ESCUELA TÉCNICA SUPERIOR DE INGENIERÍA DE TELECOMUNICACIÓN

GRADO EN INGENIERÍA EN SISTEMAS AUDIOVISUALES Y MULTIMEDIA

TRABAJO FIN DE GRADO

Nuevas Prácticas Docentes de Robótica en el Entorno JdeRobot-Academy

Autor: Carlos Awadallah Estévez

Tutor: José María Cañas Plaza

Curso académico 2017/2018

Agradecimientos

Tenía muchas ganas de escribir esta parte del trabajo, ya que me brinda una oportunidad increíble para agradecer a todos aquellos que han estado a mi lado por el apoyo, el sustento (anímico y también económico) y la orientación que me han regalado sin esperar nada a cambio.

La primera persona en quien pienso es en mi madre, mi pilar de apoyo para seguir adelante y, sobre todo, mi referente en la vida. A ella le dedico este trabajo, espero que estas simples frases sirvan para hacerle ver que le estoy muy agradecido, y que no tengo palabras para expresar lo que siento hacia ella. También al resto de mi familia, mi hermana Shadia, mi tía Rosi y mi abuela MªRosa por sus continuos mensajes de apoyo, sin ellos no habría llegado donde estoy. También a Antonio, que siempre estuvo ahí, que fue la persona que me dio la oportunidad de estudiar lo que quería con su generosidad. A todos ellos, y también a todos mis demás familiares, gracias por todo.

Sigo con mi tutor José María, sin el cual nada de esto habría sido posible. Depositó su confianza en mí, aguantó pacientemente mis continuas preguntas e inquietudes y me orientó siempre que lo necesité para sacar el proyecto adelante. Gracias por tu dedicación, y también por transmitirme el gusto por la robótica durante el trabajo, las prácticas y las clases del grado, campo en el que probablemente enfocaré mi futuro profesional.

Gracias a mi pareja, Bea, que estuvo ahí absolutamente todos los días, que aunque nos separasen 2500 km de distancia siempre estuvo tan cerca como Diciembre de Enero. Por todos los mensajes, por todas las charlas de Skype y por todas y cada una de las palabras intercambiadas que, aunque nunca te lo dije, fueron vitales para seguir aquí día a día con ilusión. Muchas gracias por estos 4 años de apoyo, que se dice pronto.

Como olvidarme de mis amigos de siempre, en especial de Joel C., Joel Y. y Ricardo, y de mi amiga Lidia. Siempre supieron sacar lo mejor de mí, y nada cambió cuando pasamos a vernos sólo unas pocas veces al año. Por todos los momentos compartidos, que me han hecho llegar a ser quien soy.

Por último, pero no por ello menos importantes, mencionar a todas las personas que conocí en Madrid. A todos ustedes: gracias. Esto no habría sido fácil si no hubieran hecho más amena la vida cotidiana. A mis chicos y chicas de la universidad y a mis chicas del gimnasio, un fuerte abrazo.

Resumen

La robótica ha experimentado un crecimiento exponencial en los últimos tiempos, lo que se traduce en una acentuada presencia en la vida cotidiana y profesional, siendo cada vez más y más importante especialistas formados en esta materia. Así, surgen recientemente entornos que acercan a los alumnos de distintas etapas la posibilidad de adquirir competencias en robótica, como JdeRobot-Academy, que pone a disposición de los estudiantes universitarios varios ejercicios clásicos de la robótica de forma sencilla, completa y eficaz.

Este proyecto se ha centrado en la elaboración de dos nuevas prácticas para JdeRobot-Academy. Para cada una de ellas se ha preparado una infraestructura (con robot real o simulado), un nodo académico, la comunicación que resuelve con robot y la interfaz gráfica, permitiendo a potenciales alumnos abstraerse de las complejidades involucradas y así centrarse en el algoritmo oportuno. También se han desarrollado sendas soluciones de referencia.

La primera de las prácticas es *"Follow Face"*, que permitirá al alumno trabajar con hardware real (cámara Sony Evi d100p) y adquirir conocimientos de procesado de imagen, empleando clasificadores en cascada para discernir si una imagen de entrada al sistema contiene caras o no, y en caso afirmativo las persiga usando un control visual reactivo.

La segunda práctica, *"Laser Loc"*, busca resolver un problema de localización basada en láser, en el cual el estudiante deberá valerse del método de filtro de partículas para lograr que un robot estime su posición con errores tolerables en un mundo conocido, del cual se proporciona un mapa binario.

Índice general

Capítulo 1

Introducción

Este Trabajo Fin de Grado (TFG) se encuadra en un entorno educativo para enseñar a programar robots. En este primer capítulo se expondrá tanto el contexto en el cual se sitúa este proyecto como la motivación principal que ha llevado a su desarrollo. Cabe comenzar por realizar una explicación general de qué es la robótica y de las aplicaciones crecientes que está ofreciendo a la sociedad.

Gran parte de la funcionalidad de los robots reside en su software. Intervienen en este aspecto diferentes elementos como los simuladores, las bibliotecas de código y los middlewares de robótica, los cuales se comentarán en la segunda sección del capítulo. En la tercera sección se describirá brevemente el panorama de la educación en robótica y el entorno docente JdeRobot-Academy en el que se ha desarrollado este TFG. La intención es extender precisamente este entorno docente con dos nuevos ejercicios que involucren algún problema de robótica. A la hora de desarrollar un sistema robótico hay numerosas cuestiones que abordar y solucionar, la mayoría de las cuales el entorno docente soluciona y oculta al estudiante, que se puede centrar en los algoritmos robóticos más que en cuestiones auxiliares aunque necesarias.

1.1. Robótica

A lo largo de la historia, el hombre siempre se ha apoyado en la ciencia y la tecnología para facilitarle la vida, para lo cual ha ideado, construido y empleado herramientas o máquinas que consiguen reducir su carga de trabajo. La robótica es una rama de la

ingeniería que emplea la informática para diseñar y desarrollar sistemas automáticos que permitan facilitar la vida del ser humano, e incluso sustituirle en determinadas tareas. Esta rama involucra conceptos de diversas disciplinas, tales como la física, las matemáticas, la electrónica, la mecánica, la inteligencia artificial, la ingeniería de control, etc. Mediante todas estas disciplinas unidas convenientemente se pueden diseñar máquinas que ejecuten distintos comportamientos autónomos en función de su propósito. Estas máquinas se denominan "Robots".

Desde el término de robot acuñado por Isaac Asimov en 1950, estos sistemas autónomos han experimentado un crecimiento exponencial en cuanto a su complejidad, versatilidad, autonomía y, sobre todo, presencia en multitud de ámbitos. Aquellos sistemas operados por seres humanos empiezan a disponer de un sistema de control propio programable que les permite desempeñar aquellas tareas repetitivas o de riesgo para las personas, englobando actividades básicas y de difícil realización, hasta el punto en el que nos encontramos en la actualidad, en el cual existen múltiples ejemplos que integran la robótica en diferentes campos y tareas. Los robots comerciales e industriales son ampliamente utilizados y realizan tareas de forma más exacta o más barata que los humanos. Los robots también se emplean en trabajos demasiado sucios, peligrosos o tediosos para los humanos. En definitiva, un campo cada vez más popular y en expansión constante.

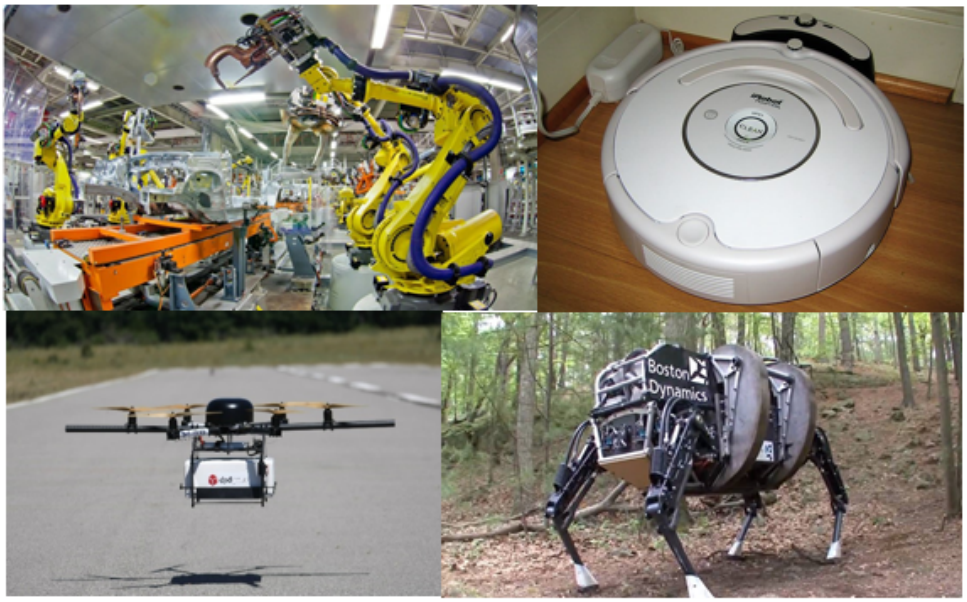

Figura 1.1: Robots modernos

Hoy en día no solamente nos rodean los robots industriales, como los involucrados en cadenas de envasado de alimentos o cadenas de producción, sino que los robots cobran cada vez más importancia en entornos alternativos, como el doméstico entre otros, algunos de los cuales quedan representados en la Figura 1.1. Las aspiradoras robóticas (Roomba, Dyson, Xiaomi,...) han llegado a los hogares con éxito para realizar una tarea doméstica necesaria. Por otro lado, los vehículos de transporte aumentan cada vez más la tecnología robótica incorporada, como son los módulos de aparcamiento automático, u otros más avanzados como los asistentes de conducción autónoma (autopiloto de Tesla), o los recientes prototipos de coches autónomos que han lanzado grandes empresas como Google o Apple. Otros ámbitos, como el militar con máquinas capaces de desactivar bombas o realizar misiones de rescate, el de la medicina con el ejemplo del robot DaVinci que permite operar a un paciente desde cualquier parte del mundo con mayor precisión que la humana, o el de la logística en almacenes de Amazon tampoco están exentos de presencia robótica. Incluso se está trabajando para construir robots androides o humanoides capaces de desarrollar "comportamientos inteligentes" como el robot Asimo de Honda (Figura 1.2) o TOPIO, creado por TOSY Robotics (Figura 1.3), que puedan desempeñar tareas de interacción con herramientas y entornos humanos, ya sea con fines experimentales como el estudio de la locomoción bípeda, o incluso como auxiliar para la tercera edad o personas con movilidad reducida.

Figura 1.2: Robot Ásimo

Figura 1.3: Robot TOPIO

El abanico de aplicaciones que pueden involucrar sistemas robóticos es tan extenso como la imaginación de la comunidad dedicada a crearlos, por lo que tareas que

nunca habíamos imaginado automatizadas son ahora labores totalmente robotizadas, como puede ser la agricultura de precisión a través de drones con análisis de imágenes térmicas y multiespectrales para aumentar el rendimiento de las explotaciones agrícolas, la automatización de aplicaciones anestésicas de bajo nivel e incluso competiciones deportivas de robots.

Es difícil no parase a pensar en la potencia que tiene esta rama de la ciencia y la tecnología, y no sólo eso, sino también en la que puede alcanzar, ya que esto es sólamente el principio. En todo lo anterior, sale a relucir la utilidad de esta área de desarrollo, que permitirá en el futuro ganar en comodidad, economía e incluso salud. Es importante advertir la importancia de dominar esta disciplina en el futuro cercano, ya que será la llave que abrirá la puerta hacia un mundo más sencillo y seguro a través de la automatización.

1.2. Software Robótico

Para la totalidad de los ejemplos mencionados es necesario que el comportamiento del software que controla dichos robots debe ser robusto. Es por eso que se divide en distintas capas (*drivers*, *middleware* y aplicaciones), cuya arquitectura es distinta según sea su aplicación final.

Lejos ya de ser controlados por personas, la mayoría de los robots están dotados de una autonomía que les permite desempeñar la tarea o tareas para las cuales han sido diseñados sin la mediación de terceros. Que dispongan de esta característica es posible gracias al minucioso desarrollo de sistemas complejos que constan de un software tal que compone algo parecido a una inteligencia autónoma. El desarrollo de software robótico parte de ciertos requisitos o tareas, entre las que se incluyen circuitos de retroalimentación, filtrado de datos, control, búsqueda de caminos y localización entre muchas otras.

En los últimos años han surgido numerosas plataformas de desarrollo de software para aplicaciones robóticas, también llamados *middleware* robóticos. Los simuladores también son ingredientes habituales en el desarrollo de software robótico, permiten realizar las pruebas pertinentes, depurar los fallos programar una versión funcional del robot evitando el alto coste del proceso real una y otra vez. Estas herramientas son indispensables hoy en día para generar el conjunto de comandos codificados que indican al robot y a su sistema electrónico las acciones a realizar, así que entraremos un poco más en detalle

en el siguiente apartado. Finalmente, las bibliotecas son de gran utilidad para generar el conjunto de comandos codificados que conforman el comportamiento del robot.

1.2.1. Middlewares robóticos

Los *middleware* robóticos pueden definirse como entornos o *frameworks* para el desarrollo de software para robots, es decir, software que conecta aplicaciones o componentes software para soportar aplicaciones complejas y distribuidas. Estos entornos suelen incluir *drivers* para controlar los sensores y los actuadores de los robots, arquitectura software para las aplicaciones que se van a crear, bloques de funcionalidad robótica ya resuelta y otra serie de herramientas como simuladores, visualizadores... Es por eso que hay muchas reseñas del *middleware* en las que se hace referencia a él bajo el nombre de "pegamento para software". Una de las tareas de las que se encarga el middleware es de conectar nuestro hardware, real o simulado, con nuestra aplicación en desarrollo. El *middleware* robótico más extendido en el mundo es ROS...

Robot Operating System (ROS)[1]

Es una plataforma de software libre para el desarrollo de software de robots que provee la funcionalidad de un sistema operativo en un clúster heterogéneo, como el control de dispositivos de bajo nivel, la abstracción del hardware, el control de dispositivos de bajo nivel y mecanismos de intercambio de mensajes entre procesos, todos ellos de vital importancia para el desarrollo en este campo. Una de las razones que hace que ROS sea el *framework* más utilizado es que, aunque se diseñó para ser utilizado principalmente en sistemas UNIX, se ha adaptado de forma que pueda utilizarse también en los otros sistemas operativos principales como Fedora, Debian, Windows, MacOS X, Arch, Slackware, Gentoo u OpenSUSE, llegando incluso a abrir la puerta a la construcción de aplicaciones multiplataforma.

Existen algunos otros interesantes como:

Orocos[2]

También encaja en el grupo de las plataformas de software libre para control avanzado de máquinas y robots basado en C++.

[1] `http://www.ros.org/`
[2] `http://www.orocos.org/`

Orca[3]

Un proyecto de software libre para aplicaciones de índole robótico que permite crear aplicaciones algo más complejas al estar orientado a componentes.

Urbi[4]

Middleware multiplataforma de código abierto en C++ que, utilizada de forma conjunta con ROS, permite desarrollar aplicaciones para sistemas complejos y completos.

JdeRobot[5]

Plataforma para desarrollar aplicaciones robóticas y de visión artificial, que involucra nodos programados con varios lenguajes de programación y que es compatible con *middlewares* de comunicaciones como ICE o ROS.

1.2.2. Simuladores robóticos

Generalmente, el diseño y puesta en escena de un robot es costoso y caro, por lo que no es aconsejable correr el riesgo de que el código implementado contenga errores que salgan a la luz en el momento de probarlo, ya que esto puede generar malfuncionamientos e incluso rotura del hardware. Es por eso que el paso previo a llevar el código a la máquina consiste en probarlo antes en un simulador orientado a este tipo de aplicaciones, lo que permite un desarrollo seguro y económico. Algunos de los simuladores más usados son:

Gazebo[6]

Es el simulador 3D de código abierto más distribuido, bajo licencia Apache 2.0. Destaca sobre los demás por sus motores de físicas, motor de renderizado avanzado y soporte para *plugins* de robots y sensores, además de un amplio repertorio de robots comerciales y sus correspondientes sensores y actuadores. Además, se integra bien con ROS, permitiendo usar exactamente el mismo código de la simulación en el robot real.

Stage[7]

Esta herramienta permite simular conjuntos numerosos de robots móviles en el plano

[3] `http://orca-robotics.sourceforge.net//`
[4] `https://github.com/urbiforge/urbi`
[5] `https://jderobot.org/`
[6] `http://gazebosim.org/`
[7] `http://wiki.ros.org/stage`

bidimensional, integrable con ROS.

Webots[8]

Simulador de robótica avanzada que permite definir modelos propios y su física, escribir controladores para ellos y hacer simulaciones a gran velocidad. Soporta el humanoide Nao.

1.2.3. Bibliotecas

En el proceso de desarrollo de la parte software en robótica hay que resolver multitud de problemas clásicos y funcionalidades básicas además de las específicas para la aplicación en cuestión, ardua tarea para el programador que debe no sólo integrarlas en su proyecto, sino también abordarlas desde cero. Con el propósito de evitar esta situación, existen las bibliotecas, que no son más que conjuntos de implementaciones de código que ofrecen una interfaz bien definida para la funcionalidad que invocan. Su fin es ser utilizada por otros programas, independientes y de forma simultánea. Las bibliotecas pueden vincularse a un programa (o a otra biblioteca) en distintos puntos del desarrollo o la ejecución de dicho programa. Algunas de las bibliotecas utilizadas en robótica son:

- OpenCV [9]: Es una biblioteca libre de visión artificial originalmente desarrollada por Intel. Contiene más de quinientas funciones que abarcan una gran gama de áreas en el proceso de visión, como reconocimiento de objetos, reconocimiento facial, calibración de cámaras, visión estéreo y visión robótica. Originalmente, OpenCV fue escrita en C++. Actualmente, la librería dispone de interfaces en C++, C, Python, Java y MATLAB. Es multiplataforma, existiendo versiones para GNU/Linux, Mac OSX, Windows y Android.

- PCL [10]: Se utiliza para el procesamiento digital de imágenes RGBD mediante el tratamiento de nubes de puntos 3D. Contiene numerosos algoritmos de última generación que incluyen filtrado, estimación de características, reconstrucción de superficies, ajuste de modelos y segmentación entre otros. Para simplificar el desarrollo, PCL se divide en una serie de bibliotecas de código más pequeñas, que se pueden compilar por separado. Es multiplataforma y ha sido compilada con éxito

[8] `https://www.cyberbotics.com/`
[9] `https://opencv.org/`
[10] `http://pointclouds.org/`

en Linux, Mac OSX, Windows y Android / iOS.

1.3. Docencia en robótica

La robótica con fines educativos está adquiriendo protagonismo estos últimos años en la enseñanza preuniversitaria, ya que proporciona un medio de aprendizaje para estudiantes de cualquier nivel, cuyo único requisito de partida es la motivación por el diseño de aplicaciones propias y su construcción. Con ello se consigue instruir a las personas en el campo de la robótica, pero no sólo eso, dado que para ello es necesario abordar temas multidisciplinarios (electrónica, informática, mecánica, física, etc.). El estudiante también adquiere conceptos de estas áreas científicas y tecnológicas. Además, proporciona una nueva visión del universo que le rodea, aprendiendo a distinguir los problemas a su alrededor y a ser capaz de tomar una decisión al respecto. La docencia en robótica en las etapas de enseñanza primaria y secundaria intenta despertar el interés de los estudiantes transformando las asignaturas tradicionales en más atractivas e integradoras, a la par que más prácticas, ya que crea entornos de aprendizaje propicios que recrean estos problemas que les rodean, y a través de los cuales pueden dar rienda suelta a su creatividad y plasmar en la realidad los conceptos teóricos que adquieren. En los centros de enseñanza primaria y secundaria se imparte la robótica con frecuencia, mediante plataformas físicas como los robots LEGO (Mindstorms, RCX, NXT, Evo, WeDo), placas Arduino, los kits de SolidWorks, etc.

1.3.1. Docencia en la universidad

En la docencia universitaria se imparten clases de robótica en ciertos Grados y Postgrados dentro de las escuelas de ingeniería. En España, podemos ver la robótica integrada en el “Grado en Ingeniería Robótica” de la Universidad de Alicante, y los Grados de “Electrónica industrial y automática” o en “Ingeniería Electrónica, Robótica y Mecatrónica” en diversas universidades, además de algunos otros que están por venir como el Grado en Ingeniería Robótica Software que habilitará la Universidad Rey Juan Carlos este próximo mes de Septiembre para el nuevo curso académico. Sin embargo, la enseñanza en robótica se concentra mayoritariamente en los programas de Postgrado, puesto que es una enseñanza más especializada. Existen Másteres destacados como el

"Máster de Visión Artificial", el "Máster Universitario en Ingeniería Mecatrónica", o el "Máster Universitario en Automática y Robótica". Ya en el ámbito internacional se pueden destacar universidades especializadas en robótica como el MIT, Carnegie Mellon University, Stanford o Georgia Institute of Technology. Además, algunas asociaciones prestigiosas como ACM (Association for Computing and Machinery) y la IEEE-CS (IEEE Computer Society) ven la robótica como un área de conocimiento imprescindible en estudios de ingeniería, informática y sistemas inteligentes. La Universidad Rey Juan Carlos cuenta con la plataforma de robótica JdeRobot, que posee un entorno académico conocido como JdeRobot-Academy, el cual es el contexto cercano de este trabajo. Este entorno educativo se ha empleado con éxito en diferentes asignaturas, como "Visión en Robótica" del Máster de Visión Artificial, o "Robótica" del Grado de Ingeniería Telemática. Asimismo, ha tenido éxito en cursos de introducción a la robótica y a la programación de drones.

1.3.2. Entorno docente JdeRobot-Academy

Los *middleware* robóticos empleado para el desarrollo de este TFG son ROS y JdeRobot, que incluye el entorno académico JdeRobot-Academy[11]. Con este trabajo se ha pretendido extender sus posibilidades de aprendizaje, ampliándolo con dos nuevos ejercicios. Los ejes en los que se apoya JdeRobot-Academy son:

a) Lenguaje Python (por su sencillez y potencia),

b) simulador Gazebo (con distintos modelos de robot, tales como drones, formula1, brazos, aspiradoras, etc.), y

c) foco en el algoritmo en vez de en el middleware, ocultando al estudiante los detalles de la infraestructura.

El entorno cuenta con un conjunto de prácticas, cada una de ellas enfocada a resolver un problema clásico de la robótica. Para cada práctica se ofrece un componente académico que resuelve tareas auxiliares como la interfaz gráfica, la conexión con sensores y actuadores concretos y la temporización entre otras, y aloja el código del estudiante,

[11] `https://jderobot.org/JdeRobot-Academy`

que así se concentra en los algoritmos de percepción y control. Cada nodo está formado por una parte específica preparada, que queda oculta, y el código del estudiante, que simplemente rellena un sencillo fichero plantilla con la lógica del robot.

Se distinguen rápidamente varias capas de distinto nivel en su composición. Aunque la capa de nivel más bajo se da resuelta al alumno, queda de manifiesto que es necesario implementarla para dar soporte a cada práctica, incluyendo la creación de los modelos del robot en el simulador y el escenario necesarios, los plugins que emplearán estos modelos como *drivers*, y la comunicación entre el simulador y el componente académico de alto nivel. Todo ello se puede ver en la Figura 1.4:

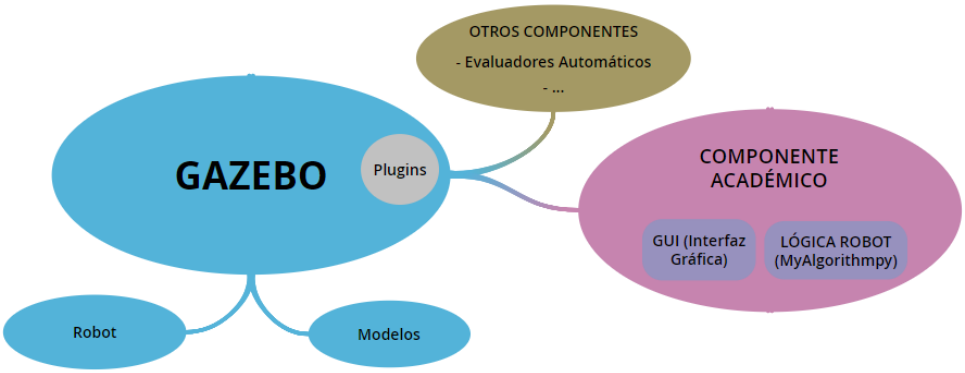

Figura 1.4: Estructura de una práctica en JdeRobot-Academy

En la Figura 1.4 se observa que los componentes académicos siguen una arquitectura software que facilita el desarrollo de las prácticas a los alumnos, los cuales únicamente deberán programar su algoritmo, ya sea el pilotaje en función de los datos que proporcionan los sensores o la planificación. Los componentes académicos cargan el código escrito por el alumno en el fichero plantilla llamado *MyAlgorithm.py* (donde se materializa su resolución), y muestran en la interfaz gráfica las pruebas o soluciones que realicen los alumnos, distintas trazas de depuración como pueden ser imágenes procesadas, datos láser, imágenes de cámaras integradas,...

Aunque el entorno suele emplear el simulador Gazebo, ha sido desarrollado para que el mismo código implementado en el fichero solución pueda ser ejecutado sobre un robot real. Es por eso que, con el driver adecuado, ese código puede operar un robot físico. El sistema operativo de base es Linux, para la cual se ha preparado la

infraestructura. Adicionalmente, se ha puesto la vista en crear prácticas abordables desde otras plataformas, utilizado herramientas como la interfaz web de Gazebo, los cuadernillos de Jupyter (ver 3.6), tecnologías web de servidor y mediante el empleo de Dockers en vistas a disponer de una versión de la plataforma en MacOS y Windows. Uno de los ejemplos de prácticas de los que dispone la plataforma es el ejercicio Color Filter que se muestra en la Figura 1.5:

Color Filter

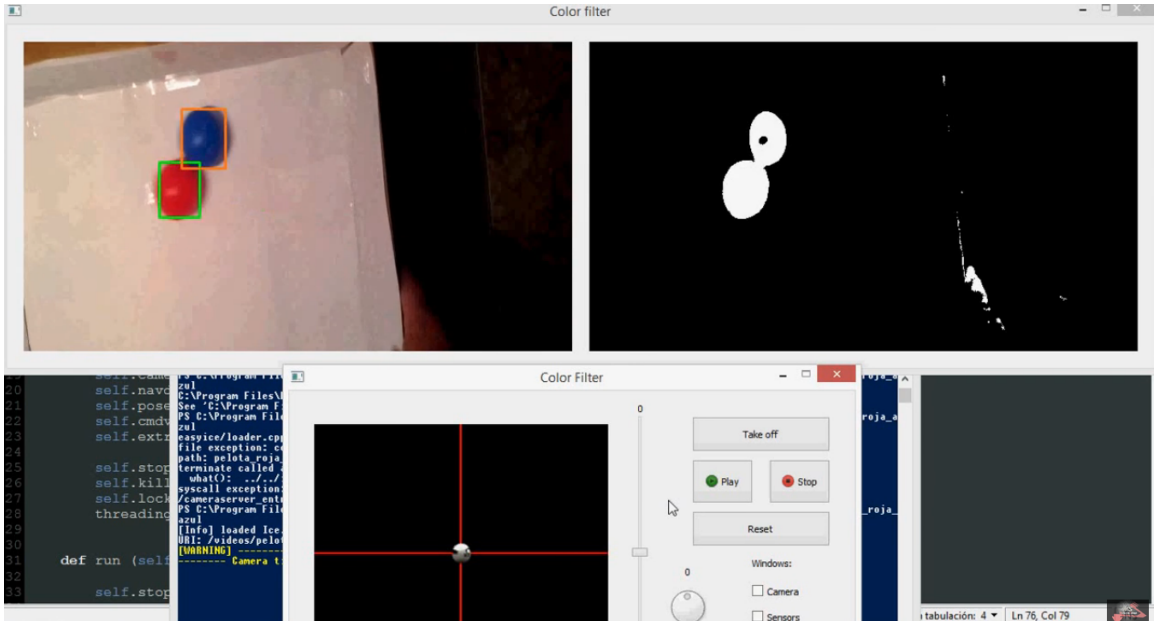

Figura 1.5: Color Filter

En este ejercicio el alumno puede experimentar con un filtro de color [12] aplicado sobre la secuencia de vídeo que proporciona un servidor de imágenes. Su código debe detectar el objeto relevante coloreado (pelotas rojas y azules) utilizando sendos filtros de color, y extraer su posición XY en las imágenes.

