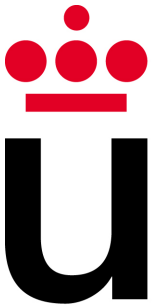


Universidad
Rey Juan Carlos

GRADO EN INGENERÍA DE ROBÓTICA SOFTWARE

Escuela de Ingeniería de Fuenlabrada

Curso Académico 2023/2024

Trabajo Fin de Grado

Aplicaciones de robótica de servicio en la plataforma académica Robotics Academy

Autora: Lucía Lishan Chen Huang
Tutor: Dr. Jose María Cañas Plaza

Agradecimientos

Agradecimientos a mi tutor Jose María, por la dedicación,
paciencia y todos los ánimos recibidos.

Agradecimientos a mis amigos, por nuestro encuentro y
acompañamiento durante todo el camino.

Agradecimientos a mi familia y personas cercanas, por
todo los apoyos recibidos.

Resumen

En la última década, impulsada por avances tecnológicos, la robótica ha experimentado un crecimiento exponencial. Los robots ya no están confinados en entornos industrial, sino que se están integrando en nuestra vida diaria. Este crecimiento en el uso de robots ha generado una fuerte demanda de profesionales en este sector, haciendo que la formación de estos expertos sea un tema de gran importancia. Existen varios canales de formación, incluyendo la formación preuniversitaria, programas universitarios y plataformas web de aprendizaje autónomo.

Robotics Academy es una plataforma educativa de robótica que ofrece una amplia variedad de ejercicios abarcando temas como la robótica de servicio, la visión artificial, la conducción autónoma, etc. La plataforma pretende, a través de estos ejercicios, permitir que los usuarios aprendan y pongan en práctica conocimientos robóticos.

En este trabajo, se busca contribuir en la mejora de dicha plataforma con la actualización y creación de dos ejercicios relacionados con la robótica de servicio. El primero, llamado Sigue-Persona, consiste en programar un robot TurtleBot2 (simulado o real) para realizar la tarea de seguir a una persona, aplicando técnicas de procesamiento de imágenes y sensores. El segundo, llamado Almacén-Amazon, aborda la tarea logística de transporte de estanterías. Se ofrecen dos tipos de robots: uno con geometría cuadrática y movimiento holonómico; y otro con geometría rectangular y movimiento Ackermann. Además, se propone el uso de OMPL (*Open Motion Planning Library*) para resolver la tarea de planificación.

Para cada ejercicio, se han construido las infraestructuras (robot simulado o real y entorno simulado), se han implementado la página web, se han desarrollado plantillas Python que resuelven problemas de bajo nivel para simplificar a los usuarios la programación de la aplicación, se ha creado documentación de apoyo para los usuarios en la resolución de estos ejercicios, y se han implementado soluciones de referencia.

Acrónimos

AGV *Automated Guided Vehicle*
AMR *Autonomous Mobile Robot*
AWS *Amazon Web Services*
DAE *Digital Asset Exchange*
DDS *Data Distribution Service*
DOM *Document Object Model*
FMT *Fast Marching Trees*
FTDI *Future Technology Devices International*
GPU *Graphics Processing Unit*
GUI *Graphical User Interface*
HAL *Hardware Abstraction Layer*
HD *High Definition*
HTML5 *HyperText Markup Language 5*
HTML *HyperText Markup Language*
OMPL *Open Motion Planning Library*
POO *Programación Orientada a Objetos*
PRM *Probabilistic Roadmap*
RAM *Robotics Application Manager*
RADI *Robotics Academy Docker Image*
RFB *Remote Frame Buffer*
ROS *Robot Operating System*
RRT *Rapidly Exploring Random Tree*
SDF *Simulation Description Format*
SLAM *Simultaneous Localization and Mapping*
STEM *Science, Technology, engineering and Mathematics*

TCP *Transmission Control Protocol*
TFG *Trabajo Fin de Grado*
URDF *Unified Robot Description Format*
URI *Uniform Resource Identifier*
URL *Uniform Resource Locators*
USB *Universal Serial Bus*
VFF *Virtual Force Field*
VNC *Virtual Network Computing*
XML *Extensible Markup Language*

Índice general

1. Introducción **1**
- 1.1. Robótica de servicio . . . 1
  - 1.1.1. Robótica social . . . 3
  - 1.1.2. Robótica en logística . . . 6
- 1.2. Robótica Educativa . . . 8
  - 1.2.1. Robotics Academy . . . 10
  - 1.2.2. Unibotics . . . 11

2. Objetivos y Metodología **12**
- 2.1. Objetivos . . . 12
- 2.2. Metodología . . . 13
- 2.3. Plan de trabajo . . . 13

3. Herramientas utilizadas **15**
- 3.1. Lenguajes de programación . . . 15
  - 3.1.1. Python . . . 15
  - 3.1.2. C++ . . . 15
  - 3.1.3. JavaScript . . . 16
- 3.2. Tecnologías web . . . 16
  - 3.2.1. React . . . 17
  - 3.2.2. HTML5 . . . 17
  - 3.2.3. WebSockets . . . 17
  - 3.2.4. VNC . . . 18
- 3.3. Docker . . . 18
- 3.4. Blender . . . 18
- 3.5. Simulador Gazebo . . . 19
  - 3.5.1. Formato SDF . . . 20
- 3.6. Middleware robótico ROS2 . . . 20
- 3.7. Biblioteca de planificación OMPL . . . 21
- 3.8. Robot TurtleBot2 . . . 22
  - 3.8.1. Base Kobuki . . . 22
  - 3.8.2. Cámara Logitech Webcam Pro 9000 . . . 23

3.8.3. Láser RPLIDAR A2 . . . 23
3.9. Plataforma Robotics Academy . . . 24

4. Ejercicio Sigue-Persona en Robotics Academy 30
4.1. Enunciado . . . 30
4.2. Infraestructura . . . 30
4.2.1. Entorno simulado de un hospital . . . 31
4.2.2. Persona teleoperada . . . 31
4.2.3. Robot TurtleBot2 simulado . . . 33
4.2.4. Robot TurtleBot2 real . . . 33
4.3. Página Web . . . 35
4.4. Plantillas Python . . . 37
4.5. Solución de referencia . . . 41
4.6. Migración a la versión 5.2 de Robotics Academy . . . 45
4.6.1. Persona teleoperada . . . 45
4.6.2. Plantillas Python renovadas . . . 48

5. Ejercicio Almacén-Amazon en Robotics Academy 49
5.1. Enunciado . . . 49
5.2. Infraestructura . . . 49
5.2.1. Entorno simulado de dos almacenes . . . 50
5.2.2. Robot logístico holonómico . . . 52
5.2.3. Robot logístico Ackermann . . . 54
5.3. Página web . . . 57
5.4. Plantillas Python . . . 60
5.5. Soluciones de referencia . . . 62
5.5.1. Planificación espacial con robot holonómico y geometría “circular” . 63
5.5.1.1. Planificación de la ruta . . . 63
5.5.1.2. Pilotaje del robot . . . 67
5.5.1.3. Máquina de estados . . . 69
5.5.2. Planificación espacial con robot holonómico y geometría “rectangular” 70
5.5.2.1. Planificación de la ruta . . . 70
5.5.2.2. Pilotaje del robot . . . 72
5.5.2.3. Máquina de estados . . . 72
5.5.3. Planificación basado en controles con robot Ackermann . . . 73
5.5.3.1. Planificación de la ruta . . . 73
5.5.3.2. Pilotaje del robot . . . 77
5.5.3.3. Máquina de estados . . . 78
5.5.4. Comparativa de las tres soluciones de referencia . . . 78

6. Conclusiones **80**
6.1. Conclusiones finales 80
6.2. Competencias adquiridas 81
6.3. Líneas futuras 81

Bibliografía **82**

Índice de figuras

1.1. Robot Pepper 3
1.2. Robot Atlas 4
1.3. Robot Aibo 4
1.4. Robot Paro 5
1.5. Robot Atlas 2030 5
1.6. Robot Da Vinci 6
1.7. Robot Flexley Stack 7
1.8. Robot iw.hub 8
1.9. Robot Skypod 8
1.10. Catálogo de cursos de The Construct 10
1.11. Catálogo de ejercicios de Robotics Academy 11

3.1. Blender 19
3.2. Simulador Gazebo 19
3.3. Estructura de OMPL 21
3.4. Robot TurtleBot2 22
3.5. Base Kobuki 23
3.6. Sensor cámara Logitech Webcam Pro 9000 23
3.7. Sensor RPLIDAR A2 24
3.8. Arquitectura de comunicación entre la RADI y página web de Robotics Academy v4.X 25
3.9. Estructura jerárquica del directorio de un ejercicio en RoboticsAcademy v5.1 27
3.10. Lógica de transción en la RADI para el lanzamiento de una aplicación . . 28
3.11. Estructura jerárquica del directorio de un ejercicio en RoboticsAcademy v5.2 29

4.1. Hospital de AWS 31
4.2. Vieja arquitectura de comunicación del teleoperador 32
4.3. Nueva versión de la arquitectura de comunicación del teleoperador 32
4.4. TurtleBot2 simulado en Gazebo 33
4.5. Página Web del ejercicio Sigue-Persona con robot simulado 35
4.6. Página Web del ejercicio Sigue-Persona con robot real 36
4.7. Estructura jerárquica del directorio react-components del ejercicio Sigue-Person 36

4.8. Arquitectura de funcionamiento de las plantillas Python 41
4.9. Algoritmo de navegación VFF 42
4.10. Arquitectura de comunicación de la persona teleoperada en Robotics Academy v5.2 45
4.11. Arquitectura de comunicación de la persona autónoma en Robotics Academy v5.2 46

5.1. Almacén pequeño 50
5.2. Diseños de la estantería del almacén pequeño 51
5.3. Almacén grande 51
5.4. Diseños de la estantería del almacén grande 52
5.5. Estructura jerárquica del directorio *amazon_robot* 52
5.6. Diseños de la plataforma del robot logístico holonónimo 53
5.7. Diseños del robot logístico holonómico 54
5.8. Diseños del robot logístico holonómico visualizando la malla de colisión . . 54
5.9. Modelado del robot Ackermann en Blender 55
5.10. Estructura jerárquica del directorio *ackermann_logistic_robot* 57
5.11. Mapas 2D de los almacenes 57
5.12. Página Web del ejercicio Almacén-Amazon 58
5.13. Estructura jerárquica del directorio *react_components* del ejercicio Almacén-Amazon 58
5.14. Planificación de la ruta para ir a la zona de carga 66
5.15. Planificación de la ruta para volver a la zona de descarga 66
5.16. Máquina de estados finitos 70
5.17. Planificación de la ruta para ir a la zona de carga 72
5.18. Planificación de la ruta para volver a la zona de descarga 72
5.19. Modelado de los elementos del entorno 73
5.20. Modelado del robot 74
5.21. Resultados de la ejecución de la ruta 78

Índice de tablas

4.1. Listado de plantillas shared utilizadas en el ejercicio Follow Person 38

5.1. Resultados obtenidos con diferentes planificadores 75
5.2. Resultados obtenidos con diferentes tiempos de ejecución 76
5.3. Resultados obtenidos con diferentes tamaños de pasos de propagación . . . 76
5.4. Resultados obtenidos con diferentes duraciones máximas de control 76

Listado de códigos

3.1. "Hola mundo" en Python 15
3.2. "Hola mundo" en C++ 16
3.3. "Hola mundo" en JavaScript en consola 16
3.4. "Hola mundo" en HTML 17

4.1. Constructor del módulo HAL en el ejercicio Sigue-Persona 39
4.2. Función para calcular el vector de atracción 43
4.3. Función para calcular el vector de repulsión 44
4.4. Función para el procesamiento de datos entrantes de la GUI 47

5.1. Ejemplo del código XML del cuerpo del robot Ackermann 56
5.2. Función para dibujar un circulo en canvas 59
5.3. Función *getRobotCoordinates()* 61
5.4. Función *isStateValid()* 64
5.5. Función *generatePath()* 65
5.6. Función *isStateValid()* 68
5.7. Función *isStateValid()* 71
5.8. Formato de los waypoints en la planificación basada en control 75

Capítulo 1
Introducción

El término robótica fue acuñado por Isaac Asimov quien lo define como la ciencia que se encarga del estudio de los robots. La palabra robot fue utilizada por primera vez por el escritor de ciencia ficción Karel Capek en su obra R.U.R (*Robots Universales Rossum*) refiriéndose a unas máquinas pensantes que se sublevan y terminan matando a su creador. Este concepto originalmente nacido en las obras de ciencia ficción poco a poco ha trascendido hacia la realidad adquiriendo un papel imprescindible en la sociedad del siglo XXI.

Los avances en campos como la mecatrónica, electrónica, informática, inteligencia artificial, entre otras disciplinas fundamentales que conforman la robótica han permitido que los sistemas robóticos lleguen a ser capaces de completar una amplia gama de tareas, desde las más simples y repetitivas, como las de *pick and place* en factorías, hasta aquellas que requieren delicadeza y hacer operaciones complejas, como los realizadas por el robot quirúrgico Da Vinci. La integración de robots en nuestra vida cotidiana está mejorando significativamente la calidad de vida de las personas.

Generalmente, la robótica se suele dividir en dos ramas principales:

- **Robótica industrial**: centrada en el desarrollo y aplicaciones de robots en entornos controlados de fabricación y producción para realizar tareas tales como *pick and place* o ensamblado de piezas. Los robots más conocidos en este campo son los brazos robóticos.

- **Robótica de servicio**: centrada en el desarrollo de robots destinados a operar en entornos como el hogar o instalaciones públicas. Incluye robots domésticos, de entretenimiento, sanitarios, de logística, de inspección, etc.

En este capítulo se va a describir el contexto del presente TFG (*Trabajo Fin de Grado*), encuadrado en la enseñanza de robótica de servicio a futuros ingenieros robóticos.

1.1. Robótica de servicio

Como se mencionó anteriormente, la robótica de servicio se enfoca en diseñar robots destinados a funcionar en entornos no industriales, los cuales se caracterizan por ser

escenarios variados y pocos controlados. Por esta razón, las tareas que deben realizar los robots de servicio son generalmente más complejas que aquellas llevadas a cabo por los robots en cadenas de producción industrial. Esto requiere que estén equipados con una mayor cantidad de sensores y actuadores, así como con algoritmos más sofisticados que les permitan responder a una variedad más amplia de necesidades y situaciones. Dependiendo del contexto específico en el que operen, los robots deben estar dotados de habilidades avanzadas tales como entendimiento del entorno, navegación autónoma, capacidad de reactividad, percepción y detección de objetos, etc..

Hoy en día, los robots de servicio están presentes en casi todos los ámbitos de trabajo. A continuación, se hace un listado de algunas de las aplicaciones más destacadas de la robótica de servicio:

- **Robots de limpieza**: incluye aspiradoras robotizadas, friegasuelos automáticos, robots de limpieza de ventanas, entre otros, y se utilizan tanto en espacios domésticos como públicos. Estos robots automatizan las tareas rutinarias de limpieza permitiendo a los usuarios ahorrar tiempo y esfuerzo.

- **Robots en logística**: comúnmente, se utilizan en almacenes, fábricas o en reparto de pedidos. Estan equipados con sistemas avanzados de navegación y sensoriales que les permiten realizar tareas como transporte de mercancía, preparación de pedidos, gestión de almacenes, etc.

- **Robots de entretenimiento**: incluye juguetes robóticos que proporcionan compañía y diversión, y robots educativos utilizados tanto en enseñanza pre-universitaria como universitaria.

- **Robots en inspección**: se utilizan en una gran variedad de entornos incluyendo plantas petrolíferas o de gas, calderas y chimeneas industriales, y otros entornos hostiles. Estos robots permiten la inspección de lugares que resultan peligrosos o difíciles de alcanzar para los humanos proporcionando una manera segura y eficiente de realizar el mantenimiento de las instalaciones.

- **Robots en agricultura**: facilitan la implementación de la agricultura de precisión maximizando la productividad agrícola. Pueden realizar tareas como sembrar, cosechar, control de malas hierbas, monitorización de cultivos de manera más eficiente y económica, y con menor impacto ambietal que los métodos tradicionales.

- **Robots en salud**: engloba todos los robots que trabajan dentro del entorno hospitalario, incluyendo robots de rehabilitación, robots quirúrgicos, robots asistenciales, etc. El propósito de estos robots es mejorar la efectividad de los tratamientos médicos.

- **Robots en conducción autónoma**: los vehículos autónomos buscan aumentar la seguridad vial al reducir los accidentes causados por errores humanos. Estos vehículos están equipados con sistemas avanzados de sensores y algoritmos de inteligencia

artificial que les permiten leer señales de tráfico, analizar la situación y reaccionar en tiempo real. Además de mejorar la seguridad, también potencia la movilidad de personas con limitaciones físicas, ofreciendo nuevas oportunidades de transporte independiente.

Dentro de las diversas subdisciplinas de la robótica de servicio se profundizará un poco más en la robótica social y en la robótica logística, ya que estos dos campos son el foco central del presente trabajo.

1.1.1. Robótica social

Los robots sociales son aquellos robots capaces de mantener relaciones de interacción con humanos. En el contexto de las interacciones robot-humano, los robots pueden tomar diversos roles: medio de entretenimiento y herramienta de educación; agente de atención al cliente, recepcionista o guía turístico; acompañante que brinda apoyo emocional, especialmente para niños y personas de tercera edad; asistente sanitario.

Dado que los robots sociales necesitan estar en estrecho contacto con las personas, los desarrolladores se esfuerzan considerablemente en el diseño de su apariencia para que resulten amigables. Hay estudios que demuestran una mayor afinidad por parte de las personas hacia los robots que presentan una apariencia cercana a la humana, siempre que no caiga en el valle inquietante[1]. Por esta razón, los robots humanoides son una rama importante de investigación y entre los más reconocidos se encuentran:

- **Pepper**[2]: robot semihumanoide fabricado por SoftBank Robotics totalmente personalizable. Está diseñado para interactuar con los humanos, es capaz de entender tanto el lenguaje verbal como no verbal, identificar rostros y emociones humanas. Muchas empresas han adoptado Pepper como robot recepcionista.

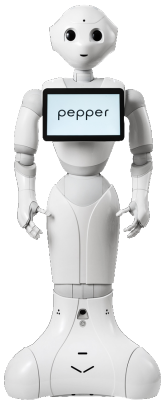


Figura 1.1: Robot Pepper

[1] https://es.wikipedia.org/wiki/Valle_inquietante
[2] https://www.aldebaran.com/es/pepper

- **Atlas**[3]: robot humanoide bípedo desarrollado por Boston Dynamics con la capacidad de correr, saltar, usar herramientas y manipular objetos con sus manos. Su aplicación se enfoca en la investigación, pero también tiene gran potencia en misiones de rescate o exploración de entornos peligrosos.

Figura 1.2: Robot Atlas

Sin embargo, el hecho de que muchos robots sociales sean humanoides no excluye la existencia de robots con otras formas que son bien recibidos por la sociedad:

- **Aibo**[4]: perro robot de la empresa Sony que se diseñó con la intención de que tomara un papel de mascota robótica. Puede hacer expresiones faciales simples como parpadear.

Figura 1.3: Robot Aibo

- **Paro**[5]: robot en forma de foca diseñado por la empresa japonesa Intelligent Systems

[3] https://bostondynamics.com/atlas/
[4] https://us.aibo.com/
[5] https://www.paroseal.co.uk/

con el objetivo de ayudar en terapias demencia y estimulación afectiva.

Figura 1.4: Robot Paro

En el ámbito sanitario, la aplicación de la robótica social abarca mucho más que el apoyo en salud mental. Los robots también desempeñan aquí un papel crucial en el apoyo a los profesionaes durante tratamientos técnicos como operaciones quirúrgicas y terapias de rehabilitación.

- **Atlas 2030**[6]: exoesqueleto pediátrico desarrollado por Marsi Bionics, diseñado para ayudar a niños con discapacidades motoras. Puede utilizarse de forma pasiva marcando patrones de movimiento personalizados o de forma activa utilizando la propia fuerza muscular del niño.

Figura 1.5: Robot Atlas 2030

- **Da Vinci**[7]: robot quirúrgico desarrollado por Intuitive Surgical para realizar cirurgías mínimamente invasivas. Puede evitar los temblores humanos y completar los movimientos con una precisión y control que superan las capacidades humanas.

[6] https://www.marsibionics.com/atlas-pediatric-exo/
[7] https://www.intuitive.com/en-us/products-and-services/da-vinci

También ofrece mejor maniobrabilidad quirúrgica con la capacidad añadida de ampliación de la imagen.

Figura 1.6: Robot Da Vinci

1.1.2. Robótica en logística

Las principales aplicaciones de la robótica en el sector logístico incluyen el almacenamiento y el transporte interno de mercancías, la preparación de pedidos y el reparto de ellos. Incorporar robots en estos procesos ofrece varias ventajas:

- Optimización del uso del espacio de almacenamiento: los robots requieren menos espacio para maniobrar, lo que permite a los almacenes organizar las mercancías de manera más densa, reduciendo el área necesario para los pasillos. Además, la capacidad para operar en altura facilita el uso de estanterías más altas, maximizando el volumen utilizable del almacén.
- Mejora de la eficiencia: los robots pueden trabajar durante periodos más largos sin descanso, manejar cargas pesadas con facilidad, mantener una control preciso de los pedidos y cometer menos errores durante el procesamiento de ellos.
- Reducción del riesgo de accidentes laborales: la incorporación de los robots elimina la necesidad de que los empleados realicen trabajos peligrosos, como por ejemplo, elevar cargas pesadas o acceder a zonas elevadas.

Durante las dos últimas décadas, la compra y venta en línea ha experimentado un crecimiento exponencial, lo que ha llevado a muchas empresas a invertir en la robótica logística para optimizar sus cadenas de suministro. En 2012, la empresa Amazon empezó a introducir la robótica en sus centros logísticos, y hoy en día cuenta con más de 100.000

robots en sus almacenes trabajando codo a codo con los empleados[3][8]. Igualmente, BMW empezó a introducir robots autónomos en su cadena de suministro en 2015.[9]

En el proceso de transporte nos podemos encontrar con AGVs (*Automated Guided Vehicle*s) y AMRs (*Autonomous Mobile Robot*s). Los AGVs son vehículos filoguiados con caminos preestablecidos y marcadores, mientras que los AMRs no necesitan marcadores y son reactivos al entorno.

Algún ejemplo de los robots logísticos existentes en la actualidad son:

- Flexley Stack[10]: robot móvil autónomo con horquillas fabricados por ABB, son carretillas elevadoras para el transporte y almacenamiento de cargas paletizadas. Destaca por su versatilidad de aplicación, es preciso, seguro y está equipado con sensores como lector de barras o pesaje de cargas.

Figura 1.7: Robot Flexley Stack

- iw.hub[11]: robot móvil autónomo desarrollado por Idealworks, un filial del Grupo BMW. Es el ganador del premio International IFOY (*Intralogistic and Forklift Truck of the Year*) en 2021. Tiene una plataforma para transportar mercancías con tracción diferencial. Equipado con Lidars, utiliza la tecnología SLAM (*Simultaneous Localization and Mapping*) para la navegación y destaca por su capacidad de operar de manera segura y eficiente.

