P2:Tree Planning Through The Trees!

Shreyas Chigurupati
Department of Robotics Engineering
Worcester Polytechnic Institute
Worcester, Massachusetts 01609

***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 snap 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. The navigation stack developed is run on DJI Tello EDU quadcopter to navigate from a start to goal position along the generated path on a given map.**

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 project, we present a comprehensive pipeline for the autonomous navigation of quadcopters within a pre-mapped environment 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 starts with the RRT* algorithm, a well-regarded sampling-based path planning method. Through Blender visualizations, we can observe the tree expansion during the planning phase, granting insights into the path's formation. Once the RRT* algorithm generates a path of waypoints for a given map, the list of waypoints are used to generate a smooth and dynamically feasible minimum snap trajectory that allows for quadrotor to smoothly transition between waypoints and ensure stability. A cascaded PID controller containing position and velocity control is used to ensure the quadcopter follows the given trajectory, adjusting in real-time to discrepancies between the desired and actual flight paths. This approach is first tested in simulation and is later implemented on a real quadcopter. A NVIDIA Jetson Orin Nano is used as flight controller for Tello drone and to send the position control commands to it.

II. Methodology

A. Map Reader

A text file containing the map which includes the boundary of the environment and the locations and sizes of the obstacles is given. A function read this map and create the environment in blender is created. An **Environment** class is created that is used 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 obstacle is 'bloated' by certain percentage in all directions. 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** - an array where '0's signify obstacles and '1's represent free spaces. RRT* algorithm takes input this 3D array as the map while searching for a path.

B. RRT* path planner

The RRT* algorithm, tailored for a 3D environment, is used to obtain a path from start location to goal location as a list of ordered waypoints. 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 tree set for expansion. Prior to each search iteration, the map is refreshed, situating the starting node appropriately.

In this algorithm we chose the distance metric between two nodes as Euclidean distance. For collision checking, we need two types of checks. The first is to check if a node lies on or inside an obstacle or outside the boundary of the map. This is easy to check since we already have an array for our map. The second collision checking is done to see if a line joining two successive waypoints collides with an obstacle. For this we used a modified Bresenham's line algorithm, that has been adapted to the 3D scenario. It checks for potential collisions between two points by evaluating every intermediary point on path.

As the RRT* algorithm iterates, it either selects the goal node or uses a random point within the 3D space. The selection depends on a predefined goal bias, ensuring a balance between exploration and direct path finding.

Once a point is decided upon, the algorithm identifies the

nearest existing node on the tree 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 parent of the nodes in question, ensuring that the evolving path is not just valid, but also cost-efficient.

The tree expansion is visualized in blender where the sampled nodes are visualized as spheres connected by cylinders in blender. An example visualization for sample map 4 can be seen in fig 1

Algorithm 1 RRT* Algorithm

Data: *n_pts*, *neighbor_size*, *goal_sample probability*, *steer distance*, *neighbor radius*

Result: Path or empty list

Procedure RRT_star(*n_pts, neighbor_size*)
  Initialize map **for** *sample_number* = 1 **to** *n_pts* **do**
    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**
      sample _node.parent ← *best_neighbor*
    rewire(*sample_node*, *neighbors*)
    if *distance to goal is small* **then**
      found ← True
  if *found* **then**
    path ← ConstructPath(*goal*) **return** *path*
  else
    print(”No path found”) **return** []

C. Trajectory generation

The waypoints generated from the path planner are discrete. The goal is to generate a smooth trajectory between waypoints that minimizes snap. A polynomial function of time is used to represent the trajectory between two waypoints. This problem is then represented as an unconstrained quadratic programming optimization problem where the sum of squares of derivatives upto snap are used as cost to obtain the optimal polynomial coeffients for all the trajectories. Additionally continuity and waypoint constraints and inequality constraints are applied while optimizing. For a better understanding of theory and math behind this process refer to [1].

D. PID Controller tuning

Once we have the trajectory generated, we have the desired coordinates at each instance of time. We then tuned a cascaded PID controller with similar architecture to that of PX4(see [2]) 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. Setting all the gains for position controller as zero, the velocity controller gains are tuned first since it is present in the inner loop in the architecture.

The best gains we got are as follows:

$$\text{For } x: K_p = 0.1;\ K_i = 0;\ K_d = 0; \quad (1)$$

$$\text{For } y: K_p = 0.1;\ K_i = 0;\ K_d = 0; \quad (2)$$

$$\text{For } z: K_p = 0.01;\ K_i = 0;\ K_d = 0; \quad (3)$$

$$\text{For } v_x: K_{px} = 2; K_i = 0; K_d = 0.1; \quad (4)$$

$$\text{For } v_y: K_{px} = 2; K_i = 0; K_d = 0; \quad (5)$$

$$\text{For } v_z: K_{px} = 20; K_i = 0.01; K_d = 0.1; \quad (6)$$

With the above parameters the drone follows the desired trajectory effectively thereby completing the implementation of our pipeline for simulation in blender. The trajectory and the position and velocity control values for all the maps are shown in fig 4, fig 6, fig 7, fig 9, fig 10, fig 12, fig 13, fig 15, fig 16, fig 18.

III. Implementation on real quadcopter

Now that we have the trajectory in simulation we use that to run on the real drone.

A. DJI Tello quadcopter

The DJI Tello EDU is a state-of-the-art drone designed with compactness and functionality in mind. It has a weight of approximately 80 grams, inclusive of its propellers and battery, and dimensions of 98×92.5×41 mm. The drone features 3-inch propellers and boasts an array of integrated systems such as a range finder, barometer, LED indicators, an advanced vision system. Additionally, a Micro USB port ensures power and data transfer capabilities. Regarding flight performance, the Tello EDU can traverse up to 100 meters, reach speeds of 8 meters per second, sustain a flight for up to 13 minutes, and achieve a maximum altitude of 30 meters. Its power system is uniquely designed with a detachable 1.1Ah/3.8V battery for extended use and convenience. Images and videos are stored in JPG and MP4 formats, respectively, and the inclusion of Electronic Image Stabilization (EIS) ensures good quality and stable footage.

