## Project: 3D Motion Planning

![Quad Image](./misc/enroute.png)

---

# Required Steps for a Passing Submission:

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

The csv file consistis of 6 variables [North, East, Alt, d\_north, d\_East, d\_alt]

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!

## [Rubric](https://review.udacity.com/#!/rubrics/1534/view) Points

### 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`

The planning\_utils.py script has 3 main helper functions and a class:

1. Create\_grid(data, drone\_altitude, safety\_distance): and outputs a grid that reflectes the environment in realtion of how high the drone is flying, and the sfatey margin assigned.
2. valid\_actions(grid, current\_node): returns valid actions, given a position. Valid actions are the ones that are in the grid and does not collide with an obstacle.
3. A\_star(grid, heuristic, start, goal): returns a path to the goal, and the cost of taking such path, by appliying the given heuritsc of reaching the goal.
4. Class Action(Enum): are 3 values in tuple form, the first 2 numbers are the difference it makes when taking this particular action, and the 3rd number is the associated cost of taking such action. This class provides a 2 functions that can return the delta and the cost of an action.

The motion\_planning.py script executes the whole plan. It has 2 classes, the States(Enum) class, which has 7 main states and it should be executed in this order:

1. MANUAL
2. ARMING
3. PLANNING
4. TAKEOFF
5. WAYPOINT
6. LANDING
7. DISARMING

Also we have the MotionPlanning(Drone) class, which connects to the drone and has all the major functions. We have the callback functions, which are the makes the drone aware of its status it in three different ways; in terms of velocity, local postion and state, this data is constantly sent from the callback registery in the class’s constructor. through this awareness, it can decide which step to take next by sending it to the transition functions. We have the 7 transition functions, which sends the commands to the Udacity Drone API, to take the command in the simulator.

And here's a lovely image of my results (ok this image has nothing to do with it, but it's a nice example of how to include images in your writeup!)

![Top Down View](./misc/high\_up.png)

Here's | A | Snappy | Table

--- | --- | --- | ---

1 | `highlight` | \*\*bold\*\* | 7.41

2 | a | b | c

3 | \*italic\* | text | 403

4 | 2 | 3 | abcd

### Implementing Your Path Planning Algorithm

#### 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 utilized the csv module to open the file then I took the first row and used split by whitespace to take the second element which was the numbers of the string. Then I used the float function to convert from a string to a float.

**with** open('colliders.csv', newline='') **as** f:

reader = csv.reader(f)

row1 = next(reader)

self.lat0 = float(row1[**0**].split(' ')[**1**])

self.lon0 = float(row1[**1**].split(' ')[**2**])

And here is a lovely picture of our downtown San Francisco environment from above!

![Map of SF](./misc/map.png)

#### 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.

I got my local position by retrieving the live (lon,lat,alt) data from drone.py. Then I used global\_to\_local function to transform the geodic position to a local format.

#TODO: retrieve current global

self.current\_global = ([self.\_longitude, self.\_latitude, self.\_altitude])

**print**(self.current\_global)

# TODO: convert to current local position using global\_to\_local()

self.local\_relative\_to\_global = global\_to\_local(self.current\_global, self.global\_home)

Meanwhile, here's a picture of me flying through the trees!

![Forest Flying](./misc/in\_the\_trees.png)

#### 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!

#### 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.

#### 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.

Because the drone did not a diagonal dimension/action option in the Action class of planning\_util.py , it only knew how to flew the basic four directions (North, East, South, West). I added diagonal actions with their respective cost of squareroot(2), which is a higher cost than going in a basic direction which is 1. As shown below:

NE = (-**1**, **1**, np.sqrt(**2**))

SE = (**1**, **1**, np.sqrt(**2**))

NW = (-**1**, -**1**, np.sqrt(**2**))

SW = (**1**, -**1**, np.sqrt(**2**))

#### 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.

For culling waypoints, I have used the collinearity test.

I have borrowed the prune\_path(path) function from the exercises and its implementation has been successful as shown in Figure 1 below:

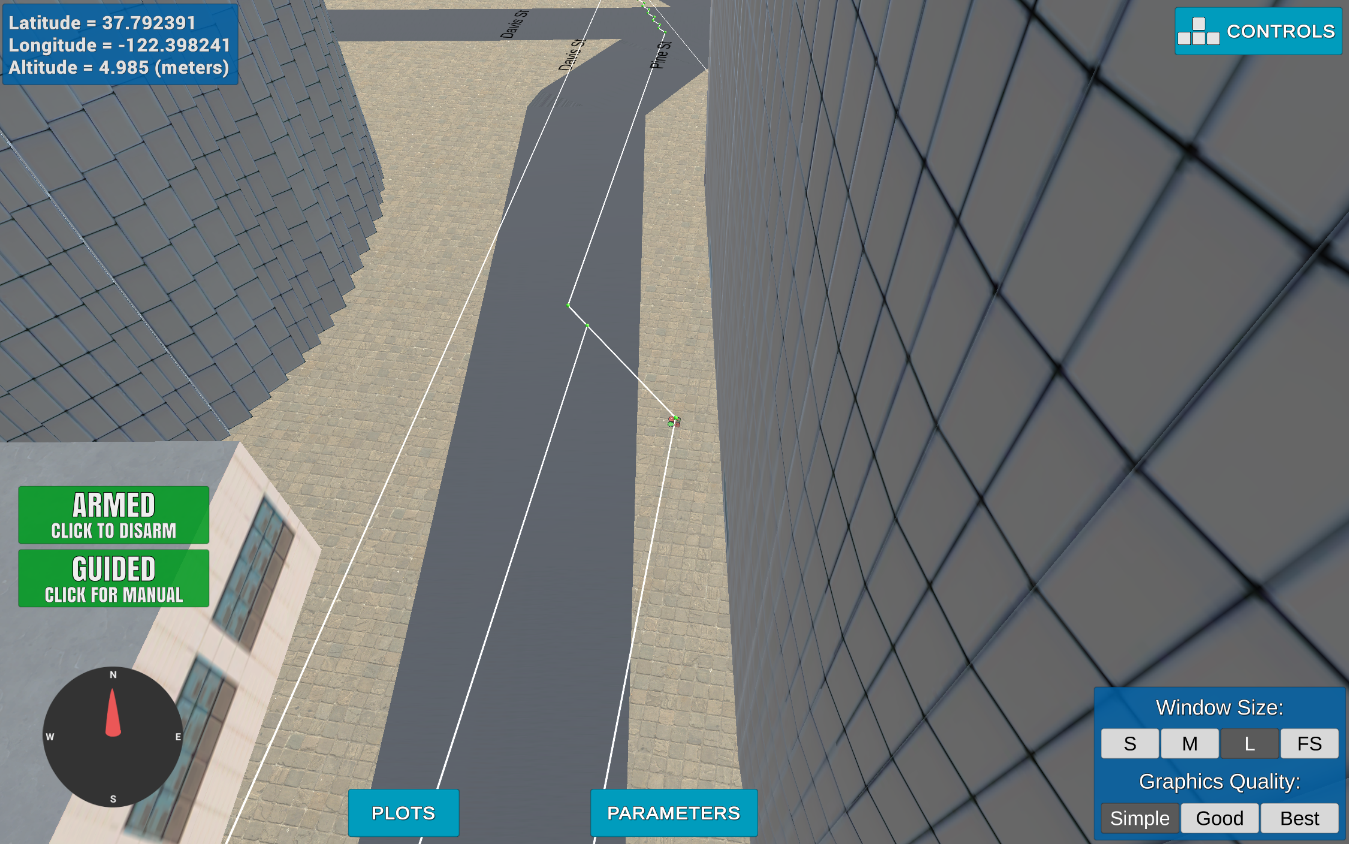


Figure : After collinearity test

**def** **point**(p):

**return** np.array([p[**0**], p[**1**], **1.**]).reshape(**1**, -**1**)

**def** **collinearity\_check**(p1, p2, p3, epsilon=**1e-6**):

m = np.concatenate((p1, p2, p3), **0**)

det = np.linalg.det(m)

**return** abs(det) < epsilon

**def** **prune\_path**(path):

pruned\_path = [p **for** p **in** path]

# TODO: prune the path!

i = **0**

**while** i < len(pruned\_path) - **2**:

p1 = point(pruned\_path[i])

p2 = point(pruned\_path[i+**1**])

p3 = point(pruned\_path[i+**2**])

# If the 3 points are in a line remove

# the 2nd point.

# The 3rd point now becomes and 2nd point

# and the check is redone with a new third point

# on the next iteration.

**if** collinearity\_check(p1, p2, p3):

# Something subtle here but we can mutate

# `pruned\_path` freely because the length

# of the list is check on every iteration.

pruned\_path.remove(pruned\_path[i+**1**])

**else**:

i += **1**

**return** pruned\_path

### Execute the flight

#### 1. Does it work?

It works!

### Double check that you've met specifications for each of the [rubric](https://review.udacity.com/#!/rubrics/1534/view) points.

# Extra Challenges: Real World Planning

For an extra challenge, consider implementing some of the techniques described in the "Real World Planning" lesson. You could try implementing a vehicle model to take dynamic constraints into account, or implement a replanning method to invoke if you get off course or encounter unexpected obstacles.