[8] `https://www.youtube.com/watch?v=IMPbKVb8y8s`
[9] `https://www.youtube.com/watch?v=-ycBqH_Reqs`
[10] `https://new.abb.com/products/robotics/es/robots/autonomous-mobile-robots/productos/flexley-stack`
[11] `https://idealworks.com/en/iw-hub-e/`

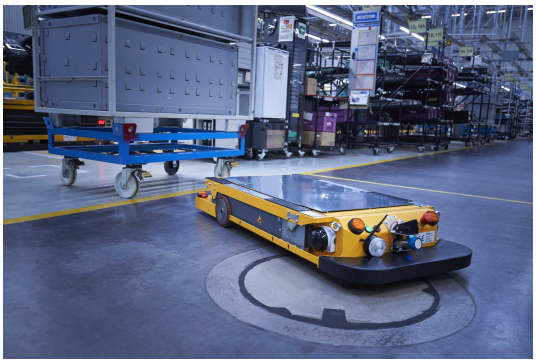


Figura 1.8: Robot iw.hub

- Skypod[12], robot móvil autónomo desarrollado por Exotec para la preparación de pedidos, se caracteriza por su capacidad de movimiento en tres dimensiones permitiendo el aprovechamiento máximo del espacio.

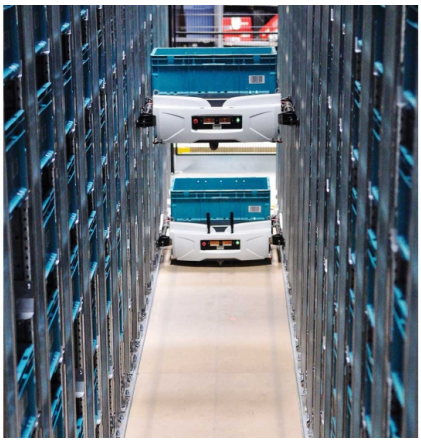


Figura 1.9: Robot Skypod

1.2. Robótica Educativa

Otra rama imprescindible en la que se encuadra el presente trabajo es la robótica educativa, la cual se puede distinguir en educación preuniversitaria y educación de ingeniería.

[12] `https://exotecbydexter.com/skypod/`

En la enseñanza preuniversitaria, la integración de la robótica en la educación permite a los estudiantes desarrollar las materias STEM (*Science, Technology, engineering and Mathematics*) de una forma entretenida. Los alumnos pueden aprender conceptos difíciles de manera práctica y no solo teórica. Mediante la programación de robots, además de desarrollar un pensamiento computacional también ayuda a adquirir competitividades en otros ámbitos como el fomento en la colaboración mediante trabajo en grupo, habilidad de aprender de los errores, fomento en la creatividad, etc. El nivel de enseñanza en esta etapa suele ser básico, utilizando sensores y actuadores fáciles de manejar para completar tareas de poca complejidad. Además, en muchos casos, se utiliza la programación por bloques visuales, como la ofrecida por Scratch[13].

Por otro lado, en el ámbito de la ingeniería, el objetivo es formar profesionales en el sector de la robótica. La educación en este nivel se vuelve más compleja y técnica. Los estudiantes comienzan a trabajar con robots más avanzados, como el TurtleBot2 o brazos robóticos industriales, que requieren una comprensión profunda de los sistemas robóticos y sus aplicaciones.

Dado que la enseñanza de la robótica con robots físicos puede tener un costo muy elevado, especialmente en la enseñanza de ingeniería, existen muchas plataformas web que ofrecen el aprendizaje en mundos simulados. Estas plataformas combinan tecnologías web con la enseñanza de robótica en ingeniería, como por ejemplo ROS (*Robot Operating System*) y simuladores como Gazebo. A continuación, se presentarán tres de las plataformas web educativas más conocidas:

- **Riders.ai**[14]: plataforma educativa para aprender robótica de forma competitiva. Tienen cursos Junior STEM dirigidos a estudiantes de escuelas intermedias y Senior STEM para estudiantes de secundaria. Ofrece simuladores realistas y competiciones en líneas donde los estudiantes pueden poner en práctica los conocimientos aprendidos.

- **CoderZ**[15]: plataforma educativa ideal para niños y adolescentes que desean iniciarse en la robótica. Permite a los jovenes fomentar sus habilidades STEM a través de cursos prácticos en entornos gamificados mediante una programción visual por bloques.

- **The Construct**[16]: plataforma educativa para aprender robótica basada en ROS. Ofrece cursos teóricos de robótica y tutoriales en línea para programar robots con ROS en simuladores situados en la nube. Los cursos abarcan desde conceptos básicos como "Matemáticas básicas para robótica" o "Conceptos Básicos de ROS", hasta niveles avanzados como "Árboles de comportamiento para ROS2" o "Seguridad ROS2". Es una plataforma adecuada tanto para estudiantes que quieran adentrarse en la robótica como para profesionales que busquen profundizar sus conocimientos.

[13] `https://scratch.mit.edu/`
[14] `https://riders.ai/en`
[15] `https://gocoderz.com/`
[16] `https://www.theconstruct.ai/`

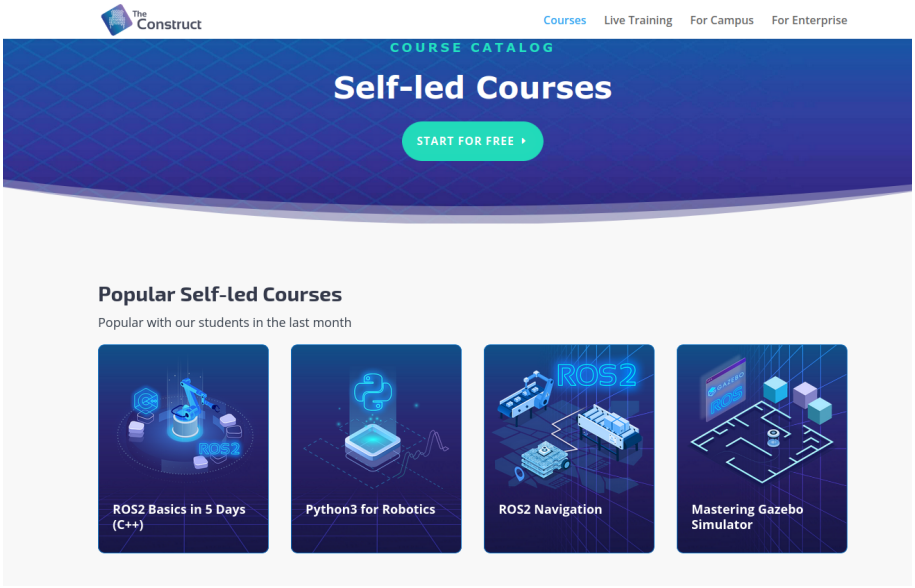


Figura 1.10: Catálogo de cursos de The Construct

1.2.1. Robotics Academy

Robotics Academy[17] es una plataforma educativa offline de robótica mantenida por la organización JdeRobot[18]. Se basa en el middleware ROS, el simulador Gazebo y el lenguaje de programación Python. Su objetivo principal es proporcionar a los usuarios un entorno práctico para aprender y reforzar conocimientos robóticos. La plataforma ofrece un amplio catálogo de ejercicios cubriendo áreas como visión computacional, robots móviles, robots de servicio, drones, etc., y cada ejercicio va acompañado de una documentación para guiar a los estudiantes en la resolución del mismo.

Los componentes principales de la plataforma son: una página web que actúa de interfaz gráfica de usuario y el contenedor docker llamado RADI (*Robotics Academy Docker Image*) que encapsula todo software backend. Para utilizar dicha plataforma el usuario tendrá primero que descargarse la imágen docker y lanzar el contendor. Una vez que el contenedor está arrancado, el usuario ya podrá dirigirse a la URL (*Uniform Resource Locators*) correspondiente de la página y seleccionar el ejercicio que quiere realizar. En la interfaz gráfica de cada ejercicio se muestrá cuatro componentes: una editor de texto donde el usuario tiene que escribir su código, una venta con el simulador Gazebo (o Rviz en el caso de utilizar robots reales), una ventana con una consola de texto para depuración y un cuarto espacio que según las peculiaridades de cada ejercicio se utilizará para un fin u otro.

[17] `https://jderobot.github.io/RoboticsAcademy/`
[18] `https://jderobot.github.io/`

Una de las ventajas que ofrece Robotics Academy es que todas las dependencias robóticas están preinstaladas en el contenedor docker, de manera que el usuario solo tiene que centrarse en el estudio e implemetación de los algoritmos.

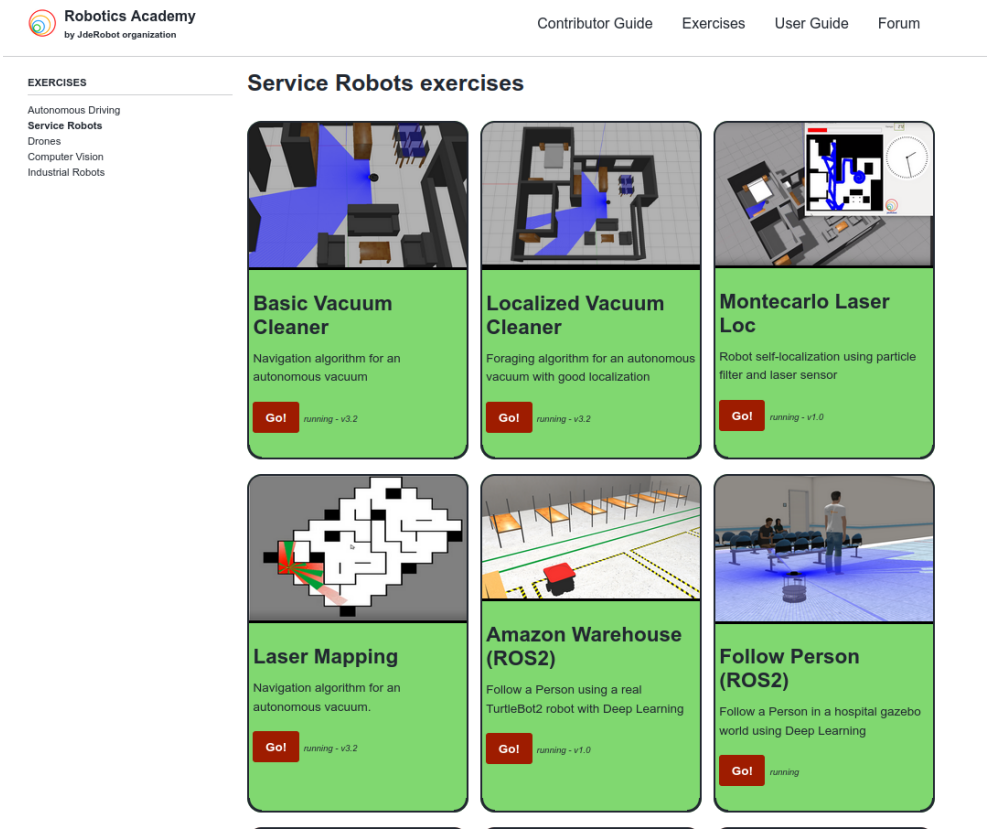


Figura 1.11: Catálogo de ejercicios de Robotics Academy

1.2.2. Unibotics

Unibotics[19] es la versión online de Robotics Academy, comparten las mismas tecnologías y lógica de funcionamiento. Adicionalmente, incorpora una base de datos de usuarios, lo que permite a los registrados guardar sus proyectos y monitorizar su progreso. Además, cuenta con un foro para que los usuarios puedan publicar sus problemas y buscar ayuda mutuamente.

[19] `https://unibotics.org/`

Capítulo 2
Objetivos y Metodología

2.1. Objetivos

El objetivo principal de este TFG es mejorar y ampliar la gama de ejercicios ofrecidos en la plataforma Robotics Academy con dos ejercicios educativos sobre Robótica de Servicio.

El primer ejercicio, denominado "Sigue Persona" (*Follow Person*), consiste en programar un robot TurtleBot2 para realizar una tarea de seguir una persona tanto si esta se mueve autonomamente como si lo hace teleoperada. Se ofrece al alumno la posibilidad de utilizar no solo un robot simulado sino también el robot real. El ejercicio no es completamente nuevo, sino que ya existía una versión implementada por Carlos Caminero en su TFG[2] en ROS 2 Foxy. Sin embargo, puesto que la plataforma ha hecho renovaciones profundas en su arquitectura de funcionamiento y ROS ha lanzado una nueva distribución LTS, ROS 2 Humble, el objetivo concreto es actualizar y migrar este ejercicio a la versión más reciente de Robotics Academy, v5.1 (hasta febrero del año 2024), utilizando ROS 2 Humble.

El segundo ejercicio, llamado "Almacén Amazon" (*Amazon Warehouse*), es totalmente nuevo y consiste en programar un robot logístico para transportar una estantería de un lugar a otro en un almacén simulado. En este ejercicio se ofrecen dos almacenes a elegir y dos tipos de robots: uno con geometría cúbica y movimiento holonómico, y otro con geometría ortoédrica y movimiento tipo Ackermann. El desafío propuesto para los alumnos es emplear la librería OMPL[5] para la planificación de la ruta de navegación del robot dentro de ese almacén para mover estanterías de un lugar a otro y la ejecución de esa ruta.

Para cada ejercicio se deben desarrollar los siguientes ingredientes:

- Infraestructura, elementos simulados o soporte del robot real.
- Componentes: plantillas Python, frontend web y backend del ejercicio en la plataforma Robotics Academy.
- Solución de referencia.
- Documentación de apoyo al usuario acompañada de un vídeo demostrativo.

2.2. Metodología

El modelo de trabajo seguido durante el desarrollo del TFG es:

- Empleo de la metodología Scrum organizando el trabajo por *sprints*. Reuniones semanales presenciales o por videollamada con el tutor para compartir los avances, resolver dudas y establecer nuevas tareas para la siguiente semana.
- Uso de un repositorio público[1] bajo la cuenta GitHub de RoboticsLabURJC para desarrollar los trabajos realizados. También se trabajó con los repositorios oficiales de RoboticsAcademy[2] y RoboticsInfraestructure[3].
- Actualización de un blog[4] con los avances semanales.
- Colaboración con el equipo internacional de desarrolladores de Robotics Academy para la migración de los ejercicios en la plataforma.
- Uso de Slack como medio de comunicación fluido tanto con el tutor como el equipo de desarrolladores de Robotics Academy.

2.3. Plan de trabajo

La planificación del TFG ha sido la siguiente:

1. Etapa de entrenamiento y familiarización con la base de código fuente.
    - Familiciación con Robotics Academy, reforzar los conocimientos de las tecnologías utilizadas (VNC, noVNC, GPU, HTML, React, Websockets, etc.) siguiendo tutoriales en Internet, instalar el software necesario y hacer pruebas locales.
    - Entender la base de código fuente de Robotics Academy, su organización y su lógica de funcionamiento para añadir nuevos ejercicios. Y participar en la construcción de la nueva imagen docker con base de Ubuntu 22.04.
2. Migración y mejora del ejercicio Sigue-Persona.
    - Revisión de la versión del ejercicio Sigue-Persona de Carlos Caminero.
    - Investigación de los drivers del TurtleBot2 simulado y real para ROS 2 Humble, primero en local y luego en el entorno dockerizado.
    - Ejercicio Sigue-Persona simulado

[1] `https://github.com/RoboticsLabURJC/2022-tfg-lucia-chen`
[2] `https://github.com/JdeRobot/RoboticsAcademy`
[3] `https://github.com/JdeRobot/RoboticsInfrastructure`
[4] `https://roboticslaburjc.github.io/2022-tfg-lucia-chen/`

        - Preparación del escenario simulado.
        - Desarrollo de las plantillas Python, el frontend web y la integración del ejercicio en la plataforma Robotics Academy.
        - Incorporación del control manual de la persona a seguir.
        - Revisión de la solución de referencia de Carlos Caminero.
    - Ejercicio Sigue-Persona real
        - Preparación del launcher del robot real.
        - Desarrollo de las plantillas Python, el frontend web y la integración del ejercicio en la plataforma Robotics Academy.
        - Revisión de la solución de referencia de Carlos Caminero.
    - Actualización de la documentación de usuario.

3. Creación del ejercicio Almacén-Amazon.
    - Investigación de la librería OMPL.
    - Primera fase: planificación puramente espacial con robot circular
        - Preparación del modelo de robot logístico holonómico simulado y creación de dos escenarios simulados: robot holonómico y almacén pequeño, robot holonómico y almacén grande.
        - Desarrollo de las plantillas python, el frontend web y la integración del ejercicio en la plataforma Robotics Academy.
        - Implementación de una solución de referencia para el robot holonómico basada en una *planificación euclídea* de la ruta de navegación.
        - Creación de la documentación de usuario.
    - Segunda fase: planificación espacial con restricciones geométricas del robot
        - Implementación de una solución de referencia para el robot holonómico incorporando restricciones por la *geometría del robot*.
        - Actualización de la documentación de usuario.
    - Tercera fase: planificación basada en controles con restricciones de geometría y movimiento del robot
        - Diseño y preparación del modelo robot logístico Ackermann, y creación de dos nuevos escenarios: robot Ackermann y almacén pequeño, robot Ackermann y almacén grande.
        - Integración de los nuevos escenarios en el ejercicio.
        - Implementación de una solución de referencia para el robot Ackermann incluyendo el *espacio de control*.
        - Actualización de la documentación de usuario.

4. Redacción de la memoria del TFG.

Capítulo 3
Herramientas utilizadas

A lo largo de este capítulo se introducirán y explicarán brevemente las tecnologías y herramientas en las que nos hemos apoyado para el desarrollo del presente trabajo.

3.1. Lenguajes de programación

3.1.1. Python

Python[1] es un lenguaje de programación interpretado de alto nivel, destacado por su énfasis en la simplicidad y legibilidad del código. Ofrece tipado dinámico, gestión automática de memoria y una extensa biblioteca estándar. Además, se trata de un lenguaje multiparadigma que incluye orientación a objetos, procedimental y funcional. Desde su creación en 1991 por Guido van Rossum, se ha establecido como uno de los lenguajes más populares y utilizados.

```
print("Hello world")
```

Código 3.1: "Hola mundo" en Python

En el presente trabajo, hemos utilizado la versión 3.10 y es el lenguaje base de la infraestructura interna de los ejercicios en Robotics Academy, así como el lenguaje empleado en las soluciones de referencia.

3.1.2. C++

C++[2] es un lenguaje de programación compilado de alto nivel desarrollado por Bjarne Stroustrup en 1979 como extensión del lenguaje de programación C añadiendo el paradigma de la POO (*Programación Orientada a Objetos*). Destaca por ser un lenguaje multiparadigma, la necesidad de gestión manual de memoria, ser fuertemente tipado y

[1] `https://www.python.org/`
[2] `https://www.cplusplus.com/`

tener una robusta biblioteca estándar, entre otras características. Además, cuenta con una de las comunidades de usuarios más grandes y activas en el ámbito de la programación.

```
#include <iostream>

int main ()
{
    std::cout << "Hola, mundo";
    return 0;
}
```

Código 3.2: "Hola mundo" en C++

En el presente trabajo, hemos utilizado la versión 20 y es el lenguaje utilizado en la programación del plugin para controlar el movimiento de una persona simulada en el simulador Gazebo mediante el teclado.

3.1.3. JavaScript

JavaScript[3] es un lenguaje de programación interpretado de alto nivel que se utiliza principalmente en el desarrollo de páginas web interactivas. Es un lenguaje dinámico basado en prototipos y multiparadigma con soporte para programación orientada a objetos, imperativa y funcional que se ejecuta en el lado del cliente de la web.

```
console.log('Hello World');
```

Código 3.3: "Hola mundo" en JavaScript en consola

3.2. Tecnologías web

En este apartado, ofreceremos una breve explicación de las tecnologías web empleadas en Robotics Academy, así como las que hemos manejado para integrar nuestros ejercicios a la plataforma. En esencia, Robotics Academy hace uso de HTML5 (*HyperText Markup Language 5*) para el diseño del frontend de los ejercicios, emplea websockets para establecer la comunicación entre el contenedor Docker y el navegador web. Además, utiliza noVNC para permitir la visualización y la interacción remota con el simulador Gazebo y consola de depuración desde el navegador.

[3] `https://developer.mozilla.org/es/docs/Web/JavaScript`

3.2.1. React

React[4] es una biblioteca de JavaScript de código abierto diseñada para construir interfaces de usuario interactivas de forma más sencilla, eficiente y organizada. Introduce el concepto de componentes reutilizables y utiliza un DOM (*Document Object Model*) virtual para mejorar el rendimiento al actualizar la interfaz del usuario.

En el presente trabajo, es la tecnología aplicada en la construcción de la página web de los ejercicios en Robotics Academy.

3.2.2. HTML5

HTML5[5] (*HyperText Markup Language 5*) es un lenguaje de marcado estándar para la creación y diseño de páginas web. Es la estructura esquelética de las páginas web y utiliza una serie de etiquetas para definir diferentes tipos de contenido, como párrafos, encabezados, listas, enlaces, imágenes, y más. En el presente trabajo hemos utilizado su versión más reciente, la ECMAScript 2023 (ES14).

```
<!DOCTYPE html>
<html lang="es">
<head>
<title>Titulo de la pagina</title>
</head>
<body>

<script>
  document.write("Hola Mundo");
</script>

</body>
</html>
```

Código 3.4: "Hola mundo" en HTML

3.2.3. WebSockets

WebSockets[6] es un protocolo de red basado en TCP (*Transmission Control Protocol*) que establece un canal de comunicación bidireccional y full-duplex entre el navegador del usuario y un servidor. Este diseño permite que el cliente y el servidor envíen datos simultáneamente manteniendo una comunicación en tiempo real con baja latencia. Los tipos de datos que pueden intercambiarse son textos o datos binarios.

[4] https://es.react.dev/
[5] https://developer.mozilla.org/es/docs/Web/HTML
[6] https://www.rfc-editor.org/rfc/rfc6455

3.2.4. VNC

Virtual Network Computing es un entorno de software libre basado en una arquitectura cliente-servidor, diseñado para permitir visualización y control remotos de un ordenador servidor desde un ordenador cliente utilizando el protocolo RFB (*Remote Frame Buffer*). Existen varias variantes de VNC (*Virtual Network Computing*) que ofrecen funciones adicionales; por ejemplo, TurboVNC[7], es el utilizado en Robotics Academy y está optimizado para aplicaciones que requieren un uso intenso de gráficos. Además, en combinación con VirtualGL[8] permite la renderización gráfica de aplicaciones 3D.

noVNC[9] es un cliente VNC de código abierto que permite el acceso a computadoras remotas directamente desde un navegador web, sin necesidad de instalar software adicional en el dispositivo cliente. Utiliza tecnologías web como HTML5 y WebSockets para establecer la conexión remota a los escritorios virtuales.

