Tree Planning Through The Trees!
Team Nimbus Navigators
One Late Day

Chaitanya Sriram Gaddipati
Department of Robotics Engineering
Worcester Polytechnic Institute
Worcester, Massachusetss 01609
Email: cgaddipati@wpi.edu

Ankit Talele
Department of Robotics Engineering
Worcester Polytechnic Institute
Worcester, Massachusetss 01609
Email: amtalele@wpi.edu

Shiva Surya Lolla
Department of Robotics Engineering
Worcester Polytechnic Institute
Worcester, Massachusetss 01609
Email: slolla@wpi.edu

***Abstract*—In this project, we implement path planning for a quadcopter using the RRT* algorithm. Waypoints generated from the path planner serve as input for trajectory planning, which aims to generate a minimum acceleration trajectory. Subsequently, a PID controller is implemented to navigate the quadcopter from its starting position to its intended goal along the generated trajectory within a pre-mapped environment.**

I. INTRODUCTION

Autonomous navigation using quadcopters demands robust path planning to determine an optimal or near-optimal route, effective trajectory planning to ensure smooth motion, and dependable control to execute the planned motion accurately. In this study, we present a comprehensive pipeline for the autonomous navigation of quadcopters within a pre-mapped environment peppered with obstacles. All visualizations, from path planning to control execution, are rendered using Blender, providing an intuitive visual perspective on the navigation processes. Our methodology commences with the RRT* algorithm, a well-regarded sampling-based path planning method. Through Blender visualizations, we can intricately observe the tree expansion during the planning phase, granting insights into the path's formation. With the path determined, our focus shifts to trajectory planning, aiming to generate a route that not only follows the waypoints but is also dynamically feasible, so as to make the quadrotor smoothly transition between waypoints and ensure stability. The culmination of our approach lies in the implementation of a Proportional-Integral-Derivative (PID) controller. The PID controller ensures the quadcopter faithfully follows the plotted trajectory, adjusting in real-time to discrepancies between the desired and actual flight paths. Our study showcases the quadcopter's navigation capabilities through Blender visualizations on various test maps, highlighting the accuracy and adaptability of our approach across different scenarios.

II. MAP READER

Our autonomous navigation process fundamentally hinges on the precise representation and understanding of the environment. To this end, our approach incorporates an **Environment** class that serves as the foundational block in reading, processing, and graphically rendering the 3D space wherein the quadcopter operates. This class parses a descriptive file which encodes key environment parameters like spatial boundaries and obstacles. In the interest of navigation safety, each block undergoes a 'bloating' process. This introduces a safety margin around the obstacle, preventing the quadcopter from venturing too close and risking potential collisions. The 3D environment is digitally mapped within a map called as **map_array** - a strategic array where '0's signify obstacles and '1's symbolize free spaces. This array serves as the digitized terrain for the RRT* algorithm.

III. RRT* PATH PLANNER

The RRT* algorithm, tailored for a 3D environment, provides a sophisticated method for pathfinding amidst obstacles. At its core, each node in the algorithm contains three spatial coordinates (x, y, z), a reference to its parent node, and an accumulated cost from the origin. The map array is initialized which is taken from the map reader. Accompanying this, a starting node and a destination node are defined, with the former being placed within a list of vertices set for expansion. Prior to each search iteration, the map is refreshed, situating the starting node appropriately.

A fundamental component of the algorithm is its distance measurement technique, leveraging the Euclidean distance formula. For collision checking, the Bresenham's line algorithm, has been adapted to the 3D context. Through this, it checks for potential collisions between two points by evaluating every intermediary point on path.

As the algorithm iterates, it either selects the goal node or crafts a random point within the 3D space. The selection depends on a predefined goal bias, ensuring a delicate balance between exploration and direct pathfinding.

Once a point is decided upon, the algorithm identifies the nearest existing node within its current vertices. From this node, it then "steers" towards the random point. If this point is distant beyond a fixed range, a new intermediate node is calculated in its direction. However, if it's closer, the point

itself gets selected. A noteworthy feature of RRT* is its ability to identify neighbors of a given node within a set radius. This forms the basis of its 'rewire' function, a mechanism designed to optimize the path. Essentially, this function evaluates if a newly added node can offer a shorter route to its neighboring nodes. If a shorter, obstacle-free path is detected, it promptly "rewires" or updates the parentage of the nodes in question, ensuring that the evolving path is not just valid, but also cost-efficient.

In essence, RRT* emerges as an innovative tree-based pathfinding algorithm. By judiciously sampling random points in a 3D landscape, expanding purposefully in their direction, and continuously refining its tree structure it returns a path. The path formed consists of waypoints leading from the start to the goal.

The tree expansion is visualized in blender where the sampled nodes are visualized as spheres connected by cylinders in blender.

Algorithm 1 RRT* Algorithm

Data: n_pts, $neighbor_size$, $goal_sample_probability$, $steer_distance$, $neighbor_radius$

Result: Path or empty list

```
1  Procedure RRT_star(n_pts, neighbor_size)
2    Initialize map  for sample_number = 1 to n_pts do
3      random_sample ←
         get_new_point(goal_sample_probability)

       nearest_node, best_dist ←
         get_nearest_node(random_sample)

       sample_node ← steer(
         random_sample, nearest_node, steer_distance)

       neighbors ← get_neighbors(
         sample_node, neighbor_radius)

       best_neighbor ← GetBestNeighbor(neighbors)
       if check_collision(sample_node, best_neighbor)
       then
4        sample_node.parent ← best_neighbor
5      rewire(sample_node, neighbors)
       if distance to goal is small then
6        found ← True
7    if found then
8      path ← ConstructPath(goal)  return path
9    else
10     print("No path found")  return []
```

IV. TRAJECTORY GENERATION

The waypoints generated from the path planner and taken for trajectory generation. The goal is to generate a smooth trajectory between waypoints that minimizes acceleration. We use cubic polynomials to represent the trajectory between two waypoints. The x, y, z coordinates are represented as below as a function (f) of time(t) between the waypoints.

$$f(t) = a_0 + a_1 t + a_2 t^2 + a_3 t^3$$

To determine the coefficients $a_0, \ldots, a_3$, we impose constraints based on the given waypoints and the desire to have a smooth trajectory.

