



Universidad
Rey Juan Carlos

Escuela Técnica Superior de Ingeniería de Telecomunicación

MOVIMIENTO DE UN BRAZO ROBÓTICO EN EL ENTORNO EDUCATIVO UNIBOTICS

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

Autor: Ignacio Malo Segura

Tutor: José María Cañas Plaza

Co-tutor: José Francisco Vélez Serrano

2019

Agradecimientos

Resumen

Índice general

1	Introducción	1
1.1	Introducción a la robótica	1
1.1.1	El sector: presente y futuro	2
1.2	Software en robótica	3
1.3	Docencia en robótica	5
1.3.1	Robótica en la escuela	5
1.3.2	Robótica al alcance de todos	6
1.3.3	JdeRobot-Academy	7
2	Objetivos	9
2.1	Objetivos	9
2.2	Requisitos	9
2.3	Metodología	10
2.4	Plan de acción	10
3	Infraestructura	13
3.1	PR2	13
3.1.1	Microsoft Kinect	13
3.2	Gazebo	14
3.3	JdeRobot	15
3.4	ROS Kinetic	16
3.5	ICE	17
3.6	C++	17
3.7	Python	17
3.8	OpenCV	18
3.9	MoveIt!: Motion Planning Framework	18
4	Aplicación con brazo robótico y planificación de trayectorias	21
4.1	Enunciado	21
4.2	Infraestructura	22
4.2.1	Mundo Gazebo	22
	Modelo PR2	22
	Modelos adicionales	23
4.3	Plantilla	24
5	Solución de referencia	27
5.1	Procesamiento de imagen	27
5.1.1	Cámara Kinect y ROS topics	27

5.1.2 Librería OpenCV	27
5.2 Planificación de trayectorias con MoveIt	29
5.2.1 MoveGroupCommander	30
5.2.2 Controladores	33
5.2.3 Planificador y configuración	34
5.3 Experimentación	35
6 Conclusiones	37
6.1 Conclusiones	37
6.2 Trabajos futuros	38
Bibliografía	39

Capítulo 1

Introducción

1.1. Introducción a la 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 sustituir tareas que realizan las personas.

Para tratar de entender la importancia del tema expuesto, es necesario 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. Ellos son los que se analizarán en mayor detalle.

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 (petróleo, electricidad) que aceleraron enormemente el crecimiento industrial y permitieron mejorar la calidad del transporte. Sumado a los avances científicos y la mejora de la calidad de vida de las personas, es evidente que estamos ante otro punto clave para la tecnología.

El siguiente gran salto es el más cercano en el tiempo y, sin duda, a pesar de que no hubiese sido posible sin los dos anteriores, es el que propicia en última instancia la 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, 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 y se evidencia con los sucesos expuestos, el ser humano quiere, puede, y debe seguir progresando tecnológicamente.

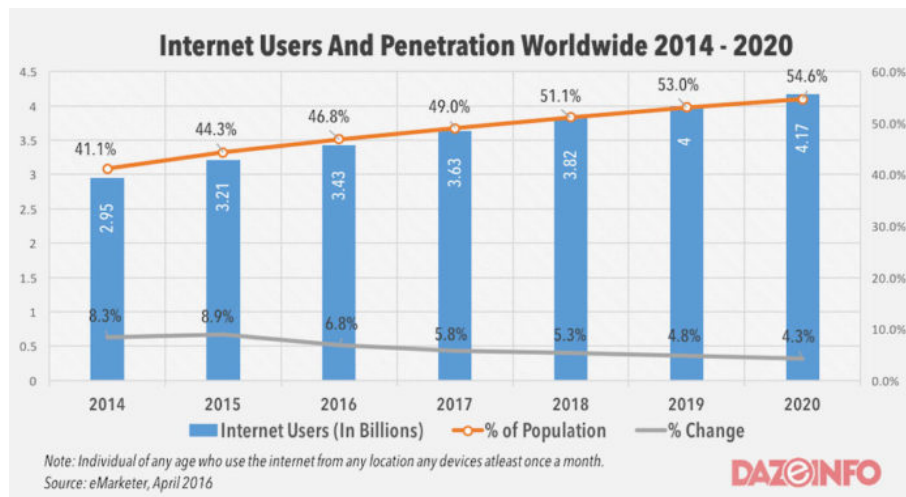


Figura 1: Predicción del número de usuarios en internet 2016-2020

En el Foro Económico Mundial de 2016 [2] 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, inteligencia artificial. Sin duda será 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

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. 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. La explicación más obvia para este hecho se fundamenta 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(creatividad, impredecibilidad) e 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 apuntan que dentro de 35 años, alrededor del 60 % de los trabajos estarán automatizados, con el impacto que eso conlleva para los diferentes trabajadores, que podrían perder puestos de trabajo en favor de profesionales del software que serán necesarios para afrontar el proceso de transición, y posterior mantenimiento de esta digitalización.

