



Escuela Técnica Superior de Ingeniería de Telecomunicación

MOVIMIENTO DE UN BRAZO ROBÓTICO EN EL ENTORNO EDUCATIVO ROBOTICS-ACADEMY

Memoria del Trabajo Fin de Grado
en Ingeniería en Sistemas Audiovisuales y Multimedia

Autor: Ignacio Malo Segura

Tutor: José María Cañas Plaza

Co-tutor: José Francisco Vélez Serrano

2019

Agradecimientos

Resumen

El nuevo período de esplendor tecnológico que se está desarrollando en los últimos años, supone un reto para los ingenieros de cada una de las áreas en crecimiento. Una de ellas es la robótica, que se encarga de utilizar la informática, la mecánica y la electrónica para desarrollar máquinas al servicio de las personas.

Hoy en día encontramos robots tanto a nivel industrial como doméstico, desde fabricación, coches autónomos o aplicaciones médicas hasta aspiradoras inteligentes o drones. A pesar de ello, la demanda de empleo en las próximas décadas no será satisfecha si se mantienen las previsiones sobre estudiantes en materias de perfil tecnológico. Ante esta situación cobra vital importancia la educación, que debe ser de calidad y accesible para garantizar el futuro de los profesionales que diseñan robots.

Este proyecto está dirigido a ampliar los recursos de Robotics-Academy, una plataforma que acerca a los estudiantes al software de control de robots a través de prácticas simplificadas, de manera que tomar contacto con la robótica no se convierta en una odisea. Estas prácticas parten de plantillas donde completar la solución a través de código, y permiten resolver algunos problemas típicos de este campo de conocimiento.

Con este Trabajo de Fin de Grado se ha creado una nueva práctica, utilizando el lenguaje de programación Python, enfocada a conocer la planificación y ejecución de trayectorias con un brazo robótico en un entorno simulado en Gazebo. Para ello, se hace uso de tecnologías extendidas como la librería OpenCV, para el procesado de las imágenes recibidas por el robot, o el *framework* MoveIt!, que ofrece todas las herramientas necesarias para trabajar con robots sobre ROS. El objetivo es conseguir detectar objetos de color, además de planificar y ejecutar movimientos en entornos con obstáculos.

Este trabajo permite acceder a los distintos recursos a través de interfaces sencillas para trabajar con imágenes y movimientos, abstrayendo al receptor de la misma de detalles de bajo nivel y configuraciones complejas. La experimentación realizada ha permitido profundizar en la planificación hasta lograr unos resultados estables, que serán presentados a continuación.

Índice general

1 Introducción	1
1.1 Robótica	1
1.1.1 El sector: presente y futuro	3
1.2 Software en robótica	4
1.3 Docencia en robótica	6
1.3.1 Robótica en la escuela	6
1.3.2 El entorno Robotics-Academy	7
2 Objetivos	11
2.1 Objetivos	11
2.2 Requisitos	11
2.3 Metodología	12
2.4 Plan de acción	13
3 Infraestructura	15
3.1 PR2	15
3.1.1 Microsoft Kinect	15
3.2 Gazebo	16
3.3 JdeRobot	17
3.4 ROS Kinetic	18
3.5 ICE	19
3.6 C++	19
3.7 Python	19
3.8 OpenCV	20
3.9 MoveIt!: Motion Planning Framework	20
4 Ejercicio de planificación de trayectorias con brazo robótico	23
4.1 Enunciado	23
4.2 Diseño	24
4.3 Infraestructura del ejercicio	25
4.3.1 Mundo Gazebo	25
Modelo PR2 enriquecido	25
Modelos adicionales	26
4.4 Plantilla	27
5 Solución de referencia	31
5.1 Procesamiento de imagen	31
5.1.1 Cámara Kinect y ROS topics	31

ÍNDICE GENERAL

5.1.2 Librería OpenCV	31
5.2 Planificación de trayectorias con MoveIt	33
5.2.1 MoveGroupCommander	34
5.2.2 Controladores	37
5.2.3 Planificador y configuración	38
5.3 Experimentación	39
5.3.1 Ajuste del entorno	41
5.3.2 Ajuste del detector de colores	41
5.3.3 Ajuste del planificador	42
6 Conclusiones	45
6.1 Conclusiones	45
6.2 Trabajos futuros	46
Bibliografía	47

Capítulo 1

Introducción

Antes de comenzar a profundizar en un proyecto de robótica es necesario comprender el contexto en el que se sitúa, así como el estado actual de su campo de estudio. A continuación se analizará el porqué del crecimiento que ha experimentado en los últimos años, para después comprender cómo se desarrolla software en robótica y terminar mostrando el funcionamiento de la docencia en esta disciplina a través del entorno Robotics-Academy, al que va dirigido este Trabajo de Fin de Grado.

1.1. Robótica

La Real Academia Española define la robótica como la "técnica que aplica la informática al diseño y empleo de aparatos que, en sustitución de personas, realizan operaciones o trabajos, por lo general en instalaciones industriales". De aquí se sacan dos ideas principales; trabajar en este área consiste en diseñar y utilizar robots a través de código, y además está enfocada, en un principio, a realizar autónomamente tareas que realizan las personas.

El desarrollo de software para robótica exige controlar cada detalle, teniendo en cuenta también el componente ético que requiere el desarrollo de tecnología. Isaac Asimov, uno de los escritores de ciencia ficción más relevantes de nuestra historia, también pasó a formar parte de la historia de la robótica al enunciar sus tres leyes fundamentales, que sin duda seguirán teniéndose muy en cuenta ahora que los robots son una realidad:

- Un robot no hará daño a un ser humano o, por inacción, permitirá que un ser humano sufra daño
- Un robot debe obedecer las órdenes dadas por los seres humanos excepto si estas órdenes entrasen en conflicto con la 1^a ley.
- Un robot debe proteger su propia existencia en la medida en que esta protección no entre en conflicto con la 1^a o la 2^a Ley.

Para entender completamente la importancia del tema expuesto, conviene asimilarlo a sucesos históricos previos. El ser humano, a lo largo de su historia, se ha caracterizado por su afán de superación, y esa necesidad por obtener más es la que le ha llevado a la excelencia en muchos aspectos tecnológicos.

1.1. ROBÓTICA

Si nos remontamos a la segunda mitad del siglo XVIII, nos encontramos con una serie de cambios en la tecnología, la sociedad y la cultura que propiciaron la Revolución Industrial. Ésta supuso uno de los grandes saltos cualitativos en cuanto al progreso humano, al centrar las actividades de trabajo en procesos que utilizaban máquinas en sustitución de tareas manuales. Ello llevó principalmente a un incremento de la producción, de la mano de una mejora en el transporte a raíz del uso de la máquina de vapor, a costa de algunos conflictos sociales o el inicio del agotamiento de recursos energéticos y el incremento de la contaminación.

A finales del siglo XIX y comienzos del XX, nos encontramos con una serie de descubrimientos y nuevos recursos, como el petróleo o la electricidad, que aceleraron enormemente el crecimiento industrial y permitieron mejorar la calidad del transporte. Estos, juntos a los avances científicos supusieron una mejora de la calidad de vida de las personas.

El siguiente gran salto tecnológico es el más cercano en el tiempo y, sin duda, es el que propicia la posterior aparición de los robots. Los descubrimientos de la segunda mitad del siglo XX en el campo de la electrónica, causaron la explosión de la informática y las telecomunicaciones, desencadenando transformaciones sociales, culturales y económicas tan importantes como las que provocó la llegada de Internet. La concienciación para el uso de las energías renovables o la aparición de nuevas tecnologías de la información y la comunicación son otros aspectos destacables.

Si se observa el imparable crecimiento del porcentaje de usuarios de Internet en el mundo desde 1990 hasta la actualidad (Figura 1), donde ya es superior al 50 % de los individuos, es fácil hacerse una idea de la influencia que ha alcanzado en todos los niveles de la sociedad. Pero no está todo hecho ya que, como se comenta al inicio del razonamiento, el ser humano quiere, puede, y debe seguir progresando tecnológicamente.

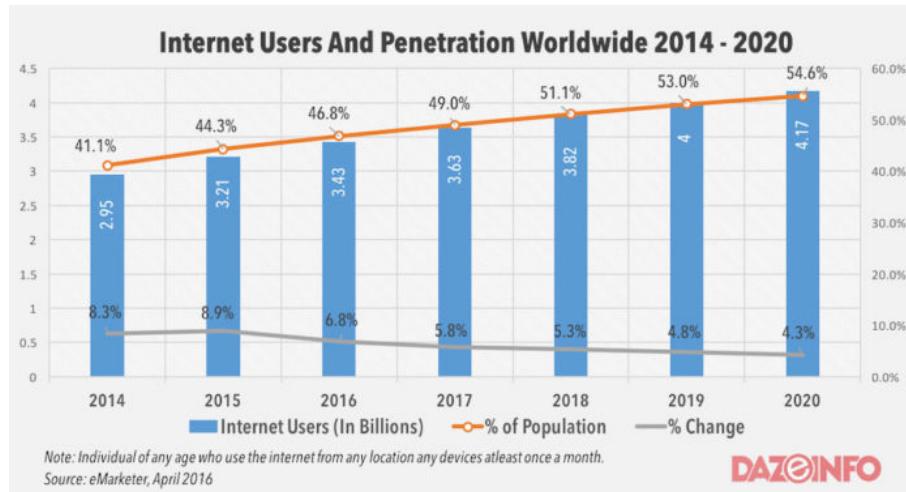


Figura 1: Predicción del número de usuarios en Internet 2016-2020

En el Foro Económico Mundial de 2016 [7] apareció el término Cuarta Revolución Industrial para referirse al siguiente movimiento de esplendor tecnológico: nanotecnología, inteligencia artificial, robótica, computación en la nube, big data, biotecnología, vehículos autónomos, nuevos sistemas de almacenamiento de energía, nuevos materiales, impresión 3D. Sin duda es un reto apasionante para los ingenieros y profesionales de cada una de las áreas de conocimiento, algo posible gracias a un cúmulo de avances a lo largo de los siglos y que ahora es el momento de continuar.

1.1.1. El sector: presente y futuro

Hoy los robots son una realidad. No sólamente son utilizadas por importantes empresas en labores industriales, como Tesla en la fabricación de vehículos o Amazon en la gestión de almacenes. La NASA utiliza robots para sus misiones espaciales, como el Curiosity para la exploración de la superficie de Marte. En el ámbito militar, el robot TEODOR permite evitar situaciones de alto riesgo para los humanos como la desactivación de explosivos. En la medicina, Da Vinci permite al cirujano operar a través de una consola para mejorar la precisión y reducir riesgos en ciertas operaciones quirúrgicas. Además, han ido adquiriendo importancia gradualmente en los hogares y la vida cotidiana de las personas. Desde robots aspiradores que reconocen el entorno, lo memorizan y limpian la casa periódicamente, hasta los avances que encaminan a los vehículos hacia la conducción autónoma.

El mercado de la robótica está llamado a convertirse en uno de los más importantes del presente siglo, alcanzando una valoración estimada superior a los 28 billones de euros en el año 2018 [19]. En los últimos años ha experimentado un gran crecimiento y se espera que continúe en los próximos años, con una previsión de un 25 % de revalorización para los próximos 5 años. Una posible explicación para este hecho se fundamentaría en la aún reciente tendencia a la automatización de tareas a nivel empresarial, sumado en menor medida a un incremento en la compra de tecnología robótica a nivel usuario.

Ahora bien, ¿cuáles son las razones que explican este crecimiento de la automatización, y hacia dónde nos lleva?. Aunque hay algunas características propias del ser humano, como la creatividad o la impredecibilidad, que son imposibles de reemplazar hasta el momento, algunas de las razones que explican las tendencias comentadas son:

- Aumento de la eficiencia
- Productividad sin descanso
- Reducción de costes
- Evita el error humano
- Más precisión
- Control de errores

Por otro lado, uno de los factores clave para comprender el futuro del sector es el impacto que puede tener en el mercado laboral tal y como lo conocemos actualmente. Algunos estudios [1] [10] apuntan que dentro de 35 años, más del 50 % de los trabajos estarán automatizados, con el impacto que eso conlleva para los diferentes trabajadores, que podrían perder puestos de trabajo en favor de profesionales del software que serán necesarios para afrontar el proceso de transición, y posterior mantenimiento de esta digitalización.

Los perfiles STEM (carreras relacionadas con ciencia, tecnología, ingeniería y matemáticas) van a ser cada vez más demandados en los próximos años, con el hándicap de que el número de nuevos profesionales relacionados con estas áreas disminuye cada año. Si vemos los datos de empleos relacionados con alta tecnología en nuestro país, y los comparamos con el resto de países europeos, observamos que el porcentaje de empleos de este tipo es muy pobre en la gran mayoría de regiones españolas, a excepción de Madrid y, en menor medida, la zona noreste de la península.

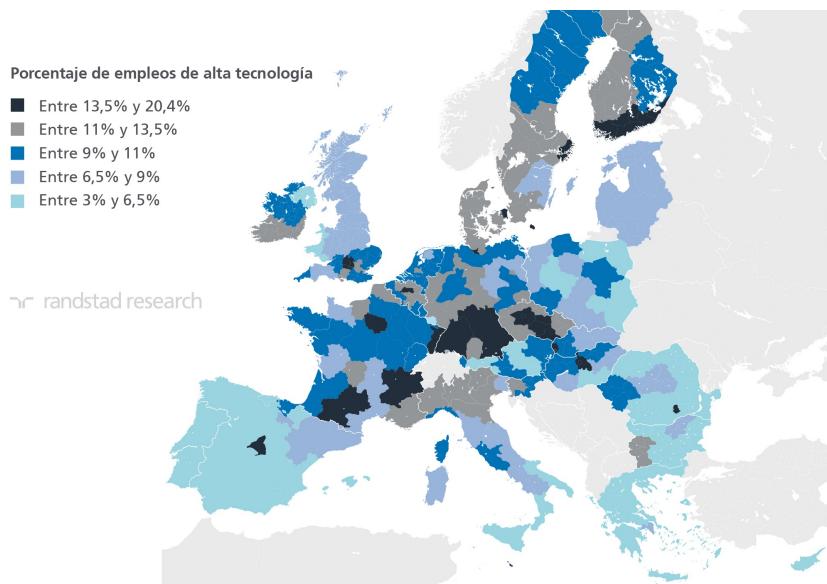


Figura 2: Empleos de alta tecnología en Europa

1.2. Software en robótica

Lograr el comportamiento esperado en un sistema robótico requiere conocer las distintas partes de las que se compone. Construir un robot supone la interacción de muchos componentes entre sí, formados a su vez por piezas más pequeñas. Entre ellos se encuentran los sensores, los actuadores y el computador.

