Carlos Sirvent Miquel
Stanimir Hristov Hristov

TRABAJO DE TEORÍA:

CREACIÓN DE UN ROBOT MÓVIL PERSONALIZADO A PARTIR DE UN FICHERO URDF

Índice:

Introducción

En este archivo se va a explicar cómo crear un robot móvil personalizado a partir de un fichero URDF y a conectarlo con ROS para usar sus funcionalidades. También se explica cómo visualizar el proceso de construcción en la herramienta RVIZ y una vez terminado, su simulación en el entorno de Gazebo.

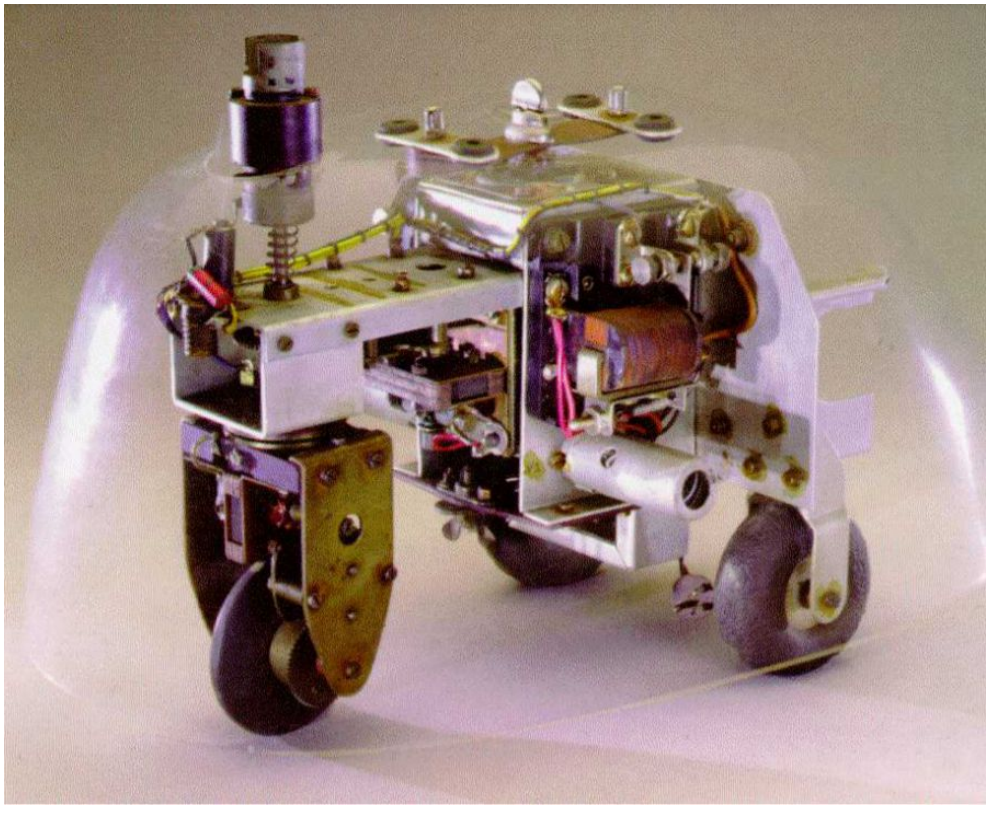
URDF

Definición:

URDF son las siglas de Unified Robot Description Format. Es el tipo de archivo que usa ROS para definir y modelar los robots en este entorno, desde robots móviles simples a complejos robots articulares o planares. Como indica su nombre, un fichero URDF es general pues ha de valer para todo tipo de robot. Es sencillo en un principio, pues el modelo del robot solamente se basa en *links* y *joints*, es decir, eslabones y articulaciones respectivamente. Son los parámetros que se especifican en cada uno de ellos lo que compone la morfología del robot.

Modelado del robot:

En este ejemplo se ha decidido seguir el tutorial del pdf que se adjunta junto a este archivo (págs. 275-397). En él se modela el siguiente robot móvil:

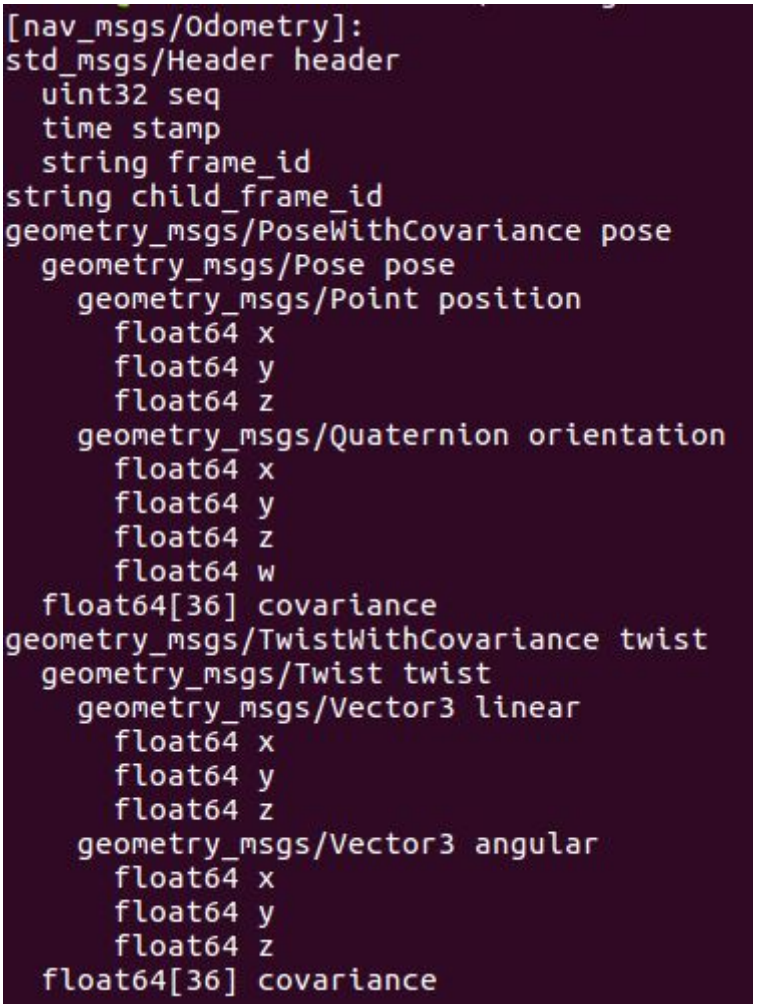


Como se puede observar, se compone de una rueda direccional y motriz delantera y dos ruedas pasivas traseras (esta configuración se cambiará con posterioridad por simplicidad, siendo las ruedas traseras las motrices). La pregunta ahora es cómo crear el archivo URDF correspondiente a este pequeño robot.

Como queremos conectar el modelo con ROS, éste deberá cumplir sus estándares. Como la funcionalidad del robot es el movimiento, nos interesará proporcionar la velocidad y la dirección del robot y conocer su posición y orientación desde un punto de vista global. Esto en ROS se refleja en los topics *cmd_vel* y *odom* respectivamente. En *cmd_vel* se publican mensajes de tipo *geometry_msgs/Twist,* que contienen lo siguiente:

```
[geometry_msgs/Twist]:
geometry_msgs/Vector3 linear
  float64 x
  float64 y
  float64 z
geometry_msgs/Vector3 angular
  float64 x
  float64 y
  float64 z
```

Esos números en coma flotante se refieren a la velocidad lineal/angular en su respectivo eje. Por defecto, el eje x apunta hacia delante, z hacia arriba e y completando un sistema dextrógiro con los otros dos ejes. Siendo así y teniendo la configuración de este robot, solamente nos interesa la velocidad lineal en el eje x y la angular en el eje z. En el topic odom se publican mensajes tipo *nav_msgs/Odometry*, que contienen lo siguiente:



Como hemos dicho anteriormente, solamente nos interesa la posición y orientación del robot desde un punto de vista global, así que de entre todos los parámetros solamente nos interesan los que se encuentran dentro de *geometry_msgs/Pose pose*, es decir el quaternio de orientación y el punto con las coordenadas. Como vamos a crear el robot en un entorno vacío posteriormente, la posición y orientación inicial del robot no nos interesa así que empezaremos en (0,0,0) y con la orientación predefinida.

Teniendo en cuenta esto pasamos a modelar el robot en URDF. Para ello creamos un entorno de ROS y el archivo llamado en ese caso *own_movile_robot.urdf* ejecutando los siguientes comandos:

```
mkdir -p ~/catkin_ws/src

cd ~/catkin_ws/

catkin_make

source devel/setup.bash

cd src

touch own_movile_robot.urdf
```

Este tipo de archivo tiene formato XML, por lo que hay que ceñirse a ese lenguaje. Vamos a modelar el robot con figuras geométricas simples como son prismas rectangulares y cilindros. Se podría dar forma mediante un .cad creado con otro programa, pero vayamos por lo simple.

