Práctica 3

Introducción a la Programación Robótica

Control del movimiento de un robot en un entorno 3D (Gazebo)

Enunciado

En esta práctica revisaremos conceptos tratados anteriormente, como la comunicación a través del modelo **publicador-suscriptor,** pero en este caso dentro de un entorno de simulación 3D como Gazebo.

En particular, el objetivo de la práctica es escribir un **script que monitorice el comportamiento de un robot.** En particular, queremos conseguir que el robot emita simplemente un mensaje del tipo "Colisión con 'x'" cuando contacte con un objeto en un escenario 3D dado. Para ello diseñaremos un script que desarrolle toda la lógica de la comunicación en el entorno de simulación para este modelo de comunicación de procesos.

La creación de un modelo de comunicación de tipo publicador-suscriptor va a implicar en este caso la realización de los siguientes pasos:

1. Instalar los paquetes necesarios para crear un escenario 3D
2. Probar el escenario y el funcionamiento de un robot
3. Crear el script detector de colisiones en el escenario

Para empezar, lanza el entorno de simulación Gazebo junto con la opción de interactuar con el robot mediante el teclado.

Parte 1: Instalación de paquetes TurtleBot3 y Teleop

Instala los siguientes paquetes que te facilitarán la simulación de un robot y la interacción con el entorno.

- Paquete `turtlebot3_gazebo`: simula el robot TurtleBot3 en Gazebo.
- Paquete `teleop_twist_keyboard`: permite controlar el robot desde el teclado.

```
# sudo apt-get install ros-noetic-turtlebot3-gazebo
```

```
# sudo apt-get install ros-noetic-teleop-twist-keyboard
```

Verifica que los paquetes se hayan instalado correctamente.

Configura además las variables de entorno para el modelo de robot que vamos a utilizar. En este caso, utilizaremos el modelo de TurtleBot3 Burger.

```
export TURTLEBOT3_MODEL=burger
```

Parte 2: Creación del escenario en Gazebo

Gazebo ofrece la opción de cargar diferentes tipos de escenarios. En nuestro caso, partiremos de un escenario vacío.

Inicia en una terminal la simulación básica de Gazebo con el robot TurtleBot3 Burger.

```
# roslaunch turtlebot3_gazebo turtlebot3_empty_world.launch
```

Ahora añadiremos un objeto al mundo manualmente:

- Abre la interfaz gráfica de Gazebo.
- En la barra superior de la ventana de simulación, busca y selecciona el objeto "unit_box" del menú.
- Coloca el "unit_box" en cualquier posición dentro del mundo, preferentemente a una distancia razonable del robot para facilitar la detección de colisiones.

Parte 3: Control del robot

En otra terminal, lanza el paquete `teleop_twist_keyboard` para controlar el movimiento del robot con el teclado.

```
# rosrun teleop_twist_keyboard teleop_twist_keyboard.py
```

Si pierdes el control del robot en el escenario, puedes situarlo en su posición original desde el menú "Edit" --> "Reset Model Poses".

Parte 4: Detección de Colisiones (Publicador-Suscriptor)

Antes de implementar el sistema de detección de colisiones, crea un paquete de ROS en tu espacio de trabajo para alojar el script en Python.

Pasos:

- Abre un terminal y navega a tu espacio de trabajo de catkin:

```
# cd ~/catkin_ws/src
```

- Crea un paquete llamado `collision_detector_pkg` con dependencias de `rospy` y `sensor_msgs`:

```
# catkin_create_pkg collision_detector_pkg rospy sensor_msgs
```

Recuerda modificar el fichero **CMakelists.txt** antes de compilar.

- Compila tu espacio de trabajo

```
# cd ~/catkin_ws
# catkin_make
```

Ahora implementaremos un sistema que detecte cuando el robot colisiona con la "unit_box", utilizando para ello una comunicación **publicador-suscriptor**.

