

ESCUELA TÉCNICA SUPERIOR DE INGENIERÍA INFORMÁTICA
GRADO EN INGENIERÍA DE COMPUTADORES

**DESARROLLO DE UN ENTORNO DE PROGRAMACIÓN
PARA UN ROBOT SIMULADO TURTLEBOT-2 CON BRAZO
MANIPULADOR WIDOWX MEDIANTE LA CONEXIÓN DE
V-REP Y MATLAB**

**DEVELOPMENT OF A PROGRAMMING ENVIRONMENT
FOR A SIMULATED TURTLEBOT-2 ROBOT WITH A
WIDOWX MANIPULATOR ARM THROUGH THE
CONNECTION OF V-REP AND MATLAB**

Realizado por

Iván Fernández Vega

Tutorizado por

Ana María Cruz Martín

Juan Antonio Fernández Madrigal

Departamento

Ingeniería de Sistemas y Automática

UNIVERSIDAD DE MÁLAGA

MÁLAGA, Junio de 2016

Fecha defensa:

El Secretario del Tribunal

Resumen

Resumen en español

El objetivo principal de este Trabajo de Fin de Grado ha sido el de implementar un entorno de programación para un robot móvil Turtlebot-2 dotado de un brazo articulado WidowX. Para ello, se ha modelado este conjunto robótico en el simulador V-REP a partir de las descripciones físicas de los mismos, y se han ajustado diversos parámetros para que el comportamiento del robot simulado sea lo más parecido al del robot real.

Para implementar las aplicaciones que controlen nuestro conjunto robótico hemos elegido MATLAB. Por tanto, es necesario establecer una conexión entre MATLAB y V-REP. Puesto que V-REP dispone de una API remota, ha sido necesario diseñar una *toolbox* que permita a un usuario potencial crear las aplicaciones de manera eficiente, sin tener que conocer las funciones propias de esa API.

Esta *toolbox* hace uso de la *Robotics System Toolbox* de MATLAB, y tiene una apariencia muy similar a la misma. De esta manera, la curva de aprendizaje es bastante suave, y es posible usar el mismo programa tanto sobre el simulador como sobre el robot real. Para ello, simplemente hay que cambiar un parámetro en el programa.

Por último, se han implementado dos aplicaciones completas: una de ellas utiliza el brazo para mover un objeto en un escenario y la otra utiliza un algoritmo de navegación reactiva para mover el conjunto robótico de un lugar a otro.

Palabras claves

Robótica, V-REP, MATLAB, WidowX, Turtlebot, Kinect, Hokuyo, Brazo robótico, Robot móvil, Navegación reactiva.

Abstract

The main objective of this Bachelor thesis is to implement a programming environment for a Turtlebot-2 mobile robot equipped with an articulated WidowX arm. To do that, this robotic assembly has been modeled in the V-REP simulator from the physical descriptions

and various parameters have been set for the behavior of the simulated robot to be as close as possible to the real robot.

For developing applications that control our robotic assembly we have chosen MATLAB. It is therefore necessary to establish a connection between MATLAB and V-REP. Since V-REP has a remote API, designed a toolbox that allows a user to create potential applications efficiently, without having to know the functions of the API.

This toolbox uses the MATLAB Robotics System Toolbox, and it has a very similar appearance. Thus, the learning curve is quite smooth, and you can run the same program on the simulator and on the actual robot. To do this, simply change a parameter.

Finally, we have implemented two complete applications: one of them uses the arm to move an object in a scene and the other one uses a reactive navigation algorithm to move the robotic assembly from one place to another.

Keywords

Robotics, V-REP, MATLAB, WidowX, Turtlebot, Kinect, Hokuyo, Robotic arm, Mobile robot, Reactive Navigation.

Agradecimientos

En primer lugar, me gustaría mostrar mi agradecimiento a todas aquellas personas que han colaborado de una forma u otra en que la realización de este trabajo haya sido posible.

Por un lado, a mis tutores, Ana y Juan Antonio, que han estado siempre ahí para solucionar aquellas dudas que han ido surgiendo durante la elaboración del trabajo y han sabido guiarme por el camino correcto en su realización.

Por otro lado, a mis padres y mi familia, ya que sin ellos no hubiera llegado hasta aquí.

Tampoco quiero olvidarme de los integrantes del proyecto CRUMB del Departamento de Ingeniería de Sistemas y Automática, especialmente de Marina y Ángel, ni de Robotnik, por su aportación de la descripción del modelo de WidowX.

Índice general

Resumen	I
Agradecimientos	III
1. Introducción	1
1.1. Descripción del trabajo	1
1.2. Objetivos	3
1.3. Fases del TFG	4
1.4. Métodos	4
1.5. Medios materiales	5
1.6. Organización de la memoria	5
2. Herramientas utilizadas	7
2.1. Conjunto Robótico	7
2.2. V-REP	11
2.3. MATLAB	12
2.4. Lua y ROS	13
3. Modelado del robot	15
3.1. Analizando el problema	15
3.2. Incluyendo el robot en V-REP	17
3.3. Implementando los sensores	25
3.3.1. Kinect	25
3.3.2. Giroscopio	27
3.3.3. <i>Cliffs</i>	28
3.3.4. <i>Bumpers</i>	29
3.3.5. <i>Wheels Drop</i>	31
3.3.6. Láser	32
3.4. Añadiendo las texturas	34
3.5. Importando el modelo de WidowX	36
3.6. Generando los <i>scripts</i> del modelo	40
3.6.1. Turtlebot	41

3.6.2. WidowX	56
3.7. Otros detalles del modelo	61
4. Diseño de la <i>toolbox</i>	65
4.1. ¿Por dónde empezar?	65
4.2. Implementando las funciones	68
4.2.1. <i>rosinit</i>	68
4.2.2. <i>rospublisher</i> y <i>rossubscriber</i>	70
4.2.3. <i>rosmessage</i>	70
4.2.4. <i>send</i>	72
4.2.5. <i>receive</i>	75
4.2.6. <i>rossvcclient</i>	81
4.2.7. <i>call</i>	81
4.2.8. <i>rostopic</i>	84
4.2.9. <i>rosservice</i>	85
4.2.10. <i>rosshutdown</i>	86
5. Resultados	89
5.1. Tiempos de respuesta	89
5.2. Primera aplicación	91
5.3. Segunda aplicación	98
6. Conclusiones y trabajos futuros	105
A. Manual del Usuario	107
A.1. Requisitos	107
A.2. Posibilidades de nuestra <i>toolbox</i>	108
A.3. Preparando el entorno	111
A.4. Una primera aplicación	114
A.5. Simulando nuestra primera aplicación	119
A.6. Mejorando el rendimiento	122
B. Manual del Desarrollador	123
B.1. Añadir un sensor al modelo	123
B.2. Añadir un <i>topic</i> en MATLAB	124
B.3. Añadir un actuador al modelo	126
B.4. Añadir un servicio en MATLAB	128
B.5. Señales en V-REP	129

Capítulo 1

Introducción

1.1. Descripción del trabajo

El Departamento de Ingeniería de Sistemas y Automática de la Universidad de Málaga dispone de un ejemplar del kit robótico móvil Turtlebot-2 [1], dotado de un brazo articulado WidowX [2].

Turtlebot-2 es un robot móvil formado por una base Kobuki [3], un *netbook* y un sensor Kinect [4]. En cuanto a la base, cabe destacar su alta precisión en lo que a odometría se refiere. Consta de varios sensores: de inclinación, de caída de la rueda, *bumpers*, etcétera. Otro de los componentes fundamentales del robot móvil es el *netbook*; será el encargado de ejecutar ROS (Robot Operating System) [5] y tomará las decisiones que produzcan el comportamiento del robot. Por último, tenemos el sensor Kinect. Permite añadir visión en 3D al robot, que será clave para la navegación del mismo, pudiendo detectar distintos objetos del entorno para así interactuar con ellos. Turtlebot-2 tiene disponibles *drivers* para ROS y para Windows/Linux.

Por otro lado, WidowX es un brazo robótico articulado, de unos 40 cm de alcance horizontal y alrededor de 50 cm de alcance vertical. Su base gira 360 grados. Sin lugar a dudas, añade al robot una capacidad más que interesante: la de agarrar objetos. La controladora del brazo es la ArbotiX-M [7], que además es compatible con el IDE de Arduino.

El principal objetivo de este Trabajo de Fin de Grado ha sido construir una herramienta de simulación tanto para fines de investigación como educativos. Para ello, hemos optado por el *software* V-REP [8]. Se trata de una plataforma de experimentación virtual para robots. Su filosofía es la de crear, componer y simular cualquier robot. De hecho, incluye bastantes modelos de robots comerciales y conocidos, por lo que es de gran ayuda para el desarrollo de aplicaciones robóticas. Soporta múltiples lenguajes de programación, y permite simular, entre otras muchas cosas, la dinámica de los cuerpos en el espacio tridimensional.

Por desgracia, V-REP no incluye a nuestro conjunto robótico. Por tanto, en este Tra-

bajo de Fin de Grado se ha modelado en dicho software, para así poder simularlo y desarrollar aplicaciones robóticas relativamente complejas. Esto ha conllevado tanto la inclusión del modelo 3-D del propio robot+brazo, coincidentes con los modelos cinemáticos teóricos de los mismos, como el de sus sensores. Para ello, hemos partido de las descripciones disponibles del robot y del brazo y las hemos adaptado al software de simulación. Además, gracias a la API remota que V-REP nos proporciona, vamos a poder conectarlo con MATLAB®¹.

La conexión con MATLAB es la segunda parte del trabajo propuesto, y propicia que el desarrollo de aplicaciones y el de actividades educativas sea menos complicado. El principal beneficio de usar MATLAB es que ya se emplea en otras asignaturas impartidas por el Departamento en diferentes titulaciones. La mayoría de los alumnos tienen al menos unas nociones básicas de este lenguaje matemático, por lo que la curva de aprendizaje se reduce notablemente.

Así pues, el beneficio logrado tras la realización de este proyecto es el de disponer de una herramienta de simulación, que puede usarse tanto con fines investigadores como académicos en las distintas asignaturas que imparte este departamento.

Con respecto a las tareas que el alumno ha realizado en este Trabajo de Fin de Grado, le han exigido conocimientos y habilidades adquiridas en varias de las asignaturas que conforman el plan de estudios que está cursando, tal y como se describe a continuación:

- Tareas relativas al Control de Sistemas [9], relacionadas con la asignatura Control por Computador, obligatoria en cuarto curso del Grado en Ingeniería de Computadores, y que ya ha sido cursada por el alumno.
- Tareas relativas a la Programación de Robots [10], relacionadas con la asignatura Programación de Robots, optativa para los tres Grados de Informática, y que ya ha sido cursada por el alumno.
- Tareas relativas al manejo de Sistemas de Tiempo Real [11], relacionadas con la asignatura Sistemas de Tiempo Real, obligatoria en tercer curso del Grado en Ingeniería de Computadores, y que ya ha sido cursada por el alumno.
- Tareas relativas a la Visión por Computador [12], relacionadas con la asignatura Visión por Computador, optativa para los tres Grados de Informática, y que ya ha sido cursada por el alumno.

A continuación se muestra un diagrama de una posible aplicación de este Trabajo de Fin de Grado (Figura 1.1). Por un lado, tenemos un PC ejecutando MATLAB, que será en el que implementaremos el programa que deseemos simular. Por otro lado, tenemos otro

¹MATLAB es una marca registrada de The MathWorks, Inc. A partir de ahora, vamos a obviar el símbolo (®) para evitar ser reiterativos, según consulta realizada a la Real Academia de la Lengua Española [15].

PC (que puede ser el mismo que el primero) ejecutando el simulador V-REP, conectado con MATLAB mediante una red de área local.

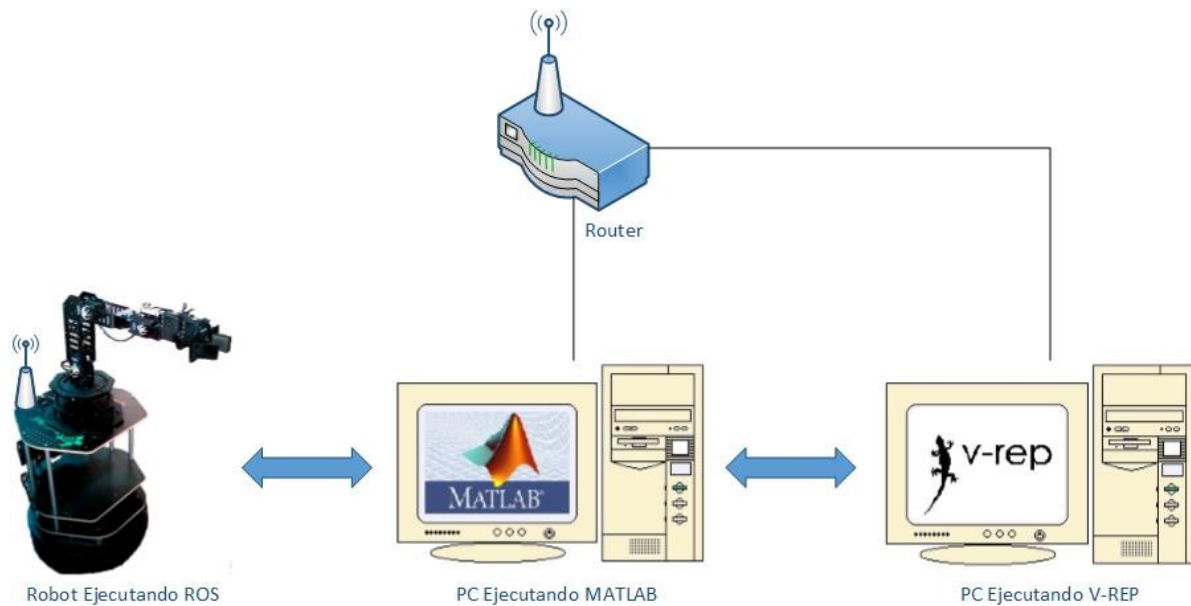


Figura 1.1: Diagrama de funcionamiento del software desarrollado.

Con el objetivo de que el programa generado en MATLAB sirva también para hacer pruebas con el robot real además de para la simulación en V-REP, hemos generado una *toolbox* en MATLAB cuya apariencia es muy similar a la de la *Robotics System Toolbox* [13], la cual permite comunicarnos directamente con el robot real, es decir, con ROS.

La *Robotics System Toolbox* está actualmente disponible en la licencia de MATLAB de la UMA, tanto para alumnos como para personal docente e investigador.

Además, si nuestra *toolbox* tiene la apariencia de la *Robotics System Toolbox*, la curva de aprendizaje para un usuario acostumbrado a ROS es bastante menos acusada. Obtenemos por un lado, la integración de un mismo programa tanto para la simulación como para el robot real, y por otro, la facilidad para depurar código y la gran cantidad de funciones que nos brinda MATLAB (por ejemplo, para Visión por Computador).

1.2. Objetivos

Los objetivos del presente Trabajo de Fin de Grado han sido los siguientes:

- Generar el modelo 3D, de la forma más completa y exhaustiva posible, del conjunto TurtleBot+brazo en el software V-REP, incluyendo tanto el modelado espacial como el cinemático.
- Analizar las posibilidades que el simulador V-REP nos proporciona, sobre todo desde el punto de vista de la API y de los múltiples lenguajes disponibles, prestando especial atención a MATLAB.

- Diseñar e implementar la conexión entre el *software* V-REP y MATLAB, así como la *toolbox* necesaria de este último para programar todos los componentes del robot. Esta *toolbox* con una apariencia similar a la *Robotics System Toolbox* que MATLAB proporciona, y hará uso de esta para que el mismo programa empleado para la simulación pueda ser ejecutado con el robot real.
- Crear una aplicación relativamente compleja que use la *toolbox* generada y demuestre su utilidad.

1.3. Fases del TFG

El trabajo propuesto se ha dividido en distintas fases:

1. Documentación sobre trabajos y *software* relacionados [6 horas].
2. Documentación y estudio del funcionamiento y características del conjunto robótico formado por Turtlebot-2 y WidowX [10 horas].
3. Modelado de dicha estructura física e inclusión del modelo cinemático del robot en el *software* de simulación V-REP [60 horas].
4. Análisis de las posibilidades del simulador, prestando especial atención a la API [10 horas].
5. Diseño e implementación del código MATLAB necesario para conectar dicho *software* al *software* de simulación robótico V-REP, esto es, implementar la *toolbox* [95 horas].
6. Pruebas finales de comunicación entre ambos entornos, y entre MATLAB y el robot real [15 horas].
7. Diseño y pruebas de una aplicación completa [20 horas].
8. Elaboración de la memoria [80 horas].

1.4. Métodos

Para la primera fase del Trabajo, correspondiente al modelado en el simulador, ha sido necesario tomar ciertas medidas del robot real. Esto incluye tanto la realización de mediciones de los sensores, como un estudio del comportamiento del robot (por ejemplo, cuánto tiempo tarda en acelerar desde reposo), para que el comportamiento simulado se parezca lo máximo posible al real.

Por otro lado, en la segunda parte, correspondiente a la conexión entre MATLAB y V-REP, ha sido necesario crear una *toolbox*. Para ello, hemos creado una clase en MATLAB

que hace uso tanto de la API remota de V-REP como de la *Robotics System Toolbox*. De esta forma, el hecho de estar trabajando con el robot simulado o con el robot real será transparente para el usuario, que podrá usar el mismo programa para ambos entornos.

1.5. Medios materiales

Hardware disponible:

- Conjunto robótico formado por el robot móvil Turtlebot-2 y el brazo WidowX.
- Ordenador de laboratorio [i7 con 8GBytes de RAM, tarjeta gráfica integrada].

Software disponible:

- Sistema operativo Windows 7 de 64 bits.
- V-REP PRO EDU V3.3.0
- MATLAB R2016a, con la Robotics System Toolbox instalada.
- Oracle VirtualBox con máquina virtual de ROS Indigo.

1.6. Organización de la memoria

Esta memoria se organiza como sigue:

- Capítulo 1: Introducción. Descripción y objetivos del TFG.
- Capítulo 2: Herramientas utilizadas. Descripción de las herramientas utilizadas en la elaboración del TFG.
- Capítulo 3: Modelado del robot. Descripción de la inclusión del conjunto robótico en el simulador V-REP.
- Capítulo 4: Diseño de la *toolbox*. Descripción del diseño e implementación en MATLAB de la *toolbox*.
- Capítulo 5: Resultados. Presentación de los tiempos de respuesta y de las dos aplicaciones implementadas.
- Capítulo 6: Conclusiones y trabajos futuros. Reflexión personal sobre qué hemos conseguido y qué se puede realizar a partir del trabajo realizado.
- Apéndice A: Manual del Usuario. Pequeño tutorial que ilustra el funcionamiento del entorno de programación.
- Apéndice B: Manual del Desarrollador. Descripción de cómo añadir nuevas funciones al entorno de programación desarrollado.

Capítulo 2

Herramientas utilizadas

En este capítulo de la memoria vamos a hablar sobre las herramientas que vamos a utilizar para la elaboración de este proyecto. Como no podría ser de otra manera, vamos a comenzar la descripción de las herramientas con nuestro conjunto robótico.

2.1. Conjunto Robótico

Este proyecto gira en base a un robot, mejor dicho, a un conjunto robótico, puesto que nuestro robot está formado a su vez por dos robots distintos: un robot móvil Turtlebot-2 y un brazo articulado denominado WidowX.

Comencemos con Turtlebot. Se trata de un kit robótico de bajo coste, pensado tanto con fines educativos como investigadores. Además, su *software* es de código abierto y soporta arquitectura ROS.

Este kit robótico consta de los siguientes componentes:

- Una base móvil Kobuki
- Un sensor Kinect de Microsoft
- Un *netbook* con Ubuntu y ROS preinstalado
- El *hardware* necesario para el sensor Kinect
- La estructura de Turtlebot
- Una *docking station* para cargar la batería

Las especificaciones mecánicas de Turtlebot-2 (figura 2.1) son que se muestran a continuación:

- Dimensiones: 354 x 354 x 420 mm
- Peso: 6.3 kg



Figura 2.1: Kit robótico Turtlebot-2, con su *docking station*

Las especificaciones de la base Kobuki (figura 2.2) son las siguientes:

- Velocidad lineal máxima: 0.70 m/s
- Velocidad angular máxima: 180 grados/s (a partir de 110 grados/s el rendimiento del giroscopio se degrada)
- Carga máxima: 5 kg (suelo duro), 4 kg (alfombra)
- Evitación de precipicios: No se moverá hacia una profundidad mayor a 5 cm
- Obstáculos: es capaz de superar obstáculos de 12 mm o menores
- Odometría: 52 *ticks*/giro del *encoder*, 2578.33 *ticks*/giro de la rueda
- Giroscopio: calibrado en fábrica, 3 ejes (110 grados/s)
- *Bumpers*: izquierdo, central, derecho
- Sensores de precipicio: izquierdo, central, derecho
- Sensores de levantamiento de rueda: izquierdo, derecho
- LEDs programables: 2 x LED bicolor
- Botones: 3
- Frecuencia de actualización de los datos de los sensores: 50Hz

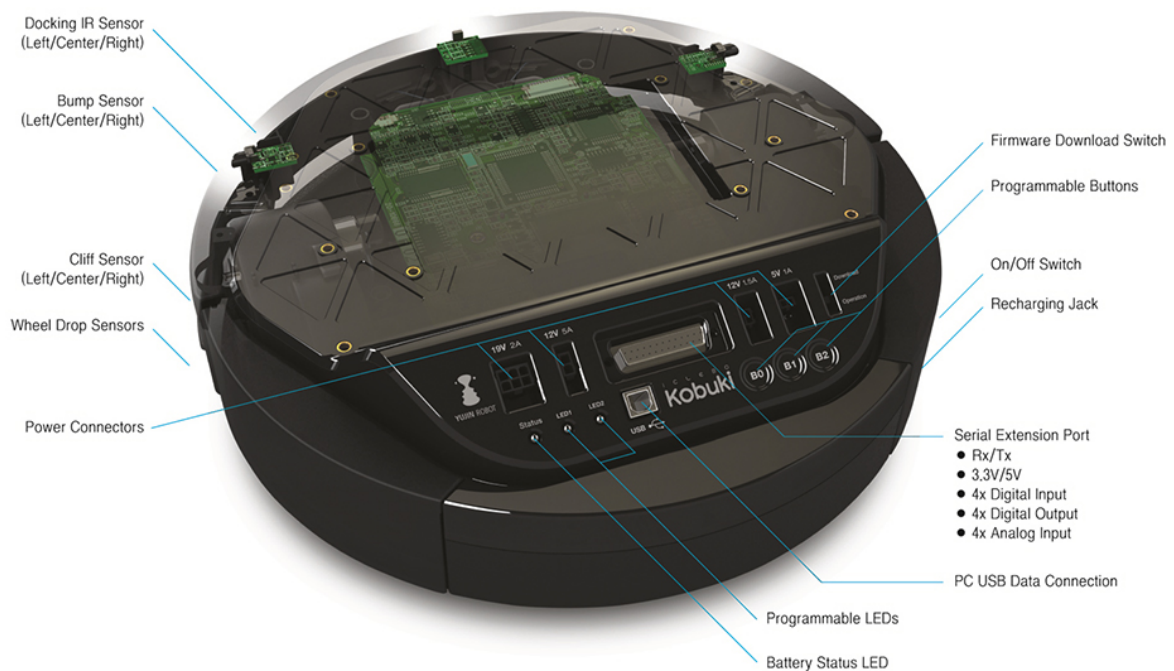


Figura 2.2: Base móvil Kobuki

En cuanto a la visión y profundidad, nos la proporciona el sensor Kinect de Microsoft (figura 2.3). Sus características son las siguientes:

- Cámara RGB con resolución de 640 x 480 píxeles @ 30 *frames* por segundo
- Cámara de profundidad con resolución de 320 x 240 puntos
- Distancia máxima detectable de 4.5 metros
- Distancia máxima detectable de 0.4 metros
- Campo de visión horizontal de 57 grados
- Campo de visión vertical de 43 grados



Figura 2.3: Sensor Kinect

Por último, tenemos al brazo robótico WidowX (figura 2.4). Sus especificaciones son las que se muestran a continuación. En el cuadro 2.1 podemos ver los rangos de ángulos admisibles para cada una de las articulaciones. La apertura de la pinza es regulable, tal y como veremos más adelante en la memoria.

- Peso: 1400 gramos
- Alcance vertical: 55 centímetros
- Alcance horizontal: 41 centímetros
- Carga máxima: hasta 800 gramos, dependiendo de la distancia a la base
- Grados de libertad: 5
- Controlador: ArbotiX-M Robocontroller, el cual es compatible con ROS

Articulación	Ángulo mínimo en grados	Ángulo máximo en grados
Base	-180	180
Hombro	-90	90
Bíceps	-90	90
Antebrazo	-90	90
Muñeca	-150	150

Cuadro 2.1: Rango de ángulos para las articulaciones de WidowX.



Figura 2.4: Brazo robótico WidowX

2.2. V-REP

Para simular nuestro robot, hemos elegido V-REP. Se trata de un simulador robótico multiplataforma (está disponible tanto para Windows como para Linux y Mac OS).

Con respecto a su licencia, vamos a emplear la versión educativa (EDU). Se trata de una versión sin limitación alguna, siempre que se use con fines educativos tal y como vamos a hacerlo nosotros. Además, el código fuente del programa está disponible, por si estimamos oportuno realizar alguna modificación en el mismo.

También está disponible una versión de la licencia, denominada *player*, que permite reproducir simulaciones en cualquier ordenador, sin la restricción de ámbito educativo. Permite realizar algunas modificaciones en la simulación, pero está bastante limitada. Puede ser interesante puesto que ocupa menos espacio que la versión educativa.

Con respecto a la versión de V-REP que vamos a utilizar, va a ser la 3.3.0, sobre sistema operativo Windows 7 de 64 bits. No obstante, hemos probado el modelo de nuestro robot en versiones posteriores y no ha habido problema alguno aparente, aunque es recomendable usar la citada versión para evitar imprevistos desagradables.

Hasta aquí hemos descrito el simulador que vamos a utilizar... ¿Pero por qué V-REP? Existen varias razones para ello. En primer lugar, hay que destacar la excelente simulación física. Nos referimos a que es muy realista (usa motores físicos que se emplean en numerosos videojuegos), y a que incluye la dinámica (aceleraciones, colisiones, masas, motores, sensores de visión e incluso partículas).

Por otro lado, tenemos que V-REP consta de una API bastante bien documentada y estructurada. Su lenguaje nativo es Lua, que es interpretado. La curva de aprendizaje de este lenguaje es bastante suave. Lo emplearemos en la implementación de los *scripts* asociados a nuestro modelo, tal y como veremos más adelante.

Esta API consta de un conjunto de funciones que son accesibles de manera remota. Se puede acceder desde distintos lenguajes (C++, MATLAB, Octave, Python, Java, entre otros). De todos ellos, el que más nos va a interesar es MATLAB, ya que es la segunda herramienta que vamos a utilizar, tal y como veremos a continuación.

Otro motivo para elegir V-REP ha sido la facilidad que presenta para el modelado de objetos en general, escenas y, cómo no, robots. No se trata de un software de edición gráfica en 3D, pero consta de una serie de herramientas que permiten un manejo sencillo para alguien no experto en esa materia. Aunque tiene una librería con bastantes objetos y robots de ejemplo, no es nada complicado añadir lo que necesitemos a la misma, bien sea creándolos desde cero o bien a partir de otros ya creados.

Si ya tenemos nuestro robot modelado en 3-D, en formato COLLADA (con AutoCAD, por ejemplo), V-REP tiene un *plugin* que permite importar esos ficheros de descripción 3-D directamente, de forma que solo tendríamos que montar las piezas en V-REP (veremos cómo se hace en el apartado de modelado de esta memoria). Si a tenemos la descripción del robot en formato URDF, podemos emplear el *plugin* que permite importar ese

formato directamente. No obstante, la importación suele presentar ciertas incongruencias con el resultado que cabría esperar. URDF es un tipo de fichero XML que contiene la descripción de un robot, incluyendo la posición de las articulaciones y sensores, además de diversas cuestiones relacionadas con el comportamiento dinámico del mismo (es decir: masas, centros de masa, inercias, rozamientos, orientaciones de componentes...).

A día de hoy, no parece existir un modelo de Turtlebot-2 ni de WidowX en V-REP. Por tanto, el hecho de realizar el modelado de ambos y aprovechar todas las características que este simulador nos brinda va a resultar muy interesante.

2.3. MATLAB

Habiendo elegido el simulador, necesitamos saber cómo programar nuestro robot; no queremos quedarnos solo con el modelado del mismo, sino que pretendemos también poder crear aplicaciones que hagan uso del mismo.

Una primera opción podría ser la de usar el propio simulador V-REP como base de programación. Esto implicaría, en principio, tener que programar en Lua. De momento esta opción está descartada, puesto que, además, solo permitiría el control local (es decir, en la propia máquina) de la simulación, cosa que limita bastante las posibilidades.

Nuestra idea es poder controlar el simulador de manera remota. Esto no elimina la posibilidad de poder hacerlo sobre la misma máquina, y nos abre un abanico de posibilidades bastante grande.

Una de las cuestiones más interesantes en lo que respecta a esta cuestión es el hecho de poder separar la máquina que ejecuta el simulador (es decir, V-REP) de donde se ejecute nuestra aplicación (el software remoto). De esta forma, podemos separar la carga computacional y llevar a cabo simulaciones mucho más complejas. Tengamos en cuenta que algunos algoritmos robóticos (por ejemplo, específicos de Visión por Computador) requieren cierta potencia computacional. Al separar el simulador de estos cálculos conseguiremos que la simulación sea más fluida y obtendremos resultados más realistas.

Parece claro que necesitamos usar la API remota de V-REP, ¿pero desde qué lenguaje la utilizamos? En este trabajo nos hemos decantado por MATLAB. MATLAB es un software matemático ampliamente usado en Ingeniería. Destaca por su versatilidad y su cantidad de funciones incorporadas. Al ser un lenguaje interpretado y fácil de aprender, es idóneo para hacer pruebas de manera rápida y elegante gracias a sus funciones. Además, tiene a sus espaldas una enorme comunidad de usuarios por todo el mundo y una documentación bastante decente. Se trata de software privativo, pero por suerte nuestra Universidad proporciona licencia para todos los miembros de la comunidad educativa, incluyendo a los propios alumnos. De esta manera, se emplea en la docencia de diversas asignaturas impartidas en nuestra Escuela (por citar algunos ejemplos: Programación de Robots, Visión por Computador, Procesamiento de Imágenes y Vídeo, Control por Computador, Métodos Estadísticos para la Computación, etcétera).

Así pues, el hecho de emplear MATLAB no debe suponer problemas para aquellos alumnos que hayan cursado alguna de esas asignaturas o que hayan aprendido MATLAB por su cuenta.

Otro de los motivos para emplear MATLAB ha sido la reciente adquisición por parte de la Universidad de la *Robotics System Toolbox*, la cual incluye gran cantidad de herramientas para trabajar con robots (permite comunicación directa con ROS), lo que la hace idónea para controlar el robot real desde el propio MATLAB con la misma aplicación que habíamos empleado para controlar el robot simulado.

2.4. Lua y ROS

Como lenguaje de programación en los *scripts* asociados al modelo de nuestro conjunto robótico vamos a emplear Lua [6]. Se trata de un lenguaje de programación imperativo, estructurado y ligero con semántica extendible. A partir de la versión 5.0, su licencia es compatible con la GPL.

Es el lenguaje de programación nativo de los *scripts* de V-REP. las variables no tienen tipos, y todas las estructuras de datos (vectores, conjuntos, ...) pueden ser representadas usando la única estructura de datos que tiene Lua: la tabla.

Internamente, los programas no son interpretados directamente, sino que se compilan a código *bytecode* antes de su ejecución en la máquina virtual. Tal y como puede verse en el siguiente código de ejemplo, su estructura es parecida a C y Perl, que son los lenguajes en los que está basado:

```
function factorial(n)
  if n == 0 then
    return 1
  else
    return n * factorial(n - 1)
  end
end
```

Por otro lado, nuestro conjunto robótico funciona con ROS [5]. ROS (*Robot Operating System*) es un *framework* para el desarrollo de software para robots que provee la funcionalidad de un sistema operativo. Se trata de software libre con licencia BSD.

ROS consta de dos partes. La primera de ellas es el sistema operativo en sí mismo, y la segunda (*ros-pkg*) son una serie de paquetes que la comunidad ha desarrollado.

Gracias a la *Robotics System Toolbox* de MATLAB, conectaremos con ROS directamente, lo que nos permitirá ejecutar nuestros programas implementados sobre el robot real.

Capítulo 3

Modelado del robot

En este capítulo de la memoria vamos a tratar el modelado de nuestro conjunto robótico. Esto es lo primero que hemos hecho en nuestro trabajo, ya que a partir del modelo diseñaremos la correspondiente *toolbox* de MATLAB.

3.1. Analizando el problema

Para empezar, puede resultar interesante estudiar las distintas posibilidades que tenemos a la hora de incluir el modelo de nuestro robot en el simulador. La primera de ellas consiste en utilizar las herramientas de modelado que incluye el propio programa. Esto supondría tomar medidas del robot real y, a partir de las mismas, utilizar las formas básicas que nos proporciona el simulador para componer el robot. La primera pega que tiene este procedimiento es el hecho de lo engorroso e inexacto que es el hecho de tomar medidas, más aún, si tenemos en cuenta el robot con el que nos encontramos, que consta de bastantes piezas y recovecos. Podría darse el caso de que tuviéramos esas medidas, ya que nos las hubiera proporcionado el fabricante; para el robot que nos ocupa, no hemos encontrado una descripción física pormenorizada de cada una de las piezas, solo las dimensiones externas del robot y su peso.

Otro detalle a tener en cuenta es que los elementos que nos proporciona V-REP para componer un robot no son demasiado sofisticados. Esto quiere decir que solo vamos a poder emplear planos, esferas, cilindros, cubos, y poco más. Por tanto, es de esperar que con un ratón de ordenador y los elementos citados el resultado no va a ser demasiado vistoso.

Existe otra opción para incluir nuestro modelo en V-REP. Consiste en importar ficheros URDF, previamente generados, gracias al *plugin* existente para tal cometido.

Los ficheros URDF, cuyas siglas se corresponden con *Unified Robot Description Format*, son una serie de ficheros en formato XML que representan el modelo de un robot. Un ejemplo de fragmento de uno de este tipo de ficheros podría ser el que se muestra en la página siguiente.

```

<joint name="wheel_left_joint" type="continuous">
  <parent link="base_link"/><child link="wheel_left_link"/>
  <origin rpy="-1.57079632679 0 0" xyz="0.00 0.115 0.0250"/>
  <axis xyz="0 0 1"/>
</joint>
<link name="wheel_left_link">
  <visual>
    <geometry>
      <mesh filename="package://kobuki_description/meshes/wheel.dae"/>
    </geometry>
    <origin rpy="0 0 0" xyz="0 0 0"/>
  </visual>
  <collision>
    <geometry>
      <cylinder length="0.0206" radius="0.0352"/>
    </geometry>
    <origin rpy="0 0 0" xyz="0 0 0"/>
  </collision>
  <inertial>
    <mass value="0.01"/>
    <origin xyz="0 0 0"/>
    <inertia ixx="0.001" ixy="0.0" ixz="0.0" iyy="0.001" iyz="0.0"
      izz="0.001"/>
  </inertial>
</link>

```

Como se puede observar, el fichero consta de todo tipo de directivas: desde tamaños de piezas hasta masas, pasando por articulaciones y sensores. Una de las cuestiones más interesantes es que permite la importación de ficheros *mesh*. Estos ficheros contienen diagramas de figuras en tres dimensiones, y pueden ser generados con alguna herramienta gráfica profesional. De esta manera, tenemos la posibilidad de que las formas que componen nuestro robot simulado sean más complejas y parecidas al real.

La cuestión es: ¿Tenemos que tomar medidas en el robot real o disponemos ya de un punto de partida?

Por suerte, en la *wiki* de ROS existe un paquete denominado “turtlebot_description” que contiene los ficheros URDF y los *meshes* que, en principio, deberían permitirnos importar el robot en V-REP. Este paquete tiene licencia BSD, así que no hay problema en que lo utilicemos como base de nuestro modelo.

Con respecto a WidowX, nos encontramos con algo similar. El proveedor del robot (Robotnik [16]) nos ha proporcionado un paquete parecido con la descripción del brazo, incluyendo los correspondientes meshes.

Por tanto, parece que llegados a este punto podemos importar ambos modelos en V-REP y unirlos, para así obtener el conjunto robótico completo.

3.2. Incluyendo el robot en V-REP

Ya hemos visto las posibilidades que tenemos para añadir nuestro robot al simulador V-REP.

Lo siguiente es descargarnos el paquete “turtlebot_description” que se encuentra en la *wiki* de ROS (figura 3.1).

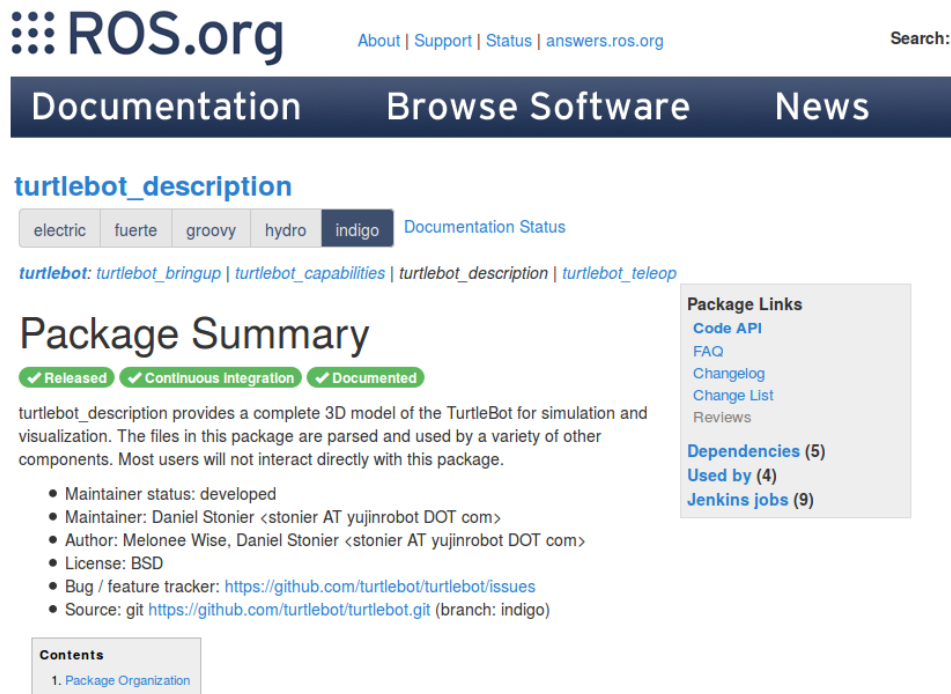


Figura 3.1: Paquete “turtlebot_description” en la *wiki* de ROS.

Nos descargamos los ficheros desde *github*, en su rama *indigo*, que es la última que tiene soporte. Dentro del fichero comprimido, encontraremos una carpeta correspondiente con los ficheros de la descripción. Estos ficheros incluyen tanto los URDF, con la descripción en sí misma, como los *meshes* con los volúmenes en tres dimensiones. No obstante, no nos encontramos con lo esperado (ver figura 3.2) parece que no están los ficheros URDF en sí mismos, si no unos ficheros de tipo Xacro.

Este tipo de ficheros contienen código escrito en lenguaje de macros XML. Según sus creadores, de esta forma es posible crear ficheros XML más cortos y comprensibles, ya que permite contraer expresiones XML largas.

El primer impedimento es que el *plugin* que tenemos en V-REP para incluir la descripción de un modelo no entiende los ficheros *Xacro*, si no que precisa que se encuentren en el formato URDF. Para ello, tenemos que convertir esos ficheros al formato correcto.

Una forma de hacerlo es con la herramienta que nos proporciona ROS para ello. Por tanto, vamos a necesitar ejecutar ROS de alguna manera. La opción que vamos a tomar

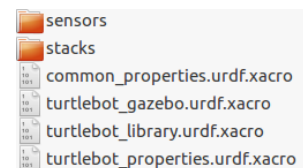


Figura 3.2: Contenido del paquete.

(de las múltiples disponibles) consiste en utilizar una máquina virtual que tenga instalado Ubuntu, junto a una preinstalación de ROS.

Por suerte, existe una máquina virtual con estas características lista para descargar, y solo hay que importarla en nuestro *software* de virtualización favorito. La máquina es descargable desde la página de Nootrix [17]. Hemos descargado la versión de 64 bits (figura 3.3), puesto que el ordenador sobre el que vamos a trabajar tiene habilitada la tecnología de virtualización y permite usar esta arquitectura. También está disponible en versión de 32 bits, para máquinas más antiguas.



Figura 3.3: Escritorio de la máquina virtual con ROS instalado.

Una vez iniciada la máquina, abrimos una terminal en el directorio donde se encuentran los ficheros *Xacro*. Podemos ejecutar el siguiente comando para intentar obtener el fichero URDF correspondiente:

```
roslaunch xacro xacro.py -o turtlebot.urdf turtlebot_gazebo.urdf.xacro
```

Donde “turtlebot.urdf” es el fichero de salida y “turtlebot_gazebo.urdf.xacro” es el de entrada. Este último consta de diversas dependencias, que intentará resolver buscando en las carpetas preconfiguradas de ROS y en la carpeta actual.

Precisamente ese es el primer impedimento con el que nos encontramos: la descripción de la base Kobuki no está incluida en la descripción de Turtlebot. Así pues, tenemos que incluirla nosotros manualmente.

De nuevo, accedemos a la *wiki* de ROS, donde encontraremos el paquete “kobuki_description”, que incluye todos los ficheros necesarios. Añadimos estos ficheros al paquete de Turtlebot y volvemos a ejecutar el comando para pasar de *Xacro* a URDF.

En esta ocasión obtenemos un fichero de salida que aparenta ser correcto y que en principio podríamos importar en V-REP (figura 3.4). No obstante, hemos de tener en cuenta lo siguiente: tenemos generado el fichero URDF, pero cuando intentemos importarlo, el simulador va a intentar buscar los ficheros que contienen los distintos *meshes*, especificados en ese fichero URDF.

Si intentamos importar el modelo en V-REP sobre Windows, nos vamos a encontrar con el problema de que el simulador no es capaz de encontrar los ficheros de los *meshes* y nuestro robot solo va a estar compuesto por formas básicas.

Una posible solución, que es por la que optamos, consiste en realizar la importación del modelo sobre una máquina que tenga ROS instalado. Para ello, vamos a usar una máquina virtual con Ubuntu y ROS. Además, tenemos que tener instalados los paquetes correspondientes a Kobuki y a Turtlebot para que a la hora de importar se encuentren los ficheros necesarios. De esta manera, sí que podríamos obtener una importación correcta.

Abrimos por tanto el simulador en la máquina virtual y ejecutamos el *plugin* disponible por defecto para la importación URDF.

En la ventana que nos aparece, usamos “Import” y seleccionamos nuestro fichero URDF previamente generado. Una vez hecho eso, y esperamos el resultado. Posiblemente encontremos algo parecido a la figura 3.5.

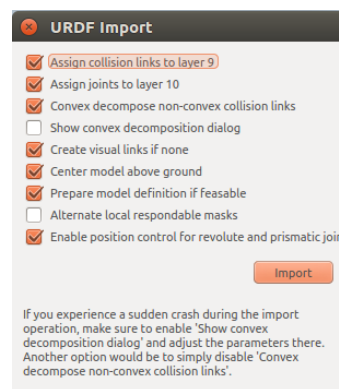


Figura 3.4: *Plugin* URDF.

