FCND Motion Planning Project Write-up

1-Explaining the starter code

-Planning_utils.py

In this file there are the following functions which is called from the (Motion_planning.py) file to help planning the drone trip:

a) creat_grid: this function is responsible of making a grid representation for the map given by the "colliders" file at a specified altitude and a safety distance around all the no-flying areas like buildings.
This function takes the coordination of the obstacles center and its dimension to make a zeros numpy array with the same size of the map in meters and specifies all obstacles with ones, then return that array of zeros & ones, and the grid center coordinates.

```python
from enum import Enum
from queue import PriorityQueue
import numpy as np


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_size-1)),
                int(np.clip(north + d_north + safety_distance - north_min, 0, north_size-1)),
                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, int(north_min), int(east_min)
```

b) Action: a class represents and returns what action the drone is going to make in order to go to the next cell in the grid in each direction (North, East, .. etc.), and the cost of each action.

```python
# Assume all actions cost the same.
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)

    @property
    def cost(self):
        return self.value[2]

    @property
    def delta(self):
        return (self.value[0], self.value[1])
```

c) valid_actions: it's a function that check all the adjacent grid cells to the current cell if they are representing an obstacle, off the grid, or free to fly in. and then return the valid actions of the empty grid cells.

```
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)

    return valid_actions
```

d) a_star: the job of this function is to search for the shortest path from the start to the goal positions if there is one, and give that path a cost relative to the valid actions in that path.

It takes the grid representation, start & goal coordinates, and heuristic function as inputs, then return wither there is a path or not and the path cells coordinates & it’s cost if there is one.

```
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 = current_cost + action.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]
```

```python
            path.append(branch[n][1])
        else:
            print('**********************')
            print('Failed to find a path!')
            print('**********************')
        return path[::-1], path_cost
```

e) heuristic: it’s a function that computes and return the linear direct distance between a given location and the goal location.

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

But I’ve neglected some of these functions (like: creat_grid, Action, and valid_actions) and changed the (a_star) function because I used [Probabilistic Roadmap] searching approach

---

-Motion_planning.py

This basic code consists of two main parts needed to guide the drone from a start position to the goal position. these two are “States” & “MotionPlanning” classes.

1- States Class: it’s the class which lists all the states that the drone will be going through in the entire flight from the start to the goal.

```python
class States(Enum):
    MANUAL = auto()
    ARMING = auto()
    TAKEOFF = auto()
    WAYPOINT = auto()
    LANDING = auto()
    DISARMING = auto()
    PLANNING = auto()
```

2- MotionPlanning Class: is responsible to check the drone current state and guide it to the next phase of the flight as follows:

a) First phase is changing the mode of the drone in the simulator from manual to guided.
b) Second phase is arming the drone and make it ready to take off be turning its blades.
c) In the path planning phase, we specify a certain target altitude and safety distance around the obstacles, then load our grid representation of the map given these parameters. After that we set our start & goal positions and call the a_star function to search for a path by passing it our grid representation, start & goal coordinates, and the heuristic function. And at the end save all waypoints from the start to the goal.
d) fourthly, ordering the drone to take off to the target altitude.
e) And then pass each waypoint that follows the current position till the drone reaches the goal position. by that, the waypoint transition phase ends.
f) The drone start landing all the way down to the ground level in the landing phase.
g) Finally disarming the drone and return it back to the manual mode.

2-Setting the Global Home Position

Here I used the **np.genfromtxt** function to read the first line only of the colider.csv file, then split up the strings separated by spaces and converted the lon0 & lat0 from strings to float numbers and assigned each value to a corresponding variable.

```
    def plan_path(self):
        self.flight_state = States.PLANNING
        print("Searching for a path ...")
        TARGET_ALTITUDE = 5
        SAFETY_DISTANCE = 5

        self.target_position[2] = TARGET_ALTITUDE

        # TODO: read lat0, lon0 from colliders into floating point values
        home_coord_data = np.genfromtxt('colliders.csv', delimiter=',', dtype='str', replace_space=',', max_rows=1)
        lon0 = float(home_coord_data[1].split()[1])
        lat0 = float(home_coord_data[0].split()[1])
        #print(lon0, lat0, type(lon0), type(lat0))
        # TODO: set home position to (lon0, lat0, 0)

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

After that sated the home position to these values.

3- local position relative to global home

In this step I get the drone local position relative to the global home position by **global_to_local** function and assign it to **local_pos** variable.

```
        # TODO: retrieve current global position

        start_global = self.global_position

        # TODO: convert to current local position using global_to_local()

        local_pos = global_to_local(start_global, self.global_home)
        print('global home {0}, position {1}, local position {2}'.format(self.global_home, self.global_position,
                                                                         self.local_position))
```

4- Setting the start point for planning

Now setting the drone start position as the home position instead of the map center and saving its values as integers.

```
        # TODO: convert start position to current position rather than map center
        start = (int(local_pos[0]), int(local_pos[1]), int(local_pos[2]))
```

5- Setting goal position from geodetic coords

By now, I set the goal as (lon, lat, alt) format, and then convert it by **global_to_local** function, then assign the integer values of them to the goal variable.

```
        # Set goal as some arbitrary position on the grid
        goal_global = (-122.399144, 37.793597, TARGET_ALTITUDE)
        # TODO: adapt to set goal as latitude / longitude position and convert
        goal = global_to_local(goal_global, self.global_home)
```

6- Search Space Algorithm

In this very point I've needed to change a lot in the starter code because I choose to use **Probabilistic Roadmap** searching approach, and for that it's a must to modify the **a_star** function also adding up some other functions in the **planning_utils.py** file to help in the process.

Let us start by importing all the libraries needed in the **planning_motion.py** file …

```
from planning_utils import a_star, heuristic, extract_polygons, collides, create_graph, polygon_for_landing #create_
from udacidrone import Drone
from udacidrone.connection import MavlinkConnection
from udacidrone.messaging import MsgID
from udacidrone.frame_utils import global_to_local, local_to_global

#import sys
#import pkg_resources
#pkg_resources.require("networkx==2.1")
import networkx as nx
from sklearn.neighbors import KDTree
from shapely.geometry import Polygon, Point, LineString
from queue import PriorityQueue
```

Also I’ve added the following functions to **planning_utils.py** file:

- extract_polygons: to read the obstacles from the **collides.csv** file and make polygons around them by sufficient safety distance.