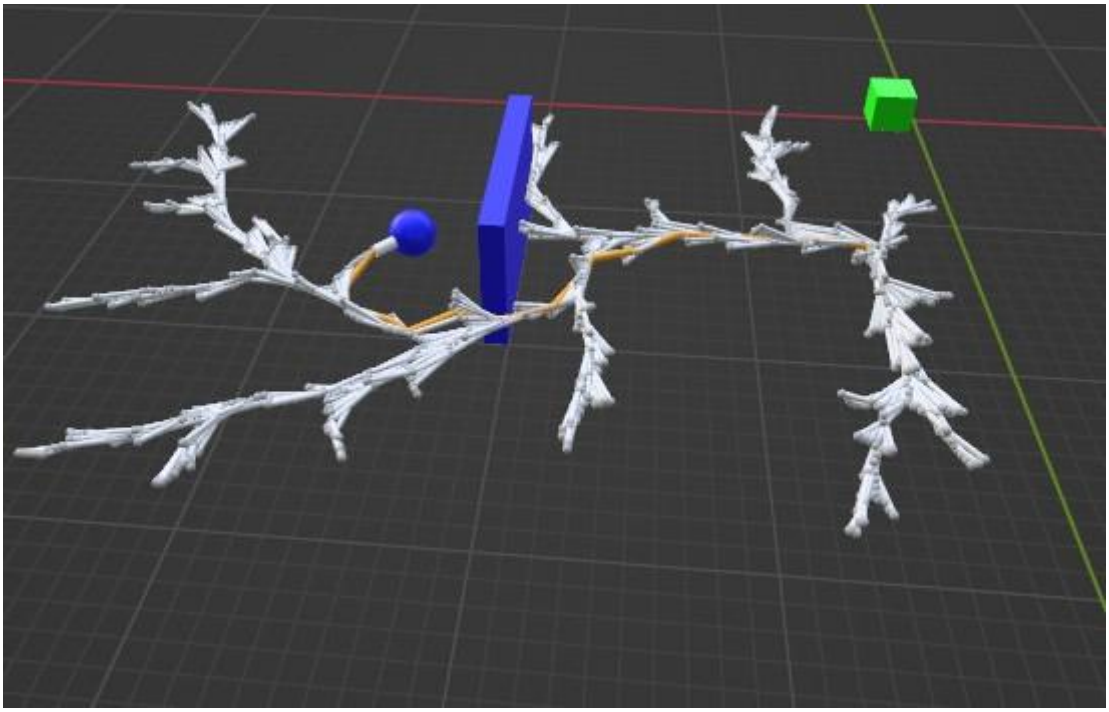

Fig. 1. Explored tree for fourth map with the path taken in the tree.

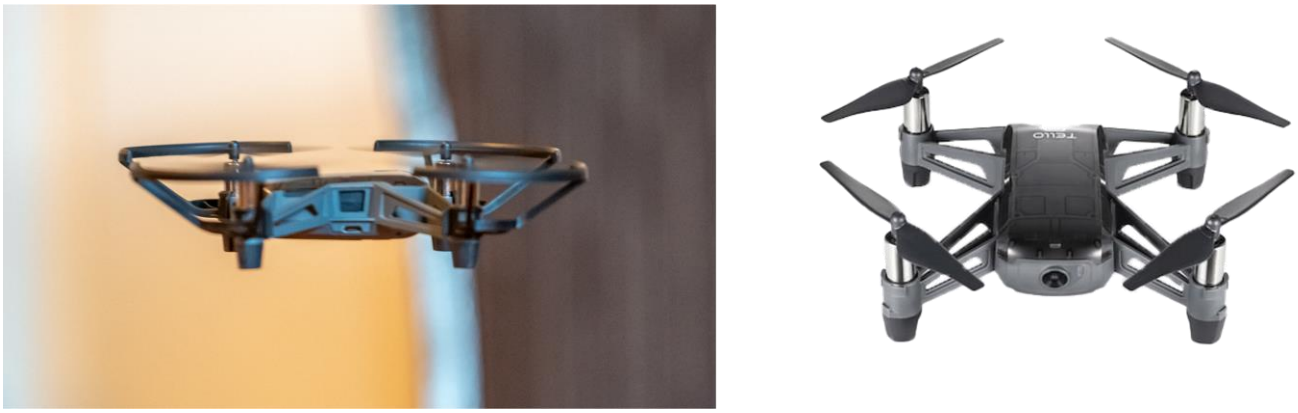

Fig. 2. DJI Tello

B. NVIDIA Jetson Orin Nano

We integrated the NVIDIA Jetson Orin module, a state- of-the-art AI computing system with our DJI Tello EDU quadcopter. This module stands out due to its energy efficiency and great performance. Specifically, the Jetson Orin delivers up to 275 trillion operations per second (TOPS), signifying an eightfold performance enhancement over its preceding generation. Such computational capacity allows for the simultaneous execution of multiple AI inference pipelines, a critical requirement for real-time processing in autonomous systems. Furthermore, it offers robust high-speed interface support, allowing a variety of sensor integrations essential for drone operations.

C. Sim2Real transfer to quadcopter

There are two ways to control the tello drone: 1. Position control and 2. Velocity control. The position control commands are discrete. Velocity control (rc control) can provide continuous velocity values. But after experimentation we saw that the velocity controller is not reliable since we are communicating wirelessly between the orin nano and the drone via UDP architecture. This led to unexpected behaviour from the drone.

So we used position control commands to traverse the obtained path. Once we have the waypoints generated, we have the desired coordinates at each instance of time. We take each of their position coordinates and save them in a csv file. We then used the position control functionality (go_xyz_speed) of Tello in order to send it the saved coordinates in sequence.

Fig. 3. Jetson Orin Nano

This enables the execution of the trajectory on the Tello. The drone follows the desired trajectory effectively in an actual environment whose map (see fig 17 was given to us with desired start and goal positions.

IV. Conclusion

In this project a 3D RRT* algorithm for the navigation of a tello drone is implemented along with the minimum snap trajectory generation in a blender simulation. Once the trajectory is obtained, a sim2real transfer is done onto a real drone and is tested on a real map. We simulated our path and trajectory planning algorithms on multiple maps in simulation and are found to be robust for all types of maps.
However deploying the found trajectory onto the real tello drone was challenging primarily due to the hardware limitations of the tello drone.

References

[1] C. Richter, A. Bry, and N. Roy, "Polynomial trajectory planning for aggressive quadrotor flight in dense indoor environments," in *Robotics Research: The 16th International Symposium ISRR*. Springer, 2016, pp. 649–666.

[2] "Px4 cascaded pid controller," https://docs.px4.io/main/en/flight_stack/controller_diagrams.html.