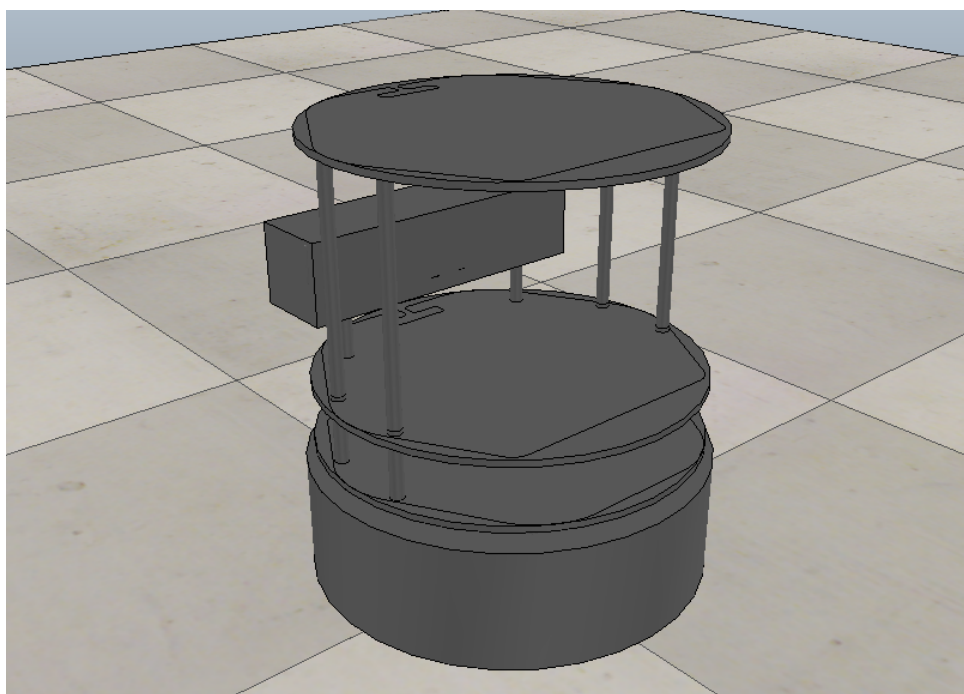
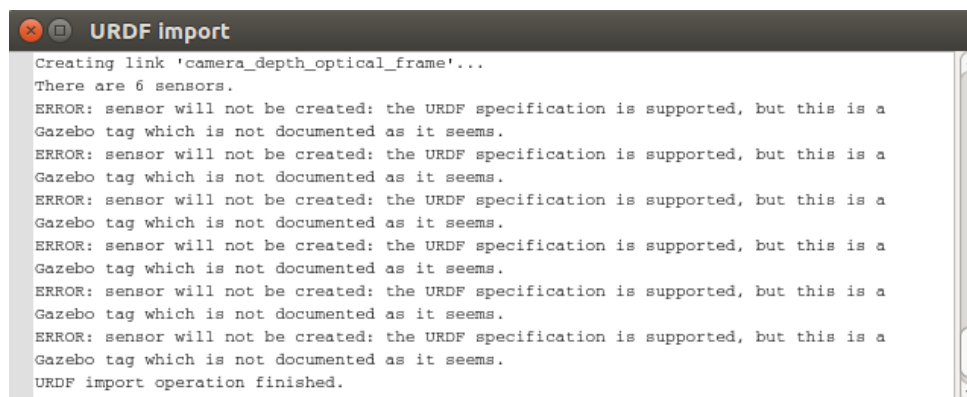


Figura 3.5: Importación de Turtlebot.

Si observamos la consola de resultados, vemos que obtenemos unos cuantos errores, tal y como muestra la figura 3.6. Estos errores son debidos a las directivas para Gazebo que incluye nuestro fichero URDF. Todo apunta a que vamos a tener que modelar manualmente algún que otro sensor.



```

URDF import
Creating link 'camera_depth_optical_frame'...
There are 6 sensors.
ERROR: sensor will not be created: the URDF specification is supported, but this is a
Gazebo tag which is not documented as it seems.
ERROR: sensor will not be created: the URDF specification is supported, but this is a
Gazebo tag which is not documented as it seems.
ERROR: sensor will not be created: the URDF specification is supported, but this is a
Gazebo tag which is not documented as it seems.
ERROR: sensor will not be created: the URDF specification is supported, but this is a
Gazebo tag which is not documented as it seems.
ERROR: sensor will not be created: the URDF specification is supported, but this is a
Gazebo tag which is not documented as it seems.
ERROR: sensor will not be created: the URDF specification is supported, but this is a
Gazebo tag which is not documented as it seems.
URDF import operation finished.

```




Figura 3.6: Consola de errores del *plugin* para URDF.

Además, vemos que el robot está modelado con formas básicas. Si nos fijamos, también se aprecia el contorno de las formas complejas. Parece que están superpuestas. Es como si nuestro robot estuviese formado tanto por formas complejas (y más vistosas) como por formas simples.

Resulta interesante echar un vistazo a la jerarquía de nuestro robot, y así darnos cuenta de cómo están conectadas cada una de las piezas que lo forman.

La jerarquía es un árbol de relaciones existentes entre los diversos componentes que forman el robot. Desde el punto de vista del modelado, es muy importante que esta jerarquía esté bien definida, puesto que de ello va a depender, en gran medida, que el comportamiento de nuestro robot sea lo más realista posible. En la figura 3.7 podemos observar dos fragmentos de la citada jerarquía.

Lo primero que debemos interpretar son cada uno de los símbolos que aparecen en la misma:

-  → *Shapes*: desde formas básicas hasta complejas agrupadas
-  → *Joints*: rotacionales, prismáticas y esféricas
-  → *Force sensors*: miden fuerzas y unen piezas entre sí

Una vez que conocemos los símbolos, analicemos la jerarquía en si misma. Vemos que todos los elementos “cuelgan” de uno, denominado “base_footprint_visual”. Podríamos decir que ese elemento es la base de nuestro robot.

Lo siguiente que nos encontramos es un sensor de fuerza, denominado “base_joint”. En V-REP, los sensores de fuerza se pueden usar para medir fuerzas existentes entre dos formas (*shapes*), y también pueden emplearse como uniones fijas entre las mismas.



Figura 3.7: Dos fragmentos de la jerarquía de nuestro robot.

En este caso, y en muchos otros que veremos más adelante, vamos a usar los sensores como puntos de unión entre dos piezas de nuestro robot. Si seguimos avanzando en la jerarquía, nos encontramos con una *shape*, de tipo primitiva (es decir, se trata de una forma básica, en este caso es un cilindro) denominada “base_link_respondable”.

Quizás este sea un buen momento para que entendamos el concepto de *respondable*. Una *shape* de tipo *respondable* va a ser empleada en el cálculo de colisiones. De esta forma, separamos las *shapes* que utilizaremos para la representación visual de nuestro robot de las *respondables*, principalmente por eficiencia computacional. Para el motor físico, requiere muchos menos cálculos el hecho de trabajar con una *shape* que sea de tipo primitiva (cilindros, cubos...) que con una *shape* con forma arbitraria.

Una *shape* con forma arbitraria puede estar compuesta por muchos (incluso miles) de triángulos. Esto hace que no sean muy recomendables para el cálculo de las colisiones, puesto que, además de enlentecerlo, puede provocar colisiones inestables.

Es por ello que, a partir de ahora, vamos a intentar separar la parte visual de nuestro robot simulado (que procurará ser visualmente agradable, con formas y curvas complejas parecidas a las reales), de la parte *respondable*.

No obstante, ya hemos visto en la primera importación que esta separación no está bien hecha. Si bien en la jerarquía vemos que existen dos *shapes* por cada pieza del robot, y que aparentemente una de ellas es la visual y otra la *respondable*, cuando observamos la representación de nuestro robot en la escena vemos como si ambas partes estuvieran

fusionadas.

Esta va a ser la primera cuestión que tenemos que solucionar en el simulador. El problema se encuentra en las *shapes* visuales. Por alguna razón, el simulador ha unido una copia de las *shapes* responsables con estas.

Vamos a ver cómo podemos solucionar esto con un ejemplo. Para ello, lo primero que vamos a hacer es doble clic sobre la *shape* denominada “base_link_visual”. Esta *shape* debería corresponderse con la parte visual de la base Kobuki. Se nos abrirá una ventana parecida a la figura 3.8.

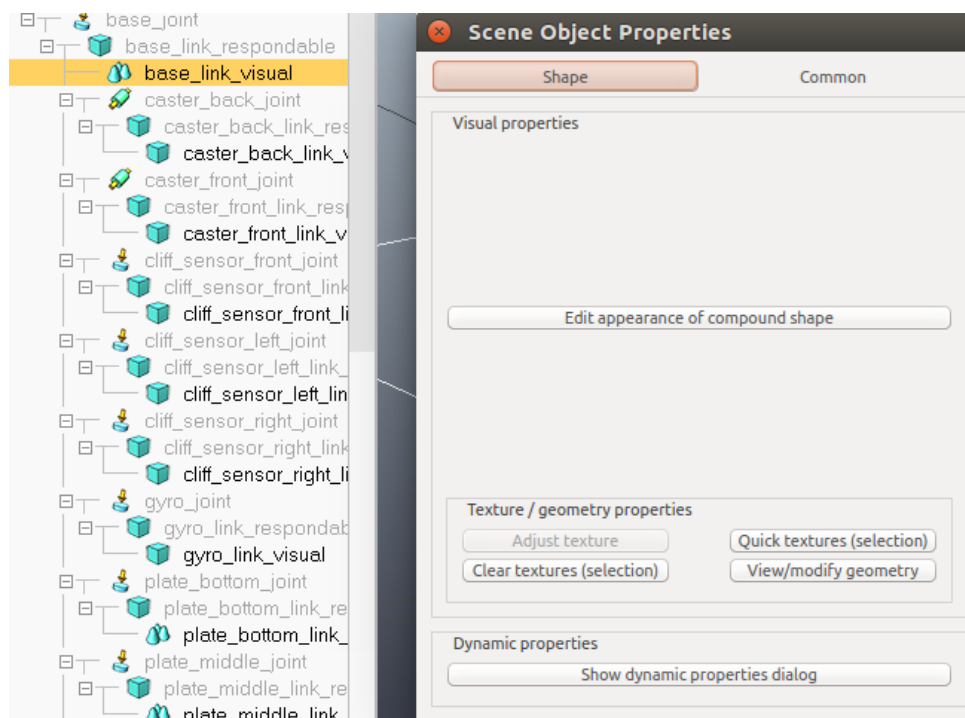


Figura 3.8: Ventana de propiedades del objeto.

En ella, hemos de hacer clic en el botón “Edit appearance of compound shape”. La idea es ver qué componentes forman la *shape* seleccionada. Tal y como muestra la figura 3.9, vemos que la *shape* está formada por dos componentes.

Esto es precisamente lo que estábamos sospechando¹, así que lo que tenemos que hacer es buscar la forma de separarlos. De esta manera, podremos eliminar el componente que no nos interesa (en este caso, la forma simple, puesto que queremos quedarnos con la más vistosa para la parte visual).

Para separar ambos componentes, tenemos que hacer clic con el botón derecho sobre la *shape* que queremos separar. Se nos abrirá un menú contextual. Posicionamos el ratón sobre el elemento “Edit”, a continuación en “Group/Mergering” y por último, hacemos clic sobre “Ungroup selected shapes”. Una vez hecho eso, obtendremos un resultado parecido

¹El hecho de sospechar que ambos componentes estaban en la misma *shape* y que podían separarse nos llevó bastante tiempo.

al de la figura 3.10.

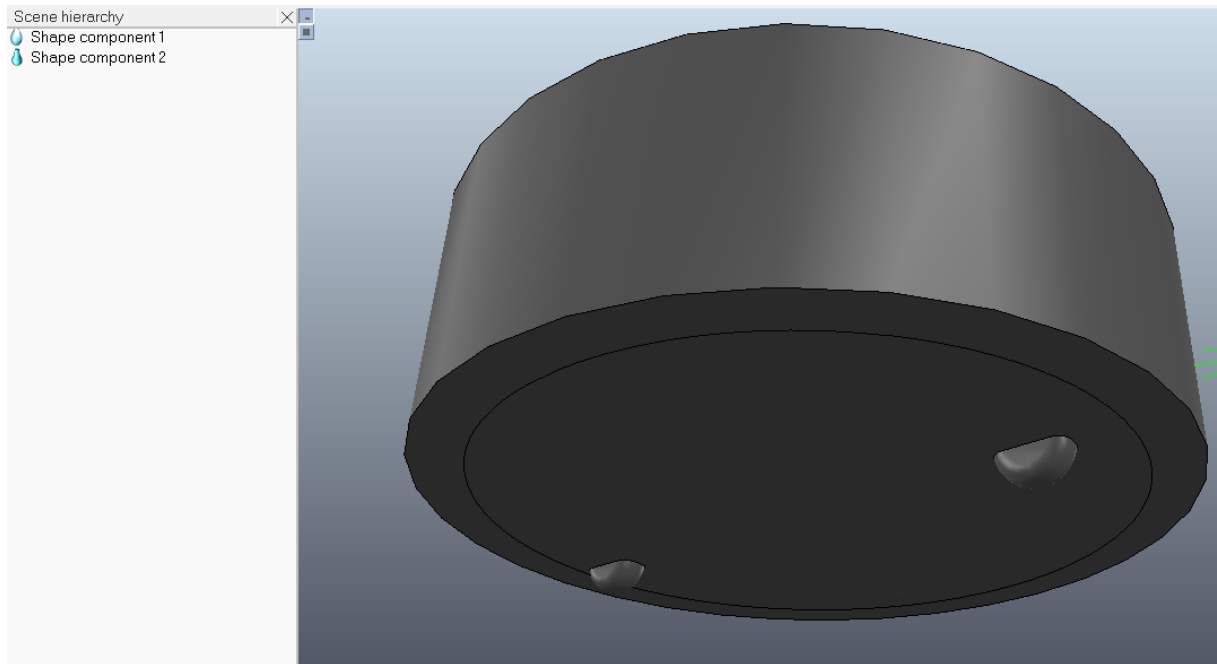


Figura 3.9: Vista de la *shape* “base_link_visual”.

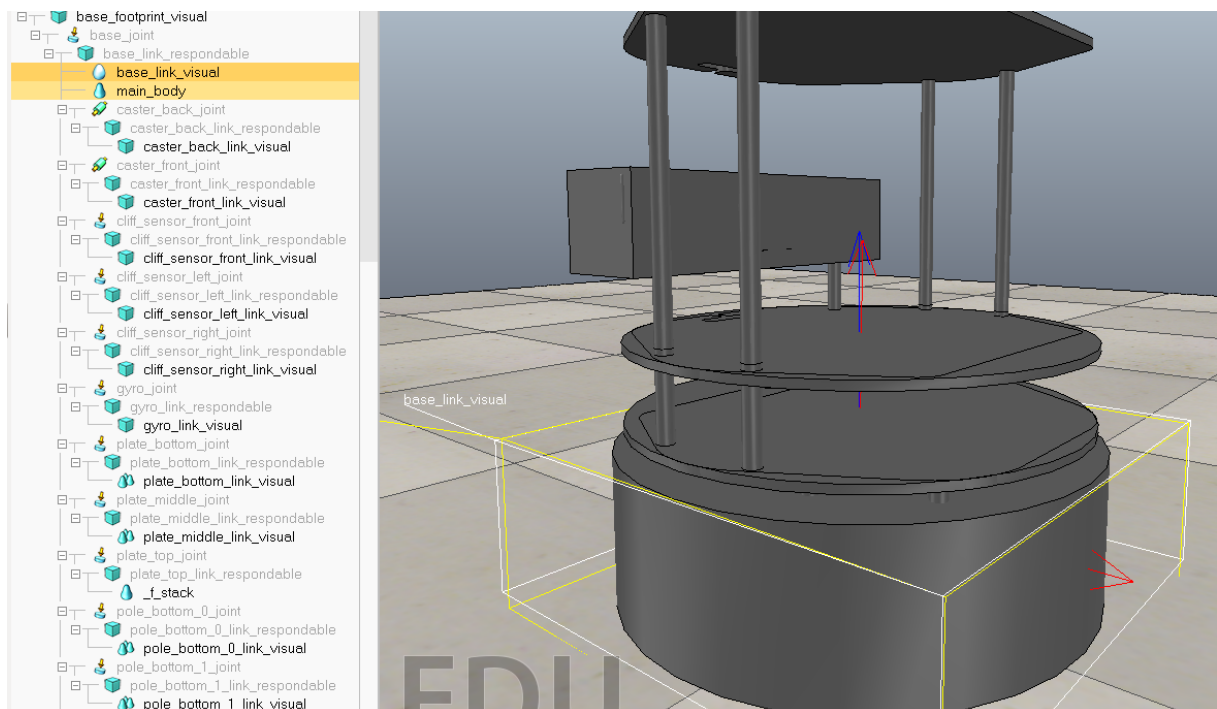


Figura 3.10: Separación de los dos componentes de una *shape*.

Si nos fijamos en los dos elementos seleccionados, vemos que uno de ellos se denomina “base_link_visual” y el otro “main_body”. Bien, pues es con el de “main_body” con el que debemos quedarnos, así que borramos el otro.

El resultado es el que se aprecia en la figura 3.11. Vemos que ahora la base Kobuki de nuestro robot tiene una forma mucho más parecida a la del robot real.

También es importante darse cuenta de la manera en la que los elementos están posicionados en la jerarquía. Es vital que la estructura esté definida de forma que las *shapes* visuales “cuelguen” de las *responsables*, tal y como se aprecia en la jerarquía de nuestro robot. Si no lo hacemos así, tendremos problemas en la simulación, puesto que el “padre” de un elemento dinámico (*respondable*) no puede ser un elemento estático (*shape* visual).

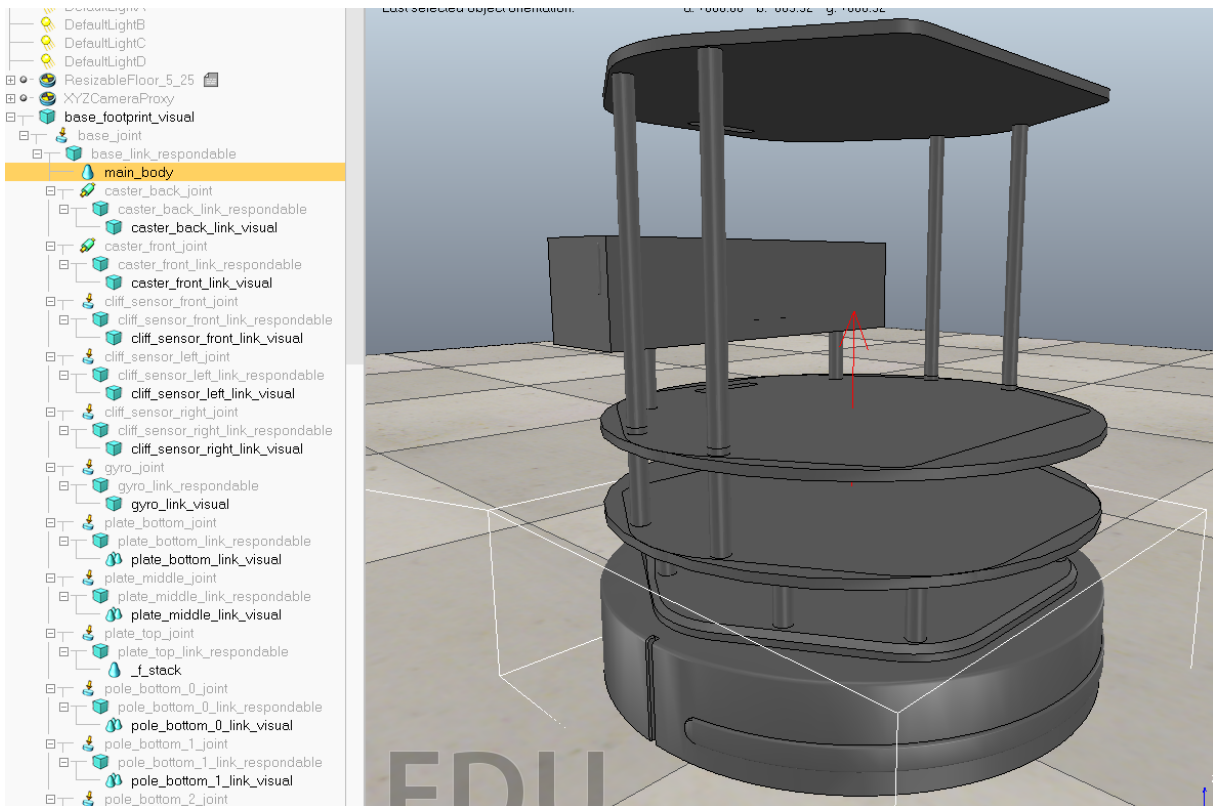


Figura 3.11: Base Kobuki en forma no primitiva.

Ahora hay que repetir este procedimiento para cada uno de los componentes de nuestro robot. Hay que separar las *shapes* visuales en sus dos componentes, y eliminar el componente que no nos interesa al igual que hemos hecho con el correspondiente a la base Kobuki. Se trata de un proceso mecánico y repetitivo, y sería muy reiterativo mostrar en esta memoria cada uno de los pasos. Una vez completada esta tarea, nos encontraremos con un resultado parecido al de la figura 3.12.

La diferencia con lo que obtuvimos al realizar la importación URDF es bastante considerable. Las formas del robot están muy conseguidas, pero aún nos queda mucho camino por recorrer. Lo siguiente que haremos será implementar los distintos sensores de Turtlebot.



Figura 3.12: Turtlebot con su parte visual correcta.

3.3. Implementando los sensores

La implementación de los sensores es uno de los puntos críticos del modelado de nuestro robot, puesto que la calidad de dicha implementación afectará de manera muy significativa al resultado.

3.3.1. Kinect

El primer sensor que vamos a añadir a nuestro modelo va a ser Kinect. Recordemos que Kinect es un sensor de visión que nos proporciona tanto una imagen en RGB (resolución VGA) como una imagen de profundidad. Este sensor es muy útil en robots móviles. Permite, entre otras cosas, ayudar a la generación de mapas y detectar obstáculos. Esta segunda cuestión la veremos más adelante, en el capítulo de resultados, con la implementación de una aplicación.

Tal y como hemos visto en la última versión de nuestro modelo (figura 3.12), el sensor Kinect está modelado físicamente. Esto quiere decir, que está su “carcasa”, pero no está definido como sensor de visión.

Una opción sería añadir dos sensores de visión: uno para la imagen RGB y otro para la imagen de profundidad, intentando emular a Kinect de la mejor manera posible.

Por suerte, V-REP incluye en su librería dos versiones de Kinect, tal y como muestra la figura 3.13. Por su simpleza, vamos a emplear la versión “kinect”, es decir, la segunda que aparece en la figura.

Añadir el sensor al modelo es bastante sencillo. Lo primero que hay que hacer es arrastrar el objeto a la escena. Una vez hecho eso, eliminamos de la escena las *shapes* que teníamos antes como representación de Kinect, para evitar duplicidades. Tenemos luego que colocar Kinect en su sitio.

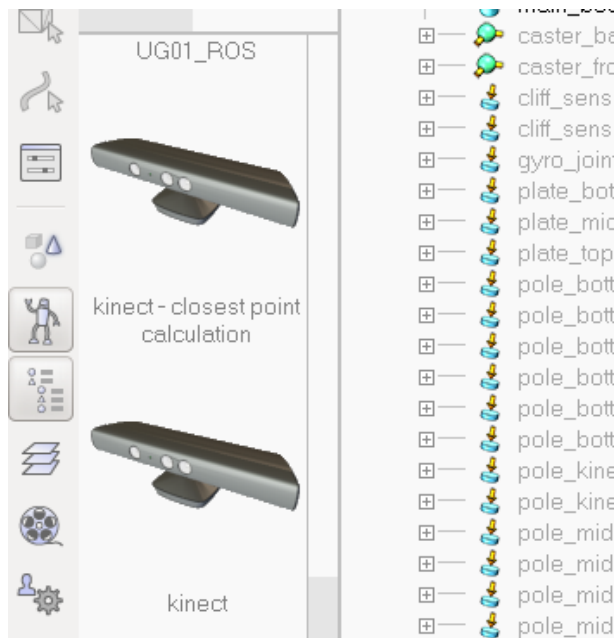


Figura 3.13: Sensor Kinect en la librería de V-REP.

Para eso, podemos hacer uso de las herramientas de “mover” y “cambiar orientación” que nos proporciona V-REP. Así, nos quedará un resultado parecido al de la figura 3.14.

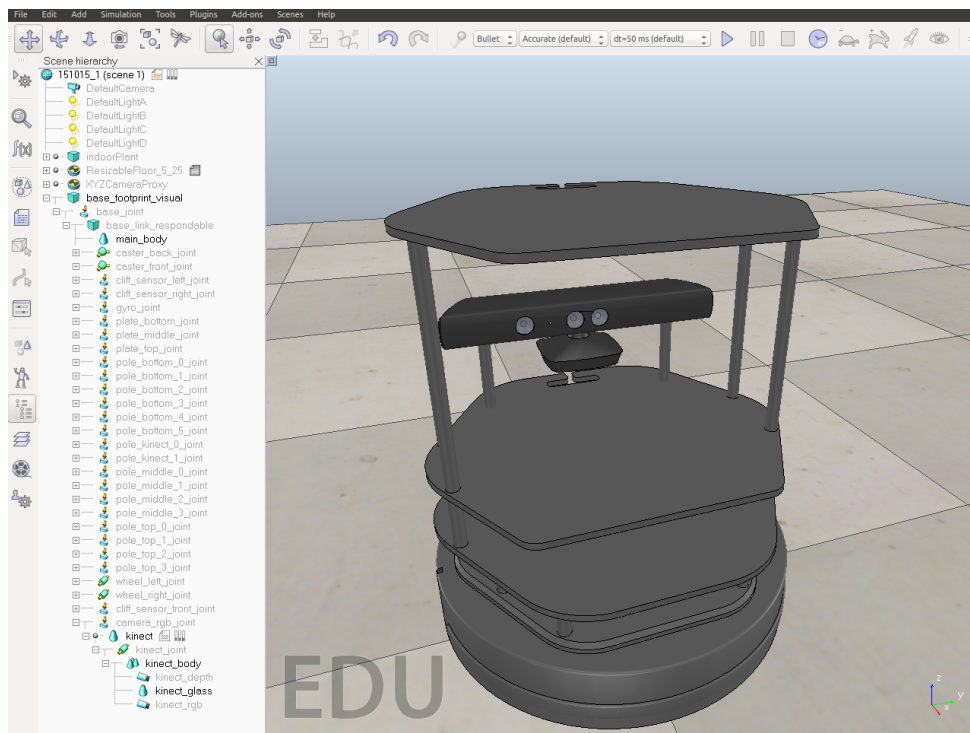


Figura 3.14: Turtlebot con Kinect colocado.

Por último, también hemos añadido el sensor Kinect a la jerarquía de nuestro robot, para que de este modo, cuando el robot se mueva, Kinect forme parte del mismo y siga

su movimiento. Si no hacemos eso, Kinect se “caería” de Turtlebot.

3.3.2. Giroscopio

El giroscopio es un sensor muy importante en nuestro robot. Nos proporciona la velocidad angular del mismo, y es clave en el cálculo de la odometría.

La base Kobuki emplea los datos del giroscopio tanto para obtener su velocidad angular (la que nos interesa principalmente es la del eje Z, es decir, la perpendicular al suelo) como su orientación con respecto al último reinicio de la odometría.

Al igual que para el caso de Kinect, V-REP incluye en su librería un giroscopio genérico, que podemos adaptar a nuestro robot.

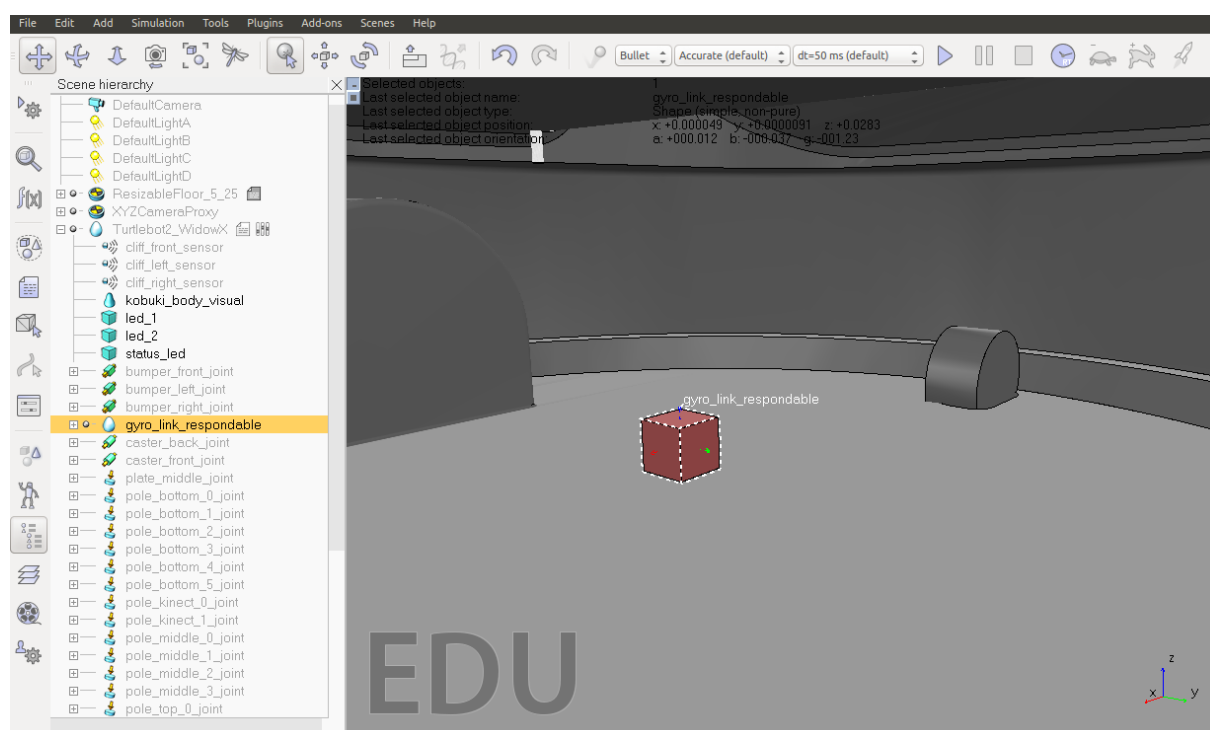


Figura 3.15: Giroscopio dentro de la base Kobuki.

En la figura 3.15 podemos observar al giroscopio dentro de la base Kobuki. Para colocarlo ahí, hemos procedido igual que con Kinect. Lo hemos arrastrado desde la librería de V-REP y lo hemos movido con la herramienta que nos proporciona para tal fin. Obsérvese que también se ha colocado en la jerarquía del modelo (marcado en amarillo oscuro).

Es vital que el giroscopio esté colocado en el centro de Kobuki, porque de otra manera podría darnos medidas que no son reales. Más adelante, en la sección de los *scripts* del modelo, veremos cómo está implementado en el código.

3.3.3. *Cliffs*

Los sensores *cliff* son los que nos ayudan a evitar que nuestro robot vuelque o caiga por un precipicio. Para ello, estos sensores miden la distancia al suelo desde la base Kobuki. El método que utilizan es por medio de luz infrarroja. Proyectan esta luz con un ángulo determinado, de manera de que si el sensor no es capaz de detectarla, es que se encuentra ante un precipicio.

También nos permiten saber si el robot va a ser capaz de sortear un escalón pequeño o por el contrario va a resultar una maniobra peligrosa, ya que miden la distancia.

Para añadirlos a nuestro robot, procedemos como en los sensores anteriores. Simplemente los arrastramos de la librería de V-REP y los colocamos en el lugar correspondiente de la base Kobuki (figura 3.16).

Aunque en la figura se pueda ver la representación de los mismos, incluyendo el rayo que proyectan, en la versión final del modelo de nuestro conjunto robótico están ocultos.

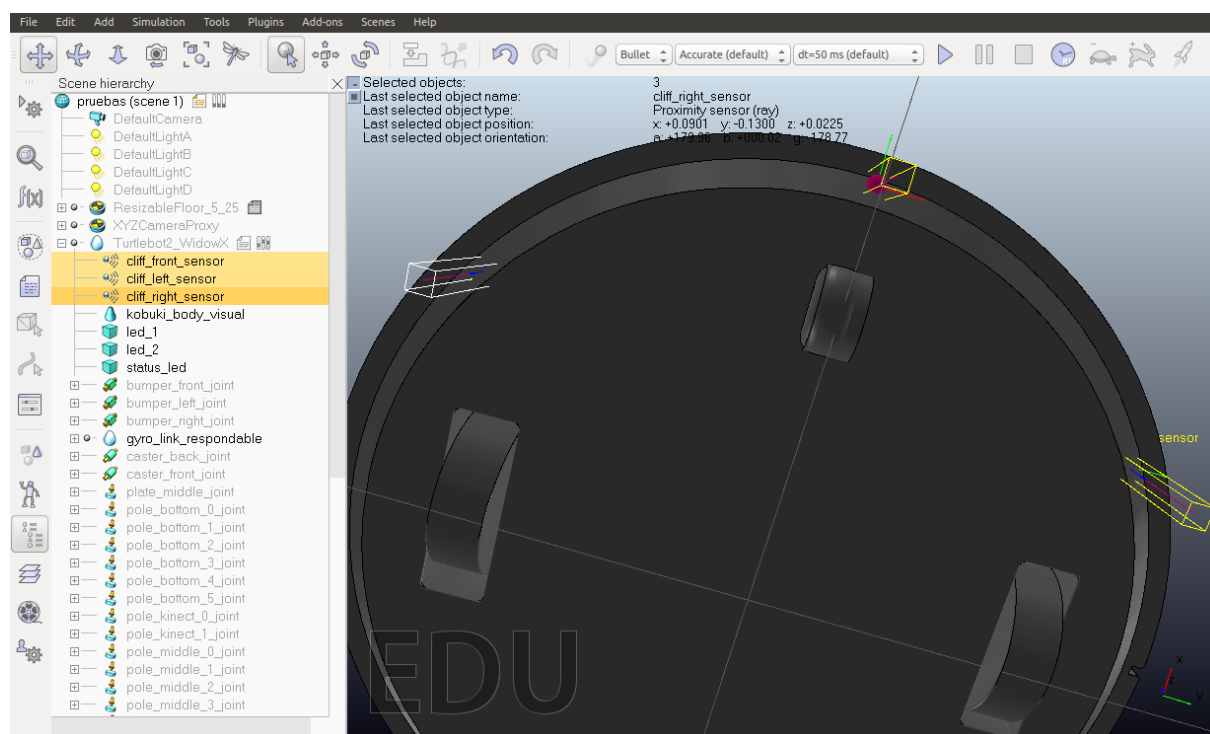


Figura 3.16: Sensores *cliff* en la base Kobuki (vista desde abajo).

Obsérvese que nuestro robot consta de tres sensores de precipicio: uno a la izquierda, otro en el centro (frontal) y otro en la derecha. Esto quiere decir que si movemos el robot hacia atrás no sabremos si hay precipicio en base a lo que nos devuelvan estos sensores. Debemos tenerlo en cuenta en el diseño de nuestras aplicaciones si queremos evitar sustos.

Por último, vemos que los sensores también se han añadido a la jerarquía del robot, tal y como hemos visto en la figura. Al ser elementos estáticos (no se mueven con respecto a

su “padre”) no es necesario que les pongamos ningún punto de unión, si no que “cuelgan” directamente del *shape responsable* que representa a la base Kobuki.

3.3.4. *Bumpers*

Otro tipo de sensores que tenemos disponibles en nuestra base Kobuki son los *bumpers*. Tal y como es de esperar, estos sensores detectan si el robot ha chocado o no contra algún objeto.

Estos sensores solo constan de dos estados: choque o no choque. En el robot real están incluidos en forma de pulsadores. Concretamente, el robot consta de tres, al igual que para el caso de los *cliffs*: uno a la izquierda, uno central y otro a la derecha.

Hasta ahora hemos podido añadir los sensores que nos han ido haciendo falta a nuestro conjunto robótico arrástrándolos desde la librería. Para el caso de los *bumpers*, vamos a tener que trabajar un poco más.

El objetivo es el siguiente: hay que modelar una pieza que se encuentre colocada en el frontal de la base Kobuki y que se mueva con respecto a esta. Además, hemos de ser capaces de detectar el movimiento de esa pieza, puesto que eso significaría que el robot ha chocado contra algo.

El primer problema que nos encontramos es que no podemos asignar un solo *shape* a tres sensores, al menos, de manera inmediata. Además, debido a la forma de realizar los cálculos que tiene el motor físico, es posible que si usamos una sola pieza como “parachoques” de Kobuki se produzcan propagaciones del choque en los tres sensores.

Así pues, la opción por la que hemos optado consiste en dividir el parachoques en tres partes, una para cada sensor. Para obtener los tres *shapes*, hemos sustraído la parte delantera de Kobuki, seleccionando los conjuntos de triángulos que abarquen las secciones correspondientes.

En la figura 3.17 pueden contemplarse en primer plano dos de las tres partes que componen el parachoques. Obsérvese que, además, se han desplazado un poco hacia delante y hacia el lado correspondiente (izquierda o derecha), con el fin de que exista un espacio que permita el desplazamiento entre la base Kobuki y las piezas creadas. Si no hacemos eso, los objetos colisionarían directamente contra la base y no serían detectados por los *bumpers*.

Con respecto a los sensores, una de las primeras opciones que barajamos fue modelar los interruptores como sensores de fuerza. De esta manera, se comprobaría que la fuerza estuviese por encima de un umbral, y en caso de superarlo se entendería que el robot ha chocado.

En principio no parece una mala opción (de hecho, tuve los *bumpers* modelados de esa forma durante bastante tiempo) y los resultados obtenidos no son malos. No obstante, esta implementación tiene un pequeño problema, y es que los sensores de fuerza no permiten

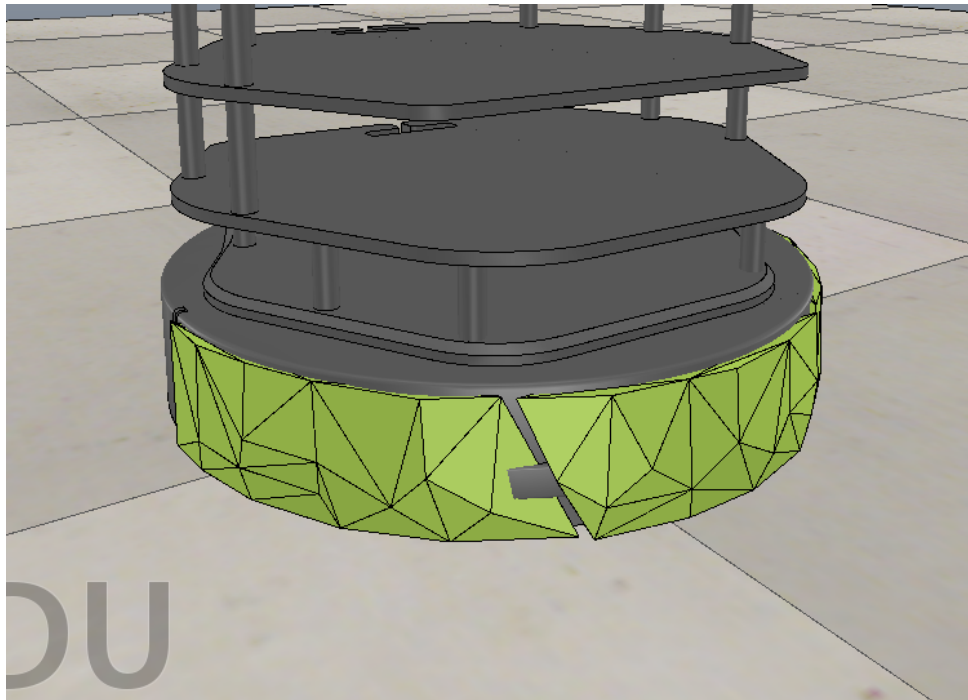


Figura 3.17: Tres *shapes* modelan el “parachoques” de Kobuki.

movimiento. Es decir, que la pieza empleada (el parachoques) no se mueve respecto a la base.

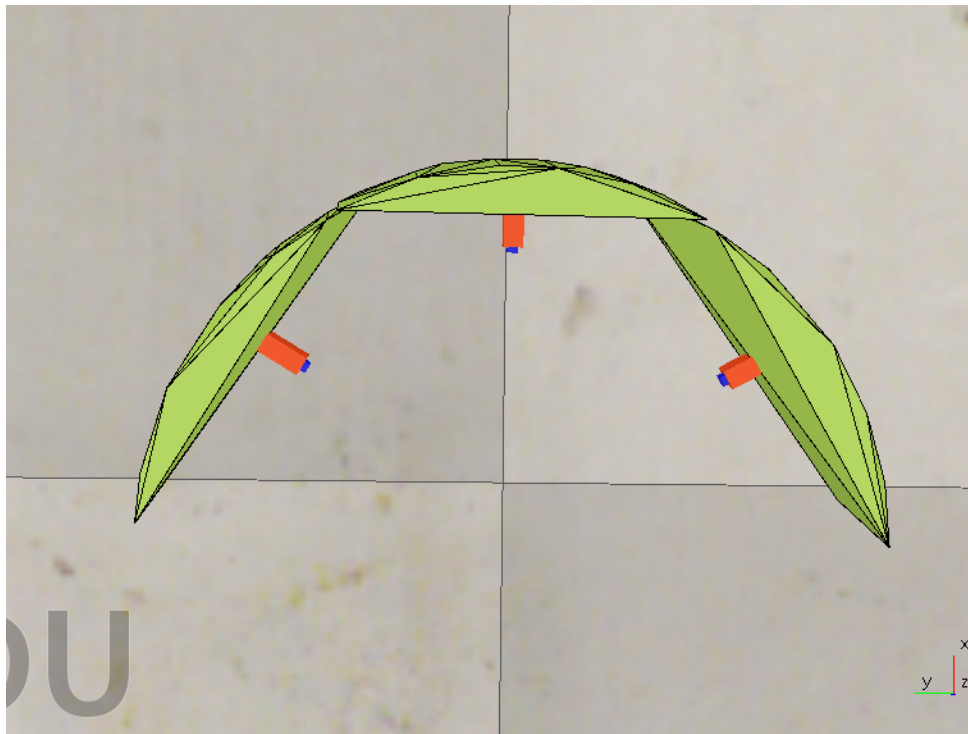


Figura 3.18: Detalle del modelado de los *bumpers*.

Podríamos decir que es un detalle menor, pero estamos intentando que el comportamiento de nuestro conjunto robótico simulado sea lo más parecido al real.

Para mejorar esto, vamos a emplear articulaciones prismáticas. De esta manera, quedará modelado el movimiento que existe en el robot real con respecto a la base Kobuki. Es importante que estas articulaciones ejerzan algún tipo de fuerza, porque en caso contrario los *bumpers* se quedarían “pulsados” y no volverían a su posición normal.

En la figura 3.18 puede observarse en detalle el modelado completo de los *bumpers*, visto desde arriba. Además de los *shapes*, aparecen las tres articulaciones prismáticas que emulan los pulsadores.

Más adelante, en la sección encargada de describir la generación de los *scripts* del modelo, veremos cómo accedemos al estado de los tres *bumpers*.

3.3.5. *Wheels Drop*

Nuestra base Kobuki consta también de unos sensores de tipo *wheel drop*. Estos sensores están ubicados en cada una de las dos ruedas motrices de la base, y permiten saber si la base se ha levantado del suelo.

Esto es útil para comprobar que el conjunto robótico ha volcado (cosa que debe evitarse a toda costa) o que alguien lo ha levantado del suelo.

En el caso del simulador, parece que la opción más probable es la primera. Por ello, puede ser bastante interesante tener modelados estos sensores.

El principio de funcionamiento de los mismos es muy parecido al de los *bumpers*. Son dos pulsadores (en este caso, normalmente pulsados) que se encuentran en la parte baja de la base Kobuki. Cuando el robot se levanta del suelo, las ruedas bajan y estos pulsadores pasan a estar abiertos.

Estos sensores no están incluidos en la librería de V-REP, y su implementación no es inmediata. La primera opción que se nos podría ocurrir sería la de intercalar un sensor de fuerza entre la base Kobuki y los motores de las ruedas. De este modo, mediríamos las fuerzas y si bajan de un umbral, diríamos que el conjunto robótico ha volcado.

Haciéndolo así, ocurriría lo mismo que dijimos para los *bumpers*, y es que el comportamiento no sería del todo fiel al del robot real. Además, no es viable desde el punto de vista de la implementación, puesto que V-REP no permite intercalar un sensor de fuerza entre una *shape* (Kobuki, en este caso) y una articulación (el motor de una rueda).

La opción que vamos a tomar es la de las articulaciones prismáticas. No obstante, nos encontramos con el mismo problema: V-REP no permite “enganchar” articulaciones entre sí de forma directa; hay que colocar una *shape* de por medio.

