Flying Car and Autonomous Flight Engineer Nanodegree

Building a Controller

Minh Gia Hoang

Motor command calculation

To control the 4 quadrotors, we need to convert the desired 3-axis moment and collective thrust command given the controller (to be built later) to 4 individual motor thrust commands. This task depends on the quadrotor configuration which is shown as follows.

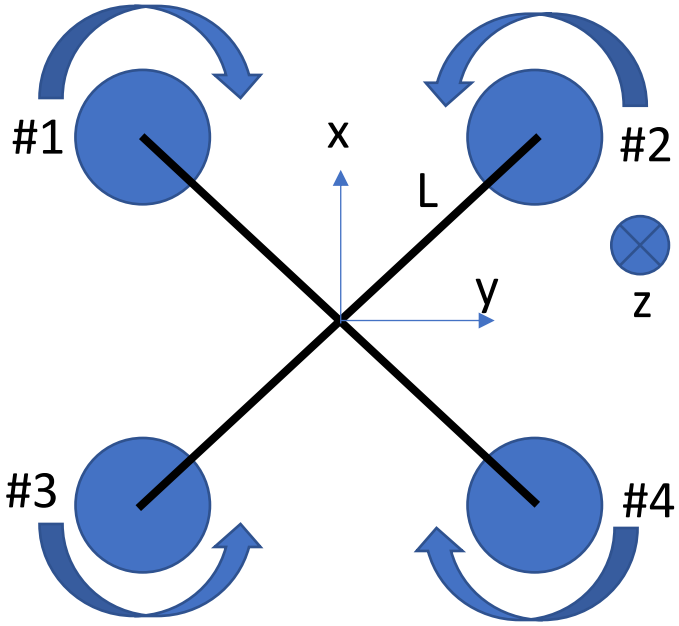

Figure 1: Quadrotor configuration (view from top).

The collective thrust F_Σ is the sum of 4 individual thrusts F_i, $i = 1, 2, 3, 4$ i.e.,

$$F_1 + F_2 + F_3 + F_4 = F_\Sigma. \quad (1)$$

The moment about x-axis M_x is made by the 4 individual thrust but the moments by F_1 and F_3 are positive while those by F_2 and F_4 are negative following the right hand rule.

$$(F_1 - F_2 + F_3 - F_4)\frac{L}{\sqrt{2}} = M_x, \quad (2)$$

where L is the distance from the vehicle origin to rotors.

Similarly for the moment about y-axis M_y, it gives

$$(F_1 + F_2 - F_3 - F_4)\frac{L}{\sqrt{2}} = M_y. \quad (3)$$

For the moment M_z about z-axis, rotors 1 and 4 rotates clockwise causing the vehicle to spin counterclockwise (according to the conservation of momentum) thus their moments τ_1 and τ_4 are negative (z-axis pointing down). Similarly, rotors 2 and 4 spin counterclockwise causing moments τ_2 and τ_4 positive.

$$-\tau_1 + \tau_2 + \tau_3 - \tau_4 = M_z.$$

Besides, $\tau_i = \kappa F_i$, where κ is the drag/thrust ratio. Thus,

$$(-F_1 + F_2 + F_3 - F_4)\kappa = M_z. \quad (4)$$

Collecting 4 equations (1), (2), (3), and (4), we solve 4 individual motor thrust. This is implemented in the following code block (lines 77 – 87 in `QuadControl.cpp`).

```
float l = L / sqrt(2.0f);
float c_bar = collThrustCmd;
float p_bar = momentCmd[0] / l;
float q_bar = momentCmd[1] / l;
float r_bar = momentCmd[2] / kappa;
cmd.desiredThrustsN[0] = 0.25f * (c_bar + p_bar + q_bar - r_bar);
cmd.desiredThrustsN[1] = 0.25f * (c_bar - p_bar + q_bar + r_bar);
cmd.desiredThrustsN[2] = 0.25f * (c_bar + p_bar - q_bar + r_bar);
cmd.desiredThrustsN[3] = 0.25f * (c_bar - p_bar - q_bar - r_bar);
```

Body rate control

The body rate control is implemented using P controller. The controller computes the angular acceleration commands by measuring the different between the desired and estimated body rates $pqrCmd$ and pqr, respectively. pqr are 3-D vector consisting of rate of roll in the body frame p, rate of pitch in body frame q, and rate of yaw in body frame r.

The output is the commanded moments in 3 axes thus we need the moments of inertia in 3 axes i.e., $I_{xx}, I_{yy}, and\ I_{zz}$. This controller is implemented in lines 112 – 114 in `QuadControl.cpp`.

```
V3F momentInertia(Ixx, Iyy, Izz);
momentCmd = momentInertia * kpPQR * (pqrCmd - pqr);
```

We tune $kpPQR = 90, 92, 20$ in `QuadControlParams.txt`.

Roll-pitch control (scenario 2)

So far, we have designed the body rate controller. We notice that it takes the desired body rates $pqrCmd$ as targets. These values are provided by other controllers. In particular, the commanded roll and pitch rates are given by a roll-pitch controller while a yaw controller provides the yaw rate value.

The roll-pitch controller is another P controller to command the roll and pitch rates in body frame p_c and q_c, respectively. As the roll and pitch decide the linear accelerations in x and y in inertial frame respectively, the goal is to compute p_c and q_c to control $\ddot{x}$ and $\ddot{y}$.

