Project 1 Report: Quadrotor Control and Tracking

Team 17: Feifei Duan, Peng Li, Guoyao Shen

I. Introduction

This lab report presents experiment results and discussion of controlling a quad rotor flying through known environment by a controller, a path planning function, and a trajectory generator written in Matlab. They were first tested in a simulator and later implemented onto a real quad rotor in lab environment. The quad rotor we used in the lab is the CrazyFile. It has a motor to motor distance of 92 mm and a mass of 30 g. An onboard IMU of the quad rotor provides feedback of angular velocity and acceleration. Our controller takes the feedback values from onboard IMU and the desired state values from trajectory generator as inputs and returns actual motor total forces and the moment vector in the body-fixed frame to the motor controller. For the first lab, the quad rotor was controlled to fly through given waypoints. For the second lab, the quad rotor was controlled to fly though a maze without collision based on a given 3D map with obstacle. Controller used in both labs are the same and is discussed in section 2. Trajectory generator used in and experiment results of lab 1 and lab 2 are discussed in section 3 and 4 respectively.

II. Controller

We used geometric nonlinear controller for both labs. The main idea is that $\hat{\boldsymbol{b}}_3$ is tilted towards the desired direction. The controller is divided into two parts: position control and attitude control. The position control is to get the total thrust u_1. The attitude control is built on top of the position control. First, the position PD control equation is:

$$\ddot{\boldsymbol{r}}^{des} = \ddot{\boldsymbol{r}}_T - \boldsymbol{K}_d(\dot{\boldsymbol{r}} - \dot{\boldsymbol{r}}_T) - \boldsymbol{K}_p(\boldsymbol{r} - \boldsymbol{r}_T) \tag{1}$$

From Newton's Equation of motion, we have,

$$m\ddot{\boldsymbol{r}}^{des} = \begin{bmatrix} 0 \\ 0 \\ -mg \end{bmatrix} + {}^A\boldsymbol{R}_B \begin{bmatrix} 0 \\ 0 \\ u_1 \end{bmatrix} \tag{2}$$

Let $\boldsymbol{F}^{des}$ be,

$$\boldsymbol{F}^{des} = {}^A\boldsymbol{R}_B \begin{bmatrix} 0 \\ 0 \\ u_1 \end{bmatrix} = m\ddot{\boldsymbol{r}}^{des} + \begin{bmatrix} 0 \\ 0 \\ mg \end{bmatrix} \tag{3}$$

Then u_1 can be derived from the following equation,

$$u_1 = ({}^A\boldsymbol{R}_B \begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix})^{-1} \boldsymbol{F}^{des} = \hat{\boldsymbol{b}}_3^T \boldsymbol{F}^{des} \tag{4}$$

The attitude control starts from Euler's Equation of Motion:

$$I \begin{bmatrix} \dot{p} \\ \dot{q} \\ \dot{r} \end{bmatrix} = u_2 - \begin{bmatrix} p \\ q \\ r \end{bmatrix} \times I \begin{bmatrix} p \\ q \\ r \end{bmatrix} \tag{5}$$

where u_2 is,

$$u_2 = \begin{bmatrix} l(F_2 - F_4) \\ l(F_3 - F_1) \\ M_1 - M_2 + M_3 - M_4 \end{bmatrix} = I \begin{bmatrix} \dot{p} \\ \dot{q} \\ \dot{r} \end{bmatrix} + \begin{bmatrix} p \\ q \\ r \end{bmatrix} \times I \begin{bmatrix} p \\ q \\ r \end{bmatrix} \tag{6}$$

As we mentioned above, $\hat{\boldsymbol{b}}_3$ is tilted towards the desired direction. Under this assumption, $\hat{\boldsymbol{b}}_3^{des}$ has the same direction as $\boldsymbol{F}_{des}$. Hence, we can obtain $\hat{\boldsymbol{b}}_3^{des}$ from the following equation:

$$\hat{\boldsymbol{b}}_3^{des} = \frac{\boldsymbol{F}_{des}}{\|\boldsymbol{F}_{des}\|} \tag{7}$$

