



ESCUELA TÉCNICA SUPERIOR DE INGENIERÍA DE
TELECOMUNICACIÓN

GRADO EN INGENIERÍA EN SISTEMAS
AUDIOVISUALES Y MULTIMEDIA

TRABAJO FIN DE GRADO

Nuevas Prácticas en el Entorno
Docente de Robótica JdeRobot-Academy

Autor: Vanessa Fernández Martínez

Tutor: José María Cañas Plaza

Curso académico 2017/2018

Agradecimientos

En primer lugar, me gustaría dar las gracias a mi familia, que me ha apoyado a lo largo de este tiempo. Todos ellos han mostrado interés por el trabajo y lo han hecho más fácil. Sobre todo quiero mencionar a mis padres y mi hermana, los cuales me han animado, me han dado todo su cariño, y me han ayudado en los momentos más difíciles, pues sin ellos no hubiese sido posible conseguirlo.

Gracias a José María, por su dedicación y apoyo durante todos estos meses de desarrollo del trabajo, por su paciencia y su ayuda. Y por supuesto, por haberme transmitido su pasión por la robótica.

Por otro lado, quiero dar las gracias a mi compañera de proyecto Irene Lope, que me ha ayudado en los momentos donde no salían las cosas y por los ánimos que me ha transmitido. Además, quiero dar las gracias a mis compañeros del laboratorio, en especial a Fran y Aitor, por su amabilidad y paciencia con todos los problemas que han surgido durante el proyecto; y por supuesto, a Carlos y Nacho, que nos han sacado una sonrisa cada día.

Por supuesto, gracias a mis amigos de la carrera: Nuria, Carolina, Isa, Miguel Ángel, Celia y Marta, que me han animado durante todos estos años y que han compartido conmigo el estrés de los exámenes. Además, gracias a mis amigos que me han apoyado durante tantos años en los buenos y los malos momentos: Marcos, Nerea, Lizaveta, Chai-mae, Manal, Marian, Juan Miguel, Isabella, etc.

Por último, no puedo olvidarme de dar las gracias al CNAH, no solamente por la natación, sino sobre todo por la gente que lo forma, especialmente el equipo femenino. Gracias a las más jóvenes del equipo, por sacarme una sonrisa en todo momento. A las más mayores: Jessi, Concha, María, Elena y Laura, que me han hecho desconectar cada día del estrés y pasármelo como nunca. No puedo olvidarme, de Manolo Revilla, que desde que llegué al club me ha tratado como una amiga y se ha preocupado por mí.

Muchas gracias a todos!

Resumen

En los últimos años, la presencia de la robótica ha aumentado considerablemente, haciendo que cada vez sea más necesario tener una formación en este ámbito. El entorno docente JdeRobot-Academy permite acercar esta materia a los alumnos de forma sencilla y eficaz. Este entorno consta de diferentes prácticas para la enseñanza de técnicas robóticas. En este proyecto se ha tratado de mejorar el abanico de posibilidades que ofrece este entorno, ampliando y mejorando las prácticas ya existentes.

Para lograr este fin se han empleado múltiples herramientas. Creando nuevos mundos en el simulador Gazebo, creando los componentes académicos necesarios, e incluso creando un evaluador automático de prácticas. El objetivo de estos componentes académicos es abstraer a los alumnos de todas las complejidades de la interfaz gráfica, y conexiones con los sensores y actuadores; haciendo posible que el alumno se centre en la resolución de los algoritmos que se plantean en cada práctica.

Se pretende mejorar la práctica “TeleTaxi” (creada con el fin de resolver un algoritmo de navegación global) para reproducir un entorno más realista, y mejorar el componente académico que sirve de apoyo al alumno. Además, se ha dotado a la práctica de un evaluador automático que permite calificar el algoritmo que programa el alumno. Asimismo, se propondrá una solución para esta práctica, donde se empleará la técnica *Gradient Path Planning*. Dicha técnica se emplea para guiar al robot desde su posición inicial hasta la posición de destino, evitando chocar con los obstáculos descritos en el mapa.

Con el propósito de mejorar el entorno docente, se creará la práctica “Aspiradora autónoma”. La elaboración de la misma incluirá la creación de la infraestructura necesaria, el componente académico y el evaluador automático. Esta práctica pretende que el alumno se familiarice con un algoritmo de navegación sin autolocalización, donde el propósito es que el robot recorra la mayor superficie posible de una casa. Este algoritmo se basará en los algoritmos de navegación que emplean los modelos de la serie 500 de Roomba de iRobot. En esta práctica se ha desarrollado una solución que resuelve el

problema planteado.

Con el mismo fin se creará la práctica “Aparcamiento automático”, la cual implicará el desarrollo de la infraestructura necesaria para crear un entorno apropiado, la creación del componente académico que simplifica al alumno la resolución de la práctica, y la creación de un evaluador automático. El propósito de esta práctica es que el alumno tome contacto con técnicas de aparcamiento autónomo. En el proyecto se desarrollará una solución “ad hoc”, que tiene en cuenta los datos que proporcionan los sensores para poder tomar decisiones y aparcar de forma correcta.

Índice general

| | |
|---|-----------|
| Índice de figuras | IX |
| Índice de tablas | X |
| Acrónimos | XI |
| 1. Introducción | 1 |
| 1.1. Robótica | 1 |
| 1.2. Software para robots | 4 |
| 1.2.1. Middlewares robóticos | 5 |
| 1.2.2. Simuladores robóticos | 5 |
| 1.2.3. Bibliotecas | 6 |
| 1.3. Docencia en robótica | 7 |
| 1.3.1. Docencia en secundaria | 8 |
| 1.3.2. Docencia en la universidad | 9 |
| 1.4. Entorno Docente JdeRobot-Academy | 11 |
| 2. Objetivos | 16 |
| 2.1. Objetivos | 16 |
| 2.2. Requisitos | 18 |
| 2.3. Metodología | 18 |
| 2.4. Plan de trabajo | 20 |
| 3. Infraestructura | 22 |
| 3.1. Simulador Gazebo | 22 |
| 3.2. Entorno JdeRobot | 24 |
| 3.3. Python | 26 |
| 3.4. Biblioteca OpenCV | 26 |
| 3.5. PyQt | 28 |

| | |
|---|-----------|
| 4. Práctica: Navegación global de un TeleTaxi con GPP | 30 |
| 4.1. Enunciado | 30 |
| 4.2. Infraestructura | 31 |
| 4.2.1. Coche Taxi_Holo | 31 |
| 4.2.2. Modelo de ciudad cityLarge | 33 |
| 4.2.3. Mundo de Gazebo | 34 |
| 4.3. Componente Académico | 35 |
| 4.3.1. Interfaz gráfica | 37 |
| 4.3.2. Código auxiliar: Clase Grid | 40 |
| 4.4. Solución de referencia | 45 |
| 4.4.1. Fundamentos de la Navegación global y el algoritmo GPP | 46 |
| 4.4.2. Construcción del mapa del gradiente | 48 |
| 4.4.2.1. Generación campo ficticio de navegación global | 48 |
| 4.4.2.2. Penalización por cercanía de obstáculos | 51 |
| 4.4.2.3. Cálculo de ruta ideal | 53 |
| 4.4.3. Pilotaje del robot | 55 |
| 4.5. Evaluador Automático | 62 |
| 4.6. Experimentación | 64 |
| 4.6.1. Ejecución típica | 64 |
| 4.6.2. Estudio de tiempos | 64 |
| 5. Práctica: Aspiradora autónoma | 69 |
| 5.1. Enunciado | 69 |
| 5.2. Infraestructura | 70 |
| 5.2.1. Modelo Roomba | 70 |
| 5.2.1.1. Sensor láser | 71 |
| 5.2.1.2. Sensor bumper | 71 |
| 5.2.2. Modelo House_int2 | 72 |
| 5.2.3. Mundo de Gazebo | 73 |
| 5.3. Componente académico | 75 |
| 5.3.1. Interfaz gráfica | 76 |
| 5.3.2. Gráfica de la derivada del porcentaje | 79 |
| 5.4. Solución de referencia | 80 |
| 5.4.1. Algoritmos empleados por distintas aspiradoras robóticas | 81 |

| | |
|---|------------|
| 5.4.2. Solución desarrollada | 84 |
| 5.5. Evaluador automático | 88 |
| 5.6. Experimentación | 90 |
| 5.6.1. Ejecución típica | 90 |
| 5.6.2. Ajuste del tiempo de navegación perimetral | 91 |
| 5.6.3. Experimentos de larga duración | 94 |
| 5.6.4. Ley de los rendimientos decrecientes | 95 |
| 6. Práctica: Aparcamiento Automático | 97 |
| 6.1. Enunciado | 97 |
| 6.2. Infraestructura | 98 |
| 6.2.1. Modelo Taxi_Holo_Laser | 98 |
| 6.2.1.1. Sensores láser | 99 |
| 6.2.2. Modelo acera | 99 |
| 6.2.3. Modelo carNoMotor | 100 |
| 6.2.4. Mundo de Gazebo | 100 |
| 6.3. Componente Académico | 103 |
| 6.3.1. Interfaz gráfica | 105 |
| 6.4. Solución de referencia | 106 |
| 6.4.1. Algoritmos empleados por los coches autónomos reales | 106 |
| 6.4.2. Solución reactiva | 110 |
| 6.5. Evaluador automático | 113 |
| 6.6. Experimentación | 115 |
| 6.6.1. Ejecución típica | 115 |
| 6.6.2. Aparcamiento lejano | 116 |
| 6.6.3. Experimento con la plaza casi superada | 117 |
| 7. Conclusiones | 119 |
| 7.1. Conclusiones | 119 |
| 7.2. Trabajos futuros | 121 |
| Bibliografía | 132 |

Índice de figuras

| | | |
|-------|--|----|
| 1.1. | Robot Asimo | 3 |
| 1.2. | Esquema del funcionamiento de un robot | 4 |
| 1.3. | Estructura | 12 |
| 1.4. | Fórmula 1 y circuito de Gazebo | 13 |
| 1.5. | Pioneer en el mundo de Gazebo y reconstrucción 3D | 14 |
| 1.6. | Drone en el mundo de Gazebo | 14 |
| 2.1. | Modelo en espiral | 19 |
| 3.1. | Simulador Gazebo | 23 |
| 3.2. | Ejemplo de componentes JdeRobot | 25 |
| 3.3. | Funciones de OpenCV | 27 |
| 4.1. | Modelo yellowTaxi | 32 |
| 4.2. | Modelo taxi_holo | 32 |
| 4.3. | Modelo antiguo cityLarge | 34 |
| 4.4. | Modelo nuevo cityLarge | 34 |
| 4.5. | Modelo cityLarge desde arriba | 34 |
| 4.6. | Interfaz gráfica (GUI) actual del GPP | 38 |
| 4.7. | Interfaz gráfica con el triángulo que representa al taxi mal pintado | 38 |
| 4.8. | Interfaz gráfica con el triángulo que representa al taxi correctamente pintado | 39 |
| 4.9. | Interfaz gráfica (GUI) antigua del GPP | 40 |
| 4.10. | Imagen antigua(izquierda) del mapa e imagen actual (derecha) | 41 |
| 4.11. | Imágenes con el centro incorrecto (izquierda) y el centro correcto (derecha) | 43 |
| 4.12. | Sistema de referencia del mundo (izquierda) y sistema de referencia del mapa (derecha) | 44 |
| 4.13. | Rotación sobre el sistema de referencia del mundo (sistema punteado) | 44 |
| 4.14. | Primera propagación del frente de onda | 49 |
| 4.15. | Esquema propagación frentes de onda | 50 |
| 4.16. | Esquema expansión del campo | 51 |
| 4.17. | Representación campos calculados | 53 |

| | |
|---|----|
| 4.18. Esquema expansión del campo | 55 |
| 4.19. Sistema de referencia absoluto (Gazebo) y sistema de referencia solidario con el robot | 58 |
| 4.20. Sistema de referencia absoluto y sistema de referencia solidario con el robot | 58 |
| 4.21. Sistema de referencia solidario con el robot y punto objetivo | 60 |
| 4.22. Posición 1 taxi en el mundo de Gazebo y en la GUI | 61 |
| 4.23. Posición 2 taxi en el mundo de Gazebo y en la GUI | 61 |
| 4.24. Posición 3 taxi en el mundo de Gazebo y en la GUI | 61 |
| 4.25. Posición 4 (destino) taxi en el mundo de Gazebo y en la GUI | 62 |
| 4.26. Evaluador Automático del GPP durante el pilotaje | 62 |
| 4.27. Imagen del mapa con el primer destino elegido | 66 |
| 4.28. Imagen del mapa con el segundo destino elegido | 67 |
| 4.29. Imagen del mapa con el segundo destino elegido | 67 |
| 5.1. Roomba de iRobot y modelo Roomba en Gazebo | 71 |
| 5.2. Mundo GrannyAnnie.world en Gazebo | 72 |
| 5.3. Modelo house_int2 en Gazebo | 73 |
| 5.4. Interfaz gráfica (GUI) de Aspiradora automática después de cierto tiempo de ejecución | 77 |
| 5.5. Sistema de referencia del mundo (izquierda) y sistema de referencia de la imagen (derecha) | 78 |
| 5.6. Gráfica de la derivada del porcentaje en función del tiempo | 80 |
| 5.7. Patrón de navegación de la aspiradora Trilobite | 81 |
| 5.8. Patrón de navegación de la aspiradora Xiaomi | 82 |
| 5.9. Algoritmo de navegación de Roomba | 84 |
| 5.10. Espiral de Arquímedes | 85 |
| 5.11. Espiral que realiza Roomba en la práctica | 86 |
| 5.12. Perímetro que recorre Roomba en la práctica | 87 |
| 5.13. Evaluador automático durante la ejecución de la práctica | 90 |
| 5.14. Evaluador automático con la nota final | 91 |
| 5.15. Superficie recorrida de las pruebas de 200 segundos de perímetro | 92 |
| 5.16. Superficie recorrida de las pruebas de 300 segundos de perímetro | 93 |
| 5.17. Superficie recorrida de las pruebas de 45 minutos | 95 |
| 5.18. Superficie recorrida de las pruebas de 90 minutos | 96 |

| | | |
|-------|---|-----|
| 6.1. | Modelo taxi_holo_Laser | 98 |
| 6.2. | Modelo acera | 99 |
| 6.3. | Modelo carNoMotor | 100 |
| 6.4. | Mundo autopark.world en Gazebo | 103 |
| 6.5. | GUI del Autopark | 106 |
| 6.6. | Posición inicial del taxi en la práctica | 111 |
| 6.7. | Posición al parar en paralelo al coche de delante de la plaza libre | 111 |
| 6.8. | Posición al dar marcha atrás con giro a la derecha | 112 |
| 6.9. | Posición al dar marcha atrás con giro a la izquierda | 112 |
| 6.10. | Coche estacionado | 113 |
| 6.11. | Evaluador Automático del Autopark durante la ejecución de la práctica . . | 114 |
| 6.12. | Secuencia de la ejecución típica de la solución de referencia | 116 |
| 6.13. | Coche en el aparcamiento lejano | 117 |
| 6.14. | Aparcamiento con la plaza casi superada | 118 |

Índice de tablas

| | |
|---|----|
| 5.1. Resultados con diferentes tiempos de recorrido del perímetro | 92 |
| 5.2. Resultados Pruebas de 15, 45 y 90 minutos | 96 |

Acrónimos

GPP Gradient Path Planning.

STEM Science, Technology, Engineering and Math.

GPS Global Positioning System.

GUI Graphical User Interface.

SDF Simulation Description Format.

XML Extensible Markup Language.

API Application Programming Interface.

ICE Internet Communications Engine.

SLAM Simultaneous Localization and Mapping.

RRT Rapidly-exploring Random Tree.

ROS Robot Operating System.

OMPL Open Motion Planning Library.

Capítulo 1

Introducción

En este capítulo se definirá el contexto en el cual se sitúa este proyecto, y la motivación principal que ha llevado a su desarrollo. Se explicará de forma general qué es la robótica, así como su uso en la docencia. Además, se expondrá el uso de simuladores.

1.1. Robótica

La robótica es una rama de la ingeniería que emplea la informática para diseñar y desarrollar sistemas que permitan facilitar la vida del ser humano, e incluso sustituirle en determinadas tareas. Esta rama usa 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 realiza diversas máquinas que ejecutan diferentes comportamientos en función de su propósito. Estas máquinas se denominan “Robots”.

El término “Robot”, viene de la palabra checa robota, cuyo significado es “trabajo forzado”. Dicha palabra fue introducida por primera vez por el dramaturgo y autor checoslovaco Karel Čapek, en su obra de teatro R.U.R (Robots Universales de Rossum) en 1921. Con este libro surgió la palabra robótica, pero entonces era un término de ciencia ficción. En base a este término se puede decir que un robot es una máquina programada para moverse, manipular objetos y realizar trabajos, para lo cual debe interaccionar con el entorno que le rodea.

Unos años más tarde Isaac Asimov (1920 – 1992) introdujo conceptos acerca de la

CAPÍTULO 1. INTRODUCCIÓN

robótica. Isaac Asimov era un escritor y bioquímico estadounidense nacido en Rusia, el cual publicó el libro “Yo Robot” en 1950. Este libro contenía tres leyes de la robótica:

1. Un robot no puede lastimar a un ser humano o permanecer inactivo ante un daño que se le pueda hacer.
2. El robot debe obedecer al ser humano excepto si contradice la primera ley.
3. El robot debe proteger su existencia salvo que entre en conflicto con las leyes anteriores.

Con este libro, Isaac Asimov consiguió que la robótica se hiciera popular. Sin embargo, no fue hasta mediados de siglo cuando los robots empezaron a disponer de un sistema de control propio. Hasta entonces eran controlados por seres humanos.

En los años 50, surgió la idea de los robots programables, los cuales realizaban tareas repetitivas. Los robots se empleaban en entornos muy controlados y eran capaces de evitar obstáculos. En esta década se creó la primera empresa dedicada a la robótica, denominada Unimation (Universal Automation). Su primera creación fue empleada para la manipulación de material en una máquina de fundición.

En los años 60, se desarrolló el robot móvil Shakey. Este robot era capaz de evitar obstáculos y mover objetos dentro de un entorno altamente estructurado. En los 70, se desarrolló el robot JPL Rover en la Jet Propulsion Laboratory, cuyo fin era la exploración espacial. Entre los años 80 y 90 surgen diferentes arquitecturas para programar los robots, así como técnicas de navegación en entornos no estructurados y técnicas de creación de mapas.

En el año 2000, Honda lanza el robot Asimo. Este humanoide pretende ayudar a las personas que carecen de movilidad completa, así como animar a la juventud para estudiar ciencias y matemáticas.



Figura 1.1: Robot Asimo

El campo de la robótica es cada vez más popular y está en constante expansión. En la actualidad nos encontramos con 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.

Hoy en día no solamente vemos robots industriales, como en cadenas de envasado de alimentos o cadenas de producción, sino que los robots cobran cada vez más importancia en los entornos domésticos. Las aspiradoras robóticas (Roomba, Dyson, Xiaomi...) han llegado a los hogares con éxito para facilitar una tarea doméstica necesaria. Los vehículos cada vez incorporan más tecnología robótica, como el aparcamiento automático en cualquier gama del mercado, asistentes de conducción autónoma (autopiloto de Tesla), o prototipos de coches autónomos que han lanzado grandes empresas como Google o Apple. Podemos ver robots en el campo de la medicina (como Da Vinci) que permiten operar siendo teleoperados desde cualquier parte del mundo; en el ámbito militar permitiendo desactivar bombas o realizar misiones rescate; en los almacenes de Amazon; o la creciente presencia de drones o robots aéreos.

Los robots tienen que interactuar con situaciones reales de forma robusta. Esto requiere que posean cierta inteligencia, la cual está presente en su software. Este software posee capas (*drivers*, *middleware* y aplicaciones), y presenta unas características diferentes según su aplicación. En los últimos años, se han incorporado a los robots ordenadores personales, micros de bajo coste y sistemas operativos generalistas.

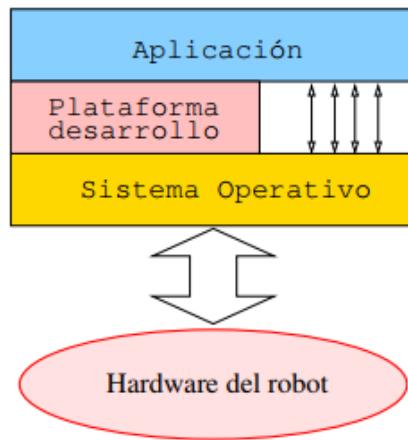


Figura 1.2: Esquema del funcionamiento de un robot

1.2. Software para robots

Muchos robots poseen autonomía, la cual proviene del desarrollo de sistemas complejos, aplicaciones e infraestructuras que les dotan de inteligencia autónoma. El desarrollo de software robótico es similar al desarrollo de software en otros ámbitos, donde se parte de ciertos requisitos y se modela un diseño que será creado. Hace años el desarrollo de software robótico se realizaba adoptando soluciones “ad hoc”, dotando a cada robot de un diseño específico, y con sensores y actuadores concretos. Esto suponía que no se podía aplicar el software desarrollado a otro robot, por lo que era necesario implementar de nuevo todo el software para un nuevo robot. En la actualidad, existen numerosas plataformas que permiten el desarrollo de aplicaciones robóticas de forma eficiente y genérica. Esto permite reutilizar gran parte de las aplicaciones creadas en otros robots, evitando el coste de realizar todo el proceso de nuevo.

Dotar al robot de cierta inteligencia conlleva desarrollar cierto software, el cual se suele programar apoyándose en herramientas, como los *middleware* robóticos, los simuladores robóticos, o las bibliotecas que facilitan algunos aspectos. A continuación, se exponen algunas de estas herramientas que se emplean en la actualidad.

1.2.1. Middlewares robóticos

En la actualidad existen numerosos *middlewares* robóticos, que permiten gestionar la complejidad y heterogeneidad del hardware y las aplicaciones, promover la integración de nuevas tecnologías, simplificar el diseño de software, y ocultar la complejidad de los sensores. Algunos de los *middlewares* robóticos más destacados son:

- Robot Operating System (ROS)¹ [1]: Es una plataforma de software libre para el desarrollo de software de robots, que provee servicios estándar de un sistema operativo como la abstracción del hardware, el control de dispositivos de bajo nivel, mecanismos de intercambio de mensajes entre procesos y un conjunto de herramientas utilizadas ampliamente en robótica. La librería está orientada para un sistema UNIX, aunque se está adaptando a otros sistemas operativos como Fedora, Mac OS X, Arch, Gentoo, OpenSUSE, Slackware, Debian o Microsoft Windows, considerados como “experimentales”.
- Orca² [1]: Es una plataforma de software libre para el desarrollo de aplicaciones robóticas. Está orientada a componentes, los cuales se pueden ejecutar de manera independiente o conjunta para crear aplicaciones más complejas. Orca permite reutilizar código, de manera que se pueden emplear componentes robóticos ya creados.
- Urbi [2]: Es una plataforma de software multiplataforma de código abierto en C++ utilizada para desarrollar aplicaciones para robótica y sistemas complejos. Urbi es compatible con ROS y se puede emplear en los sistemas operativos Linux, Windows y MAC OS X.

1.2.2. Simuladores robóticos

El diseño de un robot es costoso y caro, lo que implica que muchos componentes necesarios para la construcción de los robots solamente estén disponibles para centros de investigación y corporaciones. Cuando se emplea un robot puede que el código desarrollado falle al probarlo, pudiendo incluso romperse algún robot.

¹<http://www.ros.org/>

²<http://orca-robotics.sourceforge.net/>

Hoy en día existen numerosos simuladores robóticos, lo que permite a cualquier persona crear, programar y probar infinidad de robots de forma segura y económica. Algunos de los simuladores más empleados son:

- Gazebo³: Es un simulador 3D de código abierto distribuido bajo licencia Apache 2.0. Este simulador se ha utilizado en ámbitos de investigación en robótica e Inteligencia Artificial. Es destacado por sus motores de físicas, motor de renderizado avanzado, soporte para *plugins*, un amplio repertorio de robots comerciales, y una extensa gama de sensores y actuadores. Es fácil de integrar con ROS.
- Stage⁴: Simula robots móviles en el plano bidimensional y proporciona diversos tipos de sensores y actuadores. Su finalidad es ayudar a la investigación de sistemas autónomos de múltiples agente, para lo cual proporciona gran cantidad de dispositivos simultáneamente.
- Orocó⁵ [3] [4]: Es un proyecto de software libre para el control de robots y máquinas. Soporta 4 bibliotecas C++: Real-Time Toolkit, Kinematics and Dynamics Library, Bayesian Filtering Library y Orocó Component Library. Está orientado a componentes, permitiendo añadir funcionalidades de forma sencilla y sin recomilar todo el código. Incluye paquetes complementarios tales como Filtros de Bayes, Librerías de control Dinámico y Cinemático o Visión.
- Webots⁶ [5]: Es un simulador avanzado de robótica, que permite definir modelos propios, definir la física, escribir controladores para los robots y hacer simulaciones a gran velocidad. Se puede emplear en los sistemas operativos Linux, Windows y MacOS. Los lenguajes de programación que se pueden emplear son C++, C y Java.

1.2.3. Bibliotecas

En el desarrollo del software robótico es conveniente emplear bibliotecas que ya resuelven funcionalidades como el reconocimiento de gestos, procesamiento de imágenes, o la estimación de posición. En la actualidad existen diferentes bibliotecas que se emplean en robótica, por ejemplo:

³<http://gazebosim.org/>

⁴<http://playerstage.sourceforge.net/doc/stage-svn/>

⁵<http://www.orocos.org/taxonomy/term/18>

⁶<https://www.cyberbotics.com/>

- OpenCV ⁷: Está dirigida a la visión por computador en tiempo real. Entre las áreas de aplicación de esta biblioteca destacan: segmentación y reconocimiento de objetos, reconocimiento de gestos, seguimiento del movimiento, estructura del movimiento, y robots móviles.
- PCL ⁸ [6] [7]: Es una biblioteca de código abierto para el procesamiento de imágenes 2D/3D. Incluye numerosos algoritmos para manejar nubes de puntos en N dimensiones, y procesamiento tridimensional de las mismas. Se emplea en el manejo de información en sensores RGBD como Kinect.
- AForge.NET [8]: Es un framework C# de código abierto diseñado para desarrolladores e investigadores en los campos de Visión por Computadora e Inteligencia Artificial. Sus áreas de aplicación son: procesamiento de imágenes, redes neuronales, algoritmos genéticos, lógica difusa, aprendizaje de máquinas, robótica, etc.

1.3. Docencia en robótica

El propósito principal de este proyecto es la extensión de un entorno docente compuesto de varias prácticas para facilitar el aprendizaje de diferentes algoritmos de robótica. Estas prácticas utilizan técnicas robóticas próximas a las empleadas en la actualidad.

La robótica educativa es un medio de aprendizaje en el cual participan personas con motivación por el diseño y la construcción de creaciones propias. Esta disciplina se puede enseñar a estudiantes con muy diferentes niveles educativos. Ha crecido muy rápidamente en la última década y está en continuo desarrollo. Los robots están incorporándose en nuestra vida cotidiana, pasando de la industria a los hogares. El propósito de utilizar la robótica en la educación, a diferentes niveles de enseñanza, suele ir más allá de adquirir conocimiento en el campo de la robótica. Se pretende que el alumno sea capaz de aprender temas multidisciplinares (electrónica, informática, mecánica, física, etc), comprenda conceptos abstractos y complejos de ciencia y tecnología, y adquiriera competencias básicas que son necesarias en la sociedad de hoy día; como el aprendizaje colaborativo y la toma de decisión en equipo, entre otras.

⁷<http://opencv.org/>

⁸<http://pointclouds.org/>

CAPÍTULO 1. INTRODUCCIÓN

La robótica genera entornos colaborativos donde los participantes pueden practicar las habilidades del siglo XXI, conocidas como las 4C:

- Creatividad: Implica generar nuevas ideas mejorando las que ya existen. Es necesario ponerse en diferentes puntos de vista en cada circunstancia, tener la mente abierta y ser receptivo ante nuevas ideas y conceptos. Ayuda a la resolución de problemas de manera más eficaz.
- Pensamiento Crítico: Es imprescindible razonar con efectividad, claridad y precisión para desarrollar esta habilidad, reconociendo las conexiones que existen entre sistemas para resolver problemas y tomar decisiones. Se requiere practicar una correcta lógica de pensamientos para ver en cada situación los distintos puntos de vista.
- Colaboración: Se basa en la capacidad para trabajar de forma eficaz y respetuosa en equipos diversos. Esta habilidad implica comprometerse con los demás en la consecución de un objetivo común. Es importante asumir la responsabilidad del trabajo colaborativo, así como las aportaciones de cada miembro del equipo.
- Comunicación: Es la capacidad que permite intercambiar información de forma articulada, de dar y recibir retroalimentación de determinadas ideas con otras personas.

La robótica en la docencia intenta despertar el interés de los estudiantes transformando las asignaturas tradicionales en más atractivas e integradoras, ya que crea entornos de aprendizaje propicios que recrean los problemas del entorno que los rodea.

En el futuro, tener nociones básicas de esta disciplina será clave debido a que cada vez de forma más habitual se implantan robots en diferentes sectores laborales.

1.3.1. Docencia en secundaria

En los centros de enseñanza secundaria se imparte la robótica con frecuencia, ya que motiva a los alumnos. Esto les permite adquirir conocimientos tecnológicos, así como aprender conceptos básicos de ciencias, tecnología, ingeniería y matemáticas. Es decir, se implanta el enfoque de educación *Science, Technology, Engineering and Math (STEM)*.

CAPÍTULO 1. INTRODUCCIÓN

El concepto STEM se ha desarrollado con el fin de enseñar Ciencias y Tecnología de forma conjunta. Este método tiene dos características fundamentales:

- Enseñanza-aprendizaje de tecnología, ciencias, ingeniería y matemáticas de forma conjunta e integrada.
- Se da un enfoque a la tecnología de aprender conocimientos para resolver problemas tecnológicos reales.

La enseñanza de robótica en secundaria se realiza mediante plataformas físicas como los robots LEGO (Mindstorms, RCX, NXT, Evo, WeDo), placas Arduino, los kits de SolidWorks, etc. Se suelen enseñar conceptos básicos de sensores y actuadores empleando lenguajes gráficos como RCXcode, Scratch y mbot Blockly.

En los últimos años, diferentes universidades han planteado cursos orientados a promover el diseño de robots en estudiantes de secundaria. Un ejemplo es el departamento de Electrónica de la Universidad de Alcalá, que creó el proyecto “TuBot” con el fin de que los alumnos puedan construir su propio robot e incluso participar en una competición con el mismo.

Es importante destacar el curso JdeRobot-Kids⁹, que emplea mbot como robot móvil, una placa programable Arduino, y Python como lenguaje para introducir a los jóvenes los conceptos básicos, de tecnología, robótica y programación. De esta forma los alumnos podrán aprender de una forma más divertida conceptos interesantes de informática, electrónica y mecánica. El curso es fundamentalmente práctico, ya que la mejor manera de aprender es creando.

1.3.2. Docencia en la universidad

En la docencia universitaria se imparten clases de robótica en los Grados y los Postgrados, en concreto en escuelas de ingeniería. La enseñanza de robótica o materias similares como la inteligencia artificial, la visión por computador o el aprendizaje automático, se pueden impartir en el ámbito industrial, pero también en el ámbito informático.

⁹<http://jderobot.org/JdeRobot-kids>

CAPÍTULO 1. INTRODUCCIÓN

En España, podemos ver la robótica integrada en el “Grado en Ingeniería Robótica” de la Universidad de Alicante, donde nos encontramos con asignaturas como “Iniciación a la ingeniería robótica”, “Mecanismos y modelado de robots”, “Programación de robots”, o “Control de robots”. La enseñanza de robótica la podemos encontrar en los Grados de “Electrónica industrial y automática” que encontramos en numerosas universidades. Cabe destacar la universidad Carlos III de Madrid, donde se pueden estudiar materias como “Robótica Industrial” o “Robótica”. En diversas universidades se puede estudiar el Grado “Ingeniería Electrónica, Robótica y Mecatrónica”. Las universidades de Málaga y Sevilla cuentan con esta titulación, en la cual se imparten asignaturas como “Fundamentos de Robótica”, “Laboratorio de Robótica”, “Robótica y Automatización”, o “Ampliación de Robótica”.

En los estudios de Postgrado se imparten más asignaturas de robótica, 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” en diferentes universidades. Estos estudios de Postgrado los podemos encontrar en universidades como la Universidad Rey Juan Carlos, la Universidad Carlos III de Madrid, la Universidad del País Vasco, o la Universidad Politécnica de Cataluña.

En el ámbito internacional, 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. Se pueden destacar universidades especializadas en robótica como el MIT, Stanford, Georgia Institute of Technology, etc.

La asignatura de robótica habitualmente se imparte de forma práctica facilitando el aprendizaje de contenidos teóricos al alumno. Es común el uso de plataformas para la programación de robots. Algunas de las plataformas más destacadas son ROS¹⁰, o MATLAB con el paquete Simulink¹¹.

¹⁰<http://www.ros.org/>

¹¹<https://es.mathworks.com/products/simulink.html>

Cabe destacar The ConstructSim, que es útil para realizar simulaciones en la web. Esta plataforma web en la nube permite emplear una gran lista de simuladores por medio de un navegador web. De esta forma los usuarios no tienen que instalar nada, ni siquiera en sus navegadores.

1.4. Entorno Docente JdeRobot-Academy

La Universidad Rey Juan Carlos cuenta con el *middleware* robótico JdeRobot, que tiene asociado un entorno académico conocido como JdeRobot-Academy. 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, la Universidad ofrece cursos de introducción a la robótica y los drones, empleando dicha plataforma.

Los ejes principales de JdeRobot-Academy son: (a) lenguaje Python (por su sencillez y potencia), (b) simulador Gazebo (permite aprender robótica aunque no se disponga de robots reales, y permite tener un repertorio de robots variados —drones, formula1, brazos, aspiradoras, etc.— con los que enfocar aspectos muy variados de la robótica) y (c) foco en el algoritmo en vez de en el *middleware* ocultando los detalles de la infraestructura.

El entorno cuenta con un conjunto de prácticas. En cada una se plantea un problema robótico que tiene que resolver el alumno. En ellas se pueden distinguir varias capas, que influyen en el desarrollo del diseño de las prácticas. La capa de nivel más bajo, que será la primera que se crea, es la creación de la infraestructura de la práctica, lo que supone la creación de los modelos necesarios, los *plugins* que emplearán estos modelos, y el entorno simulado donde navegará el robot.

Por cada una de las prácticas se incluye un componente académico que le permitirá al alumno realizar la solución de las prácticas. Cada uno de estos componentes proporciona una interfaz gráfica (GUI) específica para esa práctica y código auxiliar de ayuda a la resolución de las prácticas. También suele incluir un evaluador automático, que permite llevar a cabo la corrección de las prácticas. En las prácticas se harán las conexiones necesarias del software del alumno con los sensores y actuadores que se empleen. El

propósito de los componentes académicos es permitir la abstracción por parte de los alumnos de los elementos complejos que no son parte de la resolución de las prácticas. De esta forma, el alumno lo único que tendrá que hacer es programar la solución de cada algoritmo que se le propone.

