

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	Recive 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.
isVREPEnabled	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
drawKinectImage	Dibuja la imagen de Kinect, tanto la parte RGB como la parte de profundidad, usando la función <i>surf</i> de MATLAB.
getConsoleMsgAndDispInMatlab	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.
getKobukiPoseAndOrientation	Devuelve la <i>pose</i> de la base Kobuki a partir del mensaje recibido en el <i>topic</i> “/odom”.
thetaTo360	Realiza la conversión entre un ángulo perteneciente al rango [-180, 180] al rango [0, 360].

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 habramos 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.

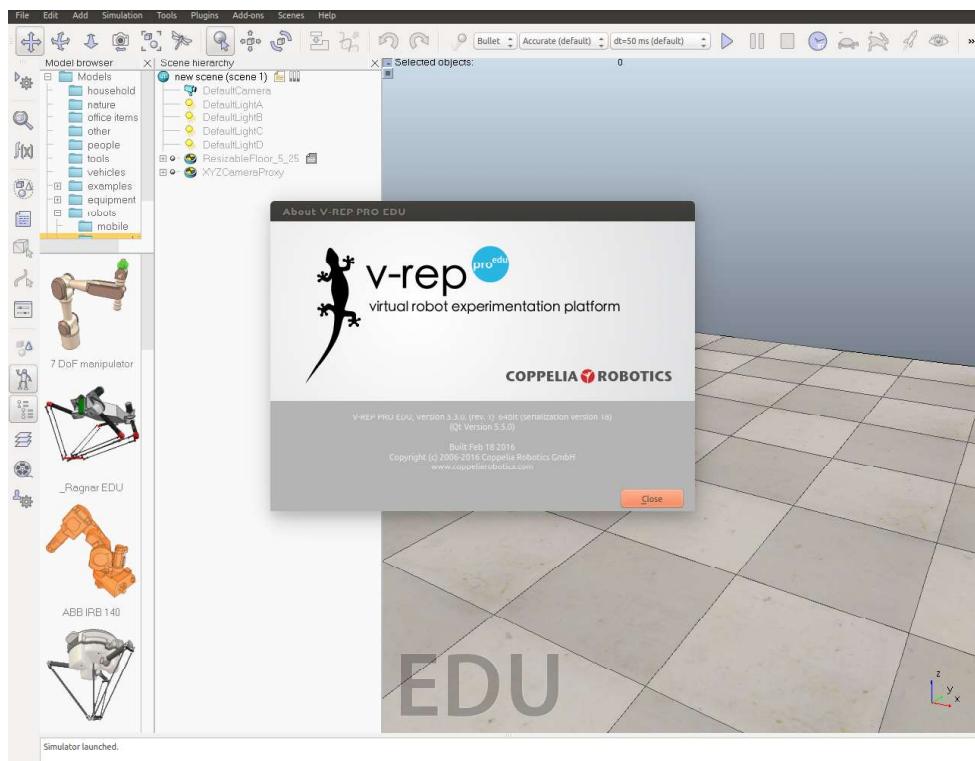


Figura A.1: Abriendo V-REP por primera vez.