3.3. Docker

Docker[10] es un entorno de código abierto diseñado para facilitar el despliegue de aplicaciones de forma aislada. Se basa en la idea de contenerización que aísla la aplicación y sus dependencias en un contenedor que puede ser trasladado y ejecutado en cualquier sistema operativo con soporte Docker instalado. Con esta tecnología, los desarrolladores de aplicaciones pueden enfocarse en la creación del software sin tener que lidiar con las peculiaridades del entorno de ejecución. Técnicamente hablando, los contenedores Docker (*Docker Containers*) son instancias en ejecución de una imagen Docker. Las imágenes Docker (*Docker Images*) se construyen a partir de un archivo de texto llamado Dockerfile que contiene todas las instrucciones necesarias.

En Robotics Academy, se utilizan los contenedores Docker para encapsular la instalación de paquetes y dependencias robóticas necesarias para ejercutar los ejercicios en el entorno web, y de esta forma permitir a los usuarios centrarse exclusivamente en el desarrollo de los algoritmos. Por lo tanto, parte del trabajo de nuestro TFG ha consistido en garantizar la correcta ejecución de dos ejercicios nuevos en la imagen Docker oficial de Robotics Academy.

3.4. Blender

Blender[11] es un programa de código abierto para la creación de modelos y gráficos tridimensionales. Ofrece una amplia gama de funcionalidades que incluye modelado 3D, iluminación, animación y renderizado, edición de vídeos y creación de juegos.

[7] `https://www.turbovnc.org/`
[8] `https://www.virtualgl.org/`
[9] `https://novnc.com/info.html`
[10] `https://www.docker.com/`
[11] `https://www.blender.org/`

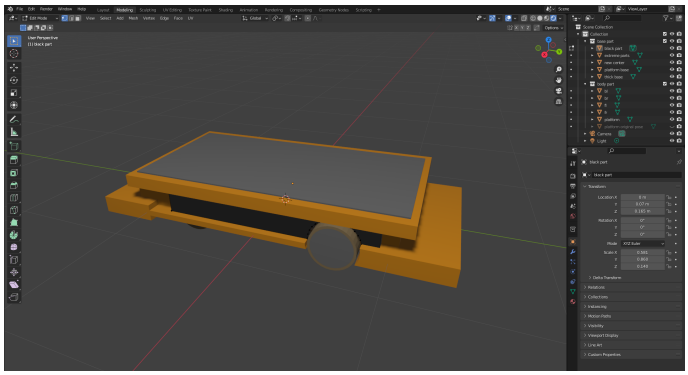


Figura 3.1: Blender

En el presente trabajo, empleamos Blender para la creación y modificación de las partes físicas de los robots, las cuales luego son exportadas en formato DAE (*Digital Asset Exchange*) facilitando su integración en la construcción del robot simulado.

3.5. Simulador Gazebo

Gazebo[12] es un simulador robótico tridimensional de código abierto que nos permite diseñar escenarios realistas y simular robots personalizados; permite a los usuarios probar algoritmos de control, planificación de movimientos, entre otros, sin el riesgo y el costo asociados con experimentos en el mundo real. Es una herramienta ampliamente utilizada en investigación y educación en robótica debido a su capadidad de simular gran variedad de sensores y actuadores, y ser compatible con varios marcos de robótica como ROS.

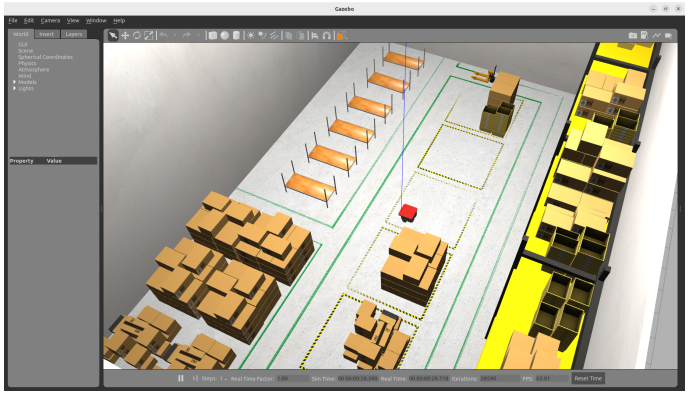


Figura 3.2: Simulador Gazebo

[12] `https://classic.gazebosim.org/`

La versión utilizada en el presente trabajo es Gazebo 11 y se utiliza como entorno de simulación para simular la operación de sigue persona con el TurtleBot2 en un hospital y para simular el trasporte de estanterías mediante robots logísticos en un almacén.

3.5.1. Formato SDF

SDF (*Simulation Description Format*)[13] es un formato XML (*Extensible Markup Language*) usado en Gazebo para la descripción de entornos y objetos. Permite representar de manera jerárquica y modular elementos como terrenos, iluminación, propiedades físicas, objetos estáticos y dinámicos reflejando la complejidad de un robot.

En el trabajo utilizamos la herramienta para crear los modelos simulados TurtleBot2, robot logístico holonómico y robot logístico Ackermann, así como los entornos simulados del hospital y almacenes.

3.6. Middleware robótico ROS2

ROS 2[14] (*Robot Operating System* 2) es un middleware utilizado en el desarrollo de software robótico. Proporciona los servicios necesarios para el diseño de sistemas robóticos complejos, como la abstracción de hardware, gestión de paquetes, comunicación entre procesos, etc. Es el sucesor de ROS y fue diseñado con el propósito de superar algunas limitaciones de la primera generación, como por ejemplo la incorporación de la comunicación en tiempo real usando DDS (*Data Distribution Service*) o mejora en la seguridad a través de la comunicación cifrada, la autenticación y el control de acceso.

Un concepto fundamental en ROS 2 son los nodos, que son procesos que realizan cálculos con propósitos específicos como por ejemplo controlar las ruedas de un robot. Estos se pueden comunicar e intercambiar mensajes a través de tópicos, servicios o acciones. En nuestro trabajo, empleamos la comunicación tópica para comandar acciones a los actuadores y recibir información de los sensores del robot. Esta comunicación sigue un modelo de comunicación asíncrona y anónima bajo el patrón de publicador-suscriptor. Los nodos que desean mandar un mensaje publican los datos en un tópico, mientras que los interesandos en recibir dicha información se suscribirán al tópico correspondiente.

Para el desarrollo de nuestro trabajo se utiliza la última versión estándar ROS 2 Humble.

[13] `https://sdformat.org/`

[14] `https://docs.ros.org/en/humble/`

3.7. Biblioteca de planificación OMPL

OMPL (*Open Motion Planning Library*) [15] es una biblioteca de software de código abierto diseñada para la planificación de movimiento basada en muestreo. Es ampliamente utilizada en robótica para la planificación de trayectorias de robots en diversos entornos, desde robots móviles hasta robots manipuladores.

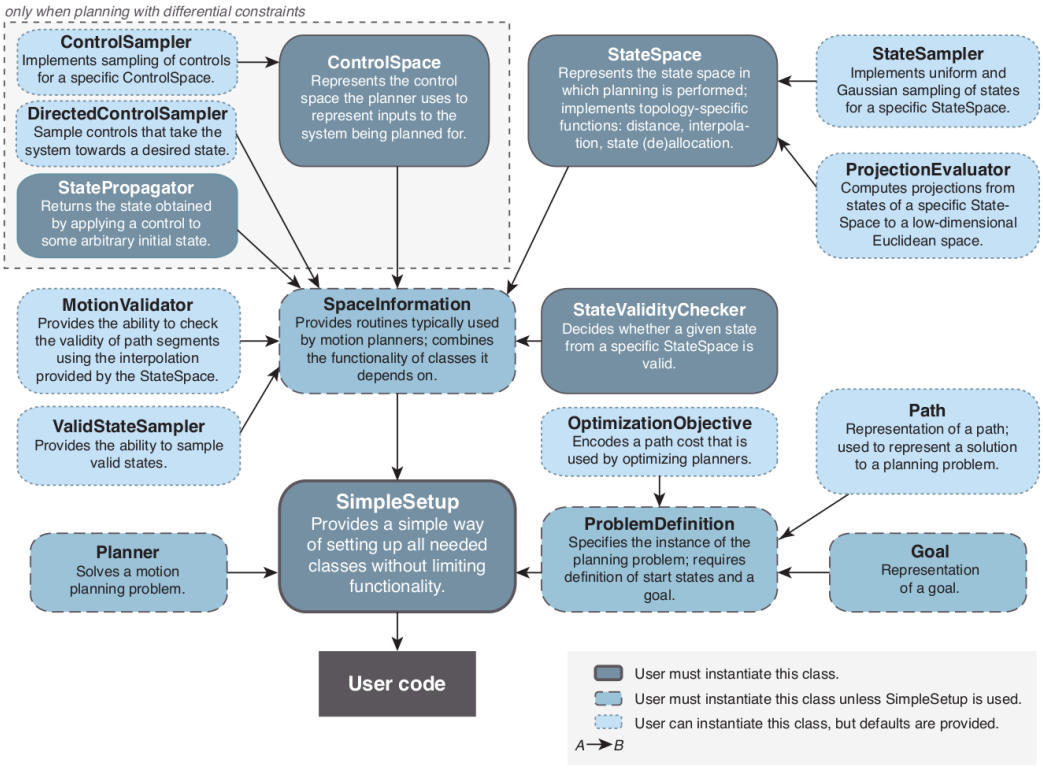


Figura 3.3: Estructura de OMPL

Los componentes principales son:

- Espacio de Estados (*State Space*), define las posibles configuraciones que puede tener un robot. Por ejemplo, *RealVectorStateSpace*, *SO2StateSpace*, *SE2StateSpace*, etc.
- Verificador de Validez de Estado (*State Validaty Checker*), determina si la configuración es válida, es decir, la configuración no colisiona con un obstáculo del entorno y respeta las restricciones del robot.
- Espacio de Control (*Control Space*), define los movimientos que puede tener un robot.

[15] `https://ompl.kavrakilab.org/`

- Propadador de Estado (*State Propagator*), indica la evolución del sistema después de aplicar un control.
- Información del Espacio (*Space Information*), es el contenedor que mantiene el espacio de estados, el verificador de validez del estado y otra información necesaria para la planificación.
- Planificador (*Planner*), responsable de generar un camino desde el inicio hasta la meta en el espacio de configuración. OMPL admite una variedad de planificadores, como RRT, PRM y FMT*.
- Camino (*Path*), es la salida del planificador, que es una secuencia de estados que representan una trayectoria para que el robot siga.

En el presente trabajo, se utiliza esta herramienta para la planificación de ruta en la solución de referencia para el ejercicio Almacén-Amazon.

3.8. Robot TurtleBot2

El TurtleBot2 es un robot móvil de bajo costo equipado con software de código abierto desarrollado por Yujin Robotics en colaboración con Willow Garage, y está diseñado tanto para investigación como para fines educativos. El robot consta de una base Kobuki y un cuerpo de soporte que permite la expansión de sus capacidades mediante la adición de sensores y actuadores. En nuestro proyecto específico, hemos ampliado sus funcionalidades con la incorporación de una cámara web y un láser RPLIDAR.

Figura 3.4: Robot TurtleBot2

3.8.1. Base Kobuki

La base, denominada Kobuki[16], es una plataforma circular con cinemática diferencial. Está equipado de sensores como odometría, giroscopio, sensores de contacto (*bumpers*) para

[16] `https://kobuki.readthedocs.io/en/devel/about.html`

detectar colisiones, sensores de desnivel, entre otros. Además, dispone de puertos USB para conectar nuestro portátil y los sensores o actuadores.

Figura 3.5: Base Kobuki

3.8.2. Cámara Logitech Webcam Pro 9000

Se trata de una cámara web desarrollada por Logitech, cuenta con un sistema de enfoque automático ultra suave y un sistema de lentes Carl Zeiss para videos más nítidos y fluidos. La cámara admite la grabación de video en HD y tiene un sensor de 2 megapíxeles que captura videos en formato 16:9 de hasta 8 megapíxeles de calidad. En nuestro trabajo, toma el papel de los 'ojos' del robot para seguir a la persona.

Figura 3.6: Sensor cámara Logitech Webcam Pro 9000

3.8.3. Láser RPLIDAR A2

El RPLIDAR A2[17] es un sensor Lidar 2D de bajo costo desarrollado por SLAMTEC para aplicaciones robóticas interiores. Tiene una capacidad de escaneo de 360 grados con frecuencia de 10 Hz y un rango de distancia de 0,2 a 12 metros. En nuestro trabajo, el dispositivo se utiliza como sensor de distancia para poder esquivar los obstáculos durante la tarea de seguir la persona.

[17] `https://www.slamtec.ai/product/slamtec-rplidar-a2/`

Figura 3.7: Sensor RPLIDAR A2

3.9. Plataforma Robotics Academy

Durante los últimos años Robotics Academy, siendo la versión *opensource* y *offline* de Unibotics, no ha parado de perfeccionar su funcionalidades y ampliar su menú de ejercicios. A continuación, se hará una breve descripción de las dos últimas versiones: la versión 5.1 y versión 5.2.

Nuestro TFG comenzó durante el desarrollo de la versión 5.1 de Robotics Academy, por lo tanto los ejercicios implementados inicialmente siguen la estructura de esta versión. Al acercarse la finalización del TFG, se inició el desarrollo de la versión 5.2 y los ejercicios fueron migrados a esta nueva versión por otros miembros de la asociación JdeRobot. Por nuestra parte, durante esta última etapa nuestra contribución se centró en la afinación y betatesting de los ejercicios.

Versión 5.1

En primer lugar, se hará una breve explicación de la versión anterior a esta, para poder posteriormente explicar con mayor claridad las renovaciones aplicadas.

La versión anterior es la versión 4.X, en la cual la RADI está basada en Ubuntu 20.04 y los ejercicios desarrollados en ROS Noetic. Además, la interfaz gráfica web está diseñada utilizando plantillas HTML. Y en el protocolo de comunicación entre la RADI y navegador utiliza cinco conexiones:

- WebServer, conexiones https.
- VNCserver para visualizar Gzviewer y la consola de texto.
- Websocket manager (Manager.py) encargado de controlar el ciclo de vida de las aplicaciones robóticas.
- code-websocket (Exercise.py), para enviar el código de usuario desde el navegador a la RADI, así como las ordenes de “run”, “stop”, “reset”, etc.

- GUI-websocket (Exercise.py), para enviar los datos internos del robot desde la RADI al navegador.

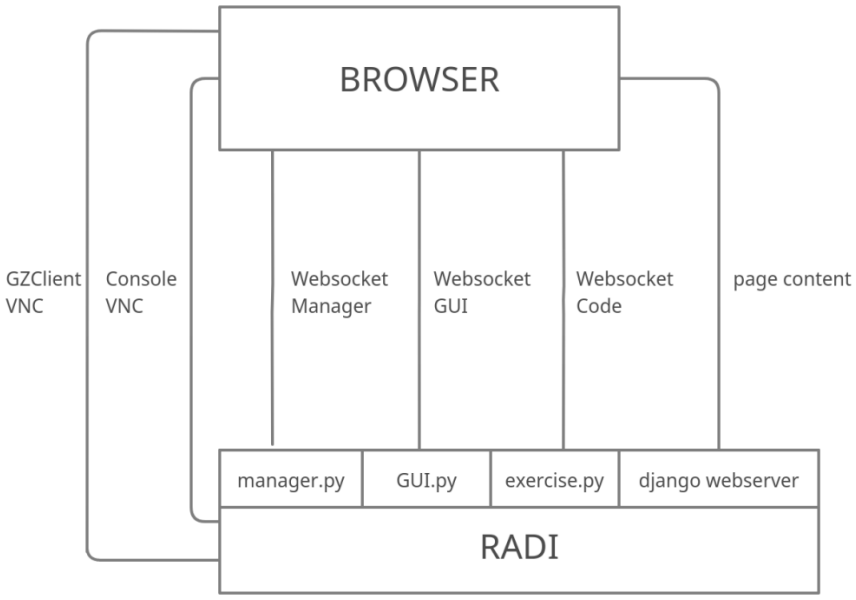


Figura 3.8: Arquitectura de comunicación entre la RADI y página web de Robotics Academy v4.X

Los repositorios implicados son:

- JdeRobot/RoboticsAcademy: en la rama principal contiene las plantillas backend y frontend de los ejercicios, y en la rama gh-pages la documentación.
- JdeRobot/CustomRobots: contiene los paquetes ROS y Gazebo no oficiales de los robots y mundos simulados.

Cada ejercicio, por un lado, tiene las plantillas Python, encargadas de resolver detalles de bajo nivel como el manejo de ROS Topics y ofrecer una interfaz de programación sencilla llamada HAL API y GUI API a los usuarios para la simplificación de la programación de las aplicaciones por parte de ellos. Por otro lado, cuentan con las plantillas web para la construcción del *frontend* del ejercicio. Para añadir un nuevo ejercicio hay que crear un directorio propio en *JdeRobot/RoboticsAcademy/exercises/static/exercises* que guarda las plantillas Python y un directorio en *JdeRobot/RoboticsAcademy/templates* que guarda las plantillas web. A pesar de compartir los mismos componentes con otros ejercicios, por ejemplo los botones, los ficheros HTML (*HyperText Markup Language*) correspondientes

tendrán que estar copiados en la carpeta del nuevo ejercicio, lo que hace dificultoso el mantenimiento de la plataforma.

Las renovaciones realizadas en la versión 5.1 son:

- Migración de los ejercicio desde ROS Noetic a ROS 2 Humble. Como consecuencia, la RADI pasa a utilizar base Ubuntu 22.04.
- Rediseño frontend y uso de React para crear componentes reutilizables que se puedan compartir entre diferentes ejercicios y facilitar el mantenimiento de ellos. Los componentes comunes están guardado en la carpeta *react_frontend* del repositorio *RoboticsAcademy*.
- Integración del concepto "universo" (configuraciones de los robots, mundos simulados, ficheros launcher de ROS, etc) e integración de un componente selector de universos en la página web.
- Nueva arquitectura RAM (*Robotics Application Manager*). Reducción de los canales de comunicación entre la RADI y el navegador a tres utilizando un único websocket.
- Refactorización de los repositorios GitHub. Translación de los mundos Gazebos y launchers ROS desde el repositorio RoboticsAcademy a RoboticsInfrastructure (anteriormente CustomRobots). De esta forma, todo objeto relacionado con la interfaz de tópicos ROS (robot, world y launcher) es considerado como universos robóticos que se podrá utilizar no solo en RoboticsAcademy sino también en las aplicaciones BT-Studio y VisualCircuit.

Los repositorios implicados son:

- JdeRobot/RoboticsAcademy[18]: en la rama principal contiene las plantillas backend y frontend de los ejercicios, y en la rama gh-pages la documentación.
- JdeRobot/RoboticsApplicationManager[19]: contiene el *manager* de las aplicaciones robótica.
- JdeRobot/RoboticsInfrastructure[20]: contiene los paquetes ROS y Gazebo no oficiales de los robots y mundos simulados, y los *launchers*.

En esta nueva versión, para cada ejercicio solo hay que crear un único directorio en *JdeRobot/RoboticsAcademy/exercises/static/exercises* que contiene tanto las plantillas Python como el frontend web específico del ejercicio:

- Directorio *python_template*:

[18] `https://github.com/JdeRobot/RoboticsAcademy`
[19] `https://github.com/JdeRobot/RoboticsApplicationManager`
[20] `https://github.com/JdeRobot/RoboticsInfrastructure/tree/humble-devel`

  - hal.py: módulo HAL (*Hardware Abstraction Layer*), actúa como una capa de abstracción de hardware para interactuar con los componentes del robot, como puede ser motores, lidars o cámaras. Dicho módulo usa unos ficheros Python ubicados en el directorio *interfaces* para comunicarnos con los nodos de ROS.
  - gui.py: módulo encargado de gestionar la actualización de la interfaz visual. Por un lado, la clase *GUI* administra los datos saliente y entrante de la página web por el puerto 2303 a tráves de websockets. Por otro lado, la clase *ThreadGUI* o *ProcessGUI*, en los ejercicios que utiliza *Python Multiprocessing*, gestiona la frecuencia de actualización de la GUI (*Graphical User Interface*).
  - exercise.py: encargado de estar escuchando las órdenes del cliente web como correr, pausar, detener el ejercicio, además de recibir el código del usuario. En la mayoría de los ejercicios, excepto *Follow Line* y el *Follow Person* implementado en el presente trabajo, también es el encargado de generar los módulos GUI y HAL utilizados en el editor de texto del usuario, y procesar el código de usuario. En estos ejercicios, estas funcionalidades se gestionan a través de la plantilla *brain.py*.

- Directorio *interfaces*: contiene las interfaces ROS de los robots.
- Directorio *react_componentes*: contiene los ficheros de JavaScript para el diseño del frontend.

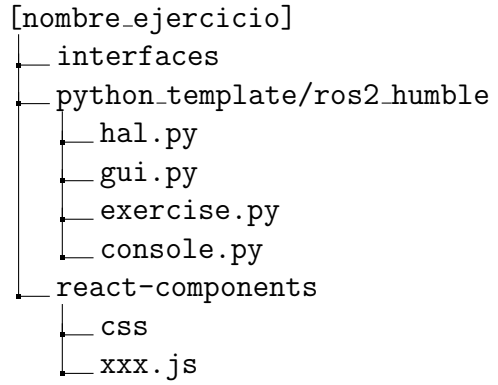


Figura 3.9: Estructura jerárquica del directorio de un ejercicio en RoboticsAcademy v5.1

Por otro lado, las aplicaciones robóticas son controladas por un *administrador*, llamado RAM. El RAM controla el ciclo de vida de las aplicaciones mediante una máquina de estados que se actualiza a petición del cliente. Los estados existentes son: `idle`, `connected`, `world_ready`, `visualization_ready`, `application_running` y `paused`. Y los comandos son: `connect`, `disconnect`, `launch_world`, `prepare_visualization`, `run_application`, `pause`, `resume`, `terminate` y `stop`.

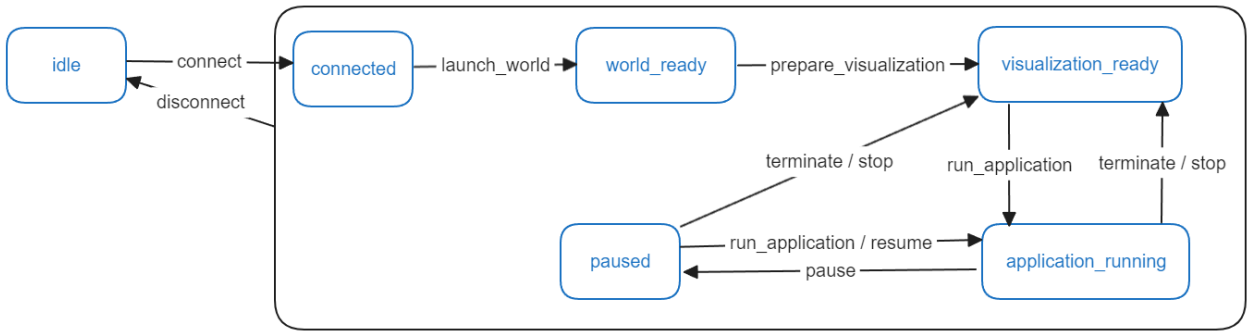


Figura 3.10: Lógica de transción en la RADI para el lanzamiento de una aplicación

Versión 5.2

Las mejoras en esta etapa respecto a la anterior son:

- Migración de los ejercicios desde plantillas gordas a plantillas finas y superfinas. Ahora, las plantillas Python solo proporcionan el módulo HAL, el módulo GUI y el controlador de frecuencia para las iteraciones del código de usuario, el resto de labor como es la recepción de los comandos para transitar de estados es asignado al RAM.
- Refactorización de el RAM a cuatro pasos independientes, ahora RAM es el encargado de gestionar toda la ejecución de las aplicaciones.
  - El navegador se conecta al RAM
  - El RAM lanza el universo lanzando el simulador Gazebo
  - El RAM prepara la visualización del navegador
  - El RAM pone a correr la aplicación robótica combinando la plantilla del ejercicio con el código del usuario.

