Manual de laboratorio del PhantomX Reactor Arm

Proyecto de TFG: *Control y simulación en ROS de un PhantomX Reactor Arm en cooperación con un TurtleBot2*

Autor: Daniel Lozano Moreno

Tutora: María del Pilar Arqués Corrales

Grado en Ingeniería Robótica

Escuela Politécnica Superior

Universitat d'Alacant
Universidad de Alicante

Alicante, Junio 2022

RESUMEN

Se presenta la documentación sobre el manual de laboratorio para el uso de un PhantomX Reactor Arm instalado en los TurtleBot2 que proporciona el laboratorio de robótica de la Escuela Politécnica Superior de la Universidad de Alicante. Este documento fue realizado durante la elaboración del proyecto de Trabajo de Fin de Grado *Control y Simulación de un PhantomX Reactor Arm en cooperación con un TurtleBot2*, por Daniel Lozano Moreno en el Grado en Ingeniería Robótica de la Escuela Politécnica de Superior de la Universidad de Alicante. La motivación para realizar esta guía de uso reside en compartir la información estudiada sobre el PhantomX Reactor Arm, su configuración y el software de control programado para el proyecto. Este proyecto se llevó a cabo debido a que los TurtleBots2 que posee el laboratorio de robótica de la universidad llevan instalados un PhantomX Reactor Arm, brazo que estaba sin configurar y del que existe poca documentación para ello. Este documento explicará cómo está instalado, su configuración y además se proporciona un código para la simulación en Gazebo y una arquitectura de control propia para facilitar su programación.

PhantomX Reactor Arm

El laboratorio de robótica de la Escuela Politécnica Superior de la Universidad de Alicante cuenta con varios robots TurtleBot2 para la enseñanza. Estos TurtleBots llevan instalados en la parte superior un PhantomX Reactor Arm.

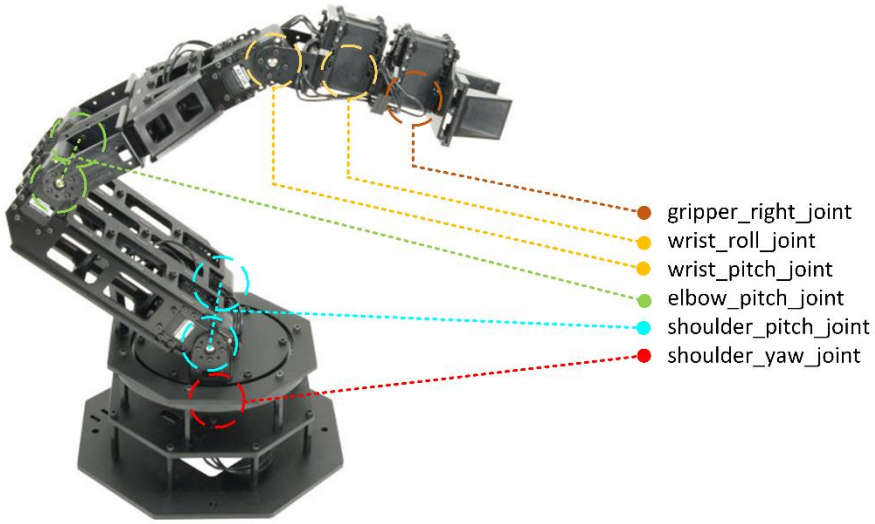
Un PhantomX Reactor Arm es un manipulador serie con 5 Grados de Libertad (GDL) con una pinza paralela de un solo motor como elemento terminal. Debido a su cadena cinemática abierta, su espacio de trabajo se extiende hasta el alcance de su efector final. El brazo se controla mediante servomotores Dynamixel modelo AX-12A, los actuadores más avanzados a nivel de robótica personal. Éstos son comandados y programados mediante la placa de desarrollo software ArbotiX-M, placa Arduino diseñada específicamente para el control de actuadores Dynamixel y programada en el lenguaje C.

Tabla 0.1 PhantomX Reactor Arm

Características	
Peso	1430g
Alcance vertical	55.5cm
Alcance horizontal	42cm
Fuerza	30cm/200G 20cm/400g 10cm/600g
Fuerza de agarre	500g
Fuerza de levantamiento de muñeca	150g
Grados de giro en la base	300°
Diámetro de giro	80cm
Número de actuadores	8
Clase de actuadores	AX-12A
Controladora	ArbotiX

Característcas del PhantomX Reactor Arm

El brazo está formado por 5 articulaciones actuadas por 7 servo-motores Dynamixel, siendo dos de ellas actuadas por dos motores de forma mímica e inversa. El elemento terminal es una pinza de dedos paralelos accionada por un servo-motor del mismo modelo. Cada motor tiene su propio número de identificación (id) para diferenciarlos.



Articuaciones del PhantomX Reactor Arm

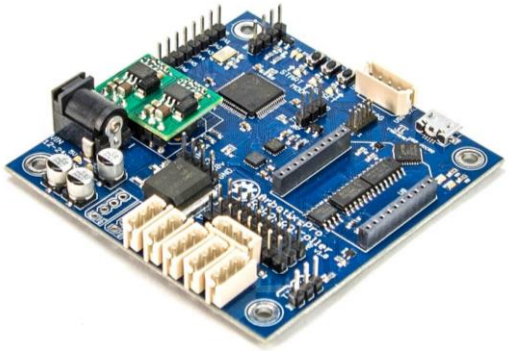
Grupo	Etiqueta	Id	Límites
Brazo	shoulder_yaw_joint	1	[-2.5, 2.5]
	shoulder_pitch_joint	2 y 3	[-1.7, 1.7]
	elbow_pitch_joint	4 y 5	[-2.5, 2.5]
	wrist_pitch_joint	6	[-1.7, 1.7]
	wrist_roll_joint	7	[-2.5, 2.5]
Pinza	gripper_right_joint	8	[0.0, 0.03]

Id de los motores del PhantomX Reactor Arm

Los servo-motores Dynamixel del modelo AX-12A [1] tienen un rango de movimiento de 300 grados. Incorporan un led informativo para conocer el estado del robot. El led tendrá un color rojo intermitente cuando el motor sufra sobrecalentamiento o sobrecarga, por ejemplo, si se comanda a una posición fuera de sus 300 grados de movilidad. Tienen una conexión física TTL y para comandarlos existen dos dispositivos:

- USB2DYNAMIXEL
- Placa microcontroladora ArbotiX-M

El PhantomX Reactor incorpora la placa ArbotiX-M, placa Arduino con microcontrolador para programar tareas con sets de servo-motores Dynamixel. El microcontrolador se programa a través de Arduino IDE con los drivers de *arbotix-master* [2], en caso de no funcionar, probar con los drivers *Arbotix-arduino-1-6*. Para conectar esta placa a cualquier CPU se necesita un cable FTDI (Future Tecknology Device International) TTL-USB. La CPU necesita tener instalado los drivers de FTDI para detectar el cable [3]. Los drivers de arbotix-master incorpora algún código para probar el PhantomX Reactor.



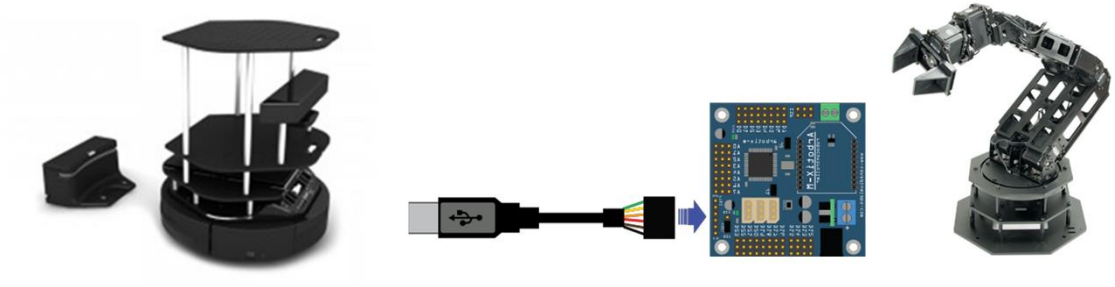
Placa ArbotiX-M

Contenido

1 Conexión TurtleBot2 – PhantomX Reactor Arm

Todo lo explicado a continuación debería estar ya realizado en los robots del laboratorio. En caso contrario, se recomienda seguir los pasos encontrados en [4]. A continuación, se explica el funcionamiento de la conexión.

El PhantomX Reactor Arm del laboratorio se conecta a la CPU del TurtleBot2 mediante un cable FTDI TTL-USB. La CPU ya tiene instalado los drivers para FTDI, la base Kobuki también se conecta a la CPU a través de uno de estos cables.



Para permitir la comunicación entre la CPU, que funciona con el sistema operativo ROS, y con ArbotiX, se necesita a ambos lados sus respectivos drivers de comunicación. Para el ArbotiX-M, se programará con el código *arbotix-master/ros* o *Arbotix-arduino-1-6/ros* la placa a través de Arduino IDE. Para el TurtleBot, se necesitará añadir el paquete *arbotix_ros* [5] al espacio de trabajo donde residan los launchers para usar el brazo. El documento *Guía de uso del paquete arbotix_ros* detalla cómo configurar la placa ArbotiX-M a través del paquete *arbotix_ros* para el control de los Dynamixel instalados.

Para que ROS encuentre la entrada USB del ArbotiX-M, se debe crear una udev rule para darle una etiqueta identificatoria a la entrada. En el caso de los TurtleBots del laboratorio, la etiqueta que identifica la entrada del ArbotiX-M es "*/dev/ttyUSB_REACTOR*".

Con todos estos pasos realizados, ya puede controlarse el PhantomX Reactor con el TurtleBot2 a través del paquete *arbotix_ros*.

