Real-time Autonomous Quadrotor Navigation and Control with DJI Tello and Jetson Nano

Ankush Singh Bhardwaj
abhardwaj@wpi.edu

Sri Lakshmi Hasitha Bachimanchi
sbachimanchi@wpi.edu

Anuj Pradeep Pai Raikar
apairaikar@wpi.edu

***Abstract*—This project presents the implementation of the navigation stack including planning and control on the DJI Tello drone in the real-world testing environment. The trajectory calculation and visualization is performed on the NVIDIA Jetson Orin Nano and is executed on the DJI Tello edu drone.**

I. Environment

The environment comprises tall "tree-like" structures represented as boxes, extending from the floor to the ceiling. These boxes are defined in the "mapx.txt" file and serve as obstacles in our trajectory calculations. The start and the goal locations in addition to the positions of boxes are used to determine the flight path of the drone. The real-world scenario can be seen in Fig 1.

Fig. 1. Sample Environment

II. Implementation

A. *Reading the Map*

Before testing on real-world environment, we simulated the map and the navigation of the drone on Blender. We used the training map given in ".txt" file for simulating the environment with boundaries and obstacles. The file contains the boundary coordinates and the positions of block obstacles that exist within the environment of the drone. We considered bloated dimensions of the obstacles for avoiding collisions to calculate the trajectory. We have considered the bloat dimensions to be half of the size of the drone considered as a cuboid with width and length as 0.3m and height as 0.5m.

B. *Implementation of Path Planner*

We have used RRT* to obtain paths from start to goal.

Rapidly Exploring Random Trees - Star (RRT*)

The algorithm attempts to find a path from a start point to a goal point while avoiding specified obstacles, expanding a tree of potential paths across the configuration search space until the goal is reached.

Deeper Look

Tree Expansion: The algorithm iteratively expands the tree from the start position by performing the following steps for a specified number of iterations *numNodes*:

1. **Random Sampling:** Generate a random point in the search space(p1)

2. **Find Nearest Node:** Find the closest node(p2) to the randomly sampled point(p1) from the existing nodes.

3. **Steering:** Generate a new node(p3) by steering from the nearest node(p2) towards random node p1(by some pre-defined distance or until p1 is reached).

4. **Collision Check:** Verify that the path from closest node(p2) to new node(p3) doesn't intersect with any obstacle.If not p3 is a valid node

Node Addition: Optionally, the algorithm explores if p3 can be connected to other nearby nodes in a manner that might provide a lower-cost path.
Rewiring: It checks other nodes and if a path from p3 to another node is shorter than the existing path to that node (and doesn't intersect with obstacles), the parent of that node is changed to p3.

5. **P3 is valid:** A new node p3 is added to the tree, with its parent set to p2 and its cost calculated as the distance from p2 plus the cost of p2

6. **P2 is within a certain distance from the goal**: it

checks whether a direct path from p2 to the goal is free from obstacles. If it is, it adds the goal to the tree, connected to p3, and terminates the algorithm.

The algorithm is implemented in 3D Space. Distances and we chose the following parameters:

a. The minimum acceptable distance to goal as 1.0
b. The number of nodes being sampled as 5000.
c. Neighbor search radius as 50

Table 2: RRT Algorithms

Algorithm 2.
T = (V, E) ← RRT*(z_{ini})

1 $T \leftarrow$ InitializeTree();
2 $T \leftarrow$ InsertNode(Ø, z_{init}, T);
3 ***for*** i=0 to i=N ***do***
4 $z_{rand} \leftarrow$ Sample(i);
5 $z_{nearest} \leftarrow$ Nearest(T, z_{rand});
6 (z_{new}, U_{new}) ← Steer ($z_{nearest}$, z_{rand});
7 ***if*** Obstaclefree(z_{new}) ***then***
8 $z_{near} \leftarrow$ Near(T, z_{new}, $|V|$);
9 $z_{min} \leftarrow$ Chooseparent (z_{near}, $z_{nearest}$, z_{new});
10 $T \leftarrow$ InsertNode(z_{min}, z_{new}, T);
11 $T \leftarrow$ Rewire (T, z_{near}, z_{min}, z_{new});
12 ***return*** T

Fig. 2. Pseudocode for RRT*

C. *Trajectory Generation*

To make the paths generated by RRT* smooth and dynamically feasible, we converted the path into a trajectory.

