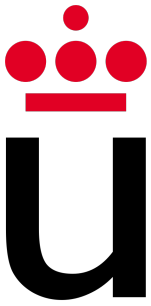


Universidad
Rey Juan Carlos

TRABAJO FIN DE GRADO

Modos de caminar y movimientos coordinados del humanoide NAO en el simulador Gazebo

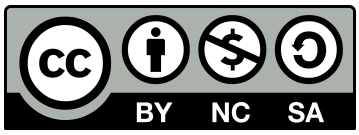
Grado en Ingeniería Robótica de Software

Escuela de Ingeniería de Fuenlabrada

Realizado por
Eva Fernández de la Cruz

Dirigido por
José María Cañas

Curso académico 2024/2025



Agradecimientos

Quisiera comenzar agradeciendo a mi tutor, José María Cañas Plaza, por su orientación y colaboración durante la realización de este Trabajo de Fin de Grado. Sin su guía este proyecto no habría posible, y valoro mucho el tiempo que me ha dedicado. Muchísimas gracias por todo, José María.

También quiero expresar mi más profundo agradecimiento a mi familia. Su apoyo constante, comprensión y sobre todo paciencia han sido clave no solo en la elaboración de este trabajo, sino a lo largo de toda mi trayectoria universitaria. Sin su confianza y cariño, este camino habría sido mucho más difícil. Millones de gracias por estar siempre ahí. Os quiero muchísimo.

Y también quiero dar gracias al lector, gracias por dedicarle tiempo a este trabajo, que ha sido fruto de mi esfuerzo, cariño y pasión durante todos estos meses, espero que sea de tu agrado y aprendas algo, igual que yo al hacerlo.

Resumen

El mundo de la robótica está en auge, y es por eso que cada vez son más visibles robots de todo tipo: Aspiradoras, friegasuelos, AMR's, cobots, robots que ayudan en cirugías, etc.

Sin embargo, y aunque también están en auge, los robots humanoides no están tan evolucionados como sus compañeros con ruedas. Esto es por la dificultad que tiene construir y programar estos robots, ya que, no sólo deben ser estables tanto dinámica como estáticamente, si no que hay que coordinar todas sus articulaciones para hacerles capaces de cumplir diferentes tareas, ya sea andar, subir escaleras, abrir una puerta, servir un vaso de agua, etc.

El objetivo de este TFG es mostrar la complicación que esto conlleva y ser capaces de construir una librería fácil para que el usuario sea capaz de programar al robot humanoide NAO de manera sencilla y convertirlo en un robot de servicio para diferentes aplicaciones, utilizando las funciones y clases de dicha librería. Esto se hará también para demostrar la potencia de la librería.

Este proyecto se ha desarrollado de manera completamente simulada, utilizando entornos virtuales para el diseño, implementación y validación del sistema, sin necesidad de hardware físico durante el proceso. Esto además nos da un abanico aún más amplio de opciones a la hora de diseñar el escenario para la aplicación de servicios que ofrecerá NAO, en este caso un invernadero.

Palabras clave: robótica, coordinación, articulaciones, robot de servicios, ROS 2, librería, abstracción, usuario, TFG, NAO.

Acrónimos

TFG - Trabajo de Fin de Grado
ROS 2 - Robot Operating System 2
URDF - Unified Robot Description Format
SDF - Simulation Description Format
IMU - Inertial Measurement Unit
JSON - JavaScript Object Notation
CSV - Comma Separated Values
API - Application Programming Interface
AMR - Autonomous Mobile Robot
AGV - Automatic Guided Vehicle
COBOT - Robot Colaborativo
KME - Kouretes Motion Editor
SLAM - Simultaneous Localization and Mapping
HTML - HyperText Markup Language
CSS - Cascading Style Sheets
URJC - Universidad Rey Juan Carlos

Índice general

1 Introducción 1
1.1. Robótica de Servicio 1
1.2. Robots Humanoides 4
1.3. Simuladores Robóticos 9

2 Objetivos 11
2.1. Problema a resolver 11
2.2. Metodología 12
2.2.1. Plan de trabajo 13

3 Herramientas Software utilizadas 15
3.1. Ecosistema: Middleware ROS 2 15
3.2. Lenguaje de programación: Python 16
3.3. Simulador Gazebo 16
3.3.1. Modelo del robot NAO simulado 16
3.4. Simulador Webots 19

4 Capa de movimiento para el humanoide 21
4.1. Preparación del modelo simulado 21
4.1.1. Compatibilidad del modelo con ROS2 21
4.1.2. Adición de sensores 27
4.1.3. Estabilidad estática 30
4.1.4. Retoque estético 32
4.2. Editor gráfico e intérprete de movimientos 33
4.2.1. Editor gráfico 34
4.2.2. Intérprete de movimientos 38
4.3. Librería de movimientos 39
4.3.1. Patrones fijos 39
4.3.2. Patrones parametrizables 41
4.3.3. Funciones que utilizan sensores 46
4.4. Aplicación de ejemplo con el humanoide 47

5 Conclusiones 51
5.1. Cumplimiento del objetivo 51
5.2. Competencias empleadas 52
5.3. Competencias adquiridas 52
5.4. Trabajos futuros 53

Bibliografía 54

Índice de figuras

1.1. Aspiradora autónoma 1
1.2. Robot logístico AGV 2
1.3. Robot logístico AMR 2
1.4. Dron de rescate 3
1.5. Robot Da Vinci 3
1.6. UR10e, Robot Colaborativo de Universal Robots 4
1.7. Robot Atlas Eléctrico 5
1.8. Robot Atlas Hidráulico 6
1.9. Robot Optimus 6
1.10. Robot Digit 7
1.11. Robot NAO 8
1.12. Lenguaje Naoqui 9

2.1. Esquema del funcionamiento de la metodología *Scrum* 12

3.1. Modelo del robot NAO utilizado 17
3.2. Inicio de Gazebo Harmonic, para acceder a la demo de control del NAO 18
3.3. Demo de Gazebo Harmonic para control de las articulaciones de NAO 18
3.4. Demo de NAO en Webots 19

4.1. *topics* sólo accesibles desde Gazebo y no desde ROS2 22
4.2. *topics* corregidos para poder hacer el puente 24
4.3. *topics* accesibles desde ROS2 y Gazebo 27
4.4. *topics* finales del robot 30
4.5. Nao retocado estéticamente 33
4.6. Esquema del funcionamiento del editor y el intérprete 34
4.7. Esquema de *topics* y articulaciones de NAO 35
4.8. Editor gráfico de movimientos 37
4.9. Esquema de parametrización de la velocidad 42
4.10. Esquema de funcionamiento de la función *setArc* 45
4.11. Mundo Invernadero para la aplicación de ejemplo 48
4.12. Aplicación GreenNao. Vista lateral 49
4.13. Aplicación GreenNao. Vista frontal 50

Índice de extractos de código

4.1. Inclusión del plugin en el modelo 23
4.2. Estructura de un `gz_bridge` 24
4.3. Ejemplo de launcher para simulación 25
4.4. Adición de la cámara al modelo 27
4.5. Adición del sensor IMU al modelo 28
4.6. Ejemplo de configuración de posición inicial de una articulación . . . 31
4.7. Ejemplo de adicición de sliders al editor 35
4.8. Ejemplo de JSON 36
4.9. Calidad de servicio utilizada para publicación de mensajes 38
4.10. Ejemplo de función que invoca un nodo ROS2 39
4.11. Función setArc 45
4.12. Aplicación GreenNao 48

1. Introducción

El mundo de la robótica está en pleno auge, y cada vez es más común encontrar robots en nuestro entorno cotidiano. Ya sea porque conocemos a alguien que tiene un robot aspirador, hemos visto noticias sobre avances tecnológicos, o incluso hemos interactuado directamente con alguno, su presencia resulta cada vez más habitual.

1.1. Robótica de Servicio

Entre los más comunes se encuentran los robots de servicio, aquellos diseñados para asistir a los humanos en tareas específicas. En este grupo se incluyen dispositivos como aspiradoras autónomas (Figura 1.1), robots logísticos como los AGV (vehículos de guiado automático, Figura 1.2) y los AMR (robots móviles autónomos, Figura 1.3), drones de rescate (Figura 1.4), el robot *Da Vinci*[1] que ayuda a los cirujanos (Figura 1.5), etc.

Figura 1.1: Aspiradora autónoma

[1] http://www.icirugiarobotica.com/cirugia-robotica-da-vinci/

Figura 1.2: Robot logístico AGV

Figura 1.3: Robot logístico AMR

Figura 1.4: Dron de rescate

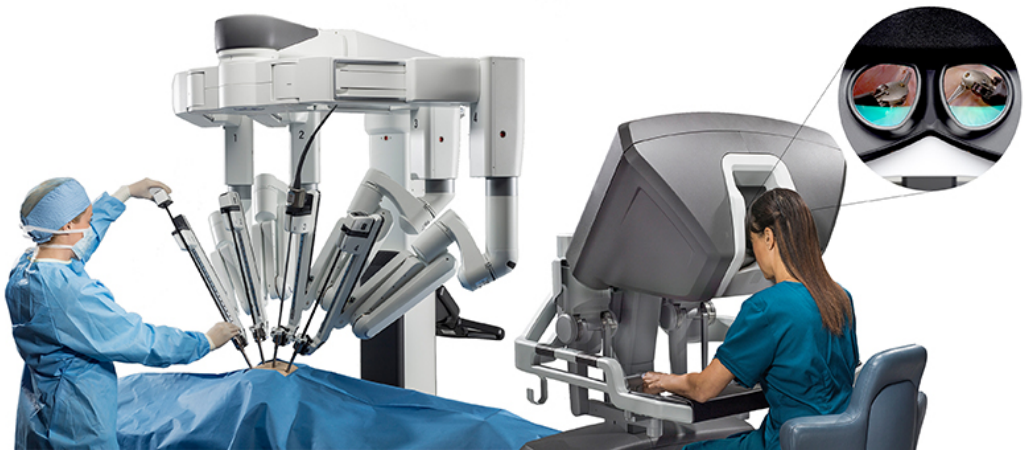


Figura 1.5: Robot Da Vinci

Estos robots también se conocen como robots de consumo, y pueden clasificarse en dos categorías: personales y profesionales. Los primeros operan en el entorno doméstico, mientras que los segundos están diseñados para entornos empresariales, como fábricas, almacenes o zonas de construcción.

Es importante no confundirlos con los robots industriales, que suelen ser brazos robóticos fijos dedicados a realizar tareas repetitivas en un entorno controlado, como una línea de producción. Una excepción son los cobots (robots colaborativos), que están diseñados para interactuar directamente con personas en el ámbito industrial. Un ejemplo de estos robots son los de la empresa *Universal Robots*[2], cuyos robots se

[2] https://www.universal-robots.com/es/

muestran en la Figura 1.6.

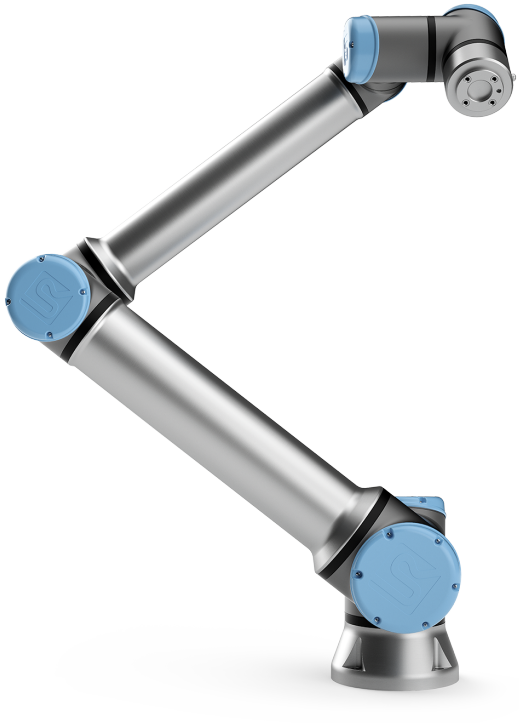


Figura 1.6: UR10e, Robot Colaborativo de Universal Robots

A diferencia de los robots industriales, los robots de servicio o consumo deben desenvolverse en entornos no controlados, e incluso potencialmente hostiles. Además, no están necesariamente limitados a una sola tarea, ya que pueden adaptarse para realizar múltiples funciones y operar en distintos contextos.

1.2. Robots Humanoides

Dentro de los robots de servicio, destacan los robots humanoides, aquellos con la forma de su cuerpo construido para parecerse al cuerpo humano. Estos robots son especialmente interesantes a la hora de realizar tareas que ayuden a las personas. Esto es gracias a su diseño, que los hace lo suficientemente versátiles a la hora de realizar dichas tareas, ya que comparten la misma complexión y anatomía que los humanos.

Un ejemplo de este tipo de robots es el famoso Atlas[3], de Boston Dynamics[4]. Este robot es completamente eléctrico y ha demostrado ser un robot extremadamente ágil, aunque no tanto como su predecesor, el robot Atlas hidráulico, capaz de moverse por entornos extremadamente complicados e incluso hacer acrobacias complejas. Ambos robots pueden verse en la Figura 1.7 y en la Figura 1.8, respectivamente. También se adjuntan vídeos[5][6] de lo que estos robots son capaces de hacer, también respectivamente.

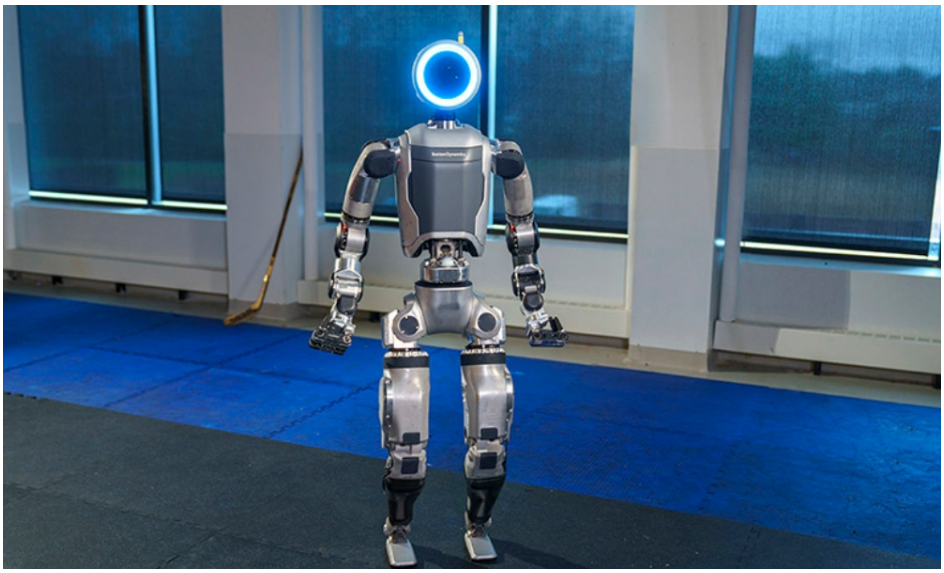


Figura 1.7: Robot Atlas Eléctrico

[3] https://bostondynamics.com/atlas/
[4] https://bostondynamics.com/
[5] https://www.youtube.com/watch?v=I44_zbEwz_w
[6] https://www.youtube.com/watch?v=tF4DML7FIWk

Figura 1.8: Robot Atlas Hidráulico

Pero Atlas no es el único robot humanoide que existe, también tenemos al robot de Tesla[7], Optimus[8], que también ha demostrado ser un robot bastante potente capaz de hacer varias tareas. Se muestra a continuación este robot en la Figura 1.9 y un vídeo[9] para conocerlo mejor.