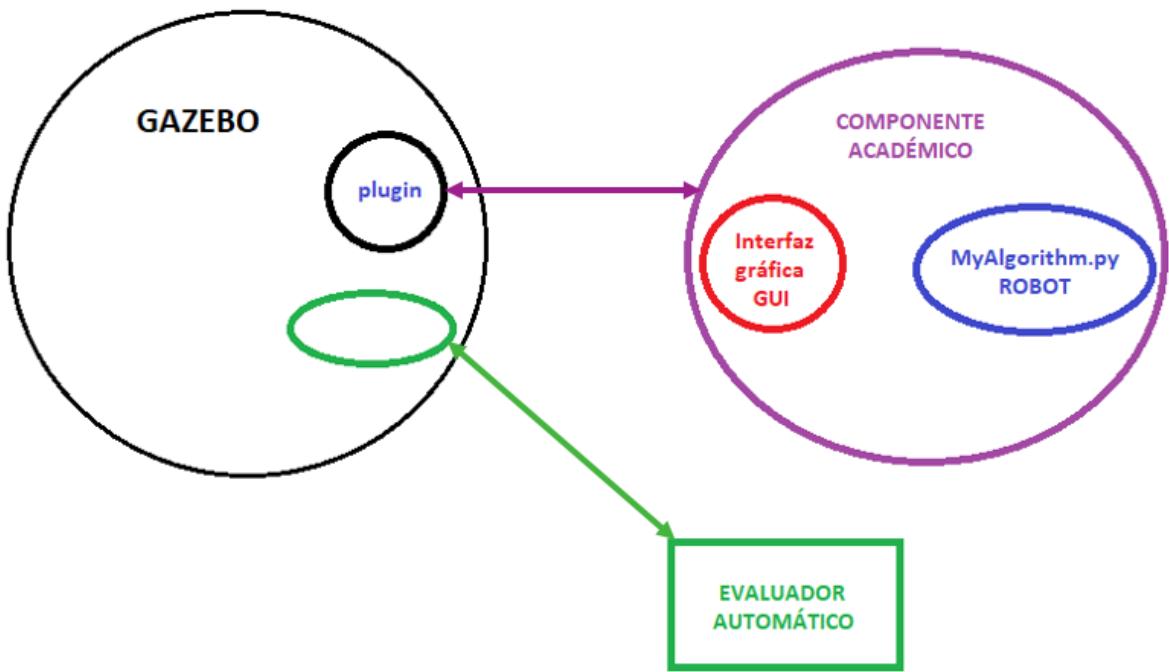


Figura 1.3: Estructura

Se han creado los componentes académicos siguiendo una arquitectura software que permite facilitar el desarrollo de las prácticas a los alumnos, los cuales únicamente deberán realizar la solución, ya sea el pilotaje en función de los datos que proporcionan los sensores o la realización de la planificación y el pilotaje. Los componentes cargan el código escrito por el alumno en el fichero *MyAlgorithm.py* (donde se lleva a cabo la resolución), mostrando en la interfaz las pruebas o soluciones que realicen los alumnos. En la Figura 1.3 podemos ver la estructura que tiene cada una de las prácticas.

Las prácticas se pueden realizar sobre robots simulados o reales, aunque lo más habitual es emplear robots simulados. Se apoyan en el simulador Gazebo, y se usa como lenguaje de programación Python. El principal sistema operativo para emplear esta plataforma es Linux. Sin embargo, se ha utilizado la interfaz web Gazebo para poder lanzarlo en Windows y MacOS mediante el empleo de Dockers.

A continuación, se presentan algunas de las prácticas ya desarrolladas en el entorno docente JdeRobot-Academy:

- Fórmula 1¹²: sigue líneas. En esta práctica el alumno debe programar el comportamiento de un coche Fórmula 1 para que realice un control PID siguiendo la línea roja pintada en el circuito de carreras. Para ello el robot posee sensores de posición y una cámara. La interfaz de movimiento es simple puesto que admite órdenes de velocidad lineal o velocidad de giro.

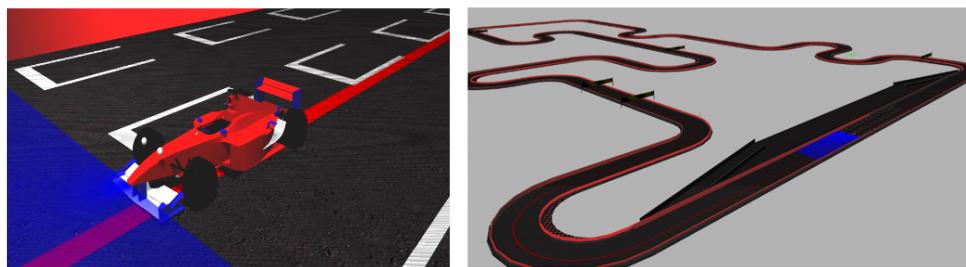


Figura 1.4: Fórmula 1 y circuito de Gazebo

- Visión¹³: reconstrucción 3D. En esta práctica el alumno debe lograr que un robot Pioneer reproduzca en 3D los elementos que están en frente del mismo. El robot cuenta con un par de cámaras estéreo. Para abordar el problema adecuadamente el alumno deberá programar un algoritmo de reconstrucción 3D clásico: detección de puntos de interés en el par de imágenes, emparejamiento de píxeles homólogos entre ambas, y triangulación espacial para calcular el punto tridimensional que origina cada pareja de píxeles homólogos. Esta práctica aborda técnicas de procesado de imagen y percepción visual.

¹²<https://www.youtube.com/watch?v=eNuSQN9egpA>

¹³<http://jderobot.org/store/jmplaza/uploads/teaching/teachingrobotics-reconstruccion3D.mp4>

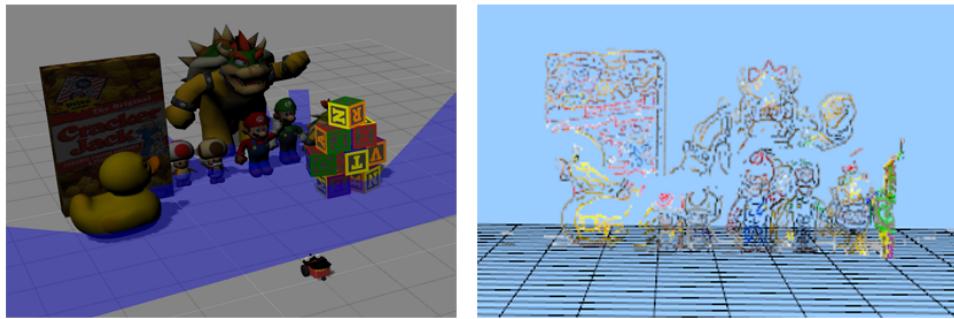


Figura 1.5: Pioneer en el mundo de Gazebo y reconstrucción 3D

- Control visual en drones¹⁴: sigue a la tortuga. En esta práctica el alumno debe conseguir que un drone siga a un robot Turtlebot que se desplaza por el suelo. Para lograr este objetivo el alumno deberá realizar los filtros de percepción visual necesarios para que el drone identifique al Turtlebot, así como programar el movimiento del drone para que siga al robot de forma adecuada. Esta práctica permite enseñar técnicas de percepción visual, de control reactivo, control basado en casos y de controladores PID.

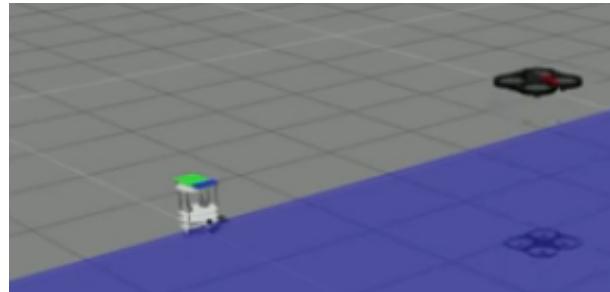


Figura 1.6: Drone en el mundo de Gazebo

El objetivo general de este proyecto es ampliar las posibilidades de esta plataforma docente, creando nuevas prácticas y mejorando alguna ya existente. En los próximos capítulos abordaremos todos los elementos necesarios para conseguir este objetivo. En el Capítulo 2 cubriremos los objetivos que se han marcado en el proyecto, así como los requisitos para cumplirlos y la metodología empleada. En el Capítulo 3 se expondrán la infraestructura utilizada durante el trabajo. En los Capítulos 4, 5 y 6, se abordarán las

¹⁴https://www.youtube.com/watch?time_continue=96&v=BoDchf_6yMQ

CAPÍTULO 1. INTRODUCCIÓN

prácticas que se han mejorado o creado. Por último, en el Capítulo 7 se expondrán las conclusiones obtenidas al realizar el proyecto, así como las posibles líneas futuras a seguir.

Capítulo 2

Objetivos

Una vez explicado el contexto de este proyecto, se describirán en este capítulo los objetivos, los requisitos y la metodología que se han empleado.

2.1. Objetivos

El propósito principal de este proyecto es la creación o mejora de tres prácticas para el entorno docente JdeRobot-Academy. En concreto mejorar una práctica de navegación global, crear una práctica de una aspiradora robótica y, por último, crear una práctica acerca del aparcamiento de coches autónomos. Siguiendo el diseño de JdeRobot-Academy, para cada práctica hay que desarrollar 4 ingredientes:

- Enunciado e infraestructura en el simulador
- Componente académico que incluye GUI y código auxiliar
- Solución de referencia
- Evaluador automático

En cada una de estas prácticas se elaborará toda la infraestructura que se comunica con el Simulador Gazebo, donde el alumno podrá ver el resultado de la ejecución de su algoritmo. Se ofrecerá, en cada práctica, un fichero *MyAlgorithm.py* donde el alumno programará su solución.

CAPÍTULO 2. OBJETIVOS

Lo que difiere en estas tres prácticas es el escenario de cada una de ellas, los diferentes elementos que se mostrarán en el interfaz gráfico, los elementos que tendrá en cuenta el evaluador automático a la hora de calcular la nota de cada alumno, y por último el algoritmo concreto de la solución. Cada una de ellas es un subobjetivo de este proyecto.

Primero, en la práctica “TeleTaxi” el objetivo es que el alumno aprenda técnicas de navegación global, en concreto, la técnica *Gradient Path Planning*. Esta práctica no es totalmente original, sino que existía una versión previa de la infraestructura y del componente académico, pero presentaban ciertos problemas. Por ello, en este proyecto se va a mejorar tanto la infraestructura como el componente académico, así como se va a desarrollar una nueva solución de referencia y un evaluador automático que nos permita calificarla.

Segundo, en la práctica “Aspiradora Autónoma” el principal objetivo es que el alumno sea capaz de proporcionar una solución para la limpieza de una casa sin autolocalización. En concreto, este algoritmo de referencia se basará en el que ejecutan los modelos de la serie 500 de Roomba de iRobot. Se realizará una solución que limpia en función de tres modos de navegación: giro en espiral, seguimiento de paredes y cruce de habitación. La aspiradora cuenta con diferentes sensores y actuadores, entre los que se encuentran un sensor láser, un sensor bumper, y actuadores que permiten dar órdenes a la aspiradora de velocidad de tracción y velocidad de giro. En este caso la práctica se ha realizado desde cero, pues no existía una versión previa. Con ella los alumnos se familiarizarán con un problema robótico real, cotidiano y cuya solución ya se está comercializando.

En último lugar, en la práctica “Aparcamiento Automático” el propósito es la realización de una solución que sea capaz de aparcar un coche de forma autónoma. El coche está dotado de tres sensores láser (que se encuentran en la parte frontal, en la parte trasera y en el lateral derecho) para medir distancias a los coches y encontrar aparcamiento. La solución propuesta es una solución “ad hoc” basada en las medidas sensoriales que se obtienen de los láseres. La solución permite al vehículo encontrar una plaza de aparcamiento libre y realizar la maniobra de aparcamiento. Esta práctica, también realizada de cero, nos permitirá acercarnos a sistemas robóticos que ya existen en el mercado (Volkswagen Tiguan, etc) y con ello aprender cómo funcionan.

2.2. Requisitos

El desarrollo del proyecto estará guiado por los subobjetivos mencionados anteriormente y deberá ajustarse a los requisitos de partida del proyecto, los cuales hacen que la solución esté condicionada. Estos requisitos son:

1. Todas las simulaciones se realizarán en el simulador Gazebo, en concreto en la versión 7. Los modelos de robots que se emplearán serán creados. En este caso se utilizarán dos taxis (los cuales tienen diferentes sensores) y el modelo Roomba.
2. Se hará uso del *middleware* robótico JdeRobot en su versión 5.5.2, que se explicará con detalle en el Capítulo 3. El uso de esta plataforma simplifica el desarrollo del comportamiento del robot.
3. El sistema operativo que se empleará para este proyecto será Ubuntu 16.04.
4. El lenguaje de desarrollo empleado para crear los *plugins* necesarios será C++. Sin embargo, en el resto de componentes se utilizará el lenguaje Python 2. Por compatibilidad con JdeRobot-5.5.2 y de éste con el *middleware* ROS Kinetic no se ha usado Python-3.X, sino que se sigue Python-2.7.
5. Las soluciones han de ser vivaces. Los algoritmos propuestos no pueden detenerse mucho tiempo a pensar cuál será el próximo movimiento del robot, porque han de reaccionar rápido, en tiempo real y con movimientos suaves.

2.3. Metodología

En el desarrollo del proyecto se ha seguido una metodología iterativa, donde cada iteración se compone de varias fases: determinar objetivos, planificación, diseño e implementación, análisis de riesgos, así como reuniones periódicas con el tutor.

Se ha optado por seguir el modelo de desarrollo en espiral, creado por Barry Boehm [9] [10]. Este modelo se adapta perfectamente a este tipo de proyectos, ya que permite separar el comportamiento final en varias subtareas más sencillas para después juntarlas. Este modelo permite una gran flexibilidad ante cambios en los requisitos, algo bastante común.

CAPÍTULO 2. OBJETIVOS

Este modelo de ciclo de vida permite ir obteniendo prototipos funcionales, a la vez que se realiza el desarrollo del producto de forma incremental. El modelo consta de iteraciones, que se pueden llamar ciclos. En cada ciclo existen cuatro fases bien diferenciadas:

- Determinar objetivos: Se definen los objetivos específicos que deben cumplirse para que el ciclo actual pueda considerarse finalizado en base a los objetivos finales. Conforme vayan sucediéndose más iteraciones, los objetivos serán más complejos.
- Análisis del riesgo: Se efectúa un análisis detallado para cada uno de los riesgos que pueda tener el objetivo fijado en la fase anterior. Se definen los pasos a seguir para minimizar los riesgos y después del análisis se planean estrategias alternativas.
- Desarrollar y probar: Se desarrolla el producto o las partes del producto que se han acordado en las fases anteriores. Además, se llevarán a cabo las pruebas oportunas que permitan asegurar la calidad de la implementación, y que pueda seguir sirviendo en iteraciones futuras.
- Planificación: Se revisan los resultados obtenidos mediante las pruebas de la fase anterior, y es donde se planifica la iteración siguiente teniendo en cuenta los posibles errores que se han cometido.

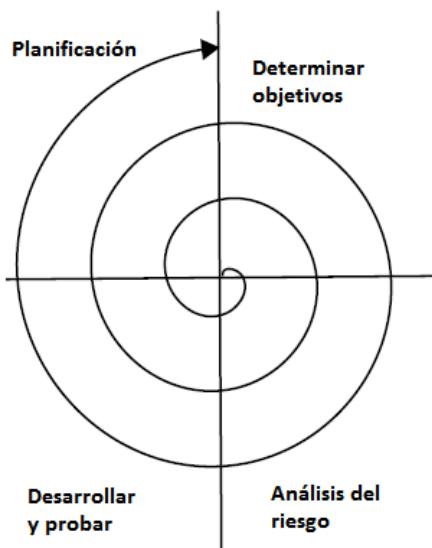


Figura 2.1: Modelo en espiral

Para llevar a cabo esta metodología se han mantenido reuniones semanales con el tutor. En estas reuniones se analizaban los resultados de cada iteración, y en función de los resultados se fijaban nuevos objetivos y se planteaban posibles vías para resolverlos. El código que se ha ido desarrollando semanalmente se ha subido al repositorio propio público de Github¹, que emplea el sistema de control de versiones. Además, las tareas realizadas se han ido mostrando semanalmente mediante explicaciones, vídeos o imágenes en la bitácora de la página de JdeRobot².

El resultado global de este TFG, las tres prácticas creadas, han sido integradas en el repositorio oficial de JdeRobot-Academy. Y están disponibles como software libre.

2.4. Plan de trabajo

Las etapas temporales en las que se ha dividido el proyecto, que además se corresponden con el modelo en espiral, son:

- Familiarización con el entorno JdeRobot y OpenCV. En esta etapa se ha descargado e instalado la plataforma JdeRobot, el entorno docente JdeRobot-Academy, y todo el software necesario para el desarrollo del proyecto. En esta fase se engloba el aprendizaje del uso de Github, para el control de versiones, y el aprendizaje básico de la librería OpenCV. Para cerrar esta fase se han realizado algunas soluciones de prácticas existentes del entorno JdeRobot-Academy.
- Familiarización con el simulador Gazebo y sus *plugins*. En esta etapa se ha estudiado código de la plataforma JdeRobot, así como el material disponible en Gazebo en la página oficial³. Además, se han realizado pruebas creando mundos simples en Gazebo mediante modelos ya disponibles. También se ha realizado un estudio de los *plugins* creados en JdeRobot (compilación e instalación), aprendizaje básico de C++, y desarrollo de algún *plugin* necesario para el desarrollo de las prácticas.
- Desarrollo de la práctica “TeleTaxi”: Desarrollo del enunciado, la infraestructura en gazebo, el componente académico, la solución de referencia y el evaluador automático.

¹<https://github.com/RoboticsURJC-students/2016-tfg-vanessa-fernandez>

²<http://jderobot.org/Vmartinezf-tfg>

³<http://gazebosim.org/>

CAPÍTULO 2. OBJETIVOS

- Desarrollo de la práctica “Aspiradora Autónoma”: Desarrollo del enunciado, la infraestructura en gazebo, el componente académico, la solución de referencia y el evaluador automático.
- Desarrollo de la práctica “Aparcamiento automático”: Desarrollo del enunciado, la infraestructura en gazebo, el componente académico, la solución de referencia y el evaluador automático.

Capítulo 3

Infraestructura

En este capítulo se explican los principales ingredientes software en los que nos hemos apoyado para desarrollar el trabajo. Tales como el simulador Gazebo (con el cual podemos simular las acciones que realizaría un robot en un mundo determinado), el entorno JdeRobot, la librería de OpenCV (empleada en todo lo relacionado con el tratamiento de imagen), PyQt (para el desarrollo de la interfaz gráfica) y Python como lenguaje de programación.

3.1. Simulador Gazebo

Gazebo es un simulador usado en robótica que permite emular diversos escenarios tridimensionales donde probar nuestro software. A la hora de realizar el software robótico es necesario hacer pruebas, las cuales saldrían muy caras si se probarán en robots reales (podría no funcionar correctamente y que nuestro robot se rompiera). Por esta razón es muy útil el empleo de simuladores, pues podemos realizar las pruebas que queramos sin peligro de que nuestro robot salga averiado. Con los simuladores se pueden diseñar robots y escenarios realistas donde ejecutar algoritmos de percepción y control.

CAPÍTULO 3. INFRAESTRUCTURA

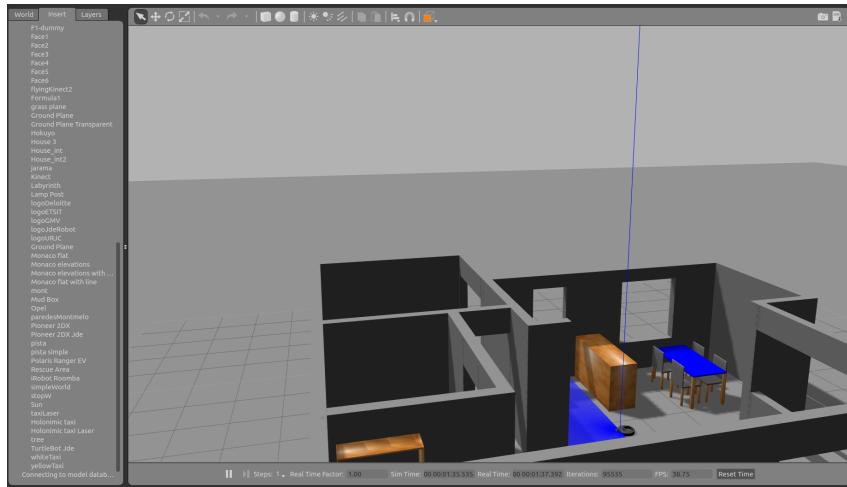


Figura 3.1: Simulador Gazebo

Gazebo es un programa de código abierto distribuido bajo licencia de Apache 2.0. Es uno de los ejes del entorno docente de JdeRobot-Academy. Se emplea en el desarrollo de aplicaciones robóticas y en inteligencia artificial. Es capaz de simular robots, objetos y sensores en entornos complejos de interior y exterior. Tiene gráficos de gran calidad y un robusto motor de físicas (masa del robot, rozamiento, inercia, amortiguamiento, etc.). Fue elegido para realizar el DARPA Robotics Challenge (2012–2015) y está mantenido por la Fundación Robótica de Código Abierto (OSRF).

En este trabajo se emplea la versión 7 de Gazebo¹. Gracias a Gazebo se pueden incluir texturas, luces y sombras en los escenarios, así como simular la física como por ejemplo choques, empujes, gravedad, etc. Además, incluye diversos sensores, como pueden ser cámaras y láseres, los cuales podrán ser incorporados en los robots que empleemos. Todo ello hace que sea una herramienta muy potente y de gran ayuda en robótica.

Los mundos simulados con Gazebo son mundos 3D, que se cargan a partir de ficheros con extensión “.world”. Son ficheros Extensible Markup Language (XML) definidos en el lenguaje Simulation Description Format (SDF). Este lenguaje contiene una descripción completa de todos los elementos que tiene el mundo y los robots, incluyendo:

- Escena: Luz ambiente, propiedades del cielo, sombras, etc.

¹<http://gazebosim.org/blog/gazebo7>

- 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.
- Luz: Los puntos y origen de la luz.

Las etiquetas empleadas en el fichero para representar estos elementos son: Scene, World, Model, Physics, Plugin, y Light.

Los modelos de robots que se emplean en la simulación son creados mediante algún programa de modelado 3D (Blender, Sketchup...). Estos robots simulados necesitan ser dotados de inteligencia para lo cual se emplean los *plugins*. Estos *plugins* pueden dotar al robot de inteligencia u ofrecer la información de sus sensores a aplicaciones externas y recibir de éstas comandos para los actuadores de los robots.

3.2. Entorno JdeRobot

JdeRobot² es un *middleware* de software libre 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 GPLv3³.

Está desarrollado en C y C++, aunque contiene componentes desarrollados en lenguajes como Python y JavaScript. El entorno que ofrece está basado en componentes, los cuales se ejecutan como procesos. Dichos componentes interoperan entre sí a través del *middleware* de comunicaciones ICE o de ROS messages. Tanto ICE como ROS-messages permiten la interoperación entre los componentes incluso estando desarrollados en diferentes lenguajes.

²http://jderobot.org/Main_Page

³<https://www.gnu.org/licenses/quick-guide-gplv3.html>

CAPÍTULO 3. INFRAESTRUCTURA

Es capaz de llevar a cabo diferentes tareas en tiempo real de forma sencilla. Cada componente *driver* está asociado a un dispositivo hardware del robot, un sensor o actuador e incluye funciones para poder emplearlo. Esto simplifica el acceso a los diferentes componentes hardware, ya que con una simple función se puede acceder a ellos.

Las aplicaciones constan de uno o varios componentes. Los que interactúan directamente con los sensores y actuadores del robot se llaman *drivers*, que son los encargados de controlar que los robots reciben órdenes a través de interfaces ICE o ROS messages. Otros llevan en su código las funciones perceptivas, procesamiento de señales o la lógica de control e inteligencia del robot. En la siguiente imagen se puede ver un ejemplo de esta comunicación con un AR Drone empleando interfaces ICE. La misma lógica de comportamiento se puede conectar al *driver* del drone real o al *driver* del drone simulado, basta con cambiar la configuración.

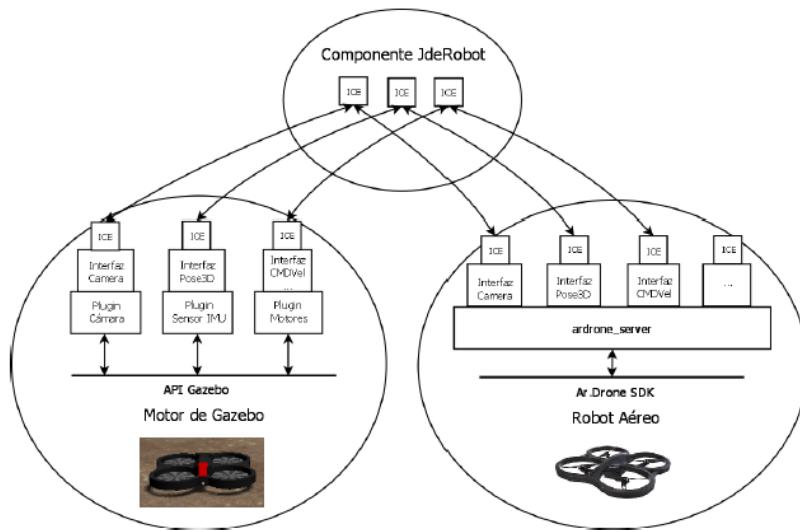


Figura 3.2: Ejemplo de componentes JdeRobot

Esta plataforma soporta gran variedad de dispositivos como el cuadricóptero AR Drone de Parrot, el robot Pioneer de MobileRobotics Inc., 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. A parte de todo esto, tiene soporte para software externo como OpenCV, OpenGL, XForms, GTK, Player y GSL.

En el desarrollo de las prácticas de este TFG se empleará la versión 5.5.2 de JdeRobot, ya que es la última versión estable.

3.3. Python

Python⁴ es un lenguaje de programación fácil de aprender y de alto nivel. 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 en honor a la serie de televisión *Monty Python's Flying Circus*.

Python incluye orientación a objetos, manejo de excepciones, listas, diccionarios, etc. A pesar de todo lo que soporta, se creó con el objetivo de que fuera un lenguaje sencillo de entender, sin perder las funcionalidades que pueden ofrecer lenguajes complejos tales como C.

Actualmente Python es un lenguaje de código abierto administrado por Python Software Foundation. Incluye módulos que permiten la entrada y salida de ficheros, *sockets*, llamadas al sistema e incluso interfaces gráficas como Qt. Además, permite dividir el programa en módulos reutilizables y no es necesario compilarlo, pues es interpretado. Es uno de los ejes de JdeRobot-Academy.

La última versión ofrecida por Python Software Foundation es la 3.6.2 , pero en nuestro caso se empleará la 2.7.12 por compatibilidad con JdeRobot 5.5.2, que a su vez sigue en esa versión de Python para ser compatible con ROS Kinetic. El código en el que están escritos los componentes académicos y las soluciones es Python.

3.4. Biblioteca OpenCV

OpenCV⁵ es una librería de código abierto desarrollada inicialmente por Intel y publicada bajo licencia de BSD. Esta librería implementa gran variedad de herramientas

⁴<https://www.python.org/>

⁵<http://opencv.org/>

CAPÍTULO 3. INFRAESTRUCTURA

para la interpretación de la imagen. Sus siglas provienen de los términos anglosajones “Open Source Computer Vision Library”, y está orientada a 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, para el rastreo, análisis de la forma, análisis del movimiento, reconstrucción 3D, segmentación de objetos y reconocimiento. 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 más de la mitad de las funciones.

Fue diseñado para tener una alta eficiencia computacional. Está escrito en C y puede aprovechar las ventajas de los procesadores multinúcleo. La biblioteca de OpenCV contiene más de 500 funciones que abarcan muchas áreas de la visión artificial. También tiene una librería de aprendizaje automático (MLL, Machine Learning Library) destinada al reconocimiento y agrupación de patrones estadísticos.

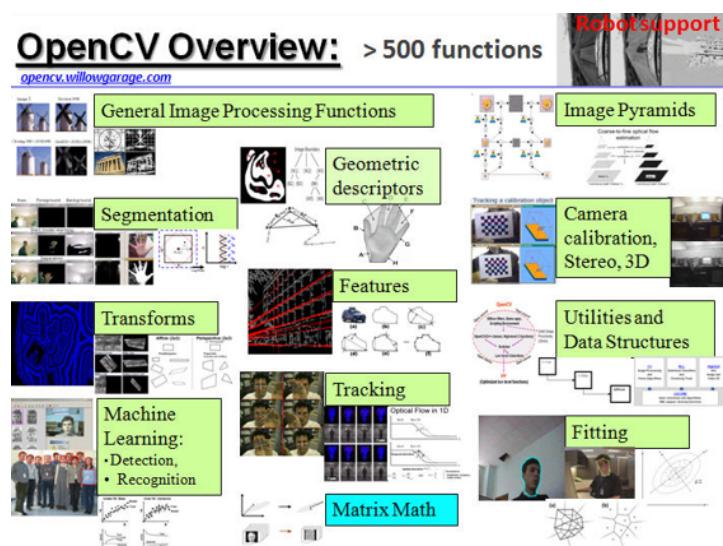


Figura 3.3: Funciones de OpenCV

OpenCV está compuesto por numerosas librerías con las cuales podemos manejar estructuras de datos, detectar bordes y esquinas, escalar o rotar imágenes, modificar el espacio de color de una imagen, realizar emparejamiento, detectar líneas y círculos, tratar objetos en 3D, crear ventanas y asociar eventos a dichas ventanas, etc. Incorpora funciones básicas para modelar el fondo, sustraer dicho fondo, generar imágenes de movimiento MHI (Motion History Images), etc. Además, incluye funciones para determinar dónde hubo movimiento y en qué dirección.

Desde su aparición OpenCV ha sido usado en numerosas aplicaciones. Hay una gran cantidad de empresas y centros de investigación que emplean estas técnicas como IBM, Microsoft, Intel, SONY, Siemens, Google, Stanford, MIT, CMU, Cambridge e INRIA.

En este trabajo se ha empleado la versión 3.2 de OpenCV en Python. Esta librería se empleará para realizar todo lo relacionado con el tratamiento de imágenes. Con ella se extraerán datos que puedan emplearse a la hora de tomar decisiones para que los robots funcionen correctamente.

3.5. PyQt

PyQt [11] [12] es un conjunto de enlaces Python para el conjunto de herramientas Qt, las cuales se emplean para el desarrollo de interfaces gráficas. Fue desarrollado por Riverbank Computing Ltd y es soportado por Windows, Linux, Mac OS/X, iOS y Android.

Qt es un entorno multiplataforma orientado a objetos desarrollado 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, pues permite emplear todas las funcionalidades ofrecidas por Qt con un lenguaje de programación tan sencillo como Python.

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.

La interfaz gráfica de los componentes académicos creados en las prácticas está escrita usando PyQt. Las clases de PyQt5 se dividen en ciertos módulos, tales como QtCore, QtGui, QtWidgets, QDom, QSql, etc. En las prácticas desarrolladas se ha hecho uso de los siguientes módulos:

- QtCore: contiene las funcionalidades principales que no tienen que ver con la GUI. 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 GUIs clásicas de escritorio.

Capítulo 4

Práctica: Navegación global de un TeleTaxi con GPP

Una vez explicado el contexto, los objetivos y las herramientas empleadas en este proyecto, en este capítulo se detallarán las mejoras de infraestructura y componente académico en la práctica de JdeRobot-Academy denominada “*TeleTaxi*”, así como el evaluador automático creado y la nueva solución de referencia realizada. Esta práctica ya formaba parte del entorno JdeRobot-Academy, pero se han realizado diferentes mejoras que se describirán a lo largo del capítulo.

4.1. Enunciado

El propósito de esta práctica es que un taxi autónomo sea capaz de navegar desde un punto de la ciudad a cualquier otro punto donde tiene que recoger o soltar a un cliente. El usuario humano podrá especificar un punto de destino al robot (taxi) picando con el ratón en algún lugar de la ciudad, cuyo mapa también se muestra en el interfaz gráfico del componente académico. El taxi dispondrá de un mapa de la ciudad y un sensor *Global Positioning System (GPS)* que le proporciona una estimación de su posición en la ciudad. Este vehículo posee un actuador de movimiento basado en velocidad lineal y velocidad de giro.

El alumno deberá programar el algoritmo de navegación global Gradient Path Planning (GPP) para obtener una planificación de la ruta que ha de seguir desde la posición actual del robot hasta la posición destino definida por el usuario en la interfaz gráfica (GUI). Una

vez calculada la ruta más corta entre los puntos mencionados, en el GUI se podrá visualizar la ruta en el mapa 2D, así como una imagen con el campo de gradiente calculado por el código del alumno (nos dará información acerca de la ruta en función de la distancia). Una vez calculada la ruta, el alumno deberá proporcionar un algoritmo de pilotaje para que el robot la siga. Este algoritmo deberá tener en cuenta el campo calculado mediante el GPP y la posición del taxi.

4.2. Infraestructura

En este punto se comentará el entorno creado para llevar a cabo la práctica. Comenzamos con una descripción del robot que se ha empleado para el desarrollo de la práctica, así como una descripción de los actuadores y sensores que posee este robot, y una explicación de la ciudad por la que se moverá nuestro robot.

4.2.1. Coche Taxi_Holo

El robot que se ha empleado en esta práctica es un nuevo modelo de robot creado en un programa de modelado 3D (como pueden ser Blender, SketchUp, etc). La versión anterior de “TeleTaxi” usaba el modelo *yellowTaxi*, el cual es un robot creado para poder moverse de forma autónoma o teledirigida por un escenario. El nuevo modelo *Taxi_Holo* posee sensores GPS que le permiten saber cuál es su posición en todo momento; así como motores que le permite moverse por el escenario de manera adecuada. No posee otros sensores como pueden ser cámaras o láseres.

El modelo *yellowTaxi* de la versión antigua de la práctica “TeleTaxi” es realista, tiene en cuenta las características propias de un automóvil, pero con el aspecto de un taxi. Este robot posee unas dimensiones grandes, ya que mide aproximadamente 5.75 metros de largo, 3 metros de ancho y posee una altura de 3 metros (Figura 4.1). Una característica importante de este modelo es que es holonómico ¹. Este coche se puede ver en la figura 4.1.

¹Holonómico quiere decir que puede girar en el sitio sin chocar con nada.



Figura 4.1: Modelo yellowTaxi

Este modelo tenía fallos en su estructura que hacían que apenas se pudiera desplazar en el mundo o no fuera capaz de rotar, aunque le enviáramos órdenes de velocidad de tracción y velocidad de rotación muy elevadas. Este inconveniente suponía que el taxi tardara mucho tiempo en llegar a un objetivo cercano o que fuera incapaz de girar suficiente en las curvas, lo que hacía que se chocara contra las paredes de los obstáculos. Por este motivo, se creó un nuevo modelo de taxi denominado *taxi_holo*.

El nuevo modelo de taxi (Figura 4.2), el cual ha sido creado por desarrolladores del proyecto JdeRobot, cuenta con sensores de GPS y motores que permiten el movimiento al igual que el modelo antiguo. Es capaz de moverse de forma fluida por el escenario. Posee unas dimensiones de menor tamaño, 4 metros de largo, 2 metros de ancho y una altura de 1.5 metros. Este taxi pesa 750 kg y es también holonómico como el anterior, pero resuelve de forma eficiente los problemas de movimiento que tenía el antiguo modelo.