```python
def extract_polygons(data, SAFETY_DISTANCE):
    print("Extracting polygons ...")
    polygons = []
    for i in range(data.shape[0]):
        x, y, alt, d_x, d_y, d_alt = data[i, :]

        #Extract the 4 corners of the obstacle
        point1 = (np.int32(x - d_x - SAFETY_DISTANCE), np.int32(y - d_y - SAFETY_DISTANCE))
        point2 = (np.int32(x + d_x + SAFETY_DISTANCE), np.int32(y - d_y - SAFETY_DISTANCE))
        point3 = (np.int32(x + d_x + SAFETY_DISTANCE), np.int32(y + d_y + SAFETY_DISTANCE))
        point4 = (np.int32(x - d_x - SAFETY_DISTANCE), np.int32(y + d_y + SAFETY_DISTANCE))

        corners = [point1, point2, point3, point4]

        #Compute the height of the polygon
        height = np.int32(alt + d_alt + SAFETY_DISTANCE)

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

    #print(polygons[0][0])
    return polygons
```

- collides: to check if the passed points to it lies inside any obstacles or around it by less than the safety distance, and remove the points that collides.

```python
def collides(polygons, point):
    #Determining if the point collides with any obstacles or not.

    p = Point(point[:2])
    for (poly, height) in polygons:
        if poly.contains(p) and height >= point[2]:
            break

    return poly.contains(p) and height >= point[2]
```

- can_connect: to check if the filtered points can be connected to each other without passing through any obstacle polygon.

```python
def can_connect (p1, p2, polygons):
    line = LineString([p1, p2])
    for polygon in polygons:
        if polygon[0].crosses(line) and polygon[1] >= min(p1[2], p2[2]):
            return False
    return True
```

- create_graph: used to connect between the points & making a graph that doesn’t collide with any obstacle.

```
def create_graph (nodes, k, polygons):
    print("Creating graph ...")
    from sklearn.neighbors import KDTree
    import numpy.linalg as LA
    import networkx as nx
    G = nx.Graph()
    tree = KDTree(nodes)
    for node in nodes:
        idxs = tree.query([node], k=k, return_distance=False)[0]

        for idx in idxs:
            node2 = nodes[idx]
            if node2 == node:
                continue

            if can_connect(node, node2, polygons):
                dist = LA.norm(np.array(node2) - np.array(node))
                G.add_edge(node, node2, weight=dist)
    return G
```

- a_star: to search for the shortest distance between the start & goal points through the graph, if there is any. Then return the result by passing the path points on the graph and its cost if it has found a route.

```python
def a_star(graph, h, start, goal):

    """
    Modified A* to work with NetworkX graphs.
    """
    print("Finding best route ...")
    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 node in graph[current_node]:
                next_node = node
                branch_cost = current_cost + graph[current_node][node]['weight']
                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)
                    queue.put((queue_cost, next_node))

    if found:
```

```python
    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])
    else:
        print('**********************')
        print('Failed to find a path!, please try again')
        print('**********************')
    return path[::-1], path_cost
```

And I will use the same Heuristic function provided

7- Back to our planning_motion.py file:

After setting the goal values, we need to pass the obstacles data to extract the polygon of it by a safety distance …

```
        polygons = extract_polygons(data, SAFETY_DISTANCE)
```

Then sampling some points randomly in the margins of our map represented in the obstacles data, also to make it more precise I reduced this margins to be a square with a side of the length of 1.5 times the biggest distance between the start and the goal in north & east directions, then filter these points by the **collides** function …

```
        print("sampling points ...")
        map_xmin = np.min(data[:, 0] - data[:, 3])
        map_xmax = np.max(data[:, 0] + data[:, 3])

        map_ymin = np.min(data[:, 1] - data[:, 4])
        map_ymax = np.max(data[:, 1] + data[:, 4])

        north_max = np.max(np.array([start[0], goal[0]]))
        north_min = np.min(np.array([start[0], goal[0]]))

        east_max = np.max(np.array([start[1], goal[1]]))
        east_min = np.min(np.array([start[1], goal[1]]))

        bigger_side = np.max(np.array([(north_max - north_min)/4, (east_max - east_min)/4]))

        n_max = north_max + bigger_side
        n_min = north_max - bigger_side

        e_max = east_max + bigger_side
        e_min = east_min - bigger_side

        scope_nmax = np.min(np.array([map_xmax, n_max]))
        scope_nmin = np.max(np.array([map_xmin, n_min]))

        scope_emax = np.min(np.array([map_ymax, e_max]))
        scope_emin = np.max(np.array([map_ymin, e_min]))

        zmin = np.int32(TARGET_ALTITUDE)
        zmax = np.int32(TARGET_ALTITUDE)

        num_samples = 20
        xvals = np.random.uniform(scope_nmin, scope_nmax, num_samples).astype(int)
        yvals = np.random.uniform(scope_emin, scope_emax, num_samples).astype(int)
        zvals = np.random.uniform(zmin, zmax, num_samples).astype(int)

        samples = list(zip(xvals, yvals, zvals))
```

Note that here I limited the margin in the height direction (zmax & zmin) by our altitude to make our sampling margin even more precise

And so we keep the filtered points in a list (to_keep).

```
        to_keep = []
        to_keep.append((np.int32(start[0]), np.int32(start[1]), np.int32(TARGET_ALTITUDE))) #np.int32(z)
        to_keep.append((np.int32(goal[0]), np.int32(goal[1]), np.int32(-goal[2])))
        for point in samples:
            if not collides(polygons, point):
                to_keep.append(point)
        print(to_keep)
```

Now it's time to make our graph based representation of the environment, so we pass the sample points, polygons, and the number of which each point will be connected to the 4 nearest available points to it …

