, sin él, el motor no funcionará a menos que este botón esté presionado sin importar

leo que se le mande.

- **Botón Track On:** Este botón también es esencial para el funcionamiento del motor ya que pone el marcador de la barra de ángulo con el valor que tiene en la simulación, de esta forma hay un seguimiento de esta información.
- **Botón Reboot:** reinicia los motores dándoles el valor inicial.
- **Barra de ángulo:** Esta barra se puede modificar arrastrando el ratón verticalmente para modificar su valor, con esto es con lo que ROMERIN mueve sus motores al ángulo que se le indica.
- **Rueda de velocidad:** esto modifica la velocidad con la que se debe mover el robot.
- **Barra de fuerza de turbina:** este es como el de velocidad pero aplicado a la fuerza de succión, de tal forma que si se marca 0, la ventosa no hará fuerza.
- **Botón de compact mode:** esta opción habilita el modo compacto de flujo de información, que consiste en enviar menos mensajes pero con más información.

Además de todos estos botones y barras, aparecen indicadores de algunos datos: ángulo del motor, velocidad, intensidad, voltaje, temperatura, fuerza de succión o presión relativa. Dentro de estos indicadores, ya que es una simulación, algunos valores han debido ser falseados ya que no hay forma de simularlos de otro modo, entre ellos están la intensidad, el voltaje y la temperatura.

Si no se modifica nada, el aspecto inicial de ROMERIN es:



Figura 7.12: Simulación de Romerín en estado estático

Pero si se pulsan botones y se modifican valores:

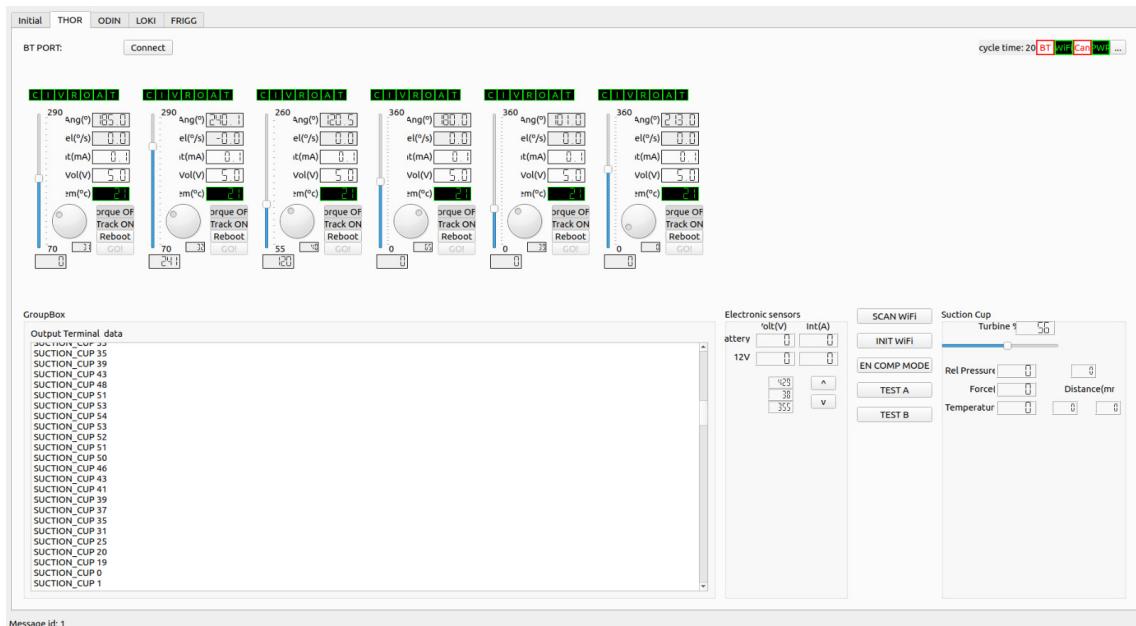


Figura 7.13: Menú de interacción con motores en uso

ROMERIN empieza a moverse, en este caso levantando la pata "THOR".

