Team Apache Stealth: Fly through "Real" trees!

Ankit Mittal
Department of Robotics Engineering
Worcester Polytechnic Institute
Email: amittal@wpi.edu

Rutwik Kulkarni
Department of Robotics Engineering
Worcester Polytechnic Institute
Email: rkulkarni1@wpi.edu

***Abstract*—This project involves the implementation of Motion Planning and Control Stack for a real quadrotor using a predefined map, the NVIDIA Jetson Orin Nano for computation, and the DJI Tello Edu Quadrotor for execution. The system involves path planning, waypoint generation, and position control to autonomously navigate from a defined start to a goal location while avoiding obstacles. The quadrotor assumes perfect knowledge of the environment, takes off and lands at specific locations, and prohibits flying over obstacles. The choice of a position or velocity controller was flexible in the problem statement, therefore both were used with the primary objective being efficient and safe navigation. The quadrotor's pose is visualized on a map using Blender throughout the project.**

I. Introduction

This project is centered around implementing the RRTstar Motion Planning Algorithm a real quadrotor, utilizing a predefined map, the computational capabilities of the NVIDIA Jetson Orin Nano, and the flight execution capabilities of the DJI Tello Edu Quadrotor. Our primary objective is to achieve autonomous and safe trajectory planning and execution, guiding the quadrotor from a defined starting point to a predetermined goal location within a mapped environment. Key components of this project encompass path planning, waypoint generation, and precise position control.

Our project relies on a pre-established map of the environment, provided in a specific file format. This map delineates the environment's coordinates and outlines obstacles in the form of boxes. Start and goal locations are identified by distinctive white strips. Overflight of obstacles is prohibited for safety purposes.

The implementation strategy draws from our previous work in Project 2a where we implemented RRTstar and generated a smooth trajectory using Quintic Polynomials. We integrated it with the DJITelloPy library for controlling the DJI Tello Edu Quadrotor. We implemented the trajectory both on a Position and Velocity Controller in the live demo section of this Project. Given below is the link to Videos of of demonstration for the training set Map Environment Implementing a Velocity controller.

Link To Videos: **Click Here**

II. Environment Setup(Map Reader for Real World)

As part of the initial setup, program needs to read environmental data from a text file. This text file should contain obstacle dimensions formatted in the following manner.

- **Boundary:** $x_{min}\ y_{min}\ z_{min}\ x_{max}\ y_{max}\ z_{max}$
- **Block:** $x_{min}\ y_{min}\ z_{min}\ x_{max}\ y_{max}\ z_{max}$
- **Color:** $r\ g\ b$

Here $x_{min}\ \ y_{min}\ \ z_{min}$ represents the lower left corner coordinates of the block/boundary and $x_{max}\ \ y_{max}\ \ z_{max}$ represents the upper right coordinates of the block/ boundary. Script reads the environment, plots it in Blender and later using this script, we path a plan and generate a trajectory for the real robot to execute.

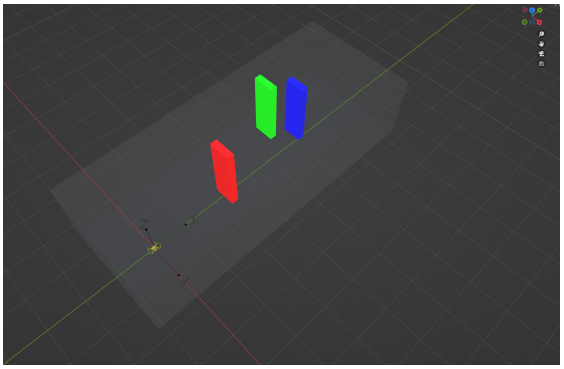

Fig. 1: Blender Representation of the Real World Map

III. Implementation Strategy and Challenges

In the early stages of our project, our approach centered around the utilization of the `go_xyz_speed()` function from the DJI Tello library for position control. This function allowed us to input waypoints generated by our RRTstar algorithm from Project2a. However, we encountered a significant challenge with this approach. The `go_xyz_speed()` function required position inputs in integer values representing centimeters, and the differences between successive trajectory points were often minuscule, rendering them insufficient for valid position inputs. In response to this limitation, we adopted an alternative strategy, making use of navigation points. Despite the inbuilt position control being commendably accurate, we observed that the quadrotor consistently came to a complete stop at each waypoint, resulting in undesirable delays and inefficiencies in its path-following behavior.

To address this issue and enhance our control capabilities, we transitioned to a velocity control approach. This involved employing the `rc_control()` function, which allowed us to publish velocity commands to the quadrotor at a fixed rate. This shift in control methodology enabled us to directly implement our time-parametrized velocity trajectory, derived from the trajectory generator, into the quadrotor's control system. However, it's important to note that the velocity input range supported by the DJITelloPy library was restricted to a range of -100 to +100 cm/sec, which is equivalent to -1 to 1 m/s. Consequently, we needed to carefully adjust our time parameters to ensure that the peak velocity remained below 1 m/s, aligning with the library's constraints.