  Con esta nueva arquitectura, cada vez que cambiamos de universo o recargamos el código, la ejecución de la aplicación es matada por completo y se inicia una nueva.
- Translación de las interfaces ROS desde el repositorio RoboticsAcademy a RoboticsInfrastructure.

Con los cambios realizados, ahora los directorios de los ejercicios en el repositorio RoboticsAcademy presentan la siguiente estructura básica:

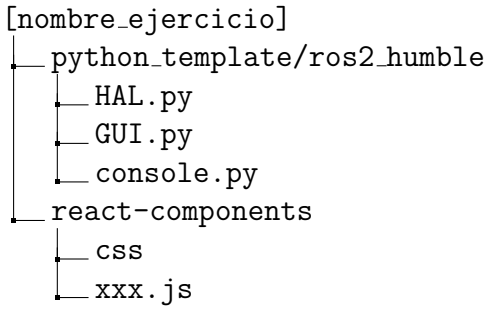


Figura 3.11: Estructura jerárquica del directorio de un ejercicio en RoboticsAcademy v5.2

- HAL.py: módulo con las funciones HAL de usuario.
- GUI.py: módulo con las funciones GUI de usuario y controlador de frecuencia del código de usuario. Además, es el encargado de procesar acciones, como entrada de teclado, que se realizan en el navegador.

Capítulo 4
Ejercicio Sigue-Persona en Robotics Academy

En este capítulo se describirá en detalle el proceso de elaboración del ejercicio Sigue-Persona para la plataforma Robotics Academy. Se explicarán la infraestructura, las plantillas backend y frontend que dan soporte al ejercicio, así como la solución de referencia.

En un principio, el ejercicio está desarrollado para la versión 5.1 de Robotics Academy y nuestra participación en la migración a la versión 5.2 ha sido parcial. Sin embargo, es muy interesante dedicar una sección al final de este capítulo para explicar brevemente el estado más reciente del ejercicio.

4.1. Enunciado

El objetivo para los estudiantes o usuarios de este ejercicio es implementar la lógica de un algoritmo de navegación que permita a un robot seguir una persona. El modelo del robot es un TurtleBot2 equipado con una cámara y un lidar. Los estudiantes tendrán que superar varios desafíos para completar exitosamente la práctica: detectar e identificar la persona, seguir a la persona y esquivar obstáculos durante el seguimiento.

Los estudiantes tendrán en disposición tanto el robot real como el robot simulado dentro de un entorno simulado de un hospital. Además, en el entorno simulado se ofrece la opción de seleccionar entre dos universos: el primero, con una persona autónoma que sigue una ruta preestablecida; y el segundo, con una persona que puede ser controlada y dirigida mediante el uso del teclado en vivo.

4.2. Infraestructura

En este apartado se introducirán los distintos elementos requeridos para el ejercicio Sigue-Persona, los objetos simulados, incluyendo el mundo y el robot simulado y sus correspondientes *drivers*, así como los *drivers* del robot real.

Como se ha mencionado previamente, este ejercicio ya tenía una versión en ROS 2 Foxy. Por lo tanto, decidimos no añadir complicaciones adicionales y mantener el diseño

utilizando el mismo entorno simulado y los mismos modelos de robots.

4.2.1. Entorno simulado de un hospital

El escenario elegido para el ejercicio simulado es el hospital proporcionado por AWS (*Amazon Web Services*). En el hospital se encuentran objetos como paredes, sillas, personas, etc., lo cual añade complejidad suficiente para poner a prueba la robustez del algoritmo implementado por el usuario.

El archivo correspondiente al mundo simulado se denomina *hospital_follow_person.world* y se encuentra ubicado en la carpeta *Worlds* del repositorio RoboticsInfrastructure. Además, existe una versión que incluye una cámara de observación siguiendo al robot, denominada *hospital_follow_person_followingcam.world*.

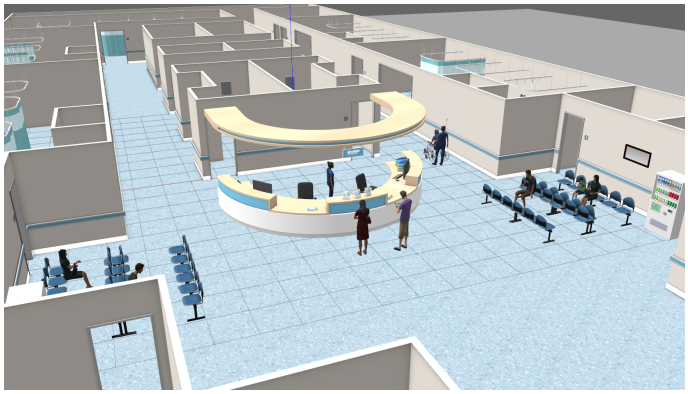


Figura 4.1: Hospital de AWS

4.2.2. Persona teleoperada

Uno de los elementos imprescindibles, además del mundo y el robot en el ejercicio simulado, es la persona móvil. La persona teleoperable es tomada del TFG de Carlos Caminero[2], en su trabajo tomó el modelo de persona simulada del TFM de Pedro Arias[4] y diseñó un plugin teleoperador para la persona en C++ alojado en el archivo *person.cpp*.

El plugin que diseñó presenta dos funcionalidades clave:

- Navegación autónoma: implementa una ruta preestablecida compuesta por una lista de puntos de paso (*waypoints*). Cada *waypoint* es una tupla *(x, y, index)*, con *(x,y)* denotando las coordenadas de la persona en el mundo simulado e *index* señalando al índice del siguiente *waypoint* en la lista.

- Control teleoperado: abre un canal de comunicación mediante socket con la RADI, y espera recibir cadenas de texto, las cuales traduce a órdenes de movimiento. Cuando

se pulsa la tecla *W* o *S* el modelo se mueve hacia delante o detrás y cuando se pulsa la tecla *A* o *D* el modelo rota hacia la izquierda o derecha.

La arquitectura de comunicación que sigue en el modo teleoperado es:

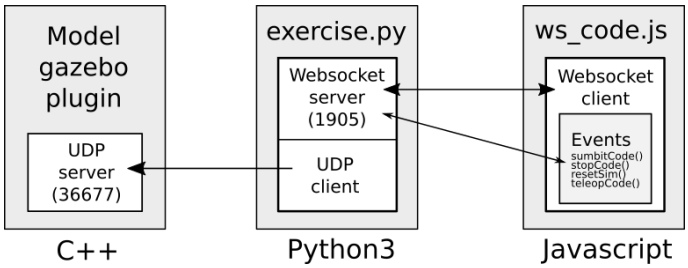


Figura 4.2: Vieja arquitectura de comunicación del teleoperador

La renovación respecto a su trabajo es la adaptación de la cadena de comunicación al nuevo diseño de la RADI y RAM, incluyendo avances como la independización de los elementos de comunicación del fichero *exercise.py* a un nuevo fichero *model_teleoperator.py*, e implementación de un nuevo programa JavaScript utilizando React. El diseño final tiene la siguiente arquitectura:

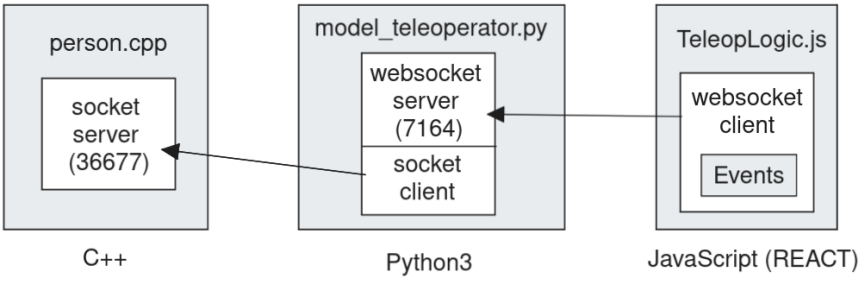


Figura 4.3: Nueva versión de la arquitectura de comunicación del teleoperador

En la página web el programa JavaScript, llamado *TeleopLogic.js* y escrito en React, capta eventos de teclado. Estos eventos se traducen a cadenas de texto y se envían a través de un websocket por el puerto 7164 a *model_teleoperator.py*, un componente auxiliar intermedio de Python que opera dentro de la RADI. Finalmente, este script elabora el mensaje recibido y lo reenvía a través de un socket por el puerto 36677 al plugin en Gazebo de la persona simulada.

Por otra parte, se renombró el modelo a *PersonToFollow* y se movió el archivo a *RoboticsInfrastructura/CustomRobots/amazon_hospital/models*.

4.2.3. Robot TurtleBot2 simulado

El modelo del TurtleBot2 simulado fue creado inicialmente por Carlos Caminero en su TFG para ROS 2 Foxy. Nuestro trabajo consistió simplemente en adaptarlo a ROS 2 Humble que gracias a la compatibilidad entre versiones no hubo mayores complicaciones.

El diseño del TurtleBot2 simulado sigue fielmente el modelo real y se organiza en dos directorios principales: *kobuki_description* y *turtlebot2*. En el directorio *kobuki_description* se encuentra el modelado de la base Kobuki, el cual fue tomado directamente del repositorio oficial de Kobuki, *kobuki_ros*. Por otro lado, el directorio *turtlebot2* contiene el diseño del cuerpo de soporte, incluyendo el lidar y la cámara, creados con URDF y Xacro utilizando figuras cúbicas y cilíndricas.

En cuanto a los controladores (*drivers*), se utiliza el plugin *libgazebo_ros_diff_drive.so*, que ofrece un topic llamado */cmd_vel* para comandar velocidades lineales y angulares a los motores. Se emplean los plugins *libgazebo_ros_ray_sensor.so* y *libgazebo_ros_camera.so* para publicar las lecturas del lidar y las imágenes capturadas por la cámara en los topics */scan* y */depth_camera/image_raw*, respectivamente.

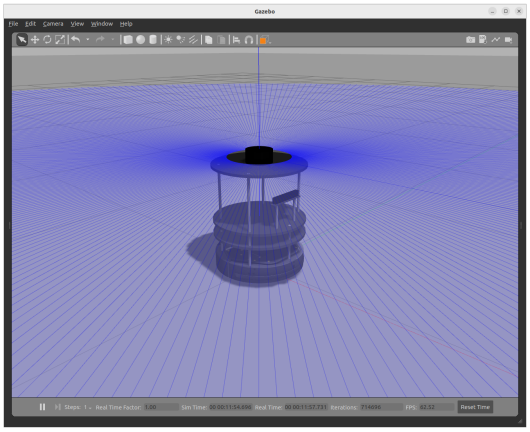


Figura 4.4: TurtleBot2 simulado en Gazebo

4.2.4. Robot TurtleBot2 real

En la sección 3.8 ya hemos realizado una descripción de la mecánica del TurtleBot2 real. Por lo tanto, en este apartado, nos centraremos en explicar cómo conectarnos a los tres *drivers* del robot. Inicialmente, abordaremos el proceso para realizar estas conexiones en un entorno local y, posteriormente, explicaremos cómo efectuarlas dentro de la RADI.

Para conectar con los motores, hemos utilizado los paquetes disponibles en los repositorios oficiales de la organización Kobuki. Específicamente, hemos usado los repositorios *kobuki_ros*, *kobuki_ros_interfaces*, *kobuki_core* y *kobuki_ftdi*, además de algunos otros de *ecl* y *sophus* que son necesarios para la correcta ejecución de estos:

- *kobuki_ros*: este repositorio contiene el paquete *kobuki_node*, que a su vez incluye el nodo *kobuki_ros_node* para controlar el motor de la base Kobuki y su odometría. Este nodo se suscribe al topic */commands/velocity* para recibir las instrucciones de movimiento a través de mensajes de tipo *geometry_msgs.msg.Twist*.
- *kobuki_ros_interfaces*: contiene las interfaces de mensajes, servicios y acciones ROS 2 para el Kobuki.
- *kobuki_core*: consta de bibliotecas y utilidades de C++ para trabajar con el robot Kobuki.
- *kobuki_ftdi*: proporciona herramientas para actualizar el convertidor USB (*Universal Serial Bus*) a serie FT232R desde FTDI en el Kobuki utilizado para asignar un indentificar fijo al dispositivo.

Para lanzar la base hemos tenido que crear un fichero de lanzamiento *kobuki_base.launch.py*.

```
ros2 launch rplidar_ros rplidar.launch.py
```

Para conectar con el RPLIDAR A2 utilizamos el paquete *rplidar_ros* del usuario de Github allenh1[1] en su rama para ROS 2. Con los paquetes instalados y compilados, podemos activar el dispositivos con el comando:

```
ros2 launch rplidar_ros rplidar.launch.py
```

La conexión de la cámara web se gestiona utilizando el paquete *usb_cam* de la rama ROS 2 del repositorio *ros_drivers*. Hemos optado por la instalación binaria, ya que es lo más sencillo:

```
sudo apt-get install ros-humble-usb-cam
```

Una vez instalado el paquete, podemos acceder a la cámara ejecutando el comando:

```
ros2 launch usb_cam camera.launch.py
```

Para simplificar el lanzamiento de los drivers, se ha creado un único fichero de lanzamiento *follow_person.launch.py* para lanzar los tres dispositivos a la vez.

[1] `https://github.com/allenh1/rplidar_ros`

Una vez probados los drivers en local, ha llegado el momento de integrarlos en la RADI. Dado el número de clonaciones de dependencias a realizar, se ha creado un fichero con la lista de los repositorios a clonar del Kobuki llamado *repos* ubicado en *kobuki_instructions*, de esta forma podemos importar todos los repositorios deseados con el comando:

```
vcs import < /kobuki_instructions/repos
```

Este comando es incluido en el fichero *Dockerfile* como instrucción a realizar al montar la imagen docker. Cuando se lance la aplicación del ejercicio en vez de lanzar un fichero de un mundo simulado, se lanza *follow_person.launch.py*.

4.3. Página Web

La página web de este ejercicio está dividida en cuatro áreas, en la izquierda se encuentra el editor de texo y la consola de depuración, y en la derecha la imagen enviada mediante la función *GUI.showImage()* y el visor del simulador Gazebo. Al entrar en el ejercicio, la página se lanza y conecta automáticamente a la RADI. El alumno solo tiene que esperar a que la aplicación esté preparada y empezar a programar.

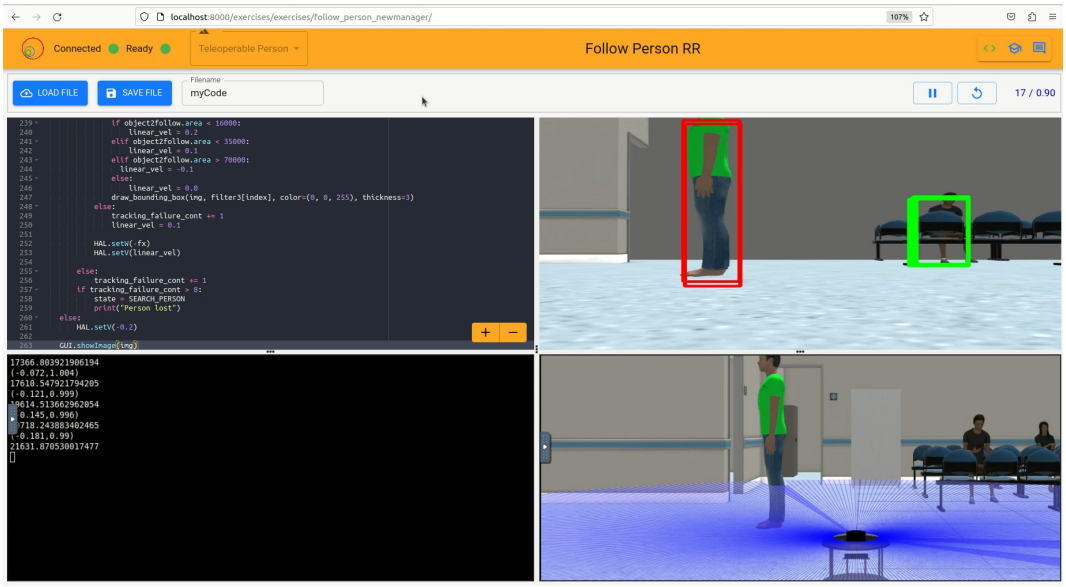


Figura 4.5: Página Web del ejercicio Sigue-Persona con robot simulado

En el caso de que hayamos elegido el universo del entorno real, se prepara una ventana Rviz en lugar del simulador Gazebo. Además, el usuario deberá esperar a que todos los

dispositivos del robot estén correctamente conectados, lo cual se puede verificar de la siguiente manera: la cámara encenderá una luz indicadora, el Lidar comenzará a girar, y la base emitirá una melodía como aviso de conexión.

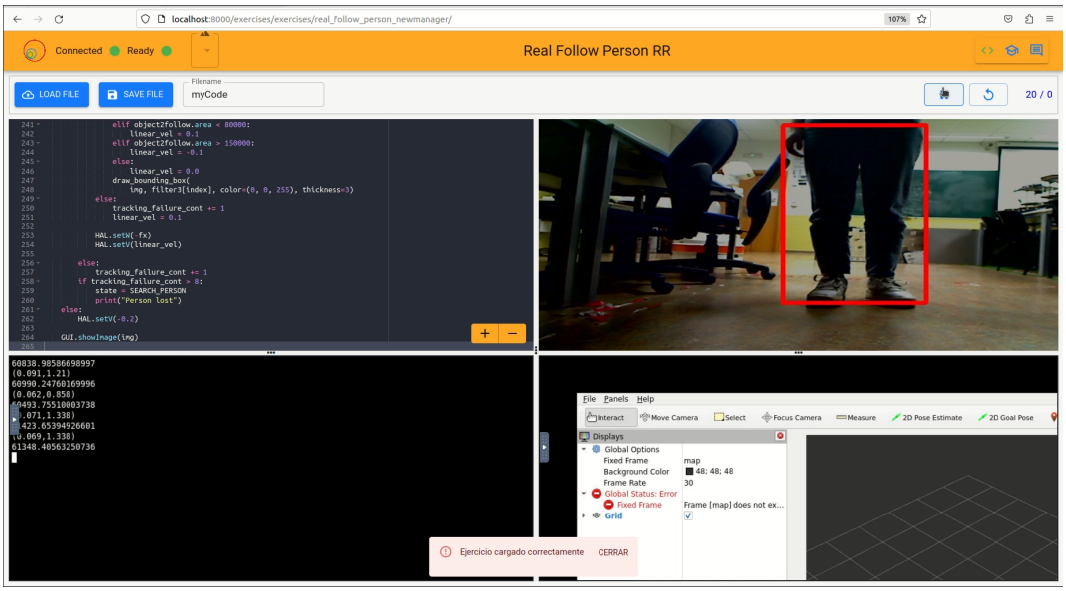


Figura 4.6: Página Web del ejercicio Sigue-Persona con robot real

Internamente, un gran avance de la plataforma respecto a la versión antigua, es el uso de plantillas React en vez de plantillas JavaScript puras.

```
react-components
├── css
│   └── FollowPersonRR.css
├── FollowPersonRR.js
├── ImgCanvas.js
├── MapSelectorPersonjs
└── TeleopLogic.js
```

Figura 4.7: Estructura jerárquica del directorio react-components del ejercicio Sigue-Person

A continuación, se detalla la utilidad de cada archivo:

- **FollowPersonRR.css**: hoja de estilo del ejercicio.
- **FollowPersonRR.js**: componente principal que importa y aplica la hoja de estilo del ejercicio a la página web.
- **ImgCanvas.js**: componente encargado de mostrar una imagen en la página web.

- **MapSelectorPerson.js**: componente selector del universo robótico del ejercicio.
- **TeleopLogic.js**: el componente se activa en el modo teleoperado del ejercicio que es cuando se abre el canal de comunicación websocket en el puerto 7164 para enviar las órdenes recibidas al *plugin* de la persona. Traduce las entradas de teclado a cadenas en formato "#key_", como por ejemplo si el evento capturado es la tecla "w", entonces el mensaje enviado sería "#key_w".

4.4. Plantillas Python

Una vez que tenemos la infraestructura preparada procedemos a construir el ejercicio. Para el desarrollo del ejercicio Sigue-Persona hemos utilizado como referencia plantillas del ejercicio *Follow Line*, aprovechando el hecho de que comparten características similares tales como el uso de un motor y una cámara como dispositivos principales, y plantillas que implementó Carlos Caminero en la versión anterior del ejercicio .

Como se menciona en la sección 3.9, todos los ejercicios incluyen tres ficheros fundamentales: *hal.py*, *gui.py* y *exercise.py*, cada uno destinado a una tarea específica. En este ejercicio, tomando de referencia las plantillas del ejercicio *Follow Line*, se emplea uno extra llamado *brain.py* y las funciones expuestas a los usuarios se implementan en el fichero *user_functions.py*. Comenzaremos explicando archivos auxiliares que complementan las funcionalidades de estas plantillas principales.

El ejercicio emplea *Python Multiprocessing* para gestionar las tareas paralelas de cada plantilla de manera paralela en diferentes procesos. Para la transferencia de información entre los procesos se utilizan variables de memoria compartida. Estas variables se tratan de objetos de archivos mapeados en memoria (*mmap*), es decir, segmentos de memoria compartida y utiliza semáforos para evitar las condiciones de carrera. En un subdirectorio, llamado *shared*, se encuentran los archivos con las clases dedicada a la administración de dichas variables. Todas las clases tienen los mismos métodos:

- constructor **__init__(self, name)**: recibe el nombre para la memoria compartida, inicializa la memoria compartida y crea un semáforo.
- **get(self)**: bloquea el semáforo, lee el valor almacenado en la memoria compartida, libera el semaforo y retorna el valor.
- **add(self, value)**: bloquea el semáforo, actualiza la memoria compartida por el valor recibido y libera el semáforo.
- **close(self)**: bloquea el semáforo, desvincula la memoria, libera y cierra el semáforo dejando libres los recursos.

