



Escuela Técnica Superior de Ingeniería de Telecomunicación

MOVIMIENTO PLANIFICADO 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

*“Ever tried. Ever failed.
No matter. Try again.
Fail again. Fail better.”*

Samuel Beckett

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	12
2.3 Metodología	12
2.4 Plan de acción	13
3 Infraestructura utilizada	15
3.1 Robot PR2	15
3.1.1 Cámara Microsoft Kinect	16
3.2 Simulador Gazebo	17
3.3 Middleware robótico ROS Kinetic	18
3.4 Lenguaje de programación C++	19
3.5 Lenguaje de programación Python	19
3.6 Biblioteca de visión OpenCV	20
3.7 Entorno de planificación de movimientos MoveIt!	20
4 Ejercicio de planificación de trayectorias con brazo robótico	23
4.1 Enunciado	23
4.2 Diseño software	24
4.3 Infraestructura del ejercicio	25
4.3.1 Mundo Gazebo	25
Modelo PR2 enriquecido	25
Modelos adicionales	27
4.4 Plantilla Python	29
5 Solución de referencia	33
5.1 Diseño de la solución	33
5.2 Movimiento inicial y procesamiento de imagen	34
5.2.1 Movimiento de cuello	34
5.2.2 ROS topics para acceder a las imágenes de Kinect	34

ÍNDICE GENERAL

5.2.3 Transformación a CVMat de OpenCV y al espacio HSV	34
5.2.4 Filtrado por colores	35
5.3 Movimiento de aproximación del brazo con MoveIt!	36
5.3.1 moveit_commander	38
5.3.2 Controladores	39
5.3.3 Planificador y configuración de MoveIt!	40
5.4 Maniobra de derribo	42
5.5 Validación experimental	43
5.6 Alternativas probadas	46
5.6.1 Ajuste del entorno mediante RViz	46
5.6.2 Ajuste del planificador de trayectorias	47
5.6.3 Otros planificadores	47
5.6.4 Restricciones de planificación	48
5.6.5 Pruebas con la mano robótica (<i>gripper</i>)	49
6 Conclusiones	51
6.1 Conclusiones	51
6.2 Trabajos futuros	52
Bibliografía	55

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 (ver 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.

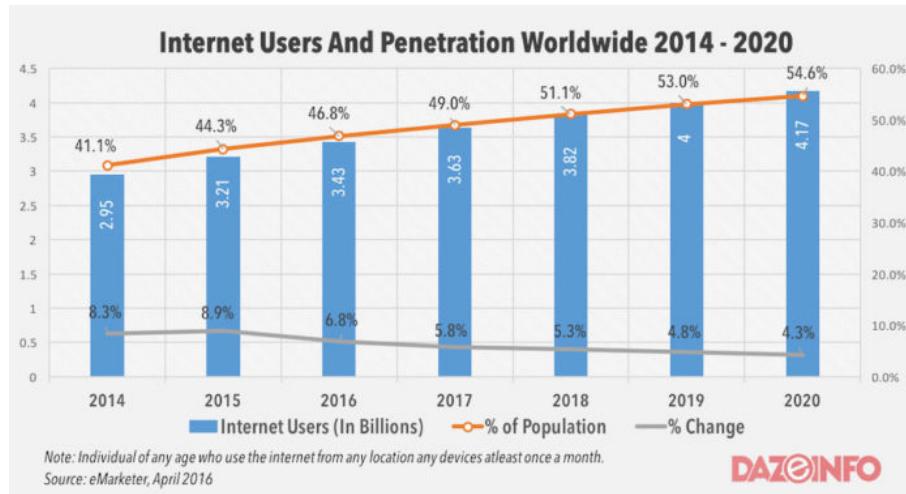


Figura 1: Predicción del número de usuarios en Internet 2016-2020

En el Foro Económico Mundial de 2016 [10] 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 [30]. 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 [2] [14] 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 (ver Figura 2) 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.

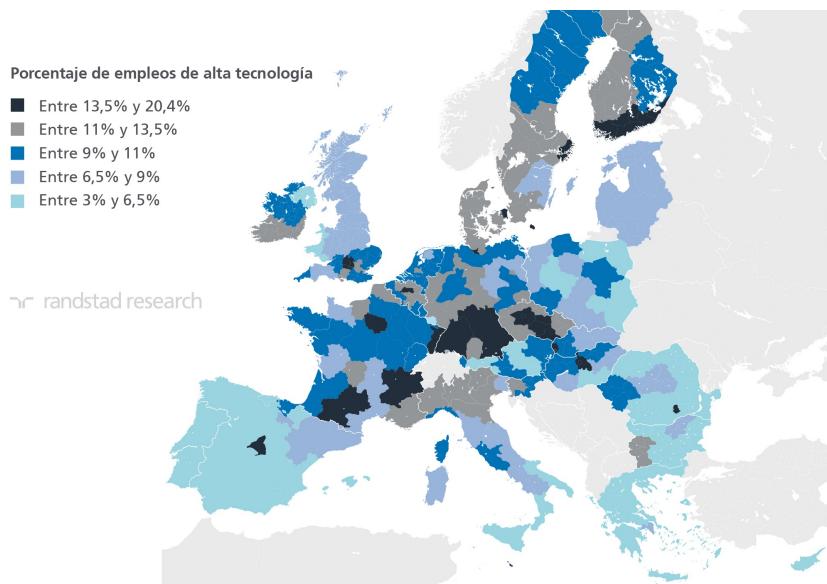


Figura 2: Empleos de alta tecnología en Europa

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 (ver 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:

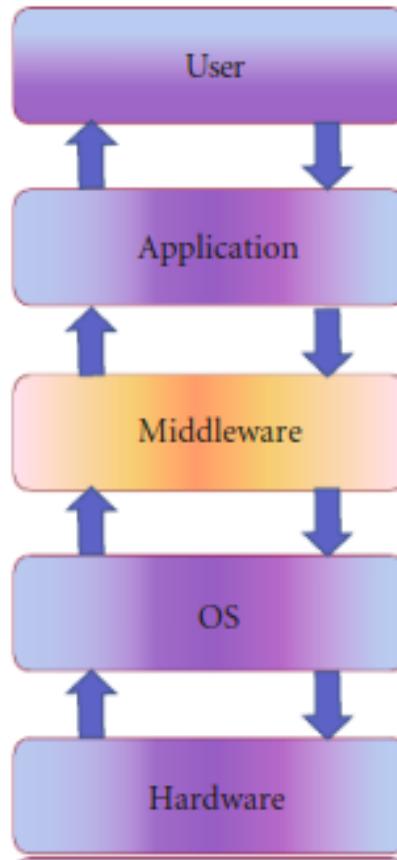


Figura 3: Situación del *middleware* en una arquitectura software

- ROS [31]: 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 [21]: 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 [5]: Promueve la reutilización de software, además de la portabilidad, modularidad y flexibilidad. Surge de la colaboración entre varias instituciones, entre ellas la NASA.
- MRDS [15]: 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? [25]. 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 (ver Figura 4).
- 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 (ver Figura 5). 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 (ver Figura 4), 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 [13], 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 abstrae 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

1.3. DOCENCIA EN ROBÓTICA

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 (ver Figura 6). 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 [4].

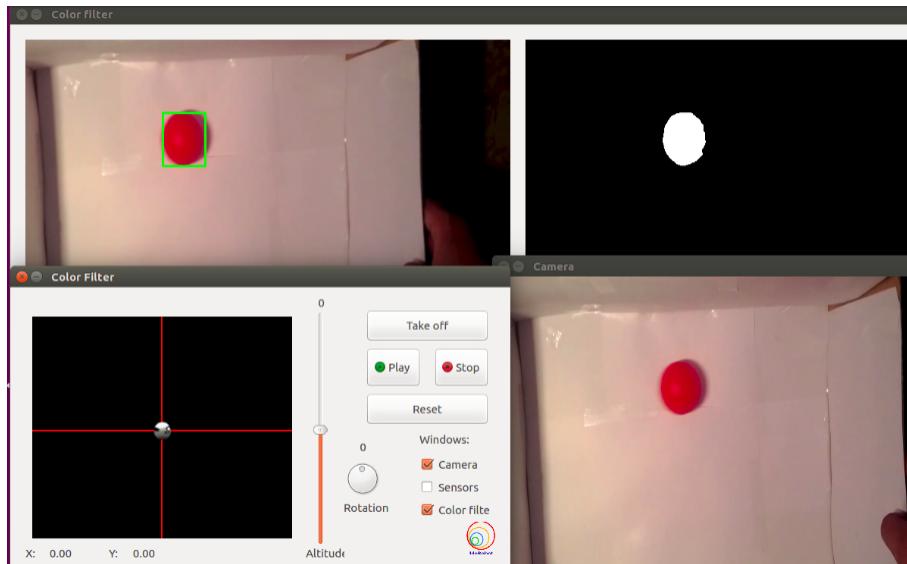


Figura 6: Filtro de color de la práctica color filter

En la práctica Drone cat&mouse (ver Figura 7) 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 (ver Figura 8) 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 [35] permite familiarizarse con algoritmos de navegación.

También hay ejemplos relacionados con los coches autónomos, como Visual follow-line behavior on a Formula1 (ver Figura 9) 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].

