ROS2 Apuntes nivel 2

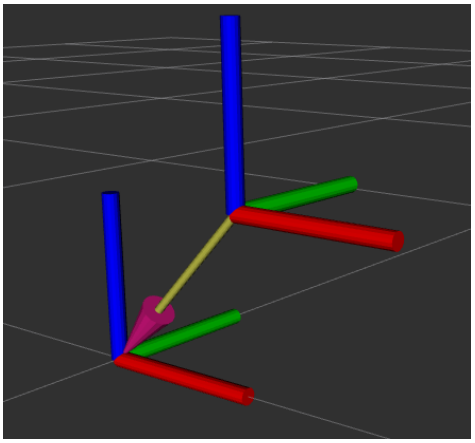
RViz y TF (TransForm)

“RViz” es una herramienta de visualización en 3D utilizada en ROS. A diferencia de Gazebo, que exploraremos más adelante, "RViz" proporciona una representación tridimensional para observar diferentes aspectos del robot.

Dentro de "RViz" puedes interactuar con el modelo del robot. Cada parte rígida del robot se representa como un "enlace" (link), y puedes habilitar o deshabilitar la visualización de estos enlaces según tus necesidades. Además, puedes ajustar la transparencia para obtener una mejor perspectiva.

La información crucial para el correcto funcionamiento de ROS reside en las transformaciones entre los diferentes enlaces del robot. Estas transformaciones, visible en la ventana TF, especifican la posición y orientación relativa de cada link. Los TF se representan como un sistema de ejes 3D, donde el **eje x** corresponde al color rojo, el **eje y** al color verde, y el **eje z** al color azul.

Puedes imaginarlo como las articulaciones del cuerpo humano. Por ejemplo, al observar el brazo, el codo actúa como el eje en el cual se realiza la transformación para el brazo. El TF especifica la posición del antebrazo en relación con el brazo y cómo se mueven entre sí. En el contexto de un robot, el TF se utiliza para describir cómo se posicionan y se mueven las partes rígidas físicas (links) del robot entre sí.



Imágen 1.1: TF en RViz

La flecha que aparece en la imágen 1.1 es la relación que tienen entre sí esos dos TF. En este caso, el TF superior, desde donde sale la flecha es el “hijo” del TF inferior.

En ROS2 hay una herramienta muy útil para conocer estas relaciones de una manera sencilla. Para ello se utiliza el paquete tf2, que suele venir al instalar ros2 pero en caso de que no fuera así, para instalarlo solo debemos escribir en consola:

```
sudo apt install ros-humble-tf2-tools
```

Una vez tenemos tf2 y dejando abierto RViz con el robot que estemos trabajando, escribimos en consola:

```
ros2 run tf2_tools view_frames
```

Esto creará el árbol de relación en un archivo PDF y otro .GV en la ubicación que estuviéramos en la consola al escribir el comando.

¿Qué problema estamos tratando de resolver con TF?

En primer lugar, queremos mantener un árbol estructurado para todas las articulaciones o marcos del robot.

Para cada transformación también tenemos una marca de tiempo, lo que significa que no solo sabemos dónde están las cosas, sino también cuándo, por ejemplo, podrías preguntar: ¿dónde está la cámara en relación con el origen del robot? ¿O dónde está la mano en relación con la cabeza, la rueda o otro robot? ¿Y qué pasó hace dos segundos? Básicamente, lo que queremos saber es cómo se colocan los marcos uno con respecto al otro y también cómo se mueven uno con respecto al otro.

Entonces, ¿qué es exactamente la transformación? Bueno, una transformación es simplemente una traslación más una rotación. Tomas un objeto, lo mueves y lo rotas, ya que estamos en el espacio 3D, entonces tienes tres componentes para la traslación y tres componentes para la rotación. Así que lo que tendrías que hacer es, para cada marco del robot, calcular la transformación con todos los demás marcos del robot.

