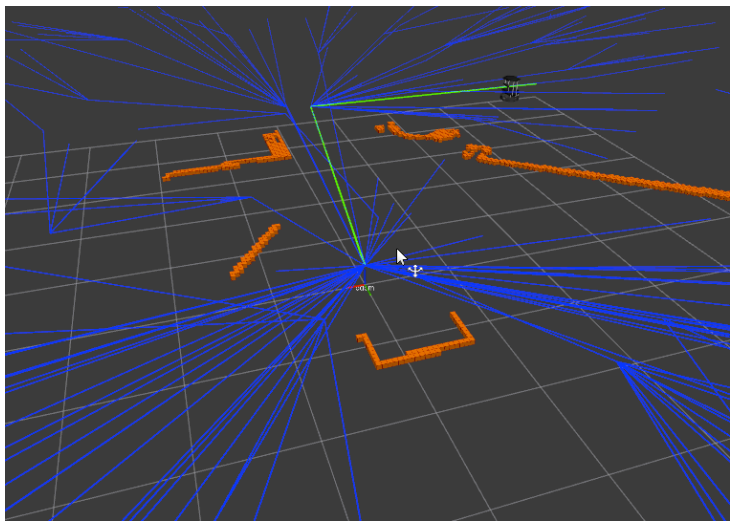
Autonomous Robots
Mapping, planning and controlling with the Turtlebot

This document will guide you through the practical work related to preparing the Turtlebot in a simulated environment to map the environment and plan collision-free paths. It will also propose you some exercises for checking the performance of these systems.



1. ROS dependencies and packages for simulation

Install dependencies (replace kinetic for the ROS distribution that you are using). This assumes that you have installed the different dependencies used previously (Octomaps, octovis, etc.):

```
sudo apt-get install ros-kinetic-ompl
```

Clone the repository (if you have not downloaded it before):

```
cd ~/catkin_ws/src
git clone https://bitbucket.org/udg_cirs/mapping_planning_tutorial.git
```

or pull in case you already have it:

```
roscd mapping_planning_tutorial
git pull
```

Compile:

```
cd ~/catkin_ws/
catkin_make
```

2. Checking vehicle's capability for building a simple map.

For these exercises we will use a simplified version of the map. This means that instead of using a full 3-dimensional (3D) map, we will create a 2D map. For doing so, Octomap-server will use the laser_scan of the Kinect camera, rather than the complete point cloud.

Before executing the turtlebot simulator, verify that the following file:

```
~/catkin_ws/src/mapping_planning_tutorial/control_turtlebot/launch/start_turtlebot_modules.launch
```

is correct according to your ROS distribution (open the file and check).

In one terminal execute (turtlebot simulator and teleoperation):

```
roslaunch control_turtlebot start_turtlebot_modules.launch
```

In a second terminal execute (mapping, planning and control system):

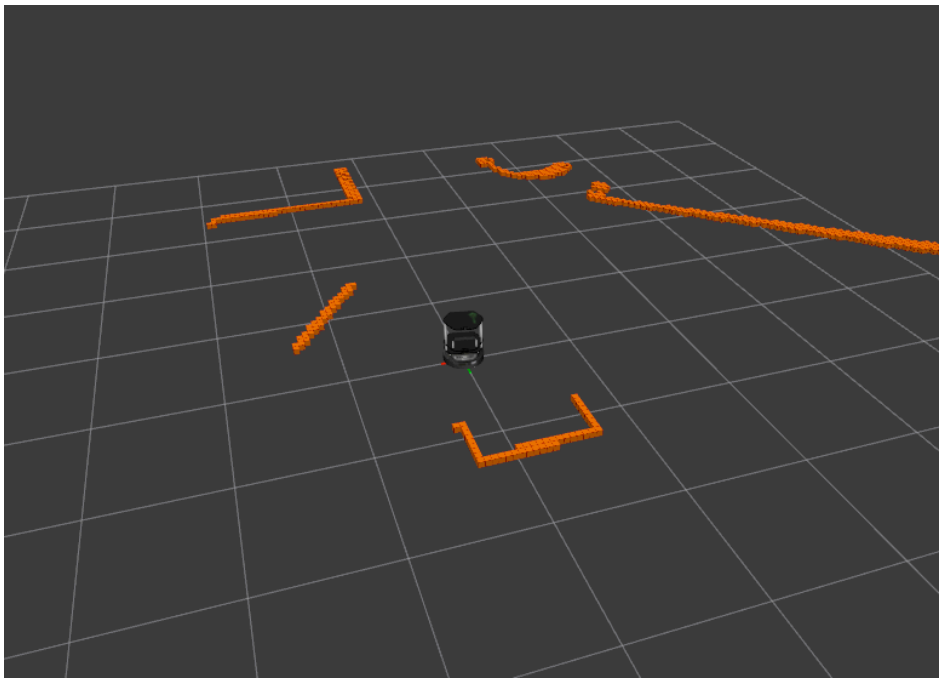
```
roslaunch control_turtlebot start_turtlebot_mapping_planning_control.launch
```

