CS-7630 Autonomous Robotics

Homework – Introduction to Mobile Robot Planning

Support: cedric.pradalier@georgiatech-metz.fr
Office hours: 10:00 to 18:00.

This homework assumes that you have completed the previous homeworks.

Submit the ros package (tar.gz or zip), by email by next Class, 10:00 to cedric.pradalier@georgiatech-metz.fr (don't forget to mention you group name)

The goal of this homework is to get some familiarity with mobile robot planning by extending a planner provided as a ROS package `occgrid_planner` with a set of three C++ nodes:

– `occgrid_planner`: receives an occupancy grid produced by gmapping and plan a path from the current location to a goal specified using Rviz "2D Nav Goal" interface. This planner is planning in a 2D grid, using an outer degree of 4 or 8. It does not pre-process the grid apart from converting it to an OpenCV image (which helps visualization). The path is then published as a sequence of poses using the ROS standard message nav_msgs/Path.
– `path_optimizer`: receives a path produced by the planner and transform it into a simple trajectory by computing bearing angles (as quaternion), time stamps and allocating a constant velocity and a feed-forward rotation speed to each pose. The output is a specific Trajectory message designed as a sequence of poses and twists.
– `path_follower`: receives the trajectory from the path optimizer and execute it to reach the goal. It applies the simple control law seen in the previous class.

All this setup assumes that a Gmapping setup us running (see the vrep_gmapping package if you don't want to use your own). Additionally, the launch files provided in the occgrid_planner instantiates the joystick controller used before, with the green button switching to manual control and the red button to automatic control.

A new task is available in the floor_nav package. It is called PlanTo, it sends a goal to the planner and wait for the destination to be reached. It would probably benefit from a bit more feedback to handle failures. Alternatively, you can input a goal from RViz by clicking the 2D Nav Goal button.

Use the scene rosControlStabilisation.ttt and build a complete map with Gmapping before starting Step 1 to 3. No work on the turtlebots this week, we'll actually work on following our trajectories next week.

Step 1: Obstacle Expansion for Point-Based Planning

As seen in the class, the configuration space obstacles are often expanded by the radius of the robot to justify planning a path for a single point. Currently, occgrid_planner is not doing this step, the first step of the homework is to implement it using the morphology operator of opencv:

http://docs.opencv.org/doc/tutorials/imgproc/erosion_dilatation/erosion_dilatation.html

Find specific examples where this operation leads to safer paths.

Step 2: From Disjkstra to A*

The current version of the occgrid_planner is a straight implementation of the Dijkstra algorithm for a graph supported by a grid. Modify the planner to add a heuristic and make it evolve from Dijkstra to A*.

What other change is required to completely take advantage of the A* heuristic?

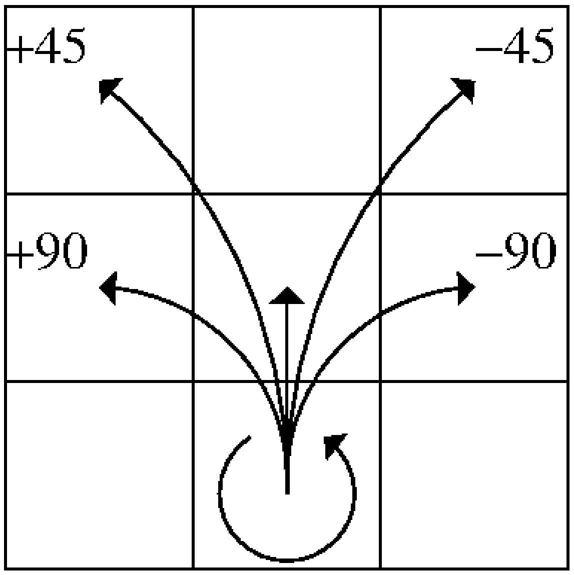
Step 3: Accounting for Heading

The default planner does not account for the robot initial heading and target heading. To solve this issue, the planner must now work in a discretised version of SE(3). The goal of this step is to modify the planner to implement SE(3) planning.

- Expand the map representation (og_) to 3 dimensions. There are (at least) two technical ways to achieve this, either by building an array (std::vector) of cv::Mat_<uint8_t>, or by using the multi-dimensional constructor of cv::Mat_<uint8_t>. http://docs.opencv.org/2.4/modules/core/doc/basic_structures.html#mat-mat

  Be careful to use the expanded obstacles for the initialisation of the map layers.
- Expand the neighbours list and related cost in the Dijkstra algorithm to point to the neighbours in SE(3). We will use transition with arc of circles forward and backward, as well as rotation on the spot.
- Evaluate how to update the heuristic from the 2D planner to account for the heading dimension.

Use (at least) 8 levels of angle (i.e. 45 degree resolution) similar to the following figure (you can decide if you want to allow reverse motions):



Note that rotation on the spot should be penalised in terms of cost.

In terms of implementation, note that cv::Point should become cv::Point3i (or cv::Point3f), and that predecessor should probably become a matrix of Vec3s. cell_value will also need to be extended to 3 dimension. Not that for multi-dimensional matrices, you can't access an element using a Point3i index.

Also, once the path is computed, the creation of the ROS path message will need to be modified to account for the heading. Check path_optimizer.cpp to see how to create quaternion from heading angle.

Report on the type of cost that provides the best result.

<u>If you have a lot of free time, the 2 next steps are typically required in a real system. It is unlikely that you will have the time to implement them at this stage, but keep them in mind for the final project.</u>

Step 4: From Path to Trajectory.

Bubblerob can move reasonably quickly but it reacts badly to high positive and negative acceleration. To improve its performance, modify `path_optimizer` to plan the trajectory accounting for a maximum positive and negative accelerations as well as rotation on the spot:

- Modify the velocity and timestamps allocated to the trajectory element based on ros param reflecting the maximum accelerations.
- Be careful to identify the spot where the trajectory is rotating on the spot (constant x and y). The robot must stop there, then rotate, then accelerate.

Once this is working, you're welcome to tune the gains of the path follower to improve the performance.

To help observe the tracking performance, `path_follower` publishes an error topic, representing (Δx, Δy, $\Delta \theta$). You can plot these values with rxplot along the twist values to check their convergence and the quality of the control.

Step 5 : Replanning

The current planner requires to know the complete map before starting to plan, because it considers unknown cells as occupied. Instead of making this conservative assumption, one could consider unknown cells as free and regularly re-plan a trajectory to account for new data once the map has been updated.

Modify occgrid_planner to be able to plan to an unknown cell, including replanning and decision when the unknown cell is finally decided occupied or unaccessible.