**Project 3D Motion Planning**

The 3D Motion Planning project was carried out with the template code given with the appropriate changes where TODO tasks have been assigned.

**1. (Comparison of Backyard\_Flyer and Motion\_Planning codes and description of the organization of starter code:**

Tested the basic **motion\_planning py** and the **backyard\_flyer\_solution.py** in the Motion Planning environment of the FCND Simulator.

A cursory inspection of both code side-by-side reveals that class **BackyardFlyer** and the **MotionPlanning** in the respective files are predominantly one and the same , except that the functions of planning the path **(plan\_path)** and sending the waypoints to the simulator **(send\_waypoints)** are added to the **Motion\_Planning class,** as a difference between the two.

There is a **geo-position data map of obstacle course given as N-E orientation** in **colliders.csv**, which will be read inside the plan\_path function to select the **grid\_start** to **grid\_goal** points to decide the drone path as it flies.

Also in the main, **instead of setting the port and host as fixed** (here they are default, if they are not given as arguments during running of the program), they can be passed as arguments.

Also **some of the functions are collected together in** a separate file **planning\_utils;** They are **exclusively used** inside the **plan\_path** function. The included functions are **a\_star, heuristics, creat\_grid**

Additional functions to prune the path computed **(prune\_path)** and to check if three points are in a line (using the determinant function to avoid travelling in a saw-tooth fashion of a grid completely **(collinearity\_check)** are added in the **planning\_utils.py.** Additionally from the **udacidrone.frame\_utils** **global\_to\_local()** function is imported to be used.

In the class\_states, apart from the main states found in the backyard\_flyer, PLANNING state is also added owing to the need of this planning.

**2. Implementation Details:**

**Criteria 1:**

In the starter code , initially the home\_position is where the drone is kept when the simulator is in its native initialized state. This is some where in the latitude, longitude specified by the colliders.csv file. (north offset of -316 and east offset of -415. This can be anywhere depending on where the flight lands after the first time. Alternatively, if we can specify the latitude and longitude to be different as an argument, it can be anywhere in the map (excluding the obstacles). For simplicity and testing, it has been retained as given.

That position has been set as global\_home using self.set\_home\_position() see the code.

self.set\_home\_position(lat0,lon0,0.00)

Assigned the values of **self.\_latitude, self.\_longitude,self.\_altitude to geodetic\_current\_position**

Assigned the values of **self.global\_home[0], self.global\_home[1], self.global\_home[2] to geodetic\_home\_coordinates**

Using the function **global\_to\_local()** with the above two **(geodetic\_current\_position, and geodetic\_home\_coordinates),** computed the NE position of local position

Changes to accommodate diagonal movement of drone, appropriate changes have been made in Action Class

NORTHEAST = (-1,1,np.sqrt(2))

NORTHWEST = (-1,-1,np.sqrt(2))

SOUTHWEST = (1,-1,np.sqrt(2))

SOUTHEAST = (1,1,np.sqrt(2))

And in valid\_action()

if x - 1 < 0 or y - 1 < 0 or grid[x - 1, y - 1] == 1:

valid\_actions.remove(Action.NORTHWEST)

if x - 1 < 0 or y + 1 > n or grid[x - 1, y + 1] == 1:

valid\_actions.remove(Action.NORTHEAST)

if x + 1 > n or y - 1 < 0 or grid[x + 1, y - 1] == 1:

valid\_actions.remove(Action.SOUTHWEST)

if x + 1 > n or y + 1 > n or grid[x + 1, y + 1] == 1:

valid\_actions.remove(Action.SOUTHEAST)

**Criteria 2:**

Now the **grid\_start (start point)** is computed with the newly computed local position value and the map center that was hardcoded

**Criteria 3:**

The goal is obtained as simple input values of North, East offsets from the current drone position (this will help in taking the drone to any arbitrary position to any other arbitrary position. For example of the path is designed to go from Point A to Point B (goal), Point B becomes the local position for any onward travel to Point C (new goal) and the Point C becomes the local position for next onward travel point (yet another new goal) and so on and so forth.

This has been done by the following lines of code. The values can be –ve or +ve depending on the direction of movement w.r.t to NE directions (North UP +ve, North Down –ve, E-right +vem E-left –ve).

N\_offset = int(input ('Enter North offset :'))

E\_offset = int(input ('Enter East offset :'))

grid\_goal = (grid\_start[0]+N\_offset, grid\_start[1]+E\_offset)

**Criteria 4:**

In the A\* code in planning\_utils(), added the diagonal motions that have the cost of sqrt(2). Tried to implement other options, but was crashing the program. After submitting, I will continue to refine the algorithms with more understanding the algorithms as well experience with Python.

**Criteria 5: (Update)**

**Original:**

Used the collinearity test in the waypoint calculation, by adding prune\_path algorithm as learned in the class. Here tried to use Bresenham algorithm, but found that the path was running into buildings.. Need to study this with more time and try other ways of the doing the same.

**Modified:**

1. Changed the prune path function to use Bresenham’s algorithm given in the package to trim the intermediate path points from the path returned by A\* algorithm. The new function is “prune\_path\_bres()”.
2. A test with 3 consecutive points at a time is done to see if the line drawn connecting the first and 3rd point is in the path of an obstacle.
3. The function starts with the first three points in the path and removes the middle point (2nd point) and the 3rd point is moved to the 2nd point position. Now the next point is marked as 3rd point and the above step 2 is continues with step 2 the nes 3 consecutive points.
4. Continue till the last point is reached, yielding only the straight line optimal waypath points at the end of this process.

**Executing the flight:**

Please see the attached flight path maps which were actually tried path with the algorithm.

**Observations:**

The drone overshoots the path ends and sometimes too close to the buildings. In some cases hitting the building and falling on the floor

Also when I tried the simple return paths for example giving -300N and -345E (NE 16, 100) from the current local position and go back to same local position where the drone started, the paths are computed differently. Though there are no obstacles, some paths are more like steps all the way to destination. I need to understand why prune\_path behaves that way. It is perhaps the collinearity check algorithm that needs checking.

**Added Artifacts:**

1. The grid maps of some of the paths traversed
2. The motion\_planning.py
3. Planning\_utils.py
4. Some screenshots of how the paths have been taken