MEAM 6200: Project 1.4 Team 19

Archit Hardikar
University of Pennsylvania
Philadelphia, USA
architnh@seas.upenn.edu

Bo Wu
University of Pennsylvania
Philadelphia, USA
bobwu@seas.upenn.edu

Gabriel Unger
University of Pennsylvania
Philadelphia, USA
gunge1@seas.upenn.edu

Saibernard Yogendran
University of Pennsylvania
Philadelphia, USA
bernie97@seas.upenn.edu

I. Introduction and System Overview

Autonomous navigation through complex and dynamic environments has been recognized as one of the *Grand Challenges* in current robotic research [1]. Thanks to increased popularity and decreased cost, drones have become a good platform to conduct research towards solving this challenge. In this work, we explored a simpler version of this challenge, and developed the tools needed for a physical quadrotor to successfully navigate through several maps in a maze like environment. We utilize a simulated python environment to develop and evaluate three critical components: a controller, a trajectory generator, and a path planning algorithm. All code was written using python version 3.8, and the python simulator was supplied by the MEAM 6200 teaching team from the University of Pennsylvania.

The simulated environment was utilized to both tune the controller and optimize the trajectory generator to avoid collisions while maximizing performance, that is minimizing overshoot and large deviations from desired trajectory. After developing these, the next step consisted of transitioning from a simulated environment to a physical one. To do this we utilized the crazyflie 2.0, an open source drone that weights 27 grams and is approximately 60 mm $\times$ 60 mm in size. During this transition we evaluated how our algorithms and implementation performed on a real robot, and tuned them for increased performance.

We demonstrated the capability of the crazyflie to fly a planned trajectory and to traverse autonomously through a static environment using a path planning algorithm and successfully avoid collisions with obstacles through tuning parameters of the controller, trajectory generation, and path planning algorithm.

In order to evaluate the performance of our physical system we utilized a Vicon motion capture system. This system recorded the position and orientation of the quad-rotor and allowed for the reconstruction of the actual flight path. All of the computation is being performed off board on Ubuntu machines taking in position data from the vicon, and then sending thrust commands to the crazyflie remotely.

II. Controller

For this project, our team used a Geometric Non-Linear controller, where the quadrotor's b_3 axis is tilted to a point in the desired direction and a thrust is applied as seen in Figure 1. The position controller is designed using a second-order response method. The intuition behind the position controller is to minimize the position error and the velocity error by calculating appropriate control inputs.

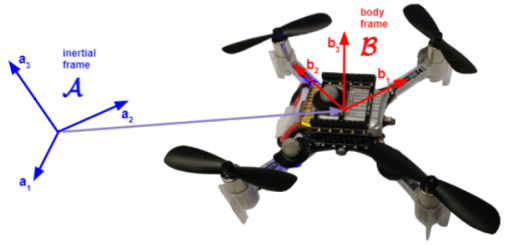

Fig. 1: Crazyflie drone showing different frames of reference. *(This image was supplied by MEAM 6200 teaching team from The University of Pennsylvania)*

The position controller uses the following equations based on a simple Proportional-Derivative control strategy:

$$\ddot{r_{des}} = \ddot{r_T} - K_d(\dot{r} - \dot{r_T}) - K_p(r - r_T) \tag{1}$$

Position Error is defined by

$$e_r = r_{current} - r_{des} \tag{2}$$

Velocity Error is defined by

$$\dot{e_r} = \dot{r_{current}} - \dot{r_{des}} \tag{3}$$

Here, K_p and K_d are the proportional and derivative control gains matrices, respectively with units of $1/s^2$, and $1/s$. These gains were tuned using a second-order response method with a desired settling time (T_s). The natural frequency (ω_n) and the control gains were calculated as follows:

Natural frequency: is defined by

$$\omega_n = \frac{2\pi}{T_s} \tag{4}$$

$$\ddot{e} + K_d\dot{e} + K_p e = 0 \tag{5}$$

$$\ddot{e} + 2\xi\omega_n\dot{e} + {\omega_n}^2 e = 0 \tag{6}$$

Combining 5 and 6 we get the following equations for the gains:

Proportional gain: is defined by

$$Kp = 0.6(\omega_n)^2 \tag{7}$$

Derivative gain: is defined by

$$K_d = 0.7(2\omega_n) \tag{8}$$

The 0.6 and 0.7 are multiplied for reducing the gain value in the actual drone compared to the simulation as a thumb value.