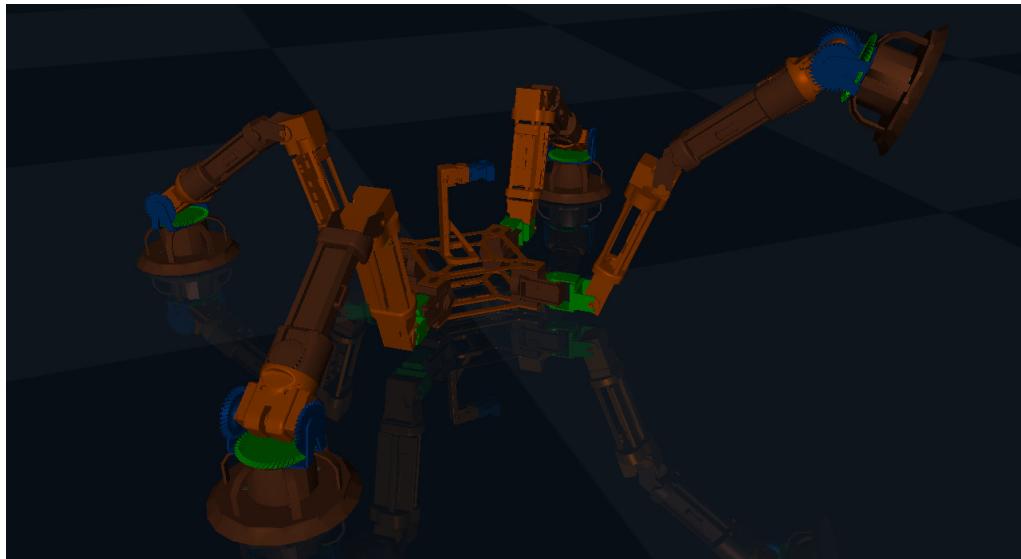


Figura 7.14: ROMERIN con una pata elevada

Como se muestra en la imagen del menú de los actuadores con botones pulsados y valores modificados, al aplicar una fuerza de succión, la superficie queda adherida, pero para que esto pase la superficie de la ventosa tiene que estar en paralelo con la superficie del suelo, como en el robot real.

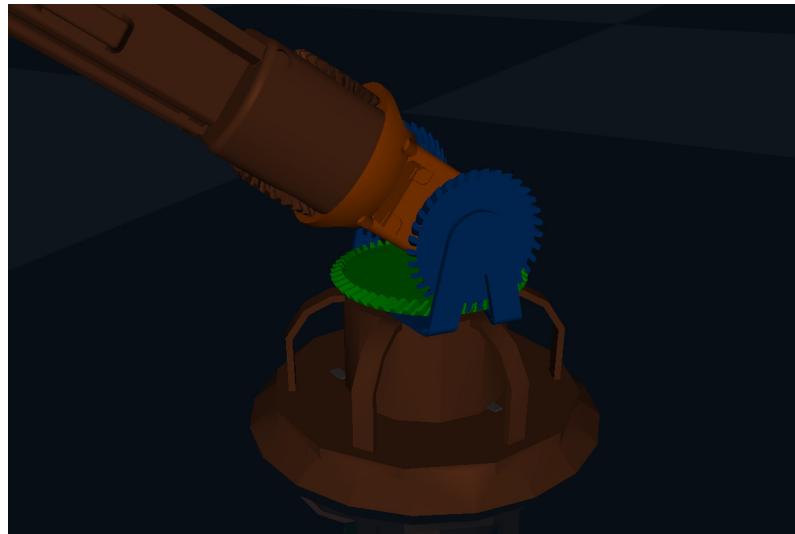


Figura 7.15: Ventosa en uso

7.4. Interacciones de la interfaz con el modelo

Para que el usuario pueda indicarle a la simulación cómo debe comportarse, se han establecido dos formas de interacción entre el modelo y el usuario.

- **Interacción del modelo con el teclado:** tomando uno de los ejemplos que proporciona MuJoCo junto a su descarga e instalación, éste proporciona un segmento de código de interacción del teclado con la ventana de simulación y, pese

a que en el resultado final no se ha usado, cabe destacarlo para posibles usos en proyectos futuros relacionados con ROMERIN.

- **Interacción del modelo con la interfaz gráfica:** Como se ha comentado anteriormente en este documento, interfaz y modelo de simulación se comunican por medio de mensajes, cada uno contiene varios datos:

- **ID:** Cada mensaje contiene un número de identificación que hace referencia a la pata (módulo) hacia la que debe ir dirigido el mensaje o de la que proviene. De esta forma, el programa se asegura de que dos o mas patas no reciban el mismo mensaje y de que lo reciba la indicada.
- **Orden:** Junto a la identificación, en todos los mensajes va un número hexadecimal. Este número representa la orden que pretende transmitir el usuario y puede significar desde encender o apagar los motores individualmente, hasta variar el ángulo o la fuerza de succión.

Para cada orden que le llega al programa, hay una función encargada de hacer los ajustes necesarios para que la simulación pueda llevarla a cabo.

- **Valor:** En algunos casos como en la acción de cambio de ángulo o de cambio de fuerza de succión, se añade también un número que es el valor que se pretende ajustar en la simulación.

En función de la acción, este número puede equivaler a una medida distinta. En el caso del cambio de ángulos, el valor está en grados (que luego el programa convierte en radianes), mientras que tratando la fuerza de succión, el valor está en Newtons.

Para la comunicación entre la interfaz y el resto del programa, se han valorado dos protocolos de simulación, **TCP** y **UDP**.

