Autores:
Alejandro Olivas González.
Edison P. Velasco Sánchez.
Francisco Candelas Herias.
Pablo Gil Vázquez.
Santiago Puente Méndez.

Grupo AUROVA

Robótica Inteligente

Máster en Inteligencia Artificial

Práctica

En la práctica de la asignatura de Robótica Inteligente se utilizará el simulador Gazebo y ROS para programar un brazo robótico modelo UR5 y un robot móvil.

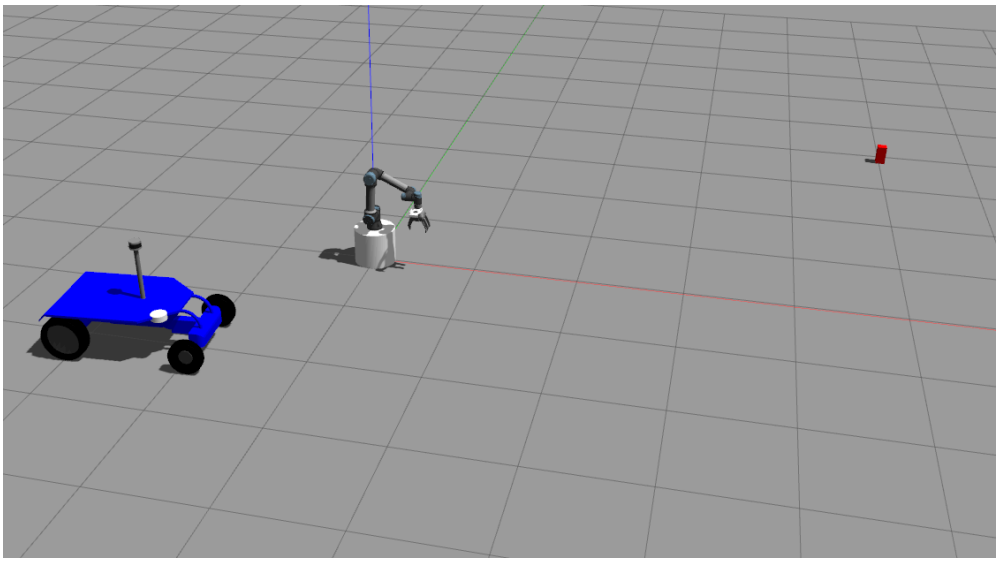

Fig. 1. Escena de la práctica en el simulador Gazebo.

Objetivos.

- Conocer conceptos de ROS y el uso del simulador Gazebo.
- Realizar el agarre de un objeto mediante el uso de un brazo robótico de 6 DoF y una pinza de 3 dedos.
- Detectar y localizar un objeto mediante una cámara RGB-D.
- Detectar mediante un sensor LiDAR 3D obstáculos que impidan el movimiento del robot móvil.
- Planificar el movimiento de un robot móvil tipo Ackermann hacia un punto deseado evadiendo posibles obstáculos detectados.
- Coordinar las tareas de agarre de un objeto con el brazo robótico de 6 DoF y la planificación de movimiento del robot móvil tipo Ackermann, de tal manera que el robot móvil se acerque al brazo robótico y este deje un objeto encima del robot móvil tipo Ackermann.

Indicaciones generales.

Esta práctica se dividirá en tres partes. La primera es agarrar un objeto de color rojo con un brazo robótico de 6 DoF. La segunda parte es mover un robot móvil tipo Ackermann hasta la zona donde está el brazo robótico, considerando la presencia de obstáculos. La última parte consiste en juntar la parte uno y dos, teniendo como objetivo que el brazo robótico deje el objeto sobre el robot móvil.

El **entregable** de la práctica será un documento en el que se explique el código realizado para poder completar las tareas, además de capturas mostrando los resultados obtenidos. La longitud máxima de este documento es de **10 páginas**. También los profesores evaluarán el trabajo realizado durante las clases.

Para facilitar el desarrollo de la práctica se utilizara un imagen docker la cual está basada en ROS:Noetic-desktop-focal y posee el ambiente simulado en Gazebo que se va a utilizar (ver Figura 1). Además la imagen docker posee el *workspace* que contiene todos los repositorios necesarios para el uso de la simulación del brazo robótico y del robot móvil.

Tanto el *workspace* como las instrucciones para construir el contenedor con un Dockerfile puede descargarse mediante del link:

https://drive.google.com/drive/folders/1GJUXzkXttHdUL47xePC1Jd44maSd1625?usp=sharing

Dentro del contenedor se tiene el *workspace* con los repositorios para la práctica (ver Figura 2).

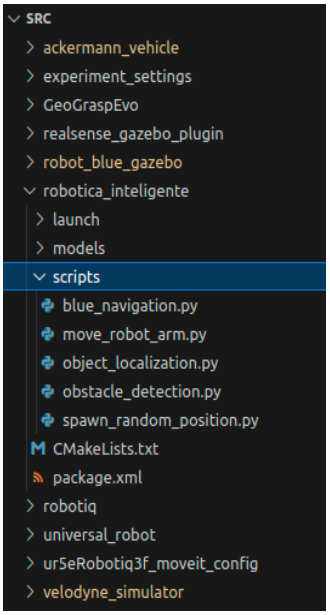

Fig. 2. Árbol de workspace

Para desarrollar la práctica, se debe completar los #TODO de los ficheros en la carpeta *robotica_inteligente/scripts/* (ver Figura 2). Cada uno de los ficheros está elaborado en Python, los cuales interactúan entre el simulador mediante topics de ROS.

Es muy **IMPORTANTE** compartir la carpeta donde modificaremos los ficheros dentro del contenedor de la imagen Docker, esto para que los cambios realizados puedan ser **guardados**.

Debido a que el contenedor de la imagen docker creada se reinicia cada vez que salimos de ella, es importante que los ficheros que se **modifiquen sean fuera del contenedor**, y para poder probar los cambios realizados, necesitamos **compartir el directorio** donde están los ficheros en el ordenador local al contenedor, para ello usaremos el directorio local:

/home/pclocal/robotica_inteligente

y el directorio dentro del contenedor será:

/home/docker/catkin_ws/src/robotica_inteligente

Dado ello podemos correr el contenedor de la imagen docker de la siguiente manera.

```
docker run --shm-size=1g --privileged --ulimit memlock=-1 --ulimit stack=67108864
--rm -it --net=host -e DISPLAY=:1 --user=root -v /tmp/.X11-unix:/tmp/.X11-unix:rw -v
/dev:/dev --name robotica_container --gpus all --cpuset-cpus=0-3 -v
/home/pclocal/robotica_inteligente:/home/docker/catkin_ws/src/robotica_inteligente
robotica-inteligente
```

A continuación se explica cada uno de los elementos al momento de lanzar la imagen docker:

1. **‘--shm-size=1g’**: Esta opción establece el tamaño de la memoria compartida que el contenedor puede usar. En este caso, se establece en 1 gigabyte.

2. **‘--privileged’**: Esta opción proporciona al contenedor acceso privilegiado al host. Esto significa que el contenedor tiene acceso a todos los dispositivos del host y puede realizar operaciones que de otro modo no serían posibles en un entorno de contenedor normal.