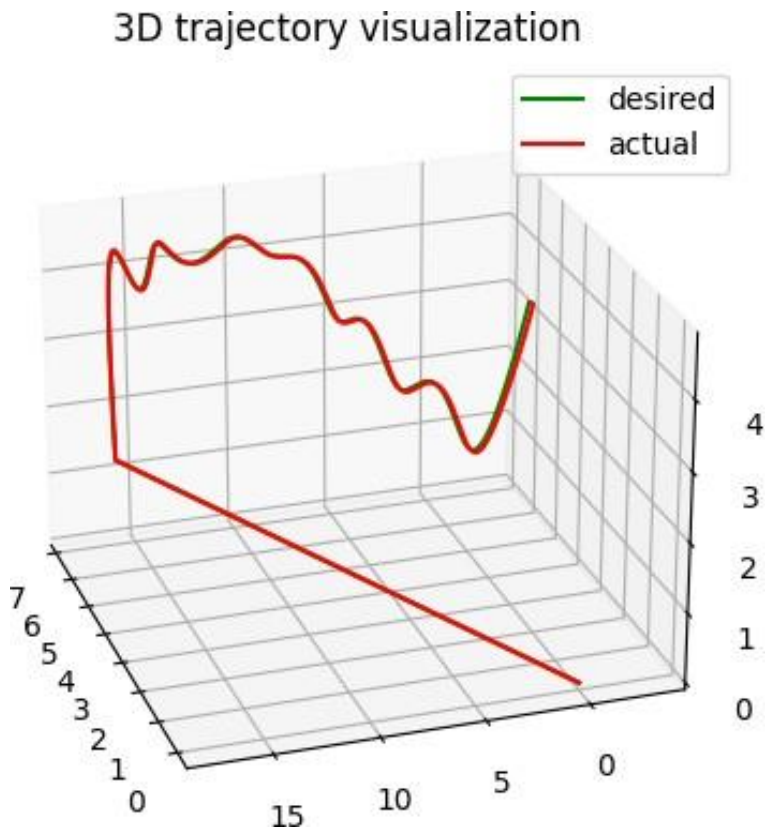

Fig. 4. Trajectory for sample map 1

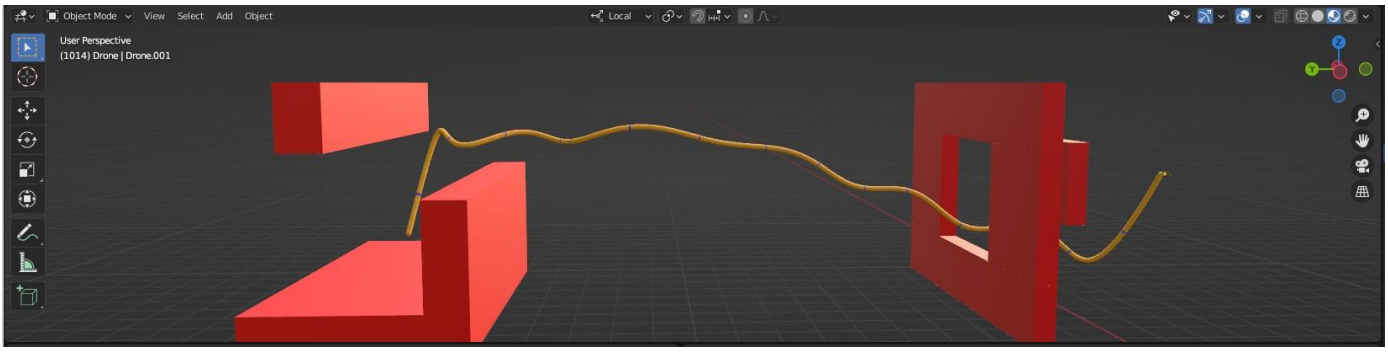

Fig. 5. Blender visualization of sample map 1

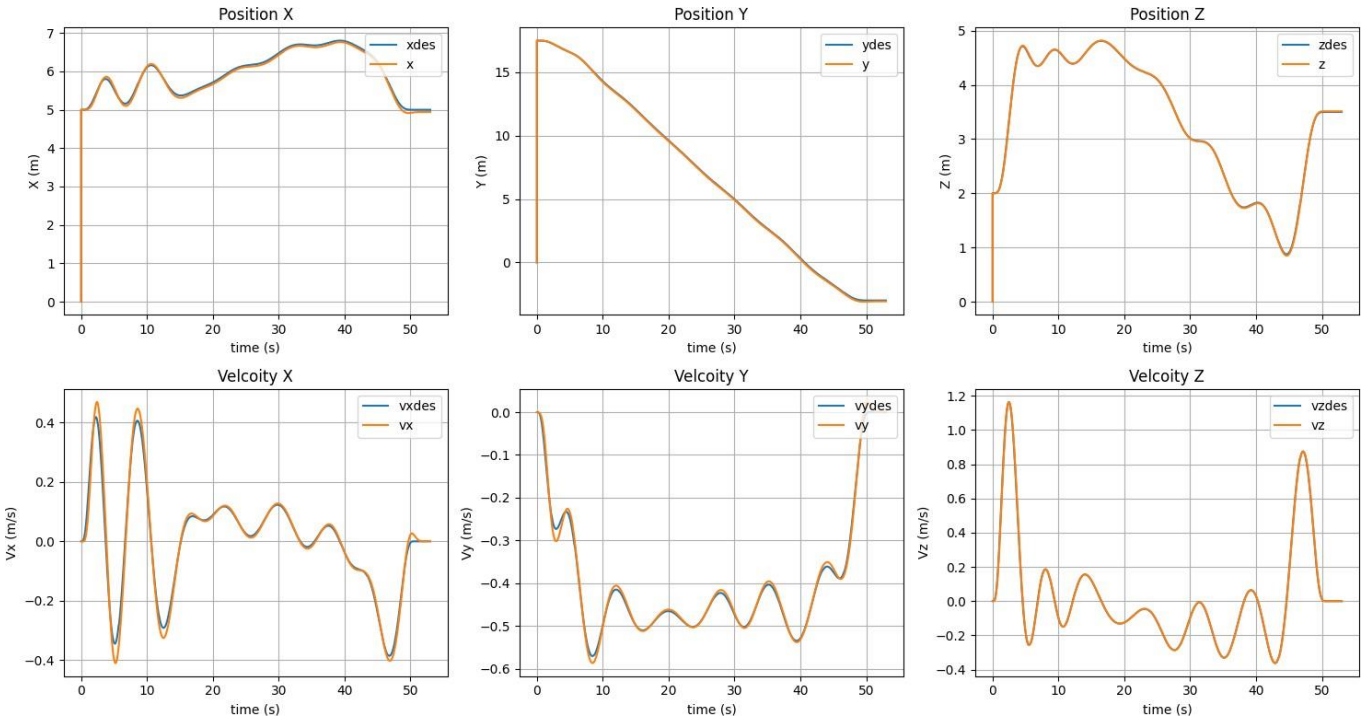

