TECHIN 516: Robotic Sensing And Mobility

Lab 1: Sensors in ROS

DATE: Jan 12th. 2023

Lecturer: Queenie Qiu, John Raiti. Student: ***Shucheng Guo***

1 Gazebo introduction

Gazebo is a high-fidelity 3D simulator stemming from the need to simulate robots in various environments and conditions. Execute the following line in a terminal:

```
$ roslaunch gazebo_ros empty_world.launch world_name:=worlds/cafe.world
```

As you may see in this example, Gazebo simulates a cafe with different objects, from simple shapes to complex models that are governed by dynamic and static properties as well as physics constraints. Now let's add a robot.

1.1 Spawning a robot in Gazebo

This method uses a small python script called spawn_model to make a service call request to the gazebo_ros ROS node (named simply "gazebo" in the rostopic namespace) to add a custom URDF into Gazebo. The spawn_model script is located within the gazebo_ros package. Open a new termianl and use this script in the following way:

```
$ roscd turtlebot3_description/urdf/
$ rosrun gazebo_ros spawn_model -urdf -file turtlebot3_burger.urdf.xacro
 -x 0 -y 0 -z 1 -model TURTLEBOT3_MODEL
```

Note: Normally, we can take advantage of the roslaunch command as we are loading the gazebo world, and add the robot spawning portion to the launch file. The resulting added code would look like the following lines:

```
<param name="robot_description" command="$(find xacro)/xacro --inorder $(find
turtlebot3_description)/urdf/turtlebot3_$(arg model).urdf.xacro" />

<node pkg="gazebo_ros" type="spawn_model" name="spawn_urdf" args="-urdf
-model turtlebot3_$(arg model) -x $(arg x_pos) -y $(arg y_pos) -z $(arg
z_pos) -param robot_description" />
```

Figure 1: illustration of the launch file.

1.2 Creating object models in Gazebo

1. You can add additional objects and shapes into your simulated environment. On the top of the right panel, click on the cylinder shape to add a pillar object into the world. When you right-click on the object you can modify its properties, like dimension and the collision area. Select edit model, the open link inspector. In both the visual and collision tabs, add the following dimensions in the Geometry section: radius = 0.25m, length = 1m. An illustration is shown below:

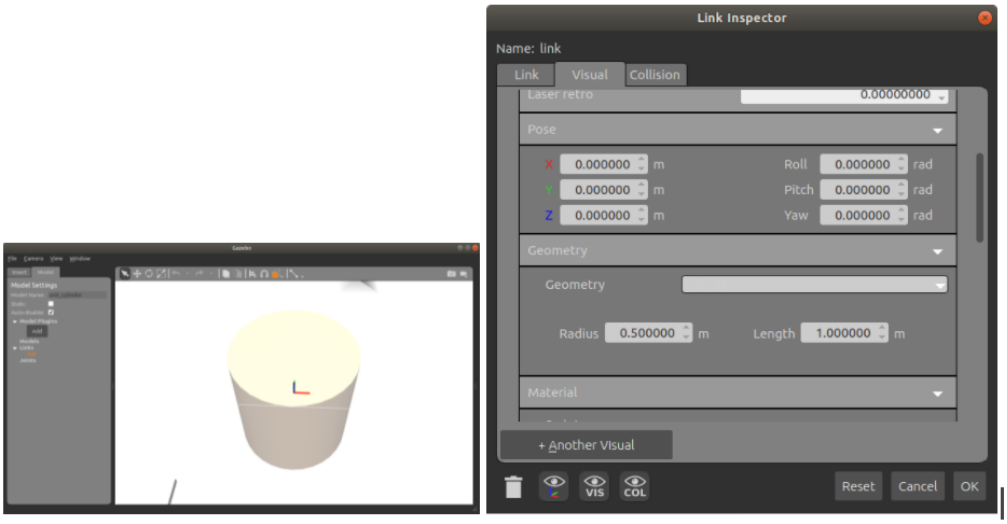

Figure 2: An illustration of creating a model in gazebo.

2. Once you have created your pillar model, you can save it (full path: /catkin_ws/src/turtlebot3_simulations/turtlebot3_gazebo/models). This will allow you to insert the object with its customized properties in new worlds. If you open this folder you will see a new folder called like the object you created, and it will contain two files: model.config and model.sdf

Deliverable: Take a screenshot of the modified cafe.world with your pillar in it.