Constraints

Given waypoints $p_0, p_1, \ldots, p_n$ at times $t_0, t_1, \ldots, t_n$:

1. The trajectory starts at p_0 at t_0:

$$f(t_0) = p_0$$

2. The trajectory goes through each waypoint:

$$f(t_i) = p_i, \quad i = 1, \ldots, n-1$$

3. The trajectory ends at p_n at t_n:

$$f(t_n) = p_n$$

4. The velocity is continuous at each waypoint:

$$f'(t_i^-) = f'(t_i^+), \quad i = 1, \ldots, n-1$$

5. The acceleration is continuous at each waypoint:

$$f''(t_i^-) = f''(t_i^+), \quad i = 1, \ldots, n-1$$

Using these constraints, we can set up a system of equations to determine the polynomial coefficients.

Matrix Formulation

We express the system of equations in matrix form $Ax = b$, where:

- A is the matrix derived from evaluating the cubic polynomial and its derivatives at the given waypoints and times.
- x is the vector of polynomial coefficients we want to find.
- b is the vector containing the waypoints and derived conditions.

To find the coefficients x, we used the pseudo inverse functionality of numpy:

$$x = A^\dagger b$$

Once the desired coefficients are obtained for the x, y and z coordinates as a function of time between every two waypoints, we now have the curve equations between two waypoints, which can be used to obtain the positions, velocities and accelerations at each instant of time between the, leading to a dynamically feasible trajectory between the waypoints.

PID CONTROLLER

Once we have the trajectory generated, we have the desired coordinates at each instance of time. We then tuned a cascaded controller inspired from the PX4 Stack for the quadrotor to follow the desired trajectory.

A total of 6 gains have been tuned with 3 gains for the position control loop and 3 gains for the velocity control loop.

Position Control Parameters

For the position control, the PID gains havebeen tuned to:

$$\begin{aligned}\text{For } x:&\\ P_x &= 1.2,\\ I_x &= 0.55,\\ D_x &= 0.05,\end{aligned}$$

$$\begin{aligned}\text{For } y:&\\ P_y &= 1.2,\\ I_y &= 0.55,\\ D_y &= 0.5,\end{aligned}$$

$$\begin{aligned}\text{For } z:&\\ P_z &= 2.5,\\ I_z &= 0.7,\\ D_z &= 0.5.\end{aligned}$$

Velocity Control Parameters

For velocity control, the PID gains are:

$$\begin{aligned}\text{For } v_x:&\\ P_{v_x} &= 1.5,\\ I_{v_x} &= 0.00,\\ D_{v_x} &= 0.15,\end{aligned}$$

$$\begin{aligned}\text{For } v_y:&\\ P_{v_y} &= 1.5,\\ I_{v_y} &= 0.08,\\ D_{v_y} &= 0.01,\end{aligned}$$

$$\begin{aligned}\text{For } v_z:&\\ P_{v_z} &= 4.05,\\ I_{v_z} &= 0.002,\\ D_{v_z} &= 0.1.\end{aligned}$$

With the above parameters the drone follows the desired trajectory effectively thereby completing the implementation of our pipeline.

V. CONCLUSION

In this study, we addressed the multifaceted challenge of autonomous quadcopter navigation within obstacle-rich environments, unveiling a cohesive pipeline that integrates the RRT* algorithm, cubic polynomial trajectory generation, and a meticulously tuned PID controller. By employing the RRT* algorithm, we achieved optimal path planning, visualized compellingly via Blender, offering insights into real-time tree expansion. This path, when passed through our trajectory generation phase, ensures dynamic feasibility, epitomizing the essence of smooth, safe navigation. Our PID controller, assures adherence to the plotted trajectory, rectifying any deviations. These intertwined components, tested across four different maps, underscore our approach's adaptability and precision.

REFERENCES

[1] https://theclassytim.medium.com/robotic-path-planning-rrt-and-rrt-212319121378

[2] N. J. Sanket, ”Trajectory and Motion Planning,” RBE595-F02-ST: Hands-On Autonomous Aerial Robotics, Class 8, [Worcester Polytechnic Institute], [2023].