La compañía Robotnik Automation, especializada en robots móviles y manipuladores de servicio y proyectos I+D (Investigación y Desarrollo), proporciona de forma pública en GitHub el meta-paquete ros *phantomx_reactor_arm* [4]. Ofrece los paquetes necesarios para un control básico de acción directa al brazo desde ROS. Incluye los parámetros de configuración del ArbotiX-M para el caso de la instalación de los motores Dynamixel del brazo. Los paquetes que ofrece son:

- *phantomx_reactor_arm_description*: define el modelo URDF del robot con los modelos geométricos reales de las piezas (definidos mediante archivos STL), los atributos dinámicos de los eslabones y los cinemáticos de las articulaciones.
- *phantomx_reactor_arm_controller*: ofrece un nodo de comunicación con los drivers de *arbotix_ros* y crea una instancia a la clase ArbotiX-M con los parámetros de configuración del brazo real. Permite comandar mediante topics los motores del brazo.
- *phantomx_reactor_arm_moveit_config*: configuración MoveIt para usar sus funcionalidades.

El lanzador ***arbotix_phantomx_reactor_arm_wrist.launch***, incluido en el paquete *phantomx_reactor_arm_controller*, establece la conexión con el brazo PhantomX Reactor Arm y crea los topics para comandar a cada una de sus articulaciones. Para ello, carga los parámetros de configuración del ArbotiX-M ubicados en el fichero ***arbotix_config_phantomx_wrist.yaml*** en el paquete *phantomx_reactor_arm_controller*.

El meta-paquete *phantomx_reactor_*arm ya viene instalado en el TurtleBot2 del laboratorio, permitiendo controlar el brazo mediante control en posición. A continuación, se proporcionarán los directorios de ubicación de los archivos necesarios para el control del brazo.

El fichero *arbotix_phantomx_reactor_arm_wrist.launch* se ubica en el directorio:

./home/turtlebot/sources/turtlebot/turtlebot_bringup/launch/includes/arms

El fichero *arbotix_config_phantomx_wrist.yaml* se ubica en el directorio:

./home/turtlebot/sources/turtlebot/turtlebot_bringup/config

Los paquetes de *arbotix_ros* se ubican en el directorio:

/opt/ros/indigo/lib/python2.7/dist-packages

El resto de paquetes de *phantomx_reactor_arm* no son necesarios para el control del brazo.

2 Puesta en marcha del PhantomX Reactor Arm

Para poner en marcha el PhantomX Reactor Arm, deben ejecutarse los pasos que se describirán a continuación.

Ejecutar el bringup TurtleBot:

```
>> roslaunch turtlebot_bringup minimal.launch
```

Ejecutar el bringup del PhantomX Reactor Arm:

```
>> roslaunch turtlebot_bringup arbotix_phantomx_reactor_arm_wrist.launch
```

¡Ya puede controlarse el brazo!

Al ejecutar este último comando, debería cargarse el nodo de control del brazo. Deberían aparecer los siguientes mensajes por la terminal:

```
>> roslaunch turtlebot_bringup arbotix_phantomx_reactor_arm_wrist.launch

SUMMARY
========

PARAMETERS
* /arbotix/joints/arm_1_joint/id: 1
* /arbotix/joints/arm_1_joint/max_speed: 75.0
* /arbotix/joints/arm_2_1_joint/id: 3
* /arbotix/joints/arm_2_1_joint/max_speed: 75.0
* /arbotix/joints/arm_2_joint/id: 2
* /arbotix/joints/arm_2_joint/max_speed: 75.0 * /arbotix/joints/arm_3_1_joint/id: 5
* /arbotix/joints/arm_3_1_joint/max_speed: 75.0
* /arbotix/joints/arm_3_joint/id: 4
* /arbotix/joints/arm_3_joint/max_speed: 75.0
* /arbotix/joints/arm_4_joint/id: 6
* /arbotix/joints/arm_4_joint/max_speed: 75.0 23 Kobuki Turtlebot II. User's Manual
* /arbotix/joints/arm_5_joint/id: 7
* /arbotix/joints/arm_5_joint/max_speed: 75.0
* /arbotix/joints/arm_gripper_joint/id: 8
* /arbotix/joints/arm_gripper_joint/max_angle: 0
* /arbotix/joints/arm_gripper_joint/max_speed: 114.0
* /arbotix/joints/arm_gripper_joint/min_angle: -90.0
* /arbotix/joints/arm_gripper_joint/range: 180
* /arbotix/port: /dev/ttyUSB0
* /arbotix/rate: 100
* /robot_description: <?xml version="1….
* /rosdistro: kinetic
* /rosversion: 1.11.21

NODES
/
arbotix (arbotix_python/arbotix_driver) phantomx_reactor_controller
(phantomx_reactor_arm_controller/phantomx_reactor_parallel_motor_j oints.py)
robot_state_publisher (robot_state_publisher/state_publisher) rviz (rviz/rviz)

ROS_MASTER_URI=http://robotnik-GE70:11311

core service [/rosout] found
process[arbotix-1]: started with pid [16444]
process[phantomx_reactor_controller-2]: started with pid [16445]
process[robot_state_publisher-3]: started with pid [16446]
process[rviz-4]: started with pid [16447]
```

```
[INFO] [WallTime: 1496152050.354436] Started ArbotiX connection on port
/dev/ttyUSB_REACTOR.
[INFO] [WallTime: 1496152050.481723] ArbotiX connected.
```

El software de control cargado permite un control en posición de las articulaciones del brazo, esto quiere decir que, cuando se comande a una de ellas, el controlador de la articulación intentará disminuir el error habido entre la posición comandada y el estado actual de la articulación, ejecutando así una trayectoria lineal. Al ejecutar una trayectoria lineal, habrá discontinuidad en las derivadas de la posición al no existir una planificación de éstas, provocando un movimiento brusco durante toda la trayectoria.

Es muy importante que la posición comandada a la articulación no salga de los límites de movilidad de la articulación (pag 4), sino, el motor se bloqueará por sobrecalentamiento y acabará apagándose, informando sobre un error a través de un led rojo intermitente. Por ahora, la única manera para reiniciar el motor es reiniciar el TurtleBot.

Los topics para publicar los comandos son los siguientes:

```
/phantomx_reactor_controller/shoulder_yaw_joint/command
/phantomx_reactor_controller/shoulder_pitch_joint/command
/phantomx_reactor_controller/elbow_pitch_joint/command
/phantomx_reactor_controller/wrist_pitch_joint/command
/phantomx_reactor_controller/wrist_roll_joint/command
/phantomx_reactor_controller/gripper_prismatic_joint/command
```

La GUI *rqt_publisher* carga una interfaz para comandar de manera simple a los topics cargados en ROS. Para cargarlo, ejecutar el siguiente comando y abrir la pestaña *Message Publisher*. Esto solo será posible si ha establecido una conexión Master/Slave entre el TurtleBot2 del laboratorio (slave) y el ordenador con el que se controla (master). Usando una conexión cliente/servidor no lo permite ya que no se pueden cargar GUIs.

```
>> rosrun rqt_publisher rqt_publisher
```

Entre los objetivos del proyecto de TFG *Control y simulación en ROS de un PhantomX Reactor Arm en cooperación con un Turtlebot2* se encuentra el de configurar el ArbotiX-M del brazo para que acepte comandos de trayectorias. Se explicará en el apartado 3.

El URDF del brazo no es cargado en este lanzador. Si se abre el fichero `arbotix_phantomx_reactor_arm_wrist.launch` se puede comprobar que el comando para lanzar la descripción del modelo está comentada:

```
<!--param name="robot_description" command="$(find xacro)/xacro.py '$(find
phantomx_reactor_arm_description)/robots/phantomx_reactor_arm_wrist.urdf.xacro'" /-->
```

Esto se debe a que el URDF comparte nombres con el URDF del TurtleBot2, provocando conflictos y errores a la hora de cargarlos. Esto impide aprovechar las funciones que otorgan los modelos URDF (por ejemplo, la visualización del brazo en rviz) y no permite una buena cooperación entre el TurtleBot2 y el brazo para la programación de tareas.

Por esta razón, en este proyecto se ha elaborado el paquete ***turtlebot_arm_description***. Este paquete incluye un modelo URDF del brazo compatible con el URDF del turtlebot (*phantomx_reactor.xacro*). También ofrece el modelo URDF del turtlebot con el brazo

instalado (turtlebot_arm). Las descripciones de estos modelos pueden ser cargados mediante los comandos:

```
>> roslaunch turtlebot_arm_description phantomx_reactor_load_description.launch
>> roslaunch turtlebot_arm_description turtlebot_arm_load_description.launch
```

Puede editarse el fichero `arbotix_phantomx_reactor_arm_wrist.launch` con la ruta del modelo phantomx_reactor.xacro o crear y ejecutar un nuevo workspace con los paquetes *arbotix_ros* y *turtlebot_arm_description*.