El protocolo **TCP** es muy seguro, es decir, no suele haber pérdidas de paquetes de información, pero debido a esta característica es lento el flujo de información entre origen y destino.

El protocolo **UDP** al contrario que el anterior, tiene un flujo de información muy rápido pero no es tan seguro, es decir, que es probable que se pierda algún paquete de información.

Teniendo en cuenta estas características, se ha decidido usar el protocolo **UDP** debido a su velocidad a la hora de comunicarse. Además, como el flujo de información es constante, no supone ningún problema que se pierda algún paquete de información ya que el siguiente que se mande solucionará este problema.

Capítulo 8

Resultados

En este capítulo se va a explicar principalmente a través de pseudo-código el funcionamiento del programa encargado de hacer la simulación de MuJoCo, incluyendo imágenes explicativas en los casos en que sea necesario.

8.1. Simulador de ROMERIN

Anteriormente en este documento se ha hecho mención a que el simulador se ha hecho en C++ utilizando en conjunto las librerías de MuJoCo y Qt, así como también usa lenguaje XML para una descripción gráfica y un script de Python.

Dentro de los numerosos ficheros que componen el programa, hay que empezar explicando "Robot.cpp", que es el que contiene el "main" y que se encarga de generar el entorno de simulación e interactuar con él (7.12).

Define el modelo de MuJoCo y almacena su información en dos estructuras almacenadas dos direcciones que contienen dos punteros llamados "m" y "d", de tipo "mjModel*" y "mjData*" respectivamente. Estos punteros circulan por todo el código ya que para trabajar con la simulación es necesario editar o acceder a su contenido, pero esto se explicará mejor más adelante.

Contiene funciones que permiten la interacción del usuario con la pantalla donde se simula al robot, estas funciones son "keyboard", "mouse_button", "mouse_move" y "scroll".

Dentro del "main" se carga el archivo "Robot.xml" (5.2.1) para simularlo y se genera tanto la ventana como la escena donde se va a poder trabajar con él. Para interactuar con el modelo se usa un puntero de la clase "Control" con el mismo nombre pero en minúsculas. A "control" se le pasan las direcciones de "m" y "d" mediante la función "setParametros" para trabajar con su contenido más adelante. Además, mediante la función "iniciarHilosMovimiento", genera dos "threads", uno de ejecución y otro de control. Esto se hace para que en plena ejecución no se pierda tiempo almacenando y enviando valores a diferentes sitios y para aumentar la precisión de los datos que se obtienen, de tal forma que se aprovecha la "condición de carrera" que tienen estos hilos para actualizar en ambos los valores de manera

instantánea. Si se cierra la ventana, se detienen estos hilos y se borran los datos temporales correspondientes para no perder recursos del ordenador.

En el constructor de "control" se generan los cuatro módulos (instancias de la clase "ModuleSimulator") referentes a cada una de las patas, a ellos se les da un nombre ("THOR", "LOKI", etc), un identificador y un entero que sirve para saber qué actuadores de los documentos XML le corresponden a cada una. Además, se abre un archivo Json que le aporta a las patas el resto de datos que necesitan, como posiciones iniciales o límites de giro mediante la función "openJson".

A parte de generar las patas, se encarga de gestionarlas obteniendo su información con "getInfoPatas" e iniciando la comunicación con la interfaz mediante las funciones 'executeMessage"(6.1) y "recibirInfoSocket". Estas funciones trabajan con comandos que vienen de las librerías de Qt, un ejemplo de ello es la forma en la que se ha programado el socket.

```
if (!control->ip_port->bind(PUERTO_ENVIO, QUdpSocket::ShareAddress)) {
    qDebug() << "Error al enlazar el socket:" << control->ip_port->errorString();
}
```

Figura 8.1: Uso de librerías de Qt en la programación del socket

Esta figura representa cómo se ha hecho el control de errores durante el "bind" del socket donde se puede ver que, como se ha explicado, usa funciones de las librerías de Qt como "QUdpSocket::ShareAddress" y "qDebug".

Mientras se obtiene la información con el hilo de control, en el de ejecución ("Hi-loLoop") se entra en un bucle que permanece activo hasta cerrar la ventana de simulación. Aquí, a su vez el código entra en los bucles propios de cada módulo.

```
void* Control::loop(void* args){
    Control* control = static_cast<Control*>(args);
    while(control->getVentanaAbierta()){
        for(auto &mod:control->modules)mod.loop();
    }
    pthread_exit(NULL);
}
```

Figura 8.2: Bucles de los cuatro módulos

Dentro del bucle de los módulos, el programa se divide en dos funciones:

- **hardwareSettings:** Esta función llama al "loop" de "joints" (instancia de la clase RomJoints que contiene todos los actuadores de la pata en la que se encuentre), que se encarga de realizar el lazo de control de cada motor, obtener la fuerza torque que debe hacer para llegar al ángulo requerido (el "goal_angle") y, mediante las direcciones que apuntan a las estructuras mjModel y mjData de MuJoCo, aplica la fuerza torque obtenida.