Ya se ha explicado que el modelado se realiza a partir de *links* y *joints,* distinguidos entre ellos por su propio nombre. Para ello lo más sencillo es imaginarse el robot como una estructura de un árbol. Parte de un *link* que será el padre del árbol y sus hijos serán otros *links* conectados a él mediante *joints*, y así sucesivamente hasta conectar todas las partes del robot. Tomaremos como el *link* padre al chasis o cuerpo del mismo, y a él le conectamos mediante *joints* las ruedas traseras y otro *link* que nos servirá para proporcionar la dirección a la rueda delantera. A este último *link* le conectamos la rueda delantera. Para cada *link* hay que definir su tamaño. Para que Gazebo pueda calcular correctamente las físicas del robot, también les añadiremos la colisión (igual que el tamaño) y la inercia(masa y valores respecto a los ejes, más fácil con figuras geométricas simples). Para cada *joint* hay que hay que definir su tipo, que será *revolute* (giro circular sin tope) en todas ellas para este robot, el *link* padre y el hijo que se unen entre sí según la jerarquía del árbol y la posición relativa del segundo respecto al primero, teniendo en cuenta que las distancias se toman desde el punto central de cada *link*. Un posible archivo URDF sería así:

```
<?xml version="1.0"?>
<robot name="own_movile_robot">
	<link name="base_link">
		<visual>
			<geometry>
				<box size="0.6 0.3 0.3"/>
			</geometry>
			<material name="silver">
				<color rgba="0.75 0.75 0.75 1"/>
			</material>
		</visual>
		<collision>
			<geometry>
				<box size="0.6 0.3 0.3"/>
			</geometry>
		</collision>
		<inertial>
			<mass value="1.0"/>
			<inertia ixx="0.015" iyy="0.0375" izz="0.0375"
				ixy="0" ixz="0" iyz="0"/>
		</inertial>
	</link>

	<link name="front_caster">
		<visual>
			<geometry>
```

