3D Motion Planning

Minh Gia Hoang

Explain the Starter Code

`motion_planning.py` contains a basic planning implementation that includes MAVLink connection with the drone simulator and callback functions for the position, velocity, and state changes similar to `backyard_flyer.py`. The main difference is the `motion_planning.py` has a new state PLANNING when `plan_path()` method is called. This method is executed before taking off to set the waypoints for the quadrotor to follow the desired trajectory. In particular, a start and a goal positions are manually set. An A* algorithm then finds a list of waypoints from the start to the goal, if the path was found. Next, the path is pruned by removing redundant intermediate waypoints. Finally, the waypoints are sent to the simulator to execute the trajectory and for visualization.

`planning_utils.py` contains utilities to aid the planning implementation. It provides a `create_grid()` creating a 2-D grid map whose cells with value 1 are considered as "obstacles", an `a_star()` implementing a grid-based A* algorithm with helper functions like a heuristic function and an Action class defining possible actions to take in the grid.

Implementing Your Path Planning Algorithm

Set your global home position

We read the `colliders.csv` file and extract the initial global position as latitude / longitude coordinates.

```
with open('colliders.csv') as f:
      home_pos_data = f.readline().split(',')
lat0 = float(home_pos_data[0].strip().split(' ')[1])
lon0 = float(home_pos_data[1].strip().split(' ')[1])
self.set_home_position(longitude=lon0, latitude=lat0, altitude=0.0)
```

Set your current local position

We determine the quadrotor's local position relative to global home to plan the route on the local map from the current location.

```
global_position = (self._longitude, self._latitude, self._altitude)
local_position = global_to_local(global_position, self.global_home)
```

Set grid start position from local position

This is another step in adding flexibility to the start location. The local position in the previous step is converted to the position on the grid map in pixels.

```
grid_start = int(local_position[0]) - north_offset, int(local_position[1]) -
east_offset
```

Set grid goal position from geodetic cords

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. Random goal position allows us to test the planner without manually changing the goal location.

First, we define a distance range in meters from the home position to the goal position and covert it to angular range in degrees using equatorial radius of 6380000 m.

```
distance_range = 100
angular_range = distance_range / 6380000 * 180 / np.pi # lat/lon range to localize a
goal relative to home global pose (in degrees).
```

We then sample the global goal (horizontal) location randomly and convert it to the position on the grid.

```
lat_goal = lat0 + (2*np.random.rand()-1)*angular_range
lon_goal = lon0 + (2*np.random.rand()-1)*angular_range
alt_goal = 0.0
global_position_goal = lon_goal, lat_goal, alt_goal
local_goal = global_to_local(global_position_goal, self.global_home)
grid_goal = int(local_goal[0]) - north_offset, int(local_goal[1]) -east_offset
```

Nest we check if the random position is valid (not inside an obstacle) and repeat the step above until a collision-free location is found.

```
while grid[grid_goal[0], grid_goal[1]]:
      lat_goal = lat0 + (2*np.random.rand()-1)*angular_range
      lon_goal = lon0 + (2*np.random.rand()-1)*angular_range
      alt_goal = 0.0
      global_position_goal = lon_goal, lat_goal, alt_goal
      local_goal = global_to_local(global_position_goal, self.global_home)
      grid_goal = int(local_goal[0]) - north_offset, int(local_goal[1]) - east_offset
```

Modify A* to include diagonal motion

Minimal requirement here is to modify the code in `planning_utils()` to update the A* to include diagonal motions on the grid that have a cost of $\sqrt{2}$. We update the actions and the associate weights as follows.

```
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))
```

The new actions are check as follows.

```
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)
```

Cull waypoints

In this step we can use a collinearity test to prune the path of unnecessary waypoints. The main idea is to delete one of the three consecutive waypoints if they lie on a line. This collinearity condition is implemented by checking if the matrix constructed by 3 waypoints has full rank. If the matrix determinant is equal to zero, the 3 waypoints lie on a line. A threshold (epsilon = 0.5) handles the case when the waypoints are approximately collinear. This helps to avoid slow and jerky movements when the waypoints are very dense relative to each other.

```
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, epsilon):
    pruned_path = [p for p in 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 collinearity_check(p1, p2, p3, epsilon):
            pruned_path.remove(pruned_path[i+1])
        else:
            i += 1
    return pruned_path
```

Results

Due to large file, please consult the result in https://drive.google.com/file/d/1c1SN8ZXCm5UHIisuo-8NUaubu254TNdf/view?usp=sharing