En lugar de hacerlo nosotros mismos, simplemente vamos a usar la funcionalidad TF de ROS, lo que hará nuestras vidas mucho más fáciles.
¿Cómo creamos TF con ROS? Bueno, en primer lugar, entender TF es lo más importante, porque entonces realmente no necesitas implementar TF directamente. En su lugar, crearás lo que se llama un archivo URDF y usarás paquetes ROS existentes.

¿Qué es URDF?

URDF significa formato unificado de descripción de robot.(Unified Robot Description Format). Los URDF se escriben utilizando XML. La parte más difícil del URDF, y también la más importante, es comprender cómo ensamblar las partes del robot con una articulación que generará una TF.

Primer URDF: Crear y visualizar links

Para crear un archivo urdf debemos darle la extensión "**.urdf**". Aquí hay un ejemplo de un archivo URDF con un link creado:

```
<?xml version="1.0"?>
<robot name="my_robot">
    <link name="base_link">
        <visual>
            <geometry>
                <box size="0.6 0.4 0.2"/>
            </geometry>
            <origin xyz="0 0 0.1" rpy="0 0 0"/>
        </visual>
    </link>
</robot>
```

`<?xml version="1.0"?>`: Esta es la declaración XML que define la versión de XML utilizada, en este caso, 1.0.

2. `<robot name="my_robot">`: Esta etiqueta define el inicio de la descripción del robot. El atributo `name` se utiliza para dar un nombre único al robot, en este caso, "my_robot".

3. `<link name="base_link">`: Esta etiqueta define un enlace en el robot. Un enlace puede ser considerado como una parte rígida del robot. El atributo `name` se utiliza para dar un nombre único al enlace, en este caso, "base_link".

4. `<visual>`: Esta etiqueta se utiliza para definir cómo se visualiza el enlace. Todo lo que está dentro de esta etiqueta afecta sólo a la apariencia del enlace, no a su comportamiento físico.

5. `<geometry>`: Esta etiqueta se utiliza para definir la forma del enlace. Puede contener otras etiquetas que definen formas específicas, como `<box>`, `<cylinder>`, `<sphere>` y `<mesh>`.

6. `<box size="0.6 0.4 0.2"/>`: Esta etiqueta define que el enlace tiene forma de caja. El atributo `size` define las dimensiones de la caja en el orden x, y, z. En este caso, la caja tiene una longitud de 0.6, una anchura de 0.4 y una altura de 0.2.

7. `<origin xyz="0 0 0.1" rpy="0 0 0"/>`: Esta etiqueta define el origen de la forma del enlace en relación con el sistema de coordenadas del enlace. El atributo `xyz` define la posición del origen en el orden x, y, z. El atributo `rpy` define la rotación del origen en torno a los ejes x, y, z (roll, pitch, yaw). En este caso, el origen está desplazado 0.1 en la dirección z y no hay rotación.

El porqué del origin en z es 0.1 es que en RViz, al centrar la vista en el base_link, el grid aparece en medio y la pieza está como enterrada en el suelo. Para solucionarlo, se configura el eje z a la mitad de la altura de la pieza.

Los archivos .urdf se pueden visualizar usando RViz. Para ello en la terminal escribimos, en mi caso:

```
ros2 launch urdf_tutorial display.launch.py
model:=/home/esparza/my_robot.urdf
```

Etiqueta material - añadiendo color

Los materiales se utilizan para dar color y textura a los enlaces de un robot en la visualización.

```
<material name="blue">
    <color rgba="0 0 0.5 1"/>
</material>
```

1. `<material name="blue">`: Esta etiqueta define un material. El atributo `name` se utiliza para dar un nombre único al material, en este caso, "blue". Este nombre puede ser luego referenciado en las etiquetas `<visual>` de los enlaces para aplicar este material.