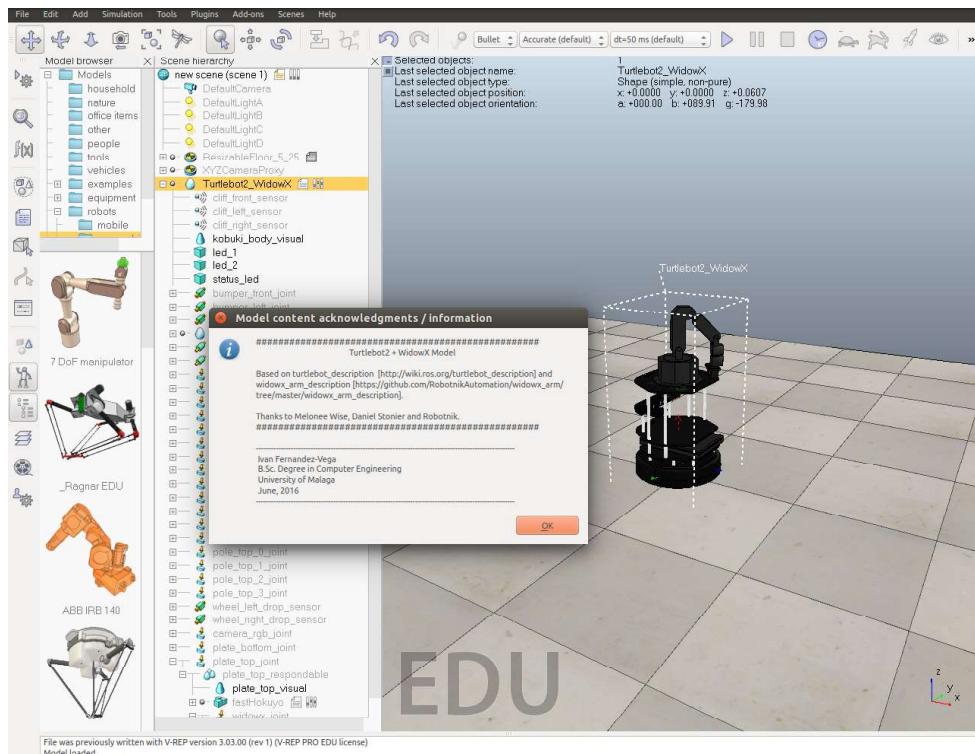


Figura A.2: Cargando el modelo en una escena de V-REP.

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.

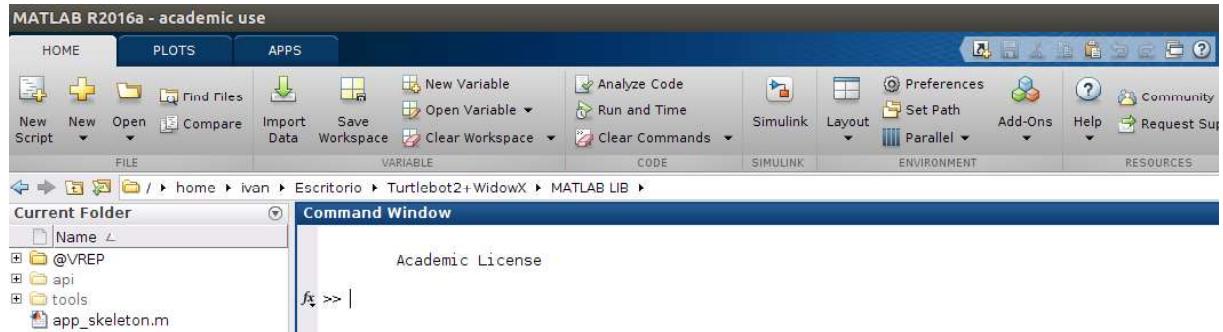


Figura A.4: MATLAB con el *path* establecido.

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 linea 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 linear 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 *rosmessage*, pasándole como argumento los dos *publishers*:

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

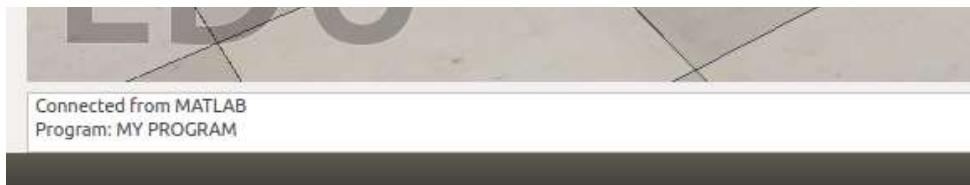


Figura A.5: Mensaje en la barra de estado de V-REP.

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 *getKobuki-PoseAndOrientation* 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 envíemos 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.

A.5. Simulando nuestra primera aplicación

Antes de simular nuestro programa, vamos a echar un vistazo a los parámetros de la simulación que podemos configurar.

En primer lugar, tenemos que tener en cuenta que la API remota solo está habilitada cuando la simulación está corriendo. Esto está configurado así en el *script* principal del modelo. Existe la posibilidad de que la API comience a escuchar cuando arranca V-REP, pero esto implica tener que modificar cuestiones relativas al propio programa y que no están por defecto en la versión que nos descargamos de la web.