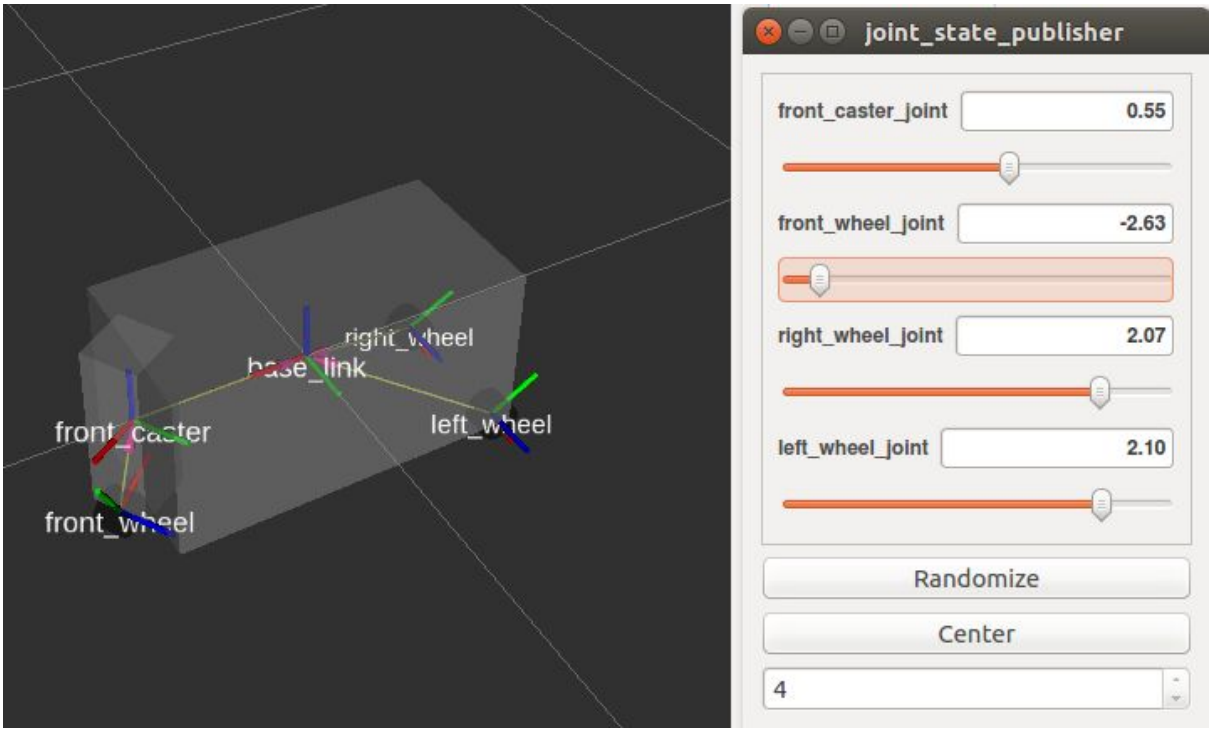
```
                  <box size="0.1 0.1 0.3"/>
            </geometry>
            <material name="silver"/>
      </visual>
      <collision>
            <geometry>
                  <box size="0.1 0.1 0.3"/>
            </geometry>
      </collision>
      <inertial>
            <mass value="0.1"/>
            <inertia ixx="0.00083" iyy="0.00083" izz="0.000167"
                  ixy="0" ixz="0" iyz="0"/>
      </inertial>
</link>

<joint name="front_caster_joint" type="continuous">
      <axis xyz="0 0 1"/>
      <parent link="base_link"/>
      <child link="front_caster"/>
      <origin rpy="0 0 0" xyz="0.3 0 0"/>
</joint>

<link name="front_wheel">
      <visual>
            <geometry>
                  <cylinder length="0.05" radius="0.035"/>
            </geometry>
            <material name="black"/>
      </visual>
      <collision>
            <geometry>
                  <cylinder length="0.05" radius="0.035"/>
            </geometry>
      </collision>
      <inertial>
            <mass value="0.1"/>
            <inertia ixx="5.1458e-5" iyy="5.1458e-5" izz="6.125e-5"
                  ixy="0" ixz="0" iyz="0"/>
      </inertial>
</link>

<joint name="front_wheel_joint" type="continuous">
      <axis xyz="0 0 1"/>
      <parent link="front_caster"/>
      <child link="front_wheel"/>
      <origin rpy="-1.5708 0 0" xyz="0.05 0 -.15"/>
</joint>

<link name="right_wheel">
      <visual>
            <geometry>
                  <cylinder length="0.05" radius="0.035"/>
```

```
                </geometry>
                <material name="black">
                        <color rgba="0 0 0 1"/>
                </material>
        </visual>
        <collision>
                <geometry>
                        <cylinder length="0.05" radius="0.035"/>
                </geometry>
        </collision>
        <inertial>
                <mass value="0.1"/>
                <inertia ixx="5.1458e-5" iyy="5.1458e-5" izz="6.125e-5"
                        ixy="0" ixz="0" iyz="0"/>
        </inertial>
</link>

<joint name="right_wheel_joint" type="continuous">
        <axis xyz="0 0 1"/>
        <parent link="base_link"/>
        <child link="right_wheel"/>
        <origin rpy="-1.5708 0 0" xyz="-0.2825 -0.125 -.15"/>
</joint>

<link name="left_wheel">
        <visual>
                <geometry>
                        <cylinder length="0.05" radius="0.035"/>
                </geometry>
                <material name="black"/>
        </visual>
        <collision>
                <geometry>
                        <cylinder length="0.05" radius="0.035"/>
                </geometry>
        </collision>
        <inertial>
                <mass value="0.1"/>
                <inertia ixx="5.1458e-5" iyy="5.1458e-5" izz="6.125e-5"
                        ixy="0" ixz="0" iyz="0"/>
        </inertial>
</link>

<joint name="left_wheel_joint" type="continuous">
        <axis xyz="0 0 1"/>
        <parent link="base_link"/>
        <child link="left_wheel"/>
        <origin rpy="-1.5708 0 0" xyz="-0.2825 0.125 -.15"/>
</joint>
</robot>
```

Visualización en RVIZ:

En cualquier momento durante la modelización del robot se puede visualizar el progreso mostrándolo en RVIZ. Esto sirve mayoritariamente para observar si la posición y orientación elegida para cada parte del robot es correcta. Además se puede comprobar el funcionamiento de los *joints* ya que aparece una pestaña que los permite girar. Esto es posible gracias a ROS, pues al ejecutar la visualización en RVIZ, los nodos robot_state_publisher y joint_state_publisher están ejecutándose. Para ello hay que abrir un terminal en el directorio donde se encuentre el archivo own_movile_robot.urdf y ejecutar el siguiente comando:

```
roslaunch urdf_tutorial display.launch model:=own_movile_robot.urdf \ gui:=True
```