[12] `https://www.youtube.com/watch?v=tiXagRiqQnY`

Bump&Go

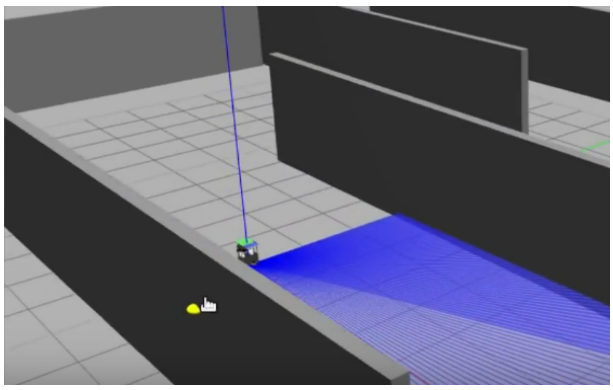

Figura 1.6: Bump&Go

El ejercicio *Bump&Go* (Figura 1.6) permite implementar una máquina de estados para controlar al robot kobuki en una simulación, un robot diseñado para ser duradero, resistente y rápido, que proporciona al programador acceso a sus motores y a sus sensores láser y de colisión. El alumno debe conseguir que el robot realice el comportamiento típico de chocar y girar[13].

Labyrinth Escape

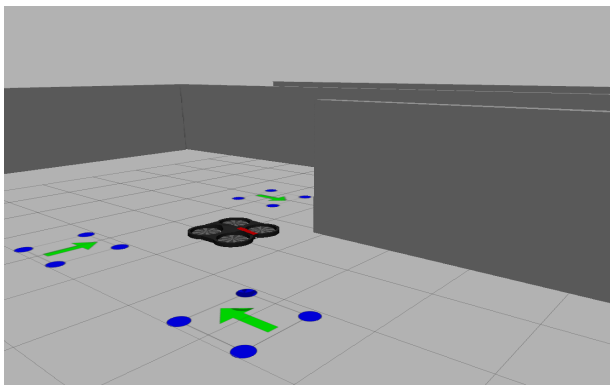

Figura 1.7: Labyrinth Escape

En el ejercicio *Labyrinth Escape* (Figura 1.7), el alumno debe programar la lógica de control de un dron con una cámara incorporada e interfaces de acceso a sus motores para lograr que salga con éxito de un laberinto a través de las imágenes que capta. Para ello,

[13] `http://jderobot.org/store/fperez/uploads/videos/bumpgoo.ogv`

debe hacer uso de las herramientas de procesado y segmentación de imagen para reconocer e interpretar una serie de flechas verdes colocadas a lo largo del escenario que le indican el trazado que debe seguir para encontrar la salida[14].

Obstacle Avoidance

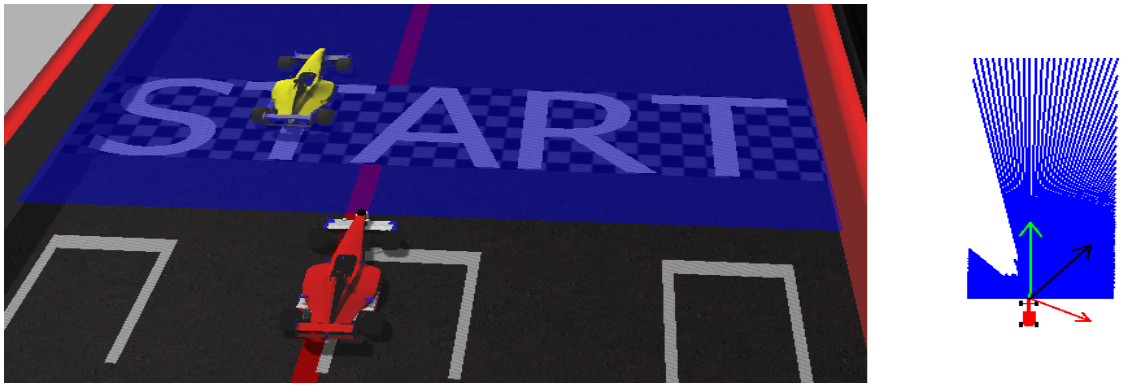

Figura 1.8: Obstacle Avoidance

En la práctica *Obstacle Avoidance* el estudiante entra en contacto con la tecnología de coches autónomos (en este caso de tipo F1 como se puede ver en la Figura 1.8) equipado con un sensor láser y dos cámaras que ofrecen imágenes desde sus lados izquierdo y derecho. El alumno tendrá que hacer un uso inteligente de la información que el robot pone a su disposición en la simulación para construir una lógica de control y navegación local, la cual se encargará de seguir la carretera del circuito, tomar correctamente las curvas y esquivar los posibles obstáculos que vayan surgiendo en el camino[15].

[14] `https://youtu.be/EdWE5P3uZak`
[15] `https://youtu.be/gVvnrRDmLkI`

Drone-Cat-Mouse

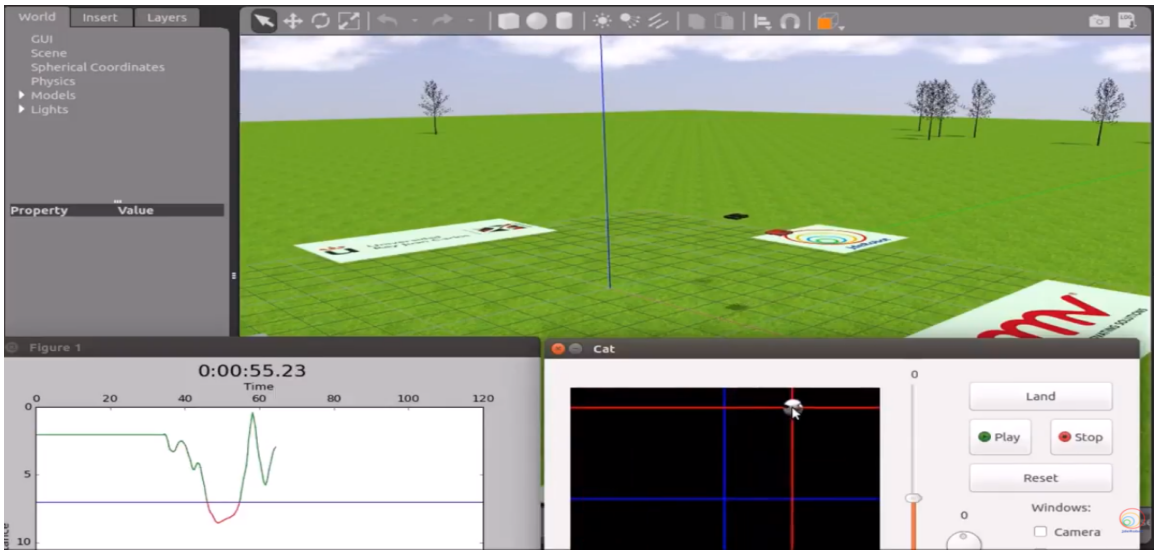

Figura 1.9: Drone-Cat-Mouse

Drone-Cat-Mouse es otra de las prácticas del entorno, en la que se ha de programar el dron negro para que persiga al dron rojo (pre-programado con diferentes niveles de dificultad) permaneciendo lo más cerca posible sin colisionar en un entorno simulado al aire libre, emulando el juego del gato y el ratón[16]. Esta práctica se ha empleado en las dos ediciones previas del campeonato *Program-A-Robot*, la primera en la URJC y la segunda en las Jornadas Nacionales de Robótica. Se va a emplear también en la tercera edición dentro de la conferencia internacional IROS[17].

1.3.3. Ejercicios recientes

El contexto inmediato de este trabajo de fin de grado reside en una serie de ejercicios elaborados recientemente también para ampliar las posibilidades de aprendizaje de JdeRobot-Academy.

Entre ellos, cabe destacar el Trabajo de Fin de Grado de Irene Lope Rodríguez *"Nuevas Prácticas en el Entorno Docente de Robótica JdeRobot-Academy"*[1], que se centró en enriquecer la plataforma JdeRobot-Academy con dos nuevas prácticas llamadas "Coche autónomo negociando un cruce" y "Aspiradora autónoma con autolocalización".

[16] `https://www.youtube.com/watch?v=DYD9oPawhWg`
[17] `https://www.iros2018.org/competitions`

La primera de ellas perseguía que un coche autónomo fuese capaz de reconocer una señal de STOP en un cruce vial a través de procesamiento de imágenes provenientes de tres cámaras a bordo del robot, desplegando el comportamiento habitual ante esta señal de detener el avance, examinar las vías perpendiculares para reconocer posibles coches circulando por ellas y, en caso de que las vías estén libres, girar a la derecha o izquierda para ocupar el carril correspondiente del sentido que se quiera seguir. La segunda de las prácticas que componían este trabajo tenía como objetivo reproducir el comportamiento de limpieza de un robot aspiradora de mercado, eliminado la suciedad en la mayor extensión posible del entorno en que se encuentra, el cual conoce a través de un mapa y un sensor de posición que le ubica en todo momento en la habitación simulada.

También hay que mencionar el Trabajo de Fin de Grado de Vanessa Fernández Martínez *"Nuevas Prácticas en el Entorno Docente de Robótica JdeRobot-Academy"*[2], en el que se añadió dos nuevas prácticas al repertorio de JdeRobot-Academy bajo los nombres de "Aspiradora autónoma" y "Aparcamiento automático", además de mejorar la práctica ya existente *"TeleTaxi"* con modelos mejorados, un mejor desempeño del algoritmo GPP de navegación global e incluso la inclusión de un evaluador automático capaz de medir el desempeño del algoritmo en función de distintos parámetros para proporcionar al alumno una nota final. En cuanto a los ejercicios creados, *vacuum cleaner* se centró en un algoritmo de navegación sin localización basado en la serie 500 de aspiradoras *Roomba* de iRobot, a través del cual debe recoger la suciedad de la mayor superficie posible de una casa. Por su parte, *autopark* tiene como objetivo implementar un algoritmo basado en estados que permita a un coche autónomo realizar una maniobra de aparcamiento, reconociendo los huecos libres, evitando chocar y respetando los espacios de seguridad a través de los datos sensoriales del robot.

Continuando en esta línea de trabajo, el presente TFG se apoya en la filosofía de estos dos trabajos precedentes para aportar a JdeRobot-Academy dos prácticas que involucren elementos hasta ahora no presentes en las demás: una empleando hardware real y otra haciendo uso de *drivers* de ROS directos.

El objetivo general de este TFG es ampliar las posibilidades de esta plataforma docente, creando nuevas prácticas y mejorando las ya existentes, aumentando su versatilidad. En los próximos capítulos abordaremos todos los elementos necesarios para conseguir este objetivo, empezando por el Capítulo 2, en el que concretaremos los objetivos que se han marcado, así como los requisitos de partida y la metodología empleada. En el Capítulo 3 se expondrá la infraestructura utilizada para llevar a cabo el proyecto. En los Capítulos 4 y 5 se abordarán las prácticas que se han creado. Por último, en el Capítulo 6 se expondrán las conclusiones obtenidas, así como las posibles líneas de mejora que se pueden seguir.

Capítulo 2

Objetivos

En este punto ya hemos introducido el contexto en el que desarrolla el trabajo, es momento de describir los objetivos concretos que se han tratado de alcanzar, los requisitos para las soluciones desarrolladas y la metodología seguida para su consecución.

2.1. Objetivos

La meta de este proyecto consiste en la creación de dos nuevas prácticas de sistemas robóticos para el entorno docente JdeRobot-Academy, una primera que consistirá en seguimiento de caras a través de una cámara mócil y otra sobre auto localización de un robot basada en láser. Para cada una de ellas se creará la infraestructura necesaria y una solución de referencia.

En primer lugar, la práctica de seguimiento de caras utilicará una cámara móvil montada en un cuello mecánico, la cual puede apuntarse hacia varias direcciones, para que el alumno ponga en práctica sus conocimientos de segmentación de imagen buscando posibles caras de personas en las imágenes, y generando órdenes para los motores de la cámara de modo que esta siga con la mirada el movimiento de la persona que tiene delante.

La segunda práctica es sobre un robot móvil y busca resolver el problema de auto localización basada en láser, de manera que dicho robot sepa en todo momento su posición y orientación en el entorno que le rodea, valiéndose únicamente de un sensor láser incorporado.

2.2. Requisitos

El software desarrollado, además de proporcionar las dos prácticas deseadas, deberá cumplir estos requisitos de partida del proyecto:

1. El sistema operativo que se empleará para este proyecto será Ubuntu 16.04 LTS.
2. Se hará uso del middleware robótico JdeRobot en su versión 5.6.1. El uso de esta plataforma simplifica el desarrollo del comportamiento del robot.
3. Se hará uso de OpenCV3 en la práctica de seguimiento de caras.
4. En el ejercicio de localización basada en láser se usará *ROS-Kinetic* y todas las simulaciones se realizarán en el simulador Gazebo, en concreto en la versión 7.9.0.
5. El lenguaje de desarrollo empleado para crear los distintos componentes será Python, en concreto en su versión 2.7.12. Por compatibilidad con JdeRobot-5.6.1 y de éste con el middleware ROS *Kinetic* no se ha usado Python-3.x
6. Las soluciones han de ejecutar algoritmos que funcionen en tiempo real, de manera que deberán ser eficientes y ejecutar movimientos suaves.

2.3. Metodología

La elaboración de este trabajo podría descomponerse en un conjunto de iteraciones con varias fases, en cada una de las cuales se establecía una reunión con el tutor para determinar los subobjetivos siguientes, planificar cómo abordarlos, analizar los posibles problemas y corregir fallos. Así, se ha conseguido un desarrollo fluido y completo, asentando los conocimientos y despejando las dudas que surgían a lo largo de los meses que se dedicó al trabajo.

Parte del seguimiento lo hemos hecho utilizando herramientas de apoyo, como la bitácora semanal en la Wiki de JdeRobot [1] donde periódicamente se redactaban los avances realizados y se añadían vídeos demostrativos de lo conseguido, imágenes representativas y texto descriptivo. El código asociado a los avances se almacenó en un

[1] `https://jderobot.org/Cawadallah-tfg`

repositorio personal de GitHub[23], al que el tutor ha podido acceder en todo momento para aportar realimentación y favorecer el progreso.

Se ha optado por seguir el modelo de desarrollo en espiral creado por Barry Boehm, el cual hemos creído que se adaptaba perfectamente a nuestras necesidades, permitiéndonos separar el comportamiento final en varias sub-tareas más sencillas, a la par que disponer de flexibilidad ante cambios en los requisitos, algo bastante común a medida que avanzaba el desarrollo. Con él, conseguimos una resolución temprana de riesgos y poder definir la arquitectura en las fases iniciales, todo ello basado en un proceso continuo de verificación de la calidad.

Figura 2.1: Modelo de desarrollo en espiral

[2] https://github.com/cawadall/Academy
[3] https://github.com/cawadall/JdeRobot

Como se puede ver en la Figura 2.1, este modelo de ciclo de vida permite ir obteniendo prototipos funcionales en primera instancia, luego mejorar el prototipo conseguido y, por último, pulir los detalles para cubrir todos los requisitos. Así, se realiza el trabajo de forma incremental, en forma de ciclos con cuatro fases bien definidas:

- **Determinar objetivos**: en esta primera fase del ciclo se definen las metas que se deben conseguir para que una vez completadas se pueda dar por finalizado el mismo.
- **Análisis del riesgo**: en la segunda fase se evalúa qué problemas es posible encontrarse al empezar el desarrollo, y se piensa en cómo abordarlos.
- **Desarrollar y probar**: en la tercera fase es en la que, tras evaluar los riesgos, se procede al propio desarrollo del trabajo, además de las correspondientes pruebas para verificar su correcta funcionalidad.
- **Planificación**: en esta última fase se valoran los resultados obtenidos en el ciclo y se planifican las siguientes etapas del proyecto.

2.4. Plan de trabajo

Para lograr los objetivos descritos, se han seguido las siguientes etapas de trabajo:

- **Familiarización con el entorno JdeRobot**. Tras la descarga e instalación del software involucrado, incluyendo dependencias, simulador y bibliotecas necesarias, se tratará de entrar en contacto con el entorno JdeRobot modificando y pulimentando algunas prácticas existentes para añadir mejoras, por ejemplo, en sus interfaces gráficas.
- **Toma de contacto con el simulador Gazebo**. En esta etapa se han estudiado distintos ejemplos disponibles en la web de Gazebo[4] y de JdeRobot. Además, se ha estudiado el funcionamiento básico de los *plugins* que Gazebo emplea para controlar los robots, sus sensores y sus actuadores. Esto ha implicado la investigación del lenguaje de programación C++, el cual también ayudará a comprender los *plugins* ROS.

[4] `http://gazebosim.org/tutorials`

- **Estudiar las bibliotecas involucradas**, entre las cuales destacan OpenCV, Threading, NumPy y PyQt5.
- **Desarrollo de la infraestructura para la práctica de seguimiento de caras**. Manejo de los *drivers* involucrados para el control de los motores y de la cámara. Se creará el nodo académico que albergará el código del estudiante. También se creará una versión de la práctica a través de Jupyter.
- **Programación de una solución de referencia para el seguimiento de caras**.
- **Preparación de la infraestructura para el ejercicio de autolocalización basada en láser**. Se desarrollará el *driver* para el robot simulado. Se creará el nodo académico que alberga el código del estudiante, además de un cuadernillo de Jupyter con la misma funcionalidad. que alberga alternativamente el código del estudiante.
- **Programación de una solución de referencia para la autolocalización**.

Capítulo 3

Infraestructura

Este capítulo se ocupa de detallar los componentes y piezas software existentes en las que nos hemos apoyado en el desarrollo del trabajo. Profundizaremos en las plataformas sobre las que se apoya el trabajo (ROS y JdeRobot), en la simulación de las acciones del robot que realiza Gazebo, en las librerías de mayor importancia (OpenCV para la primera práctica y PyQt para las interfaces gráficas) y, por último, en el proyecto Jupyter.

3.1. Simulador Gazebo

Gazebo[1] es un simulador usado en robótica que permite emular diversos escenarios tridimensionales para robots autónomos (Figura 3.1). Es particularmente adecuado para probar algoritmos de elusión de objetos y de visión artificial. Cuando se trata de escribir el código que controlará un robot, especialmente si se está aprendiendo, es necesario realizar pruebas con el software, las cuales pueden ser muy costosas de implementarse en hardware real. Por ello, usamos simuladores que nos permitan comprobar en hardware virtual el comportamiento de nuestra lógica en todo momento, agilizando el desarrollo y abaratando costes.

El simulador utilizado en este TFG será Gazebo, el cual es de código abierto. Por su versatilidad (es capaz de simular robots, objetos y sensores en entornos complejos de interior y exterior), su interfaz de gran calidad, y su robusto motor de físicas (para describir componentes como la masa del robot, rozamiento, inercia, amortiguamiento, etc.), fue

[1] `http://gazebosim.org/`

elegido para realizar el DARPA Robotics Challenge (2012–2015) y está mantenido por la Fundación Open Robotics[2].

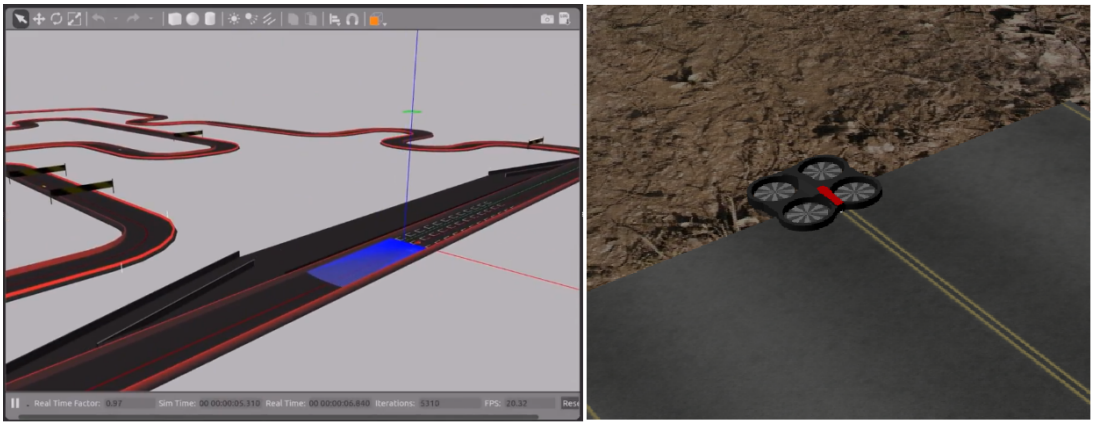

Figura 3.1: Ejemplo de mundo y modelo de Gazebo

Los mundos 3D simulados con Gazebo se describen en ficheros con extensión ".world", que no son más que ficheros XML (Extensible Markup Language) de descripción de documentos, definidos en el lenguaje Simulation Description Format (SDF), donde se recogen todos los elementos del escenario:

- Escena: Luz ambiente, propiedades del cielo, sombras, etc.
- Mundo: Representa el mundo como un conjunto de modelos, *plugins* y propiedades físicas.
- Modelo: Articulaciones, objetos de colisión, sensores, etc.
- Físicas: Gravedad, motor físico, paso del tiempo, colisiones, inercias, etc.
- *Plugins*: Sobre un mundo, modelo o sensor. Pueden crearse o utilizarse los disponibles en la red como *libroombaplugin.so*, que da acceso y control sobre todas las interfaces de programación de un cierto modelo de robot.

Todos y cada uno de ellos cuentan con una etiqueta propia. Los modelos de robots que se emplean en la simulación son creados mediante programas de diseño y modelado 3D como Blender o Sketchup. En la versión 7 de Gazebo se ha incorporado un editor de

[2] `https://www.openrobotics.org/`

modelos muy básico con el que se pueden crear robots y mundos básicos. A partir del modelo, se ha de asociar un *plugin* al mismo para recoger y publicar la información de los sensores, y para acceder a los actuadores y dotar al robot de inteligencia e interacción.

3.2. Entorno ROS

ROS (Robot Operating System) proporciona bibliotecas y herramientas para ayudar a los desarrolladores de software a crear aplicaciones de robots. Proporciona abstracción de hardware, controladores de dispositivo, bibliotecas, visualizadores, intercambio de mensajes y administración de paquetes entre otras cosas. Se distribuye bajo una licencia BSD de código abierto.

Uno de sus puntos fuertes es su integración con el simulador Gazebo a través de una serie de paquetes llamados *gazebo_ros_pkgs*[3] que proporcionan las interfaces necesarias para simular un robot en Gazebo usando *ROS Messages*, servicios y reconfiguración dinámica.

Este entorno se basa en la organización de una aplicación robótica como una colección de nodos, procesos que conllevan computación. Los nodos se combinan en un gráfico y se comunican entre sí mediante *topics* de transmisión, servicios RPC y el Servidor de Parámetros. Un sistema de control de un robot generalmente comprenderá muchos nodos, por ejemplo, un nodo controla un láser de largo alcance, otro nodo controla los motores de las ruedas, otro se encarga de la localización, se usa otro nodo para la planificación de rutas, un nuevo nodo proporciona una vista gráfica del sistema, y así sucesivamente. El uso de nodos en ROS proporciona varios beneficios para el sistema en general. Hay tolerancia adicional a fallos ya que los problemas están aislados en nodos individuales. La complejidad del código se reduce en comparación con los sistemas monolíticos.

En cuanto a los *topics* como forma de comunicación, se definen como buses sobre los cuales los nodos intercambian mensajes. Los *topics* tienen una semántica de publicación y/o suscripción anónima, que desacopla la producción de información de su consumo. En general, los nodos no saben con quién o quiénes se están comunicando. En cambio, los nodos que están interesados en ciertos datos se suscriben al *topic* relevante y por su parte aquellos nodos que generan datos de cierto tipo los publicarán en el *topic* correspondiente.

[3] `http://ros.org/wiki/gazebo_ros_pkgs`

Puede haber varios editores y suscriptores de un *topic*.

Dada la compatibilidad entre el entorno ROS, el simulador Gazebo y la plataforma JdeRobot, se puede hacer uso de *plugins* de ROS para las simulaciones que se comunicarán con el nodo académico de cada práctica a través de *topics*, por los que circulará información con el formato correspondiente.

Existe una gran variedad de *drivers* en la red para controlar las interfaces de muchos modelos, sensores y actuadores. Entre ellos, el *driver usb_cam* permite interactuar con cámaras USB estándar (por ejemplo, SonyEvi d100p o Logitech Quickcam) utilizando libusb_cam[4], una biblioteca multiplataforma para acceso genérico a dispositivos USB, y publica imágenes bajo el formato de mensajes *sensor_msgs::Image*[5] de ROS. Además, utiliza la biblioteca *image_transport*[6] para permitir el transporte de imágenes comprimidas. A través de él obtendremos las imágenes que proporciona la cámara en la primera de las prácticas que se va a crear. Otros ejemplos son *libgazebo_ros_laser* o *libgazebo_ros_diff_drive* pueden utilizarse para controlar las interfaces láser y motores respectivamente en un robot simulado.

3.3. Entorno JdeRobot

JdeRobot[7] es un *middleware* abierto para el desarrollo de aplicaciones con robots y visión artificial. Esta plataforma fue creada por el Grupo de Robótica de la Universidad Rey Juan Carlos en 2003 y está licenciada como GPLv3[8]. Su estructura ha sido desarrollada en C y C++, aunque algunas de sus componentes están escritas Python y JavaScript. El entorno que ofrece está basado en componentes, los cuales se ejecutan como procesos. Dichos componentes interoperan entre sí a través de *middlewares* de comunicaciones ICE o ROS, según sea la aplicación. Tanto ICE como *ROS-Messages* permiten la interoperación entre los componentes incluso estando desarrollados en diferentes lenguajes.

La plataforma facilita los *drivers* necesarios para poner en marcha los modelos

[4] `https://github.com/libusb/libusb/wiki`
[5] `http://docs.ros.org/api/sensor_msgs/html/msg/Image.html`
[6] `http://wiki.ros.org/image_transport`
[7] `https://jderobot.org/`
[8] `https://www.gnu.org/licenses/quick-guide-gplv3.html`

disponibles. Estos *drivers* están asociados a un dispositivo hardware del robot, ya sea sensor o actuador, e incluyen el correspondiente interfaz para acceder a él. Esto simplifica el acceso desde las aplicaciones a los diferentes componentes hardware, ya que con una simple función se puede enviar o recibir datos de ellos a través de interfaces ICE o ROS.

