Team Apache Stealth: Tree Planning Through The Trees!

Ankit Mittal
Department of Robotics Engineering
Worcester Polytechnic Institute
Email: amittal@wpi.edu

Rutwik Kulkarni
Department of Robotics Engineering
Worcester Polytechnic Institute
Email: rkulkarni1@wpi.edu

***Abstract*—(1 Late day) This paper delves into the practical implementation of the RRT* algorithm within a known 3D environment, with the primary objective of navigating a drone from an initial starting position to a predefined goal position. The paper not only elucidates the application of the RRT* algorithm but also addresses the critical post-processing step of path smoothing. This is achieved through the utilization of a trajectory generation algorithm, which optimizes the path obtained from the RRT* algorithm. Furthermore, the paper extends its focus to the development of a controller system tailored for the drone, enabling it to faithfully follow the optimized trajectory path. The comprehensive approach presented in this research paper, spanning from path planning with RRT* to trajectory generation and controller design, underscores its practical significance in the field of autonomous drone navigation within complex 3D environments.**

I. Introduction

The rise of autonomous drones has opened up exciting possibilities across numerous fields. However, navigating these drones through complex 3D environments demands advanced path planning and control solutions. This paper tackles this challenge by combining the RRT* algorithm for path planning, trajectory generation techniques, and a robust controller system. Our focus is to guide a drone from its initial position to a defined goal, ensuring smooth, obstacle-avoiding trajectories and precise control. We begin with RRT* as the foundation for path planning, extending it to 3D environments. To improve the generated path's quality, we introduce a trajectory optimization step.Crucially, we also detail the development of a tailored controller for the drone. This ensures the drone faithfully follows the optimized trajectory, even in unpredictable conditions. Our holistic approach aims to enhance autonomous drone navigation in intricate 3D spaces, offering practical benefits across various real-world applications. In the following sections, we delve into our methodology, implementation, and results, showcasing the effectiveness of our approach.

Link To Videos: **Click Here**

II. Environment Setup(Map Reader in Blender)

As part of the initial setup, program needs to read environmental data from a text file. This text file should contain obstacle dimensions formatted in the following manner.

Boundary:$x_{min}\ y_{min}\ z_{min} x_{max}\ y_{max}\ z_{max}$

Block:$x_{min}\ y_{min}\ z_{min} x_{max}\ y_{max}\ z_{max}\ r\ g\ b$

Here $x_{min}\ \ y_{min}\ \ z_{min}$ represents the lower left corner coordinates of the block/boundary and $x_{max}\ y_{max}\ z_{max}$ represents the upper right coordinates of the block/ boundary. Script reads the environment and plot it in the blender. And in block $r\ g\ b$ represents the rbg values of the block for color coding.

