RBE595 - Project 2a - Path Planning and Trajectory Generation

Pranay Katyal
Worcester Polytechnic Institute
Worcester, MA, USA
pkatyal@wpi.edu

Anirudh Ramanathan
Worcester Polytechnic Institute
Worcester, MA, USA
aramanathan@wpi.edu

Hudson Kortus
Worcester Polytechnic Institute
Worcester, MA, USA
hkortus@wpi.edu

***Abstract*—Path planning is a vital step in robot autonomy. In this project, we implement an RRT* path planner to find an efficient path in a large 3D environment. Next, we develop a minimum snap trajectory planner to convert the path into a smooth trajectory dynamically feasible by our aerial robot. We run this trajectory through a cascading controller to get the robot control input and tune the entire system to achieve fast, reliable trajectory generation and execution.**

I. Problem Statement

Given a map of the world, a robot must be able to navigate through goal points while also avoiding obstacles. Although this is a trivial task for humans and animals, it requires complex solutions for robotics. One can discretize the world into finite units and use grid-based planners like A*, but in a reasonably sized 3D environment, storing and traversing all these blocks becomes computationally infeasable.

Furthermore, once a viable path is found, it must be converted into a path that the robot can actually follow, along with a list of desired positions, velocities, and accelerations at each point, along that path.

Finally, once a suitable trajectory is created, it must be fed into a controller that generates the desired torques of each propeller that are required to execute the given trajectory.

II. Methodology

This assignment was built within matplotlib graphs 1. Maps provided by the assignment contained the boundary of the map and the location of all obstacle, which were assumed to be axis-aligned rectangular prisms.

The assignment simulated the motion of an aeiral robot with the dynamics of a DJI Tello drone 2 [1]. The drone dimensions were found from [2] and state the the drones's length, width, and height are 98, 95, and 41mm respectively. For the ease of simulation we simplified to drone to be a 100x100x50mm cuboid. We chose the slightly inflated x,y dimensions primary for convenance and extra safety margin. We chose to significantly increase the z height of the quadcopter because if an object came this close to the top or bottom of it in real life it would alter the propeller aerodynamics enough to cause a crash.

A. Planning

Rapidly Exploring Random Tree (RRT) is a sampling based planner that can generate a feasible path from a defined start

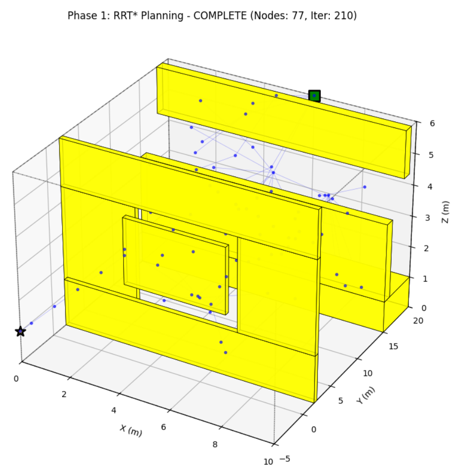

Fig. 1. Simulated Environment with Obstacle Read from Map1

Fig. 2. Enter Caption

to goal. RRT works with the knowledge of the start and goal configurations, iteratively stepping toward randomly sampled points from the closest node on the tree. RRT* is an extension of RRT that is more optimal because it utilizes cost heuristics to improve the path as the planner progresses. In each iteration, after the step distance is calculated, RRT* rewires the step node to a different parent node that reduces the overall cost-to-come of the path.

