# CMSC 828T
# Vision, Planning and Control in Aerial Robotics
# Project 1 Phase 3 (P1Ph3):
# Trajectory Generation and Control of a Quadrotor
# Due on 11:59:59PM on Oct 1st, 2017

Prof. Yiannis Aloimonos, Nitin J. Sanket,
Kanishka Ganguly, Snehesh Shrestha

September 26, 2017

## 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. Files can be downloaded from the course website.

## 2 Quadrotor, Maps, 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 2. Map definition is identical to Phase 1.

## 3 Trajectory Generation

You may already notice that in Phase 2, 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 drone can track. One example will be the 5th order polynomial fitting covered in Chapter 3 [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 a constant.

Complete the implementation of `trajectory_generator.m`. `trajectory_generator(...)` takes a path from Dijkstra or A* and convert it into a trajectory as a function of time. You are allowed to use the map for trajectory generation. You are essentially modifying the trajectory files in phase 2 (`circle.m or diamond.m`) such that they are able to accept externally specified paths. You should pre-compute as much quantities as possible and avoid fitting polynomial curves in every call of the function.

# 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.

After phase 2, you should already have an idea of how well your controller works. Additionally, trajectory smoothing may also deviate the actual trajectory from the planned path. 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 sure that no part of the robot collides with any obstacles.

However, we guarantee that for all the testing maps, with the specified start and goal locations, there will always be openings that allow 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 1. 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);
init_script;
trajectory = test_trajectory(start, stop, map, path, vis);
```

`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:

- You are not allowed to change input and output formats of any functions.
- Put your phase 1 code (`dijkstra.m, load_map.m, collide.m, controller.m etc.`) into main folder (Empty starter code files have been provided to you. Add any dependent m-files you might have created.
- You may need to slightly change your Phase 1 and/or Phase 2 implementation such that it works within the current code base. This is your change to fix any issues you might have had in your phase 1 and/or phase 2. Don't wait for phase 4 as it will be on a real drone, and it won't be fun trying to fix it then.
- Complete the implementation of `trajectory_generator.m`.
- You may need to change `init_script.m` to initialize your trajectory generator.
- Please read through `test_trajectory.m` carefully and make sure that you can run your code with the original file.
- An example testing script is given in `runsim.m`, we will use the same sequence to test your code against approximately 6 different maps. Three maps (which may or may not be part of 6) has been provided for you to for development and debugging.

# 6 Grading

Your grade will be determined by:

- How long does the quadrotor take to reach the goal (excluding planning time).
- Comparison between the length of the actual trajectory of quadrotor and the length of the shortest path (path from Dijkstra or A*).

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

## 6.1 Submission

When you are finished you must submit your code via CMSC 828T submit section. You should create a folder called `code` and copy all the submissions files listed below into it, zip it, and submit `code.zip`. Please note the zip file needs to be `.zip` format. Any other format is not valid.

Please note:

- Do NOT add or submit any sub-folders.
- Do NOT submit any visualization code. If you have any either remove them or comment them out.

- Only include the files that are listed below and any new dependent m-files you might have created.
- Do NOT include `maps folder, util folder, runsim.m, plot_path.m, quadEOM.m, quadEOM_readonly.m, test_trajectory.m`, or any other files that are not necessary and was created only for your testing/ debugging.

Your submission should contain:

- A README.txt detailing anything we should be aware of.
- All necessary files inside one folder `code` such that we can just run the automated testing script `runsim` to see your results. The expected files for phase 3 submission are:
  - `dijkstra.m`
  - `drone250x.m`
  - `collide.m`
  - `controller.m`
  - `init_script.m`
  - `load_map.m`
  - `trajectory_generator.m`
  - And any other new m-files you might have created that is necessary for running your code.

For this phase of the project, we will open up submission a bit sooner and you will be allowed a total of 10 submissions in total. Please do not plan to use submit server as your test bench. Please use them wisely. Submit server will provide some automated reports. This may take some time depending on the number of tests we are running, the state of the server, and the number of items in the queue. So please be patient with the response. If you have any issues, please raise it on Piazza.

# 7 Acknowledgements

The project has been adapted from the University of Pennsylvania MEAM 620 course.

# 8 Collaboration Policy

You can discuss with any number of people. But the solution you turn in MUST be your own. Plagiarism is strictly prohibited. Plagiarism checker will be used to check your submission. Please make sure to **cite** any references from papers, websites, or any other student's work you might have referred.

# References

[1] Peter Corke. *Robotics, vision and control: fundamental algorithms in MATLAB*, volume 73. Springer, 2011.