Notably, for this project, we chose not to implement a custom controller, be it for position or velocity control. Instead, we exclusively relied on the drone's built-in controller. This approach had its limitations, particularly in terms of fine-tuning control gains. To overcome this constraint, we had to resort to tuning the `tello.sleep()` function, which determined the frequency at which the quadrotor received velocity commands. While this adjustment ultimately allowed us to achieve satisfactory control results, it was still marked by a degree of unpredictability, especially when compared to the precision of the position controller.

In our efforts to monitor the quadrotor's progress, we accessed odometry data through functions like `get_x_speed()`, `get_y_speed()`, and `get_z_speed()`. However, it's crucial to acknowledge that these functions provided measurements in decimeters and returned rounded values. Consequently, the logged odometry data did not offer a fully accurate representation of the quadrotor's path traversal. In reality, the quadrotor followed a considerably smoother trajectory than what was reflected in the logged data, and this distinction becomes evident when observing the demonstration video of our project.

IV. Results

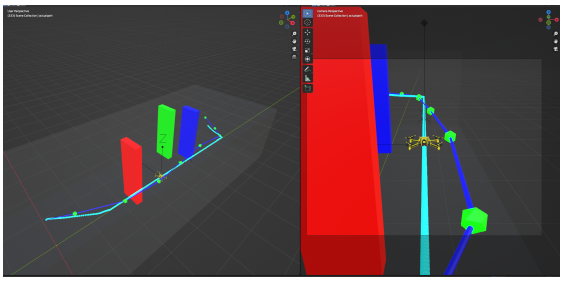

Fig. 2: Path Generated by RRTstar planner(Blue) v/s Traversed Trajectory of the Robot in Real Environment(Magenta)

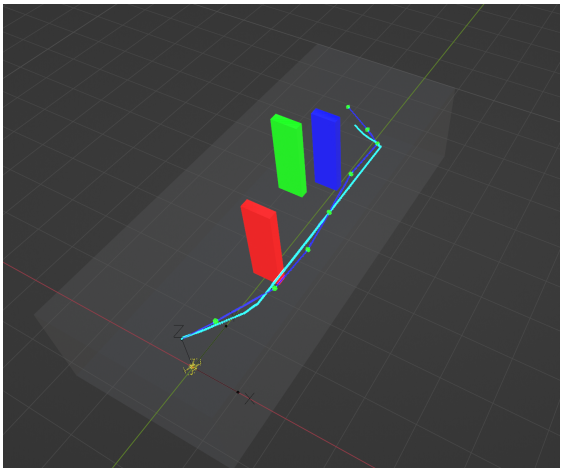

Fig. 3: Path Generated by RRTstar planner(Blue) v/s Traversed Trajectory of the Robot in Real Environment(Magenta)

- Link To position controller run: **Click Here**
- Link To velocity controller run: **Click Here**

V. Observations

A. Position Control vs. Velocity Control

- **Position Control:** We observed that the position controller provided reasonably accurate position control and exhibited greater reliability compared to velocity control. However, a notable drawback was that it consistently came to a complete stop at waypoints, resulting in slower execution.
- **Velocity Control:** In contrast, the velocity controller offered accurate position control less frequently when compared to the position controller. However, it demonstrated a smoother trajectory, leading to faster execution. This approach of control holds the potential for scalability in terms of higher velocities. The issue of reliability can potentially be addressed through improved odometry and perception capabilities.

B. Reliance on IMU for Pose Estimation

- Both the position and velocity control implementations relied on the inbuilt drone IMU for pose estimation. However, it became evident that this reliance on a single source of odometry may be insufficient for ensuring a robust navigation system.
- To enhance the system's reliability and perception, it is imperative to consider integrating additional sources of odometry data beyond the IMU, allowing for more accurate and consistent pose estimation.

These observations highlight the trade-offs between position and velocity control and emphasize the need for improved odometry and perception systems to enhance the overall performance and reliability of the navigation system.

VI. Conclusion

We explored two primary control strategies: position control, which exhibited reliability but slower execution due to waypoint stops, and velocity control, which provided a smoother trajectory and faster performance at the cost of consistent accuracy.

One significant observation was the reliance on the drone's inbuilt IMU for pose estimation in both control implementations. This single-source odometry approach may be insufficient for robust navigation, emphasizing the need for additional odometry sources for improved perception and reliability.

This project serves as a foundation for future developments in autonomous navigation systems,

where a balance between accuracy, reliability, and performance is crucial.

VII. Acknowledgment

The author would like to thank Prof. Nitin Sanket and the TA of this course RBE595.

References

[1] DJITelloPy **Link**
[2] Blender Plot API. **Link**