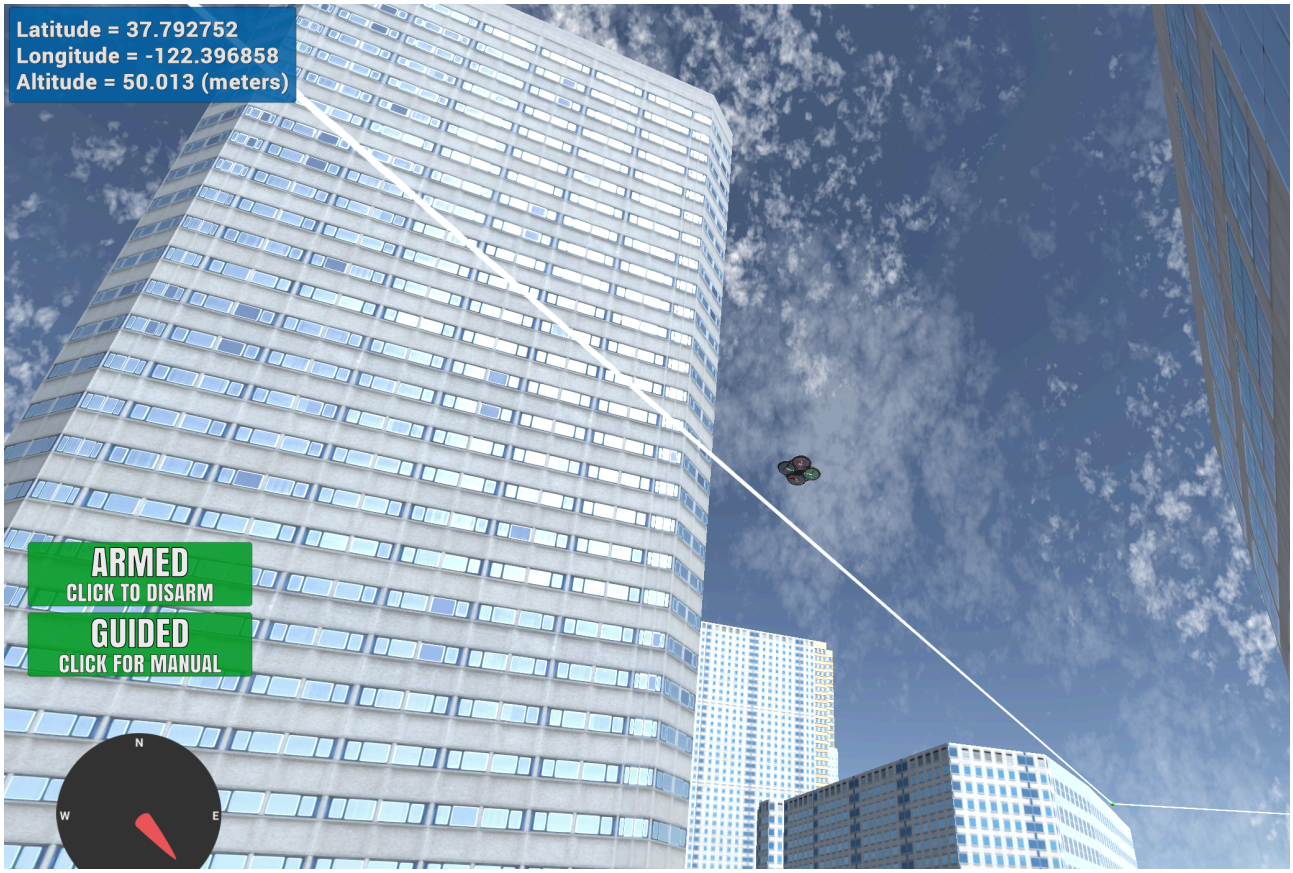
Implementation of a Quadcopter Motion Planning Algorithm

By Emmanuel Ezenwere

Project No.2 of the Udacity Flying Car Nanodegree.



Our Quadcopter navigating autonomously through a simulation of the City Of San Fransisco

Building on the the Udacity FCND planning project starter codes.

The planning project starter codes includes the motion_planning.py and planning_utils.py codes.

The planning_utils code includes basic functions to generate a grid given obstacle points contained in the colliders.csv file, an implementation of the A* search algorithm, an implementation of the euclidean distance heuristic for estimating the cost from a current node to a goal during A start search, a definition of valid actions and criteria for selecting valid actions from any given node/state in the grid.

The motion_planning.py code includes a States class and a motion planning class which contains function definitions of the various quadcopter transitions and a template for the plan_path function.