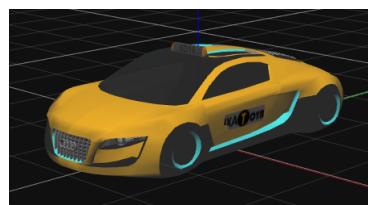


Figura 4.2: Modelo taxi_holo

Los sensores de posición (tipo GPS, odometría u otros) que incorpora el taxi son ampliamente utilizados, ya que son una gran fuente de información para los algoritmos en los que se apoya el pilotaje de nuestro robot. La odometría se emplea para estimar la posición (x , y , orientación) de un robot móvil en todo momento. La posición estimada es relativa a la localización inicial del robot. Por tanto, emplearemos estos sensores de

posición para estimar la posición del taxi en el mundo, y a partir de este dato estimar su velocidad. Los sensores de odometría estiman la posición de las ruedas izquierda y derecha en un intervalo de tiempo determinado. Al conocer las coordenadas de posición anteriores resulta más sencillo obtener la nueva posición de nuestro robot. La plataforma JdeRobot aísla de la complejidad que conllevan los sensores de odometría, facilitándonos una variable que contiene la posición (x, y, orientación) en el mundo.

En esta práctica se han empleado dos *plugins* asociados al taxi simulado:

- *holoCarPose3d*: Es el *plugin* que emplearán los componentes para poder obtener su posición en tiempo real. También se utiliza para cambiar mágicamente la posición del taxi en el mundo simulado.
- *holoCarMotors*: Es el *plugin* que permite dotar al taxi de velocidad, tanto velocidad de tracción como velocidad de rotación.

4.2.2. Modelo de ciudad cityLarge

El objetivo de esta práctica es que nuestro taxi sea capaz de navegar por una ciudad hasta un punto destino, por lo que tendremos que crear el entorno (ciudad) donde se moverá. Con este propósito se ha creado un modelo de ciudad llamado “*cityLarge*”. Este modelo fue creado con una herramienta de modelado 3D (Blender). El modelo *cityLarge* se corresponde con una ciudad de grandes dimensiones. En esta ciudad no veremos casas, semáforos, parques u otros elementos habituales en las ciudades reales, sino que se ha simplificado su creación para que sea rápido en la ejecución del simulador. Mundos con mucho detalle son más costosos computacionalmente de simular. Lo que podremos ver en esta ciudad son bloques que se corresponden con manzanas de edificios o carreteras. También podemos ver elementos propios de las carreteras como son rectas, curvas más simples o curvas pronunciadas, así como una zona amplia que se corresponde con una especie de plaza.

Este modelo se ha modificado respecto a la versión antigua, debido a que en dicha versión debajo de la ciudad había un palo de sujeción. Este elemento hacía que el coche adquiriera un poco de movimiento adicional al ejecutar la práctica. Por lo tanto, se ha

eliminado este palo de sujeción.

En la Figura 4.3 podemos ver la parte de debajo del modelo *cityLarge* para observar que había un palo de sujeción de la ciudad. En la Figura 4.4 se observa el nuevo modelo *cityLarge* sin dicho elemento de sujeción. Por último, en la Figura 4.5 tenemos un plano desde arriba de toda la ciudad.

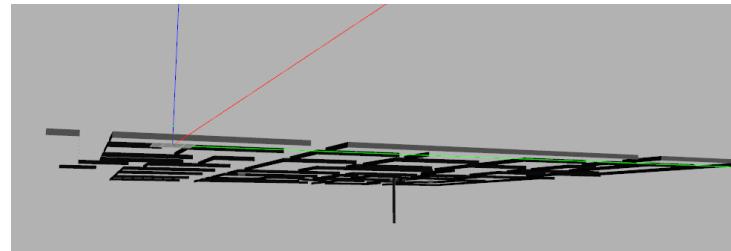


Figura 4.3: Modelo antiguo *cityLarge*

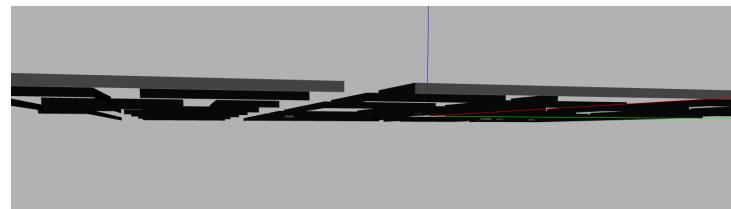


Figura 4.4: Modelo nuevo *cityLarge*

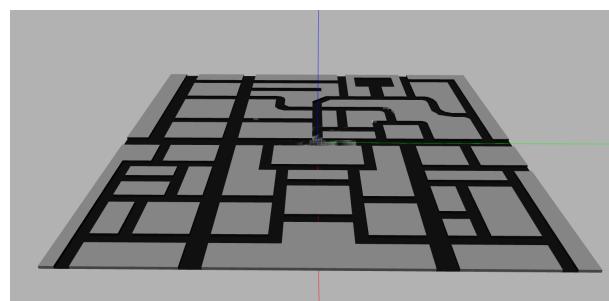


Figura 4.5: Modelo *cityLarge* desde arriba

4.2.3. Mundo de Gazebo

Los mundos que se simulan con Gazebo son mundos 3D. Estos mundos se cargan en ficheros con extensión .world, que no son más que ficheros XML definidos en el lenguaje

SDF. Este lenguaje contiene una descripción completa de todos los elementos que tiene el mundo y los robots.

Se ha creado un mundo en Gazebo (*cityLarge.world*) compuesto por el modelo de la ciudad (*cityLarge*) y el modelo del taxi (*taxi_holo*). El antiguo mundo en vez de utilizar el modelo *taxi_holo* empleaba el modelo *yellow_taxi*. El archivo *cityLarge.world* tiene el siguiente aspecto:

```
<?xml version="1.0" ?>
<sdf version="1.4">
  <world name="cityLarge">
    <!-- My city -->
    <include>
      <uri>model://cityLarge</uri>
    </include>
    <!-- My robots -->
    <include>
      <uri>model://taxi_holo</uri>
      <pose>0 0 0 0 0 0</pose>
    </include>
    <!-- A global light source -->
    <include>
      <uri>model://sun</uri>
    </include>
  </world>
</sdf>
```

4.3. Componente Académico

El componente académico para ayudar al alumno y alojar su código en esta práctica resuelve varias funcionalidades: (a) ofrece una interfaz gráfica al estudiante que le ayuda a depurar su código; (b) ofrece acceso a sensores y actuadores en forma de métodos simples (oculta el *middleware* de comunicaciones); (c) incluye código auxiliar que no es el foco del algoritmo pero que ayuda a programar la solución. Lo deja todo atado para que el

estudiante sólo tenga que incluir su código retocando el fichero *MyAlgorithm.py*.

Este componente ofrece al programador del algoritmo un Application Programming Interface (API) de sensores y actuadores. A continuación se puede ver el API concreto de esta práctica:

- *sensor.getRobotX()*: Permite obtener la posición del robot en el eje X.
- *sensor.getRobotY()*: Permite obtener la posición del robot en el eje Y.
- *sensor.getRobotTheta()*: Permite obtener la orientación del robot con respecto al mapa.
- *vel.setV()*: Para establecer la velocidad lineal.
- *vel.setW()*: Para establecer la velocidad de giro.

Es necesario escribir un archivo de configuración para que este componente pueda comunicarse con gazeboserver. En este fichero se indican los puertos de los *plugins* que usa el taxi. Este fichero (*teleTaxi.cfg*) en la práctica tiene el siguiente aspecto.

```
TeleTaxi.Motors.Proxy = Motors:default -h localhost -p 9999
TeleTaxi.Pose3D.Proxy = Pose3D:default -h localhost -p 9989
TeleTaxi.robot=Pioneer

TeleTaxi.Motors.maxV = 50
TeleTaxi.Motors.maxW = 20
```

Podemos ver que los motores emplean el Puerto 9999, mientras que la pose3d emplea el Puerto 9989. Además, se puede ver en este archivo que se indica al robot la velocidad máxima de tracción y de rotación.

Se ha dividido el trabajo de este componente académico en diferentes partes, por lo que emplearemos hilos de ejecución para llevar a cabo diferentes tareas simultáneamente. En esta práctica existen dos procesos diferenciados:

- Hilo de control: Este hilo es el encargado de actualizar los datos de los sensores y los actuadores a través de las interfaces ICE. El tiempo de refresco de este hilo es

muy importante, y debe ser un periodo de tiempo muy corto, ya que se encarga de establecer la velocidad y la dirección del robot en todo momento. Si este tiempo fuera muy grande, las decisiones que modifican la trayectoria del robot podrían ser incorrectas. Este hilo (*ThreadMotors*) se emplea para enviar órdenes a los motores y se actualiza cada 80 milisegundos.

- Hilo de la interfaz gráfica de usuario (GUI): Se encarga de ir actualizando la GUI. El intervalo de actualización de este hilo es muy importante. Es necesario que este intervalo sea pequeño, ya que tenemos que mostrar la posición del robot en el mapa en tiempo real. El hilo de ejecución de la GUI (*ThreadGUI*) se actualizará cada 50 ms.

4.3.1. Interfaz gráfica

La interfaz gráfica (GUI) de la práctica es una ayuda al alumno para realizar la solución a la práctica. Esta interfaz se realizará en PyQt5, dado que permite realizar interfaces con numerosos objetos gráficos (imágenes, botones, etc).

La GUI de la práctica (Figura 4.6) contiene una imagen del mapa del mundo de Gazebo a la izquierda. En esta imagen podemos seleccionar el destino al que deseamos que el taxi llegue. En ella se pinta continuamente la posición y orientación actuales del taxi mediante un triángulo azul. El pintado del triángulo se ha modificado puesto que al haber una desviación en la conversión del sistema de referencia del mundo a la imagen o al revés, se había hecho una traslación a la hora de pintar el triángulo que no era necesaria. Esta conversión ha sido eliminada. Además, al cambiar de modelo de taxi se vio que la orientación del taxi no era correcta porque el ángulo no estaba en grados como era de esperar. El ángulo que nos devuelve la orientación del taxi está en radianes. Por lo que se suponía que se debía pasar a grados para poder realizar el pintado del triángulo, pero la conversión era errónea, ya que se multiplicaba simplemente por 100. Esto ha sido modificado obteniendo los grados al multiplicar la orientación por 180 y dividir por π . A continuación, podemos ver dos imágenes, una con la orientación del triángulo incorrecta (Figura 4.7) y otra con la orientación correcta (Figura 4.8).

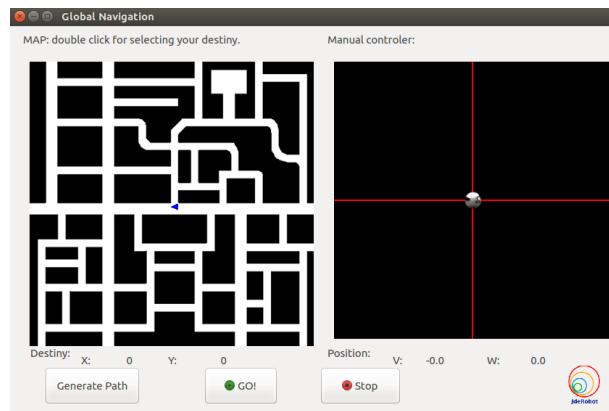


Figura 4.6: Interfaz gráfica (GUI) actual del GPP

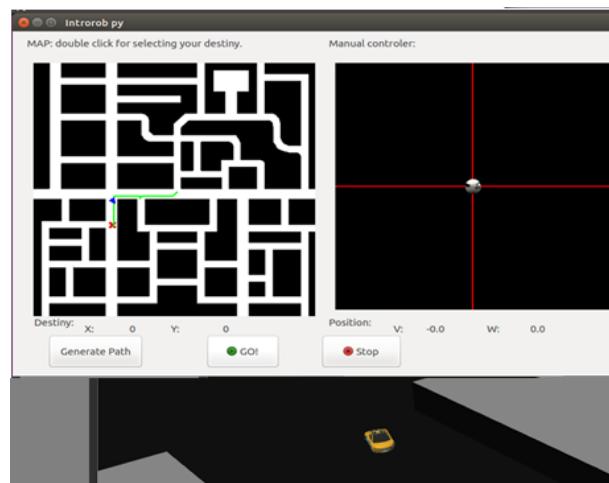


Figura 4.7: Interfaz gráfica con el triángulo que representa al taxi mal pintado

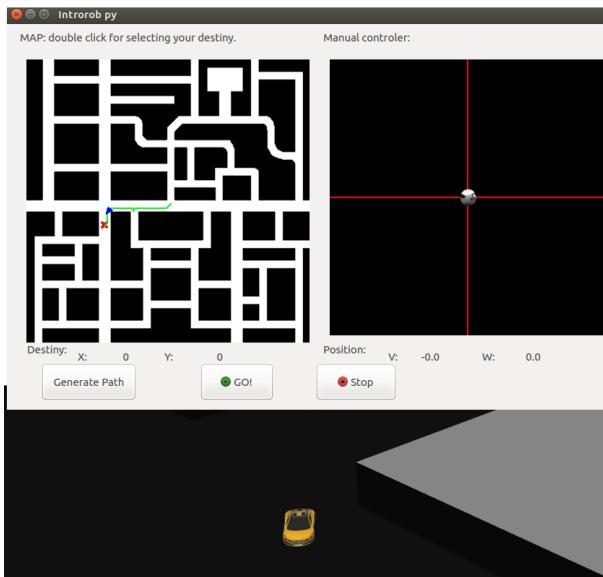


Figura 4.8: Interfaz gráfica con el triángulo que representa al taxi correctamente pintado

En la interfaz, además se muestra debajo del mapa del mundo la posición numérica exacta que tiene el taxi al iniciarse la práctica en el eje X y en el eje Y. Esta interfaz gráfica (Figura 4.6) además muestra un teleoperador a la derecha con el que se puede mover manualmente el taxi en el mundo de Gazebo si se desea. Por otra parte, debajo del teleoperador podemos ver la velocidad lineal y velocidad angular que tiene al teledirigir el taxi.

La GUI posee tres botones, de los cuales dos de ellos serán utilizados para poder ver cómo se lleva a cabo el algoritmo que ha programado el alumno. Cuando pulsamos el botón “*Generate Path*”, podemos ver el resultado del código que resuelve la parte de la planificación y cómo genera en la imagen del mapa una ruta. De ser pulsado el botón “*GO!*” se activará la parte de código que ejecuta el pilotaje del taxi, y podremos ver cómo nuestro taxi navega por las carreteras de la ciudad. El tercer botón sirve para cuando estamos empleando el teleoperador. Este botón nos permitirá parar el taxi si queremos.

Se ha modificado el tamaño del visor de la imagen que muestra la GUI, ya que antiguamente no se podía ver la parte inferior del mapa en la GUI, por lo que existían puntos de la carretera a los que no podíamos ir. En las siguientes imágenes se puede comprobar la diferencia que acabamos de mencionar entre la antigua GUI y la actual:

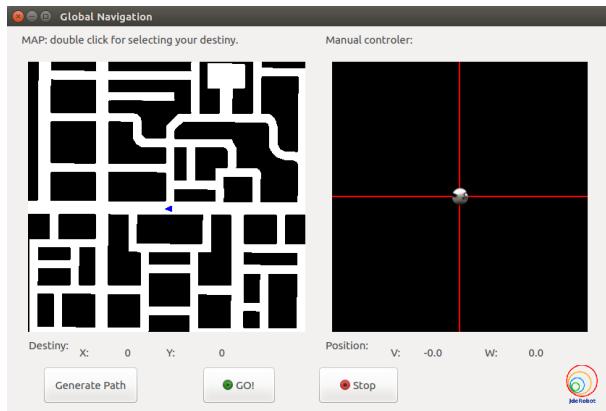


Figura 4.9: Interfaz gráfica (GUI) antigua del GPP

4.3.2. Código auxiliar: Clase Grid

Dentro del componente académico la clase *Grid* se creó en la anterior versión de “TeleTaxi”, pero se han modificado algunos aspectos en esta versión. Esta clase ofrece un mapa que ayuda al programador a resolver la práctica, permite manipular el mapa de ocupación, crear la rejilla que se emplea para el cálculo del campo y permite controlar la variable “destino de navegación”.

Este componente académico es el que se encarga de capturar información del mundo, crear una rejilla donde se almacenará el campo de la expansión y comunicarse con la interfaz gráfica. Este componente será lanzado al ejecutar la práctica y lanzar la GUI.

La clase (*Grid*) será instanciada en el programa principal (*globalNavigation.py*). Cuando es lanzado este componente, se crea una rejilla del tamaño de la imagen del mapa que tiene la interfaz gráfica (400 x 400 píxeles). Esta rejilla se crea para guardar información acerca del mundo. Será utilizada por el alumno para guardar los valores del campo del gradiente (será explicado en el punto 4.4) y ayudará a realizar el pilotaje del robot. Además, se inicializa otra rejilla, llamada *path* para almacenar la ruta más corta, la cual se pintará en verde sobre el mapa.

Al ejecutar la práctica también se inicializa la variable *map*. Esta variable se inicializa desde la interfaz gráfica y es una imagen binaria del mundo. Tiene representados en negro (valor 0) los píxeles que forman parte de los obstáculos, mientras que los píxeles que

forman parte de la carretera serán blancos (valor 255). La imagen será de tres canales, aunque para la solución de la práctica bastará con usar un canal. Este mapa ayudará al alumno a realizar la solución, ya que se puede extraer mucha información de la misma. Esta imagen del mapa ha sido cambiada en la versión actual, debido a que la anterior estaba algo torcida y faltaba una parte del mapa en la parte inferior. Se puede ver en la Figura 4.10 la diferencia entre la imagen antigua y la actual.

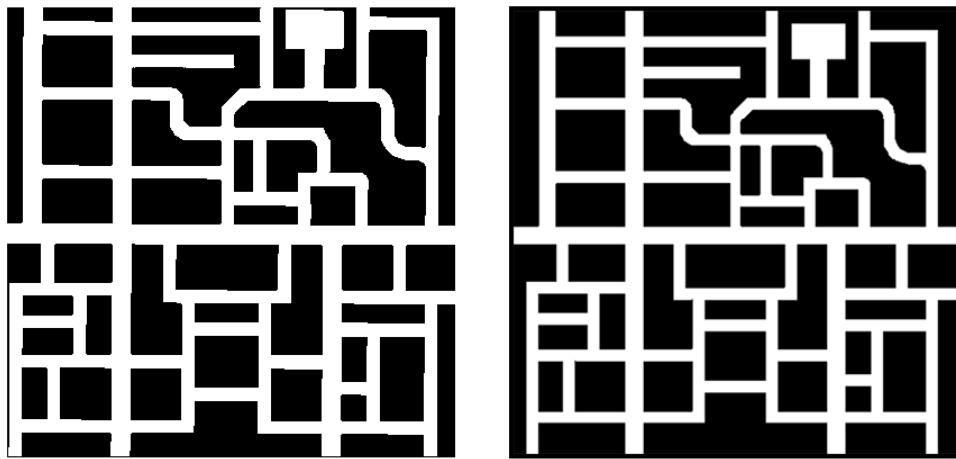


Figura 4.10: Imagen antigua(izquierda) del mapa e imagen actual (derecha)

Este objeto posee una función (llamada *setDestiny*) que será despertada una vez hagamos click sobre la interfaz gráfica para seleccionar el destino deseado. Esta función almacenará las coordenadas del destino en una variable (*destiny*). Esta variable será de gran utilidad para la resolución de la práctica.

El objeto *Grid* cuenta con funciones que le permiten relacionarse con el mapa del mundo:

- *grid.getMap()*: devuelve la imagen del mapa que se está mostrando.
- *grid.getDestiny()*: devuelve el destino seleccionado en la interfaz gráfica. Este destino se devuelve como una tupla (x, y).
- *grid.getPose()*: devuelve la posición respecto al mapa. También será una tupla (x, y).
- *grid.showGrid()*: crea una ventana en la que representa los valores del campo que se le han asignado a la rejilla. Los valores más pequeños del campo tendrán un color

más cercano a negro, y se irán haciendo más claros a medida que se trate de valores superiores.

Este objeto *Grid* también posee funciones que permiten interactuar con la rejilla del campo ficticio (donde se apunta la distancia al destino en ella). Los valores de esta rejilla son de tipo float. Las funciones son:

- *grid.setVal(x, y, val)*: Esta función establece el valor val en la posición indicada (x, y).
- *grid.getVal(x, y)*: devuelve el valor de la posición (x, y) del *grid*.

El objeto *Grid* consta de funciones que interactúan con la rejilla que contiene la ruta más corta. Los puntos de la rejilla con valor 0 serán ignorados, mientras que los valores superiores a 0 serán considerados parte del camino. Las funciones para interactuar con esta rejilla son:

- *grid.setPathVal(x,y, val)*: establece el valor val en la posición indicada (x, y).
- *grid.getPathVal(x,y)*: devuelve el valor de la posición indicada (x, y).
- *grid.setPathFinded()*: indica que se ha encontrado el camino para que empiece a pintarse.

Además, esta clase tiene funciones para pasar de coordenadas del mundo a coordenadas del mapa (fila-columna de la rejilla) y viceversa:

- *gridToWorld(gridX, gridY)*: recibe las coordenadas x e y del mapa (*gridX*, *gridY*) y devuelve una tupla con las coordenadas equivalentes en el mundo (*worldX*, *worldY*).
- *worldToGrid(worldX, worldY)*: recibe las coordenadas x e y del mundo (*worldX*, *worldY*) y devuelve una tupla con las coordenadas equivalentes en el mapa (*gridX*, *gridY*).

Las funciones *gridToWorld* y *worldToGrid* han sido modificadas en la versión actual, puesto que estas funciones tenían un sistema de conversión muy “ad hoc” apropiado para cómo estaba hecha la práctica anteriormente, pero no era genérico. Empleaba un sistema de conversión por casos, en vez de emplear matrices de rotación y traslación para pasar de

un sistema de referencia a otro. Por eso se ha modificado empleando estas matrices. Uno de los motivos del cambio fue que no era correcta la conversión, sino que era aproximada, lo que provocaba una desviación que afectaba al sistema de pilotaje. Si por ejemplo, teníamos una ruta pintada por el centro de la carretera en la GUI, al realizar el cambio a coordenadas del mundo nos daba como resultado una posición de Gazebo con la cual el coche iba pegado a las paredes, y por ello se chocaba con ellas. En la Figura 4.11 podemos ver la imagen del mapa con la antigua versión (a la izquierda) y la misma con la nueva versión (a la derecha). En este mapa se ha pintado un píxel en rojo en la posición (200, 200), que corresponde con el centro de la imagen, y con la posición donde se encuentra el taxi al lanzar la práctica.

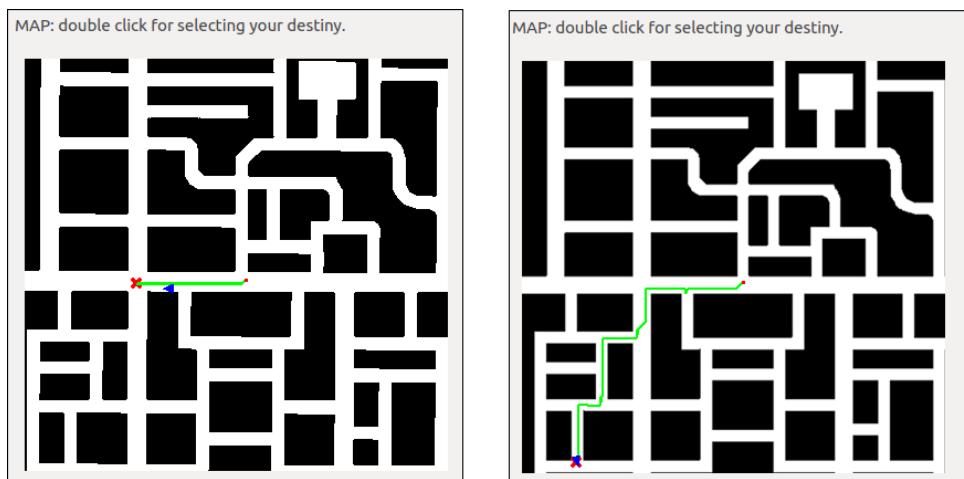


Figura 4.11: Imágenes con el centro incorrecto (izquierda) y el centro correcto (derecha)

Las matrices de rotación y traslación son muy útiles para pasar de un sistema de referencia del mundo en 3D a un sistema de referencia del mapa en 2D o, al contrario. Estos cambios de sistema de referencia son necesarios porque en la práctica si realizamos la expansión del campo del gradiente sobre una rejilla debemos saber la relación entre cada celdilla de la rejilla con el mundo. Los sistemas de referencia de cada sistema están situados de la siguiente forma:

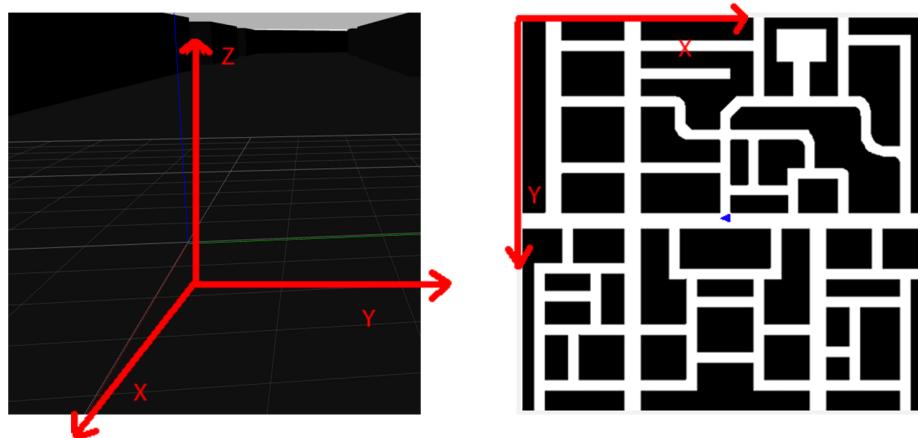


Figura 4.12: Sistema de referencia del mundo (izquierda) y sistema de referencia del mapa (derecha)

Para pasar de un sistema a otro deberemos aplicar la rotación y la traslación necesarias. Por ejemplo, en la función *WorldGrid* tenemos que pasar de una coordenada (x, y, z) en un mundo 3D a una coordenada (x', y') en 2D. Primero se aplicará una matriz de rotación de π grados sobre el eje x para que el sistema de coordenadas del mundo rote de la siguiente forma:

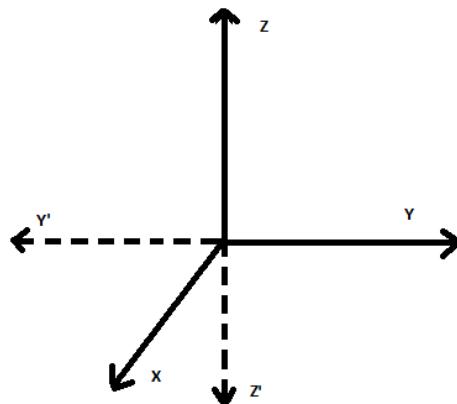


Figura 4.13: Rotación sobre el sistema de referencia del mundo (sistema punteado)

Por si tenemos que aplicar alguna traslación además de la rotación, tenemos una matriz de rotación y traslación para el eje x. Si no queremos realizar una traslación los puntos tx , ty , y tz tendrán valor 0. La traslación se debe a que queremos tener el punto $(0,0)$ de la imagen en la esquina superior izquierda. El cambio lo realizamos siguiendo la ecuación 4.1.

$$\begin{bmatrix} x' \\ y' \\ z' \\ 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & tx \\ 0 & \cos(\alpha) & -\sin(\alpha) & ty \\ 0 & \sin(\alpha) & \cos(\alpha) & tz \\ 0 & 0 & 0 & 1 \end{bmatrix} * \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} \quad (4.1)$$

Lo siguiente que tendremos que hacer es aplicar una rotación de $-\pi/2$ grados sobre el eje z y una traslación si los puntos están un poco desviados. La traslación será de 200 píxeles (ancho de la imagen/2) en x y -200 píxeles (-alto de la imagen/2) en y. Por lo tanto, tx será en este caso 200, y ty será -200, tz será 0. La matriz de rotación y traslación la aplicamos de la siguiente forma:

$$\begin{bmatrix} x' \\ y' \\ z' \\ 1 \end{bmatrix} = \begin{bmatrix} \cos(\alpha) & -\sin(\alpha) & 0 & tx \\ \sin(\alpha) & \cos(\alpha) & 0 & ty \\ 0 & 0 & 1 & tz \\ 0 & 0 & 0 & 1 \end{bmatrix} * \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} \quad (4.2)$$

Para realizar la conversión de la imagen al mundo deberemos aplicar las matrices de rotación inversas a estas.

4.4. Solución de referencia

El objetivo de esta práctica es proveer al robot de un algoritmo de navegación, que estará compuesto por un algoritmo de navegación global y otro de pilotaje. El algoritmo de cálculo del campo (navegación global) se realizará en una iteración sin tiempo límite, y el algoritmo de pilotaje responde a un control reactivo. En esta sección abordaremos una breve explicación sobre navegación autónoma de robots, una descripción de la técnica *Gradient Path Planning*, y la solución concreta de la práctica. El fichero *MyAlgorithm.py* donde se inserta el código de la solución de referencia es de naturaleza iterativa, ejecuta continuamente iteraciones y en cada una de ellas se percibe y se controla. El alumno tiene que llenar con su código la función *execute*, que el componente académico invoca periódicamente.

4.4.1. Fundamentos de la Navegación global y el algoritmo GPP

El principal problema de los robots móviles es la navegación autónoma. Es la capacidad que poseen los robots para ir desde un punto del espacio a otro cualquiera evitando chocarse con algún obstáculo, ya sean objetos fijos u objetos móviles inesperados. Esta es una tarea compleja, que provee a los robots de grandes capacidades. La navegación de los robots de forma habitual se divide en dos ramas: la navegación global y la navegación local.

La navegación global consiste en calcular o planificar una ruta de forma óptima inicialmente para que el robot la pueda seguir. Para llevar a cabo este proceso el robot debe tener un conocimiento previo del escenario por el cual se debe mover. El robot suele tener previamente esta información mediante un mapa del entorno. De no ser así, el robot adquirirá dicho conocimiento construyendo un mapa del entorno por medio de los sensores que posee. La ruta se construye empleando técnicas de búsqueda o planificación, que requieren un tiempo considerable. Ejemplos de estas técnicas son: grafos de visibilidad y *Gradient Path Planning*.

Para solucionar el problema de la navegación global, hemos escogido el algoritmo *Gradient Path Planning* (GPP), que garantiza una trayectoria mínima entre el punto desde el que partimos hasta el punto de destino. La trayectoria calculada no se basa en la distancia euclídea mínima, sino que se incorporan los obstáculos en el cálculo de la trayectoria.

Para desarrollar el algoritmo *Gradient Path Planning* hemos partido del trabajo desarrollado por Kurt Konolige [13], así como de trabajos previos realizados en la Universidad Rey Juan Carlos [14] [15]. Esta técnica obtiene el camino óptimo desde el punto de partida hasta el destino.

La técnica GPP consiste en generar un frente de onda circular que parte desde la posición de destino, y que recorre el espacio libre del mapa hasta llegar a la posición de partida del robot. El punto donde está situado el robot es el comienzo de la ruta que recorrerá el robot. En su propagación, el frente de onda asignará valores de forma creciente a cada punto libre del espacio por el que pase. Antes de comenzar la propagación del frente de onda, todos los puntos libres del espacio tienen valor 0. El frente de onda se puede expandir por todo el espacio, hasta la posición que ocupa el robot o un poco más allá.

Adicionalmente los obstáculos generarán su propio frente de onda de penalización, lo que implica que los puntos del espacio que estén próximos a los obstáculos aumentarán su valor por defecto considerablemente. El frente de onda de penalización se propaga de forma inversa al frente de onda anterior. Esto quiere decir que los puntos del espacio más próximos a los obstáculos tendrán un valor mayor a los puntos más alejados. El frente de onda de los obstáculos se expande hasta una distancia determinada, no se expanden por todo el espacio.

Sumar la expansión del frente de onda de penalización evita que el robot se acerque a los obstáculos al navegar por el espacio. De no ser así, la ruta más corta estaría pegada a los obstáculos, y de esta forma el robot rozaría con las paredes. Esto hará que el robot tenga mayor seguridad.

Una vez se ha generado el campo total, el robot podrá navegar hasta el destino evaluando en cada iteración los puntos de su alrededor. Siempre se dirigirá hacia el punto de menor valor del campo calculado. A medida que avance el robot hacia su destino, el campo calculado tendrá menor valor, pues las zonas más próximas al destino son las que menor valor poseen.

El método GPP permite generar una ruta ideal desde el punto de partida del robot hasta el destino deseado. Esta ruta se generará siguiendo el gradiente del campo calculado, y será la de menor distancia incorporando los obstáculos. Esta técnica de navegación global asegura que llegaremos al objetivo. Sin embargo, es posible que el robot durante el pilotaje no siga exactamente la ruta calculada, pues puede tener desviaciones debidas a la velocidad y rotación del robot. El pilotaje se llevará a cabo de forma reactiva, mirando en cada momento cuál es la celdilla vecina de menor valor a la que debe dirigirse, y si se encuentra próximo a algún obstáculo.

La forma reactiva se basa en el “ahora”, es decir, en cada instante de tiempo evalúa la situación y actúa. No desarrolla una solución inicial y la sigue, sino que va actuando en función de lo que ocurra en cada momento.

4.4.2. Construcción del mapa del gradiente

La navegación global mediante *Gradient Path Planning* se puede implementar de diversas formas. La solución implementada se desarrolla en el fichero “*MyAlgorithm.py*”. En este fichero podremos observar que la solución se divide en un método en el que se construye el mapa del gradiente, y otro método que se corresponde con el pilotaje del robot.

En el método “*generatePath*” del fichero “*MyAlgorithm.py*” llevaremos a cabo el desarrollo del algoritmo que genera el mapa del campo del gradiente. Esta función se ejecutará solamente cuando pulsemos el botón “*Generate Path*” en la GUI (Figura 4.6). El escenario estará representado gráficamente por una rejilla, donde iremos almacenando el campo calculado. Esta rejilla es proporcionada por la clase *Grid*.

Lo primero que debemos conocer antes de comenzar a generar el campo y expandirlo por la rejilla es el mapa, la posición inicial del robot y la posición de destino deseado. En la práctica, se dispone de un objeto *grid* que permite obtener el mapa a través de la función *grid.getMap()*. Este mapa proporciona información del escenario mediante sus valores, donde el valor 0 representa a los obstáculos y el valor 255 representa la carretera. El destino podrá conocerse, una vez el usuario haya seleccionado el destino deseado en la GUI, mediante la función *grid.getDestiny()*. Por último, podemos obtener la posición del robot respecto al mapa mediante la función *grid.getPose()*.

4.4.2.1. Generación campo ficticio de navegación global

En la función “*generatePath*” inicialmente tendremos un bucle que realiza la propagación de los frentes de onda. Es decir, no tendremos un único frente de onda, sino que vamos a tener diferentes frentes de onda, los cuales se encuentran en una lista ordenada.

El bucle de la propagación de los frentes de onda se ejecutará hasta que se expanda el campo un poco más allá de la posición del taxi. En este caso se expandirá hasta 20 celdillas más allá de la celdilla que ocupa el taxi en la rejilla. Este número podría haber sido mayor o menor, pero en el caso de que fuera mayor la propagación tardaría más tiempo en ejecución. Si el número fuera menor tendríamos menos conocimiento de los alrededores iniciales del robot.