After calculating the desired acceleration, we calculate the thrust force F^{des} using the following equation:

$$F^{des} = m\ddot{r}^{des} + \begin{bmatrix} 0 \\ 0 \\ mg \end{bmatrix} \tag{9}$$

We then project the thrust force in the quadrotor Z-axis direction. Hence, we get:

$$u_1 = b_3^T F^{des} \tag{10}$$

After calculating the command force, we calculate the desired orientation of the quadrotor R^{des}.

$$b_3^{des} = \frac{F^{des}}{\|F^{des}\|} \tag{11}$$

$$a_\psi = \begin{bmatrix} cos\psi_T \\ sin\psi_T \\ 0 \end{bmatrix} \tag{12}$$

$$b_2^{des} = \frac{b_3^{des} \times a_\psi}{\|b_3^{des} \times a_\psi\|} \tag{13}$$

$$R^{des} = [b_2^{des} \times b_3^{des}, b_2^{des}, b_3^{des}] \tag{14}$$

$$e_R = \frac{1}{2}(R^{desT} - R^T R^{Des})^V \tag{15}$$

$$u_2 = I(-K_R e_R - K_\omega e_\omega) \tag{16}$$

$$e_\omega = \omega - \omega^{des} \tag{17}$$

where K_R and K_w are diagonal gain matrices that have dimensionless and rad/s respectively.

Here, ψ_T represents the desired yaw angle which was kept as zero as to point the quadrotor in one direction. After finding out the desired orientation, the error was calculated between the desired orientation and the current orientation. This was similarly repeated for angular velocity. The gains for these quadrotor dynamics were tuned in the simulation.

1) Steady-state Error: The controller aims to minimize the position and velocity errors. In an ideal scenario, the steady-state error should approach zero as the quadrotor reaches its desired position and velocity. However, in practice, there might be a small steady-state error due to disturbances, modeling inaccuracies, or limitations in control authority. An example of a Z step response can be seen in figure 2.

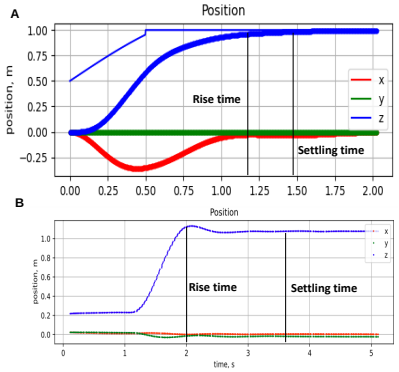

Fig. 2: A) Z step response from simulation Step B) actual z response from vicon C) Map two from simulation D)map two from vicon

2) Damping ratio: The controller was designed to be critically damped as the control gains for the position controller, kp and kd, were calculated using a second-order response method with a ratio of 1.

3) Rise time: The rise time can be estimated based on the natural frequency, ω_n, and the desired settling time (T_s). A higher natural frequency generally results in a faster rise time, as the system responds more quickly to changes in the desired position. The rise time recorded in the simulation is approximately 1.1 s and the rise time recorded in the hardware is approximately 2 s.

4) Settling time: The desired settling time (T_s) for the position controller is specified in the code as 1.627 seconds. This value was used to tune the control gains, aiming to achieve a response with this settling time. However, the actual settling time may vary depending on the specific scenario, disturbances, and other factors affecting the system's dynamics. The settling time recorded in the simulation is approximately 0.5 s and the settling time recorded in the hardware is approximately 1.3 s.

Our attitude controller is not being used directly. Instead the on board crazyflie controller is being utilized and is using the position data from the vicon as input directly.

In order to account for differences between simulation and the physical set up we had to decrease our gains by 30 % and turn our speed down to 1 m/ s. The differences between the simulation and the physical experimental set up could come from either side. The simulation is not a perfect physics simulator, meaning it could be lacking in representing certain real world physics such as drag. It also could allow for things that are infeasible in the real world, such as having the motors spin with negative thrust, or going past their motor limits if it was not specified in the code. On the physical set up side, hardware can be difficult to work reliably and thus some sort of error will always be able to be attributed to that. One such example of this is how one drone may need different gains from another since one or more of its motors may be performing less than ideal.

III. Trajectory Generator

1) Generating waypoints: The code uses a graph search algorithm (Dijkstra or A* search) to generate waypoints given a known map. Either path planner can be used by changing the input argument to True or False. Passing in True results in A* running, and it is what we used to complete all of our Maze Flight experiments. It passes in the world object, which represents the environment obstacles, and the resolution and margin parameters for the Astar algorithm. The resolution parameter represents the grid's resolution, while the margin parameter determines the safety margin around obstacles. The code uses a 0.11 m resolution in XYZ, and a collision margin of 0.25 m.

After obtaining the dense path from the graph search algorithm, the code directly assigns it to the waypoints without any post-processing.

