P2: Tree Planning Through The Trees

Mayank Bansal
Robotics Engineering
Worcester Polytechnic Institute
Email: mbansal1@wpi.edu

Siyuan 'Oliver' Huang
Robotics Engineering
Worcester Polytechnic Institute
Email: shuang4@wpi.edu

Miheer Diwan
Robotics Engineering
Worcester Polytechnic Institute
Email: msdiwan@wpi.edu

***Abstract*—The aim of this project is to implement path and trajectory planning and tune the control stack for a quadrotor to navigate from a start position to a goal position through a pre-mapped or known 3D environment. The simulation is done in Blender. The Path Planning is done through RRT* algorithm and the trajectory planning is done using minimum-snap trajectory planner. The controller is developed in a cascaded fashion with the outer loop controlling the position and the inner loop controlling the velocity of the drone. PID controllers are used to track the trajectory generated.**

I. Map Visualisation

The known maps are stored in .txt files with the axes boundary limit information of the environment and the cuboid obstacles. The cuboid obstacles also have RGB values which signify their color. The maps are visualized and simulated using Blender. The boundary of the simulated environment is made transparent using the Alpha Blending feature in Blender. This is done so that the obstacles and the drone are visible from outside.

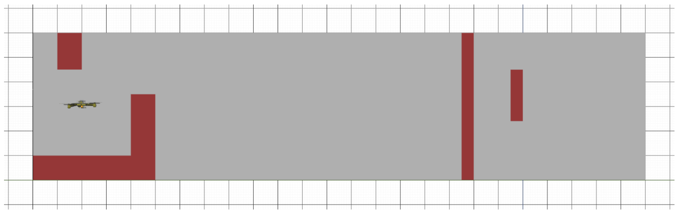
Fig. 1. Side View of Map 1

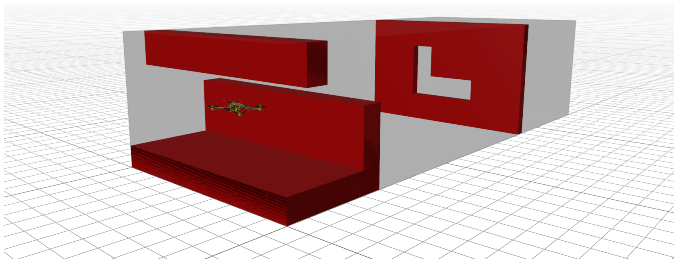
Fig. 2. Orthographic View of Map 1

II. Path Planning Using RRT*

The path from the start to end is found using RRT*. The algorithm is as follows:

Algorithm 1 RRT* (Rapidly-exploring Random Tree*)
```
1:  $Rad \leftarrow r$
2:  $G(V, E)$                         ▷ Graph containing edges and vertices
3:  for $itr$ in $range(0, n)$ do
4:      $Xnew \leftarrow RandomPosition()$
5:      if $Obstacle(Xnew) == True$ then
6:          continue                   ▷ Try again if in an obstacle
7:      end if
8:      $Xnearest \leftarrow Nearest(G(V, E), Xnew)$
9:      $Cost(Xnew) \leftarrow Distance(Xnew, Xnearest)$
10:     $Xbest, Xneighbors \leftarrow findNeighbors(G(V, E), Xnew, Rad)$
11:     $Link \leftarrow Chain(Xnew, Xbest)$
12:     for $x'$ in $Xneighbors$ do
13:         if $Cost(Xnew) + Distance(Xnew, x') < Cost(x')$ then
14:             $Cost(x') \leftarrow Cost(Xnew) + Distance(Xnew, x')$
15:             $Parent(x') \leftarrow Xnew$
16:             $G \leftarrow G \cup \{Xnew, x'\}$        ▷ Add edge to the graph
17:         end if
18:     end for
19:     $G \leftarrow G \cup Link$        ▷ Add the new link to the graph
20: end for
21: return $G$
```

The maximum number of samples(nodes) is 3000.