- **softwareSettings:** Al contrario que la anterior, esta función no se ejecuta en tiempo máquina, sino que se le llama cada 10 mili segundos. Esto es porque se encarga de enviar un mensaje a la interfaz cada vez que se pasa por ella y, como se envía mediante un socket, debe haber una pequeña pausa entre cada envío para no sobrecargar el canal de comunicación. El contenido de la información que manda es el estado actual de los actuadores de la pata que llama a esta función pero se va a explicar más a fondo esta función:

- **sendRegularMessages:** Esta función actualmente informa a la interfaz sobre el estado y las posiciones actuales de los motores, pero está pensada para en un futuro informar sobre más datos como el estado actual de las ventosas o si la comunicación se está haciendo en el modo compacto.

```
void ModuleSimulator::sendRegularMessages()
{
    //PORT::sendMessage(state_message(state));
    sendMessage(state_message(state));

    /*if(state.compact_mode){ //only one message with all the critical info compressed
        sendMessage(romerinMsg_robot_compact_data(robot_compact_data));
    }*/
    //else{ //verbose info (default mode)

        for(int i=0;i<6;i++){
            sendMessage(motor_info_message(i,joints.motors[i]));
        }

        //sendMessage(suction_cup_info_message(suction_cup.getRelPressure(), suction_cup.getForce(),
        //suction_cup.getTemperature(),suction_cup.getDistances()));
        /* #ifdef POWER_INFO
            sendMessage(analog_info_message(power.getBatteryVolt(),power.getBatteryAmp(),
            power.get12vAmp()));
        #endif
    }
}
}*/
```

Figura 8.3: Función ”sendRegularMessages”

Como se puede ver en la figura, se llama dos veces a ”sendMessage” para mandar los dos tipos de datos mencionados, pero se han dejado comentados los futuros usos de esta sección de código, ya que aparecen llamadas a ”sendMessage” con la finalidad de mandar datos como la presión de la ventosa o la fuerza que ejerce.

”sendRegularMessages” es de gran importancia ya que sin ella, la interfaz estaría ciega y no tendría información del estado del robot, de manera que el usuario no podría interactuar correctamente con el mismo.

8.2. Comparación de los resultados con el robot real

Una vez finalizado el código de la simulación, se ha hecho una prueba comparativa tanto con lo obtenido en MuJoCo, como con el robot real. En ambos casos se ha estudiado el movimiento del tercer motor de la pata con el nombre "Loki", ambos con las mismas características, un 12 por ciento de la velocidad máxima y un movimiento partiendo de los 200 grados hasta los 120 grados. En ambos casos todos los motores del robot comienzan con las mismas condiciones. A continuación se muestra un gráfico comparativo con ambos resultados, en el que el color azul representa los valores reales y el rojo los simulados:

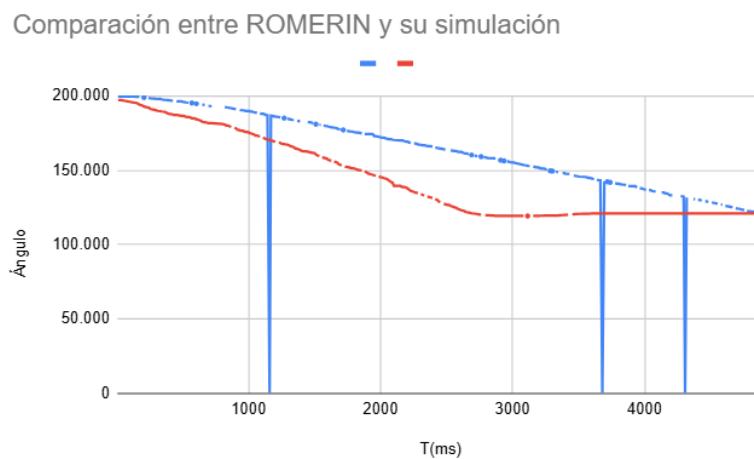


Figura 8.4: Gráfica comparativa de resultados

Como se observa en la figura, ambos motores parten del mismo punto inicial y obtienen con precisión el objetivo final. Viendo las formas de la gráfica que genera cada prueba, se observa que la simulación alcanza antes la posición objetivo, lo que indica que se mueve con una mayor velocidad y, por ello, el sistema aplica un mayor torque del que debería para hacer el movimiento.

Esta diferencia entre ambos resultados casi con total certeza es causada por el lazo de control de la simulación, seguramente realizando varias iteraciones, se consiga ajustar las constantes que intervienen en él para que el resultado sea lo más próximo a la realidad posible.

