

Modelado de robots con patas usando ROS 2 y Docker

Seykarin Mestre Muelas

Universidad Nacional de Colombia
Práctica Académica Especial
Eliana Isabel Arango
Alejandro Daniel Gomez

Julio, 2025

Contents

1	Configuración de ROS 2 en un Contenedor Docker	2
1.1	¿Qué es Docker?	2
1.2	¿Por qué usar Docker con ROS 2?	2
1.3	Paso 1: Instalación de Docker	3
1.4	Paso 2: Clonar el Repositorio	3
1.5	Paso 3: Abrir en Visual Studio Code (opcional)	3
1.6	Paso 4: Construir el Contenedor	3
2	Distribución de Carpetas del Proyecto	4
2.1	¿Qué significa estar dentro o fuera del contenedor Docker?	5
2.2	Ingresa al contenedor y comandos útiles	6
3	Construcción de proyecto	8
3.1	Creación del proyecto <code>leg_movement</code> con CMake	9
3.2	Modelado de hardware con URDF	10
3.3	Estructura básica de un archivo URDF	11
3.4	Visualización del modelo con un archivo de lanzamiento (Launcher)	15
3.5	Generación de un nodo para mover articulaciones	16
3.6	Compilación del nodo: modificación de <code>CMakeLists.txt</code> y <code>package.xml</code>	18
3.6.1	Configuración inicial de RViz y guardado de vista	21
3.6.2	Modificar el launcher para abrir una configuración específica en RViz	26
4	Uso de múltiples terminales dentro del contenedor Docker	28
5	Otros proyectos	29
5.1	Robot cuadrúpedo	29

July 26, 2025

Introducción

Este documento presenta una guía práctica para la instalación y configuración de ROS 2 (Robot Operating System versión 2) utilizando Docker. El objetivo es facilitar un entorno estandarizado y reproducible para el desarrollo de modelos robóticos articulados. La guía está dirigida especialmente a usuarios principiantes o con conocimientos básicos en robótica y programación.

Disclaimer

La instalación y configuración del contenedor Docker detallada en este documento está específicamente diseñada para el sistema operativo Ubuntu. Los pasos podrían variar o no funcionar correctamente en otros sistemas operativos.

1 Configuración de ROS 2 en un Contenedor Docker

1.1 ¿Qué es Docker?

Docker es una plataforma que permite crear, ejecutar y distribuir *contenedores*. Un contenedor es un entorno aislado que incluye todo lo necesario para ejecutar una aplicación: sistema operativo, librerías, herramientas y código. A diferencia de las máquinas virtuales, los contenedores comparten el mismo kernel del sistema, lo que los hace más livianos y rápidos.

1.2 ¿Por qué usar Docker con ROS 2?

ROS 2 puede ser complejo de instalar debido a sus múltiples dependencias. Docker permite:

- Estandarizar el entorno de desarrollo.
- Usar distintas versiones de ROS 2 sin conflictos.
- Evitar modificar o dañar la configuración del sistema principal.
- Facilitar el trabajo colaborativo y replicable en distintos equipos.

1.3 Paso 1: Instalación de Docker

A continuación se darán los pasos para utilizar docker y la configuración del contenedor, sin embargo de anima a revisar el repositorio de origen.

https://github.com/aldajo92/ROS2_Docker_UI

Abre la terminal y ejecuta los siguientes comandos en orden.

```
curl -fsSL https://get.docker.com -o get-docker.sh
```

Descarga un script oficial que automatiza la instalación de Docker.

```
sudo sh get-docker.sh
```

Ejecuta el script para instalar Docker.

```
sudo groupadd docker
```

Crea el grupo `docker`. Es normal si esta línea falla porque el grupo ya existe.

```
sudo usermod -aG docker $USER
```

Agrega el usuario actual al grupo `docker` para ejecutar comandos sin `sudo`.

```
newgrp docker
```

Aplica los cambios de grupo sin necesidad de cerrar sesión.

```
sudo systemctl restart docker.service
```

Reinicia el servicio Docker.

```
docker run hello-world
```

Verifica que Docker esté correctamente instalado.

1.4 Paso 2: Clonar el Repositorio

```
cd ~  
git clone https://github.com/aldajo92/ROS2_Docker_UI.git
```

Clona el repositorio que contiene la configuración del contenedor con ROS 2.

1.5 Paso 3: Abrir en Visual Studio Code (opcional)

```
code ~/ROS2_Docker_UI
```

Abre el proyecto en VS Code. Es recomendable instalar las extensiones `Docker` y `Remote - Containers`.

1.6 Paso 4: Construir el Contenedor

```
cd ~/ROS2_Docker_UI  
./scripts/build.sh
```

Construye el contenedor. Este script configura el entorno base de Ubuntu, instala ROS 2 y herramientas útiles. La primera vez puede tardar varios minutos.

¿Con qué frecuencia se debe ejecutar `./scripts/build.sh`?

El comando `./scripts/build.sh` solo debe ejecutarse una vez, durante la primera configuración del entorno. Este script construye la imagen del contenedor de Docker utilizando un archivo especial llamado **Dockerfile**.

beginitemize

Dockerfile: Es un archivo de texto que contiene una serie de instrucciones que Docker sigue para construir una imagen. Entre estas instrucciones se encuentran:

- El sistema operativo base (por ejemplo, Ubuntu).
- Comandos para instalar ROS 2 y otras dependencias.
- Configuraciones de entorno necesarias.
- Archivos o scripts que deben copiarse al contenedor.

Imagen de Docker: Es el resultado de procesar un Dockerfile. Representa un entorno completo e inmutable que incluye todo lo necesario para ejecutar una aplicación. Las imágenes pueden ser replicadas en distintos sistemas sin importar su configuración local, lo cual garantiza que el entorno de ROS 2 sea idéntico en cualquier equipo donde se ejecute.

Mientras no se modifique el **Dockerfile**, no es necesario volver a ejecutar `./scripts/build.sh`. Si el archivo cambia (por ejemplo, para instalar nuevas herramientas o cambiar la versión de ROS 2), entonces sí se debe volver a ejecutar el script para que los cambios se reflejen en una nueva imagen del contenedor.

2 Distribución de Carpetas del Proyecto

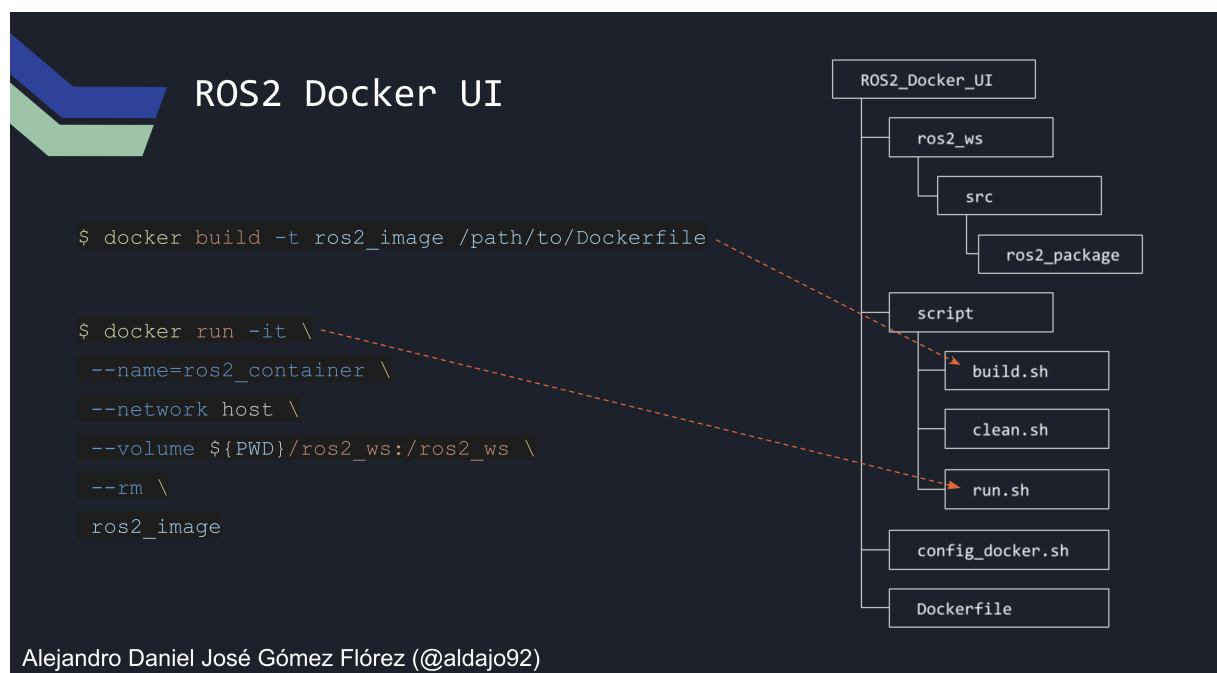


Figure 1: Estructura de las carpetas de docker

Esta imagen representa la organizacion de carpetas del proyecto `ROS2_Docker_UI`, estas separan claramente el código fuente, los scripts de utilidad y los archivos de configuración. A continuación, se detalla su estructura jerárquica ignorando totalmente las líneas código presentes en la imagen:

El proyecto `ROS2_Docker_UI` está organizado en varias carpetas que separan claramente el código fuente, los scripts de utilidad y los archivos de configuración. A continuación, se detalla su estructura jerárquica:

- **ROS2_Docker_UI:** Carpeta raíz del proyecto.
 - **ros2_ws/** (*ROS 2 Workspace*): Contiene el espacio de trabajo donde se desarrollan y compilan los paquetes de ROS 2.
 - * **src/**: Carpeta donde se ubican los proyectos que se van a trabajar dentro del contenedor. Aquí se alojan los paquetes ROS 2 personalizados que serán construidos y ejecutados.
 - **ros2_package/**: Ejemplo de un paquete ROS 2 desarrollado por el usuario.
 - **script/**: Incluye scripts de automatización que simplifican la interacción con Docker y la construcción del entorno.
 - * **build.sh**: Script para construir la imagen de Docker.
 - * **clean.sh**: Script para limpiar recursos y contenedores creados.
 - * **run.sh**: Script que ejecuta el contenedor Docker.
 - **config_docker.sh**: Archivo de configuración para definir variables y parámetros usados por los scripts.
 - **Dockerfile**: Archivo que contiene las instrucciones necesarias para construir la imagen de Docker personalizada con ROS 2.

Esta estructura permite una separación clara entre el código fuente del proyecto, los scripts de gestión del contenedor, y la configuración de la imagen. Facilita tanto el mantenimiento como la comprensión del entorno de desarrollo basado en Docker. Este documento se centrará solamente en el workspace y ligeramente en los scripts. La configuración de docker no será objeto de interés.

2.1 ¿Qué significa estar dentro o fuera del contenedor Docker?

Al trabajar con Docker, es fundamental entender la diferencia entre estar **dentro del contenedor** y **fuera del contenedor**.

- **Fuera del contenedor (host):** Es el sistema operativo de la máquina real del usuario, donde se encuentran los archivos del proyecto. Aquí se pueden editar scripts, modificar el código fuente y construir la imagen del contenedor.

- **Dentro del contenedor:** Es un entorno aislado que se crea a partir de la imagen definida en el `Dockerfile`. Dentro de este entorno se realizan tareas como la compilación del código, la ejecución de paquetes ROS 2 o el lanzamiento de nodos. Es como tener una computadora separada, con sus propias configuraciones y herramientas.

La carpeta `ros2_ws` como puente entre ambos mundos:

A diferencia de las demás carpetas, la carpeta `ros2_ws` es accesible tanto desde el host como desde dentro del contenedor. Esto permite que los cambios realizados en el código desde fuera del contenedor (por ejemplo, al editar con un editor de texto) se reflejen inmediatamente dentro del entorno de desarrollo en Docker, y viceversa. Es decir, se comporta como una carpeta compartida entre ambos entornos.

Esta característica facilita el flujo de trabajo, ya que no es necesario reconstruir el contenedor o copiar archivos cada vez que se edita el código fuente del proyecto.

2.2 Ingresar al contenedor y comandos útiles

Una vez se ha construido la imagen del contenedor utilizando el `Dockerfile`, es posible ejecutar e ingresar al entorno ROS 2 que está contenido dentro de esa imagen. Esta sección explica cómo hacerlo y qué comandos son relevantes en este proceso.

¿Cómo ingresar al contenedor?

Para ingresar inicialmente al contenedor, se debe ejecutar el siguiente comando en la terminal:

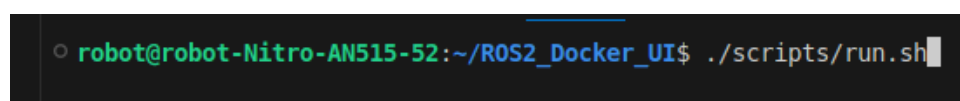
```
./scripts/run.sh
```

Este comando se encuentra dentro de la carpeta `scripts`, ubicada en el directorio raíz del proyecto. Es importante asegurarse de estar ubicado dentro de la carpeta base `~/ROS2_Docker_UI` antes de ejecutar el comando. El punto `(./)` indica que se desea ejecutar un archivo ubicado en una ruta relativa a la carpeta actual desde donde se ejecuta el comando. En este caso, se ejecuta el archivo `run.sh` que está dentro de la carpeta `scripts`. El archivo `run.sh` contiene las instrucciones necesarias para lanzar el contenedor con ROS 2, asignar el workspace compartido y abrir una terminal interactiva dentro del contenedor.

A continuación, se muestra un ejemplo del uso de este comando desde la terminal. En la imagen:

- El texto en **verde** representa el nombre del usuario.
- El texto en **azul** representa la ruta actual dentro del sistema de archivos.

Ejemplo visual:



```
robot@robot-Nitro-AN515-52:~/ROS2_Docker_UI$ ./scripts/run.sh
```

Figure 2: Ejemplo del comando `./scripts/run.sh` ejecutado desde la terminal

Comandos comunes luego de estar dentro del contenedor:

- `source /opt/ros/humble/setup.bash`
Este comando configura las variables de entorno necesarias para usar ROS 2 Humble dentro del contenedor. Es fundamental ejecutarlo si aún no se ha hecho, especialmente antes de compilar o ejecutar paquetes ROS 2.
- `colcon build`
Compila los paquetes de ROS 2 **dentro** del workspace. Este comando debe ejecutarse cada vez que se realicen cambios importantes en el código.
- `source install/setup.bash`
Una vez compilados los paquetes, este comando carga las configuraciones necesarias para que ROS 2 pueda encontrarlos y ejecutarlos correctamente.
- `ros2 run <paquete> <nodo>`
Ejecuta un nodo específico de un paquete.
- `ros2 launch <paquete> <archivo_launch.py>`
Lanza un conjunto de nodos definido en un archivo de lanzamiento.
Estos dos últimos se verán con más detalle más adelante.

Estos dos últimos comandos están escritos de forma genérica, por lo que se profundizarán más adelante. **Importante:** Antes de comenzar a trabajar con ROS 2 dentro del contenedor, es indispensable ejecutar los tres primeros comandos en el orden indicado (dentro del contenedor):

1. `source /opt/ros/humble/setup.bash`
2. `colcon build`
3. `source install/setup.bash`

De lo contrario, el contenedor no funcionará correctamente y no se podrán ejecutar los comandos de ROS 2.

Atajo

El contenedor dado en este documento tiene un comando que ejecuta los anteriores comandos a la vez, pero aún así es importante tener presente la existencia de estos. El atajo es

<code>brs2</code>

Salir del contenedor

Para salir del entorno del contenedor se puede usar el comando `exit`, lo cual devuelve al usuario al sistema operativo original (host) o también se puede usar el comando de teclas `CTRL + D`.

3 Construcción de proyecto

Para entender de forma práctica el funcionamiento de ROS 2 dentro del contenedor Docker, se desarrollarán diferentes proyectos desde cero. Cada uno de estos proyectos se ubicará dentro del workspace compartido con el host, y se construirá usando herramientas estándar de ROS 2 como `colcon`. El desarrollo puede hacerse en C++ (usando `CMake`) o en Python (usando `ament_python`). En este documento nos centraremos inicialmente en C++ con `ament_cmake` por ser común en sistemas embebidos y de control robótico.

El primer proyecto que se creará se llamará `leg_movement`. Este proyecto permitirá familiarizarse con la estructura base de un paquete ROS 2, su construcción y compilación dentro del contenedor.

Antes de comenzar es útil que conozcas dos conceptos clave:

Nodos y Topics

Este proyecto utiliza **ROS 2** para modelar y visualizar un robot articulado con forma de pie humano. ROS 2 permite dividir las funciones del sistema en programas independientes llamados **nodos**, que se comunican mediante **tópicos**.

- **Nodos:** Son unidades de ejecución que cumplen tareas específicas. Por ejemplo:
 - Un nodo publica el estado de las articulaciones (`joint_state_publisher`).
 - Otro calcula las posiciones del robot (`robot_state_publisher`).
 - Otro permite visualizarlo en 3D (`rviz`).
- **Tópicos:** Son canales de comunicación entre nodos. Un nodo publica datos en un tópico y otro puede suscribirse para recibirlos. Por ejemplo:
 - El tópico `/joint_states` transmite las posiciones articulares.
 - RViz usa esta información para mostrar el movimiento del robot.