Los nuevos modelos URDF pueden cargarse en rviz mediante los comandos:

```
>> roslaunch turtlebot_arm_description phantomx_reactor_rviz_demo.launch
>> roslaunch turtlebot_arm_description turtlebot_arm_rviz_demo.launch
```

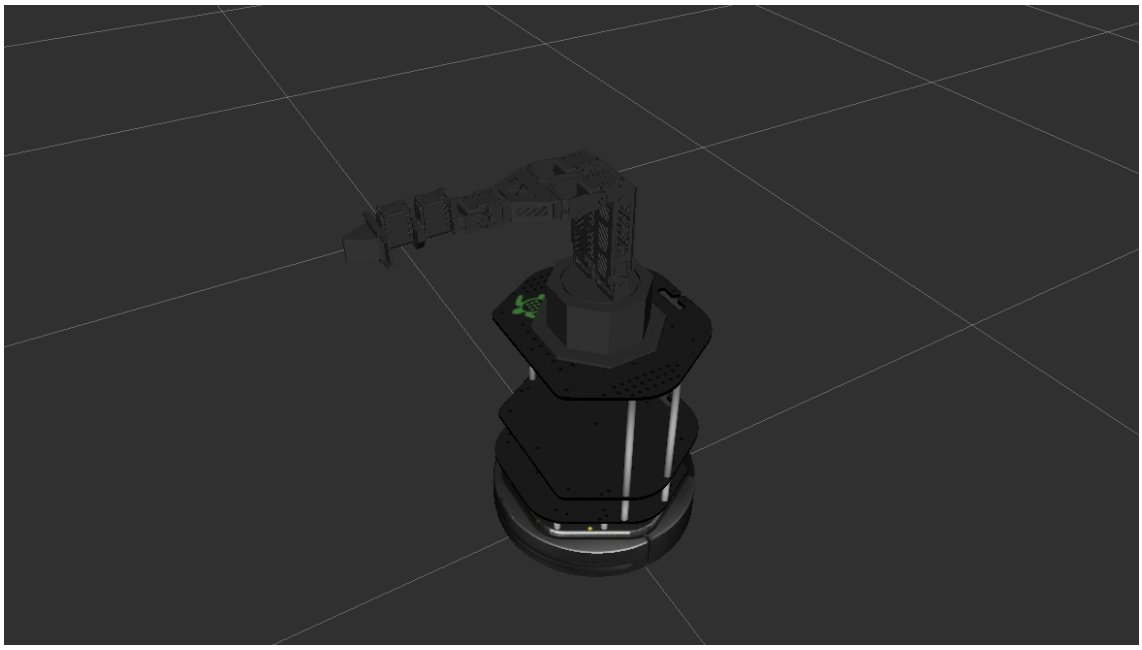


Figura 2.1 Carga del modelo URDF turtlebot_arm.xacro en rviz

3 Configuración del ArbotiX-M para un control de seguimiento de trayectorias

El ArbotiX-m del PhantomX Reactor se configura a través del paquete *arbotix_ros*. El documento *Guía de uso del paquete arbotix_ros* explica de forma detallada como configurar el ArbotiX-M para que funcione como se desea. En este capítulo se explicará como configurarlo para que acepte comandos de trayectoria.

3.1 Configuración

La conexión con un ArbotiX-M se establece ejecutando el nodo *arbotix_driver*, ubicado en el paquete *arbotix_ros/arbotix_python*. Este nodo debe ejecutarse junto a los parámetros de configuración deseados. Si los parámetros de configuración se encuentran en un fichero YAML, el nodo se carga en un launcher de la siguiente manera:

```
<!-- Load arbotix_driver -->
<node name="arbotix" pkg="arbotix_python" type="arbotix_driver" output="screen">
    <!-- Load params in ROS parameter server -->
    <rosparam file="$(find package_name)/path/to/file/config.yaml" command="load" />
</node>
```

Los parámetros de configuración que deberían definirse son:

- *port*: puerta USB a la que está conectado el ArbotiX-M.
- *rate*: frecuencia de muestreo.
- *joints*: lista de accionamientos, uno por cada motor a controlar.
- *controllers*: controladores para los accionamientos.

La puerta USB es la udev rule de la que se ha hablado en el apartado 1. En el caso de los TurtleBots del laboratorio, se debe definir este parámetro como "*/dev/ttyUSB_REACTOR".*

El parámetro *joints* es un diccionario con las etiquetas de los motores a controlar. A cada motor se le debe indicar su id.

En cuanto al parámetro *controllers*, especifica los controladores del robot. Se debe crear un diccionario con los grupos de control, en este caso, uno para el brazo (arm_controller) y otro para la pinza (grip_controller). A cada grupo se le especifica el tipo de controlador a utilizar, si se desea comandar trayectorias, se definirá el tipo "*follow_controller*". También debe definirse una lista con las articulaciones incluidas en cada grupo y el nombre del topic de la joint_trajectory_action, por ejemplo "arm_controller/follow_joint_trajectory".

En el TurtleBot2, el nodo es lanzado en el fichero *arbotix_phantomx_reactor_arm_wrist.launch* con la configuración definida en *arbotix_config_phantomx_wrist.yaml*. Si se abre este último fichero, se puede comprobar que la configuración es la siguiente:

```
# Config file for PhantomX Reactor
port: /dev/ttyUSB_REACTOR
rate: 100
joints: {
  arm_shoulder_yaw_joint: {id: 1, max_speed: 50.0},
  arm_shoulder_pitch_joint: {id: 2, max_speed: 50.0},
  arm_shoulder_pitch_mimic_joint: {id: 3, max_speed: 50.0},
  arm_elbow_pitch_joint: {id: 5, max_speed: 50.0},
  arm_elbow_pitch_mimic_joint: {id: 4, max_speed: 50.0},
  arm_wrist_pitch_joint: {id: 6, max_speed: 50.0, invert: true},
  arm_wrist_roll_joint: {id: 7, max_speed: 50.0},
  arm_gripper_revolute_joint: {id: 8, max_speed: 100.0, range: 180, min_angle: -90.0,
max_angle: 0},
}
```

Como se muestra, el parámetro *controllers* no está definido, provocando la acción directa de los comandos sobre los motores y dando lugar a un control en posición. Para conseguir un control de seguimiento de trayectorias, añadir al anterior código el siguiente fragmento:

```
14. controllers: {
15.   arm_controller: {
16.     type: 'follow_controller',
17.     rate: 100,
18.     joints: [
19.             'arm_shoulder_yaw_joint',
20.             'arm_shoulder_pitch_joint',
21.             'arm_shoulder_pitch_mimic_joint',
22.             'arm_elbow_pitch_joint',
23.             'arm_elbow_pitch_mimic_joint',
24.             'arm_wrist_pitch_joint',
25.             'arm_wrist_roll_joint'
26.             ]
27.    action_name: 'arm_controller/follow_joint_trajectory'
28.   }
29.   grip_controller: {
30.     type: 'follow_controller',
31.     rate: 100,
32.     joints: [
33.             'arm_gripper_revolute_joint'
34.             ]
35.     action_name: 'grip_controller/follow_joint_trajectory'
36.   }
37. }
```

o crear un workspace en el que esté incluido el paquete *arbotix_ros* y un lanzador para cargar el nodo *arbotix_driver* con la anterior configuración.

Si la configuración de los controladores es correcta, al cargar el *arbotix_driver* deberían aparecer en la terminal los mismos mensajes mostrados en el capítulo 2 con la adición de un mensaje como el siguiente por cada controlador:

```
[INFO] [WallTime: 1652281529.349479] Started FollowController (arm_controller).
Joints: ['arm_wrist_roll_joint', 'arm_elbow_pitch_mimic_joint',
'arm_elbow_pitch_joint', 'arm_shoulder_pitch_mimic_joint',
'arm_shoulder_pitch_joint', 'arm_shoulder_yaw_joint', 'arm_wrist_pitch_joint'] on C1
```

En caso contrario, aparecerá un error de carga de los controladores.

Con la configuración anterior, al ejecutar el comando *rostopic list* deberían aparecer los siguientes topics:

```
>> rostopic list

/arm_controller/command
/arm_controller/follow_joint_trajectory/cancel
/arm_controller/follow_joint_trajectory/feedback
/arm_controller/follow_joint_trajectory/goal
/arm_controller/follow_joint_trajectory/result
/arm_controller/follow_joint_trajectory/status
/diagnostics
/elbow_pitch_joint/command
/elbow_pitch_mimic_joint/command
/grip_controller/command
/grip_controller/follow_joint_trajectory/cancel
/grip_controller/follow_joint_trajectory/feedback
/grip_controller/follow_joint_trajectory/goal
/grip_controller/follow_joint_trajectory/result
/grip_controller/follow_joint_trajectory/status
/gripper_revolute_joint/command
```

```
/joint_states
/rosout
/rosout_agg
/shoulder_pitch_joint/command
/shoulder_pitch_mimic_joint/command
/shoulder_yaw_joint/command
/wrist_pitch_joint/command
/wrist_roll_joint/command
```

Los topics con sintaxis */[joint_names]/command* son topics creados por defecto por el nodo *arbotix_driver*, independientemente de si han sido definidos controladores en la configuración del ArbotiX-M. Estos comandos permiten comandar directamente a los motores Dynamixel, los cuales ejecutarán un control en posición. No es aconsejable usar estos topics, ya que, al comandar a los motores individualmente, en los casos en los que una articulación esté formada por varios motores habrá un conflicto entre ellos provocando la sobrecarga y apagado de todos los motores de la articulación. Se necesitaría una arquitectura de control como la que ofrece *arbotix_phantomx_reactor_arm_wrist.launch* para comandar a los motores mímicos al mismo tiempo y de forma invertida.

Si se ejecuta *rosservice list*¸ deberían aparecer los siguientes servicios:

```
>> rosservice list

/arbotix/SetupAnalogIn
/arbotix/SetupDigitalIn
/arbotix/SetupDigitalOut
/arbotix/get_loggers
/arbotix/set_logger_level
/arm_elbow_pitch_joint/enable
/arm_elbow_pitch_joint/relax
/arm_elbow_pitch_joint/set_speed
/arm_elbow_pitch_mimic_joint/enable
/arm_elbow_pitch_mimic_joint/relax
/arm_elbow_pitch_mimic_joint/set_speed
/arm_gripper_revolute_joint/enable
/arm_gripper_revolute_joint/relax
/arm_gripper_revolute_joint/set_speed
/arm_shoulder_pitch_joint/enable
/arm_shoulder_pitch_joint/relax
/arm_shoulder_pitch_joint/set_speed
/arm_shoulder_pitch_mimic_joint/enable
/arm_shoulder_pitch_mimic_joint/relax
/arm_shoulder_pitch_mimic_joint/set_speed
/arm_shoulder_yaw_joint/enable
/arm_shoulder_yaw_joint/relax
/arm_shoulder_yaw_joint/set_speed
/arm_wrist_pitch_joint/enable
/arm_wrist_pitch_joint/relax
/arm_wrist_pitch_joint/set_speed
/arm_wrist_roll_joint/enable
/arm_wrist_roll_joint/relax
/arm_wrist_roll_joint/set_speed
```

Cada motor tiene tres servicios:

- Enable: habilita el motor
- Relax: deshabilita el motor

- Set_speed: establece la velocidad del motor

Estos servicios permiten reiniciar el motor cuando ha quedado inhabilitado por sobrecarga. Sin embargo, estos servicios están inoperativos. El paquete *arbotix_msgs*¸ donde se encuentran los mensajes de los servicios, debería estar en la ruta /opt/ros/indigo/lib/python2.7/dist-packages para llamarse de forma global como lo está *arbotix_python* (por eso no es necesario definir la ruta de *arbotix_driver*). Aunque no ha sido probado, los servicios podrían quedar operativos si se añade *arbotix_msgs* a este directorio o se podría crear un nuevo workspace donde se incluya *arbotix_ros* y ejecutar en él el lanzador de *arbotix_driver*.