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

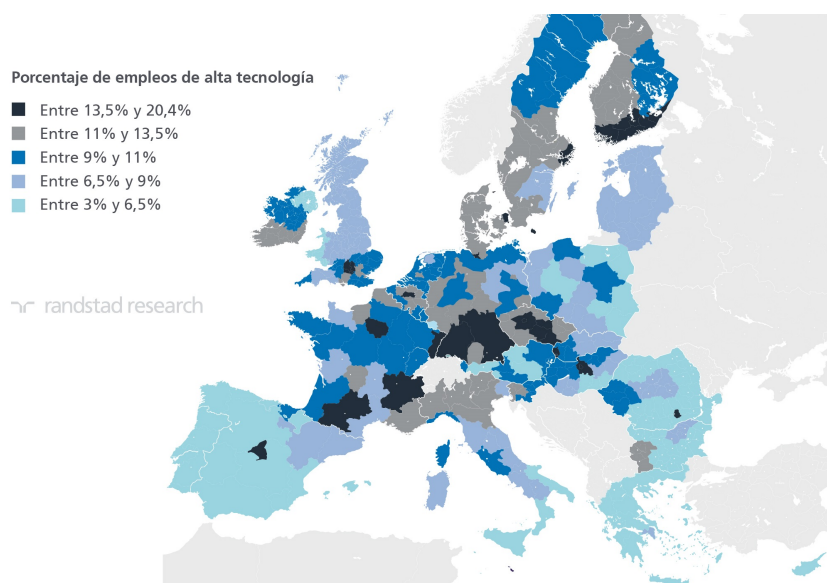


Figura 2: Empleos de alta tecnología en Europa

1.2. Software en robótica

Construir un robot, supone la interacción de muchos componentes entre sí, a su vez formados por piezas más pequeñas. El desarrollo de software para robótica debe tener controlado 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 1ª ley.
- Un robot debe proteger su propia existencia en la medida en que esta protección no entre en conflicto con la 1ª o la 2ª Ley.

Una vez están claras las bases para comenzar a trabajar en los proyectos de este área, es necesaria 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. En robótica, este software está diseñado para ocultar parte de la complejidad de bajo nivel, como la comunicación entre los distintos componentes, y que el usuario pueda diseñar y desarrollar sus soluciones de forma más eficiente. Estos son algunos de los más importantes, y sus principales objetivos:

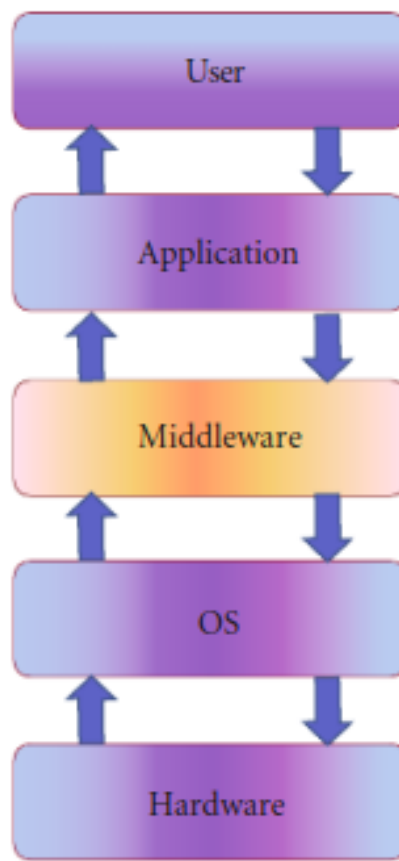


Figura 3: Situación del middleware en una arquitectura software

- ROS: 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.

- JdeRobot: Enfocada en la robótica y la visión computacional, ofrece compatibilidad con ROS y una serie de herramientas y librerías para robots, escritas en C++, Python y JavaScript.
- Orocos: 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.
- LCM: Lightweight Communications and Marshalling tiene como principal objetivo ser diferencial en contextos de baja latencia, utilizando el protocolo UDP en sus mensajes.
- CLARAty: Promueve la reutilización de software, además de la portabilidad, modularidad y flexibilidad. Surge de la colaboración entre varias instituciones, entre ellas la NASA.
- MRDS: Microsoft Robotics Developer Studio fue el primer intento de la compañía estadounidense por entrar de lleno en el mundo de la robótica. Su lanzamiento fue poco antes 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 dónde 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 muy 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?. 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. Ello por un lado requiere cambios legislativos, además de 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, algunas tan necesarias para otros aspectos profesionales y vitales como el trabajo en equipo, el pensamiento crítico, la creatividad y la resolución de problemas.

1.3.2. Robótica al alcance de todos

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 JdeRobotAcademy, 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. Otra opción si se desea aprender de manera más práctica y temprana es la de utilizar algunos de los robots educativos que distintas empresas han lanzado en los últimos años para lo más pequeños. Por ejemplo:

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



Figura 4: Sphero SPRK+ y LEGO mindstorms



Figura 5: Programando un ritmo musical mediante bloques

1.3.3. JdeRobot-Academy

Dentro del proyecto JdeRobot, encontramos un conjunto de prácticas enfocadas al aprendizaje de distintos ámbitos relacionados con la robótica a través de código python, compatibilidad con ROS y el uso del simulador Gazebo. Cada práctica enfrenta un problema típico diferente, donde 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 a que toda la configuración de la estructura y controladores del robot están definidos.

EL ejercicio de visión color filter, permite configurar un filtro de color para realizar un filtrado sobre un objeto en las imágenes proporcionadas por un fichero de vídeo. Por último, se debe detectar el movimiento de dicho objeto. Utiliza ICE como protocolo de comunicaciones y OpenCV para manejar las imágenes recibidas.