Con estos elementos, se puede simular y analizar el comportamiento del robot de forma modular y visual.

Ubicación del proyecto

Todos los proyectos deben ser creados dentro del directorio:

```
~/ROS2_Docker_UI/ros2_ws/src/
```

Este es el directorio fuente del workspace ROS 2 y actúa como puente entre el sistema anfitrión (host) y el contenedor Docker. Los cambios realizados aquí desde el exterior serán visibles dentro del contenedor, y viceversa, por lo que al entrar al contenedor debemos asegurarnos de estar en la carpeta **src**.

3.1 Creación del proyecto `leg_movement` con CMake

1. Ingrese al contenedor con el script de ejecución:

```
./scripts/run.sh
```

2. Navegue a la carpeta de proyectos dentro del workspace:

```
cd ~/ros2_ws/src
```

3. Cree el paquete `leg_movement` utilizando el generador de paquetes de ROS 2:

```
ros2 pkg create --build-type ament_cmake leg_movement --dependencies rclcpp std_msgs
```

Este comando generará una estructura base para un paquete ROS 2 en C++, incluyendo archivos como `CMakeLists.txt`, `package.xml`, y carpetas como `src/`.

Observe la estructura de carpetas mostrada en la figura 1 y verá que en la carpeta `src` ahora existe un proyecto llamado `leg_movement` que contiene las carpetas mencionadas anteriormente.

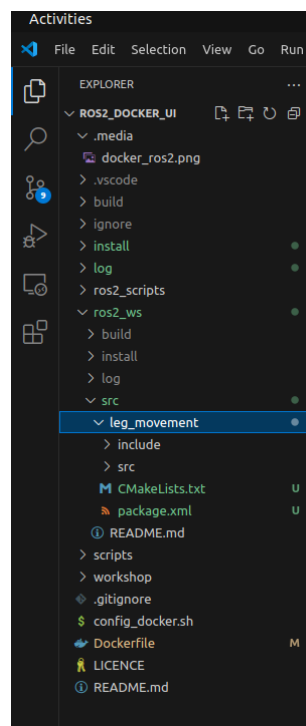


Figure 3: Estructura de carpetas luego de crear el primer paquete.

4. Regrese a la raíz del workspace:

```
cd ~/ros2_ws
```

5. Compile el workspace completo (incluyendo `leg_movement`):

```
colcon build
```

6. Fuente el entorno para poder ejecutar el nuevo paquete:

```
source install/setup.bash
```

Este flujo de trabajo se repetirá para cada nuevo proyecto que se desee construir dentro del contenedor. En las secciones siguientes, se trabajará con el paquete `leg_movement` para implementar funciones específicas relacionadas con el modelado y simulación de movimientos de piernas robóticas.

3.2 Modelado de hardware con URDF

Para representar y modelar el hardware de los proyectos robóticos en ROS 2, se utiliza un lenguaje llamado URDF (Unified Robot Description Format). Este lenguaje permite describir la estructura física del robot, incluyendo sus links (partes rígidas), joints (articulaciones), geometría, dimensiones y relaciones espaciales.

Ubicación del proyecto:

- Desde la carpeta base del contenedor `ROS2_Docker_UI`
- Ingresar a la carpeta `ros2_ws/src/`
- Ingresar al proyecto `leg_movement`
- Dentro de `leg_movement`, crear manualmente una carpeta llamada `urdf` o utilizar el siguiente código:

```
mkdir ~/ros2_ws/src/leg_movement/urdf
```

- Dentro de la carpeta `urdf`, crear un archivo llamado `pie.urdf`. Es importante agregar el tipo de archivo al final del nombre.

La carpeta `urdf/` contendrá todos los archivos relacionados con la descripción estructural del robot. Esta organización mejora la claridad y facilita la integración con archivos de lanzamiento y otros recursos.

En este caso, el modelo desarrollado representa una versión simplificada de un pie humano, compuesto por segmentos conectados mediante articulaciones. El diseño busca simular una cadena cinemática con varios grados de libertad que emule la flexión del tobillo, rodilla y cadera.

¿Qué es URDF?

URDF (Unified Robot Description Format) es un formato basado en XML que se utiliza en ROS para describir los componentes físicos de un robot. Permite definir:

- **Links (enlaces):** Representan las partes rígidas del robot (como el talón, el metatarso y los femur).
- **Joints (articulaciones):** Definen cómo se conectan los enlaces y cómo pueden moverse entre sí.

- **Geometría:** Formas básicas como cajas, cilindros o mallas para representar la forma física del robot.
- **Materiales:** Colores o texturas usadas para visualizar los modelos.
- **Transformaciones:** Posiciones y orientaciones relativas entre enlaces y articulaciones.

Esta descripción es fundamental para poder simular, visualizar y controlar el robot en herramientas como RViz y Gazebo.

3.3 Estructura básica de un archivo URDF

El archivo principal del modelo, llamado `pie.urdf`, se encuentra dentro de la carpeta `urdf/` del proyecto `leg_movement`:

```
ros2_ws/src/leg_movement/urdf/pie.urdf
```

A continuación se muestra la estructura general de un archivo URDF:

```
<?xml version="1.0"?>
<robot name="pie_humano_simplificado">
  ...
</robot>
```

Dentro del bloque `<robot>` se definen los elementos del modelo: materiales, enlaces (links) y articulaciones (joints) como se observa a continuación.

Fragmento de ejemplo: definición de materiales y base

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

<link name="base_link">
  <visual>
    <geometry><box size="0.1 0.1 1"/></geometry>
    <origin xyz="0 0 0.5" rpy="0 0 0"/>
    <material name="red"/>
  </visual>
</link>
```

Este fragmento define los primeros elementos del modelo de pierna:

- `<material>` define colores reutilizables. Por ejemplo, el color rojo (`red`) con valores RGBA (1 0 0 1) se usará para distinguir visualmente el segmento base.
- `<link name="base_link">` define un enlace rígido fijo que actúa como el punto de origen del modelo (puede considerarse como la unión de la pierna con la pelvis (no está modelada en este proyecto) o como la base de soporte en una simulación).
- Dentro de este enlace:

- `<geometry><box>` crea una forma cúbica alargada ($0.1 \times 0.1 \times 1$), que se puede interpretar como un soporte visual o estructural vertical.
- `<origin xyz="0 0 0.5">` posiciona el centro visual del cubo, levantándolo hacia arriba para que se asiente desde el suelo ($Z = 0$) hasta la altura $Z = 1$. Esto se debe a que los elementos tienden a ubicar su origen (el suelo) en el centro de la estructura, por lo que en este caso el origen en el eje z , de un cubo de 1, se encuentra en 0.5 y así al elevarlo 0.5, la parte mas alta del cubo queda 1 por encima del suelo.
- `<material name="red">` aplica el color definido previamente para visualizar este bloque en RViz.

Este primer bloque no representa directamente el fémur, pero sirve como punto de anclaje o referencia para la cadena cinemática del modelo de pierna.

Estructura de articulaciones

Las articulaciones conectan los diferentes segmentos del modelo: fémur, tibia y pie. A continuación se muestra un ejemplo de cómo se define una articulación entre dos enlaces:

```
<joint name="joint_1" type="revolute">
  <parent link="femur"/>
  <child link="tibia"/>
  <origin xyz="0 0 0" rpy="0 0 0"/>
  <axis xyz="1 0 0"/>
  <limit lower="-1.5" upper="1.5" effort="10" velocity="1"/>
</joint>
```

- `name="joint_1"`: nombre único para esta articulación.
- `type="revolute"`: permite rotación entre los enlaces. En este caso, representa la articulación de la rodilla.
- `parent="femur"` y `child="tibia"`: conectan los dos enlaces (el fémur como la sección dominante, y la tibia como la sección hija).
- `origin xyz="..."`: define la posición relativa de la articulación (en este caso, sin desplazamiento).
- `axis="1 0 0"`: la rotación ocurre alrededor del eje X (el eje de flexión).
- `limit`: restringe el rango de movimiento de la articulación (en radianes), además de definir el esfuerzo y velocidad máxima.

Para entender mejor la función de cada sección, se invita a jugar con los parámetros y observar su funcionamiento. Adicionalmente se recomienda el siguiente video [2]

Descripción general del modelo

El modelo completo de la pierna humana simplificada está compuesto por los siguientes elementos:

- **base_link**: estructura de soporte o referencia, desde la cual se monta la pierna.
- **femur**: primer segmento móvil, representa el muslo.
- **tibia**: segundo segmento, unido al fémur mediante la articulación de la rodilla.
- **pie**: segmento final, unido a la tibia mediante la articulación del tobillo.

Las articulaciones correspondientes serían:

- **joint_hip**: fija el fémur al **base_link**. Puede ser fija o rotacional, dependiendo del caso.
- **joint_knee**: articulación tipo **revolute** entre fémur y tibia.
- **joint_ankle**: articulación tipo **revolute** entre tibia y pie.

Este diseño permite simular una pierna funcional con dos grados de libertad principales: flexión en la rodilla y flexión en el tobillo. Es ideal para pruebas de locomoción, control de postura o entrenamiento de algoritmos de marcha en robots bípedos.

El código completo se encuentra a continuación:

```
<?xml version="1.0"?>
<robot name="pierna_humana_simplificada">

  <!-- Materiales -->
  <material name="red"><color rgba="1 0 0 1"/></material>
  <material name="green"><color rgba="0 1 0 1"/></material>
  <material name="blue"><color rgba="0 0 1 1"/></material>
  <material name="yellow"><color rgba="1 1 0 1"/></material>

  <!-- Base Link -->
  <link name="base_link">
    <visual>
      <geometry><box size="0.1 0.1 1"/></geometry>
      <origin xyz="0 0 0.5" rpy="0 0 0"/>
      <material name="red"/>
    </visual>
  </link>

  <!-- Link 0 (esttico) -->
  <link name="link_0">
    <visual>
      <geometry><box size="0.005 0.005 0.005"/></geometry>
      <origin xyz="0 0 0.0025" rpy="0 0 0"/>
      <material name="blue"/>
    </visual>
  </link>
```

```

<!-- Fmur -->
<link name="femur">
  <visual>
    <geometry><box size="0.1 0.1 0.3"/></geometry>
    <origin xyz="0 0 0.15" rpy="0 0 0"/>
    <material name="green"/>
  </visual>
</link>

<!-- Tibia -->
<link name="tibia">
  <visual>
    <geometry><box size="0.1 0.1 0.3"/></geometry>
    <origin xyz="0 0 0.15" rpy="0 0 0"/>
    <material name="blue"/>
  </visual>
</link>

<!-- Pie -->
<link name="pie">
  <visual>
    <geometry><box size="0.1 0.07 0.2"/></geometry>
    <origin xyz="0 0 0.1" rpy="0 0 0"/>
    <material name="yellow"/>
  </visual>
</link>

<!-- Joint base link_0 (esttico, fijo) -->
<joint name="joint_base" type="fixed">
  <parent link="base_link"/>
  <child link="link_0"/>
  <origin xyz="0.1 0 0.95" rpy="0 0 0"/>
</joint>

<!-- Joint cadera: link_0 fmur -->
<joint name="joint_hip" type="revolute">
  <parent link="link_0"/>
  <child link="femur"/>
  <origin xyz="0 0 0" rpy="2.8 0 0"/>
  <axis xyz="1 0 0"/>
  <limit lower="-3.1" upper="3.1" effort="10" velocity="1"/>
</joint>

<!-- Joint rodilla: fmur tibia -->
<joint name="joint_knee" type="revolute">
  <parent link="femur"/>
  <child link="tibia"/>
  <origin xyz="0 0 0.3" rpy="0.28 0 0"/>
  <axis xyz="1 0 0"/>
  <limit lower="-3.1" upper="3.1" effort="10" velocity="1"/>

```

```

</joint>

<!-- Joint tobillo: tibia pie -->
<joint name="joint_ankle" type="revolute">
  <parent link="tibia"/>
  <child link="pie"/>
  <origin xyz="0 0 0.3" rpy="-1 0 0"/>
  <axis xyz="1 0 0"/>
  <limit lower="-3.1" upper="3.1" effort="10" velocity="1"/>
</joint>

</robot>

```

Se debe tener cuidado con los saltos de línea ya que estos pueden que no se copien y peguen correctamente.

3.4 Visualización del modelo con un archivo de lanzamiento (Launcher)

Una vez que se ha creado y guardado el archivo URDF en la carpeta `urdf/`, es posible visualizar el modelo 3D en RViz. Para ello, se puede utilizar un archivo de lanzamiento (launch file) que configure automáticamente los nodos necesarios.

Como primera opción, se puede emplear el lanzador predeterminado que ofrece el paquete `urdf_tutorial`, el cual está diseñado específicamente para mostrar modelos URDF:

```

ros2 launch urdf_tutorial display.launch.py model:=$(pwd)/src/leg_movement/
urdf/pie.urdf

```

Se debe tener cuidado con los saltos de línea que se generan al copiar y pegar. Revisa que el código pegado tenga la ruta sin espacios.

- `urdf_tutorial` es un paquete oficial de ROS 2 que incluye herramientas básicas de visualización.
- `display.launch.py` es un archivo de lanzamiento genérico que ejecuta los nodos `robot_state_publisher`, `joint_state_publisher_gui` y `rviz`.
- El argumento `model:=...` permite pasar la ruta absoluta al archivo URDF que se desea visualizar.

Este método es útil como verificación rápida, ya que no requiere configurar archivos adicionales. Sin embargo, se recomienda crear un archivo de lanzamiento personalizado dentro del propio paquete (en este caso, `leg_movement`) para tener mayor control sobre:

- La ruta relativa al URDF, sin depender del comando `pwd`.
- La configuración visual inicial en RViz.
- La posibilidad de agregar nodos adicionales en el futuro (controladores, sensores virtuales, simuladores, etc.).

Aun así, este launcher es útil en caso de que se quiera explicar el modelo urdf.

3.5 Generación de un nodo para mover articulaciones

Una vez modelado el pie y configurado un archivo de lanzamiento, es posible agregar nodos que simulen el movimiento de las articulaciones. En esta sección se creará un nodo en C++ llamado `joint_mover`, encargado de publicar señales periódicas tipo seno sobre el tópico `/joint_states`, el cual es leído por `robot_state_publisher` para animar el modelo en RViz.

Ubicación del archivo

Este nodo se llamará `joint_mover.cpp` y deberá guardarse en el directorio `src/` del paquete `leg_movement`:

```
ros2_ws/src/leg_movement/src/joint_mover.cpp
```

Código del nodo

A continuación se muestra el código completo:

```
#include "rclcpp/rclcpp.hpp"
#include "sensor_msgs/msg/joint_state.hpp"
#include <chrono>
#include <cmath>

using namespace std::chrono_literals;

class JointMover : public rclcpp::Node {
public:
    JointMover() : Node("mover_joint"), angle_(0.0) {
        publisher_ = this->create_publisher<sensor_msgs::msg::JointState>("
            joint_states", 10);
        dt_ = 100ms; // Intervalo de tiempo
        k_ = 0.0;
        f1_ = 0.5;
        f2_ = 0.5;
        f3_ = 0.5;
        timer_ = this->create_wall_timer(dt_, std::bind(&JointMover::
            timer_callback, this));

        // Inicializa los nombres de los joints
        joint_state_msg_.name = {"joint_hip", "joint_knee", "joint_ankle"};
        joint_state_msg_.position = {0.0, 0.0, 0.0};
    }

private:
    void timer_callback() {
        // Crea una onda senoidal para joint_1
        angle_ += 0.1; // Incrementa el ngulo
        double t = k_ * (dt_.count()/1000.0); // Convierte el tiempo a segundos
        if (angle_ > 2 * M_PI) angle_ = 0.0;
```

```

    joint_state_msg_.header.stamp = this->get_clock()->now();
    joint_state_msg_.position[0] = 0.68*std::sin(2*M_PI*f1_*t); // joint_1
    joint_state_msg_.position[1] = 0.3*std::sin(2*M_PI*f2_*t);
    joint_state_msg_.position[2] = 0.5*std::sin(2*M_PI*f3_*t); // joint_3

    publisher_->publish(joint_state_msg_);
    k_ = k_ + 1;
}

rclcpp::Publisher<sensor_msgs::msg::JointState>::SharedPtr publisher_;
rclcpp::TimerBase::SharedPtr timer_;
sensor_msgs::msg::JointState joint_state_msg_;
double angle_;
std::chrono::milliseconds dt_;

double k_;
double f1_,f2_,f3_;
};