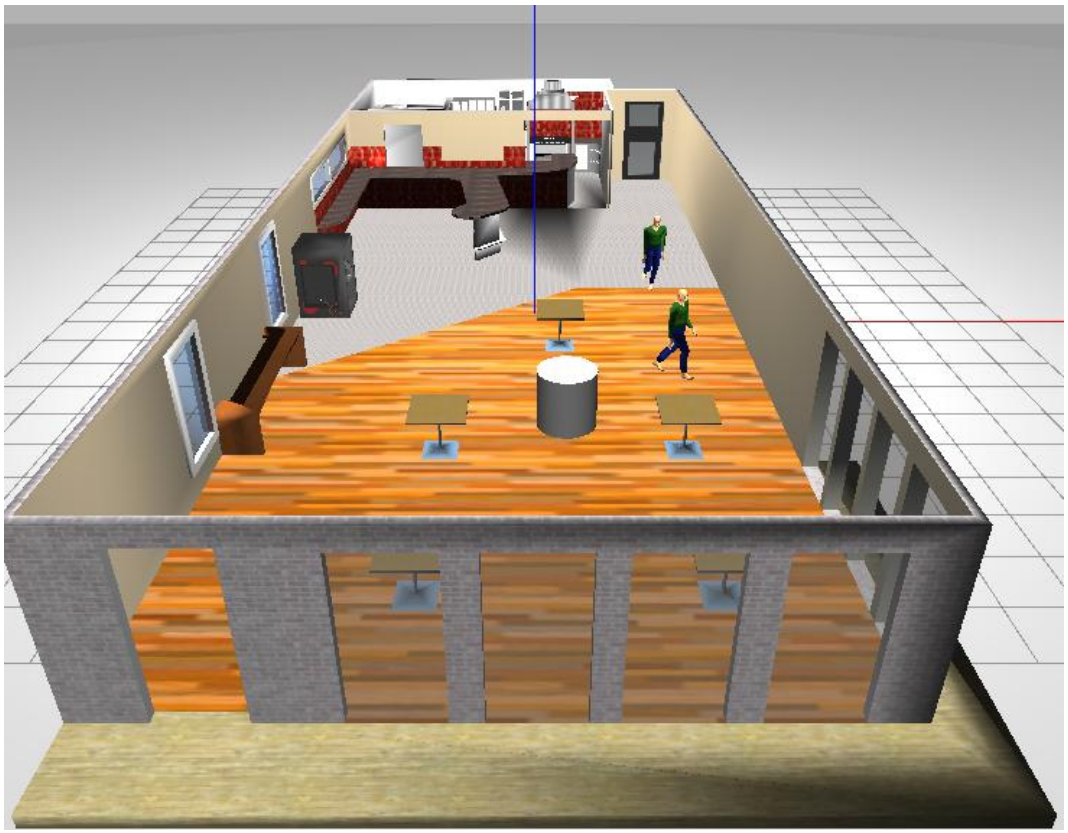

Figure 3: Modifying `cafe.world` with a pillar.

2 Setting up the working environment

First, we want to set up the simulation environment to work with the turtlebot3. We will be using a simple world with four walls where the turtlebot3 burger can interact.

1. Select the turtlebot3 model to be burger and use the 'roslaunch' command to spawn a gazebo world with a turtlebot3 in it.

   ```
   $ export TURTLEBOT3_MODEL=burger
   $ roslaunch turtlebot3_gazebo turtlebot3_stage_1.launch
   ```

2. Open a new terminal and try to visualize the robot and its sensors in RViz.

   ```
   $ rviz rviz
   ```

3. Then click on the add button, select RobotModel and add it to the RViz panel.

   Note: At this point, you will find errors related to failure to map from odom to base_link, wheel_right, wheel_left. If you take a closer look at the launch file for the gazebo world, and inspect what nodes are brought up by it, you will realize that the robot that has been spawned is only the footprint of one. To interact with variables like velocity of sensor information, we need to bring up all the necessary topics and nodes through the robot_state_publisher.

