1. EXPLAINING THE STARTER CODE (*MOTION_PLANNING.PY AND PLANNING_UTILS.PY*)

In the *plan_path* function in *motion_planning.py*, the drone is first placed in the "PLANNING" state. Then the drone altitude is set to the *TARGET_ALTITUDE*, which in this case is equal to 5 meters. Next, the obstacle data is read and the grid is created with this data using the function *create_grid.* This function just creates a map of the obstacle data given a certain altitude and safety distance. Now the start and goal points are defined, with the start being the center of the obstacle grid and the goal being 10 meters north and 10 meters east of the start point. Next, the *a_star* function is run to find the optimal path from the start to goal. In this case, the algorithm checks cells to the north, south, east, and west of the current position, given that they do not collide with an obstacle or the edge of the grid. The heuristic used in this version of *a_star* is simply the Euclidean distance between the current position and the goal position. Now the path found from *a_star* is converted to waypoints for the drone, correcting for the grid coordinate offset to convert the path into local coordinates.

2. COMPARE *MOTION_PLANNING.PY* AND *BACKYARD_FLYER_SOLUTION.PY*

The code in *motion_planning* is different than *backyard_flyer_solution.py,* because the backyard flyer is simply given four corners of a box as coordinates in the *calculate_box* function and uses no planning algorithm. This box is calculated once the drone takes off and reaches the target altitude. Meanwhile, the motion*_planning* code is given a start and goal position and calculates an optimal path between the two using methods discussed above. Other than this, the callbacks are all called in a similar manner and all other functions are implemented in the same way as before.

3. WRITEUP OF NEW PLANNING ALGORITHM

- The latitude and longitude coordinates are extracted from the *colliders* file using the *lat_lon* function. This function reads the first line of the file, separates the latitude and longitude coordinates, and converts this information into floating point values. These coordinates are then used in the *self.set_home_position()* function to set the global home. The following images show this code being implemented:

```
#Read lat0, lon0 from 'colliders' obstacle data into floating point values
lat0, lon0 = lat_lon('colliders.csv')
print('Home Lattidute: {0}, Home Longitude: {1}'.format(lat0, lon0))

self.set_home_position = (lon0, lat0, 0)
```

```
def lat_lon(file):
    #Read the lat0, lon0 from colliders data into floating point values
    with open(file) as File:
        lat_lon = File.readline()
    split = re.split(', ', lat_lon)
    lat = re.search('lat0 (.*)', split[0]).group(1)
    lon = re.search('lon0 (.*)', split[1]).group(1)
    return float(lat), float(lon)
```

- The local position is calculated relative to the global home position using the start latitude, longitude, and altitude through the function *global_to_local*. This can be seen in the image below:

```
start_latlon = [self._longitude, self._latitude, self._altitude]
start = global_to_local(start_latlon, self.global_home)
print("Local start location")
print(start)
```

- Start point is changed to the current position in the step above.
- Goal point is chosen to be an arbitrary position on the grid using geodetic coordinates input by the user running the code. These coordinates are then converted into local coordinates using the *global_to_local* function. Next, the goal location is checked to make sure it is not off the grid or at the same position as the start. If so, the drone is disarmed and stays at the start location. Finally, the goal is checked to see if the location is on the ground or on the roof of a building. This is done by first creating polygons of the obstacle data in the *extract*_polygons function. Then the goal location is checked to see if it collides with any of these polygons using the *collides* function. If it does not collide with a polygon (building), the drone sets its target altitude to 5 meters. If it does collide with a polygon (building), the target altitude is set to 5 meters above that building height. In addition, if the drone takes off from a building and is travelling to a goal location that has a lower altitude, the target altitude is set to 5 meters above the height of the starting location. This can all be seen in the code below.

```
#INPUT GOAL AS LONGITUDE/LATITUDE COORIDNATES
goal_lat_lon = (-122.394813, 37.793815)
goal_latlon = [goal_lat_lon[0], goal_lat_lon[1], 0]


#Goal location converted to local coordinate frame
goal = global_to_local(goal_latlon, self.global_home)
print("Local goal location")
print(goal)

if (goal[0] < north_min or goal[0] > north_max) or (goal[1] < east_min or goal[1] >
    east_max):
    self.disarming_transition()
    sys.exit("Goal location is off the grid!")

if LA.norm(goal[0:2] - start[0:2]) < 5:
    self.disarming_transition()
    sys.exit("Goal location is the same as start!")

#Check if the goal location is on a roof or the ground
polygons = extract_polygons(data)
does_collide = collides(polygons, goal)
if does_collide[0] == True:
    print("Landing on a roof!")
    TARGET_ALTITUDE = does_collide[1] + 5
    self.goal_altitude = does_collide[1] + 5
else:
    print("Landing on the ground!")
    TARGET_ALTITUDE = 5
    self.goal_altitude = 5

if TARGET_ALTITUDE < (-self.local_position[2] + 5):
    TARGET_ALTITUDE = int(-self.local_position[2] + 5)
```

```
def extract_polygons(data):

    polygons = []
    for i in range(data.shape[0]):
        north, east, alt, d_north, d_east, d_alt = data[i, :]

        corners = [(north - d_north, east - d_east),
                   (north - d_north, east + d_east),
                   (north + d_north, east + d_east),
                   (north + d_north, east - d_east)]

        height = alt + d_alt

        p = Polygon(corners)
        polygons.append((p, height))

    return polygons

def collides(polygons, point):

    for (p,h) in polygons:
        if p.contains(Point(point)):
            return (True, h)
    return (False, h)
```