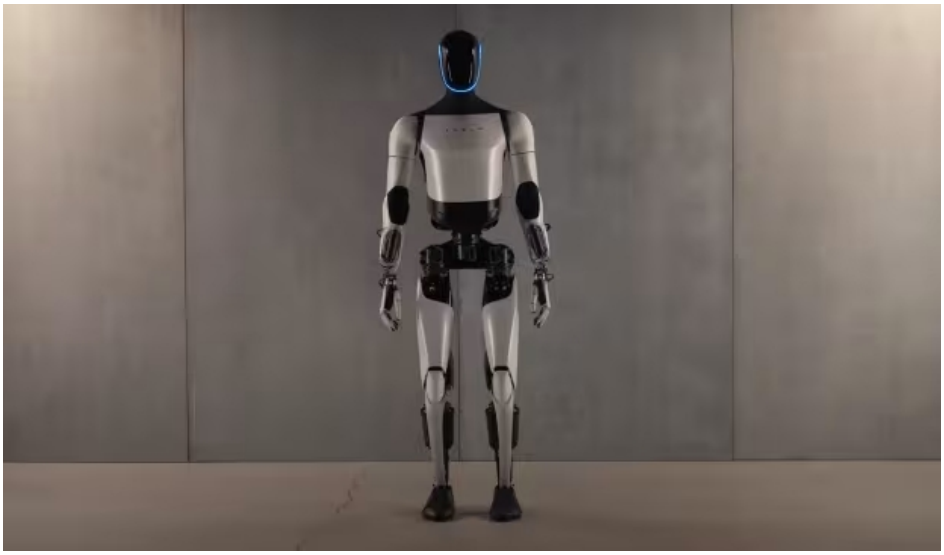


Figura 1.9: Robot Optimus

[7] https://www.tesla.com/es_es
[8] https://www.tesla.com/es_es/we-robot
[9] https://www.youtube.com/watch?v=cpraXaw7dyc

Estos robots son demostradores, que sirven para enseñar hasta dónde somos capaces de llegar con la tecnología más puntera (pero no quiere decir que en el futuro se conviertan en robots de servicio masivos, cosa que aún nos queda un poco lejos).

Robots de servicio humanoides hay pocos, sin embargo, sí existen. Como ejemplo de este hecho tenemos a Digit[10], de Agility Robotics[11], un robot humanoide destinado a la logística que ha demostrado cumplir bastante bien con su tarea; para demostrarlo, se deja a continuación un vídeo[12] donde lo vemos actuar. También se adjunta una fotografía de este robot en la Figura 1.10.

Figura 1.10: Robot Digit

Sin embargo, aunque estos robots son construibles en la vida real, suponen una dificultad muy alta, ya que hay que tener en cuenta muchos factores y desafíos.

Estos desafíos residen especialmente en la locomoción, ya que, al moverse mediante piernas, deben ser estables tanto estática cómo dinámicamente. También deben tener una anatomía adecuada y un aspecto amigable para los humanos, lo que conlleva esquivar *el valle inquietante*, un punto en el que los robots son tan realistas que causan rechazo, e incluso miedo en algunos casos. Es por eso que los que hemos visto anteriormente no son tan antropomórficos ni realistas.

También como desafío están el tamaño y el peso del robot, que, al ser de un tamaño grande, hay que compensar los pesos adecuadamente para conseguir esa

[10] https://www.agilityrobotics.com/solution
[11] https://www.agilityrobotics.com/
[12] https://www.youtube.com/watch?v=q8IdbodRG14

estabilidad estática que se mencionaba anteriormente. En cuanto a la estabilidad dinámica, esto requiere mucho tiempo de entrenamiento del robot en cuanto a modos de caminar, cosa que se suele hacer mediante entrenamiento por refuerzo en simuladores.

Cómo último ejemplo de robot humanoide se hace referencia al robot NAO[1], de Aldebaran, ahora Softbank Robotics[13]. Protagonista de este TFG.

Este robot es el pequeño humanoide de 4,3 kg de peso y 58 cm de altura que se puede ver en la Figura 1.11.

Figura 1.11: Robot NAO

Sin embargo, este robot no es un robot de servicios. Es un robot principalmente educativo, destinado a que las personas aprendan las bases de la robótica y la programación mientras lo usan, además de ser divertido para los niños e interesante para la investigación. De hecho, este robot fue la liga de hardware estandard dentro de la RoboCup soccer[14] durante 17 años consecutivos, desde 2008 hasta 2024.

Este robot también ha protagonizado muchos trabajos de fin de grado

[13] https://us.softbankrobotics.com/
[14] https://www.robocup.org/leagues/5

anteriores, por ser el humanoide más accesible tanto por nuestra universidad (por tenerlo en el laboratorio), como por los estudiantes individualmente al poder acceder a su modelo simulado. Ejemplos de dichos trabajos pueden apreciarse en [2] y [3], dónde se utiliza este robot para explorar modos de caminar, un problema muy común en humanoides y muy interesante para investigar debido a su elevada complejidad.

Cabe destacar también que este robot es muy versátil a la hora de programarlo, ya que se puede optar por su forma predeterminada mediante Naoqui[15], un *framework* específico para programar al robot mediante programación visual que puede verse en la Figura 1.12. Pero también ofrece la posibilidad de programarlo mediante el uso de ROS2, como se hará en este TFG o bien utilizando otros medios como por ejemplo el simulador Webots, al que haremos refrencia en el capítulo 3.

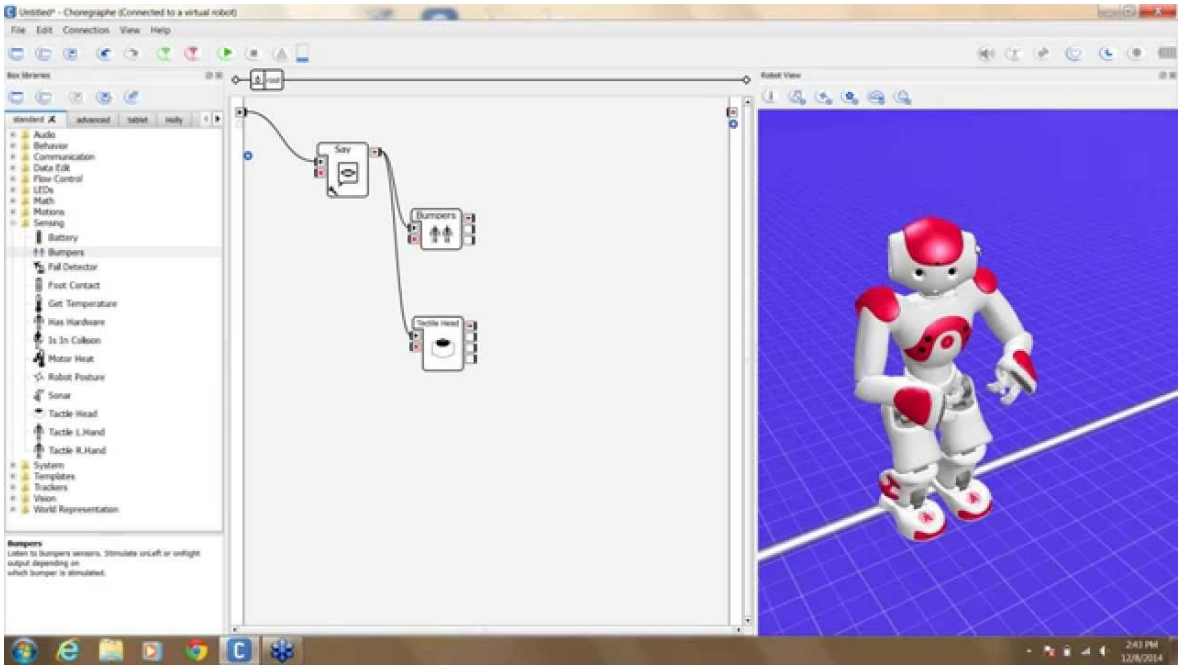


Figura 1.12: Lenguaje Naoqui

1.3. Simuladores Robóticos

Los simuladores son aplicaciones software que permiten emular *via software* de manera realista al robot y su entorno, lo que los convierte en una herramienta extremadamente útil a la hora de construir todo tipo de robots, ya que evitan posibles catástrofes en la realidad a la hora de usarlos, por ejemplo, si un robot cae

[15] http://doc.aldebaran.com/1-14/dev/naoqi/index.html

en el entorno simulado, sus componentes reales quedarán a salvo. Es por eso por lo que tienen mucha importancia a la hora del desarrollo de robots humanoides, debido a su elevado coste y complicación a la hora de construirlos. Los simuladores también son útiles para entrenar dichos robots en diferentes ámbitos y escenarios, para así brindarles la máxima versatilidad posible y que puedan realizar casi cualquier tarea que podría hacer un humano.

Existen muchos simuladores robóticos, entre los que destacan el simulador de NVIDIA[16], Isaac Sim[17], capaz de entrenar un robot para que aprenda a caminar en sólamente 2 horas, según lo indicado por CyberRobo en la plataforma X[18].

Otro simulador muy presente en la actualidad es Coppelia Sim[19], un simulador utilizado en la industria, la educación y la investigación. Originalmente fue desarrollado dentro del departamento de I+D de Toshiba[20] y actualmente está siendo desarrollado y mantenido activamente por Coppelia Robotics AG[21], una pequeña empresa ubicada en Zúrich, Suiza. Este simulador ha resultado muy útil para simulación de robots móviles con navegación autónoma, Robots que usan SLAM para mapear un entorno, brazos robóticos con *pick and place*, e incluso simulación de robots humanoides caminantes.

También existe el simulador Webots, un simulador *opensource* educativo en el que se hará más incapié en el capítulo 3.

Por último, destaca también Gazebo, simulador específico para ROS2, muy potente y además *opensource*. Éste es el simulador que se utilizará para el desarrollo de este TFG. Se hablará más en profundidad sobre él en el Capítulo 3.

Todo lo expuesto en este capítulo ha sido clave para decidir como objetivo dotar al robot NAO simulado en Gazebo de una librería de coordinación de actuadores que le permita caminar (especialmente por el desafío técnico que supone) y, además, programar una aplicación en el contexto de un invernadero como validación experimental de la biblioteca de movimientos y de su potencial uso como robot de servicio.

[16] https://www.nvidia.com/es-es/
[17] https://developer.nvidia.com/isaac/sim
[18] https://x.com/CyberRobooo/status/1921252216330912032
[19] https://www.coppeliarobotics.com/
[20] https://www.toshiba.es/
[21] https://github.com/CoppeliaRobotics

2. Objetivos

2.1. Problema a resolver

Como se ha introducido en el Capítulo 1, el objetivo principal de este TFG es dotar al humanoide NAO de una capa completa de movimientos y locomoción, para luego programar una aplicación que empleará este robot educativo en el ámbito de la robótica de servicios. Esta aplicación servirá además de validación para demostrar que dicha capa de movimiento es funcional e intuitiva de utilizar.

Sin embargo, este objetivo es muy general y amplio, así que lo mejor es dividirlo en subobjetivos, que se detallan a continuación.

- *Subobjetivo 1*: Como primer subobjetivo tenemos el ser capaces de crear patrones fijos de movimiento y que NAO pueda replicarlos fielmente y con los menores errores posibles.
- *Subobjetivo 2*: El segundo subobjetivo de este proyecto es dotar al humanoide de la capacidad de parametrizar sus movimientos y poder ofrecer modos de caminar distintos y con la velocidad variable, obteniendo así movimientos continuos parametrizables. Estos movimientos deben encapsularse en una librería para que su uso sea más sencillo y directo. Dicha librería contendrá además toda la funcionalidad de ROS2 y los patrones fijos ofrecidos por el robot (desarollados para el subobjetivo 1), para que así sea una biblioteca utilizable por cualquier usuario de manera sencilla.
- *Subobjetivo 3*: Como tercer y último subobjetivo, tenemos el desarrollar una aplicación que sirva de demostradora de la potencia de dicha libería, haciendo que el robot preste servicio en un invernadero.

  El servicio que debe ofrecer es similar al visto en el robot Digit, esto es, mover una caja de un lugar de origen a uno de destino, pero, en lugar de en un almacén, se hará en un invernadero para dar más identidad y originalidad al proyecto.

2.2. Metodología

Para alcanzar los subobjetivos planteados y, en última instancia, cumplir con el objetivo principal del proyecto, se ha optado por adoptar la metodología *Scrum* como marco de trabajo. Esta elección responde a la necesidad de mantener una dinámica de desarrollo iterativa y adaptable (Esquema del funcionamiento de esta metodología en la Figura 2.1).

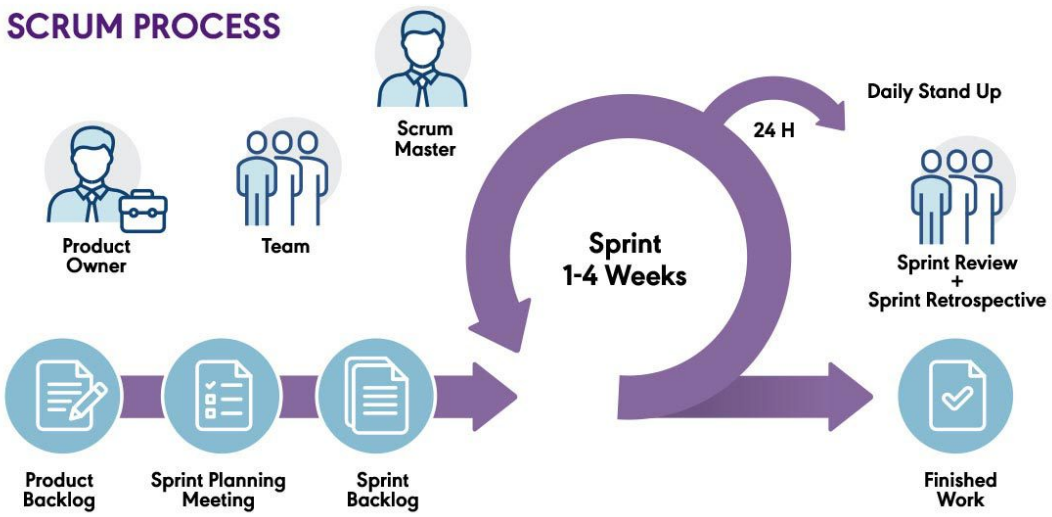


Figura 2.1: Esquema del funcionamiento de la metodología *Scrum*

En este contexto, se han establecido reuniones semanales con el tutor del proyecto, que cumplen la función de revisiones periódicas del progreso similares a las *Sprint Reviews*, ya que, durante estas sesiones, se evalúan los avances logrados en la última iteración y se definen de manera colaborativa los siguientes pasos a seguir. Este enfoque permite ajustar la planificación de forma continua en función de los resultados obtenidos, fomentar la mejora progresiva del proyecto, y asegurar que cada etapa del desarrollo esté alineada con los objetivos generales. De esta manera, se garantiza un proceso flexible, enfocado en la entrega constante de valor y la toma de decisiones informadas a lo largo del ciclo de desarrollo.

En resumen, la metodología que se ha aplicado estaba basada en *sprints* de *Scrum* semanales los cuales iba marcando mi tutor, en función a los objetivos cumplidos del sprint anterior.

