Taller 6: Cinemática y odometría de un robot diferencial

Robótica Móvil

2^{do} cuatrimestre 2025

1 Introducción

El objetivo de este taller es lograr una implementación del modelo cinemático de un robot diferencial que nos permita traducir velocidades lineal y angular en velocidades para cada uno de los motores y estimar la pose del robot en base a la información de los encoders mediante odometría.
Desde el campus pueden descargarse los paquetes que serán utilizados durante este taller. Se recomienda fuertemente que ante cualquier duda repasen las diapositivas de la clase.

1.1 Añadir paquetes al workspace

Los paquetes compilados con Colcon poseen una determinada estructura de directorios. Para añadir nuevos paquetes a un *workspace* ya creado, se requiere copiarlos al volumen compartido con el container creado (recordar que éste se mapea a `/root/ros2_ws/src/robotica`).
Una vez copiados los paquetes, es posible compilarlos utilizando el comando desde `/root/ros2_ws`:

```
$ colcon build --symlink-install
```

No olviden, luego de compilar los paquetes, configurar las variables de entorno del workspace utilizando en la raíz del directorio:

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

Les recordamos además que para correr los paquetes instalados pueden utilizar el comando

```
$ ros2 run <nombre_del_paquete> <nombre_del_ejecutable>
```

Por ejemplo, para el paquete de `modelo_diferencial` provisto en este taller:

```
$ ros2 run modelo_diferencial pioneer_odometry_node
```

Nota: Recordar volver a compilar luego de cambiar el código de algún nodo contenido en un paquete.

1.2 Teleoperación

Para la teleoperación del robot pueden usar tanto lo desarrollado durante el Taller 5, como el módulo `teleop_twist_keyboard`:

```
$ ros2 run teleop_twist_keyboard teleop_twist_keyboard
```

Ejercicio 1: Cinemática inversa

Se requiere calcular la velocidad de rotación de cada rueda a partir de un comando de velocidad. El nodo encargado de realizar esto se encuentra dentro del paquete `modelo_diferencial`, implementado en el archivo: `modelo_diferencial/src/pioneer_odometry.cpp`.

Deberán completar el callback `on_velocity_cmd`, el cual recibe un mensaje de tipo `Twist` del tópico `/cmd_vel`, y deberá publicar un mensaje de velocidad de rotación para cada motor, según el formato `std_msgs/msg/Float64`.

Para esto, inicializar `modelo_diferencial/coppeliaSim/modelo_diferencial.ttt` en el simulador CoppeliaSim. Lanzar los nodos requeridos utilizando el comando `ros2 run` y comprobar que se condice el movimiento del Pioneer en la simulación con respecto a los comandos de velocidad publicados para las ruedas.

Importante: Cuando creamos nuestros propios paquetes de ROS2, es posible que generemos estructuras de mensajes que no son las que están estandarizadas. Entonces, éstas podrían no ser reconocidas por el simulador CoppeliaSim, y se generará un mensaje de error cuando quiera comenzarse la simulación. Si es el caso, y se requiere agregarlas para que sean compatibles, al final de este enunciado se encuentra una sección explicando paso a paso cómo hacerlo.

Ejercicio 2: Odometría

A partir de la información provista por los encoders, se requiere calcular la pose del robot en referencia a la pose inicial (momento cuando se inició la simulación) y la velocidad actual del robot. Para esto deben modificar el mismo nodo del paquete `modelo_diferencial` implementado en el archivo `modelo_diferencial/src/pioneer_odometry.cpp`.

Deberán completar el callback `on_encoder_ticks`, que recibe un mensaje de tipo `EncoderTicks`. Para obtener los ticks referidos al motor izquierdo y derecho utilizar respectivamente:
`encoder_msg.ticks_left.data` y `encoder_msg.ticks_right.data`.

Recordar que las cuentas de encoder son acumulativas y poseen signo. Al avanzar, los ticks de encoder de una determinada rueda se incrementan, mientras que al retroceder disminuyen. Entonces, los ticks de encoder podrían ser negativos.

Nota 1: Para operar con π pueden utilizar `M_PI` en el código.

