3D Motion Planning Project

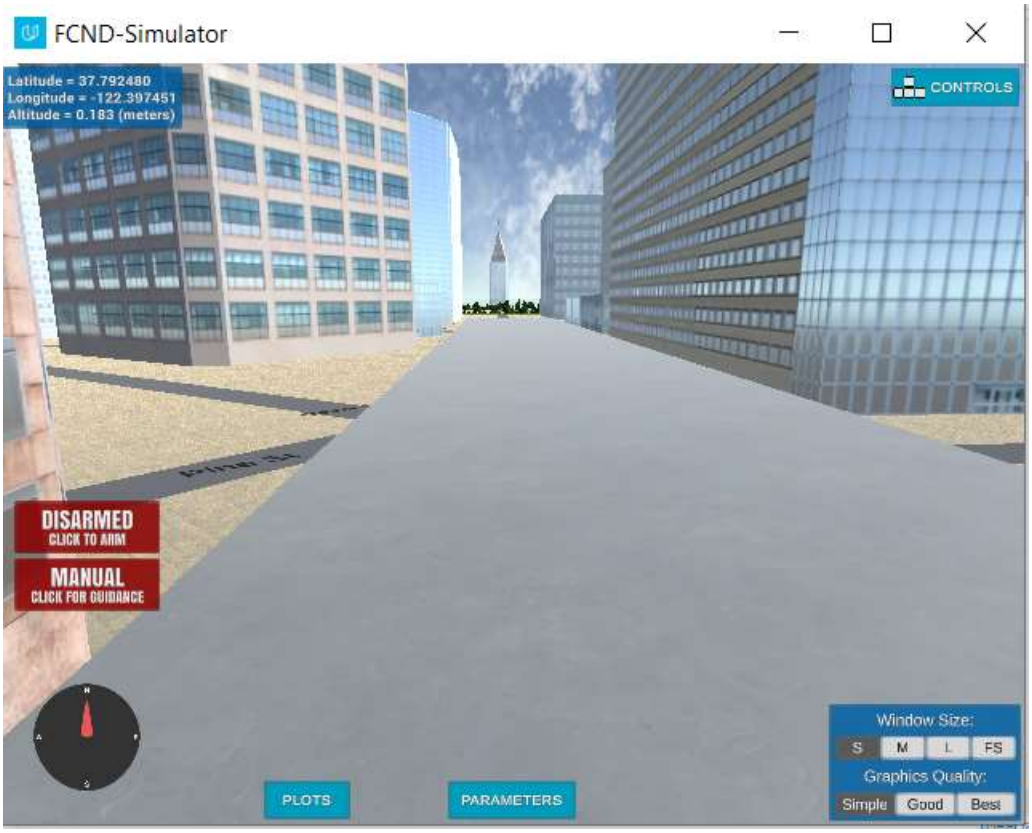
Dhar Rawal dharrawal@gmail.com 281-995-6423 11/22/2020

Udacity – Flying Car and Autonomous Flight Engineer Program

Introduction

The challenge here is to program a drone to be able to navigate autonomously between any two points in a 2.5D simulation of a real city, namely San Francisco. The simulation of the city and the drone is in the Unity environment, and the project will be written in Python.

Image of the city in Unity environment



Starter Code

The starter code consists of 2 files – motion_planning.py and planning_utls.py. The starter code works!

Motion_planning.py

This code is like backyard_flyer_solution.py. However, unlike the backyard flyer, this code does not navigate a predetermined trajectory. It navigates the drone from one point to another in a grid setup using the A* path-planning algorithm. The code is encapsulated in the MotionPlanning class with the following methods:

Methods	Description

start	Start the connection to receive events from the drone
arming_transition disarming_transition	Event handlers to arm (turn on the rotors) and disarm (turn off the rotors) the drone
takeoff_transition landing_transition	Event handlers to takeoff (given an altitude) and land
manual_transition	Event handler to switch the drone to manual control
waypoint_transition	Event handler to move the drone to the next waypoint in the path
send_waypoints	This method send the waypoints to the simulator for visualization purposes
plan_path	This method is the brains of the motion planning. It builds the path to navigate the drone to its destination via a series of waypoints, avoiding obstructions along the way. Basically, it builds a collection of ordered waypoints and sends them to the simulator

Highlighted methods are new as compared to backyard_flyer.py

Planning_utils.py

This is a set of utility functions provided to facilitate the project. We will have to use/modify these utility functions in MotionPlanning->plan_path() as we flesh out our new path planning code. The following functions are provided:

Method	Description
create_grid	Returns a grid representation of a 2D configuration space based on given obstacle data, drone altitude and safety distance
valid_actions	Returns a list of valid actions given a grid and current node. Each action is an object of the class Action that has an associated cost and delta (tuple describing relative movement)
a_star	Basic A* path planning algorithm to compute a set of waypoints given a grid, start and goal
heuristic	Heuristic function for the A* algorithm

Pseudocode for MotionPlanning->plan_path()

1. Initialize target altitude and safety distance
2. Read in the obstacle map
3. Create the grid based on the target altitude and safety distance
4. Define arbitrary starting node and goal node
5. Run the A* method to get back a path from start to goal, given the grid of obstacles and a heuristic function
6. Convert the path to a waypoint's matrix in the SIM world

7. Display the waypoints in the simulator
8. Transition the drone to takeoff and navigate the planned path of waypoints to the goal

Setting home position

The home position lat/lon are read from the first line of colliders.csv. This home position is then set in the **plan_path** function via the below line:

self.set_home_position(self.home_lon, self.home_lat, 0)

Current local position

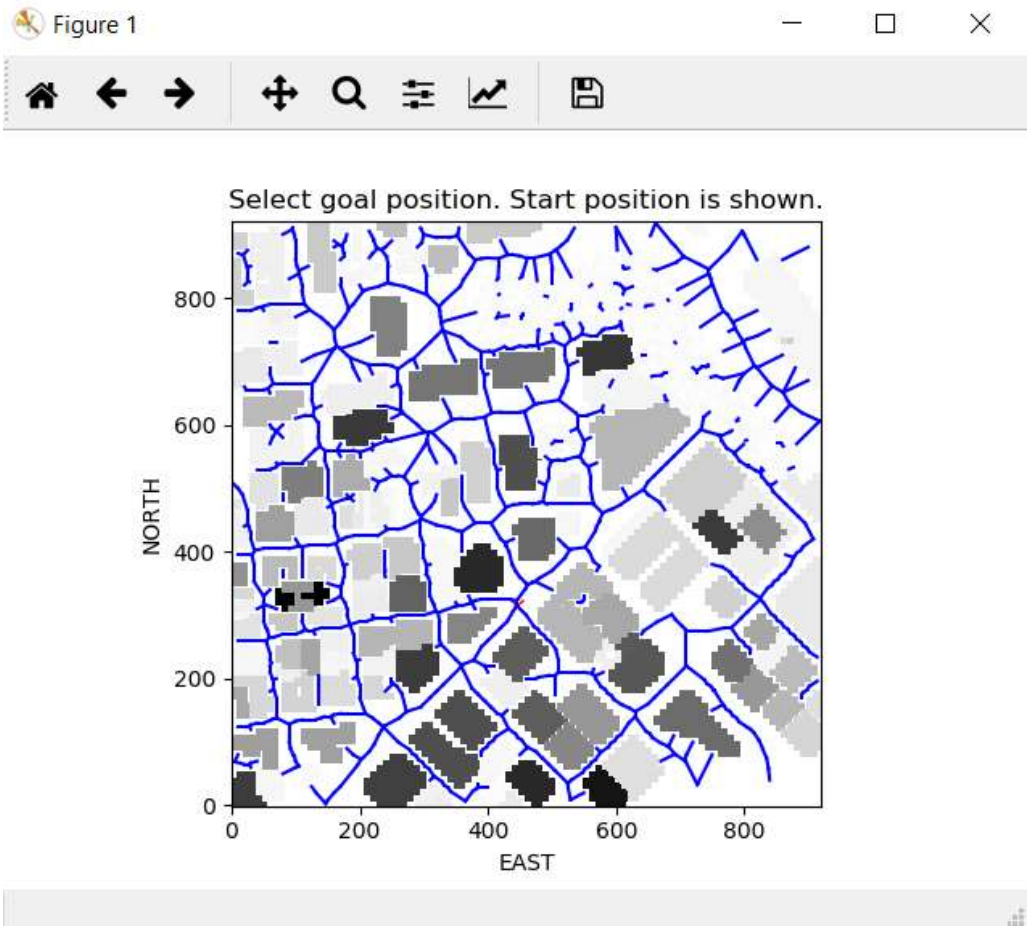
The current local position is set by default to 0, 0, 0. This basically maps the position of the drone to the global home position in the map.

