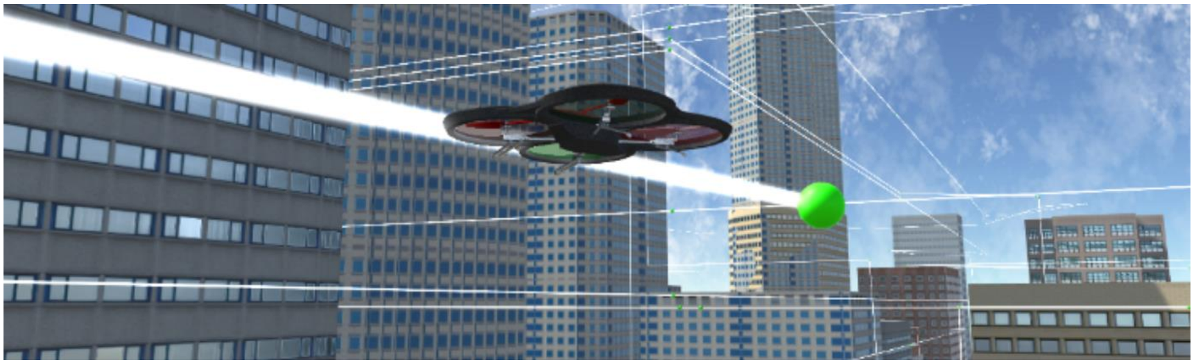


Project: 3D Motion Planning

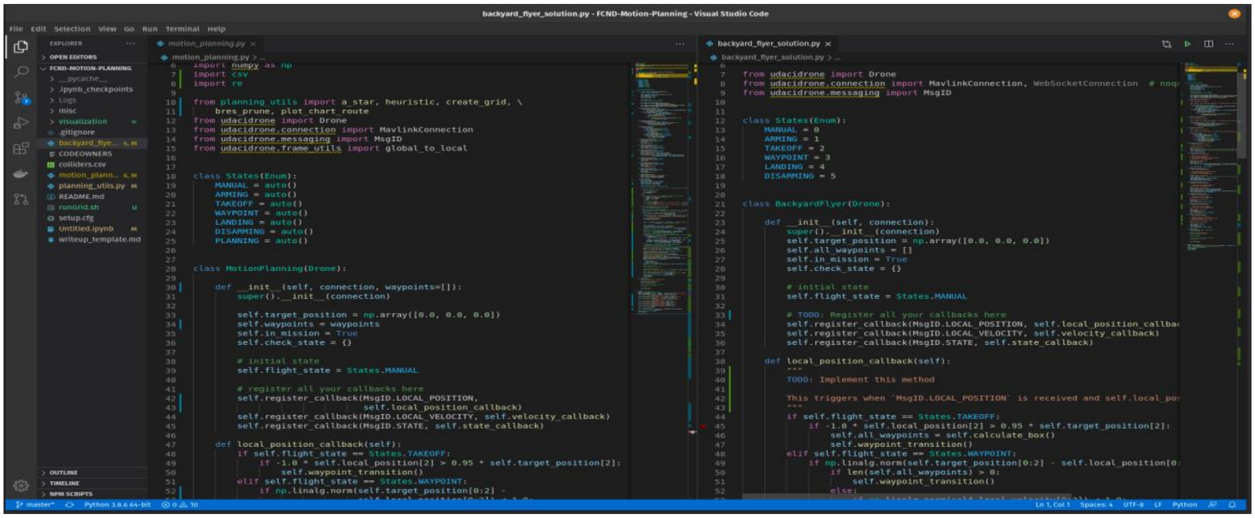
Rubric:

1. **Writeup.** Explaining how each point of the rubric was addressed.
2. **Verify** that `` `motion_planning.py` `` is a modified version of `` `backyard_flyer_solution.py` `` for simple planning. Verify and compared.
3. **Set home position**. Read first line of the csv file, extract lat0 and lon0 as floating-point values and use de self.set_home_positon method to set global home.
4. **Retrieve current position** in geodetic coordinates from `` `self. Latitude/longitute/altitude` ``. Then convert to local position using `` `self.global_home` ``.
5. **Modify harcoded** map center and other locations to some arbitary positions on the grid given geodetic coordinates.
6. Write and explain **search algorithm**. Used a simple grid A* implementation.
7. Cull waypoints from the path
8. **Tested** solution

Writeup.

