ESCUELA TÉCNICA SUPERIOR DE INGENIERÍA DE TELECOMUNICACIÓN

GRADO EN INGENIERÍA DE ROBÓTICA SOFTWARE

TRABAJO FIN DE GRADO

Ejercicios Sigue-Persona para la plataforma académica Robotics Academy, usando un robot real y simulado en ROS2

Autor: Carlos Caminero Abad
Tutor: Dr. Jose María Cañas Plaza

Curso académico 2021/2022

Agradecimientos

El final ha llegado. Escribiendo estas palabras, me acuerdo de todas aquellas personas que han estado presentes en mi etapa universitaria, y que gracias a ellos estoy aquí, a unos días de presentar un proyecto que marcará el final de una etapa de mi vida y el principio de otra.

Quisiera dar las gracias a mis padres Carlos y María del Pilar y a mi hermano Alejandro. Siempre han estado conmigo, apoyándome y ayudándome cuando lo he necesitado. Gracias a mis padres, conocí este grado universitario, y haberlo elegido ha sido una de las mejores decisiones que he tomado en mi vida. Siempre les estaré agradecido.

Agradezco a los miembros de Robotics Academy y Unibotics, a los miembros de Seiki Robotics (PowerBots) y a todos mis profesores del grado por su dedicación e interés por la enseñanza que me han transmitido y han hecho de mí un mejor estudiante.

También a mis amigos de Yepes por todos los buenos momentos compartidos. Por todas aquellas cenas en las que nos desahogábamos de nuestros problemas y por todo el apoyo y ánimo que me han transmitido. Es un placer compartir esta vida con gente tan excelente como ellos.

Por otra parte, agradezco a mis compañeros de clase por estos cuatro años que, a pesar de compartir duras y difíciles experiencias como la pandemia, también he podido compartir buenos momentos de satisfacción y alegría con ellos.

Por último, me gustaría dedicar este párrafo a mi tutor Jose María, por haberme estado orientando constantemente a lo largo de todos estos meses dedicados al proyecto. Gracias por tu profesionalidad, tu dedicación y ánimos, y por tu ayuda cuando más la necesitaba, que en momentos en los cuales estaba bloqueado he conseguido avanzar en el camino. Ha sido todo un honor haberte tenido como profesor y como tutor.

¡Muchas gracias a todos!

Resumen

La Robótica es un sector en constante crecimiento. Cada vez es más importante el perfil del Ingeniero en Robótica Software para la programación de Robots y Automatismos en esta nueva era digital. Por ello, es necesario la correcta formación del Ingeniero a través de la accesibilidad y disponibilidad de los recursos educativos que fomentarán su aprendizaje autodidacta.

El objetivo del siguiente Trabajo Fin de Grado (TFG) es desarrollar dos nuevos ejercicios educativos para la plataforma de enseñanza universitaria Robotics Academy consistentes en programar un robot modelo TurtleBot2 de Yujin para que sea capaz de seguir a una persona tanto en un entorno real como en un entorno simulado (un hospital). Con ello, los alumnos y otros usuarios adquirirán las destrezas necesarias para la programación de una tarea muy solicitada en la Robótica de Servicio. Aprenderán a seguir a una persona usando Redes Neuronales que ayudarán a la detección visual de personas en imágenes

La incorporación de estos dos nuevos ejercicios se aplica a una nueva rama de desarrollo para la plataforma Robotics Academy que usa la distribución ROS2 Foxy. Además, se ha logrado la migración y adaptación del modelo simulado del Turtlebot2 de ROS Noetic a ROS2 Foxy para futuros usos. Se ha preparado la página web operativa correspondiente a cada ejercicio y se han desarrollado las respectivas soluciones de referencia que combinan percepción visual robusta y navegación con el algoritmo VFF.

Acrónimos

CNN *Convolutional Neural Network*

CPU *Central Processing Unit*

CSS *Cascading Style Sheets*

DNN *Deep Neural Network*

FSM *Finite State Machine*

GPU *Graphics Processing Unit*

GUI *Graphical User Interface*

HAL *Hardware Abstraction Layer*

HTML *HiperText Markup Language*

HTTP *HiperText Transfer Protocol*

IA *Inteligencia Artificial*

LIDAR *Laser Imaging Detection and Ranging*

ML *Machine Learning*

PID *Proportional-Integral-Derivative*

RNA *Redes Neuronales Artificiales*

R-CNN *Region-based Convolutional Neural Network*

ROS *Robot Operating System*

UDP *User Datagram Protocol*

URDF *Unified Robot Description Format*

USB *Universal Serial Bus*

XML *eXtensible Markup Language*

Índice general

Índice de figuras

Listado de códigos

Capítulo 1
Introducción

La tarea Sigue-Persona es una de las más empleadas y solicitadas en los Robots de Servicio para multitud de sectores que incorporan soluciones robóticas. Este TFG propone desarrollar dos nuevos ejercicios para el entorno educativo Robotics Academy con el fin de que sean integrados en la plataforma web Unibotics. Ambos ejercicios requieren que los usuarios programen un robot TurtleBot2, tanto simulado como real, para que realice dicha tarea.

Este primer capítulo presenta el estado actual de varios campos de la Robótica que están directamente relacionados con este proyecto para poner en contexto al lector. Por un lado presentaremos la Robótica Móvil, un sector de la robótica que está en constante crecimiento y muy presente en la Robótica de Servicio; por otra parte, introduciremos las Redes Neuronales, un avance significativo en IA que ha permitido realizar aplicaciones muy robustas basadas en datos, tanto en la percepción como en el razonamiento. Por último hablaremos de la Robótica Educativa, e introduciremos varias plataformas como TheConstruct, Robotics Academy, o Unibotics siendo estas dos últimas las que incorporarán el ejercicio educativo Sigue-Persona.

1.1. Robótica Móvil

La robótica es la ciencia que engloba varias ramas tecnológicas o disciplinas, con el objetivo de diseñar máquinas ("robots") que sean capaces de realizar tareas automatizadas o de simular el comportamiento humano o animal, en función de la capacidad de su software [20]. Su término se remonta a la obra de ciencia ficción escrita por Isaac Asimov: Yo, Robot. (1950)

Una vez que la robótica emerge a partir de mediados del siglo XX, surgen 2 sectores: la Robótica Industrial y la Robótica de Servicio. Los Robots Industriales se encargan de realizar tareas automatizadas y muy repetitivas en escenarios industriales, donde el entorno está muy controlado. Los Robots de Servicio son aquellos que realizan tareas útiles para el ser humano y mejorar su comodidad. Entre los robots de servicio, destacan los robots móviles.

Un *robot móvil* es un sistema electromecánico capaz de desplazarse de manera autónoma sin estar sujeto físicamente a un solo punto[1]. Posee sensores que permiten monitorear en cada momento su posición relativa a su punto de origen (odometría) y a su punto de destino. Normalmente su control se basa en un sistema de lazo cerrado (modo de actuación está determinado por su estado anterior). Se clasifican en robots

móviles de locomoción con ruedas, locomoción con patas y locomoción con orugas. Los primeros son más fáciles de controlar pero no se adaptan bien a muchas superficies, sin embargo los robots con patas (cuadrúpedos o bípedos) se adaptan muy bien a distintas superficies pero requieren varios controladores para mantener su equilibrio tanto estático como dinámico. El entorno en el que se enfrentan es bastante *heterogéneo*, por tanto su software debe ser lo más robusto posible para adaptarse correctamente a los cambios.

1.1.1. Evolución histórica

Podemos destacar varios hechos importantes a lo largo del siglo XX y principios del XXI que marcaron grandes avances en la robótica:

- En 1952 aparece la primera máquina de control numérico del MIT que era capaz de automatizar algunas tareas industriales.
- En 1961, la compañía Unimate introdujo el primer robot industral en la General Motors.
- En 1966, comenzó el desarrollo del primer robot móvil llamado Shakey [Nilsson, 1984].

Figura 1.1: Shakey (1966-1972) Imagen obtenida de [11].

- En los años 70, NASA desarrolló MARS-ROVER, una plataforma móvil que integraba un brazo mecánico, sensores de proximidad, un dispositivo telemétrico láser y cámaras estéreo.
- En 1997, la NASA envió a Marte un dispositivo móvil llamado Sojourner Rover, para enviar fotografías a la Tierra del planeta rojo. Además, la empresa HONDA sacó el primer humanoide capaz de imitar comportamientos humanos.
- En 2000, Honda presenta el robot Asimo.

- En 2004, Qrio de Sony, un pequeño robot humanoide capaz de correr, bailar, y reconocer caras.
- En 2008, SoftBank Robotics presenta el robot Nao, un robot bípedo dedicado a interactuar con el ser humano y ser muy amigable. En 2014, la empresa presenta el robot Pepper, robot con forma humana pero se desplaza con ruedas. Se ha usado sobre todo como robot guía, recepcionista y en exhibiciones, sin embargo, fue abandonado en 2021 (Figura 1.2).

Figura 1.2: Nao y Pepper. Imagen obtenida de [19].

- En 2013, Boston Dynamics saca a la luz el robot Atlas, un humanoide con actuadores neumáticos. La empresa lo usa para imitar el comportamiento humano llevándolo a otro nivel haciendo acrobacias o incluso parkour. En 2020, la empresa saca a la venta un robot cuadrúpedo llamado Spot que imita el comportamiento de un perro, que se usa en algunas fábricas para transportar materiales o realizar tareas de mantenimiento (Figura 1.3).

Figura 1.3: Atlas y Spot. Imagen obtenida de [14].

1.1.2. Aplicaciones actuales

Como hemos visto, el avance de los robots móviles ha emergido durantes estas últimas décadas. Con ellos han surgido una gran diversidad de aplicaciones:

1. Aplicaciones domésticas. Podemos destacar los robots de limpieza como Roomba (robot aspirador) y Braava de iRobot (robot friegasuelos) o WinBot de Ecovacs (limpia ventanas). La realización de sus tareas se logra mediante algoritmos de navegación y cobertura para limpiar el mayor espacio en el menor tiempo posible.

2. Agricultura. Incorporación de tractores o vehículos autónomos encargados del mantenimiento y monitorización de los cultivos.

3. Inspección y mantenimiento en zonas críticas (centrales nucleares) o poco seguras para el ser humano

4. Logística: tanto para mover mercancías en almacenes como para reparto. Algunos robots móviles se usan de prototipo para las entregas a domicilio (última milla).

5. Conducción autónoma. Empresas como Tesla o Waymo han sacado coches con algunas capacidades autónomas. El objetivo para el futuro es desarrollar coches con el nivel más alto de autonomía.

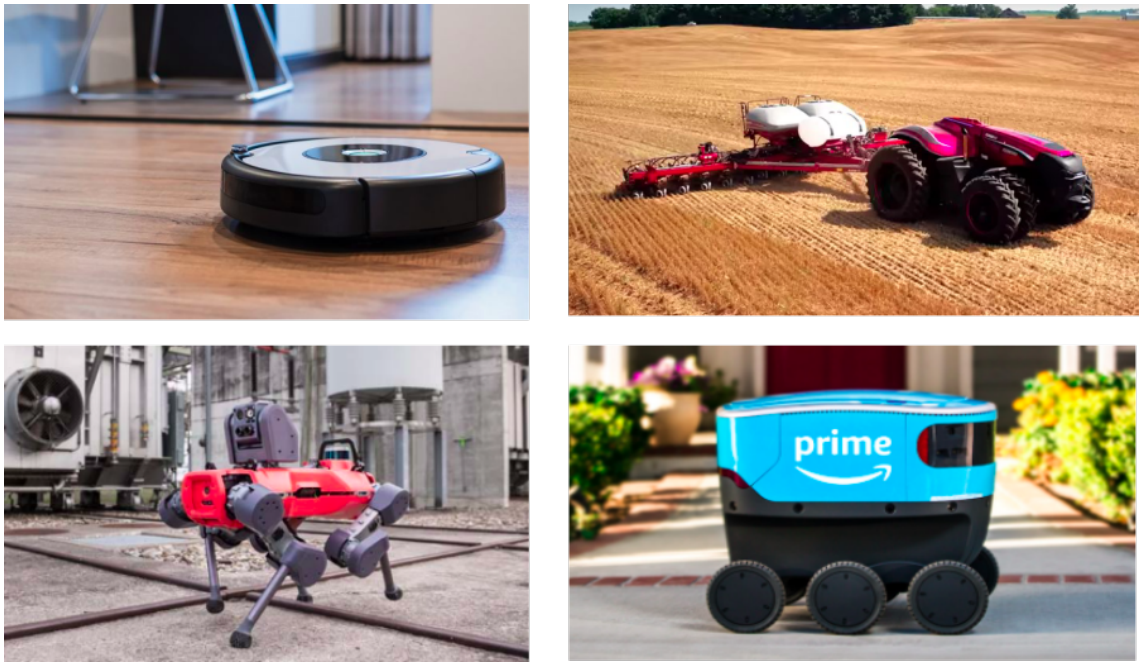

Figura 1.4: Ejemplos de aplicaciones de los Robots de Servicio

1.2. Redes Neuronales Artificiales (RNA)

La Inteligencia Artificial (IA) es un campo muy extenso que está evolucionado rápidamente en este siglo XXI. Se basa en el estudio de algoritmos que permitan a las máquinas realizar tareas de manera "inteligente" (búsqueda, lógica, teoría de juegos, planificación, etc). Cuando incorporamos a un computador la capacidad de *aprendizaje* es entonces cuando hablamos de *Machine Learning*.

Según *Arthur Samuel*, el Aprendizaje Automático o Machine Learning es el "campo de estudio que otorga a los ordenadores la capacidad de aprender sin ser programados explícitamente". Para ello se emplean diversas técnicas dependiendo del resultado deseado: Regresión Lineal, Regresión Logística, K-NN, K-Means, SVM, Redes Neuronales, Q-Learning, etc. Un paradigma de procesamiento en Aprendizaje Automático muy exitoso en los últimos años son las redes neuronales artificiales.

Una *Red Nueronal Artificial (RNA)* es un modelo computacional inspirado en el cerebro humano que sirve entre otras cosas para clasificar datos de entrada con su correspondiente salida a partir de un entrenamiento previo. Para ello se le proporciona un *dataset* (conjunto de datos), donde indicamos, por cada muestra, a qué clase pertenece (Aprendizaje Supervisado). Al iniciar el entrenamiento, se realiza un proceso de optimización ajustando los valores de unas variables que actúan como pesos o cargas de las dendritas de las neuronas, consiguiendo un nivel de clasificación determinado. Pero, ¿cómo funciona una neurona?

Una neurona, tal y como se muestra en la Figura 1.5 está formada por los siguientes elementos:

- **Valores de Entrada**: Puede ser tanto los valores de las *características* de cada muestra como la salida de la neurona de una capa anterior.
- **Pesos**: Son unos valores que se optienen al final de la etapa de entrenamiento.
- **Sumatorio**: Esta formado por los productos de los Valores de Entrada por sus correspondientes Pesos
- **Función de Decisión/Activación**: Es una función matemática que se aplica al sumatorio: Lineal, Escalón, Sigmoide, etc
- **Salida**: Es el valor de la función de decisión. Dependiendo de la función de activación usada, la salida será distinta.

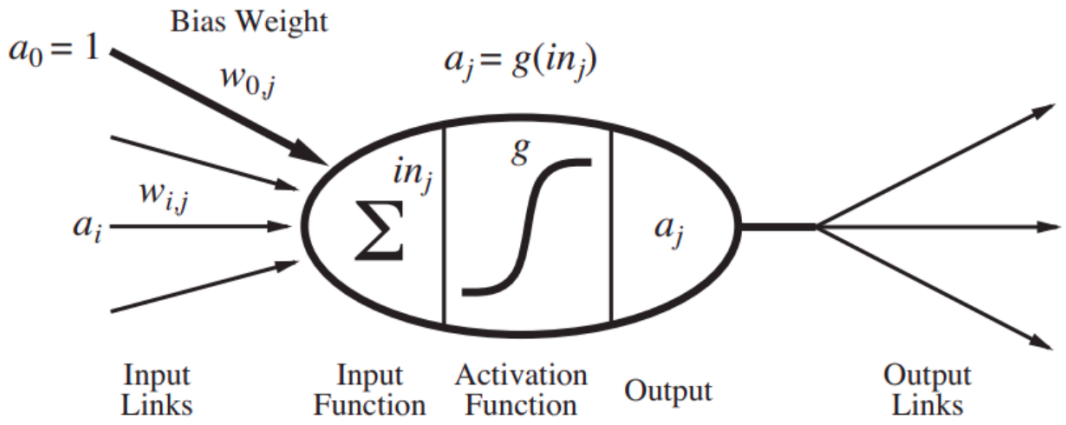

Figura 1.5: Modelo computacional de una neurona [22]

Tomando como ejemplo un Perceptrón (neurona utilizada para resolver tareas de clasificación binaria) la clasificación se realiza mediante un *hiperplano de separación* sujeto a la función de decisión (función escalón) separando en dos clases (0 o 1) los datos de entrada. La posición y orientación del hiperplano de separación está determinado por el proceso de optimización realizado durante el entrenamiento de la neurona. En la Figura 1.7 mostramos dos ejemplos de un Perceptrón entrenado a partir de cuatro datos de entrada que simulan una puerta lógica AND y OR respectivamente:

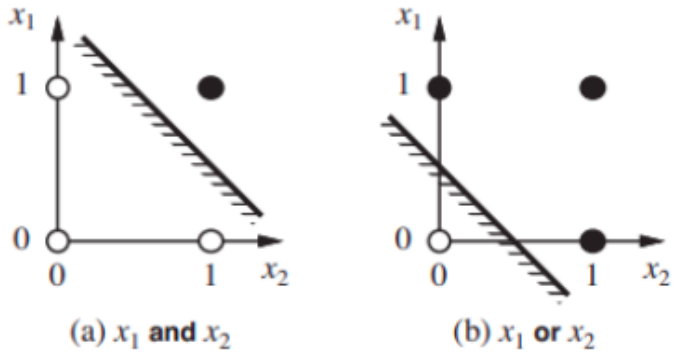

Figura 1.6: Salida de un perceptrón de tipo AND y OR [22]

Cuando los datos de entrada no son linealmente separables mediante un hiperplano de separación es cuando necesitamos aumentar el número de neuronas. Normalmente, una RNA consta de un conjunto de capas formado por varias neuronas que se encuentran interconectadas. Suelen dividirse en 1 capa de entrada, 1 o más capas ocultas y 1 capa de salida.

Cuando el problema de clasificación involucra una gran cantidad de información y entradas como puede ser un video o una imagen, se usan modelos de redes neuronales profundas (*Deep Neural Networks*, DNN) de varias capas con un elevado número de neuronas. Variante de *Machine Learning* conocida como *Deep Learning*.

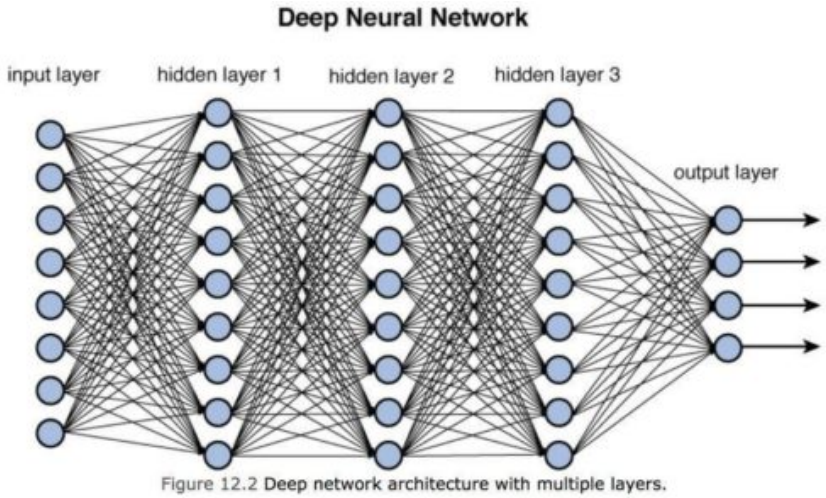

Figura 1.7: Red Neuronal Profunda. Imagen obtenida de [24]

Una aplicación muy popular de Redes Neuronales es la detección de objetos mediante *Vision Artificial*: dada una imagen/fotograma como valor de entrada, la red neuronal es capaz de detectar ciertos objetos solicitados como puede ser fruta, coches, utensilios o incluso personas y animales dependiendo del entrenamiento aplicado.

1.2.1. Redes Neuronales Convolucionales (CNN)

Las *Redes Neuronales Convolucionales, CNN* son un tipo de RNA multicapa aplicados para identificar y clasificar objetos en una imagen o video. Se basan en la *Convolución de capas*. El funcionamiento de la convolución itera sobre todos los píxeles de una imagen y aplica varios filtros de *Kernel* [1] obteniendo así varias matrices de píxeles, las cuales se les aplican una función de *decisión* obteniendo una mapa de detección de características.

La siguiente capa tendrá matrices más pequeñas debido a la reducción de los mapas de características (*Max-Polling*), y así sucesivamente, hasta que en las últimas capas obtenemos patrones que diferencian a unos objetos de otros. A la última capa oculta de neuronas se le aplica una función llamada *Softmax* generando una capa de salida con el mismo número de neuronas que de clases.

Al usar la CNN dada una imagen obtendremos una lista de porcentajes normalizados de 0 a 1 indicando el grado de probabilidad de que la imagen se corresponda con cada clase. En la Figura 1.8 podemos ver un ejemplo de OCR (Reconocimiento Óptimo de Caracteres) usando CNN.

