Project: 3D Motion Planning

Rubric Points

Writeup / README

Explain the Starter Code

1. Explain the functionality of what's provided

- motion_planning.py
  Contains main class MotionPlanning derived from Drone class. The most interesting part of code the is located in the function: plan_path. Here the drone figures out its starting location and the goal position. Afterwards it calls the method `a_star` provided in planning_utils.py script which calculates the path from start to goal. The waypoints property of MotionPlanning class instance gets the result. Other functions of MotionPlanning like in BackyardPlanning projects implement the action logic in event driven way.
- planning_utils.py
  Contains functions designed for calculating path for the drone. function **create_grid** loads data from the colliders.csv and makes the grid where every cell occupied by obstacle has the value 1. There is **Action** enum representing all possible moves within the grid. **valid_actions** function returns all feasible moves for specified cell in the grid with respect to the obstacles loaded from the csv file. **a_star** function implements a star path calculating algorithm for the specified grid, heuristic function, start and goal positions. **heuristic** function that returns the distance between 2 points in space.

Implementing Your Path Planning Algorithm

1. Set your global home position

Here students should read the first line of the csv file, extract lat0 and lon0 as floating point values and use the self.set_home_position() method to set global home. Explain briefly how you accomplished this in your code.

For retrieving lat0 and lon0 values there is function GetLat0Lon0 from planning_utils.py script The function is called from the plan_path function. It opens the csv file, gets the first line and parses values. lat0 and lot0 are used for setting the global home: self.set_home_position(lon0, lat0, 0) (in plan_path function)

2. Set your current local position

Here as long as you successfully determine your local position relative to global home you'll be all set. Explain briefly how you accomplished this in your code.

Geodetic current position is calculated by getting value from global_position property of Drone class instance (global_position = self.global_position in plan_path function). The global_position is used in the next line: local_position = global_to_local(global_position, self.global_home). That is how we get current position relative to global home in NED coordinates.

3. Set grid start position from local position

This is another step in adding flexibility to the start location. As long as it works you're good to go!

Start position is set in this line: start = (int(local_position[0]-north_offset), int(local_position[1])-east_offset). Offsets are used for calculating the position relative to (0,0) point in NED coordinates. Now the start position is not hardcoded.

4. Set grid goal position from geodetic coords

This step is to add flexibility to the desired goal location. Should be able to choose any (lat, lon) within the map and have it rendered to a goal location on the grid.

Goal position is set in these lines:

goal_position = global_to_local([-122.3962, 37.7950, 0.0], self.global_home)

goal = (int(goal_position[0])-north_offset, int(goal_position[1])-east_offset)

[-122.3962, 37.7950, 0.0] is just an example of goal in geodetic coordinates. Next line calculates the goal in NED coordinates.

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

Minimal requirement here is to modify the code in planning_utils() to update the A implementation to include diagonal motions on the grid that have a cost of sqrt(2), but more creative solutions are welcome. Explain the code you used to accomplish this step.*

New items in Action enum are introduced in order to include diagonal motions in planning:

NORTHWEST = (1, -1, 1.41421356)

NORTHEAST = (1, 1, 1.41421356)

SOUTHWEST = (-1, -1, 1.41421356)

SOUTEAST = (-1, 1, 1.41421356)

Additionally, there are changes in the method valid_actions for diagonal motions.

Actual planning involves Vorony graph. It to works in this way:

For the start and the goal positions we find the nearest nodes start_g and goal_g in the graph. After that we calculate the path from start to start_g in grid (using regular a_star), path from start_g to goal_g in the graph (using a_starGraph – a star for graph) and the path from goal_g to goal. The result is the concatenation of 3 parts. If it is not possible to calculate the path in the way described above we use regular a_star for searching a path from start to goal. All that happens in the function GetPath in motion_planning.py

Additionally, there is some caching in planning process. Grid and Graph are being cached (functions GetVoronoyGraph, GetGridAndOffsets in planning_utils.py). these function use pickle module to serialize Grid and Graph data files, so next time there is no need to figure them out.

6. Cull waypoints

For this step you can use a collinearity test or ray tracing method like Bresenham. The idea is simply to prune your path of unnecessary waypoints. Explain the code you used to accomplish this step.