Next, $\hat{\boldsymbol{b}}_2^{des}$ is perpendicular to both $\hat{\boldsymbol{b}}_3^{des}$ and the vector $\boldsymbol{a}_\psi$ that is $\boldsymbol{a}_1$ after yaw direction rotation.

$$\hat{\boldsymbol{b}}_2^{des} = \frac{\hat{\boldsymbol{b}}_2^{des} \times \boldsymbol{a}_\psi}{\left\|\hat{\boldsymbol{b}}_2^{des} \times \boldsymbol{a}_\psi\right\|} \tag{8}$$

where,

$$\boldsymbol{a}_\psi = \begin{bmatrix} \cos\psi_T \\ \sin\psi_T \\ 0 \end{bmatrix} \tag{9}$$

Now we can get $\boldsymbol{R}^{des} = \left[\boldsymbol{b}_2^{des} \times \boldsymbol{b}_3^{des}, \boldsymbol{b}_2^{des}, \boldsymbol{b}_2^{des}\right]$. Error in orientation ${\boldsymbol{R}^{des}}^T R$ can be obtained by the following equation,

$$\boldsymbol{e}_R = \frac{1}{2}({\boldsymbol{R}^{des}}^T R - \boldsymbol{R}^T \boldsymbol{R}^{des})^\vee \tag{10}$$

The real attitude control input should be:

$$\boldsymbol{u}_2 = I(-\boldsymbol{K}_R \boldsymbol{e}_R - -\boldsymbol{K}_\omega \boldsymbol{e}_\omega) \tag{11}$$

where $\boldsymbol{e}_\omega = \boldsymbol{\omega} - \boldsymbol{\omega}^{des}$. In our controller, the attitude control input we use set $\boldsymbol{\omega}^{des}$ to zero. Because the output ψ from trajectory generator is set to zero all the time.

We changed the mass in lab 1 because when we first tested the built-in hover function with our controller, the quad rotor oscillated at the goal position. Because when quad rotor at the goal position, the total thrust should has the same value as its gravitational

force in order to keep the quad rotor in stationary. If the input mass is off from the actual mass, the thrust in stationary mode will also be off, which leads to oscillation. After we adjusted mass, the oscillation decreased, but still the quad rotor was jiggling at the goal position. We started to tune $\boldsymbol{K}_p$ and $\boldsymbol{K}_p$ value by decreasing them until hovering was steady enough. The final gain we used in lab 1 is $\boldsymbol{K}_p$ = [8, 0, 0; 0, 8, 0; 0, 0, 8]; $\boldsymbol{K}_d$ = [8, 0, 0; 0, 8, 0; 0, 0, 8]; $\boldsymbol{K}_R$ = [380, 0, 0; 0, 390, 0; 0, 0, 390]; $\boldsymbol{K}_\omega$ = [20, 0, 0; 0, 22, 0; 0, 0, 25]. When looking at the units for K parameters, Kp is taking double derivative with respect to time on position difference and Kd is taking first derivative respect to time on velocity difference. This means that they use different derivatives of times as input and control the output to respond them. And can also be interpreted as "the velocity for the system to respond the error in different derivatives of the time". Kp mans the system will respond the "error in zero derivative of time (which is the position)" while Kd means the system will respond the "error in first derivative of time (which is the velocity)". Similarly, Kr refers to the velocity for system to respond the error in orientation and Kw refers to the velocity for system to respond the error in angular velocity.

As shown in fig 3a, in lab 1 we can find there are steady error in for z and x (after 12s). In general, controlling method without I control will cause steady error. For example, if we use P control, the relationship between output and input is shown in eq 12.

$$output = K_p \times (des - input) \tag{12}$$

Where des is the desired value we want to output. However, if the system comes to the steady state, which means input is the same as output, then we have:

$$output = \frac{des}{1 + \frac{1}{K_p}} \tag{13}$$

But the output which is also the new input is not equal to des, and the system already reach a steady state, thus cause the steady error. If we want to fix the steady error, we can add I control to the controller, since it will accumulate the error along time so that the system can fix it.

