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

motion_planning.py holds the main code used to drive and control and the drone. The programming technique used here works on event based paradigm. It consists of “Arming”, “TakeOff”, “Landing” etc as events, The program takes care the sequence of these events so that the itis still responsive if ,while it's moving, there's an unexpected obstacle. Also, the program is responsible forcalling the planning function so that it can find a path between a starting and a goal location. n particular, it provides a function create_grid to create a numpy 2D grid where cells with value 1 are considered as "obstacles". In addition, an implementation of a grid-based A* algorithm, a_star, is also provided, together with helper functions needed for it, like the heuristic function and an Action class defining the possible actions to take while moving within the grid. This code is the same as the one studied in the lectures.

A* is an algorithm that searches for a path(or paths) in a search space that has the minimal cost. This is done by continuously visiting nearby nodes but the ones that are closer to the goal given an estimate(heuristic). This information is passed to the A* planner which outputs a list of waypoints from start to goal, if the path was found. Next, the path is cleaned by removing redundant intermediate waypoints by some algorithm, like a colinearity check or the Bresenham algorithm. Finally, the waypoints are sent to the simulator to execute the trajectory and for visualization.

the drone is given a start position (it's current position), and a goal position, 10 meters forward in X and Y directions. This information is then passed to the A* planner to find a trajectory in the grid. Since initially the A* planner only has 4 actions, and the drone has to go diagonally, the result is a zig-zag trajectory. The A* algorithm works on the 3D grid of a map and tries to find connections of a point onthe grid to the next until the end goal. Without the modification I have added to the valid_actions()the path that the algorithm finds contains many zig-zag (diagonal) movements. It connects two pointsthrough a third by forming a triangle

Implementing the Path Planning Algorithm

→ Global Home Position Setup

The global home position is read from the first line of the CSV file:

lat0 37.792480, lon0 -122.397450

This was undertaken using string string.split techniques. Basically the first line is read, then we split into 2 tokens separated by comma, and then we split each token again separating by space. Finally, we take the second token from this separation and cast to a float value.

→ Current Local Position Setup

I set the local position relative to the global home position using the following line:current_local_pos = global_to_local(self.global_position,self.global_home)I have previously set the home position in the line:self.set_home_position(lon0, lat0, 0)The lan0 and lat0 where retrieved from the .csv file.

Finally, we obtain the local_position by using the global_to_local function provided in the udacidrone

→ Grid Start Position from local position setup

The start position is set to the local position using the following code:

```
# Convert start position to current position rather than map center
grid_start = (int(local_position[0]) - north_offset,
              int(local_position[1]) - east_offset)
```

Simply, the local_position (NED) is cast to int and transformed into the local grid coordinate system, by removing the north_offset and east_offset variables which are the minimum north and east coordinates of the grid. In other words, a local position of (north_offset, east_offset) would be equivalent to position (0,0) in the grid.

→ Grid Goal position from Geodetic Coordinates Setup

GPS position of the goal is setup under global_goal which is converted to NED Coordinate system.

Finally, we convert to grid coordinates following the same procedure as with the local_position, in previous rubric point #3:

```
# Convert to grid coordinates
grid_goal = (int(local_goal[0]) - north_offset,
             int(local_goal[1]) – east_offset)
```

→ Modifying A* to include diagonal motion

The A* algorithm is updated as follows to include diagonal motion.

First, four new actions are added to the Action enum, in planning_utils.py:

```
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))
```

Notice there is always non-zero motion in both the north and east directions. The cost of these actions is sqrt(2) instead of 1.

Finally, we also need to update the valid_actions function in order to check if these actions are valid or not.

The Valid actions are tested using :

if (x - 1 < 0 and y - 1 < 0) or grid[x - 1, y - 1] == 1: valid_actions.remove(Action.NORTHWEST) if (x + 1 > n and y - 1 < 0) or grid[x + 1, y - 1] == 1: valid_actions.remove(Action.SOUTHWEST) if (x - 1 < 0 and y + 1 > m) or grid[x - 1, y + 1] == 1: valid_actions.remove(Action.NORTHEAST) if (x + 1 > n and y + 1 > m) or grid[x + 1, y + 1] == 1: valid_actions.remove(Action.SOUTHEAST)

→ Cull Waypoints

The last step is to prune the path to remove unnecessary points to have a smoother flight. This is implemented in the prune_path function, in planning_utils.py.

Basically, we loop over the original path and take 3 points at a time, p1, p2 and p3. If it's possible to go on a straight line between p1 and p3 without crossing any obstacle, it means p2 is unnecessary, and thus we remove it from the original path.

check if there's a collision-free trajectory between two points, we implement the function _is_collision_free:

```
def _is_collision_free(p1, p2, grid):
    # Compute cells covered by the line p1-p2 using the Bresenham algorithm
    covered_cells = list(bresenham(p1[0], p1[1], p2[0], p2[1]))

    # Check if any of the cells is an obstacle cell
    for cell in covered_cells:
        if grid[cell[0], cell[1]]:
            return False
    return True
```

Here we use the Bresenham algorithm (bresenham Python package) to extract a list of cells that are covered by the line between p1 and p2. Finally, we loop over that list of cells to check if any of them lies on an obstacle in the grid. If it does, we return immediately and decide that the trajectory between p1 and p2 is NOT collision-free.

For collinearity I select continuous groups of points (3 in each step) to see if they belong in a line orapproximately belong to a line. If they can be connected to a line I replace the two waypoints with a single one (longer) and continue the search to see if I can add more way points to the same line.

Execution of Flight Path

Goal location is added and controlled using flags during execution on the terminal.

(llat0 37.792480, lon0 -122.397450 and thedrone guided itself into it. To go back just run from the goal position:python motion_planning.py --lat 37.792480 --lon -122.397450 You can run any location you like by using the parameters lat,lon .

THE EXECUTION WORKS

The extra challenge of Helical / Spiral is also implemented.