Por un lado, la parte de **publicación** se asocia al tópico "**/scan**", que es publicado por el nodo responsable de gestionar el **sensor LiDAR** del TurtleBot3. Lanzar el comando "*# roslaunch turtlebot3_gazebo turtlebot3_empty_world.launch*", como hemos hecho anteriormente, supone activar el sensor LiDAR simulado y publicar automáticamente datos asociados a su funcionamiento a través del tópico "/scan". Por tanto, no debes hacer nada en este sentido salvo confirmar que **`/scan`** está activo:

```
$ rostopic list
```

Por otro lado, la parte de **suscripción** va a necesitar de código que capture los datos del robot a fin de detectar posibles colisiones. Dentro del paquete **collision_detector_pkg**, crea la carpeta "scripts" y copia en un archivo que puedes llamar `collision_detector.py` el script para la detección de colisiones.

Dentro del paquete **collision_detector_pkg**, crea la carpeta "Scripts" y copia en un archivo que puedes llamar `collision_detector.py` el script para la detección de colisiones.

```
#!/usr/bin/env python3
import rospy
from sensor_msgs.msg import LaserScan

def callback(scan_data):
    min_distance = min(scan_data.ranges)  # Detecta la distancia mínima
    if min_distance < 0.2:  # Si el objeto está a menos de 0.2 metros
        rospy.loginfo("¡Colisión detectada!")

def collision_detector():
    rospy.init_node('collision_detector', anonymous=True)
    rospy.Subscriber("/scan", LaserScan, callback)
    rospy.spin()

if __name__ == '__main__':
    try:
        collision_detector()
    except rospy.ROSInterruptException:
        pass
```

Explicación del código:

- Sobre **importación de librerías:**
  - **from sensor_msgs.msg import LaserScan**

    Este mensaje contiene la información que recoge el sensor láser del robot, que se utiliza para detectar la distancia de los objetos alrededor. Consulta la estructura de datos de LaserScan.

- Sobre la función **callback**:
  - **scan_data.ranges:** la propiedad ranges del mensaje *sensor_msgs/LaserScan* es un vector que contiene las distancias medidas por un sensor láser (como un LIDAR) en diferentes ángulos.

    Puede tener los siguientes valores:

    - **Tipo de datos**: ***ranges*** es un array de números de punto flotante (float32), donde cada valor representa la distancia desde el sensor hasta el objeto más cercano en esa dirección.

- **Valores válidos**:

  - Distancias: Los valores en ***ranges*** son distancias en metros. Normalmente, son positivos y representan la distancia al primer objeto detectado en la dirección del rayo correspondiente.
  - Infinito: Puede haber valores que representen distancias "infinito" (o no detectadas) al final del array. En ROS, esto se suele representar como ***Inf*** o un valor muy grande (generalmente, el sensor define un límite máximo, por ejemplo, 3.5 m, 10 m, etc.).
  - NaN (Not a Number): Si el sensor no detecta ningún objeto (por ejemplo, si está obstruido o fuera de rango), el valor puede ser NaN. Esto indica que no hay datos válidos para esa lectura.
  - Longitud del array: La longitud del array *ranges* está determinada por el número de rayos emitidos por el sensor en un solo escaneo. El número de rayos puede variar según la configuración del sensor.

  Los valores en el array están ordenados desde el **ángulo de visión mínimo hasta el máximo** del sensor, con un valor correspondiente a cada ángulo. Estos valores permiten a los algoritmos de navegación y percepción calcular la posición de los obstáculos y determinar la configuración del entorno alrededor del robot.

  - **min(scan_data.ranges):** esta función busca la distancia más pequeña de todas las que el sensor detecta. Si el valor es muy pequeño (menor de 0.2 metros), significa que hay un objeto muy cerca del robot, indicando una posible colisión.
  - **rospy.loginfo:** muestra un mensaje en la consola cuando se detecta una colisión.

- Sobre la función **collision_detector**:
  - **rospy.init_node('collision_detector')**: crea y registra un nuevo nodo en ROS llamado collision_detector. Este nodo será el encargado de detectar colisiones.
  - **rospy.Subscriber("/scan", LaserScan, callback):** suscribe el nodo a los datos del sensor láser que el robot publica en el tópico "/scan". Cada vez que llegan nuevos datos del láser, se ejecuta la función *callback* para analizarlos.
  - **rospy.spin():** mantiene el nodo activo y a la espera de datos continuamente. Sin esta función, el nodo se cerraría inmediatamente después de iniciarse.

