Project: 3D Motion Planning by Robert Geister

Required Steps for a Passing Submission:

1. Load the 2.5D map in the colliders.csv file describing the environment.

2. Discretize the environment into a grid or graph representation.

3. Define the start and goal locations.

4. Perform a search using A or other search algorithm.*

5. Use a collinearity test or ray tracing method (like Bresenham) to remove unnecessary waypoints.

6. Return waypoints in local ECEF coordinates (format for `self.all_waypoints` is [N, E, altitude, heading], where the drone's start location corresponds to [0, 0, 0, 0].

7. Write it up.

8. Congratulations! Your Done!

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

Writeup / README

1. Provide a Writeup / README that includes all the rubric points and how you addressed each one. You can submit your writeup as markdown or pdf.

You're reading it! Below I describe how I addressed each rubric point and where in my code each point is handled.

Explain the Starter Code

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

These scripts contain a basic planning implementation that is invoked in the "plan_path" function. It contains the definition of a safety distance to obstacles and a target altitude. Based on that, the data from colliders.csv is read and a grid is created that contains the obstacles. The starting point in the grid is set to the initial position of the simulator (with is inside a building which causes some problems). The goal is set to a position 10m northeast from the start.

After that an implementation of an A* algorithm is invoked that calculated a path from start to goal based on movements up/down or left/right. The heuristic used for this A* implementation is the Euclidian distance between the current point and the goal.

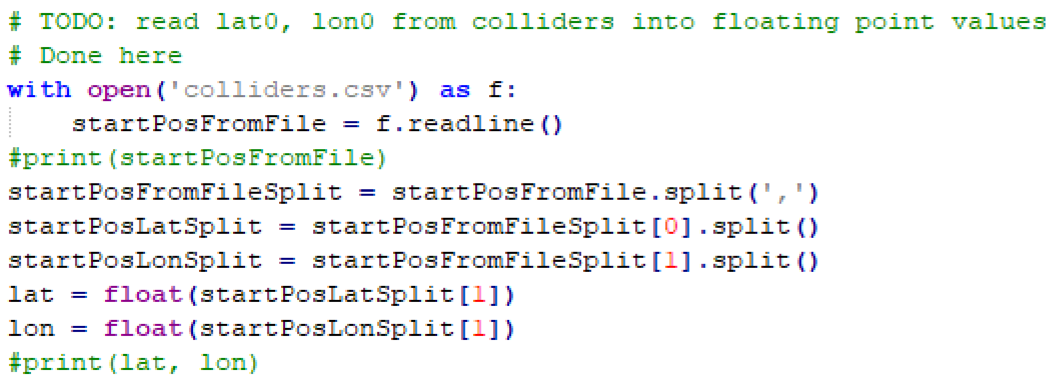
Implementing Your Path Planning Algorithm

I implemented my solution in the file motion_planning_sol_rg_outside.py and extended the file planning_utils.py I used a combination of a grid based A* and a graph based A* with a probabilistic roadmap.

1. Set your global home position

Here 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 did this is lines 130 – 137. I opened the file and read the first line with f.readline(). Afterwards I used the "split" method to separate the values from each other and the text. Then, I converted the string into float with float().



```
# TODO: read lat0, lon0 from colliders into floating point values
# Done here
with open('colliders.csv') as f:
    startPosFromFile = f.readline()
#print(startPosFromFile)
startPosFromFileSplit = startPosFromFile.split(',')
startPosLatSplit = startPosFromFileSplit[0].split()
startPosLonSplit = startPosFromFileSplit[1].split()
lat = float(startPosLatSplit[1])
lon = float(startPosLonSplit[1])
#print(lat, lon)
```

Figure 1: Code used

Then I used the values to set the home position with "self.set_home_position(lon, lat, 0.0)" in line 142. Interestingly, if you print global home afterwards, it will not show the updated global home until the function plan_path is complete.

2. Set your current local position

Here as long as you successfully determine your local position relative to global home you'll be all set. Explain briefly how you accomplished this in your code.

In order to do this, I retrieved the current global position with "global_pos_current = [self.global_position[0], self.global_position[1], self.global_position[2]]" and converted it to the current local position with "local_pos_current = global_to_local(global_pos_current, self.global_home)".

