Autonomous Robots
Lab 3 - Localization and mapping with Turtlebot

SAVININEN BONHEUR DARJA STOEVA
MENG DI ALBERT CLÉRIGUES
Universitat de Girona
May 7, 2017

1 Building a Map

1.1 Octomap

The map is built using Octomap library in the simulator. An octree is a hierarchical data structure for spatial subdivision in 3D and is used for building and representing the map. Each node in the octree represents the space contained in a cubic volume which is always called a voxel. This volume can be recursively subdivided into eight subvloumes until the minimum voxel size is reached. And the size of the voxel is the resolution of the map which is set by users passing by parameters in the launch file.

The turtlebot has Kinect sensor which can sense the point clouds of the configuration in the 3D space. But how does the Octomap convert the point clouds into occupied voxels? The Octomap is actually providing a probabilistic representation of the 3D world in form of octree. The voxels can shown occupied if the points sensed landing inside the cubic volume reach a certain amount. That means this voxel has a high probability to be occupied. And the voxels are shown free when there's no points landing inside.

As we can see in Figure 1, the objects in the left map are kind of broken and not fully present. Since the turtlebot didn't captured enough point clouds to fill up the voxels, the objects are shown partially. The objects shown in the right figure have more details since the sensed point clouds are as much as the voxels can be considered occupied.

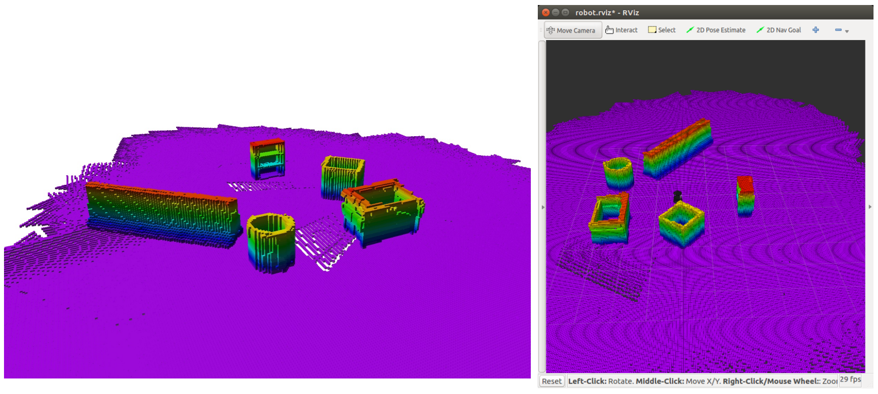
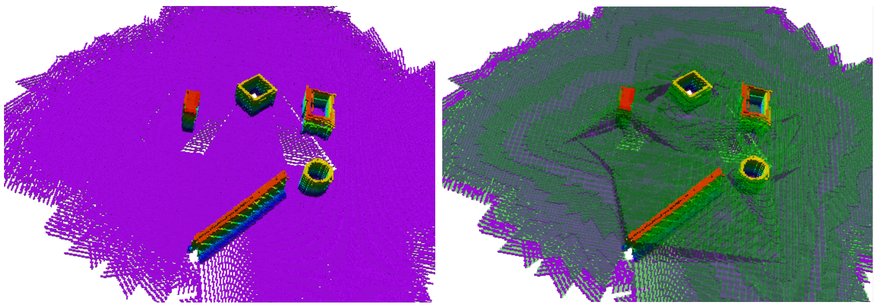


Figure 1: The map built in simulator

1.2 Map visualization

For the built map in Rviz, we can save it in the form of .bt and visualize the map by Octovis which is a visualization tool for the OctoMap library.

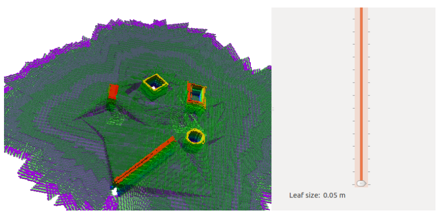
We can visualize the map in different representations. When we press 1 the map can be shown as colorful voxels which each color represents the height 3D information Figure 2 (a). And it can also shown as gradient voxesl with probability of occupancy when we press f Figure 2 (b). The voxels with light color are free space obtained by clearing the space on a ray from the sensor origin to each end point. We can clearly see the occupancy information of each voxel form this representation.



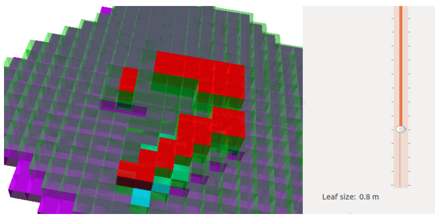
(a) (b)

Figure 2: Visualizing the map in octovis viewer

The map can also be shown in different resolutions. As we explained about how the Octomap works, the resolution depends on the division times of the octree which is the size of the minimum volume. The larger size of voxels we set, the lower resolution the map.



