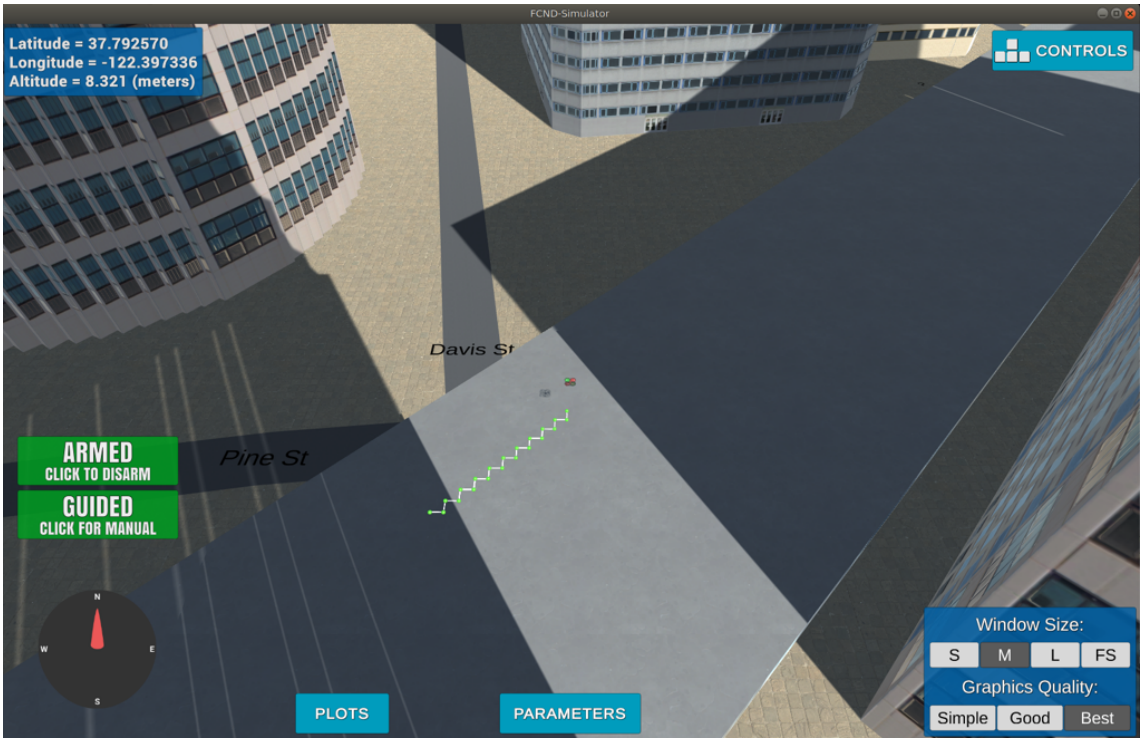
Project 2: 3D Motion Planning

This project aims to design a motion planner for a 6DOF drone model in a simulated urban environment. Various topics like coordinate transformation, collision detection algorithms, A* path finding algorithm, and path pruning are covered in the scope of this project.

Tasks

1. Test that motion_planning.py is a modified version of backyard_flyer_solution.py for simple path planning. Verify that both scripts work. Then, compare them side by side and describe in words how each of the modifications implemented in motion_planning.py is functioning.

   backyard_flyer_solution.py follows waypoints that are manually entered by the user. On the other hand, motion_planning.py just takes start and goal positions and generates waypoints. These waypoints should not collide with buildings, so 'colliders.csv' file is used to import building locations and *create_grid* function gives free zones to flight. A* algorithm takes start and goal position, and grid map then gives waypoints. Below a picture of simulation is given.



2. Modify your code to read the global home location from the first line of the colliders.csv file and set that position as global home (self.set_home_position())

   Code snippet is given below.

```
# TODO: read lat0, lon0 from colliders into floating point values
filename = 'colliders.csv'
f = open(filename, "r")
lat_str, lon_str = f.readline().split(',')
lat0 = float(lat_str.split()[1])
lon0 = float(lon_str.split()[1])
f.close()
# TODO: set home position to (lon0, lat0, 0)
self.set_home_position(lon0,lat0,0.0)
```

3. Retrieve your current position in geodetic coordinates from self._latitude, self._longitude and self._altitude. Then use the utility function global_to_local() to convert to local position (using self.global_home as well, which you just set)

```
# TODO: retrieve current global position
global_position = (self._longitude, self._latitude, self._altitude)

# TODO: convert to current local position using global_to_local()
local_position = global_to_local(global_position, self.global_home)
```

4. In the starter code, the start point for planning is hardcoded as map center. Change this to be your current local position.

```
grid_start = (int(local_position[0] - north_offset),
int(local_position[1] - east_offset))
```

5. In the starter code, the goal position is hardcoded as some location 10 m north and 10 m east of map center. Modify this to be set as some arbitrary position on the grid given any geodetic coordinates (latitude, longitude)

```
# The goal position will be selected randomly.It should be confirmed
that
# this position is not occupied by any buildings.
valid_goal = False
print('Random goal search started...')

while not valid_goal:

   goal_n = np.random.randint(north_offset, north_max)
   goal_e = np.random.randint(east_offset, east_max)
   grid_goal = (-north_offset + goal_n, -east_offset + goal_e)
   print('Searching ...')
   if grid[grid_goal] == 0:
       valid_goal = True
       print('Goal assignment completed.')
```

6. Write your search algorithm. Minimum requirement here is to add diagonal motions to the A* implementation provided, and assign them a cost of sqrt(2).

```
NORTHWEST = (-1, -1, np.sqrt(2))
NORTHEAST = (-1, 1, np.sqrt(2))
SOUTHWEST = (1, -1, np.sqrt(2))
SOUTHEAST = (1, 1, np.sqrt(2))

if x - 1 < 0 or y - 1 < 0 or grid[x - 1, y - 1] == 1:
   valid_actions.remove(Action.NORTHWEST)
if x - 1 < 0 or y + 1 > m or grid[x - 1, y + 1] == 1:
   valid_actions.remove(Action.NORTHEAST)
if x + 1 > n or y - 1 < 0 or grid[x + 1, y - 1] == 1:
   valid_actions.remove(Action.SOUTHWEST)
if x + 1 > n or y + 1 > m or grid[x + 1, y + 1] == 1:
   valid_actions.remove(Action.SOUTHEAST)
```

7. Cull waypoints from the path you determine using search.

   Collinearity test is used to reduce the number of waypoints. Without this reduction, zig-zag waypoints may be obtained. This function calculates the area between 3 consecutive waypoints, if the area is close to zero these points are almost linear. Epsilon value is critical here. Very small numbers like 1e-6 do not help to reduce some “almost linear” waypoints, and some zig-zag waypoints are remained. Epsilon is taken as 10 to obtain smooth lines.

```
def point(p):
   return np.array([p[0], p[1], 1.]).reshape(1, -1)

def collinearity_check(p1, p2, p3, epsilon=10):
   m = np.concatenate((p1, p2, p3), 0)
   det = np.linalg.det(m)
   return abs(det) < epsilon

def path_pruning(path):
   pruned_path = [p for p in 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
```

8. Executing the flight

Multiple scenarios are tried and all accomplished. Some pictures of these scenarios are given below.