2.2.1. Plan de trabajo

El primer *sprint* de trabajo fue elegir una plataforma para alojar el proyecto y facilitar el seguimiento del mismo.

Para cumplir con eso, se ha decidido alojar el código fuente generado en un repositorio de GitHub[1], plataforma que ayuda a los desarrolladores a almacenar y gestionar su código, así como a rastrear y controlar los cambios en él, lo que lo convierte en la plataforma de control de versiones preferida a la hora de desarrollar proyectos grandes, como es el caso de este TFG.

Que sea un sistema de control de versiones nos ayuda a tener registradas todas y cada una de las versiones del proyecto, pudiendo volver a una versión anterior o seguir avanzando a placer. Esto es de gran ayuda por si es necesario volver a una versión anterior por cualquier razón.

El repositorio dónde se aloja este proyecto[2] sigue la siguiente estructura:

- *Directorio CoordMoves*: Es en este directorio donde se aloja el proyecto completo, códigos, modelos, mundos, este documento, etc.
- *Directorio pruebas*: En este directorio fue donde se trabajó durante la fase experimental del proyecto, con todo lo que eso conlleva.
- *Directorio docs*: Directorio dónde se aloja un blog de seguimiento semanal del proyecto. Esto se tratará con profundidad más adelante.
- Fichero *README.md*: Un fichero en el que se describe de forma resumida todo el proyecto, ya que éste es público.

También se ha decidido utilizar Github Pages[3], un servicio de alojamiento de sitios estáticos que toma archivos HTML, CSS y JavaScript directamente de un repositorio en GitHub, los ejecuta opcionalmente mediante un proceso de compilación y publica un sitio web. Ha sido utilizado en el proyecto para llevar un blog semanal[4], en el que se comentan avances, percances o ideas que han surgido durante el desarrollo del proyecto, desde su inicio, hasta su fin.

Este blog es una forma muy potente de documentar el proceso de desarrollo y poder seguir el esquema *Scrum*, ya que, no sólo es interesante para ver cómo se ha hecho o qué se ha hecho, si no que ha servido de mucha ayuda para que mi

[1] https://github.com/
[2] https://github.com/RoboticsLabURJC/2024-tfg-eva-fernandez
[3] https://pages.github.com/
[4] https://roboticslaburjc.github.io/2024-tfg-eva-fernandez/

tutor tuviera un seguimiento más rígido del proceso y se pudieran abarcar todas las cuestiones y avances realizados en las reuniones.

Los sprints siguientes fueron más variados, cómo buscar un modelo, prepararlo junto a un primer mundo, descubrir cómo controlar este modelo con ROS2, etc. Lo que nos dejó 3 fases principales a la hora del desarrollo:

- *Fase 1*: Preparación del modelo y un mundo vacío
- *Fase 2*: Desarrollo de un intérprete de movimientos y un editor
- *Fase 3*: Implementación de la librería y llenado del mundo
- *Fase 4*: Desarrollo de la aplicación

3. Herramientas Software utilizadas

En este capítulo, se explicarán las herramientas software utilizadas para llevar a cabo este TFG, es decir, se comentarán los recursos que han servido para la programación y estructuración del proyecto.

3.1. Ecosistema: Middleware ROS 2

ROS[1] es un middleware que ofrece un conjunto de bibliotecas y herramientas de software que ayudan a crear aplicaciones robóticas. Desde controladores hasta algoritmos de vanguardia, y con potentes herramientas de desarrollo, tiene todo lo necesario para consruir un proyecto de robótica. Y todo es de código abierto.

Tiene dos versiones, ROS (también conocida cómo ROS1, es la primera que salió y la más antigua) y ROS2, la más moderna y la que ha sido utilizada para este proyecto. Además de ser el middleware por excelencia utilizado a lo largo de la carrera por diferentes asignaturas, ROS2 es una potente herramienta utilizada alrededor del mundo para manejar robots.

Dispone de varias distribuciones que van saliendo cada cierto tiempo, cada una compatible con una distribución de Ubuntu concreta. En mi caso, se ha utilizado la anterior dsitribución de ROS2, Humble Hawksbill[2], debido a que la distribución Ubuntu utilizada ha sido la 22.04 LTS, distribución directamente compatible con Humble.

Para trabajar con ROS, es necesario crear paquetes [4] para alojar los diferentes programas para publicar o suscribirse a los *topics* del robot [5] y especificaciones necesarias para la aplicación robótica a desarrollar, y pueden crearse directamente para los lenguajes C++, o Python, de los cuales se ha optado por el segundo.

[1] https://www.ros.org/

[2] https://docs.ros.org/en/humble/Installation.html

3.2. Lenguaje de programación: Python

Como bien se ha dicho anteriormente, el lenguaje de programación utilizado ha sido Python[3], lenguaje de alto nivel de programación interpretado cuya filosofía hace hincapié en la legibilidad de su código. Se trata de un lenguaje de programación multiparadigma, ya que soporta parcialmente la orientación a objetos, programación imperativa y, en menor medida, programación funcional. Es un lenguaje interpretado, dinámico y multiplataforma.

Se ha optado por este lenguaje porque está ampliamente extendido y es fácil de interpretar y utilizar, además de que es compatible con ROS2, pudiendo crear paquetes directamente para este lenguaje. La versión utilizada ha sido la 3.10, versión por defecto instalada en Ubuntu 22.04.

3.3. Simulador Gazebo

Una vez elegidos el middleware y el lenguaje de programación, es necesario elegir un robot y un escenario. Cómo adquirir un robot es algo costoso y mi situación personal no permitía trabajar con los robots que ofrece nuestro laboratorio en la URJC, se optó por trabajar en un entorno completamente simulado.

Para ello, Gazebo[4] fue la mejor opción, por su directa compatibilidad con ROS2 y potencia de simulación, además de que, como ROS, es de código abierto.

A la hora de elegir la versión a utilizar, se decidió usar la versión más reciente del simulador, Gazebo Harmonic, para así no quedarnos tan atrás a la hora de desarrollar, ya que usábamos la versión anterior de ROS2.

3.3.1. Modelo del robot NAO simulado

Una vez escogido el entorno de simulación, era hora de buscar un robot adecuado para el proyecto. Como ya se ha mencionado, nuestro protagonista es el Robot NAO, de Aldebaran, por lo que era necesario encontrar un modelo en el formato de Gazebo (SDF) para poder hacerlo funcionar.

[3] https://www.python.org/
[4] https://gazebosim.org/home

Para esto, Gazebo dispone de una amplia biblioteca[5] de mundos y modelos ya creados por empresas o la comunidad, así que sólo era buscarlo en esta biblioteca.

Al final, el modelo utilizado sería el ofrecido por OpenRobotics, que se muestra a continuación:

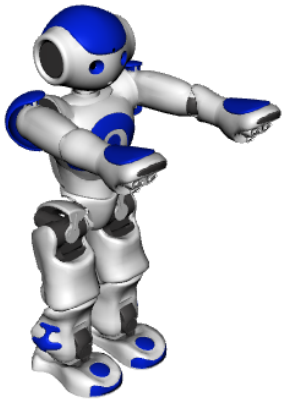


Figura 3.1: Modelo del robot NAO utilizado

Este modelo[6] cuenta con un sistema de control de articulaciones ya construido, por lo que resulta perfecto para el proyecto. De hecho, el propio Gazebo Harmonic, al abrirse, tiene una demo que prueba este control de los *joints* de este mismo modelo, cosa que puede verse en la Figura 3.2.

A continuación se adjunta una captura de pantalla de dicha demostración (Figura 3.3), así cómo un vídeo[7] para que se aprecie bien el funcionamiento.

[5] https://app.gazebosim.org/dashboard

[6] https://app.gazebosim.org/OpenRobotics/fuel/models/NAO%20with%20Ignition%20position%20controller

[7] https://youtu.be/y6Rn2WOl_Xw

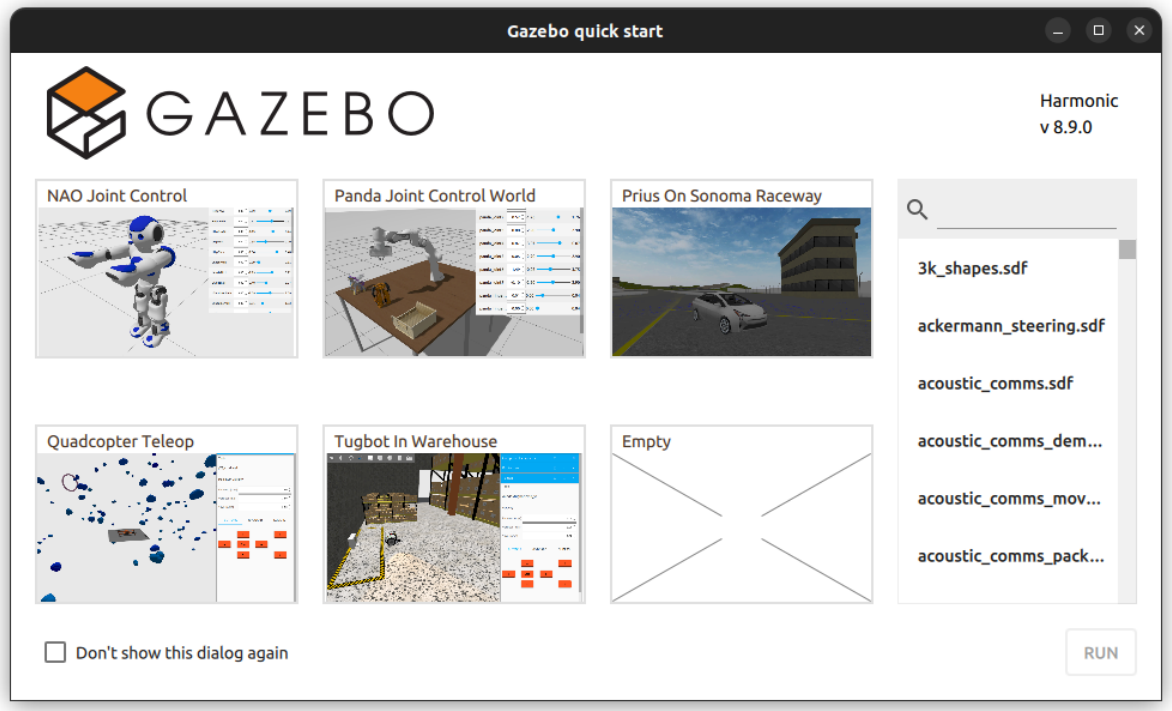


Figura 3.2: Inicio de Gazebo Harmonic, para acceder a la demo de control del NAO

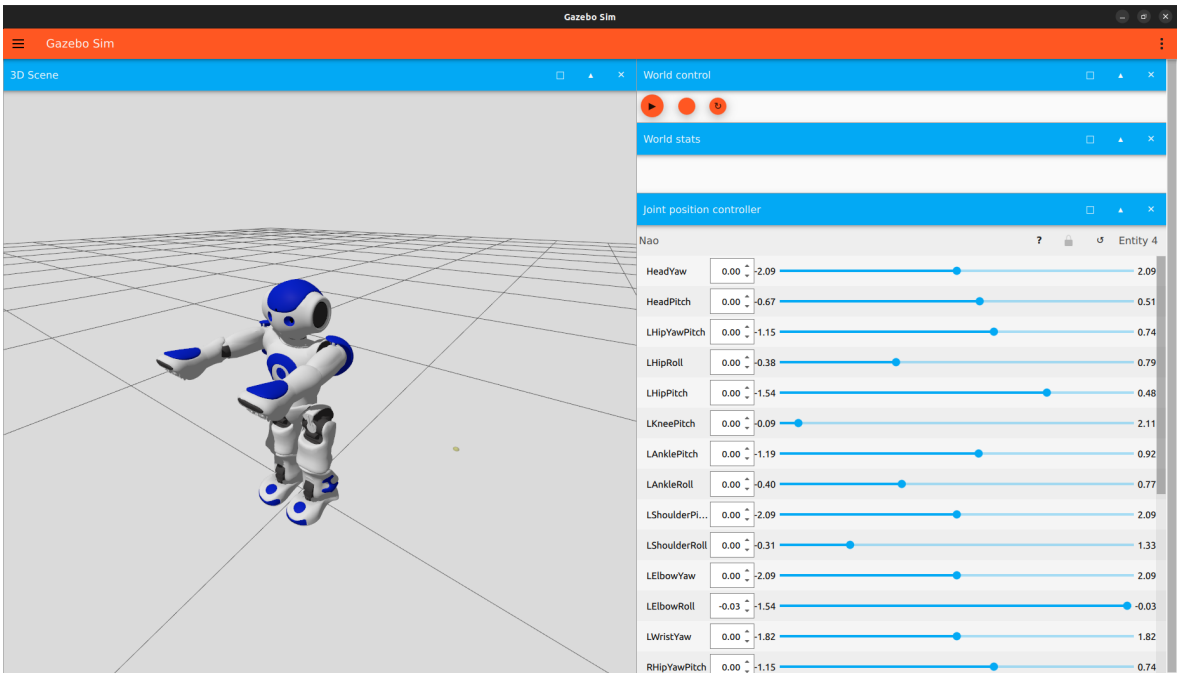


Figura 3.3: Demo de Gazebo Harmonic para control de las articulaciones de NAO

El soporte básico de este modelo es ofrecer acceso a los actuadores de las articulaciones individualmente, sin embargo, no ofrece ningún mecanismo de coordinación o locomoción que involucre a varias articulaciones de forma

ordenada o coordinada. Pero el uso de Gazebo y este modelo en particular es necesario para probar los mecanismos de coordinación desarrollados en este TFG, y también para dotar al robot de un escenario para validar la aplicación robótica creada.

3.4. Simulador Webots

Webots[8] es una aplicación de escritorio multiplataforma de código abierto que se utiliza para simular robots. Proporciona un entorno de desarrollo completo para modelar, programar y simular robots.

