AER1217 – Winter 2024

Development of Unmanned Aerial Vehicle

Final Project Report

Kevin Hu, Yifeng Han, Chris Anderson

University of Toronto

Date of submission: April 28, 2204

1.0 Overview of Algorithms and Equations Used

1.1 Planning Algorithm Overview

The main planning algorithm for this project, RRT*, was selected for its ability to quickly find valid, albeit suboptimal, paths to a goal. Unlike search-based algorithms like A*, RRT* is more efficient and can dynamically improve paths over time, adding flexibility to the design. This implementation of RRT* employs biased sampling, using a hyperparameter to dictate the likelihood of samples directly sampling the goal position. This bias accelerates goal discovery and may yield lower-cost paths, reducing dependency on sample proximity to the goal. For every biased sample, a path is temporarily created from the closest node to the goal to the goal, and discarded if there is a collision. The bias level is critical—it must balance between efficiently finding feasible paths and encouraging exploration when direct paths are obstructed. Our RRT* algorithm assumes no path constraints, allowing the drone unrestricted movement, mirroring its real-world capabilities.

1.2 Problem-Solving with RRT*

In order to solve the problem of passing through several gates in the correct order, multiple instances of RRT* were used. Each instance of RRT* was used to find a path between a gate and the subsequent gate except for the first instance where RRT* was used to find a path between the starting position and the first gate. The reason for using multiple instances of RRT* is because RRT* can only find one goal position and in this problem, multiple goal positions (locations of each gate) need to be traveled to.

For each gate, there are two points a set distance away from the center of the gate, and a path can be formed between them such that it is perpendicular to the gate and is the optimal path for the drone to take since the path is the furthest away from the gate frame at all times. The endpoint for RRT* is one of the points on the path segment. Each instance of RRT* will end at one of the endpoints of the path segment for a particular gate and the next instance of RRT* will begin at the other endpoint of the path segment for the same gate. Each path segment is added to the path between the paths that each instance of RRT* returns.

The way RRT* chooses which endpoint of the path segment to expand to for a particular gate is to expand to the endpoint closest to the starting point of that instance of RRT*, which in general would allow RRT* to generate the shorter path. This method of choosing an endpoint was used because of its simplicity and consistency in finding a collision free path.

1.3 Method for Collision Detection

The problem was initially designed to use both 3D and 2D configuration spaces (search spaces) for increased design flexibility. The collision detection method works in both 2D and 3D configurations but may not necessarily be the most efficient method for 2D collision detection.

Each obstacle was defined as a cylinder in 3D space. Due to the uncertainty of the obstacle position, the radius of the cylinder was defined to be the radius of the obstacle + the uncertainty of the obstacle position + a safety distance. For each of the gates, they were defined as a cylinder in 3D space (where the gate frame rests on), and a gate frame with a collision area consisting of 4 rectangles. The 4 rectangles are placed so that they surround the gate completely with some extra space in between for safety. Each of the four rectangles are perpendicular to the ground and surround each side of the gate, including the two sides the drone is supposed to pass through. The reason for this definition scheme is because the collision detection between a line segment and a cylinder in 3D space and the collision detection between a line segment and a rectangle in 3D space are efficient and straightforward.

1.4 Collision Detection Between Line Segment and Cylinder

First the algorithm checks if there is a collision between a line segment and an infinite cylinder or a cylinder with infinite height. A line segment can be defined parametrically as a line, with a starting point P0, an unnormalized (not magnitude 1) normal vector in the direction of travel between the start and end point N, and a time t, where at t = 0 is the starting point and at t = 1 is the endpoint. The equation of an infinite cylinder is $(x-x_c)^2 + (y-y_c)^2 = R^2$, where x_c and y_c are the coordinates of the center of the cylinder and R is the radius of the cylinder. To find a possible intersection point between the line segment and the infinite cylinder, the equation of the line segment can be substituted into the equation for the infinite cylinder which results in a quadratic equation in terms of t. There is a collision if there is a solution for t that is a sensical value between 0 and 1. The coefficients for the quadratic equation and its derivation are shown below. The algorithm first checks if the discriminant for the quadratic equation is positive. If negative, there is no collision since t would have to be an imaginary number. If positive the algorithm then finds the two roots of the quadratic equation and checks if they are between 0 and 1. If so, the algorithm checks if the solution would cause the z coordinate of the intersection to be below the height of the cylinder which would mean that there is a collision.

