Trajectory Generation and Control of a Quadrotor

MEAM 620 Project 1, Phase 3

February 3, 2022

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. You will integrate everything that you have done in Phase 1 and 2 as well as implement an improved trajectory generator.

2 Quadrotor, Map, etc.

The simulated quadrotor is assumed to be a sphere with a radius of 0.25 m. Other properties of the quadrotor are identical to Phase 1. The map specification file format is the same as in Phase 2.

3 Trajectory Generation

By now you have a controller that can stabilize the quadrotor to some desired setpoint and a graph search algorithm that can find obstacle free paths through a given environment. All you need to do is stitch the two together, right? Well, not exactly. Recall, in Phase 1 your quadrotor was tasked with visiting a sequence of waypoints. Depending on your `WaypointTraj` implementation you may have noticed that while the quadrotor did visit the desired waypoints it did not do so in a very precise manner. For cases when the lines connecting given waypoints formed sharp corners, it is likely that your quadrotor had some overshoot when making the turn or worse, became unstable and flew away. 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 further compounding this issue. There must exist a post-processing step wherein the points you get from your graph search algorithm are tidied up and connected in a manner suitable for flight. This is the task of trajectory generation.

There exist many ways to plan dynamically feasible trajectories through obstacle filled environments and it remains an active field of research in robotics. One method is to apply trajectory smoothing techniques to convert sharp turns into smooth trajectories that the robot can track. This involves removing redundant points from you graph search output and constructing polynomials to smoothly interpolate between them. A specific example is using minimum jerk or minimum snap polynomial segments as discussed in class. You will need to determine the start and end point of each polynomial segment (using your graph search points), allocate travel time to each segment, and determine what boundary conditions should be enforced.

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 be careful how you select your segment times (i.e. how fast the quadrotor is commanded to go). On the other hand, increasing `margin` too much can cause shortcuts to be closed off or worse: cause the map to become infeasible. Another dial you can turn is the grid `resolution`. A smaller resolution will result in a longer planning time but could unlock shortcuts to your goal that dramatically cut down your travel time. Experiment with each parameter and tune them to suit your graph search implementation and controller. Remember, in the end you should make 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 sphere with radius 0.5 m to pass through.

5 Coding Requirements

You will be provided with a project packet containing the code you need to complete the assignment. For this phase you will need your graph search algorithm and controller from Phases 1 and 2. After extracting the project packet be sure to replace `proj1_3/code/se3_control.py` and `proj1_3/code/graph_search.py` with your implementations.

You will need to complete the implementation of the `WorldTraj` class in the provided `world_traj.py` file. `WorldTraj` functions in a similar manner to the previous `WaypointTraj` class with a few added inputs. As usual, the provided code stubs are thoroughly documented and should be your first point of reference. In summary:

1. Copy over your Phase 1 `se3_control.py` (`waypoint_traj.py` need not be copied as it is now obsolete).
2. Copy over your Phase 2 `graph_search.py` (a fresh `occupancy_map.py` will be provided).
3. Now that all your code resides in `proj1_3/code` be sure your `import` commands are adjusted accordingly (e.g. `from proj1_2.code.occupancy_map` becomes `from proj1_3.code.occupancy_map`).
4. Complete the implementation of `code/world_traj.py`.
5. Use the provided `code/sandbox.py` to aid in tuning and analysis.
6. Test your implementation on a collection of given maps using `util/test.py`

6 Grading

For this assignment your grade will be determined by automated testing. You must find trajectories through six obstacle filled environments which your quadrotor must then quickly and accurately follow without collision. Performance will be measured in terms of the in-simulation flight time from start to goal. For each map, you will earn 9 points for a safe flight and up to an additional 6 points for a fast completion time. The time targets for each map will be in your Gradescope report; attaining full marks on every trajectory may be extremely challenging.

In addition to these automated tests, you must also create an original, challenging `mymap.json` test environment to show off the performance of your system. Your `mymap.json` must follow the map format used in Phase 2 so that it can be imported using `World.from_file('mymap.json')`. Include with your submission (detailed below) your `mymap.json`. Particularly interesting maps may be shared with the class. You will present your map and solution in a separate summary slide.

7 Submission

7.1 Code Implementation

When you are finished, submit your code via Gradescope which can be accessed via the course Canvas page. Your submission should contain:

1. A `readme.txt` file detailing anything we should be aware of (collaborations, references, etc.).
2. Files `se3_control.py`, `graph_search.py`, `occupancy_map.py`, `world_traj.py` and any other Python files you need to run your code.
3. Your `mymap.json`.

Shortly after submitting you should receive an e-mail from `no-reply@gradescope.com` stating that your submission has been received. Once the automated tests finish running the results will be available on the Gradescope submission page. 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 game the automated test system. This is an individual submission and must reflect your own work. You are encouraged to discuss the project with your peers in detail, but you may not share code. Please acknowledge any assistance in detail in your `readme.txt`.

7.2 Summary "Slide"

Submit your single "slide" to the separate Gradescope assignment summarizing your approach. Please include the following:

1. A 3D perspective illustration showing your original map, the planned trajectory, and the flown path.
2. Briefly describe the key components of your solution, emphasizing the new elements or changes since P1-1 and P1-2.
3. Briefly comment on how you selected your map resolution and margin.