(a) Size of voxel: 0.05m



(b) Size of voxel: 0.8m

Figure 3: Visualizing the map in different resolutions

1.3 Mapping with the real Turtlebot

In order to build a map with a turtlebot in a real time, firstly we need to set up a connection between the turtlebot and our computer. After the connection is set, we can move the turtlebot using the `control_turtlebot` package, which allows us to control and teleoperate the turtlebot using the keyboard. Since we are building a map in a real time we want to receive registered depth points from the kinect sensor, the launch file `start_turtlebot_mapping_control` is edited accordingly. To build and save the map of the environment we need to launch Octomap before we start to move the turtlebot around. Once the Octomap is launched, the next step is controlling the turtlebot to move around the environment, while the map is build in octomap and later saved. The saved map is visualized with octovis and it is shown in Figure 4.

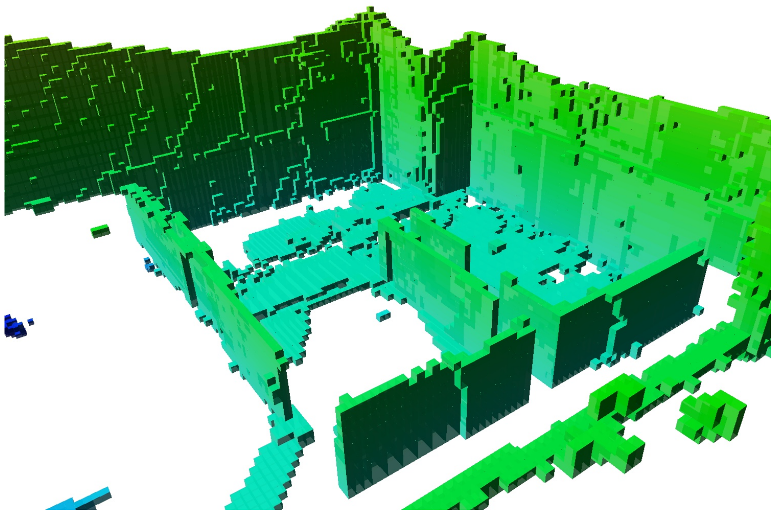


Figure 4: Octomap build with a real turtlebot

From the figure it can be seen that although there is a presence of noise and several insignificant artifacts, the map was build accurately. The white space that occurs in the map, is the area that the turtlebot has not mapped. The reason for this is that we did not move the turtlebot back to the starting point. One way of improving the mapping would be controlling the turtlebot to go back to the initial position. That will allow the robot to revisit the path he already mapped, which will give him more information to build a more accurate map with less noise and artifacts.

2 Planning with OMPL

2.1 Laserscan sensor and Octomap

We will use an OMPL based planner in 2D, to which we will provide a 2D slice of the real 3D map at a fixed height. Since Octomap is expecting a PCL PointCloud object, the node `laserscan2pointcloud.cpp` takes care of converting it from the ROS message object `sensor_msgs/LaserScan`. The node publishes the resulting pointcloud through the topic `pc_from_scan` which is redirected, as specified in the launch file, to the `cloud_in` topic of the octomap node.

2.2 Planning with Simple Setup

When the node `offline_planner_with_services_R2` starts, an instance of the class `OfflinePlannerR2` is created that will read the configuration file `planner_parameter.yaml` where the algorithm, space bounds and maximum time are specified.

When calling the planning service `findPathToGoal`, an instance of the class `SimpleSetup` is created and configurated according to the specified parameters in the configuration file. Finally, the resulting path is returned as an array of Pose2D and a message is sent to Rviz to plot the result as well as the intermediate RRT nodes. Once the plan is made we command the turtlebot to follow it by calling the `goto` service.

The map is still being updated while following the RRT plan, although it cannot be seen in the images due to the shape of the path, which doesn't make the robot face any new obstacles. However, even when the map is updated with new obstacles the path remains static. When a new plan is requested an instance of the object `OmFclStateValidityCheckerR2` is made that requests the last version of the map, with which the plan is made. Once the solution path is returned, there is no code checking the validity of the path in the updated map and replanning if necessary, which is why a plan from an incomplete map can become obsolete as we are following it.

