RBE595: Hands-On Autonomous Aerial Robotics - P2a: Tree Planning Through The Trees!

Shreyas Devdatta Khobragade
Department of Robotics Engineering
Worcester Polytechnic Institute
Email: skhobragade@wpi.edu

Brijan Vaghasiya
Department of Robotics Engineering
Worcester Polytechnic Institute
Email: bcvaghasiya@wpi.edu

***Abstract*—This paper presents the implementation of a motion planning and control pipeline for a quadrotor in a cluttered 3D environment. The system consists of three core modules: an RRT* path planner for global navigation, a spline-based trajectory generator for smoothing and dynamic feasibility, and a cascaded PID controller for trajectory tracking. The proposed approach was implemented in Python and tested in a simulation environment, demonstrating the ability to reach goal states while avoiding obstacles.**

***Index Terms*—RRT*, path planning, collision checking, position control, velocity control, Splines, PID Control, Autonomous UAVs**

I. Introduction

Safe and efficient quadrotor navigation in cluttered 3D environments requires solving three key problems: path planning, trajectory smoothing, and control. In this project, we designed a motion planning pipeline inspired by the PX4 flight stack and classical robotics algorithms. The system is tested in a Python simulation framework with obstacle maps provided as input text files.

II. Methodology

The implementation consists of four modules: environment parsing, path planning, trajectory generation, and control.

A. Environment Parsing

The simulation environment is defined by a rectangular boundary and a set of cuboidal obstacles specified in a text-based map file. Each line encodes either the global boundary or an obstacle block, including coordinates and RGB values for visualization. The `Environment3D` class manages this parsing and storage. A safety margin is added around each obstacle to account for the quadrotor's physical dimensions and reduce collision risk from controller overshoot. The quadrotor itself is modeled as a cuboid, ensuring collisions are checked against the vehicle volume rather than a point. A $0.5\,\text{m}$ margin provides dual protection, preventing boundary grazing and improving robustness in cluttered regions.
The environment module provides:

- **Map loading**: Extracts and stores boundaries and obstacle geometry.
- **Collision checking**: Validates points and line segments (e.g., RRT* edges) via discretized checks, incorporating both vehicle footprint and inflated margins.
- **Sampling**: Generates random free-space points for RRT* expansion.
- **Environment summary**: Reports dimensions, obstacle count, margins, and start–goal distance.

These utilities ensure the quadrotor operates strictly within free space while enabling efficient feasibility checks during path planning.

B. Path Planning with RRT*

We implemented a 3D RRT* algorithm to generate collision-free paths from start to goal. The planner grows a tree of nodes by sampling free points in the environment with a 15% probability of directly sampling the goal (goal bias). For each iteration:

- **Sampling:** Random free points are generated within the map boundaries.
- **Nearest node:** The closest existing node x_{near} is found using Euclidean distance

$$d(x, y) = \|x - y\|_2.$$

- **Steer:** A new node x_{new} is created by stepping from x_{near} toward the sample with a fixed step size.
- **Collision check:** Line segments are discretized to ensure x_{new} lies in free space.
- **Parent selection:** Among neighboring nodes within radius $r(n) = \gamma\big(\frac{\log n}{n}\big)^{1/3}$, the parent that minimizes cumulative cost is chosen.
- **Rewiring:** Neighboring nodes are rewired through x_{new} if it reduces their path cost.

The process continues until a node reaches within the goal radius, at which point the path is extracted by tracing back through parent links. The output is a sequence of waypoints representing a feasible, near-optimal trajectory.
RRT* Parameter Selection The performance of RRT* is sensitive to parameters such as the number of iterations, step size, goal radius, and search radius. In our implementation, these values were chosen adaptively based on the Euclidean distance d between the start and goal positions:

- **Maximum iterations:**

$$N_{\text{iter}} = \max(4500, \max(1000,\ \lfloor 150 \cdot d \rfloor))$$

ensuring sufficient exploration for larger environments.
- **Step size:**

$$\Delta = \min(1.5,\ d/10),$$

which allows longer steps in open spaces while remaining conservative in tighter environments.
- **Goal radius:**

$$r_g = \max(0.8, \min(1.5,\ d/15)),$$

providing a flexible termination threshold that scales with the planning distance.
- **Search radius:**

$$r_s = 4.6 \cdot \Delta,$$

which balances the number of neighbors considered for rewiring with computational efficiency.

These adaptive parameters allow the planner to remain efficient across maps of different sizes while preserving the probabilistic completeness and asymptotic optimality of RRT*.