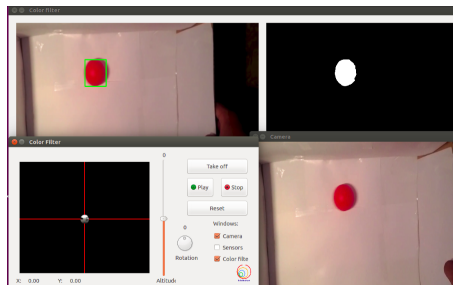


Figura 6: Filtro de color de la práctica color filter

En la práctica Drone cat&mouse dos drones deben jugar al gato y el ratón. Uno de ellos debe moverse de forma impredecible mientras el otro debe detectar el movimiento del otro y perseguirle, haciendo uso de las cámaras que lleva integradas y el procesamiento en tiempo real de las mismas.

Entrando en el terreno de la autolocalización y aprovechando la potencia de los sensores de profundidad, la práctica Vacuum-cleaner with visualSLAM permite simular el comportamiento de un robot aspirador de limpieza, detectando las partes de la casa por las que ha pasado y completando de forma eficiente el trabajo programado.

También hay ejemplos relacionados con los coches autónomos, como Visual follow-line

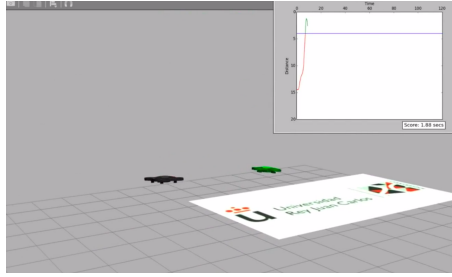


Figura 7: Persecución de drones en cat&mouse

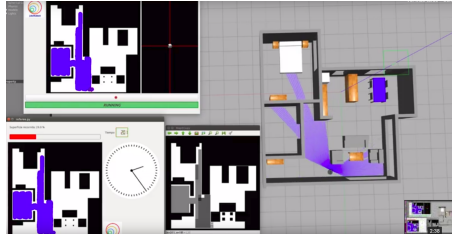


Figura 8: Mapeo generado en tiempo real en la práctica de la aspiradora

behavior on a Formula1 que permite a los estudiantes programar un Formula 1 que siga la línea roja en el centro de la carretera para dar una vuelta al circuito.

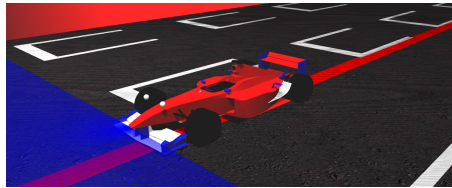


Figura 9: Mundo Gazebo para la práctica de conducción con Fórmula 1

Capítulo 2

Objetivos

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

2.1. Objetivos

El objetivo principal de este trabajo consiste en añadir una nueva práctica al entorno educativo Unibotics de JdeRobot. Dicha práctica tiene como meta profundizar en la planificación de trayectorias utilizando MoveIt!, el framework de ROS enfocado a la robótica que ofrece herramientas para gestionar movimientos, manipulación, control, físicas y navegación. Además, se deben analizar y filtrar imágenes para detectar objetos y colores, para lo que será necesario conocer algunas de las posibilidades que ofrece la librería OpenCV. Para obtener la solución final, será necesario controlar los movimientos de un robot PR2, el procesamiento de imágenes en tiempo real, la creación de modelos y mundos en el simulador Gazebo y la planificación y ejecución de trayectorias evitando obstáculos. Se ofrecerá al destinatario de la práctica educativa una plantilla donde podrá añadir su código, y que le abstraerá de los detalles de implementación más complejos, a los que podrá acceder posteriormente si desea profundizar en alguna parte. De esta forma el estudiante se podrá centrar en su desarrollo sin distraerse con problemas que ya han sido contemplados y resueltos, tomando contacto con la programación en robótica utilizando lenguaje python y conociendo como funciona MoveIt!, a través de una práctica completa cuyo eje es la planificación. El software debe realizar acciones en tiempo real, con valores aceptables para las ejecuciones en el simulador. Por último, debe ser capaz de obtener trayectorias válidas en situaciones con varios obstáculos.

2.2. Requisitos

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

- El sistema operativo utilizado es Ubuntu 16.04 LTS.

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

2.3. Metodología

La metodología de trabajo se ha basado en el modelo en espiral propuesto en 1986 por el ingeniero Barry Boehm. 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, 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 consiste en anticipar errores, siendo éste uno de los principales ahorros de costes que podemos encontrar en un proyecto software.
- Planificación. Se revisa el estado de los objetivos y se definen las tareas a llevar a cabo con los recursos disponibles para la iteración en curso.

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

2.4. Plan de acción

Al enfrentar un proyecto de características técnicas, y marcar unos determinados objetivos, se debe marcar un plan de acción que determine cómo alcanzar dichos objetivos, y de qué