En concreto, manejamos cuatro tipos de variables, dos de las cuales fueron copiadas del ejercicio *Follow Line* (*sharedValue* y *sharedImage*) y dos diseñadas específicamente para este ejercicio (*sharedLaserData* y *sharedPose3D*):

- sharedValue: pensado para gestionar un valor float como puede ser la velocidad lineal del robot
- sharedImage: compuesto por una tupla de 4 canales pensado para almacenar imágenes.
- sharedLaserData: es parecida a *sharedValue*, pero el objeto *mmap* que se crea ahora tiene un tamaño de 1440 bytes para guardar las 360 medidas del lidar.
- sharedPose3D: creado con la intención de almacenar los valores x, y e yaw del robot, por lo que el tamaño del objeto es 12 bytes.

fichero	nombre del objeto	tipo
value.py	sharedValue	float
image.py	sharedImage	[int32, int32, int32, int64]
laserdata.py	sharedLaserData	float
pose3d.py	sharedPose3D	float

Tabla 4.1: Listado de plantillas shared utilizadas en el ejercicio Follow Person

Por otro lado, se encuentran las plantillas de las interfaces ROS utilizadas en *hal.py*. En cuanto a estas plantillas, hemos aprovechado el trabajo que hizo Carlos Caminero. En su proyecto, diseñó una interfaz llamada *ssd_detection.py*, la cual emplea una red neuronal convolucional para la detección de objetos en imágenes. Específicamente, creó una clase denominada *NeuralNetwork*, que utiliza la función *cv2.dnn.readNetFromTensorFlow(model, config)* de la librería Visión Artificial OpenCV junto con los ficheros de configuración *ssd_inception_v2_coco.pb* y *ssd_inception_v2_coco.pbtxt* para definir una red neuronal Inception. Además, desarrolló una interfaz *bounding_boxes.py* para establecer marcos de detección de los objetos. La lista de objetos que la red neuronal puede detectar se encuentra en el fichero *coco_labels.py*. A parte de estas interfaces, también tenemos:

- camera.py, para la cámara.
- laser.py, para el lidar.
- motors.py, para los motores del robot.
- pose3d.py, para la odometría.

Una vez listos todos los archivos auxiliares, empezamos a desarrollar los ficheros nucleares. En el archivo *hal.py* creamos la clase HAL y en su constructor inicializamos cinco variables de memoria compartida para almacenar la velocidad lineal, la velocidad angular, la imagen capturada por la cámara, las medidas tomadas por el lidar y la odometría del robot. Luego, establecemos los nodos de suscripción y publicación de cada *driver*.

```
def __init__(self):
    print("HAL initializing", flush=True)
    rclpy.init(args=sys.argv)
    rclpy.create_node('HAL')

    # Shared memory variables
    self.shared_image = SharedImage("halimage")
    self.shared_v = SharedValue("velocity")
    self.shared_w = SharedValue("angular")
    self.shared_laserdata = SharedLaserData("laserdata")
    self.shared_pose = SharedPose3D("pose")

    # ROS Topics
    self.motors = PublisherMotors("/cmd_vel", 4, 0.3)
    self.camera = ListenerCamera("/depth_camera/image_raw")
    self.laser = ListenerLaser("/scan")
    self.odometry = ListenerPose3d("/odom")

    self.start_time = 0

    # Update thread
    self.thread = ThreadHAL(self.update_hal)
    print("HAL initialized", flush=True)
```

Código 4.1: Constructor del módulo HAL en el ejercicio Sigue-Persona

El siguiente paso es crear las funciones para leer los valores de los sensores y comandación a los actuadores:

- **setV(self)**: obtiene el valor almacenado en la variable de memoria compartida *velocity* y lo comanda a los motores del robot a través de la interfaz *motors* que publica sobre el tópico */cmd_vel*.
- **setW(self)**: obtiene el valor almacenado en la variable de memoria compartida *angular* y lo comanda a los motores del robot a través de la interfaz *motors* que publica sobre el tópico */cmd_vel*.
- **getLaserData(self)**: obtiene una lista de 360 lecturas a través de la interfaz *laser* suscrita al tópico *scan* y la almacena en la variable de memoria compartida *laserdata*.
- **getPose3d(self)**: obtiene la posición actual del robot a través de la interfaz *pose3d* suscrita al tópico *odom* y la almacena en la variable de memoria compartida *pose*.
- **getImage(self)**: obtiene la imagen capturada por la cámara del robot a través de la interfaz *pose3d* suscrita al tópico */depth_camera/image_raw* y la almacena en la variable de memoria compartida *halimage*.

- **update_hal(self)**: llama continuamente a las funciones *getLaserData()*, *getImage()*, *setV()*, *setW()*, *getPose3d()* para que estén actualizando los valores de las variables.
- **start_thread(self)**: inicializa el hilo del módulo HAL.

En *gui.py*, se encuentran dos clases: *GUI*, encargado de enviar los datos, como las imágenes tomadas por la cámara del robot, al cliente a través de websockets para ser mostrada en la página web; y *ProcessGUI*, encargado de gestionar la frecuencia de actualización de la GUI.

Luego, las funciones de apoyo ofrecidas al usuario (user_functions.py) son:

- Funciones HAL:
  - **sendV(self, velocity)**: almacena el valor recibido en la variable *velocity*.
  - **sendW(self, angular)**: almacena el valor recibido en la variable *angular*.
  - **getLaserData(self)**: lee el valor almacenado en la variable *laserdata* y lo retorna.
  - **getPose3d(self)**: lee el valor almacenado en la variable *posoe* y lo retorna.
  - **getImage(self)**: lee el valor almacenado en la variable *halimage* y lo retorna.
  - **getBoundingBoxes(self, img)**: realiza una detección sobre la imagen recibida llamando al modelo de la red neuronal y devuelve una lista de *bounding boxes*.
- Funciones GUI:
  - **showImage(self, image)**: en el caso de que la imagen recibida solo tenga un canal, lo remodela a tres y la almacena en la variable de memoria compartida *halimage*.

Las plantillas *exercise.py* y *brain.py* se han tomado directamente del ejercicio *Follow Line* y no han requerido ninguna modificación, por lo tanto, no se hará énfasis en explicarlas detalladamente.

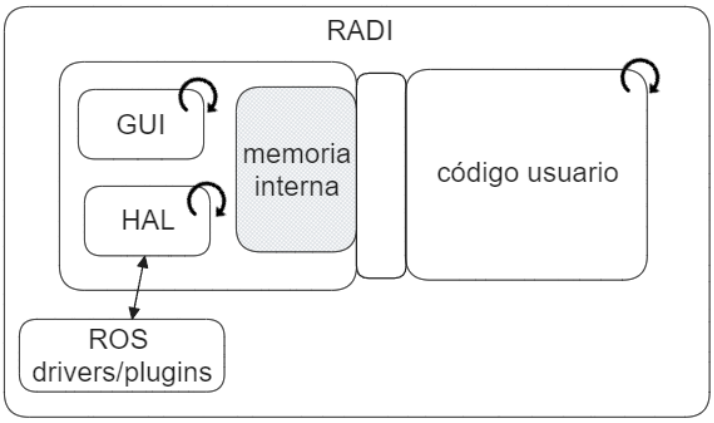


Figura 4.8: Arquitectura de funcionamiento de las plantillas Python

4.5. Solución de referencia

Las soluciones de referencia de ambas versiones del ejercicio provienen principalmente del trabajo del compañero Carlos Caminero [2]. Por lo tanto, solo haremos una explicación genérica y no detallaremos todo el código.

En el flujo principal está diseñado una máquina de estados formada por el estado buscar-persona (*SEARCH_PERSON*) y seguir-persona (*FOLLOW_PERSON*). El robot se inicializa en el estado *SEARCH_PERSON* donde gira lentamente haciendo un escaneo visual del entorno en busca de una persona. Para esto, utiliza la función *HAL.getBoundingBoxes()* que devuelve una lista de cajas de detección (*bounding boxes*), cada una con un identificador, una puntuación (*score*) que representa el porcentaje de credibilidad de la detección, y las coordenadas de las cuatro esquinas de la caja, de lo cual podremos calcular su área.

Para localizar específicamente a una persona, el robot filtra las cajas de detección con el identificador "person" que presenten una puntuación alta. Una vez que una persona es detectada, transita al estado *FOLLOW_PERSON*. En este estado, comienza el seguimiento regulando su velocidad lineal según el área de la caja de detección: si el tamaño de la caja disminuye, lo cual indica que la persona se está alejando, el robot aumenta su velocidad para reducir la distancia y mantener un seguimiento constante. Además, el robot intenta mantener la persona centrada en la imagen analizando la posición de la caja de detección, si la caja se desplaza hacia la izquierda, el robot gira en esa dirección para recentrar a la persona en su campo visual. En el caso de perder de vista a la persona volverá al estado previo y reiniciará la búsqueda.

Por otro lado, durante el seguimiento, el robot implementa el algoritmo de navegación

VFF (*Virtual Force Field*) para mantenerse alejado de los obstáculos circundantes. Este algoritmo consiste en aplicar fuerzas virtuales: la persona que el robot sigue ejerce una fuerza de atracción sobre él, incentivando al robot a acercarse, mientras que los obstáculos ejercen fuerzas repulsivas, alejando al robot de posibles colisiones. Para determinar el movimiento óptimo, se realiza una suma vectorial de estas fuerzas obteniendo una fuerza resultante, la cual se utiliza para calcular las velocidades adecuadas.

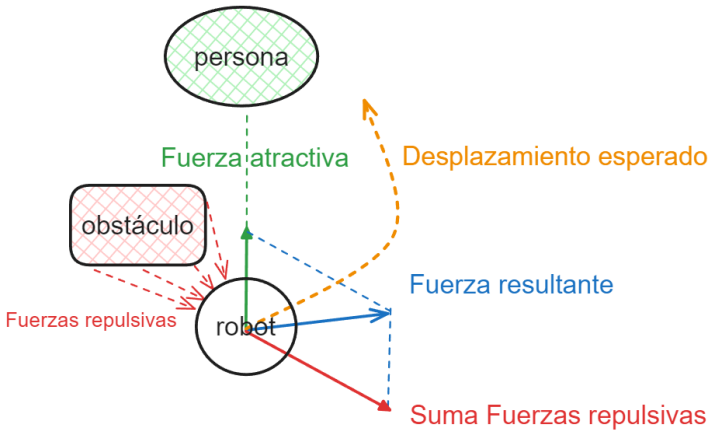


Figura 4.9: Algoritmo de navegación VFF

Sobre el código original de Carlos Caminero, hemos realizado una reimplementación de las funciones de repulsión y atracción, además de aplicar ajustes menores en algunos parámetros de velocidad. Para el cálculo de la fuerza de atracción, hemos implementado un controlador PD (Proporcional-Derivativo) para determinar el componente x del vector que se utilizará para calcular la velocidad angular.

$$u(t) = K_p \cdot e(t) + K_d \cdot \frac{de(t)}{dt}$$

Calculamos el error como la diferencia entre el centro de la imagen y el centroide de la persona, y escalado por un factor de 0,01. La derivada del error se define como la diferencia del valor del error en la iteración anterior y el valor del error de la iteración actual. Las constantes proporcional (K_p) y derivativa (K_d) están establecidas en valores de 0.5 y 0.8, respectivamente. Por último, limitamos los valores entre -0.5 y 0.5 con el fin de mantener el peso adecuado en el momento de combinarse con la fuerza repulsiva.

```
def attraction_vector(centroid_x, width_img):
    global prev_error
    width_img = width_img / 2
    error = (width_img - centroid_x) * 0.01
    kp = 0.5, kd = 0.8

    tx = error * kp + (prev_error - error) * kd
    ty = 1

    if tx < -0.5: tx = -0.5
    elif tx > 0.5: tx = 0.5

    if abs(tx) < 0.08:
        tx = 0

    return (tx, ty)
```

Código 4.2: Función para calcular el vector de atracción

En cuanto al cálculo de la fuerza de repulsión, respecto al código original, hemos añadido una condición adicional al analizar las lecturas del Lidar para que el robot solo tome en consideración los obstáculos cercanos a menos de un metro.

```
def repulsion_vector(laser_data):
    """ It returns a vector (x, y), being the sum of all repulsions vectors of
        each laser and minimized by a scale factor """
    x = y = 0
    scale_laser = 0.2
    dist_sum = 0, ang_sum = 0, counter = 0

    for dist_rep, ang in laser_data:
        if dist_rep < 1:
            dist_rep = scale_laser / dist_rep
            dist_sum += dist_rep
            ang_sum += ang
            counter += 1

    if (counter > 0):
      dist_rep = dist_sum/counter
      ang = ang_sum / counter
      x += (dist_rep**2) * math.cos(ang)
      y += -(dist_rep**2) * math.sin(ang)
    return (x, y)
```

Código 4.3: Función para calcular el vector de repulsión

Finalmente, se han realizado ajustes en la determinación de la velocidad lineal, incluyendo velocidades negativas para que el robot retroceda cuando la persona se acerca demasiado y mantener constantemente una distancia de seguridad adecuada. Los valores establecidos para el robot real y simulado son ligeramente diferentes:

- Con el robot simulado:
  - Si el área del *bounding box* es menor de 16000, la velocidad lineal es 0.2 m/s.
  - Si el área del *bounding box* se encuentra entre 16000 y 25000, la velocidad lineal es 0.1 m/s.
  - Si el área del *bounding box* está fuera de los rangos anteriores, la velocidad lineal es nula.
- Con el robot real:
  - Si el área del *bounding box* es menor de 16000, la velocidad lineal es 0.2 m/s.
  - Si el área del *bounding box* se encuentra entre 16000 y 80000, la velocidad lineal es 0.1 m/s.
  - Si el área del *bounding box* es mayor que 150000, la velocidad lineal es -0.1 m/s.

- Si el área del *bounding box* está fuera de los rangos anteriores, la velocidad lineal es nula.

El vídeo demostrativo de la solución de referencia en el universo simulado se puede encontrar en el siguiente enlace: `https://www.youtube.com/watch?v=eSwLHWgYMsY`

El vídeo demostrativo de la solución de referencia con el robot real se puede encontrar en el siguiente enlace: `https://www.youtube.com/watch?v=VeRk_F9IsDM`

4.6. Migración a la versión 5.2 de Robotics Academy

La migración del ejercicio a la versión 5.2 no ha sido el foco principal de este trabajo, pero hemos participado en afinar algunos detalles y realizar el betatesting. A continuación proporcionaremos una breve explicación de las actualizaciones realizadas sobre el ejercicio.

4.6.1. Persona teleoperada

En la migración del ejercicio a la versión 5.2 de RoboticsAcademy, se modificó el protocolo de comunicación. Se eliminó el websocket adicional que se utilizaba y se consolidaron los mensajes de teclado a través del puerto 2303. Para conseguir este comportamiento, diferenció entre persona autónoma y persona teleoperable creando un nuevo modelo denominado *PersonToControl*. La diferencia entre estos modelos es que el nuevo sustituye el plugin *person.cpp* por un plugin oficial de ROS tipo *libgazebo_ros_state.so*. Y la opción de control teleoperado en PersonToFollow se deja obsoleta.

Ahora, *TeleopLogic.js* envía las órdenes recibidas por teclado al proceso GUI, que actúa como cliente de los servicios ROS *get_entity_state* y *set_entity_state*, por el puerto 2303. En GUI.py, después de recibir el comando de movimiento, se calculan las nuevas coordenadas y se actualiza el modelo simulado mediante estos servicios.

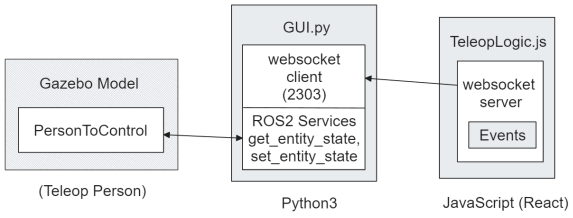


Figura 4.10: Arquitectura de comunicación de la persona teleoperada en Robotics Academy v5.2

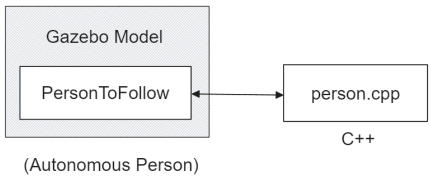


Figura 4.11: Arquitectura de comunicación de la persona autónoma en Robotics Academy v5.2

No obstante, en este nuevo diseño, inicialmente el modelo no tenía cambios de orientación y solo contaba con desplazamientos lineales en el eje x y eje y sobre el suelo. Las teclas `AD` controlaban el movimiento de la persona a lo largo del eje x, mientras que las teclas `WS` lo hacían a lo largo del eje y. Para lograr un movimiento más natural, decidimos añadir la capacidad de rotación.

En el código, definimos la distancia a mover por cada pulsación de tecla como *mov_dist* y el ángulo como *rot_angle*. Luego, al pulsar `AD` se actualiza la orientación, definido en cuaterniones (w, x, y, z), aplicando las siguientes fórmulas:

$$w' = w \cdot \cos\left(\frac{rot\_angle}{2}\right) - z \cdot \sin\left(\frac{rot\_angle}{2}\right)$$

$$x' = x \cdot \cos\left(\frac{rot\_angle}{2}\right) + y \cdot \sin\left(\frac{rot\_angle}{2}\right)$$

$$y' = y \cdot \cos\left(\frac{rot\_angle}{2}\right) - x \cdot \sin\left(\frac{rot\_angle}{2}\right)$$

$$z' = w \cdot \sin\left(\frac{rot\_angle}{2}\right) + z \cdot \cos\left(\frac{rot\_angle}{2}\right)$$

Y para actualizar las coordenadas (x, y) utilizamos:

$$yaw = \text{atan2}(2 \cdot (wz - xy), 1 - 2 \cdot (y^2 + z^2))$$

$$new\_x = old\_x + mov\_dist \cdot \sin(yaw)$$

$$new\_y = old\_y - mov\_dist \cdot \cos(yaw)$$

```
def gui_in_thread(self, ws, message):
    # In this case, messages can be either acks or key strokes
    if "ack" in message:
        [...]
    else:
        # Get the current pose
        [...]

        # Define movement and rotation parameters
        mov_dist = 0.1 # meters (default for forward movement)
        rot_angle = 0.17 # radians (default for left rotation)

        # Check for movement direction
        if  "key_s" in message: mov_dist *= -1 # reverse for backward movement
        if "key_d" in message: rot_angle *= -1 # reverse for right rotation

        # Update accordingly
        if "key_w" in message or "key_s" in message: # forward or backward
            movement
            siny_cosp = 2 * (pose.orientation.w * pose.orientation.z -
                pose.orientation.x * pose.orientation.y)
            cosy_cosp = 1 - 2 * (pose.orientation.y * pose.orientation.y +
                pose.orientation.z * pose.orientation.z)
            yaw = atan2(siny_cosp, cosy_cosp)
            pose.position.x += mov_dist * sin(yaw)
            pose.position.y += -mov_dist * cos(yaw)
        elif "key_a" in message or "key_d" in message: # turning movement
            w = pose.orientation.w * cos(rot_angle / 2) - pose.orientation.z *
                sin(rot_angle / 2)
            x = pose.orientation.x * cos(rot_angle / 2) + pose.orientation.y *
                sin(rot_angle / 2)
            y = pose.orientation.y * cos(rot_angle / 2) - pose.orientation.x *
                sin(rot_angle / 2)
            z = pose.orientation.w * sin(rot_angle / 2) + pose.orientation.z *
                cos(rot_angle / 2)
            pose.orientation.w, pose.orientation.x, pose.orientation.y,
                pose.orientation.z = w, x, y, z

        # Send the new pose
        [...]
```

Código 4.4: Función para el procesamiento de datos entrantes de la GUI

4.6.2. Plantillas Python renovadas

Por un lado, las funciones que anteriormente realizaban las plantillas exercise.py y brain.py han sido encapsuladas en el trabajo de RAM. Por otro lado, en los módulos HAL y GUI, se ha eliminado el uso de memoria compartida, la administracción de procesos ha cambiado de utilizar *Python Multiprocessing* a *Python Threading*, y las funciones de usuario son implementadas directamente en los ficheros *HAL.py* y *GUI.py*.

Capítulo 5
Ejercicio Almacén-Amazon en Robotics Academy

En este capítulo se describirá en detalle el proceso de elaboración del ejercicio Almacén-Amazon para la plataforma Robotics Academy. Se explicarán la infraestructura, las plantillas backend y frontend que dan soporte al ejercicio, así como la solución de referencia.

5.1. Enunciado

El objetivo para los estudiantes o usuarios de este ejercicio es implementar la lógica que permite a un robot logístico transportar estanterías de un lugar a otro utilizando la odometría o autolocalización fiable para su navegación. El robot dispone en su parte superior de una plataforma que se puede elevar y bajar. El proceso requiere que el robot realice varias tareas específicas: aproximarse y posicionarse debajo de la estantería, levantarla, trasladarla hasta un área de descarga designada y depositarla en su ubicación final. El principal desafío radica en determinar el camino más eficiente para completar esta secuencia de tareas. Aunque la tarea subyacente es un problema clásico de planificación de navegación, el enfoque educativo de este ejercicio no está en implementar un algoritmo de navegación desde cero. En cambio, el reto se centra en que los estudiantes aprendan a utilizar eficazmente la librería existente OMPL (sección 3.7), que es software libre, para alcanzar el objetivo propuesto.

5.2. Infraestructura

En este apartado se introducirán los distintos elementos requeridos para el ejercicio Almacén-Amazon, enfocandonos en los objetos simulados, como el mundo, los robots y sus correspondientes *drivers*. Resumiendo, contamos con cuatro elementos clave: dos almacenes (pequeño y grande) y dos tipos de robots (holonómico y Ackermann), los cuales se combinan de manera que forman cuatro universos a elegir por el usuario: almacén pequeño con robot holonómico, almacén grande con robot holonómico, almacén pequeño con robot Ackermann, y almacén grande con robot Ackermann. Los cuatro universos están nombrados

como *World1*, *World2*, *World3* y *World4* respectivamente.

5.2.1. Entorno simulado de dos almacenes

Los almacenes utilizados en este ejercicio son mundos diseñados por otros contribuyentes de Robotics Academy incluidos en el repositorio RoboticsInfraestructure:

- Almacén pequeño: con una dimensión de 20,62 x 13,6 metros, incluye objetos estáticos como bloques de cajas, estanterías fijas, contenedores, etc., y 6 estanterías móviles para ser transportadas por el robot logístico. Los objetos estáticos utilizan figuras DAE, mientras que las estanterías móviles están modeladas en formato SDF.

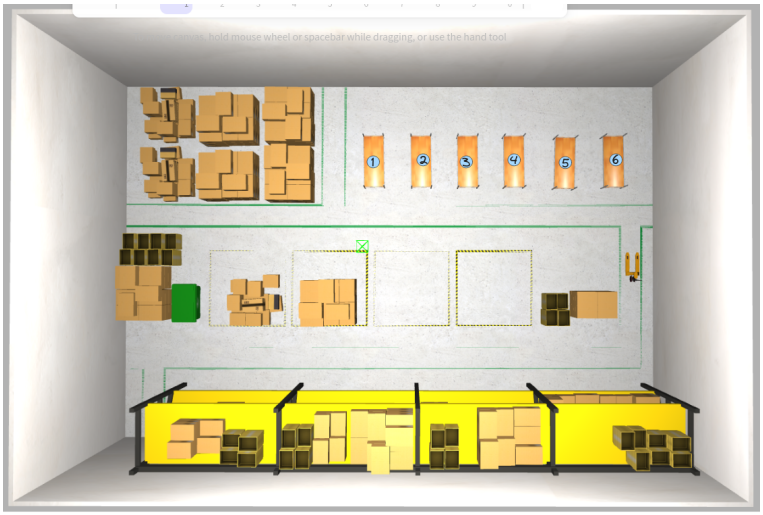
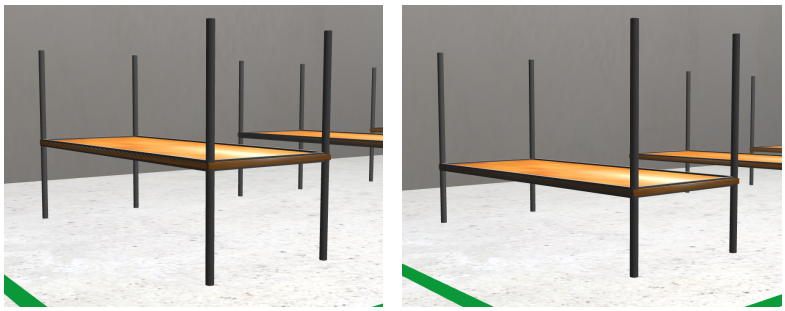


Figura 5.1: Almacén pequeño

Debido a la diferencia en propiedades y altura de carga entre los dos robots, se reajustó y adaptó el diseño de las estanterías móviles reduciendo su altura y aumentando la fuerza de fricción de la superficie que contacta con el robot. Los cambios se han aplicado directamente en el fichero SDF de las estanterías modificando la valor de z de posición del estante.



