Project Write Up: 3D-Motion Planning

Talha Khalid

August 12th, 2018

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

The plan_path() method begins by initializing the target altitude and safety distance for the vehicle. The home and current positions are then set before beginning

the planning algorithm. Next, the grid is then discretized, with start and goal states defined. Using A* an overall path plan is found and then further trimmed using some pruning method to minimize the number of waypoints. Finally, the paths are converted to waypoints which then feed into the send_waypoints() method.

The key difference between motion_planning.py and backyard_flying_solution.py is the how the waypoints are determined. For the current project, waypoints are defined based off a discretized map with points determined based off search algorithms (namely A*). The backyard_flying_solution project had waypoints that were hardcoded and not based off a dynamic grid or planned.

The planning_utils.py code serves as a basis for creating the grid, defining the overall path plan from start state to goal state via A*, assessing cost of the path and defining valid movements that the vehicle can make.

2.Setting global home position

The first line containing the home latitude and longitude was extracted from the provided csv file via readline() feature and separating the string from the values and storing the values in the variables lon0 and lat0. Using the built in set_home_position() function, the home position was set with longitude, latitude and altitude. This is all built into the read_home() function defined under planning_utils.py Altitude was set to 0 as it is assumed that the vehicle starts from a ground level.

```python
        # TODO: read lat0, lon0 from colliders into floating point values
        with open('colliders_2.csv') as f:
            unpack = f.readline().rstrip().replace('lat0','').replace('lon0 ','').split(',')
            lat0 = float(unpack[0])
            lon0 = float(unpack[1])

        #Lat0=37.79248
        #Lon0=-122.39745
        # TODO: set home position to (lon0, lat0, 0)
        self.set_home_position(lon0,lat0,0)

        # TODO: retrieve current global position
        global_position=(self._longitude,self._latitude,self._altitude)
```

Figure 1. Code snippet from motion_planning.py for setting home position

3.Setting current local position

Local position was set via the use of the global_to_local() function. However the inputs required are global_positon And global_home. global_position was set via self._longitude, self._latitude, self._altitude. global_home was set via they self.global_home attribute.

4.Setting grid start position from local position

Grid start position was set based off of the conversion from global to local position via the global_to_local() function. The outputs of the function were north starting position n_start, east starting position e_start.

```
        n_start,e_start,a_start= global_to_local(global_position, self.global_home)
```

Figure 2: code snippet from motion_planning.py for setting grid start position

5.Setting grid goal position from geodetic coordinates

Grid goal position is determined based off a random number generated in the north and east direction, within the range of north_min to north_max and east_min to east_max. An additional check was added to determine if the randomly chosen goal location was an obstacle. If so, the vehicle stops planning and lands back at the home position.

```
        grid_goal = (abs(random.randint(north_min,north_max)),abs(random.randint(east_min,east_max)))
        #grid_goal = (223,239)
        print('Local Start and Goal: ', grid_start, grid_goal)

        if grid[grid_goal[0]][grid_goal[1]] > 0:   # a collision
            print("\nUh-Oh: specified goal is an obstacle, trip canceled!")
            print("Let go back to where we started\n")
            grid_goal=grid_start
```

Figure 3: code snippet from motion_planning.py for setting goal and checking for collision