3. **‘--ulimit memlock=-1'**: Esto establece el límite de bloqueo de memoria en ilimitado. Permite que el contenedor bloquee toda la memoria física disponible.

4. **‘--ulimit stack=67108864’**: Esto establece el límite del tamaño de la pila para el contenedor en 67108864 bytes (64 megabytes).

5. **‘--rm’**: Esta opción indica a Docker que elimine el contenedor después de que finalice su ejecución.

6. **‘-it’**: Esto indica a Docker que mantenga abierta la entrada estándar (stdin) incluso si no está conectada y que asigne un terminal pseudo-TTY.

7. ‘**--net=host’**: Esto permite al contenedor utilizar la pila de red del host en lugar de su propia pila de red independiente.

8. **‘-e DISPLAY=:1’**: Esta opción establece la variable de entorno ‘DISPLAY’ dentro del contenedor, que es necesaria para que las aplicaciones GUI dentro del contenedor se comuniquen con el servidor X en el host.

9. **‘--user=root’**: Esto establece el usuario que ejecuta los procesos dentro del contenedor como ‘root’, lo que proporciona acceso completo a los recursos del sistema.

10. **‘-v /tmp/.X11-unix:/tmp/.X11-unix:rw’**: Esto monta el socket X11 del host dentro del contenedor para permitir que las aplicaciones GUI dentro del contenedor se comuniquen con el servidor X en el host.

11. **‘-v /dev:/dev’**: Esto monta el directorio ‘/dev’ del host dentro del contenedor, lo que le da al contenedor acceso completo a todos los dispositivos del host.

12. ‘**--name robotica_container’**: Esto asigna un nombre al contenedor, en este caso, "robotica_container".

13. **‘--gpus all’**: Esto permite al contenedor acceder a todas las GPU disponibles en el host.

14. **‘--cpuset-cpus=0-3’**: Esto establece los conjuntos de CPU que el contenedor puede utilizar, limitándolo a las CPUs 0 a 3.

15. ‘**-v /home/pclocal/robotica_inteligente:/home/docker/catkin_ws/src/robotica_inteligente’**: Esto monta el directorio `/home/pclocal/robotica_inteligente` del host en `/home/docker/catkin_ws/src/robotica_inteligente` dentro del contenedor.

16. **‘robotica-inteligente’**: Este es el nombre de la imagen de Docker que se utilizará para crear el contenedor.

Estas opciones están diseñadas para configurar y ejecutar el contenedor Docker de forma que pueda acceder a una amplia gama de recursos del sistema del host, incluidas GPU, dispositivos, red y entorno gráfico; recursos necesarios para poder desarrollar la práctica.

En los ordenadores de clase, se han de ajustar los permisos del sistema de ventanas de Linux, para poder mostrar las aplicaciones GUI que se ejecutan en la docker.

```
xhost +local:
```

Si se necesita una ventana nueva del contenedor se debe ejecutar:

```
docker exec -it robotica_container /bin/bash
```

Introducción a ROS

ROS (Robot Operating System) es una plataforma de código abierto que ofrece herramientas y bibliotecas para ayudar en el desarrollo de software para robots. Provee servicios similares a un sistema operativo para trabajar con robots: abstracción de hardware, comunicación mediante mensajes, control de dispositivos a bajo nivel, manejo de paquetes y otros comandos y utilidades.

Una de las características distintivas de ROS es su enfoque en la modularidad y la reutilización de código. Está compuesto por una serie de paquetes que pueden ser utilizados y combinados de manera flexible para adaptarse a las necesidades específicas de cada proyecto. Además, ROS cuenta con una activa comunidad de desarrolladores que contribuyen con nuevos paquetes y funcionalidades de forma regular.

Paquetes

Los *packages* son la unidad básica de organización de ROS. Pueden contener procesos (*nodes)*, librerías, o archivos de configuración. Los paquetes de ROS suelen tener la siguiente estructura:

- **src/:** Código fuente de los procesos.
- **include/:** Cabeceras de las librerías de C++.
- **scripts/:** Procesos ejecutables en Python.
- **launch/:** Contiene los archivos *launch* para lanzar varios procesos.
- **msg/:** Contienen los tipos de mensaje.
- **param/** y **config/:** Contienen archivos donde se definen parámetros de los procesos.
- **CMakeLists.txt:** Se utiliza durante la compilación del paquete. Se definen los procesos del paquete y sus dependencias.
- **package.xml:** Archivo que contiene metadatos del paquete, como las dependencias a otros paquetes.

Los *workspaces* en ROS son entornos de desarrollo dedicados que proporcionan una estructura organizada para la creación, compilación y gestión de proyectos robóticos. Un *workspace* típico de ROS incluye tres carpetas principales: `src`, `devel` y `build`. La carpeta `src` es donde residen los paquetes de ROS que constituyen los componentes individuales del proyecto. El proceso de compilación y construcción de los paquetes se realiza utilizando herramientas de compilación de ROS:

```
catkin_make
```

Es importante destacar que cada terminal utilizada para el desarrollo en ROS debe ser configurada adecuadamente para reconocer el *workspace* actual y sus paquetes. Esto se logra mediante el comando `source devel/setup.bash`, que establece las variables de entorno necesarias para que ROS pueda encontrar los paquetes y ejecutables del proyecto. En esta práctica el proyecto se encuentra en la carpeta **`catkin_ws`**.

Nodos

Los nodos son procesos individuales que realizan tareas específicas dentro de un sistema robótico. Estos nodos representan unidades de software autónomas que pueden comunicarse entre sí a través de mensajes ROS.

Cada nodo en ROS tiene su propio proceso independiente y puede ejecutarse en el mismo equipo o en diferentes equipos dentro de una red. Esto proporciona una arquitectura distribuida y modular para el desarrollo de sistemas robóticos, donde diferentes nodos pueden ejecutarse en diferentes plataformas y comunicarse entre sí de manera eficiente. Se comunican mediante mensajes de ROS en canales conocidos como *topics.*

Los nodos en ROS pueden desempeñar una amplia variedad de funciones, desde controlar sensores y actuadores hasta procesar datos, planificar movimientos, realizar percepción y más. Al dividir el software en múltiples nodos, se facilita la modularidad, la reutilización del código y la escalabilidad del sistema.

En ROS, los nodos necesitan de un *Master* para establecer las conexiones. Una vez establecida, es una conexión entre pares. El *Master* se encarga de registrar los *topics* de cada nodo. Para iniciar el *Master* se utiliza el comando:

```
roscore
```

Para ejecutar un nodo se utiliza el comando `rosrun`:

```
rosrun nombrePaquete nombreNodo
```

También se puede utilizar `rosnode list` para obtener la lista de nodos activos y `rosnode info nombreNodo` para obtener información del nodo, como qué topics utiliza para comunicarse y con qué otros nodos.

También se pueden utilizar los archivos *launch* para lanzar varios nodos a la vez. Estos archivos son ficheros tipo XML, que además de lanzar nodos, se utilizan para configurar los parámetros de estos. Estos archivos son esenciales para la ejecución de aplicaciones y sistemas robóticos complejos en ROS.