3.2 Ejecución de trayectorias

Los controladores follow_controller están diseñados para aceptar comandos de tipo **trajectory_msgs/JointTrajectory**, mensaje definido mediante la siguiente estructura de datos:

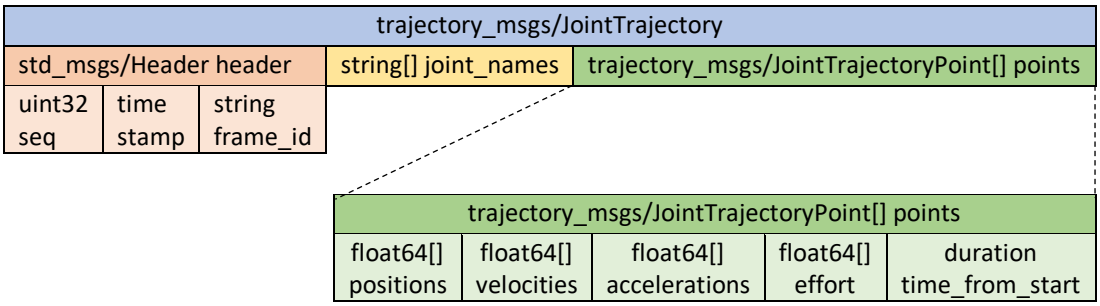


Figura 3.1 Estructura de datos del mensaje trajectory_msgs/JointTrajectory

Field	Type	Description
header	std_msgs/Header	Cabecera del mensaje
seq	uint32	Contador mensaje
stamp	time	Timestamp de publicación
frame_id	string	Identificador del menaje
joint_names	string[]	Articulaciones a las que se va a efectuar la trayectoria
points	trajectory_msgs/JointTrajectoryPoint[]	Waypoints de la trayectoria
positions	float64[]	Posición de cada articulación en el waypoint
velocities	float64[]	Velocidad de cada articulación en el waypoint
accelerations	float64[]	Aceleración de cada articulación en el waypoint
effort	float64[]	Torque de cada articulación en el waypoint
time_from_start	duration	Timestamp en el que se debe ejecutar el waypoint

Figura 3.2 Campos del mensaje trajectory_msgs/JointTrajectory

JointTrajectory representa una trayectoria. Está compuesta por el vector joint_names, que incluye los nombres de todas las articulaciones a las que afecta esta trayectoria, y que debe coincidir con todas los nombres de las articulaciones del grupo controlado por el controlador al que va a ir comandado; y por el vector points, que incluye todos los waypoints de la treyectoria. Cada waypoint reflejará el estado de todas las articulaciones en un instante de tiempo.

Es común en ROS que cuando se cargue un controlador que acepte mensajes de tipo joint_trajectory se cree el topic para publicar los mensajes:

```
/controller_name/command
```

El controlador ejecuta esta trayectoria consiguiendo así el movimiento planificado de las articulaciones. En ocasiones también se crea el topic para conocer el estado del controlador en

cada iteración de control (estados actuales de las articulaciones, deseados y errores). Lamentablemente, este topic solo no se crea con el controlador *follow_controller*:

```
/controller_name/state
```

También es común en ROS que cuando se cargue un controlador que acepte mensajes de tipo jointTrajectory, a su vez se cargue la acción **joint_trajectory_action**, nodo que lleva un seguimiento de la trayectoria para monitorear su estado. Cuando se carga, se crean los siguientes topics:

```
/controller_name/follow_joint_trajectory/cancel
/controller_name/follow_joint_trajectory/feedback
/controller_name/follow_joint_trajectory/goal
/controller_name/follow_joint_trajectory/result
/controller_name/follow_joint_trajectory/status
```

/goal describe la trayectoria que el robot debe seguir; /feedback describe el progreso de seguimiento de trayectoria; /cancel permite cancelar la trayectoria que está siendo ejecutada; y /status información sobre el goal. Para más información sobre estos topics, ver [6].

4 Simulación del PhantomX Reactor Arm en Gazebo

El paquete *turtlebot_arm_description*, elaborado para este proyecto, ofrece los modelos URDF del PhantomX Reactor Arm de forma individual (*phantomx_reactor.xacro*) y conjunta con el TurtleBot2 (*turtlebot_arm.xacro*). Para la simulación en Gazebo, se han incluido a los modelos URDF los modelos de los motores y transmisores del brazo, los cuales funcionan con interfaz hardware de posición (ya que simulan un servo-motor).

A los modelos robóticos se les ha añadido los plugins necesarios para la simulación en gazebo. Es importante seguir los siguientes pasos para preparar el modelo URDF para su simulación en Gazebo.

Los modelos URDF necesitan los siguientes plugins:

- gazebo_ros_control: este plugin permite el uso de controladores de *controller_manager* en ROS. Lo más seguro es que tu máquina virtual ya tenga instalado este plugin y no sea necesario preocuparse. En caso contrario, instalar el plugin, ir al modelo del robot y cambiar el *filename* por el directorio donde está ubicado el fichero *libgazebo_ros_control.so*.
- mimic_joint_plugin: este plugin permite comandar articulaciones de forma mímica, necesario para la pinza donde cada dedo es una articulación distinta en el modelo URDF. Lo más seguro es que este plugin no lo tengas instalado. Sigue los siguientes pasos de la sección 8.1 para su instalación.

Se ha creado un nuevo paquete para guardar todos los ficheros relacionados con la simulación en Gazebo del robot: *turtlebot_arm_gazebo*. Este paquete contiene los modelos de los controladores y los lanzadores de la simulación. La carpeta *launch* incluye todos los lanzadores creados para probar una simulación.

Para cargar al modelo turtlebot_arm en gazebo ejecutar:

```
>> roslaunch turtlebot_arm_gazebo turtlebot_arm_gazebo.launch
```

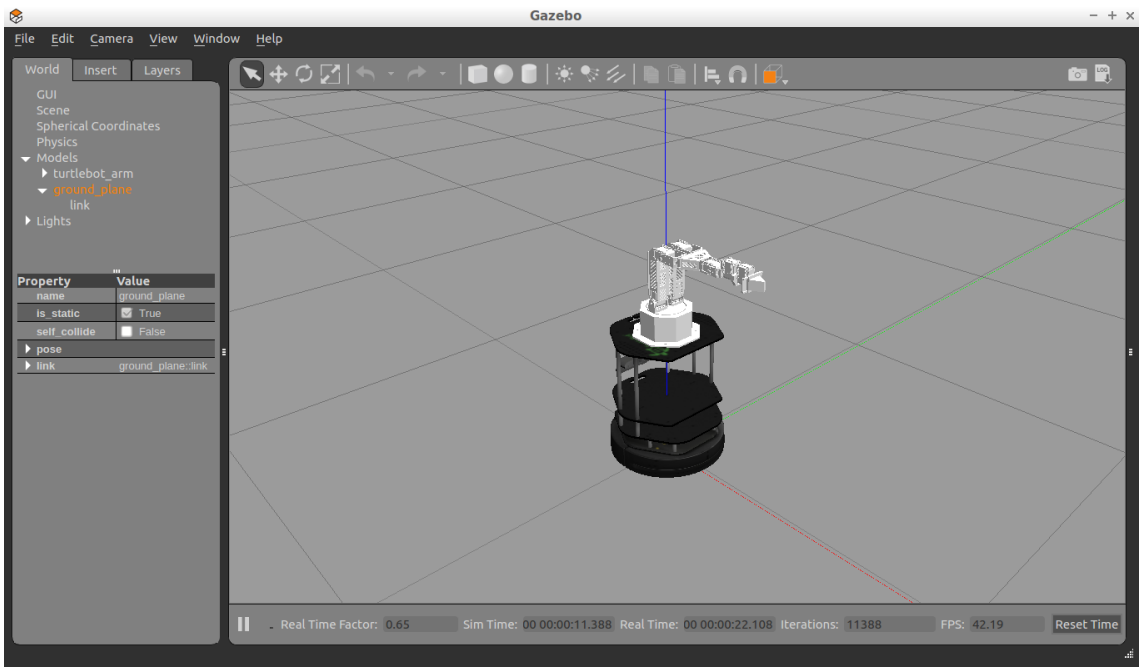


Figura 4.1 Simulación del tutlebot_arm en gazebo

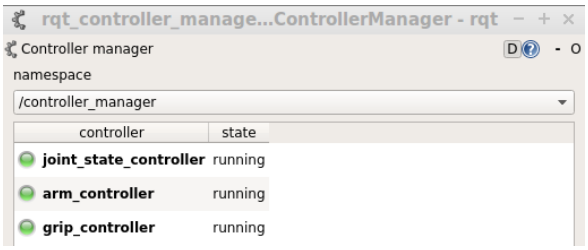
En ocasiones, los controladores no se cargan correctamente junto a la simulación. Para asegurarse de que los controladores han sido cargados, al ejecutar el lanzador deberían verse los siguientes mensajes al final de la terminal:

```
[ INFO] [1655023456.673025952, 0.321000000]: Loading gazebo_ros_control plugin
[ INFO] [1655023456.674409873, 0.321000000]: Starting gazebo_ros_control plugin in
namespace:
[ INFO] [1655023456.678233308, 0.321000000]: gazebo_ros_control plugin is waiting for
model URDF in parameter [/robot_description] on the ROS param server.
[ INFO] [1655023457.229027779, 0.321000000]: Loaded gazebo_ros_control.
[INFO] [1655023457.449251, 0.491000]: Controller Spawner: Waiting for service
controller_manager/switch_controller
[INFO] [1655023457.450493, 0.491000]: Controller Spawner: Waiting for service
controller_manager/unload_controller
[INFO] [1655023457.451929, 0.492000]: Loading controller: joint_state_controller
[INFO] [1655023457.521237, 0.545000]: Loading controller: arm_controller
[INFO] [1655023458.275941, 1.106000]: Loading controller: grip_controller
[INFO] [1655023458.645713, 1.300000]: Controller Spawner: Loaded controllers:
joint_state_controller, arm_controller, grip_controller
[INFO] [1655023458.651320, 1.303000]: Started controllers: joint_state_controller,
arm_controller, grip_controller
```