From

$$\ddot{x} = cb^x,$$
$$\ddot{y} = cb^y,$$

where c is the total commanded thrust, $b^x = \boldsymbol{R}_{13}$ and $b^y = \boldsymbol{R}_{23}$, $\boldsymbol{R}$ the 3-by-3 rotation matrix from the body frame to inertial frame, b^x and b^y are chosen as "control knobs". Thus, the P control is given as

$$\dot{b}^x = k_p(b_c^x - b^x),$$
$$\dot{b}^y = k_p(b_c^y - b^y),$$

where k_p is the P control constant to be tuned, b_c^x, b_c^y the commanded rotation matrix entries and $\dot{b}_c^x$ and $\dot{b}_c^y$ are mapped directly to the commanded angular body rates as follows

$$\begin{pmatrix} p_c \\ q_c \end{pmatrix} = \frac{1}{\boldsymbol{R}_{33}} \begin{pmatrix} \boldsymbol{R}_{21} & -\boldsymbol{R}_{11} \\ \boldsymbol{R}_{22} & -\boldsymbol{R}_{12} \end{pmatrix} \begin{pmatrix} \dot{b}_c^x \\ \dot{b}_c^y \end{pmatrix}.$$

The roll-pitch control is implemented in lines 145 – 159 in `QuadControl.cpp` as follows. There are a couple notices in the implementation. First, as $collThrustCmd$ input is the magnitude thus positive but the z-axis points down, a minus is added (in the first line). Second, b_c^x and b_c^y represent the direction of the collective thrust in the inertial frame and thus are constrained by the sine of maximum title angle of the quadrotor (lines 4 - 5).

```
float collThrustCmdNorm = -collThrustCmd / mass;
float b_c_x_target = accelCmd[0] / collThrustCmdNorm;
float b_c_y_target = accelCmd[1] / collThrustCmdNorm;
b_c_x_target = CONSTRAIN(b_c_x_target, -sin(maxTiltAngle), sin(maxTiltAngle));
b_c_y_target = CONSTRAIN(b_c_y_target, -sin(maxTiltAngle), sin(maxTiltAngle));
float b_c_x_dot = kpBank * (b_c_x_target - R(0, 2));
float b_c_y_dot = kpBank * (b_c_y_target - R(1, 2));
float p_c = (R(1, 0) * b_c_x_dot - R(0, 0) * b_c_y_dot) / R(2, 2);
float q_c = (R(1, 1) * b_c_x_dot - R(0, 1) * b_c_y_dot) / R(2, 2);
pqrCmd[0] = p_c;
pqrCmd[1] = q_c;
```

We tune $kpBank = 15$ in `QuadControlParams.txt`.

Figure 2 and Figure 3 presents the results of body rate and roll-pitch controls (scenario 2).

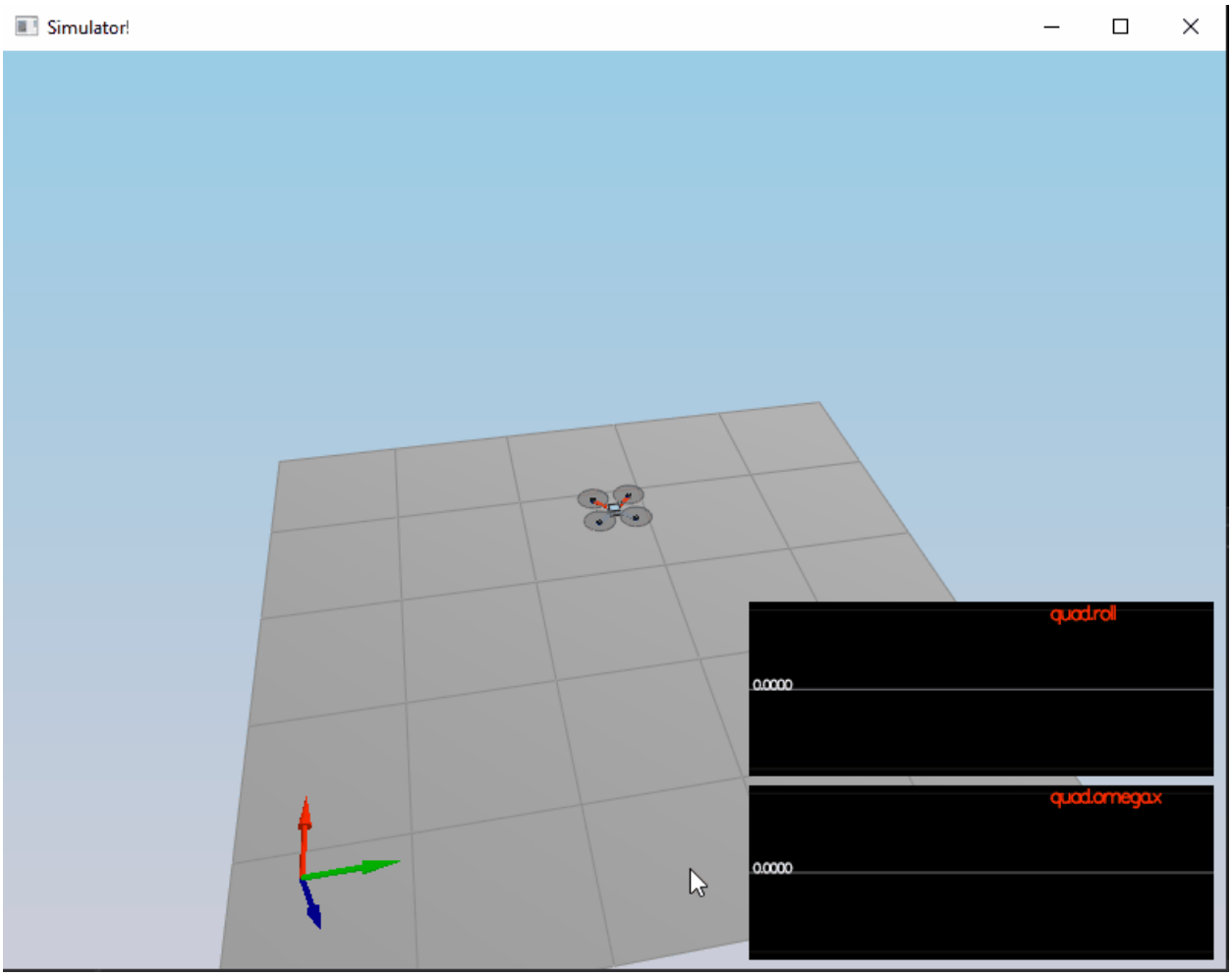

Figure 2: Result for attitude control (scenario 2).