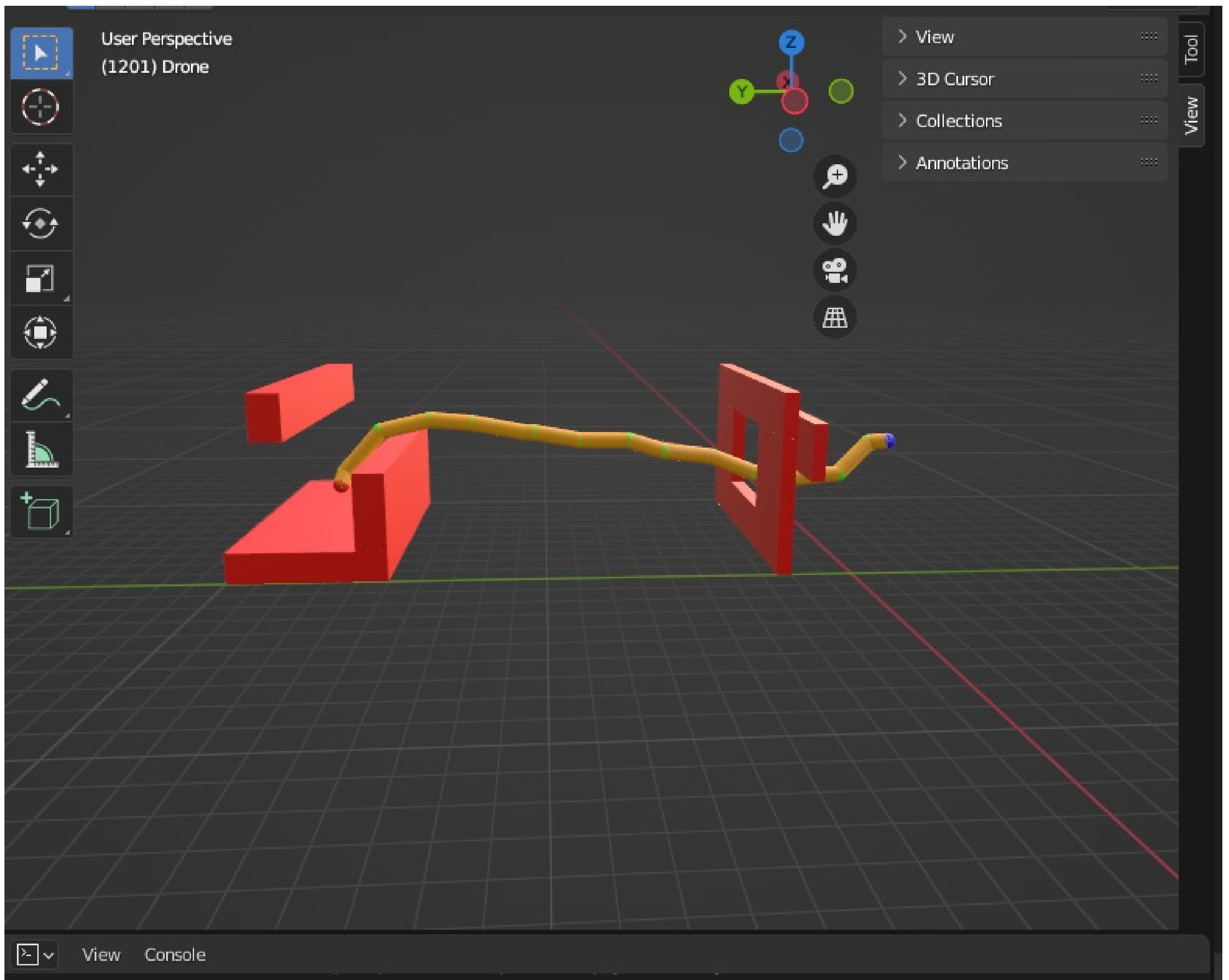

Fig. 1. Trajectory for the first map

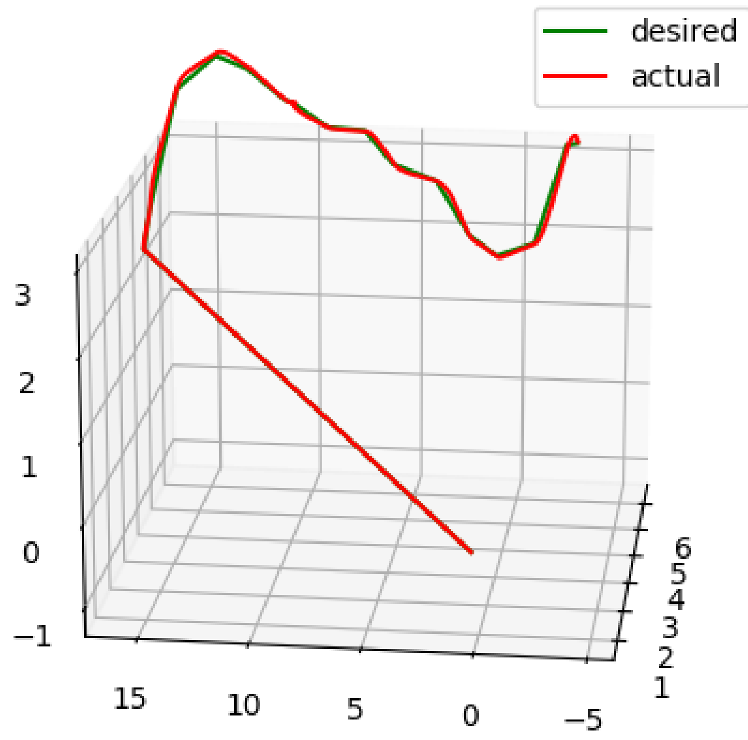

Fig. 2. Plot for the first map