4. In another terminal, type the following command to visualize the transforms tree. You will notice there is only a transformation between the /odom reference frame and the /base_footprint. The latter represents the point on the floor where the robot is supposed to occupy.

```
$ rosrun rqt_tf_tree rqt_tf_tree
```

5. In a new terminal you will bring up a full robot model with all its transformations.

```
$ roslaunch turtlebot3_bringup turtlebot3_remote.launch
```

After that, you can retry the rqt_tf_tree command and expect to see the following:

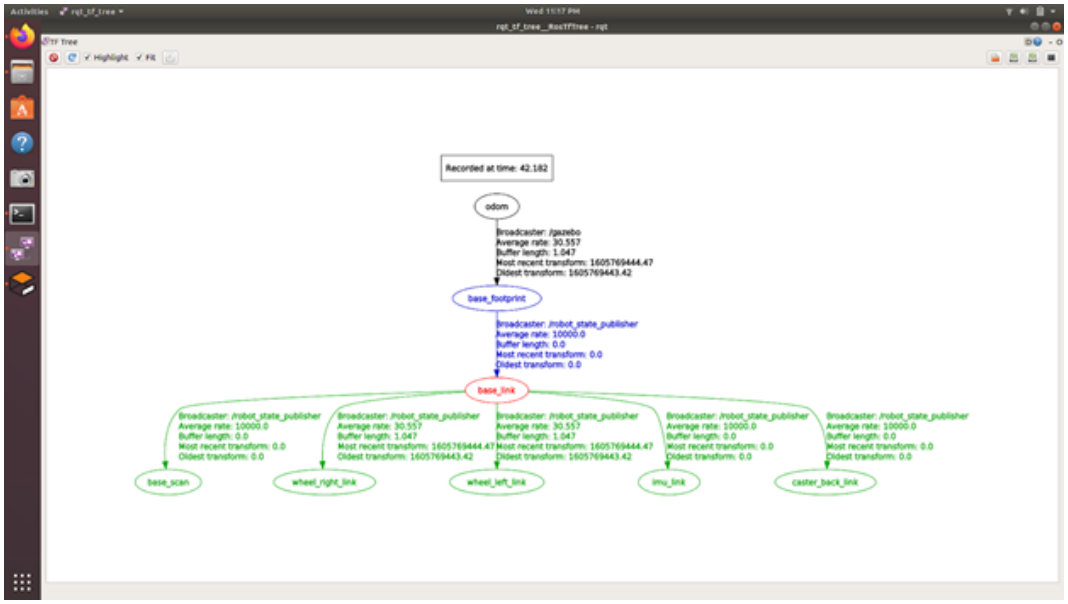

Figure 4: rqt_tf_tree.

Note: If your rviz visualization has no robot, you can change the fixed frame to "odom", at this point the robot should be visualized in RViz.

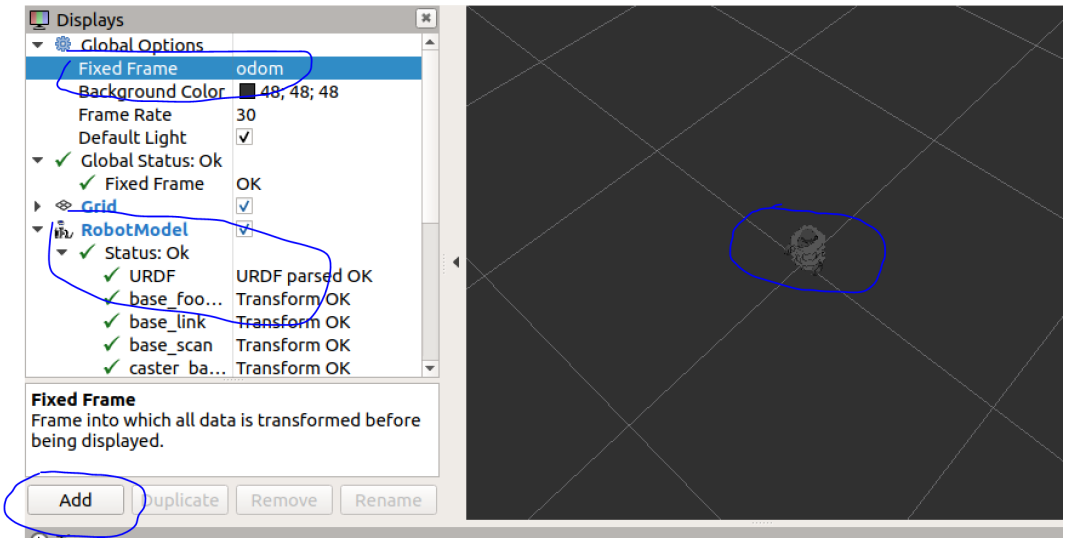

Figure 5: rviz visualization.

