

Universidad de Guadalajara
Centro Universitario de Ciencias Exactas e
Ingenierías



Alumno: José de Jesús Fernández García

Código: 214758062

Carrera: INNI

Dependencia: CUCEI / Departamento de Ciencias Computacionales

Programa: Apoyo Laboratorio de Sistemas Inteligentes

Receptor: Mtra. María Elena Romero Gastelu

Profesor: Dr. Carlos Alberto Lopez Franco

Reporte 3 26-07-2021 - 26-09-2021

INDICE DE CONTENIDO

INTRODUCCION	4
DESARROLLO	6
Espacio de trabajo ROS con Docker	6
Configuración de espacio de trabajo ROS usando Docker	6
URDF	7
Introduccion	7
Paquetes necesarios para las practicas	10
Construccion de un modelo de robot visual con URDF desde cero utilizando RViz como visualizador	15
Introduccion a Xacro	57
Construccion de un modelo de robot visual utilizando Xacro	61
Introduccion a la robotica	62
Introduccion	62
Partes de un robot	64
Cinematica del robot	65
¿Que sigue?	94
Gazebo	97
Introduccion	97
Instalacion de Gazebo con integracion ROS	98
Uso de URDF en Gazebo	103
Complementos Gazebo	111
Comunicacion ROS con Gazebo	124
RViz - Gazebo	126
Introduccion	126
Controladores	126
Paquete “ros_control”	135
Uso de “ros_control” en Gazebo usando modelo de robot en URDF	147
Modificando una fabrica (simulada) en URDF	174
Paquetes necesarios	174
Visualizacion de la fabrica	176
Modificaciones a la fabrica	179
¿Que se podria hacer?	184
Navegacion autonoma	186
Introduccion a TurtleBot	186
Instalacion de TurtleBot3	187
Control del TurtleBot3	194
Uso de RViz para visualizar lo que esta haciendo el robot	199

¿Que es necesario para la navegacion?	203
Teoria del mapeo	203
Mapeo con Turtlebot	204
Teoria de la localizacion	205
Localizacion con Turtlebot	205
Pila de navegacion ROS	215
Planificacion de rutas	216
Otros mundos del TurtleBot	221
Material adicional	223
Repasso y material utilizado	224
Repasso	224
Material	225
CONCLUSION	226

INTRODUCCION

Algo del trabajo realizado es lo siguiente:

- Material de apoyo:
 - <https://www.edx.org/es/course/hello-real-world-with-ros-robot-operating-system>
Es un curso bastante bueno que me sirvió de apoyo para aprender los fundamentos de ROS más afondo.
 - <https://classic.gazebosim.org/tutorials/browse>
 - <http://wiki.ros.org/ROS/Tutorials>
- Se introdujo como modelar un mundo de tal manera que ROS pueda interactuar en él, ya que modelar un robot y su entorno es una parte importante de la creación de una aplicación ROS exitosa.
- Se hablo sobre el formato de archivo URDF (Descripción Unificada de Robot) y se aprendio a usarlo. Se trataron conceptos como enlaces, uniones y geometrías (links, joints, geometry).
- Se creo un modelo de robot visual con URDF desde cero (en general, es el robot R2-D2, personaje de Star Wars).
- Se hablo sobre RViz, el cual es una herramienta de visualización 3D para ROS.
- Se hablo sobre Gazebo, el cual es un simulador de robótica 3D de código abierto (Unity también es un simulador).
- Se hablo sobre los tipos de articulaciones, ya que se utilizan a la hora de crear nuestro modelo de robot con URDF (estas articulaciones pueden ser *fixed*, *continuous*, *revolute*, *prismatic*, *planar* y *floating*). Recordemos que en un robot se ven involucradas muchos tipos de articulaciones (de ruedas, de cabeza, de pinza, de brazo, etc.).
- Se hablo sobre cómo agregar propiedades físicas básicas a nuestros modelos URDF (así como de inercia, coeficientes de contacto, dinámica conjunta, etc.), ya que para que un modelo se simule correctamente se debe definir varias propiedades físicas del robot.
- Una vez introducido lo que es URDF, se hablo de Xacro, el cual nos sirve para limpiar un archivo URDF, es decir, para reducir la cantidad de código en un archivo URDF (es el siguiente paso al modelado de robot, ya que nos facilita aun más la tarea de modelar robots).

- Se aprendio a simular un modelo de robot creado en URDF en Gazebo.
Recordemos que en el tutorial de “pick and place” se utilizó un modelo de robot en URDF que se simulo en Unity, por lo que podemos simular modelos de robot tanto en Unity como en Gazebo. Pero ahora se aprenderá a simular un modelo de robot en Gazebo, ya que es un simulador más fácil de usar que Unity y es más fácil de utilizar en un entorno de trabajo ROS.
- Se hablo sobre la relación que existe entre URDF y SDF (Formato de Descripción de Simulación, el cual es un formato XML que describe objetos y entornos para simuladores, visualización y control de robots).
- Se hablo sobre los complementos Gazebo, ya que son parte fundamental para que el robot sea interactivo con nosotros y ROS.
Los complementos de Gazebo brindan a nuestros modelos URDF una mayor funcionalidad, los cuales pueden vincular mensajes ROS y llamadas de servicio para, por ejemplo, la salida de un sensor y la entrada de un motor (los complementos que se pueden utilizar son *World*, *Model*, *Sensor*, *System*, *Visual* y *GUI*, pero no en todos se puede hacer referencia a través de un archivo URDF).
- Se hablo sobre cómo usar RViz para monitorear el estado de nuestro robot simulado mediante la publicación directamente desde Gazebo.
- Se hablo sobre controladores (los cuales sirven para activar las articulaciones de nuestro robot), trasnmisiones (son los elementos encargados de transmitir el movimiento desde los actuadores hasta las articulaciones), interfaces (cada controlador tiene su respectiva interfaz), entre otros.
Cuando se trata de robots, debemos asegurarnos de que su movimiento este controlado, es decir, que las acciones no sean ni demasiado lentas ni demasiado rápidas, y que sigan la trayectoria especificada. En este tema se hablará un poco sobre los controladores PID.
- Se aprendio a usar un robot móvil llamado “TurtleBot” (en específico el “TurtleBot3”, ya que el “TurtleBot2” no me funciono muy bien).
Se abarco el tema de la navegación utilizando el robot móvil “TurtleBot” (así como lo que es el mapeo, la localización, planificación de trayectorias, entre otros).

DESARROLLO

Se trabajó en un sistema operativo Linux (en una distribución Ubuntu 20.04).

NOTA: Consultar el apartado “Material” para consultar el material utilizado.

Espacio de trabajo ROS con Docker

URL del curso de ROS que se tomó como apoyo:

<https://www.edx.org/es/course/hello-real-world-with-ros-robot-operating-system>

Configuración de espacio de trabajo ROS usando Docker

Lo primero que haremos será construir la imagen de ROS Docker mediante el archivo Dockerfile que se ha proporcionado ejecutando los siguientes comandos:

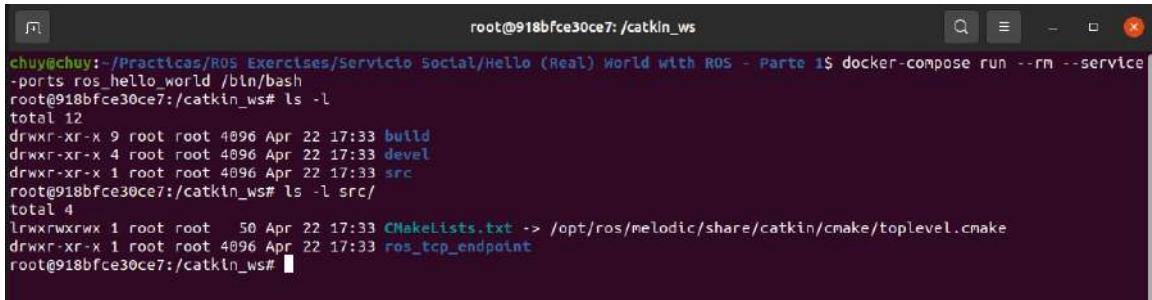
NOTA: Es necesario estar ubicados fuera del directorio.

- `docker build -t chuy7/ros_hello_world:test_practice2 -f ros_docker/Dockerfile .`

El Dockerfile proporcionado utiliza la imagen base melodica de ROS (ros:melodic-ros-base) que se puede encontrar en **Docker Hub** (Docker Hub es un repositorio publico en la nube para distribuir los contenedores).

Ahora iniciamos un contenedor Docker de la imagen recién construida utilizando **docker-compose**. Para esto necesitamos crear un archivo llamado “docker-compose.yaml”, similar al que se presenta en el material proporcionado (cabe destacar que el archivo “docker-compose.yaml” que se presenta en el material es el archivo final, ya que se estuvo modificando en varias ocasiones). Despues ejecutamos el siguiente comando:

- `docker-compose run --rm --service-ports ros_hello_world /bin/bash`



```
root@918bfce30ce7:/catkin_ws
chuy@chuy:~/Practicas/ROS Exercises/servicio Social/Hello (Real) World with ROS - Parte 1$ docker-compose run --rm --service
-ports ros_hello_world /bin/bash
root@918bfce30ce7:/catkin_ws# ls -l
total 12
drwxr-xr-x 9 root root 4096 Apr 22 17:33 build
drwxr-xr-x 4 root root 4096 Apr 22 17:33 devel
drwxr-xr-x 1 root root 4096 Apr 22 17:33 src
root@918bfce30ce7:/catkin_ws# ls -l src/
total 4
lrwxrwxrwx 1 root root    50 Apr 22 17:33 CMakeLists.txt -> /opt/ros/melodic/share/catkin/cmake/toplevel.cmake
drwxr-xr-x 1 root root 4096 Apr 22 17:33 ros_tcp_endpoint
root@918bfce30ce7:/catkin_ws#
```

Hasta aquí el espacio de trabajo ROS ya está listo para aceptar comandos.

URDF

Introducción

Modelar un robot y su entorno es una parte importante de la creación de una aplicación ROS.

ROS nos facilita la creación de descripciones o modelos suficientemente detallados de los robots y sus entornos utilizando un formato de archivo llamado “Descripción Unificada de Robot”, o URDF.

URDF es el formato que usa ROS para almacenar modelos de robots.

¿Qué es realmente URDF?

Es un lenguaje de modelado *específico de dominio* (es *específico de dominio* porque utiliza nombres y terminología específicos de los dominios de robótica y modelado de robots) basado en XML, que nos permite codificar la cinemática (es la rama de la física que estudia los movimientos y estados en que se encuentran los cuerpos), algunas partes de la dinámica (es la parte de la física que estudia la relación entre el movimiento y las causas que lo producen (las fuerzas)) y varias otras piezas de metadatos relacionados con un robot en un formato legible.

URDF es el formato de archivo que usa ROS para describir el diseño del cuerpo de un robot con todos sus enlaces, articulaciones, formas y colores.

Tambien nos permite almacenar informacion adicional como:

- El rango del movimiento de todas las articulaciones del robot
- La rapidez con la que pueden realizarse esos movimientos

Sobre la implementacion:

Un archivo URDF es solo un archivo de texto que contiene etiquetas XML (palabras clave especificas que ROS reconoce como parte de URDF).

Algunas de esas etiquetas XML se refieren a otros archivos o modelos 3D de partes de robot, lo que nos permite hacer uso de archivos externos.

Contenido de un URDF:

La mayor parte de un archivo URDF se compone de elementos llamados:

- <link>s
Dan forma a los robots (tambien conocidos como partes).
- <joint>s
Conectan enlaces y determinan como pueden moverse entre si.

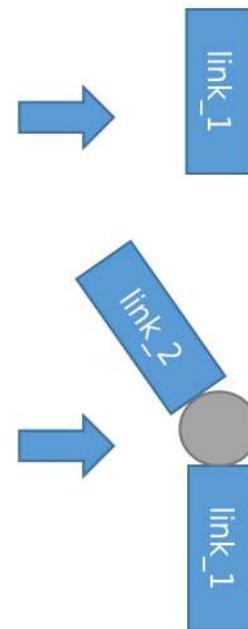
Por lo que **un robot es un conjunto de enlaces conectados por articulaciones en un cierto orden.**

Veamos un pequeno ejemplo de URDF:

- robot1.urdf

```
<robot name="robot">
    <link name="link_1"/>
</robot>
```
- robot2.urdf

```
<robot name="robot">
    <link name="link_1"/>
    <link name="link_2"/>
    <joint name="joint_1" type="<type>">
        <parent link="link_1" />
        <child link="link_2" />
    </joint>
</robot>
```



NOTA: En el tipo de articulacion no se coloco nada porque aun no se han visto los tipos de articulacion, pero es necesario, para todas las etiquetas <joint>, especificar el tipo de articulacion (es obligatorio).

Aunque solo hay un tipo de enlace (link), hay hasta 6 tipos de articulaciones (joints) compatibles con URDF:

- *Fixed (articulaciones fijas)*
Conectan rigidamente el padre con el hijo, haciendo imposible el movimiento (realmente no es una articulacion, ya que no permite el movimiento).
- *Revolute (articulaciones angulares)*
Permiten la rotacion del hijo con respecto al padre, pero solo en una dimension. Tienen un rango limitado especificado.
- *Continuous (articulaciones continuas)*
Son iguales que la articulacion “Revolute” pero sin limites. Pueden rotar una cantidad infinita de veces.
- *Prismatic (articulaciones prismaticas)*
Solo permiten la translacion del hijo con respecto al padre en una dimension. Tienen un rango limitado especificado.
- *Planar (articulaciones planas)*
Son variantes bidimensionales de las articulaciones “Prismatic”. Permite el movimiento en un plano perpendicular al eje.
- *Floating (articulaciones flotantes)*
Permiten el movimiento y la rotacion en todas las direcciones sin limite alguno. Permiten el movimiento de los 6 grados de libertad.

Las articulaciones se definen en terminos de parente e hijo.

Por tanto, un archivo URDF no es mas que un archivo donde viene descrito como es fisicamente el robot con el que estamos trabajando.

Como trabajamos con ROS, deberemos meter este archivo URDF, que explica la estructura de nuestro robot, dentro del servidor de parametros para que pueda ser interpretado por ROS (con nombre de parametro “**robot_description**”).

Los archivos URDF pueden venir en formato **.urdf** o en formato **.xacro** (el cual es uno de los mas utilizados).

Paquetes necesarios para las practicas

Lo primero que haremos sera ejecutar los siguientes comandos:

rosdep update

apt-get update

Despues instalamos los siguientes paquetes:

- *rviz*

Comando a ejecutar:

sudo apt-get install ros-melodic-rviz

Para poder ejecutar RViz desde un contenedor debemos hacer lo siguiente:

- Debemos hacer algunas modificaciones al archivo “docker-compose.yaml” con el que lanzamos nuestro contenedor ROS (el archivo con las modificaciones se encuentra en el material proporcionado).
- Antes o despues de iniciar un contenedor necesitamos ejecutar el siguiente comando:

xhost +/local:docker

NOTA: Es muy importante, ya que si no se ejecuta este comando no se nos permitira abrir RViz desde un contenedor Docker.

El comando “xhost” permite a un usuario dar permisos a otros para que puedan sacar ventanas en su pantalla. Solo el propietario de la pantalla puede cambiar los permisos con el comando “xhost”.

De esta manera ya podemos abrir RViz desde un contenedor Docker:

roscore &

rosrun rviz rviz

The terminal window shows the output of the command `roslaunch rviz rviz`. The log messages indicate the version of rviz (1.13.24), Qt (5.9.5), and OGRE (1.9.0). It also mentions OpenGL support and device information.

The RViz application window titled "default.rviz - RViz" is displayed. The left panel shows "Displays" configuration, including "Global Options" (Fixed Frame: map, Background Color: #48:48:48, Frame Rate: 30, Default Light checked), "Global Status" (WARN), and "Grid". The main view shows a 3D coordinate system grid. The right panel shows "Views" configuration for the current view "Orbit (rviz)" with parameters: Near Clip Dist: 0.01, Inverse Far Dist: 1000000000000000000.0, Target Frame: <Fixed Frame>, Distance: 10, Focal Shape...: 0.05, Focal Shape...: checked, Yaw: 0.785398, Pitch: 0.785398, and Focal Point: (0.0, 0.0, 0.0).

- *joint_state_publisher_gui*

Comando a ejecutar:

NOTA: El siguiente comando instala los paquetes "joint_state_publisher" y "joint_state_publisher_gui".

sudo apt install ros-melodic-joint-state-publisher-gui

- *robot_state_publisher*

Comando a ejecutar:

sudo apt-get install ros-melodic-robot-state-publisher

- *xacro*

Comando a ejecutar:

sudo apt-get install ros-melodic-xacro

Tener en cuenta que todos los paquetes que se instalan mediante el comando "apt-get install" se instalan en el directorio:

/opt/ros/melodic/share

```
root@chuy:/catkin_ws# rospack list | grep -E 'rviz|joint_state|robot_state|xacro'
joint_state_publisher /opt/ros/melodic/share/joint_state_publisher
joint_state_publisher_gui /opt/ros/melodic/share/joint_state_publisher_gui
robot_state_publisher /opt/ros/melodic/share/robot_state_publisher
rviz /opt/ros/melodic/share/rviz
xacro /opt/ros/melodic/share/xacro
root@chuy:/catkin_ws#
```

Observemos las variables de entorno relacionadas con ROS:

```
root@chuy:/catkin_ws# printenv | grep ROS
ROS_WORKSPACE=/catkin_ws
ROS_ETC_DIR=/opt/ros/melodic/etc/ros
ROS_ROOT=/opt/ros/melodic/share/ros
ROS_MASTER_URI=http://localhost:11311
ROS_VERSION=1
ROS_PYTHON_VERSION=2
ROS_PACKAGE_PATH=/catkin_ws/src:/opt/ros/melodic/share ←
ROS_LISP_PACKAGE_DIRECTORIES=/catkin_ws/devel/share/common-lisp
ROS_DISTRO=melodic
root@chuy:/catkin_ws#
```

Como podemos ver tenemos dos directorios donde podemos almacenar paquetes ROS:

- El directorio que ya conocemos, el cual se encuentra en nuestro espacio de trabajo ROS (en donde hemos creado paquetes desde cero).
- El directorio que acabamos de mencionar en donde se instalan los paquetes cuando utilizamos el comando “apt-get install”.

Creamos un paquete llamado “urdf_tutorial” en donde tendremos todo lo relacionado a URDF (asi como lo necesario para visualizar nuestro modelo de robot en RViz). Para ello ejecutamos los siguientes comandos:

```
cd $ROS_WORKSPACE/src
catkin_create_pkg urdf_tutorial rospy roscpp joint_state_publisher
                joint_state_publisher_gui robot_state_publisher rviz xacro
cd src/urdf_tutorial
mkdir launch rviz urdf
```

Mas adelante se iran poblando estas carpetas de archivos que se iran creando.

¿Que es “joint_state_publisher”, “joint_state_publisher_gui” y “robot_state_publisher”?

Todos son paquetes, pero veremos a continuacion que

URDF es, en ultima instancia, una estructura de arbol con un enlace raiz.
Lo que significa que la posicion del hijo depende de la posicion del padre.

funcion tienen:

- *joint_state_publisher*
Lee el parametro “robot_description” del servidor de parametros y obtiene la posicion de todos las articulaciones no fijas del robot y publica los valores de estado de las articulaciones de cada articulacion en el tema “joint_states” (sensor_msgs/JointState).
- *joint_state_publisher_gui*
Contiene una herramienta GUI para establecer y publicar valores de estado de articulaciones para un URDF determinado.
- *robot_state_publisher*
Obtiene la informacion de:
 - Un archivo URDF (el que se encuentra en el servidor de parametros con el nombre de “robot_description”)
 - Los valores que publica “joint_state_publisher” en el tema “joint_states”

Con esa informacion es capaz de sacar la posicion real del robot.

Siempre que queramos que un robot complete una tarea especifica (por ejemplo, moverse una cierta distancia en un entorno, recoger un objeto, etc.), tenemos que tener una forma de saber la posicion y la velocidad de cada articulacion. El “joint_state_publisher” hace exactamente eso (realiza un seguimineto de la posicion de cada articulacion de un robot y publica los valores).

Despues, el “robot_state_publisher” toma dos entradas principales (el modelo de robot y la informacion del tema donde se publica la posicion de cada articulacion del robot) y con esa informacion genera la posicion y orientacion de cada *marco de coordenadas del robot* y publica estos datos en el paquete **tf2**.

Los marcos de coordenadas y la capacidad de transformar datos de un marco de coordenadas a otro son muy importantes para una navegacion autonoma precisa.

En ROS se especifican varios marcos de coordenadas, entre ellos:

- *map (marco de coordenadas fijo en el mundo)*

Este marco tiene su origen en algun punto del mundo elegido arbitrariamente.

- *odom (marco de coordenadas fijo en el mundo)*
Este marco tiene si su origen en el punto donde se inicializa el robot.
- *base_link (marco de coordenadas no fijo)*
Este marco tiene su origen directamente en el centro del robot (este marco se mueve a medida que se mueve el robot).
- Entre otros

Cada uno tiene su origen en algun punto (ya sea del robot o del mundo) y puede ser fijo o no fijo.

Afortunadamente, ROS tiene un paquete llamado “tf2” que sirve para manejar todas estas transformaciones de coordenadas por nosotros.

El paquete “tf2” es responsable de realizar un seguimiento de la posicion y la orientacion de todos los marcos de coordenadas de un robot a lo largo del tiempo. Por tanto, podemos en cualquier momento consultar el paquete “tf2” para averiguar la posicion y la orientacion de cualquier marco de coordenadas en relacion con otro marco de coordenadas.

Podemos tener:

- Marcos de coordenadas que no cambian entre si a lo largo del tiempo.
- Marcos de coordenadas que cambian entre si a lo largo del tiempo.

NOTA: En general, todos son lanzados en un archivo de lanzamiento junto con RViz para la visualizacion del modelo de robot.

Construccion de un modelo de robot visual con URDF desde cero utilizando RViz como visualizador

Lo que faremos sera crear varios archivos .urdf en los que iremos agregando mas y mas componentes a cada archivo hasta llegar al modelo de robot que deseamos construir (es el robot R2-D2 de star wars).

Para empezar crearemos los siguientes archivos:

- *display.launch* (en la carpeta “launch”)

```
<launch>
    <!-- Declaramos un argumento para poder guardar el nombre del archivo URDF que queremos cargar -->
    <arg name="model" default="$(find urdf_tutorial)/urdf/01-myfirst.urdf"/>
    <!-- Declaramos un argumento para especificar si deseamos usar la GUI del paquete "joint_state_publisher" -->
    <arg name="gui" default="true"/>
    <!-- Declaramos un argumento para indicar el archivo de configuracion de pantalla (.rviz) para cargar en RViz -->
    <arg name="rvizconfig" default="$(find urdf_tutorial)/rviz/urdf.rviz" />

    <!-- Declaramos un argumento que tendra la descripcion de nuestro modelo -->
    <param name="robot_description" command="$(find xacro)/xacro $(arg model)" />

    <!--
        if/unless
        | - if=value      -> Si el valor se evalua como verdadero, se incluye la etiqueta y su contenido.
        | - unless=value  -> Si el valor se evalua como false, se incluye la etiqueta y su contenido.
    -->
    <!-- Iniciamos el nodo "joint_state_publisher", ya sea con la GUI o sin la GUI -->
    <node if="$(arg gui)" name="joint_state_publisher" pkg="joint_state_publisher_gui" type="joint_state_publisher_gui" />
    <node unless="$(arg gui)" name="joint_state_publisher" pkg="joint_state_publisher" type="joint_state_publisher" />
    <!-- Iniciamos el nodo "robot_state_publisher" -->
    <node name="robot_state_publisher" pkg="robot_state_publisher" type="robot_state_publisher" />
    <!-- Iniciamos el nodo "rviz" y le indicamos un archivo de configuracion de pantalla (.rviz) para cargar en RViz -->
    <node name="rviz" pkg="rviz" type="rviz" args="-d $(arg rvizconfig)" required="true" />
</launch>
```

Siempre al lanzar este archivo de lanzamiento se abriran dos ventanas:

- joint_state_publisher_gui

Aqui podremos editar los valores de las articulaciones no fijas de nuestro modelo de robot.

- RViz

Es la ventana principal de RViz en la cual se muestra el modelo de robot que carguemos a RViz en una visualizacion en tres dimensiones y con un menu que ofrece bastante informacion con respecto al modelo.

Informacion relacionada a ROS a la hora de lanzar el archivo de lanzamiento:

```
root@chuy:/catkin_ws/src/urdf_tutorial/urdf# rosnode list
/joint_state_publisher
/robot_state_publisher
/rosout
/rviz
root@chuy:/catkin_ws/src/urdf_tutorial/urdf# rostopic list
/joint_states
/rosout
/rosout_agg
/tf
/tf_static
root@chuy:/catkin_ws/src/urdf_tutorial/urdf# rostopic info /joint_states
Type: sensor_msgs/JointState
```

Publishers:
* /joint_state_publisher (http://chuy:35819/)

Subscribers:
* /robot_state_publisher (http://chuy:42303/)

```
root@chuy:/catkin_ws/src/urdf_tutorial/urdf# █
```

```
root@chuy:/catkin_ws/src/urdf_tutorial/urdf# rosnode info /rviz
```

Node [/rviz]
Publications:
* /rosout [rosgraph_msgs/Log]

Subscriptions:
* /tf [tf2_msgs/TFMessage]
* /tf_static [tf2_msgs/TFMessage]

Services:
* /rviz/get_loggers
* /rviz/load_config
* /rviz/reload_shaders
* /rviz/save_config
* /rviz/set_logger_level

```
contacting node http://chuy:36765/ ...
Pid: 30644
Connections:
* topic: /rosout
  * to: /rosout
    * direction: outbound (45271 - 192.168.3.9:46378) [14]
    * transport: TCPROS
* topic: /tf
  * to: /robot_state_publisher (http://chuy:42303/)
    * direction: inbound (56336 - chuy:60167) [24]
    * transport: TCPROS
* topic: /tf_static
  * to: /robot_state_publisher (http://chuy:42303/)
    * direction: inbound (56338 - chuy:60167) [25]
    * transport: TCPROS
```

```
root@chuy:/catkin_ws/src/urdf_tutorial/urdf# █
```

```

root@chuy:/catkin_ws/src/urdf_tutorial/urdf# rostopic info /tf
Type: tf2_msgs/TFMessage

Publishers:
* /robot_state_publisher (http://chuy:42303/)

Subscribers:
* /rviz (http://chuy:36765/)

root@chuy:/catkin_ws/src/urdf_tutorial/urdf# rostopic info /tf_static
Type: tf2_msgs/TFMessage

Publishers:
* /robot_state_publisher (http://chuy:42303/)

Subscribers:
* /rviz (http://chuy:36765/)

root@chuy:/catkin_ws/src/urdf_tutorial/urdf# ■

```

```

root@chuy:/catkin_ws/src/urdf_tutorial/urdf# rosnode list
/joint_state_publisher/get_loggers
/joint_state_publisher/set_logger_level
/robot_state_publisher/get_loggers
/robot_state_publisher/set_logger_level
/rosout/get_loggers
/rosout/set_logger_level
/rviz/get_loggers
/rviz/load_config
/rviz/reload_shaders
/rviz/save_config
/rviz/set_logger_level
root@chuy:/catkin_ws/src/urdf_tutorial/urdf# ■

```

Con esta informacion podemos deducir lo siguiente (la comunicacion entre los nodos):

- Anteriormente hablamos de los nodos “joint_state_publisher” y “robot_state_publisher”, los cuales uno depende del otro, y como podemos observar, ambos estan ejecutandose.
 - En el tema “/joint_states” publica el nodo “joint_state_publisher” y el nodo “robot_state_publisher” se suscribe a dicho tema.
 - El nodo “rviz” esta suscrito a los temas “/tf” y “/tf_static”, en donde publica el nodo “robot_state_publisher”.
- Con respecto al tipo de mensaje (tf2_msgs/TFMessage) de ambos temas:

```

root@chuy:/catkin_ws/src/urdf_tutorial/urdf# rosmsg show tf2_msgs/TFMessage
geometry_msgs/TransformStamped[] transforms
  std_msgs/Header header
    uint32 seq
    time stamp
    string frame_id
    string child_frame_id
  geometry_msgs/Transform transform
    geometry_msgs/Vector3 translation
      float64 x
      float64 y
      float64 z
    geometry_msgs/Quaternion rotation
      float64 x
      float64 y
      float64 z
      float64 w
root@chuy:/catkin_ws/src/urdf_tutorial/urdf# █

```

Recordemos que **tf2** nos permite realizar un seguimiento de multiples marcos de coordenadas a lo largo del tiempo. Tambien permite transformar puntos, vectores, etc. entre dos marcos de coordenadas cualesquiera en cualquier momento deseado.

- Por tanto, como podemos observar, cada nodo tiene relevancia en otro.
- *urdf.rviz* (en la carpeta “rviz”)

Este es un archivo de configuracion de pantalla que se cargara en RViz (es una configuracion inicial a la hora de que se inicie RViz, esto con el objetivo de que se visualize nuestro modelo de robot que se encuentra cargado en el servidor de parametros con el nombre de “robot_description”).

Antes de continuar, mostraremos las etiquetas mas usadas (ya que es un formato XML) a la hora de crear un archivo URDF (las cuales son muy importantes comprender):

Una articulacion se define en terminos de padre e hijo.

- *<robot>*

El elemento raiz en un archivo de descripcion de robot debe ser *<robot>*, y todos los demas elementos deben estar encapsulados dentro.

Elementos:

- *<link>*
- *<joint>*
- *<transmission>*

- <gazebo>

Atributos:

- name

- <sensor>

Describe las propiedades basicas de un sensor visual (es decir, camara, etc.)

Elementos:

- <parent>
- <origin>

Atributos:

- name
- update_rate

- <link>

Describe un cuerpo rigido con inercia, caracteristicas visuales y propiedades de colision.

Elementos:

- <inertial>
- <visual>
- <collision>

Atributos:

- name

- <joint>

Describe la cinematica y la dinamica de la articulacion y tambien especifica los limites de seguridad de la articulacion.

Elementos:

- <origin>
- <parent>
- <child>
- <axis>
- <calibration>
- <dynamics>
- <limit>
- <mimic>
- <safety_controller>

Atributos:

Cuando un cuerpo rigido sigue un movimiento de translacion, la resistencia a toda modificacion de su movimiento es llamada **inercia** (se trata de su masa). Para un cuerpo rigido en rotacion, esta resistencia a toda modificacion de su estado es llamada su **momento de inercia**.

Cinematica

La cinematica del robot estudia el movimiento que realiza este con respecto a un sistema de referencia y sin considerar las fuerzas que intervienen.

Dinamica

En cambio, la dinamica del robot estudia la relacion entre las fuerzas que actuan sobre un cuerpo y el movimiento que se produce.

- *name*
- *type*
- *<transmission>* (es una extension del URDF)

Describe la relacion entre un actuador y una articulacion (multiples actuadores pueden estar vinculados a multiples articulaciones a traves de una transmision compleja).

Elementos:

- *<type>*
- *<joint>*
- *<actuator>*

Atributos:

- *name*

- *<gazebo>* (es una extension del URDF)

Se utiliza con fines de simulacion en el simulador Gazebo.

Para usar un archivo URDF en Gazebo, se deben agregar algunas etiquetas adicionales especificas de la simulacion para que funcione correctamente en Gazebo (Gazebo convierte el URDF a SDF, ya que Gazebo trabaja con archivos SDF).

- *<model_state>*

Describe el estado basico de un modelo URDF.

URDF vs. SDF

Debemos tener en cuenta que URDF especifica un robot, pero SDF tambien especifica un mundo para que viva el robot, es decir, SDF esta diseñado para representar un superconjunto de todo lo que se puede representar en URDF.

A continuacion mostraremos como estan conformados los archivos URDF que iremos creando hasta llegar al modelo de robot terminado, asi como su visualizacion en RViz (los siguientes archivos URDF se crearan en la carpeta “urdf”):

Nos dirigimos primeramente al siguiente directorio:

```
roscd urdf_tutorial/urdf/
```

Como introduccion antes de continuar, es importante conocer los marcos de coordenadas en RViz:

RViz utiliza el sistema de transformacion “tf” para transformar los datos del marco de coordenadas al que llegan en un marco de referencia global.

Hay dos marcos de coordenadas que es importante conocer en el visualizador:

- *Fixed Frame*

Es el mas importante de los dos marcos. Este marco de referencia es utilizado para denotar el marco del “mundo”.

Para obtener resultados correctos, el ‘Fixed Frame’ no debe moverse en relacion con el mundo.

- *Target Frame*

Este marco es el marco de referencia para la vista de la camara.

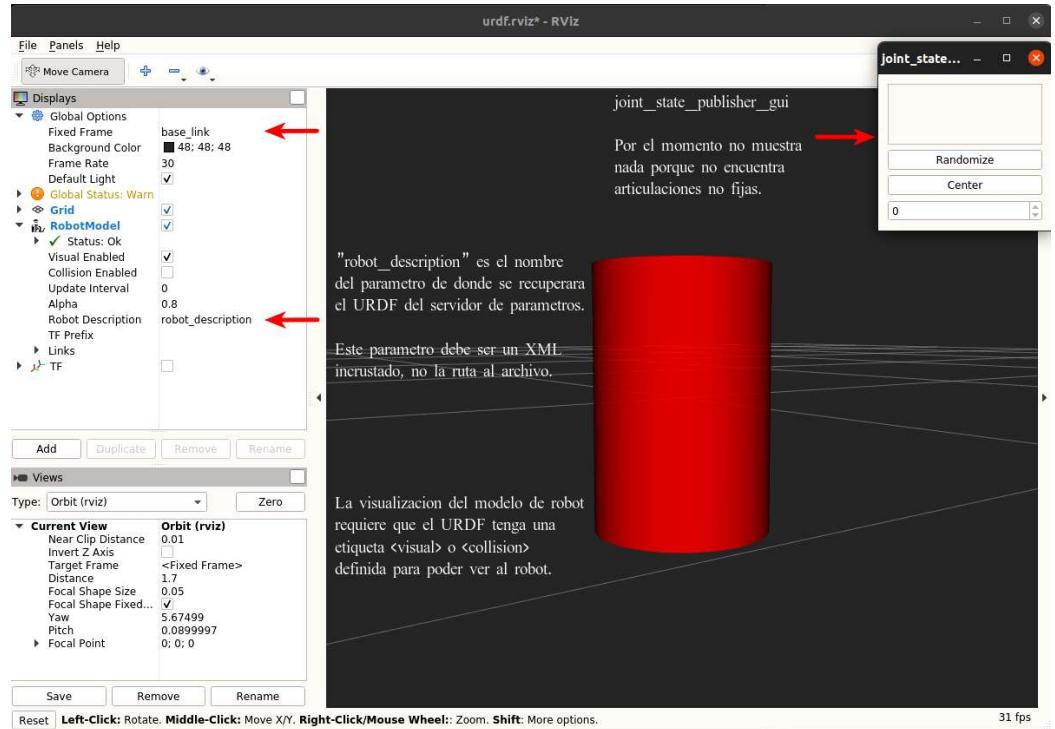
- *01-myfirst.urdf*

Lo que se indica en el archivo es lo siguiente:

- Se declara una etiqueta <robot> con el nombre “myfirst”.
- Se declara un enlace (una parte de robot) con el nombre “base_link” y se le agrega un componente visual con una geometria de tipo cilindro, con altura de 0.6 mts y radio de 0.2 mts.

Visualizacion en RViz:

```
roslaunch urdf_tutorial display.launch  
model:=01-myfirst.urdf
```



Por defecto el origen del elemento se especifica en el origen del sistema de coordenadas.

- *02-multipleshapes.urdf*

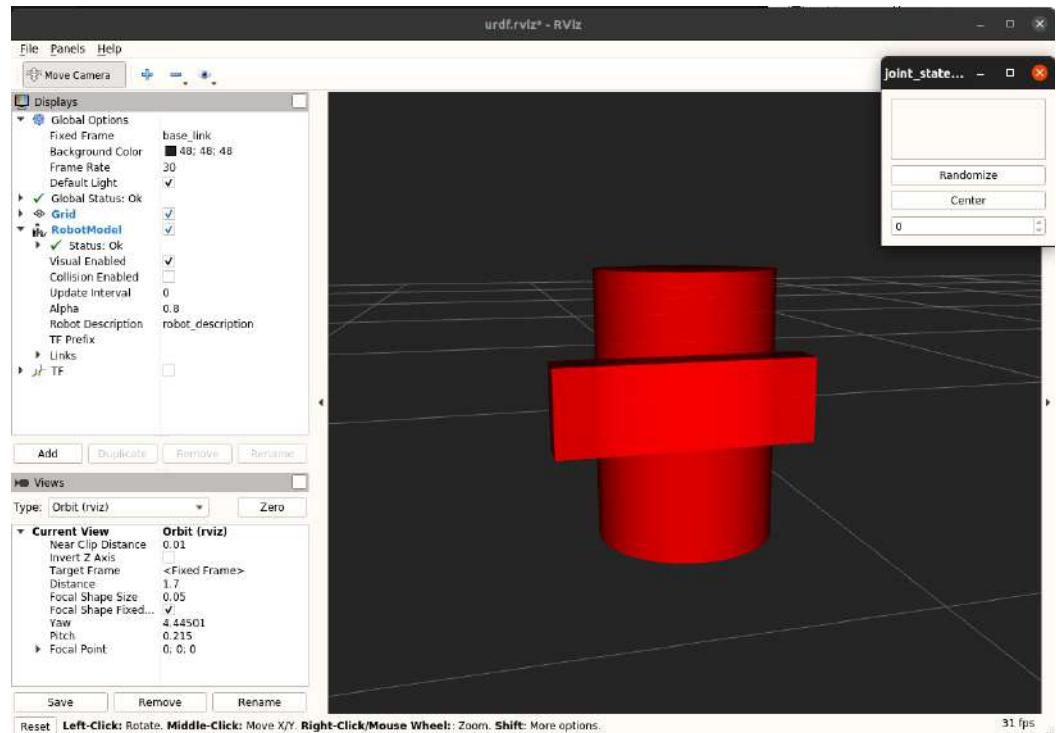
Lo que se indica en el archivo es lo siguiente:

- Se declara una etiqueta `<robot>` con el nombre “multipleshapes”.
- Se declara un enlace (una parte de robot) con el nombre “base_link” y se le agrega un componente visual con una geometria de tipo cilindro, con altura de 0.6 mts y radio de 0.2 mts.
- Se declara un enlace (una parte de robot) con el nombre “right_leg” y se le agrega un componente visual con una geometria de tipo caja, con longitudes de 0.6 mts x 0.1 mts x 0.2 mts (largo x ancho x alto).
- Se declara una articulacion con el nombre de “base_to_right_leg”, con tipo de articulacion fija y como padre se especifica el enlace “base_link” y como hijo el enlace “right_leg”.

Todas las geometrias tendrán su origen en el centro de su geometria por defecto.

Visualizacion en RViz:

```
roslaunch urdf_tutorial display.launch
model:=02-multipleshapes.urdf
```



Como podemos observar, ambas formas se superponen entre si, esto debido a que comparten el mismo origen. Para que no se superpongan debemos definir sus orígenes (la posición en donde queremos que estén).

● 03-origins.urdf

Lo que se indica en el archivo es lo siguiente:

- Se declara una etiqueta <robot> con el nombre “origins”.
- Se declara un enlace (una parte de robot) con el nombre “base_link” y se le agrega un componente visual con una geometría de tipo cilindro, con altura de 0.6 mts y radio de 0.2 mts.
- Se declara un enlace (una parte de robot) con el nombre “right_leg”, se le agrega un componente visual con una geometría de tipo caja, con longitudes de 0.6 mts x 0.1 mts x

rpy (roll pitch yaw)

(Imaginemos una avioneta):

- **roll**: rotación alrededor del eje longitudinal.
- **pitch**: rotación alrededor del eje lateral.
- **yaw**: rotación alrededor del eje vertical.

0.2 mts (largo x ancho x alto) y se le agrega un componente origen con las siguientes especificaciones:

<origin>

- rpy: en radianes
- xyz: en metros

- $(roll, pitch, yaw) = (0, 1.57075, 0)$

En grados seria $(0, 90, 00)$.

- $(x,y,z) = (0, 0, -0.3)$

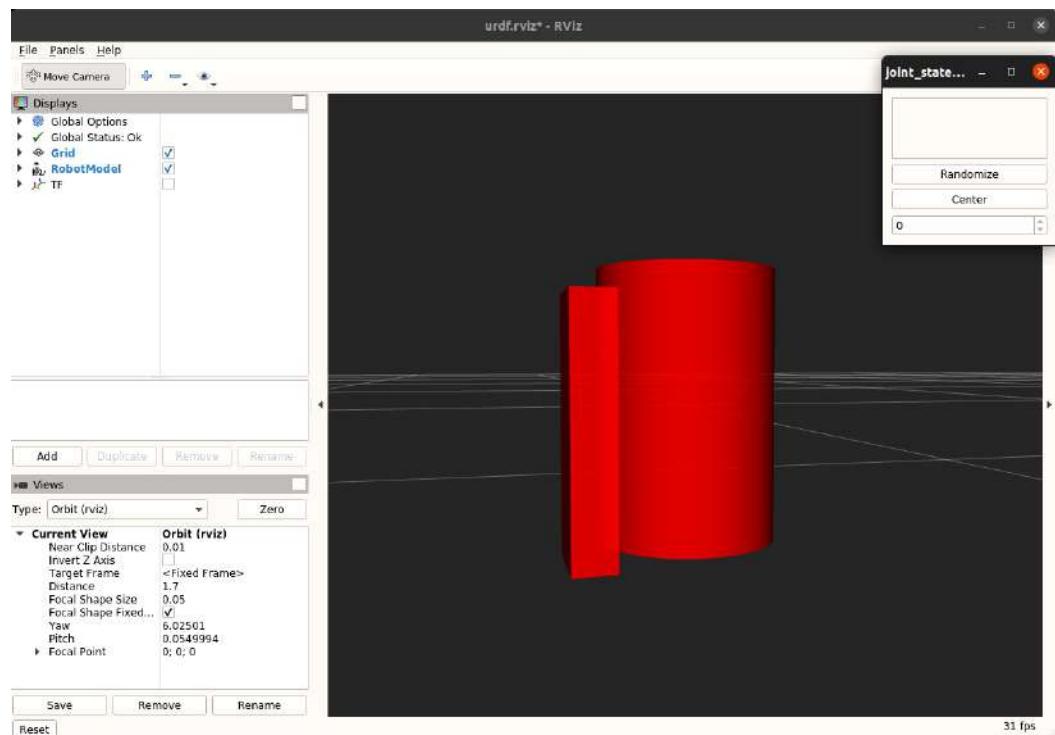
- Se declara una articulacion con el nombre de "base_to_right_leg", con tipo de articulacion fija y como padre se especifica el enlace "base_link" y como hijo el enlace "right_leg". Ademas se le agrega un componente origen con las siguientes especificaciones:

- $(x,y,z) = (0, -0.22, 0.25)$

NOTA: Las posiciones y orientaciones siempre seran relativas a un marco de referencia (puede ser definida por una articulacion).

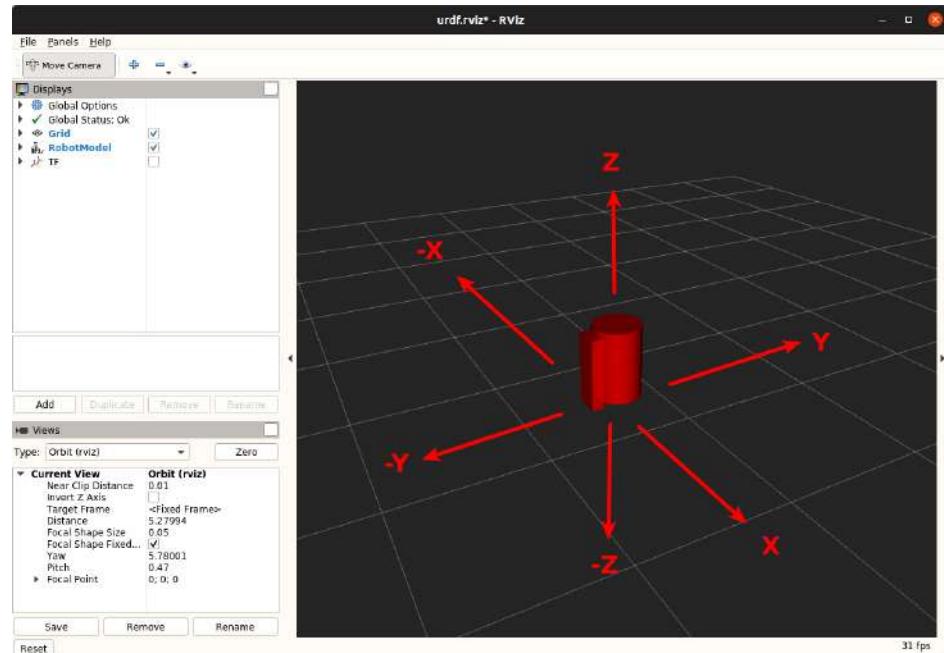
Visualizacion en RViz:

```
roslaunch urdf_tutorial display.launch
model:=03-origins.urdf
```

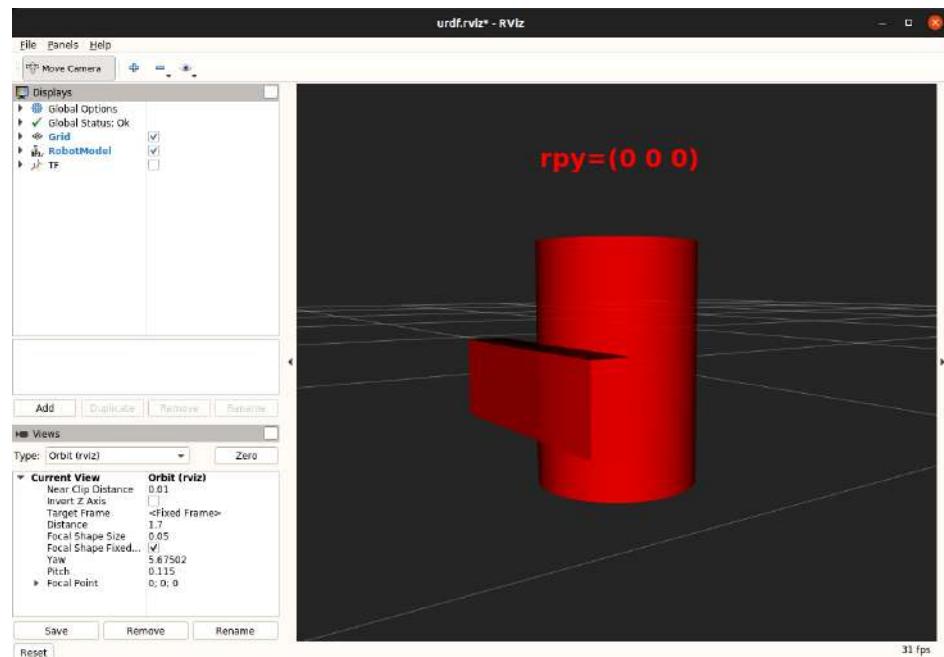


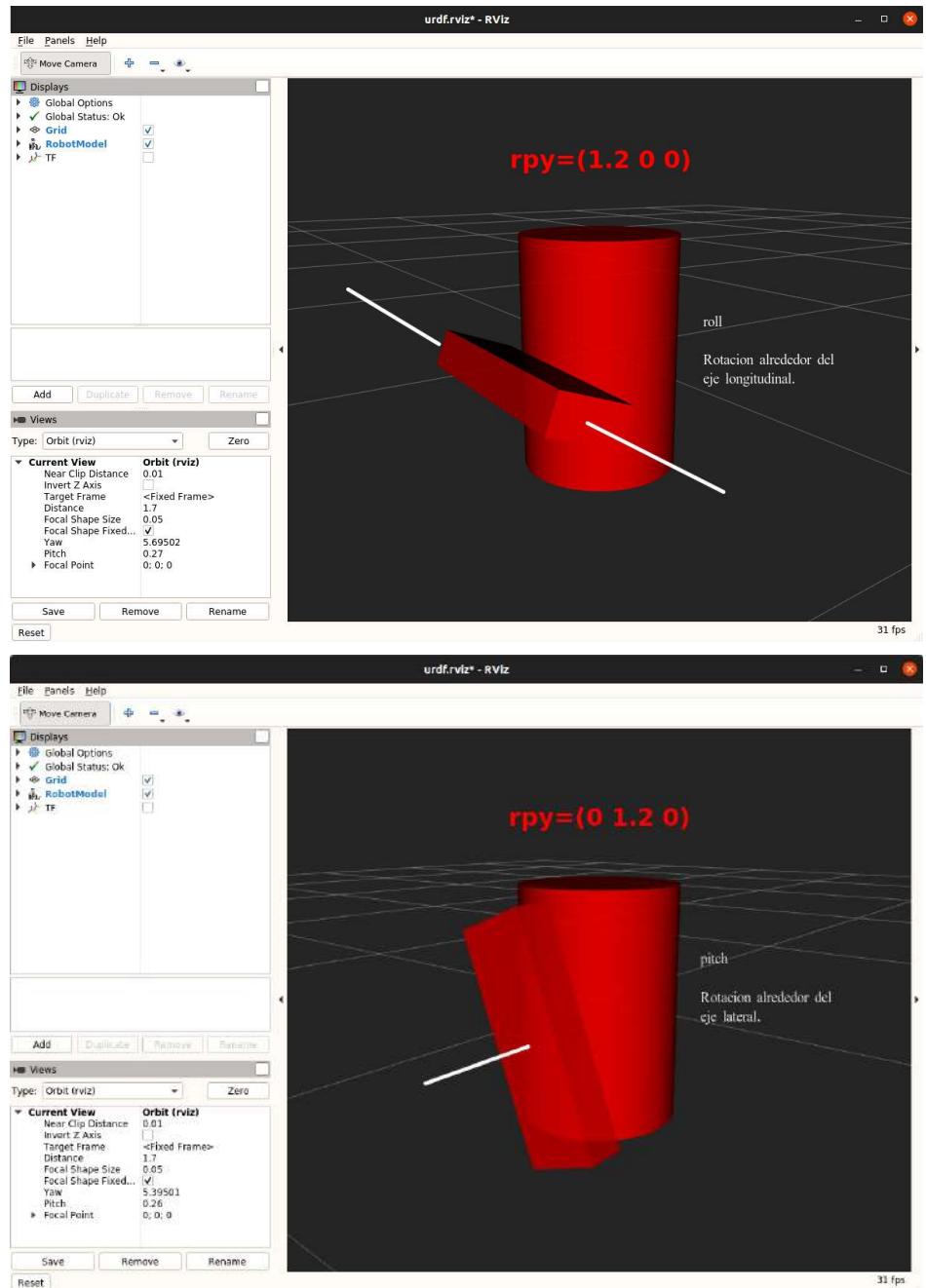
Para entender mejor el posicionamiento y rotacion de un componente en RViz veamos las siguientes imagenes:

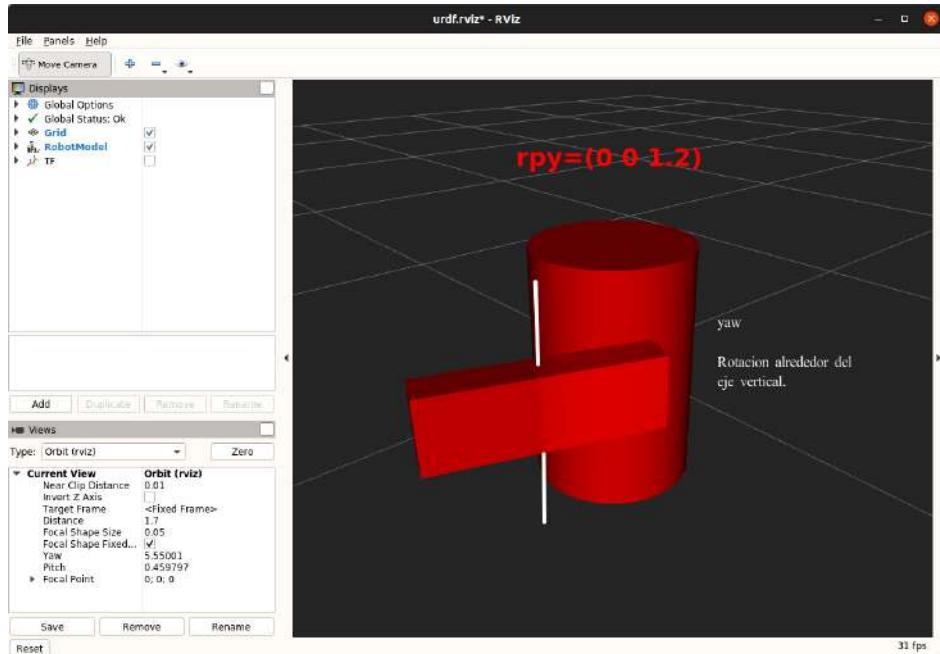
Posicionamiento **xyz**:



Rotacion **rpy**:







- *04-materials.urdf*

Tomando como referencia el archivo “03-origins.urdf”, lo que se agrego fue lo siguiente:

- La etiqueta `<robot>` ahora tiene el nombre “materials”.
- Se declaran dos materiales:
 - Un material con el nombre “blue” en el cual se especifica el color azul por medio del componente `<color>` y su atributo `rgba`.
El enlace “base_link” hara uso de este material.
 - Un material con el nombre “white” en el cual se especifica el color blanco por medio del componente `<color>` y su atributo `rgba`.
El enlace “right_leg” y el enlace “left_leg” haran uso de este material.
- Se declara un enlace (una parte de robot) con el nombre “left_leg”, se le agrega un componente visual con una geometria de tipo caja, con longitudes de 0.6 mts x 0.1 mts x 0.2 mts (largo x ancho x alto) y se le agrega un componente origen con las siguientes especificaciones:
 - $(roll, pitch, yaw)=(0, 1.57075, 0)$
En grados seria $(0, 90, 00)$.
 - $(x,y,z)=(0, 0, -0.3)$

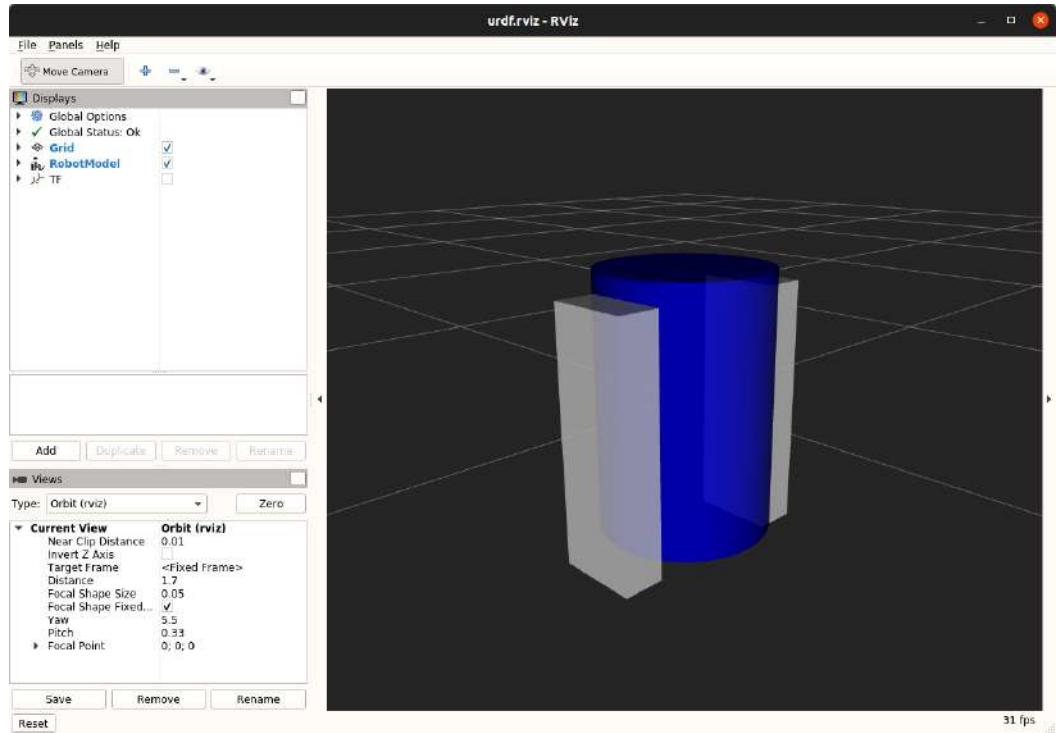
- Se declara una articulacion con el nombre de “base_to_left_leg”, con tipo de articulacion fija y como padre se especifica el enlace “base_link” y como hijo el enlace “left_leg”. Ademas se le agrega un componente origen con las siguientes especificaciones:
 - $(x,y,z)=(0, 0.22, 0.25)$

En realidad este ultimo enlace y esta ultima articulacion conforman la pierna izquierda del robot (ya que ya habiamos declarado la pierna derecha). Y como podemos observar, es muy similar a lo que se especifico para la pierna derecha solo que ahora la articulacion para la pierna izquierda esta situada en otra posicion en el eje Y (contraria a la articulacion de la pierna derecha).

Con la etiquete <material> podemos definir colores y texturas (en la textura se debe especificar un archivo de imagen el cual se utiliza para colorear el objeto) para los enlaces.

Visualizacion en RViz:

```
roslaunch urdf_tutorial display.launch  
model:=04-materials.urdf
```



- **05-visual.urdf**

Aqui estaremos trabajando con unas mallas que se utilizaran como geometrias. Estas mallas se encuentran en:

https://github.com/ros/urdf_tutorial/tree/ros1/meshes

Crearemos una carpeta llamada “meshes” en el paquete “urdf_tutorial” donde incluiremos dichos archivos.

Estas mallas son para la pinza que tendra el robot, ya que es necesario usar mallas debido a que no podemos utilizar formas como cubos, cilindros, esferas, para dar forma de pinza.

Continuando, tomando como referencia el archivo “04-materials.urdf”, lo que se agrego fue lo siguiente:

- La etiqueta <robot> ahora tiene el nombre “visual”.
- Se crearon nuevos enlaces.
- Se crearon nuevas articulaciones que unen ciertas partes (para que el robot tenga movilidad) especificando el tipo de articulacion (muy importante).
- Se colocaron los enlaces de tal forma que se pareciera al robot R2-D2 (personaje de Star Wars), lo cual se logro modificando los atributos *xyz* y *rpy* (para la ubicacion y la rotacion de las

partes).

- Con respecto a la union entre articulacion y enlace, tenemos lo siguiente:

(Parte derecha del robot)

- *base_to_right_leg*
 - Padre: *base_link*
 - Hijo: *right_leg*
 - Tipo de articulacion: *fixed*

Une fijamente el enlace base con el enlace de la pierna derecha.
- *right_base_joint*
 - Padre: *right_leg*
 - Hijo: *right_base*
 - Tipo de articulacion: *fixed*

Une fijamente el enlace de la pierna derecha con el enlace donde iran las ruedas del lado derecho.
- *right_front_wheel_joint*
 - Padre: *right_base*
 - Hijo: *right_front_wheel*
 - Tipo de articulacion: *fixed*

Une fijamente el enlace donde iran las ruedas con la rueda derecha delantera.
- *right_back_wheel_joint*
 - Padre: *right_base*
 - Hijo: *right_back_wheel*
 - Tipo de articulacion: *fixed*

Une fijamente el enlace donde iran las ruedas con la rueda derecha trasera.

