Jesús Pestana <jesus.pestana@pro2future.at>
Friedrich Fraundorfer <fraundorfer@icg.tugraz.at>

Practical Part 2 – Path planning for safe navigation

ROS related theory

- Path planning: C-Space and random sampling-based planners PRM and RRT
- Obstacle maps: path-length, occupancy and clearance
- Coding a ROS node with a path planning library and using an octomap as an obstacle-map:
  - ompl::base::ValidityChecker – interface between planner and obstacle-map
  - ob::OptimizationObjectivePtr – cost function determining properties of the calculated optimal path

Task Description

The goal of this task is to generate obstacle-free paths for the drone to reach a goal position without colliding with obstacles. Obstacle-free path generation is one of the main modules used by robots during autonomous operation to be able to navigate and approach locations of interest and perform their assigned tasks.

Path planning – for example for robotic manipulators – is performed in configuration space or C-Space. One point in C-Space corresponds to a clearly defined position of the robot, in the sense that the position for every part of the robot is specified by the coordinates of the point in C-space.

Intuition probably leads to suppose that for drones with hovering capability – that can fly in place without moving – the 3D position and rotation coordinates (or pose) would define a point in C-Space. However, due to the dynamical model that describes the drone motion as a function of the control inputs [R1], a point in C-Space is defined by the 3D position of the drone (e.g. {x, y, z} position coordinates). The heading (yaw) rotation angle can be defined separately depending on the task. The pitch and roll angles are defined by the acceleration profile and the shape of the trajectory and by the heading (yaw) rotation at each position along the trajectory.

For the path planning we will be using either of these algorithms: RRT [R2], PRM [R3], RRT* / RRTStar [R4] or PRM* / PRMStar [R4]. Concretely, their implementation on the Open Motion Planning Library (OMPL) [R5] will be used. These algorithms explore the C-Space of the robot using random sampling and checking whether the sampled positions are free or occupied by an obstacle.

Additional information that is useful for path planners is (1) the length of segments that connect 2 positions (this is not necessarily simple to calculate, consider for instance paths combining rotations and straight segments) and (2) the clearance (distance from a position/trajectory-segment in C-space to the nearest obstacle). The length provides a metric to be minimized, so that the resulting paths are short (ideally of minimal length). The clearance provides a second metric to be maximized, so that the paths stay away from obstacles (contributing to safety during navigation). In practice a combined metric is defined so that the generated paths have somewhat both properties. Note: if, in addition to the obstacle-free path generation, a metric is to be optimized; then the RRT* / RRTStar or the PRM* / PRMStar algorithms are to be used [R4].

Remark: the paths that we will generate on this practical can be navigated by the drone only slowly. For a drone to be able to navigate a path at high speed, the path needs to be continuous and several times differentiable (as a function of time). The third practical will deal with the generation of such smooth trajectories based on the sampled/rough trajectories gotten as a solution on this practical.

Recommended steps

1. Read up information about the Rapidly-exploring Random Tree (RRT) [R2], the Wikipedia entry can suffice to understand the algorithm in enough detail. Another possibility is to watch the following YouTube videos: [V1][V2][V3].
   Note: either the RRT* / RRTStar or the PRM* /PRMStar planners are recommended.
2. Download the OMPL library [SW1][R5] and read the example:
   demos/OptimalPlanning.cpp
   a. In particular, it is very important to understand the function implemented by the ompl::base::ValidityChecker . For the path planning it is required: (1) an obstacle map and (2) a function that communicates to the planner whether points are occupied or free. This is basically what the ValidityChecker object does, it serves as an interface between the planner and the obstacle map. When optimizing for clearance, then the clearance value also needs to be provided to the planner through the ValidityChecker. Rather than using the obstacle map from the previous practical part, it is better to start by using a toy obstacle map example (see Fig. 1).

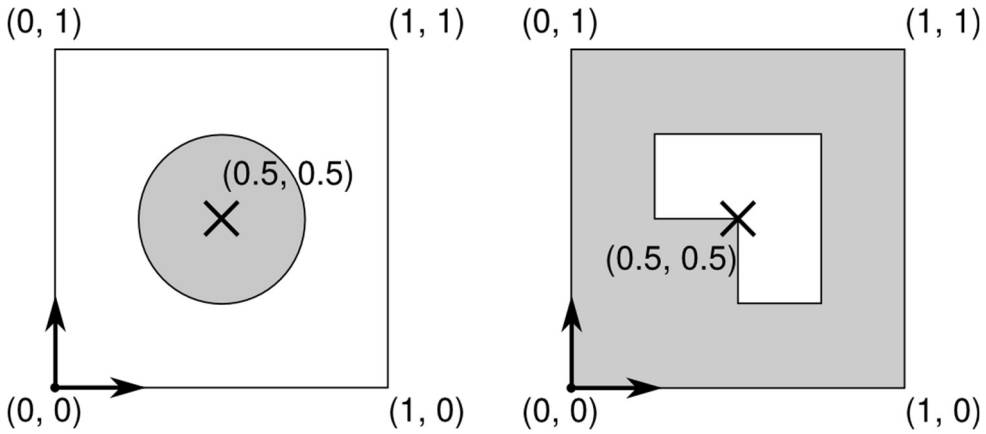

Fig. 1. 2D Toy obstacle map examples:
(left) map used in the OMPL demo demos/OptimalPlanning.cpp;
(right) another possible toy obstacle-map example