Diseñado para uso profesional, se utiliza ampliamente en la industria, la educación y la investigación. Cyberbotics Ltd.[9] ha mantenido Webots como su producto principal desde 1998. Se muestra una imagen de una demo del robot NAO disponible en este simulador en la Figura 3.4

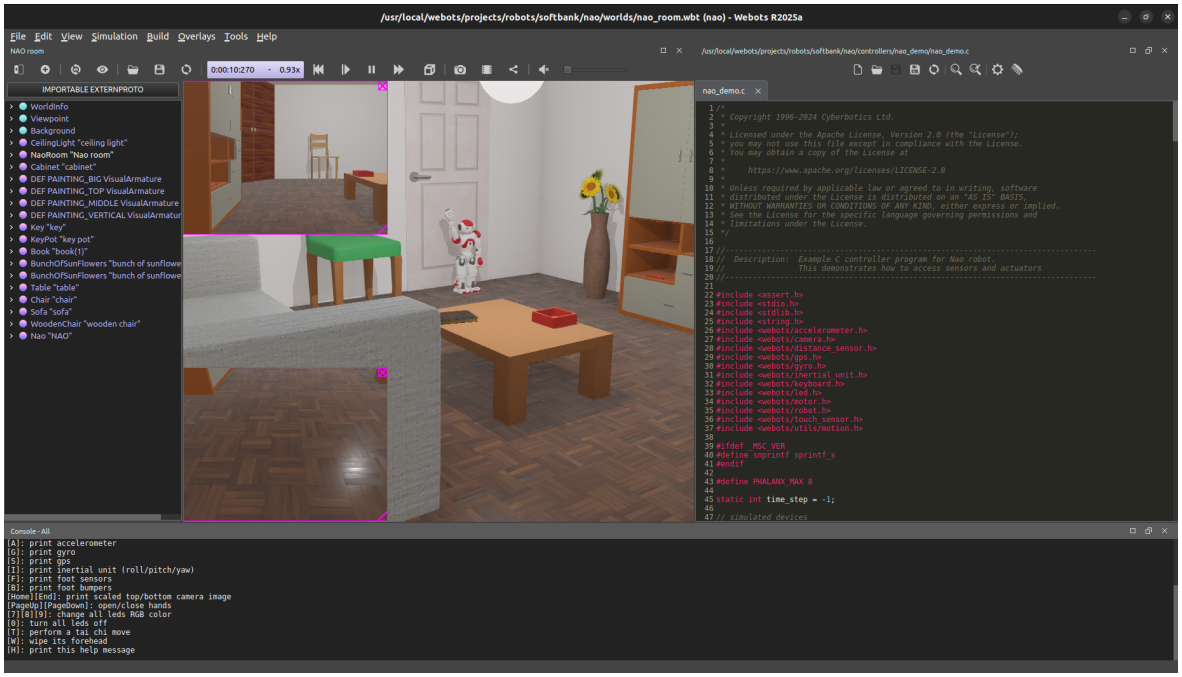


Figura 3.4: Demo de NAO en Webots

Se ha utilizado la versión R2025a, que, además de ser la más moderna, ya tiene resueltos patrones de movimiento del robot humanoide NAO que se han reciclado para el desarrollo de este TFG, los cuales son los siguientes:

[8] https://cyberbotics.com/

[9] https://es.linkedin.com/company/cyberbotics

- Caminar hacia adelante una secuencia de 10 pasos.
- Caminar hacia atrás una secuencia de 2 pasos.
- Levantarse del suelo si se ha caído boca abajo.
- Caminar de forma lateral una secuencia de 2 pasos hacia la derecha.
- Caminar de forma lateral una secuencia de 2 pasos hacia la izquierda.
- Girar en el sitio hacia la derecha un ángulo de 40 grados.
- Girar en el sitio hacia la derecha un ángulo de 60 grados.
- Girar en el sitio hacia la izquierda un ángulo de 40 grados.
- Girar en el sitio hacia la izquierda un ángulo de 60 grados.
- Girar en el sitio hacia la izquierda un ángulo de 180 grados.

El uso de estos patrones y sus modificaciones se extenderá en el Capítulo 4.

4. Capa de movimiento para el humanoide

Una vez explicadas las herramientas utilizadas en el capítulo anterior, en este se explicará el proyecto como tal, desglosando adecuadamente cada una de las partes involucradas en él.

4.1. Preparación del modelo simulado

Cómo se mencionó en el capítulo 3, el modelo utilizado fue extraído de la librería abierta de Gazebo. Sin embargo, este modelo no era suficiente para poder desarrollar todo el proyecto directamente y fue necesario prepararlo y adaptarlo.

4.1.1. Compatibilidad del modelo con ROS2

Lo primero que se hizo para poder usar el modelo fue preparar los *topics* de ROS2 necesarios para publicar posiciones en sus articulaciones y así hacer el modelo compatible con este middleware, ya que como se aprecia en la Figura 4.1, tenía los *topics* preparados para Gazebo, y no para ROS. Esto es porque en la versión de Gazebo de 2018, Gazebo Ignition/Gazebo Sim, el simulador se rediseñó desde cero con la idea de ser más independiente de ROS, para que así los usuarios no estuvieran obligados a utilizar dicho middleware para lanzar simulaciones.

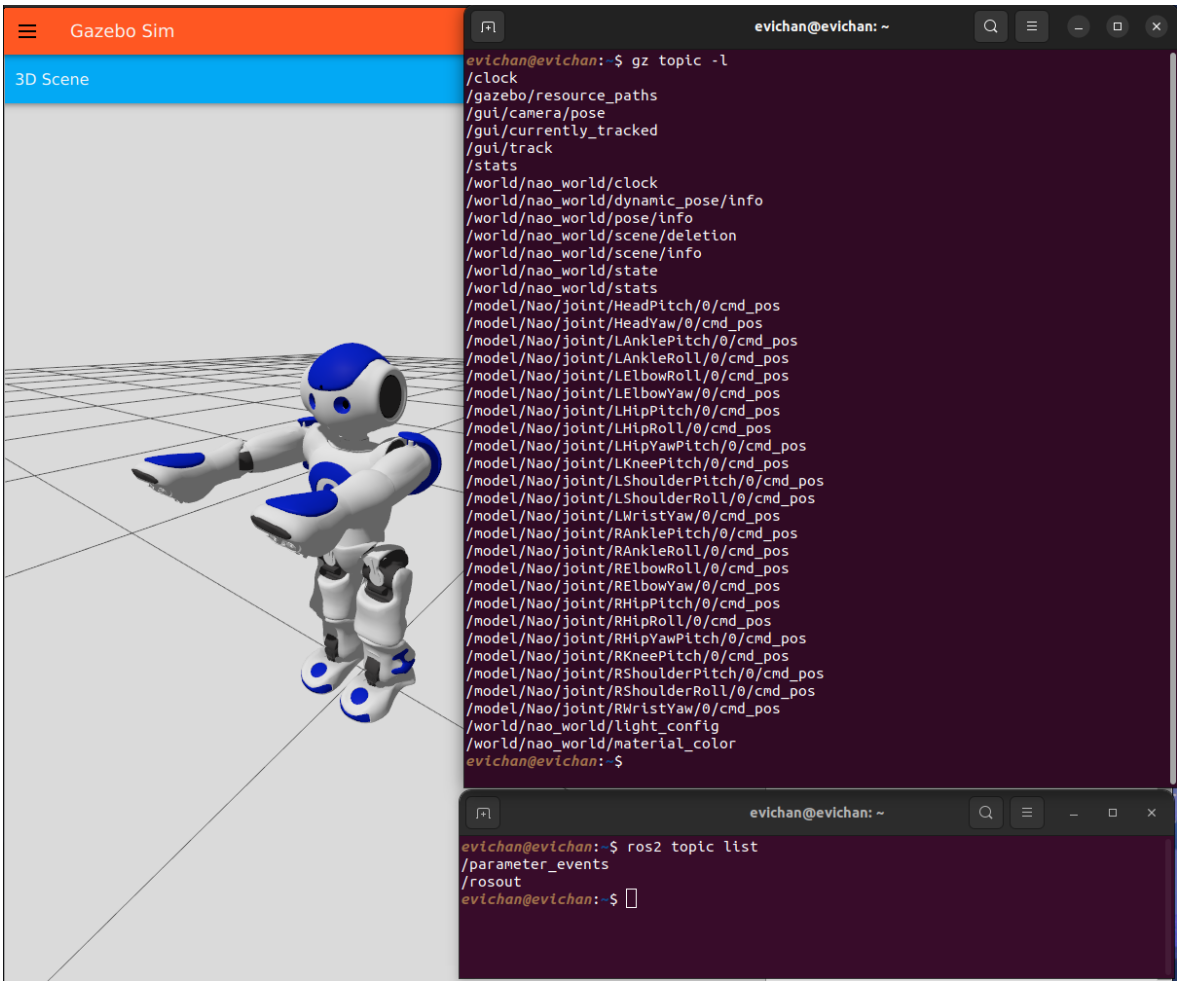


Figura 4.1: *topics* sólo accesibles desde Gazebo y no desde ROS2

La solución para este pequeño problema es utilizar un puente que conecta los *topics* de Gazebo con los *topics* de ROS2, disponible en el paquete *ros_gz_bridge*[1], que se encarga de que los *topics* de Gazebo sean también visibles al ejecutar el comando `ros2 topic list`, y no solo al ejecutar `gz topic -l`, permitiendo trabajar con ellos desde ROS2.

Para aplicar este puente, se instala el paquete con `sudo apt install ros-humble-ros-gz-bridge`. Después, es necesario crear el paquete de ROS que será usado para alojar todos los programas que se iban a lanzar y, además de eso, se debe crear un *launcher.py*, encargado de lanzar la simulación y todos los puentes necesarios para los *topics* de NAO accesibles en Gazebo (todos aquellos que siguen la estructura */model/NAO/joint/..../0/cmd_pos*).

Sin embargo, los nombres de los *topics* de Gazebo contienen el número 0 en sus nombres, cosa incompatible con los *topics* de ROS2. Por lo que el primer cambio que

[1] https://github.com/gazebosim/ros_gz/tree/ros2/ros_gz_bridge

se tuvo que hacer al modelo fue cambiar el nombre de todos los *topics* relacionados con el movimiento del robot, para así eliminar este número 0.

Para ello, en el fichero SDF del robot, primero se tenía que localizar dónde se alojaban esos *topics*, ya que el modelo no disponía de una etiqueta *topic* o similar. Esto es porque que para poder crear estos *topics*, Gazebo utiliza *plugins*[2].

Estos plugins son fragmentos de código (generalmente en C++) que se cargan en tiempo de ejecución dentro del simulador para extender o modificar su comportamiento, permitiendo agregar funcionalidades personalizadas a los modelos o mundos.

En el caso de los *topics*, el *plugin* responsable se especifica debajo de cada *joint* (articulación) en el archivo SDF del robot NAO de la forma mostrada en el Extracto de código 4.1:

```
<plugin
  filename="ignition-gazebo-joint-position-controller-system"
  name="ignition::gazebo::systems::JointPositionController">
  <joint_name>HeadYaw</joint_name>
  <p_gain>10</p_gain>
  <i_gain>0.1</i_gain>
  <d_gain>0.01</d_gain>
  <i_max>1</i_max>
  <i_min>-1</i_min>
  <cmd_max>1000</cmd_max>
  <cmd_min>-1000</cmd_min>
</plugin>
```

Extracto de código 4.1: Inclusión del plugin en el modelo

En dicho código se especifican todas las condiciones necesarias para que la articulación se comporte como un motor real, incluyendo incluso un controlador PID, con sus ganancias p_gain, i_gain y d_gain. Lo importante aquí es que el nombre del *topic* no está definido, y es por eso por lo que se recoge el nombre por defecto, el que incluye el número cero en su nombre.

Para cambiar esto, sólo es necesario añadir la línea `<topic>nombre_deseado</topic>` en cualquier lugar de la definición del *plugin*. Una vez hecho esto para todos los *topics*, quedaron como se muestra en la Figura 4.2:

[2] https://gazebosim.org/libs/plugin/

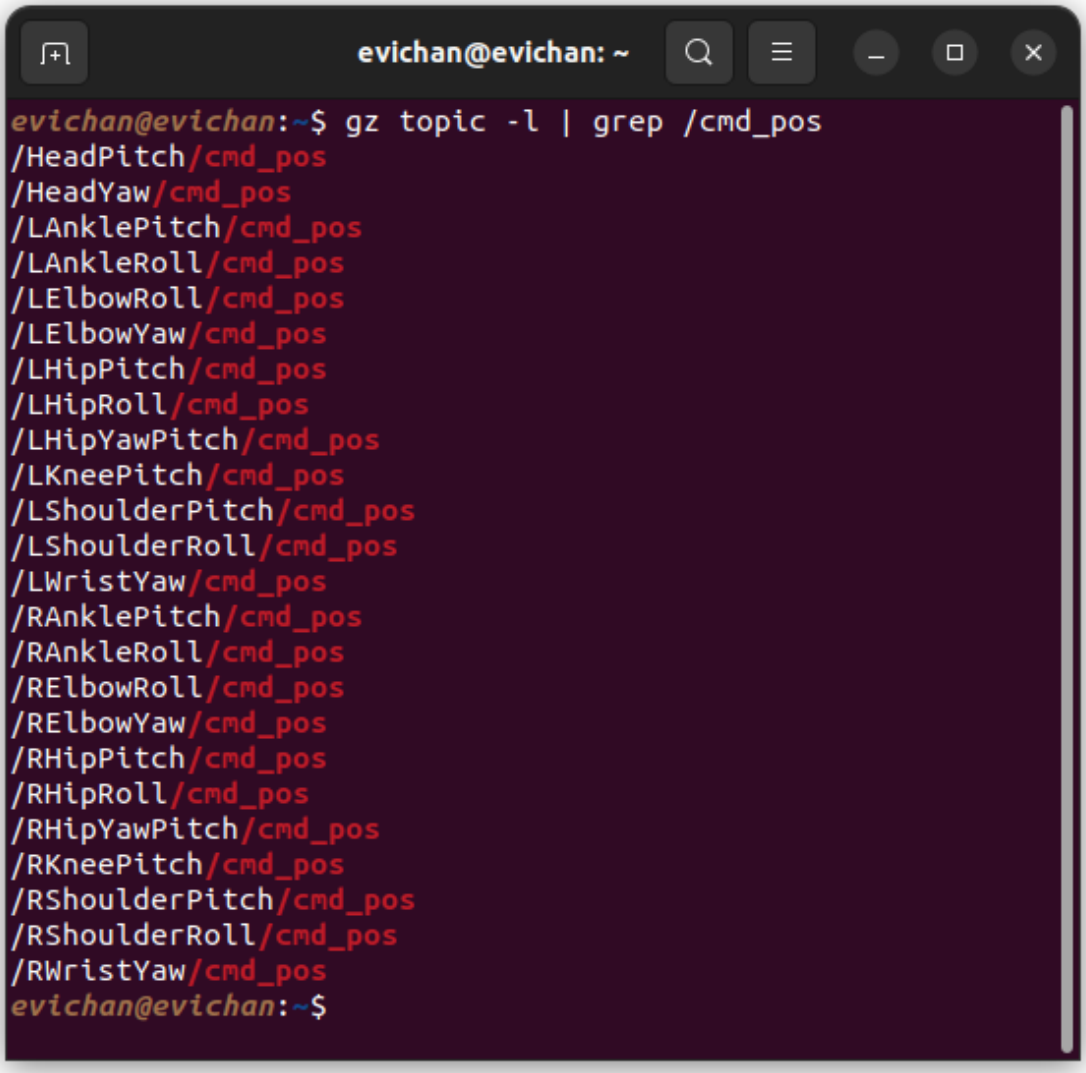


Figura 4.2: *topics* corregidos para poder hacer el puente

Ahora, los *topics* están en el formato correcto para ROS2, por lo que el puente es realizable.

Para crear este puente es necesario incluir en el lanzador del simulador los comandos necesarios para ello. Esto se hace añadiendo las líneas de código que se muestran en el Extracto de código 4.2 al *launcher.py* ([6]):

```
gz_bridge_1 = Node(
    package="ros_gz_bridge",
    executable="parameter_bridge",
    name="gz_bridge",
    arguments=[
```

```
        "nombre_topic" + "direccion_del_puenteTipo_de_mensaje_ros" + "
    direccion_del_puenteTipo_de_mensaje_gazebo"
    ],
    output="screen",
)
```

Extracto de código 4.2: Estructura de un `gz_bridge`

Para la dirección del puente, hay que escribir uno de los siguientes símbolos:

- `[`: Puente unidireccional desde Gazebo a ROS. Este puente sólo permite que ROS se suscriba a los *topics*, pero no puede publicar en ellos.
- `]`: Puente unidireccional desde ROS a Gazebo. Este es el caso contrario al anterior, ROS puede publicar, pero no suscribirse.
- `@`: Puente bidireccional. Este puente permite mensajes en ambas direcciones, por lo que ROS puede publicar o suscribirse sin problemas.