Map 1 - Train Set

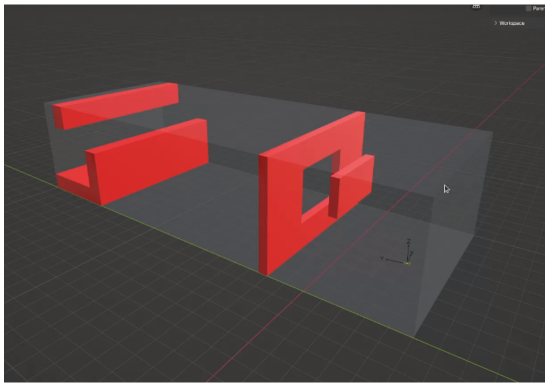

Fig. 1: Environment Setup in blender

III. Path Planner

The RRT* planning algorithm is employed for path planning from the starting position to the goal position. RRT* represents an enhanced and optimized iteration of the original RRT algorithm. In RRT random points are generated and linked to the nearest accessible node. Before creating a vertex, a check is performed to ensure that it is positioned outside of any obstacles. Additionally, when connecting the vertex to its nearest neighbor, precautions are taken to avoid obstacles. The algorithm terminates either when a node is generated inside the desired goal region or when a predefined limit is reached.The basic principle of RRT* is the same as RRT, but two key additions to the algorithm result in significantly different results. In the RRT* algorithm, each vertex maintains a record of the distance it has covered relative to its parent vertex, which is denoted as its ”cost.” Once the nearest node within the graph is identified, the algorithm looks at a set of nearby vertices within a fixed radius around the newly created node. If a vertex with a lower cost than the closest proximal node is discovered, it replaces the proximal node. This feature has a noticeable impact on the tree structure, resulting in the emergence of fan-shaped branches and eliminating the cubic structure seen in the standard RRT algorithm. Another key enhancement introduced by RRT* is the concept of tree rewiring. Once a vertex has been linked to its most cost-effective neighbor, the algorithm goes on to reevaluate the neighboring vertices. It assesses whether rewiring a neighbor to the recently added vertex would result in a reduction in their cost. If such an improvement is observed, the algorithm proceeds to rewire that neighbor to the newly added vertex. This mechanism contributes to creating smoother and more efficient paths in the tree structure. In this project, we employ a random number generator to determine positions in space. However, to improve the efficiency of our sampling strategy, we incorporate a heuristic based on the goal position. This heuristic allows us to bias the selection of random points towards the vicinity of the goal. As a result, our tree expansion predominantly occurs in the direction of the goal, facilitating a more targeted exploration of the search space.

The above alogorithm gives set of waypoints from start position to the goal position. Then the waypoints are passed through trajectory generator for path smoothing.

Map 1 - Train Set

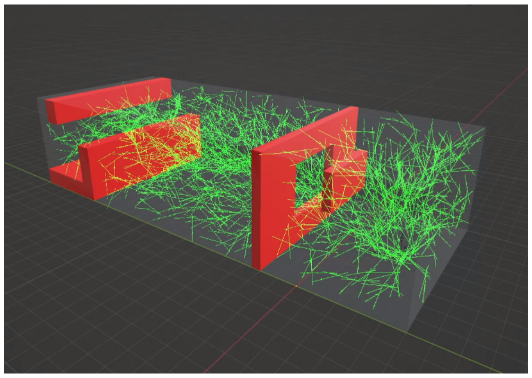

Fig. 2: Expanded RRT* Tree

Algorithm 1 RRT* Algorithm

```
$V \leftarrow \{x_{\text{init}}\}$;    $E \leftarrow \emptyset$;
for $i = 1$ to $n$ do
    $x_{\text{rand}} \leftarrow \text{SampleFree}_i$;
    $x_{\text{nearest}} \leftarrow \text{Nearest}(G = (V, E), x_{\text{rand}})$;
    $x_{\text{new}} \leftarrow \text{Steer}(x_{\text{nearest}}, x_{\text{rand}})$;
    if ObstacleFree($x_{\text{nearest}}$, $x_{\text{new}}$) then
        $X_{near} \leftarrow Near(G = (V, E), x_{\text{new}}, \min(\gamma(\log(\text{card}(V))/\text{card}(V))^{1/d}, \eta))$;
        $V \leftarrow V \cup \{x_{\text{new}}\}$;
        $x_{\text{min}} \leftarrow x_{\text{nearest}}$;
        $c_{\text{min}} \leftarrow \text{Cost}(x_{\text{nearest}}) + c(\text{Line}(x_{\text{nearest}}, x_{\text{new}}))$;
        for each $x_{\text{near}} \in X_{\text{near}}$ do
            if CollisionFree($x_{\text{near}}$, $x_{\text{new}}$) $\wedge$ Cost($x_{\text{near}}$) + $c(\text{Line}(x_{\text{near}}, x_{\text{new}})) < c_{\text{min}}$ then
                $x_{\text{min}} \leftarrow x_{\text{near}}$;
                $c_{\text{min}} \leftarrow \text{Cost}(x_{\text{nearest}}) + c(\text{Line}(x_{\text{nearest}}, x_{\text{new}}))$;
            end if
            $E \leftarrow E \cup \{(x_{\text{new}}, x_{\text{near}})\}$;
            for each $x_{\text{near}} \in X_{\text{near}}$ do    ▷ Rewire
                if CollisionFree($x_{\text{near}}$, $x_{\text{new}}$) $\wedge$ Cost($x_{\text{new}}$) + $c(\text{Line}(x_{\text{near}}, x_{\text{new}})) < \text{Cost}(x_{near})$ then
                    $x_{parent} \leftarrow \text{Parent}(x_{near})$;
                    $E \leftarrow E \setminus \{(x_{\text{parent}}, x_{\text{near}})\} \cup \{(x_{\text{new}}, x_{\text{near}})\}$;
                end if
            end for
        end for
    end if
end for
return $G = (V, E)$
```