JdeRobot ofrece soporte a una amplia gama de dispositivos entre los que podemos encontrar cuadricópteros (AR Drone de Parrot), el robot Kobuki de Yujin Robot, el humanoide NAO de Aldebaran Robotics, cámaras firewire, USB e IP, los escáneres laser LMS de SICK y URG de Hokuyo, los simuladores Stage y Gazebo, sensores de profundidad como Kinect y otros dispositivos X10 de domótica. También incluye *drivers* específicos para control de cámaras y drones entre otros. Destaca para este trabajo el *driver EVIcam*, de utilidad para la primera de las prácticas que se va a incluir en el entorno, que ha sido desarrollado e incluido en la plataforma JdeRobot para teleoperar cámaras vía software a través de una conexión USB. En concreto, utiliza una interfaz ICE para poder controlarla a través de comandos de movimiento. El código fuente puede encontrarse en:

`https://github.com/JdeRobot/JdeRobot/tree/master/src/drivers/evicam_driver`

En las aplicaciones en JdeRobot se utilizan bibliotecas de software libre que son estándares de facto en su campo como OpenCV para visión, PCL para manejo de nubes de puntos o Eigen para álgebra. JdeRobot es compatible con ROS, en particular con la versión ROS-Kinetic, por lo que las aplicaciones pueden incorporar fluidamente nodos de ROS y conectarse a ellos.

Las prácticas que componen este trabajo se harán a través de la versión 5.6.1, última versión estable de la misma.

3.4. Lenguaje Python

Python[9] es un lenguaje de programación interpretado, orientado a objetos y de alto nivel con semántica dinámica. Suele destacar por su apariencia intuitiva, resultando en un lenguaje fácil de aprender. Su creador fue Guido van Rossum, un investigador holandés que trabajaba en el centro de investigación CWI (Centrum Wiskunde & Informatica). La primera versión surgió en 1991 pero no fue publicado hasta tres años después. Guido dio el nombre de Python a su lenguaje en honor a la serie de televisión *Monty Python's Flying*

[9] `https://www.python.org/`

Circus.

Sus estructuras de datos integradas de alto nivel, combinadas con el tipado dinámico y el enlace dinámico, lo hacen muy atractivo para el desarrollo rápido de aplicaciones, también para usarlo como scripting o como lenguaje para conectar componentes existentes. La sintaxis simple y fácil de aprender de Python enfatiza la legibilidad y, por lo tanto, reduce el costo del mantenimiento del programa. Python admite módulos y paquetes, lo que fomenta la modularidad del programa y la reutilización de código.

El intérprete de Python y la extensa biblioteca estándar están disponibles en formato fuente o binario sin cargo para las plataformas principales, y se pueden distribuir libremente.

La última versión ofrecida por Python Software Foundation, su administrador, es la 3.6.5. En este TFG vamos a emplear el lenguaje en su versión 2.7.12 junto con JdeRobot 5.6.1, que a su vez utiliza dicha versión por compatibilidad con ROS Kinetic. Usaremos este lenguaje para programar tanto los nodos académicos como las soluciones de referencia.

3.5. Biblioteca OpenCV

OpenCV[10] es una librería de código abierto desarrollada inicialmente por Intel y publicada bajo licencia de BSD. Esta librería implementa gran variedad de herramientas para el procesado de imagen y el aprendizaje máquina. Sus siglas provienen del inglés, y significan *"Open Source Computer Vision Library"*. Su propósito es facilitar la programación de aplicaciones de visión por computador en tiempo real.

Esta librería puede ser usada en MacOS, Windows, Android y Linux, y existen versiones para C#, Python y Java, a pesar de que originalmente era una librería en C/C++. Además, hay interfaces en desarrollo para Ruby, Matlab y otros lenguajes. OpenCV principalmente implementa algoritmos para las técnicas de calibración, detección de rasgos, seguimiento de caras, rastreo de objetos, análisis de la forma, análisis del movimiento, reconstrucción 3D, segmentación de objetos, reconocimiento, clasificación de acciones humanas en vídeos,... Los algoritmos se basan en estructuras de datos flexibles acopladas con estructuras IPL (*Intel Image Processing Library*), aprovechándose de la arquitectura de Intel en la optimización de la mayoría del paquete. También aprovecha la

[10] `https://opencv.org/`

aceleración de cómputo al poder usar tarjetas gráficas avanzadas (GPUs). Fue diseñado para tener una alta eficiencia computacional. Está escrito en C y puede aprovechar las ventajas de los procesadores *multicore*. Tantas son las ventajas y las posibilidades grandes compañías como Google, Yahoo, Microsoft, Intel, IBM, Sony, Honda o Toyota emplean la librería, que se ha convertido en el estándar de facto en el campo.

En este trabajo se ha empleado la versión 3.2 de OpenCV en Python. Esta librería se empleará para todo lo relacionado con el tratamiento de imágenes (análisis y procesado).

3.6. Biblioteca PyQt

PyQt[11] es un conjunto de enlaces Python para el conjunto de herramientas Qt, un *framework* multiplataforma orientado a objetos en C++ que permite desarrollar interfaces gráficas e incluye sockets, hilos, Unicode, bases de datos SQL, etc. PyQt combina todas las ventajas de Qt y Python, permite emplear todas las funcionalidades ofrecidas por Qt con un lenguaje de programación tan sencillo como Python. Fue desarrollado por Riverbank Computing Ltd y es soportado por Windows, Linux, Mac OS/X, iOS y Android.

En este proyecto se ha empleado la versión 5 de PyQt. PyQt5 es un conjunto de enlaces Python para Qt5, disponible en Python 2.x y 3.x. Tiene más de 620 clases y 6000 funciones y métodos. PyQt5 dispone de una licencia dual, es decir, los desarrolladores pueden elegir entre una licencia GPL (General Public Licence) o una licencia comercial. Las clases de PyQt5 se dividen en ciertos módulos, tales como QtCore, QtGui, QtWidgets, QtXml o QtSql. En las prácticas desarrolladas se ha hecho uso de los siguientes módulos para implementar las interfaces de usuario:

- QtCore: contiene las funcionalidades principales no relacionadas con la interfaz gráfica. Este módulo se emplea para trabajar con archivos, diferentes tipos de datos, hilos, procesos, url, etc.
- QtGui: contiene clases para el desarrollo de ventanas, gráficos 2D, imágenes y texto.
- QtWidgets: dispone de clases que proporcionan un conjunto de elementos de interfaz de usuario para crear interfaces de usuario clásicas de escritorio.

[11] `https://pypi.org/project/PyQt5/`

3.7. IPython 3.0 - Jupyter

Jupyter Notebook[12] es una aplicación web de código abierto que permite al usuario crear y compartir documentos que contengan códigos empotrados, ecuaciones, visualizaciones y textos narrativos. Entre sus usos destacan: limpieza y transformación de datos, simulación numérica, modelado estadístico, visualización de datos, aprendizaje automático, aunque su funcionalidad es mucho mayor. Jupyter es el nombre que desde hace poco se usa para referirse a IPython 3.0.

La herramienta principal de la aplicación son sus cuadernillos (*Notebooks*), que son los documentos producidos por Jupyter Notebook que contienen código de computadora (por ejemplo, Python), elementos de texto enriquecido (párrafos, ecuaciones, figuras, enlaces, etc.) y demás. Los documentos del cuaderno son documentos legibles por humanos que contienen la descripción del análisis y los resultados (figuras, tablas, etc.) así como documentos ejecutables para realizar análisis de datos. Los *Notebooks* consisten en una secuencia lineal de celdas. Hay cuatro tipos básicos:

- **Celdas de código**: entrada y salida del código en vivo que se ejecuta en el kernel.
- **Casillas de reducción**: texto narrativo con ecuaciones LaTeX incrustadas.
- **Encabezado de celdas**: 6 niveles de organización jerárquica y formato.
- **Celdas sin formato**: texto sin formato que se incluye, sin modificaciones, cuando los cuadernillos se convierten a diferentes formatos mediante *nbconvert*.

Así, la aplicación servidor-cliente permite editar y ejecutar estos *Notebooks* a través de un navegador web. La aplicación Jupyter Notebook se puede ejecutar en un escritorio local que no requiere acceso a Internet o puede instalarse en un servidor remoto y acceder a ella a través de Internet. Además de mostrar, editar y ejecutar documentos, la aplicación tiene un *Dashboard* (Panel de instrumentos del cuadernillo), que es similar a un "panel de control" que muestra los archivos locales y permite abrir documentos del cuaderno o cerrar sus núcleos (*kernels*).

Estos *kernels* son "motores computacionales" que ejecutan el código contenido en un *Notebook* (el núcleo IPython ejecuta el código Python). Existen núcleos o *kernels* oficiales

[12] `http://jupyter.org/`

para muchos lenguajes (Python, Julia, R, Ruby Haskell, Scala,...), e incluso versiones distintas de un mismo lenguaje de programación. Cuando se abre un *Notebook*, el *kernel* asociado se inicia automáticamente, de manera que al ejecutar cada celdilla con código, realiza el cálculo y produce los resultados. Dependiendo del tipo de cálculos, el *kernel* puede consumir CPU y RAM significativas, pudiendo ejecutar procesos más exigentes.

Estos documentos pueden ser salvados y almacenados en el sistema de ficheros local, como un documento con extensión *.ipynb*, para ser ejecutado cuando se quiera. Esto nos permitirá realizar una versión análoga a la que se hará en JdeRobot a través de Jupyter, con el código de la práctica empotrado, widgets interactivos y una pequeña guía para el alumno. Los *Notebooks* que haremos se basarán en código Python versión 2.7.

Capítulo 4

Ejercicio de Detección y seguimiento de caras con cámara PTZ

En este punto ya se han presentado contexto, los objetivos y las herramientas empleadas para llevar a cabo este proyecto. Este capítulo está reservado para detallar la realización de la primera de las dos prácticas creadas en y para el entorno JdeRobot. Se explicará la infraestructura desarrollada que le da soporte, el hardware involucrado y su forma de comunicación e interacción, el componente académico diseñado y la solución de referencia programada.

4.1. Enunciado

El objetivo para el estudiante en esta práctica es que una cámara real sea capaz de seguir con su movimiento caras de personas. La cámara sólo proporcionará sus imágenes captadas, a través de una capturadora de vídeo que irá proporcionando al nodo académico los fotogramas sucesivos. Además, la cámara dispone de dos actuadores de movimiento que controlarán sus giros en horizontal y vertical, a los cuales a partir de ahora llamaremos *Pan* y *Tilt* respectivamente.

En la solución canónica el alumno deberá programar un algoritmo de segmentación de caras de personas en la imagen y un algoritmo de control gradual en base a los datos que proporciona la imagen segmentada. Para ello, hay disponibles distintos elementos de depuración en el interfaz gráfico (GUI), ya que incluye espacios para la visualización de

la imagen captada por la cámara y de la imagen segmentada, accesibles con dos sencillos objetos Python.

El algoritmo canónico de control responde a un control reactivo, que en cada instante actuará en función de los datos de los sensores y sus propias variables internas. El control reactivo permitirá establecer en todo momento el movimiento del robot y responder ante situaciones imprevistas.

4.2. Infraestructura

En este apartado se introducirán los distintos componentes sobre los que se apoya la práctica, principalmente del hardware empleado y sus drivers controladores (Figura 4.1).

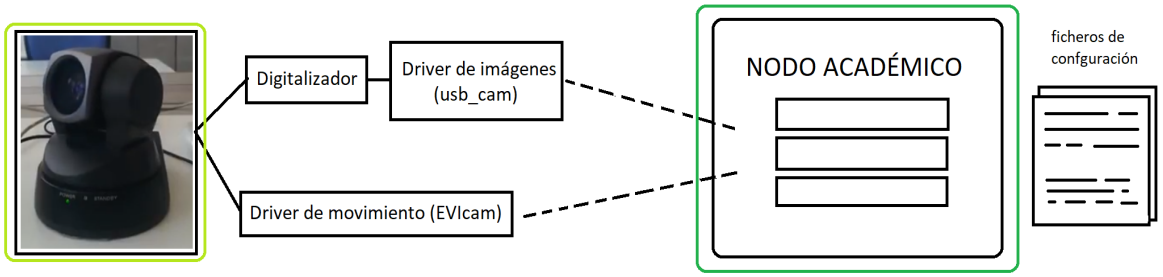

Figura 4.1: Infraestructura de Follow Face

4.2.1. Cámara Sony Evi d100p

El robot utilizado en esta práctica será la cámara Sony Evi modelo d100p, como el que se muestra en la Figura inferior (Figura 4.2), de la multinacional japonesa especializada en electrónica, *gaming* y entretenimiento.

Figura 4.2: Cámara Sony Evi d1oop

Figura 4.3: Digitalizador Dazzle DVD Recorder

Este modelo cuenta con las siguientes características:

- 440,000 pixeles que permiten tomas en alta resolución.
- Además de la acción de Pan y Tilt de alta velocidad, la mejora del mecanismo de reducción de ruido para vídeo en color.
- Posibilidad de operación desde computador a través del protocolo VISCA.
- Buffer de mensajes que permite memorizar hasta 6 combinaciones de la posición y el estado de la cámara.
- Control remoto multifunción.
- Compatibilidad de comandos con modelos anteriores.

Dado que el hardware captura vídeo analógico, en combinación con la cámara se ha usado una capturadora de vídeo, que es un periférico de entrada cuya función es recibir señales de audio/video procedentes de dispositivos analógicos y convertirlas en señales digitales que puedan ser almacenadas o manipuladas por medio de software.

La capturadora utilizada es Dazzle DVD Recorder HD de la marca Pinnacle (Figura 4.3), y con conexión USB 2.0 con el ordenador. En cuanto a las capacidades del hardware en conjunto, nos interesan:

El valor de *Pan* puede estar entre -164º y 164º y su velocidad puede tomar valores en un rango donde 1 es el mínimo y 24 es el máximo. El valor de *Tilt* puede oscilar entre -30º y 30º, con velocidades en una escala de 1 a 20.

La cámara utilizará *drivers* para el acceso a la imagen y a los actuadores y para la comunicación con el nodo académico.

4.2.2. Ficheros de configuración

Al haber varios *drivers* involucrados, son necesarios ciertos ficheros de configuración en el nodo académico para calibrar las conexiones con ellos. Dado que el primero de ellos es un driver de ROS (*usb_cam*), emplea la herramienta *roslaunch*[1], que permite lanzar fácilmente múltiples nodos ROS de manera local o remota. Para ello necesita un fichero de configuración XML como el de la Figura 4.4:

```
1  <launch>
2    <node name="usb_cam" pkg="usb_cam" type="usb_cam_node" output="screen" >
3      <param name="video_device" value="/dev/video1" />
4      <param name="image_width" value="640" />
5      <param name="image_height" value="480" />
6      <param name="pixel_format" value="yuyv" />
7      <param name="camera_frame_id" value="usb_cam" />
8      <param name="io_method" value="mmap"/>
9    </node>
10 </launch>
```

Figura 4.4: Fichero de configuración de usb_cam

En él deben especificarse los parámetros del nodo o nodos a establecer, así como los dispositivos sobre los cuales se lanza el nodo, en nuestro caso, la cámara (*/dev/video1*).

El otro *driver* involucrado (*EVIcam*) ha de lanzarse también con un fichero de configuración que indique el par *IP:puerto* en el que la cámara espera órdenes y el dispositivo. Un ejemplo se muestra en la Figura 4.5:

```
1  PanTilt.Endpoints=default -h 0.0.0.0 -p 9977:ws -h 0.0.0.0 -p 11077
2  PanTilt.Device=/dev/ttyUSB0
```

Figura 4.5: Fichero de configuración de evicam_driver

[1] `http://wiki.ros.org/roslaunch`

4.3. Nodo Académico

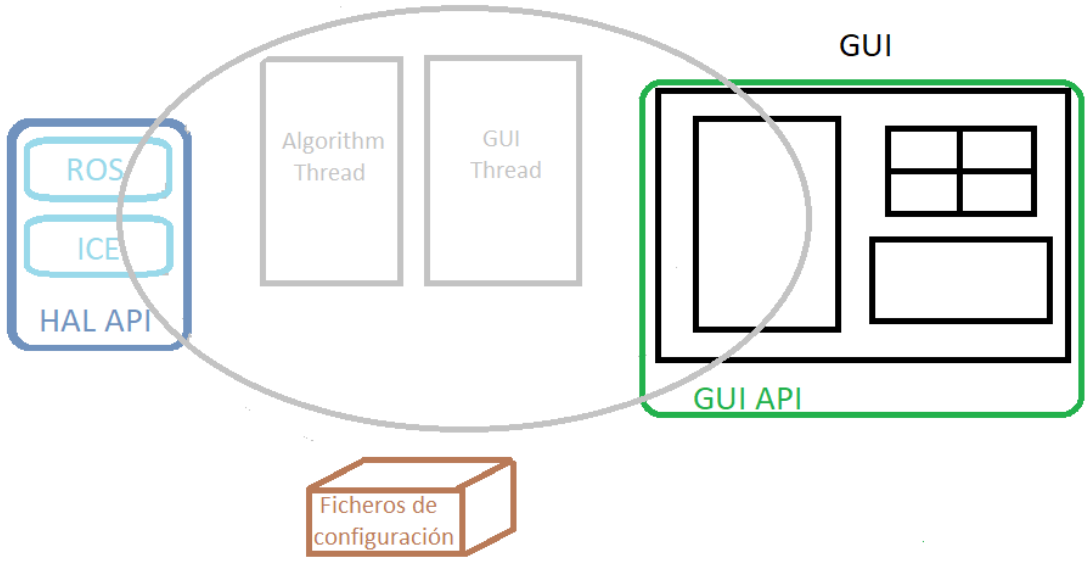

Figura 4.6: Diseño de Nodo Académico de Follow Face

El componente académico de esta práctica (Figura 4.6) resuelve varias funcionalidades auxiliares que sirven de gran ayuda para poder abordarla:

a) Proporciona un HAL-API, API de la *Hardware Abstraction Layer*, es decir, los sensores y actuadores de la cámara en este caso, en forma de métodos simples (oculta el *middleware* de comunicaciones con el hardware);

b) Ofrece una interfaz gráfica al usuario que le ayuda a depurar su código, y proporciona un GUI-API, un interfaz simple para que el estudiante visualice cosas interesantes en este ejercicio;

c) Incluye código auxiliar que no es el foco del algoritmo y que ayuda a programar la solución.

El nodo deja todo preparado para que el estudiante sólo tenga que incluir su código retocando el método *execute* en el fichero *MyAlgorithm.py*, e ir realizando las pruebas pertinentes.

4.3.1. Arquitectura software

En cuanto a la capacidad de cómputo para realizar todas las tareas simultáneamente, el componente emplea dos hilos de ejecución para agilizar la respuesta de los distintos módulos:

→ Hilo de algoritmo de percepción y control: se encarga del refresco de la ejecución del algoritmo, ya que este se ejecuta de modo cíclico. El tiempo de refresco de este hilo es muy importante, dado que su velocidad permitirá reaccionar ante situaciones imprevistas, y adaptar el movimiento de la cámara a la velocidad de movimiento de las personas, para poder seguir su cara. Es por eso que el tiempo de refresco es de 80 ms.

→ Hilo de la interfaz gráfica de usuario (GUI): Este hilo es el encargado de actualizar la interfaz gráfica. También consta de los manejadores de eventos del GUI. El intervalo de actualización de la interfaz debe ser pequeño, ya que se tiene que mostrar las imágenes captadas por la cámara en forma de elemento vídeo, así como las imágenes segmentados en tiempo real. Por ello, se ha fijado el tiempo de refresco a 50 ms.

4.3.2. Interfaz de sensores y actuadores

El componente ofrece un API de cámara y actuadores al programador con el cuál acceder a la funcionalidad que necesita:

- *self.camera.getImage()*: devuelve la imagen capturada por la cámara activa (local o cámara usb).
- *self.motors.getLimits()*: método a través del cual se pueden obtener los límites de Pan y Tilt (en grados).
- *self.motors.setPTMotorsData(pan, tilt, panSpeed, tiltSpeed)*: método para enviar comandos de *Pan* y *Tilt* a la cámara. Necesita los valores respectivos en grados y la velocidad horizontal y vertical para alcanzar la posición fijada.

4.3.3. Interfaz gráfica y uso desde código

La interfaz gráfica de usuario (GUI) de la práctica sirve para representar información importante para desarrollar correctamente el algoritmo que lleve a la solución, además de para proporcionar funcionalidad básica para iniciar o para el algoritmo o para teleoperar la cámara. Se ha programado en Python con PyQt5, y es la mostrada en la Figura 4.7.

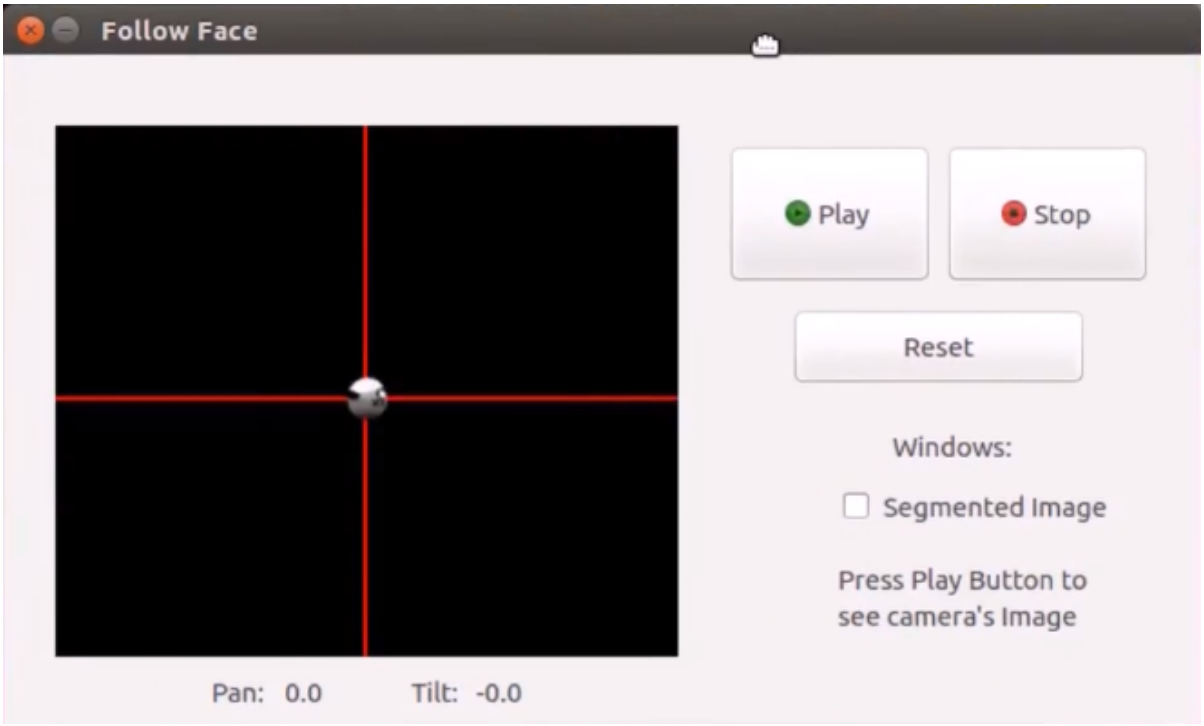

Figura 4.7: Interfaz Gráfica de Follow Face

Se puede ver en ella, a la izquierda, el teleoperador para comprobar que el movimiento de la cámara este operativo. Se trata de un dial bidimensional que controla las componentes *Pan* y *Tilt* del robot. Únicamente será necesario mover el *joystick* central en sentido vertical, para controlar el ángulo de *Pan* y en sentido horizontal para fijar el *Tilt*. Cuánto más se acerque el *joystick* a los límites del teleoperador, más acentuado será el valor de la componente hacia la que se esté movimiento el mismo, ya sea éste un valor negativo si se mueve hacia abajo o hacia la izquierda, y positivo en caso de que se mueva hacia arriba o a la derecha. A su derecha, se han colocado varios botones cuya funcionalidad es la de iniciar el algoritmo, pararlo o resetearlo. Recordemos que la cámara dispone de un *buffer* de mensajes, de manera que pulsar el botón Stop se traducirá en pausar el algoritmo, pero la cámara puede continuar en movimiento si había encolado

mensajes, hasta que vacíe dicho *buffer*. Inmediatamente debajo tenemos un *check* que muestra la herramienta de imagen (Figura 4.8), en la cual se pueden ver las imágenes que provee la cámara (a la izquierda) y la imagen segmentada (a la derecha, en caso de haberla establecido):

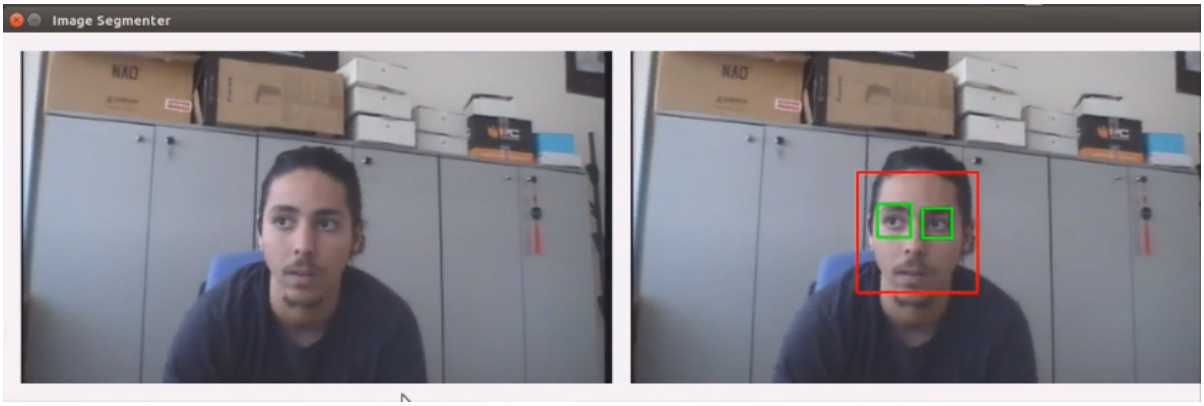

Figura 4.8: Visualización de imagen procesada en el GUI

Los métodos que pone a disposición del alumno son los siguientes:

- *self.camera.setColorImage(img)*: método que recibe una imagen y la muestra en el espacio dedicado del interfaz para imágenes capturadas.
- *self.camera.setThresholdImage(img)*: método que recibe una imagen tratada y la muestra en el espacio reservado en el GUI para imágenes filtradas, umbral o en blanco y negro.

Con ellos el estudiante podrá establecer imágenes en el GUI desde su código implementado.

Además, dentro del nodo académico existe la clase *CameraSegment*. Esta clase se ha creado por controlar y establecer la visualización de las imágenes que tienen que ver con la práctica, sustituyéndolas por imágenes negras si son nulas o estableciendo el vídeo capturado por la cámara y el vídeo segmentado creado y seleccionado por el programador.

El objeto *self.camera* del cual dispone también el algoritmo de percepción y control se construye a partir de ésta clase, por lo cual incluye algunos métodos propios además de los esperados por la cámara. Hablamos de las funciones que permiten mostrar las imágenes en el interfaz, descritas en 4.3.3. Por tanto, actúa de nexo entre la herramienta de segmentación (*segment widget*) y el algoritmo, proporcionando un API fácil de utilizar.

Por último, esta clase actúa de intermediario entre la cámara y el interfaz gráfico, conectando ambos componentes. Esto es necesario dado que las imágenes capturadas por la cámara y las que espera el interfaz gráfico no son completamente compatibles: para poder ser visualizadas, las imágenes deben redimensionarse para encajar en la aplicación gráfica.

