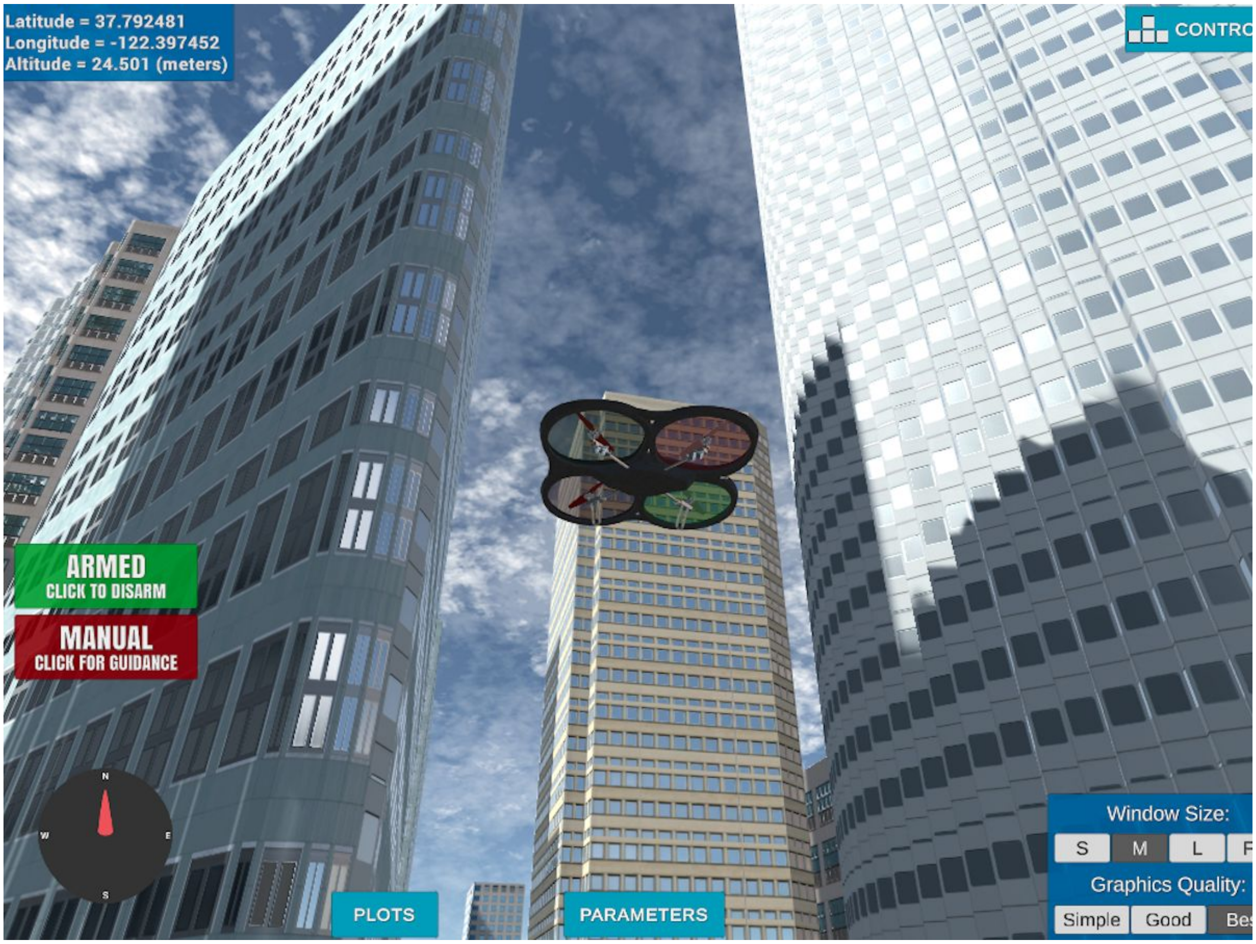
Project: 3D Motion Planning

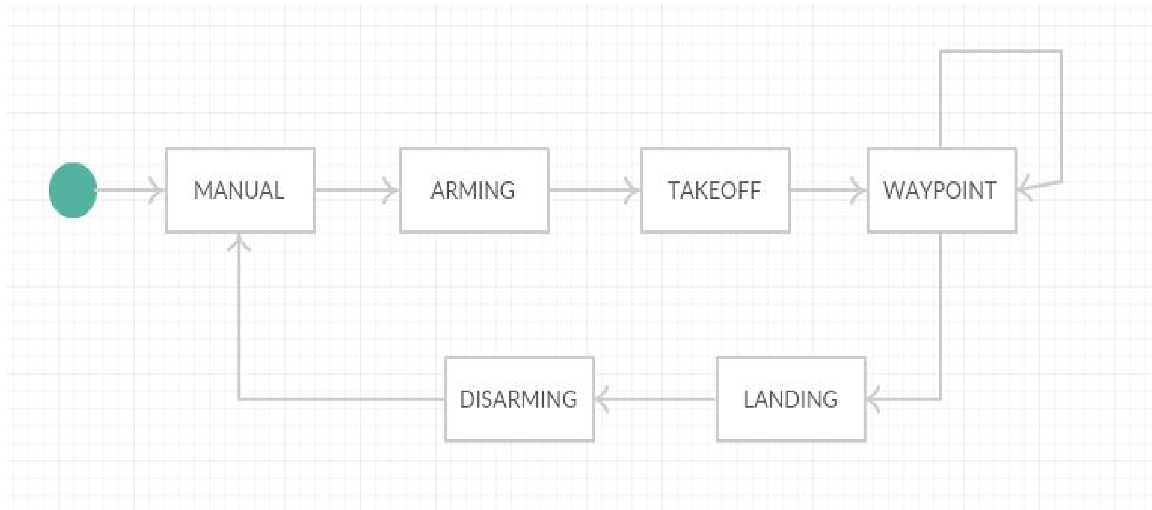


Below I describe how I addressed each rubric point and where in my code each point is handled.

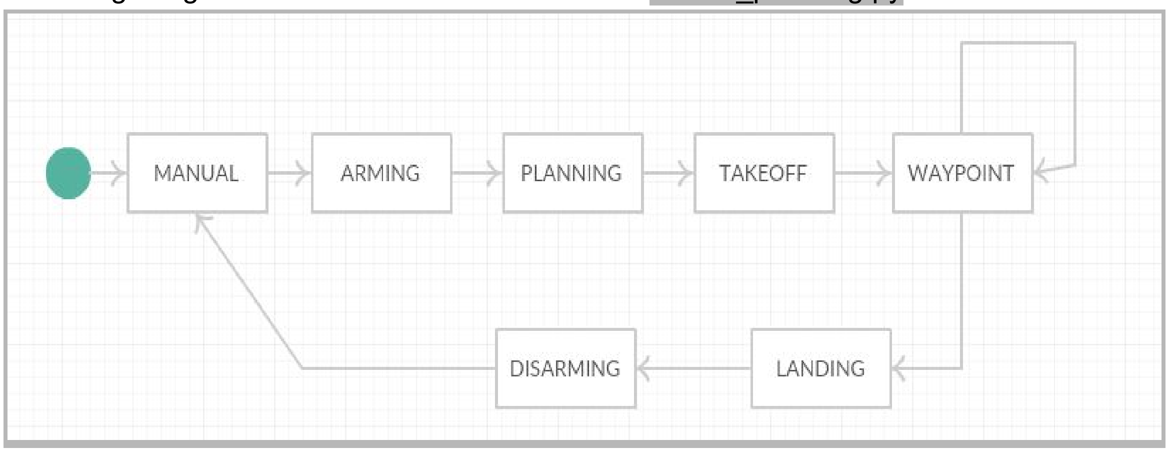
1. Explanation about the functionality of what's provided in `motion_planning.py` and `planning_utils.py`

These implementations contains a basic implementation of motion planning. Main file is motion_planning.py which includes planning_utils.py.