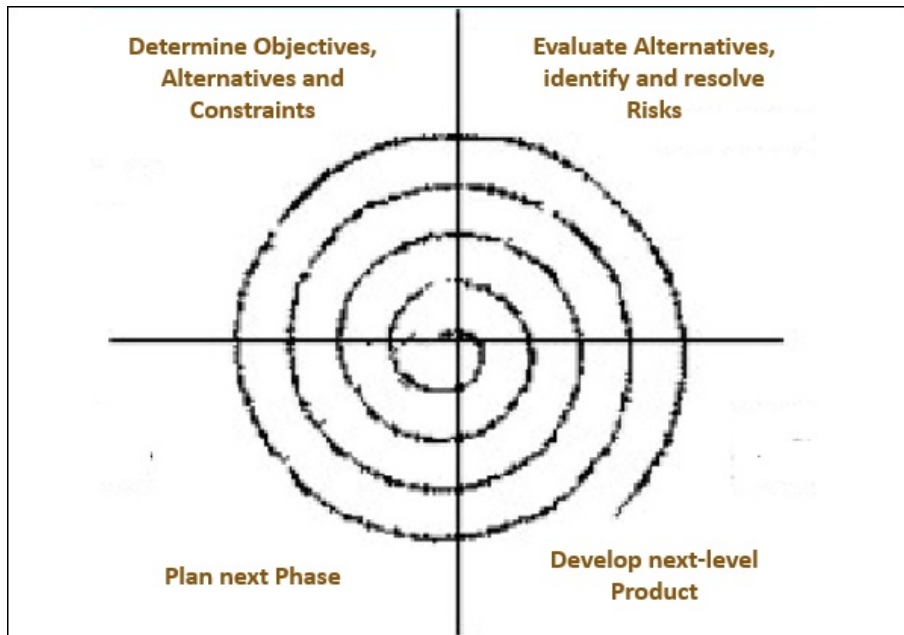


Figura 10: Representación gráfica del modelo en espiral para desarrollo software

manera se lograrán. Para completar la práctica de planificación, se han requerido los siguientes pasos:

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

Capítulo 3

Infraestructura

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

3.1. PR2

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

3.1.1. Microsoft Kinect

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

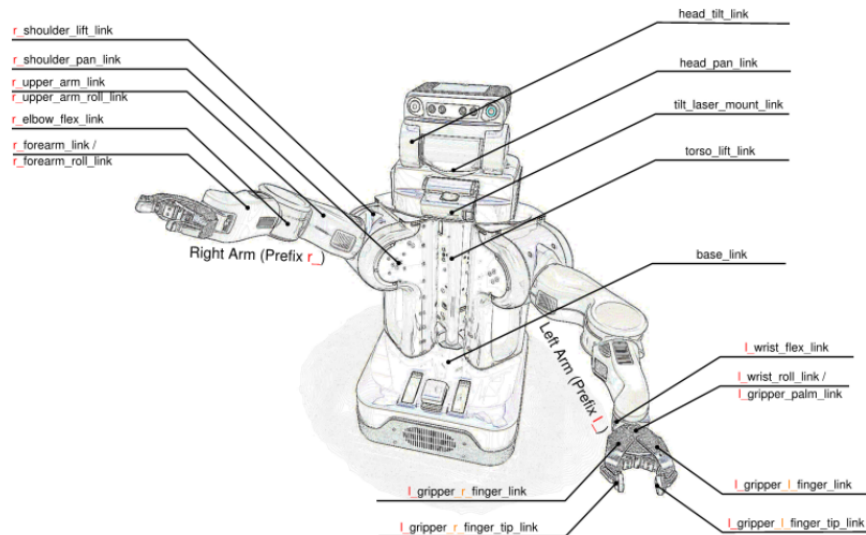


Figura 11: Estructura del robot PR2



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

3.2. Gazebo

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

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

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

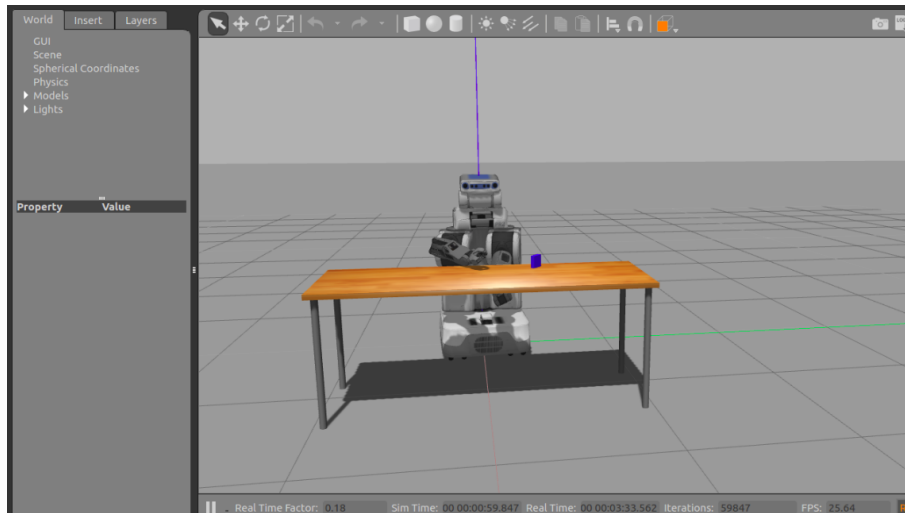


Figura 13: Mundo sencillo en Gazebo con un robot PR2, una mesa, y un objeto

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

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

La versión utilizada para este proyecto es Gazebo7.