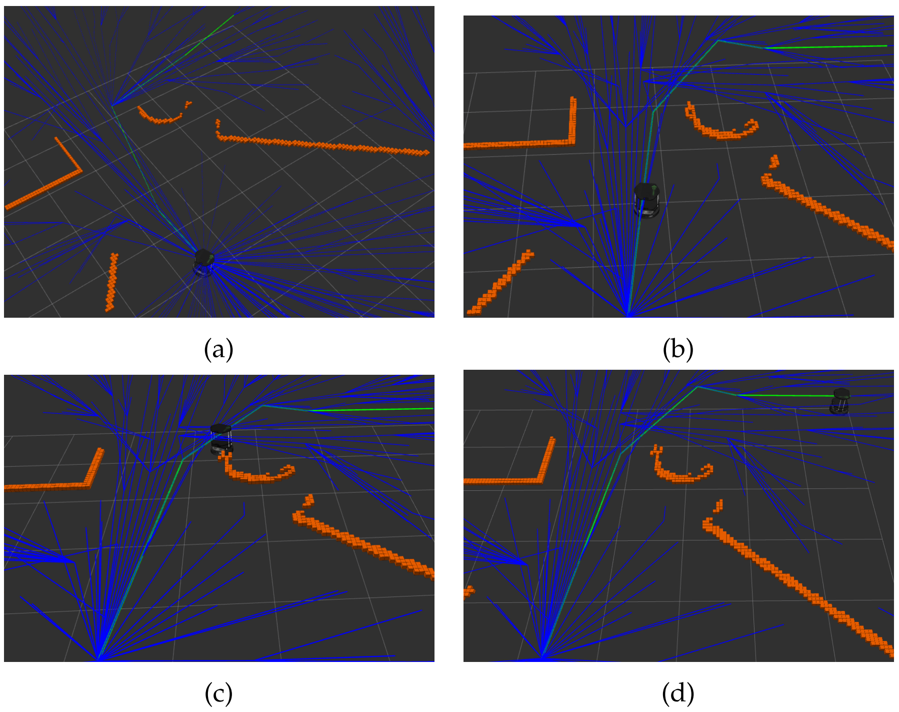


Figure 5: Simulation Turtlebot following the RRT computed path.

There exist no problems when planning more than one path, since the planner service will receive an updated version of the map at the time of request. Hence it would be a good idea to add an obsolete path checker routine that would replan if it detected a new obstacle in the current plan.

2.3 Planning with the real Turtlebot

The first steps of planning with the turtlebot is to build a map of its environment. It works the exact same way as what we have done so far but , instead of simulating the turtlebot and what does it see, we are connected to the turtlebot via ssh.

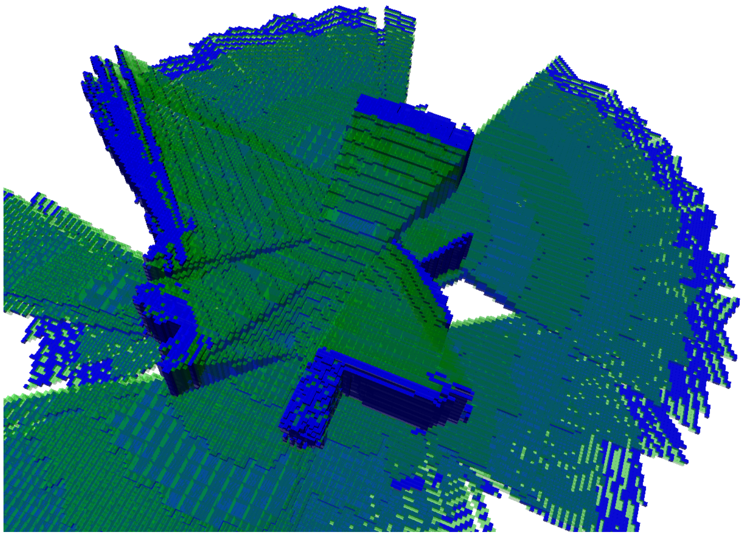


Figure 6

Once the map is gathered we can launch OMPL in the exact same way as before, only changing the goal parameters (we find them by moving the robot manually) :

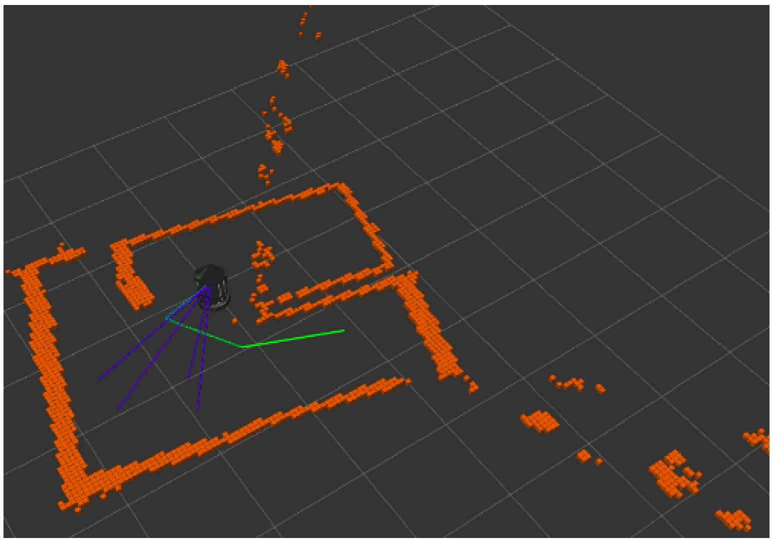


Figure 7

As before we aren't updating the map as we follow the path and, as before, we can apply several path finded motivated movement in a row.
It is interesting to note the artefacts appearing on the map. In fact no objects

were in those positions while we were building the map. Those appeared due to false measurements from the Kinect. However, through the way Octomap is updating the map, it's really unlikely to declare as empty an occupied voxel. This implies that artefacts, although only appearing once, will tend to stay in the map. In our case, those artefact doesn't allow the robot to find a path to, or from, the top right corner of the map. It is then really important to be careful with the acquired map and it's representation in Octomap while using the OMPL library.