Eso es precisamente lo que hemos hecho: generar dos *shapes*, que van a ser dos cilindros, para colocarlos entre los motores de las ruedas y las articulaciones prismáticas que modelan los *wheel drop*.

Estos cilindros podrían modelar perfectamente el peso de los motores, incluyendo la caja reductora que incluyen los mismos. En la figura 3.19 puede verse la implementación definitiva. Obsérvense los dos cilindros de color azul claro, que son las dos *shapes* que unen los motores de las ruedas (ubicados en sus ejes) con la base Kobuki (transparente en esa figura). También pueden observarse las articulaciones prismáticas que modelan los sensores, en color naranja y azul, perpendiculares a las ruedas.

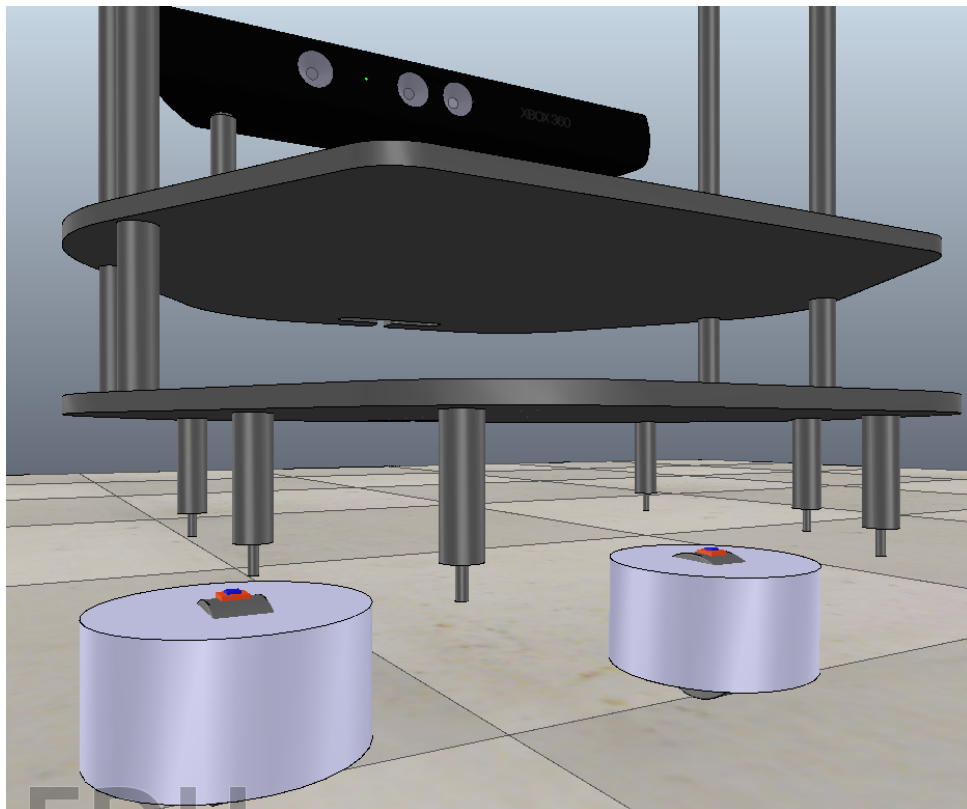


Figura 3.19: Detalle del modelado de los *bumpers*.

Más adelante, en la sección de generación de los *scripts* del modelo, veremos cómo comprobamos el estado de estos sensores.

3.3.6. Láser

Antes de añadir el modelo de WidowX a Turtlebot, vamos a completar este último con un sensor láser de distancias. Nuestro robot real no dispone actualmente de uno de este tipo, pero es previsible que en un futuro cercano sí que disponga. Por ello, es interesante que el modelo ya disponga de uno y que además esté hecha la comunicación del mismo con MATLAB, tal y como veremos en el capítulo de diseño de la *toolbox*. Este tipo de sensores se utiliza mucho en navegación de robots, puesto que suelen tener un ángulo de visión bastante grande y una muy buena precisión.

Si bien es posible utilizar el sensor Kinect para este cometido, nos encontramos con

que su ángulo de visión es tan solo de 57 grados, por lo que la información obtenida puede quedarse un poco corta.

El sensor concreto que vamos a emplear es el Hokuyo UGR-04LX, que se encuentra incluido en la librería de V-REP, así que lo que hacemos es arrastrarlo a la escena y colocarlo en el lugar correspondiente.

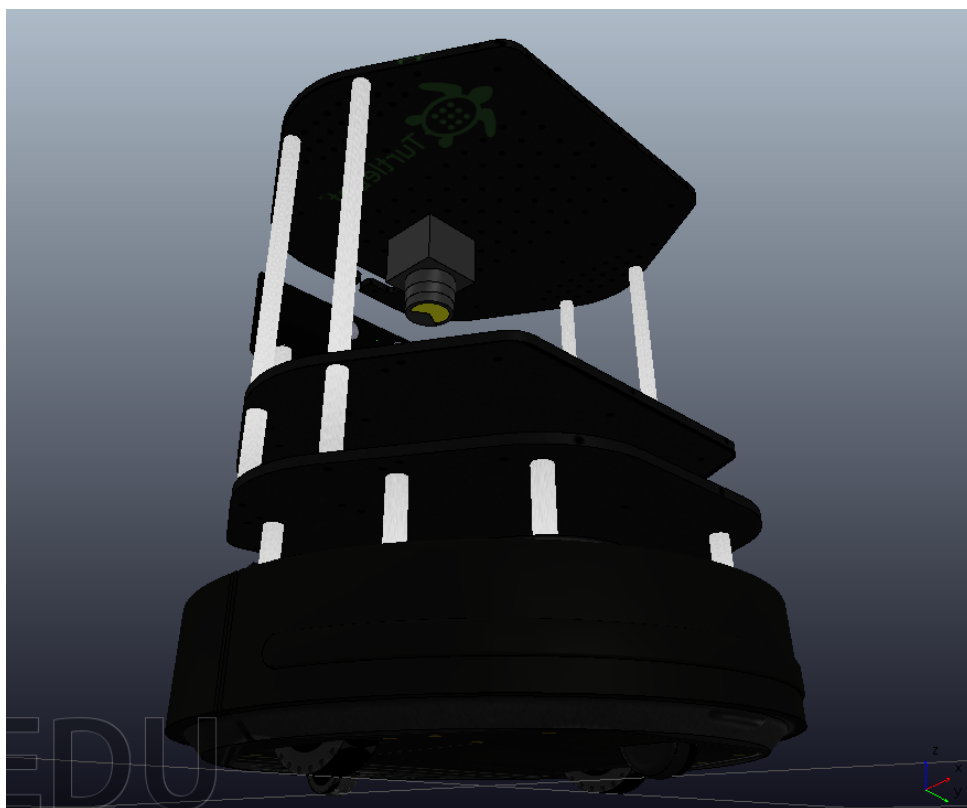


Figura 3.20: Sensor Hokuyo colocado en Turtlebot.

La ubicación es provisional; la definitiva dependerá de dónde sea posible en el robot real. Delante de Kinect nos ha parecido un buen sitio, porque además no resta ángulo de visión a este último. En la figura 3.20 puede verse dónde está colocado exactamente. Nótese que está colocado boca abajo, puesto que en la parte superior de esa “balda” de Turtlebot irá WidowX.

Por tanto, la ubicación definitiva dependerá de las posibilidades desde el punto de vista de la sujeción que tengamos una vez que intentemos colocar el sensor Hokuyo en el robot real. En cualquier caso, el hecho de cambiar el sensor de lugar es muy sencillo en V-REP usando las herramientas de mover, de la misma forma que hemos hecho hasta ahora.

Para acabar, podemos ver en la figura 3.21 la representación de los rayos del sensor (por defecto está deshabilitada), donde nos hacemos una idea del ángulo que es capaz de abarcar y de las zonas ciegas debidas a los soportes verticales de la plataforma.

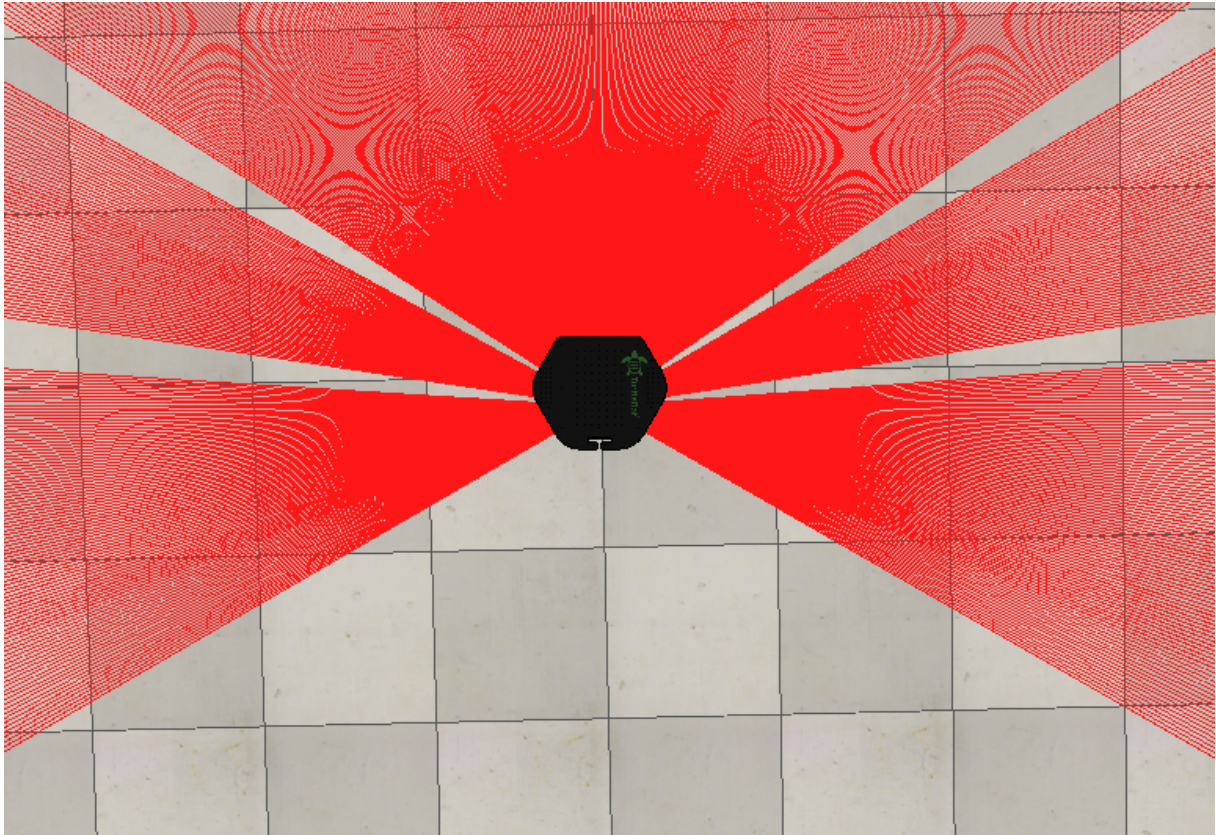


Figura 3.21: Ángulo de visión del sensor Hokuyo.

3.4. Añadiendo las texturas

Una forma muy sencilla de hacer nuestro modelo más vistoso es el empleo de texturas. Así, nuestro conjunto robótico dejará de ser un conjunto de formas (más o menos complejas) con un color sólido pasando a tener una apariencia más realista.

Por suerte, el paquete “turtlebot_description”, que fue del que partimos para el modelado de Turtlebot, incluye los ficheros de las texturas del mismo.

Añadir las a V-REP es un proceso bastante mecánico y simple, puesto que consiste en seleccionar cada una de las piezas de Turtlebot, abrir el cuadro de diálogo específico para la textura y ajustarla (escalarla y posicionarla) de forma que quede lo mejor posible en la pieza. Por ello, no merece la pena que expliquemos el proceso, más aún siendo una cuestión que no tiene relevancia en el comportamiento del modelo.

Las figuras 3.22 y 3.23 muestran los más que satisfactorios resultados obtenidos. Solo tenemos que comparar estas dos figuras con la 3.12 para ver cuánto ha mejorado visualmente nuestro modelo.



Figura 3.22: Vista frontal de Turtlebot con sus texturas.



Figura 3.23: Vista trasera de Turtlebot con sus texturas.

3.5. Importando el modelo de WidowX

Llegados a este punto, ya tenemos incluido definitivamente el modelo de Turtlebot en el simulador. En esta sección vamos a abordar el hecho de importar el brazo manipulador WidowX en V-REP, para posteriormente unirlo a Turtlebot y así tener el conjunto robótico completo.

El proceso de importación es muy parecido al que llevamos a cabo para Turtlebot. En este caso, partimos del paquete que incluye los elementos necesarios para trabajar en ROS con WidowX, incluyendo el apartado de la descripción física.

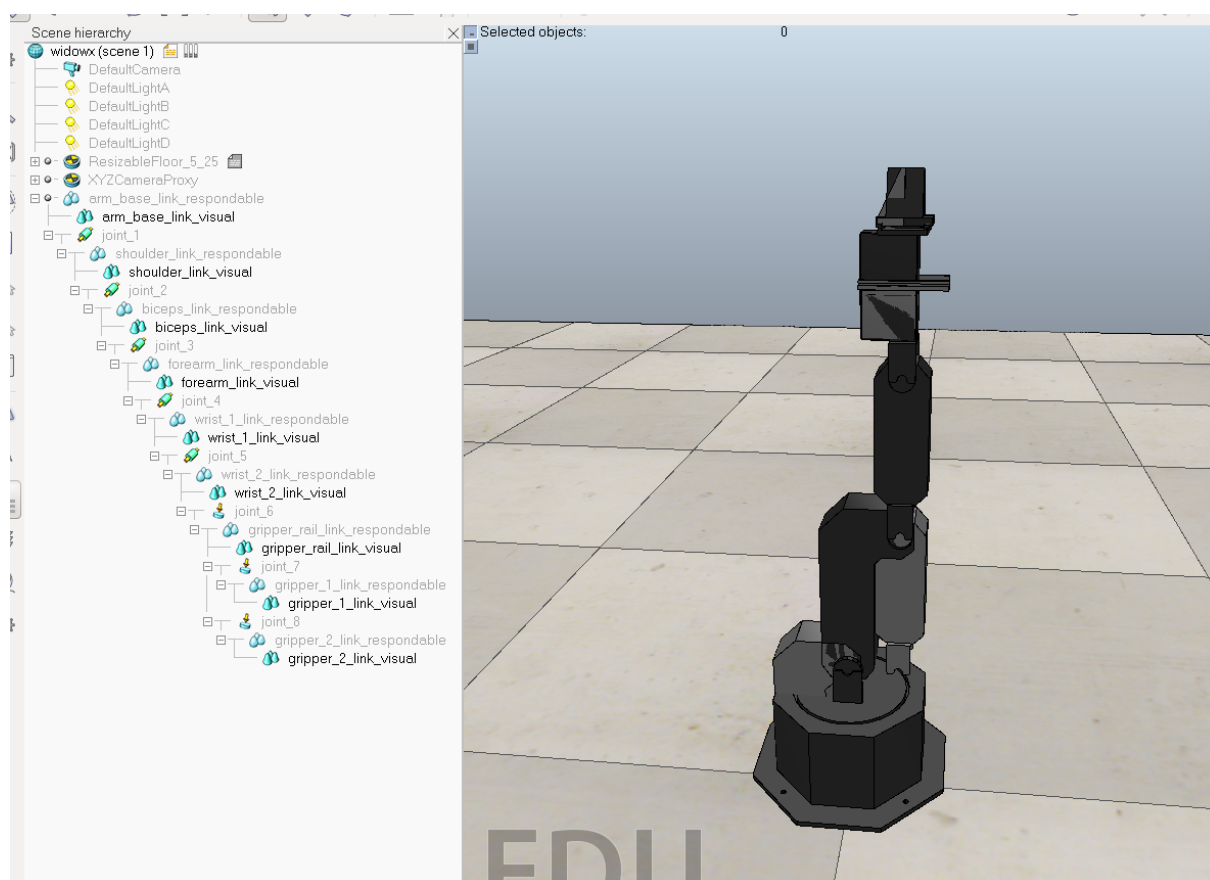


Figura 3.24: Primera importación de WidowX.

Este paquete nos lo podemos descargar del *github* de Robotnik. La forma más sencilla que hemos encontrado para importarlo en V-REP consiste en utilizar la máquina virtual con ROS. En esa máquina, hay que colocar el paquete descargado en el *path* correspondiente de ROS, que dependerá de la instalación que se haya hecho.

Si no colocamos el paquete en ese lugar, cuando V-REP intente importar el modelo no será capaz de encontrar los ficheros con los modelos en tres dimensiones de cada una de las piezas de WidowX.

Una vez preparada toda la información, podemos volver a usar el *plugin* de importación URDF que nos proporciona V-REP para importar este tipo de ficheros. Si todo ha ido

bien, nos encontraremos con algo parecido a la figura 3.24.

El resultado obtenido deja bastante que desear; no hace falta observar muy atentamente la figura para darse cuenta de que hay piezas que no están en su sitio. Más aún, hay piezas que están duplicadas (las de color gris) y que vamos a tener que eliminar de alguna manera.

En cuanto a la jerarquía del robot, vemos que también presenta algún que otro fallo. El más grave de ellos es que no está modelada la articulación que abre y cierra la pinza, así que vamos a tener que solucionar eso también.

Vamos a comenzar por arreglar la duplicidad y la posición de algunas piezas. Para ello, seguimos un orden ascendente en el brazo. Con respecto a la base y al hombro no encontramos nada raro, así que proseguimos ascendiendo.

Llegados al bíceps, nos encontramos el primer problema. Por un lado, tenemos que la parte *respondable* está conectada de manera errónea a la articulación del hombro. En concreto, está conectada por la parte superior, cuando realmente debería estarlo por la inferior. Por otro lado, tenemos que la parte visual está bien colocada, pero su *shape* está formada tanto por la parte visual como por la parte *respondable* (es decir, esta última está repetida).

Esto podemos verlo más claro en la figura 3.25, donde observamos que la *shape* visual del bíceps está compuesta por dos componentes.

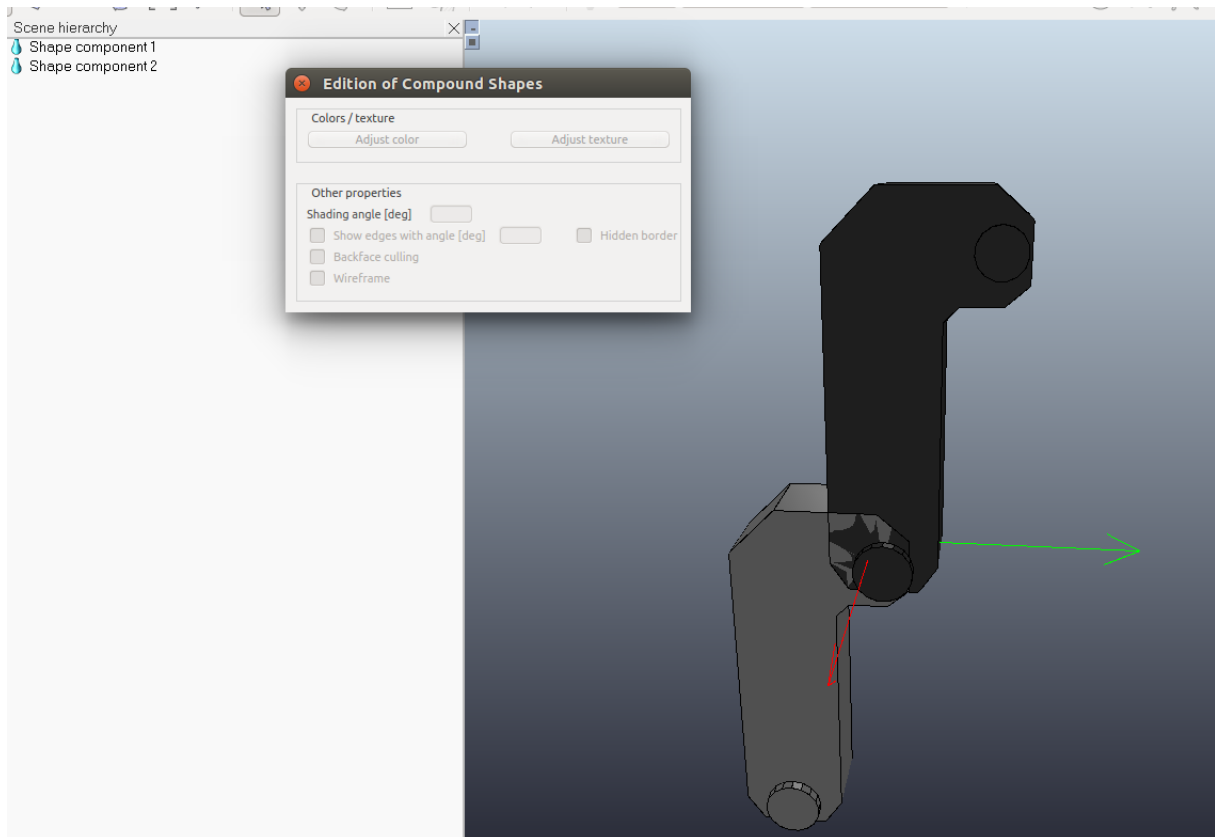


Figura 3.25: Duplicidad en una *shape* de WidowX.

Una forma de proceder consistiría en dividir esa *shape* en sus dos componentes, y eliminar el que nos sobra, que es el gris de abajo.

Una vez hecho eso, lo que nos queda es colocar en su sitio la *shape responsable* del bíceps. Para hacerlo de la forma más precisa posible, hemos de copiar la posición de la parte visual y establecérsela a la parte *responsable*.

Hemos de repetir este proceso para cada una de las piezas de WidowX que estén mal colocadas y repetidas. Puesto que es un proceso bastante mecánico, merece la pena plasmar en esta memoria cada paso.

Una vez hecho eso, deberíamos obtener un resultado parecido al de la figura 3.26.

Otra de las cuestiones que debemos solucionar en el modelo de WidowX es la pinza. Como dijimos anteriormente, resulta que no está modelada la articulación que abre y cierra la pinza.

Para obtener resultados más fiables, vamos a echar un vistazo a la librería de modelos de V-REP a ver si encontramos algún brazo parecido al nuestro, y así imitar su pinza. El más parecido que nos encontramos es el PhantomX Pincher, cuya pinza es prácticamente igual a la de nuestro brazo WidowX.

En la figura 3.27 podemos observar a PhantomX (a la izquierda) junto a WidowX (a la derecha). En esa misma figura aparece la jerarquía de la escena, donde PhantomX tiene correctamente modelada la articulación de la pinza.

Gracias a las herramientas de V-REP, no es muy complicado copiar la pinza de PhantomX a WidowX. El poder hacer esto nos garantizará que los resultados van a ser buenos, puesto que se trata de una implementación incluida en la librería del simulador.

En cuanto a la implementación en sí, está formada por dos articulaciones. Recordemos que V-REP nos proporciona articulaciones prismáticas, esféricas y rotacionales. En principio, no existe manera de utilizar una sola articulación para modelar el cierre de una pinza de “dos dedos”. Por ello, la solución consiste en usar dos articulaciones prismáticas, una dependiente de la otra. Así pues, cuando se cierre un lado de la pinza (esto lo controlaremos con la articulación “gripper_1_joint”) el otro se cerrará de forma simétrica, automáticamente.

En la figura 3.28 podemos observar la implementación definitiva de la pinza², donde aparecen tanto las dos partes *responsables* de la misma (en celeste) junto a las dos articulaciones prismáticas (naranja con el eje azul).

Estas articulaciones constan de un pequeño *script* asociado para su control. El contenido del mismo lo describiremos en la sección siguiente de esta memoria.

Para acabar, solo nos queda colocar a WidowX sobre Turtlebot (figura 3.29). Lo haremos añadiendo un sensor de fuerza que actúa como nexo de unión entre WidowX y la

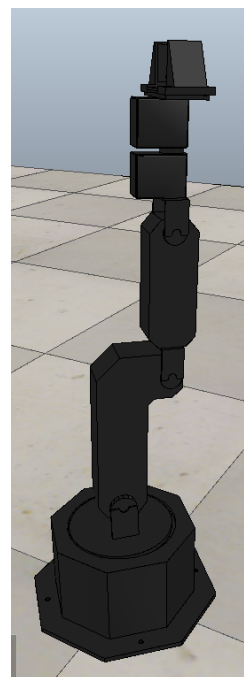


Figura 3.26: WidowX.

²Evidentemente, esta vista de la misma está oculta en condiciones normales.

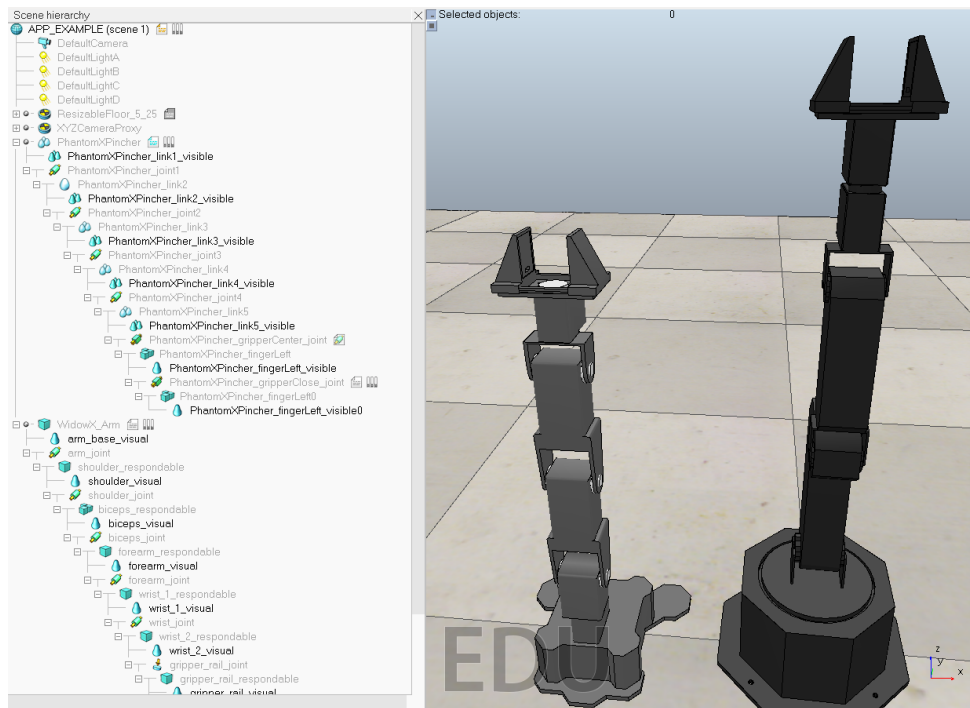


Figura 3.27: WidowX junto a PhantomX.

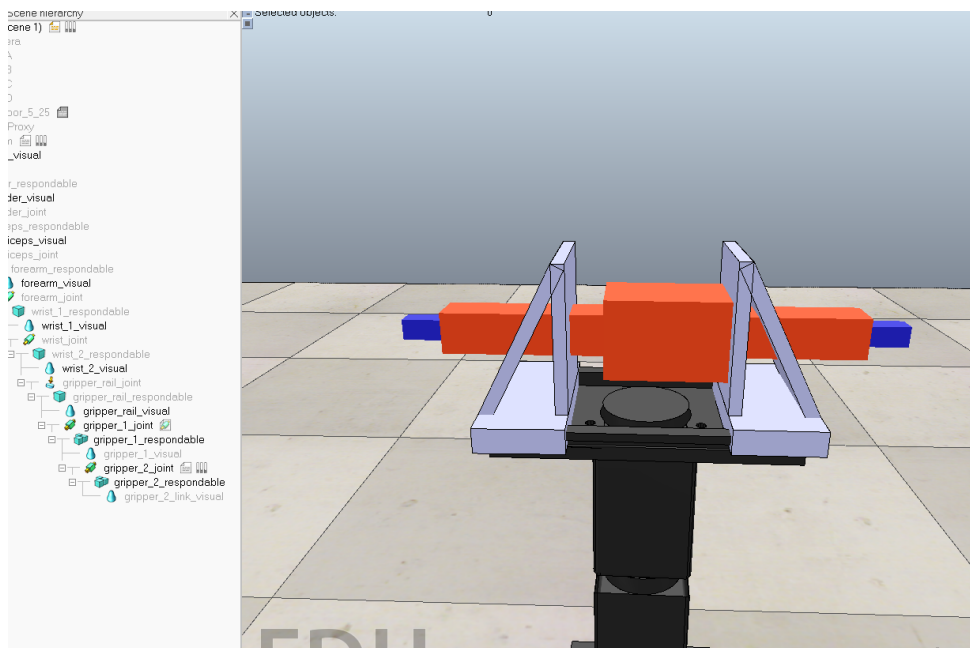


Figura 3.28: Detalle de la implementación de la pinza en WidowX.

“balda” superior de Turtlebot y colocamos a WidowX en el centro de la misma. Además, hemos establecido las articulaciones de WidowX en su posición de reposo.

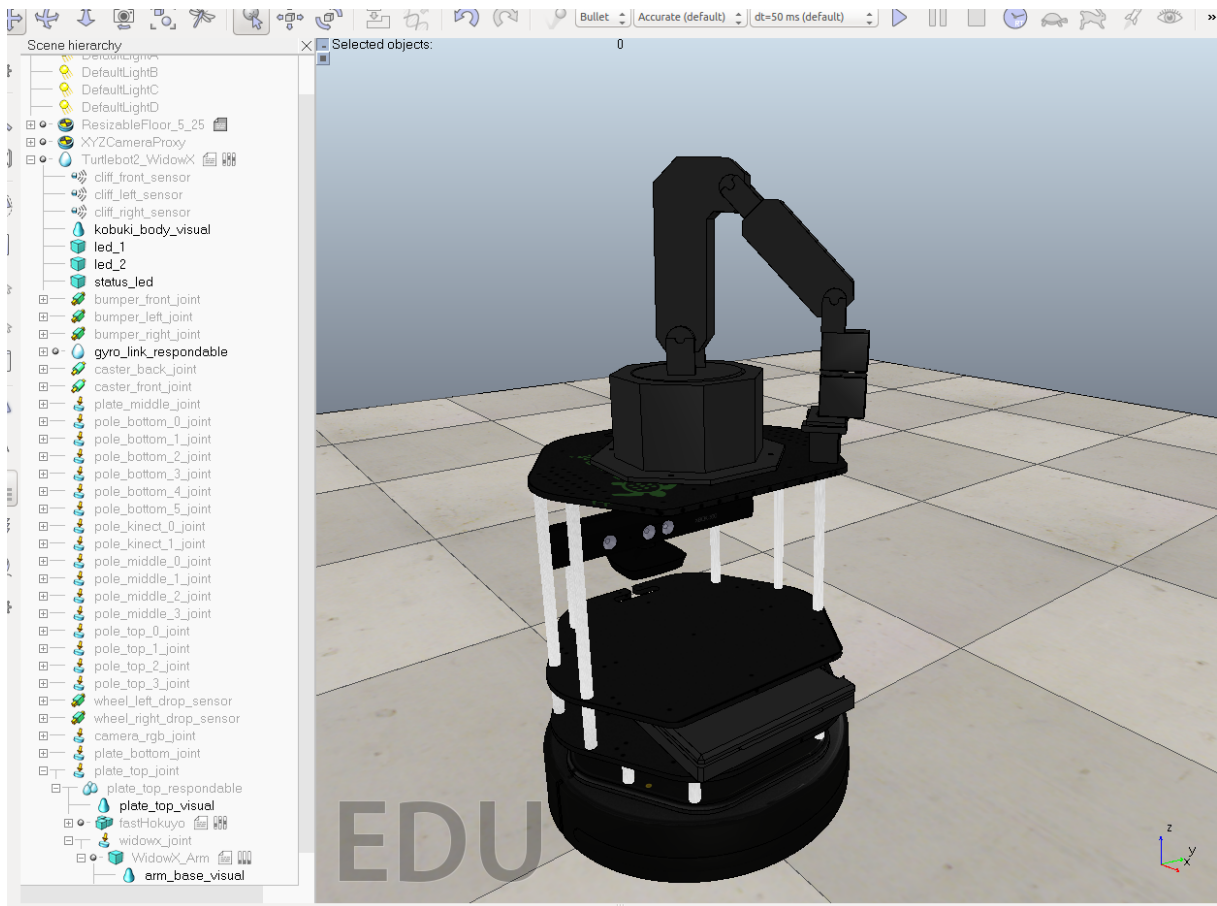


Figura 3.29: Turtlebot con WidowX colocado.

3.6. Generando los *scripts* del modelo

Llegados a este punto, tenemos definida la parte visual y dinámica de nuestro robot en V-REP. Aún nos queda una cuestión muy importante: cómo controlar y recibir información del robot.

Para hacer esto, uno de los caminos que podemos tomar consiste en emplear *scripts* asociados al modelo. Estos *scripts* se ejecutarán en cada paso de la simulación, a una frecuencia definida por el usuario³.

Los *scripts* suelen tener cuatro partes:

- *Initialization*. Se ejecuta solo al comienzo de la simulación.
- *Sensing*. Se ejecuta en cada pase de la simulación, y es donde se suelen tomar los datos de los sensores.
- *Actuation*. Se ejecuta en cada paso de la simulación, y es el fragmento de código encargado de tomar las decisiones de actuación pertinentes (por ejemplo, mandar comandos de velocidad a los motores).

³En el manual de usuario está explicado este concepto más detalladamente.

- *Cleanup*. Se ejecuta solo cuando finaliza la simulación, y es donde se debe hacer la “limpieza”. Ejemplo de ello podría ser cerrar el *socket* de la API remota.

El lenguaje de programación es Lua, y en estos *scripts* podemos incluir, en principio, el código que queramos. Dentro de las cuestiones que podemos añadir se encuentran las llamadas a las funciones de la API de V-REP, entre las que nos encontramos funciones tanto para leer sensores como para mandar comandos a articulaciones.

Recordemos que el objetivo de este trabajo es poder controlar el conjunto robótico remotamente desde MATLAB, así que debemos desarrollar el código de los *scripts* teniendo en cuenta el diseño de la *toolbox*, cosa de lo que hablaremos largo y tendido en el siguiente capítulo de la memoria.

Vamos a comenzar con la descripción del *script* asociado a Turtlebot.

3.6.1. Turtlebot

El *script* de Turtlebot es el más extenso del modelo, puesto que la mayor parte de los sensores se encuentran en esta parte del conjunto robótico. Para abrir el *script* hay que hacer doble clic en el icono con forma de hoja de papel que existe sobre la base del modelo, en la jerarquía.

Como no podría ser de otra manera, lo primero que nos encontramos es con la parte de inicialización. Los *scripts* admiten parámetros (o argumentos, como queramos llamarlos). Estos parámetros permiten realizar cambios en el comportamiento del modelo de manera muy simple.

Para establecerlos, hacemos doble clic en el icono de los tres “potenciómetros” que existe justo al lado del botón para acceder al *script*.

En el caso que nos ocupa, solo tenemos un parámetro. Establece si queremos que la API remota esté habilitada o no. Por defecto sí lo está. Para obtener el estado de este parámetro en el código, procedemos tal y como muestra la llamada a función de V-REP en el fragmento de código que se muestra a continuación.

```

if (sim_call_type == sim_childscriptcall_initialization) then
    remoteAPI = simGetScriptSimulationParameter(sim_handle_self,
        'remoteAPIEnabled')

    local controller_type = 0
    local controller_p    = 100.000007629
    local controller_i    = 0.10000000149
    local controller_d    = 2.0
    local hardware_ver    = {1, 0, 4}
    local firmware_ver    = {1, 2, 0}
    local software_ver    = {0, 6, 0}
    local udid            = {98434864, 959859523, 1126263088}
    local features        = 3

```

En ese fragmento también se muestra la inicialización de varias constantes, asociadas a valores definidos en el robot real. Estos valores carecen de significado en el modelo, pero nos servirán más adelante para que los *topics* en MATLAB tengan todos los valores correspondientes.

Seguidamente, nos encontramos con la definición de los colores para los dos LEDs configurables de la base Kobuki. El color “black_Led” se emplea cuando el LED está apagado. También nos encontramos con una colección de variables que nos ayudarán a definir el comportamiento del modelo, y que iremos describiendo según nos hagan falta:

```

blackLed = {0.4, 0.4, 0.4}
greenLed = {0, 1, 0}
orangeLed = {1, 0.5, 0}
redLed = {1, 0, 0}

lastTimeActuation = 0
previousRWPosition = 0
previousLWPosition = 0
totalRWPosition = 0
totalLWPosition = 0
lastSpeedCommand = 0
lastSpeedCommandCtr = 0
led_1_old_state = 0
led_2_old_state = 0
button_0 = 0
button_1 = 0
button_2 = 0

```

Almacenamos el manejador del modelo, puesto que nos hará falta más adelante para algunas cuestiones:

```

turtleWidowX = simGetObjectAssociatedWithScript(sim_handle_self)
modelBase = simGetObjectHandle('Turtlebot2_WidowX')
modelBaseName = simGetObjectHandle(modelBase)

```

Vamos a necesitar más manejadores para poder trabajar con los distintos sensores y actuadores del modelo. El siguiente fragmento de código muestra la obtención de los manejadores de la base Kobuki, donde se incluyen los motores de las dos ruedas, los LEDs, los *bumpers*, los *cliffs*, los *wheels drop* y el giroscopio:

```

— Kobuki Wheels
t_rightWheel = simGetObjectHandle('wheel_right_joint')
t_leftWheel = simGetObjectHandle('wheel_left_joint')

— Kobuki LEDs
t_statusLed = simGetObjectHandle('status_led')
t_led_1 = simGetObjectHandle('led_1')
t_led_2 = simGetObjectHandle('led_2')

```



```

— Kobuki Bumpers
t_frontBumper = simGetObjectHandle('bumper_front_joint')
t_rightBumper = simGetObjectHandle('bumper_right_joint')
t_leftBumper  = simGetObjectHandle('bumper_left_joint')

— Kobuki Cliff Sensors
t_frontCliff  = simGetObjectHandle('cliff_front_sensor')
t_rightCliff  = simGetObjectHandle('cliff_right_sensor')
t_leftCliff   = simGetObjectHandle('cliff_left_sensor')

— Kobuki Wheel drop sensors
t_rightWheelDrop = simGetObjectHandle('wheel_right_drop_sensor')
t_leftWheelDrop  = simGetObjectHandle('wheel_left_drop_sensor')

— Kobuki Gyroscope
ref              = simGetObjectHandle('gyro_link_visual')
lastTime        = simGetSimulationTime()
oldTransformationMatrix = simGetObjectMatrix(ref, -1)

```

Además, inicializamos la matriz que empleamos en el giroscopio para calcular cuánto ha girado la base Kobuki con respecto al pase de simulación anterior.

Para acabar con la inicialización de los manejadores, adquirimos los correspondientes al sensor Kinect y al brazo WidowX. Para el caso de Kinect, primero comprobamos que el usuario no ha deshabilitado los sensores de visión, en cuyo escenario no habría que obtener los manejadores.

Con respecto a WidowX, tomamos los manejadores de las cinco primeras articulaciones. La de la pinza no la adquirimos, puesto que como dijimos anteriormente, existe un *script* específico para manejarla y la forma de enviarle comandos es diferente.

```

— Kinect
if(simGetBoolParameter(sim_boolparam_vision_sensor_handling_enabled)
    == true) then
    depthCam = simGetObjectHandle('kinect_depth')
    depthView = simFloatingViewAdd(0.9, 0.9, 0.2, 0.2, 0)
    simAdjustView(depthView, depthCam, 64)
    colorCam = simGetObjectHandle('kinect_rgb')
    colorView = simFloatingViewAdd(0.69, 0.9, 0.2, 0.2, 0)
    simAdjustView(colorView, colorCam, 64)
end

— WidowX Joints
w_arm      = simGetObjectHandle('arm_joint')
w_shoulder = simGetObjectHandle('shoulder_joint')
w_biceps   = simGetObjectHandle('biceps_joint')
w_forearm  = simGetObjectHandle('forearm_joint')
w_wrist    = simGetObjectHandle('wrist_joint')

```

Para comunicar el modelo con V-REP, vamos a emplear señales. Las señales pueden verse como variables globales a las cuales se les puede asignar valores, leer esos valores y limpiarlas. Son muy fáciles de implementar, y lo mejor es que si tenemos que guardar un conjunto de valores, podemos empaquetarlos en una señal de tipo *string*. Por ello, este será el tipo más empleado.

El siguiente trozo de código muestra las señales asociadas a los sensores. Por un lado, tenemos una que contiene los estados de todos los sensores, otra que almacena la *pose* y la orientación (en forma de cuaternión), otra que guarda los estados de las orientaciones y otra que permite reiniciar la odometría. También tenemos una señal de tipo *float* que contiene el tiempo de simulación:

```

— Sensing
  — All sensors
  simSetStringSignal(modelBaseName..'_kobuki_sensor_state', 'empty')

  — Pose and Quaternion
  simSetStringSignal(modelBaseName..'_kobuki_quaternion_vel_accel',
    'empty')

  — Joint states
  simSetStringSignal(modelBaseName..'_joint_states', 'empty')

  — Simulation Time
  simSetFloatSignal(modelBaseName..'_simulation_time', 0)

  — Odometry
  simSetIntegerSignal(modelBaseName..'_kobuki_odometry_reset', 0)

```

También disponemos de señales para tareas de actuación, como pueden ser los dos motores de la base Kobuki (que se inicializan a velocidad cero, por supuesto) y los LEDs, que se inicializan en estado apagado.

Para acabar con las señales, definimos dos en base a las constantes que definimos al principio del *script*. Una contiene la información del controlador de Kobuki y la otra la versión hardware y software de la misma. Tal y como dijimos, estas señales se incluyen por completitud y carecen de significado en el modelo. Simplemente nos ayudarán a la emulación de los *topics* en nuestra *toolbox* de MATLAB:

```

— Actuation
  — Kobuki Wheels
  simSetIntegerSignal(modelBaseName..'_kobuki_motors_enabled', 1)
  wheelSpeedData = simPackFloats({0, 0, 0}, 0, 3)
  simSetStringSignal(modelBaseName..'_kobuki_wheels_speed',
    wheelSpeedData)

  — Kobuki LEDs
  simSetIntegerSignal(modelBaseName..'_kobuki_led_1', 0)

```

```
simSetIntegerSignal(modelBaseName.. '_kobuki_led_2', 0)
```

— Other

```
local controllerInfoData = simPackFloats({ controller_type ,
controller_p , controller_i , controller_d }, 0, 4)
simSetStringSignal(modelBaseName.. '_kobuki_controller_info' ,
controllerInfoData )
```

```
local versionInfoData = simPackFloats({ hardware_ver[1] ,
hardware_ver[2] , hardware_ver[3] , firmware_ver[1] ,
firmware_ver[2] , firmware_ver[3] , software_ver[1] ,
software_ver[2] , software_ver[3] , udid[1] , udid[2] , udid[3] ,
features }, 0, 13);
simSetStringSignal(modelBaseName.. '_kobuki_version_info' ,
controllerInfoData )
```

Una cuestión sobre la que no hemos hablado aún es de que también hemos añadido a nuestro conjunto robótico una serie de interfaces de usuario en la simulación.