3.3. JdeRobot

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

3.4. ROS Kinetic

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

```
~$ roscore
```

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

```
~$ rosrun turtlesim turtlesim_node
```

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

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

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

```
~$ rostopic echo /turtle1/cmd_vel
```

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



```
$ rosrun rqt_graph rqt_graph
```

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

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


3.5. ICE

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

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

3.6. C++

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

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

3.7. Python

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

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

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

3.8. OpenCV

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

3.9. MoveIt!: Motion Planning Framework

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

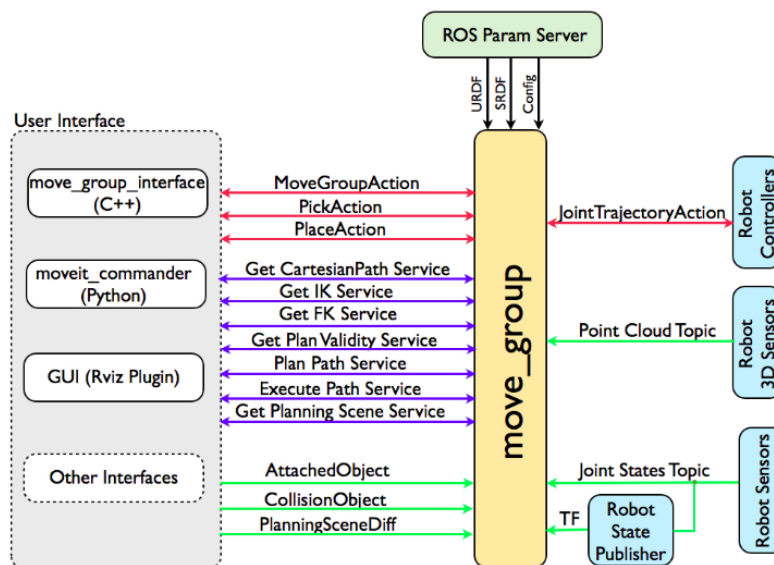


Figura 14: Resumen esquematizado de las posibles interacciones que ofrece `move_group`

`move_group` necesita varios parámetros de entrada:

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

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

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

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

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

Capítulo 4

Aplicación con brazo robótico y planificación de trayectorias

En este capítulo se describirán en detalle tanto el objetivo como el desarrollo técnico de la práctica "Movimiento de un brazo robótico" para el entorno Unibotics. Incluye el enunciado de la misma, la infraestructura necesaria y la plantilla de código python a utilizar.

4.1. Enunciado

En este capítulo abordaremos el desarrollo de la solución escogida como objetivo del proyecto. Incluye la planificación de trayectorias con obstáculos en un entorno simulado, el reconocimiento de objetos analizando espacios de color, y la planificación y ejecución de movimientos.

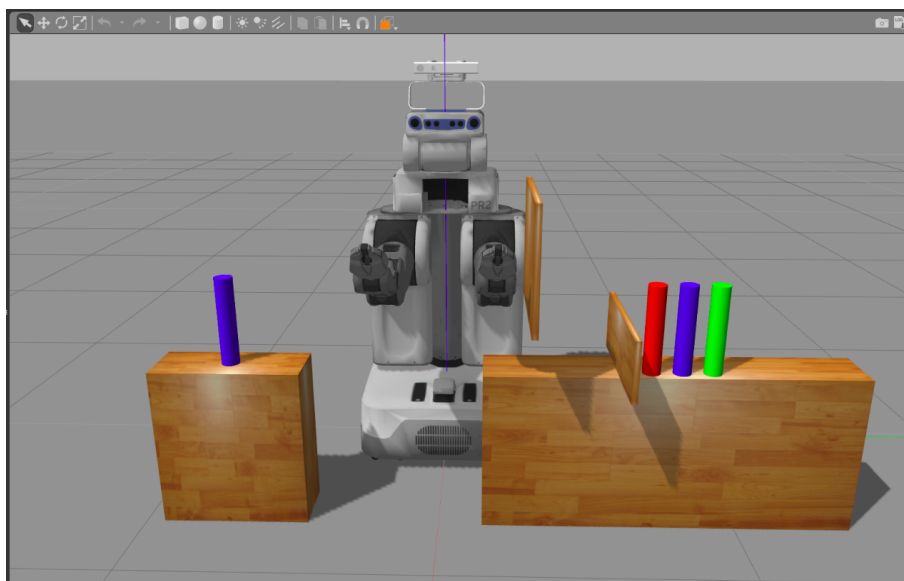


Figura 15: Mundo Gazebo que incluye todo lo necesario para el desarrollo de la práctica: zonas, objetos, robot y obstáculos.

El escenario de la práctica consiste en un mundo Gazebo(Fig. 15) que incluye dos zonas de acción(A y B) 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 B tendrán los colores rojo, verde y azul, y el de la zona A tiene únicamente uno de los colores mencionados(azul por defecto). Una vez presentado el mundo, a continuación se relatará a grandes rasgos cuál es el funcionamiento esperado para la práctica.

En primer lugar, el robot PR2 moverá la cabeza y observará la Zona A. 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 B, será capaz de planificar ,y posteriormente ejecutar, una trayectoria que le sitúe frente al objeto en la zona B cuyo color coincida con el observado en la zona A. Para ello será necesario gestionar la planificación de trayectorias evitando los obstáculos necesarios y, por último, realizar un movimiento de derribo.

4.2. Infraestructura

Esta sección tiene como objetivo mostrar los elementos necesarios para lograr una práctica completa de planificación. Entre ellos se incluyen el mundo creado en Gazebo y todo lo que ello contiene, como el robot PR2 y los plugins utilizados para el procesamiento de imagen.