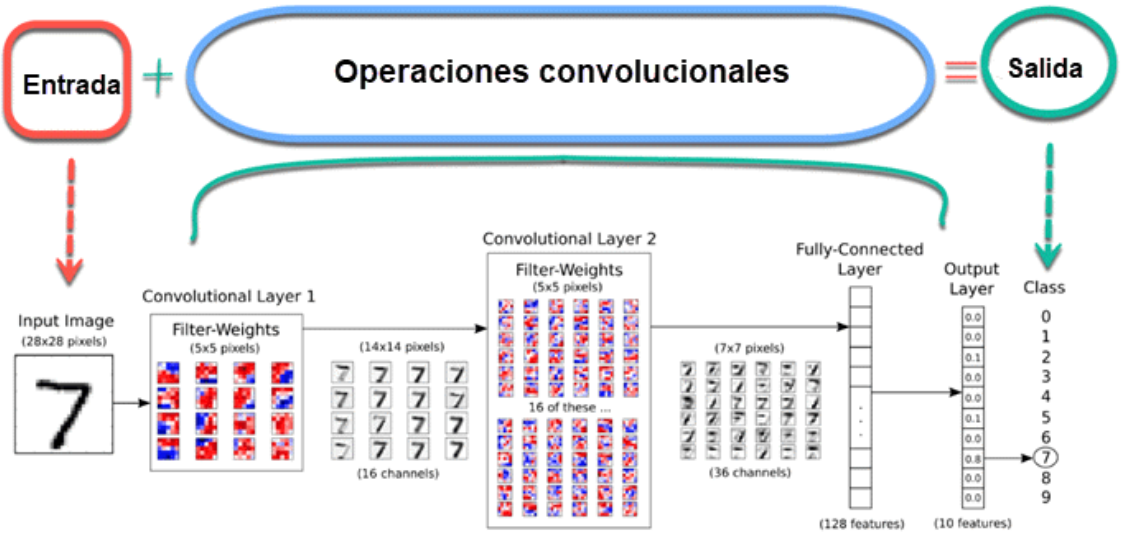

Figura 1.8: OCR usando CNN. Imagen obtenida de [24]

A partir de CNN surge otro tipo de RNA, denominado R-CNN (*Region-based Convolutional Neural Network*) que permite detectar varios objetos en una imagen y que hemos utilizado en este proyecto para realizar la tarea de seguir a una persona. Su funcionamiento se basa en la extracción de regiones de interés que pasan a través de una CNN devolviendo el resultado de una clasificación. Las regiones que pintamos según el resultado de la CNN se conocen como *Bounding Boxes* (cajas de detección).

[1] **Filtro Kernel**: En Vision Artificial, es una matriz cuadrada a la que se le aplica un producto escalar a una región de píxeles con la misma dimensión

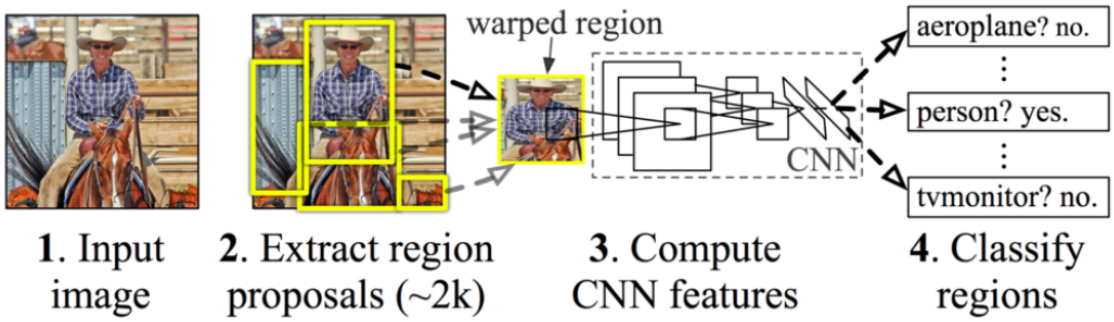

Figura 1.9: R-CNN. Imagen obtenida de [25]

Actualmente, hay disponibles varias arquitecturas de R-CNN para implementar aplicaciones de Machine Learning basada en la detección de objetos como YOLO (*You Only Look Once*) o SSD Inception (*Single-Shot Detector*). Cada una de estas redes poseen características específicas aportando distintas ventajas e inconvenientes. En el capítulo 5 realizaremos un estudio para seleccionar la arquitectura CNN óptima para usarla en los nuevos ejercicios de Robotics Academy basados en *Deep Learning*.

1.3. Educación en Robótica

La evolución de la robótica ha provocado una creciente demanda de profesionales y por tanto la necesidad de su formación. Para ello, las universidades están ofreciendo cada vez más grados y postgrados en robótica, tanto en España como en el resto de Europa y del mundo. Por otro lado han surgido plataformas en línea que imparten esta formación apoyándose en tecnologías web e Internet.

1.3.1. Grado en Ingeniería de Robótica Software (URJC, España)

La Universidad Rey Juan Carlos (España) imparte en el campus de Fuenlabrada, el grado en Ingeniería de Robótica Software (primer año: 2018) para enseñar a los futuros ingenieros interesados en este campo a programar la inteligencia de los robots del futuro.

Tal y como indica el título del grado, se trata de una carrera universitaria de programación, donde los alumnos abordan varios temas como por ejemplo Inteligencia Artificial, visión artificial, programación de lenguajes de alto nivel (C++, Python...), Seguridad, Drones, robots de servicio, robótica industrial, y mucho más. Las clases se imparten tanto en salas de ordenadores con sistemas operativos Linux (distribución Ubuntu) como en el laboratorio de Robótica, donde los alumnos pueden programar directamente robots reales. Además, se anima a los alumnos a participar anualmente en el campeonato internacional de la *Robocup*[2] puediendo usar el laboratorio como

[2]**Robocup**: Competición de robótica internacional que pretende promover la investigación y educacion en IA por todo el mundo

escenario de competición

Figura 1.10: Laboratorio de Robotica ETSIT durante la RoboCup@Home 2022

El robot usado en este Trabajo Fin de Grado y proporcionado por el grado es un modelo TurtleBot2 de Yujin. Consta de una base similar a la de un robot de limpieza con sensores de contacto, y un soporte encima que le permite incorporar un RPLIDAR, una cámara, y colocar el portátil del estudiante cuando desarrolla una solución al problema.

Figura 1.11: Robots Turtlebot2 del laboratorio de robótica

1.3.2. The Construct

The Construct es una empresa que mantiene una plataforma para aprender robótica con ROS, un ecosistema-entorno para programar robots muy utilizado en este sector. Imparten varios cursos donde los usuarios acceden a máquinas virtuales con Linux instalado y todas las dependencias adquiridas.

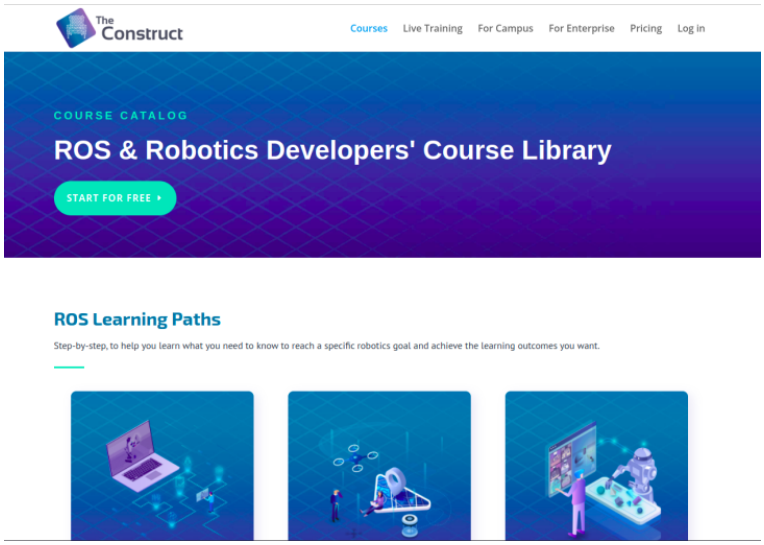

Figura 1.12: Plataforma web de The Construct[3]

1.3.3. Riders

Riders es una plataforma de competición y programación de robots. En ella los usuarios aprenden robótica programando robots en un entorno simulado usando C++ o Python como lenguajes. Mediante la competición, Riders consigue fomentar la ingeniería, la robótica y la programación.

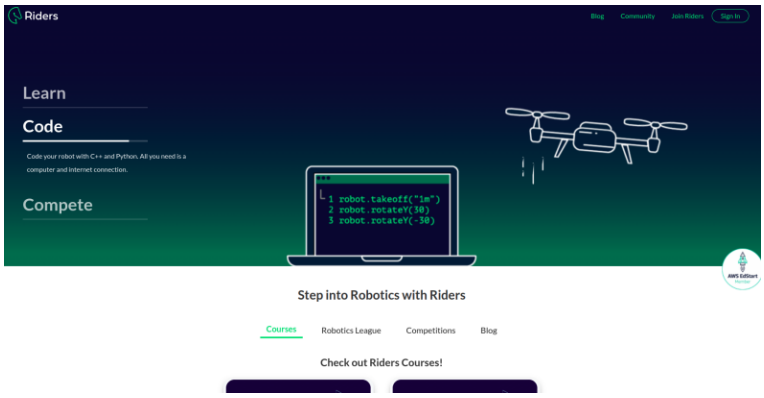

Figura 1.13: Plataforma web de Riders (riders.ai) [17]

1.3.4. Robotics Academy

Robotics Academy [12] es una colección de ejercicios y desafíos para aprender robótica. Está mantenido por la organización JdeRobot. Sus ejercicios abarcan varios temas: drones, robótica móvil, visión artificial, coches autónomos, robótica industrial, etc. Además es una plataforma de software abierto, por lo que cualquier interesado/a puede contribuir desde Github.

En sus orígenes, solo se podía ejecutar sobre un sistema operativo Linux, pero desde sus últimas versiones puede ejecutar también sobre Windows y MacOS. Gracias a la incorporación de plantillas web los ejercicios se realizan a través de un navegador estableciendo una conexión a través de WebSockets con un contenedor software (Docker) que lanza el usuario, permitiendo el despliegue de los ejercicios sobre un sistema operativo virtualizado y con todas sus dependencias ya instaladas.

Los usuarios una vez que escogen un ejercicio y lanzan el contenedor obtienen acceso a una dirección web local donde pueden empezar a programar. La plantilla web incorpora un Editor de Texto y la posibilidad de visualizar un simulador y un terminal mediante una conexión VNC con el contenedor. En la Figura 1.14 vemos un ejemplo de plantilla web correspondiente al ejercicio *Follow Line*. La programación se realiza con Python (un lenguaje fácil de entender y aprender, y muy usado en Robótica).

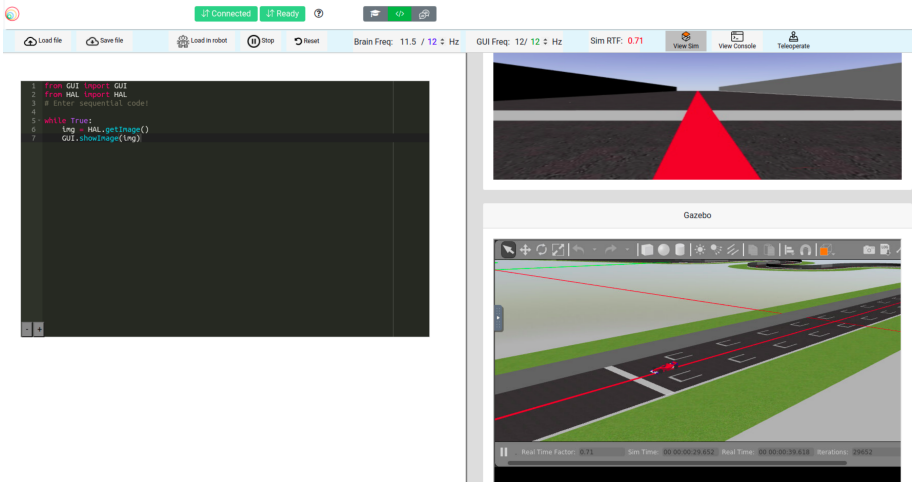

Figura 1.14: Robotics Academy (Web Template)

Una ventaja de usar Robotics Academy es que permite al usario centrarse única y exclusivamente en el algoritmo que tiene que implementar. Toda la complejidad relacionada con la comunicación con el hardware del robot: motores, laser, cámara, etc. que, por lo general se realiza mediante ROS queda encapsulado por 2 módulos de Python: HAL y GUI. De esta manera, el usuario dispone de una API de HAL (Hardware Abstraction Layer) para comandar a los actuadores o recibir información de los sensores, y una API de GUI (Graphical User Interface) para visualizar por el navegador información, como puede ser una imagen, un mapa o incluso vectores.

1.3.5. Unibotics

Unibotics[3] es una plataforma de Robótica que incorpora parte de la coleción de ejercicios de Robotics Academy. Con Unibotics, JdeRobot da un nuevo paso, permitiendo a los usuarios registrarse en un servidor donde pueden guardar sus códigos, acceder cuando deseen, y contar con la posibilidad de lanzar los ejercicios en un servidor remoto proporcionando la opción de no instalar un contenedor software en su propio ordenador.

Una vez que el usuario introduce sus credenciales y accede a su sesión, tiene una lista de ejercicios disponibles (Figura 1.15), que funcionan de la misma manera que aquellos que proporciona *Robotics Academy*. Actualmente todos los ejercicios disponibles están implementados dentro del contenedor software usando la distribución ROS Noetic como base.

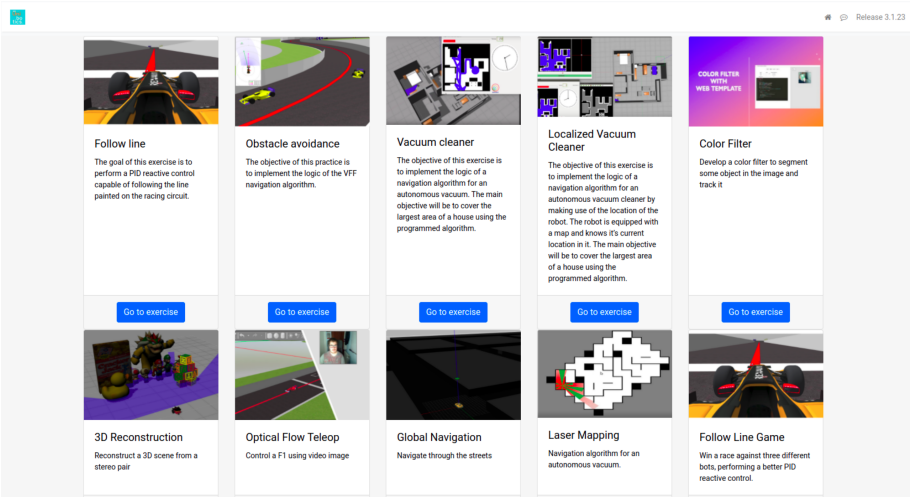

Figura 1.15: Unibotics (Menú)

Con los dos nuevos ejercicios Sigue-Persona que desarrollamos en este proyecto, pretendemos integrarlos en Unibotics a través de una nueva rama que soporta ROS2 Foxy. De esta manera, marcamos el inicio de la migración de varios ejercicios de ROS Noetic a ROS2 Foxy.

[3]**Unibotics**: `https://unibotics.org/`

Capítulo 2
Objetivos y Metodología de Trabajo

Una vez descritos en el primer capítulo la motivación y el marco en el que se encuadra este TFG (el contexto de la robótica, la Inteligencia Artificial y las plataformas educativas), en este segundo capítulo fijamos los objetivos concretos que nos hemos planteado para este proyecto.

2.1. Objetivos

La importancia de los robots de servicio ha ido aumentando en los últimos años, sobre todo en campos como la salud y el entretenimiento. Con este proyecto se pretende enseñar a realizar una tarea muy común en este sector: *seguir a un objetivo*.

El objetivo principal de este TFG es desarrollar dos ejercicios educativos en Robotics Academy para programar un robot que realice la tarea de seguir a una persona. Estos ejercicios, una vez superen ciertos tests de calidad, se incorporarán a la plataforma Unibotics. Un ejercicio consistirá en programar un robot TurtleBot2 simulado para realizar dicha tarea en un hospital, y el otro, usando el robot real. Con ello, conseguiremos fomentar la participación del alumnado o usuario con dos ejercicios donde tendrán que poner en práctica lo aprendido tanto en una simulación como en el mundo real. Para lograr esta meta, se han marcado estos subobjetivos:

1. Crear soporte simulado del robot TurtleBot2 en ROS2 Foxy. La última versión simulada del TurtleBot2 se encuentra en ROS Noetic distribuido por The Construct[1]. Profundizaremos más en el capítulo 4

2. Entender la infraestructura de Robotics Academy y aprender a desarrollar un nuevo ejercicio, diseñando su plantilla web (frontend) y sus propios ficheros específicos (exercise.py, interfaces con ROS, y módulos HAL y GUI). Veremos más en el capítulo 5

3. Diseñar un escenario en el simulador para el ejercicio Sigue-Persona simulado (incorporando un hospital de un repositorio externo de AWS) y programando un *plugin* [2] que permita controlar a una persona simulada desde el navegador web.

4. Controlar el robot real a través de un contenedor Docker. Un robot que se conecta a un portatil abre muchos dispositivos (/dev) que necesitan ser mapeados para ser utilizados desde dentro del contenedor.

[1] **Turtlebot2 (Noetic):** `https://bitbucket.org/theconstructcore/turtlebot/src/noetic/`
[2] **plugin**: Complemento que añade una funcionalidad extra o mejora a un programa

5. Diseñar una solución de referencia para cada ejercicio. De este modo demostramos la posibilidad de usar el nuevo ejercicio como recurso educativo para futuros cursos académicos en el grado en Ingeniería de Robótica Software y en otros ámbitos. Además, enseñamos un posible modo de resolver el problema Sigue-Persona en el capítulo 6

2.2. Metodología

El TFG comenzó en octubre de 2021 y finalizó en mayo de 2022. Durante estos meses se ha seguido el siguiente protocolo:

- Reuniones semanales con el tutor de TFG para comentar los avances y recibir retroalimentación. De esta manera abordábamos varios problemas buscando otras soluciones.
- Se ha usado la plataforma Slack[3] para estar en comunicación con el equipo de desarolladores de Robotics Academy y de Unibotics.
- Anotaciones semanales en un *Blog*[4] de las pruebas realizadas o logros conseguidos. De esta manera mostramos el progreso del trabajo desde sus comienzos.
- Todas las pruebas realizadas así como el código fuente del proyecto se han ido subiendo, a lo largo de estos meses, en un repositorio en Github habilitado dentro de la cuenta de *Robotics Lab URJC*[5]

2.3. Plan de Trabajo

A lo largo de estos meses, la planificación del TFG ha sido la siguiente:

1. Etapa de Entrenamiento. Se correspondió al primer mes donde se empezó a probar distintas tecnologías para elegir correctamente el tema a desarrollar: Behavior Trees, Groot, aplicaciones en ROS2, ROS Bridge, etc.
2. Inicio del TFG. Una vez conocido el tema, comenzamos a organizar el proyecto, preparando el entorno ROS, programando un *plugin* para mover una persona simulada y migrando el TurtleBot2 a ROS2 Foxy. También fue necesario aprender varias herramientas y tecnologías para su elaboración:
   - Aprender a construir imágenes Docker personalizadas.
   - Profundizar en Xacro y *plugins* a través de la página oficial de ROS2 Foxy[6].
   - Para profundizar en HTML, CSS y JavaScript se ha consultado el curso del profesor *Juan González Gomez* de la asignatura de Construcción de Servicios y Aplicaciones Audiovisuales en Internet (CSAAI) [9].
3. Programación de la infraestructura de los ejercicios.

[3]**Slack:** `https://slack.com/intl/es-es/`
[4]**Weblog**: `https://roboticslaburjc.github.io/2021-tfg-carlos-caminero/`
[5]**Repositorio TFG:** `https://github.com/RoboticsLabURJC/2021-tfg-carlos-caminero`
[6]**Ros Foxy doc**: `https://docs.ros.org/en/foxy/`

- Primero se empezó con el ejercicio Sigue-Persona simulado: se implementó el escenario del simulador Gazebo y se programó un *plugin* para mover a una persona simulada con el teclado. Después desarrollamos la plantilla web para la programación del ejercicio y la integramos en un contenedor Docker, usando como referencia otros ejercicios de Robotics Academy como por ejemplo *Color Filter* [7] o *Follow Line* [8], y, adaptándolo a ROS2.
- De manera similar, hicimos lo mismo con el ejercicio Sigue-Persona Real. Al tener que usar un robot real, contaba con todo el hardware necesario tanto por parte de mi tutor (todo tipo de cámaras para hacer pruebas) como por parte del laboratorio de Robótica de la ETSIT (el robot TurtleBot2, y los láseres RPLIDAR).

4. Programación de una solución de referencia para cada ejercicio Sigue-Persona de Robotics Academy.

5. Memoria de Trabajo Fin de Grado. La redacción de la presente memoria se realizó durante los últimos 4 meses del curso.

[7] **Color filter (foxy github):** `https://github.com/JdeRobot/RoboticsAcademy/tree/foxy/exercises/static/exercises/color_filter/web-template`

[8] **Follow Line (noetic github):** `https://github.com/JdeRobot/RoboticsAcademy/tree/master/exercises/static/exercises/follow_line/web-template`

Capítulo 3
Herramientas utilizadas

El desarrollo de los dos nuevos ejercicios de Robotics Academy ha abarcado varios campos: programación, Docker, modelado y simulación de robots, ROS y ROS2, visión artificial, Machine Learning, desarrollo web... por lo que es necesario introducir y describir las herramientas y tecnologías utilizadas.

3.1. Lenguajes de programación

3.1.1. C++

C++[1] es un lenguaje de programación diseñado en 1979 por Bjarne Stroustrup. Deriva del lenguaje C, y destaca principalmente porque incorpora el paradigma de la *Programación Orientada a Objetos* (POO). Además, incorpora una novedosa librería que facilita el desarrollo de Software de calidad llamada STL (Standard Template Library): vector, stack, map, etc. Desde sus comienzos, C++ ha tenido varias versiones que han incorporado nuevas características al lenguaje (98, 11, 14, 17...) siendo C++20 la última.

```
#include <iostream>

int main(int argc, char ** argv)
{
   std::cout << "Hello World!\n";
   return 0;
}
```

Código 3.1: Hola mundo en C++

¿Cuál ha sido la utilidad de C++ en este proyecto? La programación de un *plugin* que permite mover a una persona simulada en el simulador Gazebo usando el teclado del ordenador (versión C++17).