III. Trajectory Generator

The trajectory generator we used in lab 1 is simple without velocity profile. The desire velocity is set to zero all the time. We just define the desire position that changes with time and let the controller handle the velocity. When the trajectory generator is first called, it store the path in a persistent variable called *path* and then every sub-paths between waypoints are discretized evenly by adding new waypoints in between and this new path is stored into another persistent variable called *path_planning*. Every time when trajectory generator is called, it returns a new position in *path_planning* every 0.02 second. Given that time allocation, to force the quad rotor to pass every waypoint without velocity profile, the waypoints in the original path are inserted repeatedly into *path_planning*, see Figure 1. The tradeoffs of this approach are, since there is no velocity profile, $\boldsymbol{v}^{des} = [0, 0, 0]^T$, the velocity is slow and tuning position and velocity gain plays a significant role in the control system. If Kp is too small and Kd is large, the quad rotor may not be able to follow the path closely. Even though adding intermediate waypoints multiple times into trajectory path is a way to force the quad rotor pass waypoints, it can be dangerous in environment with obstacle and when tuning manually either *Kp* and *Kp* or the number of repeated waypoints is not allowed.

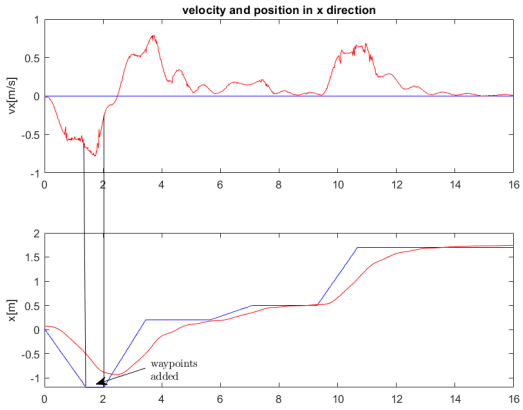
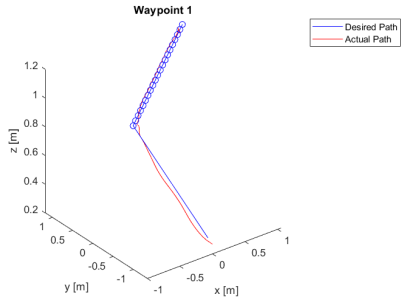
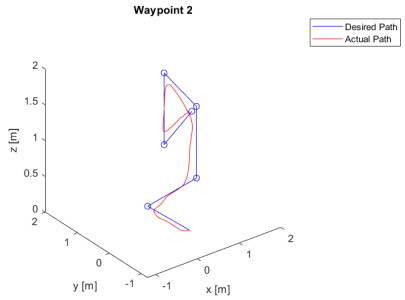


Fig. 1: position and velocity of x axis

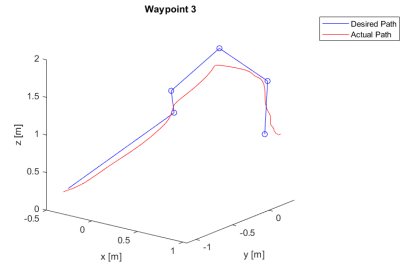
From our lab results shown in figure 2 above, quad rotor follows the path with distinct offset when direction changing is involved in the path, such as in figure 2b and 2c. Let's take a look into the results of the second trajectory, see figure 3. We notice that there is a time offset and a distance offset from figure 3a. The reason of occurrence of error of this kind may be small *Kp* value. So we need to be more aggressive, increasing *Kp* by putting more weight on position error. It will be faster than the first solution but this may lead to oscillate during hover (when it reaches the goal). Or we can add more repeated waypoints into *path_planning* and this will allow more time for quad rotor to catch up the path which isn't good during limited time operation. Since for this part we use a generator which only gives the desired position, we decided not to be so aggressive since we need the controller to handle velocity and acceleration (for lab2 generator please refer to section 4-B). A better solution is adding continuous velocity profile. Hence we