En primer lugar, es necesario poder obtener información del entorno a través de la medición de diversas magnitudes físicas. Los sensores deben estar diseñados para obtener las más adecuadas para la tarea a realizar. Las cámaras, sensores láser, sensores de temperatura, de posición o de aceleración son algunos ejemplos.

La percepción de lo que hay alrededor requiere procesar esos datos mediante un computador para que sean útiles, es decir, se tomen decisiones y se den órdenes a los actuadores, que son los componentes encargados de interactuar con el mundo a través del movimiento.

El software constituye por tanto el elemento fundamental del robot, dotándolo de inteligencia y determinando su comportamiento. Para lograrlo de una manera adecuada, es necesario adaptarse a algunos requisitos. En primer lugar, el hecho de interactuar con un mundo, el real, que se encuentra en constante movimiento, requiere sistemas que respondan en tiempo real a dichos cambios. Para ello, se requiere hacer uso de la concurrencia para poder tomar decisiones con la mayor rapidez posible, algo crítico teniendo en cuenta la necesaria comunicación entre los distintos componentes.

De cara al usuario, es necesario construir interfaces que faciliten la interacción con los robots y la depuración de los diferentes problemas que puedan surgir. Estos pueden estar derivados por el hecho de tener que adaptar los programas al hardware heterogéneo existente, o la dificultad que presenta la reutilización de software de fuentes externas.

Actualmente, la implementación del comportamiento de los robots se realiza mediante software de control. En este contexto de desarrollo es muy útil un entorno que facilite la gran complejidad a la que se enfrenta el ingeniero. En informática, se denomina *middleware*

a un software que hace de enlace entre el sistema operativo y las aplicaciones [Figura 3]. En robótica, este software está diseñado para ocultar parte de la complejidad de bajo nivel, como la comunicación entre los distintos componentes. Así, el desarrollador puede diseñar y desarrollar sus soluciones de forma más eficiente. EN los siguientes párrafos se presentan algunos de los *middlewares* más importantes, y sus principales objetivos:

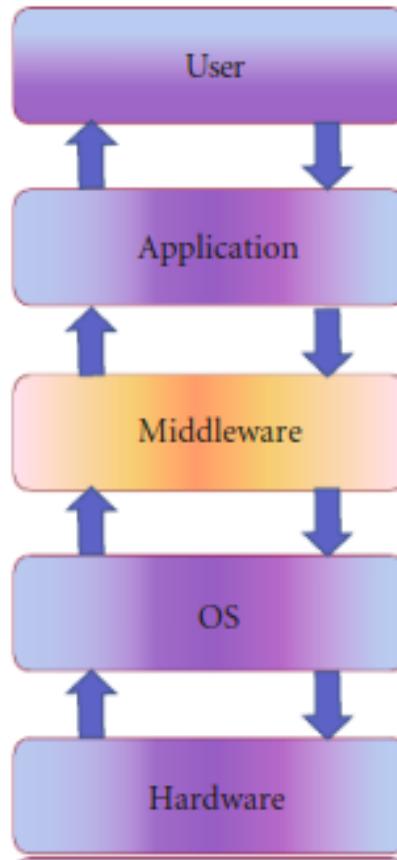


Figura 3: Situación del *middleware* en una arquitectura software

- ROS [20]: Robot Operating System continua creciendo hacia su objetivo de ser un estándar en la industria. Se trata de una solución de código abierto. Ofrece robustez, rapidez, abstracción, colaboración entre usuarios y una gran variedad de paquetes y funcionalidades, facilitando la labor de desarrollo. Se estima que en los próximos años supere el umbral y más del 50 % de los robots comerciales estén basados en él.
- Orocó [15]: Open Robot Control Software centra sus esfuerzos en ofrecer resultados en tiempo real y enfocado principalmente en robots industriales. Para ofrecer algunas funcionalidades, se apoya en otros *middlewares* similares. Es multiplataforma.
- CLARAty [2]: Promueve la reutilización de software, además de la portabilidad, modularidad y flexibilidad. Surge de la colaboración entre varias instituciones, entre ellas la NASA.
- MRDS [11]: Microsoft Robotics Developer Studio fue el primer intento de la compañía estadounidense por entrar de lleno en el mundo de la robótica. Su lanzamiento se produjo poco antes que el de ROS, por lo que nunca ganó suficiente atención.

1.3. Docencia en robótica

En un mundo necesitado de profesionales de la tecnología, pero cuyos individuos no van a cubrir la demanda de empleos en los próximos años, es necesario hacer accesible el conocimiento. Aquí es donde la docencia, en cualquiera de sus formas, cobra vital importancia. A continuación se comentarán algunos aspectos relacionados con la enseñanza en robótica en la actualidad.

1.3.1. Robótica en la escuela

Los conocimientos de robótica y programación han estado, hasta fechas muy recientes, alejados de las escuelas, donde aún su influencia es aún limitada. En las universidades, la robótica está presente solamente en algunas carreras especializadas. Ahora bien, ¿cómo es posible que herramientas básicas para un importante porcentaje de los empleos que vienen en los próximos años hayan comenzado a implantarse hace tan poco tiempo? [18]. Es una pregunta compleja en la que influyen muchos factores, aunque podemos afirmar que algunos de ellos se derivan del hecho de estar hablando de un área de conocimiento joven. Otros factores pueden deberse a la necesidad de cambios legislativos, a necesitar unas bases de conocimiento sólidas en el profesorado, y probablemente un presupuesto superior a otras áreas de enseñanza.

Si tenemos en cuenta que los futuros profesionales están hoy en las aulas, es necesario mostrar la robótica de una manera cercana, comprensible y que permita generar entusiasmo en los alumnos interesados. Al fin y al cabo, las decisiones que se toman en las últimas etapas de estudios previos a la universidad, son claves para decidir el futuro, y para hacerlo con criterio el alumno necesita el mayor abanico de conocimiento y opciones posible. La robótica educativa debe ser más relevante en los próximos años si se quieren formar y educar profesionales para la Cuarta Revolución Industrial. Entre las competencias que aporta al alumnado, están algunas tan necesarias para otros aspectos profesionales y vitales como el trabajo en equipo, el pensamiento crítico, la creatividad y la resolución de problemas. Existen algunas empresas que tratan de facilitar esta tarea ofreciendo robots educativos. Por ejemplo:

- Sphero SPRK+: Enfocada a los más pequeños, programable desde aplicaciones móviles a través de bloques, permite planificar rutas para la bola inteligente. Tiene un diseño transparente para poder ver y aprender de su interior.
- Dash: Solución para niños basada en programación con bloques y ampliable con accesorios
- Zowi: La propuesta educativa de BQ tiene su propia interfaz para la programación por bloques, llamada Bitbloq. La app para controlar el robot tiene como objetivo demostrar que se puede aprender jugando.
- Mindstorms: La línea de LEGO dedicada a construir robots a partir de sus míticas piezas, permite después aprender a programarlos a través de bloques para que realicen diversas acciones.



Figura 4: Sphero SPRK+ y LEGO mindstorms



Figura 5: Programando un ritmo musical mediante bloques

En el ámbito universitario, los dispositivos conectados permiten hoy en día acceder a un sinfín de información a través de Internet, ofreciendo a aquel que tenga interés en un tema la posibilidad de aprenderlo por sus propios medios. Las principales plataformas de robótica tienen disponibles sus propios tutoriales, donde destacan los de ROS o Robotic-Academy, aunque también se pueden encontrar en otras plataformas de cursos más generalistas. Otros recursos especializados son los disponibles en Robot Ignite Academy, pueden resultar de gran ayuda.

1.3.2. El entorno Robotics-Academy

Dentro de la organización sin ánimo de lucro JdeRobot[9], dedicada al desarrollo de software para robótica e Inteligencia Artificial, encontramos un conjunto de prácticas *open-source* enfocadas al aprendizaje de distintos ámbitos relacionados con la robótica a través de código python, compatibilidad con ROS y el uso del simulador Gazebo. Se trata de soluciones documentadas y en constante crecimiento, que han llegado a participar en el Google Summer of Code en tres ocasiones(2015,2017,2018). Cada práctica enfrenta un problema típico diferente, dónde el estudiante encuentra una plantilla que le abstraerá de toda la complejidad de bajo nivel del *middleware* de manera que sólo deba preocuparse de sus algoritmos. Aunque se utiliza el simulador Gazebo para observar el comportamiento del robot con la solución, la adaptación para ejecutar las prácticas en robots reales es sencilla debido a que toda la configuración de la estructura y controladores del robot están definidos. A continuación se presentan algunas prácticas disponibles en Robotics-

Academy:

El ejercicio de visión color filter, permite configurar un filtro de color para realizar un filtrado sobre un objeto en las imágenes proporcionadas por un fichero de vídeo. Por último, se debe detectar el movimiento de dicho objeto. Utiliza ICE como protocolo de comunicaciones y OpenCV para manejar las imágenes recibidas. Sobre tratamiento de imagen podemos encontrar también *Follow Face*, que se encarga de detectar y perseguir caras en los fotogramas obtenidos por la cámara de vídeo[23].

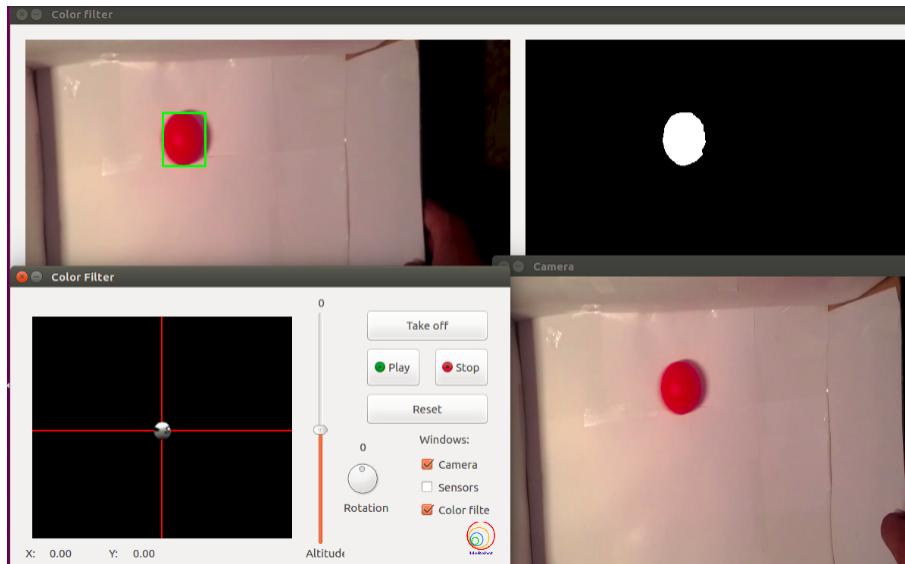


Figura 6: Filtro de color de la práctica color filter

En la práctica Drone cat&mouse dos drones deben jugar al gato y el ratón. Uno de ellos debe moverse de forma impredecible mientras el otro debe detectar el movimiento del otro y perseguirle, haciendo uso de las cámaras que lleva integradas y el procesamiento en tiempo real de las mismas. Este ejercicio fue utilizado en la competición *program-a-robot*, celebrada en 2018 dentro del programa IROS(International Conference on Intelligent robots and Systems) y promovida por el grupo de robótica de la Universidad Rey Juan Carlos.

Entrando en el terreno de la autolocalización y aprovechando la potencia de los sensores de profundidad, la práctica Vacuum-cleaner with visualSLAM permite simular el comportamiento de un robot aspirador de limpieza, detectando las partes de la casa por las que ha pasado y completando de forma eficiente el trabajo programado. En el mismo campo, la práctica Aspiradora autónoma [25] permite familiarizarse con algoritmos de navegación.

También hay ejemplos relacionados con los coches autónomos, como Visual follow-line behavior on a Formula1 que permite a los estudiantes programar un Formula 1 que siga la línea roja en el centro de la carretera para dar una vuelta al circuito. Existen alternativas similares, que emplean drones en lugar de coches para seguir una determinada carretera[24].