El hecho de que la API remota esté habilitada incluso con la simulación parada permite iniciar y parar la misma remotamente. Si bien esto puede resultar interesante en algún caso, implicaría manipular funciones internas de la API, cosa que para un usuario poco experimentado puede resultar engoroso.

Por tanto, en principio vamos a usar el funcionamiento implementado. Es decir, iniciaremos la simulación en V-REP (quedando la API remota habilitada y escuchando en el puerto 19999) y a continuación ejecutaremos nuestro programa en MATLAB.

En lo que a la simulación en sí se refiere, podemos ajustar varios parámetros. Algunos de ellos están disponibles con la simulación parada, y otros con la simulación en marcha. Para ello, vamos a echarle un vistazo a los botones disponibles en la barra superior de V-REP (figura A.6).



Figura A.6: Parámetros configurables de la simulación.

De izquierda a derecha, lo primero que nos encontramos es un botón con una especie de bola saltando. Este botón solo puede ser pulsado durante la simulación, y nos permite ver el contenido dinámico de una escena.

Es bastante útil, puesto que permite saber qué formas y qué elementos se utilizan para el cálculo de las colisiones. Tengamos en cuenta que los elementos que se utilizan para calcular las colisiones suelen ser formas más básicas que las empleadas para la parte visual. En la figura A.7, podemos ver el contenido dinámico de nuestra escena de ejemplo. En este caso, solo tenemos al conjunto robótico y al suelo.

Justo al lado de ese botón, hay un desplegable. Permite elegir el motor físico a utilizar durante la simulación, entre los siguientes:

- *Bullet*. Es de código abierto y es el recomendado para el conjunto robótico.
- *ODE*. También de código abierto, pero suele dar problemas con la pinza de WidowX.
- *Vortex*. Opción propietaria y limitada a 20 segundos de simulación.
- *Newton*. Es una versión BETA, de momento.

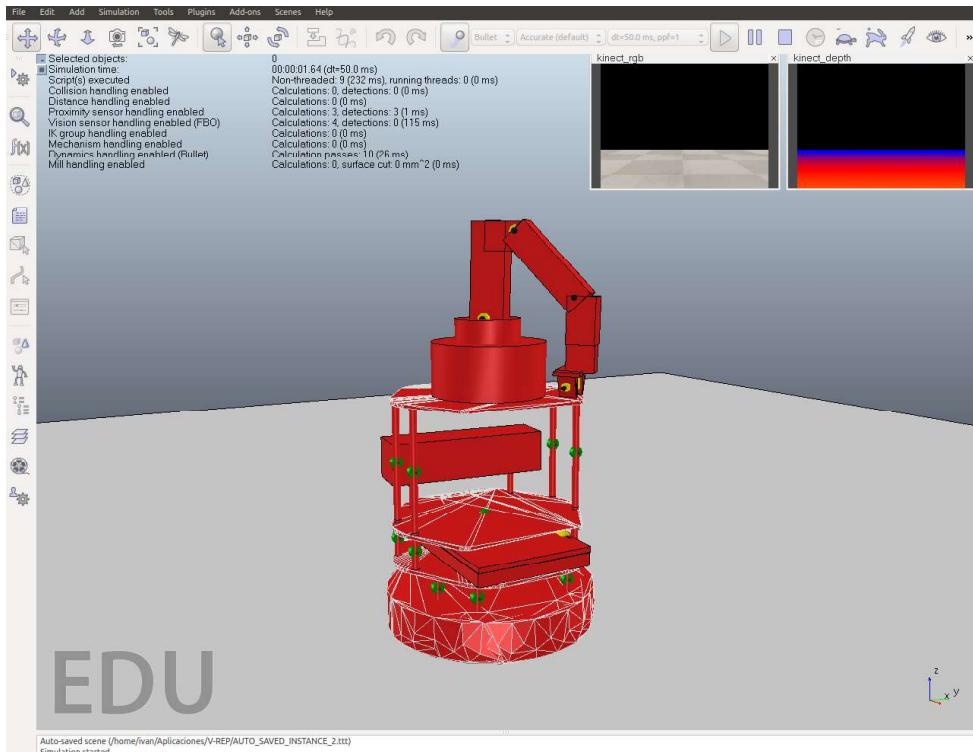


Figura A.7: Contenido dinámico de la escena.