Fig. 6. Position and velocity control plots for sample map 1

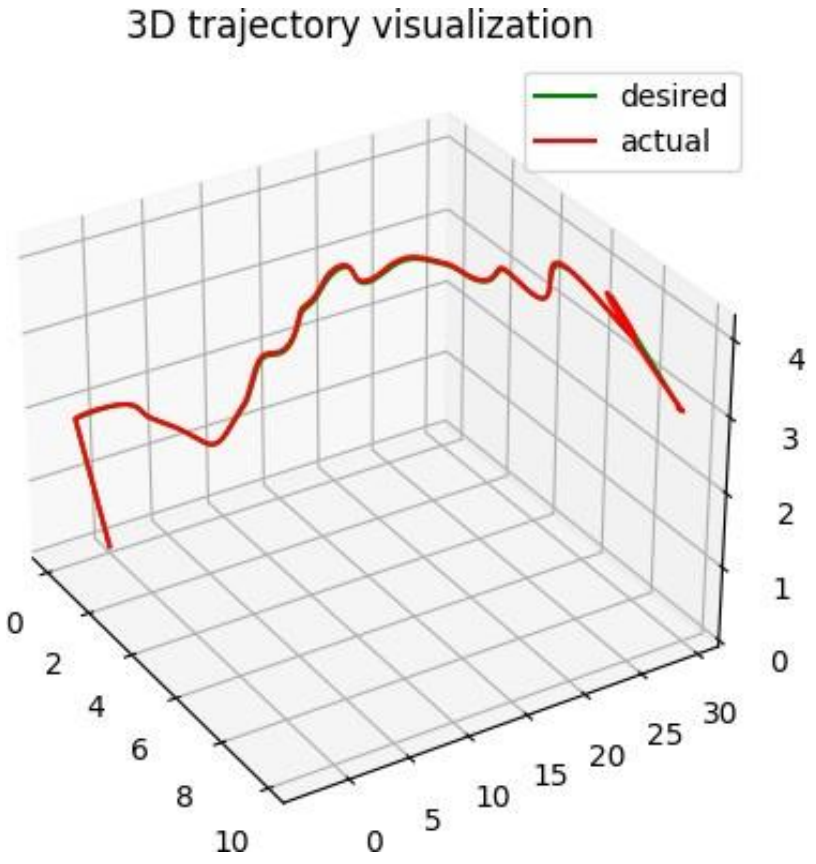

Fig. 7. Trajectory for sample map 2

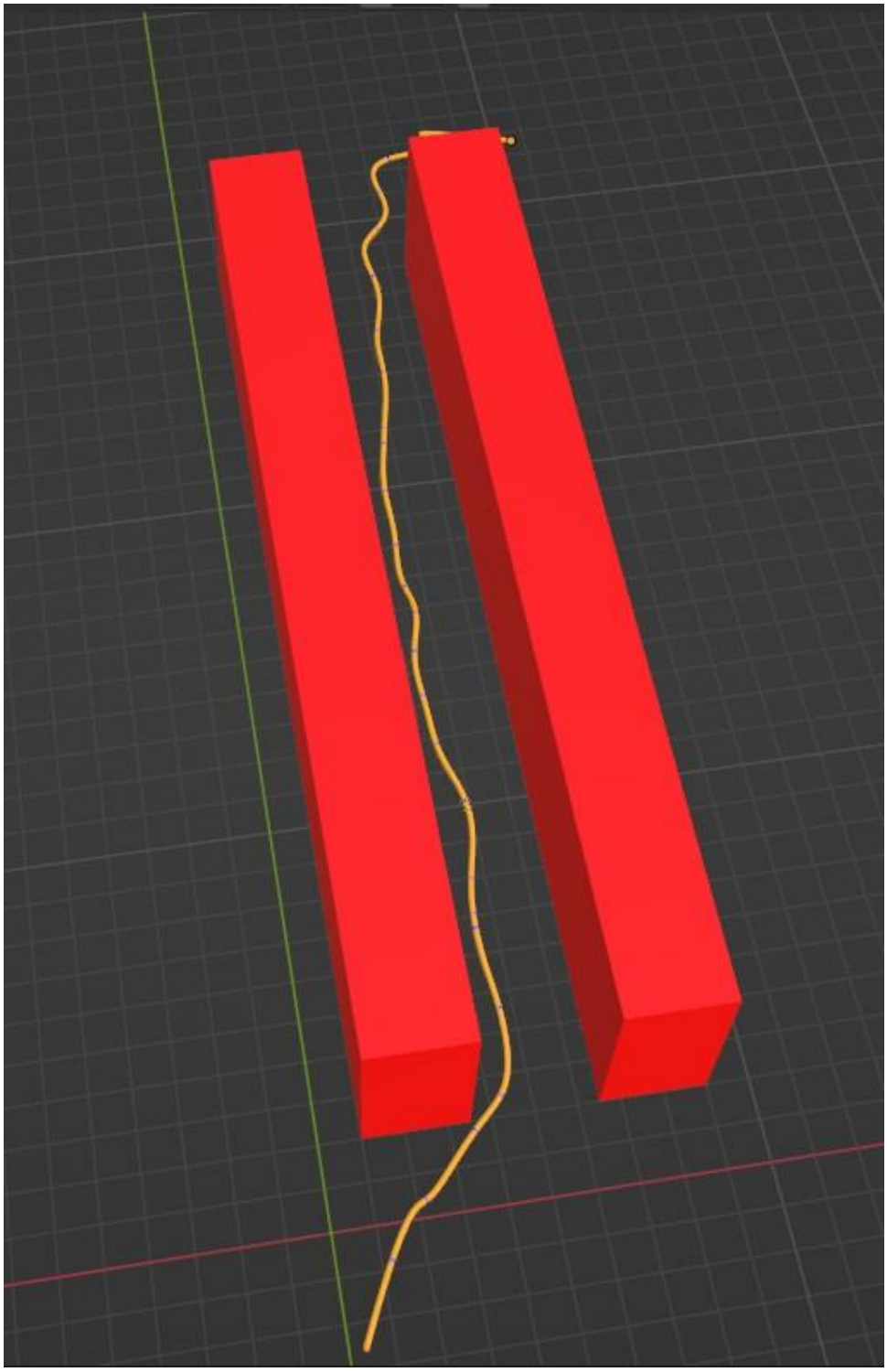

Fig. 8. Blender visualization of sample map 2

