Project Report: Quadrotor Planning and Control

Sahachar Reddy

I. Introduction and System Overview

The objective of the in-person labs was to create trajectory generation and controller code that would allow a quadrotor robot to navigate safely from an initial point to an end point in an environment with obstacles. In the first session, the controller code was demonstrated, showing that the robot was capable of controlled flight following a basic trajectory. In second session, the trajectory generation code was demonstrated, showing that the code was capable of generating a collision-free path through an environment with obstacles and that the robot could follow it from start to finish without colliding or crashing.

The hardware used was: (1) Crazyflie 2.0 quadrotor drone, (2) Vicon system, and (3) lab computers. Code was written in Python 3 and ROS was used to interface with the robot. Position and orientation sensing was performed by the Vicon system, which captures motion of reflective markers mounted on board the Crazyflie, through the use of cameras mounted around the lab. Computation was performed on the lab computers, and control was performed by the robot. The Vicon system sends robot position and orientation data to the lab computers. The lab computers then combine this data with the generated trajectory to calculate the controller inputs, which are then sent from the lab computers to the robot to run the electric motors powering the propellers to maintain flight. At the start of each experiment, the lab computers are used to drive the quadrotors to a hover position. At the end, they are used to reduce the motor speeds to land the robot.

II. Controller

The tracking controller implemented was a geometric non-linear controller. The intuition behind this controller is that, since the robot generally applies force along the vertical axis (in body frame) rather than directly along the direction of the desired acceleration due to being underactuated, the robot should always be trying to point that axis in that direction. In order to orient the robot in the desired attitude, it is necessary to vary the thrust produced by the forward and rear rotors, to control pitch, the left and right rotors, to control roll, and the clockwise and counterclockwise rotors to control yaw.

To find the direction and magnitude of the desired acceleration, a PD controller was used that compared the position and velocity of the robot relative to the generated trajectory. We can then obtain a thrust input by projecting the desired force along the robot's vertical axis. We can also obtain a moment input by calculating the rotation error between the direction of the desired force and the robot's current attitude.

Taking the sum of the external forces and moments of the system, we obtain the following relations. For Eqn. 1, quantitatively, the force applied by the four rotors is defined as the following vector, where k_f and ω_i are the thrust coefficient and the angular velocity of the i-th rotor, respectively:

$$\begin{bmatrix} F_1 \\ F_2 \\ F_3 \\ F_4 \end{bmatrix} = \begin{bmatrix} k_f\omega_1^2 \\ k_f\omega_2^2 \\ k_f\omega_3^2 \\ k_f\omega_4^2 \end{bmatrix} \tag{1}$$

By combining the equations for net forces and moments with Newton-Euler equations for the system, we obtain the following set of equations. We define L as the arm length. $L = r_i$. Denoting A as the body fixed frame and B as the inertial frame, for sum of the forces:

$${}^A\omega^B = p\mathbf{b_1} + q\mathbf{b_2} + r\mathbf{b_3} \tag{2}$$

$$m\ddot{r} = \begin{bmatrix} 0 \\ 0 \\ -mg \end{bmatrix} + R \begin{bmatrix} 0 \\ 0 \\ F_1 + F_2 + F_3 + F_4 \end{bmatrix} \tag{3}$$

While for the sum of the moments:

$$I \begin{bmatrix} \dot{p} \\ \dot{q} \\ \dot{r} \end{bmatrix} = \begin{bmatrix} L(F_2 - F_4) \\ L(F_3 - F_1) \\ M_1 - M_2 + M_3 - M_4 \end{bmatrix} - \begin{bmatrix} p \\ q \\ r \end{bmatrix} \times I \begin{bmatrix} p \\ q \\ r \end{bmatrix} \begin{bmatrix} 0 \\ 0 \\ F_1 + F_2 + F_3 + F_4 \end{bmatrix} \tag{4}$$

It is possible then to define the thrust input as the vector:

$$u_1 = \begin{bmatrix} 1 & 1 & 1 & 1 \end{bmatrix} \begin{bmatrix} F_1 & F_2 & F_3 & F_4 \end{bmatrix}^T \tag{5}$$

Intuitively, this commands the magnitude of the desired acceleration. These thrusts are known in the body fixed frame. Similarly, it is possible to define the moment input as the vector, given that L is arm length and γ is drag-to-thrust ratio:

$$u_2 = \begin{bmatrix} 0 & L & 0 & -L \\ L & 0 & -L & 0 \\ \gamma & -\gamma & \gamma & -\gamma \end{bmatrix} \begin{bmatrix} F_1 \\ F_2 \\ F_3 \\ F_4 \end{bmatrix} \tag{6}$$

This commands the direction of the desired acceleration. These two equations, given our thrust and moment input, allows us to calculate the required force and thus angular velocity of each rotor.

