Project 1.4 Report: Quadrotor Planning and Control

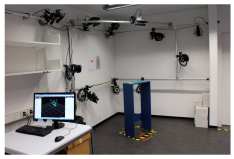
Team 15: Tianyu Feng, Dhyey Shah, Paul Young, and Jinyuan Zhang

I. Introduction and System Overview

The overall objective of this in-person lab session is to test our previous project work on real hardware, as opposed to the past work that has been developed in simulation.

The hardware used in the lab includes: 1 Crazyflie 2.0, 2 lab computers running ROS (1 for the TA and 1 for the students), and a Vicon motion capture system for position and orientation sensing.

(a) Crazyflie 2.0



(b) Example Vicon setup

Fig. 1: Main Lab Hardware used

The system communication involved the student computer running control, graph search, and trajectory generation code, sending desired dynamics to the TA lab computer, which then relayed motor speed commands to the Crazyflie via ROS. The VICON system tracked the quadrotor's position and orientation, sending data back through the TA lab computer to the student computer, closing the feedback loop.

The first lab session focused on tuning the $SE(3)$ position and orientation PD controller on the hardware. Step input tests refined the gains, verified through waypoint navigation. The second session tested graph search and trajectory generation in a real-world setting, with the Crazyflie navigating a physical maze, assessing the feasibility and robustness of our code despite misalignments between the virtual and physical environments.

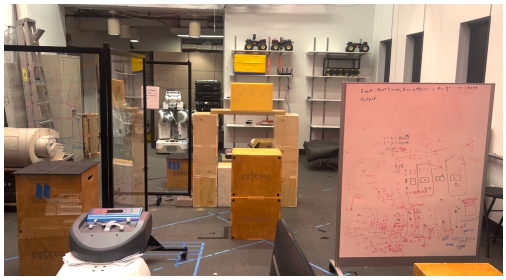


Fig. 2: Image of the lab maze setup

II. Controller

TABLE I: Quadrotor Gains Parameter

Parameter	Value
diag(K_p) [$1/s^2$]	(3.75, 3.75, 30)
diag(K_d) [$1/s$]	(3.25, 3.25, 8.25)
diag(K_R) [Nm/rad]	(1500, 1500, 80)
diag(K_ω) [Nms/rad]	(100, 100, 20)

A. Position Controller and Attitude Controller

We implement a geometric nonlinear tracking controller, also referred to as an $SE(3)$ controller. In contrast to traditional methods that separate position and orientation into two control layers, the $SE(3)$ controller is to align the quadrotor's thrust with the required acceleration to track the reference trajectory in $\mathbb{R}^3$, while simultaneously managing the orientation on $SO(3)$. Suppose we have a target (or reference) trajectory $\mathbf{r}_T(t)$, with velocity $\dot{\mathbf{r}}_T(t)$ and acceleration $\ddot{\mathbf{r}}_T(t)$ gained from trajectory planar, then we write down our PD controller, the desired acceleration $\ddot{\mathbf{r}}^{\text{des}}(t)$ needed to be commanded is calculated by

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

Here, $\mathbf{r}$ and $\dot{\mathbf{r}}$ denote the current position and velocity, respectively. K_p and K_d are diagonal, positive-definite gain matrices. The intuition is that the desired acceleration is obtained from the feed forward term $\ddot{\mathbf{r}}_T$ plus corrections based on both velocity and position errors to ensure the system converges to the target trajectory. Tuning the control gain matrices needed to be very precisely. The final gain values are shown in the Table I. Proportional gain K_p directly feeds back the error signal. A suitable value of K_p accelerates convergence to the target. In general, increasing K_p reduces rise time and steady state error but increases overshoot. Meanwhile, the derivative gain K_d provides feedback on the rate of change of the error. If the system's error grows rapidly, this derivative term acts like a damping force, suppressing overshoot and oscillations. To be more precise, the following equations illustrate how the PD gains affect the second order system's performance. Note ζ and ω_n are the damping ratio and natural frequency, respectively.

