Project: 3D Motion Planning

Explain the Starter Code

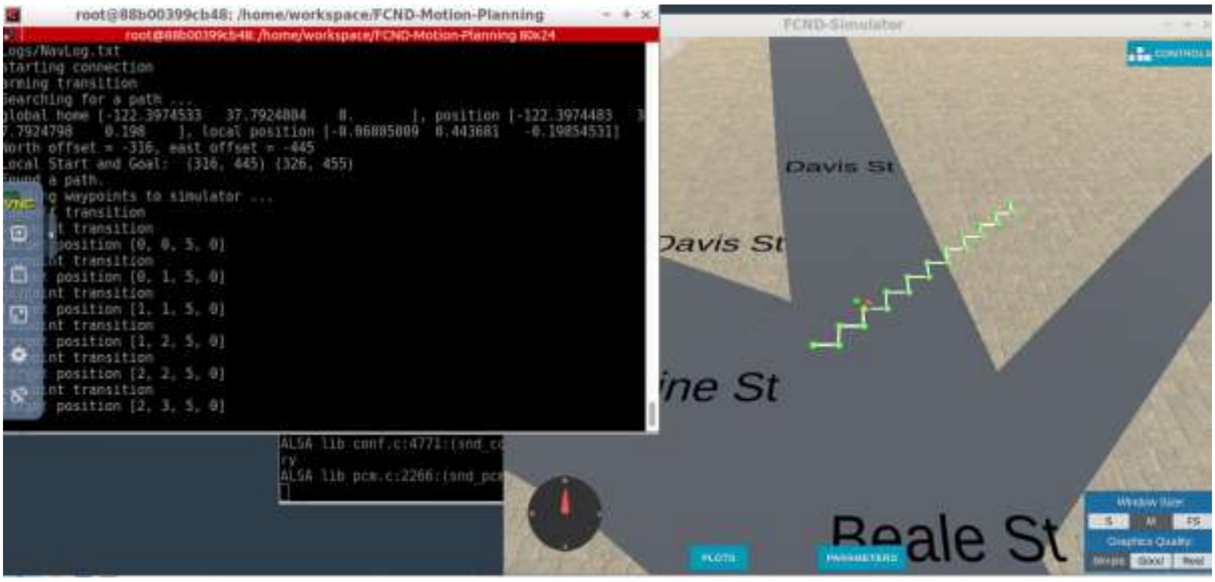

Image 1: starter code

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

These scripts contain a basic planning implementation that includes basic heurisctic, search algorithm (a* search) which are used to plan the path for drone to arm and fly 10 metres and land. ***Backyard_flyer.py*** calculates waypoint using 'calculate_box' method which defines 4 waypoints to draw a square whereas ***motion_planning.py*** uses 'plan_path' method to set start and goal point and calls a_star function from ***planning_utils.py*** to calculate waypoints.

Implementing Your Path Planning Algorithm

1. Set your global home position

I opened the ***colliders.csv*** and read the first row with code block below.

```
# TODO: read lat0, lon0 from colliders into floating point values
# Reading csv file and first row
with open('colliders.csv', newline='') as f:
    reader = csv.reader(f)
    row1 = next(reader)
```

row1 returned ***latitude*** and ***longitude*** information with "name value"pattern and as strings. In order to pass the values as arguments to ***self.set_home_position*** method I stripped the name values and converted values to ***float***.

```python
# TODO: set home position to (lat0, lon0, 0)
#retrieving value of lat0 and lon0 and setting home position to (lat0,lon0,0)
lat = row1[0].strip('lat0')
lon = row1[1].strip(' lon0')
#converting string values to float
lat0 = float(lat)
lon0 = float(lon)
print(lat0)
print(lon0)
self.set_home_position( lon0, lat0,0.0)
```

