EPD 5: Path Planning

Objectives

- implement a graph-search based algorithm to solve the robot path planning problem.
- Use of the ROS costmap_2d as a grid for planning.

Assignments

1 Introduction

The goal of this EPD is to implement a path planning algorithm that obtains a collision-free path from the initial localization to the goal by considering an environment model (map) to avoid the static obstacles (see Figure 1). The path is defined by a list of waypoints.

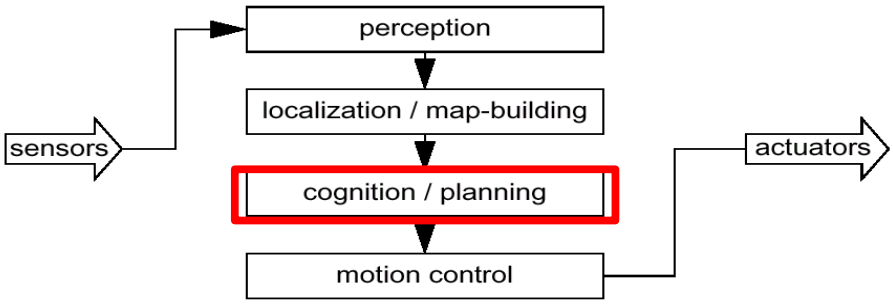

Figure 1. Deliberative architecture for robot navigation

In the EB classes we have seen different graph-search methods for computing a feasible path from an initial node to a final node (ordered according to their implementation complexity):

- Breadth-first search
- Depth-first search
- Dijkstra
- A*
- Theta*

We encourage you to try to implement one of these methods, preferably Dijkstra or A*.

2 Generating navigation costmaps using maps and robot features: costmap_2d package

In this EPD we will use the ROS package *costmap_2d* which builds on the map information (published by the map_server package as we presented in the EPD4). This package provides a 2D or 3D occupancy grid with costs based on the map of the environment and other features. The costmap uses the concept of overlapped layers. For instance, we could define and configure different layers as:

- static layer, which uses the data of the static obstacles included in the map.
- inflation layer, which adds an extra inflation cost around the obstacles.
- obstacle layer, which is feed from the range sensors onboard the robot, so it is capable of adding/removing non-mapped obstacles to/from the costmap dinamically.

We will use the static layer and the inflation layer. The costmap and its layers have different configuration parameters that we can assign through a yaml file. These parameters will define, among other things, how the costmap will be inflated. The inflating procedure is shown in the Figure 2. The configuration file of the costmap is available in our path_planning package at *config/global_costmap_params.yaml*.
You can find a description of all the available parameters at http://wiki.ros.org/costmap_2d

