AIMS Mobile Robotics 2020: Path Planning and Control Task

March 2020

1 Introduction

The task for today is to implement a ROS package which plans a path in continuous space from the current location of the robot to a goal location, and publishes velocity commands to execute the path and navigate to the desired pose.

Navigation will be implemented as a ROS action server. Some of the code for this task is provided in the aims_student_repo in the aims_navigation package.

1.1 Understanding ROS Actions

We will be implementing navigation using a ROS action. In the ROS tutorial on Monday, we saw that there are cases when one ROS node needs to request another node to perform some task, and receive a reply to the request. This can be achieved with ROS services.

However, if the service takes a long time to execute, the user might want the ability to cancel the request or get feedback about the progress of the request. The achieve this, instead of implementing a ROS service we can instead implement a ROS action using the actionlib package. This allows us to create action servers that execute long-running goals or "actions" that can be preempted and provide feedback. Additionally, calls to ROS action servers are non-blocking, allowing other code to be executed while the action is carried out.

The file aims_navigation.py initialises the action server. You should not have to edit this file, but you should read through it to understand what is happening. The action callback first calls a planner to find a valid path to the goal. This path is subsequently passed to a controller to execute the path. You will need implement functionality for the planner and controller in planner.py and controller.py respectively.

2 Task

The task for today will be described here. However, **please read this entire document before you start writing code as it contains more information which may be useful.**

2.1 Path planner

You should implement a path planner in planner.py in which skeleton code is provided, by modifying the plan() function. Your planner should be able to plan a path in continuous space from any starting position to any valid goal position in the map. You may choose any suitable planning algorithm to implement.

planner.py includes functions to:

- Check if a location is in free space.
- Check if the line adjoining two points is always in free space.
- A function to publish a path to visualise in rviz

You may find it useful to use scipy.spatial.KDTree to quickly find nearest neighbours.

To test your planner you can send a goal to the action server through rviz as explained below. The green line displayed in rviz visualises the path produced by your planner.

2.2 Controller

Once you have a working path planner, you will need to also create a controller in controller.py to follow that path by modifying the control() function and publishing velocity commands to the cmd_vel topic. You may start by writing a simple controller which does not consider obstacles detected by the laser, and follows the path "blindly". As an extension, you should modify your controller so that the robot avoids obstacles detected by the laser.

controller.py includes functions to

- Compute the yaw (rotation about the vertical axis) from a ROS pose message.
- Compute the yaw error in the range $[-\pi, \pi]$ given two yaw angles.
- Find the point furthest along the path within a certain distance of the robot.

Note that your controller will also need to make the robot face in the correct direction once reaching the goal. To test your controller send navigation goals using rviz.

3 Initialisation

We will start by developing the planner and controller in simulation on the Ubuntu laptops. When you are ready to start testing, there are a few commands that you will need to run on your laptop before you are able to process navigation goals. You can use the start_jackal_sim.tmux script to avoid typing these repeatedly.

Start by initialising the simulator:

```
roslaunch aims_jackal_sim jackal_sim.launch joystick:=true
```

Start the map server to publish the map:

```
roslaunch aims_navigation map.launch
```

We will require the estimated position of the robot to know the starting location for the planner, and to control the robot to follow the path. As such, you should start your particle filter and localise the robot

```
roslaunch pf_localisation particle_filter.launch
```

Launch navigation. This starts the action server defined in navigate.py. This will also start a node which listens to navigation goals sent via the rviz interface and sends them to the action server.

```
roslaunch aims_navigation navigation.launch
```

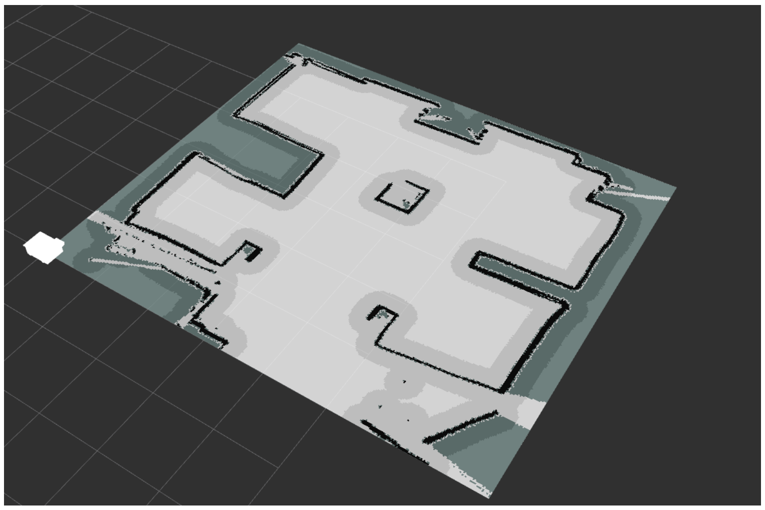
Start rviz using the configuration file provided. If run in the aims_student_repo folder this is

```
rviz -d jackal_viz.viz
```

Once you have implemented your planner and controller, you can send a navigation goal with RViz by clicking on "2D Nav Goal" and then clicking where you want the robot to go.

4 Costmap

When you open rviz you should see the following.



The black lines indicate the walls and obstacles in the environment. The dark grey inflates the area around the obstacles. To account for the width of the robot, valid paths should not pass through the dark grey space. As such, only the light grey areas are considered free space in the functions provided in planner.py.