Algorithm 1 RRT* Algorithm

```
Require: Initial state $x_{init}$, goal region $X_{goal}$, number of iterations $N$, neighborhood radius $\gamma$
Ensure: Tree $T = (V, E)$ with near-optimal path from $x_{init}$ to $X_{goal}$
1: $V \leftarrow \{x_{init}\}$
2: $E \leftarrow \emptyset$
3: cost($x_{init}$) $\leftarrow 0$
4: parent($x_{init}$) $\leftarrow$ NULL
5: for $i = 1$ to $N$ do
6:     $x_{rand} \leftarrow$ SAMPLE()    ▷ Sample random state
7:     $x_{nearest} \leftarrow$ NEAREST($T, x_{rand}$)    ▷ Find nearest node
8:     $x_{new} \leftarrow$ STEER($x_{nearest}, x_{rand}$)    ▷ Extend toward sample
9:     if COLLISIONFREE($x_{nearest}, x_{new}$) then
10:        $X_{near} \leftarrow$ NEAR($T, x_{new}, \gamma$)    ▷ Find nearby nodes
11:        $V \leftarrow V \cup \{x_{new}\}$
12:                                        ▷ Choose best parent
13:        $x_{min} \leftarrow x_{nearest}$
14:        $c_{min} \leftarrow$ cost($x_{nearest}$) + COST($x_{nearest}, x_{new}$)
15:        for $x_{near} \in X_{near}$ do
16:            $c_{new} \leftarrow$ cost($x_{near}$) + COST($x_{near}, x_{new}$)
17:            if COLLISIONFREE($x_{near}, x_{new}$) and $c_{new} < c_{min}$ then
18:                $x_{min} \leftarrow x_{near}$
19:                $c_{min} \leftarrow c_{new}$
20:            end if
21:        end for
22:        $E \leftarrow E \cup \{(x_{min}, x_{new})\}$    ▷ Add edge to tree
23:        parent($x_{new}$) $\leftarrow x_{min}$
24:        cost($x_{new}$) $\leftarrow c_{min}$
25:                                        ▷ Rewire tree
26:        for $x_{near} \in X_{near} \setminus \{x_{min}\}$ do
27:            $c_{new} \leftarrow$ cost($x_{new}$) + COST($x_{new}, x_{near}$)
28:            if COLLISIONFREE($x_{new}, x_{near}$) and $c_{new} <$ cost($x_{near}$) then
29:                $x_{parent} \leftarrow$ parent($x_{near}$)
30:                $E \leftarrow E \setminus \{(x_{parent}, x_{near})\}$
31:                $E \leftarrow E \cup \{(x_{new}, x_{near})\}$
32:                parent($x_{near}$) $\leftarrow x_{new}$
33:                cost($x_{near}$) $\leftarrow c_{new}$
34:            end if
35:        end for
36:    end if
37: end for
38: return $T = (V, E)$
```

B. Trajectory

While RRT* effectively finds a valid path, the planner does not take into account the complete dynamics of the robot since it is sampling in a discrete space. For this reason, the generated paths would need to be converted into continuous space and set to follow real-world constraints such as position, velocity, and acceleration at each trajectory point.

Quintic (5th-order) polynomial splines solve each trajectory segment independently between consecutive waypoints. For each segment, the polynomial is:

$$p(t) = c_0 + c_1 t + c_2 t^2 + c_3 t^3 + c_4 t^4 + c_5 t^5$$

Six boundary conditions are enforced at the start ($t = 0$) and end ($t = T$) of each segment:

$$p(0) = q_0, \quad p(T) = q_f \quad \text{(position)}$$
$$\dot{p}(0) = v_0, \quad \dot{p}(T) = v_f \quad \text{(velocity)}$$
$$\ddot{p}(0) = a_0, \quad \ddot{p}(T) = a_f \quad \text{(acceleration)}$$

This creates a linear system $\mathbf{Ac} = \mathbf{b}$ that is solved independently for each segment:

$$\begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 2 & 0 & 0 & 0 \\ 1 & T & T^2 & T^3 & T^4 & T^5 \\ 0 & 1 & 2T & 3T^2 & 4T^3 & 5T^4 \\ 0 & 0 & 2 & 6T & 12T^2 & 20T^3 \end{bmatrix} \begin{bmatrix} c_0 \\ c_1 \\ c_2 \\ c_3 \\ c_4 \\ c_5 \end{bmatrix} = \begin{bmatrix} q_0 \\ v_0 \\ a_0 \\ q_f \\ v_f \\ a_f \end{bmatrix}$$

Critical limitations:

- Segments are optimized in isolation without considering global path quality, leading to suboptimal trajectories
- Only guarantees continuity up to acceleration; **jerk is discontinuous** at waypoints, causing abrupt control changes
- No optimization criterion—finds *any* solution, not the *best* one
- Cannot incorporate inequality constraints (boundaries, obstacles) during solving

Septic (7th-order) minimum snap trajectories solve the trajectory optimization problem by formulating it as a global quadratic programming (QP) optimization. For a trajectory with N segments, each using a 7th-order polynomial:

$$p_i(t) = c_{i,0} + c_{i,1}t + c_{i,2}t^2 + c_{i,3}t^3 + c_{i,4}t^4 + c_{i,5}t^5 + c_{i,6}t^6 + c_{i,7}t^7$$

where $i \in \{1, 2, \ldots, N\}$ indexes the segments. The optimization finds all $N \times 8$ polynomial coefficients simultaneously.