Map 1 - Train Set

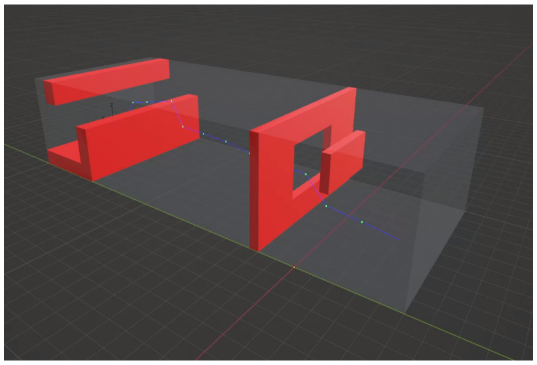

Fig. 3: Path Generated

IV. Trajectory Generation

The approach used for trajectory generation utilizes polynomial interpolation to smoothly connect a series of waypoints, ensuring continuous motion profiles with respect to position, velocity, and acceleration in all directions. The trajectories are then visualized using Matplotlib, allowing for an in-depth analysis of the generated motion profiles.

We use the following Polynomial Equation for Trajectory Generation:

$$q(t) = a_0 + a_1 t + a_2 t^2 + a_3 t^3 + a_4 t^4 + a_5 t^5$$

where:
$q(t) \longrightarrow$ position at time t
$a_0, a_1, a_2, a_3, a_4, a_5$ are coeffs. of the polynomials

This polynomial equation is used to calculate position, velocity, and acceleration profiles for each segment of the trajectory. Quintic polynomials are used to interpolate between waypoints. You need to define the initial and final positions, velocities, and accelerations for each segment of your trajectory. These waypoints serve as boundary conditions that the quintic polynomial must satisfy.

The quintic polynomial equation has six coefficients: a_0 through a_5. These coefficients need to be determined to create the desired trajectory. To do this, set up a system of equations based on the boundary conditions. For a single segment of the trajectory between time t_0 and t_f, you can use the following equations:

Position Boundary Conditions:

$$q(t_0) = a_0 + a_1 t_0 + a_2 t_0^2 + a_3 t_0^3 + a_4 t_0^4 + a_5 t_0^5 = p_0 \tag{1}$$

$$q(t_f) = a_0 + a_1 t_f + a_2 t_f^2 + a_3 t_f^3 + a_4 t_f^4 + a_5 t_f^5 = p_f \tag{2}$$

Velocity Boundary Conditions:

$$q'(t_0) = a_1 + 2a_2 t_0 + 3a_3 t_0^2 + 4a_4 t_0^3 + 5a_5 t_0^4 = v_0 \tag{3}$$

$$q'(t_f) = a_1 + 2a_2 t_f + 3a_3 t_f^2 + 4a_4 t_f^3 + 5a_5 t_f^4 = v_f \tag{4}$$

Acceleration Boundary Conditions:

$$q''(t_0) = 2a_2 + 6a_3t_0 + 12a_4t_0^2 + 20a_5t_0^3 = a_0 \tag{5}$$

$$q''(t_f) = 2a_2 + 6a_3t_f + 12a_4t_f^2 + 20a_5t_f^3 = a_f \tag{6}$$

These equations constitute a system of six equations with six unknowns (a_0 to a_5). We can use Linear Algebra Solvers to determine the coefficients of the quintic polynomial by solving this system.

Once we have determined the coefficients (a_0 through a_5), we substitute them into the quintic polynomial equation to calculate position, velocity, and acceleration values at any time t within the segment.

Quintic polynomials offer control over position, velocity, and acceleration. To regulate the time taken to traverse a segment, we linearly interpolate time between t_0 and t_f according to the desired duration. These time values can be either hardcoded or dynamically adjusted based on user-defined conditions, such as distance as a heuristic.