[1] **C++**: `https://www.cplusplus.com/`

3.1.2. Python

Python[2] es un lenguaje de programación *interpretado*[3] que destaca por su legibilidad, facilidad y soportabilidad con el paradigma POO. Posee características particulares que lo diferencian de otros lenguajes de programación como puede ser el uso estricto de indentación en bloques de código, o incluso la capacidad de crear listas mezcladas de diferentes tipos de datos. Las versión más reciente de Python es la 3.10.

```
print("Hola mundo")
```

Código 3.2: Hola mundo en Python

¿Cuál ha sido la utilidad de Python en este proyecto? El desarrollo de la infraestructura interna que da soporte a los dos nuevos ejercicios de Robotics Academy, programación de nodos de ROS (más en la sección 3.2) y ficheros de lanzamiento, creación de los módulos HAL y GUI y desarollo de las soluciones de referencia de los ejercicios (versión Python 3.8)

3.2. ROS (Robot Operating System)

ROS (Robot Operating System)[4] es un *middleware*[5] que ayuda a la programación de robots a través de varias librerías y herramientas desarrolladas por la comunidad de software libre. Entre las ayudas que proporciona este entorno/ecosistema están la abstracción del hardware, controladores de dispositivos, herramientas de visualización (rviz) y geometría espacial (transformadas, tfs) entre otros. ROS está mantenido por la fundación OpenRobotics.

El funcionamiento de ROS se basa en una arquitectura *cliente-servidor* centralizado donde un nodo principal se encarga del intercambio de mensajes entre otros nodos que forman la aplicación (remotos o locales), los cuales se comportan como publicadores o suscriptores del proceso de comunicación. El objetivo es dividir el software robótico en secciones o *nodos* que se encargan de realizar tareas específicas de manera concurrente: recogida de datos de la cámara, ejecución de una máquina de estados o árboles de comportamiento (Behavior Trees), recogida de las lecturas del láser, navegación, etc.

El intercambio de mensajes se realiza a través *topics*. Un nodo puede tener publicadores, suscriptores o ambos. Cuando un nodo publica un mensaje lo hace a través de un *topic* que gestiona el nodo maestro. Si otro nodo quiere leer ese mensaje tendrá que suscribirse al *topic* correspondiente. Por ejemplo: en nuestro robot TurtleBot2 podemos comandar velocidades enviando mensajes de tipo geometry_msgs.msg.Twist a través del *topic* /cmd_vel. En la Figura 3.1 se puede ver con más

[2]**Python**: `https://docs.python.org/3/`

[3]**Interprete** (programación): programa informático que se encarga de ejecutar las instrucciones de otro programa sin compilación previa

[4]**ROS**: `http://wiki.ros.org/es`

[5]**Middleware**: software que se sitúa entre las aplicaciones y el sistema operativo

detalle el funcionamiento de las comunicaciones en ROS.

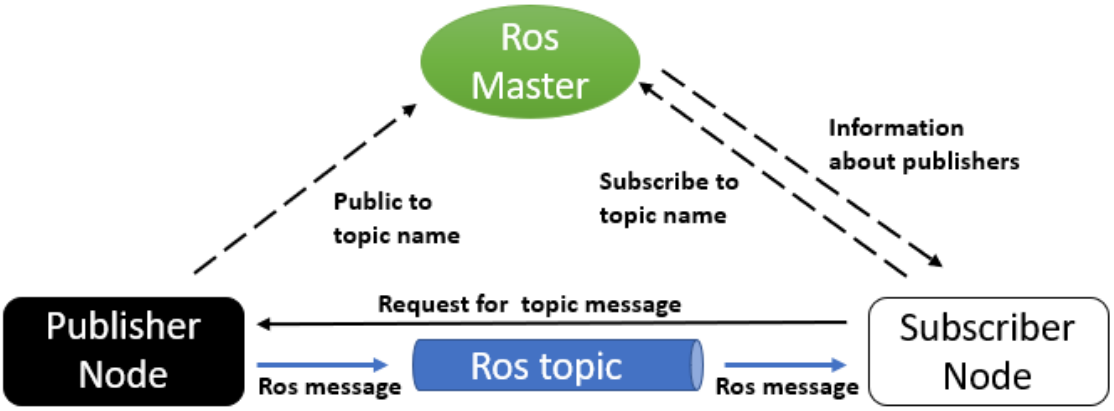

Figura 3.1: Comunicación del nodo Master con los nodos Intermedios (ROS). Imagen obtenida de [18]

Desde un principio, ROS se ha ejecutado sobre sistemas operativos de tipo Linux, aunque han ido surgiendo versiones beta para otros SOs como Windows o MacOS X. La primera distribución de ROS salió en 2010 (ROS Box Turtle) y a partir de ahí han ido surgiendo nuevas distribuciones siendo la actual y estable ROS Noetic.

Las librerías que incorpora ROS están desarrolladas en *C++* y *Python*. Con estos lenguajes de programación podemos desarrollar los nodos de los paquetes que tendrá nuestra aplicación. También incorpora un conjunto de comandos de *Shell* [6] para realizar tareas de depuración, creación, compilación, etc.

En 2016 surge ROS2, nuevo ecosistema basado en ROS que incorpora nuevas mejoras como la calidad de servicio, los *lifecycle nodes*, o seguridad mediante DDS para asegurar la autenticación de los nodos y la integridad de los mensajes. Al igual que con ROS, ha tenido varias distribuciones siendo la primera ROS2 Ardent. La versión más reciente y estable es ROS2 Foxy, que es la que hemos usado en este proyecto.

¿Cuál ha sido la utilidad de ROS2 en este proyecto?

- Crear los ficheros *interfaces* (laser.py, motors.py, camera.py, ...) que se comunican con los topics necesarios para el control del robot TurtleBot2 tanto real como simulado. Estos ficheros usan nodos con publicadores o suscriptores de distintos *topics* como /cmd_vel, /scan, /odom o /raw_image.
- Crear los paquetes de la simulación del hospital y la adaptación del TurtleBot2 simulado.
- Incorporar de Github los paquetes de ROS2 para controlar la base Kobuki del robot real, y el láser RPLIDAR.

[6] **Shell**: Programa que toma entradas del usuario a través del teclado y pasa esos comandos al sistema operativo para realizar una función específica

Durante la etapa de entrenamiento del TFG hicimos unas primeras pruebas utilizando el Robot TurtleBot2 simulado de ROS Noetic desde ROS2. Para lograr esto, usamos *ROS Bridge* [7]. Se trata de un paquete que proporciona un puente de red que permite el intercambio de mensajes entre ROS y ROS2. De esta manera, podíamos lanzar una simulación en ROS y ejecutar un programa en ROS2 (el cual se comunica con los *topics* lanzados en el nodo maestro de ROS).

3.3. Gazebo

Gazebo[8] es un simulador robótico creado por la Open Source Robotics Foundation que permite integrar escenarios realistas con robots simulados para desarrollar y probar algoritmos. Gazebo surgió en 2002 en la Universidad del Sur de California. En 2009 un ingeniero superior de investigación integró ROS y el robot PR2 en Gazebo.

Gazebo presenta características relevantes para nuestro proyecto:

- Usa motores de físicas de alto rendimiento incluido Bullet, OGRE, Simbody y DART
- Puede simular sensores y su ruido característico para asemejarlo con el mundo real (RPLIDAR, camaras Kinect, sensores de contacto).
- Se puede programar *plugins* para incorporar nuevas características y dispositivos. Gazebo incorpora su propia API para ello.
- Muchos robots tienen soporte en Gazebo como PR2, Pioneer2 DX, o TurtleBot2 y 3. También permite construir robots propios usando SDF (o URDF).

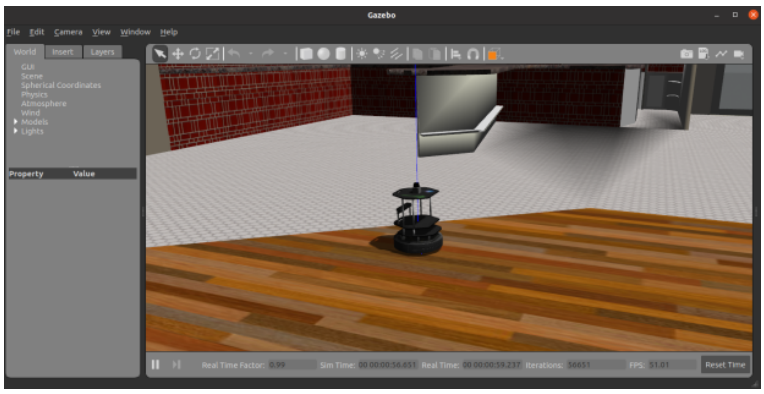

Figura 3.2: Simulador Gazebo (demostración)

¿Cuál ha sido la utilidad de Gazebo en este proyecto? Incorporar un escenario proporcionado por AWS que simula un hospital e integrar el robot simulado TurtleBot2

[7] **ROS Bridge**: `https://github.com/ros2/ros1_bridge`
[8] **Gazebo**: `https://classic.gazebosim.org/`

(sensores incluidos) adaptado para ROS2 Foxy. El nuevo ejercicio Sigue-Persona Simulado hace uso de dicho simulador. La versión utilizada ha sido *Gazebo 11*.

3.4. Modelado de robots con URDF y Xacro

3.4.1. URDF

URDF[9] *United Robotics Description Format* es un lenguaje de marcado que usa la gramática de XML para describir robots (eslabones y uniones de eslabones) y sensores (cámaras, láseres y muchos más). Se utiliza tanto para robots reales como simulados.

Por lo general el robot se divide en eslabones (*links*) y uniones entre eslabones (*joints*), de manera que se forma un árbol jerárquico de transformadas (TFs) desde un eslabón principal (*root*) hasta los eslabones terminales (*camera_link*, *laser_link*, *effector1*, ...).

Para definir un *link* hay que proporcionar un modelo visual, un modelo de colisión y un modelo inercial (similar a los videojuegos) tal y como se muestra en el Código 3.3. Después definimos un *joint* que une el eslabón que hemos definido con otro mediante relación jerárquica de padre e hijo.

```
<?xml version="1.0"?>
<robot name="turtlebot">
   <link name="base">
      <visual>
         ...
      </visual>
      <collision>
         ...
      </collosion>
      <inertial>
         ...
      </inertial>
   </link>
</robot>
```

Código 3.3: Estructura URDF de la definición de un link

¿Cuál ha sido la utilidad de URDF en este proyecto? Crear el modelo TurtleBot2 simulado para ROS2. En la Figura 3.3 podemos ver el resultado. Veremos más en el capítulo 4 que tratará del proceso de creación del modelo.

[9] **URDF:** `http://wiki.ros.org/urdf`

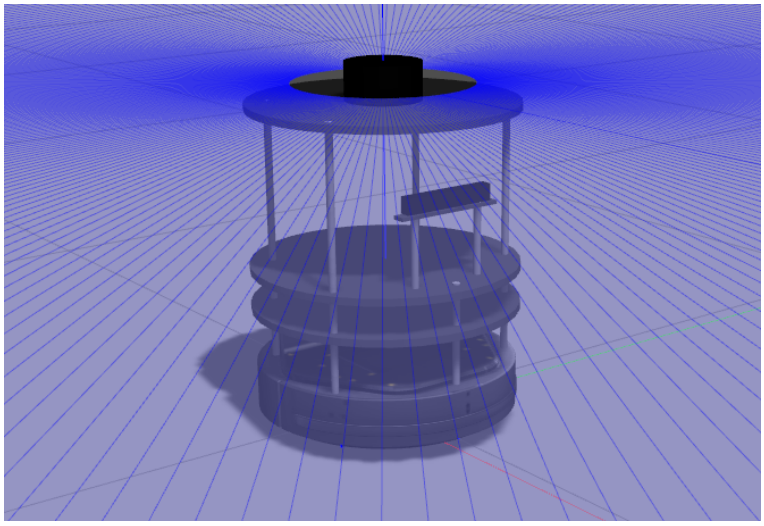

Figura 3.3: Modelo Turtlebot2 simulado (definido mediante URDF)

3.4.2. Xacro

Xacro[10] (XML Macros) es un lenguaje XML diseñado para crear macros[11] en URDF. Con Xacro conseguimos diseños URDF mucho más legibles y sencillos. De esta manera podemos estructurar el diseño de un robot en componentes más pequeños y repetitivos. Por ejemplo: para el diseño del robot TurtleBot2 creamos una macro que se pueda aplicar a todos las barras verticales que sujetan la estructura del robot.

Una vez que tenemos nuestro diseño en Xacro, podemos exportarlo a URDF mediante el paquete de ROS denominado *Xacro*. Su comportamiento es semejante al preprocesador de los lenguajes de programación como C: la macro la sustituye por el código URDF deseado teniendo en cuenta el valor de la variables que haya definido el programador.

¿Cuál ha sido la utilidad de Xacro en este proyecto? Diseñar un diseño sencillo y flexible del robot TurtleBot2. En el capítulo 4 mostraremos mejor la utilidad de Xacro aplicado a nuestro modelo.

3.5. Tecnologías web de frontend

3.5.1. HTML y CSS

HTML (HiperText Markup Language) es un lenguaje de marcado de tipo XML para el diseño de la estructura de las páginas web. Es un lenguaje que entiende el navegador web y se suele acompañar mediante una hoja de estilos que proporciona el

[10] **Xacro**: `http://wiki.ros.org/xacro`
[11] **macro**: plantilla que define un patrón

lenguaje CSS (Cascading Style Sheets) para crear el estilo gráfico de la página web.

```
<p>Esto es un parrafo en HTML</p>
```

```
p {
   background-color: #FF0000;
}
```

Código 3.4: Ejemplo de HTML y CSS: Todos los párrafos tienen el fondo rojo

¿Cuál ha sido la utilidad de HTML y CSS en este proyecto? Diseñar y dar estilo a la plantilla web de los dos nuevos ejercicios de Robotics Academy. El diseño se basó en tomar como referencia las plantillas web de otros ejercicios realizando algunos cambios e incorporando características propias (como por ejemplo, el botón de *teleoperación*).

3.5.2. JavaScript

JavaScript es un lenguaje de programación interpretado que destaca por ser utilizado en el lado del cliente, en el *frontend* de una página web que lee el navegador. Con JavaScript podemos crear páginas web dinámicas a través de la definición de eventos que provocan resultados en el *backend* (si tiene) o en la propia estructura HTML. JavaScript puede usarse para multitud de tareas: Websockets, juegos, animaciones, etc.

¿Cuál ha sido la utilidad de JavaScript en este proyecto? En las plantillas web de Robotics Academy se usa JavaScript para comunicar los eventos de la página con el fichero maestro de cada ejercicio que se llama `exercise.py`. La versión de JavaScript utilizada es ECMAScript 6.

3.6. Docker

Docker[12] es un proyecto de código abierto que nos permite desplegar aplicaciones dentro de contenedores software (máquinas virtuales ligeras que permiten ejecutar múltiples instancias de espacios de usuario) que pueden ejecutar sobre cualquier sistema operativo proporcionando portabilidad al usuario. Por ejemplo, podemos lanzar una aplicación de Linux usando un contenedor Docker, que implementa características del Kernel de Linux, sobre un SO Windows.

La creación de imágenes se realiza mediante ficheros *Dockerfile* donde el programador indica los comandos (de Unix) necesarios para la construcción y despliegue de su aplicación. Durante el desarrollo de la imagen se sigue estos pasos:

[12] **Docker**: `https://www.docker.com/`

1. Importamos una imagen base para empezar. Por ejemplo *FROM osrf/ros:foxy-desktop*

2. Definimos las variables Dockerfile y de entorno que necesitaremos con *ARG* y *ENV* respectivamente.

3. Definimos con *RUN* todos los comandos necesarios para la construcción del la imagen *Docker* lanzados desde el directorio de trabajo *WORKDIR*. Con *COPY* copiamos ficheros desde nuestro espacio de usuario al contenedor y con *CMD* indicamos el comando inicial que se ejecutará siempre que lancemos el contenedor.

¿Cuál ha sido la utilidad de Docker en este proyecto? Cuando Robotics Academy crea una nueva versión construye una imagen Docker con la instalación de todos los repositorios base y de terceros necesarios para la ejecución del entorno web en una distribución de Linux. Gracias a la ejecución de un contenedor el usuario solamente necesita instalar Docker en su SO. Con el comando *docker pull* elige la imagen que quiere descargar y con *docker run* lanza un contenedor para realizar los ejercicios siguiendo sus correspondientes instrucciones. En este proyecto, fue necesario crear una imagen particular en el que funcionara los dos nuevos ejercicios para posteriormente fusionarlos en la nueva imagen oficial RADI 4. La versión de Docker utilizada ha sido 20.10.12

En el código [3.5] vemos qué comando tendría que ejecutar el usuario para lanzar un contenedor con la última imagen de Unibotics y poder interactuar con los ejercicios de la plataforma.

```
$> docker run --rm -it -p 7681:7681 -p 2303:2303 -p 1905:1905 -p 8765:8765
   -p 6080:6080 -p 1108:1108 jderobot/robotics-academy:latest
   ./start_manager.sh
```

Código 3.5: Comando de lanzamiento de un contenedor Docker en Unibotics

3.7. Robotics Academy

Robotics Academy[13] es una plataforma web que proporciona una colección de ejercicios de Robótica. Para su ejecución usamos unos contenedores *Docker* [3.6] (imagen RADI) que lanza un servidor local (Django) proporcionando todos los ejercicios (las dependencias quedan resueltas) y permitiendo al usuario programar sus soluciones a través del navegador.

Por lo general, el usuario dispone en cada ejercicio de un editor de texto, una ventana al simulador de Gazebo y un terminal para depurar su programa. Además, dentro del

[13]**Robotics Academy**: `http://jderobot.github.io/RoboticsAcademy/`

RADI se lanzan distintos nodos de ROS o incluso el simulador dependiendo del ejercicio. La encapsulación proporcionada hace creer al usuario que todo el procesamiento ocurre en el navegador mientras que en realidad es en el contenedor. El funcionamiento y protocolo de comunicación queda reflejado en la Figura 3.4:

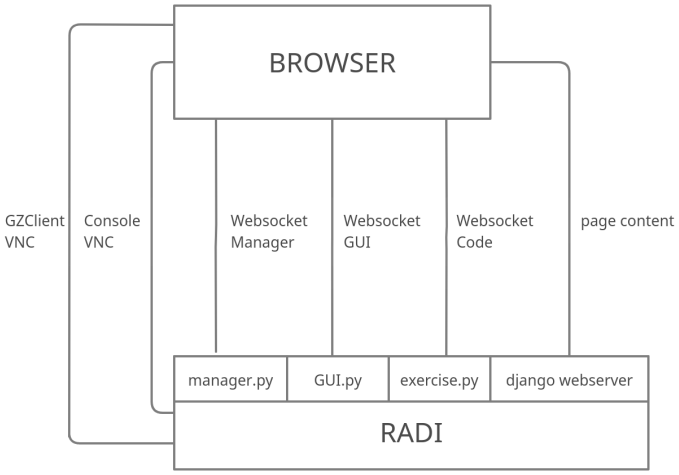

Figura 3.4: Arquitectura de Robotics Academy

Cuando lanzamos el contenedor, se ejecuta un servidor *Django* local (usa HTTP) proporcionando los ficheros web (html, css, y js) del menú de inicio; y un fichero de control y administración llamado `manager.py`. El `manager.py` se encarga de controlar la apertura, comunicación y cierre con el ejercicio que solicita el usuario a través del *WebSocket Manager* con el navegador. Cada ejercicio implementa una plantilla web específica y un fichero maestro llamado `exercise.py` que se comunica tanto con el `manager.py` como con el navegador. Este fichero proporciona la comunicación con el cerebro del robot mediante el *WebSocket Code* pudiendo de esta manera interactuar con el robot a través de la API de los módulos HAL y GUI. La API de HAL interactúa directamente con los nodos de ROS que cada ejercicio lanza de manera independiente. La visualización de los datos del robot sobre la plantilla web (como las cámaras) se realiza a través del *WebSocket GUI* que proporciona el fichero `GUI.py`. Por último, para la visualización del simulador y del terminal en el propio navegador se emplean dos conexiones VNC.

En la Figura 3.5 resumimos de un modo general la conexión existente entre el navegador y la aplicación de ROS.

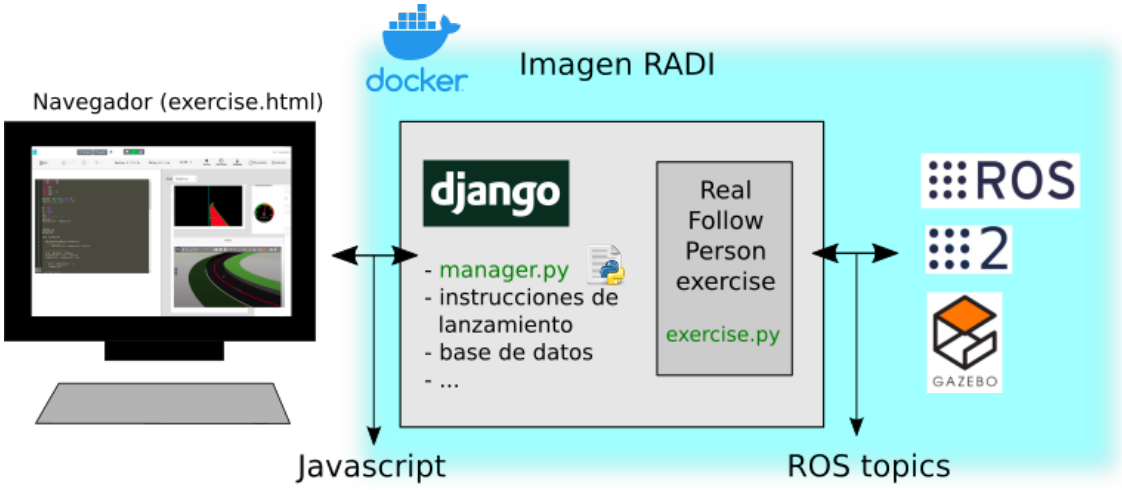

Figura 3.5: Arquitectura de Robotics Academy (II)

Actualmente, Robotics Academy dispone de dos ramas de desarrollo: una rama principal que proporciona ejercicios con ROS Noetic, y una rama secundaria (en versión beta) que usa ROS2 Foxy. Los dos ejercicios de este TFG han sido incluidos en esta última rama.