Unfortunately, there is no way to reset this local position in the drone API (Without actually navigating), we will use the default local position as the starting point for the drone

Choosing the goal location

The drone location will be chosen by the user on a map. The map shows the city, the drone's starting point as well as a navigable Voronoi graph around the obstructions. The user can select the drone goal location. **Any location on the map may be selected as a goal – including the roof's of buildings and enclosed spaces.**

Image of map for goal selection



Goal selection code

```
def select_grid_start_and_goal(grid, north_offset, east_offset, SAFETY_DISTANCE):

    grid_start = local_position_2_grid_coord((0, 0, 0), grid, north_offset, east_offset)


    # Define a graph for a particular altitude and safety margin around obstacles

    graph = create_graph(data, grid, north_offset, east_offset, SAFETY_DISTANCE)


    plt.imshow(grid, origin='lower', cmap='Greys')

    plt.plot(grid_start[1], grid_start[0], 'rx')


    for e in graph.edges:

        p1 = e[0]

        p2 = e[1]
```

```
            plt.plot([p1[1], p2[1]], [p1[0], p2[0]], 'b-')


        plt.xlabel('EAST')
        plt.ylabel('NORTH')
        plt.title('Select goal position. Start position is shown.')
        pt_goal = plt.ginput(1, timeout=0)[0]
        plt.show(block=False)
        plt.close()


        # TODO: convert pts to grid values
        grid_goal = (int(pt_goal[1]), int(pt_goal[0]))


        starting_alt = np.int(grid[grid_start[0], grid_start[1]])
        grid_start_3d = (grid_start[0], grid_start[1], starting_alt+SAFETY_DISTANCE+1)


        landing_alt = np.int(grid[grid_goal[0], grid_goal[1]])
        grid_goal_3d = (grid_goal[0], grid_goal[1], landing_alt+SAFETY_DISTANCE+1)


        return graph, grid_start_3d, grid_goal_3d, landing_alt
```

The Voronoi graph generated here is unique in the sense that every navigable edge is checked to ensure there is adequate spacing on ALL sides, not just below the edge. Below is the code snippet from the *create_graph* function that does this:

```
    if (grid[n-sd:n+sd, e-sd:e+sd] > safety_distance).any():
        in_collision = True
        break
```

Search Algorithm

2 search algorithms were implemented

1. Grid A* including diagonal navigation
2. Hybrid Graph A* + Probabilistic RoadMap

Grid A* including diagonal navigation

A Numpy matrix is generated from the map data and every obstruction is marked in this matrix. The unique twist in my *create_grid* function is that instead of putting a 1 in every cell that has an obstruction, the actual obstruction altitude is recorded. See the code snippet below.

grid[obstacle[0]:obstacle[1]+1, obstacle[2]:obstacle[3]+1] = ***alt + d_alt***

The benefits of doing this are two-fold:

1. It enables landing on buildings because when the user picks any location on the map, we know its actual altitude
2. It enables efficient 3-D path planning and 3-D edge culling using the Bresenham technique. This will be critical in properly implementing the Hybrid Graph A* + Probabilistic RoadMap search

Diagonal grid navigation

This is implemented via the follow changes

In *class Action(Enum)*

```
# diagonal actions

NORTHEAST = (-1, 1, 1.414)

NORTHWEST = (-1, -1, 1.414)

SOUTHEAST = (1, 1, 1.414)

SOUTHWEST = (1, -1, 1.414)
```

In *def valid_actions(grid, current_node, safety_distance):*

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


if y - 1 < 0 or grid[x, y - 1] > safety_distance:
    valid_actions.remove(Action.WEST)
    if Action.NORTHWEST in valid_actions:
```

