3D MOTION PLANNING PROJECT

STARTER CODE EXPLANATION: BACKYARD-FLYER-SOLUTION.PY

In *backyard_flyer_solution.py* upon arming the drone (*self.armed = True*), we transition to takeoff state (*self.flight_state = States.TAKEOFF*). Next, once the target altitude is reached, the waypoints to follow are determined using the method *calculate_box()* and we transition to the state *States.WAYPOINT*. The drone then proceeds to navigate its way to each waypoint until there are none left. When all waypoints have been visited the drone initiates the landing sequence by transitioning to *States.LANDING*.

STARTER CODE EXPLANATION: MOTION-PLANNING.PY

In *motion_planning.py* upon arming the drone (*self.armed = True*), we first plan a path from a starting waypoint to a goal waypoint using the method *plan_path()*. In the provided code, this accomplished as follows:

- **Grid Generation:** An occupancy grid of the word is generated using the method *create_grid* from *planning_utils.py*. This method generates the occupancy grid using 2.5D map data, a safety distance, and the drone's target altitude.

- **Start Determination:** The drone's starting location is then defined in terms of the generated grid. In the provided code, this starting point is taken to be the center of the grid. Later on in my solution, I show how an arbitrary lat,lon pair is converted to a grid cell.

- **End Determination:** The drone's goal location is then also defined in terms of the generated grid. Like the starting point, the provided code sets the end point to be 10 grid cells to the north and east of the starting point.

- **Path Determination:** A path through the grid is then computed using the A* algorithm (method *a_star* in *planning_utils.py*). In the provided coded the only available moves through the grid involve moving north, south, east, or west with each move having a cost of 1. This is why the returned path is a zig-zag. Later in my solution, diagonal moves are added (ie. north-east, north-west, south-east, and south-west).

- **Grid to Way Points:** Finally, the grid cell coordinates making up the path are converted to waypoint coordinates in the drone's local coordinate frame.

Once planning is complete, we proceed to takeoff (*self.flight_state = States.TAKEOFF*). When the drone reaches the target altitude it begins following the determined waypoints (*self.flight_state = States.WAYPOINT*). When all waypoints have been visited the drone initiates the landing sequence by transitioning to States.LANDING.

PATH PLANNING IMPLEMENTATION: SETTING HOME POSITION

The drone's global home position is set to be the lat,lon pair found in the first line of collider.csv along with an altitude of 0 as shown below (lines 162-165 of motion_planning.py).

```
1.  # TODO: read lat0, lon0 from colliders into floating point values
2.  lat0, lon0 = self.getHomePosition()
3.  # TODO: set home position to (lat0, lon0, 0)
4.  self.set_home_position(lon0, lat0, 0)
```

The lat,lon pair is read using the method getHomePosition (shown below; line 114 of motion_planning.py).

```
1.  def getHomePosition(self):
2.    with open('colliders.csv', 'r') as fh:
3.      position = fh.readline()
4.      lat_str, lon_str = position.split(',')
5.      lat = lat_str.split(' ')[-1]
6.      lon = lon_str.split(' ')[-1]
7.      return float(lat), float(lon)
```

PATH PLANNING IMPLEMENTATION: START AND GOAL COORDINATES AND GRIDS

The drone's local position is derived from its global position using method global_to_local as shown below (line 168 of motion_planning.py):

```
1.  local_position = global_to_local(self.global_position, self.global_home)
```

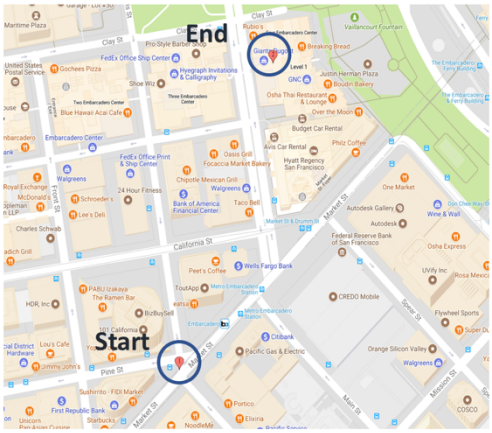
Furthermore, the local position is then converted to grid cell coordinates in the world's grid map as shown below (line 180 of motion_planning.py)

```
1.  grid_start = (int(local_position[0]) - north_offset, int(local_position[1]) - east_offset)
```

Similarly, the grid cell coordinates of the goal are also determined as show below (lines 184-185 in motion_planning.py)

```
1.  local_goal = global_to_local((-122.396332, 37.795121, TARGET_ALTITUDE), self.global_home)
2.  grid_goal = (int(local_goal[0]) - north_offset, int(local_goal[1]) - east_offset)
```

The map to the right shows the start and goal coordinates for the drone. Note that in the simulator the provided goal coordinates terminate on the road (as shown in a later image), but when visualized in Google Maps the goal coordinates fall on a building.