3.8. TurtleBot2

El laboratorio de Robótica de la ETSIT URJC tiene a su disposición una decena de robots TurtleBot2 para que los alumnos hagan uso de ellos en algunas asignaturas del grado. Este robot móvil, ideal para la enseñanza e investigación en robótica, consta de dos partes principales: una base Kobuki y un cuerpo de soporte al que se le ha incorporado un láser RPLIDAR y una cámara Intel RealSense.

3.8.1. Base Kobuki

La base inferior del TurtleBot2 se denomina Kobuki. Es similar a las aspiradoras de limpieza robóticas como Roomba (iRobot). Respecto al hardware presenta tres sensores de contacto (bumpers) para detectar colisiones con el entorno, odometría, leds programables, sensor de caída y un giroscopio entre otros. Alcanza una velocidad lineal máxima de 0.7 m/s y una velocidad angular de 180 deg/s. Además tiene una batería recargable que opera entre 3 y 7 horas con alta velocidad de recarga. Los drivers del Kobuki se encuentran en este repositorio de Github: *Kobuki Core*[14]. Con ROS usamos un repositorio[15] que actúa de envoltorio o *wrapper* para comunicarnos con el driver real a través de un paquete denominado *kobuki_node*

[14]**Drivers Kobuki (ROS)**: `https://github.com/yujinrobot/kobuki_core`
[15]**Wrapper Kobuki (ROS)**: `https://github.com/yujinrobot/kobuki`

Figura 3.6: Base Kobuki. Imagen obtenida de [21]

3.8.2. Cuerpot del Turtlebot2

Se trata de la base superior del robot donde el alumno puede colocar su portátil y conectarse a la base Kobuki mediante USB. Está adaptado para incorporar varios tipos de sensores como cámaras (RGB y RGB-D), láseres e incluso actuadores como un brazo robótico.

Figura 3.7: TurtleBot2. Imagen obtenida de [2]

Para este proyecto hemos necesitado tener en el cuerpo del TurtleBot2 un láser RPLIDAR A1 para evitar obstáculos y una cámara modelo Intel Realsense 3D R200 para poder realizar detección de objetos en la tarea de seguir a una persona.

3.8.3. RPLIDAR A1

Se trata de un láser de 360 grados con un alcance de 12 metros y una frecuencia de muestreo de 5.5 Hz. En ROS tenemos el siguiente repositorio de la organización SLAMTec que nos permite usar los láseres de la familia RPLIDAR: `https://github.com/Slamtec/rplidar_ros`

Figura 3.8: Láser RPLIDAR A1. Imagen obtenida de [23]

En la Figura [3.9] vemos la visualización de las lecturas de un láser con ROS a través de *Rviz*[16]:

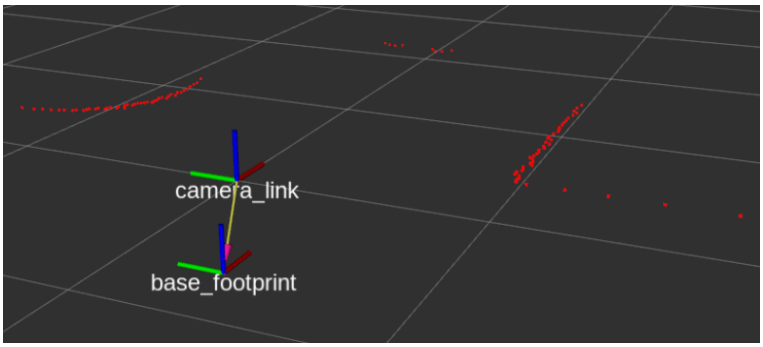

Figura 3.9: Visualización del láser Rviz

3.8.4. Cámara Intel Realsense 3d R200

Durante este proyecto hemos probado varias cámaras: Asus XTION PRO Live, Realsense D435 y T265. Debido a la inexistencia de repositorios soportados para ROS2 Foxy, Intel Realsense 3D R200 fue la cámara candidata. Sin embargo, este modelo no permitía usar la profundidad 3D en ROS, por lo que la utilizamos como

[16]**Rviz**: Visualizador 3D incorporado con ROS

una simple cámara RGB. Cualquier cámara que abra dispositivos en /dev/video* (como por ejemplo una webcam) podrá ser utilizada en el ejercicio Sigue-Persona Real. Simplemente tendremos que realizar un mapeo de puertos cuando lancemos el contenedor Docker indicado.

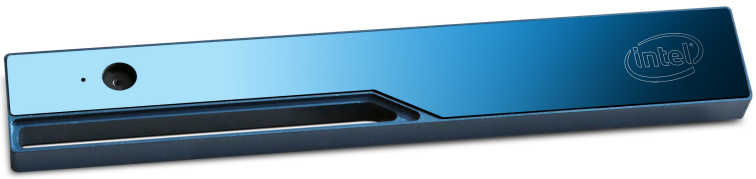

Figura 3.10: Cámara Intel Realsense 3D R200. Imagen obtenida de [15]

En la Figura 3.11, vemos la visualización del la nube de puntos 3D generada por una cámara RGB-D con ROS a través del visualizador Rviz:

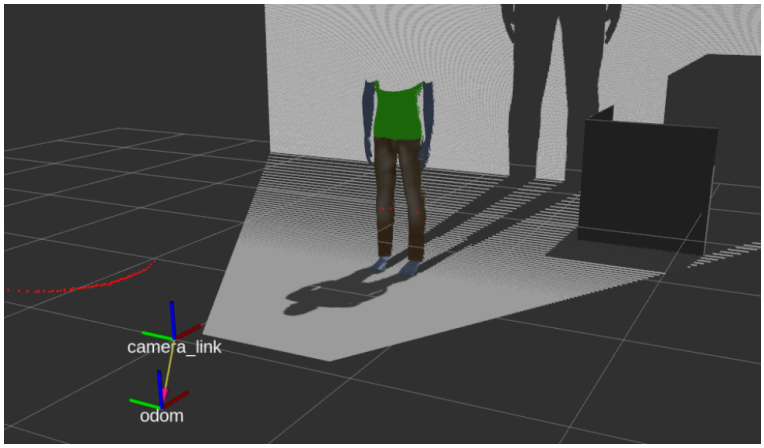

Figura 3.11: Visualización de una cámara RGBD en Rviz

Capítulo 4
Soporte de TurtleBot2 en ROS Foxy

En este capítulo se presenta el proceso de migración del robot TurtleBot2 de ROS Noetic a ROS2 Foxy. Se describirán todos los cambios relevantes realizados como puede ser la modificación de sus ficheros URDF/Xacro, su lanzamiento en Gazebo a través de los *launchers* de ROS2 o incluso su integración sobre una imagen Docker.

El soporte del robot en ROS2 Foxy supuso el primer paso del proyecto para desarrollar los nuevos ejercicios de Robotics Academy.

4.1. Robot TurtleBot2 simulado en Gazebo

El soporte más estable del TurtleBot2 se encuentra en la rama de ROS Melodic (tanto real como simulado) aunque también funciona en ROS Noetic[1]. Sin embargo, para ROS2 Foxy solamente hay un repositorio[2] que actúa de *wrapper* con el driver del robot real (veremos más en la Sección 4.2). En esta sección abarcaremos el proceso de modelado del cuerpo del robot y la creación de los ficheros de lanzamiento para Gazebo.

4.1.1. Base Kobuki

El repositorio oficial de la base Kobuki contiene un paquete denominado *kobuki_description*, que contiene los ficheros de descripción Xacro (Sección 3.4.2) y URDF (Sección 3.4.1) de la base. En estos ficheros se describe el árbol de tranformadas que existen entre todos los *links* (tramos, partes) del robot, y de esta manera permite conocer la localización de todos los marcos de coordenadas. También se añaden *plugins* para facilitar el control de movimiento del robot (controladores de velocidad, odometría ...). Sin embargo, este paquete carece de ficheros de lanzamiento para el simulador, de modo que tuvimos que diseñar nuestros propios *launchers*.

Para conseguir la representación de la base Kobuki en Gazebo hicimos un copia[3] del repositorio oficial e implementamos un nuevo paquete llamado *kobuki_gazebo*. Dentro

[1] **TurtleBot 2 (Noetic)**: `https://bitbucket.org/theconstructcore/turtlebot/src/noetic/`
[2] **Base Kobuki (foxy)**: `https://github.com/kobuki-base/kobuki_ros`
[3] **Kobuki ROS (fork)**: `https://github.com/Carlosalpha1/kobuki_ros`

se incluyeron ficheros `launch.py` para lanzar tanto el simulador como la base kobuki simulada. A continuación describiremos los ficheros más importantes del nuevo paquete implementado:

- **empty_world.launch.py**: Un punto de partida en muchas ocasiones cuando creamos un robot simulado es diseñar un fichero de lanzamiento que únicamente lance un mundo vacío en Gazebo. De esta manera, puedes incluir ese fichero en otros ficheros de lanzamiento y dividimos un problema complejo en varios subproblemas. Para lanzar Gazebo en ROS2 ejecutamos dos ficheros de lanzamientos en este orden:

  1. gazebo_ros - `gzserver.launch.py`: Lanza un servidor de Gazebo sin ventana gráfica, permitiendo ejecutar programas sin necesidad de visualizar el resultado en el simulador.
  2. gazebo_ros - `gzclient.launch.py`: Lanza un cliente de Gazebo que activa una ventana gráfica donde podemos ver el mundo solicitado.

```
def generate_launch_description():

   ld = LaunchDescription()

   pkg_gazebo_ros = get_package_share_directory('gazebo_ros')

   gazebo_server = IncludeLaunchDescription(
      PythonLaunchDescriptionSource(os.path.join(pkg_gazebo_ros, 'launch',
          'gzserver.launch.py'))
   )

   gazebo_client = IncludeLaunchDescription(
      PythonLaunchDescriptionSource(os.path.join(pkg_gazebo_ros, 'launch',
          'gzclient.launch.py'))
   )

   ld.add_action(gazebo_server)
   ld.add_action(gazebo_client)

   return ld
```

Código 4.1: kobuki_gazebo: empty_world.launch.py

- **spawn_model.launch.py**: Este fichero pasa los datos de descripción URDF de la base Kobuki a un parámetro denominado */robot_description*, publica el estado del robot, sus transformadas y ejecuta el fichero de gazebo_ros `spawn_entity.py` para visualizar el modelo en el simulador:

```
kobuki_model = Node(
   package='robot_state_publisher',
   executable='robot_state_publisher',
   parameters=[{'robot_description': robot_desc}],
   arguments=[urdf_file]
)

joint_state_publisher_node = Node(
   package='joint_state_publisher',
   executable='joint_state_publisher',
   name='joint_state_publisher'
)

spawn_entity = ExecuteProcess(
   cmd=['ros2', 'run', 'gazebo_ros', 'spawn_entity.py', '-topic',
       '/robot_description', '-entity', 'kobuki'], output='screen')
```

Código 4.2: kobuki_gazebo: spawn_model.launch.py

Podemos visualizar la base Kobuki en el simulador ejecutando los siguientes comandos:

```
ros2 launch kobuki_gazebo empty_world.launch.py &
ros2 launch kobuki_gazebo spawn_model.launch.py
```

Código 4.3: Comandos para lanzar la base Kobuki en Gazebo

En la Figura 4.1 se puede ver el resultado de la base Kobuki en Gazebo

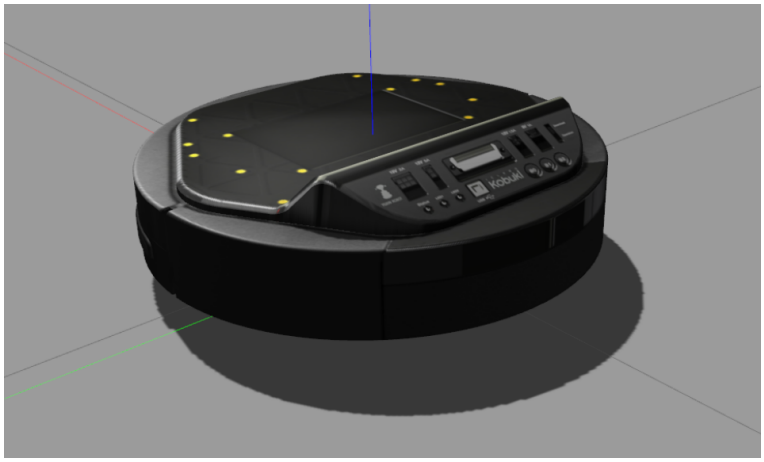

Figura 4.1: Modelo simulado Kobuki (ROS2)

4.1.2. Cuerpo del robot

Aparte de la base Kobuki, la otra mitad que caracteriza al TurtleBot2 es la *parte superior* o *cuerpo* que permite colocar, sensores, actuadores o incluso portátiles (en los robots reales). En esta sección abordaremos su modelado y simulación con URDF y Xacro.

La pregunta es ¿por qué no usamos ficheros turtlebot_description de la rama Noetic o Melodic si el lenguaje URDF es el mismo? En realidad, el paso de ROS a ROS2 conlleva cambios tanto en la manera de crear nodos, ficheros de lanzamiento y su funcionamiento interno como en el uso de URDF. El modelo TurtleBot2, al tener una gran cantidad de ficheros URDF que dependían unos de otros, y estos a su vez de otros paquetes con dependencias en ROS, no facilitaba la tarea de obtener un modelo creando únicamente ficheros de lanzamiento `launch.py` como hicimos con la base Kobuki.

La *solución* fue crear la estructura restante del robot a mano, usando la sintáxis URDF y Xacro, diseñando un modelo lo más semejante posible al real. A continuación mostraremos las fases de desarrollo. Una vez terminado el modelo del TurtleBot2, tendremos por un lado un directorio con todos los paquetes de *kobuki_base* y otro directorio con la definición del nuevo soporte creado.

Usar Xacro permitió diseñar *macros* que facilitaran la estructura y la legibilidad del modelo. Con Xacro está permitido incluir de manera sencilla, nuevos elementos en el modelo y establecer la jerarquía de transformadas entre *links* de padres a hijos (siempre es importante indicar las relaciones jerárquicas para realizar futuras operaciones basadas en marcos de coordenadas).

Primero, en un fichero `structures.urdf.xacro` definimos dos macros para crear los *links* que necesitamos y definimos las relaciones jerárquicas de padres a hijos. Las nuevas macros son *cylinder_structure* y *cube_structure*, para crear en el fichero `turtlebot2.urdf.xacro` elementos como los siguientes:

```
<xacro:cylinder_structure name="base_tick5" x="0.15" y="0.0" z="0.14"
    length="0.15" radius="0.005" parent="base_link"/>
<xacro:cube_structure name="camera_support_base" x="0.13" y="0" z="0.0975"
    x_size="0.0175" y_size="0.15" z_size="0.005" parent="middle_base_link"/>
```

Código 4.4: Creación de dos *links* usando dos nuevas macros definidas (TurtleBot2 ROS Foxy)

Después definimos en el fichero `colors.urdf.xacro` algunas macros para incluir colores en los *links* del modelo de Gazebo:

```
<xacro:macro name="create_color" params="name value">
   <material name="${name}">
      <color rgba="${value}"/>
   </material>
</xacro:macro>

<xacro:macro name="gazebo_color" params="link color">
   <gazebo reference="${link}">
      <material>Gazebo/${color}</material>
   </gazebo>
</xacro:macro>

<xacro:create_color name="Gray" value="0.5 0.5 0.5 1"/>
<xacro:gazebo_color link="base_tick1_link" color="Gray"/>
```

Código 4.5: Creación y establecimiento de un color a un link

Por útlimo, en el fichero `turtlebot2.urdf.xacro` incluimos con la macro *xacro:include* las definiciones URDF de los ficheros del paquete *kobuki_description*. Una vez llegados a este punto tendremos el robot TurtleBot2 simulado sin sensores. El siguiente paso es incorporar un láser de 360 grados y una cámara RGB-D.

Para la visualización del modelo URDF completo en Gazebo me basé en los mismos ficheros que hice con la base Kobuki: `empty_world.launch.py` y `spawn_model.launch.py` (Sección 4.1.1)

La única diferencia fue definir tres argumentos por defecto para establecer la posición inicial (xyz) del robot, y poder especificar el punto de partida cuando se quiera importar el modelo en cualquier otro mundo de Gazebo. Estos tres argumentos se pasan al nodo *spawn_entity*. A continuación podemos ver la sección referente a la posición inicial del fichero `spawn_model.launch.py`:

```
   # Set (x, y, z) default position of turtlebot2
   x_pos = LaunchConfiguration('-x', default='0')
   y_pos = LaunchConfiguration('-y', default='0')
   z_pos = LaunchConfiguration('-z', default='0')

   spawn_entity_node = Node(
      package='gazebo_ros',
      executable='spawn_entity.py',
      name='entity_spawner',
      output='screen',
      arguments=["-topic", "/robot_description", "-entity", "turtlebot2",
          "-x", x_pos, "-y", y_pos, "-z", z_pos]
   )
```

Código 4.6: Establecimiento de la posición por defecto del TurtleBot2 en el simulador

Para lanzar el robot en el simulador ejecutamos los siguientes comandos:

```
ros2 launch turtlebot2 empty_world.launch.py &
ros2 launch turtlebot2 spawn_model.launch.py
```

Código 4.7: Comandos para lanzar el robot TurtleBot2 en el simulador (ROS2 Foxy)

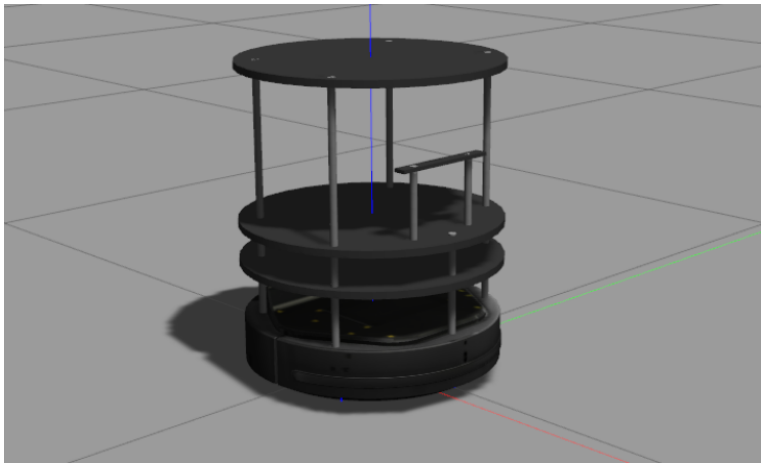

Figura 4.2: Robot TurtleBot2 simulado sin sensores (ROS2 Foxy)

4.1.3. Sensores Láser y Cámara

Para usar un láser en Gazebo utilizamos un fichero .xacro que actuará como plantilla[4] y permita posicionar el láser donde el programador quiera. En el fichero `lidar.urdf.xacro` indicamos algunas propiedades como el rango del láser, el ruido y el alcance.

Para colocar el sensor en el robot, tuvimos que incluir esta línea en el fichero `turtlebot2.urdf.xacro` donde indicamos su posición xyz y el *link* padre:

```
<xacro:set_lidar name="lidar" xyz="0 0 0.0275"
    parent_frame="upper_base_link"/>
```

Código 4.8: Colocación del láser en el robot simulado

[4]**Láser Xacro**: https://github.com/RoboticsLabURJC/2021-tfg-carlos-caminero/blob/main/turtlebot2/turtlebot2/urdf/sensors/lidar.urdf.xacro

Con la cámara RGB-D[5] hicimos lo mismo. En su fichero Xacro `camera.urdf.xacro` definimos algunas propiedades como el alcance, el tamaño de los fotogramas o el *campo de visión* (FOV).

Para colocar la cámara en el robot, tan solo teníamos que incluir esta línea en el fichero `turtlebot2.urdf.xacro` donde indicamos su posición xyz, su orientación y el *link* padre:

```
<xacro:set_camera name="camera" xyz="0.13 0 0.32" rpy="0 0 0"
    parent_frame="base_link"/>
```

Código 4.9: Colocación de la cámara en el robot simulado

Con esto, tendríamos todos los ficheros necesarios para usar el robot TurtleBot2 simulado completo. Con la incorporación de los sensores, su directorio quedó de la siguiente manera:

```
turtlebot2
├── kobuki_base
│   ├── kobuki_ros_interfaces
│   └── kobuki_ros
│       ├── Ficheros de kobuki_gazebo 4.1.1
│       └── ⋮
└── turtlebot2
    ├── launch
    │   ├── empty_world.launch.py
    │   └── spawn_model.launch.py
    ├── rviz
    ├── urdf
    │   ├── structures.urdf.xacro
    │   ├── colors.urdf.xacro
    │   ├── turtlebot2.urdf.xacro
    │   └── sensors
    │       ├── camera.urdf.xacro
    │       └── lidar.urdf.xacro
    ├── spawn.sh
    ├── CMakeLists.txt
    └── package.xml
```

Figura 4.3: Estructura de directorios completa del Turtlebot2 (ROS2 Foxy)

[5]**Cámara Xacro**: `https://github.com/RoboticsLabURJC/2021-tfg-carlos-caminero/blob/main/turtlebot2/turtlebot2/urdf/sensors/camera.urdf.xacro`

En la figura 4.4 podemos ver la evolución del proceso creativo del TurtleBot2 simulado hasta su resultado final:

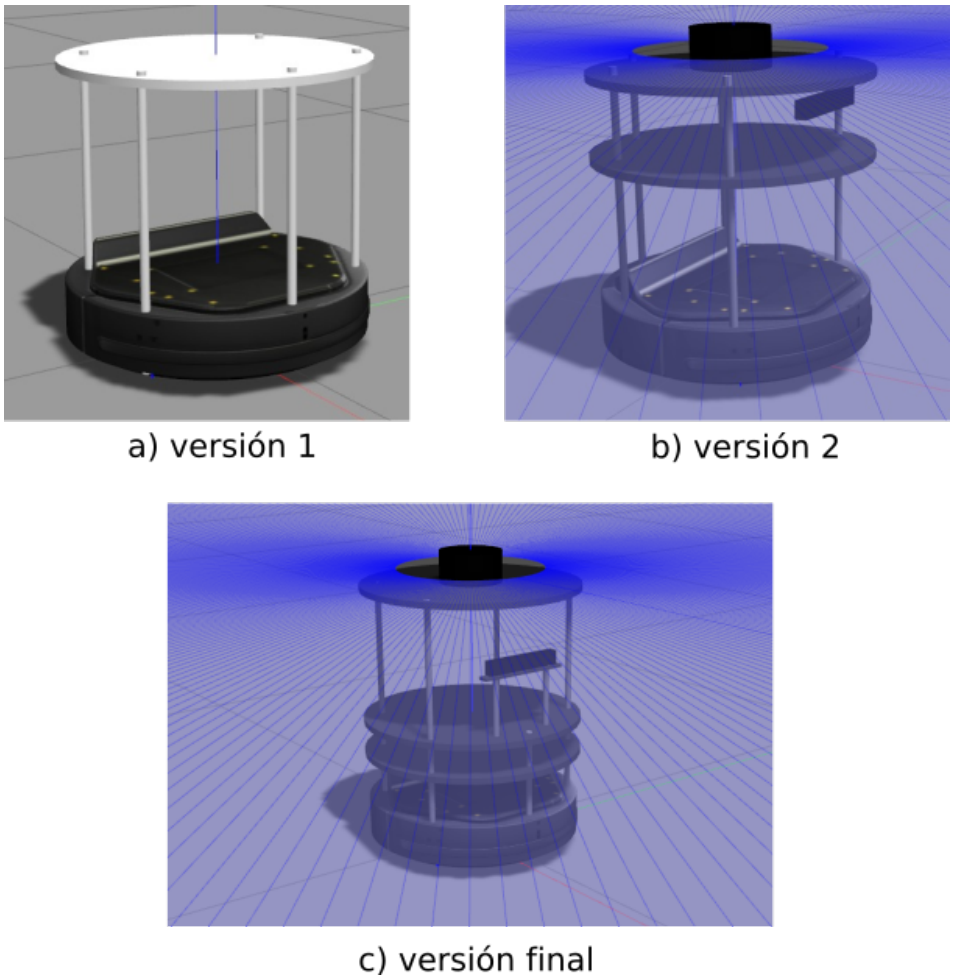

Figura 4.4: Evolución TurtleBot2 simulado

4.1.4. ROS topics

Una vez que hemos descrito la parte mecánica del robot (modelo del TurtleBot2) es necesario incorporar *drivers* que cojan datos de los sensores simulados y los ofrezcan en forma de *topics* de ROS. Además, necesitamos incorporar *drivers* que ofrezcan *topics* con comandos de movimiento para enviarlos a los motores simulados. Para ello, usaremos los *plugins* del simulador Gazebo 11.