Otra manera de comprobar esto es ejecutar *rqt_controller_manager* (entrego este paquete junto al workspace por si no lo tienes instalado):

```
>> rosrun rqt_controller_manager rqt_controller_manager
```

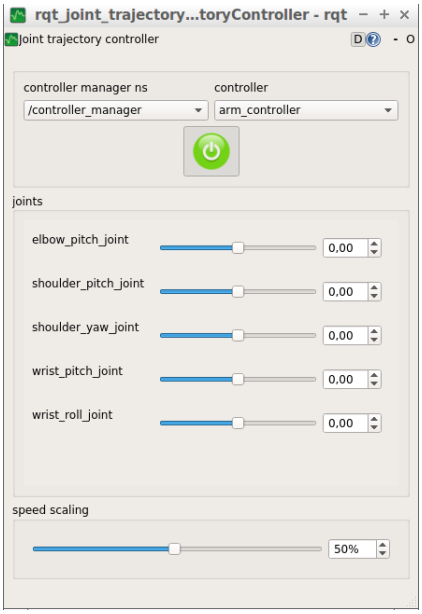
Debería abrirse una ventana con los tres controladores de la simulación. El botón de cada controlador indica en color verde si está habilitado y en gris sino. Debería verse así:



Si no se han cargado los controladores, reiniciar la simulación porque las articulaciones no podrán ser comandadas.

Ejecutando el siguiente comando, se carga una GUI para controlar de manera teleoperada las articulaciones:

```
>> rosrun rqt_joint_trajectory_controller rqt_joint_trajectory_controller
```



Los límites de movilidad de las articulaciones se han definido tal y como son en la realidad. En caso de encontrar un límite mal puesto, editar el fichero *turtlebot_arm_description/urdf/phantomx_reactor_plan_model.urdf.xacro*.

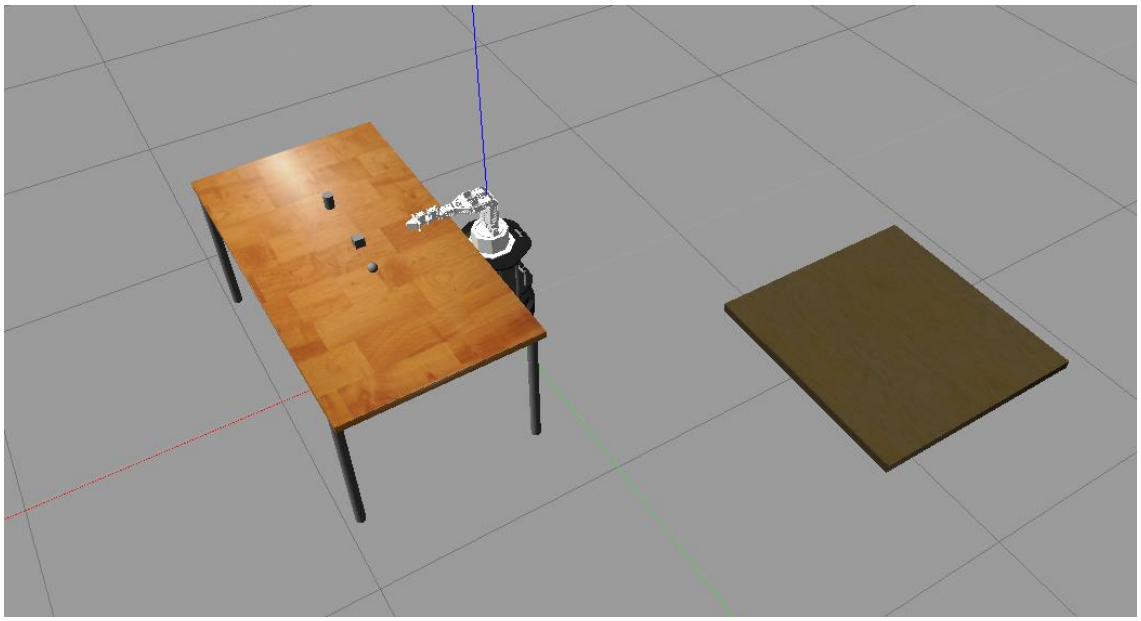
Para ejecutar el modelo *phantomx_reactor.xacro*, ejecutar el siguiente comando:

```
>> roslaunch turtlebot_arm_gazebo phantom_reactor_gazebo.launch
```

El problema con simular el brazo de manera individual es que su base no soporta el peso del brazo en algunas poses, provocando errores en posición y la inestabilidad del robot.

Se ha creado una demo de teleoperación del turtlebot_arm para probar el agarre de objetos. El robot se carga en un mapa con dos mesas, en una de las cuales hay tres objetos de diferentes geometrías. Se busca transportar un objeto de una mesa a la otra. Ejecutar el siguiente comando para su apertura:

```
>> roslaunch turtlebot_arm_gazebo turtlebot_arm_teleop_demo_gazebo.launch
```



Usar el siguiente comandos para teleoperar la base móvil del robot:

```
>> roslaunch turtlebot_teleop keyboard_teleop.launch
```

4.1 Programación de un launcher para la simulación del turtlebot_arm

Para una correcta simulación en Gazebo del modelo *turtelbot_arm.xacro*, se deben realizar las siguientes ejecuciones en un launcher:

1. Cargar el mundo de la simulación
2. Cargar la descripción del modelo del robot
3. Cargar el spawner del modelo en gazebo
4. Cargar el velocity muxel del TurtleBot
5. Cargar los bumpers del TurtleBot
6. Cargar el sensor láser
7. Cargar los controladores

Por ejemplo, el código del lanzador *turtlebot_arm_gazebo* es el siguiente:

```
1.  <launch>
2.
3.    <!-- Load map: empty world -->
4.   <arg name="gui" default="true"/>
5.   <include file="$(find gazebo_ros)/launch/empty_world.launch">
6.          <arg name="world_name" value="$(find
    turtlebot_arm_gazebo)/worlds/turtlebot_arm_table.world"/>
7.     <arg name="use_sim_time" value="true"/>
8.     <arg name="debug" value="false"/>
9.     <arg name="gui" value="$(arg gui)" />
10.  </include>
11.
12.   <!-- Robot description -->
13.  <include file="$(find
    turtlebot_arm_description)/launch/turtlebot_arm_load_description.launch"/>
14.
15.   <!-- Gazebo model spawner -->
16.  <node name="spawn_turtlebot_model" pkg="gazebo_ros" type="spawn_model"
17.                 args="$(optenv ROBOT_INITIAL_POSE) -unpause -urdf -param
    robot_description -model turtlebot_arm"/>
18.
19.   <!-- Velocity muxer -->
20.  <node
    pkg="nodelet" type="nodelet" name="mobile_base_nodelet_manager" args="manager"/>
21.  <node pkg="nodelet" type="nodelet" name="cmd_vel_mux"
22.          args="load yocs_cmd_vel_mux/CmdVelMuxNodelet mobile_base_nodelet_manager">
23.    <param name="yaml_cfg_file" value="$(find turtlebot_bringup)/param/mux.yaml" />
24.    <remap from="cmd_vel_mux/output" to="mobile_base/commands/velocity"/>
25.  </node>
26.
27.   <!-- Bumper/cliff to pointcloud (not working, as it needs sensors/core messages)
    -->
28.  <include file="$(find
    turtlebot_bringup)/launch/includes/kobuki/bumper2pc.launch.xml"/>
29.
30.   <!-- Fake laser -->
31.  <node
    pkg="nodelet" type="nodelet" name="laserscan_nodelet_manager" args="manager"/>
32.  <node pkg="nodelet" type="nodelet" name="depthimage_to_laserscan"
33.          args="load depthimage_to_laserscan/DepthImageToLaserScanNodelet
    laserscan_nodelet_manager">
34.    <param name="scan_height" value="10"/>
35.    <param name="output_frame_id" value="/camera_depth_frame"/>
36.    <param name="range_min" value="0.45"/>
37.    <remap from="image" to="/camera/depth/image_raw"/>
38.    <remap from="scan" to="/scan"/>
39.  </node>
40.
```

```
41.   <!-- Load controllers -->
42.  <rosparam file="$(find
    turtlebot_arm_gazebo)/config/turtlebot_arm_controllers.yaml" command="load"/>
43.  <node
    name="controller_spawner" pkg="controller_manager" type="spawner" respawn="false"
44.         output="screen"
45.         args="joint_state_controller arm_controller grip_controller"/>
46.
47. </launch>
```

4.2 Controladores de la simulación

Si para el robot real se necesita *arbotix*_ros para el control de los motores, para la simulación se necesita *controller_manager* [7]. Se han definido los controladores de tipo joint_trajectory_controller para la ejecución de trayectorias de tipo JointTrajectory, tal y como lo hace *follow_controller* en *arbotix_ros*.

Existe un controlador por cada articulación del robot, en lugar de un controlador por cada motor como ocurre con el robot real. A la hora de crear una arquitectura de control que sea compatible para la simulación y para el robot real, se debe tener en cuenta que para las articulaciones *elbow_pitch_joint* y *shoulder_pitch_joint* se utilizan dos controladores mímicos en el robot real, por lo que se debe hacer una conversión. La arquitectura de control diseñada para este proyecto lo ha tenido en cuenta (sección 5).