A* SEARCH ALGORITHM

The A* algorithm implementation in planning_utils.py was modified to allow the drone to take diagonal moves. This was accomplished by defining those moves and their cost as well as modifying the method valid_actions() to check when those moves are allowed. Below, one can see how the new moves (NORTH_EAST, NORTH_WEST, SOUTH_EAST, and SOUTH_WEST) and their costs were defined and how valid actions are determined.

```
1.  class Action(Enum):
2.      """
```

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

    SOUTH = (1, 0, 1)
    SOUTH_EAST = (1, 1, np.sqrt(2))
    SOUTH_WEST = (1, -1, np.sqrt(2))

    @property
    def cost(self):
        return self.value[2]

    @property
    def delta(self):
        return (self.value[0], self.value[1])
```

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

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

    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)

    return valid_actions
```

PATH PRUNING

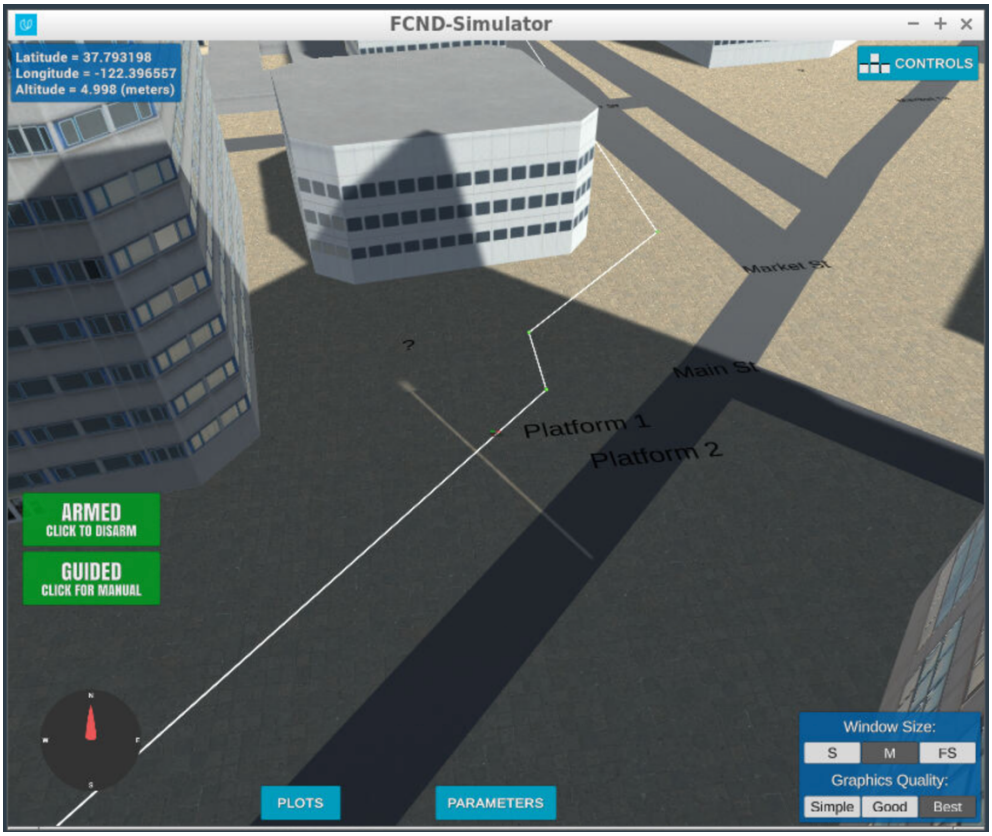
The path returned by the A* algorithm is pruned using a collinearity check. More specifically, points along the path are examined in triplets. Three points on the path are determined to be collinear if the matrix formed by those points has a determinant near zero. In that case the middle point is dropped, and the terminal end points are retained. This process is repeated until all points along the path have been examined as shown in the code below (lines 133-151 of motion_planning.py.