2) Time allocation between waypoints: The code allocates time between waypoints based on the distance between consecutive waypoints and a fixed velocity value of 1.0 m/s

For each segment, it calculates the time as the distance divided by the fixed velocity. That is:

$$\delta t = \frac{Distance}{Velocity} \tag{18}$$

Then, the start times for each waypoint are stored in self.start_times.

3) Trajectory generator: The trajectory generator in this code uses a simple piece wise linear function. The position x between two consecutive way points is determined by linear interpolation based on the elapsed time t since the start of the current segment. The velocity $\dot{x}$ is constant within each segment, as it is set to the product of the fixed velocity and the normalized direction vector between two consecutive waypoints. Higher-order derivatives (acceleration, jerk, and snap) are all set to zero, indicating no change in velocity during each segment. Given the current time t, the desired position x and velocity $\dot{x}$ are computed as follows:

$i \leftarrow length(StartTime)$
if $StartTime$[i] $\leq time \leq StartTime$[i+1] **then**
  $xdot = velocity * directions[i]$;
  $x = waypoints[i] + xdot * (t - start\_times[i])$;
end if

where directions is defined by the unit vector from the previous way point to the next. and way points are a list of points found from the path planner. This implementation is relatively simple and does not account for smooth transitions between way points, nor does it consider higher-order constraints like acceleration, jerk, or snap.

IV. Maze Flight Experiments

In order to evaluate our controller and trajectory's capabilities, we tested our system's ability to successfully navigate through three different trajectory in a maze environment. The maze consisted of large obstacles like boxes, carts, nets and hurdles. An examples of this environment can be seen in figure 3. The three flight experiments we did were designed to test the capability of our planner to locate a safe path, and our controller to maintain the desired trajectory. The location of the obstacles, the way points, the planned trajectory, and the actual flight path for each of these experiments can be seen in Figure 4. Flight footage of each map can be seen for map one, two, and three respectively as well (Movies: S1-S3).

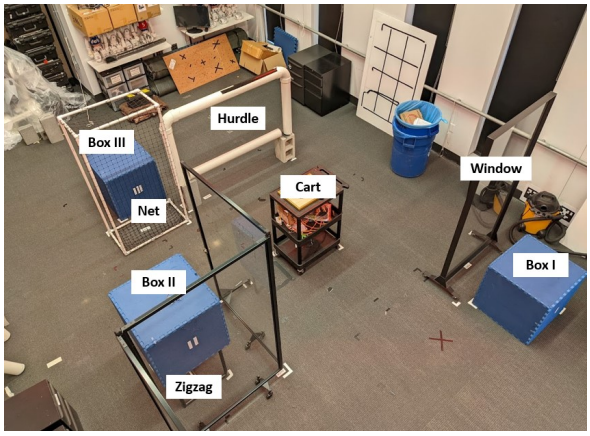

Fig. 3: Maze from 2022, provided by teaching team of MEAM 6200 at The University of Pennsylvania

For each flight test, in addition to the actual flight path, the position vs time for the actual flight was recorded. An example can be seen for map two in figure 5.

Based on plots in figure 4, the flight is tracking the planned trajectory in a close manner, there exists overshoot at the turning corners of the trajectory, we also plotted the planned position vs. actual position of the drone for a closer look. It turns out that position tracking error (overshoot) is around the magnitude of 0.1m for the drone and the settling time is approximately 0.5s (figure 2). Given this observation, it is advisable to multiply the margin used in path planning by a factor α, where $\alpha > 1$ and $\alpha \propto$ settling time.

During the experiment, we also tried to make the paths more aggressive. For example, for maze experiment 3, by reducing the margin, the planned path can alternatively make the drone to fly through the hurdle. However, the overshoot of the drone makes it unavoidably crash into the edge. Without significantly reducing the velocity or doing a thorough fine tune of the controller, a safer way to do the path planning is to increase the margin. This will result in the planner viewing the hurdle as a wall that is not traversable, ending up to generate a less aggressive path.

After evaluating the performance of these flight test, future work could be done to improve the speed and the reliability of our system in the maze. This presented system would likely gain a drastic by changing the trajectory generation. Currently this implementation uses a constant velocity trajectory gen-