```
        g = create_graph(to_keep, 4, polygons)
        print("Number of edges", len(g.edges))
```

Finally, we run our a_star function to find the best route between the start and the goal positions, if there is any. so we pass to it our graph, heuristic function, start, and goal points.

And print the numbers of nodes & the cost of the path (which is the distance here) …

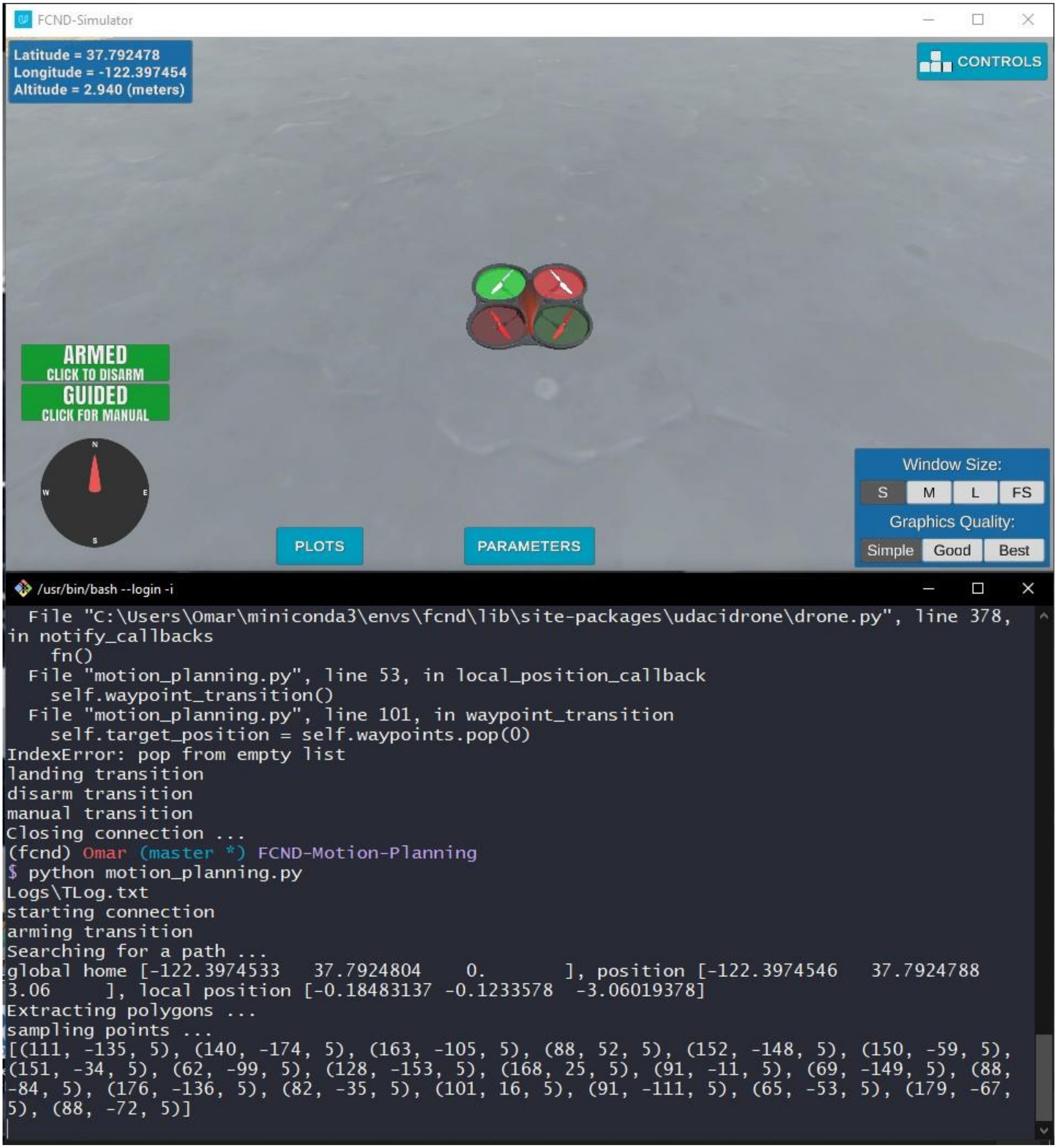
```
        a_start = (np.int32(start[0]), np.int32(start[1]), np.int32(TARGET_ALTITUDE))
        a_goal = (np.int32(goal[0]), np.int32(goal[1]), np.int32(-goal[2]))
        #print(type(a_start), type(a_goal))
        path, cost = a_star(g, heuristic, a_start, a_goal)
        print("Number of nodes in the Path:", len(path))
        print("Path Cost: ", cost)
```

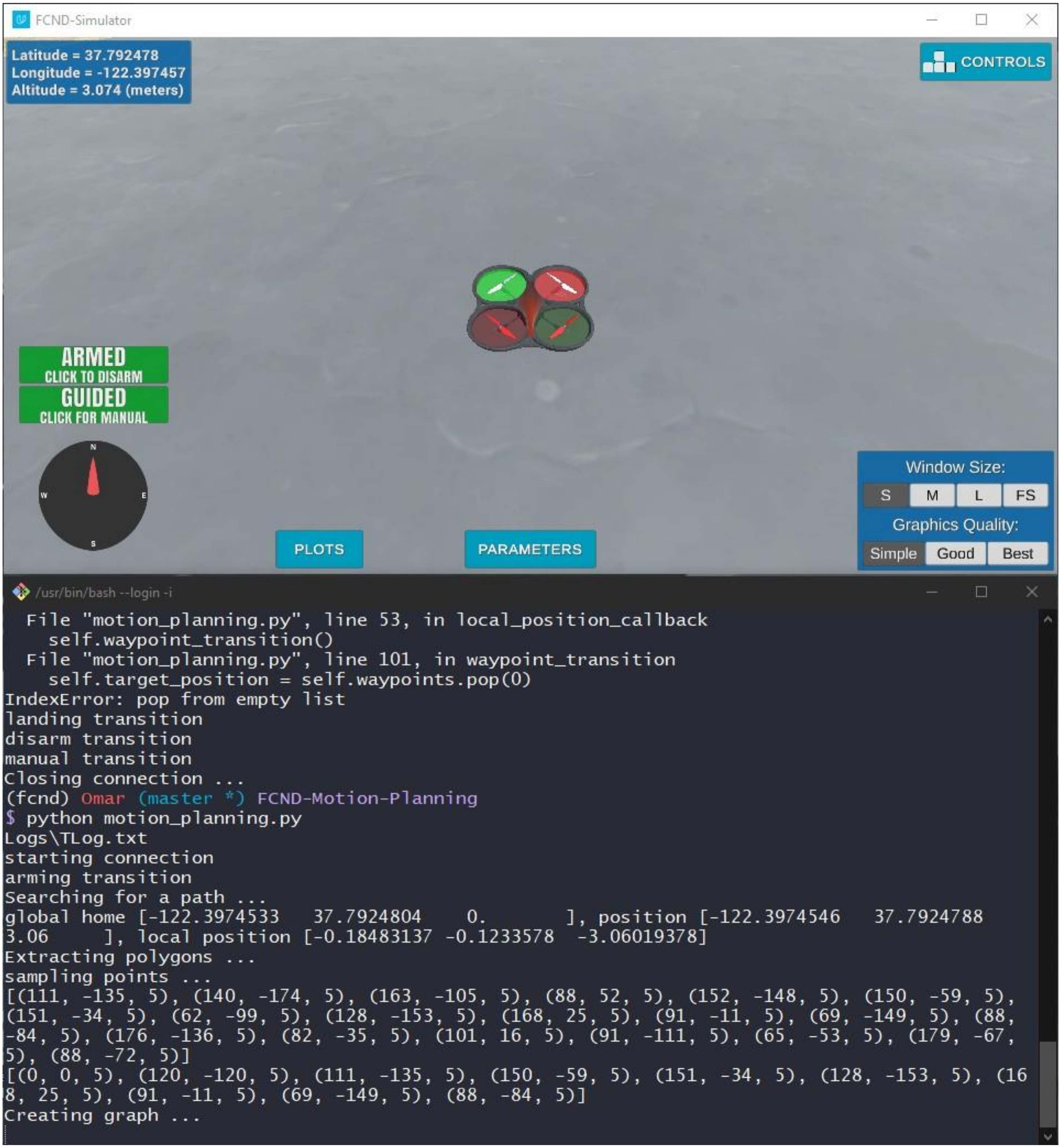
8- Once it finishes the search, we pass the returned path to extract the waypoints from it …

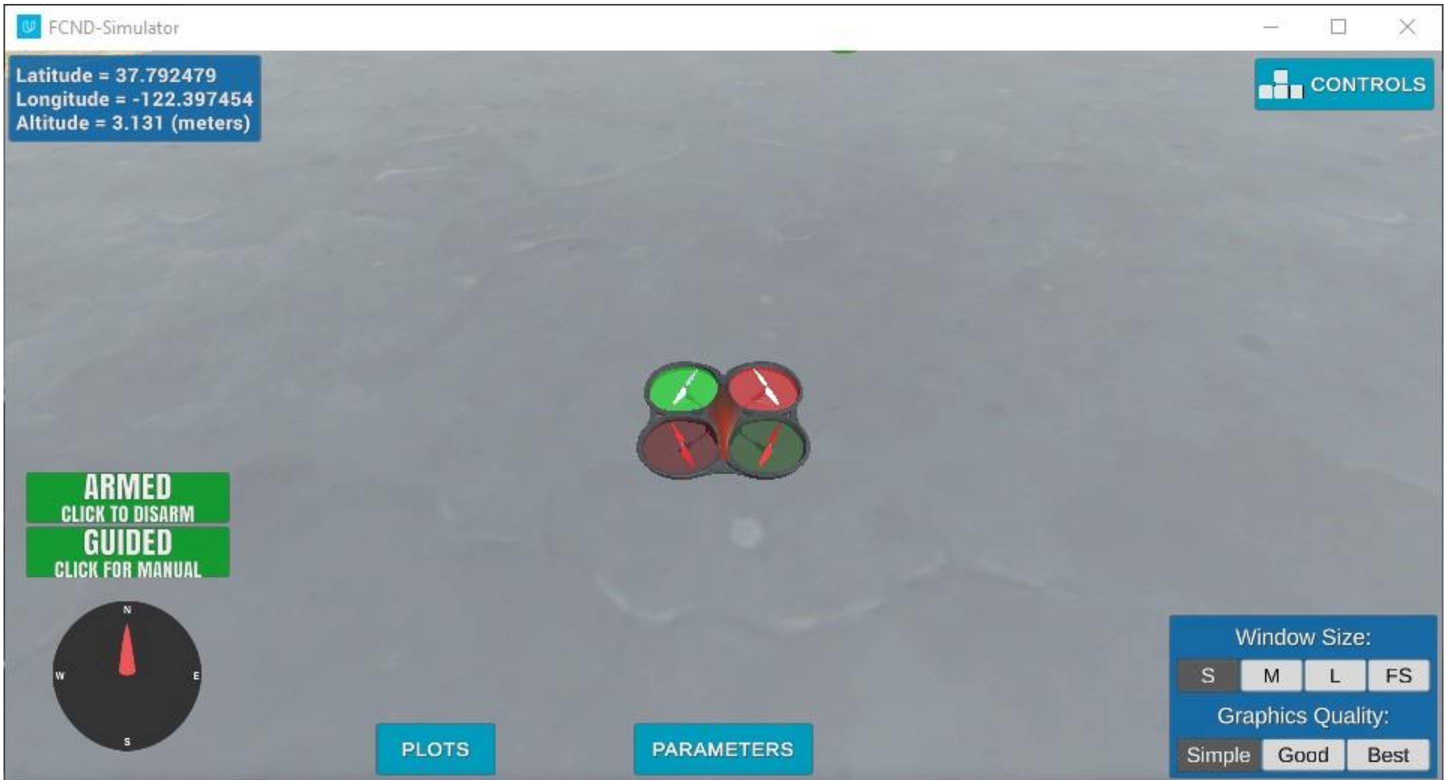
```
        # Convert path to waypoints
        #waypoints = [[p[0] + north_offset, p[1] + east_offset, TARGET_ALTITUDE, 0] for p in path]
        waypoints = [[int(p[0]), int(p[1]), int(p[2]), 0] for p in path]
        print(waypoints)
        # Set self.waypoints
        self.waypoints = waypoints
        # TODO: send waypoints to sim (this is just for visualization of waypoints)
        self.send_waypoints()
```

Important to mention that I used limited number of sample points and the number of connections between sample points to avoid making the process last more than 1 minute or error message will appear.