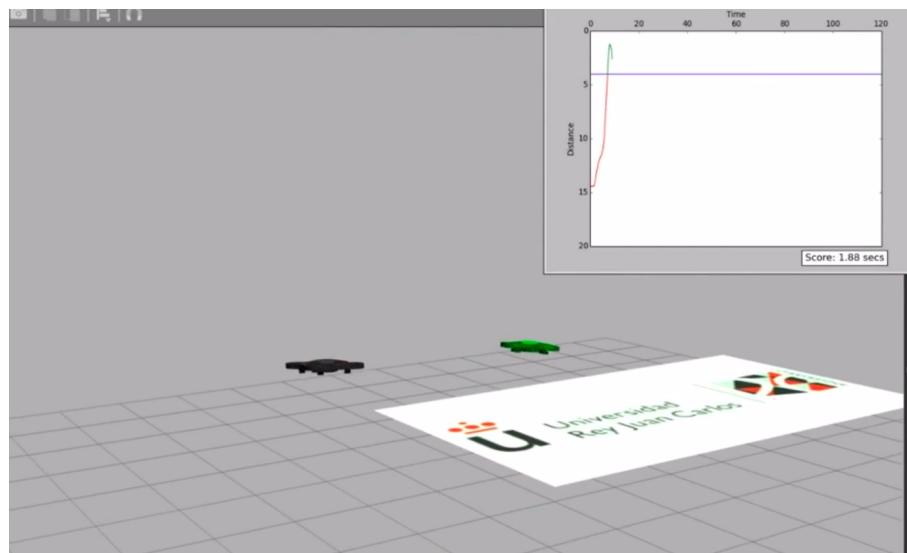


Figura 7: Persecución de drones en cat&mouse

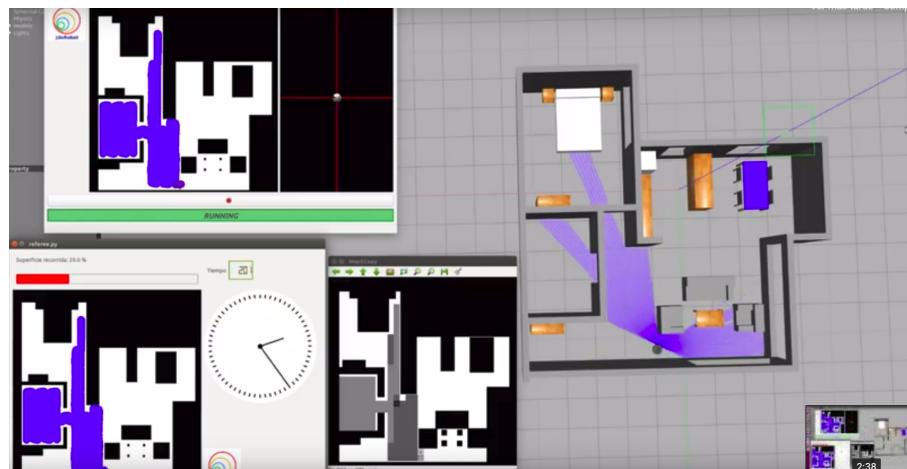


Figura 8: Mapeo generado en tiempo real en la práctica de la aspiradora

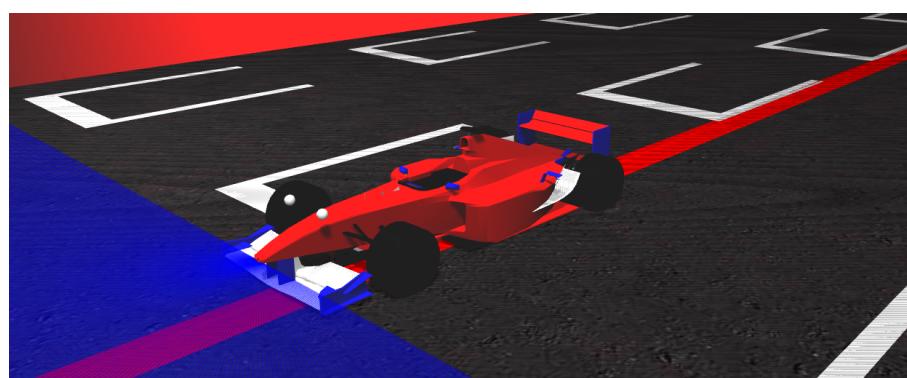


Figura 9: Mundo Gazebo para la práctica de conducción con Fórmula 1

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 y el plan de acción que se han seguido para alcanzarlos.

2.1. Objetivos

El objetivo principal de este trabajo consiste en añadir una nueva práctica al entorno educativo Robotics-Academy de la organización JdeRobot. Dicha práctica tiene como meta profundizar en la planificación de trayectorias de un brazo robótico. Para realizar dicha práctica se utilizará MoveIt!, el entorno de ROS 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.

Este objetivo general se ha articulado en tres subobjetivos:

1. Desarrollar la infraestructura necesaria, principalmente a través de un nuevo mundo generado en el simulador que incluirá los modelos para el robot y los distintos objetos.
2. Desarrollar una plantilla en Python que permita al receptor de la práctica tomar contacto con el tratamiento de imágenes y la planificación de trayectorias en MoveIt! de manera sencilla, abstrayéndole de los detalles de implementación más complejos, a los que podrá acceder posteriormente si desea profundizar en ellos.
3. Desarrollar la solución de referencia para lograr satisfacer el comportamiento esperado en esta práctica con un brazo robótico, incluyendo la planificación en entornos con obstáculos, la detección de objetos de color mediante procesado de imagen y la experimentación para optimizar los resultados

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.

2.2. Requisitos

A continuación, se enumeran las diferentes condiciones requeridas en el software y hardware para la consecución de los objetivos planteados:

- El software debe realizar acciones en tiempo real, con valores aceptables para los tiempos de ejecución en el simulador en un ordenador convencional.
- El software debe ser capaz de obtener trayectorias válidas en situaciones con varios obstáculos.
- Se utilizará el sistema operativo 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.

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 [34]. 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 (ver 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 [16] [36] 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

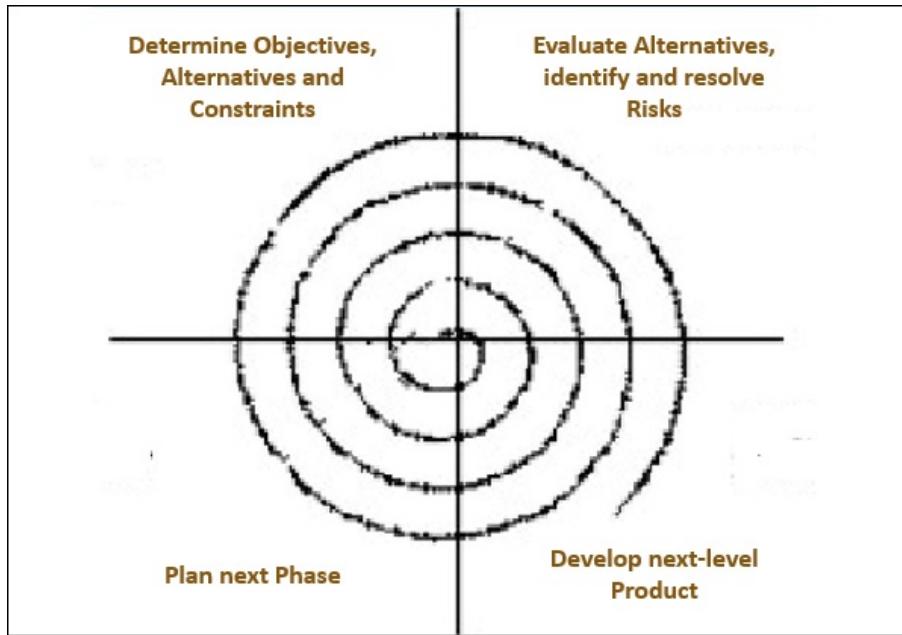


Figura 10: Representación gráfica del modelo en espiral para desarrollo software

posibles riesgos relativos al proyecto para no derivar en problemas en etapas avanzadas del desarrollo.

La aplicación de los procedimientos propuestos por Boehm ha supuesto la realización de reuniones semanales de revisión del estado del proyecto, dónde se definen tareas, se proponen objetivos y se recopilan los avances. En la etapa de desarrollo, se ha ido alimentando una bitácora [3] con los hitos alcanzados y los problemas surgidos, además del repositorio disponible en GitHub [27]. Por tanto, se han gestionado los cambios con el sistema de control de versiones Git, añadiéndolos a la rama *master* principal una vez validados.

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 alcanzarlos, 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.
- Familiarización con el entorno educativo Robotics-Academy 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 realizar la 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. Posteriormente, utilizar dicho conocimiento para crear el mundo simulado para este ejercicio.

2.4. PLAN DE ACCIÓN

- Descubrir y profundizar en el entorno 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.
- Utilizar los conocimientos adquiridos para ofrecer una plantilla de código Python que abstraiga al usuario final de la complejidad de MoveIt! y OpenCV, y permita completar una práctica de planificación y ejecución de trayectorias con un brazo robótico en un entorno simulado.
- Aplicar los conceptos previos para resolver los problemas necesarios y programar una solución de referencia para la práctica de robótica escogida.

Capítulo 3

Infraestructura utilizada

En este capítulo se detalla el software que se ha empleado para el desarrollo del TFG. Desde ROS Kinetic y JdeRobot como *middlewares* robóticos, pasando por la herramienta MoveIt! para facilitar el trabajo con robots hasta llegar al simulador Gazebo. Todo ello corriendo en el sistema operativo Ubuntu 16.04 y desarrollado en Python. Las comunicaciones entre los distintos componentes utilizan principalmente ROS topics.

3.1. Robot PR2

El robot (ver Figura 11) sobre el que se ha desarrollado el software es el modelo PR2 [29] de la empresa Willow Garage, especializada en robótica y responsable en su día también del software ROS y otras herramientas de código abierto. 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 de su lanzamiento.



Figura 11: Imagen de un robot PR2 real

3.1. ROBOT PR2

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. Su diseño modular (ver Figura 12) permite integrarlo con otros *grippers*, brazos o sensores.

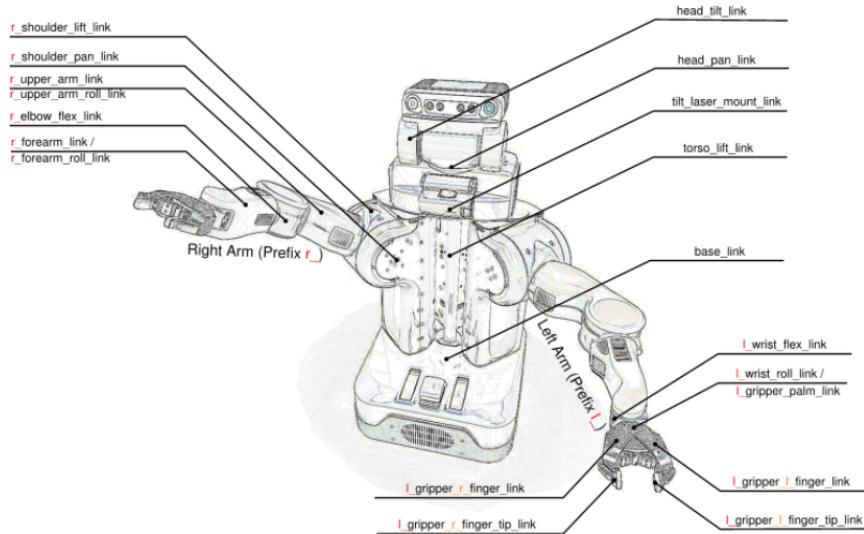


Figura 12: Estructura del robot PR2

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. Cámara Microsoft Kinect

La cámara Kinect (ver Figura 13), que ha sido incorporada al PR2 canónico para poder obtener imágenes desde la perspectiva del robot, 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.



Figura 13: Apariencia de una cámara Kinect de Microsoft

Kinect es propiedad de Microsoft, y fue lanzada originalmente en el año 2010 con el objetivo de ofrecer nuevas posibilidades para su consola Xbox, aunque fue perdiendo protagonismo con el paso de los años. Más tarde se comercializó también para ordenadores, dando el salto así a proyectos de índole científica que se sirven de su detección de movimiento y reconocimiento de voz.