En este caso, se ha utilizado un puente bidireccional en todos los *topics*, para poder publicar mensajes y suscribirnos a ellos si es necesario.

Para los tipos de mensajes utilizables, es necesario consultar la compatibilidad entre ellos en la tabla[3] dada en el repositorio oficial del paquete *ros_gz_bridge*, no sin antes consultar qué tipo tienen los *topics* de Gazebo utilizando el comando `gz topic -i -t nombre_del_topic`.

Una vez definido el tipo de mensajes a utilizar (en el caso de este proyecto, *std_msgs/msg/Float64 para* ROS y *gz.msgs.Double* para Gazebo), se desarrolló de forma sencilla el *launcher* del robot con los *topics* de sus articulaciones compatibles con ROS2. Para ello, se siguió el esquema mostrado en la Extracto de código 4.2 y se usaron los componentes necesarios para crear un lanzador adecuado. Un fragmento de dicho *launcher* se ve en el Extracto de código 4.3.

```
#!/usr/bin/env python3
# Importaciones necesarias
def generate_launch_description():
    set_gazebo_version = SetEnvironmentVariable(
        name="GAZEBO_VERSION", value="8.9"
    )
    # Para poder lanzar el mundo
```

[3] https://github.com/gazebosim/ros_gz/tree/ros2/ros_gz_bridge/README.md

```
    world_file = PathJoinSubstitution(['/home/evichan/Desktop/2024-tfg-eva-fernandez/GreenNao/greennao/worlds/greenhouse_world', 'greenhouse_world.sdf'])
    declare_world_arg = DeclareLaunchArgument(
        "world", default_value=world_file, description="SDF world file"
    )
    # Para lanzar simulacion de gazebo
    gz_sim = IncludeLaunchDescription(
        PythonLaunchDescriptionSource(
            PathJoinSubstitution(
                [
                get_package_share_directory("ros_gz_sim"),
                    "launch",
                    "gz_sim.launch.py",
                ]
            )
        ),
        launch_arguments={"gz_args": world_file}.items(),
    )
    # Ros bridges para controlar articulaciones de NAO van aqui, uno por grado de libertad
    [...]
    return LaunchDescription(
        [
            declare_world_arg,
            set_gazebo_version,
            SetParameter(name="use_sim_time", value=True),
            gz_sim,
            gz_bridge_1,
            [...] # resto de bridges
        ]
    )
```

Extracto de código 4.3: Ejemplo de launcher para simulación

Una vez ejecutado este *launcher,* además de ver el mundo creado y la simulación completa, podemos ver en la terminal cómo ahora los *topics* se comparten entre ROS2 y Gazebo. Un ejemplo de esto se muestra en la Figura 4.3.

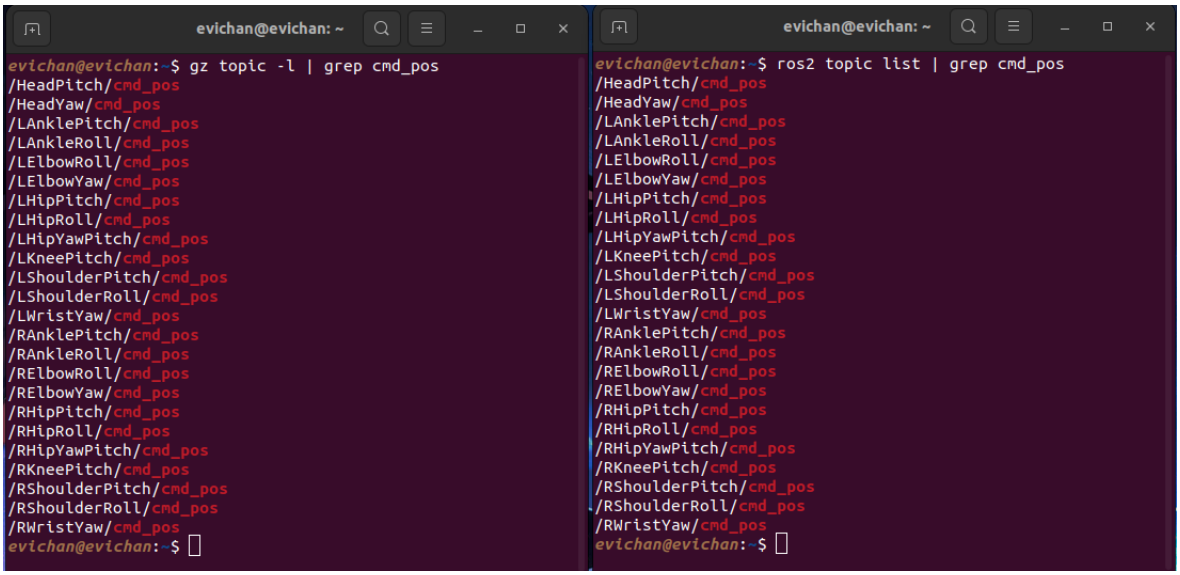


Figura 4.3: *topics* accesibles desde ROS2 y Gazebo

4.1.2. Adición de sensores

No sólo fue necesario hacer el modelo compatible con ROS2, sino que también fue necesario enriquecer un poco al robot dotándole de sensores, ya que el modelo de Gazebo no los ofrecía. Se optó por añadir una cámara y un sensor IMU para que fueran usables en el futuro, además de que son sensores muy utilizados en robótica en general.

Cámara

Para añadir una cámara al robot y dotarle de visión, fue necesario insertarle al modelo los campos necesarios para que tuviera una cámara completamente funcional y usable con ROS2, así que, para ello, se añadieron los elementos mostrados en el Extracto de código 4.4 al SDF del modelo.

```
# Definicion del joint necesario para anclar la camara al torso del robot
[...]
<link name="camera_rgb_frame">
  # Definicion de masa e inercia para en sensor
  [...]
  <sensor name="camera" type="camera">
    <always_on>true</always_on>
    <visualize>true</visualize>
    <update_rate>30</update_rate>
```

```
    <topic>NAO/camera/image_raw</topic>
    <gz_frame_id>camera_rgb_frame</gz_frame_id>
    <camera name="intel_realsense_r200">
      <camera_info_topic>NAO/camera/camera_info</camera_info_topic>
      <horizontal_fov>1.02974</horizontal_fov>
      <image>
        <width>1920</width>
        <height>1080</height>
        <format>R8G8B8</format>
      </image>
      <clip>
        <near>0.02</near>
        <far>300</far>
      </clip>
      <noise>
        <type>gaussian</type>
        <mean>0.0</mean>
        <stddev>0.007</stddev>
      </noise>
    </camera>
  </sensor>
</link>
<plugin filename="gz-sim-sensors-system" name="gz::sim::systems::Sensors"
    >
    <render_engine>ogre2</render_engine>
</plugin>
```

Extracto de código 4.4: Adición de la cámara al modelo

Sensor IMU

Para que el robot pudiera localizarse utilizando una IMU, fue necesario añadir los siguientes campos al modelo dentro del *link* del torso:

```
<sensor name="imu_sensor" type="imu">
    <pose>0 0 0 0 0 0</pose>
    <always_on>true</always_on>
    <update_rate>50</update_rate>
    <visualize>true</visualize>
```

```
    <topic>NAO/imu_sensor</topic>
    <imu>
    <angular_velocity>
        <noise>
        <type>gaussian</type>
        <mean>0.0</mean>
        <stddev>0.01</stddev>
        </noise>
    </angular_velocity>
    <linear_acceleration>
        <noise>
        <type>gaussian</type>
        <mean>0.0</mean>
        <stddev>0.01</stddev>
        </noise>
    </linear_acceleration>
    </imu>
</sensor>
</link>
<plugin filename="gz-sim-imu-system" name="gz::sim::systems::Imu">
    <topic>NAO/imu_sensor</topic>
</plugin>
```

Extracto de código 4.5: Adición del sensor IMU al modelo

Después de añadir estos sensores al SDF del modelo, se añadieron también los puentes necesarios al *launcher* para que ROS pudiera acceder a sus lecturas, de la misma forma vista anteriormente en el Extracto de código 4.3.

Una vez los sensores fueron añadidos correctamente, los *topics* finales del robot eran los siguientes:

Figura 4.4: *topics* finales del robot

4.1.3. Estabilidad estática

Un problema que tienen los robots humanoides, como se comentaba en el capítulo 1, es que tienen que ser estables tanto dinámica cómo estáticamente, y el NAO no es una excepción.

Cuando se probó el robot en el mundo por primera vez, éste caía de boca al suelo, cosa que indicaba que el modelo no era estáticamente estable.

Para dotarle de esta estabilidad, se compensó el peso del humanoide con las posiciones iniciales de sus articulaciones y su posición inicial, a base prueba y error.

Cambiar la posición inicial no tiene ningún misterio, ya que simplemente es

cambiar el campo *pose* que está al incio del modelo. Sin embargo, las posiciones iniciales de las articulaciones tuvieron más complicación. Para asignarlas, fue necesario añadir los campos `<initial_position>` y `<use_velocity_commands>true</use_velocity_commands>`, seguido de los campos `cmd_max` y `cmd_min` al plugin de las articulaciones para que la velocidad no fuera demasiado brusca para el movimiento y la articulación se moviera correctamente a la posición deseada al comienzo de la simulación. Se adjunta a continuación un ejemplo de estos cambios (Extracto de código 4.6), y un vídeo[4] del resultado final de este cambio.

```
<plugin filename="ignition-gazebo-joint-position-controller-system" name=
    "ignition::gazebo::systems::JointPositionController">
     [...]
     <use_velocity_commands>true</use_velocity_commands>
     <cmd_max>2.0</cmd_max>
     <cmd_min>-2.0</cmd_min>
     [...]
     <initial_position>1.39626</initial_position>
</plugin>
```

Extracto de código 4.6: Ejemplo de configuración de posición inicial de una articulación

Por tanto, para lograr esta estabilidad estática, el robot debe comenzar con las siguientes posiciones:

- Posición inicial del robot: x=0 y=0 z=0.5 rx=0 ry=0.240818894505798 rx=0
- /HeadPitch/cmd_pos: 0
- /HeadYaw/cmd_pos: 0
- /LAnklePitch/cmd_pos: -0.479
- /LAnkleRoll/cmd_pos: 0
- /LElbowRoll/cmd_pos: -1.0472
- /LElbowYaw/cmd_pos: -1.39626
- /LHipPitch/cmd_pos: -0.179
- /LHipRoll/cmd_pos: 0

[4] https://youtu.be/u6OsTDhBnuk

- /LHipYawPitch/cmd_pos: 0
- /LKneePitch/cmd_pos: 0.698132
- /LShoulderPitch/cmd_pos: 1.39626
- /LShoulderRoll/cmd_pos: 0.198132
- /LWristYaw/cmd_pos: -0.192
- /RAnklePitch/cmd_pos: -0.479
- /RAnkleRoll/cmd_pos: 0
- /RElbowRoll/cmd_pos: 1.0472
- /RElbowYaw/cmd_pos: 1.39626
- /RHipPitch/cmd_pos: -0.179
- /RHipRoll/cmd_pos: 0
- /RHipYawPitch/cmd_pos: 0
- /RKneePitch/cmd_pos: 0.698132
- /RShoulderPitch/cmd_pos: 1.39626
- /RShoulderRoll/cmd_pos: -0.198132
- /RWristYaw/cmd_pos: 0.192

También fue necesario editar el peso de ambos pies. Esto porque el robot tenía más peso en el tren superior que en el inferior y esto le impedía levantarse, así que se sumó 1 kilogramo de peso a cada pie y se modificaron sus matrices de inercia para adaptarlos correctamente a su nueva masa.

4.1.4. Retoque estético

Cómo último preparativo del modelo, se editó su textura para que fuera de color verde, así se lograría dar un poco de identidad al modelo y, de cara a la aplicación final en un invernadero, que fuera más vistoso.

El resultado de este retoque se puede ver en la Figura 4.5.

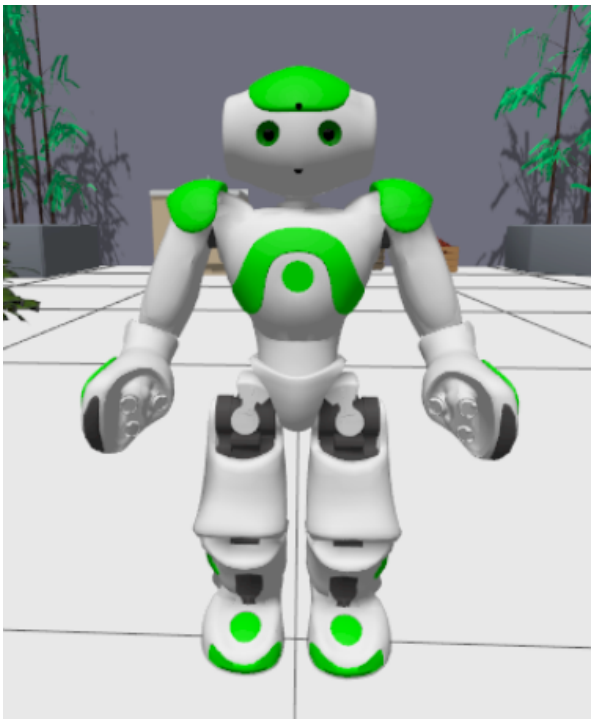


Figura 4.5: Nao retocado estéticamente

Una vez terminadas estas modificaciones, el modelo estaba completamente preparado, ya era posible su programación, y por ende el desarrollo del resto del proyecto como tal.

4.2. Editor gráfico e intérprete de movimientos

El primer elemento clave de este proyecto es el editor de movimientos, una herramienta capaz de crear patrones fijos de movimiento para coordinar las articulaciones de NAO de forma cómoda.

Una vez creados estos patrones, para materializarlos en movimiento, es necesario crear un intérprete que permita a NAO replicar dichos patrones, también de forma cómoda. Es por esto que editor e intérprete van de la mano y se describen a continuación. También se adjunta en la Figura 4.6 un esquema para ilustrar de forma adecuada este concepto.

Figura 4.6: Esquema del funcionamiento del editor y el intérprete

4.2.1. Editor gráfico

Este editor consiste en un programa que permite al usuario crear patrones de movimiento para que posteriormente sean procesados por el intérprete y a su vez replicados por el robot. Está basado en *KME* ([7],[8]), un editor de movimientos basado en fotogramas clave (*keyframes*). Permite al usuario crear secuencias de posiciones para cada articulación del robot en momentos específicos del tiempo. Es especialmente útil para programar movimientos complejos y que requieren de la coordinación de varios actuadores simultáneamente, como saludar o tomar una pose concreta, siendo estos patrones siempre fijos y no modificables a la hora de interpretarse.

El programa se ha desarrollado en Pyhton, y utiliza el simulador Pybullet[5] para poder ofrecer al usuario una visión directa del NAO y sus movimientos a la hora de crear el patrón.

Para que todo funcionase correctamente en Pybullet, era necesario tener disponible al modelo 3D de NAO, para cumplir la parte de la visualización en tiempo real de los movimientos.

Sin embargo, el formato SDF del modelo que se preparó no es compatible con Pybullet, siendo el formato requerido URDF. Por suerte, se encontró un modelo de NAO en URDF disponible para descargar en un repositorio público de github[6]. También fue necesario modificar este URDF para que fuera lo más igual posible al modelo en SDF, esto es, ponerle las posiciones iniciales de las articulaciones (cosa que se hace directamente en el código del editor), editar los pesos de los pies para que fueran iguales a los del SDF, ajustar los límites de las articulaciones, etc.

