# Part 1: A ‘practical’ approach to flying fast with trajectory generation and control[1]

## Improvements made to achieve 66.2s total flight time

1. **Adjusting map resolution and margin:** in order to find the most efficient path through the map, I experimented with adjusting margin and resolution. By reducing the size of the cells in the map (higher resolution) from 0.25 to 0.2, my graph search algorithm was able to find shortcuts and create straighter paths. This is most likely the case in switchback and staircase where I noticed a significant time improvement. I increased margin slightly from 0.5 to 0.6 as the path needed to be further from obstacles to avoid collisions due to my path smoothing (I used a fairly large tolerance for my path smoothing algorithm).

2. **Path smoothing:** to reduce the number of waypoints on my path, I first implemented my own smoothing algorithm. The algorithm iterates through the waypoints sequentially, and checks the distance of this waypoint and all previous excluded waypoints from the hypothetical path between the last included waypoint and the next waypoint. If any of these distances exceeded a limit, the current waypoint was included. This worked, but since I evaluated the waypoints sequentially, I didn’t necessarily select the most extreme points/corners of the path (my algorithm might include a waypoint close to the most extreme point). Therefore, in the end I just implemented the Ramer-Douglas-Peucker (RDP) algorithm, as RDP will always include the most extreme locations. This is important to avoid cutting corners and keep my margin as low as possible. Map margin and path smoothing tolerance was tuned together to create the smoothest paths possible with no collisions.

3. **Non-constant time allocation of path segments:** Distance-based distribution of time for each segment is inefficient, as the quadcopter can accelerate and decelerate over long segments to cover them faster. Therefore, time is allocated according to: $t = 2\frac{\sqrt{as+u^2}}{a}$ where s = segment distance; u = minimum velocity (adjustable); a = acceleration & deceleration in segment
The equation is essentially the time to cover a distance given constant velocity and acceleration, but since I have acceleration and deceleration in each segment, the equation has been modified to account for that.

4. **Variable velocity and constant acceleration in each segment:** Instead of creating a smooth trajectory with splines such as minimum jerk or minimum snap, I kept position in a straight line between each waypoint, but layered in constant acceleration/deceleration between each waypoint. The quad accelerates in the first half of the segment and decelerates in the second half. Position is still in a straight line between each waypoint, but is no longer linear with time.

5. **Acceleration and velocity feed forward:** The controller requires some time to react to changes in acceleration and velocity, therefore will always lag the given commands. Furthermore, since I still kept a minimum speed at each waypoint rather than stopping to zero, to avoid overshooting, I fed-forward acceleration and velocity by ~0.4s to account for controller lag and smooth out the cornering.

6. **Other fine tuning and improvements:** I improved the gains on my controller by slightly increasing the Kd term (in proj1_1 slides I mentioned it was under-damped slightly). I also cut the z axis Kr and Kω to avoid motor saturation with yaw control. I experimented with other things as well, such as varying time based on the ‘jaggedness’ of the path and slowing down paths with a lot of vertical accel/deceleration to avoid going inverted or the quad not keeping up with vertical acceleration going up.

1 This is the approach in my active submission on Gradescope.

## Custom map with planned trajectory and flown path

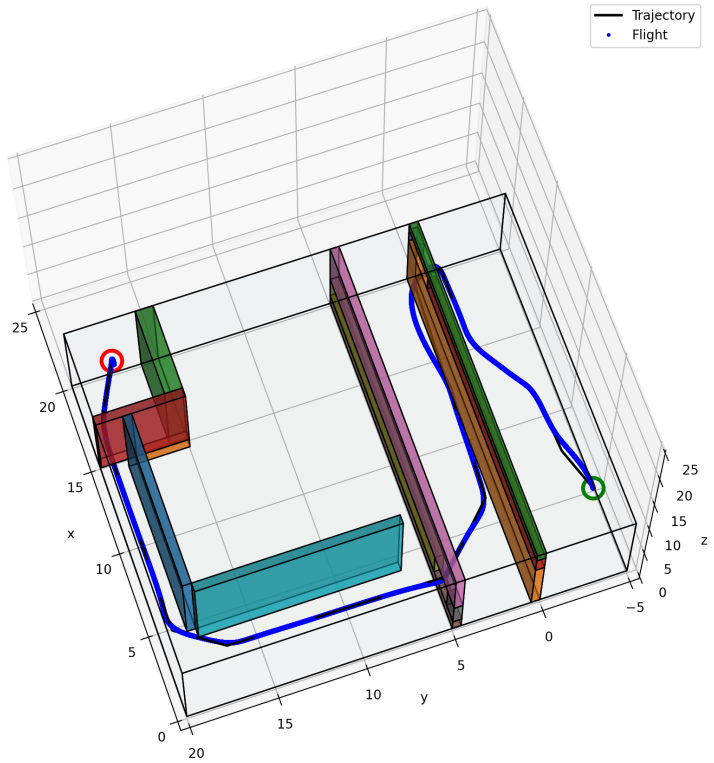


**Figure 1: 3D perspective of custom map, planned trajectory and flown path** (Planned trajectory in black, flight path in blue)