3.2. Simulador 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 [11] 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 de referencia para esta práctica y los desarrollos sobre ROS en general, permite 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 realizar muchas modificaciones, disminuyendo tanto los costes económicos como los derivados de las pruebas en fase de desarrollo que pueden dañar el robot. Ofrece visualización 3D y un potente motor de físicas (ver Figura 14) para, en el caso de un brazo robótico, poder determinar parámetros como fricción, rango de movimiento o colisiones.

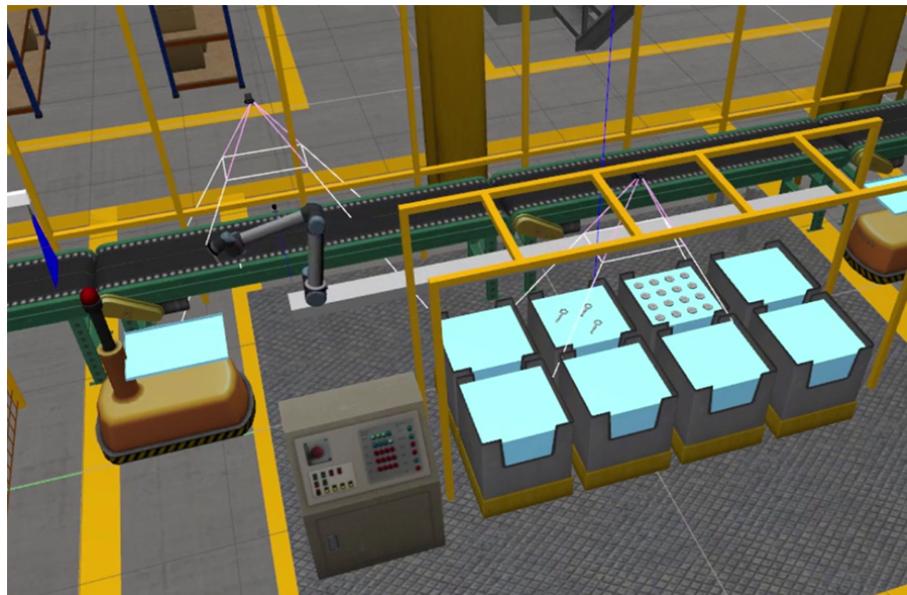


Figura 14: Mundo Gazebo que simula un entorno industrial

La simulación nace a partir de un fichero *.launch* que arranca Gazebo. El entorno de la simulación está compuesto por un mundo que contiene diferentes modelos. 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. Middleware robótico ROS Kinetic

ROS es un entorno para el desarrollo de software para robots. Nació en 2007 en el Laboratorio de Inteligencia Artificial de Standford. La arquitectura de grafos de ROS (Robot Operating System) se basa en nodos que se comunican a través de mensajes. La práctica con un brazo robótico desarrollada tendrá como eje un nodo ROS que se comunicará con los drivers del PR2 en Gazebo y con las librerías de manejo de imágenes y planificación.

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.

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, y pueden estar programados en distintos lenguajes como Python y C++. 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.

Un nodo ROS se puede lanzar automáticamente a través de ficheros *.launch*, que serán muy habituales en el desarrollo.

Para ejecutar un nodo, necesitaremos el paquete y el nombre del nodo. Una vez lanzado el nodo, podemos publicar *topics* de un tipo concreto que el nodo comprenda, consiguiendo así que realice una acción determinada. Podemos consultar en cualquier momento la lista de los nodos o *topics* disponibles, o ver los mensajes publicados en un determinado *topic*. (ver Algoritmo 1).

Las comunicaciones entre nodos son la base del desarrollo con robots en ROS, y son representables en forma de grafo a través de la herramienta *rqt* (ver Figura 15).



Figura 15: Representación gráfica utilizando *rqt* de un nodo teleoperador comunicándose con un nodo tortuga a través de un *topic*

Algoritmo 1 Comandos básicos para trabajar con ROS desde un terminal Linux

```
# Para lanzar un nodo turtlesim y publicar un mensaje en el
# topic /turtle1/cmd_vel
$ roscore
$ rosrun turtlesim turtlesim_node
$ rostopic pub -1 /turtle1/cmd_vel geometry_msgs/Twist -- '
  [2.0, 0.0, 0.0] ' '[0.0, 0.0, 1.8] '
# Para obtener los mensajes publicados en el topic:
$ rostopic echo /turtle1/cmd\_\_vel
# Para representar gráficamente las comunicaciones actuales:
$ rosrun rqt_graph rqt_graph
# Para obtener los nodos y topics disponibles:
$ rosnode list
$ rostopic list
```

3.4. Lenguaje de programación 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 y portátil. Se rige por un estándar de ISO (International Organization for Standardization) cuya última versión es C++17, lanzada en 2017.

Es lenguaje con un grado de complejidad elevado, que permite una gran versatilidad y potencia, pero también tiene una curva de aprendizaje elevada para dominarlo. Algunos IDEs recomendados para trabajar con C++ son VisualStudio y Code::Blocks. Como curiosidad, se trata del cuarto lenguaje de programación más popular hoy en día según el conocido ranking de TIOBE [26], sólo por detrás de Java, C y Python. Alcanzó su máximo de popularidad 2003 según los datos de esta empresa de software.

3.5. Lenguaje de programación 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 tercer lenguaje de programación en popularidad, alcanzando en 2019 su máximo nivel en la lista.

Se ha utilizado la versión 2.7.12 debido principalmente a su compatibilidad con ROS Kinetic. En este lenguaje se han desarrollado la plantilla y la solución de referencia para la aplicación de planificación de movimientos

3.6. Biblioteca de visión OpenCV

Es una biblioteca de código abierto enfocada a proyectos de visión artificial, con interfaces disponibles para varios lenguajes de programación (C++, Python, Java, MATLAB...). Open Source Computer Vision Library (OpenCV) [22] es muy utilizada, con un número de descargas de alrededor de 15 millones y una comunidad de 47000 seguidores. Este dato está potenciado por el hecho de ser multiplataforma, soportando sistemas Windows, Linux, MacOS, Android e iOS. Algunos ejemplos de uso de la librería son compañías de la envergadura e influencia de Google, Yahoo, Microsoft, Intel, IBM, Sony, Honda o Toyota.

Los algoritmos que contiene están desarrollados de forma nativa en C++ debido a su mayor rendimiento en comparación con los demás lenguajes. El resto de ellos utilizan interfaces para acceder a toda la funcionalidad disponible. En el caso de Python, se utiliza a través del paquete cv2, que internamente utiliza la librería Numpy, altamente optimizada e indicada para el cálculo científico y las operaciones matemáticas con matrices.

Su integración en el proyecto permite aprovechar toda la potencia del análisis de imágenes en un entorno robótico para, en este caso, analizar imágenes de lo que observa el robot y poder detectar objetos de color dinámicamente. En este trabajo se ha empleado la versión 3.3.1.

3.7. Entorno de planificación de movimientos MoveIt!

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 de código abierto 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. El nodo *move_group* necesita varios parámetros de entrada (ver Figura 16):

- 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.

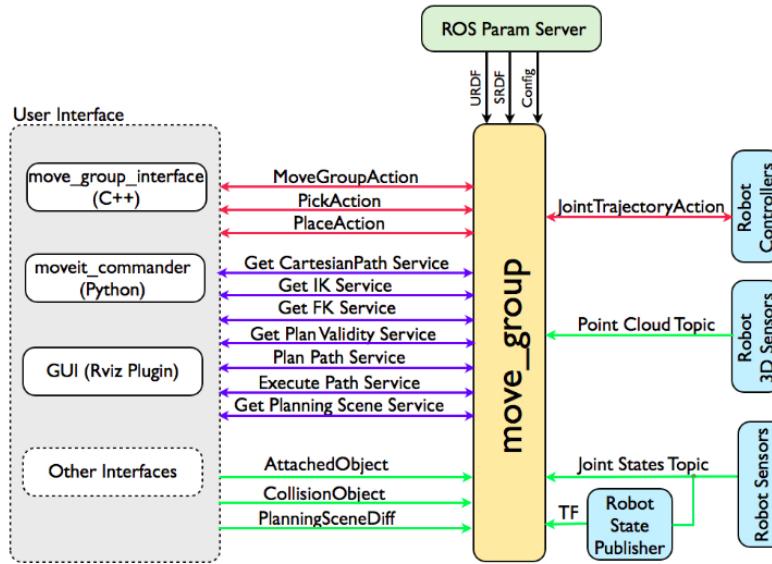


Figura 16: Resumen esquematizado de las posibles interacciones que ofrece *move_group*

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.

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 articulaciones que controla.

Al tratarse de la principal herramienta utilizada para completar el ejercicio, su funcionamiento y la manera en la que se ha empleado se detallarán en los siguientes capítulos.

Capítulo 4

Ejercicio de planificación de trayectorias con brazo robótico

En este capítulo se describirá en detalle la práctica ”Movimiento planificado de un brazo robótico” en el entorno Robotics-Academy, incluyendo el enunciado de la práctica, 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 observar un objeto y 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 utilizarán la librería OpenCV para el análisis de imágenes y el paquete MoveIt! para la fase de planificación.

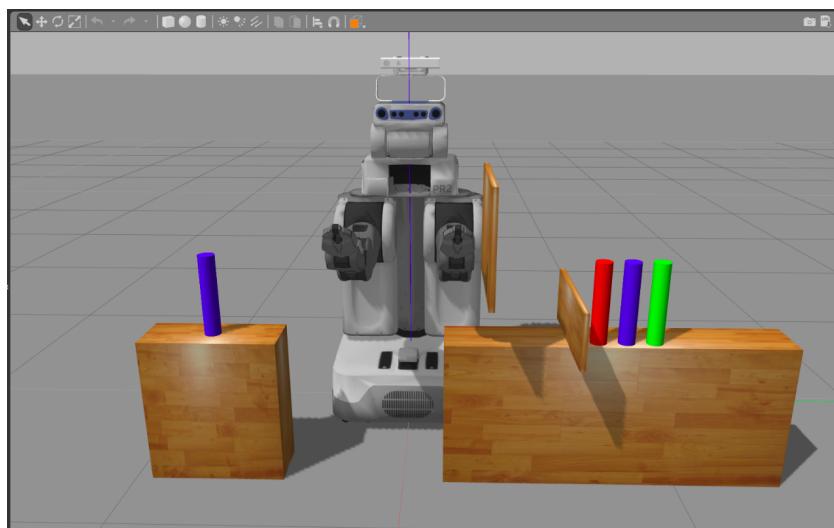


Figura 17: Mundo Gazebo que incluye todo lo necesario para el desarrollo de la práctica: zonas, objetos, robot y obstáculos.