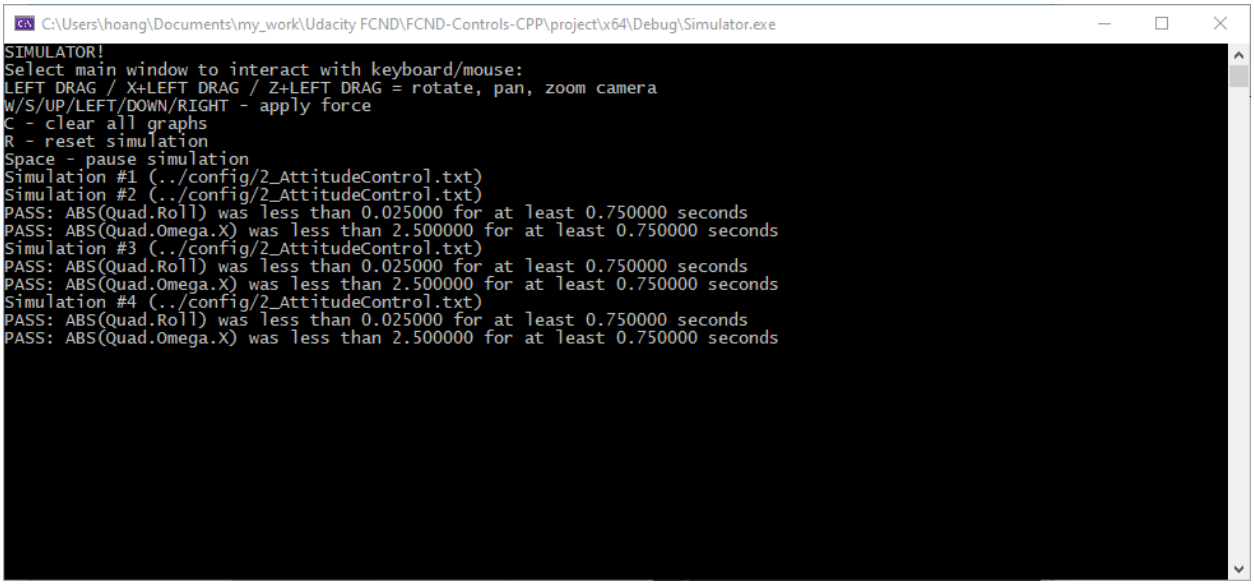

Figure 3: PASS message for attitude control (scenario 2).

Yaw control

Yaw control is decoupled from roll-pitch control. A P controller is used and implemented in the following code block (lines 290 – 299 in `QuadControl.cpp`). The yaw angle must be unwrapped to $(-\pi, \pi]$.

```
// angle normalization
while (yawCmd > F_PI) yawCmd -= 2. * F_PI;
while (yawCmd < -F_PI) yawCmd += 2. * F_PI;

float yaw_error = yawCmd - yaw;
// angle normalization
while (yaw_error > F_PI) yaw_error -= 2. * F_PI;
while (yaw_error < -F_PI) yaw_error += 2. * F_PI;

yawRateCmd = kpYaw * yaw_error;
```

We tune $kpYaw = 3.5$ in `QuadControlParams.txt`.

Lateral position control

The lateral controller can use either a cascaded P controller or a PD controller. The cascaded P controller is implemented in the code block below (lines 256 – 268 in `QuadControl.cpp`). One advantage of this is the ability to constrain the desired velocity which should be smaller than $maxSpeedXY$ in this case. Besides, the acceleration command is constrained by $maxAccelXY$ and feed-forward velocity $velCmd$ and acceleration $accelCmd = accelCmdFF$ are also implemented.

```
V3F velocity_cmd = kpPosXY * (posCmd - pos) + velCmd;
velocity_cmd.z = 0;

// Limit speed
if (velocity_cmd.mag() > maxSpeedXY)
    velocity_cmd = velocity_cmd.norm() * maxSpeedXY;

accelCmd = accelCmd + kpVelXY * (velocity_cmd - vel);
accelCmd.z = 0;

// Limit acceleration
if (accelCmd.mag() > maxAccelXY)
    accelCmd = accelCmd.norm() * maxAccelXY;
```

In (Lupashin), it is shown that for a "critically damped" system, $kpVelXY/kpPosXY = 4$. We tune $kpVelXY = 13$ and $kpPosXY = 4$.

Altitude control

The altitude z is a 2nd order control through vertical acceleration in inertial frame $\ddot{z}$. The $\ddot{z}$ can be converted to collective thrust c by

$$\ddot{z} = cb^z + g,$$

where $b^z = \boldsymbol{R}_{33}$, g the gravity.

Again, the 2nd order control can be implemented by a cascaded P controller or a PD controller. The cascaded P controller is implemented in the code block below (lines 193 – 202 in `QuadControl.cpp`).

One advantage of this is the ability to constrain the desired velocity which is in range $[-maxAscentRate,\ maxDescentRate]$ in this case.

```
// Cascaded P control
float hdot_cmd = kpPosZ * (posZCmd - posZ) + velZCmd;

// Limit the ascent/descent rate
hdot_cmd = CONSTRAIN(hdot_cmd, -maxAscentRate, maxDescentRate);

float acceleration_cmd = accelZCmd + kpVelZ * (hdot_cmd - velZ);

thrust = mass * (acceleration_cmd - CONST_GRAVITY) / R(2, 2);

thrust = -1.0 * thrust;
```

In (Lupashin), it is shown that for a "critically damped" system, $kpVelZ/kpPosZ = 4$. We simply choose $kpVelZ = 4$ and $kpPosZ = 1$.