```
            valid_actions.remove(Action.NORTHWEST)
        if Action.SOUTHWEST in valid_actions:
            valid_actions.remove(Action.SOUTHWEST)
    if y + 1 > m or grid[x, y + 1] > safety_distance:
        valid_actions.remove(Action.EAST)
        if Action.NORTHEAST in valid_actions:
            valid_actions.remove(Action.NORTHEAST)
        if Action.SOUTHEAST in valid_actions:
            valid_actions.remove(Action.SOUTHEAST)
```

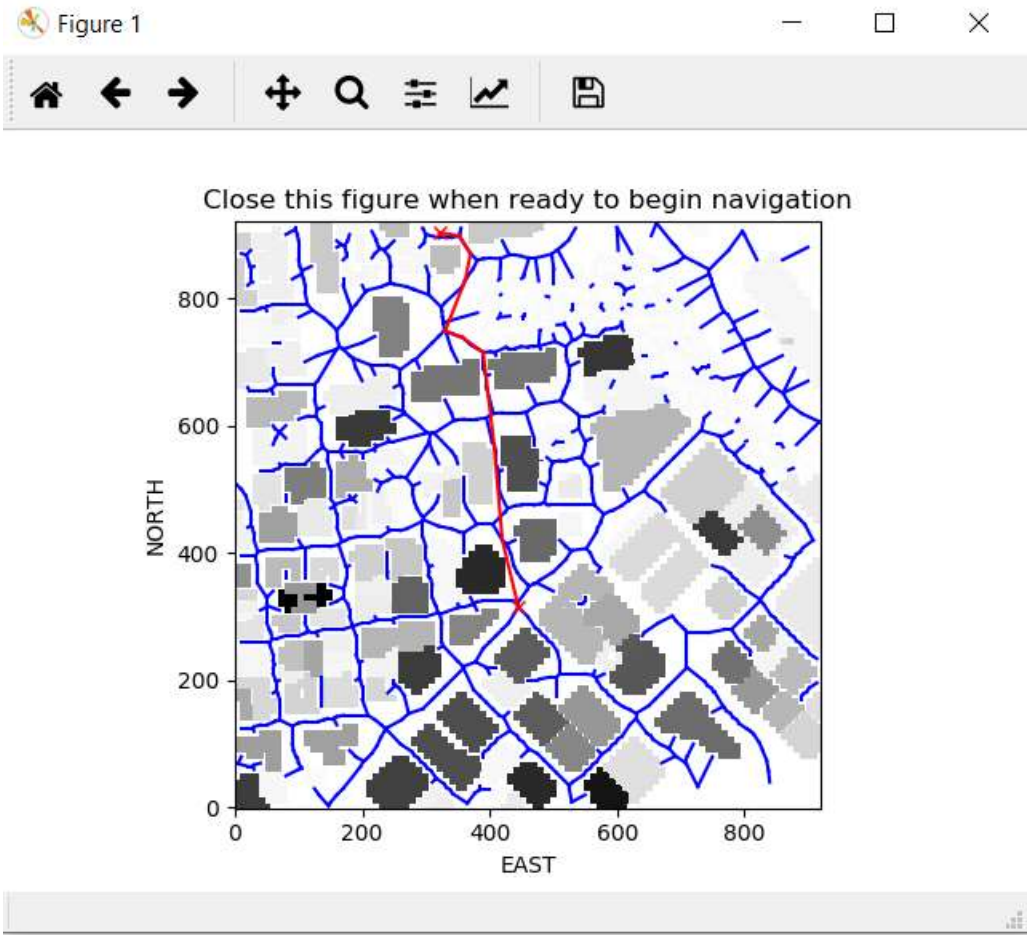
Note: There is a discrepancy between the map data provided in colliders.csv and that used in the simulator for a particular building. Colliders.csv records the building height as 1.5m whereas the simulator shows this building as at least 50m high. This creates a problem navigating around this building, so I had to put in a hack in *create_grid* as follows:

```
# HACK TO WORK AROUND BUG - SIM ALTITUDE DATA <> COLLIDERS ALTITUDE DATA
# FOR BUILDING AROUND GRID COORDINATES (383, 658)
if alt < safety_distance and  (north > -15 and north < 160) and (east > 120 and east < 310):
    alt = 25
    d_alt = 25
```

Hybrid Graph A* + 3D Probabilistic Roadmap

This is the recommended path search technique. It uses Voronoi graph search to find a path to the goal. **However, when the goal is completely enclosed on all sides, it uses Graph A* to get as close to the goal as possible, and then switches to a 3D probabilistic roadmap for the final leg into the enclosed space.**

Image of the planned path



See the code snippet below from the *plan_highlevel_path_using_graph_Astar* function.

```
print("Searching for a path ...")