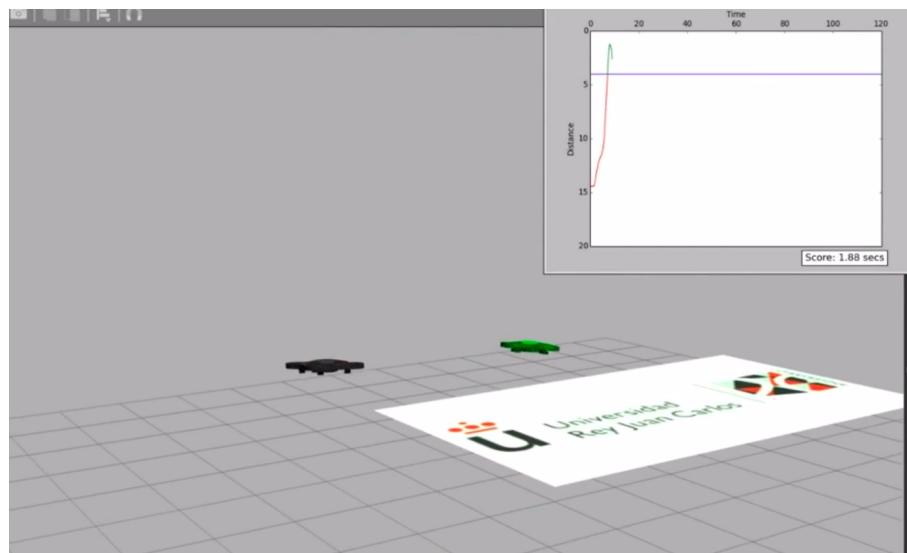


Figura 7: Persecución de drones en cat&mouse

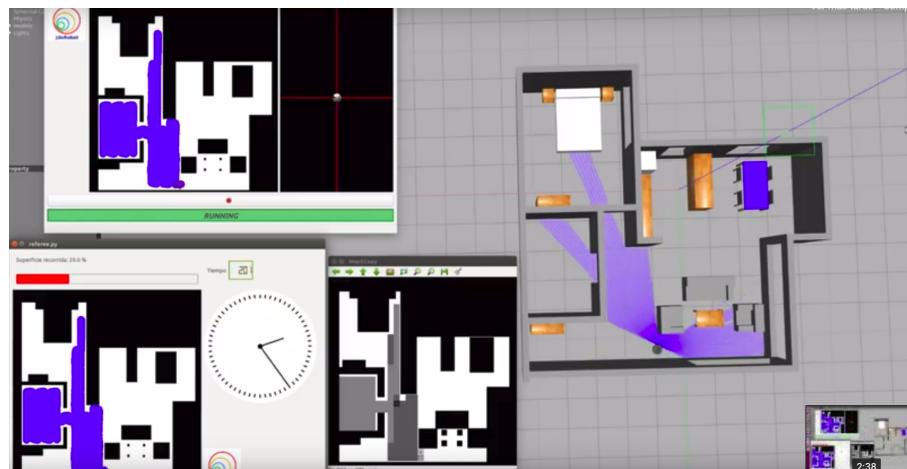


Figura 8: Mapeo generado en tiempo real en la práctica de la aspiradora

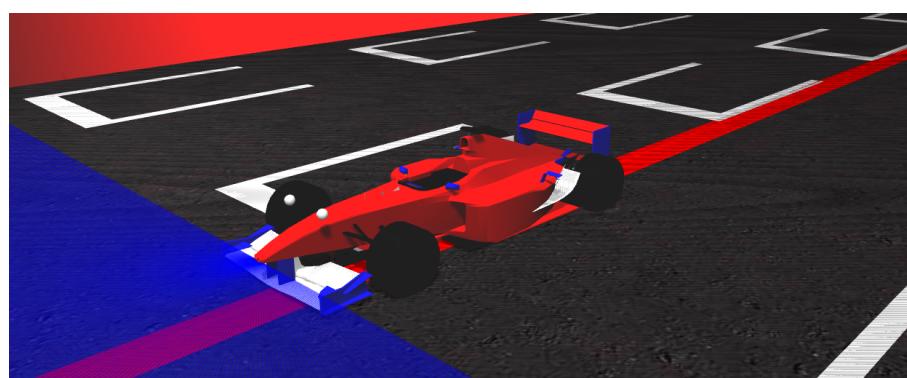


Figura 9: Mundo Gazebo para la práctica de conducción con Fórmula 1

Capítulo 2

Objetivos

Comprendido el contexto en que se ha desarrollado el proyecto, a continuación se profundizará en los objetivos iniciales del mismo, los requisitos necesarios para el desarrollo de la solución, la metodología de trabajo que ha sido necesaria para conseguirlos y el plan de acción requerido para ser capaz de completar el proceso.

2.1. Objetivos

El objetivo principal de este trabajo consiste en añadir una nueva práctica al entorno educativo Robotics-Academy de JdeRobot. Dicha práctica tiene como meta profundizar en la planificación de trayectorias de un brazo robótico. Para realizar dicha práctica se utilizará MoveIt!, el framework de ROS enfocado a la robótica que ofrece herramientas para gestionar movimientos, manipulación, control, físicas y navegación. Además, se deben analizar y filtrar imágenes para detectar objetos y colores, para lo que será necesario conocer algunas de las posibilidades que ofrece la librería OpenCV.

Para obtener la solución final, será necesario controlar los movimientos de un robot PR2, el procesamiento de imágenes en tiempo real, la creación de modelos y mundos en el simulador Gazebo y la planificación y ejecución de trayectorias evitando obstáculos. Se ofrecerá al destinatario de la práctica educativa una plantilla donde podrá añadir su código, y que le abstraerá de los detalles de implementación más complejos, a los que podrá acceder posteriormente si desea profundizar en alguna parte.

De esta forma el estudiante se podrá centrar en su desarrollo sin distraerse con problemas que ya han sido contemplados y resueltos, tomando contacto con la programación en robótica utilizando lenguaje Python y conociendo cómo funciona MoveIt!, a través de una práctica completa cuyo eje es la planificación.

Además, el software debe realizar acciones en tiempo real, con valores aceptables para las ejecuciones en el simulador. Por último, debe ser capaz de obtener trayectorias válidas en situaciones con varios obstáculos.

2.2. Requisitos

A continuación, se enumeran los diferentes recursos de software y hardware que han servido como base para la consecución de los objetivos:

2.3. METODOLOGÍA

- El sistema operativo utilizado es Ubuntu 16.04 LTS.
- El *middleware* de referencia para trabajar es ROS en su versión Kinetic, debido a motivos de compatibilidad con su solución MoveIt!.
- El simulador de referencia es Gazebo 7, la versión recomendada para trabajar con ROS Kinetic.
- El lenguaje de programación utilizado para los desarrollos es Python en su versión 2.7.12, nuevamente por razones de compatibilidad. Además, son necesarias nociones básicas de C++ para comprender el código de los repositorios de ROS.
- El hardware utilizado debe ser suficiente para soportar el renderizado y procesamiento de la simulación sin sobresaltos. Un equipo con 8GB de RAM, procesador i7 de octava generación y gráficos Intel UHD Graphics 620 cumple las expectativas.

2.3. Metodología

La metodología de trabajo se ha basado en el modelo en espiral propuesto por primera vez en 1986 por el ingeniero Barry Boehm [22]. Es un modelo iterativo, que enfoca los proyectos desde un punto de vista evolutivo, creciendo a partir de la realimentación y la definición periódica de objetivos. La propuesta original de Boehm se compone de las siguientes fases [Figura 10], realizadas en bucle:

- Determinar objetivos. Incluye definir las tareas pendientes, las especificaciones, las restricciones y las estrategias para evitar riesgos.
- Análisis de riesgos. Consiste en estudiar las amenazas de cada etapa para tratar de minimizar los potenciales problemas y maximizar la eficiencia del trabajo.
- Desarrollo y testing. En esta etapa, el ingeniero genera su solución, la verifica, y la valida. Es decir, se asegura de que se está desarrollando el producto como se ha definido y de que funciona como se espera que funcione. Parte de las tareas de pruebas consisten en anticipar errores, siendo éste uno de los principales ahorros de costes [12] [28] que podemos encontrar en un proyecto software.
- Planificación. Se revisa el estado de los objetivos y se definen las tareas a llevar a cabo con los recursos disponibles para la iteración en curso.

Algunas de las ventajas de este modelo son la capacidad de adaptación, la reacción rápida frente a problemas, la reducción de riesgos y el enfoque realista. Sin embargo, también podría generar algunos inconvenientes, como la necesidad de un alto conocimiento de los posibles riesgos relativos al proyecto para no derivar en problemas en etapas avanzadas del desarrollo.



Figura 10: Representación gráfica del modelo en espiral para desarrollo software

2.4. Plan de acción

Al enfrentar un proyecto de características técnicas, y marcar unos determinados objetivos, se debe marcar un plan de acción que determine cómo alcanzar dichos objetivos, y de qué manera se lograrán. Para completar la práctica de planificación, se han requerido los siguientes pasos:

- Estudiar el lenguaje de programación C++ para poder comprender, y afrontar en caso de necesidad, los desarrollos necesarios para trabajar con robots en el contexto comentado.
- Familiarización con el entorno educativo de JdeRobot a través de diversas prácticas para tomar contacto con la robótica y comenzar a comprender sus principios básicos. Este paso sienta las bases para poder comenzar a desarrollar el proyecto.
- Conocer el simulador Gazebo a través de los tutoriales oficiales, y búsqueda de un robot comercial soportado completamente en el simulador. Incluye revisar los lenguajes de descripción URDF y SDF para construir mundos y modelos, y también cómo construir plugins para Gazebo.
- Descubrir y profundizar en el framework de desarrollo software ROS, nuevamente a través de los tutoriales oficiales. Entre las acciones más importantes están construir espacios de trabajo, entender qué es un nodo ROS, como funciona ROS topics o aprender a generar ficheros roslaunch.
- Comenzar a trabajar con MoveIt! a través de los tutoriales oficiales para ROS Kinetic, inicialmente para el robot PR2 y recientemente migrados a Panda. Algunas de las tareas clave han sido aprender a configurar robots, planificar y ejecutar movimientos, evitar obstáculos, añadir restricciones de planificación o utilizar correctamente el visualizador RVIZ.

2.4. PLAN DE ACCIÓN

- Utilizar los conocimientos adquiridos para resolver los problemas necesarios y desarrollar una solución a la práctica de robótica escogida, ofreciendo una plantilla al usuario final y un resultado satisfactorio.

Capítulo 3

Infraestructura

En este capítulo se detalla el software necesario para el desarrollo de las aplicaciones construidas. Desde ROS Kinetic y JdeRobot como *middlewares* robóticos, pasando por la herramienta MoveIt! para facilitar el trabajo con robots(PR2) hasta el simulador Gazebo. Todo ello corriendo en el sistema operativo Ubuntu 16.04 y desarrollado en Python y C++. Las comunicaciones entre los distintos componentes utilizan el motor de comunicaciones ICE además de ROS topics.

3.1. PR2

El robot sobre el que se han desarrollado las aplicaciones es el PR2 de Willow Garage, empresa especializada en robótica responsable también del software ROS y otras herramientas open-source. Se compone de dos brazos(derecho e izquierdo)capaces de generar un amplio abanico de oportunidades de movimiento que no serían posibles con un único brazo, y que terminan en una pinza o 'gripper' que permite agarrar objetos. El nombre viene de Personal Robot 2, haciendo referencia a la idea de que fuera un asistente en casa y no un robot industrial como la mayoría de los comercializados hasta el momento. Su diseño modular permite integrarlo con otros grippers, brazos o sensores. Su muñeca tiene dos grados de movimiento, que se suman a la libertad que aportan las articulaciones del hombro y el codo para permitir prácticamente cualquier movimiento necesario para una labor doméstica. Además de su estructura, una característica fundamental del robot PR2 son las cámaras y láseres que permiten conocer el entorno y actuar según lo que percibe: detectar objetos, saber a qué distancia se encuentran e identificar zonas objetivo permiten llevar a cabo acciones como abrir una puerta o llevar objetos de un lugar a otro.

3.1.1. Microsoft Kinect

La cámara Kinect permite conocer el entorno a través de imágenes procesadas en tiempo real. Contiene un sensor de color(RGB) y uno de profundidad, y se sitúa en la parte superior del robot.

3.2. GAZEBO

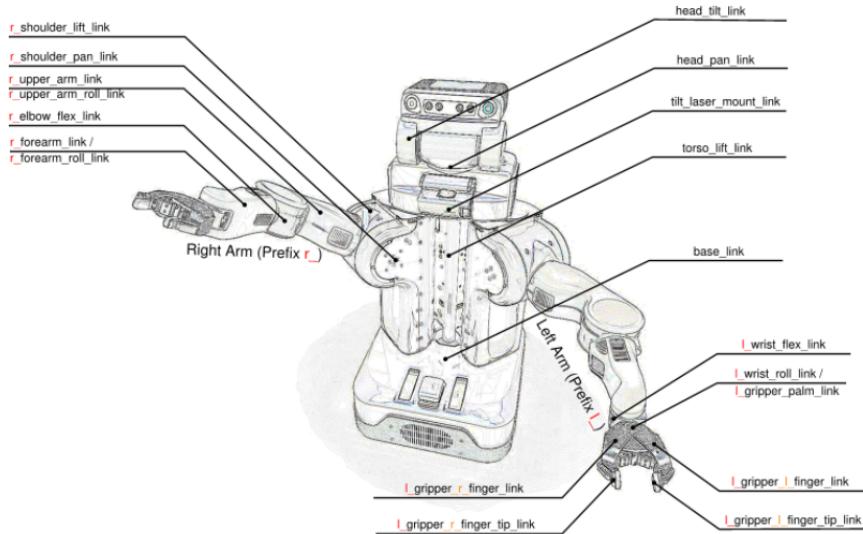


Figura 11: Estructura del robot PR2



Figura 12: Cámara Kinect integrado sobre la cabeza del robot PR2

3.2. Gazebo

Este simulador open source nació en el año 2002 en la Universidad del Sur de California(USC) como proyecto entre un profesor y uno de sus estudiantes. El objetivo era poder trabajar con robots bajo condiciones de alta fidelidad, pensando principalmente en entornos exteriores. Se comenzó a integrar con ROS en 2009 y dos años después la citada Willow Garage comenzó a financiar el desarrollo para este software, del que se hizo cargo la Open Source Robotics Foundation(OSRF). Gazebo fue adaptado para participar en la prestigiosa DARPA Robotics Challenge (DRC) en el año 2013. Hay una comunidad activa detrás del proyecto que permite introducir mejoras, resolver bugs y consultar dudas técnicas.