And below I've taken some screenshots of the simulator while it runs ☺

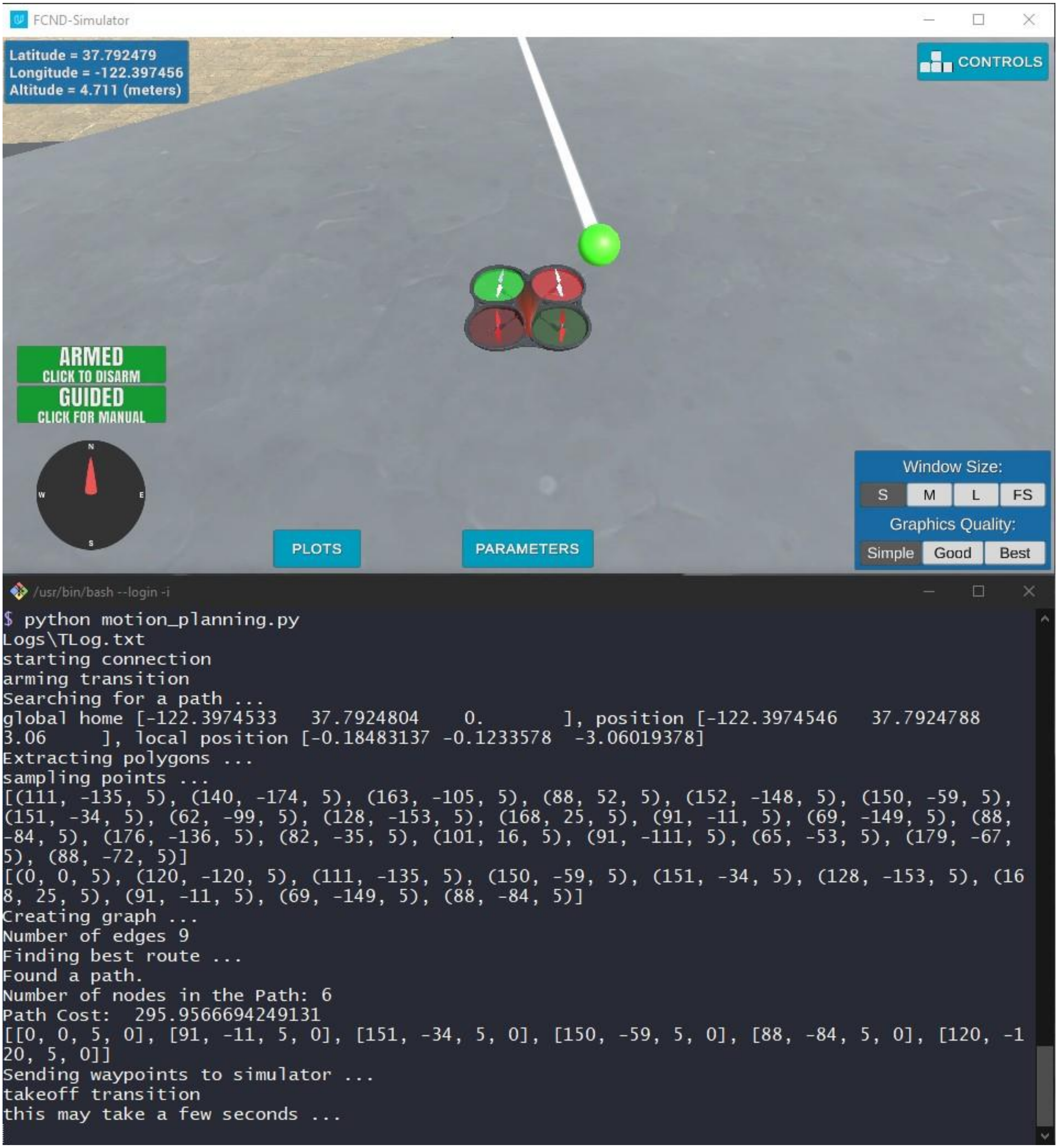


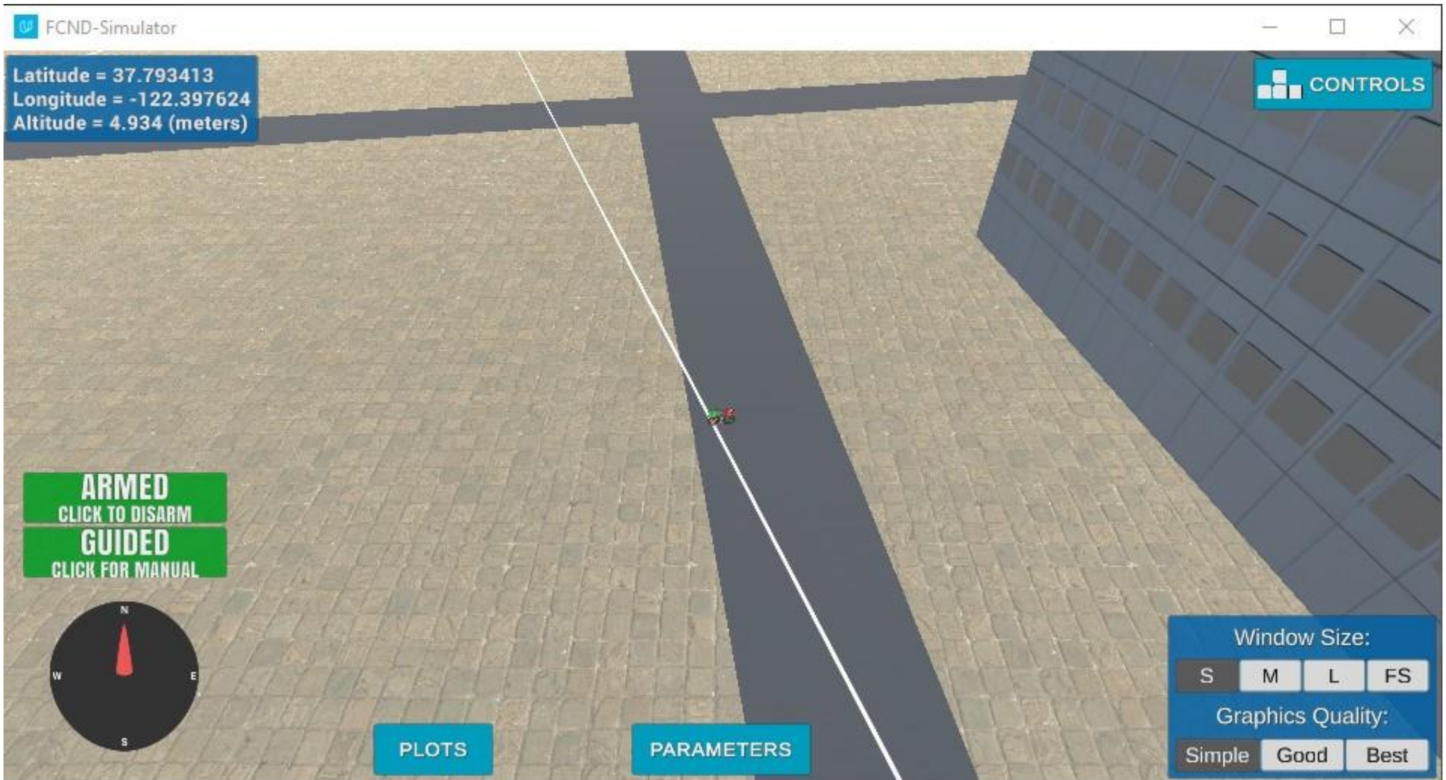

FCND-Simulator
Latitude = 37.792478
Longitude = -122.397457
Altitude = 3.074 (meters)
CONTROLS
ARMED
CLICK TO DISARM
GUIDED
CLICK FOR MANUAL
N
W
E
S
PLOTS
PARAMETERS
Window Size:
S
M
L
FS
Graphics Quality:
Simple
Good
Best
/usr/bin/bash --login -i
  File "motion_planning.py", line 53, in local_position_callback
    self.waypoint_transition()
  File "motion_planning.py", line 101, in waypoint_transition
    self.target_position = self.waypoints.pop(0)
IndexError: pop from empty list
landing transition
disarm transition
manual transition
Closing connection ...
(fcnd) Omar (master *) FCND-Motion-Planning
$ python motion_planning.py
Logs\TLog.txt
starting connection
arming transition
Searching for a path ...
global home [-122.3974533   37.7924804   0.   ], position [-122.3974546   37.7924788
3.06   ], local position [-0.18483137 -0.1233578 -3.06019378]
Extracting polygons ...
sampling points ...
[(111, -135, 5), (140, -174, 5), (163, -105, 5), (88, 52, 5), (152, -148, 5), (150, -59, 5),
(151, -34, 5), (62, -99, 5), (128, -153, 5), (168, 25, 5), (91, -11, 5), (69, -149, 5), (88,
-84, 5), (176, -136, 5), (82, -35, 5), (101, 16, 5), (91, -111, 5), (65, -53, 5), (179, -67,
5), (88, -72, 5)]
[(0, 0, 5), (120, -120, 5), (111, -135, 5), (150, -59, 5), (151, -34, 5), (128, -153, 5), (16
8, 25, 5), (91, -11, 5), (69, -149, 5), (88, -84, 5)]
Creating graph ...



/usr/bin/bash --login -i