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.

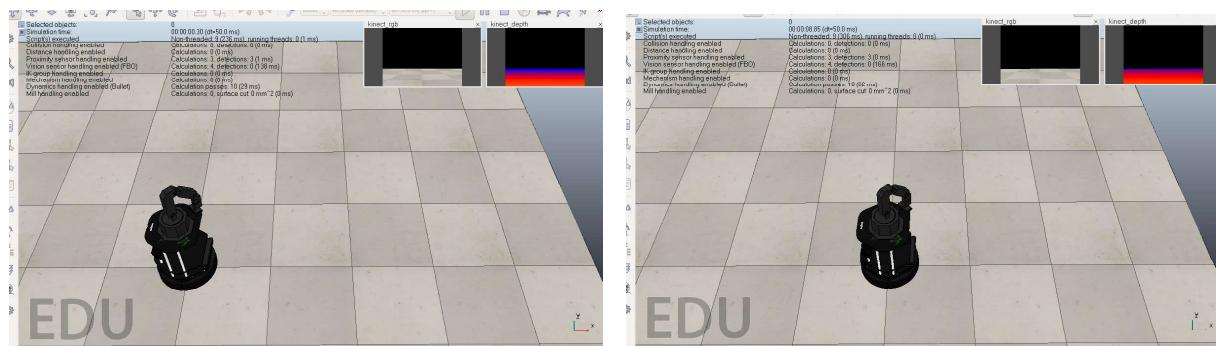


Figura A.8: Resultado de ejecución del programa de prueba.

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 ícono 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.