Como simulador de robótica, lo que ofrece es crear aplicaciones para robots sin necesidad de depender de la máquina física. Así, las aplicaciones creadas para un modelo concreto podrán utilizarse posteriormente en el mundo real sin necesidad de realizar modificaciones, disminuyendo tanto los costes tanto económicos como derivados de la peligrosidad de las

pruebas en fase de desarrollo. Ofrece visualización 3D y un potente motor de físicas para, en el caso de un brazo robótico, poder determinar parámetros como fricción, rango de movimiento, colisiones etc. La simulación nace a partir de un fichero launch que arranca

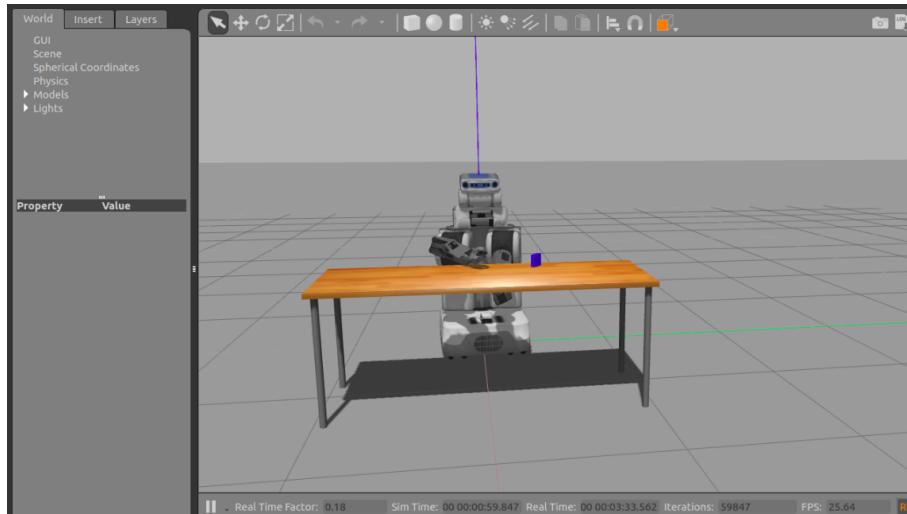


Figura 13: Mundo sencillo en Gazebo con un robot PR2, una mesa, y un objeto

Gazebo. El entorno de la simulación está compuesto por un world que contiene diferentes models. Estos modelos están definidos en un fichero con formato SDF con los siguientes componentes principales:

- Links: contiene las propiedades físicas de un trozo del modelo, incluyendo algunas de visualización y colisión, sensores, inercias o iluminación.
- Joints: conexión entre dos links.
- Plugins: librería externa que controla un modelo.

La versión utilizada para este proyecto es Gazebo7.

3.3. JdeRobot

Este software open source permite desarrollar aplicaciones en el campo de la Robótica a partir de código C++, Python o JavaScript. La compatibilidad con ROS, principalmente con la versión Kinetic es una de las grandes ventajas de esta herramienta. Los diferentes nodos o componentes se comunican a través del framework ICE(Internet Communications Engine) o de mensajes ROS. Ambas opciones permiten realizar dichas comunicaciones de forma agnóstica al lenguaje en el que están desarrollados los componentes. De esta estructura parte JdeRobot-Academy, una iniciativa para el aprendizaje en Robótica y Visión Computacional que incluye múltiples ejercicios para que los estudiantes puedan entender y añadir su propio código. JdeRobot contiene una amplia variedad de componentes y librerías que pueden ser reutilizados por la comunidad de desarrolladores. La última versión estable es la 5.6.4, lanzada en mayo de 2018. Uno de los últimos hitos es su participación en el Google Summer of Code 2018.

3.4. ROS Kinetic

Framework para el desarrollo de software para robots. Nació en 2007 en el Laboratorio de Inteligencia Artificial de Standford. La arquitectura de ROS(Robot Operating System) se basa en nodos que se comunican a través de mensajes, lo que nos permite obtener grafos fácilmente. Es un sistema de código abierto que encarga de mantener la OSRF(Open Source Robotics Foundation). Aporta al usuario herramientas como por ejemplo planificación de movimiento(MoveIt) o reconocimiento del entorno y visualización (Rviz), además de soporte para un amplio abanico de Robots. La versión Kinetic fue lanzada en mayo de 2016, enfocada al uso desde Ubuntu 16.04. La versión Gazebo7 es la recomendada en ROS Kinetic para este simulador. -Estructura y comunicaciones Un paquete de ROS agrupa varios programas o nodos con funcionalidades similares. Estos nodos se comunican a través de mensajes, llamados ROS Topics, que les permiten interactuar entre sí. ROS Core es la herramienta encargada de arrancar el nodo máster y gestionar todas las comunicaciones, por lo que es la primera que debe ser arrancada. Es lanzado automáticamente con los ficheros .launch, que serán muy habituales en el desarrollo.

```
~$ roscore
```

Para ejecutar un nodo, necesitaremos el paquete y el nombre del nodo:

```
~$ rosrun turtlesim turtlesim_node
```

Una vez lanzado el nodo, podemos publicar topics de un tipo concreto que el nodo comprenda, consiguiendo así que realice una acción determinada. Estas comunicaciones son la base del desarrollo con robots en ROS.

```
~$ rostopic pub -1 /turtle1/cmd_vel geometry_msgs/Twist --  
  '[2.0, 0.0, 0.0]' '[0.0, 0.0, 1.8]'
```

También se pueden obtener por línea de comandos los mensajes publicados en un topic determinado.

```
~$ rostopic echo /turtle1/cmd\_\_vel
```

ROS dispone de herramientas como rqt que permiten observar grafos de las comunicaciones actuales. Si seguimos con el ejemplo de turtlesim:



```
$ rosrun rqt_graph rqt_graph
```

Podemos ver en cualquier momento los nodos o topics disponibles con los siguientes comandos:

```
~$ rosnode list  
~$ rostopic list
```

3.5. ICE

Se trata de un framework RPC(Remote Procedure Call) compatible con lenguajes como C++, C#, Java, JavaScript o Python. Nace a partir de un modelo cliente-servidor con el objetivo de permitir la comunicación entre distintos lenguajes y sistemas operativos. Esta abstracción ofrece al desarrollador la capacidad de despreocuparse por cómo se gestionan las comunicaciones entre los diferentes módulos para así enfocarse en el problema a resolver. Permite comunicaciones bidireccionales, tanto síncronas como asíncronas.

Fue diseñado para aplicaciones que requieren un performance y escalabilidad exigentes, minimizando el consumo de CPU y ancho de banda. IceSSL permite añadir seguridad a las comunicaciones, a través de autenticación y encriptación de los datos que viajan en las llamadas.

3.6. C++

Este lenguaje de programación fue creado por Bjarne Stroustrup en 1979, que lo bautizó en sus inicios como ‘C con clases’. No en vano tiene una sintaxis similar a la de C, manteniendo el vínculo con el lenguaje del que proviene y añadiendo mecanismos de programación orientada a objetos a su predecesor. Su nombre además hace referencia a ese C incrementado, mejorado. Es un lenguaje fuertemente tipado, combinado y portátil. Se rige por un estándar de ISO(International Organization for Standardization) cuya última versión es C++17, lanzada ese mismo año.

El hecho de ser un lenguaje con un grado de complejidad elevado supone una gran versatilidad y potencia, pero también una curva de aprendizaje elevada para dominarlo. Algunos IDEs recomendados para trabajar con C++ son VisualStudio y Code::Blocks. Como curiosidad, se trata del tercer lenguaje de programación más popular en 2018 según el conocido ranking de TIOBE, sólo por detrás de Java y C. Alcanzó su máximo de popularidad 2003 según los datos de esta empresa de software. (<https://www.tiobe.com/tiobe-index/>)

3.7. Python

Fue creado en los años 80 por Guido van Rossum. Es un lenguaje de programación interpretado, que utiliza tipado dinámico. Su sintaxis está enfocada a ser fácilmente legible, lo que hace que tenga una curva de aprendizaje suave en relación con otros lenguajes como Java o el propio C. Soporta programación orientada a objetos y es multiplataforma.

Una característica interesante es que puede extenderse con módulos de C o C++. Según el mencionado ranking TIOBE, Python es el cuarto lenguaje de programación en popularidad llegando al máximo nivel en 2011.

Se ha utilizado la versión 2.7.12 debido principalmente a su compatibilidad con ROS Kinetic. En este lenguaje se han desarrollado las aplicaciones de planificación de movimientos y pick&place.

3.8. OpenCV

Librería de código abierto enfocada a proyectos de visión artificial, con interfaces disponibles para varios lenguajes de programación(C++, Python, Java). Open Source Computer Vision Library es muy utilizada, con un número de descargas de alrededor de 15 millones y una comunidad de 47000 seguidores. Su integración en el proyecto permite aprovechar, en un entorno robótico, toda la potencia del análisis de imágenes.

3.9. MoveIt!: Motion Planning Framework

MoveIt! nació en octubre del año 2011 con la idea de agrupar todos los avances relacionados con planificación de movimientos, manipulación, percepción 3D, cinemática, control y navegación en una única herramienta. Actualmente es el software open-source más utilizado para manipulación con robots, con más de 65 autómatas soportados. Tiene un nodo principal llamado move_group que actúa como integrador, permitiendo a todos los componentes comunicarse entre sí realizando las acciones soportadas. Se puede acceder a move_group por tres vías: C++, Python y RVIZ. Como se puede ver en la figura,

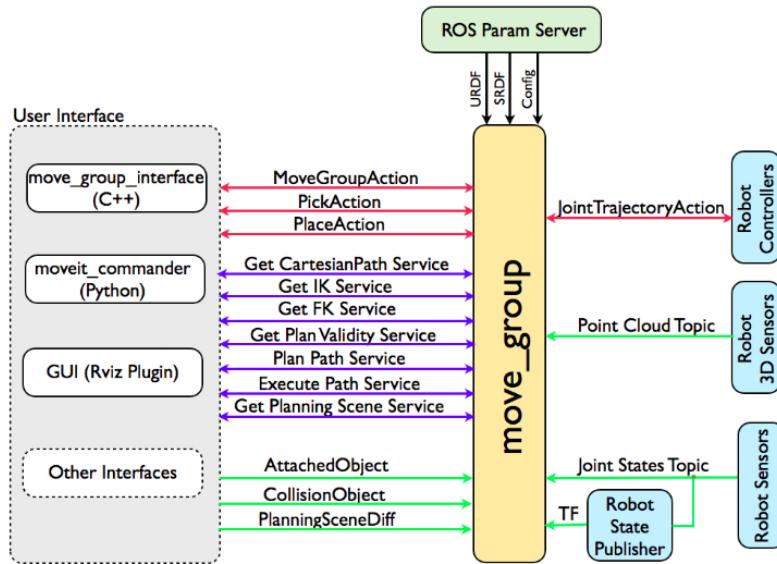


Figura 14: Resumen esquematizado de las posibles interacciones que ofrece move_group
move_group necesita varios parámetros de entrada:

- URDF(Universal Robot Description Format): Es el formato que utiliza ROS para la descripción del robot. Se ha utilizado el del paquete ‘pr2_description’, correspondiente al autómata utilizado.
- SRDF(Semantic Robot Description Format) complementa al URDF, añadiendo más información como grupos de joints, configuraciones por defecto para el robot o chequeo de colisiones. Para construir este fichero, se recomienda utilizar el MoveIt! Setup Assistant. Esta herramienta permite configurar cualquier robot para ser utilizado en MoveIt! A través de una interfaz gráfica que simplifica el trabajo.

- MoveIt! Configuration: Incluye otros ficheros de configuración específicos de MoveIt!, normalmente generados también a través del Setup Assistant. Contienen información necesaria para planificación de movimientos, cinemática o percepción del entorno.

Las principales interfaces de MoveIt! Permiten acceder a las diferentes funcionalidades a través de código C++. Sin embargo, existe un paquete llamado moveit_python que permite acceder a las principales interfaces de MoveIt! utilizando Python:

- MoveGroupInterface: Utilizada para acceder a move_group y mover el brazo.
- PlanningSceneInterface: Utilizada para añadir o eliminar objetos al entorno y cambiar su apariencia, ya sean conectados o de colisión.
- PickPlaceInterface: Utilizada para realizar acciones de pick&place.

Con el objetivo de conseguir la integración de MoveIt! con Gazebo, y poder así enviar los movimientos al robot real(o simulado), debemos configurar correctamente los controladores. Para ello necesitaremos dos ficheros clave: controllers.yaml y joint_names.yaml. En ellos se especifican el tipo de mensajes que el robot recibe y las joints que controla.

Capítulo 4

Ejercicio de planificación de trayectorias con brazo robótico

En este capítulo se describirán en detalle tanto el objetivo como el desarrollo técnico de la práctica ”Movimiento planificado de un brazo robótico” para el entorno Robotics-Academy. Incluye el enunciado de la misma, la infraestructura específica necesaria y la plantilla de código Python desarrollada para albergar la solución del estudiante.

4.1. Enunciado

El objetivo de este ejercicio consiste en un robot PR2 que debe mover la cabeza para observar un objeto y posteriormente planificar un movimiento con el brazo izquierdo sorteando obstáculos, hasta golpear un objeto final que será del mismo color que el observado al comienzo. Incluye por tanto la planificación de trayectorias con obstáculos en un entorno simulado, el reconocimiento de objetos analizando espacios de color, y la ejecución de movimientos. Se utilizará el paquete MoveIt! para la fase de planificación.

El escenario de la práctica contiene un mundo Gazebo(Fig. 15) que incluye dos zonas de acción(muestra y derribo) y un total de cuatro objetos, uno en la primera zona y tres en la segunda. Entre ellas, un robot PR2 y algunos obstáculos. Los objetos en la zona de derribo tendrán los colores rojo, verde y azul, y el de la zona de muestra tiene únicamente uno de los colores mencionados(azul por defecto).