El punto de partida del frente de onda inicial es el destino, que es la celdilla de la rejilla que posee la distancia 0. Será nuestro primer nodo que expandirá el frente de onda a sus vecinos. Cada nodo expandirá el valor de la distancia a sus 8 celdillas vecinas. Este valor de distancia asignado a las celdillas vecinas será el valor del nodo (distancia) + 1 o el valor del nodo + 1.4 en las diagonales. Cuando se encuentre un obstáculo, se almacenará la celdilla de dicho obstáculo en un array y no se le asignará ninguna distancia. De esta forma aseguramos tener almacenadas las celdillas que pertenecen al borde de un obstáculo para posteriormente penalizar a las celdillas que se encuentren muy próximas a los obstáculos.

Cada nodo va a expandir antes el frente a sus vecinos que se encuentren a una distancia +1, y posteriormente expandirá el frente a sus vecinos que estén en una distancia + 1.4. Por lo que se podría decir que el frente de onda de cada nodo se divide en dos frentes de onda. De esta forma nos aseguramos que el frente de onda sea aproximadamente circular (Figura 4.14).

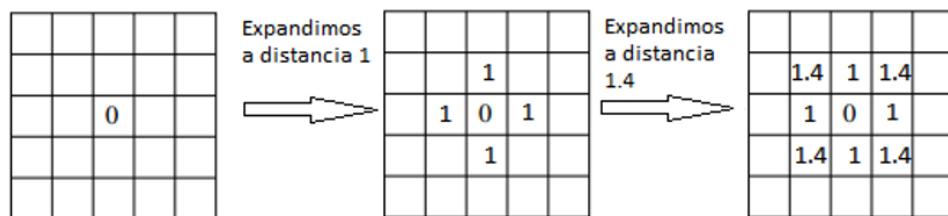


Figura 4.14: Primera propagación del frente de onda

Este proceso de expansión (Figura 4.15) se irá haciendo sucesivamente hasta llegar a 20 celdillas más alejadas de la posición inicial del taxi. A la hora de realizar la expansión, si las celdillas vecinas tuvieran un valor mayor al que calculara el nodo, dicho valor se actualizaría por el valor que expanda el nodo actual. Esto permite asegurar que siempre tendremos el menor valor posible en cada celdilla.

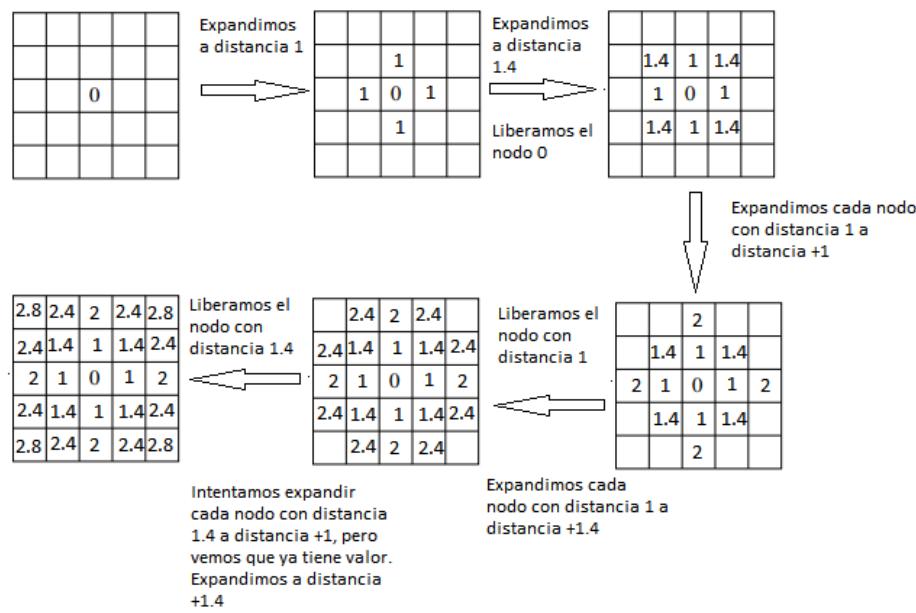


Figura 4.15: Esquema propagación frentes de onda

En la rejilla del campo que hemos generado (Figura 4.16) se puede apreciar en color más oscuro los puntos más cercanos al destino, puesto que poseen un valor menor de distancia. Por el contrario, los puntos más lejanos del destino son los que poseerán un color más claro. A continuación, podemos observar una progresión de la expansión del campo.

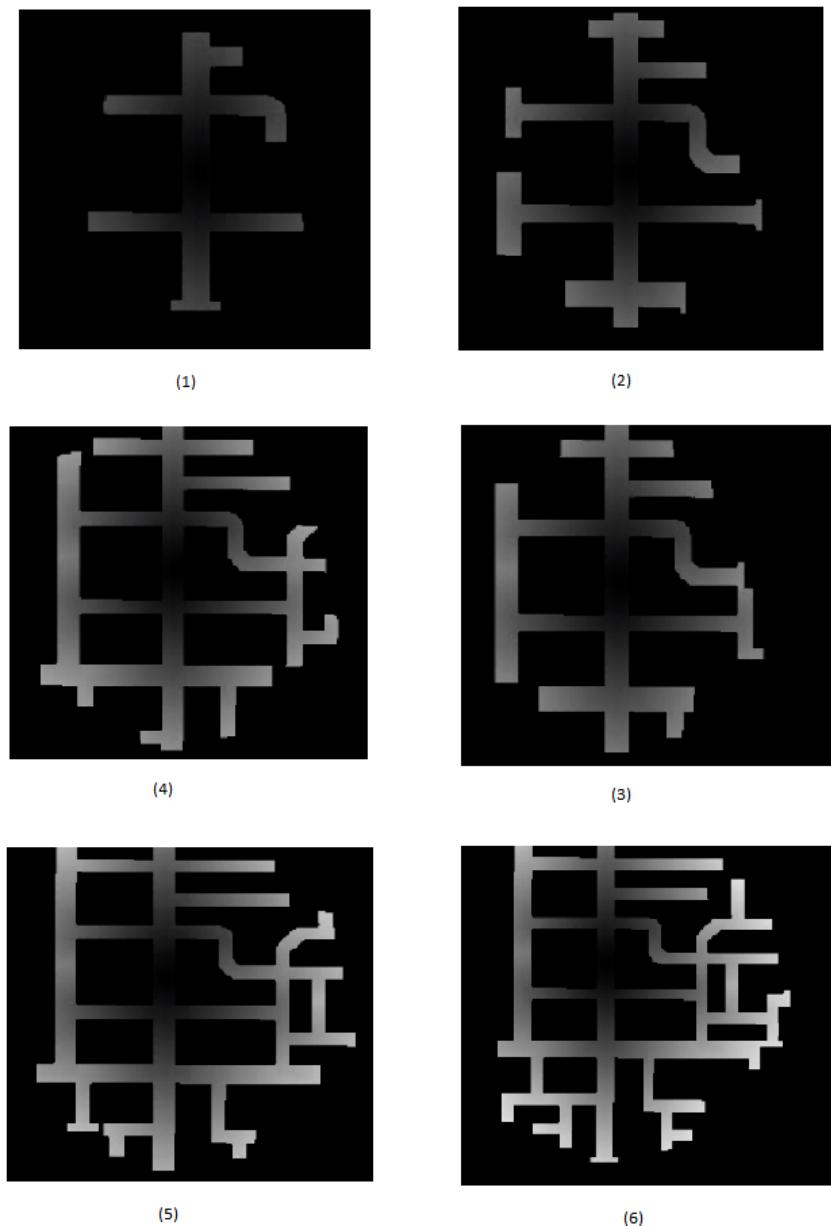


Figura 4.16: Esquema expansión del campo

4.4.2.2. Penalización por cercanía de obstáculos

El segundo paso, es la penalización que realizan los bordes de los obstáculos a las celdillas más próximas. Esta penalización se lleva a cabo para evitar que la ruta más corta desde el robot hasta el destino esté pegada a las paredes de los obstáculos y haga que nuestro taxi roce con las paredes. Con esta penalización nos aseguramos de que en el pilotaje haya un margen de seguridad entre el taxi y las paredes.

Como hemos mencionado antes, las celdillas que pertenecen a bordes de obstáculos están almacenadas en un array (llamado *posObstaclesBorder*). Estas celdas (obstáculos) sumarán una penalización a las celdillas vecinas que formen parte de la carretera en función de la distancia a la que se encuentran de la celdilla obstáculo.

Para llevar a cabo la penalización por obstáculos se ha creado una nueva rejilla, la cual inicialmente posee un valor 0 en todas sus celdillas. En esta rejilla almacenaremos los valores de penalización. Finalmente sumaremos la rejilla del campo y la de los obstáculos para actualizar sus valores con dichas penalizaciones.

La penalización que realizan los obstáculos se llevará a cabo mediante un bucle que recorre el array *posObstaclesBorder*. Cada posición de dicho array penalizará a las celdillas vecinas que están a una distancia de +1, +2 y +3 de la misma. Las celdillas que estén a una distancia de +1 del borde del obstáculo se penalizarán con un valor de 174. Las celdillas con una distancia +2 tendrán una penalización de 168, mientras que las celdillas con una distancia +3 se penalizarán con un valor de 162.

En cada penalización que adjuntemos a una celdilla antes comprobaremos su valor en la rejilla de penalización, puesto que dos celdillas del borde del obstáculo pueden querer penalizar a una misma celdilla, pero esta celdilla solamente se puede penalizar una vez. Cuando vayamos a penalizar y comprobemos si ha sido penalizada una celdilla, comprobaremos su valor. Si el valor de penalización que tiene dicha celdilla es menor que el que se iba a añadir, se sustituirá el valor de penalización por el mayor.

Estas penalizaciones harán que las celdillas próximas a los obstáculos tengan un valor mayor y aparezcan con un color más claro (blanco) al mostrar el *grid* (Figura 4.17). El *grid* lo podemos mostrar mediante la función *grid.showGrid()*. Esta función crea una ventana en la que representa los valores del campo que se le han asignado a la rejilla. Podemos ver en la Figura 4.17 una serie de ejemplos de diferentes campos calculados con la penalización por obstáculos incluida.

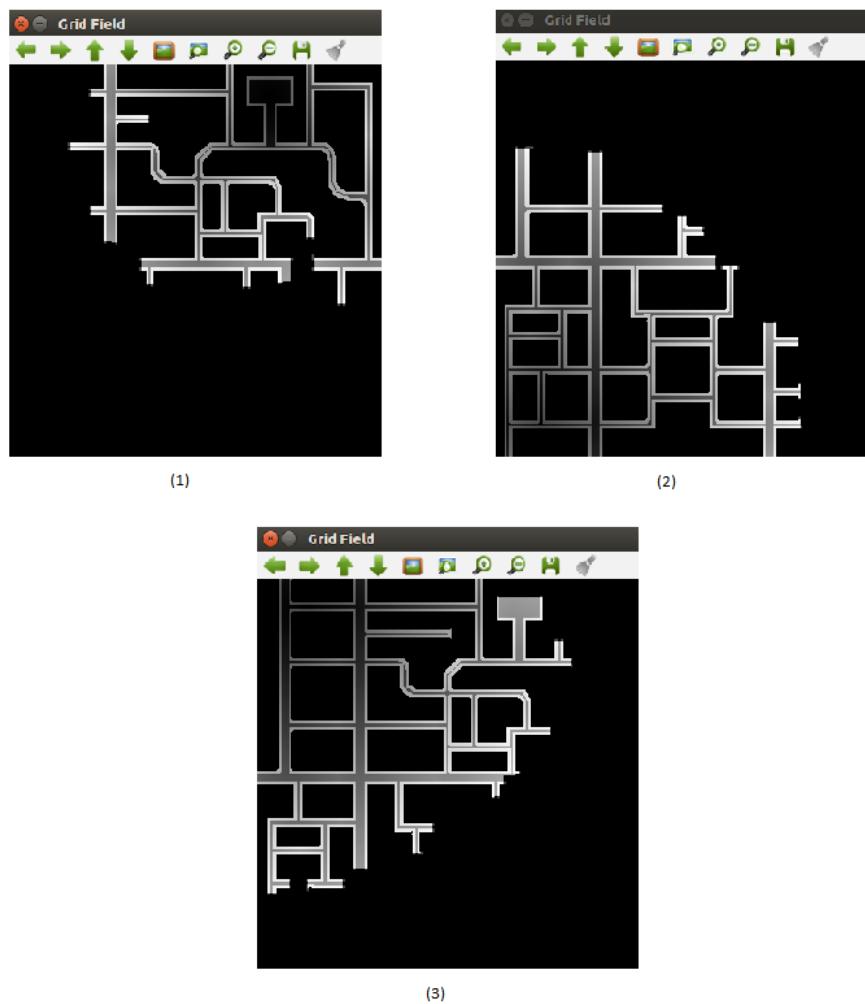


Figura 4.17: Representación campos calculados

4.4.2.3. Cálculo de ruta ideal

El paso siguiente es el cálculo de la ruta más corta. Esta ruta se calcula para observar cuál sería la ruta ideal que debe seguir nuestro taxi. Dicha ruta sigue las celdillas de menor valor de distancia.

El cálculo de la ruta más corta comienza en la celdilla donde se encuentra situado el taxi y termina al alcanzar la celdilla que posee el valor de distancia 0, es decir, el destino. Para ir almacenando la ruta tenemos que hacer uso de la función *grid.setPathVal*, la cual establece el valor en la posición que se le indica, tomando como ruta las celdillas que poseen un valor diferente a 0.

Comenzamos el cálculo de la ruta añadiendo la posición inicial del taxi a la ruta. El siguiente paso es comprobar las celdillas vecinas de la celdilla que ocupa el taxi. Entre estos vecinos añadiremos a la ruta la de menor valor de distancia. Después, comprobaremos los vecinos de esta nueva celdilla y así continuamente hasta llegar a la celdilla destino.

Una vez que alcancemos la celdilla del destino tendremos que indicar que hemos terminado de calcular la ruta más corta y que se puede comenzar a pintar. Para ello empleamos la función *grid.setPathFound*.

En la Figura 4.18 se pueden observar diferentes rutas calculadas en función del destino que hemos elegido. En las imágenes (1), (2) y (3), podemos ver las rutas calculadas aplicando la penalización de los obstáculos; mientras que en las imágenes (4), (5) y (6) vemos las rutas que se han calculado para los mismos destinos que en (1), (2) y (3), pero sin realizar la penalización de los obstáculos.

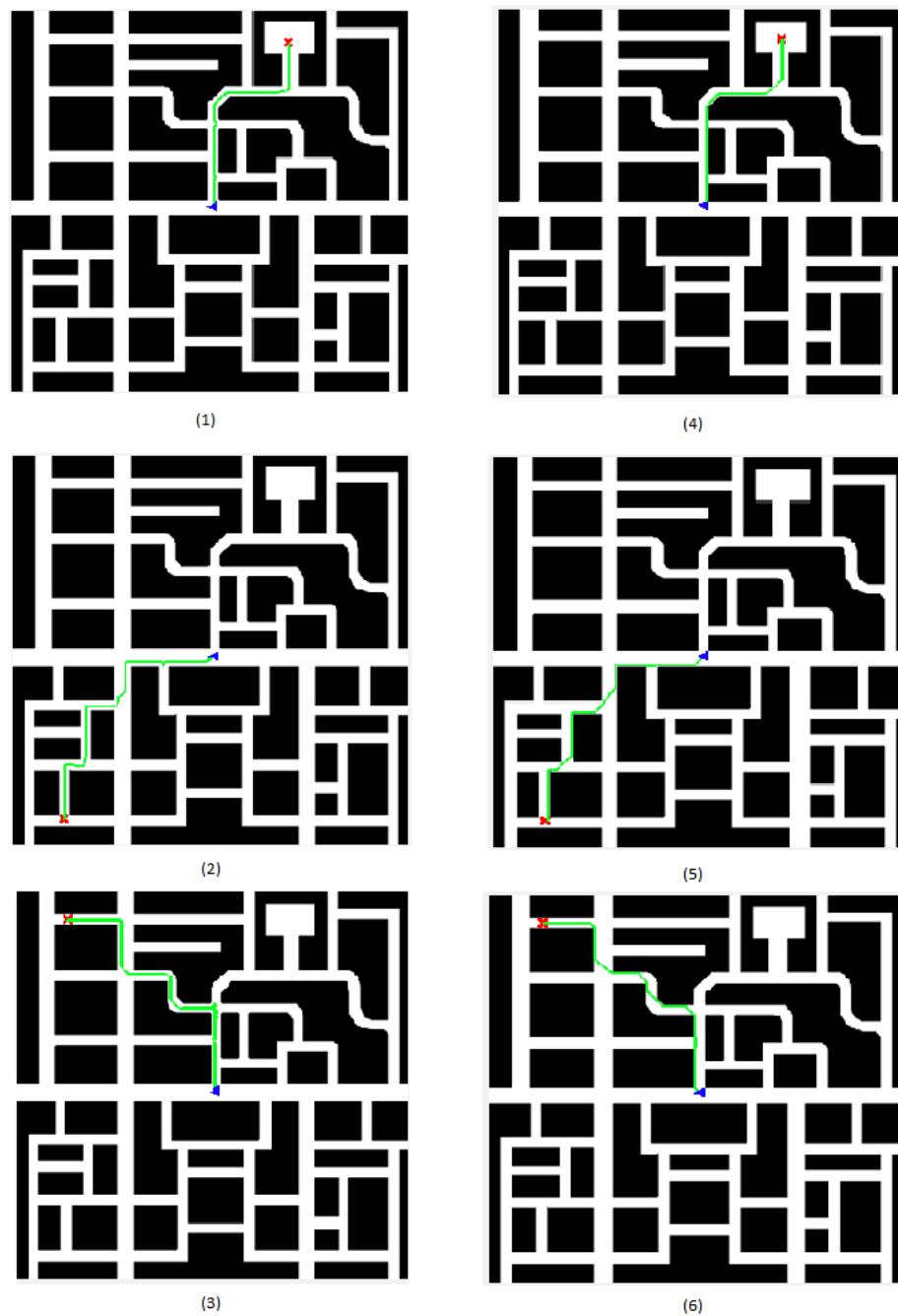


Figura 4.18: Esquema expansión del campo

4.4.3. Pilotaje del robot

En el método “*execute*” del fichero “*MyAlgorithm.py*” incluimos el código del algoritmo correspondiente al pilotaje del taxi. Este método se ejecuta periódicamente para que el pilotaje sea un control reactivo.

Este algoritmo se encarga de pilotar el robot desde su posición inicial hasta la posición del destino mediante el campo calculado en el método *generatePath*. La dificultad está en la elección de la velocidad de tracción y la velocidad de rotación que debemos ordenar al taxi.

El pilotaje se ha realizado sin tener en cuenta la ruta más corta calculada en el punto anterior, ya que dicha ruta es el ideal a seguir, pero el taxi al seguir órdenes de velocidad de tracción y de rotación puede desviarse un poco de dicha ruta. Si intentara seguir la ruta ideal estrictamente, el taxi realizaría movimientos muy forzados hasta llegar al destino. Lo ideal es que el taxi se mueva de una forma suave, como lo haría un taxi real. En el pilotaje se ha tenido en cuenta el campo calculado. De esta forma, en cada iteración el taxi irá comprobando el valor de distancia de las celdillas que se encuentran en un cierto radio de distancia con respecto a su posición. De estas celdillas elegirá como objetivo la celdilla de menor valor. Este planteamiento permite que el taxi tenga un comportamiento reactivo ante imprevistos y que se asemeje un poco a la navegación local, ya que no tiene en cuenta únicamente la ruta más corta calculada previamente, también tiene en cuenta la situación del taxi.

Inicialmente en el pilotaje debemos comprobar la pose de nuestro taxi mediante el sensor de posición y la posición del destino para ver cuál es la situación del taxi, ya que si el taxi ha alcanzado el destino debe detenerse. Conocemos la celdilla que ocupa el destino en el *grid*, pero el sensor de posición devuelve la posición del taxi con respecto al mundo. Esto implica que debemos convertir las coordenadas del destino en el *grid* en coordenadas respecto al mundo para comprobar si hemos llegado a dicho destino. Para ello se usa la función (*grid.gridToWorld*) que realiza la correspondencia de las coordenadas del *grid* con la posición que tendrían estas coordenadas en el mundo de Gazebo.

El siguiente paso es calcular el objetivo local próximo al robot. Vamos a ir calculando en cada iteración un objetivo que se encuentra a cierto radio de distancia del robot. Comprobando los valores de distancia del campo que habíamos calculado en el punto anterior. Por lo que tendremos que obtener la posición del taxi en el sistema de coordenadas del *grid* (mediante la función *worldToGrid*). En nuestro caso comprobaremos las celdillas si-

tuadas a una distancia de 5 celdillas con respecto a la posición del robot y escogeremos la de menor valor como objetivo local. El objetivo no es exactamente el anterior mencionado, sino que vamos a calcular un segundo objetivo situado a 5 celdillas del primer objetivo, y posteriormente haremos la interpolación de estos dos objetivos obteniendo el objetivo final al que queremos llegar. El motivo de esta interpolación y ese cálculo doble es obtener un pilotaje con movimientos más suaves. Además, esto nos permite que el taxi gire adecuadamente en las curvas.

El objetivo local final calculado está en coordenadas del *grid*, por lo que tendremos que hacer un cambio de coordenadas relativas al mundo para obtener el objetivo en coordenadas del mundo. Una vez lo tengamos, podremos calcular el vector de dirección que tendrá nuestro robot para llegar hasta este y el ángulo. Para ello debemos tener en cuenta la pose que nos devuelve el sensor de posición, así como la orientación del robot (en radianes), las cuales están en coordenadas absolutas del mundo.

En esta práctica hay dos sistemas de referencia. Por un lado, tenemos el sistema de referencia absoluto, en el cual se pueden representar la posición del robot y la de cualquier otro objeto. En nuestro caso serán los ejes de Gazebo, por lo cual el punto fijo de referencia será el punto de coordenadas (0, 0, 0) en el mundo de Gazebo. El otro sistema de referencia que vamos a tener en cuenta es el sistema de referencia solidario con el robot. Según se mueva el taxi este sistema de referencia solidario con el robot se desplazará. También hay que mencionar que inicialmente el robot comienza con una rotación de $-\pi/2$ radianes.

En la Figura 4.19 podemos ver el sistema de referencia absoluto (son los ejes finos azul, verde y rojo que atraviesan la imagen) y el sistema de referencia solidario con el robot (flechas gruesas azul, roja y verde que salen del robot):

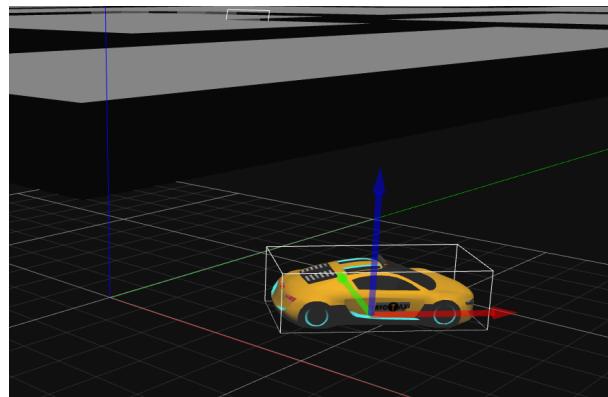


Figura 4.19: Sistema de referencia absoluto (Gazebo) y sistema de referencia solidario con el robot

Se ha decidido definir el sistema de referencia solidario con el robot de la siguiente forma: el eje X de este sistema es el que señala al frente del robot, mientras que el eje Y es el que señala a la izquierda del robot. El origen de este sistema es un punto situado en el centro geométrico del robot.

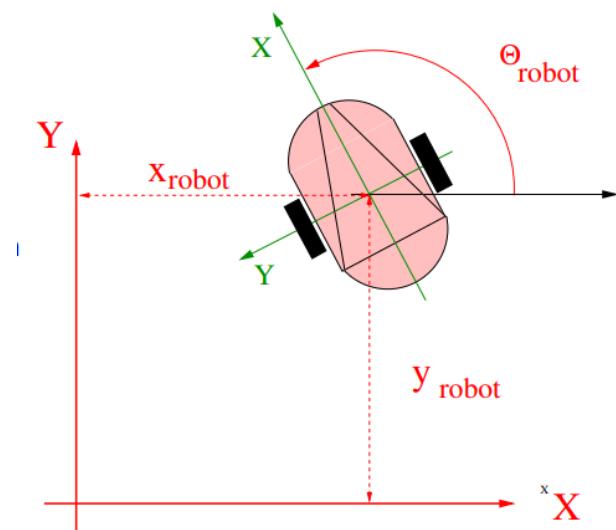


Figura 4.20: Sistema de referencia absoluto y sistema de referencia solidario con el robot

El sistema de referencia solidario con el robot se emplea para expresar las coordenadas relativas de los objetivos próximos o de obstáculos respecto al robot. Si tenemos un punto P que no se mueve en el espacio para el sistema de referencia absoluto no habrá movimiento, pero para el sistema de referencia solidario con el robot las coordenadas de

este punto P varían.

Sabiendo las coordenadas absolutas de un punto, y las coordenadas absolutas del robot y su orientación, podemos pasar las coordenadas absolutas a relativas o al revés. En nuestro caso deberemos pasar las coordenadas absolutas de cada objetivo próximo a coordenadas relativas al sistema del robot para calcular la velocidad de tracción y rotación que debe tener el taxi.

Con la transformación de coordenadas absolutas a relativas obtenemos un vector de dirección, con el que podremos calcular el ángulo que hay entre el origen del sistema relativo al robot y la posición relativa del objetivo local. Para obtener el vector de dirección hemos creado una función, en la que primero calcularemos la diferencia entre las coordenadas absolutas del objetivo y las coordenadas absolutas del robot (dx , dy), y después a esta diferencia le aplicamos una matriz de rotación para obtener el vector de dirección. En esta matriz de rotación tendremos en cuenta la orientación (Θ) del robot.

$$x' = dx \cos(\Theta) - dy \sin(\Theta) \quad (4.3)$$

$$y' = dx \sin(\Theta) - dy \cos(\Theta) \quad (4.4)$$

Con este vector de dirección conocemos dónde se encuentra situado el punto objetivo con respecto a nuestro origen del sistema del robot, ahora podremos calcular el ángulo que debe rotar el robot para alinearse con este punto. Este ángulo lo calcularemos mediante el cálculo del arco tangente. En la Figura 4.21 podemos ver un dibujo del ángulo α que queremos calcular y el punto P (objetivo). Conociendo estos datos podemos calcular α con la arco tangente.

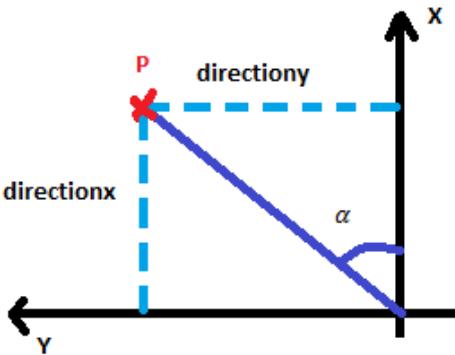


Figura 4.21: Sistema de referencia solidario con el robot y punto objetivo

El ángulo α lo podemos calcular de la siguiente forma:

$$\alpha = \text{arcotang} \left(\frac{\text{directiony}}{\text{directionx}} \right) \quad (4.5)$$

El ángulo calculado α está en radianes. Si este ángulo es muy grande significará que debemos dotar al taxi de una velocidad de rotación alta. Por el contrario, si este ángulo es muy pequeño significa que el robot se encuentra más o menos alineado con el objetivo y que probablemente la velocidad de rotación de nuestro vehículo sea 0.

En esta solución se ha realizado un control por casos en función del ángulo α calculado. En función de este ángulo ordenaremos al taxi mayor o menor velocidad de tracción y de rotación en esa iteración de control. Si el ángulo calculado es muy elevado aplicaremos una velocidad de tracción reducida y una velocidad de rotación elevada (el coche puede estar en una curva o necesitar realizar un gran giro). Sin embargo, si el ángulo es pequeño, le daremos al taxi una velocidad de tracción elevada (ya que se encuentra en una recta) y poca velocidad de rotación.

En las Figuras 4.22, 4.23, 4.24 y 4.25 podemos observar una secuencia de imágenes en la que se ha calculado una ruta y podemos ver cómo aproximadamente el taxi sigue esta ruta, pero a veces se desvía un poco, como habíamos mencionado anteriormente que podría suceder. El taxi alcanza el destino deseado con éxito.

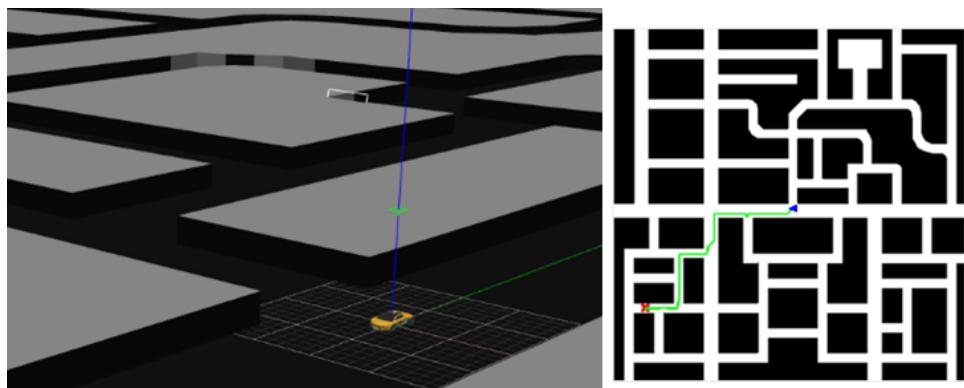


Figura 4.22: Posición 1 taxi en el mundo de Gazebo y en la GUI

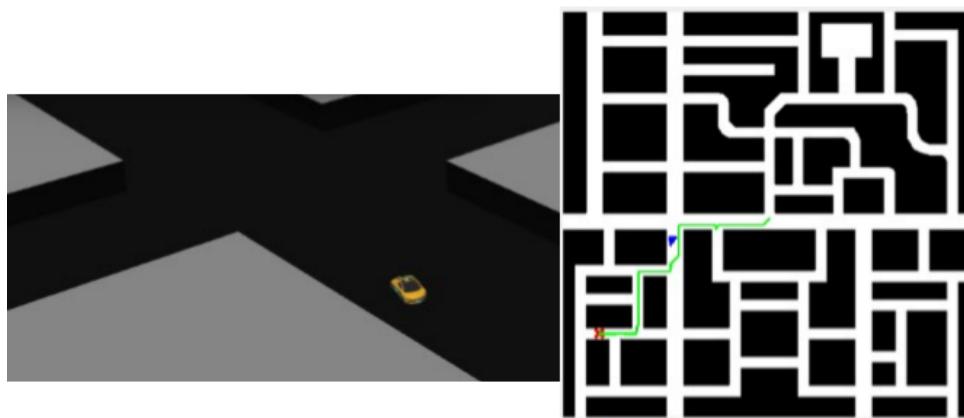


Figura 4.23: Posición 2 taxi en el mundo de Gazebo y en la GUI



Figura 4.24: Posición 3 taxi en el mundo de Gazebo y en la GUI

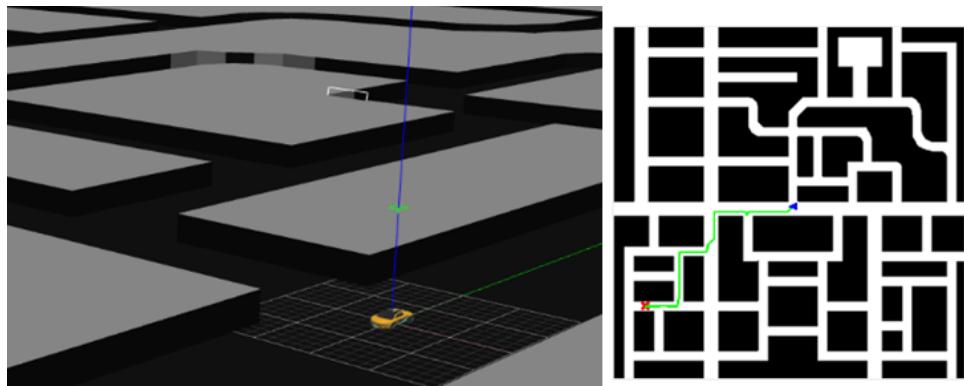


Figura 4.25: Posición 4 (destino) taxi en el mundo de Gazebo y en la GUI

4.5. Evaluador Automático

La práctica consta además de un evaluador automático que tiene en cuenta ciertos parámetros para calificar el algoritmo que programa el alumno. Este evaluador automático tiene una interfaz gráfica que muestra los diferentes parámetros, así como la nota final. Para crear el evaluador automático se ha empleado PyQt5 y se han creado clases diferentes para cada parámetro que queramos mostrar. Estas clases serán instanciadas en una clase principal llamada *MainWindowReferee*, que contiene la ventana principal del evaluador automático.

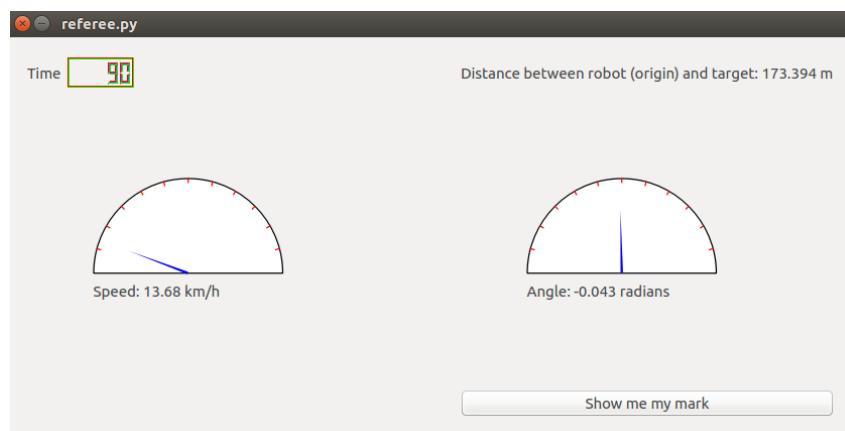


Figura 4.26: Evaluador Automático del GPP durante el pilotaje

El evaluador (Figura 4.26) tiene en su visor un reloj digital que va mostrando los

segundos que han pasado desde que se arrancó la práctica. Este visor de tiempo está programado en una clase (llamada *timeWidget*), en la cual se almacenará en una variable los segundos que el taxi está realizando el pilotaje.

En segundo lugar, se ha creado una clase (*distanceWidget*) para mostrar la distancia euclídea que existe entre la posición inicial del taxi y el destino que se ha marcado en la GUI. En todos los casos con esta clase se mostrará un mensaje en el visor de la aplicación. Si el destino no ha sido seleccionado todavía mostrará el mensaje “*Distance between robot and target: Destination not yet selected*”. Por el contrario, si ya hemos escogido el destino deseado, mostrará el mensaje: “*Distance between robot (origin) and target: X m*”. Con X nos referimos a que dependiendo de donde esté colocado el destino se mostrará una distancia u otra.

En tercer lugar, tenemos un visor de la velocidad (en km/h) que alcanza nuestro taxi durante el pilotaje. En esta clase se realiza un pintado de un velocímetro, que marca con una aguja la velocidad que lleva nuestro vehículo. Cuando el coche está totalmente parado, la aguja aparecerá tumbada hacia la izquierda. Además, debajo del velocímetro aparecerá un mensaje con la velocidad numérica en km/h que tiene nuestro taxi en cada momento.

