Project Report: Quadrotor Planning and Control

Team 19: Jinyuan Guo, Yilin Li, Thomas Zhang

I. Introduction and System Overview

The overarching goal of the two in-person lab components of this project is to successfully deploy our code for various aspects of quadrotor path planning and control, which were designed in simulation, to a real quadrotor. Specifically, the first lab session was dedicated to getting familiar with the VICON experimental set-up as well as simple tests to tune and validate the efficacy of our controllers. The second lab session was dedicated to deploying our trajectory generation module in tandem with the controller and judging the efficacy of the path execution. In general, it was demonstrated that the code we wrote for a simulated quadrotor could in fact be successfully deployed onto a real-life quadrotor, as long as certain (simple) tweaks to account for the simulation-to-reality gap are made.

To summarize the experimental set-up: the backbone of the entire operation is the VICON motion capture system. From our understanding, the VICON set-up used in this set of experiments is a passive-optical motion capture, where retroreflective markers are placed on an object of interest (in this case the quadrotor), and an array of cameras situated around the experiment area track said markers using infrared light. For a sufficiently high frame rate and aptly placed markers (usually at least two on distinct rotors and one in the center of the quadrotor), this allows for the precise tracking of the quadrotor's physical pose within the experiment space, i.e. it's position and orientation. In order for a computer to communicate to and process the data from the VICON camera set-up, a literal black box (the "sync box") is used. The exact technical details are not provided, but the overall purpose of the sync box is to both translating commands sent by the computer as well as processing the raw camera data into a computer-parsable format. With the configuration and trajectory data in hand, further derivatives can be calculated on the computer, such as through smoothing techniques, which gives us access to approximate higher-order derivatives of the trajectory if necessary. However, in the case of the Crazyflie 2.0, the angular velocities and accelerations are provided by an onboard inertial measurement unit, which are separately communicated to the computer via a 2.4GHz CrazyRadio (cf. Project 1.1 spec).

With online access to trajectory and derivative data sent by the sync box and onboard measurement units, we have sufficient information to close the loop with our control algorithm. Our controller is a (geometric non-linear) Proportional-Derivative controller, which is a feedback control system. In other words, the controller takes in as input the current measured velocity and position of the quadrotor center of mass and compares it to the expected position, velocity, and acceleration as computed by our path planning/trajectory generation module. Using physical equations of motion, the desired acceleration as returned by the PD controller can be turned into desired thrust (thus motor speeds) for the quadrotor (details are in the next section). After computing the desired motor speeds on the computer, they are then communicated back to the quadrotor via the CrazyRadio, finally closing the feedback control system loop. To the best of our knowledge, the computer on which we executed our code is in charge of the path planning and control input computations, while the TA's computer is in charge of receiving the outputs of our algorithms and relaying them to the drone via the CrazyRadio, and vice versa: sending the VICON pose information and CrazyFlie IMU data to our computer.

II. Controller

Our position PD controller is as shown in Equation (1)

$$\ddot{\mathbf{r}}^{\text{des}} = \ddot{\mathbf{r}}_T - K_d(\dot{\mathbf{r}} - \dot{\mathbf{r}}_T) - K_p(\mathbf{r} - \mathbf{r}_T) \tag{1}$$

where K_d and K_p are diagonal matrices. The most recently tuned values we settled on are

$$K_p = \text{diag}(2.5, 2.5, 2.5)\text{s}^{-2}, \tag{2}$$

$$K_d = \text{diag}(1.25, 1.25, 1.25)\text{s}^{-1}, \tag{3}$$

where K_p impacts how fast the system responds to the desired state: the larger K_p, the shorter the response time but the larger possible overshoot. Larger K_d smooths out oscillations and decreases overshoot, but if it is tuned too high, it will take too long for the system to make adjustments for the desired state.

From Equation (1), we can get the total commanded force $\mathbf{F}^{\text{des}}$:

$$\mathbf{F}^{\text{des}} = m\ddot{\mathbf{r}}^{\text{des}} + \begin{bmatrix} 0 \\ 0 \\ mg \end{bmatrix} \tag{4}$$

while

$$u_1 = \sum_{i=1}^{4} F_i = \mathbf{b}_3^\top \mathbf{F}^{\text{des}}, \tag{5}$$

where $\mathbf{b}_3 = R[0,0,1]^\top$ is the z-axis of quadrotor in the world frame. We want $\mathbf{b}_3$ to align with $\mathbf{F}^{des}$, so the attitude controller is applied to align them together with the orientation error, where the error vector is defined as:

$$\mathbf{e}_R = \frac{1}{2}(R^{\text{des}\top} R + R^\top R^{\text{des}})^\vee \tag{6}$$

and the attitude controller input is calculated by:

$$\mathbf{u}_2 = I(-K_R \mathbf{e}_R - K_w \mathbf{e}_w). \tag{7}$$

The given desired state is $\ddot{\mathbf{r}}_T, \dot{\mathbf{r}}_T, \mathbf{r}_T$ and the orientation control is the premise to realize position control, so the attitude controller is not being used directly.