La ejecución típica comienza con el robot PR2 moviendo la cabeza y observando la zona de muestra. En ese momento, debe ser capaz de obtener las imágenes de la cámara Kinect que incorpora, que serán analizadas mediante código Python para determinar si hay o no un objeto y de qué color es. Después, conociendo el color objetivo y las posiciones de los objetos en la zona de derribo, será capaz de planificar ,y posteriormente ejecutar, una trayectoria con su brazo izquierdo que le sitúe frente al objeto en la zona de derribo cuyo color coincide con el observado en la zona de muestra. Para ello será necesario gestionar la planificación de trayectorias evitando los obstáculos necesarios para, por último, realizar un movimiento de derribo.

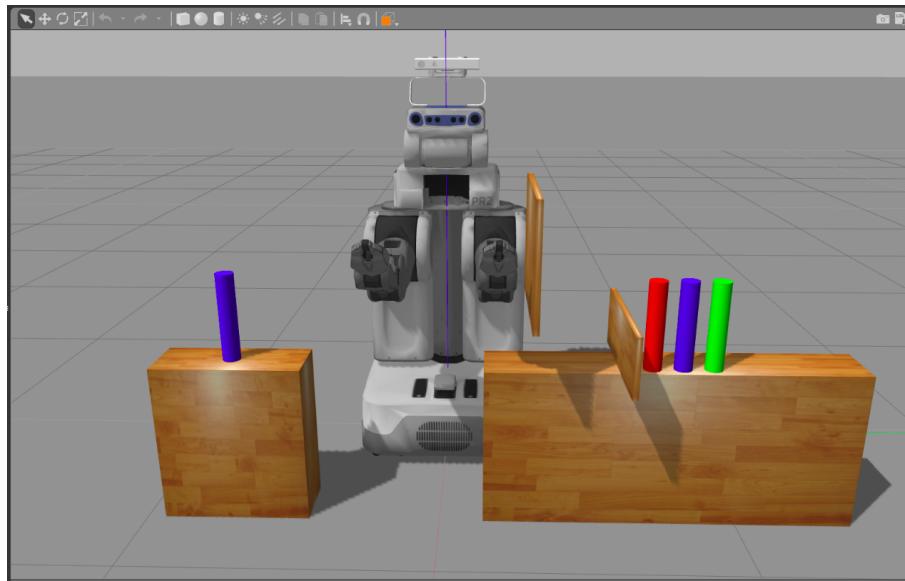


Figura 15: Mundo Gazebo que incluye todo lo necesario para el desarrollo de la práctica: zonas, objetos, robot y obstáculos.

4.2. Diseño

La estructura diseñada [16] tiene como núcleo el sistema ROS. A través de él, el nodo creado puede comunicarse con el simulador y las librerías externas. La interacción con el simulador debe ser bidireccional ya que ambos deben conocer que está ocurriendo al otro lado; el modelo del robot en Gazebo necesita recibir mensajes para saber hacia dónde moverse, así como la planificación de trayectorias dirigida por MoveIt! necesita saber la posición actual para enviar sus mensajes de movimiento.

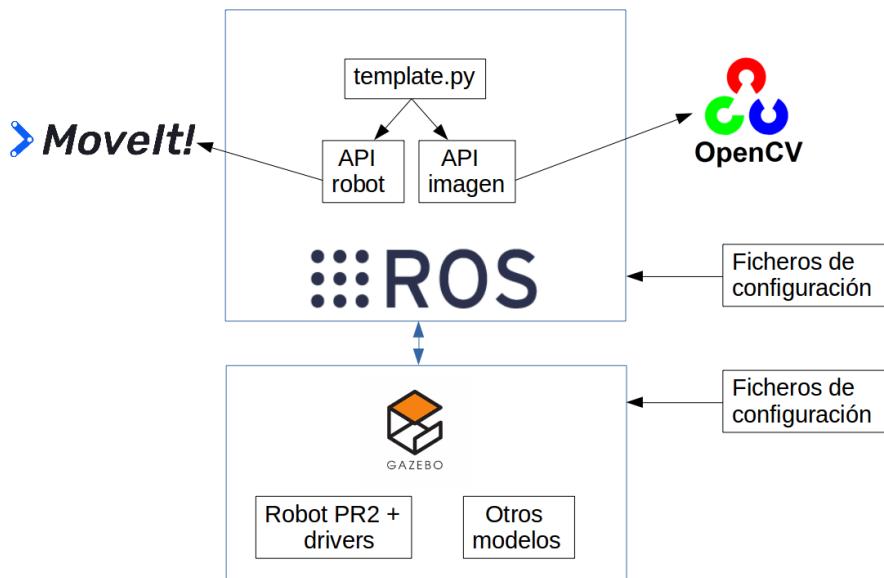


Figura 16: Esquema de la arquitectura software utilizada.

Por su parte, el simulador Gazebo contiene el modelo del robot, incluyendo sus controladores, así como el resto de modelos necesarios, como las zonas de acción, los objetos y los

obstáculos.

El nodo ROS principal, ofrece las APIs de manejo de imágenes y de manejo del robot, que son utilizadas desde la plantilla codificada en Python. Para ofrecer las funcionalidades disponibles a través de dichas interfaces de programación, son necesarias las librerías OpenCV y MoveIt!.

4.3. Infraestructura del ejercicio

Los elementos necesarios para lograr esta práctica completa de planificación son el mundo creado en Gazebo y todos los objetos estáticos que contiene (mesas, obstáculos, cilindros de colores), el robot PR2 y sus drivers para las articulaciones y la captura de imágenes, y los plugins utilizados para la captura de imagen.

4.3.1. Mundo Gazebo

En primer lugar, ha sido necesaria la creación de un mundo en Gazebo [\[15\]](#) que contiene los objetos a detectar y empujar, las dos zonas de acción, los obstáculos, el robot PR2 y la cámara Kinect.

La cámara Kinect se incorpora en la parte superior de la cabeza del robot, de forma que se encuentran completamente integrados y podemos controlar la visión del robot a través de movimientos en su parte superior.

Para lanzar este entorno, se debe ejecutar un fichero .launch [\[1\]](#), que pondrá en funcionamiento los modelos necesarios para las dos zonas de referencia, los modelos de los objetos y obstáculos, y el robot PR2 con todos sus controladores. La cámara Kinect integrada se pasa como parámetro, y está configurada para aparecer por defecto.

```
roslaunch pr2_gazebo tfg_launch.launch
```

Modelo PR2 enriquecido

El modelado principal del robot PR2 utilizado en esta práctica corresponde a los paquetes pr2_gazebo y pr2_common. Se trata de paquetes disponibles para ROS Kinetic, que contienen todo lo necesario para simular el robot de Willow Garage en Gazebo, como información sobre articulaciones, físicas, colisiones o dimensiones.

De cara a lograr el objetivo final, ha sido necesario hacer ligeras modificaciones. En primer lugar, para añadir una cámara Kinect [\[8\]](#) en la parte alta del modelo. Esta es una opción disponible en el paquete original pero que debemos configurar como opción por defecto (la original lanza el robot sin ninguna cámara), utilizando el argumento KINECT2 al lanzar el fichero .launch del robot PR2.

En segundo lugar, en la fase inicial de pruebas con reconocimiento de imagen en el entorno creado, se evidenciaban problemas a la hora de detectar los colores a causa del

Algoritmo 1 Fichero .launch para el mundo gazebo y el robot PR2.

```
<launch>
    <!-- start up world -->
    <arg name="gui" default="true"/>
    <arg name="headless" default="false" />
    <arg name="debug" default="false" />
    <arg name="paused" default="true"/>
    <arg name="KINECT2" default="$(optenv_KINECT2_true)" />
    <include file="$(find_gazebo_ros)/launch/empty_world.launch">
        <arg name="world_name" value="worlds/tfg_world.world"/>
        <arg name="gui" value="$(arg_gui)" />
        <arg name="headless" value="$(arg_headless)" />
        <arg name="paused" value="$(arg_paused)" />
        <arg name="debug" value="$(arg_debug)" />
        <arg name="use_sim_time" value="true" />
    </include>
    <!-- start pr2 robot with Kinect camera -->
    <include file="$(find_pr2_gazebo)/launch/pr2.launch">
        <arg name="KINECT2" value="$(arg_KINECT2)" />
    </include>
</launch>
```

sensor láser incluido por defecto en el modelo. Este generaba ruido en las imágenes RGB obtenidas dificultando la caracterización de los objetos, motivo por el cual también hubo que desactivarlo en los paquetes generados.

Para llevar a cabo estas modificaciones, es necesario conocer el concepto de overlaying [29], utilizado en ROS para hablar de modificaciones en el espacio de trabajo [30] del usuario sobre los paquetes originales descargados de los repositorios oficiales. De esta manera podemos hacer que ROS priorice nuestros propios repositorios, donde se habrán modificado los ficheros necesarios para lograr un determinado funcionamiento, como por ejemplo la desactivación del sensor láser.

Modelos adicionales

Se ha desarrollado un fichero .world[2] que define los modelos necesarios para la consecución de la práctica. Para cada uno de los elementos se establecen su posición, cómo será su visualización, sus criterios de colisión y sus dinámicas. Para ello se utiliza el formato SDF [21], open-source y basado en XML y que permite describir modelos de cualquier complejidad a través de sencillas etiquetas, desde esferas o prismas, hasta robots complejos.

Algoritmo 2 Fragmento del fichero .world que define el modelo de la zona de muestra en Gazebo

```

<model name="base_A">
  <pose>0.6 -0.7 0.275 0 0 0</pose>
  <static>true</static>
  <link name="link">
    <collision name="surface">
      <pose>0.0 0.0 0.0 0 0 0</pose>
      <geometry>
        <box>
          <size>0.2 0.5 0.55</size>
        </box>
      </geometry>
      <surface>
        <friction>
          <ode>
            <mu>10</mu>
            <mu2>10</mu2>
          </ode>
        </friction>
      </surface>
    </collision>
    <visual name="visual1">
      <pose>0.0 0.0 0.0 0 0 0</pose>
      <geometry>
        <box>
          <size>0.2 0.5 0.55</size>
        </box>
      </geometry>
      <material>
        <script>
          <uri>file:///media/materials/scripts/gazebo.material</uri>
          <name>Gazebo/Wood</name>
        </script>
      </material>
    </visual>
  </link>
</model>
```

4.4. Plantilla

Como se muestra en la sección anterior, el nodo Python será capaz de detectar a partir de una imagen de entrada si hay algo azul, rojo o verde delante de la cámara, y en caso afirmativo, determinar si es el objeto esperado, eliminando posibles errores provocados por ruido. Para ello, el nodo importa las librerías *cv2* y *rospy*. Además, para la planificación de trayectorias, se importan *moveit_commander* y los distintos mensajes de *MoveIt!*

4.4. PLANTILLA

necesarios para la planificación de trayectorias.

Existen dos ficheros Python, uno llamado *final_prototype_template_ready.py* y otro llamado *template.py*[3]. El primero contiene la mayor parte de la complejidad de la práctica, y tiene como objetivo abstraer al usuario final de los entresijos necesarios para hacer funcionar un brazo robótico con *MoveIt!* y gestionar imágenes con *OpenCV*, simplificando dichas tareas a través de algunas funciones sencillas. El segundo, será el fichero a llenar y ejecutar por el alumno:

```
rosrun moveit_tutorials template.py
```

El interfaz de programación accesible desde el fichero *template.py* ofrece funciones de manejo de imágenes y funciones de manejo del robot.

- API para el manejo de imágenes
 - *convert_to_cv2(msg)*: convierte el mensaje recibido en el callback en una imagen preparada para trabajar con OpenCV. Para ello utiliza la librería de ROS llamada *CvBridge*[5], creada precisamente con este propósito. La función recibe una imagen de ROS y la convierte a una OpenCV, con formato bgr8.
 - *convert_to_hsv(img)*: convierte una imagen BGR al espacio de color HSV utilizando la función de OpenCV *cvtColor* [16], que recibe como parámetros una imagen de entrada y un código de transformación.
 - *save_image(img)*: guarda la imagen en formato .jpeg en un directorio local, utilizando la función *imwrite* de OpenCV, que recibe una imagen y el nombre destino.
 - *def detect_objects(img, lower, upper, color)*: recibe como parámetros los valores mínimos y máximos de HSV y aplica la máscara para detectar objetos de color teniendo en cuenta su tamaño y eliminando falsos positivos. Se utiliza la función *inRange* para generar una máscara según los valores dados para el espacio de color HSV, y *bitwise_and* para observar el resultado en el color original. Para determinar si hay un objeto utiliza *findContours*, que detecta puntos consecutivos en la imagen. Sin embargo, para descartar que dicha secuencia se deba a ruido, se ha establecido un valor por defecto de 500 puntos consecutivos para ser considerado un objeto.
- API para el manejo del robot
 - *look_at_object(x,y,z)*: Mueve la cabeza del robot PR2 configurando y enviando un mensaje *PointHeadGoal* con los parámetros recibidos. Para ello utiliza la librería *SimpleActionClient*, que permite enviar un mensaje a través de ROS topics, en este caso al controlador de la cabeza del robot.
 - *Subscriber(image_topic, Image, image_callback)*: es el encargado de llamar al método *image_callback* para que realice acciones cuando se recibe una imagen en el topic definido. Para la cámara Kinect utilizada, se corresponde con */head_mount_kinect2/rgb/image_raw*.

- `start_planning()`: instancia los objetos necesarios para obtener y modificar valores del robot, la escena y la planificación. Estos son, respectivamente, *RobotCommander*, *PlanningSceneInterface* y *MoveGroupCommander*. Añade los objetos, zonas y obstáculos a considerar como obstáculos en la planificación y configura tanto el tiempo máximo como el número de intentos para la misma.
- `set_orientation_constraints(link, orientation, x_tol,y_tol,z_tol)`: define restricciones de tolerancia para cada eje de una determinada articulación del robot. Para ello utiliza un mensaje de MoveIt! de tipo *orientation_constraints*.
- `move_to_goal(color)`: planifica y ejecuta el movimiento al objeto destino del color pasado como parámetro. Conoce las posiciones destino y divide en dos el proceso, ya que en primer lugar esquiva los obstáculos y se sitúa frente al objeto destino, y en segundo lugar lo derriba. Para planificar las trayectorias utiliza los métodos *set_pose_target* y *plan* de la librería *MoveGroupCommander*.