3. Set grid start position from local position

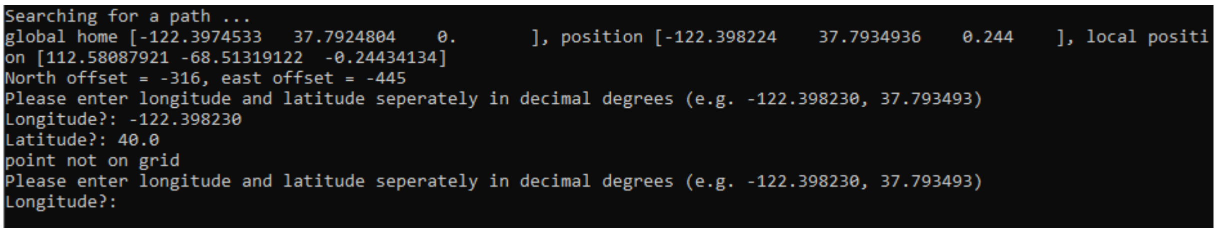
This is another step in adding flexibility to the start location. As long as it works you're good to go!

I did this in line 164 with "grid_start = (int(np.around(self.local_position[0]))-north_offset, int(np.around(self.local_position[1]))-east_offset)". In order to workj with the grid, the position needs to be an integer value. That is why I rounded the current local position to the nearest integer value.

4. Set grid goal position from geodetic coords

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.

I realized this twofold. In the current implementation, the user is prompted to enter lon/lat values for the goal. The entered values are being assessed if they are allowable float values that lie on the grid. On the one hand, I used a function "check_string_to_float" to check if the entered string is a float value. On the other hand, I converted the entered position to local coordinated and checked if they lie on the grid in line 202. Additionally, the location is checked for obstacles. In order to do this, I created a second grid with an altitude of "0" and the safety distance defined. That grid is used to check if the entered position in in an obstacle. If the entered goal location is valid, the goal location is set. If not, the user is prompted again. This part can be commented out and a goal location can manually be set in line 219.



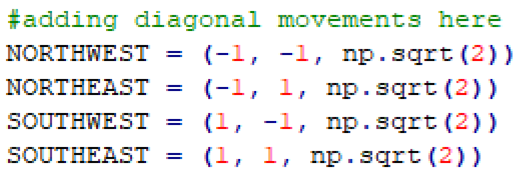
```
Searching for a path ...
global home [-122.3974533   37.7924804    0.       ], position [-122.398224    37.7934936    0.244    ], local positi
on [112.58087921 -68.51319122  -0.24434134]
North offset = -316, east offset = -445
Please enter longitude and latitude seperately in decimal degrees (e.g. -122.398230, 37.793493)
Longitude?: -122.398230
Latitude?: 40.0
point not on grid
Please enter longitude and latitude seperately in decimal degrees (e.g. -122.398230, 37.793493)
Longitude?:
```

Figure 2: Rejected goal location

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

Minimal requirement here is to 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 accomplished that in the planning_utils.py file in lines 112 – 116. I extended the "Action" class with the possible actions northwest, northeast, southwest and southeast. The actions are carried out in the grid with two movements, one in the north/south direction and one in the east/west direction. These four movements have a cost of sqrt(2).



```
#adding diagonal movements here
NORTHWEST = (-1, -1, np.sqrt(2))
NORTHEAST = (-1, 1, np.sqrt(2))
SOUTHWEST = (1, -1, np.sqrt(2))
SOUTHEAST = (1, 1, np.sqrt(2))
```

Figure 3: Diagonal movements

Furthermore, I implemented the function "a_star_graph" in the file planning_utils.py to be able to use a NetworkX graph to find a path with A*. The heuristic as well as the costs are implemented as Euclidian distances. In order to use A* on a graph, I had to create a graph first.

I did this in my motion planning file. I put the code for it in the "main" function as it runs a lot faster there, before making the connection to the simulator. My graph is based on a probabilistic roadmap. In lines 412 – 415, the variable values of drone altitude, safety distance as well as the number of samples for the random graph and the number of neighbors for the possible edges can be set. I used

250 samples and 10 neighbors with an altitude of 25m and a safety distance of 7m. The figure below shows two different graphs with different altitudes (left 20m, right 90m). The green dots show samples that were rejected as they were within obstacles.