(a) Trajectory 1

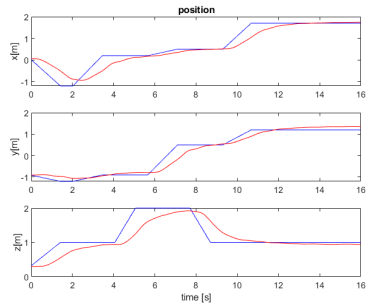


(b) Trajectory 2

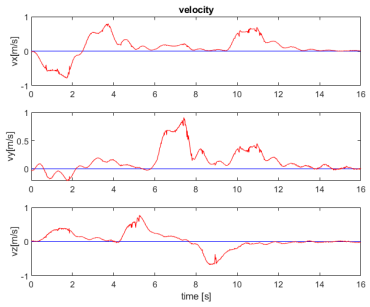


(c) Trajectory 3

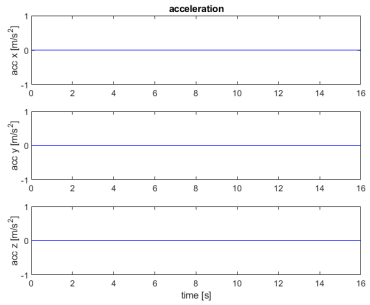
Fig. 2: 3D plot of Trajectories



(a) position of trajectory 2



(b) velocity of trajectory 2



(c) acceleration of trajectory 2

Fig. 3: Position, Velocity, Acceleration Plot of the second trajectory

changed our trajectory generator to a ”stop and go” in lab 2, defining not just desired position, but also velocity and acceleration. For the trajectory generator we use in lab 2, please refer to section 4-A. As for the oscillation, we can find them both in lab1 and lab2, and we'll discuss this further in section 4-B. Since the generator we use for lab1 is simple, the trade off for this is that we give up the control of velocity and acceleration which in return we have a pretty easy generator to use. But we use a more complicated generator in lab2, the tradeoff for that will be discussed in section 4-A.

IV. Flying Through the Maze

A. Modified Trajectory Generator

In the lab2 session, we made a few changes to let the quadrator better fly through the maze. First, we changed our trajectory generator to ”stop and go” in lab 2, which means the boundary velocity and acceleration are set to zero and use quintic interpolation (min acceleration) between each two waypoints.We observed that the quadrator followed the path more closely after we specified the planned velocity and acceleration besides the planned position. The ”stop and go” alleviated overshooting since the planned velocity would be close to zero when it need to change the directions. Second, we reduce the number of waypoints when two adjacent sections are in the same or very close directions. Without this step, the quadrator may have to accelerate and slow down many times unnecessarily when multiple waypoints are close to each other and almost forms a straight line. This significantly helped smoothing the path and worked very well together with our stop and go strategy.

The followings show more details of our new trajectory generator. Since the trajectory is stop and go,

we need to reduce the number of the waypoints used for interpolation. In addition, we also want the final waypoints keep the whole shape of the trajectory to avoid obstacles. For instance, as you can see in figure 4, the number of waypoints calculated by the pathfinder is 231, but "important" points we want to keep for our interpolation would be points indicated by green circles in figure 5. To achieve this, we need to solve 2 problems: delete waypoints that are "in same directions" and refine "sawtooth shape" into straight line. A typical example for "sawtooth shape" is shown in figure 6 as the black line.

ans	*1x1 struct*
map	*1x1 struct*
path	*231x3 double*
start	[6,-4.9000,3.2000]
stop	[8,18,3]
trajectory	*1x1 cell*

