Autonomous navigation of drones in known environment

1st Venkateshkrishna
Masters in Robotics
Worcester Polytechnic Institute
Worcester, MA 01609
vparsuram@wpi.edu

2nd Athithya, Lalith
Masters in Robotics
Worcester Polytechnic Institute
Worcester, MA 01609
lnavaneethakrishnan@wpi.edu

3rd Gampa, Varun
Masters in Robotics
Worcester Polytechnic Institute
Worcester, MA 01609
vgampa@wpi.edu

using one late day

***Abstract*—This project focuses on creating an autonomous quadrotor navigation system within a known 3D environment through simulation-based experimentation. The objectives encompass developing robust path planning algorithms, including RRT-connect*, for collision-free trajectory generation, as well as fine-tuning the control stack to ensure stable and precise execution.To enable autonomous navigation in complex terrains, with RRT-connect* serving as a foundational step towards achieving this goal.**

I. Introduction

In this project, we present a comprehensive implementation approach comprising four fundamental components. Firstly, we introduce a Map/Environment reader and visualization method to interpret the pre-mapped 3D environment. Secondly, we detail the integration of an RRT* (Rapidly-exploring Random Tree Star) path planner, aimed at generating collision-free paths connecting the predefined start and goal positions. Next, we discuss the development of a trajectory planner designed to refine the path produced by the RRT-connect* algorithm, ensuring the creation of dynamically feasible trajectories. Lastly, we delve into the implementation of a PID (Proportional-Integral-Derivative) controller, responsible for guiding the quadrotor along the generated trajectory from start to goal while maintaining collision avoidance. This project's multidisciplinary approach seeks to enable autonomous navigation within a known 3D environment, encompassing path planning, trajectory optimization, and control mechanisms.

II. Generation of Map

The map provided for this project was defined by a collection of cuboids, each serving a specific purpose within the environment. The initial cuboid served as the boundary, outlining the limits of the drone's navigable area. Subsequent cuboids were utilized to represent obstacles within the environment. Each cuboid was characterized by two key points: the lower-left vertex and the upper-right vertex, defining its spatial dimensions. To facilitate visualization and mapping, Blender's primitive cube function was employed, generating a comprehensive representation of the environment. This map, consisting of boundary constraints and obstacle delineations, served as the foundation for subsequent path planning, trajectory optimization, and control system development for autonomous quadrotor navigation.

III. Sampling based planning using RRT*

RRT* is a sampling based algorithm in which the search tree rapidly expands from a start node. Subsequent points are randomly generated in the search space. Then the nearest node is found in the graph to the random point. A new node is generated at fixed step distance from the nearest point in the direction of the random point. If this node doesn't collide with any obstacle and the line joining this point and the nearest node is not passing through any obstacle, then this node is added as a vertex to the graph and the edge between the nearest node and new node is created and added to the graph. The method to check for collisions is explained in the subsequent section. In RRT* further this graph is optimized as per a heurestic cost so that an optimal path within the graph is selected at every iteration. While this slows down the path planning algorithm, the obtained path is generally much smoother. To speed up the path planning algorithm connect strategy was used. In RRT* generally a single node is added to the graph in every iteration. In our variation of RRT* multiple vertices are generated at fixed step size which are added based on the line connecting the nearest node to the x_{rand} until an obstacle is identified at which stage points are no longer added. This is called as the connect strategy which is explained in 2. The entire algorithm is explained in 1 .This connect algorithm replaces Steer in the standard RRT* algorithm. We used the step size as 0.6m. Also we generated more points even after the path is found to refine the same.

IV. Collision Checking

To check for collisions with an obstacle. First we inflated the obstacle by the largest dimension of the drone, which was 0.4m. Then we treated the drone as a point object. To check for collisions we simply checked if each dimension of the center of the drone would be between the extreme points provided for each block. Hence to check for collision in the path between two vertices, we just checked for collision on multiple evenly