Estos *plugins* actúan como nodos ROS de manera homóloga a como actúan los drivers del robot real. Proporcionan los mismos *topics* o parecidos compartiendo los mismos mensajes de transmisión. Los *plugins* que ofrece Gazebo se incluyen en los ficheros URDF del robot:

- En el fichero principal URDF de la base Kobuki: `kobuki_gazebo.urdf.xacro` usamos el *plugin* `libgazebo_ros_diff_drive.so` para ofrecer un *topic* llamado /cmd_vel que permite comandar velocidades lineales y angulares a los motores.
- Para el láser del fichero `lidar.urdf.xacro` del TurtleBot2 usamos el *plugin* `libgazebo_ros_ray_sensor.so` que publica sobre un *topic* llamado /scan las lecturas del láser.
- Para la cámara del TurtleBot2, usamos en el fichero `camera.urdf.xacro` el *plugin* `libgazebo_ros_camera.so` que publica sobre un *topic* llamado /depth_camera/image_raw los fotogramas que captura la cámara simulada.

En la Tabla 4.1 mostramos los *topics* relevantes del TurtleBot2 simulado y cómo poder utilizarlos (mediante publicación o suscripción):

ROS topics TurtleBot 2 simulado			
Componente	**Tipo**	**Topic**	**Mensaje**
Motores	Publicador	/cmd_vel	geometry_msgs.msg.Twist
Láser	Suscriptor	/scan	sensor_msgs.msg.LaserScan
Cámara	Suscriptor	/depth_camera/image_raw	sensor_msgs.msg.Image
Odometría	Suscriptor	/odom	nav_msgs.msg.Odometry

Cuadro 4.1: ROS Topics TurtleBot2 simulado (ROS Foxy)

4.2. TurtleBot2 real

En esta sección nos centramos en los *wrappers* para conectarnos a los *drivers* del TurtleBot2 real usando ROS2 Foxy.

El desarrollo de *Kobuki Core* en ROS2 se separó del repositorio original originando un cambio de dirección URL[6]. Para controlar la base Kobuki usamos el repositorio *Kobuki_ROS*[7] (el mismo que usamos en la sección 4.1.1) para usar el paquete *kobuki_description* que actúa de *wrapper* conteniendo los paquetes *kobuki_node* y *kobuki_keyop*:

- **kobuki node**: Es el paquete que contiene un fichero de configuración denominado `kobuki_node_params.yaml` con el que podemos conectarnos al robot real usando el puerto /dev/ttyUSB0, así como la definición de algunos marcos de coordenadas (base, odom). También contiene los ficheros de los nodos necesarios para controlar la base Kobuki y su odometría. Oculta al programador la complejidad de comunicarse directamente con el hardware (motores, leds...) del robot usando nodos ROS (mediante publicadores y suscriptores). El nodo que lanza *kobuki_ros_node* se suscribe al *topic* /commands/velocity para recibir mensajes de tipo *geometry_msgs.msg.Twist* y poder mover mediante velocidad lineal y angular la base del robot. Este directorio no tiene relación con *kobuki_description*.

[6]**Drivers Kobuki (ROS2)**: `https://github.com/kobuki-base/kobuki_core`
[7]**Wrapper Kobuki (ROS2)**: `https://github.com/kobuki-base/kobuki_ros`

- **kobuki_keyop**: Es un paquete que permite mover la base del robot con el teclado del ordenador. Se crea un nodo que publica por cada pulsación del teclado, un mensaje geometry_msgs.msg.Twist en el *topic /cmd_vel*, sin embargo, tendremos que cambiar a */commands/velocity* si queremos controlar el robot real. Este paquete puede ayudar en un principio a comprobar que tenemos conexión con el hardware del robot.

Para usar el láser RPLIDAR A1 en la distribución Foxy se ha usado el repositorio *rplidar_ros*[8] del usuario de Github *allenh1* en su rama para ROS2, cuyo fichero de lanzamiento `rplidar.launch.py` permite recoger lecturas con el sensor.

La cámara IntelRealsense R200 no tenía soporte para ROS2 y por lo tanto, no podíamos aprovecharnos de sus características (profundidad). De modo que hemos usado el paquete de ROS2 *v4l2-camera* que sirve para publicar sobre un *topic* llamado /image_raw las imágenes que captura la cámara a una frecuencia de 60 Hz.

Con los drivers de la base Kobuki, el láser y la cámara tenemos todo lo necesario para usar la distribución Foxy en el robot real. En la Tabla 4.2 mostramos los ROS topics que usa el robot. Como se puede ver, algunos topics han cambiado, de modo que lo hemos tenido en cuenta para la creación de los ejercicios.

ROS topics TurtleBot 2 real			
Componente	**Tipo**	**Topic**	**Mensaje**
Motores	Publicador	/commands/velocity	geometry_msgs.msg.Twist
Láser	Suscriptor	/scan	sensor_msgs.msg.LaserScan
Cámara	Suscriptor	/image_raw	sensor_msgs.msg.Image
Odometría	Suscriptor	/odom	nav_msgs.msg.Odometry

Cuadro 4.2: ROS Topics TurtleBot2 real (ROS Foxy)

Cuando publicamos en */commands/velocity* el nodo de la base Kobuki (fichero `kobuki_ros.cpp` del paquete *kobuki_node*) se suscribe al topic y comanda directamente las velocidades al hardware: *kobuki_.setBaseControl(msg->linear.x, msg->angular.z)*

4.3. Integración en el RADI 4

Para integrar el TurtleBot2 simulado y real en Robotics Academy se ha empezado creando una rama externa de desarrollo[9] donde se ha ido realizando varias pruebas con el contenedor Docker. Para su integración hemos usado los ficheros *Dockerfile* del RADI 4 (Robotics Academy Docker Image) donde hemos especificado los comandos necesarios para su instalación y soporte

[8] **rplidar_ros (ROS2)**: `https://github.com/allenh1/rplidar_ros`
[9] **RADI-prueba**: `https://github.com/Carlosalpha1/RoboticsAcademy/tree/test-radi`

En el repositorio de terceros CustomRobots, subimos el soporte del robot[10] tanto real como simulado, con algunas dependencias necesarias: *Kobuki_ros* modificado, *kobuki_ros_interfaces* y el diseño del cuerpo del robot. Por tanto, en el Dockerfile clonamos la rama *foxy-devel* de dicho repositorio. Además, instalamos algunos paquetes extras que son necesarios para la compilación del espacio de trabajo:

```
# Follow Person: Turtlebot2 dependencies
RUN apt-get update && apt-get -y --quiet --no-install-recommends install \
    ros-$ROS_DISTRO-ecl-build \
    ros-$ROS_DISTRO-diagnostic-updater \
    ros-$ROS_DISTRO-kobuki-core \
    ros-$ROS_DISTRO-ecl-errors \
    ros-$ROS_DISTRO-ecl-exceptions \
    ros-$ROS_DISTRO-ecl-geometry \
    ros-$ROS_DISTRO-ecl-linear-algebra \
    ros-$ROS_DISTRO-ecl-sigslots \
    ros-$ROS_DISTRO-joint-state-publisher \
    ros-$ROS_DISTRO-v4l2-camera \
  && apt-get -y autoremove \
  && apt-get clean autoclean \
  && rm -rf /var/lib/apt/lists/{apt,dpkg,cache,log} /tmp/* /var/tmp/*
```

Código 4.10: Instalación de dependencias para el TurtleBot2 (Dockerfile.base)

```
# Custom Robot Repository
RUN mkdir -p /opt/jderobot && cd /opt/jderobot && \
  git clone -b $ROS_DISTRO-devel
      https://github.com/JdeRobot/CustomRobots.git

# Adding RPLIDAR ROS
RUN cd /opt/jderobot/CustomRobots && \
  git clone https://github.com/allenh1/rplidar_ros.git -b ros2 && \
  cd rplidar_ros/launch && \
  sed -i "$(grep -n serial_port rplidar.launch.py | cut -d: -f1)
      s/\/dev\/ttyUSB0/\/dev\/ttyUSB1/g" rplidar.launch.py
```

Código 4.11: Instalación de dependencias para el TurtleBot2 (Dockerfile)

Para el láser hemos cambiado en el fichero `rplidar.launch.py` el puerto /dev/ttyUSB0 por /dev/ttyUSB1 usando el comando de Linux *sed* ya que la base Kobuki usa /dev/ttyUSB0 por defecto.

Una vez instaladas las dependencias y construida la imagen Docker, se han de tener en cuenta los dispositivos conectados en el portátil del usuario para el lanzamiento

[10]**Custom Robots TB2**: `https://github.com/JdeRobot/CustomRobots/tree/foxy-devel/turtlebot2`

del contenedor. Para indicar al contenedor qué dispositivos del sistema operativo queremos integrar dentro del sistema virtualizado usaremos el parámetro *–device*. Cuando usemos el robot real, tendremos que seguir las siguientes reglas en el siguiente orden (preferiblemente):

- **Base Kobuki**. Al encender la base Kobuki y conectarla al portátil, su conexión abrirá el dispositivo /dev/ttyUSB0.
- **RPLIDAR A1**. Al conectar su cable USB *después* de conectar la base Kobuki se habilitará el dispositivo /dev/ttyUSB1.
- **Cámara Intel Realsense R200**. Su conexión abre 6 dispositivos /dev/video[0-5]. Nosotros nos quedamos con /dev/video4 cuya salida se muestra en el espacio de color RGB (comprobado con este comando [11]). Como se ha integrado en Robotics Academy para las cámara que abran el dispositivo /dev/video0 tendremos que usar un remapeo: /dev/video4:/dev/video0.

El comando para lanzar el RADI 4 usando la cámara R200 será el siguiente (para otras cámaras el remapeo puede ser distinto):

```
$> docker run -it --rm --device /dev/ttyUSB0 --device /dev/ttyUSB1 --device
   /dev/video4:/dev/video0 -p 8000:8000 -p 2303:2303 -p 1905:1905 -p
   8765:8765 -p 6080:6080 -p 1108:1108 jderobot/robotics-academy:4.3.0
   ./start.sh
```

Código 4.12: Lanzamiento del RADI 4 con el robot real

Si usamos el robot simulado, no necesitaremos los puertos USB físicos citados anteriormente. De manera opcional podremos usar el parámetro *–device /dev/dri* para habilitar la aceleración gráfica y mejorar así el rendimiento en FPS de la simulación.

En el siguiente capítulo, nos centramos en la construcción de los nuevos ejercicios que usan respectivamente el soporte del robot TurtleBot2 simulado y del TurtleBot2 físico que hemos descrito en este capítulo.

[11] **cvlc v4l2:///dev/video4**

Capítulo 5
Ejercicios Sigue-Persona en Robotics Academy

En este capítulo profundizamos en el proceso creativo de la infraestructura de los dos ejercicios Sigue-Persona para Robotics Academy. Cada uno de ellos tiene dos partes: a) una plantilla Python en la que se ejecuta el ćodigo fuente del usuario (`exercise.py`) y b) una página web (`exercise.html`) que incluye tanto el editor en línea como la interfaz gráfica preparada para ese ejercicio. Ambas partes componen el ejercicio permaneciendo en comunicación entre sí a través de *Websockets* como vimos en la Sección 3.7. El objetivo es proporcionar al usuario una plantilla web óptima con la que pueda desarrollar su algoritmo cómodamente.

5.1. Entorno simulado de un hospital

La primera tarea fue integrar un escenario de Gazebo para el ejercicio Sigue-Persona simulado. El escenario candidato que elegimos fue un Hospital debido a las siguientes ventajas:

1. El robot se enfrenta a un entorno complejo (paredes, obstáculos, varias personas...).

2. La tarea Sigue-Persona tiene lugar en un entorno en el cuál tiene sentido verlo en el mundo real. Los robots en el ámbito de la salud están en continua integración y más desde el año 2020.

De modo que incorporamos el siguiente escenario de Gazebo que proporciona AWS (Amazon Web Service) en uno de sus repositorios de Github [1]:

[1] **aws hospital**: `https://github.com/aws-robotics/aws-robomaker-hospital-world`

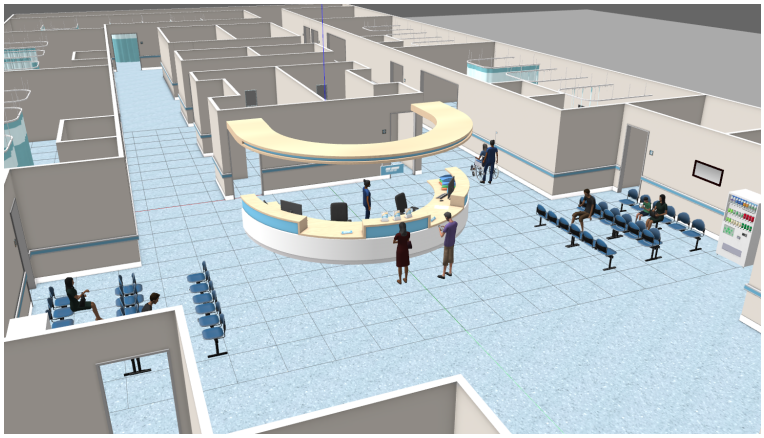

Figura 5.1: Hospital de AWS en Gazebo

El repositorio proporcionaba varios ficheros `.world` con distintas versiones del Hospital: solo planta baja, una planta y dos plantas. Elegimos por comodidad la primera.

5.2. Teleoperador

La meta final es que el usuario que use la plantilla web pueda controlar manualmente a una persona del Hospital para que el robot pueda seguirla, por tanto, el siguiente paso es integrar un modelo de Gazebo que pueda desplazarse por el escenario. Para ello teníamos que desarrollar un *teleoperador*.

El primer punto de partida era integrar una persona en el nuevo entorno simulado, por lo que accedimos a este repositorio: `https://github.com/osrf/gazebo_models` que incorpora una librería de modelos para Gazebo e incorporamos el modelo *person standing* en el repositorio de Robotics Academy de terceros Custom Robots.

Figura 5.2: Persona simulada en Gazebo

Ahora bien, el modelo es *estático*, carece de capacidad de desplazamiento, por lo que fue necesario desarrollar un *plugin* para Gazebo que permitiera controlarlo o que pudiera desplazarse a través de una ruta que eligiera el programador. De modo que en el mismo paquete donde teníamos los ficheros de lanzamiento del hospital diseñamos el *plugin* (escrito en C++) que denominamos `libpersonplugin.so` para incorporarlo en el fichero `model.sdf` (similar a URDF) de la persona. En este enlace se puede ver el código fuente[2]. Como punto de partida, tomamos como referencia un *plugin* de una persona simulada que hizó *Pedro Arias* en su TFM[3]

El nuevo *plugin* proporciona dos funcionalidades:

1. Comunicación remota para el control manual por teclado. La intención es que el usuario se comunique con el modelo simulado, por lo tanto, se ha diseñado un *socket* de comunicaciones para dicha tarea.

2. Establecimiento de una ruta por defecto y capacidad de incorporar nuevas rutas. Esta última funcionalidad no es necesaria para el ejercicio, además de que puede suponer cierta molestia al usuario, pero no se descarta su utilidad para un futuro. Básicamente, dado un vector de tuplas de tipo <float, float, int>, los dos primeros elementos indican la posición X e Y y el último parámetro apunta al siguiente punto de paso, permitiendo implementar una ruta en un bucle infinito (el último punto de paso tiene que apuntar al primero).

5.2.1. Comunicación remota

En el propio fichero `person.cpp` se crearon dos hilos (threads): uno actuaría como servidor de un socket de comunicaciones que usaría el protocolo de transporte UDP (no

[2]**person plugin**: `https://github.com/JdeRobot/CustomRobots/blob/foxy-devel/amazon_hospital/hospital_world/src/person.cpp`

[3]**TFM Pedro Arias**: `https://github.com/RoboticsLabURJC/2021-tfm-pedro-arias`

está orientado a la conexión y es más rápido) y otro hilo se encargaría de actualizar la posición del modelo. Dentro del socket se implementó un protocolo de comunicación que entendiera el servidor, el cual se comporta únicamente como receptor de los mensajes del cliente. Los mensajes (de 3 caracteres) que puede recibir son:

- **"UVF"** (User Velocity Forward). El modelo se mueve hacia delante.
- **"UVB"** (User Velocity Backward). El modelo se mueva hacia atrás.
- **"UAR"** (User Angular Right). El modelo gira hacia la derecha.
- **"UAL"** (User Angular Left). El modelo gira hacia la izquiera.
- **"US-"** (User Stop). El modelo se detiene.
- **"A–"** (Autonomous). El modelo pasa a modo autónomo. Sigue la ruta establecida (actualmente desactivada).

Pero ¿dónde entra en juego el cliente? El fichero `exercise.py` incorpora un socket de comunicación UDP que se conecta al servidor del *plugin* a través del puerto 36677. Además, el exercise.py actúa como servidor de un WebSocket en comunicación con la plantilla web. Cuando el usuario haga click en el botón "Teleoperate"de la página web del ejercicio, el fichero de eventos de JavaScript envía a través de un Websocket (usa el puerto 1905) las teclas pulsadas para que el `exercise.py` mande los comandos correspondientes al *plugin*. Al igual que en la comunicación *plugin*-`exercise.py`, se implementó un protocolo de comunicación para `exercise.py`-`exercise.html`. Los mensajes que puede recibir son:

- **"#teleop_true"**. Activa la teleoperación. A partir de ese momento, el usuario puede pulsar los botones "awsdx". Envía un mensaje **"US-"** al plugin.
- **"#teleop_false"**. Desactiva la teleoperación. Pasa a modo autónomo (si estuviera activado). Envía una mensaje **"A–"** al plugin.
- **"#key_a"**. Envía un mensaje **"UAR"** al plugin.
- **"#key_d"**. Envía un mensaje **"UAL"** al plugin.
- **"#key_w"**. Envía un mensaje **"UVF"** al plugin.
- **"#key_s"**. Envía un mensaje **"UVB"** al plugin.
- **"#key_x"**. Envía un mensaje **"US-"** al plugin.

En la Figura 5.3 se puede ver un esquema que resume la comunicación existente entre la interfaz del navegador web `exercise.html` (incorpora el fichero `ws_code.js` que es el manejador de eventos del menú superior de la plantilla web) con el *plugin*:

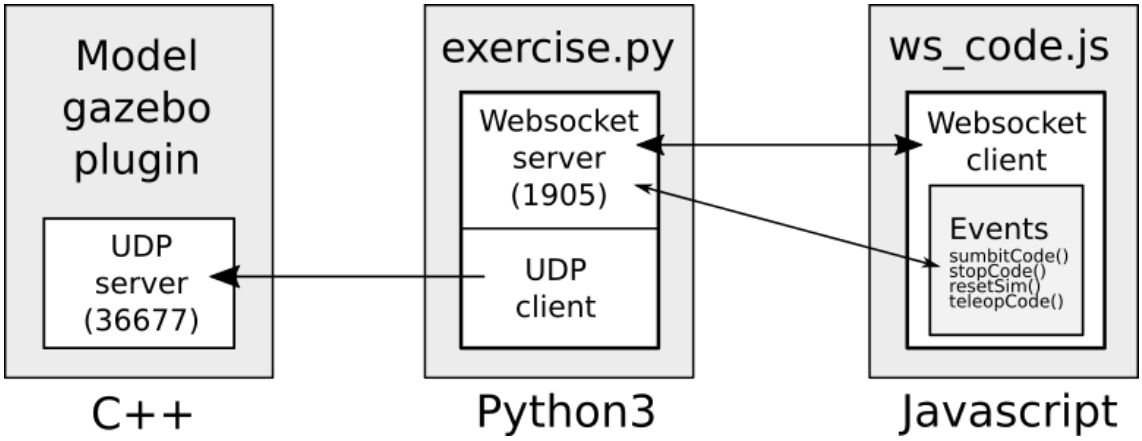

Figura 5.3: Comunicación en dos pasos del teleoperador de la persona simulada

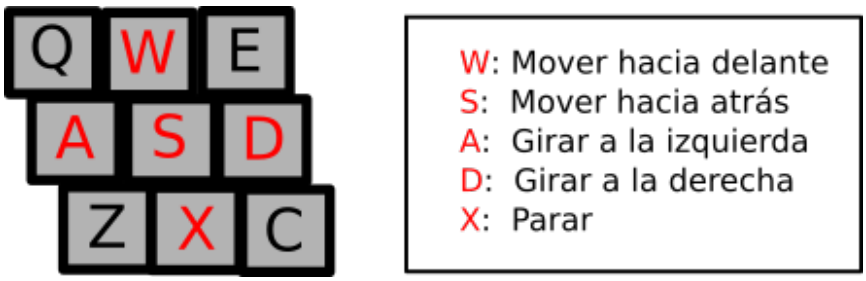

Figura 5.4: Controles del Teleoperador de la persona simulada

5.3. Plantillas Python

Todos los ejercicios de Robotics Academy poseen un fichero denominado `hal.py` que proporciona el módulo HAL (Hardware Abstraction Layer) y permite al usuario controlar el robot protagonista del ejercicio. Dicho módulo usa unos ficheros plantilla de Python que se encuentran en un directorio llamado *interfaces* que nos permite comunicarnos con nodos de ROS. El fichero `hal.py` se utiliza desde la plantilla Python, `exercise.py`, de cada ejercicio en Robotics Academy.