Fig. 4: Variables for Trajectory Generator

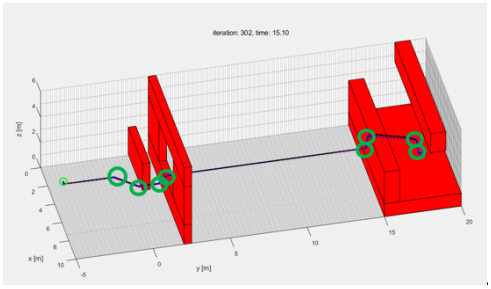


Fig. 5: Points Kept in the Trajectory

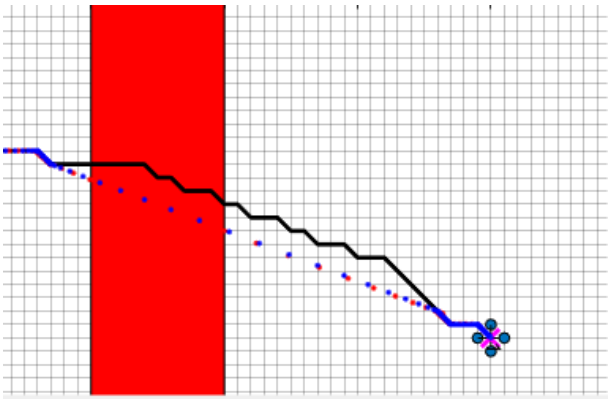
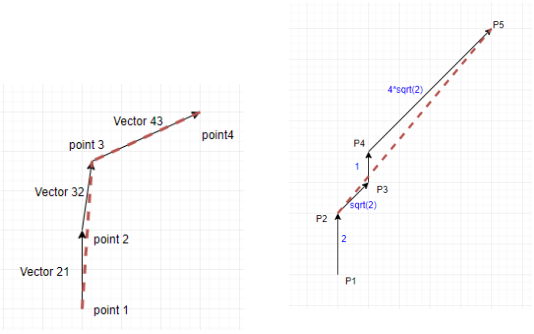


Fig. 6: Refinement of Sawtooth Trajectory

To achieve the first goal which is deleting "same direction" waypoints, the general idea we do this is to calculate the normalized orientation vector between every two points, and delete the in-between waypoints if two normalized orientation vectors are "same in certain threshold". An instance is shown in figure 7a. The direction of vector 21 and vector 32 are same within threshold, while vector 43 is much different from former two, so in this case, we would only delete point 2 and keep point 1, 3, 4 to guarantee the "general shape".



(a) (b)

Fig. 7: Reduce Waypoints Example I

To remove "sawtooth", the same-direction-filter we discussed above won't help, since sawtooth generally have 45 degree orientation difference in 2D. As you can find in figure 6, the length of each piece of the sawtooth are pretty small, from 1 time of the resolution to 2 times of the diagonal of the resolution of x-y plane. So in this case, we can refine the sawtooth by delete points whose length is in certain field, 1 to $\sqrt{2}$ resolution, for example. As shown in figure 7b, if we delete the latter point if the length between two points is between 1 to $\sqrt{2}$ time of the resolution, we will delete P3 and P4, and relink P2 with P5 so that we remove the sawtooth in the path.

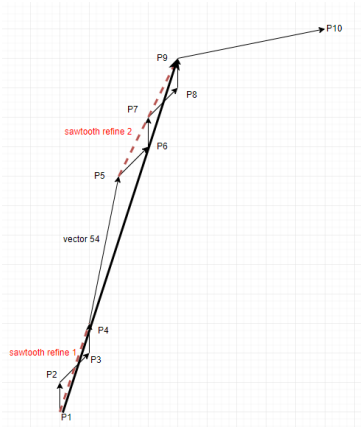


Fig. 8: Reduce Waypoints Example II

Remember that points generated by pathfinder are points neighbor every two in 26-neighbors, means that they always within the filed of the 1 to $\sqrt{2}$ (or $\sqrt{3}$) resolution which are going to be deleted, so we can't use the

sawtooth-refine directly. What we do is first filter and delete same direction points, leave the sawtooth and they are the only vectors whose length are between 1 to $\sqrt{2}$, then use sawtooth-refine, finally use same-direction-filter to refine the path again. The final step is to guarantee that the direction of two piece of sawtooth are close and can be seen as one piece, an example is given in figure 7b. Red dash lines are path after sawtooth-refine, since the direction of sawtooth refine 1, vector 54 and sawtooth refine 2 are close, so finally we will delete all other points and only keep P1, P9 and P10.