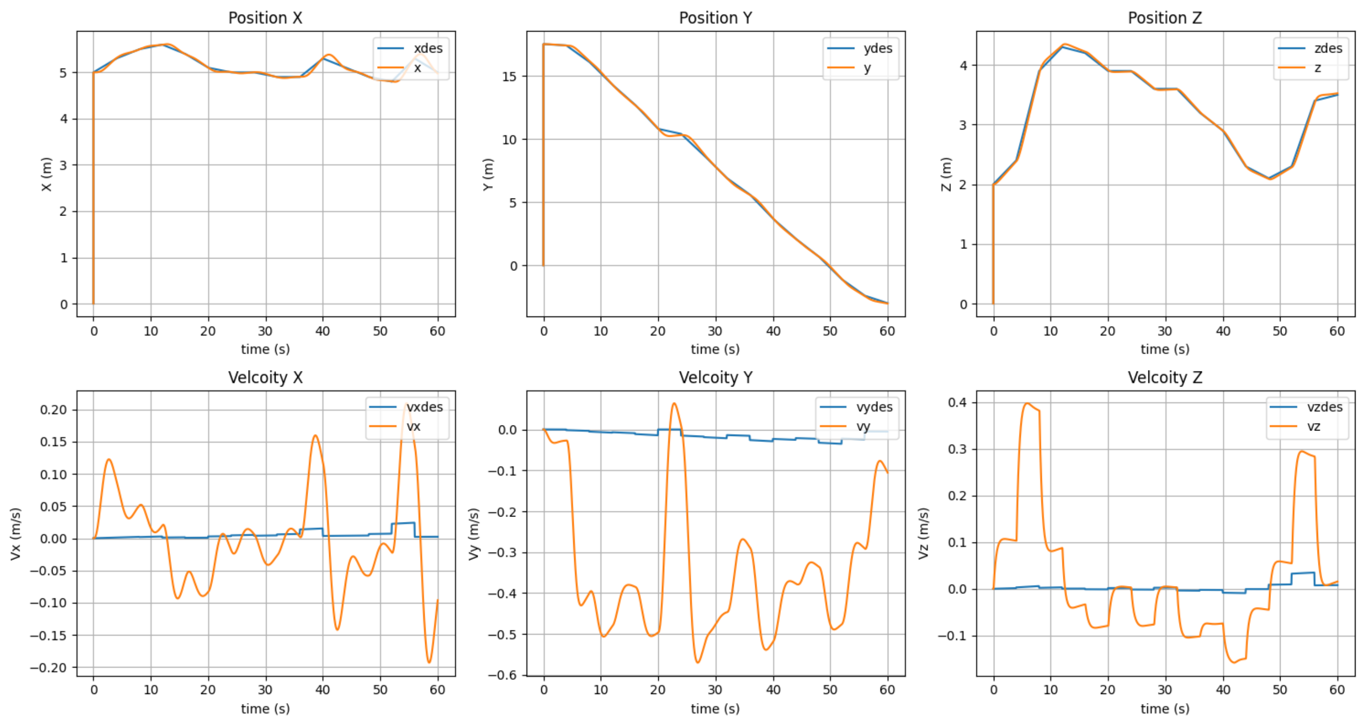

Fig. 3. Control plots for map 1

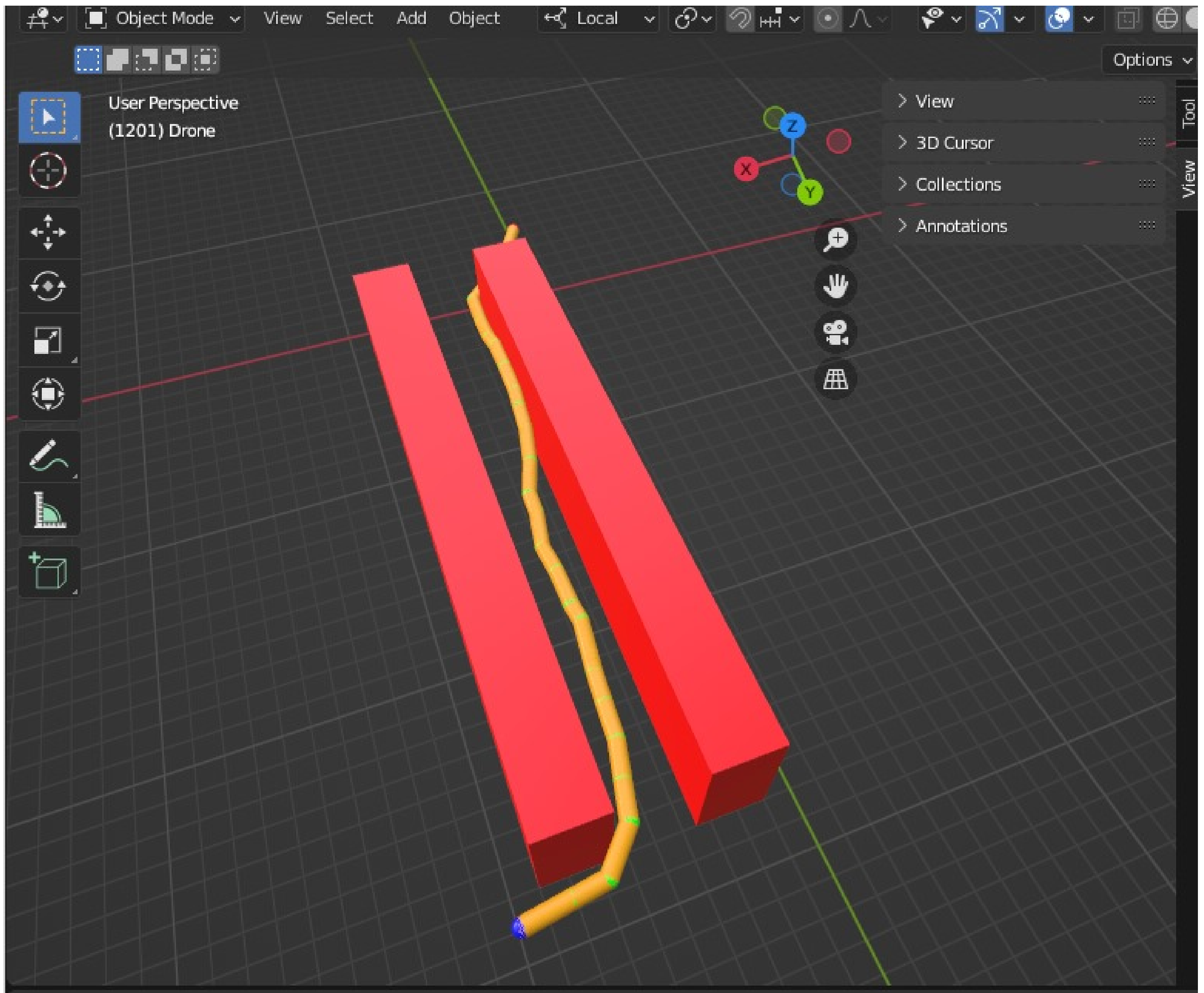

