Project-2: 3D Motion Planning of a Quadrotor

1. Description

This second project on Udacity's Flying Car Nanodegree consists of planning and executing a trajectory of a drone in an urban environment. Built on top of the event-based strategy utilized on the first project, the complexity of path planning in a 3D environment is explored. The code communicates with Udacity FCND Simulator using Udacidrone API.

2. Rubric Points

Here I will consider the rubric points individually and describe how I addressed each point in my implementation.

2.1 Explain the Starter Code

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

`State_callback` function in the `motion_planning.py` includes

```
self.plan_path()
elif self.flight_state == States.PLANNING:
self.takeoff_transition()
```

The state order For backyard_flyer.py is: MANUAL > ARMING > TAKEOFF > WAYPOINT > LANDING > DISARMING

For `motion_planning.py` is: MANUAL > ARMING > PLANNING > TAKEOFF > WAYPOINT > LANDING > DISARMING

The `motion_planning.py` has a planning state and a plan_path method, where as the `backyard_flyer.py` has not.

The plan_path method includes the following steps, currently missing steps are marked with (T)

- Set the target_position, TARGET_ALTITUDE
- (T) read lat0, lon0 from colliders into floating point values
- (T) set home position to (lon0, lat0, 0)
- (T) retrieve current global position
- (T) convert to current local position using global_to_local()
- Read the obstacle map
- Define a grid for a particular altitude and safety margin around obstacles
- Define starting point on the grid (this is just grid center)
- (T) Convert start position to current position
- Set goal as some arbitrary position on the grid
- (T) Adapt to set goal as latitude / longitude position and convert
- Run A* to find a path from start to goal, with a heuristic cost
- Add diagonal motions with a cost of sqrt(2) to your A* implementation
- Prune path
- Convert path to waypoints
- Send the waypoints to the sim to visualize

There are three main functions to perform planning, all of them are present in the `planning_utils.py`.

1. heuristic

This function provides a heuristic function of the distance between two given points.

2. c**reate_grid**

This function converts the map to a 2D grid at the given altitude and using the specified safety distance.

3. a_star

This function creates a path using the a_star algorithm. It takes the grid, heuristic function, start position and the goal position as inputs. Currently, it performs the search by applying all possible actions at a state. It selects the next state with the lowest cost.

Currently, the goal position is hardcoded as some location 10 m north and 10 m east of map center. There are no diagonal motions. Therefore, the planning results in one step north and one step east.

Implementing Your Path Planning Algorithm (2D Grid A* algorithm)

There are five algorithms implemented. These are

1. 2D Grid A* Algorithm
2. 3D Voxmap A* Algorithm
3. 2D Graph
4. Probabilistic Roadmap
5. RRT

To run 2D grid A* planning, there are two options. One is to run the `motion_planning.py` file. The possible parameters are `global_goal, local_goal, grid_goal`. If more than one arguments are sent, only the first one would have affect. The other option is the `motionplot.ipnyb` notebook.

1. Set your global home position

The plan_path method reads the `colliders.csv` file and gets the lat0 and lon0 values. Convert these values to float, and send them to the self.set_home_position function. This function sets the global home position.

2. Set your current local position

The local position is the output of the `self.global_to_local` function. The inputs for the function are, `self.global_position` and the `self.global_home`.

3. Set grid start position from local position

The start position is calculated from the local position obtained in the previous step. The local position value consists of the north value, the east value and the altitude.

However, these north and east values are not ready to use. It is required to subtract the offset values. Offset values are calculated in the create_grid function in the `planning_utils.py` file. They are the minimum north value, and the minimum east value from the data read from the `colliders.csv`. The create_grid function convert these offset values to integer. The values in the grid_start tuple are also converted to integer.

For 3D planning, grid_start tuple also contains the altitude. That value is calculated by summing the altitude value in the global position and the TARGET_ALTITUDE value.

4. Set grid goal position from geodetic coords

There are two ways to set the grid goal. Sending a geodetic coord(lat, lon) is one way. If there is no parameter sent, then the plan_path method sets the goal randomly.