Algoritmo 3 Plantilla simplificada para la práctica de planificación de trayectorias

```

def image_callback(msg):
    global count
    count=count+1
    if count==1:
        cv2_img = convert_to_cv2(msg)
        save_image(cv2_img)
        hsv_img = convert_to_hsv(cv2_img)
        save_image(hsv_img)
        lower_red= np.array([H_min_R,S_min,V_min])
        upper_red= np.array([H_max_R,S_max,V_max])
        lower_blue = np.array([H_min_B,S_min,V_min])
        upper_blue = np.array([H_max_B,S_max,V_max])
        lower_green = np.array([H_min_G,S_min,V_min])
        upper_green = np.array([H_max_G,S_max,V_max])
        objects_detected={"red":0,"green":0,"blue":0}
        detected_color=detect_objects(cv2_img,low_red,up_red,"red")
        objects_detected["red"]=detected_color
        detected_color=detect_objects(cv2_img,low_green,up_green,
                                       "green")
        objects_detected["green"]=detected_color
        detected_color=detect_objects(cv2_img,low_blue,up_blue,"blue")
        )
        objects_detected["blue"]=detected_color
        start_planning()
        set_orientation_constraints("l_wrist_roll_link"
                                    ,1.0,0.5,0.5,0.5)
        for x in objects_detected:
            if objects_detected[x]==1:
                move_to_goal(x)
                rospy.spin()
def main():
    rospy.init_node('image_listener')
    x = 0.7
    y = -0.7
    z = 0.4
    look_at_object(x,y,z)
    count=0
    image_topic = "/head_mount_kinect2/rgb/image_raw"
    rospy.Subscriber(image_topic, Image, image_callback)
    rospy.spin()

```

Capítulo 5

Solución de referencia

5.1. Procesamiento de imagen

Para generar una posición de destino correcta, es necesario analizar la imagen proporcionada por la cámara adherida al robot e identificar el objeto en la zona de inicio, si es que lo hay. A continuación profundizaremos en los elementos que hacen posible este proceso y como se implementa cada uno de ellos en nuestra práctica de planificación de trayectorias.

5.1.1. Cámara Kinect y ROS topics

La cámara Kinect se incorpora en la cabeza del robot, y utiliza el plugin openni_Kinect [4]. Éste permite obtener imágenes con resolución 680x480 y formato BGR, con 8 bits para cada uno de los colores primarios de la luz y por tanto un total de 24 bits por píxel. Además, al tratarse de una de las comúnmente llamadas cámara RGB-D, ofrece imágenes de profundidad que permitirían detectar lo cerca o lejos que se encuentra un objeto de la cámara. El plugin está configurado para refrescar las imágenes cada segundo, de forma que se puedan detectar de forma dinámica cambios en el entorno. En nuestro caso, necesitaremos controlar esto para recibir una única imagen y que ello suceda cuando el robot se encuentre mirando a la zona destino, de forma que no se tengan problemas innecesarios de procesamiento debido a código ineficiente.

Para obtener la información de imagen desde nuestra aplicación, utilizamos ROS Topics. El image_callback será el encargado de detectar que hay un nuevo mensaje de ROS de tipo imagen y realizar la conversión a OpenCV, que permitirá hacer más sencillas las maniobras relacionadas con el tratamiento de la imagen.

5.1.2. Librería OpenCV

El sensor Kinect situado sobre la cabeza del robot ofrece imágenes RGB-D. Utilizando la librería open source OpenCV, es posible acceder vía código Python a un enorme conjunto de posibilidades relacionadas con el procesamiento de imagen. [17]

En nuestro caso, además necesitaremos el llamado cv_bridge para transformar las imágenes del topic a cv2, que en código Python se puede realizar de la siguiente manera:

Algoritmo 4 Extracto del fichero `kinect2.gazebo.xacro`, donde se define el plugin para la cámara

```

<xacro:macro name="kinect2_rgb_gazebo_v0" params="link_name
    frame_name camera_name">
  <gazebo reference="${link_name}">
    <sensor type="depth" name="${name}_rgb_sensor">
      <always_on>true</always_on>
      <update_rate>1.0</update_rate>
      <camera>
        <horizontal_fov>${57.0*M_PI/180.0}</horizontal_fov>
        <image>
          <format>B8G8R8</format>
          <width>640</width>
          <height>480</height>
        </image>
        <clip>
          <near>0.01</near>
          <far>5</far>
        </clip>
      </camera>
      <plugin name="${link_name}_controller" filename="
          libgazebo_ros_openni_kinect.so">
        <alwaysOn>true</alwaysOn>
        <updateRate>1.0</updateRate>
        <cameraName>${camera_name}_rgb</cameraName>
        <imageTopicName>/${camera_name}/rgb/image_raw</
          imageTopicName>
        <cameraInfoTopicName>/${camera_name}/rgb/camera_info</
          cameraInfoTopicName>
        <depthImageTopicName>/${camera_name}/depth/image_raw</
          depthImageTopicName>
        <depthImageCameraInfoTopicName>/${camera_name}/depth/
          camera_info</depthImageCameraInfoTopicName>
        <pointCloudTopicName>/${camera_name}/depth_registered/
          points</pointCloudTopicName>
        <frameName>${frame_name}</frameName>
        <pointCloudCutoff>0.5</pointCloudCutoff>
      </plugin>
    </sensor>
    <material value="Gazebo/Red" />
  </gazebo>
</xacro:macro>
```

```
cv2_img = bridge.imgmsg_to_cv2(msg, "bgr8")
```

Una vez tenemos nuestra imagen lista para trabajar con OpenCV, se realiza una conversión

al espacio de color HSV, utilizando la función COLOR_BGR2HSV. Este espacio de color se caracteriza por aportar información sobre matiz de color, algo clave para diferenciar de una manera más precisa los diferentes colores, abstrandéndonos de otros parámetros menos intuitivos y determinando correctamente el color del objeto observado por el robot.

```
color_HSV = cv2.cvtColor(cv2_img, cv2.COLOR_BGR2HSV)
```

Como el objetivo final relativo al tratamiento de imágenes es poder diferenciar entre los colores rojo, verde y azul, son necesarias tres máscaras de color que permitan realizar un filtrado de la información de color según su matiz, saturación y brillo, y caracterizar así lo que hay frente a la cámara. Una vez definidos los umbrales mínimos y máximos para cada uno de los colores a partir de valores del espacio HSV, buscaremos objetos a partir de los contornos encontrados y su área. Si la imagen detecta algún conjunto de puntos conectados de un determinado color, y además esos puntos tienen un área considerable, el robot detecta ese objeto. De esta manera evitamos falsos positivos debidos a ruido en la imagen, consiguiendo un comportamiento robusto en un entorno controlado.

```
lower_blue = np.array([110,50,50])
upper_blue = np.array([130,255,255])
blueMask = cv2.inRange(color_HSV, lower_blue,
                       upper_blue)
cv2.imwrite('camera_image_hsv_blueMask.jpeg', blueMask)
resBlue = cv2.bitwise_and(cv2_img, cv2_img, mask=
                           blueMask)
cv2.imwrite('blueResult.jpeg', resBlue)
im2, contoursB, hierarchy = cv2.findContours(blueMask,
                                              cv2.RETR_TREE, cv2.CHAIN_APPROX_NONE)
contoursB = sorted(contoursB, key = cv2.contourArea,
                    reverse = True)[:1]
```

OpenCV ofrece otras funcionalidades como la posibilidad de visualizar o descargar las imágenes de una ejecución, algo de gran utilidad para el programador en la etapa de depuración. Dependiendo del objeto origen detectado, se llamará con los parámetros necesarios a las funciones encargadas de la planificación y ejecución de movimientos.

5.2. Planificación de trayectorias con MoveIt

Una vez identificado el color del objeto de la zona origen, el robot deberá utilizar su brazo izquierdo para superar obstáculos y llegar a la posición destino predefinida para dicho color. Una vez allí, realizará un movimiento de derribo.

La función move_to_goal es la encargada de realizar la planificación, tomando como parámetro el color detectado. Hace uso de la clase MoveItCommander, definida en el fichero move_group.py.

La trayectoria final se calcula utilizando mensajes de tipo Pose, donde se define la posición destino del brazo y la orientación del mismo en los distintos ejes. Este mensaje de

MoveIt! es suficiente para lanzar la planificación y posterior ejecución del movimiento de la siguiente manera:

```
pose_target = geometry_msgs.msg.Pose()
pose_target.orientation.w = 1.0
pose_target.position.x = 0.34
if color=='red':
    pose_target.position.y = y_red
elif color =='blue':
    pose_target.position.y = y_blue
elif color =='green':
    pose_target.position.y = y_green
else:
    print("No destination found")
pose_target.position.z = 0.8
group_left.set_pose_target(pose_target)
plan1 = group_left.plan()
group_left.execute(plan1)
```

La planificación de trayectorias requiere un fichero de lanzamiento y una serie de controladores para cada una de las articulaciones del brazo. En el fichero de lanzamiento se debe especificar la localización de los paquetes relacionados con planificación, así como los parámetros configurables asociados a cada uno de ellos.

5.2.1. MoveGroupCommander

MoveGroupCommander es una librería externa que nos permite acceder a las principales funcionalidades de MoveIt! a través de Python. A continuación enumeraremos algunas de las más relevantes, bien por su uso en la práctica, o por su utilidad en proyectos relacionados.

- get_name
- get_active_joints
- get_joints
- get_variable_count
- has_end_effector_link
- get_end_effector_link
- set_end_effector_link
- get_pose_reference_frame
- set_pose_reference_frame

Algoritmo 5 Fichero pr2_planning_execution.launch. Es el encargado de inicializar todo lo necesario para poder planificar trayectorias en MoveIt! con el robot PR2.

```
<launch>
<rosparam command="load" file="$(find_pr2_moveit_config)/config
/joint_names.yaml"/>
<include file="$(find_pr2_moveit_config)/launch/
planning_context.launch" >
<arg name="load_robot_description" value="true" />
</include>

<node name="joint_state_publisher" pkg="joint_state_publisher"
type="joint_state_publisher">
<param name="/use_gui" value="false"/>
<rosparam param="/source_list">[/joint_states]</rosparam>
</node>

<include file="$(find_pr2_moveit_config)/launch/move_group.
launch">
<arg name="publish_monitored_planning_scene" value="true" />
</include>

<param name="trajectory_execution/execution_duration_monitoring
" value="false" />
<param name="allowed_execution_duration_scaling" value="3.0"/>
</launch>
```

- get_planning_frame
- get_current_joint_values
- get_current_pose
- get_current_rpy
- get_random_joint_values
- get_random_pose
- set_start_state_to_current_state
- set_start_state
- get_joint_value_target
- set_joint_value_target
- set_rpy_target
- set_orientation_target
- set_position_target

- set_pose_target
- set_pose_targets
- clear_pose_target
- clear_pose_targets
- set_random_target
- remember_joint_values
- get_remembered_joint_values
- forget_joint_values
- get_goal_tolerance
- get_goal_joint_tolerance
- get_goal_position_tolerance
- get_goal_orientation_tolerance
- set_goal_tolerance
- set_goal_joint_tolerance
- set_goal_position_tolerance
- set_goal_orientation_tolerance
- get_known_constraints
- get_path_constraints
- set_path_constraints
- set_constraints_database
- set_planning_time
- get_planning_time
- set_num_planning_attempts
- set_max_velocity_scaling_factor
- set_max_acceleration_scaling_factor
- plan
- execute
- attach_object
- detach_object

- pick
- place
- set_support_surface_name

5.2.2. Controladores

Para que la planificación y ejecución de los movimientos del robot sean adecuadas, necesitamos unos controladores bien configurados para cada una de las articulaciones del brazo [4]. Son los encargados de procesar la información recibida por los distintos sensores, haciendo las funciones de cerebro del robot y controlando sus mecanismos. Para ello se especifican los nombres de cada una de ellas de la misma manera que se configuraron al crear el paquete de MoveIt! del robot PR2, cuyos pasos son intuitivos si seguimos los pasos del MoveIt! Setup Assistant desde los tutoriales oficiales. [14]

Algoritmo 6 Fichero controllers.yaml, dónde se definen los controladores que permiten mover cada una de las articulaciones necesarias.

```
controller_manager_ns: pr2_controller_manager
controller_list:
- name: r_arm_controller
  action_ns: follow_joint_trajectory
  type: FollowJointTrajectory
  default: true
  joints:
    - r_shoulder_pan_joint
    - r_shoulder_lift_joint
    - r_upper_arm_roll_joint
    - r_elbow_flex_joint
    - r_forearm_roll_joint
    - r_wrist_flex_joint
    - r_wrist_roll_joint
- name: l_arm_controller
  action_ns: follow_joint_trajectory
  type: FollowJointTrajectory
  default: true
  joints:
    - l_shoulder_pan_joint
    - l_shoulder_lift_joint
    - l_upper_arm_roll_joint
    - l_elbow_flex_joint
    - l_forearm_roll_joint
    - l_wrist_flex_joint
    - l_wrist_roll_joint
```

5.2.3. Planificador y configuración

Las posibilidades de configuración que ofrece MoveIt! son ingentes. Entre ellas se encuentran las posibilidades de seleccionar distintos planificadores. Podemos encontrar desde Open Motion Planning Library(OMPL), que está completamente soportada e integrada, hasta otras como STOMP, SBL o CHOMP que pese a ofrecer sus ventajas propias en determinados contextos de planificación, no están completamente integradas en la plataforma y dificultan el trabajo de programación. OMPL es la librería open source utilizada en la práctica, ya que está completamente soportada y los algoritmos de planificación que ofrece son con los que trabaja MoveIt! por defecto. [13] El objetivo de cualquier planificación en robótica consiste en resolver el problema de ir de un lado a otro sin violar las restricciones establecidas. OMPL es un sistema basado en muestreo, que consiste en generar aleatoriamente una serie de posiciones válidas en el espacio en función de los parámetros de entrada y conectarlas entre sí.

