[CS-475] Assignment 4: Mapping with Known Poses

Michael Maravgakis
maravgakis@csd.uoc.gr

Release date: 05/04/2023
Deadline: 26/04/2023

1 Overview

In this assignment, you will design an occupancy grid mapping algorithm for the turtlebot in gazebo. In the previous assignments you performed state estimation and localization by either using a GPS module or utilizing the given map. Now, we are assuming that the poses $[x, y, \theta]^T$ of the robot are known (with high accuracy) and the goal is to use those poses and the noisy scan measurements to create the 2D map of the finite space that the robot moves. The turtlebot is equipped with a lidar that can provide range measurements z_t, where z_t is the scan at time t and it is pass-through/hit information (see section 3). In order to create the occupancy grid, you can use the algorithms described in the *Probabilistic Robotics* book, Tables 9.1 and 9.2.

2 Installation

In your zip file, you are provided with a ROS package named assign4/. As always copy paste it to your ros workspace and compile. In case you need more information, check out the previous assignments. This package contains a launch file (*"burger.launch"*), that opens the gazebo simulation with the turtlebot and a custom world. This "room" is your workspace that you will need to map. Also, when you run the launch file it will open an rviz window that will help you visualize your results and also your measurements. In order to save time at your first tries you can avoid opening the gazebo simulation GUI by setting the `"gui"` parameter in the launch file to `false` and when you want to navigate the robot with the teleop package to check if your algorithm works you can switch it back on. As a reminder you can use:

1. `$ roslaunch assign4 burger.launch`
2. `$ roslaunch turtlebot3_teleop turtlebot3_teleop_key.launch`

If everything is set up, when you run your launch file, the windows from the image below should pop up.

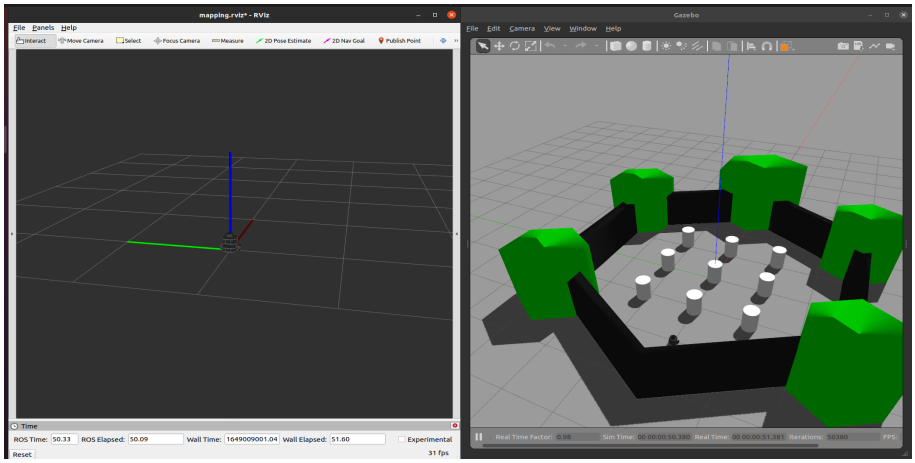

Figure 1: Gazebo + Rviz

3 Implementation

This implementation is fairly easy when you understand what you need to do. Imagine you have a laser that shoots a beam and you know the distance that the beam traveled when it hit an object. You can use this information to extract the following conclusions:

1. Between your laser and the distance that you measured there is empty space
2. At the direction you are shooting and at range equal to the distance you know there is an obstacle
3. Behind this obstacle you don't have a clue what is happening

After you save this information you turn your laser a bit and you measure again. This is exactly how a lidar sensor works. Now, I'll walk you through on how to implement this algorithm, but most of the coding is left up to you. In case, you get stuck somewhere feel free to ask anytime.

First of all the data structures that you are going to use are the following:

1. `sensor_msgs/LaserScan`, contains the distance measurements from the lidar

2. `nav_msgs/OccupancyGrid`, this will be the output message that will be visualized from the rviz

3. `nav_msgs/Odometry`, will be used to extract the pose of the robot

The topics that will be subscribed/published are the /scan, /map, /odom accordingly. You are provided with some parameters at the beginning of your node `mapping.py`, read them carefully. The size of the room is `6x6(m)` and the resolution `0.01 (m)`.