4.2.1. Mundo Gazebo

En primer lugar, ha sido necesaria la creación de un mundo en Gazebo que contiene los objetos a detectar y empujar, las dos zonas de acción, los obstáculos, el robot PR2 y la cámara Kinect. La cámara Kinect se incorpora en la parte superior de la cabeza del robot, de forma que se encuentran completamente integrados y podemos controlar la visión del robot a través de movimientos en su parte superior. Para lanzar este entorno, se debe ejecutar un fichero .launch [1], que pondrá en funcionamiento los modelos necesarios para las dos zonas de referencia, los modelos de los objetos y obstáculos, y el robot PR2 con todos sus controladores. La cámara Kinect integrada se pasa como parámetro, y está configurada para aparecer por defecto.

Modelo PR2

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 joints, 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 [3] 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. Por último, las primeras fases de pruebas con reconocimiento de imagen en

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

```
<launch>
  <!-- start up world -->
  <arg name="gui" default="true"/>
  <arg name="headless" default="false" />
  <arg name="debug" default="false" />
  <arg name="paused" default="true"/>
  <arg name="KINECT2" default="$(optenv KINECT2 true)" />
  <include file="$(find gazebo_ros)/launch/empty_world.launch">
    <arg name="world_name" value="worlds/tfg_world.world"/>
    <arg name="gui" value="$(arg gui)" />
    <arg name="headless" value="$(arg headless)" />
    <arg name="paused" value="$(arg paused)" />
    <arg name="debug" value="$(arg debug)" />
    <arg name="use_sim_time" value="true" />
  </include>
  <!-- start pr2 robot with Kinect camera -->
  <include file="$(find pr2_gazebo)/launch/pr2.launch">
    <arg name="KINECT2" value="$(arg KINECT2)" />
  </include>
</launch>
```

el entorno creado, evidenciaban problemas a la hora de detectar los colores a causa del sensor láser incluido por defecto en el modelo. Éste generaba ruido en las imágenes RGB obtenidas dificultando la caracterización de los objetos, motivo por el cual también está desactivado en los paquetes generados.

Para ello, es necesario conocer el concepto de overlaying [9], utilizado en ROS para hablar de modificaciones en el espacio de trabajo [**wstool**] del usuario sobre los paquetes originales descargados de los repositorios oficiales. De esta manera podemos hacer que *ros* priorice nuestros propios repositorios, donde se habrán modificado los ficheros necesarios para lograr un determinado funcionamiento, como por ejemplo la desactivación del sensor láser.

Modelos adicionales

Se ha desarrollado un fichero .world[2] que define los modelos necesarios para la consecución de la práctica. Para cada uno de los elementos se establecen su posición, como será su visualización, sus criterios de colisión y sus dinámicas. Para ello se utiliza el formato SDF [7], open-source y basado en XML y que permite describir modelos de cualquier complejidad a través de sencillas etiquetas, desde esferas o prismas, hasta robots completos.