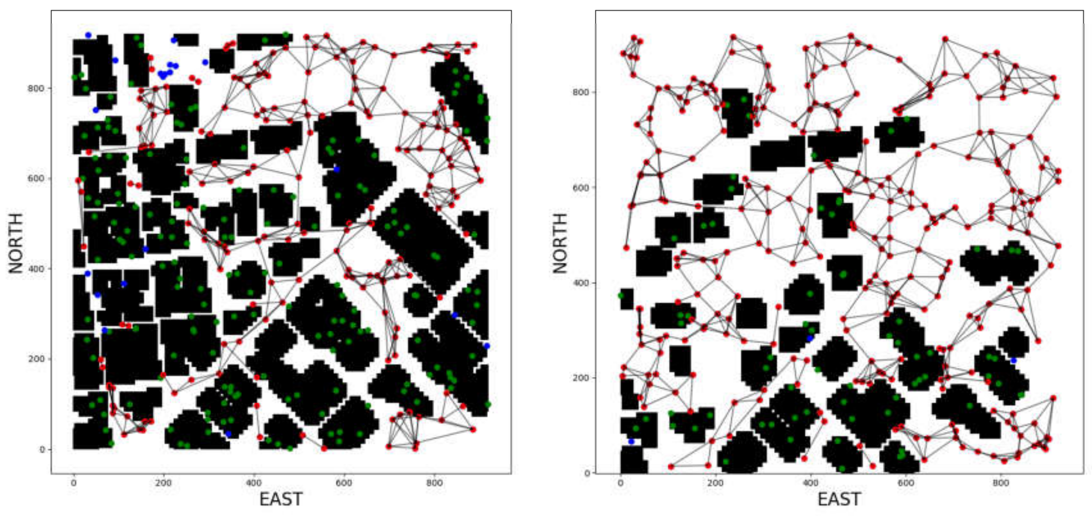


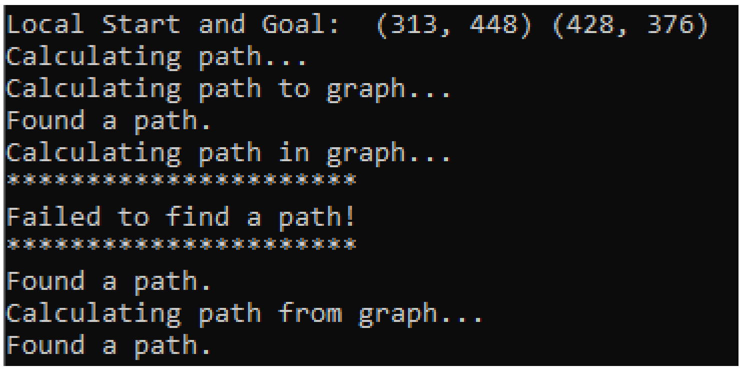
Figure 4: Random graphs with different altitudes

In order to crate the graph, I created random samples based on the size of the grid (lines 434 – 438). I used the drone altitude for the z values in order to reduce complexity and use the 2.5D representation. Therefore, changes in altitude during the flight are not envisaged.

Based on the samples, I check each point if it is within an obstacle with the “collides” function implemented in planning_utils.py. The obstacle polygons were created when creating the grid with the shapely package. Each point is checked against all polygons with the “contains” function. I did not use a tree search here in order to check all polygons.

Afterwards, I used the create_graph function in my planning_utils.py file to create a NetworkX graph. Based on the number of neighbors set above, it tries to connect the nodes with the can_connect function. Within that function a line string is crated between two nodes and it is checked if it crosses a polygon. If not, it is a valid edge. Afterwards I create a search tree in line 454 with the nodes of the graph.

