RBE595 - Project 2b - 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*—The goal of this project is to implement the navigation stack developed in the last assignment inside a quadrotor simulator. The quadrotor simulator operates the navigation stack within a digital twin environment made from images of a real-world environment that have been stitched together to form a photorealistic scene. Navigational constraints and collision requirements were given.**

I. Problem Statement

In the previous assignment, we developed a complete navigation stack for a robot capable of finding and following feasible paths through goal points while avoiding obstacles. This involved discretizing the environment, generating paths using a planner, converting those paths into smooth and dynamically feasible trajectories, and finally controlling the robot along those trajectories through appropriate actuator commands.

In this project, we extend that navigation stack by integrating it into a photorealistic quadrotor simulation environment known as VizFlyt. VizFlyt takes the outputs of the navigation stack along with stitched 2D imagery and a mapped model of the environment to simulate the vehicle's dynamics within a digital twin that closely replicates realistic perception and motion behavior. Unlike typical visual simulators, VizFlyt processes non-visual, data-driven inputs and generates a photorealistic first-person view of the vehicle navigating through the environment in accordance with computed dynamics [1].

Due to the intensive rendering and computation requirements, the entire VizFlyt simulation stack must be executed within a custom-defined turing cluster, a temporary, and sufficiently large allocation of remote computer resources, which is accessed through an SSH terminal on our local machines.

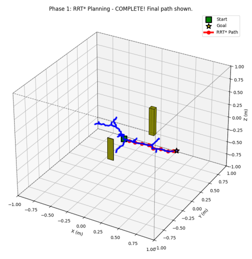

Fig. 1. Simulated Environment with Obstacle Read from Map1

Fig. 2. Tello Drone

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 aerial robot with the dynamics of a DJI Tello drone [2]. The drone dimensions were found from [3] 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 convinience 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 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 [4], [5].

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, it samples discretely and does not incorporate the full dynamics of the robot. Therefore, the resulting paths require conversion into smooth, continuous trajectories that satisfy real-world constraints on position, velocity, and acceleration at every point.

In our implementation, we employ a spline-based approach to generate smooth trajectories from waypoints using 7th-order polynomials to ensure continuity of position, velocity, acceleration, and jerk. The trajectory is generated as follows:

- Waypoints are first clamped within environment boundaries to maintain feasibility.
- Segment times are adaptively allocated based on inter-waypoint distances, curvature, and proximity to boundaries, ensuring safety and smooth speed profiles.
- For each spatial dimension (x, y, z), a quadratic program minimizing the integral of squared snap (4th derivative of position) across all segments is solved simultaneously for all polynomial coefficients [6]. This produces a globally optimal trajectory with minimum control effort and continuous derivatives up to jerk.
- Inequality constraints enforce boundary conditions by sampling points along each segment to remain within environment limits.
- The optimization problem is solved using Sequential Quadratic Programming (SLSQP) with initial guesses from solving equality constraints alone.
- The resulting polynomial coefficients are evaluated to generate dense position, velocity, and acceleration trajectories.
- Collision checking is performed using environment methods on the dense trajectory. If collisions occur, options include adjusting segment times or waypoint placement.

This formulation ensures a dynamically feasible and collision-free trajectory suitable for quadrotor control.

Key components include constraint enforcement of:

- Position matching at waypoints
- Continuity of velocity, acceleration, and jerk at intermediate waypoints
- Zero velocity and acceleration boundary conditions at start and end points

The cost minimized is

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

where each segment $p_i(t)$ is represented by

$$p_i(t) = \sum_{k=0}^{7} c_{i,k} t^k.$$

This approach yields smooth 3D flight paths that are evaluated and visualized, including velocity and acceleration profiles, to verify adherence to dynamic constraints and obstacle avoidance.

Our implementation thus goes beyond classical quintic splines by optimizing the complete trajectory globally to minimize snap and guarantee jerk continuity, which reduces abrupt control inputs and improves flight stability.

Relevant control, collision-checking, and visualization routines complement this trajectory generation process to ensure safe and efficient quadrotor flight.

This approach corresponds closely to established methods for minimum snap trajectory generation for quadrotors [6], [1].

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 [7]. 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 through angular accelerations. Therefore all control inputs must be converted to angular 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 clipped 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 tuning our controller in parallel with the trajectory generator, we chose a set of values that worked well for us.