(Parte izquierda del robot)

- *base_to_left_leg*
 - Padre: *base_link*
 - Hijo: *left_leg*
 - Tipo de articulacion: *fixed*

Une fijamente el enlace base con el enlace de la pierna izquierda.
- *left_base_joint*

- Padre: *left_leg*
 - Hijo: *left_base*
 - Tipo de articulacion: *fixed*

Une fijamente el enlace de la pierna izquierda con el enlace donde iran las ruedas del lado izquierdo.
 - *left_front_wheel_joint*
 - Padre: *left_base*
 - Hijo: *left_front_wheel*
 - Tipo de articulacion: *fixed*

Une fijamente el enlace donde iran las ruedas con la rueda izquierda delantera.
 - *left_back_wheel_joint*
 - Padre: *left_base*
 - Hijo: *left_back_wheel*
 - Tipo de articulacion: *fixed*

Une fijamente el enlace donde iran las ruedas con la rueda izquierda trasera.
- (Pinza del robot)
- *gripper_extension*
 - Padre: *base_link*
 - Hijo: *gripper_pole*
 - Tipo de articulacion: *fixed*

Une fijamente el enlace base con el enlace donde ira la pinza.
 - *left_gripper_joint*
 - Padre: *gripper_pole*
 - Hijo: *left_gripper*
 - Tipo de articulacion: *fixed*

Une fijamente el enlace donde ira la pinza con el enlace de la pinza izquierda.
 - *left_tip_joint*
 - Padre: *left_gripper*
 - Hijo: *left_tip*
 - Tipo de articulacion: *fixed*

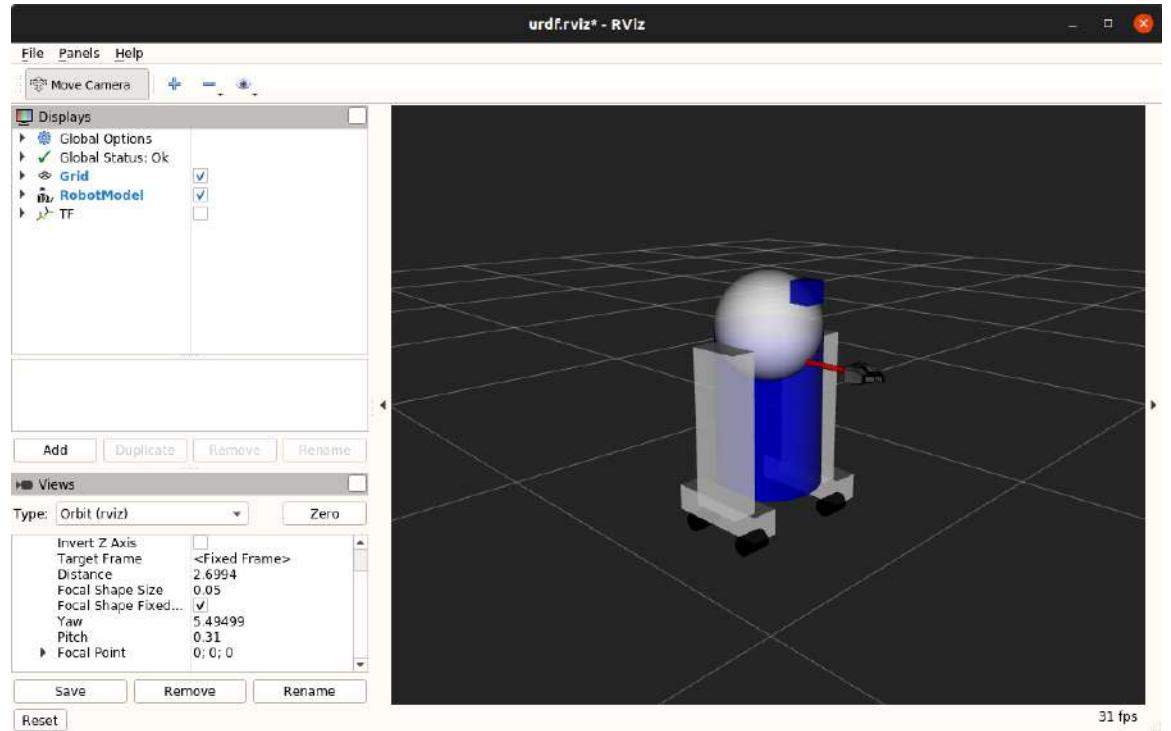
Une fijamente el enlace de la pinza izquierda con el enlace de la punta de la pinza izquierda.

- *right_gripper_joint*
 - Padre: *gripper_pole*
 - Hijo: *right_gripper*
 - Tipo de articulacion: *fixed*
Une fijamente el enlace donde ira la pinza con el enlace de la pinza derecha.
 - *right_tip_joint*
 - Padre: *right_gripper*
 - Hijo: *right_tip*
 - Tipo de articulacion: *fixed*
Une fijamente el enlace de la pinza derecha con el enlace de la punta de la pinza derecha.
- (Cabeza del robot)
- *head_swivel*
 - Padre: *base_link*
 - Hijo: *head*
 - Tipo de articulacion: *fixed*
Une fijamente el enlace base con el enlace de la cabeza.
 - *tobox*
 - Padre: *head*
 - Hijo: *box*
 - Tipo de articulacion: *fixed*
Une fijamente el enlace de la cabeza con el enlace de un cubo (el cual simula el ojo del robot y mas aparte nos ayuda a identificar la rotacion de la cabeza)

En total tenemos 15 articulaciones definidas que unen dos enlaces mediante un tipo de articulacion (cambiando el tipo de articulacion es como definiremos el movimiento relativo del enlace hijo con respecto al enlace padre).

Visualizacion en RViz:

```
roslaunch urdf_tutorial display.launch
model:=05-visual.urdf
```



Mientras esta lanzado el archivo de lanzamiento veamos lo siguiente:

```
^Croot@chuy:/catkin_ws/src/urdf_tutorial/urdf# rostopic info /tf_static
Type: tf2_msgs/TFMessage

Publishers:
* /robot_state_publisher (http://chuy:36299/)

Subscribers:
* /rviz (http://chuy:39979/)

root@chuy:/catkin_ws/src/urdf_tutorial/urdf# rosmsg show tf2_msgs/TFMessage
geometry_msgs/TransformStamped[] transforms
  std_msgs/Header header
    uint32 seq
    time stamp
    string frame_id
    string child_frame_id
  geometry_msgs/Transform transform
    geometry_msgs/Vector3 translation
      float64 x
      float64 y
      float64 z
    geometry_msgs/Quaternion rotation
      float64 x
      float64 y
      float64 z
      float64 w

root@chuy:/catkin_ws/src/urdf_tutorial/urdf#
```

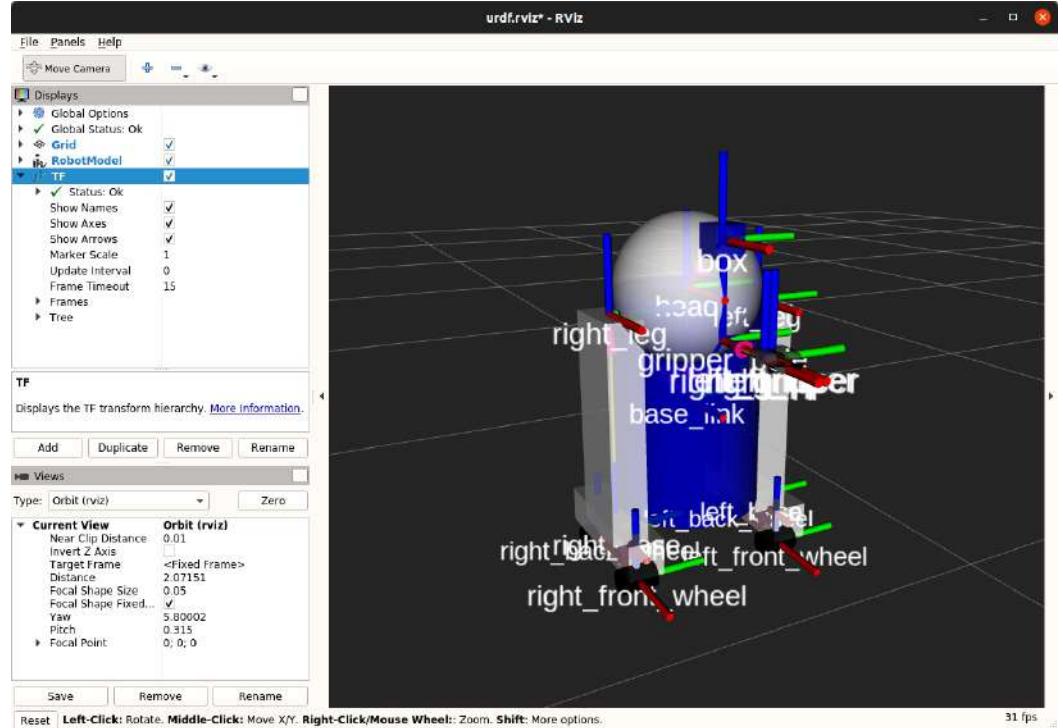
```
root@chuy:/catkin_ws/src/urdf_tutorial/urdf# rostopic echo /tf_static
transforms:
  -
    header:
      seq: 0
      stamp:
        secs: 1652723947
        nsecs: 586731663
      frame_id: "base_link"
      child_frame_id: "left_leg"
      transform:
        translation:
          x: 0.0
          y: 0.22
          z: 0.25
        rotation:
          x: 0.0
          y: 0.0
          z: 0.0
          w: 1.0
    header:
      seq: 0
      stamp:
        secs: 1652723947
        nsecs: 586741414
      frame_id: "base_link"
      child_frame_id: "right_leg"
      transform:
        translation:
          x: 0.0
          y: -0.22
          z: 0.25
        rotation:
          x: 0.0
          y: 0.0
          z: 0.0
          w: 1.0
    header:
      seq: 0
      stamp:
        secs: 1652723947
        nsecs: 586742238
      frame_id: "base_link"
      child_frame_id: "gripper_pole"
      transform:
        translation:
          x: 0.19
          y: 0.0
          z: 0.2
        rotation:
          x: 0.0
          y: 0.0
          z: 0.0
          w: 1.0
  -
    header:
```

Como podemos observar, en el tema “/tf_static” al momento de suscribirnos obtenemos informacion con respecto a las articulaciones fijas, es decir, informacion relacionada con la traslacion y orientacion del enlace hijo con respecto al enlace padre (realmente los valores son los mismos, no cambian).

El tipo de mensaje de respuesta es una lista que almacena los marcos de coordenadas de un enlace hijo con respecto a un enlace padre (son 15 en total).

En los temas “/tf” y “/joint_states” no se publica nada (aun no).

En RViz si seleccionamos la casilla “TF” veremos lo siguiente:



Lo que hemos hecho es mostrar el arbol de transformacion.

Tenemos tres datos opcionales para mostrar:

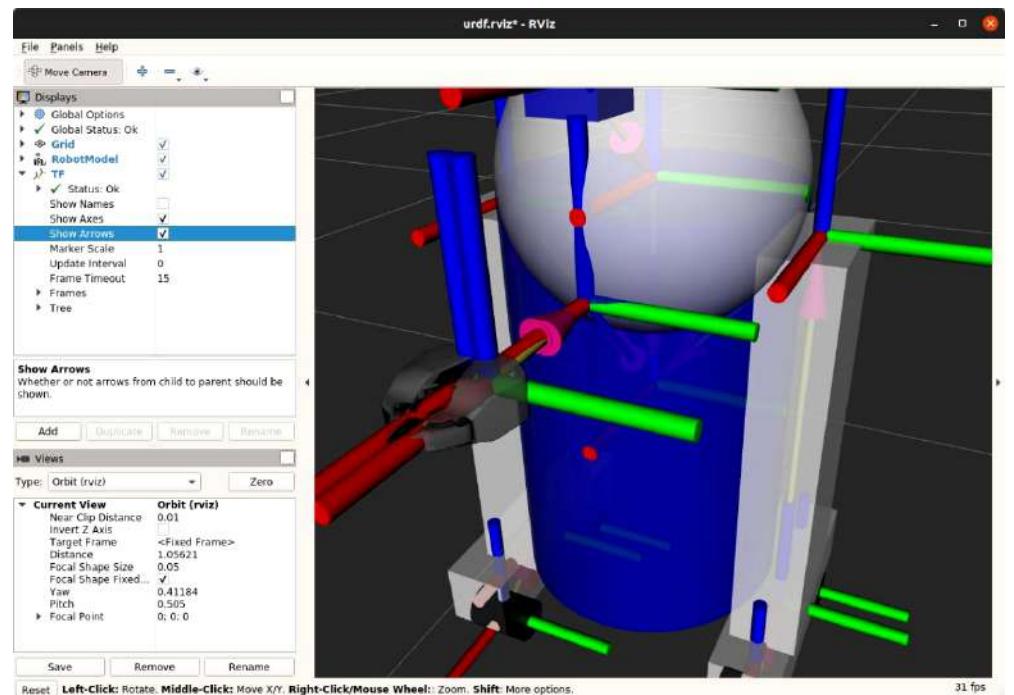
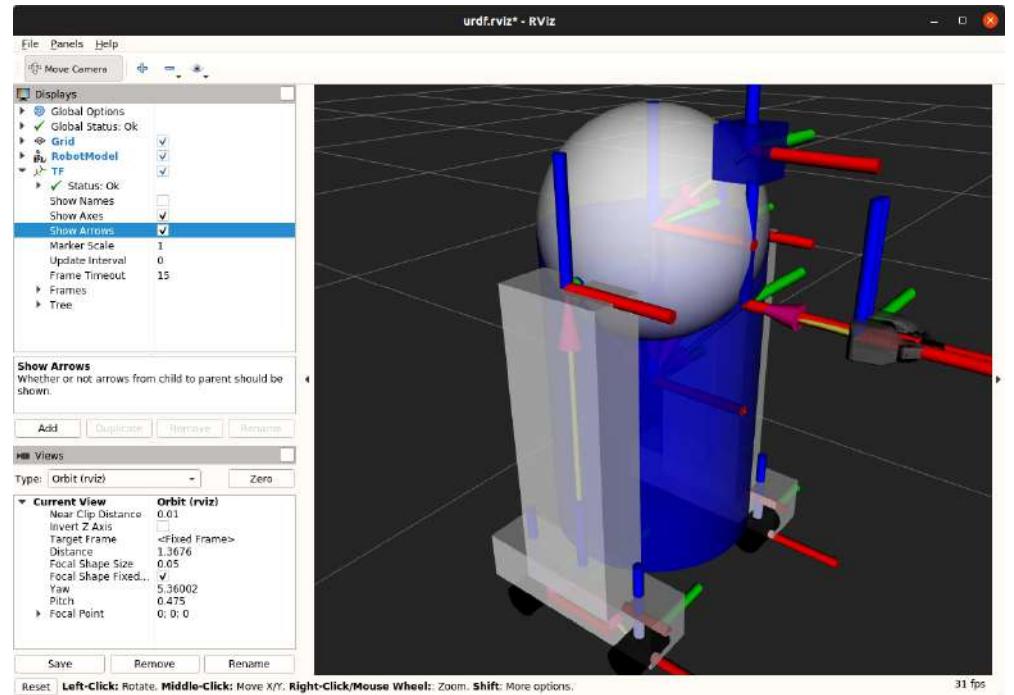
- El nombre del marco
- Los ejes del marco
- Una flecha desde el marco hasta su padre

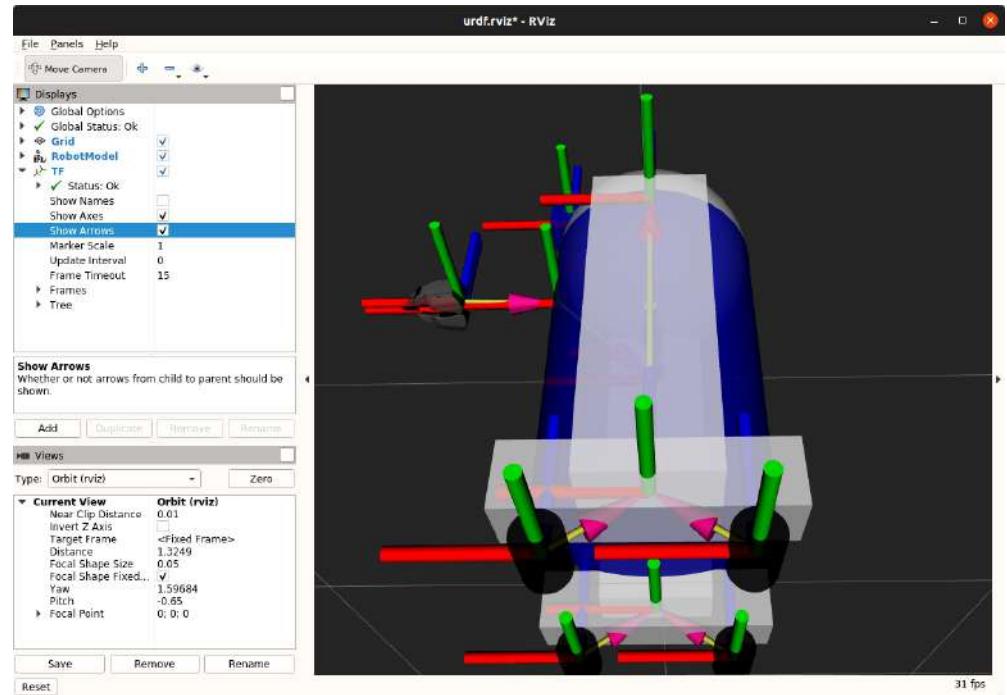


Veamos mas a fondo estas opciones:

Los ejes se muestran de la siguiente manera:

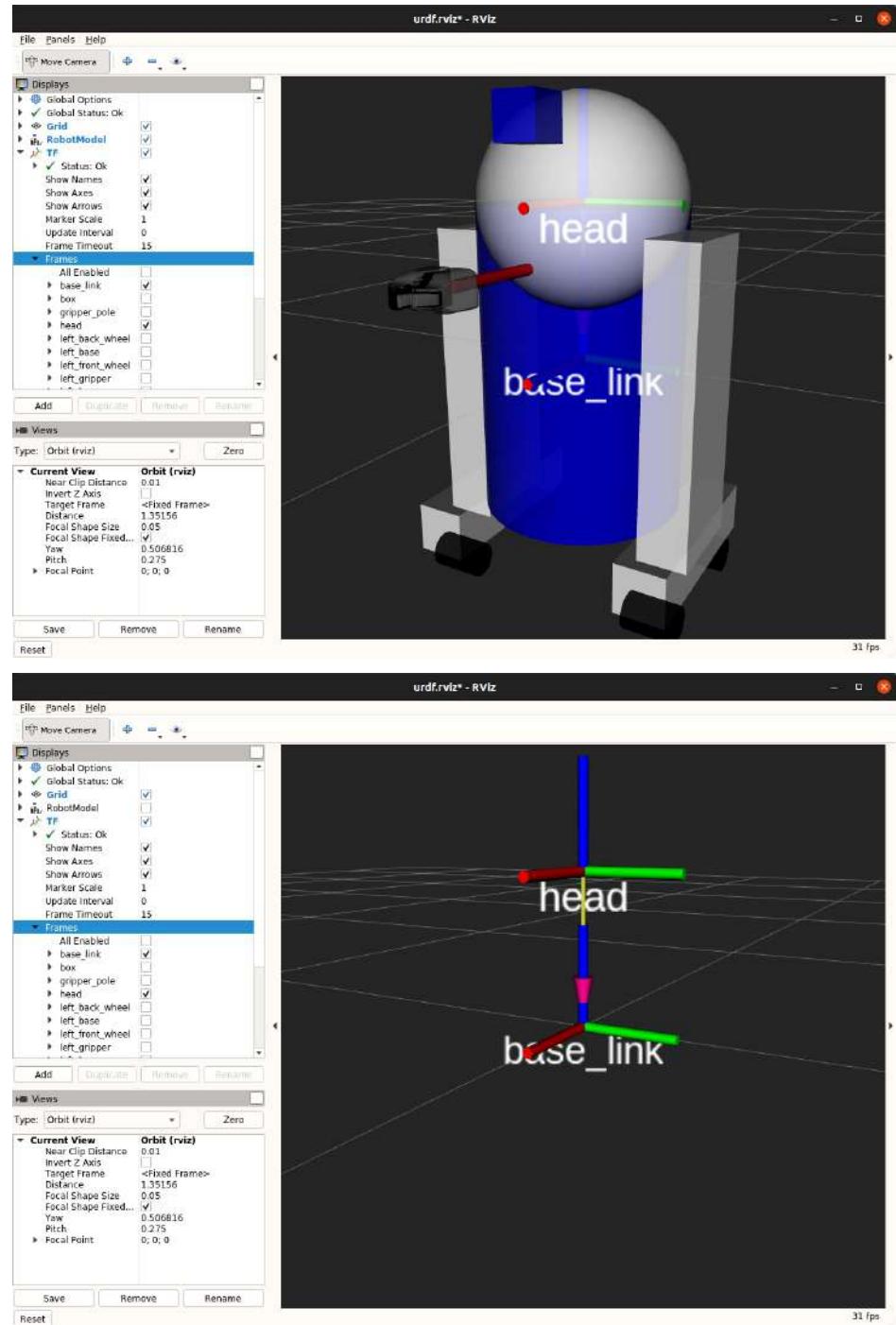
- El eje **X** se indica en **rojo**
- El eje **Y** se indica en **verde**
- El eje **Z** se indica en **azul**





Quitando un poco los nombres de los marcos, podemos observar de manera mas clara donde se encuentran los orígenes de los marcos y como estan relacionados con su marco padre (mediante una flecha).

Tomemos las siguiente imagenes para analizar lo siguiente:



En el URDF se tiene lo siguiente:

```
<link name="head">
  <visual>
    <geometry>
      <sphere radius="0.2"/>
    </geometry>
    <material name="white"/>
  </visual>
</link>
<joint name="head_swivel" type="fixed">
  <parent link="base_link"/>
  <child link="head"/>
  <origin xyz="0 0 0.3"/>
</joint>
```

Podemos observar que la cabeza esta unida al enlace base por medio de una articulacion fija, cuya articulacion tiene su origen a 0.3 mts en el eje Z con respecto al marco del enlace padre, que en este caso es el enlace base (el cual esta en el centro de su geometria, en el centro del cilindro de color azul).

Recordemos que al definir articulaciones, el origen de la articulacion es con respecto al origen del enlace padre, y al especificar el origen de la articulacion, el enlace hijo tendra el mismo origen (esto en el caso de que no definamos un origen al enlace hijo).

Por ultimo, es importante comprender como es que un modelo de robot se inicia desde un enlace raiz y desde este enlace salen otros enlaces unidos por articulaciones, y al final de todo terminamos con una estructura de arbol como el siguiente:

```

▼ Tree
  ▼ base_link
    ▼ gripper_pole
      ▼ left_gripper
        left_tip
      ▼ right_gripper
        right_tip
    ▼ head
      box
    ▼ left_leg
      ▼ left_base
        left_front_wheel
        left_back_wheel
    ▼ right_leg
      ▼ right_base
        right_front_wheel
        right_back_wheel

```

- *06-flexible.urdf*

Tomando como referencia el archivo “05-visual.urdf”, lo que se agrego fue lo siguiente:

- La etiqueta <robot> ahora tiene el nombre “flexible”.
- Ahora algunas articulaciones seran moviles (cambiaremos el tipo de articulacion en algunas articulaciones). En concreto, las articulaciones moviles seran:

Articulaciones de las ruedas del robot

- *right_front_wheel_joint (continuous)*
- *right_back_wheel_joint (continuous)*
- *left_front_wheel_joint (continuous)*
- *left_back_wheel_joint (continuous)*

Articulacion del brazo de la pinza del robot

- *gripper_extension (prismatic)*

Articulaciones de las pinzas del robot

- *left_gripper_joint (revolute)*
- *right_gripper_joint (revolute)*

Articulacion de la cabeza del robot

- *head_swivel (continuous)*

Y las no moviles (fixed) seran:

- *base_to_right_leg*
- *right_base_joint*

- *base_to_left_leg*
- *left_base_joint*
- *left_tip_joint*
- *right_tip_joint*
- *tobox*

<axis>

- Eje de rotacion para las articulaciones giratorias (continuous)
- Eje de translacion para las articulaciones prismáticas (prismatic).
- Superficie normal (perpendicular) al eje para las articulaciones planas (plane).

Informacion sobre el tipo de articulacion:

- *continuous*

Permite la rotacion del hijo con respecto al padre alrededor de un eje y no tiene limites superior e inferior (sin limites).

<limit>

Solo es requerido para las articulaciones “revolute” y “prismatic”.

En este tipo de articulacion definimos lo siguiente en el URDF:

```
<joint name="..." type="continuous">
  <axis rpy="0 0 0" xyz="..."/>
  ...
</joint>
```

Lo que modificamos es el atributo **xyz** (el vector debe estar normalizado).

En el caso de las articulaciones de las ruedas del robot, se define como **xyz="0 1 0"**, es decir, las ruedas giraran alrededor del eje Y. (de esta manera podran hacer que el robot avance).

Y en el caso de la articulacion de la cabeza del robot, se define como **xyz="0 0 1"**, es decir, la cabeza girara alrededor del eje Z.

- *prismatic*

Permite la translacion del hijo con respecto al padre en una dimension.

En este tipo de articulacion definimos lo siguiente en el

URDF:

```
<joint name="..." type="prismatic">
  <axis rpy="0 0 0" xyz="..." />
  <limit effort="..." lower="..." upper="..."
    velocity="..."/>
  ...
</joint>
```

Lo que modificamos es:

■ **<axis>**

- El atributo **xyz** (el vector debe estar normalizado).

En el caso de la articulacion del brazo de la pinza del robot, se define como **xyz="1 0 0"**, es decir, el brazo podra trasladarse por el eje X (con ciertos limites).

■ **<limit>**

- El atributo **effort** el cual se especifica para hacer cumplir el maximo esfuerzo de la articulacion (en Nm).
- El atributo **lower** el cual especifica el limite inferior de la articulacion (en mts).
- El atributo **upper** el cual especifica el limite superior de la articulacion (en mts).
- El atributo **velocity** el cual se especifica para hacer cumplir la velocidad maxima de la articulacion (en m/s).

● *revolute*

Permite la rotacion del hijo con respecto al padre alrededor de un eje y tiene un rango limitado especificado por los limites superior e inferior.

En este tipo de articulacion definimos lo siguiente en el URDF:

```
<joint name="..." type="revolute">
```

```

<axis rpy="0 0 0" xyz="..."/>
<limit effort="..." lower="..." upper="..."
  velocity="..."/>
...
</joint>

```

Lo que modificamos es:

- **<axis>**
 - El atributo **xyz** (el vector debe estar normalizado).

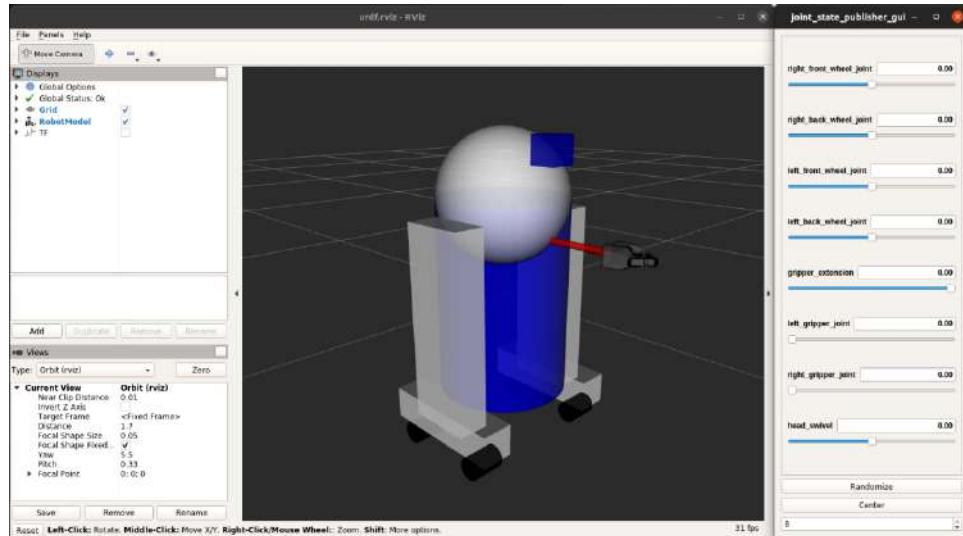
En el caso de las articulaciones de las pinzas del robot, se define como **xyz="0 0 1"**, es decir, las pinzas giraran alrededor del eje Z (con ciertos limites).
- **<limit>**
 - El atributo **effort** el cual se especifica para hacer cumplir el maximo esfuerzo de la articulacion (en Nm).
 - El atributo **lower** el cual especifica el limite inferior de la articulacion (en rad).
 - El atributo **upper** el cual especifica el limite superior de la articulacion (en rad).
 - El atributo **velocity** el cual se especifica para hacer cumplir la velocidad maxima de la articulacion (en rad/s).

Visualizacion en RViz:

```

roslaunch urdf_tutorial display.launch
model:=06-flexible.urdf

```



Cuando definimos articulaciones móviles , al cargar el modelo en RViz, se nos mostrará una GUI (“joint_state_publisher”) que nos permitirá controlar los valores de las articulaciones no fijas. A medida que movemos los controles deslizantes en la GUI, el modelo se mueve en RViz. Esto se logra gracias a que la GUI analiza el URDF y encuentra todas las articulaciones no fijas y sus límites. Luego, usa los valores de los controles deslizantes para publicar mensajes de tipo “sensor_msgs/JointState”. Luego, “robot_state_publisher” los usa para calcular todas las transformaciones entre las diferentes partes del robot.

Veamos que está sucediendo mientras se visualiza nuestro robot en RViz:

Recordemos que el nodo “joint_state_publisher” publica en el tema “/joint_states” los estados de las articulaciones (el cual tiene una GUI que nos ayuda a modificar valores de las articulaciones no fijas), y a su vez el nodo “robot_state_publisher” se suscribe al tema “/joint_states” y también publica en el tema “/tf” en donde está suscrito el nodo “rviz” para visualizar cada movimiento del robot.

El nodo “joint_state_publisher” lee el parámetro “robot_description” del servidor de parámetros, encuentra todas las articulaciones no fijas y publica sus valores en el tema

“/joint_states”, y el nodo “robot_state_publisher” se suscribe a dicho tema y publica transformaciones para todos los estados de las articulaciones.

Veamos que se esta publicando en el tema “/joint_states”:

```
root@chuy:/catkin_ws/src/urdf_tutorial/urdf# rostopic info /joint_states
Type: sensor_msgs/JointState

Publishers:
* /joint_state_publisher (http://chuy:36525/)

Subscribers:
* /robot_state_publisher (http://chuy:42345/)

root@chuy:/catkin_ws/src/urdf_tutorial/urdf# rosmsg show sensor_msgs/JointState
std_msgs/Header header
  uint32 seq
  time stamp
  string frame_id
string[] name
float64[] position
float64[] velocity
float64[] effort

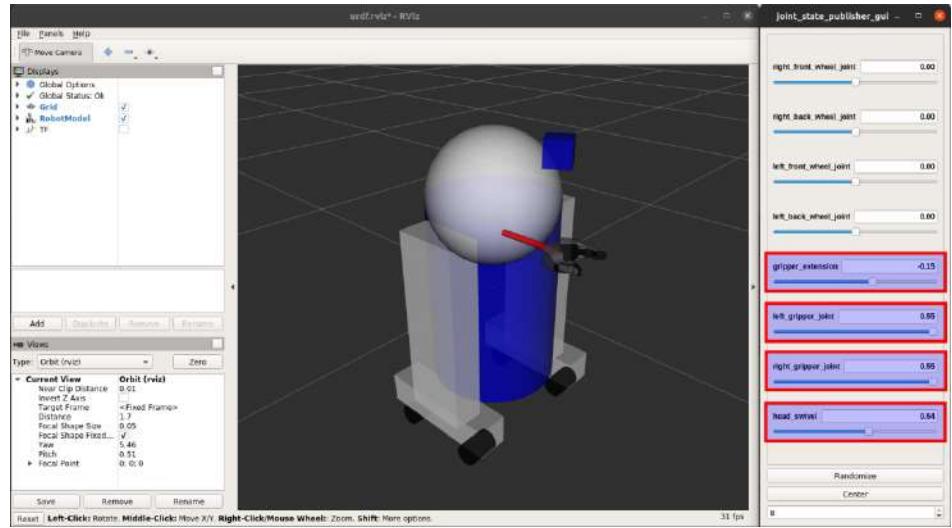
root@chuy:/catkin_ws/src/urdf_tutorial/urdf#
```

El tipo de mensaje “sensor_msgs/JointState” contiene datos para describir el estado de un conjunto de articulaciones controladas por torsion.

El estado de cada articulacion (revolute o prismatic) esta definido por:

- La posicion de la articulacion (rad o m)
- La velocidad de la articulacion (rad/s o m/s)
- El esfuerzo que se aplica en la junta (Nm o N)

Cada articulacion se identifica de forma unica por su nombre. El encabezado especifica el momento en que se registraron los estados de las articulaciones (todos los estados de las articulaciones en un mensaje deben registrarse al mismo tiempo).



Los valores que se modificaron estan señalados en rojo (a modo de ejemplo).

Al suscribirnos al tema “/joint_states” obtenemos lo siguiente (solo escuchamos una vez, ya que se publica constantemente en el tema):

```
root@chuy:/catkin_ws/src/urdf_tutorial/urdf# rostopic echo /joint_states -n 1
header:
  seq: 14793
  stamp:
    secs: 1652970289
    nsecs: 243932962
    frame_id: ''
name: [right_front_wheel_joint, right_back_wheel_joint, left_front_wheel_joint, left_back_wheel_joint,
       gripper_extension, left_gripper_joint, right_gripper_joint, head_swivel]
position: [0.0, 0.0, 0.0, 0.0, -0.1480479999999999, 0.548, 0.548, 0.5422388926095984]
velocity: []
effort: []
...
root@chuy:/catkin_ws/src/urdf_tutorial/urdf#
```

Como podemos observar, las articulaciones no fijas son las que aparecen en la lista.

Cuando no realizamos ninguna modificacion, es decir, dejamos todas las articulaciones como se encuentran desde un inicio, tenemos lo siguiente:

```
root@chuy:/catkin_ws/src/urdf_tutorial/urdf# rostopic echo /joint_states -n 1
header:
  seq: 27252
  stamp:
    secs: 1652971535
    nsecs: 144049882
    frame_id: ''
name: [right_front_wheel_joint, right_back_wheel_joint, left_front_wheel_joint, left_back_wheel_joint,
       gripper_extension, left_gripper_joint, right_gripper_joint, head_swivel]
position: [0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0]
velocity: []
effort: []
...
root@chuy:/catkin_ws/src/urdf_tutorial/urdf#
```

Veamos que se esta publicando en el tema “/tf”:

Al suscribirnos al tema “/tf” obtenemos lo siguiente (solo escuchamos una vez, ya que se publica constantemente en el tema):

```
root@chuy:/catkin_ws/src/urdf_tutorial/urdf# rostopic echo /tf -n 1
transforms:
- header:
    seq: 0
    stamp:
      secs: 1652972102
      nsecs: 43735980
    frame_id: "base_link"
    child_frame_id: "gripper_pole"
    transform:
      translation:
        x: 0.19
        y: 0.0
        z: 0.2
      rotation:
        x: 0.0
        y: 0.0
        z: 0.0
        w: 1.0

- header:
    seq: 0
    stamp:
      secs: 1652972102
      nsecs: 43735980
    frame_id: "base_link"
    child_frame_id: "head"
    transform:
      translation:
        x: 0.0
        y: 0.0
        z: 0.3
      rotation:
        x: 0.0
        y: 0.0
        z: 0.0
        w: 1.0

- header:
    seq: 0
    stamp:
      secs: 1652972102
      nsecs: 43735980
    frame_id: "left_base"
    child_frame_id: "left_back_wheel"
    transform:
      translation:
        x: -0.133333333333
        y: 0.0
        z: -0.085
      rotation:
        x: 0.0
        y: 0.0
        z: 0.0
        w: 1.0

- header:
```

En total son 8 resultados, para cada articulacion no fija que tenemos en nuestro modelo de robot.

Como podemos observar, podemos realizar un seguimiento de multiples marcos de coordenadas a lo largo del tiempo (las transformaciones nos dan informacion sobre donde se

encuentra el marco de coordenadas del hijo con respecto al padre).

Con respecto a los temas “/tf” y “/tf_static” (del que ya se hablo anteriormente) tenemos que:

- En el tema “/tf_static” se tienen los marcos que no cambian con el tiempo.
- En el tema “/tf” se tienen los marcos que cambian con el tiempo.

En robotica, las posiciones y orientaciones de objetos a menudo se expresan como transformaciones de un marco de coordenadas en otro.

En general, ***la ubicacion de un marco se especifica en relacion con su padre***, y como ya se vio, cada marco tiene un nombre unico, llamado “frame_id”.

Veamos un ejemplo para entender esto anterior:

Tomaremos como referencia la siguiente transformacion del marco padre “head” (la cabeza del robot) al marco hijo “box” (el “ojo” del robot), los cuales no cambian con el tiempo:

Es uno de tantos que se obtienen al suscribirse al tema “/tf_static”.

```
header:  
  seq: 0  
  stamp:  
    secs: 1652968801  
    nsecs: 199959065  
    frame_id: "head"  
    child_frame_id: "box"  
  transform:  
    translation:  
      x: 0.1814  
      y: 0.0  
      z: 0.1414  
    rotation:  
      x: 0.0  
      y: 0.0  
      z: 0.0  
      w: 1.0
```

En el URDF tenemos lo siguiente:

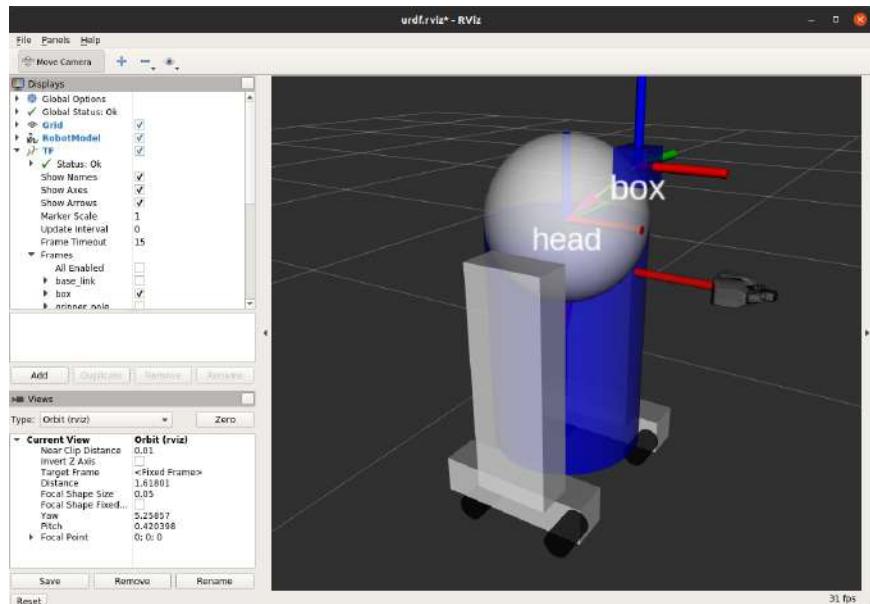
```

<link name="box">
  <visual>
    <geometry>
      <box size="0.08 0.08 0.08"/>
    </geometry>
    <material name="blue"/>
  </visual>
</link>

<joint name="tobox" type="fixed">
  <parent link="head"/>
  <child link="box"/>
  <origin xyz="0.1814 0 0.1414"/>
</joint>

```

En RViz tenemos lo siguiente:



Lo que nos dice la transformacion es donde se encuentra el marco del hijo con respecto al marco del padre. En este caso tenemos lo siguiente:

(la “f” es solo para hacer mención a “father”)

Sean ***if*** (*en direccion del eje X*), ***jf*** (*en direccion del eje Y*) y ***kf*** (*en direccion del eje Z*) los vectores base (vectores unitarios) del marco de coordenadas del padre.

El origen del marco de coordenadas del hijo se encuentra en la posicion:

$$0.1914 \mathbf{if} + 0.0 \mathbf{jf} + 0.1414 \mathbf{kf}$$

Con respecto al marco de coordenadas del padre.

Con respecto al marco del padre, mediante la transformacion

```
transform:  
translation:  
  x: 0.1914  
  y: 0.0  
  z: 0.1414  
rotation:  
  x: 0.0  
  y: 0.0  
  z: 0.0  
  w: 1.0
```

podemos obtener cualquier punto en el marco del hijo (podemos representar cualquier punto en el marco de coordenadas del hijo utilizando un poco de calculo vectorial).

Por ejemplo:

Para este caso solo tenemos una translacion del marco del hijo con respecto al marco del padre.

(la "c" es solo para hacer mención a "child")

Sean ***ic* (en dirección del eje X), *jc* (en dirección del eje Y) y *kc* (en dirección del eje Z)** los vectores base (vectores unitarios) del marco de coordenadas del hijo.

Un punto P(x,y,z) en el marco de coordenadas del hijo puede ser representado de la siguiente manera:

$$\text{Sea } \mathbf{u} = 0.1914 \mathbf{if} + 0.0 \mathbf{jf} + 0.1414 \mathbf{kf}$$

El vector que va del origen del marco de coordenadas del padre al origen del marco de coordenadas del hijo.

$$\text{Sea } \mathbf{v} = x \mathbf{ic} + y \mathbf{jc} + z \mathbf{kc}$$

El vector que va del origen del marco de coordenadas del hijo al punto P.

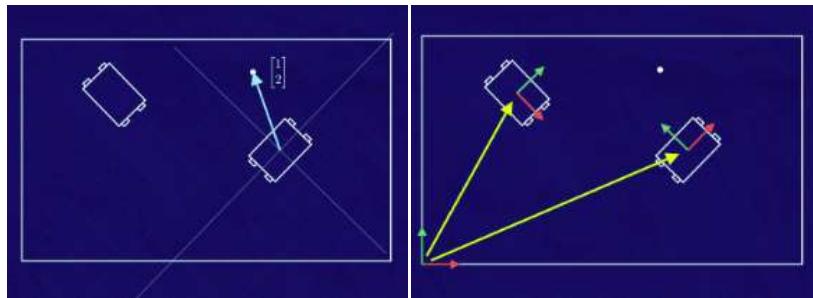
$$\mathbf{OP} = \mathbf{u} + \mathbf{v}$$

OP es el vector que va del origen del marco de coordenadas del padre al punto P.

¿Por qué son tan importantes las transformaciones?

Para responder a esta pregunta, veamos dos ejemplos de sistemas roboticos donde las transformaciones son utilies:

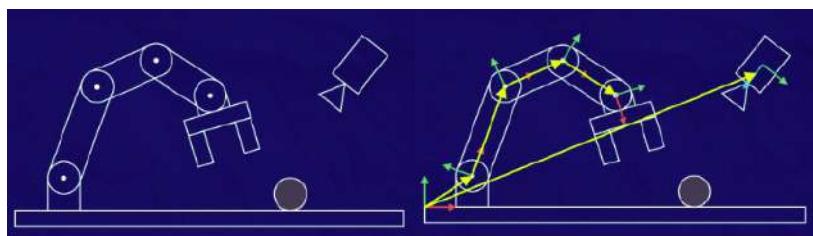
- Robot movil



Supongamos que dos robots moviles estan explorando y uno de ellos encuentra un objeto de interes en la posicion (1,2) con respecto a su marco de coordenadas. La pregunta seria, ¿cómo sabrá el otro robot cómo llegar a él?

NOTA: En los casos de robots moviles, es muy importante qque haya nodos que publiquen la ubicacion del robot dentro del entorno en el que se encuentra.

- Robot manipulador



Supongamos que una camara montada ha detectado un objetivo y un manipulador necesita mover la pinza hacia el. La pregunta seria, ¿cómo sabemos el movimiento correcto desde la pinza hasta el objetivo?

Para resolver los problemas anteriores, lo primero que debemos hacer es asignar sistemas de coordenadas o marcos a los componentes de nuestro sistema y despues definir transformaciones entre los marcos.

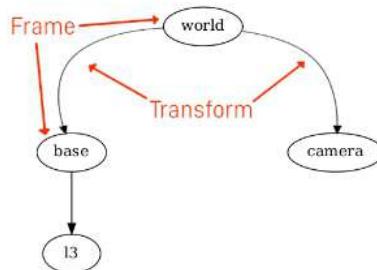
Una **transformacion** nos dice las traslaciones y rotaciones requeridas para convertir un marco en un marco diferente, y se puede revertir facilmente para ir

en sentido contrario.

Si tenemos un sistema en el que cada marco se define por su relacion con otro (y solo uno), esto creara una *estructura de arbol* en el que podremos convertir un punto conocido en un marco en cualquier otro marco en el arbol.

Podriamos crear nuestras propias transformaciones por nuestra cuenta si es que tenemos buen dominio del tema, o bien, ROS nos proporciona un sistema llamado **tf2** el cual nos ayuda a manejar estas transformaciones por nosotros.

A continuacion mostraremos una imagen que muestra una parte de un arbol de transformaciones:



Podemos usar las transformaciones para convertir puntos de cualquier marco a puntos de cualquier otro marco.

Como podemos observar, los marcos "base" y "camera" se definen en relacion al marco "world", y el marco "l3" se define en relacion al marco "base".

Los detalles de como usamos TF dependeran de muchos factores, ya sea si nuestro robot es movil o manipulador (o un manipulador movil), y si estamos ejecutando simulaciones o en hardware fisico.

Recordemos que podemos tener

- **Transformaciones estaticas** (que no cambian con el tiempo)
Las cuales se publican en el tema "/tf_static".
- **Transformaciones dinamicas** (que pueden cambiar con el tiempo).

Las cuales se publican en el tema “/tf”.

Es muy importante comprender como funcionan las cosas antes de empezar a usar paquetes que nos ayuden en ciertas tareas.

- **07-physics.urdf**

Tomando como referencia el archivo “06-flexible.urdf”, lo que se agrego fue lo siguiente:

- La etiqueta `<robot>` ahora tiene el nombre “physics”.
- Ahora agregaremos propiedades de colision y de inercia a los enlaces, y tambien agregaremos dinamica de articulaciones a las articulaciones.

Hasta ahora solo hemos especificado nuestros enlaces con un unico subelemento `<visual>`, que define como se ve el robot. Sin embargo, para que la deteccion de colisiones funcione o para simular el robot en algo como “Gazebo”, tambien necesitamos definir un elemento de colision, `<collision>`.

Sistemas de coordenadas en ROS

Son siempre en 3D y estan basadas en la regla de la mano derecha, es decir:

- X hacia adelante
- Y hacia la izquierda
- Z hacia arriba.



Puntos a tomar en cuenta:

- El elemento de colision debe estar al mismo nivel que la etiqueta `<visual>`.
- El elemento de inercia debe estar al mismo nivel que la etiqueta `<visual>`.
- El elemento de colision define su forma de la misma manera que lo hace el elemento visual, con una etiqueta `<geometry>`.

Algo a tener en cuenta es que realizar detección de colisiones para mallas es mucho mas complejo que para geometrias simples.

Tendremos algo como esto:

```
<link name="...">  
  <visual>  
    ...  
  </visual>
```

```

<collision>
  <origin rpy="..." xyz="..."/>
  <geometry>
    ...
  </geometry>
</collision>

<inertial>
  <mass value="..."/> (en kilogramos)
  <inertia ixx="..." ixy="..." ixz="..."
            iyy="..." iyz="..." izz="..."/>
</inertial>
</link>

```

Sobre las propiedades fisicas tenemos lo siguiente:

- Inercia

Es la propiedad que poseen los cuerpos de oponerse a un cambio de su estado de reposo o movimiento en que se encuentran.

La matriz de inercia rotacional de 3x3 se especifica con la etiqueta `<inertia>` (subelemento de la etiqueta `<inertial>`). Dado que es simetrica, solo puede representarse con 6 elementos (los que estan en negritas):

$$\begin{matrix}
 \mathbf{ixx} & \mathbf{ixy} & \mathbf{ixz} \\
 \mathbf{iyx} & \mathbf{iyy} & \mathbf{iyz} \\
 \mathbf{izx} & \mathbf{izy} & \mathbf{izz}
 \end{matrix}$$

Esta es una matriz que caracteriza la distribucion de masa de un solido a medida que gira en torno a un eje arbitrario, es decir, es una matriz que caracteriza la inercia rotacional que tiene el solido al girar en torno a un eje.

El tensor de inercia depende tanto de la masa como de la distibucion de masa del objeto.

Tambien se puede especificar una etiqueta <origin> para especificar el centro de gravedad y el marco de referencia inercial (en relacion con el marco de referencia del enlace).

La inercia de las primitivas geometricas (cylinder, box, sphere) se pueden calcular usando la “**lista de tensores de momento de inercia**” que se encuentra en el siguiente enlace:

https://es.wikipedia.org/wiki/Anexo:Tensores_de_momento_de_inercia_3D

- Coeficientes de contacto

Tambien es posible definir como se comportan los enlaces cuando estan en contacto entre si. Esto se hace con un subelemento de la etiqueta <collision> llamado <contact_coefficient>. Hay tres atributos para especificar:

- **mu**: coeficiente de friccion
- **kp**: coeficiente de rigidez
- **kd**: coeficiente de amortiguacion

- Dinamica de articulaciones

La forma en que se mueve la articulacion se define mediante la etiqueta <dynamics> de la articulacion. Aqui hay dos atributos:

- **friction**: la friccion fisica estatica
- **damping**: el valor de amortiguacion fisico

Si no se especifican, estos coeficientes se establecen por defecto en cero.

Otras etiquetas:

Hay dos etiquetas restantes para ayudar a definir las articulaciones:

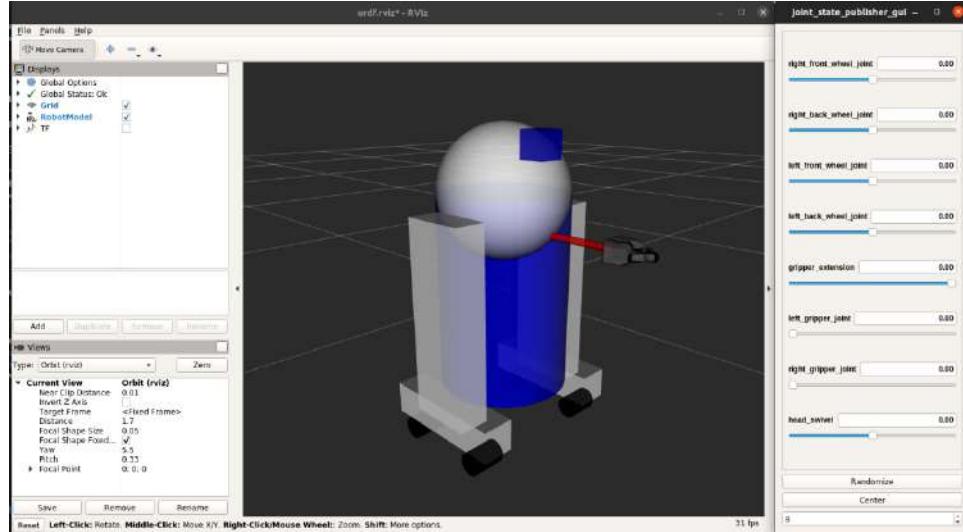
- La etiqueta <calibration ...>
- La etiqueta <safety_controller ...>

Enlaces a los que se les agrego propiedades fisicas:

- *Enlace base* (se le agrego de colision y de inercia)
- *Brazos del robot* (se les agrego de colision y de inercia)
- *Bases donde van las ruedas del robot* (se les agrego de colision y de inercia)
- *Ruedas del robot* (se les agrego de colision y de inercia)
- *Brazo de la pinza del robot* (se le agrego de colision y de inercia)
- *Pinzas del robot* (se les agrego de colision y de inercia)
- “*Dedos*” de las pinzas del robot (se les agrego de colision y de inercia)
- *Cabeza del robot* (se le agrego de colision y de inercia)
- “*Ojo*” del robot (se le agrego de colision y de inercia)

Visualizacion en RViz:

```
roslaunch urdf_tutorial display.launch  
model:=07-physics.urdf
```



No hay mucha diferencia pero ahora ya hemos agregado propiedades fisicas que nos ayudaran a poder simular nuestro modelo en Gazebo (lo cual lo veremos mas adelante).

Ahora, como siguiente paso, usaremos Xacro para limpiar un archivo URDF, es decir, para reducir la cantidad de codigo en un archivo URDF.

Introducción a Xacro

Xacro hace tres cosas que son muy útiles:

- Declaración de constantes
- Matemáticas simples
- Macros

Veremos como todos estos atajos nos ayudan a reducir el tamaño general de un archivo URDF para facilitar su lectura y mantenimiento.

Como su nombre lo indica, Xacro es un lenguaje de macros. El programa Xacro ejecuta todas las macros y genera el resultado. Un comando típico se parece al siguiente:

```
xacro model.xacro > model.urdf
```

Se puede generar automáticamente el URDF en un archivo de lanzamiento de la siguiente manera:

```
<param name="robot_description" command="xacro  
'$find <package>/robots/<name>.xacro'"/>
```

En la parte superior del archivo URF se debe especificar un espacio de nombres para que el archivo se analice correctamente. Por ejemplo, estas son dos líneas válidas de un archivo Xacro:

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

Ahora hablemos de las cosas que son muy útiles en Xacro:

- Constantes

En Xacro podemos especificar propiedades que actúan como constantes de la siguiente manera:

```
<xacro:property name="<var_name>"  
value="<value>"/>  
(Se pueden definir en cualquier lugar)
```

En un atributo de etiqueta se obtiene su valor de la siguiente manera:

```
<<label> ... <attribute>="${<var_name>}">
```

- Matematicas

Se pueden construir expresiones arbitrariamente complejas en "\${...}" usando las cuatro operaciones basicas (+, -, *, /) y parentesis.

- Macros

Es el componente mas grande y util del paquete Xacro.

Veamos algunos tipos de macros:

- Macro simple

Ejemplo:

```
<xacro:macro name="">
    <origin xyz="..." rpy="..." />
</xacro:macro>
```

```
<xacro:<name>/>
```

Aqui llamamos al macro que acabamos de crear especificando su nombre.

Lo que genera el codigo anterior es lo siguiente:

```
<origin xyz="..." rpy="..." />
```

Puntos importantes:

- El nombre debe especificarse para poder usar el macro.
- Cada instancia de <xacro:<name>/> se reemplaza con el contenido de la etiqueta <xacro:macro>.
- Si no se encuentra el Xacro con un nombre especifico, no se expandira y no generara un error.

- Macro parametrizada

Tambien se pueden parametrizar macros.

Ejemplo:

```
<xacro:macro name=""
params="">
    <<label> ... <attribute>="${<param>}">
</xacro:macro>
```

```
<xacro:<name> <param>="...">
Aqui llamamos al macro.
```

Para especificar varios parametros seria de la siguiente manera:

```
<xacro:macro name="<name>"  
params="<param1> <param2> ...">  
...  
</xacro:macro>
```

```
<xacro:<name> <param1>="..."  
<param1>="..." .../>
Aqui llamamos al macro.
```

Tambien podemos utilizar bloques enteros como parametros. Para especificar un parametro de bloque, debemos incluir un asterisco antes de su nombre de parametro, por ejemplo:

```
<xacro:macro name="<name>"  
params="*<block1>">  
    <xacro:insert_block name="<block1>" />  
    Aqui se inserta un bloque usando el comando  
    "insert_block".  
</xacro:macro>

<xacro:<name>>  
    <<label> .../>      <-- Bloque 1  
    ...                  <-- Bloque n  
    Estos bloques pueden ser lo que sea (una  
    geometria, etc.).  
    Los bloques se pueden insertar tantas veces  
    como se desee.  
</xacro:<name>>
Aqui llamamos al macro.
```

Tambien tiene bloques condicionales. Esto es util para cosas como robots configurables o para cargar diferentes complementos de Gazebo. Sigue esta sintaxis:

NOTA: La expresion debe evaluarse como “0”, “1”, “true” o “false”, ya que de lo contrario se generara un error.

```
<xacro:if value="<expression>">  
    <... Algo de XML aqui>  
</xacro:if>  
<xacro:unless value="<expression>">  
    <... Algo de XML aqui>  
</xacro:unless>
```

Tambien Xacro nos permite usar ciertos comandos ROS que admite “roslaunch” (en los archivos de lanzamiento) con “\$()”. Por ejemplo:

```
<... <attribute>="$(find xacro)"/>  
<... <attribute>="$(arg <my_var>)"/>  
Los argumentos deben especificarse en la linea de comando utilizando la siguiente sintaxis:
```

```
<my_var>:=<value>
```

Veamos un uso de practico de usar Xacro:

En nuestra creacion del model de robot R2-D2 hay algunas formas utiles en que se puede usar Xacro. Por ejemplo:

- Macro de pierna

A menudo se desea crear varios objetos de apariencia similar en diferentes ubicaciones, y a menudo habra cierta simetria en las ubicaciones, por lo que usar una macro y algunas matematicas simples nos ayudara a reducir la cantidad de codigo que se tiene que escribir, como son las dos piernas del robot.

Cabe destacar que todos los URDF contienen una etiqueta `<robot>` y solo puede haber una de estas en el archivo, por lo que no podriamos crear un mundo con multiples robots en el. Tampoco hay forma en URDF de importar otros archivos URDF directamente. En cambio, al utilizar Xacro, podemos agregar varios robots en un archivo Xacro incluyendo dentro del archivo otros archivos Xacro y utilizandolos.

Las macros no necesariamente tienen que definirse en el mismo archivo en el que se utilizan, ya que con una etiqueta de inclusion especial las macros definidas en otros archivos se pueden importar.

A continuacion lo que haremos sera usar macros en nuestro modelo de robot que creamos anteriormente.

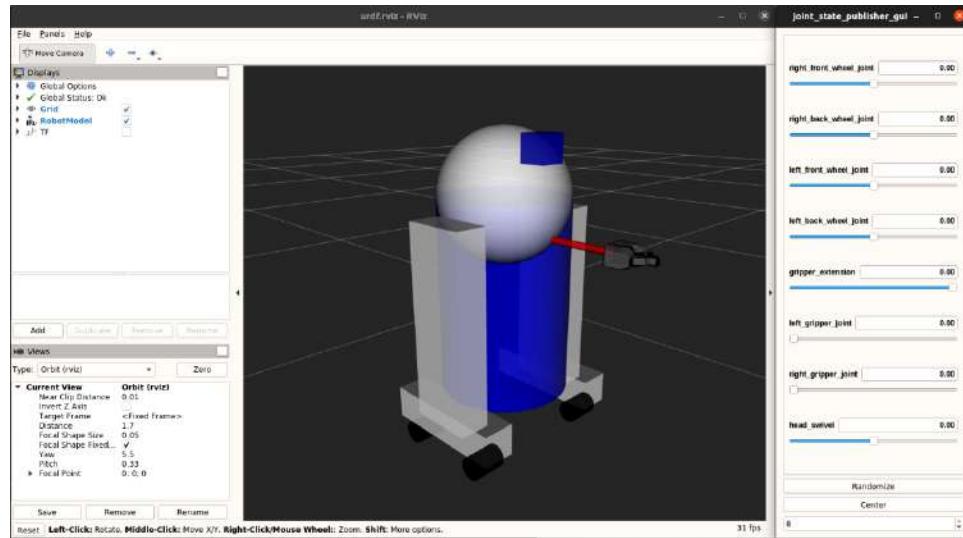
Construcion de un modelo de robot visual utilizando Xacro

Tomaremos como referencia el modelo de robot R2-D2 que creamos anteriormente.

Crearemos un archivo llamado “08-macro.urdf.xacro” el cual tendra lo mismo que el archivo “07-physics.urdf” pero ahora se utilizaran macros (como ya se explico) para reducir el codigo y para reutilizar ciertas partes del robot.

Por ultimo, ejecutamos el siguiente comando:

```
roslaunch urdf_tutorial display.launch  
model:=08-macro.urdf.xacro
```



Para convertir el archivo Xacro a URDF debemos ejecutar el siguiente comando:

```
xacro 08-macro.urdf.xacro > 08-macro.urdf
```

Como podemos observar obtenemos los mismos resultados que cuando

trabajamos con solo URDF, pero ahora el codigo se vuelve mas mantenible y facil de leer.

Introduccion a la robotica

A continuacion hablaremos sobre algunos conceptos importantes a tener en cuenta en la robotica. Esto con el motivo de entrar en contexto con lo que se estara haciendo mas adelante.

Introduccion

Para comenzar, es importante mencionar que los robots pueden ser clasificados de acuerdo a varios criterios:

- Por la tecnologia de sus controladores y actuadores (electricos, mecanicos o neumaticos).
- Por la estructura cinematica del mecanismo.
- Por la naturaleza de su movimiento.
- Por el numero de grados de libertad
- Entre otros.

Los robots se dividen en tres grandes categorias:

- *Robots seriales*

Se dice que un robot es serial si su cadena cinematica es abierta, es decir, sus enlaces se encuentran conectados uno a uno en serie.

- *Robots paralelos*

Un robot paralelo posee una cadena cinematica cerrada en la cual se pueden identificar varios lazos (como elementos en paralelo de un circuito electrico).

- *Robots hibridos*

En un robot hibrido se conectan ambos tipos de cadena cinematica.

