Neema Rustin Badihian
4/30/18

Motion-Planning Writeup

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 variables MANUAL, ARMING, TAKEOFF, WAYPOINT, LANDING, and DISARMING use `auto()` from `enum` to be initialized.

The function calculate_box does not exist since we're not flying in a square like we were in backyard_flyer. This means that local_position_callback does not call calculate_box since it doesn't exist in this file.

arming_transition does not set the drone's home position.

In `takeoff_transition`, `self.takeoff()` uses `target_position[2]` instead of `target_altitude` because `target_position[2]` is set to the target altitude. Same function, less lines.

In `waypoint_transition`, cmd_poisition commands the drone to move with a heading of whatever is held in target_position[3] instead of just 0.0.

`send_waypoints` sets the waypoints for the drone, using the data from `data` that gets `waypoints` from the points in `path`

`plan_path` uses the data from `colliders.csv` to create a path for the drone and then sends the drone the waypoints

At the end of the file, we parse the information to run the connection beforehand.

Students 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. Explain briefly how you accomplished this in your code.

I used a standard `open()` to open `colliders.csv`. I read the first line with `readline()` and split it into two variables using `.split()`. The variables I got were then reassigned to two new variables but only the float objects in them.

Determine your local position relative to global home you'll be all set. Explain briefly how you accomplished this in your code.

I created a variable as set its value to that of `self.global_position`. That gave me the current global position of the drone. Then I used `global_to_local()` to convert my new variable to local. This new value was then given to a new variable I named `local_position`.

Neema Rustin Badihian
4/30/18

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. Explain the code you used to accomplish this step.

I didn't actually modify `a_star` for this, it didn't seem necessary. Instead, I modified `Action` and `valid_action`.

In `Action`, I added four variables named `NORTHWEST`, `NORTHEAST`, `SOUTHWEST`, and `SOUTHEAST`. Since the value of `SOUTH` is 1 and the second value of `WEST` is -1, I set the first two values of `SOUTHWEST` as 1, -1. The third value is the cost of the action which we were told is the square root of 2 so I set the third value as 2**0.5, or 2 to the power of ½ which is the same as the square root of 2. I applied this logic to the three other new variables and gave them their appropriate values.

In `valid_actions`, I added four if statements, each with two layers, to remove the actions `NORTHWEST`, `NORTHEAST`, `SOUTHWEST`, and `SOUTHEAST`. For `NORTHEAST`, I took the if statement designed to remove `Action.EAST` and nested it in the if statement designed to remove `Action.NORTH`. If both conditions were met, `valid_actions.remove(Action.NORTHEAST)` was executed. I did the same to remove the other three actions.

You can use a collinearity test or ray tracing method like Bresenham. The idea is simply to prune your path of unnecessary waypoints. Explain the code you used to accomplish this step.

To prune the path, I added `prune_path`, `point`, and `collinearity_check` from our exercises into planning_utils.py. I imported these methods into motion_planning.py and ran `prune_path` with `path`.

`prune_path` takes a given path creates an array, `pruned_path`, from the values in the path. It then counts the length of the array and as long as a variable, i, is less than the count - 2, it performs the following:

> A variable called p1 is created which is created by running `pruned_path[i]` through `point`.
>
> `point` takes the first two values of `pruned_path[i]` and sets these two values along with a third value of 1 to a numpy array. This numpy array is then reshaped with (1, -1) and is returned to p1 in `prune_path`. The same is done with two more variables created in `prune_path` named p2 and p3 but in these cases, `pruned_path[i+1]` and `pruned_path[i+2]` are used.
>
> p1, p2, and p3 are passed to `collinearity_check`. `collinearity_check` tests to see if these three points are in a line. If the points are in a line, `collinearity_check` returns

True to `prune_path` and the middle point, `p2` or `pruned_path[i+1]` is removed from `pruned_path`. If the points are not in a line and `collinearity_check` returns False, i is incremented.

When i is finally equal to the length of the count of `pruned_path` - 2, `prune_path` returns `pruned_path` to the variable `path` in `motion_planning.py`.

A QUICK NOTE

My code works but it doesn't work 100% of the time. Sometimes, no path is found. Right now, I have it set up so that a goal is created with a random integer from `north_min` to `north_max` and `east_min` to `east_max` from `create_graph` are used as the graph's coordinates. I will try to find a way to change this so that if no path is found, a new destination is set and a new path is attempted to be found until one is successful.