Figure 4 and Figure 5 presents the results of position control with cascaded controllers (scenario 3).

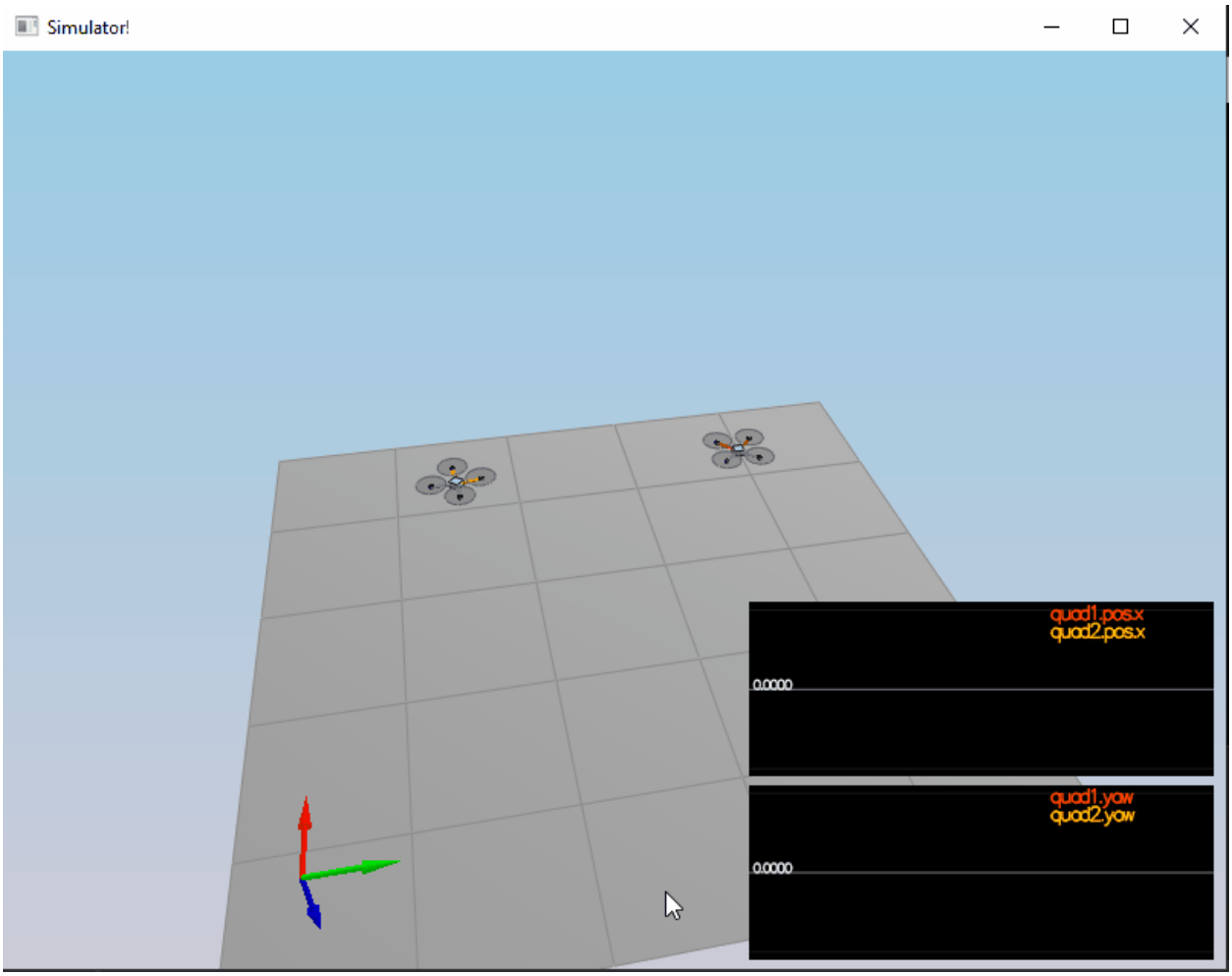

Figure 4: Result for position control with cascaded controllers (scenario 3).