Treating u_1 as inputs for the position controller (outer loop) and u_2 as inputs for the attitude controller (inner loop), the inner loop can be tuned first to track desired roll angle, followed by the tuning the outer loop.

The proportional gains act like a spring response, introducing oscillation, while the derivative gains act as resistance or adds stiffness to the system. A high proportional term makes the system more likely to overshoot; while a low value gives a soft response. A high derivative gain causes the system to be over damped, making the system converge slowly. For a stable (as

opposed to marginal stable or unstable) system, all gains must be positive.

In tuning the inner loop (u_2), the robot was initialized to a small deviation in orientation. K_r gains were tuned such that there was fast convergence to the desired orientation with slight overshoot. K_w was then tuned to dampen this overshoot. The same methodology was applied to tuning the controls for u_1, keeping in mind that to hover, the sum of the net forces equal to zero, and so known physical parameter mass, can be used to estimate suitable gains. The optimal gains for the controller were:

$$K_p = \begin{bmatrix} 13 & 0 & 0 \\ 0 & 13 & 0 \\ 0 & 0 & 13 \end{bmatrix} [s^{-2}]; K_d = \begin{bmatrix} 7.75 & 0 & 0 \\ 0 & 7.75 & 0 \\ 0 & 0 & 7 \end{bmatrix} [s^{-1}]$$

$$K_R = \begin{bmatrix} 4000 & 0 & 0 \\ 0 & 4000 & 0 \\ 0 & 0 & 3000 \end{bmatrix} \left[\frac{s}{kg \cdot m^2 \cdot rad}\right]$$

$$K_\omega = \begin{bmatrix} 170 & 0 & 0 \\ 0 & 170 & 0 \\ 0 & 0 & 325 \end{bmatrix} \left[\frac{s}{kg \cdot m^2 \cdot rad}\right]$$

The attitude controller is not actually being used directly because whatever thrust it commands is purely in order to generate changes in attitude and will result in zero linear force commanded. In other words, the attitude controller simply orients the robot by commanding moments so that the position controller, which actually commands the thrust such that the robot will change position, can apply the thrust in the appropriate direction.

In simulation, tight time constraints had to be met, so gains were tuned for high velocity flight and aggressive maneuvers. For lab experiment 1, the high velocities and high gains were reduced by 30% for flight during experiments to be deemed safe for the Crazyflie platform. This would prevent instabilities and in turn, crashes. For lab experiment 2, only the velocity was reduced. Gains were not altered since the cluttered maps required a tightly tuned controller. To add, while increasing z-direction position gains in simulation allowed steady-state errors to be reduced, this was not the case during real experiments. We believe this is because the motors on the physical robot may not be able to provide the thrust required to counter the additional weight on the real robot (perhaps from hardware), which is unaccounted for in simulation. Finally, when battery levels on the Crazyflie were low, the quadrotor failed to achieve desired positions and orientations, simply because insufficient power was being sent to motors. These inconsistencies do not exist in simulation.

Figure 1 shows a step response of the system with the following characteristics: rise time $\approx 0.75s$; steady state error $\approx 0.25m$, damping ratio ≈ 0.97 (meaning critically damped); and settling time $\approx 3.5s$. Figure 2 shows the system response to predefined waypoints. The robot was able to easily achieve the desired y-axis and x-axis values, while z-axis values were not met. Despite increasing z-position proportional gains, the steady state error remained similar. There was simply more overshoot when proportional gains were increased. We attribute these discrepancies with the added weight, as explained above.