Fig. 4. Trajectory for the second map

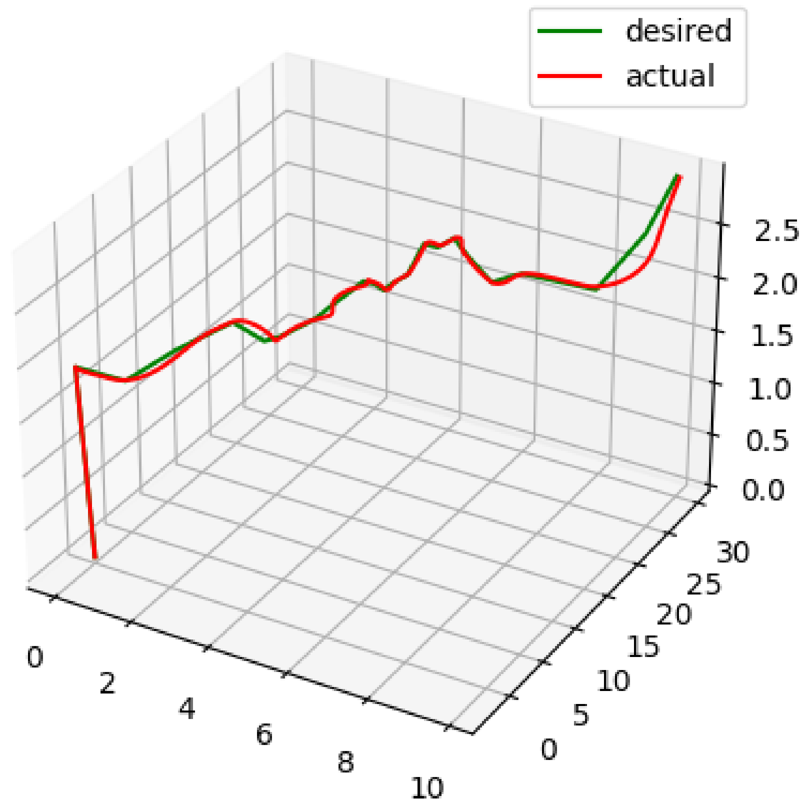

Fig. 5. Plot for the second map

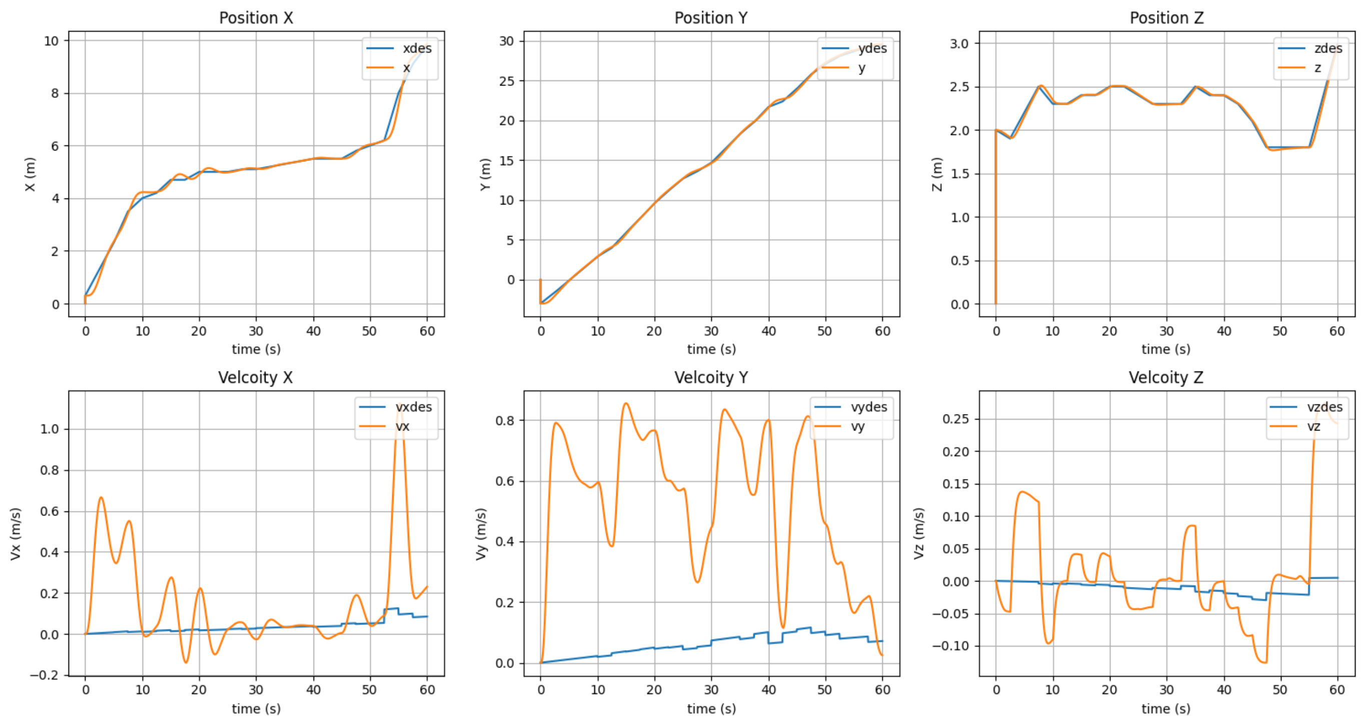