The **cost function** minimizes the integral of squared snap across all segments:

$$J = \sum_{i=1}^{N} \int_0^{T_i} \left(\frac{d^4 p_i}{dt^4} \right)^2 dt$$

For a 7th-order polynomial, the snap (4th derivative) is:

$$\frac{d^4p_i}{dt^4} = 24c_{i,4} + 120c_{i,5}t + 360c_{i,6}t^2 + 840c_{i,7}t^3$$

The integral becomes a quadratic form:

$$J = \frac{1}{2}\mathbf{x}^T\mathbf{Q}\mathbf{x}$$

where $\mathbf{x} = [c_{1,0}, c_{1,1}, \ldots, c_{1,7}, c_{2,0}, \ldots, c_{N,7}]^T$ is the vector of all coefficients and $\mathbf{Q}$ is the global cost matrix.

Each block $Q_{i,j}$ in $\mathbf{Q}$ contains terms of the form:

$$Q_{i,j} = \frac{i!}{(i-4)!} \cdot \frac{j!}{(j-4)!} \cdot \frac{T^{i+j-7}}{i+j-7}$$

for coefficients $i, j \geq 4$, derived from integrating products of polynomial derivatives.

The optimization is subject to **equality constraints** $\mathbf{A}_{eq}\mathbf{x} = \mathbf{b}_{eq}$ that enforce:

1) **Position matching at waypoints:**

$$p_i(0) = w_i, \quad p_i(T_i) = w_{i+1}$$

2) **Derivative continuity at intermediate waypoints:**

$$\left.\frac{dp_i}{dt}\right|_{t=T_i} = \left.\frac{dp_{i+1}}{dt}\right|_{t=0} \quad \text{(velocity)}$$

$$\left.\frac{d^2p_i}{dt^2}\right|_{t=T_i} = \left.\frac{d^2p_{i+1}}{dt^2}\right|_{t=0} \quad \text{(acceleration)}$$

$$\left.\frac{d^3p_i}{dt^3}\right|_{t=T_i} = \left.\frac{d^3p_{i+1}}{dt^3}\right|_{t=0} \quad \text{(jerk)}$$

3) **Boundary conditions at endpoints:**

$$\left.\frac{dp_1}{dt}\right|_{t=0} = 0, \quad \left.\frac{dp_N}{dt}\right|_{t=T_N} = 0 \quad \text{(velocity)}$$

$$\left.\frac{d^2p_1}{dt^2}\right|_{t=0} = 0, \quad \left.\frac{d^2p_N}{dt^2}\right|_{t=T_N} = 0 \quad \text{(acceleration)}$$

Each constraint becomes a row in $\mathbf{A}_{eq}$ by evaluating polynomial derivatives at specific times. For example, velocity continuity at waypoint k requires:

$$c_{i,1} + 2c_{i,2}T_i + 3c_{i,3}T_i^2 + \cdots + 7c_{i,7}T_i^6 = c_{i+1,1}$$

Inequality constraints $\mathbf{A}_{ineq}\mathbf{x} \leq \mathbf{b}_{ineq}$ enforce boundary limits by sampling points along each segment (typically 25 per segment) and requiring:

$$x_{min} \leq p_i(t_k) \leq x_{max} \quad \forall k \in \text{samples}, \forall i \in \{1, \ldots, N\}$$

for each spatial dimension (x, y, z).

