Criteria	Meet Specification
Explain the Starter Code	
Test that `motion_planning.py` is a modified version of `backyard_flyer_solution.py` for simple path planning. Verify that both scripts work. Then, compare them side by side and describe in words how each of the modifications implemented in `motion_planning.py` is functioning.	The goal here is to understand the starter code. We've provided you with a functional yet super basic path planning implementation and in this step, your task is to explain how it works! Have a look at the code, particularly in the `plan_path()` method and functions provided in `planning_utils.py` and describe what's going on there. This need not be a lengthy essay, just a concise description of the functionality of the starter code. *[* *1) The create_grid() `planning_utils.py` and in the function loads the data from colliders.csv file and calculate obstacles .* *2) In the plan_path() function in motion_planning.py file North offset = -316, east offset = -445 calculated and set as the start the grid coordinates .* *3) In the Goal is hardcoded to set (326, 455) by increasing start Northing and Easting by 10.* *4) The A-star algorithm finds the path from Start to Goal.* *5) The path put in on waypoints and moves zigzag to the goal* *6) Since no Diagonal motions and Smoothing not performed it is moving in zigzag.* *]*
Implementing Your Path Planning Algorithm	
In the starter code, we assume that the home position is where the drone first initializes, but in reality you need to be able to start planning from anywhere. Modify your code to read the global home location from the first line of the `colliders.csv` file and set that position as global home (`self.set_home_position()`)	Here you 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. [*In the motion_planning.py line no 193 - 200 added following code to achieve* `first_line = next(open("colliders.csv"))` `gps = first_line.split(',')` `lat0 = np.float64(gps[0].lstrip().split(' ')[1])` `lon0 = np.float64(gps[1].lstrip().split(' ')[1])` `#print("The Lat and Long = ", lat0, lon0)`

	`# TODO: set home position to (lat0, lon0, 0)` `self.set_home_position(lon0, lat0, 0)`]
In the starter code, we assume the drone takes off from map center, but you'll need to be able to takeoff from anywhere. Retrieve your current position in geodetic coordinates from `self._latitude`, `self._longitude` and `self._altitude`. Then use the utility function `global_to_local()` to convert to local position (using `self.global_home` as well, which you just set)	Here as long as you successfully determine your local position relative to global home you'll be all set. *[* ***The settings of home position reflected in the global_postion that takes the lat,lon &alttitude and picks the current position*** ***In the motion planning code line no : 241 added the following to get current position.*** `current_local_pos = global_to_local(self.global_position, self.global_home)` *]*
In the starter code, the `start` point for planning is hardcoded as map center. Change this to be your current local position.	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 the current posstion relative to the north_offset and eastoffset by adding to get start position in the Grid `grid_start = (-north_offset + int(current_local_pos[0]), -east_offset + int(current_local_pos[1]))` *]*
In the starter code, the goal position is hardcoded as some location 10 m north and 10 m east of map center. Modify this to be set as some arbitrary position on the grid given any geodetic coordinates (latitude, longitude)	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. *[* ***I have written an overloaded function in the motion_planning.py line no 168 it take 4 arguments And return the Grid goal location by applying the offset.*** `This is helper functiom to get arbitrary goal location by taking the lat,lon and northing and easting offset` `def global_to_local(self,lat,lon,north_offset, east_offset):`

	``grid_goal = self.global_to_local(sys.argv[2], sys.argv[4], north_offset, east_offset)```]
Write your search algorithm. Minimum requirement here is to add diagonal motions to the A* implementation provided, and assign them a cost of sqrt(2). However, you're encouraged to get creative and try other methods from the lessons and beyond!	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. In your writeup, explain the code you used to accomplish this step. [***I've Added more Enum in the Class Action*** ***In the planning_utils () line no 143 – 146*** ```# Diagonal movements add standard cost of 2 . But later will do sqrt(2) for diagnol movement NORTH_WEST = (-1, -1, 2) NORTH_EAST = (-1, 1, 2) SOUTH_WEST = (1, -1, 2) SOUTH_EAST = (1, 1, 2)``` ***And added remove conditions in the valid_actions function in the planning_utils() function*** ```try: if x - 1 < 0 and y - 1 < 0 or grid[x - 1, y - 1] == 1: # print("Removing Nort west Grid ") valid_actions.remove(Action.NORTH_WEST) except IndexError: valid_actions.remove(Action.NORTH_WEST)``` ***Adding exception handling if the range goes out of bound . avoid breaking the code.*** ***In the planning-utils() functions line no: 267 to 270 added the following to handle cost for diagonal movement ..Note below*** ```diagonal_cost = np.sqrt(2)``` ```if a.cost == 2: new_cost = current_cost + diagonal_cost + h(next_node, goal) else: new_cost = current_cost + a.cost + h(next_node, goal)```]

Cull waypoints from the path you determine using search.	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. In your writeup, explain the code you used to accomplish this step. [***In the motion_planning.py Line 137 to 165 declared the following functions to prune the path and called in the line no : 255. And then given to waypoints to execute the path.*** `pruned_path = self.prune_path(path)` ``` def point(self,p): return np.array([p[0], p[1], 1.]).reshape(1, -1) def collinearity_check(self, p1, p2, p3, epsilon=0.9e-1): m = np.concatenate((p1, p2, p3), 0) det = np.linalg.det(m) return abs(det) < epsilon # We're using collinearity here, but you could use Bresenham as well! def prune_path(self, path): pruned_path = [p for p in path] # TODO: prune the path! i = 0 while i < len(pruned_path) - 2: #print("current path********* ",pruned_path[i]) p1 = self.point(pruned_path[i]) p2 = self.point(pruned_path[i + 1]) p3 = self.point(pruned_path[i + 2]) if self.collinearity_check(p1, p2, p3): #print("pruned_path ", pruned_path[i+1]) pruned_path.remove(pruned_path[i + 1]) else: i += 1 return pruned_path ``` *]*