En cuanto a la naturaleza de su movimiento, los robots pueden clasificarse en:

- *Planares*

Si es que su movimiento se define sobre planos paralelos o en dos dimensiones.

La **cinematica** se encarga del estudio del movimiento de algun cuerpo sin considerar las fuerzas que lo provocan.

La **dinamica** del robot relaciona el movimiento del robot y las fuerzas implicadas en el mismo.

- *Espaciales*

Si es que su movimiento se define en el espacio tridimensional.

Debemos tener en cuenta que para que un robot sea util debe ser controlado, por lo que para ello es necesario previamente conocer un modelo matematico del comportamiento del sistema a partir del cual generar las acciones de control requeridas.

Una **cadena cinemática** es una colección de enlaces y articulaciones conectadas.

Con respecto a este punto, sabemos que un robot esta conformado por enlaces conectados mediante articulaciones, así como el robot R2-D2 que hemos creado.

Esto solo hablando de la estructura externa del robot, ya que como bien hemos mencionado, falta la parte de la programación, el modelo matematico que describe al sistema, falta hablar sobre los actuadores, entre otros aspectos.

Con respecto a los enlaces (tambien llamados eslabones), un enlace puede ser clasificado por:

- El numero de nodos que contiene (es decir, su grado de conexión).
- El orden (es decir, el numero de enlaces que conecta).

Este seria un enlace binario (con dos nodos).

Con respecto a las articulaciones, una articulación es una conexión entre dos enlaces mediante sus nodos que permite un movimiento relativo entre ellos (los enlaces se moverán de acuerdo a las limitantes impuestas por las articulaciones). Una articulación puede ser clasificada por:

- El numero de grados de libertad que permite.
Los grados de libertad se relacionan directamente con su movilidad.

Un grado de libertad es cada uno de los movimientos básicos (giratorios y de desplazamiento) independientes que una

articulacion permite efectuar entre dos enlaces de una cadena.

Por ejemplo, un objeto libre en el espacio tiene 6 grados de libertad diferentes:

- Puede trasladarse en tres dimensiones mutuamente perpendiculares (posisionamiento)
- Puede girar en torno a tres ejes, tambien perpendiculares (orientacion).

Cualquier movimiento, sin importar que tan complejo sea, se puede resolver en estos seis movimientos basicos.

- El orden (es decir, el numero de enlaces que conecta).

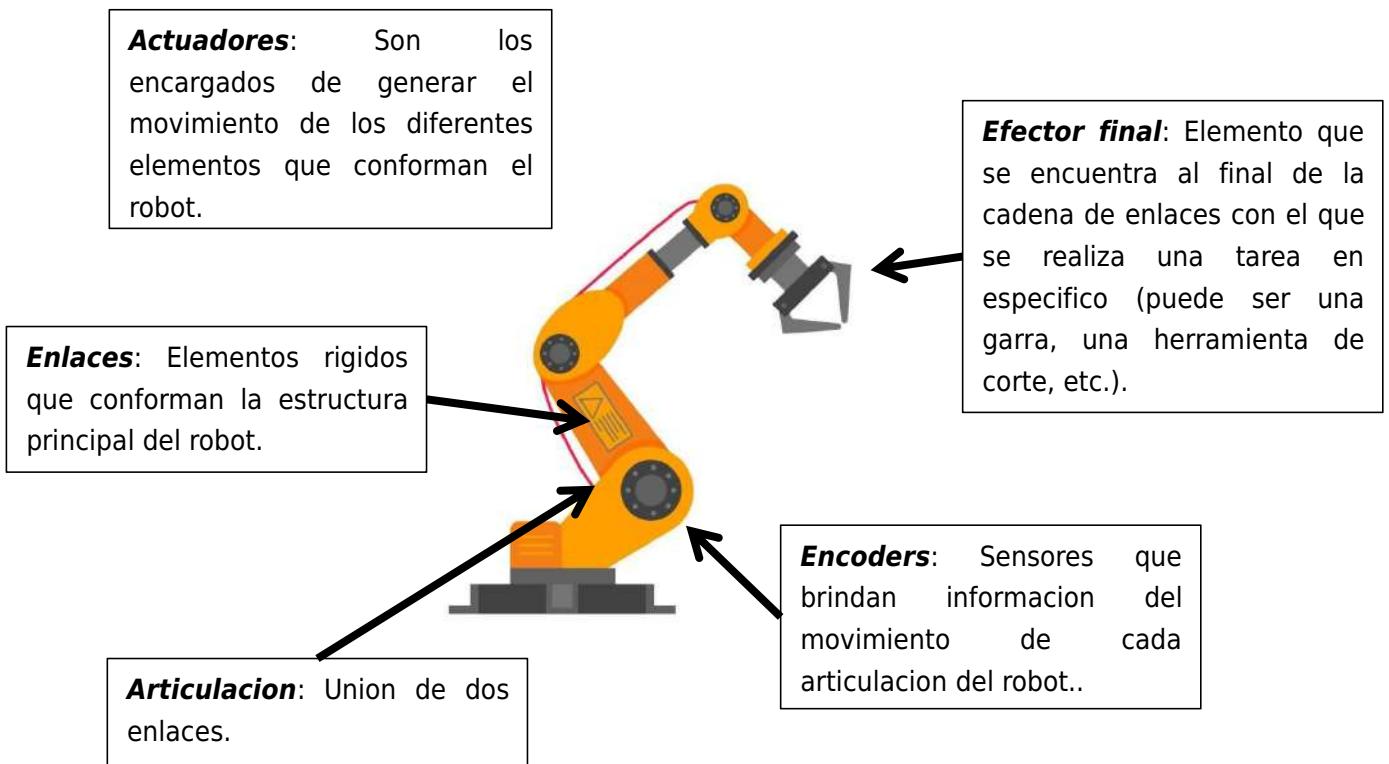
Es importante tener en cuenta que siempre tendremos:

- Un enlace rigidamente unido a un sistema de referencia, denominado base o tierra (en robots estaticos).
- Un enlace movil unido a un sistema de referencia, denominado base (en robots moviles).

La estructura cinematica del robot junto con la naturaleza de su movimiento y el numero de grados de libertad que posee determinan la complejidad del robot.

Partes de un robot

Basandonos en un brazo robotico (como ejemplo), podemos decir que tiene los siguientes componentes:



En la practica, para modelar un robot se ignoran los actuadores, los encoders y se hace una simplificacion geometrica de cada enlace.

Cinematica del robot

Se definen dos problemas con respecto a la cinematica del robot:

Llaremos \mathbf{q} al vector que contiene los valores de las articulaciones del robot, y llamemos \mathbf{x} al vector que contiene los valores de las coordenadas de la pose del robot (posicion y orientacion).

- **Cinematica directa**

Consiste en determinar la posicion y orientacion del referencia del efecto final del robot con respecto de un referencial base, dados los valores de las variables de articulacion del robot.

Por tanto, buscamos una funcion donde:

$$\mathbf{x} = \mathbf{f}(\mathbf{q})$$

- **Cinematica inversa**

Consiste en determinar los valores de las variables de articulacion del robot dados los valores de las coordenadas de la pose del robot (posicion y orientacion).

Por tanto, buscamos una funcion donde:

$$\mathbf{q} = \mathbf{g}(\mathbf{x})$$

Estos son los problemas mas basicos con los que nos podemos encontrar en la parte de la cinematica del robot.

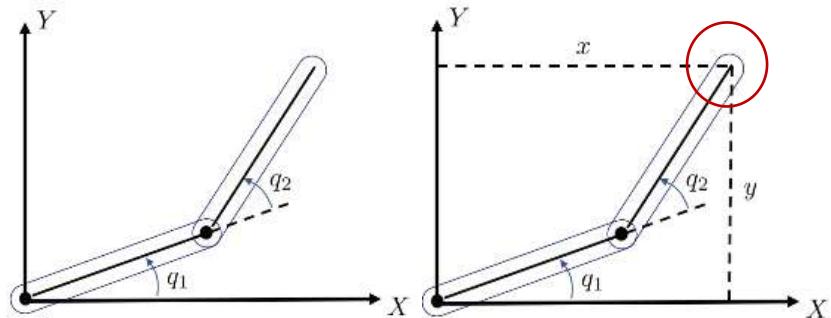
Para solucionar estos problemas, contamos con algunos metodos:

- ***Utilizando el metodo geometrico***

Consiste en ver las relaciones geometricas que hay utilizando funciones trigonometricas y medidas de enlaces para representar la pose del robot.

Veamos un ejemplo:

Calcularemos la cinematica de un robot de 2 GDL (grados de libertad).

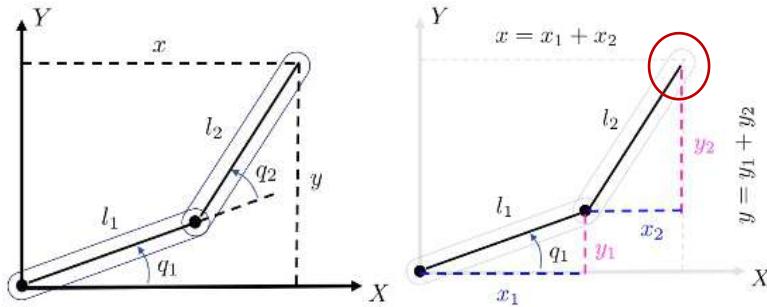


Consideraciones:

- Los angulos se miden de forma relativa.
- La posicion y orientacion del robot se mide de forma absoluta, es decir, con respecto al referencial base (X,Y).
- Las longitudes de los enlaces (eslabones) se deben de conocer.

Ahora bien, lo que haremos sera calcular la cinematica directa y la cinematica inversa del robot:

- **Cinematica directa** (recordemos que es determinar la pose del efecto final en función de las variables de las articulaciones del robot)



Tenemos que:

$$x = x_1 + y_1$$

$$y = y_1 + y_2$$

$$\sin x = \frac{\text{cateto opuesto}}{\text{hipotenusa}}$$

$$\cos x = \frac{\text{cateto adyacente}}{\text{hipotenusa}}$$

Donde:

$$\cos(q_1) = x_1 / L_1 \rightarrow x_1 = L_1 \cos(q_1)$$

$$\sin(q_1) = y_1 / L_1 \rightarrow y_1 = L_1 \sin(q_1)$$

$$\cos(q_1 + q_2) = x_2 / L_2 \rightarrow x_2 = L_2 \cos(q_1 + q_2)$$

$$\sin(q_1 + q_2) = y_2 / L_2 \rightarrow y_2 = L_2 \sin(q_1 + q_2)$$

Por tanto:

$$x = x_1 + y_1 = L_1 \cos(q_1) + L_2 \cos(q_1 + q_2)$$

$$y = y_1 + y_2 = L_1 \sin(q_1) + L_2 \sin(q_1 + q_2)$$

En forma vectorial, podemos representarla de la siguiente manera:

Recordemos que **x** representa la pose (de forma absoluta), es decir, la orientación y traslación del robot (por el momento solo es la traslación).

$$\mathbf{x} = \mathbf{f}(\mathbf{q})$$

$$\mathbf{x} = \begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} L_1 \cos q_1 + L_2 \cos(q_1 + q_2) \\ L_1 \sin q_1 + L_2 \sin(q_1 + q_2) \end{bmatrix}$$

- **Cinematica inversa** (recordemos que es determinar los valores de las variables de las articulaciones en función de la pose del efecto final)

NOTA: Es muy importante conocer de identidades trigonométricas.

Posibles escenarios:

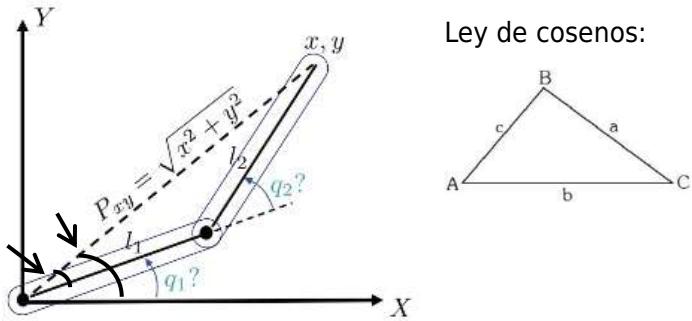
- Que el problema no tenga solución (esto ocurre cuando queremos alcanzar un punto que no es alcanzable por el robot)
- Si el punto es alcanzable, podemos tener dos posibles soluciones (podemos llegar de dos formas diferentes, ajustando los valores de las articulaciones, a dicho punto)

NOTA: Estaremos usando mucho la función "arctan2" que es una mejora para la función "arctan" (que se encuentra disponible en algunos lenguajes de programación y recibe como entrada dos parámetros) para que el problema de encontrar un valor de articulación (un ángulo) sea válido en cualquier punto del espacio de trabajo del robot.

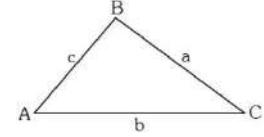
Tomemos en cuenta lo siguiente:

Notar que para la inversa de la función tangente no tenemos un dominio acotado.

Función	Dominio	Rango
$\sin^{-1} x$	$[-1, 1]$	$\left[-\frac{\pi}{2}, \frac{\pi}{2} \right]$
$\cos^{-1} x$	$[-1, 1]$	$[0, \pi]$
$\tan^{-1} x$	$(-\infty, \infty)$	$\left(-\frac{\pi}{2}, \frac{\pi}{2} \right)$



Ley de cosenos:



$$\begin{aligned} a^2 &= b^2 + c^2 - 2bc \cos A \\ b^2 &= a^2 + c^2 - 2ac \cos B \\ c^2 &= a^2 + b^2 - 2ab \cos C \end{aligned}$$

Tenemos que:

$$OP = \sqrt{x^2 + y^2}$$

Vector que va del origen al punto P (donde esta posicionado el efecto final)

$$q_2 = \pi - \beta$$

$$q_1 = \phi - \alpha$$

Donde:

Para q_2 :

$$q_2 = \pi - \beta$$

$$\cos(q_2) = \cos(\pi - \beta)$$

$$\cos(q_2) = -\cos(\beta)$$

$$-\cos(q_2) = \cos(\beta)$$

Por la ley de cosenos, tenemos:

$$|OP|^2 = L_1^2 + L_2^2 - 2L_1L_2 \cos(\beta)$$

$$\cos(\beta) = -(|OP|^2 - L_1^2 - L_2^2) / (2L_1L_2)$$

Sustituyendo tendremos:

$$-\cos(q_2) = -(|OP|^2 - L_1^2 - L_2^2) / (2L_1L_2)$$

$$\cos(q_2) = (|OP|^2 - L_1^2 - L_2^2) / (2L_1L_2) = D$$

Basandonos en la identidad trigonometrica fundamental, tenemos:

$$\sin^2(q_2) + \cos^2(q_2) = 1$$

$$\sin(q_2) = \pm \sqrt{1 - \cos^2(q_2)}$$

$$\sin(q_2) = \pm \sqrt{1 - D^2} // Sustituyendo por D$$

Por tanto:

$$\tan(q_2) = \sin(q_2)/\cos(q_2)$$

$$\tan(q_2) = \pm \sqrt(1-D^2)/D$$

$$q_2 = \arctan2(\pm \sqrt(1-D^2), D)$$

Para q_1 :

$$q_1 = \phi - \alpha$$

$$\tan(\phi) = y/x$$

$$\phi = \arctan2(y, x)$$

$$\cos(q_2) = ca/L_2$$

$$ca = L_2 \cos(q_2)$$

$$\sin(q_2) = co/L_2$$

$$co = L_2 \sin(q_2)$$

$$\tan(\alpha) = co/(L_1 + ca)$$

$$\tan(\alpha) = (L_2 \sin(q_2)) / (L_1 + L_2 \cos(q_2))$$

$$\alpha = \arctan2(L_2 \sin(q_2), L_1 + L_2 \cos(q_2))$$

Por tanto:

$$q_1 = \phi - \alpha$$

$$q_1 = \arctan2(y, x) -$$

$$\arctan2(L_2 \sin(q_2), L_1 + L_2 \cos(q_2))$$

Por tanto:

$$q_2 = \arctan2(\pm \sqrt(1-D^2), D)$$

$$q_1 = \arctan2(y, x) -$$

$$\arctan2(L_2 \sin(q_2), L_1 + L_2 \cos(q_2))$$

Notar que tenemos:

- q_2 en función de (x, y)
- q_1 en función de q_2

$$\mathbf{q} = \mathbf{g}(\mathbf{x})$$

En forma vectorial, podemos representarla de la siguiente manera:

Recordemos que \mathbf{q} representa los valores de las articulaciones del robot.

$$\mathbf{q} = \begin{bmatrix} q_1 \\ q_2 \end{bmatrix} = \begin{bmatrix} \arctan2(y, x) - \arctan2(L_2 \sin(q_2), L_1 + L_2 \cos(q_2)) \\ \arctan2(\pm \sqrt{1 - D^2}, D) \end{bmatrix}$$

El \pm es para cada una de las dos posibles soluciones (de las que se mencionaron anteriormente para alcanzar un punto).

El orden para calcular los valores es q_2 y despues q_1 (es muy importante tenerlo en cuenta).

Como podemos observar, no es buena practica utilizar este metodo para cuando tenemos un robot con mas grados de libertad y en teoria mas complejo, ya que deducir todos estos calculos podra ser muy tedioso, y con mas probabilidad de equivocarnos en los calculos.

- **Utilizando matrices de transformaciones homogeneas**

Se utilizan para representar la posicion y orientacion de un sistema girado y trasladado con respecto a un marco fijo (es una matriz que representa la transformacion de un sistema de coordenadas a otro).

La idea es que a cada punto de nuestro robot donde hay una articulacion se coloque un marco de referencia, cuya transformacion nos lleva de un marco de referencia a otro (asi como las transformaciones que se publican en el tema "/tf" cuando visualizamos nuestro robot en RViz).

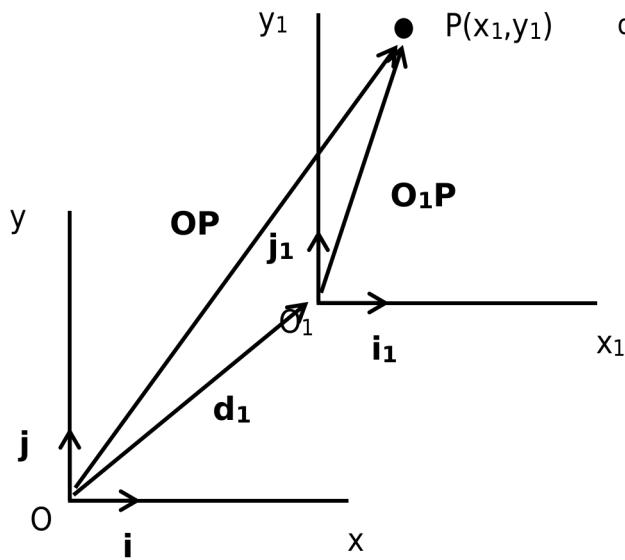
Antes de pasar a mostrar un ejemplo, veamos de donde salen estas matrices de transformaciones homogeneas (para ver que problema resuelven):

A continuacion mostraremos tres situaciones diferentes.

Es importante mencionar que estaremos trabajando con vectores y matrices, por lo que es necesario tener un poco de conocimiento en algebra lineal y en calculo vectorial para poder entender lo que se hara a continuacion.

Situacion 1: Traslacion de un marco de referencia

Tomar a \mathbf{OP} , siendo O el origen de un marco de referencia y P un punto cualquiera, como el vector que va del origen al punto P.



Marco de referencia base (siempre tendremos un marco de referencia fijo con respecto al cual se mide).

Utilizando calculo vectorial, el punto P podria representarse en el marco de referencia base de la siguiente manera:

$$\mathbf{OP} = \mathbf{d}_1 + \mathbf{O}_1\mathbf{P}$$

$$\begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} i_x & j_x \\ i_y & j_y \end{bmatrix} \begin{bmatrix} d_{1x} \\ d_{1y} \end{bmatrix} + \begin{bmatrix} i_{1x} & j_{1x} \\ i_{1y} & j_{1y} \end{bmatrix} \begin{bmatrix} x_1 \\ y_1 \end{bmatrix}$$

Es facil verlo si pensamos en que el vector \mathbf{d}_1 es una combinacion lineal de los vectores base \mathbf{i} y \mathbf{j} , y que el vector $\mathbf{O}_1\mathbf{P}$ es una combinacion lineal de los vectores \mathbf{i}_1 y \mathbf{j}_1 .

Sabiendo que $\mathbf{i}_1=\mathbf{i}$ y $\mathbf{j}_1=\mathbf{j}$, tenemos que:

$$\begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} i_x & j_x \\ i_y & j_y \end{bmatrix} \begin{bmatrix} d_{1x} \\ d_{1y} \end{bmatrix} + \begin{bmatrix} i_x & j_x \\ i_y & j_y \end{bmatrix} \begin{bmatrix} x_1 \\ y_1 \end{bmatrix}$$

$$\begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} i_x & j_x \\ i_y & j_y \end{bmatrix} \begin{bmatrix} d_{1x} + x_1 \\ d_{1y} + y_1 \end{bmatrix}$$

$$\begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} i_x(d_{1x} + x_1) + j_x(d_{1y} + y_1) \\ i_y(d_{1x} + x_1) + j_y(d_{1y} + y_1) \end{bmatrix}$$

Si tenemos que $\mathbf{i} = \begin{bmatrix} 1 \\ 0 \end{bmatrix}$ $\mathbf{j} = \begin{bmatrix} 0 \\ 1 \end{bmatrix}$, entonces:

$$\begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} d_{1x} + x_1 \\ d_{1y} + y_1 \end{bmatrix}$$

Por tanto, tendremos a x y y en función de x_1 y y_1 (sabiendo que \mathbf{d}_1 es fijo).

Ahora bien, veamos que la siguiente multiplicación de matrices nos da el mismo resultado que obtuvimos anteriormente de una manera menos complicada:

$$\begin{bmatrix} x \\ y \\ 1 \end{bmatrix} = \begin{bmatrix} i_x & j_x & d_{1x} \\ i_y & j_y & d_{1y} \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x_1 \\ y_1 \\ 1 \end{bmatrix}$$

$$\begin{bmatrix} x \\ y \\ 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & d_{1x} \\ 0 & 1 & d_{1y} \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x_1 \\ y_1 \\ 1 \end{bmatrix}$$

$$\begin{bmatrix} x \\ y \\ 1 \end{bmatrix} = \begin{bmatrix} d_{1x} + x_1 \\ d_{1y} + y_1 \\ 1 \end{bmatrix}$$

$$\begin{bmatrix} x \\ y \\ 1 \end{bmatrix} = \begin{bmatrix} i_x & j_x & d_{1x} \\ i_y & j_y & d_{1y} \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x_1 \\ y_1 \\ 1 \end{bmatrix}$$

Notar el acomodo de los vectores y la matriz (todo tiene un orden).

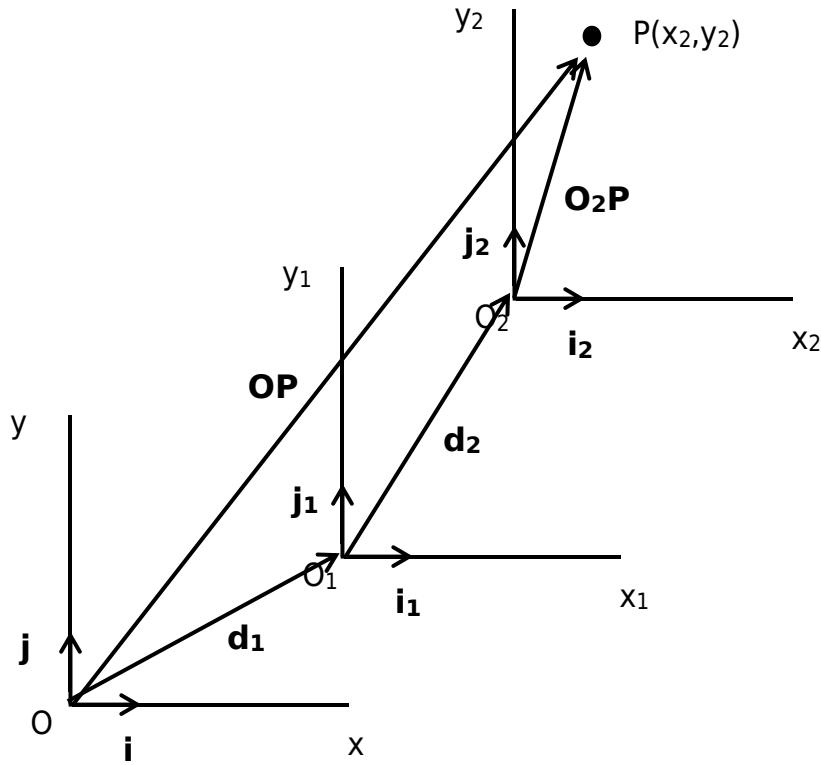
Al vector de esa forma se le conoce como **vector de coordenadas homogéneas** y a la matriz de esa forma se le conoce como **matriz de transformación homogénea**.

¿Por qué realizar multiplicación de matrices?

Lo que se busca es de alguna manera vectorizar los cálculos obtenidos anteriormente utilizando cálculo vectorial, porque como bien sabemos, las computadoras hacen muy rápido el trabajo de multiplicar matrices y, más aparte, es mucho más claro y entendible (más adelante extenderemos esta idea a traslaciones y rotaciones continuas).

Situación 2: Traslaciones continuas de un marco de

referencia



El punto P podria representarse en el marco de referencia base de la siguiente manera:

$$\mathbf{OP} = \mathbf{d}_1 + \mathbf{d}_2 + \mathbf{O}_2\mathbf{P}$$

$$\begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} i_x & j_x \\ i_y & j_y \end{bmatrix} \begin{bmatrix} d_{1x} \\ d_{1y} \end{bmatrix} + \begin{bmatrix} i_{1x} & j_{1x} \\ i_{1y} & j_{1y} \end{bmatrix} \begin{bmatrix} d_{2x} \\ d_{2y} \end{bmatrix} + \begin{bmatrix} i_{2x} & j_{2x} \\ i_{2y} & j_{2y} \end{bmatrix} \begin{bmatrix} x_2 \\ y_2 \end{bmatrix}$$

Es facil verlo si pensamos en que el vector \mathbf{d}_1 es una combinacion lineal de los vectores base \mathbf{i} y \mathbf{j} , que el vector \mathbf{d}_2 es una combinacion lineal de los vectores base \mathbf{i}_1 y \mathbf{j}_1 , y que el vector $\mathbf{O}_2\mathbf{P}$ es una combinacion lineal de los vectores \mathbf{i}_2 y \mathbf{j}_2 .

Sabiendo que $\mathbf{i}_2=\mathbf{i}_1=\mathbf{i}$ y $\mathbf{j}_2=\mathbf{j}_1=\mathbf{j}$, tenemos que:

$$\begin{aligned}
\begin{bmatrix} x \\ y \end{bmatrix} &= \begin{bmatrix} i_x & j_x \\ i_y & j_y \end{bmatrix} \begin{bmatrix} d_{1x} \\ d_{1y} \end{bmatrix} + \begin{bmatrix} i_x & j_x \\ i_y & j_y \end{bmatrix} \begin{bmatrix} d_{2x} \\ d_{2y} \end{bmatrix} \\
&\quad + \begin{bmatrix} i_x & j_x \\ i_y & j_y \end{bmatrix} \begin{bmatrix} x_2 \\ y_2 \end{bmatrix} \\
\begin{bmatrix} x \\ y \end{bmatrix} &= \begin{bmatrix} i_x & j_x \\ i_y & j_y \end{bmatrix} \begin{bmatrix} d_{1x} + d_{2x} + x_2 \\ d_{1y} + d_{2y} + y_2 \end{bmatrix} \\
\begin{bmatrix} x \\ y \end{bmatrix} &= \begin{bmatrix} i_x(d_{1x} + d_{2x} + x_2) + j_x(d_{1y} + d_{2y} + y_2) \\ i_y(d_{1x} + d_{2x} + x_2) + j_y(d_{1y} + d_{2y} + y_2) \end{bmatrix}
\end{aligned}$$

Si tenemos que $\mathbf{i} = \begin{bmatrix} 1 \\ 0 \end{bmatrix}$ $\mathbf{j} = \begin{bmatrix} 0 \\ 1 \end{bmatrix}$, entonces:

$$\begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} d_{1x} + d_{2x} + x_2 \\ d_{1y} + d_{2y} + y_2 \end{bmatrix}$$

Por tanto, tendremos a x y y en funcion de x_2 y y_2 (sabiendo que \mathbf{d}_1 y \mathbf{d}_2 son fijos).

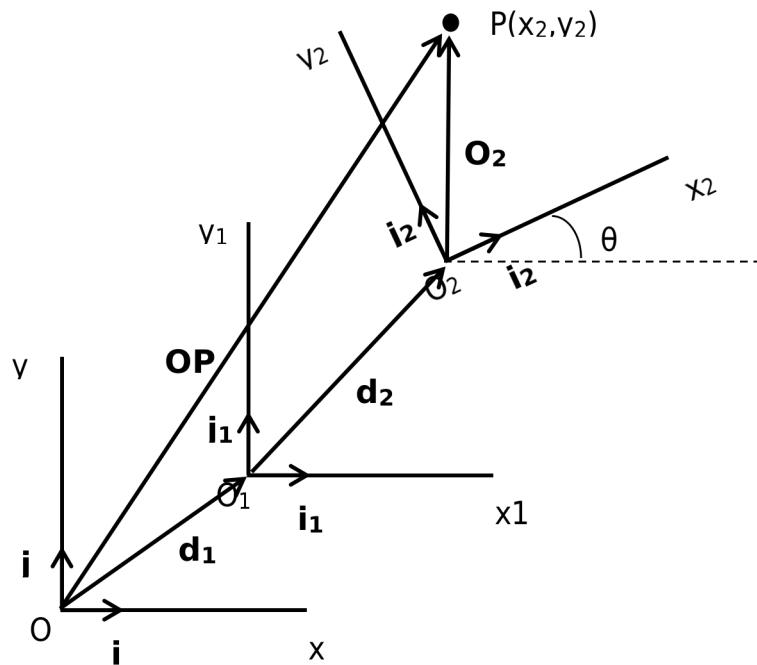
Ahora bien, veamos que la siguiente multiplicacion de matrices nos da el mismo resultado que obtuvimos anteriormente de una manera menos complicada:

Matriz de traslacion	Matriz de traslacion
$\begin{bmatrix} x \\ y \\ 1 \end{bmatrix} = \begin{bmatrix} i_x & j_x & d_{1x} \\ i_y & j_y & d_{1y} \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} i_{1x} & j_{1x} & d_{2x} \\ i_{1y} & j_{1y} & d_{2y} \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x_2 \\ y_2 \\ 1 \end{bmatrix}$	
$\begin{bmatrix} x \\ y \\ 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & d_{1x} \\ 0 & 1 & d_{1y} \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & d_{2x} \\ 0 & 1 & d_{2y} \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x_2 \\ y_2 \\ 1 \end{bmatrix}$	
$\begin{bmatrix} x \\ y \\ 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & d_{1x} + d_{2x} \\ 0 & 1 & d_{1y} + d_{2y} \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x_2 \\ y_2 \\ 1 \end{bmatrix}$	
$\begin{bmatrix} x \\ y \\ 1 \end{bmatrix} = \begin{bmatrix} d_{1x} + d_{2x} + x_2 \\ d_{1y} + d_{2y} + y_2 \\ 1 \end{bmatrix}$	

La multiplicacion de matrices se llevan a cabo en el

orden en el que se encuentran (de izquierda a derecha).

Situacion 3: Traslaciones continuas y una rotacion de un marco de referencia



El punto P podria representarse en el marco de referencia base de la siguiente manera:

$$\mathbf{OP} = \mathbf{d}_1 + \mathbf{d}_2 + \mathbf{O}_2\mathbf{P}$$

$$\begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} i_x & j_x \\ i_y & j_y \end{bmatrix} \begin{bmatrix} d_{1x} \\ d_{1y} \end{bmatrix} + \begin{bmatrix} i_{1x} & j_{1x} \\ i_{1y} & j_{1y} \end{bmatrix} \begin{bmatrix} d_{2x} \\ d_{2y} \end{bmatrix} + \begin{bmatrix} i_{2x} & j_{2x} \\ i_{2y} & j_{2y} \end{bmatrix} \begin{bmatrix} x_2 \\ y_2 \end{bmatrix}$$

Es facil verlo si pensamos en que el vector \mathbf{d}_1 es una combinacion lineal de los vectores base \mathbf{i} y \mathbf{j} , que el vector \mathbf{d}_2 es una combinacion lineal de los vectores base \mathbf{i}_1 y \mathbf{j}_1 , y que el vector $\mathbf{O}_2\mathbf{P}$ es una combinacion lineal de los vectores \mathbf{i}_2 y \mathbf{j}_2 .

Sabiendo que $\mathbf{i}_1=\mathbf{i}$, $\mathbf{j}_1=\mathbf{j}$ y \mathbf{i}_2 y \mathbf{j}_2 son una combinacion lineal

de los vectores base \mathbf{i} y \mathbf{j} en función del ángulo θ (mas adelante hablaremos sobre como representar estas rotaciones, ya que realmente los vectores \mathbf{i}_2 y \mathbf{j}_2 son una rotacion de los vectores base \mathbf{i} y \mathbf{j} , ya que la rotacion es con respecto a estos vectores), tenemos que:

$$\begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} i_x & j_x \\ i_y & j_y \end{bmatrix} \begin{bmatrix} d_{1x} \\ d_{1y} \end{bmatrix} + \begin{bmatrix} i_x & j_x \\ i_y & j_y \end{bmatrix} \begin{bmatrix} d_{2x} \\ d_{2y} \end{bmatrix} + \begin{bmatrix} i_{2x} & j_{2x} \\ i_{2y} & j_{2y} \end{bmatrix} \begin{bmatrix} x_2 \\ y_2 \end{bmatrix}$$

$$\begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} i_x & j_x \\ i_y & j_y \end{bmatrix} \begin{bmatrix} d_{1x} + d_{2x} \\ d_{1y} + d_{2y} \end{bmatrix} + \begin{bmatrix} i_{2x} & j_{2x} \\ i_{2y} & j_{2y} \end{bmatrix} \begin{bmatrix} x_2 \\ y_2 \end{bmatrix}$$

$$\begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} i_x(d_{1x} + d_{2x}) + j_x(d_{1y} + d_{2y}) + i_{2x}x_2 + j_{2x}y_2 \\ i_y(d_{1x} + d_{2x}) + j_y(d_{1y} + d_{2y}) + i_{2y}x_2 + j_{2y}y_2 \end{bmatrix}$$

Si tenemos que $\mathbf{i} = \begin{bmatrix} 1 \\ 0 \end{bmatrix}$ $\mathbf{j} = \begin{bmatrix} 0 \\ 1 \end{bmatrix}$, entonces:

$$\begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} d_{1x} + d_{2x} + i_{2x}x_2 + j_{2x}y_2 \\ d_{1y} + d_{2y} + i_{2y}x_2 + j_{2y}y_2 \end{bmatrix}$$

Por tanto, tendremos a x y y en función de x_2 y y_2 (sabiendo que \mathbf{d}_1 y \mathbf{d}_2 son fijos).

Ahora bien, veamos que la siguiente multiplicación de matrices nos da el mismo resultado que obtuvimos anteriormente de una manera menos complicada:

Recordemos que todos estos calculos los puede hacer una computadora, no es necesario realizarlos a mano.

Matriz de traslacion Matriz de traslacion Matriz de rotacion

$$\begin{bmatrix} x \\ y \\ 1 \end{bmatrix} = \underbrace{\begin{bmatrix} i_x & j_x & d_{1x} \\ i_y & j_y & d_{1y} \\ 0 & 0 & 1 \end{bmatrix}}_{\text{Matriz de traslacion}} \underbrace{\begin{bmatrix} i_{1x} & j_{1x} & d_{2x} \\ i_{1y} & j_{1y} & d_{2y} \\ 0 & 0 & 1 \end{bmatrix}}_{\text{Matriz de traslacion}} \underbrace{\begin{bmatrix} i_{2x} & j_{2x} & 0 \\ i_{2y} & j_{2y} & 0 \\ 0 & 0 & 1 \end{bmatrix}}_{\text{Matriz de rotacion}} \begin{bmatrix} x_2 \\ y_2 \\ 1 \end{bmatrix}$$

$$\begin{bmatrix} x \\ y \\ 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & d_{1x} \\ 0 & 1 & d_{1y} \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & d_{2x} \\ 0 & 1 & d_{2y} \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} i_{2x} & j_{2x} & 0 \\ i_{2y} & j_{2y} & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x_2 \\ y_2 \\ 1 \end{bmatrix}$$

$$\begin{bmatrix} x \\ y \\ 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & d_{1x} + d_{2x} \\ 0 & 1 & d_{1y} + d_{2y} \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} i_{2x} & j_{2x} & 0 \\ i_{2y} & j_{2y} & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x_2 \\ y_2 \\ 1 \end{bmatrix}$$

$$\begin{bmatrix} x \\ y \\ 1 \end{bmatrix} = \begin{bmatrix} i_{2x} & j_{2x} & d_{1x} + d_{2x} \\ i_{2y} & j_{2y} & d_{1y} + d_{2y} \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x_2 \\ y_2 \\ 1 \end{bmatrix}$$

$$\begin{bmatrix} x \\ y \\ 1 \end{bmatrix} = \begin{bmatrix} d_{1x} + d_{2x} + i_{2x}x_2 + j_{2x}y_2 \\ d_{1y} + d_{2y} + i_{2y}x_2 + j_{2y}y_2 \\ 1 \end{bmatrix}$$

La multiplicacion de matrices se llevan a cabo en el orden en el que se encuentran (de izquierda a derecha).

Con estas tres situaciones que se presentaron podemos tener una noción más clara de lo que son las matrices de transformaciones homogéneas y como es que resuelven el problema de las transformaciones de marcos de referencia utilizando la multiplicación de matrices (actualmente la mayoría de los problemas matemáticos se busca que se puedan resolver mediante la multiplicación de matrices).

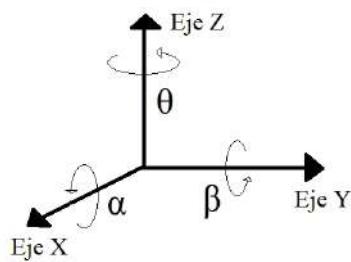
Tener en cuenta que estos ejemplos que se han presentado han sido en un plano bidimensional para facilitar el entendimiento de lo que se quiere dar a entender. Un procedimiento similar puede ser utilizado para trabajar en un plano tridimensional (que es lo más común, ya que un robot normalmente lo representamos en un plano tridimensional).

Con respecto a las matrices de rotaciones (tema que ya se mencionó pero no se profundizó), tenemos lo siguiente:

En un plano tridimensional tendremos tres posibles rotaciones (que podemos representar con tres ángulos diferentes):

Para el **sentido de la rotación** utilizaremos la **regla de la mano derecha**, como se ilustra a continuación:





- Una rotación alrededor del eje X

Pensar en el eje X como si estuviera saliendo de la pantalla.

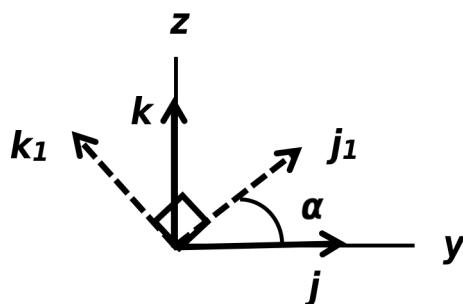
Identidades trigonometricas:

Suma de angulos

$$\sin(a \pm b) = \sin(a)\cos(b) \pm \cos(a)\sin(b)$$

$$\cos(a \pm b) = \cos(a)\cos(b) \mp \sin(a)\sin(b)$$

Nos enfocaremos solo en el plano formado por los ejes Y y Z.



Tenemos lo siguiente:

$$\mathbf{i}_1 = \mathbf{i} \text{ (no cambia este vector)}$$

$$\cos \alpha = \frac{j_{1y}}{|j_1|}$$

$$j_{1y} = \cos \alpha$$

$$\sin \alpha = \frac{j_{1z}}{|j_1|}$$

$$j_{1z} = \sin \alpha$$

$$\mathbf{j}_1 = \cos \alpha \mathbf{j} + \sin \alpha \mathbf{k}$$

$$\cos\left(\frac{\pi}{2} + \alpha\right) = \frac{k_{1y}}{|k_1|}$$

$$k_{1y} = \cos\left(\frac{\pi}{2} + \alpha\right) = -\sin \alpha$$

$$\begin{aligned}\sin\left(\frac{\pi}{2} + \alpha\right) &= \frac{k_{1z}}{|k_1|} \\ k_{1z} &= \sin\left(\frac{\pi}{2} + \alpha\right) = \cos \alpha \\ \mathbf{k}_1 &= -\sin \alpha \mathbf{j} + \cos \alpha \mathbf{k}\end{aligned}$$

Por tanto, esta es la relacion que encontramos:

$$\begin{aligned}\mathbf{i}_1 &= \mathbf{i} \\ \mathbf{j}_1 &= \cos \alpha \mathbf{j} + \sin \alpha \mathbf{k} \\ \mathbf{k}_1 &= -\sin \alpha \mathbf{j} + \cos \alpha \mathbf{k}\end{aligned}$$

Por lo que:

$$\begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} = \begin{bmatrix} i_x & 0 & 0 & 0 \\ i_y & \cos \alpha & -\sin \alpha & 0 \\ i_z & \sin \alpha & \cos \alpha & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x_1 \\ y_1 \\ z_1 \\ 1 \end{bmatrix}$$

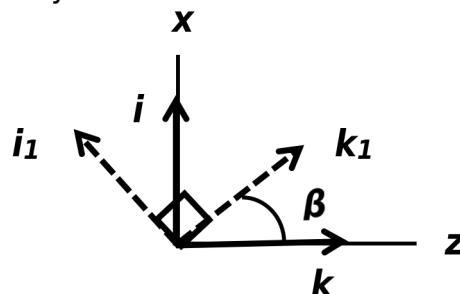
$$\text{Si } \mathbf{i} = \begin{bmatrix} 1 \\ 0 \\ 0 \\ 0 \end{bmatrix}$$

$$\begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos \alpha & -\sin \alpha & 0 \\ 0 & \sin \alpha & \cos \alpha & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x_1 \\ y_1 \\ z_1 \\ 1 \end{bmatrix}$$

Por tanto, tendremos a x , y y z en funcion de x_1 , y_1 , z_1 y α (el angulo de rotacion alrededor del eje X).

- Una rotacion alrededor del eje Y
Pensar en el eje Y como si estuviera saliendo de la pantalla.

Nos enfocaremos solo en el plano formado por los ejes X y Z.



Tenemos lo siguiente:

$$\cos\left(\frac{\pi}{2} + \beta\right) = \frac{i_{1z}}{|i_1|}$$

$$i_{1z} = \cos\left(\frac{\pi}{2} + \beta\right) = -\sin \beta$$

$$\sin\left(\frac{\pi}{2} + \beta\right) = \frac{i_{1x}}{|i_1|}$$

$$i_{1x} = \sin\left(\frac{\pi}{2} + \beta\right) = \cos \beta$$

$$\mathbf{i}_1 = \cos \beta \mathbf{i} - \sin \beta \mathbf{k}$$

$$\mathbf{j}_1 = \mathbf{j} \text{ (no cambia este vector)}$$

$$\cos \beta = \frac{k_{1z}}{|k_1|}$$

$$k_{1z} = \cos \beta$$

$$\sin \beta = \frac{k_{1x}}{|k_1|}$$

$$k_{1x} = \sin \beta$$

$$\mathbf{k}_1 = \sin \beta \mathbf{i} + \cos \beta \mathbf{k}$$

Por tanto, esta es la relacion que encontramos:

$$\mathbf{i}_1 = \cos \beta \mathbf{i} - \sin \beta \mathbf{k}$$

$$\mathbf{j}_1 = \mathbf{j}$$

$$\mathbf{k}_1 = \sin \beta \mathbf{i} + \cos \beta \mathbf{k}$$

Por lo que:

$$\begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} = \begin{bmatrix} \cos \beta & j_x & \sin \beta & 0 \\ 0 & j_y & 0 & 0 \\ -\sin \beta & j_z & \cos \beta & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x_1 \\ y_1 \\ z_1 \\ 1 \end{bmatrix}$$

$$\text{Si } \mathbf{j} = \begin{bmatrix} 0 \\ 1 \\ 0 \end{bmatrix}$$

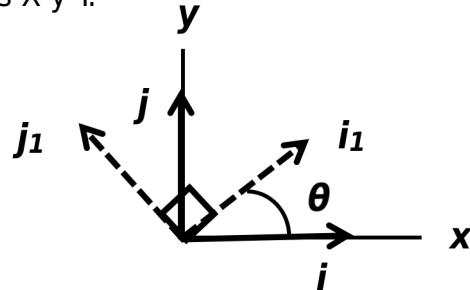
$$\begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} = \begin{bmatrix} \cos \beta & 0 & \sin \beta & 0 \\ 0 & 1 & 0 & 0 \\ -\sin \beta & 0 & \cos \beta & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x_1 \\ y_1 \\ z_1 \\ 1 \end{bmatrix}$$

Por tanto, tendremos a x , y y z en función de x_1 , y_1 , z_1 y β (el ángulo de rotación alrededor del eje Y).

- *Una rotación alrededor del eje Z*

Pensar en el eje Z como si estuviera saliendo de la pantalla.

Nos enfocaremos solo en el plano formado por los ejes X y Y.



Tenemos lo siguiente:

$$\cos \theta = \frac{i_{1x}}{|i_1|}$$

$$i_{1x} = \cos \theta$$

$$\sin \theta = \frac{i_{1y}}{|i_1|}$$

$$i_{1y} = \sin \theta$$

$$\mathbf{i}_1 = \cos \theta \mathbf{i} + \sin \theta \mathbf{j}$$

$$\cos(\frac{\pi}{2} + \theta) = \frac{j_{1x}}{|j_1|}$$

$$j_{1x} = \cos(\frac{\pi}{2} + \theta) = -\sin \theta$$

$$\sin(\frac{\pi}{2} + \theta) = \frac{j_{1y}}{|j_1|}$$

$$j_{1y} = \sin(\frac{\pi}{2} + \theta) = \cos \theta$$

$$\mathbf{j}_1 = -\sin \theta \mathbf{i} + \cos \theta \mathbf{j}$$

$$\mathbf{k}_1 = \mathbf{k} \text{ (no cambia este vector)}$$

Por tanto, esta es la relacion que encontramos:

$$\mathbf{i}_1 = \cos \theta \mathbf{i} + \sin \theta \mathbf{j}$$

$$\mathbf{j}_1 = -\sin \theta \mathbf{i} + \cos \theta \mathbf{j}$$

$$\mathbf{k}_1 = \mathbf{k}$$

Por lo que:

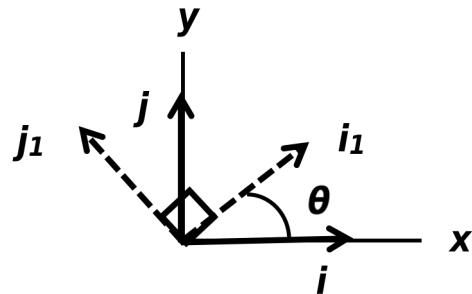
$$\begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} = \begin{bmatrix} \cos \theta & -\sin \theta & k_x & 0 \\ \sin \theta & \cos \theta & k_y & 0 \\ 0 & 0 & k_z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x_1 \\ y_1 \\ z_1 \\ 1 \end{bmatrix}$$

$$\text{Si } \mathbf{k} = \begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix}$$

$$\begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} = \begin{bmatrix} \cos \theta & -\sin \theta & 0 & 0 \\ \sin \theta & \cos \theta & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x_1 \\ y_1 \\ z_1 \\ 1 \end{bmatrix}$$

Por tanto, tendremos a x , y y z en funcion de x_1 , y_1 , z_1 y θ (el angulo de rotacion alrededor del eje Z).

En un plano bidimensional tendremos una sola rotacion (que seria como una rotacion alrededor del eje Z en un espacio tridimensional). Veamos un ejemplo:



Tenemos lo siguiente:

$$\cos \theta = \frac{i_{1x}}{|i_1|}$$

$$i_{1x} = \cos \theta$$

$$\sin \theta = \frac{i_{1y}}{|i_1|}$$

$$i_{1y} = \sin \theta$$

$$\mathbf{i}_1 = \cos \theta \mathbf{i} + \sin \theta \mathbf{j}$$

$$\cos(\frac{\pi}{2} + \theta) = \frac{j_{1x}}{|j_1|}$$

$$j_{1x} = \cos(\frac{\pi}{2} + \theta) = -\sin \theta$$

$$\sin(\frac{\pi}{2} + \theta) = \frac{j_{1y}}{|j_1|}$$

$$j_{1y} = \sin(\frac{\pi}{2} + \theta) = \cos \theta$$

$$\mathbf{j}_1 = -\sin \theta \mathbf{i} + \cos \theta \mathbf{j}$$

Por tanto, esta es la relacion que encontramos:

$$\mathbf{i}_1 = \cos \theta \mathbf{i} + \sin \theta \mathbf{j}$$

$$\mathbf{j}_1 = -\sin \theta \mathbf{i} + \cos \theta \mathbf{j}$$

Por lo que:

Matriz de rotacion

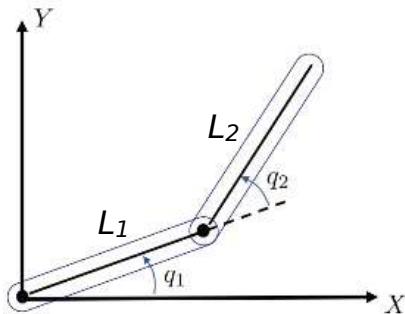
$$\begin{bmatrix} x \\ y \\ 1 \end{bmatrix} = \underbrace{\begin{bmatrix} \cos \theta & -\sin \theta & 0 \\ \sin \theta & \cos \theta & 0 \\ 0 & 0 & 1 \end{bmatrix}}_{\text{Matriz de rotacion}} \begin{bmatrix} x_1 \\ y_1 \\ 1 \end{bmatrix}$$

$$\begin{bmatrix} x \\ y \\ 1 \end{bmatrix} = \begin{bmatrix} x_1 \cos \theta - y_1 \sin \theta \\ x_1 \sin \theta + y_1 \cos \theta \\ 1 \end{bmatrix}$$

Por tanto, tendremos a x y y en funcion de x_1 , y_1 y θ (el angulo de rotacion).

Ahora veamos un ejemplo utilizando matrices de transformaciones homogeneas:

Calcularemos la cinematica de un robot de 2 GDL (grados de libertad).



Consideraciones:

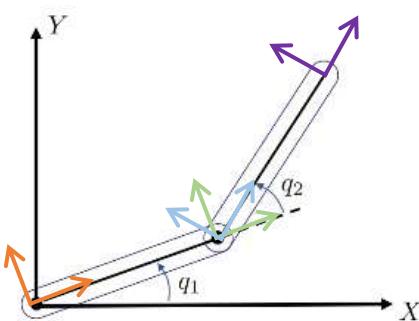
- Nuestro marco de referencia base esta conformado por los vectores base **i** y **j**.
- Siempre tendremos la transformacion de un marco de referencia a otro contenida en una matriz (marcos consecutivos).
- La multiplicacion consecutiva de las matrices de transformaciones homogeneas nos llevaran del marco de referencia del efector final al marco de referencia base (podremos obtener la pose absoluta del efector final).

Ahora bien, lo que haremos sera calcular la cinematica directa y la cinematica inversa del robot:

- **Cinematica directa** (recordemos que es determinar la pose del efector final en funcion de las variables de las articulaciones del robot)

Lo primero que haremos sera determinar las traslaciones y rotaciones, donde a cada una de estas se le asignara un marco de referencia con respecto al marco de referencia anterior.

Tenemos entonces:



1. Rotacion de q_1 con respecto al marco de referencia base (recordemos que este tipo de rotacion es como si fuera una rotacion al rededor del eje Z en un plano de tres dimensiones).
2. Traslacion de L_1 a lo largo del eje X con respecto al marco de referencia anterior.
3. Rotacion de q_2 con respecto al marco de referencia anterior.
4. Traslacion de L_2 a lo largo del eje X con respecto al marco de referencia anterior.

Hasta aqui tendriamos situado nuestro marco de referencia del efecto final, que era lo que buscabamos.

Por tanto, todo lo anterior lo podemos representar en forma matricial (como ya se vio) de la siguiente manera:

De izquierda a derecha, las matrices son: de rotacion, de traslacion, de rotacion, y de traslacion.

- La primera matriz de transformacion homogenea relaciona el marco de referencia 1 con el marco de referencia base (podriamos llamarlo marco de referencia 0).

La cual podemos llamar H_1^0

- La segunda matriz de transformacion homogenea relaciona el marco de referencia 2 con el marco de referencia 1. La cual podemos llamar H_2^1

- La tercera matriz de transformacion homogenea relaciona el marco de referencia 3 con el marco de referencia 2. La cual podemos llamar H_3^2

- La cuarta matriz de transformacion homogenea relaciona el marco de referencia 4 con el marco de referencia 3. La cual podemos llamar \mathbf{H}_4^3

La multiplicacion continua de dichas matrices nos daria la matriz de transformacion homogenea que relaciona el marco de referencia 4 con el marco de referencia base (o marco de referencia 0):

$$\mathbf{H}_4^0 = \mathbf{H}_1^0 \mathbf{H}_2^1 \mathbf{H}_3^2 \mathbf{H}_4^3$$

$$\begin{aligned} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix} &= \begin{bmatrix} i_{1x} & j_{1x} & 0 \\ i_{1y} & j_{1y} & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} i_{2x} & j_{2x} & d_{1x} \\ i_{2y} & j_{2y} & d_{1y} \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} i_{3x} & j_{3x} & 0 \\ i_{3y} & j_{3y} & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} i_{4x} & j_{4x} & d_{2x} \\ i_{4y} & j_{4y} & d_{2y} \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x_4 \\ y_4 \\ 1 \end{bmatrix} \\ &= \begin{bmatrix} \cos q_1 & -\sin q_1 & 0 \\ \sin q_1 & \cos q_1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & L_1 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} \cos q_2 & -\sin q_2 & 0 \\ \sin q_2 & \cos q_2 & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & L_2 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x_4 \\ y_4 \\ 1 \end{bmatrix} \\ \begin{bmatrix} x \\ y \\ 1 \end{bmatrix} &= \begin{bmatrix} \cos(q_1 + q_2) & -\sin(q_1 + q_2) & L_2 \cos(q_1 + q_2) + L_1 \cos q_1 \\ \sin(q_1 + q_2) & \cos(q_1 + q_2) & L_2 \sin(q_1 + q_2) + L_1 \sin q_1 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x_4 \\ y_4 \\ 1 \end{bmatrix} \\ \begin{bmatrix} x \\ y \\ 1 \end{bmatrix} &= \begin{bmatrix} L_2 \cos(q_1 + q_2) + x_4 \cos(q_1 + q_2) + L_1 \cos q_1 - y_4 \sin(q_1 + q_2) \\ L_2 \sin(q_1 + q_2) + y_4 \cos(q_1 + q_2) + L_1 \sin q_1 + x_4 \sin(q_1 + q_2) \\ 1 \end{bmatrix} \end{aligned}$$

Recordemos que \mathbf{x} representa la pose (de forma absoluta), es decir, la orientacion y traslacion del robot (por el momento solo es la traslacion).

$$\mathbf{x} = \mathbf{f}(\mathbf{q})$$

$$\mathbf{x} = \begin{bmatrix} x \\ y \end{bmatrix}$$

$$= \begin{bmatrix} L_2 \cos(q_1 + q_2) + x_4 \cos(q_1 + q_2) + L_1 \cos q_1 - y_4 \sin(q_1 + q_2) \\ L_2 \sin(q_1 + q_2) + y_4 \cos(q_1 + q_2) + L_1 \sin q_1 + x_4 \sin(q_1 + q_2) \end{bmatrix}$$

La notacion que se ha venido usando es simplemente una notacion propia, ya que en algunos libros de robotica se tiene otra notacion.

- **Cinematica inversa** (recordemos que es determinar los valores de las variables de las articulaciones en funcion de la pose del efector final)

Antes de continuar, debemos tener en claro las siguientes propiedades:

- $(\mathbf{H}_1^0)^{-1} = \mathbf{H}_0^1$

Esto lo podemos comprobar con el siguiente ejemplo:

$$\mathbf{x}_0 = \mathbf{H}_1^0 \mathbf{x}_1$$

Tenemos una relacion que relaciona el marco de referencia 1 con el marco de referencia 0.

$$(\mathbf{H}_1^0)^{-1} \mathbf{x}_0 = (\mathbf{H}_1^0)^{-1} \mathbf{H}_1^0 \mathbf{x}_1$$

$$(\mathbf{H}_1^0)^{-1} \mathbf{x}_0 = \mathbf{I}_n \mathbf{x}_1$$

$$\mathbf{x}_1 = (\mathbf{H}_1^0)^{-1} \mathbf{x}_0$$

$$\mathbf{x}_1 = \mathbf{H}_0^1 \mathbf{x}_0$$

(lo podemos reescribir de esta manera)

Tenemos una relacion que relaciona el marco de referencia 0 con el marco de referencia 1.

- La inversa de una matriz de rotacion es igual a su transpuesta.

$$\begin{bmatrix} \cos q_n & -\sin q_n & 0 \\ \sin q_n & \cos q_n & 0 \\ 0 & 0 & 1 \end{bmatrix}^{-1} = \begin{bmatrix} \cos q_n & -\sin q_n & 0 \\ \sin q_n & \cos q_n & 0 \\ 0 & 0 & 1 \end{bmatrix}^T = \begin{bmatrix} \cos q_n & \sin q_n & 0 \\ -\sin q_n & \cos q_n & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

- En general, la inversa de una matriz de transformacion homogenea esta dada por la siguiente relacion:

$$\mathbf{H} = \begin{bmatrix} \cos q_n & -\sin q_n & d_{nx} \\ \sin q_n & \cos q_n & d_{ny} \\ 0 & 0 & 1 \end{bmatrix}$$

Diremos que $\mathbf{R} = \begin{bmatrix} \cos q_n & -\sin q_n \\ \sin q_n & \cos q_n \end{bmatrix}$ (matriz de rotacion).

Diremos que $\mathbf{P} = \begin{bmatrix} d_{nx} \\ d_{ny} \end{bmatrix}$ (vector de posicion).

Diremos que $\mathbf{0} = [0 \ 0]$ (vector de ceros).

$$\mathbf{H} = \begin{bmatrix} \mathbf{R} & \mathbf{P} \\ \mathbf{0} & 1 \end{bmatrix}$$

$$\mathbf{H}^{-1} = \begin{bmatrix} \mathbf{R} & \mathbf{P} \\ \mathbf{0} & 1 \end{bmatrix}^{-1} = \begin{bmatrix} \mathbf{R}^T & -\mathbf{R}^T \mathbf{P} \\ \mathbf{0} & 1 \end{bmatrix}^T$$

Como podemos observar, la inversa se puede obtener facilmente tomando en cuenta la relacion anterior (de esta manera no recurrimos a los metodos tradicionales para calcular la inversa de una matriz, que como sabemos puede ser costoso computacionalmente para matrices de gran tamaño).

Continuando, lo que haremos sera basarnos en lo siguiente (se menciono sobre esto anteriormente):

$$\mathbf{H}_4^0 = \mathbf{H}_1^0 \mathbf{H}_2^1 \mathbf{H}_3^2 \mathbf{H}_4^3$$

$$\mathbf{H}_4^0 = \begin{bmatrix} i_{1x} & j_{1x} & 0 \\ i_{1y} & j_{1y} & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} i_{2x} & j_{2x} & d_{1x} \\ i_{2y} & j_{2y} & d_{1y} \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} i_{3x} & j_{3x} & 0 \\ i_{3y} & j_{3y} & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} i_{4x} & j_{4x} & d_{2x} \\ i_{4y} & j_{4y} & d_{2y} \\ 0 & 0 & 1 \end{bmatrix}$$

$$\mathbf{H}_4^0 = \begin{bmatrix} \cos q_1 & -\sin q_1 & 0 \\ \sin q_1 & \cos q_1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & L_1 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} \cos q_2 & -\sin q_2 & 0 \\ \sin q_2 & \cos q_2 & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & L_2 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

$$\mathbf{H}_4^0 = \begin{bmatrix} \cos(q_1 + q_2) & -\sin(q_1 + q_2) & L_2 \cos(q_1 + q_2) + L_1 \cos q_1 \\ \sin(q_1 + q_2) & \cos(q_1 + q_2) & L_2 \sin(q_1 + q_2) + L_1 \sin q_1 \\ 0 & 0 & 1 \end{bmatrix}$$

$$\begin{bmatrix} i_{nx} & j_{nx} & d_{nx} \\ i_{ny} & j_{ny} & d_{ny} \\ 0 & 0 & 1 \end{bmatrix}$$

$$= \begin{bmatrix} \cos(q_1 + q_2) & -\sin(q_1 + q_2) & L_2 \cos(q_1 + q_2) + L_1 \cos q_1 \\ \sin(q_1 + q_2) & \cos(q_1 + q_2) & L_2 \sin(q_1 + q_2) + L_1 \sin q_1 \\ 0 & 0 & 1 \end{bmatrix}$$

Matriz de transformacion homogenea que

relaciona el marco de referencia 4 con el marco de referencia 0 (o marco de referencia base).