(a) Modelo original (b) Modelo ajustado

Figura 5.2: Diseños de la estantería del almacén pequeño

- Almacén grande: este almacén, como su nombre indica, tiene un tamaño mayor. Tiene una dimensión de 34 x 22 metros y cuenta exclusivamente con estanterías móviles, sumando un total de 44.

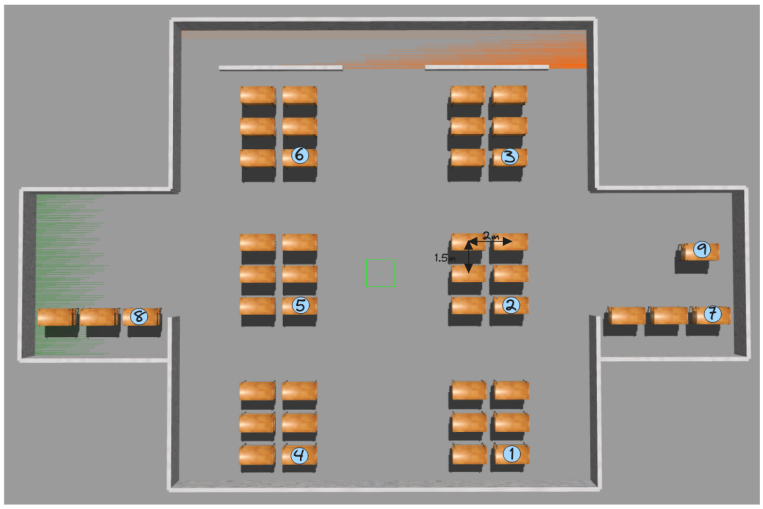
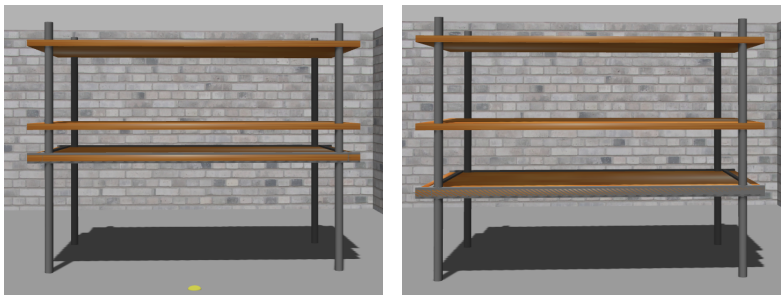


Figura 5.3: Almacén grande

Al igual que en el Almacén pequeño, se modificó la altura del último estante de estas estanterías y su fuerza de fricción para asegurar su compatibilidad con ambos tipos de robots.



(a) Modelo original (b) Modelo ajustado

Figura 5.4: Diseños de la estantería del almacén grande

5.2.2. Robot logístico holonómico

El robot logístico holonómico utilizado en el ejercicio proviene de un modelo integrado por otros contribuyentes de JdeRobot del repositorio RoboticsInfraestructure. Ha sido necesario realizar varias modificaciones y mejoras para adaptarlo a las necesidades específicas de nuestro ejercicio.

El modelo original contaba con una base, dos ruedas diferenciales, una plataforma elevadora, un Lidar y una cámara. Los ficheros necesarios para la simulación del robot se encuentran en la carpeta *CustomRobots/amazon_robot/models/amazon_robot* del repositorio. Esta incluye el directorio *meshes* con las modelos DAE de cada pieza del robot, un fichero SDF y un fichero de configuración.

```
amazon robot
├── meshes
│   ├── cam.dae
│   ├── fixed_support.dae
│   ├── lds.dae
│   ├── left_tire.dae
│   ├── platform2.dae
│   ├── platform.dae
│   ├── right_tire.dae
│   └── waffle_pi_base.dae
├── model.config
└── model.sdf
```

Figura 5.5: Estructura jerárquica del directorio *amazon_robot*

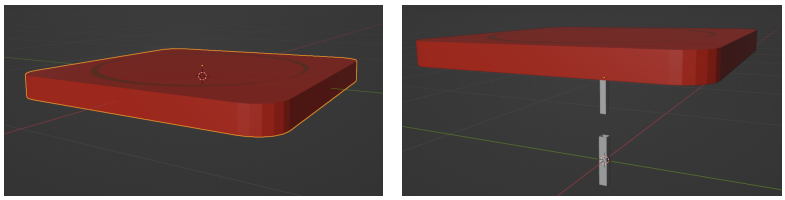
En cuanto a la simulación de movimientos, las ruedas emplean articulaciones tipo *revolute* para realizar movimientos giratorios sobre el eje x, mientras que la plataforma elevadora utiliza una articulación *prismatic* para realizar movimientos verticales. Para

el control de los actuadores, se utiliza el plugin *libgazebo_ros_diff_drive.so*, que ofrece un tópico llamado */amazon_robot/cmd_vel* para comandar velocidades lineales y angulares a los motores. Y para el control de la plataforma se gestiona recibiendo valores de fuerzas en la articulación, para ello se utiliza un publicador recogido como plantilla Python del ejercicio que se explicará de forma más detallada en la sección 5.4.

Sin embargo, el modelo original presentaba algunas deficiencias que afectaba su realismo y funcionalidad. Entre estas, la plataforma se desprendía visualmente de su soporte al elevarse a cierta altura, discrepancias entre las mallas de colisión y visual, deslizamientos de la estantería cargada y un centro de masa inestable particularmente durante las aceleraciones y frenadas.

Para mejorar el diseño y la funcionalidad del robot se implementaron los siguientes cambios:

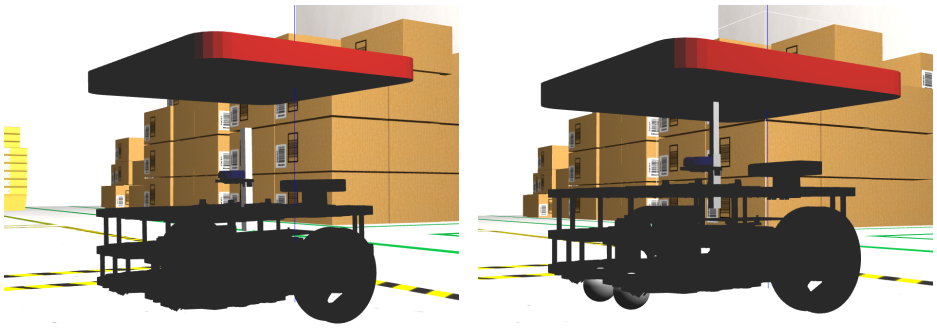
- Desactivación de sensores lidar y cámara ya que no se hace uso de ellos en este ejercicio.
- Rediseño del modelado de la plataforma elevadora para visualización más realista. Como se puede observar, el diseño original solo cuenta con la tableta de la plataforma, el ortoedro que la sostiene es una figura definida en XML. En el nuevo diseño se ha añadido dos componentes: una barra vinculada a la plataforma y un tubo adherido a la base, mejorando así la coherencia del movimiento en la simulación.



(a) Modelo original (b) Modelo ajustado

Figura 5.6: Diseños de la plataforma del robot logístico holonónimo

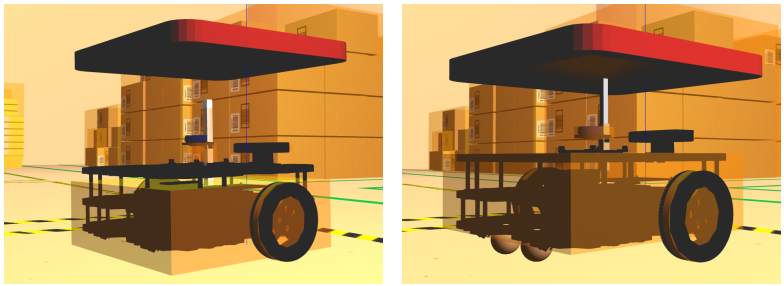
- Adición de ruedas de apoyo para mejorar la distribución del peso y solucionar el problema del centro de masa inestable.



(a) Modelo original (b) Modelo ajustado

Figura 5.7: Diseños del robot logístico holonómico

- Ajuste en la malla de colisión para que coincida con la visual.



(a) Modelo original (b) Modelo ajustado

Figura 5.8: Diseños del robot logístico holonómico visualizando la malla de colisión

- Incremento de la fuerza de fricción entre la superficie de la plataforma y las estanterías para un transporte más seguro.

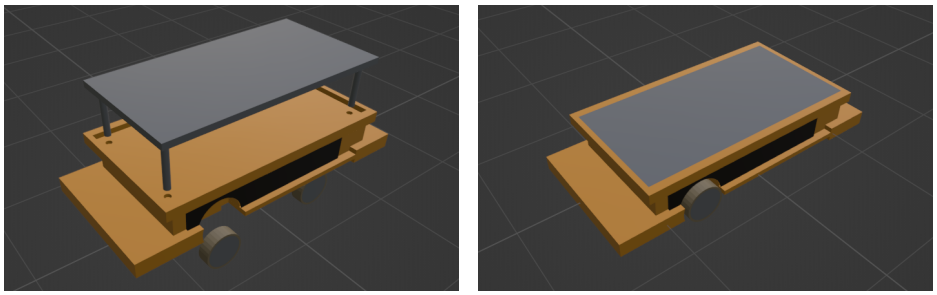
5.2.3. Robot logístico Ackermann

El robot logístico Ackermann se diseñó y modeló desde cero, inspirándonos en el robot logístico de BMW descrito en la sección 1.1.2. Elegimos este robot por varias razones:

- Geometría rectangular, algo que buscabamos para añadir complejidad al enunciado del ejercicio y requerir soluciones más diversas.
- Equipado con una plataforma elevadora para cargar, transportar y descargar mercancías.

- Posibilidad de implementar el movimiento Ackermann.

El robot está modelado completamente en Blender y consta de tres partes principales: la base o cuerpo del robot, cuatro ruedas y la plataforma elevadora nombrada como *chassis.dae*, *wheel.dae* y *platform.dae* respectivamente.



(a) Desmontado (b) Montado

Figura 5.9: Modelado del robot Ackermann en Blender

Una vez diseñada la parte mecánica del robot, procedimos a crear el fichero SDF para su simulación en Gazebo. Hemos tomado como referencia el modelo del robot holonómico y el modelo *autocar* del ejercicio *Autoparking*.

Comenzamos definiendo los elementos estructurales: *chassis_link* para el cuerpo; *bl_1* y *br_1* para las ruedas traseras ("l" indica izquierda y "r" derecha); *fl_1* y *fr_1* para las ruedas delanteras, *l_steer_1*, *r_steer_1* para los mecanismos de dirección; y *lift_link* para la plataforma elevadora. Para cada componente añadimos su malla visual y de colisión especificando la URI de los archivos DAE y definimos la inercia. Ajustamos el posicionamiento de cada parte para un ensamblaje preciso y hemos tenido que escalar las piezas a 0.5 para adaptarlo al tamaño del mundo en la simulación.

En segundo lugar, definimos los *joints* correspondientes a cada *link* que son los mismos que se utilizan en el robot logístico holonómico. La única diferencia reside en los *joints* de las ruedas delanteras del robot, que además del giro sobre el eje x, también giran sobre el eje z para simular el movimiento Ackermann.

Por último, añadimos los *drivers* a los motores y la plataforma para que reaccionen a las órdenes de movimiento. Tomando como ejemplo el modelo *autocar*, hemos integrado fácilmente el plugin *libgazebo_ros_ackermann_drive.so* para comandar velocidades lineales y angulares, o dicho con más precisión, el ángulo de giro de las ruedas. En cuanto a la plataforma, se utiliza el mismo publicador que en el robot holonómico.

```
<link name='chassis_link'>
    <inertial>
      <pose>1e-06 0.07307 0.65096 0 -0 0</pose>
      <mass>1580</mass>
      <inertia>
        [...]
      </inertia>
    </inertial>
    <collision name='chassis_link_fixed_joint_lump__chassis_link_collision'>
      <pose>0 0 0.0 0 -0 0</pose>
      <geometry>
        <mesh>
          <scale>0.5 0.5 0.5</scale>
          <uri>model://ackermann_logistic_robot/meshes/chassis.dae</uri>
        </mesh>
      </geometry>
      <surface>
        [...]
        <friction>
          <ode>
            <mu>0.2</mu>
            <mu2>0.2</mu2>
          </ode>
        </friction>
      </surface>
    </collision>
    <visual name='chassis_link_fixed_joint_lump__chassis_link_visual'>
      <pose>0 0 0.0 0 -0 0</pose>
      <geometry>
        <mesh>
          <scale>0.5 0.5 0.5</scale>
          <uri>model://ackermann_logistic_robot/meshes/chassis.dae</uri>
        </mesh>
      </geometry>
    </visual>
    <self_collide>1</self_collide>
  </link>
```

Código 5.1: Ejemplo del código XML del cuerpo del robot Ackermann

El fichero SDF del robot logístico Ackermann se encuentra ubicado en el repositorio RoboticsInfraestructure dentro de la carpeta *CustomRobots/ackermann_logistic_robot*. Dicha carpeta contiene también el directorio *meshes* con los ficheros DAE y el fichero de configuración *model.config*.

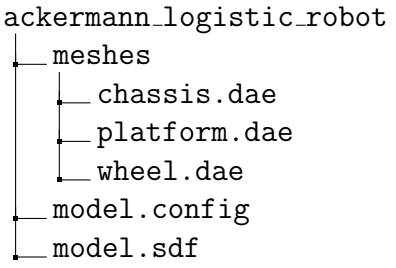
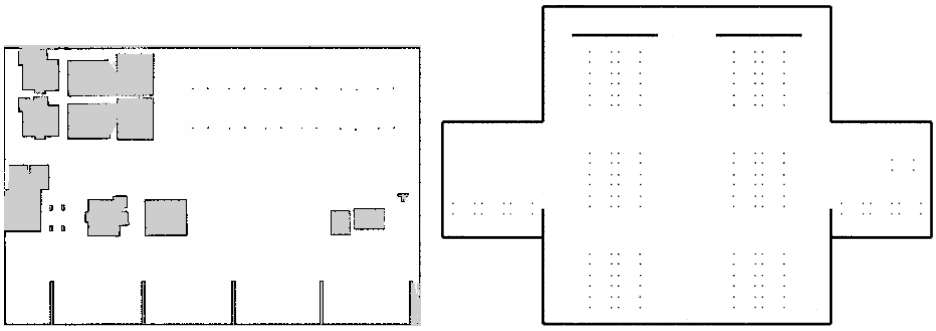


Figura 5.10: Estructura jerárquica del directorio *ackermann_logistic_robot*

5.3. Página web

En la página web, además de los componentes básicos (editor de texto, consola de depuración y simulador Gazebo), también se ofrecen los mapas 2D de los almacenes. En estos mapas, los pixeles negros representan zonas ocupadas, los pixeles blancos indican zonas libres y los pixeles grises zonas no exploradas.



(a) Almacén pequeño (b) Almacén grande

Figura 5.11: Mapas 2D de los almacenes

En el mapa que se visualiza en la página web, el robot se representa como un triángulo rojo y su trayectoria recorrida se muestra en color azul. Además, los usuarios pueden utilizar la función *GUI.showPath()* para dibujar manualmente una ruta sobre el mapa, el cual se pinta de color verde.

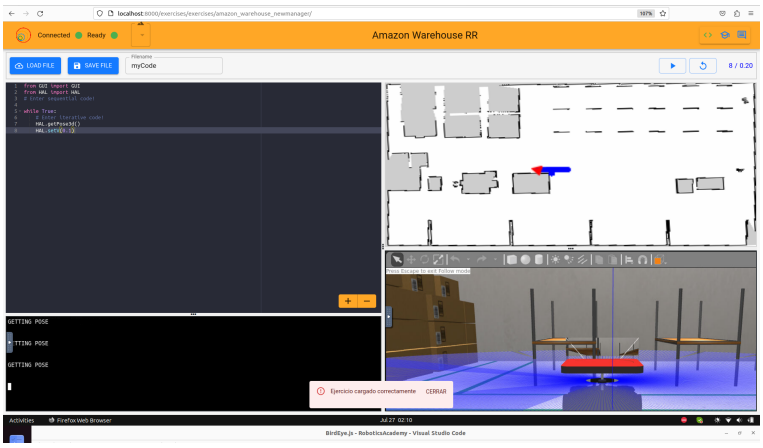


Figura 5.12: Página Web del ejercicio Almacén-Amazon

En cuanto al *software*, los componentes React específicos de este ejercicio son:

```
react-components
├── css
│   └── AmazonWarehouseRR.css
├── helpers
│   └── bird_eye_amazon_-
│       warehouse.js
├── AmazonWarehouseRR.js
├── SpecificAmazonWarehouse.js
└── WorldSelectorAmazonWarehouse.js
```

Figura 5.13: Estructura jerárquica del directorio *react_components* del ejercicio Almacén-Amazon

A continuación, se detalla la utilidad de cada archivo:

- **AmazonWarehouseRR.css**: hoja de estilo del ejercicio.
- **AmazonWarehouseRR.js**: componente principal que importa y aplica la hoja de estilo del ejercicio a la página web.
- **WorldSelectorAmazonWarehouse.js**: componente selector que permite al usuario cambiar de universo robótico.
- **SpecificAmazonWarehouse.js**: componenete encargado de gestionar la visualización gráfica del mapa 2D y las acciones de dibujo sobre él.

- **bird_eye_amazon_warehouse.js**: componente auxiliar que contiene funciones de ayuda para dibujar figuras en el mapa.

Para implementar los archivos *SpecificAmazonWarehouse.js* y *bird_eye_amazon_warehouse.js* hemos tomado como referencia los ejercicios *Basic Vacuum Cleaner* y *Global Navigation*. La lógica interna de estos componentes implica definir el mapa del almacén como imagen fondo del elemento canvas y utilizar métodos canvas como *beginPath()*, *closePath()*, *moveTo()*, *lineTo()*, *arc()*, etc. para dibujar figuras que representan diferentes objetos en el mapa. Por ejemplo, el código para dibujar una círculo azul sería el siguiente:

```
function drawCircle(x, y) {
  ctx.beginPath();
  ctx.arc(x, y, 1.5, 0, 2 * Math.PI);
  ctx.closePath();

  ctx.lineWidth = 1.5;
  ctx.strokeStyle = "#0000FF";
  ctx.stroke();

  ctx.fillStyle = "#0000FF";
  ctx.fill();
}
```

Código 5.2: Función para dibujar un circulo en canvas

En el fichero *bird_eye_amazon_warehouse.js* se encuentran implementadas las funciones auxiliares de dibujo:

- **drawCircle(x, y)**: dibuja un círculo azul en la posición indicada.
- **drawTriangle(posx, posy, angx, angy)**: dibuja un triangulo rojo basado en la posición y orientación indicada.
- **drawTrial(px, py)**: guarda las coordenadas recibidas en la variable *trail*, que es una lista de posiciones anteriores recorridas por el robot. Luego, recorre toda la lista y pinta la ruta completa como una serie de círculos.
- **draw(canvas, x, y, ax, ay)**: dibuja el robot y su recorrido histórico en el mapa. Utiliza la función *drawTriangle(posx, posy, angx, angy)* para representar el robot y *drawTrial(px, py)* para mostrar la ruta.
- **generatePath(data, canvas)**: recibe una lista de coordenadas y representa la ruta como tramos rectos que conectan cada par de coordenadas consecutivas.
- **clearMap()**: limpia el canvas.

El componente *SpecificAmazonWarehouse* está diseñdo para procesar dos tipo de información: *map data* y *array data*. El *map data* contiene las coordenadas (x, y) y la orientación del robot, cuado el componente recibe este tipo de datos llama la función *draw()* para actualizar la posición del robot en el mapa. Por otro lado, *array data* se trata de una lista bidimensional destinada a guardas coordenadas *(x, y)* que representan una ruta, cuando se recibe este tipo de datos se llama a la función *generatePath()*.

5.4. Plantillas Python

Este ejercicio consta de los tres ficheros principales *hal.py*, *gui.py*, *exercise.py* y uno adicional *map.py* que es un fichero auxiliar diseñado para ser usado junto con *gui.py*. Al principio, al igual que el ejercicio Sigue-Persona, se utilizó *Python Multiprocessing* para la gestión de procesos empleando memoria compartida. Sin embargo, por razones de simplificación, se cambió a usar *Python Threading* y para compartir datos entre el módulo HAL y GUI se utiliza variable compartida definiendo la clase *HAL* como variable dentro del módulo GUI.

En este ejercicio, el módulo HAL ofrece a los usuarios seis funciones:

- **getPose3d()**: para obtener la odometría (x, y, yaw) del robot.
- **getSimTime()**: para obtener el tiempo de simulación, en segundos (*sec*) y nanosegundos (*nanosec*).
- **setV()**: para comandar velocidad lineal.
- **setW()**: para comandar velocidad angular, en el caso del robot Ackermann indica el angulo de giro de las ruedas.
- **lift()**: para elevar la plataforma.
- **putdown()**: para bajar la plataforma.

Los ficheros de interfaces ROS utilizados en el módulo HAL son:

- **motors.py**, para los motores del robot.
- **pose3d.py**, para la odometría.
- **platform_controller.py** y **platform_publisher.py**, para la plataforma.

  En *platform_publisher.py* se encuentra el nodo publicador encargado de recibir las ordenes de usuario, traducirlas a cadenas de texto “lift” o “putdown” y publicarlas en el tópico correspondiente. En *platform_controller.py*, por un lado se encuentra el nodo suscriptor encargado de leer los mensajes del tópico, convertir las cadenas de texto a valores númericos que indica la fuerza que debe ejercer la plataforma, y por otro lado está el cliente del servicio *apply_joint_effort* que se encarga de mandar las fuerzas a la articulación *lift_joint*.

- **simtime.py**, para el tiempo de simulación.