Para los dos nuevos ejercicios hemos tomado como referencia un fichero `hal.py` y el directorio *interfaces* de otros ejercicios como *Follow Line* o *Color Filter* y lo hemos actualizado a ROS2 incorporando las funciones necesarias para poder controlar el robot tanto real como simulado. Como plantillas *interfaces* utilizamos:

- La cámara a través de `camera.py`.
- Los motores del robot a través de `motors.py`.
- La odometría a través de `pose3d.py`.

- El láser a través de `laser.py`.
- La RNA a través de `ssd_detection.py` (nueva parte de la plantilla).

Los cambios más importantes de ROS a ROS2 aplicados en los ficheros interfaces y `hal.py` fueron el modo de construcción de nodos y la creación de los publicadores y suscriptores.

En la nueva parte de la plantilla Python (`ssd_detection.py`), para utilizar una R-CNN creamos una clase *BoundingBox* que permite diseñar fácilmente marcos de detección y una clase *NeuralNetwork* para actuar de *wrapper* de la red neuronal. La integración de R-CNN se llevó a cabo utilizando la librería de Visión Artificial OpenCV que posee una función llamada cv2.dnn.readNetFromTensorFlow(model, config) que permite usar una RNA pasándole sus ficheros de configuración.

La elección de una R-CNN que pueda detectar objetos se basó en este criterio: tiene que ejecutar de manera óptima en una CPU o en un contenedor Docker sin aceleración gráfica, para permitir al usuario la opción de programar cómodamente y poder disfrutar de la experiencia. Para ello, hicimos un previo estudio de los FPS (fotogramas por segundo) cuando se usan dos modelos diferentes de RNA en una aplicación de Visión Artificial: YOLO a través de Darknet ROS y SSD Inceptión V2.

Primero probamos un paquete *fork*[4] del repositorio oficial de *Darknet ROS* para ROS2 Foxy que incluía un fichero de lanzamiento *yolov4-tiny.launch.py* que ejecutaba una RNA profunda con menos capas que la original, provocando un aumento de rendimiento pero menor precisión. La ventaja de usar Darknet ROS es que indicas en un fichero de configuración el *topic* sobre el que se publican las imágenes de OpenCV [5] y automáticamente la Red Neuronal procesa la imagen y publica los resultados en tres *topics*: `/darknet_ros/bounding_boxes`, `/darknet_ros/found_object` y `/darknet_ros/detection_image`.

Mientras ejecutábamos Darknet ROS lanzamos este comando para medir los FPS y volcar los datos en un fichero:

```
ros2 topic hz /darknet_ros/detection_image > darknet_ros_hz
```

Después probamos la velocidad de procesamiento de la red Inception de SSD (cuyos ficheros de configuración obtuvimos de este repositorio[6]) mientras ejecutábamos un programa en Python [5.1] y mostrábamos las cajas de detección (*Bounding Boxes*) en pantalla. Al igual que hicimos con *Darknet ROS* volcamos los resultados en otro fichero.

[4]**Darknet ROS**: `https://github.com/Ar-Ray-code/darknet_ros/tree/foxy/darknet_ros/`

[5]para la cámara IntelRealsense R200 usamos este comando para publicar la imagen sobre un topic: **ros2 run v4l2_camera v4l2_camera_node –ros-args -p video_device:=/dev/video4"**

[6]**ficheros SSD**: `https://github.com/iitzco/OpenCV-dnn-samples/tree/master/tensorflow`

```
cap = cv2.VideoCapture(4)
net = NeuralNetwork()

start_time = time.time()
rate = 1
counter = 0

while True:
   ret, image = cap.read()

   if ret:
      detections = net.detect(image)

      # Process detection ...

      counter += 1
      if (time.time() - start_time) > rate:
         print("FPS: ", counter / (time.time() - start_time))
         counter = 0
         start_time = time.time()

cap.release()
```

Código 5.1: Programa para medir los FPS para SSD Inception V2

Una vez realizadas las 2 mediciones comprobamos los resultados mediante una gráfica de Python (usando el módulo *matplotlib*). El resultado fue el siguiente:

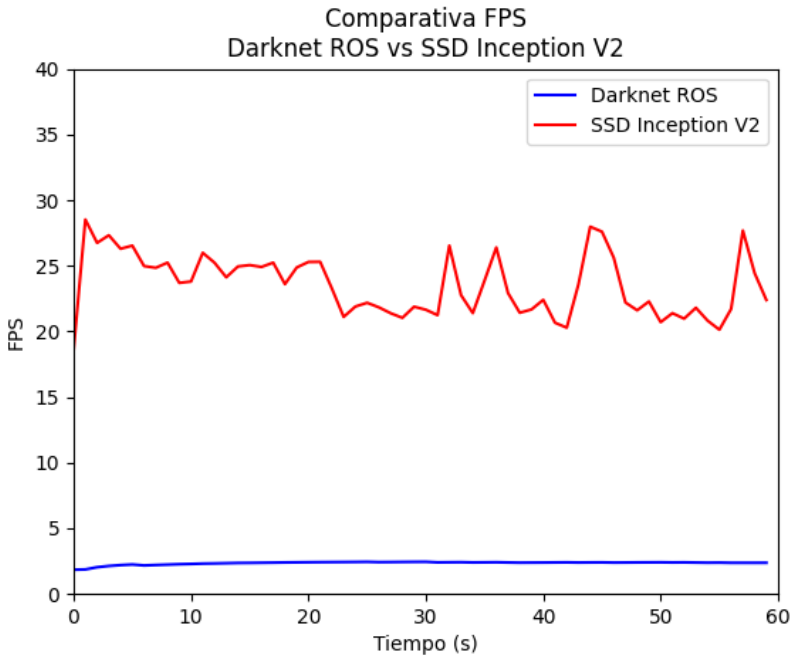

Figura 5.5: Comparativa FPS entre Darknet ROS y SSD Inception

Como vemos en la Figura 5.5, Darknet ROS no funciona de manera óptima para portátiles sin GPU (media de 2.5 fps). En cambio, SSD Inception sorprende con una velocidad de unos 25 fotogramas por segundo de media. Por tanto la elección de SSD se ve clara, sin embargo, estos resultados no serán los esperables cuando el usuario lance un contenedor Docker. El contenedor tendrá que lanzar ROS, un escenario de Gazebo y varios módulos de Python de manera concurrente (incluido VNC para visualización remota), pero con *SSD Inception* habremos conseguido proporcionar un ejercicio de Deep Learning con mayor agilidad de procesamiento.

Una vez elegido SSD Inception como RNA diseñamos la clase *Bounding Box*. que tiene los siguientes atributos:

- **id**: Es un número entero que identifica un tipo de objeto.
- **class_id**: Es una cadena de texto que identifica un tipo de objeto. Con el atributo *id* forman una pareja (clave - valor) que se puede observar en un fichero que importamos llamado *coco_labels.py* donde se registran todos los tipos de objetos que puede detectar la red neuronal:

```
LABEL_MAP = {
    0: "unlabeled",
    1: "person",
    2: "bicycle",
    3: "car",
    4: "motorcycle",
    5: "airplane",
    6: "bus",
    7: "train",
    8: "truck",
...
}
```

- **score**: Es un número en coma flotante que va de 0 a 1 que indica la probabilidad de que el objeto detectado clasificado por la red neuronal *coincida* con el objeto real. Su elección es causa de una selección como el objeto con mayor porcentaje de la lista LABEL_MAP
- **xmin** e **ymin**: Indican las coordenadas (x, y) del extremo superior izquierdo de la caja de detección.
- **xmax** e **ymax**: Indican las coordenadas (x, y) del extremo inferior derecho de la caja de detección.

En la clase *NeuralNetwork*, utilizamos los ficheros que definen la red neuronal Inception: `ssd_inception_v2_coco.pb` y `ssd_inception_v2_coco.pbtxt`. Usamos la función cv2.dnn.readNetFromTensorFlow(model, config) y creamos un método llamado *detect(self, img)* que encapsula la llamada al modelo RNA para realizar una detección

sobre una imagen.

Una vez listos los ficheros *interfaces*, creamos la API de HAL en `hal.py` cuya API se comunica con los ROS Topics descritos la Sección 4.1.4 utilizando las plantillas Python. El módulo HAL tiene las siguientes funciones:

- **setV(velocity)**. Permite ordenar velocidades lineales. Internamente usa la interfaz *motors* que publica sobre los *topics* que comandan velocidades a los motores. En el robot simulado usamos el *topic* /cmd_vel y en el robot real /commands/velocity.
- **setW(velocity)**. Permite comandar velocidades angulares (radianes). Usa también la interfaz *motors* para publicar sobre los mismos *topics* citados anteriormente dependiendo del ejercicio.
- **getLaserData()**. Devuelve una lista de 180 lecturas del láser a través de la interfaz *laser* que se suscribe al *topic* /scan. En el robot simulado, el láser 360 grados del fichero de configuración URDF está colocado de tal manera que el ángulo 0 apunta hacia delante del robot y en sentido anti-horario, por lo tanto fue necesario retornar 180 valores del láser en el orden que mostramos en la Figura 5.6. De manera similar, tuvimos que realizar el mismo paso con el robot real.

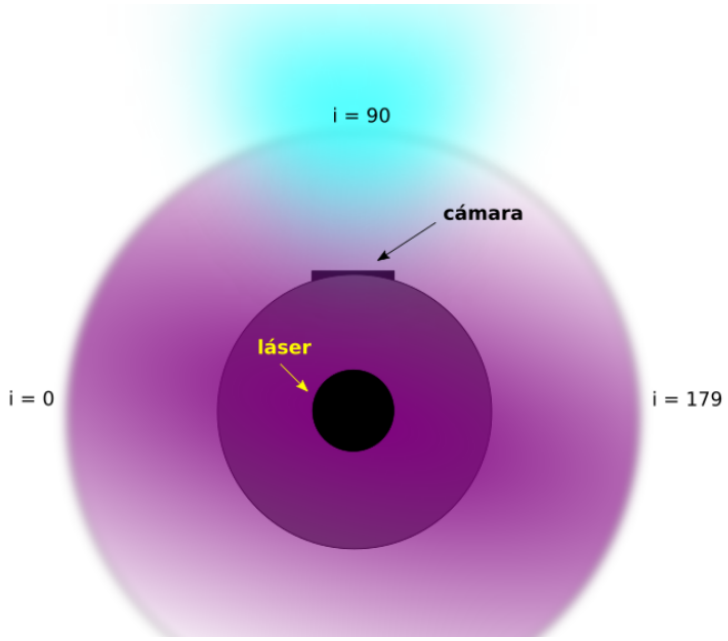

Figura 5.6: Láser TurtleBot2

- **getImage()**. Devuelve una imagen en formato OpenCV a través de la interfaz *camera*. En el robot simulado tenemos que suscribirnos al *topic* /depth_camera/image_raw y en el robot real al *topic* /image_raw
- **getPose3d()**. Obtiene la posición actual del robot a través de la interfaz *pose3d* suscrita al *topic* /odom

- **getBoundingBoxes(img)**. Dada una imagen realiza una llamada al modelo de red neuronal para realizar una detección y devolver una lista de **Bounding Boxes**.

```
class HAL:
   def __init__(self):
      ...

   def setV(self, velocity):
      self.motors.sendV(velocity)

   def setW(self, velocity):
      self.motors.sendW(velocity)

   def getLaserData(self):
      values = self.laser.getLaserData().values
      return values[90:0:-1] + values[0:1] + values[360:270:-1]

   def getImage(self):
      return self.camera.getImage().data

   def getPose3d(self):
      return self.odometry.getPose3d()

   def getBoundingBoxes(self, img):
      rows = img.shape[0]
      cols = img.shape[1]
      detections = self.net.detect(img)
      bounding_boxes = []
      for detection in detections:
         bounding_box = BoundingBox(
            int(detection[1]),
            LABEL_MAP[int(detection[1])],
            float(detection[2]),
            detection[3]*cols,
            detection[4]*rows,
            detection[5]*cols,
            detection[6]*rows)
         bounding_boxes.append(bounding_box)
      return bounding_boxes
```

Código 5.2: Módulo HAL en Sigue-Persona Simulado

Finalmente, registramos el módulo HAL en el fichero `exercise.py` para que el usuario pueda utilizarlo a través del editor de texto del ejercicio.

5.4. Plantillas web

De la misma manera que todos los ejercicios de Robotics Academy poseen un fichero `exercise.py` también poseen un fichero `exercise.html` que se encarga de proporcionar la página web en el navegador para que el usuario pueda realizar el ejercicio. Con los conocimientos adquiridos en HTML, CSS y JavaScript tomamos como referencia algunas plantillas web de otros ejercicios para diseñar las nuestras.

En la plantilla del ejercicio Sigue-Persona Simulado ha sido necesario incorporar un editor de texto para programar, una ventana que muestra el simulador de Gazebo, otra que muestra una consola de comandos para depurar las soluciones de los usuarios, y un *canvas* para visualizar los fotogramas capturados por la cámara simulada. Además, se incorporó en el menú de control de la plantilla el botón de Teleoperacion que permite controlar a la persona a la que tenemos que seguir en el escenario del hospital con el teclado (Código 5.3).

```
<button id="teleop_button" type="button" onclick="teleopCode()" ... />
```

```
var activate_teleop = false;
var key_pressed = "";

// Function to teleoperate a model
function teleopCode() {
   activate_teleop = !activate_teleop;
   if (activate_teleop) {
      teleop_btn.style.background = '#BEBEBE';
   }
   else {
      teleop_btn.style.background = 'whitesmoke';
   }
   var message = "#teleop_"+activate_teleop;
   websocket_code.send(message);
}

// Key events to move the model
window.onkeydown = (e) => {
   if (activate_teleop) {
      key_pressed = e.key;
      websocket_code.send("#key_"+key_pressed);
   }
}
```

Código 5.3: Integración del botón de teleoperación en la plantilla web del ejercicio Sigue-Persona Simulado

En cuanto a los demás elementos de la página, se realizaron algunas modificaciones como el tamaño del *canvas* de la cámara (Código 5.4) o la ventana del simulador.

```
<div id="visual">
<!-- Canvas -->
   <h3 class="output_heading">Camera Image</h3>
   <canvas id="gui_canvas"></canvas>
</div>
```

```
#gui_canvas {
    margin-left: 0%;
    height: 379px;
    width: 506px;
    border: 3px solid #555;
}
```

Código 5.4: Personalización del canvas de la cámara en el ejercicio Sigue-Persona simulado (`exercise.html` / `gui.css`)

En la Figura 5.7 podemos observar el resultado final de la plantilla web del ejercicio.

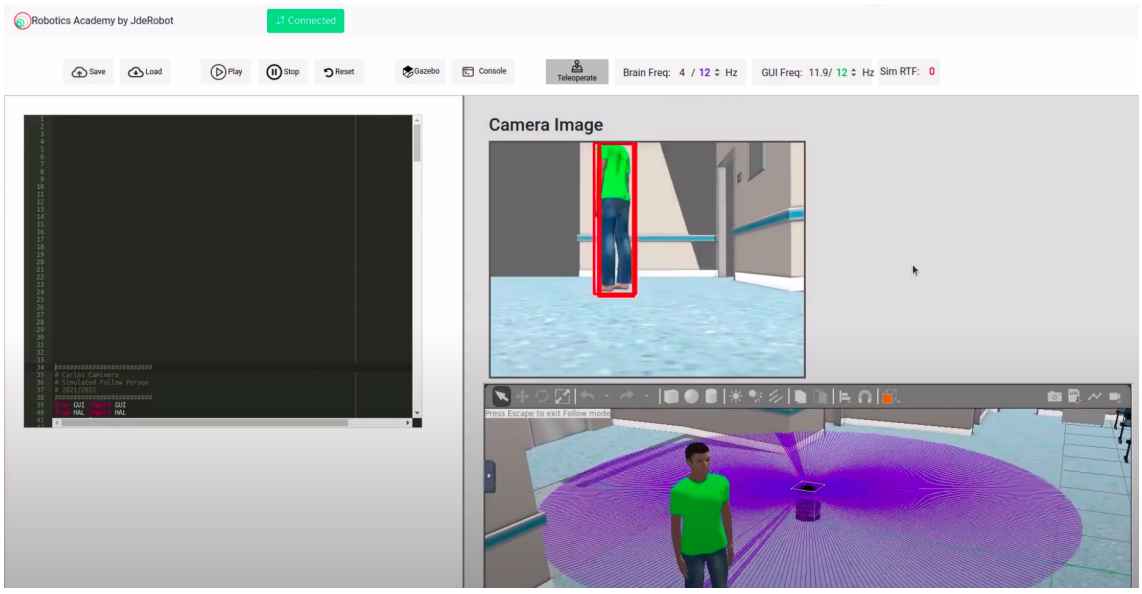

Figura 5.7: Plantilla web del ejercicio Sigue-Persona Simulado

En el ejercicio Sigue-Persona Real solamente necesitamos un editor de texto, una ventana para visualizar los fotogramas de la cámara y un terminal para depurar. Al no haber simulador, evitamos una conexión VNC extra, y sus elementos de frontend correspondientes. Para aprovechar la plantilla, aumentamos el tamaño del *canvas* de la cámara a través del fichero `gui.css`. En la Figura 5.8 vemos el resultado la plantilla web del ejercicio.

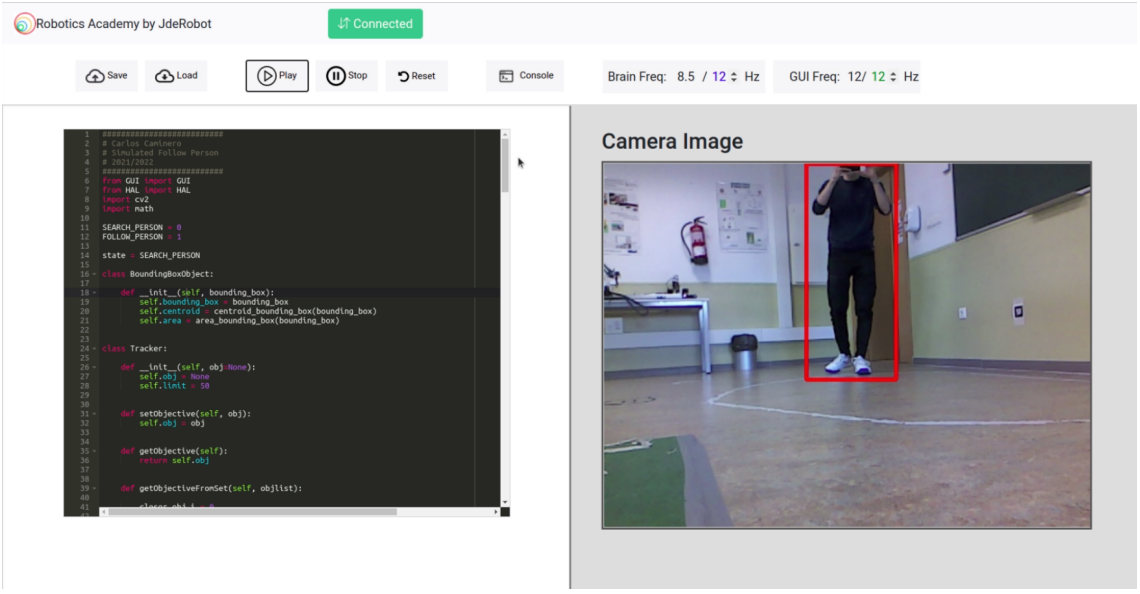

Figura 5.8: Plantilla web del ejercicio Sigue-Persona Real

Una vez terminada la infraestructura de los dos ejercicios creamos la documentación de cada uno de ellos para la página web de Robotics Academy proporcionando las instrucciones de lanzamiento y API, junto con la teoría (explicada en profundidad el capítulo 6. Los enlaces son los siguientes:

- Sigue-Persona Simulado.
  `http://jderobot.github.io/RoboticsAcademy/exercises/MobileRobots/follow_person`
- Sigue-Persona Real.
  `http://jderobot.github.io/RoboticsAcademy/exercises/MobileRobots/real_follow_person`

Capítulo 6
Soluciones de referencia

En este capítulo abordamos una solución de referencia para los ejercicios Sigue-Persona Simulado y Sigue-Persona Real. Con los dos ejercicios el usuario podrá poner a prueba su algoritmo. Se pretende poner en práctica la experiencia del programador robótico cuando desarrolla una aplicación robótica. El primer paso es probar su programa en una simulación (Sigue-Persona Simulado). Una vez que su solución es bastante robusta, pasa al siguiente nivel, probándolo en el mundo real (Sigue-Persona Real). El programador se dará cuenta de que es muy probable que el mismo programa en simulación no funcione exactamente igual con un robot real. Si su solución es bastante buena y robusta, solamente tendrá que ajustar algunos parámetros y tener en cuenta ciertos factores (como por ejemplo la luz ambiental o el rendimiento de CPU) y habrá logrado su objetivo.