De forma absoluta, esta matriz representa la orientacion y traslacion del marco de referencia del efector final con respecto al marco de referencia base.

$$\begin{bmatrix} i_{nx} & j_{nx} & d_{nx} \\ i_{ny} & j_{ny} & d_{ny} \\ 0 & 0 & 1 \end{bmatrix}$$

Como sabemos, estos datos los conocemos para calcular la cinematica inversa.

La idea es encontrar ecuaciones que podamos resolver y que involucren a las variables de articulacion q_1 y q_2 en terminos de las variables que conocemos.

Tendremos lo siguiente:

NOTA: Por el momento solo nos centraremos en las igualdades que involucren a las variables d_{nx} y d_{ny} (que vienen siendo las componentes del vector de posicion **P** del marco de referencia del efector final con respecto al marco de referencia base), ya que son datos que conocemos.

- $\mathbf{H}_4^0 = \mathbf{H}_1^0 \mathbf{H}_2^1 \mathbf{H}_3^2 \mathbf{H}_4^3$

$$\begin{aligned} & \begin{bmatrix} i_{nx} & j_{nx} & d_{nx} \\ i_{ny} & j_{ny} & d_{ny} \\ 0 & 0 & 1 \end{bmatrix} \\ &= \begin{bmatrix} \cos(q_1 + q_2) & -\sin(q_1 + q_2) & L_2 \cos(q_1 + q_2) + L_1 \cos q_1 \\ \sin(q_1 + q_2) & \cos(q_1 + q_2) & L_2 \sin(q_1 + q_2) + L_1 \sin q_1 \\ 0 & 0 & 1 \end{bmatrix} \end{aligned}$$

Ecuaciones de interes:

- $d_{nx} = L_2 \cos(q_1 + q_2) + L_1 \cos q_1 \quad \textcolor{red}{6.7}$

■ $d_{ny} = L_2 \sin(q_1 + q_2) + L_1 \sin q_1$ **6.7**

● $(\mathbf{H}_1^0)^{-1} \mathbf{H}_4^0 = \mathbf{H}_2^1 \mathbf{H}_3^2 \mathbf{H}_4^3$

$$\begin{bmatrix} i_{nx} \cos q_1 + i_{ny} \sin q_1 & j_{nx} \cos q_1 + j_{ny} \sin q_1 & d_{nx} \cos q_1 + d_{ny} \sin q_1 \\ i_{ny} \cos q_1 - i_{nx} \sin q_1 & j_{ny} \cos q_1 - j_{nx} \sin q_1 & d_{ny} \cos q_1 - d_{nx} \sin q_1 \\ 0 & 0 & 1 \end{bmatrix}$$

$$\begin{bmatrix} \cos q_2 & -\sin q_2 & L_1 + L_2 \cos q_2 \\ \sin q_2 & \cos q_2 & L_2 \sin q_2 \\ 0 & 0 & 1 \end{bmatrix}$$

Ecuaciones de interes:

■ $d_{nx} \cos q_1 + d_{ny} \sin q_1 = L_1 + L_2 \cos q_2$ **6.7**

■ $d_{ny} \cos q_1 - d_{nx} \sin q_1 = L_2 \sin q_2$ **6.7**

● $(\mathbf{H}_2^1)^{-1} (\mathbf{H}_1^0)^{-1} \mathbf{H}_4^0 = \mathbf{H}_2^2 \mathbf{H}_3^3 \mathbf{H}_4^3$

$$\begin{bmatrix} i_{nx} \cos q_1 + i_{ny} \sin q_1 & j_{nx} \cos q_1 + j_{ny} \sin q_1 & d_{nx} \cos q_1 - L_1 + d_{ny} \sin q_1 \\ i_{ny} \cos q_1 - i_{nx} \sin q_1 & j_{ny} \cos q_1 - j_{nx} \sin q_1 & d_{ny} \cos q_1 - d_{nx} \sin q_1 \\ 0 & 0 & 1 \end{bmatrix}$$

$$\begin{bmatrix} \cos q_2 & -\sin q_2 & L_2 \cos q_2 \\ \sin q_2 & \cos q_2 & L_2 \sin q_2 \\ 0 & 0 & 1 \end{bmatrix}$$

Ecuaciones de interes:

■ $d_{nx} \cos q_1 - L_1 + d_{ny} \sin q_1 = L_2 \cos q_2$ **6.7**

■ $d_{ny} \cos q_1 - d_{nx} \sin q_1 = L_2 \sin q_2$ **6.7**

● $(\mathbf{H}_3^2)^{-1} (\mathbf{H}_2^1)^{-1} (\mathbf{H}_1^0)^{-1} \mathbf{H}_4^0 = \mathbf{H}_4^3$

$$\begin{bmatrix} i_{nx} \cos(q_1 + q_2) + i_{ny} \sin(q_1 + q_2) & j_{nx} \cos(q_1 + q_2) + j_{ny} \sin(q_1 + q_2) & d_{nx} \cos(q_1 + q_2) + d_{ny} \sin(q_1 + q_2) - L_1 \cos q_2 \\ i_{ny} \cos(q_1 + q_2) - i_{nx} \sin(q_1 + q_2) & j_{ny} \cos(q_1 + q_2) - j_{nx} \sin(q_1 + q_2) & d_{ny} \cos(q_1 + q_2) - d_{nx} \sin(q_1 + q_2) + L_1 \sin q_2 \\ 0 & 0 & 1 \end{bmatrix}$$

$$\begin{bmatrix} 1 & 0 & L_2 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

Ecuaciones de interes:

■ $d_{nx} \cos(q_1 + q_2) + d_{ny} \sin(q_1 + q_2) - L_1 \cos q_2 = L_2$ **6.7**

■ $d_{ny} \cos(q_1 + q_2) - d_{nx} \sin(q_1 + q_2) + L_1 \sin q_2 = 0$ **6.7**

Ahora, el procedimiento para encontrar los valores de las variables de articulacion es algo complicado, ya que tenemos ecuaciones no lineales, y se requiere de un buen manejo algebraico de las ecuaciones (notar que las ecuaciones se enumeraron para referirlas mas facilmente).

Para q_2 :

De la ecuacion 1) tenemos que:

$$\cos(q_1 + q_2) = \frac{d_{nx} - L_1 \cos q_1}{L_2}$$

De la ecuacion 2) tenemos que:

$$\sin(q_1 + q_2) = \frac{d_{ny} - L_1 \sin q_1}{L_2}$$

De la ecuacion 5) tenemos que:

$$\cos q_2 = \frac{d_{nx} \cos q_1 - L_1 + d_{ny} \sin q_1}{L_2}$$

Las ecuaciones anteriores las sustituimos en la ecuacion 7):

$$d_{nx} \left(\frac{d_{nx} - L_1 \cos q_1}{L_2} \right) + d_{ny} \left(\frac{d_{ny} - L_1 \sin q_1}{L_2} \right) - L_1 \left(\frac{d_{nx} \cos q_1 - L_1 + d_{ny} \sin q_1}{L_2} \right) = L_2$$

Al desarrollar la ecuacion anterior necesitaremos hacer uso de la siguiente ecuacion (se obtiene de la ecuacion 5)):

$$d_{nx} \cos q_1 + d_{ny} \sin q_1 = L_1 + L_2 \cos q_2$$

Despues de sustituir y seguir resolviendo, al final llegaremos al siguiente resultado:

$$\cos q_2 = \frac{d_{nx}^2 + d_{ny}^2 - L_1^2 - L_2^2}{2L_1 L_2} = D$$

Donde tambien tenemos:

$$\cos^2 q_2 + \sin^2 q_2 = 1$$

$$\sin q_2 = \pm \sqrt{1 - \cos^2 q_2} = \pm \sqrt{1 - D^2}$$

Por tanto:

$$\tan q_2 = \frac{\sin q_2}{\cos q_2} = \frac{\pm \sqrt{1 - D^2}}{D}$$

$$q_2 = \arctan(\pm \sqrt{1 - D^2}, D)$$

Para q_1 :

De la ecuacion 7) tenemos que:

$$\cos(q_1 + q_2) = \frac{L_2 + L_1 \cos q_2 - d_{ny} \sin(q_1 + q_2)}{d_{nx}}$$

De la ecuacion 8) tenemos que:

$$\cos(q_1 + q_2) = \frac{d_{nx} \sin(q_1 + q_2) - L_1 \sin q_2}{d_{ny}}$$

Igualamos ambas ecuaciones:

$$\frac{L_2 + L_1 \cos q_2 - d_{ny} \sin(q_1 + q_2)}{d_{nx}} = \frac{d_{nx} \sin(q_1 + q_2) - L_1 \sin q_2}{d_{ny}}$$

$$d_{ny}L_2 + d_{ny}L_1 \cos q_2 + d_{nx}L_1 \sin q_2 = (d_{nx}^2 + d_{ny}^2) \sin(q_1 + q_2)$$

De la ecuacion 2) tenemos que:

$$\sin(q_1 + q_2) = \frac{d_{ny} - L_1 \sin q_1}{L_2}$$

Al sustituir en la ecuacion anterior tenemos que:

$$d_{ny}L_2 + d_{ny}L_1 \cos q_2 + d_{nx}L_1 \sin q_2 = (d_{nx}^2 + d_{ny}^2) \left(\frac{d_{ny} - L_1 \sin q_1}{L_2} \right)$$

Al resolver tendremos que:

$$\sin q_1 = \frac{d_{ny}(d_{nx}^2 + d_{ny}^2) - d_{ny}L_2^2 + d_{ny}L_1L_2 \cos q_2 + d_{nx}L_1L_2 \sin q_2}{L_1(d_{nx}^2 + d_{ny}^2)} = E$$

Donde tambien tenemos:

$$\cos^2 q_1 + \sin^2 q_1 = 1$$

$$\cos q_1 = \pm \sqrt{1 - \sin^2 q_1} = \pm \sqrt{1 - E^2}$$

Por tanto:

$$\tan q_1 = \frac{\sin q_1}{\cos q_1} = \frac{E}{\pm \sqrt{1 - E^2}}$$

$$q_1 = \arctan 2(E, \pm \sqrt{1 - E^2})$$

Por tanto:

$$q_2 = \arctan 2(\pm \sqrt{1 - D^2}, D)$$

$$q_1 = \arctan 2(E, \pm \sqrt{1 - E^2})$$

En forma vectorial, podemos representarla de la siguiente manera:

Recordemos que \mathbf{q} representa los valores de las articulaciones del robot.

$$\mathbf{q} = \mathbf{g}(\mathbf{x})$$

El \pm es para cada una de las dos posibles soluciones que se tienen para alcanzar un punto.

$$\mathbf{q} = \begin{bmatrix} q_1 \\ q_2 \end{bmatrix} = \begin{bmatrix} \arctan 2(E, \pm \sqrt{1 - E^2}) \\ \arctan 2(\pm \sqrt{1 - D^2}, D) \end{bmatrix}$$

Como podemos observar, calcular la cinematica inversa mediante las matrices de transformaciones homogeneas es un procedimiento bastante complicado, ya que no es clara la solucion y se tiene que realizar algunas manipulaciones algebraicas en las ecuaciones resultantes para ir encontrando las soluciones.

¿Que sigue?

¿Que hemos visto hasta el momento?

Hasta el momento solo hemos cubierto el problema de la cinematica del robot y como podemos resolverla mediante metodos geometricos y mediante matrices de transformaciones homogeneas. Pero realmente esto es una pequena parte de lo que necesitamos saber para seguir comprendiendo lo que se hara mas adelante.

Recordemos que para poder mover un robot necesitamos que los actuadores puedan mover cada articulacion del robot de manera controlada para alcanzar un objetivo (o una pose) y, dependiendo el tipo de robot, pueda realizar la tarea que se le tiene programada. Realmente intervienen muchos factores.

Hasta aqui, las visualizaciones que hemos realizado en RViz de nuestro modelo, las transformaciones entre dos marcos de coordenadas que se publican y todo lo que hemos visto de la cinematica del robot, todo queda un poco mas claro.

En resumidas cuentas, lo que necesitamos comprender o tener en cuenta cuando trabajamos con algun robot es lo siguiente:

- La descripcion de la posicion y la orientacion de los objetos que conforman al robot en el espacio tridimensional.

Con respecto a la cinematica del robot tenemos:

- Cinematica directa
- Cinematica inversa

- Analisis del movimiento del robot.

Donde se trata con:

- La velocidad del movimiento
- Las fuerzas que se requieren para ocasionar el movimiento

- Generacion de trayectorias.

Para hacer que el robot se mueva necesitamos que cada articulacion se mueva segun lo especificado por una funcion continua del tiempo (comunmente cada articulacion inicia y termina su movimiento al mismo tiempo).

A menudo una ruta se puede describir no solamente mediante un destino, sino tambien mediante algunas ubicaciones intermedias a traves de las cuales se debe pasar para llegar al destino.

El algoritmo de generacion de trayectorias debe planear todos los detalles del movimiento:

- Las velocidades para cada una de las articulaciones
- Tiempo de duracion del movimiento

- Etc.
- Diseño del robot.
Cuando se diseña un robot se debe considerar lo siguiente:
 - El numero de articulaciones y su arreglo geometrico
 - El tamaño
 - La velocidad con la que se trabaja
 - La capacidad de carga
 - La elección y ubicacion de los actuadores
 - Los sistemas de transmision
 - Los sensores de posicion interna (y algunas veces de fuerza)
 - Entre otros
- Uso de controladores
Es claro que el uso de controladores no debe faltar, ya que todo lo que hace el robot debe ser controlado de alguna manera.

Se tienen:

- Sistemas de control de posicion

Recordemos que los robots se controlan mediante los actuadores que posee, y para mover al robot se necesita, mediante los actuadores, suministrar una *fuerza* o un *momento de torsion* para occasionar el movimiento de cada enlace (de cada parte) del robot.

Aqui tendremos algoritmos que estaran vigilando los sensores de posicion y velocidad, y determinando los comandos de momento de torsion para los actuadores.

El sistema de control de posicion debe utilizar la retroalimentacion de los sensores de las articulaciones para mantener al robot en su curso.

- Sistemas de control de fuerza

Son complementarios al sistema de control de posicion (se encargan de suministrar la fuerza o momento de torsion necesarias para mover cada enlace mediante los actuadores).

- Sistemas de control de velocidad

Son complementarios al sistema de control de posicion.

- Y muchos otros (se pueden tener sistemas de control para todo)

Usualmente los robots requieren un control mixto, es decir, que tengan sistemas de control de posicion y de fuerza, ya que normalmente un robot requiere moverse y a su vez ejecutar una tarea de fuerza (ya sea sujetando algun objeto, recorriendo un objeto, etc.).

- Punto operacional

Cuando se trabaja con manipuladores (algun brazo robotico) se tiene que definir un punto central de la herramienta (ya sea de una pinza, etc.), el cual se tendra de referencia, y sera este punto el cual deba pasar por la ruta especificada por el generador de trayectorias.

Una vez que ya hemos repasado algunos conceptos de la robotica, ahora podremos comprender un poco mejor lo que se estara haciendo mas adelante, ya que mediante la programacion y la simulacion estaremos repasando todos los conceptos antes mencionados.

Con este conocimiento adquirido anteriormente es posible crear nuestras propias librerias o paquetes para calcular transformaciones, la cinematica de nuestro robot, entre otras cosas mas. Estos calculos ya no los brindan ciertos paquetes que utilizaremos mas adelante, pero siempre es bueno entender el detras de todo lo que usamos (la comprension de estos temas hace mucho mas interesante lo que se estara viendo mas adelante).

Gazebo

Introduccion

Gazebo es un simulador de multiples robots en un mundo abierto al aire libre. Es capaz de simular una serie de robots, sensores y objetos en un mundo tridimensional.

Gazebo es un software gratuito y esta disponible en distros GNU/Linux.

Algunas de las caracteristicas que presenta Gazebo son las siguientes:

- Simulacion dinamica

- Graficos 3D avanzados
- Permite el uso de sensores
- Permite el uso de plugins
- Incluye muchos modelos de robots para simular

Instalacion de Gazebo con integracion ROS

Para la distribucion ROS Melodic que estamos usando en nuestro contenedor Docker, se recomienda instalar Gazebo 9.X, asi que lo que insalemos tendra que tener la version 9.

Para instalar Gazebo tendremos que ejecutar los siguientes comandos:

Primero instalamos el paquete “wget”:

```
sudo apt-get install wget
```

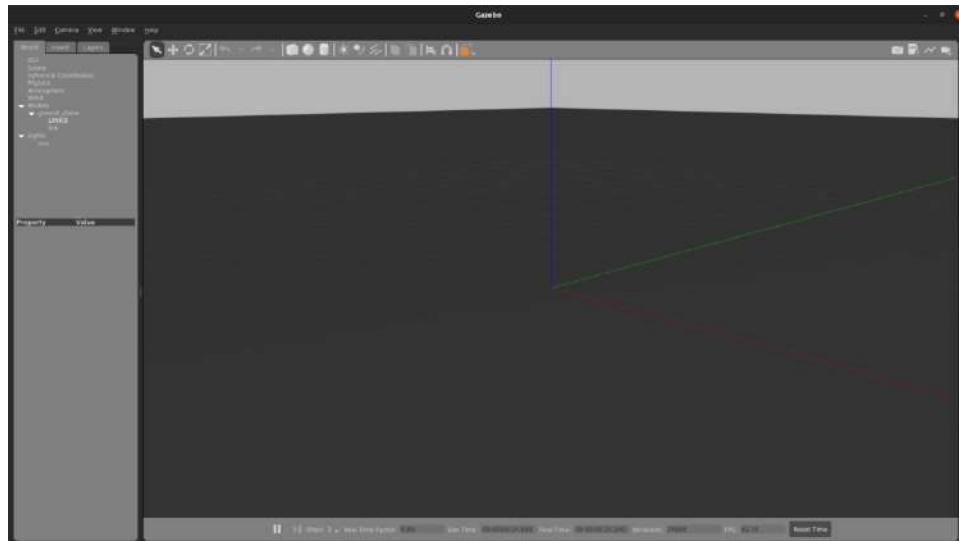
Despues ejecutamos los siguientes comandos:

- ```
sudo sh -c 'echo "deb
http://packages.osrfoundation.org/gazebo/ubuntu-stable
/etc/apt/sources.list.d/gazebo-stable.list'
```
- ```
cat /etc/apt/sources.list.d/gazebo-stable.list
```
- ```
wget https://packages.osrfoundation.org/gazebo.key -O - |
sudo apt-key add -
```
- ```
sudo apt-get update
```
- ```
sudo apt-get install gazebo9
```
- ```
sudo apt-get install libgazebo9-dev
```

NOTA: Es importante hacer un “*docker commit <id_container> <repository>:<tag>*” si es que deseamos conservar la instalacion de Gazebo en la imagen de Docker.

Para probar si Gazebo se instalo correctamente ejecutamos el siguiente comando:

```
gazebo
```



```

root@chuy:/catkin_ws# gazebo --version
Gazebo multi-robot simulator, version 9.19.0
Copyright (C) 2012 Open Source Robotics Foundation.
Released under the Apache 2 License.
http://gazebosim.org

Gazebo multi-robot simulator, version 9.19.0
Copyright (C) 2012 Open Source Robotics Foundation.
Released under the Apache 2 License.
http://gazebosim.org

root@chuy:/catkin_ws# 

```

```

root@chuy:/catkin_ws# apt list --installed | grep gazebo
WARNING: apt does not have a stable CLI interface. Use with caution in scripts.

gazebo9/unknown,now 9.19.0-1~bionic amd64 [installed]
gazebo9-common/unknown,now 9.19.0-1~bionic all [installed,automatic]
gazebo9-plugin-base/unknown,now 9.19.0-1~bionic amd64 [installed,automatic]
libgazebo9/unknown,now 9.19.0-1~bionic amd64 [installed,automatic]
libgazebo9-dev/unknown,now 9.19.0-1~bionic amd64 [installed]
root@chuy:/catkin_ws# 

```

El conjunto de paquetes ROS para interactuar con Gazebo esta contenido dentro de un metapaquete llamado “gazebo_ros_pkgs”. Para instalar dichos paquetes ejecutamos el siguiente comando:

```

sudo apt-get install ros-melodic-gazebo-ros-pkgs
ros-melodic-gazebo-ros-control

```

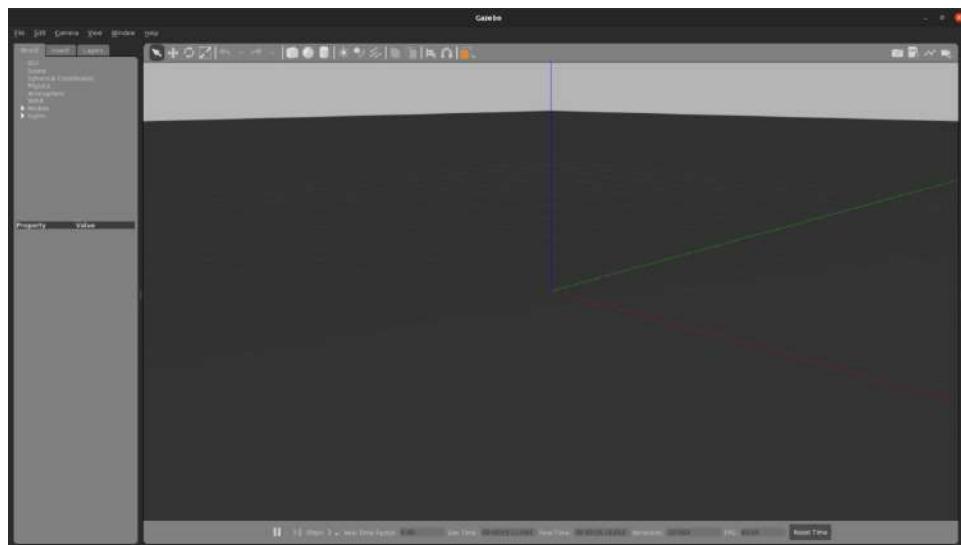
Comprobemos que todo se ha instalado correctamente:

```
root@chuy:/catkin_ws# rospack list | grep gazebo
gazebo_dev /opt/ros/melodic/share/gazebo_dev
gazebo_msgs /opt/ros/melodic/share/gazebo_msgs
gazebo_plugins /opt/ros/melodic/share/gazebo_plugins
gazebo_ros /opt/ros/melodic/share/gazebo_ros
gazebo_ros_control /opt/ros/melodic/share/gazebo_ros_control
root@chuy:/catkin_ws#
```

Podemos iniciar Gazebo ejecutando los siguientes comandos:

roscore &

rosrun gazebo_ros gazebo &



Veamos que nodos, temas y servicios estan disponibles al momento de ejecutar el comando anterior:

```

root@chuy:/catkin_ws# rosnode list
/gazebo
/rosout
root@chuy:/catkin_ws# rosnode info /gazebo
-----
Node [/gazebo]
Publications:
* /clock [rosgraph_msgs/Clock]
* /gazebo/link_states [gazebo_msgs/LinkStates]
* /gazebo/model_states [gazebo_msgs/ModelStates]
* /gazebo/parameter_descriptions [dynamic_reconfigure/ConfigDescription]
* /gazebo/parameter_updates [dynamic_reconfigure/Config]
* /rosout [rosgraph_msgs/Log]

Subscriptions:
* /clock [rosgraph_msgs/Clock]
* /gazebo/set_link_state [unknown type]
* /gazebo/set_model_state [unknown type]

Services:
* /gazebo/apply_body_wrench
* /gazebo/apply_joint_effort
* /gazebo/clear_body_wrenches
* /gazebo/clear_joint_forces
* /gazebo/delete_light
* /gazebo/delete_model
* /gazebo/get_joint_properties
* /gazebo/get_light_properties
* /gazebo/get_link_properties
* /gazebo/get_link_state
* /gazebo/get_loggers
* /gazebo/get_model_properties
* /gazebo/get_model_state
* /gazebo/get_physics_properties
* /gazebo/get_world_properties
* /gazebo/pause_physics
* /gazebo/reset_simulation
* /gazebo/reset_world
* /gazebo/set_joint_properties
* /gazebo/set_light_properties
* /gazebo/set_link_properties
* /gazebo/set_link_state
* /gazebo/set_logger_level
* /gazebo/set_model_configuration
* /gazebo/set_model_state
* /gazebo/set_parameters
* /gazebo/set_physics_properties
* /gazebo/spawn_sdf_model
* /gazebo/spawn_urdf_model
* /gazebo/unpause_physics

contacting node http://chuy:43053/ ...
Pid: 22513
Connections:
* topic: /rosout
  * to: /rosout
  * direction: outbound (60773 - 192.168.3.9:59842) [27]

```

```
root@chuy:/catkin_ws# rostopic list
/clock
/gazebo/link_states
/gazebo/model_states
/gazebo/parameter_descriptions
/gazebo/parameter_updates
/gazebo/set_link_state
/gazebo/set_model_state
/rosout
/rosout_agg
root@chuy:/catkin_ws# rosservice list
/gazebo/apply_body_wrench
/gazebo/apply_joint_effort
/gazebo/clear_body_wrenches
/gazebo/clear_joint_forces
/gazebo/delete_light
/gazebo/delete_model
/gazebo/get_joint_properties
/gazebo/get_light_properties
/gazebo/get_link_properties
/gazebo/get_link_state
/gazebo/get_loggers
/gazebo/get_model_properties
/gazebo/get_model_state
/gazebo/get_physics_properties
/gazebo/get_world_properties
/gazebo/pause_physics
/gazebo/reset_simulation
/gazebo/reset_world
/gazebo/set_joint_properties
/gazebo/set_light_properties
/gazebo/set_link_properties
/gazebo/set_link_state
/gazebo/set_logger_level
/gazebo/set_model_configuration
/gazebo/set_model_state
/gazebo/set_parameters
/gazebo/set_physics_properties
/gazebo/spawn_sdf_model
/gazebo/spawn_urdf_model
/gazebo/unpause_physics
/rosout/get_loggers
/rosout/set_logger_level
root@chuy:/catkin_ws#
```

Como podemos observar, Gazebo implementa demasiados servicios, los cuales podemos utilizar para modificar ciertos aspectos de la simulacion.

Otras maneras de iniciar Gazebo:

- *rosrun gazebo_ros gazebo (inicia el servidor y el cliente juntos)*
- *rosrun gazebo_ros gzserver (inicia solo el servidor Gazebo)*
- *rosrun gazebo_ros gzclient (inicia solo el cliente Gazebo)*
- *rosrun gazebo_ros debug (inicia el servidor Gazebo en modo depuracion)*
- *roslaunch gazebo_ros empty_world.launch (inicia Gazebo usando “roslaunch”)*

Gazebo utiliza una arquitectura distribuida con bibliotecas separadas para simulación de física, interfaz de usuario, comunicación y generación de sensores. Además, gazebo proporciona dos programas ejecutables para ejecutar simulaciones:

- **gzserver**. Es un servidor para simular la física, el renderizado y los sensores.
Este es el nucleo de Gazebo y se puede utilizar independientemente de una interfaz grafica.
- **gzclient**. Es un cliente que proporciona una interfaz gráfica para visualizar e interactuar con la simulación.

Hasta aqui, Gazebo se encuentra instalado correctamente y con la integracion ROS.

Enlaces donde se consulto la instalacion de Gazebo y de `gazebo_ros_pkgs`:

- https://classic.gazebosim.org/tutorials?tut=install_ubuntu
- https://classic.gazebosim.org/tutorials?tut=ros_installing&cat=connect_ros

Uso de URDF en Gazebo

A continuacion aprenderemos a usar un URDF en Gazebo.

Recordemos que URDF es un formato de archivo XML utilizado en ROS para describir todos los elementos de un robot.

Para usar un archivo URDF en Gazebo se deben agregar algunas etiquetas adicionales especificas de simulacion para que funcionen correctamente en Gazebo.

Gazebo trabaja con archivos SDF, pero al agregar algunas etiquetas adicionales al archivo URDF, lo que hacemos es que Gazebo podra convertir el URDF a SDF automaticamente (sin necesidad de meternos de lleno con el formato SDF).

SDF o **SDFormat**, es un formato XML que describe objetos y entornos para simuladores, visualizacion y control de robots. Desarrollado originalmente como parte del simulador de robot Gazebo.

El primer paso para que nuestro robot funcione en Gazebo es tener un archivo URDF funcional (asi como el ultimo que creamos).

Descripcion general de la conversion a Gazebo:

Hay varios pasos para que un robot URDF funcione correctamente en Gazebo. Lo siguiente es una descripcion general de los pasos:

- (Requerido) Un elemento `<inertia>` dentro de cada elemento `<link>` debe especificarse y configurarse correctamente.

Recordemos que en esta etiqueta especificamos una matriz de inercia respecto de su centro de masa con respecto a un marco de coordenadas.

- (Opcional) Agregar un elemento `<gazebo>` para cada `<link>`.
Nos permite:
 - Agregar complementos de sensor.
- (Opcional) Agregar un elemento `<gazebo>` para cada `<joint>`.
Nos permite:
 - Agregar complementos de control de actuadores.
- (Opcional) Agregar un elemento `<gazebo>` para el elemento `<robot>`.

Para empezar, lo que haremos sera lo siguiente:

- Lista de paquetes que estaremos necesitando:
 - *controller_manager* (*ya instalado*)
Esta paquete (el administrador de controladores) proporciona la infraestructura para interactuar con los controladores (proporciona diferentes herramientas para ejecutar controladores).
 - *diff_drive_controller* (*se instalara mas adelante*)
 - *position_controllers* (*se instalara mas adelante*)
 - *joint_state_controller* (*se instalara mas adelante*)
 - *gazebo_ros* (*ya instalado*)
Este paquete proporciona complementos ROS que ofrecen servicios y publicadores de mensajes para interactuar con Gazebo a traves de ROS.
 - *gazebo_ros_control* (*ya instalado*)

Este paquete nos ayuda a agregar controladores articulares que implementan las interfaces de hardware de “ros_control” (es un conjunto de paquetes que incluyen interfaces de controlador, administradores de controlador, transmisiones e interfaces de hardware).

Las **transmisiones** son los elementos encargados de transmitir el movimiento desde los actuadores hasta las articulaciones.

Mas adelante hablaremos de “ros_control”.

- ***robot_state_publisher* (ya instalado)**

Este paquete lee un archivo URDF del servidor de parametros (especificamente el parametro “robot_description”), se suscribe al tema “/joint_states” y publica las transformaciones entre diferentes marcos de coordenadas de nuestro robot utilizando la informacion del URDF y del estado de las articulaciones (los temas son “/tf” y “/tf_static”).

- ***rqt_robot_steering***

Para instalar este paquete ejecutamos el siguiente comando:

```
apt-get install ros-melodic-rqt-robot-steering
```

Este paquete proporciona un complemento GUI para dirigir un robot usando comandos “Twist”.

ROS utiliza el tipo de mensaje “Twist” para la publicacion de comandos de desplazamiento:

```
root@chuy:/catkin_ws# rosmg show geometry_msgs/Twist
geometry_msgs/Vector3 linear
  float64 x
  float64 y
  float64 z
geometry_msgs/Vector3 angular
  float64 x
  float64 y
  float64 z
root@chuy:/catkin_ws#
```

Twist.linear son las velocidades lineales en direccion a los ejes X, Y y Z.

Twist.angular son las velocidades angulares sobre los ejes X, Y y Z.

Por ejemplo, en un robot terrestre los mas probable es

que su velocidad angular sea sobre el eje Z, es decir, la velocidad de giro del robot. Y su velocidad lineal sera principalmente en direccion del eje X, es decir, la velocidad de movimiento del robot (con solo girar y avanzar se puede recorrer todo un escenario).

Por ejemplo, un nodo controlador pudiera suscribirse al tema donde se publican los mensajes “Twist” y traducir esos mensajes en señales para algunos motores para que hagan girar unas ruedas.

Este paquete lo utilizaremos ya que hayamos configurado todo para poder conducir nuestro robot R2-D2 mediante la GUI.

- *rviz (ya instalado)*
- *xacro (ya instalado)*
- *urdf_tutorial (ya creado)*

Solo como nota informativa, una lista de complementos de controlador disponibles para enviar y recibir comandos al hardware es la siguiente (tambien es posible crear nuestros propios complementos de controlador):

- *joint_state_controller*
- *position_controllers*
- *velocity_controllers*
- *effort_controllers*
- *joint_trajectory_controller*
- *Entre otros*

Algunos de ellos contienen mas de un controlador (algunos los usaremos mas adelante).

Para mas informacion en el siguiente enlace:

http://wiki.ros.org/ros_controllers

Para los paquetes que hacen falta de instalar, lo que necesitamos hacer es instalar “ros_control” y “ros_controllers” (el cual contiene una lista de complementos de controlador) ejecutando el siguiente comando:

```
sudo apt-get install ros-melodic-ros-control  
ros-melodic-ros-controllers
```

```
root@chuy:/catkin_ws# rospack list | grep controller  
ackermann_steering_controller /opt/ros/melodic/share/ackermann_steering_controller  
controller_interface /opt/ros/melodic/share/controller_interface  
controller_manager /opt/ros/melodic/share/controller_manager  
controller_manager_msgs /opt/ros/melodic/share/controller_manager_msgs  
diff_drive_controller /opt/ros/melodic/share/diff_drive_controller  
effort_controllers /opt/ros/melodic/share/effort_controllers  
force_torque_sensor_controller /opt/ros/melodic/share/force_torque_sensor_controller  
forward_command_controller /opt/ros/melodic/share/forward_command_controller  
gripper_action_controller /opt/ros/melodic/share/gripper_action_controller  
imu_sensor_controller /opt/ros/melodic/share imu_sensor_controller  
joint_state_controller /opt/ros/melodic/share/joint_state_controller  
joint_trajectory_controller /opt/ros/melodic/share/joint_trajectory_controller  
position_controllers /opt/ros/melodic/share/position_controllers  
velocity_controllers /opt/ros/melodic/share/velocity_controllers  
root@chuy:/catkin_ws#
```

Como podemos observar, ahora ya tenemos mas controladores, los cuales podremos usar.

- Crearemos un nuevo paquete ROS llamado “urdf_sim_tutorial”
En la lista de dependencias del paquete agregaremos los paquetes antes mencionados (ya que los necesitaremos).

Para crear el paquete ejecutamos el siguiente comando ubicados en “\$ROS_WORKSPACE/src”:

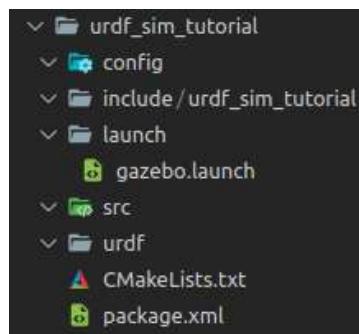
```
catkin_create_pkg urdf_sim_tutorial controller_manager  
diff_drive_controller joint_state_controller  
position_controllers gazebo_ros gazebo_ros_control  
robot_state_publisher rqt_robot_steering rviz urdf_tutorial  
xacro rospy roscpp
```

Despues creamos tres carpetas llamadas “config”, “launch” y “urdf” dentro del paquete que acabamos de crear y dentro de la carpeta “launch” creamos un archivo de lanzamiento llamado “gazebo.launch”, el cual contendra lo siguiente:

- Se declaran varios argumentos (con valores por defecto).
- Se carga por defecto el ultimo modelo de robot que se creo utilizando Xacro en el argumento “model” (el de nombre “08-macro.urdf.xacro”).
- Se inicia un mundo vacio en Gazebo incluyendo un archivo de lanzamiento (el de nombre “empty_world.launch” del paquete

"gazebo_ros") y se especifican algunos argumentos para dicho archivo de lanzamiento.

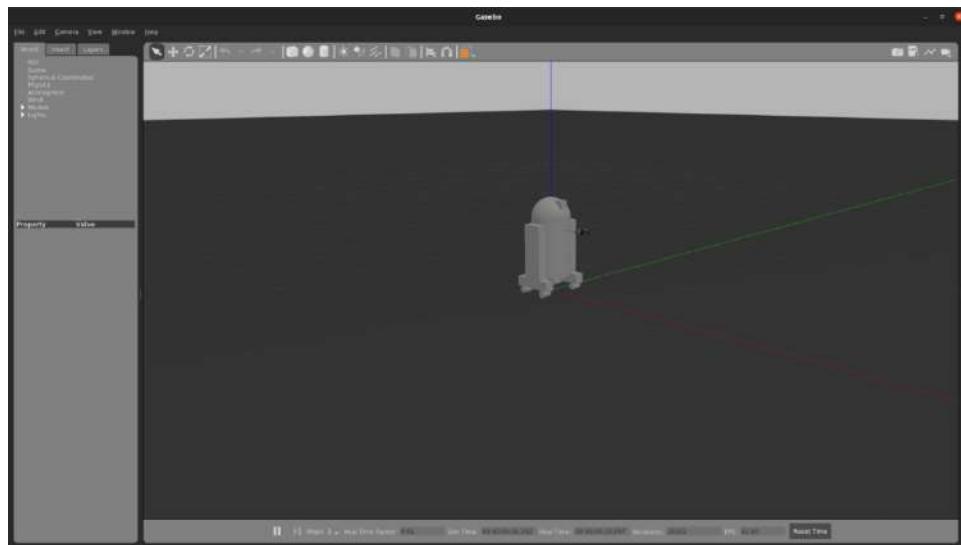
- Se carga el modelo (URDF) al servidor de parametros (en el parametro "robot_description").
- Se genera el modelo de robot en Gazebo utilizando el paquete "gazebo_ros".
- Se inicia el nodo "robot_state_publisher" (el que publica transformaciones entre marcos de coordenadas).



La carpeta "include" y "src" se crean por defecto.

Ahora ejecutamos el siguiente comando para lanzar Gazebo con nuestro modelo de robot cargado:

```
roslaunch urdf_sim_tutorial gazebo.launch &
```



Al lanzar este archivo de lanzamiento tenemos lo siguiente:

```

root@chuy:/catkin_ws# rosnode list
/gazebo
/gazebo_gui
/robot_state_publisher
/rosout
root@chuy:/catkin_ws# rosnode info /gazebo
-----
Node [/gazebo]
Publications:
* /clock [rosgraph_msgs/Clock]
* /gazebo/link_states [gazebo_msgs/LinkStates]
* /gazebo/model_states [gazebo_msgs/ModelStates]
* /gazebo/parameter_descriptions [dynamic_reconfigure/ConfigDescription]
* /gazebo/parameter_updates [dynamic_reconfigure/Config]
* /rosout [rosgraph_msgs/Log]

Subscriptions:
* /clock [rosgraph_msgs/Clock]
* /gazebo/set_link_state [unknown type]
* /gazebo/set_model_state [unknown type]

Services:
* /gazebo/apply_body_wrench
* /gazebo/apply_joint_effort
* /gazebo/clear_body_wrenches
* /gazebo/clear_joint_forces
* /gazebo/delete_light
* /gazebo/delete_model
* /gazebo/get_joint_properties
* /gazebo/get_light_properties
* /gazebo/get_link_properties
* /gazebo/get_link_state
* /gazebo/get_loggers
* /gazebo/get_model_properties
* /gazebo/get_model_state
* /gazebo/get_physics_properties
* /gazebo/get_world_properties
* /gazebo/pause_physics
* /gazebo/reset_simulation
* /gazebo/reset_world
* /gazebo/set_joint_properties
* /gazebo/set_light_properties
* /gazebo/set_link_properties
* /gazebo/set_link_state
* /gazebo/set_logger_level
* /gazebo/set_model_configuration
* /gazebo/set_model_state
* /gazebo/set_parameters
* /gazebo/set_physics_properties
* /gazebo/spawn_sdf_model
* /gazebo/spawn_urdf_model
* /gazebo/unpause_physics

```

```

root@chuy:/catkin_ws# rosnode info /gazebo_gui
-----
Node [/gazebo_gui]
Publications:
* /rosout [rosgraph_msgs/Log]

Subscriptions:
* /clock [rosgraph_msgs/Clock]

Services:
* /gazebo_gui/get_loggers
* /gazebo_gui/set_logger_level

```

Tenemos dos nodos nuevos llamados “/gazebo” y “/gazebo_gui”.

```

root@chuy:/catkin_ws# rosnode info /robot_state_publisher
-----
Node [/robot_state_publisher]
Publications:
* /rosout [rosgraph_msgs/Log]
* /tf [tf2_msgs/TFMessage]
* /tf_static [tf2_msgs/TFMessage]

Subscriptions:
* /clock [rosgraph_msgs/Clock]
* /joint_states [unknown type]

Services:
* /robot_state_publisher/get_loggers
* /robot_state_publisher/set_logger_level

```

```
root@chuy:/catkin_ws# rostopic list
/clock
/gazebo/link_states
/gazebo/model_states
/gazebo/parameter_descriptions
/gazebo/parameter_updates
/gazebo/set_link_state
/gazebo/set_model_state
/joint_states
/rosout
/rosout_agg
/tf
/tf_static
root@chuy:/catkin_ws# rosservice list
/gazebo/apply_body_wrench
/gazebo/apply_joint_effort
/gazebo/clear_body_wrenches
/gazebo/clear_joint_forces
/gazebo/delete_light
/gazebo/delete_model
/gazebo/get_joint_properties
/gazebo/get_light_properties
/gazebo/get_link_properties
/gazebo/get_link_state
/gazebo/get_loggers
/gazebo/get_model_properties
/gazebo/get_model_state
/gazebo/get_physics_properties
/gazebo/get_world_properties
/gazebo/pause_physics
/gazebo/reset_simulation
/gazebo/reset_world
/gazebo/set_joint_properties
/gazebo/set_light_properties
/gazebo/set_link_properties
/gazebo/set_link_state
/gazebo/set_logger_level
/gazebo/set_model_configuration
/gazebo/set_model_state
/gazebo/set_parameters
/gazebo/set_physics_properties
/gazebo/spawn_sdf_model
/gazebo/spawn_urdf_model
/gazebo/unpause_physics
/gazebo_gui/get_loggers
/gazebo_gui/set_logger_level
/robot_state_publisher/get_loggers
/robot_state_publisher/set_logger_level
/rosout/get_loggers
/rosout/set_logger_level
root@chuy:/catkin_ws#
```

Como vemos, Gazebo implementa muchos servicios, los cuales nos ayudan a comunicarnos con Gazebo desde ROS.

```
root@chuy:/catkin_ws# rostopic info /joint_states
Type: sensor_msgs/JointState

Publishers: None

Subscribers:
* /robot_state_publisher (http://chuy:40901/)

root@chuy:/catkin_ws#
```

Sobre este tema, anteriormente el paquete “joint_state_publisher” nos ayudaba, junto con la GUI, a publicar los estados de las articulaciones no fijas, los cuales podia leer desde un inicio del archivo URDF y mediante la GUI poder modificar dichos estados de las articulaciones. Pero ahora no contaremos con ese paquete.

Lo que realmente se busca es que el propio robot pueda proporcionar

esa informacion (los estados de las articulaciones) en el mundo real o en Gazebo (en una simulacion). Sin embargo, sin especificar eso, Gazebo no sabe publicar esa informacion.

Para que el robot sea interactivo (con nosotros y ROS), necesitamos especificar dos cosas:

- **Complementos**
- **Transmisiones**

Los cuales veremos mas adelante.

Como podemos observar, esto aun no hace nada, solo se visualiza el robot, ya que aun falta mucha informacion por agregar.

Complementos Gazebo

Para que ROS interactue con Gazebo tenemos que vincularnos dinamicamente a la biblioteca “libgazebo_ros_control.so” la cual le dire a Gazebo que hacer.

Para vincular Gazebo y ROS necesitamos agregar lo siguiente justo antes de la etiqueta de cierre </robot> en el archivo URDF:

```
<!-- Gazebo plugin for ROS Control -->
<gazebo>
  <plugin name="gazebo_ros_control" filename="libgazebo_ros_control.so">
    <robotNamespace>/</robotNamespace>
  </plugin>
</gazebo>
</robot>
```

El elemento <gazebo> es una extension del URDF que se utiliza para especificar propiedades adicionales necesarias para fines de simulacion en Gazebo. Nos permite especificar propiedades que se encuentran en el formato SDF que no estan incluidas en el formato URDF.

Lo que haremos primero sera crear tres archivos:

- *joints.yaml* (ubicado en la carpeta “config”)

Aqui especificaremos algunos parametros que se cargarán en el servidor de parametros. En concreto:

Los **controladores** generalmente se definen con archivos “yaml” (almacenados en una carpeta de configuracion).

- *type*: El tipo de controlador (aqui especificamos el controlador de estado articular)
- *publish_rate*: Tasa de publicacion (aqui especificamos un valor de 50 Hz)

```

1 # Este controlador publica el estado de las articulaciones
2 type: "joint_state_controller/JointStateController"
3 publish_rate: 50

```

Este controlador se encuentra en el paquete “joint_state_controller” y publica el estado de las articulaciones del robot en ROS directamente desde Gazebo.

Recordemos la importancia de conocer los estados de cada una de las articulaciones del robot en todo momento, ya que de estas se realizan calculos para poder mover al robot, etc. (en la vida real tenemos sensores de posicion articular, etc.).

- *09-joints.launch* (ubicado en la carpeta “launch”)

En el archivo se tiene:

- Se declaran dos argumentos, uno para especificar el modelo de robot a cargar y otro para especificar el archivo de configuracion de pantalla para RViz.
- Se incluye el archivo de lanzamiento “gazebo.launch” del paquete “urdf_sim_tutorial” y se le pasa el argumento del modelo de robot a cargar.

Recordemos que este archivo declara argumentos, inicia un mundo vacio en Gazebo, carga el modelo de robot al servidor de parametros, genera el modelo de robot en Gazebo (lo crea y lo posiciona en Gazebo) e inicia el nodo “robot_state_publisher” (el que publica transformaciones entre los diferentes marcos de coordenadas).

- Se carga el archivo de parametros que creamos anteriormente llamado “joints.yaml” y se le asigna un espacio de nombres “r2d2_joint_state_controller” para los parametros listados en el archivo.
- Se declara el nodo para RViz.
- Se declara un nodo que ejecuta el script “spawner” del paquete

“controller_manager” (el encargado de iniciar controladores, asi como de otras tareas) y se le pasa como argumento el espacio de nombres “r2d2_joint_state_controller”, el cual sera el nombre del controlador (este nombre es la raiz para todos los parametros especificos del controlador, en donde debe existir un parametro llamado “type” el cual especifica que controlador se va a cargar).

- *09-publishjoints.urdf.xacro* (ubicado en la carpeta “urdf”)

Este archivo es similar al archivo “08-macro.urdf.xacro” creado en el paquete “urdf_tutorial”, con la diferencia de que se le agrega el complemento de Gazebo justo antes de la etiqueta de cierre </robot> en el archivo, esto para vincular Gazebo y ROS.

```
<!-- Gazebo plugin for ROS Control -->
<gazebo>
  <plugin name="gazebo_ros_control" filename="libgazebo_ros_control.so">
    <robotNamespace>/</robotNamespace>
  </plugin>
</gazebo>

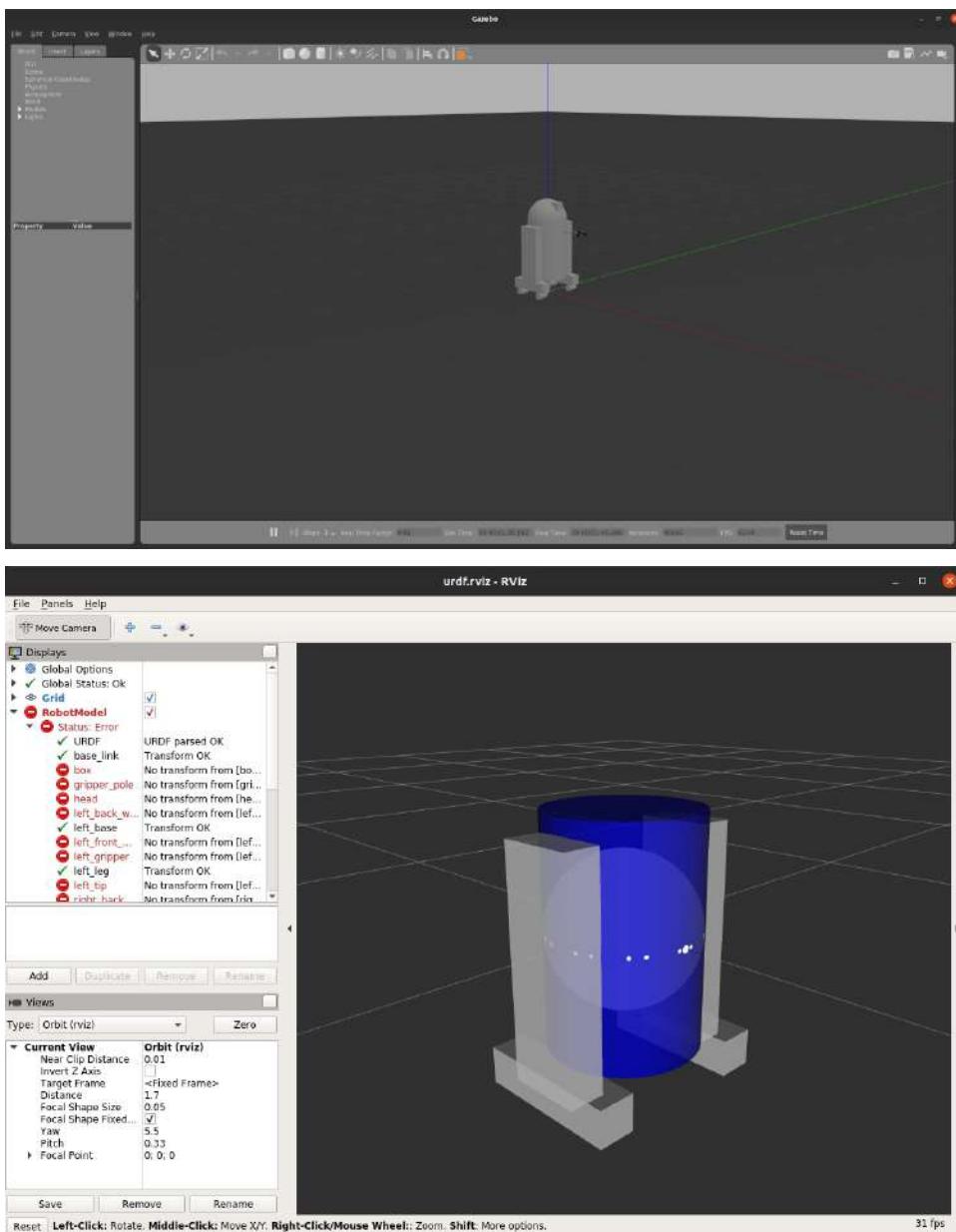
</robot>
```

NOTA: En <robotNamespace>, es el espacio de nombres ROS que se utilizara para la instancia del complemento, la cual debe ser el mismo que el espacio de nombres utilizado para definir y generar los controladores (en el archivo de lanzamiento).

Ahora ejecutamos el siguiente comando:

```
roslaunch urdf_sim_tutorial 09-joints.launch &
```

Anteriormente usabamos el nodo “joint_state_publisher” (el que publica los estados de las articulaciones en el tema “/joint_states”) para depurar nuestro URDF, y con ayuda del nodo “robot_state_publisher” poder visualizar el modelo de robot en RViz. Ahora, cuando todo esta correcto en RViz, podemos simular el robot en Gazebo, pero ahora sera Gazebo quien publique dichos estados de las articulaciones en el tema “/joint_states” por medio de complementos de controladores.



Como podemos observar, no hay transformaciones entre marcos de coordenadas donde se tienen articulaciones no fijas, ya que “robot_state_publisher” esta leyendo contenido sin informacion del tema “/joint_states”.

El comando anterior ejecutara el controlador (el “joint_state_controller”) el cual publicara en el tema “/joint_states” desde Gazebo pero sin informacion aun de las articulaciones (ya que aun no hemos especificado eso).

Por tanto, el controlador quiere saber sobre que articulaciones publicar informacion, de lo cual hablaremos mas adelante.

Como podemos observar, aun no esta del todo listo, ya que en RViz aparece nuestro robot con la cabeza en medio del cuerpo, sin ruedas, sin pinzas; solo aparecen los enlaces con articulaciones fijas. Esto se debe porque aun no se especifican las **transmisiones** (el elemento <transmission> debe definirse para vincular los actuadores a las articulaciones).

Mas adelante hablaremos sobre las transmisiones y como agregarlas.

Con respecto a lo que se esta ejecutando en este momento tenemos lo siguiente:

```
root@chuy:/catkin_ws# rosnode list
/gazebo
/gazebo_gui
/r2d2_controller_spawner
/robot_state_publisher
/rosout
/rviz
root@chuy:/catkin_ws# rostopic list
/clock
/gazebo/link_states
/gazebo/model_states
/gazebo/parameter_descriptions
/gazebo/parameter_updates
/gazebo/set_link_state
/gazebo/set_model_state
/joint_states
/rosout
/rosout_agg
/tf
/tf_static
root@chuy:/catkin_ws# rosparam list
/gazebo/auto_disable_bodies
/gazebo/cfm
/gazebo/contact_max_correcting_vel
/gazebo/contact_surface_layer
/gazebo/enable_ros_network
/gazebo/erp
/gazebo/gravity_x
/gazebo/gravity_y
/gazebo/gravity_z
/gazebo/max_contacts
/gazebo/max_update_rate
/gazebo/sor_pgs_iters
/gazebo/sor_pgs_precon_iters
/gazebo/sor_pgs_rms_error_tol
/gazebo/sor_pgs_w
/gazebo/time_step
/r2d2_joint_state_controller/publish_rate
/r2d2_joint_state_controller/type
/robot_description
/robot_state_publisher/publish_frequency
/rosdistro
/roslaunch/uris/host_chuy_43193
/rosversion
/run_id
/use_sim_time
root@chuy:/catkin_ws#
```

```
root@chuy:/catkin_ws# rostopic info /joint_states
Type: sensor_msgs/JointState

Publishers:
* /gazebo (http://chuy:40303/)

Subscribers:
* /robot_state_publisher (http://chuy:38277/)

root@chuy:/catkin_ws#
```

```
root@chuy:/catkin_ws# rostopic echo /joint_states -n 1
header:
  seq: 36855
  stamp:
    secs: 739
    nsecs: 55000000
  frame_id: ''
name: []
position: []
velocity: []
effort: []
---
root@chuy:/catkin_ws#
```

Tenemos al nodo “/r2d2_controller_spawner” el cual ejecuta nuestro controlador (el cual por el momento solo publica informacion vacia desde Gazebo del estado de las articulaciones).

```
root@chuy:/catkin_ws# rosservice list
/controller_manager/list_controller_types
/controller_manager/list_controllers
/controller_manager/load_controller
/controller_manager/reload_controller_libraries
/controller_manager/switch_controller
/controller_manager/unload_controller
```

Tenemos mas servicios pero estos son los que mas nos interesan por el momento, es decir, los que implementa el “controller_manager”.

```

root@chuy:/catkin_ws# rosservice call /controller_manager/list_controllers
controller:
  - name: "r2d2_joint_state_controller"
    state: "running"
    type: "joint state controller/JointStateController"
    claimed_resources:
      hardware_interface: "hardware_interface::JointStateInterface"
      resources: []
types: [ackermann_steering_controller/AckermannSteeringController, diff_drive_controller/DiffDriveController,
effort_controllers/GripperActionController, effort_controllers/JointEffortController,
effort_controllers/JointGroupEffortController, effort_controllers/JointGroupPositionController,
effort_controllers/JointPositionController, effort_controllers/JointTrajectoryController,
effort_controllers/JointVelocityController, force_torque_sensor_controller/ForceTorqueSensorController,
imu_sensor_controller/IMUSensorController, joint_state_controller/JointStateController,
pos_vel_acc_controllers/JointTrajectoryController, pos_vel_controllers/JointTrajectoryController,
position_controllers/GripperActionController, position_controllers/JointGroupPositionController,
position_controllers/JointPositionController, position_controllers/JointTrajectoryController,
velocity_controllers/JointGroupVelocityController, velocity_controllers/JointPositionController,
velocity_controllers/JointTrajectoryController, velocity_controllers/JointVelocityController]
base_classes: ['controller_interface::ControllerBase', 'controller_interface::ControllerBase',
'controller_interface::ControllerBase', 'controller_interface::ControllerBase']

```

Mostramos los controladores que se estan ejecutando en este momento y ademas mostramos la lista de los tipos de controladores con los que contamos (los cuales estan disponibles y podemos hacer uso).

Por tanto, lo que tenemos pendiente, y que se explicara mas adelante, es que Gazebo pueda publicar los estados de las articulaciones desde la simulacion utilizando complementos de controladores.

En un archivo URDF contamos con tres tipos diferentes de elementos `<gazebo>`:

- Uno para la etiqueta `<robot>` (la que se uso anteriormente)
Si se utiliza un elemento `<gazebo>` sin una propiedad `reference=""`, se asume que el elemento `<gazebo>` es para todo el modelo de robot.

Los elementos dentro de una etiqueta `<gazebo>`, excepto `<static>`, se insertaran directamente en la etiqueta SDF `<model>` para la SDF generada. Esto es particularmente util para complementos.

- Otro para las etiquetas `<link>`

Podemos agregar diferentes elementos individuales, asi como de:

- `<material>`
- `<gravity>`
- `<maxVel>`
- Entro otros

Recordemos que con RViz monitoreamos el estado de nuestro robot simulado en Gazebo.

Ejemplo:

```
<gazebo reference="<nombre_enlace>">  
  <mu1>0.2</mu1>  
  <mu2>0.2</mu2>  
</gazebo>
```

Cualquier elemento dentro de la etiqueta `<gazebo>` que no sea uno de los listados anteriormente, se insertara en la etiqueta SDF `<link>` correspondiente en el SDF (util para los complementos).

- Otro para las etiquetas `<joint>`

Podemos agregar diferentes elementos individuales, asi como de:

- `<stopCfm>`
- `<stopErp>`
- `<provideFeedback>`
- *Entro otros*

Cualquier elemento dentro de la etiqueta `<gazebo>` que no sea uno de los listados anteriormente, se insertara en la etiqueta SDF `<joint>` correspondiente en el SDF (util para los complementos).

Ahora veamos el uso de complementos Gazebo con ROS:

Ya vimos la importancia del complemento “`gazebo_ros_control`” y como agregarlo en un archivo URDF.

Ahora aprenderemos que tipos de complementos existen, la importancia que tienen y como poder implementar nuestros propios complementos.

Los complementos de Gazebo brindan a nuestros modelos URDF una mayor funcionalidad y pueden vincular mensajes ROS y llamadas de servicio para, por ejemplo, la salida de un sensor y la entrada de un motor.

En general, un complemento es un fragmento de codigo que se compila como una biblioteca compartida y se inserta en la simulacion. El complemento tiene acceso a toda la funcionalidad de Gazebo a traves de

las clases estandar de C++.

¿Cuando debemos utilizar un complemento?

- Cuando deseamos alterar programaticamente una simulacion.
Por ejemplo: mover modelos, responder a eventos, insertar nuevos modelos dado un conjunto de condiciones previas, etc.
- Cuando tenemos un codigo que podria beneficiar a otros y queremos compartirlo.

Los complementos Gazebo-ROS se almacenan en un paquete ROS llamado "gazebo_plugins", el cual tiene complementos Gazebo independientes de robots para sensores, motores y componentes dinamicos reconfigurables.

Actualmente existen 6 tipos de complementos:

Se debe elegir un tipo de complemento en funcion de la funcionalidad deseada.