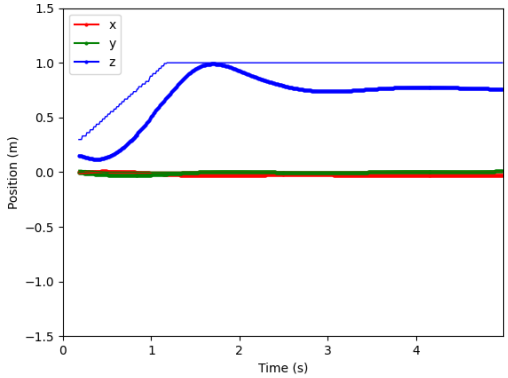

Fig. 1. Z-direction Step Response

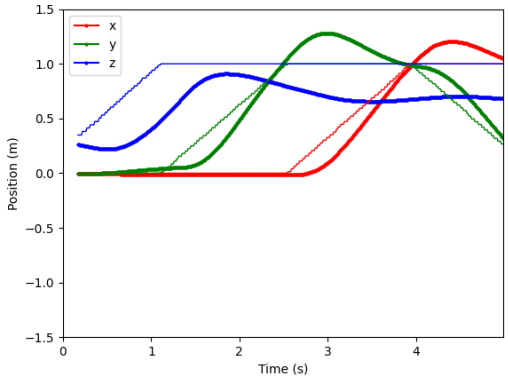

Fig. 2. Waypoint Response

III. Trajectory Generator

A *minimum snap* (4th derivative) trajectory generator was implemented using the following methods:

1) A* graph search algorithm
2) Co-linear and proximity pruning
3) Progressive segment duration scaling
4) Smooth trajectory generation using a constraint matrix

A*, a variant of Dijkstra's Algorithm, was implemented to identify an optimal path from start to goal nodes using a 26-voxel neighborhood for any provided world map with rectilinear obstacles. Euclidean distance was used as the heuristic for node selection priority. The output of this algorithm is an array of points representing a dense path connecting the metric centers of all sequential voxel neighbors which comprise the optimal path.

To avoid unnecessary computational complexity and address redundancy within this dense path, two pruning techniques (co-linear pruning and proximity pruning) were used to generate a sparse path of waypoints. The co-linear pruning function simply removes points which lay along the line between their neighbors. The subsequent proximity pruning function then reduces close groupings (such as those defining corners in the path) to a single point.

When using this sparse path to generate a trajectory, however, preserving more intermediate points in co-linear regions was ultimately required for two reasons. First, the co-linear pruning

created very long segments, yielding high segment variance which proved challenging for the progressive duration scaling step - giving undesirable excursions between waypoints. Second, these longer segments were prone to collision because this method did not include inequality or corridor constraints which could prevent collisions along all points in the smooth trajectory. A manually-tuned parameter was thus included which limits the number of successive co-linear points which can be pruned.

Initially, segment durations were assigned by dividing each segment length by a desired average linear velocity for the entire path. However, this produced an inefficient trajectory with large excursions and/or tight cornering requirements. The approach taken in this step progressively scales durations such that short segments required more time proportional to their endpoint distance as compared to longer segments. This progressive scaling was achieved with Eqn. (7) below:

$$T_i' = T_i \sqrt{\frac{\alpha}{T_i}} \tag{7}$$

Where T_i is the naive duration based on average linear velocity of the ith segment, T_i' is the scaled duration of the ith segment, and α is a manually-tuned parameter whose value (1.2s worked well) is the duration above and below which segment durations are scaled lower and higher, respectively.

The final process generates a minimum snap smooth 'optimal' function for each segment beginning at its start time 0 and final time T, given by Eqn. (8).

$$x^\star(t) = argmin_{x(t)} \int_0^T \mathcal{L}(x^{(4)}, \dddot{x}, \ddot{x}, \dot{x}, x, t)dt \tag{8}$$

Which satisfies the condition specified by the Euler Lagrange Equation below.

$$\frac{\partial \mathcal{L}}{\partial x} - \frac{d}{dt}\left(\frac{\partial \mathcal{L}}{\partial \dot{x}}\right) + \frac{d^2}{dt^2}\left(\frac{\partial \mathcal{L}}{\partial \ddot{x}}\right) + ... + (-1)^n \frac{d^n}{dt^n}\left(\frac{\partial \mathcal{L}}{\partial x^{(n)}}\right) = 0 \tag{9}$$

With a minimum snap trajectory, n = 4 and the minimization function becomes $\mathcal{L} = (x^{(4)})^2$. Thus Eqn. (10) reduces to the following.

$$\frac{d^4}{dt^4}\left(\frac{\partial \mathcal{L}}{\partial x^{(4)}}\right) = \frac{d^4}{dt^4}(2x^{(4)}) = 0,\ x^{(8)} = 0 \tag{10}$$