Los topics para comandar a los controladores son los mismos que los del controlador del arbotix_ros (sección 3.1), quitando las etiquetas de los motores mímicos. Esta vez, el topic para conocer el estado del controlador en cada iteración "*/controller_name/state*" si se carga.

Los controladores de la simulación usan un control PID, cuyos parámetros ya han sido ajustados para la correcta simulación. El controlador PID puede ajustarse en tiempo de ejecución de la simulación ejecutando:

```
>> rosrun rqt_reconfigure rqt_reconfigure
```

Los controladores pueden encontrarse definidos en el fichero YAML *turtlebot_arm_controllers.yaml*.

5 Arquitectura de control PhantomXReactor

Para facilitar los comandos de acción sobre las articulaciones del PhantomX Reactor, se ha desarrollado una arquitectura de control compatible para el robot real y la simulación en Gacebo.

Se ha creado el paquete *turtlebot_arm_controller* que incluye una configuración ArbotiX-M para ejecutar comandos de trayectoria (*arbotix_config_phantomx_reactor.yaml*) y el software de control (*phantomx_reactor.py*).

La configuración ArbotiX-M es como la mostrada en el apartado 3.1 pero sin los prefijos “arm_” en las etiquetas de los motores. Debe utilizarse esta configuración para que la arquitectura de control sea compatible.

La arquitectura de control ha sido programada en Python e incluye:

- Planificadores de trayectorias
- Comandos para ordenar a los actuadores del robot mediante trayectorias
- Retroalimentación de los estados de las articulaciones
- Gráficos de trayectorias planificadas y ejecutadas

5.1 Planificadores de trayectorias

Se han desarrollado varios planificadores de trayectorias implementadas en funciones. Todas ellas son implementación propia y fueron desarrolladas para la práctica 2 ”Interpolación de trayectorias” de la asignatura de Control de Robots en el lenguaje MATLAB. En este proyecto, se han pasado a código python. Para implementar todas las funciones matemáticas de los planificadores y que MATLAB ofrece, se ha utilizado la librería de Python numpy. Los planificadores implementados son:

- tpoly3(): interpolador cúbico punto a punto que solicita una posición inicial y otra final, una duración, y la velocidad inicial y final.
- jtraj3(): planifica varias trayectorias para varias articulaciones usando un interpolador cúbico punto a punto a partir de sus posiciones iniciales y finales, una duración, y sus velocidades iniciales y finales.
- tpolyN(): interpolador de grado N punto a punto que solicita una posición inicial y otra final, una duración, y la velocidad inicial y final.
- jtrajN(): planifica varias trayectorias para varias articulaciones usando un interpolador de grado N punto a punto a partir de sus posiciones iniciales y finales, una duración, y sus velocidades iniciales y finales.
- mstrajSplines(): interpolador usando splines e inetpolador cúbico para varias articulaciones a partir de sus los puntos de paso, la duración de os tramos duración, y y la velocidad final e inicial de los puntos.

Los planificadores de trayectorias nombrados no dan lugar a mensajes de tipo jointTrajectory ya que he preferido hacer estas funciones abstraídas de ROS. En su lugar, he creado la clase Planning(), que representa una lista de waypoints. No posee parámetros de construcción y sus atributos son una lista de posiciones, velocidades, aceleraciones y de timestamps. Todos son del mismo tamaño y cada índice corresponde a un waypoint. En el caso de los planificadores jtraj3(), jtrajN() y mstrajSplines(), devuelven una lista de instancias a esta clase, una por cada articulación.

```
1. class Planning():
2.   def __init__(self):
3.     self.positions = list()
4.     self.velocities = list()
5.     self.accelerations = list()
6.     self.times = list()
```

Estos planificadores permiten decidir si se desea graficar los perfiles de posición, velocidad y aceleración planificados y que se muestren y/o se guarden en algún directorio.

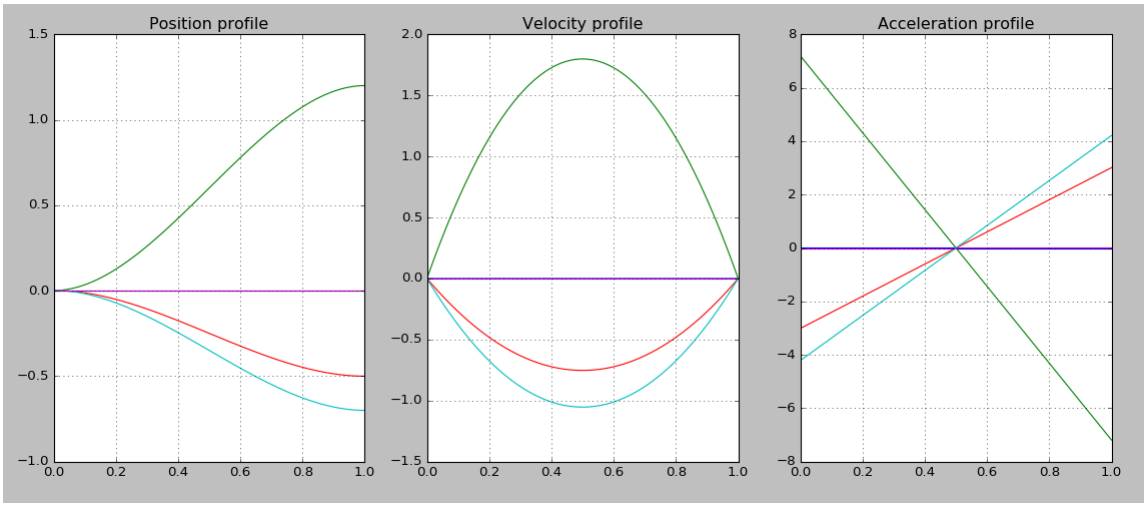


Figura 5.1 Perfiles de posición, velocidad y aceleración planificados por la implementación en pyt hon de jtraj3() para trayectorias reposo a reposo de 4 articulaciones, una sin movimiento

5.2 Clase PhantomXReactor

La clase PhantomXReactor() permite la representación de un PhantomX Reactor Arm, comandarlo mediante mensajes de tipo trajectory_msgs/JointTrajectory y la retroalimentación de información. Esta clase viene declarada junto a las los planificadoras de trayectorias y la clase Planning en el fichero phantomx_reactor.py. Se ha desarrollado para este proyecto la guía de uso en el laboratorio del PhantomX Reactor donde se explica detalladamente esta clase, sus métodos y atributos y como usarlos para obtener varias funcionalidades.

PhantomXReactor permite la ejecución de trayectorias tanto en el PhantomX Reactor real como en el simulado, sin apenas hacer cambios en el código para facilitar el paso de uno al otro. La clase está compuesta principalmente por:

- El constructor de la clase
- Funciones getters para obtener algunos atributos de la clase
- Funciones callback de los topics a los que está subscrito
- Función ejecutoras de trayectoria

A continuación, se explicarán estos puntos para más adelante mostrar algunas de las funcionalidades que puede ofrecer.

5.2.1 Constructor de la clase

El constructor de la clase se declara en python como def __init__() En el siguiente código se muestran los parámetros de construcción. Los parámetros de construcción de esta clase permiten representar al PhantomX Reactor con los nombres de las articulaciones, controladores o espacios de nombre que se hayan configurado, otorgando flexibilidad a la

hora de representar varios modelos personalizados. Por defecto, sus valores funcionan con el modelo programado en este proyecto, por lo que no es necesario actualizar estos parámetros usando esta configuración. Los parámetros de construcción son:

```
1.  def __init__(self,
2.          arm_controller_name = 'arm_controller',
3.          grip_controller_name = 'grip_controller',
4.          ns = '',
5.          arm_joint_names = [
6.                              "shoulder_yaw_joint",
7.                              "shoulder_pitch_joint",
8.                              "elbow_pitch_joint",
9.                              "wrist_pitch_joint",
10.                             "wrist_roll_joint"
11.                              ],
12.         grip_joint_names = [
13.                             "gripper_right_joint"
14.                              ],
15.         mimic_real_joints = [
16.                         ["shoulder_pitch_joint", "shoulder_pitch_mimic_joint"],
17.                         ["elbow_pitch_joint", "elbow_pitch_mimic_joint"]
18.                              ],
19.         ):
```

- arm_controller_name: nombre del controlador para el grupo de articulaciones del brazo.
- grip_controller_name: nombre del controlador para el grupo de articulaciones de la pinza.
- ns: espacio de nombres del robot.
- arm_joint_names: grupo de nombres del brazo
- grip_joint_names: grupo de nombres de la pinza
- mimic_real_joints: matriz de Nx2 donde N es el número de articulaciones que poseen motores mímicos en el robot real. La primera columna corresponde al nomrbre de la articuación presente en el modelo simulado y la segunda al nombre de la artiuclació mímica únicamente presente en el real.

Una vez llamada la clase, comienza su inicialización. Primeramente, el constuctor buscará un modelo URDF cargado con el nombre de 'robot_description'. En caso de encontrarse cargado, se mostrará el siguiente error:

[ERROR] [1653819243.079743]: Error in PhantomxReactor.__init__(): robot_description is not defined or robot model is not loaded.
Solution: Load with robot_description tag one of this robots models:
- phantomx_reactor
- turtlebot_arm

Luego de cargar 'robot_description', el constructor convertirá a un string todo el archivo xml del modelo del URDF. Esto con la finalidad de obtener del URDF los límites de las articulaciones especificadas en los parámetros de construcción mediante la función propia **get_joints_from_URDF()**. Esto permitirá revisar a la hora de ejecutar una trayectoria que ninguno de los waypoints planificados se salga de los límites de la articulación y así no sobrecargar los motores. Seguidamente, guarda los parámetros de construcción y los límites de las articulaciones como atributos y carga algunos atributos sobre el seguimiento de trayectorias. La inicialización acaba inicializando publicaciones a los topics para comandar a los controladores y subscripciones a los topics de retroalimentación de estados:

- Publicadores a ns/[controller_name]/follow_joint_trajectory/goal: publica las trayectorias planificadas en el controlador para su ejecución.
- Subscriptores a ns/[controller_name]/follow_joint_trajectory/result: obtiene el resultado fnal de la trayectoria ejecutada en el controlador.
- Subscriptores a ns/[controller_name]/state: obtiene el estado del controlador durante su ejecución.