   b. The other important component of the example is the definition of the cost function. In our case, for the drones' lecture, we would like to implement a cost function that considers path-length and clearance and which also penalizes changes in altitude (so that the paths tend to be constrained to a horizontal plane).
3. Modify the path planner example code (OptimalPlanning.cpp) that internally uses the OMPL version of the RRT* / RRTStar planner to calculate paths in a toy 3D obstacle-map. Inspired by the Fig 1. (left), use the state space [0.,1.] x [0.,1.] x [0.,1.] with an obstacle sphere of radius 0.25 centered in the point (0.5, 0.5, 0.5).
   Note: you may prefer to start from our own version of the example code that has been recoded as a ROS Node (OptimalPlanning.cpp > dla2_path_planner). Our code is contained inside the zip-file "dla2_path_planner_y2019m12d20.zip", please follow the instructions inside the README.md file to compile and test the node. You may also use this code for step 5 and onwards. For step 5 the visualization in ROS Rviz is already included (at least for 2D paths). This code will allow you to save time and concentrate yourself on the more theoretical parts of the task - hopefully contributing to a better learning experience.

a. Remember that the ValidityChecker object serves as an interface between the planner and the obstacle-map. Modify the ValidityChecker to fit the 3D toy example.
b. Modify the cost function to: (1) minimize path length, (2) optimize path clearance and (3) prefer horizontal paths over 3D paths – add a cost that penalizes path changes in altitude, the easiest way to do this and still land on a cost function that is compatible with Dijkstra's algorithm [R10] is to modify the length calculation adding an over cost on the z direction, for instance: (Eq. 1) $length = \sqrt{\Delta x^2 + \Delta y^2 + 100 \cdot \Delta z^2}$.
Tip: in order to do this, start by modifying the motionCost function of the class PathLengthOptimizationObjectiveZPenalized in the helper_functions.h file of the provided code.
Tip 2: in our example code we have change the optimization objective from "10.0*lengthObj + clearObj" to "lengthObj + 10.0*clearObj", which provides paths further away from obstacles.

4. After calculating a path, print out relevant information about it on the terminal, for instance: (bool) point_reached, length, calculation time, minimum clearance, maximum clearance and average clearance.
5. Code a ROS Node using your code from the previous points that subscribes to 2 ROS topics providing (1) the current position and (2) the goal point (using the message type geometry_msgs/Point [R6]) and that publishes the paths [R7] to be visualized in ROS Rviz (using the message type visualization_msgs/Marker [R8]).