Then the method converts the goal location to local coordinates, and then the grid coordinates. If it exceeds an edge of the grid, then the value is changed back to the value of the edge in order to stay in the grid.

For 3D planning, the randomly assigned altitude value has a maximum that is defined by the MAX_ALTITUDE.

5. Modify A* to include diagonal motion (or replace A* altogether)

For 2D planning, the action object in the `planning_utils.py` file includes 4 additional actions that are diagonal.

In the first version of the a_star function in the `planning_utils.py` file, when a node can be reached with a lower cost after it was added to the branch, it is not possible to change the path for that node. In order to correct this error, the code location to add a node to the visited list is changed. A node is added to the visited list, only when it is the current node. Previously, it was added when it was the next_node. This change enabled a node to be placed in the queue more than once. It will be visited with the lowest queue cost. After it is visited, the other items in the queue, containing the same node, will be skipped without processing.

One problem was timeout. If planning takes too much time, it can't send the waypoints to the simulator. Increasing the timeout parameter of the connection object was not a solution. In that case, the MotionPlanning (Drone) instance and the simulator stuck at the TAKEOFF state.

The solution for this problem, creating a new MotionPlanning instance and sending the waypoints with that instance.

Another problem was zigzag moves. The solution was to first check the previous action. For a new node, if applying the previous action is valid, then first extend to that node. This provide a more prioritized order to that action in the queue.

Another thing to try was adding a cost for changing action. This cost shall be significantly lower than the action cost (action cost / 100). Otherwise, it changes the behavior of the

a_star planning. This cost increased the process time by 50%. Therefore, it doesn't present in the current planner.

To increase process speed, number of steps taken with each step was increased. Current planner takes five steps with a single action. If the current node and the goal are closer than 5 * sqrt(2), then it is only possible to move a single step. The parameter to modify the number of steps is max_move in the plan_path method.

6. Cull waypoints

The collinearity_check was performed to determine whether a point in the path can be removed or not. If three points fit in the same line, then the second of them was removed.

3D Grid A* algorithm

The 3D A* planning is implemented in the `motionplot3Dvox` notebook.

There are two different voxel maps. The former, which is the smaller one, has a voxel size of 5, and the latter has 1 voxel size. The main planning runs on the smaller voxel map. Since the voxel representation of the start and the goal points may not correspond to the actual points, the other planning calculates the path from start point to voxel and the path from voxel to goal point on the larger voxel map with voxel size 1.

To decrease the planning time, the a_star algorithm runs in both ways. From start to goal and from goal to start. In cases such as planning to a hole, this algorithm significantly lowers the computation time. When a node is visited by both of the paths, it is accepted as the midpoint. And the path from start to the midpoint and the path from the midpoint to the goal are combined to represent the complete path.

Graph A* algorithm

The 2D Graph A* planning is implemented in the `motionplotgraph` notebook. 2D grid A* planning converted to plan from a graph instead of a grid. There is Grid class to store created grid and graph instances.

Probabilistic Roadmap

The Probabilistic Roadmap is implemented in the `probabilisticroadmap` notebook. 2D Graph A* planning converted to create a graph with a probabilistic method. There is GridProbabilistic class to store created grid and graph instances, and the hyper parameters and methods to create a probabilistic graph.

RRT

The Rapidly-Exploring Random Tree is implemented in the `rrt notebook`. The probabilistic Roadmap converted to the RRT method. There is `GridRRT` class to store created grid and graph instances, and the hyper parameters and methods to create a rrt.

Executing the flight

To execute the code you need to change to the repo directory and create the conda environment with the following command:

```
conda env create -f environment.yml
```