La guía del estilo de los archivos *launch* se encuentra en la documentación de ROS. De las etiquetas que se puede utilizar, las más importantes son:

- **`<node name="example" pkg="package_example" type="example.py>`**: Con esta etiqueta se lanza el nodo "example.py" del paquete "package_example".

- **`<include file="$(find pkg-name)/path/filename.launch"/>`**: Permite incluir otro archivo *launch* en el actual. En este ejemplo se utiliza una sustitución de variable, `$(find pkg-name)`, *find* se utiliza para encontrar el directorio de un paquete. La otra sustitución de variable más utilizada es `$(arg arg_name)`, que utiliza el valor de la variable *arg_name* previamente definida.

- **`<arg name="arg_name" default="arg_value"/>`**: Define una variable con un valor por defecto. Si en lugar de *default* se utiliza *value,* el valor de la variable no puede ser cambiado. Es utilizada para pasar argumentos a los archivos *launch* y se puede pasar su valor por la línea de comandos.

- **`<param name:"param_name" value="param_value"/>`**: Define un parámetros que se carga en Parameter Server de ROS. Los nodos pueden acceder a estos parámetros.

De forma parecida a `rosrun`, los archivos *launch* se ejecutan con:

```
roslaunch nombrePaquete archivoLaunch
```

En este caso, si no se ha iniciado el *Master,* `roslaunch` se encarga de iniciarlo.

Topics

La comunicación en ROS se realiza mediante canales de mensajes llamados *topics.* Los nodos se pueden subscribirse o publicar en estos canales, de manera que la comunicación entre nodos es anónima, en el sentido que los nodos no saben con quién se están comunicando. En lugar de eso, los nodos que necesitan datos se suscriben al *topic* relevante, y los que generan datos los publican en el *topic* correspondiente.

Los *topics* son fuertemente tipados, definidos por un mensaje de ROS. Estos mensajes se pueden definir en los paquetes de ROS de esta manera, y existen numerosos paquetes que definen mensajes estándar. En esta práctica se utilizará principalmente geometry_msgs/Pose, geometry_msgs/PoseArray, ackermann_msgs/AckermannDrive y sensor_msgs/PointCloud2, entre otros.

El comando rostopic nos ofrece un conjunto de utilidades para manejar los mensajes de los topics. Por un lado, `rostopic list` nos muestra la lista completa de los *topics* actuales utilizados por los nodos. Por otro lado, `rostopic echo /nombreTopic` mostrará por pantalla los mensajes publicados en ese *topic.* También se pueden publicar mensajes con `rostopic pub`.

Un nodo de ROS puede tener múltiples *publishers* que se encargan de publicar los mensajes. Un ejemplo de un publisher de la práctica es el siguiente.

```
self.ackermann_command_publisher = rospy.Publisher(
            "/blue/ackermann_cmd",
            ackermann_msgs.msg.AckermannDrive,
            queue_size=10,
        )
```

Este *publisher* publica los comandos de velocidad y ángulo de giro para controlar el robot móvil. Publica mensajes de tipo "`ackermann_msgs.msg.AckermannDrive`" en el *topic* `/blue/ackermann_cmd`, donde *queue_size* es el tamaño máximo de mensajes que conserva antes de desecharlos. Para publicar un mensaje `ackermann_control` definido anteriormente se utiliza:

```
self.ackermann_command_publisher.publish(ackermann_control)
```

De manera similar, un ejemplo de *subscriber* de la práctica es el siguiente.

```
self.obstacles_subscriber = rospy.Subscriber("/obstacles",
   sensor_msgs.msg.PointCloud2, self.obstacles_callback, queue_size=1)
```

El *subscriber* recibe la nube de puntos que contiene los obstáculos alrededor del robot móvil, subscribiéndose al *topic* `/obstacles`. Estos son del tipo "sensor_msgs.msg.PointCloud2" y al recibir un mensaje, el nodo ejecuta la función `obstacles_callback`. Esta función se encarga de procesar el mensaje, que lo recibe como argumento de la función.

Los nodos de ROS se ejecutan constantemente y se debe configurar cuando se llaman a los *callbacks* de los *subscribers*. Existen dos formas estándar para ejecutar los nodos de ROS. La primera es definir la frecuencia de funcionamiento del nodo y un bucle principal donde se llaman a las funciones pertinentes:

```
rate = rospy.Rate(10)
while not rospy.is_shutdown():
    #Llamar a las funciones para realizar la tarea.

    rate.sleep()
```

En este ejemplo, se establece la frecuencia del nodo a 10 Hz, y luego se entra en el bucle principal donde se llamarían a las funciones del nodo, y luego se ejecuta la función `ros.sleep()` que espera 0.1 s, pero además llama a los *callbacks* de los *subscribers* del nodo si han llegado nuevos mensajes.

Por otro lado, una vez inicializado el nodo, se puede llamar a la función `ros.spin()`.
Esta función configura el nodo para esperar a que se reciban los mensajes, ejecutando entonces los *callbacks.* De esta manera, la tarea que realiza el nodo se debe ejecutar en los *callbacks*, ya que no tiene en este caso un bucle principal.

En ambos casos, se debe utilizar `ctrl+c` para detener el nodo y finalizar su ejecución.

RViz y Gazebo

Rviz es una herramienta de visualización 3D que permite la visualización interactiva de datos, mediante la suscripción a sus *topics*. RViz es altamente personalizable y puede utilizarse para visualizar una variedad de datos, incluyendo nubes de puntos, imágenes, modelos de robot, trayectorias planificadas, entre otros. Esto hace a RViz una herramienta útil para la depuración, permitiendo la comprensión visual del entorno y la evaluación del rendimiento de los sistemas robóticos.

El sistema de *frames* es una herramienta fundamental para gestionar la cinemática y la geometría de un sistema robótico. Proporciona una manera de representar y mantener un seguimiento de la posición y orientación relativas de los diferentes componentes del robot, así como de otros objetos en el entorno.

El sistema de *frames* de ROS se basa en una estructura de grafo que describe las relaciones de transformación entre los diferentes marcos de referencia (*frames*) en el sistema. Las transformaciones entre estos frames pueden ser tanto estáticas como dinámicas, lo que permite modelar la cinemática del robot en movimiento. Estas transformaciones se publican de manera constante en el topic `/tf`, de manera que los distintos componentes del sistema robótico están sincronizados.