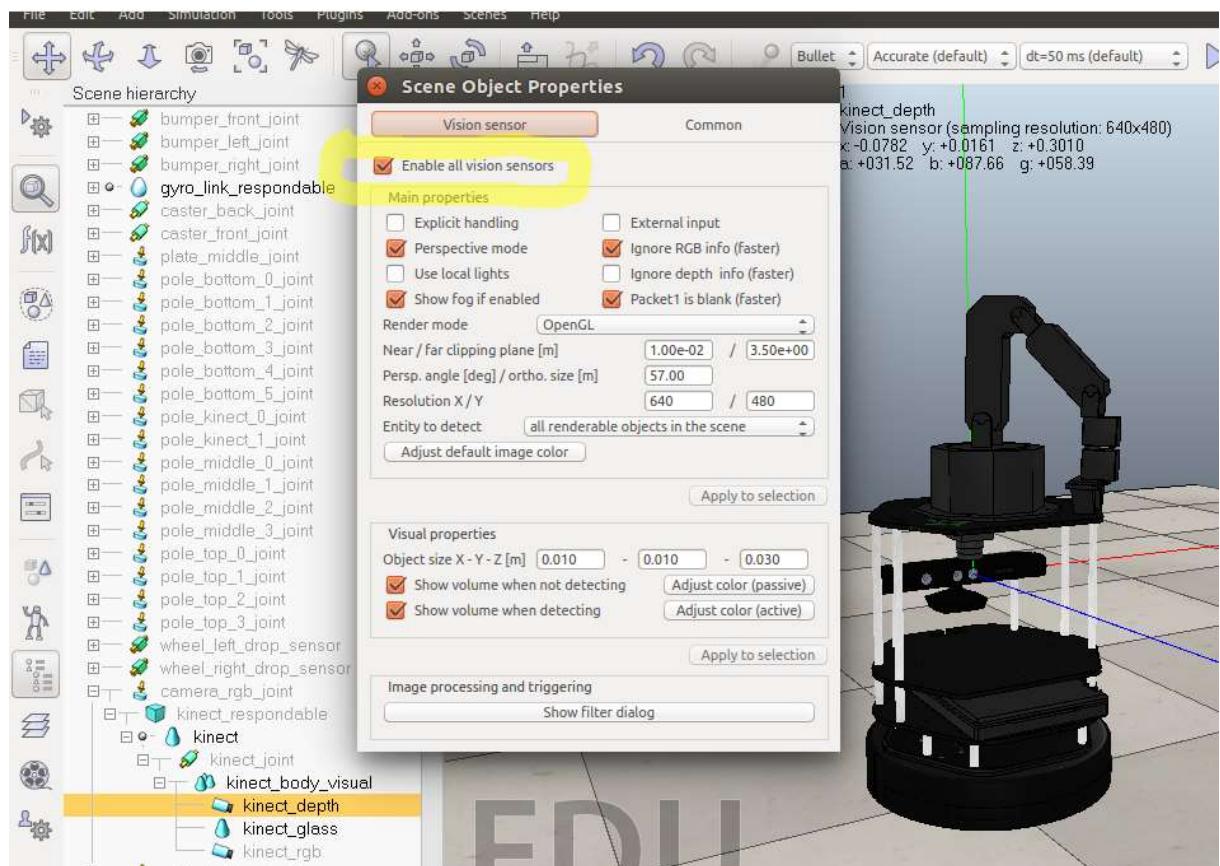


Figura A.9: Deshabilitar los sensores de visión.

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.