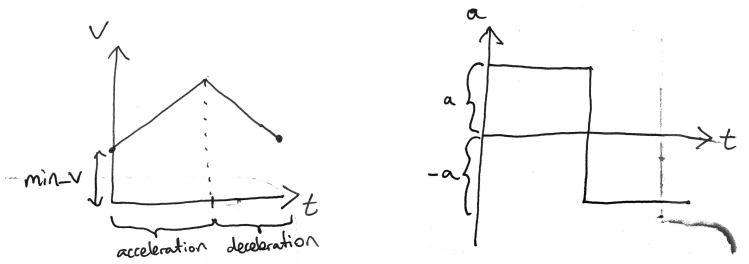


**Figure 2: Illustrative diagram of velocity (left) and acceleration (right) in each segment**. Time for each segment was calculated using this too.

# Part 2: An more robust, academic approach to going even faster

Despite getting a good time using the methods described on the previous slide, my appetite for learning was not quite satisfied, especially when Jimmy posted on Piazza that he has seen the Maze map being flown as fast as 4.5 seconds. As someone with an insatiable ”need for speed”, the thought of someone else’s quadcopter flying faster than mine was simply unacceptable. So I did what any reasonable software engineer would do: I spent my weekend implementing a new solution to save 2.44s of flying time[1]. Sadly only got down to 4.9s, not 4.5s. One day though, one day.

## 1. Replacing the occupancy grid with a visibility graph

Visibility graphs offer the advantage over occupancy grids that they can find the actual shortest path between a starting and stopping location, as they use the vertices and edges of the obstacles to generate the grid nodes rather than having grid nodes placed at a pre-set spacing. Empty space also doesn’t require unnecessary nodes. Therefore, they can provide better performance for maps with few obstacles and large spaces. The number of nodes in an occupancy grid scales with the size of the grid, whereas in a visibility scales with the number of obstacles and does not depend on map size.

I decided to implement a 3D visibility graph to get the actual shortest path and fewest number of waypoints to define that path. I still had a resolution factor to define the spacing of the nodes along the edges of the blocks, but it could be significantly higher at 0.5. The margin was as low as 0.28, allowing me to find even shorter paths.

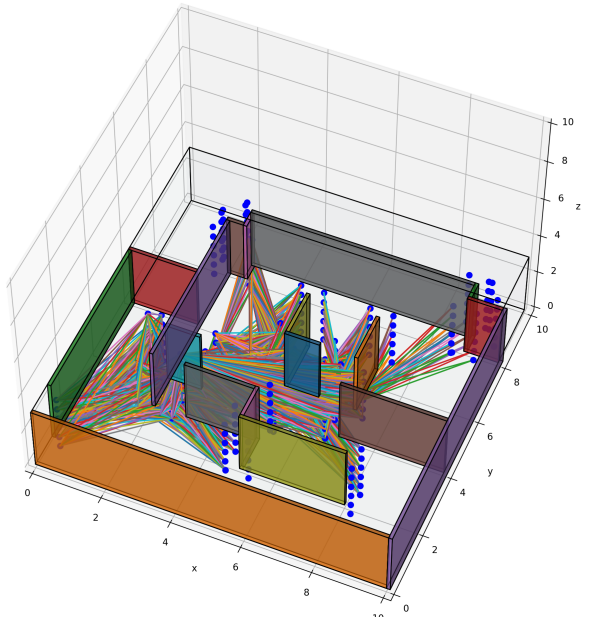

**Figure 3: visibility graph for showing nodes & edges**

Note that finding the neighbors in a visibility graph is computationally expensive, as it requires ray tracing between a node and every other node to check for any obstacles in the way. Therefore, I only calculate edges when a node is expanded, hence in fig. 3 some nodes do not have edges as they are never expanded.

The map Stairwell has presumably too many blocks for this to be efficient and times out. Certainly further optimizations could be done on my approach and implementation, but this was definitely out of scope of this project. Maybe in a future academic paper?

1 2.44s on Maze, other maps had time savings too. Some maps would require increasing the margin to avoid collision.