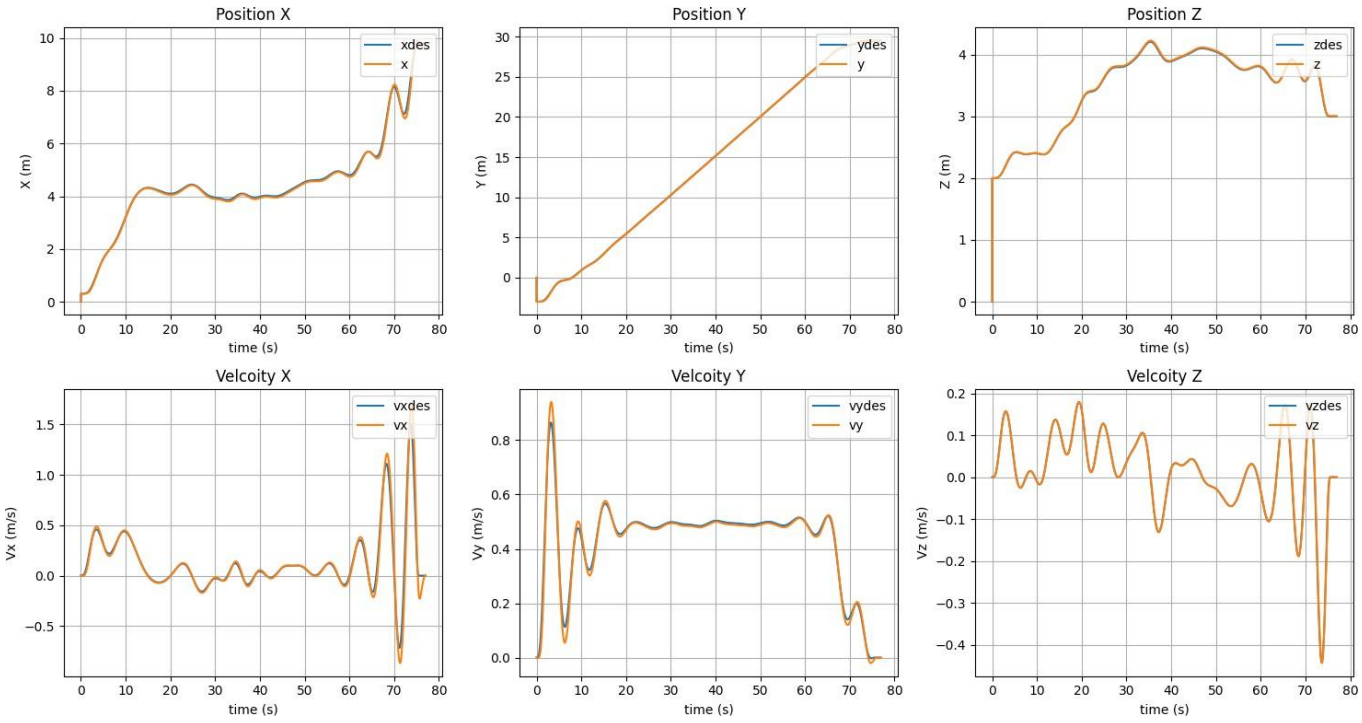

Fig. 9. Position and velocity control plots for sample map 2

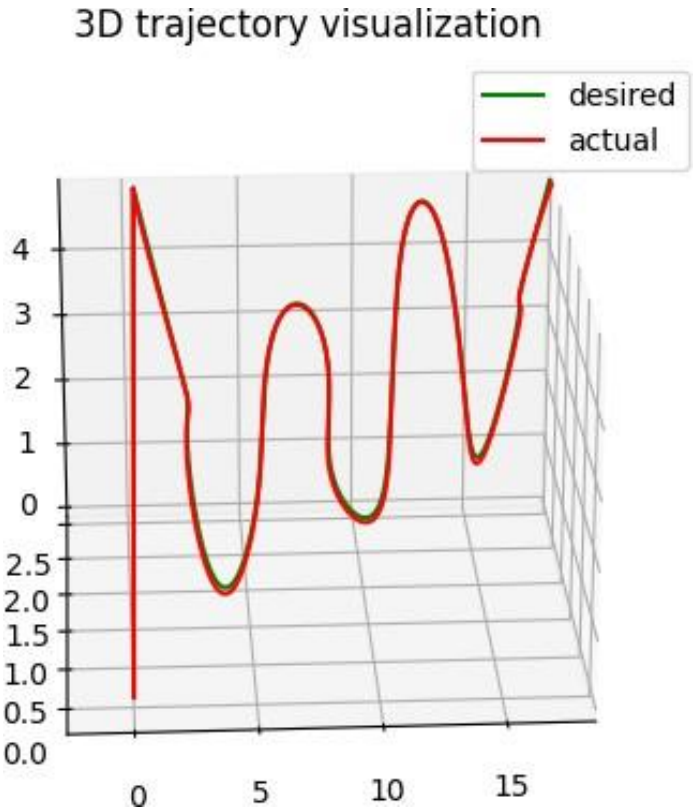

Fig. 10. Trajectory for sample map 3

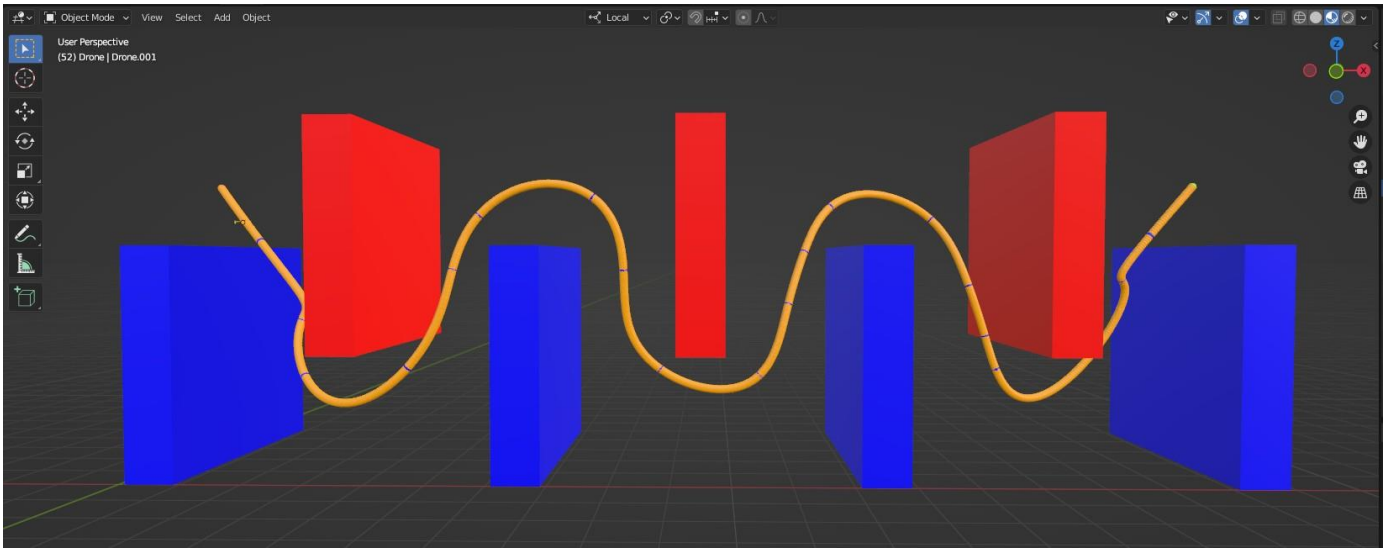