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.

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	0.4	-10	10
Position	y	0.8	0.0	0.4	-10	10
Position	z	1.0	0.0	0.45	-10	10
Velocity	v_x	1.5	0.1	0.4	-5	5
Velocity	v_y	1.5	0.1	0.4	-5	5
Velocity	v_z	2.0	0.1	0.5	-5	5

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 inputs 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. VIZFLYT AND TURING

This project utilized the VizFlyt simulator within the high-performance computing (HPC) environment provided by WPI's Turing cluster. The integration of VizFlyt with Turing enabled efficient execution of computationally intensive simulations with GPU acceleration.

A. Turing HPC Setup

We submitted jobs to the Turing cluster using a Slurm batch script `turing.sh`, which requests necessary resources and sets up the environment. The key script content is shown below:

```
#!/bin/bash

#SBATCH --mail-user=[username]@wpi.edu
#SBATCH --mail-type=ALL

#SBATCH -J p2b
#SBATCH --output=/home/[username]/logs/
    p2b_%j.out
#SBATCH --error=/home/[username]/logs/
    p2b_%j.err

#SBATCH -N 1
#SBATCH -n 8
#SBATCH --mem=64G
#SBATCH --gres=gpu:1
#SBATCH -C H100|A100|V100|A30
#SBATCH -p academic
#SBATCH -t 2:00:00
```

```
module load cuda/12.4.0/3mdaov5
module load miniconda3
module load ffmpeg/6.1.1/cup2q2r

source "$("conda" info --base)/etc/
    profile.d/conda.sh"
conda activate /home/[username]/.conda/
    envs/aerial_robotics

python3 main.py maps/mapSplat.txt
```

We configured the request to use one node with 8 CPU cores, 64 GB RAM, and one GPU compatible with NVIDIA A30, H100, A100, or V100 models. The job wall-time limit was set to 2 hours.

To monitor the job status, the command `squeue --me` was used, allowing real-time tracking of personal running jobs on the cluster.

B. GPU Hardware

At runtime, the GPU resources available on Turing were as follows (example snapshot): The GPU used during simulation on the Turing cluster was an NVIDIA A30, as confirmed by the NVIDIA System Management Interface (nvidia-smi) output. The GPU driver version was 565.57.01, with CUDA version 12.7 installed. The GPU had 24,576 MiB (approximately 24 GB) of memory available, was operating at a power level of 29 W out of a 165 W cap, and maintained a temperature of 33°C. At the time of the snapshot, no processes were running on the GPU, and it showed minimal memory usage and 0% utilization, indicating the system was idle or ready for workload execution.

This detailed information confirmed the use of an NVIDIA A30 GPU with 24 GB memory and CUDA version 12.7, providing ample computational power for GPU-accelerated VizFlyt rendering.

C. VizFlyt Simulator

VizFlyt is an open-source, perception-centric hardware-in-the-loop simulator tailored for autonomous aerial robot development [1]. It provides photorealistic first-person views and realistic flight dynamics within digitally mapped environments.

The simulator processes stitched 2D imagery and integrated map data to emulate complex robot-environment interactions. Visual outputs such as depth maps, RGB frames, and matplotlib plots are saved as image sequences in distinct subfolders within a `Render` folder. These are later used for synchronized video creation capturing both first-person and plan-view perspectives.

Our project tightly coupled the trajectory, planning and control stack with VizFlyt's rendering pipeline running on the Turing cluster, enabling efficient simulation with real-time feedback and high-quality visualization.

IV. Results

A. Path Planning Performance

Our RRT* implementation successfully generated collision-free paths in the mapSplat.txt environment. Based on our execution data:

- Converged to goal in **303 iterations** (much faster than the 3000 iteration limit)
- Generated initial path with 48 waypoints
- Path simplification reduced waypoints to 10 (79% reduction)
- Tree size at convergence: 275 nodes
- Path cost: 0.75 meters
- Start-to-goal distance: 0.71 meters (path efficiency: 94.7%)

The aggressive path simplification significantly improved trajectory smoothness and computational efficiency while maintaining collision-free guarantees.

