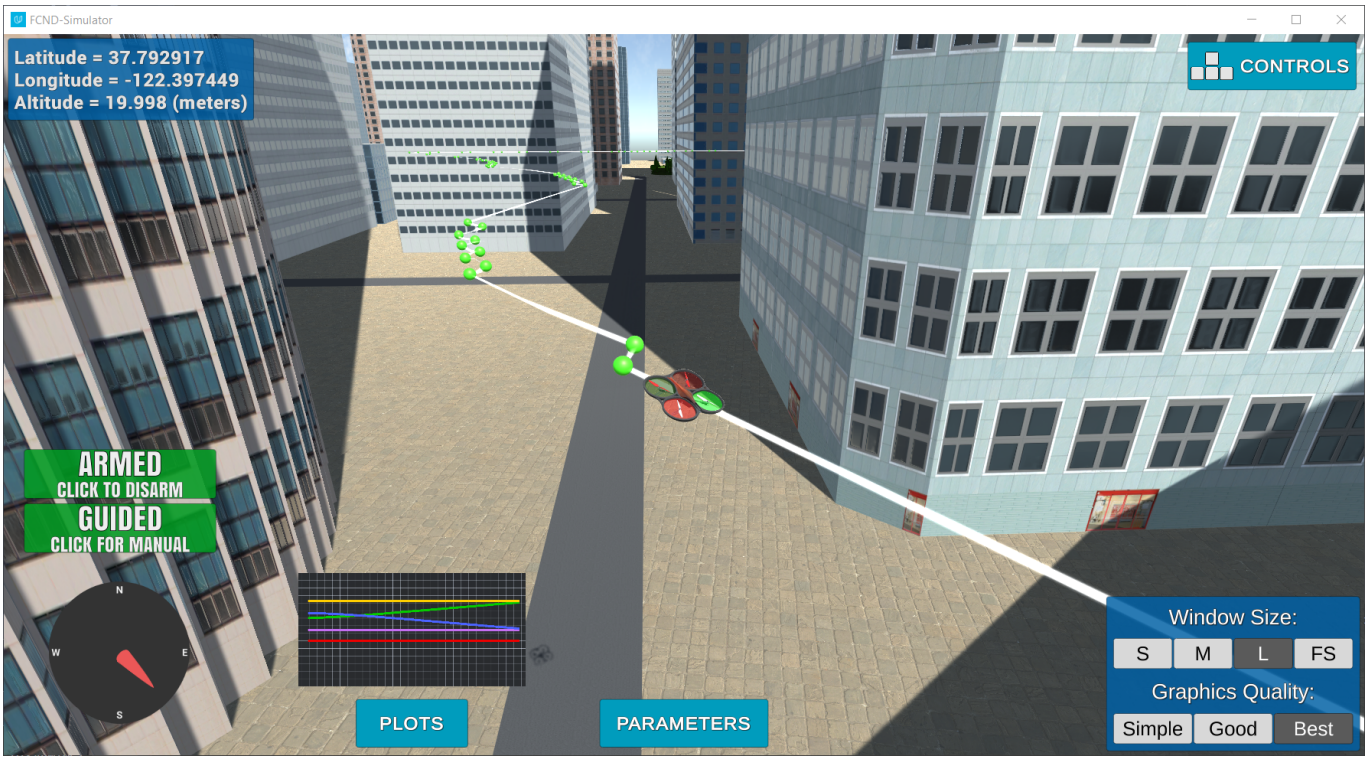
Project 2: 3D Motion Planning

Chris Dalke



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!

Usage Instructions

To set the goal position, I defined several command-line arguments:

- `--goal_lat=...` Set the goal latitude.
- `--goal_lon=...` Set the goal longitude.
- `--goal_alt=...` Set the goal altitude.

When testing, I ran the program with the following goal:

```
python motion_planning.py --goal_lat=37.793732 --goal_lon=-122.396188 --
goal_alt=20.0
```

Writeup

1. Explain the Starter Code

The starter code in `motion_planning.py` and `planning_utils.py` builds on the framework from Backyard Flyer to set up path planning functionality. Similarly to the Backyard Flyer, the code runs through a state machine which arms & executes takeoff, waypoints, and landing.