Unlike the backyard_flyer_solution.py code which restricts the trajectory/ waypoints of the drone to a box defined in the calculate_box function.

```
def calculate_box(self):
    print("Setting Home")
    local_waypoints = [[10.0, 0.0, 3.0], [10.0, 10.0, 3.0], [0.0, 10.0, 3.0], [0.0, 0.0, 3.0]]
    return local_waypoints
```

Also for the backyard_flyer_solution, the box trajectory is made after take_off transition while in the motion_planning.py the path is created immediately after the arming transition before take_off.

```
def state_callback(self):
    if self.in_mission:
        if self.flight_state == States.MANUAL:
            self.arming_transition()
        elif self.flight_state == States.ARMING:
            if self.armed:
                self.takeoff_transition()
        elif self.flight_state == States.DISARMING:
            if ~self.armed & ~self.guided:
                self.manual_transition()
```

backyard_flyer_solution.py state_callback definition

```
def state_callback(self):
    if self.in_mission:
        if self.flight_state == States.MANUAL:
            self.arming_transition()
        elif self.flight_state == States.ARMING:
            if self.armed:
                self.plan_path()
        elif self.flight_state == States.PLANNING:
            self.takeoff_transition()
        elif self.flight_state == States.DISARMING:
            if ~self.armed & ~self.guided:
                self.manual_transition()
```

motion_planning.py state callback definition

Set Home Position for Quadcopter

I wrote code to read the first line of the colliders.csv file to extract lat0 and lon0 as floating point values then I used the `self.set_home_position()` method to set our global home position to be the obtained lat0, lon0 and default alt = 0 values.

```
# TODO: read lat0, lon0 from colliders into floating point values.
filename = 'colliders.csv'
# sub_data has the format 'lat0 37.792480, lon0 -122.397450\n'.
sub_data = open(filename).readline()
# re-format "sub_data" string to remove strings "lat0", "long0" and "\n".
lat0, lon0 = [float(i) for i in sub_data.replace('\n', '').replace('lat0 ', '').replace(' lon0 ', '').split(',')]

# TODO: set home position to (lon0, lat0, 0)
self.set_home_position(lon0, lat0, 0)
print("\nglobal home position (lat, lon) : " + str((lat0, lon0)))
```

motion_planning.py - code to set home portion

Determined local position relative to global home position.

In the starter code, it is assumed the drone takes off from map center in order to make the choice of where the drone takes off more flexible I had to retrieve the current position of the drone and compute the relative position of the drone to the defined global home position.

To obtain the current global position of the drone I used `self._latitude`, `self._longitude` and `self._altitude and` `global_to_local()` to obtain the relative position of the drone with the current global position and `self.global_home` as parameters.

```
# TODO: read lat0, lon0 from colliders into floating point values.
filename = 'colliders.csv'
# sub_data has the format 'lat0 37.792480, lon0 -122.397450\n'.
sub_data = open(filename).readline()
# re-format "sub_data" string to remove strings "lat0", "long0" and "\n".
lat0, lon0 = [float(i) for i in sub_data.replace('\n', '').replace('lat0 ', '').replace(' lon0 ', '').split(',')]

# TODO: set home position to (lon0, lat0, 0)
self.set_home_position(lon0, lat0, 0)
print("\nglobal home position (lat, lon) : " + str((lat0, lon0)))
```

motion_planning.py - code to compute relative local position

Added flexibility to the start location

I changed the start position of our path planner from the center of the grid to the current position of the drone in a generated grid.

To generate the grid,I considered the full extent of all obstacle points from the colliders.csv file and set the dimensions of the grid such that the grid encompasses all obstacles retaining their relative positions (from the global home).

To obtain the current position/ now start position of the drone on the grid, I applied the north_offset and east_offset values to translate the data points generated from the colliders.csv file to the newly defined grid and obtain their new co-ordinates.

```
# ************************************* generating grid ************************************************
# Read in obstacle map
data = np.loadtxt('colliders.csv', delimiter=',', dtype='Float64', skiprows=3)

# Define a grid for a particular altitude and safety margin around obstacles
grid, north_offset, east_offset = create_grid(data, TARGET_ALTITUDE, SAFETY_DISTANCE)
# The north and east offsets are the north and east values by which the actual layout points have been
# translated to fit on the grid.
print("\nNorth offset = {0}, east offset = {1}".format(north_offset, east_offset))
# ******************************************************************************************************

# TODO: convert start position to current position rather than map center
# Define starting point on the grid (this is just grid center)
# grid_start = (-north_offset, -east_offset)

# changing grid start position to become current local position.

grid_north_pos = int(np.ceil(local_north_position - north_offset))
grid_east_pos = int(np.ceil(local_east_position - east_offset))

grid_start = (grid_north_pos, grid_east_pos)
print("\ndrone mission start grid position:", grid_start)
```

motion_planning.py - code to modify start location on the grid to equivalent current position on grid

Added flexibility to goal location

I changed the start position of our path planner from the default point 10m north, 10m east of center of the grid to a random un-occupied point on the grid.