- *World*
Se utiliza para controlar las propiedades del mundo, como el motor de fisica, la iluminacion ambiental, etc.
- *Model*
Se utiliza para controlar las articulaciones y el estado de un modelo.
- *Sensor*
Se utiliza para adquirir informacion del sensor (un sensor en especifico) y controlar las propiedades del sensor.
- *System*
Este complemento le da al usuario control sobre el proceso de inicio de Gazebo.
- *Visual*
- *GUI*

Todos ellos se pueden conectar a ROS, pero solo se puede hacer referencia a algunos tipos a traves de un archivo URDF:

- *ModelPlugins*
- *SensorPlugins*
- *VisualPlugins*

Los complementos estan diseñados para ser simples.

GNU GCC reconoce todo lo siguiente como archivos C++:

- .C
- .cc
- .cpp
- .CPP
- .c++
- .cp
- .cxx

Un complemento “World” basico contiene una clase con algunas funciones miembro.

Veamos un ejemplo de como construir un complemento “World” basico para familiarizarnos un poco con este tipo de complemento:

Lo primero que haremos sera crear una carpeta llamada “gazebo_plugin_tutorial” y dentro de la carpeta crearemos un archivo llamado “hello_world.cc”:

```
cd $ROS_WORKSPACE/src  
mkdir gazebo_plugin_tutorial  
cd gazebo_plugin_tutorial  
touch hello_world.cc  
chmod +x hello_world.cc
```

En dicho archivo se tiene lo siguiente:

- Se incluye un archivo “.hh” (de definicion de clase) el cual incluye un conjunto basico de funciones basicas de Gazebo.
- Se declara un espacio de nombres llamado “gazebo”. Todos los complementos deben estar en el espacio de nombres “gazebo”.
- Se crea una clase y se hereda de la clase “WorldPlugin”.
 - En el constructor se inicializa primero la clase base y despues la clase derivada, en donde se imprime un “Hello World!”.
 - Se declara un metodo “Load” (el cual es obligatorio), la cual recibe un elemento SDF que contiene los elementos y atributos especificos en el archivo SDF cargado.
- Finalmente, el complemento se registra en el simulador mediante la macro “GZ_REGISTER_WORLD_PLUGIN” (el unico parametro de esta macro es el nombre de la clase del complemento).

Ahora lo que haremos sera compilar el complemento. Crearemos un archivo “CMakeLists.txt” el cual contendra lo siguiente:

```

1  cmake_minimum_required(VERSION 2.8 FATAL_ERROR)
2
3  find_package(gazebo REQUIRED)
4  include_directories(${GAZEBO_INCLUDE_DIRS})
5  link_directories(${GAZEBO_LIBRARY_DIRS})
6  list(APPEND CMAKE_CXX_FLAGS "${GAZEBO_CXX_FLAGS}")
7
8  add_library(hello_world SHARED hello_world.cc) ←
9  target_link_libraries(hello_world ${GAZEBO_LIBRARIES})
10 |

```

Lo que esta subrayado en rojo es donde agregamos la libreria (libreria compartida).

Ahora creamos un directorio de compilacion llamado “build”:

```

mkdir build
cd build

```

Compilamos el codigo ejecutando los siguientes comandos:

```

cmake ../
make

```

```

root@chuy:/catkin_ws/src/gazebo_plugin_tutorial/build# cmake ../
-- The C compiler identification is GNU 7.5.0
-- The CXX compiler identification is GNU 7.5.0
-- Check for working C compiler: /usr/bin/cc
-- Check for working C compiler: /usr/bin/cc -- works
-- Detecting C compiler ABI info
-- Detecting C compiler ABI info - done
-- Detecting C compile features
-- Detecting C compile features - done
-- Check for working CXX compiler: /usr/bin/c++
-- Check for working CXX compiler: /usr/bin/c++ -- works
-- Detecting CXX compiler ABI info
root@chuy:/catkin_ws/src/gazebo_plugin_tutorial/build# ll
total 108
drwxr-xr-x 4 root root 4096 Jun 20 16:04 .
drwxr-xr-x 3 root root 4096 Jun 20 16:03 ..
-rw-r--r-- 1 root root 79613 Jun 20 16:04 CMakeCache.txt
drwxr-xr-x 5 root root 4096 Jun 20 16:04 CMakeFiles/
drwxr-xr-x 2 root root 4096 Jun 20 16:04 CMakeTmp/
-rw-r--r-- 1 root root 5007 Jun 20 16:04 Makefile
-rw-r--r-- 1 root root 1526 Jun 20 16:04 cmake_install.cmake
root@chuy:/catkin_ws/src/gazebo_plugin_tutorial/build# make
Scanning dependencies of target hello_world
[ 50%] Building CXX object CMakeFiles/hello_world.dir/hello_world.cc.o
[100%] Linking CXX shared library libhello_world.so
[100%] Built target hello_world
root@chuy:/catkin_ws/src/gazebo_plugin_tutorial/build# ll
total 360
drwxr-xr-x 4 root root 4096 Jun 20 16:05 .
drwxr-xr-x 3 root root 4096 Jun 20 16:03 ..
-rw-r--r-- 1 root root 79613 Jun 20 16:04 CMakeCache.txt
drwxr-xr-x 5 root root 4096 Jun 20 16:05 CMakeFiles/
drwxr-xr-x 2 root root 4096 Jun 20 16:04 CMakeTmp/
-rw-r--r-- 1 root root 5007 Jun 20 16:04 Makefile
-rw-r--r-- 1 root root 1526 Jun 20 16:04 cmake_install.cmake
-rwxr-xr-x 1 root root 255368 Jun 20 16:05 libhello_world.so* ←
root@chuy:/catkin_ws/src/gazebo_plugin_tutorial/build#

```

La compilacion dara como resultado una biblioteca compartida con el nombre “libhello_world.so”, la cual se puede insertar en una simulacion de Gazebo.

Ahora usemos el complemento. Una vez que tengamos un complemento compilado como una biblioteca compartida, podemos adjuntarlo a un mundo o modelo en un archivo SDF. Al inicio, Gazebo analiza el archivo SDF, localiza el complemento y carga el codigo.

NOTA: Es importante que Gazebo sea capaz de encontrar el complemento.

Cmake es una herramienta de compilacion multiplataforma. CMake mediante un archivo CMakeLists.txt, usa el comando “cmake” para convertir el archivo CMakeLists.txt en un archivo “Makefile” requerido por “make”, y finalmente se usa el comando “make” para compilar el codigo fuente para generar un programa ejecutable o una biblioteca compartida.

Creamos un archivo con extension “.world” llamado “hello.world”, el cual contendra lo siguiente:

```
cd ..  
touch hello.world  
chmod +x hello.world
```

```
1  <?xml version="1.0"?>  
2  <sdf version="1.4">  
3   <world name="default">  
4     <plugin name="hello_world" filename="/catkin_ws/src/gazebo_plugin_tutorial/build/libhello_world.so"/>  
5   </world>  
6 </sdf>
```

Especificamos la ruta completa al complemento.

Ahora lo abrimos con “gzserver”:

```
gzserver ./hello.world --verbose
```

```
root@chuy:/catkin_ws/src/gazebo_plugin_tutorial# gzserver ./hello.world --verbose  
Gazebo multi-robot simulator, version 9.19.0  
Copyright (C) 2012 Open Source Robotics Foundation.  
Released under the Apache 2 License.  
http://gazebosim.org  
  
[Msg] Waiting for master.  
[Msg] Connected to gazebo master @ http://127.0.0.1:11345  
[Msg] Publicized address: 192.168.3.9  
[Msg] Loading world file [/catkin_ws/src/gazebo_plugin_tutorial./hello.world]  
Hello World!
```

Al final de los mensajes en consola debemos ver el mensaje “Hello World!”, el cual se muestra al inicializar la clase del complemento.

Informacion sobre el comando “gzserver”:

```
root@chuy:/catkin_ws# gzserver -h
gzserver -- Run the Gazebo server.

'gzserver' [options] <world_file>

Gazebo server runs simulation and handles commandline options, starts a Master, runs World update and sensor generation loops.

Options:
  -v [ --version ]          Output version information.
  --verbose                 Increase the messages written to the terminal.
  -h [ --help ]              Produce this help message.
  -u [ --pause ]             Start the server in a paused state.
  --lockstep                Lockstep simulation so sensor update rates are
                            respected.
  -e [ --physics ] arg      Specify a physics engine
                            (ode|bullet|dart|simbody).
  -p [ --play ] arg          Play a log file.
  -r [ --record ]             Record state data.
  --record_encoding arg (=zlib) Compression encoding format for log data
                            (zlib|bz2|txt).
  --record_path arg          Absolute path in which to store state data.
  --record_period arg (=1)   Recording period (seconds).
  --record_filter arg        Recording filter (supports wildcard and regular
                            expression).
  --record_resources          Recording with model meshes and materials.
  --seed arg                  Start with a given random number seed.
  --iters arg                 Number of iterations to simulate.
  --minimal_comms            Reduce the TCP/IP traffic output by gzserver
  --server-plugin arg        Load a plugin.
  -o [ --profile ] arg       Physics preset profile name from the options in
                            the world file.

root@chuy:/catkin_ws#
```

Otra manera de crear un complemento es creando un paquete ROS con dependencias “gazebo_ros” y “roscpp”. Esto nos creara ya una estructura predefinida en la que podemos agregar o modificar archivos. Los complementos se definen en la carpeta “src” y lo que se modifica son los archivos “CMakeLists.txt” y “package.xml”. Por ultimo, solo faltaria ejecutar el comando “catkin_make” en el paquete para que se construya el paquete. La compilacion debera darnos como resultado una biblioteca compartida (de esta manera podemos tener un paquete que contenga varios complementos personalizados listos para usar).

Recordemos que los complementos nos permiten controlar modelos, sensores, propiedades del mundo e incluso la forma en que se ejecuta Gazebo.

Hasta aqui ya hemos visto un poco de lo que es el uso de complementos de Gazebo con ROS.

Recordemos que **gzserver** es el nucleo de Gazebo y se puede utilizar independientemente de una interfaz grafica.

Comunicacion ROS con Gazebo

Gazebo proporciona un conjunto de APIs ROS que permite a los usuarios modificar y obtener informacion sobre varios aspectos del mundo simulado en Gazebo.

Terminologias:

- La pose y la torsion de un objeto rigido se conoce como su estado.
- Un objeto tiene propiedades intrinsecas (que es propio o caracteristico de el) como coeficientes de masa y friccion.
- En Gazebo, un cuerpo se refiere a un cuerpo rigido, sinonimo de enlace en el contexto URDF.
- Un modelo de Gazebo es un conglomerado (que esta compuesto) de cuerpos conectados por articulaciones.

Acerca del “gazebo_ros_api_plugin”:

El complemento “gazebo_ros_api_plugin”, ubicado en el paquete “gazebo_ros”, inicializaun nodo ROS llamado “gazebo” (este complemento solo se carga con “gzserver”).

Esta API ROS permite al usuario manipular las propiedades del entorno de simulacion desde ROS.

Acerca del “gazebo_ros_path_plugin”:

Proveniente tambien del paquete “gazebo_ros”, este complemento simplemente permite a Gazebo encontrar recursos ROS, es decir, resolver nombres de rutas de paquetes ROS.

Parametros publicados de Gazebo:

- */use_sim_time (de tipo booleano)*
Si es verdadero significa que “gazebo_ros” esta publicando en el tema ROS “/clock” para proporcionar un sistema ROS con tiempo sincronizado con la simulacion.

Temas de suscripciones a Gazebo:

- *~/set_link_state*

- Establece el estado (pose/giro) de un enlace.
- `~/set_model_state`
Establece el estado (pose/giro) de un modelo.

Temas de publicaciones de Gazebo:

- `/clock`
Publica el tiempo de simulacion.
- `~/link_states`
Publica los estados de todos los enlaces en la simulacion.
- `~/model_states`
Publica los estados de todos los modelos en la simulacion.

Servicios de Gazebo:

- *Crear y destruir modelos en simulacion*
Estos servicios permiten al usuario generar y destruir modelos dinamicamente en la simulacion.
 - `~/spaw_urdf_model`
Utilice este servicio para generar un modelo a partir de un URDF.
 - `~/spaw_sdf_model`
Utilice este servicio para generar un modelo a partir de un SDF.
 - `~/delete_model`
Utilice este servicio para eliminar un modelo de la simulacion (basta con saber el nombre del modelo).
- *Captadores de estado y de propiedad*
Estos servicios permiten al usuario establecer informacion de estado y propiedad sobre la simulacion y los objetos en la simulacion.
- *Establecedores de estado y de propiedad*
Estos servicios permiten al usuario recuperar informacion sobre el estado y las propiedades de la simulacion y los objetos en la simulacion.
- *Control de fuerza*
Estos servicios permiten al usuario aplicar fuerzas a cuerpos y articulaciones en simulacion.
- *Control de simulacion*
Estos servicios permiten al usuario pausar y reanudar la fisica en la simulacion.

RViz - Gazebo

Introducción

Con RViz podemos monitorizar el estado de nuestro robot simulado en Gazebo.

Anteriormente habíamos agregado un complemento a nuestro modelo de robot desde el archivo URDF. Lo que hemos hecho fue vincularlos dinámicamente a la biblioteca ROS que le dirá a Gazebo qué hacer. Ya que lo que buscamos es que Gazebo pueda publicar el estado de las articulaciones y que RViz pueda construir bien el robot tal cual como se encuentra en la simulación.

Más adelante veremos conceptos como: controladores, complementos, transmisiones, interfaces, etc. Además configuraremos controladores simulados para activar las articulaciones de nuestro robot.

También hablaremos sobre los paquetes "ros_control", los cuales contienen interfaces de controlador, administradores de controlador, transmisiones e interfaces de hardware, los cuales podemos utilizar para nuestros robots.

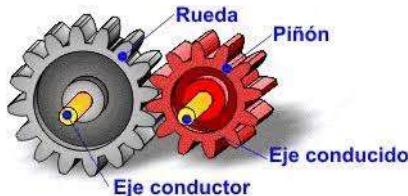
Controladores

Un robot está formado por los siguientes elementos:

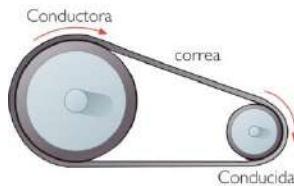
- Estructura mecánica
- Transmisiones

Son los elementos encargados de transmitir el movimiento desde los actuadores hasta las articulaciones. Algunos de las transmisiones que nos podemos encontrar en los robots son:

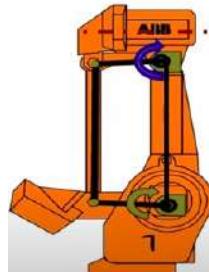
- Transmisión por engranaje



- Transmisión por polea-correa



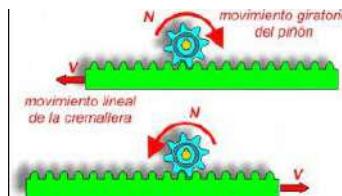
- Transmision por paralelogramo



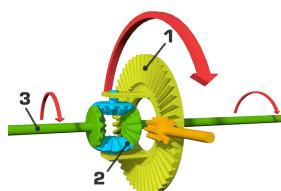
- Transmision por husillo de bolas



- Transmision por piñon-cremallera



- Transmision diferencial



- Entre muchos otros

- Actuadores

Los actuadores tienen como misión generar el movimiento de los elementos del robot según las órdenes dadas por la unidad de control. Se clasifican en tres grandes grupos, según la energía que utilizan:

- Neumáticos (convierten la energía del aire comprimido en trabajo mecánico)
- Hidráulicos (son mecanismos que constan de un cilindro dentro

- del cual se desplaza un émbolo o pistón, y transforman la presión de un líquido, mayormente aceite, en energía mecánica)
- Electricos (son accionados por medio de corrientes electricas)
- Sensores
- Elementos terminales (tambien llamados efectores finales, son los encargados de interaccionar directamente con el entorno del robot)
- Controladores

Para comenzar, hablemos sobre los controladores en ROS con los que contamos:

Paquetes controladores:

```
root@chuy:/catkin_ws# rospack list | grep controller
ackermann_steering_controller /opt/ros/melodic/share/ackermann_steering_controller
controller_interface /opt/ros/melodic/share/controller_interface
controller_manager /opt/ros/melodic/share/controller_manager
controller_manager_msgs /opt/ros/melodic/share/controller_manager_msgs
diff_drive_controller /opt/ros/melodic/share/diff_drive_controller
effort_controllers /opt/ros/melodic/share/effort_controllers
force_torque_sensor_controller /opt/ros/melodic/share/force_torque_sensor_controller
forward_command_controller /opt/ros/melodic/share/forward_command_controller
gripper_action_controller /opt/ros/melodic/share/gripper_action_controller
imu_sensor_controller /opt/ros/melodic/share imu_sensor_controller
joint_state_controller /opt/ros/melodic/share/joint_state_controller
joint_trajectory_controller /opt/ros/melodic/share/joint_trajectory_controller
position_controllers /opt/ros/melodic/share/position_controllers
velocity_controllers /opt/ros/melodic/share/velocity_controllers
root@chuy:/catkin_ws#
```

Tenemos:

- *ackermann_steering_controller*
Controlador para una base movil de direccion.
- *controller_interface*
Clase base de interfaz para controladores.
- *controller_manager*
El administrador del controlador.

Proporciona un bucle en tiempo real para controlar un mecanismo de robot, que esta representado por una instancia de la clase “hardware_interface::RobotHW”.

```
root@chuy:/catkin_ws# rospack list | grep hardware
hardware_interface /opt/ros/melodic/share/hardware_interface
root@chuy:/catkin_ws#
```

“hardware_interface” es la clase base de interfaz de hardware.

Este paquete proporciona la infraestructura para cargar, descargar, iniciar y detener los controladores.

Al cargar un controlador, el “controller_manager” usara el nombre del controlador como la raiz para todos los parametros especificos del controlador (los parametros especificos del controlador deberan estar en el espacio de nombres del nombre del controlador).

Podemos interactuar con el “controller_manager” desde:

- Línea de comandos

Se tienen comandos para cargar, descargar, iniciar, detener, cargar e iniciar (a la vez), y detener y descargar (a la vez) controladores, así como también obtener el estado de los controladores, entre otros mas.

- Un archivo de lanzamiento

Se pueden iniciar los controladores desde un archivo de lanzamiento.

Para esto necesitamos usar el script “spawner” para cargar, descargar, iniciar y detener automáticamente un controlador desde el archivo de lanzamiento. Cuando se elimine el archivo de lanzamiento se detendrá y se descargará el controlador automáticamente.

Ejemplo:

```
<launch>
  <node pkg="controller_manager"
    type="spawner"
    args="controller_name1 controller_name2" />
</launch>
```

Para interactuar con los controladores de otro nodo ROS, el administrador del controlador (“controller_manager”) proporciona algunas llamadas de servicio:

```
root@chuy:/catkin_ws# rossrv show controller_manager_msgs/
controller_manager_msgs/ListControllerTypes          controller_manager_msgs/ReloadControllerLibraries
controller_manager_msgs/ListControllers             controller_manager_msgs/SwitchController
controller_manager_msgs/LoadController              controller_manager_msgs/UnloadController
root@chuy:/catkin_ws# rossrv show controller_manager_msgs/[]
```

- */<node>/controller_manager/load_controller*
(controller_manager_msgs/LoadController)

```
root@chuy:/catkin_ws# rossrv show controller_manager_msgs/LoadController
string name
---
bool ok
```

A

- *controller_manager/unload_controller*
(controller_manager_msgs/UnloadController)

```
root@chuy:/catkin_ws# rossrv show controller_manager_msgs/UnloadController
string name
---
bool ok
```

A

- *controller_manager/switch_controller*
(controller_manager_msgs/SwitchController)

```
root@chuy:/catkin_ws# rossrv show controller_manager_msgs/SwitchController
int32 BEST EFFORT=1
int32 STRICT=2
string[] start_controllers
string[] stop_controllers
int32 strictness
bool start_asap
float64 timeout
---
bool ok
```

La solicitud contiene una lista de nombres de controladores para iniciar, una lista de nombres de controladores para detener y un valor entero que indica el rigor.

- *controller_manager/list_controllers*
(controller_manager_msgs/ListControllers)

```
root@chuy:/catkin_ws# rossrv show controller_manager_msgs/ListControllers
---
controller_manager_msgs/ControllerState[] controller
    string name
    string state
    string type
    controller_manager_msgs/HardwareInterfaceResources[] claimed_resources
        string hardware_interface
        string[] resources
```

Devuelve todos los controladores que estan cargados actualmente.

- *controller_manager/list_controller_types*
(controller_manager_msgs/ListControllerTypes)

```
root@chuy:/catkin_ws# rossrv show controller_manager_msgs/ListControllerTypes
---
string[] types
string[] base_classes
```

Devuelve todos los tipos de controlador que son conocidos por “controller_manager”.

- *controller_manager/reload_controller_libraries*
(*controller_manager_msgs/ReloadControllerLibraries*)

```
root@chuy:/catkin_ws# rossrv show controller_manager_msgs/ReloadControllerLibraries
bool force_kill
...
bool ok
```

Recarga todas las bibliotecas de controladores que estan disponibles como complementos. Esto es conveniente cuando se esta desarrollando un controlador y se desea probar el nuevo codigo del controlador sin reiniciar el robot cada vez. Solo funciona si no hay un controlador cargado.

- *controller_manager_msgs*

Mensajes y servicios para el administrador del controlador.

```
root@chuy:/catkin_ws# rosmsg show controller_manager_msgs/
controller_manager_msgs/ControllerState           controller_manager_msgs/ControllersStatistics
controller_manager_msgs/ControllerStatistics        controller_manager_msgs/HardwareInterfaceResources
root@chuy:/catkin_ws# rosmsg show controller_manager_msgs/
```

- *diff_drive_controller*

Controlador para ruedas de transmision diferencial.

La traccion diferencial en un robot es una configuracion muy comun debido a que permite al robot girar sobre su propio eje (por ejemplo, se puede deducir que un giro sobre el eje central de rotacion del robot es posible haciendo que una rueda avance hacia adelante y la otra hacia atras a la misma velocidad). La traccion diferencial utiliza dos ruedas que son controladas individualmente.

- *effort_controllers*

Controladores de esfuerzo.

- *force_torque_sensor_controller*

Controlador para publicar el estado de los sensores de fuerza-par.

- *forward_command_controller*

Controlador de articulacion unica.

- *gripper_action_controller*

Controlador para ejecutar una accion de comando de pinza.

- *imu_sensor_controller*
Controlador para publicar el estado de los sensores IMU (los sensores inerciales o unidad de medición inercial, IMU, son pequeños dispositivos capaces de medir la aceleración lineal y la velocidad angular).
- *joint_state_controller*
Controlador que lee un archivo URDF del servidor de parámetros (específicamente el parámetro “robot_description”) y publica el estado de las articulaciones en el tema “/joint_states”.
- *joint_trajectory_controller*
Controlador para ejecutar trayectorias de espacio articular en un grupo de articulaciones.
- *position_controllers*
Controladores de posición.
- *velocity_controllers*
Controladores de velocidad.

Todos estos controladores son con los que contamos para implementarlos sobre nuestro robot.

Recordemos que cuando se trata de robots, debemos asegurarnos de que su movimiento esté controlado, es decir, que las acciones no sean ni demasiado lentas ni demasiado rápidas y que sigan la trayectoria especificada.

Un tipo de controlador muy conocido es el **controlador PID**, el cual se puede utilizar para controlar el movimiento de nuestros robots.

Un controlador PID (controlador proporcional, integral y derivativo) es un mecanismo de control que a través de un lazo de retroalimentación permite regular la velocidad, la temperatura, la presión, entre otras variables de un proceso en general.

Este tipo de controlador lo estaremos viendo implementado en algunos controladores que usaremos más adelante.

También podemos escribir nuestros propios complementos de controlador

si lo deseamos (se necesita bastante conocimiento matematico).

Veamos como configurar controladores:

Los controladores generalmente se definen en archivos YAML (almacenados en una carpeta de configuracion, normalmente llamada "config" dentro de un paquete ROS).

Para el ejemplo definiremos los siguientes controladores:

- Un *JointStateController* para publicar los estados de las articulaciones (con una tasa de publicacion establecida en 50 Hz).
- Dos *JointPositionController* para mover dos articulaciones al recibir una posicion objetivo.

El archivo YAML contendra lo siguiente:

```
1 # "my_robot" sera nuestro espacio de nombres (todos los controladores estaran dentro
2 # de este espacio de nombres).
3 my_robot:
4     #Controlador para publicar los estados de las articulaciones
5     joint_state_controller:
6         type: "joint_state_controller/JointStateController"
7         publish_rate: 50
8
9     # Para los controladores de posicion podemos definir "pid gains" (ganancias
10    # del controlador PID).
11    # El objetivo de los ajustes de los parametros PID es lograr que el bucle de control
12    # corrija eficazmente y en el minimo tiempo los efectos de las perturbaciones.
13
14    # Controladores de posicion
15    joint1_position_controller:
16        type: "effort_controllers/JointPositionController"
17        joint: joint1 # Nombre de una articulacion
18        pid: {p: 100.0, i: 0.01, d: 10.0}
19    joint2_position_controller:
20        type: "effort_controllers/JointPositionController"
21        joint: joint2 # Nombre de una articulacion
22        pid: {p: 100.0, i: 0.01, d: 10.0}
```

Notar que hemos definido un espacio de nombres "my_robot", el cual sera muy importante recordar mas adelante.

Este archivo de configuracion se puede cargar en un archivo de lanzamiento, antes de la generacion de los controladores. Este archivo contendra lo siguiente:

```

1 <launch>
2   <!-- Cargamos las configuraciones desde el archivo YAML al servidor de parametros -->
3   <rosparam command="load" file="$(find <package>/config/<name_file>.yaml" />
4
5   <!-- Cargamos los controladores -->
6   <node name="controller_spawner" pkg="controller_manager" type="spawner" respawn="false" output="screen" ns="/my_robot">
7     args="joint_state_controller joint1 position_controller joint2 position_controller"/>
8
9   <!-- Convertimos los estados de las articulaciones a transformaciones TF para RViz -->
10  <node pkg="robot_state_publisher" type="robot_state_publisher" name="robot_state_publisher" respawn="false" output="screen">
11    <!--
12      El tema donde se estaran publicando los estados de las articulaciones es: "/my_robot/joint_states"
13
14      Reasignacion de temas ROS para nodos (remap)
15      La reasignacion nos permite "engañar" a un nodo ROS para que cuando crea que se esta suscribiendo o publicando en
16      "/some_topic" en realidad se esta suscribiendo o publicando en "/some_other_topic".
17
18      Atributos:
19      - from: "/some_topic"
20      - to: "/some_other_topic"
21    >>>
22    <remap from="/joint_states" to="/my_robot/joint_states"/>
23  </node>
24 </launch>

```

Si deseamos simular el robot en Gazebo con estos controladores, se debe agregar el complemento correspondiente al modelo URDF (agregando el complemento “gazebo_ros_control”, cargando la biblioteca compartida llamada “libgazebo_ros_control.so”, como ya se explico).

NOTA: Debemos tener en cuenta que los controladores deben generarse dentro del mismo espacio de nombres donde se han definido los controladores (en el archivo YAML), en este caso “my_robot”. Ademas el complemento de Gazebo debera estar dentro de este espacio de nombres, de la siguiente manera:

```

<!-- Gazebo plugin for ROS Control -->
<gazebo>
  <plugin name="gazebo_ros_control" filename="libgazebo_ros_control.so">
    <robotNamespace>/my_robot</robotNamespace>
  </plugin>
</gazebo>

</robot>

```

(Esto dentro de un archivo URDF)

En `<robotNamespace>` se define el espacio de nombres que se utilizara para la instancia del complemento.

Como podemos observar, solo necesitamos crear un archivo de configuracion (.yaml) y un archivo de lanzamiento (.launch) para nuestros controladores que interactuaran con Gazebo.

Todo esto es un ejemplo basico para entender como configurar controladores para un modelo de robot.

Paquete "ros_control"

"ros_control" (un nuevo estandar en ROS para interfaces de controlador) es un conjunto de paquetes que incluye interfaces de controlador, administradores de controlador transmisiones e interfaces de hardware.

La documentacion se encuentre en el siguiente sitio:
http://wiki.ros.org/ros_control

ROS proporciona un conjunto de paquetes que se pueden utilizar para controlar el movimiento de nuestros robots mediante controladores PID (los cuales ya mencionamos anteriormente).

El paquete "ros_control" se puede utilizar para nuestros robots ahorrandonos el tiempo de reescribir el codigo de los controladores.

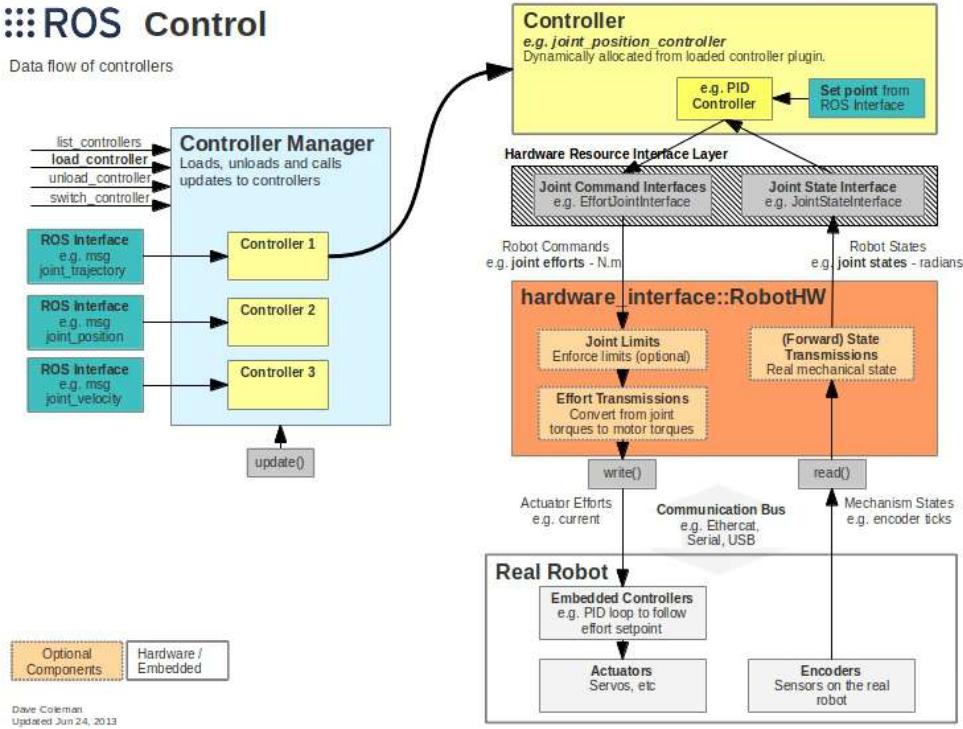
Veamos el marco "ros_control":

El marco "ros_control" proporciona la capacidad de implementar y administrar controladores de robot.

Consiste principalmente en un mecanismo de retroalimentacion, muy probablemente un bucle PID, que puede recibir un punto de ajuste y controlar la salida (por ejemplo, el esfuerzo) utilizando la retroalimentacion de los actuadores. Esta salida se pasa al robot a traves de la interfaz de hardware (las interfaces de hardware son utilizadas para enviar y recibir comandos al hardware).

ROS Control

Data flow of controllers



Podemos observar lo siguiente:

- Tenemos al administrador del controlador, el cual se encarga de cargar, descargar, entre otras cosas mas, los controladores.

Por medio de ROS podemos ver que es lo que hacen los controladores, por ejemplo, el controlador “`joint_state_controller`” publica el estado de las articulaciones desde la simulacion en el tema “`/joint_states`”, tema el cual podemos consultar desde ROS.

Tener en cuenta que podemos utilizar varios controladores, los cuales ya se mencionaron. Entre los controladores ROS principales tenemos:

- **`effort_controllers`**
Los comandos de esfuerzo se utilizan para controlar posiciones, velocidades o esfuerzos de las articulaciones.
- **`position_controllers`**
Los comandos de posicion se utilizan para controlar las posiciones de las articulaciones.
- **`velocity_controllers`**
Los comandos de velocidad se utilizan para controlar las velocidades de las articulaciones.

Para establecer una trayectoria completa, tenemos el siguiente controlador:

- *joint_trajectory_controller*

Ademas, tenemos un controlador que publica los estados de las articulaciones de un robot:

- *joint_state_controller*

- Por medio de la interfaz ROS podemos enviar el objetivo (o punto de ajuste) a los controladores, por ejemplo, enviando un mensaje que especifique las trayectorias que deben de seguir un grupo de articulaciones (puede ser solo una articulacion).

Por ejemplo, tenemos el mensaje “*trajectory_msgs/JointTrajectory.msg*”:

```
root@chuy:/catkin_ws# rosmsg show trajectory_msgs/JointTrajectory
std msgs/Header header
  uint32 seq
  time stamp
  string frame_id
string[] joint_names
trajectory msgs/JointTrajectoryPoint[] points
  float64[] positions
  float64[] velocities
  float64[] accelerations
  float64[] effort
duration time_from_start

root@chuy:/catkin_ws#
```

Aqui los controladores son los responsables de calcular el comando de salida requerido para lograr el objetivo establecido.

- El controlador como tal, como ya se menciono, es el que recibe el objetivo o punto de ajuste (dicho controlador puede ser un controlador PID) y se encarga de calcular el comando de salida requerido para lograr el objetivo establecido.

Dichos comandos de salida pueden ser esfuerzos articulares (en Newton metro, Nm), etc.

Los controladores deben hablar con los actuadores, y para esto necesitan de la interfaz de hardware.

- Tenemos una pequena capa que es la conexion entre los controladores y la interfaz de hardware.

Aqui tenemos una serie de interfaces, de las cuales destacan:

- *Joint Command Interface*

- *Effort Joint Interface*: para enviar el comando de esfuerzo.
- *Velocity Joint Interface*: para enviar el comando de velocidad.
- *Position Joint Interface*: para enviar el comando de posicion.
- *Joint State Interfaces*: para recuperar los estados de las articulaciones.
- *Actuator State Interfaces*
- *Actuator Command Interfaces*
 - *Effort Actuator Interface*
 - *Velocity Actuator Interface*
 - *Position Actuator Interface*
- *Pos Vel Joint Interface*
- *Pos Vel Acc Joint Interface*
- *Force-torque sensor Interface*
- *IMU sensor Interface*
- La interfaz de hardware se encuentra entre los controladores y el hardware real y ordena a los actuadores que se muevan y obtengan la retroalimentacion de los sensores articulares.

Las interfaces de hardware son utilizadas para enviar y recibir comandos al hardware.

Aqui se hacen cumplir los limites articulares y se realizan algunas conversiones (por ejemplo, convertir un torque articular a un torque de motor, ya que el comando que enviamos es para los actuadores).

Como hemos visto, los controladores no se comunican directamente con el hardware, sino que lo hacen a traves de la capa de abstraccion de hardware, lo que permite que los controladores sean independientes del hardware.

- Por ultimo, tendremos un medio de comunicacion por el cual el robot recibe todos los comandos necesarios para que pueda moverse.

Recordemos que un robot consta de actuadores, encoders, algun microcontrolador, entre otros elementos.

Asi es como funciona “ros_control”.

Recordemos que “hardware_interface” es la clase base de interfaces de hardware.

Con respecto a las interfaces, tenemos los siguientes paquetes relacionados con interfaces:

```
root@chuy:/catkin_ws# rospack list | grep interface
controller_interface /opt/ros/melodic/share/controller_interface
hardware_interface /opt/ros/melodic/share/hardware_interface
joint_limits_interface /opt/ros/melodic/share/joint_limits_interface
transmission_interface /opt/ros/melodic/share/transmission_interface
root@chuy:/catkin_ws#
```

Para controlar a un robot dado usando “ros_control”, se debe implementar una clase derivada de la clase:

- Si no se trata de simulacion (robot real):
hardware_interface::RobotHW
- Si se trata de simulacion:
hardware_interface::RobotHWSim

Estas clases deben admitir una o mas de las interfaces estandar ya mencionadas (como por ejemplo, “EffortJointInterface”).

Veamos un ejemplo de codigo el cual muestra la clase implementada para un robot con dos articulaciones controladas por posicion:

(para familiarizarnos un poco con el codigo)

Podemos tener articulaciones controladas por:

- *Esfuerzo*
- *Posicion*
- *Velocidad*

El codigo presentado en este
y mas ejemplos estaran
dados en lenguaje **C++**.

Vamos a suponer un robot con dos articulaciones A y B, donde ambas articulaciones estan controladas por posicion. Dicho robot se define con una clase derivada de “hardware_interface::RobotHW” que debe proporcionar:

- *PositionJointInterface*
- *JointStateInterface*

de modo que podamos reutilizar todos los controladores que ya estan escritos.

```

1 // Importamos las declaraciones de las clases (interfaces) que necesitaremos
2 #include <hardware_interface/joint_command_interface.h>
3 #include <hardware_interface/joint_state_interface.h>
4 #include <hardware_interface/robot_hw.h>
5
6 // Heredamos de la clase "hardware_interface::RobotHW"
7 class MyRobot : public hardware_interface::RobotHW{
8     public:
9         /*
10            Dentro del constructor inicializamos los recursos del robot (articulaciones, sensores, actuadores)
11            y las interfaces.
12        */
13    MyRobot(){
14        /*
15            Creamos un objeto de tipo "JointStateHandle" para cada articulación y los registramos con
16            el objeto de tipo "JointStateInterface".
17            "JointStateHandle" es un "handle" (manejador) que se usa para leer el estado de una sola
18            articulación. Actualmente los campos de posición, velocidad y esfuerzo son obligatorios.
19        */
20        hardware_interface::JointStateHandle state_handle_a("A", &pos[0], &vel[0], &eff[0]);
21        joint_state_interface.registerHandle(state_handle_a);
22        hardware_interface::JointStateHandle state_handle_b("B", &pos[1], &vel[1], &eff[1]);
23        joint_state_interface.registerHandle(state_handle_b);
24        /*
25            Registraremos el objeto de tipo "JointStateInterface" que contiene las articulaciones de solo
26            lectura.
27        */
28        registerInterface(&joint_state_interface);
29
30        /*
31            Creamos un objeto de tipo "JointHandle" (lectura y escritura) para cada articulación
32            controlable usando los identificadores de articulación de solo lectura dentro del objeto de
33            tipo "JointStateInterface" y los registramos en el objeto de tipo "PositionJointInterface".
34            "JointHandle" es un "handle" (manejador) que se utiliza para leer y controlar una sola
35            articulación.
36        */
37        hardware_interface::JointHandle pos_handle_a(joint_state_interface.getHandle("A"), &cmd[0]);
38        position_joint_interface.registerHandle(pos_handle_a);
39        hardware_interface::JointHandle pos_handle_b(joint_state_interface.getHandle("B"), &cmd[1]);
40        position_joint_interface.registerHandle(pos_handle_b);
41        /*
42            Registraremos el objeto de tipo "PositionJointInterface" que contiene las articulaciones de solo
43            lectura/escritura.
44        */
45        registerInterface(&position_joint_interface);
46    }
47    ~MyRobot(){}
48 private:
49     // Declaramos las interfaces que necesitaremos
50     hardware_interface::PositionJointInterface position_joint_interface;
51     hardware_interface::JointStateInterface joint_state_interface;
52     /*
53        Declaramos una matriz para almacenar los comandos del controlador que se envían a los recursos del
54        robot (articulaciones, actuadores).
55    */
56    double cmd[2];
57    /*
58        Declaramos matrices para almacenar el estado de los recursos del robot (articulaciones, sensores).
59    */
60    double pos[2], vel[2], eff[2];
61 };

```

Cuando se ejecuta el administrador del controlador (“controller_manager”), los controladores leeran las variables “pos”, “vel” y “eff” (son las variables en donde se guarda el estado de las articulaciones), y el controlador escribira el comando deseado en la variable “cmd” (variable donde se guarda el comando que se desea que ejecute el actuador).

Es obligatorio asegurarnos de que las variables “pos”, “vel” y “eff” siempre tengan el ultimo estado de las articulaciones disponible, y tambien asegurarnos de que todo lo que este escrito en la variable “cmd”

sea ejecutado por el robot. Lo cual se puede hacer haciendo uso de los metodos:

- *hardware_interface::RobotHW::read()*
- *hardware_interface::RobotHW::write()*

Estas funciones nos sirven para llenar las variables “pos”, “vel” y “eff” con los ultimos valores disponibles del robot y para ejecutar el comando disponible en “cmd” en el robot.

NOTA: Estas funciones se llamaran en el **bucle de control** antes y despues del calculo del comando de control (recordar que el controlador debe generar el comando deseado una vez conozca el estado de las articulaciones. Una vez generado dicho comando, ahora si se ejecuta en el robot).

Bucle de control:

Lectura	Actualizacion	Escritura
----------------	----------------------	------------------

Una vez creada nuestra clase de robot con dos articulaciones A y B controladas por posicion, la funcion “main()” de un nodo se puede implementar de la siguiente manera:

```

1 #include <ros/ros.h>
2 #include <controller_manager/controller_manager.h>
3 // Incluimos el archivo .h de la definición de nuestro robot.
4 // ...
5
6 int main(int argc, char** argv){
7     // Inicializamos un nodo ROS.
8     ros::init(argc,argv, "my_robot");
9
10    // Creamos una instancia de nuestro robot.
11    MyRobot::MyRobot robot;
12
13    /*
14     * Creamos una instancia del administrador del controlador y le pasamos por referencia
15     * la instancia del robot.
16     */
17    controller_manager::ControllerManager cm(&robot);
18
19    /*
20     * Configuramos un hilo separado que se utilizara para dar servicio a las
21     * devoluciones de llamada de ROS (se configura de forma asíncrona).
22     *   El numero de hilos es la cantidad de subprocesos que se utilizaran.
23
24     * Recordar que tambien existe otro metodo llamado "spin()", el cual entra en un
25     * bucle para procesar devoluciones de llamada.
26     */
27    ros::AsyncSpinner spinner(1); // Numero de hilos
28    spinner.start();
29
30    // Configuracion para el bucle de control
31    ros::Time prev_time=ros::Time::now();
32    ros::Rate rate(10.0); // Tasa de 10 Hz
33
34    /*
35     * ros::ok() devuelve verdadero mientras el nodo no este apagado.
36     */
37    while(ros::ok()){
38        /*
39         * Contabilidad basica para obtener la hora del sistema con el fin de calcular
40         * el periodo de control.
41         */
42        const ros::Time time=ros::Time::now();
43        // Aquí obtenemos el tiempo transcurrido hasta el momento.
44        const ros::Duration period=time-prev_time;
45
46        // Leemos los datos del hardware del robot.
47        robot.read();
48
49        // Actualizamos todos los controladores activos (con la información reciente).
50        cm.update(time, period);
51
52        // Escribimos comandos en el hardware del robot (articulaciones, actuadores).
53        robot.write();
54
55        // Todos estos pasos se seguirán repitiendo con la tasa especificada (10 Hz).
56        rate.sleep();
57    }
58    return 0;
59 }

```

Como podemos observar, todo sigue un bucle de control (leer, actualizar, escribir).

Cabe mencionar que el administrador del controlador (“controller_manager”) realiza un seguimiento de los recursos que utiliza cada uno de los controladores (sabe exactamente qué controlador ha solicitado qué recursos).

También es posible implementar interfaces de hardware personalizadas y definir recursos personalizados.

La clase “RobotHW” tiene una implementacion predeterminada simple para la administracion de recursos.

Este metodo simplemente evita que dos controladores que estan usando el mismo recurso se ejecuten al mismo tiempo.

Si este sencillo esquema de gestion de recursos se ajusta a nuestro robot, entonces no sera necesario hacer nada mas, ya que el administrador del controlador aplicara automaticamente este esquema. De lo contrario, si nuestro robot necesita un esquema diferente, es posible crear uno personalizado.

Por ultimo, a continuacion se muestra un ejemplo de un robot con interfaces estandar y especificas del robot:

```
1 // Importamos las declaraciones de las clases (interfaces) que necesitaremos
2 // ...
3 #include <hardware_interface/robot_hw.h>
4
5 /*
6 Para un robot es posible utilizar interfaces estandar (cuando es posible y compatible)
7 como tambien interfaces especificas del robot.
8
9     - Interfaces estandar nos referimos a las interfaces ya definidas y listas para usar.
10
11    - Interfaces especificas del robot nos referimos a las que nosotros mismos creamos
12      especificas de nuestro robot.
13
14 A continuacion se muestra un ejemplo de un robot con interfaces estandar y especificas
15 del robot.
16 */
17
18 // Heredamos de la clase "hardware_interface::RobotHW"
19 class MyRobot : public hardware_interface::RobotHW{
20     public:
21         /*
22             Dentro del constructor inicializamos los recursos del robot.
23         */
24     MyRobot(){
25         // Registrarmos interfaces de hardware estandar
26         registerInterface(...);
27
28         // Registrarmos algunas interfaces especificas del robot
29         registerInterface(&my_interface);
30     }
31     ~MyRobot(){}
32     private:
33         // Declaramos las interfaces que necesitaremos
34         // ...
35         MyCustomInterface my_interface;
36
37         // Algunas variables
38         // ...
39 };
```

Con respecto a la interfaz de hardware (lo cual ya se explico), opcionalmente la

interfaz de hardware se puede mejorar considerando los **limites de las articulaciones** y modelando la dinamica de **transmision** de las articulaciones:

- *Limites de articulaciones*

El paquete “joint_limits_interface” contiene:

- Estructuras de datos para representar limites de articulaciones
- Metodos para poblarlos a traves de archivos URDF o yaml
- Metodos para hacer cumplir estos limites

```
root@chuy:/catkin_ws# rospack list | grep joint_limits
joint_limits_interface /opt/ros/melodic/share/joint_limits_interface
root@chuy:/catkin_ws#
```

Recordemos que especificar limites articulares es muy importante, ya que en un robot real puede haber problemas cuando se superan ciertos limites articulares.

Ejemplos de especificacion de limites articulares:

- En formato URDF

```
<joint name="$foo_joint" type="revolute">
    <!-- other joint description elements -->

    <!-- Joint limits -->
    <limit lower="0.0"
          upper="1.0"
          effort="10.0"
          velocity="5.0" />

    <!-- Soft limits -->
    <safety_controller k_position="100"
                        k_velocity="10"
                        soft_lower_limit="0.1"
                        soft_upper_limit="0.9" />
</joint>
```

En la etiqueta `<limit>` se especifican los limites articulares y en la etiqueta `<safety_controller>` se especifican los limites de seguridad, por ejemplo, se puede especificar que un controlador no pueda comandar un esfuerzo superior a 30 N ni inferior a -30 N en la articulacion.

- En formato YAML

```

joint_limits:
  foo_joint:
    has_position_limits: true
    min_position: 0.0
    max_position: 1.0
    has_velocity_limits: true
    max_velocity: 2.0
    has_acceleration_limits: true
    max_acceleration: 5.0
    has_jerk_limits: true
    max_jerk: 100.0
    has_effort_limits: true
    max_effort: 5.0
  bar_joint:
    has_position_limits: false # Continuous joint
    has_velocity_limits: true
    max_velocity: 4.0

```

En este formato tiene sus ventajas, ya que ademas de poder especificar los limites de aceleracion y tirones, se pueden anular los valores de posicion, velocidad y esfuerzo contenidos en una descripcion URDF.

Este archivo YAML se carga en el servidor de parametros.

Las estructuras de datos de los limites articulares (que se definen en un archivo C++) se pueden poblar de diferentes maneras:

- Estableciendo los limites articulares manualmente
- Obteniendolos de un archivo URDF
- Obteniendolos del servidor de parametros (del archivo YAML que se cargo)
- *Transmisiones*

Las interfaces de transmision implementan transmisiones mecanicas.

Con respecto a las transmisiones en URDF, el elemento de transmision es una extension del URDF que se utiliza para describir la relacion entre un actuador y una articulacion.

Se pueden conectar multiples actuadores a multiples articulaciones a traves de una transmision compleja.

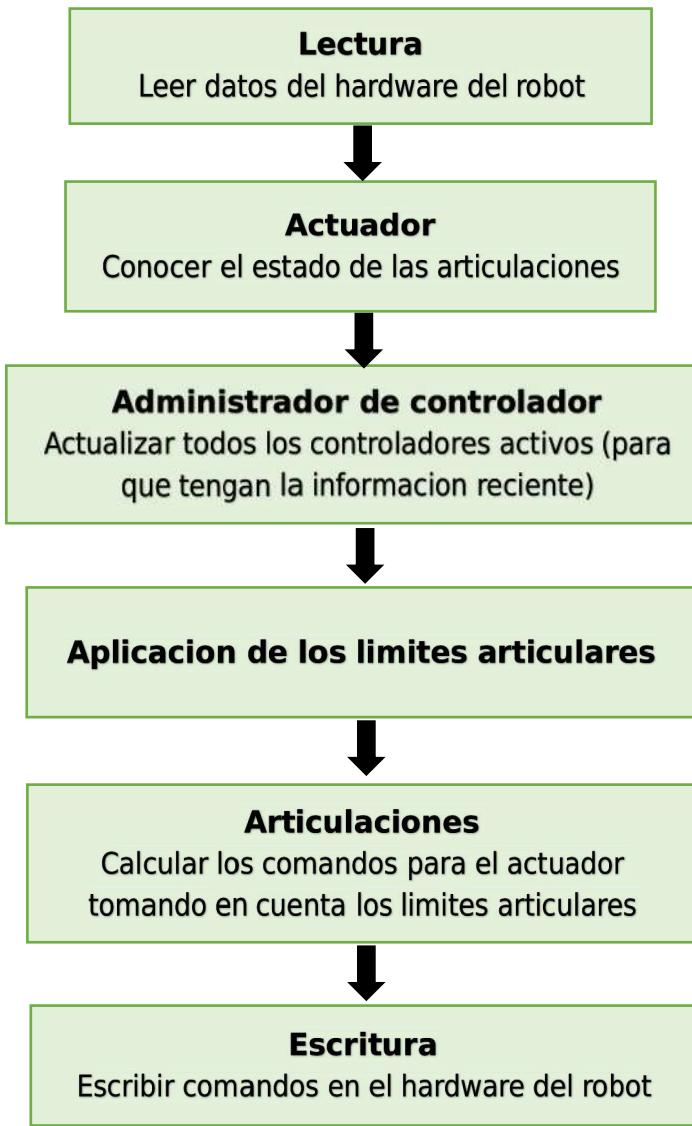
El elemento de transmision tiene los siguientes elementos:

- *<type> (una ocurrencia)*
El tipo de transmision.

- ***<joint> (una o mas ocurrencias)***
Una articulacion a la que esta conectada la transmision.
 - ***<hardwareInterface> (una o mas ocurrencias)***
Se especifica una interfaz de hardware.
- ***<actuator> (una o mas ocurrencias)***
Un actuador al que esta conectada la transmision.
 - ***<mechanicalReduction> (opcional)***
Se especifica una reduccion mecanica en la transmision de la articulacion/actuador.
 - ***<hardwareInterface> (opcional)***
Se especifica una interfaz de hardware (para ciertas versiones de ROS se especifica aqui la interfaz de hardware, pero el lugar correcto es especificarla en la etiqueta *<joint>*).

Al igual que teniamos un paquete llamado “hardware_interface”, tambien tenemos un paquete llamado “transmission_interface” el cual contiene estructuras de datos para representar transmisiones mecanicas y metodos para propagar las variables de posicion, velocidad y esfuerzo entre el actuador y los espacios articulares (este paquete almacena los datos sin procesar existentes, por ejemplo, la posicion actual del actuador, etc.). La interfaz de transmision (“transmission_interface”) no es utilizada por los propios controladores, sino que opera antes o despues de la actualizacion de los controladores.

Ahora el bucle de control completo al considerar los limites de las articulaciones y la existencia de transmisiones es el siguiente:



Recordemos que las funciones **read()** y **write()** se deben de llamar en el bucle de control antes y después del cálculo del comando de control.

Con este lazo de control extendido podemos hacer uso de transmisiones mecánicas y cuidar los límites de las articulaciones.

A continuación veremos como preparar Gazebo usando “ros_control” para la simulación de nuestro robot, ya que recordemos, quedó pendiente (ya que faltaba conocer otros temas).

Uso de “ros_control” en Gazebo usando modelo de robot en URDF

Para usar “ros_control” en Gazebo, el modelo URDF del robot debe incluir dos elementos adicionales:

- Transmisiones
- El complemento gazebo

Es necesario agregar un complemento Gazebo al URDF para analizar realmente las etiquetas de transmision y para cargar las interfaces de hardware y el administrador de controlador adecuados. Por ejemplo:

```
<gazebo>
  <plugin name="gazebo_ros_control" filename="libgazebo_ros_control.so">
    <robotNamespace>/</robotNamespace>
  </plugin>
</gazebo>
```

Recordemos que “gazebo_ros_control” es un paquete:

```
root@chuy:/catkin_ws# rospack list | grep gazebo_ros_control
gazebo_ros_control /opt/ros/melodic/share/gazebo_ros_control
root@chuy:/catkin_ws#
```

Elementos secundarios que podemos agregar:

- *<robotNamespace>*

Aqui se define el espacio de nombres ROS que se utilizara para la instancia del complemento.

Este espacio de nombres debe ser el mismo que el espacio de nombres utilizado para definir los controladores (en el archivo YAML) y para generar los controladores (en el archivo de lanzamiento).

- *<controlPeriod>*

Aqui se define el periodo de la actualizacion del controlador (en segundos), la cual por defecto es el periodo de Gazebo.

- *<robotParam>*

Aqui se define la ubicacion de “robot_description” (el URDF) en el servidor de parametros, la cual por defecto es “/robot_description”.

- *<robotSymType>*

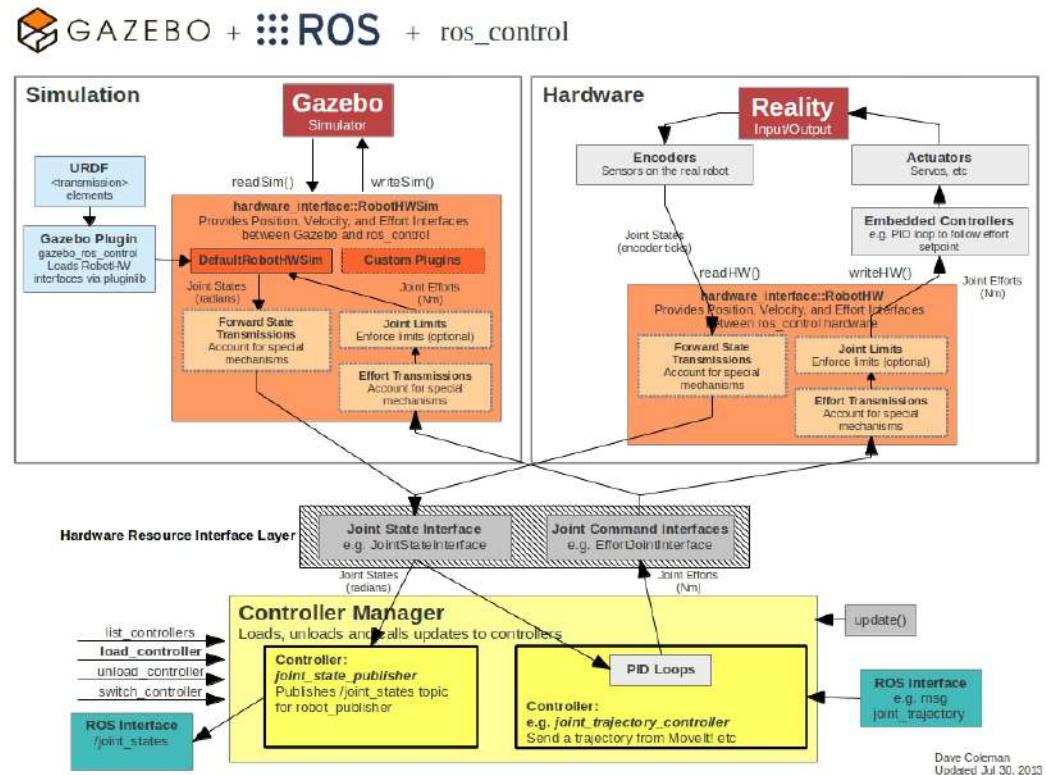
Aqui se especifica el nombre de una interfaz de simulacion de robot personalizada que se utilizara, la cual por defecto es “DefaultRobotHWSim”.

Una vez agregado el complemento Gazebo a nuestro URDF, este se encarga de cargar las interfaces de hardware y el administrador del

controlador (el cual se encarga de cargar, descargar, entre otras tareas, controladores).

Hay que dejar en claro que la simulacion de los controladores de un robot en Gazebo se puede lograr usando “ros_control” y un simple adaptador de complemento de Gazebo (el complemento “gazebo_ros_control”).

Veamos un diagrama mas general de la union de Gazebo, ROS y ros_control:



Como ya habiamos mencionado, las interfaces de hardware estan modeladas con clases que heredan de las clases:

- **hardware_interface::RobotHW**

Para cuando trabajamos con un robot real.

Proporciona interfaces de posicion, velocidad y esfuerzo entre “ros_control” y el hardware.

- **hardware_interface::RobotHWSim**

Para cuando trabajamos en una simulacion.

Proporciona interfaces de posicion, velocidad y esfuerzo entre Gazebo y “ros_control”.

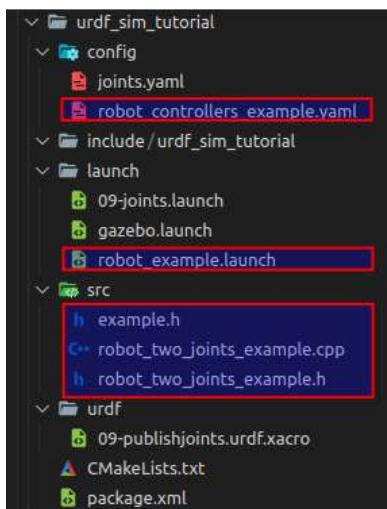
¿Cuales son las diferencias?

En la simulacion se necesita de un archivo URDF bien definido, con sus enlaces, articulaciones, transmisiones, etc., se necesita de un complemento Gazebo que se agrega al archivo URDF y todos los comandos de lectura y escritura estan realizadas al robot que se encuentra simulado en Gazebo.

En la vida real no necesitamos de un archivo URDF pero si de un robot real, del cual las lecturas se obtienen de los encoders del robot y las escrituras se mandan al microcontrolador del robot el cual se encarga de mandar comandos a los actuadores para que el robot se mueva. Aqui debe haber un bus de comunicacion entre ROS y el robot para que puedan comunicarse entre si.

Continuando con el uso de un URDF en Gazebo, retomaremos nuestro modelo de robot R2-D2 que habiamos creado anteriormente.

La estructura de carpetas por el momento es la siguiente:



Lo que esta encerrado en los cuadrados rojos son archivos que se crearon a modo de ejemplo para los temas anteriores, por lo que no son de

importancia y no se usaran.

Recapitulando un poco, habiamos hecho lo siguiente:

- Probamos que el modelo que ya creamos funcione correctamente en Gazebo.
- Agregamos el complemento Gazebo en el URDF (“gazebo_ros_control”).

Ahora lo que haremos sera agregar y definir controladores y transmisiones a nuestro robot:

- Transmisiones

Para cada articulacion no fija necesitamos especificar una transmision, la cual le dice a Gazebo que hacer con la articulacion.