int main(int argc, char *argv[]) {
    rclcpp::init(argc, argv);
    rclcpp::spin(std::make_shared<JointMover>());
    rclcpp::shutdown();
    return 0;
}

```

Explicación del nodo

Este nodo tiene como función generar órdenes periódicas para cada articulación del modelo robótico, simulando su movimiento mediante funciones seno. Cada línea que define un `std::sin()` representa una orden directa de posición enviada a una articulación específica, produciendo un movimiento oscilatorio suave.

- `joint_state_msg_.name`: define los nombres de las articulaciones simuladas (`joint_hip`, `joint_knee`, `joint_ankle`).
- `timer_callback()`: se ejecuta cada 100 ms y actualiza las posiciones articulares en función del tiempo transcurrido.
- `joint_state_msg_.position[0-2]`: contiene los valores articulares calculados con ondas senoidales de la forma $A \sin(2\pi f t)$, donde:
 - **A**: amplitud del movimiento, controla el rango angular.
 - **f**: frecuencia de oscilación, define la rapidez del movimiento.
 - **t**: tiempo simulado, crece con cada iteración del temporizador.

- `publisher_`: publica el mensaje con los nuevos valores articulares en el t pico `/joint_states`, lo que permite su visualizaci n en RViz.

Este nodo ofrece una base clara y modificable para experimentar con diferentes perfiles de movimiento, permitiendo ajustar frecuencia y amplitud para observar sus efectos en la simulaci n del robot. Es una forma intuitiva de generar locomoci n artificial a partir de funciones matem ticas simples.

3.6 Compilaci n del nodo: modificaci n de `CMakeLists.txt` y `package.xml`

Aunque este documento no tiene como objetivo ense ar la sintaxis ni el funcionamiento detallado de los archivos `CMakeLists.txt` ni `package.xml`, es importante tener presente que ambos deben modificarse para que ROS 2 pueda compilar y reconocer correctamente el nodo `joint_mover`.

Archivo `CMakeLists.txt`

Este archivo se encuentra en la ra z del paquete `leg_movement`. A continuaci n, se presenta una versi n funcional del mismo, con las modificaciones necesarias para compilar el nodo y asegurar que se instalen correctamente las carpetas relevantes del proyecto:

```
cmake_minimum_required(VERSION 3.5)
project(leg_movement)

# Encuentra las dependencias
find_package(ament_cmake REQUIRED)
find_package(rclcpp REQUIRED)
find_package(sensor_msgs REQUIRED)

# Instala carpetas launch, urdf y rviz
install(
  DIRECTORY launch urdf rviz
  DESTINATION share/${PROJECT_NAME}
)

# Nodo joint_mover
add_executable(joint_mover src/joint_mover.cpp)
ament_target_dependencies(joint_mover rclcpp sensor_msgs)
install(TARGETS joint_mover DESTINATION lib/${PROJECT_NAME})

ament_package()
```

Archivo `package.xml`

Este archivo describe las dependencias y metadatos del paquete. Tambi n debe actualizarse para reflejar que el paquete depende de bibliotecas como `rclcpp`, `sensor_msgs`, y los paquetes necesarios para lanzar el modelo en RViz. La siguiente es una versi n completa y v lida del archivo:

```
<?xml version="1.0"?>
<package format="3">
  <name>leg_movement</name>
  <version>0.0.0</version>
  <description>Visualizacin de un rectngulo URDF</description>
  <maintainer email="dockeruser@todo.todo">dockeruser</maintainer>
  <license>Apache-2.0</license>

  <!-- Slo para CMake -->
  <buildtool_depend>ament_cmake</buildtool_depend>

  <!-- Dependencias en tiempo de ejecucin -->
  <exec_depend>launch</exec_depend>
  <exec_depend>launch_ros</exec_depend>
  <exec_depend>joint_state_publisher</exec_depend>
  <exec_depend>robot_state_publisher</exec_depend>
  <depend>rclcpp</depend>
  <depend>sensor_msgs</depend>

  <!-- Para tests de estilo, si los usas -->
  <test_depend>ament_lint_auto</test_depend>
  <test_depend>ament_lint_common</test_depend>

  <export>
    <build_type>ament_cmake</build_type>
  </export>
</package>
```

Importante

Después de guardar ambos archivos, es necesario volver a compilar el paquete desde la raíz del workspace:

```
cd ~/ros2_ws
colcon build
```

Luego, se debe actualizar el entorno para que ROS 2 pueda encontrar el nodo:

```
source install/setup.bash
```

Con esto, el nodo `joint_mover` quedará correctamente compilado e integrado al sistema, y podrá ser lanzado desde consola o desde un archivo `.launch.py`.

subsection Archivo de lanzamiento propio: `ver_pie.launch.py`

Con el fin de visualizar el modelo del pie junto con su movimiento (creado por nosotros), es necesario lanzar varios nodos simultáneamente. Para ello, se crea un archivo de lanzamiento personalizado llamado `ver_pie.launch.py`, que debe guardarse dentro de la carpeta `launch` del proyecto.

Este archivo lanza los siguientes nodos:

- **joint_state_publisher:** aunque en este caso el nodo `joint_mover` publica directamente los estados articulares, este nodo es requerido por compatibilidad con `robot_state_publisher`.

- **robot_state_publisher:** interpreta el modelo URDF y publica las transformaciones (tf) de cada articulación en función del estado recibido.
- **joint_mover:** es el nodo personalizado que simula el movimiento de las articulaciones del pie mediante funciones senoidales. Publica en el tópico /joint_states.
- **rviz2:** herramienta de visualización 3D utilizada para mostrar el modelo del robot.

Estructura del comando de lanzamiento:

Para ejecutar este archivo de lanzamiento, se utiliza el comando:

```
ros2 launch <nombre_del_paquete> <archivo_launch.py>
```

En este caso particular:

```
ros2 launch leg_movement ver_pie.launch.py
```

Donde:

- **leg_movement** es el nombre del paquete que contiene la carpeta launch.
- **ver_pie.launch.py** es el archivo de lanzamiento que define los nodos a ejecutar.

Este comando se encarga de iniciar automáticamente todos los nodos definidos en el archivo de lanzamiento de manera simultánea, facilitando la integración y visualización del modelo robótico.

```
from launch import LaunchDescription
from launch_ros.actions import Node
import os

from ament_index_python.packages import get_package_share_directory

def generate_launch_description():
    # Ruta al archivo URDF
    urdf_file = os.path.join(
        get_package_share_directory('leg_movement'),
        'urdf',
        'pie.urdf'
    )

    # Leer el contenido del URDF como string
    with open(urdf_file, 'r') as infp:
        robot_description_content = infp.read()

    return LaunchDescription([
        # Nodo joint_state_publisher (requerido por robot_state_publisher)
        Node(
            package='joint_state_publisher',
            executable='joint_state_publisher',
            name='joint_state_publisher'
        ),

        # Nodo robot_state_publisher
```

```

Node(
    package='robot_state_publisher',
    executable='robot_state_publisher',
    name='robot_state_publisher',
    parameters=[{'robot_description': robot_description_content}]
),

# Nodo personalizado joint_mover
Node(
    package='leg_movement',
    executable='joint_mover',
    name='joint_mover',
    output='screen'
),

# Nodo RViz
Node(
    package='rviz2',
    executable='rviz2',
    name='rviz2',
    output='screen'
),
])

```

Sin embargo, es importante destacar que este archivo `launch` simplemente ejecuta RViz, pero **no le indica qué mostrar**. Por esta razón, al iniciarse, RViz puede aparecer vacío o sin cargar el modelo.

Para resolver esto, se debe configurar manualmente RViz y guardar una vista personalizada. Este proceso se explica en la siguiente subsección.

3.6.1 Configuración inicial de RViz y guardado de vista

Para mantener el proyecto organizado, se recomienda crear una carpeta llamada `rviz` dentro del paquete `leg_movement`. Esta carpeta servirá para almacenar archivos de configuración personalizados de RViz.

Desde la raíz del proyecto, puede crearse con el siguiente comando:

```
mkdir -p src/leg_movement/rviz
```

Luego, cuando se configure RViz por primera vez, el archivo de configuración podrá guardarse directamente en esta ubicación.

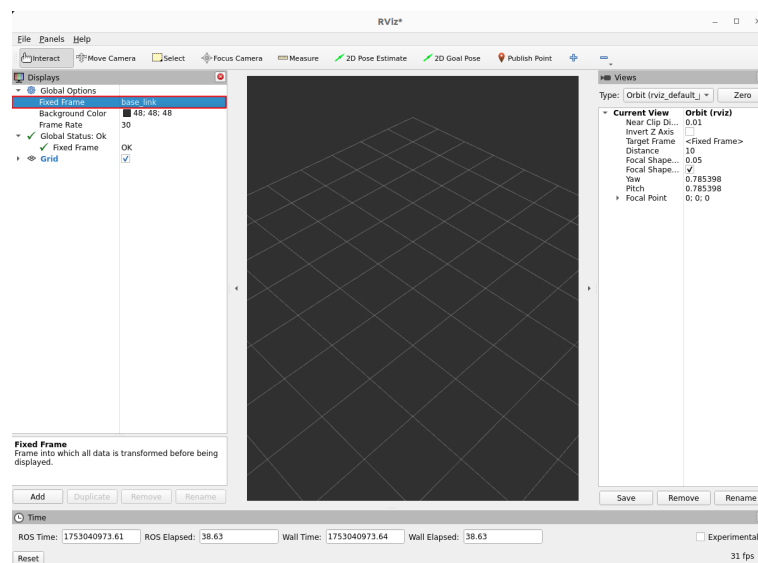
Una vez compilado el proyecto con `colcon build`, es posible lanzar el entorno completo utilizando el archivo de lanzamiento creado previamente:

```
ros2 launch leg_movement ver_pie.launch.py
```

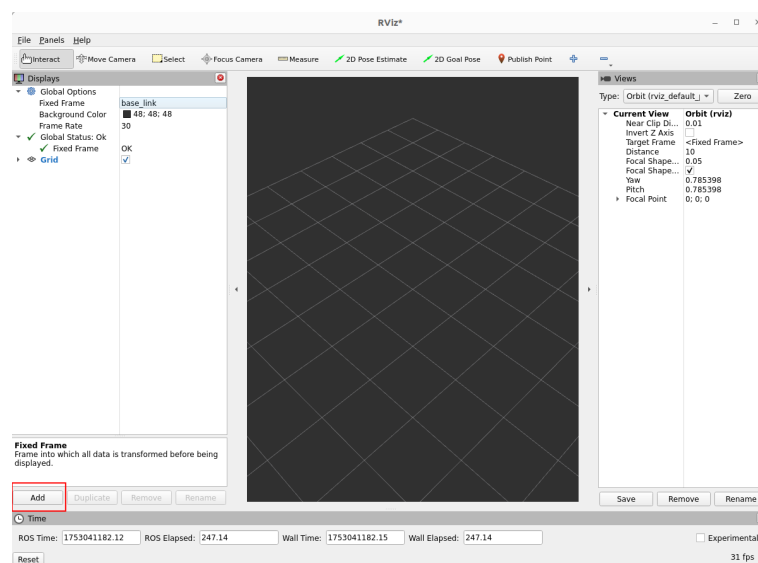
Este comando abrirá RViz junto con los nodos necesarios para visualizar el modelo del pie y simular su movimiento. Sin embargo, como RViz no ha sido configurado todavía, la interfaz aparecerá vacía inicialmente. Es necesario realizar algunos pasos manuales para que el robot se muestre correctamente.

Pasos para visualizar el modelo:

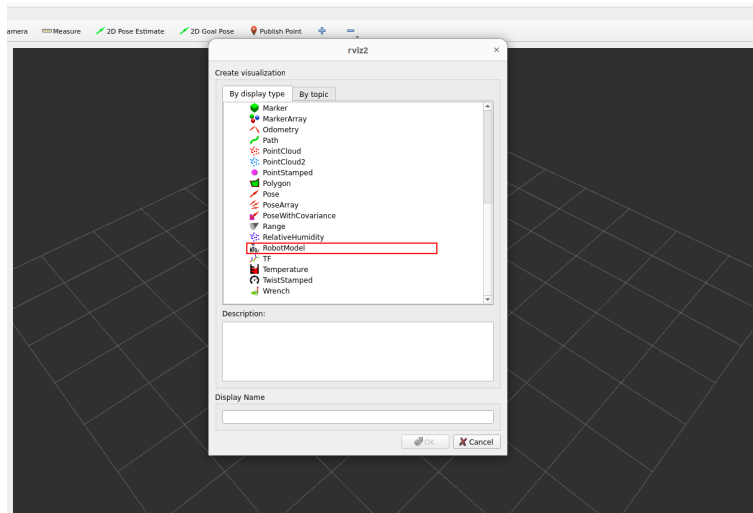
1. En la parte superior izquierda de RViz, cambiar el **Fixed Frame** a `base_link`.



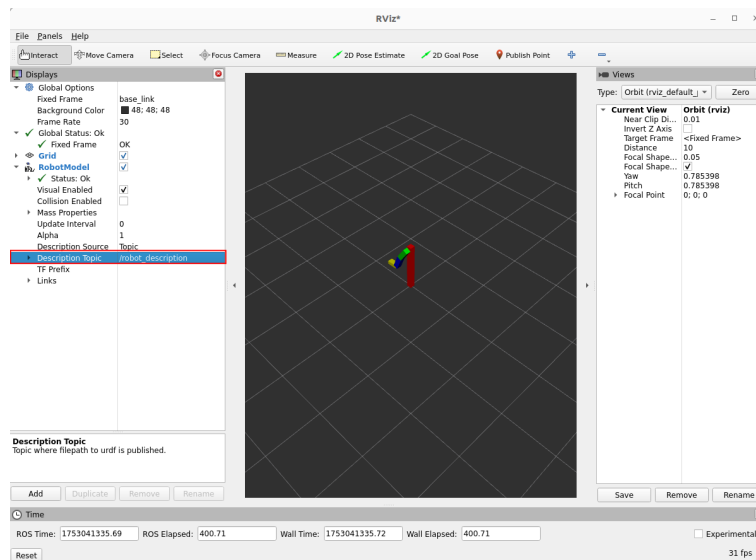
2. En la parte inferior izquierda, hacer clic en el botón **Add**.



3. Seleccionar la opción `RobotModel` y hacer clic en **OK**.



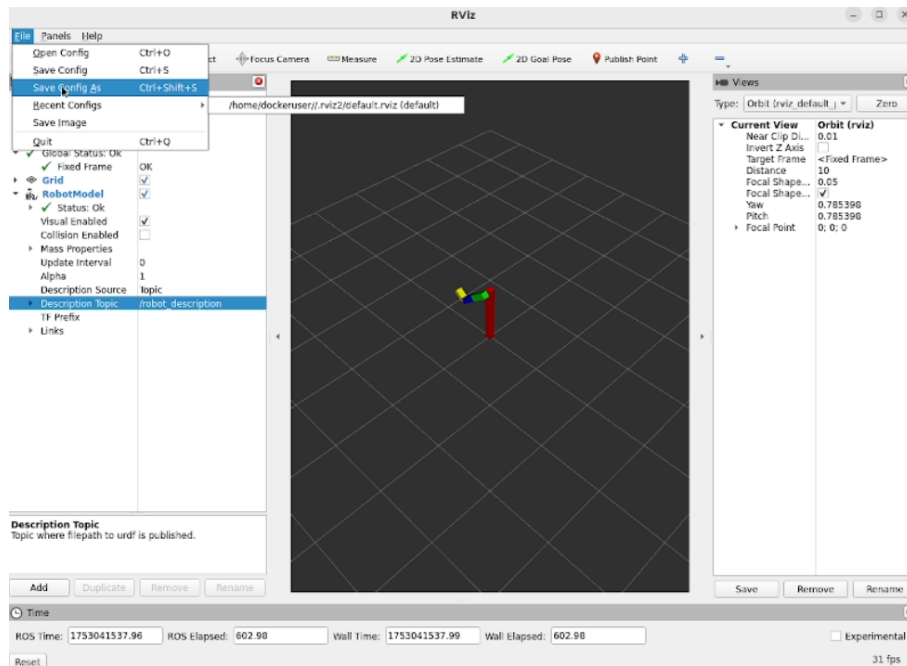
4. Confirmar que el campo Description Topic sea robot_description.



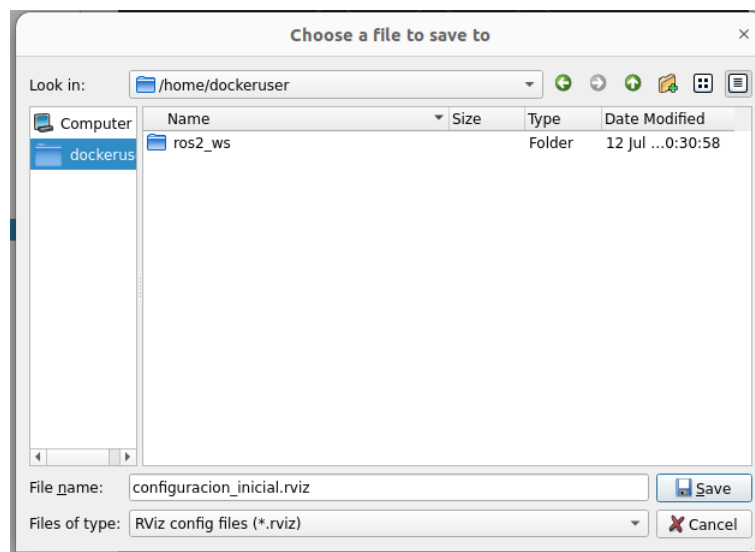
Una vez completados estos pasos, el modelo del robot aparecerá en la escena 3D.

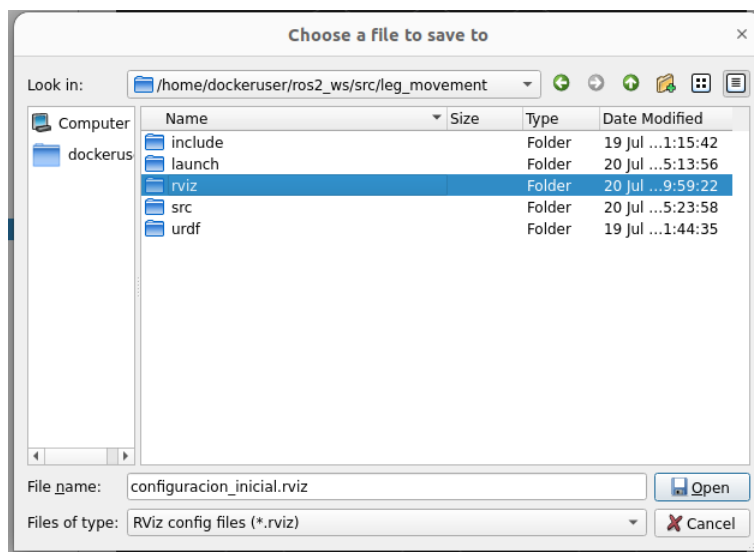
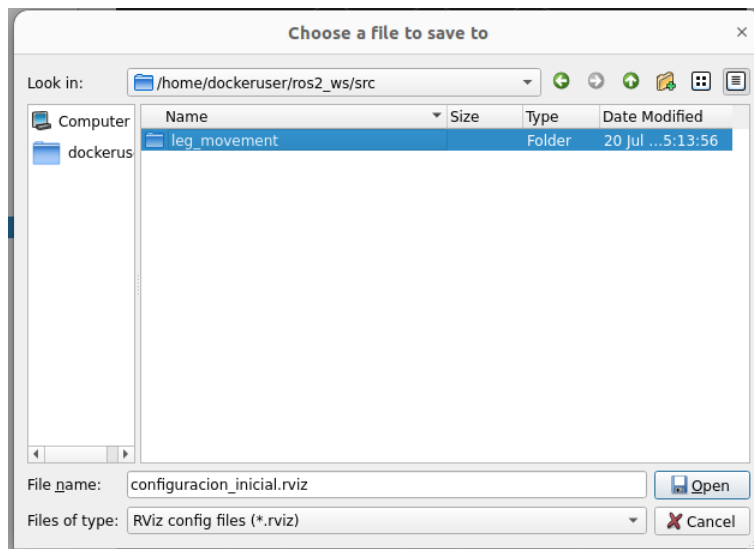
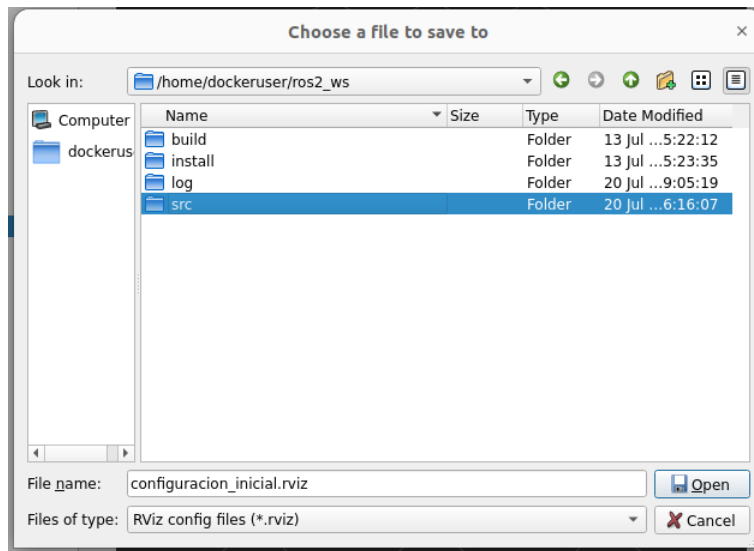
Guardado de configuración personalizada:

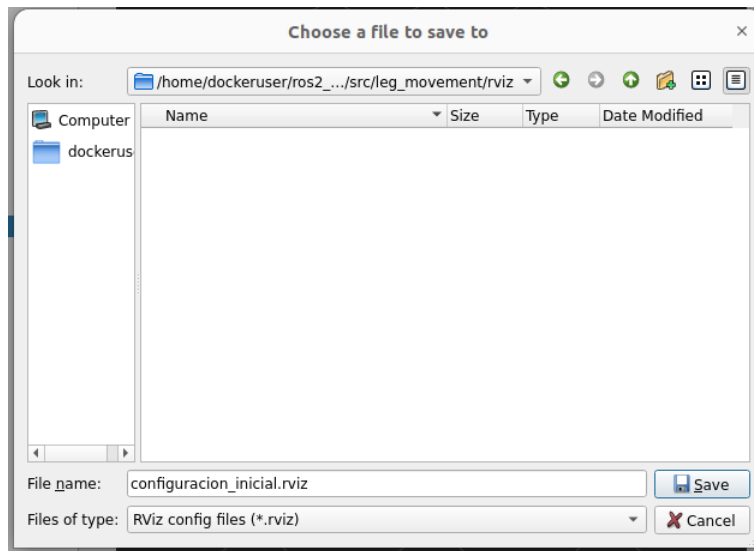
1. Ir al menú File → Save Config As...



2. Guardar el archivo dentro de la carpeta siguiendo la ruta conocida. `rviz/` del proyecto, por ejemplo con el nombre `configuracooin_inicial.rviz`







Una vez guardada esta configuración, se podrá editar el archivo `ver_pie.launch.py` para que RViz cargue automáticamente esta vista en futuras ejecuciones.

3.6.2 Modificar el launcher para abrir una configuración específica en RViz

Después de haber guardado el archivo de configuración personalizada de RViz, podemos modificar el archivo de lanzamiento `ver_pie.launch.py` para que RViz abra automáticamente este archivo en futuras ejecuciones.

Ruta esperada:

- Archivo: `configuracion_inicial.rviz`
- Ubicación: `src/leg_movement/rviz/configuracion_inicial.rviz`

El siguiente fragmento muestra cómo debe actualizarse el launcher del proyecto.