5.2.2 Funciones getters

Estas funciones devuelven información sobre la clase. Se encuentran las siguientes:

- get_[joints_group_name]_joints_names: devuelve un vector con los nombres de las articulaciones de un grupo o todas las de la clase.
- get_traj_started: devuelve True si una trayectoria ha a acabado de ejecutarse.
- get_traj_terminated: devuelve True si una trayectoria a empezado a ejecutarse.
- get_actual_state: devuelve el estado de las articulaciones al momento de ejecutarse.
- get_last_[joint_group_name]_traj_controller_states: obtiene una lista de los estados de las articulaciones durante toda la ejecución de una trayectoria.

En lo referente a la función **get_last_[joint_group_name]_traj_controller_states**, los estados vienen definidos en una variable de tipo **JointTrajectoryControllerState**. Se trata de un tipo de mensaje devuelto por el topic ns/[controller_name]/state, que otorga información sobre el estado de un controlador. En este topic solo se publica cuando existe una trayectoria ejecutándose en el controlador, por lo que esta función solo debería usarse cuando una trayectoria ya se haya ejecutado, y además, mostrará los estados de la última ejecutada. Permite conocer los estados de las articulaciones que controla, el estado que deberían tener y el error entre ambas. Esto proporciona la posibilidad de que, al acabar una trayectoria, pueda compararse cómo se ha ejecutado la trayectoria en función de lo planificado. En la figura Figura 5.2 se muestra la estructura de datos de un mensaje de tipo JointTrajectoryControllerState. Esta función solo funciona para el modelo simulado, ya que el controlador del modelo real no crea el topic ns/[controller_name]/state, donde se publica este tipo de mensaje.

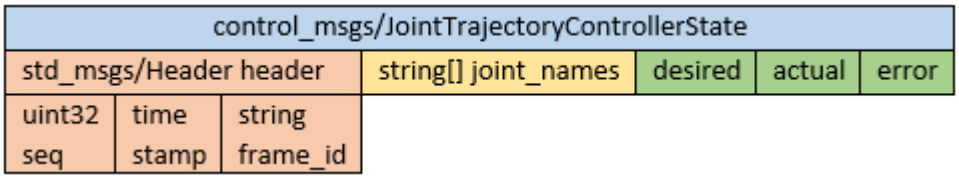


Figura 5.2 Estructura de datos del mensaje control_msgs/JointTrajectoryControllerState, donde los campos desired, actual y error son de tipo trajectory_msgs/JointTrajectoryPoint, definido en la Sección 3.2

5.2.3 Funciones Callbacks

Estas funciones se llaman cada vez que se hace una publicación en los topics a los que la clase está subscrito. Sirven para llevar un seguimiento sobre la ejecución de una trayectoria. Son las siguientes:

- [controller_name]_state_cb: llamado por el subscriptor ns/[controller_name]/state, apila en una lista el estado actual del controlador, definido en una variable de tipo JointTrajectoryControllerState.

- [joints_group_name]_FollowJointTrajectoryActionResult_cb: actualiza el estado de la trayectoria, guardado en lo atributos de seguimiento de trayectoria.

5.2.4 Funciones ejecutoras de trayectorias

Estas funciones permiten la ejecución de las trayectorias planificadas por los planificadores de trayectorias descritos en la Sección 5.1. Los controladores implementados aceptan comandos de tipo JointTrajectory, pero los planificadores devuelven una lista de tipo Planning, por lo que se debe crear un conversor de variable antes de la ejecución. Posteriormente, la trayectoria puede comandarse a los controladores.

El método **createJointTrajectory()** devuelve una variable de tipo JointTrajectory a partir de una planificación de trayectoria. Para ello, solicita una cadena con los nombres de las articulaciones en la trayectoria (joint_names) y una lista de tipo Planning (plan), que debe tener la misma longitud que la cadena joint_names.

A la hora de planificar una trayectoria, no hay que tener en cuenta las articulaciones mímicas. En caso de ejecutar el programa en el robot real, se determinará declarando el parámetro ”real” a True, en cuyo caso, gracias al atributo mimic_real_joints, se añadirán las planificaciones mímicas a la trayectoria, que serán copia inversa de sus articulaciones ligadas.

Esta función también comprueba que ningún waypoint de la trayectoria se salga de los límites articulares, en cuyo caso devolverá None y mostrará un error como el siguiente:

[ERROR] [1653819243.079743]: Error in PhantomxReactor.creteJointTrajectory(): values in ’gripper_right_joint’ trajectory out of limits.

```
1. def creteJointTrajectory (
2.     self,
3.     joint_names ,
4.     plan ,
5.     real = False):
```

Una vez se tiene la trayectoria de tipo JointTrajectory, se puede llamar a la función **executeTrajectory** para su ejecución. En ese caso, se debe determinar el grupo de articulaciones a la que va dirigida: para el grupo ”arm”, el parámetro ”arm_traj” se declara con la trayectoria, e igual para el grupo ”grip” con ”grip_traj”. Esto quiere decir que se permite la ejecución de trayectorias en ambos grupos al mismo tiempo. No se determinará como terminada la trayectoria (get_traj_terminated()) hasta que hayan acabado de ejecutarse ambas. Esta función también preparará las variables de seguimiento de trayectoria para la retroalimentación de información, como borrar la lista de tipo JointTrajectoryControllerState para actualizarla durante la ejecución.

```
1. def executeTrajectory (
2.    self,
3.    arm_traj = None,
4.    grip_traj = None):
```

Todos los comandos de las articulaciones está en radianes, a excepción de la pinza que está en metros. Esto funciona en el robot simulado porque la articulación de la pinza es prismática, sin embargo, en el robot real es un motor el que mueve la pinza. Es por ello que he programado dos conversores para la ejecución de trayectorias en la pinza del robot real.

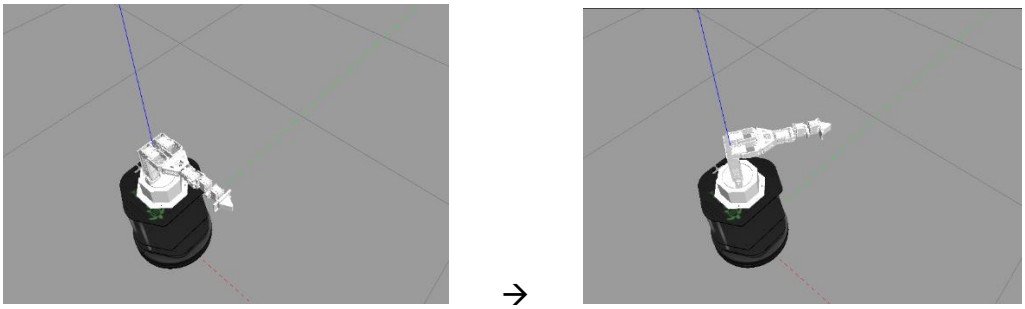
- angleToDistance: convierte un valor en radianes del motor a metros recorridos por la pinza.
- distanceToAngle: convierte un valor en metros recorridos por la pinza a radianes del motor. Estas funciones solo se utilizan con el robot real para obtener en metros la posición actual de la pinza y comandar en metros a la articulación.

6 Ejemplos de uso de la arquitectura de control

El fichero *phantomx_reactor.py* incluye una función main donde programar todas las tareas de manipulación. A continuación, se mostrarán algunos ejemplos de código para entender cómo usar la arquitectura software para tus propios comportamientos.

Trayectoria PTP de una articulación en simulación

Trayectoria con interpolación de grado 7 de la articulación *shoulder_yaw_joint* desde la posición actual hasta la posición 1 rad en 1 segundo.



```
1.    phantomx_reactor = PhantomXReactor()
2.    joint_names = phantomx_reactor.get_arm_joint_names()
3.    actual_state = phantomx_reactor.get_actual_state(joint_names)
4.
5.    q0 = actual_state.positions
6.    qf = [1,0,0,0,0]
7.    T = 1
8.    N = 7
9.    plan = jtrajN(N, q0, qf, T)
10.   traj = phantomx_reactor.creteJointTrajectory(joint_names, plan)
11.   phantomx_reactor.executeTrajectory(arm_traj = traj)
12.   while not phantomx_reactor.get_traj_terminated():
13.     g = 0
14.   compare(plan, phantomx_reactor.get_last_arm_traj_controller_states(), 0)
```

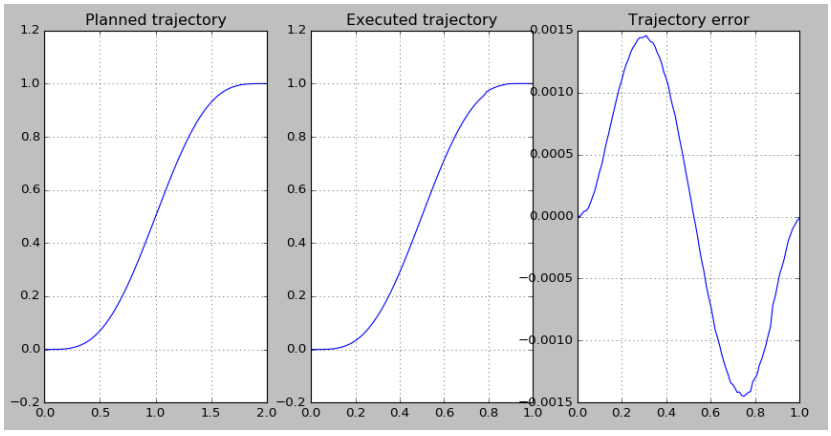
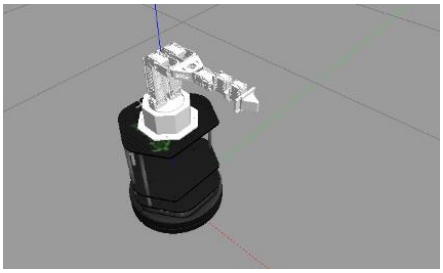
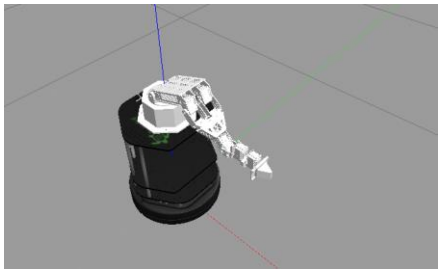


