Python Applications for Robotics

Spring 2021

Instructor:	Zeid Kootbally	**RWA#3:**	Autonomous navigation.
Email:	zeidk@umd.edu	**Due date:**	04/24/2021

Contents

1 Introduction

In this assignment you need to map an environment, teleoperate the robot to go to different locations, and store these locations into a file using a ROS service. Two packages are provided to you:

- aws-robomaker-small-house-world consists of the gazebo world used for this assignment. This world is a modified version of the aws-robomaker-small-house-world world.
- The mapping package provided in class has been slightly modified and the newest version must be used for this assignment.

You will also need to write your own package which consists of:

- A service file (`.srv`).
- A Python file in which you will write the service client, the service server, and the code to write a YAML file with data you get from the topic **amcl_pose**.

Objective: Complete all the **Exercise** sections in this document.
Note: This document is long because I provide a lot of background. The only challenging part of this assignment is in Section 5.

Figure 1: Modified version of the small house world.

2 Group Assignment or Individual Assignment

This assignment will be reused in the final project and can (should) be done as a group. Each group can have up to 4 students. You are free to choose your teammates. Groups created for this assignment will stay the same for the final

project. If you do not want to perform this assignment as a group, you are free to do it on your own but you will have to do the final project on your own as well. Students not part of a group will be manually placed in groups (unless you want to go the solo route).
If you are given 2 weeks for this assignment it is because it requires two weeks to complete. You will have a bad time if you try to do it in one week.

2.1 Exercise

I will upload a spreadsheet on Google Drive. Please fill out the spreadsheet by Monday (4/11, 5 pm) to create your groups.

3 Mapping

The first thing you need in order to perform autonomous navigation is to create a map of the environment for static obstacles.
`start_mapping.launch` includes two other launch files, both located in the mapping package.

- `bringup.launch`: This file starts the Gazebo world, set the start position of the robot, and start RViz with a specific RViz configuration file (`bringup.rviz`).
- `turtlebot3_gmapping.launch`: This file was copied from the turtlebot3_slam package (located in *opt/ros/melodic/share/turtlebot3_slam/launch*). Copying a file from the Debian package into your own workspace allows you to edit the file. Remember that you are not allowed to edit files from the Debian package but you can copy those files to your workspace and edit them.

  This file starts the **turtlebot3_slam_gmapping** node from the gmapping package. **turtlebot3_slam_gmapping** subscribes to two important topics.

  - **scan**: this is the name of the topic that is publishing the laser data from the LIDAR mounted on your robot.
  - **tf**: this is the topic that publishes the robot frames.

  The **turtlebot3_slam_gmapping** node also sets some very important parameters on the parameter server:

  - **base_frame**: The frame attached to the mobile base (**base_footprint** in this case).
  - **odom_frame**: The frame attached to the odometry system (**odom** in this case).
  - **map_frame**: The frame attached to the map (**map** in this case).

  `turtlebot3_gmapping.launch` also loads parameters from `gmapping_params.yaml`. The latter is located in the mapping package and was copied from *opt/ros/melodic/share/turtlebot3_slam/config*. I copied this file into mapping to give you the ability to edit it if needed. You may want to edit some of the parameters described in this file:

  - **maxUrange**: This parameter sets how far your laser will be considered to create the map. Greater range will create maps faster. The downside its that consumes more resources.
  - **throttle_scans**: Number of scans to process. If you remove **throttle_scans** from this file then default is 1. A value of 1 for this parameter will process all scans. If you set the value of this parameter to 10 (for example), only 1 scan out of every 10 scans will be processed. You may want to edit this value if you want to lower resource consumption.
  - **particles**: Number of particles used by the particle filter implementing the Kalman Filter that computes the map. You can also play with this number to lower resource consumption.

  If you need more information about all these parameters, please visit: http://wiki.ros.org/gmapping.