Asegúrate de que el archivo es ejecutable:

```
# chmod +x ~/catkin_ws/src/collision_detector_pkg/scripts/collision_detector.py
```

Lanza el script :

```
# rosrun collision_detector_pkg collision_detector.py
```

Después de agregar el script, vuelve a compilar tu espacio de trabajo:

```
# cd ~/catkin_ws
# catkin_make
# source devel/setup.bash
```

Parte 5: Prueba del sistema

Controla el robot con el teclado y dirige el TurtleBot3 hacia la "unit_box". Observa en la terminal si se detecta la colisión.

Preguntas:

- ¿Cómo podrías mejorar la precisión de la detección de colisiones?

Parte 6: Ampliación de la funcionalidad del robot (I)

Amplía el código de detección de colisiones que desarrollaste previamente para que el robot gire automáticamente hacia la izquierda cuando **detecte** un obstáculo a menos de **0.2 metros**.

Requisitos:

- Debes añadir un publicador que envíe comandos de velocidad angular (giro) al robot utilizando el mensaje llamado `Twist`.
- El robot debe iniciar un giro cuando detecte un objeto cercano y dejar de girar cuando no haya peligro de colisión.
- Controla el robot con el teclado usando `teleop`_twist_keyboard, como en momentos anteriores, y observa cómo responde automáticamente cuando se encuentra cerca de un obstáculo.

... Observaciones sobre **`Twist`** ...

Para controlar el movimiento del TurtleBot3, debes utilizar el mensaje `Twist`, que pertenece al tipo de mensajes de ROS `geometry_msgs`. Este mensaje permite controlar tanto la velocidad lineal (movimiento hacia adelante o hacia atrás) como la velocidad angular (giro). El mensaje tiene dos componentes principales:

- **linear**: Controla la velocidad en las tres direcciones (x, y, z). Para nuestro robot, normalmente solo se utiliza el eje x para avanzar o retroceder.
- **angular**: Controla la velocidad de giro en torno a los tres ejes. En nuestro caso, utilizaremos el eje z para hacer que el robot gire hacia la izquierda o derecha.

En esta ampliación, basta con que modifiques el valor de **angular.z** para que el robot gire automáticamente cuando detecte una colisión.

Parte 7: Ampliación de la funcionalidad del robot (II)

Añadir un segundo objeto al escenario y modificar el código para que el robot pueda detectar y evitar **múltiples obstáculos** de forma autónoma. Esto permitirá que el robot navegue entre los objetos, reaccionando de manera adecuada al detectar colisiones con cualquiera de ellos.

1. **Añade un segundo objeto al escenario:**

    a. En Gazebo, inserta manualmente otro objeto al escenario, además de la "unit_box". Puedes elegir otro objeto, como un cilindro o una caja adicional.
    b. Posiciona este segundo objeto en una parte diferente del entorno, donde el robot pueda moverse entre ambos obstáculos.
    c. El escenario ahora debe incluir dos objetos que el robot debe detectar y evitar al navegar. Asegúrate de que ambos objetos estén dentro del alcance del sensor láser del TurtleBot3.

2. **Modifica el código para que el robot detecte y evite ambos objetos**.

    En lugar de evaluar solo la distancia mínima frente al robot, ajusta el código para que el robot pueda reaccionar a múltiples objetos a su alrededor.

    Requisitos:

    a. Debes procesar los datos del láser para detectar obstáculos en diferentes direcciones y no solo en el frente.
    b. Si el robot detecta un objeto a su derecha o izquierda, debe girar en la dirección contraria.

c. Si detecta un objeto directamente al frente, debe detenerse o girar automáticamente para evitarlo.

3. Prueba los cambios:

Conduce el TurtleBot3 usando el teclado y observa cómo el robot detecta y evita ambos objetos. Haz que el robot se acerque a los dos objetos desde diferentes ángulos para comprobar si responde correctamente a colisiones tanto al frente como a los lados.