Por último, cabe resaltar que en el periodo estudiado en el robot real se producen tres picos en la gráfica, esto seguramente sea causado por pequeños fallos en el envío de paquetes del socket UDP que, como se ha explicado anteriormente, no es un protocolo tan seguro como otros pero hay una notable diferencia con otro en relación a la velocidad con la que se pueden enviar estos paquetes.

Capítulo 9

Conclusiones y futuros desarrollos

En este último capítulo, se lleva a cabo una reflexión sobre los resultados expuestos en el capítulo anterior. Para ello se analiza si los resultados satisfacen los objetivos establecidos inicialmente. Además, se incluyen algunas comparaciones relativas a herramientas usadas en este proyecto y se dedica una sección a señalar posibles mejoras y qué posibles líneas de investigación futuras hay.

9.1. Introducción

Para poder cumplir los objetivos establecidos, se ha desarrollado una aplicación en C++ y que recibe ayuda en ciertos momentos de un script de Python. Los procesos para diseñar esta aplicación se han tomado con la finalidad de cumplir ciertos criterios:

- Proporcionar una interfaz simple.
- Hacer una interfaz sencilla y fácil de usar de tal forma que un usuario sin conocimientos de robótica o informática sea capaz de usarlo sin problemas.
- Que la aplicación sea eficiente y no requiera de demasiados recursos, esto hace no sea necesaria una máquina muy potente para poder trabajar con este proyecto.
- Tener un control fuerte de errores que mejore la experiencia del usuario y haciendo que no deba preocuparse por los aspectos técnicos.

En rasgos generales, se considera que el proyecto cumple satisfactoriamente los objetivos iniciales. Sin embargo, este proyecto está sujeto a posibles mejoras

9.2. Desarrollos futuros

En esta sección se exploran posibles líneas futuras de trabajo y desarrollo cuya finalidad es mejorar el proyecto y avanzar en el mismo. Se proponen métodos para optimizar el programa. Estas propuestas incluyen la resolución de errores encontrados durante el desarrollo o implementación del proyecto. La sección se centra en cómo abordar ciertos problemas y en qué proyectos pueden partir de éste.

9.2.1. Resolución de errores y propuestas de mejora

Se ha conseguido cumplir con los objetivos iniciales, como así demuestra la simulación obtenida y su rendimiento, pero bien es cierto que hay más margen de mejora en la exactitud de la simulación. Además, se ha identificado algún error.

En relación a la exactitud de la simulación, se ha obtenido un muy buen resultado, teniendo gran similitud con los movimientos del robot real. Sin embargo, hay ciertos aspectos que pueden mejorar para aproximar más el resultado a la realidad. Entre estos aspectos se encuentra el ajuste del lazo de control, tanto su PID de posición como su PD de velocidad (ambos en cascada), que contienen las constantes referentes al proporcional, integrador y derivador. Con las constantes K_p y K_d, es posible ajustar más sus valores para mejorar la simulación, pero se podría añadir la K_i, ya que esta parte se ha omitido por no ser esencial. Por ello se recomienda trabajar en la forma de añadir el integrador al código del lazo de control y ajustar sus constantes a valores más precisos.

En cuanto al flujo de información entre los dos programas, hay un aspecto que se puede mejorar, y es en lo compactos que son los mensajes. Esto consiste en que, en vez de mandar un mensaje por cada instrucción, se envíe un único mensaje para varias instrucciones o mandatos, de esta forma el canal de comunicación se aprovecha más y hay mayor eficiencia en el proceso. Se ha comenzado a realizar esta mejora pero debido a la falta de tiempo, el "*compact mode*" (así se ha llamado a esa forma de flujo de información) no ha sido posible acabarlo, por lo que se recomienda acabarlo ya que supone una mejora en el rendimiento de la aplicación.

9.2.2. Ajuste para poder trabajar en Windows

Como se ha comentado anteriormente, este proyecto se ha desarrollado en Ubuntu 22, pensado para trabajar en él usando el sistema operativo Linux, pero haciendo varios cambios en el CMakeLists.txt y añadiendo las librerías de MuJoCo para Windows, se puede conseguir que funcione en ambos sistemas operativos.

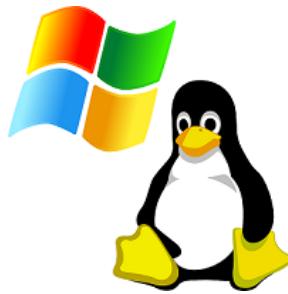


Figura 9.1: Logo de Windows y Linux

9.2.3. Dotar a la simulación de ROMERIN para andar

La principal línea de trabajo futura consiste en lograr que la simulación de ROMERIN sea capaz de caminar. Esto abriría la posibilidad de realizar diversas pruebas, desde analizar teóricamente el comportamiento del robot en distintos entornos, hasta evaluar si futuros cambios, como modificar componentes, añadir peso o reducirlo, podrían generar problemas en su funcionamiento.