La librería contiene diferentes planificadores que se diferencian en la manera en la que generan las muestras, y ofrecen, generalmente, soluciones más rápidas que otros planificadores con métodos deterministas a la hora de calcular las trayectorias, generalmente cuando se requieren resultados precisos basados en la optimización. Dentro de los planificadores que ofrece OMPL, se utiliza una variante de algoritmo RRT(Rapidly-exploring Random Trees), que como su propio nombre indica, ofrece soluciones rápidas no deterministas, lo que aplicado a un brazo robótico se podría considerar como una analogía de la intuición humana.

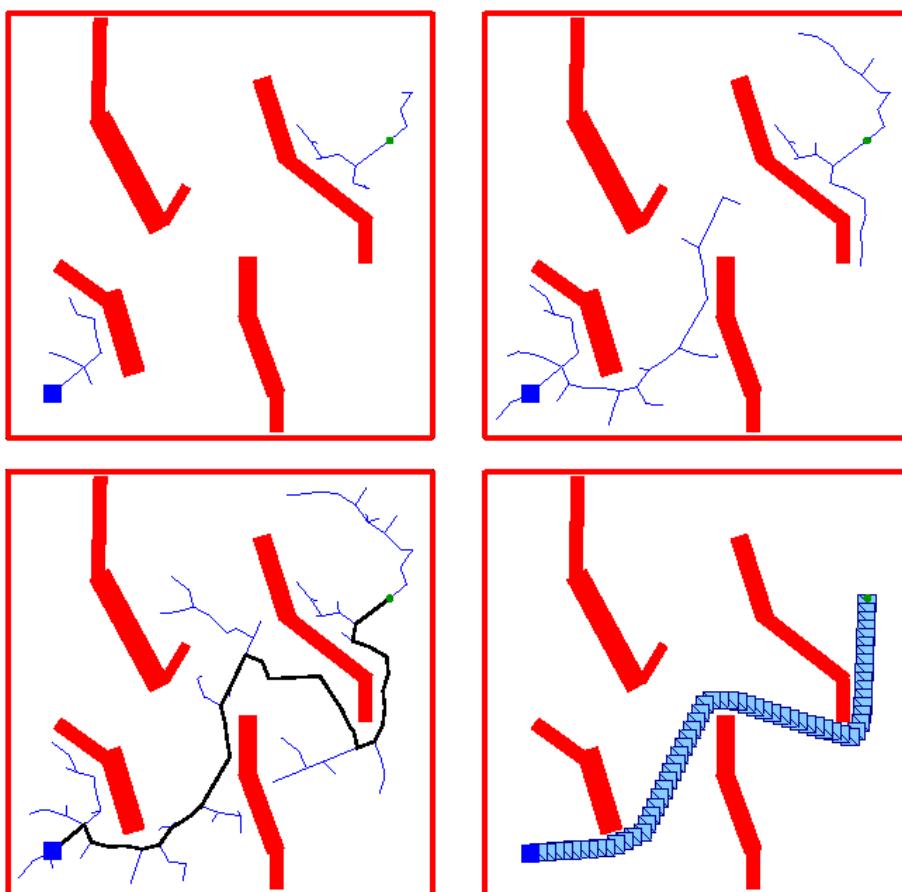


Figura 17: Explicación gráfica del algoritmo RRT Connect utilizado.

Aplicado a nuestra práctica, el algoritmo consiste en comenzar a generar de forma simultánea un ”árbol” desde el inicio y el destino del movimiento. El árbol inicio se irá expandiendo por posiciones cercanas aleatorias que cumplen las distintas restricciones programadas, hasta que toque al árbol destino, que sigue el mismo patrón de comportamiento, en un determinado punto. En ese momento, se habrá encontrado una solución para la planificación, es decir, un camino válido entre la posición inicial y la final que debe poder ser ejecutado sin impedimentos.

5.3. Experimentación

Como se ha deslizado en los anteriores capítulos, la ejecución típica [26] [27] de la práctica requiere el lanzamiento de varios ficheros. En primer lugar, se debe ejecutar un comando *source* que permita utilizar los paquetes personalizados por el desarrollador en su espacio de trabajo. Dicho comando podría añadirse al fichero *.bashrc* para que se detecten los paquetes dentro del directorio creado cada vez que se arranque un terminal en Linux. El fragmento a escribir en línea de comandos sería:

```
source ~/pr2_gazebo_tfg-devel/setup.bash
```

A continuación se pueden ejecutar, en distintos terminales, los ficheros que cargarán el entorno Gazebo y el entorno de planificación. Posteriormente, se ejecuta la plantilla *template.py* con la solución al ejercicio. Para ello, se requieren los siguientes comandos:

- Lanzar el mundo gazebo, que incluye el robot PR2 y sus modelos.

```
roslaunch pr2_gazebo tfg.launch
```

- Lanzar MoveIt! y las configuraciones de planificación necesarias para el robot PR2.

```
roslaunch pr2_moveit_config pr2_planning_execution_tfg.launch
```

- Lanzar el script *template.py*.

```
rosrun moveit_tutorials template.py
```

Una vez realizados los pasos anteriores, podremos observar en el simulador Gazebo la ejecución típica de la práctica de planificación y ejecución de trayectorias con MoveIt!, así como las imágenes del proceso descargadas en la máquina local.

En primer lugar cuando el robot observa la zona de muestra, obtiene una imagen RGB que debe transformar al espacio HSV para aplicar las máscaras de color [Figura 18] para el rojo, el verde y el azul. El funcionamiento del detector de objetos que ofrece la API de manejo de imágenes, permite detectar únicamente el color válido y descartar aquellos en los que se observa algo de ruido al aplicar la máscara de color, ya que de otra manera el robot detectaría objetos que en realidad no son válidos.

Una vez procesadas las imágenes y detectado el color del objeto en la zona de muestra, comenzará la planificación. Se realizarán entonces los dos trayectos[Figura 19] en que se

5.3. EXPERIMENTACIÓN

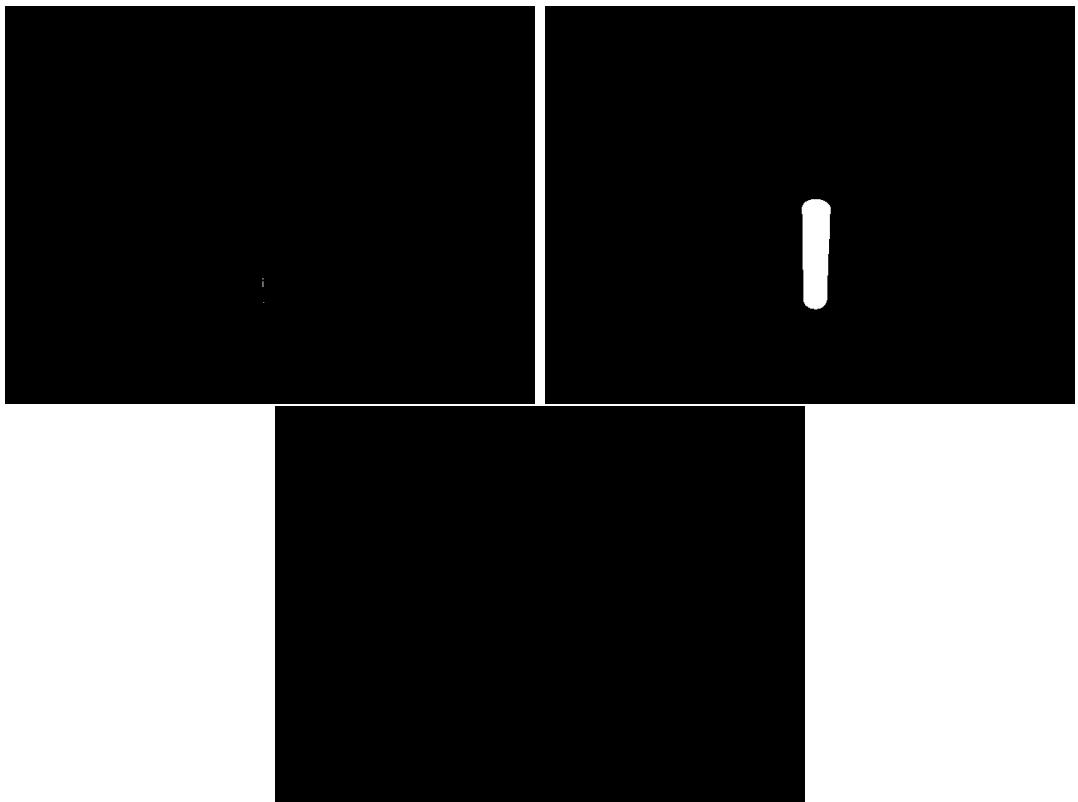


Figura 18: Resultado tras aplicar las máscaras de color para el rojo, azul y verde, respectivamente.

divide el movimiento. El primero de ellos consiste en esquivar los obstáculos para situar el brazo izquierdo frente al objeto a derribar, para después realizar un movimiento de derribo sobre el objeto cuyo color es igual al de la zona de muestra, haciendo caer el objeto correspondiente y finalizando así la ejecución de la práctica.

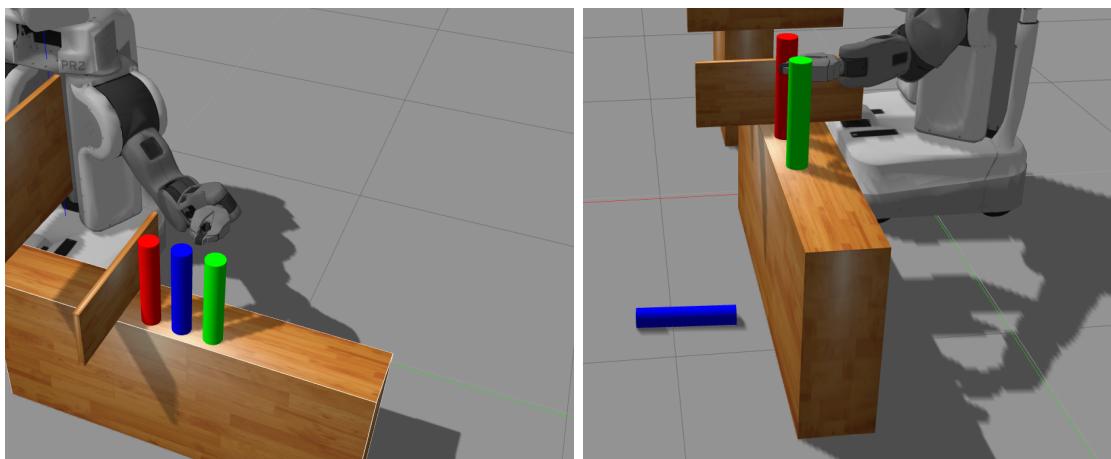


Figura 19: Posiciones del brazo robótico tras finalizar la ejecución de cada una de las dos trayectorias en que se divide su movimiento

5.3.1. Ajuste del entorno

Durante el desarrollo de la práctica ha sido necesario realizar pruebas con diferentes distancias entre el robot, los obstáculos, los objetos y las zonas. Todo ello en búsqueda de los límites del robot, una complejidad adecuada para la ruta de planificación entre los obstáculos del mundo [Figura 20] o la distancia idónea para que el apartado visual y estético.

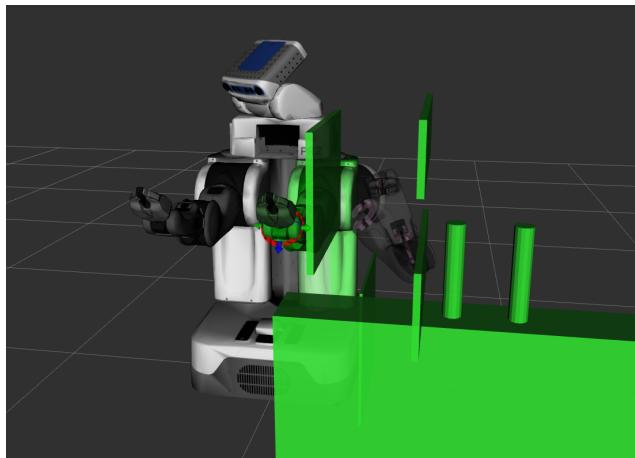


Figura 20: Situación creada dinámicamente en RVIZ durante las pruebas con obstáculos

Para llevar a cabo este ensayo se ha utilizado el visualizador RVIZ, una herramienta de MoveIt! que permite configurar los elementos del entorno sin necesidad de utilizar el simulador Gazebo, haciendo posible validar planificaciones secuencialmente sin reiniciar ningún proceso. Entre otras tareas permite planificar movimientos con las diferentes articulaciones del robot, visualizar información de los sensores a través de un determinado topic, añadir objetos ficticios, realizar pruebas con diferentes configuraciones del planificador u obtener coordenadas en el espacio.

Se ha generado un fichero llamado `textitpr2_planning_execution_tfg_rviz.launch` que permite lanzar el entorno de planificación incluyendo el visualizador, de manera que el desarrollador pueda probar nuevos escenarios o depurar sus cambios de forma sencilla. Para ello, es necesario añadir algunas líneas al fichero `.launch` [**planning_rviz**].

Algoritmo 7 Líneas a añadir en el fichero de lanzamiento del entorno de planificación para incluir RVIZ.

```
<include file="$(find_pr2_moveit_config)/launch/moveit_rviz.launch">
<arg name="config" value="true"/>
</include>
```

5.3.2. Ajuste del detector de colores

Con el objetivo de obtener los resultados esperados tras el procesamiento de las imágenes con la librería *OpenCV*, ha sido necesario realizar varios ajustes destacados.