For multiple segments in the path generated by our planning algorithm, we ensure continuity by matching the final conditions (position, velocity, and acceleration) of one segment with the initial conditions of the next. The way we do this is summarized below:

- A loop iterates through pairs of waypoints, each representing the start and end points of a trajectory segment. It establishes the initial and final conditions for each segment.
- For the first segment, both initial velocity and acceleration are set to zero. For subsequent segments, the initial conditions are determined based on the final values of the previous segment, ensuring smooth transitions.
- The final segment's velocity and acceleration are set to zero as there is no subsequent segment. Velocity is calculated as the change in position divided by the segment duration, and acceleration as the rate of change of velocity, divided by the segment duration, for all segments except the last one.

V. Controller Design and Tuning

A. *Cascaded PID Controller*

The provided controller is a cascaded PID (Proportional-Integral-Derivative) controller for a quadrotor, which is a popular control architecture(used by PX4) for stabilizing and controlling the flight of drones. This cascaded design consists of multiple PID controllers stacked in a hierarchical manner.

Position Controller : The outermost layer of the cascaded controller focuses on position control in NED (North-East-Down) coordinates. It uses three PID gains to regulate the quadrotors position's along X,Y and Z direction. These controller take the desired position waypoints from our trajectory and current position state of our drone as input and compute the desired velocity setpoints to reach the desired position.
Velocity Controller : The next layer of the controller focuses on velocity control in NED coordinates. It adjusts the quadrotor's velocity based on desired velocity setpoints. Three additional PID gains are used to control the quadrotor's velocity along the X, Y, and Z axes. These controllers take the desired velocity setpoints and the current velocity as input and calculate acceleration setpoints to achieve the desired velocity.
Angular Rate Controller : The innermost layer of the controller deals with controlling the quadrotor's angular rates. It uses three PID gains to regulate the quadrotor's roll, pitch, and yaw rates. These controllers take the desired angular rates and the current angular rates as input and compute torque commands to achieve the desired rates.

B. *PID Gain Tuning Methodology*

The cascaded control architecture divides the control process into multiple layers and dimensions, each with its set of PID gains. Managing and fine-tuning all these gains concurrently can be complex and time-consuming. This issue was addressed by sequentially tuning the controller starting from the inner-most layer. By first focusing on the innermost layer (angular rate) and progressively moving outward to velocity and position control, it becomes more manageable to isolate and fine-tune each layer's gains.

Angular Rate Controller : This controller was already tuned in the starter code provided. Starting with this controller is crucial for maintaining stability and responsiveness in flight.

Velocity Controller : The focus then shifts to the velocity controller. Tuning the velocity controller, especially in the Z-axis (vertical control), helps the quadrotor achieve smooth and precise motion in terms of ascent and descent. Once the Z-axis velocity controller is tuned, it becomes easier to tune the X and Y-axis velocity controllers, which control horizontal motion. The PID gains used for this controller are :
Velocity in X-direction - **P**: 1.0, **I**: 0.1, **D**: 0.05
Velocity in Y-direction - **P**: 4.0, **I**: 0.1, **D**: 0.01
Velocity in Z-direction - **P**: 20.0, **I**: 0.0, **D**: 0.05

Position Controller : Finally, tuning the position controller in the Z-axis and then the X and Y axes follows a logical progression. This approach builds on the foundation of well-tuned velocity control to achieve accurate position control. Tuning the position controller ensures that the quadrotor can accurately reach and maintain desired waypoints in space. The PID gains used for this controller are :
Position in X-direction - **P**: 1.0, **I**: 0.0, **D**: 0.1
Position in Y-direction - **P**: 1.0, **I**: 0.0, **D**: 0.1
Position in Z-direction - **P**: 1.0, **I**: 0.0, **D**: 0.1

VI. Results

The algorithm's performance was evaluated by training it on one map and testing it on two separate test maps. During the testing phase, the algorithm successfully guided the drone along a path while avoiding obstacles. This achievement was confirmed through visual inspection and further validated by comparing the extracted plots of the drone's current and desired states.