# define graph start and graph goal
graph_start = closestNode(graph, (grid_start[0], grid_start[1]))
graph_goal = closestNode(graph, (grid_goal[0], grid_goal[1]))

start = time.time()

# Run A* to find a path from start to goal
# TODO: move to a different search space such as a graph (not done here)
foundPath2Goal, path, _ = graph_a_star(graph, heuristic, graph_start, graph_goal)
```

```
    if len(path) > 0:
        # altitude targets for smoothly transitioning to target altitude
        alts = np.linspace(grid_start[2], grid_goal[2], len(path))

        # construct 3d path points
        path = [[p[0], p[1], int(alt)] for p, alt in zip(path, alts)]

        # insert the grid_start into the graph-based path
        path.insert(0, list(grid_start))

        if not foundPath2Goal:
            # We have a disconnected graph,
            # let's join the disconnected sections using probabilistic roadmap planning
            foundPath2Goal, patchPath = createProbabilisticRoadMap(grid, tuple(path[-1]),
grid_goal, SAFETY_DISTANCE)
            if not foundPath2Goal:
                print('**********************')
                print('Could not find even a probabilistic roadmap. This goal is unreachable!')
                print('**********************')
                return []

            patchPath = [[int(p[0]), int(p[1]), int(p[2])] for p in patchPath]
            path += patchPath

        # put the grid goal into the graph-based path
        path.append(list(grid_goal))

        # TODO: prune path to minimize number of waypoints
        print("Pruning the path ...")
```

```
path = prune_path_bresenham(path, grid, SAFETY_DISTANCE)


# Convert path to waypoints
waypoints = [[p[0] + north_offset, p[1] + east_offset, p[2]] for p in path]
```

Note how the code linearly interpolates to generate drone altitudes for the waypoints along the path and generates 3D way points. This allows the drone to land on top of buildings

The code for *graph_a_star* has been modified to return the node closest to the goal, if navigation to the goal is not possible. This node then becomes the goal for the Graph A* and also the starting point for the 3D probabilistic roadmap. See the code below from *graph_a_star*.

```
# disjointed edges, find node nearest to goal that has a path to start
foundPath2Goal = nx.has_path(nxGraph,start,goal)
if (not foundPath2Goal) and failIfPathNotFound:
   print('**********************')
   print('Failed to find a path!')
   print('**********************')
   return foundPath2Goal, None, path_cost


listNodes = list(nxGraph)
if not foundPath2Goal:
   closestNodesIndices = np.argsort(np.linalg.norm(goal - np.array(listNodes), axis=1))
   for idx in closestNodesIndices:
      node = listNodes[idx]
      if (start == node or goal == node):
         continue


      if nx.has_path(nxGraph,start,node):
         goal = node
         break
```

3D Probabilistic Roadmap

This path search algorithm is used when Graph A* fails to find a path to the goal. While it is computationally expensive and slower, it is very effective when used to find a path into enclosed areas such as open-air atriums inside buildings. The code is shown below.

```
def createProbabilisticRoadMap(grid, grid_start, grid_goal, safety_distance):
    nmin = grid_start[0] if grid_start[0] < grid_goal[0] else grid_goal[0]
    nmax = grid_start[0] if grid_start[0] > grid_goal[0] else grid_goal[0]
    emin = grid_start[1] if grid_start[1] < grid_goal[1] else grid_goal[1]
    emax = grid_start[1] if grid_start[1] > grid_goal[1] else grid_goal[1]

    amin = grid_start[2] + safety_distance
    amax = grid[nmin-safety_distance:nmax+safety_distance,
            emin-safety_distance:emax+safety_distance].max() + 2*safety_distance

    # we are tackling situation where start and goal are on opposite sides of a barrier
    # only reachable by going over the top or through some opening
    num_of_samples = 300

    nsamples = np.random.random_integers(nmin, nmax, num_of_samples)
    esamples = np.random.random_integers(emin, emax, num_of_samples)
    asamples = np.random.random_integers(amin, amax, num_of_samples)
    samplePoints = list(zip(nsamples, esamples, asamples))

    # throw out ones that are inside or too close to an obstruction
    sd = 2
    idx = 0
    while idx < len(samplePoints):
        n = samplePoints[idx][0]
        e = samplePoints[idx][1]
        a = samplePoints[idx][2]
```

```
    if (grid[n-sd:n+sd, e-sd:e+sd] + safety_distance > a).any():
        samplePoints.pop(idx)
        continue
    idx += 1

# add the grid start and goal to the list of sample points
samplePoints.append(grid_start)
samplePoints.append(grid_goal)