El escenario de la práctica contiene un mundo Gazebo (ver Figura 17) que incluye dos zonas de acción (zona de muestra y zona de 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 solución 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 coincida con el observado en la zona de muestra. Para ello será necesario gestionar la planificación de trayectorias evitando los obstáculos necesarios y, por último, realizar un movimiento de derribo.

4.2. Diseño software

La estructura diseñada (ver Figura 18) tiene como núcleo el sistema ROS. A través de él, la plantilla creada puede comunicarse con el simulador y las librerías externas. La interacción con el simulador debe ser bidireccional ya que ambos deben conocer qué 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.

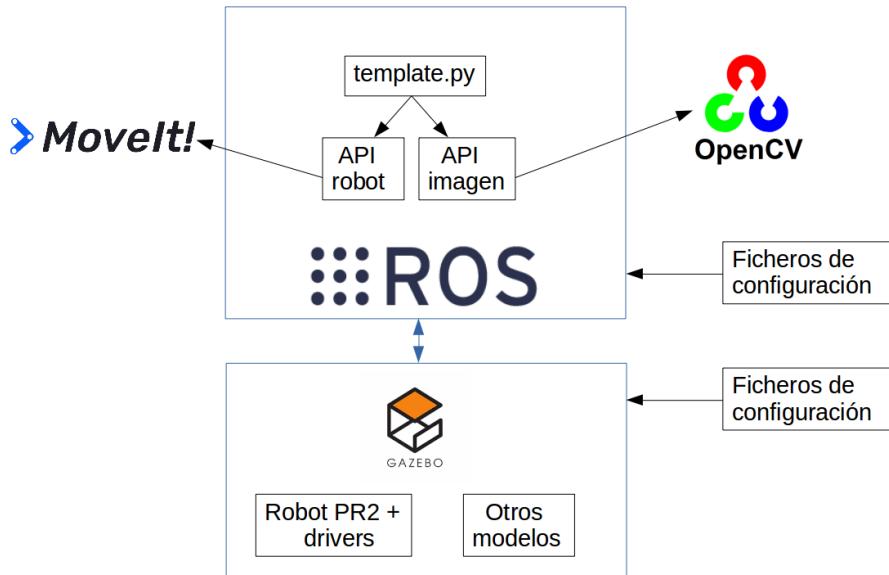


Figura 18: Esquema de la arquitectura software utilizada.

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.

La plantilla es un nodo ROS que ofrece las APIs de manejo de imágenes y de manejo del robot, que son utilizadas desde la solución del estudiante 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 (ver Figura 17) 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. Utiliza el plugin openni_Kinect (ver Algoritmo 2), que 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 permitirán detectar lo cerca o lejos que se encuentra un objeto de la cámara.

Para lanzar este entorno, se debe ejecutar un fichero *.launch* (ver Algoritmo 3), 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, aunque está configurada para aparecer por defecto. Se puede lanzar el mundo escribiendo en un terminal siguiente comando:

```
$ rosrun 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 [12] 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.

Algoritmo 2 Extracto del fichero kinect2.gazebo.xacro, donde se define el plugin para la cámara Kinect

```
<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>
```

Algoritmo 3 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>
```

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 sensor láser incluido por defecto en el modelo. El sensor generaba ruido en las imágenes RGB obtenidas, dificultando la caracterización de los objetos, motivo que provocó su desactivación en los paquetes generados.

Para llevar a cabo estas modificaciones, es necesario conocer el concepto de *overlaying* [37], utilizado en ROS para hablar de modificaciones en el espacio de trabajo [38] 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 (ver Algoritmo 4) 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 [33], de código abierto y basado en XML, que permite describir modelos de cualquier complejidad a través de sencillas etiquetas, desde esferas o prismas, hasta robots completos.

Algoritmo 4 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 Python

Para simplificar la realización de la práctica a los estudiantes, se ha desarrollado una plantilla en Python que ofrece interfaces de programación sencillas sobre las cuales el alumno escribe su solución. Esta plantilla es un nodo ROS que utiliza OpenCV y MoveIt!.

Como se ha mencionado, la solución 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! necesarios para la planificación de trayectorias.

Existen dos ficheros Python, uno llamado *final_prototype_template_ready.py* y otro llamado *template.py* (ver Algoritmo 5). 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, que se puede lanzar desde un terminal con el siguiente comando:

```
$ 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. Con ellas la plantilla simplifica al alumno el desarrollo de su solución a la práctica.

- 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* [8], 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* [23], 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 5 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

La solución de referencia desarrollada sobre ROS para este ejercicio consta de cinco fases. Una fase inicial de movimiento del cuello del robot, una segunda fase de percepción utilizando OpenCV, una tercera deplanificación utilizando MoveIt!, la cuarta de realización del movimiento del brazo, y finalmente una de maniobra de derribo.

5.1. Diseño de la solución

La resolución del problema a resolver en esta práctica ha supuesto en primer lugar imaginar y estructurar las diferentes fases que lo constituyen. La totalidad de la solución se puede dividir en las siguientes fases:

- Fase 1. El comienzo del ejercicio consiste en lograr que el robot PR2 modifique su posición original y mueva su cabeza hacia la derecha, de manera que pueda observar la zona de muestra. Por simplicidad y velocidad, el movimiento se ejecuta actuando directamente sobre los controladores de la cabeza del robot a través de ROS, con mensajes *PointHeadAction*.
- Fase 2. Una vez la cámara sobre la cabeza del robot apunta al objeto de muestra, se aplica el procesamiento de imágenes para identificar que lo que hay allí es realmente un objeto, y determinar su color. Para ello se utiliza la librería OpenCV, debido a su potencia.
- Fase 3. Se trata de la fase más importante de la solución de referencia. El robot debe planificar, a través de MoveIt!, una trayectoria con su brazo izquierdo que supere los obstáculos necesarios para situarse frente al objeto cuyo color corresponda con el de muestra. Debe ser lo suficientemente preciso para que la siguiente fase se pueda realizar satisfactoriamente en lazo abierto.
- Fase 4. Ejecución del movimiento planificado en la fase anterior, de modo que el brazo sorteá los obstáculos intermedios y alcanza con éxito la zona de derribo.
- Fase 5. Planificar y ejecutar un movimiento de derribo, que envía la pinza del robot unos centímetros hacia delante para golpear el objeto y hacerlo caer.

5.2. Movimiento inicial y procesamiento de imagen

Antes de 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.

5.2.1. Movimiento de cuello

El escenario inicial muestra al robot PR2 mirando hacia el frente, con la zona de muestra situada a su derecha. Con el objetivo de observar el objeto en dicha zona, el robot debe apuntar hacia él girando su cuello hacia la derecha y hacia abajo (ver Figura 19).

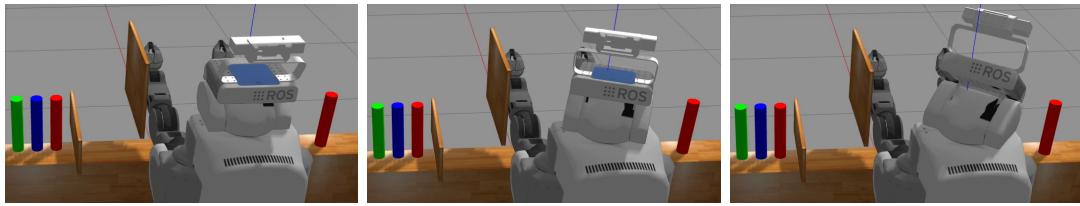


Figura 19: Secuencia del movimiento de la cabeza del robot

5.2.2. ROS topics para acceder a las imágenes de Kinect

El plugin que utiliza la cámara Kinect está configurado para refrescar las imágenes cada segundo, de forma que se puedan detectar dinámicamente 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, se utilizan 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.

5.2.3. Transformación a CVMat de OpenCV y al espacio HSV

Para realizar operaciones con imágenes en OpenCV, es necesario el llamado `cv_bridge`. Este paquete permite transformar las imágenes del mensaje de ROS a una de tipo `cv::Mat`, que en código Python se puede realizar de la siguiente manera:

```
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`:

```
color_HSV = cv2.cvtColor(cv2_img, cv2.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, abstrayéndonos de otros parámetros menos intuitivos y determinando correctamente el color del objeto observado por el robot.

5.2.4. Filtrado por colores

Como el objetivo final relativo al tratamiento de imágenes es diferenciar entre los colores rojo, verde y azul, son necesarias tres máscaras de color que realicen un filtrado de la información de color según su matiz, saturación y brillo, y caracterizar así lo que hay frente a la cámara (ver Figura 20).

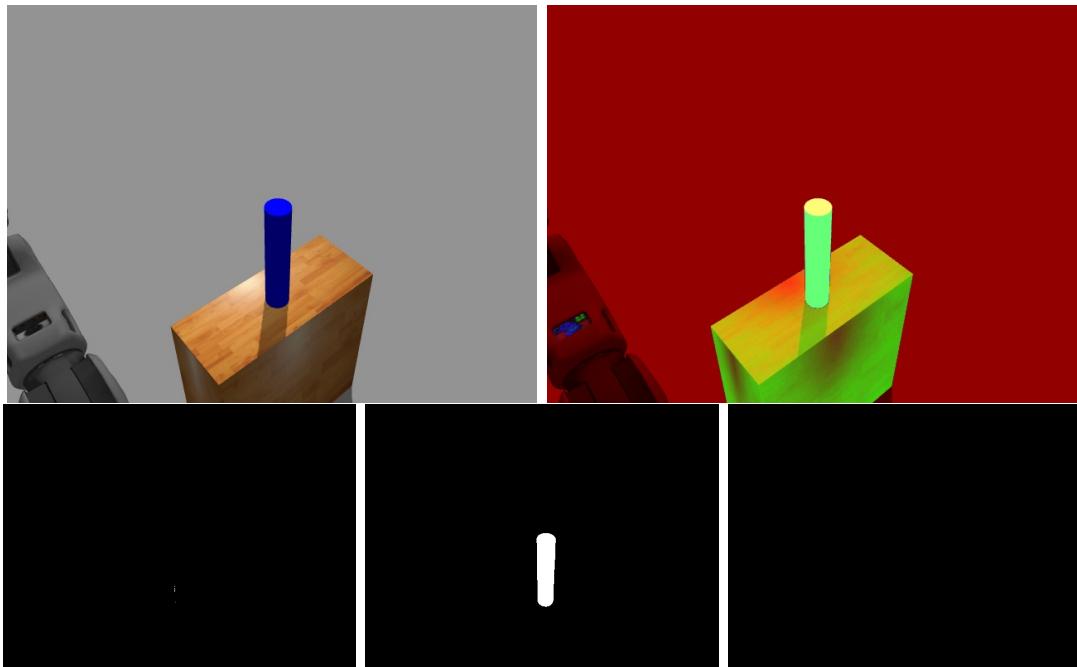


Figura 20: Secuencia que muestra la transformación al espacio HSV y los resultados tras aplicar las máscaras de color para el rojo, azul y verde, respectivamente

Una vez definidos los umbrales mínimos y máximos para cada uno de los colores a partir de valores del espacio HSV [6] [9], 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.

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*.

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 (ordenando todos los contornos encontrados y escogiendo

el primero de la lista) 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. Este procedimiento se realizará para los colores rojo, verde y azul (ver Algoritmo 6).

Algoritmo 6 Detección de un objeto azul utilizando OpenCV desde Python