[5] https://Pybullet.org/wordpress/index.php/forum-2/

[6] https://github.com/ros-naoqi/nao_robot/blob/master/nao_description/urdf/naoV40_generated_urdf/nao.urdf

La aplicación del editor gráfico es capaz de conectarse a los *joints* de NAO gracias a Pybullet, que dispone de una API específica para hacerlo, por lo que es necesario tener en cuenta todos los *topics* (de las articulaciones) que se iban a manejar y qué movimientos y grados de libertad tenemos disponibles. Para eso es muy útil el esquema mostrado en la Figura 4.7:

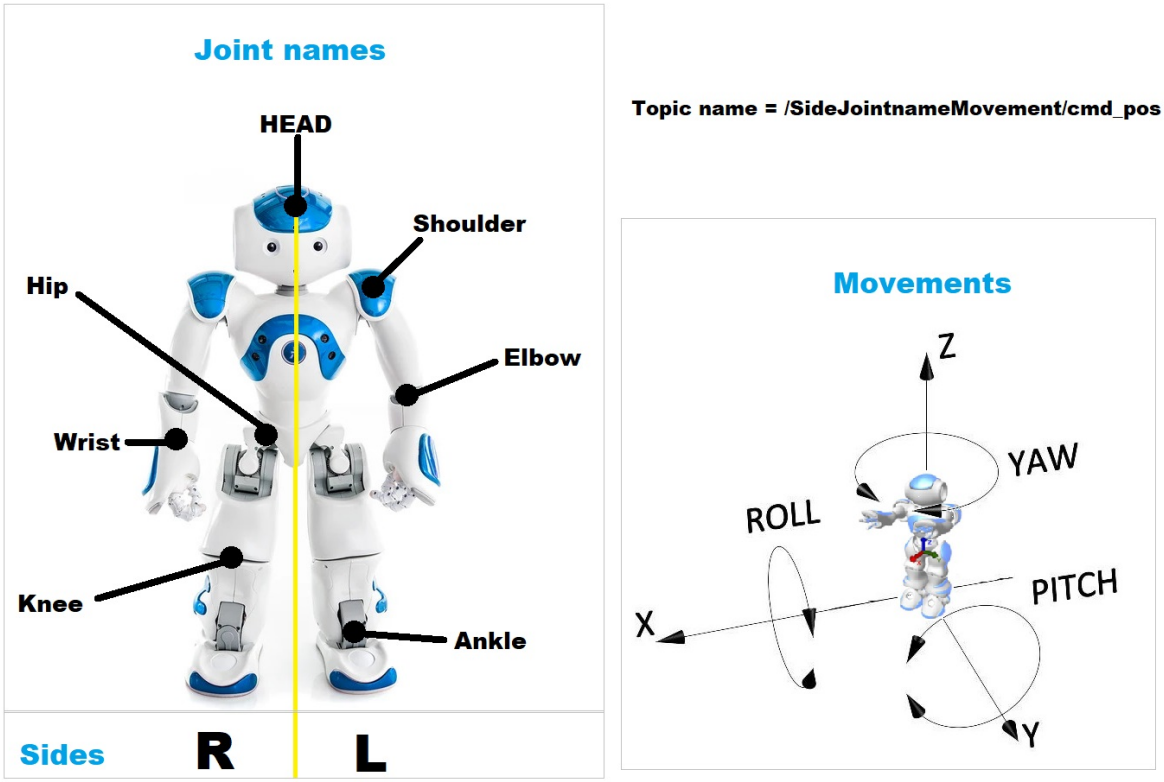


Figura 4.7: Esquema de *topics* y articulaciones de NAO

Cómo se puede ver, este robot ofrece mucha variedad de movimientos, con un total de 24 grados de libertad.

Para controlar cada uno de ellos en Pybullet de forma cómoda se ofrece una interfaz gráfica con barras deslizantes o *sliders* para que el manejo sea más visual y fácil de abarcar, esto es porque las articulaciones tienen un límite y no todas las posiciones son válidas. Con estas barras el usuario puede ver fácilmente dichos límites.

Este sistema de *sliders* lo ofrece Pybullet de forma cómoda, simplemente escribiendo lo mostrado en el Extracto de código 4.7 en el código del editor:

```

# Preparar el slider
```

```
joint_slider = p.addUserDebugParameter("nombre del joint", minimo, maximo
    , posicion inicial)

# Dentro del bucle principal (omitido ahora) Lectura del slider
joint_value = p.readUserDebugParameter(joint_slider)

# Ejecucion del movimiento en el modelo
p.setJointMotorControl2(model,1, p.POSITION_CONTROL, targetPosition=
    joint_value, maxVelocity=2)
```

Extracto de código 4.7: Ejemplo de adicición de sliders al editor

Con esto para cada una de las articulaciones, el usuario es capaz de controlar íntegramente a NAO, aún sin posibilidad de guardar los movimientos deseados.

Para esto se han utilizado ficheros JSON, de modo que se han añadido 2 *sliders* adicionales, uno para indicar el tiempo en el que se desea adoptar esa posición (tiempo del fotograma), y otro más que indica al programa que se quiere guardar el patrón, el cual, cuando cambia de posición, guarda automáticamente el fichero y avisa al usuario.

Un ejemplo del formato que tienen estos ficheros JSON se muestra en el Extracto de código 4.8.

```
[
    {
        "tiempo": 1, # marca de tiempo
        "articulaciones": [
            {
                "articulacion": "HeadYaw",
                "posicion": 2.0901734911206137e-08
            },
            {
                "articulacion": "HeadPitch",
                "posicion": 1.8516262613600133e-08
            },
            {
                "articulacion": "LHipYawPitch",
                "posicion": -1.8007272938374389e-09
            },
            {
```

```
                "articulacion": "LHipRoll",
                "posicion": -6.113291356754397e-09
            },
            [...] # resto de articulaciones
        ]
    }
]
```

Extracto de código 4.8: Ejemplo de JSON

Se adjunta un enlace a un vídeo[7] demostrativo del funcionamiento de este programa. Cabe destacar que se deja caer al robot al inicio del programa para que el usuario vea que efectivamente hay movimientos y físicas realistas involucrados en él. Se muestra una imagen de este editor en la Figura 4.8.

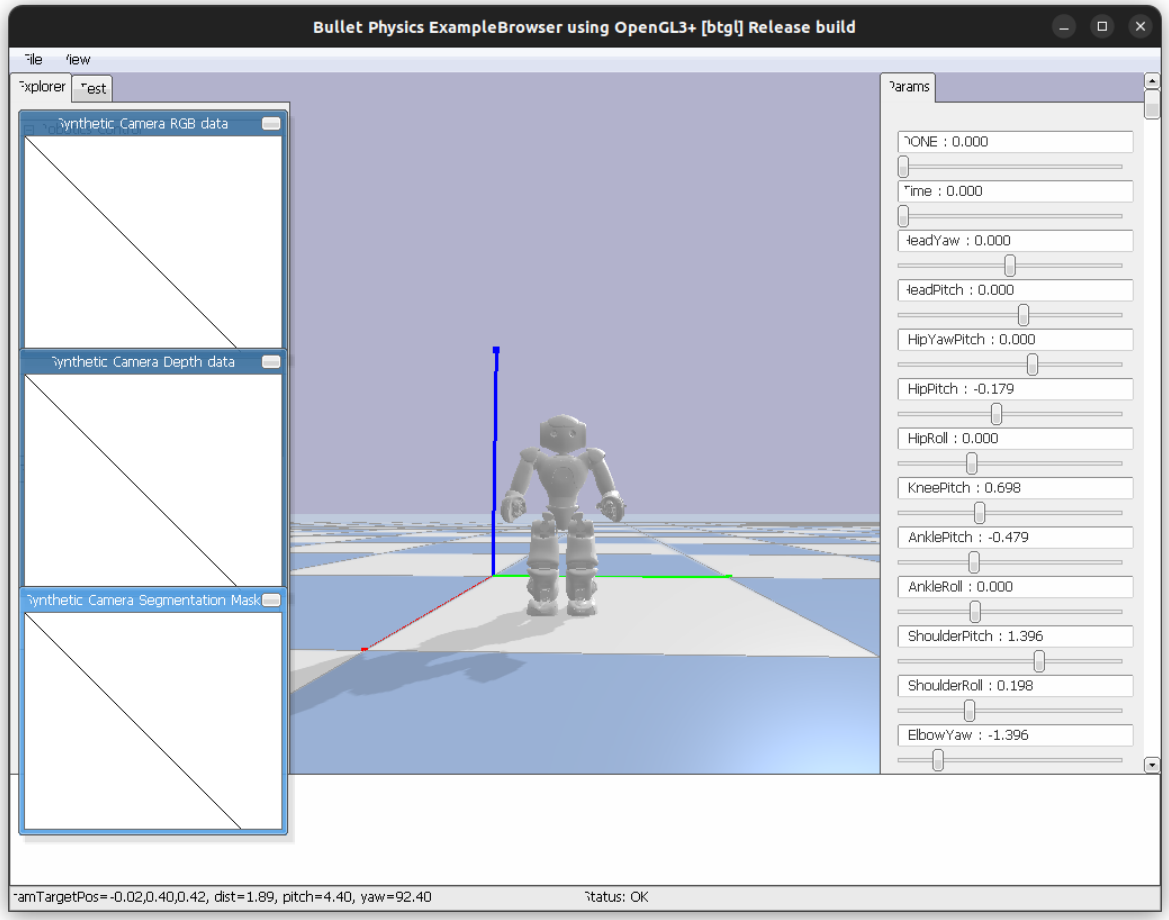


Figura 4.8: Editor gráfico de movimientos

[7] https://youtu.be/jkSdVGjrn4s

4.2.2. Intérprete de movimientos

El editor por si solo no resulta muy útil sin que el robot NAO simulado replique los patrones de movimiento creados. Es por eso por lo que existe el intérprete de movimientos. Este programa consiste en un nodo ROS2 programado en Python que lee los datos del fichero JSON que se le indica por argumento, y se encarga de hacer las publicaciones a los *topics* necesarios en el momento adecuado (replica cada fotograma en la marca de tiempo especificada).

También el usuario podría querer utilizar otros formatos para ficheros que almacenan los movimientos concretos, como el .motion, ofrecido por el simulador Webots[8]. Además, muchos patrones de movimiento ofrecidos en esta aplicación se han conseguido a partir de los ficheros de este simulador, cosa que se explicará con detalle en la seguiente sección. Como Python no ofrece soporte para leer ese tipo de ficheros directamente, se optó por adpatarlos a formato CSV y después añadir también al nodo intérprete la posibilidad de leer este tipo de ficheros.

Para hacer este nodo, se ha utilizado una calidad de servicio como la que se muestra en el Extracto de código 4.9 para evitar la fatiga a la hora de publicar varios mensajes, ya que estaremos casi constantemente publicando mensajes en cada articulación.

```
qos_profile = QoSProfile(
            reliability=ReliabilityPolicy.RELIABLE,
            history=HistoryPolicy.KEEP_ALL,
            depth=100
)
```

Extracto de código 4.9: Calidad de servicio utilizada para publicación de mensajes

A continuación, se adjunta un vídeo[9] demostrativo de este intérprete de movimientos en funcionamiento para ejecutar movimientos.

Este intérprete se creó inicialmente como un programa aislado, un nodo ROS independiente, sin embargo, se refactorizó como una función dentro de la librería que se ha desarrollado, de modo que cualquier aplicación con el humanoide pueda utilizarlo para materializar la locomoción y los movimientos del robot.

[8] https://cyberbotics.com/
[9] https://youtu.be/sLobRujULKE

4.3. Librería de movimientos

En cuanto a la librería desarrollada, cuyo nombre es *CoordmovesLib.py*, se encarga de que el uso de ROS2 sea parcialmente transparente para el usuario, ya que su instalación y uso básico (creación de un paquete para utilizarla, compilarlo, etc) sigue siendo necesario.

Para lograr este encapsulamiento, la biblioteca debe incluir los nodos necesarios (clases de Python responsables de publicar las posiciones de las articulaciones) y una función que los active, la cual será invocada por el usuario.

La activación de estos nodos requiere el uso de rclpy[10], que proporciona la API oficial de Python para interactuar con ROS 2.

Un ejemplo de cómo han de crearse estas clases para que los nodos funcionen correctamente se muestra en el Extracto de código 4.10.

```
def Interpreter(file_name: str, printable=True):
    rclpy.init()
    node = Interpreter_class(file_name, printable)

    try:
        rclpy.spin_once(node, timeout_sec=2)

    finally:
      node.destroy_node()
      rclpy.shutdown()
```

Extracto de código 4.10: Ejemplo de función que invoca un nodo ROS2

Hay 3 tipos de estas funciones, las que sirven para llamar a métodos cuya tarea es replicar patrones fijos creados con el intérprete; las que sirven para invocar a métodos capaces de replicar movimientos parametrizables y las que sirven para leer los sensores.

4.3.1. Patrones fijos

El nombre de la función para movimientos genéricos es *Interpreter* (mostrada en la Extracto de código 4.10) y recibe como argumento el nombre del fichero a replicar

[10] https://docs.ros.org/en/rolling/p/rclpy/

como *string* y el parámetro opcional booleano *printable*, que indica si es necesario o no imprimir un *log* final. Este parámetro es utilizado por todos los nodos y funciones que se llaman desde otras funciones, para que cada una tenga sus *logs* por separado.

Un ejemplo de esto sería una función que llame a este intérprete, llamada hipotéticamente *función*, la cual tiene un *log* final como este: `[Función]: Mensaje final`. Sin el parámetro *printable*, se imprimirían el *log* final de la función del intérprete (`[Intepreter]: Movimientos de fichero fichero.json completados`) y después el de la función anteriormente mencionada, lo que podría llevar a confusión al usuario por ver nombres de funciones que puede no haber llamado explícitamente.

Si este parámetro *printable* se pasa como verdadero, el *log* se imprimirá, lo contrario ocurriría si el parámetro toma el valor de falso. Si no se le pasa (al ser este un parámetro opcional), su valor predeterminado es *True*.

Esta función *Interpreter* se encarga de leer el fichero epecificado, crear un publicador para cada aticulación que aparezca en el fichero y publicar en el tiempo especificado la posición indicada para cada articulación, consiguiendo así una secuencia estable de fotogramas. Su funcionamiento es el mismo que el explicado en la sección 4.2.2, incluida la calidad de servicio.

También a este tipo de funciones pertenecen las siguientes, que ya incorporan movimientos concretos para el humanoide habituales y útiles en aplicaciones robóticas:

- *wakeup_face_down*: Esta función no recibe parámetros y se encarga de llamar al intérprete para ejecutar el patrón de levantar al robot del suelo en caso de haber caído boca abajo o de *cubito prono*. Este patrón fue recogido del simulador Webots. Se demuestra su correcto funcionamiento en el siguiente vídeo[11]
- *wakeup_face_up*: Esta función no recibe parámetros y se encarga de llamar al intérprete para ejecutar el patrón de levantar al robot del suelo en caso de haber caído boca arriba o de *cubito supino*. Este patrón consiste en que nao se de la vuelta (patrón creado con el editor en formato JSON), y después ejecutar el patrón que lo levanta desde *cubito prono*. Se adjunta un vídeo[12] demostrativo.
- *stand_still*: Esta función recibe el parámetro opcional *printable* y sirve para que NAO adopte la postura de inicio o reposo predeterminada, explicada en la

[11] https://youtu.be/e-WkWSMkjDw
[12] https://youtu.be/GxNZoqB1smw

sección 4.1. Lo hace llamando al intérprete con el fichero que indica esta posición, creado con el editor en fomato JSON. En este enlace[13] puede verse un vídeo demostrativo.