The complete QP problem is:

$$\begin{aligned} \text{minimize} \quad & \frac{1}{2}\mathbf{x}^T\mathbf{Q}\mathbf{x} \\ \text{subject to} \quad & \mathbf{A}_{eq}\mathbf{x} = \mathbf{b}_{eq} \\ & \mathbf{A}_{ineq}\mathbf{x} \leq \mathbf{b}_{ineq} \end{aligned}$$

This is solved using **Sequential Quadratic Programming (SLSQP)**. The solver iteratively refines the coefficients by:

1) Computing gradients: $\nabla J = \mathbf{Q}\mathbf{x}$
2) Satisfying constraints through Lagrange multipliers
3) Updating the solution until convergence

An initial guess is generated by solving the unconstrained problem (equality constraints only) using KKT conditions:

$$\begin{bmatrix} \mathbf{Q} & \mathbf{A}_{eq}^T \\ \mathbf{A}_{eq} & \mathbf{0} \end{bmatrix} \begin{bmatrix} \mathbf{x} \\ \boldsymbol{\lambda} \end{bmatrix} = \begin{bmatrix} \mathbf{0} \\ \mathbf{b}_{eq} \end{bmatrix}$$

This provides a good starting point for SLSQP, which then enforces the inequality constraints while maintaining optimality. The result is a globally optimal trajectory with continuous jerk, minimal control effort, and guaranteed constraint satisfaction. Each spatial dimension (x, y, z) is solved independently to produce the final 3D path.

C. Control

To enable the aerial robot to follow the computed trajectory, the team tuned a cascaded PID controller based on the PX4 control stack [3]. The controller receives the desired position, velocity, and acceleration setpoints and ultimately outputs the motor torques.

First, the position error is processed by the position controller to generate a feedback velocity. This velocity is added to the trajectory's commanded velocity, and the sum is passed to the velocity controller, which produces a feedback acceleration. That acceleration is combined with the trajectory's desired acceleration to yield the total control acceleration.

Next, the control acceleration is mapped to an angular acceleration in the robot's body frame.This is necessary because while we are commanding linear position, velocity, and accelerations, a quadcopter can only move thorugh angular accelerations. Therefore all control inputs must be converted to angluar accelerations before we can send commands to the motors. The angular accelerations are fed into the final PID controller to determine the control angular rates. Finally, the overall thrust in the drone's body Z-axis is computed from the control accelerations, allowing the motor velocities to be calculated.

While this cascading architecture is beautifully elegant, tuning was difficult because the position and velocity controllers were tightly coupled. To tune this controller, we first set the desired position to be constant, and desired velocity and acceleration to be 0. We set all values in the position controller to be 1 and tuned the velocity controller. Once the robot could reliably maintain its position, we continued to continued to adjust values, watching the τ_x, τ_y, τ_x making sure they changed smoothly and were not getting cliped at their max values. Once we were satisfied with the velocity controller, we made the robot follow simple trajectories with low accelerations and velocities. Critically, we reduced the velocity gains as we added the position controller to prevent our control inputs from being clipped. Additionally we kept the position controller an order of magnitude lower than the velocity controller to make sure the velocities were prioritized over the position. After we had a tune we were satisfied with we could being to increase the allowable velocity and accelerations, continuing to adjust the gains.

After tuneing our controller in parralel with the trejectory generator, we chose a set of values that worked well for us.

TABLE I
PID Gains for Position and Velocity Controllers

Controller	Axis	K_p	K_i	K_d	Min	Max
Position	x	0.8	0.0	0.2	-10	10
	y	0.8	0.0	0.2	-10	10
	z	1.0	0.0	0.25	-10	10
Velocity	v_x	1.5	0.1	0.3	-5	5
	v_y	1.5	0.1	0.3	-5	5
	v_z	2.0	0.1	0.4	-5	5

Technically, the integral gains of the velocity and derivative gains of the position controller should be 0 because that gain was already reflected in the other controller. The integral gain of the velocity controller is mathematically equivalent to the proportional gain of the position controller, and the derivative gain of the position controller is equivalent to the proportional velocity gain. While we tried this, we found that the attached gains were more stable and gave us excellent responses.

We also found it helpful to increase the max actuation (maxAxt) value of the controller because we found that our controller was over saturating and our control imputs were being clipped. The final value we chose was .5; in the real world this number could cause problems such as delay between motor commands and actuation and nonlinear turbulent propeller dynamics might make our drone unstable in the higher actuation commands.

III. Results

We were able to evaluate the performance of our planner, trajectory generator, and controller using the generated Matplotlib outputs.