You’re reading it!

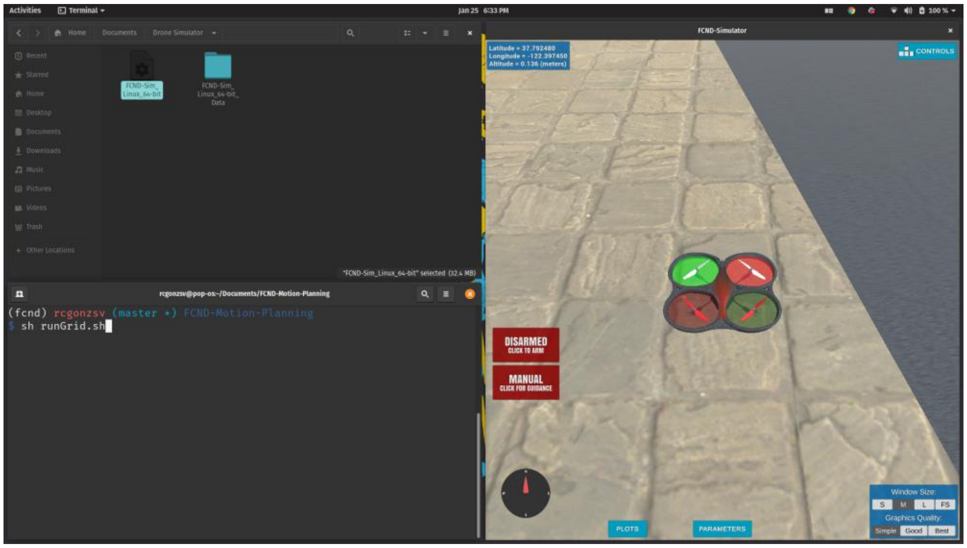
I took the starter code from previous project `***backyard_flyer_solution.py***` and modified in order to load the 2.5 map in the **colliders.csv** for environment description and discretize it into a grid or graph representation.



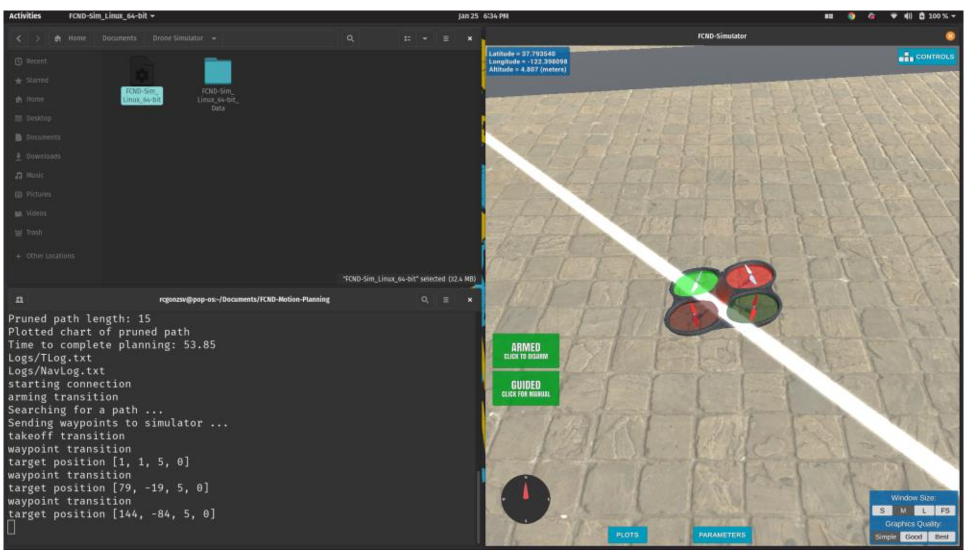
The submitted scripts (`**motion_Planning.py**` and `***plannning_utils.py***`) implements the 3D motion planning using a simple grid search, and for convenience can be executed with the `***runGrid.sh***` file or directly in python specifying (or not) the goal path to follow as arguments. Please note that if you don't specify those arguments, the drone will fly to an arbitrary point in the simulator