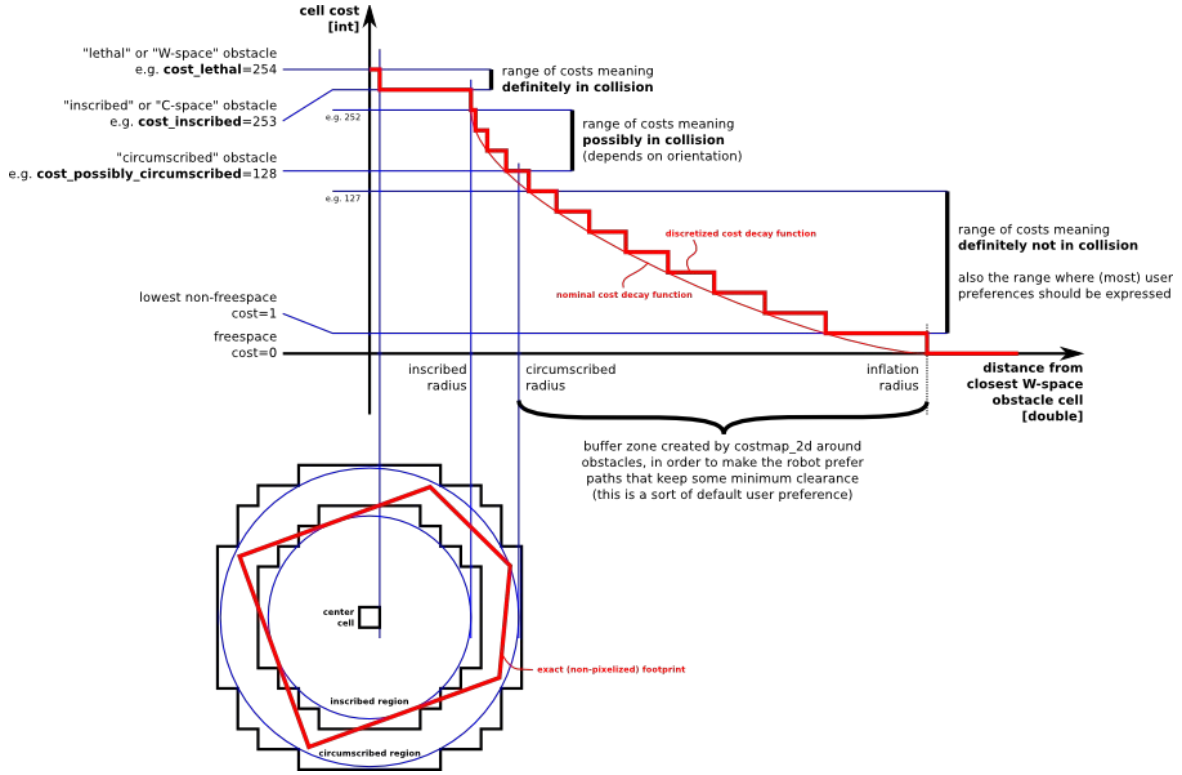

Figure 2. Inflation procedure employed by the costmap_2d node.

The simulation that we will use in this EPD can be seen in the left image of Figure 3. In the right image, we can visualize the Costmap in Rviz. We will use this costmap grid to plan feasible paths from the robot position to a given goal.

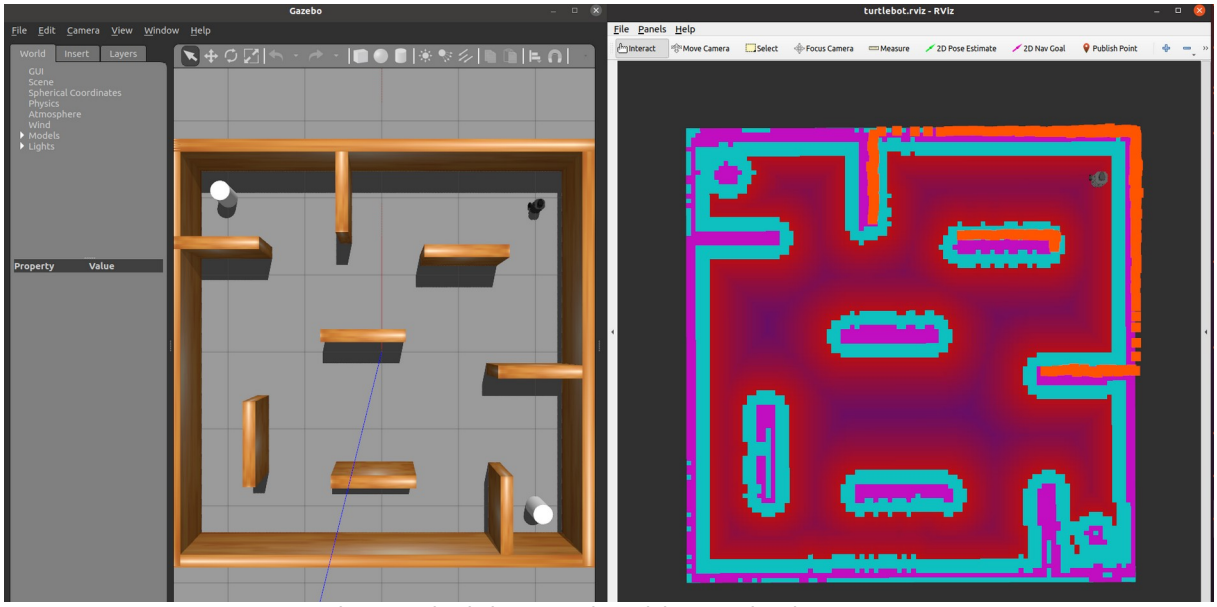

Figure 3. Simulation scenario and the associated costmap.