5.3. EXPERIMENTACIÓN

El primero de ellos está relacionado con definir umbrales válidos para cada uno de los colores en el espacio HSV, que permitan identificarlos minimizando el margen de error. Para ello se ha utilizado como referencia bibliografía [3] [6] sobre dicho espacio de color.

Tras aplicar las máscaras, se obtiene una imagen en blanco y negro donde el negro representa los píxeles que no están entre los máximos y mínimos definidos, y el blanco los que sí. En este punto, es necesario un procedimiento para diferenciar entre objetos reales y píxeles independientes que han pasado el filtro pero no deberían ser considerados un objeto. Para ello se utilizan las funciones de *OpenCV* *findContours* y *contourArea*, como se comentó previamente. La primera obtiene los grupos de píxeles contiguos existentes tras aplicar la máscara, y la segunda es capaz de calcular el área de cada uno de ellos. La solución escogida consiste en obtener el área más grande y compararlo con un valor por defecto que garantice resultados correctos en el contexto del ejercicio. Tras experimentar con diferentes valores, se ha establecido en 500 píxeles.

5.3.3. Ajuste del planificador

La primera de las trayectorias ejecutadas por el brazo robótico del PR2 es la de mayor complejidad, trabajando con la planificación de trayectorias para esquivar varios obstáculos. En este punto ha sido necesario ajustar el parámetro del planificador OMPL llamado *longest_valid_segment_fraction*, que define la distancia a la que se considera asumible viajar por el espacio en búsqueda del siguiente nodo del grafo sin validar el estado de colisiones. Es decir, valores altos de este parámetro serán más rápidos pero podrían provocar errores en zonas estrechas o esquinas, y valores bajos garantizarán que el robot no se choque en su trayectoria a cambio de más lentitud.

Entender como se calculan las posibles trayectorias en OMPL implica que el valor predefinido por MoveIt! para este parámetro, que es configurable en el fichero *ompl_planning.yaml* e igual a 0.05 por defecto, no tiene por qué ser el idóneo. Para adaptarlo a las necesidades de esta práctica ha sido necesario modificarlo, tras el análisis realizado simulando 30 planificaciones para diferentes combinaciones según el número de obstáculos y la configuración de *longest_valid_segment_fraction*. Como se puede observar en la [Figura 21],

Obstáculos	longest_valid_segment_fraction	Tiempo medio de ejecución	Tasa de error	%Warnings
0	0,05	0,269	0,47	0,00%
0	0,005	0,575	0,00	0,00%
0	0,001	1,276	0,00	0,00%
1	0,05	0,470	0,53	42,86%
1	0,005	2,960	0,07	0,00%
1	0,001	6,142	0,00	0,00%
2	0,05	0,395	0,67	40,00%
2	0,005	3,531	0,07	28,57%
2	0,001	9,446	0,00	0,00%

Figura 21: Respuesta del sistema ante distintas situaciones de entorno y diferentes configuraciones del planificador.

cuanto más complejo es el escenario, mayor precisión es necesaria para no obtener errores o *warnings* en la planificación que pueden provocar problemas al ejecutar la ruta. En base a los resultados obtenidos, se ha decidido utilizar un valor de 0.001, ya que es el que garantiza un resultado fiable en el escenario final con 2 obstáculos. Por motivos de

fiabilidad y optimización, el algoritmo calculará dos trayectorias y elegirá el mejor resultado, como se ha configurado a través del método *set_num_planning_attempts* que ofrece *MoveGroupCommander*.

Capítulo 6

Conclusiones

El capítulo final está enfocado a transmitir cuáles han sido los objetivos cumplidos satisfactoriamente, qué ha aportado el camino realizado para lograrlos, y qué líneas de trabajo se podrían seguir para mejorar o aprovechar los resultados del Trabajo de Fin de Grado.

6.1. Conclusiones

El objetivo formal propuesto al comienzo del desarrollo del proyecto ha sido alcanzado con éxito, generando una práctica completa de planificación de trayectorias utilizando herramientas modernas y extendidas en el mundo robótico. El desarrollo de esta práctica ha supuesto importantes retos que enfrentar. En primer lugar, enfrentarse a un área de conocimiento nueva hasta ser capaz de aprovechar las posibilidades que ofrece y completar una solución técnica. Por el camino, lo más importante, ha sido necesario aprender cada semana para resolver los problemas del día a día hasta alcanzar los objetivos planteados, mejorando así una de las facetas clave de la ingeniería y adquiriendo conocimientos nuevos.

Dicho crecimiento ha surgido a partir de las bases obtenidas durante el grado, tanto teóricas como prácticas y soft skills adquiridas durante la carrera. Para poder comenzar a trabajar con garantías, han sido necesarios los conocimientos y experiencia adquiridos en las distintas asignaturas relacionadas con la programación y el software, así como el tratamiento de imágenes o los fundamentos físicos. Durante el proceso, han ido acompañadas de otras habilidades como el rendimiento bajo presión, la comunicación, la adaptación al cambio, la motivación o el pensamiento crítico. La metodología de trabajo escogida ha permitido recibir feedback de forma dinámica sobre el progreso realizado, para conseguir adaptar el proyecto a las realidades que se enfrentan y afrontar las dificultades con garantías de éxito. La práctica de planificación de trayectorias, tuvo como primer reto la toma de contacto con la robótica a través de algunas de las prácticas más sencillas de JdeRobot. Enseguida se consideró necesario el aprendizaje de un nuevo lenguaje de programación(C++), tanto para poder comprender el código ya existente en los repositorios en este lenguaje, como para adaptar algunos tutoriales que sólo estaban disponibles en C++ al lenguaje elegido, Python. Además se realizaron algunos tutoriales sobre ICE, el protocolo de comunicaciones utilizando en las prácticas comentadas. El siguiente paso consistió en descubrir el simulador Gazebo, que posteriormente tendría un papel fundamental para la solución final. Una vez resueltos estos primeros pasos, se comenzó a trabajar con ROS, acercándose a él a través de los tutoriales oficiales. Con los conocimientos sobre

nodos, topics o creación de espacios de trabajo, fue suficiente para afrontar el núcleo de este proyecto: MoveIt!.

Para comenzar se trabajó con el proceso de configuración de un robot, en este caso el modelo PR2, además de utilizar como apoyo el visualizador RVIZ y viendo efectos prácticos desde el primer momento.

Ya sentadas las bases del proyecto, fue posible considerar toda la solución en su conjunto y centrar los esfuerzos en profundizar en la planificación de trayectorias y en otras herramientas necesarias como la librería OpenCV. La última parte del trabajo consistió en refinar detalles de los diferentes componentes de la práctica. Por un lado se añadieron algunos obstáculos al mundo gazebo y se configuró un movimiento inicial en la cabeza del robot para mirar a la zona de origen, para terminar mejorando la mecánica de movimiento ajustando parámetros como restricciones en la articulación de la muñeca del robot, los tiempos e intentos de la planificación o el plugin que controla la cinemática del ejercicio.

Además del apartado puramente técnico, la motivación ha permitido superar el reto de completar un proyecto de ingeniería actual e innovador utilizando metodología de desarrollo de software extendida laboralmente. Ello ha permitido adquirir conocimientos y habilidades útiles para el futuro laboral, como pueden ser ampliar los conocimientos de programación y tratamiento de imágenes, conocer el mundo de la robótica, diseñar modelos y arquitecturas, optimizar costes o trabajar en una solución integrada por distintos componentes que se comunican entre sí.

6.2. Trabajos futuros

La solución aquí propuesta puede abrir la puerta a algunas líneas de trabajo futuras que, o bien supongan avances para la práctica realizada, o bien supongan nuevas prácticas que hagan uso de los conceptos y código desarrollados.

Un primer avance podría ser hacer uso de las imágenes de profundidad ya aportadas por la cámara Kinect para calcular la posición relativa de los objetos en la zona destino, y utilizar dichas posiciones para calcular el destino de la planificación. Esta funcionalidad podría completarse calculando la localización y tamaño de los obstáculos y objetos para tenerlos en cuenta al planificar.

Otra idea para profundizar en el uso de MoveIt!, sería añadir una tercera zona de acción en la que el robot deposite el objeto del color correspondiente, a través de maniobras de pick&place. Para ello sería necesario aprender a manejar los mensajes y dispositivos de agarre, para lo que los tutoriales oficiales de MoveIt! pueden ser de gran ayuda.

Por último, podría ser necesario adaptar el código a futuras versiones de ROS que introduzcan mejoras importantes relacionadas con planificación, movimientos del robot o nueva funcionalidad. De manera análoga, igual que el robot PR2 se escogió por motivos de diseño y compatibilidad, afrontar algunas mejoras podría requerir migrar la solución a un nuevo robot.

Si el trabajo futuro se centrase en el estudio teórico o la obtención de métricas relacionadas con la planificación, una opción sería profundizar en los algoritmos de planificación disponibles en MoveIt!, o incluso generar los propios para estudiar los casos de mayor rendimiento para cada uno de ellos.

Bibliografía

- [1] *Automation Forecast*. URL: <https://www.interquestgroup.com/insights/infographics/2018/robotic-process-automation-industry-forecast> (vid. pág. 3).
- [2] *CLARAty*. URL: <https://www-robotics.jpl.nasa.gov/facilities/facility.cfm?Facility=2> (vid. pág. 5).
- [3] *Colores HSV*. URL: <http://www.workwithcolor.com/red-color-hue-range-01.htm> (vid. pág. 42).
- [4] *Configure MoveIt! controllers*. URL: <http://www.theconstructsim.com/control-gazebo-simulated-robot-moveit-video-answer/> (vid. pág. 37).
- [5] *cvbridgePython*. URL: http://wiki.ros.org/cv_bridge/Tutorials/ConvertingBetweenROSImageAndOpenCV (vid. pág. 28).
- [6] *Espacios de color con Python y OpenCV*. URL: https://opencv-python-tutroals.readthedocs.io/en/latest/py_tutorials/py_imgproc/py_colorspaces/py_colorspaces.html (vid. pág. 42).
- [7] *Foro Económico Mundial 2016*. URL: <https://www.weforum.org/agenda/2016/01/the-fourth-industrial-revolution-what-it-means-and-how-to-respond/> (vid. pág. 2).
- [8] *Gazebo camera plugins*. URL: http://gazebosim.org/tutorials?tut=ros_gzplugins (vid. pág. 25).
- [9] *JdeRobot*. URL: https://jderobot.org/Main_Page (vid. pág. 7).
- [10] *Mckinsey: A future that works*. URL: <https://www.mckinsey.com/~/media/mckinsey/featured%5C%20insights/Digital%5C%20Disruption/Harnessing%5C%20automation%5C%20for%5C%20a%5C%20future%5C%20that%5C%20works/MGI-A-future-that-works-Executive-summary.ashx> (vid. pág. 3).
- [11] *Microsoft Robotics Developer Studio*. URL: <https://www.microsoft.com/en-us/download/details.aspx?id=29081> (vid. pág. 5).
- [12] *Minimizing code defects to improve software quality and lower development costs*. URL: <ftp://ftp.software.ibm.com/software/rational/info/do-more/RAW14109USEN.pdf> (vid. pág. 12).
- [13] *MoveIt! Planners*. URL: <https://moveit.ros.org/documentation/planners/> (vid. pág. 38).
- [14] *MoveIt! tutorials*. URL: http://docs.ros.org/kinetic/api/moveit_tutorials/html/index.html (vid. pág. 37).
- [15] *Open Robot Control Software*. URL: <http://www.orocos.org/> (vid. pág. 5).

BIBLIOGRAFÍA

- [16] *OpenCV: Image Transformations*. URL: https://docs.opencv.org/2.4/modules/imgproc/doc/miscellaneous_transformations.html (vid. pág. 28).
- [17] *OpenCV tutorial*. URL: https://docs.opencv.org/3.4.1/d9d/tutorial_py_colorspaces.html (vid. pág. 31).
- [18] *Primeros graduados con conocimientos en robótica*. URL: https://elpais.com/ccaa/2018/10/24/madrid/1540404720_746066.html (vid. pág. 6).
- [19] *Robotics Market*. URL: <https://www.mordorintelligence.com/industry-reports/robotics-market> (vid. pág. 3).
- [20] *ROS*. URL: <http://www.ros.org/> (vid. pág. 5).
- [21] *SDF Format*. URL: <http://sdformat.org/spec> (vid. pág. 26).
- [22] *Spiral Model*. URL: <http://www.dimap.ufrn.br/~jair/ES/artigos/SpiralModelBoehm.pdf> (vid. pág. 12).
- [23] *TFG Carlos Awadallah*. URL: https://gsyc.urjc.es/jmplaza/students/tfg-JdeRobot_Academy-carlos_awadallah-2018.pdf (vid. pág. 8).
- [24] *TFG Pablo Moreno*. URL: <https://github.com/RoboticsURJC-students/2017-tfg-pablo-moreno/blob/master/TFG/Memoria.pdf> (vid. pág. 8).
- [25] *TFG Vanessa Fernández*. URL: https://gsyc.urjc.es/jmplaza/students/tfg-JdeRobot_Academy-vanessa_fernandez-2017.pdf (vid. pág. 8).
- [26] *Vídeo final TFG 1*. URL: <https://youtu.be/OU1iNR0tzyA> (vid. pág. 39).
- [27] *Vídeo final TFG 2*. URL: https://youtu.be/Np2g_3XHoH4 (vid. pág. 39).
- [28] *Why Every Software Startup Should Have a Testing Process Through Launch*. URL: <https://www.inc.com/aj-agrawal/why-every-software-startup-should-have-a-testing-process-through-launch.html> (vid. pág. 12).
- [29] *Workspace overlaying in ROS*. URL: http://wiki.ros.org/catkin/Tutorials/workspace_overlaying (vid. pág. 26).
- [30] *wstool*. URL: <http://wiki.ros.org/wstool> (vid. pág. 26).