6.1. Solución Sigue-Persona Simulado

La solución que resuelve la aplicación de *Seguir a una persona* la podemos dividir en 3 subobjetivos: a) detección mediante redes neuronales y seguimiento mediante un Tracker, b) desarrollo de un algoritmo de evitación de obstáculos y c) creación de una máquina de estados.

6.1.1. Detección mediante RNA y creación de un Tracker

Para detectar a la persona usamos la función del módulo HAL llamada *getBoundingBoxes*. Esta función realiza una inferencia del modelo de red neuronal para detectar todos los objetos posibles dada una imagen de entrada y devuelve una lista de objetos de cajas de detección (explicados en la Sección [5.3]). Crearemos una función llamada *draw_bounding_box(img, bbox, color=(23, 230, 210), thickness=2)* con la que podremos dibujar sobre la imagen una caja de detección dada como entrada. En este caso usaremos el color verde (0, 255, 0) e iteraremos sobre todos las cajas de detección obtenidas (Figura 6.1)

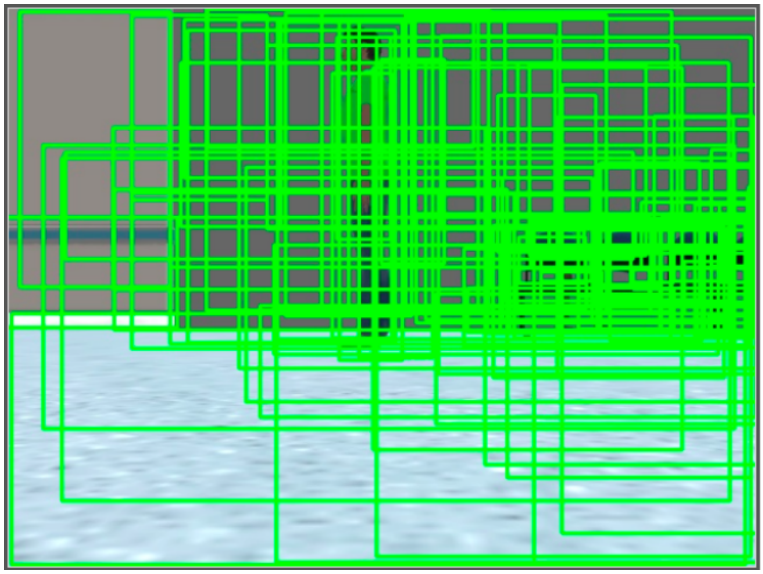

Figura 6.1: Detección mediante SSD sin filtro

Un modelo de red neuronal genera una puntuación o *score* para cada objeto detectado dependiendo de su entrenamiento. Por ejemplo, en una imagen donde puede aparecer una persona y una escultura humana, es probable, que la persona detectada tenga una puntuación de un 90 % y la escultura un 70 % debido a su parecido. Está en la labor del programador filtrar por *score* para desechar los falsos positivos. Para ello, generamos una función que llamaremos *bounding_boxes_by_score(bounding_boxes, score_limit)* que filtre aquellas cajas de detección que superen una puntuación límite. Debido a la naturaleza de este modelo neuronal preentrenado, las óptimas detecciones se han comprobado empíricamente que funcionan con una puntuación superior a 0.3 (el rango va de 0 a 1). Aplicando el filtro obtenemos el siguiente resultado (Figura 6.2)

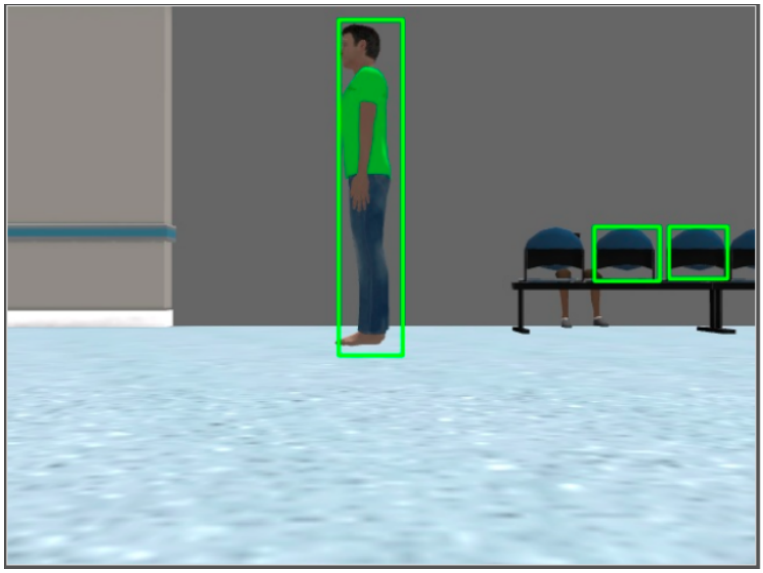

Figura 6.2: Detección mediante SSD filtrando la puntuación de clasificación

Tal y como vemos en la Figura 6.2, tras el filtro de puntuación obtenemos una imagen donde hay tres cajas de detección detectadas: una persona y dos sillas. Para quedarnos solo con las personas aplicaremos un *segundo filtro*, en la cual nos fijaremos en la clase del Bounding Box. Recordemos que el objeto Bounding Box tenía un atributo *id* que correspondía a un número entero y un atributo *class_id* que era la traducción de dicho número a una cadena de texto (ej: 1 - "person", 2 - "bicycle"). Crearemos una función llamada *bounding_boxes_by_name(bounding_boxes, name)* al cual dada una lista de objetos detectados pasaremos como segundo parámetro de entrada el nombre de la clase que queremos filtrar (en este caso "person"). Aplicando el filtro obtenemos una correcta detección de personas (Figura 6.3):

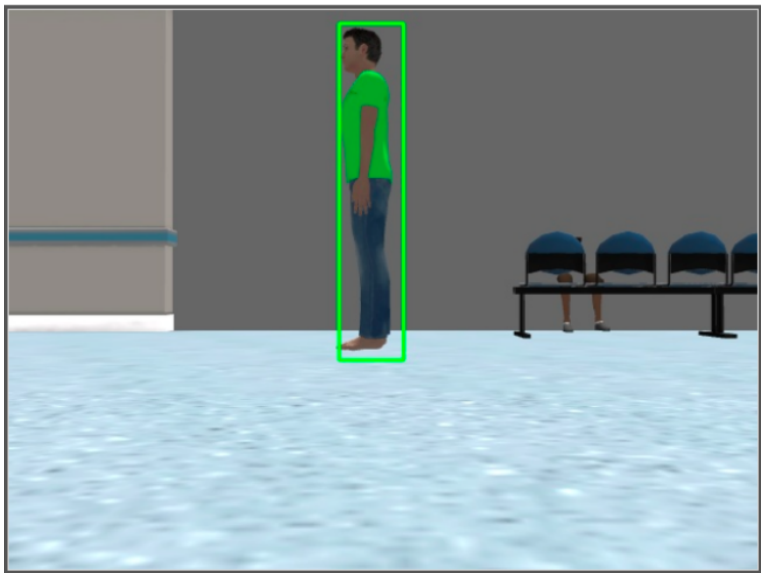

Figura 6.3: Detección mediante SSD filtrando la puntuación y la clase

Como último filtro y muy recomendable, debemos rechazar aquellas detecciones que no superen un *área* determinado. Con este filtro, evitamos que el robot se confunda con personas que se encuentren a una distancia lejana o con un falso positivo. Para ello, crearemos una función llamada *bounding_boxes_by_area(bounding_boxes, min_area)* al cual dada una lista de objetos detectados pasaremos como segundo parámetro de entrada el área mínimo que debe filtrar.

El siguiente paso es crear un *Tracker* de seguimiento. Nuestro *tracker* se basa en la distancia de los centroides de las cajas de detección del fotograma actual con el candidato del fotograma anterior. Recordemos que una caja de detección tenía otros 4 parámetros más: x_{min} e y_{min} correspondían a las coordenadas (x,y) del extremo superior izquierdo de la caja de detección, y x_{max} e y_{max} correspondían a las coordenadas (x,y) del extremo inferior derecho. Para obtener el centroide aplicamos la siguiente fórmula:

$$C_x = \frac{x_{min} + x_{max}}{2}$$
$$C_y = \frac{y_{min} + y_{max}}{2}$$

Creamos una clase llamada *BoundingBoxObject* que tiene 3 atributos: La caja de detección correspondiente, su centroide y su área. El área lo obtenemos de la siguiente manera: $A = (x_{max} - x_{min})(y_{max} - y_{min})$.

Posteriormente, creamos una clase llamada *Tracker* que proporcionará una capa de abstracción al usar los siguientes métodos que definimos:

- **setObjective(self, obj)**: Establece el objetivo BoundingBoxObject de seguimiento.
- **getObjective(self)**: Devuelve el objetivo *BoundingBoxObject* actual de seguimiento.
- **getObjectiveFromSet(self, objlist)**: Dada una lista o conjunto de objetos *BoundingBoxObject*, devuelve el objetivo candidato que más se acerca al anterior, y lo actualiza. El candidato seleccionado será aquel cuyo centroide esté más cerca del actual, no supere una distancia límite(50) con él, y que la diferencia de área entre el *BoundingBoxObject* anterior y el actual sea menor a 30000. Para calcular la distancia entre centroides usaremos la Distancia Euclídea:
$$d = \sqrt{(C'_x - C_x)^2 + (C'_y - C_y)^2}$$

En la Figura 6.4 se puede ver con más detalle en qué se basa este procedimiento: se compara cada centroide con el centroide candidato del fotograma anterior y se elige aquel que cumpla los requisitos descritos anteriormente.

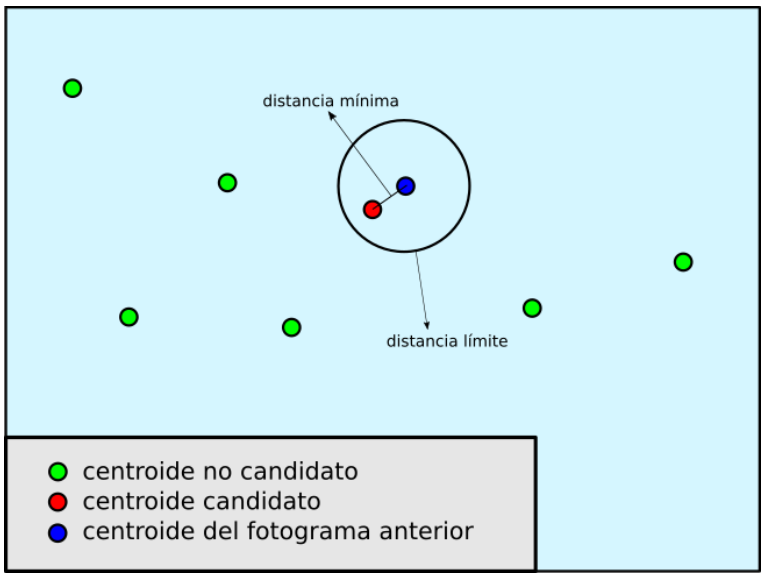

Figura 6.4: Modo de obtención del centroide candidato

Al principio del programa inicializamos una instancia de tipo Tracker, y, en cada iteración del bucle principal llamamos al método *getObjectiveFromSet(self, objlist)* para obtener un *BoundingBoxObject* que dibujamos en la imagen de color rojo. El resultado queda de la siguiente manera:

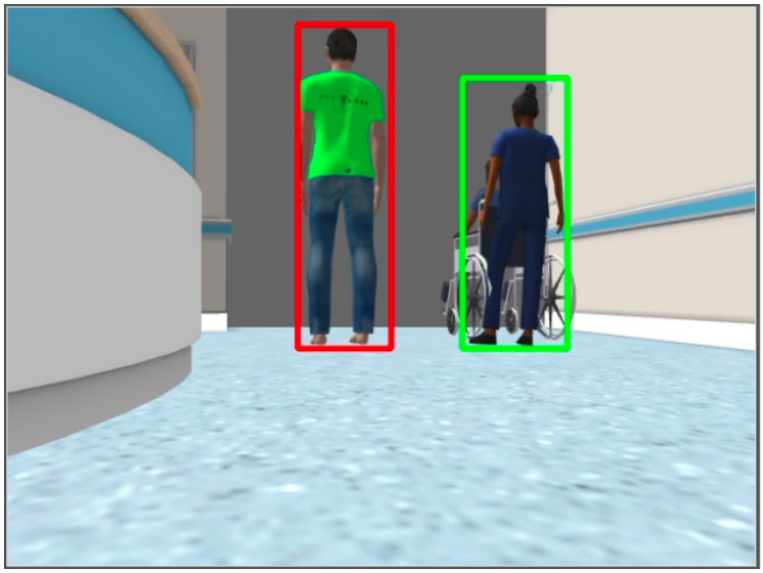

Figura 6.5: Uso del Tracker para no perder al objetivo

Como criterio de selección de la persona a la que seguiremos, deberá cumplir los 3 filtros indicados anteriormente, aparecer en el segundo tercio de la imagen (dividiremos la imagen en 3 y analizaremos la mitad) y ser aquel *BoundingBoxObject* con un área mayor que el resto.

6.1.2. Algoritmo de navegación VFF

Seguir a una persona en un mundo vacío es sencillo, pero cuando el entorno presenta obstáculos, es importante tener ese riesgo en cuenta:

- ¿Qué ocurre si la persona que seguimos gira por un pasillo a la derecha? ¿El robot sufriría una colisión con la pared antes de girar?
- ¿Qué ocurre si el robot se está acercando a la pata de una mesa? ¿Y si está pasando cerca de otra persona a la que ha descartado en el proceso de Tracking?

El algoritmo de *Campo de Fuerzas Virtuales (VFF)* está muy indicado para esas situaciones. Se basa en la suma vectorial de un vector de atracción y de repulsión originando un vector resultante que indica la nueva dirección de desplazamiento. La suma vectorial estará influenciada por el valor de unos pesos α y β aplicados a los vectores de atraccion y repulsión respectivamente, pudiendo de esta manera dar más

valor a uno que otro.

El *vector de atracción* será aquel que desde el robot apunta al objetivo. Al usar una cámara en la que no tenemos en cuenta la profundidad, establecemos que el módulo del vector será siempre 2 metros. Para obtener el ángulo nos basamos en el Campo de Visión (FOV) horizontal de la cámara. Al ser una cámara simulada, en su fichero `camera.urdf.xacro` (tal y como vimos en el capítulo 4) indicamos que FOV fuera 1.04 radianes (60 grados). Por lo tanto, 0 grados se corresponderá al centro de la imagen y los extremos a los ángulos -30 y 30. Para conocer el ángulo con el objetivo necesitaremos conocer el ancho de la imagen, y la posición de la coordenada x del centroide. La ecuación es la siguiente:

$$\alpha = \frac{H_{FOV} \cdot C_x}{w} - \frac{H_{FOV}}{2}$$

Conociendo el ángulo (α) y módulo (m) podremos obtener las componentes x e y del vector de atracción A:

$$A_x = m \cdot sin(\alpha)$$
$$A_y = m \cdot cos(\alpha)$$

El *vector de repulsión* se calculará a partir de todas las lecturas del láser. Cada obstáculo presente en el rango del láser provocará un vector en sentido contrario cuyo módulo estará influenciado por una constante y será inversamente proporcional a la distancia con el obstáculo (es decir, cuanto más cerca, mayor será el módulo). Sabiendo la constante (K) y la distancia (d) de cada lectura del láser (n lecturas), la ecuación del vector de repulsión (R) es la siguiente:

$$\begin{aligned} R_x &= \sum_{i=1}^{n} \left(\frac{K}{d_i}\right)^2 \cdot cos(\alpha) \\ R_y &= \sum_{i=1}^{n} \left(\frac{-K}{d_i}\right)^2 \cdot sin(\alpha) \end{aligned}$$

En la ecuación vemos cómo la componente R_y va en dirección contraria al desplazamiento del robot. La potencia de 2 se aplica para evitar mínimos locales.

El *vector final (F) o resultante* lo obtenemos sumando las componentes de los vectores de atracción y repulsión. Ambos multiplicados por dos constantes α y β respectivamente. La ecuación queda de esta manera:

$$\begin{aligned} F_x &= \alpha \cdot A_x + \beta \cdot R_x \\ F_y &= \alpha \cdot A_y + \beta \cdot R_y \end{aligned}$$

Queda en la labor del programador ajustar empíricamente los parámetros α y β dependiendo del comportamiento que desee. Si queremos darle más peso a la repulsión aumentaremos β pero en casos extremos podríamos llegar a perder el objetivo. En cambio, si queremos darle más peso a la atracción conseguiremos que evite los obstáculos sin perder al objetivo pero si está mal ajustado podríamos colisionar a causa de una escasa repulsión. En la Figura 6.6 resumimos el algoritmo VFF de manera gráfica para una mejor comprensión.

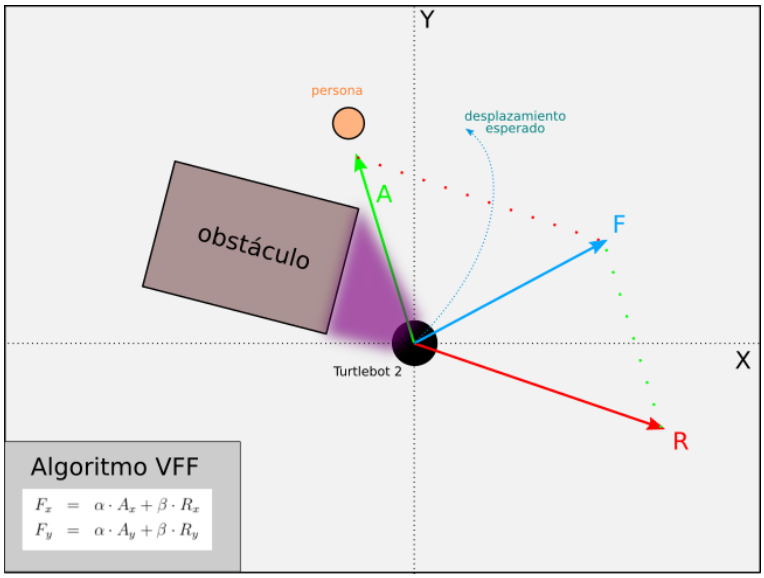

Figura 6.6: Algoritmo VFF

En nuestra solución de referencia, los valores α y β que usamos fueron 0.5 y 0.05 respectivamente. Aplicamos solamente la componente F_x del vector final como velocidad angular, ya que la velocidad lineal está *basada en casos* dependiendo de la distancia al objetivo. Esta última depende del tamaño de la caja de detección:

- Si el área es menor que 16000, el objetivo se encontrará lejos, por tanto comandaremos una velocidad lineal de 0.2 m/s.
- Si el área se encuentra entre 16000 y 45000, significa que el objetivo está cerca, por tanto comandaremos una velocidad lineal de 0.1 m/s.
- Si no se cumple ninguna de estas condiciones, el objetivo se encontrará muy cerca por tanto el robot se detendrá.

6.1.3. Máquina de Estados

Como último paso de la solución, queda implementar una máquina de estados que le dé al robot la capacidad de tomar distintas acciones dependiendo del estado en el

que se encuentre. En nuestro caso, hemos definido 2 estados: Buscar-Persona y Seguir-Persona.

El estado *Buscar-Persona* será aquel con el que empecemos la ejecución del programa. Al principio, girará a una velocidad angular determinada mientras aplica los 3 filtros de detección sobre la imagen para detectar personas. Transitará al siguiente estado cuando el robot identifique un candidato a seguir (mayor área del *BoundingBoxObject* y posicionado en la franja central de la imagen).

En el estado *Seguir-Persona* aplicaremos el algoritmo de *Tracking* y VFF. Además, la velocidad lineal cambiará dependiendo de la distancia al objetivo. Para ello crearemos unas condiciones en las que dependiendo del área de la caja de detección aplicaremos una velocidad u otra. Cuanto más pequeño sea el área, más lejos estará nuestro objetivo. Si por algún motivo perdemos a la persona, incrementaremos un contador de fallo y de esta manera, en caso de no detectarla en el siguiente fotograma, nos quedaremos con el mismo valor del centroide candidato. Cuando el contador supere un límite, transitaremos al primer estado: *Buscar-Persona*.

En la Figura 6.7 se puede ver un esquema de la Máquina de Estados Finitos (FSM) utilizada.

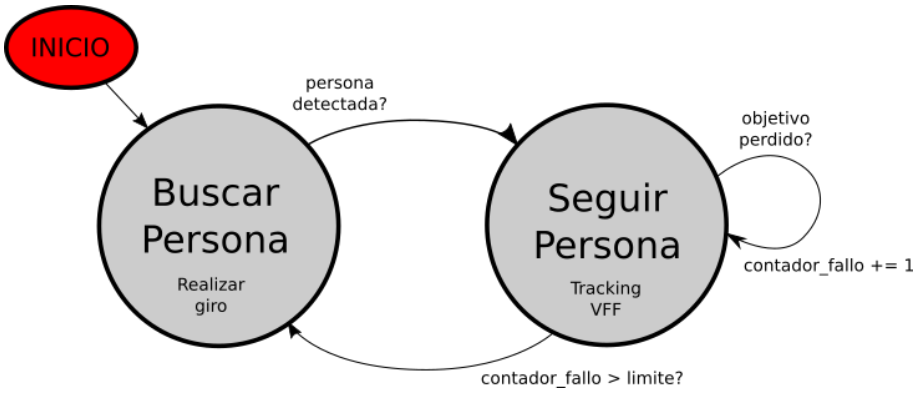

Figura 6.7: Máquina de Estados Sigue-Persona

6.1.4. Validación experimental

A continuación describimos la ejecución de la solución de referencia donde pusimos en prueba nuestro algoritmo:

- Para probar el algoritmo de navegación VFF, colocamos dos objetos de Gazebo en frente del robot con el fin de que pudiera esquivarlos sin perder al objetivo. Primero, el robot se encontraba en el estado Buscar-Persona hasta que detecta a su objetivo en el medio de la imagen. Al transitar de estado, el robot comienza a utilizar la navegación VFF con una velocidad lineal de 0.1 m/s debido a la distancia que le separa con la persona. Al principio, debido a la repulsión del

obstáculo que tiene a su derecha, el robot gira a la izquierda, pero debido al valor incremental de la componente A_x de la fuerza del vector de atracción, el robot bordea el obstáculo. Seguidamente, realiza lo mismo con el obstáculo que tiene a su izquierda. En la Figura 6.8 se puede ver la prueba realizada. Después de esquivar los dos obstáculos, el robot recuperaba su estabilidad.