```
$ python motion_planning.py
Logs\TLog.txt
starting connection
arming transition
Searching for a path ...
global home [-122.3974533   37.7924804    0.        ], position [-122.3974546   37.7924788
3.06      ], local position [-0.18483137 -0.1233578  -3.06019378]
Extracting polygons ...
sampling points ...
[(111, -135, 5), (140, -174, 5), (163, -105, 5), (88, 52, 5), (152, -148, 5), (150, -59, 5),
(151, -34, 5), (62, -99, 5), (128, -153, 5), (168, 25, 5), (91, -11, 5), (69, -149, 5), (88,
-84, 5), (176, -136, 5), (82, -35, 5), (101, 16, 5), (91, -111, 5), (65, -53, 5), (179, -67,
5), (88, -72, 5)]
[(0, 0, 5), (120, -120, 5), (111, -135, 5), (150, -59, 5), (151, -34, 5), (128, -153, 5), (16
8, 25, 5), (91, -11, 5), (69, -149, 5), (88, -84, 5)]
Creating graph ...
Number of edges 9
Finding best route ...
Found a path.
Number of nodes in the Path: 6
Path Cost:  295.9566694249131
[[0, 0, 5, 0], [91, -11, 5, 0], [151, -34, 5, 0], [150, -59, 5, 0], [88, -84, 5, 0], [120, -1
20, 5, 0]]
Sending waypoints to simulator ...
takeoff transition
this may take a few seconds ...
```


FCND-Simulator
Latitude = 37.792479
Longitude = -122.397456
Altitude = 4.711 (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
/usr/bin/bash --login -i
$ python motion_planning.py
Logs\TLog.txt
starting connection
arming transition
Searching for a path ...
global home [-122.3974533   37.7924804   0.   ], position [-122.3974546   37.7924788
3.06   ], local position [-0.18483137 -0.1233578 -3.06019378]
Extracting polygons ...
sampling points ...
[(111, -135, 5), (140, -174, 5), (163, -105, 5), (88, 52, 5), (152, -148, 5), (150, -59, 5),
(151, -34, 5), (62, -99, 5), (128, -153, 5), (168, 25, 5), (91, -11, 5), (69, -149, 5), (88,
-84, 5), (176, -136, 5), (82, -35, 5), (101, 16, 5), (91, -111, 5), (65, -53, 5), (179, -67,
5), (88, -72, 5)]
[(0, 0, 5), (120, -120, 5), (111, -135, 5), (150, -59, 5), (151, -34, 5), (128, -153, 5), (16
8, 25, 5), (91, -11, 5), (69, -149, 5), (88, -84, 5)]
Creating graph ...
Number of edges 9
Finding best route ...
Found a path.
Number of nodes in the Path: 6
Path Cost: 295.9566694249131
[[0, 0, 5, 0], [91, -11, 5, 0], [151, -34, 5, 0], [150, -59, 5, 0], [88, -84, 5, 0], [120, -1
20, 5, 0]]
Sending waypoints to simulator ...
takeoff transition
this may take a few seconds ...



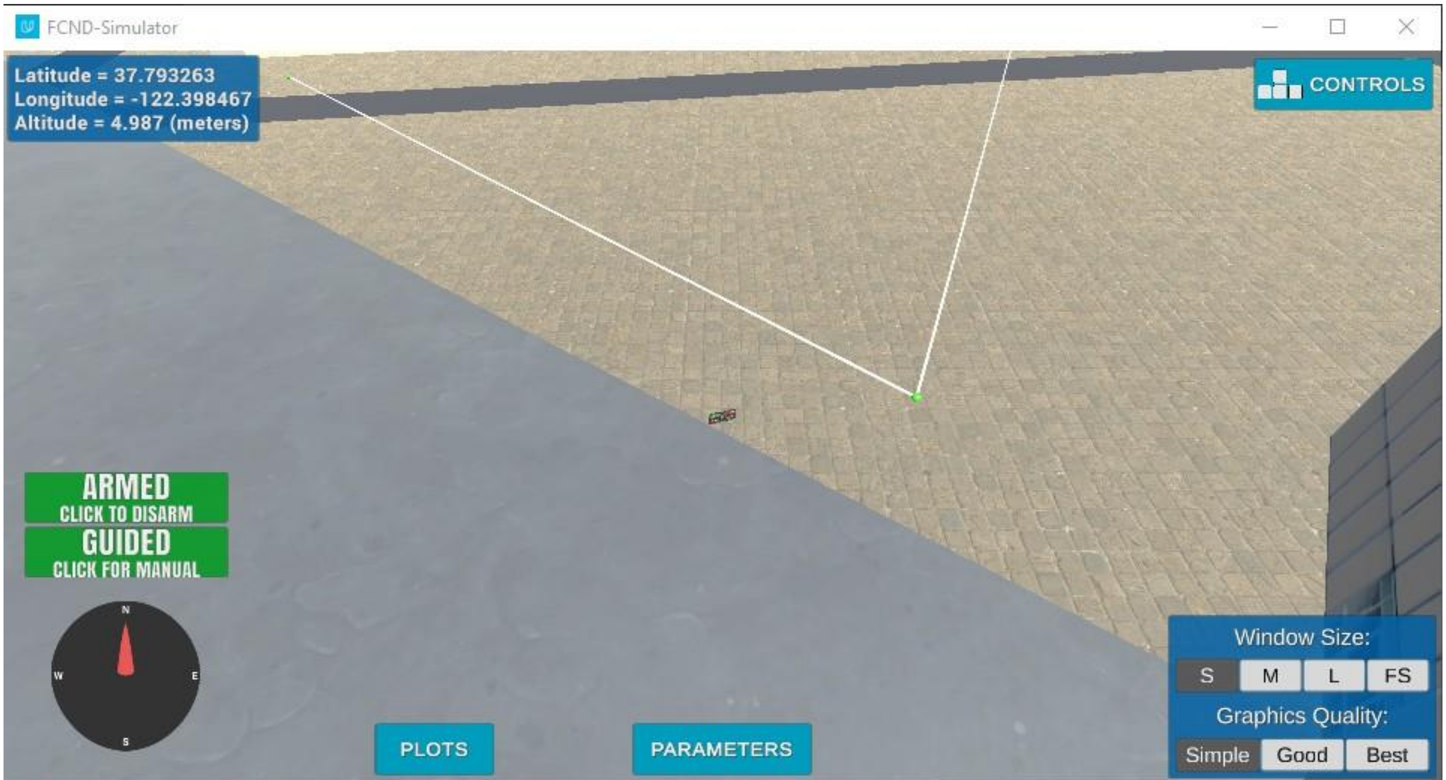
/usr/bin/bash --login -i