1) **Fitting a Spline:**

- We are creating quintic trajectories. We are considering pairs of waypoints, taking the segment length and dividing by average velocity to find the time taken to traverse between the individual segment lengths. Acceleration at the waypoints are always zero.

  Position, Velocity and Acceleration Trajectories are given by:
  $x = a_0 + a_1 t + a_2 t^2 + a_3 t^3 + a_4 t^4 + a_5 t^5$
  $\dot{x} = a_1 + 2a_2 t + 3a_3 t^2 + 4a_4 t^3 + 5a_5 t^4$
  $\ddot{x} = 2a_2 + 6a_3 t + 12a_4 t^2 + 20a_5 t^3$

- **Bounding conditions**:
  1. **Start and the First way-point:** the initial velocity and acceleration are zero, while the final velocity is the assumed average velocity(**v**).
  2. **Intermediate way-points:** The velocity profiles are such that the initial and final velocity are both set to **v**.
  3. **Final way-point:** The initial velocity is **v** and the final velocity and acceleration is zero.

- Through solving the equations above we get the coefficients for the spline equations and the time stamps corresponding to the positions in the path. The resultant spline is pruned and optimized.

2) **Pruning the Trajectory**:

The waypoints generated previously are checked for collision with our obstacles and the unnecessary waypoints are eliminated. The profiles for position, velocity and acceleration are stored in a csv file and is saved on the computer.

3) **Smoothing the Trajectory**:

We are performing **gradient descent smoothing** on the path (a sequence of points) by iteratively adjusting each point (except for the first and the last ones) based on its original position and the positions of its neighbors. The adjustment is governed by two weights (weight data and weight smooth) and continues until the total change for all points in one iteration is less than a defined tolerance.

Deeper Look: In repeated cycles, every point (except the first and last) is nudged: Partly toward its original position in the un-smoothened path. Partly toward the average of its neighbors in the smoothed path. These nudges continue until the points stop moving significantly (i.e., total movement across all points is below a tiny threshold).

D. *Controller Strategy and Tuning Gains*

The quadrotor is tuned to follow the desired trajectory generated from the path planned by RRT* algorithm without collisions. The controller designed for the quadrotor is a cascaded controller with outermost loop as the position controller and the penultimate loop as the velocity controller. The position controller uses PID controller with 3 sets of gains for x,y and z for a stable position control for positioning the quadrotor at desired locations. The velocity controller again uses PID controller with 3 sets of gains.

The position control loop and the velocity control loop are tuned individually with the given sample trajectory file with position, velocity and acceleration values corresponding to a helical trajectory. And the tuned parameters are verified with few more trajectory files to check the hovering of the quadrotor at a fixed location. The results of the tuned controller with the desired and actual positions, velocities are shown in the plots below.

The gains tuned are shown in TABLE 1

TABLE I
PID CONTROLLER PARAMETERS

Parameter	K_p	K_i	K_d
position_x	1	0.1	0
position_y	1	0.1	0
position_z	1	0	0
velocity_x	1	0	0
velocity_y	1	0	0
velocity_z	3	0.3	0.1

PID gains for X and Y directions are kept same as it is a symmetric drone with same dynamics along both the axes. And the velocity control loop is tuned first by keeping the gains of the position control loop to 0 and considering one gain.

