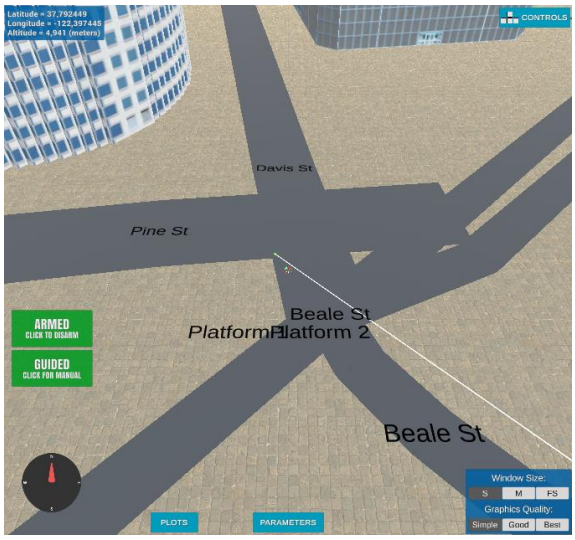
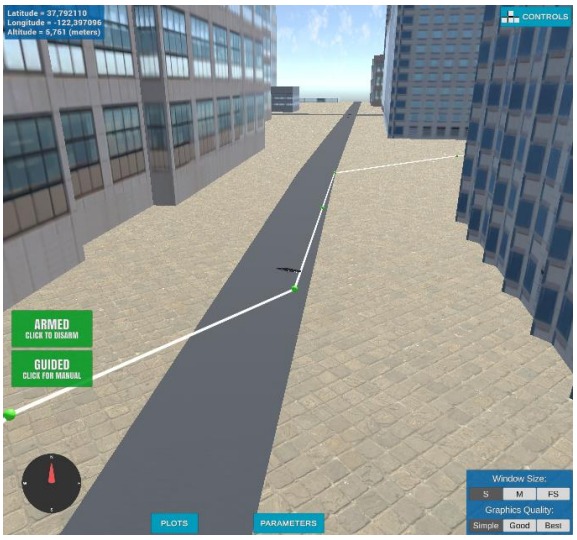
Project: 3D Motion Planning

Student: Enrico Baracaglia





Required Steps for a Passing Submission:

1. Load the 2.5D map in the colliders.csv file describing the environment.
2. Discretize the environment into a grid or graph representation.
3. Define the start and goal locations.
4. Perform a search using A* or other search algorithm.
5. Use a collinearity test or ray tracing method to remove unnecessary waypoints.
6. Return waypoints in local ECEF coordinates (format for `self.all_waypoints` is [N, E, altitude, heading], where the drone's start location corresponds to [0, 0, 0, 0].
7. Write it up.

Here I will consider the rubric points individually and describe how I addressed each point in my implementation.

Write-up / README

1. Provide a Write-up / README that includes all the rubric points and how you addressed each one. You can submit your write-up as markdown or pdf.

The current document! Below I describe how I addressed each rubric point and where in my code each point is handled.

Explain the Starter Code

1. Explain the functionality of what's provided in `motion_planning.py` and `planning_utils.py`

The main differences between ***backyard_flyer_solution.py*** and ***motion_planning.py*** are:

1. We have an additional state called PLANNING;
2. The function *calculate_box()* has been replaced by *plan_path()*, used to calculate the new path between the buildings of the city; in this function we also define the target altitude and the safety distance to keep between the drone and the obstacles;
3. A simple function called *send_waypoints()* has been added in order to visualise the planned waypoints in the simulator.

As regards planning_utils.py, it contains:

1. The *create_grid* function used to generate a 2D grid representation of the obstacles. The grid contains zeros when there is feasible space and ones when there is an obstacle.
2. The class Action, used to return a list of the valid actions that the drone can perform in order to reach the desired waypoints
3. A* function, used to generate the path from grid_start to grid_goal, given an heuristic function
4. Heuristic function

3. Set your global home position

Here, the method used to read and get the first line (raw1) of the csv file was by using the *csv.reader*. Then, from this raw, lat0 and lon0 are extracted and used to set global home.

4. Set your current local position

Here, I use *self.global_position* to retrieve the drone's current location in global coordinates and then this location is converted into local position by using the *global_to_local()* function.

5. Set grid start position from local position

In order to set the *grid_start* position, I used the *self.local_position* values and subtracted the north and east offsets. After that, the float numbers have been rounded to integers in order to be used while calculating the path using the grid.

4. Set grid goal position from geodetic coordinates

In order to set the goal position, I use the *grid_start* position and add random values within the limits of the north and east offsets. Moreover, we make sure that the chosen point is not within an obstacle.

5. Modify A* to include diagonal motion (or replace A* altogether)

Here, the A* function in *planning_utils()* has been modified in order to include diagonal motions on the grid that have a cost of sqrt(2). As a first step, four new actions (North-East, North-West, South-East, South-West) have been added in the Action class, specifying the deltas of each action and its relative cost (sqrt2 = 1.14). Secondly, in the *valid_actions()* function, we add some if-statements to check that the diagonal actions do not make the drone fly off the grid and into obstacles. During planning, if the diagonal actions are valid, they are kept, otherwise they get removed from the list of valid actions.

6. Cull waypoints

For this step, I use a collinearity test with an epsilon constant (epsilon = 5) to shorten the full path and eliminate the unnecessary collinear waypoints.

After the full path gets calculated, it goes into the *shorten_path()* function in *planning_utils()*, which returns a shorter list of waypoints after having performed a collinearity check between each 3 successive points of the full path.

Executing the flight

1. Does it work?

The code runs successfully, allowing the drone to go from *grid_start* to *grid_goal* without hitting any obstacles along the way. One example of successful trial is presented in the images on the first page of this document. Moreover, different goal locations have been tried, starting from various points in the city. The only issue that has been encountered is that, if the start and goal locations are too far apart from each other, the simulator cuts off the connection without performing the flight. This is probably due to the quite intensive calculations that the simulator has to go through. One solution was to simply use a closer goal location.