Therefore, to achieve minimum snap, our trajectory takes the form of a 7th-order polynomial given by Eqn. (11).

$$x(t) = c_7t^7 + c_6t^6 + c_5t^5 + c_4t^4 + c_3t^3 + c_2t^2 + c_1t + c_0 \tag{11}$$

A matrix was then built to accommodate the $8m$ constraints necessary to solve for the coefficients of each polynomial segment, where m is the number of segments in the trajectory. A certain set of constraints are required to compute this trajectory for the physical system. For instance, up to the $(n-1)^{th}$ derivative must be continuous along the entirety, yielding $4(m-1)$ constraints (1 positional, 3 derivative for every intermediate waypoint). Additionally, $(m+1)$ positional constraints define all waypoint locations, and the start and goal waypoints are assigned 6 total derivative boundary constraints to accomodate the (n-1)th derivative. This leaves $8m - [4(m-1) + (m+1) + 6] = 3(m-1)$ constraints with which maximum smoothness criterion were imposed. This equates to imposing continuity constraints on the higher-order (4th, 5th, and 6th) derivatives.

The constraints were then assembled as a Matrix system of equations of the form expressed in Eqn. (12), where A is the constraint matrix containing polynomial expressions of t and its derivatives, C is the coefficient matrix to be solved for, and P is the matrix of corresponding constraint values.

$$A_{8\text{m}\times 8\text{m}} C_{8\text{m}\times 3} = P_{8\text{m}\times 3} \tag{12}$$

Note, C and P have dimensions 8m×3 because each point consists of 3-dimensional coordinates. The spreadsheet linked here demonstrates how these constraints were assembled in A for an example 3-waypoint path.

Ultimately, this trajectory generator proved quite feasible, and with some fine-tuning of the map resolution and acceptable margin (0.2m and 0.175m, respectively) it was able to find a collision-free trajectory through each of 3 experimental maze maps. Given the aggressive attitude gain re-tuning from part one (controller step testing) to part 2 (maze trials), our trajectory is feasible even at speeds higher than expected - up to an average linear velocity input of 7.5m/s. Simulation flight tracked the planned trajectory with negligible error at an average 1.5m/s. And from Fig. (4) and it is evident there is little positional tracking error during the real flight. The slight z offset and phase shift in the real flight results is likely due to the tracking system and un-modeled physical aspects of the system, including motor delays. The deviations from expected velocity in Fig. (5) likely result from additional controller compensation for some of these non-idealities.

IV. Maze Flight Experiments

Three real world maze maps (Map 1, Map 2, and Map 3) were used to test our system i.e. the trajectory planner and controller on the Crazyflie 2.0. On the first run, our system found the optimal path, tracked it using the minimum snap trajectory generator combined with an aggressive geometric controller to reach the goal position. The Vicon system was used to obtain the real flight observations which upon careful processing show that the real measurements conform to the simulation with high confidence. The results are comparatively quantified in Table I.

The way-points, planned trajectory and actual flight path for the three maze flight experiments are shown in Figure 3. From Figure 3(c), it can be observed that the trajectory is planned through the window and the actual flight path tracks it closely through the window. This through window performance explains the efficacy of the planner, trajectory generator and the controller in that it allows for the Crazyflie to move through such small places without collision.

The positional and velocity tracking behaviour of the real flight can be observed from Figures 4, and 5 wherein the real flight position seems to perfectly track the simulated behaviour. However, a slightly larger tracking error is observed in the case of altitude (Z), this can be attributed to the lack of integral term in the controller causing a steady state error in Z direction and the aggressive gains for attitude controller adds more to the cause in order to cope with the real world dynamics.

Exp	Absolute mean tracking error	x	y	z
Map1	Position (m)	0.0427	0.0589	0.1261
Map1	Velocity (m/s)	0.0030	0.0115	0.0332
Map2	Position (m)	0.0073	0.0122	0.1192
Map2	Velocity (m/s)	0.0007	0.0077	0.0339
Map3	Position (m)	0.0074	0.0430	0.1015
Map3	Velocity (m/s)	0.0056	0.0054	0.0015