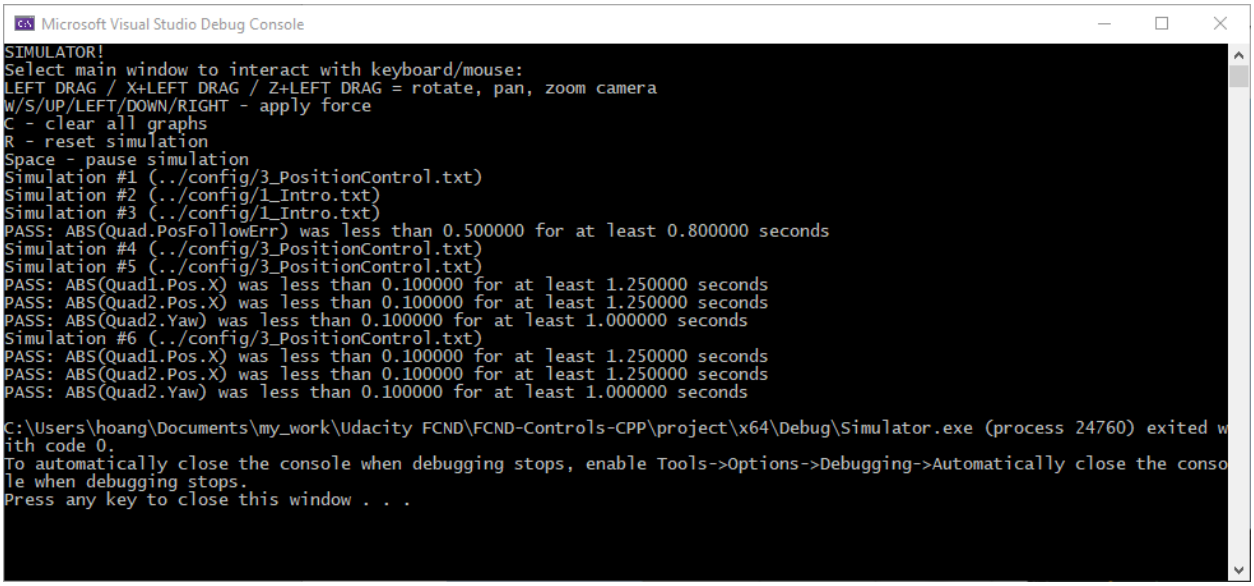

Figure 5: PASS message for position control with cascaded controllers (scenario 3).

On the other hand, we also implemented the full PID controller for altitude control (lines 207 – 218 in `QuadControl.cpp`) because the integral part can help with the different-mass vehicle in Section Non-idealities and robustness (scenario 4).

```
// PID control
float error = posZCmd - posZ;
float error_dot = velZCmd - velZ;

integratedAltitudeError += error * dt;

float acceleration_cmd = kpPosZ * error + kpVelZ * error_dot + KiPosZ *
integratedAltitudeError + accelZCmd;

float a = (acceleration_cmd - CONST_GRAVITY) / R(2, 2);

thrust = mass * CONSTRAIN(a, -maxAscentRate / dt, maxAscentRate / dt);

thrust = -1.0 * thrust;
```

We tune $kpPosZ = 25$, $kpVelZ = 14$, and $kiPosZ = 42$ in `QuadControlParams.txt`.

Figure 6 and Figure 7 show the results for position control (scenario 3) with PID altitude control. Note that the lateral control still uses cascaded P controller.

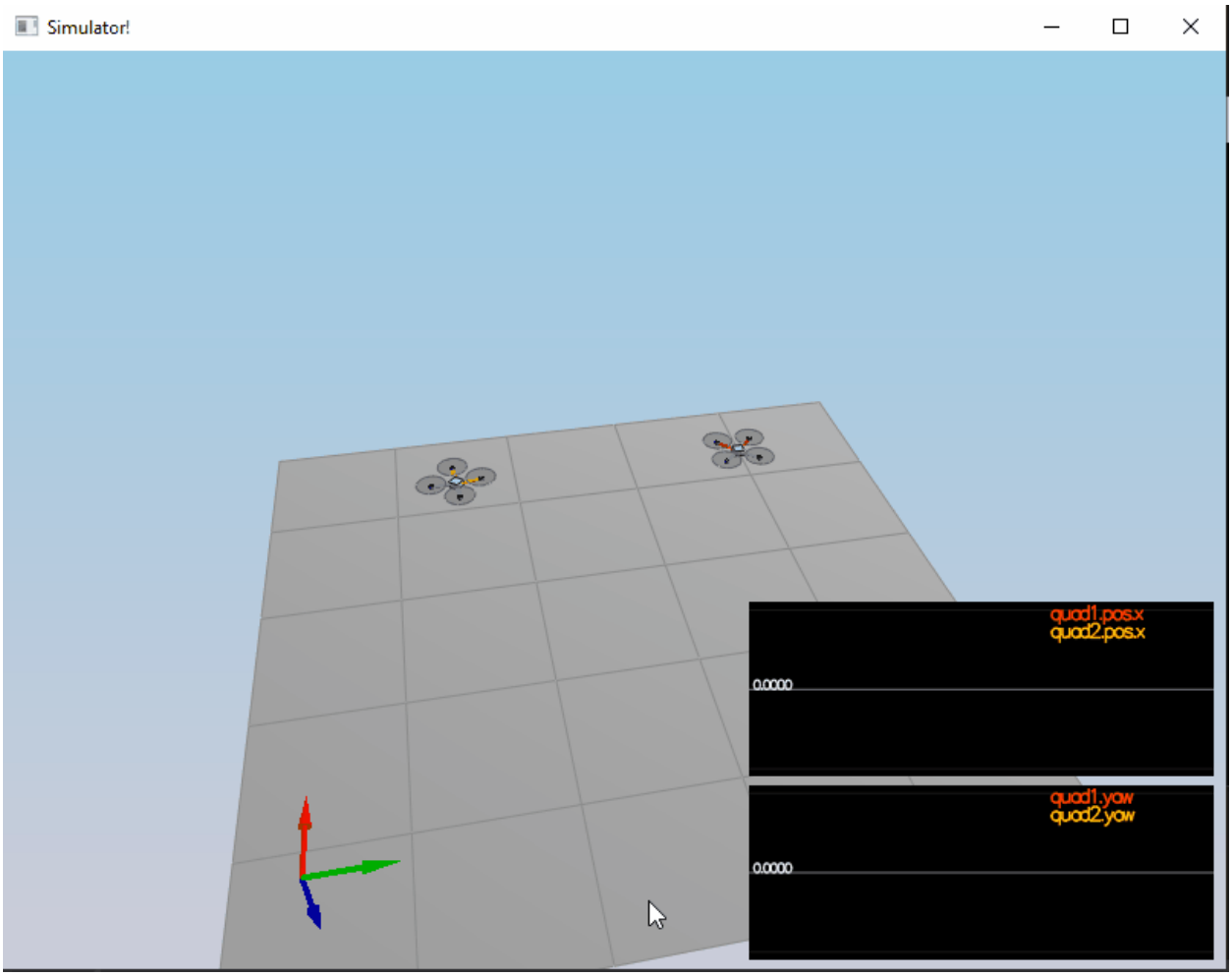

Figure 6: Result for position control with PID altitude control (scenario 3).