1. motion_planning.py is similar to backyard_flyer.py but includes a planning state.
    a. Following image shows the state machines in the backyard_flyer.py.



b. Following image shows the state machines in the motion_planning.py



2. planning_utils.py contains a few functions and a class which will be used in the planning phase of the flight. These functions are:
    a. create_grid(data, drone_altitude, safety_distance) is the function which returns a grid representation of a 2D configuration space based on given obstacle data, drone altitude and safety distance arguments.
    b. Action class is there which defines the action states.
    c. valid_actions(grid, current_node) returns the list of valid actions which can be taken from a particular cell in the grid.
    d. a_star(grid, h, start, goal) function given the grid, heuristic, start and goal state ,runs the a_star search algorithm to find a path from start to goal location.
    e. heuristic(position, goal_position) defines one possible heuristic to be used in the a_star algorithm.

2. Setting global home position

Here I read the first line of the csv file, extracted lat0 and lon0 as floating point values and used the self.set_home_position() method to set global home.

```
        filename = 'colliders.csv'
        # Reading in the data skipping the first two lines.

        # reading lat0, lon0 from colliders into floating point values
        f = open(filename, "r")
        temp = f.read().split('\n')
        lat,lon = temp[0].split(",")
        lat0 = float(lat.strip('lat0'))
        lon0 = float(lon.strip(' lon0 '))

        f.close()

        print(lat0,lon0)

        # setting home position to (lon0, lat0, 0)
        self.set_home_position(lon0, lat0, 0.0)

```

3. Setting 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.

```
        # retrieving current global position
        global_position = (self._longitude,self._latitude,self._altitude)

        # converting to current local position using global_to_local()
        local_position = global_to_local(global_position, self.global_home)

```

4. Setting grid start position from local position

This is another step in adding flexibility to the start location.

```
        # Reading in obstacle map
        data = np.loadtxt('colliders.csv', delimiter=',', dtype='Float64', skiprows=2)

        # Defining a grid for a particular altitude and safety margin around obstacles
        grid, north_offset, east_offset = create_grid(data, TARGET_ALTITUDE, SAFETY_DISTANCE)
        print("North offset = {0}, east offset = {1}".format(north_offset, east_offset))
        # converting start position to current position rather than map center
        grid_start = (-north_offset + int(self.local_position[0]), -east_offset + int(self.local_position[1]))
```

5. Setting grid goal position from geodetic coords

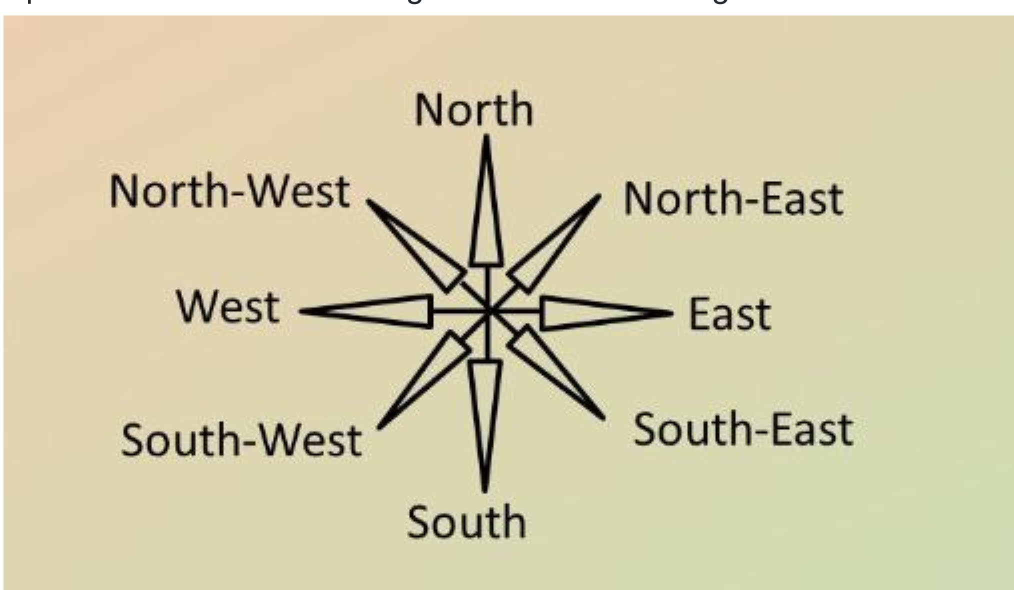
This step is to add flexibility to the desired goal location. Should be able to choose any (lat, lon) within the map and have it rendered to a goal location on the grid.

```
        # Setting goal as some arbitrary position on the grid
        # adapt to set goal as latitude / longitude position and convert
        goal_local = global_to_local ([-122.395914,37.795267,0],self.global_home)
        #goal_local = global_to_local ([-122.398993,37.792522,0],self.global_home)

        grid_goal = ((-north_offset + int(goal_local[0])),(-east_offset + int(goal_local[1])))
```