3.1 Summary

We are using a custom launch file `start_mapping.launch` to start the **slam_gmapping** node from the gmapping package. This node subscribes to the Laser topic (**scan**) and the transform topic (**tf**) in order to get the data it needs to build a map. The generated map is published during the whole process on the **map** topic. Since we are subscribed to the **map** topic in RViz we are able to see the map being built.

3.2 Exercise

1. You may want to create a new catkin workspace or use the current one. If you use the current one, make sure you zip the current mapping package because it will be replaced with the new one.
2. In the catkin workspace, place the two packages that are provided.
3. Build your workspace and source the *devel* folder.
4. In `start_mapping.launch`:
   - Edit `<arg name="world_name" default="$(find turtlebot3_gazebo)/worlds/turtlebot3_world.world"/>` to point to the world used for this assignment (located in the package aws-robomaker-small-house-world).
5. In `bringup.launch` edit the following section to spawn the robot at position (3.831436, -1.234452) with a yaw of 0:

```
<arg name="x" default="-0.5" />
<arg name="y" default="-0.5" />
<arg name="yaw" default="-2.65" />
```

6. Map the environment by repeating the process seen in class:
   - Start everything needed for mapping: `roslaunch mapping start_mapping.launch`
   - Start teleop node.
   - Teleoperate your robot and map the inside of the house. This may take some time and requires computer memory.
   - Save your file in the *maps* folder in your package. Name it **small_house_map**. After saving your map, ensure 2 files were generated: `small_house_map.pgm` and `small_house_map.yaml`. The pgm file is the one that contains the occupancy data of the map and the YAML file contains some metadata about the map, such as the map dimensions, resolution, and the path to the pgm file.

4 Navigation

Once you have a map of the environment, you have to tell other nodes how to access this map so you can perform autonomous navigation. The map_server package provides the **map_server** node, which reads a map file from the disk and provides the map to any other nodes that request it via a ROS service. The **move_base** node (from the move_base package) requests the current map in which the robot is moving in order to perform path planning. The **amcl** node (from the amcl package) uses the map in order to figure out where in the map the robot is.

The `start_navigation.launch` file (in mapping) does the following:

- Loads the Gazebo world and spawn the robot in this world.

```
<include file="$(find mapping)/launch/bringup.launch">
  <arg name="world_name" value="$(arg world_name)" />
  <arg name="model" value="$(arg model)" />
```

```
  <arg name="x" value="$(arg x)" />
  <arg name="y" value="$(arg y)" />
  <arg name="yaw" value="$(arg yaw)" />
  <arg name="rviz_config" value="$(arg rviz_config)" />
</include>
```

- Starts the map server node, which points to the map file on disk.

```
<node pkg="map_server" name="map_server" type="map_server" args="$(arg map_file)" />
```

- Calls another launch file which will start the **amcl** node.

```
<include file="$(find mapping)/launch/start_amcl.launch">
  <arg name="initial_pose_x" value="$(arg x)" />
  <arg name="initial_pose_y" value="$(arg y)" />
  <arg name="initial_pose_a" value="$(arg yaw)" />
</include>
```

- Starts the **move_base** node by calling another launch file.

```
<include file="$(find turtlebot3_navigation)/launch/move_base.launch">
  <arg name="model" value="$(arg model)" />
</include>
```

4.1 Localization with amcl

During localization the robot generates many random guesses as to where it is going to move next. These guesses are known as particles and each particle contains a full description of a possible future pose. When the robot observes the environment with sensor readings, it discards particles that do not match with these readings, and generates more particles close to those that look more probable. By doing so, the robot ensures that most of the particles will converge in the most probable pose that the robot is in. In other words, the more the robot moves, the more data the robot will get from sensor readings, which results in more accurate localization. These particles are depicted by arrows (as seen in RViz). This is known as the Monte Carlo Localization (MCL) algorithm or particle filter localization.