TABLE I

TRACKING ERRORS IN POSITION AND VELOCITY FOR ALL MAPS

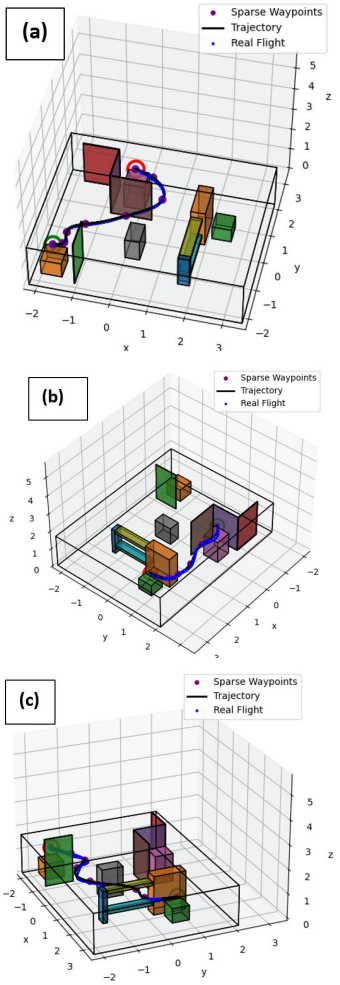

Fig. 3. 3D plots showing the world obstacles, waypoints, planned trajectory, and actual flight path. for (a) Map-1 (b) Map-2 (c) Map-3

From Table I we can safely conclude that the tracking errors are very small, although relatively larger for the Z direction and the developed system has high enough aggressive gains to be tested for more challenging environments. The already aggressive gains allow for the controller to track the minimum snap trajectory easily at high speeds (tested up to 7.5 m/s). Further testing with increasing gains might converge to a better performance. However, considering minimal tracking errors and steady state errors in Z direction the possibility of overshoot and instability are higher with making the gains more aggressive. So, testing for better convergence can safely be practised with small increments in the present gains. Margins can be set to the sum of mean tracking errors and current margins to determine safe trajectories, essentially including an additional buffer of safety. Alternatively, if possible, flight speeds can be reduced for path segments with larger tracking errors.

One thing we could have attempted in order to improve the reliability of our system was to use our extra degrees of freedom in the trajectory generation algorithm in order to perform some sort of cost optimization or satisfy corridor constraint, rather than satisfy the maximally smooth constraint. This could have generated trajectories that deviates less from the straight line path between waypoints, either by disincentivizing such deviations (in the former) or by outright forbidding it (in the latter). This is particularly useful with waypoints relatively close together that could not be pruned, as these tend to create very sharp, tight turns. Ensuring that our trajectory has low deviation would lower the probability of a crash or collision; gentler turns would also allow for faster speeds without losing control. We could also have spent more time tuning the controller gains, which would improve speed and reliability, though this is a process with diminishing returns.

If we had more time for in lab experiments, it would be great to try flying through some narrow section, like a tunnel of hula hoops. It would be fun to see the Crazyflie perform a flip in the air too!

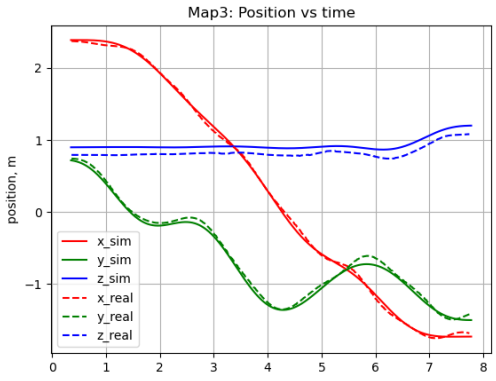

Fig. 4. Actual vs Simulated Flight Position vs Time for Map 3

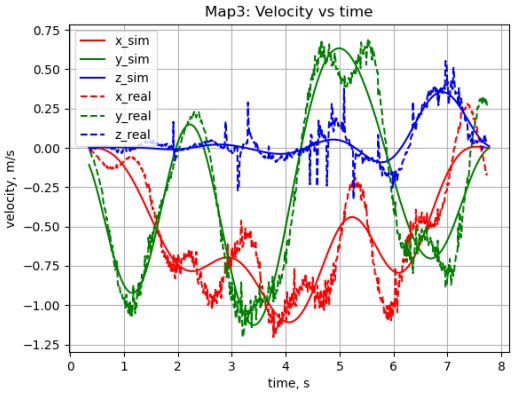

Fig. 5. Actual vs Simulated Flight Velocity vs Time for Map 3