Now that the robot is fully brought up and functioning in simulation, we will go through the examples available with the turtlebot3 package.

3 Running turtlebot3 examples

You have spawned the robot in a Gazebo simulation, just in case you closed them, you need to open two terminals and type:

For one terminal:

```
$ roslaunch turtlebot3_gazebo turtlebot3_stage_1.launch
```

For another terminal:

```
$ roslaunch turtlebot3_bringup turtlebot3_remote.launch
```

Then explore the different examples available from the turtlebot3 metapackage. Additional information can be found in this link:

1. Move using Interactive Markers: this example uses input coming from rviz. An overlayed marker consisting of arrows and a ring surrounding the robot's position are used to modify the cmd_vel topic with linear and angular velocity changes. **Note**: you need to add the interactive markers to the panel.

```
$ roslaunch turtlebot3_example interactive_markers.launch
$ rosrun rviz rviz -d `rospack find turtlebot3_example`/rviz
/turtlebot3_interactive.rviz
```

As you manipulate the interactive markers in rviz, you may be able to use new terminals to inspect the active nodes and topics (through rqt_graph) or to echo the changes done to the cmd_vel topic (rostopic echo /cmd_vel) as you visualize both in rviz and gazebo the result of these interactions.

Deliverables:

(a) Take a screenshot of the resulting rqt_graph.

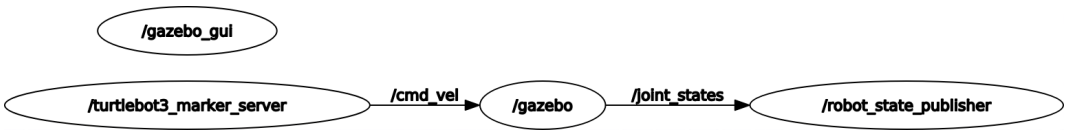

Figure 6: Resulting `rqt` graph.

(b) What information is flowing between the nodes?

Answer: Velocity and direction changes in the topic `/cmd_vel` are the information exchanged between the nodes.

(c) Describe, in detail, what this information is used for within the robot.

Answer: The information show linear (x, y) and angular (z) changes in both direction and velocity. Positive values in x and y drive the robot forward, while those in z to the left; negative values in x and y drive the robot backward, while z to the right. Values indicate the magnitude of the velocity change, in the unit of meters per second.

(d) Describe the resulting behavior of the robot when you use either the arrows or the ring on the interactive marker. Copy one of the echo messages for velocity.

Answer: Velocity only changes in one plane if only the arrow or the thing is pressed, meaning there is only one non-zero value on x, y and z axes. For one click on the arrow, the echoed messages are shown below:

```
linear:
  x: -0.005202962551265955
  y: 0.0
  z: 0.0
angular:
  x: 0.0
  y: 0.0
  z: 0.0
```

2. Point Operation: in this example the user specifies an input corresponding to x | y | z. The first two numbers are distance numbers and the third is an angle value (-180 to 180), referenced to the world origin in Gazebo (the robot spawned in the origin (0 | 0 | 0). The symbol | represents a space bar character, a correct command for the robot would be “1 1 90”.

```
$ roslaunch turtlebot3_example turtlebot3_pointop_key.launch
```

Tip: You can reset the robot's position to its original pose in Gazebo:
Edit -> Reset Model Poses (Ctrl + Shift + R)

Deliverables:

(a) Give the robot a command to reach the point x: 1 y: 1.5 w: -90. Describe how the robot is reaching the goal and what do you observe while watching the robot. Does it match position and orientation at the same time? Or does it do it in a specific sequence?

Answer: To reach the goal of `(1, 1.5, -90)`, from the top view, the robot walks 1.5 units left and 1 up to its northwest, and then turns 90 degrees, facing east.
After receiving the command, the robot walks to the designated point `(x, y)` before it turns the direction toward that described in `z`.