Fig. 6. Control plots for the second map

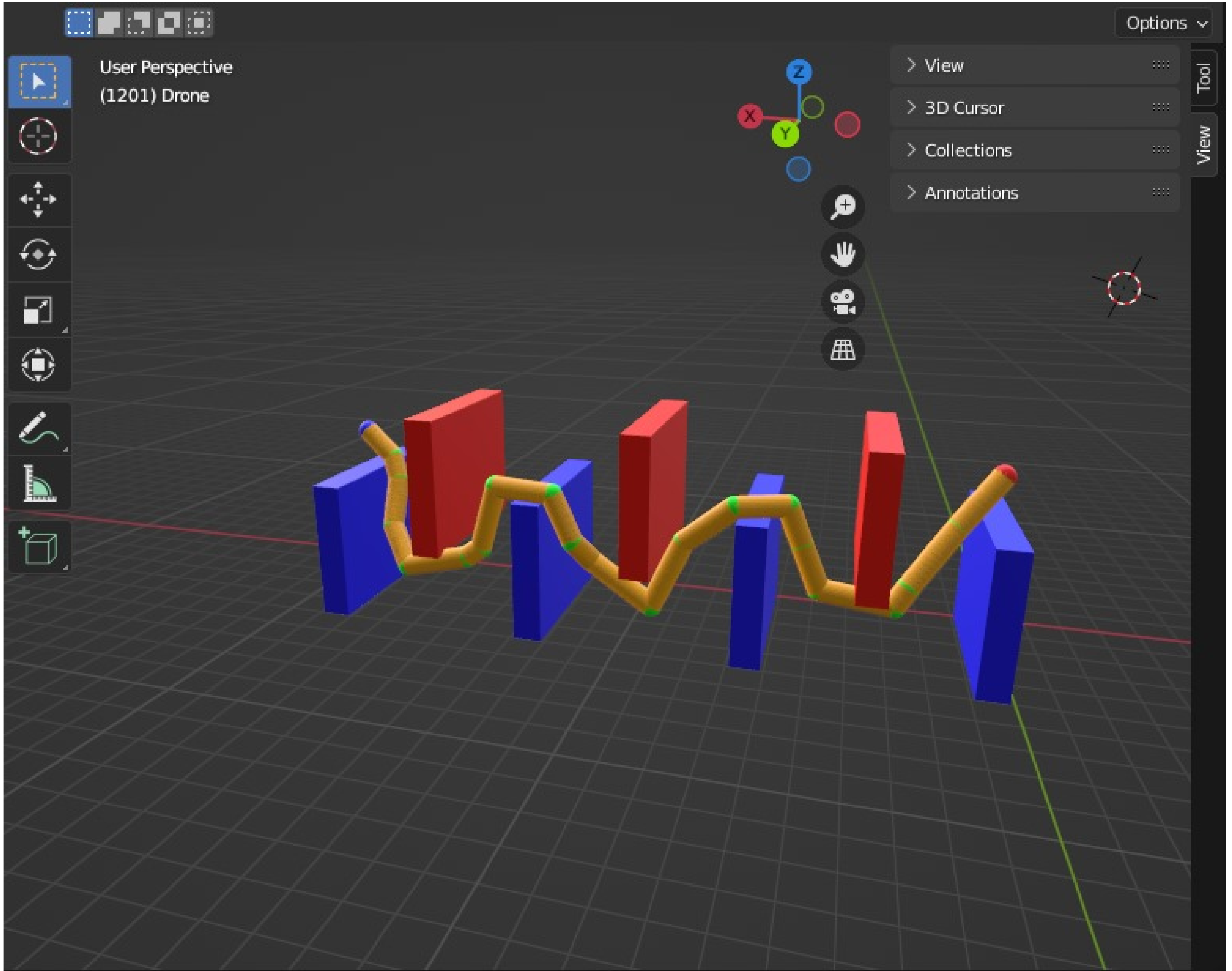

Fig. 7. Trajectory for the third map