Comencemos con la articulacion de la cabeza. Para ello lo que haremos sera duplicar el archivo “09-publishjoints.urdf.xacro” del paquete “urdf_sim_tutorial” que se encuentra en la carpeta “urdf” y le pondremos como nombre “10-firsttransmission.urdf.xacro”, en el cual agregaremos lo siguiente:

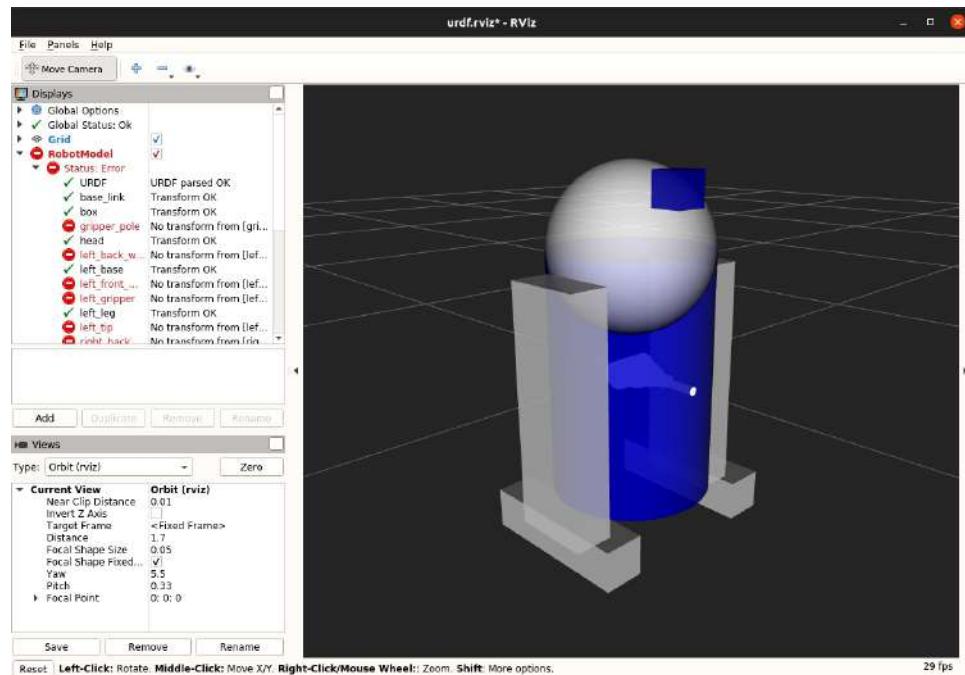
Recordemos que un elemento de **transmision** es una extension del URDF que se utiliza para describir la relacion entre un actuador y una articulacion.

```
209 <joint name="head_swivel" type="continuous">
210   <parent link="base_link"/>
211   <child link="head"/>
212   <axis xyz="0 0 1"/>
213   <origin xyz="0 0 ${bodylen/2}"/>
214 </joint>
215
216 <!--
217   Aqui conectamos la articulacion "head_swivel" a un actuador (motor)
218 -->
219 <transmission name="head_swivel_trans">
220   <!-- Especificamos el tipo de transmision -->
221   <type>transmission_interface/SimpleTransmission</type>
222   <!-- Especificamos el actuador al que esta conectada la transmision -->
223   <actuator name="#head_swivel_motor">
224     <!--
225       Especificamos una reduccion mecanica en la articulacion.
226
227       Esto es una forma de reducir o aumentar la rotacion de una articulacion a partir de la
228       rotacion original de algun motor.
229     -->
230     <mechanicalReduction>1</mechanicalReduction>
231   </actuator>
232   <!-- Especificamos la articulacion a la que estara conectada la transmision -->
233   <joint name="head_swivel">
234     <!--
235       Especificamos una interfaz de hardware para la articulacion.
236
237       Recordemos que estas interfaces, junto con uno de los controladores ROS, son utilizadas
238       para enviar y recibir comandos al hardware.
239       NOTA: Cada controlador tiene su respectiva interfaz.
240     -->
241     <hardwareInterface>hardware_interface/PositionJointInterface</hardwareInterface>
242   </joint>
243 </transmission>
```

Lo que hemos agregado es la etiqueta <transmission>.

Ahora podemos ejecutar este URDF con nuestra configuración de lanzamiento anterior (la que se encuentra en el archivo "09-joints.launch" en la carpeta "launch" del paquete "urdf_sim_tutorial"):

```
roscd urdf_sim_tutorial  
roslaunch urdf_sim_tutorial 09-joints.launch  
model:=urdf/10-firsttransmission.urdf.xacro &
```



```

root@chuy:/catkin_ws/src/urdf_sim_tutorial# rostopic info /joint_states
Type: sensor_msgs/JointState

Publishers:
* /gazebo (http://chuy:44725/)

Subscribers:
* /robot_state_publisher (http://chuy:46721/)

root@chuy:/catkin_ws/src/urdf_sim_tutorial# rostopic echo /joint_states -n 1
header:
  seq: 10318
  stamp:
    secs: 206
    nsecs: 653000000
  frame_id: ''
name: [head_swivel]
position: [-1.7632541293721715e-07]
velocity: [-2.474900105774736e-06]
effort: [0.0]
---
root@chuy:/catkin_ws/src/urdf_sim_tutorial# 
```

Ahora la cabeza se muestra correctamente en RViz (anteriormente se mostraba mal) porque la articulación de la cabeza se enumera en los mensajes del tema “/joint_states”.

Recordemos que anteriormente en RViz aparecía nuestro robot con la cabeza en medio del cuerpo, lo cual se debía a que aun no especificábamos una transmisión.

En RViz, como podemos observar, en el apartado “RobotModel” se muestra un estado de error, en los que tenemos errores que dicen:

“Sin transformacion de [<child_link>] a [<parent_link>]”

Esto por falta de transmisiones, como ya lo mencionamos.

“RobotModel” muestra una representación visual de un robot en la pose correcta (según lo definido por las transformaciones TF actuales).

Recordemos que en un archivo de lanzamiento, al trabajar con “ros_control”, declaramos lo siguiente:

- Cargamos las configuraciones de los controladores de

- articulaciones desde un archivo YAML al servidor de parametros.
- Cargamos los controladores (usando el script “spawner” del paquete “controller_manager”).
 - Convertimos los estados de las articulaciones a transformaciones TF para RViz.
Esto se logra iniciando el nodo “robot_state_publisher” que simplemente escucha los mensajes “/joint_states” (donde publica “joint_state_controller”) y luego publica las transformaciones en el tema “/tf”. Esto nos permite ver a nuestro robot simulado en RViz.

Podemos seguir agregando transmisiones para todas las articulaciones no fijas para que todas las articulaciones se publiquen correctamente.

Ahora agregaremos transmisiones para las articulaciones de agarre (la pinza) de manera similar. Para ello lo que haremos sera duplicar el archivo “10-firsttransmission.urdf.xacro” del paquete “urdf_sim_tutorial” que se encuentra en la carpeta “urdf” y le pondremos como nombre “12-gripper.urdf.xacro”, en el cual agregaremos lo siguiente:

```

146  <!-- Transmision para la articulacion del brazo de la pinza -->
147  <transmission name="gripper_extension_trans" type="SimpleTransmission">
148  | <type>transmission_interface/SimpleTransmission</type>
149  | <actuator name="gripper_extension_motor">
150  | | <mechanicalReduction>1</mechanicalReduction>
151  | </actuator>
152  | <joint name="gripper_extension">
153  | | <hardwareInterface>hardware_interface/PositionJointInterface</hardwareInterface>
154  | </joint>
155  </transmission>

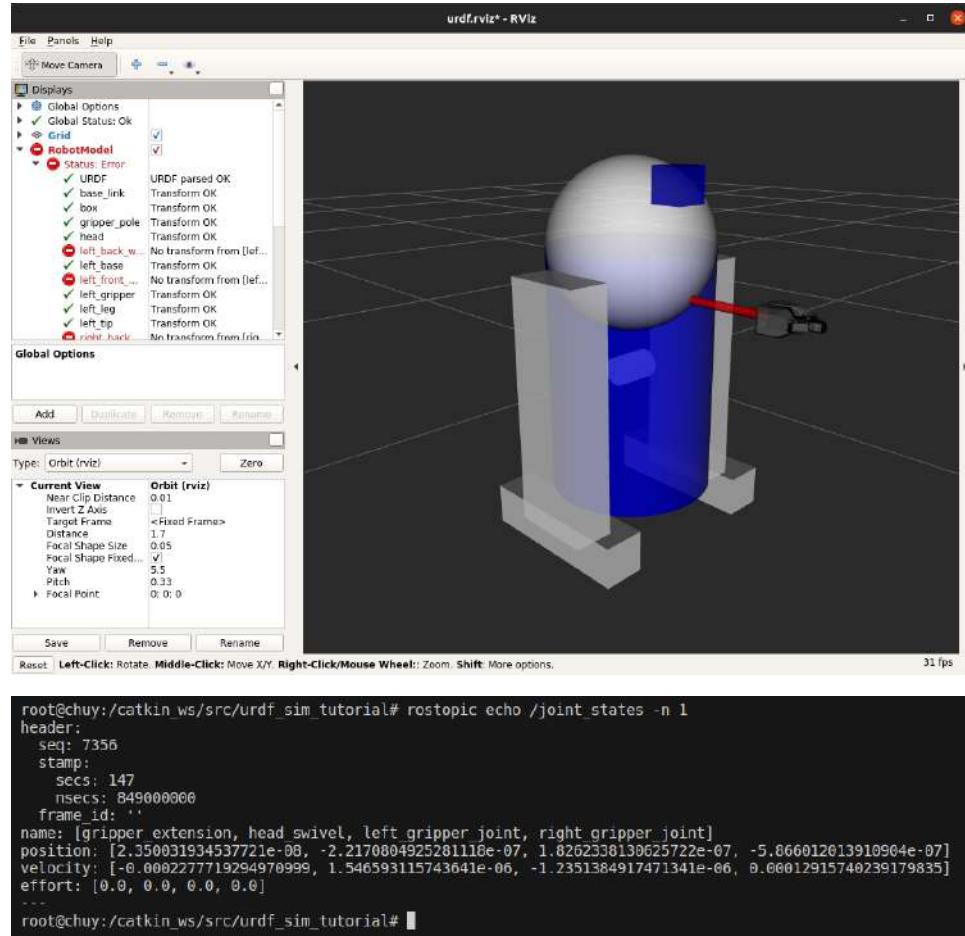
181  <!-- Transmision para la articulacion de agarre (la cual seran dos transmisiones) -->
182  <transmission name="${prefix}_gripper_trans" type="SimpleTransmission">
183  | <type>transmission_interface/SimpleTransmission</type>
184  | <actuator name="${prefix}_gripper_motor">
185  | | <mechanicalReduction>1</mechanicalReduction>
186  | </actuator>
187  | <joint name="${prefix}_gripper_joint">
188  | | <hardwareInterface>hardware_interface/PositionJointInterface</hardwareInterface>
189  | </joint>
190  </transmission>
```

Esta ultima transmision esta definida dentro de una macro Xacro.

Ahora podemos ejecutar este URDF con nuestra configuracion de lanzamiento anterior (la que se encuentra en el archivo “09-joints.launch” en la carpeta “launch” del paquete

“urdf_sim_tutorial”):

```
roscd urdf_sim_tutorial  
roslaunch urdf_sim_tutorial 09-joints.launch  
model:=urdf/12-gripper.urdf.xacro &
```



Ahora la pinza y el brazo robotico se muestran correctamente en RViz (anteriormente se mostraban mal) porque las articulaciones de la pinza y la articulacion del brazo robotico se enumeran en los mensajes del tema “/joint_states”.

Ahora, para conducir al robot, debemos especificar otra transmision para cada una de las ruedas. Para ello lo que haremos sera duplicar el archivo “12-gripper.urdf.xacro” del paquete “urdf_sim_tutorial” que se encuentra en la carpeta “urdf” y le pondremos como nombre “13-diffdrive.urdf.xacro”, en el cual agregaremos lo siguiente:

```

64      <joint name="${prefix}_${suffix}_wheel_joint" type="continuous">
65        <axis xyz="0 1 0" rpy="0 0 0" />
66        <parent link="${prefix}_base"/>
67        <child link="${prefix}_${suffix}_wheel"/>
68        <origin xyz="${baseLen*reflect/3} 0 -${wheelDiam/2+.05}" rpy="0 0 0"/>
69      </joint>
70
71      <!--
72        Especificamos al simulador (Gazebo) informacion sobre algunas propiedades fisicas
73        adicionales.
74        Dado que las ruedas van a tocar el suelo y, por tanto, interactuaran fisicamente con el,
75        se tienen que especificar informacion adicional sobre el material de las ruedas.
76      -->
77      <gazebo reference="${prefix}_${suffix}_wheel">
78        <mu1 value="200.0"/>
79        <mu2 value="100.0"/>
80        <kp value="10000000.0" />
81        <kd value="1.0" />
82        <material>Gazebo/Grey</material>
83      </gazebo>
84
85      <!-- Transmision que conecta la articulacion de la rueda a un actuador (motor) -->
86      <transmission name="${prefix}_${suffix}_wheel_trans">
87        <type>transmission_interface/SimpleTransmission</type>
88        <actuator name="${prefix}_${suffix}_wheel_motor">
89          <mechanicalReduction>1</mechanicalReduction>
90        </actuator>
91        <joint name="${prefix}_${suffix}_wheel_joint">
92          <hardwareInterface>hardware_interface/VelocityJointInterface</hardwareInterface>
93        </joint>
94      </transmission>
95
96    </xacro:macro>

```

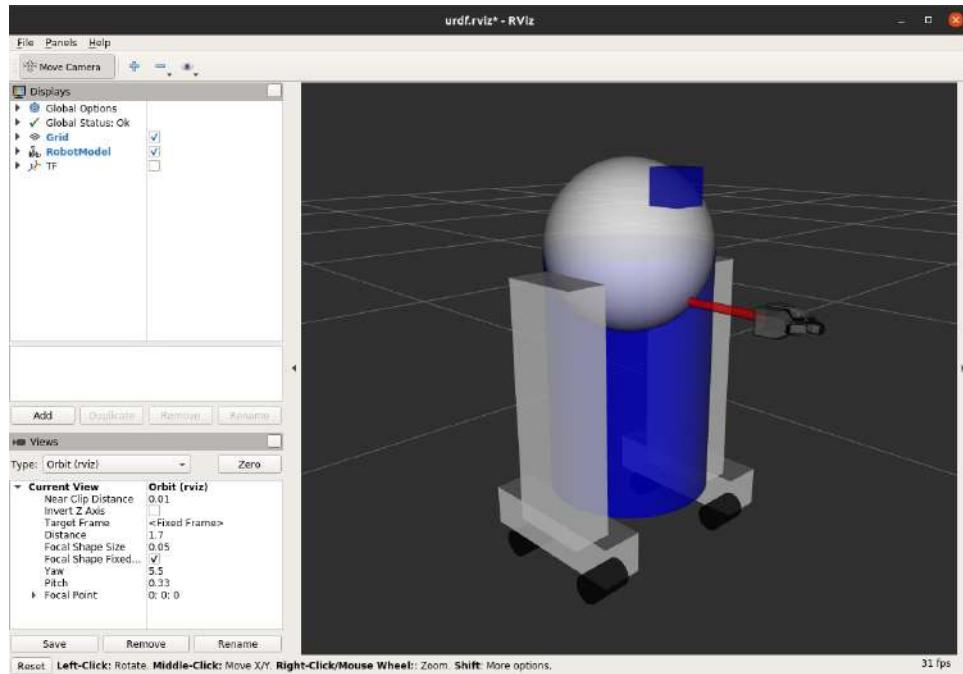
Esta transmision esta definida dentro de una macro Xacro.

Ahora podemos ejecutar este URDF con nuestra configuracion de lanzamiento anterior (la que se encuentra en el archivo “09-joints.launch” en la carpeta “launch” del paquete “urdf_sim_tutorial”):

```

roscd urdf_sim_tutorial
roslaunch urdf_sim_tutorial 09-joints.launch
model:=urdf/13-diffdrive.urdf.xacro &

```



```
root@chuy:~/catkin_ws/src/urdf_sim_tutorial# rostopic echo /joint_states -n 1
header: {seq: 2328, stamp: {secs: 140, nsecs: 65000000}, frame_id: "base_link"}
name: [right_gripper_extension, head_swivel, left_hack_wheel_joint, left_front_wheel_joint, left_grripper_joint,
right_beck_wheel_joint, right_front_wheel_joint, right_grripper_joint]
position: [-4.7113344647232e-09, 4.833124109e-09, -8.89932062373772e-09, -4.9166945317842e-09, -4.5553888691221e-09, 8.000159770648037e-09, 8.000240532210711e-09, 1.47914695497709e-08]
velocity: [0.0, 0.0, 0.0, 0.0, 0.0, 0.0]
effort: [0.0, 0.0, 0.0, 0.0, 0.0, 0.0]
root@chuy:~/catkin_ws/src/urdf_sim_tutorial[]
```

Ahora las ruedas se muestran correctamente en RViz (anteriormente se mostraban mal) porque las articulaciones de las ruedas se enumeran en los mensajes del tema “/joint_states”.

- Controladores

Definiremos un controlador para la articulacion de la cabeza del robot:

En un archivo YAML definimos el siguiente controlador, el cual tendra por nombre “head.yaml” y estara ubicado en la carpeta “config” del paquete “urdf_sim_tutorial”:

```
1  # Especificamos el uso de un "JointPositionController" del paquete "position_controllers"
2  # para controlar la articulacion de la cabeza
3  type: "position_controllers/JointPositionController"
4  joint: head_swivel
```

NOTA: La interfaz de hardware en el URDF para esta articulacion debe coincidir con el tipo de controlador (recordemos que el URDF la interfaz de hardware que se eligio es “hardware_interface/PositionJointInterface”).

Ahora podemos lanzar esto creando un nuevo archivo de

lanzamiento con el nombre “10-head.launch” ubicado en la carpeta “launch”, el cual sera una copia del archivo “09-joints.launch”. Dicho archivo nuevo le agregaremos lo siguiente:

```

1 <launch>
2   <!-- Se cargara por defecto el modelo de robot especifico para este archivo de lanzamiento -->
3   <arg name="model" default="$(find urdf_sim_tutorial)/urdf/10-firsttransmission.urdf.xacro"/>
4   <!-- Declaramos un argumento para indicar el archivo de configuracion de pantalla (.rviz) para cargar en RViz -->
5   <arg name="rvizconfig" default="$(find urdf_tutorial)/rviz/urdf.rviz" />
6
7   <!--
8     Incluimos el archivo de lanzamiento que inicia un mundo vacio en Gazebo y genera el modelo
9     de robot en Gazebo (el cual se le pasa como argumento), iniciando ademas el nodo
10    "robot_state_publisher" (el que publica transformaciones entre los diferentes marcos de coordenadas).
11   -->
12   <include file="$(find urdf_sim_tutorial)/launch/gazebo.launch">
13     <arg name="model" value="$(arg model)" />
14   </include>
15
16   <!-- Iniciamos el nodo "rviz" y le indicamos un archivo de configuracion de pantalla (.rviz) para cargar en RViz -->
17   <node name="rviz" pkg="rviz" type="rviz" args="-d $(arg rvizconfig)" />
18
19   <!--
20     Cargamos:
21       - Un archivo de parametros a un espacio de nombres llamado "r2d2_joint_state_controller".
22       - Un archivo de parametros a un espacio de nombres llamado "r2d2_head_controller".
23   -->
24   <rosparam command="load"
25     file="$(find urdf_sim_tutorial)/config/joints.yaml"
26     ns="r2d2_joint_state_controller" />
27   <rosparam command="load"
28     file="$(find urdf_sim_tutorial)/config/head.yaml"
29     ns="r2d2_head_controller" />
30
31   <!--
32     Iniciamos un nodo que ejecuta el script "spawner" del paquete "controller_manager" pasandole como
33     argumentos los nombres de los espacios de nombres en donde se encuentran los parametros que
34     especifican los tipos de controlador a usar.
35
36     "r2d2_joint_state_controller" sera el nombre de un controlador,
37     "r2d2_head_controller" sera el nombre de otro controlador.
38   -->
39   <node name="r2d2_controller_spawner" pkg="controller_manager" type="spawner"
40     args="r2d2_joint_state_controller
41           r2d2_head_controller
42           --shutdown-timeout 3"/>
43 </launch>
```

Como podemos observar, ahora cargamos los dos archivos de configuracion en donde definimos los controladores:

- joints.yaml
- head.yaml

Ahora ejecutamos el siguiente comando para lanzar el archivo de lanzamiento que acabamos de crear:

roslaunch urdf_sim_tutorial 10-head.launch &

```

root@chuy:/catkin_ws# rosparam list
/gazebo/auto_disable_bodies
/gazebo/cfm
/gazebo/contact_max_correcting_vel
/gazebo/contact_surface_layer
/gazebo/enable_ros_network
/gazebo/erp
/gazebo/gravity_x
/gazebo/gravity_y
/gazebo/gravity_z
/gazebo/max_contacts
/gazebo/max_update_rate
/gazebo/sor_pgs_iters
/gazebo/sor_pgs_precon_iters
/gazebo/sor_pgs_ms_error_tol
/gazebo/sor_pgs_w
/gazebo/time_step
/r2d2_head_controller/joint
/r2d2_head_controller/type
/r2d2_joint_state_controller/publish_rate
/r2d2_joint_state_controller/type
/robot_description
/robot_state_publisher/publish_frequency
/rosdistro
/roslaunch_uris/host_chuy_ 34975
/rosversion
/run_id
/use_sim_time
root@chuy:/catkin_ws# rosnode list
/gazebo
/gazebo_gui
/r2d2_controller_spawner
/robot_state_publisher
/rosout
/rviz

```

Los parametros estan cargados correctamente en el servidor de parametros.

```

root@chuy:/catkin_ws# rostopic list
/clock
/gazebo/link_states
/gazebo/model_states
/gazebo/parameter_descriptions
/gazebo/parameter_updates
/gazebo/set_link_state
/gazebo/set_model_state
/joint_states
/r2d2_head_controller/command ←
/rosout
/rosout_agg
/tf
/tf_static
root@chuy:/catkin_ws# rostopic info /r2d2_head_controller/command
Type: std_msgs/Float64

Publishers: None

Subscribers:
* /gazebo (http://chuy:43771/)

root@chuy:/catkin_ws# rosmsg show std_msgs/Float64
float64 data

root@chuy:/catkin_ws# []

```

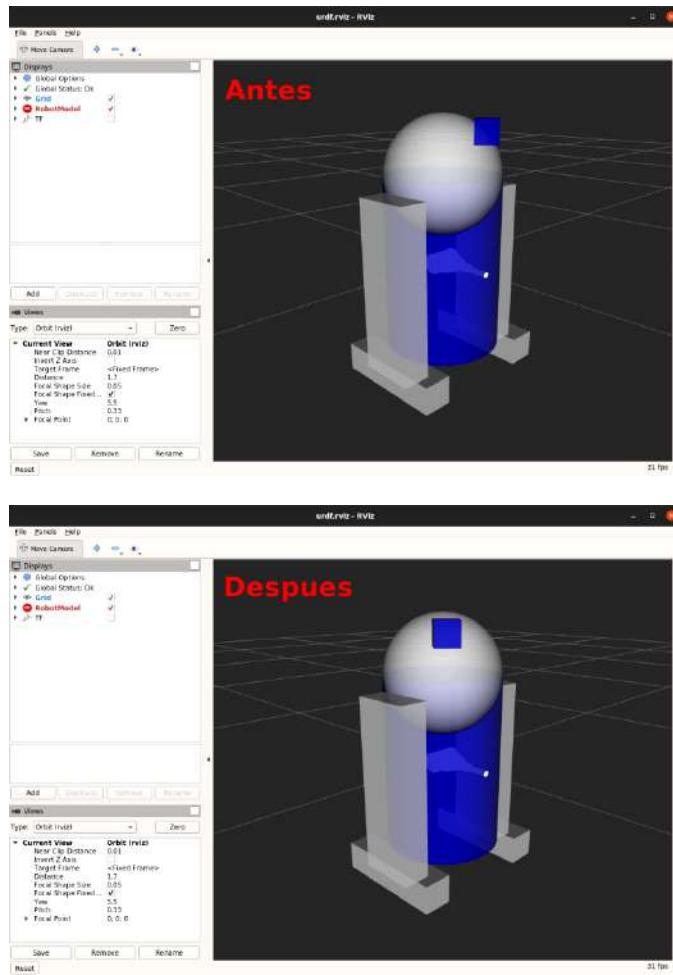
Como podemos observar tenemos un tema nuevo, en el cual esta suscrito Gazebo. En el podemos controlar la posicion de la cabeza publicando un valor en el tema desde ROS.

Al publicar un valor en dicho tema tendremos lo siguiente:

```

rostopic pub /r2d2_head_controller/command
std_msgs/Float64 "data: -0.707"

```



Cuando se publica este comando, la posicion cambia inmediatamente al valor especificado. Esto se debe a que no hemos especificado ningun limite en nuestro URDF.

Ahora lo que haremos sera agregar limites a la articulacion de la cabeza (ya que como mencionamos, no hemos agregado dichos limites). Lo que haremos sera duplicar el archivo "10-firsttransmission.urdf.xacro" que se encuentra en la carpeta "urdf" del paquete "urdf_sim_tutorial" y ponerle por nombre "11-limittransmission.urdf.xacro", al cual le agregaremos lo siguiente:

```

209 <joint name="head_swivel" type="continuous">
210   <parent link="base_link"/>
211   <child link="head"/>
212   <axis xyz="0 0 1"/>
213   <origin xyz="0 0 ${bodylen/2}"/>
214   <limit effort="30" velocity="1.0"/> ←
215 </joint>

```

Ahora ejecutamos el siguiente comando:

```

roscd urdf_sim_tutorial
roslaunch urdf_sim_tutorial 10-head.launch
model:=urdf/11-limittransmission.urdf.xacro &

```

Ahora al publicar, como lo hicimos anteriormente, tomara su tiempo para cambiar de posicion la cabeza.

Ahora definiremos un controlador para las articulaciones de agarre y del brazo robotico:

Ahora en lugar de controlar individualmente cada articulacion de la pinza con su propio tema ROS, es posible que deseemos agruparlos. Para esto, solo necesitamos especificar un controlador diferente.

En un archivo YAML definimos el siguiente controlador, el cual tendra por nombre “gripper.yaml” y estara ubicado en la carpeta “config” del paquete “urdf_sim_tutorial”:

```

1 # Especificamos el uso de un "JointGroupPositionController" del paquete "position_controllers"
2 # para controlar un grupo de articulaciones a la vez
3 type: "position_controllers/JointGroupPositionController"
4 joints:
5   - gripper_extension
6   - left_gripper_joint
7   - right_gripper_joint

```

NOTA: La interfaz de hardware en el URDF para este grupo de articulaciones debe especificarse como “hardware_interface/PositionJointInterface”.

Ahora podemos lanzar esto creando un nuevo archivo de lanzamiento con el nombre “12-gripper.launch” ubicado en la carpeta “launch”, el cual sera una copia del archivo

“10-head.launch”. Dicho archivo nuevo le agregaremos lo siguiente:

```

1 <launch>
2   <!-- Se cargara por defecto el modelo de robot especifico para este archivo de lanzamiento -->
3   <arg name="model" default="$(find urdf_sim_tutorial)/urdf/12-gripper.urdf.xacro"/>
4   <!-- Declaramos un argumento para indicar el archivo de configuracion de pantalla (.rviz) para cargar en RViz -->
5   <arg name="rvizconfig" default="$(find urdf_sim_tutorial)/rviz/urdf.rviz" />
6
7   <!--
8     Incluimos el archivo de lanzamiento que inicia un mundo vacio en Gazebo y genera el modelo
9     de robot en Gazebo (el cual se le pasa como argumento), iniciando ademas el nodo
10    "robot_state_publisher" (el que publica transformaciones entre los diferentes marcos de coordenadas).
11   -->
12   <include file="$(find urdf_sim_tutorial)/launch/gazebo.launch">
13     <arg name="model" value="$(arg model)" />
14   </include>
15
16   <!-- Iniciamos el nodo "rviz" y le indicamos un archivo de configuracion de pantalla (.rviz) para cargar en RViz -->
17   <node name="rviz" pkg="rviz" type="rviz" args="-d $(arg rvizconfig)" />
18
19   <!--
20     Cargamos:
21       - Un archivo de parametros a un espacio de nombres llamado "r2d2_joint_state_controller".
22       - Un archivo de parametros a un espacio de nombres llamado "r2d2_head_controller".
23   -->
24   <rosparam command="load"
25     file="$(find urdf_sim_tutorial)/config/joints.yaml"
26     ns="r2d2_joint_state_controller" />
27   <rosparam command="load"
28     file="$(find urdf_sim_tutorial)/config/head.yaml"
29     ns="r2d2_head_controller" />
30   <rosparam command="load"
31     file="$(find urdf_sim_tutorial)/config/gripper.yaml"
32     ns="r2d2_gripper_controller" />
33
34   <!--
35     Iniciamos un nodo que ejecuta el script "spawner" del paquete "controller_manager" pasandole como
36     argumentos los nombres de los espacios de nombres en donde se encuentran los parametros que
37     especifican los tipos de controlador a usar.
38
39     "r2d2_joint_state_controller" sera el nombre de un controlador.
40     "r2d2_head_controller" sera el nombre de otro controlador.
41   -->
42   <node name="r2d2_controller_spawner" pkg="controller_manager" type="spawner"
43     args="r2d2_joint_state_controller
44           r2d2_head_controller
45           r2d2_gripper_controller
46           --shutdown-timeout 3"/>
47 </launch>
```

Como podemos observar, ahora cargamos los tres archivos de configuracion en donde definimos los controladores:

- joints.yaml
- head.yaml
- gripper.yaml

Ahora ejecutamos el siguiente comando para lanzar el archivo de lanzamiento que acabamos de crear:

roslaunch urdf_sim_tutorial 12-gripper.launch &

```

root@chuy:/catkin_ws/src/urdf_sim_tutorial# rostopic list
/clock
/gazebo/link_states
/gazebo/model_states
/gazebo/parameter_descriptions
/gazebo/parameter_updates
/gazebo/set_link_state
/gazebo/set_model_state
/joint_states
/r2d2_gripper_controller/command ←
/r2d2_head_controller/command
/rosout
/rosout_agg
/tf
/tf_static
root@chuy:/catkin_ws/src/urdf_sim_tutorial# rostopic info /r2d2_gripper_controller/command
Type: std_msgs/Float64MultiArray

Publishers: None

Subscribers:
* /gazebo (http://chuy:37707/)

root@chuy:/catkin_ws/src/urdf_sim_tutorial# rosmsg show std_msgs/Float64MultiArray
std_msgs/MultiArrayLayout layout
    std_msgs/MultiArrayDimension[] dim
        string label
        uint32 size
        uint32 stride
        uint32 data_offset
    float64[] data
root@chuy:/catkin_ws/src/urdf_sim_tutorial# █

```

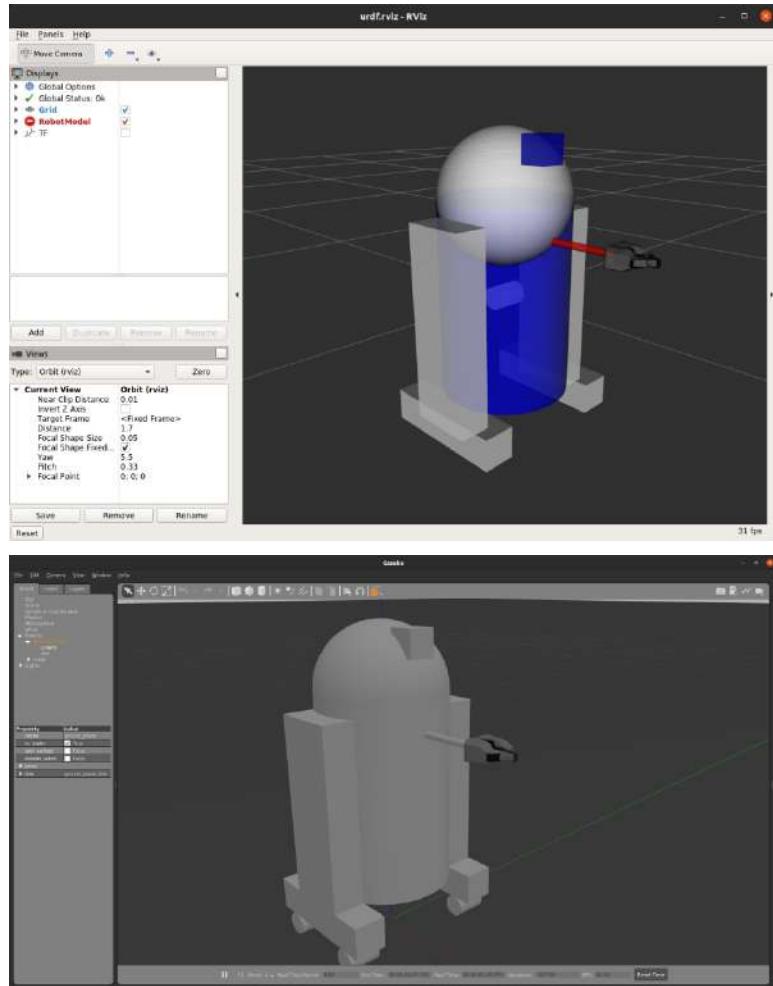
Como podemos observar tenemos un tema nuevo, en el cual esta suscrito Gazebo. En el podemos controlar la posicion de la pinza y del brazo robotico con una matriz publicando dichos valores en el tema.

Al publicar en dicho tema tendremos lo siguiente:

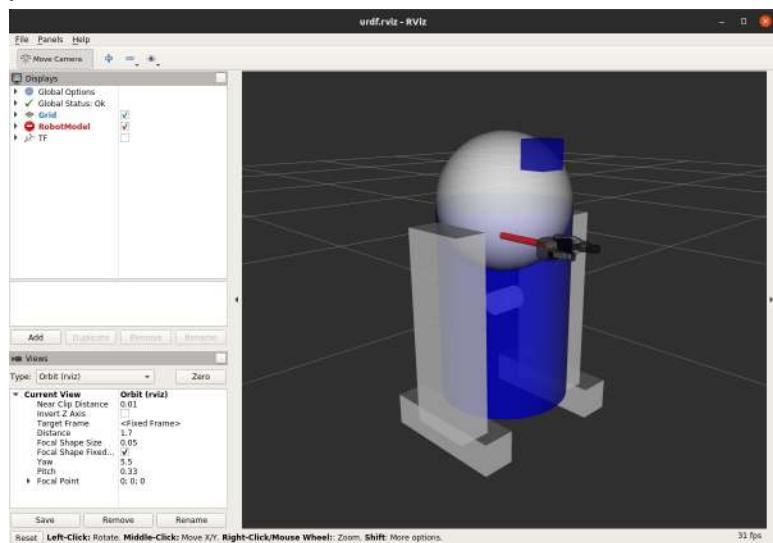
*rostopic pub /r2d2_gripper_controller/command
std_msgs/Float64MultiArray "data: [-0.2, 0.5, 0.5]"*

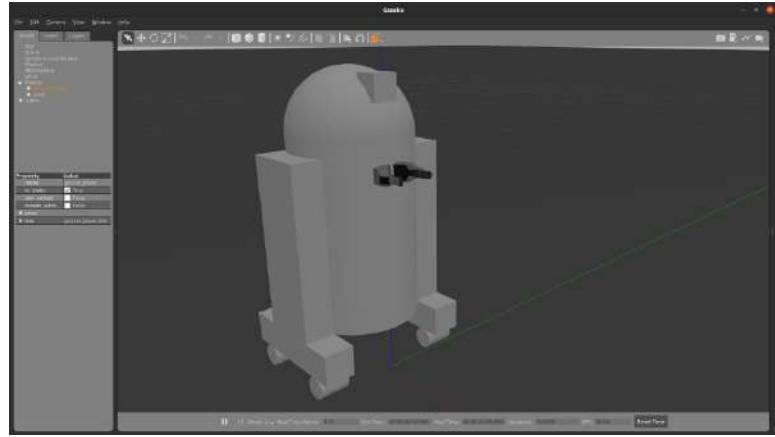
Lo que le indicamos es que el brazo robotico se retraija y se abra la pinza.

Antes:



Despues:





Cuando se publica este comando, las posiciones cambian de forma controlada al valor especificado. Esto se debe a que si se han especificado limites en dichas articulaciones en nuestro URDF.

Por ultimo, definiremos un controlador para las articulaciones de las ruedas:

Podemos especificar controladores para cada una de las ruedas individuales, pero por ahora controlaremos todas las ruedas juntas.

En un archivo YAML definimos el siguiente controlador, el cual tendra por nombre “diffdrive.yaml” y estara ubicado en la carpeta “config” del paquete “urdf_sim_tutorial”:

```

1 # Especificamos el uso de un "DiffDriveController" del paquete "position_controllers"
2 # para controlar las ruedas del robot (en esencia se utiliza para controlar una base
3 # móvil de accionamiento diferencial).
4 # Un robot móvil de cinemática diferencial es aquel que cuenta con dos ruedas motrices
5 # unidas por un eje. Cada una de estas ruedas tiene su propio motor que las hace girar.
6 type: "diff_drive_controller/DiffDriveController"
7 publish_rate: 50
8
9 left_wheel: ['left_front_wheel_joint', 'left_back_wheel_joint']
10 right_wheel: ['right_front_wheel_joint', 'right_back_wheel_joint']
11
12 wheel_separation: 0.44
13
14 # Los robots móviles usan la odometría para estimar su posición relativa a su localización inicial.
15 pose_covariance_diagonal: [0.001, 0.001, 0.001, 0.001, 0.001, 0.03]
16 twist_covariance_diagonal: [0.001, 0.001, 0.001, 0.001, 0.001, 0.03]
17
18 # Marco base
19 base_frame_id: base_link
20
21 # Límites de velocidad y aceleración del robot
22 linear:
23   x:
24     has_velocity_limits : true
25     max_velocity        : 0.2    # m/s
26     has_acceleration_limits: true
27     max_acceleration    : 0.6    # m/s^2
28 angular:
29   z:
30     has_velocity_limits : true
31     max_velocity        : 2.0    # rad/s
32     has_acceleration_limits: true
33     max_acceleration    : 6.0    # rad/s^2
34

```

NOTA: La interfaz de hardware en el URDF para las articulaciones de las ruedas debe especificarse como “hardware_interface/VelocityJointInterface”.

Ahora podemos lanzar esto creando un nuevo archivo de lanzamiento con el nombre “13-diffdrive.launch” ubicado en la carpeta “launch”, el cual sera una copia del archivo “12-gripper.launch”. Dicho archivo nuevo le agregaremos lo siguiente:

```

1  <launch>
2    <!-- Se cargara por defecto el modelo de robot especifico para este archivo de lanzamiento -->
3    <arg name="model" default="$(find urdf_sim_tutorial)/urdf/r3-diffdrive.urdf.xacro">
4    <!-- Declaramos un argumento para indicar el archivo de configuracion de pantalla (.rviz) para cargar en RViz -->
5    <arg name="rvizconfig" default="$(find urdf_sim_tutorial)/rviz/rviz.rviz" />
6
7    <!--
8      Incluimos el archivo de lanzamiento que inicia un mundo vacio en Gazebo y genera el modelo
9      de robot en Gazebo (el cual se le pasa como argumento), iniciando ademas el nodo
10     "robot_state_publisher" (el que publica transformaciones entre los diferentes marcos de coordenadas).
11   -->
12   <include file="$(find urdf_sim_tutorial)/launch/gazebo.launch">
13     <arg name="model" value="$(arg model)" />
14   </include>
15
16   <!-- Iniciamos el nodo "rviz" y le indicamos un archivo de configuracion de pantalla (.rviz) para cargar en RViz -->
17   <node name="rviz" pkg="rviz" type="rviz" args="-d $(arg rvizconfig)" />
18
19   <!--
20     Cargamos:
21       - Un archivo de parametros a un espacio de nombres llamado "r2d2_joint_state_controller".
22       - Un archivo de parametros a un espacio de nombres llamado "r2d2_head_controller".
23       - Un archivo de parametros a un espacio de nombres llamado "r2d2_gripper_controller".
24       - Un archivo de parametros a un espacio de nombres llamado "r2d2_diff_drive_controller".
25   -->
26   <rosparam command="load" file="$(find urdf_sim_tutorial)/config/joints.yaml" ns="r2d2_joint_state_controller" />
27   <rosparam command="load" file="$(find urdf_sim_tutorial)/config/head.yaml" ns="r2d2_head_controller" />
28   <rosparam command="load" file="$(find urdf_sim_tutorial)/config/gripper.yaml" ns="r2d2_gripper_controller" />
29   <rosparam command="load" file="$(find urdf_sim_tutorial)/config/diffdrive.yaml" ns="r2d2_diff_drive_controller" />
30
31   <!--
32     Iniciamos un nodo que ejecuta el script "spawner" del paquete "controller_manager" pasandole como
33     argumentos los nombres de los espacios de nombres en donde se encuentran los parametros que
34     especifican los tipos de controlador a usar.
35
36     "r2d2_joint_state_controller" sera el nombre de un controlador,
37     "r2d2_head_controller" sera el nombre de otro controlador,
38     "r2d2_gripper_controller" sera el nombre de otro controlador,
39     "r2d2_diff_drive_controller" sera el nombre de otro controlador.
40   -->
41   <node name="r2d2_controller_spawner" pkg="controller_manager" type="spawner"
42     args="r2d2_joint_state_controller r2d2_head_controller r2d2_gripper_controller r2d2_diff_drive_controller --shutdown-timeout 3"/>
43
44   <!-- Este paquete proporciona un complemento GUI para dirigir un robot mediante mensajes Twist -->
45   <node name="rqt_robot_steering" pkg="rqt_robot_steering" type="rqt_robot_steering">
46     <param name="default_topic" value="/r2d2_diff_drive_controller/cmd_vel"/>
47   </node>
48 </launch>

```

Como podemos observar, ahora cargamos los cuatro archivos de configuracion en donde definimos los controladores:

- joints.yaml
- head.yaml
- gripper.yaml
- diffdrive.yaml

Tambien iniciamos “rqt_robot_steering”, el cual es una GUI que nos ayuda a mover a nuestro robot publicando comandos de velocidad. En general se usan mensajes Twist.

```
root@chuy:/catkin_ws# rospack list | grep rqt_robot_steering
rqt_robot_steering /opt/ros/melodic/share/rqt_robot_steering
root@chuy:/catkin_ws#
```

```
root@chuy:/catkin_ws# rosmsg show geometry_msgs/Twist.msg
geometry_msgs/Vector3 linear
  float64 x
  float64 y
  float64 z
geometry_msgs/Vector3 angular
  float64 x
  float64 y
  float64 z
```

Ahora ejecutamos el siguiente comando para lanzar el archivo de lanzamiento que acabamos de crear:

```
roslaunch urdf_sim_tutorial 13-diffdrive.launch &
```

```
root@chuy:/catkin_ws# rosnode list
/gazebo
/gazebo_gui
/r2d2_controller_spawner
/robot_state_publisher
/rosout
/rqt_robot_steering
/rviz
root@chuy:/catkin_ws# rosnode info /rqt_robot_steering
-----
Node [/rqt_robot_steering]
Publications:
* /r2d2_diff_drive_controller/cmd_vel [geometry_msgs/Twist]
* /rosout [rosgraph_msgs/Log]

Subscriptions:
* /clock [rosgraph_msgs/Clock]

Services:
* /rqt_robot_steering/get_loggers
* /rqt_robot_steering/set_logger_level

contacting node http://chuy:38929/ ...
Pid: 16704
Connections:
* topic: /r2d2_diff_drive_controller/cmd_vel
  * to: /gazebo
  * direction: outbound (33665 - 192.168.3.9:58470) [21]
  * transport: TCPROS
* topic: /rosout
  * to: /rosout
  * direction: outbound (33665 - 192.168.3.9:58468) [15]
  * transport: TCPROS
* topic: /clock
  * to: /gazebo (http://chuy:36255/)
  * direction: inbound
  * transport: TCPROS
root@chuy:/catkin_ws#
```

Informacion sobre el nodo “rqt_robot_steering”.

Este nodo publica en el tema “/r2d2_diff_drive_controller/cmd_vel” el cual se especifico en el archivo de lanzamiento.

```

root@chuy:/catkin_ws# rostopic list
/clock
/gazebo/link_states
/gazebo/model_states
/gazebo/parameter_descriptions
/gazebo/parameter_updates
/gazebo/set_link_state
/gazebo/set_model_state
/joint_states
/r2d2_diff_drive_controller/cmd_vel ←
/r2d2_diff_drive_controller/odom ←
/r2d2_diff_drive_controller/parameter_descriptions ←
/r2d2_diff_drive_controller/parameter_updates ←
/r2d2_gripper_controller/command
/r2d2_head_controller/command
/rosout
/rosout_agg
/tf
/tf_static
root@chuy:/catkin_ws# rostopic info /r2d2_diff_drive_controller/cmd_vel
Type: geometry_msgs/Twist

Publishers:
* /rqt_robot_steering (http://chuy:38929/)

Subscribers:
* /gazebo (http://chuy:36255/)

root@chuy:/catkin_ws# rostopic info /r2d2_diff_drive_controller/odom
Type: nav_msgs/Odometry

Publishers:
* /gazebo (http://chuy:36255/)

Subscribers: None

root@chuy:/catkin_ws# rostopic info /r2d2_diff_drive_controller/parameter_descriptions
Type: dynamic_reconfigure/ConfigDescription

Publishers:
* /gazebo (http://chuy:36255/)

Subscribers: None

root@chuy:/catkin_ws# rostopic info /r2d2_diff_drive_controller/parameter_updates
Type: dynamic_reconfigure/Config

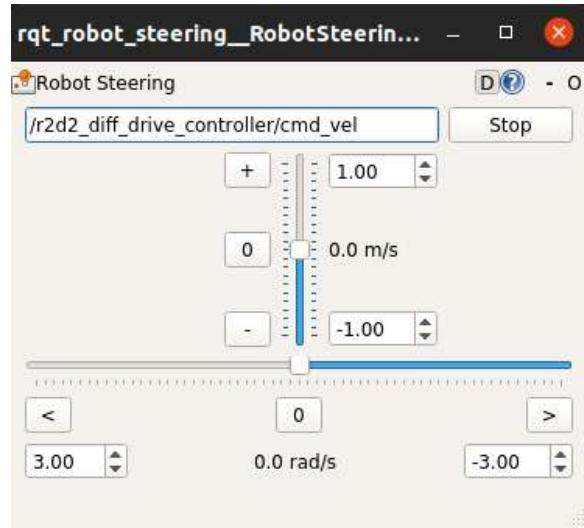
Publishers:
* /gazebo (http://chuy:36255/)

Subscribers: None

root@chuy:/catkin_ws# █

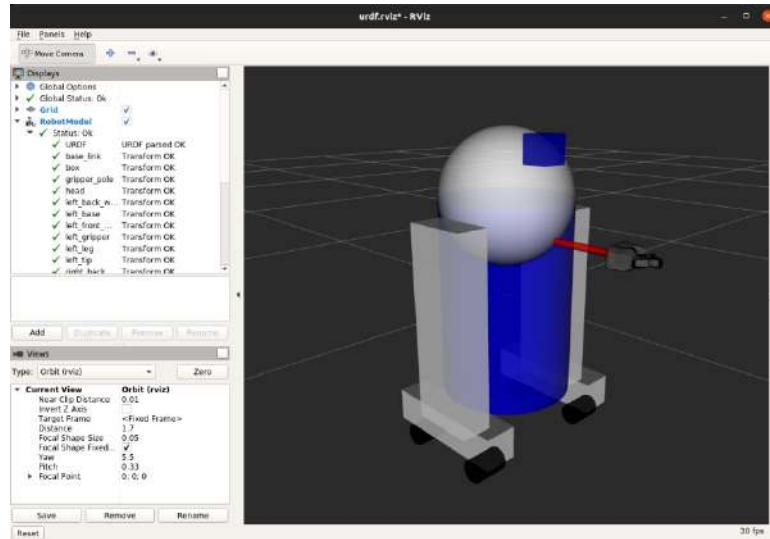
```

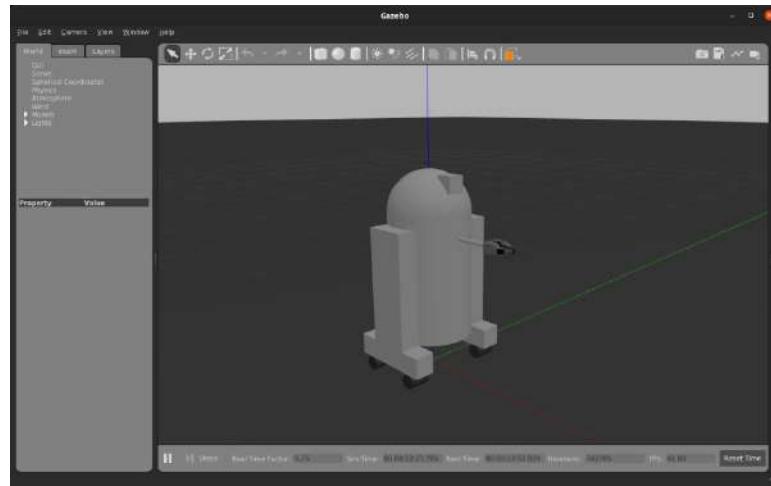
Como podemos observar tenemos temas nuevos.



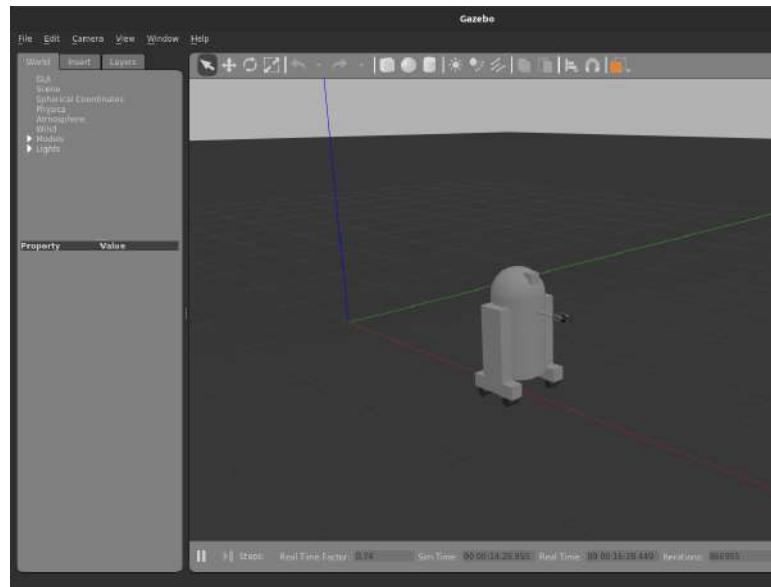
Esta es la GUI que nos permite mover a nuestro robot, la cual publica en el tema "/r2d2_diff_drive_controller/cmd_vel" (donde esta suscrito Gazebo).

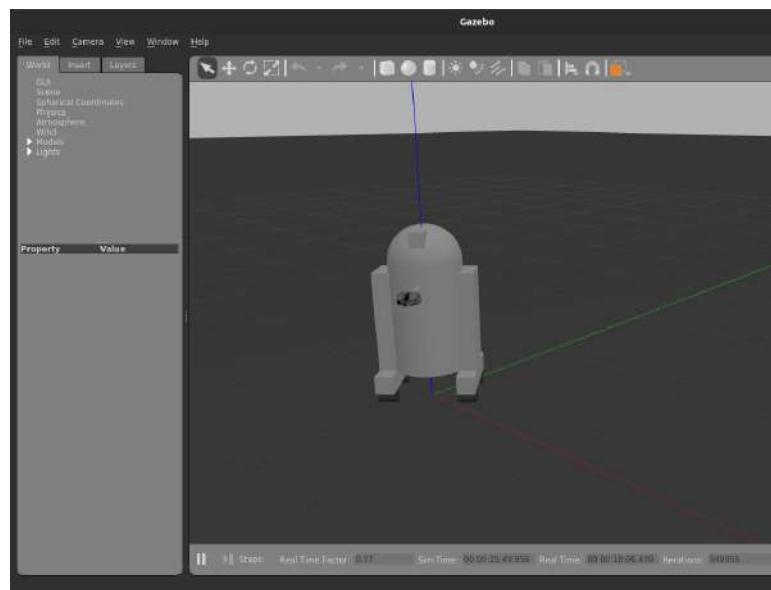
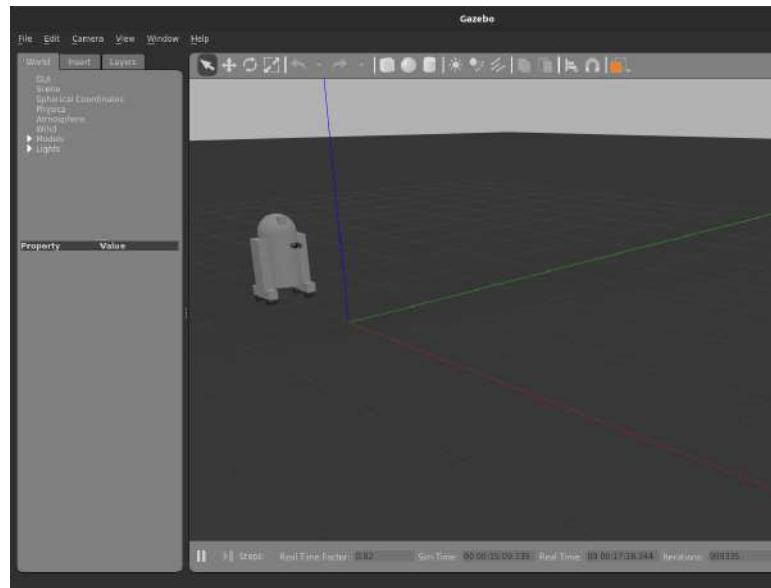
Antes de mover a nuestro robot, en RViz y Gazebo tenemos lo siguiente:

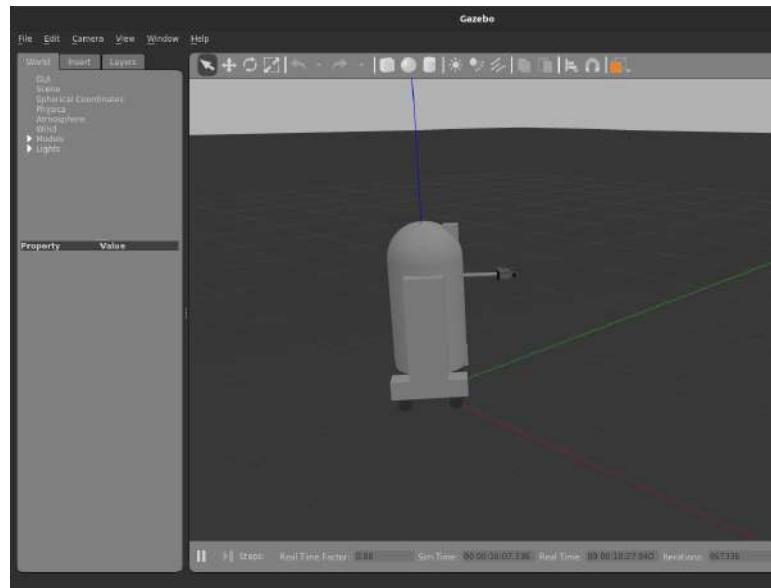




Al usar la GUI para mover al robot tenemos lo siguiente:







Lo que podemos hacer es lo siguiente:

- Moverlo hacia adelante o hacia
- Hacerlo rotar sobre su propio eje

Hasta aqui hemos terminado esta parte del uso de “ros_control” en Gazebo utilizando un modelo de robot en URDF.

Como repaso general, lo que hemos visto hasta el momento es:

- Como crear un modelo de robot con URDF.
- Como usar macros en URDF para simplificar el código y poder reutilizar ciertas definiciones en el URDF.
- Comousar un URDF en Gazebo.
- Como usar complementos de Gazebo en ROS.
- Como funciona “ros_control”.

Aquí estuvimos hablando mucho de controladores, de interfaces de hardware, interfaces de controlador, interfaces de transmisión, entre otras cosas.

- Como agregar elementos de transmisión a un URDF (los cuales vinculan los actuadores a las articulaciones).
- Al final de cuentas, como trabajar conjuntamente con “ros_control”, RViz, Gazebo y ROS.

Modificando una fabrica (simulada) en URDF

Lo que haremos sera visualizar una fabrica en RViz, tendremos algunos robots ya predefinidos (creados por una comunidad) y realizaremos algunas modificaciones. Todo esto con la idea de familiarizarnos un poco con lo que es la creacion de un entorno de trabajo para robots.

Haremos uso de un paquete llamado “hrwros_support” que nos proporciona el curso “Hello (Real) World with ROS – Robot Operating System” de la plataforma “edx”.

<https://www.edx.org/es/course/hello-real-world-with-ros-robot-operating-system>

A este paquete leharemos algunas modificaciones a modo de practica.

NOTA: Este paquete estara incluido en el material utilizado para la practica (es un paquete que puede descargarse de la plataforma iniciando sesion en la plataforma y accediendo al material del curso).

Paquetes necesarios

Lo primero que haremos sera instalar los siguientes paquetes:

- *universal_robots* (en realidad es un conjunto de paquetes)
Se encuentra en el siguiente enlace:

https://github.com/ros-industrial/universal_robot/tree/melodic-devel

Este conjunto de paquetes es parte del programa “ROS-Industrial”, el cual cuenta con modelos URDF para varios brazos roboticos entre otras cosas mas.

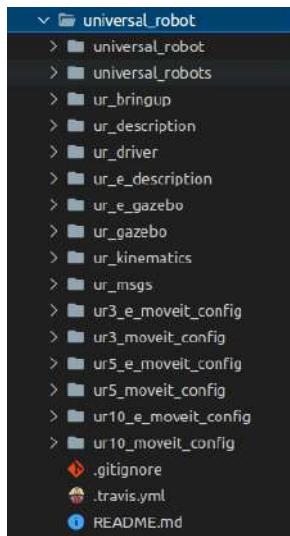
Lo que haremos sera clonar el paquete (conjunto de paquetes) en nuestro espacio de trabajo ROS dentro del directorio “src”, instalaremos las dependencias necesarias del paquete y por ultimo construiremos el paquete ejecutando el comando “catkin_make”. Los comandos a ejecutar son los siguientes:

```
cd $ROS_WORKSPACE/src
```

```

git clone -b melodic-devel
    https://github.com/ros-industrial/universal_robot.git
cd ..
rosdep update
rosdep install --rostdistro melodic --ignore-src --from-paths
    src
catkin_make -DCATKIN_WHITELIST_PACKAGES=""

```



- *hrwros_support* (paquete del cual ya se hablo y se encuentra en el material de la practica)
Este paquete hace uso de algunos paquetes de “universal_robots” para cargar modelos de robot ya predefinidos (ya que, recordando, crear un modelo de robot desde cero es bastante complicado y se necesita de mucho conocimiento).

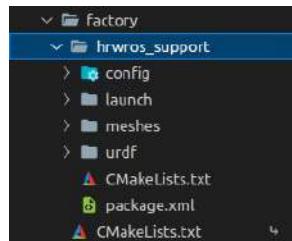
Lo que haremos sera copiar dicho paquete en nuestro espacio de trabajo ROS dentro del directorio “src/factory” (creamos la carpeta “factory”) y despues construimos el paquete ejecutando el comando “catkin_make”. Los comandos a ejecutar son los siguientes:

Para copiar archivos/carpetas del computador locar al contenedor Docker se ejecuta el siguiente comando:

(el comando se ejecuta en una terminal del computador local)
`docker cp <ruta-local>`
`<id-contenedor>:<ruta-contenedor>`

(una vez copiado dicho paquete proseguimos con los comandos)

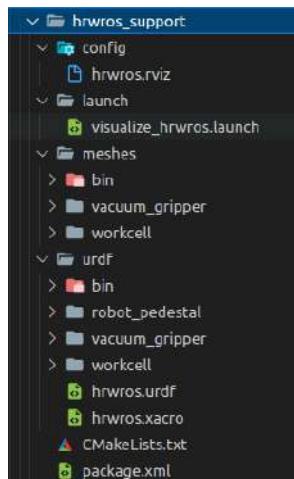
```
cd $ROS_WORKSPACE/src  
catkin_make --source $ROS_WORKSPACE/src/factory  
source devel/setup.bash
```



```
root@chuy:/catkin_ws# rospack list | grep hrwros  
hrwros_support /catkin_ws/src/factory/hrwros_support  
root@chuy:/catkin_ws#
```

Visualizacion de la fabrica

Antes de visualizar la fabrica veamos como esta conformado el paquete “hrwros_support”:



- *config*

Aqui se encuentra un archivo de configuracion de pantalla (.rviz) para cargar en RViz.

- *launch*

Aqui se encuentra el archivo de lanzamiento para visualizar la fabrica en RViz.