2. `<color rgba="0 0 0.5 1"/>`: Esta etiqueta define un color para el material. El atributo `rgba` define el color en el espacio de color RGBA (Red, Green, Blue, Alpha). Los valores van de 0 a 1. En este caso, el color es azul con una opacidad

completa. Los primeros tres valores representan el rojo, el verde y el azul, respectivamente. Como los valores para el rojo y el verde son 0, y el valor para el azul es 0.5, el color resultante es azul. El último valor representa la opacidad (alpha), donde 1 significa completamente opaco y 0 significa completamente transparente. En este caso, el valor es 1, por lo que el color es completamente opaco.

Joints

Las articulaciones son las conexiones entre los enlaces en un robot y definen cómo estos enlaces pueden moverse en relación entre sí.

Ejemplo de joint y su explicación:

```
<joint name="base_second_joint" type="fixed">
    <parent link="base_link"/>
    <child link="second_link"/>
    <origin xyz="0 0 0.2" rpy="0 0 0"/>
</joint>
```

1. `<joint name="base_second_joint" type="fixed">`: Esta etiqueta define una articulación. El atributo `name` se utiliza para dar un nombre único a la articulación, en este caso, "base_second_joint". El atributo `type` define el tipo de articulación. En este caso, es "fixed", lo que significa que los dos enlaces conectados por esta articulación no pueden moverse en relación entre sí.
2. `<parent link="base_link"/>`: Esta etiqueta define el enlace padre de la articulación. El atributo `link` se utiliza para referenciar el nombre del enlace padre, en este caso, "base_link".

3. `<child link="second_link"/>`: Esta etiqueta define el enlace hijo de la articulación. El atributo `link` se utiliza para referenciar el nombre del enlace hijo, en este caso, "second_link".

4. `<origin xyz="0 0 0.2" rpy="0 0 0"/>`: Esta etiqueta define el origen de la articulación en relación con el sistema de coordenadas del enlace padre.

Aclaraciones:

- Hay quien define los joint en medio de los dos links a los que hace referencia y otros que ponen todos los links y luego todos los joints. A gusto de cada cual.

- Para el nombre de los joint se suele poner primero el nombre del link padre y luego el del link hijo, seguido de _joint.

Consejo:

Como en cada link o joint hay una etiqueta origin, lo mejor al principio es ponerlas todas a 0 0 0 , 0 0 0 y luego en RViz configurarlo adecuadamente. Primero siempre hacer que los origin de los joint estén correctamente, y solo entonces tocar el origin del link que aun sigue mal.

Documentación sobre joints y links

Pulsar en los enlaces para conocer los atributos y etiquetas disponibles en links y joints.

Ejecutar robot_state_publisher y RViz en la terminal

Hasta ahora se utilizaba RViz, robot_state_publisher y demás usando el paquete urdf-tutorial. Esto está bien para el primer contacto, pero luego no lo usaremos más. En esta sección se explica cómo seguir utilizando todo lo anterior sin necesidad de utilizar dicho paquete.

Para ejecutar robot_state_publisher desde la consola, debemos escribir lo siguiente:

```
ros2 run robot_state_publisher robot_state_publisher –ros-args
-p robot_description:=”$(xacro my_robot.urdf)”
```

Donde “my_robot_urdf” es el archivo que queremos utilizar, pero hay que tener en cuenta la ruta correcta. “xacro” se explica más adelante. Con esto tendremos el nodo en funcionamiento.

Ahora ejecutamos el plugin que permitía ver el movimiento de los joints en RViz:

```
ros2 run joint_state_publisher_gui joint_state_publisher_gui
```

Y para ejecutar RViz:

```
ros2 run rviz2 rviz2
```

Una vez abierto RViz, por defecto no viene con los displays de “RobotModel” y “TF”, hay que añadirlos usando el botón Add. Hecho esto, en “GlobalOptions” hay que seleccionar el “Fixed Frame” adecuado y en “RobotModel” elegir el “Description topic”.