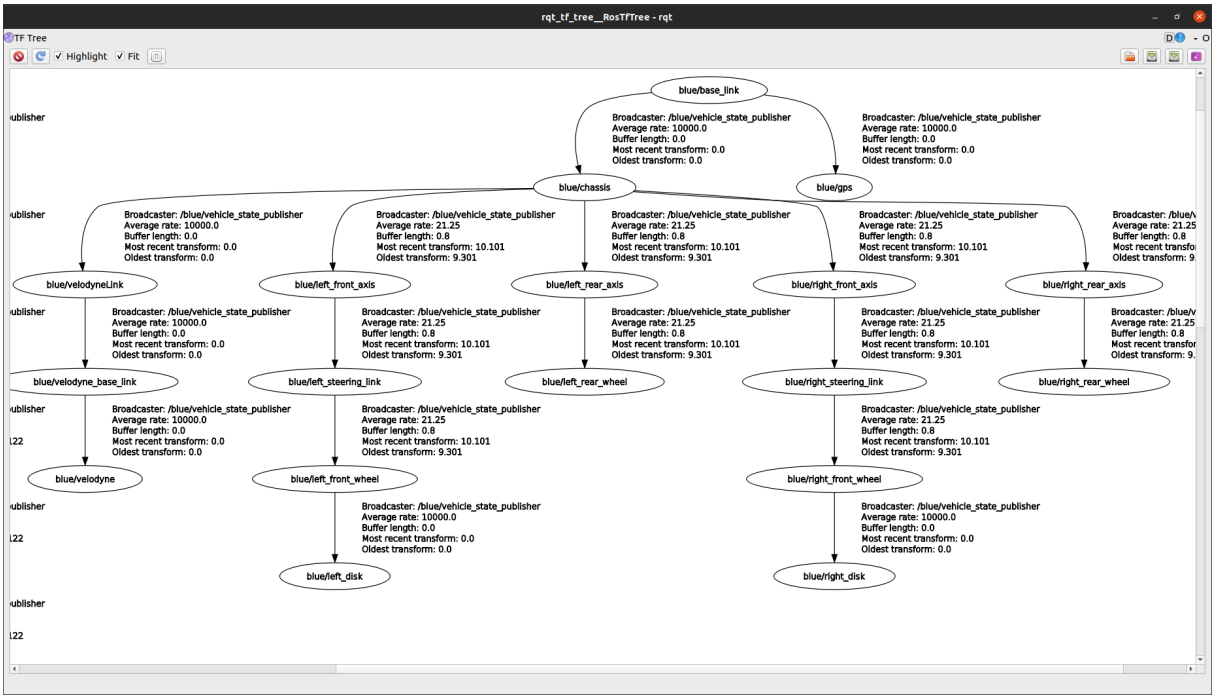

Fig. 3. Árbol de los *frames* del robot móvil de la práctica.

Rviz utiliza estas transformaciones para mostrar la información. Por ejemplo, si se muestra la nube de puntos desde el *frame* `/blue/velodyne`, mostrará los puntos en la posición que son capturados por el sensor LiDAR, pero en el *frame* `/blue/base_link` se mostrarán en base al robot. Esto es posible, porque los mensajes suelen contar con una estructura Header que indica el momento (*timestamp)* en el que se ha enviado el mensaje así como el *frame* de referencia. Esta información es importante para coordinar los sistemas.

Gazebo es un simulador robótico 3D que permite simular entornos y robots de manera realista. Gazebo proporciona un entorno de simulación configurable donde los desarrolladores pueden probar algoritmos, validar el comportamiento del robot y simular escenarios complejos antes de utilizar el código en hardware real. Gazebo, al ser compatible con ROS, se integra fácilmente con el desarrollo de software en ROS. Permite simular no solo el movimiento del robot, sino también la interacción con el entorno, como la física de objetos, la percepción de sensores y la interacción con otros objetos y robots simulados.

Primera parte: agarrar un objeto con un brazo robótico

En la primera parte de la práctica se desea mover el brazo robótico UR5 para el agarre de objeto de color rojo que se encuentra en el escenario. En el ambiente de simulación realizado con Gazebo, se tiene una cámara en la parte superior del escenario, de esta manera la cámara genera una imagen desde arriba del entorno. Esta cámara es esencial para localizar el objeto, y planificar así el movimiento del brazo robótico hacia el mismo.
De esta manera, dividimos esta primera parte en las secciones: Cargar escenario, Conocer espacio de trabajo, Localizar Objeto y Agarre de objeto.

Cargar escenario

Para mover el UR5, se utilizará el paquete **MoveIt.** Este paquete permite planificar el movimiento del robot utilizando su cinemática directa, resolver la cinemática inversa del robot, generar las trayectorias articulares para los controladores de bajo nivel ,considerar las colisiones con el entorno, entre otras herramientas para la planificación de movimiento de robots (https://moveit.ros.org/).

Para cargar la escena hay que lanzar el siguiente archivo *launch*:

```
roslaunch robotica_inteligente load_scene.launch
```

Nota: Recordar configurar cada terminal con `source devel/setup.bash`. *Además, para poder utilizar aplicaciones de interfaz gráfica dentro del contenedor de la imagen de Docker, se ha de ejecutar en el ordenador la instrucción* `xhost +local:` *.Es necesario hacerlo cada vez que se enciende el ordenador.*

Este archivo *launch* cargará el ambiente en Gazebo con: Un brazo robotico UR5, un objeto a manipular (en este caso un cubo de color rojo) y una cámara RGB-D que permite visualizar el escenario. El *launch* abrirá una ventana de Gazebo y dos de RViz, donde una mostrará la nube de puntos del robot móvil y otra el UR5 y permite controlarlo mediante **MoveIt**. En esta primera parte de la práctica nos centraremos en la ventana del UR5 (ver Fig. 4).

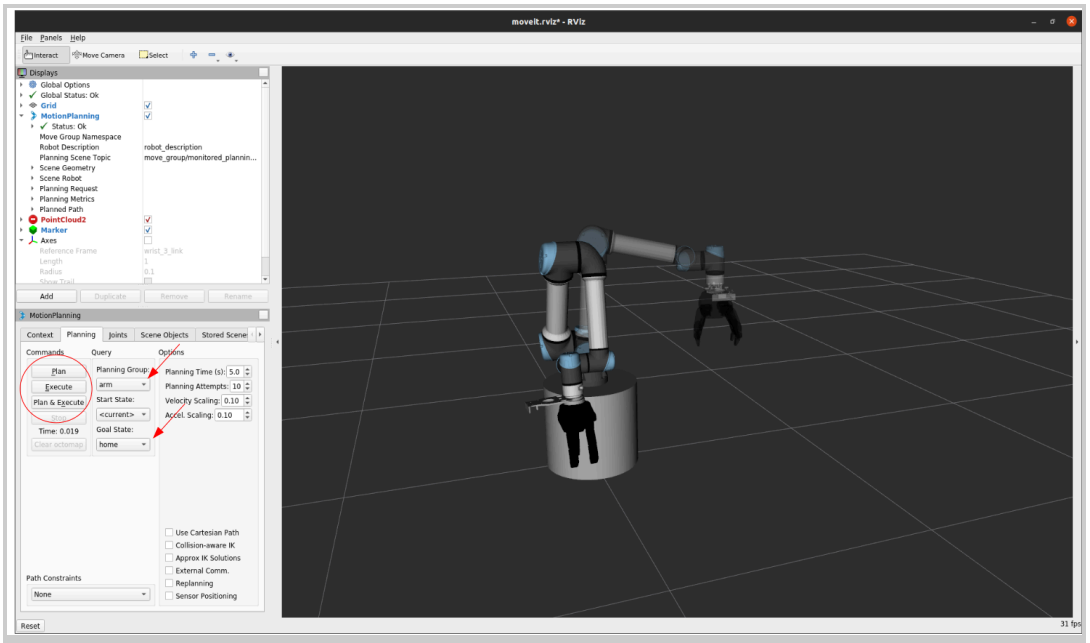

Fig. 4. Ventana RViz donde se muestra el brazo robótico UR5.