```
lower_blue = np.array([110,50,50])
upper_blue = np.array([130,255,255])
blueMask = cv2.inRange(color_HSV, lower_blue, upper_blue)
resBlue = cv2.bitwise_and(cv2_img, cv2_img, mask= blueMask)
im2,contoursB,hierarchy = cv2.findContours(blueMask, cv2.
    RETR_TREE, cv2.CHAIN_APPROX_NONE)
contoursB = sorted(contoursB, key = cv2.contourArea, reverse
    = True) [:1]
if len(contours) > 0:
    area= cv2.contourArea(contours [0])
    if area > 500:
        print("es un objeto azul")
```

OpenCV ofrece además 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.

5.3. Movimiento de aproximación del brazo 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 (ver Figura 21).

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 (ver Algoritmo 7).

La planificación de trayectorias requiere un fichero de lanzamiento (ver Algoritmo 8) 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.

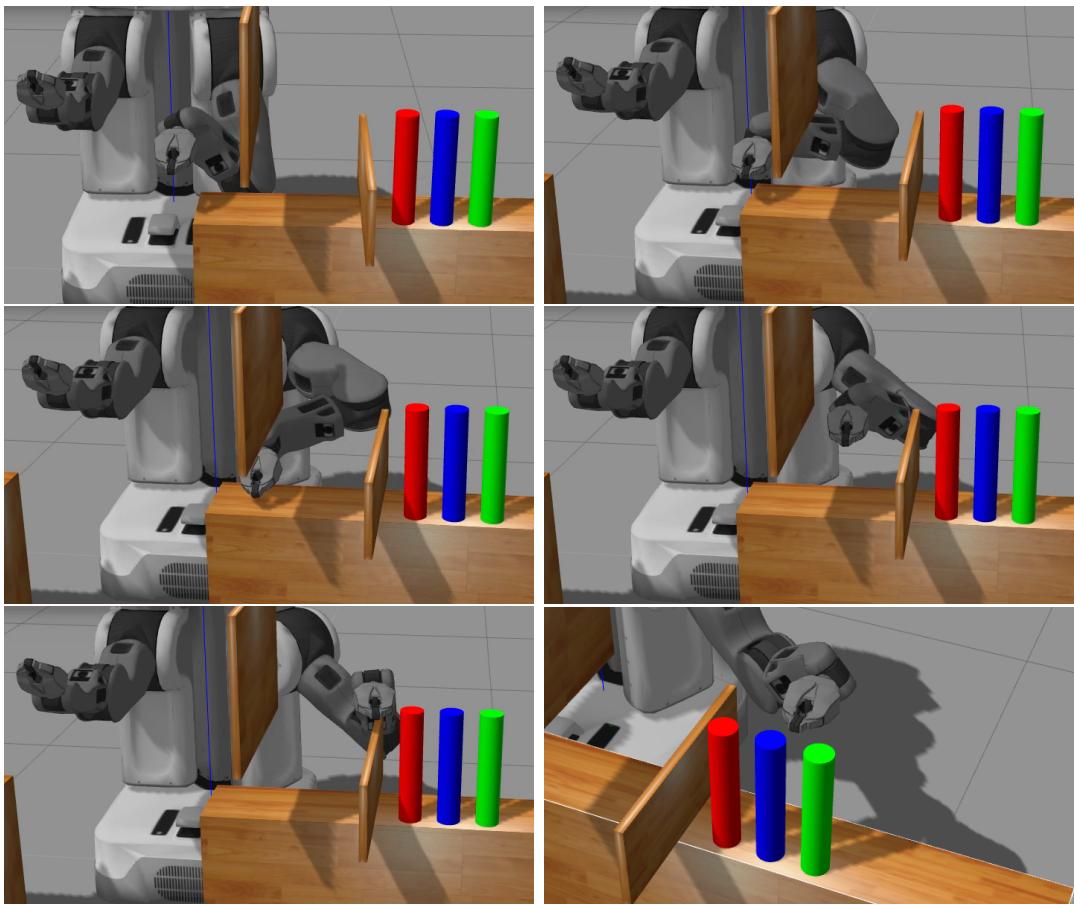


Figura 21: Secuencia de la ejecución de la trayectoria de aproximación planificada. El primer fotograma muestra la posición inicial de la maniobra, y el último la posición final. Entre ellas, es necesario sortear dos obstáculos.

Algoritmo 7 Planificación y ejecución de la primera fase del movimiento

```

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)
  
```

Algoritmo 8 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>

```

5.3.1. moveit_commander

moveit_commander es un módulo que permite acceder a las principales funcionalidades de MoveIt! a través de Python. Contiene tres clases llamadas *MoveGroupCommander*, *PlanningSceneInterface* y *RobotCommander*, que permitirán realizar todas las acciones necesarias para completar el ejercicio de planificación y ejecución de movimientos en un entorno con obstáculos.

La clase *RobotCommander* ofrece una interfaz al robot y la capacidad de obtener información sobre él. Son especialmente útiles dos de las funciones disponibles:

- *get_group_names*: Obtiene los grupos definidos para el robot, es decir, conjuntos de articulaciones y enlaces bajo un mismo nombre. Por ejemplo el grupo 'arms' engloba todo lo que hay en los brazos derecho e izquierdo.
- *get_current_state*: Obtiene el estado actual del robot, incluyendo todas las articulaciones del robot y sus posiciones relativas.

Para poder acceder al mundo que rodea al robot, es necesario utilizar la clase *PlanningSceneInterface*. Algunas de las funciones relacionadas con los obstáculos utilizadas en la práctica son las siguientes:

- *addBox*: Añade un objeto con forma de ortoedro al mundo para que sea considerado como colisión durante la planificación.
- *addCylinder*: Añade un objeto con forma de cilindro al mundo para que sea considerado como colisión durante la planificación.
- *removeCollisionObject*: Elimina un objeto del mundo. No se tendrá en cuenta para el cálculo de la planificación.

Las funciones que ofrece *MoveGroupCommander* [17] se pueden agrupar en aquellas que permiten obtener información sobre el robot o la planificación, y aquellas que permiten definir el comportamiento del robot o la planificación. Algunas de las más relevantes para el desarrollo de este ejercicio son:

- *get_planning_frame*: Permite obtener el nombre del marco de referencia para la planificación. A partir de él se obtienen las posiciones del resto de elementos del robot.
- *get_end_effector_link*: Permite obtener el nombre del elemento final del robot. En este caso, corresponde a la pinza que tiene el robot PR2 en el brazo izquierdo.
- *set_pose_target*: Permite establecer el objetivo final de una planificación de movimiento.
- *set_path_constraints*: Permite configurar restricciones relativas a alguna de las articulaciones para la planificación. Ello limitará las posiciones posibles en el espacio de configuraciones.
- *set_planning_time*: Permite definir el tiempo máximo dedicada a la planificación antes de generar un error.
- *set_num_planning_attempts*: Permite definir el número de intentos de planificación a realizar en cada ejecución.
- *plan*: Permite planificar una trayectoria entre dos puntos.
- *execute*: Permite ejecutar un movimiento previamente planificado.

5.3.2. Controladores

Para que la planificación y ejecución de los movimientos del robot sean adecuadas, son necesarios unos controladores bien configurados para cada una de las articulaciones del brazo [7]. 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 (ver Algoritmo 9) 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. [19]