Crear my_robot_description package para instalar URDF

Para ello se crea un paquete como se hace siempre y sin dependencias. Eliminamos las carpetas de include y src ya que no se utilizarán y creamos otra llamada urdf donde irán los archivos con esta extensión.

Ahora para instalar urdf, en el archivo CMakeList.txt añadimos lo siguiente:

```
install(
  DIRECTORY urdf
  DESTINATION share/${PROJECT_NAME}/
)
```

Guardamos, volvemos a compilar el paquete y ya está instalado por el momento.

Crear un launch file para ejecutar RViz, robot_state_publisher y el gui al mismo tiempo

Se va a seguir utilizando un paquete específico para los launcher, que ya está explicado en apuntes anteriores y se da por hecho que ya se sabe hacer todo, incluso su configuración.

En la carpeta launch del paquete my_robot_bringup se crearán los archivos launch. En esta ocasión el launch file se crea usando xml. Es igual de válido usar python o xml indistintamente.

Explicación de “display.launch.xml”

```
<?xml version="1.0"?>
<launch>
```

```
    <let name="urdf_path" value="$(find-pkg-share
my_robot_description)/urdf/my_robot.urdf"/>

    <node pkg="robot_state_publisher"
exec="robot_state_publisher">
        <param name="robot_description" value="$(command
'xacro $(var urdf_path)')"/>
    </node>

    <node pkg="joint_state_publisher_gui"
exec="joint_state_publisher_gui"/>

    <node pkg="rviz2" exec="rviz2" output="screen"/>
</launch>
```

1. `<let name="urdf_path" value="$(find-pkg-share my_robot_description)/urdf/my_robot.urdf"/>`: Esta línea está definiendo una variable `urdf_path` que contiene la ruta al archivo URDF (Unified Robot Description Format) del robot.
2. `<node pkg="robot_state_publisher" exec="robot_state_publisher">`: Esta línea está lanzando un nodo del paquete `robot_state_publisher`.

3. `<param name="robot_description" value="$(command 'xacro $(var urdf_path)')"/>`: Este es un parámetro para el nodo `robot_state_publisher` que especifica la descripción del robot. Se utiliza el comando `xacro` para procesar el archivo URDF.

4. `<node pkg="joint_state_publisher_gui" exec="joint_state_publisher_gui"/>`: Esta línea está lanzando un nodo del paquete `joint_state_publisher_gui`.

5. `<node pkg="rviz2" exec="rviz2" output="screen"/>`: Esta línea está lanzando el nodo `rviz2`, que es una herramienta de visualización 3D para ROS. `output="screen"` significa que la salida del nodo se mostrará en la pantalla.

Hecho esto y compilado, para ejecutar el archivo de lanzamiento, escribimos en consola:

```
ros2 launch my_robot_bringup display.launch.xml
```

En el paquete está también la versión en python, algo más complicada que usando xml por todos los import que necesita sobre todo.

Haciendo el URDF Compatible con Xacro

Xacro es una característica de ROS que te permite limpiar tu urdf y hacerlo más dinámico, permitiendo agregar variables y funciones a tu urdf utilizando las propiedades. Así que si logramos tener algo que ya sea un poco más limpio, dinámico, modular y escalable, será mucho más fácil para nosotros en el futuro.

Para hacer un urdf compatible con xacro:

1. Añadir la extensión .xacro al archivo URDF. Es igual de válido un archivo .xacro que un archivo .urdf.xacro.
2. En la etiqueta raíz, añadir a continuación del nombre lo siguiente:

```
xmlns:xacro="http://www.ros.org/wiki/xacro"
```

Crear variables con xacro property

Xacro incorpora constantes que podemos usar en nuestro xml, como por ejemplo **pi.** También es posible crear nuestras propias variables usando `xacro:property`

```
<xacro:property name="base_length" value="0.6"/>
```