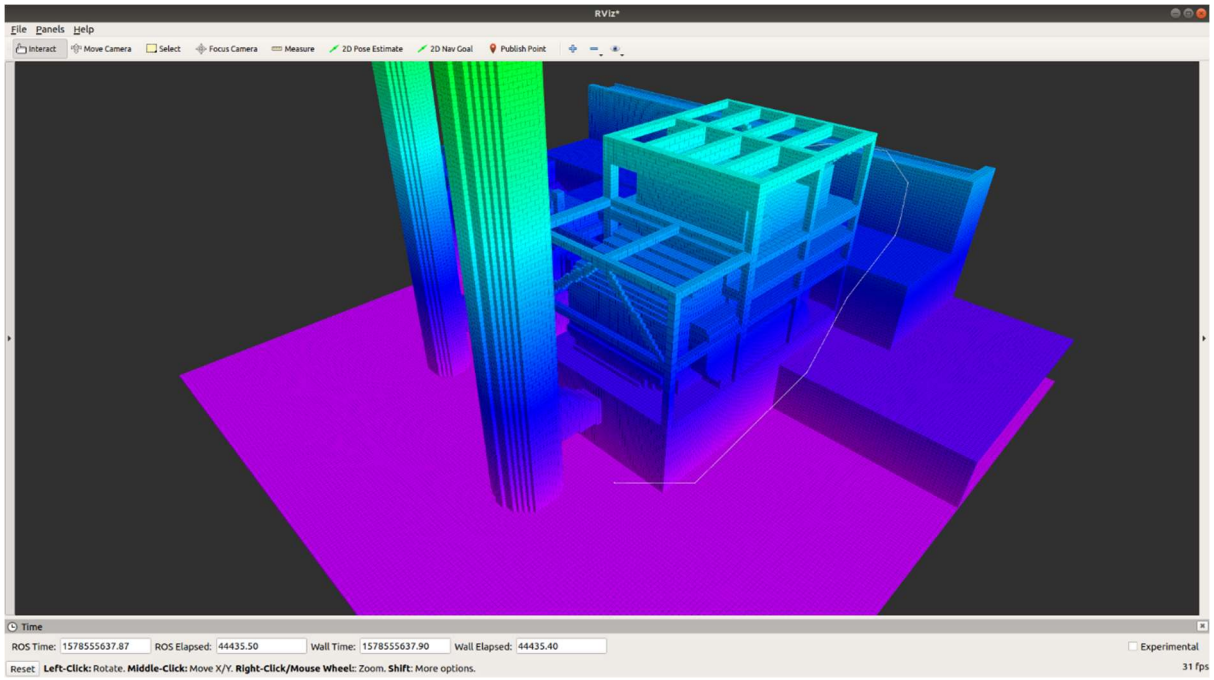

Fig. 2. power_plant.bt octomap [R9] and a planned path, obtained using the PRMStar planner. See the README.md file of the provided source-code to know how to visualize the octomap in ROS RViz. The shown path was planned from the point [0, 0, 3] to the point [10, -27, 15].

6. Our objective is to plan paths using an octomap as the obstacle map of the planner (see Fig. 2). In order to achieve this, make a second version of the code from the recommended steps 3-through-5 that takes as an input an octomap file, use power_plant.bt [R9] (or otherwise your octomap obstacle map generated from the practical part 1).

For the "ValidityChecker", you will need to check the occupancy on the octomap and the clearance with the dynamicEDT3D [SW2] library.
Note: You may want to modify your octomap by hand to include a ceiling. Following the advice from the tips section, may lead you to consider this task list:

a. Look for the repository holding the dynamicEDT3D library online
b. Browse through the examples provided by the developers of the library
c. Use a relevant example to extract the occupancy and clearance information from the octomap (by using the dynamicEDT3D library)
d. Then modify the ValidityChecker on the path-planner ROS Node

7. (Optional) When working through this assignment, you may have noticed the following 2 behaviors, which you may decide to solve:
    a. The path-planner is usually configured to use a given allotted time-duration to plan an obstacle-free trajectory from an initial point to a goal point. In order to have an early stopping before the allotted time has passed, it is possible to provide other stopping criteria. You may decide to explore the possibilities provided by the OMPL library regarding this issue.
    b. You can test the path-planner by providing an initial position outside the factory and a goal position inside (the factory in the power_plant.bt octomap is hollow). The desired behavior is in this case that the path-planner does not provide any solution trajectory in the allotted time. Depending on your implementation of the ValidityChecker, a trajectory may be calculated as valid that goes through the walls. To avoid this buggy behavior, you can test using the ray-casting functions of the octomap library to check for obstacles along trajectory-segments / path-segments.
8. Take screenshots of the visualization and write a small report for yourself, for the case that you need this code in the future (other than for assignment 3).

Tips

When coding a ROS Node it is important to divide the work in smaller tasks that can be implemented and tested incrementally, so that the code reaches maturity at the last step, but whose parts have by then already been tested. This is the criteria that was used to design the list of steps listed above.

Software libraries (for instance the OMPL) often include toy examples to describe how the functions of the library are to be used. It is important to read them and look for relevant examples. In this case your implementation can be based on the example stated above (demos/OptimalPlanning.cpp), or you can also look through the examples for something else that may be useful for the practical.

The visualization in ROS Rviz and the print-out of data (steps 5 and 7) are very useful to be able to evaluate the quality of the obtained results. It is also an excellent way to find bugs early on.

Take screenshots of the visualization and write a small report for yourself, for the case that you need this code in the future.

The OMPL library makes a heavy use of shared pointers, you can take a look at them here: https://www.learncpp.com/cpp-tutorial/15-6-stdshared_ptr/ . In this website you can see the different options on how to create and destroy objects using shared pointers. The important thing about *shared_ptr*, is that you can have several pointers pointing to the same object in memory, and be sure that the object is deleted only when the last *shared_ptr* stops pointing to the object.

Grading Scheme, assignment 2 score: 20 out of 60 points

The submission needs to include a small report – PowerPoint or PDF slides – about your work on this assignment, along with the source-code you have generated (or submit your catkin workspace source-code folder, the "$HOME/catkin_ws/src" folder of your virtual machine).