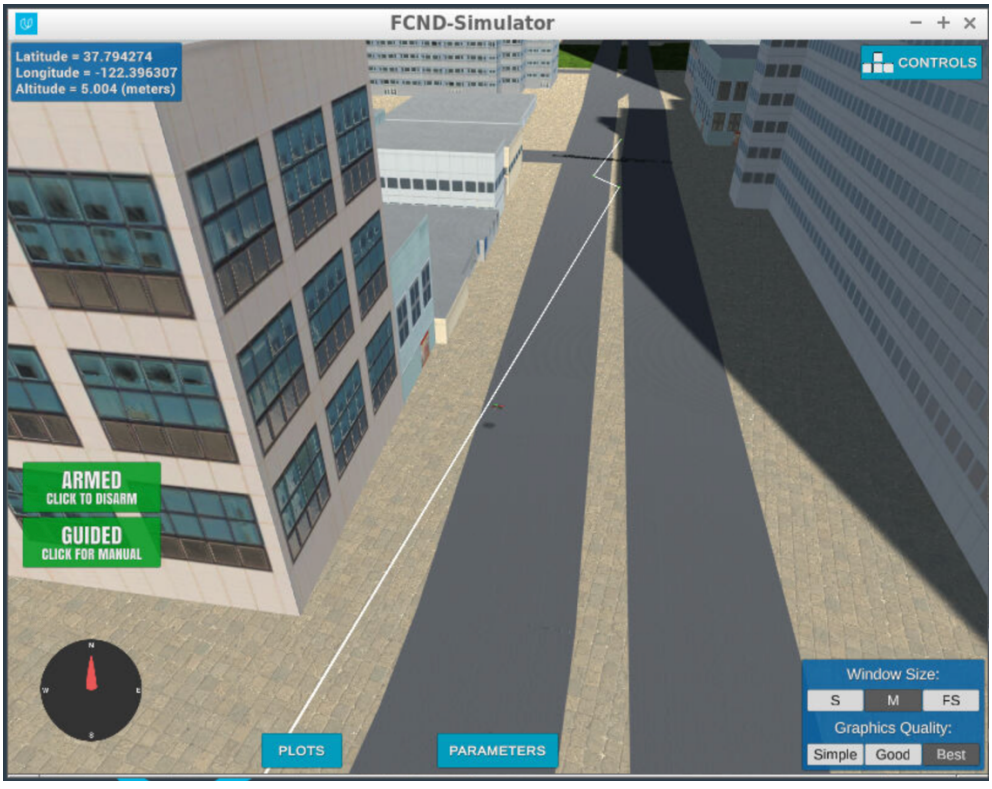
```
1.  def prune_path(self, path):
2.    """Loops over path points and removes middle points of a line segment."""
3.    pruned_path = []
4.    pruned_path.append(path[0])
5.    idx=0
6.    while idx + 2 < len(path):
7.      if self.collinearity_check(self.point(path[idx]),
8.                                 self.point(path[idx+1]),
9.                                 self.point(path[idx+2])):
10.       pruned_path.append(path[idx+2])
11.     else:
12.       pruned_path.extend([path[idx+1], path[idx+2]])
13.     idx += 2
14.
15.   if idx + 2 == len(path)-1:
16.     return pruned_path
17.   else:
18.     pruned_path.extend(path[idx+1:])
19.     return pruned_path
```

When applied to the path determined by A* for the start and goal location shown in the Google Maps image above, the original path length was 294 and the pruned path length is 7.

RESULTS

The images below show the drone traveling along different segments of the pruned path from the start location to the goal location illustrated in the Google Maps image above.


FCND-Simulator
Latitude = 37.793198
Longitude = -122.396557
Altitude = 4.998 (meters)
CONTROLS
Market St
Main St
Platform 1
Platform 2
ARMED
CLICK TO DISARM
GUIDED
CLICK FOR MANUAL
Window Size:
S
M
FS
Graphics Quality:
Simple
Good
Best
PLOTS
PARAMETERS


FCND-Simulator
Latitude = 37.794274
Longitude = -122.396307
Altitude = 5.004 (meters)
CONTROLS
ARMED
CLICK TO DISARM
GUIDED
CLICK FOR MANUAL
Window Size:
S
M
FS
Graphics Quality:
Simple
Good
Best
PLOTS
PARAMETERS


FCND-Simulator
Latitude = 37.795112
Longitude = -122.396372
Altitude = 0.008 (meters)
CONTROLS
DISARMED
CLICK TO ARM
MANUAL
CLICK FOR GUIDANCE
Drumm St
PLOTS
PARAMETERS
Window Size:
S
M
FS
Graphics Quality:
Simple
Good
Best