For the first lab section, all controller gains are lowered to $\frac{1}{4}$ of their original values, since they are tuned for the quadrotor to achieve some very aggressive trajectories with high speed in simulated test cases. The acceleration will be too large for aggressive dynamics, which will result in required motor speeds exceeding speed limits. For safety purposes and steady control of the quadrotor in reality, the trajectory speed is slowed down to 1m/s and all controller gains are decreased. This also reduces the chance that sim2real errors cause a catastrophic failure.

The results of step response of z direction and and a Big Cube trajectory in the first lab section are shown in Figure 1 and 2, respectively.

From Figure 1, it is can be briefly calculated that the system damping ratio is now:

$$\zeta = \frac{-\ln M_p}{\sqrt{\ln {M_p}^2 + \pi^2}} \approx 0.66 \tag{8}$$

where $M_p \approx 6.25\%$ is the overshoot. The rising time is about $T_s = 0.9s$ and steady state error is about $e \approx 0.05\text{m}$.

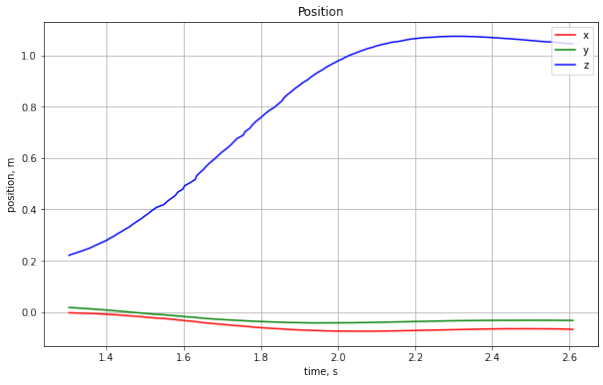


Fig. 1. Step Response of z Axis

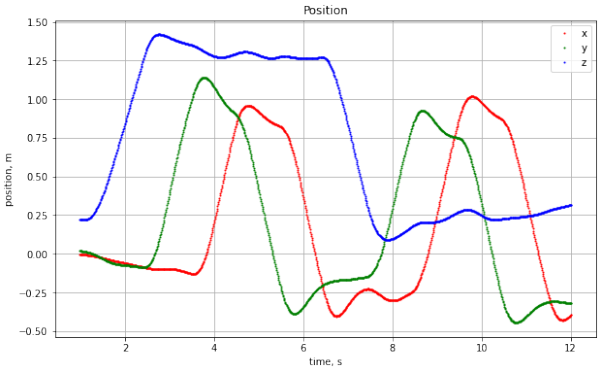


Fig. 2. Experimental Position of Big Cube Trajectory

III. Trajectory Generator

The trajectory generator consists of three phases: path searching, waypoints pruning, and trajectory planning.

For the first phase, path searching is based on the A* algorithm, with the Euclidean distance as the heuristic. The map resolution is set to be 0.1m^3 and the margin is set to be 0.28m which is slightly larger than the radius of the drone 0.25m to give the trajectory planning more freedom without collision but also ensure a path exists.

After that, we can acquire a set of dense waypoints $\mathcal{P}^{\text{dense}} = \{P_i^{\text{dense}}, i = 1, \ldots, N\}$ including the start and goal point for post processing. We illustrate some details of the waypoint pruning process

1) 1st-round delete:
   Choose the start point as P_1, then from the goal point iterate waypoints P_2 backwards until a non-collision takes place between the line segment $\overline{P_1 P_2}$ and the map obstacle. Then add all the points between P_1 and P_2 to delete waitlist $\mathcal{P}^{\text{del}}$. Set P_1 as P_2 and keep iterating until P_1 reaching the goal point. The remainder set of waypoints is $\mathcal{P}^{\text{sparse}}$.
2) 2nd-round delete:
   From the start point, iterate over $\mathcal{P}^{\text{sparse}}$, find the distance of $d(\overline{P_{i-1} P_i})$ and $d(\overline{P_i P_{i+1}})$. If either of them is less than 0.5m, then add point P_i to $\mathcal{P}^{\text{del}}$. Delete point P_i from $\mathcal{P}^{\text{sparse}}$.
3) Add-back:
   Iterate point P_i^{del} over $\mathcal{P}^{\text{del}}$, find the distance of P_i^{del} to the closest point P^s in $\mathcal{P}^{\text{sparse}}$. If $d(\overline{P_i^{\text{del}} P^s}) > 1.5\text{m}$, then add P_i^{del} back to $\mathcal{P}^{\text{sparse}}$, and remove it from $\mathcal{P}^{\text{del}}$.