The amcl (Adaptive Monte Carlo Localization) package provides the **amcl** node, which uses the MCL algorithm in order to track the localization of a robot moving in a 2D space. This node subscribes to the data of the laser (**scan** topic), the map (**map** topic), and the transformations of the robot (**tf** topic), and publishes the robot's estimated pose in the map. The estimated pose is published on the topic **amcl_pose**. On startup, the **amcl** node initializes its particle filter according to the parameters provided. Parameters for the **amcl** node can be found in the `start_amcl.launch` file

(located in the mapping package). Description of the topics, services, and parameters used by this package can be found at http://wiki.ros.org/amcl.

Some important parameters are:

- **min_particles**, **max_particles**: These parameters set the number of particles (min and max) that the filter will use in order to localize the robot. The more particles you use, the more precise the localization will be, but the more resources it will consume.
- **laser_max_range**: Max range of the laser beams.

- **odom_model_type**: This parameter defines the type of drive that we are using for the robot. In the case of the turtlebot we use differential drive (diff). If you do not set this parameter properly the odometry will not be properly calculated.

4.2 Path Planning with move_base

To perform path planning, you need to combine everything described above. You also need to start the **move_base** node from the move_base package. The file `start_navigation.launch` starts the **move_base** node by launching `move_base.launch`, located in the turtlebot3_navigation package:

```
<include file="$(find turtlebot3_navigation)/launch/move_base.launch">
  <arg name="model" value="$(arg model)" />
</include>
```

Since we are not going to edit the parameters in `move_base.launch` we are not going to copy it in our mapping package. Instead, we use the default one, located in the package turtlebot3_navigation.

4.3 Exercise

1. In `start_navigation.launch`:
   - Edit `<arg name="world_name" default="$(find turtlebot3_gazebo)/worlds/turtlebot3_world.world" />` to point to the world used for this assignment.
   - Edit `<arg name="map_file" default="$(find mapping)/maps/example_world.yaml" />` to point to the map file you generated earlier.
   - Edit the following to start the robot at position (3.831436, -1.234452) with a yaw of 0. Note that this information is used by **amcl** to localize your robot when you start the simulation.
     ```
     <arg name="x" default="-0.5" />
     <arg name="y" default="-0.5" />
     <arg name="yaw" default="-2.65" />
     ```
2. Start everything needed: `roslaunch mapping start_navigation.launch`
3. Use the 2D Nav Goal feature in RViz (as seen in class) to move your robot around in the house and ensure the robot does not collide with any object. If you see collisions then your map is probably not accurate enough.
4. Reuse the function `movebase_client()` (seen in class) and drive to different goals to make sure everything works fine.

5 Recording Locations

Once you have checked that the navigation system is working properly in the previous exercise, you will need to record 4 entries in a YAML file. Each entry corresponds to a room in the house. For each room, assign a tag (with the name of the room) alongside its coordinates in the map. You will need to teleoperate your robot into each room of the house (see Figure 2) and call a ROS service to save the location in a YAML file. You do not need to teleoperate your robot to a specific location in each room, just make sure the robot is in the room. The 4 tags you will use are: `bedroom`, `livingroom`, `recreationroom`, and `kitchen`.
You can access the coordinates of each location (which correspond to the pose of the robot) by looking into the topic the amcl node publishes on. This topic is **amcl_pose** (see Figure 3). The only data you really need to save are the position and the orientation. In class, we used the topic **odom** to get the pose of the robot. **amcl_pose** is more accurate than **odom** as it does not rely on the wheel encoders (like **odom** does).