A. Results

1) Figure 4,5,6 - Positions in 3D (Desired Path v/s Actual Path Positions)
2) Figure 7,8,9 - Velocity Trajectory Comparison for Train and Test Sets w.r.t Time
3) Figure 10,11,12 - Position Trajectory Comparison for Train and Test Sets w.r.t Time

B. Path Comparison in 3D

Map 1 - Train Set

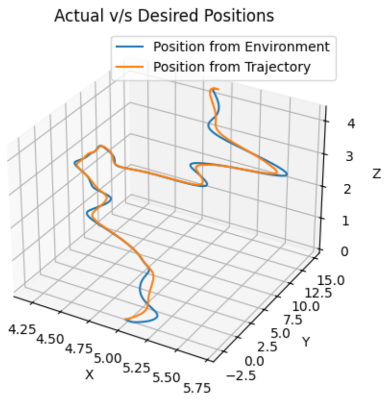

Fig. 4: Comparison of Actual Vs Desired path for Map 1

Map 2 - Test Set

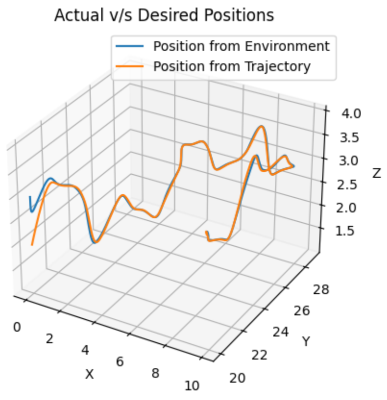

Fig. 5: Comparison of Actual Vs Desired path for Map 2

Map 3 - Test Set

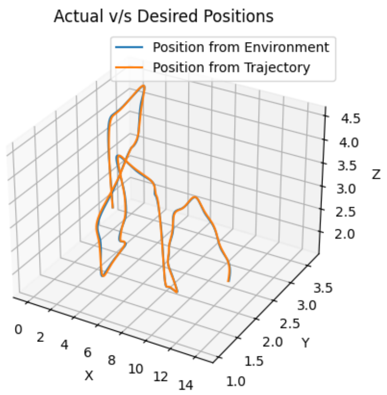

Fig. 6: Comparison of Actual Vs Desired path for Map 3

C. Velocity Trajectory Comparison

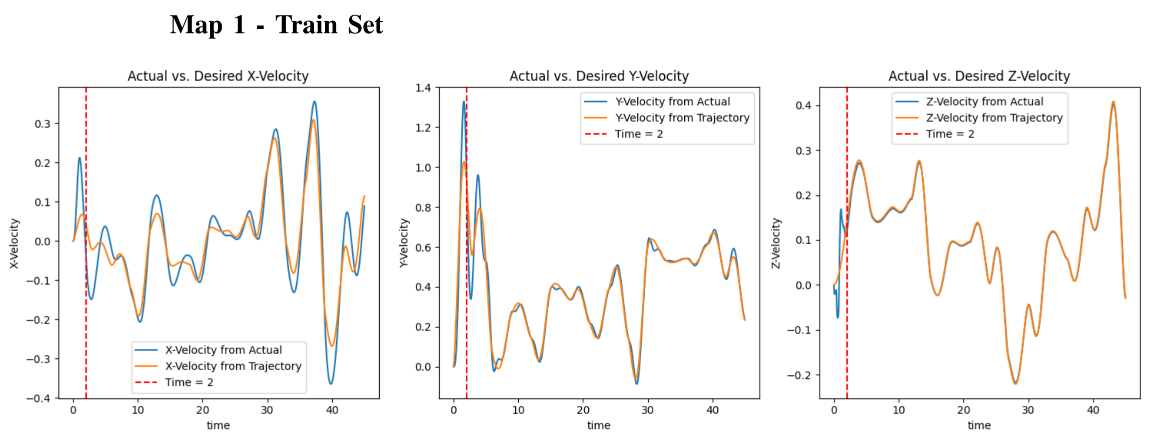

Fig. 7: Comparison of Actual Vs Desired Velocities for Map 1

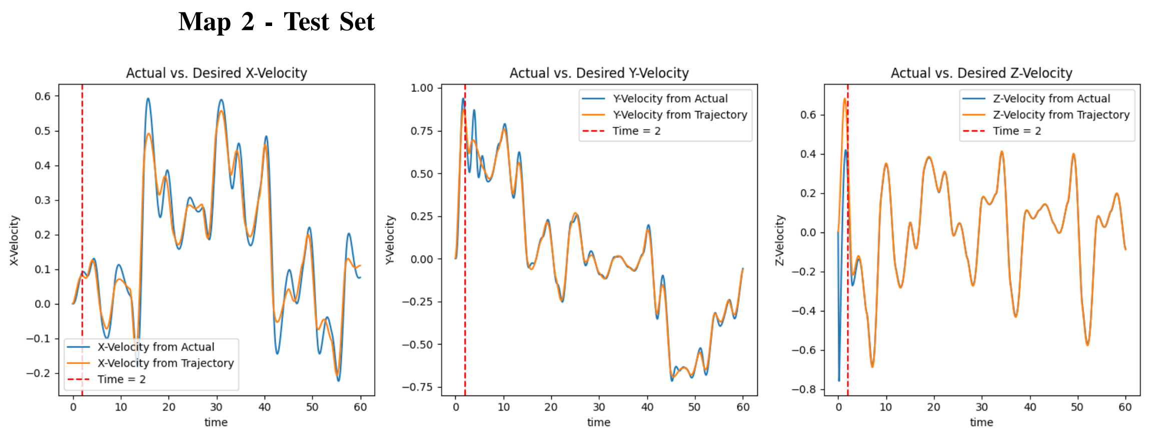

Fig. 8: Comparison of Actual Vs Desired Velocities for Map 2

Map 3 - Test Set