The drone will capable to fly in any coordinates you give as goal, within obstacles and with the safety distance, using the code as described below:

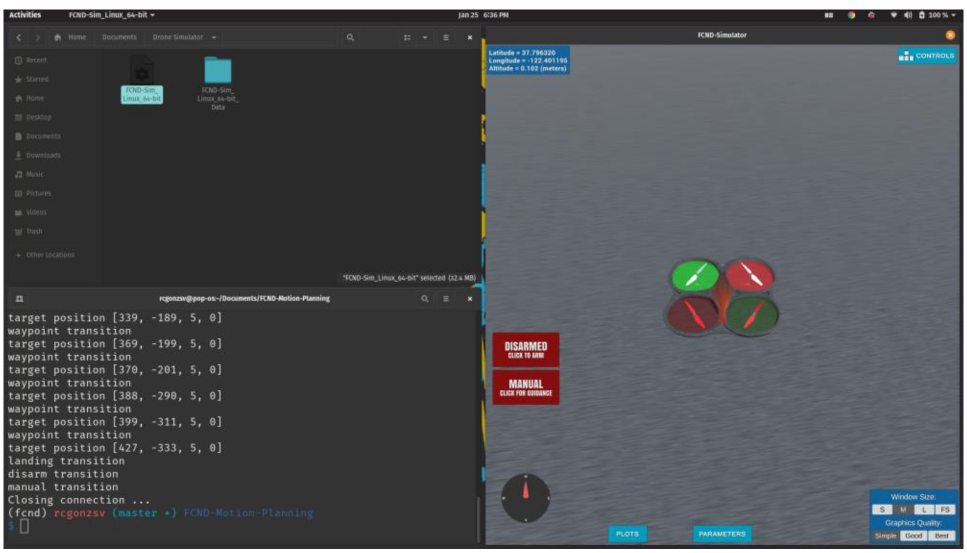
State class, containing the flight states.



Motion planning class, as main class using **Udacidrone API** as parameters, instantiating drone's connection, target position in the 3D grid, initial waypoints, initial flight state as manual and call backs registered such position, velocity and flight state. Please refer to the code lines 18 to 127 to clearly identify each of the function controlling the drone.



Once **motion planning** begins, proceeds with identify global starting location, set global home, set drone's local position (represented in **NED coordinates**) using global_to_local conversion function after reading/loading data from ***colliders.csv***. Then, a grid representation of the environment is created including obstacles data passed to the `*create_grid*` function, dorne's starting position, and the goal on the grid. As stated before, the **A* search algorithm** (lines 188 to 194 in `*motion_planning.py*`) Is used to find a path form the `*grid_start*` to the `*grid_goal*` and **Bresenham** (lines 195 to 212) ray tracing is used to prune the path in order to remove unnecessary waypoints in order to make the route more efficient and send the waypoints to the drone simulator.



Please refer to the folder `***FullPathExecution***` folder to observe complete sequencing. Please note that other drone commands implemented by **Udacidrone API** directly are not covered in the writeup, but feel free to refer to the **Udacidrone API** documentation for more information.

The file `*planning_utils.py*` file contains several helper functions for convenience, such as: `*create_grid*` which returns a 2D representation of a 2D configuration space, based on obstacle data, drone altitude and safety distance arguments; `*action class*`, represents the deltas of the valid movement actions that the drone takes on based on its current grid position (and associated cost); `*valid_actions*`, returns movements the drone makes given a grid and current node; `*a_star*`, **A* implementation** that returns a path from a given start point to a destination goal; `*heuristic*`, calculates the **Euclidean distance** between a given point and the goal, used as an approximation to guide A* search and make it mor efficient to return a path; `*point*`, returns a 3D point as numpy array; `*collinearity_check*`, returns true inf 3 points are collinear within the threshold set by Epsilon. `*prune_path*`, prunes the path from the start point to the goal which was determined by **A* search** to remove unnecessary waypoints using the `*collinearity_check*` function determining if the points are in line; `*bres_prune*`, uses **bresenham ray tracing** instead a collinearity to prune the path of unnecessary waypoints.

Please refer to the folder **`*visualization*`** to observe the grid-based plan for the motion planning.

It works!!! please run `*runGrid.sh*` or the `*motion_planning.py*` directly to test.