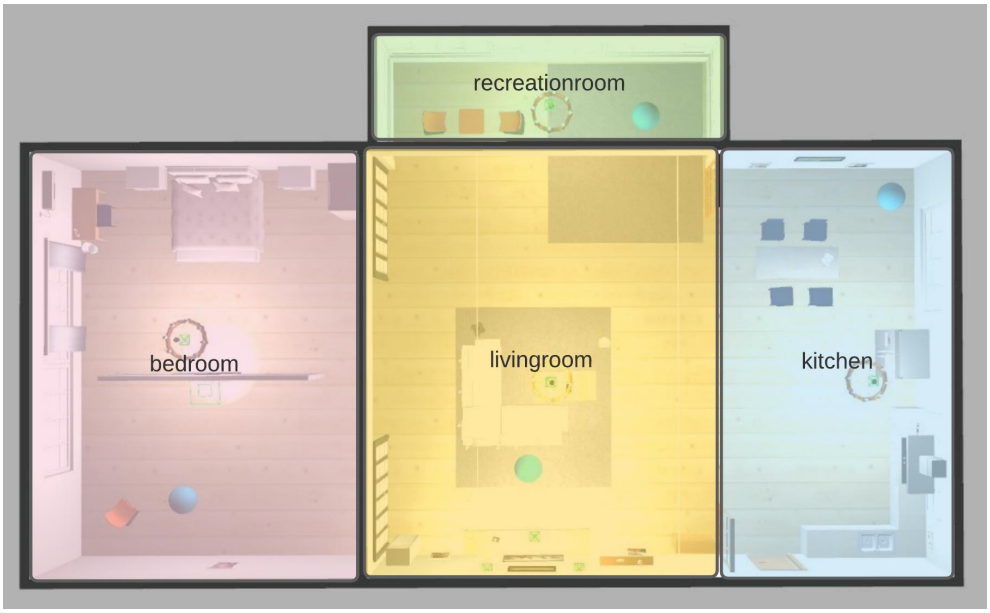

Figure 2: Rooms to visit with your robot.

```
zeid@zeid-MAX-15:~/catkin_ws$ rostopic echo amcl_pose
header:
  seq: 0
  stamp:
    secs: 0
    nsecs: 770000000
  frame_id: "map"
pose:
  pose:
    position:
      x: -0.491042562248
      y: -0.498742898346
      z: 0.0
    orientation:
      x: 0.0
      y: 0.0
      z: -0.971298100996
      w: 0.237865506118
  covariance: [0.17695810736975362, -0.006572467439192514, 0.0, 0.0, 0.0, 0.0, -0.00657246
.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0
0.06672956326227913]
---
```

Figure 3: Data published on the topic **amcl_pose**.

Before you start the exercise for this section, you need to create a new package (location_recorder) which depends on rospy and geometry_msgs. In this package create the following folders and files:

- *nodes* directory which contains the Python file `room_recorder`.
- *srv* directory which contains the service file `RoomPose.srv`.
- *records* directory.

5.1 ROS Service

Edit RoomPose.srv to create the request and the response. For the request we are going to pass the name of the room. The response from the server will be another string which is a custom message, such as "Entry saved in file". Below is the structure of your service RoomPose.srv:

```
# request
string room_label
---
#response
string confirmation
```

In room_recorder write the Python code for the service server and the service client. We had 2 separate files for the service and the client in class. For this assignment we will use only one Python file. In class, we created the service server as follows:

```
# Create the server that listens to the request
rospy.Service("add_two_numbers", AddTwoNumbers, callback)
```

For this assignment, it should be:

```
# Create the server that listens to the request
rospy.Service("room_service", RoomPose, callback)  # Haha room_service, get it?
```

- The service client will send a request to the server. The request is a string with the label of the room. This request should be one of the room labels ("bedroom", "livingroom", "recreationroom", or "kitchen").
- The service server will retrieve the pose of the robot (position and orientation) from the topic **amcl_pose** and save it to a file inside the *records* directory. You will need a subscriber to this topic. Also, note that messages published on **amcl_pose** are of type **geometry_msgs/PoseWithCovarianceStamped** (try `rostopic type amcl_pose`). So, make sure you import this module in your Python file.
- Once the entry is saved, the server sends a response to the client with the message "Entry saved in file".