(b) Edit the initial position of the robot to be different from the origin (x:0 y:0 w:0) using the Gazebo interface. Run the same command as before to reach the point x:1 y:1.5 w:-90.

i. Describe the motion of the robot and with respect to which reference frame (the world's on Gazebo or its own reference frame).
Answer: The origin of the robot is set to `(-1, -1.5)`, which is 1.5 units right and 1 down. The goal is `(1, 1.5)`, which is 1.5 units left and 1 up from `(0, 0)`.
To reach the destination, the robot walks 3 units left and 2 up to the same point as in the last question, and then turns east.

ii. Determine the transformation between the Gazebo origin and the robot's new initial position. How would the instruction look (x: <value> y: <value> w: <value>) if it was from the robot's own reference frame. Tip: you can use vector representation or a diagram.
Answer: As explained in the last answer, the robot walks left for 3 units and up for 2, while rotating the same angle.
From its own reference frame, the instruction would be `(x: 2, y: 3, z: -90)`.

iii. Discuss the relevance of absolute and relative transformations.
Answer: Absolute transformation is the positional and angular change from the Gazebo origin `(0, 0, 0)`, which relative transformation is one compared to the robot's initial position.
The coordinates we command to transform are absolute transformations, as they're relative to the Gazebo origin; the relative one is calculated based on two transformations, current and final.
They are not necessarily related, but the calculation of the relative transformation involves the absolute transformations.

3. Patrol: this example requires to run two nodes, a server and a client. The client can run through roslaunch; the server node through the rosrun command. In the client terminal, the user can specify a command to determine the mode of patrol (square, triangle, circle), the distance parameter (side or radius), and the number of cycles in the patrol pattern.

```
$ roslaunch turtlebot3_example turtlebot3_client.launch
$ rosrun turtlebot3_example turtlebot3_server
```

Note: These patrolling movements are performed using odometry information to know how much the robot has moved. Notice the accumulation of inaccuracy in the patrol shape as the distances or the number of cycles are larger.

Tip: In the event of colliding with the environment in Gazebo, you can reset the robot's position Edit -> Reset Model Poses (Ctrl + Shift + R)

Deliverables:

(a) Inspect the code for the turtlebot3_server node:

i. What are the topics used in the subscribers of the code and what are the message types?

A. Topic: `joint_states` Type: `JointState`;

B. Topic: `odom` Type: `Odometry`;

ii. Where is the data used in the code?
Answer: All of the three arguments are passed in an instance of the class object `Turtlebot3Action`.
The patrol mode `mode` and the distance parameter `area` are both called in the method `execute_cb`.
The number of cycles to move `count` is called in the method `go_front`.

(b) Describe how the code can perform different shapes (turn different angles). How is the information from the sensors used?
Answer: By passing the `mode` argument that specifies the patrol pattern, we can determine the robot's angles in motion. Square is `s`, Triangle `t` and Circle `c`.
Odometry information is passed in and recorded by the method `get_odom`, and then converted to the current position, with which the last movement is calculated, by the method `get_state`.

4 Sensors

In this section, we will take a closer look at the information coming from odometry and the LiDAR sensor. We will use the patrol example and create a node which subscribes to the /odom topic to plot the position of the robot throughout the patrol behavior. Additionally, we will create a new node that subscribes to the /scan topic, and moves the robot towards an object that is close by.

4.1 Creating nodes to visualize sensor data from the odometry topic

Create a node to subscribe to the odometry topic to create an x-y plot of the robot's reported position. You will find a plot_odom.py file in the lab1 folder. This code is currently incomplete.

You will need to fill out the TODOs to make it work. You may need to move the file to a ros package (recommended turtlebot3_examples/nodes) and make the file to be executable before rosrunning it (using the chmod +x command in the terminal).
Tip: You may need to install the necessary packages to run the plot_odom.py file correctly.

```
$ sudo apt install python-numpy python-matplotlib
```

Deliverables:

1. Run your patrol example to complete one cycle of 1m square. (the parameters to fill in the client program would be: s 1 1)

   (a) Describe the behavior of the robot in terms of inaccuracies associated with the odometry. What do you observe about the ending position and orientation of the robot?