After the refinement discussed above, we calculate the vector between every two points and calculate the length of the vector. We set a average velocity to the path, together with the length of each vector, we can calculate the time for each piece of the interpolation. However, if the time assigned for certain piece is too small, it will make it hard for the controller to follow, so after calculating the time assigned for each piece, we will check if the time is smaller than certain threshold, if yes, we will assign it to a new smallest time (which can be tuned in code). Finally, with the time for each piece of the waypoints, we can calculate at what time the quadrotor should at which point, and use them together with boundary velocity and acceleration set to zero as the condition to calculate quintic interpolation.

So generally, we need to adjust the orientation error in same-direction-filter and min time in time filter and the field of length in saw-tooth-refinement. When adjusting the total time, just adjust the average velocity. As you can see, we only delete points from the points we get from pathfinder, and give up the no-stopping solution. In general this is safe and stable even though we might use more time for same map compared with the no-stopping. Infact, as we test the nonstop min snap interpolation in simulation, we find that if we do the interpolation after deleting points, the trajectory generated will overfit the waypoints, so we need to add new points along the trajectory in a certain density. However, the density of adding new points has relation with the overall shape of the trajectory, means the for one density value can not fit all kinds of the map trajectory. Even though we can use this during the lab since there is only one map, the solution can not guarantee adaption for different maps. So finally we decide to use the stop-and-go generator.

B. Lab2 Result

The followings are the result from lab2. The 3D plot shows that Our quadrator's actual trajectory are pretty close to the planned trajectory. However, when looking

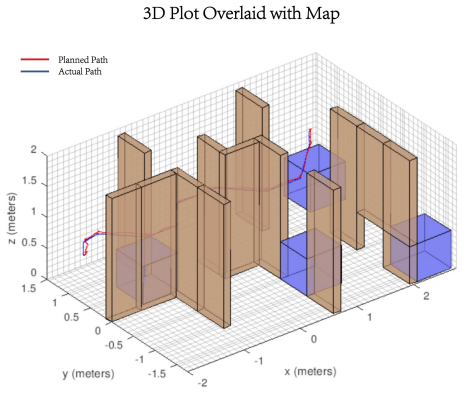


Fig. 9: 3D Plot Overlaid with Map

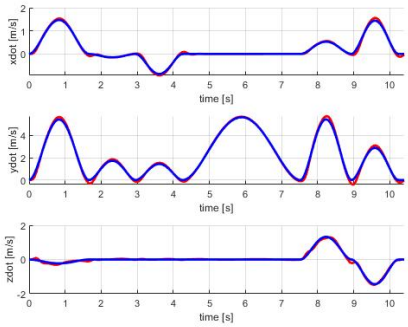
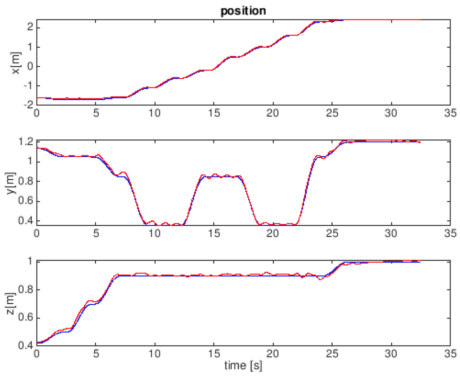
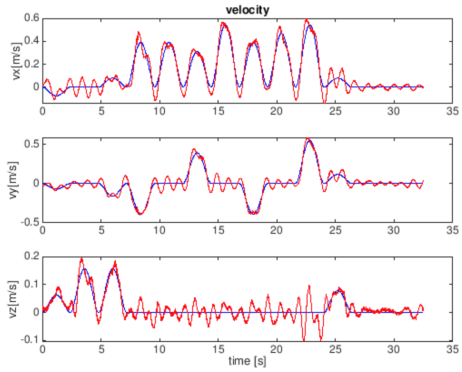