$$\ddot{e} + K_d\,\dot{e} + K_p\,e = 0, \iff \ddot{e} + 2\zeta\,\omega_n\,\dot{e} + \omega_n^2\,e = 0. \quad (2)$$

The PD controller uses real-time error to compute the desired acceleration $\ddot{\mathbf{r}}^{\text{des}}(t)$ in the inertial frame. We then apply Newton's second law to convert this acceleration into the body-frame control input u_1, which represents the total thrust. The relevant equation is:

$$u_1 = \mathbf{b}_3^T \Big(m\, \ddot{\mathbf{r}}^{\text{des}}(t) + \begin{bmatrix} 0 \\ 0 \\ m\, g \end{bmatrix} \Big). \tag{3}$$

Here, $\mathbf{b}_3 = R\,[0\ 0\ 1]^T$ denotes the quadrotor's z-axis expressed in the inertial frame, where R is the rotation matrix describing the orientation with respect to the inertial frame. Denote the part in bracket as $\mathbf{F}^{des}$. To compute the control input that generates the moments that control roll, pitch, and yaw, we define $\mathbf{u}_2$ as follows.

$$\mathbf{b}_3^{\text{des}} = \frac{\mathbf{F}^{\text{des}}}{\|\mathbf{F}^{\text{des}}\|}, \quad \mathbf{a}_\psi = \begin{bmatrix} \cos\psi_T \\ \sin\psi_T \\ 0 \end{bmatrix}, \tag{4}$$

$$\mathbf{b}_1^{\text{des}} = \frac{\mathbf{b}_3^{\text{des}} \times \mathbf{a}_\psi}{\|\mathbf{b}_3^{\text{des}} \times \mathbf{a}_\psi\|}, \tag{5}$$

$$R^{\text{des}} = \begin{bmatrix} \mathbf{b}_2^{\text{des}} \times \mathbf{b}_3^{des} & \mathbf{b}_2^{\text{des}} & \mathbf{b}_3^{\text{des}} \end{bmatrix}, \tag{6}$$

$$\mathbf{e}_R = \tfrac{1}{2} \big(R^{\text{des}T}\, R \ - \ R^T\, R^{\text{des}} \big)^{\text{v}}, \tag{7}$$

$$\mathbf{e}_\omega = \omega - \omega^{des} \tag{8}$$

$$\mathbf{u}_2 = I\, \big(-K_R\, \mathbf{e}_R \ - \ K_\omega\, \mathbf{e}_\omega \big). \tag{9}$$

The main idea is straightforward, we construct the reference orientation from the commanded force $\mathbf{F}^{\text{des}}$and heading vector $\mathbf{a}_\psi$ so that the quadrotor aligns with the thrust direction and matches the target yaw ψ_T. Using Equations (4)and (5), we then obtain the desired orientation R^{des}. Next, we define the orientation error $\mathbf{e}_R$ (7) and the angular velocity error $\mathbf{e}_\omega$ (8), and use the inertia matrix I to compute the commanded control input $\mathbf{u}_2$ (9). In our implementation, we represent orientation using quaternions $[i,\ j,\ k,\ w]$; however, when operating in $SE(3)$, we convert these quaternions to rotation matrices. After obtaining u_1 and $\mathbf{u}_2$, we calculate the required force for each of the four propellers based on the quadrotor's configuration, and then use the thrust coefficient k_{thrust} to compute the commanded angular velocities.

B. Experiments and Analysis

To evaluate our controller's performance, we conducted a step response test along the z-axis, where the reference height is set to $1\,\text{m}$. Figure 3 shows the quadrotor's position over time.

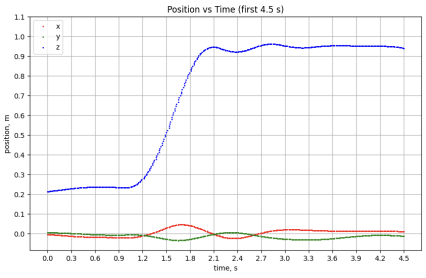


Fig. 3: Z-axis step response with a target position of $z = 1\,\text{m}$ and $x = y = 0$.