Algoritmo 9 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
```

5.3.3. Planificador y configuración de MoveIt!

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. [18]

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 utiliza 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, más indicados 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. Este tipo de algoritmos son útiles en situaciones donde la presencia de obstáculos o la cantidad de grados de libertad del robot aumentan considerablemente la complejidad de los cálculos.

Para comprender su funcionamiento, es importante introducir el concepto de espacio de configuraciones. Se denomina configuración a la especificación completa de las posiciones de cada punto de un sistema mecánico, es decir, el espacio de configuraciones del robot PR2 es cada una de las orientaciones y posiciones válidas que puede alcanzar.

Los obstáculos del entorno, por ejemplo, supondrían posiciones inválidas en el espacio de configuraciones, así como aquellas que supongan colisiones con el propio robot. Se diferencia así del espacio de trabajo, que englobaría todas las posiciones posibles sin tener en cuenta si son o no válidas (ver Figura 22).

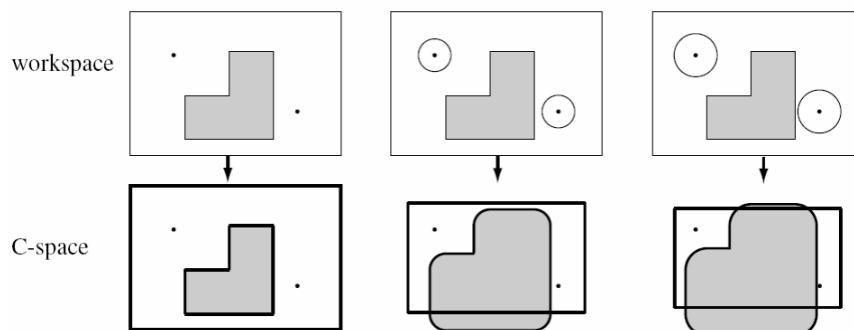


Figura 22: Explicación gráfica en 2D de la diferencia entre el espacio de trabajo y el espacio de configuraciones.

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 cumplan 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 (ver Figura 23). Nótese que cuanto más precisos sean los saltos hacia el siguiente punto del árbol, menor será la probabilidad de infringir zonas inválidas del espacio de configuraciones del robot.

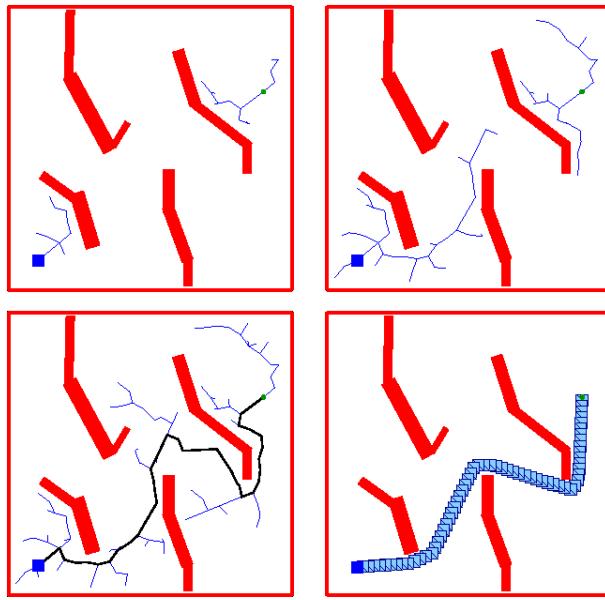


Figura 23: Explicación gráfica del algoritmo RRT Connect utilizado.

5.4. Maniobra de derribo

Una vez ejecutado el primer movimiento, se vuelve a realizar una planificación para el movimiento de derribo (ver Figura 24). Se utiliza el mismo mensaje de tipo *Pose*, pero añadiendo 15cm a la posición x (que corresponde a ir hacia delante para el robot).

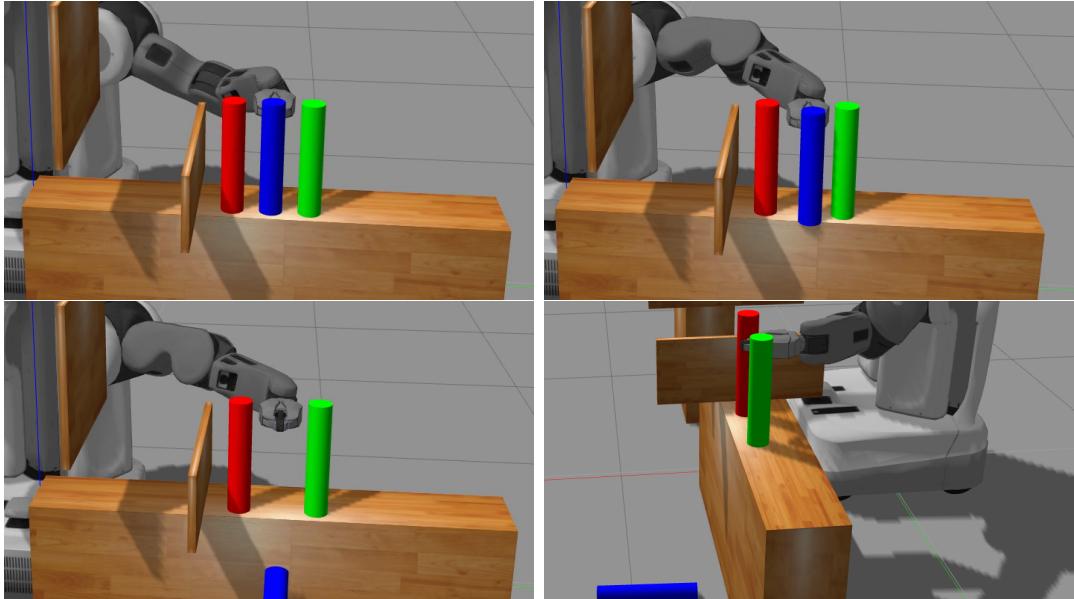


Figura 24: Secuencia que muestra el movimiento de derribo

Así, el código para el segundo movimiento (ver Algoritmo 10) logra situar la mano del robot donde se encontraba el objeto de color, haciéndolo caer. Nótese que en este punto debemos dejar de considerar el objeto destino como colisión, utilizando la interfaz al mundo que ofrece *moveit_commander*.

Algoritmo 10 Planificación y ejecución del movimiento de derribo

```

if color=='red':
    p.removeCollisionObject("redCylinder")
elif color =='blue':
    p.removeCollisionObject("blueCylinder")
elif color =='green':
    p.removeCollisionObject("greenCylinder")
else:
    print("No objects to remove")
pose_target.position.x = pose_target.position.x + 0.15
group_left.set_pose_target(pose_target)
plan2 = group_left.plan()
group_left.execute(plan2)

```

5.5. Validación experimental

La ejecución típica ^{1 2} 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.launch
```
- Lanzar MoveIt! y las configuraciones de planificación necesarias para el robot PR2.
- ```
$ roslaunch pr2_moveit_config
```
- 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 tiene lugar la fase 1, donde podemos observar al robot PR2 iniciando un movimiento de su cabeza hacia el lado derecho, apuntando la cámara Kinect hacia la zona de muestra (ver Figura 25).

<sup>1</sup>Vídeo de ejemplo para la ejecución típica de la práctica con un objeto de color azul:  
<https://youtu.be/OU1iNR0tzyA>

<sup>2</sup>Vídeo de ejemplo para la ejecución típica de la práctica con un objeto de color rojo:  
[https://youtu.be/Np2g\\_3XHoH4](https://youtu.be/Np2g_3XHoH4)

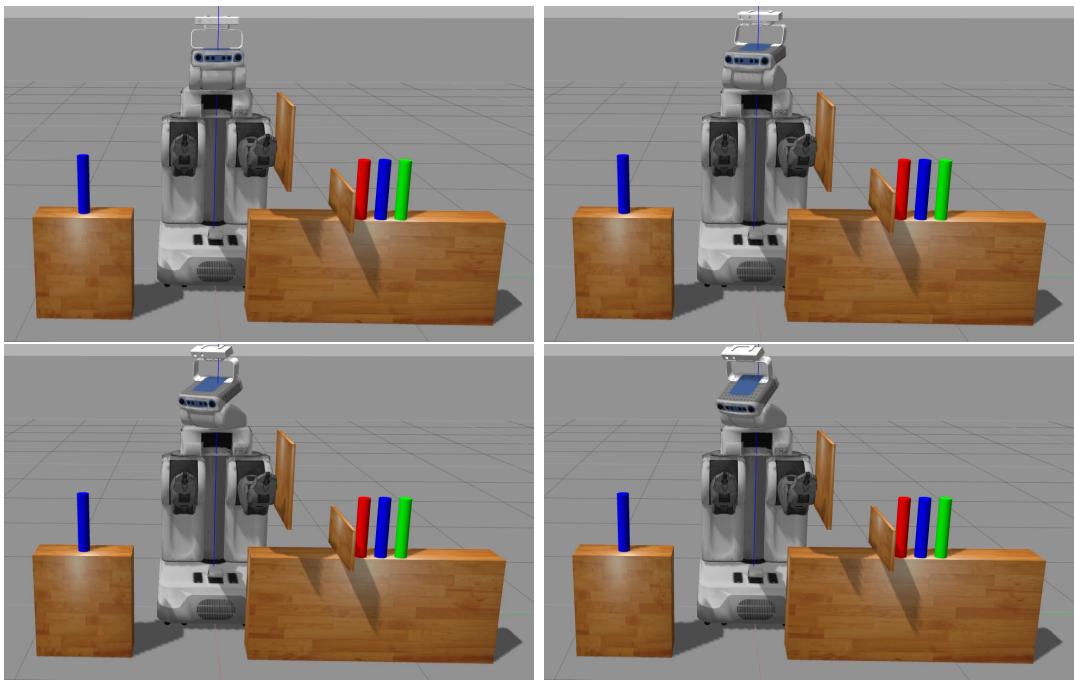


Figura 25: Secuencia que muestra la primera fase del ejercicio

A continuación se obtiene una imagen RGB a través de Kinect que se debe transformar al espacio HSV utilizando OpenCV para después aplicar las máscaras de color 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 conocido el color del objeto, y por tanto la posición destino a utilizar como meta del primer movimiento, se calcula una trayectoria hasta ella utilizando MoveIt!. Se realizarán entonces los dos trayectos en que se divide el movimiento. El primero de ellos es el más complejo, y consiste en pasar por debajo del primer obstáculo y por encima del segundo y situarse frente al objeto cuyo color coincide con el de la zona de muestra (ver Figura 26).

Por último, se puede observar el movimiento de derribo con el extremo del brazo izquierdo del robot, haciendo caer el objeto esperado y finalizando así la ejecución de la práctica (ver Figura 27).