Turn the turtlebot around to create a simple map that includes the nearby obstacles:

```
Over the first terminal, and using the keyboard, control the turtlebot to turn around:
j      l
```

Verify that the turtlebot's map is coherent according to Gazebo simulated environment:

```
Use octomap_server to save the map and octovis to visualize and compare these 2D maps with 3D maps done in previous exercises
```



3. Planning a collision-free path using Octomap

For planning collision-free paths we are going to use the Open Motion Planning Library (OMPL). This library contains several sampling-based algorithms, such RRT, PRM, RRT*, EST, among others. There are different parameters that can be adjusted, however the basic ones include the bounds of the configuration space, the start configuration and the available time for the planner to find a path. Such parameters can be setup in the following file:

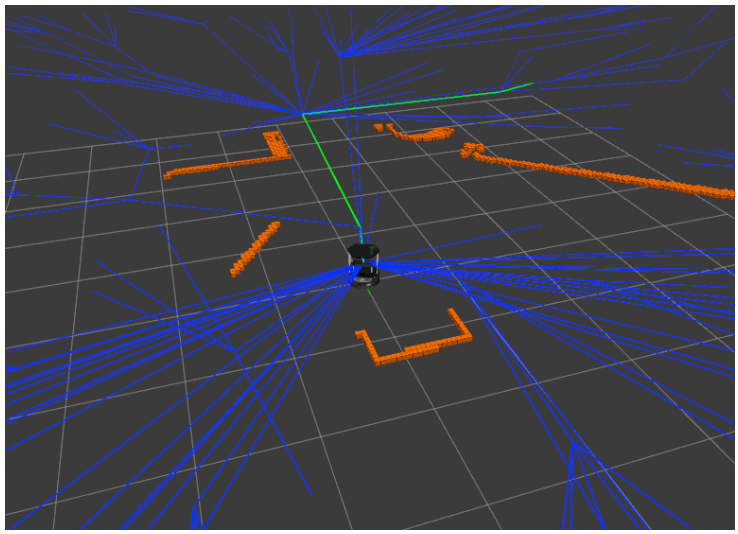
```
~/catkin_ws/src/mapping_planning_tutorial/control_turtlebot/launch/planner_parameters.yaml
```

The path planning node defines one service that attempts to find a collision-free path from the current turtlebot position to a desired goal. The goal must be within the bound established in the previous files. This node is implemented in the following file:

```
~/catkin_ws/src/mapping_planning_tutorial/control_turtlebot/src/offline_planner_R2.cpp
```

In a third terminal use the service to plan a collision-free path:

```
rosservice call /controller_turtlebot/find_path_to_goal #then press tab twice to complete
rosservice call /controller_turtlebot/find_path_to_goal "goal_state_x: -5.0 goal_state_y: -5.0"
```



4. Mapping the environment, planning a path, and following the path:

Once the path has been calculated, we can attempt to follow the path. This, however, assumes that the environment is fully mapped (which is probably not true), and that the environment is static, meaning that obstacles perceived remain in its initial position. We can verify if the turtlebot if capable of following the path by using a controller that is included in the following file:

```
~/catkin_ws/src/mapping_planning_tutorial/control_turtlebot/src/controller_turtlebot.py
```

In the third terminal use the service to plan a collision-free path and follow it:

```
rosservice call /controller_turtlebot/goto #then press tab twice to complete
rosservice call /controller_turtlebot/goto "goal_state_x: -5.0 goal_state_y: -5.0"
```

5. Submission.

WORK TO DO: After following the previous sections, perform some movements with the Turtlebot for checking the following systems:

a) Map building. Move the turtlebot through the simulated environment and verify if the planner can find paths from the current position of the vehicle to a give goal. Do not forget that such a goal position must be included in the planner area specified in the planning bounds (planner_parameters.yaml).

b) Planner and map consistency. What happen to the map when the turtlebot follows a path? does the map change? If it does, why? If it does not, why?

c) Repeatability. Can the map be used for a second start-to-goal query (try to send the robot to a second position, after reaching the first one, with the existing map)?

d) Repeat the same exercise using the real robot (optional).

e) Report. Write a report explaining all the work done on this exercise. Submit the report in pdf and also the files corresponding to the maps (.bt).