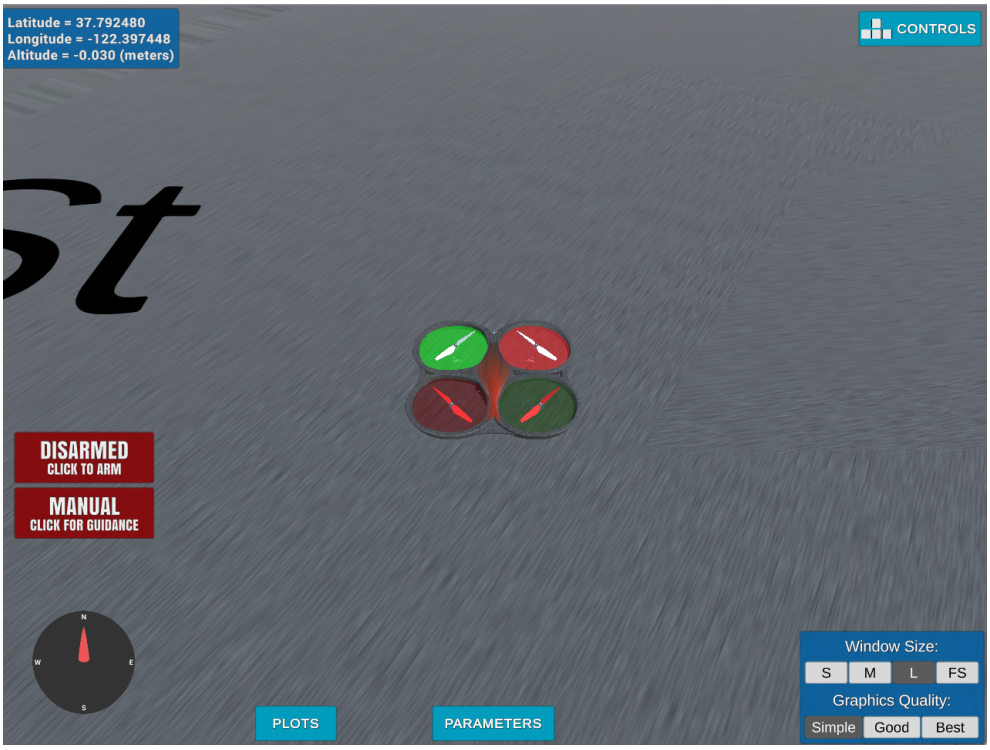
Note: This environment configuration is provided by Udacity at the FCND Term 1 Starter Kit repo.

Activate your environment with the following command:

```
source activate fcnd
```

Start the drone simulator. You will see something similar to the following image:

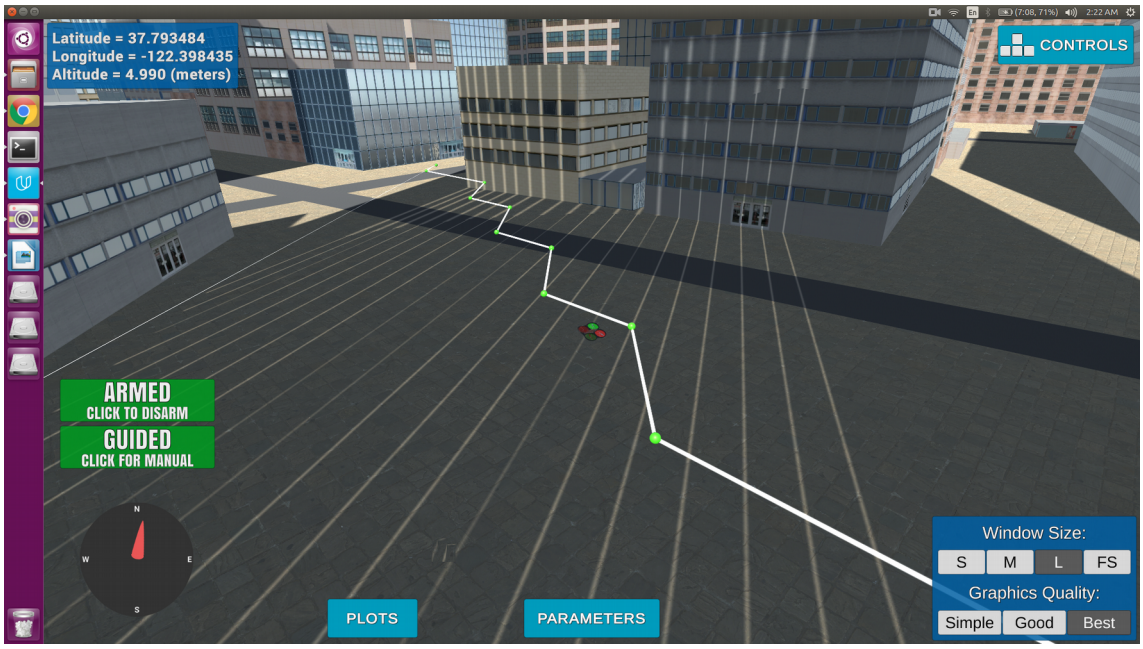
Select the Motion Planning project, and you will get to the following environment:

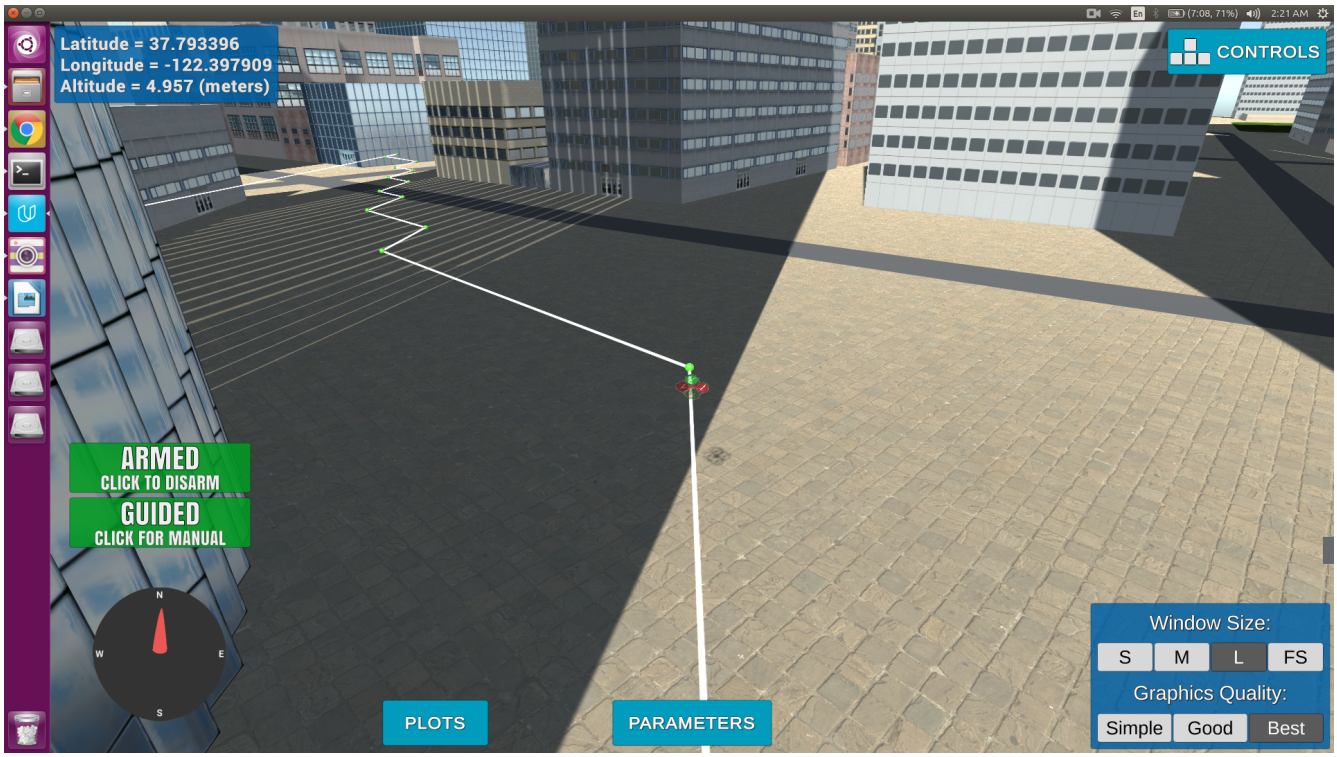


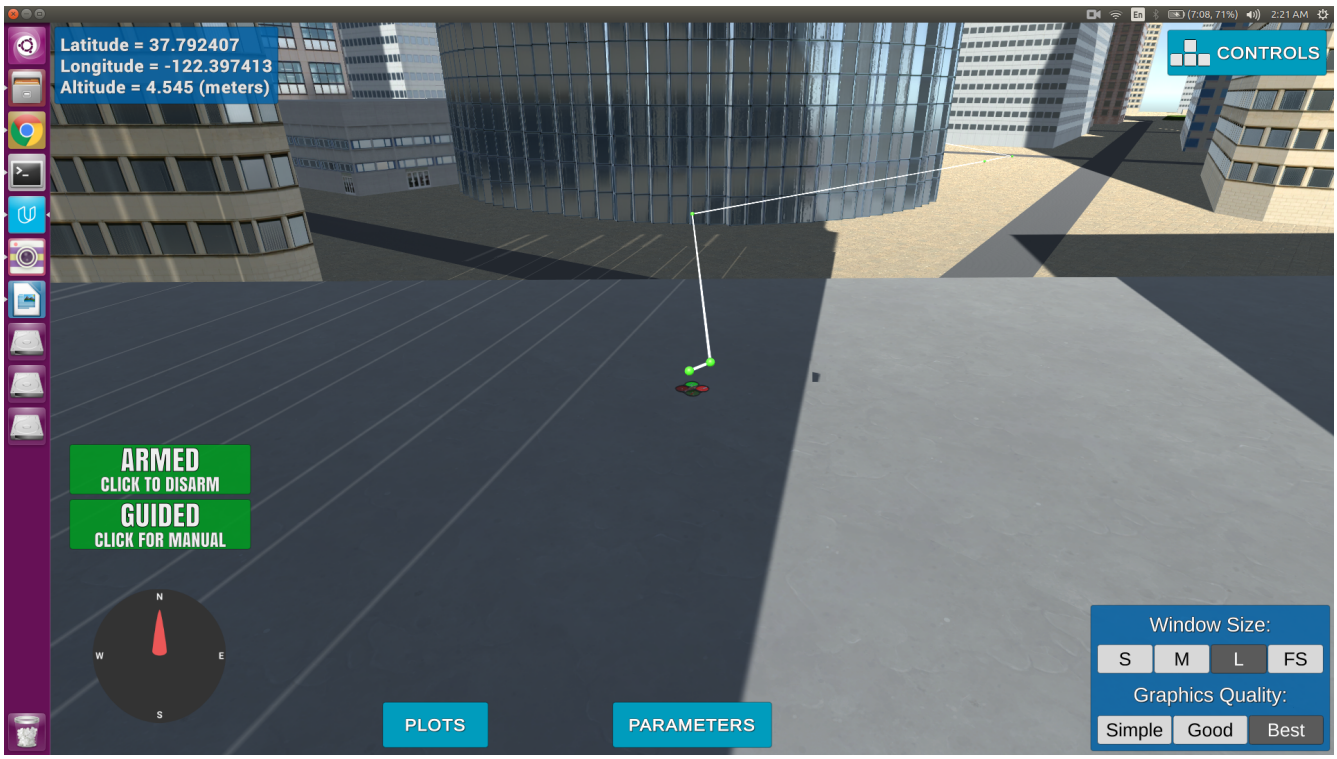
Now is time to run the code, for the A* grid implementation:

```
python motion_planning.py
```

A visualization of the planned path is shown in the below images and in the attached video:




Latitude = 37.793396
Longitude = -122.397909
Altitude = 4.957 (meters)
CONTROLS
ARMED
CLICK TO DISARM
GUIDED
CLICK FOR MANUAL
N
W
E
S
PLOTS
PARAMETERS
Window Size:
S
M
L
FS
Graphics Quality:
Simple
Good
Best


Latitude = 37.792407
Longitude = -122.397413
Altitude = 4.545 (meters)
CONTROLS
ARMED
CLICK TO DISARM
GUIDED
CLICK FOR MANUAL
N
W
E
S
PLOTS
PARAMETERS
Window Size:
S
M
L
FS
Graphics Quality:
Simple
Good
Best