With all that, the path is planned in the following way: With the help of the search tree, the closest node in the graph is selected with “distStart, indexStart = mytree.query(pointStart)”. The problem is that the graph is random and we cannot know how far the actual starting point is away from the closest point of the graph and if obstacles are in between. Therefore, I am using the grid based A* to get from the starting location to the closest point on the graph. From there I use the graph based A* to get to the goal. To get there, I again use the search tree to get the closest node from the goal location. However, I cannot know if the closest point on the graph is connected to the start location of the graph. Therefore, I use the search tree to get the 3 nearest neighbors “distGoal, indexGoal = mytree.query(pointGoal, 3)”. Of course this number can be changed. I then try to find a path with A* to the closest neighbor. If that fails, I use the second closest and so on. If all that should fail, I use the grid based A* all the way. Again. The drone has to move from the closest connected node to the goal again and for that part I use the grid based A* again.



```
Local Start and Goal:  (313, 448) (428, 376)
Calculating path...
Calculating path to graph...
Found a path.
Calculating path in graph...
**********************
Failed to find a path!
**********************
Found a path.
Calculating path from graph...
Found a path.
```

Figure 5: Output of the search

The figure above shows how the search with the closest neighbor to the goal fails but the second closest is connected to the start location.

6. Cull waypoints

For this step 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.

After the partial paths are connected, the waypoints are pruned with the prune_path function that checks of three points on the path a collinear with the determinant method. From that path, the waypoints are created and sent to the simulator. In order to display the path in the sim, I had to round the graph points to integer values. Otherwise the simulator got stuck.

Execute the flight

1. Does it work?

It works! The only problem is that the initial position of the simulator is within a building that should not be there. So the drone collides during the start. I can be manually flown out with a collision, and afterwards it works. Due to the random nature of the graph, the paths can be a detour compared to grid based A*. But for longer paths, the computation time reduces significantly with the use of the graph. The figures below show an example flight.