4.3.4. Fichero de configuración

El nodo académico requiere un fichero de configuración que indique los *endpoints* que utilizan la cámara y los motores PT. Este archivo tiene extensión *.yml*, lo cual implica que debe seguir el formato de serialización de datos YAML, el cual produce datos fáciles de leer por humanos y máquinas para todos los lenguajes de programación. Tiene el siguiente aspecto:

```yaml
Follow_face:

  PTMotors:
    Server: 1 # 0 -> Deactivate, 1 -> Ice
    Proxy: "PanTilt:default -h localhost -p 9977"
    Topic: "/usb_cam/PTMotors"
    Name: FollowFacePTMotors

  Camera:
    Server: 2 # 0 -> Deactivate, 1 -> Ice , 2 -> ROS
    Proxy: "cameraA:default -h localhost -p 9999"
    Format: RGB8
    Topic: "/usb_cam/image_raw"
    Name: FollowFaceCamera

  NodeName: Follow_face

```

Figura 4.9: Fichero de configuración YAML de FollowFace

Se puede ver claramente que las interfaces empleadas por cámara y motores son distintas, siendo en el primer caso ROS y en el segundo ICE. Se observa también que puede incluir información adicional, como el formato de imagen.

Por último, el nodo académico incluye dos ficheros de descripción XML, que servirán

de entrenamiento para los clasificadores que necesitaremos en la segmentación, explicados con mayor detalle en el apartado 4.4.1.1.

4.4. Solución de referencia

En esta sección abordaremos brevemente los fundamentos de segmentación y detección de caras y el control reactivo gradual de hardware. Con ello construiremos una solución que se establecerá como solución de referencia, pero que sólo es uno de los muchos posibles métodos con el cual alcanzar el fin. El código quedará recogido en el fichero *MyAlgorithm.py*, que es de naturaleza iterativa, de manera que en cada iteración se percibe, se actúa en consecuencia y se controla. Se implementa dentro del método *execute*, el cual debe contener la lógica que se ejecuta de manera cíclica en el hilo de percepción y control.

4.4.1. Detección de caras

Las cámaras que incorporan los robots son una de las principales fuentes de información con las que cuentan, pero toda esta información es inútil sin la lógica que sea capaz de extraer la información relevante de la imagen. Algoritmos incapaces de diferenciar lo necesario de lo inane sólo conseguirán sobrecargar a la máquina de información inoperante, reduciendo su rendimiento e incluso llegando a producir comportamientos indeseados.

La detección de rostros se puede considerar como un caso específico de detección de "clases de objetos". En la detección de clase de objeto, la tarea es encontrar las ubicaciones y tamaños de todos los objetos en una imagen que pertenecen a una clase determinada. Algunos ejemplos muy comunes en casos reales son torsos superiores, peatones y automóviles. De la misma forma, debemos ser capaces de encontrar caras en una imagen digital, sea cual sea su tamaño o ubicación en ella. Existen muchos algoritmos de detección de rostros para ubicar a una persona en una escena, algunos más fáciles y otros más complejos:

Cuando la imagen consta de un fondo monocolor simple o un fondo estático predefinido (conocido), es muy fácil detectar caras, dado que eliminar dicho fondo siempre tendrá como resultado los límites de la cara. Sin embargo, sabemos que en nuestro caso el robot

debe comportarse correctamente en cualquier escenario, de manera que no es una solución válida.

Otra manera pasa por utilizar el color de piel típico para encontrar segmentos faciales, si se dispone de imágenes en color. La desventaja en este caso es que no funcionará con todo tipo de colores de piel, y que su robustez es mínima ante condiciones de iluminación variables.

Aunque resulte sorprendente, la forma de abordar esta tarea con mayor éxito es utilizar imágenes en escala de grises, con lo cual reducimos el coste computacional. A su vez, hay varios métodos que las emplean, como son la detección basada en modelo, en emparejamiento de bordes por orientación, usando la distancia de Hausdorff[2],etc. Pero el más utilizado es mediante el uso de "clasificadores en cascada" encapsulado dentro de las técnicas de aprendizaje máquina que, utilizando características simples de Haar, puede producir resultados impresionantes.

Hay muchas motivaciones para usar características en la comparación en lugar de hacerlo con los píxeles directamente. La razón más común es que las características pueden codificar conocimiento del dominio *ad-hoc*, lo cual es difícil usando una cantidad finita de datos de entrenamiento. Para este sistema también hay una segunda motivación crítica para emplear características: el sistema basado en características funciona mucho más rápido que un sistema basado en píxeles.

Cada "ventana" de la Figura 4.10 se coloca en la imagen a segmentar para calcular una sola característica. Esta característica es un valor único que se obtiene al restar la suma de píxeles debajo de la parte blanca de la ventana de la suma de los píxeles debajo de la parte negra de la ventana. Así, todos los tamaños posibles de cada ventana se colocan en todas las ubicaciones posibles de cada imagen para calcular muchas características. Por ejemplo, en la Figura 4.10 estamos extrayendo dos características. La primera se centra en la propiedad de que la región de los ojos a menudo es más oscura que el área de la nariz y las mejillas. La segunda característica se basa en la propiedad de que los ojos son más oscuros que el puente de la nariz.

[2] `https://en.wikipedia.org/wiki/Hausdorff_distance`

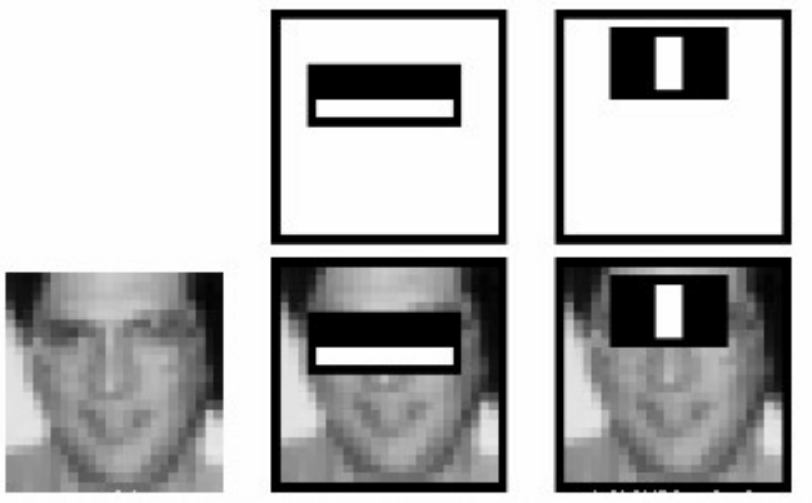

Figura 4.10: Extracción de características

Entre todas estas características calculadas, la mayoría de ellas son irrelevantes. Por ejemplo, cuando se usa en la mejilla, las ventanas se vuelven irrelevantes porque ninguna de estas áreas es más oscura o más clara que otras regiones en las mejillas, todos los sectores en ella son iguales. Se debe descartar de inmediato las características irrelevantes y conservar únicamente aquellas importantes con una técnica llamada *Adaboost* que selecciona sólo aquellas características conocidas para mejorar la precisión de clasificación (cara / no cara) del clasificador. Verifica si una ventana es una región sin rostro, y si no lo es, la desecha de inmediato para no volverla a procesar. Así el clasificador es capaz de enfocarse principalmente en el área donde exista una cara. Es por eso por lo que reciben el nombre de clasificadores en cascada (Figura 4.11), dado que se usan en primera instancia clasificadores simples para descartar esas áreas y luego clasificadores más complejos para eliminar falsos positivos.

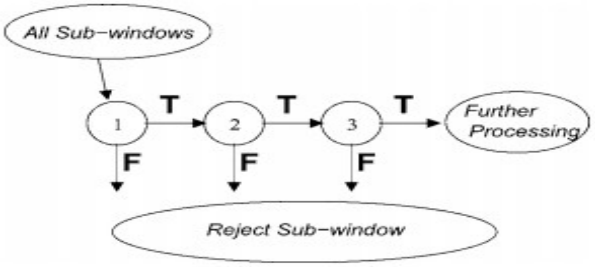

Figura 4.11: Clasificadores en cascada

4.4.1.1. Clasificadores Máquina

La manera escogida para abordar la tarea de detección en nuestra solución de referencia pasa por utilizar un clasificador en cascada. Estudiando la librería OpenCV para Python, hemos encontrado la manera de crear entrenadores y detectores, los cuales se pueden entrenar de la forma que se desee. Además la librería incluye sistemas pre-entrenados para detectar facciones concretas de la cara como ojos, sonrisa, cejas e incluso la cara en sí.

Estos sistemas de entrenamiento se encuentran en ficheros XML provistos por la librería, dos de los cuales se han incluido en el nodo académico de esta práctica para poder abordarla siguiendo este método propuesto por Paul Viola y Michael Jones en su *paper "Rapid Object Detection using a Boosted Cascade of Simple Features"* en 2001. Estos ficheros son:

haarcascade_frontalface_default.xml → entrenamiento para detectar caras en posición frontal.

haarcascade_eye.xml →entrenamiento para clasificador de ojos.

Con ellos es suficiente para realizar una detección y segmentación genérica en condiciones normales. El alumno tiene la posibilidad de entrenar todos los clasificadores que quiera para su programa, simplemente incluyendo los ficheros *.xml* necesarios en el directorio de trabajo. Una vez obtenidos, simplemente se invocará la función:

```
face_cascade = cv.CascadeClassifier('haarcascade_frontalface_default.xml')
                                    o
eye_cascade = cv.CascadeClassifier('haarcascade_eye.xml')
```

Para obtener un objeto Python con el clasificador deseado.

4.4.1.2. Eliminación de falsos positivos

Usar clasificadores en cascada es la mejor manera de reducir la tasa de falsos positivos en la detección. Sin embargo, bajo condiciones cambiantes de luminosidad o entornos con

mucho contenido se tienen resultados etiquetados como positivos en casos que claramente no lo son (Figura 4.12):

Figura 4.12: Falso positivo

Proporcionar al sistema un entrenamiento más meticuloso sería la opción ideal para reducir aún más la tasa de error. Sin embargo, las prácticas en este entorno buscan que el alumno se enfrente únicamente a la lógica de control y obtención de datos de los robots, de manera que este problema debe abordarse con código de filtrado de los resultados.

Tras obtener la posición de todas las caras de la imagen con el método *detectMultiScale* del objeto clasificador de OpenCV (en nuestro caso, *face_cascade*), aplicamos un filtrado que elimine positivos de tamaño inferior a un umbral mínimo, o superior a un umbral máximo para mejorar los resultados. Además, incluimos un clasificador para la detección de ojos que resultará más restrictivo a la hora de hacer una detección final. Finalmente, se utiliza la función *rectangle* del paquete con las coordenadas de la cara para hacer visual la detección. Una vez aplicado el filtro, el resultado cambia (Figura 4.13):

Figura 4.13: Falso positivo solucionado

4.4.2. Control del movimiento de la cámara

En las cámaras PTZ estas siglas indican el control que se puede ejercer sobre la máquina, siendo que se puede modificar el valor que toma en *Pan* (P), el que toma en *Tilt* (T) y el que toma de *zoom* (Z). Nuestra cámara no dispone de *zoom*, se podría decir que consta de un control PT.

Ahora bien, tratándose de control en tiempo real, ¿puede tomar cada componente cualquier valor dentro de los límites del hardware? No, debemos contar con el tiempo físico que tarda la cámara en ejecutar el movimiento solicitado. También debe tenerse en cuenta el *buffer* asociado, que encola mensajes hasta que puedan ser atendidos. Si se envía la misma orden dos veces antes de que el hardware sea capaz de terminar la primera (algo muy común), el resultado se traducirá en un "temblor" u oscilación de la cámara entorno a esos valores.

Es por eso que hemos hablado de control gradual. Se hace necesario que el movimiento del cuello mecánico sea a la vez fluido y suave, sin ejecutar movimientos demasiado bruscos, y sin oscilaciones que repercutan en la calidad de la imagen captada, para que el algoritmo de visión pueda seguir funcionando de manera óptima durante los movimientos. Por ello, hemos decidido implementar un control (Figura 4.14) que trabaje en incrementos

o sustracciones pequeñas en grados de los valores *Pan* y *Tilt*. Con ello conseguimos también aprovechar la máxima velocidad de movimiento que ofrece el hardware, sin balanceos, obteniendo un movimiento adecuado en combinación con el tiempo de refresco del hilo de ejecución de algoritmo. En cada iteración se calcula el centro de la cara detectada y el control trabaja para que dicho centro coincida con el centro de la imagen captada. En caso de no ser así, se obtendrá el error de posición y se modificarán los valores de control de la cámara de la forma conveniente. En caso de coincidencia, se detendrán los motores y se hará que la cámara permanezca en espera al próximo movimiento.

```
limits = self.motors.getLimits()
if len(faces)>0:
    self.first_time = True
    width = self.camera.client.getImage().width
    height = self.camera.client.getImage().height
    imgXCenter = width/2
    imgYCenter = height/2

    #Find out if the X component of the face is to the left of the middle of the image.
    if(center[0] < (imgXCenter - 60)):
        self.pan -= 2
        print str(center[0]) + " > " + str(imgXCenter) + " : Pan Right : " + str(self.pan)
     #Find out if the X component of the face is to the right of the middle of the image.
    elif(center[0] > (imgXCenter + 60)):
        self.pan += 2
        print str(center[0]) + " < " + str(imgXCenter) + " : Pan Left : " + str(self.pan)
    else:
        print str(center[0]) + " ~ " + str(imgXCenter) + " : " + str(self.pan)

    # Consider min and max Pan value of the camera
    self.pan = min(self.pan, limits.maxPan)
    self.pan = max(self.pan, limits.minPan)
```

Figura 4.14: Código del control de la cámara

4.4.3. Gestión de la "No Detección"

Hemos programado el comportamiento en caso de detección, la eliminación de falsos positivos, la lógica de control y la segmentación pero, ¿qué hace el programa propuesto en caso de no detectar caras en la imagen? Existen varios caminos posibles: el primero sería simplemente no hacer nada. El hardware permanece en espera a que una persona entre en escena. Se establece un módulo condicional en el código que detiene los motores en la

posición actual si no detecta rostros. Otra opción sería hacer que el harware retornase a la posición de reposo, pero la solución por la que nos hemos decantado ha sido por iniciar un proceso de búsqueda de rostros dentro del "campo de visión" de la cámara.

Cuando en la imagen no se encuentran secciones que casen con una cara, la cámara esperará 5 segundos como margen para seguir comprobando en las siguientes iteraciones si realmente no hay caras visibles, o si por el contrario se trataba de un error de clasificación. Pasados estos 5 segundos, la cámara comienza un movimiento horizontal (a lo largo del eje X) hacia la derecha, dado que este eje es en el que más probabilidad existe de encontrar una cara, no sólo por las características de las personas (no podemos desplazarnos en vertical, volar), sino también porque es el eje con mayor recorrido del hardware: 328º frente a 60º. Mientras se desplaza sigue ejecutando el algoritmo de detección, de manera que en caso de obtener un positivo, se continúa con la ejecución normal de seguimiento y, en caso contrario, se efectúa el movimiento hasta que se llegue al extremo. Una vez en el extremo, efectúa el mismo movimiento hacia el otro lado. La cámara permanecerá en movimiento en todo momento hasta la detección de una nueva cara.

4.5. Experimentación

En este apartado validaremos la funcionalidad descrita en los apartados anteriores a través de experimentos realizados para poner a prueba tanto la infraestructura de la práctica, como el nodo y la solución.

4.5.1. Ejecución típica

Se ha preparado un documento de texto (*README.md*) que sirve de guía para que alumno no encuentre dificultades durante la ejecución y el testeo, que incluye también información del API de utilización. La forma de ejecutar la práctica es la siguiente:

1. Empleamos un primer terminal para inicial el driver de ROS *usb_cam* (previamente habrá que asegurarse de haber instalado el paquete *ros-kinetic-usb-cam packet*):

```
$ roslaunch usb_cam-test.launch
```

2. Utilizamos un segundo terminal para iniciar el driver *evicam_driver*:

```
$ evicam_driver evicam_driver.cfg
```

3. Por último, se lanza en otro terminal el nodo académico *follow_face*:

```
$ python2 ./follow_face.py follow_face_conf.yml
```

El resultado de la ejecución de la solución canónica se muestra en la Figura 4.15. Además, dos vídeos demostrativos del comportamiento pueden verse en [3] y [4]

[3] https://www.youtube.com/watch?v=pdSPnftumf8

[4] https://www.youtube.com/watch?v=YZEMIXkXwM0

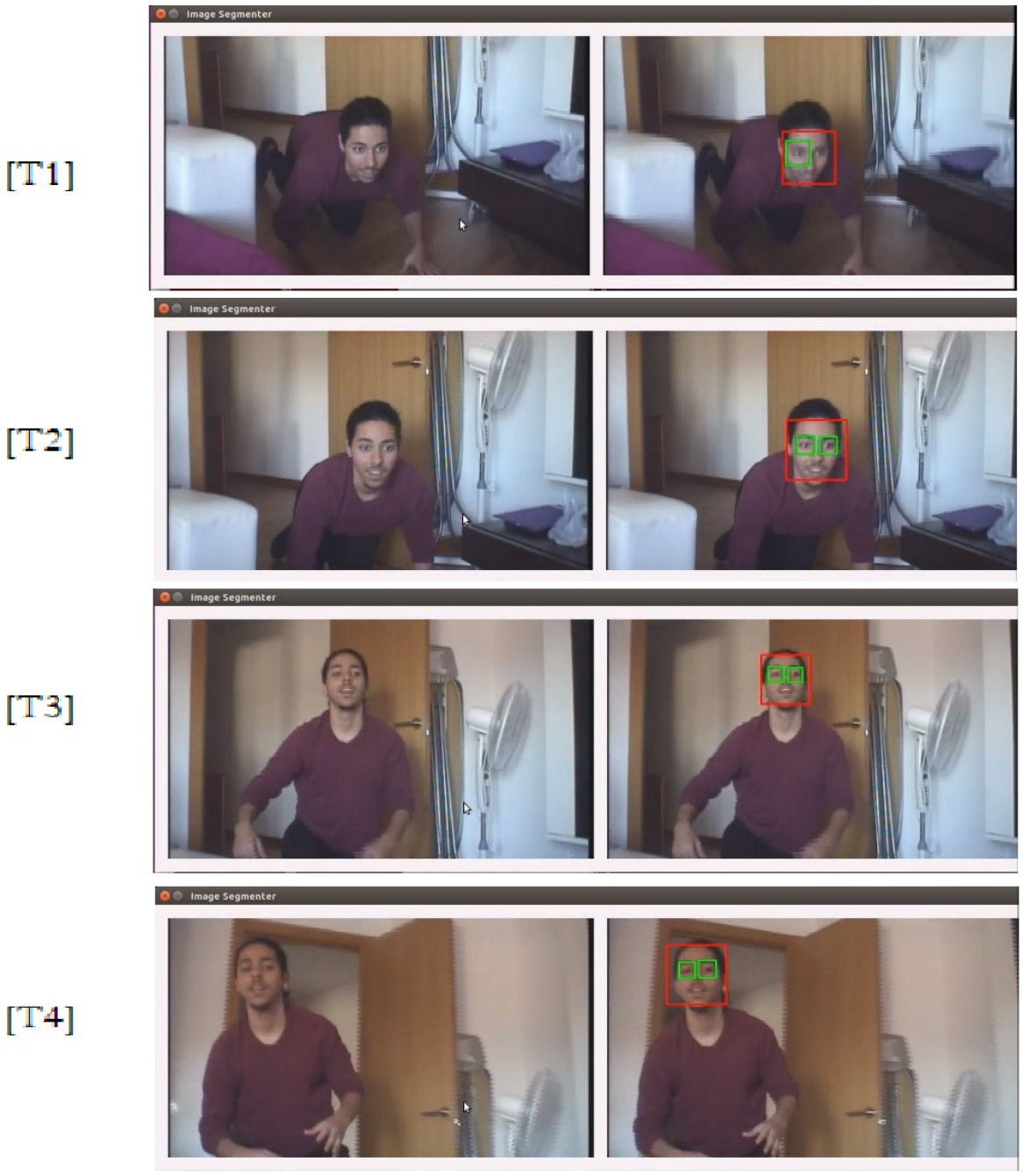

Figura 4.15: Output Follow Face

La cámara persigue con éxito la cara segmentada en caso de haberla a través tanto de movimientos horizontales como verticales en función de las coordenas obtenidas en la imagen, ejecutando movimientos suaves y controlados. Cuando no detecta rostros comienza el proceso de búsqueda hasta encontrar una nueva cara.

4.5.2. Comportamiento ante casos extremos

Hemos tratado de poner el algoritmo a prueba para comprobar su validez. Una primera situación que debimos probar fue la ejecución con movimiento rápido. La persona al frente de la cámara se movió de manera brusca, cambiando de dirección y con subidas y bajadas periódicas. Gracias al movimiento gradual implementado, el algoritmo es capaz de seguir en todo momento el movimiento, salvo en aquellos casos en los que el sujeto se sale bruscamente del plano.

Por otro lado, probamos la ejecución en situaciones de contraluz. En este caso, el comportamiento empeoraba, ya que la imagen captada, al convertirse a escala de grises, no mostraba apenas diferencia de niveles en píxeles que conforman secciones de la cara que normalmente si presentan dicha diferencia. Esto se debe a las bases de la detección con clasificadores en cascada. Se podría tratar de solucionar incorporando técnicas de detección de color de piel, entre otras.

Por último, en situaciones con muchos candidatos a caras (posters, muñecos y elementos con formas abstractas), el algoritmo se equivocaba el 5 % de las veces, siendo que en el resto de casos escogía el candidato correcto para seguir.

4.6. Cuadernillo académico Jupyter

En el apartado 3.7 se introdujo la herramienta Jupyter Notebook, que permite al usuario crear y compartir documentos que contengan multitud de elementos y widgets programables a través de distintos lenguajes. Hemos ampliado el alcance inicial de esta práctica preparando, además de la versión con nodo académico en python, una versión en un cuadernillo Jupyter. Esto permite que el estudiante no tenga que editar el fichero de python directamente, sino utilizar el navegador web para ello. Esto abre puertas en JdeRobot-Academy hacia versiones multiplataforma de los ejercicios, especialmente los que involucran al simulador Gazebo.

Antes de comenzar, debe instalarse el código del proyecto, proceso del que se detallen las instrucciones en la página oficial del proyecto[5].

Hemos preparado un *Notebook* de Jupyter al que hemos llamado *follow_face.ipynb*,

[5] `http://jupyter.org/install`

que junto con algunos cambios en la estructura del nodo académico proporcionan el soporte necesario para tener la misma práctica disponible a través de esta aplicación. En concreto, el nodo programado en el fichero *follow_face.py* de la práctica original ha sido recodificado para tener estructura de clase Python: la clase *FollowFace*, para poder ser instanciada desde el código del cuadernillo, que detallaremos un poco más adelante. Los cambios más significativos se producen al eliminar la interfaz gráfica directa, dado que será ahora la propia aplicación web la que actúe de interfaz de usuario. Eliminados los componentes gráficos y los widgets que mostraban las imágenes en él, se ha creado una nueva clase (la clase *Printer*, Figura 4.16) que permite visualizar las imágenes que el alumno establece a través de métodos disponibles en la clase nodo (*FollowFace*) en la interfaz de Jupyter. Para ello utilizamos los métodos disponibles en la librería *matplotlib* y su módulo *pyplot*, una colección de funciones cuyo estilo es como un comando que hacen que *matplotlib* funcione como MATLAB en cuanto a representaciones, lo cual simplifica la visualización en la herramienta.

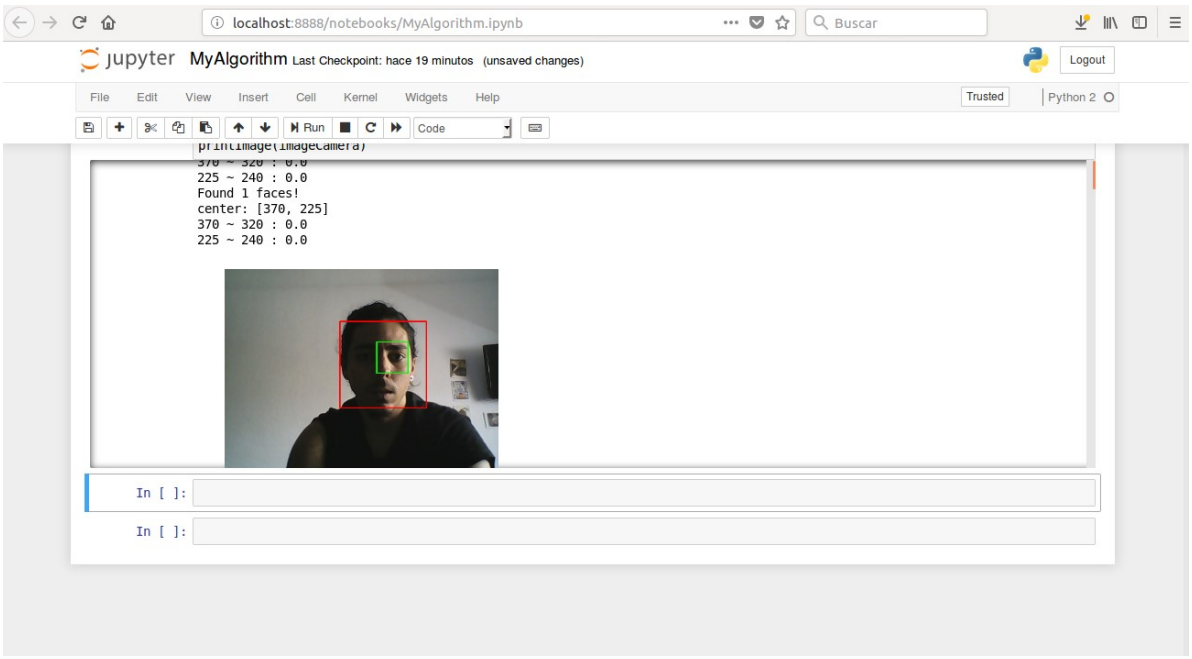

Figura 4.16: Printer

Para asociar el código del cuadernillo con el del nodo existente se ha usado la librería *types* de Python. Esta librería proporciona acceso a los nombres de algunos tipos de objetos que son utilizados por el intérprete estándar de Python, de manera que podemos modificar el comportamiento normal de los mismos. Utilizaremos la función *MethodType* para hacer

corresponder la función que el estudiante retoca desde la aplicación de Jupyter con el método que realiza el cómputo en el hilo de algoritmo del nodo académico modificado.

Por último, necesitamos de la intervención de dos *drivers* para controlar la cámara de la que disponemos, de manera que la adaptación del nodo incorpora el lanzamiento de dos subprocesos, uno para cada *driver*, a través de la librería *subprocess*, que permite generar nuevos procesos y conectarse a sus tuberías de entrada / salida / error, además de obtener sus códigos de retorno. Con esto, y unas pequeñas incorporaciones al nodo para poder parar o ejecutar el algoritmo desde Jupyter, completamos la práctica a través de la aplicación web.

El cuadernillo generado (Figura 4.17) no sólo incorpora celdillas de código ejecutable embebidas, sino que también incluye imágenes representativas y texto descriptivo para orientar al alumno en todo momento (durante la ejecución, aportando información adicional para abordar la práctica y explicando el API de utilización con los nuevos métodos de visualización y actualización del código.

Así, el alumno puede realizar la práctica de manera totalmente guiada, y ejecutar los distintos pasos que va programando para observar su salida con herramientas disponibles, para depurar errores o matizar su solución.