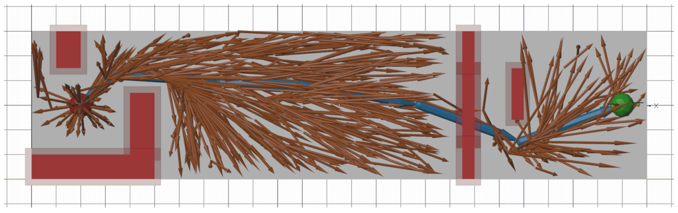
Fig. 3. Side View of RRT* Path

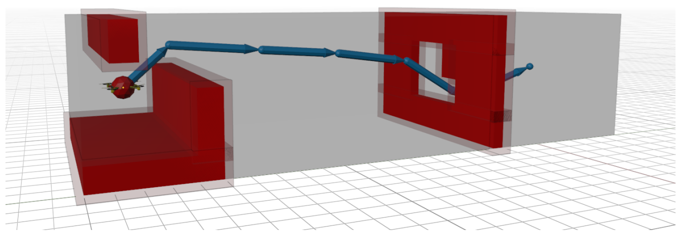
Fig. 4. Orthographic View of the final RRT* Path

III. Trajectory Generation

Using the RRT* global path planner algorithm, we are able to generate the shortest path from the 'Start' position to the 'Goal' position within the map environment. Since the path generated is not smooth and has sharp turns, it is dynamically unfeasible as this will cause the quadrocopter to overshoot significantly. Therefore, we convert the waypoints obtained in the previous step to a smooth spline. Initially, we tried to implement a quintic polynomial trajectory to solve this problem. However, the trajectory generated did not accurately incorporate all the waypoints. Hence, we decided to use a minimum snap trajectory generator to generate a smooth trajectory as shown in [1]. It aims to minimize the "snap" or the fourth derivative of position with respect to time, ensuring that the trajectory is not only continuous but also has continuous velocity, acceleration, and jerk profiles. This results in smoother and more natural movements, reducing wear and tear on the robot or vehicle and improving overall efficiency and safety. Through trial and error, we found out that a velocity of 3 m/s was best for traversing the generated trajectories. The code for trajectory generation was heavily inspired by this GitHub repository: Quadrotor-Simulation.

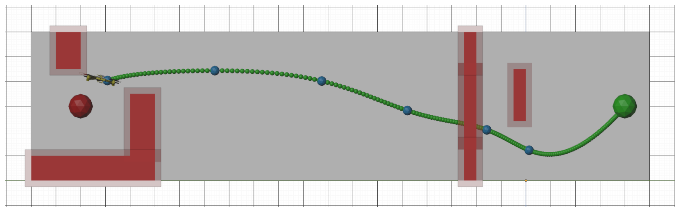
Fig. 5. Side view of the generated trajectory

IV. Collision Handling

In RRT*, for collision handling, we are generating the 3D coordinates of each point on the line connecting the sampled node with the nearest node on the tree. Then we check if any of these points lie inside an obstacle or outside the boundary of the environment. If not, the new node is added to the tree. To avoid the drone getting too near the obstacles, the boundaries of the obstacles are bloated to create a safe distance for the drone.

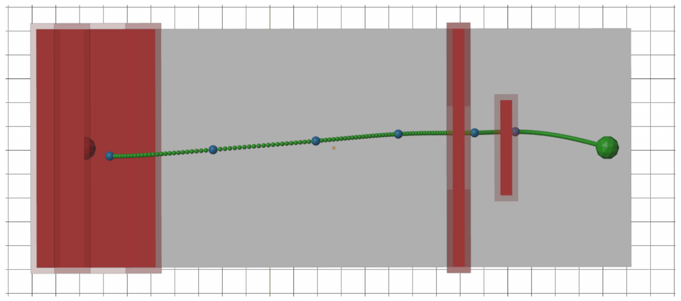
Fig. 6. Top view of the generated trajectory

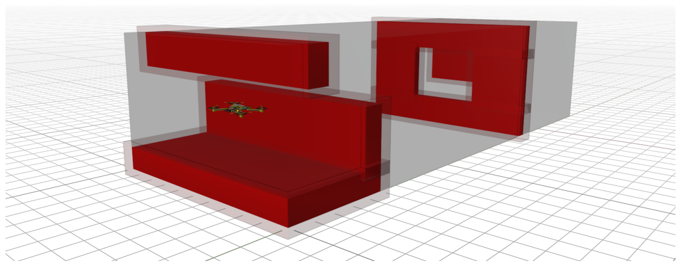
Fig. 7. Orhtographic view of Map 1 with bloated obstacles

V. Controller Design

The controller is designed in a cascaded manner in which there is an outer position control loop and then there is an inner velocity control loop. Both of these are PID controllers. The controllers are tuned in a step-by-step manner. First, the velocity controller is tuned since it is the inner control loop. The PID tuning values are: (1,0,0) for x-direction; (1,0,0) for y-direction and (0.1,0.01,0.1) for z-direction. When the velocity controller is satisfactorily tuned, the position control loop is tuned. The PID values are: (2,0.5,0.5) for the x-direction; (2,0.5,0.5) for the y-direction and (20,0.1,0.1) for the z-direction.