III. TESTING IN REAL ENVIRONMENT

The trajectory generated from the path planner is tested on the DJI Tello in real given tested map scenario with obstacles in the given time of 15 minutes. The given test map is read from the "map.txt" file and the trajectory path in the .csv path is given to the quadrotor to follow using Jetson Nano. The DJI Tello was able to execute the trajectory decently reaching the goal position of the environment. The video of the testing is attached as in the submitted files.

Jetson Nano and DJITelloPy

We have used NVIDIA Jetson Orin Nano to send commands to the DJI Tello connected over wifi. We have done this using the DJITelloPy python library that allows control of the drone using Python. We used methods like *connect()*, *takeoff()* and *go_xyz_speed()* to precisely maneuver and control the drone. Additionally we have used methods *get_height* and *get_current_state* for monitoring and tracking the state of the drone.

The trajectory that is produced from feeding the relevant map is stored in a csv file in a local location. Each column of the file contains the position, velocity and acceleration profiles at each waypoint along the time-bounded quintic trajectory. Another program interprets the positions from the csv file and processes the relative

IV. INTERESTING OBSERVATIONS

- The coordinate system for the drone is as in Fig 3.
- The average velocity was chosen to be 20 centimetres per second to generate the trajectory.
- If the points along the quintic spline we generated from our controller, are beyond the boundary ranges; they are clipped to lie within them.
- From our usage of DJITelloPy package's ***go_xyz_speed*** to command our drone to move along the time bounded trajectories. We noticed that:
  - The command takes in arguments as relative positions with respect to current state.
  - The absolute changes in the X, Y and Z positions cannot exceed 20 centimetres simultaneously, and the same was mentioned in the Tello SDK user guide
  - We expanded the bloat dimensions to the size of the robot to have a safer and better trajectory calculation.
- The last waypoint was calculated to be close to the goal, however not at it. Adding the goal node as the last waypoint assisted the drone to execute landing close to the desired goal location.
- Continuously using the drone led to odometry succumbing to skew in sensing.

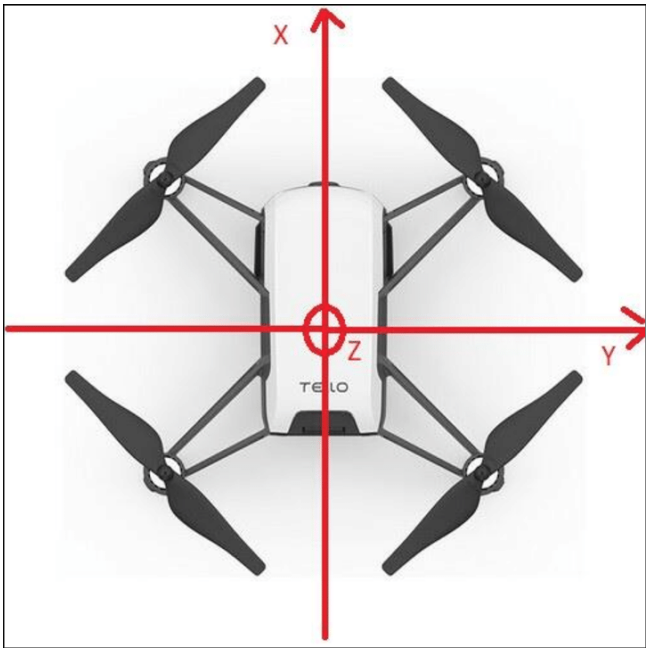

Fig. 3. The co-ordinate system for DJI Tello

V. REFERENCES

1 Principles of Robot Motion: Theory, Algorithms, and Implementations" by Howie Choset, Kevin M. Lynch, et al.
2 https://docs.px4.io/main/en/flight_stack/controller_diagrams.html
3 https://theclassytim.medium.com/robotic-path-planning-rrt-and-rrt-212319121378
4 https://github.com/damiafuentes/DJITelloPy/tree/master/djitellopy
5 https://dl-cdn.ryzerobotics.com/downloads/Tello/Tello SDK 2.0 User Guide.pdf