```
3.06      ], local position [-0.18483137 -0.1233578  -3.06019378]
Extracting polygons ...
sampling points ...
[(111, -135, 5), (140, -174, 5), (163, -105, 5), (88, 52, 5), (152, -148, 5), (150, -59, 5),
(151, -34, 5), (62, -99, 5), (128, -153, 5), (168, 25, 5), (91, -11, 5), (69, -149, 5), (88,
-84, 5), (176, -136, 5), (82, -35, 5), (101, 16, 5), (91, -111, 5), (65, -53, 5), (179, -67,
5), (88, -72, 5)]
[(0, 0, 5), (120, -120, 5), (111, -135, 5), (150, -59, 5), (151, -34, 5), (128, -153, 5), (16
8, 25, 5), (91, -11, 5), (69, -149, 5), (88, -84, 5)]
Creating graph ...
Number of edges 9
Finding best route ...
Found a path.
Number of nodes in the Path: 6
Path Cost:  295.9566694249131
[[0, 0, 5, 0], [91, -11, 5, 0], [151, -34, 5, 0], [150, -59, 5, 0], [88, -84, 5, 0], [120, -1
20, 5, 0]]
Sending waypoints to simulator ...
takeoff transition
this may take a few seconds ...
waypoint transition
target position [0, 0, 5, 0]
waypoint transition
target position [91, -11, 5, 0]
waypoint transition
target position [151, -34, 5, 0]
```

15°C مشمس جزئيا ENG 2:49 PM 3/11/2022

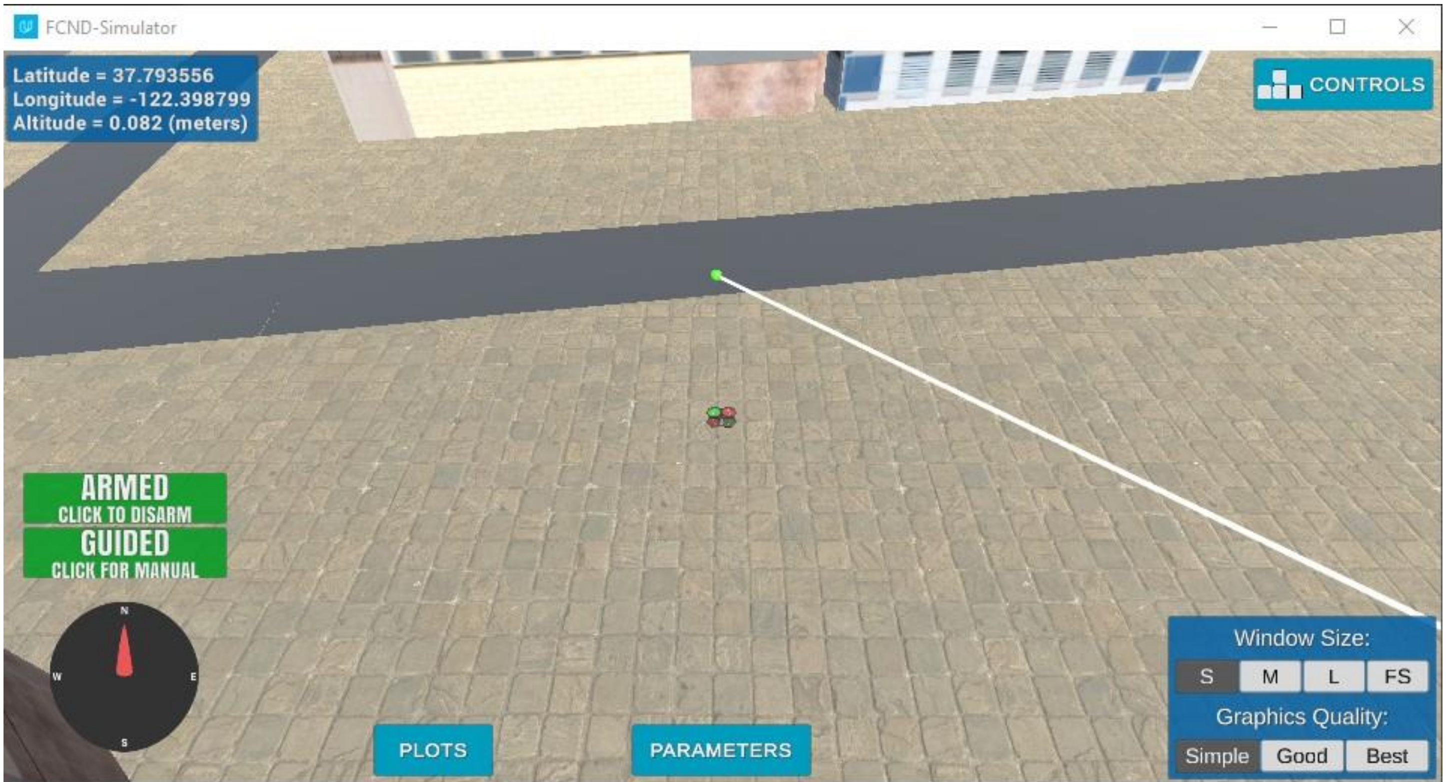
/usr/bin/bash --login -i