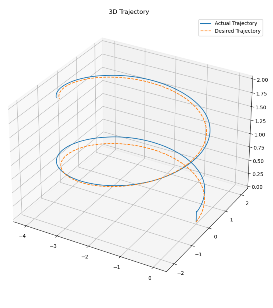

Fig. 8. Helical trajectory with tuned controller

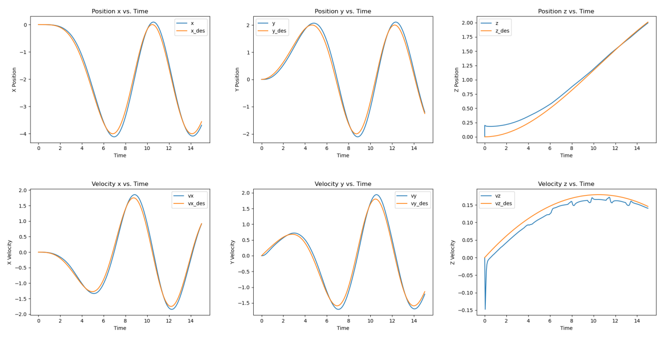
Fig. 9. Position and Velocity plots for Helical trajectory

VI. Results

- The trajectory plot and the plots of the position and velocity for map 1 are shown in Fig. 10 and Fig. 11
- The trajectory plot and the plots of the position and velocity for map 2 are shown in Fig. 12 and Fig. 13
- The trajectory plot and the plots of the position and velocity for map 3 are shown in Fig. 14 and Fig. 15
- The trajectory plot and the plots of the position and velocity for map 4 are shown in Fig. 16 and Fig. 17
- Video Submission Link

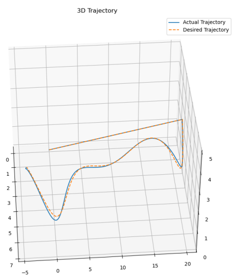

Fig. 10. Trajectory plot for Map 1

VII. Deployment on Tello EDU Drone

We used the procedure described above to navigate the environment in the real world. We integrated our code with the DJITelloPy library and deployed it on the real DJI Tello EDU drone. Fig. 18 shows the drone.

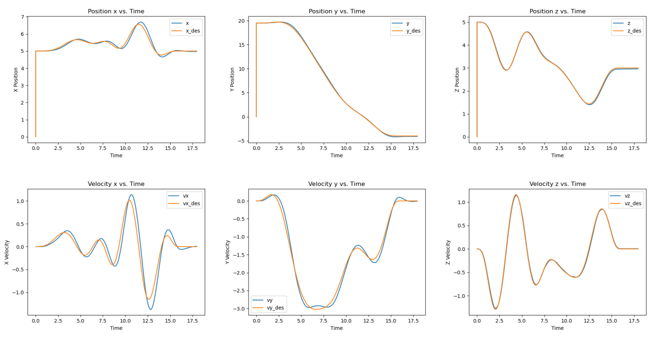
Fig. 11. Position and Velocity plots for Map 1

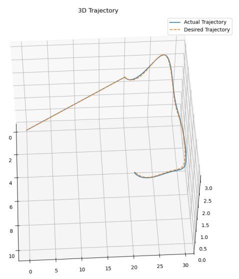

Fig. 12. Trajectory plot for Map 2

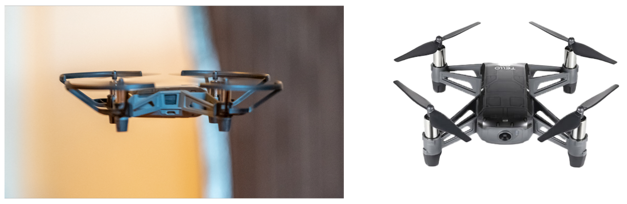
Fig. 18. DJI Tello EDU Drone

A. Sim-2-Real

To traverse our generated trajectory we implemented a simple position controller and passed commands to the drone using the 'go x y z speed' command. The results were satisfactory and the drone was able to reach the Goal without colliding with any of the obstacles during multiple tests.
However, the traversed path was not smooth and hence we decided to use a velocity controller for smooth trajectory tracking. The velocity values are obtained from the trajectory generator at each time step and are passed to the drone using the command send_rc_control which takes in velocity values in the format (left_right_velocity, forward_backward_velocity, up_down_velocity, yaw_velocity). The velocity values are directly taken from the trajectory generator and the maximum velocity for the drone was taken as 100 cm/s. The drone