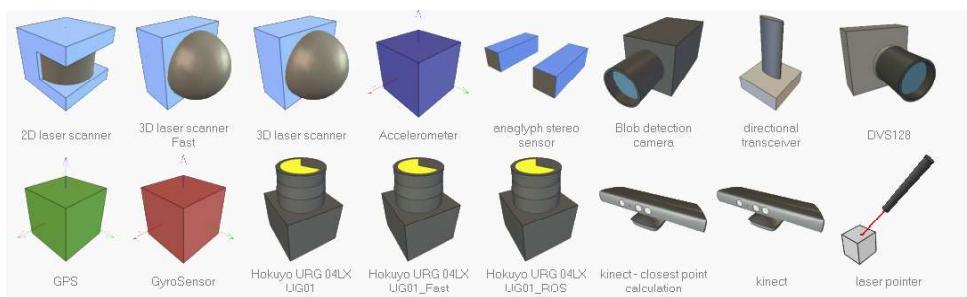


Figura B.1: Algunos sensores de la librería de V-REP

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, generaremos 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 llenar 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 llenos 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
```

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

A parte 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 ícono 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.

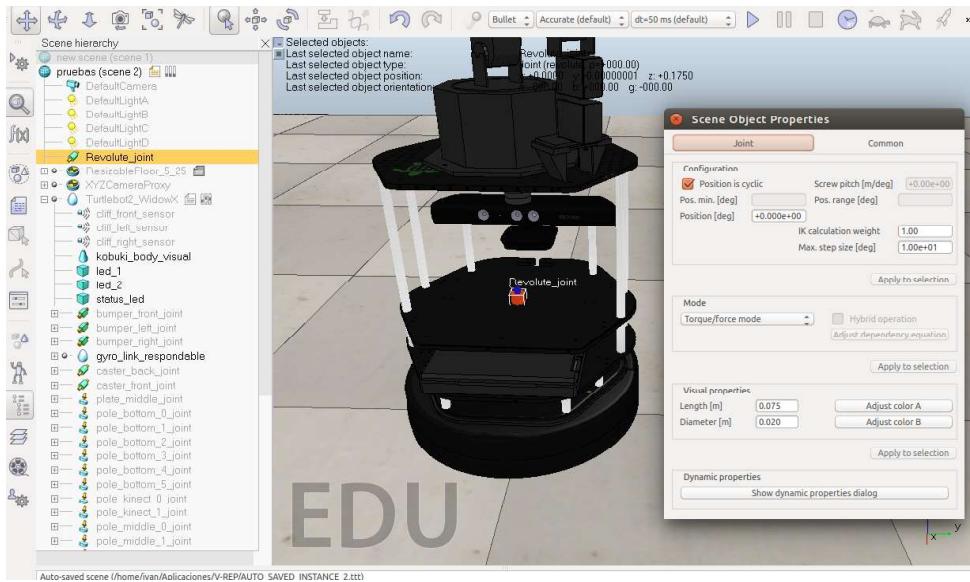


Figura B.2: Una articulación nueva sobre Turtlebot

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.

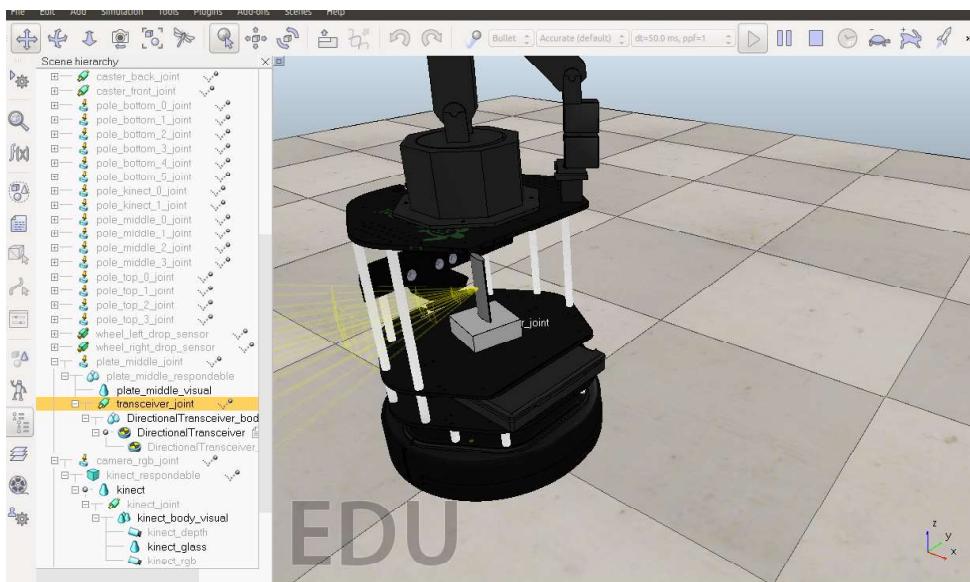


Figura B.3: Transmisor sobre servomotor en Turtlebot

Estas cuestiones hemos de establecerlas haciendo doble clic sobre el ícono de la articulación, y pulsando el botón “Show dynamic properties dialog”.

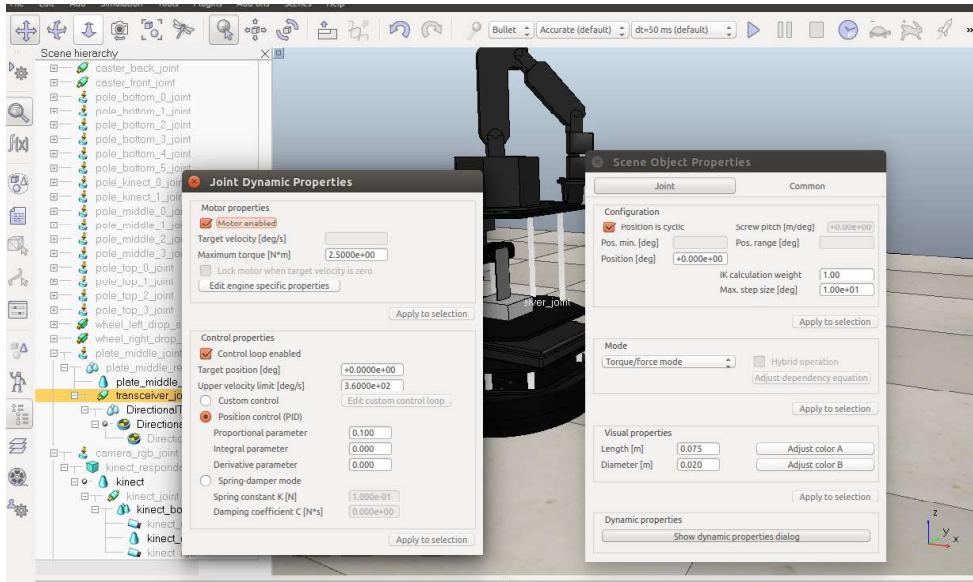


Figura B.4: Ventana de configuración de la articulación

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 pertenientes (también pueden ser cambios en los parámetros del modelo o de la simulación), y devolvemos el mensaje de respuesta creado con la función *rosmessage* 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 = rosmessage('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
Float 1	Tiempo de simulación, en segundos
Float 2	Estado del bumper derecho (1 → choque)
Float 3	Estado del bumper central (1 → choque)
Float 4	Estado del bumper izquierdo (1 → choque)
Float 5	Estado del wheel_drop derecho (1 → dropped)
Float 6	Estado del wheel_drop izquierdo (1 → dropped)
Float 7	Estado del cliff derecho (1 → precipicio)
Float 8	Estado del cliff central (1 → precipicio)
Float 9	Estado del cliff izquierdo (1 → precipicio)
Float 10	Distancia a suelo del cliff derecho (metros)
Float 11	Distancia a suelo del cliff central (metros)
Float 12	Distancia a suelo del cliff izquierdo (metros)
Float 13	Contador de encoder derecho (grados, circular entre 0-65535)
Float 14	Contador de encoder izquierdo (grados, circular entre 0-65535)
Float 15	Estado del botón 0 (1 → pulsado)
Float 16	Estado del botón 1 (1 → pulsado)
Float 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
Float 1	Coordenada X de la pose, calculada en base a los encoders (metros)
Float 2	Coordenada Y de la pose, calculada en base a los encoders (metros)
Float 3	Coordenada Z de la pose, siempre vale 0 (metros)
Float 4	Primera componente del cuaternión
Float 5	Segunda componente del cuaternión
Float 6	Tercera componente del cuaternión
Float 7	Cuarta componente del cuaternión