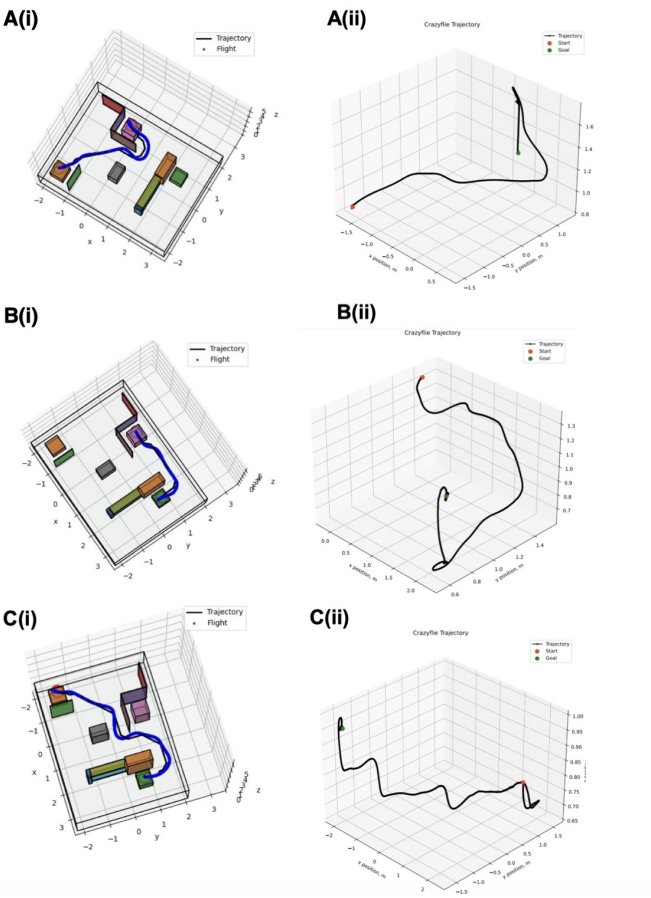

Fig. 4: A(i) Maze one planned trajectory A(ii) Maze one actual flight path B(i) Maze two planned trajectory B(ii) Maze two actual flight path C(i) Maze three planned trajectory C(ii) Maze three actual flight path

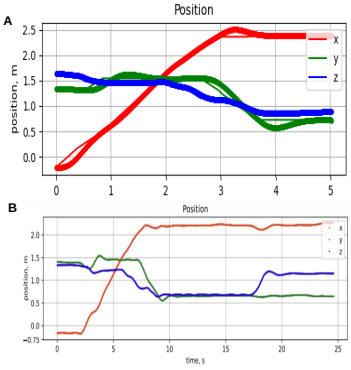

Fig. 5: Position & Velocity vs Time for Maze two from simulation

erator, but changing to a generator that minimizes snap, like seen here [2], would likely allow us to increase the top speed. In addition to this, the minimization of the snap should also allow us to have some increased aggressive trajectory due to being able to have variable velocities, and its derivatives. This would allow for tighter turns, better paths, and increases maneuverability between way points.

Provided more time in the lab we would have liked to have tested our final tuned parameters on the remaining two maps. We were able to put together a list of parameters to pass every map, however they did differ from each other map to map. In the end we were able to find a set of parameters that passed all three maps in simulation, but were untested on map one and map two in the physical set up.

In addition to this, it would be interesting to try and have the drone perform some function. One example would be to great a trajectory that could mimic utilizing the drone for surveillance. To do this, added trajectory constraints could be constructed such that the drone tries to maintain a max distance off the ground. This could be analogous to a drone flying over a a desired space to record data. In order to gather data like clear pictures, or sensor data that has a maximum range the drone would need to be close to the ground, except when avoiding obstacles. The addition of further sensing and perception would be a better way to implemented this, however this still could be performed in the physical set up. Lastly it could be interesting to see if and how the distance of the ground variable would affect which trajectory the robot takes.

V. Conclusion

Our demonstrated system, while only being a simple constant velocity controller, successfully implemented our algorithms onto a physical system and was able to safely navigate through three different maps in a maze environment. The system could be further improved through more robust trajectory generators allowing for more aggressive maneuvers.

References

[1] Guang-Zhong Yang, Jim Bellingham, Pierre E. Dupont, Peer Fischer, Luciano Floridi, Robert Full, Neil Jacobstein, Vijay Kumar, Marcia McNutt, Robert Merrifield, Bradley J. Nelson, Brian Scassellati, Mariarosaria Taddeo, Russell Taylor, Manuela Veloso, Zhong Lin Wang, and Robert Wood. The grand challenges of iscience robotics/i. *Science Robotics*, 3(14), January 2018.

[2] Daniel Mellinger and Vijay Kumar. Minimum snap trajectory generation and control for quadrotors. In *2011 IEEE International Conference on Robotics and Automation*, pages 2520–2525, 2011.

Supplementary Materials

Available upon Request. Please contact Gabriel Unger at gunge1@seas.upenn.edu

Movie S1.
Flight test of Map one

Movie S2.
Flight test of Map two

Movie S3.
Flight test of Map three