Para poder usar las variables que creemos o las que ya vengan con xacro, hay que usarlas junto con `${ }`

```
<box size="${base_length} 0.4 0.2"/>
```

Crear funciones con xacro macros

Ante todo esto es una herramienta útil pero tampoco hay que usarla todo el tiempo porque sí. En el ejemplo se usa una macro para crear las ruedas la cual tiene sentido si van a ser iguales y más de una, pero para determinadas cosas tampoco es necesario sobreoptimizar de más.

```
<xacro:macro name="wheel_link" params="prefix">
```

```
        <link name="${prefix}_wheel_link">
            <visual>
                <geometry>
                    <cylinder radius="${wheel_radius}"
length="${wheel_length}"/>
                </geometry>
                <origin xyz="0 0 0" rpy="${pi/2.0} 0 0"/>
                <material name="grey"/>
            </visual>
        </link>
    </xacro:macro>

    <xacro:wheel_link prefix="left"/>
    <xacro:wheel_link prefix="right"/>
```

Es pácticamente igual que usar funciones en C++, por ejemplo. Se define la función con nombre y los parámetros si hubiese, dentro de la función se desarrolla la lógica que queramos y posteriormente se llama a esa función. La llamada debe de estar después de la definición o ésta no la encontrará.

Separar xacro files en varios usando include

Para tenerlo todo más legible y modular es posible crear varios archivos .xacro y luego utilizar `xacro:include` para que se comporten como si fueran un único archivo.

En los archivos .xacro que no sean el main, esta sería la plantilla a utilizar:

```
<?xml version="1.0"?>
<robot xmlns:xacro="http://www.ros.org/wiki/xacro">
</robot>
```

Como se puede ver la etiqueta robot no tiene el atributo name. El name solo se utiliza en el archivo que consideremos el principal.

```
<?xml version="1.0"?>
<robot name="my_robot"
xmlns:xacro="http://www.ros.org/wiki/xacro">

    <xacro:include filename="common_properties.xacro"/>
    <xacro:include filename="mobile_base.xacro"/>
</robot>
```

Instalación de Ignition gazebo fortress

La instalación de ignition gazebo se hace siguiendo los pasos de esta página.
Los tutoriales para empezar a utilizar gazebo también se encuentran disponibles desde la misma web.

Por lo visto hay que utilizar una combinación específica para que sean compatibles entre sí las versiones. Lo recomendado es:

- Ubuntu Jammy 22.04
- ROS 2 Humble Hawksbill
- Gazebo Fortress

Para iniciar gazebo fortress con interfaz gráfica: `ign gazebo` y aparecerá un menú. También puedes entrar directamente a la simulación con

```
ign gazebo nombreSim.sdf
```

Añadir la inercia en los archivos URDF

La inercia en ROS2, y en robótica en general, se refiere a la resistencia de un objeto a cualquier cambio en su velocidad. Esto incluye cambios en la velocidad de rotación y en la velocidad de traslación. En el contexto de ROS2 y la descripción de robots, la inercia se utiliza para describir cómo se moverá un objeto en respuesta a las fuerzas aplicadas.

Ejemplo de una macro para calcular la inercia de un cilindro:

```
<xacro:macro name="cylinder_inertia" params="m r h xyz rpy">
 <inertial>
   <origin xyz="${xyz}" rpy="${rpy}"/>
   <mass value="${m}"/>
   <inertia ixx="${(m/12) * (3*(r*r) + h*h)}" ixy="0" ixz="0"
            iyy="${(m/12) * (3*(r*r) + h*h)}" iyz="0"
            izz="${(m*(r*r)) / 2}"/>
 </inertial>
</xacro:macro>
```

En el código anterior, se toman parámetros como la masa del objeto (`m`), sus dimensiones (`h` `r` para el cilindro), y la posición y orientación del centro de masa (`xyz`, `rpy`), y se calculan los momentos de inercia correspondientes.