Fig. 11. Blender visualization of sample map 3

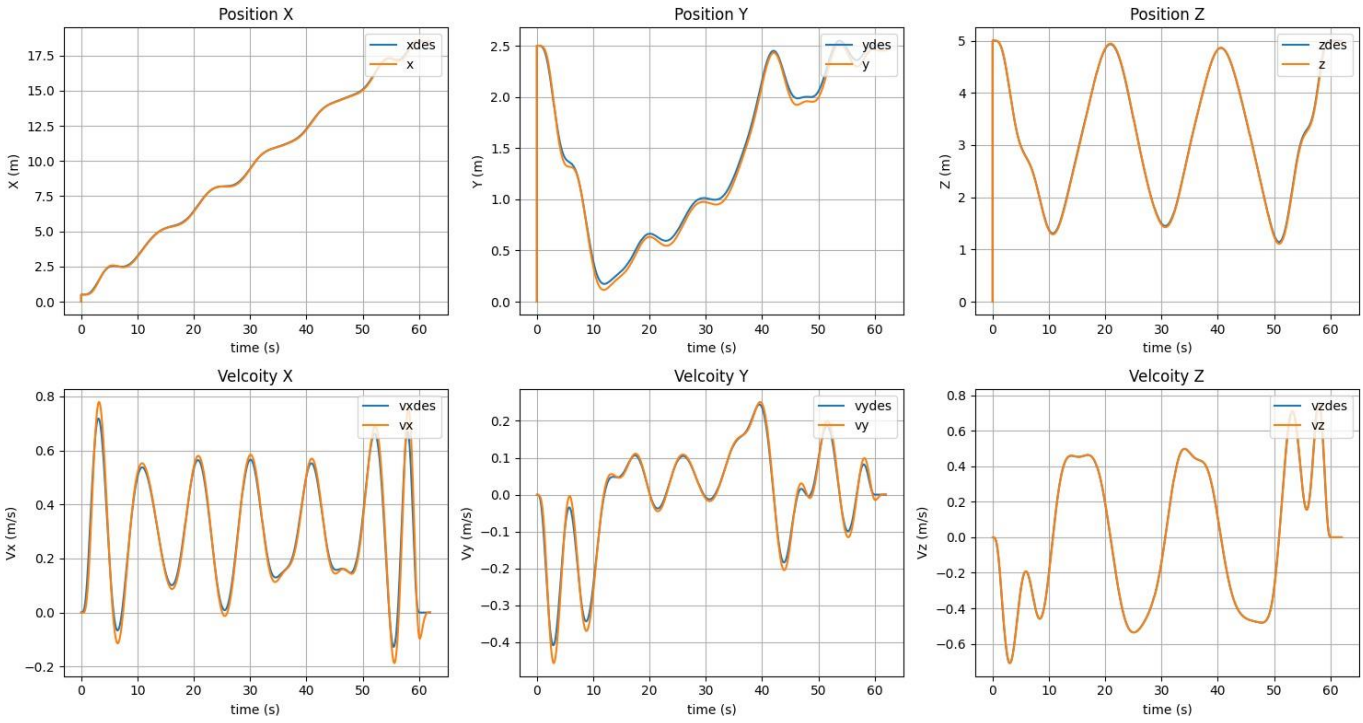

Fig. 12. Position and velocity control plots for sample map 3

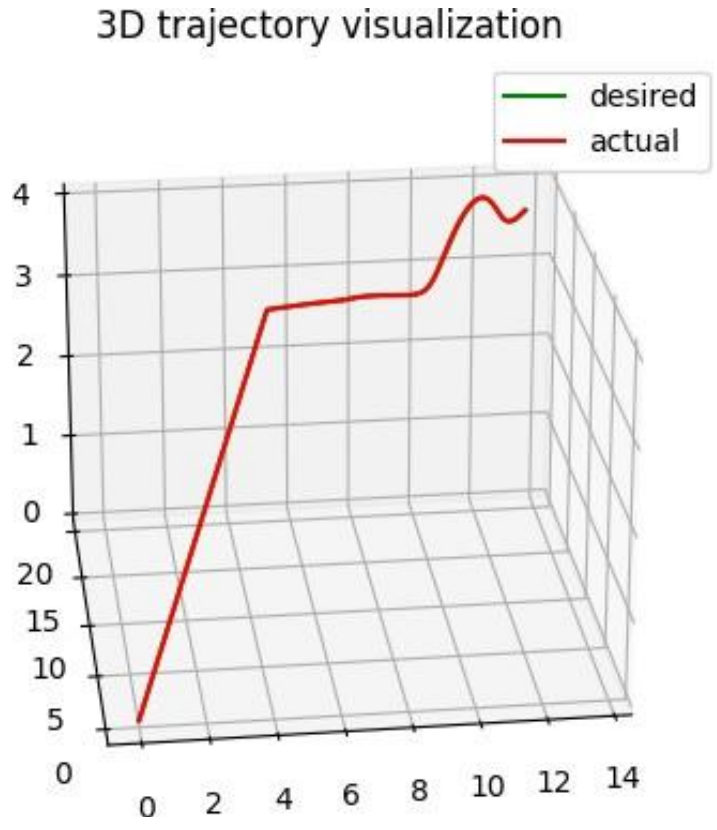

Fig. 13. Trajectory for sample map 4