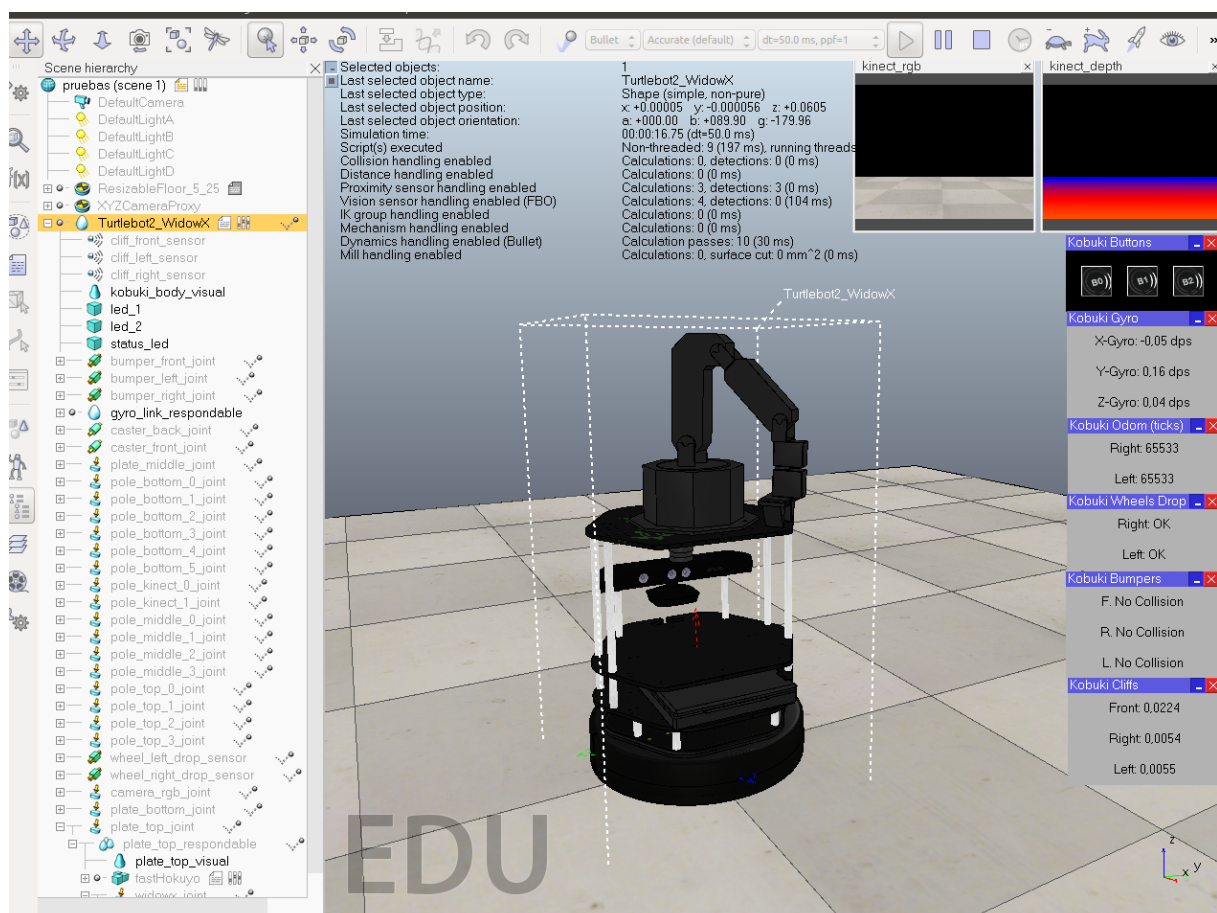


Figura 3.30: Interfaces de usuario asociadas al modelo.

Estas interfaces de usuario (ver figura 3.30), solo visibles en la simulación y si el

conjunto robótico está seleccionado, nos permiten conocer ciertos valores y estados de los sensores en tiempo real. Concretamente, tenemos las siguientes interfaces:

- *Kobuki Buttons*. Consta de tres botones, emulando los tres botones que existen en la base Kobuki del robot real.
- *Kobuki Odom*. Muestra los datos de los dos *encoders* de los motores de la base Kobuki. Los valores se encuentran en el rango [0 - 65535] circular, y su unidad es el *tick*. Por cada milímetro lineal avanzado, los *encoders* aumentan 11,7 *ticks*.
- *Kobuki Wheels Drop*. Muestra los estados de los dos sensores *wheel drop* de Kobuki.
- *Kobuki Bumpers*. Muestra los estados de los tres *bumpers* de Kobuki.
- *Kobuki Cliffs*. Muestra los estados de los tres sensores *cliff* de Kobuki.

A continuación se muestra el código que captura los manejadores de cada interfaz:

```
— Buttons
buttons_ui = simGetUIHandle( 'Kobuki_Buttons_UI' )

— Bumpers
bumpers_ui = simGetUIHandle( 'Kobuki_Bumper_sensor_UI' )

— Cliff Sensors
cliff_ui = simGetUIHandle( 'Kobuki_Cliff_sensor_UI' )

— Wheel drop
wheeldrop_ui = simGetUIHandle( 'Kobuki_Wheel_drop_UI' )

— Odometry
odometry_ui = simGetUIHandle( 'Kobuki_Odometry_UI' )

— Gyroscope
gyro_ui = simGetUIHandle( 'Kobuki_Gyro_sensor_UI' )
```

Para acabar la fase de inicialización del *script*, inicializamos el estado de los LEDs, poniendo el de estado en color verde y los otros dos apagados. También ponemos en funcionamiento la API remota, si corresponde. Además, guardamos la posición inicial del robot para tomarla como punto de partida, e inicializamos la *pose* del robot:

```
— INITIALIZE LEDs —
    simSetShapeColor( t_statusLed , nil , 0 , greenLed )
    simSetShapeColor( t_led_1 , nil , 0 , blackLed )
    simSetShapeColor( t_led_2 , nil , 0 , blackLed )

— REMOTE API —
    if( remoteAPI ) then
        simExtRemoteApiStart( 19999 , 1300 , false , false )
```

```

                end

    — SAVING INITIAL POSITION AND ORIENTATION —
        originPosition = simGetObjectPosition(ref, -1)
        originOrientation = oldTransformationMatrix

    — INITIAL SIMULATED POSITION AND ORIENTATION —
        pose_x = 0
        pose_y = 0
        pose_tita = 0
        wheel_bias = 0.23
        wheel_radius = 0.035
        lastScriptCallTime = simGetSimulationTime()
end

```

El siguiente trozo de código se corresponde con la fase de limpieza o *cleanup* del *script*. Aquí tenemos dos cuestiones que abordar. Si la API remota ha sido activada, debemos pararla. Por otro lado, reestablecemos el estado de los LEDs a su estado de apagado, tal y como si hubiéramos apagado el robot real:

```

if (sim_call_type == sim_childscriptcall_cleanup) then

    — REMOTE API —
        if(remoteAPI) then
            simExtRemoteApiStop(19999)
        end

    — Shutdown LEDs —
        simSetShapeColor(t_statusLed, nil, 0, blackLed)
        simSetShapeColor(t_led_1, nil, 0, blackLed)
        simSetShapeColor(t_led_2, nil, 0, blackLed)
end

```

Lo siguiente que nos encontramos en el *script* es la fase de *sensing*, que es en la que vamos a obtener los valores de los sensores, entre otras cosas.

Dentro de esta fase, lo primero que nos encontramos es el código asociado al giroscopio. El principio de funcionamiento no es muy complicado: se toma la matriz asociada al giroscopio en el momento actual de simulación. Acto seguido, se calcula la inversa de la matriz anterior y se multiplican ambas. Estas matrices contienen la rotación del giroscopio con respecto al sistema universal en un momento dado, así que al multiplicarlas entre sí obtenemos la matriz de rotación entre un pase de la simulación (el actual) y el anterior. A partir de ella podemos obtener los tres ángulos, correspondientes a cada eje. Seguidamente, comprobamos que han pasado más de 0.25 segundos desde la última medición, para que los valores en la interfaz de usuario no se actualicen demasiado rápido y sean difíciles de leer. Al dividir los ángulos calculados entre el tiempo transcurrido, obtenemos las velocidades angulares en cada uno de los ejes, y las mostramos en la interfaz de usuario.

```

if (sim_call_type == sim_childscriptcall_sensing) then
  — Gyroscope
  local transformationMatrix = simGetObjectMatrix(ref, -1)
  local oldInverse          =
    simGetInvertedMatrix(oldTransformationMatrix)
  local m                   = simMultiplyMatrices(oldInverse,
    transformationMatrix)
  local euler               = simGetEulerAnglesFromMatrix(m)
  local currentTime        = simGetSimulationTime()

  if(currentTime - lastTime > 0.25) then
    local gyroData = {0,0,0}
    local dt       = currentTime - lastTime
    if (dt ~= 0) then
      gyroData[1] = euler[1] / dt
      gyroData[2] = euler[2] / dt
      gyroData[3] = euler[3] / dt
    end

    simSetUIButtonLabel(gyro_ui, 3, string.format(
      "X-Gyro: %2f dps ", ((gyroData[1] * 180) / math.pi)))

    simSetUIButtonLabel(gyro_ui, 4, string.format(
      "Y-Gyro: %2f dps ", ((gyroData[2] * 180) / math.pi)))

    simSetUIButtonLabel(gyro_ui, 5, string.format(
      "Z-Gyro: %2f dps ", ((gyroData[3] * 180) / math.pi)))

    oldTransformationMatrix = simCopyMatrix(transformationMatrix)
    lastTime = currentTime
  end

```

Lo siguiente que nos encontramos es el cálculo de la orientación. Partimos de la inversa de la matriz de origen (la que tomamos en la fase de inicialización) y la multiplicamos por la matriz actual. Así conseguimos la matriz de rotación de nuestro robot con respecto al punto de partida. A partir de esta matriz, podemos obtener el cuaternión usando una de las funciones de la API de V-REP para tal fin. Por otro lado, obtenemos la velocidad lineal y angular, tomando como referencia el giroscopio (cuyo manejador es *ref*). Además, establecemos la aceleración lineal en '0' para cada uno de los ejes, puesto que el robot real no incluye acelerómetro:

```

— Orientation
oldOrientationInverse = simGetInvertedMatrix(originOrientation)
matrix_quat          =
  simMultiplyMatrices(oldOrientationInverse, transformationMatrix)

quaternion           = simGetQuaternionFromMatrix(matrix_quat)

```

```

— Speed
  linearSpeed , angularSpeed = simGetObjectVelocity(ref)

— Acceleration
  linearAcceleration = {0, 0, 0}

```

Lo que nos espera ahora es el cálculo de los valores de los *encoders*. Puesto que el cálculo es idéntico para los dos motores de la base Kobuki, solo vamos a mostrar el de la rueda derecha para evitar ser reiterativos:

```

— Right
dr = simGetJointPosition(t_rightWheel) - previousRWPosition
if (dr >= 0) then
  dr_aux = math.mod(dr + math.pi, 2 * math.pi) - math.pi
else
  dr_aux = math.mod(dr - math.pi, 2 * math.pi) + math.pi
end

totalRWPosition = totalRWPosition + dr_aux
previousRWPosition = simGetJointPosition(t_rightWheel)
RWdegrees = (180 * totalRWPosition) / math.pi
RWTicks = RWdegrees * (2578.333869740464 / 360)
RWTicks = math.mod(RWTicks, 65535)
if(RWTicks < 0) then
  RWTicks = RWTicks + 65535
end

simSetUIButtonLabel(odometry_ui, 6,
  string.format("Right: %d", RWTicks))
[...]
```

El procedimiento para el cálculo de los valores de los *encoders* es el siguiente:

1. Calculamos la diferencia de ángulo entre la posición actual del motor y la posición anterior
2. Ajustamos el ángulo anterior al intervalo $[-\pi, \pi]$
3. Calculamos la suma del ángulo girado con el ángulo girado previamente (en radianes)
4. Pasamos el ángulo anterior a grados
5. Calculamos los *ticks* usando la correspondencia grados-*ticks* que aparece en las especificaciones de Kobuki.
6. Ajustamos los *ticks* para que estén dentro del rango $[0, 65535]$

Para acabar, se actualiza el valor en el campo correspondiente de la interfaz.

Lo siguiente que nos encontramos en el código es el cálculo de la posición y la orientación de nuestro conjunto robótico. Para imitar lo máximo posible al robot real, vamos a calcular la posición en base a los valores de los encoders y la orientación la vamos a obtener a partir del giroscopio.

Comencemos por la *pose*. En nuestro robot, va a tomar dos variables: *pose* sobre el eje X y *pose* sobre el eje Y. Las expresiones para calcularlas son las siguientes:

$$pose_x = pose_x_anterior + (\Delta tiempo * media_vel_motores * \cos(pose_tita))$$

$$pose_y = pose_y_anterior + (\Delta tiempo * media_vel_motores * \sin(pose_tita))$$

```

— Simulated position and orientation
currentTime = simGetSimulationTime()
if(currentTime > 0.5) then
    local t_inc = (currentTime - lastScriptCallTime)
    lastScriptCallTime = currentTime
    local rightWheel_vel = (dr_aux / t_inc) * wheel_radius
    local leftWheel_vel = (dl_aux / t_inc) * wheel_radius

    pose_x = pose_x +
    (t_inc * ((rightWheel_vel + leftWheel_vel) / 2) *
    math.cos(pose_tita))

    pose_y = pose_y +
    (t_inc * ((rightWheel_vel + leftWheel_vel) / 2) *
    math.sin(pose_tita))

    euler_orientation = simGetEulerAnglesFromMatrix(matrix_quat)
    pose_tita = euler_orientation[3] — From gyroscope
end

```

En el fragmento de código anterior también puede verse cómo se obtiene *tita* a partir de la matriz usada para el cálculo del cuaternión. Las velocidades se calculan con los diferenciales de tiempo, en vez de con la función de la API de V-REP existente para ello. La idea es, como hasta ahora hemos venido haciendo, imitar al robot real lo máximo posible.

A continuación nos encontramos el fragmento de código asociado a uno de los *bumpers*. Puesto que el procedimiento es idéntico para cada uno de los tres que tiene la base Kobuki, mostramos solo uno para evitar ser reiterativos.

El código es bastante simple gracias a las funciones de la API de V-REP. Recordemos que los *bumpers* están modelados como articulaciones prismáticas, así que van a tener un movimiento lineal. Para comprobar si el robot ha chocado o no, leemos el estado de esta articulación, y si el resultado es que su posición ha retrocedido un milímetro, entendemos que se ha producido el choque.

En cualquier caso, actualizamos el estado de la interfaz gráfica correspondiente, además de la variable de estado asociada al *bumper* en cuestión. Esta variable de estado será

utilizada más adelante para generar las señales que emplearemos en la comunicación con MATLAB:

```
— Front Bumper
front_bumper_pos = simGetJointPosition(t_frontBumper)
if(front_bumper_pos < -0.001) then
    simSetUIButtonLabel(bumpers_ui, 6,
        string.format("F. COLLISION!"))
    bumperCenterState = 1
else
    simSetUIButtonLabel(bumpers_ui, 6,
        string.format("F. No Collision"))
    bumperCenterState = 0
end

[...]
```

Con respecto a los sensores *cliff*, nos encontramos con el código del siguiente recuadro. De nuevo, solo mostramos el fragmento asociado al *cliff* frontal:

```
— Front Cliff
resultFC, distanceFC = simReadProximitySensor(t_frontCliff)
if(resultFC == 1) then
    simSetUIButtonLabel(cliff_ui, 6,
        string.format("Front: %4f", distanceFC))
    frontCliffState = 0
else
    simSetUIButtonLabel(cliff_ui, 6,
        string.format("Front: CLIFF!"))
    frontCliffState = 1
    distanceFC = 4096
end

[...]
```

En este caso, empleamos la función que nos proporciona V-REP, denominada “simReadProximitySensor”, y que es específica para la lectura de sensores de proximidad, como los del caso que nos ocupa. Esta función nos devuelve dos valores. El primero de ellos nos dice si la medición se ha efectuado correctamente. Un caso típico de medición incorrecta es el que el conjunto robótico se acerque al precipicio de la escena. El segundo nos devuelve la distancia en metros hasta el suelo.

Vemos que en el código también actualizamos los valores de los campos de la interfaz gráfica correspondiente a los *cliffs*.

Lo siguiente con lo que nos encontramos es con los sensores *wheel drop*. Puesto que estos sensores están modelados de una forma muy similar a los *bumpers*, el procedimiento para consultar su estado también va a ser muy parecido. Por ello, para comprobar que el robot se ha levantado del suelo, comprobamos la posición de la articulación correspondiente. Si

esta ha retrocedido (es decir, la rueda ha bajado con respecto a la base Kobuki), entonces interpretamos que el robot ha sido levantado del suelo. En otro caso, el robot está en su estado “normal”. Sea cual sea el resultado de la comprobación, mostramos en la interfaz de usuario el estado correspondiente y hacemos lo propio con la variable de control:

```
— Wheel Drop
  — Right
  wheel_drop_right_pos = simGetJointPosition(t_rightWheelDrop)
  if(wheel_drop_right_pos < -0.001) then
    simSetUIButtonLabel(wheeldrop_ui, 6,
      string.format("Right: DROP!"))
    wheel_drop_right_state = 1
  else
    simSetUIButtonLabel(wheeldrop_ui, 6,
      string.format("Right: OK"))
    wheel_drop_right_state = 0
  end

[...]
```

El fragmento de código que se muestra a continuación se encarga de comprobar el estado de los botones. Cuando se lanza la simulación, el estado de todos los botones es ‘0’, es decir, todos están sin pulsar. Esto va a ser siempre así puesto que los botones están ocultos cuando la simulación está parada.

Cada vez que se pulsa o se suelta un botón se produce un evento. Así pues, lo que podemos hacer es comprobar si hay algún evento nuevo en cada pase de la simulación. En caso afirmativo, buscaremos a qué botón corresponde, y permutaremos el estado de su variable asociada. Debido a la velocidad con la que se ejecuta la simulación, es muy improbable que se produzcan dos eventos al mismo tiempo (de hecho, no es posible pulsar dos botones a la vez con el ratón).

```
— Buttons
  button = simGetUIEventButton(buttons_ui)
  if (button == 3) then
    if(button_0 == 0) then
      button_0 = 1
    else
      button_0 = 0
    end
  elseif (button == 4) then
    if(button_1 == 0) then
      button_1 = 1
    else
      button_1 = 0
    end
  elseif (button == 5) then
    if(button_2 == 0) then
```

```

        button_2 = 1
    else
        button_2 = 0
    end
end
end

```

Otra de las cuestiones que tenemos que abordar en la etapa de *sensing* es el hecho de almacenar el tiempo actual de simulación, ya que lo emplearemos en algunas señales a la hora de empaquetar los datos.

Por otro lado, tenemos que guardar el estado actual de cada una de las articulaciones de nuestro conjunto robótico. En el fragmento de código siguiente se muestra la captura de una de ellas (concretamente, de la rueda izquierda de Kobuki). El procedimiento es análogo para cada una de las articulaciones, así que evitamos mostrar todo el código para evitar ser reiterativos:

```

— SIMULATION TIME —
simTime = simGetSimulationTime()

— JOINT STATES —
wlj_pos = simGetJointPosition(t_leftWheel);
_, wlj_vel = simGetObjectFloatParameter(t_leftWheel,
    sim_jointfloatparam_velocity);
wlj_eff = -1;

[...]

```

Obsérvese que obtenemos dos parámetros asociados a cada articulación. Por un lado, la posición (con la ayuda de la función *simGetJointPosition*), y por otro, la velocidad (llamando a *simGetObjectFloatParameter* con el parámetro correspondiente). El esfuerzo se establece en '-1', puesto que en el robot real no está implementado este campo.

Llegados a este punto, ya estamos acabando la etapa de *sensing* (la más larga del *script*). Solo nos queda empaquetar los datos que hemos obtenido de cada uno de los sensores, para que podamos acceder a ellos de forma remota.

Para ello, primero creamos tres variables locales, y llamamos a la función *simPackFloats* para empaquetar unas cuantas variables de tipo *float* en un *string*. Una vez hecho eso, procedemos a cambiar el estado de cada una de las señales con los datos actualizados.

Obsérvese que la señal correspondiente al tiempo de simulación es de tipo *float* (solo consta de un campo), por lo que no es necesario empaquetar esa información. En el capítulo de esta memoria dedicado al diseño de la *toolbox* hablaremos con mayor profundidad acerca de las señales y su composición.

```

— PACK AND UPDATE SIGNALS —
local sensorsData = simPackFloats({simTime,
    bumperRightState, bumperCenterState,
    bumperLeftState, wheel_drop_right_state,

```

```

wheel_drop_left_state, rightCliffState,
frontCliffState, leftCliffState, distanceRC,
distanceFC, distanceLC, RWTicks,
LWTicks, button_0, button_1, button_2}, 0, 17);

local poseQuaternionData = simPackFloats({pose_x
, pose_y, 0, quaternion[1], quaternion[2],
quaternion[3], quaternion[4], linearSpeed[1],
linearSpeed[2], linearSpeed[3], angularSpeed[1],
angularSpeed[2], angularSpeed[3],
linearAcceleration[1], linearAcceleration[2],
linearAcceleration[3]}, 0, 16);

local jointStatesData = simPackFloats({wlj_pos,
wlj_vel, wlj_eff, wrj_pos, wrj_vel, wrj_eff,
a1j_pos, a1j_vel, a1j_eff, a2j_pos, a2j_vel,
a2j_eff, a3j_pos, a3j_vel, a3j_eff, a4j_pos,
a4j_vel, a4j_eff, a5j_pos, a5j_vel, a5j_eff,
gpj_pos, gpj_vel, gpj_eff}, 0, 24);

simSetStringSignal(modelBaseName..
'_kobuki_sensor_state', sensorsData)

simSetStringSignal(modelBaseName..
'_kobuki_quaternion_vel_accel', poseQuaternionData)

simSetStringSignal(modelBaseName..
'_joint_states', jointStatesData)

simSetFloatSignal(modelBaseName..
'_simulation_time', simTime)
end

```

Entramos ahora en la parte de *actuation* del *script*. Recordemos que esta parte es la encargada de generar los comandos para los actuadores en base a las decisiones tomadas con los valores de los sensores.

El primer fragmento de código que nos encontramos tiene que ver con los motores de la base Kobuki. A estos motores (articulaciones rotacionales) se les manda la velocidad angular deseada. El motor aplicará el par máximo establecido sobre su eje mientras no se alcance esa velocidad. Una vez alcanzada, se mantiene, y si baja, se vuelve a aplicar el par máximo.

Kobuki dispone de un mecanismo de seguridad que hace que si pasados 0,6 segundos no se ha recibido ningún comando de velocidad nuevo, la base se detiene automáticamente. Para emular esto, la señal que se utiliza para controlar los motores tiene tres parámetros (velocidad motor derecho, velocidad motor izquierdo y número de comando).

Utilizando el parámetro “número de comando”, cuyo valor se incrementa cada vez que se envía un nuevo comando desde MATLAB, podemos saber si el usuario ha mandado un nuevo comando. Así pues, solo tenemos que comprobar que la diferencia entre el tiempo actual de simulación y el último comando sea menor a 0,6. En ese caso, paramos los motores:

```

if (sim_call_type == sim_childscriptcall_actuation) then

    — KOBUKI WHEELS —
    if(simGetIntegerSignal(modelBaseName..'_kobuki_motors_enabled') == 1)
    then
        wheelSpeedData = simGetStringSignal(modelBaseName..
            '_kobuki_wheels_speed')
        wheelSpeed      = simUnpackFloats(wheelSpeedData, 0, 3, 0)

        simSetJointTargetVelocity(t_rightWheel, wheelSpeed[1])
        simSetJointTargetVelocity(t_leftWheel,  wheelSpeed[2])

        if(wheelSpeed[3] ~= lastSpeedCommandCtr) then
            lastSpeedCommand      = simGetSimulationTime()
            lastSpeedCommandCtr = wheelSpeed[3]
        end

        if(wheelSpeed[1] ~= 0 or wheelSpeed[2] ~= 0) then
            if(simGetSimulationTime() - lastSpeedCommand > 0.6) then
                simSetJointTargetVelocity(t_rightWheel, 0)
                simSetJointTargetVelocity(t_leftWheel,  0)
            end
        end
    else
        simSetJointTargetVelocity(t_rightWheel, 0)
        simSetJointTargetVelocity(t_leftWheel,  0)
    end

```

Kobuki nos permite reiniciar sus datos odométricos. Si hacemos eso, la pose del robot pasará a su estado inicial (es decir $X=0$, $Y=0$, $\theta=0$) y los *encoders* tomarán el valor '0'. Para ello, guardamos de nuevo la matriz de rotación asociada al robot, con respecto al sistema universal y reinicializamos las variables pertinentes:

```

    — KOBUKI ODOMETRY RESET —
    if(simGetIntegerSignal(modelBaseName..'_kobuki_odometry_reset') == 1)
    then
        originPosition      = simGetObjectPosition(ref, -1)
        originOrientation    = simGetObjectMatrix(ref, -1)
        previousRWPosition  = simGetJointPosition(t_rightWheel)
        previousLWPosition  = simGetJointPosition(t_leftWheel)
        totalRWPosition     = 0
        totalLWPosition     = 0
    end

```

```

pose_x           = 0
pose_y           = 0
pose_tita       = 0
simSetIntegerSignal(modelBaseName..' _kobuki_odometry_reset', 0)
end

```

Lo último que nos encontramos en el *script* principal de nuestro modelo es el código asociado a los LEDs. Simplemente, se cambia el color del mismo en función del valor que tenga la señal:

- 0 → apagado
- 1 → verde
- 2 → naranja
- 3 → rojo

```

local led_1_new_state = simGetIntegerSignal(modelBaseName..'
 _kobuki_led_1')
if(led_1_new_state ~= led_1_old_state) then
  if      (led_1_new_state == 0) then simSetShapeColor(t_led_1, nil, 0,
    blackLed)
  elseif (led_1_new_state == 1) then simSetShapeColor(t_led_1, nil, 0,
    greenLed)
  elseif (led_1_new_state == 2) then simSetShapeColor(t_led_1, nil, 0,
    orangeLed)
  elseif (led_1_new_state == 3) then simSetShapeColor(t_led_1, nil, 0,
    redLed)
  end
  led_1_old_state = led_1_new_state
end
end
end

```

3.6.2. WidowX

En este apartado presentamos el *script* asociado a WidowX. En él se encuentra el código que controla las articulaciones del brazo.

Como el *script* anterior, este también comienza con una fase de inicialización. En ella, lo primero que hacemos es almacenar el nombre del modelo base en una variable. Esto nos va a ser de ayuda para evitar tener que escribir el nombre completo de las señales, además de que de esta manera, si cambia el nombre del modelo, no tenemos que ir cambiando las señales una por una.

Dentro de esta inicialización, también nos encontramos con la definición de las fricciones internas de cada *servo*. Estas fricciones solo se harán efectivas cuando la articulación en cuestión esté relajada o deshabilitada.

Además, almacenamos los manejadores en variables para tal fin y establecemos la relación de las articulaciones en “false”:

```
if (sim_call_type==sim_childscriptcall_initialization) then

    modelBase      = simGetObjectHandle('Turtlebot2_WidowX')
    modelBaseName = simGetObjectHandle(modelBase)

    mx_64_friction = 0.50
    mx_28_friction = 0.26
    ax_12_friction = 0.13

    — Handles
    w_arm          = simGetObjectHandle('arm_joint')
    w_shoulder    = simGetObjectHandle('shoulder_joint')
    w_biceps      = simGetObjectHandle('biceps_joint')
    w_forearm     = simGetObjectHandle('forearm_joint')
    w_wrist       = simGetObjectHandle('wrist_joint')
    w_gripper     = simGetObjectHandle('gripper_1_joint')

    w_arm_relaxed      = false
    w_shoulder_relaxed = false
    w_biceps_relaxed   = false
    w_forearm_relaxed  = false
    w_wrist_relaxed    = false
```

Lo siguiente que hacemos en la inicialización es establecer las posiciones iniciales de las articulaciones del brazo. Para establecer estas posiciones empleamos las señales que están definidas para tal fin. Una vez que paremos la simulación, las articulaciones volverán a estas posiciones, ya que en el cuadro de propiedades de cada articulación está especificado así.

Estas serán las posiciones en las que el brazo permanecerá mientras que no se reciba el primer comando. Concretamente, son las que se muestran a continuación (el sentido de los ángulos está establecido del mismo modo que en el robot real):

- Pinza → 0 (abierta completamente)
- Base → 45 grados
- Hombro → 0 grados
- Bíceps → -45 grados
- Antebrazo → -45 grados
- Muñeca → 0

Por otro lado, inicializamos las señales que especifican el modo en el que está cada una de las articulaciones, con el valor '1' (es decir, habilitadas para recibir comandos).

La otra señal con la que nos encontramos (acabada en "reset_dynamics") sirve para reiniciar la dinámica de una articulación, tal y como veremos más adelante. Este hecho resulta imprescindible cuando se cambia la velocidad máxima de la misma.

```

— WidowX Joints position
simSetFloatSignal(modelBaseName.. '_widowx_gripper_joint', 0)
simSetFloatSignal(modelBaseName.. '_widowx_arm_joint',
    ((math.pi / 180) * 45))
simSetFloatSignal(modelBaseName.. '_widowx_shoulder_joint', 0)
simSetFloatSignal(modelBaseName.. '_widowx_biceps_joint',
    ((math.pi / 180) * - 45))
simSetFloatSignal(modelBaseName.. '_widowx_forearm_joint',
    ((math.pi / 180) * - 45))
simSetFloatSignal(modelBaseName.. '_widowx_wrist_joint', 0)

— WidowX Joints Configuration
— 0 -> DISABLED
— 1 -> ENABLED
— 2 -> RELAXED
simSetIntegerSignal(modelBaseName.. '_widowx_arm_joint_conf', 1)
simSetIntegerSignal(modelBaseName.. '_widowx_shoulder_joint_conf', 1)
simSetIntegerSignal(modelBaseName.. '_widowx_biceps_joint_conf', 1)
simSetIntegerSignal(modelBaseName.. '_widowx_forearm_joint_conf', 1)
simSetIntegerSignal(modelBaseName.. '_widowx_wrist_joint_conf', 1)
simSetIntegerSignal(modelBaseName.. '_widowx_gripper_joint_conf', 1)

— WidowX Reset Dynamics signal
simSetIntegerSignal(modelBaseName.. '_widowx_reset_dynamics', 0)
end

```

El siguiente fragmento de código contiene la fase de limpieza o **cleanup** de WidowX.

En esta fase tenemos que realizar varias cosas. Para cada una de las articulaciones, hay que reestablecer la fuerza máxima original (estos valores están basados en los *datahsets* de cada una de ellas), habilitar el controlador P que las comanda y establecer la velocidad angular máxima de las mismas en 100 grados por segundo.

```

if (sim_call_type==sim_childscriptcall_cleanup) then
    simSetJointForce(w_arm, 2.5)
    simSetJointForce(w_shoulder, 6)
    simSetJointForce(w_biceps, 2.5)
    simSetJointForce(w_forearm, 2.5)
    simSetJointForce(w_wrist, 1.5)

    simSetObjectInt32Parameter(w_arm,
        sim_jointintparam_ctrl_enabled, 1)
    simSetObjectInt32Parameter(w_shoulder,

```



```

    sim_jointintparam_ctrl_enabled , 1)
simSetObjectInt32Parameter(w_biceps ,
    sim_jointintparam_ctrl_enabled , 1)
simSetObjectInt32Parameter(w_forearm ,
    sim_jointintparam_ctrl_enabled , 1)
simSetObjectInt32Parameter(w_wrist ,
    sim_jointintparam_ctrl_enabled , 1)

simSetObjectFloatParameter(w_arm, sim_jointfloatparam_upper_limit ,
    1.74533)
simSetObjectFloatParameter(w_shoulder , sim_jointfloatparam_upper_limit ,
    1.74533)
simSetObjectFloatParameter(w_biceps , sim_jointfloatparam_upper_limit ,
    1.74533)
simSetObjectFloatParameter(w_forearm , sim_jointfloatparam_upper_limit ,
    1.74533)
simSetObjectFloatParameter(w_wrist , sim_jointfloatparam_upper_limit ,
    1.74533)

```

end

Este *script* no tiene fase de *sensing*, puesto que WidowX no consta de sensores estereoceptivos. Así pues, lo que nos queda por ver es la parte de *actuation*, que será la encargada de establecer las posiciones objetivo de las articulaciones, entre otras cosas.

En ella, lo primero que tenemos que hacer es comprobar la señal correspondiente al reinicio dinámico de alguna articulación. Para ello, si su valor es distinto de '0', seleccionamos el manejador de la articulación que corresponda.

Una vez reiniciada la articulación, establecemos el valor de la señal en '0', para que en la próxima ejecución del *script* no vuelva a producirse el reinicio.

El hecho de tener que reiniciar una articulación para poder cambiar su velocidad máxima es una limitación que nos impone V-REP. No es posible evitarla, y además, su ejecución puede dar lugar a comportamientos extraños (por ejemplo, que la articulación dé un salto).

Así pues, no es recomendable llamar a esta función muy asiduamente, y las veces que lo hagamos mejor que sea cuando el brazo no tiene objetos agarrados.

```

if (sim_call_type == sim_childscriptcall_actuation) then
  — WIDOWX JOINTS —
  — Reset Dynamics
  local resetHandle
  local resetDynamics = simGetIntegerSignal(modelBaseName..
    '_widowx_reset_dynamics');
  if (resetDynamics ~= 0) then
    if (resetDynamics == 1) then resetHandle = w_arm
    elseif (resetDynamics == 2) then resetHandle = w_shoulder
    elseif (resetDynamics == 4) then resetHandle = w_biceps

```

```

elseif (resetDynamics == 8) then resetHandle = w_forearm
elseif (resetDynamics == 16) then resetHandle = w_wrist
elseif (resetDynamics == 32) then resetHandle = w_gripper
end

simResetDynamicObject(resetHandle)

simSetIntegerSignal(modelBaseName..' _widowx_reset_dynamics', 0)
end

```

Lo siguiente que vamos a hacer es almacenar en variables locales los valores de las señales que establecen el modo de funcionamiento para cada una de las articulaciones:

```

local w_arm_conf      = simGetIntegerSignal(modelBaseName..'
  '_widowx_arm_joint_conf')
local w_shoulder_conf = simGetIntegerSignal(modelBaseName..'
  '_widowx_shoulder_joint_conf')
local w_biceps_conf   = simGetIntegerSignal(modelBaseName..'
  '_widowx_biceps_joint_conf')
local w_forearm_conf  = simGetIntegerSignal(modelBaseName..'
  '_widowx_forearm_joint_conf')
local w_wrist_conf    = simGetIntegerSignal(modelBaseName..'
  '_widowx_wrist_joint_conf')
local w_gripper_conf  = simGetIntegerSignal(modelBaseName..'
  '_widowx_gripper_joint_conf')

```

Lo último que nos queda por hacer es enviar a la articulación el comando de posición correspondiente, en función del modo seleccionado. Esta funcionalidad emula la existente en el robot real, ya que cada articulación puede encontrarse en tres estados:

- Deshabilitada (0) → no admite comandos (solo opone al movimiento la fricción interna)
- Habilitada (1) → admite comandos y mantiene la última posición
- Relajada (2) → admite comandos pero solo opone al movimiento la fricción interna

El código que modela ese comportamiento se muestra a continuación. Solo vamos a mostrar el de una articulación, puesto que el código es muy similar para cada una de ellas. En el caso de estado deshabilitado, es muy sencillo. Simplemente establecemos la fuerza máxima en la fricción interna⁴ y deshabilitamos el controlador. De este modo, por muchos comandos que enviemos a la articulación, esta no va a responder.

El segundo caso es el de articulación habilitada. En ese caso, establecemos la fuerza máxima de la articulación en el valor que nos proporciona el *datasheet*. Además, habilitamos

⁴La fricción interna es una estimación proporcional al tamaño del *servo*

el controlador, puesto que podría estar deshabilitado. Una vez hecho eso, establecemos la posición objetivo de la articulación en la que nos dicta la señal para tal cometido.

El último caso se corresponde con articulación relajada. Lo primero que comprobamos es si la articulación estaba ya relajada. En ese caso, si el valor objetivo ha cambiado (es decir, se ha enviado un comando), entonces la articulación deja de estar relajada y pasa a estado “habilitada”. En caso de que no estuviera, establecemos su fuerza máxima en la fricción interna y deshabilitamos el controlador. Además, establecemos la posición objetivo en ‘999’, que es el valor que utilizamos para interpretar que no se han enviado comandos:

```
if (w_arm_conf == 0) then
    simSetJointForce(w_arm, mx_28_friction)
    simSetObjectInt32Parameter(w_arm, sim_jointintparam_ctrl_enabled, 0)

elseif (w_arm_conf == 1) then
    simSetJointForce(w_arm, 2.5)
    simSetObjectInt32Parameter(w_arm, sim_jointintparam_ctrl_enabled, 1)
    simSetJointTargetPosition(w_arm, simGetFloatSignal(modelBaseName..
        '_widowx_arm_joint'))

elseif (w_arm_conf == 2) then
    if(w_arm_relaxed) then
        if ((simGetFloatSignal(modelBaseName.. '_widowx_arm_joint') ~=
            999)) then
            simSetIntegerSignal(modelBaseName.. '_widowx_arm_joint_conf',
                1)
            w_arm_relaxed = false
        end
    else
        w_arm_relaxed = true
        simSetJointForce(w_arm, mx_28_friction)
        simSetObjectInt32Parameter(w_arm,
            sim_jointintparam_ctrl_enabled, 0)
        simSetFloatSignal(modelBaseName.. '_widowx_arm_joint', 999)
    end
end
end
[...]
```

3.7. Otros detalles del modelo

Ya casi hemos terminado el modelado de nuestro conjunto robótico. Solo nos queda ajustar unos pocos detalles.

Con respecto al sensor Hokuyo, hemos de decir que consta de un *script* asociado, de la misma forma que Turtlebot y WidowX. Al ser un *script* proporcionado con el sensor,

no vamos a describirlo detalladamente, puesto que su elaboración no ha sido fruto de este trabajo.

En dicho *script* se han hecho unas pequeñas modificaciones. En concreto, se ha añadido el empaquetado de los datos recogidos por el sensor y el establecimiento de los mismos a una señal, para poder obtenerlos de forma remota. Además, el primer *float* de la señal se corresponde con el tiempo de simulación:

```
[...]  
table.insert(measuredData, simGetSimulationTime())  
[...]  
data = simPackFloats(measuredData)  
simSetStringSignal(modelBaseName..'_hokuyo_data', data)  
[...]
```

Algo parecido ocurre con el *script* asociado a la pinza de WidowX. Se trata de un *script* que no hemos implementado, pero al que le hemos hecho una pequeña modificación, que consiste en tomar el valor de apertura de la misma de una señal. Además, se ha normalizado ese valor para que se encuentre en el rango $[0, 2.3]$, donde '0' significa completamente abierta y '2.3' completamente cerrada:

```
[...]  
value = simGetFloatSignal(modelBaseName..'_widowx_gripper_joint')  
simSetJointTargetPosition(openCloseJoint, 0.01472 * value)  
[...]
```

Hay muchas cuestiones que influyen a la hora del comportamiento dinámico del modelo. Una de ellas es el tipo de material definido para cada una de las piezas. Por ejemplo, es importante que tanto las ruedas motrices de Kobuki como la parte que agarra de la pinza de WidowX tengan establecido un material con alto coeficiente de rozamiento.

Para realizar este ajuste, tenemos que abrir la ventana de las propiedades dinámicas de la pieza en cuestión (figura 3.31). Simplemente elegimos el material deseado en la lista disponible, o bien podemos definir nosotros uno propio. Para el caso de la pinza, empleamos el denominado "bulletMaterial_sticky_special".

También podemos cambiar los valores de masa y las inercias de las *shapes responsables* en ese mismo cuadro de diálogo. Eso hará que el comportamiento dinámico del conjunto robótico sea lo más parecido al robot real posible.

Con esto ya tenemos terminado el modelado de nuestro conjunto robótico en V-REP. Lo siguiente será diseñar la *toolbox* para poder programarlo.

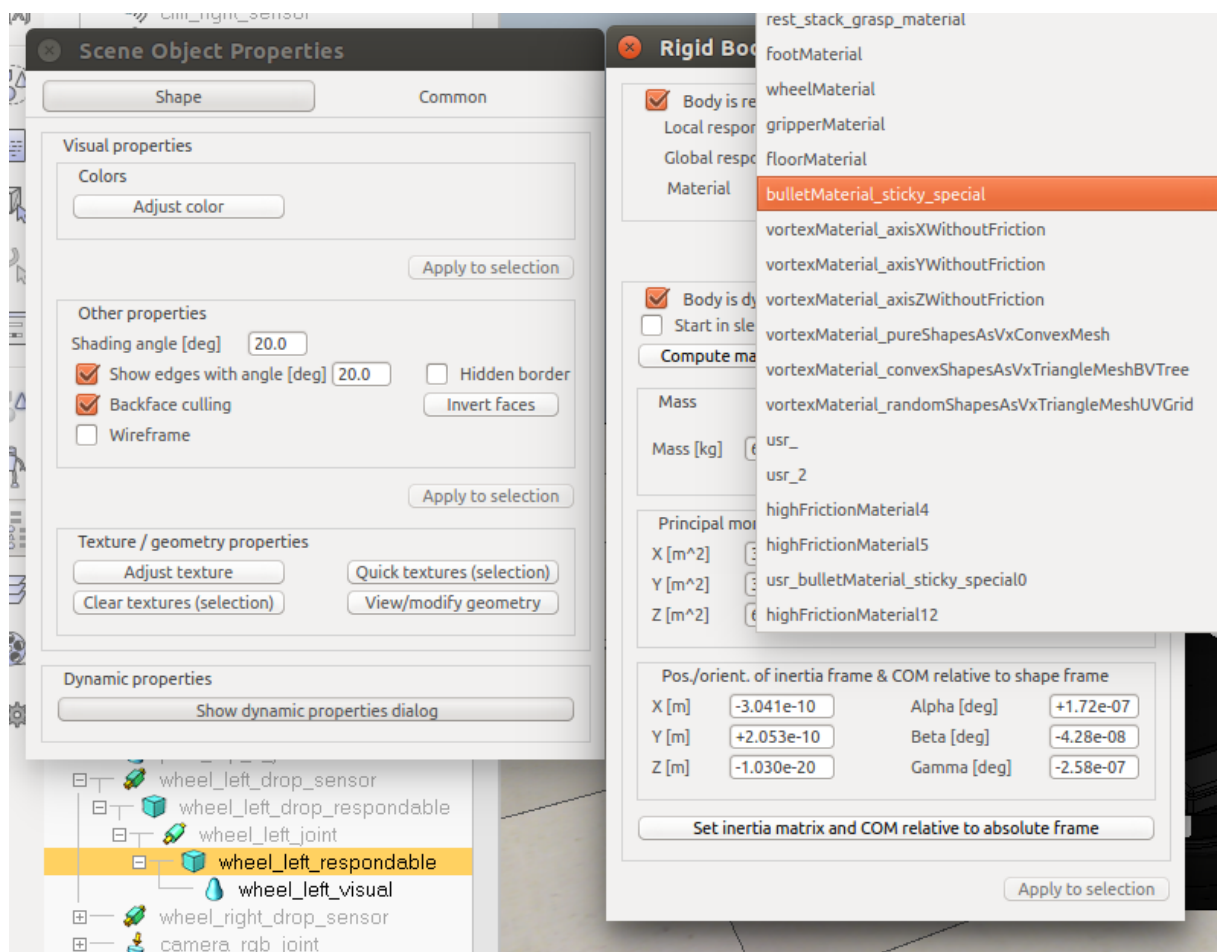


Figura 3.31: Cambio del material de una pieza.

Capítulo 4

Diseño de la *toolbox*

En este capítulo de la memoria vamos a abordar la elaboración de la *toolbox*. Esta es la parte que mayor tiempo nos ha ocupado, ya que no partimos de ningún trabajo previo. Tal y como se comentó anteriormente, la *toolbox* va a tener una apariencia parecida a la *Robotics System Toolbox* que tenemos incluida en nuestra licencia de MATLAB. De esta manera, un usuario que tenga cierta soltura con ROS va a poder implementar programas empleando nuestra *toolbox* sin demasiadas complicaciones.

4.1. ¿Por dónde empezar?

Antes de comenzar la implementación de la *toolbox*, debemos echar un vistazo a las posibilidades que tenemos y de dónde partimos.

En primer lugar, V-REP consta de una API remota que permite controlar la simulación desde una aplicación externa. Existen varias posibilidades, entre las que se encuentran C++, Python, Java, MATLAB/Octave, Urbi y Lua. En nuestro caso, esa aplicación externa va a ser MATLAB.

Esta API remota nos proporciona los ficheros necesarios para poder emplearla directamente en MATLAB, incluyéndose los siguientes:

- `remoteApiProto.m` → Protocolo de comunicación con V-REP
- `remApi.m` → Contiene las funciones de la API a las que podemos llamar
- `remoteApi.[dll, dylib, so]` → Librería de comunicación, depende del sistema operativo empleado

Estos tres ficheros se encuentran en la carpeta de instalación de V-REP. Por tanto, para poder usar la API remota solamente tenemos que copiar esos tres ficheros a una carpeta conocida, y colocar en la misma el *path* de MATLAB. De ese modo, ya podríamos llamar a las funciones tal y como si nos encontrásemos en el propio V-REP (eso sí, respetando los nombres de las mismas, ya que suelen empezar por “*simx*”). Eso implicaría una

programación “a bajo nivel”, puesto que obligaríamos al usuario a acceder directamente a cada una de las partes que componen el robot (por ejemplo, a los motores de las ruedas), y eso no es lo que pretendemos.