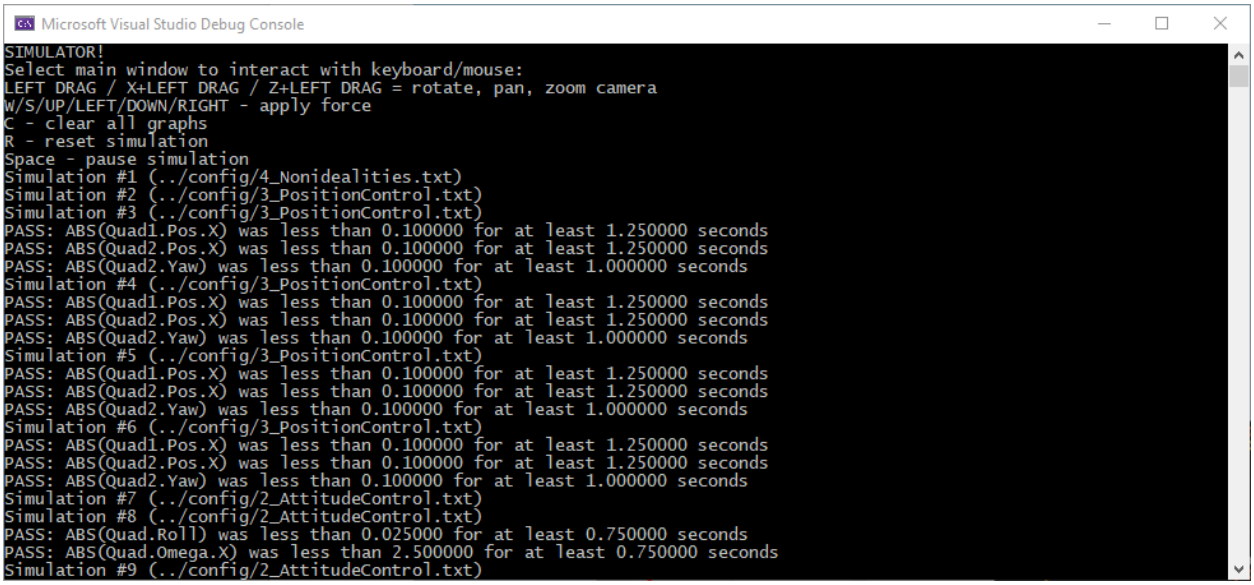

Figure 7: PASS message for position control with PID altitude control (scenario 3).

Non-idealities and robustness (scenario 4)

In this part, we will explore some of the non-idealities and robustness of a controller. For this simulation, we will use Scenario 4. This is a configuration with 3 quads that are all are trying to move one meter forward. However, this time, these quads are all a bit different:

- The green quad has its center of mass shifted back,
- The orange vehicle is an ideal quad,
- The red vehicle is heavier than usual.

With proper finetuning and integral part in the altitude control, our control is robust to non-idealities as shown in Figure 8 and Figure 9.