A sparse waypoints set $\mathcal{P}^{\text{sparse}}$ is then acquired by applying the above algorithm. We are now ready for trajectory planning.

For trajectory planning, we used a minimum jerk trajectory for the second lab section. First, the trajectory speed is set as approximately 1m/s. Based on the distance between two neighbour sparse waypoints, the time

interval for each curve segment is allocated. Especially, to avoid very dynamic curve planning at the first and last curve and make the dynamics at the start and end more mild to facilitate control performance. The time for the initial segments and last segments is allocated to be $2\times$ and $3\times$ longer, respectively. Assume we have m segments, then there are 6 boundary conditions for initial and last points, $2(m-1)$ intermediate waypoints position constraints, $4(m-1)$ continuity constraints for intermediate waypoints. The trajectory function of time for each segment is a 5th order polynomial:

$$y(t) = t^5c_5 + t^4c_4 + t^3c_3 + t^2c_2 + tc_1 + c_0 \quad (9)$$

The end point boundary constraints are:

$$p_1(0) = p_0 \quad (10)$$
$$p_m(t_m) = p_m \quad (11)$$
$$\dot{p}_1(0) = 0 \quad (12)$$
$$\dot{p}_m(t_m) = 0 \quad (13)$$
$$\ddot{p}_1(0) = 0 \quad (14)$$
$$\ddot{p}_m(t_m) = 0 \quad (15)$$

For segment i, the position constraint is:

$$p_i(t_i) = p_i \quad (16)$$
$$p_{i+1}(0) = p_i \quad (17)$$

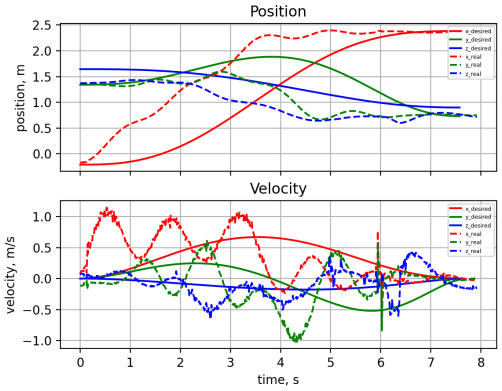
And the continuity constraints for segment i are:

$$\dot{p}_i(t_i) = \dot{p}_{i+1}(0) \quad (18)$$
$$\ddot{p}_i(t_i) = \ddot{p}_{i+1}(0) \quad (19)$$
$$p_i^{(3)}(t_i) = p_{i+1}^{(3)}(0) \quad (20)$$
$$p_i^{(4)}(t_i) = p_{i+1}^{(4)}(0) \quad (21)$$

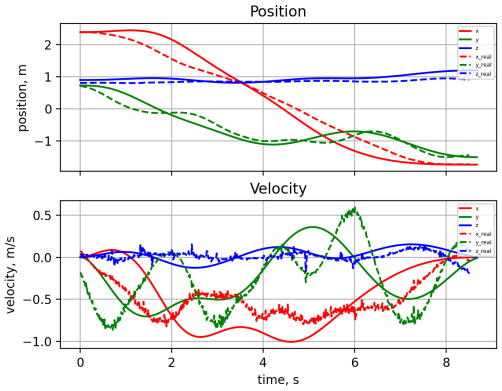
Some examples of the resulting generated path as well as the simulated and real quadrotor's tracking performance are depicted in Figure 3 and 4. From Figure 4, it is apparent that all three planned trajectories are smooth and mild without any collision and sharp turns even though at the beginning and the goal point.

IV. Maze Flight Experiments

We now report the performance of our path planning and controller code on the various maze runs. In Figure 4, we show the computed waypoints, simulated quadrotor trajectory, and actual flight path with respect to the maze world obstacles for all three maze segments. We see that despite the computed target trajectory and corresponding simulated flight path being extremely close (such that on the plots they are completely overlaid), the actual flight path taken noticeably differs from the simulated path. The discrepancy is made even clearer in Figure 3, where we plot the position and velocity over time of the desired trajectory versus the actual trajectory.



(a) Maze segment 2



(b) Maze segment 3

Fig. 3. Position and velocity over time of simulated trajectory vs actual trajectory

Since our simulated trajectory tracks the desired trajectory perfectly, this implies the discrepancy arises from various aspects of the sim2real gap.