Nuestra idea es la de desarrollar un entorno de programación que permita trabajar al usuario tanto con el robot simulado como con el robot real. Si usamos directamente las funciones de la API remota de V-REP, el programa que hagamos solo va a servir para el simulador.

Una de las posibilidades para este entorno de programación es la creación de una *toolbox*. Una *toolbox* es un conjunto de herramientas que nos permiten realizar tareas complejas utilizando funciones predefinidas.

Por tanto, una opción sería “inventarnos” un conjunto de funciones para que el usuario pudiera programar el robot usando las mismas. Es decir, tendríamos, por ejemplo, una función que podría llamarse *moverRobotHaciaDelante(metros)* que moviese el robot hacia delante una determinada distancia. Y así cuantas queramos.

No obstante, nos encontramos ante el mismo problema. El programa desarrollado no serviría más que para el robot simulado en V-REP. Lo que ocurre es que el robot real funciona con ROS. ¿Cómo hacemos para comunicar MATLAB con ROS?

Pues gracias a que tenemos disponible la *Robotics System Toolbox* de MATLAB (figura 4.1), tenemos la capacidad de comunicarnos directamente con el robot real, sin intermediarios.

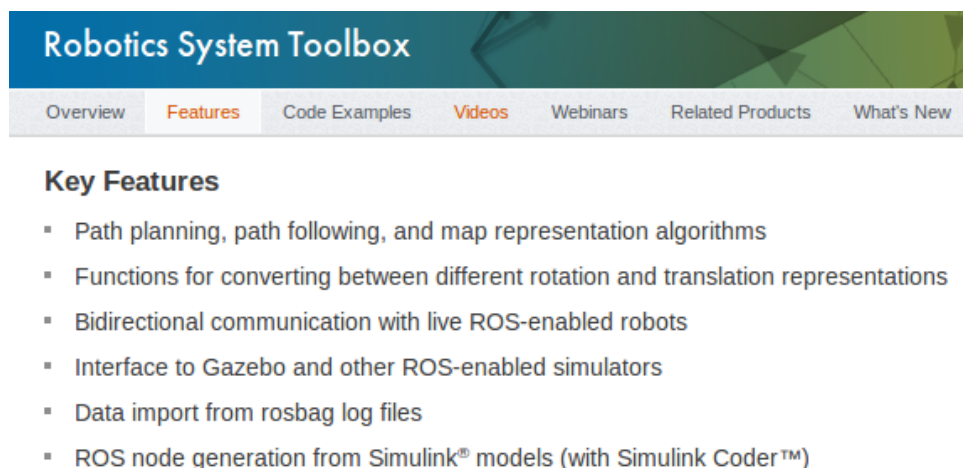


Figura 4.1: Robotics System Toolbox de MATLAB.

Por tanto la opción que llevamos a cabo es la siguiente: implementamos una *toolbox* con una apariencia muy similar (mismos nombres de funciones) que la *Robotics System Toolbox*. Cuando estemos usando el simulador, estas funciones llamarán a las de la API remota de V-REP, y cuando estemos usando el robot real, llamarán a las funciones de la *Robotics System Toolbox*. Además, vamos a establecer un modo mixto que permita trabajar a la vez con el robot simulado y con el real.

Esto implica que vamos a tener que “emular” los *topics* del robot real, porque de otra manera no se podría usar el mismo programa para el robot simulado que para el real. Cada *topic* devuelve (o permite que se publique) un mensaje con ciertos datos de sensores o comandos de actuación.

Una posibilidad sería utilizar las funciones de la API remota para ir leyendo los valores de los sensores según nos vayan haciendo falta. El problema es que hay *topics*¹ que devuelven valores de varios sensores, como por ejemplo el “/mobile_base/sensors/core”.

No debemos olvidar que esta API remota funciona a través de *sockets*. Por tanto, cada llamada a una función de la API remota va a implicar cierto tiempo, ya que tiene que ser enviada al servidor (V-REP), ejecutada, y devolver el resultado al cliente (MATLAB).

Es por ello por lo que en la fase de modelado hemos definido un conjunto de señales. El hecho de leer el contenido de una señal equivale a la ejecución de una función. Esto es mucho más eficiente que ir leyendo sensor por sensor cada vez que haya que rellenar el mensaje asociado al *topic* “/mobile_base/sensors/core”, por ejemplo.

Además, esto va a unificar mucho más las funciones de la *toolbox*, ya que la gran mayoría de ellas van a usar señales para los distintos *topics*, y con vistas a la ampliación de la misma, nos vamos a encontrar con un código mucho más entendible y mantenible.

Con respecto a las funciones que vamos a implementar, imitando a la *Robotics System Toolbox*, van a ser las de la tabla 4.1. Estas funciones estarán disponibles para el usuario para su uso con los *topics* del robot (tanto simulado como real).

Nombre	Descripción
rosinit	Inicia la conexión con ROS. <i>master</i>
rostopic	Proporciona información sobre los <i>topics</i> . Implementaremos las opciones ' <i>list</i> ', que devuelve un listado con los <i>topics</i> disponibles, y ' <i>type</i> ', que devuelve el tipo de mensaje para un <i>topic</i> dado.
rosservice	Proporciona información sobre los servicios. Implementaremos la opción ' <i>list</i> ', que devuelve los servicios disponibles.
roshutdown	Cierra la conexión con ROS.
rospublisher	Crea un <i>publisher</i> para un <i>topic</i> dado.
rossubscriber	Crea un <i>subscriber</i> para un <i>topic</i> dado.
rossvcclient	Crea un cliente para un servicio.
rosmessage	Crea un mensaje para un <i>topic</i> o servicio dado.
send	Envía un mensaje a través de un <i>topic</i> .
receive	Recibe un mensaje a través de un <i>topic</i> .
call	Llama a un servicio.

Cuadro 4.1: Listado de funciones implementadas.

Además, existe un conjunto de funciones que solo sirven para cuando nos encontramos en el modo de simulación, y que se encuentran detalladas en el manual de usuario.

¹En el manual del usuario existe una tabla con el listado de *topics* implementados del robot real.

4.2. Implementando las funciones

Llegados a este punto, la cuestión es cómo hacer para llamar a las funciones de la *Robotics System Toolbox* desde nuestras funciones (que tienen el mismo nombre).

Podemos hacer eso empleando una clase en MATLAB. De este modo, llamaremos a nuestras funciones escribiendo, por ejemplo, *objeto.rosinit(...)*, y en el código de la misma pondremos la llamada a la función *rosinit* de la *Robotics System Toolbox*.

Para generar una clase en V-REP, hemos de crear una carpeta con el nombre de la clase y una '@' delante. En nuestro caso, la clase va a llamarse VREP, así que creamos una carpeta con "@VREP" como nombre.

Dentro de esa carpeta, colocaremos los ficheros correspondientes a las funciones que implementemos, junto al fichero de definición de clase, que contiene la lista de funciones y establece cuáles son públicas y cuáles privadas, entre otras cosas. En la versión terminada de la *toolbox*, esta carpeta consta de 54 ficheros, así que no parece razonable explicar el código de cada uno de ellos, teniendo en cuenta que muchas son parecidas. Así pues, explicaremos el código de aquellas más importantes y representativas.

4.2.1. *rosinit*

Vamos a comenzar con la primera función a la que deberemos llamar en nuestro programa, y que es la encargada de iniciar la comunicación entre MATLAB y V-REP. Se trata de *rosinit*, y necesita las direcciones IP y los puertos que vamos a usar. En caso de que trabajemos de forma local, pondremos como IP "127.0.0.1", por ejemplo. Su cabecera se muestra en el siguiente recuadro:

```
function [returnCode] = rosinit(robot , v_ip_addr , v_port , r_ip_addr , r_port)
```

Dentro de la misma, lo primero que hacemos es mostrar una cabecera, notificando al usuario de que se está iniciando la comunicación, junto a la versión de la Robotics System Toolbox que tiene, además de otras cuestiones relacionadas:

```
disp( '#####' );  
disp( 'Starting .... ' );  
ver robotics;  
disp( '_____ ' );
```

A continuación, comprobamos si el modo *real* o el modo *mixed* están seleccionados, en cuyo caso será necesario comunicarse con el robot real, usando la Robotics System Toolbox. Además, almacenamos la IP y el puerto para futuras referencias:

```
if(robot.operating_mode == 1 || robot.operating_mode == 2)  
    rosinit(r_ip_addr , r_port);    showspaces=false ]  
    robot.r_ip_addr = r_ip_addr;  
    robot.r_port    = r_port;  
end
```

Lo siguiente que hacemos es comprobar si el modo *vrep* (o *mixed*) está habilitado. En caso, tendremos que iniciar la comunicación con V-REP. Hemos de crear el objeto necesario para tal cometido, usando la API remota de V-REP. Una vez hecho eso, finalizamos las conexiones que hubieran anteriormente y tratamos de establecer una nueva:

```

if(robot.operating_mode == 0 || robot.operating_mode == 2)
    robot.v_ip_addr = v_ip_addr;
    robot.v_port    = v_port;
    robot.vrep      = remApi(strcat('remoteApi_', computer));
    robot.vrep.simxFinish(-1);
    robot.v_clientID = robot.vrep.simxStart(v_ip_addr, v_port, true, ...
    true, 5000, 5);

```

Hay que comprobar que la conexión se ha establecido correctamente, ya que podría producirse un fallo por dirección IP incorrecta, por ejemplo. Si la conexión ha sido exitosa, mostramos un mensaje en la barra de estado de V-REP, y llamamos a la función que adquiere los manejadores necesarios (especialmente, los del sensor Kinect). También inicializamos dos variables de control del sensor Hokuyo:

```

if (robot.v_clientID > -1) % Connection OK
    robot.vrep.simxAddStatusBarMessage(robot.v_clientID, ...
    'Connected from MATLAB', robot.vrep.simx_opmode_oneshot);

    robot.acquireVREPHandlers();

    robot.setVREPTopics();
    robot.setVREPServices();

    robot.header_seq_number = 0;
    robot.hokuyo_last_measure = 0;

```

Seguidamente, inicializamos los *streamings* de Kinect y de las señales que hemos definido (solo se muestra una inicialización para evitar ser reiterativos) y mostramos por consola el *ping* entre MATLAB y V-REP:

```

    robot.vrep.simxGetVisionSensorImage2(robot.v_clientID, ...
    robot.v_handles(2, 1), 0, robot.vrep.simx_opmode_streaming);
    [...]

    fprintf('Ping time: %d msecs.\n', robot.getPingTime());
    disp('-----');

    pause(0.5);

    disp(' [OK] ');
    disp('#####');

    returnCode = 0;

```

Si ha ocurrido algún error, lanzamos un *error* de MATLAB y notificamos al usuario. Esto provocaría que el programa implementado por el usuario acabara aquí. Con esto damos por terminada la implementación de la función *rosinit*:

```
        else
            error('Cannot connect to V-REP, ...
                check IP and port and restart simulation');
        end
    end
end
```

4.2.2. *rospublisher* y *rossubscriber*

Es altamente probable que lo siguiente que haga el usuario en su programa sea obtener los *publishers* y los *subscribers* de los *topics* que vaya a emplear en su programa. Para ello dispone de dos funciones: *rospublisher* y *rossubscriber*. Estas funciones tienen una implementación muy similar (de hecho, sustituyendo las ocurrencias de “publisher” por “subscriber” se obtiene la contraria), así que solo vamos a ver el código de una de ellas. La cabecera es la siguiente:

```
function [publisher , returnCode] = rospublisher(robot , topicname)
```

En el código, comprobamos si el modo es *vrep*, en cuyo caso habrá que crear una estructura con el nombre del *topic*, o si por el contrario está habilitado ROS, pudiendo usar la función *rospublisher* de la Robotics System Toolbox:

```
    if robot.operating_mode == 0
        publisher = struct('TopicName', topicname);
        returnCode = 0;

    elseif (robot.operating_mode == 1 || robot.operating_mode == 2)
        publisher = rospublisher(topicname);
        returnCode = 0;
    end
end
```

4.2.3. *rosmessage*

Otra de las cuestiones que va a necesitar hacer el usuario va a ser el hecho de crear mensajes para poder enviarlos por los *topics* con el tipo correcto. Estos mensajes suelen tener unos cuantos campos, y sería bastante engorroso crearlos desde cero. Para ello, disponemos de la función *rosmessage*, cuya cabecera es la siguiente:

```
function [msg, returnCode] = rosmessage(robot , publisher)
```

El código es bastante simple. En caso de que nos encontremos en modo *vrep*, llamará a la función auxiliar *getVREPMessage* (no visible para el usuario desde su programa), la cual devolverá el mensaje correcto para un *topic* dado por su nombre. La implementación de dicha función la veremos a continuación.

Si nos encontramos en modo *real* o *mixed*, entonces hará uso directo de la función *rosmmessage* de la *Robotics System Toolbox*. Cuando estamos en modo V-REP, hemos de comprobar si el mensaje es para un *topic* o para un servicio:

```

if robot.operating_mode == 0
    if(isfield(entity, 'TopicName'))
        [msg, returnCode] = robot.getVREPMessage(entity.TopicName);
    elseif(isfield(entity, 'ServerName'))
        [msg, returnCode] = robot.getVREPMessage(entity.ServerName);
    else
        returnCode = -1;
    end

    elseif (robot.operating_mode == 1 || robot.operating_mode == 2)
        msg = rosmmessage(publisher);
        returnCode = 0;
    end
end

```

La función *getVREPMessage* es la encargada de devolver el mensaje apropiado para el nombre de *publisher* que se le pase como parámetro. Su cabecera es la siguiente:

```

function [message, returnCode] = getVREPMessage(robot, publish)

```

Lo primero que se hace dentro de la misma es inicializar la variable para control de errores y “trocear” el nombre del *publisher*, lo que nos va a ayudar a buscar el *topic* en cuestión:

```

returnCode = 0;
publisher = strsplit(publish, '/');

```

Para encontrar el *topic*, simplemente usamos una estructura de *switch* e *if-elseif* tal y como se muestra en el siguiente ejemplo. Una vez encontrado el *topic*, llamamos a la función *rosmmessage* de la *Robotics System Toolbox* para generar el mensaje con el tipo adecuado. Evidentemente, hemos de hacer esto para cada uno de los *topics* (y servicios) de nuestro robot:

```

switch (char(publisher(2)))
    case 'mobile_base'
        if(strcmp(publisher(3), 'commands'))
            if(strcmp(publisher(4), 'velocity'))
                message = rosmmessage('geometry_msgs/Twist');
            [...]

```

```

        else
            returnCode = -1;
        end
    end
    [...]
    otherwise
        returnCode = -1;
    end
end
end

```

4.2.4. *send*

Una vez que ya tenemos los *publishers*, los *subscribers* y los mensajes creados, querramos utilizarlos para enviar y recibir mensajes por los *topics*. Para enviar un mensaje por un *topic*, disponemos de la función *send*. A esta función se le pasa como argumento el *publisher* y el *mensaje* que se pretende enviar. Para el caso de modo *mixed*, es posible enviar mensajes distintos al robot real y al robot simulado.

La cabecera de la función se muestra a continuación:

```

function [returnCode] = send(robot, in_1, in_2, in_3, in_4)
% {V-REP mode} -> [returnCode] = send(robot, publisher, message)
% {ROS mode} -> [returnCode] = send(robot, publisher, message)
% {MIXED mode} -> [returnCode] = send(robot, publisher_r, ...
%                                     message_r, publisher_v, message_v)

```

El código de la misma es bastante sencillo. En caso de modo *vrep*, llamamos a la función *sendToVREP*, la cual describiremos a continuación. Si el modo está establecido en *real*, llamamos a la función *send* de la Robotics System Toolbox. En caso de modo *mixed*, llamamos a ambas:

```

switch (robot.operating_mode)
    case 0 % V-REP
        returnCode = sendToVREP(robot, in_1.TopicName, in_2);

    case 1 % ROS
        send(in_1, in_2);
        returnCode = 0;

    case 2 % MIXED
        send(in_1, in_2);
        returnCode = sendToVREP(robot, in_3.TopicName, in_4);
end
end
end

```

La función *sendToVREP* es la que se encarga de seleccionar la función auxiliar a la que habrá que llamar en función del *topic* pedido. Su cabecera se muestra a continuación.

```
function [returnCode] = sendToVREP(robot , publish , msg)
```

Lo primero que hacemos es inicializar el código de retorno (para control de errores) y a continuación troceamos el *publisher*, dividiéndolo por cada una de las barras ('/');

```
returnCode = -2;  
publisher = strsplit(publish , '/');
```

Seguidamente, buscamos el *publisher* con la ayuda de varias estructuras *switch* anidadas, tal y como se muestra en las siguientes líneas. Una vez encontrada, llamamos a la función auxiliar correspondiente, con los parámetros contenidos en el mensaje necesarios. Solo mostramos dos ejemplos para evitar ser reiterativos.

```
switch(char(publisher(2)))  
    case 'mobile_base'  
        switch(char(publisher(3)))  
            case 'commands'  
                switch(char(publisher(4)))  
                    case 'velocity'  
                        returnCode = setKobukiVelocity(robot , ...  
                            msg.Linear.X, msg.Angular.Z);  
                    [...]  
                end  
            case 'arm_1_joint'  
                switch(char(publisher(3)))  
                    case 'command'  
                        returnCode = setWidowXJointPosition(robot , 'arm' ,...  
                            msg.Data);  
                    end  
                end  
            [...]  
        end  
    end
```

En caso de no encontrar el *topic* definido en el *publisher*, lanzamos un error y notificamos al usuario de la situación. Nótese que esto acabará con la ejecución del programa del usuario:

```
if(returnCode == -2)  
    error('SEND: not implemented topic');  
end  
end
```

La función *sendToVREP* tiene a su disposición un conjunto de funciones auxiliares para realizar las distintas tareas. Este conjunto de funciones es el que se muestra en el cuadro 4.2.

No vamos a describir una por una estas funciones, puesto que la memoria quedaría demasiado extensa. Además, la implementación de las mismas sigue prácticamente el

Función Auxiliar	Descripción
<i>setKobukiVelocity</i>	Manda comandos de velocidad a la base Kobuki
<i>setKobukiLed</i>	Manda comandos de estado a los LEDs de la base Kobuki
<i>setKobukiMotorsEnabled</i>	Habilita o deshabilita los motores de Kobuki
<i>resetKobukiOdometry</i>	Reinicia la odometría de Kobuki
<i>setWidowXJointPosition</i>	Establece una posición objetivo para una articulación de WidowX
<i>addVREPStatusBarMessage</i>	Añade un mensaje a la barra de estado de V-REP
<i>manageAuxiliaryVREPConsole</i>	Gestiona la consola auxiliar de V-REP

Cuadro 4.2: Listado de funciones auxiliares a *sendToVREP*.

mismo esquema (uso de señales para enviar los datos), así que no aporta demasiado el describirlas todas.

Aún así, vamos a echar un vistazo por encima a la función *setKobukiVelocity*, para así hacernos una idea de cómo están las demás. La cabecera de la misma se muestra en la página siguiente.

```
function [returnCode] = setKobukiVelocity(robot , linearX , angularZ)
```

Lo primero que necesitamos es definir la distancia entre las ruedas y el radio de las mismas, en metros:

```
wheelbase = 0.23;
R = 0.035;
```

Seguidamente, inicializamos en su caso la variable de control o la incrementamos en una unidad, teniendo en cuenta un máximo. Recordemos que esta variable de control sirve para saber si el usuario ha enviado un nuevo comando pasados 0,60 segundos, tal y como vimos en la fase de modelado:

```
persistent lastCommand;
if isempty(lastCommand)
    lastCommand = 0;
else
    lastCommand = mod(lastCommand + 1, 10241024);
end
```

Ya podemos calcular la velocidad que hemos de aplicar a cada uno de los motores de la base Kobuki, teniendo en cuenta la velocidad lineal y angular dada. Para ello, partimos de las siguientes expresiones [14]:

$$vel_ang_motor_izq = vel_lin_deseada - \frac{dist_entre_ruedas * vel_ang_deseada}{2 * radio_rueda}$$

$$vel_ang_motor_der = vel_lin_deseada + \frac{dist_entre_ruedas * vel_ang_deseada}{2 * radio_rueda}$$

Y solo tenemos que añadirlas a nuestra función de MATLAB.


```

leftMotorSpeed = (linearX - (wheelbase / 2) * angularZ) / R;
rightMotorSpeed = (linearX + (wheelbase / 2) * angularZ) / R;

```

Una vez que tenemos calculadas las velocidades angulares a aplicar a cada una de las ruedas, las empaquetamos en un *string* y establecemos este último en la señal que hemos definido para tal fin en la fase de modelado. Obsérvese que la ejecución de esta función (*setKobukiVelocity*) precisa de una sola llamada a la API remota de V-REP. Otra forma de hacerlo sería con una llamada por cada motor de Kobuki, pero eso enlentecería en exceso la ejecución de nuestra función.

```

wheelSpeedData = robot.vrep.simxPackFloats([rightMotorSpeed, ....
leftMotorSpeed, lastCommand]);

```

```

result = robot.vrep.simxSetStringSignal(robot.v_clientID, ...
'Turtlebot2_WidowX_kobuki_wheels_speed', wheelSpeedData, ...
robot.vrep.simx_opmode_oneshot);

```

Para acabar, realizamos una simple comprobación de errores, y en caso de que lo haya, establecemos el valor '-1' en la variable de retorno:

```

returnCode = 0;
if(result ~= robot.vrep.simx_return_ok)
    returnCode = -1;
end
end

```

4.2.5. *receive*

Otra de las funciones necesarias es *receive*. Nos va a permitir recibir mensajes a través de los *topics* de nuestro conjunto robótico.

Esta función consta de tres parámetros de entrada. El primero se corresponde con el objeto "robot", el segundo con el *subscriber* correspondiente y el tercero es un tiempo de espera, *timeout*. Si pasado ese tiempo de espera no se ha recibido ningún mensaje, se lanza un error y el programa termina. Si no se especifica este tiempo, por defecto será infinito. La cabecera de la misma se muestra en el siguiente recuadro:

```

function [out_1, out_2, out_3] = receive(robot, in_1, in_2, in_3)
% {VREP Mode} [msg, returnCode] = RECEIVE(robot, sub, timeout)
% {REAL Mode} [msg, returnCode] = RECEIVE(robot, sub, timeout)
% {MIXED Mode} [r_msg, v_msg, returnCode] = RECEIVE(robot, sub, timeout)

```

En el código de la misma, lo primero que nos encontramos es la comprobación de si el usuario ha pasado el *timeout* por parámetro. Seguidamente, nos encontramos una estructura *switch*, donde seleccionamos el trozo de código a ejecutar en función del modo seleccionado. Si estamos en modo V-REP, llamamos a la función *receiveFromVREP*, que nos devolverá el mensaje (si es que lo hay):

```

% Checking if timeout is specified or not
if(nargin < 3)
    timeout = Inf;
end
% Switching operating mode
switch (robot.operating_mode)
    case 0 % V-REP

        % Receiving message from V-REP
        [msg, errorVREP] = robot.receiveFromVREP(in_1.TopicName, ...
            timeout);

        % Checking for errors
        if(errorVREP ~= 0)
            if(errorVREP == 1)
                error('RECEIVE: No events occurred');
            else
                error('RECEIVE: Error in communication.');

```

El segundo caso con el que nos encontramos es con que el modo seleccionado sea *real*, por lo que llamamos directamente a la función *receive* de la *Robotics System Toolbox*. Obsérvese que en este caso el valor de retorno siempre vale '0', puesto que en caso de error será la función llamada la que detenga la ejecución del programa:

```

    case 1 % ROS
        out_1 = receive(in_1, timeout);
        out_2 = 0;          % Return code
        out_3 = [];
```

El último caso que nos encontramos es que esté habilitado el modo *mixed*, en cuyo caso habrá que ejecutar, por un lado, la llamada a la función *receive* de la *Robotics System Toolbox*, y por otro a *receiveFromVREP*:

```

    case 2 % Mixed
        out_1 = receive(in_1, timeout);
        [out_2, errorVREP] = robot.receiveFromVREP(in_2.TopicName, ...
            timeout);
```

Nótese que en este caso, la función devuelve dos mensajes distintos. Esto es debido a que puede haber variaciones entre lo medido por los sensores del robot simulado con respecto a lo medido por los sensores del robot real.

Al igual que para el caso en el que teníamos el modo *vrep*, comprobamos en busca de errores (esencialmente, si el *timeout* ha expirado:

```

        % Checking for errors
        if(errorVREP ~= 0)
            if(errorVREP == 1)
                error('RECEIVE: No events occurred');
            else
                error('RECEIVE: Error in communication.');
            end
        else
            % ReturnCode
            out_3 = 0;
        end
    end
end

```

Con respecto a la función *receiveFromVREP*, nos encontramos con que su cabecera es la siguiente:

```

function [v_msg, returnCode] = receiveFromVREP(robot, subscrib, timeout)

```

Lo primero que hacemos en la misma es “trocear” el *subscriber*. A continuación, simplemente tenemos que llamar a la función auxiliar correspondiente al mismo (solo se muestran algunos ejemplos, más adelante veremos el listado completo en una tabla):

```

    subscriber = strsplit(subscrib, '/');
    switch (char(subscriber(2)))
        case 'odom'
            [v_msg, returnCode] = robot.getKobukiOdometryMsg();
            [...]
        case 'mobile_base'
            switch (char(subscriber(3)))
                case 'sensors'
                    switch (char(subscriber(4)))
                        case 'core'
                            [v_msg, returnCode] = ...
                                robot.getKobukiSensorState();
                        case 'imu_data'
                            [v_msg, returnCode] = robot.getKobukiImuMsg();
                    end
                [...]
            end
        otherwise
            error('RECEIVE: topic not implemented.');
    end
end

```

El ejemplo de función que vamos a analizar es *getKobukiOdometryMsg*, que nos devuelve el mensaje que contiene los datos de odometría de la base Kobuki. Su cabecera es la que se muestra a continuación.

```
function [v_msg, returnCode] = getKobukiOdometryMsg(robot)
```

Lo primero que hacemos es obtener de V-REP la señal que contiene los datos que necesitamos. Para ello, usamos la función *simxGetStringSignal*, tal y como viene siendo de costumbre:

```
[result , poseQuaternionData] = ...  
robot.vrep.simxGetStringSignal(robot.v_clientID , ...  
'Turtlebot2_WidowX_kobuki_quaternion_vel_accel', ...  
robot.vrep.simx_opmode_buffer);
```

A continuación, desempaquetamos los datos y los asignamos a distintas variables que harán el código mucho más legible:

```
poseQuaternionVel = robot.vrep.simxUnpackFloats(poseQuaternionData);  
x = poseQuaternionVel(1);  
y = poseQuaternionVel(2);  
z = poseQuaternionVel(3);  
xq = poseQuaternionVel(4);  
yq = poseQuaternionVel(5);  
zq = poseQuaternionVel(6);  
wq = poseQuaternionVel(7);  
lin_vel_x = poseQuaternionVel(8);  
lin_vel_y = poseQuaternionVel(9);  
lin_vel_z = poseQuaternionVel(10);  
ang_vel_x = poseQuaternionVel(11);  
ang_vel_y = poseQuaternionVel(12);  
ang_vel_z = poseQuaternionVel(13);
```

Lo siguiente que hacemos es crear un nuevo mensaje, usando la función *rosmmessage* de la *Robotics System Toolbox*, con su tipo correspondiente. Una vez que tenemos el mensaje vacío, procedemos a asignar a sus campos los valores que hemos recibido de V-REP:

```
v_msg = rosmmessage('nav_msgs/Odometry');  
v_msg.ChildFrameId = 'base_footprint';  
v_msg.Header = robot.getMessageHeader();  
v_msg.Header.FrameId = 'odom';  
  
v_msg.Pose.Pose.Position.X = x;  
v_msg.Pose.Pose.Position.Y = y;  
v_msg.Pose.Pose.Position.Z = z;  
  
v_msg.Pose.Pose.Orientation.X = xq;  
v_msg.Pose.Pose.Orientation.Y = yq;  
v_msg.Pose.Pose.Orientation.Z = zq;  
v_msg.Pose.Pose.Orientation.W = wq;
```

Continuamos asignando campos (los datos de la matriz de covarianza han sido extraídos del robot real).

```

v_msg.Pose.Covariance = ...
    [0.1 0 0 0 0 0 ...
     0 0.1 0 0 0 0 ...
     0 0 1.7976931348623157e+308 0 0 0 ...
     0 0 0 1.7976931348623157e+308 0 0 ...
     0 0 0 0 1.7976931348623157e+308 0 ...
     0 0 0 0 0 0.05]';

v_msg.Twist.Twist.Linear.X = lin_vel_x;
v_msg.Twist.Twist.Linear.Y = lin_vel_y;
v_msg.Twist.Twist.Linear.Z = lin_vel_z;

v_msg.Twist.Twist.Angular.X = ang_vel_x;
v_msg.Twist.Twist.Angular.Y = ang_vel_y;
v_msg.Twist.Twist.Angular.Z = ang_vel_z;

v_msg.Twist.Covariance = zeros(36, 1);

```

Acabamos nuestra función comprobando si ha ocurrido algún error a la hora de recibir los datos desde V-REP:

```

returnCode = 0;
if(result ~= robot.vrep.simx_return_ok)
    returnCode = -1;
end
end

```

Esta función es solo una de las funciones auxiliares disponibles para *receiveFromVREP*. Evidentemente, estas funciones están ocultas para el usuario, ya que lo que pretendemos es que utilice la función *receive* con el *topic* correspondiente.

En el cuadro 4.3 se recogen las implementadas hasta el momento. En su mayoría devuelven mensajes de Kobuki, pero también tenemos una que devuelve los estados de las artiuclaciones (*getJointStates*), que incluye las del brazo articulado WidowX.

Observe que existen cuatro funciones (*getKobukiBumperEvent*, *getKobukiButtonEvent*, *getKobukiCliffEvent*, *getKobukiWheelDropEvent*) que solo devuelven un mensaje si se ha producido un evento en un determinado tiempo. Como ya sabemos, este tiempo puede ser, o bien definido en la llamada a *receive* por el usuario, o bien infinito. Si pasado ese tiempo no se ha producido un evento, se lanzará un error con la correspondiente detención de la ejecución del programa.

Por otro lado, tenemos tres funciones cuyo cometido es generar mensajes asociados al simulador. La primera de ellas (*getKobukiSimulationPose*) devuelve la pose de Kobuki en la escena, usando la función que V-REP nos proporciona para ello. Esto quiere decir que no es una pose calculada en base a la odometría, si no que se trata de una pose “real”. Por otro lado, tenemos el tiempo de simulación (*getSimulationTimeMsg*) y el tiempo del último comando (*getLastCmdTimeMsg*).

Función Auxiliar	Descripción
<i>getKobukiOdometryMsg</i>	Devuelve un mensaje con la odometría de Kobuki
<i>getJointStatesMsg</i>	Devuelve un mensaje con los estados de las articulaciones (Kobuki y WidowX de forma alterna)
<i>getHokuyoDataMsg</i>	Devuelve un mensaje con los datos del sensor Hokuyo
<i>getKobukiSensorState</i>	Devuelve un mensaje con datos de los sensores de Kobuki
<i>getKobukiImuMsg</i>	Devuelve un mensaje con datos de la IMU
<i>getKobukiImuRawMsg</i>	Devuelve un mensaje con datos de la IMU (con ruido)
<i>getKobukiBumperEvent</i>	Devuelve un mensaje si se produce un evento en algún <i>bumper</i>
<i>getKobukiButtonEvent</i>	Devuelve un mensaje si se produce un evento en algún botón de Kobuki
<i>getKobukiCliffEvent</i>	Devuelve un mensaje si se produce un evento en algún <i>cliff</i>
<i>getKobukiWheelDropEvent</i>	Devuelve un mensaje si se produce un evento en algún <i>wheel drop</i>
<i>getKobukiControllerInfo</i>	Devuelve un mensaje con la información del controlador de Kobuki
<i>getKobukiVersionInfo</i>	Devuelve un mensaje con la versión de Kobuki
<i>getKinectImageMsg</i>	Devuelve un mensaje con una imagen de Kinect
<i>getKobukiSimulationPoseMsg</i>	Devuelve un mensaje con la pose de Kobuki en la escena del simulador
<i>getSimulationTimeMsg</i>	Devuelve un mensaje con el tiempo de simulación
<i>getLastCmdTimeMsg</i>	Devuelve el tiempo en el que ocurrió el último comando

Cuadro 4.3: Listado de funciones auxiliares a *receiveFromVREP*.

4.2.6. *rossvcclient*

Además de *topics*, nuestro conjunto robótico dispone de ciertos *servicios*. Estos servicios pueden ser también llamados desde MATLAB, y afectan principalmente al brazo WidowX. El listado de los mismos puede encontrarse en el manual del usuario.

Para ello, el usuario ha de crear un cliente (algo parecido a un *publisher* o a un *subscriber*) que le permita llamar al servicio deseado. Eso puede hacerlo con la función *rossvcclient*, cuya cabecera se muestra a continuación:

```
function [client , returnCode] = rossvcclient(robot , servicename)
```

El código de la misma es bastante sencillo. Si estamos en el modo *vrep*, creamos una estructura con el nombre del servicio. En otro caso, llamamos a la función *rossvcclient* de la *Robotics System Toolbox*:

```
if robot.operating_mode == 0
    client = struct('ServerName', servicename);
    returnCode = 0;
elseif (robot.operating_mode == 1 || robot.operating_mode == 2)
    client = rossvcclient(servicename);
    returnCode = 0;
end
end
```

4.2.7. *call*

Una vez que hemos creado el objeto que representa al servidor de cada uno de los servicios que vamos a usar (con *rossvcclient*), necesitamos disponer de un mecanismo para llamar a los mismos.

Este es el cometido de la función *call*, que es la que vamos a describir a continuación. Esta función permite llamar a un servicio, con la opción de añadir un mensaje “de petición”. Una vez se haya ejecutado el servicio, recibiremos un mensaje de respues por parte del servidor. La cabecera de la función es la que se muestra a continuación:

```
function [out_1, out_2, out_3] = call(robot , in1 , in2)
```

Dentro del cuerpo de la misma, nos encontramos algo parecido a lo que hemos estado viendo hasta ahora. Tenemos una estructura *switch* que selecciona el trozo de código a ejecutar en función del modo de operación seleccionado.

Si el modo es *vrep*, llamamos a la función auxiliar *callVREPService* con el nombre del servidor:

```
switch (robot.operating_mode)
    case 0 % V-REP
        [out_2, out_1] = callVREPService(robot , in1.ServerName , in2);
        out_3 = [];
```

En caso de modo *real*, llamamos a la función *call* de la *Robotics System Toolbox* con los parámetros correspondientes. Por último, si nos encontramos ante el modo *mixed*, hemos de llamar tanto a la función *call* de la *Robotics System Toolbox* como a la *callVREPSERVICE*. Obsérvese que en este último caso, nuestra función devolverá dos mensajes de respuesta, ya que podría haber diferencias entre lo devuelto por el robot simulado y lo devuelto por el robot real:

```

    case 1 %ROS
        switch nargin
            case 2
                out_1 = call(in1);
            case 3
                out_1 = call(in1 , in2);
        end
        out_2 = 0;          % Return code

    case 2 %MIXED
        switch nargin
            case 2
                out_1 = call(in1);
            case 3
                out_1 = call(in1 , in2);
        end
        [out_3, out_2] = callVREPSERVICE(robot , in1.ServerName , in2);
    end
end
end

```

La función *callVREPSERVICE* es la encargada de llamar a la función auxiliar correspondiente dependiendo del nombre del servicio de entrada. La estructura de la misma es muy similar a la análoga que empleamos para *send* y *receive*. Su cabecera se muestra a continuación:

```

function [response_msg, returnCode] = callVREPSERVICE(robot , service , ...
    request)

```

Dentro de ella, lo primero que hacemos es trocear el nombre del servicio, con la ayuda de *strsplit*:

```

serv = strsplit(service , '/');

```

Una vez hecho eso, utilizamos varias estructuras *switch* anidadas hasta encontrar la coincidencia del nombre del servicio. Obsérvese que además de llamar a la función auxiliar correspondiente, generamos el mensaje de respuesta con la ayuda de la función *rosmesssage* de la *Robotics System Toolbox*:

```

switch (char(serv(2)))
    case 'arm_1_joint'
        switch (char(serv(3)))

```



```

    case 'relax'
        % Relaxing the arm_1_joint (arm joint)
        returnCode = robot.setWidowXJointRelaxed('arm');
        response_msg = rosmesssage('arbotix_msgs/RelaxResponse');

    case 'enable'
        % Enabling/disabling the arm_1_joint (arm joint)
        returnCode = robot.setWidowXJointEnabled('arm', ...
            request.Enable);
        response_msg = rosmesssage('arbotix_msgs/EnableResponse');

    case 'set_speed'
        % Setting max speed of the arm_1_joint (arm joint)
        returnCode = robot.setWidowXJointSpeed('arm', ...
            request.Speed);
        response_msg = ...
            rosmesssage('arbotix_msgs/SetSpeedResponse');

    otherwise
        error('%s %s %s', 'ERROR from "call":', v_serv, ...
            'not supported');
    end
    [...]
    otherwise
        error('%s %s %s', 'ERROR from "call":', v_serv, ...
            'not supported');
    end
end
end

```

En cuanto a las funciones auxiliares, nos encontramos con las presentadas en la tabla 4.4. Obsérvese que solo hay tres, y se corresponden con servicios del brazo articulado WidowX.

Función Auxiliar	Descripción
<i>setWidowXJointRelaxed</i>	Relaja una articulación de WidowX
<i>setWidowXJointEnabled</i>	Habilita o deshabilita una articulación de WidowX
<i>setWidowXJointSpeed</i>	Establece la velocidad máxima de una articulación de WidowX

Cuadro 4.4: Listado de funciones auxiliares a *callFromVREP*.

Vamos a echar un vistazo a una de ellas. La función *setWidowXJointRelaxed* establece el modo de relajación de una articulación de WidowX. Su cabecera es la que se muestra a continuación:

```
function [returnCode] = setWidowXJointRelaxed(robot, joint)
```

Lo primero que hacemos es inicializar las variables que emplearemos para el control de errores:

```
error = 0;
result = robot.vrep.simx_return_ok;
```

Seguidamente, empleamos una estructura *switch* para buscar la articulación en cuestión. Una vez encontrada, establecemos la señal de configuración de esa articulación en el valor '2', que tal y como vimos en el capítulo de modelado, se corresponde con el modo de relajación. Solo se muestra el caso de una articulación, para evitar ser reiterativos:

```
% Searching for the desired joint and changing it's V-REP state to '2'
switch(joint)
    case 'arm'
        result = robot.vrep.simxSetIntegerSignal(robot.v_clientID, ...
            'Turtlebot2_WidowX_widowx_arm_joint_conf', 2, ...
            robot.vrep.simx_opmode_oneshot);
    [...]
    otherwise
        error = 1;
end
```

Por último, comprobamos si se ha producido algún error (al establecer el valor de la señal o que la articulación no ha sido reconocida), y en ese caso, establecemos el valor de *returnCode* a '-1':

```
% Checking for errors
returnCode = error;
if(result ~= robot.vrep.simx_return_ok)
    returnCode = -1;
end
end
```

4.2.8. *rostopic*

En el conjunto de funciones de nuestra *toolbox*, nos encontramos también con *rostopic*. Se trata de una función que devuelve información acerca de los *topics* de nuestro modelo. Su cabecera es la siguiente:

```
function [out_1, returnCode] = rostopic(robot, arg0, arg1)
```

Dentro de la misma, lo primero que hacemos es inicializar el valor de retorno a '-1':

```
returnCode = -1;
```

Segidamente, buscamos el modo de operación. En caso de que estemos en el modo *real* o en el modo *mixed*, llamamos a la función *rostopic* de la *Robotics System Toolbox*, con el número de argumentos que nos hayan pasado.

```

if(robot.operating_mode == 1 || robot.operating_mode == 2)
    disp( '———— REAL ROBOT ———— ');
    if nargin == 2
        rostopic(arg0);
        returnCode = 0;

    elseif nargin == 3
        rostopic(arg0, arg1);
        returnCode = 0;

    end
    disp( '————— ');
end

```

Por otro lado, si el modo seleccionado es *vrep* o *mixed*, hemos de emular la llamada a la función *rostopic* de la *Robotics System Toolbox*. Por ello, comparamos el segundo argumento. Si se trata de 'list', devolvemos la lista con los *topics* implementados. Si por el contrario, se trata de 'type', devolvemos el tipo de un *topic* pasado como el segundo parámetro de la función.

Para ello, hacemos uso de la función auxiliar *rostopicTypeVREP*, que simplemente busca el *topic* en la lista de *topics* y devuelve el tipo correspondiente.

La función *rostopic* de la *Robotics System Toolbox* tiene más opciones, pero las más interesantes para el usuario seguramente sean las que hemos implementado:

```

if(robot.operating_mode == 0 || robot.operating_mode == 2)
    disp( '———— VIRTUAL ROBOT ———— ');
    if(strcmp(arg0, 'list'))
        out_1 = robot.v_topics;
        returnCode = 0;

    elseif (strcmp(arg0, 'type'))
        out_1 = robot.rostopicTypeVREP(arg1);
        returnCode = 0;

    end
    disp( '————— ');
    end
end

```

4.2.9. *rosservice*

Al igual que para los *topics*, tenemos una función que nos devuelve información acerca de los servicios. Su cabecera es la siguiente:

```

function returnCode = rosservice(robot, arg0)

```

En primer lugar, comprobamos que el modo *real* o *mixed* está habilitado. En ese caso, llamamos a la función *rosservice* de la *Robotics System Toolbox*:

```
% ROS and MIXED
if(robot.operating_mode == 1 || robot.operating_mode == 2)
    if(strcmp(arg0, 'list'))
        disp('———— REAL ROBOT ————');
        rosservice(arg0);
        returnCode = 0;
        disp('—————');
    else
        returnCode = -1;
    end
end
end
```

A continuación, comprobamos si el modo *vrep* o *mixed* está habilitado. En ese caso, comprobamos que el argumento de entrada sea igual a “*list*” y mostramos por pantalla los servicios disponibles para el robot simulado:

```
% V-REP and MIXED
if(robot.operating_mode == 0 || robot.operating_mode == 2)
    if(strcmp(arg0, 'list'))
        disp('———— VIRTUAL ROBOT ————');
        for k = 1 : size(robot.v_services)
            disp(robot.v_services(k));
        end
        returnCode = 0;
        disp('—————');
    else
        returnCode = -1;
    end
end
end
end
```

4.2.10. *rosshutdown*

Todo programa que use nuestra *toolbox* (o la *Robotics System Toolbox* de MATLAB) debe acabar con una llamada a *rosshutdown*, que es la encargada de cerrar la conexión. Su cabecera es la que se muestra a continuación:

```
function [returnCode] = rosshutdown(robot)
```

En primer lugar, verificamos si se ha establecido comunicación con el robot real. Para ello, comprobamos que el modo seleccionado sea *real* o *mixed*. En caso de que así sea, llamamos a *rosshutdown* de la *Robotics System Toolbox*:

```
if(robot.operating_mode == 1 || robot.operating_mode == 2)
    rosshutdown;
end
```

Por otro lado, si el modo seleccionado es *vrep* o *mixed*, entonces eso quiere decir que hemos de cerrar la conexión con V-REP. Para ello, lo primero que hacemos es notificar de tal acontecimiento al usuario en la barra de estado de V-REP.

Seguidamente (y si había ya una conexión establecida), mandamos un comando arbitrario para garantizar que el último que se envió ha sido procesado correctamente. Una vez hecho eso, ya podemos finalizar la conexión, con *simxFinish*.