5.2 Exercise

- Create your package and write your ROS service. Make sure to edit CMakeLists.txt and package.xml as seen in class. Remember that the service was not working in class and I sent you another email afterwards to resolve this issue.
  - In CMakeLists.txt, make sure you have the following:

```
find_package(catkin REQUIRED COMPONENTS
    rospy
    geometry_msgs
    std_msgs
    message_generation
)
```

```
add_service_files(
    FILES
    RoomPose.srv
)
```

```
generate_messages(
    DEPENDENCIES
    std_msgs
)
```

```
catkin_install_python(PROGRAMS
   nodes/room_recorder
   DESTINATION ${CATKIN_PACKAGE_BIN_DESTINATION}
 )
```

  - In `package.xml`, check that the following lines are included:

```
<build_depend>message_generation</build_depend>
<exec_depend>message_runtime</exec_depend>
```

- Build your package: `catkin build`
- Source your *devel* folder.
- Check your service file is listed with `rossrv list` (use with `grep RoomPose`)
- Start navigation: `roslaunch mapping start_navigation.launch`
- Your robot spawns in the living room: `rosservice call room_service "livingroom"`. This step should create a file `location_recorder.yaml` in the *records* directory. It should look like the snippet below. Of course, you will get real numbers from the topic **amcl_pose** instead of `<x>`, `<y>`, `<z>`, and `<w>`. Note: You are generating a YAML file and the indentation is crucial. Be sure that indentation is properly done when writing in the YAML file.

```
livingroom:
    position:
        x: <x>
        y: <y>
        z: <z>
    orientation:
        x: <x>
        y: <y>
        z: <z>
        w: <w>
```

- Move the robot to the recreation room, then `rosservice call room_service "recreationroom"`
- Move the robot to the kitchen, then `rosservice call room_service "kitchen"`
- Move the robot to the bed room, then `rosservice call room_service "bedroom"`
- You can move the robot using teleop or 2D Nav Goal. The last three steps above should open `location_recorder.yaml` and append each new entry to it. In the end, your file should have the entries below:

```
livingroom:
    position:
        x: <x>
        y: <y>
        z: <z>
    orientation:
        x: <x>
```

```
        y: <y>
        z: <z>
        w: <w>
recreationroom:
    position:
        x: <x>
        y: <y>
        z: <z>
    orientation:
        x: <x>
        y: <y>
        z: <z>
        w: <w>
kitchen:
    position:
        x: <x>
        y: <y>
        z: <z>
    orientation:
        x: <x>
        y: <y>
        z: <z>
        w: <w>
bedroom:
    position:
        x: <x>
        y: <y>
        z: <z>
    orientation:
        x: <x>
        y: <y>
        z: <z>
        w: <w>
```

6 Submission

- You should submit only 1 package, the one you created. Even though you edited `start_mapping.launch` and `start_navigation.launch` through the **Exercise** Sections, submit only one package. Name it **group#_rwa3** if you are in a group. If you are working by yourself, then have your package named **firstname_lastname_rwa3**.
- Make sure your code is documented or you will lose points. This is even more important now because multiple people work on the same code.
- Penalty for late submission will be 4 pts if you submit it within 2 days after the due date. The penalty for submissions beyond 2 days after the due date will be 6 pts. No need to request for an extension. Submit your package whenever you want but make sure you understand that a penalty will be applied.
- Do not send us your code to debug it for you. If you need help, set up a zoom meeting and show us where the problem is. We will give you some pointers.
- Please, no last-minute (day of submission) help request.

7 Grading Rubric

- To grade this assignment we will drive the robot to each room and we will call the ROS service in each room. We will then look at the yaml file to make sure everything was properly recorded. This file will be reused in the final project.

+25 pts – **YAML File**: The YAML file is generated and all the entries are correct after calling the ROS service for each room.

+5 pts – **Documentation**: Code properly documented and PEP8 guideline followed. There is no need to generate any documentation.