Nota 2: Para distinguir en RViz2 la pose verdadera (*ground truth*) respecto de la estimada, al exportar el archivo `URDF` cambien `robot_base` por `base_link_gt`, en lugar de `base_link`, como se utilizó en el taller anterior. Esta última se utilizará para identificar la pose estimada. De esta forma, reconoceremos el marco correspondiente al centro del robot calculado por su odometría como `base_link`, y al marco de referencia *ground-truth* como `base_link_gt`.

Deberán publicar un mensaje de odometría por el tópico `/robot/odometry`. Tener en cuenta que el mensaje de odometría contiene tanto la pose del robot como la velocidad del mismo. Leer con atención los comentarios en el código para la construcción de los mensajes.

Una vez que finalicen de implementar la función de callback `on_encoder_ticks` deberán:

a) Inicializar `modelo_diferencial/coppeliaSim/modelo_diferencial.ttt` en el simulador CoppeliaSim. Utilizar el nodo de teleoperación para enviar comandos de velocidad al robot y utilizar el RViz2 para visualizar la odometría publicada.

b) La simulación provee información de *ground-truth* de manera de poder comparar la calidad de la odometría computada. Evaluar lo que sucede cuando requerimos una velocidad únicamente angular al robot. Analizar cómo se comporta la odometría calculada en comparación a la información de *ground-truth*.

c) Modificar las propiedades dinámicas del suelo de la simulación reduciendo la fricción que aplica sobre objetos:

   1. Expandir objeto `ResizableFloor`.
   2. Hacer doble click en `ResizableFloor_element` (sobre el ícono del cubo turquesa).
   3. Hacer click en `Dynamic properties dialog`.
   4. Hacer click en `Engine properties` .

5. Cambiar los valores de fricción en el apartado de *bullet* (motor físico por defecto), según la versión de *bullet* que esté utilizando.

¿Que sucede con la odometría calculada en esta situación? Probar con valores 10 y 100 veces menores que el inicial. Sacar conclusiones.

Agregar estructuras de mensaje nuevas a CoppeliaSim

Para poder vincular el simulador CoppeliaSim con ROS2, los desarrolladores de este simulador generaron una interfaz, `sim_ros2_interface`, que le permite interactuar con ROS2 generando tópicos y nodos. Para esto, dicha interfaz define las estructuras de mensajes para que el simulador pueda entregar coherentemente la información a ROS2. Si se desean agregar tipos de mensajes nuevos, es necesario modificar esta interfaz para **agregar** la referencia a la estructura del nuevo mensaje. Como ejemplo, veamos cómo añadir el mensaje de `robmovil_msgs/msg/EncoderTicks`, provisto en el material de este Taller. Es importante que el paquete `robmovil_msgs` se encuentre ya previamente instalado.

Dentro del container de docker, la interfaz se encuentra en el directorio:
`/root/ros2_ws/src/sim_ros2_interface`.

- En `/root/ros2_ws/src/sim_ros2_interface/meta/interfaces.txt`, agregar en una nueva linea el nombre del nuevo tipo de mensajes (se puede usar tanto `nano`, `vim` o `Visual Studio Code` para editar el texto):

```
robmovil_msgs/msg/EncoderTicks
```

- En `/root/ros2_ws/src/sim_ros2_interface/package.xml`, agregar la dependencia:

```
<depend>robmovil_msgs</depend>
```

- En `/root/ros2_ws/src/sim_ros2_interface/CMakeLists.txt`, pedir que encuentre la dependencia de `robmovil_msgs`:

```
find_package(robmovil_msgs REQUIRED)
```

  y el link con el compilador `ament` (dentro de `ament_target_dependencies`):

```
robmovil_msgs
```

Una vez hecho esto, volver a compilar `sim_ros2_interface`, desde `/root/ros2_ws`:

```
colcon build --packages-select sim_ros2_interface
```

Por último, es necesario volver a abrir el simulador CoppeliaSim para que tome estos cambios. Hecho esto, ahora reconocerá este nuevo tipo de mensajes.

Importante: Como esta modificación fue hecha en el container y no sobre la imagen con que se crea el container, si éste container se destruye y se crea uno nuevo, será necesario volver a realizar este procedimiento. Esto se puede llegar a resolverse modificando la imagen, compilándola nuevamente, y generando los container a partir de esta imagen modificada (no lo recomendamos).