$$(P_{0x} + tN_x - x_c)^2 + (P_{0y} + tN_y - y_c)^2 = R^2$$

$$(N_x^2 + N_y^2)t^2 + (2P_{0x}N_x + 2P_{0y}N_y - 2N_x x_c - 2N_y y_c)t + (P_{0x}^2 + P_{0y}^2 + x_c^2 + y_c^2 - 2P_{0x}x_c - 2P_{0y}y_c - R^2) = 0$$

$$a = N_x^2 + N_y^2$$
$$b = 2P_{0x}N_x + 2P_{0y}N_y - 2N_x x_c - 2N_y y_c$$
$$c = P_{0x}^2 + P_{0y}^2 + x_c^2 + y_c^2 - 2P_{0x}x_c - 2P_{0y}y_c - R^2$$

Equation 1: For Collision Detection Between Line Segment and Cylinder

1.5 Collision Detection Between Line Segment and Rectangle

For every gate, the algorithm does 4 collision checks between a line segment and a rectangle in 3D space. First the algorithm checks for a collision between a line segment and a plane parallel to the rectangle, then checks inside the bounds of the rectangle for a collision. Each rectangle is characterized by two vectors where the length of the vector is the length of the

corresponding side of the rectangle, and a corner point R_0 which is on the rectangle. The same parameterization of the line segment is used except this time the normal vector of travel is normalized and is called D instead of N, and time t is between 0 and the magnitude of the unnormalized normal vector in the direction of travel. The algorithm first calculates the normal vector N to the rectangle using the cross product of the two vectors characterizing the rectangle. Then the algorithm solves for the time by solving the equation below. If the time is a nonsensical value less than 0 or greater than the magnitude of the unnormalized normal vector in the direction of travel, then there is no collision. If the time is a sensical value, the intersection point is calculated and checked to see if it is inside the bounds of the rectangle. If the intersection is within the bounds, there is a collision.

$$t = \frac{(R_0 - P_0) \cdot N}{D \cdot N}$$

Equation 2: Solving for Time t

1.6 Control Commands

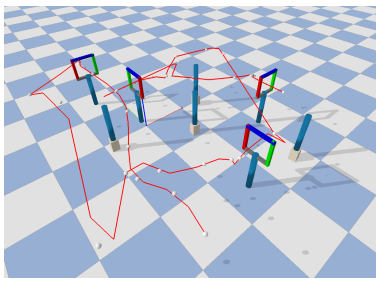
The control system utilized high-level commands like takeoff() and land() for basic maneuvers, complemented by low-level cmdFullState() commands for precise navigation through the gates. The trajectory planning involved using RRT* to generate waypoints, which were then smoothed using CubicSpline interpolation from the scipy package. This interpolation ensured a maximum distance of 0.3m between points to align closely with the waypoints and mitigate collision risks due to sensory noise.

The trajectory was segmented according to the control frequency and a predetermined gaining factor of 40, balancing the drone's speed with adherence to the planned route. The simulation resulted in a total of 1350 simulation iterations, with the initial 90 for takeoff, followed by 1230 for navigation through the gates, and the remainder for landing. This structure ensured a smooth takeoff and sufficient time for the drone to complete its course, maintaining a safe and efficient flight path within the set simulation constraints.

2.0 Simulation Results

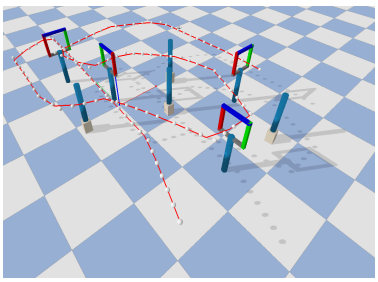
Two versions of simulation results are presented to compare algorithms' performance and improvements. Figure 1 (a) didn't set the maximal waypoint distance, so the resulting trajectory didn't perfectly lie on all the waypoints, and overshot at the places where sharp turning happened. Then we improved our algorithm by setting up the maximal waypoint distance and fixing some bugs. In Figure 1 (b), the trajectory was smoother and fitted all the waypoints perfectly by setting the maximal waypoint distance to 0.2m. With 35 as the gaining factor, the drone flew precisely following the trajectory.

Figure 2 shows the trajectory planned with the gate order of 1-3-4-2-1-4 for the drone racing, and the maximal waypoint distance of 0.3m. The trajectory flying through the gate 1 second time is not as straight as desired.



(a)

Figure 1. Trajectory generated to pass the gates in the order of 4-1-3-2.