Most of the score obtained for this assignment (75%) is decided from the following content:

- Image(s) showing your successful "first tests" using the OMPL library (for instance, using Fig. 1-left as obstacle map).
- Image(s) of your generated obstacle-free trajectories in the power_plant.bt octomap (Fig. 2).
- Same as previous, but of trajectories showing good clearance (distance to any obstacles).
- Same as previous, but of trajectories that are optimized to avoid unnecessary movement in the altitude direction, preferring longer horizontal paths instead.

Choose any 2 points to work on from the following, which will decide the rest of your score (25%):

- Benchmarking the behavior of the trajectory planner for different values of the over-cost parameter for the altitude component of trajectory segments ("100" in Eq. 1).
- Find a test case that causes your trajectory planner to generate a trajectory that goes through a wall. To avoid this buggy behavior, for instance, define a MotionValidator for your trajectory planner, which overrides the default implementation and uses the ray-casting functionality of the Octomap library to determine that a trajectory segment is obstacle-free.
- Add a parameter to your trajectory planner to define the size of the drone (as a sphere). Then, show with examples that this parameter works properly, for instance, show that it prevents trajectories through small holes or windows.
- Find out how to make the OMPL library stop the trajectory search process, other than using a fixed time allotment for this task. Report on the advantages of your alternate solution.
- Consider the advantages and disadvantages of utilizing the RRT* versus using the PRM* planning algorithm. Select a performance metric, e.g., trajectory generation time, and do a small benchmark to show how one of them is better for some specific use.
- You can optionally discuss in detail issues you have encountered, or any other work which you have done for the assignment.

Software packages

[SW1] Open Motion Planning Library (OMPL): https://github.com/ompl/ompl

[SW2] dynamicEDT3D – see previous practical part
Lau, B., Sprunk, C., & Burgard, W. (2013). Efficient grid-based spatial representations for robot navigation in dynamic environments. *Robotics and Autonomous Systems*, *61*(10), 1116-1130.
Link: https://docs.ros.org/melodic/api/dynamic_edt_3d/html/

Videos

[V1] Robotics - 2.3.1.1 - Introduction to Probabilistic Road Maps: https://youtu.be/hFGhaSRV1zY

[V2] Lecture 7: RRT and D Star: https://youtu.be/TNajwFQP-Ys

[V3] Dijkstra's Algorithm – Computerphile: https://youtu.be/GazC3A4OQTE

References

[R1] Mellinger, D., & Kumar, V. (2011, May). Minimum snap trajectory generation and control for quadrotors. In 2011 IEEE International Conference on Robotics and Automation (ICRA2011).

[R2a] RRT - LaValle, S. M. (1998). Rapidly-exploring random trees: A new tool for path planning. Wikipedia: https://en.wikipedia.org/wiki/Rapidly-exploring_random_tree

[R2b] Lavalle's path planning book is freely available on the internet:
LaValle, S. M. (2006). Planning algorithms. Cambridge university press.
http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.225.1874&rep=rep1&type=pdf

[R3] PRM - Kavraki, L. E., Svestka, P., Latombe, J. C., & Overmars, M. H. (1996). Probabilistic roadmaps for path planning in high-dimensional configuration spaces. IEEE transactions on Robotics and Automation (1996).

[R4] RRT* & PRM* : Karaman, S., & Frazzoli, E. (2011). Sampling-based algorithms for optimal motion planning. The International Journal of Robotics Research (IJRR2011).
Freely available online here: https://journals.sagepub.com/doi/pdf/10.1177/0278364911406761

[R5] OMPL - Open Motion Planning Library (OMPL)
https://ompl.kavrakilab.org/
https://github.com/ompl/ompl
https://github.com/ompl/ompl"/blob/master/demos/OptimalPlanning.cpp
http://wiki.ros.org/ompl
Paper: Sucan, I. A., Moll, M., & Kavraki, L. E. (2012). The open motion planning library. IEEE Robotics & Automation Magazine (RAM2012).

[R6] geometry_msgs/PointStamped Message:
http://docs.ros.org/melodic/api/geometry_msgs/html/msg/PointStamped.html

[R7] rviz Display Types Marker: http://wiki.ros.org/rviz/DisplayTypes/Marker

[R8] ROS Rviz – GUI for real-time 3D Visualization of data: http://wiki.ros.org/rviz

[R9] Octomap examples (Note: you can visualize the octomaps in Rviz or in octovis):
(power_plant.bt) https://euroc.iphpbb3.com/forum/download/file.php?nxu=20112706nx57588&id=10
(other) http://ais.informatik.uni-freiburg.de/projects/datasets/octomap/

[R10] See requirements of the heuristic function h(n) in the description section of the Wikipedia article on the Astar / A* algorithm
Link: https://en.wikipedia.org/wiki/A*_search_algorithm#Description