Solución Parte 5:

Para mejorar la precisión de las colisiones se podrían hacer 2 cosas:

a) Ajustar el umbral de distancia:

Para ajustar el umbral de distancia en el código, solo necesitas modificar la condición que detecta la colisión basándose en la distancia mínima obtenida por el sensor láser. Aquí te doy un código donde puedes cambiar fácilmente este umbral (la distancia mínima antes de considerar que el robot está en riesgo de colisión).

```
# Umbral de distancia ajustable (en metros)
DISTANCIA_UMBRAL = 0.5 # Cambia este valor para ajustar el umbral
...
def callback(scan_data):
        min_distance = min(scan_data.ranges) # Detecta la distancia mínima
        if min_distance < DISTANCIA_UMBRAL: # Si la distancia es menor que el umbral
            rospy.loginfo("¡Colisión detectada! Distancia mínima: %.2f m" % min_distance)
```

b) Ajustar el promedio de las distancias:

```
#!/usr/bin/env python3
import rospy
from sensor_msgs.msg import LaserScan

def callback(scan_data):
    # Seleccionamos un rango de ángulos al frente del robot para promediar las distancias
    # Tomamos los primeros y últimos 10 puntos del láser (frente)

    front_angles = scan_data.ranges[0:10] + scan_data.ranges[-10:]

    # Filtramos valores 'inf' que representan la ausencia de detección en algunas direcciones
    filtered_distances = [d for d in front_angles if d < float('inf')]

    if len(filtered_distances) > 0:
        # Calculamos el promedio de las distancias
        avg_distance = sum(filtered_distances) / len(filtered_distances)

        rospy.loginfo(f"Distancia promedio: {avg_distance:.2f} metros")

        # Si la distancia promedio es menor a 0.2 metros, se considera una colisión
        if avg_distance < 0.2:
            rospy.loginfo("¡Colisión detectada!")
    else:
        rospy.loginfo("No se detectaron objetos al frente.")
```

```
def collision_detector():
  rospy.init_node('collision_detector', anonymous=True)
  rospy.Subscriber("/scan", LaserScan, callback)
  rospy.spin()

if __name__ == '__main__':
  try:
    collision_detector()
  except rospy.ROSInterruptException:
    pass
```

Explicación de los cambios:

1. **Selección de ángulos frontales**:
   a. En lugar de usar el valor mínimo de todas las distancias (min(scan_data.ranges)), seleccionamos un conjunto de puntos que corresponden a la vista frontal del robot.
   b. **front_angles = scan_data.ranges[0:10] + scan_data.ranges[-10:]**:
   Esto toma los primeros y últimos 10 puntos de los datos del LIDAR, que corresponden a la parte frontal del robot.
2. **Filtrado de valores inf**:
   a. Algunos puntos pueden no tener lecturas válidas y aparecer como inf (infinito). Filtramos esos valores usando una lista filtrada con *filtered_distances.*
3. **Cálculo del promedio**:
   a. Si tenemos distancias válidas (número mayor a 0), calculamos el promedio de esas distancias con *sum(filtered_distances) / len(filtered_distances).*
4. **Detección de colisión**:
   a. Si el promedio de las distancias es menor a 0.2 metros, consideramos que se ha detectado una colisión y mostramos un mensaje en la consola.

Solución Parte 6:

En este caso, se hace uso del mensaje *geometry_msgs.msg*, con lo cual deberá crear un nuevo paquete para cubrir este apartado que incluya esa dependencia.

```
#!/usr/bin/env python3
import rospy
from sensor_msgs.msg import LaserScan
from geometry_msgs.msg import Twist  # Para enviar comandos de movimiento

def callback(scan_data):
  min_distance = min(scan_data.ranges)  # Detecta la distancia mínima
  if min_distance < 0.2:  # Si el objeto está a menos de 0.2 metros
    rospy.loginfo("¡Colisión detectada! Girando...")
```

```
        # Creamos un mensaje de giro
        twist_msg = Twist()
        twist_msg.angular.z = 0.5  # Velocidad de giro hacia la izquierda
        pub.publish(twist_msg)  # Publicamos el comando de giro
    else:
        twist_msg = Twist()  # Detenemos el giro si no hay colisión
        pub.publish(twist_msg)

def collision_detector():
    global pub
    rospy.init_node('collision_detector', anonymous=True)
    rospy.Subscriber("/scan", LaserScan, callback)
    pub = rospy.Publisher('/cmd_vel', Twist, queue_size=10)  # Publicador para el movimiento
    rospy.spin()

if __name__ == '__main__':
    try:
        collision_detector()
    except rospy.ROSInterruptException:
        pass
```