6. Modifying A* to include diagonal motion (or replace A* altogether)

1. Here, I used two different approaches, firstly using A* search based on map divided in a grid and defined action space based on that grid. I modified the code in planning_utils() to update the A* implementation to include diagonal motions on the grid that have a cost of sqrt(2)=1.414 .



```
    WEST = (0, -1, 1)
    EAST = (0, 1, 1)
    NORTH = (-1, 0, 1)
    SOUTH = (1, 0, 1)
    NORTH_EAST = (-1,1,1.414)
    NORTH_WEST = (-1,-1,1.414)
    SOUTH_EAST = (1,1,1.414)
    SOUTH_WEST = (1,-1,1.414)
```

Here, we have to check which actions are possible in a particular grid cell to stay within the map.

```
    # 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 < 0 or  y+1 > m or grid[x - 1, y + 1] == 1:
        valid_actions.remove(A
    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  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)
```

2. Second approach is a vornoi graph based approach which is implemented in motion_planning_advance.py and planning_utils_advance.py. It involved creating a grid representation of a 2D configuration space along with Voronoi graph edges given obstacle data and the drone's altitude.

Planning_utils_advance.py contains the code for this approach.

```python
    graph = Voronoi(points)

    # check each edge from graph.ridge_vertices for collision
    edges = []
    for v in graph.ridge_vertices:
        p1 = graph.vertices[v[0]]
        p2 = graph.vertices[v[1]]
        cells = list(bresenham(int(p1[0]), int(p1[1]), int(p2[0]), int(p2[1])))
        hit = False

        for c in cells:
            # First check if we're off the map
            if np.amin(c) < 0 or c[0] >= grid.shape[0] or c[1] >= grid.shape[1]:
                hit = True
                break
            # Next check if we're in collision
            if grid[c[0], c[1]] == 1:
                hit = True
                break

        # If the edge does not hit on obstacle
        # add it to the list
        if not hit:
            # array to tuple for future graph creation step)
            p1 = (p1[0], p1[1])
            p2 = (p2[0], p2[1])
            edges.append((p1, p2))

    return grid, edges, int(north_min), int(east_min)
```

6. Culling waypoints

1.For the grid based approach in planning_utils.py, I used a collinearity test. The idea is simply to prune your path of unnecessary waypoints.