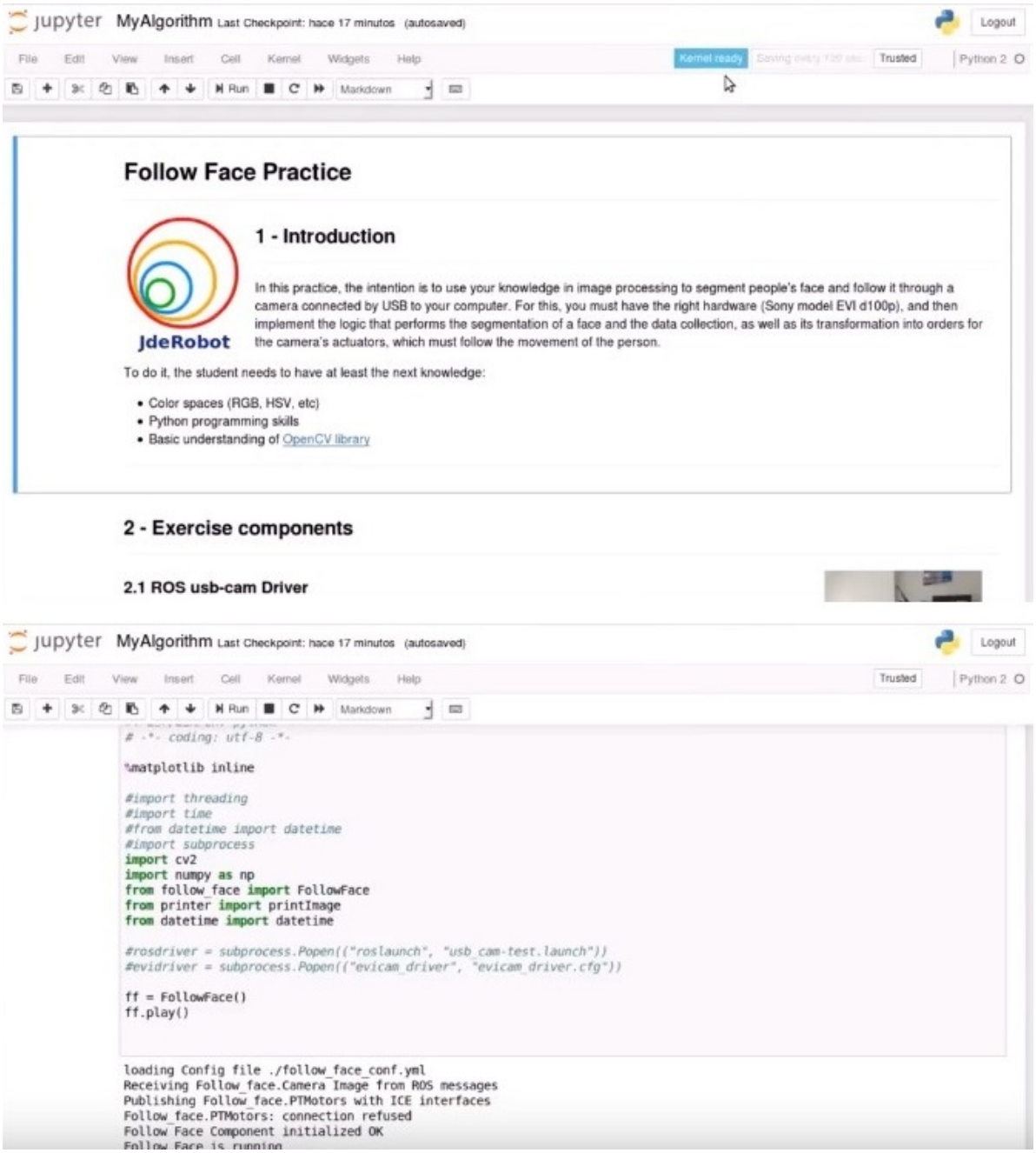

Figura 4.17: Notebook Follow Face

Un vídeo demostrando el comportamiento de la práctica a través de Jupyter se puede ver en [6].

[6] `https://www.youtube.com/watch?v=9fOqkw7fUHg`

Capítulo 5

Ejercicio de Autolocalización con Láser

Este capítulo recoge el proceso de elaboración y testeo de la segunda práctica creada para el entorno de aprendizaje JdeRobot-Academy. En él, se explicará el desarrollo que ha sido necesario para su infraestructura, las características de simulación bajo las que se ha creado la práctica, el nodo académico y la solución de referencia programada, así como la versión en cuadernillo Jupyter del ejercicio.

5.1. Enunciado

El objetivo de esta práctica es que el alumno entre en contacto con uno de los problemas clásicos de la robótica moderna: la auto-localización. En muchos casos, es necesario para que un robot desempeñe su tarea que conozca en todo momento su posición. Este problema se resuelve empleando alguno de los diversos algoritmos de auto-localización existentes. En concreto, este ejercicio se orienta a emplear el método del filtro de partículas, asumiendo que el robot conoce de antemano el mapa del escenario en que se va a mover. Se usará un robot aspiradora *Roomba* que incorpora únicamente un sensor láser y un sensor de odometría, además de motores para poder efectuar movimientos en cualquier dirección.

El estudiante deberá programar un algoritmo de auto-localización que permita a la aspiradora estimar su posición en un mundo del cual sólo se proporciona un mapa binario en escala. El interfaz gráfico (GUI) de esta práctica incluye distintos *widgets* que facilitan

la depuración, con espacios reservados para una visualización adaptada de las lecturas del sensor láser y el propio mapa del entorno, donde se marcarán en todo momento la posición del robot en el mundo, las estimaciones que se van obteniendo y otros componentes elementales que van surgiendo como salida del algoritmo mencionado.

En este caso, el algoritmo funcionará en paralelo con la navegación, el robot puede teleoperarse o deambular autónomamente y el algoritmo de autolocalización funciona independientemente.

5.2. Infraestructura desarrollada

En este apartado se describirá el soporte que sostiene la práctica. Se comenzará describiendo el modelo de robot empleado, así como los sensores y actuadores que posee. Después, los entornos simulados y sus características, de vital importancia para la práctica, serán descritos.

5.2.1. Modelo de robot de interiores con sensor láser

El robot en el que se ha inspirado esta práctica es la aspiradora autónoma modelo *Roomba* de la serie 500, que fue comercializado por la empresa iRobot. Esta aspiradora robótica está equipada con sensores varios con los que medir características del entorno y actuadores que le permiten moverse adecuadamente por el escenario. Los aspiradores de esta serie poseen sensores infrarrojos, un sensor detector de suciedad, un sensor detector de desniveles, y cuentan además con un *bumper*. No todos serán necesarios para abordar la tarea de auto-localización. Para la práctica se ha creado el modelo *RoombaROS*, que no tiene detector de desniveles debido a que en el escenario elegido no hay escaleras o cambios de nivel entre habitaciones, y tampoco tiene detector de suciedad, puesto que no es necesario. El objetivo de la práctica es hacer énfasis en el algoritmo de navegación, de manera que de los sensores restantes sólo se usarán el de odometría, para detectar el número de veces que giran las ruedas, y el láser, para medir distancias a los obstáculos; ambos detallados más adelante.

El robot *Roomba* de la serie 500 posee una anchura de 340 milímetros, 92 milímetros de altura, y un peso de 3.6 kg. El modelo RoombaROS de JdeRobot mide aproximadamente

330 mm de ancho, 90 mm de altura, y un peso de 2.5 kg.

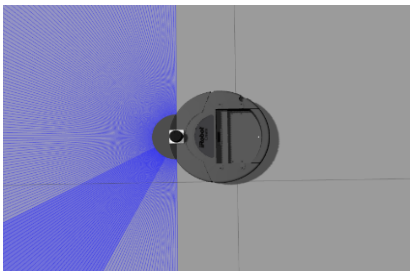

Figura 5.1: Modelo Roomba ROS

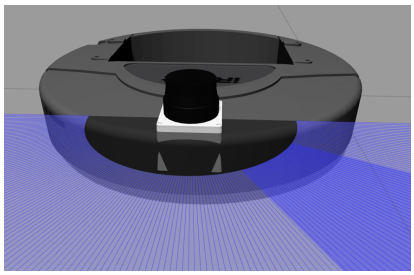

Figura 5.2: Modelo Hokuyo sobre Roomba

Partiendo del modelo de la Figura 5.1, se enriquecerá el mismo para que incluya todas las interfaces de sensores y actuadores anteriores que se comunicarán a través de *plugins* de ROS.

En cuanto al sensor láser, hemos utilizado un modelo ya creado llamado *hokuyo* (Figura 5.2) que ya se comunica a través de *ROS Messages*, el cual se ha incrustado en la parte frontal del robot.

Para implementar toda la funcionalidad anterior en la aspiradora, hemos usado los siguientes *plugins*, todos ellos compatibles con la versión *kinetic* de ROS:

- *libgazebo_ros_bumper.so* Para gestionar el sensor de contacto
- *libgazebo_ros_laser.so* Para recibir datos del sensor láser
- *libgazebo_ros_diff_drive.so* Para obtener odometría y controlar motores

Una vez creado el modelo, basta con incluirlo en el mundo o escenario deseado para poder usarlo.

5.2.1.1. Sensor láser

En la parte frontal del robot se ha modelado un sensor láser es el sensor más importante a estos efectos.

Está compuesto por un *array* de 180 medidas, que puede medir distancia alrededor de 180 grados en milímetros, con precisión de 1 grado. Su funcionamiento se basa en emitir rayos láser en todas estas orientaciones, que rebotan sobre los objetos existentes (si los

hay) de manera no especular, de tal manera que al recibir el rayo devuelto se calcula la distancia al objeto según el tiempo de vuelo.

El nodo académico se encargará de recibir los datos del sensor y adaptarlos en forma de un API sencillo de utilización, para que estén accesibles desde el algoritmo que el alumno debe programar. Se ha construido un analizador para la práctica que agrupa los datos en forma de *array* de 180 distancias, donde el índice es el ángulo de proyección del haz del láser.

5.2.1.2. Sensor Odométrico

En ésta práctica jugará un papel importante el saber cuánto se ha desplazado el robot en un intervalo. Ésta es precisamente la función de la odometría: el uso de sensores de movimiento para determinar el cambio de posición incremental del robot en relación con una posición conocida.

Si el robot conoce el diámetro de sus ruedas, simplemente le basta con contar el número de revoluciones de las mismas para determinar qué tan lejos ha viajado. Normalmente, el conteo de vueltas se realiza a través de *encoders*, que emiten un número fijo de pulsos por revolución, siendo éstos los registrados por el software. Así, el sentido de giro y el número de vueltas que da cada rueda nos ayudará a saber en qué dirección se ha movido el robot, y cuánta distancia ha recorrido.

5.2.2. Escenario de casa asimétrica

Dado que la aspiradora necesita un entorno acotado para determinar su localización (el láser tiene un alcance limitado), se ha creado un modelo para que la aspiradora navegue en él. Hemos partido del diseño *house_int2* que utiliza la práctica *Vacuum Cleaner* de JdeRobot-Academy como punto de partida. Este modelo se puede ver en la Figura 5.3:

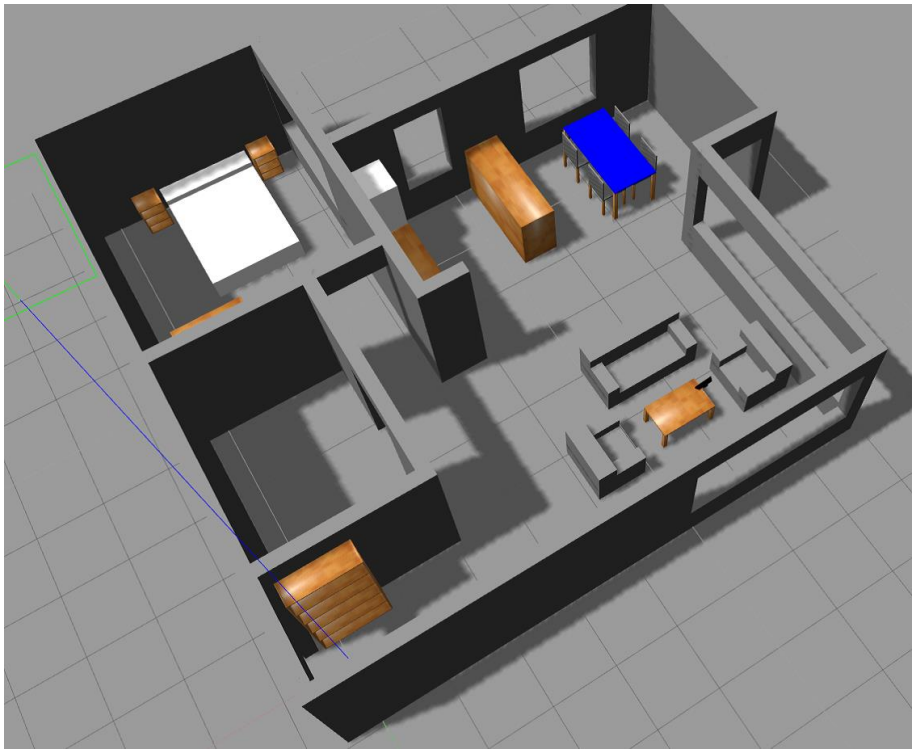

Figura 5.3: Modelo house_int2

Las razones que han motivado la elección de este modelo son:

- Entorno acotado por paredes, ideal para obtener lecturas del sensor láser representativas.
- Abundantes obstáculos, lo cual favorece la diferencia entre lecturas en cada punto de la casa.
- Malla de colisiones bien definida, importante cuando se quiere que el robot navegue por la casa sin la aparición de sucesos extraños, como objetos voladores al colisionar con ellos.

Aún con todo ello, este escenario emplea elementos de difícil modelado en una imagen 2D (necesaria para establecer el mapa del mundo), como puede ser la mesa y las sillas: cada pata debía tener el tamaño exacto a escala y estar en la posición adecuada para no obtener procesados erróneos. La zona bajo la cama también resultó un problema.

Dadas las múltiples ventajas que presenta, se ha creado un nuevo modelo, basado en el anterior, que sustituye todos los elementos difíciles de modelar por paredes, claramente

definidas y representadas, mucho más fáciles de modelar en un mapa. Al nuevo mundo que incluía este modelo y el modelo de aspiradora se le ha llamado *Asymmetric-Easy-To-Model.world* (Figura 5.4):

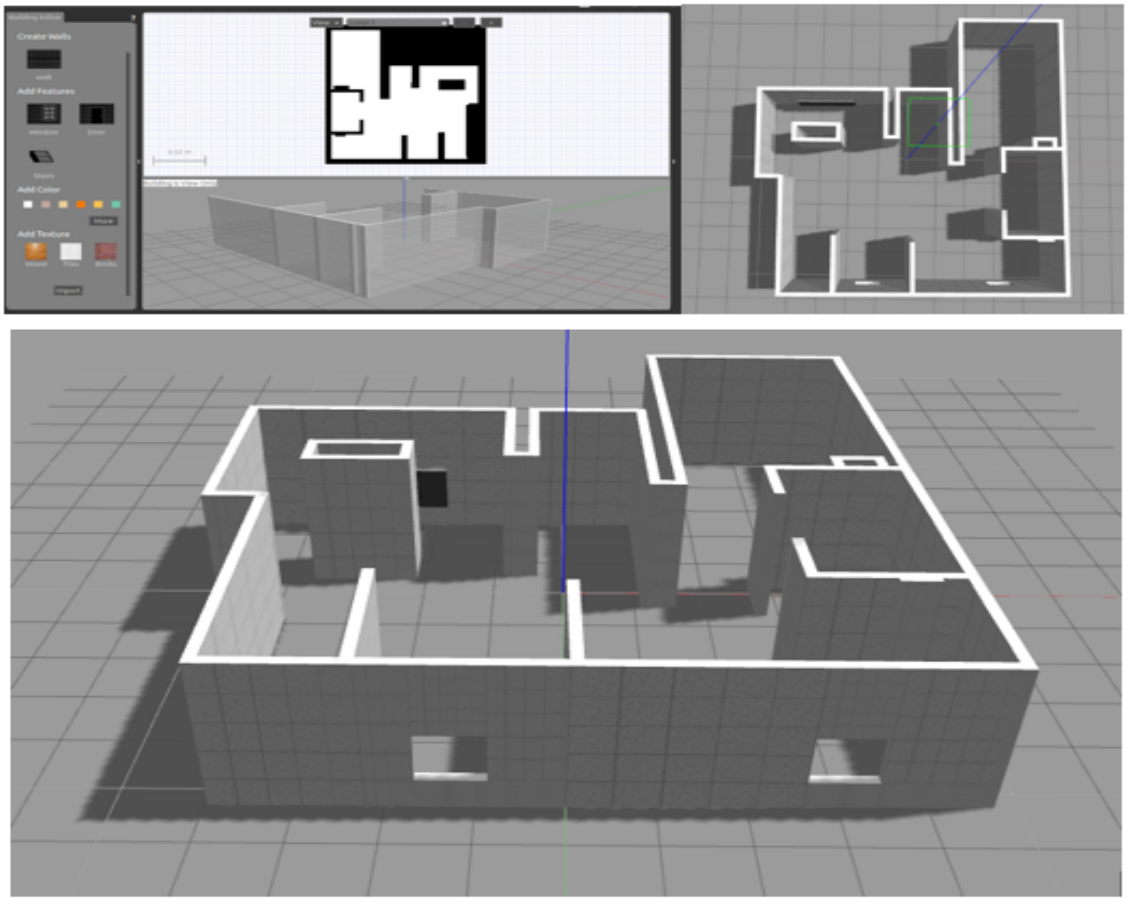

Figura 5.4: Modelo Asymmetric-Easy-To-Model

5.2.3. Modelo de nave simétrica

El modelo anterior es el caso más común en el que nos encontramos con el problema de la auto-localización, no obstante, también existen lugares donde las formas se repiten o las superficies límite son iguales o similares en distintas orientaciones. Es por eso que hemos creado el mundo de Gazebo *Choso.world*, el cual es una habitación cuadrada con sólo dos elementos, que ocupan muy poco espacio y por tanto pasan desapercibidos en la mayoría de casos. Este modelo se puede ver en la siguiente imagen (Figura 5.5):

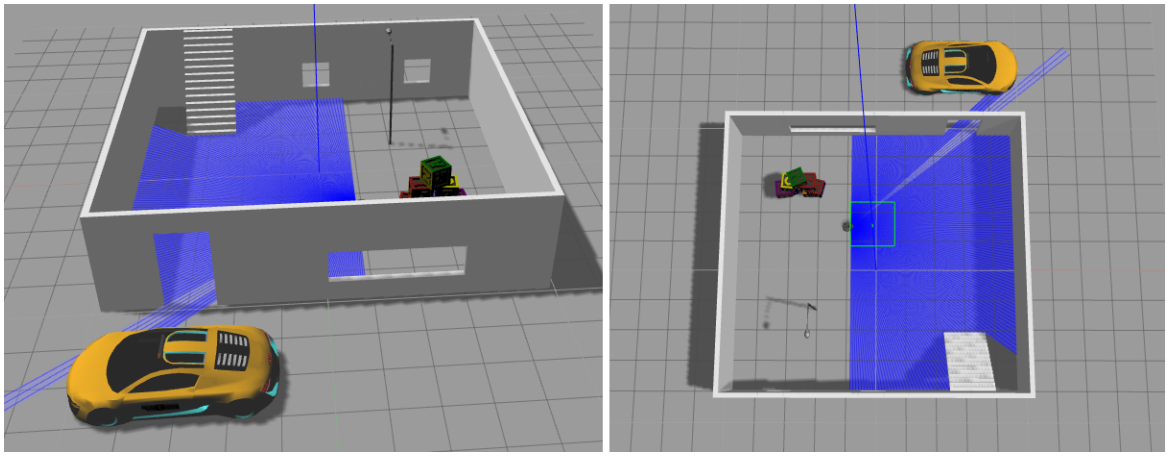

Figura 5.5: Modelo Choso

Introduciremos al robot en este modelo con fines de experimentación y verificación del comportamiento del algoritmo de autolocalización.

5.2.4. Ficheros de configuración

Incorporando los dos modelos involucrados (*Asymmetric-Easy-To-Model* y *Roomba-ROS*) y algunas fuentes de luz como el sol, luz ambiente y algunas sombras, se crea el mundo principal en el que abordar la práctica: *Asymmetric-Easy-To-Model.world* (Figura 5.6). Debido a la extensión del fichero de descripción del mundo, sólo incluimos a continuación una parte del mismo para observar su aspecto. En él se puede ver la combinación del modelo de robot creado y de los parámetros de física, aspecto, iluminación y posición que componen el mundo:

```
<?xml version="1.0" encoding="UTF-8"?>
<sdf version='1.6'>
  <world name='default'>

    <!-- Name it uniquely to not conflict with <include>'d model  -->
    <model name="roomba_ROS">
      <include>
        <uri>model://roombaROS</uri>
      </include>
    </model>

    <light name='sun' type='directional'>
      <cast_shadows>1</cast_shadows>
      <pose frame=''>0 0 10 0 -0 0</pose>
      <diffuse>0.8 0.8 0.8 1</diffuse>
      <specular>0.2 0.2 0.2 1</specular>
      <attenuation>
        <range>1000</range>
        <constant>0.9</constant>
        <linear>0.01</linear>
        <quadratic>0.001</quadratic>
      </attenuation>
      <direction>-0.5 0.1 -0.9</direction>
    </light>
    <model name='ground_plane'>
      <static>1</static>
      <link name='link'>
        <collision name='collision'>
          <geometry>
            <plane>
              <normal>0 0 1</normal>
              <size>100 100</size>
            </plane>
```

Figura 5.6: Código del mundo

5.3. Nodo Académico

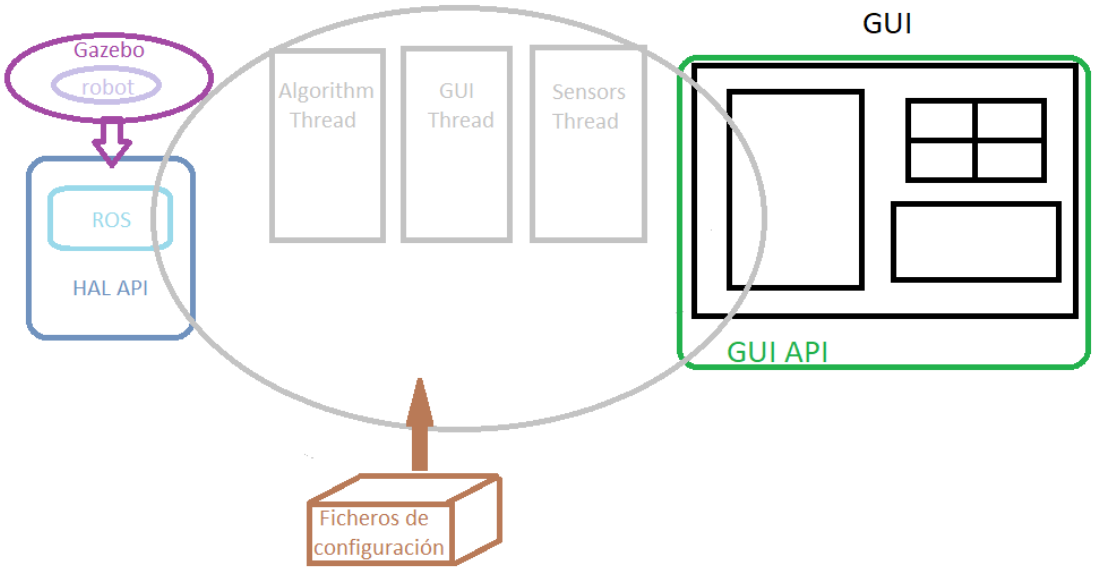

Figura 5.7: Diseño de Nodo académico de Laser Loc

El componente académico que se ha preparado para esta práctica (Figura 5.7) recoge toda la funcionalidad para ayudar al alumno a enfrentarse a ella y resolverla con éxito. Estas son las piezas que conforman el componente y que quedan resueltas a través de él:

a) Ofrece acceso a todas las interfaces que el robot posee, tanto sensores como actuadores, en forma de métodos simples. Oculta el *middleware* de comunicaciones.

b) Ofrece una interfaz gráfica al usuario que le ayuda a depurar su código.

c) Incluye código auxiliar, como analizadores o constructores de objetos, que no son parte del algoritmo a realizar, pero que sirven de ayuda para hacer la tarea más sencilla.

El componente pone la base que el estudiante culmina incluyendo su código en el método *execute* del fichero *MyAlgorithm.py*, siempre realizando las pruebas que considere oportuno para pulir su solución.

5.3.1. Arquitectura software

El componente se ha dividido en distintos hilos para favorecer la simultaneidad de ejecución de distintas tareas clave. En un principio, el componente sólo contaba con dos hilos básicos de ejecución, pero el diseño final elegido es con tres, de modo que se agiliza su funcionamiento:

→ Hilo de algoritmo de localización: se encarga del refresco de la ejecución del algoritmo, ya que este se ejecuta de modo iterativo o cíclico. El tiempo de refresco es importante, dado que el algoritmo necesario para solventar la práctica es bastante sensible a pequeños cambios en las lecturas, que deben desencadenar correcciones en la ejecución. Por ello, el tiempo de refresco es de 10 ms.

→ Hilo de la interfaz gráfica de usuario (GUI): es el encargado de actualizar la interfaz gráfica y los *widgets* incorporados en ella. El intervalo de actualización de la interfaz debe ser pequeño, de tal manera que los cambios en los sensores o en el mapa puedan ser representados en tiempo real. Por ello, se ha fijado el tiempo de refresco a 20 ms.

→ Hilo de sensores: el tercer hilo se ha añadido para actualizar los datos de los sensores y los actuadores a través de las interfaces que correspondan (ROS). El tiempo de refresco de este hilo es muy importante, dado que un intervalo largo radicaría en lecturas desactualizadas de los sensores. Por ello, dicho tiempo también se ha fijado en 20 ms.

Con todo ello, el nodo dispone la interfaz gráfica de usuario con algunos elementos de depuración (accesibles a través de un API descrito en 5.3.2 y 5.3.3) para que el usuario se centre en el algoritmo a escribir. La manera de integrar dicho algoritmo en el nodo para que llegue al robot también es tarea del componente académico, por lo cual se reserva un espacio concreto para que el alumno inserte la lógica. Todo ello está debidamente indicado en el fichero de instrucciones incluido (*README.md*).

5.3.2. Interfaz de sensores y actuadores y código auxiliar

El nodo ofrece un API de sensores y actuadores al programador, además de un API específico auxiliar para la práctica en el que se encontrarán otros métodos e incluso

variables disponibles para facilitar la tarea. En cuanto al API del robot, tenemos:

- *self.sensors.motors.sendVelocities(vel)*: Para enviar comandos de velocidad a través de:
  - ◇ *vel = CMDVel()*: constructor de comandos de velocidad
  - ◇ *vel.vx = valor*: velocidad lineal
  - ◇ *vel.az = valor*: velocidad angular
- *self.sensors.laserdata* o *self.gui.getLaserData()*: dos posibilidades equivalentes para obtener los datos láser.
- *self.parse_laser_data(laser)*: Para transformar los datos láser en una estructura más fácilmente manejable.
- *self.pose3d.getPose3d().x*, *self.pose3d.getPose3d().y*, *self.pose3d.getPose3d().yaw*: para obtener los datos de posición y orientación del robot.

5.3.2.1. Partículas

El elemento principal en el que se basa el algoritmo que se debe emplear en la solución son las partículas. Estas partículas no son más que puntos en el espacio 2D en los que el robot se proyecta a sí mismo, es decir, posiciones y orientaciones que el robot puede ocupar y tomar en el escenario. Por ello, cada partícula consta de:

→ Un vector de coordenadas (x, y, z) que representa su posición en el espacio. La coordenada z se fija siempre a 0, dado que el escenario no contiene desniveles.