```
sampling points ...
[(111, -135, 5), (140, -174, 5), (163, -105, 5), (88, 52, 5), (152, -148, 5), (150, -59, 5),
(151, -34, 5), (62, -99, 5), (128, -153, 5), (168, 25, 5), (91, -11, 5), (69, -149, 5), (88,
-84, 5), (176, -136, 5), (82, -35, 5), (101, 16, 5), (91, -111, 5), (65, -53, 5), (179, -67,
5), (88, -72, 5)]
[(0, 0, 5), (120, -120, 5), (111, -135, 5), (150, -59, 5), (151, -34, 5), (128, -153, 5), (16
8, 25, 5), (91, -11, 5), (69, -149, 5), (88, -84, 5)]
Creating graph ...
Number of edges 9
Finding best route ...
Found a path.
Number of nodes in the Path: 6
Path Cost:  295.9566694249131
[[0, 0, 5, 0], [91, -11, 5, 0], [151, -34, 5, 0], [150, -59, 5, 0], [88, -84, 5, 0], [120, -1
20, 5, 0]]
Sending waypoints to simulator ...
takeoff transition
this may take a few seconds ...
waypoint transition
target position [0, 0, 5, 0]
waypoint transition
target position [91, -11, 5, 0]
waypoint transition
target position [151, -34, 5, 0]
waypoint transition
target position [150, -59, 5, 0]
```



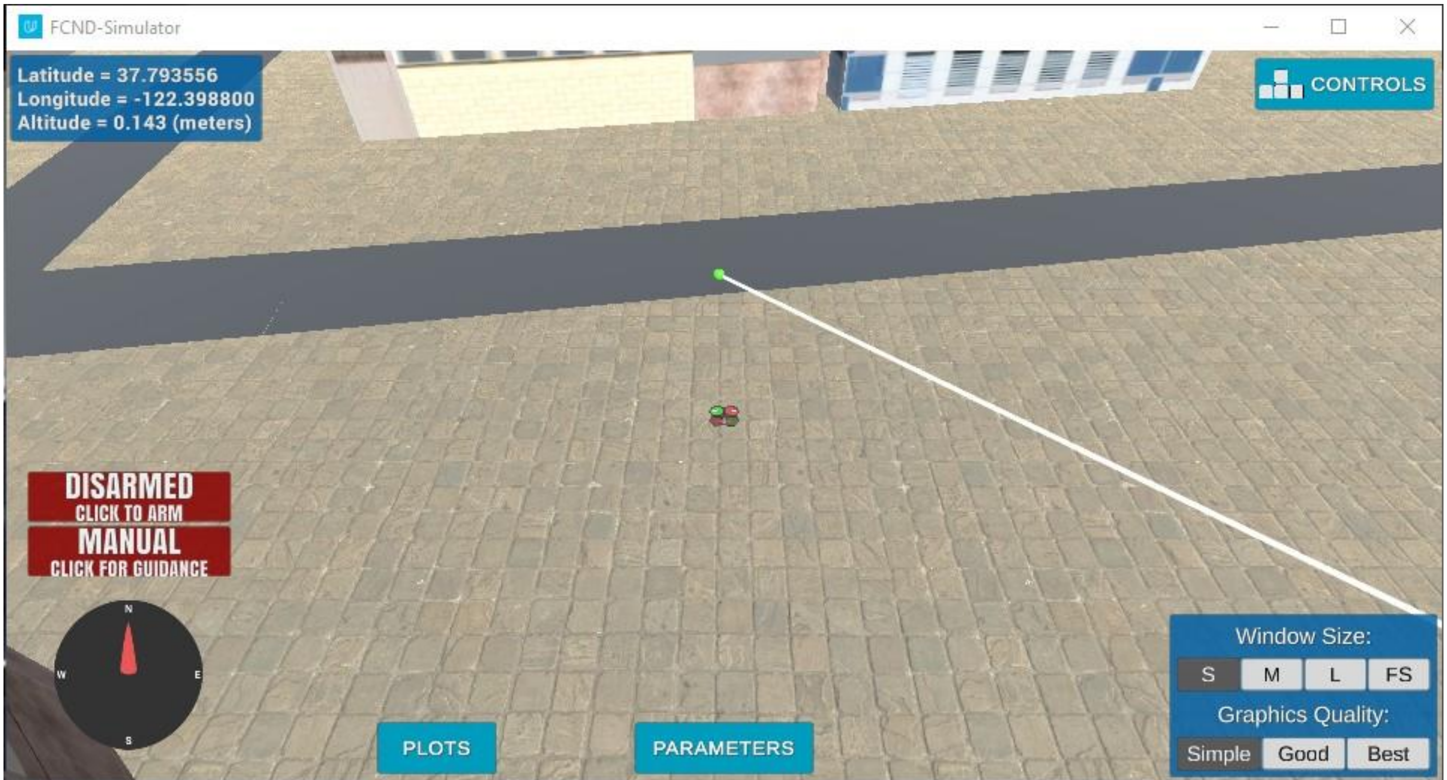
/usr/bin/bash --login -i