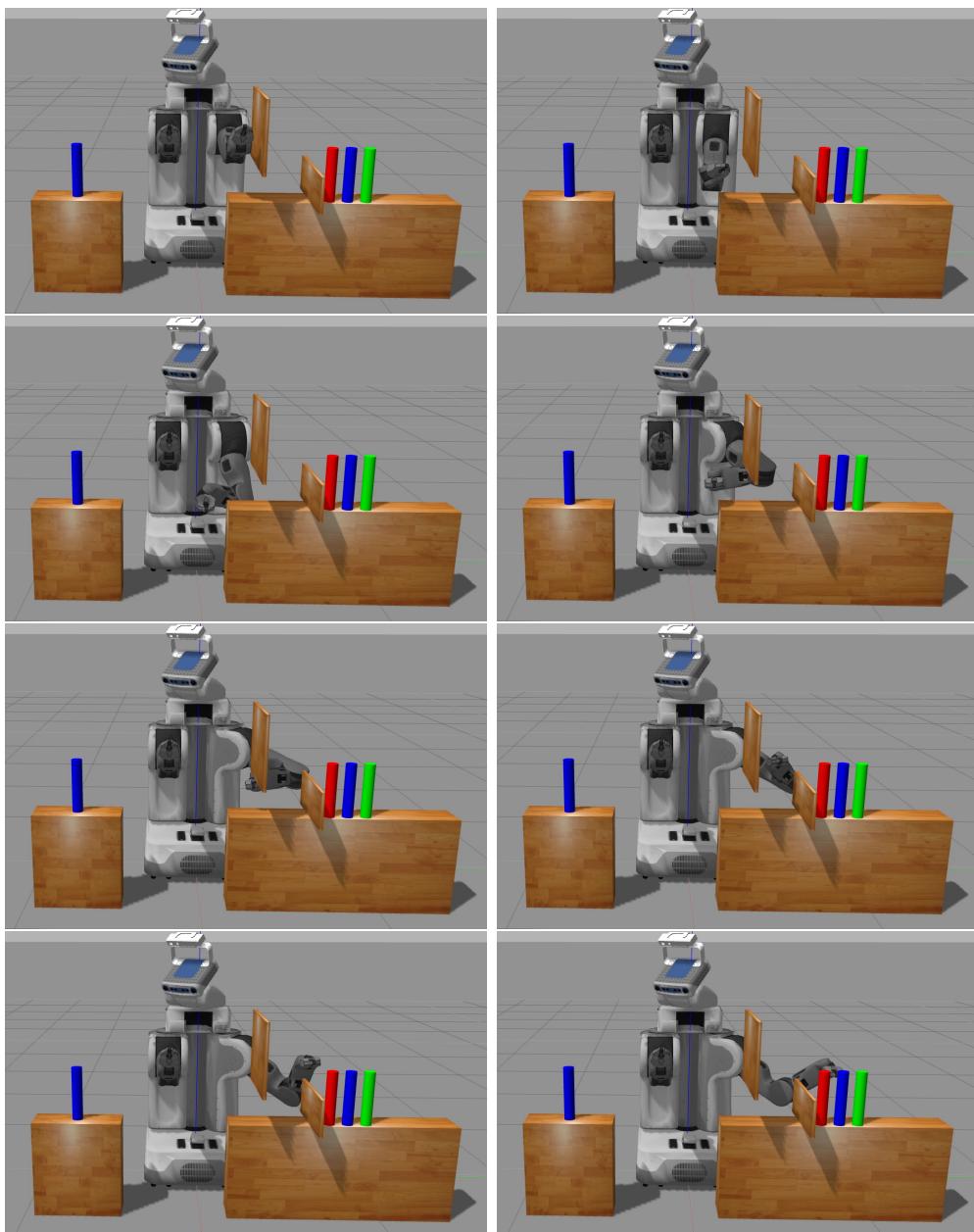


Figura 26: Secuencia que muestra el movimiento de aproximación del brazo

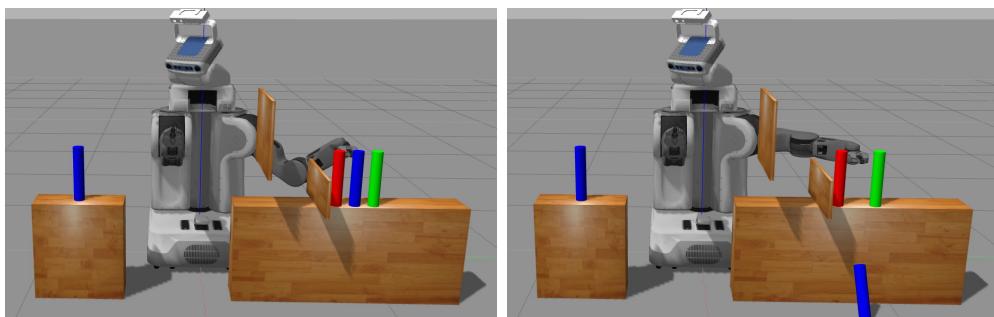


Figura 27: Secuencia que muestra el movimiento de derribo del brazo

## 5.6. Alternativas probadas

La solución de referencia desarrollada ha requerido el estudio de diferentes alternativas en búsqueda de un comportamiento adecuado de la planificación de trayectorias. Se han realizado ajustes en el entorno y el planificador, se han considerado planificadores alternativos, se han establecido restricciones de planificación y se han realizado pruebas de *Pick and Place*.

### 5.6.1. Ajuste del entorno mediante RViz

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 (ver Figura 28) o la distancia idónea para el apartado visual y estético.

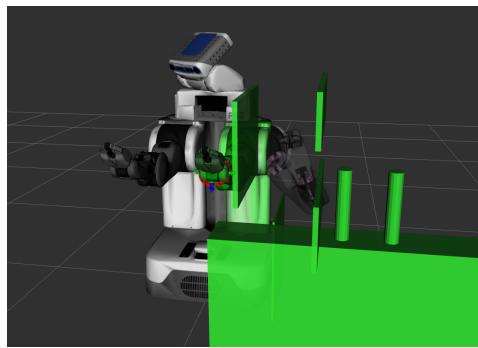


Figura 28: Situación creada dinámicamente en RViz durante las pruebas con obstáculos

Para llevar a cabo este ensayo se ha utilizado el visualizador RViz, una herramienta de ROS integrada en 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 *pr2\_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* (ver Algoritmo 11).

---

**Algoritmo 11** 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>
```

---

### 5.6.2. Ajuste del planificador de trayectorias

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 cómo se calculan las posibles trayectorias en OMPL implica que el valor pre-determinado por MoveIt! para este parámetro, que es configurable a través del 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* (ver Figura 29).

Obstáculos	<i>longest_valid_segment_fraction</i>	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 29: Respuesta del sistema ante distintas situaciones de entorno y diferentes configuraciones del planificador.

Como se puede observar, 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*.

### 5.6.3. Otros planificadores

Aunque RRT Connect es el algoritmo por defecto que utiliza MoveIt!, la interfaz de Python que ofrece permite seleccionar otros planificadores. Se han valorado otras alternativas de OMPL, concretamente algoritmos con diferencias de funcionamiento considerables, como SBL y KPIECE.

El algoritmo SBL está basado en árboles, creando uno desde el origen y otro desde el destino de manera análoga a RRT Connect. Sin embargo, cada vez que se añade un nodo a los árboles, se tratan de conectar. En ese punto, se valida si la solución obtenida es correcta y en caso negativo, se eliminan esos puntos y la búsqueda continua. Las validaciones adicionales en comparación con RRT Connect penalizan su rendimiento para

el problema propuesto en este ejercicio, ya que no obtiene resultados en el tiempo máximo de 40 segundos establecido para la planificación. Aumentando dicho tiempo, se comprueba que sus tiempos son alrededor de 10 veces superiores al planificar RRT Connect con la misma configuración.

El algoritmo KPIECE realiza una discretización del espacio en diferentes niveles de precisión, utilizando celdas. A través de priorizar los límites no explorados y combinar los resultados de cada nivel, obtiene un camino correcto que une el inicio y el fin de la planificación. Se observan resultados similares a RRT Connect para el caso de estudio, aunque ligeramente superiores en tiempo y con mayor irregularidad.

#### 5.6.4. Restricciones de planificación

Con el objetivo de respetar el recorrido en forma de cueva para sortear los obstáculos presentes en el mundo creado, son necesarias algunas restricciones de planificación. La interfaz de programación de manejo del robot creada para el ejercicio ofrece una función llamada *set\_orientation\_constraints* que permite restringir los movimientos de una determinada articulación.

Las restricciones en esta práctica aseguran que el robot siga la ruta esperada y no tome atajos que no se consideran válidos para el ejercicio, como por ejemplo encoger el brazo y sortear los obstáculos de golpe por el lado en lugar de por encima o por debajo. A través de mensajes de MoveIt! de tipo *OrientationConstraints*, se han definido restricciones de orientación y de giro en los ejes (ver Algoritmo 12). De esta manera el agarre del brazo izquierdo siempre se encuentra orientado hacia delante, y además el giro en la muñeca es limitado lo suficiente para garantizar que se planifica una trayectoria adecuada para los objetivos propuestos.

---

**Algoritmo 12** Establecimiento de restricciones de planificación para uno de los elementos del robot utilizando mensajes de MoveIt!

---