Conocer espacio de trabajo

En esta práctica, Moveit nos permite mover las articulaciones del brazo robótico y de la pinza mediante los sliders de la Figura 5 o de manera gráfica como se muestra en la Figura 6. De esta forma podemos conocer el espacio de trabajo del robot y saber cual es la distancia adecuada para colocar el robot en el escenario.

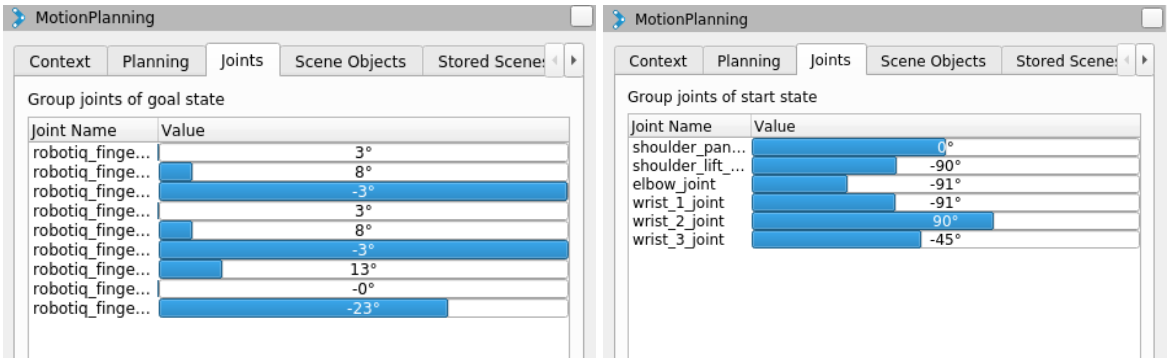

Fig. 5. Pestaña *Joints*, que muestra las posiciones articulares, dependiendo del grupo de planificación de la pestaña *Planning.* A la izquierda las del brazo y a la derecha de la pinza

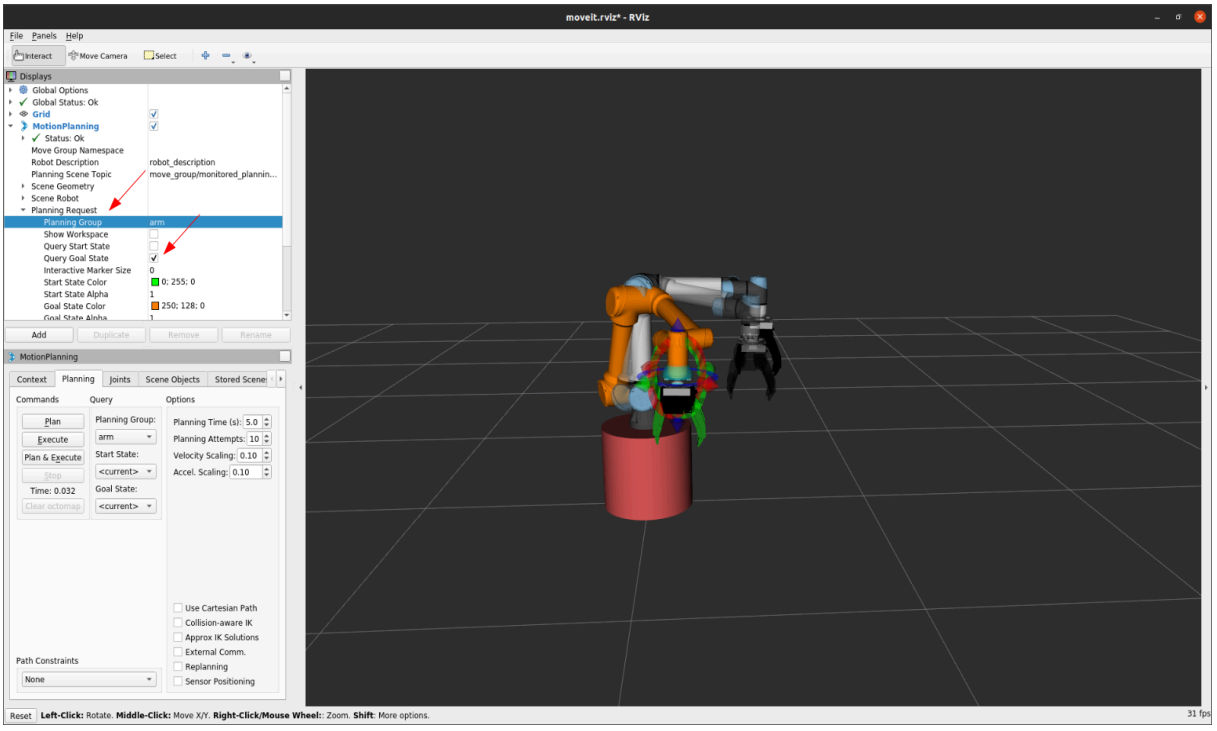

Fig. 6. Seleccionar una posición objetivo de manera gráfica.

Para realizar esta parte, en primer lugar habrá que elegir la posición del robot UR5. Ahora mismo, se carga en una posición lejana del objeto y no puede cogerlo. El objeto aparecerá en una posición aleatoria alrededor del punto **x = 5.0 , y = 3.5**, en un radio de 0.5 m. El objeto es un prisma rectangular con una orientación fija de 0°.

La posición del robot se ha de cambiar en el archivo `load_scene.launch` que hemos lanzado. En la siguiente línea (línea 5 del archivo `load_scene.launch`) se define la posición XYZ y la orientación *roll, pitch yaw* del brazo robótico.

```
<arg name="world_pose" default="-x 0 -y 0 -z 0 -R 0 -P 0 -Y 0" doc="Pose to spawn the ur5 robot at"/>
```

Una selección adecuada de coordenadas XYZ y de orientación RPY permitirán que el brazo robótica logre coger el objeto.

Localizar Objeto

Ahora que el brazo robótico UR5 se encuentra en una posición correcta para el agarre, debemos conocer la posición del objeto con respecto al brazo robótico. En esta sección utilizaremos el fichero "`object_localization.py`". Los comentarios `#TODO` mostrarán lo que se debe agregar en este *script*.

Este *script* permitirá localizar el objeto en el escenario con una cámara RGB-D. Mediante el filtrado por color de una imagen en formato HSV, se obtiene el centroide de cada objeto detectado y, conociendo la profundidad (Distancia cámara-objeto) y los parámetros intrínsecos de la cámara, se logra calcular las coordenadas XYZ de los objetos con respecto a la cámara.