2. Set your current local position

Current global position is a tuple of 3 values
self._latitude, ***self._longitude*** and ***self._altitude.*** In the first line below we set a global_position variable to hold and initialize our current global position. Then passed the value of this variable and ***self.global_home*** to ***global_to_local()*** method to convert current global position to local position.

```python
# TODO: retrieve current global position
global_position = (self._longitude,self._latitude,self._altitude)
# TODO: convert to current local position using global_to_local()
local_position =  global_to_local(global_position,self.global_home)
```

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!

```python
# TODO: convert start position to current position rather than map center
(east_current, north_curent, _, _) = utm.from_latlon(self._latitude,self._longitude)
print("North current = {0}, east current = {1}".format(north_curent, east_current))
grid_start = (int(str(north_curent)[:3]), int(str(east_current)[:3]))
```

4. Set grid goal position from geodetic coords

I decleared two variable ***latitude_goal*** and ***longitude_goal*** and set their initial values to **37.793837 , -122.397745** float values but we can also initialize those variables with values returned by another function or gui interaction. We converted those geodetic coordinates to NED coordinates because we need NED coordinates to calculate motion planning.

```python
# TODO: adapt to set goal as latitude / longitude position and convert
latitude_goal =  37.793837
longitude_goal = -122.397745

goal_global = [ longitude_goal , latitude_goal , 0]
goal_local = global_to_local (goal_global,self.global_home)

north_goal = int(goal_local[0])
east_goal = int(goal_local[1])

grid_goal = ( ( north_goal + -north_offset )  , (east_goal + -east_offset) )
```

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

including diagonal motions on the grid that have a cost of sqrt(2) .We need to include all possible diagonal motions to our action space. I used the below code to include them with a cost of sqrt(2)

```
NORTH_WEST = (-1, -1, np.sqrt(2))
NORTH_EAST = (-1, 1, np.sqrt(2))
SOUTH_WEST = (1, -1, np.sqrt(2))
SOUTH_EAST = (1, 1, np.sqrt(2))
```

We also have to check if our diagonal movements is off the grid or colliding in an obstacle.

```
if (x - 1 < 0 or y - 1 < 0) or grid[x - 1, y - 1] == 1:
    valid_actions.remove(Action.NORTH_WEST)
if (x - 1 < 0 or y + 1 > m) or grid[x - 1, y + 1] == 1:
    valid_actions.remove(Action.NORTH_EAST)
if (x + 1 > n or y - 1 < 0) or grid[x + 1, y - 1] == 1:
    valid_actions.remove(Action.SOUTH_WEST)
if (x + 1 > n or y + 1 > m) or grid[x + 1, y + 1] == 1:
    valid_actions.remove(Action.SOUTH_EAST)
```

6. Cull waypoints

I used a ***collinearity*** test to prune unnecessary waypoints. By this way I got rid of additional waypoints on the straight line other thanstarting and ending waypoint.

```
def prune_path(path):
    pruned_path = [p for p in path]

    i = 0
    while i < len(pruned_path) - 2:
        p1 = point(pruned_path[i])
        p2 = point(pruned_path[i+1])
        p3 = point(pruned_path[i+2])


        if collinearity_check(p1, p2, p3):
            pruned_path.remove(pruned_path[i+1])
        else:
            i += 1
    return pruned_path

def point(p):
    return np.array([p[0], p[1], 1.]).reshape(1, -1)

def collinearity_check(p1, p2, p3, epsilon=1e-2):
    m = np.concatenate((p1, p2, p3), 0)
    det = np.linalg.det(m)
    return abs(det) < epsilon
```

I pruned path before converting path to waypoints and used pruned path to find my new waypoints.

```
# TODO: prune path to minimize number of waypoints
# TODO (if you're feeling ambitious): Try a different approach altogether!
pruned_path = prune_path(path)

# Convert path to waypoints
waypoints = [[p[0] + north_offset, p[1] + east_offset, TARGET_ALTITUDE, 0] for p in pruned_path]
```

Last execution output results are included in Waypoints.txt