C. Trajectory Generation

The raw RRT* output is jagged and dynamically infeasible for quadrotor flight. To obtain smooth and executable trajectories, we fit a quintic B-spline $r(u)$ through the discrete waypoints, with parameter $u \in [0, 1]$. The quintic formulation ensures continuity of position, velocity, and acceleration, which is essential for stable control. The spline is reparameterized by arc length $s(u)$ to achieve uniform progression along the path. A trapezoidal (or triangular) velocity profile $v(t)$ is then imposed, subject to maximum velocity $v_{\max}$ and acceleration $a_{\max}$. The distance profile $s(t)$ is defined as:

$$s(t) = \begin{cases} \frac{1}{2} a_{\max} t^2, & t \le t_a, \\ d_a + v_{\max}(t - t_a), & t_a < t \le t_c, \\ L - \frac{1}{2} a_{\max}(T - t)^2, & t > t_c, \end{cases} \tag{1}$$

where $t_a = v_{\max}/a_{\max}$ is the acceleration duration, $d_a = \frac{1}{2} a_{\max} t_a^2$ is the distance covered during acceleration, T is the total trajectory time, and L is the total spline arc length. For short paths where $L < 2d_a$, a triangular velocity profile is used instead. Velocities and accelerations along the trajectory are obtained from the spline derivatives using the chain rule:

$$\dot{r}(t) = r'(u)\, \dot{u}, \tag{2}$$

$$\ddot{r}(t) = r''(u)\, \dot{u}^2 + r'(u)\, \ddot{u}, \tag{3}$$

with $\dot{u} = v(t)/\|r'(u)\|$ and $\ddot{u}$ derived from the acceleration profile $a(t)$. The output is a set of time-stamped trajectory points $\{r(t)\}$, velocities $\dot{r}(t)$, and accelerations $\ddot{r}(t)$. A final uniform time-scaling step ensures that all velocity and acceleration constraints are satisfied, guaranteeing both smoothness and dynamic feasibility.

D. Cascaded PID Controller

To track the reference trajectory, we implemented a cascaded control architecture inspired by the PX4 stack. The structure consists of nested loops, each regulating progressively faster dynamics:

- **Position loop (outer)**: A PID controller regulates position error

$$e_p = p_{\text{des}} - p,$$

generating velocity setpoints v_{sp}. Tuned gains:

$$K_p = [0.6,\ 0.6,\ 0.8], \quad K_i = 0, \quad K_d = [0.1,\ 0.1,\ 0.2].$$

- **Velocity loop (middle)**: A PID controller regulates velocity error

$$e_v = v_{\text{sp}} - v,$$

producing acceleration setpoints a_{sp}. Tuned gains:

$$K_p = [0.6,\ 0.6,\ 0.8], \quad K_i = [0.1,\ 0.1,\ 0.2], \quad K_d = [0.2,\ 0.2,\ 0.3].$$

- **Attitude control**: Desired acceleration is mapped to a quaternion orientation q_{sp}, aligning the body z-axis with the thrust vector. The quaternion error

$$q_{\text{err}} = q^{-1} \otimes q_{\text{sp}}$$

yields desired angular rates pqr_{sp}. A proportional controller with time constant $\tau = 0.8$ is applied, with yaw control scaled down by 0.4 to prioritize roll and pitch tracking.
- **Angular rate loop (inner)**: PID controllers regulate roll, pitch, and yaw rate errors to compute torque commands (τ_x, τ_y, τ_z). Tuned gains:

$$K_p = 6.0, \quad K_i = 0, \quad K_d = 0.4.$$

The collective thrust is computed as

$$T = m\, \|a_{\text{sp}} - g\|,$$

where m is the quadrotor mass and g is gravity. Rotor thrusts u_1, u_2, u_3, u_4 are obtained through a linear mixer:

$$\begin{bmatrix} u_1 \\ u_2 \\ u_3 \\ u_4 \end{bmatrix} = \frac{T}{4} + M \begin{bmatrix} \tau_x \\ \tau_y \\ \tau_z \end{bmatrix},$$

with M representing the allocation matrix. The outputs are clipped to actuator limits to ensure feasibility. This cascaded structure, with carefully tuned PID gains, ensures stable trajectory tracking: slow dynamics (position) are corrected by the outer loops, while fast dynamics (attitude and rates) are stabilized by the inner loops, enabling smooth and collision-free flight.