The code uses a more sophisticated algorithm to generate waypoints than Backyard Flyer. In Backyard Flyer, I generated waypoints directly as ECEF frame positions, forming a simple shape that did not take into account any obstacles or geodetic positions.

In motion planning, the waypoints are generated based on a path planning algorithm. The waypoints are generated when the drone is armed, and step through the following logic:

- Load up the collider definition from a file, building a 2D grid from all obstacles at a particular "slice" of height given by `TARGET_ALTITUDE`.
- Set a grid start and goal position. In the starter code, this is the grid center and a slight offset.
- Run the A* path planning algorithm, navigating the obstacles to reach the target position.
- Using the result of the A* algorithm, generate a list of waypoints which the drone framework will execute.

In the finished projects, there will be several modifications needed to the starter code. For example, the starter code does not use geodetic positions for home/goal positions. Additionally, the starting code will need the A* algorithm improved to allow diagonal paths.

2. Implementing the Path Planning Algorithm

2.1 Read Global Home Position

The first step to setting up the environment is to obtain the global home coordinates, which will be used as the origin of the local frame.

The first line of `colliders.csv` contains the global home as lat/lng coordinates. I read the coordinates with a regex that matches `lat0 <LATITUDE> lon0 <LONGITUDE>`:

```
collidersFile = open('colliders.csv', 'r')
rawHomePosLine = collidersFile.readline()
homePosMatch = re.match("^lat0 ([-+]?[0-9]*\.?[0-9]+), lon0 ([-+]?[0-9]*\.?[0-
9]+)$", rawHomePosLine)
```

Once the global home coordinates are obtained, they're set as the home position for the drone.

2.2 Read Local Position Relative to Global Home

Converting any global home position to the local frame can be done with the `global_to_local` method. For example, to convert the drone's current global position into a local position:

```
local_start_pos = global_to_local(self.global_position, self.global_home)
```

2.3 Start at Current Local Position

I obtained the local start position above. I use this as the input to the graph-search algorithm. Because I'm using a grid-based search with the Medial-Axis algorithm, the start/end points are not necessarily available. I utilized the code from the Medial Axis exercise to clamp the positions to a grid coordinate that is unblocked.

2.4 Set Goal to Arbitrary Global Position

In my implementation, the goal is specified via the command-line arguments `goal_lat`, `goal_lon`, and `goal_alt`. Similarly to the start position, the goal position is clamped to the nearest unblocked grid position.

2.5 Write Search Algorithm

For my search algorithm, I chose to use grid-based A* with minimal modifications besides adding the ability to travel along diagonals. To allow A* to travel along diagonals, you add a new set of actions for the diagonals:

```
NORTH_WEST = (-1, -1, math.sqrt(2))
NORTH_EAST = (-1, 1, math.sqrt(2))
SOUTH_WEST = (1, -1, math.sqrt(2))
SOUTH_EAST = (1, 1, math.sqrt(2))
```

and a corresponding set of validations:

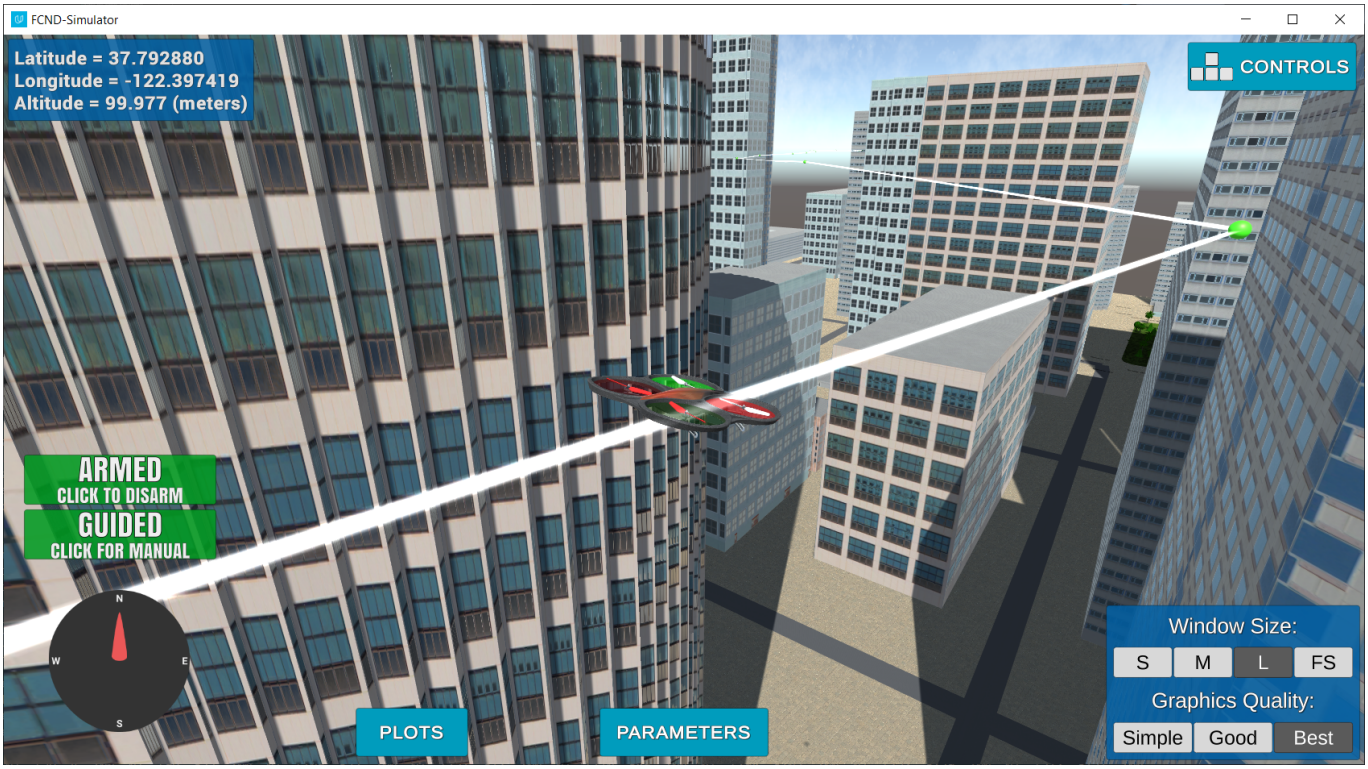
```
if x - 1 < 0 or y - 1 < 0 or grid[x - 1, y - 1] == 1:
    valid_actions.remove(Action.NORTH_WEST)
if x - 1 < 0 or y + 1 > m or grid[x - 1, y + 1] == 1:
    valid_actions.remove(Action.NORTH_EAST)
if x + 1 > n or y - 1 < 0 or grid[x + 1, y - 1] == 1:
    valid_actions.remove(Action.SOUTH_WEST)
if x + 1 > n or y + 1 > m or grid[x + 1, y + 1] == 1:
    valid_actions.remove(Action.SOUTH_EAST)
```

I use the Medial-Axis algorithm to efficiently obtain a list of positions at the center of the free space, reducing the state space significantly for search. I call `skimage.morphology`'s implementation.

2.6 Prune Unnecessary Waypoints

I used the colinearity test to prune unnecessary waypoints, using the same implementation from the Colinearity exercise.

3. Executing the Flight



It flies! I've tested the simulation with several positions/altitudes.

The screenshots above are actually from an earlier version of my solution before I switched to using the medial axis algorithm. I implemented an initial solution using graph-based search and random sampling, but found that the number of samples required to build a complete map was a performance issue.

Extra Challenges

Heading Waypoints

The heading of each waypoint is calculated based on the angle between the previous and current waypoint. My algorithm computes the heading for each waypoint and adds that information before sending to the simulator.

Other Notes / simulator issues

I was able to successfully execute the path planner, but I had several issues with the simulator that are worth noting. In many cases, my solution fails because of one of the issues below:

1. You start inside a building, and exiting the building flings the quad upwards (It recovers, but takes a while)
2. If the algorithm takes too long to compute a path, the connection will time out.
3. In some cases, I found the coordinates being reported by the GPS simulation to be off significantly (By 10+ meters). This got in the way of the simulation, because it meant that the drone would stop for a long period of time at one waypoint. This may be intentional to simulate GPS noise, but it was hard to tell. I tested changing the deadband to 5+ meters, but always hit this issue regardless of how large I set the deadband.