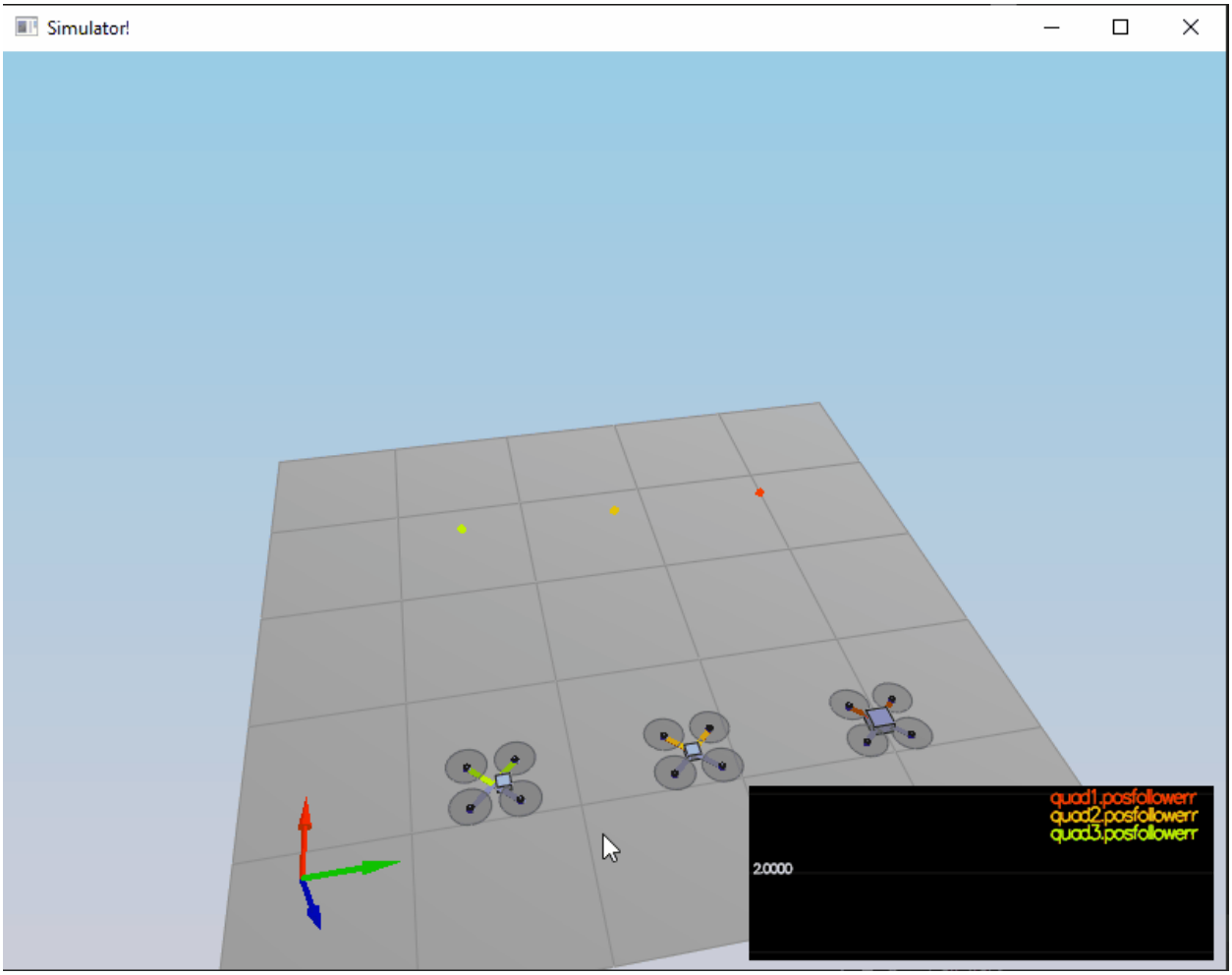

Figure 8: Result for non-idealities and robustness of a controller (scenario 4).

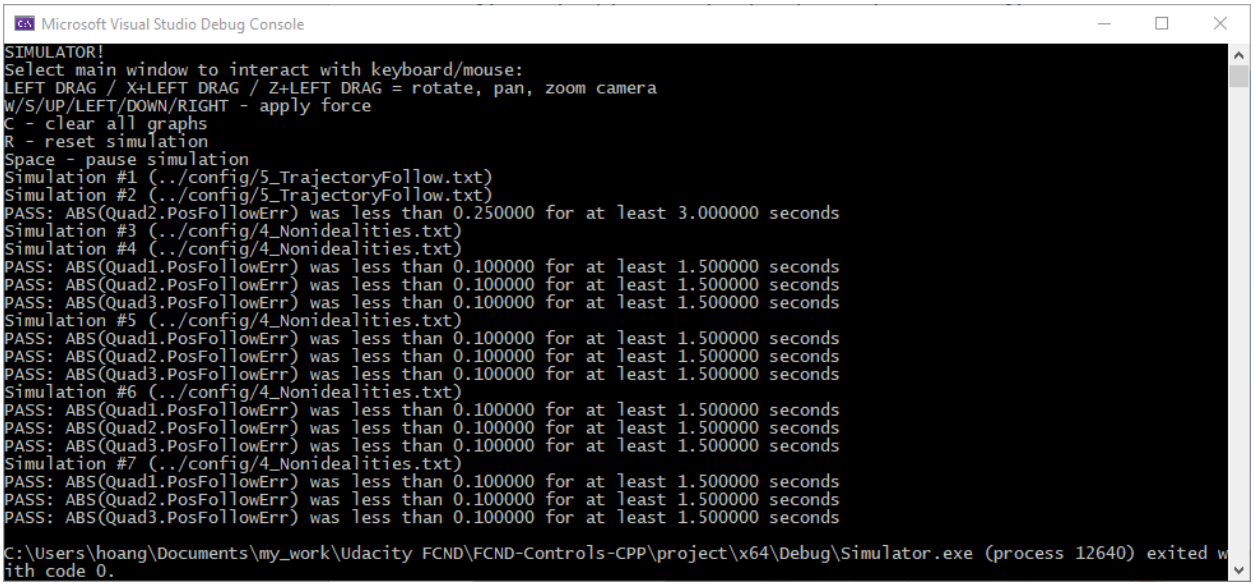

Figure 9: PASS message for non-idealities and robustness of a controller (scenario 4).

Tracking trajectories (scenario 5)

In this part, we test our controller with a more challenging trajectory e.g., a figure 8 to evaluate if the vehicle can follow well. We use scenario 5 having two quadcopters:

- The orange one is following `traj/FigureEight.txt`
- The yellow one is following `traj/FigureEightFF.txt` - for now this is the same trajectory.

Figure 10 and Figure 11 show the result for the figure 8 trajectory following. We observed that the only the yellow vehicle follows well the figure 8. Though both have the same controller and follow the same trajectory, we will study the reason in the next section.