6.Modifying A* to include diagonal motion

In order to modify A* to include diagonal motion, the (x,y) coordinates (tuples) had to be set for Northeast, Northwest, Southeast and Southwest and each had to be assigned a cost of square root of 2 (distance traveled diagonally with a x and y coordinate of 1 yields square root of 2) in the valid_actions() function. After assigning the tuple, valid_actions() had to be set to ensure that the vehicle didn't attempt to travel in a direction if there was an obstacle present or if it was off the grid. The heuristic used was the Euclidian method instead of the provided Manhattan distance heuristic due to slightly faster computation time. The A* method implemented in this project was for a grid-based map.

```python
def valid_actions(grid, current_node):
    """
    Returns a list of valid actions given a grid and current node.
    """
    valid_actions = list(Action)
    n, m = grid.shape[0] - 1, grid.shape[1] - 1
    x, y = current_node

    # check if the node is off the grid or
    # it's an obstacle

    if x - 1 < 0 or grid[x - 1, y] == 1:
        valid_actions.remove(Action.NORTH)
    if x + 1 > n or grid[x + 1, y] == 1:
        valid_actions.remove(Action.SOUTH)
    if y - 1 < 0 or grid[x, y - 1] == 1:
        valid_actions.remove(Action.WEST)
    if y + 1 > m or grid[x, y + 1] == 1:
        valid_actions.remove(Action.EAST)
    if x + 1 > n or y + 1 > m or grid[x + 1, y + 1] == 1:
        valid_actions.remove(Action.SE)
    if x - 1 < 0 or y + 1 > m or grid[x - 1, y + 1] == 1:
        valid_actions.remove(Action.NE)
    if x + 1 > n or y - 1 < 0 or grid[x + 1, y - 1] == 1:
        valid_actions.remove(Action.SW)
    if x - 1 < 0 or y - 1 < 0 or grid[x - 1, y - 1] == 1:
        valid_actions.remove(Action.NW)
    return valid_actions
```

Figure 4: code snippet from planning_utils.py for encompassing diagonal movements

```python
class Action(Enum):

    """
    An action is represented by a 3 element tuple.

    The first 2 values are the delta of the action relative
    to the current grid position. The third and final value
    is the cost of performing the action.
    """

    WEST = (0, -1, 1)
    EAST = (0, 1, 1)
    NORTH = (-1, 0,1)
    SOUTH = (1, 0,1)
    SE = (1, 1, math.sqrt(2))
    NE = (-1, 1, math.sqrt(2))
    SW = (1, -1, math.sqrt(2))
    NW = (-1, -1, math.sqrt(2))
```

Figure 5: code snippet from planning_utils.py for defining diagonal movements as well as cost of movement

7.Cull waypoints

Cutting down waypoints and "pruning" the path was achieved by using the collinearity test between 3 consecutive points.

If the points were collinear (as defined in the collinearity() function that tested for a matrix determinant less than or equal to 1e-6), the middle point(s) was removed from the path to shorten the waypoints. The final path was returned and sent as waypoints to the vehicle.

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

def collinearity(p1, p2, p3):
        m = np.concatenate((p1, p2, p3), 0)
        det = np.linalg.det(m)
        return abs(det) < 1e-6

