Trajectory Generation and Control of a Quadrotor

MEAM 620 Project 1, Phase 3

1 Introduction

It is time to put everything together! In this Phase, you will need to autonomously control a simulated quadrotor through a 3D environment with obstacles. It is essentially an integration of everything that you have done in Phase 1 and 2, plus a few improvements to make your quadrotor fly better.

2 Quadrotor, Map, etc.

The simulated quadrotor is assumed to be a cylinder with radius of 0.15 m and height of 0.1 m. Other properties of the quadrotor are identical to Phase 1. Map definition is identical to Phase 2.

3 Trajectory Generation

You may have already noticed that in Phase 1, although your quadrotor was asked to precisely track a given trajectory, it was never able to do so. For cases such as sharp turns in the `diamond` trajectory, it is likely that your quadrotor will have some overshoot when making the turn (depending on the how fast the quadrotor is flying). Unfortunately, the optimal path output from your implementation of Dijkstra or A* usually contains many sharp turns due to the voxel grid based discretization of the environment. To improve the performance, you may apply trajectory smoothing techniques to convert sharp turns into smooth trajectories that the robot can track. One example will be to use minimum jerk or minimum snap polynomial segments as discussed in class. These and other techniques are also reviewed in [Chapter 3, PC 2017] [1]. You will have to determine the start and end points of the polynomial curve as well as all other boundary conditions.

Note that you will need to decide on a velocity profile to turn the path from Dijkstra or A* into a trajectory. The speed of the robot does not need to be constant.

Complete the implementation of `trajectory_generator.m`. The function `trajectory_generator(...)` takes a path from Dijkstra or A* and converts it into a trajectory as a function of time. You are allowed to use the map for trajectory generation. You should pre-compute values as much as possible and avoid fitting polynomial curves in every call of the function (e.g. persistent variables in Matlab functions is one way of solving this problem).

4 Collisions

Your quadrotor should fly as fast as possible. However, a real quadrotor is not allowed to collide with anything (video). Therefore, we have zero tolerance towards collision – if you collide, you crash, you get zero for that test. For this part, collisions will be counted as if the free space of the robot is an open set; if you are on the boundary of a collision, you are in collision.

As a result of smoothing, your trajectory may deviate from the straight line path connecting the points generated by your graph search algorithm. Therefore, you should make good use of the `margin` parameter and set your speed carefully. Please be aware that the robot is assumed to be a cylinder. You should make

[1] P. Corke, Robotics, Vision and Control: Fundamental Algorithms in Matlab, Springer Tracts in Advanced Robotics, 2017

sure that no part of the robot collides with any obstacles. However, we guarantee that for all the maps used for testing, there will always be openings that allow a cylinder with a radius of 0.5 m and height of 0.5 m to pass through.

5 Coding Requirements

First, you should generate your optimal path following the same procedure as Phase 2. Then, the path will be converted to a trajectory and tracked by the quadrotor. An example sequence is:

```
map = load_map('maps/map1.txt', 0.1, 1.0, 0.2);
start = [0.0, -4.9, 0.2];
stop = [8.0, 18.0, 3.0];
path = dijkstra(map, start, stop, true);
trajectory_generator([], [], map, path);
trajectory = test_trajectory(start, stop, map, path, true);
```

`trajectory = test_trajectory(...)` will return the actual trajectory executed by the robot. See comments in the code for details. You are free to add visualization code to this file to see intermediate results, however, we will not use your version when testing your code. Make sure that you can run your code with the original version of `runsim.m`. Performance evaluation will be based on the output `trajectory`, which contains 100 Hz samples of the actual trajectory of the quadrotor. You are free to reuse any or all of your Phase 1 and 2 code. Your Phase 3 implementation should be self-contained. Here are important coding requirements for Phase 3:

1. You are not allowed to change input and output formats of any functions.
2. Copy over your Phase 1 code (`controller.m`, `crazyflie.m`, etc.)
3. Copy over your Phase 2 code (`dijkstra.m`, etc.)
4. You may need to slightly change your Phase 1 and/or Phase 2 implementation such that it works within the current code base.
5. Complete the implementation of `trajectory_generator.m`, being sure to take care of initialization (see how the first call is made in `runsim.m`).
6. Please read through `test_trajectory.m` carefully and make sure that you can run your code with the original file.
7. An example testing script is given in `runsim.m`, we will use the same sequence to test your code against approximately 5 different maps.

6 Grading

The majority of your grade is determined by your quantitative performance on the test maps:

1. How long the quadrotor takes to reach the goal (excluding planning time).
2. The length of the actual trajectory of the quadrotor.

Remember, if you collide, *you get zero for that map*, so do not go too crazy.

In addition to these automated tests, you must also create an original, challenging `mymap.txt` test environment to show off the performance of your system. Your `mymap.txt` must follow the map format outlined in the project 1 Phase 2 handout so that it can be imported using `load_map` and visualized with `plot_path`. Submit a single .png image of your `mymap.txt` to Canvas capturing an interesting view of the environment (manipulate the 3D perspective of your map plotted with `plot_path` and then save the figure as a .png file) you've created and your quadrotor's trajectory. Particularly interesting solutions/slides may be shared with the class. Please include your `mymap.txt` file in your 'turnin' submission.

6.1 Submission

When you are finished you may submit your code via turnin. The project name for this assignment is titled "`proj1phase3`" so the command to submit should be
`turnin -c meam620 -p proj1phase3 -v *`
Your turnin submission should contain:

1. A README detailing anything we should be aware of.
2. All necessary files to run your trajectory planner using the automated testing script `runsim`.
3. Your challenging `mymap.txt` file

Shortly after submitting you should receive an e-mail from `meam620@seas.upenn.edu` stating that your submission has been received. You can check on the status of your submission at https://alliance.seas.upenn.edu/~meam620/monitor/. Once the automated tests finish running, you should receive another e-mail containing your test results. This report will only tell you whether you passed or failed tests; it will not tell you what the test inputs were or why your code failed. Your code will be given at most 10 minutes to complete all the tests. There is no limit on the number of times you can submit your code.

Please do not attempt to determine what the automated tests are or otherwise try to game the automated test system.

Also remember to submit your single PowerPoint slide to Canvas as a .pdf.