Al igual que sucede con el velocímetro, tenemos un visor similar para pintar la orientación absoluta que tiene nuestro taxi en todo momento, a modo de brújula. Esta orientación irá desde un ángulo de $-\pi$ (izquierda) a un ángulo de π (derecha). En esta clase (*angleWidget*), también, se mostrará un mensaje con la orientación del taxi.

Por último, tenemos la clase que calcula la nota final. En la interfaz gráfica tenemos un botón con el mensaje “*Show me my mark*”. Si pulsamos este botón mostrará nuestra nota. Si el botón se pulsa antes de que el taxi llegue al destino fijado nos indicará en un mensaje que el destino aún no se ha alcanzado. La nota final sólo se calcula si hemos llegado al destino. Si hemos llegado a destino ya tendremos como mínimo una nota de 7, y como máximo una nota de 8, a la cual le sumaremos hasta 2 puntos como máximo en función de la velocidad media del taxi en el pilotaje. Si nos hemos quedado a una distancia máxima de 2 metros del destino tendremos un 8 de nota, a la que le sumaremos hasta 2 puntos

como máximo. Por el contrario, si nos quedamos hasta 4 metros de distancia del objetivo el valor de la nota de la que partimos será 7.5; y si por el contrario nos hemos quedado hasta 5 metros del destino, la nota de partida será de 7 puntos. Si nos hemos quedado a más de 5 metros del destino se considerará que no hemos llegado aún. El cálculo del campo del gradiente no se tiene en cuenta en la nota, puesto que es muy difícil comprobar si se ha realizado el cálculo del campo correctamente, ya que hay diferentes modos de realizarlo. Para saber si el campo está bien habría que comprobar la imagen que tenemos como resultado del campo o emplear un vídeo de cómo se realiza la expansión.

4.6. Experimentación

4.6.1. Ejecución típica

Para lanzar la práctica hay que abrir tres terminales y ejecutar en cada uno de ellos:

1. Lanzar Gazebo: gazebo cityLarge.world
- 1b. Si el ordenador que se emplea no tiene muchos recursos se puede arrancar el simulador sin interfaz gráfico: gzserver cityLarge.world
2. Ejecutar el componente académico: python2 globalNavigation.py --mapConfig=taxiMap.conf –Ice.Config=teleTaxi.cfg
3. Ejecutar el evaluador automático: python2 referee.py –mapConfig=taxiMap.conf –Ice.Config=teleTaxi.cfg

Se han realizado numerosos experimentos con éxito para probar y validar la solución de referencia programada. La ejecución típica representativa ya se ha ilustrado en la Sección 4.4.3, que describe una ejecución normal. Una ejecución típica se puede ver en este video ².

4.6.2. Estudio de tiempos

En la práctica es muy importante el tiempo de ejecución, ya que este tiempo influye en la nota de la práctica (velocidad media del taxi) y cuanto más rápido sea el algoritmo

²<https://www.youtube.com/watch?v=bNnUfMMXC64>

mejor. En el tiempo que tarda nuestro taxi en llegar al destino, durante el pilotaje, influirá el ordenador que empleemos. Este es un inconveniente, ya que quien posea mejor ordenador obtendrá tiempos de ejecución menores que quien posea un ordenador sin tantas capacidades. La ejecución de Gazebo consume muchos recursos del ordenador haciendo que el taxi sea más lento.

En la parte inferior de Gazebo se puede ver el *Real Time*, el *Sim. Time* (tiempo simulado) y el *Real Time Factor*, los cuales tienen mucho que ver en el tiempo de ejecución de Gazebo. El parámetro *Real Time* expresa el tiempo real en ejecución. El factor *Sim. Time* expresa el tiempo simulado. Si utilizáramos un ordenador potente entonces el *Sim. Time* debería estar próximo al *Real Time*. Mientras que si usamos un ordenador sin tantas capacidades veremos que el *Sim. Time* es mucho menor que el *Real Time*. Por su parte, el factor *Real Time Factor* es un producto de la tasa de actualización y el tamaño del paso. Si queremos obtener un tiempo de simulación bajo deberá estar alrededor de 1. Si este parámetro es menor que 1 veremos que la ejecución es más lenta, y cuando se aproxima a 0.2 o menos es demasiado lenta.

En el caso del ordenador concreto que se ha empleado el *Real Time Factor* es muy bajo en algunas ocasiones durante el pilotaje, lo que hace que el *Real Time* sea mucho mayor que el *Sim. Time*. En este ordenador el *Real Time Factor* normalmente oscila entre 0.1 y 0.75, siendo en grandes ocasiones cercano a 0.1.

En el tiempo total de ejecución de la práctica se puede diferenciar el tiempo de planificación y el tiempo de pilotaje. Dependiendo del destino que elijamos ambos tiempos variarán, siendo menor si escogemos un destino cercano. Se han realizado varias pruebas con diferentes destinos para ver la diferencia entre el tiempo de ejecución.

- Destino lejano. Hemos elegido un destino bastante alejado de la posición inicial del robot, lo cual se puede observar en la Figura 4.27. Al realizar la prueba, el tiempo de planificación es de tan solo 16''. El tiempo de pilotaje es bastante mayor, alrededor de 2' 30''. Para tener conciencia de cómo influye el ordenador en la ejecución de la práctica, se ha comprobado el tiempo *Real Time* y *Sim. Time* (se inicializan nada más ejecutar la práctica, aunque aún no se haya comenzado a ejecutar el algoritmo) cuando el taxi alcanza el objetivo. El resultado obtenido es que el *Real time* es 3'

26"; mientras que el parámetro *Sim. Time* adquiere un valor de 1' 30". La diferencia es excesivamente grande, casi de 2 minutos, lo que implica que en un ordenador con las capacidades al máximo el tiempo de ejecución sería relativamente corto.

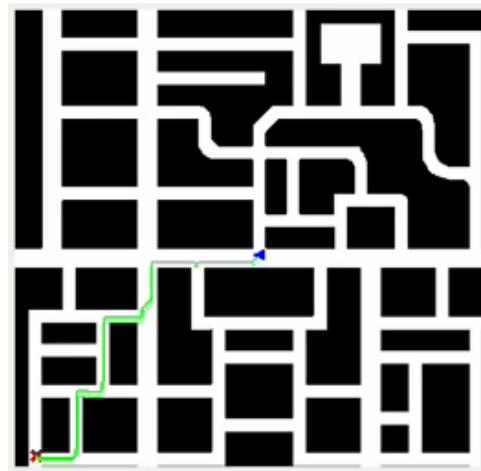


Figura 4.27: Imagen del mapa con el primer destino elegido

- Segundo destino: En esta ocasión se ha elegido un punto en la plaza, que se podrá ver en la Figura 4.28. Al comprobar el tiempo de planificación se ha obtenido un tiempo de 16', al igual que en el caso anterior. El tiempo de pilotaje es de 1' 57". En esta ocasión el tiempo de pilotaje ha sido aproximadamente 30" más rápido. Si comprobamos el *Real Time* vemos que es de 3' 06"; mientras que el *Sim. Time* es de 1' 35". Es decir, en esta ocasión la diferencia entre el tiempo simulado y el tiempo real es menor, lo que nos lleva a obtener un menor tiempo de ejecución.

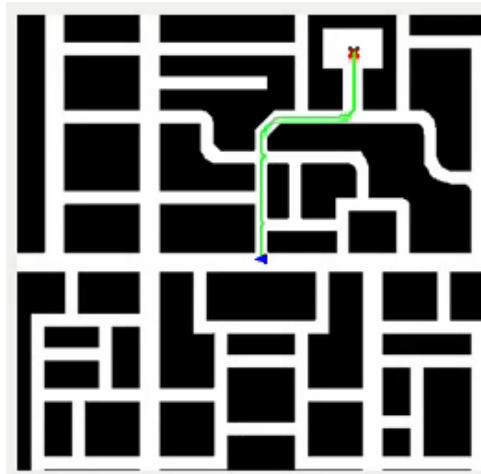


Figura 4.28: Imagen del mapa con el segundo destino elegido

- Destino a distancia media: El punto escogido lo podemos ver en la Figura 4.29. En este caso el tiempo de la planificación ha sido de 19', algo superior a las ocasiones anteriores. El tiempo que tarda el taxi en llevar a cabo el pilotaje es de 1' 12", inferior que en el resto de ocasiones puesto que es un destino más cercano. En esta ocasión el *Real Time* ha sido de 2' 19"; y el *Sim. Time* ha sido de 1' 17". La diferencia ha sido menor que en las anteriores ocasiones.

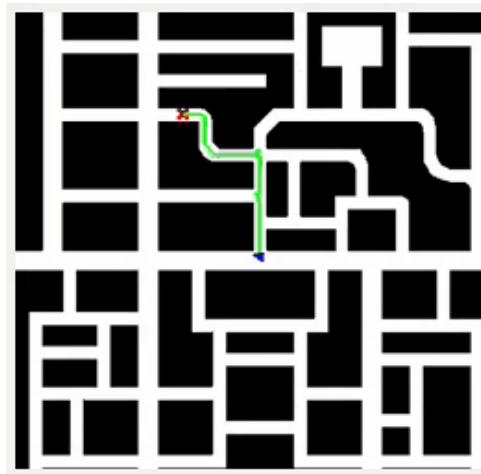


Figura 4.29: Imagen del mapa con el segundo destino elegido

Hemos podido comprobar cómo influyen diferentes aspectos en el tiempo de simulación: la lejanía del destino y las capacidades del ordenador que empleemos. Además, hemos

observado que el tiempo de planificación suele ser bastante corto, esto es debido a que el algoritmo es rápido. El pilotaje por su parte ha sido más lento debido a distintos aspectos. Sería posible mejorar el tiempo de ejecución empleando quizás algún otro algoritmo más rápido o mejorando el existente. Además, se podría mejorar el tiempo de ejecución dotando al taxi de una mayor velocidad.

Capítulo 5

Práctica: Aspiradora autónoma

En este capítulo se expondrá el desarrollo de una nueva práctica para la plataforma de JdeRobot-Academy, que se llama “Aspiradora autónoma”. Se aborda el desarrollo de su infraestructura, su componente académico correspondiente, así como el evaluador automático creado y la solución de referencia llevada a cabo.

5.1. Enunciado

El objetivo de esta práctica es que una aspiradora robótica sea capaz de limpiar de forma autónoma la mayor superficie posible en una casa. La aspiradora no tendrá ninguna forma de obtener su posición absoluta en el mundo, ya que esa es la situación en los modelos de gama media y baja existentes en el mercado de electrodomésticos robóticos. El único dato que conocerá la aspiradora es su orientación (fácilmente disponible en modelos reales con una brújula). Además, esta aspiradora posee un sensor láser, que le permite medir la distancia a la que se sitúan los obstáculos. Esta aspiradora tiene un actuador de movimiento que permite controlar su velocidad lineal y velocidad de giro.

En esta práctica el alumno deberá programar el algoritmo capaz de limpiar un gran porcentaje de la casa sin autolocalización. En la interfaz gráfica del componente académico se puede visualizar un mapa de la casa, así como la posición de la aspiradora en el mapa y los lugares por donde ha pasado. Este mapa no estará disponible para el algoritmo de control, únicamente se utiliza como visualización para el alumno y medida de resultados.

El algoritmo 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 controlar en todo momento el movimiento del robot y responder ante situaciones imprevistas.

5.2. Infraestructura

En este apartado se describirá el entorno creado en el simulador para realizar la práctica “Aspiradora autónoma”. Se detallará el robot que se ha utilizado, sus sensores y actuadores, su modelo dentro del simulador Gazebo, así como el escenario (una casa típica con varias habitaciones, comedor, mesas, sillas,puertas, etc) por el que se mueve este robot.

5.2.1. Modelo Roomba

El robot en el que se ha inspirado esta práctica es el Roomba de la serie 500, que fue comercializado por la empresa iRobot (en Estados Unidos). Esta aspiradora robótica está equipada con un conjunto de sensores con los que explorar sus alrededores y unos actuadores que le permiten moverse adecuadamente. Los modelos de Roomba de esta serie poseen sensores infrarrojos, un sensor detector de suciedad, un sensor detector de desniveles, y cuentan además con un *bumper*.

En el caso de la práctica, se empleó el modelo de *Roomba* de JdeRobot, que no tiene detector de desniveles debido a que el escenario de nuestra práctica no los contiene (no hay escaleras); y tampoco tiene detector de suciedad, puesto que no se ha simulado la suciedad. El objetivo de la práctica era hacer énfasis en el algoritmo de navegación. En nuestro modelo de *Roomba* tenemos un sensor *bumper*, que permite detectar los choques con objetos; y un sensor láser, que permite medir la distancia a los objetos. Este sensor láser es capaz de hacer un barrido de 180 grados, con precisión de 1 grado. Además, este robot posee sensores de orientación (la posición no se usará en la solución de referencia, a propósito).

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 *Roomba* de JdeRobot mide aproximadamente 330 mm de ancho, 90 mm de altura, y un peso de 2.5 kg.



Figura 5.1: Roomba de iRobot y modelo Roomba en Gazebo

En esta práctica se han utilizado cuatro *plugins* que ejercen de *drivers* de la aspiradora:

- **pose3di**: Los componentes harán uso de este *plugin* para obtener su posición en tiempo real, sólo se empleará la orientación.
- **motorsi**: Este *plugin* interactúa con el componente, dotándole de velocidad, tanto velocidad de tracción como velocidad de rotación.
- **Laseri**: Este *plugin* será usado por los componentes para obtener información de la distancia que hay hasta los obstáculos.
- **Bumperi**: El componente podrá usar este *plugin* para recolectar información acerca de su situación de colisión con otro objeto cualquiera.

5.2.1.1. Sensor láser

En la parte frontal del robot se ha instalado un sensor láser. Este sensor se utilizará ampliamente en la práctica. Está compuesto por un array de 180 medidas, que puede medir distancia alrededor de 180 grados en milímetros.

Una vez más la plataforma JdeRobot encapsula la complejidad de este sensor y nos devuelve los datos que ofrece el mismo, en forma de un array de 180 distancias.

5.2.1.2. Sensor bumper

El sensor *bumper* es un sensor de contacto que permite detectar una colisión con un objeto. Esto va a ser muy útil para realizar el algoritmo de esta práctica. Este sensor

comprueba el número de contactos con obstáculos del entorno, y si dicho número es mayor que cero es porque el robot ha chocado con algún objeto. La plataforma JdeRobot abstrae del funcionamiento interno, y permite conocer si ha habido algún choque devolviéndonos simplemente un 0 o un 1. Si el robot se ha chocado con algún objeto, entonces el *bumper* nos devolverá un 1. Por el contrario, si el robot no colisiona con ningún obstáculo dará como resultado un 0.

5.2.2. Modelo House_int2

Ha sido necesario crear un modelo de casa para que la aspiradora navegue en ella. Hemos empleado el modelo de casa que empleó Juan Navarro en su proyecto Fin de Carrera [16] en el mundo *GrannyAnnie.world* (Figura 5.2), pero hemos realizado algunas modificaciones.

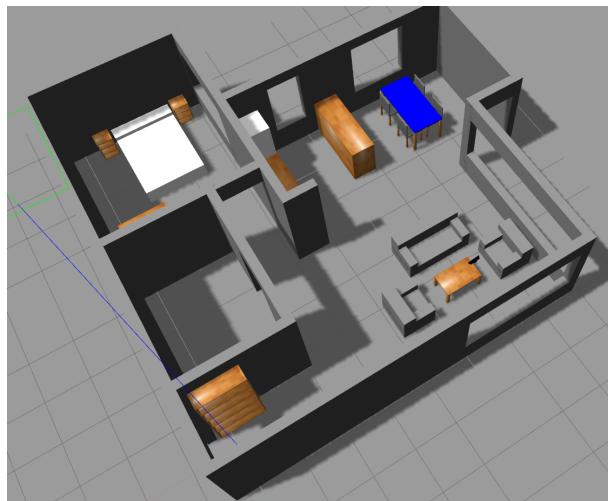


Figura 5.2: Mundo *GrannyAnnie.world* en Gazebo

Primero, había puertas que daban al exterior de la casa, entonces si pusiéramos a nuestra aspiradora a recorrer la casa se podría salir de la misma y no queremos que suceda esto. Segundo, había alguna pared que tenía mal la malla de colisiones, esto hacía que el robot pudiera atravesar algunas paredes. Se modificó la malla de colisiones en algunas zonas de la casa. Tercero, se modificaron las masas del inmobiliario incrementando su valor. La masa que tenían anteriormente no era suficiente y la aspiradora al colisionar con los muebles (mesas, sillones, etc) los desplazaba. El nuevo modelo de casa se llama “*house_int2*” y

lo podemos ver en la Figura 5.3, donde aparecen los cambios marcados con un círculo rojo.

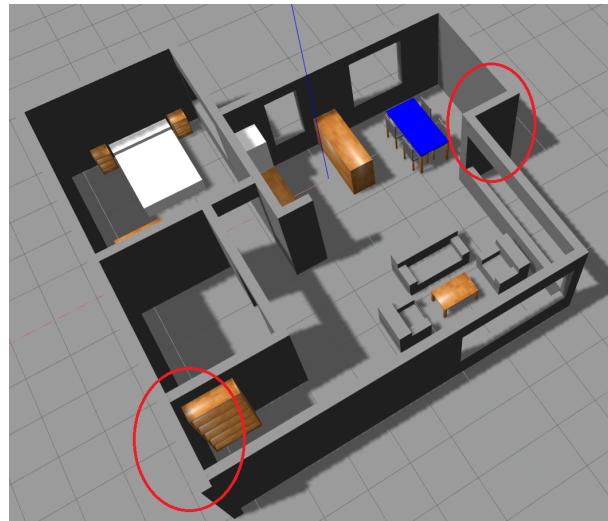


Figura 5.3: Modelo *house_int2* en Gazebo

5.2.3. Mundo de Gazebo

Si queremos ver el comportamiento de la aspiradora dentro de la casa, antes debemos crear un mundo de Gazebo que esté formado por el modelo de la casa “*house_int2*” y el modelo de aspiradora “*Roomba*”. Por este motivo se ha creado un mundo llamado “*Vacuum.world*”, donde también se incluyen luces. El archivo del mundo tiene el siguiente aspecto:

```
<?xml version="1.0" ?>
<sdf version='1.4'>
  <world name=' Vacuum '>
    <include>
      <uri>model://roomba</uri>
      <pose>-1 1.5 0 0 0 0</pose>
    </include>
    <include>
      <uri>model://house_int2</uri>
      <pose>0 0 0 0 0 0</pose>
    </include>
```

```
<include>
  <uri>model://ground_plane</uri>
</include>

<light name='sun' type='directional'>
  <cast_shadows>1</cast_shadows>
  <pose>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>

<scene>
  <ambient>0.4 0.4 0.4 1</ambient>
  <background>0.7 0.7 0.7 1</background>
  <shadows>1</shadows>
</scene>

<gui fullscreen='0'>
  <camera name='user_camera'>
    <pose>0.126197 6.13852 18.8314 0 1.08764 -2.14299</pose>
    <view_controller>orbit</view_controller>
  </camera>
</gui>
</world>
</sdf>
```

5.3. Componente académico

El componente académico creado resuelve varias funcionalidades auxiliares en la práctica: (a) ofrece una interfaz gráfica al usuario que le ayuda a depurar su código; (b) ofrece acceso a sensores y actuadores en forma de métodos simples (oculta el *middleware* de comunicaciones); (c) incluye código auxiliar que no es el foco del algoritmo y que ayuda a programar la solución. El componente deja todo preparado para que el estudiante sólo tenga que incluir su código retocando el método *execute* en el fichero *MyAlgorithm.py*.

Este componente ofrece al programador del algoritmo este API de sensores y actuadores:

- *pose3d.getYaw()*: Permite obtener la orientación del robot con respecto al mapa.
- *bumper.getBumperData().state*: Devuelve un 1 si el robot colisiona y un 0 si no ha chocado.
- *laser.getLaserData()*: Permite obtener los datos del sensor láser, que se compone de 180 pares de valores (0-180°, distancia en milímetros).
- *motors.sendV()*: Para establecer la velocidad lineal.
- *motors.sendW()*: Para establecer la velocidad de giro.

Este componente académico tiene un archivo de configuración (terminado con la extensión .cfg) que sirve para indicar los puertos que utilizan cada uno de los *plugins* que posee Roomba. Es necesario para que la aplicación pueda comunicarse con gazeboobserver. Tiene el siguiente aspecto:

```
VacuumCleaner.Motors.Proxy = Motors:default -h localhost -p 9003  
VacuumCleaner.Pose3D.Proxy = Pose3D:default -h localhost -p 9003  
VacuumCleaner.Laser.Proxy = Laser:default -h localhost -p 9003  
VacuumCleaner.Bumper.Proxy = Bumper:default -h localhost -p 9003  
VacuumCleaner.Motors.maxV = 5  
VacuumCleaner.Motors.maxW = 20
```

Tanto los motores, como la Pose3D, el láser y el *bumper* emplean el mismo puerto, el 9003. Gracias a cómo está configurado JdeRobot en su versión actual, se puede emplear

el mismo puerto. Además, se puede observar que se establece la velocidad de tracción y rotación máxima que tendrán los motores de la aspiradora.

El componente emplea dos hilos de ejecución para realizar diferentes tareas al mismo tiempo, ya que se divide el comportamiento en tareas más sencillas.

- Hilo de control: Es el encargado de actualizar los datos de los sensores y los actuadores a través de las interfaces ICE. El tiempo de refresco de este hilo es crucial, y tiene que ser muy corto, ya que este componente se encarga de establecer la velocidad y la dirección del robot en todo momento, así como de actualizar los datos de los sensores. Si este tiempo fuera muy grande, las decisiones que modifican la navegación del robot podrían ser incorrectas. El hilo de control de actuadores y sensores se actualiza cada vez que se actualiza la GUI, es decir, cada 50 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 la posición del robot en el mapa que se muestra en la interfaz en tiempo real. Este intervalo de tiempo es de 50 ms.

5.3.1. Interfaz gráfica

La interfaz gráfica de usuario (GUI) de la práctica sirve para representar información importante de ayuda para resolver adecuadamente el algoritmo. Se ha programado en Python con PyQt5. Se muestra en la Figura 5.4 y en ella, en la esquina superior izquierda el logo de JdeRobot.



Figura 5.4: Interfaz gráfica (GUI) de Aspiradora automática después de cierto tiempo de ejecución

Contiene también una imagen del mapa de la casa por la que navega Roomba. Este mapa es una imagen binaria, donde aparecen con un color negro (valor 0) los obstáculos, mientras que el suelo, que es la zona por donde podrá navegar la aspiradora, tiene un color blanco (valor 255). Esta imagen es un simple visor de la situación de la aspiradora, ya que este mapa no se proporciona al alumno para la solución de la práctica. En el mapa también aparecerá pintada la posición de Roomba como un triángulo (se pintan sus aristas) en color rojo. Esto permitirá saber dónde está situada nuestra aspiradora en el mapa, así como la orientación que tiene. También se pintarán en color azul las zonas de la casa por donde ya ha pasado la aspiradora. Esto ayudará a ver que hay por zonas por las que puede que haya pasado varias veces, mientras que por otras aún no habrá pasado.

Para pintar tanto el triángulo como las zonas por donde ha pasado el robot en el mapa hay que emplear un sistema de conversión del sistema de referencia del mundo (3D) al sistema de referencia del mapa (2D). Para ello se han empleado matrices de rotación y traslación, como sucedía en el Capítulo 4. En el caso de esta práctica los sistemas de referencia están situados de la siguiente forma:

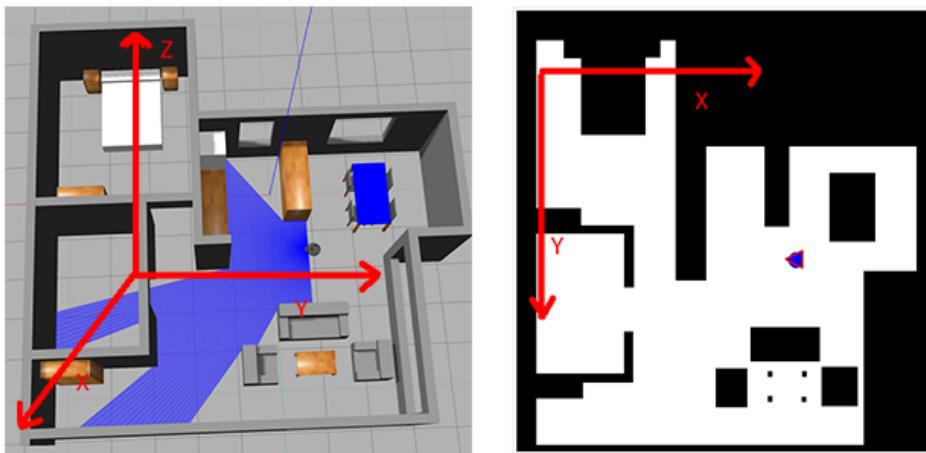


Figura 5.5: Sistema de referencia del mundo (izquierda) y sistema de referencia de la imagen (derecha)

Para saber dónde está situado nuestro robot en la imagen tenemos que pasar de una coordenada (x, y, z) en 3D a una coordenada (x', y') en 2D. En concreto, se aplica una rotación de π grados (será el ángulo α de la matriz) sobre el eje y. La traslación se realiza porque queremos tener el punto $(0,0)$ de la imagen en la esquina superior izquierda, y para que se corresponda correctamente cada punto de la imagen con cada punto del mundo. En este caso la traslación que se ha aplicado es de 0.6 (será tx) en el eje x; y -1 (será ty) en el eje y. La matriz de rotación y traslación sobre el eje y es la de la ecuación 5.1:

$$\begin{bmatrix} x' \\ y' \\ z' \\ 1 \end{bmatrix} = \begin{bmatrix} \cos(\alpha) & 0 & \sin(\alpha) & tx \\ 0 & 1 & 0 & ty \\ -\sin(\alpha) & 0 & \cos(\alpha) & tz \\ 0 & 0 & 0 & 1 \end{bmatrix} * \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} \quad (5.1)$$

De esta forma podemos pintar el triángulo, representación de la aspiradora, pasando la coordenada del mundo que nos devuelve el *pose3d* a coordenada en la imagen, la cual será el centro del triángulo que dibujaremos con una pequeña desviación. El triángulo es isósceles para mostrar cuál es la orientación del robot. La aspiradora estará orientada hacia el vértice de ángulo menor.

Esta rotación también sirve para pintar en azul los puntos del mapa por donde ha

pasado la aspiradora en cualquier momento. Se pintará un círculo en cada iteración dado que la aspiradora ocupa un determinado volumen. Los puntos por los que ha pasado la aspiradora se guardan en un array para saber por dónde ha pasado y pintarlos todos en cada iteración.

Por otra parte, en la esquina superior derecha de la GUI tenemos un dial bidimensional para teleoperar el robot. Este teleoperador controla las velocidades lineal y angular del robot. La velocidad lineal del robot se puede controlar moviendo el *joystick* en sentido vertical. Cuanto más subamos el *joystick* más velocidad tendrá el robot hacia delante, y si lo bajamos del todo más velocidad lineal tendrá el robot hacia atrás. La velocidad angular del robot se controla moviendo el *joystick* en sentido horizontal, según lo movamos a izquierda o a la derecha, el robot girará en un sentido u otro.

En la aplicación gráfica hay además dos botones importantes. El botón superior, que aparece con un símbolo de STOP, es el que emplearemos cuando teledirigimos a la aspiradora y queremos que pare en un punto y no siga navegando. El botón inferior, en el cual pone “*Run my algorithm*”, es el botón con el que le ordenaremos al componente que comience a ejecutar la solución que se ha programado en el fichero *MyAlgorithm.py*. Este botón cambia de color al pulsarlo. Si queremos que este código pare en un determinado momento, pulsaremos el mismo botón haciendo que pare; y si queremos reanudar su comportamiento lo volveremos a pulsar.

5.3.2. Gráfica de la derivada del porcentaje

Para ayudar con la comprensión del desempeño del algoritmo de navegación se ha incluido en el GUI una gráfica de la derivada del porcentaje de casa recorrida en función del tiempo. Con ella se puede comprobar cómo evoluciona el “rendimiento” de la navegación. Esta gráfica es una ventana opcional dentro del GUI del componente académico.

Podemos definir la derivada del porcentaje respecto del tiempo como la diferencia del porcentaje en cada unidad de diferencia del tiempo. En el eje horizontal se representa el tiempo, el cual se divide en 9 intervalos de tiempo. Cada intervalo de tiempo se corresponde con un intervalo de 100 segundos. Por lo tanto, en la práctica vamos a observar el

comportamiento del porcentaje a lo largo de 900 segundos, es decir, 15 minutos. En el eje vertical se representa la diferencia de porcentaje en cada intervalo de tiempo. Por ejemplo, en el primer intervalo de tiempo se representará la diferencia de porcentaje recorrido entre el segundo 100 de la práctica y el segundo 0, en el segundo intervalo la diferencia entre el segundo 200 y el 100, etc.

Cuando la aspiradora realiza el algoritmo de navegación puede que en ciertas ocasiones pase varias veces por una misma zona o se quede un rato atascada en ciertas zonas. Al principio cuando ejecutamos la práctica es muy sencillo que nuestra aspiradora recorra un gran porcentaje de casa, pero más adelante será más complicado que pase por zonas por las que no había pasado aún. Esto implica que posiblemente la derivada del porcentaje en función del tiempo sea mayor al comienzo de la ejecución de la práctica que por el final. Cada vez que se ejecute la práctica esta gráfica variará.

En la Figura 5.6 se puede observar un ejemplo de la gráfica de la derivada del porcentaje recorrido en función del tiempo.

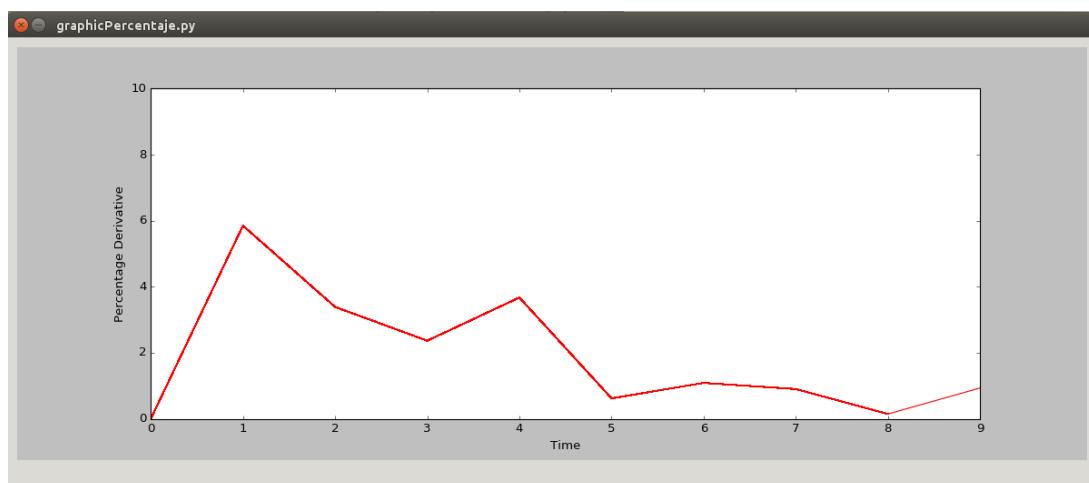


Figura 5.6: Gráfica de la derivada del porcentaje en función del tiempo

5.4. Solución de referencia

En este punto abordaremos una breve explicación sobre algoritmos que emplean aspiradoras robóticas disponibles en el mercado, una descripción de la técnica que emplea

Roomba de iRobot, y la solución de referencia que se ha desarrollado para esta práctica.

5.4.1. Algoritmos empleados por distintas aspiradoras robóticas

En los últimos 15 años han aparecido múltiples modelos de aspiradoras robóticas, las cuales son cada vez más innovadoras. Es importante tener en cuenta los algoritmos de navegación que incorpora cada uno de los modelos, ya que en función de estos algoritmos la aspiradora limpiará en menor o mayor tiempo la casa. Además, es importante tener en cuenta en estos algoritmos los posibles obstáculos con los que se encontrará la aspiradora al recorrer la casa. A continuación, podemos ver diferentes algoritmos de navegación que existen hoy en día en función del modelo:

- Uno de los primeros modelos de aspiradora Trilobite de Electrolux navega empleando ultrasonidos. Esta aspiradora usa la siguiente estrategia de navegación: explora el perímetro del entorno (siguiendo un comportamiento de sigue la pared); después de que la aspiradora llegue al punto de partida donde comenzó a recorrer la pared, el robot estima el tamaño del entorno y comienza con una trayectoria aleatoria.

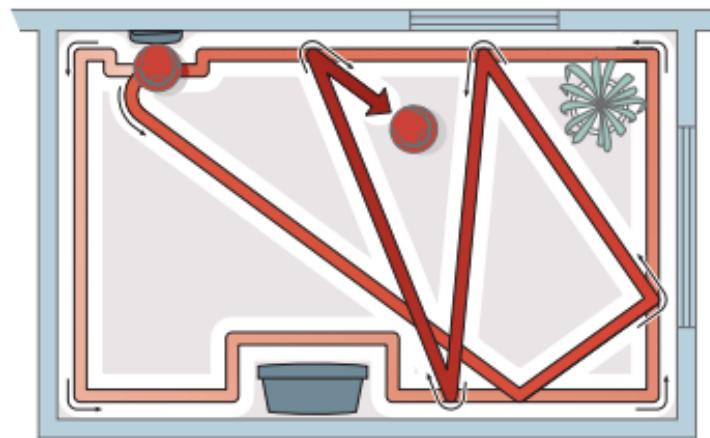


Figura 5.7: Patrón de navegación de la aspiradora Trilobite

- El sistema de navegación de la aspiradora robótica Xiaomi está guiado por un láser. Utiliza el algoritmo Simultaneous Localization and Mapping (SLAM) para generar un mapa de la casa en la que se encuentra y calcula patrones inteligentes para moverse a través de la casa. Xiaomi afirma que su modelo de aspiradora no solamente sigue un mapa y trata de limpiar el 100 % de la superficie del suelo de la

casa, sino que su aspiradora puede pensar y calcular el mejor patrón de limpieza para la casa. Cuando arranca Xiaomi, hace un escaneo de la zona circundante y divide la habitación en secciones de aproximadamente 4 por 4 metros. A continuación, el robot realiza un barrido del perímetro alrededor de esta área seccionada, trazando los obstáculos que puedan interponerse en el camino de la trayectoria. Después, “rellena” esta sección haciendo barridos horizontales.



Figura 5.8: Patrón de navegación de la aspiradora Xiaomi

- También es importante destacar la gama de robot LG Hombot. El aspirador LG Hombot Turbo es capaz de crear un mapa de la superficie, memorizando cada obstáculo, evitando el movimiento aleatorio y minimizando los choques.
- El más conocido de las aspiradoras robóticas es Roomba de iRobot. Roomba posee diferentes modelos, desde los más antiguos hasta los más innovadores. Existen grandes diferencias entre estos en cuanto a su patrón de limpieza. Los modelos de las series 500, 600, 700 y 800, calculan la ruta de limpieza óptima y determinan cuándo es necesario utilizar sus diversos modos de movimiento: giro en espiral, seguimiento de paredes, cruce de habitación. En cambio, los modelos de la serie 900 de Roomba, que tienen navegación iAdapt 2.0, se adaptan al entorno y disponen de localización visual para limpiar. A través de la localización visual, crea un mapa con referencias para recorrer toda la casa y saber por dónde ha pasado y dónde tiene que ir. Limpia de forma muy eficiente en áreas abiertas moviéndose en líneas paralelas mientras que, gracias a los sensores, adapta su trayectoria cuando se necesita.