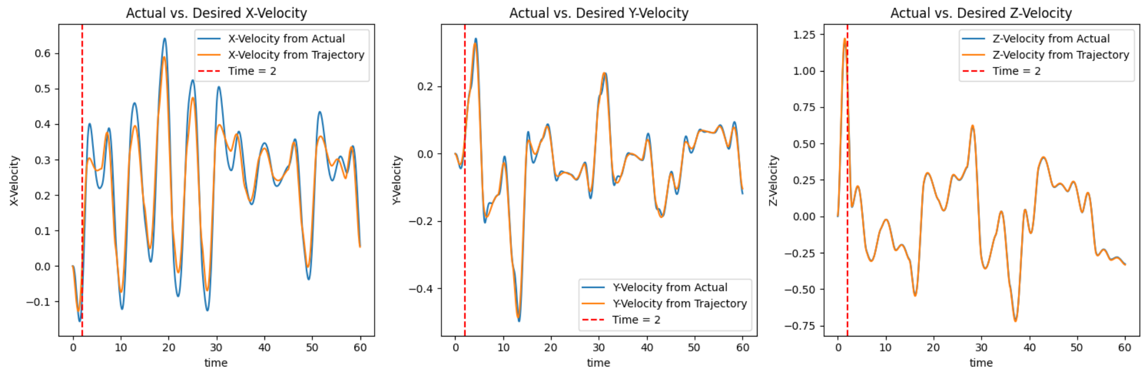

Fig. 9: Comparison of Actual Vs Desired Velocities for Map 3

D. *Position Trajectory Comparison*

Map 1 - Train Set

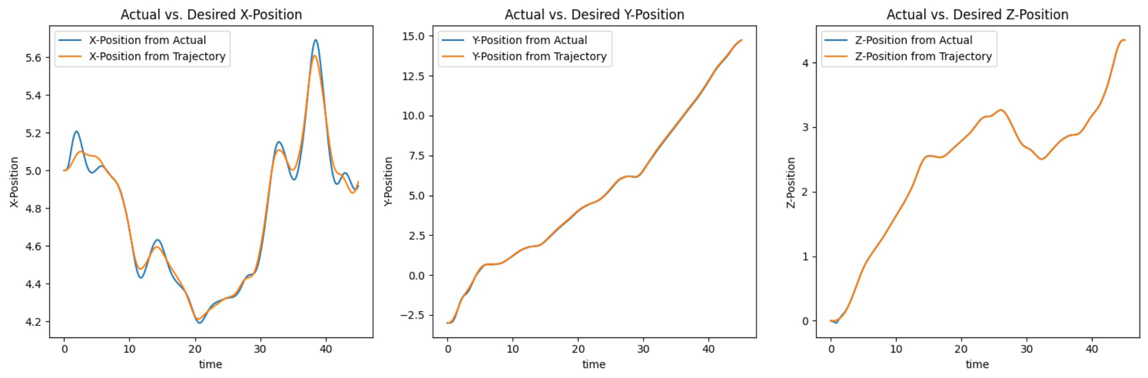

Fig. 10: Comparison of Actual Vs Desired Positions for Map 1

Map 2 - Test Set

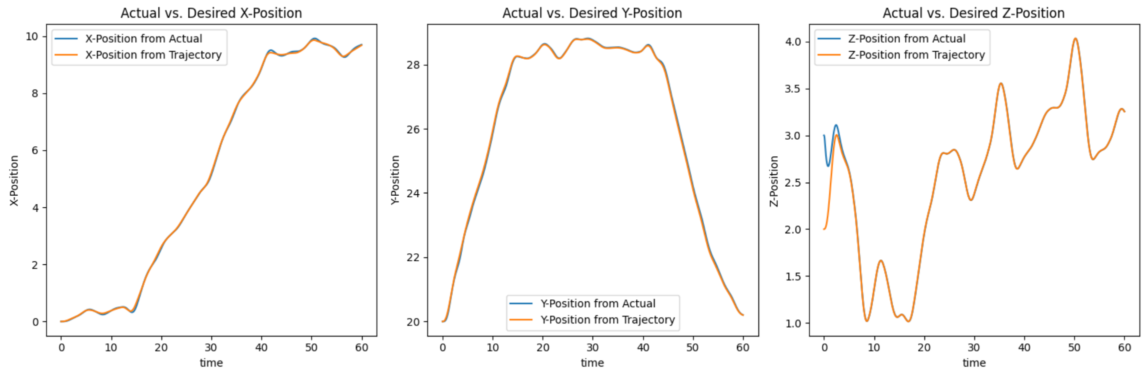

Fig. 11: Comparison of Actual Vs Desired Positions for Map 2

Map 3 - Test Set

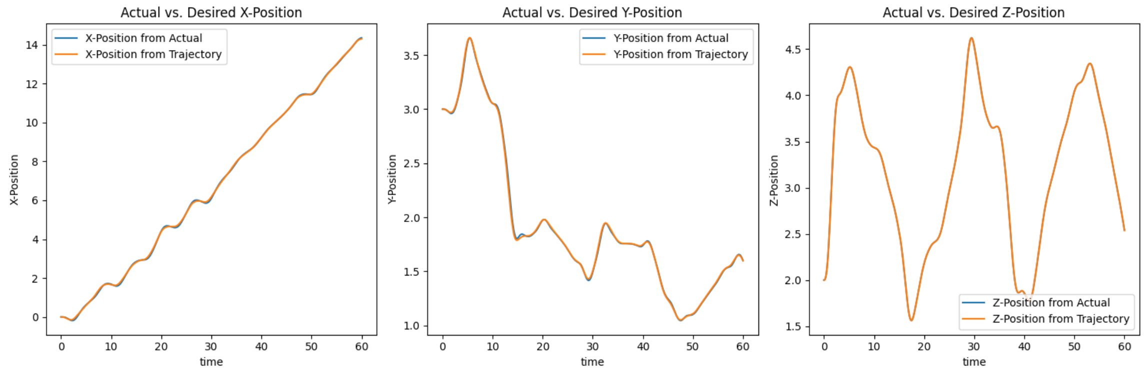

Fig. 12: Comparison of Actual Vs Desired Positions for Map 3

VII. Observations

1) PID Values only work for a range of velocity values and they are tedious to tune. More Robust control strategies possible for drone flight control should be employed. That is currently if we want the drone to go at higher velocities, need to fine-tune the PID Values again.
2) This drone assumes a perfect knowledge of the environment which is seldom the case for real robots. Therefore, the current iteration of this implementation realies too much on the perception module to be perfect. It has to be improved to take into consideration a certain amount of uncertainty.

VIII. Conclusion

Therefore, a simple Motion Planning and Controls Module of Drone was implemented in a simulation environment(Blender) and the following things were implemented :

- A Map reader Function using Blender API.
- Motion Planning in 3D Environment using RRT* with Collision Checking
- Trajectory Generation using Quintic Splines.

IX. Acknowledgment

The author would like to thank Prof. Nitin Sanket and the TA of this course RBE595.

References

[1] Karaman, Sertac, and Emilio Frazzoli. "Sampling-based algorithms for optimal motion planning." The international journal of robotics research 30, no. 7 (2011): 846-894. **Link**
[2] Blender Plot API. **Link**