Los cambios que se aconseja realizar son los siguientes:

- Agregar los topics a los que se debe suscribir el programa en la función: ros_node.
- Generar el filtro de color para objetos de color rojo basandose en el codigo de filtrado del robot Blue. Esto se debe hacer en la función: color_image_callback
- Calcular la máscara y el centroide del filtro generado en el paso anterior. Se pueden calcular los centroides mediante los momentos de la imagen (cv2.moments). Esto se debe hacer en la función: filter_img_objects
- Dado el centroide calculado anteriormente, se debe calcular la distancia Cámara-Objeto mediante la imagen de profundidad. Esto se debe modificar en la función: depth_image_callback.
- Dado el centroide en la máscara, la distancia cámara-objeto y la matriz de parámetros intrínsecos de la cámara, localizar el robot Blue y el objeto rojo. Para esto, en la función img_xyz se deben extraer los datos del foco (fx,fy) y centroide (cx,cy) de la cámara. Dado los parámetros intrínsecos de la cámara (fx,fy,cx,cy) y el centroide del objeto (px,py) calcular las coordenadas x,y,z del objeto.

Para ejecutar el nodo se debe hacer mediante:

```
rosrun robotica_inteligente object_localization.py
```

El script pública los topics `/filtered_image/object`, `/filtered_image/robot` y `/pose/array`. Estos publican las imágenes filtradas del objeto y del robot Blue (ver Figura 7), y la pose de cada uno de estos en el escenario con respecto a la cámara (ver Figura 8).

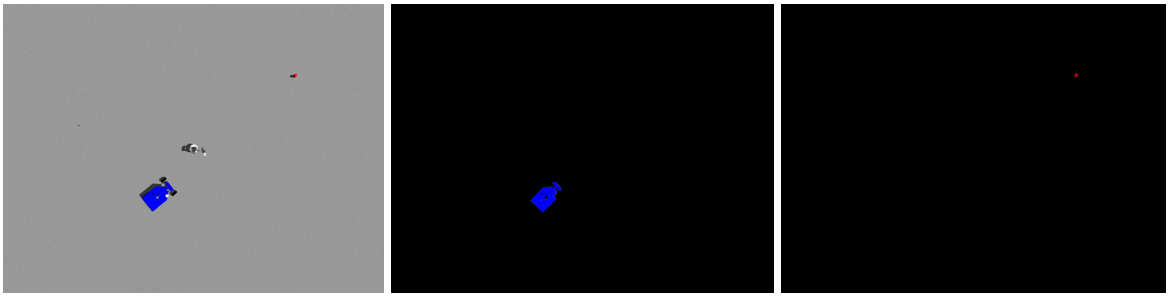

Fig. 7. Resultado esperado en las máscaras de segmentación por color.

```
XYZ object: 4.764, 3.439, 9.809
XYZ robot : -1.606, -2.098, 9.013
XYZ object: 4.764, 3.439, 9.809
XYZ robot : -1.606, -2.098, 9.013
```

Fig. 8. Resultado esperado de las coordenadas XYZ del robot Blue y el objeto rojo.

Agarre de Objeto

Después de tener localizado el objeto, se puede realizar el agarre del mismo con el brazo robótico UR5. Para ello es necesario modificar el archivo `move_robot_arm.py` del paquete `robotica_inteligente`.

En el nodo `move_robot_arm.py` se comunica con **MoveIt** para enviar las posiciones objetivo y mover el robot. Cuenta con funciones definidas para ir a posiciones articulares del brazo y la pinza, ir a posiciones cartesianas del efector final del brazo y añadir y enganchar el objeto a la pinza (para calcular colisiones durante la planificación). También está definido el *subscriber* para el *topic* en el que se envía la posición del objetivo, así como la función *callback*.

Cuando se quiere ir a una pose cartesiana, se debe enviar la posición cartesiana del efector final (la pinza) así como su orientación. Esta orientación está expresada en cuaterniones. Para profundizar en el uso de cuaterniones en ROS, se puede leer el tutorial básico. Hay dos maneras fáciles de establecer la orientación de la pinza. Por un lado, se puede pensar en ángulos de Euler y pasarlos a cuaterniones. Mediante,

```
geometry_msgs.msg.Quaternion(*quaternion_from_euler(X, Y, Z))
```

se construye el cuaternión para la rotación XYZ. Por otro lado, se puede guardar la orientación de la posición actual del robot, si queremos utilizar la misma para futuros movimientos:

```
self.move_arm.get_current_pose().pose.orientation
```

Los cambios que se aconseja realizar son los siguientes:

- Definir la posición global del robot, para calcular la posición del objeto respecto al UR5.
- Definir una posición inicial del robot (no se carga siempre en la misma posición articular), así cómo las posiciones de la pinza para abrirla y cerrarla.

- Definir las funciones necesarias para agarrar el objeto, es decir, acercarse a él, cerrar la pinza y levantarlo.

Hay que tener en cuenta que el planificador **MoveIt** puede no encontrar soluciones en algunas ocasiones. A veces es porque no puede llegar a la posición objetivo al encontrarse fuera de su área de trabajo, pero otras veces no encuentra una solución para llegar al objetivo y basta con volver a intentar planificar la trayectoria.

Cómo se ha explicado anteriormente, para ejecutar estos nodos se debe ejecutar en otras terminales:

```
rosrun robotica_inteligente move_robot_arm.py
rosrun robotica_inteligente object_localization.py
```

O crear un nuevo roslaunch para ejecutar ambos.

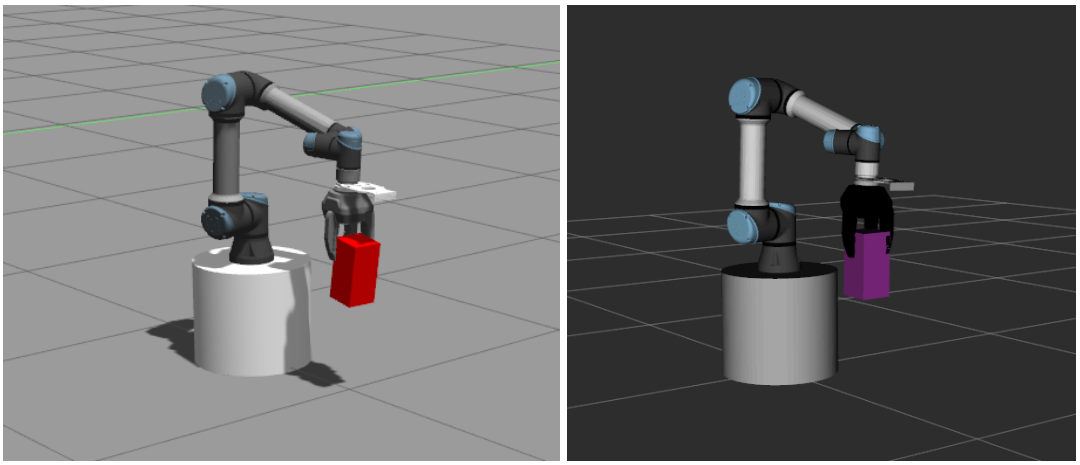

Fig. 9. Resultado esperado en Gazebo y RViz.

Segunda parte: mover el robot móvil hasta el UR5

El robot móvil utilizado en esta práctica es el robot móvil BLUE (roBot for Localization on Unstructured Environments) diseñado por el grupo de investigación AUROVA. Este robot tiene una geometría de tipo Ackermann y en esta práctica utiliza el sensor LiDAR 3D Velodyne VLP-16. El objetivo de esta parte es que el robot BLUE navegue hasta una posición cercana a la posición del robot UR5 establecida en la primera parte de la práctica. Esta posición debe ser lo suficientemente cercana para que el UR5 pueda dejar el objeto encima del robot móvil.