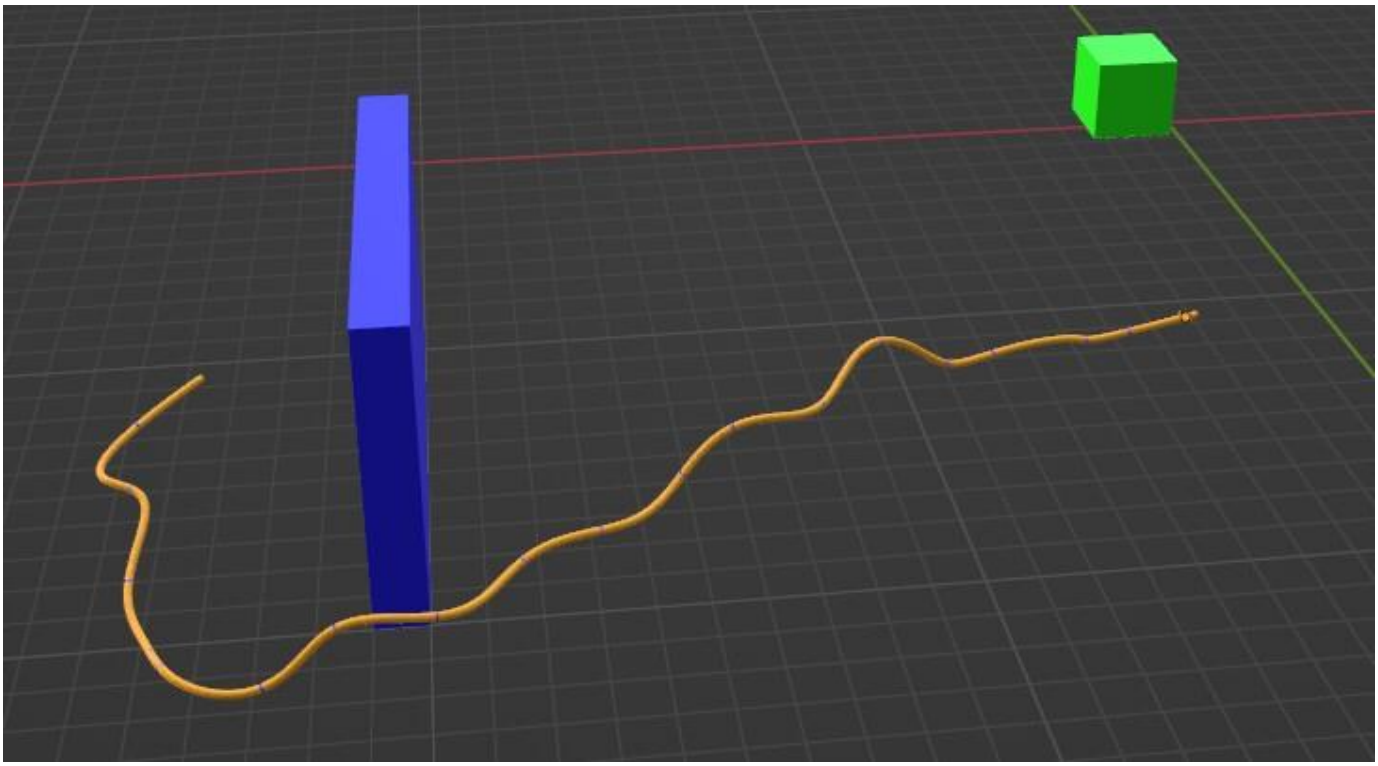

Fig. 14. Blender visualization of sample map 4

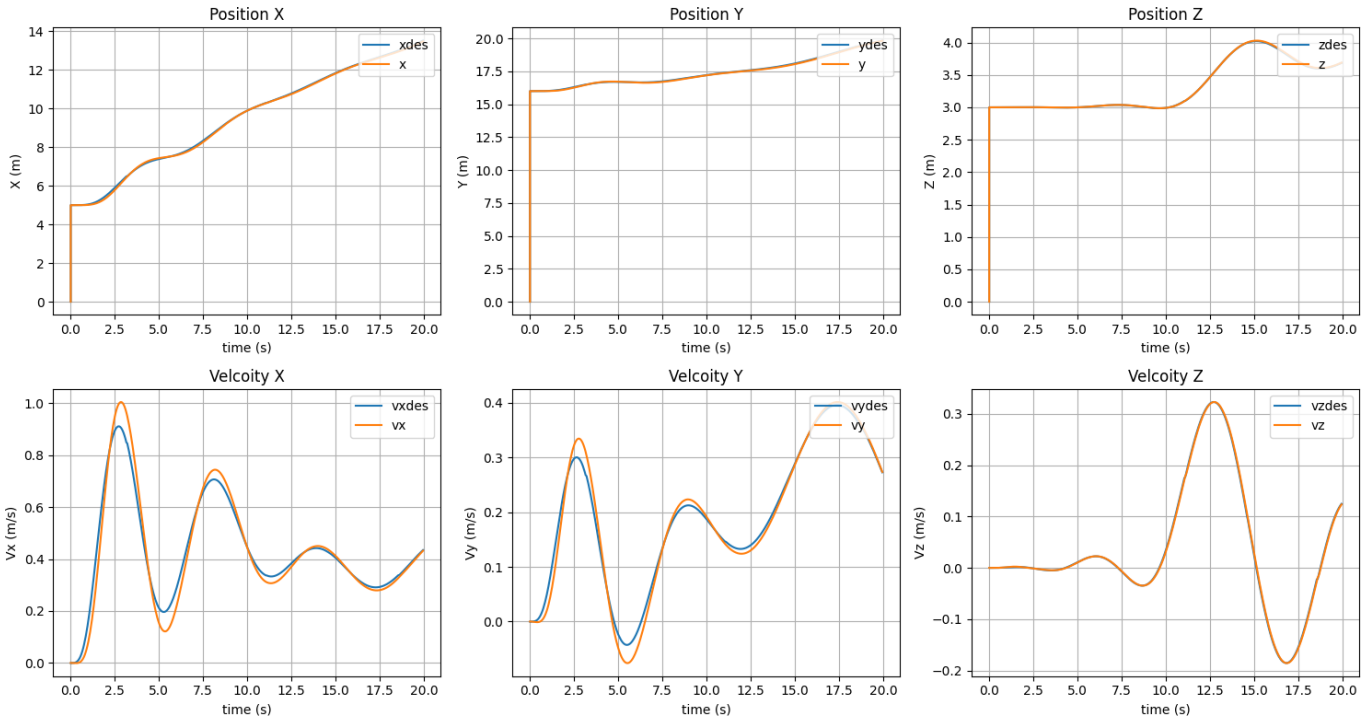

Fig. 15. Position and velocity control plots for sample map 4

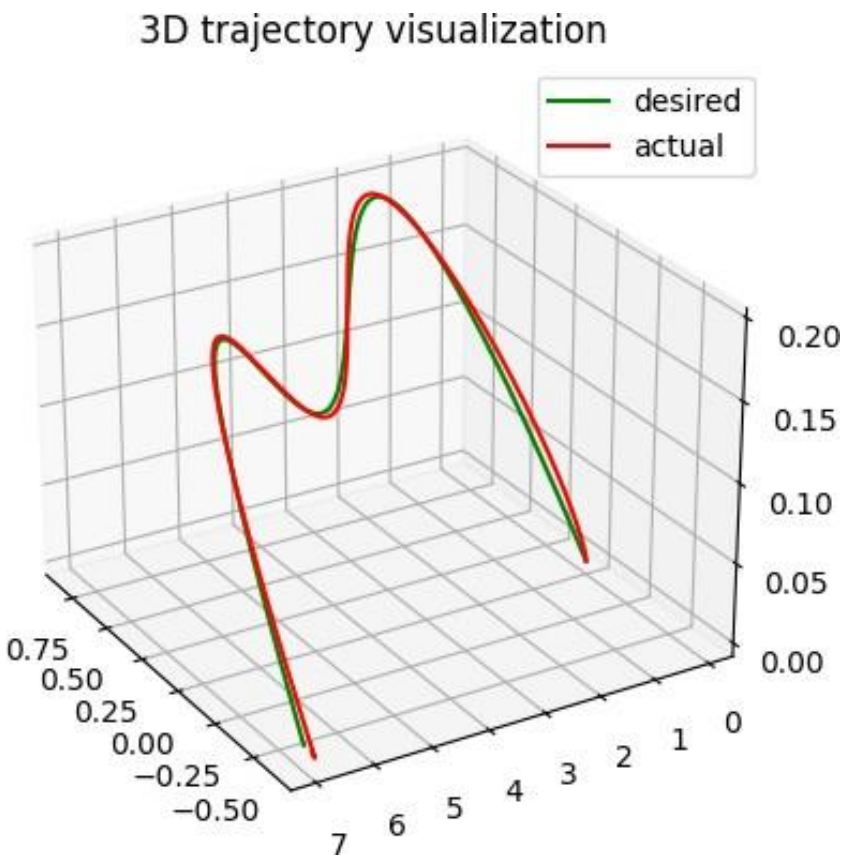