Figura 6.1 Salida de la función compare. Perfil ejecutado, planificado y error de la articulación shoulder_yaw_joint

Trayectoria PTP de varias articulaciones en simulación

Trayectoria con interpolación de grado 7 de las articulaciones del brazo desde la posición actual hasta la posición [0,1.2,-0.5,-0.7,0] rad en 1 segundo.

```
1.    phantomx_reactor = PhantomXReactor()
2.    joint_names = phantomx_reactor.get_arm_joint_names()
3.    actual_state = phantomx_reactor.get_actual_state(joint_names)
4.
5.    q0 = actual_state.positions
6.    qf = [0,1.2,-0.5,-0.7,0]
7.    T = 1
8.    N = 7
9.    plan = jtrajN(N, q0, qf, T)
10.   traj = phantomx_reactor.creteJointTrajectory(joint_names, plan)
11.   phantomx_reactor.executeTrajectory(arm_traj = traj)
12.   while not phantomx_reactor.get_traj_terminated():
13.     g = 0
14.   compare(plan, phantomx_reactor.get_last_arm_traj_controller_states(), 1)
```

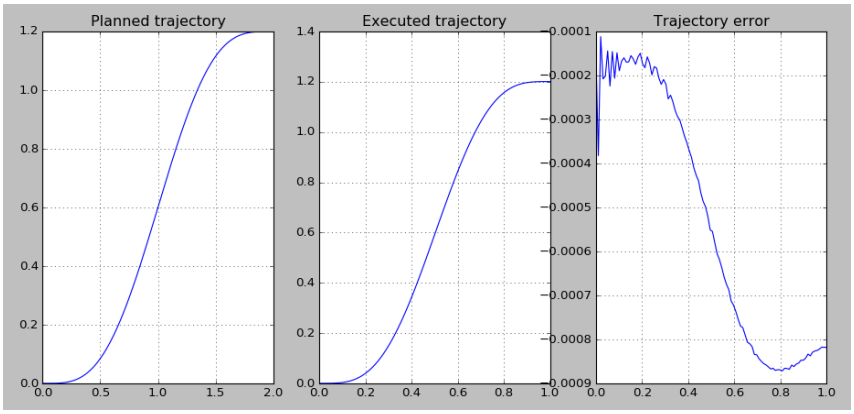


Figura 6.2 Salida de la función compare. Perfil ejecutado, planificado y error de la articulación shoulder_pitch_joint

Obtención de las posiciones actuales del robot

El siguiente código permite obtener las posiciones de las articulaciones en la pose actual. Sirve para obtener los waypoints de una trayectoria deseada. En caso de utilizar este código en el robot real, hacerlo sin que los motores estén habilitados. Ejecutar el código, mover el brazo a una pose y presionar 'enter'. Se imprimirá en la terminal las posiciones de las articulaciones especificadas en el código. Para el robot real:

```
1.  def get_arm_positions():
2.    phantomx_reactor = PhantomXReactor()
3.    arm_joint_names = phantomx_reactor.get_arm_joint_names()
4.    enter = ''
5.    while not rospy.is_shutdown():
6.      print "Press enter to print actual arm pose "
7.      something = raw_input()
8.      arm_actual_state =
9.      phantomx_reactor.get_actual_state(arm_joint_names, real=True)
10.     rospy.loginfo(arm_actual_state.positions)
```

Ejecutar una trayectoria en la pinza real

```
phantomx_reactor = PhantomXReactor()
  joint_names = phantomx_reactor.get_grip_joint_names()
  actual_state = phantomx_reactor.get_actual_state(joint_names, real = True)
  q0 = actual_state.positions
  qf = [0.017]
  T = 1
  N = 7
  plan = jtrajN(N, q0, qf, T)
  traj = phantomx_reactor.creteJointTrajectory(joint_names, plan, real = True)
  phantomx_reactor.executeTrajectory(grip_traj = traj)

  while not phantomx_reactor.get_traj_terminated():
    g = 0
```

Ejecutar una trayectoria con splines en el robot simulado

```
def myTraj1():

  # Create control
  phantomx_reactor = PhantomXReactor()
  arm_joint_names = phantomx_reactor.get_arm_joint_names()
  grip_joint_names = phantomx_reactor.get_grip_joint_names()
  actual_state = phantomx_reactor.get_actual_state(arm_joint_names)
  grip_actual_state = phantomx_reactor.get_actual_state(grip_joint_names)
  q0 = actual_state.positions
  grip0 = grip_actual_state.positions
  OPEN = [0.03]
  CLOSE = [0.00]

  # Plan
  Pick_plan = mstrajSplines([q0, PICK1, PICK2], [1,1])
  Open_grip_plan = jtrajN(7, grip0, OPEN, 1)
  Close_grip_plan = jtrajN(7, OPEN, CLOSE, 1)

  # Execute PICK
  PICK_traj = phantomx_reactor.creteJointTrajectory(arm_joint_names, Pick_plan)
  OPEN_traj = phantomx_reactor.creteJointTrajectory(grip_joint_names, Open_grip_plan)
  CLOSE_traj = phantomx_reactor.creteJointTrajectory(grip_joint_names, Close_grip_plan)
  phantomx_reactor.executeTrajectory(arm_traj = PICK_traj, grip_traj = CLOSE_traj)

  while not phantomx_reactor.get_traj_terminated():
    g = 0
```

7 Workspace del proyecto

Todo lo desarrollado para este proyecto ha sido integrado en un mismo espacio de trabajo. A continuación, describiré los paquetes desarrollados:

- *turtlebot_arm_description*: contiene el modelo mecánico del robot, tanto el del PhantomX Reactor Arm en solitario como el del acoplado en el TurtleBot, descrito en un URDF. También contiene los ficheros launch para cargar el modelo con el nombre robot_description en el programa de monitorización rviz.
- *turtlebot_arm_moveit_config*: contiene los modelos SRDF para el uso de MoveIt.
- *turtlebot_arm_gazebo*: contiene los ficheros launch para cargar en Gazebo al robot, tanto el PhantomX Reactor en solitario como el acoplado en el TurtleBot. También tiene las cargadores para el uso de Gazebo y MoveIt simultáneamente y el tutorial de teleoperación.
- *turtlebot_arm_controller*: contiene la arquitectura de control del robot PhantomXReactor y la configuración para *arbotix_ros*.
- *arbotix_ros*: contiene los drivers de comunicación entre ArbotiX y ROS.

8 Anexo

8.1 Instalación del plugin mimic_gazebo

Un requisito para la correcta simulación en Gazebo del PhantomX Reactor es tener instalado el plugin **mimic_joint_plugin**. Si nunca lo has usado, lo más seguro es que no lo tengas instalado. Puedes consultar la página Plugins 101 para saber cómo instalar un plugin en gazebo o seguir los pasos descritos a continuación para instalar éste en específico:

1. Descargar el paquete *roboticsgroup_gazebo_plugins* :

git clone https://github.com/roboticsgroup/roboticsgroup_gazebo_plugins.git

2. Guardar este paquete en un lugar seguro donde no lo vuelvas a mover de ahí. Yo recomiendo crear en /home/user la carpeta *NoBorrar* y guardarla ahí. Si después de ejecutar los siguientes comandos decides cambiar el directorio del paquete, no serás capaz de usar el plugin.

3. Crear el fichero del plugin *libroboticsgroup_gazebo_mimic_joint_plugin.so* ejecutando los siguientes comandos:

cd ~/roboticsgroup_gazebo_plugins

mkdir build

cd build

cmake ../

make

4. Se te habrá creado el fichero *libroboticsgroup_gazebo_mimic_joint_plugin.so* en el directorio *~/ roboticsgroup_gazebo_plugins/build/devel/lib/libroboticsgroup_gazebo_mimic_joint_plugin.so* . Este fichero es el que hay que ejecutar en tus proyectos para usar el plugin. Para incluirlo a gazebo realizar en la terminal:

export GAZEBO_PLUGIN_PATH=${GAZEBO_PLUGIN_PATH}:~/roboticsgroup_gazebo_plugins/build/devel/lib

Y ya estaría listo! A mi, sin embargo, este último paso no me funciona, así que he tenido que poner en el modelo URDF del robot la ruta del fichero de lanzamiento. Cambia tanto en *phantomx_reactor.xacro* como en *turtlebot_arm.xacro* la ruta del campo *filename* en el plugin por la tuya propia.

9 Referencias

[1] D. Motors, «Dynamixel AX-12A,» [En línea]. Available: https://sandorobotics.com/producto/902-0003-001/.

[2] «Getting Started with the Arbotix-M,» [En línea]. Available: https://github.com/AalborgUniversity-ControlLabs/start-here/blob/master/crust-crawler-arms/getting-started-with-arbotix-m.md.

[3] «FTDI Chip,» [En línea]. Available: https://ftdichip.com/drivers/.

[4] «phantomx_reactor_arm,» [En línea]. Available: https://github.com/RobotnikAutomation/phantomx_reactor_arm.

[5] «arbotix_ros,» [En línea]. Available: https://github.com/vanadiumlabs/arbotix_ros.

[6] RobotnikAutomation, «https://github.com/RobotnikAutomation/phantomx_reactor_arm,» 2019. [En línea].