1) Steady-State Error: The commanded height is $1.0\,\text{m}$, but the quadrotor settles at approximately $0.95\,\text{m}$, resulting in a steady-state error of about $0.05\,\text{m}$. This discrepancy may be caused by ground-effect disturbance(or other aerodynamic drag reasons) . Introducing an integral term could help reduce this residual steady-state error.

2) Damping Ratio: Our system exhibits a slight overshoot, with oscillations of approximately ± 0.3 observed. This indicates system is slightly underdamped, with an estimated damping ratio of about 0.65. Additionally, we observed small deviations in the x and y positions, which can be attributed to minor variations in the quadrotor's orientation causing the thrust vector to deviate slightly from the vertical direction.

3) Rise Time: The rise time is defined as the duration required for the response to increase from 10% to 90% of its steady-state value. In our experiment, the response rises from 10% (0.275 m) to 90% (0.875 m) over the interval from 1.2 s to 1.9 s, yielding a rise time of approximately 0.7 s, which is fine.

4) Settling Time: The quadrotor initiates lift at approximately 0.2 m and starts step response at 1s, then settles by about 3.2 s, which yields a settling time of roughly 2.2 s.

Overall, the controller performance is acceptable based on the metrics discussed above. The Sim2Reall gap can be think to several factors. First, our dynamic model does not fully capture the physical behavior of the real system; for instance, aerodynamic effects are not modeled. Additionally, asymmetries in the quadrotor (which affect the inertia matrix I) and noise in the Vicon system can cause small state estimation errors. The Sim2Real gap can be reduced through iterative tuning of the PD gain matrices. In simulation, larger PD gains can be used to achieve a faster response without causing instability. However, in the actual system, due to these uncertainties, using excessively high PD gains may lead to instability. Consequently, we reduced both K_p and K_d slightly through experimental tuning. Also, we set a limit to our speed for safety.

III. Trajectory Generator

i) The graph_search function in graph_search.py performs a graph-based search to find a collision-free path from start to goal in a 3D environment, represented by the world object. The environment was discretized into a 3D occupancy grid using the OccupancyMap class, mapping obstacles into voxels based on resolution and safety margin. Each voxel is a node, with 26 neighbors in a 3x3x3 grid (excluding itself). The cost to move to a neighbor was taken as the Euclidean distance between voxel centers, scaled by resolution. A* searches the grid, using a priority queue prioritized by f(n)=g(n)+h(n), where g(n) is the cost from the start and h(n) is the heuristic cost. Invalid or occupied neighbors were skipped. The path was constructed by backtracking voxel indices, returning None if no path exists. In world_traj.py, graph_search was called with astar=True to compute the dense path.

Yes, we applied post-processing to the dense path to reduce the number of waypoints and ensure the trajectory is smooth and efficient. This was done in our world_traj.py through two main steps:

- Ramer-Douglas-Peucker (RDP) Algorithm: It simplified the dense path by removing redundant points while preserving the path's overall shape.
- Collinearity Check: We further simplified the path by removing points that lie on a straight line between two other points, as these points are redundant for a smooth trajectory.

TABLE II: Tuning Parameters and Values

Parameter	**Value**
Resolution	$[0.2, 0.2, 0.2]$ m
Safety Margin	initial = 0.50 m, min = 0.1 m, step = 0.05 m
A* Weight	1
Neighbor Offsets	$x, y, z \in \{-1, 0, 1\}$, except $(0, 0, 0)$ (26)
RDP Epsilon	0.26 m
Collinearity Tol.	1×10^{-3}
Segment Velocities	4.2 m/s (> 3.2 m), 3.2 m/s (1.2–3.2 m), 2.2 m/s (≤ 1.2 m)

ii) Time allocation between waypoints was handled in the _compute_segment_times method of the WorldTraj class in world_traj.py for our code. The goal was to assign timestamps to each waypoint such that the drone moves smoothly from one waypoint to the next, with velocities that adapt to the distance between consecutive waypoints. This adaptive velocity approach ensured that the drone moves faster over longer segments and slower over shorter ones, balancing speed and control of the drone for our control parameters.

