TUM Institute for Cognitive Systems (ICS)
RoboCup@Home course

Tutorial 3: Robot navigation and motion planning

M.Sc Fengyi Wang

12.05.2023

This tutorial provide the basic knowledge and skills needed to navigate a robot in a simulator. A set of four exercises will be requested. These exercises will be evaluated for the grade. Please type the commands instead of copying them from the PDF to prevent bad string formating in the linux terminal.

Exercises

The objective of this tutorial is to navigate and manipulate the Tiago robot in a simulated environment. First, we will start by manually commanding the robot using the input from your keyboard to create the map of the world. Then, we will complete two packages that localize, navigate and manipulate the robot.
This tutorial will be completed in the Gazebo simulator.

Figure 0.1: More information about the Gazebo project in `http://gazebosim.org`

1 Exercise 1: Generate maps for two worlds using Tiago / HSRB

Creating a map for robot is an essential step in robot navigation because it enables the robot to to estimate its position within the environment relative to the map's coordinate frame and plan its movements more efficiently.
"gmapping" package allows to build 2D occupancy grid maps with laser sensor data and robot position.
`wiki.ros.org/gmapping`

1.1 Generate the map for the default world

First of all, open a terminal and source TIAGo's public simulation workspace, and split it into two:

```
cd /tiago_public_ws/
source ./devel/setup.bash
```

In one terminal, launch the simulation with mapping function.

```
roslaunch tiago_2dnav_gazebo tiago_mapping.launch public_sim
   :=true
```

In another terminal, launch the key_teleop node.

```
rosrun key_teleop key_teleop.py
```

Wait until the robot arm is fully tucked for safety reason. Then, by pressing the arrow keys in this terminal, drive TIAGo around the world. The map being generated will be shown in RViz.

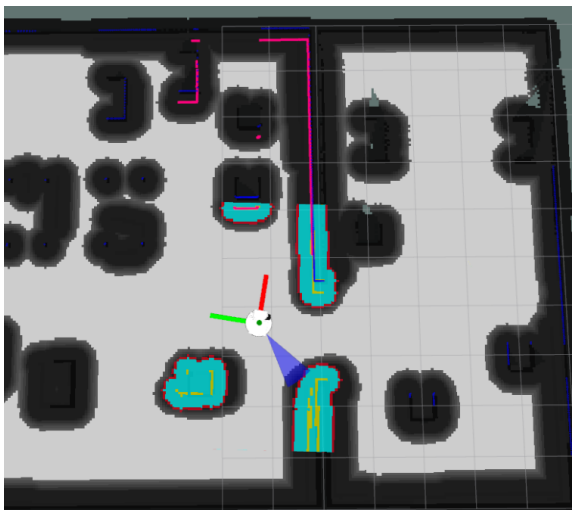

Figure 1.1: A part of the completed map.

Press 'q' in the key_teleop terminal and save the map as follows

```
rosservice call /pal_map_manager/save_map "directory: ''"
```

By default, the service call will save the map in the following folder

```
\~{}/.pal/tiago\_maps/config
```

1.2 Generate the map for another world

If the file you are launching specifies arguments that different from the default setting, you can do so using an identical syntax to ROS remapping arguments. Launch the simulation with another world by remapping the "world" argument to "tutorial_office" and generate its map.

```
roslaunch tiago_2dnav_gazebo tiago_mapping.launch public_sim
   :=true world:=tutorial_office
```

2 Exercise 2: Localization

In simulation, the robot can be accurately spawned at a predetermined location, but in reality, the pose of mobile robot is unknown before the localization process. To move from one point to another without collision, a robot needs to know its current position and orientation relative to the environment.
“amcl” package implements probabilistic localization. The employed technique is called Monte Carlo Localization. For more information about amcl package: `wiki.ros.org/amcl`
In this exercise, the robot spawns at a different location by set the argument `lost` to `true`. The robot poses in Gazebo simulator and in RViz will be different.
By default the map launched is **$(env HOME)/.pal/tiago_maps/configurations/$(arg world)** where `world` is by default `small_office`. Here, you have to use the map that you created in exercise 1. The robot will be spawn at a different location. Launch the simulation and remapping the `map` option to your own map folder with the following command:

```
roslaunch tiago_2dnav_gazebo tiago_navigation.launch
   public_sim:=true lost:=true map:=<your_map_folder>
```