- An A* search algorithm on a medial axis skeleton was used for this part of the program. First, the grid of obstacle data is created using the *create_grid* function and the skeleton is created using the information from this grid. Next, starting and stopping points are found on the skeleton, which are the closest points to the final start and goal locations. This is done with the *find_start_goal* function. The *a_star* function is used to search for an optimal path on this skeleton. The heuristic for the search algorithm is simply the Euclidean distance between the current point and the goal location on the skeleton. The *a_star* function uses the *Action* class which defines the 8 directions the drone may travel. The *valid_*actions function then determines which directions should be removed if they do not follow the path of the skeleton. The rest of the *a_*star algorithm is written the same as in previous exercises. If no path is found, the drone is disarmed. If a path is found, the final goal location is added to the path and the waypoints are ready to be pruned. This code can all be seen below:

```
    #Create a grid with obstacle data
    grid = create_grid(data, TARGET_ALTITUDE, SAFETY_DISTANCE)
    skeleton = medial_axis(invert(grid))

    #Find a start.goal location on the skeleton
    skel_start, skel_goal = find_start_goal(skeleton, [start[0] - north_min, start[1] -
        east_min], [goal[0] - north_min, goal[1] - east_min])

    #Find a path on the skeleton
    path, cost, found = a_star(invert(skeleton).astype(np.int), heuristic, tuple(skel_s
        tuple(skel_goal))
    if found == False:
        self.disarming_transition()
        sys.exit("No path was found!")
    path.append((int(goal[0]) - int(north_min), int(goal[1]) - int(east_min)))

def create_grid(data, drone_altitude, safety_distance):
    """
    Returns a grid representation of a 2D configuration space
    based on given obstacle data, drone altitude and safety distance
    arguments.
    """

    # minimum and maximum north coordinates
    north_min = np.floor(np.min(data[:, 0] - data[:, 3]))
    north_max = np.ceil(np.max(data[:, 0] + data[:, 3]))

    # minimum and maximum east coordinates
    east_min = np.floor(np.min(data[:, 1] - data[:, 4]))
    east_max = np.ceil(np.max(data[:, 1] + data[:, 4]))

    # given the minimum and maximum coordinates we can
    # calculate the size of the grid.
    north_size = int(np.ceil(north_max - north_min))
    east_size = int(np.ceil(east_max - east_min))

    # Initialize an empty grid
    grid = np.zeros((north_size, east_size))

    # Populate the grid with obstacles
    for i in range(data.shape[0]):
        north, east, alt, d_north, d_east, d_alt = data[i, :]
        if alt + d_alt + safety_distance > drone_altitude:
            obstacle = [
                int(np.clip(north - d_north - safety_distance - north_min, 0, north_siz
                int(np.clip(north + d_north + safety_distance - north_min, 0, north_siz
                int(np.clip(east - d_east - safety_distance - east_min, 0, east_size-1)
                int(np.clip(east + d_east + safety_distance - east_min, 0, east_size-1)
            ]
            grid[obstacle[0]:obstacle[1]+1, obstacle[2]:obstacle[3]+1] = 1

    return grid

def heuristic(position, goal_position):
    return np.linalg.norm(np.array(position) - np.array(goal_position))
```

```
def find_start_goal(skel, start, goal):

    start= np.array(start)
    goal = np.array(goal)
    skel_point = np.array(np.transpose(skel.nonzero()))
    start_dist = (np.linalg.norm(skel_point - start, axis = 1)).argmin()
    near_start = skel_point[start_dist]
    goal_dist = (np.linalg.norm(skel_point - goal, axis = 1)).argmin()
    near_goal = skel_point[goal_dist]
    return near_start, near_goal
```

```
def a_star(grid, h, start, goal):

    path = []
    path_cost = 0
    queue = PriorityQueue()
    queue.put((0, start))
    visited = set(start)

    branch = {}
    found = False

    while not queue.empty():
        item = queue.get()
        current_node = item[1]
        if current_node == start:
            current_cost = 0.0
        else:
            current_cost = branch[current_node][0]

        if current_node == goal:
            print('Found a path.')
            found = True
            break
        else:
            for action in valid_actions(grid, current_node):
                # get the tuple representation
                da = action.delta
                next_node = (current_node[0] + da[0], current_node[1] + da[1])
                branch_cost =  action.cost + current_cost
                queue_cost = branch_cost + h(next_node, goal)

                if next_node not in visited:
                    visited.add(next_node)
                    branch[next_node] = (branch_cost, current_node, action)
                    queue.put((queue_cost, next_node))

    if found:
        # retrace steps
        n = goal
        path_cost = branch[n][0]
        path.append(goal)
        while branch[n][1] != start:
            path.append(branch[n][1])
            n = branch[n][1]
        path.append(branch[n][1])

    return path[::-1], path_cost, found
```

```
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)
    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))

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

    return valid_actions
```