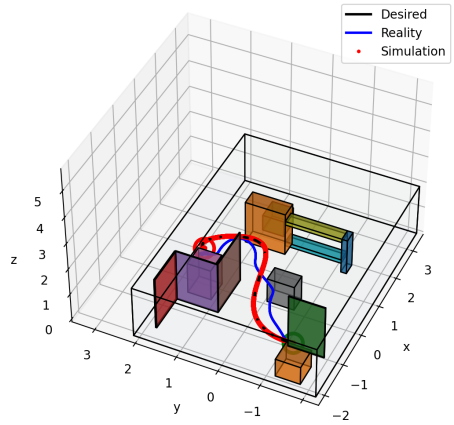
Over the course of testing on maze runs, we made the multiple adjustments to various parameters to help counter the perceived sim2real gap. Before starting any tests, we reduced all gains in our controller by $\frac{3}{4}$, and desired velocity to approximately 1m/s. However, after some initial runs on the maze (such as that depicted in Figure 3a, we observed that there was a noticeable constant offset in the z-axis, after which we manually increased the mass parameter fed into our controller by 5%, which seemed to significantly help with tracking error in subsequent maze runs, such as that depicted in Figure 3b.

Focusing on Figure 3a, we see that over the course of the run, there was significant deviation in both the overall position (exceeding 0.5m in some segments), as well as the angular velocity, which seemed to oscillate at a much higher frequency than simulated. We also note that at 6 seconds, despite a very smooth simulated

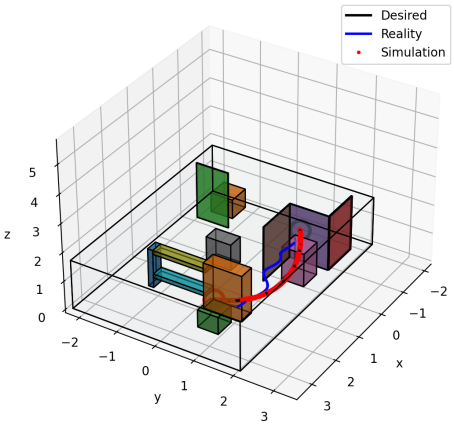
trajectory at that point, it seems the actual angular velocity encountered a very sharp discontinuity. Indeed, this is reflected in our logs, which indicates that the commanded thrusts exceeded the limit during this segment. This was a perplexing issue during our lab session, since a minimum jerk trajectory (or a minimum snap trajectory, which we also tried) should in principle prevent such discontinuities from occurring. We also tuned our position proportional and derivative gains to try to address this issue but to no avail–when we increased our proportional gains to try to improve tracking, the discontinuous behavior would be exacerbated and the quadrotor crashed. When we increased our derivative gains to address the oscillations in angular velocity, the quadrotor also crashed.

It is highly unlikely we could have safely executed more aggressive maneuvers without painstakingly experimenting with gain parameters, as we encountered some tracking issues with the very smooth trajectories generated by the minimum jerk approach. To reduce the tracking error and improve the reliability of our controller, some additional gain tuning would be beneficial–we seemed to have seen marked improvement at the end of our session on maze segment 3 (Figure 3b). Once we can verify that our tuned controller tracks well enough, we can certainly increase the desired velocity by some margin–we kept the velocity conservative during our lab sessions as instructed by the TAs and also to prevent damage to the quadrotor during the gain-tuning process.

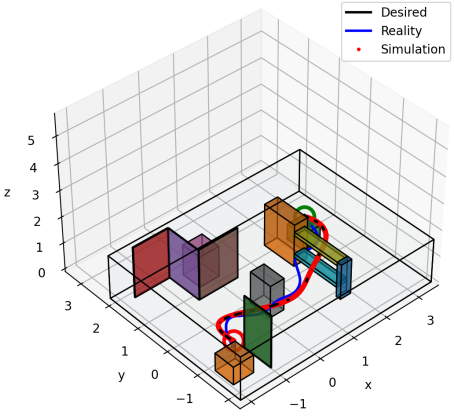
If we were given another lab session, some things we could try include: precisely measuring the Crazyflie's physical parameters (such as mass, wingspan) and adjust the simulation parameters accordingly, as this seemed to be the likely source of sim2real error during our 2nd lab session. After doing so, we can re-run some step responses to finetune gains to account for any further sim2real error. If we can certify that this controller tracks well on, say, the maze runs, it would be interesting to see how our path planner and controller perform on aggressive maneuvers, such as zigzags or even loop-the-loops, which are purportedly possible with the geometric nonlinear controller paradigm (cf. Project 1.1 spec). During our first lab session, we tried an (unobstructed) zigzag maneuver and saw promising behavior with minimal tuning.



(a) Maze segment 1; simulated path vs actual path taken



(b) Maze segment 2; simulated path vs actual path taken



(c) Maze segment 3; simulated path vs actual path taken

Fig. 4. 3D plot of world obstacles, computed waypoints, simulated trajectory, and actual flight path taken for each of the three maze segments