First, you will define the message and the publisher that will be responsible for publishing your occupancy grid map. (use `frame_id ="odom"`). If you type `$ rosmsg show nav_msgs/OccupancyGrid`, the following output will appear:

```
std_msgs/Header header
  uint32 seq
  time stamp
  string frame_id
nav_msgs/MapMetaData info
  time map_load_time
  float32 resolution
  uint32 width
  uint32 height
  geometry_msgs/Pose origin
    geometry_msgs/Point position
      float64 x
      float64 y
      float64 z
    geometry_msgs/Quaternion orientation
      float64 x
      float64 y
      float64 z
      float64 w
int8[] data
```

You will need to setup the following:

1. `frame_id`

2. `resolution`

3. `width and height`

4. `origin position and orientation (0,0,0,1)`

In my case, I set the origin position at (-3,-3,0), although this is not mandatory, I suggest to do the same. The data list contains the information on whether or not the cells are occupied. For each cell, it is 0 (white) if it's free and 100 (black) if it's occupied. Anything in between is how sure we are that this cell is

occupied or not, the more gray the more sure we are it is indeed occupied. For example when you don't have any information regarding the occupancy state of a cell, the data part for that cell will be 50 (% probability)

After you define the occupancy grid move on to the initialization part, where you set every cell as unknown (50). Try publishing your map and you should see the following:

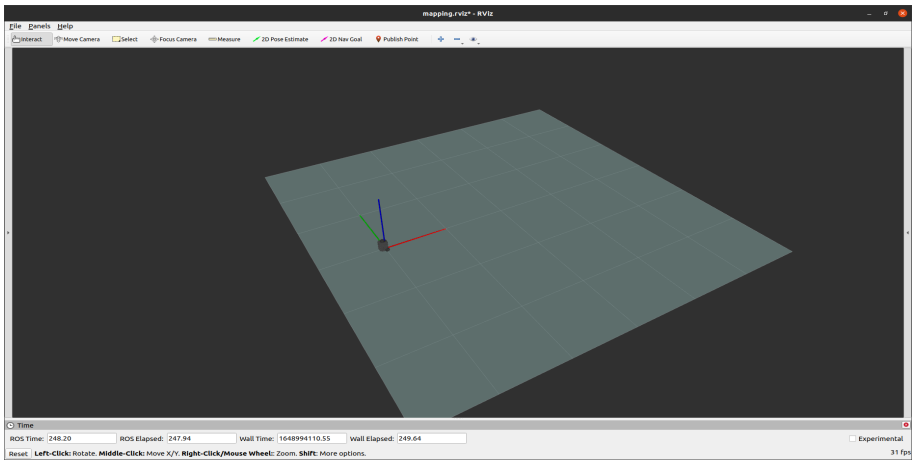

Figure 2: Initial grid (maybe different color)

Now, every time you get the position from the /odom topic, it is with respect to the world frame, so the first thing that you'll do is to translate each time the robot's position to the origin of your map. The robot is located at (-2,0) with respect to world frame. After the translation, I suggest to navigate the robot and check if the measurements are correct visually.

Finally, you will update the map. The update process is fairly simple but it can get a bit confusing with the indices of the occupancy grid so I suggest to draw an example where the robot has a certain orientation and trace a single beam to check if your math makes sense. The only math prerequisite is simple trigonometry. Also to avoid confusion convert the problem to binary at first, meaning update the cells with either 0 or 100 (free or occupied). You will need to use additional information from the /scan topic, not only the ranges, but also the `angle_min, angle_max` etc. If the update part is done correctly, when you publish your map you should see the following:

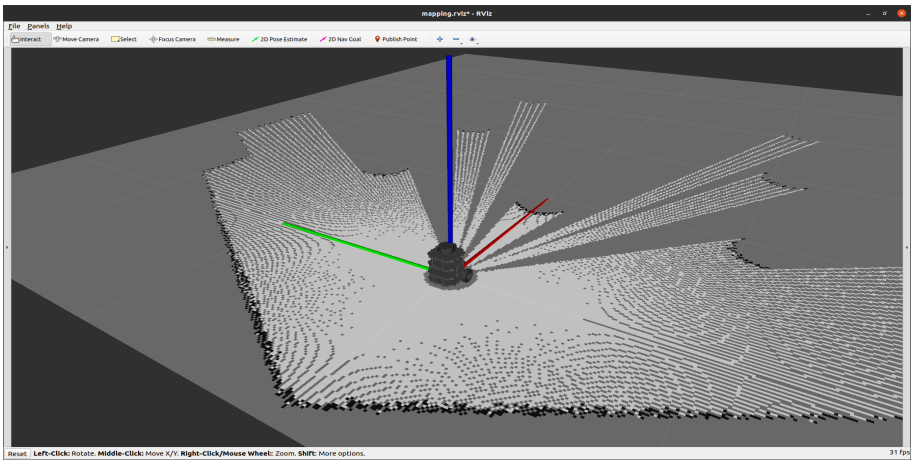

Figure 3: Initial results

After you move the robot around the map should get bigger and better. Something like this:

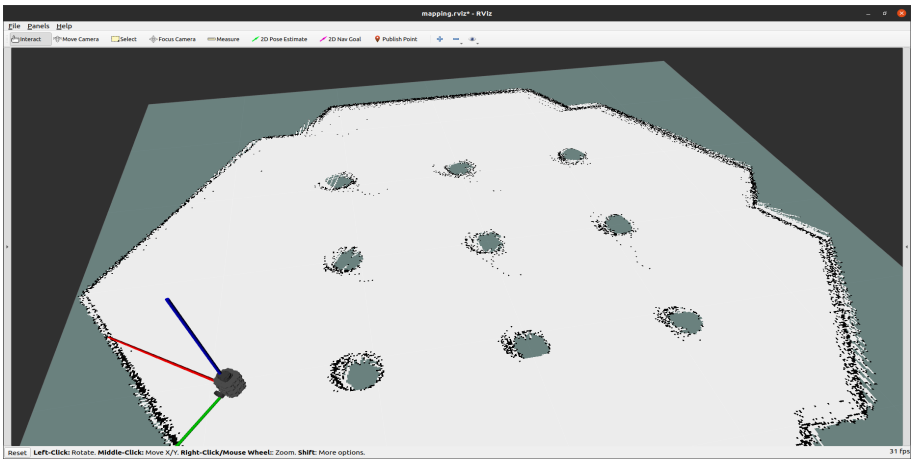

Figure 4: Mapping with binary (ignore the greenish color template)

Although the map looks good as is, you can easily notice that it is far from perfect and the noise only propagates over time and will be even worse later on. The above steps to convert the problem to binary are not mandatory, but implementing the algorithm this way at first makes it a bit more easier. At this point you are almost done, the final step is to use the **log-odds** to extract a more accurate probabilistic occupancy grid. Instead of 0 and 100, now the data

values of your grid can be anything in between. If everything is implemented correctly the final result should look like this:

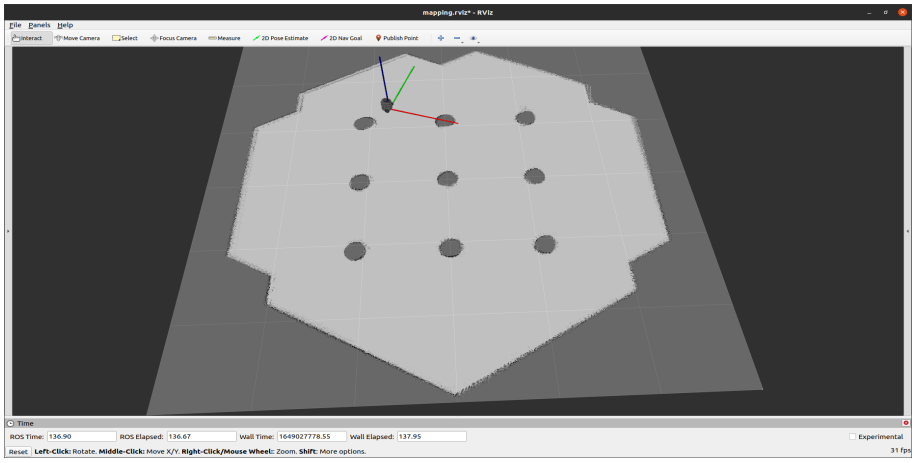

Figure 5: Final Result with log-odds

4 Submission

Sent your node (`mapping.py`) attached via email at: **maravgakis@csd.uoc.gr** with subject `"[CS-475] Assignment 4 submission"` Don't forget to mention your name and registration number. The deadline is at **26/04/2023 23:59**