Estos modelos que acabamos de mencionar son solamente unos pocos dentro de una

CAPÍTULO 5. ASPIRADORA AUTÓNOMA

amplia variedad de modelos de aspiradoras robóticas que existen hoy en día. Existen modelos de aspiradoras que emplean patrones aleatorios para limpiar la casa; y, por otro lado, hay otros modelos que inicialmente se construyen un mapa de la casa para recorrer la superficie de una forma más óptima. Para resolver el algoritmo sin autolocalización que se plantea en la práctica nos hemos basado en el algoritmo de navegación que siguen los modelos de las series 500, 600, 700 y 800 de Roomba, que no emplean un mapa de la casa y que detallamos a continuación.

Mientras Roomba de estas series está limpiando evita caerse por las escaleras o por un terreno empinado. Esto lo hace gracias a cuatro sensores infrarrojos en la parte inferior delantera. Estos sensores envían constantemente señales de infrarrojos, y si estas señales se pierden es porque el robot ha llegado a una zona de elevada pendiente por donde podría caerse.

Cuando Roomba choca con un objeto el *bumper* se retrae, gira y avanza hasta que encuentra una ruta despejada. Además, esta aspiradora tiene otro sensor de infrarrojos (llamado *Wall sensor*) situado en la parte delantera del robot. Este sensor permite que Roomba navegue muy cerca de las paredes, los objetos, etc.

Cuando presionamos el botón “*Clean*”, lo primero que hace Roomba es calcular el tamaño de la habitación según la información que recibe a través de sus sensores. Este robot envía una señal infrarroja y comprueba el tiempo que tarda en volver la señal. De esta manera calcula el tiempo que tendrá que pasar limpiando. Esto le permite optimizar la cobertura por habitación. El tiempo de funcionamiento que calcula depende del tamaño de la habitación y de la carga de la batería.

Roomba ejecuta su algoritmo 67 veces por segundo, obteniendo constantemente la información sobre su entorno y recomponiendo su trayectoria.

Roomba comienza a limpiar siguiendo un patrón de espiral. Esta espiral se va haciendo cada vez más grande, saliendo la espiral hacia fuera sobre un área más grande y más grande hasta que golpee un objeto. Cuando Roomba encuentre un objeto, seguirá a lo largo de este objeto recorriendo el perímetro de la habitación durante un periodo de

tiempo determinado. Cuando pase este periodo de tiempo, comenzará el modo cruce de la habitación, tratando de averiguar la máxima distancia que puede navegar sin chocar con un objeto. Sin embargo, si lleva un periodo de tiempo grande navegando sin chocarse con ningún obstáculo va a comenzar otra vez a realizar la espiral, porque supone que está en un amplio espacio abierto.

Estos patrones que eligieron los creadores de Roomba se basan en los algoritmos basados en los comportamientos de los animales cuando van buscando áreas de comida. Es bastante eficaz y robusto en situaciones del mundo real. Podemos observar en la Figura 5.9 el patrón de comportamiento que sigue Roomba.



Figura 5.9: Algoritmo de navegación de Roomba

5.4.2. Solución desarrollada

La solución se desarrolla en el fichero “*MyAlgorithm.py*”, en el método “*execute*”, el cual se ejecuta periódicamente. De esta forma, el pilotaje se llevará a cabo como un control reactivo, es decir, la aspiradora podrá comprobar los datos de sus sensores en cada instante y basándose en estos datos optar por realizar una acción u otra. No se realiza ningún tipo de planificación, sino que se lleva a cabo el pilotaje directamente, de modo reactivo.

El primer paso que lleva a cabo la solución desarrollada es realizar una navegación siguiendo un patrón en espiral (al igual que lo hacía la Roomba real). Si queremos realizar

un patrón en espiral debemos elegir qué tipo de espiral queremos, ya que existen numerosos tipos. Algunos de los tipos son: espiral de Arquímedes, logarítmica, hiperbólica, Fermat, de Durero, etc. En el caso de esta práctica se ha optado por realizar la espiral de Arquímedes, que es uniforme. Esta espiral se define como el lugar geométrico de un punto moviéndose a velocidad constante sobre una recta que gira sobre un punto de origen fijo a velocidad angular constante. En la Figura 5.10 se puede ver el patrón que sigue la espiral de Arquímedes.

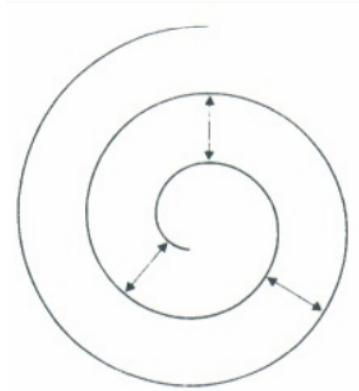


Figura 5.10: Espiral de Arquímedes

Para que la aspiradora pueda realizar un patrón de este tipo debe tener una velocidad angular constante y una velocidad lineal incremental. La velocidad lineal irá aumentando en cada iteración un determinado valor. Se ha optado por tomar como velocidad lineal la multiplicación de dos valores (los cuales inicialmente son valores muy pequeños). Un valor de radio inicial será multiplicado por una variable, la cual irá aumentando en cada iteración. El valor del radio inicial es de 0.1, mientras que la variable inicialmente posee un valor de 0.01. Esta variable será incrementada un 0.012 en cada iteración. De esta forma obtenemos un patrón en espiral similar a la espiral de Arquímedes.

Esta espiral se va haciendo cada vez más grande, saliendo hacia fuera sobre un área más grande. Roomba seguirá este patrón en espiral hasta que choque con un obstáculo. En la Figura 5.11 podemos ver el patrón que ha realizado la aspiradora.

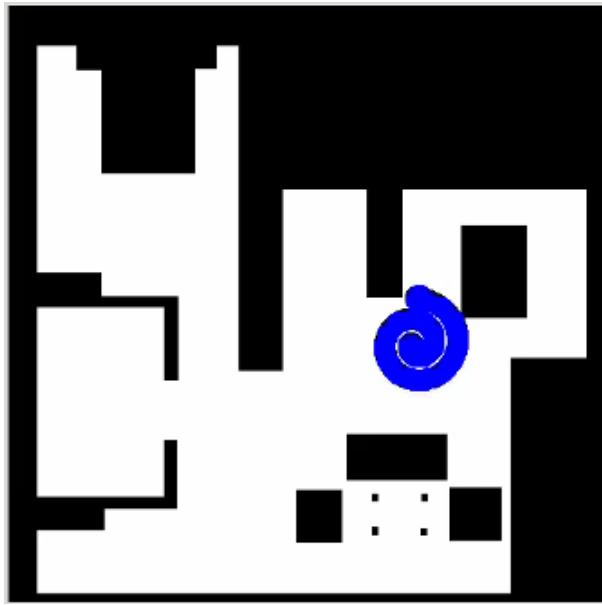


Figura 5.11: Espiral que realiza Roomba en la práctica

Cuando Roomba choca con un obstáculo, sigue a lo largo del contorno de este obstáculo recorriendo el perímetro de la casa durante un determinado periodo de tiempo, 300 segundos. Se ha optado por esta duración porque en las pruebas realizadas es el que mejores resultados presentaba.

Para seguir el perímetro se han empleado los datos que recoge el sensor láser que posee Roomba. En este subestado lo primero que hace Roomba es separarse un poco de la pared para no rozar todo el rato con ella. Después gira un poco hasta que encuentra que tiene la pared a su derecha. Para saber si tiene la pared a la derecha emplea el valor de distancia del ángulo 0 (rayo que está a la derecha de Roomba) y el ángulo 45 del array.

Tras saber que ya tenemos la pared a la derecha de la aspiradora, se pone a recorrer el perímetro. Para ello hay que comprobar continuamente el láser frontal (situado en el ángulo 90). Si la aspiradora está en un rincón, la distancia que obtiene ese láser será muy pequeña. Si se da el caso de que se encuentra en un rincón, Roomba girará a la izquierda 90 grados.

Otro dato que comprobará es si el láser del ángulo 135 (más a la izquierda) tiene una distancia muy pequeña y además ha detectado un choque. Esto indica que la aspiradora

CAPÍTULO 5. ASPIRADORA AUTÓNOMA

se ha quedado enganchada en algún hueco. Si se da este caso la aspiradora retrocede un poco y gira hacia la izquierda 90 grados.

Si la aspiradora no se encuentra en un rincón y no se ha quedado atascada en algún hueco, entonces recorrerá el perímetro siguiendo la pared. Para ello se comprueba el láser derecho (ángulo 0) y el láser del ángulo 45. Este algoritmo se realiza de forma reactiva comprobando continuamente los datos y tomando decisiones en base a ellos. Si la distancia a la pared es demasiado pequeña es que estamos muy pegados a la pared. En este caso la aspiradora gira un poco a la izquierda para no chocarse. Si el valor de distancia es muy grande, entonces gira un poco a la derecha para no desviarse de la pared. Si la distancia a la pared no es ni muy pequeña ni muy grande es que podemos seguir rectos tomando una velocidad lineal constante y una velocidad angular nula.

Teniendo en cuenta todas estas situaciones que se pueden dar al recorrer la pared, la aspiradora contornea el perímetro correctamente. En la Figura 5.12 se puede ver cómo después de realizar el patrón en espiral, Roomba ha realizado el perímetro durante un intervalo de tiempo.

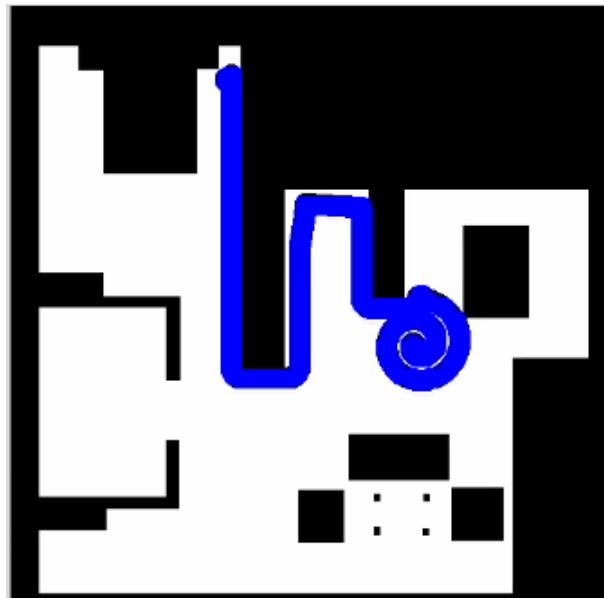


Figura 5.12: Perímetro que recorre Roomba en la práctica

En el siguiente subestado se utiliza un algoritmo aleatorio, que realiza un cruce de

habitación (similar a la Roomba real). Lo que hace en este caso Roomba es comprobar continuamente si la aspiradora ha chocado o no en cada iteración. Si ha chocado entonces retrocede un poco para apartarse de la pared, y calcula un ángulo aleatorio y el signo aleatoriamente. Este ángulo es el que girará Roomba para tomar una nueva orientación. El signo aleatorio es para que nuestra aspiradora gire algunas veces hacia la izquierda y otras veces a la derecha. Para elegir el signo se emplea una función que proporciona Python para calcular aleatoriamente números enteros. En este caso se define un número entre 0 y 1. Si el signo sale 0, entonces Roomba gira a la derecha; si por el contrario es 1, gira a la izquierda.

Tras calcular el ángulo y el signo aleatoriamente, Roomba gira hasta haber rotado el ángulo que se le ha indicado. Cuando ha realizado el giro, la aspiradora toma como velocidad de giro 0 y una velocidad lineal constante. De esta forma Roomba irá recta hacia la orientación que se haya quedado después del giro. Roomba seguirá la recta hasta que vuelva a chocar. Cuando choque volverá a realizar el algoritmo aleatorio hasta finalizar el tiempo de la práctica.

5.5. Evaluador automático

Adicionalmente se ha creado un evaluador automático que en función de diferentes parámetros califica el algoritmo que ha programado el alumno. El evaluador automático muestra en una interfaz gráfica diferentes parámetros, así como la nota final. Se ha creado utilizando PyQt5 al igual que el componente académico. Para programarlo se han creado clases diferentes para cada parámetro que queramos mostrar en la interfaz. Estas clases serán instanciadas en una clase principal (llamada *MainWindow*), que contiene la ventana principal del evaluador.

En la esquina superior izquierda de la Figura 5.13, tiene un visor que muestra una barra de progreso. En esta barra se pinta en rojo el porcentaje de superficie recorrida por la aspiradora en cada momento. Encima de esta barra se mostrará esa misma información de modo numérico. El porcentaje de superficie recorrida se calcula teniendo en cuenta que el 100 % es el suelo donde no hay obstáculos situados. Es decir, todos los píxeles en blanco

CAPÍTULO 5. ASPIRADORA AUTÓNOMA

que aparecen en el mapa que muestra la interfaz.

En la esquina inferior izquierda se muestra la imagen del mapa de la casa similar a la usada en el componente académico. En la esquina superior derecha, tenemos un reloj digital, donde se muestran los segundos restantes hasta el final de la prueba. Este visor comienza mostrando 900 segundos (15 minutos) y cada vez que ha pasado un segundo lo va descontando de este valor. Se ha considerado un intervalo de tiempo adecuado para evaluar la práctica, ya que si pusieramos un intervalo mayor de tiempo sería un periodo excesivo y lento para evaluar la práctica. Si ponemos un intervalo de tiempo menor a 15 minutos no se podría evaluar la práctica con claridad, ya que en los primeros minutos de la práctica es muy sencillo recorrer bastante porcentaje de suelo comparando con los minutos posteriores.

Debajo de este visor del tiempo digital se muestra un visor de un reloj analógico. En este reloj analógico sí que avanza el tiempo.

Por último, hay que mencionar que cuando terminen los 900 segundos, a la derecha del reloj digital aparecerá un mensaje con la nota que ha obtenido el alumno. Esta nota se calculará en función del porcentaje recorrido por la aspiradora. Se ha establecido un porcentaje máximo para estos 15 minutos, ya que no es posible recorrer toda la casa en este tiempo. Si la aspiradora recorre un 30 % o más, el alumno obtendrá una nota de 10 puntos. Si por el contrario, la aspiradora recorre menos porcentaje, entonces se calcula la nota haciendo una regla de tres sabiendo que una nota de 10 es un 30 %.

En la Figura 5.13, podremos ver el evaluador automático durante el pilotaje.

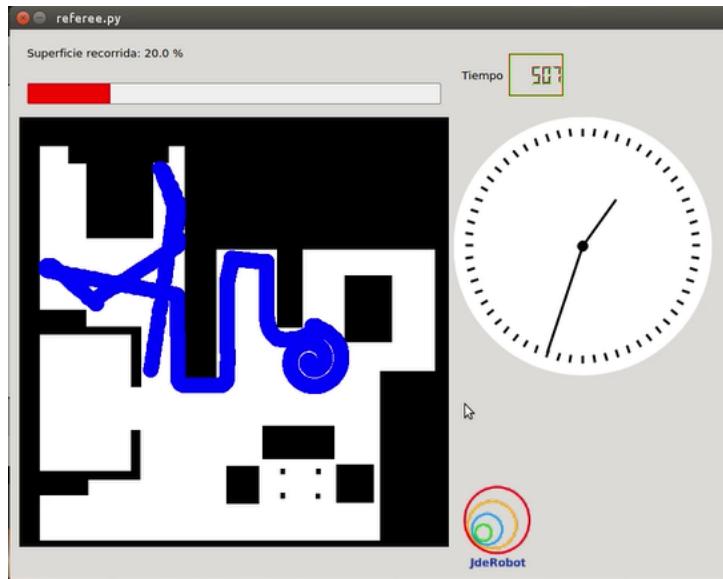


Figura 5.13: Evaluador automático durante la ejecución de la práctica

5.6. Experimentación

5.6.1. Ejecución típica

La práctica se ejecuta abriendo tres terminales y escribiendo lo siguiente en cada uno de ellos:

1. Lanzar Gazebo: gazebo Vacuum.world
- 1b. Si el ordenador que se emplea no tiene muchos recursos se puede arrancar el simulador sin interfaz gráfica: gzserver Vacuum.world
2. Ejecutar el componente académico: python2 vacuumCleaner.py – –Ice.Config = vacuumCleaner.cfg
3. Ejecutar el evaluador automático: python2 referee.py – –Ice.Config=vacuumCleaner.cfg

En la Figura 5.14 se puede observar el resultado del evaluador automático al finalizar la prueba tras evaluar una ejecución típica de la solución de referencia. Se puede observar que se consigue recorrer el 28 % de la superficie, obteniendo una nota de 9.33. Además, al emplear en el último subestado de la solución un algoritmo aleatorio podemos ver

que la aspiradora no recorre únicamente una habitación, sino que navega por diferentes habitaciones. Una ejecución típica se puede ver en este video ¹.

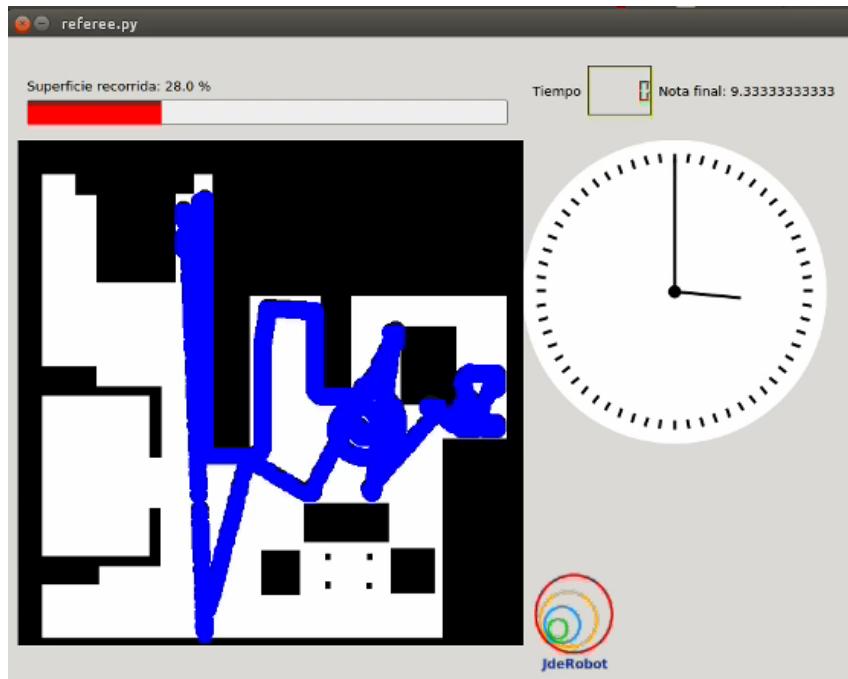


Figura 5.14: Evaluador automático con la nota final

5.6.2. Ajuste del tiempo de navegación perimetral

Un punto importante de la solución de referencia es decidir cuál es el periodo de tiempo que la aspiradora se mantendrá recorriendo el perímetro. En la práctica se ha optado por 300 segundos. Para elegir un periodo de tiempo se han hecho pruebas con diferentes valores. Se han realizado tres pruebas por cada periodo de tiempo para sacar una media del porcentaje recorrido y elegir el periodo con mayor porcentaje, aunque al ser aleatorio, si hiciéramos la prueba mil veces puede que saliera otro resultado, pero hemos elegido este periodo de 300 segundos en base a estas pruebas. Los periodos de tiempo probados son 200, 300, 400, 500 y 600 segundos. Se puede ver en la Tabla 5.1 los resultados obtenidos para cada periodo.

¹https://www.youtube.com/watch?time_continue=106&v=ThTXrqTDJ_A

Tabla 5.1: Resultados con diferentes tiempos de recorrido del perímetro

| | 1º Prueba | | 2º Prueba | | 3º Prueba | | Media | |
|-----|-----------|------------|-----------|------------|-----------|------------|-------|------------|
| | Nota | Porcentaje | Nota | Porcentaje | Nota | Porcentaje | Nota | Porcentaje |
| 200 | 6.66 | 20 % | 10 | 33 % | 6 | 18 % | 7.55 | 23.66 % |
| 300 | 9.33 | 28 % | 8 | 24 % | 9 | 27 % | 8.77 | 26.33 % |
| 400 | 9.33 | 28 % | 8 | 14 % | 8.66 | 26 % | 8.66 | 26 % |
| 500 | 6.66 | 20 % | 8.33 | 25 % | 6 | 18 % | 7 | 21.66 % |
| 600 | 7.33 | 22 % | 7 | 21 % | 7.33 | 22 % | 7.22 | 21.66 % |

En el caso del periodo de 200 segundos se puede ver que el 33 % es un gran resultado, que nos da como nota de calificación 10. Sin embargo, el resto de resultados obtenidos no son muy buenos, lo que hace que baje notablemente la nota. En la Figura 5.15 se puede ver la imagen del mapa con la superficie recorrida en azul para las tres pruebas mencionadas. En el caso de la segunda imagen se puede ver que hay más superficie recorrida.

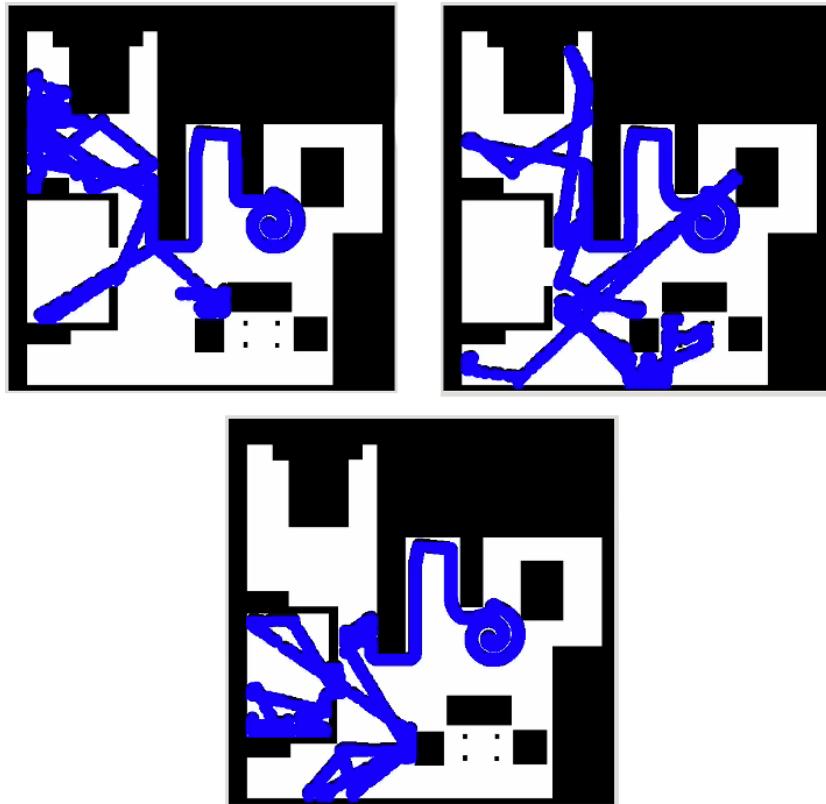


Figura 5.15: Superficie recorrida de las pruebas de 200 segundos de perímetro

En la prueba del periodo de 300 segundos, vemos que tanto la nota como el porcentaje, es mayor en este caso que en el de 200 segundos. En este caso no hemos obtenido en ninguna prueba individual una nota de 10 como en el periodo de 200 segundos, pero la media es mejor. En la Figura 5.16 vemos el mapa con la superficie recorrida para las tres pruebas realizadas.

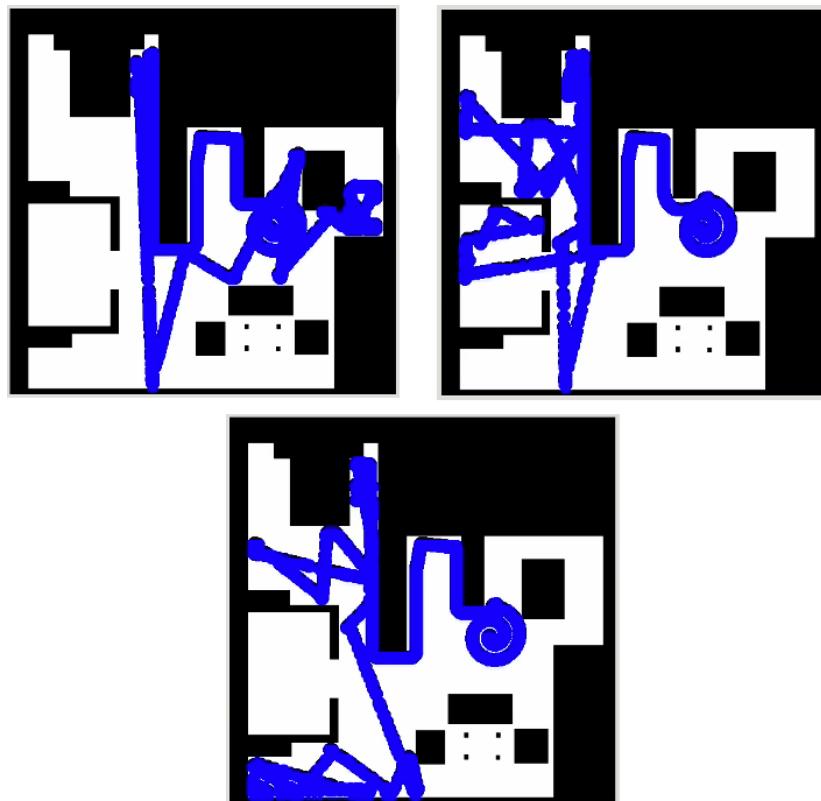


Figura 5.16: Superficie recorrida de las pruebas de 300 segundos de perímetro

Cuando se han realizado pruebas con un periodo de 400 segundos para recorrer el perímetro, se han logrado resultados muy similares al periodo de 300 segundos, por lo que un periodo de 400 segundos también podría haberse empleado en la práctica, ya que da buenos resultados.

En el caso de emplear un periodo de 500 segundos, los resultados que se obtienen no son tan buenos como en los casos anteriores, por lo que se descartó usar este intervalo de tiempo.

Por último, empleando un intervalo de tiempo de 600 segundos los resultados que se

consiguen son similares a los obtenidos con un intervalo de 500 segundos.

5.6.3. Experimentos de larga duración

Si tuviéramos una casa sin obstáculos, sin paredes y sin ningún hueco donde se pudiera quedar atascada la aspiradora, entonces el porcentaje que podría recorrer la aspiradora sería mucho mayor. Esto se debe a que estos obstáculos hacen que la aspiradora se quede mucho tiempo en algunas zonas y no recorra otras. Es posible que la aspiradora navegue por una zona varias veces, y, sin embargo, no recorra otras zonas en estos 15 minutos. Para comprobar la diferencia de porcentaje recorrido, se han hecho tres pruebas de 45 minutos, con el intervalo de perímetro de 300 segundos, por lo que la aspiradora realizará un algoritmo aleatorio durante mucho tiempo.

En las pruebas que se han realizado se han obtenido unos resultados de porcentaje del 49 %, 42 % y 45 %. Estos porcentajes dan una media de 45.33 %. Por lo tanto, se puede ver que la media está cercana a la mitad de porcentaje de la casa. En la Figura 5.17 se puede ver el mapa con la superficie recorrida en las tres pruebas. En estas imágenes se puede observar que en algunas ocasiones hay algunas habitaciones recorridas casi en su totalidad.

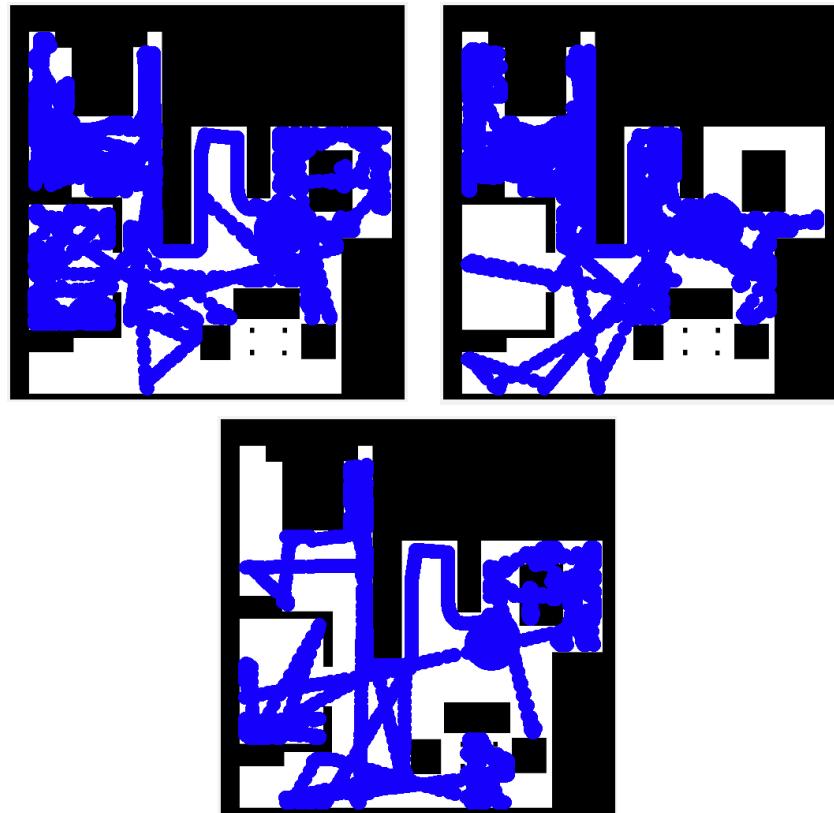


Figura 5.17: Superficie recorrida de las pruebas de 45 minutos

5.6.4. Ley de los rendimientos decrecientes

Se ha estudiado el porcentaje recorrido en función de diferentes duraciones de tiempo. Viendo los porcentajes obtenidos en la prueba de 45 minutos, se podría decir que si la aspiradora navevara durante hora y media podría recorrer el 90 % o 100 % dependiendo de la ocasión. Pero no es así, la aspiradora tardaría más tiempo, ya que cada vez es más complicado alcanzar nuevos puntos en la casa por donde no haya pasado. Se han realizado varias pruebas de hora y media para comprobar el porcentaje recorrido. Hemos comprobado que existe una ley de rendimiento decreciente, a mayor tiempo de ejecución del algoritmo menor será el rendimiento en cuanto a porcentaje recorrido. Este aspecto lo podemos ver en la Tabla 5.2, donde comparamos las pruebas de 15, 45 y 90 minutos. En la Figura 5.18 se puede ver la superficie recorrida en las tres pruebas de 90 minutos.

Tabla 5.2: Resultados Pruebas de 15, 45 y 90 minutos

| | 1º Prueba | 2º Prueba | 3º Prueba |
|----|------------|------------|------------|
| | Porcentaje | Porcentaje | Porcentaje |
| 15 | 28 % | 24 % | 27 % |
| 45 | 49 % | 42 % | 45 % |
| 90 | 53 % | 60 % | 55 % |

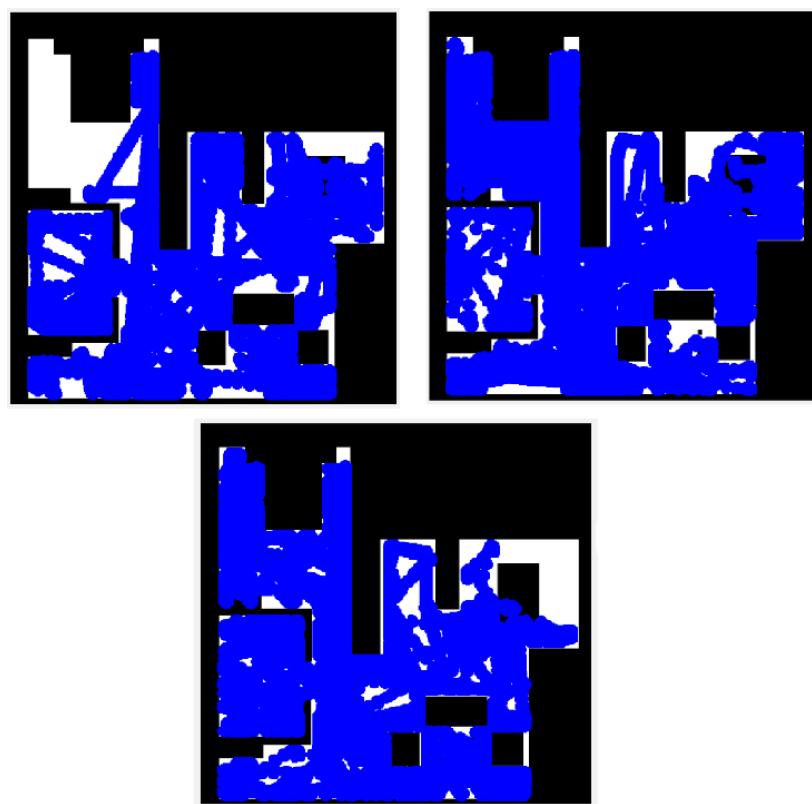


Figura 5.18: Superficie recorrida de las pruebas de 90 minutos

Capítulo 6

Práctica: Aparcamiento Automático

En este capítulo se describirá el desarrollo de una nueva práctica para el entorno JdeRobot-Academy, que se denomina “Aparcamiento automático”. En este capítulo se aborda el desarrollo de su infraestructura, su componente académico, así como el evaluador automático que se ha creado y la solución de referencia realizada.

6.1. Enunciado

El problema abordado en esta práctica es que un coche autónomo sea capaz de aparcar en una plaza de aparcamiento en línea sin chocar con los coches que están delante y detrás de la plaza libre de aparcamiento. El coche dispondrá de un GPS que le proporciona una estimación de su posición en el entorno en que se encuentra. Este coche tiene también tres sensores láser mediante los cuales podrá obtener información acerca del entorno por el que se mueve. Este vehículo posee un actuador de movimiento que se basa en la velocidad de tracción y la velocidad de giro.

El alumno deberá programar el comportamiento de este vehículo autónomo para que pueda aparcar. En la interfaz gráfica del componente académico desarrollado se puede visualizar un mapa local del entorno por el que se mueve el coche, así como la posición de este vehículo en el mapa. En esta interfaz, hay un visor que muestra gráficamente lo que “ve” cada láser del entorno. Esta interfaz gráfica facilitará al alumno la resolución de la práctica.

El algoritmo propuesto como solución de referencia responde a un control reactivo,

donde en cada momento el coche actuará en función de los datos de los sensores o de variables internas. El control reactivo permitirá controlar en todo momento el movimiento del vehículo de forma que pueda responder ante situaciones imprevistas.

6.2. Infraestructura

En este apartado se describirá el entorno que se ha creado para poder realizar la práctica “Aparcamiento automático”. Se comenzará describiendo el modelo de coche empleado, así como los sensores y actuadores que posee el mismo. Despues, se realizará una explicación del entorno simulado por el cual se moverá el coche.

6.2.1. Modelo Taxi_Holo_Laser

El robot que se ha empleado en esta práctica es un nuevo modelo de coche basado en el modelo de taxi que se empleó en la práctica “global_navigation”. Este modelo de coche se denomina *taxi_holo_Laser* y lo podemos ver en Figura 6.1. Posee sensores GPS que le permiten saber cuál es su posición en todo momento; tiene tres sensores láser, uno se sitúa en la parte frontal del coche, otro en la parte trasera, y el último a la derecha del coche. También incorpora motores que le permiten moverse por el escenario de manera adecuada.