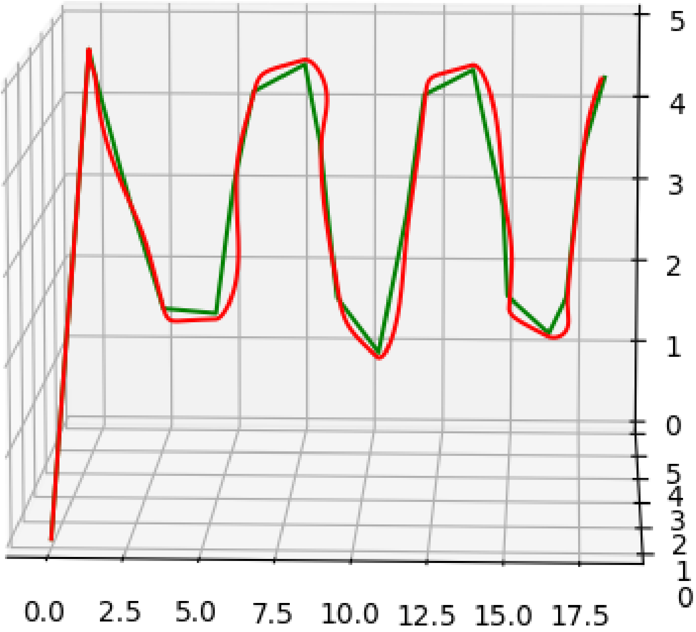

Fig. 8. Plot for the third map

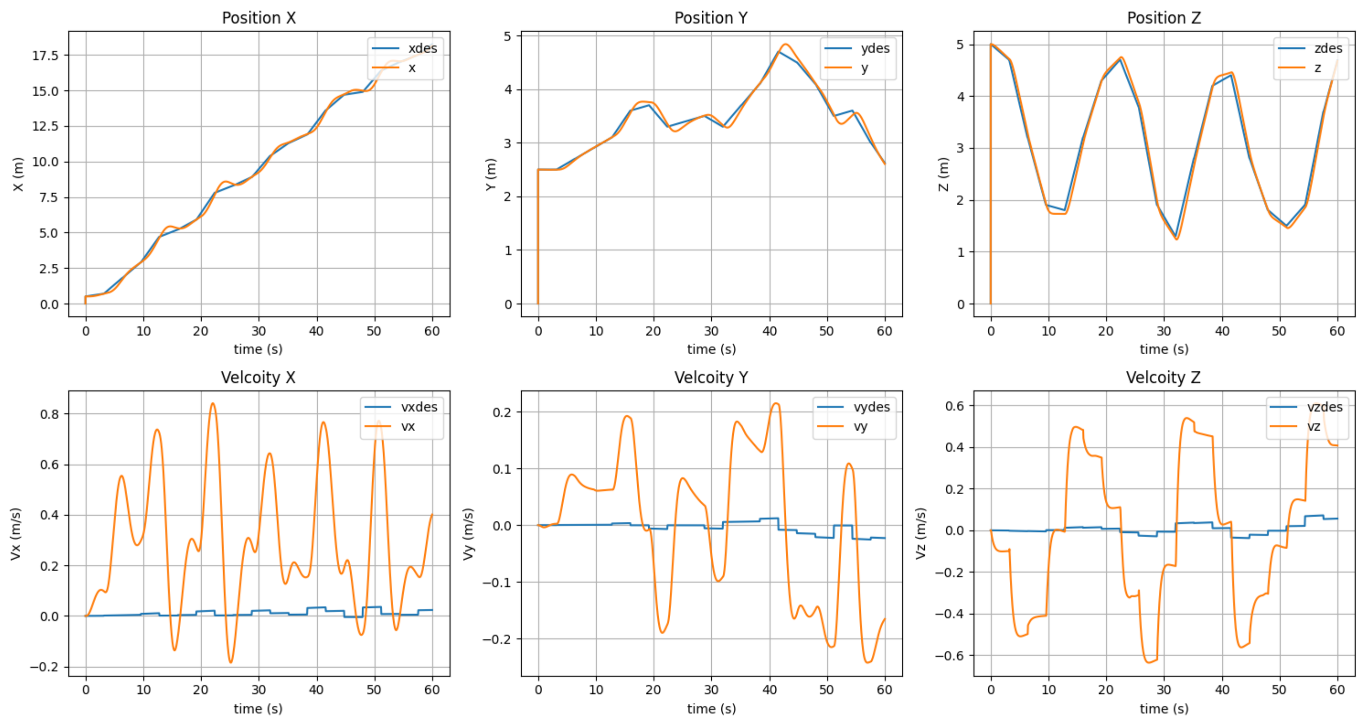

Fig. 9. Control plots for the third map.