Next, you need to finish the code inside `tiago_localization_node.cpp`. This node must provide:

- a **Subscriber** used to receive the estimated pose published by `\amcl` node.
- a **Publisher** used to publish velocity commands of type `geometry_msgs::Twist` to topic `/mobile_base_controller/cmd_vel` to control the robot.
- a **ServiceClient** used to initilize the localization by calling `/global_localization` service with an empty srv of type `std_srvs::Empty`.

Use the provided template `tiago_localization_node.cpp` to finish the node by looking for all `TODO`. Basically, what you need to do is:

1. Complete the publisher, subscribers and service client in the **initialize()** function.
2. Load the parameters of this node form the ros parameter. This time we require the spinning speed of the robot and a threshold value as termination condition.
3. Call the `/global_localization` service with the ServiceClient you created in the **initialize()** function.
4. Next we will complete the callback function of the subscriber. Check if the message in the topic was published within 0.5 second.

5. We implement a naive termination condition, which is to check the sum of all items of the covariance in the message. If the sum of covariance is below the threshold, localization is complete and shutdown the node.

Test your package by launching `tiago_localization.launch`.
Note that the localization is not always successful, tune the parameters to get higher successful rate and fast termination.

3 Exercise 3: Navigate the Map

Complete the package `tiago_move` to navigate Tiago in the "small_office" world. The robot should continuously and autonomously move between waypoints A,B,C and loop as shown in the picture (the coordinates of these points don't have to be very precise).
Launch the simulation with the following command:

```
roslaunch tiago_2dnav_gazebo tiago_navigation.launch
  public_sim:=true
```

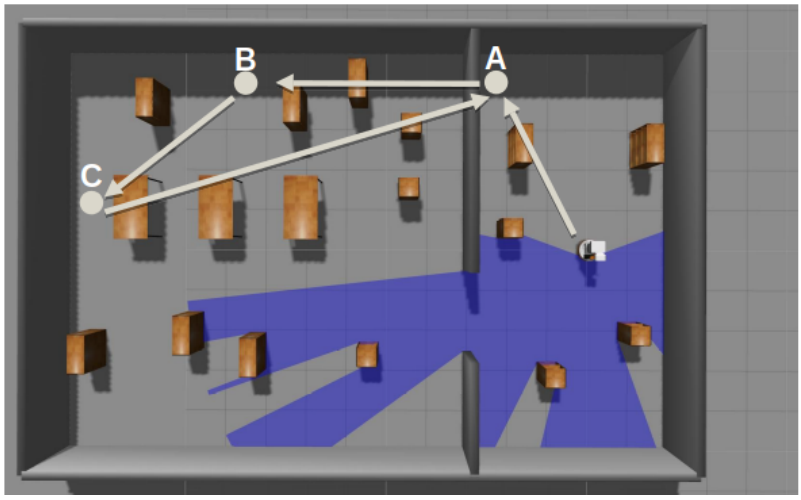

Figure 3.1: Illustration of the patrol task.

3.1 Find goal coordinates on the map

Use rviz button Publish Point to find the coordinates of the mouse on the map. The current coordinates is shown on the bottom left corner of RViz.

3.2 Autonomous Navigation

Now, the coordinates of waypoints A,B and C are known. For autonomous navigating the robot between these three points, you will complete the package `tiago_move`.
The goal of this package is to robustly navigate the robot to a user-provided goal while avoiding collisions with sensed obstacles using ROS navigation stack.
`http://wiki.ros.org/navigation`
To send goals to the navigation stack with an action client, you need to finish the code inside `tiago_move_node.cpp` and `tiago_move_node.h`. (Hint: Look for the »»TODO:Exercise3

tags inside the code).