Algorithm 1 RRT* Algorithm

$V \leftarrow \{x_{\text{init}}\}; E \leftarrow \emptyset;$
for $i = 1$ to n **do**
  $x_{\text{rand}} \leftarrow \text{SampleFree}_i;$
  $x_{\text{nearest}} \leftarrow \text{Nearest}(G = (V, E), x_{\text{rand}});$
  $X_{\text{new}} \leftarrow \text{Connect}(x_{\text{nearest}}, x_{\text{rand}});$
  for all $x_{\text{new}} \in X_{\text{new}}$ **do**
    if $\text{ObstacleFree}(x_{\text{nearest}}, x_{\text{new}})$ **then**
      $x_{\text{min}} \leftarrow x_{\text{nearest}}$
      $c_{\text{min}} \leftarrow \text{Cost}(x_{\text{nearest}}) + c(\text{Line}(x_{\text{nearest}}, x_{\text{new}}));$
      $V \leftarrow V \cup \{x_{\text{new}}\};$
      for all $x_{\text{near}} \in X_{\text{near}}$ **do**
        if $\text{CollisionFree}(x_{\text{near}}, x_{\text{new}}) \wedge$
        $\text{Cost}(x_{\text{near}}) + c(\text{Line}(x_{\text{near}}, x_{\text{new}})) < c_{\text{min}}$ **then**
          $x_{\text{min}} \leftarrow x_{\text{near}}$
          $c_{\text{min}} \leftarrow \text{Cost}(x_{\text{near}}) + c(\text{Line}(x_{\text{near}}, x_{\text{new}}))$
        end if
      end for
      $E \leftarrow E \cup \{(x_{\text{min}}, x_{\text{new}})\}$
      $c_{\text{near}} \leftarrow \text{Cost}(x_{\text{near}})$
      for all $x_{\text{near}} \in X_{\text{near}}$ **do**
        if $\text{CollisionFree}(x_{\text{new}}, x_{\text{near}}) \wedge$
        $\text{Cost}(x_{\text{new}}) + c(\text{Line}(x_{\text{new}}, x_{\text{near}})) < c_{\text{min}}$ **then**
          $x_{\text{parent}} \leftarrow \text{Parent}(x_{\text{near}});$
          $E \leftarrow (E \setminus \{(x_{\text{parent}}, x_{\text{near}})\}) \cup \{(x_{\text{new}}, x_{\text{near}})\};$
        end if
      end for
    end if
  end for
end for
return $G = (V, E);$

Algorithm 2 Connect algorithm

$X_{\text{new}} \leftarrow \emptyset;$
$x_{\text{rand}} \leftarrow \text{SampleFree}_i;$
$x_{\text{nearest}} \leftarrow \text{Nearest}(G = (V, E), x_{\text{rand}});$
$x_{\text{new}} \leftarrow \text{Steer}(x_{\text{nearest}}, x_{\text{rand}});$
while $\text{ObstacleFree}(x_{\text{new}}) \vee \text{Distance}(x_{\text{new}}, x_{\text{rand}})$ **do**
  $X_{\text{new}} \leftarrow X_{\text{new}} \cup x_{\text{new}}$
  $x_{\text{new}} \leftarrow \text{Steer}(x_{\text{new}}, x_{\text{rand}})$
end while
return X_{new}

spaced points on the line between two vertices. The points on the line were spaced by 0.2m.

V. Trajectory Generation

Using the waypoints generated by RRT*, we need to compute trajectory which is the desired position, velocity,acceleration and yaw. First the trajectory obtained from RRT* is refined. For this, we check, starting from the first point if the line joining this point and the next point in the waypoint list are is collision free, if yes then the next point is removed from the waypoints list. This is repeated until we encounter a case wherein we observe a collision in the line joining start point and the next point. When that occurs, the last removed point is re inserted and we repeat the same procedure as above, with the reintroduced point taking the place of the start point. Next to generated the trajectory between the refined way points we used a 7th degree polynomial fit between every two points. Given a time difference to reach from one point to the next and the boundary conditions we can obtain the coeffecients of all the 7th degree polynomial. To find the time instants for every two adjacent points we used an optimizer. The cost for the optimizer is a function of the time instants. More specifically based on the time instants, the polynomials are generated and from them, snap of the polynomial is calculated which is part of the cost, along with the time difference between two points. We used the COBYLA optimizer to optimize the trajectory. To add in the constraints of maximum velocity, we added the condition of minimum time required to traverse between two points such that it would result in feasible velocity.

VI. Controller Design

A. PID Controller Design

Thee controller present is a cascaded PID. It has three loops. The outermost loop is the position controller. It takes in the desired position and the current position and gives the velocity adjustment needed. The inner loop is the velocity controller. It's reference value is a combination of feedforward and feedback. That is, the reference velocity for the controller is the sum of desired velocty and the adjustment given by the position controller. This gives the desired acceleration and that is converted to the desired body velocity rates. The desired acceleration also gives the thrust required. These values are then sent as reference values for the innermost PID loop which is the attitude controller, which provides the torques in the body frame. Then using the thrust and the torques calculated, These values are fed into the dynamics for the drone to move.