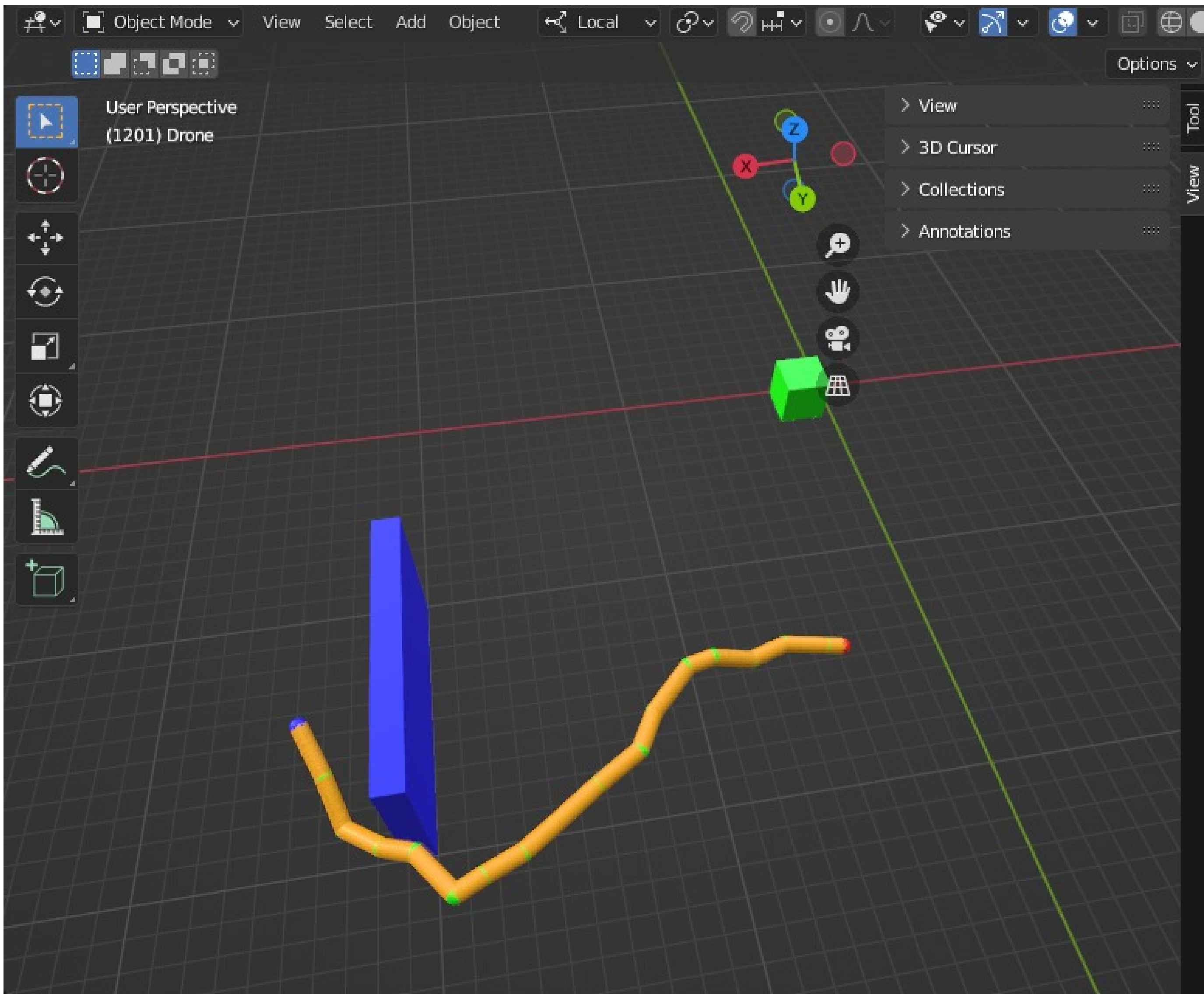

Fig. 10. Trajectory for the fourth map

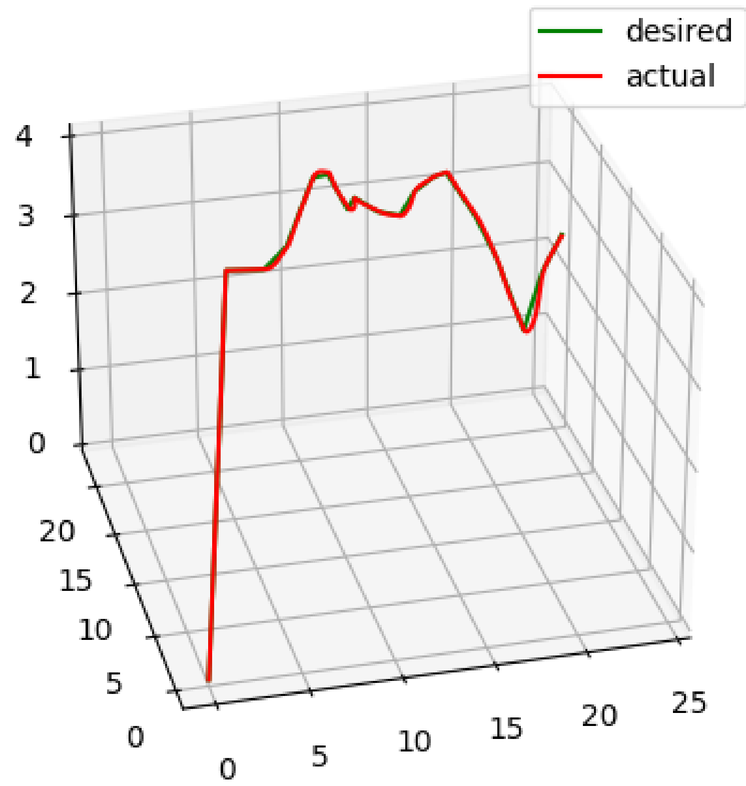

Fig. 11. Plot for the fourth map

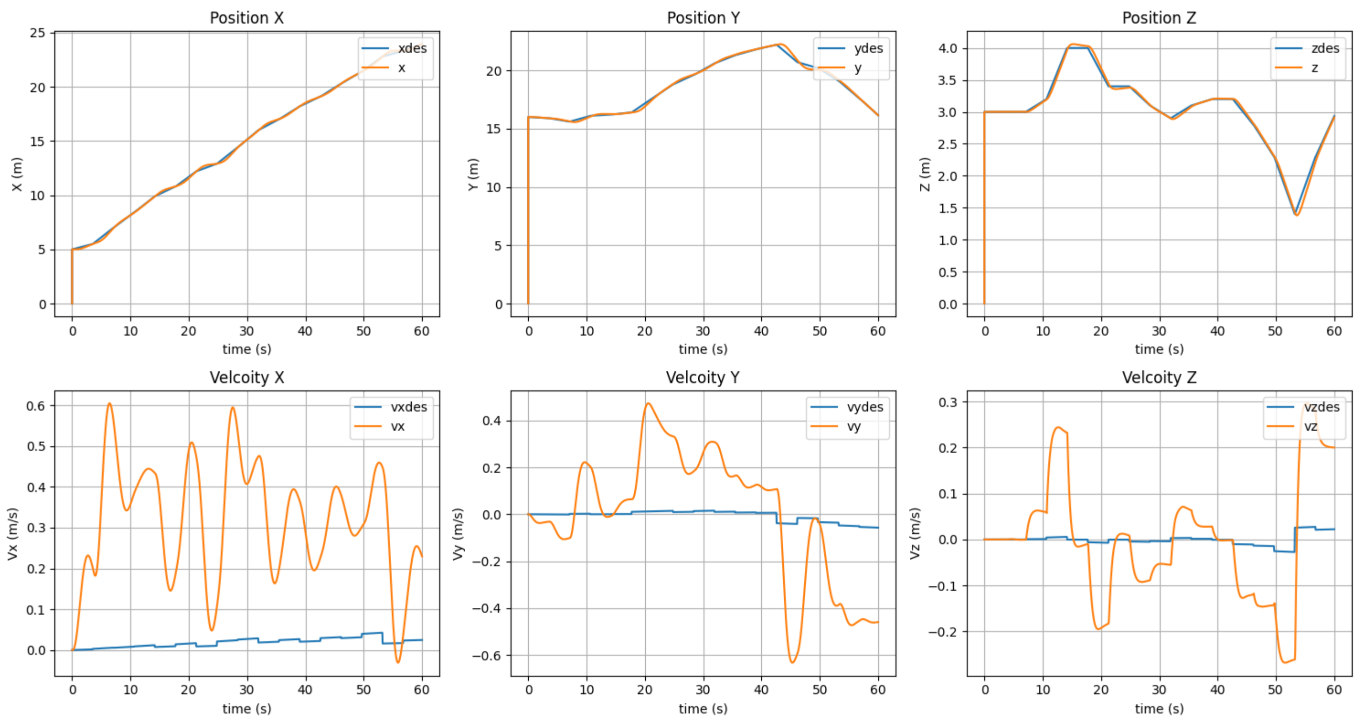

Fig. 12. Control plots for the fourth map.

Fig. 13. Explored tree for fourth map with the path taken in the tree.