Algoritmo 2 Fragmento del fichero .world que define el modelo de la zona origen 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>
```

4.3. Plantilla

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

necesarios para la planificación de trayectorias.

Existen dos ficheros python, uno llamado `final_prototype_template_ready.py` y otro llamado `template.py`[3]. El primero contiene la mayor parte de la complejidad de la práctica, y tiene como objetivo abstraer al usuario final de los entresijos necesarios para hacer funcionar un brazo robótico con MoveIt! y gestionar imágenes con OpenCV, simplificando dichas tareas a través de algunas funciones sencillas. De esta manera, el alumno tendrá disponibles las siguientes interfaces desde el fichero `template.py`:

- `look_at_object(x,y,z)`: Mueve la cabeza del robot PR2 configurando y enviando un mensaje `PointHeadGoal` con los parámetros recibidos.
- `Subscriber`: es el encargado de llamar al callback para que realice acciones cuando se recibe una imagen en el topic.
- `convert_to_cv2(msg)`: convierte el mensaje recibido en el callback en una imagen preparada para trabajar con OpenCV.
- `convert_to_hsv(img)`: convierte una imagen BGR al espacio de color HSV.
- `save_image(img)`: guarda la imagen en formato .jpeg en un directorio local.
- `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.
- `start_planning()`: instancia los objetos necesarios para obtener y modificar valores del robot, la escena y la planificación. Añade los objetos 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.
- `move_to_goal(color)`: planifica y ejecuta un movimiento al objeto destino del color pasado como parámetro.

Algoritmo 3 Plantilla simplificada para la práctica de planificación de trayectorias

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

Capítulo 5

Solución de referencia

5.1. Procesamiento de imagen

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

5.1.1. Cámara Kinect y ROS topics

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

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

5.1.2. Librería OpenCV

El sensor Kinect situado sobre la cabeza del robot ofrece imágenes RGB-D. Utilizando la librería open source OpenCV, es posible acceder vía código python a un enorme conjunto de posibilidades relacionadas con el procesamiento de imagen. [6]

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

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

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

<pre>cv2_img = bridge.imgmsg_to_cv2(msg, "bgr8")</pre>
--

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

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

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

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

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

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

5.2. Planificación de trayectorias con MoveIt

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

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

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

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

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

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

5.2.1. MoveGroupCommander

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

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

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

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

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

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

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

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

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

- pick
- place
- set_support_surface_name

5.2.2. Controladores

Para que la planificación y ejecución de los movimientos del robot sean adecuadas, necesitamos unos controladores bien configurados para cada una de las articulaciones del brazo. [1] Para ello se especifican los nombres de cada una de ellas de la misma manera que se configuraron al crear el paquete de MoveIt! del robot PR2, cuyos pasos son intuitivos si seguimos los pasos del MoveIt! Setup Assistant desde los tutoriales oficiales. [5]

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

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

5.2.3. Planificador y configuración

Las posibilidades de configuración que ofrece MoveIt! son ingentes. Entre ellas se encuentran las posibilidades de seleccionar distintos planificadores. Podemos encontrar desde Open Motion Planning Library(OMPL), que está completamnete 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. [4] El objetivo de cualquier planificación en robótica consiste en resolver el problema de ir de un lado a otro sin violar las restricciones establecidas. OMPL es un sistema basado en muestreo, que consiste en generar aleatoriamente una serie de posiciones válidas en el espacio en función de los parámetros de entrada y conectarlas entre sí.

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

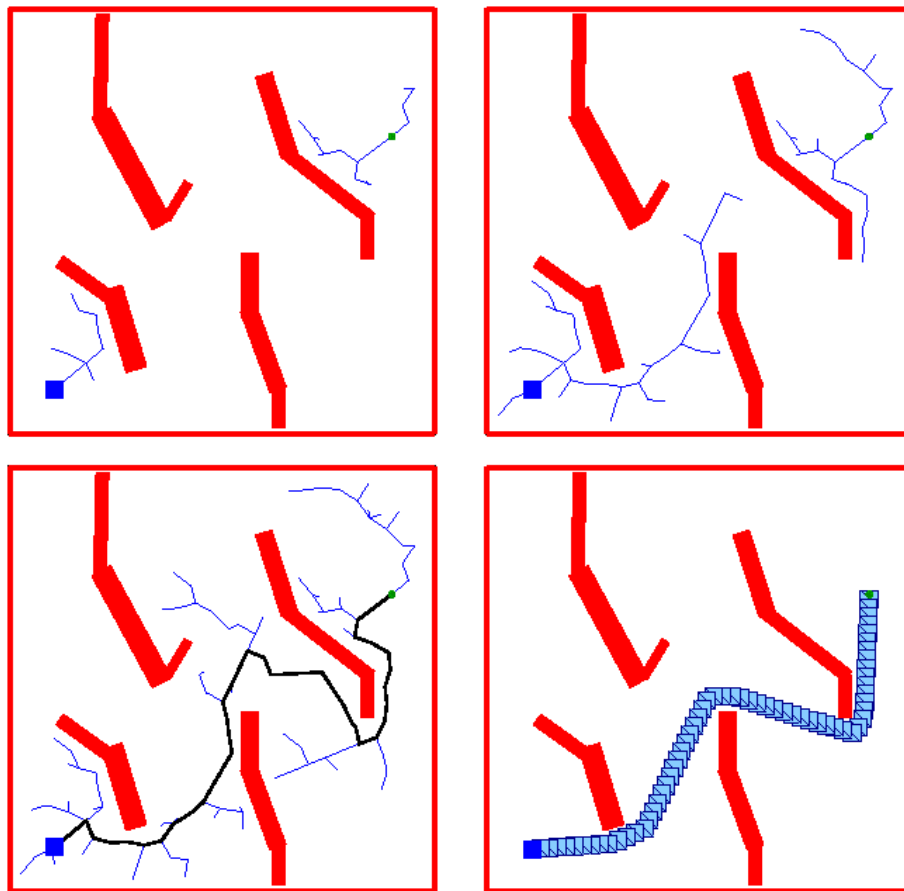


Figura 16: Explicación gráfica del algoritmo RRT Connect utilizado.

Aplicado a nuestra práctica, el algoritmo consiste en comenzar a generar de forma simultánea un "árbol" desde el inicio y el destino del movimiento. El árbol inicio se irá expandiendo por posiciones cercanas aleatorias que 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.

5.3. Experimentación

La ejecución típica [8] de la práctica requiere los siguientes pasos:

- 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 pr2_planning_execution_tfg.launch
```

- Lanzar el script template.py.

```
roslaunch moveit_tutorials template.py
```

Una vez realizados los pasos anteriores, podremos observar en el simulador Gazebo la práctica de planificación y ejecución de trayectorias con MoveIt!.