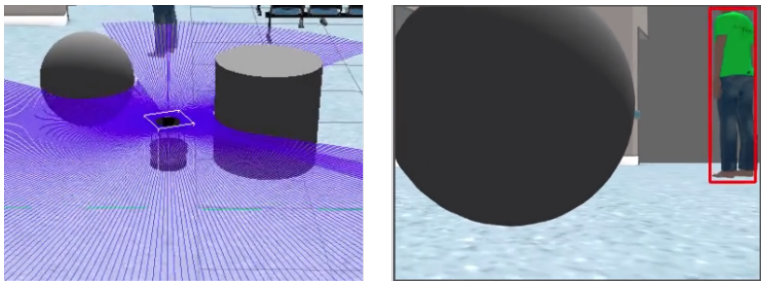

Figura 6.8: Solución Sigue-Persona Simulado: prueba de navegación VFF

- Luego, para probar el algoritmo de seguimiento (*tracking*), dirigimos a la persona simulada a través del recibidor del hospital (la zona del escenario donde hay más personas) pudiendo comprobar que el robot era capaz de seguir a su objetivo sin confundirse. Poco después de comenzar el seguimiento, utilizamos a la persona simulada para alejarla lo máximo posible del robot y vimos cómo su velocidad lineal cambiaba de 0.1 m/s a 0.2 m/s. Luego pasamos cerca de una enfermera cuya caja de detección aumentaba en la parte derecha de la imagen conforme el robot se acercaba, pero no perdía a su objetivo inicial. En la Figura 6.9 se puede ver la prueba que realizamos para el algoritmo de *tracking* perceptivo.

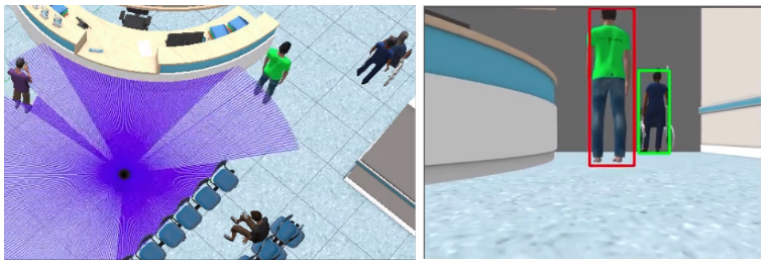

Figura 6.9: Solución Sigue-Persona Simulado: prueba de seguimiento

El vídeo de la solución que muestra todas las pruebas descritas se puede ver en el siguiente enlace: `https://youtu.be/fDAU465eVxQ`

6.2. Solución Sigue-Persona Real

El modo de resolver este problema es idéntico al Sigue-Persona simulado, sin embargo, el entorno sobre el que se mueve el robot, y el rendimiento del ejercicio debido a una distinta carga computacional causada por la ausencia del simulador puede

provocar que el comportamiento de la ejecución en el robot real no sea como la ejecución en el robot simulado.

6.2.1. Adaptaciones

En la solución de referencia seguimos los mismos pasos que en la sección 6.1 y en algunos de ellos ha sido necesario adaptar o reajustar alguna parte:

- Implementación del *Tracker* de seguimiento. Los pasos son los mismos:
  1. Localizar a una persona aplicando los filtros de *score* o puntuación, clase y área.
  2. Aplicamos el criterio de *selección*: dividir la imagen en 3 secciones y elegir como objetivo la persona más cercana que esté situada en la sección central
  3. Crear una clase Tracker que guarde constantemente y vaya actualizando el objetivo de seguimiento en cada fotograma.
  4. Para no perder al objetivo, en cada fotograma nos quedaremos con el centroide de la caja de detección más cercana al centroide candidato del fotograma anterior, mientras no supere un límite de distancia. En caso de no detectar al objetivo, iremos incrementando un contador de fallo.

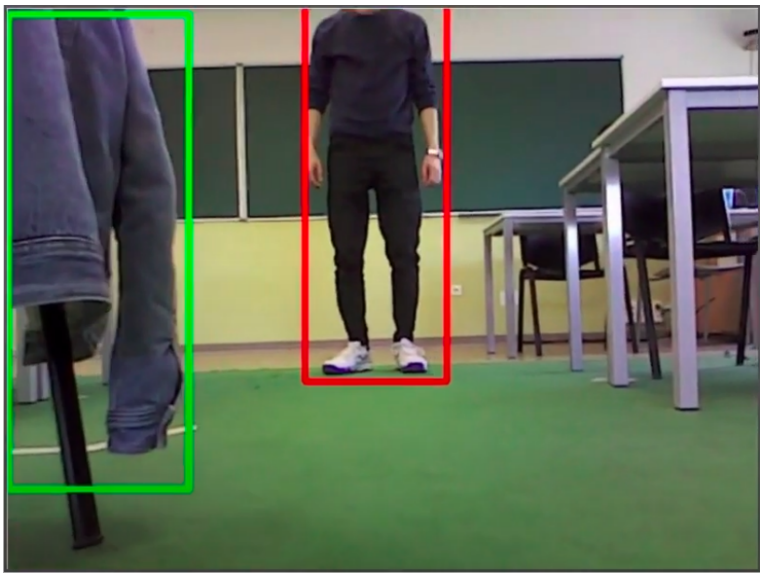

Figura 6.10: Tracker en el ejercicio Sigue-Persona Real

- Implementación del algoritmo VFF. Como la frecuencia de actualización cambia ante la ausencia del simulador, ha sido necesario reajustar los parámetros α y β del algoritmo para equilibrar la relación entre la fuerza de atracción y de repulsión. Es recomendable que el párametro α que multiplica a la fuerza de atracción sea

mayor que β para no perder al objetivo. Sin embargo, tenemos que tener en cuenta que β debe estar correctamente ajustado para esquivar con seguridad y suavidad los obstáculos que se encuentre por su camino. Un buen ajuste de parametros permitirá por ejemplo seguir a una persona a través de un pasillo manteniendo la misma distancia con ambas paredes sin importar la colocación de su objetivo. En nuestro caso, los valores de α y β, comprobados empíricamente, fueron 1.5 y 0.5 respectivamente.

- Máquina de Estados Finitos. Seguimos usando dos estados maestros: Buscar-Persona y Seguir-Persona. La transición del estado Seguir-Persona a Buscar-Persona estaba determinado por el contador de fallo que iniciamos en el proceso de seguimiento del Tracking. El límite del contador dependerá de los fotogramas por segundo. En el ejercicio Sigue-Persona Simulado, la velocidad de fotogramas era baja por lo tanto el límite era menor, pero en este caso, al haber aumentado la velocidad de refresco, tiene sentido aumentar el límite del contador para que, si perdemos a la persona en un fotograma, podamos seguir guardando el centroide de la última caja de detección candidata.

6.2.2. Validación experimental

Para su validación se realizaron las mismas pruebas que en el ejercicio Sigue-Persona Simulado:

- Para probar el algoritmo de navegación VFF, pusimos una silla en mitad del escenario (Figura 6.11 izquierda). Cuando el robot detectaba a su objetivo, empezaba a acercarse, rodeando el obstáculo que tenía a su izquierda. Cada vez que el centroide de la caja de detección se alejaba más del centro de la imagen, mayor era la fuerza de atracción aplicada a la velocidad angular del robot. También, aprovechando el escenario de la *Robocup* en el Laboratorio de Robótica, probamos la navegación a través de un pasillo pudiendo observar que, debido al ajuste correcto de los parámetros de repulsión y atracción, el robot era capaz de seguir a la persona manteniendo la misma distancia con las paredes (Figura 6.11 - derecha). Una vez que el robot tenía hueco para salir del pasillo, conseguía mantenerse en frente de su objetivo.

Figura 6.11: Solución Sigue-Persona Real: prueba de navegación VFF

- Durante una de las ejecuciones, después de que el robot detectara a la persona, y la empezara a seguir por el laboratorio, vimos que la detección mostraba un falso positivo en el robot Pepper debido a su aspecto físico parecido al ser humano, además de la chaqueta colocada sobre la silla (Figura 6.12). Como aplicamos el criterio de selección del centroide más cercano al anterior mediante la distancia Euclidea, el robot logra realizar su tarea de seguimiento correctamente.

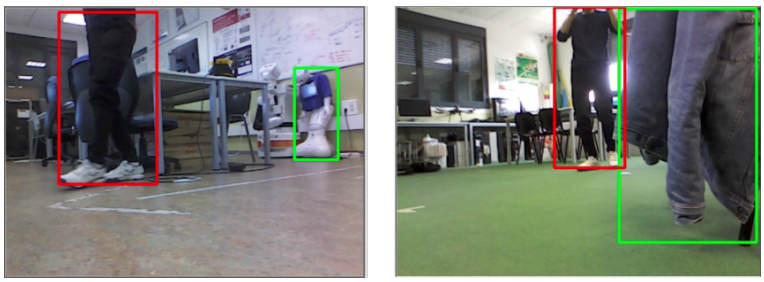

Figura 6.12: Solución Sigue-Persona Real: prueba de seguimiento

El vídeo probando el algoritmo de navegación VFF a través de un pasillo se puede ver a través de este enlace: `https://www.youtube.com/watch?v=qIcsjsmKjH8`

El vídeo de la ejecución donde mostramos las pruebas descritas esta accesible desde el enlace: `https://youtu.be/54Jb4KJwyDM`

6.3. Variantes alternativas

La solución mostrada en las secciones 6.1 y 6.2 no es la única para abordar la aplicación. Podemos enfocar el problema Sigue-Persona de varias maneras. A continuación indicamos algunas posibles opciones que tuvimos en cuenta antes de llegar a la solución final de referencia:

- **Control de velocidad basado en franjas**. En este modelo de seguimiento, dividimos la imagen en franjas y dependiendo de la posición C_x del centroide usamos una velocidad angular determinada. En el código 6.1 podemos ver un ejemplo de implementación.

```
vels = [0.3, 0.2, 0.1, 0, -0.1, -0.2, -0.3] # vel > 0 === left turn

width = img.shape[1]
step = width/len(vels)

while True:
   # ...
   cx, cy = centroid
   # ...

   HAL.setW(vels[int(cx/step)]) # Angular velocity
```

Código 6.1: Ejemplo de control de velocidad basado en franjas

Esta fue la primera solución alternativa utilizada, cuyo video se puede ver a través de este enlace: `https://www.youtube.com/watch?v=58ckb5fFvrs`
La ventaja de este algoritmo es que el movimiento angular está controlado en un rango elegido por el programador y los cambios de velocidad se realizan de manera suave. La desventaja es que puede provocar pequeñas oscilaciones cuando la caja de detección de la persona se encuentre en el centro de la imagen. Además, hay que tener en cuenta la repulsión del láser, cuya suma puede afectar al movimiento del robot al no existir un vector de atracción.

- **Velocidad basada en un controlador PID**. Con un controlador PID bien configurado obtendríamos una respuesta rápida y precisa para no perder a la persona. Su algoritmo de seguimiento sería semejante al ejercicio *Follow Line*[1]. Una desventaja de usar un controlador PID es el hecho de verse afectado a causa de una baja velocidad de fotogramas por segundo provocando oscilaciones constantes. Esta alternativa no tendría en cuenta los obstáculos.

- **Detección de color**. Como solución de escape en caso de tener problemas de alto de rendimiento, pensamos la opción de seguir a la persona dependiendo del color de la camiseta. En el caso del hospital pusimos la camiseta del modelo teleoperado de color verde para facilitar la detección. En un escenario real, la persona podría ponerse alguna camiseta de algún color llamativo para facilitar la detección. Para abordar esa solución, tendríamos que seguir estos pasos:

  1. Cambiar el espacio de color de la imagen de RGB a HSV[2]. El espacio HSV (Hue, Saturation, Value) facilita la detección de colores.
  2. Obtener el rango HSV de valores mínimo y máximo del color de la camiseta y crear una máscara que la filtre correctamente.
  3. Siendo p_i el valor del pixel (0 o 1) de la máscara y x_i las coordenadas del pixel correspondiente, el centroide lo obtendremos de la siguiente manera (ecuacion obtenida de [16]):

$$c = \frac{\sum_{i=1}^{n} (p_i \cdot x_i)}{\sum_{i=1}^{n} (p_i)}$$

[1] **Follow Line**: `http://jderobot.github.io/RoboticsAcademy/exercises/AutonomousCars/follow_line/`

[2] **HSV**: `https://es.wikipedia.org/wiki/Modelo_de_color_HSV`

4. Una vez obtenido el centroide, aplicar el algoritmo VFF igual que en la solución de referencia.

En la Figura 6.13 se puede ver un filtro de color verde que utilizamos al principio para el ejercicio Sigue-Persona Simulado:

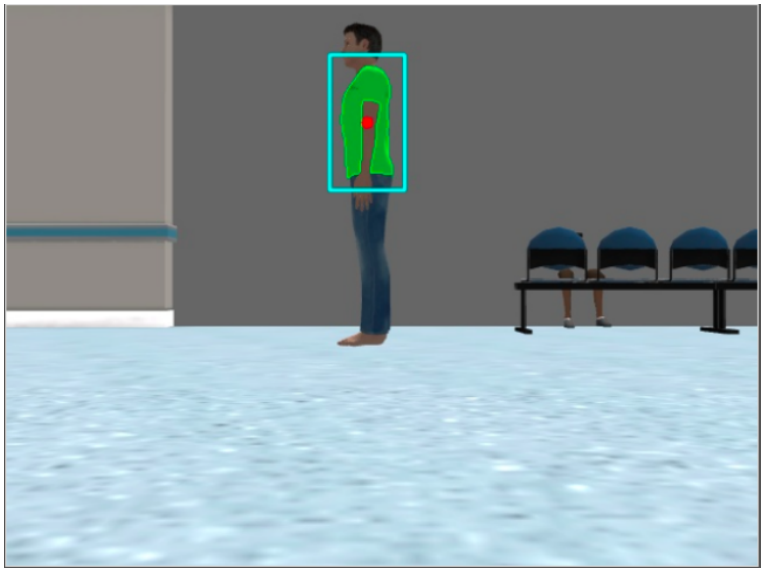

Figura 6.13: Variante alternativa usando un filtro de color

La desventaja de esta solución es que la percepción visual basada en filtros de color es frágil porque requiere que la persona a seguir se vista de una determinada manera. Es preferible una percepción basada en RNA porque es mucho más robusta, funciona en un amplio abanico de situaciones y no requiere que la persona se vista de ninguna manera especial.

Capítulo 7
Conclusiones

Finalizamos este Trabajo Fin de Grado con las conclusiones y un resumen con los objetivos que hemos alcanzado, además de algunas posibles líneas para futuras derivaciones, investigaciones o ampliaciones que superen las metas alcanzadas en este proyecto

7.1. ¿Qué ha aportado este trabajo?

Una vez terminado el proyecto, hemos logrado alcanzar los objetivos que nos propusimos al principio en el capítulo 2:

1. Logramos migrar un modelo TurtleBot2 simulado y adaptarlo de ROS Noetic a ROS2 Foxy. Los ficheros fuente se integraron en el repositorio oficial de Custom Robots [1] (rama foxy).

2. Uno de los objetivos principales conseguidos era entender la infraestructura de Robotics Academy para desarrollar los dos nuevos ejercicios. Actualmente han sido incorporados al conjunto de ejercicios de Robotics Academy: Sigue-Persona Simulado[2] y Sigue-Persona Real[3].

3. Había que diseñar un escenario para el ejercicio Sigue-Persona Simulado, de modo que desarrollamos un *plugin* en ROS2 para controlar una persona simulada en Gazebo usando los botones del teclado a través de la página web. Además incorporamos un hospital, que proporcionaba AWS, en el simulador.

4. Se ha conseguido integrar por primera vez un *robot real* para un ejercicio de Robotics Academy.

5. Como último objetivo que marcamos, pudimos proporcionar una *solución* robusta y funcional del algoritmo Sigue-Persona para los dos nuevos ejercicios.

Como aporte adicional, hemos logrado incorporar un modelo de red neuronal óptimo para arquitecturas y software de bajo rendimiento computacional (ejemplo: Docker, Raspberry Pi) para Robotics Academy.

[1] `https://github.com/JdeRobot/CustomRobots/tree/foxy-devel`
[2] `http://jderobot.github.io/RoboticsAcademy/exercises/MobileRobots/follow_person`
[3] `http://jderobot.github.io/RoboticsAcademy/exercises/MobileRobots/real_follow_person`

7.2. Competencias adquiridas

Durante la realización del TFG se han ido adquiriendo varios conocimientos y competencias sobre distintas tecnologías:

- Conocimiento avanzado de Docker: creación de imágenes y uso de contenedores.
- Profundización en HTML, CSS y JavaScript.
- Ampliación de conocimientos en ROS, ROS2 y ROS Bridge.
- Creación de *plugins* para Gazebo.
- Ampliación de conocimientos en URDF y Xacro.
- Funcionamiento del frontend y backend de Robotics Academy.
- Creación de una solución más robusta en el problema robótico de seguir a una persona utilizando algoritmos como *tracking* perceptivo y navegación con *VFF*.
- Correcta metodología para trabajar en proyectos de software libre en Github.

7.3. Líneas futuras

Una vez lograda la meta principal de este proyecto, han quedado abiertas varias ramas interesantes para mejorar o investigar:

- Sería posible incorporar varios modelos de redes neuronales ligeros parecidos a SSD Inception V2 para ejercicios de *Deep Learning*.
- Crear nuevos ejercicios para Robotics Academy usando el TurtleBot2 real o su modelo simulado. Por ejemplo una navegación global en el entorno del Laboratorio de Robótica de la ETSIT-URJC.
- Encontrar soporte para cámaras RGB-D en ROS2 Foxy que puedan ser utilizadas en *Robotics Academy* para aprovechar la característica de estimar la distancia de los objetos presentes en los fotogramas o manejar nubes de puntos 3D.
- Sería interesante mejorar la solución Sigue-Persona aprovechando la profundidad de una cámara RGB-D. De esta manera, al identificar al objetivo, podríamos lanzar una transformada desde el marco de coordenadas *base_footprint* y seguir constantemente a la persona gracias a su marco de coordenadas dinámico. Con ello, mejoraríamos en precisión y sería más difícil perder a la persona. También se podría usar la profundidad para determinar el módulo del vector de atracción al utilizar VFF.

Bibliografía

[1] Barrientos Sotelo, Víctor Ricardo, Garcia Sánchez, José Rafael y Silva Ortigoza, Ramon. *Robots Móviles. Evolución y Estado del Arte. Polibits*. 2007.

[2] ROS Components. *Turtlebot2*. 2022. URL: https://www.roscomponents.com/es/robots-moviles/9-turtlebot-2.html.

[3] The Construct. *The Construct*. 2022. URL: https://www.theconstructsim.com/.

[4] CPlusPlus. *Cplusplus Documentation*. 2022. URL: https://www.cplusplus.com/.

[5] Docker. *Guía Docker*. 2022. URL: https://docs.docker.com/get-started/overview/.

[6] Open Source Robotics Foundation. *ROS*. 2022. URL: http://wiki.ros.org/es.

[7] Open Source Robotics Foundation. *Simulador Gazebo*. 2022. URL: https://classic.gazebosim.org/.

[8] Python Software Foundation. *Python Documentation*. 2022. URL: https://docs.python.org/3/.

[9] Juan González Gómez. *Construcción de Servicios y Aplicaciones Audiovisuales en Internet*. 2020. URL: https://github.com/myTeachingURJC/2020-2021-CSAAI/wiki.

[10] Guru99. *Clasificación de imágenes de TensorFlow: CNN (Red Neural Convolucional)*. 2022. URL: https://guru99.es/convnet-tensorflow-image-classification/.

[11] SRI International. *Shakey the Robot*. 2022. URL: https://www.sri.com/hoi/shakey-the-robot/.

[12] JdeRobot. *Plataforma Robotics Academy*. 2022. URL: http://jderobot.github.io/RoboticsAcademy/.

[13] JdeRobot. *Unibotics*. 2022. URL: https://unibotics.org/.

[14] Agencia NotiPress. *Conoce a Spot y Atlas, los robots de Boston Dynamics*. 2019. URL: https://www.mypress.mx/negocios/conoce-a-spot-y-atlas-los-robots-de-boston-dynamics-6151.

[15] José Núñez. *Soporte del Intel Realsense para Ubuntu*. 2017. URL: https://costaricamakers.com/soporte-del-intel-realsense-para-ubuntu/.

[16] Programmerclick. *Calcula el centroide de la imagen en OpenCV*. 2022. URL: https://programmerclick.com/article/29231312914/.

[17] Riders. *Riders*. URL: https://riders.ai/.

[18] Jhon Robert. *Hands-On Introduction to Robot Operating System(ROS)*. 2020. URL: https://trojrobert.github.io/hands-on-introdution-to-robot-operating-system(ros)/.

[19] SoftBank Robotics. *SoftBank Robotics*. 2022. URL: https://www.softbankrobotics.com/emea/en.

[20] Revista De Robots. *Qué es la robótica y para qué sirve*. 2021. URL: https://revistaderobots.com/robots-y-robotica/que-es-la-robotica/.

[21] Yujin Robot ROS Components. *Kobuki*. 2022. URL: https://www.roscomponents.com/es/robots-moviles/97-kobuki.html.

[22] Stuart Russell y Peter Norvig. *Artificial Intelligence: A modern approach (3rd Edition)*. 2010.

[23] Shanghai Slamtec. *RPLIDAR A1*. 2022. URL: https://www.slamtec.com/en/Lidar/A1.

[24] Tutor. *Redes Neuronales Profundas: Tipos y Características*. 2019. URL: https://www.codigofuente.org/redes-neuronales-profundas-tipos-caracteristicas/.

[25] Yugesh Verma. *R-CNN vs Fast R-CNN vs Faster R-CNN - A Comparative Guide*. 2021. URL: https://analyticsindiamag.com/r-cnn-vs-fast-r-cnn-vs-faster-r-cnn-a-comparative-guide/.