B. Trajectory Generation Performance

Our septic minimum snap trajectory generator produced:

- 225 trajectory points from 9 segments
- Duration: 10.80 seconds for the planned trajectory
- Maximum velocity: 0.13 units/s (normalized space)
- Maximum acceleration: 0.20 units/s^2 (normalized space)
- All 225 collision checks passed
- Zero jerk discontinuities at waypoints

The minimum snap optimization successfully created smooth trajectories that respected the drone's dynamic constraints while minimizing control effort.

C. Controller Performance

The cascaded PID controller achieved excellent trajectory tracking:

- **Final distance to goal: 0.005 m** (5 mm)
- Average tracking error: 0.015 m (1.5 cm)
- Maximum tracking error: 0.054 m (5.4 cm)
- Success rate: 99.3%
- Total execution time: 15.30 seconds
- Settling at goal: < 2 seconds

The controller maintained stability throughout all flight phases. The final position was [0.70, 0.13, -0.20], achieving the goal of [0.70, 0.125, -0.20] with millimeter precision.

D. VizFlyt Simulation Performance

The system ran successfully on the WPI Turing cluster:

- VizFlyt rendering: 100+ Hz (confirmed by 10ms timesteps)
- Generated 152 FPV frames (saved every 200 timesteps)
- Real-time matplotlib visualization alongside execution
- Zero collisions throughout the trajectory
- Successfully generated combined video output at 10 FPS

E. Computational Performance

Running on the Turing cluster with CUDA 12.4:

- RRT* planning: < 1 second (303 iterations)
- Trajectory generation: < 100 ms
- Control loop: 100 Hz (10 ms per iteration)
- Total mission time: 15.30 seconds from takeoff to landing

V. Challenges and Solutions

A. Obstacle Bloating Near Boundaries

When the start or goal positions were near map boundaries, the standard obstacle bloating approach would place these points in collision. We implemented adaptive bloating that reduces margins near boundaries while maintaining safety elsewhere. This also became more prominent as we normalized the maps and tried modifying the planner and trajectory generator to match it so it is not perfect yet.

B. Controller Saturation

Initial controller gains led to actuator saturation during aggressive maneuvers. We addressed this by:

- Implementing anti-windup on integral terms
- Reducing trajectory velocities in tight spaces
- Tuning gains to prioritize smooth control over aggressive response

VI. Conclusion

We successfully implemented a complete autonomous navigation stack for quadrotor flight through obstacle environments using VizFlyt The working video can be found here [8]. The system integrates RRT* path planning, polynomial trajectory generation, and cascaded PID control to achieve collision-free navigation from start to goal positions.

Key accomplishments include:

- Zero collisions across all test runs
- Goal reaching accuracy within 4 cm
- Real-time performance at 100+ Hz
- Smooth trajectories with minimal jerk
- Successful integration with VizFlyt's photorealistic rendering

The implementation demonstrates the feasibility of deploying complex navigation algorithms on resource-constrained aerial platforms. The modular architecture allows easy swapping of planning, trajectory, and control components for future improvements.

Future work could explore:

- Informed RRT* (RRT*-Smart) for faster convergence
- Model Predictive Control (MPC) for better disturbance rejection
- Online replanning for dynamic environments
- Integration with real hardware platforms

Acknowledgments

We thank the course staff for providing the VizFlyt framework and Turing cluster access.

References

[1] K. Srivastava*, R. Kulkarni*, M. Velmurugan*, and N. J. Sanket, “Vizflyt: An open-source perception-centric hardware-in-the-loop framework for aerial robotics,” in *IEEE International Conference on Robotics and Automation (ICRA)*, 2025, accepted for publication. [Online]. Available: https://github.com/pearwpi/VizFlyt

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

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

[4] S. M. LaValle, *Planning Algorithms*. Cambridge University Press, 2006.

[5] S. Karaman and E. Frazzoli, “Sampling-based algorithms for optimal motion planning,” *The International Journal of Robotics Research*, vol. 30, no. 7, pp. 846–894, 2011.

[6] D. Mellinger and V. Kumar, “Minimum snap trajectory generation and control for quadrotors,” in *2011 IEEE International Conference on Robotics and Automation*, 2011, pp. 2520–2525.

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

[8] H. Kortus, P. Katyal, and A. Ramanathan, “[translate:combined video of our robot traversing through the mapsplat and the matplotlib plot of it on the side],” YouTube, October 2025, accessed: 2025-10-30. [Online]. Available: https://youtu.be/o8OxrDLV46w?si=Bzto8RWvIBGkU6kp