Float 8	Velocidad lineal en el eje X (metros por segundo)
Float 9	Velocidad lineal en el eje Y (metros por segundo)
Float 10	Velocidad lineal en el eje Z (metros por segundo)
Float 11	Velocidad angular en el eje X (grados por segundo)
Float 12	Velocidad angular en el eje Y (grados por segundo)
Float 13	Velocidad angular en el eje Z (grados por segundo)
Float 14	Aceleración lineal en el eje X, siempre vale 0 (metros/segundo cuadrado)
Float 15	Aceleración lineal en el eje Y, siempre vale 0 (metros/segundo cuadrado)
Float 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
Integer 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 están 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 1</i>	Tipo de controlador (por defecto o configurado por el usuario)
<i>Float 2</i>	Constante P del controlador
<i>Float 3</i>	Constante I del controlador
<i>Float 4</i>	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, veáse 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
Float 1	Posición de la rueda izquierda de <i>Kobuki</i>
Float 2	Velocidad de la rueda izquierda de <i>Kobuki</i>
Float 3	Torque de la rueda izquierda de <i>Kobuki</i>
Float 4	Posición de la rueda derecha de <i>Kobuki</i>
Float 5	Velocidad de la rueda derecha de <i>Kobuki</i>
Float 6	Torque de la rueda derecha de <i>Kobuki</i>
Float 7	Posición de la articulación de la base de <i>WidowX</i>
Float 8	Velocidad de la articulación de la base de <i>WidowX</i>
Float 9	Torque de la articulación de la base de <i>WidowX</i>
Float 10	Posición del hombro de <i>WidowX</i>
Float 11	Velocidad del hombro de <i>WidowX</i>
Float 12	Torque del hombro de <i>WidowX</i>
Float 13	Posición del biceps de <i>WidowX</i>
Float 14	Velocidad del biceps de <i>WidowX</i>
Float 15	Torque del biceps de <i>WidowX</i>
Float 16	Posición del antebrazo de <i>WidowX</i>
Float 17	Velocidad del antebrazo de <i>WidowX</i>
Float 18	Torque del antebrazo de <i>WidowX</i>
Float 19	Posición de la muñeca de <i>WidowX</i>
Float 20	Velocidad de la muñeca de <i>WidowX</i>
Float 21	Torque de la muñeca de <i>WidowX</i>
Float 22	Posición de la pinza de <i>WidowX</i>
Float 23	Velocidad de la pinza de <i>WidowX</i>
Float 24	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 1</i>	Tiempo de simulación en que fueron recogidos los datos
<i>Float N</i>	Enésima coordenada X del punto detectado
<i>Float N + 1</i>	Enésima coordenada Y del punto detectado

Cuadro B.14: Descripción de la señal “Turtlebot2_WidowX_hokuyo_data ”.

Variable	Descripción
<i>Float 1</i>	Valor actual de tiempo de simulación, en segundos.

Cuadro B.15: Descripción de la señal “Turtlebot2_WidowX_simulation_time ”.