  Durante la simulación, el simulador Gazebo publica su tiempo de simulación en el tópico */clock*. Los mensajes publicados son del tipo *rosgraph_msgs/msgs/Clock* que contiene una única variable *clock* con dos componentes: *sec* y *nanosec*. Para exponer esta información a los usuarios se ha creado una clase llamada *ListenerSimTime()*, en ella, se crea un nodo suscriptor al tópico */clock* y se almacena la última marca de tiempo en una variable local. Los usuarios pueden acceder a este dato haciendo llamada a la función *HAL.getSimTime()*.

En el archivo *map.py*, se ha implementado la clase *MAP* que procesa los valores de odometría del robot para su uso en el componente *SpecificAmazonWarehouse*. Concretamente, cuenta con 3 funciones:

- constructor **__init__(self, pose3d, circuit)**: inicializa *self.pose3d*, que almacena la odometría del robot; y *self.circuit*, el nombre del universo lanzado en la GUI.
- **getRobotCoordinates(self)**: convierte las coordenadas *(x, y)* del mundo Gazebo en coordenadas píxeles para representarse gráficamente en el mapa. Primero, se procesa para alinearse con el sistema de referencia de la imagen PNG y luego se escala para cuadrar en la imagen canvas. Los valores de escala varían dependiendo de las dimensiones específicas de cada almacén.

```
def getRobotCoordinates(self):
    pose = self.pose3d.getPose3d()
    x = pose.x
    y = pose.y

    if self.circuit == "World2" or self.circuit == "World4": # Warehouse 2
        '''
        x = (h_gazebo / 2 - x) * (h_image / h_gazebo) * (h_canvas / h_image)
        y = (w_gazebo / 2 - y) * (w_image / w_gazebo) * (w_canvas / w_image)
        '''
        x = (12.0 - x) * 29.13 * 0.22
        y = (17.0 - y) * 31.62 * 0.28

    else: # Warehouse 1 (small one)
        x = (6.8 - x) * 20.22 * 0.545
        y = (10.31 - y) * 20.17 * 0.72

    return y, x
```

Código 5.3: Función *getRobotCoordinates()*

- **getRobotAngle(self)**: calcula un vector de dirección basado en el ángulo *yaw* del robot.

El módulo GUI ofrece a los usuarios dos funciones:

- **getMap(url)**: devuelve la imagen almacenada en la URL especificada. La imagen se representa como un arreglo tridimensional de numpy, cada elemento representa un píxel en formato (R, G, B, A).
- **showPath(array)**: espera recibir una lista bidimensional de coordenadas *(x, y)* y pinta la ruta en el mapa.

Internamente, el módulo GUI maneja un diccionario llamado *payload* que almacena los datos *map* y *array*. Cuando un usuario llama a *GUI.showPath()*, la función primero escala las coordenadas proporcionadas y las convierte en una cadena de texto que se almacena en la variable *array* del diccionario. Para los datos *map*, se obtiene la odometría del robot a través del módulo HAL, y se aplican las funciones *getRobotCoordinates()* y *getRobotAngle()* del módulo MAP para adaptarse al sistema de referencia de la imagen canvas, luego los valores resultantes se convierten en una cadena de texto y se almacena en la variable *map*. Por último, los datos del diccionario son enviados a la página web a través de websockets.

5.5. Soluciones de referencia

En este apartado se explicarán detalladamente las soluciones de refencia desarrolladas para el ejercicio Almacén-Amazon. En total, hemos implementado tres soluciones de referencia, todas descompuestas en dos fases: la planificación de la ruta de navegación y la ejecución de la ruta planificada. Las dos primeras se basan en una planificación espacial probadas en el robot logístico holonómico; y la tercera en una planificación basada en control testeada en el universo con el robot logístico Ackermann.

Como se había mencionado previamente, el objetivo de este ejercicio es utilizar OMPL para la resolución del problema de planificación. Por lo tanto, primero vamos a hacer una breve mención de los pasos básicos a seguir para la creación y resolución de un plan con OMPL:

1. Crear el espacio de estados y establecer los límites, es decir, las dimensiones.
2. Instanciar el espacio de información.
3. Implementar el criterio de validación y establecer el comprobador de estados válidos
4. Crear una instancia de problema.
5. Establecer el estado inicial y estado final.
6. Establecer un planificador. En los problemas de planificación geométricos se puede especificar un objetivo de optimización y utilizar un planificador óptimo.
7. Establecer la instancia de problema.

8. Resolver el problema.

En las planificaciones basadas en controles, adicionalmente, hay que:

1. Crear el espacio de controles y establecer los límites, es decir, las velocidades máxima y mínima.

2. Implementar el criterio de propagación de los estados y establecer el propagador de estados.

5.5.1. Planificación espacial con robot holonómico y geometría "circular"

Para el caso del robot logístico holonómico se propone dos soluciones de referencia, ambas con planificación espacial. La primera es basándonos únicamente en el espacio euclídeo aproximando la geometría del robot a un circulo y la segunda es incorporando las restricciones geométricas precisas del robot con restricciones de movimiento, su morfología rectangular (explicada en la sección 5.5.2). La principal diferencia de estas dos formas cae en el espacio de estados utilizado y en la estrategia de validación de los estados válidos e inválidos.

5.5.1.1. Planificación de la ruta

La idea central de esta solución es tratar el robot como un punto 2D, es decir, un píxel en el mapa, y aplicar un proceso de dilatación sobre el mapa para extender las fronteras de los obstáculos por un margen igual al radio del robot garantizando una distancia de seguridad adecuada para evitar colisiones durante la navegación.

Es importante distinguir entre el trayecto de ida, cuando el robot va solo, y el de vuelta, cuando tiene cargada una estantería. Una vez el robot tiene cargada una estantería, ésta deja de ser un obstáculo y pasa a ser parte del robot, modificando su geometría a una forma rectangular. Este cambio implica un aumento en el 'radio" del robot, y consecuentemente en el margen necesario para la dilatación del mapa para mantener la distancia de seguridad. Además, se debe adaptar el mapa ignorando los píxeles oscuros que inicialmente representaban la estantería. Para ello, lo que hacemos es antes de realizar el proceso de dilatación, borrar la estantería del mapa rellenando los píxeles que representaban el área previamente ocupada por esta en blanco.

Todo el código relacionado con la planificación está incluida en una función llama *generatePath(initial, end)*. Esta función recibe como parámetro el estado inicial y final, intenta buscar una ruta que conecte estos dos estados y en el caso de que se haya encontrado una ruta válida, se guardan los estados que conforman la ruta en una lista llamada *path* y la retorna. En la implementación de la función *generatePath(initial, end)*, el primer paso es instanciar el espacio de estados, en este caso, trabajamos con un espacio euclídeo *(x, y)*, concretamente, *RealVectorStateSpace*. Luego, definimos su dimensión de acuerdo al largo y ancho del mapa, instanciamos el espacio de información y establecemos el comprobador

de estados válidos. El criterio de validación aplicado se basa en la examinación del color de los píxeles. Se considera como estado válido aquellos píxeles cuyo valor de cada uno de los tres canales (R, G, B) es mayor que 250. Esta parte del código está implementada en la función *isStateValid()*.

```
def isStateValid(state):
    w = min(int(state[0]), MAP_WIDTH - 1)
    h = min(int(state[1]), MAP_HEIGHT - 1)

    c = eroded_map[h, w]
    free = c[0] * 255 > 250 and c[1] * 255 > 250 and c[2] * 255 > 250

    return free
```

Código 5.4: Función *isStateValid()*

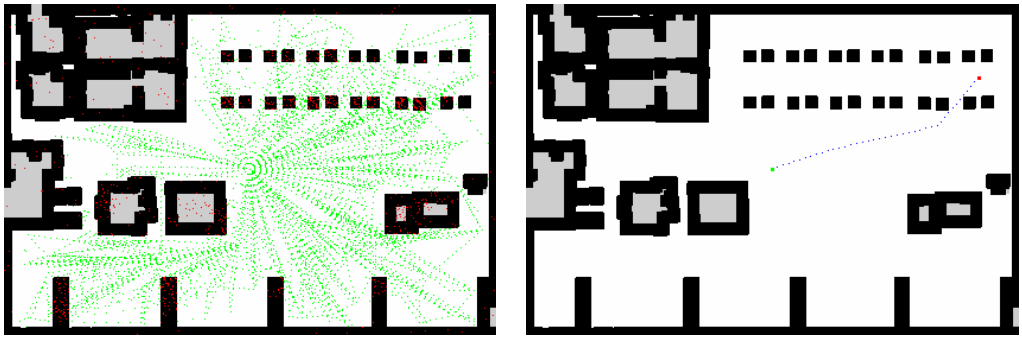
Una vez establecido el comprobador de estados válidos, el siguiente paso es definir el problema a resolver. Indicamos el estado inicial, donde el robot comienza su trayectoria que coincide con la posición actual del robot, y el estado final, el punto destino al que debe llegar. A continuación, añadimos un objetivo de optimización, en este caso, optamos por el *PathLengthOptimizationObjective* que busca minizar la longitud total de la ruta. Finalmente, seleccionamos un planificador adecuado y procedemos a resolver el problema.

Después de llevar a cabo varias experimentaciones con diferentes planificadores, no se observaron diferencias significativas en su rendimiento. Esto indicó que la elección del planificador no era una condición crítica en nuestro problema y decidimos utilizar el planificar optimizado *RRTstar*. Sin embargo, sí notamos mejoras al aumentar el tiempo de ejecución del algoritmo y tras evaluar varios intervalos, establecimos el tiempo de ejecución en 10.0 segundos, dado que con él se lograron buenos resultados.

```
def generatePath(initial, end):
    # Instace the state space and add dimensions
    space = ob.RealVectorStateSpace()
    space.addDimension(0.0, MAP_WIDTH)
    space.addDimension(0.0, MAP_HEIGHT)

    # Instace space information
    si = ob.SpaceInformation(space)
    si.setStateValidityChecker(ob.StateValidityCheckerFn(isStateValid))
    si.setup()

    # Create a problem instance
    pdef = ob.ProblemDefinition(si)

    # Set robot's starting and goal state
    start = ob.State(space)
    start[0] = initial[0]
    start[1] = initial[1]
    goal = ob.State(space)
    goal[0] = end[0]
    goal[1] = end[1]
    pdef.setStartAndGoalStates(start, goal)

    # Set optimization objective and an optimal planner
    pdef.setOptimizationObjective(ob.PathLengthOptimizationObjective(si))
    optimizingPlanner = og.RRTstar(si)

    # Set the problem instance
    optimizingPlanner.setProblemDefinition(pdef)
    optimizingPlanner.setup()

    # Attempt to solve the planning problem in the given runtime
    solved = optimizingPlanner.solve(10.0)

    # If a solution is found path list will be fill
    path_lst = []
    if solved:
        solution = pdef.getSolutionPath()
        for i in range(path.getStateCount()):
            path_lst.append([path.getState(i)[0], path.getState(i)[1]])
    else:
        print("No solution found")
    return path_lst
```

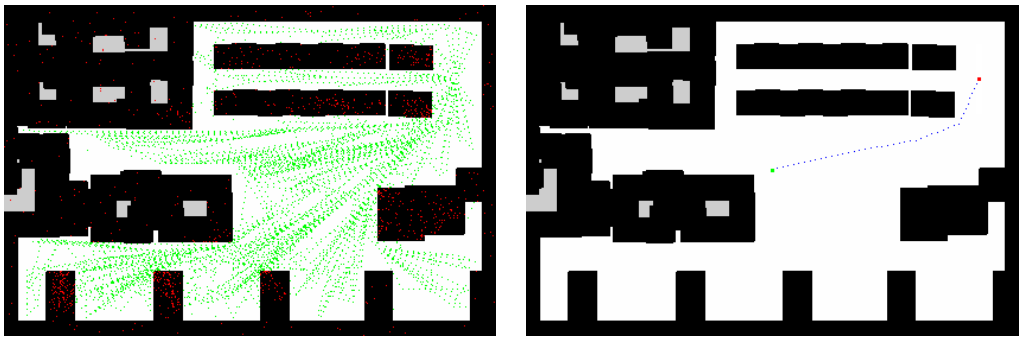
Código 5.5: Función *generatePath()*

En las figuras 5.14 y 5.15 se muestran los resultados experimentales del proceso de planificación. En las subfiguras 5.14(a) y 5.15(a), se muestran los resultados del proceso de validación de estados, el color verde indica estados válidos y el color rojo estados inválidos, y en las subfiguras 5.14(b) y 5.15(b) se muestran la solución de ruta encontrada.



(a) Proceso de validación de estados (b) Ruta válida encontrada

Figura 5.14: Planificación de la ruta para ir a la zona de carga



(a) Proceso de validación de estados (b) Ruta válida encontrada

Figura 5.15: Planificación de la ruta para volver a la zona de descarga

Los resultados obtenidos del proceso de planificación de rutas son en gran medida positivos. No obstante, es importante reconocer las limitaciones de este enfoque que podrían restringir la aplicabilidad de este sistema en ciertos contextos. La simplificación del robot como un punto 2D en el proceso de planificación facilita el cálculo y reduce la complejidad del problema, pero también omite consideraciones importantes como la orientación del robot. Un ejemplo destacado se observa cuando el robot tiene cargada una estantería y la geometría del conjunto no puede aproximarse a un simple punto. Para compensar esta inadecuación y garantizar una ruta segura, se realiza una dilatación adicional para cubrir el borde más largo de la estantería. Esto puede resultar en que se marquen estados válidos

como no válidos, lo cual distorsiona la evaluación de rutas viables y puede conducir a la selección de rutas subóptimas. Si bien esto no parece resultar un problema mayor, en entornos estrechos, como pasillos, donde el espacio navegable se reduce drásticamente y la libertad de maniobra se vuelve limitada, esta restricción puede ser un problema bloqueante. En estos casos, la dilatación "redundante" puede haber eliminado todos los estados válidos impidiendo al robot encontrar cualquier ruta viable, a pesar de que en realidad sí existiera.

Para abordar estas limitaciones, una solución más avanzada es incorporar una representación más precisa de la forma física del robot y su orientación. A cerca de esta idea, se implementó la segunda solución de referencia.

5.5.1.2. Pilotaje del robot

Una vez planificada la ruta de navegación, el siguiente paso es implementar un algoritmo responsable de ejecutar eficazmente esta ruta.

La función principal responsable de gestionar el pilotaje se llama *reachWaypoint(waypoint)*. Esta función recibe como parámetro un *waypoint*, que es el subdestino en el mapa al que el robot debe dirigirse. Se analiza la posición actual del robot respecto al *waypoint* para determinar la acción a realizar, así como la velocidad lineal y angular adecuadas para su implementación.

El robot intenta primero alinearse con el destino y luego aproximarse. Para calcular la velocidad angular, se calcula la orientación ideal que el robot debería tener para alinearse con el *waypoint*. Luego, se compara esta orientación deseada con la orientación actual del robot y se comanda una velocidad angular con la función *HAL.setW()* para minimizar la diferencia. Cuando la diferencia es menor de 0.01 rad, se considera alineado y el robot empieza a comandar una velocidad lineal constante de 0.1 m/s con *HAL.setV()* hasta llegar al *waypoint*.

El algoritmo está diseñado para ser reactivo, de modo que el robot está continuamente comprobando su alineación y corrige inmediatamente cualquier desviación.

```
def reachWaypoint(waypoint):
    robotx, roboty, robotyaw = HAL.getPose3d().x, HAL.getPose3d().y,
        HAL.getPose3d().yaw

    # Check the distance to goal
    wpx, wpy = pixels2World(waypoint[1], waypoint[0])
    reached = isClose([robotx, roboty], [wpx, wpy], 0.02)

    # Set an appropriate angular speed
    wpyaw = getYaw([robotx, roboty], [wpx, wpy])
    w = getAngularVelocity(robotyaw, wpyaw)

    # Set an appropriate lineal speed, first align and then lineal moves
    if w == 0:
        if reached: # waypoint reached
            v = 0
        else: # waypoint unreached
            v = 0.1
    else:
        v = 0

    HAL.setW(w)
    HAL.setV(v)

    return reached
```

Código 5.6: Función *isStateValid()*

A continuación, se lista el resto de funciones implementadas para completar la tarea de navegación:

- **world2Pixels(x,y)**: convierte las coordenadas del mundo Gazebo a coordenadas en píxeles del mapa.
- **pixels2World(x,y)**: convierte las coordenadas en píxeles a coordenadas del mundo Gazebo.
- **isClose(p1, p2, e)**: comprueba la proximidad de los puntos *p1* y *p2* . Se considera aproximado cuando la distancia entre los dos puntos es menor que *e*.
- **getYaw(robot_pos, goal_pos)**: devuelve la orientación que el robot tiene que tener para alinearse con el *goal_pos*.
- **getTurnDirection(robot_yaw, goal_yaw)**: calcula el sentido de rotación que el robot debe tener para completar el movimiento de la forma más eficiente.

- **getAngularVelocity(robot_yaw, goal_yaw)**: calcula la velocidad angular necesaria para alinear el robot hacia el waypoint. Se considera no alineado cuando la diferencia de orientación es mayor que 0.01 rad, en este caso se analiza hacia qué sentido debe girar y aplica una velocidad constante de 0.03 rad/s. En caso de estar alineado devuelve velocidad nula.

5.5.1.3. Máquina de estados

La realización completa de la tarea se gestiona mediante una máquina de estados finitos de 4 estados:

- *GENERATE_PATH*: estado de inicio. Se llama a la función *generatePath(initial, end)* para planificar la ruta. Según el estado de la plataforma (valor guardado en la variable *platform_state* que puede ser *NO_LIFTED* o *LIFTED*) podemos diferenciar si se trata de la trayectoria de ida o de vuelta y decidir los valores de los parámetros *initial* y *end*, además de la acción a aplicar sobre el mapa (margen de dilatación e ignoración de la estantería en el mapa). Si el estado de la plataforma es *NO_LIFTED* podemos entender que no tiene cargada la estantería y se trata del trayecto de ida, de lo contrario, se trata del trayecto de vuelta. Una vez generado la ruta pasa al estado *REACH_DESTINY*.

- *REACH_DESTINY*: con la ayuda de la función *reachWaypoint()* el robot navega de un *waypoint* a otro en orden hasta llegar al destino final y pasa al estado *LOAD_-UNLOAD_SHELF*.

- *LOAD_UNLOAD_SHELF*: si el valor de *platform_state* es *NO_LIFTED*, el robot toma la acción de *HAL.lift()*, actualiza *platform_state* a *LIFTED*, y vuelve al estado *GENERATE_PATH* para calcular la ruta de vuelta. De lo contrario, realizar la acción de *HAL.putdown()* y termina en el estado *FINISH*.

- *FINISH*: detiene el robot.

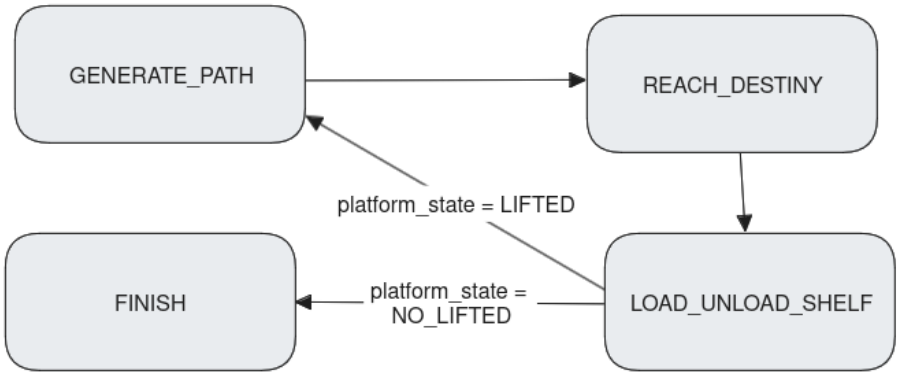


Figura 5.16: Máquina de estados finitos

El vídeo demostrativo de la solución de referencia con planificación puramente espacial con robot circular se puede encontrar en el siguiente enlace: `https://www.youtube.com/watch?v=EVt9vYqEoDg`

5.5.2. Planificación espacial con robot holonómico y geometría "rectangular"

5.5.2.1. Planificación de la ruta

La incorporación de la forma física y la orientación del robot lleva a la necesidad de utilizar un espacio de estados más complejo, específicamente el *SE2StateSpace*. En este espacio, cada estado se representa mediante la tupla *(x, y, yaw)*, donde *(x, y)* son las coordenadas de posición del robot en un plano bidimensional y *yaw* es el ángulo de orientación. Además, es esencial contar con un validador de movimiento para añadir restricciones de movimientos al robot. Este validador comprobará la viabilidad de un movimiento entre dos estados.

Entre los diversos espacios de estados que la librería ofrece, hemos encontrado un espacio que encaja perfectamente con nuestra necesidades, el espacio de estado Dubins. Este espacio está montado sobre el espacio *SE2StateSpace* y tiene incorporado un validador de movimientos que limita al robot a movimientos lineales hacia adelante y giros en arco con un radio determinado. Aunque el uso más común del espacio Dubins sea en vehículos no holonómicos, ajustando un ángulo de giro bajo, podemos configurar el espacio para que el arco de movimiento sea tan reducido que permita al robot girar prácticamente sobre sí mismo.

Con respecto al comprobador de estados válidos, ahora representamos al robot como una figura geométrica 2D compuesta por un bloque rectangular de píxeles, posicionando

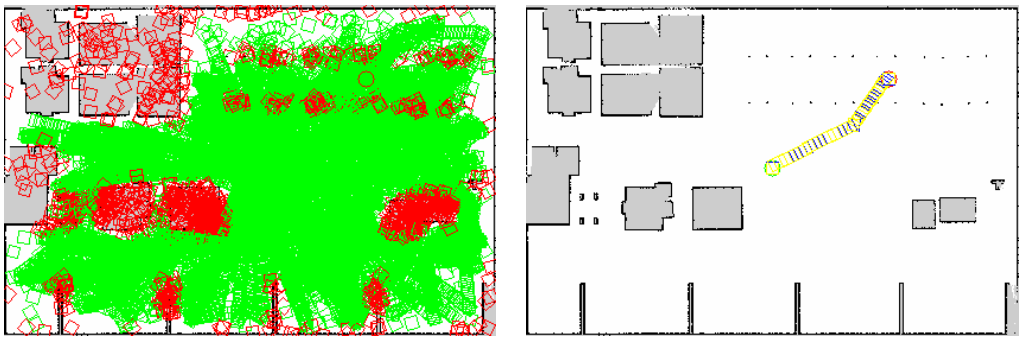
el estado en evaluación en el píxel central. De manera que, un estado es considerado válido cuando todos los píxeles del conjunto estén libres. Para alinear correctamente la representación geométrica del robot con su orientación real, aplicamos una matriz de rotación antes de examinar los píxeles:

$$x' = x \cdot \cos\theta - y \cdot \sin\theta$$
$$y' = x \cdot \sin\theta + y \cdot \cos\theta$$

```
def isStateValid(state):
    w = min(int(state.getX()), MAP_WIDTH - 1)
    h = min(int(state.getY()), MAP_HEIGHT - 1)
    yaw = state.getYaw()

    for x in range(-robotw, robotw, 1):
        for y in range(-robotl, robotl, 1):
            newx, newy = rotationMatrix(x, y, w, h, yaw)
            if 0 < newx < MAP_WIDTH and 0 < newy < MAP_HEIGHT:
                c = mapImg[newy, newx]
                free = c[0] * 255 > 250 and c[1] * 255 > 250 and c[2] * 255 > 250
            else:
                free = False
            if not free:
                break
        if not free:
            break

    return free
```

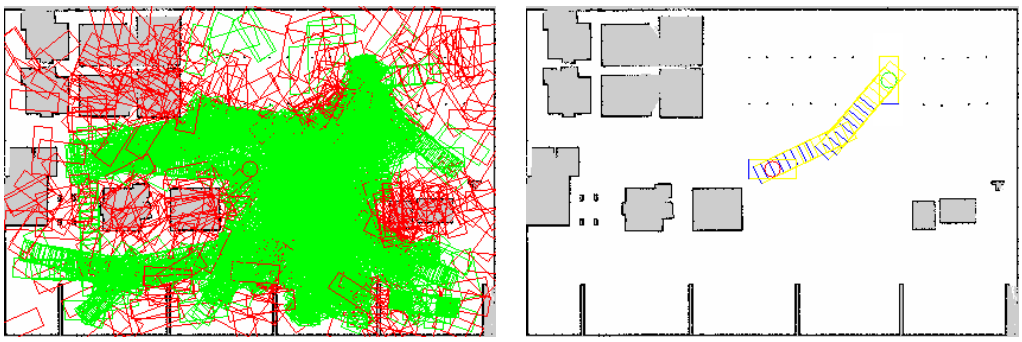
Código 5.7: Función *isStateValid()*

A continuación, se ve ilustrada el proceso de planificación de rutas. En las subfiguras 5.17(a) y 5.18(a) se muestran el proceso de validación de los estados, los bloques verde son los estádos válidos mientras que los bloques rojos son los válidos. Y en las subfiguras 5.17(b) y 5.18(b) se ve trazadas las rutas solución encontradas.