3 Planning feasible paths to a given goal

Given a geometric map with the static obstacles of the environment, we must implement a path planning algorithm able of computing a feasible path from the current robot position (we need the localization in the map, as we showed in EPD4), to given goal in the map frame (see Figure 4). The output must be a path represented as a list of waypoints (see the ROS message nav_msgs/Path: http://docs.ros.org/en/noetic/api/nav_msgs/html/msg/Path.html).

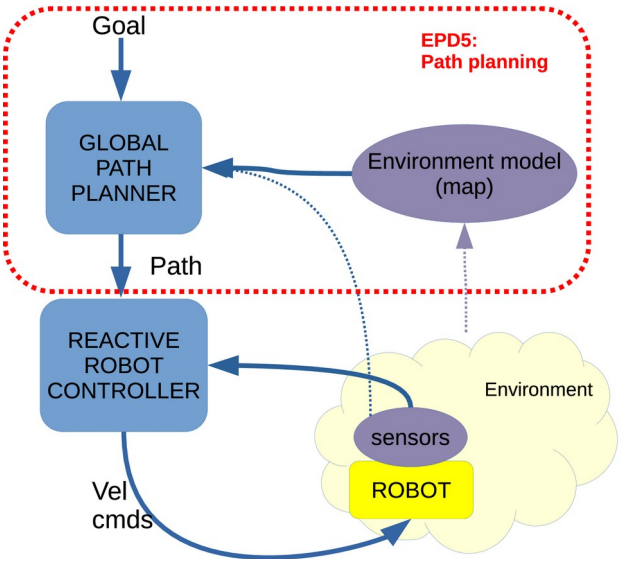

Figure 4. Navigation diagram

To develop out path planning algorithm, we will use a new ROS package called “path_planning”. You can download it from the folder EPD5 in the Aula Virtual, and place it in your ROS workspace. This package contains:

- config:
  - global_costmap_params.yaml → configuration file of the costmap that we want to use for planning.
  - planning.rviz → a Rviz configuration file to visualize all that we need.
- launch:
  - turtlebot3_sim.launch → launch file to run the simulation of the Turtlebot in the objective environment.
  - localization.launch → launch file to run the localization: nodes map_server y amcl.
  - path_planner.launch → main launch file that includes the other two launch files besides running the costmap_2d and your path planner node (path_planner.py).
- scripts:
  - path_planner.py → main ros node. Here, the costmap for planning as well as the navigation goal are received through ros topics. Once a goal is received, a call to the planning algorithm is performed. After computing the path, it is published in the topic “/path” (it will be visualized in Rviz).
  - myplanner.py → scheme of the path planning algorithm. It contains the methods that we must implement as well as some auxiliary functions that can be useful.

****IMPORTANT****: the script **path_planner.py can not be modified**. You must follow the planning estructure defined in this script. You must just implement you path planning method in the script myplanner.py which is incomplete.

To run the whole system you just need to launch the file path_planner.launch:

```
$ roslaunch path_planner path_planner.launch
```

Once the system is up, you can send navigation goals through Rviz by using the “2D Nav Goal” button. Then, the plan computed by your algorithm should be visualized.

Figure 5 shows an example capture of a path computed by using the Dijkstra algorithm.