```
ocm = moveit_msgs.msg.OrientationConstraint()
ocm.link_name = link
ocm.header.frame_id = "base_link"
ocm.orientation.w = orientation
ocm.absolute_x_axis_tolerance = x_tol
ocm.absolute_y_axis_tolerance = y_tol
ocm.absolute_z_axis_tolerance = z_tol
ocm.weight = 1.0
constraints= moveit_msgs.msg.Constraints()
constraints.orientation_constraints.append(ocm)
group_left.set_path_constraints(constraints)
```

---

### 5.6.5. Pruebas con la mano robótica (*gripper*)

Una de las alternativas probadas durante el desarrollo de la práctica consiste en utilizar las posibilidades de la clase *MoveGroupCommander* para lograr la funcionalidad de *Pick and Place*. En este comportamiento, se transportaría un objeto durante la trayectoria desde la zona de muestra hasta una determinada posición en la zona de destino, que sustituiría a la actual zona de derribo. Esta funcionalidad implica por tanto agarrar un objeto con la mano robótica en la zona de muestra y soltarlo en la actual zona de derribo.

Trabajando con los tutoriales oficiales de MoveIt! para ROS Kinetic como referencia, y adaptando los referentes a *Pick and Place* del lenguaje C++ a código Python, se logró utilizar las funciones *pick* y *place*. En el visualizador RViz es posible realizar un movimiento que agarre y transporte un objeto adherido a la pinza del robot a un lugar destino (ver Figura 30), utilizando mensajes de MoveIt! de tipo *Grasp*. Sin embargo, los problemas llegaron al tratar de llevar el comportamiento al mundo simulado, enfrentando problemas de integración con el *gripper* del robot PR2, cuyos controladores experimentaban problemas de *timeout* al tratar de abrir la pinza en Gazebo.

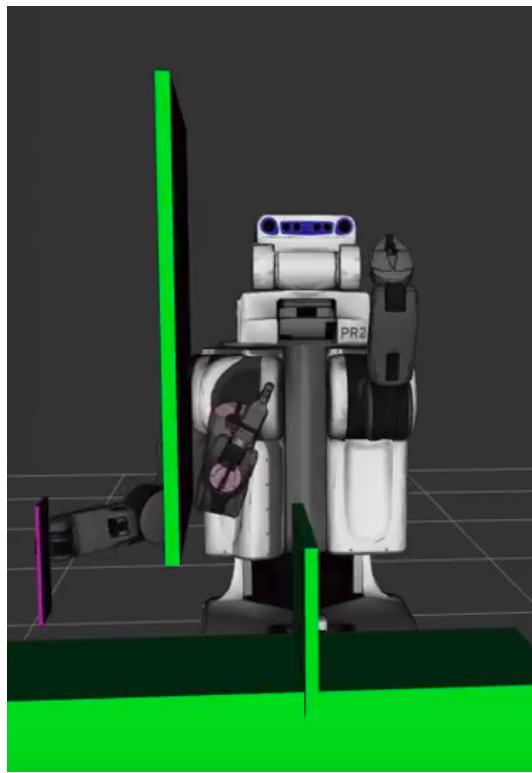


Figura 30: Objeto adherido al robot en el visualizador RViz utilizando las funciones de *Pick and Place* que ofrece la clase *MoveGroupCommander*

Tras realizar consultas sobre el problema a través de los canales oficiales de ROS<sup>3</sup> y MoveIt!<sup>4</sup>, se concluye que los tutoriales para el robot PR2 necesitan actualizaciones para ROS Kinetic. Esto es debido a que los esfuerzos están centrados en adaptar los tutoriales de MoveIt! para ROS Kinetic a un nuevo robot, el Panda de Franka Emika [28], que se convirtió en el robot de referencia para los tutoriales de ROS Kinetic en mayo de 2018.

<sup>3</sup>Consulta en la comunidad de ROS: <https://answers.ros.org/question/293834/how-to-integrate-pr2-gripper-with-moveit-gazebo/>

<sup>4</sup>Consulta en los repositorios de MoveIt!: <https://github.com/ros-planning/moveit/issues/110>



# 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 de este Trabajo de Fin de Grado.

### 6.1. Conclusiones

El objetivo principal 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 ámbito robótico. Se logra así enriquecer el entorno educativo Robotics-Academy con el primer ejercicio sobre brazos robóticos disponible.

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 que incluye el habitual software heterogéneo de robótica. Desde el nodo ROS creado se han integrado elementos que van desde la biblioteca de procesamiento de imágenes OpenCV hasta el entorno de planificación de movimientos MoveIt!.

El camino hasta la solución de referencia ha sido largo, con numerosas alternativas probadas. Estas dificultades han requerido aprender cada semana para resolver los problemas del día a día, mejorando así en una de las facetas clave de la ingeniería y adquiriendo nuevos conocimientos.

Primero, dentro del primer subobjetivo, se ha creado la infraestructura de la práctica "Movimiento planificado de un brazo robótico". Para ello se ha creado un mundo en Gazebo (ver sección 4.3) que consiste en un robot PR2, dos zonas de acción, cilindros de colores y algunos obstáculos, que sirve como punto de partida para poder desarrollar el ejercicio de planificación. El modelo del robot PR2 ha requerido algunos cambios, como añadir por defecto una cámara Kinect sobre su cabeza o la desactivación del sensor láser para evitar problemas de ruido en las imágenes obtenidas por la cámara.

Segundo, se ha desarrollado una plantilla ROS en Python que simplifica el contacto con el procesamiento de imagen y la planificación con robots, se ha desarrollado un fichero *template.py* que permite acceder a través de dos APIs a las funcionalidades relacionadas con cada uno de dichos procesos (ver sección 4.4). De esta manera, los alumnos pueden

abstraerse de la complejidad de OpenCV y MoveIt! y preocuparse únicamente de obtener una solución válida, sin necesidad de configurar las transformaciones entre espacios de color, los umbrales de detección de contornos en imágenes, los controladores del PR2, los umbrales adecuados para el planificador RRT Connect o los ROS *topics* utilizados para definir las posiciones objetivo de los distintos movimientos.

Al hilo del tercer subobjetivo, se ha desarrollado una solución de referencia (ver sección 5) a través del diseño de varias fases que dividen el problema a resolver. La primera de ellas consiste en actuar sobre los controladores de la cabeza del PR2 para lograr que gire a la derecha y observe la zona de muestra. A continuación, se utiliza la cámara Kinect para obtener una imagen que se transforma al espacio de color HSV con OpenCV y detecta si realmente hay un objeto y su color. Después, la interfaz de MoveIt! para Python *moveit\_commander* permite planificar y ejecutar una trayectoria de aproximación con el brazo izquierdo del robot hasta situarlo frente al objeto a derribar, que será aquel cuyo color coincide con la zona de muestra. Por último, se realiza un movimiento de derribo que hace caer el cilindro.

En cuanto a la evaluación personal de este TFG, 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 ampliamente extendida. Ello ha permitido profundizar en habilidades útiles para el entorno laboral, 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 abre la puerta a algunas líneas de trabajo futuras que, 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 respecto del robot, y utilizar dichas posiciones para calcular el destino de la planificación. Esta funcionalidad podría completarse estimando también la localización y tamaño de los objetos alrededor del robot y tomándolos 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, es decir, extenderlo al comportamiento *Pick and 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.

Tercero, podría ser conveniente adaptar el código a la nueva versión ROS2 [32], acompañada recientemente de la versión alfa de MoveIt!2 [20]. Ambas introducen mejoras importantes relacionadas con planificación, movimientos del robot y nueva funcionalidad.

Cuarto, igual que el robot PR2 se escogió por motivos de diseño y compatibilidad con ROS Kinetic, sería conveniente migrar la solución a un nuevo robot. Los nuevos lanzamientos incluyen el robot modular MARA [1], de Acutronic Robotics, como el primer robot compatible.

Quinto, se podría hacer un estudio teórico y obtener métricas relacionadas con la planificación. Una opción sería profundizar en los algoritmos de planificación disponibles en MoveIt! y comparar su rendimiento en distintas situaciones, o incluso generar los propios para controlar su funcionamiento en búsqueda del mayor beneficio.



# Bibliografía

- [1] *Acutronic Robotics: MARA*. URL: <https://acutronicrobotics.com/products/mara/> (vid. pág. 52).
- [2] *Automation Forecast*. URL: <https://www.interquestgroup.com/insights/infographics/2018/robotic-process-automation-industry-forecast> (vid. pág. 3).
- [3] *Bitácora en GitHub Pages del TFG*. URL: <http://roboticsurjc-students.github.io/2016-tfg-Ignacio-Malo/> (vid. pág. 13).
- [4] *Carlos Awadallah Estévez. Nuevas Prácticas en el Entorno Docente de Robótica JdeRobot-Academy, 2018*. URL: [https://gsyc.urjc.es/jmplaza/students/tfg-JdeRobot\\_Academy-carlos\\_awadallah-2018.pdf](https://gsyc.urjc.es/jmplaza/students/tfg-JdeRobot_Academy-carlos_awadallah-2018.pdf) (vid. pág. 8).
- [5] *CLARAty*. URL: <https://www-robotics.jpl.nasa.gov/facilities/facility.cfm?Facility=2> (vid. pág. 5).
- [6] *Colores HSV*. URL: <http://www.workwithcolor.com/red-color-hue-range-01.htm> (vid. pág. 35).
- [7] *Configure MoveIt! controllers*. URL: <http://www.theconstructsim.com/control-gazebo-simulated-robot-moveit-video-answer/> (vid. pág. 39).
- [8] *cvbridgePython*. URL: [http://wiki.ros.org/cv%5C\\_bridge/Tutorials/ConvertingBetweenROSImagesAndOpenCVImagesPython](http://wiki.ros.org/cv%5C_bridge/Tutorials/ConvertingBetweenROSImagesAndOpenCVImagesPython) (vid. pág. 29).
- [9] *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](https://opencv-python-tutroals.readthedocs.io/en/latest/py_tutorials/py_imgproc/py_colorspaces/py_colorspaces.html) (vid. pág. 35).
- [10] *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).
- [11] *Gazebo*. URL: <http://gazebosim.org/> (vid. pág. 17).
- [12] *Gazebo camera plugins*. URL: [http://gazebosim.org/tutorials?tut=ros\\_gzplugins](http://gazebosim.org/tutorials?tut=ros_gzplugins) (vid. pág. 25).
- [13] *JdeRobot*. URL: [https://jderobot.org/Main\\_Page](https://jderobot.org/Main_Page) (vid. pág. 7).
- [14] *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).
- [15] *Microsoft Robotics Developer Studio*. URL: <https://www.microsoft.com/en-us/download/details.aspx?id=29081> (vid. pág. 5).

## BIBLIOGRAFÍA

---

- [16] *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).
- [17] *MoveGroupCommander.* URL: [http://docs.ros.org/jade/api/moveit\\_commander/html/classmoveit\\_\\_commander\\_1\\_1move\\_\\_group\\_1\\_1MoveGroupCommander.html](http://docs.ros.org/jade/api/moveit_commander/html/classmoveit__commander_1_1move__group_1_1MoveGroupCommander.html) (vid. pág. 39).
- [18] *MoveIt! Planners.* URL: <https://moveit.ros.org/documentation/planners/> (vid. pág. 40).
- [19] *MoveIt! tutorials.* URL: [http://docs.ros.org/kinetic/api/moveit\\_tutorials/html/index.html](http://docs.ros.org/kinetic/api/moveit_tutorials/html/index.html) (vid. pág. 39).
- [20] *MoveIt!2.* URL: <https://moveit.ros.org/moveit!/ros/2019/05/31/moveit2-alpha-release.html> (vid. pág. 52).
- [21] *Open Robot Control Software.* URL: <http://www.orocos.org/> (vid. pág. 5).
- [22] *OpenCV.* URL: <https://opencv.org/> (vid. pág. 20).
- [23] *OpenCV: Image Transformations.* URL: [https://docs.opencv.org/2.4/modules/imgproc/doc/miscellaneous\\_transformations.html](https://docs.opencv.org/2.4/modules/imgproc/doc/miscellaneous_transformations.html) (vid. pág. 29).
- [24] *Pablo Moreno Vera. Nuevas Prácticas Docentes en el Entorno Robotics-Academy, 2017.* URL: <https://github.com/RoboticsURJC-students/2017-tfg-pablo-moreno/blob/master/TFG/Memoria.pdf> (vid. pág. 8).
- [25] *Primeros graduados con conocimientos en robótica.* URL: [https://elpais.com/ccaa/2018/10/24/madrid/1540404720\\_746066.html](https://elpais.com/ccaa/2018/10/24/madrid/1540404720_746066.html) (vid. pág. 6).
- [26] *Ranking TIOBE.* URL: <https://www.tiobe.com/tiobe-index/> (vid. pág. 19).
- [27] *Repositorio GitHub del TFG.* URL: <https://github.com/RoboticsURJC-students/2016-tfg-Ignacio-Malo> (vid. pág. 13).
- [28] *Robot PANDA.* URL: <https://www.franka.de/technology> (vid. pág. 49).
- [29] *Robot PR2.* URL: <http://www.willowgarage.com/pages/pr2/overview> (vid. pág. 15).
- [30] *Robotics Market.* URL: <https://www.mordorintelligence.com/industry-reports/robotics-market> (vid. pág. 3).
- [31] *ROS.* URL: <http://www.ros.org/> (vid. pág. 5).
- [32] *ROS2.* URL: <https://index.ros.org/doc/ros2/Releases/> (vid. pág. 52).
- [33] *SDF Format.* URL: <http://sdformat.org/spec> (vid. pág. 27).
- [34] *Spiral Model.* URL: <http://www.dimap.ufrn.br/~jair/ES/artigos/SpiralModelBoehm.pdf> (vid. pág. 12).
- [35] *Vanessa Fernández Martínez. Nuevas Prácticas en el Entorno Docente de Robótica JdeRobot-Academy, 2017.* URL: [https://gsyc.urjc.es/jmplaza/students/tfg-JdeRobot\\_Academy-vanessa\\_fernandez-2017.pdf](https://gsyc.urjc.es/jmplaza/students/tfg-JdeRobot_Academy-vanessa_fernandez-2017.pdf) (vid. pág. 8).
- [36] *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).
- [37] *Workspace overlaying in ROS.* URL: [http://wiki.ros.org/catkin/Tutorials/workspace\\_overlaying](http://wiki.ros.org/catkin/Tutorials/workspace_overlaying) (vid. pág. 27).

- [38] *wstool*. URL: <http://wiki.ros.org/wstool> (vid. pág. 27).