Para acabar, eliminamos el objeto *robot* que fue creado con el constructor de la clase VREP:

```
% V-REP
if(robot.operating_mode == 0 || robot.operating_mode == 2)
    % User's notification
    robot.vrep.simxAddStatusBarMessage(robot.v_clientID, ...
        'Disconnected from Matlab.', robot.vrep.simx_opmode_oneshot);
    if (robot.v_clientID > -1)
        % Before closing the connection to V-REP, make sure that the
        % last command sent out had time to arrive. It can be
        % guaranteed with (for example):
        robot.vrep.simxGetPingTime(robot.v_clientID);
        robot.vrep.simxFinish(robot.v_clientID);
    end
    robot.vrep.delete();
end
```


Capítulo 5

Resultados

En este capítulo de la memoria vamos a analizar los resultados que hemos obtenido gracias a el trabajo desarrollado. En primer lugar, mostraremos los tiempos de respuesta entre MATLAB y V-REP y entre MATLAB y el robot real. A continuación, implementaremos dos aplicaciones que hagan uso del entorno de programación.

5.1. Tiempos de respuesta

Una cuestión interesante es la medida de los tiempos de respuesta entre nuestra *toolbox* y el robot real. Para ello, implementamos en MATLAB un pequeño programa que, utilizando cierto número de muestras, grafique los tiempos de respuesta para distintos *topics*.

La implementación de dicho programa es más bien un ejercicio de MATLAB, así que vamos a obviar su código. La figura 5.1 muestra cinco gráficas para cinco *topics* distintos del robot real, tomando 200 muestras de cada uno. Para la realización de este experimento, hemos utilizado una red Wi-Fi *hot spot* creada con un portátil, y hemos conectado el *netbook* del robot a la misma.

Nótese que cada gráfica tiene en su título una breve descripción del *topic* al que pertenece, además del valor medio de las muestras. A la vista de los resultados, nos movemos alrededor de los 25 milisegundos, exceptuando algunos picos espurios.

Para el caso de Kinect, la media de tiempos es de unos 30 milisegundos, puesto que el hecho de que la imagen circule a través de la red lleva más tiempo que si nos encontramos ante datos de sensores simples.

En cualquier caso, los tiempos se encuentran dentro de los límites aceptables que permiten el control del robot, así que nuestra *toolbox* puede servirnos para controlar el robot real directamente, tal y como pretendíamos. Además, hemos de tener en cuenta que el portátil empleado para ejecutar MATLAB es un poco antiguo, así que es de esperar que con un ordenador más potente se obtengan resultados incluso mejores. También va a influir bastante si la red empleada está congestionada o no, y su velocidad.

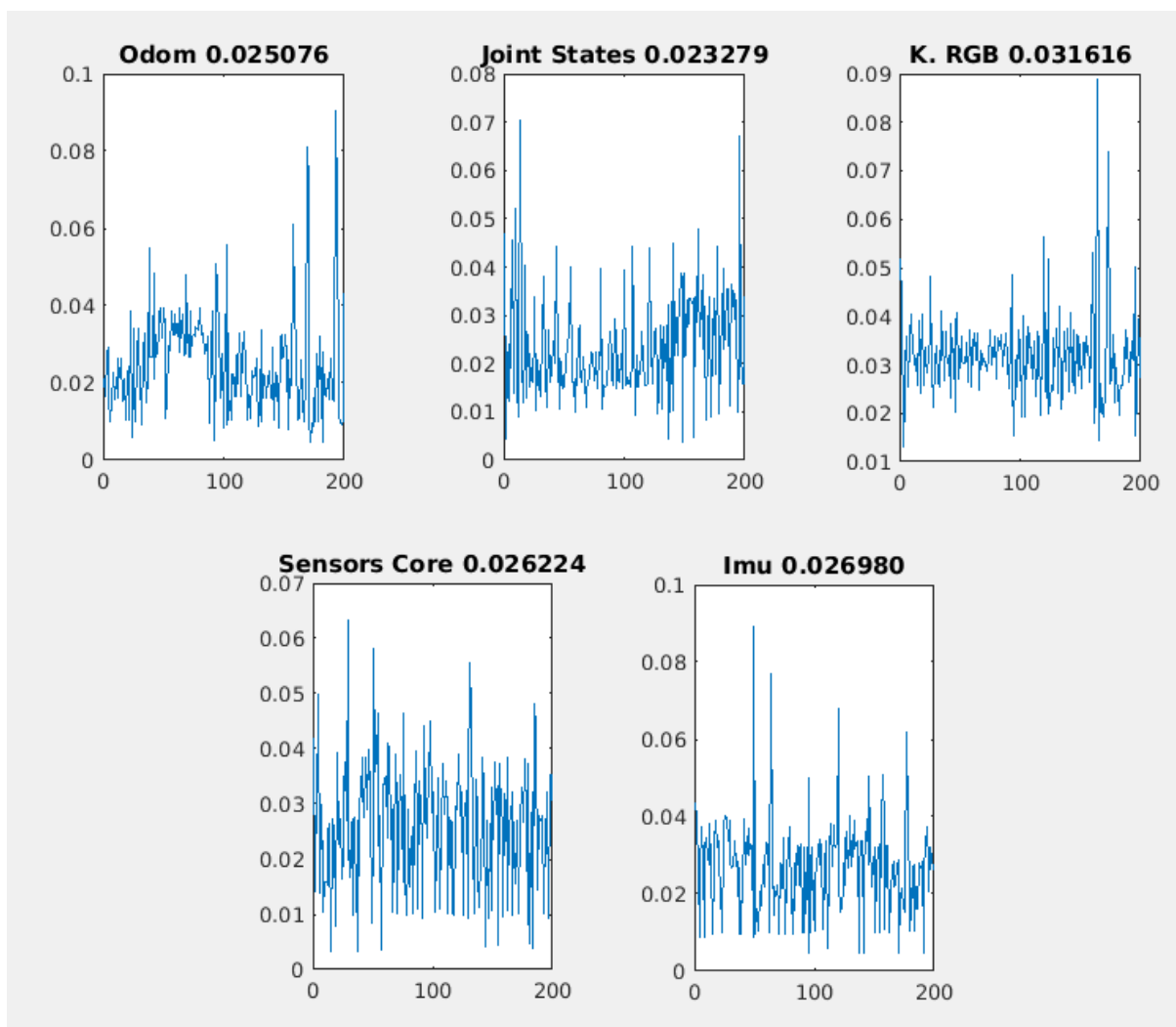


Figura 5.1: Tiempos de respuesta entre MATLAB y el robot real (segundos).

Vamos a realizar mediciones también entre nuestra *toolbox* y el simulador V-REP. Para ello, empleamos el mismo programa, ya que solo tenemos que cambiar el modo de ejecución del mismo, de *real* a *vrep*.

Los resultados pueden verse en la figura 5.2. Nótese que para el caso de “*Joint States*” y para “*Imu*” tenemos un multiplicador de 10^{-3} , por lo que los tiempos están en el mismo orden que los demás *topics*.

Estos tiempos son bastante menores que los que obtuvimos al realizar las mediciones con el robot real. Una de las cuestiones que explica esto es que no nos encontramos en una red Wi-Fi, y que además el ordenador empleado para las pruebas es mucho más rápido.

No obstante, el objetivo no es el de comparar los tiempos del robot real con el simulado, si no darnos cuenta de que los tiempos son razonables para implementar aplicaciones con nuestra *toolbox*, cosa que hemos visto que es así.

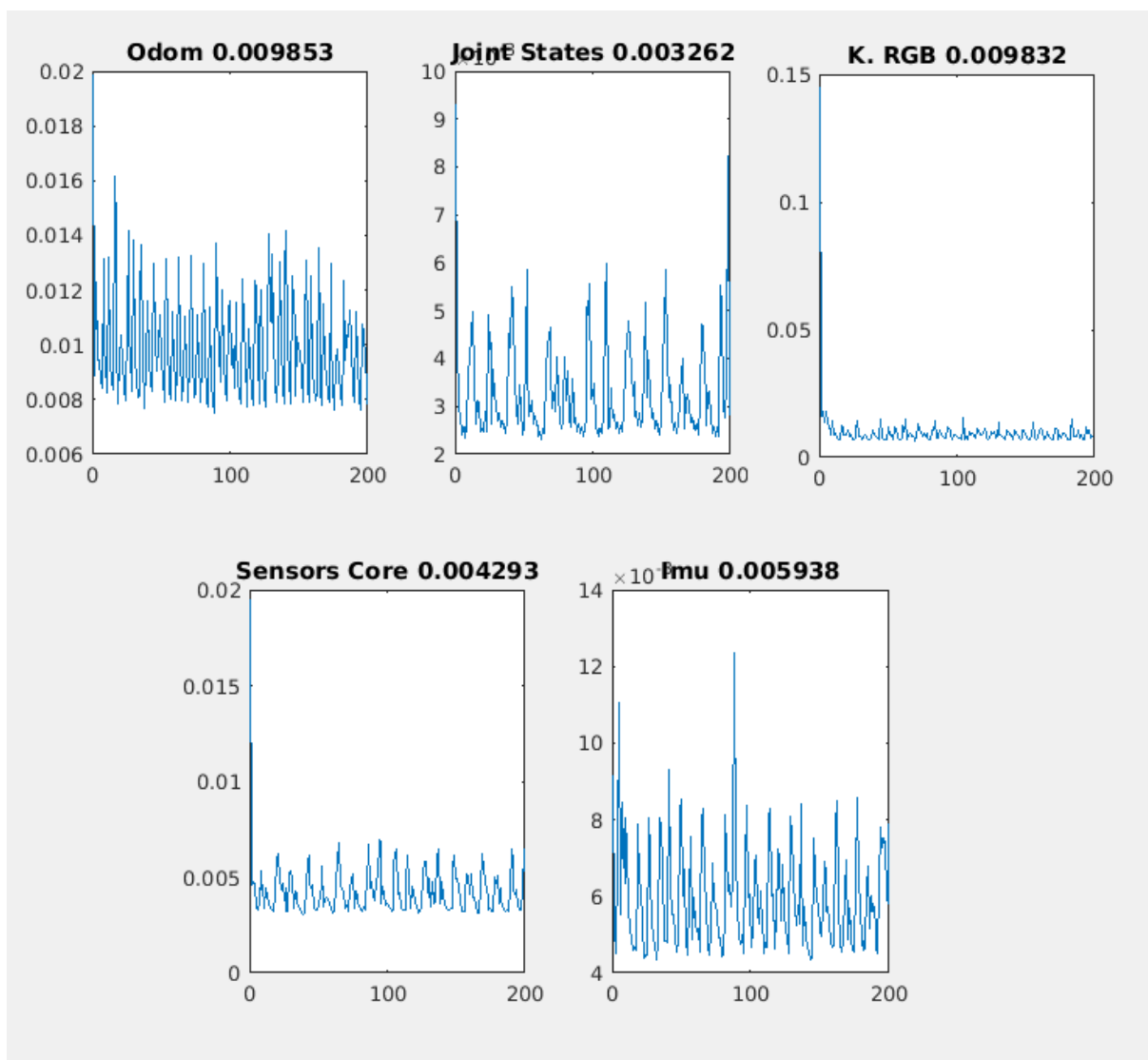


Figura 5.2: Tiempos de respuesta entre MATLAB y V-REP (segundos).

5.2. Primera aplicación

Llegados a este punto, ya tenemos nuestra *toolbox* terminada y hemos comprobado que los tiempos de ejecución de la misma nos permite implementar aplicaciones robóticas complejas.

Una buena forma de comprobar que tanto la *toolbox* como el modelo en V-REP funcionan como se espera es implementar una aplicación. En nuestro caso, vamos a implementar dos. La primera de ellas es la que veremos en esta sección.

Dicha aplicación va a consistir en el traslado de un objeto de un punto a otro de una escena. Para ello, se ha creado la escena que puede verse en la figura 5.3. Esta escena está inspirada en el laboratorio 2.1.5 de la E.T.S.I. Informática de la Universidad de Málaga.

En ella se ha colocado a nuestro conjunto robótico, en el centro. Por otro lado, se han colocado también dos mesas de color verde, con las dimensiones de una mesa auxiliar

comercial. De este modo, sería posible ejecutar esta aplicación usando el robot real en el laboratorio. Como objeto a transportar, hemos creado un cubo de Rubik de 3 centímetros de lado, y lo hemos colocado en la mesa del fondo.

Así pues, tenemos que generar el código MATLAB que, usando nuestra *toolbox*, haga que el conjunto robótico agarre el objeto de la mesa del fondo y lo suelte sobre la de la entrada.

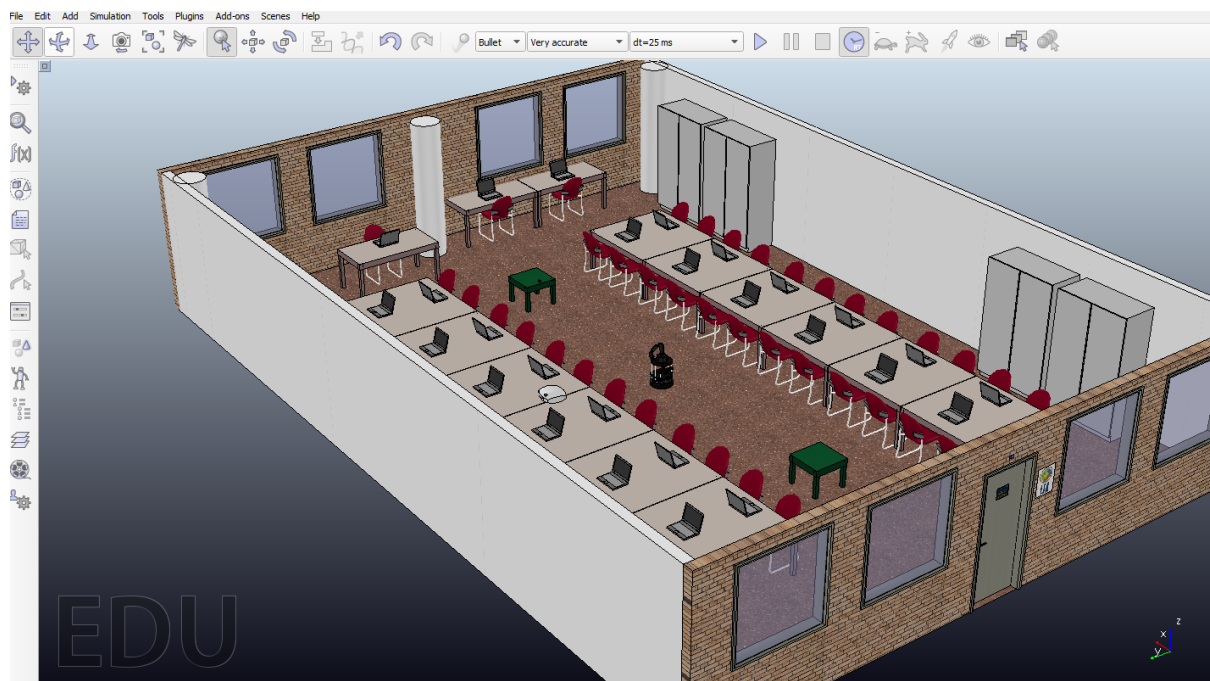


Figura 5.3: Escena creada para la primera aplicación.

Vamos a echarle un vistazo al código. La primera parte, que se corresponde con la inicialización, no vamos a mostrarla. Se trata de una parte común a casi todas las aplicaciones que hagamos, y está explicada en el manual de usuario. Tampoco vamos a explicar los fragmentos de código que escriben en la consola auxiliar de V-REP, por el mismo motivo.

Una vez inicializados todos los *topics* y demás cuestiones, tenemos que hacer que nuestro conjunto robótico se mueva hacia la mesa del fondo. Para ello, vamos a emplear odometría.

Creamos un bucle *while* donde no salgamos hasta que el conjunto robótico haya recorrido 2,915 metros lineales hacia delante. Para ello, recibimos el mensaje por el *topic* de odometría y calculamos la pose y la orientación del robot en base al mensaje que nos acaba de llegar:

```
odom_msg = myTurtle.receive(odom_sub);  
x        = getKobukiPoseAndOrientation(odom_msg);  
  
while x < 2.915 % Meters
```

```

% Reading the current pose and orientation
odom_msg      = myTurtle.receive(odom_sub);
[x, y, theta] = getKobukiPoseAndOrientation(odom_msg);

```

Como medida de seguridad, no está de más emplear algunos sensores para comprobar que no tenemos ningún obstáculo cerca. Para ello, recibimos el mensaje que contiene los datos de los sensores de Kobuki. En caso de que encontremos un obstáculo, retrocedemos y giramos el conjunto robótico. Si no es así, continuamos hacia delante, usando la orientación devuelta por el giroscopio para corregir la pequeñas desviaciones, si las hubiere:

```

% Receiving the sensors core message
sens_msg = myTurtle.receive(sensors_sub);
% Check for obstacle
if(sens_msg.Bumper ~= 0 || sens_msg.Cliff ~= 0)
    % Obstacle
    velocity_msg.Linear.X = -0.2;
    velocity_msg.Angular.Z = 0;
    myTurtle.send(kob_velocity_pub, velocity_msg);
    myTurtle.pause(1.5);

    velocity_msg.Linear.X = 0;
    velocity_msg.Angular.Z = defaultNormalAngularVelocity;
    myTurtle.send(kob_velocity_pub, velocity_msg);
    myTurtle.pause(1.5);
else
    % No obstacle
    velocity_msg.Linear.X = defaultLinearVelocity;
    velocity_msg.Angular.Z = -0.1 * theta;
    myTurtle.send(kob_velocity_pub, velocity_msg);
end
end
end

```

Una vez salimos del bucle *while*, habremos llegado a nuestro primer destino. Por tanto, lo siguiente que hacemos es parar la tortuga, poniendo las velocidades a '0':

```

% Stopping the turtle
velocity_msg.Linear.X = 0;
velocity_msg.Angular.Z = 0;
myTurtle.send(kob_velocity_pub, velocity_msg);
myTurtle.pause(1);

```

Ya estamos en la mesa donde se encuentra el objeto en cuestión, así que lo que tenemos que hacer es agarrar el objeto con el brazo WidowX. Para ello, colocamos el brazo en la posición adecuada, mandando a cada una de las articulaciones el ángulo correspondiente.

Cuando ya tenemos el brazo en su posición, solo nos falta cerrar la pinza lo suficiente para que el objeto quede agarrado y no se nos escape¹.

¹El hecho de agarrar objetos en un simulador suele dar bastantes problemas por las fricciones.

```

% Biceps
wid_joint_cmd_msg.Data = pi / 2;          % 90 deg.
myTurtle.send(wid_3_pub, wid_joint_cmd_msg);
myTurtle.pause(1.5);

% Forearm
wid_joint_cmd_msg.Data = - (pi / 2);     % -90 deg.
myTurtle.send(wid_4_pub, wid_joint_cmd_msg);
myTurtle.pause(1.5);

% Arm
wid_joint_cmd_msg.Data = 0;              % 0 deg.
myTurtle.send(wid_1_pub, wid_joint_cmd_msg);
myTurtle.pause(1.5);

% Shoulder
wid_joint_cmd_msg.Data = deg2rad(65);    % 65 deg.
wid_speed_msg.Speed = deg2rad(10);
myTurtle.call(wid_2_s_serv, wid_speed_msg);
myTurtle.send(wid_2_pub, wid_joint_cmd_msg);
myTurtle.pause(5);
wid_speed_msg.Speed = deg2rad(100);     % 100 deg.
myTurtle.call(wid_2_s_serv, wid_speed_msg);

% Wrist
wid_joint_cmd_msg.Data = 0;              % 0 deg.
myTurtle.send(wid_5_pub, wid_joint_cmd_msg);
myTurtle.pause(1.5);

% Gripper
wid_joint_cmd_msg.Data = 1;              % 44 percent opened.
myTurtle.send(wid_gripper_pub, wid_joint_cmd_msg);
myTurtle.pause(2);

```

Ya tenemos agarrado el objeto, así que colocamos el brazo en una posición que nos permita un transporte seguro del mismo. Para ello, colcamos el brazo semiflexionado con la pinza orientada hacia el techo:

```

% Shoulder
wid_joint_cmd_msg.Data = deg2rad(-15);
myTurtle.send(wid_2_pub, wid_joint_cmd_msg);

% Forearm
wid_joint_cmd_msg.Data = pi / 2;
myTurtle.send(wid_4_pub, wid_joint_cmd_msg);

% Biceps
wid_joint_cmd_msg.Data = -(pi / 2);

```

```

myTurtle.send(wid_3_pub, wid_joint_cmd_msg);

% Wrist
wid_joint_cmd_msg.Data = 0;
myTurtle.send(wid_5_pub, wid_joint_cmd_msg);

% Shoulder
wid_joint_cmd_msg.Data = -(pi / 2);
myTurtle.send(wid_2_pub, wid_joint_cmd_msg);
myTurtle.pause(1);

```

A continuación, giramos el conjunto robótico 180 grados, para poder llevar el objeto hacia la otra mesa. Obsérvese que utilizamos los datos de la odometría (donde se incluye la orientación) para realizar el giro. Además, cuando estemos cerca del objetivo (180 grados) bajaremos la velocidad para evitar pasarnos:

```

% Reading the first orientation
odom_msg = myTurtle.receive(odom_sub);
[~, ~, theta] = getKobukiPoseAndOrientation(odom_msg);

while theta > 180.5 || theta < 179.5
    % Reading the current orientation
    odom_msg = myTurtle.receive(odom_sub);
    [~, ~, theta] = getKobukiPoseAndOrientation(odom_msg);
    if theta > 180.5
        if(abs(theta - 180.5) > 15)
            angularVelocity = -defaultNormalAngularVelocity;
        else
            angularVelocity = -defaultSlowAngularVelocity;
        end
    else
        if(abs(theta - 179.5) > 15)
            angularVelocity = defaultNormalAngularVelocity;
        else
            angularVelocity = defaultSlowAngularVelocity;
        end
    end

    velocity_msg.Linear.X = 0;
    velocity_msg.Angular.Z = angularVelocity;
    myTurtle.send(kob_velocity_pub, velocity_msg);
end

```

Una vez hecho el giro, paramos el conjunto robótico:

```

velocity_msg.Linear.X = 0;
velocity_msg.Angular.Z = 0;
myTurtle.send(kob_velocity_pub, velocity_msg);

```

Ahora, movemos el conjunto robótico hasta la ubicación de la segunda mesa, de forma análoga a como lo hicimos para la primera. Obsérvese que establecemos el valor objetivo de x en -3 metros, ya que esta mesa está por detrás de la posición inicial del conjunto robótico:

```
% First pose and orientation acquirement
odom_msg = myTurtle.receive(odom_sub);
[x, y, theta] = getKobukiPoseAndOrientation(odom_msg);

while x > -3
    % Current pose and orientation acquirement
    odom_msg = myTurtle.receive(odom_sub);
    [x, y, theta] = getKobukiPoseAndOrientation(odom_msg);

    sens_msg = myTurtle.receive(sensors_sub);

    % Checking for obstacle
    if(sens_msg.Bumper ~= 0 || sens_msg.Cliff ~= 0)
        velocity_msg.Linear.X = -defaultNormalAngularVelocity;
        velocity_msg.Angular.Z = 0;
        myTurtle.send(kob_velocity_pub, velocity_msg);
        myTurtle.pause(1.5);

        velocity_msg.Linear.X = 0;
        velocity_msg.Angular.Z = defaultNormalAngularVelocity;
        myTurtle.send(kob_velocity_pub, velocity_msg);

        myTurtle.pause(1.5);
    else
        velocity_msg.Linear.X = defaultLinearVelocity;
        velocity_msg.Angular.Z = 0;
        myTurtle.send(kob_velocity_pub, velocity_msg);
    end
end
```

Una vez hemos llegado a la segunda mesa, paramos el conjunto robótico, de nuevo estableciendo las velocidades a '0':

```
% Stopping the turtle
velocity_msg.Linear.X = 0;
velocity_msg.Angular.Z = 0;
myTurtle.send(kob_velocity_pub, velocity_msg);
myTurtle.pause(1);
```

Para ir acabando, soltamos el objeto sobre la segunda mesa, de nuevo dando valores a las articulaciones de WidowX:

```
% Shoulder
wid_joint_cmd_msg.Data = deg2rad(40);
```

```

myTurtle.send(wid_2_pub, wid_joint_cmd_msg);

% Biceps
wid_joint_cmd_msg.Data = deg2rad(50);
myTurtle.send(wid_3_pub, wid_joint_cmd_msg);

% Forearm
wid_joint_cmd_msg.Data = -(pi / 2);
myTurtle.send(wid_4_pub, wid_joint_cmd_msg);
myTurtle.pause(2);

% Gripper
wid_joint_cmd_msg.Data = 0;
myTurtle.send(wid_gripper_pub, wid_joint_cmd_msg);
myTurtle.pause(2);

```

Colocamos las articulaciones de WidowX en su posición *home*, es decir, todas a '0':

```

wid_joint_cmd_msg.Data = 0;

% Shoulder
myTurtle.send(wid_2_pub, wid_joint_cmd_msg);

% Forearm
myTurtle.send(wid_4_pub, wid_joint_cmd_msg);

% Biceps
myTurtle.send(wid_3_pub, wid_joint_cmd_msg);
myTurtle.pause(2);

```

Y acabamos el programa llamando a los servicios de relajación de cada articulación. Esto último es simplemente para comprobar que funcionan tal y como se espera:

```

myTurtle.call(wid_1_r_serv, wid_relax_msg);
myTurtle.call(wid_2_r_serv, wid_relax_msg);
myTurtle.call(wid_3_r_serv, wid_relax_msg);
myTurtle.call(wid_4_r_serv, wid_relax_msg);
myTurtle.call(wid_5_r_serv, wid_relax_msg);

```

En la página siguiente nos encontramos con dos figuras (figura 5.4 y figura 5.5) en las que podemos ver dos momentos de la ejecución del programa.

En la primera de ellas (5.4) se puede ver el conjunto robótico desde la primera mesa, sobre la cual está colocado el cubo de Rubik que utilizamos como objeto a portar.

En la segunda, (5.5) vemos al conjunto robótico transportando el objeto de una mesa a otra. Obsérvese que en esta figura aparece la consola auxiliar de V-REP, en la que se va mostrando en todo momento la pose y la orientación del conjunto robótico.

Tanto el fichero con el código de la aplicación como el fichero con la escena empleada están incluidos en el paquete de la *toolbox* listos para su ejecución.

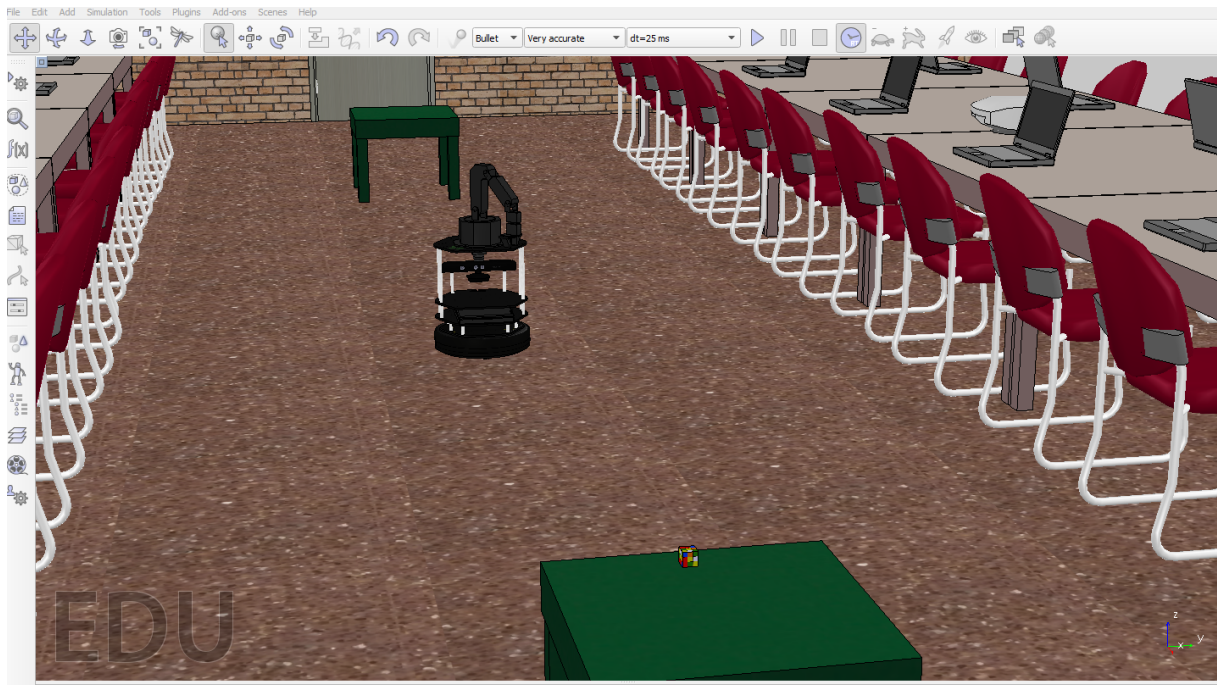


Figura 5.4: Vista del conjunto robótico desde la primera mesa, donde se encuentra el cubo de Rubik.

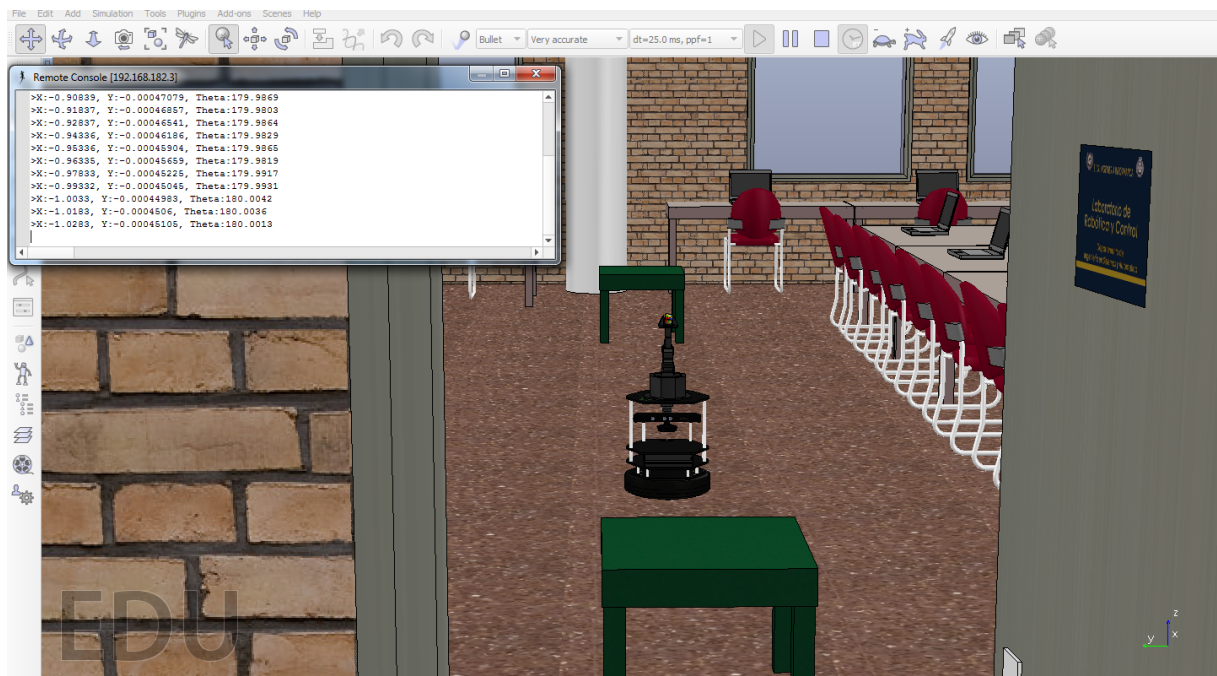


Figura 5.5: Vista del conjunto robótico transportando el objeto de una mesa a otra.

5.3. Segunda aplicación

Con la primera aplicación hemos probado el brazo articulado y la movilidad de nuestro conjunto robótico. En esta segunda aplicación, vamos a emplear uno de sus sensores, para

implementar un algoritmo de navegación reactiva [18]. Este algoritmo hará que nuestro conjunto robótico se traslade de un punto a otro de una escena.

El sensor que vamos a utilizar es Kinect, y nos ayudará a evitar los obstáculos en la medida de lo posible. Con respecto al escenario, vamos a emplear el que se muestra en la figura 5.6. Se trata de una representación de varios laboratorios, ubicados en la tercera planta del módulo 2 de la E.T.S.I. Informática de la Universidad de Málaga.

El objetivo es que el conjunto robótico se mueva desde su posición inicial (laboratorio de la derecha de abajo en la figura) hasta el laboratorio de la izquierda de arriba.

Se han colocado ciertos obstáculos en su recorrido (especie de balizas) que tendrá que esquivar. Además, consta de varios *targets*, porque de otra manera es muy posible que el algoritmo encuentre mínimos locales, y no sea capaz de alcanzar el objetivo final.

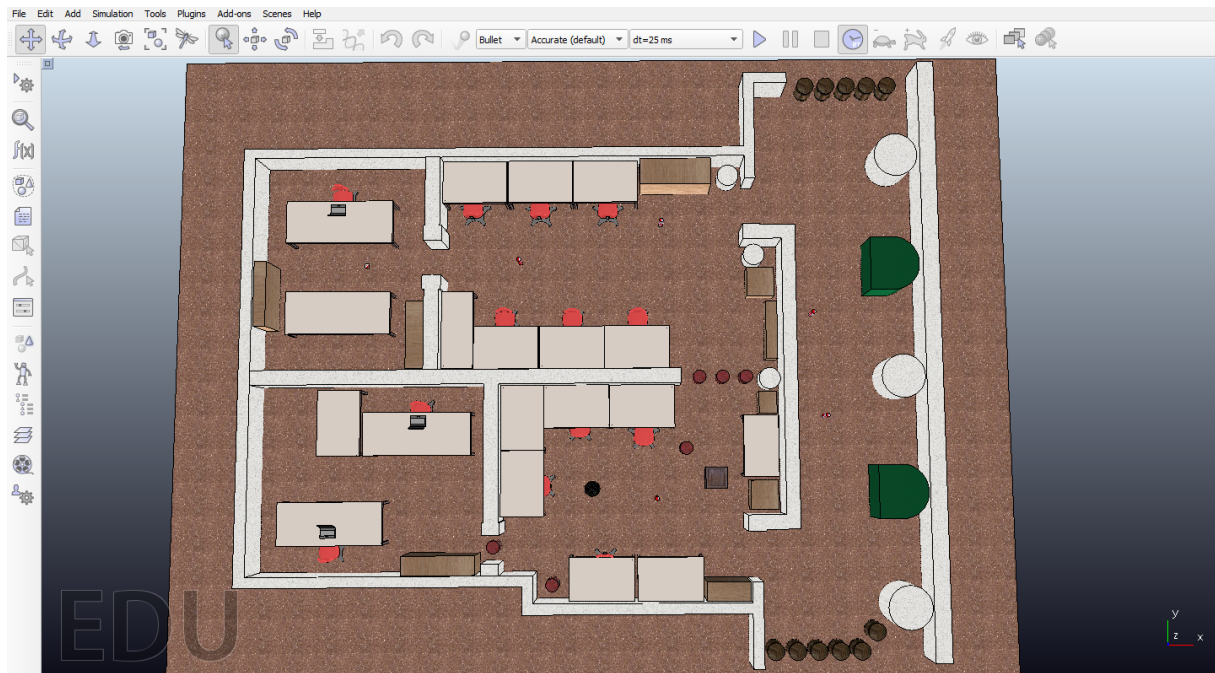


Figura 5.6: Escenario creado para la segunda aplicación.

Al igual que hicimos con la aplicación anterior, omitimos la parte de inicialización en la explicación, ya que suele ser parecida en todas las aplicaciones. En el caso que nos ocupa, creamos un bucle *for* que recorra los objetivos, y computamos la distancia que tenemos actualmente con el que toca en la iteración que estemos:

```
for i = (1:size(targets, 1))
    target = targets(i, :);

    odom_msg = myTurtle.receive(odom_sub);
    [x, y] = getKobukiPoseAndOrientation(odom_msg);

    % Computing the distance to the target point
    d = sqrt(sum(([x, y] - target) .^ 2));
```

Seguidamente, entramos en un bucle *while*, del que no saldremos mientras la distancia con el objetivo sea mayor de cinco centímetros. Lo primero que hacemos en el cuerpo de este bucle es leer la pose actual y calcular la distancia que tenemos con el objetivo.

Una vez hecho eso, leemos el mensaje de Kinect (solo recibimos la imagen de profundidad) y nos quedamos con la fila central de la misma para simplificar los cálculos (es lo que guardamos en la matriz fila *vision_line*):

```
while d > 0.05
    % Reading the current pose and orientation
    odom_msg      = myTurtle.receive(odom_sub);
    [x, y, theta] = getKobukiPoseAndOrientation(odom_msg);

    % Computing the distance to the target point
    d = sqrt(sum(([x, y] - target) .^ 2));

    % Receiving the Kinect depth message
    kinect_depth_msg = myTurtle.receive(kinect_depth_sub);
    img = readImage(kinect_depth_msg);

    % Taking of the vision line
    vision_line = img(240, :);
```

Una vez tenemos la matriz fila, buscamos el mayor valor de la misma, que será el correspondiente al objeto que tengamos más cerca. Si no está lo suficientemente cerca, interpretamos que no nos “estorba” y por tanto la fuerza virtual de repulsión del mismo será ‘0’:

```
    % Searching of the closest point
    [max_value, max_index] = max(vision_line);

    % Virtual repulsion force calculation
    if(max_value < 120)
        virtual_repulsion_force = 0;
    else
        virtual_repulsion_force = double(KR * (max_value - 120));
    end
```

A continuación, calculamos una estimación del ángulo al que se encuentra el objeto con respecto al conjunto robótico. Para ello, tenemos en cuenta que el ángulo de visión de Kinect es de 58 grados y que la matriz fila consta de 640 puntos.

Con la estimación del ángulo obtenido, creamos un vector con longitud proporcional a la fuerza de repulsión calculada y dirección opuesta al ángulo del objeto (por eso le sumamos π al ángulo calculado).

Este vector está calculado en el sistema de referencia del conjunto robótico.

```
    % Closest object angle
    object_angle_estim = deg2rad(((320 - max_index) * (28.5 / 320)));
```

```

% Virtual repulsion vector in robot's reference
frepx = virtual_repulsion_force * (cos(object_angle_estim + ...
    deg2rad(theta) + pi));
frepy = virtual_repulsion_force * (sin(object_angle_estim + ...
    deg2rad(theta) + pi));

```

Ahora pasamos a calcular el vector de atracción virtual, que irá en la dirección del *target* al que estemos intentando llegar en este momento. Como fuerza de atracción, empleamos la distancia calculada anteriormente.

Si nos encontramos ante el último *target*, la velocidad de avance será menor, para evitar pasarnos. El cálculo del vector es una simple resta de las coordenadas de la pose del conjunto robótico a las coordenadas del *target*:

```

% Virtual attraction force
if (d < 0.5)
    virtual_attraction_force = KA * 0.5;
    if (i == size(targets, 1))
        velocity_msg.Linear.X = defaultLinearVelocity;
    else
        velocity_msg.Linear.X = defaultFastLinearVelocity;
    end
else
    virtual_attraction_force = KA * d;
    velocity_msg.Linear.X = defaultFastLinearVelocity;
end

% Virtual attraction vector in robot's reference
targetx = virtual_attraction_force * (target(1, 1) - x);
targety = virtual_attraction_force * (target(1, 2) - y);

```

Una vez que tenemos los dos vectores virtuales (el de repulsión y el de atracción), hemos de sumarlos para obtener el vector resultante, cuyas coordenadas se almacenan en *ftotx* y *ftoty*. A continuación, calculamos un vector de módulo 10 que tenga como ángulo la orientación que tiene el conjunto robótico actualmente con respecto al punto de partida, y calculamos el ángulo existente entre ambos vectores:

```

% Resultant force
ftotx = frepx + targetx;
ftoty = frepy + targety;

myVectorx = 10 * cos(deg2rad(theta));
myVectory = 10 * sin(deg2rad(theta));

% Resultant angle
angle = atan2(ftoty, ftotx) - atan2(myVectory, myVectorx);

```

Ese ángulo va a servirnos para calcular la velocidad angular a aplicar a nuestro conjunto robótico. En primer lugar, buscamos el camino más corto para alcanzar ese ángulo (ya

que podemos girar en sentido positivo y en sentido negativo), así que si el valor absoluto del ángulo es mayor que π , cambiamos su sentido.

Además, acotamos el ángulo al rango $[-\frac{\pi}{2}, \frac{\pi}{2}]$ para evitar que nuestro conjunto robótico gire excesivamente rápido:

```
if(abs(angle) > pi)
    angular_velocity = -angle;
else
    angular_velocity = angle;
end

if(angular_velocity > pi/2)
    angular_velocity = pi/2;
elseif(angular_velocity < -pi/2)
    angular_velocity = -pi/2;
end
```

Mandamos el comando de velocidad al conjunto robótico (recordemos que la velocidad lineal ha sido establecida cuando hemos calculado el vector de atracción) y con eso cerramos tanto el bucle *while* como el bucle *for*:

```
velocity_msg.Angular.Z = angular_velocity;
myTurtle.send(kob_velocity_pub, velocity_msg);
end
end
```

Para acabar la aplicación, paramos el conjunto robótico:

```
% Stopping the turtle
velocity_msg.Linear.X = 0;
velocity_msg.Angular.Z = 0;
myTurtle.send(kob_velocity_pub, velocity_msg);
myTurtle.pause(1);
```

En las siguientes páginas se muestran tres figuras (figura 5.7, figura 5.8 y figura 5.9) donde se pueden observar distintos puntos de la ejecución de la aplicación.

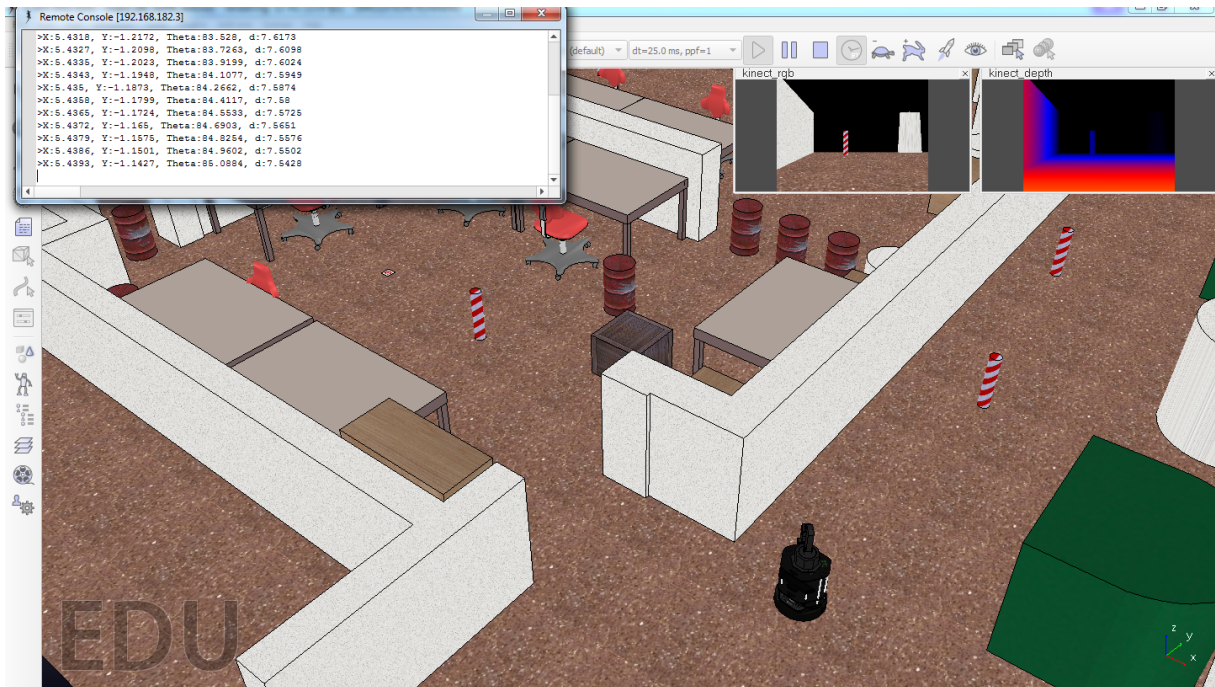


Figura 5.7: Vista del conjunto robótico justo después de haber girado una esquina.