(b)

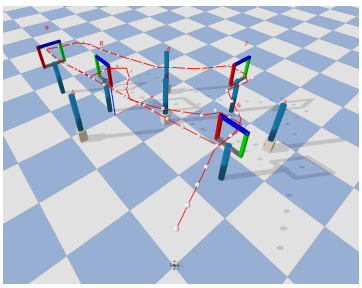


Figure 2. Simulation in the order 1-3-4-2-1-4

3.0 Challenges Encountered

3.1 Debugging

One main challenge encountered for the project was creating a system that would allow debugging the code to be easier and for the code to run successfully. To solve this issue, three stages of testing were created. The first stage of testing required testing the planning algorithm only on a simple Python script without the simulation environment. Obstacles were plotted using the Matplotlib Python library and lines were plotted representing the connections between nodes and the found paths. Plotting proved to be somewhat challenging and time consuming since our group did not have too much knowledge about Matplotlib. The second stage of testing required testing the planning algorithm in simulation and adjusting parameters until the drone passed successfully. The third stage of testing was the real life test and the results were taken into account to make final adjustments to the algorithm parameters.

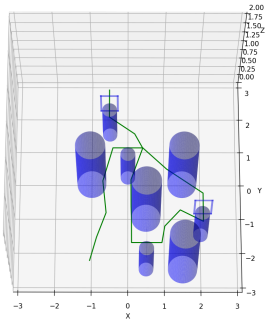


Figure 3: RRT* Plotting for First Stage of Testing

3.2 Path Planning/RRT* algorithm

We set a random seeding for the path planning algorithm, but the algorithm always outputs different paths, which brings a lot more uncertainty and trouble for parameter tuning in the following stages.

3.3 Planned Trajectory Performing in Real Flight

In order to generate a trajectory from the waypoints, we used CubicSpline as a curve-fitting tool to fit the waypoints. To ensure the trajectory aligns with the RRT* path, more waypoints are required. However, waypoints are discrete so that the fitted trajectory could have sharp turns which cause the drone to crash at high speed. We tested our controller in real flights at a relatively low speed, the drone went out of control during 90-degree turns multiple times. We finally solved it by adjusting RRT* parameters to make it find a path with a slightly larger curvature and increasing the gaining factor from 25 to 40 to slow down the drone's speed.

Some maneuvers could only be done in the simulation environment. To successfully implement the algorithms in real flight, we had to tune our controller and the path planner according to how the drone behaved in real flight.

3.4 Fitting a Trajectory to Waypoints

We initially designed two endpoints on both sides of each gate to ensure the drone passes through the gate from the center. The current method of trajectory fitting sometimes makes the actual path deviate, which causes the drone to crash on the gate frame. To solve this issue, one possible way could be separating the entire trajectory into different segments, and only connecting the endpoint from the previous gate to the endpoint from the next gate using the waypoints. Then, manually connect the endpoints for a gate to ensure straight paths.

3.5 Design a More Robust Controller

We adopted the general structure from the original controller, so the performance of flight commands is limited to the simulation environment when dealing with position control and agile maneuvers like sharp turning, as well as adjusting flight speed at different stages.

4.0 Conclusion

The project successfully demonstrated the application of RRT* and our controller in drone navigation. Although our controller limits on performing agile maneuvers, the stability has been proved in real-life tests through take-off, passing through all the gates without collision, and successful landing. Future work could focus on further optimizations and testing the controller in real flight by addressing the challenges mentioned above.

References

1. Liu, Hugh, "AER1517: Development of Autonomous Unmanned Aerial Systems", Course Lectures, Winter 2024, University of Toronto
2. Pybullet instruction manual: https://github.com/bulletphysics/bullet3/tree/master/docs. Last accessed: 28th April 2024.
3. LaValle, S. M. (2006). Planning Algorithms. Cambridge University Press.
4. J. J. Kuffner and S. M. LaValle, "RRT-Connect: An efficient approach to single-query path planning," in Proc. IEEE Int. Conf. on Robotics and Automation (ICRA), San Francisco, CA, USA, 2000, pp. 995-1001.
5. O. Khatib, "Real-time obstacle avoidance for manipulators and mobile robots," in Proc. IEEE Int. Conf. on Robotics and Automation (ICRA), St. Louis, MO, USA, 1985, pp. 500-505.
6. E. Frazzoli, M. A. Dahleh, and E. Feron, "Real-time motion planning for agile autonomous vehicles," Journal of Guidance, Control, and Dynamics, vol. 25, no. 1, pp. 116-129, Jan.-Feb. 2002.
7. D. Ferguson, N. Kalra, and A. Stentz, "Replanning with RRTs," in Proc. IEEE Int. Conf. on Robotics and Automation (ICRA), Orlando, FL, USA, 2006, pp. 1243-1248.