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

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

def prune_path(path):
    pruned_path = [p for p in path]
    # pruning the 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 the 3 points are in a line remove
        # the 2nd point.
        # The 3rd point now becomes and 2nd point
        # and the check is redone with a new third point
        # on the next iteration.
        if collinearity_check(p1, p2, p3):
            # Something subtle here but we can mutate
            # `pruned_path` freely because the length
            # of the list is check on every iteration.
            pruned_path.remove(pruned_path[i+1])
        else:
            i += 1
    return pruned_path
```

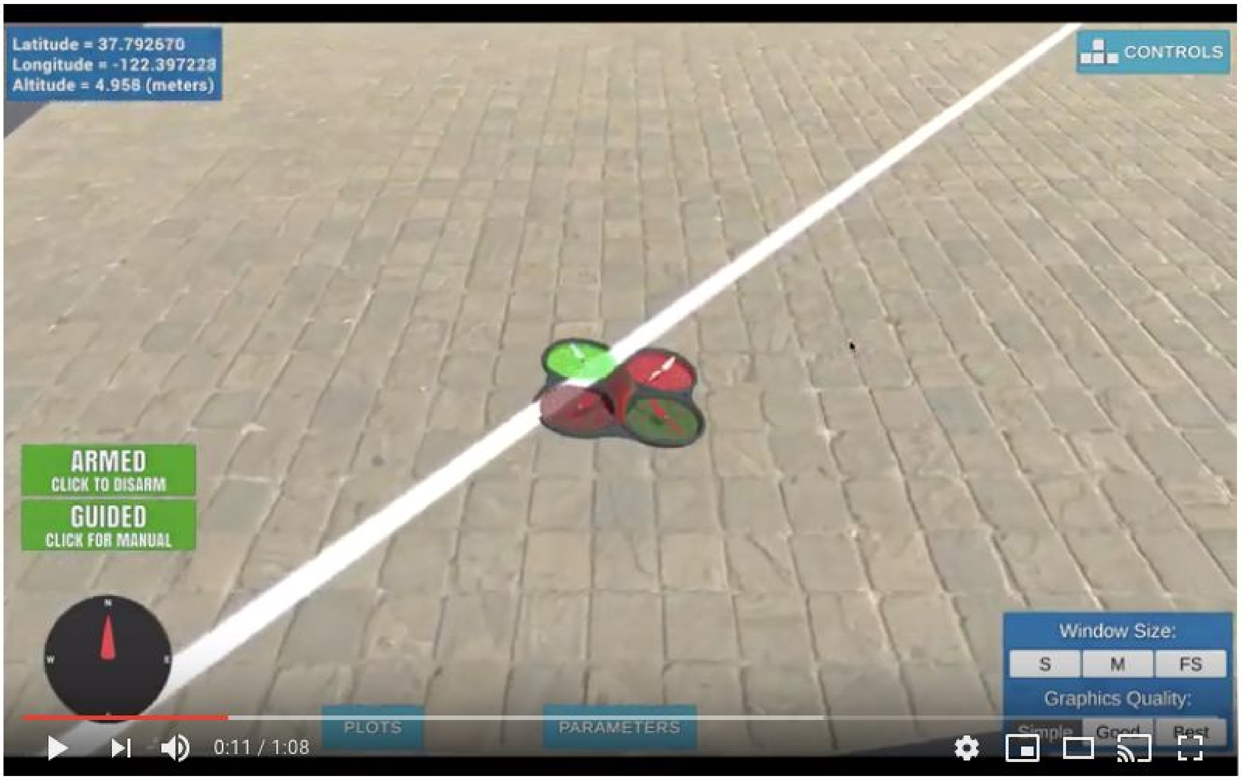
2. For the vornoi graph based approach in planning_utils_advance.py, I used the Bresenham module to trim unneeded waypoints from path.

```python
def prune_path(grid,path):
    """
    Use the Bresenham module to trim uneeded waypoints from path
    """
    pruned_path = [p for p in path]
    i = 0
    while i < len(pruned_path) - 2:
        p1 = pruned_path[i]
        p2 = pruned_path[i + 1]
        p3 = pruned_path[i + 2]
        # if the line between p1 and p2 doesn't hit an obstacle
        # remove the 2nd point.
        # The 3rd point now becomes the 2nd point
        # and the check is redone with a new third point
        # on the next iteration.
        if  all((grid[pp] == 0) for pp in bresenham(int(p1[0]), int(p1[1]), int(p3[0]), int(p3[1]))):
            # Something subtle here but we can mutate
            # `pruned_path` freely because the length
            # of the list is checked on every iteration.
            pruned_path.remove(p2)

        else:
            i += 1
    return pruned_path
```

Executing the flight

Using the grid based approach, here we can see the output of our 3d Motion Planning.
https://youtu.be/XsBuXVgsRjk



3DMotionPlanningGrid

Using the vornoi graph based approach, here we can see the output of our 3d Motion Planning. https://youtu.be/yVZNJJu-71M?t=80

3DMotionPlanningGraph

Code can be run using the following command.

python motion_planning.py **or** **python motion_planning_advance.py**