- Computing Segment Distances: We calculate the Euclidean distance between consecutive waypoints.
- Assigning Velocities: Then we assign the appropriate velocity for each segment based on its distance, using predefined thresholds.
- Calculating Segment Times: Next we determine the time to traverse each segment as

$$t_i = t_{i-1} + \frac{\text{dist}_i}{\text{velocity}_i}, \quad t_0 = 0,$$

where $\text{dist}_i = \|p_i - p_{i-1}\|$ is the Euclidean distance between waypoints.

(iii) The _compute_smooth_trajectory method fits a cubic spline $s(t)$ to waypoints for each axis (x, y, z). Between timestamps t_i and t_{i+1}, the spline is a cubic polynomial:

$$s(t) = a_i + b_i(t - t_i) + c_i(t - t_i)^2 + d_i(t - t_i)^3.$$

Continuity Conditions: The spline ensures continuity: - Position: $s(t_i) = x_i$ (or y_i, z_i).
- Velocity: $s'(t_i^-) = s'(t_i^+)$.
- Acceleration: $s''(t_i^-) = s''(t_i^+)$.
Boundary Condition: The bc_type='clamped' sets:

$$s'(t_0) = 0, \quad s'(t_{N-1}) = 0,$$

ensuring zero velocity at start and end.

Trajectory Evaluation (update): The 'update' method evaluates the trajectory at time t: - If $t \leq t_0$, position is p_0, with velocity, acceleration, and jerk as zero. - If $t \geq t_{N-1}$, position is p_{N-1}, with zero derivatives. - Otherwise:

- Position: $\text{pos}(t) = [s_x(t), s_y(t), s_z(t)]$.
- Velocity: $\text{vel}(t) = [s'_x(t), s'_y(t), s'_z(t)]$, where $s'(t) = b_i + 2c_i(t - t_i) + 3d_i(t - t_i)^2$.
- Acceleration: $\text{acc}(t) = [s''_x(t), s''_y(t), s''_z(t)]$, where $s''(t) = 2c_i + 6d_i(t - t_i)$.
- Jerk: $\text{jerk}(t) = [s'''_x(t), s'''_y(t), s'''_z(t)]$, where $s'''(t) = 6d_i$.
- Yaw: $\text{yaw}(t) = \arctan 2(v_y(t), v_x(t))$.

Properties: The cubic spline ensures continuity, minimizing acceleration for smoothness. Jerk is piecewise constant, with discontinuities at waypoints mitigated by sparse waypoints. The trajectory is dynamically feasible, with adaptive velocities ensuring safe navigation.

(iv) Smoothness: The trajectory ensures smoothness via cubic spline interpolation (as stated above), achieving continuity (continuous position, velocity, and acceleration). The piecewise polynomials between each pair of waypoints at times ti and ti+1 was fitted by the spline (cubic polynomial). The _compute_smooth_trajectory method fits splines to waypoints, with clamped boundary conditions ensuring smooth starts and stops. Jerk is constant within segments but discontinuous at waypoints; RDP sparsification (epsilon = 0.26 m) reduces

these discontinuities. Adaptive velocity allocation enhances smoothness by slowing the drone around turns, minimizing overshooting. Overall, the trajectory was smooth, well-suited for the Crazyflie, with minimal abrupt changes.