Los momentos de inercia son valores que describen cómo la masa de un objeto está distribuida en relación a los ejes de rotación. Estos valores son importantes para la simulación precisa del movimiento del robot en un entorno de simulación como Gazebo.

Las fórmulas para calcular la inercia de figuras en 3D se pueden conseguir, por ejemplo, en el siguiente enlace de la wikipedia. También es conveniente consultar la documentación de ROS que explica cómo añadir la inercia y la colisión.

Si se utilizan meshes, lo mejor para obtener la inercia es importarla directamente desde el software CAD donde creas la pieza.

Añadir la colisión en los archivos URDF

La colisión se suele añadir después de la etiqueta de cierre </visual> de cada link. Para estas formas simples bastaría con añadir la misma <geometry> y <origin> que la parte visual del link, como se ve en el siguiente código de las ruedas:

```
<collision>
  <geometry>
    <cylinder radius="${wheel_radius}"
              length="${wheel_length}"/>
  </geometry>
 <origin xyz="0 0 0" rpy="${pi/2.0} 0 0"/>
</collision>
```

La cosa cambia para los meshes. En estos casos se suele simplificar la geometría de la colisión ya que consume demasiados recursos si queremos hacerlo totalmente preciso a la forma de la figura. Por lo tanto si utilizamos meshes, para la colisión usaremos las dimensiones de una figura simple como las cajas, cilindros o esferas.

Spawnear URDF en Gazebo Fortress

Este proceso está bastante bien explicado en la documentación de Gazebo Fortress sobre spawnear URDF.

Si se utiliza Xacro primero hay que convertirlo a urdf usando el paquete de xacro, aquí hay un ejemplo de como se ha convertido el archivo .urdf.xacro que se está usando en los ejemplos:

```
ros2 run xacro xacro
/home/esparza/ROS2_nivel2/ros2_ws_lvl2/src/my_robot_descriptio
n/urdf/my_robot.urdf.xacro > my_robot.urdf
```

Una vez tenemos el archivo URDF ya podemos spawnear el robot, lo mejor es ir a la documentación y leerla bien. Dejo un ejemplo de cómo se spawnea el urdf anterior:

```
ign service -s /world/empty/create --reqtype
ignition.msgs.EntityFactory --reptype ignition.msgs.Boolean
--timeout 1000 --req 'sdf_filename:
"/home/esparza/ROS2_nivel2/ros2_ws_lvl2/src/my_robot_descripti
on/urdf/my_robot.urdf", name: "urdf_model"'
```

Uso de plugins para mover el robot

Ver el vídeo sobre cómo usar plugins para mover el robot.

Observaciones de la conversión de urdf a sdf

Por lo que he visto (no sé si siempre es así), los links que estén conectados a un joint “fixed”, al convertir el urdf a sdf los trata como `<frame>`. Estos frames no tienen propiedades físicas, pero están `attached_to` a un joint o link. Si sigues quién está conectado a quién, al final llegas a un link o joint que sí está definido el en sdf. Es ahí donde debes buscar y añadir las propiedades que necesites.

Comento esto ya que la esfera de apoyo tenía fricción y el robot pegaba botes al avanzar. En el urdf he indicado la fricción del `caster_wheel_link`, pero al spawnear el robot en un mundo (que añade al sdf el robot desde un urdf) no ha incluido esta propiedad ya que ha tratado al link como un frame. Si sigues los attached llegas a un <link>, y ese link tiene una etiqueta de collision, en este caso en concreto era esta:

‹collision name='base_footprint_fixed_joint_lump__caster_wheel_link_collision_1'›

Es ahí donde tienes que añadir las propiedades que no se hayan convertido desde el urdf y funcionará correctamente.

Uso de sensores en Ignition Fortress

En la documentación explica cómo usar sensores, vale la pena seguir el tutorial ya que todo lo que encuentro son ejemplos en gazebo classic y pocos de ignition.