To create a robust action client for robot navigation:

1. In the **header** file, declare a **SimpleActionClient** that communicate with the `move_base` action server. The public member function of `actionlib::SimpleActionClient` class used in the following instruction can be found at : actionlib::SimpleActionClient API
2. In the **header** file, declare a vector called `nav_goals` and store all three waypoints in message type `move_base_msgs::MoveBaseGoal`.
3. In the **initialize** function, waits for the ActionServer to connect to this client for 5 seconds with `waitForServer` function of `SimpleActionClient` class.
4. In the **initialize** function, load the 3D coordinates of three waypoints from the ROS parameter server.
5. Store three waypoints in the vector `nav_goals` in message type `move_base_msgs::MoveBaseGoal`. We provide a auxiliry function `createGoal` to convert coordinates and to `move_base_msgs::MoveBaseGoal` message.
6. In the **main loop**, send the current goal to the action server with `sendGoal` function of `SimpleActionClient` class.
7. Blocks until this goal finishes with `waitForResult` function of `SimpleActionClient` class.
8. Get the state information for this goal with `getState` function of `SimpleActionClient` class and check if the state of goal is SUCCEEDED. Find the returned type on actionlib::SimpleActionClient API.

4 Exercise 4: Planning in Cartesian space with MoveIt!

We will keep using the code inside `tiago_move_node.cpp` and `tiago_move_node.h` for easier integration of navigation and motion planning. (Hint: Look for the »»TODO:Exercise4 tags inside the code).

1. Uncomment the `move_arm` function in `tiago_move_node.cpp` and `tiago_move_node.h`.
2. In the header file creates a **moveit::planning_interface::MoveGroupInterface** as a private member of `Controller` class for the move group `arm_torso` that you will control and plan for .
3. In the **initialize** function, load the parameters of this node form the ROS parameter. This time we require the 6D target pose of the end-effector.
4. Set the planner of your `MoveGroupInterface` in the **initialize** function with `setPlannerId` function. Avaible planner ID can be found in `ompl_planning.yaml` file in `ompl_planning.yaml` under the `config` folder in `tiago_moveit_config` package.

5. Set the reference frame of your target pose with `setPoseReferenceFrame` function in the **initialize** function.

   (Hint: The path of a ros package can be found by command)

```
rospack find <package name>
```

6. In the **move_arm** function, convert the argument of the function to a message of type `geometry_msgs::PoseStamped goal_pose`.
7. Set the target pose for the planner `setPoseTarget` function in the **move_arm** function.
8. Start the planning by calling member function `plan` and pass the `motion_plan` instance as argument
9. In the **move_arm** function, execute the plan by calling member function `move` of the `MoveGroupInterface` instance.
10. In the main loop, call the **move_arm** function at proper waypoint (when the robot reached waypoint "C") with a proper target pose of the robotic hand in Cartesian space to lift the table.

5 Delivery

Create a read-me file containing four sections, one for every exercise. Start every section with the name of the robot you used in that section, followed with the name of the package (or packages) required for every section. Then, write all the commands to compile, run and operate your solution. Compress the maps you generated in the exercise 1, the read-me file and the required packages to compile and run your solutions into a tar/zip file and name it as:

`Name_lastName_roboCupHome_tutorial3`

Send the files to: `robocupathome.ics@xcit.tum.de`

Pay attention to the following remarks:

- **DO NOT** copy all the Workspace into the zip file. Copy only the packages needed for the solution.
- Be clear when writing your read me file.
- Specify which robot you used in the first line of every section.
- Include the list of the packages needed for every solution.
- Include all the commands to compile, execute and operate your solutions.
- Follow the naming conventions specified in this Document.
- Make sure that your solution compiles and runs in a single try.
- Deliver your solution before the deadline **18th of May at 23:59**.