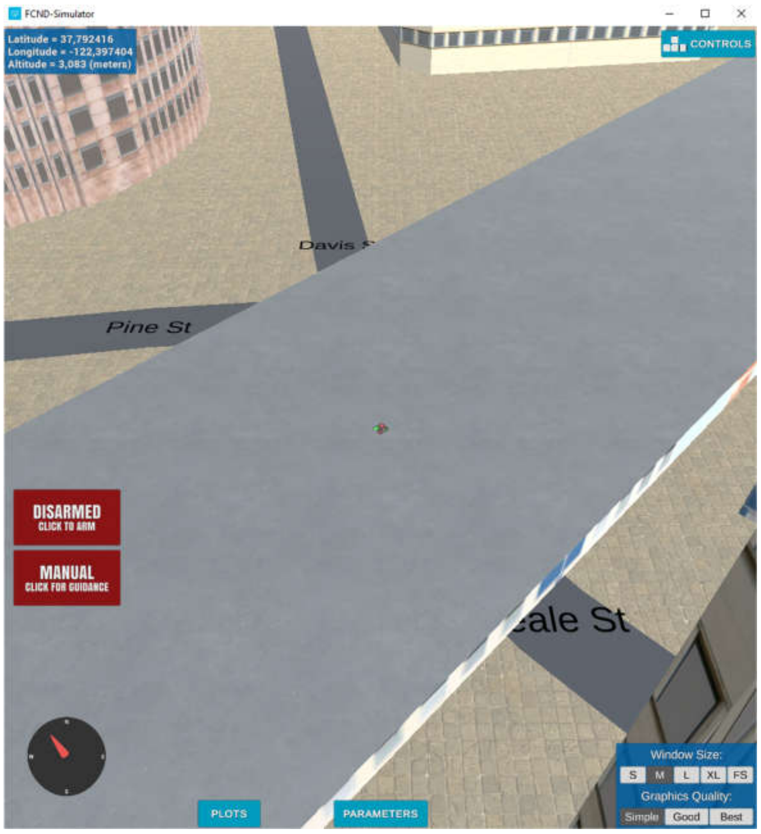


Figure 6: Drone start location

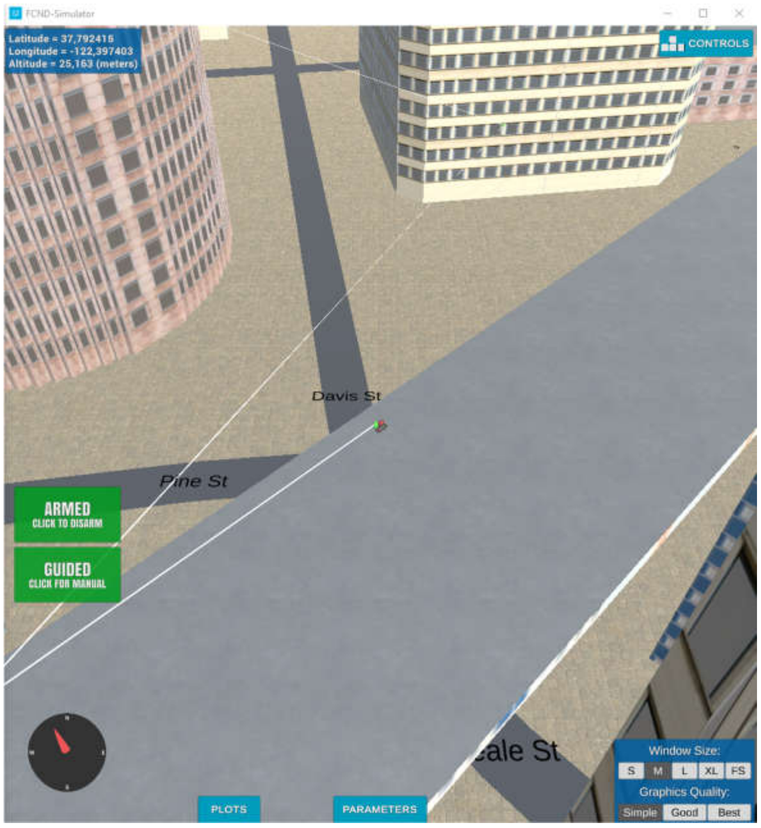


Figure 7: Execution of the flight with path

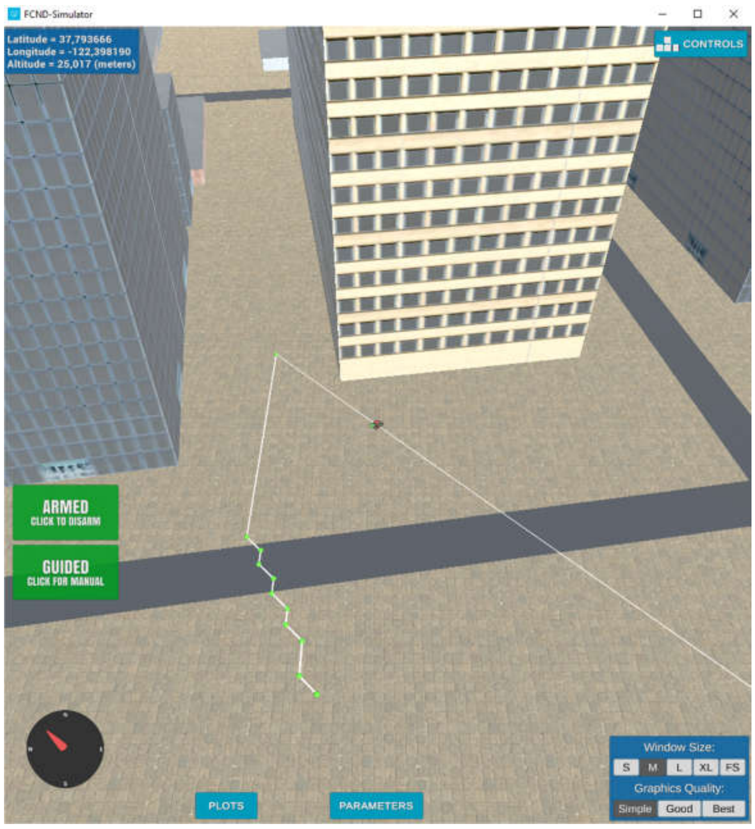


Figure 8: Arrival at goal location with transition from graph to grid

Extra Challenges: Real World Planning

I wanted to use these methods and also to create a graph AFTER the start and goal locations are known to reduce to effort, but unfortunately when connected to the simulator, the execution of the code is really slow so I could not use it.