B. Position Controller Gains

We assumed that the drone is symmetric about the z-axis and hence the gains for x and y axis would be the same. To tune these gains, first we set the desired trajectory as 0,0,0 wherein the drone would hover at the origin. We even had to tune the velocity controller for the same.

C. Velocity Controller Gains

We further fine tuned the velocity controller by using the sample trajectory which was given to us, which was a helical path.

VII. Results

We have tested this algorithm for autonomous navigation in a known environment using blender simulation. The map and inflated obstacles are seen in 1. Next, the RRT tree can be seen in 2. The optimal path from RRT along with the trajectory generated by the optimizing algorithm is seen in 3. To evaluate our controller's performance, we also plotted the actual path

taken by the drone as can be seen in 4. You can see that the RRT tree very quickly expands into the entire navigatable area. And the trajectory generated is very smooth and passes through the points given by RRT* which are highlighted in green. It can also be noticed how the trajectory demands the drone to speed up and slow down, by looking at the spline in the figure and videos attached. Finally we can also see that the controller is able to track the trajectory pretty well.

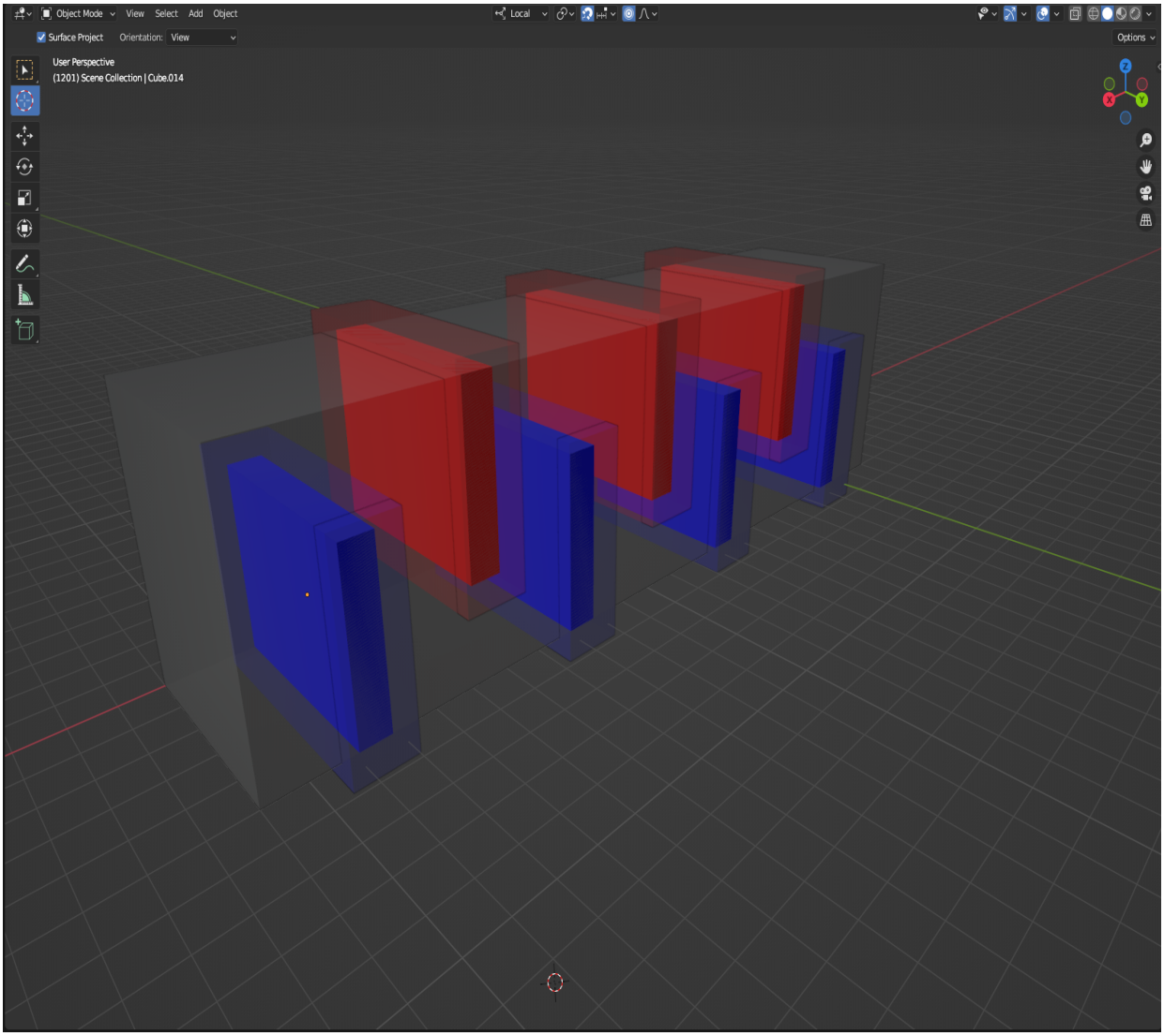

Fig. 1. Inflated environment

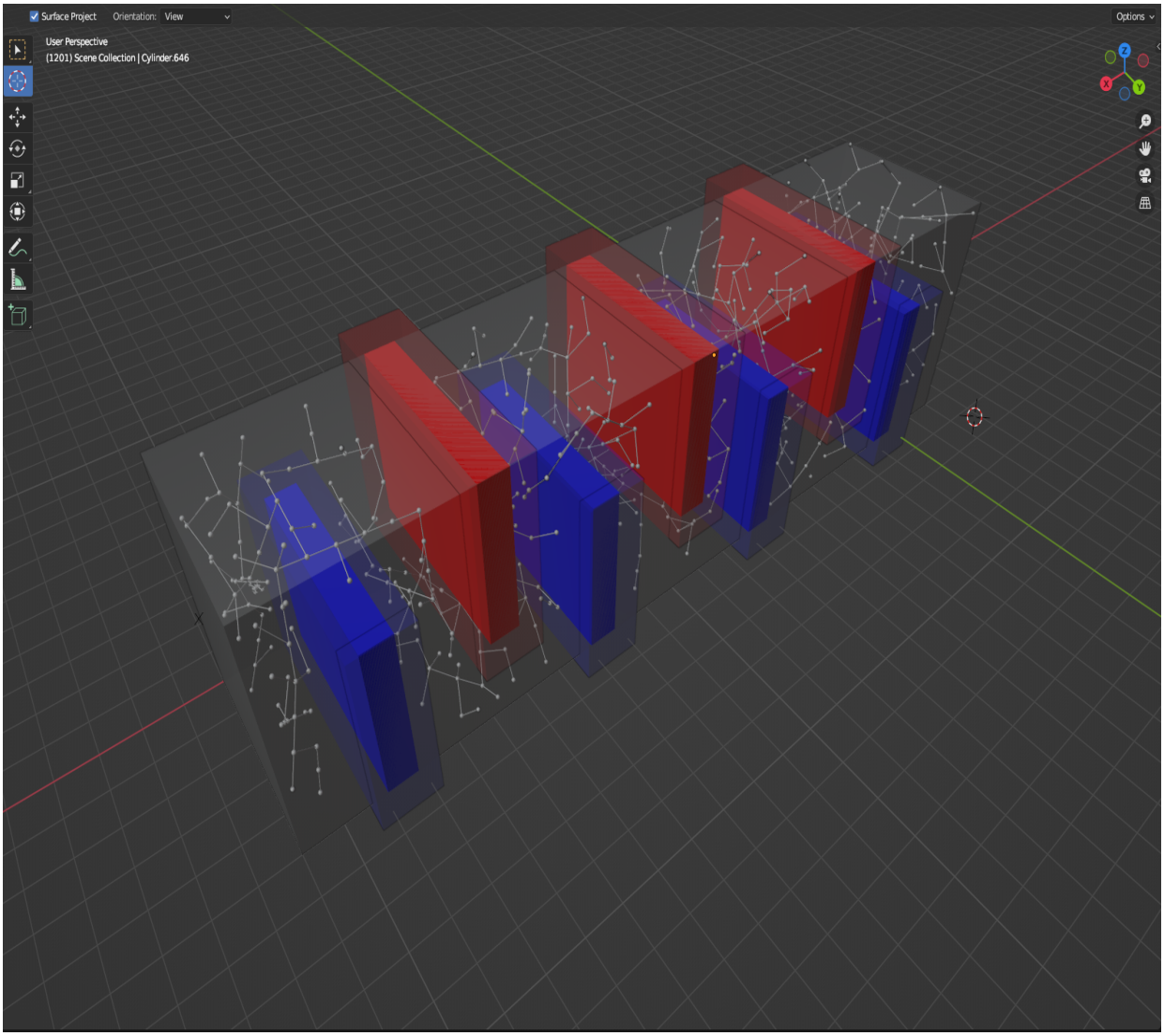

Fig. 2. RRT tree

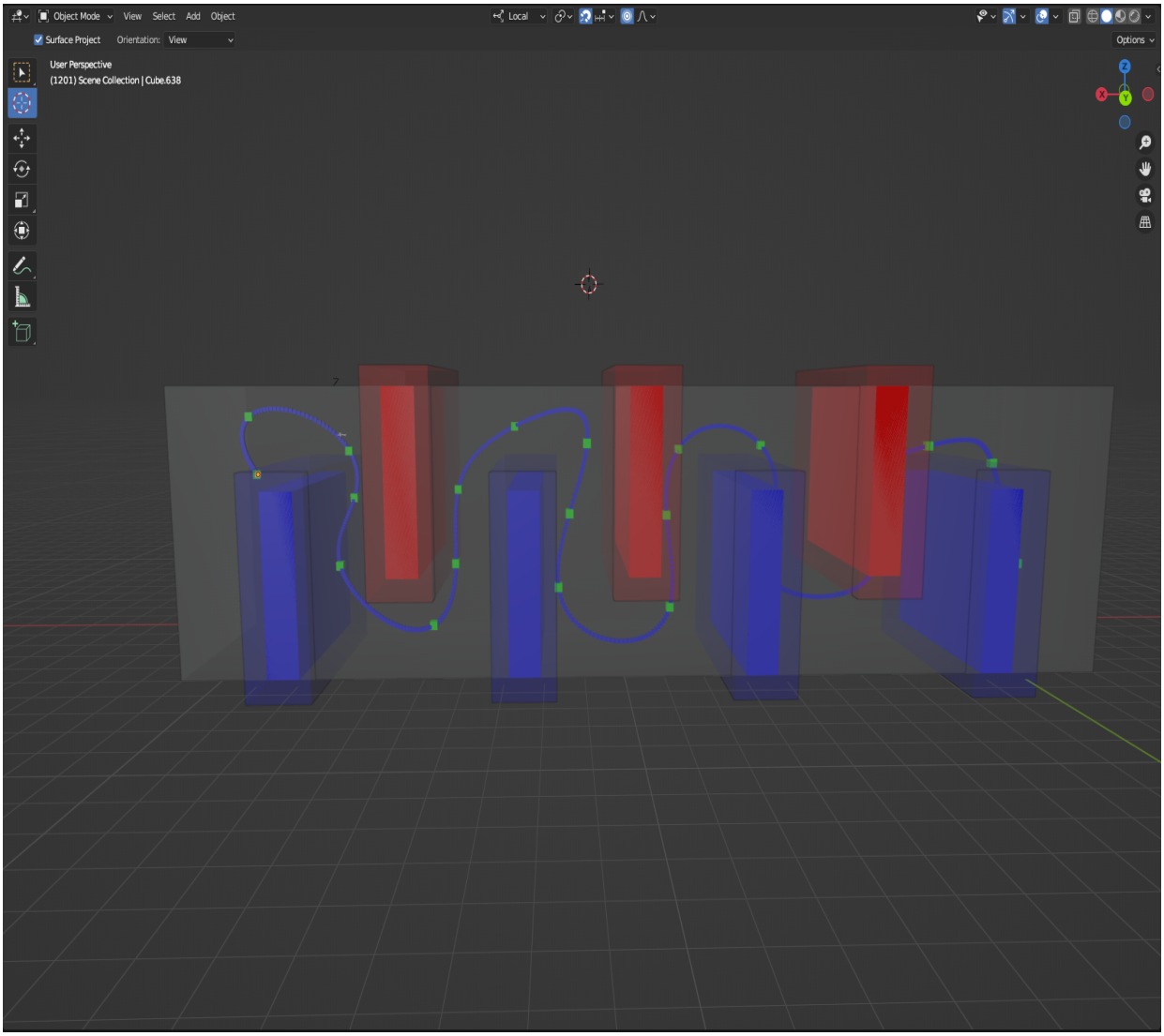

Fig. 3. Optimal path to goal (green) and trajectory(blue)