→ Un atributo de orientación, que representa su rotación con respecto al mapa.

→ Un valor de probabilidad, que almacena el valor numérico obtenido al comparar el láser de la partícula con la lectura real.

A fin de organizar el código, se ofrece al programador la clase *Particle*¸ la cual actúa como constructor de partículas, asociando a cada una sus correspondientes características accesibles como atributos de un objeto Python:

```
class Particle:
    def __init__(self, x, y, yaw, prob, mapAngle):
        self.x = x
        self.y=y
        self.yaw=mapAngle+yaw
        self.prob=prob
```

El nodo académico incluye este código auxiliar en el cual el estudiante puede apoyarse para desarrollar su solución, que utilizará a través de:

- ||**Clase Particle**||: *Particle(x, y, yaw, prob, self.map.robotAngle)*: Constructor de partículas.

5.3.3. Interfaz gráfica y uso desde código

La interfaz gráfica de usuario (GUI) se emplea para representar información que ayuda a resolver el algoritmo planteado, lo cual resultará vital en el camino hacia la resolución de la misma. Se ha construido a través de PyQt5, y se encarga de mostrar la salida de la ejecución del algoritmo en cada instante, ideal para depurar y observar su evolución.

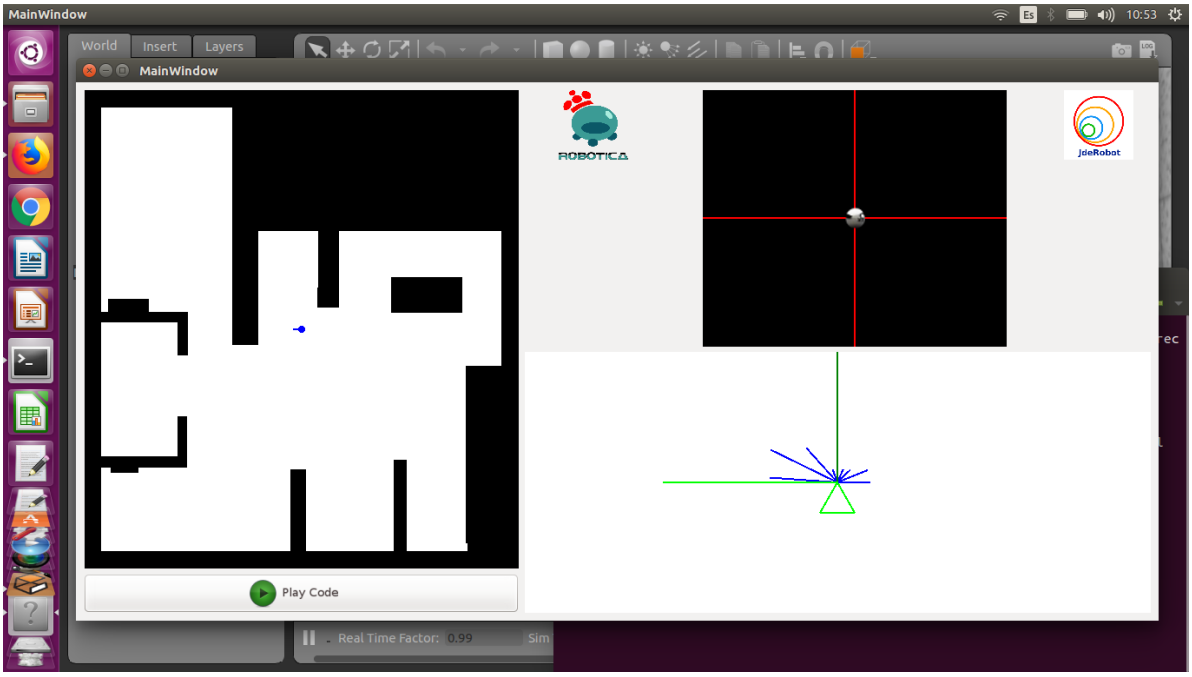

Figura 5.8: Interfaz Gráfica Laser Loc

Esta GUI (Figura 5.8) está formada por 3 *widgets* para el control y visionado del comportamiento del robot. El primero de ellos, situado a la izquierda, es quizás el más importante, pues muestra al programador el mapa binario que el robot utiliza para realizar cálculos pertinentes para llevar a cabo su auto-localización. También muestra información gráfica con la que éste no cuenta: las sucesivas generaciones de partículas que surgen del algoritmo evolutivo. En el mapa se ven reflejados los obstáculos (zonas negras), las zonas libres (espacio en blanco), la posición y orientación del robot en el espacio dado (a través de un punto azul), la evolución de las partículas e incluso las trayectorias.

El segundo de los elementos del interfaz, situado en la parte superior derecha, es el clásico teleoperador, para controlar la posición del robot y su rotación a través de órdenes de velocidad lineal y angular. Esto facilita el proceso de depuración y sirve de apoyo en los primeros pasos hacia la solución.

Por último, el espacio inferior derecho se ha reservado para la representación de las lecturas que ofrece el láser del robot en cada momento, y también para dibujar de manera aproximada el cálculo de láser teórico que el robot hace de cada partícula.

En la parte inferior se ha incluido un botón para ejecutar el algoritmo y pararlo cuando sea necesario, deteniendo todas las representaciones y el movimiento del robot.

Para hacer uso de la funcionalidad de depuración que ofrece el interfaz gráfico, se ha dispuesto un interfaz de programación con los siguientes métodos:

- *self.setParticles([p1,p2,p3,...])*: Representa una lista de partículas en el GUI.
- *self.setEstimation(particle)*: Para mostrar una estimación en el interfaz.
- *self.paintTheoricalLaser(theoricalLaser)*: para representar un láser en la interfaz.
- *img = self.map.pixmap.toImage()*: para obtener la imagen (mapa) del interfaz.
- **(variable)** *self.particleClicked*: almacena la última partícula sobra la que se ha hecho click en el GUI.
- *self.map.map2pixel((x,y))*: obtiene el pixel correspondiente a las coordenadas (x,y).
- *self.map.pixel2map((px,py))*: obtiene las coordenadas reales asociadas al pixel [px,py] del mapa.

5.3.3.1. Gráfica de los láseres Real y Teórico

Uno de los elementos de visualización que incluye la práctica es la posibilidad de ver de forma gráfica los datos que ofrece el sensor láser, así como una representación de los haces láser de cada partícula calculados a partir de la lectura que ofrecería el robot en el caso de que su posición y orientación coincidiesen con los de esta. Así, lo que se representa es lo siguiente:

1. **- Láser Real**

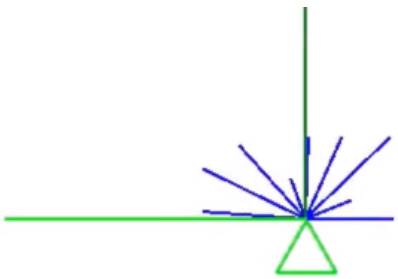

Figura 5.9: Laser Real

Dado que las lecturas utilizadas para esta práctica están formadas por 9 haces en total (uno cada 20º con respecto a la normal de la orientación del robot), lo que se representa es un conjunto de segmentos cuya longitud se corresponde con la distancia (a escala 50:1) al primer obstáculo en la dirección marcada por el ángulo del haz (Figura 5.9). Las medidas del láser constan de 180 pares de valores, y su reducción a los 9 utilizados se describirá en 5.4.2. Los datos del sensor se analizan convenientemente antes de ser representados, y la representación es en todo caso relativa al propio robot.

2. **- Láser Teórico**

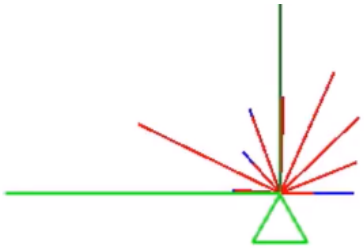

Figura 5.10: Laser Teórico

Como veremos más adelante, para cada partícula involucrada en el algoritmo se deberá calcular un "láser teórico" (Figura 5.10), es decir, la lectura que se obtendría en el sensor en el supuesto de que el robot ocupase la posición de dicha partícula y compartiese su orientación. En la representación se muestra la superposición de ambos datos láser: el real en azul y el teórico en rojo, de manera que en todo momento se puede comprobar si una partícula ofrece una lectura similar a la del robot o no.

Dado que el algoritmo se compone de muchas partículas, para activar la representación del láser teórico de una partícula determinada bastará con hacer click sobre ella en el *widget* que contiene el mapa de referencia.

5.3.3.2. Mapa de Referencia

El elemento gráfico vital es el mapa que robot utiliza para hacer cálculos sobre su entorno. Este mapa es una imagen con sólo dos valores posibles para cada pixel: blanco y

negro. Esto facilita las tareas de procesado. El robot se basa en dicho mapa para realizar el cálculo de los datos teóricos láser, llevando a cabo un trazado de rayos sobre la imagen.

Este mapa emplea una codificación de color para expresar visualmente la probabilidad de cada partícula, como el que se muestra en la Figura 5.11. Así, cada partícula se representará con un color dentro de este rango, en función de sus posibilidades de ser la posición real en la que se encuentra el robot.

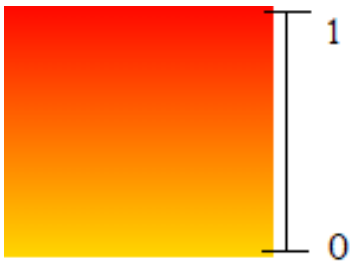

Figura 5.11: Codificación de color

Además, en el mapa se mostrarán:

- La posición real del robot, también representada en forma de partícula, utilizará un color azul oscuro en todo momento
- Las estimaciones que el robot hará de su posición se representarán en azul claro.

Por último, este lienzo también superpondrá al mapa la trayectoria real que el robot sigue en todo momento representada en negro, y la trayectoria estimada por el robot para localización en movimiento, representada también en azul claro, como se ve en la Figura 5.12:

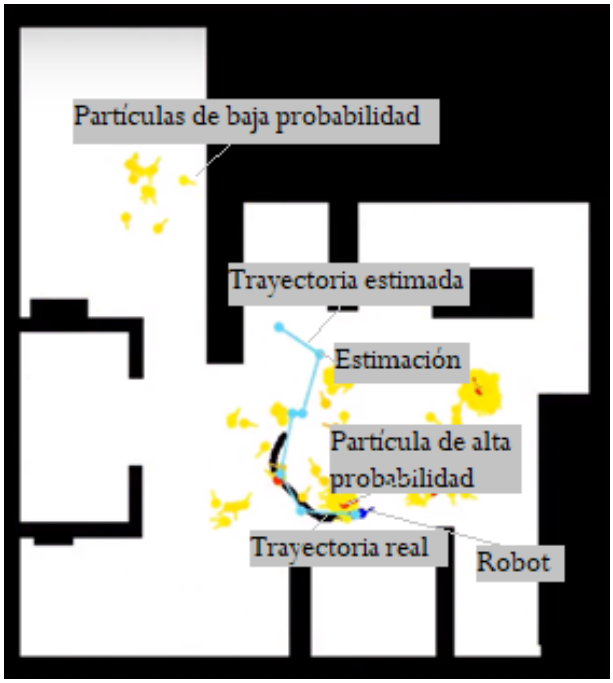

Figura 5.12: Mapa de referencia

5.3.3.3. Trayectorias

En cuanto a las trayectorias que se representan en el gráfico, cabe decir que la ruta real está formada por las últimas 100 posiciones diferentes que ha ocupado el robot para su representación, de manera que se modifica con las sucesivas iteraciones. Sirve como apoyo para pulir el comportamiento del algoritmo, ya que el programador podrá compararla en todo momento con la trayectoria que su propio algoritmo de localización va estimando. Su representación es automática y se encarga la interfaz del nodo académico.

Por otro lado, la trayectoria estimada es obtenida a través de la unión de las distintas estimaciones de posición que el algoritmo del robot hace a lo largo del tiempo. Se ha dotado al componente con un API para el establecimiento de estimaciones a visualizar en la gráfica, con lo cual se puede hacer un seguimiento del comportamiento del algoritmo a largo plazo:

self.setEstimation([x,y])

5.3.4. Comunicaciones con sensores y actuadores

La práctica debe ser auto-contenida, de manera que el nodo académico debe icluir en su fichero principal (*laser_loc.py*) todo lo necesario para establecer y mantener la comunicación con el robot, inicializando los distintos nodos de comunicación con sensores y actuadores y suscribiendose o publicando en un *topic* adecuado para cada interfaz del robot. Además, debe realizar la conversión entre los mensajes recibidos y los datos que el nodo académico puede manejar.

Por tanto, dicho fichero debe contener:

1. El código necesario para conectar con cada nodo del robot, para lo cual se usan sentencias tipo:

```
node = rospy.init_node(ymlNode["NodeName"], anonymous=True)
```

2. La suscripción de cada nodo inicializado a su correspondiente *topic* de ROS, es decir, a su bus de intercambio de mensajes en el cual el driver encargado del control del nodo (ya sea sensor o actuador) publica o recibe mensajes con cierta forma:

```
self.sub = rospy.Subscriber(self.topic, LaserScan, self.__callback)
```

o

```
self.sub = rospy.Subscriber(self.topic, Odometry, self.__callback)
```

3. La transformación de los mensajes que se intercambian por la red a la estructura utilizada por el nodo de la práctica (Figura 5.13):

```
def odometry2Pose3D(odom):
    '''
    Translates from ROS Odometry to JderobotTypes Pose3d.

    @param odom: ROS Odometry to translate

    @type odom: Odometry

    @return a Pose3d translated from odom

    '''
    pose = Pose3d()
    ori = odom.pose.pose.orientation

    pose.x = odom.pose.pose.position.x
    pose.y = odom.pose.pose.position.y
    pose.z = odom.pose.pose.position.z
    #pose.h = odom.pose.pose.position.h
    pose.yaw = quat2Yaw(ori.w, ori.x, ori.y, ori.z)
    pose.pitch = quat2Pitch(ori.w, ori.x, ori.y, ori.z)
    pose.roll = quat2Roll(ori.w, ori.x, ori.y, ori.z)
    pose.q = [ori.w, ori.x, ori.y, ori.z]
    pose.timeStamp = odom.header.stamp.secs + (odom.header.stamp.nsecs *1e-9)

    return pose
```

Figura 5.13: Trnasformación de datos de odometría

Para todo ello, el fichero principal amplía su funcionalidad con una serie de clases que se encargan de la construcción de objetos que tengan en cuenta todas las especificaciones anteriores.

5.3.5. Fichero de configuración

El único elemento externo que el nodo académico necesita es un fichero de configuración que recoja los *endpoints* que las interfaces de los sensores y actuadores descritos en 5.3.2 utilizan para publicar la información o recibirla del nodo. Este archivo tiene extensión *.yml*, lo cual implica que debe cumplir las especificaciones del formato de serialización de datos YAML. Para esta práctica en concreto, añadimos además la configuración necesaria para establecer el mapa binario de referencia del robot.

La asociación entre cada nodo y su *topic* se ha de facilitar también en el fichero a la hora de ejecutar. Tiene el siguiente aspecto (Figura 5.14):

```yaml
LaserLoc:

  Motors:
    Server: 2 # 0 -> Deactivate, 1 -> Ice , 2 -> ROS
    Proxy: "Motors:tcp -h localhost -p 9001"
    Topic: "/cmd_vel"
    Name: LaserLocMotors

  Pose3D:
    Server: 2 # 0 -> Deactivate, 1 -> Ice , 2 -> ROS
    Proxy: "Pose3D:tcp -h localhost -p 9001"
    Topic: "/odom"
    Name: LaserLocPose3d

  Laser:
    Server: 2 # 0 -> Deactivate, 1 -> Ice , 2 -> ROS
    Proxy: "Laser:tcp -h localhost -p 9001"
    Topic: "/laser/scan"
    Name: LaserLocLaser

  Bumper:
    Server: 2 # Deactivate, Ice , ROS
    Proxy: Bumper:tcp -h 0.0.0.0 -p 9000
    Topic: "/mobile_base/events/bumper"
    Name: LaserLocBumper

  maxV: 1
  maxW: 20
  NodeName: LaserLoc

  Map:
    Server: 0 # 0 -> Deactivate, 1 -> Ice , 2 -> ROS
    Img: resources/images/map_autoloc.png #max (630x680)
    WorldHeight: 10
    WorldWidth: 10
    OriginX: -1
    OriginY: 1.5
    Angle: 180
    Scale: 50.0

```

Figura 5.14: Fichero de configuración YAML con interfaces de ROS

Vemos que la información del mapa cuenta con atributos como el tamaño, la escala o el *path* que conduce a la imagen binaria. Por último, se establecen las velocidades máximas de tracción y rotación en base a las necesidades de la práctica.

5.4. Solución de referencia

Para esta práctica no es adecuado incluir también un algoritmo de pilotaje, pues se dispone de un teleoperador y el foco está en la autolocalización que se ejecuta de modo independiente mientras el robot se mueve (teleoperado o autónomo). Construiremos una solución que se establecerá como solución de referencia en el fichero *MyAlgorithm.py* que tiene naturaleza iterativa, de manera que en cada iteración se obtiene datos de los sensores, se procesan y se evoluciona el algoritmo de localización. El código a escribir irá de nuevo a parar al método *execute*, el cual ejecuta la lógica de manera cíclica en el *thread* de algoritmo y computación.

5.4.1. Autolocalización basada en filtros de partículas

El filtro de partículas aplicado al problema de la autolocalización de robots consiste en mantener un conjunto muestral de posibles posiciones y calcular la probabilidad de cada una. Mediante remuestreo estadístico, estas posiciones acaban convergiendo hacia la posición correcta del robot. El número reducido de muestras (comparado con el de la localización probabilística basada en rejillas) hace que se minimicen los costes computacionales y que el algoritmo sea escalable a grandes entornos. Se trata de un filtro bayesiano recursivo en el que el conjunto de muestras

$$\{x_i, P(x_i)\}, i = 1..n, \tag{5.1}$$

estaría formado por coordenadas de posición y orientación

$$x_i = (x, y, \theta), \tag{5.2}$$

inicialmente escogidas al azar sobre el escenario, que emplea su probabilidad P(xi) para ir seleccionando aquellas posiciones más probables. Cada muestra se denomina "partícula", y el objetivo es conseguir que la nube de partículas converja en la posición real del robot. En cada iteración del algoritmo se evoluciona a una nueva población a partir de la anterior, la

cual está formada por "hijos" de partículas que tenían alta probabilidad en dicha población anterior. El algoritmo se realiza en tres pasos que se ejecutan siempre y de forma iterativa:

- **Modelo de movimiento**: se desplazan las muestras $(\Delta r, \Delta\theta)$, equivalente al desplazamiento efectuado por el robot desde la última vez que se incorporó el movimiento del mismo. Esto permite incorporar información de desplazamiento incremental si se dispone de ella.
- **Modelo de observación**: se calculan las nuevas probabilidades de todas las muestras a partir de la lectura del sensor láser. Se ha de ajustar una función de salud que se encargue de asociar probabilidades altas a aquellas zonas compatibles con la lectura del láser real y bajar la de las zonas incompatibles.
- **Remuestreo**: se genera una nueva población a partir de la anterior siguiendo el algoritmo de la ruleta (Figura 5.15), que consiste en girar la ruleta tantas veces como hijos queremos generar. Cada partícula tiene un sector proporcional a su probabilidad en la ruleta, de manera que si el valor está incluido en el sector de una partícula concreta, esta genera un hijo "idéntico" en la siguiente población.

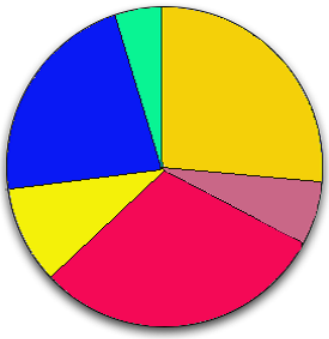

Figura 5.15: Algoritmo de la Ruleta

Debajo de este modelo existen muchos parámetros relevantes para la implementación.

Figura 5.16: Ejemplo de modelo de observación

La Figura 5.16 muestra un ejemplo de este modelo de observación, en el cual las partículas más lejanas a la posición real del robot (azul) y en distinta orientación toman valores más bajos de probabilidad representados con un color más claro (amarillo), mientras que las cercanas a la posición real toman un valor más alto, representado en un color intenso (rojo). El cálculo del "láser teórico" es lo que permite asignar a cada una su probabilidad. Para obtenerlo, se trazan rayos sobre el mapa, tirando una línea recta en el espacio bidimensional euclidiano desde la posición de cada partícula en la imagen hasta el punto donde se encuentre el máximo de distancia que puede abarcar el láser en las 9 direcciones que componen la lectura láser. En esta recta se va comprobando cada cierta distancia (paso λ) si el color del píxel es blanco (espacio libre) o negro (obstáculo), en cuyo caso se obtiene el punto en el que rebotaría el haz láser. Con ello, y conociendo la escala del mapa dado, obtenemos una distancia para cada haz que representa la que se obtendría en el mundo si el robot ocupase esa posición. Así,

$$X_{obstaculo} = (PX_{partícula} + \lambda * cos(\alpha_{haz})) * escala \tag{5.3}$$

$$Y_{obstaculo} = (PY_{partícula} + \lambda * sin(\alpha_{haz})) * escala \tag{5.4}$$

$$Distancia = \sqrt{(X_{obstaculo} - X_{partícula})^2 + (Y_{obstaculo} - Y_{partícula})^2} \tag{5.5}$$

Con esta distancia de cada haz, basta con compararla con la distancia observada por el sensor homólogo, con lo cual se obtendrá un error (error = LaserReal- LaserTeórico).

Sólo es necesario acumular este error en cada orientación para asignar posteriormente una probabilidad. Cuanto menor error, las lecturas son más parecidas y se asignará la mayor probabilidad.

Por otro lado, el modelo de movimiento es de gran importancia para descartar información inservible y revalidar información adecuada. Se incorpora el movimiento incremental del robot a la generación de partículas en coordenadas polares $(\Delta r, \Delta\theta)$, para lo cual:

$$\Delta x = x_t - x_{t-1} \tag{5.6}$$

$$\Delta y = y_t - y_{t-1} \tag{5.7}$$

$$\Delta\theta = \theta_t - \theta_{t-1} \tag{5.8}$$

$$\Delta r = \sqrt{(\Delta x)^2 + (\Delta y)^2} \tag{5.9}$$

Bastará con sumar el incremento a la posición de cada partícula de modo consecuente con la orientación de la partícula (Figura 5.17), o más en concreto con el ángulo que forma ésta con respecto a la orientación del robot.

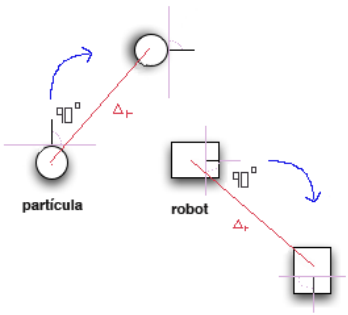

Figura 5.17: Orientación Relativa

Hemos decidido aplicar geometría y trigonometría para materializarlo. En primer lugar, se construye una recta que pase por las coordenadas de la partícula (x_0, y_0) a la cual se le va a sumar el movimiento. Esta recta se regirá por el vector de dirección creado a partir de la orientación de la partícula resultante al añadir el incremento angular:

$$a = cos(\theta_0 + \Delta\theta) \tag{5.10}$$

$$b = sin(\theta_0 + \Delta\theta) \tag{5.11}$$

$$y = y_0 + \frac{b}{a} * (x - x_0) \tag{5.12}$$

Conociendo la distancia recorrida (Δr), calculamos la coordenada x del nuevo punto al cual ha de desplazarse la partícula (y luego la coordenada y con la ecuación de la recta):

$$\Delta r = \sqrt{(x - x_0)^2 + ((y_0 + \frac{b}{a} * (x - x_0)) - y_0)^2} \Rightarrow x\,del\,nuevo\,punto \tag{5.13}$$

Una vez hecho, ante un movimiento del robot se tiene lo que se muestra en la Figura 5.18:

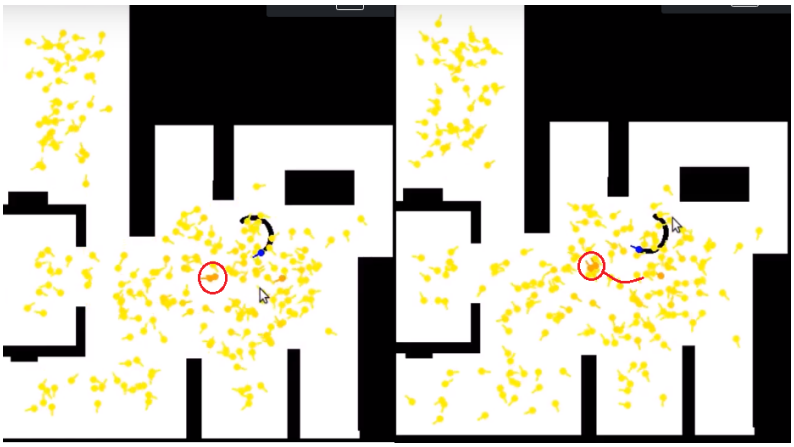

Figura 5.18: Incorporación relativa de odometría

Para hacer evolucionar el conjunto hacia nuevas generaciones de partículas de manera que converjan a la posición del robot se emplea el algoritmo de la ruleta. Para ello se acumula la probabilidad de toda la generación $Pac(x_i) = P(x_i) + Pac(x_{i-1})$, y luego se asocia a cada partícula un sector cuyo ancho es proporcional a la probabilidad individual de la partícula. El girar la ruleta se materializará a través de la obtención de un número aleatorio, el cual formará parte de un único sector dentro de la ruleta, saliendo a relucir la partícula que generará descendencia (Figura 5.19). Las partículas que tienen alta probabilidad producen "picos" en la probabilidad acumulada, y por lo tanto tendrán más opciones de ser elegidas a la hora de muestrear. Así, una misma partícula puede tener muchos o hijos, o ninguno.

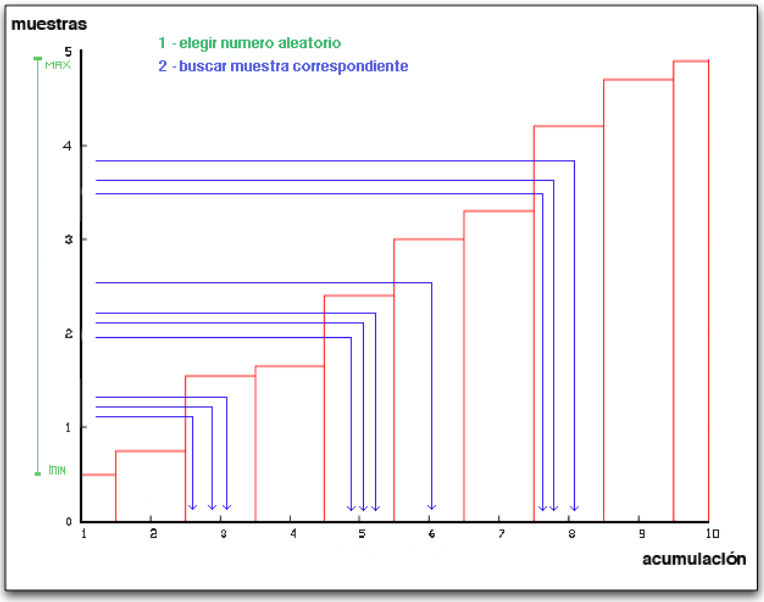

Figura 5.19: Algoritmo de la Ruleta y probabilidad

5.4.2. Solución desarrollada

En la solución de referencia desarrollada se han utilizado poblaciones de 650 partículas distribuidas por la escena, para cada una de las cuales se calculan 9 haces láser teóricos para asociarles una probabilidad, procesando en cada una la imagen del mapa en función de su posición. Estos valores de compromiso permiten su uso en aplicaciones en tiempo real. El utilizar 9 haces por láser en lugar de los 180 disponibles se debe a que este número de haces es suficiente para obtener lecturas representativas del entorno en cualquier punto, de manera que todos los otros rayos sólo aportarán en casos muy extremos, pero generalmente ralentizarán en gran medida el comportamiento del algoritmo y reducirán su eficiencia.