Dicho archvo hace lo siguiente:

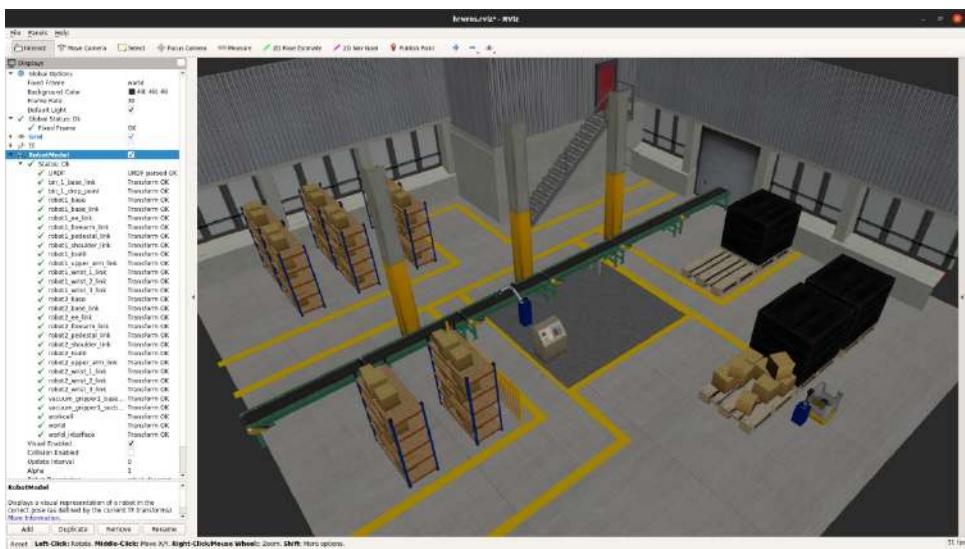
- Declara unos argumentos.
- Carga el archivo URDF al servidor de parametros (en el parametro “robot_description”).
- Inicia el nodo “robot_state_publisher” (el cual se suscribe al tema “/joint_states” y publica transformaciones entre marcos de referencia en el tema “/tf”, en el cual escucha rviz).
- Inicia el nodo “joint_state_publisher” con algunos valores articulares por defecto (el cual publica el estado de las articulaciones no fijas que lee del parametro “robot_description” del contenido del URDF y las publica en el tema “/joint_states”).
- Inicia el nodo “rviz” (al cual se le pasa un archivo de configuracion de pantalla inicial y abre RViz).
- *meshes*
Aqui se encuentran algunas mallas para cargar en el archivo URDF para algunas partes de robot.
- *urdf*
Aqui se encuentra el modelo de la fabrica (creado en Xacro y despues convertido a URDF).

Para convertir un archivo Xacro a URDF se utiliza el siguiente comando:

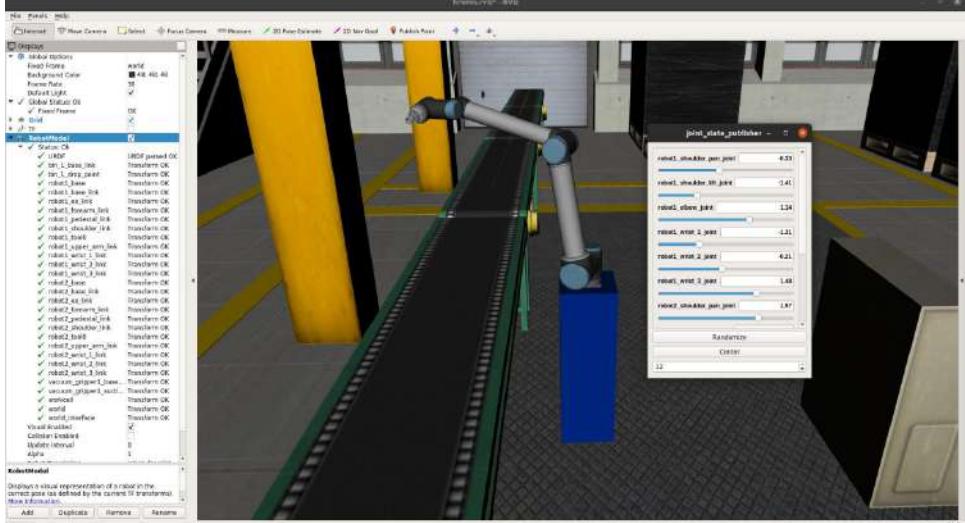
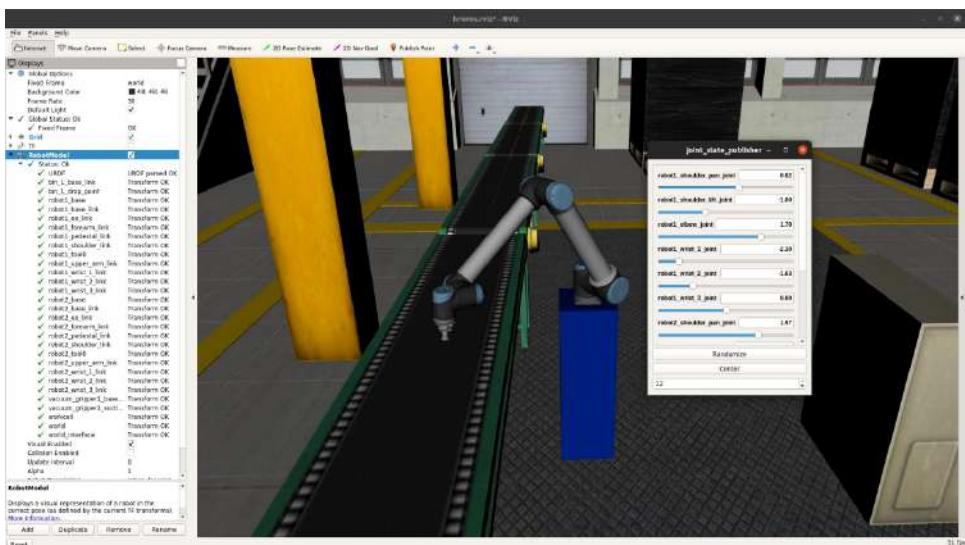
```
rosrun xacro xacro <archivo-xacro> > <archivo-urdf>
```

Para visualizar la fabrica ejecutamos el siguiente comando:

```
roslaunch hrwros_support visualize_hrwros.launch &
```



Visualización de la fábrica.



Los robots presentes los podemos manipular realizando modificaciones a los valores articulares (esta es la GUI del nodo “joint_state_publisher”).

```
root@chuy:/catkin_ws# rosnome list
/joint state publisher
/robot state publisher
/rosout
/rviz
root@chuy:/catkin_ws# rostopic list
/clicked_point
/initialpose
/joint_states
/move_base_simple/goal
/rosout
/rosout_agg
/tf
/tf_static
root@chuy:/catkin_ws# rostopic info /initialpose
Type: geometry_msgs/PoseWithCovarianceStamped
Publishers:
* /rviz (http://chuy:34715/)

Subscribers: None

root@chuy:/catkin_ws# rostopic info /move_base_simple/goal
Type: geometry_msgs/PoseStamped
Publishers:
* /rviz (http://chuy:34715/)

Subscribers: None

root@chuy:/catkin_ws# rostopic info /clicked_point
Type: geometry_msgs/PointStamped
Publishers:
* /rviz (http://chuy:34715/)

Subscribers: None

root@chuy:/catkin_ws#
```

Tenemos algunos temas nuevos en donde publica RViz.

```
root@chuy:/catkin_ws# rostopic echo /joint_states -n 1
header:
  seq: 5735
  stamp:
    sec: 165731132
    nsec: 484365940
  frame_id: ''
name: [robot1_shoulder_pan_joint, robot1_shoulder_lift_joint, robot1_elbow_joint, robot1_wrist_1_joint,
       robot1_wrist_2_joint, robot1_wrist_3_joint, robot2_shoulder_pan_joint, robot2_shoulder_lift_joint,
       robot2_elbow_joint, robot2_wrist_1_joint, robot2_wrist_2_joint, robot2_wrist_3_joint]
position: [-0.3290872286209409, -1.40554055240160, 1.189069222658692, 1.300150188954876, -0.2079734336676582, 1.4790618213101725, 1.5707963267950605, -1.5525750894041779, 1.5525750894041783, -1.578793626795, -1.53435385213356, 0.8]
velocity: []
effort: []

root@chuy:/catkin_ws#
```

Podemos observar que en el tema “/joint_states” se esta publicando el estado de todas las articulaciones no fijas presentes en el URDF.

Como vimos, el comando anterior nos abrio RViz con nuestro mundo cargado y visualizado, ademas de la GUI “joint_state_publisher” con controles deslizables que nos permite cambiar los valores de las articulaciones no fijas de nuestros robots, haciendo que los robots visualizados en RViz se muevan.

Modificaciones a la fabrica

Como practica realizaremos tres modificaciones a la fabrica, es decir, realizaremos modificaciones al URDF que modela a la fabrica:

Los archivos se crearan en el paquete “hrwros_support” en la carpeta “urdf”.

Al archivo de lanzamiento “visualize_hrwros.launch” le agregamos lo siguiente para poder cargar los URDF que iremos creando:

```
<group if="$(eval $arg(file)!=")">
| <param name="robot_description" command="$(find xacro)/xacro $(find hrwros_support)/urdf/practice$(arg file).xacro" />
</group>
```

Mediante un argumento llamado “file” se especificara el archivo URDF a cargar.

- Agregar un contenedor en frente de “robot1” (al otro lado de la banda transportadora).

Lo quearemos sera lo siguiente:

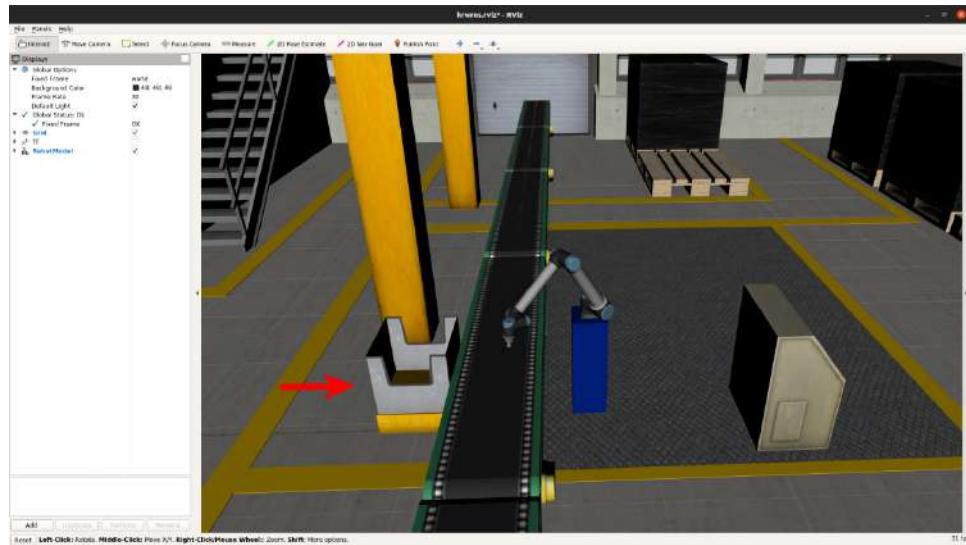
- Incluir el archivo Xacro con la macro para el contenedor.
- Llamar a la macro para incluir el contenedor.
- Agregar una articulacion fija que conecte el contenedor con el mundo de la fabrica.

Creamos una copia del archivo “hrwros.xacro” y le damos por nombre “practice1.xacro”, al cual le agregamos lo siguiente:

```
126  <!--
127  Agregamos un contenedor en frente de "robot1" (al otro lado de la banda transportadora).
128  -->
129
130  <xacro:bin_urdf prefix="bin_2_" />
131
132  <joint name="bin_2_joint" type="fixed">
133  | <parent link="robot1_pedestal_link" />
134  | <child link="bin_2_base_link" />
135  | <origin xyz="1.5 0 0" />
136  </joint>
```

Ejecutamos el siguiente comando para visualizar dichos cambios en la fabrica:

```
roslaunch hrwros_support visualize_hrwros.launch file:=1
```



Es importante mencionar que podemos hacer mediciones en RViz.

RViz proporciona una herramienta para hacer mediciones dentro de la visualización de nuestro modelo. Con esta herramienta podemos ayudarnos a posicionar correctamente nuestros enlaces en el mundo de la fabrica. Ademas, RViz cuenta con otras herramientas bastante interesantes.



La linea roja fue la medicion que se hizo dentro de la fabrica (de esta manera podemos posicionar de mejor manera los enlaces).

- Agregar una esfera de color verde posicionada por debajo de la escalera.

Lo que haremos sera lo siguiente:

- Agregar un enlace.
 - Especificar un elemento visual
 - Especificar su origen
 - Especificar un elemento de geometria de tipo esfera
 - Especificar un elemento de material de tipo color (será color verde)
 - Agregar una articulación fija que conecte a la esfera con el mundo de la fabrica.

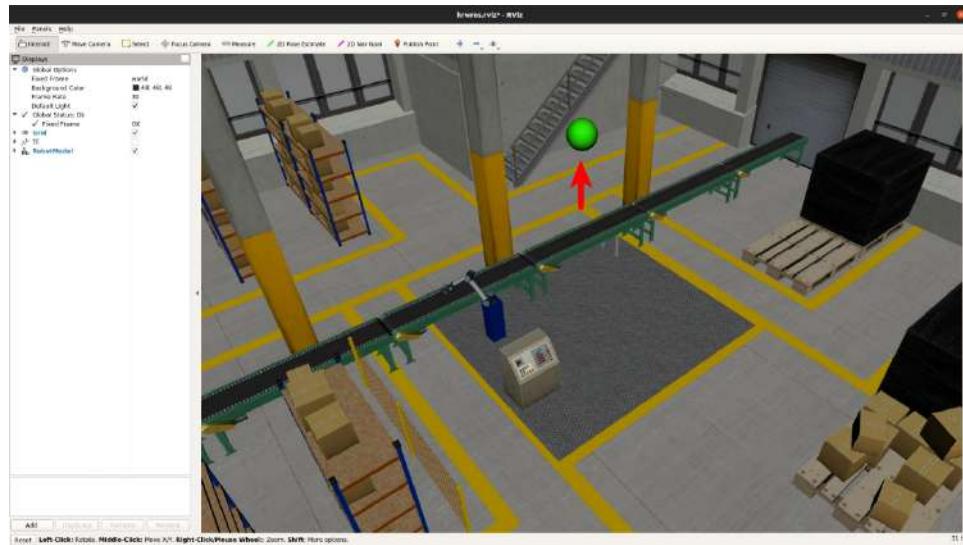
Creamos una copia del archivo “hrwros.xacro” y le damos por nombre “practice2.xacro”, al cual le agregamos lo siguiente:

```

126  <!--
127  Agregamos una esfera de color verde posicionada por debajo de la escalera.
128  -->
129
130  <link name="green_sphere_link">
131  |   <visual>
132  |   |   <origin xyz="0 0 0.5"/>
133  |   |   <!--
134  |   |   |   La forma del objeto visual puede ser una de las siguientes:
135  |   |   |   - <box> (size)
136  |   |   |   - <cylinder> (radius, length)
137  |   |   |   - <sphere> (radius)
138  |   |   |   - <mesh> (filename, scale)
139  |   |   |
140  |   |   <geometry>
141  |   |   |   <sphere radius="0.5"/>
142  |   |   </geometry>
143  |   |   <material name="green_sphere_color">
144  |   |   |   <!-- Solo se admiten valores entre 0 y 1 (por eso hacemos la conversion) -->
145  |   |   |   <color rgba="#{49/255} #{236/255} #{19/255} 1"/>
146  |   |   </material>
147  |   |   </visual>
148  |   </link>
149
150  <joint name="green_sphere_joint" type="fixed">
151  |   <parent link="world_interface"/>
152  |   <child link="green_sphere_link"/>
153  |   <origin xyz="6 -5 0"/>
154  </joint>
```

Ejecutamos el siguiente comando para visualizar dichos cambios en la fabrica:

roslaunch hrwros_support visualize_hrwros.launch file:=2



- Reemplazar “robot2” (el modelo UR5) por un nuevo robot (el modelo Fanuc LR Mate 200ic).

Lo que haremos sera lo siguiente:

- Incluir el archivo Xacro con la macro para el robot.
- Llamar a la macro para incluir al robot.
- Agregar una articulacion fija que conecte al robot con el mundo de la fabrica.

Aqui necesitaremos de un robot y mallas que nos proporcionan en el curso antes mencionado. Como nuevo tendremos:

Son directorios que adentro contienen archivos.

- *hrwros_support/meshes/lrmate200ic*
- *hrwros_support/urdf/robot_lrmate200ic*
- *hrwros_support/urdf/include*

(todo el material se encuentra incluido en el material utilizado para la practica)

Creamos una copia del archivo “hrwros.xacro” y le damos por nombre “practice3.xacro”, al cual le agregamos lo siguiente:

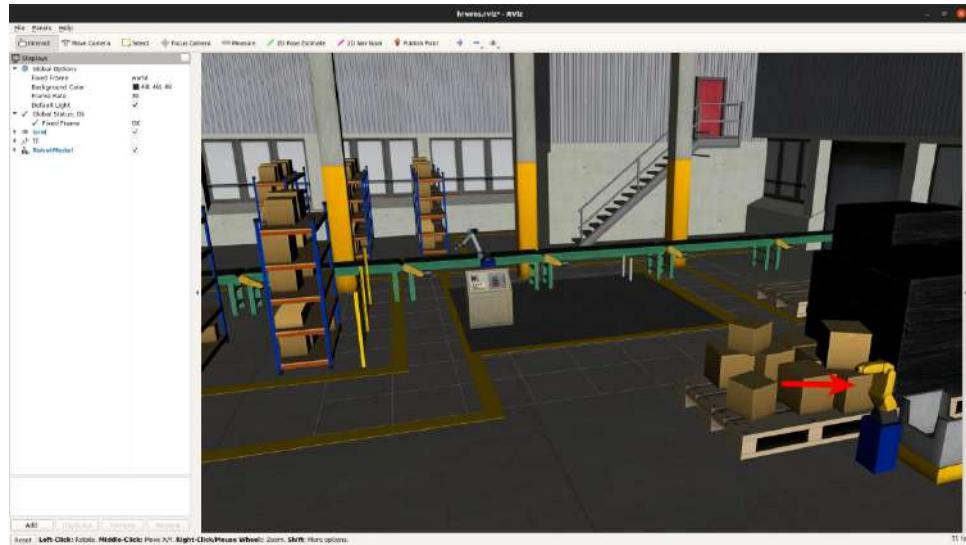
```

118  <!-- Comentamos este robot -->
119  <!--
120  <xacro:include filename="$(find ur_description)/urdf/ur5.urdf.xacro" />
121  <xacro:ur5_robot prefix="robot2_" joint_limited="true" />
122  <joint name="robot2-robot2_pedestal_joint" type="fixed">
123  | <parent link="robot2_pedestal_link" />
124  | <child link="robot2_base_link" />
125  | <origin xyz="0 0 0.6" rpy="0 0 ${radians(90)}" />
126  </joint>
127  -->
128
129  <!--
130  Reemplazamos "robot2" (el UR5) por un nuevo robot (el Fanuc LR Mate 200iC).
131  -->
132
133  <xacro:include filename="$(find hrwros_support)/urdf/robot_lrmate200ic/lrmate200ic_macro.xacro" />
134  <xacro:fanuc_lrmate200ic prefix="robot2_" joint_limited="true" />
135  <joint name="robot2-robot2_pedestal_joint" type="fixed">
136  | <parent link="robot2_pedestal_link" />
137  | <child link="robot2_base_link" />
138  | <origin xyz="0 0 0.6" rpy="0 0 ${radians(90)}" />
139  </joint>

```

Ejecutamos el siguiente comando para visualizar dichos cambios en la fabrica:

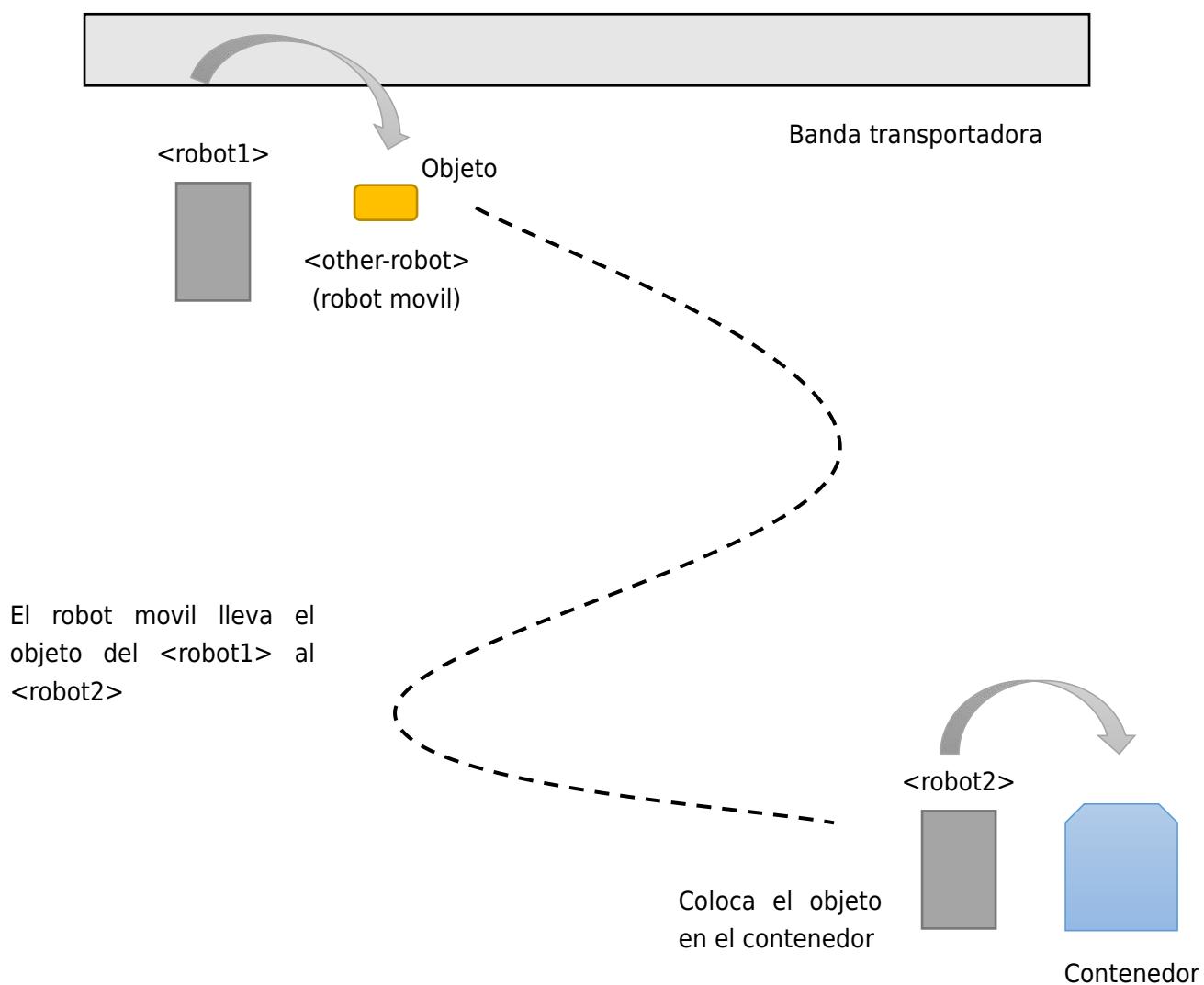
roslaunch hrwros_support visualize_hrwros.launch file:=3



¿Que se podria hacer?

NOTA: El escenario de la fabrica ya no lo volveremos a usar pero pondremos unsa situacion en la cual podria ser util su uso (mas adelante utilizaremos otros escenarios para fines practicos, ya que el escenario de la fabrica es algo complejo de usar).

Podriamos tener el siguiente escenario:



La tarea que puede realizar un robot manipulador puede estar definida por una serie de trayectorias que debe seguir el brazo robótico para recoger un objeto y dejarlo en otra posición.

Cabe mencionar que todos los robots involucrados de alguna manera deben estar comunicados para optimizar los tiempos de producción y para que estén coordinados unos con otros. Aquí es donde entra ROS.

Ahora bien, para el robot móvil que necesita desplazarse en la fábrica necesita planificar su trayectoria para llegar de un punto a otro, la cual dicha trayectoria puede verse modificada en el instante ante cualquier obstrucción o algún inconveniente en el camino. Este es un tema bastante común que necesita

solucionarse en los robots moviles.

A continuacion hablaremos sobre la navegacion de robots moviles.

Navegacion autonoma

La navegacion de un robot nos permite guiar el curso de un robot movil a traves de un entorno con obstaculos.

Las tareas incolumbradas en la navegacion de un robot movil son:

- La percepcion del entorno a traves de sus sensores (los cuales pueden ser camaras, sensores de distancia, etc.), de modo que le permita al robot crear una abstraccion del mundo.
- La planificacion de una trayectoria libre de obstaculos para alcanzar su destino.
- El guiado del vehiculo a traves de la referencia construida.
Recordando que el vehiculo puede interaccionar con ciertos elementos del entorno mientras se lleva la tarea de navegacion.

Introduccion a TurtleBot

Estremos usando un robot movil llamado “Turtlebot”.

Turtlebot es un robot movil pequeno, programable y basado en ROS para su uso en educacion, investigacion, pasatiempos, etc.

Tenemos dos versiones de este robot para usar en ROS:

- *Turtlebot2*

Este robot fue instalado pero el problema que se tenia era que a la hora de simularlo se movia bastante lento y parecia no respetar los comandos que se le daban. Por tanto, este robot no se utilizara.

- *Turtlebot3* (como el de la imagen)

Este robot movil es el que estaremos usando debido a que no se encontraron problemas a la hora de utilizarlo en la practica.



La tecnologia central de Turtlebot3 es SLAM, la navegacion y la manipulacion, lo que lo hace adecuado para robots de servicio domestico.

El Turtlebot puede ejecutar algoritmos SLAM (localizacion y mapeo simultaneos) para construir un mapa y poder conducir sobre el.

Instalacion de TurtleBot3

Para instalar Turltebot necesitamos instalar tres paquetes (aunque hay mas, pero estos son los mas importantes), los cuales se mencionan en el sitio web:

<http://wiki.ros.org/turtlebot3>

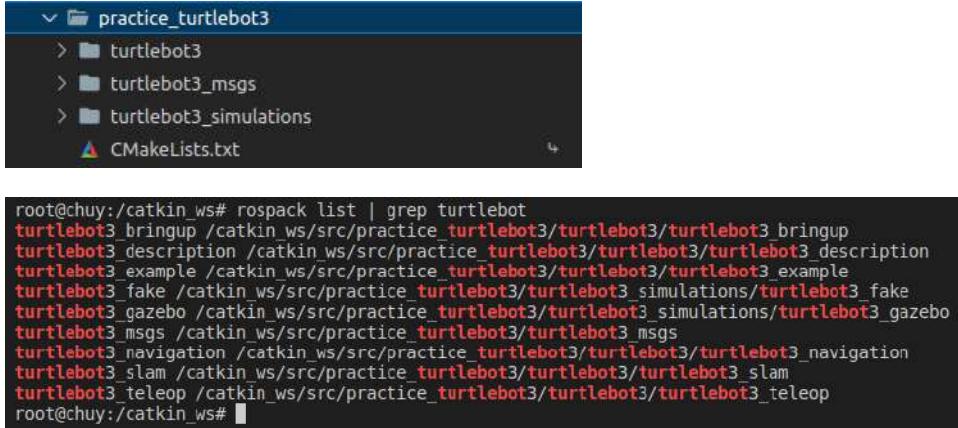
SLAM (localizacion y mapeo simultaneos):
Los algoritmos SLAM permiten contruir o actualizar un mapa de un entorno desconocido y, al mismo tiempo, realizar un seguimiento de la ubicacion de un robot dentro de el.

Crearemos una carpeta llamada “practice_turtlebot3” dentro de la carpeta “src” de nuestro espacio de trabajo ROS. Dentro de esta carpeta clonaremos los siguientes paquetes:

```
mkdir $ROS_WORKSPACE/src/practice_turtlebot3  
cd $ROS_WORKSPACE/src/practice_turtlebot3  
● turtlebot3  
  git clone  
  https://github.com/ROBOTIS-GIT/turtlebot3/tree/melodic-devel  
● turtlebot3_msgs  
  git clone  
  https://github.com/ROBOTIS-GIT/turtlebot3_msgs/tree/melodic-devel  
● turtlebot3_simulations  
  git clone https://github.com/ROBOTIS-GIT/turtlebot3_simulations.git
```

Despues ejecutamos los siguientes comandos:

```
cd ../../..  
rosdep update  
catkin_make --source $ROS_WORKSPACE/src/practice_turtlebot3  
source devel/setup.bash
```



```

root@chuy:/catkin_ws# rospack list | grep turtlebot
turtlebot3 bringup /catkin_ws/src/practice_turtlebot3/turtlebot3/turtlebot3 bringup
turtlebot3_description /catkin_ws/src/practice_turtlebot3/turtlebot3/turtlebot3_description
turtlebot3_example /catkin_ws/src/practice_turtlebot3/turtlebot3/turtlebot3_example
turtlebot3_fake /catkin_ws/src/practice_turtlebot3/turtlebot3_simulations/turtlebot3_fake
turtlebot3_gazebo /catkin_ws/src/practice_turtlebot3/turtlebot3_simulations/turtlebot3_gazebo
turtlebot3_msgs /catkin_ws/src/practice_turtlebot3/turtlebot3_msgs
turtlebot3_navigation /catkin_ws/src/practice_turtlebot3/turtlebot3/turtlebot3_navigation
turtlebot3_slam /catkin_ws/src/practice_turtlebot3/turtlebot3/turtlebot3_slam
turtlebot3_teleop /catkin_ws/src/practice_turtlebot3/turtlebot3/turtlebot3_teleop
root@chuy:/catkin_ws#

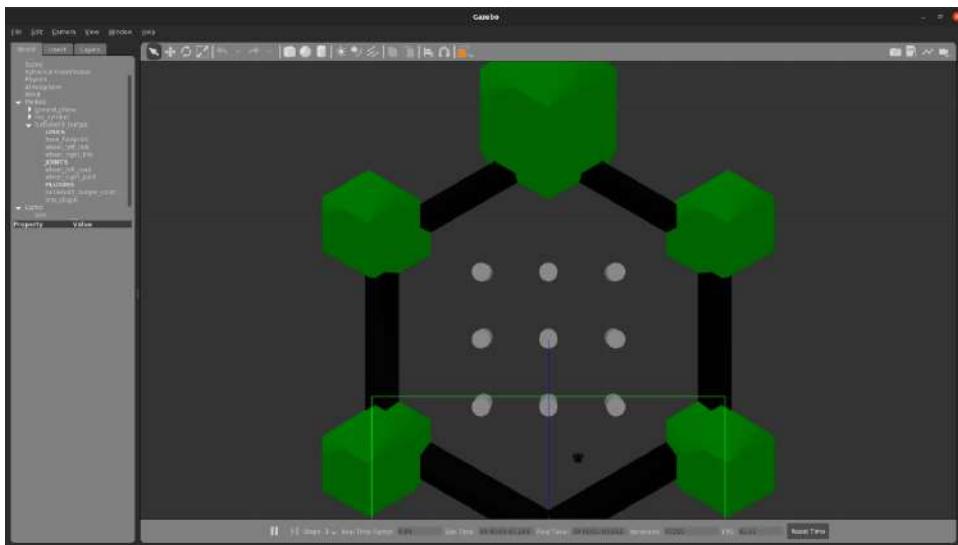
```

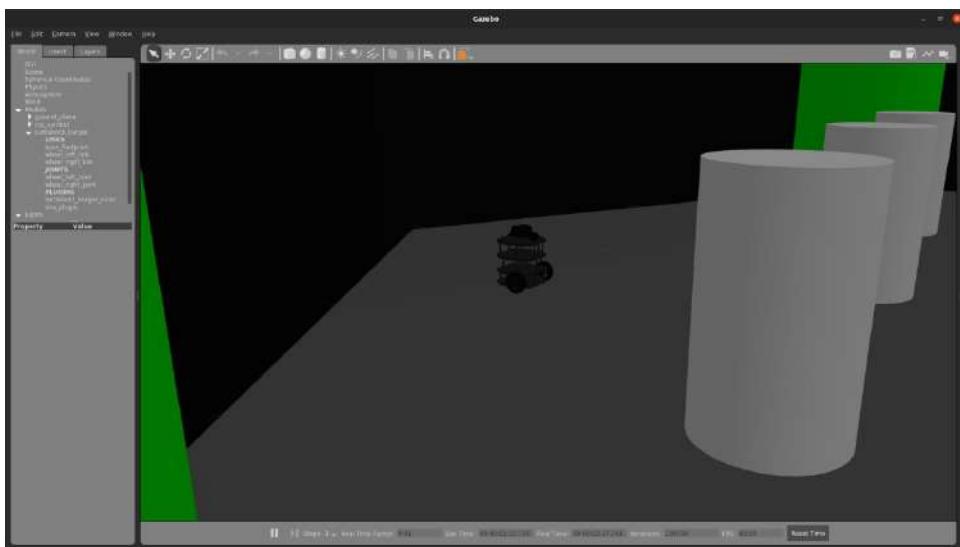
Antes de iniciar con la simulación debemos establecer en una variable de entorno el nombre por defecto del modelo de Turtlebot3 que deseamos usar (usaremos el “burger”):

```
export TURTLEBOT3_MODEL="burger"
```

Ya hecho todo lo anterior, al ejecutar el siguiente comando se nos abrirá un mundo en Gazebo para empezar a trabajar con Turtlebot:

```
rosrun turtlebot3_gazebo turtlebot3_world.launch
```





Este es el mundo que se carga con Turtlebot por defecto.

```
root@chuy:/catkin_ws# rosnode list
/gazebo
/gazebo_gui
/rosout
root@chuy:/catkin_ws# rostopic list
/clock
/cmd_vel
/gazebo/link_states
/gazebo/model_states
/gazebo/parameter_descriptions
/gazebo/parameter_updates
/gazebo/set_link_state
/gazebo/set_model_state
 imu
/joint_states
/odom
/rosout
/rosout_agg
/scan
/tf
```

Aqui se muestran los nodos que se estan ejecutando y los temas disponibles.

```
root@chuy:/catkin_ws# rosnode info /gazebo
-----
Node [/gazebo]
Publications:
 * /clock [rosgraph_msgs/Clock]
 * /gazebo/link_states [gazebo_msgs/Linkstates]
 * /gazebo/model_states [gazebo_msgs/ModelStates]
 * /gazebo/parameter_descriptions [dynamic_reconfigure/ConfigDescription]
 * /gazebo/parameter_updates [dynamic_reconfigure/Config]
 * /imu [sensor_msgs/Imu]
 * /joint_states [sensor_msgs/JointState]
 * /odom [nav_msgs/Odometry]
 * /rosout [rosgraph_msgs/Log]
 * /scan [sensor_msgs/LaserScan]
 * /tf [tf2_msgs/TFMessage]

Subscriptions:
 * /clock [rosgraph_msgs/Clock]
 * /cmd_vel [unknown type]
 * /gazebo/set_link_state [unknown type]
 * /gazebo/set_model_state [unknown type]
```

Podemos ver los temas a los que publica y a los que se suscribe el nodo “/gazebo”. Tenemos temas nuevos en donde publica Gazebo como lo son “/imu”, “/odom”, “/scan” y “/tf”.

Con respecto a estos temas, tenemos:

- “/odom” y “/tf”

Estos temas estan relacionados con la publicacion de informacion de odometria a traves de ROS.

Recordemos que los robots moviles usan la odometria para estimar su posicion relativa con respecto a su localizacion inicial.

La odometria es el uso de datos de sensores de movimiento para estimar el cambio de posicion a lo largo del tiempo.

En el tema “/odom” se publica informacion de odometria de un marco de referencia hijo con respecto a un marco de referencia padre, donde se muestra la pose del mismo y su velocidad.

```
root@chuy:/catkin_ws# rostopic info /odom
Type: nav_msgs/Odometry
Publishers:
  * /gazebo (http://chuy:45623/)
Subscribers: None

root@chuy:/catkin_ws# rosmsg show nav_msgs/Odometry
std_msgs/Header header
  uint32 seq
  time stamp
  string frame_id
string child_frame_id
geometry_msgs/PoseWithCovariance pose
  geometry_msgs/Pose pose
    geometry_msgs/Point position
      float64 x
      float64 y
      float64 z
    geometry_msgs/Quaternion orientation
      float64 x
      float64 y
      float64 z
      float64 w
    float64[36] covariance
geometry_msgs/TwistWithCovariance twist
  geometry_msgs/Twist twist
    geometry_msgs/Vector3 linear
      float64 x
      float64 y
      float64 z
    geometry_msgs/Vector3 angular
      float64 x
      float64 y
      float64 z
    float64[36] covariance
```

<frame_id>: aqui se especifica el origen de la transformacion de la odometria.

<child_frame_id>: aqui se especifica el destino de la transformacion de la odometria.

(estos campos se utilizaran en el tema “/tf”)

En el tema “/tf” se publica la transformacion del marco de coordenadas “base_footprint” (hijo) con respecto a un marco de coordenadas “odom” (padre).

```

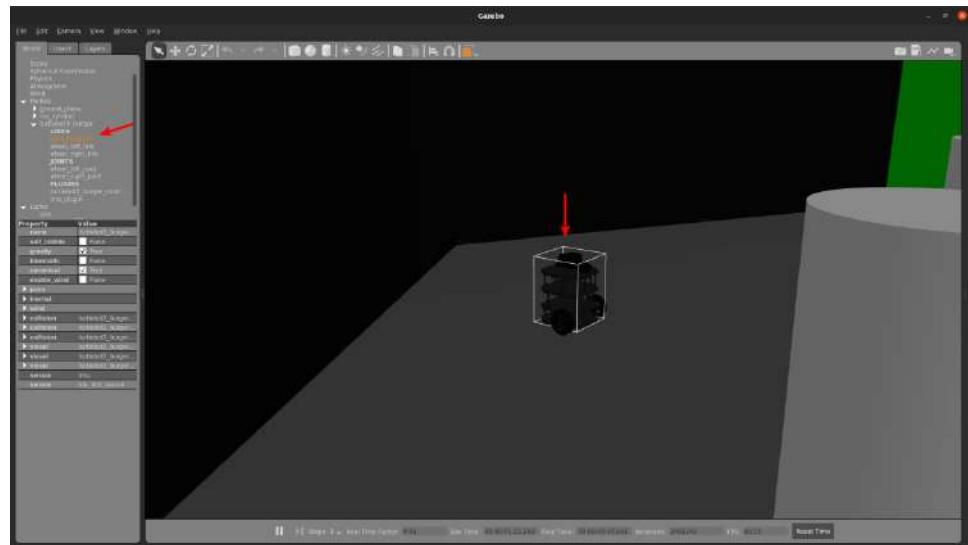
root@chuy:/catkin_ws# rostopic info /tf
Type: tf2_msgs/TFMessage

Publishers:
* /gazebo (http://chuy:45623/)

Subscribers: None

root@chuy:/catkin_ws# rostopic echo /tf -n 1
transforms:
-
  header:
    seq: 0
    stamp:
      secs: 2311
      nsecs: 591000000
    frame_id: "odom"
    child_frame_id: "base_footprint"
    transform:
      translation:
        x: -2.00628075953
        y: -0.498566720816
        z: -0.00100139582215
      rotation:
        x: -5.06800265815e-05
        y: 0.00385279818817
        z: 0.0113218269743
        w: 0.999528482248
-
root@chuy:/catkin_ws#

```



Cabe mencionar que todo esto esta relacionado con el controlador de transmision diferencial (el cual es un controlador de transmision diferencial que se puede conectar a un modelo con cualquier numero de ruedas izquierdas y derechas).

- “/imu”

```
root@chuy:/catkin_ws# rostopic info /imu
Type: sensor_msgs/Imu

Publishers:
 * /gazebo (http://chuy:44265/)

Subscribers: None

root@chuy:/catkin_ws# rosmsg show sensor_msgs/Imu
std_msgs/Header header
  uint32 seq
  time stamp
  string frame_id
geometry_msgs/Quaternion orientation
  float64 x
  float64 y
  float64 z
  float64 w
float64[9] orientation_covariance
geometry_msgs/Vector3 angular_velocity
  float64 x
  float64 y
  float64 z
float64[9] angular_velocity_covariance
geometry_msgs/Vector3 linear_acceleration
  float64 x
  float64 y
  float64 z
float64[9] linear_acceleration_covariance
```

Este tipo de mensaje sirve para almacenar datos de una IMU (unidad de medida inercial).

Una IMU es un dispositivo electronico que mide e informa acerca de la velocidad, orientacion y fuerzas gravitacionales de un aparato, usando una combinacion de acelerometros y giroscopos.

- “/scan”

```
root@chuy:/catkin_ws# rostopic info /scan
Type: sensor_msgs/LaserScan

Publishers:
 * /gazebo (http://chuy:44265/)

Subscribers: None

root@chuy:/catkin_ws# rosmsg show sensor_msgs/LaserScan
std_msgs/Header header
  uint32 seq
  time stamp
  string frame_id
float32 angle_min
float32 angle_max
float32 angle_increment
float32 time_increment
float32 scan_time
float32 range_min
float32 range_max
float32[] ranges
float32[] intensities
```

Este tipo de mensaje sirve para almacenar datos de un telemetro laser.

Un telemetro laser es un telemetro (es un dispositivo capaz de medir distancias de forma remota) que utiliza un rayo laser para determinar la distancia hasata un objeto.

```
root@chuy:/catkin_ws# rosnode info /gazebo_gui
-----
Node [/gazebo_gui]
Publications:
 * /rosout [rosgraph_msgs/Log]

Subscriptions:
 * /clock [rosgraph_msgs/Clock]
```

Este es el nodo de la interfaz grafica de Gazebo.

Control del TurtleBot3

Para poder al Turtlebot necesitamos ejecutar los siguientes comandos:

```
roslaunch turtlebot3_gazebo turtlebot3_world.launch &
roslaunch turtlebot3_teleop turtlebot3_teleop_key.launch
```

```
started roslaunch server http://chuy:44511/
SUMMARY
=====
PARAMETERS
 * /model: burger
 * /rostdistro: melodic
 * /rosversion: 1.14.12
NODES
/
  turtlebot3_teleop_keyboard (turtlebot3_teleop/turtlebot3_teleop_key)
ROS_MASTER_URI=http://localhost:11311
process[turtlebot3_teleop_keyboard-1]: started with pid [21166]
Control Your TurtleBot3!
-----
Moving around:
      w
    a   s   d
      x
w/x : increase/decrease linear velocity (Burger : ~ 0.22, Waffle and Waffle Pi : ~ 0.26)
a/d : increase/decrease angular velocity (Burger : ~ 2.84, Waffle and Waffle Pi : ~ 1.82)
space key, s : force stop
CTRL-C to quit
```

```

currently:    linear vel 0.0   angular vel 0.0
xxcurrently: linear vel -0.01      angular vel 0.0
scurrently:  linear vel -0.01      angular vel -0.1
scurrently:  linear vel 0.0   angular vel -0.1
currently:   linear vel 0.01  angular vel -0.1
wcurrently:  linear vel 0.02  angular vel -0.1
currently:   linear vel 0.02  angular vel 0.0

Control Your TurtleBot3!
-----
Moving around:
    w
    a   s   d
    x

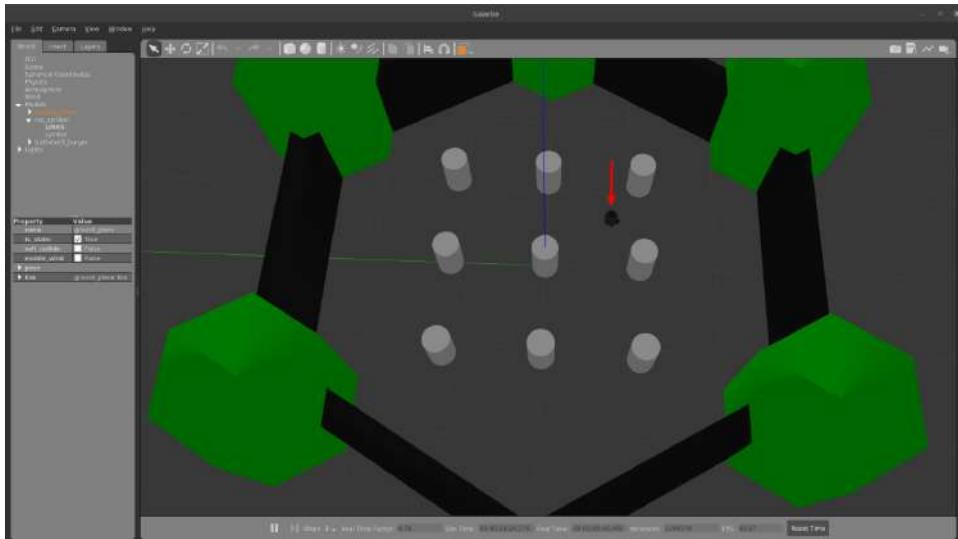
w/x : increase/decrease linear velocity (Burger : ~ 0.22, Waffle and Waffle Pi : ~ 0.26)
a/d : increase/decrease angular velocity (Burger : ~ 2.84, Waffle and Waffle Pi : ~ 1.82)

space key, s : force stop

CTRL-C to quit

acurrently:  linear vel 0.02  angular vel 0.1
currently:   linear vel 0.02  angular vel 0.2
wcurrently:  linear vel 0.03  angular vel 0.2
currently:   linear vel 0.04  angular vel 0.2
wcurrently:  linear vel 0.05  angular vel 0.2
currently:   linear vel 0.06  angular vel 0.2
currently:   linear vel 0.07  angular vel 0.2
dcurrently:  linear vel 0.07  angular vel 0.1
dcurrently:  linear vel 0.07  angular vel 0.0
dcurrently:  linear vel 0.07  angular vel -0.1
currently:   linear vel 0.0   angular vel 0.0

```



Como podemos observar, podemos aumentar/disminuir la velocidad lineal y aumentar/disminuir la velocidad angular, asi como tambien de forzar la detencion.

El ultimo comando ejecutado lanza un archivo de lanzamiento que a su vez lanza un nodo llamado “turtlebot3_teleop_keyboard” que ejecuta el script “turtlebot3_teleop_key” del paquete “turtlebot3_teleop”. Este script de Python publica comandos de velocidad en el tema “/cmd_vel” con el tipo de mensaje “geometry_msgs/Twist”.

```
root@chuy:/catkin_ws# rosnode list
/gazebo
/gazebo_gui
/rosout
/turtlebot3_teleop_keyboard
root@chuy:/catkin_ws# rosnode info /turtlebot3_teleop_keyboard
-----
Node [/turtlebot3_teleop_keyboard]
Publications:
 * /cmd_vel [geometry_msgs/Twist]
 * /rosout [rosgraph_msgs/Log]

Subscriptions:
 * /clock [rosgraph_msgs/Clock]

Services:
 * /turtlebot3_teleop_keyboard/get_loggers
 * /turtlebot3_teleop_keyboard/set_logger_level
```

```
root@chuy:/catkin_ws# rostopic info /cmd_vel
Type: geometry_msgs/Twist

Publishers:
 * /turtlebot3_teleop_keyboard (http://chuy:37075/)

Subscribers:
 * /gazebo (http://chuy:44651/)
```

```
root@chuy:/catkin_ws# rosmsg show geometry_msgs/Twist
geometry_msgs/Vector3 linear
  float64 x
  float64 y
  float64 z
geometry_msgs/Vector3 angular
  float64 x
  float64 y
  float64 z
```

Anteriormente hemos podido mover el Turtlebot usando la teleoperacion del teclado.

Tambien podemos mover al Turtlebot publicando comandos de velocidad directamente desde la terminal:

Para hacer esta tarea, recordemos, necesitamos publicar en le tema “cmd/vel” con un tipo de mensaje “geometry_msgs/Twist”.

Ejecutamos los siguientes comandos:

```
roslaunch turtlebot3_gazebo turtlebot3_world.launch &
■ rostopic pub /cmd_vel geometry_msgs/Twist '{linear:
{x: 0.5,y: 0,z: 0}, angular: {x: 0,y: 0,z: 0}}'
■ rostopic pub -r 10 /cmd_vel geometry_msgs/Twist
'{linear: {x: 0.5,y: 0,z: 0}, angular: {x: 0,y: 0,z: 0}}'
```

El primer comando publica en el tema y se queda publicando

hasta presionar Ctrl+C y el segundo comando publica en el tema con una frecuencia especifica (con una frecuencia de 10 Hz).

Despues de esto, veremos que el Turtlebot se mueve en linea recta.

Por ultimo, y como no podia faltar, tambien podemos controlar al Turtlebot por medio de un script de Python:

Lo que haremos sera escribir un pequeño programa para publicar mensajes sobre un tema especifico.

El nombre del nodo que publicara en dicho tema le pondremos por nombre “/drive_turtlebot_circle” (recordar que hay que dar nombres descriptivos a nuestros nodos).

Lo que intentaremos hacer es publicar comandos de velocidad para que el robot de vueltas en circulo.

Hay que tener en cuenta que publicaremos en el tema “/cmd_vel” con el tipo de mensaje “geometry_msgs/Twist”.

Para empezar, crearemos un nuevo paquete dentro del directorio “practice_turtlebot3” y despues crearemos el script de Python que publicara en el tema especificado. Para esto ejecutamos los siguientes comandos:

```
cd $ROS_WORKSPACE/src/practice_turtlebot3  
catkin_create_pkg test_drive_turtlebot rospy geometry_msgs  
cd test_drive_turtlebot  
mkdir launch scripts  
cd scripts  
touch drive_turtlebot_circle.py  
chmod +x drive_turtlebot_circle.py
```

Al script le agregamos lo siguiente:

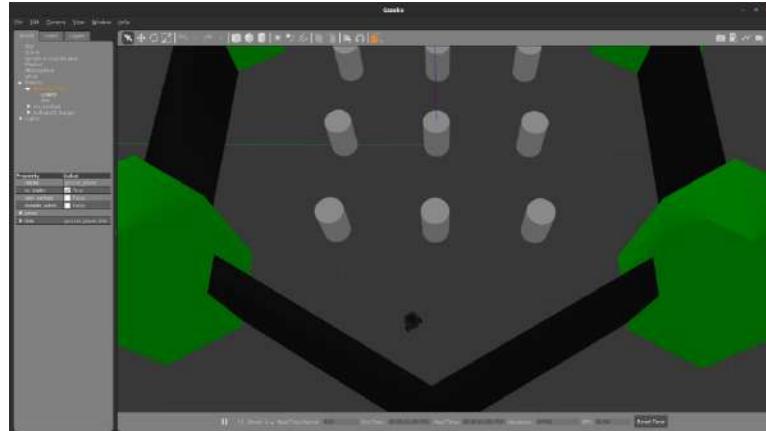
```

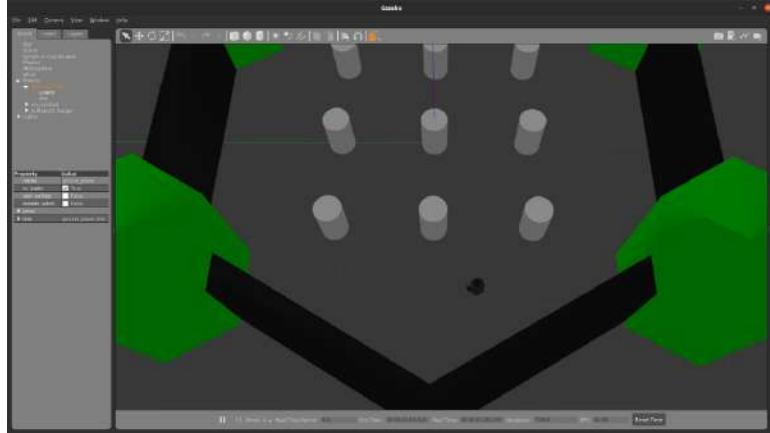
1  #!/usr/bin/env python
2
3  # Crearemos un nodo que manejará el turtlebot en círculos
4
5  import rospy
6  from geometry_msgs.msg import Twist
7
8  if __name__ == '__main__':
9
10     node_name="drive_turtlebot_circle"
11     topic_name="/cmd_vel"
12
13     rospy.init_node(node_name)
14     pub=rospy.Publisher(topic_name, Twist, queue_size=1)
15     rate=rospy.Rate(2)
16
17     move=Twist()
18     move.linear.x=0.2 # Es el eje para el movimiento lineal del robot
19     move.angular.z=0.5 # Es el eje de rotación del robot
20
21     rospy.loginfo('Listo, se empezará a publicar en el tema: %s'%(topic_name))
22
23     try:
24         while not rospy.is_shutdown():
25             pub.publish(move)
26             rate.sleep()
27     except rospy.exceptions.ROSInterruptException:
28         rospy.loginfo('Se ha apagado el nodo')

```

Después ejecutamos los siguientes comandos:

*rosrun turtlebot3_gazebo turtlebot3_world.launch &
rosrun test_drive_turtlebot drive_turtlebot_circle.py*





Veremos como el Turtlebot empieza a dar vueltas en circulo, esto debido a que hemos establecido una velocidad lineal y una velocidad angular al Turtlebot, por lo que al mismo tiempo que avanza tambien rota (sobre el eje z).

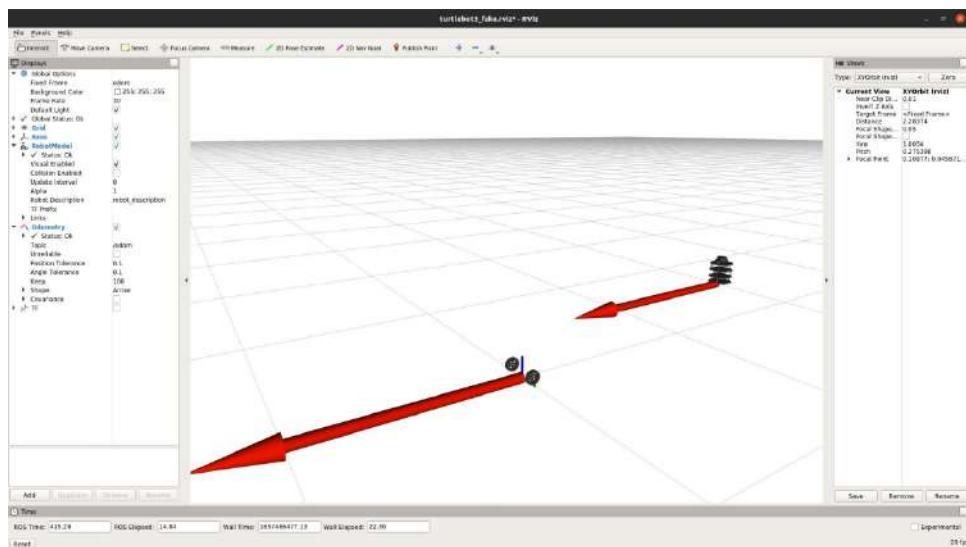
Lo que hemos realizado seria bastante interesante si la trayectoria seguida por el robot movil fuera controlada por un controlador (en otras palabras, que pueda navegar de forma autonoma), es decir, que tuvieramos un punto de partida y un punto de destino, en el cual el controlador en cada momento esta calculando dichos comandos de velocidad que se le deben aplicar al robot, es decir, a los motores de las ruedas (todo esto tiene que ver con la planificacion de una trayectoria).

A continuacion usaremos RViz para visualizar lo que esta haciendo el robot.

Uso de RViz para visualizar lo que esta haciendo el robot

Para visualizar en RViz el Turtlebot necesitamos ejecutar los siguientes comandos:

```
roslaunch turtlebot3_gazebo turtlebot3_world.launch &
roslaunch turtlebot3_fake turtlebot3_fake.launch
```



NOTA: El problema que se tiene es que aparece doble en RViz y parpadea el Turtlebot en la escena. Este problema puede darse a que estamos abriendo ventanas desde Docker, entre otros factores, pero realmente todo funciona bien, solo es un problema de visualización en RViz.

```
root@chuy:/catkin_ws# rosnome list
/gazebo
/gazebo_gui
/robot_state_publisher
/rosout
/rviz
/turtlebot3_fake_node
root@chuy:/catkin_ws# rosnode info /turtlebot3_fake_node
-----
Node [/turtlebot3_fake_node]
Publications:
 * /joint_states [sensor_msgs/JointState]
 * /odom [nav_msgs/Odometry]
 * /rosout [rosgraph_msgs/Log]
 * /tf [tf2_msgs/TFMessage]

Subscriptions:
 * /clock [rosgraph_msgs/Clock]
 * /cmd_vel [unknown type]
```

Información sobre el nodo “turtlebot3_fake_node”.

```
root@chuy:/catkin_ws# rosnode info /rviz
-----
Node [/rviz]
Publications:
 * /clicked_point [geometry_msgs/PointStamped]
 * /initialpose [geometry_msgs/PoseWithCovarianceStamped]
 * /move_base_simple/goal [geometry_msgs/PoseStamped]
 * /rosout [rosgraph_msgs/Log]

Subscriptions:
 * /clock [rosgraph_msgs/Clock]
 * /odom [nav_msgs/Odometry]
 * /tf [tf2_msgs/TFMessage]
 * /tf_static [tf2_msgs/TFMessage]
```

```

root@chuy:/catkin_ws# rosnode info /robot_state_publisher
Node [/robot_state_publisher]
  Publications:
    * /rosout [rosgraph_msgs/Log]
    * /tf [tf2_msgs/TFMessage]
    * /tf_static [tf2_msgs/TFMessage]

  Subscriptions:
    * /clock [rosgraph_msgs/Clock]
    * /joint_states [sensor_msgs/JointState]

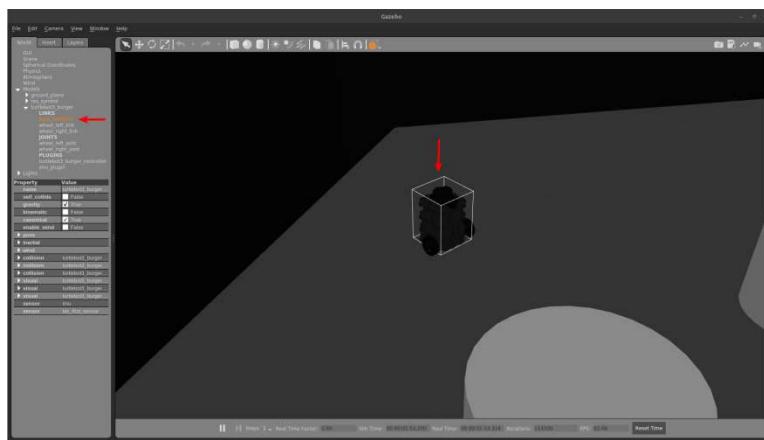
```

Con respecto al archivo de lanzamiento “turtlebot3_fake.launch” del paquete “turtlebot_fake”:

- Se declara un argumento que guarda el nombre del modelo a utilizar (en nuestro caso el Turtlebot 3 Burger).
- Se incluye un archivo de lanzamiento que inicia el mundo en Gazebo con el Turtlebot dentro.
- Se inicia un nodo que ejecuta el script “turtlebot3_fake_node” (el cual en realidad es un alias para el script “turtlebot3_fake.cpp”) del paquete “turtlebot3_fake”.

Este nodo, como observamos en la información de dicho nodo, publica el estado de las articulaciones del robot, publica la odometria del robot y ademas publica transformaciones entre los marcos de referencia “odom” y “base_footprint”.

En la navegación, el mensaje “nav_msgs/Odometry” almacena una estimación de la posición y velocidad del robot en el espacio libre.



```

root@chuy:/catkin_ws# rostopic echo /tf -n 1
transforms:
-
  header:
    seq: 0
    stamp:
      secs: 281
      nsecs: 965000000
      frame_id: "odom"
    child frame_id: "base_footprint"
    transform:
      translation:
        x: 0.0
        y: 0.0
        z: 0.0
      rotation:
        x: 0.0
        y: 0.0
        z: 0.0
        w: 1.0
  ...
root@chuy:/catkin_ws# 

```

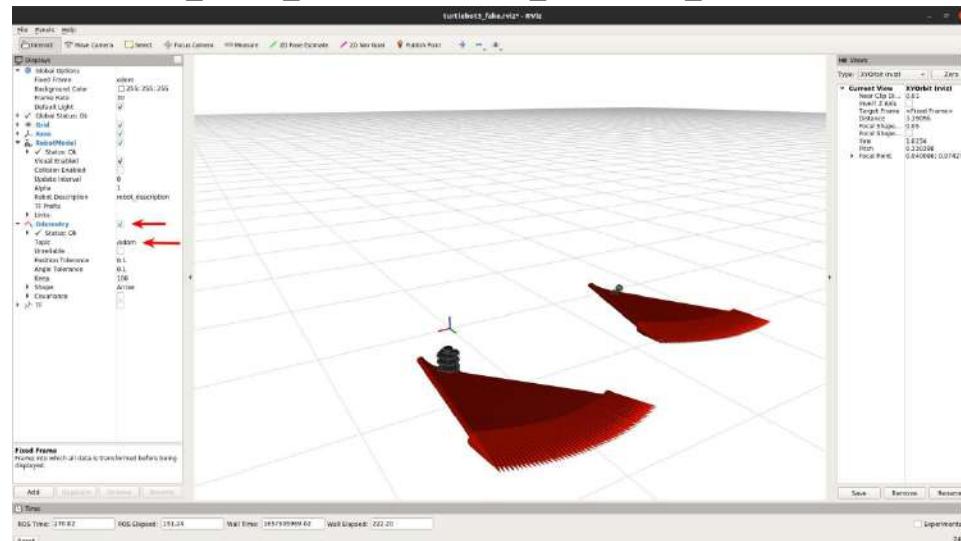
Tambien esta suscrito al tema “cmd_vel” por el cual controla la velocidad de traslacion y rotacion del robot (en m/s y rad/s, respectivamente).