   Answer: The robot deviates from its path of patrol after making turns, and walking distances. The more cycles it patrols, the more deviation it makes. Typically with the side/radius set to 1m, the robot hits the wall as of the third cycle of patrol. The hypothesis is that the inaccurate information obtained from odometry leads to the gaps in the calculations of the angle to turn and the length to walk.

   (b) How does the shape traversed by the robot compare to the original square input instructions?

   Answer: The shape traversed by the robot is more similar to a parallelogram, rather than a square. Due to the non-right angles and unequal sides, it doesn't safisfy the requirements as a square.

   (c) In your own words, describe what portion of the movement is creating the most inaccuracies.

   Answer: Based on my observation, there are two parts of the movement that create the most inaccuracies:

      i. One is controling the angle when making turns at the vertices of the patrol shape, e.g. when patroling in squares, it's obvious that the robot isn't walking right angles as it should.

      ii. The other is controling the distance of different sides of the shape, e.g. when patroling in squares, it can be easily observed in Figure 7 that the horizontal sides are longer than the vertical ones.

   (d) Save the x-y plot and include it in your write-up.

   Answer: Refer to the figure 7 for the instruction required in the question, and compare it with the figure 8, where the robot walks two times the distance before hitting the wall and stopping the plot.

2. Select one of the shapes for patrol, and run it with a number of cycles greater than 1 (e.g. t 1 3). Discuss your observations of accumulation of errors and its impact in the shape of the patrol.

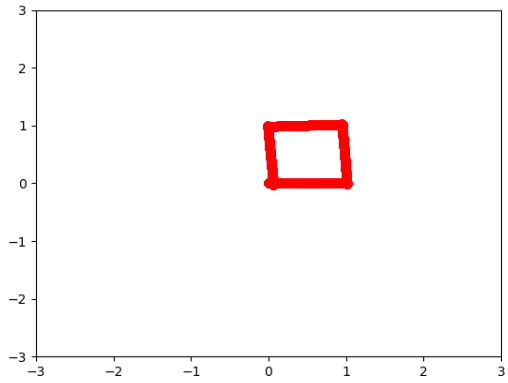

Figure 7: Plotting the patrol `s 1 1`.

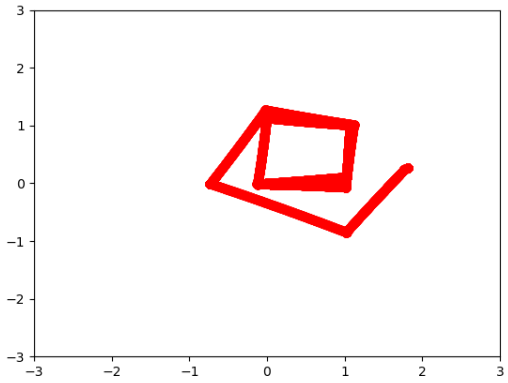

Figure 8: Plotting the patrol `s 1 2`.

(a) Before you input the parameters to the client program, open a new terminal and record a rosbag containing the information from relevant topics. You will upload this file with your assignment submission.

```
$ rosbag record /odom /cmd_vel /tf /scan -O <your_name>_patrol.bag
```

Answer: The required bag file is attached in the submission.

(b) Save the x-y plot and include it in your write-up.

Answer: Based on the observation, the error will accumulate as the robot makes more turns and patrols more cycles. At first, it only affects the orientation toward which it walks. But later when it's away from the origin, the path will be so deformed that it looks nothing as planned, and sometimes it hits the wall before completing the patrol.

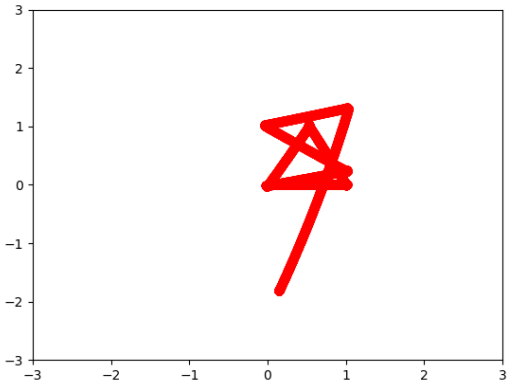

Figure 9: Plotting the patrol `t 1 3`.