También el hecho de segmentar la funcionalidad de la práctica en tres hilos de ejecución se debe a esto, ya que existían distintas tareas que se podían realizar de manera concurrente y compatible, ya que ninguna de ellas se cruza con las demás (toma de datos de sensores, interfaz gráfico y algoritmo). Por otro lado, también se han precomputado todos los datos posibles para aumentar la eficiencia al máximo en cada iteración.

5.4.2.1. Inicialización

Para comenzar el algoritmo, si no se ha establecido generación de partículas se utiliza el método *sendParticles()* para establecerlas de la manera aleatoria inicial. Este método se encarga de todo lo necesario para obtener una primera generación, incluyendo el trazado de rayos de cada partícula para la obtención de su láser teórico con *doRayTracing()*. Hay involucradas otras funciones que comprueban si el color de un píxel es blanco o negro (*isWhitePixel()*), que realizan el análisis de la información láser (*parse_laser_data()*) y otras como el método *uniform* de la biblioteca random para obtener las posiciones aleatorias entre otras.

5.4.2.2. Modelo de observación

Usaremos un modelo de observación para asignar una probabilidad en función del resultado del trazado de rayos, que genera una lectura láser teórica siguiendo el código de la Figura 5.20:

```
def doRayTracing(self, particle):
    teoricalLaser = []
    for i in range (-90,90,22): # 8 laser beams
        angle = particle.yaw+math.radians(i)
        stepx = 0.1*math.cos(angle)
        stepy = 0.1*math.sin(angle)
        x = particle.x-0.15*math.cos(particle.yaw)
        y = particle.y-0.15*math.sin(particle.yaw)

        while self.isWhitePixel(x,y):
            x = x-stepx
            y = y-stepy

        dist = math.sqrt(math.pow((x-particle.x),2)+math.pow(y-particle.y,2))
        teoricalLaser.append((dist, math.radians(i)))

    return teoricalLaser
```

Figura 5.20: Código del Trazado de Rayos

Para asignar una probabilidad a cada partícula se invoca el método *calculateProb()*. El valor de probabilidad se asignará en función de los parecidos existentes entre cada par de haces homólogos de los laseres real y teórico, en otras palabras, se realizará una

comparación haz por haz de ambos láseres. Se evalua la diferencia de distancia existente entre el haz real y el teórico en cada orientación, generando un error acumulado de distancia. Mediante este error mediremos el parecido de ambos láseres y asignaremos una probabilidad. Para ello se tiene la función que conforma el propio modelo de observación (Figura 5.21), que se ha ajustado hasta obtener resultados adecuados para esta aplicación.

```
for i in range(0,len(realLaser),22):
    realDist = realLaser[i][0]
    teoricalDist = laserP[i/22][0]
    error += math.sqrt(math.pow(realDist-teoricalDist,2))

if error <= 4.5:
    prob = 1.0-(error-1.0)*0.028
elif error > 4.5 and error <= 10.0:
    prob = 0.9-(error-4.5)*0.146
else:
    prob = math.exp(-0.23*error)
```

Figura 5.21: Código del Modelo de Observación

5.4.2.3. Modelo de movimiento

El siguiente paso es la incorporación de la odometría a través del modelo de movimiento, donde se actualizan todas las partículas y se gestiona si el movimiento incorporado invalida alguna partícula (tras el movimiento, algunas partículas pueden quedar fuera de los límites del espacio libre), en cuyo caso se genera una nueva partícula aleatoria. Esta incorporación es relativa a la posición y orientación de cada partícula. En este punto tendremos algo similar a la Figura 5.22:

Figura 5.22: Primera generación de partículas

5.4.2.4. Generación de la siguiente población

La población de partículas se va modificando iterativamente, y el conjunto va evolucionando si procede (con *calculateNewGeneration()*). En todas las iteraciones del algortimo no se realiza el cálculo de nuevas generaciones, ya que esto consumiría toda la capacidad del procesador. En base a distintas pruebas realizadas, hemos creado un reloj de generación, que evolucionará generaciones cada 300ms.

Se evoluciona de manera más inteligente si se aplican además algunas modificaciones, como el elitismo o el ruido térmico. Una vez seleccionado el progenitor, hemos aplicado técnicas elitistas que persiguen conservar características de unas generaciones a otras preservando aquellas partículas de mayor probabilidad, distinguiendo y conservando la "élite" y eliminado las partículas que no aportan información. Así, si la partícula seleccionada para generar descendencia tiene una probabilidad muy superior a un umbral prestablecido, se copia en la siguiente generación, si su valor es ligeramente superior al umbral, se aplica un ruido térmico (gaussiano) de posición y orientación centrado en el progenitor para generar el hijo y si no supera el umbral se lanza una nueva partícula aleatoria, como al inicio del algoritmo. Esto permite también explorar los alrededores de las partículas que acumulan mayor probabilidad, pudiendo así encontrar partículas de

mayor calidad en cada generación. Esto también aplica en el cálculo de la probabilidad acumulada, ya que si ésta no supera un umbral, se considera que no aporta información y se descarta la generación entera, remuestreando con una nueva generación aleatoria.

Si se trata de una *iteración de generación*, los pasos son:

- Comprobar si la generación converge. Para ello seleccionamos la partícula de mayor probabilidad de la generación y calculamos la distancia euclidiana entre esta y cada una de las otras partículas. Si todas ellas se encuentran comprendidas en una circunferencia de radio 2m, se considera que la generación converge, y es hora de hacer una estimación con la partícula de mayor probabilidad.
- En caso de no convergencia, y teniendo en cuenta las técnicas elitistas, en cada nueva generación se comprueba si la partícula más probable supera el 99 % de coincidencia, en cuyo caso también se crea una nueva estimación concreta de posición.
- En el resto de casos se calcula una nueva generación a partir de la anterior, calculando en primer lugar la probabilidad acumulada como la suma de todas las probabilidades, luego ejecutando el algoritmo de la ruleta una vez por cada nueva partícula que se quiera generar, en nuestro caso 650 veces, y con la partícula seleccionada se haga el filtro de partícula correspondiente (a través de *particlesFilter*). Se ha diseñado este filtro para que aplique las técnicas de selección, el ruido gaussiano y el remuestreo cuando proceda, en función de la salud de la partícula.
- En todo caso, tras una estimación se crea una nueva generación aleatoria.
- Habrá iteraciones de generación en las que no se haga ninguna estimación.
- Hay condiciones de parada, como un número máximo de iteraciones de generación sin estimación, para descartar datos que no llevan a una estimación correcta.

5.5. Experimentación

La funcionalidad descrita en los apartados anteriores ha sido ajustada a través de experimentos realizados para poner a prueba cada una de las partes. También se han hecho

pruebas globales que validan experimentalmente a todo el conjunto: la infraestructura de la práctica, el nodo académico y la solución desarrollada.

5.5.1. Ejecución típica

Se ha preparado un documento de texto plano (*README.md*) que sirve de guía para que alumno esté guiado durante la ejecución y las pruebas, que incluye también información del API de utilización, e incluso una referencia a un vídeo demostrativo.

La forma de ejecutar la práctica basada en interfaces de ROS es la siguiente:

1. Empleamos un primer terminal para ejecutar el componente que lanza un mundo Gazebo basado en componentes de ROS:

```
$ roslaunch laser_loc.launch
```

2. Utilizamos un segundo terminal para iniciar el componente académico:

```
$ python2 laser_loc.py laser_loc_conf.yml
```

Pulsando sobre el botón "Play Code" se podrá ver el resultado de la ejecución del algoritmo. Ejemplos de la ejecución pueden encontrarse en [1] y [2].

5.5.1.1. Localización estática

A continuación, una serie de capturas de una ejecución representativa (Figura 5.23),

[1] https://www.youtube.com/watch?v=FmUN_tzM9MM
[2] https://www.youtube.com/watch?v=4OFljaeag_I

Figura 5.23: Evolución del algoritmo de localización en una ejecucion típica

En este ejemplo, el robot (representado por el punto azul oscuro) permaneció estático. Se puede ver en la primera captura cómo la generación inicial es aleatoria, dentro de los límites de la habitación. Las siguientes 4 capturas muestran cómo la nube de partículas va convergiendo hacia la posición real del robot, siendo que en cada iteración la probabilidad acumulada es mayor y, por tanto, hay más partículas con alta probabilidad de ser la posición real. En la última captura se puede ver la estimación que realiza el robot una vez la generación converge en una circunferencia de radio dos, y esa estimación coincide con la posición verdadera.

5.5.1.2. Localización en movimiento

Al moverse el robot, no sólo se debe gestionar la comunicación con sus interfaces sensoriales, sino que también se envían órdenes al interfaz para el teleoperador, para repintar el robot en el mapa, para actualizar la trayectoria y para añadir el modelo de movimiento a las partículas. Gracias al tiempo de evolución y a la programación multihilo esto es posible de manera simultánea. La localización con movimiento implicado es una manera de poner a prueba el algoritmo en un caso complejo.

Salvando fallos iniciales en algunas ejecuciones (estimaciones erróneas por algunos metros en las primeras iteraciones, pero tolerables al principio de la ejecución), los errores de posición estimada obtenidos no superan en ningún caso los 20cm, siendo la media de menos de 10cm (Figura 5.24):

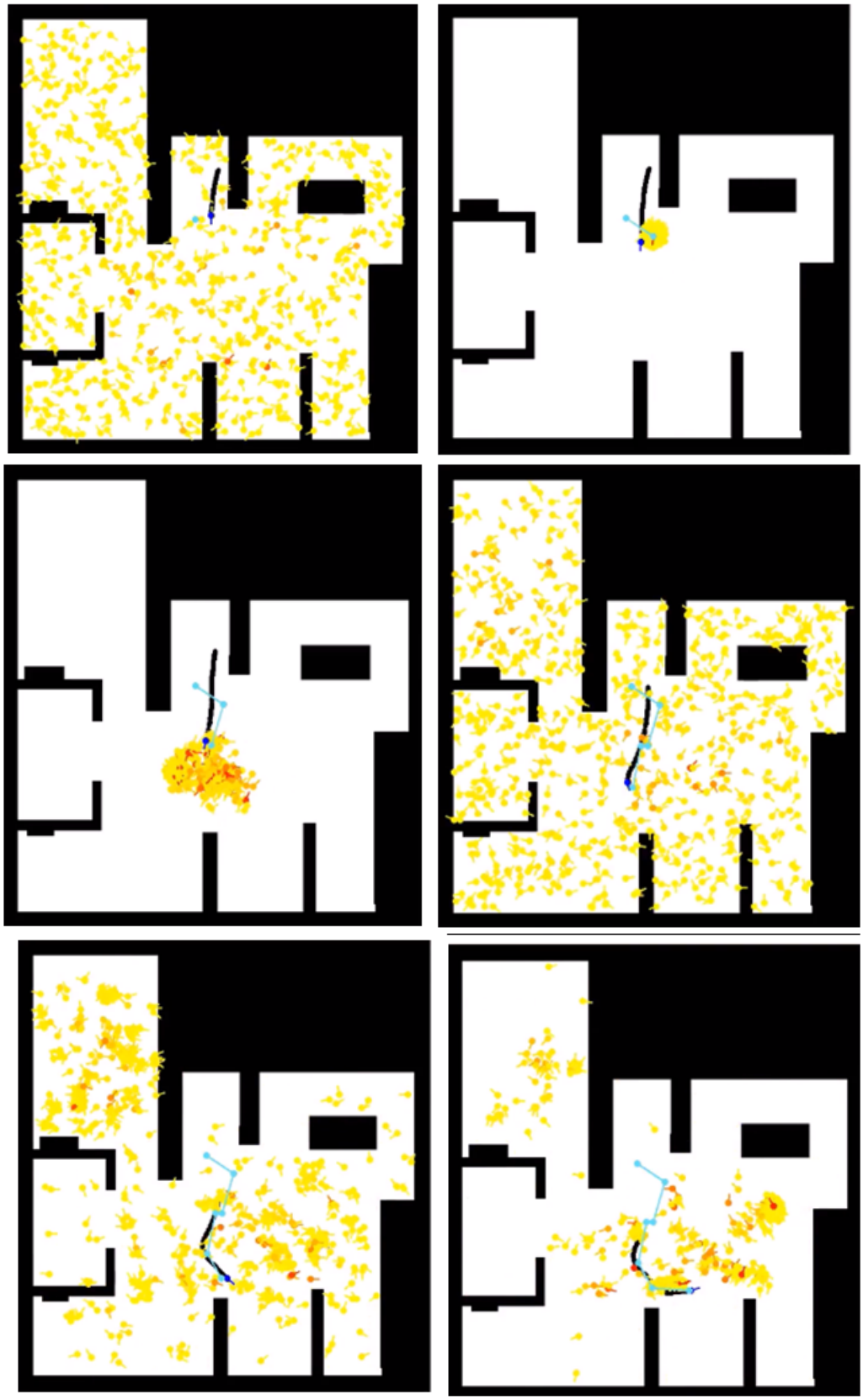

Figura 5.24: Evolución del algoritmo de localización con el robot moviéndose

Estas capturas representan distintos instantes temporales, en cada uno de los cuales se ha realizado una nueva estimación de posición. Así, mientras el robot está siendo teleoperado (generando la trayectoria representada en negro) el algoritmo actúa para obtener estimaciones que la interfaz conectará entre sí, generando una trayectoria estimada (representada en azul claro). El algoritmo funciona del mismo modo, agrupando la generación en torno a la posición real, y con cada estimación se reinicia de modo aleatorio para obtener la siguiente. Algunas de estas capturas muestran un momento de agrupación, y otras un instante de reinicio.

5.5.2. Autolocalización en situaciones extremas

Una vez validados los casos más ilustrativos pusimos a prueba el algoritmo en condiciones complejas (simetrías, cercanía a paredes, etc.), para comprobar si la solución propuesta era lo suficientemente buena como para lidiar con los problemas que podían surgir en entornos reales.

Aunque los escenarios simulados están compuestos por elementos de distinta naturaleza, en muchas ocasiones se tienen espacios concretos con una geometría similar (Figura 5.25), que para el ojo humano es claramente distinguible, pero no así a priori para los sensores que puede incorporar un robot.

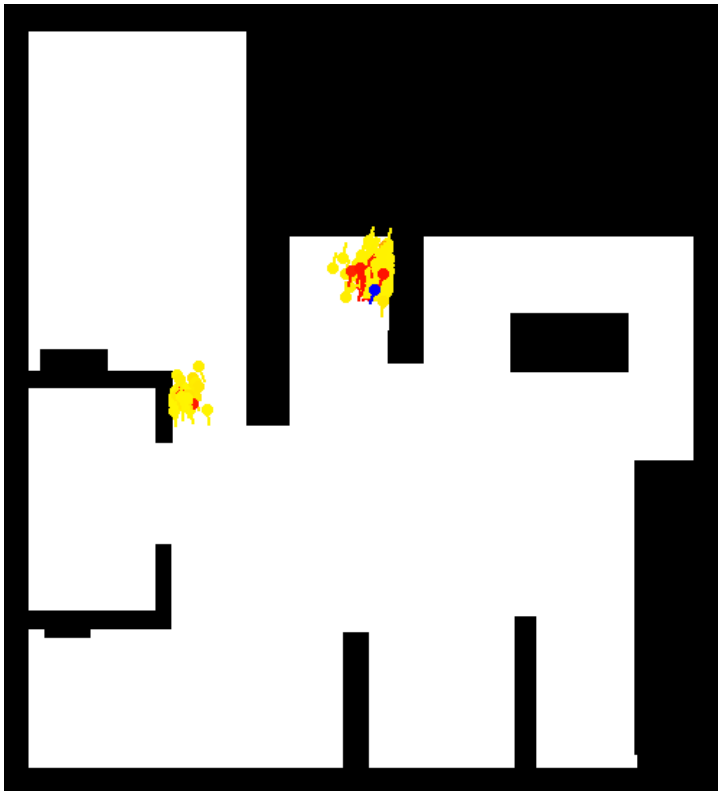

Figura 5.25: Espacios de geometría similar y zonas con simetrías

En las pruebas realizadas en escenarios con estas simetrías, tras unas cuantas iteraciones se forman una serie de grupos de partículas que acumulan una probabilidad similar (tantos como lugares con geometría parecida existan en el entorno). Ante casos como éste, el algoritmo de localización necesitó bastantes más iteraciones de evolución de generaciones (al menos el doble en la mayoría de los casos), pero en el 70 % de ellos la estimación final venía del grupo situado en la posición correcta, es decir, en los alrededores del robot, como demuestra el punto azul celeste de la Figura 5.26).

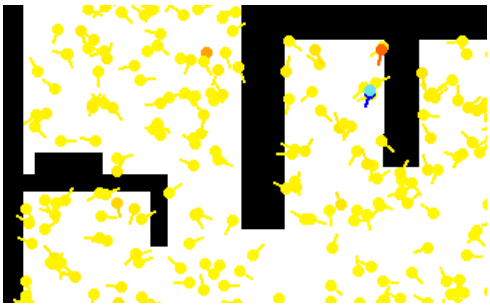

Figura 5.26: Estimación en espacios similares

Normalmente, estos problemas de simetría se solucionan añadiendo algún otro sensor que ayude a descartar unos datos frente a otros, o simplemente incorporando movimiento, pues las posibles similitudes se irán desvaneciendo a medida que el robot avance.

Por otro lado, quisimos ver qué sucedía en caso de situar al robot muy cerca de las superficies límite (de los obstáculos). En estos casos (Figura 5.27), el robot obtiene muchos puntos en los que podría obtener una lectura dentro de los márgenes de similitud con la lectura real, de manera que se generan muchos grupos de partículas, que pueden evolucionar favorablemente o no, dependiendo de cómo fuese la distribución aleatoria inicial (recordemos que las sucesivas generaciones surgen siempre de alguna partícula preexistente). Este experimento confirmó que un algoritmo de localización robusto conviene que use datos de distintos sensores: puede basarse en láser pero ayudaría si incorporase por ejemplo información de textura para lidiar con esto.

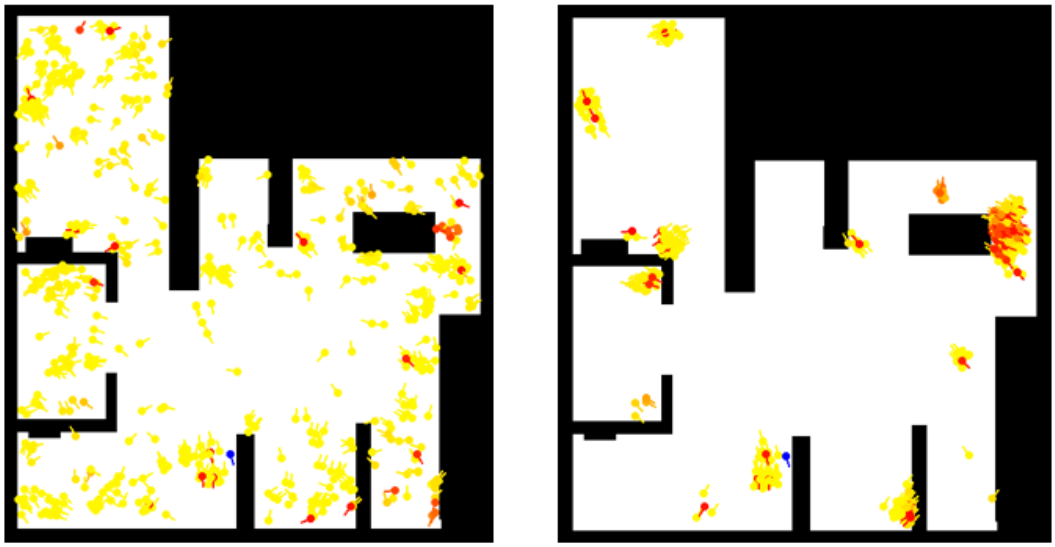

Figura 5.27: Posición cercana a los obstáculos

Por último, sabiendo que en la realidad se dan muchos espacios simétricos (salas cuadradas, mobiliario rectangular,...), quisimos hacer un experimento en un entorno altamente simétrico, para lo cual sirvió el modelo explicado en 5.2.3, el cual es cuadrado y sólo incorpora algunos elementos que pueden generar lecturas distintas en algunos puntos (Figura 5.28):

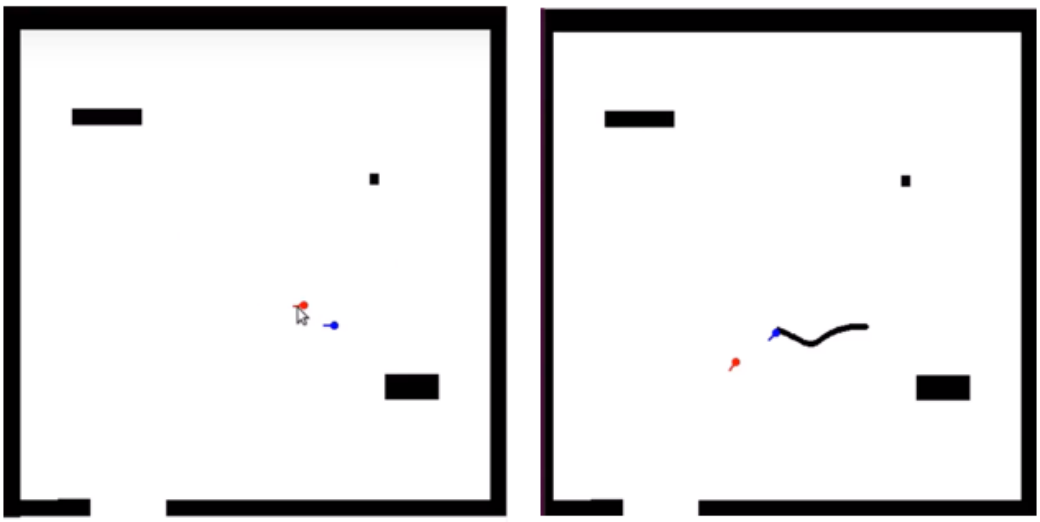

Figura 5.28: Estimación de autolocalización en entorno altamente simétrico

El algoritmo continuó funcionando y realizando estimaciones de posición, aunque éstas en muchos casos obtenían errores más elevados, en torno a los 10-15 cm. En muchas ocasiones, estos errores pueden verse incrementados por las desviaciones que se hayan cometido en el modelado del mapa, pero en cualquier caso el factor más influyente es la simetría, que al ser notable puede traducirse en distintos puntos de la sala con lecturas muy parecidas. La generación inicial aleatoria influirá en el error cometido, y por tanto la localización varía aunque se trate del mismo punto en distintas ejecuciones. En cualquier caso, las pruebas realizadas consiguieron localizar al robot cometiendo un error al límite de lo tolerable.

5.5.3. Ajuste del modelo de observación

Para alcanzar la solución se ha pasado por un proceso de ajuste de cada módulo de manera independiente. Para ajustar el modelo de observación láser empleado, y que las probabilidades que asigne se adecúen a la escena, se han usado distintos métodos:

Malla de partículas

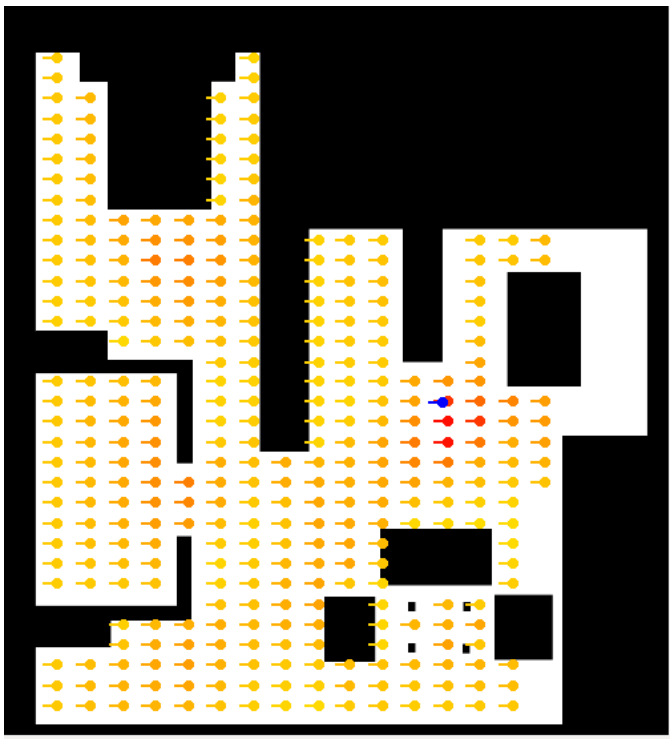

Figura 5.29: Malla de partículas

Se creó una especie de rejilla de partículas (Figura 5.29), de manera que se puede ver de forma gráfica y sencilla la distribución de la probabilidad. Para partículas cercanas al robot, las representaciones asociadas deben tomar un color más rojo, mientras que las lejanas deben tirar más hacia el amarillo. Jugando con la separación entre partículas obtuvimos la función del modelo de observación idónea para el tipo de comparación implementado entre el láser real y el teórico.

Slider de Orientación de partículas

De manera análoga a lo anterior, se incluyó un *slider* (Figura 5.30) que permitía controlar la orientación de todas las partículas al unísono y recalcular su probabilidad tras el cambio.

Figura 5.30: *Slider* de orientación

Esto permitió ajustar en mayor medida el modelo concreto de observación, ya que partículas en posición cercana y orientación similar (con variaciones entre 0º y 10º) deben tener una probabilidad alta, pero en posiciones similares y orientaciones distintas (>20º) deben tener probabilidades bajas (Figura 5.31).

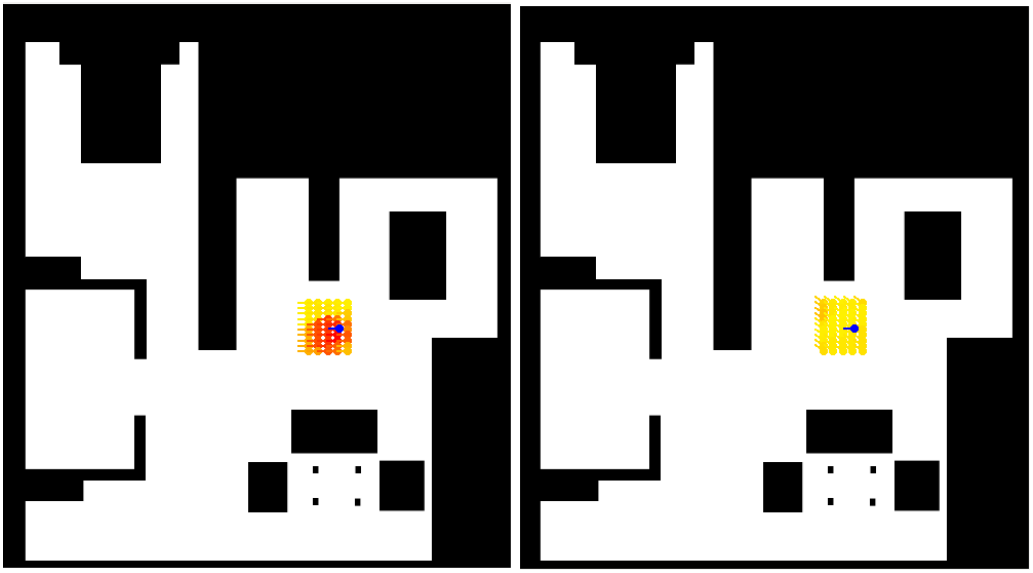

Figura 5.31: Ajuste de Modelo de Observación (diferencia angular de 0º vs. diferencia angular de 40º con la orientación real)

5.5.4. Ajuste del Ruido Gaussiano en la generación de partículas

Para llevar a cabo la exploración de una zona en la que existe alrededor una partícula de alta probabilidad, la descendencia de la misma se crea añadiendo un pequeño ruido que no debe dispersar demasiado las nuevas partículas generadas, ya que la intención es encontrar una nueva posición más favorable, la cual tiene alta probabilidad de estar muy cerca y en una orientación muy parecida. Se probó con diferentes configuraciones de

ruido, para al final decantarnos por una distribución normal centrada en la posición del progenitor con una desviación típica σ de 0.1.