Gazebo

Para poder visualizar en gazebo este modelo de robot primero debemos crear un ROS package. No es la única forma de hacerlo pero es una de las más intuitivas. Para ello abrimos un terminal en el directorio `catkin_ws/src` y ejecutamos los siguientes comandos:

```
catkin_create_pkg own_movile_robot std_msgs urdf controller_manager joint_state_controller robot_state_publisher gazebo_ros rviz
```

```
mkdir launch
```

```
cd launch
```

```
touch own_movile_robot_gazebo.launch
```

```
cd ..

mkdir urdf
```

Dentro de la carpeta urdf hay que colocar el archivo own_movile_robot.urdf anteriormente creado. Como se puede observar, se ha creado el archivo own_movile_robot_gazebo.launch dentro de la carpeta launch. En él escribiremos lo siguiente:

```
<launch>

	<!-- Load the TortoiseBot URDF model into the parameter server -->

		<param name="robot_description" textfile="$(find own_movile_robot)/urdf/own_movile_robot.urdf" />

	<!-- Start Gazebo with an empty world -->

	<include file="$(find gazebo_ros)/launch/empty_world.launch"/>

	 <!-- Spawn a TortoiseBot in Gazebo, taking the description from the parameter server -->

	<node name="spawn_urdf" pkg="gazebo_ros" type="spawn_model"

	args="-param robot_description -urdf -model own_movile_robot" />

</launch>
```

Cuando se lance este archivo se cargará en el servidor de parámetros el robot modelado en own_movile_robot.urdf guardadas como /robot_description. Después se lanza gazebo con un entorno vació y finalmente se crea en él una instancia de nuestro robot. Para que esto ocurra hay que abrir un terminal en `catkin_ws` y ejecutar el siguiente comando:

```
catkin_make

source devel/setup.bash

roslaunch own_movile_robot own_movile_robot_gazebo.launch
```

Teleoperación

Ya que hemos modelado el robot según los estándares de ROS, está al alcance de nuestra mano poder teleoperarlo. Para ello debemos crear un plugin de control de los motores, o en nuestro caso encontrar uno predefinido. Para ello hay que introducir las siguientes líneas en el archivo own_movile_robot.urdf, dentro del objeto <robot>:

```
<gazebo>
      <plugin name="differential_drive_controller"
              filename="libgazebo_ros_diff_drive.so">
                  <leftJoint>left_wheel_joint</leftJoint>
                  <rightJoint>right_wheel_joint</rightJoint>
                  <robotBaseFrame>base_link</robotBaseFrame>
                  <wheelSeparation>0.25</wheelSeparation>
                  <wheelDiameter>0.07</wheelDiameter>
                  <publishWheelJointState>true</publishWheelJointState>
      </plugin>
</gazebo>
```

Estas líneas de código definen que las ruedas traseras son las motrices y de tipo diferencial, mientras que la delantera se queda como una rueda pasiva. Ahora si tenemos gazebo en ejecución con el .launch correspondiente al robot (own_movile_robot_gazebo.launch), creamos otro terminal pulsando Ctrl+Shift+T y escribimos en él:

```
rostopic pub -1 cmd_vel geometry_msgs/Twist   '{linear: {x: 0.0}, angular: {z: 0.3}}'
```

Con este comando estamos escribiendo directamente en el topic cmd_vel, que recordamos que es en el que se define la velocidad lineal y angular del robot. Modificando los valores numéricos que se encuentran entre llaves{} se obtienen diferentes movimientos. En este caso el robot gira sobre sí mismo.

Otra función que nos ofrece ROS es teleoperar el robot mediante el teclado. Para ello primero es necesario descargar el siguiente paquete escribiendo en un terminal:

```
sudo apt-get install ros-kinetic-teleop-twist-keyboard
```

Ahora, con gazebo abierto y el robot cargado, en un terminal ejecutamos la siguiente línea y ya podemos teleoperar el robot desde el teclado:

```
rosrun teleop_twist_keyboard teleop_twist_keyboard.py
```

Conclusiones

URDF permite crear cualquier tipo de robot para ROS. Es una herramienta un poco tediosa pero obtiene muy buenos resultados. En un principio queríamos diseñar un robot como si fuera un coche teledirigido (4 ruedas, delanteras para dirección y traseras para propulsión), pero no supimos hacer que las dos ruedas delanteras giraran a la vez y en el mismo sentido ni encontramos un plugin que controlase los motores de la forma que queríamos, pues la mayoría son diferenciales de dos

ruedas activas. Por ello nos limitamos a hacer el tutorial y el resultado ha sido exitoso como se observa en el vídeo.