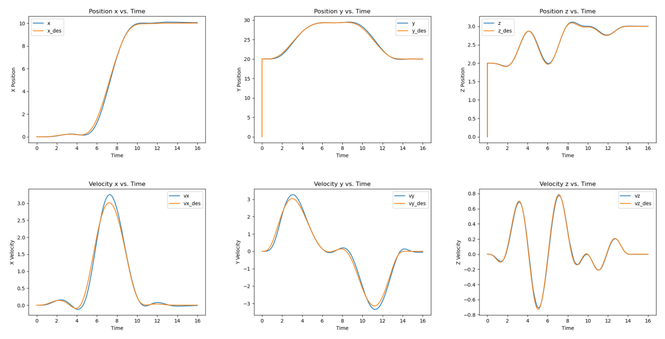

Fig. 13. Position and Velocity plots for Map 2

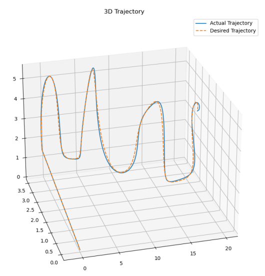

Fig. 14. Trajectory plot for Map 3

motion was observed to be smooth and followed the trajectory pretty well. However, because of the high-velocity value, the drone consistently converged at a distance of 40 cm away Goal position in the X and Y direction.

- Video Submission Link

B. Limitations

1) Initially we implemented a position controller. Though it does well to move the drone through the waypoints generated by RRT*, due to limitations in how the API is developed such as the drone will move to the specified point only if the distance between the current location and the next waypoint is greater than 30 cm, which makes it difficult to accurately command the drone to reach the goal.
2) One more problem was that the rc_control accepts velocity commands only in integers but the trajectory generator generates float velocity values. So, we had to round off these values to the nearest integers before sending them to the drone. This led to the drone under-shooting the actual goal position because we were often sending commands that were not accurate according to the generated velocity. This proved to be a challenge when implementing a velocity controller.

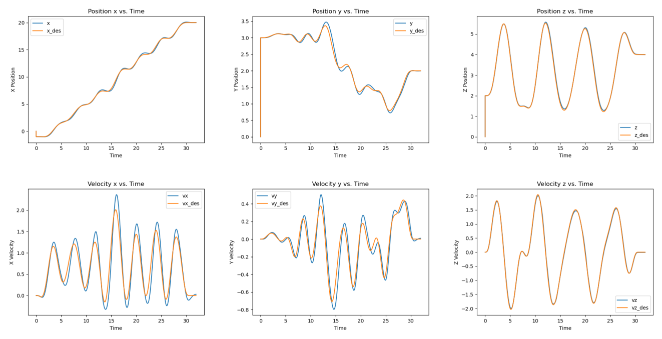

Fig. 15. Position and Velocity plots for Map 3

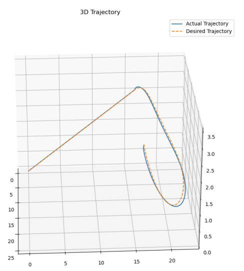

Fig. 16. Trajectory plot for Map 4

3) The major problem we faced was that since the connection between Jetson and the drone was not secure and the communication between the two used UDP architecture, there might be losses in wireless data transfer that often lead to the drone performing unexpected maneuvers. This is more of a problem in the velocity controller since we need to continuously publish the correct velocity commands for the drone to perform the maneuver correctly.

REFERENCES

[1] C. Richter, A. Bry, and N. Roy, “Polynomial trajectory planning for aggressive quadrotor flight in dense indoor environments,” in *International Symposium of Robotics Research*, 2016. [Online]. Available: https://api.semanticscholar.org/CorpusID:9070368

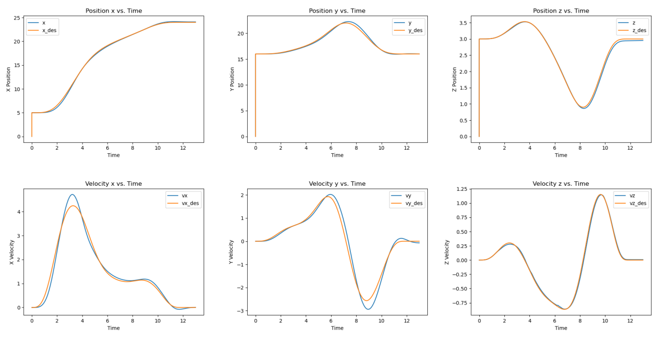

Fig. 17. Position and Velocity plots for Map 4