(a) Proceso de validación de estados (b) Ruta válida encontrada

Figura 5.17: Planificación de la ruta para ir a la zona de carga



(a) Proceso de validación de estados (b) Ruta válida encontrada

Figura 5.18: Planificación de la ruta para volver a la zona de descarga

5.5.2.2. Pilotaje del robot

En esta planificación, la secuencia de estados devuelta por la función *generatePath()* tiene el formato (*x, y, yaw*). Puesto que ya nos indica la orientación del robot para su alineación con el siguiente *waypoint*, en principio, no sería necesario realizar cálculos adicionales. Sin embargo, esta implementación no aplica un pilotaje reactivo, lo que puede conllevar a una acumulación de errores y resultar en una navegación poco precisa, especialmente en rutas largas. Por esta razón, se opta por utilizar la misma metodología explicada en la sección 5.5.1.2.

5.5.2.3. Máquina de estados

El diseño de la máquina de estados es casi idéntica a la explicada en la sección 5.16, la única diferencia radica en el estado *GENERATE_PATH*. En este caso, no es necesario

aplicar ninguna dilatación al mapa, y hay que especificar el ancho y largo del robot en pixeles. Para el trayecto de ida, la dimensión del robot será de 7 x 7 pixeles, y para el trayecto será de 20 x 9 píxeles.

El vídeo demostrativo de la solución de referencia con planificación espacial con restricción de movimientos se puede encontrar en el siguiente enlace: `https://www.youtube.com/watch?v=-2D90I-wZKs&t=16s`

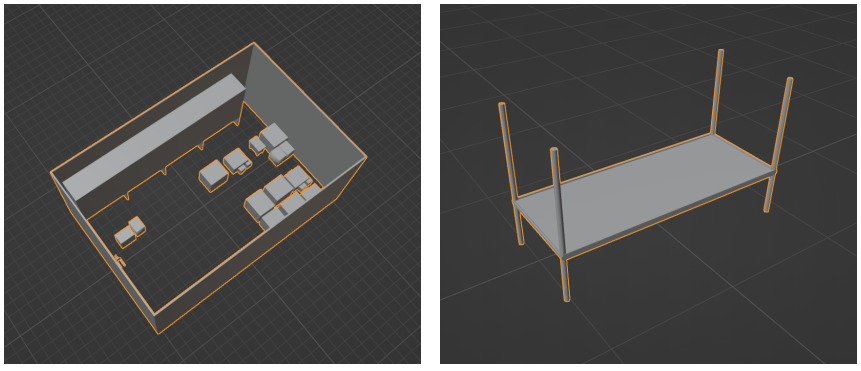
5.5.3. Planificación basado en controles con robot Ackermann

Habiedo obtenido resultados favorables con soluciones geométricas, la atención ahora se dirige hacia un enfoque diferente: la planificación basada en controles, es decir, incorporando restricciones de movimiento entre las posibilidades motrices del robot. Por ejemplo, un robot tipo Ackermann, que son frecuentes en robótica, no puede girar en el sitio como un robot holonómico.

5.5.3.1. Planificación de la ruta

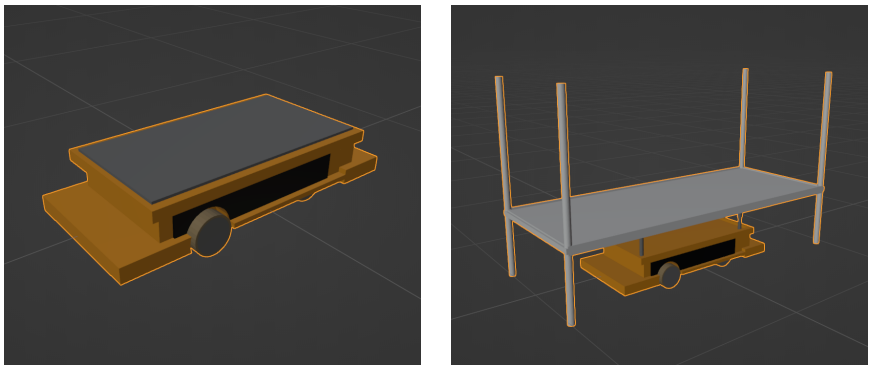
En la tercera solución de referencia propuesta, se utiliza la clase *app* para la resolución del problema de planificación. Esta clase facilita la definición eficiente del espacio de estados mediante ficheros CAD que especifican las mallas de colisión.

Antes de proceder con la planificación, se crearon nuevos modelos DAE del entorno y robot simulado (figuras 5.19 y 5.20) con Blender, definiendo cada elemento como una pieza única para facilitar su definición posterior en el problema. Se ha decidido tratar las estanterías como entidades independientes del entorno simulado, permitiendo su exclusión individual del espacio de estados cuando son parte del robot. Debido a las limitaciones de la clase *app* , que no permite la reubicación de elementos del entorno una vez definidos, se tuvo que crear un fichero DAE distinto para cada estantería, incorporando sus posiciones fijas directamente en estos ficheros.



(a) Modelado del almacén (b) Modelado de la estantería

Figura 5.19: Modelado de los elementos del entorno



(a) Modelado del robot solo

(b) Modelado del robot con una estantería cargada

Figura 5.20: Modelado del robot

Una vez preparadas las mallas de los modelos, procedemos con el proceso de planificación. Los pasos son los siguientes:

1. Establecer el tipo de robot. Puesto que el robot logístico Ackermann es un modelo de coche cinemático, se utiliza la clase *KinematicCarPlanning* para la planificación.

2. Configurar el espacio de estados definiendo la malla del entorno. Se añade la malla del almacén con la función *setEnviromentMesh()* y se añade las estanterías individualmente mediante la función *addEnviromentMesh()*.

3. Configurar la malla del robot con la función *setRobotMesh()*. El modelo del robot utilizado en el trayecto de ida es el de la figura 5.20(a) y en el de vuelta es el de la figura 5.20(b).

4. Establecer los límites del espacio de estados, es decir, las dimensiones.

5. Establecer los límites del espacio de controles, incluyendo la velocidad lineal máxima y mínima, y el ángulo máximo y mínimo de rotación de las ruedas delanteras del robot.

6. Establecer el estado inicial y final, un planificador, el tamaño del paso de propagación y la duración mínima y máxima de control en unidades de pasos (*steps*).

7. Intentar resolver el problema definido. La ruta resultante del planificador comprende una serie de *waypoints*, que detallan la velocidad lineal y ángulo de rotación de las ruedas a aplicar en cada paso. Los *waypoints* tienen el siguiente formato:

```
At state Compound state [
RealVectorState [0 0]
SO2State [0]
]
  apply control RealVectorControl [3.76703 0.00946806]
  for 1 steps
```

Código 5.8: Formato de los waypoints en la planificación basada en control

La interpretación del ejemplo superior es: estando en la posición (0, 0) con una orientación de 0, aplicar una velocidad lineal de 3.76703 m/s y un ángulo de rotación de 0.00946806 rad durante 1 *step*.

La ruta proporcionada por el planificador no asegura ser la óptima y, en la mayoría de casos, no devuelve una solución exacta sino una aproximada, cuyo estado final difiere del destino establecido.

A diferencia de la planificación geométrica, en este caso, el tipo de planificador influye considerablemente en la solución, así como los valores de tiempo de ejecución (*runtime*), tamaño de paso de propagación (*stepsize*) y duración máxima de control (*maxduration*). Por esta razón, se realizaron pruebas comparativas analizando la longitud de la ruta obtenida (*length*) y la distancia de error entre el estado final de la ruta solución respecto al estado final establecido (*error*) para determinar la configuración óptima de la planificación, analizando diferentes planificadores y configuraciones de tiempo de ejecución, tamaño de paso y duración máxima de control.

Por cada configuración se generaron 100 caminos divididos en 10 grupos. Se calculó la media de los valores *length* y *error* de los 10 caminos de cada grupo y posteriormente la media entre los 10 grupos. Los resultados obtenidos para los diferentes paŕametros de la planificación fueron los siguientes:

- Planificador: *runtime* = 20.0 s, *stepsize* = 0.1 s, *maxduration* = 10 steps

Planner	RRT	SST	EST	KPIECE	PDST	SyclopRRT	SycloEst
Length (m)	43.07	28.18	24.88	29.2	25.136	17.82	17.92
Error (m)	0.19	0.29	0.50	0.72	0.48	2.82	2.62

Tabla 5.1: Resultados obtenidos con diferentes planificadores

Como se puede observar, los planificadores RRT y SST muestran un error relativamente bajo, sin embargo, la longitud media de la ruta es de las más altas, por lo que no serían planificadores adecuados. Lo mismo ocurre con los planificadores SyclopRRT y SyclopEST, a pesar de mostrar longitudes corta, el error es demasiado alto. Finalmente, optamos por el planificador PDST.

- Tiempo de ejecución: *planner* = PDST, *stepsize* = 0.1 s, *maxduration* = 10 steps

Run time (s)	5	10	15	20	25	30	35	40	45	50
Length (m)	25.52	24.44	26.42	25.45	25.46	25.72	26.17	27.20	26.55	26.54
Error (m)	0.73	0.64	0.57	0.45	0.44	0.39	0.42	0.45	0.36	0.31

Tabla 5.2: Resultados obtenidos con diferentes tiempos de ejecución

A partir de 20s, la longitud media se estabiliza en alrededor de 25m y el error en alrededor de 0.45m, hasta los 45s que muestra una disminución del error. Sin embargo, no conviene elegir un valor tan alto porque hay que encontrar un equilibrio entre eficiencia y eficacia.

- Tamaño de paso de propagación: *planner* = PDST, *runtime* = 20.0 s, *maxduration* = 10 steps

Stepsize (s)	0.01	0.05	0.1	0.15	0.2	0.25	0.3	0.35	0.4	0.45
Length (m)	29.33	30.22	25.06	26.38	24.57	24.26	23.57	24.95	25.87	25.24
Error (m)	4.91	0.64	0.46	0.40	0.38	0.34	0.34	0.35	0.36	0.35

Tabla 5.3: Resultados obtenidos con diferentes tamaños de pasos de propagación

A partir de 0.25s la longitud se estabiliza alrededor de 25m y el error en 0.35m.

- Duración máxima de control: *planner* = PDST, *runtime* = 20.0 s, *stepsize* = 0.1 s

Max duration (steps)	5	10	15	20	25	30	35	40	45	50
Length (m)	31.45	25.24	23.16	21.77	20.50	20.80	19.27	20.50	20.25	18.51
Error (m)	0.50	0.49	0.48	0.43	0.52	0.43	0.44	0.46	0.43	0.43

Tabla 5.4: Resultados obtenidos con diferentes duraciones máximas de control

A partir de 25 pasos la longitud se estabiliza alrededor de 20m y el error en 0.43m.

En conclusión, un tiempo de ejecución de 25.0s, un paso de propagación de 0.25s y una duración máxima de 25 pasos son valores adecuados para la planificación. Sin embargo, esta configuración solo asegura la generación de rutas relativamente mejores, pero sigue sin garantizar una ruta realmente factible, ya que los valores analizados son una media. Para asegurar la selección de una ruta factible, se implementó un criterio de optimización adicional que involucra la generación de múltiples caminos, en este caso 10, y la selección

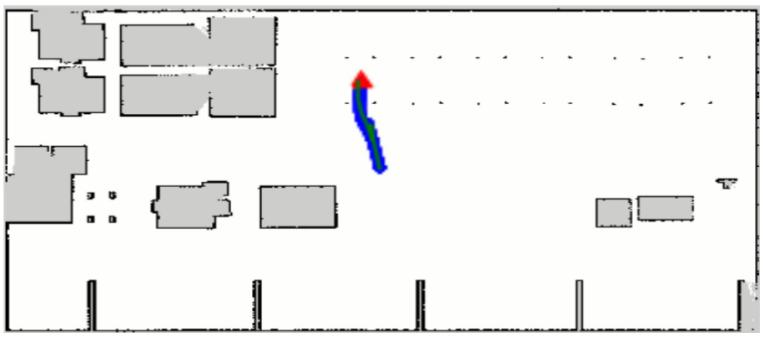
del óptimo entre estos basado en un equilibrio entre la longitud de la ruta y el error. Concretamente, se elige la ruta más corta siempre y cuando el error de la posición final con respecto a la deseada no sea muy alta, en nuestro caso, hemos decidido que debe ser menor de 0.2 m. Si ninguna de las 10 rutas tiene un umbral menor de 0.2, se escogerá el plan con menor error y menor longitud en general.

5.5.3.2. Pilotaje del robot

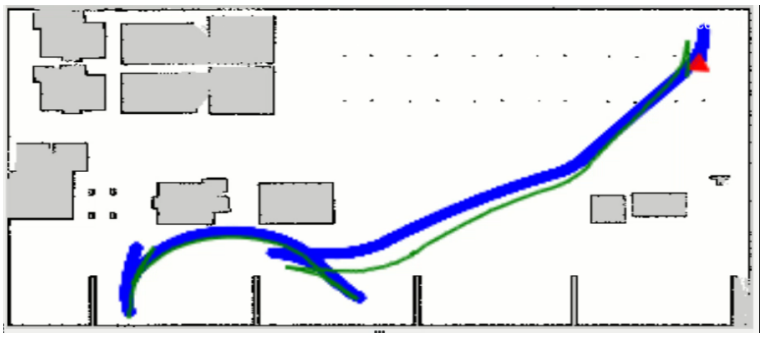
A diferencia de las otras soluciones, que se indexaba por posición recorriendo la secuencia de coordenadas, en este caso se implementa la solución mediante indexado en el tiempo, recorriendo la secuencia de los pares de velocidad *(v, w)*. Nos guardamos una marca de tiempo al principio de la acción y haciendo la diferencia del tiempo actual con esta marca podemos saber cuánto tiempo ha transcurrido. Cuando este tiempo sea igual o mayor a la duración indicada salta al siguiente *waypoint* y actualiza la marca de tiempo.

La idea parece directa y todo parece tener que ir bien, sin embargo, hay que tener en cuenta otro factor: la frecuencia de iteración de la propia plataforma. Este factor puede conllevar a la generación de errores que influye la precisión de la navegación. Por ejemplo, si la duración tiene que ser 1 segundo pero el waypoint se actualiza en el segundo 1.1, ese 0.1 sobrante hace que el robot haya navegado una distancia $v \times 0.1$ mayor que la planificada. Y el error producido en cada subtrayecto se va acumulando hasta un punto en el que el robot se desvía completamente de la ruta planificada.

Para mitigar estos errores la solución propuesta es ampliar los intervalos de tiempo, alargar la duración de la acción y reducir la velocidad lineal en proporción a esta alargación. De esta forma, la distancia recorrida sigue siendo la misma, pero el error generado va a ser menor. Sin embargo, este diseño solo garantiza generar errores menores, pero no la eliminación completa de ellos. Por lo que, la acumulación de pequeñas desviaciones sigue teniendo gran impacto en rutas largas como se puede observar en la figura 5.21. Para solucionar este problema habría que implementar un algoritmo de correción de errores que al menos pueda evitar la acumulación de ellos.



(a) Ruta corta



(b) Ruta larga

Figura 5.21: Resultados de la ejecución de la ruta

5.5.3.3. Máquina de estados

La estructura de la máquina de estados es igual a la presentada en la sección 5.16. El proceso comienza en el estado *GENERATE_PATH*, dependiendo del estado de la plataforma se carga un modelo u otro del entorno y robot, y se busca una ruta aplicando el criterio explicado en la sección 5.5.3.1. Luego, pasa al estado *REACH_DESTINY*, donde el robot recorre toda la secuencia de *(v, w)* y pasa al estado *LOAD_UNLOAD_SHELF*. Si el estado de la plataforma es *NO_LIFTED*, el robot eleva la plataforma para cargar la estantería, actualiza el estado de la plataforma a *LIFTED* y vuelve al estado *GENERATE_PATH*; en el caso contrario, baja la plataforma y pasa al estado *FINISH*, donde el robot se detiene.

El vídeo demostrativo de la solución de referencia con planificación basada en controles se puede encontrar en el siguiente enlace: `https://youtu.be/0PO4S8Omn30`

5.5.4. Comparativa de las tres soluciones de referencia

A continuación, se realiza una comparativa de las tres soluciones de referencia, destancando las ventajas y limitaciones observadas en cada una de ellas:

- Planificación geométrica en un espacio puramente euclídeo
  - Ventajas:
    - Generación de rutas óptimas
    - Diseño de pilotaje reactivo permitiendo una navegación precisa basada en posición
  - Desventajas:
    - No aplicable en espacios estrechos
- Planificación geométrica con restricciones de control del robot
  - Ventajas:
    - Generación de rutas óptimas
    - Aplicable en espacios estrechos
    - Diseño de pilotaje reactivo permitiendo una navegación precisa basada en posición
  - Desventajas:
    - Implementación compleja en comparación con las otras dos
- Planificación basada en control
  - Ventajas:
    - Aplicable en espacios estrechos
  - Desventajas:
    - Obtención de rutas subóptima
    - Sacrificio de velocidad de navegación por precisión de navegación
    - Diseño de pilotaje no reactivo, por lo que conlleva a acumulación de error.

Capítulo 6
Conclusiones

En este capítulo final, hacemos un resumen de los objetivos logrados y las competencias adquiridas. Además, se exponen posibles líneas de mejoras futuras.

6.1. Conclusiones finales

Hemos logrado cumplir con el objetivo propuesto en la sección 2 de mejorar la plataforma educativa robótica, Robotics Academy, cumpliendo con los dos subojetivos propuestos:

1. Actualizar y migrar el ejercicio Sigue-Persona a la versión 5.1 de la plataforma Robotics Academy.

   Partiendo de una versión previa en ROS 2 Foxy implementada por Carlos Caminero en su TFG[2], se realizó una revisión de los drivers del TurtleBot2 (simulado y real) para ROS 2 Humble y se integró satisfactoriamente este trabajo en la plataforma para ser utilizado en el ejercicio. Posteriormente, se actualizaron las plantillas de la página web a base de React, se crearon nuevas plantillas Python adaptadas a la nueva arquitectura de Robotics Academy y se reimplementó la cadena de comunicación del componente teleoperador de la persona simulada a seguir. Por último, se añadieron mejoras a la solución de referencia de Carlos Caminero y se actualizó la documentación de apoyo al usuario.

2. Crear un nuevo ejercicio Almacén-Amazon para la plataforma Robotics Academy v5.1.

   Se hicieron ajustes en los modelos 3D de los almacenes y del robot holonómico, y se diseñó un nuevo modelo de un robot logístico con geometría rectangular y movimiento Ackermann; se desarrolló la página web incluyendo visores de los almacenes que muestran el robot, la ruta planificada y la ruta seguida; se crearon las plantillas Python necesarias para simplificar el acceso a los estudiantes de Robotics Academy; se implementaron tres soluciones de referencia, desde la más sencilla hasta la más completa, que tienen en cuenta la geometría del robot (con y sin estantería) así como sus restricciones de movimientos (vehículos Ackermann y holonómicos); finalmente,

se creó la documentación de apoyo al usuario junto con vídeos demostrativos de las soluciones de referencia implementadas.

Y como logro adicional, el ejercicio Almacén-Amazon ha sido utilizado en la asignatura Robótica de Servicio de universidad Rey Juan Carlos por alrededor de 30 alumnos.

6.2. Competencias adquiridas

Las competencias adquiridas durante el desarrollo del TFG son:

- Conocimientos avanzados de Docker: creación y uso de imágenes complejas.
- Aprendizaje básico de tecnologías web sobre React, HTML, CSS y Javascript.
- Aprendizaje y manejo de la librería OMPL.
- Correcta metodología para trabajar en proyectos de software libre en Github.

6.3. Líneas futuras

Lograda la meta principal del TFG, se propone algunas líneas de desarrollo futuras:

- Mejoras de la solución con planificación basada en control, añadir correción de errores para mantener valores de desviación bajos.
- Explorar la adaptación de la solución geométrica para el modelo del robot Ackermann.
- Adaptar la solución basada en controles para el modelo del robot holonómico.
- Adaptar una versión del ejercicio Amazon Warehouse para que sea utilizable con robots TurtleBot2.

Bibliografía

[1] Bmw group introduce robots autónomos en la logística de suministro, March 2016. https://www.press.bmwgroup.com/latin-america-caribbean/article/detail/T0258125ES/bmw-group-introduce-robots-aut%C3%B3nomos-en-la-log%C3%ADstica-de-suministro?language=es.

[2] C. C. Abad. Ejercicio sigue-persona para la plataforma académica robotics academy, usando un robot real y simulado en ros2, July 2021. https://github.com/RoboticsLabURJC/2021-tfg-carlos-caminero/blob/main/memoria.pdf.

[3] E. H. Amazon. Los robots en números: datos y cifras sobre los robots en amazon, January 2019. https://www.xataka.com/robotica-e-ia/este-es-el-primer-almacen-robotizado-de-amazon-en-espana.

[4] P. A. Pérez. Infraestructura de programación de robots aéreos y aplicaciones visuales con aprendizaje profundo, February 2022. https://github.com/RoboticsLabURJC/2021-tfm-pedro-arias/blob/main/tfm/20220126_tfm_pedroariasperez.pdf.

[5] I. A. Şucan, M. Moll, and L. E. Kavraki. The Open Motion Planning Library. *IEEE Robotics & Automation Magazine*, 19(4):72–82, December 2012. https://ompl.kavrakilab.org.