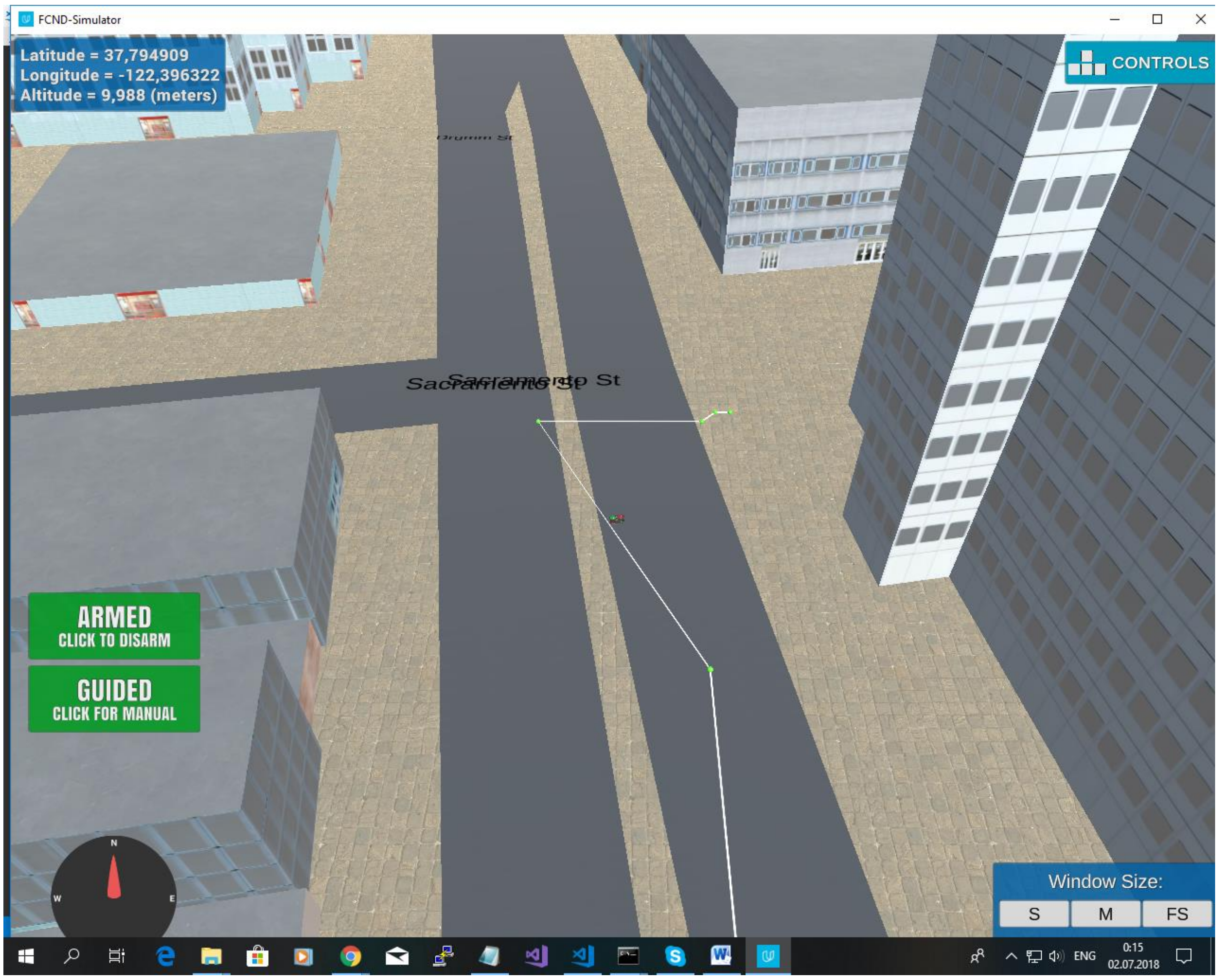
There is prune_path function in planning_utils.py that eliminates unnecessary points from the path. It uses collinearity check to determine the points to dump. That happens in function collinearity_check in planning_utils.py.

Execute the flight

1. Does it work?

It works!

That is a picture of getting to destination :



For convenience two arguments were introduced for script motion_planning.py:

--n   the N coordinate for goal

--e the E coordinate for goal

For example, to get to the point [250,720]: we can type in console like this:

```
python motion_planning.py --n 250 --e 720
```