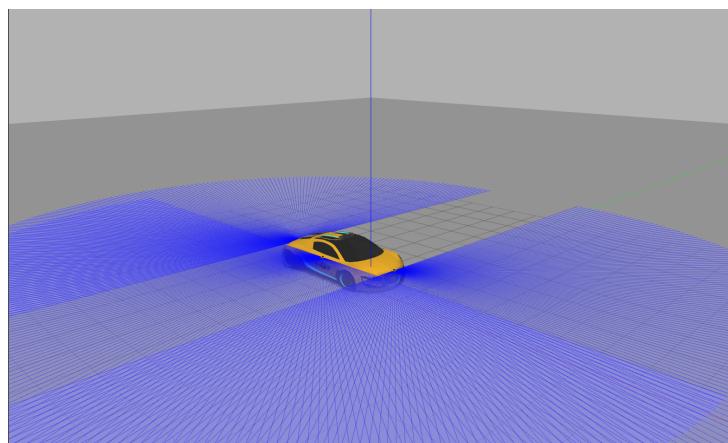


Figura 6.1: Modelo taxi_holo_Laser

En esta práctica se han empleado los *plugins*:

- *holoCarPose3D*: Los componentes emplean este *plugin* para obtener la posición del coche en tiempo real.
- *holoCarMotors*: El componente académico interactúa con este *plugin*. Este *plugin* permite dotar al componente de velocidad, tanto velocidad de tracción como velocidad de rotación.
- *laser*: Este *plugin* será empleado por los componentes para obtener datos de la distancia que hay hasta los obstáculos.

6.2.1.1. Sensores láser

En este taxi se han instalado tres sensores láser. Un sensor se ha colocado en la parte frontal del vehículo, otro en la parte trasera, y el último en la parte derecha del coche. Estos sensores serán empleados en el algoritmo de la práctica. Los sensores láser están formados por un array de 180 medidas, al igual que los empleados en el Capítulo 5.

6.2.2. Modelo acera

El objetivo de la práctica es que el coche sea capaz de aparcar de forma autónoma. Para ello es necesario crear un entorno (los modelos en gazebo) donde se moverá el coche. El coche deberá aparcar paralelo a una acera, por lo tanto, se ha creado el modelo “acera”. Este modelo se puede ver en la Figura 6.2.

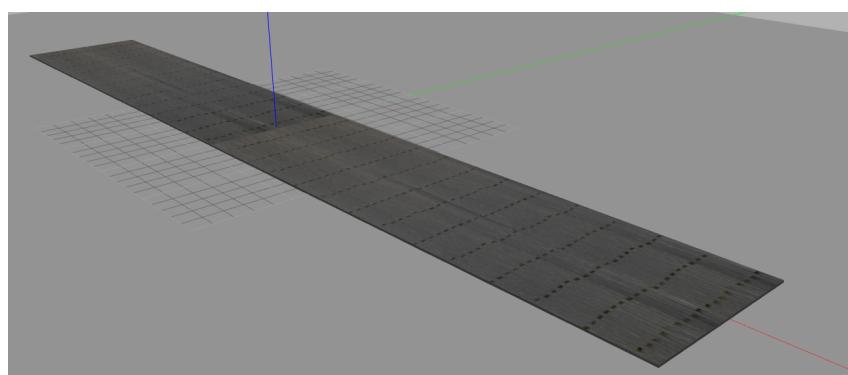


Figura 6.2: Modelo acera

6.2.3. Modelo carNoMotor

El coche deberá aparcar entre dos vehículos, y habrá más coches estacionados. Por este motivo se ha creado el modelo “*carNoMotor*”, el cual no posee motores, ya que queremos que esté estacionado. Este modelo se utilizará varias veces al crear el mundo de Gazebo. En la Figura 6.3, podemos ver este modelo.

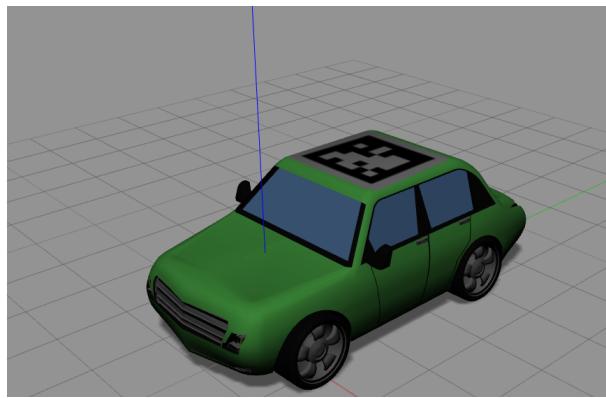


Figura 6.3: Modelo carNoMotor

6.2.4. Mundo de Gazebo

Este mundo estará formado por un modelo de carretera (*road*) que tiene Gazebo, poseerá dos aceras para lo cual emplearemos el modelo “*acera*”, además se emplearán varios coches del modelo “*carNoMotor*” que estarán aparcados; y, por último, se incluirá el modelo del coche (*taxis_holo_Laser*), que ejecutará la solución que le indiquemos. Para tener este escenario se ha creado un mundo en Gazebo llamado “*autopark.world*”:

```
<?xml version="1.0"?>
<sdf version="1.4">
  <world name="default">

    <scene>
      <grid>true</grid>
    </scene>

    <!-- A global light source -->
    <include>
```

```
<uri>model://sun</uri>
</include>

<!-- Ground -->
<include>
  <uri>model://ground_plane_sincolor</uri>
</include>
<include>
  <uri>model://acera</uri>
  <pose>5 9 0 0 0 0</pose>
</include>
<include>
  <uri>model://acera</uri>
  <pose>5 -9 0 0 0 0</pose>
</include>

<!-- A taxi with lasers-->
<include>
  <uri>model://taxi_holo_Laser</uri>
  <pose>-7 2.5 0 0 0 0</pose>
</include>

<!-- Cars -->
<include>
  <uri>model://carNoMotor</uri>
  <pose>-20 -3 0 0 0 1.57</pose>
</include>
<include>
  <uri>model://carNoMotor</uri>
  <pose>-13.5 -3 0 0 0 1.57</pose>
</include>
<include>
  <uri>model://carNoMotor</uri>
  <pose>-7 -3 0 0 0 1.57</pose>
</include>
```

```
<include>
  <uri>model://carNoMotor</uri>
  <pose>0.5 -3 0 0 0 1.57</pose>
</include>
<include>
  <uri>model://carNoMotor</uri>
  <pose>14 -3 0 0 0 1.57</pose>
</include>
<include>
  <uri>model://carNoMotor</uri>
  <pose>21.5 -3 0 0 0 1.57</pose>
</include>
<include>
  <uri>model://carNoMotor</uri>
  <pose>29 -3 0 0 0 1.57</pose>
</include>

<!-- A Road -->
<road name="my_road_1">
  <width>10</width>
  <point>-25 0 0.02</point>
  <point>35 0 0.02</point>
</road>

</world>
</sdf>
```

En la Figura 6.4 podemos ver cuál es el aspecto que tiene el mundo que hemos creado en Gazebo.

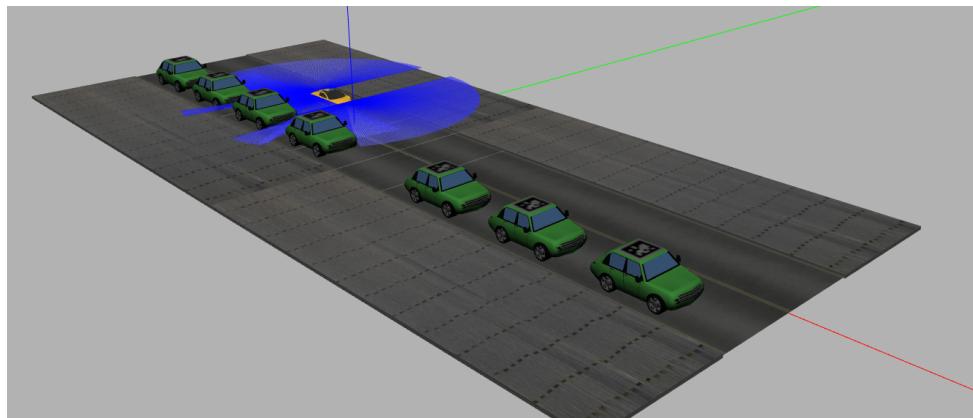


Figura 6.4: Mundo autopark.world en Gazebo

6.3. Componente Académico

El componente académico desarrollado resuelve diversas funcionalidades en la práctica: (a) ofrece una interfaz gráfica al usuario, que le ayuda a depurar su código; (b) ofrece acceso a sensores y actuadores en forma de métodos simples (oculta el *middleware* de comunicaciones); (c) incluye código auxiliar que no es el centro del algoritmo y que ayuda a programar la solución. El componente deja todo listo para que el estudiante sólo tenga que incorporar su código rellenando el método *execute* en el fichero *MyAlgorithm.py*.

Este componente ofrece al programador del algoritmo este API de sensores y actuadores:

- *pose3d.getX()*: Permite obtener la posición del robot en el eje X.
- *pose3d.getY()*: Permite obtener la posición del robot en el eje Y.
- *pose3d.getYaw()*: Permite obtener la orientación del robot con respecto al mapa.
- *laser.getLaserData()*: Permite obtener los datos del sensor láser, que se compone de 180 pares de valores (0-180°, distancia en milímetros). Esta función se empleará en cada uno de los láseres para obtener sus datos.
- *motors.sendV()*: Para establecer la velocidad lineal.
- *motors.sendW()*: Para establecer la velocidad de giro.

Es necesario crear un archivo de configuración en la práctica similar a los de las prácticas anteriores:

```
Autopark.Motors.Proxy = Motors:default -h localhost -p 9999
Autopark.Pose3D.Proxy = Pose3D:default -h localhost -p 9989
Autopark.Laser1.Proxy = Laser:default -h localhost -p 8996
Autopark.Laser2.Proxy = Laser:default -h localhost -p 8997
Autopark.Laser3.Proxy = Laser:default -h localhost -p 8998

Autopark.Motors.maxV = 250
Autopark.Motors.maxW = 20
```

Los motores emplean el puerto 9999, el Pose3D emplea el puerto 9989, y los sensores láser utilizan los puertos 8996, 8997, 8998. En este archivo se indica también la velocidad lineal máxima y la velocidad angular máxima.

En la práctica el comportamiento se divide en varias tareas. Para realizar estas tareas simultáneamente, empleamos dos hilos de ejecución:

- Hilo de control: Este hilo se encarga de actualizar los datos de los sensores y los actuadores a través de las interfaces ICE. El tiempo de actualización de este hilo es muy importante, ya que este componente establece la velocidad y la dirección del robot en todo momento. Este intervalo de actualización debe ser un periodo de tiempo muy corto. Si fuera muy grande, las decisiones que modifican la trayectoria del robot podrían influir en su comportamiento, haciendo que la trayectoria sea incorrecta. En esta práctica el hilo de control de actuadores y sensores se actualizará cada vez que se actualice la GUI, es decir, cada 50 ms.
- Hilo de la interfaz gráfica de usuario (GUI): Se encarga de actualizar la interfaz gráfica. El intervalo de actualización de la interfaz gráfica debe ser corto, puesto que se tiene que mostrar la posición del robot en el mapa que muestra la interfaz en tiempo real. El intervalo es de 50 ms.

6.3.1. Interfaz gráfica

La interfaz gráfica de usuario (GUI) se emplea para representar información que pueda ayudar a resolver el algoritmo planteado. Esta interfaz sirve para ejecutar el código que da solución a la práctica. Esta interfaz es programada en PyQt5.

Esta GUI (Figura 6.5) contiene un lienzo en blanco donde se ha pintado parte del mundo de Gazebo, así como la posición del coche. En concreto, hay ciertas partes pintadas en negro, que se corresponden con las aceras y los coches aparcados, es decir, los obstáculos. Por el contrario, las zonas que aparecen en blanco es donde no hay obstáculos, es decir, la carretera sin objetos. El coche aparece representado mediante un rectángulo amarillo con ruedas en negro. Este rectángulo también rota cuando el coche lo hace, de esta forma se puede visualizar adecuadamente su comportamiento. En este pequeño mapa local del escenario podemos ver que aparecen pintadas de verde las aristas de un rectángulo, el cual se corresponde con la posición ideal que debería conseguir el coche al aparcar. También podemos observar unos puntos rojos que aparecen consecutivamente, los cuales se corresponden con la estela del coche, es decir, las posiciones que ha tenido recientemente. Pero todas las posiciones que tiene el coche no se pintan, solamente las últimas. Hemos creado un array de cierto tamaño para representar únicamente unos cuantos puntos y no todos. Es un “array circular”, se van añadiendo las nuevas posiciones por el final del array y cuando llega a un cierto tamaño vamos borrando la primera posición del array y desplazando el contenido de sus posiciones a una posición anterior.

En la zona central del GUI se representa el coche mediante un rectángulo amarillo (con sus ruedas en negro), y los datos que ofrecen los sensores láser. Cada sensor láser se ha pintado de un color diferente para distinguirlos. Si comparamos esta representación con el mundo de Gazebo se podría ver que la representación es idéntica a cómo se ven los sensores láser en Gazebo. Para representar cada uno de los sensores en la posición adecuada se han empleado matrices de rotación y traslación al igual que sucedía en otras prácticas.

A la derecha del GUI, tenemos un teleoperador para mover al robot. Este teleoperador controla las velocidades lineal y angular del robot.

En la interfaz aparecen dos botones. El botón de debajo del mapa permitirá ejecutar el código de solución a la práctica. También permitirá parar el código de la solución, de forma que el coche se quede parado donde estuviera situado. El botón que aparece debajo de la gráfica con los sensores láser y el teleoperador, permitirá detener al robot si lo estamos teledirigiendo con el teleoperador.

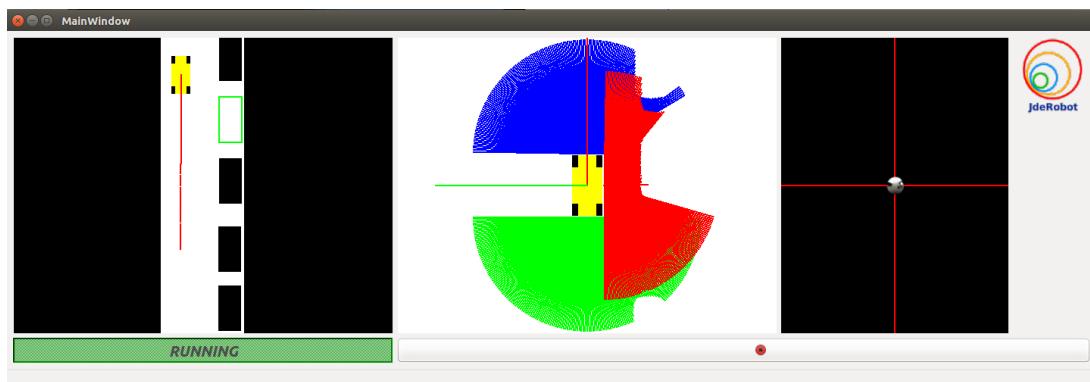


Figura 6.5: GUI del Autopark

6.4. Solución de referencia

En esta sección se describirá cómo aparcan los coches autónomos reales, así como la solución “ad hoc” que se ha desarrollado. Se exploran qué algoritmos han diseñado los fabricantes de coches para equipar a sus vehículos con esa capacidad (Volkswagen Tiguan...).

6.4.1. Algoritmos empleados por los coches autónomos reales

Durante la conducción, una de las maniobras más complicadas es el aparcamiento del vehículo, en especial en ciertas zonas donde los huecos disponibles son bastante escasos. En la actualidad existen distintos avances tecnológicos para facilitar el aparcamiento al conductor.

Uno de los primeros prototipos de coche con aparcamiento autónomo fue producido en Francia, hace alrededor de 20 años por el INRIA (Institut National de Recherche en Informatique et en Automatique). Fue el primero en realizar un aparcamiento autónomo

en paralelo. Hoy en día, cada vez más fabricantes de vehículos incorporan tecnologías de conducción autónoma. Se ha definido un estándar, es J3016, que establece cinco niveles de conducción autónoma según la capacidad del vehículo. Además, existe el nivel 0, que no tiene capacidad autónoma.

- Nivel 0: No hay automatización de la conducción. Las tareas de conducción son realizadas en su totalidad por el conductor.
- Nivel 1: Asistencia al conductor. El vehículo tiene algún sistema de automatización de la conducción (control de crucero, autoaparcamiento), ya sea para el control de movimiento longitudinal o el movimiento lateral, pero no ambas cosas al mismo tiempo. El conductor realiza el resto de tareas de conducción, por lo que el conductor debe estar siempre atento.
- Nivel 2: Automatización parcial. Considera que el conductor ya no tiene que conducir en todo momento y que el coche empieza a ser realmente autónomo, aunque con matices. El vehículo es capaz de actuar de forma independiente dentro de escenarios controlados y en situaciones específicas de conducción. El coche no detecta o responde ante objetos externos y es el conductor el que debe seguir prestando atención a lo que ocurre a su alrededor para evitar posibles riesgos. Un buen ejemplo de Nivel 2 de conducción autónoma pueden ser los modelos BMW Serie 7 o el Mercedes Clase E, capaces de ir solos durante un tiempo o con el sistema de asistente de atascos.
- Nivel 3: Automatización condicional. En este nivel, el coche empieza a interactuar con lo que le rodea y es capaz de analizar posibles riesgos externos para evitarlos. Ya no se habla de conductor sino que hablamos de un usuario preparado para intervenir, es decir, el coche ya conduce completamente solo y el conductor es un simple vigilante de que todo funcione correctamente. El coche está preparado para ser conducido de manera habitual sin problema.
- Nivel 4: Alta autonomía. En este nivel el sistema cuenta tanto con los sistemas de automatización presentes en el anterior nivel, como con sistemas de detección de objetos y eventos. Además, es capaz de responder ante ellos. El sistema de automatización de la conducción tiene un sistema de respaldo para actuar en caso

de fallo del sistema principal y poder conducir hasta una situación de riesgo mínimo. En algunas situaciones es posible que el vehículo no siga conduciendo.

- Nivel 5: Autonomía total. Este nivel cuenta con todos los beneficios del sistema de automatización del nivel 4. Sin embargo, la diferencia es que en este caso el vehículo podría seguir conduciendo en todo momento o circunstancia.

Ejemplos importantes de conducción autónoma son: el DARPA Grand Challenge y el Urban Challenge. El DARPA Grand Challenge organizado en 2005 en Estados Unidos, fue una carrera de vehículos autónomos que debían recorrer 120 kms por el desierto de Nevada sin intervención humana y disponiendo únicamente de un listado de puntos intermedios entre el principio del circuito y el final. El Urban Challenge organizado en 2007, fue una carrera de vehículos autónomos por zona urbana en la que debían recorrer 96 km en menos de 6 horas.

La tecnología que se emplea para el aparcamiento autónomo tiene que tener en cuenta diversos factores para aparcar: espacio disponible, maniobras a realizar y posición. Para conseguir este objetivo los coches emplean sensores que miden la distancia desde el coche hasta los límites de la plaza y los otros coches u obstáculos, tanto en el parachoques trasero como en el parachoques delantero. Estos sensores suelen ser de ultrasonidos, y su número y distribución depende del tamaño y diseño del coche, aunque suelen ser cuatro o cinco por parachoques. En algunas ocasiones se puede emplear un radar como sensor.

En los laterales de los paragolpes es necesario que haya más sensores orientados de forma transversal. De esta forma pueden medir la distancia hacia el lateral del coche. Estos sensores permiten identificar si existe un hueco en el que aparcar y si el tamaño del hueco es suficiente para que el coche entre.

La firma Bosch lidera el suministro de tecnología para la automoción y tiene a la totalidad de marcas fabricantes de automóviles como clientes. Un sistema que Bosch proporciona a las marcas es la “cámara de marcha atrás”. Esta cámara está ubicada en la parte posterior del vehículo, y se activa de forma automática cuando el conductor da marcha atrás. Inmediatamente aparecen imágenes en color del entorno en el monitor y las líneas límite de maniobra para evitar roces, así como advertencias acústicas.

Otro sistema importante de Bosch es el sistema Multi-cámara, que cuenta con cuatro cámaras con apertura de 190° que captan todo el entorno del vehículo y se reflejan en la pantalla del salpicadero (un ejemplo es Nissan Quasquai). Estas imágenes son imágenes en 3D sin distorsión.

Bosch también incorpora un dispositivo llamado “freno de emergencia en maniobras”. Este dispositivo detecta obstáculos y objetos en movimiento (mediante sensores ultrasónidos) hasta una distancia de cuatro metros y a una velocidad de no más de 10 km/h. Si hay riesgo de colisión avisa al conductor y si éste no reacciona, activa una frenada de emergencia.

La nueva generación del Audi A8 es uno de los primeros coches dotados de un sistema de conducción autónoma completa. Cuenta con el innovador sistema que se llama “*Traffic Jam Assist*”, el cual permite llevar a cabo una conducción autónoma. Además, incluye el denominado “*Park Assist*” para aparcar el coche desde fuera del vehículo por medio de una aplicación para el teléfono móvil.

El nuevo modelo de Tesla, llamado Model 3, incluye mejoras en su sistema de conducción autónomo e incorpora el nuevo sistema inteligente *Autopilot* que le permite aparcar sólo. Gracias a las cámaras instaladas en la parte frontal y trasera del coche, es capaz de aparcar adecuadamente sin ayuda de un conductor humano.

Las compañías alemanas Daimler y Bosch han presentado el sistema “*Automated Valet Parking*” que combina la tecnología en el propio coche con los sensores presentes en el edificio de aparcamientos para conseguir aparcar el vehículo automáticamente y sin nuestra atención. El concepto es simple: llegas al parking, te detienes en un lugar indicado y te bajas del coche. Basta con pulsar un botón en la aplicación del *smartphone* para que automáticamente el coche busque una plaza de aparcamiento entre las que están libres dentro del edificio del parking.

Estas son algunas de las tecnologías que emplean los coches autónomos o parcialmente autónomos en aparcamiento. Con estos sensores los vehículos son capaces de realizar

maniobras complicadas, que a las personas nos resultan más tediosas. Las tecnologías avanzan rápido, y en este campo han avanzado mucho. Ya existen coches como el de Google que son autónomos por completo, aunque no se hayan comercializado aún por motivos legales.

6.4.2. Solución reactiva

En esta práctica hemos desarrollado una solución “ad hoc” que resuelve el problema del aparcamiento autónomo. La solución reactiva se programa en el fichero *MyAlgorithm.py*, en el método “*execute*”, que se ejecuta periódicamente. Esto permite que la práctica se ejecute como un control reactivo, donde el coche recoge los datos de los sensores en cada instante y toma decisiones de movimiento basándose en estos datos. En esta práctica no se realiza ningún tipo de planificación antes del pilotaje, sino que se realiza el pilotaje directamente.

En la solución que se ha desarrollado el coche obtiene los datos que le proporciona cada láser en primer lugar. Para obtener estos datos, se emplea la función *laser.getLaserData()* para cada correspondiente láser. Esta función nos devuelve los datos crudos del sensor láser, los cuales consisten en 180 pares de valores (distancia y ángulo). El siguiente paso ha sido hacer una media de los 180 valores del láser para tener conocimiento de cuál es la situación global del coche. Por la disposición espacial de los tres sensores se cubre casi los 360° alrededor del vehículo.

Al comienzo de la ejecución de la práctica el coche se encontrará estacionado más atrás de la plaza de aparcamiento, como se puede ver en la Figura 6.6. Los datos del láser son comprobados en cada iteración. El coche avanza a velocidad constante si no ha encontrado aún ninguna plaza de aparcamiento libre. Comprueba la distancia del láser de la derecha y el láser del parachoques trasero para ver si puede parar y estacionar el vehículo. Si los dos láseres tienen un valor entre un cierto rango de valores significa que el coche ha encontrado una plaza de aparcamiento libre. El coche parará en paralelo al coche de delante de la plaza de aparcamiento un momento antes de comenzar a aparcar. Podemos ver en la siguiente imagen cómo el coche alcanza el vehículo de delante de la plaza y para.

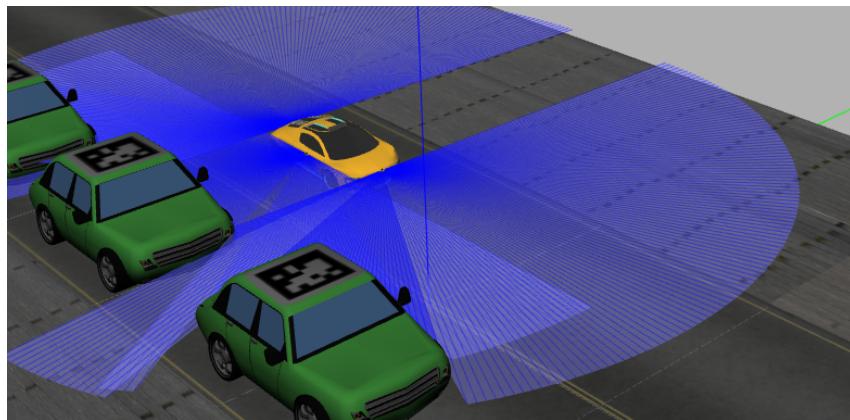


Figura 6.6: Posición inicial del taxi en la práctica

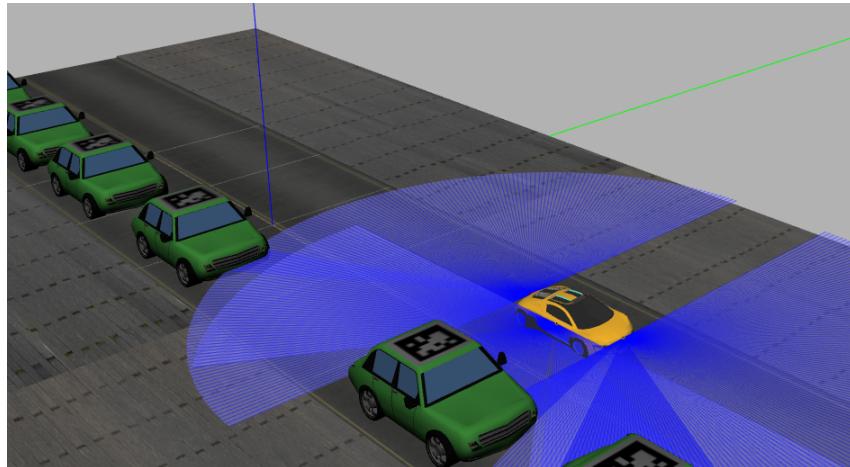


Figura 6.7: Posición al parar en paralelo al coche de delante de la plaza libre

Cuando el coche ya está parado en paralelo al vehículo de delante de la plaza de aparcamiento, entonces comienza la maniobra. Al principio el coche comenzará a dar marcha atrás lentamente, es decir, con una velocidad de tracción determinada. Pero también tendrá cierta velocidad de giro, con lo que describirá una especie de arco al aparcar (como lo hace un coche real al comenzar a aparcar en línea). El giro será hacia la derecha hasta que tenga una determinada inclinación. Esta maniobra la podemos ver en la Figura 6.8.

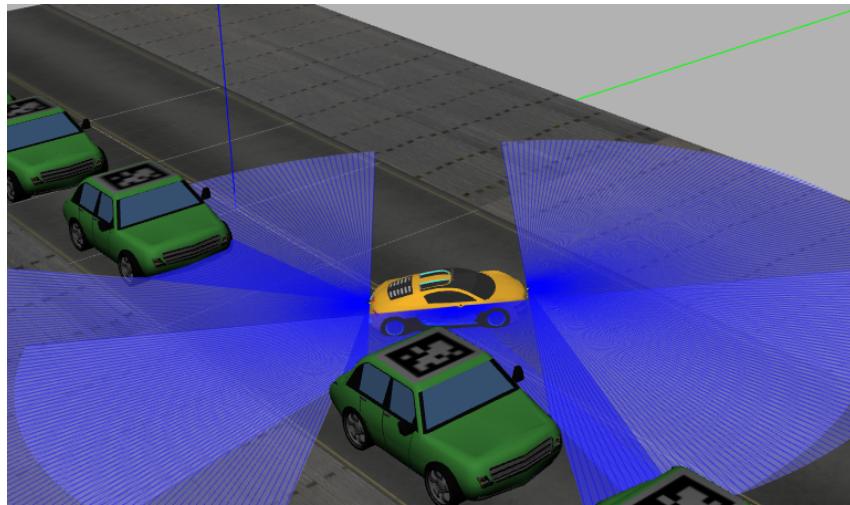


Figura 6.8: Posición al dar marcha atrás con giro a la derecha

Cuando el coche alcanza una determinada orientación, entonces se realiza otra maniobra para enderezar el vehículo. Ahora que ya ha entrado parte del coche en la plaza de aparcamiento, dará marcha atrás y realizará otra especie de arco. Es decir, que en esta ocasión el coche tendrá una velocidad de giro, pero hacia la izquierda. Esta maniobra se puede ver en la Figura 6.9.

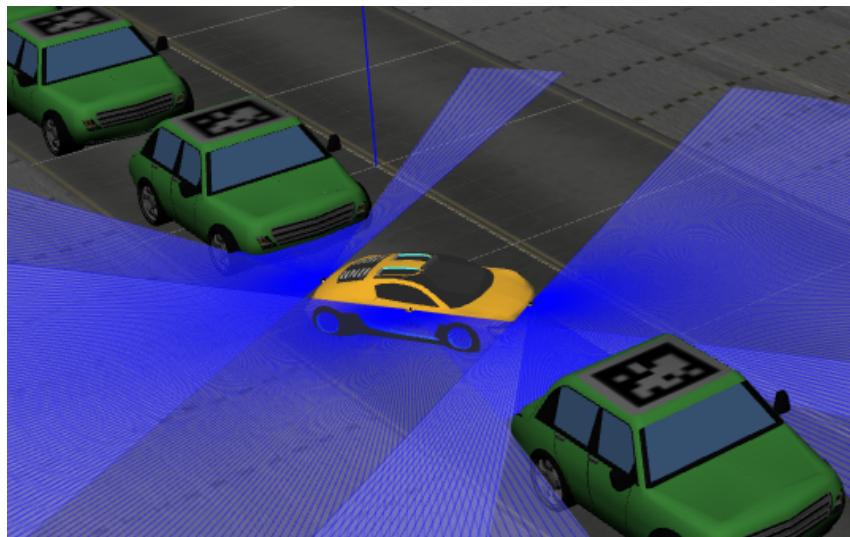


Figura 6.9: Posición al dar marcha atrás con giro a la izquierda

El coche comprobará los datos del láser trasero en todo momento para saber si está cerca del coche de atrás. Seguirá marcha atrás y girando hacia la izquierda hasta que detecte que está demasiado cerca del coche de atrás. Cuando detecte esta situación,

entonces parará y comenzará a rectificar. Para realizar la rectificación dará marcha hacia delante muy despacio y girando hacia la derecha. Esta maniobra la realiza hasta quedar recto, es decir, de forma paralela a la acera. Cuando esté paralelo a la acera, el coche comprobará con los sensores láser frontal y trasero si se ha quedado muy pegado a un coche u otro. Si se ha quedado muy cerca del coche de atrás y tiene mucho margen por delante, entonces dará marcha hacia delante hasta quedar más o menos centrado en la plaza de aparcamiento. Si estuviera muy pegado al coche de delante, daría marcha atrás hasta estar centrado. Cuando el coche detecte que se ha quedado centrado en la plaza de aparcamiento, entonces parará. En la Figura 6.10, se puede observar cómo ha quedado perfectamente estacionado.

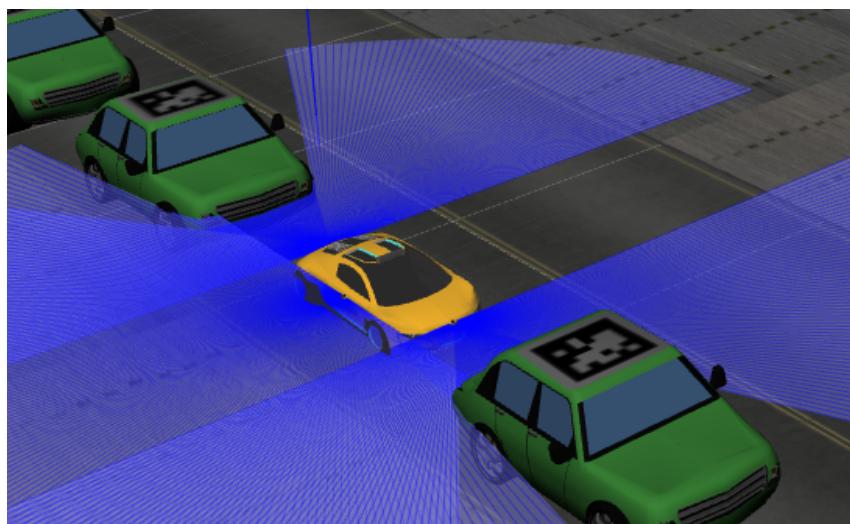


Figura 6.10: Coche estacionado

6.5. Evaluador automático

Se ha desarrollado también un evaluador automático que permite calificar objetivamente la solución a la práctica teniendo en cuenta diferentes parámetros. El evaluador automático muestra estos parámetros en una interfaz gráfica, así como la nota que obtiene el alumno. El evaluador automático se ha creado empleando PyQt5. Como sucedía en las anteriores prácticas, se ha programado mediante clases, que serán instanciadas en una clase principal. En la Figura 6.11 se muestra el evaluador automático durante la ejecución de la práctica.



Figura 6.11: Evaluador Automático del Autopark durante la ejecución de la práctica

En la esquina superior izquierda de su interfaz gráfica (Figura 6.11), hay un visor que muestra un reloj digital. Este reloj va mostrando los segundos que han pasado desde que se ha ejecutado la práctica.

En la esquina inferior izquierda de la interfaz gráfica se representa parte de un círculo formado por diferentes colores, donde hay una aguja. Este semicírculo representa la orientación del robot, en función de determinados colores. Esto permite ver si nuestro vehículo está correctamente alineado o si por el contrario está algo desviado. Para que el coche esté perfectamente alineado tiene que tener una orientación de 0 grados. El color verde representa una orientación entre -15 y 15 grados. La zona anaranjada representa el rango de orientación entre -45 y -15 grados; y, entre 15 y 45 grados. El color rojo representa una orientación entre -45 y 45 grados. La aguja apuntará a un color u otro, y a cierta zona en función de la orientación que posea el robot.

En la esquina superior derecha se muestra en el visor diferentes distancias numéricas. La distancia a la acera que está pegada a la zona de aparcamiento, la distancia al coche de delante del hueco de aparcamiento, y la distancia al coche de atrás del hueco de aparcamiento. Esto permite comprobar el espacio libre delante y detrás y podemos ver si nos estamos acercando mucho a alguno de estos coches al realizar el aparcamiento, lo que podría ser peligroso.

Debajo de los mensajes de distancias el visor tiene una barra de progreso. Esta barra (de color rojo) aumentará en caso de que nos choquemos con algún coche al intentar aparcar.

En la parte central superior de la interfaz hay un botón, el cual se debe pulsar para saber la puntuación obtenida. Este botón (“*Show me my mark*”) mostrará un mensaje con la nota conseguida. Esta nota se calcula en base al tiempo que se tarda en aparcar, la distancia a la que nos encontramos del hueco por si el taxi se ha quedado algo desviado, la orientación del taxi por si acaso se ha quedado torcido; y los choques que hayamos tenido con algún vehículo.