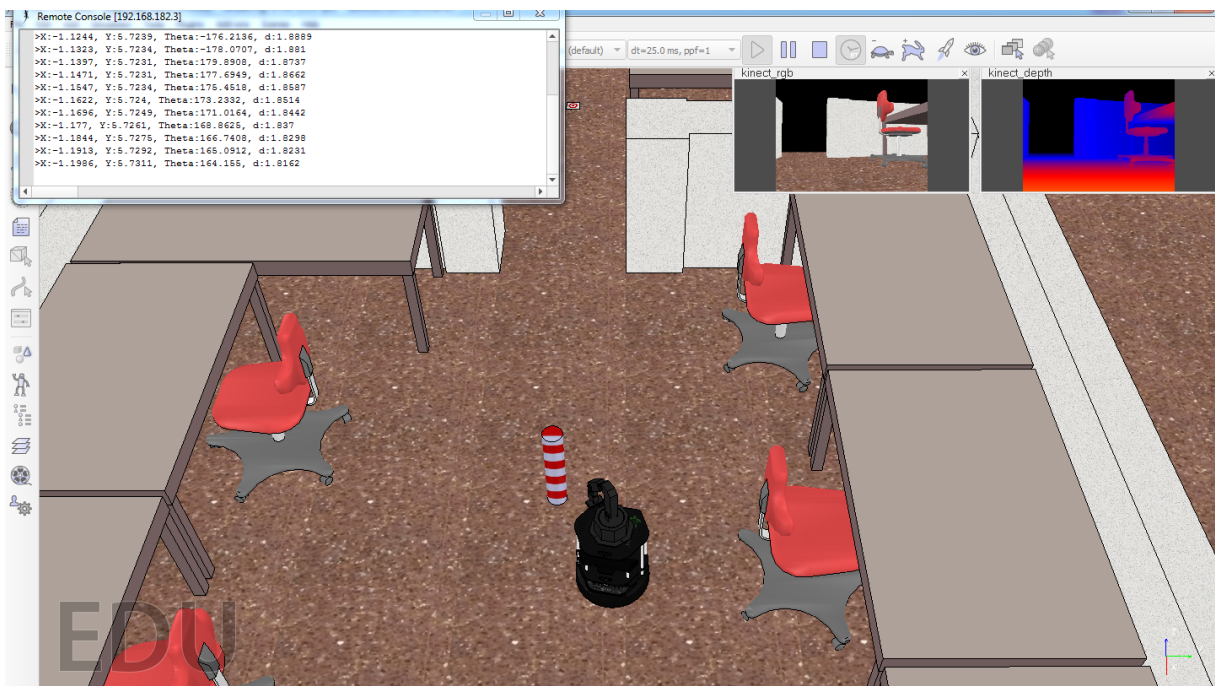


Figura 5.8: Vista del conjunto robótico sorteando un obstáculo.

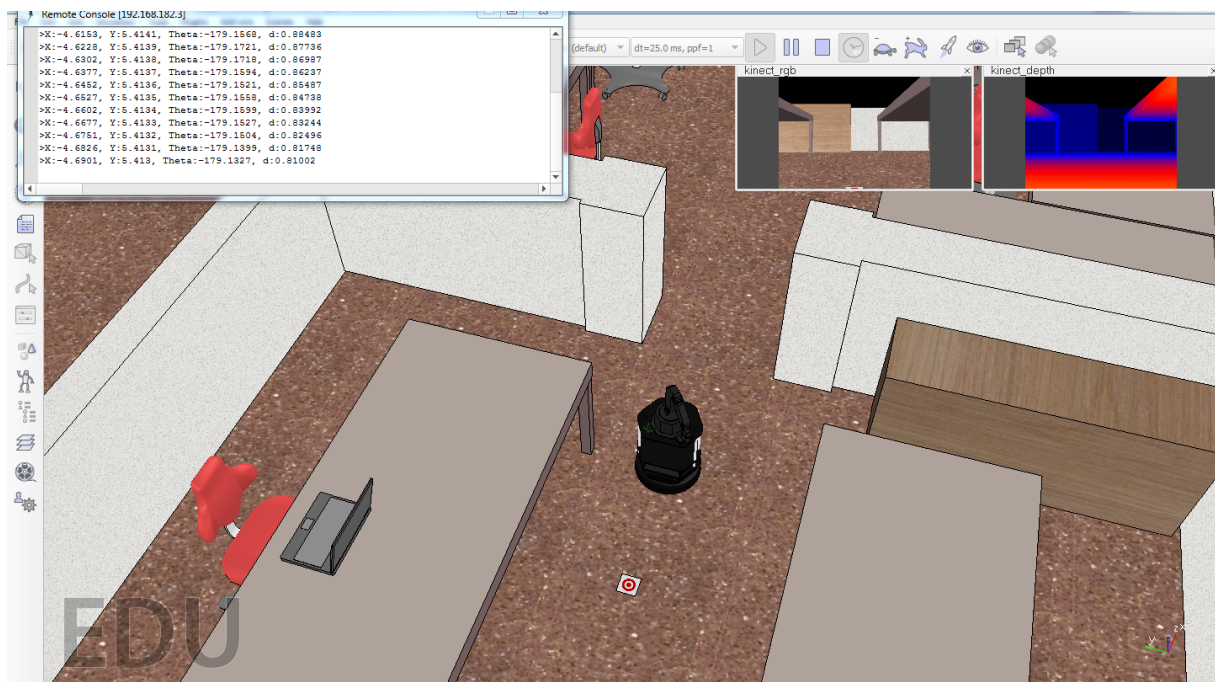


Figura 5.9: Vista del conjunto robótico a punto de llegar a la meta.

Capítulo 6

Conclusiones y trabajos futuros

Llegados a este punto, ya tenemos terminado nuestro Trabajo de Fin de Grado. El camino ha sido largo, pero el resultado ha merecido la pena. Tenemos modelado nuestro conjunto robótico en V-REP de una manera bastante fiel al robot real. Además, hemos implementado con éxito la *toolbox* en MATLAB que nos permite programarlo.

Sin lugar a dudas, esta puede ser una herramienta de gran utilidad tanto con fines educativos como para investigación. MATLAB nos brinda una serie de herramientas muy sofisticadas y potentes (el conjunto de funciones que nos proporciona), y eso hace que se puedan crear programas complejos en un tiempo bastante corto. También tenemos modelado el laboratorio 2.1.5 del Departamento, por lo que puede resultar muy interesante de cara a mostrar ejemplos a los alumnos de robótica.

Por otro lado, tenemos la amplia comunidad de usuarios de MATLAB, por lo que es raro encontrarse con algún problema que no haya planteado alguien antes y su correspondiente solución.

Lo bueno es que podemos usar MATLAB para crear pruebas o experimentos rápidos, y una vez que tengamos los resultados que esperamos, implementamos nuestro programa en ROS directamente. Como estamos usando la *Robotics System Toolbox*, no es muy complicado migrar el código de un lenguaje a otro.

No obstante, no tenemos por qué hacer eso, ya que, como hemos visto en el capítulo de resultados, los tiempos de respuesta son bastante razonables para la mayoría de las aplicaciones. Podemos usar el programa MATLAB primero sobre el simulador, para hacer las pruebas, y a continuación conectamos con el robot real para llevar a cabo su ejecución en un entorno real.

Gracias a la potencia de los motores físicos de V-REP, y a que hemos modelado nuestro conjunto robótico intentando ajustarnos lo máximo al real, es de esperar que los resultados obtenidos en simulación sean bastante parecidos a los que obtengamos cuando usemos el robot real.

Por todo esto, creo que el resultado del Trabajo de Fin de Grado ha sido muy satisfactorio y se han cumplido los objetivos planteados. Con toda seguridad, será de gran

utilidad en el desarrollo de aplicaciones robóticas con Turtlebot y WidowX.

Además, este Trabajo de Fin de Grado da lugar a numerosos trabajos futuros. Por un lado, tenemos la ampliación del mismo. Esto puede hacerse bien ampliando el modelo o bien añadiendo un modelo nuevo a V-REP.

El hecho de haber dividido el código en tantas funciones como tareas han hecho falta implementar, hacen el código muy mantenible y ampliable. Simplemente hay que seguir la estructura definida y añadir las funciones que necesitemos.

Para ayudar tanto al usuario como al desarrollador, terminamos esta memoria con dos apéndices. Uno de ellos es un manual de usuario, donde se explica cómo usar el modelo y la *toolbox*, y el otro es un manual de desarrollador, donde se dan unas pautas a seguir para ampliar el trabajo.

Apéndice A

Manual del Usuario

A.1. Requisitos

En este apéndice vamos a explicar cómo usar nuestra *toolbox* y nuestro modelo de V-REP desde el punto de vista de un usuario. Lo primero que tenemos que hacer es asegurarnos de que tenemos disponible todo lo necesario:

- Carpeta con ficheros de la *toolbox*
- V-REP en su versión 3.3.0 o posterior
- MATLAB con la *Robotics System Toolbox* instalada

Evidentemente, vamos a necesitar al menos un ordenador para ejecutar todo esto. Los sistemas operativos soportados son Windows, Linux y MacOS. No es necesario realizar ningún ajuste en función del sistema operativo utilizado, ya que la propia *toolbox* es capaz de detectar el sistema operativo y cargar la librería correspondiente para establecer la comunicación.

En cuanto al *hardware* del ordenador empleado, hemos de tener en cuenta que las simulaciones pueden tener un coste computacional bastante alto. Esto se pone especialmente de manifiesto cuando usemos sensores de visión (por ejemplo, *Kinect*). En este caso, se recomienda disponer de un ordenador bastante potente. Una tarjeta gráfica dedicada ayudará bastante. También es posible separar la simulación del programa MATLAB. Para ello, podemos ejecutar MATLAB en un ordenador que se encuentre en la misma red que el que ejecuta V-REP. Más adelante veremos cómo configurar estos parámetros.

Si queremos trabajar de forma remota, va a ser necesario disponer de una infraestructura de red. Esto va a resultar imprescindible para trabajar con el robot real, a no ser que ejecutemos MATLAB en el *netbook* del robot real¹. Esta infraestructura puede ser una simple red *Ad-hoc* (un portátil con MATLAB conectado al *netbook* del robot real mediante *Wi-fi*). Todo va a depender de las necesidades y de los recursos disponibles.

¹Cosa que no es muy recomendable.

A.2. Posibilidades de nuestra *toolbox*

Antes de ejecutar nuestro primer programa en el entorno de simulación, vamos a echar un vistazo a qué cosas podemos hacer. En primer lugar, nuestra *toolbox* está desarrollada de forma que su apariencia sea bastante similar a la *Robotics System Toolbox* que nos proporciona MATLAB.

Esto supone dos ventajas. La primera de ellas es que si el usuario está acostumbrado a trabajar con la citada *toolbox* de MATLAB, la adaptación a nuestra *toolbox* desarrollada es casi inmediata. Por otro lado, tenemos la ventaja de que el mismo programa desarrollado para la simulación servirá para ser ejecutado en el robot real. No es necesario implementar programas separados. Lo único que vamos a tener que hacer es especificar el modo que deseamos al comienzo del programa, de acuerdo a la tabla A.1.

Modo	Descripción
Simulación (<i>vrep</i>)	Modo de simulación en V-REP. En este modo, se emulan los <i>topics</i> del robot real, trabajando con el robot simulado.
Real (<i>real</i>)	Modo real. Se trabaja con el robot real (es decir, con ROS). También es posible trabajar con un entorno ROS que simule el robot real (por ejemplo, <i>Gazebo</i>), siendo preciso que los <i>topics</i> y sus correspondientes mensajes coincidan.
Simulación + Real (<i>mixed</i>)	Modo mixto. Permite trabajar a la vez tanto con el robot simulado como con el robot real.

Cuadro A.1: Modos de operación de la *toolbox*.

Hay que prestar atención al usar el modo mixto. Para que el uso de este modo tenga sentido, es muy importante que el ordenador donde se está ejecutando V-REP tenga capacidad suficiente como para que la simulación corra a velocidad de tiempo real. Si no es así, la simulación va a ir más lenta que el robot real y no vamos a poder sacar conclusiones fehacientes de que nuestro programa funciona igual de bien en el robot simulado que en el real.

Como no podría ser de otra manera, nuestra *toolbox* emula los *topics* y servicios del robot real. Puesto que el robot real funciona sobre ROS, sería interesante que el usuario tuviera ciertos conocimientos de dicho *software* robótico. Al menos, debe tener claros los conceptos de *topic* y *servicio*, y qué funcionalidades nos proporcionan cada uno de ellos.

El listado de *topics* del conjunto robótico y a qué componente pertenecen puede verse en la tabla A.2.

Obsérvese que aparecen unos cuantos *topics* que tienen un asterisco (*). Estos *topics* solo están disponibles para el modo *vrep*, ya que modelan ciertas funciones que solo el simulador puede prestar.

Topic	Sentido	Componente
/arm_1_joint/command	MATLAB → Robot	WidowX
/arm_2_joint/command	MATLAB → Robot	WidowX
/arm_3_joint/command	MATLAB → Robot	WidowX
/arm_4_joint/command	MATLAB → Robot	WidowX
/arm_5_joint/command	MATLAB → Robot	WidowX
/camera/depth/image	Robot → MATLAB	Kinect
/camera/depth/image/compressed	Robot → MATLAB	Kinect
/camera/rgb/image_color	Robot → MATLAB	Kinect
/camera/rgb/image_color/compressed	Robot → MATLAB	Kinect
/camera/rgb/image_mono	Robot → MATLAB	Kinect
/camera/rgb/image_mono/compressed	Robot → MATLAB	Kinect
/gripper_1_joint/command	MATLAB → Robot	WidowX
/mobile_base/commands/led1	MATLAB → Robot	Kobuki
/mobile_base/commands/led2	MATLAB → Robot	Kobuki
/mobile_base/commands/motor_power	MATLAB → Robot	Kobuki
/mobile_base/commands/reset_odometry	MATLAB → Robot	Kobuki
/mobile_base/commands/velocity	MATLAB → Robot	Kobuki
/mobile_base/controller_info	Robot → MATLAB	Kobuki
/mobile_base/events/bumper	Robot → MATLAB	Kobuki
/mobile_base/events/button	Robot → MATLAB	Kobuki
/mobile_base/events/cliff	Robot → MATLAB	Kobuki
/mobile_base/events/wheel_drop	Robot → MATLAB	Kobuki
/mobile_base/sensors/core	Robot → MATLAB	Kobuki
/mobile_base/sensors/imu_data	Robot → MATLAB	Kobuki
/mobile_base/sensors/imu_data_raw	Robot → MATLAB	Kobuki
/mobile_base/version_info	Robot → MATLAB	Kobuki
/odom	Robot → MATLAB	Kobuki
/simulation/pose*	V-REP → MATLAB	V-REP
/simulation/sim_time*	V-REP → MATLAB	V-REP
/scan	Robot → MATLAB	Hokuyo
/vrep/aux_console/clear*	MATLAB → V-REP	V-REP
/vrep/aux_console/create*	MATLAB → V-REP	V-REP
/vrep/aux_console/delete*	MATLAB → V-REP	V-REP
/vrep/aux_console/hide*	MATLAB → V-REP	V-REP
/vrep/aux_console/show*	MATLAB → V-REP	V-REP
/vrep/aux_console/print*	MATLAB → V-REP	V-REP
/vrep/last_cmd_time*	V-REP → MATLAB	V-REP
/vrep/status_bar_message*	MATLAB → V-REP	V-REP

Cuadro A.2: Listado de *topics* implementados.

Con respecto a las funciones que podemos emplear como usuarios, nos encontramos con las de la tabla A.3. Si el lector ya ha usado la *Robotics System Toolbox* de MATLAB, se dará cuenta de que los nombres de las coinciden, por lo que la curva de aprendizaje será totalmente plana, en ese caso. Aparte de esas funciones, existen otras que son específicas para V-REP, y que nos proporcionan cuestiones relativas a la *toolbox*, entre otras cosas.

Todas las funciones tienen añadidas los comentarios de ayuda correspondientes. En ellos se detallan los parámetros de entrada y de salida en función del modo de funcionamiento de la *toolbox* seleccionado. Simplemente escribimos en la consola de MATLAB 'help función' y nos aparecerá dicha información².

Nombre	Descripción
rosinit	Inicia la conexión a un ROS <i>master</i> . En los modos <i>vrep</i> y <i>mixed</i> , inicia la conexión con V-REP.
rostopic	Proporciona información sobre los <i>topics</i> . Están disponibles las opciones ' <i>list</i> ', que devuelve un listado con los <i>topics</i> disponibles, y ' <i>type</i> ', que devuelve el tipo de mensaje para un <i>topic</i> dado.
rosservice	Proporciona información sobre los servicios. Está disponible la opción ' <i>list</i> ', que devuelve los servicios disponibles.
rosshutdown	Cierra la conexión con ROS. En los modos <i>vrep</i> y <i>mixed</i> , cierra la conexión con V-REP.
rospublisher	Crea un <i>publisher</i> para un <i>topic</i> dado.
rossubscriber	Crea un <i>subscriber</i> para un <i>topic</i> dado.
rossvcclient	Crea un cliente para un servicio.
rosmessage	Crea un mensaje para un <i>topic</i> o servicio dado.
send	Envía un mensaje a través de un <i>topic</i> .
receive	Recibe un mensaje a través de un <i>topic</i> .
call	Llama a un servicio.
getPingTme	Devuelve el tiempo de respuesta entre MATLAB y V-REP. Este tiempo de respuesta incluye la emisión de un comando, el tiempo de ejecución en V-REP y la recepción de la respuesta.
pause	Detiene la ejecución del programa durante un tiempo determinado (en segundos). En los casos de modo <i>vrep</i> y <i>mixed</i> , este tiempo se toma de la simulación en V-REP. De esta manera, se garantiza que el tiempo de parada del programa va a ser siempre el mismo independientemente de las capacidades computacionales del ordenador que estemos usando.
isVREPEEnabled	Devuelve si la comunicación con V-REP está habilitada o no.
isROSEnabled	Devuelve si la comunicación con ROS está habilitada o no.

Cuadro A.3: Listado de funciones implementadas.

Aparte de estas funciones, existe un pequeño conjunto de herramientas que tienen por

²Para ello es necesario que la *toolbox* se encuentre en el *path* de MATLAB, tal y como veremos más adelante.

objetivo hacer los programas más legibles y facilitar la realización de tareas repetitivas. Ejemplo de ello va a ser el obtener la pose y la orientación del robot a partir del mensaje recibido a través del *topic* `/odom`. Estas funciones se encuentran bajo la carpeta `'tools'` de nuestra *toolbox*, tal y como veremos más adelante. La idea es que el usuario añada sus propias funciones a esa carpeta, según sus necesidades. No obstante, la carpeta incluye alguna que otra función, tal y como muestra la tabla

Nombre	Descripción
<code>drawKinectImage</code>	Dibuja la imagen de Kinect, tanto la parte RGB como la parte de profundidad, usando la función <i>surf</i> de MATLAB.
<code>getConsoleMsgAndDispInMatlab</code>	Devuelve el mensaje listo para ser enviado a la consola auxiliar de V-REP y lo muestra en la consola de MATLAB. Este mensaje incluye el tiempo de simulación en el momento de su generación.
<code>getKobukiPoseAndOrientation</code>	Devuelve la <i>pose</i> de la base Kobuki a partir del mensaje recibido en el <i>topic</i> <code>"/odom"</code> .
<code>thetaTo360</code>	Realiza la conversión entre un ángulo perteneciente al rango <code>[-180, 180)</code> al rango <code>[0, 360)</code> .

Cuadro A.4: Listado de funciones auxiliares implementadas.

A.3. Preparando el entorno

Vamos a preparar el entorno para poder ejecutar nuestro primer programa. Partimos de la base de que tenemos instalado MATLAB con la *Robotics System Toolbox* y V-REP.

Bien, pues lo primero que podemos hacer es abrir V-REP. En mi caso, si hago clic en el menú `'Help → About'`, aparece la versión del programa que he utilizado, tal y como muestra la figura A.1.

Una vez hecho eso, estamos en disposición de cargar el modelo de nuestro conjunto robótico. Para ello, debemos disponer del fichero que lo contiene. Este fichero se denomina `'Turtlebot2_WidowX.ttm'`, y se proporciona junto a la *toolbox*. Para abrir el modelo, hacemos clic en el menú `'File → Load Model...'`, seleccionando el fichero en cuestión. Abrimos y obtendremos algo parecido a la figura A.2.

Existe otra opción para cargar el modelo, que consiste en añadir el mismo a la librería de modelos que tiene V-REP. Para ello, es necesario acceder al *path* de instalación de dicho programa, que dependerá de dónde lo habremos instalado y del sistema operativo empleado. Una vez allí, accedemos a `'models/robots/mobile'` y pegamos ahí el fichero del modelo. De este modo, lo tendremos disponible en la librería de V-REP, y solo habrá que arrastrarlo a la escena.

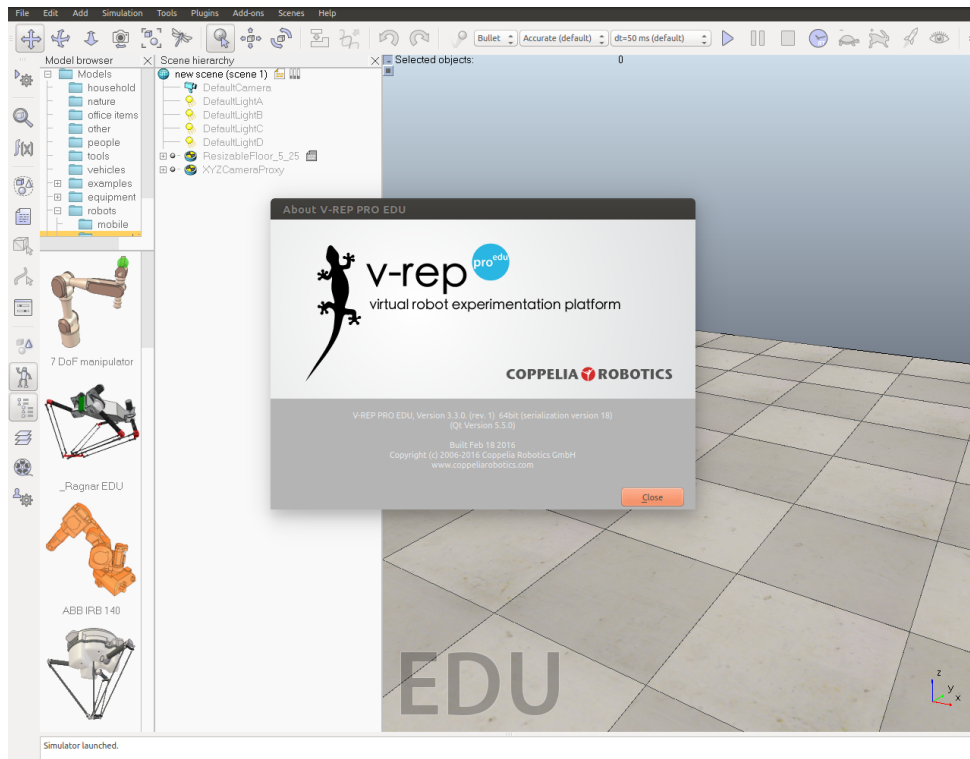


Figura A.1: Abriendo V-REP por primera vez.

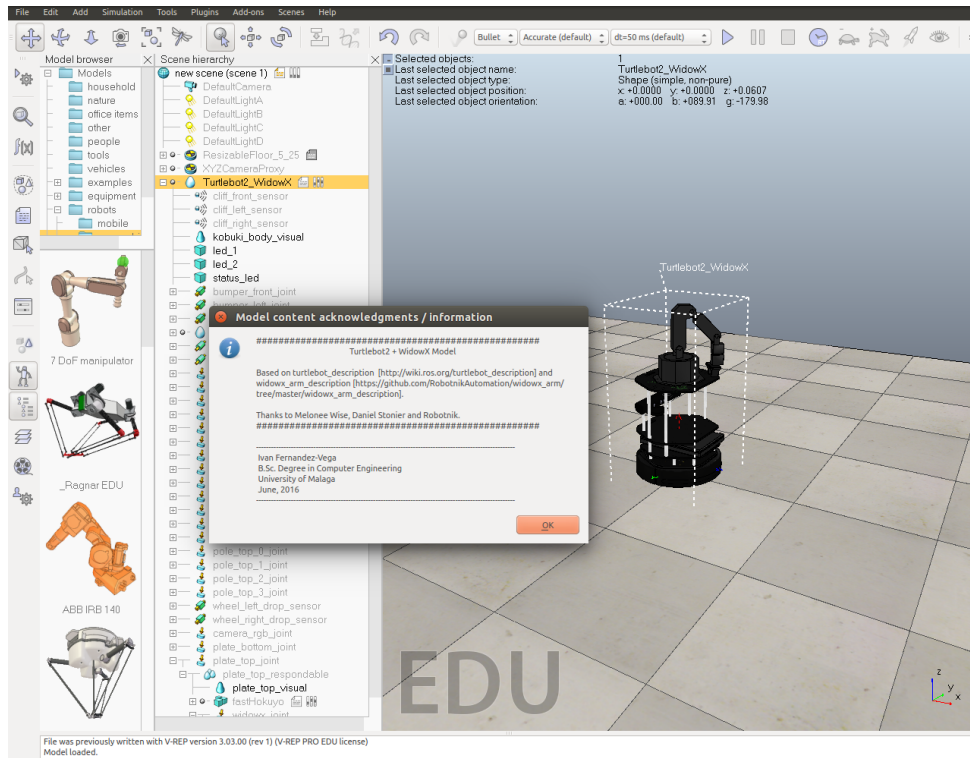


Figura A.2: Cargando el modelo en una escena de V-REP.

Si queremos añadir más elementos a la escena, este puede ser un buen momento. Podemos emplear cualquiera de los múltiples elementos que nos proporciona V-REP, donde encontraremos desde mesas y sillas hasta paredes y ventanas. Si no está disponible el elemento que necesitamos, siempre podemos crearlo a partir de otros o a partir de formas básicas. El límite lo pone la imaginación. Si además jugamos con texturas, los resultados pueden ser bastante sorprendentes (figura A.3).

No obstante, el hecho de añadir gran cantidad de elementos a una escena hace que la simulación se enlentezca, sobre todo si usamos sensores de visión. Más adelante abordaremos el tema del rendimiento con mayor profundidad.

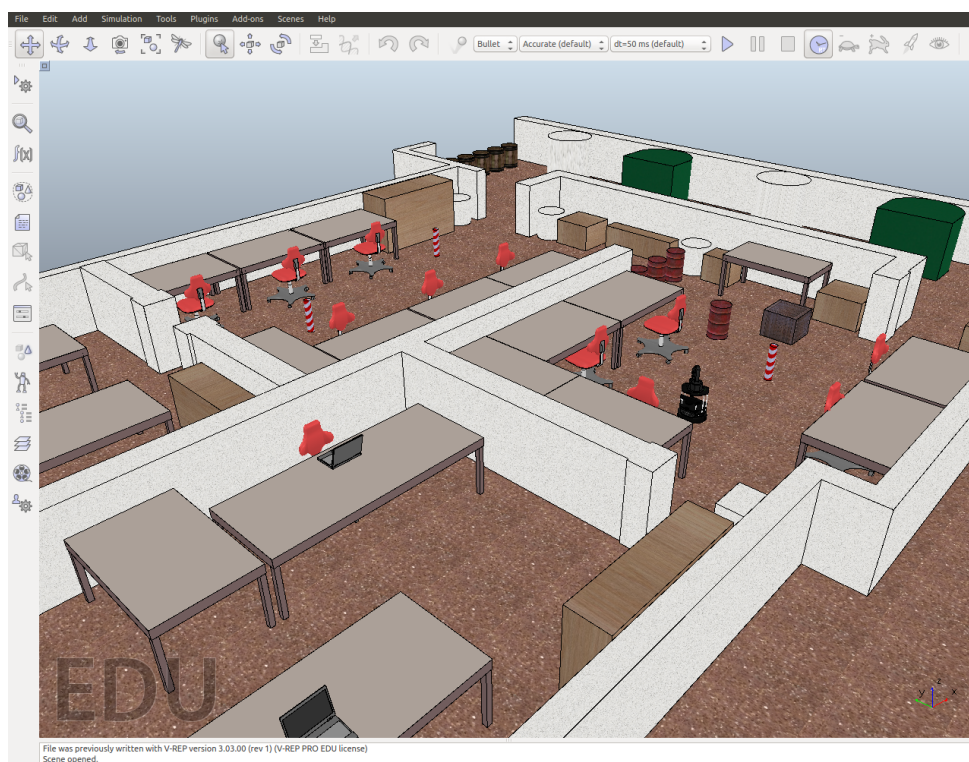


Figura A.3: Ejemplo de escena medianamente compleja.

El siguiente paso es abrir MATLAB. Una vez hecho eso, tenemos que establecer el *path* de MATLAB en la carpeta raíz de la *toolbox*. Esto es muy importante, ya que de otra manera, MATLAB no encontrará las funciones y el programa que desarrollemos no podrá funcionar (al menos correctamente). También existe la posibilidad de añadir los ficheros de la *toolbox* al *path*, de forma que podamos crear nuestros programas en cualquier sitio y sea MATLAB quien se encargue de buscar las funciones que necesita. Esto lo dejamos para usuarios más avanzados.

El resultado debería ser parecido a la figura A.4. Dentro del *path* tenemos los elementos necesarios para que la comunicación con V-REP sea satisfactoria.

Por un lado, tenemos la carpeta '@VREP'. Esta carpeta contiene las funciones de la clase VREP, donde se incluyen tanto las funciones accesibles para el usuario (tabla A.3)

como las funciones internas y privadas. Estas últimas no son accesibles para el usuario directamente, ya que son las encargadas de emular los *topics* y llamar a las funciones que proporcionan la comunicación con V-REP.

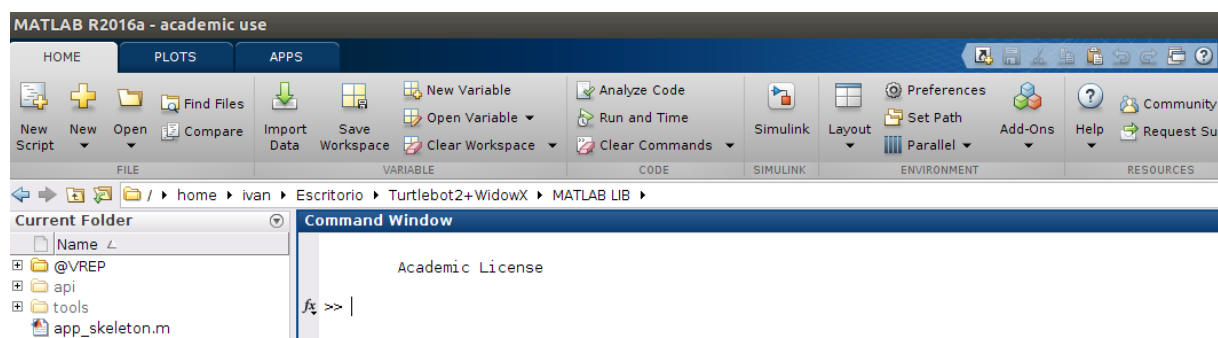


Figura A.4: MATLAB con el *path* establecido.

Por otro lado, tenemos la carpeta 'api'. Esta carpeta contiene los ficheros necesarios para comunicar MATLAB con V-REP. Esta carpeta no tiene que (ni debe) ser modificada por el usuario. En caso de modificación accidental, lo mejor sería restaurarla desde la versión original.

En lo que a carpetas se refiere, solo nos queda 'tools'. Contiene herramientas para el usuario. Si bien ya trae de por sí algunas (tabla A.4), el usuario está invitado a añadir las suyas propias con el objetivo de hacer los programas más elegantes o realizar operaciones complejas en código y repetitivas.

A.4. Una primera aplicación

Para que el usuario no tenga que partir de cero en su primera aplicación, la *toolbox* incluye un programa de ejemplo, denominado 'app_skeleton.m'. Este programa establece una comunicación con V-REP (o con ROS, según el modo que especifiquemos) y hace que el conjunto robótico avance en línea recta durante tres segundos y luego se pare.

Quizá sea interesante analizar el código e ir explicando sobre la marcha el uso de algunas funciones. Lo primero que nos encontramos en el código es la llamada a la función *addpath* de MATLAB, en dos ocasiones. La primera de ellas añade al *path* la carpeta 'api', imprescindible para poder conectarse a V-REP. La segunda llamada es opcional, no es necesaria si no pretendemos usar las funciones que se encuentran bajo la carpeta *tools*:

```
addpath('api');
addpath('tools');
```

A continuación, nos encontramos los parámetros del programa. Estos parámetros incluyen las direcciones IP (la de V-REP y la del robot real, en su caso) y los puertos correspondientes. Puesto que en este caso vamos a usar V-REP en la misma máquina que MATLAB, ponemos la dirección IP igual a '127.0.0.1'.

Por otro lado, tengo por costumbre definir dos valores de referencia para la velocidad de la base *Kobuki*. Uno de ellos es la velocidad lineal y otro la velocidad angular, ambos dentro de los límites de movimiento de la base.

```
v_ip_addr = '127.0.0.1';
v_port = 19999;
r_ip_addr = '10.42.0.17';
r_port = 11311;
defaultLinearVelocity = 0.2;
defaultAngularVelocity = pi / 2;
```

Llegados a este punto, podemos crear la instancia de un objeto de la clase 'VREP'. Este objeto es el que contendrá las funciones a las que podemos acceder y mantendrá la comunicación con V-REP. Para ello, simplemente llamamos al constructor de la clase, estableciendo el modo de operación que deseamos en el primer argumento de la llamada (tabla A.1). También podemos llamar a la función sin argumentos, en cuyo caso el modo de operación se establecerá por defecto en *vrep*.

Existe un segundo parámetro opcional en la llamada al constructor, que sirve para establecer si queremos o no que se modele el ruido en la unidad de medida inercial (la *IMU*, por sus siglas en Inglés). Si no pasamos ese tercer argumento, el ruido estará habilitado por defecto. Las dos posibilidades son '*noise*'→Ruido habilitado o '*noiseless*'→Ruido deshabilitado. En este programa vamos a dejarlo habilitado:

```
myTurtle = VREP('vrep');
```

Ahora ya estamos en condiciones de iniciar la comunicación con V-REP (en este caso lo hemos configurado así, pero el procedimiento para hacerlo con ROS sería idéntico). Para ello, llamamos a la función *rosinit* de nuestra *toolbox*, con las direcciones IP y los puertos que definimos al comienzo el programa. Es importante darse cuenta de que llamamos a la función a través del objeto *myTurtle* que acabamos de crear. Si no lo hacemos así, estaríamos llamando a la función que tiene el mismo nombre en la Robotics System Toolbox cosa que no es lo que pretendemos:

```
myTurtle.rosinit(v_ip_addr, v_port, r_ip_addr, r_port);
```

A continuación, podemos crear los publishers para nuestro programa. La elección de cada uno de ellos va a depender de lo que nuestro programa necesite. La lista de *topics* implementados se encuentra en la tabla A.2. En este programa, vamos a emplear el *topic* para enviar comandos de velocidad a *Kobuki* y los necesarios para mostrar mensajes por la barra de estado de V-REP y manejar la consola auxiliar:

```
kob_velocity_pub = myTurtle.rospublisher('/mobile_base/commands/velocity');
vrep_st_bar_pub = myTurtle.rospublisher('/vrep/status_bar_message');
vrep_console_open_pub = myTurtle.rospublisher('/vrep/aux_console/create');
vrep_console_print_pub = myTurtle.rospublisher('/vrep/aux_console/print');
```

Hacemos lo propio para los *subscribers*. En nuestro programa, vamos a emplear el *topic* de la odometría para mostrar por consola la pose de nuestro robot, y el *topic* propio de V-REP que devuelve el tiempo de simulación para mostrarla junto a la pose:

```
odom_sub      = myTurtle.rossubscriber('/odom');
sim_time_sub  = myTurtle.rossubscriber('/simulation/sim_time');
```

Otra cuestión que vamos a hacer en el apartado de inicialización es crear los dos mensajes que vamos a usar para los *topics* donde publicaremos (el de la velocidad de *Kobuki*, el de la barra de estado y el de la consola auxiliar). Usamos la función *rosmmessage*, pasándole como argumento los dos *publishers*:

```
velocity_msg  = myTurtle.rosmmessage(kob_velocity_pub);
status_bar_msg = myTurtle.rosmmessage(vrep_st_bar_pub);
console_open_msg = myTurtle.rosmmessage(vrep_console_open_pub);
console_print_msg = myTurtle.rosmmessage(vrep_console_print_pub);
```

Por cortesía, vamos a mostrar en la barra de estado de V-REP un mensaje con el programa que estamos ejecutando. Esto podemos hacerlo tal y como aparece en el siguiente recuadro. En primer lugar, establecemos la carga del mensaje (su campo *Data*) en el texto que queremos que se muestre. Seguidamente, publicamos ese mensaje en el *topic* correspondiente usando la función *send* que nos proporciona la *toolbox*:

```
status_bar_msg.Data = 'Program: MY PROGRAM';
myTurtle.send(vrep_st_bar_pub, status_bar_msg);
```

Cuando ejecutemos el programa, veremos cómo el mensaje efectivamente se muestra en la barra de estado, tal y como muestra la figura A.5)

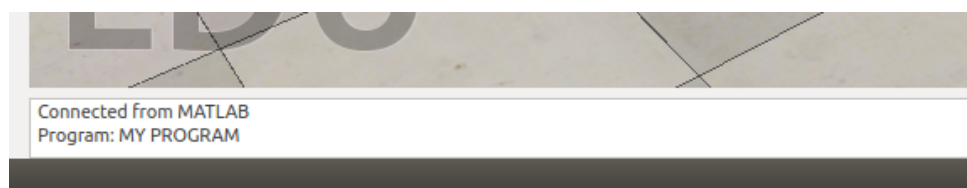


Figura A.5: Mensaje en la barra de estado de V-REP.

Seguidamente, podemos inicializar la consola auxiliar que tendremos disponible en V-REP a nuestra disposición para mostrar los datos que estimemos oportunos. Para ello, publicamos el mensaje correspondiente a la creación de la misma en el *topic* que tiene tal función:

```
myTurtle.send(vrep_console_open_pub, console_open_msg);
```

Hasta aquí llega la inicialización de nuestro programa. Vamos a notificar tanto por la consola de MATLAB como por la auxiliar de V-REP de este hecho. En MATLAB lo hacemos simplemente con una llamada a *disp*, mientras que para hacerlo en la consola de V-REP hemos de establecer la carga del mensaje que queremos y publicarlo en el *topic* correspondiente.

```

disp('Initialized OK. ');
console_print_msg.Data = 'Initialized OK. ';
myTurtle.send(vrep_console_print_pub, console_print_msg);

```

La siguiente parte de nuestro programa es la encargada de mover la tortuga en línea recta durante tres segundos. De nuevo notificamos al usuario del programa, de forma muy similar a como lo hicimos en el cuadro anterior:

```

disp('Moving the turtle for 3 seconds... ');
console_print_msg.Data = strcat('Moving the turtle for 3 seconds... ');
myTurtle.send(vrep_console_print_pub, console_print_msg);

```

Hemos optado por mover la tortuga durante un tiempo determinado, pero también podríamos haber usado la odometría para moverla una distancia determinada. En nuestro caso, hemos de tomar el tiempo inicial de simulación para poder contar los tres segundos que vamos a moverla, de la siguiente manera:

```

sim_time_msg = myTurtle.receive(sim_time_sub);
startingSimTime = sim_time_msg.Data;
currentSimTime = startingSimTime;

```

Ahora ya podemos comenzar nuestro bucle. Va a ser un *while*, cuya condición de finalización va a ser que la resta entre el tiempo actual y el inicial sea mayor o igual a tres segundos:

```

while(currentSimTime - startingSimTime < 3)

```

Dentro del cuerpo del bucle, lo primero que vamos a hacer es recibir un mensaje de odometría. Su contenido va a servirnos para mostrar en todo momento por la consola auxiliar de V-REP la pose del robot. Obsérvese que estamos usando la función *getKobukiPoseAndOrientation* para obtener la pose y la orientación del robot a partir del mensaje recibido. Esta función es una de las proporcionadas en la carpeta 'tools' de nuestra *toolbox*.

```

odom_msg = myTurtle.receive(odom_sub);
[x, y, theta] = getKobukiPoseAndOrientation(odom_msg);

```

Lo siguiente que hacemos es recibir el mensaje que contiene el tiempo de simulación, para así almacenarlo en la variable 'currentSimTime':

```

sim_time_msg = myTurtle.receive(sim_time_sub);
currentSimTime = sim_time_msg.Data;

```

A continuación, mostramos por la consola auxiliar de V-REP la pose y la orientación actual de la tortuga, generando el mensaje con la ayuda de *strcat* y *num2str*. Para ello, publicamos el mensaje en el *topic* habilitado para ello:

```

console_print_msg.Data = strcat('X: ', num2str(x), ', Y: ', ...
    num2str(y), ', Theta: ', num2str(theta));
myTurtle.send(vrep_console_print_pub, console_print_msg);

```

Dentro del cuerpo del bucle, solo nos queda mandar el comando de velocidad a *Kobuki*. Ponemos la velocidad lineal en su valor por defecto (el que definimos al comienzo del programa) y la velocidad angular en 0.

Es importante que enviemos este comando de velocidad en cada iteración del bucle, puesto que de acuerdo a las especificaciones de *Kobuki*, si no mandamos un comando de velocidad después de 0,6 segundos, el robot detendrá su movimiento.

```
velocity_msg.Linear.X = defaultLinearVelocity;  
velocity_msg.Angular.Z = 0;  
myTurtle.send(kob_velocity_pub, velocity_msg);  
end
```

Si llegamos a este punto es porque los tres segundos ya han pasado. Recordemos que estos tres segundos no tienen por qué corresponderse con tres segundos “reales”, si no que siempre estarán supeditados al tiempo de simulación (que dependerá, a su vez, de la potencia de la máquina que estemos usando).

Lo que toca ahora es parar la tortuga. Si bien podríamos esperar a que pasaran esos 0,60 segundos que citamos antes y que la tortuga se parase automáticamente, vamos a tratar de ser un poco más civilizados y vamos a enviar un comando de velocidad 0, tal y como muestran las siguientes líneas. Nótese que antes de enviar el comando, notificamos al usuario de tal acontecimiento:

```
disp('Stopping the turtle...');  
console_print_msg.Data = strcat('Stopping the turtle...');  
myTurtle.send(vrep_console_print_pub, console_print_msg);  
  
velocity_msg.Linear.X = 0;  
velocity_msg.Angular.Z = 0;  
myTurtle.send(kob_velocity_pub, velocity_msg);
```

Antes de acabar el programa, podemos esperar un segundo a que la tortuga termine de pararse. Si bien podríamos terminar directamente, esto ayudará a que la desconexión de V-REP no sea tan abrupta.

Para ello, llamamos a *pause*, pero lo hacemos a través de nuestro objeto *myTurtle*. De este modo, estamos garantizando que ese segundo será un segundo en el tiempo de simulación, y no en el tiempo “real”.

```
myTurtle.pause(1);
```

Acabamos nuestro programa llamando a *rosshutdown*, que se encargará de cerrar la conexión y llevar a cabo las tareas pertinentes para que el programa finalice adecuadamente.

```
myTurtle.rosshutdown;
```

¡Ya tenemos listo nuestro primer programa! En la siguiente sección, abordaremos la simulación del mismo.