```
5), (88, -72, 5)]
[(0, 0, 5), (120, -120, 5), (111, -135, 5), (150, -59, 5), (151, -34, 5), (128, -153, 5), (16
8, 25, 5), (91, -11, 5), (69, -149, 5), (88, -84, 5)]
Creating graph ...
Number of edges 9
Finding best route ...
Found a path.
Number of nodes in the Path: 6
Path Cost:  295.9566694249131
[[0, 0, 5, 0], [91, -11, 5, 0], [151, -34, 5, 0], [150, -59, 5, 0], [88, -84, 5, 0], [120, -1
20, 5, 0]]
Sending waypoints to simulator ...
takeoff transition
this may take a few seconds ...
waypoint transition
target position [0, 0, 5, 0]
waypoint transition
target position [91, -11, 5, 0]
waypoint transition
target position [151, -34, 5, 0]
waypoint transition
target position [150, -59, 5, 0]
waypoint transition
target position [88, -84, 5, 0]
waypoint transition
target position [120, -120, 5, 0]
```



/usr/bin/bash --login -i

```
[(0, 0, 5), (120, -120, 5), (111, -135, 5), (150, -59, 5), (151, -34, 5), (128, -153, 5), (16
8, 25, 5), (91, -11, 5), (69, -149, 5), (88, -84, 5)]
Creating graph ...
Number of edges 9
Finding best route ...
Found a path.
Number of nodes in the Path: 6
Path Cost:  295.9566694249131
[[0, 0, 5, 0], [91, -11, 5, 0], [151, -34, 5, 0], [150, -59, 5, 0], [88, -84, 5, 0], [120, -1
20, 5, 0]]
Sending waypoints to simulator ...
takeoff transition
this may take a few seconds ...
waypoint transition
target position [0, 0, 5, 0]
waypoint transition
target position [91, -11, 5, 0]
waypoint transition
target position [151, -34, 5, 0]
waypoint transition
target position [150, -59, 5, 0]
waypoint transition
target position [88, -84, 5, 0]
waypoint transition
target position [120, -120, 5, 0]
landing transition
```



/usr/bin/bash --login -i

```
Finding best route ...
Found a path.
Number of nodes in the Path: 6
Path Cost:  295.9566694249131
[[0, 0, 5, 0], [91, -11, 5, 0], [151, -34, 5, 0], [150, -59, 5, 0], [88, -84, 5, 0], [120, -1
20, 5, 0]]
Sending waypoints to simulator ...
takeoff transition
this may take a few seconds ...
waypoint transition
target position [0, 0, 5, 0]
waypoint transition
target position [91, -11, 5, 0]
waypoint transition
target position [151, -34, 5, 0]
waypoint transition
target position [150, -59, 5, 0]
waypoint transition
target position [88, -84, 5, 0]
waypoint transition
target position [120, -120, 5, 0]
landing transition
disarm transition
manual transition
Closing connection ...
(fcnd) Omar (master *) FCND-Motion-Planning
$
```