- *say_hi*: Esta función toma como parámetro la *string hand*, que puede tomar los valores L, left, LEFT, R, right o RIGHT, que sirve para indicar a NAO con qué mano debe saludar llamando al intérprete con el fichero adecuado. Los patrones para ambas manos fueron creados con el editor en formato JSON. A continuación se adjunta una demostración de su funcionamiento mediante un vídeo[14].
- *turn*: Esta función toma 3 parámetros: el booleano *printable*, una *string* igual que la de la función *say_hi*, para indicar el lado a girar, y un entero, que sirve para indicar los grados a girar (40, 60 o 180, siendo este último válido solo para giros a la izquierda). Aunque podría parecer una función parametrizable, en realidad se limita a seleccionar entre patrones fijos de giro predefinidos recogidos de Webots según los valores de los parámetros. Se ha optado por encapsular esta lógica de esta forma para facilitar su uso. Su funcionamiento puede apreciarse en el siguiente vídeo[15].
- *grab_box*: Esta función no recibe parámetros y se encarga de que el intérprete lance el patrón necesario para coger una caja, creado con el editor en formato JSON. Esta función es útil para la aplicación de ejemplo realizada en este TFG. En el siguiente vídeo[16] puede verse su funcionamiento.
- *release_box*: Esta función no recibe parámetros y es análoga a la anterior, ya que es igual, pero el patrón es para dejar la caja recogida. Patrón también creado con el editor e formato JSON. A continuación[17] puede verse el vídeo demostrativo de esta función.

4.3.2. Patrones parametrizables

Estas funciones son más complejas que las anteriores, debido a que cada una de ellas debe ser capaz de editar en tiempo de ejecución el patrón requerido en función del parámetro facilitado.

[13] https://youtu.be/0Rx5PmfkZaE
[14] https://youtu.be/dkeeOZ4hFeU
[15] https://youtu.be/D2QoRB7XO-U
[16] https://youtu.be/N4m_QlzfNNg
[17] https://youtu.be/XMkKKl0hzyk

En el caso de este TFG, y porque para los modos de caminar es necesario (sin ellos no se podría llevar a cabo ninguna aplicación útil para NAO), se ha optado por parametrizar la velocidad de los movimientos, en este caso los pasos.

Todas las funciones de este tipo se encargan de modos de caminar para el humanoide, y hacen que la caminata sea más rápida o más lenta.

La manera de poder parametrizar este valor de la velocidad es modificando los tiempos del fotograma, de esta forma, si el tiempo aumenta, el movimiento será más lento, si disminuye, será más veloz.

Para que este parámetro tuviera una interfaz sencilla, se ha seguido el esquema mostrado en la Figura 4.9 a la hora de parametrizarlo.

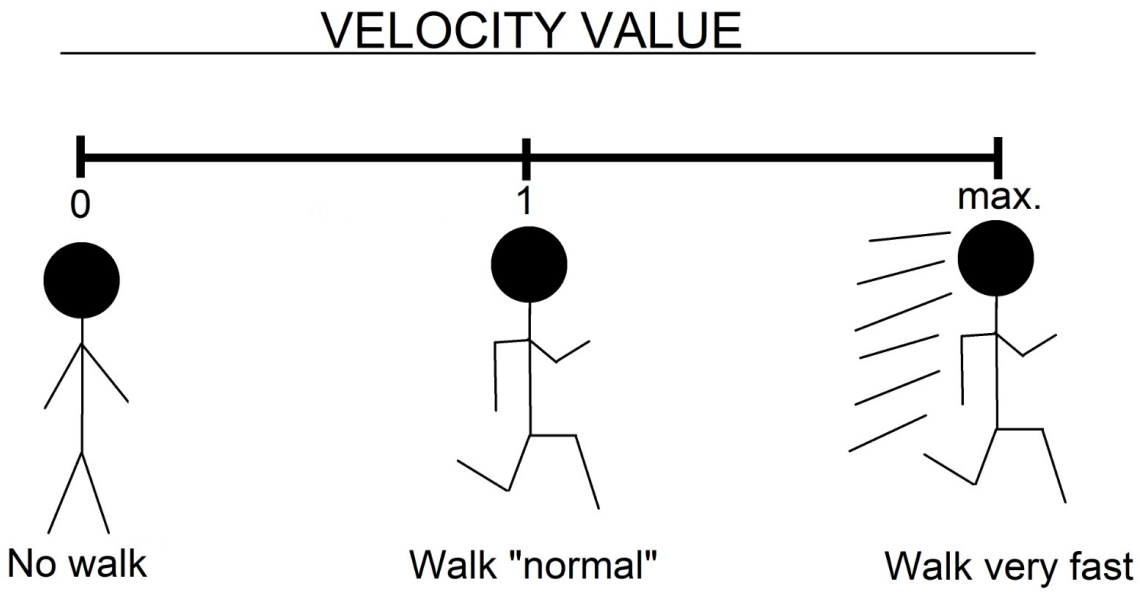


Figura 4.9: Esquema de parametrización de la velocidad

Para conseguir este efecto, ha sido necesario dividir el tiempo de cada fotograma entre el parámetro de velocidad introducido, de esta manera, se consigue que cuando el parámetro sea menor que 1, el tiempo aumente, y lo contrario en caso de que el parámetro sea mayor que 1. Así también nos aseguramos de que el parámetro quede "predeterminado" si le pasamos velocidad igual a 1.

Sin embargo, se debe tener precaución a la hora de pasar este parámetro, ya que una velocidad demasiado baja hará que los fotogramas no vayan con la fluidez mínima requerida y el robot se mueva de manera poco realista, dejando demasiado espacio temporal entre un fotograma y el siguiente. De forma análoga, una

velocidad demasiado alta provocará inestabilidad absoluta en la caminata, y los fotogramas siguientes "atropellarán" a los anteriores, haciendo que el robot caiga.

Para evitar que el usuario utilice valores inadecuados, cada una de estas funciones cuenta con un valor mínimo y uno máximo permitidos. Estos límites se especificarán en la explicación correspondiente de cada función, presentada más adelante.

Esta velocidad puede ser lineal, angular o lateral y tomar valores positivos y negativos, siguiendo estas normas:

- *Velocidad positiva*: Implica movimientos hacia adelante o a la derecha, dependiendo de si la velocidad es lineal o angular/lateral, respectivamente.
- *Velocidad negativa*: Implica movimientos hacia atrás o la izquierda, dependiendo de si la velocidad es lineal o angular/lateral, respectivamente.

Este efecto se logra gracias a que hay patrones para ambas direcciones, y dependiendo del caso se utiliza uno u otro.

Cabe destacar que no sólo se ha parametrizado la velocidad, también se ha parametrizado el número de pasos que el robot puede dar. Este mínimo viene dado por el patrón predeterminado en formato CSV y puede ser 10 o 2, dependiendo del caso. Para parametrizarlo, simplemente se repite en un bucle *for* el patrón `pasos_indicados/mínimo` veces, por lo que el parámetro de los pasos también debe ser múltiplo del mínimo. Dicho valor mínimo también se detallará a continuación.

Cada una de estas funciones son como la mostrada en la Extracto de código 4.10, debido a que estos patrones se logran mediante nodos ROS2 que deben ser invocados.

Estas funciones son las siguientes:

- *setL*: Esta función tiene como objetivo que NAO se desplace lateralmente (utilizando velocidad lateral). Los patrones utilizados se han recogido de webots y tiene un mínimo de 2 pasos, con velocidad mínima de -0.35 ó 0.35, dependiendo de la dirección a la que se quiera avanzar y una máxima de 4.35 ó -4.35. A continuación se adjunta un enlace[18] a su vídeo demostrativo.
- *setV*: Esta función se encarga de hacer al robot caminar en linea recta (utilizando velocidad lineal). El mínimo de pasos en este caso es algo especial, ya que, si se le pasa velocidad negativa, el patrón sólo tiene 2 pasos, por lo que en este caso, el bucle se repite *5*pasos_indicados*, para que el número de

[18] https://youtu.be/I7xlrR8_ZJk

pasos aplique a ambos patrones, ya que el patrón de caminata hacia adelante tiene un mínimo de 10 y así ambos lo comparten. Ambos fueron recogidos de webots y los máximos y mínimos de velocidad son los mismos que para la función setL. Se puede apreciar su funcionamiento en este enlace[19].

- *turnVel*: Esta función sirve para que el robot sea capaz de girar en el sitio con velocidad angular parametrizada. Ambos patrones utilizados son recogidos de webots, tienen un mínimo de 2 pasos y unas velocidades de 0.35 y -0.35 para el mínimo, y 1.9 y -1.9 para el máximo. Su vídeo demostrativo se encuentra en el siguiente enlace[20].
- *setW*: Esta función se encarga de mover a NAO describiendo una trayectoria de arco, a una velocidad lineal (V) fija por defecto, sus parámetros son los mismos que la función anterior. Los patrones utilizados fueron creados a base de editar el patrón CSV de la caminata hacia adelante. Puede apreciarse su funcionamiento en el siguiente vídeo[21].
- *setNW*: Esta función es igual que la anterior, pero los arcos descritos son hacia atrás. Los patrones utilizados fueron creados a base de editar el patrón CSV de la caminata hacia atrás. No se adjunta vídeo de esta función por ser tan parecida a la anterior.

Pero, estas funciones por sí solas no representan una forma intuitiva de hacer que el robot camine, por lo que es necesaria una sexta función que encapsule todas estas (excepto la de setL, ya que esa es un modo de caminar más específico y se deja fuera) en una sola con una interfaz amigable para el usuario. Esta es la función *setArc*, que recibe todos los parámetros necesarios para cada función explicada anteriormente (número de pasos, velocidad V o W, y el parámetro opcional *printable*) para después llamarlas en función de los parámetros recibidos.

Para facilitar la comprensión de esta función, se incluye un esquema que muestra su impacto en la trayectoria del NAO en la Figura 4.10.

[19] https://youtu.be/bpGAtWTDSik
[20] https://youtu.be/NH_6tGjlq7U
[21] https://youtu.be/X8-WW8bdEck

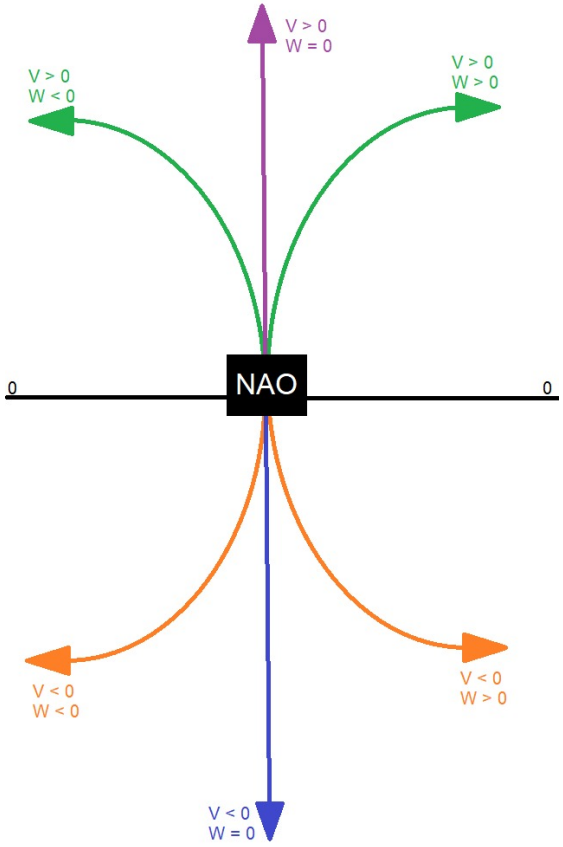


Figura 4.10: Esquema de funcionamiento de la función *setArc*

El código de esta función tan importante y sencilla al mismo tiempo se encuentra en el Extracto de código 4.11

```
1 def setArc(v,w,steps = 10):
2     if (not ((0.35 <= abs(w) <= 1.9) or abs(w) == 0) or not (2 <= steps)
    or (steps%2 != 0)) or (not ((0.35 <= abs(v) <= 4.35) or abs(v) == 0)
    or not (10 <= steps) or (steps%10 != 0)):
3         print("[setArc]: ERROR: La velocidad lineal debe estar entre
    +-0.35 y +-4.35 y la angular entre +-0.35 y +-1.9, y los pasos deben
    ser multiplos de 10")
4         sys.exit(1)
5
6     if v != 0 and w == 0:
7         setV(v, steps, False)
8
9     elif v != 0 and  w != 0:
```

```
10          if v > 0:
11              setW(w, steps, False)
12          else:
13              setNW(w, steps, False)
14
15      elif v == 0 and w != 0:
16          turnVel(w, steps, False)
17
18      elif v == 0 and w == 0:
19          stand_still(False)
20
21      else:
22          print("[setArc] ERROR: Patron de movimiento no valido")
23          sys.exit(1)
24
25      print("[setArc]: Pasos completados")
```

Extracto de código 4.11: Función setArc

4.3.3. Funciones que utilizan sensores

También se han incluido en la biblioteca funciones cuya misión es leer los datos sensoriales y brindar información al respecto. Este es el caso de las funciones *Read_IMU* y *get_face*.

La primera de las funciones se encarga de devolver la aceleración en z que está experimentando el robot, para que después *get_face* la recoja y nos indique, en el caso de caída, si NAO está de *cubito supino* (boca arriba), *cubito prono* (boca abajo) o en una posición indeterminada.

Esto sirve para saber por ejemplo a qué patrón fijo de levantarse debemos llamar.

En el caso de la cámara, no se ha implementado una función concreta para utlizarla, se hablará de esto en el Capítulo 5.

En el siguiente enlace[22] se adjunta un vídeo demostrativo de estas funciones, aunque solo se ve *get_face*, porque usa *Read_IMU* para funcionar.

[22] https://youtu.be/xJUYh2QJNuk

4.4. Aplicación de ejemplo con el humanoide

Esta aplicación, además de utilizar al NAO como robot de servicio, demuestra que la librería creada funciona correctamente. Esto es porque se ha desarrollado íntegramente utilizando sus funciones.

La aplicación consiste en que NAO transporte una caja hecha a medida para él, ya que su tamaño no permite otra alternativa. Como el modelo utilizado no tiene dedos, fue necesario adaptar la caja para que pudiera sostenerla de forma segura. El objetivo es moverla desde una mesa azul hasta una mesa naranja, ambas ajustadas a la escala del robot y ubicadas dentro de un invernadero. Por esta razón, la aplicación ha sido llamada *GreenNao*.

El escenario utilizado ha sido íntegramente creado dentro de este Trabajo Fin de Grado, con ayuda de *assets opensource* de Blender[23]. Este mundo también está en formato SDF ([9],[10]) y se muestra en la Figura 4.11.