- First, the path is pruned using the *coll* (collinearity) and *bres* (bresenham) functions. Then headings are calculated for each waypoint on the pruned path. Finally, the waypoints are converted into local coordinates and set for the drone to use. This is seen in the code below:

```
if len(path) > 0:
    print("Pruning Waypoints")
    #Use collinearity, bresenham to prune path
    path = coll(path)
    path = bres(path, grid)

    #Find heading for each waypoint
    path = heading(path)

    # Convert path to waypoints
    waypoints = [[p[0] + int(north_min), p[1] + int(east_min), TARGET_ALTITUDE, p[2]] for p
        in path]
    print("Number of waypoints: ", len(waypoints))
    print("Waypoints: ")
    print(waypoints)
self.waypoints = waypoints
print("set wayponits")
```

```
def bres(path, grid):
    i = 0
    while True:
        if i > (len(path)-3):
            break
        p1 = path[i]
        j = len(path) - 1
        while True:
            if j == (i+1):
                break
            p2 = path[j]
            cells = list(bresenham(int(p1[0]), int(p1[1]), int(p2[0]), int(p2[1])))
            in_collision = False
            for c in cells:
                if c[0] < 0 or c[1] <0 or c[0] >= grid.shape[0] or c[1] >= grid.shape[1]:
                    in_collision = True
                    break
                if grid[c[0], c[1]] == 1:
                    in_collision = True
                    break
            if in_collision == False:
                for r in range(i+1,j):
                    path.remove(path[i+1])
                break
            else:
                j = j-1
        i = i+1

    return path

def coll(path):
    i = 0
    while True:

        if collinearity(path[i], path[i+1], path[i+2], epsilon=1e-2) == True:
            path.remove(path[i+1])
            if i > len(path) - 3:
                break
        else:
            i = i + 1
            if i > len(path) - 3:
                break
    return path
```

```
def collinearity(p1, p2, p3, epsilon):
    collinear = False
    p1= np.append(p1, 1)
    p2= np.append(p2, 1)
    p3= np.append(p3, 1)
    mat= np.vstack((p1, p2, p3))
    det= np.linalg.det(mat)
    if det < epsilon:
        collinear = True

    return collinear


def heading(path):
    path[0] = list(path[0])
    path[0].append(0)
    for i in range(0, len(path)-1):
        p1 = path[i]
        p2 = path[i+1]
        head = np.arctan2((p2[1] - p1[1]), (p2[0] - p1[0]))
        path[i+1] = list(path[i+1])
        path[i+1].append(head)

    return path
```

- The drone is able to start from the ground or roof and similarly land on the ground or roof. All points that were tested worked as expected. When testing the code, feel free to try starting/landing from any point on the map and let me know how it does!
- Some other functions that were changed in *motion_planning* include *local_position_callback* and *velocity_callback.* The position callback was modified to make the waypoint transition criteria a function of velocity. Here, if the difference between the target and local position is smaller than the magnitude of the velocity +1, there is a waypoint transition. This function of velocity was manipulated and tested until it seemed the drone was making smooth transitions at multiple speeds. The addition of 1 was to ensure the drone did not get stuck if it came to rest. The velocity callback was modified to ensure that if the drone was landing on a building, it would disarm at the proper time. This code can be seen below:

```
def local_position_callback(self):
    if self.flight_state == States.TAKEOFF:
         if -1.0 * self.local_position[2] > 0.95 * self.target_position[2]:
            self.waypoint_transition()
    elif self.flight_state == States.WAYPOINT:
        if np.linalg.norm(self.target_position[0:2] - self.local_position[0:2]) <
            (np.linalg.norm(self.local_velocity[0:2]) + 1):
            if len(self.waypoints) > 0:
                self.waypoint_transition()
            else:
                if np.linalg.norm(self.local_velocity[0:2]) < 1.0:
                    self.landing_transition()

def velocity_callback(self):
    if self.flight_state == States.LANDING:
        if abs(self.local_position[2]) < (self.goal_altitude - 5 + .01):
            self.disarming_transition()
```