The following is the output for the planner:

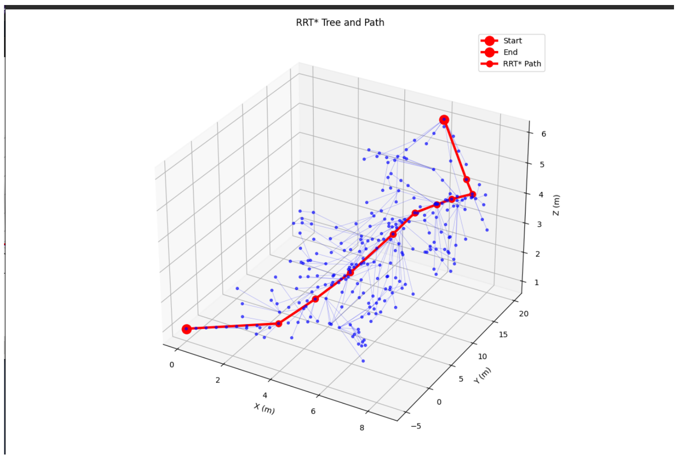

Fig. 3. RRT* on Map 1

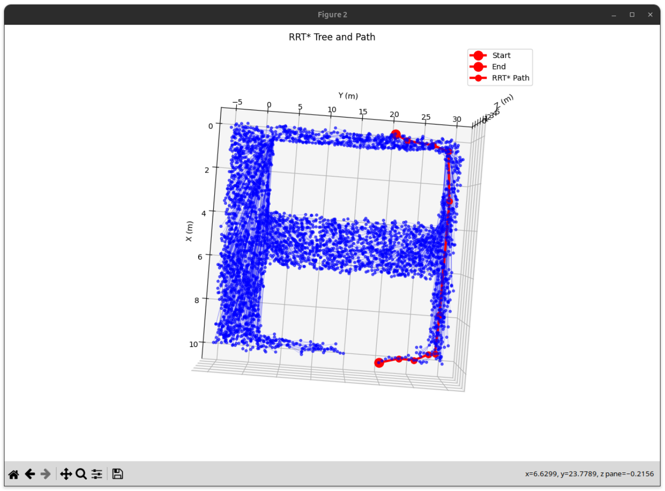

Fig. 4. RRT* on Map 2

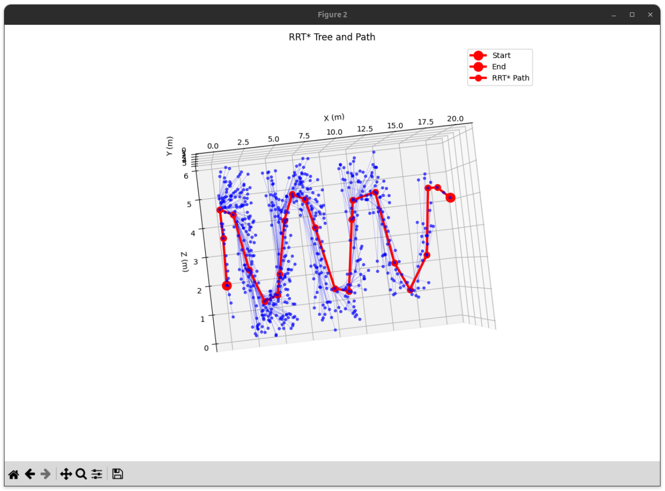

Fig. 5. RRT* on Map 3

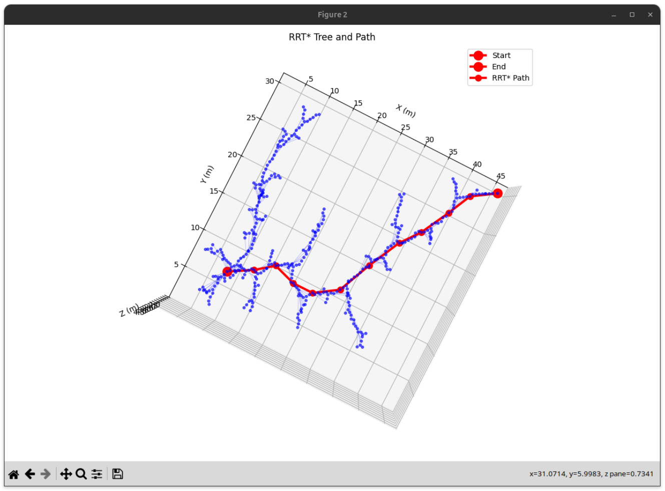

Fig. 6. RRT*on Map 4

The following is the output for the trajectory generation:

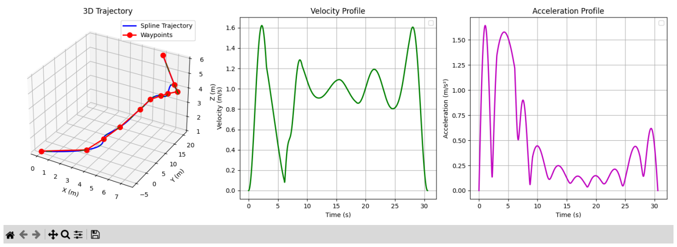

Fig. 7. Trajectory Generation on Map 1

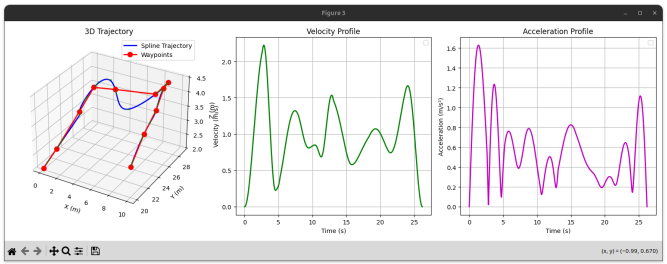

Fig. 8. Trajectory Generation on Map 2

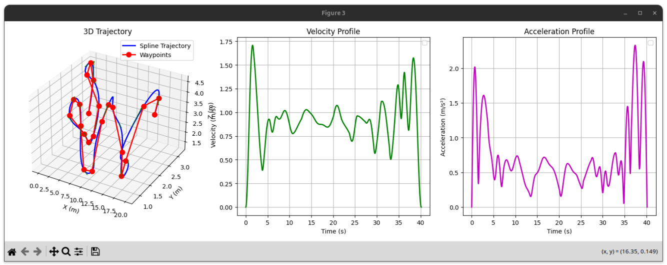

Fig. 9. Trajectory Generation on Map 3

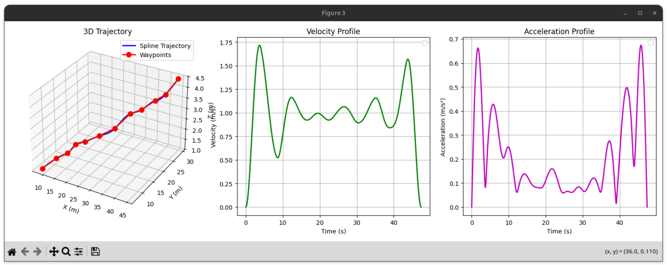

Fig. 10. Trajectory Generation on Map 4

The following is the output for the controller:

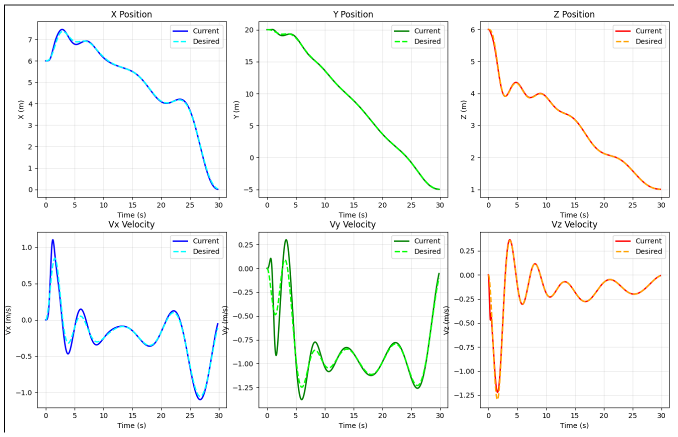

Fig. 11. Position and Velocity Control on Map 1

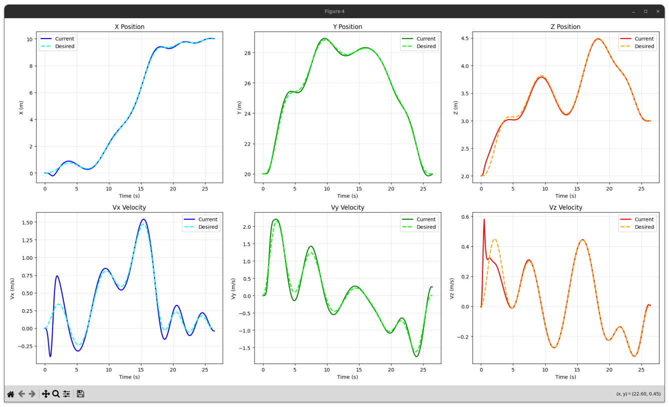

Fig. 12. Position and Velocity Control on Map 2

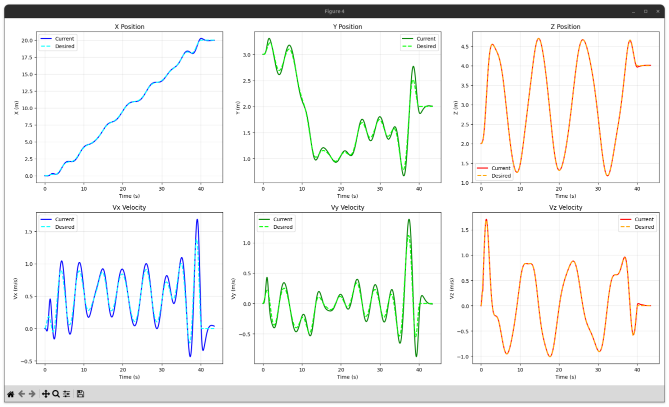

Fig. 13. Position and Velocity Control on Map 3

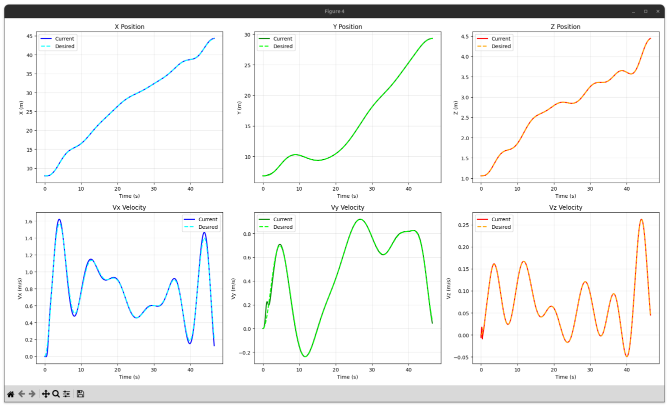

Fig. 14. Position and Velocity Control on Map 4

For all 4 test cases, RRT* was able to find a feasible path for our drone. While the trajectory generator was able to follow the constraints and formulate a smooth trajectory, there were some specific cases where the magnitude of these constraints caused instability and made the path appear as though it was moving through the defined obstacles or boundaries. In order to complete Map 1, we had to reduce the size of the first obstacle so that RRT* could find a path with a safety margin we were comfortable with. For each map, the controller initially experienced divergence and a large overshoot; however, over a short time, it was able to converge to the desired position and velocity values.

The video shows[4] the aerial robot successfully navigating all 4 maps. In the video we speed up the simulation by 10 times to make it easier to watch. While this simulation took a very long time, the simulated time the aerial robot took was very good. We traced this long lag to the matplotlib re-plotting method.

IV. Conclusion

In all the maps where the start and end positions were coincident with the map boundary, the aerial robot would enter the boundary's safety margin as it approached the goal. We assumed this behavior was acceptable, since the aerial robot was explicitly commanded to reach the boundary.

We are also using 2 of our Late Days.

References

[1] "Tello," https://store.dji.com/product/tello, DJI (Ryze Tech), 2025, accessed: 2025-09-30.

[2] "Tello specs," https://www.ryzerobotics.com/tello/specs, Ryze Tech, 2025, accessed: 2025-09-30.

[3] "Controller diagrams — px4 guide," https://docs.px4.io/main/en/flight_stack/controller_diagrams.html, PX4 Development Team, 2025, accessed: 2025-09-30.

[4] P. Katyal, A. Ramanathan, and H. Kortus, "Project files for rbe595 - project 2a - video," Google Drive, 2025, accessed: 2025-10-01. [Online]. Available: https://drive.google.com/drive/folders/14fKImx4btvPeq52bd4vC4GYgGMfyh-vo?usp=sharing