Esta tarea presenta un alto nivel de dificultad, tanto en términos de cálculo como de lógica. Existe una diferencia significativa entre lograr movimiento en un robot que se desplace mediante ruedas y este caso particular: un robot compuesto por cuatro módulos independientes que, además, deben coordinarse para escalar superficies verticales y desplazarse de forma invertida sobre una superficie horizontal.

9.2.4. Colaboración de simulación con robot real

Una de las líneas más interesantes y ambiciosas para futuros desarrollos consiste en lograr una plena colaboración entre la simulación de ROMERIN y su versión física. Este enfoque permitirá optimizar tanto el diseño como el funcionamiento del robot al combinar los beneficios de un entorno controlado de simulación con la validación en el mundo real.

El objetivo principal y que es la principal línea de trabajo, es la colaboración e implementación de un sistema que permita sincronizar las interacciones entre la simulación y el robot físico. Esto incluiría la transferencia de datos en ambas direcciones: por un lado, enviar comandos desde la simulación para controlar directamente al robot físico y, por otro, recopilar datos del comportamiento del robot en el entorno físico y utilizarlos para ajustar y mejorar la simulación o al contrario. Este flujo bidireccional posibilitará un ciclo continuo de aprendizaje y ajuste, mejorando progresivamente la precisión del modelo simulado.

Además, esta colaboración conjunta abre la puerta a la validación de hipótesis futuras y el desarrollo de pruebas más seguras y económicas. Por ejemplo, se podrían evaluar movimientos complejos o realizar simulaciones en entornos complejos para el robot antes de implementarlos físicamente, reduciendo riesgos materiales como daños al robot y optimizando el tiempo de pruebas.

Entre los desafíos de este trabajo se encuentra garantizar la consistencia en la comunicación y sincronización entre ambos sistemas. Será necesario diseñar un protocolo de intercambio de datos muy eficiente y robusto, que sea efectivo y asegure que los datos clave (como posiciones, ángulos, velocidades o fuerzas) sean interpretados de forma coherente tanto en la simulación como en el robot real.

Finalmente, para habilitar esta colaboración efectiva, será esencial avanzar en la compatibilidad de hardware y software entre ambos sistemas. En esto se incluye el desarrollo de métodos que permitan la comunicación directa con los actuadores y sensores del robot físico desde la simulación. Con estas mejoras, el sistema ROME-RIN se convertiría en un programa capaz de combinar lo mejor de la simulación y la robótica física en un único ecosistema integrado.

Bibliografía

- [1] Pata de romerin. https://www.researchgate.net/figure/Physical-model-of-ROMERIN-robot-with-vacuum-generation-system_fig5_349055671, Consultado en Diciembre de 2024.
- [2] Coppeliasim - virtual robot experimentation platform. <https://www.coppeliarobotics.com>, Consultado en Enero de 2025.
- [3] Documentación oficial de isaac sim. <https://docs.nvidia.com/isaac/isaac-sim/>, Consultado en Enero de 2025.
- [4] Gazebo - robot simulation made easy. <http://gazebosim.org>, Consultado en Enero de 2025.
- [5] Mujoco - multi-joint dynamics with contact. <https://mujoco.org>, Consultado en Enero de 2025.
- [6] Nvidia isaac sim. <https://developer.nvidia.com/isaac-sim>, Consultado en Enero de 2025.
- [7] Nvidia omniverse developer resources. <https://developer.nvidia.com/nvidia-omniverse>, Consultado en Enero de 2025.
- [8] Webots. <https://cyberbotics.com>, Consultado en Enero de 2025.
- [9] Acerca de qt. https://wiki.qt.io/About_Qt, sortkey = 6, Consultado en Noviembre de 2024.
- [10] Coppeliasim. <https://www.coppeliarobotics.com/>, Consultado en Noviembre de 2024.
- [11] Gazebo, qué es. <https://gazebosim.org/home>, Consultado en Noviembre de 2024.
- [12] Introduction to c++. https://www2.eii.uva.es/fund_inf/cpp/temas/1_introduccion/introduccion.html, Consultado en Noviembre de 2024.
- [13] Libro de webots. <https://cyberbotics.com/doc/guide/introduction-to-webots>, Consultado en Noviembre de 2024.
- [14] Manual de latex. https://manualdelatex.com/?utm_source=chatgpt.com, Consultado en Noviembre de 2024.
- [15] Por qué ros. <https://www.ros.org/blog/why-ros/>, Consultado en Noviembre de 2024.