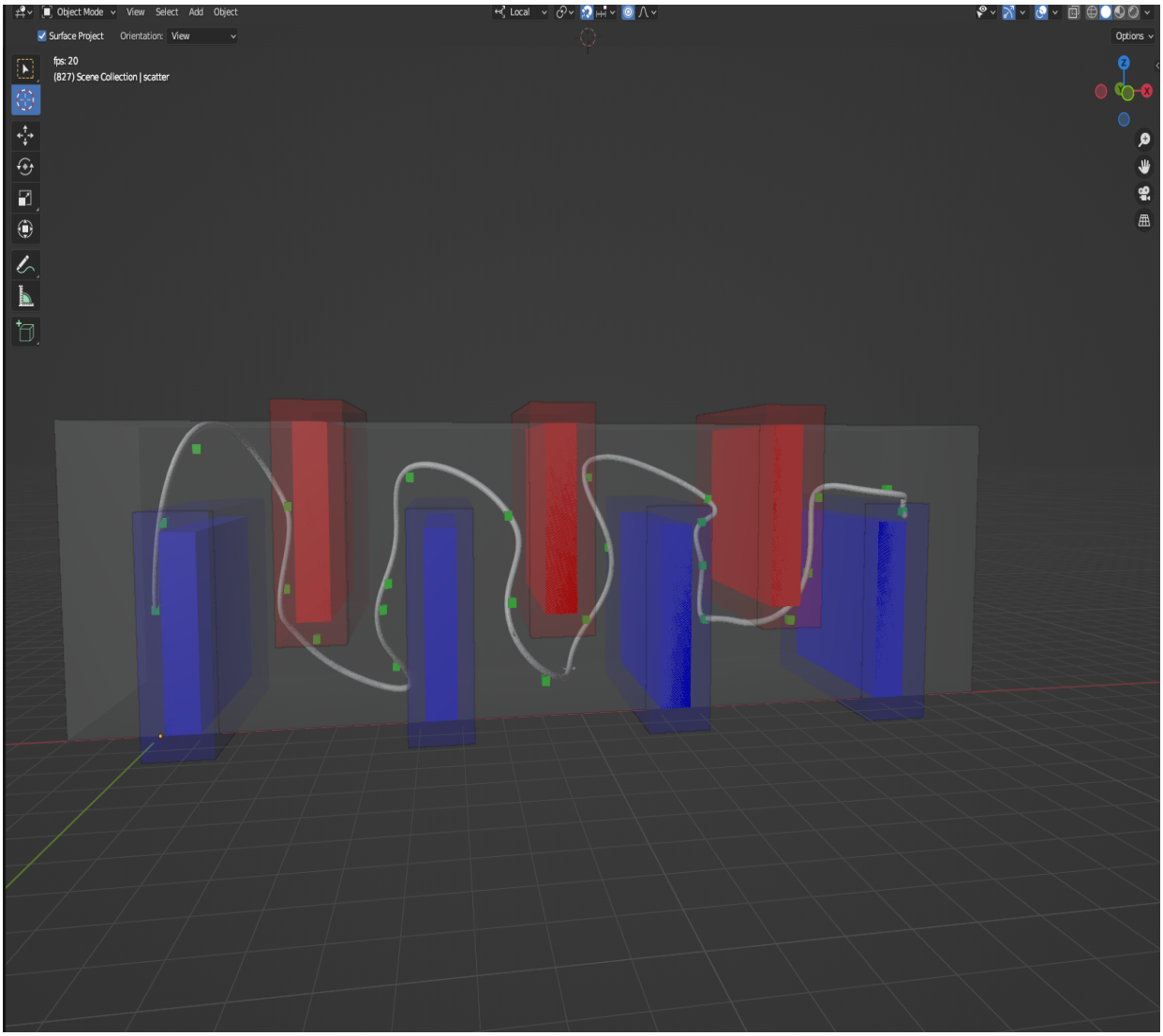

Fig. 4. Path followed by the drone

VIII. Videos

The videos of autonomous navigation in different environments can be found at videos

IX. Conclusion

In this project we immplemented an autonomous stack to control a drone in a known environment. The stack was implemented on python and tested in blender simulation setup. Our optiimzed stack is able to generate the path and trajectories quickly, as well as making sure that the drone is safe and able to get to the goal quickly. Finally the gains of the controller are fine tuned to be able to track the desired trajectory.

References

[1] RRT Star: link
[2] Trajectory Optimization: link