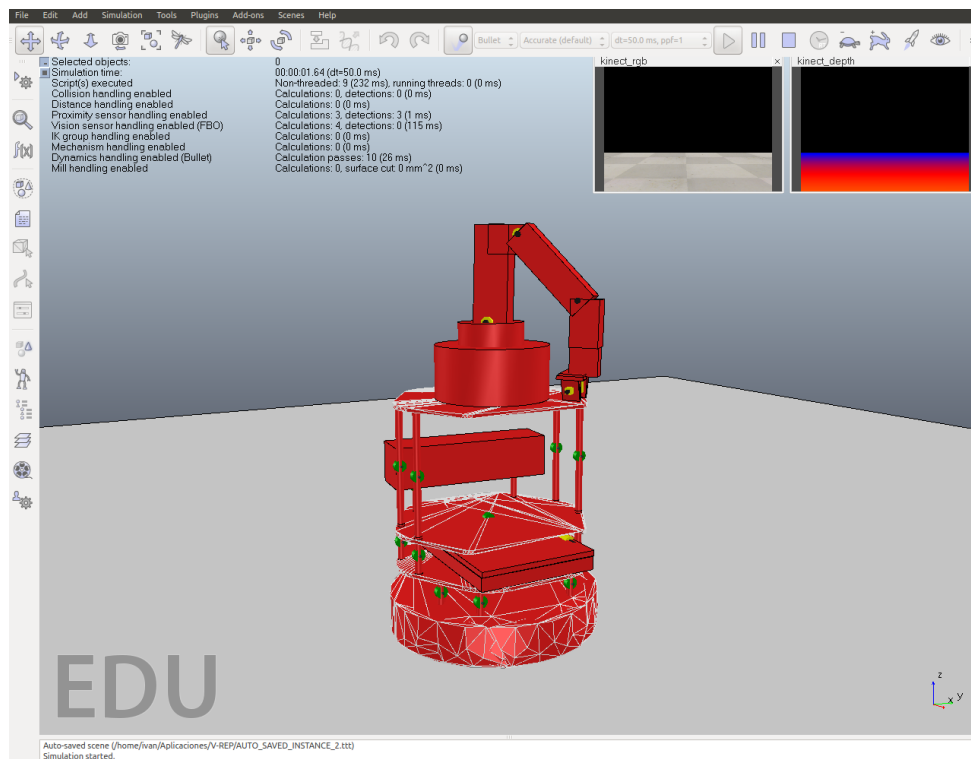


Figura A.7: Contenido dinámico de la escena.

Si seguimos observando la barra de herramientas de la figura A.6, nos encontramos con otro menú desplegable que permite establecer la **precisión** del motor físico. Evidentemente, mayor precisión implica mayor carga computacional. La elección del valor para este campo va a depender bastante de la potencia de la máquina sobre la que ejecutemos V-REP. En cualquier caso, en el valor por defecto ('Accurate'), los resultados son bastante aceptables.

Por otro lado, tenemos un desplegable que nos permite establecer el diferencial de tiempo para la simulación. Dicho en pocas palabras, este parámetro especifica cada cuanto se ejecutan los *scripts* de la escena. También determina la velocidad a la que se ejecuta el motor físico.

Así pues, este parámetro puede provocar cambios notables en la simulación. Además, de este parámetro también va a depender el tiempo de respuesta de las llamadas desde MATLAB. El diferencial de tiempo por defecto (50 milisegundos) hace que la frecuencia de los cálculos sea de 20 Hertzios. Es un valor aceptable, pero si disponemos de una máquina potente podremos seleccionar valores más pequeños, como 25 ó 10 milisegundos. Esto hará que la simulación sea más fluida si cabe.

El siguiente botón que nos encontramos es el del reloj. Cuando este botón está pulsado, el simulador tratará de que la simulación corra en tiempo real. Esto solo será posible si la máquina sobre la que se está ejecutando tiene potencia suficiente. En otro caso, irá al máximo posible.

Personalmente, siempre suelo dejar este botón pulsado. De esta manera, si una simulación consta de elementos simples, no irá más rápido de lo normal, sino que irá a tiempo real.

A continuación, nos encontramos los tres típicos botones de *Start*, *Pause* y *Stop*, cuyo cometido no creo que necesite explicación. A la derecha de estos, se encuentran una tortuga, una liebre y un cohete.

Pulsar sobre la tortuga hace que la simulación se enlentezca. Por contra, tal y como es de esperar, pulsar sobre la liebre hace que la simulación vaya más rápido. También tenemos el cohete, que activa la denominada *threaded rendering*. Si lo activamos, se separará la parte de renderizado³ de los cálculos de la simulación. Puede ser útil en algunos casos en los que no conseguimos que la simulación siga el ritmo que necesitamos.

Bien, pues solo nos falta pulsar el botón de *Start* de nuestro simulador, y ejecutar el programa de prueba en MATLAB. Una vez hecho eso, el resultado debe ser parecido al de la figura A.8). Obsérvese que la tortuga ha avanzado aproximadamente una baldosa, ya que la velocidad está establecida en un valor bajo (20 centímetros por segundo).

Estas baldosas son de 50x50 centímetros, así que en tres segundos la tortuga debería haber avanzado aproximadamente 60 centímetros (lo que viene siendo una baldosa y un poco más), pero hay que tener en cuenta el tiempo de arranque y parada. Si nuestro objetivo fuera recorrer esos 60 centímetros, deberíamos haber empleado los datos odométricos en vez de el tiempo de simulación.

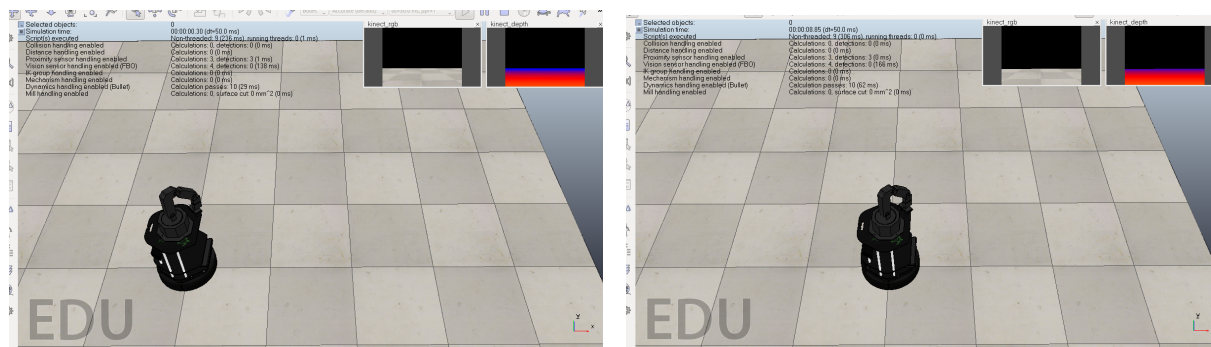


Figura A.8: Resultado de ejecución del programa de prueba.

A partir de aquí, es cuestión de que el usuario explore las posibilidades que proporciona la *toolbox* por su cuenta. Un experimento interesante sería ejecutar este mismo programa en el robot real, y comprobar que el resultado es muy similar al que hemos obtenido en el simulador.

³Básicamente, es la encargada de realizar los cálculos necesarios para formar las imágenes de los sensores de visión

A.6. Mejorando el rendimiento

Llegados a este punto, ya hemos creado y simulado nuestro primer programa. Para desarrollar programas más complejos, podemos echar un vistazo a las dos aplicaciones que han sido implementadas (ver capítulo de resultados de esta memoria).

Una de las aplicaciones utiliza el brazo para mover un objeto de un punto a otro de la escena, y la otra emplea navegación reactiva para llevar el conjunto robótico de un lugar a otro.

De cara a mejorar rendimiento de la simulación, puede ser interesante deshabilitar los sensores de visión. Evidentemente, solo podremos hacer esto cuando nuestra aplicación no haga uso de ellos. Este hecho hace que, si no se dispone de una máquina muy potente, aumente el número de *frames* por segundo considerablemente.

Para deshabilitarlos, debemos hacer doble clic en el icono de la cámara de alguno de ellos, y se abrirá la ventana que puede verse en la figura A.9. Desmarcamos la opción rodeada con subrayador amarillo y ya tendremos deshabilitados todos los sensores de visión de la escena⁴.

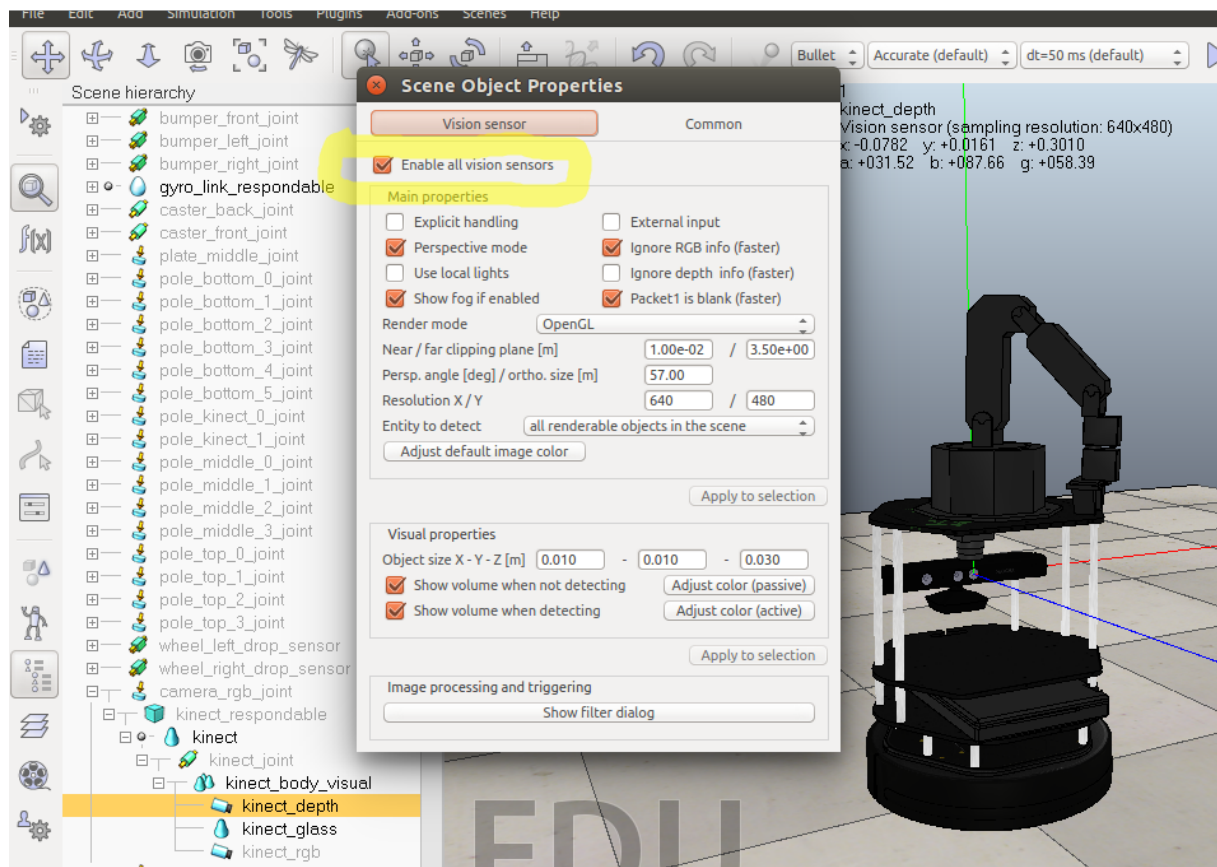


Figura A.9: Deshabilitar los sensores de visión.

⁴Ojo: el sensor láser Hokuyo está modelado con dos sensores de visión, así que si hacemos esto, también lo deshabilitaremos.

Apéndice B

Manual del Desarrollador

Este apéndice vamos a dedicarlo al manual del desarrollador. En él vamos a explicar cómo ampliar el trabajo desarrollado, de forma que se puedan añadir nuevas funcionalidades.

B.1. Añadir un sensor al modelo

Seguramente, una de las primeras formas de ampliación que se nos pueden ocurrir consiste en añadir un sensor al modelo.

Para ello, lo primero que deberíamos hacer es añadir ese sensor al robot real, para así saber su ubicación concreta en el mismo y colocarlo en el mismo lugar del robot simulado.

Seguidamente, abrimos V-REP y echamos un vistazo a su librería. Concretamente, hemos de buscar dentro de la carpeta “components”, y dentro de la misma, en “sensors”. Allí nos encontramos con algo parecido¹ a la figura B.1.

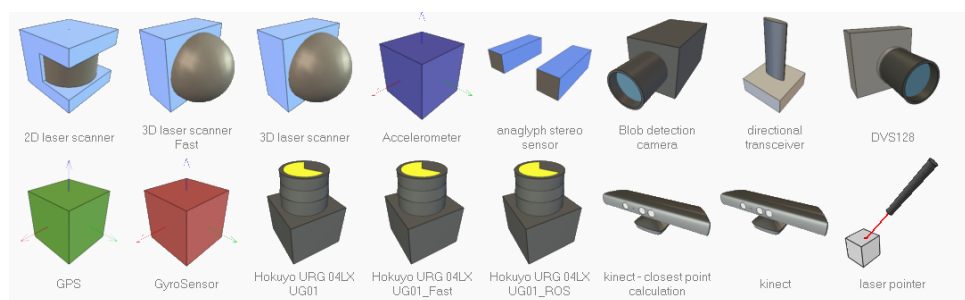


Figura B.1: Algunos sensores de la librería de V-REP

Si tenemos la suerte de que el sensor aparece en la librería, o uno muy parecido que podamos ajustar, estamos de enhorabuena. En caso contrario, tocará pensar un poco cómo modelar ese sensor con las herramientas que nos proporciona V-REP y los elementos básicos.

¹Recordemos que para la elaboración de este trabajo se ha empleado la versión 3.3.0 de V-REP

Una vez que hemos colocado el sensor en el modelo, y que tiene ajustados los parámetros pertinentes, tenemos que establecer la comunicación con MATLAB.

Para ello, lo que vamos a tener que hacer es observar cómo está implementado desde el punto de vista *software* ese sensor en el robot real. Necesitamos saber cómo acceder a los datos que nos proporciona (típicamente, se publicarán los mismos en un *topic*) y del tipo que son.

A partir de esa información, generamos el canal de comunicación V-REP → MATLAB. Una opción bastante sencilla de hacer esto es mediante el uso de una señal. Si el sensor es muy simple desde el punto de vista *software* (es decir, no requiere cálculos en Lua), podemos añadir esta señal directamente en el *script* de Turtlebot o de WidowX, según corresponda:

```
data = simPackFloats(misDatos)
simSetStringSignal(modelBaseName.. '_nameOfSensor_data', data)
```

Por contra, si el sensor requiere de cierto número de líneas de código, puede resultar interesante añadir un *script* propio al sensor, de manera que no sobrecargaremos en exceso los *scripts* principales. El sensor Hokuyo está implementado de esta forma, así que puede ser un buen punto de partida.

Hemos de prestar especial atención a qué componentes tendrá el mensaje publicado en el *topic* para así generar una señal que los incluya todos. Por ejemplo, para el caso del sensor Hokuyo, es necesario conocer el tiempo de simulación en el que fueron tomados los datos.

B.2. Añadir un *topic* en MATLAB

Una vez que tenemos nuestro sensor añadido al modelo, lo siguiente que debemos hacer es crear en MATLAB el *topic* o los *topics* necesarios para poder trabajar con el mismo como si se tratara del sensor real.

Para ello, vamos a crear una función privada a la *toolbox* que traiga los datos del sensor desde V-REP y genere el mensaje correspondiente. Un ejemplo de prototipo para la misma podría ser:

```
[message, returnCode] = getNameOfSensorMsg(robot)
```

Dentro de la función, hemos de traernos los datos desde V-REP. Una forma de hacerlo podría ser la siguiente (en este caso, el sensor devuelve un conjunto de datos encapsulados en un *string*, pero podría ser que el sensor solo devolviera un valor y no hubiera que “desencapsular” los datos):

```
[returnCode, sensorDataString] = robot.vrep.simxGetStringSignal...
    (robot.v_clientID, 'Turtlebot2_WidowX_nameOfSensor_data', ...
    robot.vrep.simx_opmode_buffer);
sensorData = robot.vrep.simxUnpackFloats(sensorDataString);
```

A continuación, generamos el mensaje en sí mismo. Para ello, primero debemos comprobar si el tipo de mensaje se encuentra dentro de los soportados por la Robotics System Toolbox de MATLAB. En ese caso, estamos de enhorabuena, puesto que obtener el mensaje es directo:

```
message = rosmessage('messageType');
```

Si no es así, tendremos que generar nosotros el mensaje. La forma más sencilla de hacer eso es mediante un *struct* de MATLAB. Tendremos que fijarnos en el mensaje del sensor real e imitarlo en sus campos.

Una vez que tenemos el mensaje, tenemos que proceder a rellenar sus campos. Para ello, haremos las modificaciones oportunas sobre los datos del sensor, e iremos haciendo las asignaciones a cada uno de los campos del mensaje.

Si el mensaje tiene un *header*, podemos obtener uno con el número de secuencia consecutivo y sus campos rellenos de la siguiente manera:

```
message.Header = robot.getMessageHeader();
```

No está de más añadir un control de errores a nuestra nueva función. Obsérvese, que en el ejemplo que hemos hecho dicho control de errores está ubicado en la llamada a la función que trae los datos desde V-REP (*simxGetStringSignal*). Dependiendo de la complejidad de la función, puede ser interesante añadir más puntos de control.

Con esto ya tenemos terminada nuestra función. La guardamos dentro de la carpeta “@VREP”, y añadimos su cabecera al fichero de definición de clase (“VREP.m”). Es importante que añadamos la cabecera al apartado de funciones privadas, para que así no sea visible para el usuario.

Siguiente cuestión: inicializar el canal de comunicación de esa nueva señal. Puesto que el modo de operación² que hemos establecido en la llamada a la función que trae los datos de V-REP ha sido “buffer”, hemos de inicializarlo antes de la primera llamada.

Para hacer esto, hemos de añadir la línea que se muestra en el siguiente recuadro en la función “rosinit.m”, que se encuentra en nuestra *toolbox*. Si no hacemos esto, no obtendremos los valores del sensor y muy probablemente el programa terminará con un error.

```
robot.vrep.simxGetStringSignal(robot.v_clientID, ...  
    'Turtlebot2_WidowX_nameOfSensor_data', ...  
    robot.vrep.simx_opmode_streaming);
```

Muy bien, pues ya tenemos terminada la función que genera el mensaje de nuestro nuevo sensor. Lo que nos falta por hacer es emular el *topic* correspondiente (o los *topics*, según el caso ante el que nos encontremos).

Para ello, solo vamos a tener que tocar una función: “receiveFromVREP”. Evidentemente, si el sensor tiene alguna acción de control (por ejemplo, una cámara con un motor

²Puede ser interesante que el desarrollador eche un vistazo a los distintos modos de operación que tiene la API remota de V-REP.

en la base que la gire), tendremos que añadir el *topic* de control que sea necesario, pero eso lo abordaremos en la siguiente sección.

Si observamos la estructura de esa función, veremos que no es muy compleja. Simplemente, se “trocea” el *topic* y se van comparando cada uno de sus términos hasta encontrar la coincidencia. El funcionamiento puede verse en el siguiente ejemplo:

```
subscriber = strsplit(subscrib , '/');

% TOPIC: /my_sensor/image/RGB_data

switch (char(subscriber(2)))
    case 'my_sensor'
        switch (char(subscriber(3)))
            case 'image'
                switch (char(subscriber(4)))
                    case 'RGB_data'
                        [v_msg, returnCode] = ...
                        robot.getNameOfSensorMsg();
                    end
                end
            end
        end
    end
end
```

Así pues, lo único que tenemos que hacer es añadir el *topic* que nos ocupa, comparando cada una de sus partes y llamando a la función que acabamos de crear.

Para acabar, hemos de añadir el nombre y el tipo del nuevo *topic* a la función “setVREPTopics”. Esto hará que nuestro *topic* se muestre al usuario en caso de llamar a la función que lista los que tenemos disponibles (*rostopic('list')*).

B.3. Añadir un actuador al modelo

Aparte de sensores, también podemos añadir actuadores a nuestro modelo. Un ejemplo de ello sería añadir un servomotor. Este servomotor podría servirnos para hacer girar un sensor que tengamos ensamblado sobre el mismo.

Para ello, procedemos de forma similar a la de añadir un sensor. Tenemos que comenzar por añadir una articulación, haciendo clic con el segundo botón en cualquier punto de la escena. En este caso, vamos a añadir una articulación de tipo *revolute*.

Esta articulación se añadirá en la posición X=0, Y=0, por lo que si tenemos al conjunto robótico en esa posición, es posible que no la veamos.

Ahora, debemos mover esa articulación al punto que deseemos, y cambiamos el tamaño de la misma haciendo doble clic en su icono de la jerarquía. Nótese que tanto la longitud como el diámetro de la misma son solo parámetros visuales, no afectando al comportamiento dinámico de la articulación.

En la figura B.2 puede verse un ejemplo de colocación.

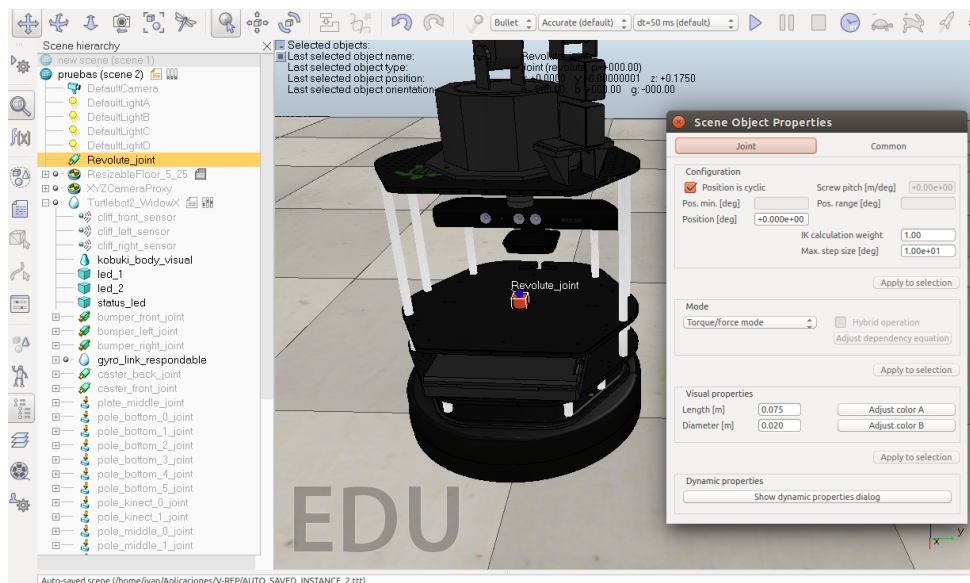


Figura B.2: Una articulación nueva sobre Turtlebot

Este servomotor puede servirnos para mover un transmisor, tal y como puede verse en la figura B.3. De esta manera, podemos hacer que gire, estableciendo una velocidad constante, o implementar un controlador de manera que siempre apunte hacia el mismo lado, independientemente del ángulo del conjunto robótico.

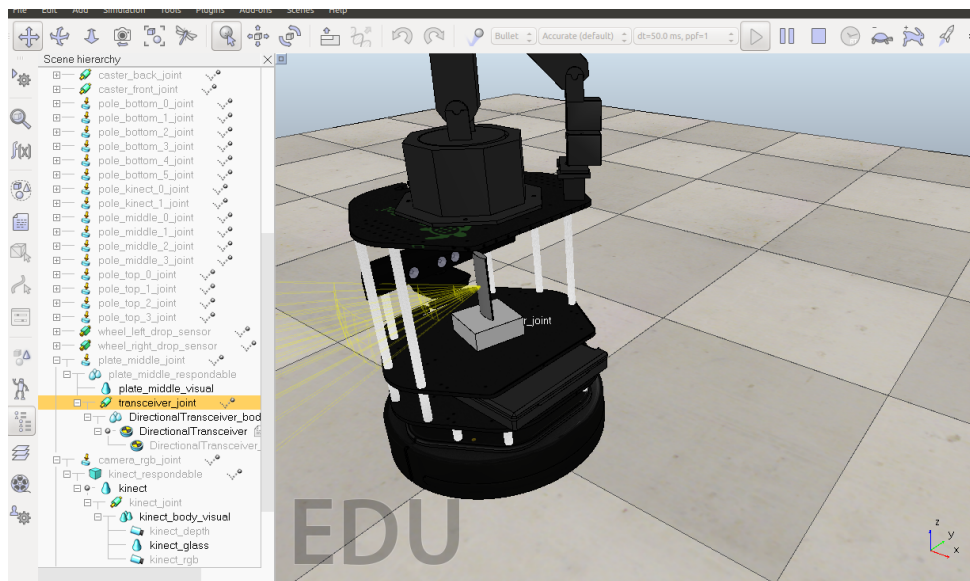


Figura B.3: Transmisor sobre servomotor en Turtlebot

Estas cuestiones hemos de establecerlas haciendo doble clic sobre el icono de la articulación, y pulsando el botón “Show dynamic properties dialog”.

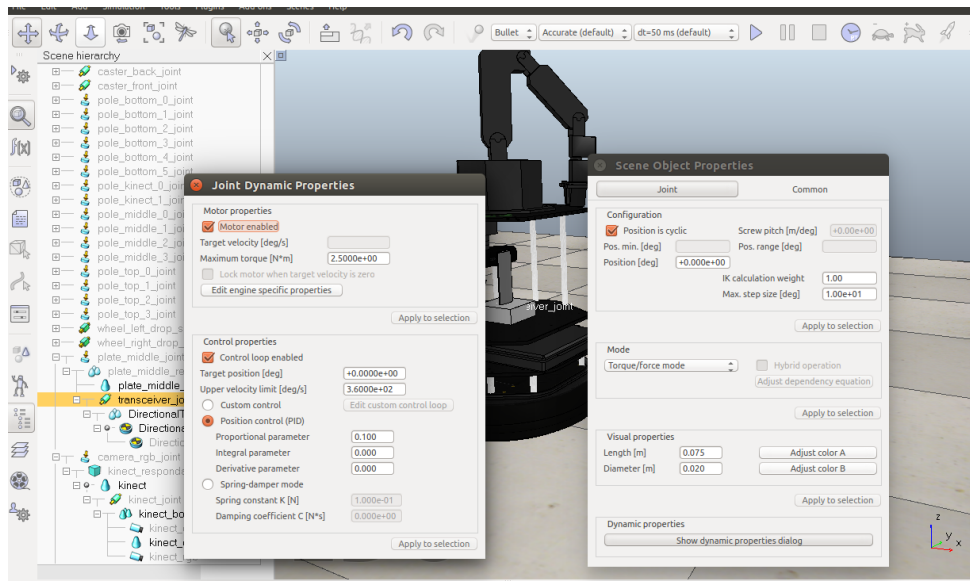


Figura B.4: Ventana de configuración de la articulación

B.4. Añadir un servicio en MATLAB

Otra de las cosas que podemos llevar a cabo es añadir servicios a nuestro entorno de programación.

Recordemos que un servicio podría entenderse como una llamada a un procedimiento a la que se suele obtener una respuesta por parte del servidor que la ejecuta. Ejemplo de ello son los servicios del brazo WidowX, donde nos encontramos los que relajan las articulaciones del mismo.

La manera de hacerlo es muy similar a la de añadir un *topic*. Primero, creamos una función que ejecute el servicio. Esto significa que, por norma general, vamos a emular la ejecución de dicho servicio en MATLAB, con idea de no sobrecargar en exceso a V-REP.

En esa función añadimos las actuaciones o lecturas de sensores que estimemos pertinentes (también pueden ser cambios en los parámetros del modelo o de la simulación), y devolvemos el mensaje de respuesta creado con la función *rosmesssage* que nos proporciona la Robotics System Toolbox.

Lo último que tendríamos que hacer es llamar a esa función en *callVREPSERVICE*, de forma similar a como hemos hecho con los *topics*:

```
switch (char(serv(2)))
    case 'arm_1_joint'
        switch (char(serv(3)))
            case 'relax'
                returnCode = robot.setWidowXJointRelaxed('arm');
                response_msg = rosmesssage('arbotix_msgs/RelaxResponse');
```

B.5. Señales en V-REP

Las señales son pieza clave en la comunicación entre V-REP y MATLAB. Por ello, son merecedoras de una sección en la que las describamos de manera pormenorizada, indicando cada uno de sus campos, tipos y valores posibles.

Es posible añadir campos a estas señales para ampliar funcionalidades o añadir tantas señales nuevas como precisemos. De hecho, a no ser que la ampliación consista en, por ejemplo, una actualización del *firmware* de Kobuki que añada valores a un *topic*, no es recomendable modificar las señales ya existentes.

Si por el contrario, añadimos un sensor nuevo al modelo, lo ideal es añadir una señal nueva, tal y como se sugiere en la sección correspondiente de este manual.

Turtlebot2_WidowX_kobuki_sensor_state

Tal y como se muestra en el cuadro B.1, esta señal está formada por 17 variables de tipo *float* encapsuladas en un *string*. Contiene los estados de varios sensores de la base Kobuki, además del tiempo de simulación en el que fueron capturados los datos.

Esta señal se utiliza, por ejemplo, para el *topic* “/mobile_base/sensors/core”.

Variable	Descripción
<i>Float</i> 1	Tiempo de simulación, en segundos
<i>Float</i> 2	Estado del bumper derecho (1 → choque)
<i>Float</i> 3	Estado del bumper central (1 → choque)
<i>Float</i> 4	Estado del bumper izquierdo (1 → choque)
<i>Float</i> 5	Estado del wheel_drop derecho (1 → dropped)
<i>Float</i> 6	Estado del wheel_drop izquierdo (1 → dropped)
<i>Float</i> 7	Estado del cliff derecho (1 → precipicio)
<i>Float</i> 8	Estado del cliff central (1 → precipicio)
<i>Float</i> 9	Estado del cliff izquierdo (1 → precipicio)
<i>Float</i> 10	Distancia a suelo del cliff derecho (metros)
<i>Float</i> 11	Distancia a suelo del cliff central (metros)
<i>Float</i> 12	Distancia a suelo del cliff izquierdo (metros)
<i>Float</i> 13	Contador de encoder derecho (grados, circular entre 0-65535)
<i>Float</i> 14	Contador de encoder izquierdo (grados, circular entre 0-65535)
<i>Float</i> 15	Estado del botón 0 (1 → pulsado)
<i>Float</i> 16	Estado del botón 1 (1 → pulsado)
<i>Float</i> 17	Estado del botón 2 (1 → pulsado)

Cuadro B.1: Descripción de la señal “Turtlebot2_WidowX_kobuki_sensor_state”.

Turtlebot2_WidowX_kobuki_quaternion_vel_accel

La siguiente señal con la que nos encontramos contiene 16 variables de tipo *float* encapsuladas en un *string* (ver cuadro B.2). A través de esta señal, es posible obtener la pose, el cuaternión, las velocidades y las aceleraciones de la base Kobuki. Las base Kobuki no consta de acelerómetro (al menos de momento), pero los campos están ya que forman parte del *topic* y pueden ser útiles en un futuro.

Esta señal se utiliza, por ejemplo, para la generación del mensaje del *topic* “/odom”.

Variable	Descripción
<i>Float</i> 1	Coordenada X de la pose, calculada en base a los encoders (metros)
<i>Float</i> 2	Coordenada Y de la pose, calculada en base a los encoders (metros)
<i>Float</i> 3	Coordenada Z de la pose, siempre vale 0 (metros)
<i>Float</i> 4	Primera componente del cuaternión
<i>Float</i> 5	Segunda componente del cuaternión
<i>Float</i> 6	Tercera componente del cuaternión
<i>Float</i> 7	Cuarta componente del cuaternión
<i>Float</i> 8	Velocidad lineal en el eje X (metros por segundo)
<i>Float</i> 9	Velocidad lineal en el eje Y (metros por segundo)
<i>Float</i> 10	Velocidad lineal en el eje Z (metros por segundo)
<i>Float</i> 11	Velocidad angular en el eje X (grados por segundo)
<i>Float</i> 12	Velocidad angular en el eje Y (grados por segundo)
<i>Float</i> 13	Velocidad angular en el eje Z (grados por segundo)
<i>Float</i> 14	Aceleración lineal en el eje X, siempre vale 0 (metros/segundo cuadrado)
<i>Float</i> 15	Aceleración lineal en el eje Y, siempre vale 0 (metros/segundo cuadrado)
<i>Float</i> 16	Aceleración lineal en el eje Z, siempre vale 0 (metros/segundo cuadrado)

Cuadro B.2: Descripción de la señal “Turtlebot2_WidowX_kobuki_quaternion_vel_accel”.

Turtlebot2_WidowX_kobuki_odometry_reset

Esta señal es de tipo *integer*, así que consta de un solo valor. El cuadro B.3 muestra su descripción. Básicamente, reinicia la odometría de Kobuki, si su valor es igual a 1. Una vez reiniciada, la señal se vuelve a poner a ‘0’ en V-REP.

Reiniciar la odometría implica poner a ‘0’ tanto la pose del robot, como su orientación y el valor de sus *encoders*.

Variable	Descripción
<i>Integer</i> 1	Reiniciar odometría, (1 → reiniciar, 0 → estado normal)

Cuadro B.3: Descripción de la señal “Turtlebot2_WidowX_kobuki_odometry_reset”.

Turtlebot2_WidowX_kobuki_motors_enabled

Esta señal, de tipo *integer*, habilita o deshabilita los dos motores de la base Kobuki (1 → habilitados, 0 → deshabilitados). Cuidado, deshabilitar los motores con la base en movimiento provoca un frenazo en seco. El cuadro B.4 contiene la descripción de su único campo.

Variable	Descripción
<i>Integer</i> 1	Motores habilitados (1 → sí, 0 → no)

Cuadro B.4: Descripción de la señal “Turtlebot2_WidowX_kobuki_motors_enabled”.

Turtlebot2_WidowX_kobuki_wheels_speed

Esta señal está formada por tres *floats* encapsulados en un *string*. Establece la velocidad objetivo de los motores de la base Kobuki. Véase cuadro B.5.

Variable	Descripción
<i>Float</i> 1	Velocidad angular objetivo del motor derecho, en radianes por segundo
<i>Float</i> 2	Velocidad angular objetivo del motor izquierdo, en radianes por segundo
<i>Float</i> 3	Valor de control (0-10241024 circular).

Cuadro B.5: Descripción de la señal “Turtlebot2_WidowX_kobuki_wheels_speed”.

Turtlebot2_WidowX_kobuki_led_[1, 2]

Esta señal está formada por un solo *integer*, que establece el estado del correspondiente LED. En el cuadro B.6 se encuentra la descripción de cada uno de los estados posibles.

Variable	Descripción
<i>Integer</i> 1	Estado del LED: <ul style="list-style-type: none">▪ 0 → Apagado▪ 1 → Encendido, color verde▪ 2 → Encendido, color naranja▪ 3 → Encendido, color rojo

Cuadro B.6: Descripción de las señales “Turtlebot2_WidowX_kobuki_led_[1, 2]”.

Turtlebot2_WidowX_kobuki_controller_info

Esta señal está formada por cuatro *floats* encapsulados en un *string* (cuadro B.7). Contiene información del controlador de los motores de la base Kobuki. Carece de significado en el comportamiento del modelo, está por completitud.

La idea es que si un programa utiliza estos valores por cualquier motivo, cogiéndolos del *topic* correspondiente del robot real, también estén disponibles si ese mismo programa se emplea para simulación.

Para modificar los datos, hay que hacerlo en el *script* principal del modelo en V-REP.

Variable	Descripción
<i>Float</i> 1	Tipo de controlador (por defecto o configurado por el usuario)
<i>Float</i> 2	Constante P del controlador
<i>Float</i> 3	Constante I del controlador
<i>Float</i> 4	Constante D del controlador

Cuadro B.7: Descripción de las señales “Turtlebot2_WidowX_kobuki_controller_info”.

Turtlebot2_WidowX_kobuki_version_info

Esta señal está compuesta por trece *floats* encapsulados en un *string*. Contiene información de la versión de la base Kobuki. Al igual que la señal anterior, carece de significado en el comportamiento del modelo, está por completitud.

Para modificar los datos, hay que hacerlo en el *script* principal del modelo en V-REP.

Turtlebot2_WidowX_widowx_gripper_joint

Esta señal es de tipo *float*, y por tanto contiene un solo campo. Establece la posición objetivo de la pinza de WidowX de acuerdo al cuadro B.9.

Turtlebot2_WidowX_widowx_[arm, shoulder, biceps, forearm, wrist]_joint

Estas señales son de tipo *float*, y por tanto contienen un solo campo. Establecen el ángulo objetivo de la articulación en cuestión de WidowX (cuadro B.10).

No tiene por qué ser el ángulo real de la misma, si no que la articulación recibirá ese comando como ángulo objetivo. El hecho de que lo alcance o no dependerá de la fuerza máxima de la articulación y de que esté chocando o no contra algún objeto.

Variable	Descripción
<i>Float 1</i>	Primer campo de la versión <i>hardware</i>
<i>Float 2</i>	Segundo campo de la versión <i>hardware</i>
<i>Float 3</i>	Tercer campo de la versión <i>hardware</i>
<i>Float 4</i>	Primer campo de la versión <i>firmware</i>
<i>Float 5</i>	Segundo campo de la versión <i>firmware</i>
<i>Float 6</i>	Tercer campo de la versión <i>firmware</i>
<i>Float 7</i>	Primer campo de la versión <i>software</i>
<i>Float 8</i>	Segundo campo de la versión <i>software</i>
<i>Float 9</i>	Tercer campo de la versión <i>software</i>
<i>Float 10</i>	Primer campo del udid
<i>Float 11</i>	Segundo campo del udid
<i>Float 12</i>	Tercer campo del udid
<i>Float 13</i>	Campo <i>features</i>

Cuadro B.8: Descripción de la señal “Turtlebot2_WidowX_kobuki_version_info”.

Variable	Descripción
<i>Float 1</i>	Posición objetivo: entre 0 [abierta] y 2,5 [cerrada] (sin unidad métrica)

Cuadro B.9: Descripción de la señal “Turtlebot2_WidowX_widowx_gripper_joint”.

Variable	Descripción
<i>Float 1</i>	Ángulo objetivo, en radianes

Cuadro B.10: Descripción de la señal “Turtlebot2_WidowX_widowx_[arm, shoulder, biceps, forearm, wrist]_joint”.

Turtlebot2_WidowX_widowx_[arm, shoulder, biceps, forearm, wrist, gripper]_joint_conf

Estas señales, de tipo *integer*, constan de un solo campo. Establecen el modo de configuración de cada una de las articulaciones de WidowX.

La información de los distintos modos de funcionamiento pueden verse en el cuadro B.11.

Turtlebot2_WidowX_widowx_reset_dynamics

Esta señal, de tipo *integer*, reinicia dinámicamente una articulación. Esta operación hay que realizarla cada vez que se cambia la velocidad máxima de una de ellas (cuadro B.12).

Si la señal vale ‘0’, no hay que reiniciar ninguna articulación. En otro caso, la articulación a reiniciar vendrá dada por el valor de la misma. Una vez reiniciada, volverá a valer ‘0’. Para más información, véase la implementación de la función *setWidowXJointMaxSpeed* en la *toolbox* de MATLAB.

Variable	Descripción
<i>Integer</i> 1	Configuración de la articulación: <ul style="list-style-type: none">▪ 0 → Deshabilitada, no acepta comandos. No ejerce fuerza, salvo el rozamiento interno.▪ 1 → Habilitada, estado “normal”. Acepta comandos y ejerce la máxima fuerza que puede desarrollar para llegar al ángulo objetivo.▪ 2 → Relajada. No ejerce fuerza, salvo el rozamiento interno. Acepta comandos, y una vez que recibe un nuevo comando pasa automáticamente a estado 1.

Cuadro B.11: Descripción de la señal “Turtlebot2_WidowX_widowx_[arm, shoulder, biceps, forearm, wrist]_joint_conf”.

Variable	Descripción
<i>Integer</i> 1	Identificador de la articulación a reiniciar

Cuadro B.12: Descripción de la señal “Turtlebot2_WidowX_widowx_reset_dynamics”.

Turtlebot2_WidowX_joint_states

Esta señal está compuesta por *floats* encapsulados en un string. Contiene información del estado de las articulaciones del modelo, es decir, tanto de Turtlebot como de WidowX (ver cuadro B.13).

Variable	Descripción
<i>Float 1</i>	Posición de la rueda izquierda de <i>Kobuki</i>
<i>Float 2</i>	Velocidad de la rueda izquierda de <i>Kobuki</i>
<i>Float 3</i>	Torque de la rueda izquierda de <i>Kobuki</i>
<i>Float 4</i>	Posición de la rueda derecha de <i>Kobuki</i>
<i>Float 5</i>	Velocidad de la rueda derecha de <i>Kobuki</i>
<i>Float 6</i>	Torque de la rueda derecha de <i>Kobuki</i>
<i>Float 7</i>	Posición de la articulación de la base de <i>WidowX</i>
<i>Float 8</i>	Velocidad de la articulación de la base de <i>WidowX</i>
<i>Float 9</i>	Torque de la articulación de la base de <i>WidowX</i>
<i>Float 10</i>	Posición del hombro de <i>WidowX</i>
<i>Float 11</i>	Velocidad del hombro de <i>WidowX</i>
<i>Float 12</i>	Torque del hombro de <i>WidowX</i>
<i>Float 13</i>	Posición del biceps de <i>WidowX</i>
<i>Float 14</i>	Velocidad del biceps de <i>WidowX</i>
<i>Float 15</i>	Torque del biceps de <i>WidowX</i>
<i>Float 16</i>	Posición del antebrazo de <i>WidowX</i>
<i>Float 17</i>	Velocidad del antebrazo de <i>WidowX</i>
<i>Float 18</i>	Torque del antebrazo de <i>WidowX</i>
<i>Float 19</i>	Posición de la muñeca de <i>WidowX</i>
<i>Float 20</i>	Velocidad de la muñeca de <i>WidowX</i>
<i>Float 21</i>	Torque de la muñeca de <i>WidowX</i>
<i>Float 22</i>	Posición de la pinza de <i>WidowX</i>
<i>Float 23</i>	Velocidad de la pinza de <i>WidowX</i>
<i>Float 24</i>	Torque de la pinza de <i>WidowX</i>

Cuadro B.13: Descripción de la señal “Turtlebot2_WidowX_widowx_joint_states”.

Turtlebot2_WidowX_hokuyo_data

684 parejas de *floats* encapsulados en un *string*. Contiene las distancias detectadas por el sensor Hokuyo. Donde $N = 2, 4, 6, 8, \dots$ (ver cuadro B.14).

Turtlebot2_WidowX_simulation_time

Esta señal es de tipo *float*, y su único campo contiene el valor de tiempo actual de simulación (ver cuadro B.15).

Variable	Descripción
<i>Float</i> 1	Tiempo de simulación en que fueron recogidos los datos
<i>Float</i> N	Enésima coordenada X del punto detectado
<i>Float</i> N + 1	Enésima coordenada Y del punto detectado

Cuadro B.14: Descripción de la señal “Turtlebot2_WidowX_hokuyo_data”.

Variable	Descripción
<i>Float</i> 1	Valor actual de tiempo de simulación, en segundos.

Cuadro B.15: Descripción de la señal “Turtlebot2_WidowX_simulation_time”.

Bibliografía

- [1] TurtleBot-2 (2016) <http://www.turtlebot.com/>
- [2] WidowX (2016) <http://www.trossenrobotics.com/widowxrobotarm>
- [3] Base Kobuki (2016) <http://kobuki.yujinrobot.com/>
- [4] Kinect (2016) <https://en.wikipedia.org/wiki/Kinect>
- [5] ROS (2016) https://en.wikipedia.org/wiki/Robot_Operating_System
- [6] Lua (2016) [https://en.wikipedia.org/wiki/Lua_\(programming_language\)](https://en.wikipedia.org/wiki/Lua_(programming_language))
- [7] ArbotiX-M (2016) <http://www.trossenrobotics.com/p/arbotix-robot-controller.aspx>
- [8] V-REP (2016) <http://www.coppeliarobotics.com/index.html>
- [9] Control Systems Engineering. Norman S. Nise, International Student Version, Sixth Edition, Wiley.
- [10] Peter Corke, Robotics, Vision and Control. Fundamental Algorithms in Matlab, Springer, 2011
- [11] P.A.Laplante, Real-Time Systems Design and Analysis, IEEE Press 0-471-22855-9 2004
- [12] Javier González Jiménez, Visión por Computador, ITP Paraninfo, 1999
- [13] Robotics System Toolbox (2016) <http://es.mathworks.com/products/robotics/>
- [14] Differential Driver Actuator (2016) http://www.openrobots.org/morse/doc/stable/user/actuators/v_omega_diff_drive.html
- [15] Real Academia de la Lengua Española (2016) <http://www.rae.es/>
- [16] Robotnik (2016) <http://www.robotnik.es/>
- [17] Nootrix Virtual Machine (2016) <http://nootrix.com/software/ros-indigo-virtual-machine/>

- [18] Javier Gonzalez-Jimenez, J.R. Ruiz-Sarmiento, Cipriano Galindo. (2013). Improving 2D Reactive Navigators with Kinect.
- [19] Lucas Nogueira (2014).Comparative Analysis Between Gazebo and V-REP Robotic Simulators.