Fig. 10: Velocity Stays Constant Example

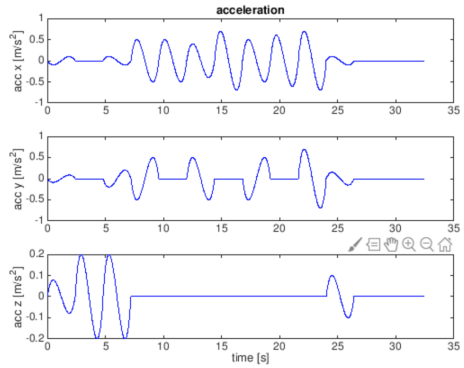
at the velocity of x, y and z axes, we can find that there are fluctuations between desired velocity and actual velocity, this is easier to find when the desired velocity stays as a constant, such as about 2 $\sim$ 5 s in vx and vy, 7 $\sim$ 25 s in vz and after 27s in all three axes. In fact, when simulated on Matlab, the "actual" velocity (and acceleration) can stay constant when desired value stays constant, such as shown in fig 10. Recall the first time in lab testing the hover for the quadrotors, the quadrotor may have small "fluctuation" even though the command is to hover at a certain point. The reason for this is that during the simulation, all K parameters are set and tuned according to a "good and precise" mass and other physical parameters, but the actual mass in real word and other physical values may have difference compared with in simulation, and there is also deviation between sensors and the controller which may cause this small fluctuation. In fact, if we look back again at the velocity of all three axes shown in fig 11b after 26s, the desired velocity for all three axes are zero, means to hover at that point, but the actual velocity still have small fluctuation. So this means that the "real" state can't be really "constant" if we want to hover and the real behavior to "hover" is to stay around the desired point within threshold under the control of the controller.



(a) position



(b) velocity



(c) acceleration

Fig. 11: Position, Velocity, Acceleration Plot of Maze trajectory, start position: left, goal position: 1 left)

Even though the actual velocity have small fluctuation, but they all stay under 0.1 m/s. And the position response worked well as shown in fig 11a. The whole system performs well without hitting any obstacles when flying through the maze. The quadrotor performs best in row direction, while shows a little bit oscillating and overshooting especially in the pitch and yaw directions. But all the deviation are relatively small and allow the quadrotor fly away from the obstacles. We tried two different start point locations and two different goal locations, and our quadrator flied through the maze safely with reasonable speed. Since the generator we use for this part is stop-and-go, if we want be more aggressive, just adjust the average velocity bigger. Remember that we have a time filter which will rest the time for a fragment if the time for it is too small, so all we need to do is to guarantee that the maximum speed for the longest fragment of the trajectory is no bigger than the maximum safe speed of the quadrotors.

V. Conclusion

Overall, our system achieved great performance. After we observed the result from the first lab, we improved our trajectory generator by applying stop-and-go strategy and reducing waypoints. As a result, our quadrator succeeded in the maze challenge given four different start and goal configurations. Slight overshooting and oscillation still exists, but in general, our system allow the quadrator followed the planned trajectories very closely.

Future work involves in using min snap to smooth the path and tuning the controller parameters to reduce oscillation. In the intermediate points, we only need to specify the continuity in derivatives and positions instead of letting the quadrator's velocity and acceleration become zero. Considering the dynamics of quadrator, applying min snap could avoid requiring the quadrator to do sharp turns or instantaneous velocity/acceleration change, which may help address the problem of overshooting. In addition, we may also apply some tricks as we did the last phase of project1. For example, we may need to add some additional waypoints back when necessary to ensure the spline trajectory won't result in obstacle collision. As discussed in section 4-A's tradeoff, the density of adding points has relation with the shape of the trajectory, so maybe we can try to "recognize" what shape it is (like if it has too many sharp turns or long straight lines) and use these data to recalculate and adjust the density of adding points. This would be more like to tell the quadrotors to "learn" the map and adjust its strategy according to it.