E. Live Quadrotor Simulator

To integrate all components, a real-time simulation framework was implemented in `Simulator.py`. The system executes the full pipeline in three phases with live 3D visualization:

- **Phase 1: RRT* Planning** The simulator incrementally builds the tree, visualizing sampled nodes, edges, and rewiring in real-time.
- **Phase 2: B-spline Trajectory Generation** The extracted waypoints are fit with a B-spline, and time-parameterization is applied using a trapezoidal velocity profile. Velocity vectors are displayed along the path to verify feasibility.
- **Phase 3: Trajectory Execution** The quadrotor dynamics are simulated and the cascaded PID controller tracks the reference trajectory, while the visualization shows the quadrotor's live position, executed trail, and distance to goal.

III. Results

We evaluated the framework on training and test maps. The video outputs from rotplot.py are included in this link.

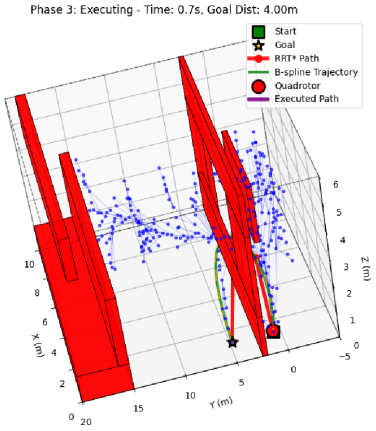

Fig. 1: Results on Map 1.

IV. Conclusion

This work presented an integrated pipeline for quadrotor navigation in cluttered 3D environments, combining RRT* path planning, spline-based trajectory generation, and cascaded PID control. The adaptive planning parameters ensured scalability across different maps, while the quintic B-spline with time-parameterization produced smooth and dynamically feasible trajectories. The cascaded control structure, with carefully tuned gains, enabled stable tracking in simulation. Overall, the framework demonstrated the ability to plan, smooth, and execute collision-free trajectories, laying a foundation for future extensions such as real-world deployment and advanced perception-driven planning.

Acknowledgment

The authors thank the instructor, teaching assistant and dataset providers.

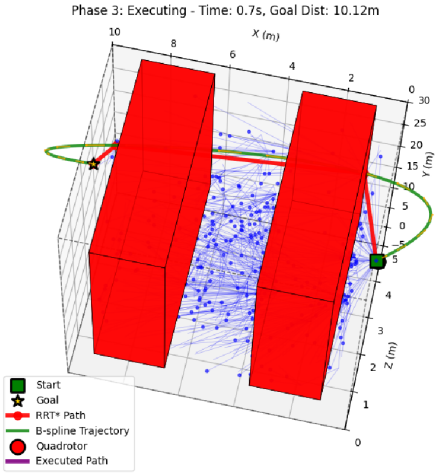

Fig. 2: Results on Map 2.

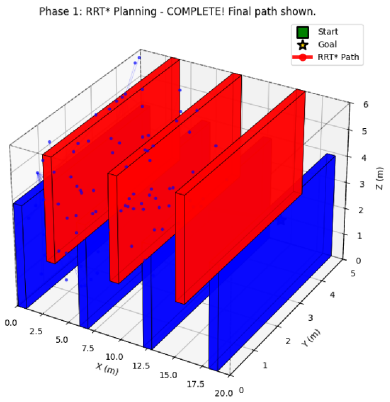

Fig. 3: Results on Map 3.

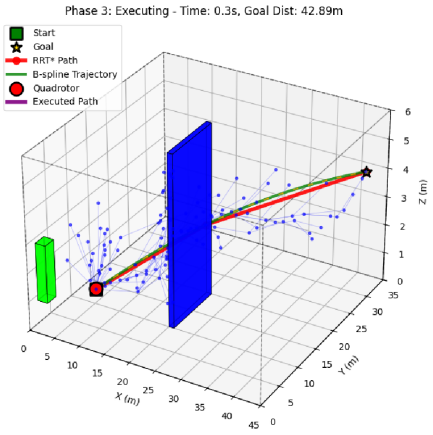

Fig. 4: Results on Map 4.

References

[1] N. J. Sanket, "Tree Planning Through The Trees!" School of Engineering and Applied Science, University of Pennsylvania, 2025. Email: nitin-

san@seas.upenn.edu.
[2] RBE595 Project 2, “Phase 2a Documentation,” *RBE595 Course Website*, Fall 2025, https://rbe549.github.io/rbe595/fall2025/proj/p2a/.
[3] YouTube Video, ”RRT* Algorithm Explained” https://www.youtube.com/watch?v=_aqwJBx2NFk.
[4] YouTube Video, ”Motion Planning: Rapidly Exploring Random Trees (RRT): Algorithm Implementation Step by Step!” https://www.youtube.com/watch?v=OXikozpLFGo&t=866s.