5.5.5. Ajuste del tiempo de evolución de las partículas

Ya se ha mencionado que las generaciones no evolucionan en todas las iteraciones del algoritmo, sino que se ha establecido un tiempo de evolución. El ajuste de este tiempo se basado en la rapidez del algoritmo en proporcionar una estimación: de media han sido necesarias 13 generaciones para obtener una estimación, y en vistas a lograr la localización de un robot en movimiento, se decidió que era necesaria al menos 1 estimación cada 4 segundos, de manera que se fijó dicha evolución en intervalos de 300 ms.

5.6. Cuadernillo académico Jupyter

Esta práctica lleva mucha más carga y complejidad gráfica que la descrita en el Capítulo 4, de manera que hemos tenido que explorar las opciones de interfaz gráfica que ofrece Jupyter. Aunque permite lanzar subprocesos desde los *Notebooks*, la idea que nos ha resultado más atractiva ha sido la de construir *widgets* interactivos, controlados por código Python, a través de bibliotecas de representación como *MatPlotLib*, que recogen eventos de usuario y los materializan en la funcionalidad necesaria, permitiendo así disponer de una interfaz gráfica en el cuadernillo. Con estos nuevos integrantes sumados a los que ya habíamos empleado (celdillas de código, imágenes, texto con formato, etc.) hemos preparado un cuadernillo de Jupyter al que hemos llamado *laser_loc.ipynb* (Figura 5.32), a través del cual accederemos a la versión de Jupyter de esta práctica.

En esta ocasión, los cambios respecto del nodo son mayores que en el ejercicio anterior, dado que es necesario gestionar toda la funcionalidad de interacción desde el mismo, y además construir la forma de comunicación entre el cuadernillo y el *kernel* de Jupyter. En concreto, el nodo programado en el fichero *lase_loc.py* de la práctica original ha sido recodificado para tener estructura de clase Python (*LaserLoc*), a la cual le hemos incorporado todos los atributos y métodos necesarios tanto para el soporte de la práctica como para su parte gráfica.

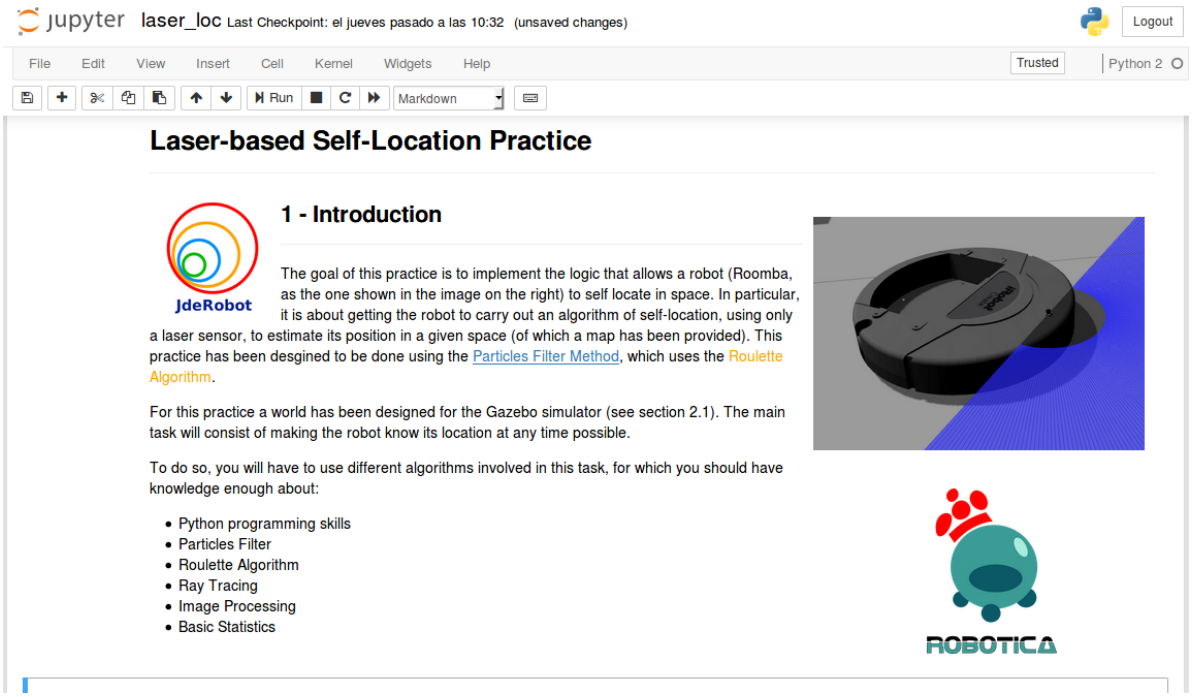

Figura 5.32: Cuadernillo del ejercicio de localización por filtro de partículas

Aunque los elementos principales del *Notebook* son los *widgets* interactivos y las celdas de código, también se ha incluido texto de apoyo para los pasos a seguir en la resolución de la práctica e imágenes representativas de las salidas que se pueden ir obteniendo, todo lo cual creemos que es de gran ayuda para orientar al alumno, en especial, la descripción del API de funcionalidad que la clase *LaserLoc* del cuadernillo pone a su disposición.

En cuanto a los elementos interactivos, hemos incluido principalmente dos dinámicos y uno estático basados en los que ya había en la interfaz de la versión de escritorio de la práctica. El *widget* del teleoperador (Figura 5.33) no es más que una representación básica de dos rectas sobre un fondo negro. Sin embargo, ha sido preparado para recoger eventos del ratón desde la aplicación (*press* y *release*), y para enviar órdenes a los actuadores del robot en función de estos y que se reflejen en la simulación. Así, si el usuario hace click sobre este *widget*, arrastra la cruz hasta un punto determinado y deja de presionar el ratón, consigue materializar un movimiento proporcional al movimiento de la cruz en el robot. Con este elemento mantenemos la posibilidad de probar el algoritmo en condiciones de localización en movimiento.

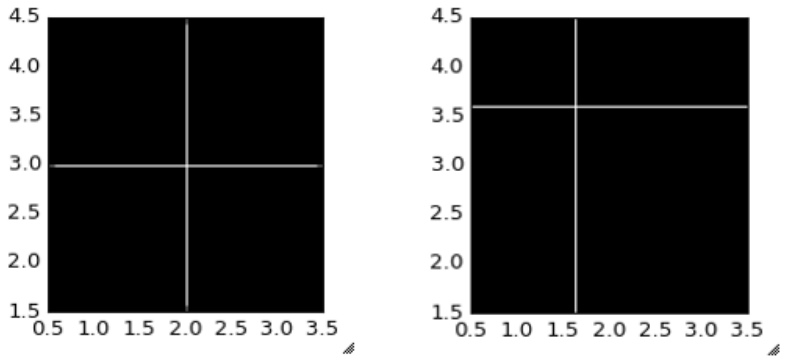

Figura 5.33: Teleoperador en Jupyter

El *widget* del mapa binario (Figura 5.34) consta de varios elementos. Como se puede ver en la Figura inferior, se han dispuesto 3 botones que también recogen la interacción. El primero (en rojo), tiene que ver con el *widget* anterior, y sirve para detener el movimiento del robot y restablecer la posición del teleoperador, para facilitar el control del mismo. El segundo (en azul) es el que se utiliza para ejecutar la función *cretaeNewGeneration*, una de las dos que el alumno debe implementar para alcanzar la solución. Esta función es la encargada de hacer evolucionar las generaciones bajo la orden del alumno, con cada click genera un nuevo conjunto de partículas basado en el precedente. Por último (en verde), el botón *Play Code* ejecuta el código del método *execute*, el cual también debe ser codificado por el alumno, que recoge la funcionalidad principal del algoritmo, y que debe gestionar también el cálculo del láser de una posible partícula sobre la que se haga click en el mapa.

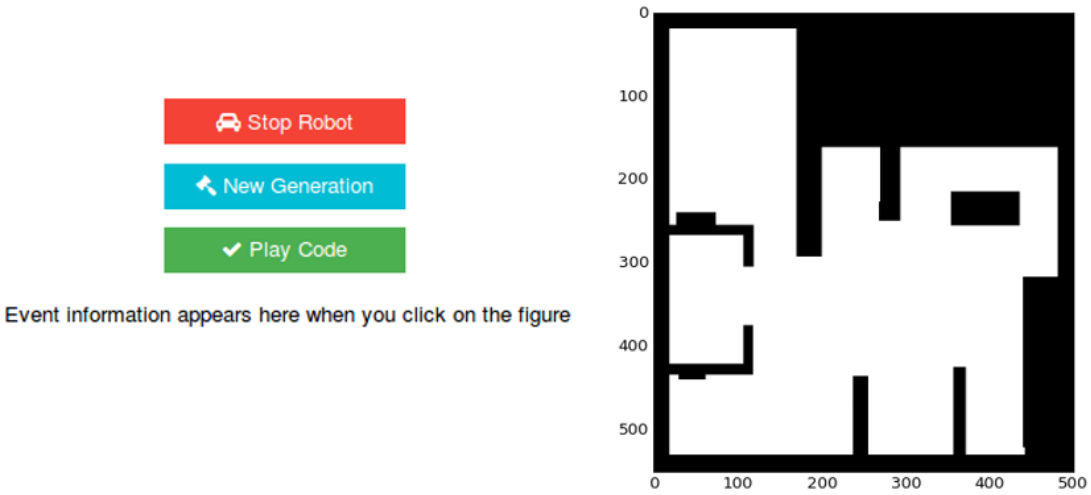

Figura 5.34: Widget de Mapa Binario en Jupyter

El elemento principal del *widget* es el mapa, que se ha extrapolado directamente desde la interfaz de la práctica, siendo una imagen binaria que permitirá visualizar la evolución del algoritmo. También es un elemento dinámico que variará cada vez que evolucione una generación, y que recogerá los clicks del usuario para traducir las coordenadas del ratón en coordenadas de una partícula y así poder realizar el cálculo de su láser y su representación.

La representación láser (Figura 5.35) es un *widget* estático equivalente a la gráfica de los láseres real y teórico de la práctica original. En cuanto se hace click sobre una partícula en el mapa, si se ejecuta la celdilla de código del *Notebook* indicada para la representación láser, se refresca su salida, donde el láser teórico (rojo) se superpone a los datos reales (azul). La celdilla de este elemento también mostrará por su salida la probabilidad asociada a la partícula sobre la que se ha hecho click.

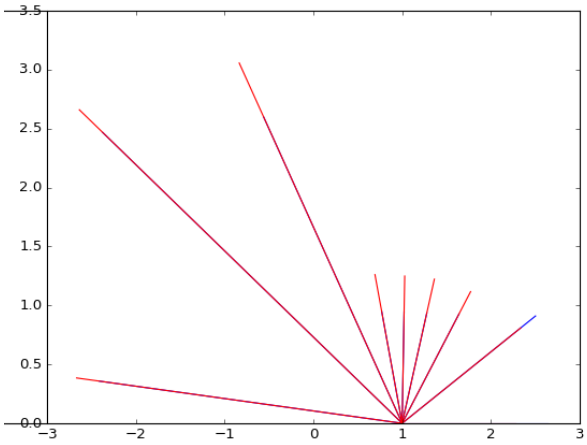

Figura 5.35: Widget Láser en Jupyter

Todos estos elementos contribuyen a que la funcionalidad de la práctica esté soportada desde Jupyter, de manera que en una ejecución concreta se obtienen salidas equivalentes a los que se obtendrían fuera de esta aplicación (Figura 5.36).

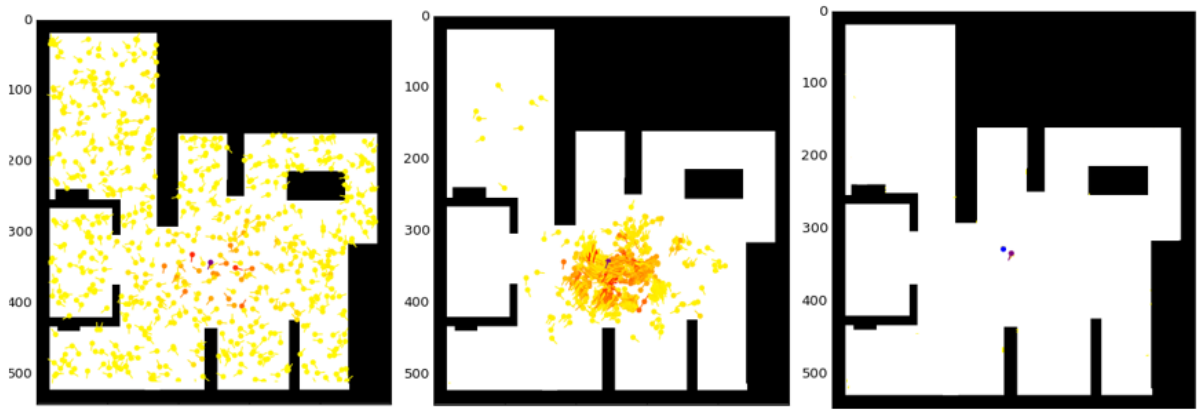

Figura 5.36: Evolución de una ejecución típica del ejercicio en un cuadernillo de Jupyter

En [3] se puede ver un vídeo demostrando el comportamiento de la práctica a través de un cuadernillo Jupyter.

[3] `https://www.youtube.com/watch?v=kF7szwrC6a0`

Capítulo 6

Conclusiones

Tras detallar en profundidad las dos prácticas e implementaciones de las mismas que componen este proyecto, este capítulo se ha reservado para ver hasta qué punto se han logrado los objetivos establecidos, para resumir los conocimientos adquiridos durante su desarrollo y para exponer las posibles mejoras que se pueden introducir en las prácticas, así como trabajos futuros para completar o extender la funcionalidad.

6.1. Conclusiones

El objetivo global de este proyecto era crear dos prácticas nuevas en el entorno docente JdeRobot-Academy. Este objetivo se ha alcanzado con éxito a través de las dos prácticas totalmente funcionales y validadas experimentalmente para comprobar su correcto funcionamiento. Dicho objetivo principal se subdivide en distintos objetivos secundarios, de los cuales a continuación se comentará hasta qué punto se han logrado y cómo.

El primero de ellos consistía en la creación de la práctica de seguimiento de objetos con una cámara móvil, la cual involucraba hardware real. El objetivo se cumplió satisfactoriamente con el desarrollo de la infraestructura y el nodo académico, además de utilizar los *drivers* que permitieron la comunicación entre el nodo y el robot. Hubo que lidiar con ciertos parámetros implícitos en el robot real que muchas veces no son tenidos en cuenta en simulaciones, como la presencia de un *buffer* de mensajes y con el tiempo físico de ejecución de los movimientos del cuello mecánico. Se ha dejado resuelta toda

la configuración y se ha añadido un fichero de instrucciones y enunciado que ayudará a futuros usuarios, teniendo así una práctica totalmente funcional y orientada.

El segundo subobjetivo fue la elaboración de práctica de autolocalización de un robot usando un filtro de partículas y sensor láser. Aunque esta práctica implicaba bastante complejidad gráfica y mucho peso computacional, el objetivo se alcanzó utilizando técnicas de programación multihilo, precomputación y un manejo inteligente de los datos involucrados. En este caso, se preparó un mundo y un modelo de simulación (tanto en aspecto como en funcionalidad), se creó un nodo académico y su infraestructura, y se dejó preparado para incrustar una solución.

Asociado a ambos subobjetivos hubo que construir sendas soluciones de referencia (una para cada una de las prácticas). No sólo servirían como modelo de posibles soluciones para cada ejercicio, sino que también traería consigo el estudio y asimilación de técnicas de control, procesado, captación y actuación que emplean los robots reales.

En el caso de la primera práctica, se hizo uso del aprendizaje máquina para entrenar un sistema básico capaz de reconocer caras en posición frontal y otro adicional para reconocer ojos, que juntos permitieron discriminar si una imagen dada contenía caras o no. Estos datos analizados sirvieron para implementar un algoritmo de control gradual, de tal manera que la segmentación de la imagen se traducía en órdenes para los motores de la cámara Sony Evi d100p, la cual seguía al individuo de la imagen. Para hacer la solución algo más completa y robusta, también se incluyó una lógica de gestión en caso de no detectar rostros.

Para la práctica de localización se propuso un algoritmo que utiliza el filtro de partículas. Este proceso involucró varias funciones que solventaban distintas partes del mismo, como el algoritmo de la ruleta para la selección de una partícula a evolucionar, la geometría euclidiana para materializar desplazamientos en las partículas, o la estadística básica para implementar un modelo de observación de partículas. Esta solución se completó con técnicas de mejora de la eficiencia, con soporte para la localización en movimiento y con una serie de ayudas gráficas.

Adicionalmente, queríamos estudiar los requisitos necesarios para construir prácticas que estuviesen disponibles para la mayoría de usuarios. Esto ha sido posible gracias a la plataforma Jupyter, que nos permitió migrar nuestras aplicaciones de escritorio a una versión web, que conservaba la funcionalidad de las originales. Esto abre la puerta para

su disponibilidad en distintas plataformas.

Por último, una motivación personal implícita era adquirir el conocimiento y las habilidades necesarias para abordar un problema real de ingeniería, en el cual hubieran involucrados componentes hardware y software. Así, se ha logrado integrar los conceptos que sustentan la infraestructura que hay bajo un proyecto de robótica, las interfaces involucradas, los componentes y, finalmente, la parte visible por el usuario. Por el camino se han adquirido otra serie de competencias útiles y presentes en muchos proyectos, como el modelado de elementos de simulación, el diseño y manejo de interfaces gráficas, el tratamiento de imagen y distintas técnicas de programación.

6.2. Trabajos futuros

El desarrollo de este trabajo ha abierto las puertas a distintas ideas que pueden estudiarse y realizarse en próximos proyectos:

El más destacado es el de alcanzar enteramente unas aplicaciones multiplataforma. El entorno JdeRobot-Academy ha desarrollado recientemente una versión web que permite utilizar el simulador Gazebo de manera remota en consonancia con la plataforma Jupyter. Utilizando un servidor remoto que contenga las prácticas, son ejecutables en cualquier lugar y a través de las plataformas principales (Linux, MacOS, Windows). Aunque aún está en proceso de depuración, más adelante resultaría interesante retocar las prácticas creadas para que estén disponibles a través de dicha herramienta web.

Otro tema interesante de estudio es la localización basada en visión. Se podría construir una nueva práctica, basada en el nodo académico ya creado (o una versión muy parecida) para abordar el problema de la localización utilizando otro tipo de sensores, como una o varias cámaras. Incluso, podría enriquecerse el modelo empleado para incorporar distintos sensores de diferentes tipos que recogieran información que compusiera una solución más robusta.

Con la primera práctica resultaría interesante montar la cámara en un pequeño robot móvil, y abordar el seguimiento de caras en el espacio 3D. Para ello, sería necesario explorar los tipos de robots móviles, sus motores y sus formas de comunicación, para luego hacerlos compatibles con el nodo e implementar una lógica de seguimiento de personas.

Siempre queda abierta la posibilidad de abordar las mismas prácticas haciendo uso de otro tipo de algoritmos, para así poder realizar una comparación entre ellos y extraer conclusiones acerca de la validez, ventajas y desventajas de cada uno.

Finalmente varios flecos menores también se pueden abordar para mejorar las dos prácticas creadas en este TFG:

- La solución propuesta para la práctica Follow Face es capaz de reconocer caras frontales, y en el mejor de los casos rostros girados como mucho 45º desde el eje frontal. Una posible mejora podría ser entrenar el sistema de clasificación con observaciones que incluyan caras rotadas con distintos ángulos, a fin de reconocer todos los casos positivos.

- Las comunicaciones en esa práctica emplean dos drivers, uno de los cuales utiliza interfaces ICE. Para la mejora de accesibilidad a la práctica se podría crear uno equivalente que se comunicase a través de mensajes de ROS.

- En la práctica de autolocalización, el elevado peso computacional hace que, incluso con las técnicas de optimización empleadas, la eficiencia siga siendo un punto que se puede pulir. Convendría implementar nuevas optimizaciones.

Bibliografía

[1] Irene Lope Rodríguez. Nuevas Prácticas en el Entorno Docente de Robótica JdeRobot-Academy, 2018.

[2] Vanessa Fernández Martínez. Nuevas Prácticas en el Entorno Docente de Robótica JdeRobot-Academy, 2017.

[3] John J. Craig. *Introduction to Robotics: Mechanics and Control*. 1986.

[4] Simone Ceriani and Martino Migliavacca. *Middleware in robotics. Advanced Methods of Information Technology for Autonomous Robotics. Internal Report For Advanced Methods of Information Technology for Authonomous Robotics, Politecnico di Milano, 2012.*

[5] J. Baillie, A. Demaille, Q. Hocquet, M. Nottale, and S Tardieu. *The Urbi Universal Platform for Robotics. Simulation, Modeling and Programming for Autonomous Robots. 4th International Conference, SIMPAR 2014, Bergamo, Italia, 2014.*

[6] Historia de la robótica. `http://wiki.robotica.webs.upv.es/wiki-de-robotica/introduccion/historia/`.

[7] Herman Bruyninckx and Peter Soetens. The OROCOS Project. `https://people.mech.kuleuven.be/~orocos/pub/stable/documentation/rtt/v1.10.x/doc-xml/orocos-overview.html`, 2007.

[8] Cyberbotics. Webots. `https://www.cyberbotics.com/webots.php`, 2015.

[9] AForge.NET. AForge.NET Framework. `http://www.aforgenet.com/framework/`.

[10] Simulación en Gazebo. `https://robologs.net/2016/06/25/gazebo-simulator-simular-un-robot-nunca-fue-tan-facil/`.

[11] Definición de Robot. `http://conceptodefinicion.de/robot/`.

[12] Definición de Robótica. `https://www.definicionabc.com/tecnologia/robotica.php`.

[13] Educatronics. Importancia de la Robótica en la Educación. Ciencia, arte y arquitectura. http://educatronics.com/publicaciones/importancia-de-la-rob%C3%B3tica-en-la-educacion.

[14] J.M. Cañas, A. Martí, E. Perdices, F. Rivas, and R. Calvo. Entorno docente para la programación de la inteligencia de los robots. Revista Iberoamericana de Automática e Informática Industrial, 2016.

[15] Galo Fariño R. Modelo Espiral de un proyecto de desarrollo de software, Administración y Evaluación de Proyectos. http://www.ojovisual.net/galofarino/modeloespiral.pdf.

[16] Modelo de desarrollo en Espiral. https://pdfs.semanticscholar.org/presentation/2403/7678f5cfd0d23d1a7d59b9b9ff2babc31d99.pdf.

[17] Simulador Gazebo. https://dev.px4.io/en/simulation/gazebo.html.

[18] J.M Cañas. Programación de robots con la plataforma JdeRobot. Universidad de Málaga., 2009.

[19] José M. Cañas, José A. Fernández, Diego Jiménez, and Jorge Vela. Programación de aplicaciones para drones con el entorno software JdeRobot. Congreso CivilDRON 2018, 2018.

[20] ¿Qué es Python? https://www.python.org/doc/essays/blurb/.

[21] Historia de Python. https://www.codejobs.biz/es/blog/2013/03/03/historia-de-python.

[22] Descripción de OpenCV. https://opencv.org/about.html.

[23] Definición del fromato SDF. http://sdformat.org/.

[24] Qué es y para qué se usa la biblioteca PyQt5. https://pypi.org/project/PyQt5/.

[25] PyQt5. https://www.riverbankcomputing.com/software/pyqt/intro.

[26] Plataforma Jupyter, página oficial. http://jupyter.org/.

[27] Guía de Jupyter Notebook. https://www.datacamp.com/community/tutorials/tutorial-jupyter-notebook.

[28] Jupyter Notebook, descripción del proyecto. http://nbviewer.jupyter.org/github/jupyter/notebook/blob/master/docs/source/examples/Notebook/What%20is%20the%20Jupyter%20Notebook.ipynb#.

[29] IPython3. https://ipython.org/ipython-doc/3/whatsnew/version3.html.

[30] Manual de la cámara Sony Evi d100p. https://www.manualslib.com/manual/157936/Sony-Evi-D100.html?page=3#manual.

[31] Digitalizadora Dazzle DVD Recorder HD. http://www.informaticamoderna.com/Captura_video_externo.htm.

[32] Driver usb_cam. http://wiki.ros.org/usb_cam.

[33] Algoritmos de detección de caras. https://en.wikipedia.org/wiki/Face_detection#Definition_and_related_algorithms.

[34] Face Detection And Recognition. https://facedetection.com/algorithms/.

[35] Detección de rostros con OpenCV. https://www.superdatascience.com/opencv-face-detection/.

[36] Paul Viola and Michael J. Jones. Robust Real-Time Face Detection. SECOND INTERNATIONAL WORKSHOP ON STATISTICAL AND COMPUTATIONAL THEORIES OF VISION - MODELING, LEARNING, COMPUTING, AND SAMPLING. Vancouver, Canadá, 2001.

[37] Shai Shalev-Shwartz and Shai Ben-David. Understanding Machine Learning: From Theory to Algorithms, 2014.

[38] Librería Matplotlib. https://matplotlib.org/users/pyplot_tutorial.html.

[39] Biblioteca types de Python. https://docs.python.org/2/library/types.html.

[40] Sensor láser. http://wiki.robotica.webs.upv.es/wiki-de-robotica/sensores/sensores-proximidad/sensor-laser/.

[41] What is odometry? https://groups.csail.mit.edu/drl/courses/cs54-2001s/odometry.html.

[42] Topics de ROS. http://wiki.ros.org/Topics.

[43] M. Isard and A. Blake. Condensation-conditional density propagation for visual tracking. Int. J. Computer Vision, in press, 1998.

[44] S. Thrun. Probabilistic algorithms in robotics. AI Magazine, 2000.

[45] Andrew S. Glassner. An Introduction to Ray Tracing, 1989.

[46] Dieter Fox, Wolfram Burgard, Frank Dellaert, and Sebastian Thrun. Monte Carlo localization: efficient position estimation for mobile robots. *In Proceedings of the 16th AAAI National Conference on Artificial Intelligence*, pages 343–349. Orlando (Florida, USA), July 1999.

[47]] D.J.C. Mackay. Introduction to Monte Carlo methods. Cambridge University.

[48] J. Alvarez Cedillo, E. Acosta Gonzaga, J. Herrera Lozada, T. Alvarez Sanchez, J. Sandoval Gutierrez, and M. Aguilar Fernandez. PARTICLE FILTER BASED LOCALIZATION FOR A MOBILE ROBOT. *DYNA New Technologies*, 2017.

[49] Robot Kobuki . `http://kobuki.yujinrobot.com/wiki/online-user-guide/`.