[23] https://www.blender.org/

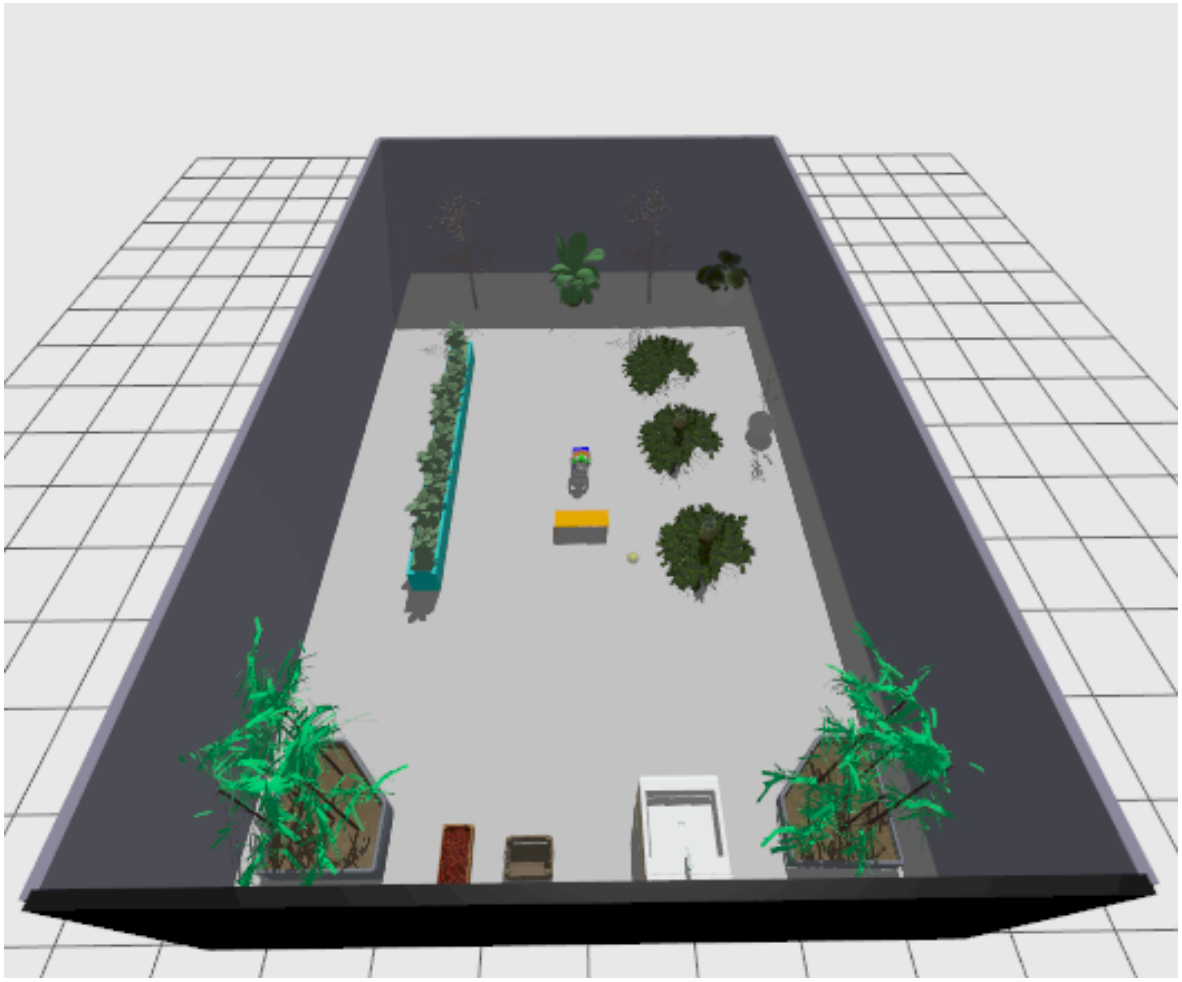


Figura 4.11: Mundo Invernadero para la aplicación de ejemplo

Para llevar a cabo esta aplicación, simplemente se ha ido llamando a las funciones necesarias de la librería `CoordMovesLib` hasta conseguir el resultado esperado.

A continuación, en el Extracto de código 4.12 se muestra el código fuente de esta aplicación para demostrar el uso de funciones de la librería y se adjunta un vídeo[24] demostrativo de la misma, además de un par de fotos en la Figura 4.12 y también en la Figura 4.13 para que se pueda apreciar mejor el montaje de esta aplicación de ejemplo.

```
1 import CoordMovesLib
2 import time
3
4 time.sleep(5)
5 CoordMovesLib.grab_box()
```

[24] https://youtu.be/obWjmRzKRCQ

```
time.sleep(3)

for i in range(3):
  CoordMovesLib.turn("L", 60)

CoordMovesLib.turn("L", 40)

CoordMovesLib.setArc(1, 0)
time.sleep(3)

CoordMovesLib.release_box()
print("Terminado")
```

Extracto de código 4.12: Aplicación GreenNao

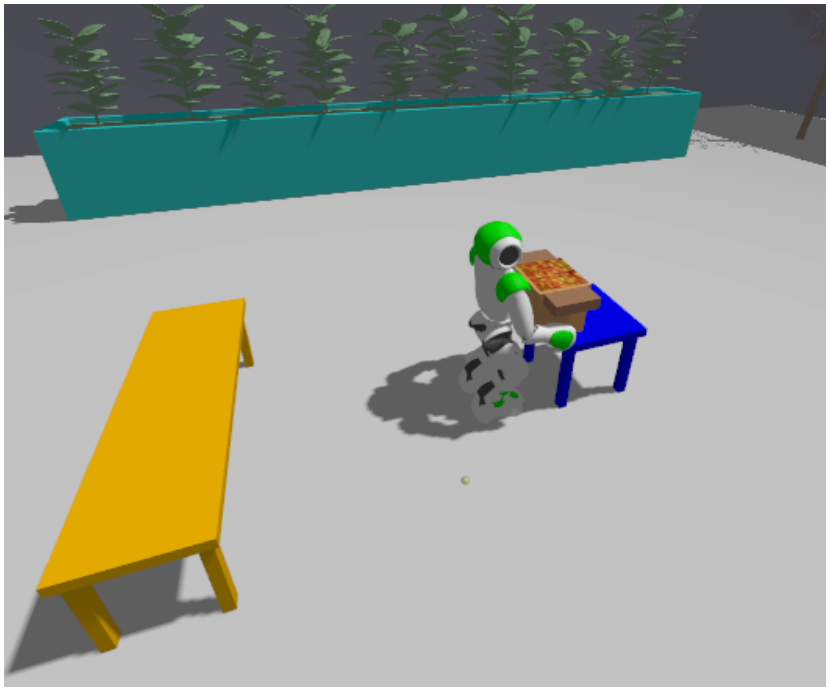


Figura 4.12: Aplicación GreenNao. Vista lateral

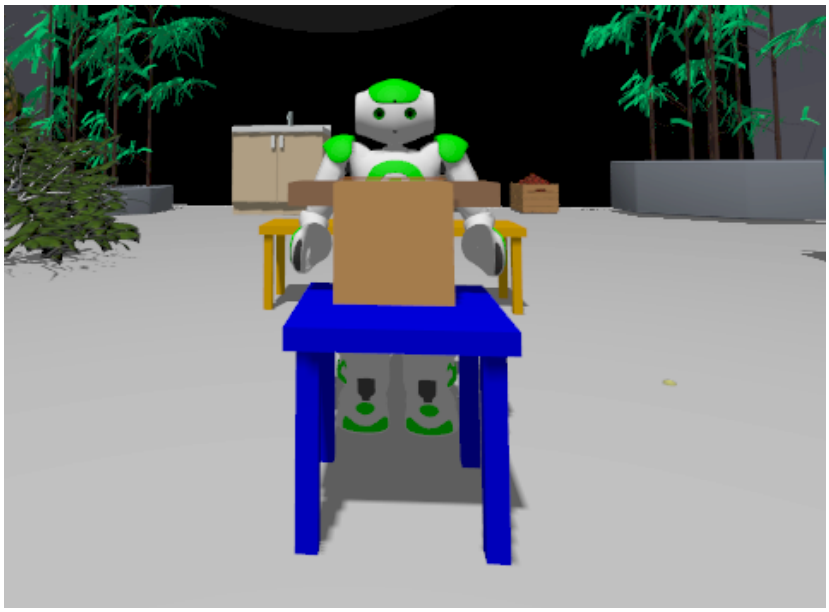


Figura 4.13: Aplicación GreenNao. Vista frontal

5. Conclusiones

Una vez visto lo que se ha hecho en este proyecto, se procede a concluir si se han cumplido o no los objetivos, y qué posibilidades de mejora tiene para el futuro. Sin olvidar las competencias adquiridas y utilizadas para su desarrollo.

5.1. Cumplimiento del objetivo

Ya visto el desarrollo completo del proyecto, se puede decir que efectivamente se ha cumplido el objetivo final, debido a que se han cumplido cada uno de los subobjetivos.

Se ha alcanzado el subobjetivo 1, mencionado en el Capítulo 2, ya que se desarrolló un editor basado en secuencias de fotogramas (explicado en el Capítulo 4). Este editor permite organizar referencias de posición para cada uno de los actuadores que participan en el movimiento coordinado, en la locomoción basado KME, como se describe en el Capítulo 4.2.1.

Además, se ha desarrollado un intérprete capaz de comunicarse con los controladores individuales de cada articulación para que las replique, como bien se ha visto en el Capítulo 4.2.2. El resultado de esto es que se puede ver que el robot es capaz de saludar, levantarse, coger la caja, etc.

El subobjetivo 2, también explicado en el Capítulo 2, se ha visto cumplido en el Capítulo 4, mediante las clases de la librería que permiten la entrada de parámetros y modifican los tiempos de ejecución de los movimientos, y mediante las funciones que ejecutan patrones fijos de movimiento. Esto se ve claramente en las secuencias de caminata recta de 10 pasos, de caminata lateral, la caminata en arco, las funciones de coger y dejar la caja, los saludos, las formas de levantarse, etc. Estas secuencias dotan al robot de una primera capa de locomoción que permite programar sus movimientos(ya sean fijos o parametrizables) en aplicaciones robóticas de modo razonablemente simple, sin que el desarrollador tenga que programar explícitamente la coordinación de todos los actuadores individuales involucrados.Todo esto gracias a la librería desarrollada para ello.

Y, por último, también se ha cumplido el subobjetivo 3, también explicado en el Capítulo 2, porque, como se ha visto en el Capítulo 4.4, la aplicación ha quedado

resuelta, demostrándonos que NAO cumple con los requisitos requeridos: La manipulación de la caja a la hora de recogerla y dejarla en su lugar de destino y el hecho de ser capaz de caminar con ella en brazos, todo ello utilizando la librería *CoordmovesLib* desarrollada.

5.2. Competencias empleadas

Estos objetivos se han cumplido gracias a los conocimientos adquiridos a la hora de cursar las siguientes asignaturas de mi grado:

- *Modelado y Simulación de robots*: En esta asignatura se trata cómo funciona una simulación robótica, en términos generales, utilizando ROS2 y Gazebo.
- *Arquitecturas Software para robots*: Asignatura dedicada a enseñarnos a utilizar ROS2 y programar con este middleware.
- *Laboratorio de sistemas*: En esta asignatura aprendimos a manejar el sistema operativo Ubuntu, utilizado en este TFG. Además de enseñarnos a utilizar la herramienta *git* para el manejo de repositorios remotos de Github.
- *Robótica de servicios*: En esta asignatura nos enseñaron los fundamentos de los robots de servicio y su forma de programarlos.
- *Fundamentos de la programación*: En esta asignatura aprendimos a manejar el lenguaje Python.

5.3. Competencias adquiridas

Este proyecto ha hecho que adquiera diferentes competencias, útiles para diferentes campos de la robótica en general.

La primera de ellas ha sido la capacidad de encapsular ROS2 mediante una librería, cosa que es útil para desarrollar capas de control de cualquier tipo para cualquier sistema que utilice este middleware. Este conocimiento no entra en ninguna asignatura vista en el grado.

La segunda ha sido la capacidad de desarrollar un mundo completo para Gazebo, ya que, aunque como se mencionó en la sección anterior que se utilizaron los conocimientos aduiridos en la asignatura *Modelado y simuación de robots*, el diseño del mundo no entraba en el itinerario de la asignatura. Así como el uso

íntegro de Gazebo Harmonic (los puentes con ROS2, los plugins necesarios para adaptar el modelo, etc), debido a que en esta asignatura se trabajó con una versión anterior.

También se ha adquirido la capacidad de tratar con ficheros de formato JSON, porque no había tratado con ellos anteriormente.

5.4. Trabajos futuros

Como trabajos futuros, tenemos un abanico bastante amplio de posibilidades.

En primer lugar, llevar este proyecto a un robot real. Cabe destacar que el trabajo en simulación también es válido y potente, sin embargo, no deja de ser algo que no existe del todo en la realidad, por lo que sería muy interesante hacer el paso conocido cómo *sim 2 real*, para que un NAO real disponga de las funcionalidades que la librería de locomoción desarrollada ofrece.

Un segundo trabajo futuro, es hacer los modos de caminar más robustos y estables, ya que, como se ha visto en el Capítulo 4, hay algunos movimientos que no están del todo conseguidos (como es el caso de los arcos hacia atrás) o son algo inestables (como los demás modos de caminar). También se podría investigar para reducir el número de pasos mínimo. También sería interesante introducir funciones para la lectura de la cámara, como se mencionó en la sección 4.3.3

Un tercer posible trabajo futuro sería construir más aplicaciones de robótica de servicios con el robot NAO, creando más escenarios y desafíos para el robot. Esto sería interesante en el ámbito educativo, por ejemplo, para proponer prácticas a los estudiantes y que deban resolverlas utilizando la librería, siendo este abanico de posibilidades casi infinito.

Bibliografía

[1] Softbank Robotics. Nao: Personal robot teaching assistant, 2025. URL https://us.softbankrobotics.com/nao.

[2] TFG: Francisco Miguel Rivas Montero. Grado en Ingeniería Técnica en Informática de Sistemas. Caminata basada en ondas acopladas para el robot humanoide nao, 2010. URL https://gsyc.urjc.es/jmplaza/students/pfc-caminata_nao-2010.pdf.

[3] TFG: Francisco Pérez Salgado. Ingeniería Informática Superior. Caminatas basadas en ondas acopladas para el humanoide nao en gazebo-5 y jderobot, 2015. URL https://gsyc.urjc.es/jmplaza/students/pfc-nao_gazebo-fperez-2015.pdf.

[4] Open Robotics. Creating a package, 2025. URL https://docs.ros.org/en/foxy/Tutorials/Beginner-Client-Libraries/Creating-Your-First-ROS2-Package.html.

[5] Open Robotics. Writing a simple publisher and subscriber (python), 2022. URL https://docs.ros.org/en/humble/Tutorials/Beginner-Client-Libraries/Writing-A-Simple-Py-Publisher-And-Subscriber.html.

[6] The Construct. Gazebo sim with ros2 — ros2 developers open class #177, 2024. URL https://www.youtube.com/watch?v=DsjJtC8QTQY.

[7] Michail G. Lagoudakis Georgios Pierris. An interactive tool for designing complex robot motion patterns, 2009. URL https://www.pierris.gr/me/downloads/kme.pdf.

[8] Syamimi Shamsuddin, Luthffi Idzhar Ismail, Hanafiah Yussof Nur, and Ismarrubie Zahari. Humanoid robot nao: Review of control and motion exploration, november 2021. URL https://www.researchgate.net/publication/254029603_Humanoid_robot_NAO_Review_of_control_and_motion_exploration.

[9] Articulated Robotics. Gazebo de customworld, february 2025. URL https://www.youtube.com/watch?v=K4rHglJW7Hg.

[10] Open Source Robotics. Physic parameters, 2014. URL https://classic.gazebosim.org/tutorials?tut=physics_params.