Capítulo 6

Conclusiones

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

6.1. Conclusiones

El objetivo formal propuesto al comienzo del desarrollo del proyecto ha sido alcanzado con éxito, generando una práctica completa de planificación de trayectorias utilizando herramientas modernas y extendidas en el mundo robótico. El desarrollo de esta práctica ha supuesto importantes retos que enfrentar. En primer lugar, enfrentarse a un área de conocimiento nueva hasta ser capaz de aprovechar las posibilidades que ofrece y completar una solución técnica. Por el camino, lo más importante, ha sido necesario aprender cada semana para resolver los problemas del día a día hasta alcanzar los objetivos planteados, mejorando así una de las facetas clave de la ingeniería y adquiriendo conocimientos nuevos. Dicho crecimiento ha surgido a partir de las bases obtenidas durante el grado, tanto teóricas como prácticas y soft skills adquiridas durante la carrera. Para poder comenzar a trabajar con garantías, han sido necesarios los conocimientos y experiencia adquiridos en las distintas asignaturas relacionadas con la programación y el software, así como el tratamiento de imágenes o los fundamentos físicos. Durante el proceso, han ido acompañadas de otras habilidades como el rendimiento bajo presión, la comunicación, la adaptación al cambio, la motivación o el pensamiento crítico. La metodología de trabajo escogida ha permitido recibir feedback de forma dinámica sobre el progreso realizado, para conseguir adaptar el proyecto a las realidades que se enfrentan y afrontar las dificultades con garantías de éxito. La práctica de planificación de trayectorias, tuvo como primer reto la toma de contacto con la robótica a través de algunas de las prácticas más sencillas de JdeRobot. Enseguida se consideró necesario el aprendizaje de un nuevo lenguaje de programación(C++), tanto para poder comprender el código ya existente en los repositorios en este lenguaje, como para adaptar algunos tutoriales que sólo estaban disponibles en C++ al lenguaje elegido, python. Además se realizaron algunos tutoriales sobre ICE, el protocolo de comunicaciones utilizando en las prácticas comentadas. El siguiente paso consistió en descubrir el simulador Gazebo, que posteriormente tendría un papel fundamental para la solución final. Una vez resueltos estos primeros pasos, se comenzó a trabajar con ROS, acercándose a él a través de los tutoriales oficiales. Con los conocimientos sobre nodos,

topics o creación de espacios de trabajo, fue suficiente para afrontar el núcleo de este proyecto: MoveIt!. Para comenzar se trabajó con el proceso de configuración de un robot, en este caso el modelo PR2, además de utilizar como apoyo el visualizador RVIZ y viendo efectos prácticos desde el primer momento. Ya sentadas las bases del proyecto, fue posible considerar toda la solución en su conjunto y centrar los esfuerzos en profundizar en la planificación de trayectorias y en otras herramientas necesarias como la librería OpenCV. La última parte del trabajo consistió en refinar detalles de los diferentes componentes de la práctica. Por un lado se añadieron algunos obstáculos al mundo gazebo y se configuró un movimiento inicial en la cabeza del robot para mirar a la zona de origen, para terminar mejorando la mecánica de movimiento ajustando parámetros como restricciones en la articulación de la muñeca del robot, los tiempos e intentos de la planificación o el plugin que controla la cinemática del ejercicio. Además del apartado puramente técnico, la motivación ha permitido superar el reto de completar un proyecto de ingeniería actual e innovador utilizando metodología de desarrollo de software extendida laboralmente. Ello ha permitido adquirir conocimientos y habilidades útiles para el futuro laboral, como pueden ser ampliar los conocimientos de programación y tratamiento de imágenes, conocer el mundo de la robótica, diseñar modelos y arquitecturas, optimizar costes o trabajar en una solución integrada por distintos componentes que se comunican entre sí.

6.2. Trabajos futuros

La solución aquí propuesta puede abrir la puerta a algunas líneas de trabajo futuras que, o bien supongan avances para la práctica realizada, o bien supongan nuevas prácticas que hagan uso de los conceptos y código desarrollados. Un primer avance podría ser hacer uso de las imágenes de profundidad ya aportadas por la cámara Kinect para calcular la posición relativa de los objetos en la zona destino, y utilizar dichas posiciones para calcular el destino de la planificación. Esta funcionalidad podría completarse calculando también la localización y tamaño de los obstáculos y objetos para tenerlos en cuenta al planificar. Otra idea para profundizar en el uso de MoveIt!, sería añadir una tercera zona de acción en la que el robot deposite el objeto del color correspondiente, a través de maniobras de pick&place. Para ello sería necesario aprender a manejar los mensajes y dispositivos de agarre, para lo que los tutoriales oficiales de MoveIt! pueden ser de gran ayuda. Por último, podría ser necesario adaptar el código a futuras versiones de ROS que introduzcan mejoras importantes relacionadas con planificación, movimientos del robot o nueva funcionalidad. De manera análoga, igual que el robot PR2 se escogió por motivos de diseño y compatibilidad, afrontar algunas mejoras podría requerir migrar la solución a un nuevo robot. Si el trabajo futuro se centrara en el estudio teórico o la obtención de métricas relacionadas con la planificación, una opción sería profundizar en los algoritmos de planificación disponibles en MoveIt!, o incluso generar los propios para estudiar los casos de mayor rendimiento para cada uno de ellos.

Bibliografía

- [1] *Configure MoveIt! controllers*. URL: <http://www.theconstructsim.com/control-gazebo-simulated-robot-moveit-video-answer/> (vid. pág. 33).
- [2] *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).
- [3] *Gazebo camera plugins*. URL: http://gazebo.org/tutorials?tut=ros_gzplugins (vid. pág. 22).
- [4] *MoveIt! Planners*. URL: <https://moveit.ros.org/documentation/planners/> (vid. pág. 34).
- [5] *MoveIt! tutorials*. URL: http://docs.ros.org/kinetic/api/moveit_tutorials/html/index.html (vid. pág. 33).
- [6] *OpenCV tutorial*. URL: https://docs.opencv.org/3.4.1/df/d9d/tutorial_py_colorspaces.html (vid. pág. 27).
- [7] *SDF Format*. URL: <http://sdformat.org/spec> (vid. pág. 23).
- [8] *Video final TFG*. URL: <https://youtu.be/OU1iNR0tzyA> (vid. pág. 35).
- [9] *Workspace overlaying in ROS*. URL: http://wiki.ros.org/catkin/Tutorials/workspace_overlaying (vid. pág. 23).