- [16] Qué es ros. <https://www.renesas.com/en/key-technologies/motor-control-robotics/robot-operating-system?>, Consultado en Noviembre de 2024.
- [17] Simuladores de robots. <https://www.sw.siemens.com/es-ES/technology/robotics-simulation/>, Consultado en Noviembre de 2024.
- [18] Python: qué es, para qué sirve. <https://www.cursosaula21.com/que-es-python/>, Consultado en Octubre de 2024.
- [19] What is xml. <https://aws.amazon.com/es/what-is/xml/>, Consultado en Octubre de 2024.
- [20] Cmake projects in visual studio. <https://learn.microsoft.com/es-es/cpp/build/cmake-projects-in-visual-studio?view=msvc-170>, Consultado en Septiembre de 2024.
- [21] Mujoco. <https://mujoco.org/>, Consultado en Septiembre de 2024.
- [22] Qué es gitlab y para qué sirve. <https://formadoresit.es/que-es-gitlab-y-para-que-sirve/>, Consultado en Septiembre de 2024.
- [23] Romerin. <https://blogs.upm.es/romerin/>, Consultado en Septiembre de 2024.
- [24] DataScientest. Visual studio code: A versatile code editor for beginners and advanced programmers alike. *DataScientest Blog*, 2025. Consultado en Octubre de 2024.
- [25] Universidad Complutense de Madrid. Introducción a latex. *Universidad Complutense de Madrid - PIMCD 2014*, 2025. Consultado en Diciembre de 2024.
- [26] Ubunlog. Texstudio: Crear documentos latex. *Ubunlog*, 2025. Consultado en Noviembre de 2024.

Apéndice A

Anexo

En esta sección, se explica detalladamente cómo instalar tanto la aplicación de la interfaz como la simulación de MujoCo.

A.1. Guía de instalación ROMERIN

En esta sección, se explica detalladamente cómo instalar tanto la interfaz como la simulación.

El simulador de MujoCo, como se ha comentado anteriormente, se ha diseñado para trabajar en Linux, ya que se ha desarrollado en Ubuntu 22 y utiliza ciertas librerías de este sistema operativo. Por esta razón, es necesario tener Linux para usar este proyecto.

Para facilitar la instalación del programa, se va a explicar el proceso mediante los comandos de terminal de Ubuntu necesarios para descargar los archivos, aplicaciones y librerías necesarias.

A.1.1. Instalación de interfaz

Todos los ficheros pertenecientes a la Interfaz, están contenidos en "*ROMERIN/TFG/firmware-emulation*", pero antes de trabajar con estos archivo, es necesario instalar ciertas librerías, así como el marco de trabajo de Qt:

■ Instalación de Qt:

- Asegurarse de tener actualizado el ordenador: **sudo apt get-update**
- Instalar Qt 5 y Qt Creator: **sudo apt-get install qtbase5-dev qtcreator**

Con esto se instala el entorno de desarrollo de Qt 5, sus bibliotecas principales y el IDE de Qt Creator.

Para asegurarse de que el archivo.desktop de Qt Creator en ”`/usr/share/applications`”, se puede usar `ls /usr/share/applications | grep qt-creator`. Si existe, debería salir el nombre del archivo, que puede ser algo como ”`org.qt-project.qtcreator.desktop`”.

- Para tener la aplicación en el escritorio, se usa `cp /usr/share/applications/org.qt-project.qtcreator.desktop /Escritorio/`. Esto en caso de que el nombre del archivo sea ”`org.qt-project.qtcreator.desktop`”.
- Por último, si el escritorio lo requiere, se cambian los permisos del fichero mediante `chmod +x /Escritorio/org.qt-project.qtcreator.desktop`.
- Descargar 2 librerías de Qt: `sudo apt install libqt5serialport5 libqt5serialport5-dev`.
- Descargar Clang++ ()en caso de no tenerlo) con: `sudo apt install clang`.
- Debido a que en un futuro se va a usar un gamepad para controlar a ROMERIN, hay que insatalar:
 - `sudo apt install libqt5gamepad5`
 - `sudo apt install libqt5gamepad5-dev`

De esta forma, el marco de trabajo de Qt queda instalado con las librerías necesarias para usar el proyecto.

Resumen de comandos:

```
sudo apt-get update
sudo apt-get install qtbase5-dev qtcreator
ls /usr/share/applications | grep qtcreator
cp /usr/share/applications/org.qt-project.qtcreator.desktop
chmod +x /Escritorio/org.qt-project.qtcreator.desktop
sudo apt install libqt5serialport5 libqt5serialport5-dev
sudo apt install libqt5gamepad5
sudo apt install libqt5gamepad5-dev
```

Una vez Qt Creator está instalado, hay que configurar el proyecto. Para ello, al abrir la aplicación, hay que ejecutar File/Open_File_or_Project y dirigirse a /ROMERIN/TFG/firmware-emulation/romerin-interface y seleccionar el archivo ”`romerin_interface.pro`” y clicar en ”open”. Tras esto, se abre otro menú en el que clicar ”Configure project”.