El robot móvil BLUE deberá moverse siguiendo una velocidad y ángulo deseados que se envían como mensaje tipo ackermann_msgs/AckermannDrive en el *topic* `/blue/ackermann_cmd`. Es decir, se controla con las variables *speed* y *steering_angle* de ese mensaje. Una vez establecido el objetivo global, se utilizará el algoritmo de planificación local *Naive-Valley-Path Obstacle Avoidance* para generar un trayectoria que evita los obstáculos, y se tendrá que analizar qué comandos de velocidad y ángulo de giro son adecuados para seguir esa trayectoria.

De esta manera, dividimos esta segunda parte en las secciones: Detección de Obstáculos y Navegación.

Detección de obstáculos

Para una correcta navegación el robot es necesario conocer el entorno por donde se está desplazando. En esta sección de la práctica se llevará a cabo la detección de obstáculos en el escenario donde el robot Blue va a navegar. El archivo a editar en esta sección es `obstacle_detection.py`.

Este script se suscribe a la nube de puntos dada por el sensor láser Velodyne VLP-16 que se encuentra sobre el robot BLUE. Esta nube de puntos se puede leer mediante el topic `/blue/velodyne_points`. Después, el programa detectará los posibles obstáculos en el escenario dependiendo de la altura de estos con respecto al piso. Es importante tomar en cuenta que el sensor láser está a un nivel superior del piso, por lo tanto se debe considerar la altura a la que está el sensor para poder identificar un obstáculo en el escenario.

Los cambios que se aconseja realizar son los siguientes:

- Colocar los valores adecuados de altura y radio para detectar los obstáculos que rodean al robot.
- En la función `filter_obstacles_function`, dada la altura de filtrado, agregar los puntos que la nube de puntos que superen esta altura en la variable `obstacles_points`.(ver Figura 10 puntos rojos)
- En la función `free_zone_function`, dada la altura y el radio alrededor del robot Blue, generar un anillo alrededor del robot de las zonas libre de obstáculos. Este anillo se añade en la variable `free_zone`. (ver Figura 10 puntos verdes)

Al realizar estos cambios, se podrán visualizar en RViz los topics `/obstacles` y `/free_zone` que contienen la nube de puntos de los pasos anteriores.

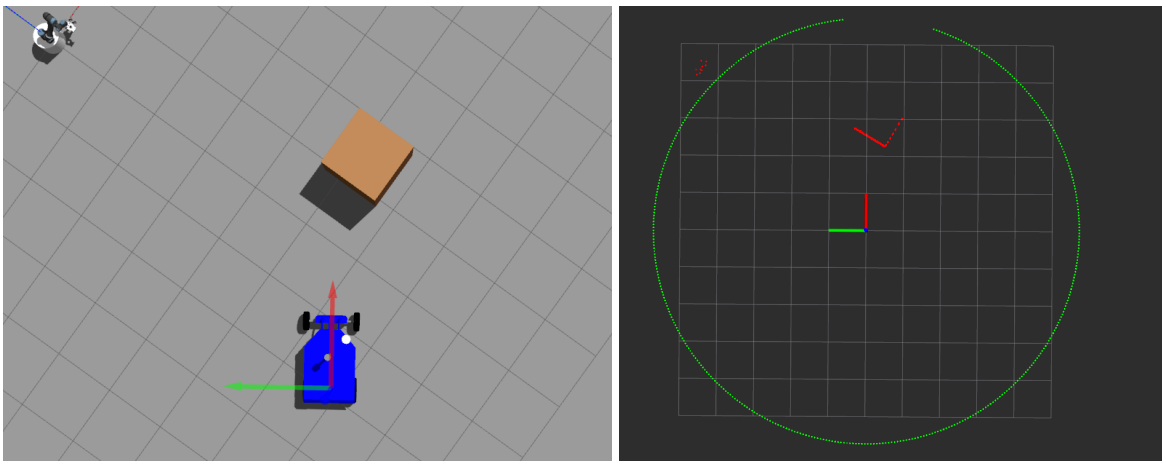

Fig 10. (Izquierda) Vista superior de la simulación de gazebo con un objeto dentro del radio del robot. (Derecha) Puntos de los topics `/obstacles` y `/free_zone` visualizados en RViz.

Navegación

Tras tener localizado el robot Blue e identificado los posibles obstáculos en el escenario, es posible la navegación del robot. El archivo que se debe editar en esta parte es `blue_navigation.py`.

El nodo `blue_navigation.py` se encarga de la planificación local del robot, y los cambios que se aconseja hacer en este son los siguientes:

- Definir el punto objetivo de esta tarea. También se pueden definir varios puntos para maniobrar y acercarse mejor al UR5.
- El *subscriber* de la posición del robot se suscribe actualmente a la pose real del robot proporcionada por Gazebo. Se debe adaptar para que no dependa de esta, si no que utilice la posición del robot captada por la cámara. Hay que tener en cuenta que se necesita la pose del robot, es decir, que además de las posiciones *x* e *y* de robot se necesita también el ángulo de dirección θ.
- Utilizar el modelo de bicicleta explicado en teoría para calcular la cinemática del robot y generar las trayectorias de este según las velocidades y ángulos de giro. Descartar las trayectorias en las que hay riesgo de colisiones, y evaluar las restantes para escoger la mejor en relación a la distancia y el ángulo hacia el objetivo.
- Por último, el robot debe ser capaz de adaptarse a obstáculos en el camino. Para testear este hecho, se debe añadir un obstáculo en Gazebo que se encuentre en la trayectoria del robot (ver Figura 11).

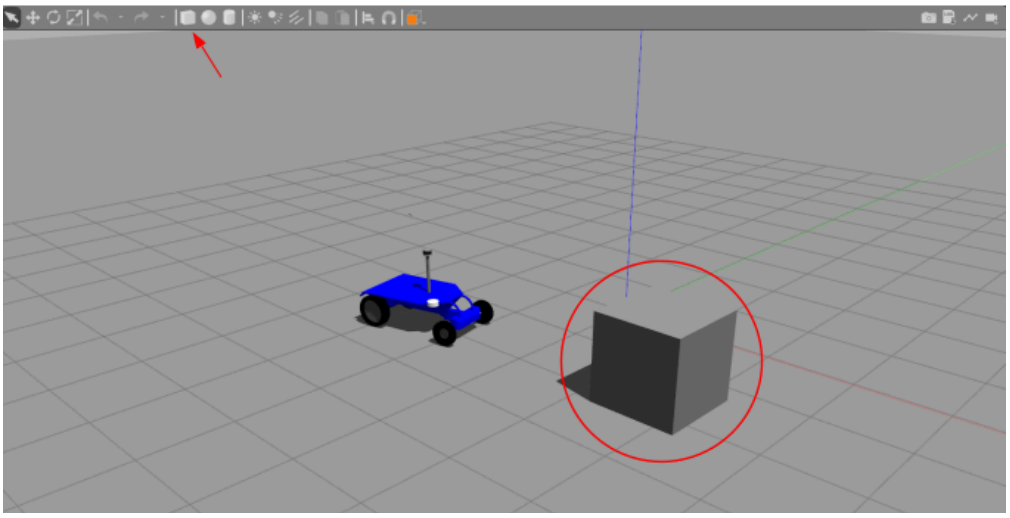

Fig. 11. Obstáculo añadido a la escena de Gazebo.