- **Explicación de las modificaciones**:
    - **Publicador de comandos de movimiento** (/cmd_vel): Hemos añadido un publicador que envía comandos de velocidad angular (giro) al robot. Esto se hace utilizando el mensaje Twist, que controla la velocidad lineal y angular del robot.
    - **Giro automático**: Cuando se detecta que la distancia a un objeto es menor de 0.2 metros, el código envía un comando para que el robot gire hacia la izquierda (twist_msg.angular.z = 0.5). Si no hay colisión, el robot deja de girar.

- **Prueba**:
    - Controla el robot con el teclado como antes, pero esta vez, cuando el TurtleBot3 detecte una colisión con la "unit_box" u otro objeto, automáticamente girará a la izquierda para evitar el choque.

Solución Parte 7:

```
#!/usr/bin/env python3
import rospy
from sensor_msgs.msg import LaserScan
from geometry_msgs.msg import Twist

# Ajuste del umbral de distancia
threshold_distance = 0.5  # Cambia este valor según tu necesidad (0.5 metros en este caso)

def callback(scan_data):
```

```
    # Extraemos las distancias del láser
    left_distance = min(scan_data.ranges[0:30])  # Distancia en el lado izquierdo
    front_distance = min(scan_data.ranges[30:150])  # Distancia al frente
    right_distance = min(scan_data.ranges[150:180])  # Distancia en el lado derecho

    # Lógica para evitar colisiones
    if front_distance < threshold_distance:  # Si hay un objeto al frente
        rospy.loginfo(f"¡Colisión al frente! Objeto a {front_distance:.2f} metros. Girando a la derecha.")
        twist_msg = Twist()
        twist_msg.angular.z = -0.5  # Gira a la derecha
        pub.publish(twist_msg)
    elif left_distance < threshold_distance:  # Si hay un objeto a la izquierda
        rospy.loginfo(f"¡Colisión a la izquierda! Objeto a {left_distance:.2f} metros. Girando a la derecha.")
        twist_msg = Twist()
        twist_msg.angular.z = -0.5  # Gira a la derecha
        pub.publish(twist_msg)
    elif right_distance < threshold_distance:  # Si hay un objeto a la derecha
        rospy.loginfo(f"¡Colisión a la derecha! Objeto a {right_distance:.2f} metros. Girando a la izquierda.")
        twist_msg = Twist()
        twist_msg.angular.z = 0.5  # Gira a la izquierda
        pub.publish(twist_msg)
    else:
        twist_msg = Twist()  # Detenemos el giro si no hay colisión
        pub.publish(twist_msg)

def collision_detector():
    global pub
    rospy.init_node('collision_detector', anonymous=True)
    rospy.Subscriber("/scan", LaserScan, callback)
    pub = rospy.Publisher('/cmd_vel', Twist, queue_size=10)
    rospy.spin()

if __name__ == '__main__':
    try:
        collision_detector()
    except rospy.ROSInterruptException:
        pass
```

Descripción del Código:

1. **Distancias Lateral y Frontal**:
    a. Se obtienen las distancias al objeto en tres direcciones: izquierda (left_distance), frente (front_distance) y derecha (right_distance).
    b. Se utilizan segmentos del array ranges del mensaje LaserScan para capturar estas distancias.
2. **Lógica de Evitación**:

   a. Si el robot detecta un objeto a menos de threshold_distance en cualquiera de las tres direcciones, ejecutará un giro hacia la dirección opuesta para evitar la colisión:
      i. Si hay un objeto al frente, gira a la derecha.
      ii. Si hay un objeto a la izquierda, también gira a la derecha.
      iii. Si hay un objeto a la derecha, gira a la izquierda.
   b. Si no hay objetos cerca, se envía un mensaje para detener el movimiento.
3. **Publicación del Comando**:
   a. Los comandos de movimiento se publican a través del tópico /cmd_vel utilizando mensajes del tipo Twist.