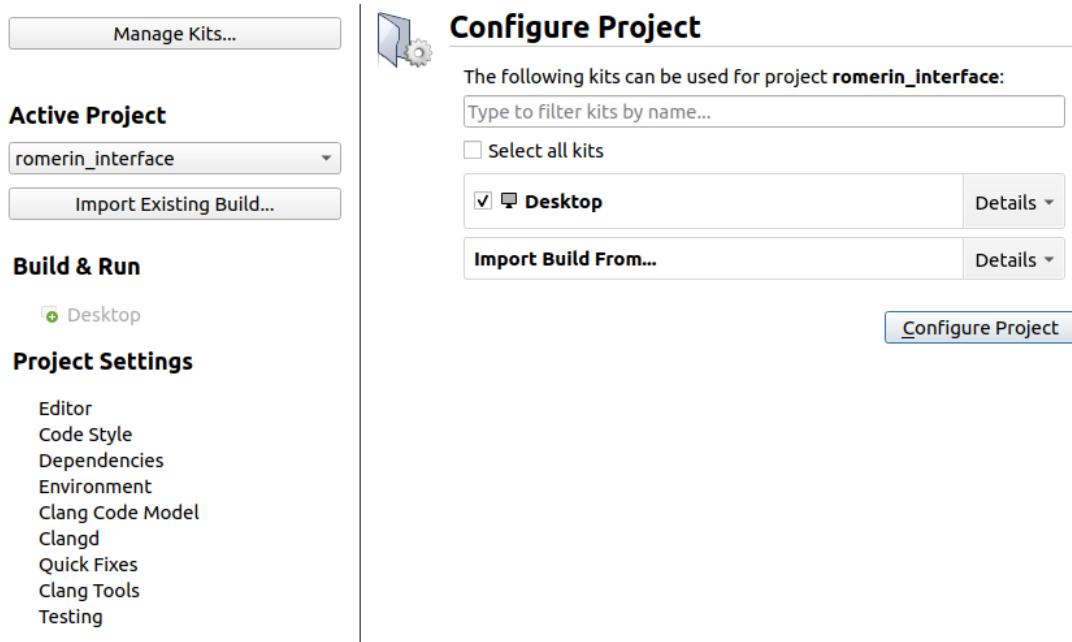


Figura A.1: Configuración de proyecto en Qt Creator

Con esto ya está totalmente instalado y listo para usar.

A.1.2. Instalación de simulación

Para poder trabajar con este proyecto, es necesario disponer de la herramienta CMake, que es el compilador que se ha usado no solo para esta función, sino que también sirve para crear el ejecutable.

para comprobar si el ordenador ya dispone de CMake, al usar el comando **cmake –version**, la propia terminal indica si hay que descargarlo o no. En caso de ser necesaria la instalación de esta herramienta, se deben seguir dos pasos:

- Asegurarse de que el sistema está actualizado por medio de: **sudo apt update**.
- Instalar la herramienta por medio de: **sudo apt install cmake**.

De esta forma, si se vuelve a usar **cmake –version**, debebe salir algo similar a "cmake version 3.22.1".

Por último y para que la simulación funcione, se deben añadir dos librerías, para ello se usa el comando: **sudo apt install libglfw3 libglfw3-dev**.

Se debe tener en cuenta que, si se desea actualizar la aplicación en proyectos futuros, pueden darse dos casos:

- **Añadir o eliminar ficheros:** En este caso, hay que editar el documento "CMakeLists.txt" situado en "ROMERIN/TFG/mujoco-3.2.3-linux-x86_64". Tras esto, se debe abrir una terminal e ir a la dirección "ROMERIN/TFG/mujoco-3.2.3-linux-x86_64/build" y, estando en este directorio, usar los comandos **cmake ..** y **make**. Así queda preparado el nuevo ejecutable del programa.
- **Editar los ficheros existentes:** Si no se ha añadido ningún fichero y únicamente se ha editado el código existente, la terminal debe abrirse y dirigirse a la carpeta build mencionada en el párrafo anterior y aplicar el comando **make**, que actualiza el ejecutable aplicando los cambios en el código que se han realizado.

De forma adicional, si se desea usar el mismo editor de código que el usado durante el desarrollo de este proyecto, para instalar VS Code (Visual Studio Code) se debe escribir en la terminal el comando **sudo snap install code --classic**.

Resumen de comandos:

```
cmake --version
sudo apt update
sudo apt install cmake
sudo apt install libglfw3 libglfw3-dev
cmake ..
make
sudo snap install code --classic
```