Fig. 16. Trajectory for real map 1

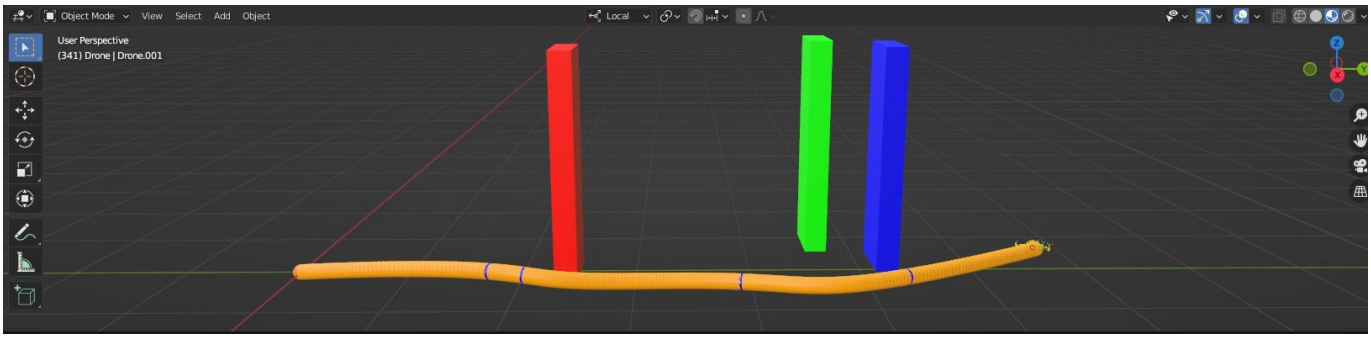

Fig. 17. Blender visualization of real map 1

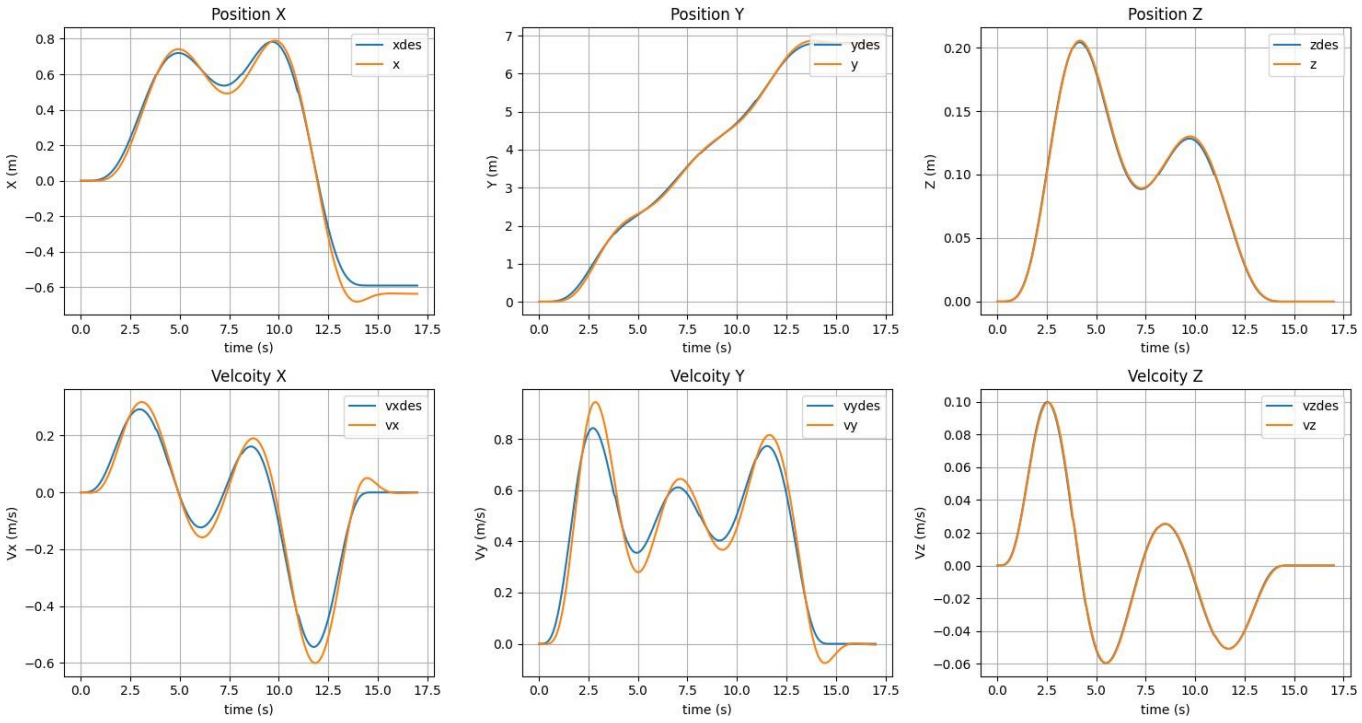

Fig. 18. Position and velocity control plots for real map 1