- Se inicia el nodo “robot_state_publisher” (el cual se suscribe al tema “/joint_states” y publica transformaciones entre marcos de referencia en el tema “/tf”).
- Se inicia el nodo “rviz” que lanza RViz.

Lo que visualizaremos en RViz sera la odometria:

Ejecutamos el siguiente comando para mover al Turtlebot en circulos (del script ya mencionado):

rosrun test_drive_turtlebot drive_turtlebot_circle.py



En RViz veremos al Turtlebot moviendose y flechas apuntando en la direccion en la que se mueve.

Hasta aqui RViz nos ha ayudado a visualizar lo que hace el robot en la simulacion (en Gazebo).

Lo que nos gustaria hacer ahora es darle al Turtlebot una posicion objetivo y que pueda navegar de forma autonoma hasta esa posicion. Mas adelante estaremos hablando sobre esto.

¿Que es necesario para la navegacion?

Para la navegacion de un robot necesitamos en primer lugar:

- Un mapa (del mundo)
 - Saber nuestra ubicacion (localizacion)
 - Un objetivo (un planificado de rutas)
- Para llegar a dicho objetivo en ocasiones se elegira la ruta mas segura y en otras la mas rapida.
- Saber evitar obstaculos

A continuacion profundizaremos mas en cada uno de estos puntos.

Teoria del mapeo

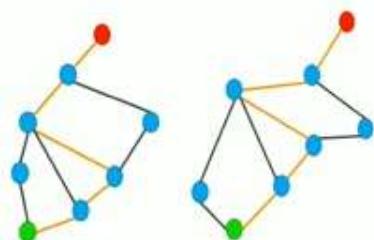
Un mapa es absolutamente obligatorio para la navegacion.

¿Como obtener un mapa?

Representaciones para un mapa:

- Representacion topologica

Es basicamente un grafo (aqui lo unico que nos interesa es saber como estan conectadas las diferentes ubicaciones).



Ambas representaciones son las mismas.

Este tipo de representaciones de mapa tienen la ventaja de que son muy ligeras.

- Representacion metrica

Es una descripcion detallada de como se ve el entorno y usa coordenadas precisas.

Maneras de obtener un mapa:

- Usar un mapa preexistente
- Hacer un mapa nosotros mismos usando el robot
La creacion de un mapa se llama **mapeo**.

Por ejemplo, mientras nos movemos con el robot en un entorno simulado en Gazebo podemos ir construyendo un mapa en RViz con lo que se vaya capturando de los sensores del robot.

A este proceso se le llama **SLAM**, es decir, localizacion y mapeo simultaneos (el robot esta construyendo un mapa mientras realiza un seguimiento de su propia posicion).

Mapeo con Turtlebot

Usaremos una implementacion de SLAM especifica llamada “**gmapping**”.

El SLAM es una tecnica para dibujar un mapa estimando la ubicacion actual del robot en un espacio arbitrario.

Sobre los metodos SLAM tenemos:

- *Gmapping*

Con este metodo podemos crear un mapa 2D a partir de datos de posicion y laser recopilados por el robot movil.

- *Cartographer*

Este metodo proporciona localizacion y mapeo simultaneos en tiempo real en 2D y 3D.

- *Hector*

Este metodo se puede utilizar sin odometria.

- *Karto*

Este metodo necesita que el robot movil proporcione datos de

odometria y que este equipado con un telémetro láser fijo.

Con los escaneres laser, el robot puede recopilar informacion sobre donde se encuentran otros objetos en el entorno.

Hay un nodo que se encarga de procesar la informacion recopilada del escaneo laser y luego publica en el tema “/map”.

NOTA: Al crear un mapa con TurtleBot3, es una buena práctica escanear todos los rincones del mapa.

Teoria de la localizacion

Tenemos varias formas de determinar donde nos encontramos:

- Metodos globales
 - GPS
 - WiFi: Hotspots (son puntos de acceso a Internet que nos permiten conectarnos a una red WiFi)
 - Torres celulares (son donde se montan los equipos de comunicaciones electricas y las antenas)

Con estos metodos obtenemos informacion de donde nos encontramos con respecto al mundo.

- Metodos locales
 - Escaneo laser
 - Camaras

Con estos metodos obtenemos informacion local.

Cada metodo ofrece precisiones diferentes, por lo que unos pueden ser mas precisos que otros pero de igual manera mas caros.

Localizacion con Turtlebot

Lo primero que haremos sera realizar un mapeo del entorno con Turtlebot para obtener un mapa.

Antes de comenzar, necesitamos instalar los siguientes paquetes ejecutando los siguientes comandos:

```
sudo apt-get install ros-melodic-slam-gmapping
```

```
sudo apt-get install ros-melodic-map-server
```

```
sudo apt-get install ros-melodic-navigation
```

Esto nos evitara tener algunos problemas mas adelante.

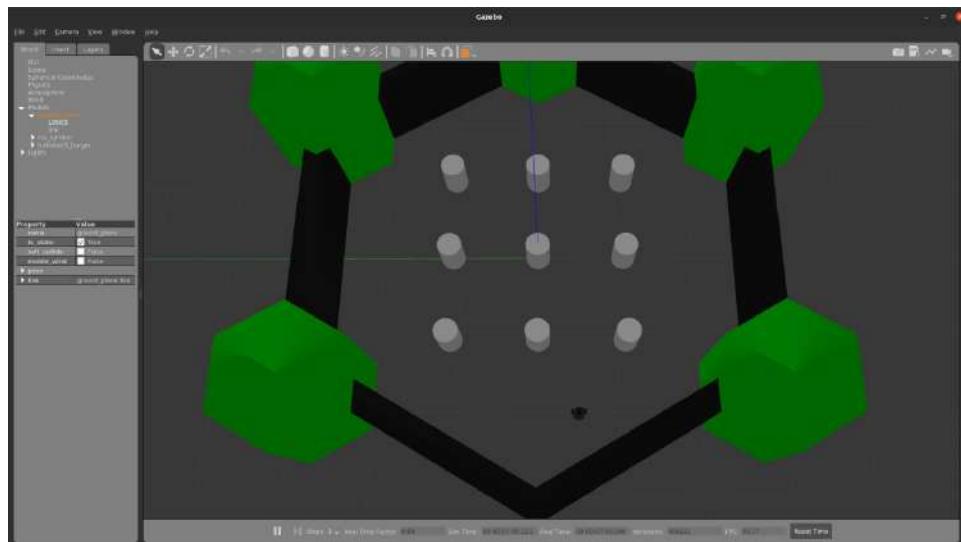
Ahora, lo primero que haremos sera ejecutar los siguientes comandos:

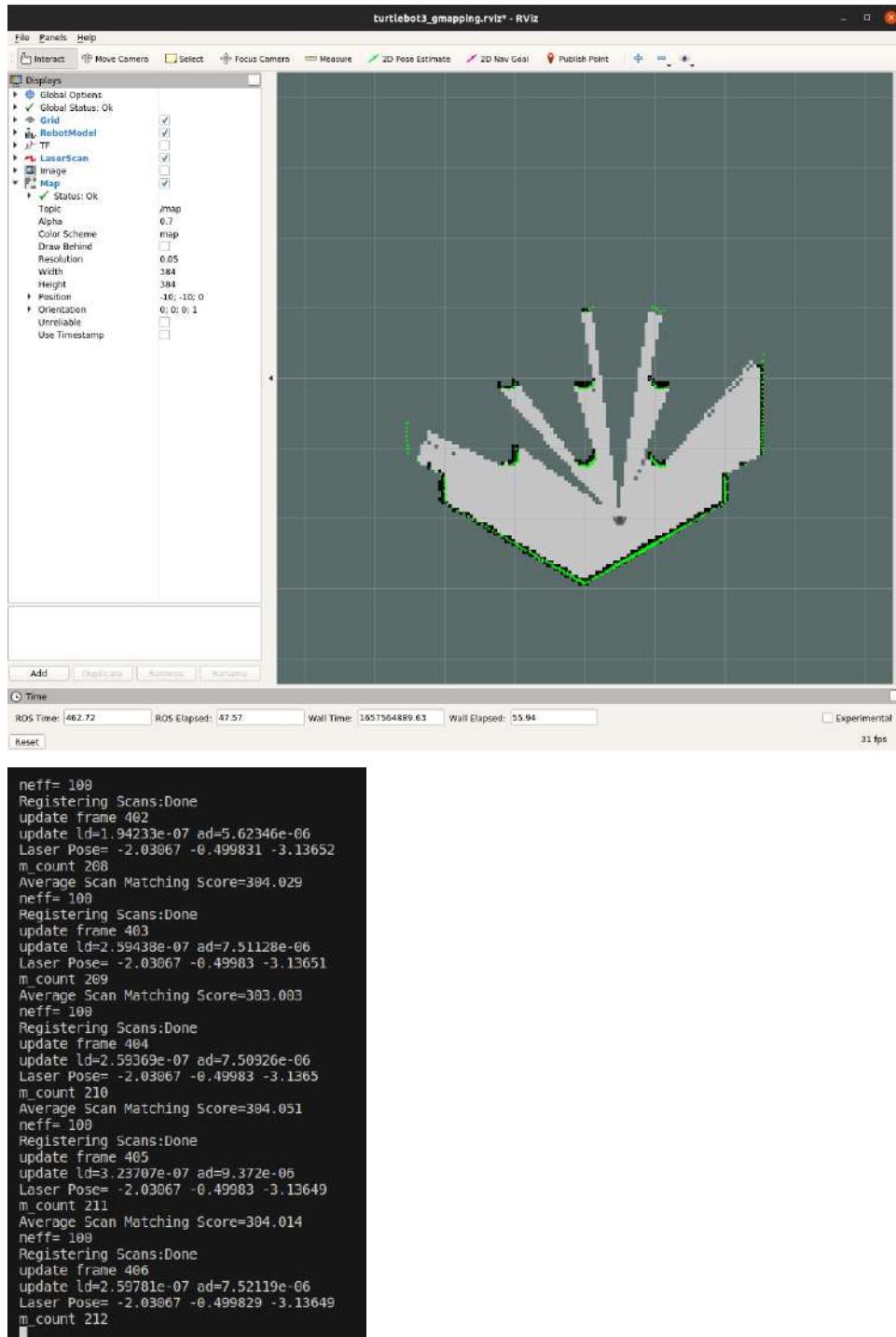
```
rosrun turtlebot3_gazebo turtlebot3_world.launch &
```

```
rosrun turtlebot3_slam turtlebot3_slam.launch
```

```
slam_methods:=gmapping
```

Este ultimo comando iniciara un nodo llamado “turtlebot3_slam_gmapping” y abrirá RViz con una visualización de nuestro Turtlebot en el mundo usando los datos del mapeo (de los sensores) para pintar un mapa.





Esto es parte de lo que se muestra en consola despues de ejecutar el ultimo comando (si quisieramos seguir ejecutando comandos necesitamos abrir otra terminal).

```

root@chuy:/catkin_ws# rosnode list
/gazebo
/gazebo_gui
/robot_state_publisher
/rosout
/rviz
/turtlebot3_slam_gmapping
root@chuy:/catkin_ws# rosnode info /turtlebot3_slam_gmapping
...
Node [/turtlebot3_slam_gmapping]
Publications:
  * /map [nav_msgs/OccupancyGrid]
  * /map_metadata [nav_msgs/MapMetaData]
  * /rosout [rosgraph_msgs/Log]
  * /tf [tf2_msgs/TFMessage]
  * /turtlebot3_slam_gmapping/entropy [std_msgs/FloatingPoint]

Subscriptions:
  * /clock [rosgraph_msgs/Clock]
  * /scan [sensor_msgs/LaserScan]
  * /tf [tf2_msgs/TFMessage]
  * /tf_static [tf2_msgs/TFMessage]

Services:
  * /dynamic_map
  * /turtlebot3_slam_gmapping/get_loggers
  * /turtlebot3_slam_gmapping/set_logger_level

```

Informacion del nodo “/turtlebot3_slam_gmapping”.

Con respecto al tema “/map”, se publica un tipo de mensaje “nav_msgs/OccupancyGrid”, el cual representa un mapa de cuadricula 2D.

```

root@chuy:/catkin_ws# rosmsg show nav_msgs/OccupancyGrid
std_msgs/Header header
  uint32 seq
  time stamp
  string frame_id
nav_msgs/MapMetaData info
  time map_load_time
  float32 resolution
  uint32 width
  uint32 height
  geometry_msgs/Pose origin
    geometry_msgs/Point position
      float64 x
      float64 y
      float64 z
    geometry_msgs/Quaternion orientation
      float64 x
      float64 y
      float64 z
      float64 w
  int8[] data

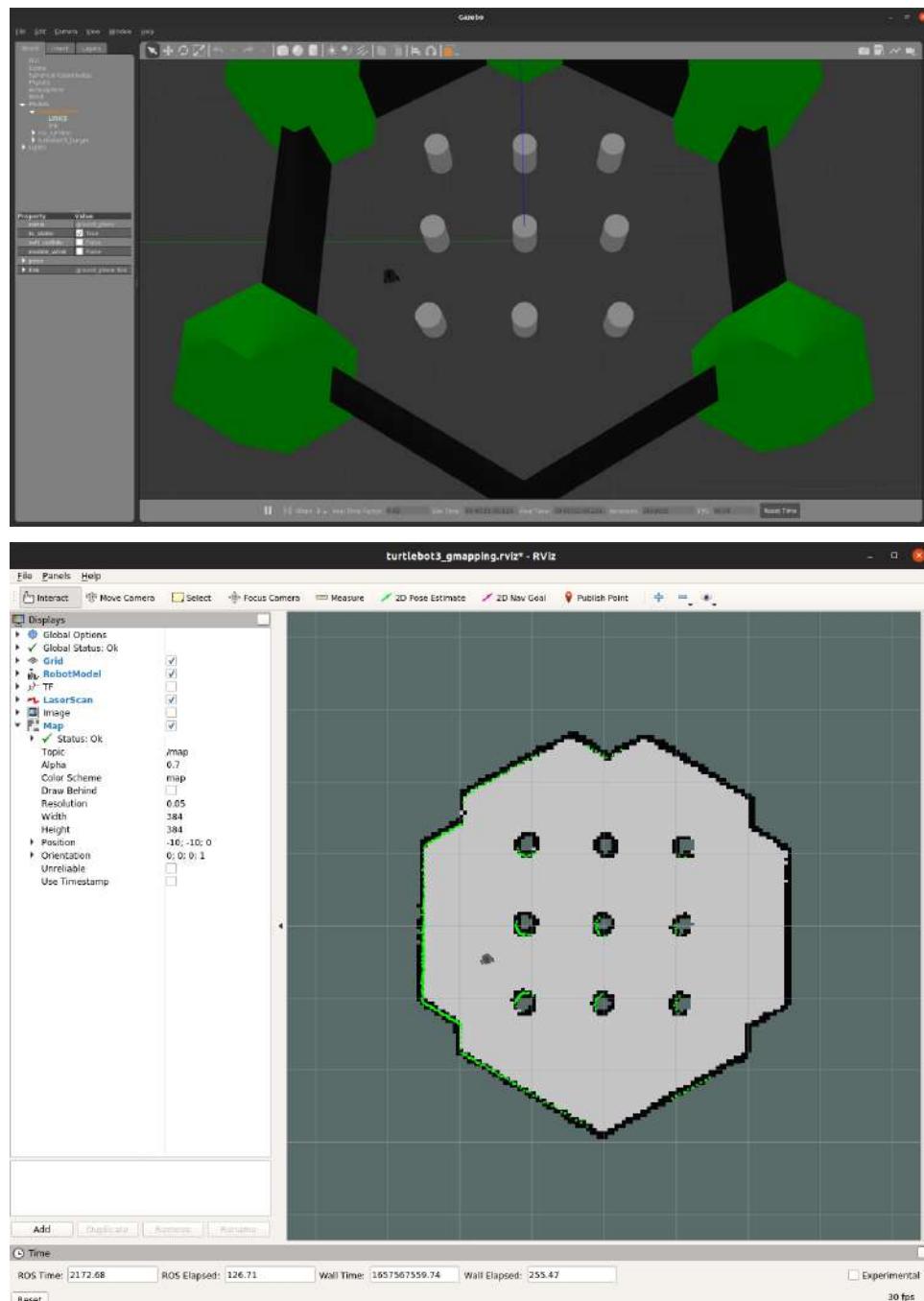
```

En otra terminal ejecutamos el siguiente comando para mover al Turtlebot con el objetivo de realizar el mapeo del entorno:

roslaunch turtlebot3_teleop turtlebot3_teleop_key.launch

Mientras nos movemos en Gazebo, en RViz se va pintando un mapa mejor (identificando los obstaculos).

A la hora del mapeo mediante la teleoperacion del Turtlebot, para garantizar la calidad del mapa, debemos movernos lentamente e intentar mantener un objeto a la vista en todo momento.

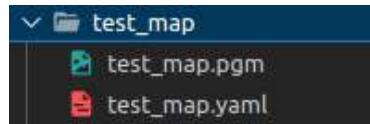


Despues de movernos por un tiempo, habremos creado un gran mapa. Este mapa sera muy util cuando queramos navegar por el mundo.

Para guardar el mapa en alguna parte lo primero que haremos sera crear una carpeta llamada “test_map” en el directorio “\$ROS_WORKSPACE/src/practice_turtlebot3” y despues ejecutamos el siguiente comando:

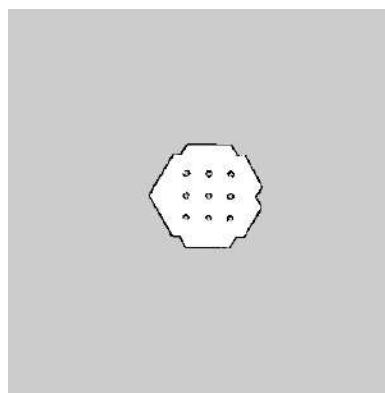
```
rosrun map_server map_saver -f  
$ROS_WORKSPACE/src/practice_turtlebot3/test_map/test_map
```

En el directorio “test_map” encontraremos dos archivos:



- *test_map.pgm*

Este archivo contiene una imagen similar a la que se visualiza en RViz.



- *test_map.yaml*

Este archivo apunta al anterior (al .pgm) y contiene informacion sobre la resolucion, el origen, etc.

Ahora lo que haremos sera cargar nuestro mapa e intentaremos que Turtlebot se localice correctamente.

Primero iniciamos Turtlebot en Gazebo como lo hemos hecho anteriores veces:

```
roslaunch turtlebot3_gazebo turtlebot3_world.launch &
```

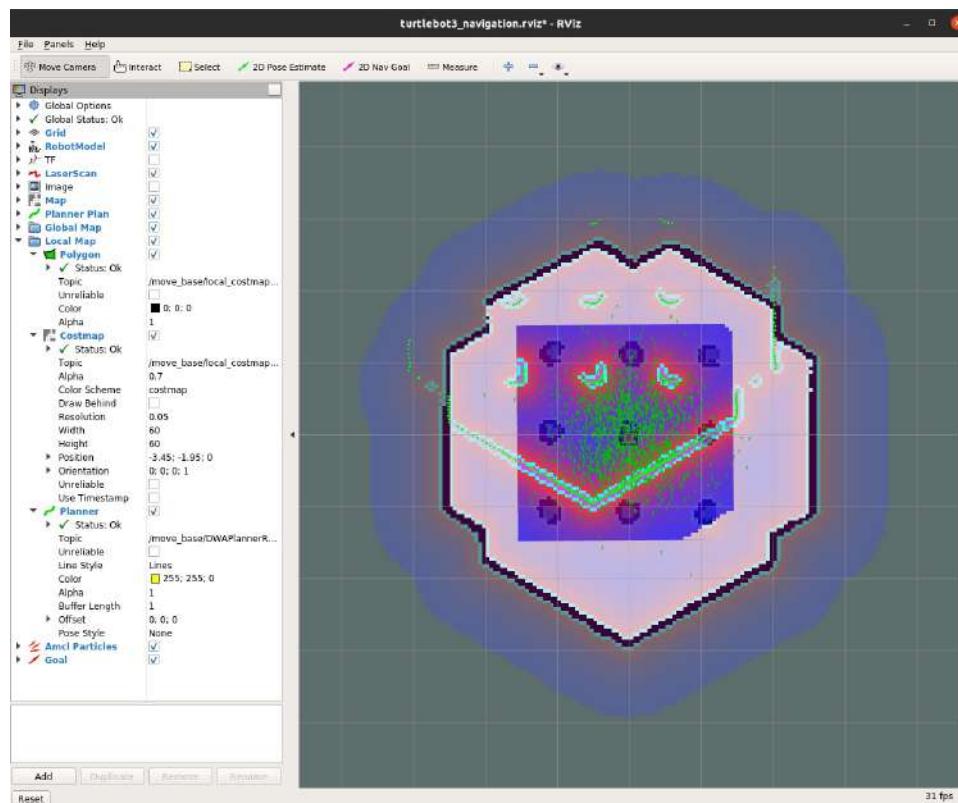
Agregamos permisos de ejecucion a los archivos generados por el mapeo ejecutando los siguientes comandos (para que puedan abrirlos otras aplicaciones):

```
cd $ROS_WORKSPACE/src/practice_turtlebot3/test_map  
chmod +x -R .
```

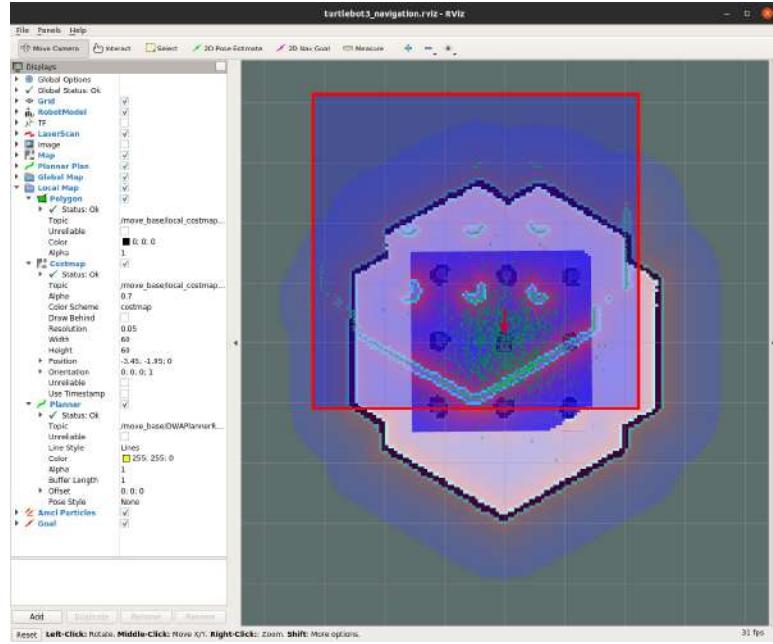
Para iniciar la navegacion ejecutamos el siguiente comando:

```
roslaunch turtlebot3_navigation turtlebot3_navigation.launch  
map_file:=$ROS_WORKSPACE/src/practice_turtlebot3/test_map/test  
_map.yaml
```

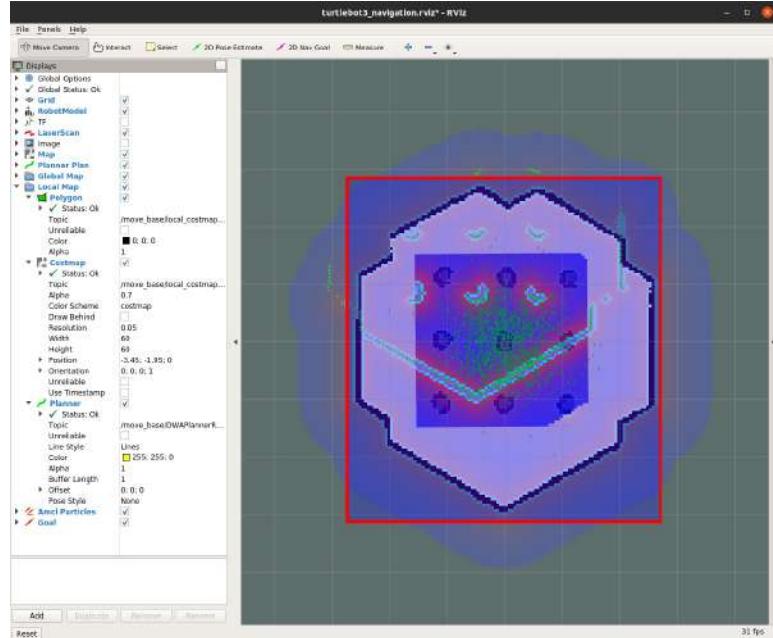
Este comando nos abrirá RViz con una visualización bastante extraña. El robot comenzará en algún lugar aleatorio en la vista del mapa en RViz.



Analicemos la imagen:



Como se observa, se muestra la posicion del Turtlebot con respecto al entorno de la simulacion, la cual se encuentra desfasada con la de la siguiente imagen (el cual es el mapa que se creo anteriormente, mediante el proceso de mapeo):

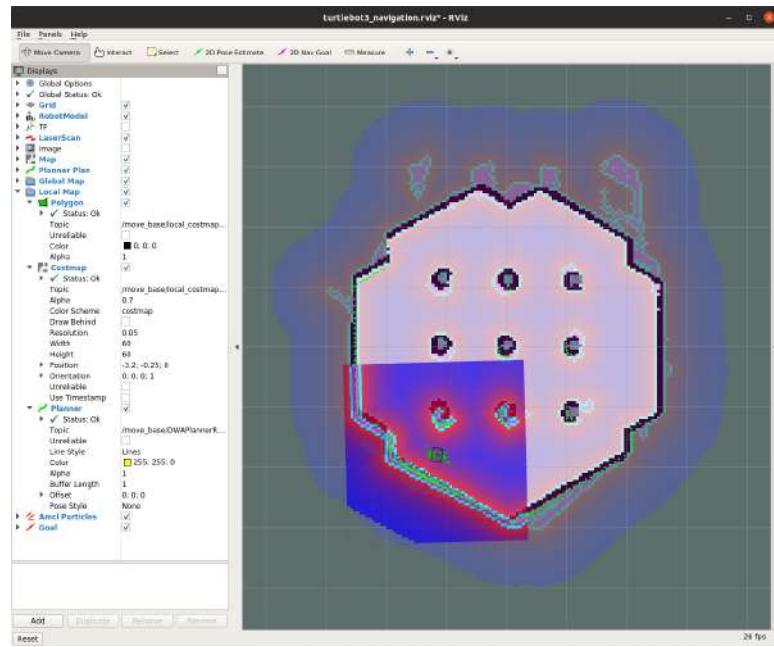


Lo que buscamos es que la ubicacion del Turtlebot coindica con la del mapa que creamos anteriormente.

Mientras vallamos teleoperando el Turtlebot por el mapa, el

Turtlebot se ira localizando mejor (ya el mapa estara mejor acomodado, hasta coincidir la visualizacion de RViz con la de Gazebo). Para esto ejecutamos el siguiente comando para teleoperar al Turtlebot:

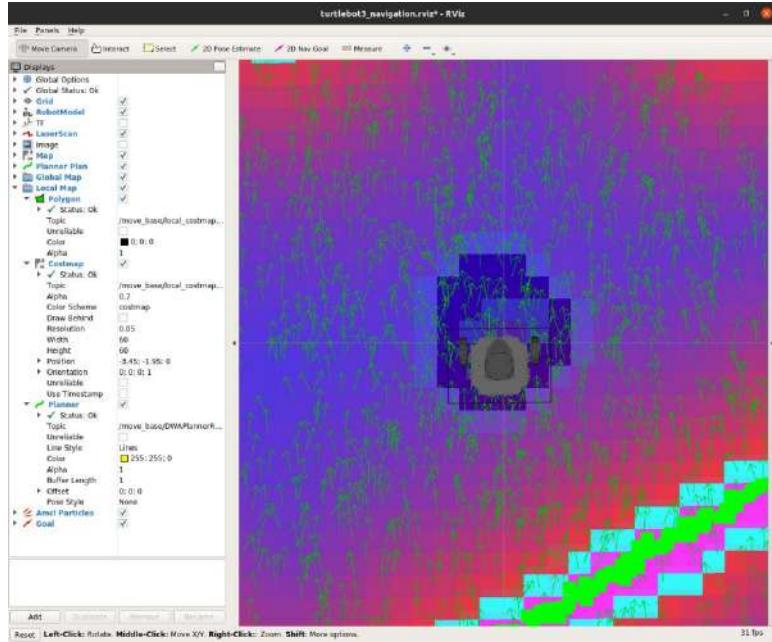
```
roslaunch turtlebot3_teleop turtlebot3_teleop_key.launch
```



La localizacion solo se puede obtener moviendo el robot primero y tomando medidas.

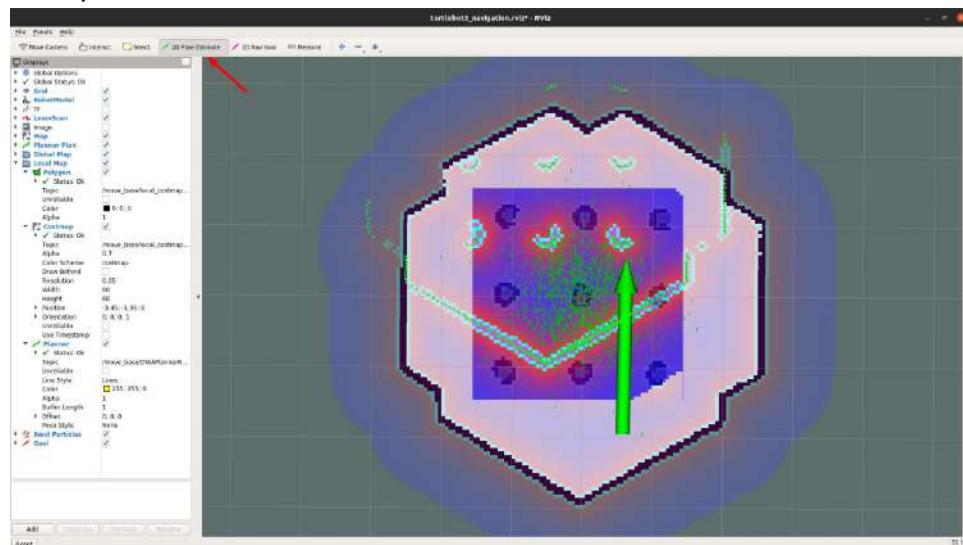
Tendremos un cuadrado de color púrpura con una cierta dimensión que sigue a nuestro Turtlebot, el cual indica el rango de las mediciones.

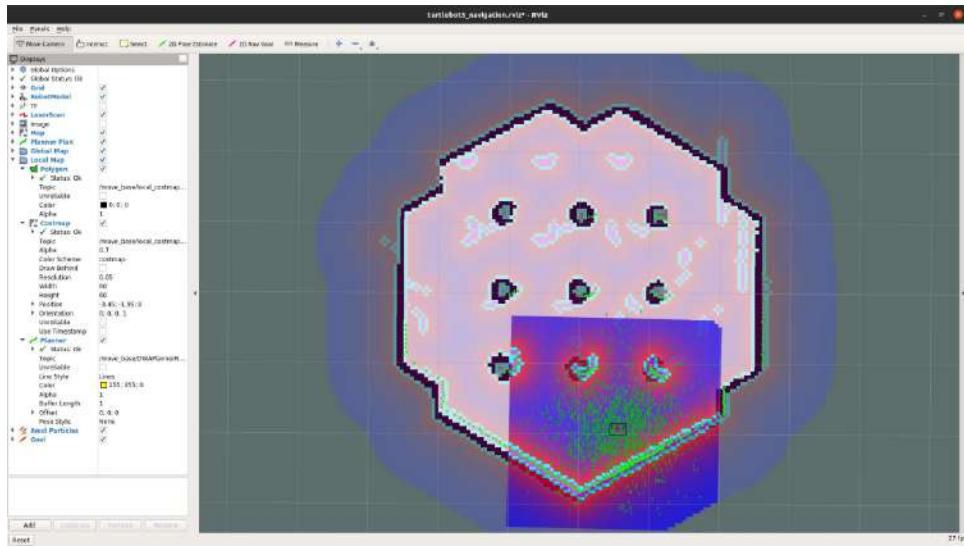
Si vemos la primera imagen que se mostro mas de cerca tendremos lo siguiente:



Las flechas verdes que rodean al Turtlebot son estimaciones probabilísticas de su posición real.

En RViz podemos darle al Turtlebot una pista de donde se encuentra haciendo clic en el botón “2D Pose Estimate” en RViz y dando clic en alguna parte del mapa y dando una orientación (automaticamente el Turtlebot aparecerá en esa ubicación):





Ahora solo es cuestion de teleoperar un poco el Turtlebot para que se localice mejor.

Pila de navegacion ROS

Para lograr la navegacion de nuestro Turtlebot utilizaremos la pila de navegacion ROS, la cual esta destinada a la navegacion bidimensional.

Entradas:

- Odometria
- Flujo de sensores
- Pose objetivo

Salidas:

- Comandos de velocidad seguros que se envian a la base movil

Limitaciones:

- Solo esta diseñada para robots de ruedas de accionamiento diferencial.
- Requieren un laser planar montado en la base.
- Se desarrollo en un robot cuadrado, por lo que puede haber dificultades con robots rectangulares grandes.

Esta pila de navegacion si es compatible con Turtlebot.

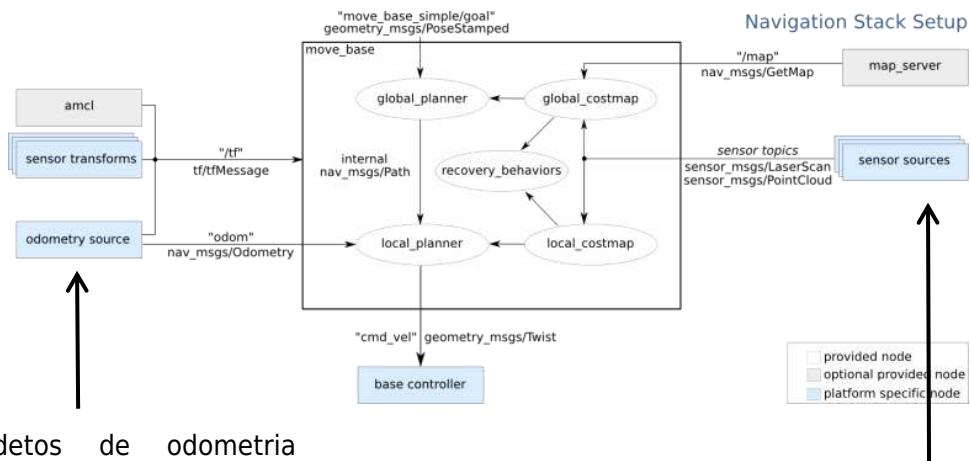
Dentro de la pila de navegacion hay un nodo bastante importante que es el nodo “move_base”, el cual:

- Proporciona una implementacion de una accion y maneja la

- comunicacion dentro de la pila.
- Utiliza un planificador global y local para lograr su objetivo.

Echemos un vistazo a la descripcion general de “move_base”:

El nodo “move_base” proporciona una interfaz ROS para configurar, ejecutar e interactuar con la pila de navegacion en un robot.



Usando datos de odometria podemos estimar donde esta el robot con respecto a su posicion inicial. Esta informacion debe publicarse en la pila de navegacion.

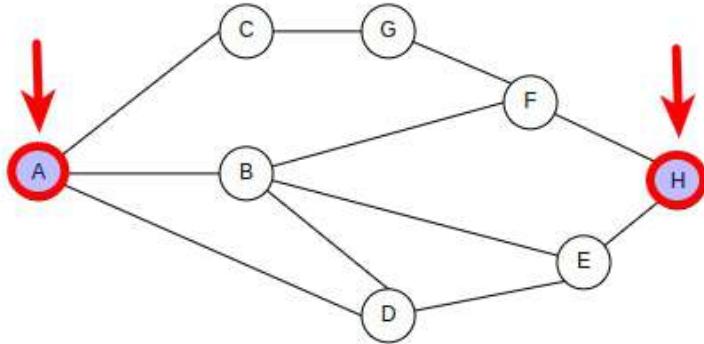
Los datos capturados por los diferentes sensores del robot deben referirse a un marco de referencia comun, que usualmente suele ser el enlace “base_link”.

Despues de que la pila de navegacion haya realizado sus calculos, los controladores base convertiran la salida de la pila en las correspondientes velocidades del motor del robot.

Los sensores se utilizan en la pila de navegacion para detectar obstaculos, tambien para la localizacion.

Planificacion de rutas

Podemos trabajar con grafos para situaciones como estas:



Nos encontramos en el punto A y queremos llegar al punto H.

Para llegar de un punto a otro podemos tener en consideracion:

- La distancia mas corta
- El tiempo mas corto (puede haber obstaculos)

Ahora es momento de poner en practica todo lo que hemos aprendido.

Hasta ahora hemos aprendido a crear un mapa y a localizar correctamente al robot en el entorno. Ahora haremos que el robot navegue hasta una posicion objetivo de forma autonoma.

NOTA: La estimacion de pose inicial debe realizarse antes de ejecutar la navegacion, lo cual implica que el Turtlebot tiene que estar ubicado correctamente en el mapa con los datos del sensor hasta que se supongan perfectamente con el mapa mostrado en RViz.

Lo que haremos sera lo siguiente:

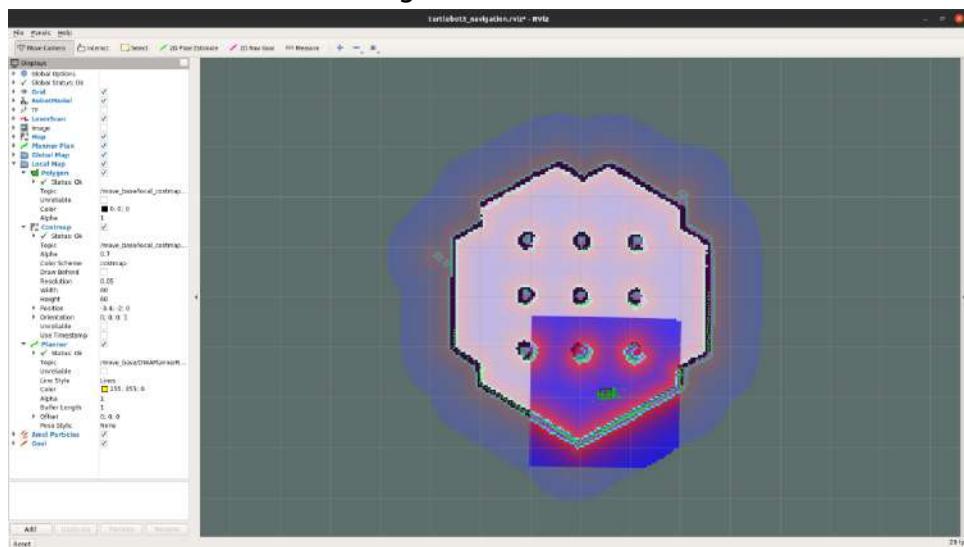
- Ejecutamos primeramente estos comandos:
`roslaunch turtlebot3_gazebo turtlebot3_world.launch &`
`roslaunch turtlebot3_navigation turtlebot3_navigation.launch`
`map_file:=$ROS_WORKSPACE/src/practice_turtlebot3/test_map/`
`test_map.yaml`
- Hacemos clic en el boton “2D Pose Estimate” en el menu de RViz.
- Hacemos clic en el mapa donde se encuentra el robot real y arrastramos la flecha verde grande hacia la direccion en la que mira el robot.
- Iniciamos el nodo de teleoperacion del teclado para ubicar con precision

el robot en el mapa.

Para esto ejecutamos el siguiente comando:

```
roslaunch turtlebot3_teleop turtlebot3_teleop_key.launch
```

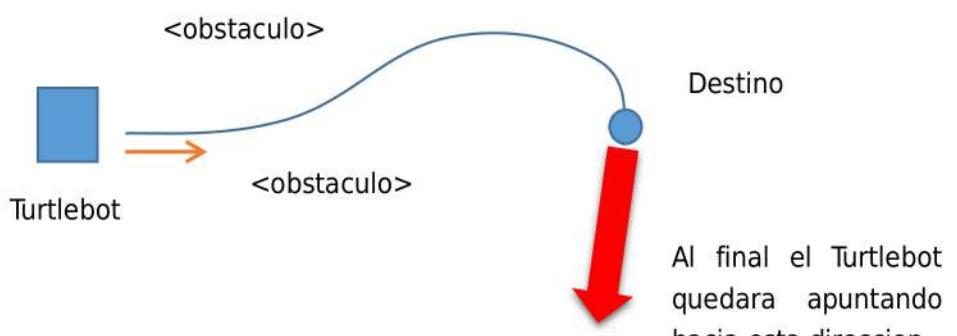
- Movemos el robot hacia adelante y hacia atras un poco para recopilar la informacion del entorno y reducir la ubicacion estimada del Turtlebot en el mapa que se muestra con pequeñas flechas verdes.
- Terminamos el nodo de teleoperacion.
NOTA: Si no se termina y se inicia la navegacion, el Turtlebot quedara como atascado.
- Finalmente tendremos lo siguiente:



Ahora, para establecer el objetivo de navegacion haremos lo siguiente:

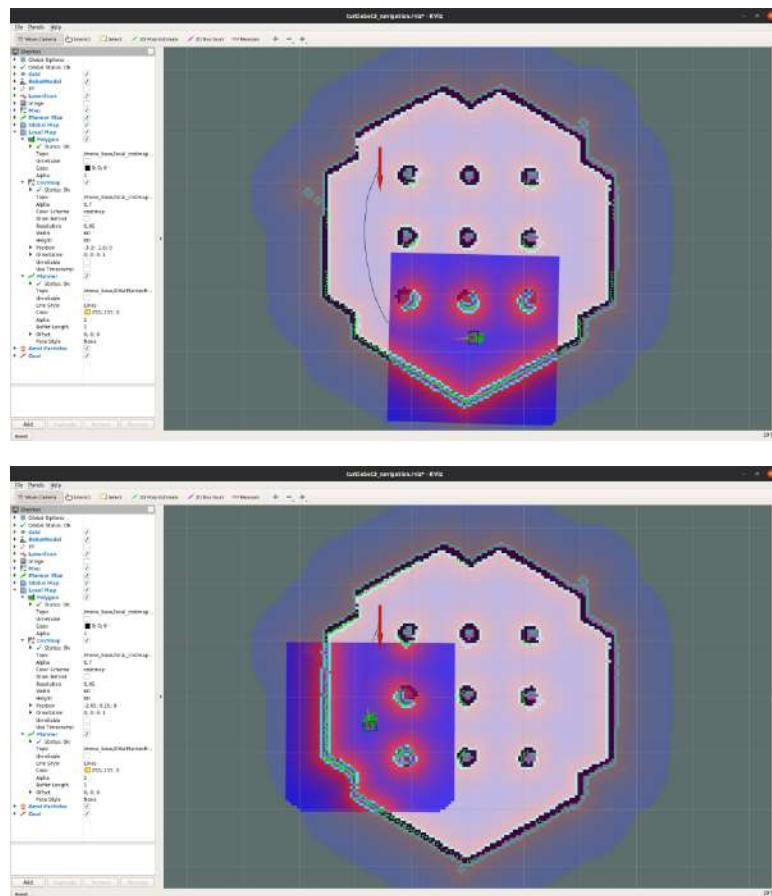
- Hacemos clic en el boton “2D Nav Goal” en el menu de RViz.
- Hacemos clic en el mapa para establecer el destino del robot y arrastramos la flecha verde hacia la direccion en la que se dirigira el robot.

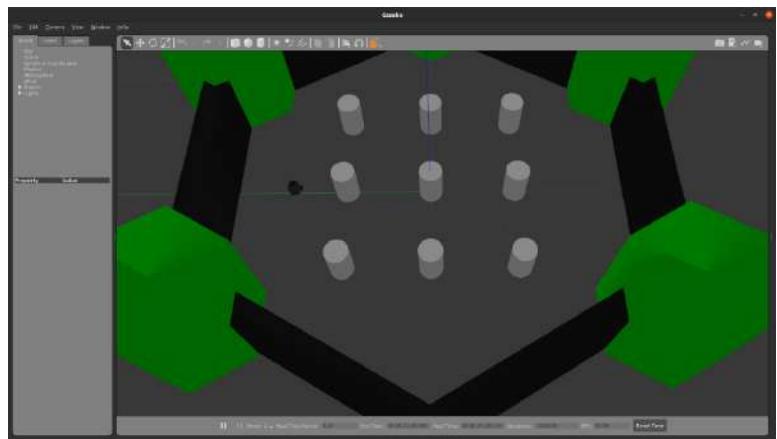
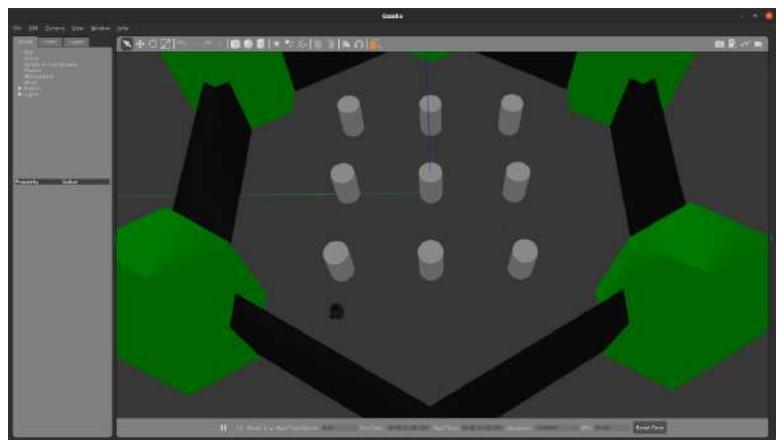
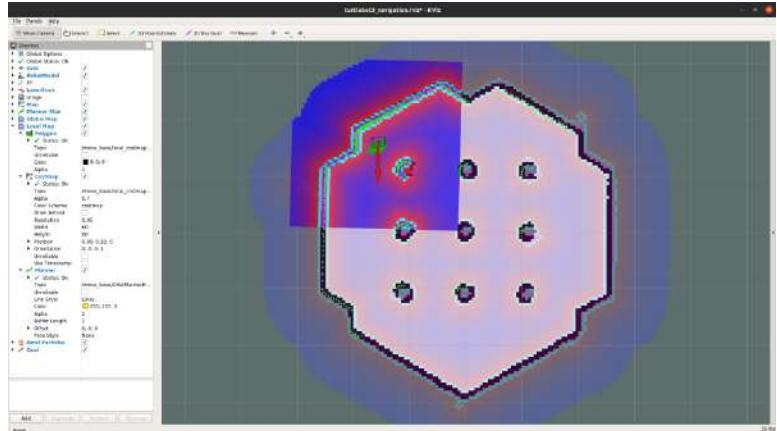
El resultado es algo como esto:

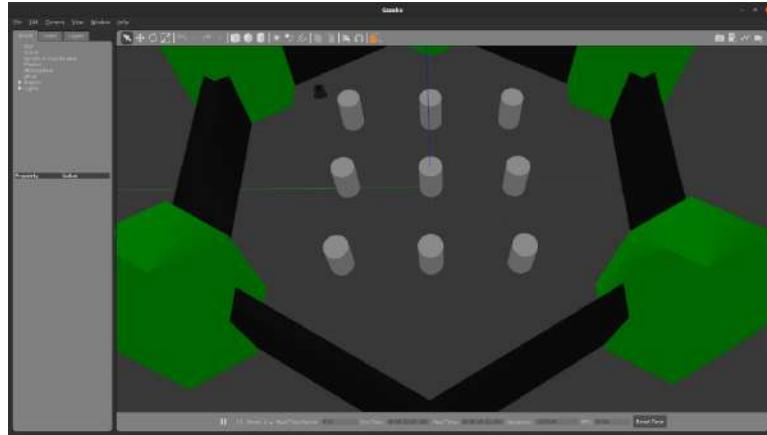


- El robot creara una ruta para llegar al objetivo de navegacion segun el planificador de ruta global. Luego, el robot se movera a lo largo del camino.
Si se coloca un obstaculo en el camino, la navegacion utilizara el planificador de ruta local para evitar el obstaculo.
- La configuracion de un objetivo de navegacion puede fallar si no se puede crear la ruta al objetivo de navegacion.
Si deseamos detener al robot antes de que alcance la posicion objetivo, establecemos la posicion actual del Turtlebot como objetivo de navegacion.

Los resultados son los siguientes:







Si quisieramos probar la funcionalidad de evitacion de obstaculos desconocidos, lo que hariamos es agregar un obstaculo adicional en Gazebo mientras el Turtlebot navega por el mapa.

Esto se puede hacer facilmente usando el menu superior de Gazebo.



La pila de navegacion ROS proporciona una implementacion del planificador local DWA para planificar alrededor de obstaculos desconocidos y dinamicos, es decir, el camino se puede modificar dinamicamente cuando aparecen obstaculos previamente desconocidos.

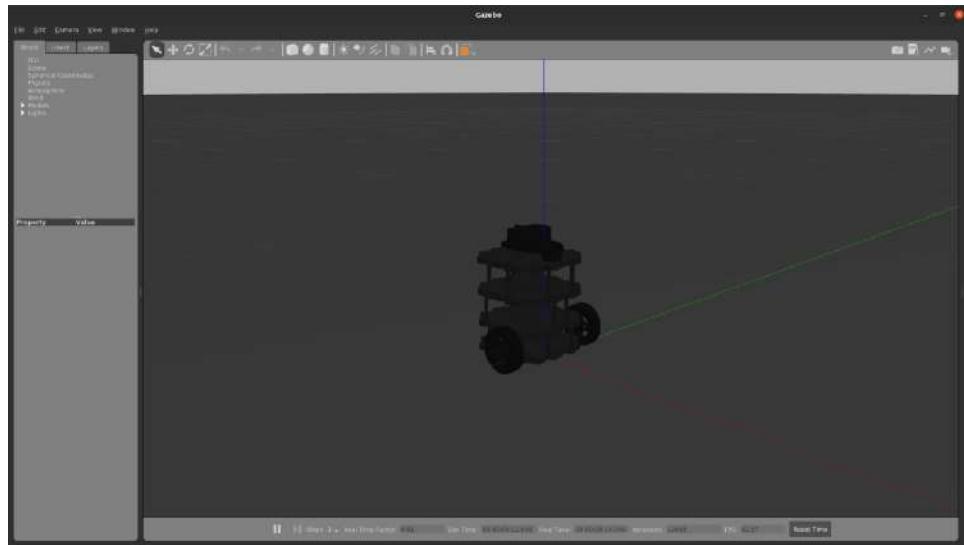
Cabe destacar que es posible mandar objetivos de navegacion al Turtlebot usando un nodo ROS que implemente una interfaz de cliente de accion simple para el servidor de accion “move_base”.

De hecho, es el servidor de acciones “move_base” el que hasta ahora ha estado sirviendo a todos nuestros objetivos de navegacion que enviamos a traves de RViz.

Otros mundos del TurtleBot

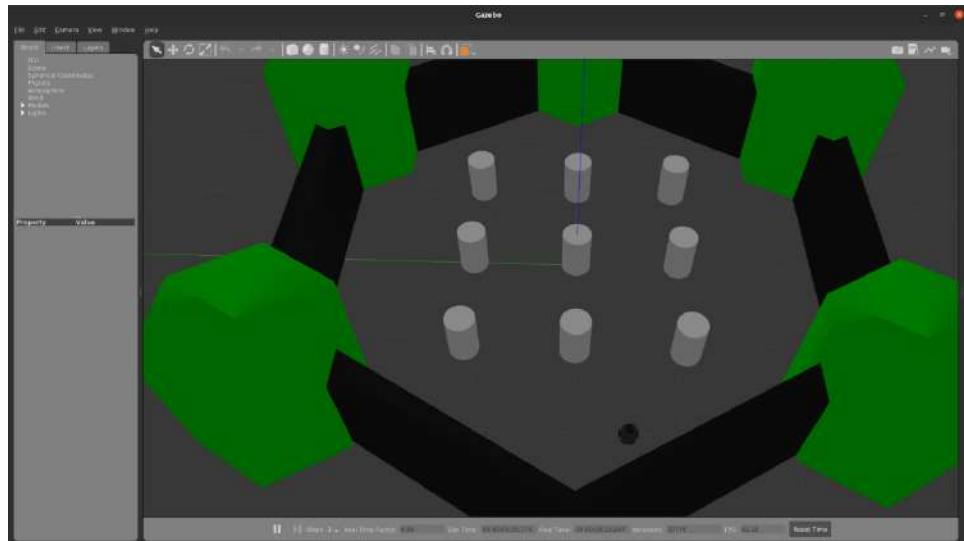
Contamos con tres mundos que podemos lanzar desde el paquete “turtlebot3_gazebo”:

- `roslaunch turtlebot3_gazebo turtlebot3_empty_world.launch`



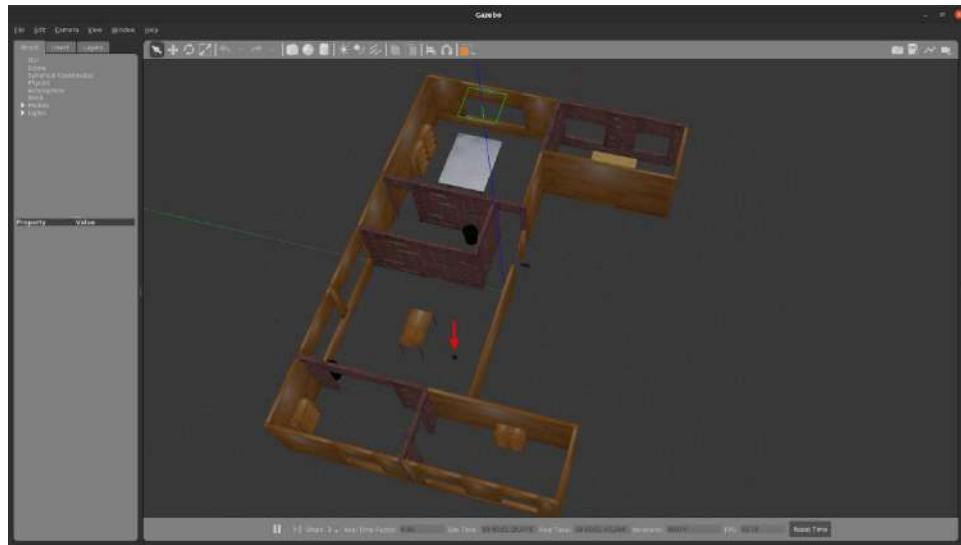
Este es un mundo vacío.

- `roslaunch turtlebot3_gazebo turtlebot3_world.launch`



Este es el mundo con el que hemos venido trabajando.

- `roslaunch turtlebot3_gazebo turtlebot3_house.launch`



Este mundo es un poco mas complejo y trata de simular el entorno de una casa.

Para cualquier mundo, despues de crear un mapa de algun mundo prediseñado mediante la teleoperacion del robot por el mundo (mediante el uso de sensores), y despues de localizar correctamente al robot en el entorno, ya estamos en condiciones de hacer que el robot navegue a una posicion objetivo de forma autonoma.

A grandes rasgos, haciamos lo siguiente:

- 1) Mapeado del mundo (teleoperabamos el robot por todo el mundo en Gazebo).
- 2) Guardar el mapa visualizado en RViz (como resultado del mapeado).
- 3) Cargar el mapa previamente guardado (se vizualiza en RViz).
- 4) Localizar correctamente el robot en el mapa haciendo uso del menu de RViz.
Que coindica la posicion del robot visualizado en RViz con la que tenemos en Gazebo.
- 5) Planificar una trayectoria haciendo uso del menu de RViz.

Material adicional

Esta es una lista de temas para leer en caso de que estemos interesados en la teoria detras de lo que hemos visto en este apartado:

NOTA: Yo siempre he apostado por entender las cosas antes de

implementarlas, ya que se vuelve mas facil la implementacion.

- SLAM
 - EKF SLAM
 - Graph Slam
 - Occupancy Grid SLAM
- Localizacion
 - Markov localization
 - Grid localization
 - Monte Carlo localization
- Planificacion de rutas
 - A* search
 - B* search
 - Probabilistic roadmap

Repaso y material utilizado

Repaso

Hasta el momento hemos realizado lo siguiente:

- Creacion de un espacio de trabajo ROS utilizando Docker.
- Modelado de robots utilizando URDF y Xacro.
Aqui se hablo sobre los enlaces, articulaciones, controladores, transmisiones, interfaces, macros (con respecto a Xacro), complementos, entre otros.
- Visualizacion de robots utilizando RViz.
- Simulacion de robots utilizando Gazebo.
- Integracion de RViz con Gazebo para visualizar lo que esta haciendo el robot en la simulacion.
- Uso del robot Turtlebot3 (Burger).
- Navegacion autonoma de un robot (y lo que implica toda la navegacion).

Se consulto mucha documentacion con respecto a ROS, Gazebo, RViz y diferentes sitios web para llevar a cabo las practicas realizadas. Tambien se tuvo de apoyo el curso llamado "*Hello (Real) World with ROS - Robot*

Operating System" que lo podemos encontrar en el siguiente sitio:

<https://www.edx.org/es/course/hello-real-world-with-ros-robot-operating-system>

Material

- Imagen Docker

La imagen ROS Docker como resultado de todo lo hecho, la podemos encontrar en el siguiente repositorio en Docker Hub:

https://hub.docker.com/r/chuy7/ros_hello_world/tags
(TAG: practice2)

Para descargar la imagen a nuestro repositorio local tendremos que ejecutar el siguiente comando en la terminal (recordando que esta imagen es el resultado de lo que se estuvo haciendo):

`docker pull chuy7/ros_hello_world:practice2`

Para crear un contenedor de la imagen anterior necesitamos utilizar el archivo “docker-compose.yaml” que se proporciona en el material. Para ello ejecutamos los siguientes comandos en la terminal (necesitamos estar ubicados donde se encuentra el archivo docker-compose):

`xhost +local:docker`

Con este comando le damos permisos a Docker de que pueda abrir ventanas desde el contenedor Docker.

`docker-compose run --rm --service-ports ros_hello_world /bin/bash`

- Archivos ROS

- Archivos ROS de la practica :

<https://github.com/ChuyFernandez/ROS-practice2>

Viene incluido:

- Archivos para la construccion de la imagen ROS Docker. Carpeta “**ros_docker**”.
 - Paquetes ROS creados (algunos no se incluyeron porque fueron instalados desde los paquetes Linux). Carpeta “**ros_packages**”.
 - Archivo “**docker-compose.yaml**”.

De igual manera todos los paquetes utilizados se pueden corroborar en la imagen Docker resultante.

CONCLUSION

A todos los scripts creados, tanto en C++ como en Python, se les ha colocado comentarios para una mayor comprensión para el lector.

Hasta aqui hemos cubierto gran parte de los conceptos relacionados a la robotica, asi como lo son la simulacion, la programacion de algoritmos, el modelado del robot, la navegacion autonoma del robot, entre otros; y temas un poco mas complicados como lo son el uso de controladores, transmisiones, sensores, entre muchos otros, que estan involucrados en la programacion del robot.

Realmente el control de un robot es bastante complejo y conlleva muchas matematicas, un buen dominio de la programacion (asi como de dominar algunos lenguajes en especifico de programacion), de programas relacionados a la robotica (como Gazebo, RViz), tener conocimientos de circuitos electricos, de fisica, entre otros.

A parte de la navegacion autonoma que se estuvo viendo, tambien es muy importante la manipulacion de un robot, en especial para los robots manipuladores, los cuales manejan ciertas herramientas que les sirven para realizar tareas especificas. Tambien tenemos la vision por computador, en la cual nos metemos un poco mas a lo que es la inteligencia artificial.

En resumidas cuentas, hay mucho que seguir aprendiendo.