Feasibility: The trajectory was dynamically feasible as seen during our experiments. The adaptive velocity (within drone limits) reduced overshooting by slowing down for short segments. Jerk discontinuities were mitigated by sparse waypoints and adaptive velocities, with RDP ensuring waypoints are not too close. The A* algorithm in graph_search ensured collision-free paths with a safety margin (0.50 m, adjustable to 0.1 m), making the trajectory feasible in constrained environments. A limitation was the zero yaw rate, which could be improved.

Overall: The trajectory was smooth and feasible, balancing continuity and dynamic/environmental constraints for the experiments and led us to pass the window in the real world map.

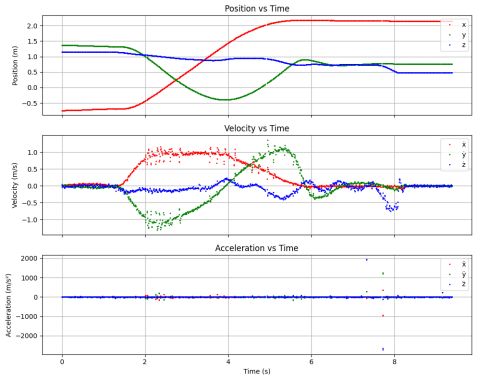


Fig. 4: Derivative plots for Map 2

IV. Maze Flight Experiments

This section presents the results of Maze Flight Experiments, as shown in Fig.5 to Fig.7. These 3D plots display world obstacles, waypoints, planned trajectories, and actual flight paths. In addition, a plot shows the position and velocity vs time in maze 1 for actual flight is shown in Fig.8.

From Fig.8, we can see that the tracking error between planned and actual positions is quite small, with a maximum error of around 0.1. The similar tracking error occurs in maze 3 but in maze 2 it could be much larger, with a maximum error of around 0.5. From the shape of trajectory in fig.6 to fig.8, we speculate that the cause of the error is related to the shape of the trajectory, and the larger the overall curvature of the trajectory, the greater the error. The trajectory in maze 2 clearly has a larger curvature than those in mazes 1 and 3, so it has a larger tracking error. To make our planned trajectories safer, we could ensure that the planned trajectory follows a straight line as much as possible and reduce the quadrotor's speed when encountering large turns.

In our lab, at first we use a faster speed and larger margin, which resulted in quadrotor bypassing it instead of passing through it. Then we reduced the margin, but we failed due to a collision with the inner wall of the window. Therefore, our trajectory may not be more aggressive in speed, otherwise we have to increase the margin to find a safer trajectory.

Although the overall speed could not be improved, we could still change the local speed such as increasing the linear speed while decreasing the curve speed. This requires us to have a precise relationship between velocity and trajectory curvature. In addition, we observed that in the window, where the quadrotor is the closest to obstacles, is the place most likely to fail. So, we could also decrease the speed in narrow passages to make the system more reliable.

If we were to add another session, we think it would be very interesting to challenge the entire maze from 1 to 2 and then to 3 continuously without landing. Hovering at the designated task point requires us to break down the task into three small segments, which is challenging and interesting but not particularly difficult.

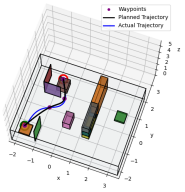


Fig. 5: 3D plot of Maze 1 Experimental Results

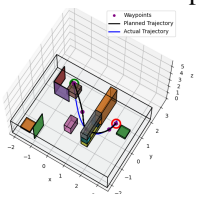


Fig. 6: 3D plot of Maze 2 Experimental Results

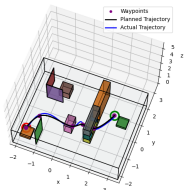


Fig. 7: 3D plot of Maze 3 Experimental Results

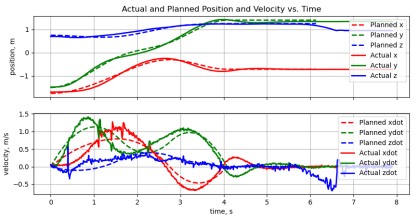


Fig. 8: Tracking Error Between Planned and Actual Trajectory of Maze 1