6.6. Experimentación

6.6.1. Ejecución típica

La práctica “Aparcamiento automático” se puede ejecutar abriendo tres terminales y escribiendo lo siguiente en cada uno de ellos:

1. Lanzar Gazebo: gazebo autopark.world
- 1b. Si el ordenador que se emplea no posee muchos recursos se puede arrancar el simulador sin interfaz gráfico: gzserver autopark.world
2. Ejecutar la práctica y lanzar la interfaz gráfica (GUI): python2 autopark.py --Ice.Config=autopark.cfg
3. Ejecutar el evaluador automático: python2 referee.py --Ice.Config=autopark.cfg

En la Figura 6.12 podemos ver una secuencia de imágenes donde empleando el algoritmo de la solución de referencia el coche autónomo es capaz de aparcar en línea correctamente. Una ejecución típica se puede ver en este video ¹.

¹<https://www.youtube.com/watch?v=2SYEb3DyWEE>

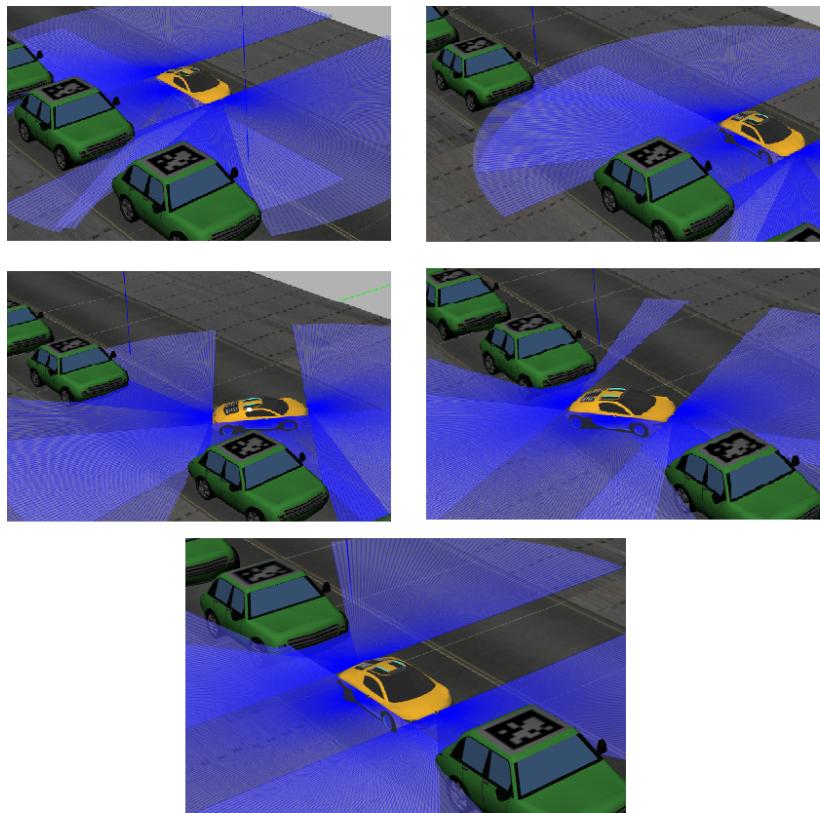


Figura 6.12: Secuencia de la ejecución típica de la solución de referencia

6.6.2. Aparcamiento lejano

En este experimento hemos puesto inicialmente el coche más alejado de la plaza de aparcamiento para ver si era capaz de conducir hasta encontrarla y aparcar. El coche lo hemos desplazado en Gazebo 8 metros más atrás de la calle. En las pruebas hemos podido observar que el vehículo era capaz de hacer más recorrido conduciendo hacia delante hasta que encontraba la plaza de aparcamiento y estacionaba de forma correcta. A continuación, podemos ver una secuencia de imágenes, donde se puede apreciar desde qué posición iniciaba el pilotaje y cómo era capaz de aparcar.

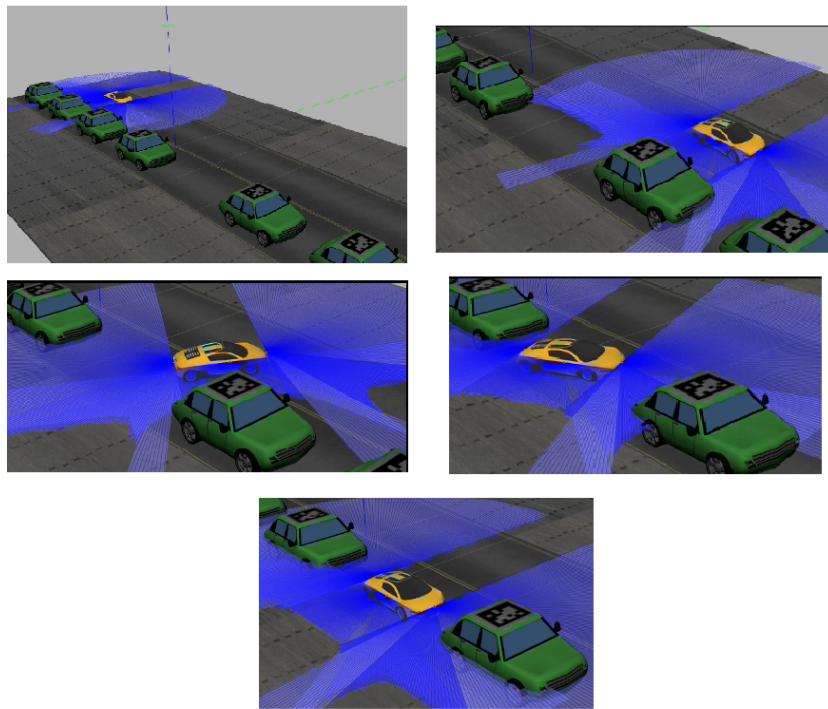


Figura 6.13: Coche en el aparcamiento lejano

6.6.3. Experimento con la plaza casi superada

En la siguiente prueba se ha desplazado la posición inicial del vehículo hasta tenerlo en paralelo al coche que está delante de la plaza de aparcamiento. En esta prueba se ha comprobado que el coche no avanza hacia delante para buscar una plaza libre, sino que detecta que hay una plaza libre justo detrás de él y comienza a realizar la maniobra de aparcamiento. Este era el objetivo que se buscaba en esta prueba y el coche lo ha conseguido con éxito. En la Figura 6.14 tenemos una secuencia de imágenes que muestran el coche al inicio y las maniobras realizadas por el mismo para aparcar.

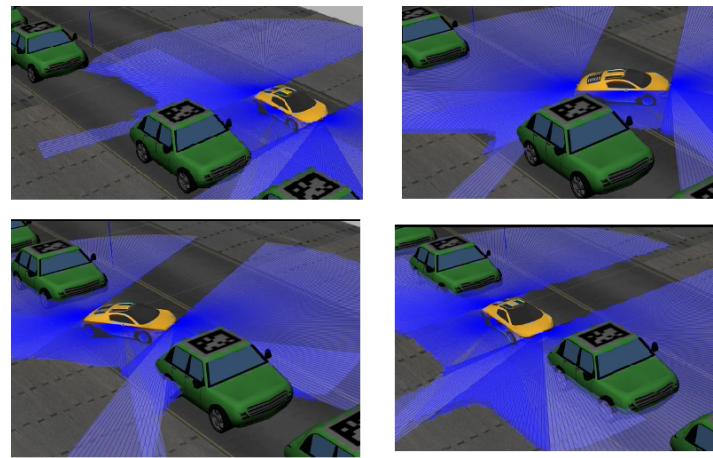


Figura 6.14: Aparcamiento con la plaza casi superada

Capítulo 7

Conclusiones

En este capítulo se exponen las conclusiones finales obtenidas, así como los posibles trabajos futuros, tras analizar las tres prácticas aportadas (su infraestructura, aplicación académica, solución de referencia y evaluador automático de cada una de ellas) y algunos experimentos relevantes con las mismas.

7.1. Conclusiones

En este proyecto se ha alcanzado satisfactoriamente el objetivo global de crear tres prácticas académicas destinadas a la docencia en robótica en el entorno JdeRobot-Academy, extendiendo el repertorio de prácticas existente previamente. El propósito de este entorno es que los alumnos puedan afianzar los conocimientos teóricos programando algunas técnicas robóticas, poniendo estos conocimientos en práctica.

Además de la infraestructura y el software de apoyo, se ha desarrollado una solución de referencia por cada una de estas prácticas, que se ha validado experimentalmente. Se ha seguido el diseño de JdeRobot-Academy buscando resolver en cada componente académico todos los problemas secundarios, pero que es necesario solventar, para que los alumnos puedan programar la solución del algoritmo de cada práctica. Es decir, se abstrae a los alumnos de los problemas complejos que conlleva la práctica, tales como la comunicación con Gazebo, con los sensores y actuadores del robot o la programación de la interfaz gráfica.

En primer lugar, se ha mejorado la versión anterior de la práctica “TeleTaxi”: fallos

CAPÍTULO 7. CONCLUSIONES

corregidos, coche nuevo, solución de referencia nueva y evaluador automático. Asimismo, se ha desarrollado una solución de referencia empleando el algoritmo *Gradient Path Planning*, como se comentó en el Capítulo 4. En esta solución se ha realizado una planificación global con el algoritmo GPP que sirve para que el robot pueda realizar el pilotaje correctamente. Se ha programado mediante una estructura que expande el gradiente desde el punto de destino deseado hasta un poco más allá del punto donde se encuentra el robot. Esta planificación dará lugar a una rejilla con información del campo calculado, lo que ayudará al robot a navegar correctamente hasta el destino. Este campo ha sido calculado de tal forma que el robot no intente navegar pegado a los obstáculos y, por tanto, choque con ellos. El robot en el pilotaje evalúa en cada momento cuál es la mejor opción. Esto produce ciertas oscilaciones y afecta a la velocidad que debe llevar el robot. A pesar de estas dificultades, se han logrado unos resultados buenos como se describieron en la Sección 4.4.

El segundo subobjetivo era la creación de una nueva práctica llamada “Aspiradora Autónoma”. Se ha resuelto satisfactoriamente con: creación de infraestructura, componente académico, solución de referencia y evaluador automático respectivos. Se ha desarrollado una solución de referencia que no necesita que la aspiradora tenga autolocalización. Tal y como se vio en el Capítulo 5 este algoritmo se inspira en el que emplean los modelos 500, 600, 700 u 800 de Roomba. En primer lugar realiza un patrón en espiral, a continuación, recorre el perímetro de la casa durante un cierto periodo de tiempo, y posteriormente hace un algoritmo de “cruce de habitación”. Hemos podido comprobar con los resultados que el tiempo en el estado “recorrer perímetro” influía en los resultados conseguidos. El “cruce de habitación” consistía en un algoritmo, donde el robot se movía hacia delante hasta chocar con un obstáculo, giraba hasta conseguir un ángulo aleatorio, y se movía de nuevo hacia delante. Al ser aleatorio, los resultados obtenidos pueden variar bastante, puede que unas veces la aspiradora sea capaz de recorrer un gran porcentaje de la casa, mientras que otras no. Además, conforme vaya pasando el tiempo resulta más difícil que la aspiradora recorra zonas que no haya visitado ya.

Para alcanzar el tercer subobjetivo se ha creado la práctica “Aparcamiento Automático”: su infraestructura, su componente académico, una solución de referencia y un evaluador automático. Se ha programado una solución de referencia empleando un algoritmo “ad hoc” en el que se usan los datos de los sensores láser para aparcar el robot correctamente.

Vimos en el Capítulo 6 cómo el vehículo busca una plaza libre de aparcamiento en base a los datos sensoriales ofrecidos por los tres láseres. Gracias a estos datos era capaz de frenar justo en paralelo al coche de delante de la plaza libre y realizar la maniobra de aparcamiento hasta tener el coche perfectamente paralelo a la acera y haber dejado más o menos el mismo espacio delante y detrás del vehículo. En los experimentos que se han realizado hemos podido validar que era capaz de aparcar correctamente.

Además de alcanzar los tres subobjetivos, se han satisfecho también otros requisitos que están implícitos en cada práctica, como emplear el simulador Gazebo. En los mundos virtuales para cada práctica se han simulado diferentes objetos, que suponen obstáculos para el robot.

En cuanto a los aportes personales, hemos aprendido a utilizar la plataforma JdeRobot para programar el comportamiento de diferentes robots autónomos. Uno de los elementos fundamentales de aprendizaje de esta plataforma es cómo se comunican los robots con los sensores y actuadores que poseen. Este proyecto ha servido además para comprender las diferentes fases en las que se divide un trabajo de esta envergadura. Gracias a ello se ha aprendido a dividir un gran objetivo en pequeños objetivos de ingeniería, haciendo más fácil la solución de los mismos. Además, han surgido problemas típicos de ingeniería durante el proyecto, más sencillos o más complejos, los cuales ha habido que solventar bien mediante más pruebas y experimentos, bien cambiando la técnica que se estaba empleando, o bien refinando el algoritmo que se empleaba hasta obtener los objetivos deseados.

7.2. Trabajos futuros

Debido a que éste es un Trabajo de Fin de Grado no ha sido posible alargarlo ad infinitum para realizar más mejoras sobre el mismo. A continuación se describirán posibles líneas concretas en las que se puede mejorar cada una de las prácticas.

Varias posibles mejoras que se podrían realizar en un futuro sobre la práctica “TeleTaxi” son:

- En este proyecto se ha resuelto el problema de la planificación mediante la técnica

Gradient Path Planning. Se podrían explorar otras técnicas de planificación, como un grafo de visibilidad, para comparar los resultados obtenidos con cada técnica y llevar a cabo un estudio más completo del problema.

- Una dificultad incorporada en la navegación del robot era que el robot iba mirando en cada iteración sólo cuál era su posición respecto al mapa y el campo del gradiente calculado. Esto conllevaba que el robot pudiera navegar demasiado cerca de obstáculos o incluso chocar. Una posible mejora sería incorporar sensores en el robot, como sensores láser o cámaras, para detectar obstáculos y poder navegar aún con mayor precaución.
- En este proyecto se ha validado el algoritmo únicamente sobre un simulador. Una posible mejora sería llevar el algoritmo propuesto a un robot real. Para ello es necesario conocer la posición del robot respecto al lugar por donde se mueve, por lo que antes habría que dotarle de un algoritmo de autolocalización.

En la práctica “Aspiradora Autónoma”, varias posibles mejoras que se podrían llevar a cabo son:

- La aspiradora funciona aquí sin autolocalización, lo que limita el algoritmo, puesto que no tiene conocimientos previos de la casa antes de la navegación. Una posible mejora sería emplear el algoritmo SLAM para crear un mapa de la casa. Esta información dotaría al robot de un mayor conocimiento de la casa, lo que haría posible que se lleve a cabo un algoritmo planificado, por el cual se puede recorrer la casa en menor tiempo.
- Se podrían realizar pruebas con una aspiradora real que incorporara el algoritmo desarrollado para comprobar qué resultados se obtendrían en diferentes habitaciones o incluso en una casa.

Varias posibles mejoras de la práctica “Aparcamiento Automático” son:

- El robot empleado en esta práctica únicamente poseía tres sensores láser, lo que limita el conocimiento de los alrededores del robot. Se podría añadir más sensores al robot, por ejemplo cámaras, para dotarle de mayor precisión en el conocimiento de su entorno, lo que haría que se pudiera llevar a cabo un algoritmo más seguro ante imprevistos.

- El algoritmo que se ha desarrollado es reactivo basado directamente en los datos sensoriales. Otra posibilidad sería conocer el mapa del entorno por el que se mueve el robot y desarrollar algún tipo de planificación. Una posibilidad sería emplear el algoritmo *Rapidly-exploring Random Tree (RRT)*, que es un método de planificación de trayectorias; o emplear la librería *Open Motion Planning Library (OMPL)*, que es un software con varios algoritmos de planificación. En estos ejemplos se podría tener en cuenta que se emplea un robot no holonómico.
- Al igual que en las prácticas anteriores, esta práctica se ha validado sobre el simulador Gazebo. Una línea a explorar sería ver cómo se comporta un robot real dotado con tres sensores láser al intentar aparcar en una plaza de aparcamiento, al igual que lo hacía nuestro robot simulado. Sería un experimento interesante para comprobar la generalidad de la solución desarrollada.

Bibliografía

- [1] Simone Ceriani and Martino Migliavacca. Middleware in robotics. *Advanced Methods of Information Technology for Autonomous Robotics*. [Accedido 12 de Septiembre de 2017].
- [2] J. Baillie, A. Demaille, Q. Hocquet, M. Nottale, and S Tardieu. The Urbi Universal Platform for Robotics. *Simulation, Modeling and Programming for Autonomous Robots*, pages 580–591, 2008. [Accedido 23 de Septiembre de 2017].
- [3] 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. [Accedido 23 de Septiembre de 2017].
- [4] Jorge Coronado Vallés. *Desarrollo de aplicaciones embebidas de control en robots móviles*. PhD thesis, Universidad Politécnica de Valencia, 2013. [Accedido 23 de Septiembre de 2017].
- [5] Webots, *Cyberbotics*. <https://www.cyberbotics.com/webots.php>. [Accedido 12 de Septiembre de 2017].
- [6] Alberto Manuel Mireles Suárez. Re-identificación de personas en redes de sensores RGBD, *Universidad de las Palmas de Gran Canaria*. file:///C:/Users/jessi/Downloads/0710907_00000_0000.pdf, 2015. [Accedido 23 de Septiembre de 2017].
- [7] Javier Nuño Simón. *Reconocimiento de objetos mediante sensor 3D Kinect*. PhD thesis, Universidad Carlos III de Madrid, 2012. [Accedido 23 de Septiembre de 2017].
- [8] AForge.NET Framework, *AForge.NET*. <http://www.aforgenet.com/framework/>. [Accedido 12 de Septiembre de 2017].
- [9] Galo Fariño R. Modelo Espiral de un proyecto de desarrollo de software, *Administración y Evaluación de Proyectos*. <http://www.ojovisual.net/galofarino/modeloespiral.pdf>, 2011. [Accedido 7 de Agosto de 2017].
- [10] Modelo Espiral. <http://modeloespiral.blogspot.com.es/>, 2009. [Accedido 7 de Agosto de 2017].

BIBLIOGRAFÍA

- [11] What is PyQt?, *Riverbank Computing Limited.* <https://riverbankcomputing.com/software/pyqt/intro>, 2016. [Accedido 9 de Agosto de 2017].
- [12] Jan Bodnar. Introduction to PyQt5. <http://zetcode.com/gui/pyqt5/introduction/>, 2017. [Accedido 9 de Agosto de 2017].
- [13] Kurt Konolige. A Gradient Method for Realtime Robot Control. *SRI International*, 2000.
- [14] José Raúl Isado. *Navegación global de un robot usando el método del gradiente*. PhD thesis, Universidad Rey Juan Carlos.Escuela Superior de Ciencias Experimentales y Tecnología, 2005. [Accedido 10 de Agosto de 2017].
- [15] Julio Manuel Vega. *Navegación y autolocalización de un robot guía de visitantes*. PhD thesis, Universidad Rey Juan Carlos. Ingeniería Informática, 2009. [Accedido 10 de Agosto de 2017].
- [16] Juan Navarro Bosgos. *Construcción de Mapas 3D Compactos desde Sensores RGBD*. PhD thesis, Universidad Rey Juan Carlos.Ingeniería de Telecomunicación, 2015. [Accedido 17 de Agosto de 2017].
- [17] Florencia Ucha. Definición de Robótica. <https://www.definicionabc.com/tecnologia/robotica.php>, 2009. [Accedido 5 de Agosto de 2017].
- [18] Definición de Robot. <http://conceptodefinicion.de/robot/>, 2014. [Accedido 5 de Agosto de 2017].
- [19] Marilyn Perdigón. El padre de la robótica, *Los primeros robots*. <http://marilynpg.blogspot.com.es/>, 2012. [Accedido 5 de Agosto de 2017].
- [20] Historia. <http://wiki.robotica.webs.upv.es/wiki-de-robotica/introduccion/historia/>. [Accedido 5 de Agosto de 2017].
- [21] Yanet Maritza Sagua Alanguía. Clasificación de los Robots, *Robótica Puno*. <http://roboticapuno.blogspot.com.es/2013/01/clasificacion-de-los-robots.html>. [Accedido 5 de Agosto de 2017].
- [22] Clasificación de robots. <http://wiki.robotica.webs.upv.es/wiki-de-robotica/introduccion/clasificacion-de-robots/>. [Accedido 5 de Agosto de 2017].

BIBLIOGRAFÍA

- [23] Generaciones de la Robótica, *Todo sobre la Robótica*. <http://conozcamoslarobotica.blogspot.com.es/p/generaciones-de-la-robotica.html>. [Accedido 5 de Agosto de 2017].
- [24] Robots Militares, *Actualidad Gadget*. <https://www.actualidadgadget.com/tag/robots-militares/>. [Accedido 5 de Agosto de 2017].
- [25] Víctor R. González. Definición del Robot Industrial, *Robots industriales*. http://platea.pntic.mec.es/vgonzale/cyr_0204/ctrl_rob/robotica/industrial.htm, 2002. [Accedido 5 de Agosto de 2017].
- [26] Importancia de la Robótica en la Educación, *Educatronics. Ciencia, arte y arquitectura*. <http://educatronics.com/publicaciones/importancia-de-la-rob%C3%BCtica-en-la-educacion>. [Accedido 5 de Agosto de 2017].
- [27] I. Moreno, L. Muñoz, J. Rolando, J. Quintero, K. Pittí, and J. Quiel. La robótica educativa, una herramienta para la enseñanza-aprendizaje de las ciencias y las tecnologías. *Revista Teoría de la Educación: Educación y Cultura en la Sociedad de la Información.*, 13(2):74–90, July 2012.
- [28] J.M. Cañas, A. Martí, E. Perdices, F. Rivas, and R. Calvo. Entorno docente para la programación de la inteligencia de los robots. *Iberoamericana de Automática e Informática Industrial*, pages 1–12, 2016.
- [29] L.F Bello and J.Gutierrez. Modelo en Espiral y Modelo Basado en Prototipos. <https://prezi.com/y7slahvparel/modelo-en-espiral-y-modelo-basado-en-prototipos-para-el-desarrollo-de-software/>, 2017. [Accedido 7 de Agosto de 2017].
- [30] Sonia Posligua. Modelo en Espiral. <https://es.slideshare.net/soniaposligua/modelo-enespiral>, 2013. [Accedido 7 de Agosto de 2017].
- [31] Carlos Santana. Historia de Python. <https://www.codejobs.biz/es/blog/2013/03/03/historia-de-python>, 2013. [Accedido 9 de Agosto de 2017].
- [32] Python Software Foundation. Tutorial de Python. <http://docs.python.org.ar/tutorial/3/real-index.html>, 2017. [Accedido 9 de Agosto de 2017].

BIBLIOGRAFÍA

- [33] Python, *EcuRed*. <https://www.ecured.cu/Python>, 2017. [Accedido 9 de Agosto de 2017].
- [34] Julio M. Vega. Navegación y autolocalización de un robot guía de visitantes. <https://gsyc.urjc.es/jmvega/documentation/2008-finalProject-guideRobot-talk.pdf>, 2008. [Accedido 9 de Agosto de 2017].
- [35] Navegación en Robots Móviles, *Planificación de Trayectorias para Robots Móviles*. <http://webpersonal.uma.es/~VFMM/PDF/cap2.pdf>. [Accedido 10 de Agosto de 2017].
- [36] J.M. Cañas, J.L. Isado, and L. García. Robot navigation combining the Gradient Method and VFF inside JDE architecture. 2005.
- [37] Adrián González. *Planificación de Movimiento en Robótica Móvil utilizando retículas de estados*. PhD thesis, Universidad de Santiago de Compostela.Escola Técnica Superior de Enxeñaria, 2011. [Accedido 10 de Agosto de 2017].
- [38] Julio Manuel Vega. *Navegación visual del robot Pioneer*. PhD thesis, Universidad Rey Juan Carlos.Ingeniería Técnica en Informática de Sistemas, 2005. [Accedido 10 de Agosto de 2017].
- [39] Navegación autónoma, *Cuentos Cuánticos*. <https://cuentos-cuanticos.com/2011/11/12/navegacion-autonoma/>, 2011. [Accedido 10 de Agosto de 2017].
- [40] Mariano Gómez Plaza. *Planificación óptima de movimiento y aprendizaje por refuerzo en vehículos móviles autónomos*. PhD thesis, Universidad de Alcalá. Escuela Politécnica Superior, 2009. [Accedido 11 de Agosto de 2017].
- [41] J.M Cañas. *Programación de robots con la plataforma Jderobot*. PhD thesis, Universidad de Málaga, 2009. [Accedido 11 de Agosto de 2017].
- [42] Lucas Martín. Gazebo, simulador de robótica, *Automatismos Mar del Plata*. <http://www.automatismos-mdq.com.ar/blog/2017/01/gazebo-simulador-de-robotica.html>, 2017. [Accedido 12 de Agosto de 2017].
- [43] Gazebo Simulator: simular un robot nunca fue tan fácil, *Robologs*. <https://robologs.net/2016/06/25/>

BIBLIOGRAFÍA

- gazebo-simulator-similar-un-robot-nunca-fue-tan-facil/, 2016. [Accedido 12 de Agosto de 2017].
- [44] Grupo SIRP. Tipos de Movimiento y Grados de Libertad. <https://es.slideshare.net/EducaredColombia/tipos-de-movimiento-y-grados-de-libertad>. [Accedido 13 de Agosto de 2017].
- [45] Lego NXT Holonómico, *Lego NXT Mindstorms*. <http://www.lejosconlego.com/2013/02/lego-nxt-holonomico.html>. [Accedido 13 de Agosto de 2017].
- [46] Que es la odometría, *Arqphys Arquitectura*. <http://www.arqphys.com/contenidos/quees-odometria.html>, 2017. [Accedido 13 de Agosto de 2017].
- [47] SDF. <http://sdformat.org/>, 2017. [Accedido 13 de Agosto de 2017].
- [48] Simuladores. https://eva.fing.edu.uy/pluginfile.php/127423/mod_resource/content/1/simuladores.pdf. [Accedido 13 de Agosto de 2017].
- [49] iRobot Corporation. *Manual Roomba 500. Robot de Limpieza Aspirador*, 2010. [Accedido 14 de Agosto de 2017].
- [50] iRobot Corporation. *Manual Roomba 800*, 2014. [Accedido 14 de Agosto de 2017].
- [51] Elena Rodríguez. *Sistema de Control de Aspiradora Automática*. PhD thesis, Universidad Pontificia Comillas. Escuela Técnica Superior de Ingeniería, 2008. [Accedido 14 de Agosto de 2017].
- [52] Lucía Ruiz. Las Mejores Aspiradoras Robot del 2017 – Guía de Compra, *Aspiradoras de mano*. <http://www.aspiradorasdemano.com/mejores-aspiradoras-robot/>, 2017. [Accedido 14 de Agosto de 2017].
- [53] Ana Misiego. *Visión Artificial y su aplicación en la automatización de tareas de limpieza*. PhD thesis, Universidad de Valladolid. Escuela de Ingenierías Industriales, 2017. [Accedido 15 de Agosto de 2017].
- [54] Jason Roberts. Xiaomi Mi robot vacuum cleaner – what lays behind the hype, *Vacuums Guide*. <https://www.vacuumsguide.com/xiaomi-mi-robot-vacuum-cleaner/>, 2016. [Accedido 15 de Agosto de 2017].

BIBLIOGRAFÍA

- [55] Test Robot aspiradora Xiaomi Mi, *Las aspiradoras*. <http://www.lasaspiradoras.com/test-robot-aspiradora-xiaomi-mi/>, 2017. [Accedido 15 de Agosto de 2017].
- [56] Electrolux. *Manual Trilobite*, 2001. [Accedido 15 de Agosto de 2017].
- [57] Guido Zunino. *Simultaneous Localization and Mapping for Navigation in Realistic Environments*. PhD thesis, Royal Institute of Technology, 2002. [Accedido 16 de Agosto de 2017].
- [58] Julia Layton. How Robotic Vacuums Work, *HowStuffWorks*. <http://electronics.howstuffworks.com/gadgets/home/robotic-vacuum2.htm>. [Accedido 16 de Agosto de 2017].
- [59] Robot aspirador LG Hombot Turbo. Especial casas con mascotas, niños y alfombras, *LG*. <http://www.lg.com/es/aspiradoras/lg-VR65710LVMP>, 2017. [Accedido 16 de Agosto de 2017].
- [60] Roomba navigation algorithm, *Random Sequence*. <http://www.randseq.org/2012/11/roomba-navigation-algorithm.html>, 2012. [Accedido 16 de Agosto de 2017].
- [61] Introducción a las Espirales. <http://www.ieshumanes.com/nw/Espiral/espirales.pdf>. [Accedido 17 de Agosto de 2017].
- [62] Damiani Díaz. Espirales y funciones polares, *Pontificia Universidad Católica de Ecuador*. <http://calculodiferencial-pucesd.blogspot.com.es/2012/12/espirales-y-funciones-polares.html>, 2012. [Accedido 17 de Agosto de 2017].
- [63] La Espiral de Arquímedes. http://www.ite.educacion.es/formacion/enred/web_espiral/matematicas_1/arquimedes.htm. [Accedido 17 de Agosto de 2017].
- [64] Jordan Van Duyne. Página del repositorio AutonomousParking. <https://github.com/jovanduy/AutonomousParking>, 2017. [Accedido 19 de Agosto de 2017].
- [65] Avances tecnológicos en el aparcamiento autónomo, *Central Recambio Original*. <http://www.recambiooriginal.com/blog/recambios-originales/mecanica/avances-tecnologicos-aparcamiento-autonomo/>, 2017. [Accedido 19 de Agosto de 2017].

BIBLIOGRAFÍA

- [66] Nacho Teso. Aparcamiento autónomo: cómo funciona su tecnología,*Noticias coches.com*. <https://noticias.coches.com/noticias-motor/aparcamiento-autonomo-como-funciona-su-tecnologia/234288>, 2016. [Accedido 19 de Agosto de 2017].
- [67] Ibáñez. Tecnología para el coche: sistemas de aparcamiento automático,*xataka*. <https://www.xataka.com/automovil/tecnologia-para-el-coche-sistemas-de-aparcamiento-automatico>, 2012. [Accedido 19 de Agosto de 2017].
- [68] José Manuel Maletá. Daimler y Bosch, un ensayo de 'parking' autónomo desde el móvil en el museo Mercedes-Benz,*El Mundo*. <http://www.elmundo.es/motor/2017/07/24/59762c8522601d11508b467a.html>, 2017. [Accedido 19 de Agosto de 2017].
- [69] Ibáñez. De 0 a 5: cuáles son los diferentes niveles de conducción autónoma, a fondo,*xataka*. <https://www.xataka.com/automovil/de-0-a-5-cuales-son-los-diferentes-niveles-de-conduccion-autonoma>, 2017. [Accedido 19 de Agosto de 2017].
- [70] Markus Maurer, J. Christian Gerdts, Barbara Lenz, and Hermann Winner. *Autonomous Driving. Technical, Legal and Social Aspects*. Springer Open, Daimler und Benz-Stiftung, Ladenburg, 2016.
- [71] Julio Benítez. Sistemas de aparcamiento inteligente. <http://www.vidapremium.com/sistemas-de-aparcamiento-inteligente-2031.htm#5>, 2017. [Accedido 19 de Agosto de 2017].
- [72] Página Oficial de OpenCV. <http://opencv.org/>, 2017.
- [73] Jose Cabrera Lozano. Definición de robótica educativa*Edukative, Robótica educativa*. <https://edukative.es/definicion-robotica-educativa/>, 2014. [Accedido 11 de Septiembre de 2017].
- [74] Iniciativas que promueven las competencias 4C en los alumnos: creatividad, pensamiento crítico, colaboración y comunicación,*innedu*. <http://www.innedu.es/iniciativas-que-promueven-las-competencias-4c-en-los-alumnos-creatividad-pensamiento-critico-colaboracion-y-comunicacion/#.WbbxB8hJbIV>. [Accedido 11 de Septiembre de 2017].

BIBLIOGRAFÍA

- [75] Importancia de los videojuegos para las habilidades del siglo XXI, *ÁrbolABC.com*. <https://arbolabc.com/blog/importancia-de-los-videojuegos-para-las-habilidades-del-siglo-xxi/>. [Accedido 11 de Septiembre de 2017].
- [76] Las Cuatro Habilidades De Una Educación Del Siglo 21, *Fundación SM*. <http://www.seminariointernacional.com.mx/blog/Las-cuatro-habilidades-de-una-educacion-del-siglo-21>, 2015. [Accedido 11 de Septiembre de 2017].
- [77] G. Ocaña, I. Romero, F. Gil, and A. Codina. Implantación de la nueva asignatura “Robótica” en Enseñanza Secundaria y Bachillerato. *Investigación en la Escuela*, 13(87):65–79, 2015. [Accedido 12 de Septiembre de 2017].
- [78] JdeRobot-kids, *JdeRobot*. <http://jderobot.org/Robotica-en-secundaria>, 2017. [Accedido 12 de Septiembre de 2017].
- [79] Tutoriales de robóticaTutoriales de robótica. https://www.solidworks.es/sw/education/9931_ESN_HTML.htm. [Accedido 12 de Septiembre de 2017].
- [80] Ricardo Tellez. A thousand robots for each student: using cloud robot simulations to teach robotics. *The Construct Sim*, 2016. [Accedido 12 de Septiembre de 2017].
- [81] Robótica en la Universidad de Alcalá. Proyecto TuBot 2015, *Alcabot*. <http://asimov.depeca.uah.es/robotica/mod/resource/view.php?id=1188>, 2015. [Accedido 12 de Septiembre de 2017].
- [82] Grado en Ingeniería Robótica, *Universidad de Alicante*. <https://cvnet.cpd.ua.es/webcvnet/planestudio/planestudiond.aspx?plan=C211#>, 2016. [Accedido 12 de Septiembre de 2017].
- [83] Plan de Estudios Electrónica, Robótica y Mecatrónica, *Universidad de Málaga*. <http://www.uma.es/grado-en-ingineria-electronica-robotica-y-mecatronica/info/9857/plan-de-estudios-electronica-robotica-y-mecatronica/>, 2017. [Accedido 12 de Septiembre de 2017].
- [84] 471 Másters oficiales de ingeniería electrónica robotica mecatróica, *educaweb*. <http://www.educaweb.com/masters-oficiales-de/>

BIBLIOGRAFÍA

- ingenieria-electronica-robotica-mecatronica/, 2017. [Accedido 12 de Septiembre de 2017].
- [85] Robotics middleware, *Wikipedia*. https://en.wikipedia.org/wiki/Robotics_middleware, 2017. [Accedido 12 de Septiembre de 2017].
- [86] Open Source Robotics Software/Hardware List, *Learn Robotix*. <http://learnrobotix.com/open-source-robotics-software.html>. [Accedido 12 de Septiembre de 2017].
- [87] V. M. Arévalo, J. González, and G. Ambrosio. *La Librería de Visión Artificial OpenCV, Aplicación a la Docencia e Investigación*. PhD thesis, Dpto. De Ingeniería de Sistemas y Automática, Universidad de Málaga, 2004. [Accedido 12 de Septiembre de 2017].
- [88] BoofCV. https://boofcv.org/index.php?title=Main_Page. [Accedido 12 de Septiembre de 2017].