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(p1,p2,p3):
            pruned_path.remove(pruned_path[i+1])
        else:
            i+=1
    print ("This is pruned path:", pruned_path)
    return pruned_path
```

Figure 6: code snippet from planning_utils.py for collinearity and pruning the path

8.Does it work?

For short distances, path is computed quickly and follows through the plan entirely from initial takeoff to landing but for long distances the simulator times out due to the long computation time. In the case the path is through an obstacle, it will fail to avoid the obstacle and attempt to go right through it. The code was unsuccessful in those scenarios.

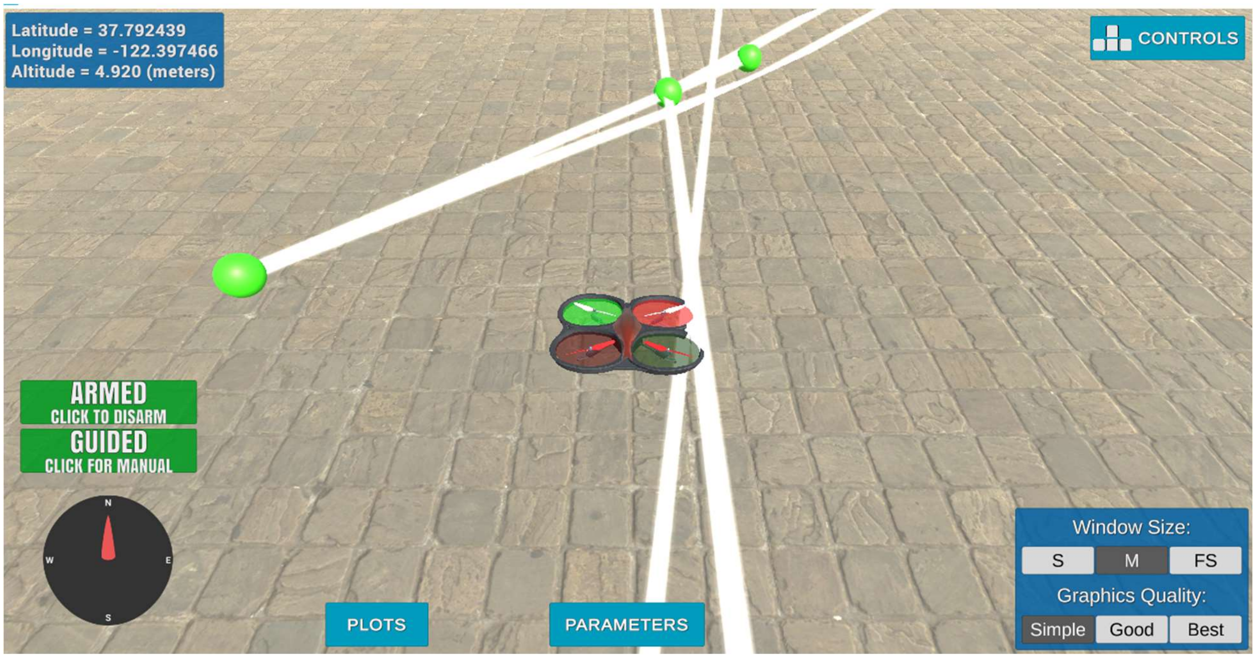

Figure 3: Screenshot of vehicle navigating through waypoints successfully

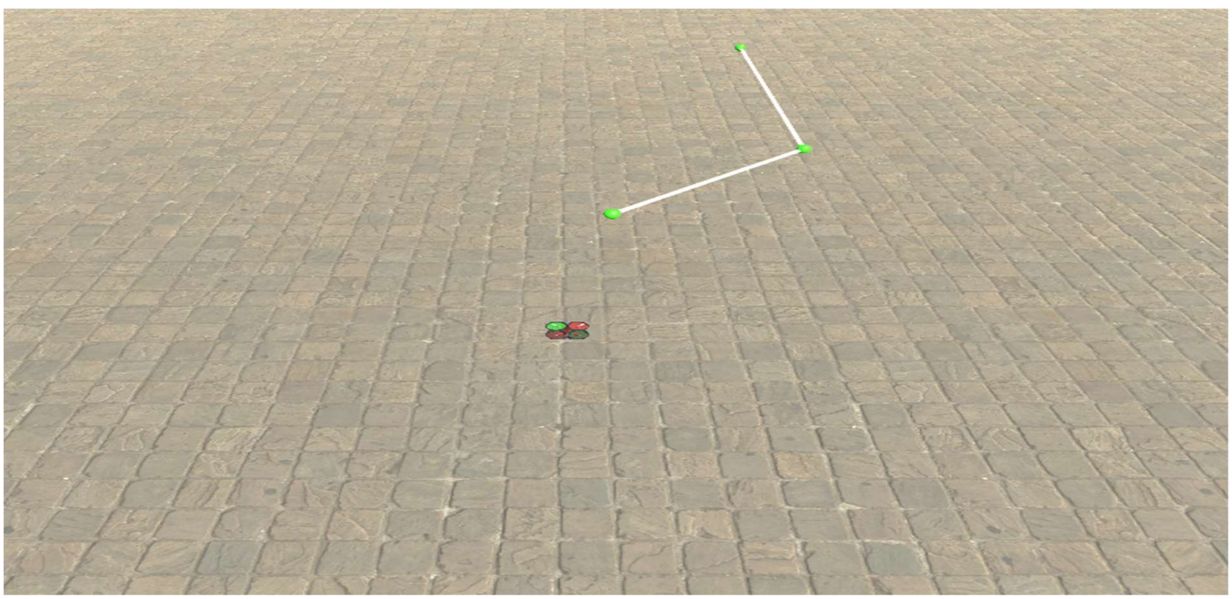

Figure 4: Screenshot of vehicle navigating in an open environment.