De la misma manera que la parte anterior, para ejecutar estos nodos se debe ejecutar en otras terminales:

```
rosrun robotica_inteligente blue_navigation.py
rosrun robotica_inteligente object_localization.py
rosrun robotica_inteligente obstacle_detection.py
```

También se puede utilizar el archivo `blue_navigation.launch`, que ejecuta esos nodos.

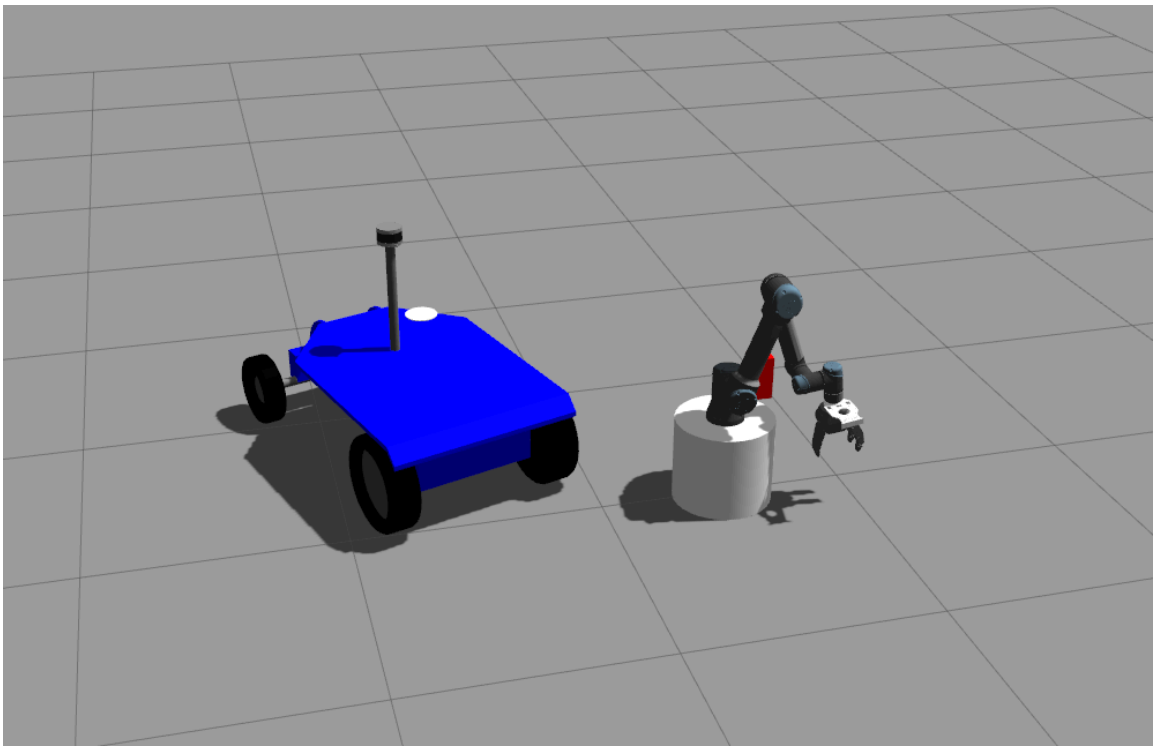

Fig. 12. El robot BLUE se ha acercado al UR5.

Tercera parte: coordinar ambas tareas

En esta última parte, se han de coordinar ambas tareas para que el robot UR5 deje el objeto encima del robot BLUE. Para ello,se deben implementar los siguientes procesos:

- Declarar nuevos *subscribers* y *publishers* para realizar la comunicación entre ambos robots, de manera que el BLUE sepa cuando el UR5 ha cogido el objeto y deba acercarse a este, y después que le comunique al UR5 que ha terminado su trayectoria y puede dejar el objeto.
  La declaración de *subscribers* y *publishers* está explicada en la Introducción a ROS de este documento, en la sección Topics, o podéis mirar el tutorial de ROS. Para esta comunicación es suficiente con utilizar alguno de los mensajes estándar std_msgs.
- Definir las funciones necesarias para que el UR5 deje el objeto en el robot BLUE. Si el robot BLUE no está suficientemente cerca del brazo robot UR5, habrá que reconsiderar los puntos objetivos de la segunda parte para poder dejar el objeto. Hay que tener en cuenta que el robot BLUE tiene más espacio en la parte trasera de este.

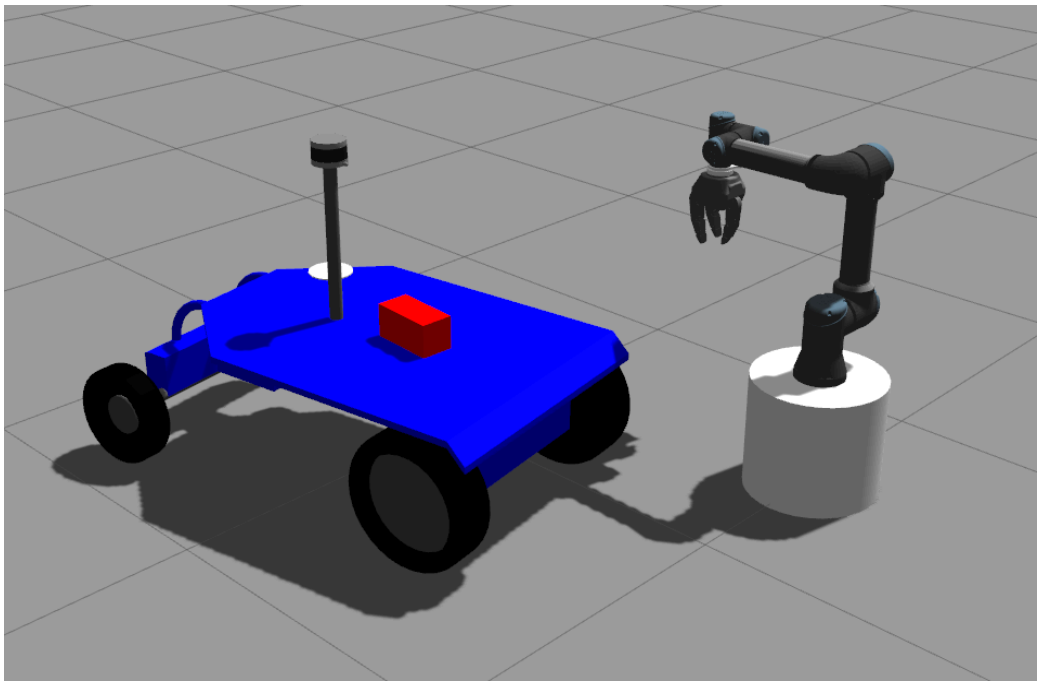

Fig. 13. El objeto ha sido depositado en el BLUE.