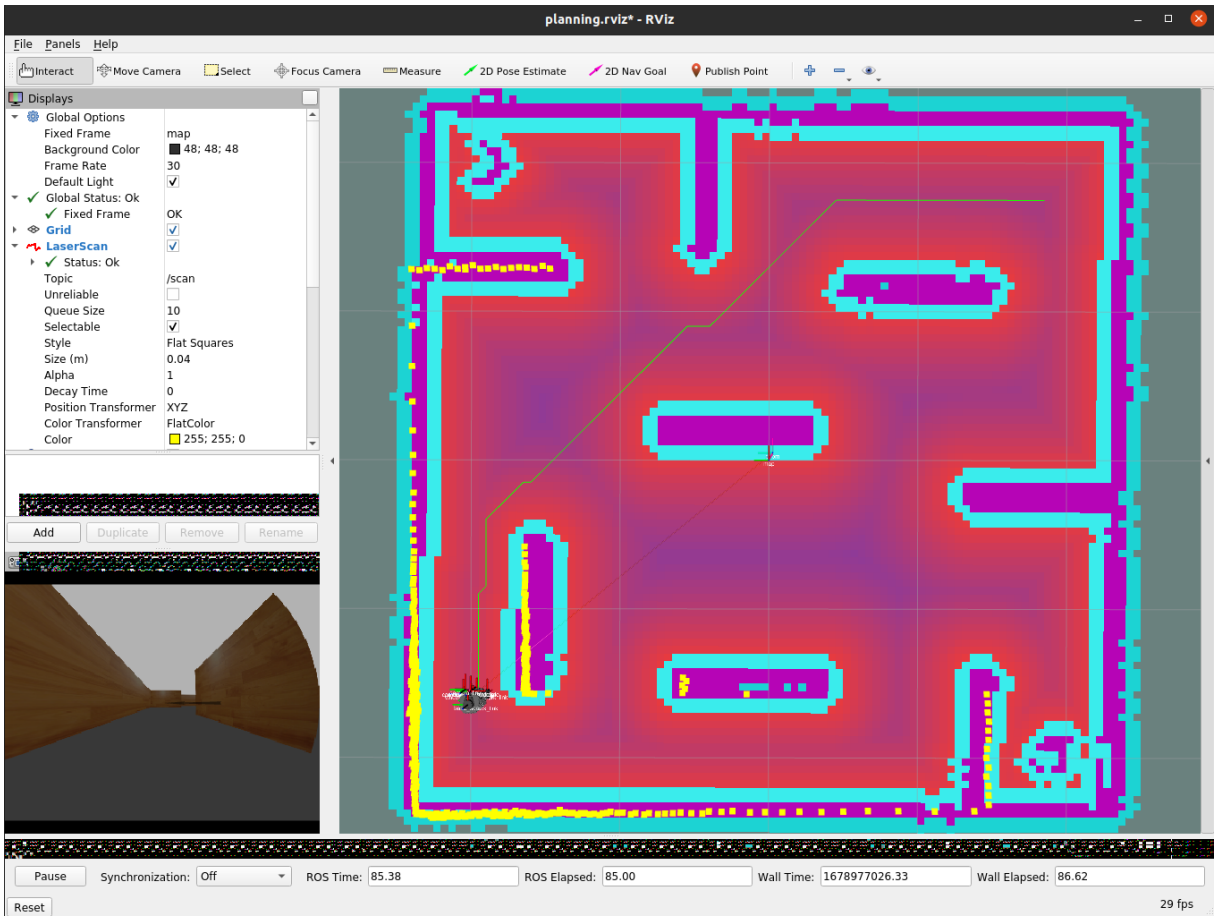

Figure 5. Example of a path computed by the Dijkstra algorithm.

Optional exercise: replanning

We can add an additional layer to the costmap package that takes into account information from the range sensors. This way, the dynamic obstacles detected by the 2D laser sensor will be added to the costmap, and therefore, the plans can be computed to avoid the obstacles non-mapped previously. To know how to add this layer, we can consult the information given here: http://wiki.ros.org/costmap_2d

Moreover, we need to compute a new path everytime we receive an updated costmap. To do that, you can take a look to the "update_map" parameter in the *path_planner.py* node. You will also need to add a new call to the *computePath* method in order to plan a path not only when a new goal is received. Only in this case, it is allowed to slightly modify the script path_planner.py.