```
from launch import LaunchDescription
from launch_ros.actions import Node
import os
from ament_index_python.packages import get_package_share_directory

def generate_launch_description():
    pkg_dir = get_package_share_directory('leg_movement')
    urdf_path = os.path.join(pkg_dir, 'urdf', 'pie.urdf')
    rviz_config_path = os.path.join(pkg_dir, 'rviz', 'configuracion_inicial.rviz')

    return LaunchDescription([
        # Nodo para publicar el estado de las articulaciones
        Node(
            package='joint_state_publisher',
            executable='joint_state_publisher',
            name='joint_state_publisher',
            output='screen',
```

```

),

# Nodo para publicar el modelo del robot
Node(
    package='robot_state_publisher',
    executable='robot_state_publisher',
    name='robot_state_publisher',
    parameters=[{'robot_description': open(urdf_path).read()}],
    output='screen',
),

# Nodo propio que mueve las articulaciones
Node(
    package='leg_movement',
    executable='joint_mover',
    name='joint_mover',
    output='screen',
),

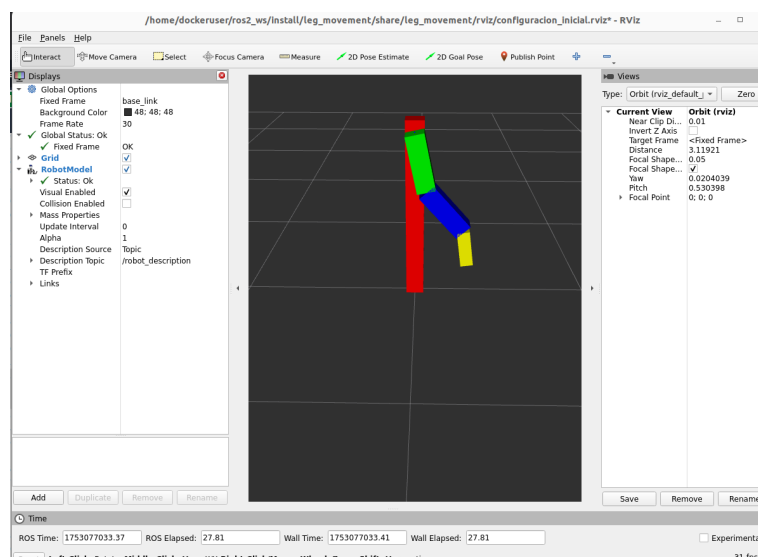
# Nodo de visualización con archivo de configuración
Node(
    package='rviz2',
    executable='rviz2',
    name='rviz2',
    arguments=['-d', rviz_config_path],
    output='screen',
),
)
)

```

Con este cambio, cada vez que se ejecute:

```
ros2 launch leg_movement ver_pie.launch.py
```

RViz abrirá automáticamente el entorno configurado previamente, mostrando el modelo del pie sin necesidad de realizar los pasos manuales de visualización.



Repositorio del proyecto

En caso de presentarse inconvenientes al copiar los fragmentos de código incluidos en este documento (por ejemplo, errores por espaciado o formato), se recomienda clonar directamente el repositorio completo del proyecto desde GitHub. Para ello, abra una terminal dentro del contenedor y ejecute:

```
cd ROS2_Docker_UI/ros2_ws/src
git clone https://github.com/Smacds/Proyecto-Pie-Rob-tico-en-ROS-2.git
```

Esto descargará el proyecto con toda su estructura (URDF, lanzadores, nodo y configuración de RViz) lista para compilarse y ejecutarse correctamente.

Nota: El nombre de la carpeta del proyecto en el repositorio es **Proyecto-Pie-Rob-tico-en-ROS-2**, diferente del nombre usado localmente en este documento (**leg_movement**). Asegúrese de ajustar los comandos y rutas si decide usar el proyecto clonado.

4 Uso de múltiples terminales dentro del contenedor Docker

Cuando se está ejecutando un proceso dentro de una terminal en el contenedor Docker, dicha terminal queda ocupada y no es posible ingresar nuevos comandos mientras el proceso esté en ejecución. Por esta razón, es necesario abrir una nueva terminal e ingresar nuevamente al contenedor para poder interactuar simultáneamente con el contenedor.

Para acceder nuevamente al mismo contenedor y ejecutar comandos adicionales de ROS 2, utilice el siguiente comando en la nueva terminal:

```
./scripts/bash
```

Este comando le permitirá abrir una sesión interactiva dentro del contenedor ya en ejecución, desde donde podrá ejecutar comandos que pueden ser de utilidad como por ejemplo:

- **ros2 node list:** permite conocer que nodos están activos.
- **ros2 topic list:** permite conocer que topics están activos.

Adicionalmente, para visualizar de forma gráfica y en tiempo real la comunicación entre nodos y tópicos, puede ejecutar el siguiente comando dentro del contenedor:

```
ros2 run rqt_graph rqt_graph
```

Esta herramienta abre una interfaz gráfica que muestra la topología del sistema ROS 2, facilitando la comprensión del flujo de datos entre los diferentes componentes.

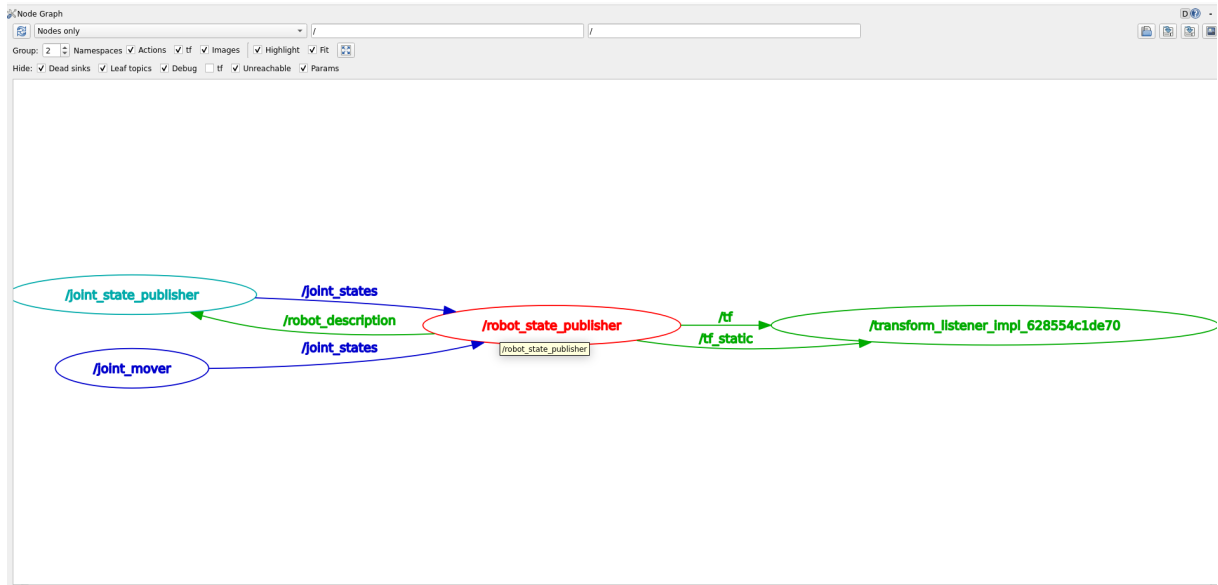


Figure 4: Visualización de los nodos y tópicos.

La estructura varia dependiendo del proyecto.

Cada terminal abierta representa una sesión independiente dentro del contenedor, por lo que se puede mantener múltiples terminales activas para diferentes propósitos simultáneamente.

5 Otros proyectos

Además del proyecto del pie robótico, también se han desarrollado otros modelos robóticos educativos que pueden ser utilizados como base para ejercicios más complejos o prácticas de control.

5.1 Robot cuadrúpedo

Se ha creado otro robot, esta vez con forma de cuadrúpedo, con el fin de explorar configuraciones multienlace y sistemas locomotores más avanzados. Este modelo puede ser utilizado para experimentar con algoritmos de coordinación de patas, control centralizado o sistemas CPG (Generadores Centrales de Patrones).

El proyecto del robot cuadrúpedo se encuentra disponible públicamente en el siguiente repositorio:

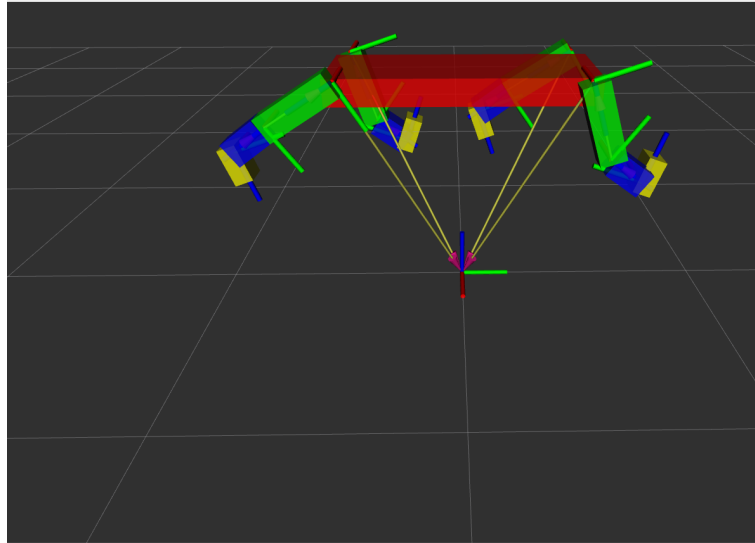
- **Repositorio GitHub:** https://github.com/Smacds/four_leg_movement

Para clonar este proyecto, se recomienda hacerlo desde la terminal en el contenedor, siguiendo los pasos:

```
cd ROS2_Docker_UI/ros2_ws/src
git clone https://github.com/Smacds/four_leg_movement.git
```

Una vez clonado, puede compilarse junto con los demás proyectos en el workspace.

Este modelo es útil para quienes desean comenzar a trabajar con robots con múltiples patas y servomecanismos coordinados, pero manteniendo la estructura sencilla vista a lo largo de este documento.



Debido a un problema, puede el robot no entre en movimiento al lanzar el launcher, por lo que se debe terminar el proceso y volver a lanzar el launcher hasta que el movimiento inicie.

References

- [1] Alejandro Gomez. *ROS2 Docker UI*. https://github.com/alda92/ROS2_Docker_UI. Accedido en julio de 2025. 2022.
- [2] Articulated Robotics. *How do we describe a robot? With URDF! — Getting Ready to build Robots with ROS 7*. <https://www.youtube.com/watch?v=Cwdbsvcp0HM>. Accedido en julio de 2025. 2021.
- [3] Smacds. *Four Leg Movement*. https://github.com/Smacds/four_leg_movement. Accedido en julio de 2025. 2025.
- [4] Smacds. *Proyecto Pie Robótico en ROS 2*. <https://github.com/Smacds/Proyecto-Pie-Rob-tico-en-ROS-2>. Accedido en julio de 2025. 2025.