## 2. Minimum jerk trajectory

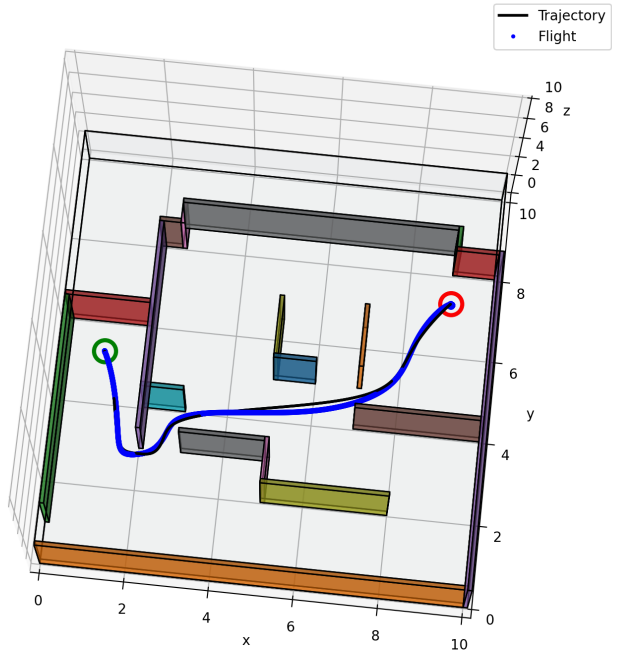


**Figure 4: Min jerk trajectory for Maze**

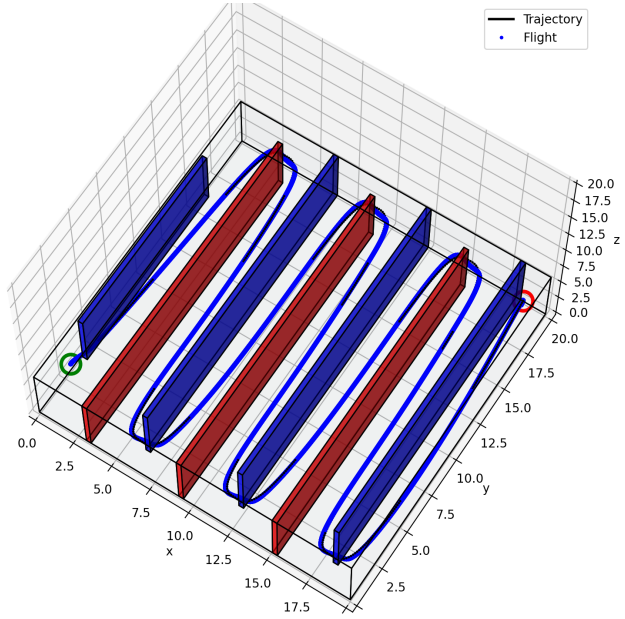


**Figure 5: Min jerk trajectory for a custom map**

I also implemented the minimum jerk trajectory from the lecture slides to further improve my solution. To constrain the trajectory to my desired path, ideally I would implement a corridor constraint. But I found that simply adding in a midpoint on each segment also worked. Since the time allocation stayed the same as in part 1, every waypoint added causes a decrease in quadrotor speed. So for really long segments, I did not add the midpoint in as it was unnecessary (the trajectory already closely followed the path). The midpoint was offset from the centerline based on the direction of the incoming and outgoing segments. On really short segments, this allowed quadrotor to still follow a curve between waypoints.

My minimum jerk trajectory required almost no acceleration / velocity feed forwards for the quad to fly exactly the trajectory as planned. I also included a bunch of other minor improvements (e.g., in the time-based allocation I accounted for more time in the first and last segments). Combining my visibility graph with minimum jerk trajectories allowed me to find a ‘shortcut’ in Maze and fly that trajectory, in 4.9s.

# Leaderboard

Search

| RANK | SUBMISSION NAME | TOTAL TIME, S | MAZE, S | OVER_UNDER, S | WINDOW, S | SLALOM, S | STAIRWELL, S | SWITCHBACK, S |
|---|---|---|---|---|---|---|---|---|
| **1** | **Lukas** | **66.2** | **7.38** | **10.28** | **7.72** | **16.48** | **10.78** | **13.52** |
| 2 | [illegible] | 71.5 | 8.09 | 12.14 | 5.24 | 15.25 | 14.84 | 15.9 |
| 3 | [illegible] | 78.9 | 10.6 | 13.54 | 6.8 | 17.05 | 16.94 | 14 |
| 4 | [illegible] | 79.6 | 8.53 | 10.98 | 9.25 | 17.1 | 11.75 | 22.03 |
| 5 | [illegible] | 82.5 | 11.11 | 14.11 | 9.15 | 17.97 | 18.29 | 11.87 |
| 6 | [illegible] | 83.4 | 11.5 | 15.13 | 7.8 | 18.05 | 16.33 | 14.61 |
| 7 | [illegible] | 83.8 | 9.66 | 13.09 | 9.12 | 18.12 | 14.75 | 19.05 |
| 8 | [illegible] | 84.7 | 8.81 | 11.49 | 9.77 | 18.64 | 12.31 | 23.72 |
| 9 | [illegible] | 85.8 | 9.82 | 14.13 | 8.33 | 16.67 | 16.08 | 20.75 |
| 10 | [illegible] | 86.6 | 10.43 | 14.9 | 6.42 | 20.71 | 19.33 | 14.79 |
| 11 | [illegible] | 87.4 | 8.53 | 11.13 | 9.75 | 19.65 | 12.47 | 25.9 |
| 12 | [illegible] | 90.7 | [illegible] | [illegible] | 9.67 | 20.86 | 12.6 | 27.07 |

(60 entries in total)