# construct a navigable graph using the sample points
g = nx.Graph()
tree = KDTree(samplePoints)
for samplePt in samplePoints:
    idxs = tree.query([samplePt], k=12, return_distance=False)[0]
    for i in idxs:
        if samplePt == samplePoints[i]:
            continue

        if can_connect_3d(grid, samplePt, samplePoints[i], safety_distance)):
            dist = np.linalg.norm(np.array(samplePt) - np.array(samplePoints[i]))
            g.add_edge(samplePt, samplePoints[i], weight=dist)

if g.number_of_nodes() == 0:
    return False, None

foundPath2Goal, path, _ = graph_a_star(g, heuristic, grid_start, grid_goal, True)

if foundPath2Goal:
    # don't return grid start and grid_goal.
```

```
        # grid_start is already in the path. We will be adding grid_goal later
        return foundPath2Goal, path[1:-1]
    else:
        return foundPath2Goal, None
```

Note: If the above function were called at each waypoint or continuously, it could easily be used with some small modifications to implement the Receding Horizon path planning technique or even re-planning.

Videos

- Video of Graph A* with 3D Probabilistic Roadmap – **Simple**

  https://youtu.be/cV8Xm6xJyyc

- Video of Graph A* with 3D Probabilistic Roadmap – **Roof Landing**

  https://youtu.be/7SzpCOfmCsQ

- Video of Graph A* with 3D Probabilistic Roadmap – **Landing in enclosed space**

  https://youtu.be/7uULcpbmyEs

Comparison of Grid A* vs Graph* with Probabilistic Roadmap

Grid A*with diagonal search	Graph A* + Probabilistic Roadmap
Slow – on the order of 100's of seconds	Superfast when probabilistic roadmap is not required. Order of subseconds Fast when probabilistic roadmap is required. Order of a few seconds
Thorough search. Guaranteed to find a path if it exists	Graph A* by itself may fail since not all nodes are guaranteed reachable Graph A* + Probabilistic Roadmap is better than Grid A*, even for enclosed spaces. However, no guarantees
Implementation is simple	Implementation is complex

Path Pruning

3D Bresenham path pruning is used to cull edges that would intersect obstructions or that do not have an adequate safety distance from obstructions. See code below

```
def prune_path_bresenham(path, grid, safety_distance):
    idx = 0
    while idx < len(path)-2:
```

```
            if can_connect_3d(grid, path[idx], path[idx+2], safety_distance):
                path.pop(idx+1)
                continue

            idx += 1

        return path

    def can_connect_3d(grid, p1, p2, safety_distance):
        bList = Bresenham3D(int(p1[0]), int(p1[1]), int(p1[2]),
                            int(p2[0]), int(p2[1]), int(p2[2]))
        sd = 2
        in_collision = False
        for pt in bList:
            n = int(pt[0])
            e = int(pt[1])
            a = int(pt[2])

            # grid[n, e] is the altitude
            if (grid[n-sd:n+sd, e-sd:e+sd] + safety_distance > a).any():
                in_collision = True
                break

        return not in_collision
```

Note: The Bresenham3D code was borrowed from https://www.geeksforgeeks.org/bresenhams-algorithm-for-3-d-line-drawing/

Waypoint Dead-banding

Given that there is lag between the actual drone position and the position information received by the navigation algorithm, waypoint detection needs to be around an area surrounding the zone, rather than

exactly on the waypoint. This is necessary for smooth movement of the drone during waypoint transition.

The dead-band radius around the waypoint is based on the drone's speed. The implementation is as follows:

Initialization code

```
# used in deadbanding

MAX_DRONE_SPEED = 10

drone.deadband_multiplier = SAFETY_DISTANCE/MAX_DRONE_SPEED
```

In local_position_callback function

```
elif self.flight_state == States.WAYPOINT:

    localPos = np.array([self.local_position[0], self.local_position[1], -self.local_position[2]])

    dist2Target = np.linalg.norm(self.target_position - localPos)

    droneSpeed = np.linalg.norm(self.local_velocity[0:2])

    if len(self.waypoints) > 0:

        if dist2Target < self.SAFETY_DISTANCE + droneSpeed*self.deadband_multiplier:   # deadbanding

            self.waypoint_transition()
```

Performance Considerations

Path planning algorithms are time-intensive, especially Grid A*! This means we cannot execute these algorithms on the fly after the drone has taken off.

To solve this problem, the path must be planned BEFORE the drone is connected to the simulator to avoid a connection timeout. This is accomplished in the main program as follows:

```
if __name__ == "__main__":

    parser = argparse.ArgumentParser()

    parser.add_argument('--port', type=int, default=5760, help='Port number')

    parser.add_argument('--host', type=str, default='127.0.0.1', help="host address, i.e.
'127.0.0.1'")

    args = parser.parse_args()


    SAFETY_DISTANCE = 3


    # TODO: read lat0, lon0 from colliders into floating point values
```

```
    firstRowData = np.genfromtxt('colliders.csv', delimiter=',', dtype=None, max_rows=1,
encoding=None)
    lat0 = float(firstRowData[0].split()[1])
    lon0 = float(firstRowData[1].split()[1])
    global_home = np.array([lon0, lat0, 0])

    # Read in obstacle map
    data = np.loadtxt('colliders.csv', delimiter=',', dtype='Float64', skiprows=2)

    # Define a grid for a particular altitude and safety margin around obstacles
    grid, north_offset, east_offset = create_grid(data, SAFETY_DISTANCE)
    print("North offset = {0}, east offset = {1}".format(north_offset, east_offset))

    # select grid start and goal.
    # goal is a randomly picked lat/lon converted to grid coordinates
    graph, grid_start_3d, grid_goal_3d, landing_alt = \
       select_grid_start_and_goal(grid, north_offset, east_offset, SAFETY_DISTANCE)
    print('Grid Start and Goal: ', grid_start_3d, grid_goal_3d)

    waypoints = []

    print('Planning a path using graph Astar')
    waypoints = plan_highlevel_path_using_graph_Astar(graph, grid, north_offset,
                                        east_offset,
                                        SAFETY_DISTANCE, grid_start_3d,
                                        grid_goal_3d)

    #if len(waypoints) == 0:
    #    print('Planning a path using grid Astar')
```

```
    #    waypoints = plan_highlevel_path_using_grid_Astar(grid, north_offset,
    #                                    east_offset,
    #                                    SAFETY_DISTANCE, grid_start_3d,
    #                                    grid_goal_3d)

    if len(waypoints) > 0:
        conn = MavlinkConnection('tcp:{0}:{1}'.format(args.host, args.port), timeout=60)
        drone = MotionPlanning(conn)
        time.sleep(1)

        # initialize drone safety distance (for use in receding horizon calcs)
        drone.SAFETY_DISTANCE = SAFETY_DISTANCE

        # store home position (drone home position is set later in plan_path from these values)
        drone.home_lon = lon0
        drone.home_lat = lat0

        # store grid info for receding horizon calculations while navigating waypoints
        drone.grid = grid
        drone.north_offset = north_offset
        drone.east_offset = east_offset

        # Set drone waypoints
        drone.waypoints = waypoints

        # set initial target altitude for takeoff and landing alt (since we could land on top of a
building)
        drone.target_position[2] = SAFETY_DISTANCE
        drone.landing_alt = landing_alt
```

 # used in deadbanding

MAX_DRONE_SPEED = 10

drone.deadband_multiplier = SAFETY_DISTANCE/MAX_DRONE_SPEED

drone.start()

Known Bugs

When the drone takes off, instead of going up to 3m as specified in the program, it shoots up over 125m. I suspect this is due to a slow computer or not enough memory, but was not able to workaround it. It may also be due to the more compute intensive code in *local_position_callback* implemented for deadbanding.

Recommendations

- Receding horizon planning could help plan on the fly with a faster computer
- Potential field planning could be implemented to prevent collisions in flight. Again need a fast computer
- Could predict actual position based on prior position, velocity and acceleration data and use it for waypoint transition
- Drone control is not implemented. Flight constraints need to be modeled and implemented for realistic navigation

Helper Functions

The following helper function were implemented:

- def closestNode(graph, node):
- def local_position_2_grid_coord(localPos, grid, north_offset, east_offset):
- def grid_coord_2_local_position(grid_Coord, north_offset, east_offset):
- def are_collinear(p1, p2, p3):
- def can_connect_3d(grid, p1, p2, safety_distance):
- def prune_path_collinearity(path):
- def prune_path_bresenham(path, grid, safety_distance):

Notes

- Waypoint coordinates must be integers
- Graph edge – Cannot add points as np array. Use tuples

References

- Udacity course materials – Flying Cars and Autonomous Flight Engineer Course
- https://www.geeksforgeeks.org/bresenhams-algorithm-for-3-d-line-drawing/

Appendix

All code for the project including this document can be found at https://github.com/dharrawal/FCND-Motion-Planning