To generate a random un-occupied point on the grid I wrote a function `random_goal_search()` to choose a random un-occupied point on the grid and compute the equivalent global position of that point.

```
def random_goal_search(grid, global_home, north_offset, east_offset):
    """
     Random Goal Search randomly selects an un-occupied point on the grid and converts to global_position given a
     global_home position and off-set values during co-ordinate translation.

    :param grid: numpy array occupancy grid of any size.
    :param global_home: reference global position (lon,lat,alt) on grid.
    :param north_offset: translation along the north during co-ordinate transformation to grid.
    :param east_offset: translation along the east during co-ordinate transformation to grid.
    :return global_position: np.array (lon, lat, alt) of random un-occupied point on grid.
    """
    # global_position_ref
    rows, cols = np.where(grid == 0)
    free_points = [(rows[i], cols[i]) for i in range(rows.shape[0])]

    # obtain indices of free states on the grid.
    random_index = np.random.randint(len(free_points))
    grid_pos = free_points[random_index]

    # relative distances to grid reference point: lat0, lon0
    local_north = grid_pos[0] + north_offset
    local_east = grid_pos[1] + east_offset

    local_position = (local_north, local_east, 0)
    global_position = local_to_global(local_position, global_home)

    return global_position
```

```
        # TODO: adapt to set goal as latitude / longitude position and convert
        global_goal_pos = random_goal_search(grid, self.global_home, north_offset, east_offset)
        local_goal_pos = global_to_local(global_goal_pos, self.global_home)
        grid_north_pos = int(np.ceil(local_goal_pos[0] - north_offset))
        # avoided taking np.ceil because global_to_local gives an overestimate by +1 of local_goal_pos[1] from
        # local_goal_pos[1] value computed by random_goal_search. Look up the def random_goal_search function in
        # planning_utils.py.
        grid_east_pos = int(local_goal_pos[1] - east_offset)

        grid_goal = (grid_north_pos, grid_east_pos)
        print("drone mission goal grid position:", grid_goal)
```

Added diagonal motions to A* search algorithm

I edited the Actions class in the planning_utils.py. code to include diagonal motions in the NW, NE, SW & SE directions with an action cost of root 2 corresponding to the diagonal distance between vertices of a 1unit x 1 unit block. To enable the A* search algorithm used to determine a path to the goal consider diagonal motions during planning.

```
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)
    NORTHEAST = (-1, 1, np.around(np.sqrt(2), decimals=3))
    NORTHWEST = (-1, -1, np.around(np.sqrt(2), decimals=3))
    SOUTHWEST = (1, -1, np.around(np.sqrt(2), decimals=3))
    SOUTHEAST = (1, 1, np.around(np.sqrt(2), decimals=3))
```

Also, I added conditions for which translations by diagonal motion is valid from a current node in the `valid_actions()` function definition.

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

    return valid_actions_
```

Pruned path of unnecessary waypoints using a Collinearity Test algorithm.

I implemented a collinearity check function to test if any three points are collinear (lie on the same line). If a point turns out to be collinear with two other points I eliminated the point.

This significantly smoothened the trajectory of the drone and reduced the stops the drone had to proceed to the next waypoint. along a given straight line trajectory.

```
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):
    if path is not None:
        # TODO: prune the path!
        p1 = path[0]
        p2 = path[1]

        pruned_path = [p1, p2]

        for p in path[2:]:

            if collinearity_check(point(p1), point(p2), point(p)):

                pruned_path = pruned_path[0:-1]
                pruned_path.append(p)

            else:
                pruned_path.append(p)

            p2 = pruned_path[-1]
            p1 = pruned_path[-2]

    else:
        pruned_path = path
```

In addition, I implemented probabilistic roadmap planning. I experimented with various number of nodes and branching factors and observed experimentally that the greater the values for b and d, the more likely it is that there will be a sequence of edges that connects an arbitrary start node to another arbitrary goal node. I also included code to enable the drone align it's heading with the next waypoint during flight.

In Summary:

I really enjoyed this project because of the use of the Udacity FCND simulator to simulate the behaviour of the drone while implementing the path planning algorithm and the immediate applicability of the software on an actual drone.