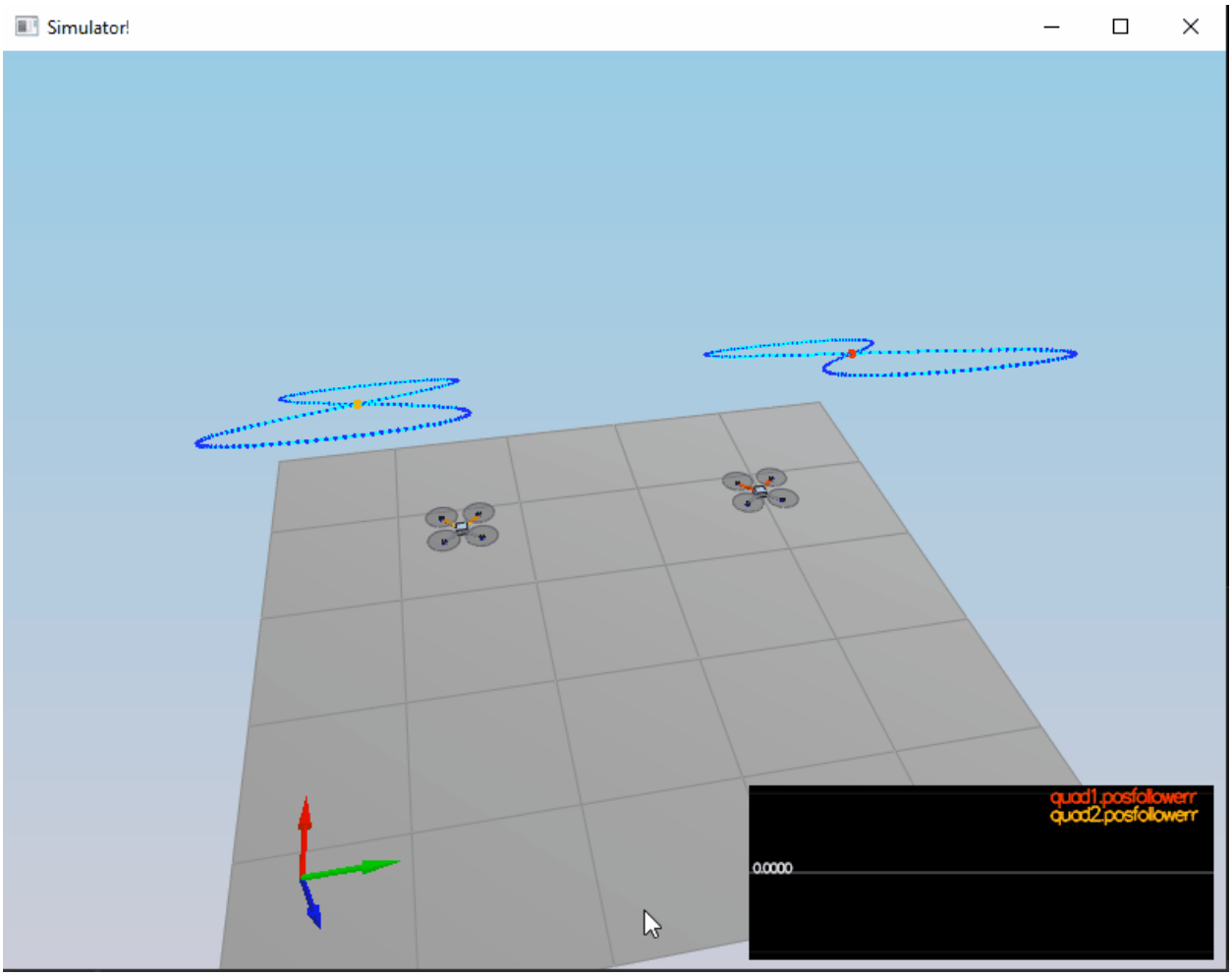

Figure 10: Result for figure 8 trajectory following (scenario 5).

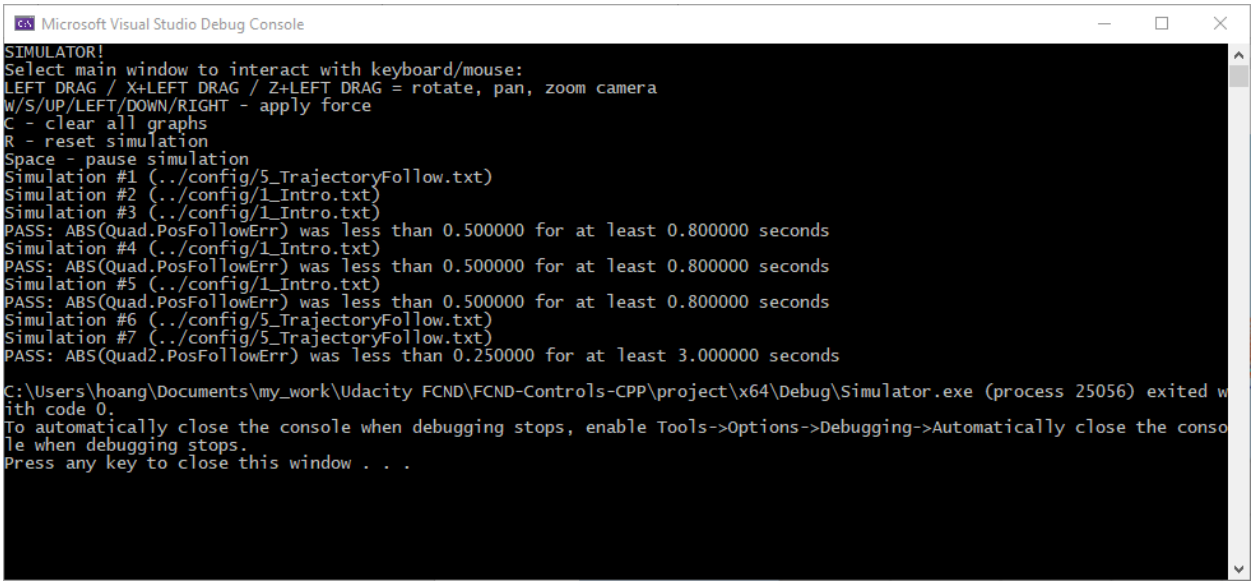

Figure 11: PASS message for figure 8 trajectory following (scenario 5).

Extra Challenge 1

In this section, we investigate the reason why the orange vehicle cannot follow the figure 8 as good as the yellow vehicle. We found that though `FigureEight.txt` (used by the orange) and `FigureEightFF.txt` (used by the yellow) are the same trajectory, the later includes also velocity information that activates the feedforward velocity in the yellow vehicle's controller.

To use feedforward velocity in the orange vehicle's controller, we regenerate the `FigureEight.txt` with velocity information by updating `MakePeriodicTrajectory.py` (lines 29 – 35) as follows.

```
vx = (x - px) / timestep
vy = (y - py) / timestep
vz = (z - pz) / timestep

px = x
py = y
pz = z
```

Figure 12 shows that now the 2 vehicles can follow the figure 8 trajectories.

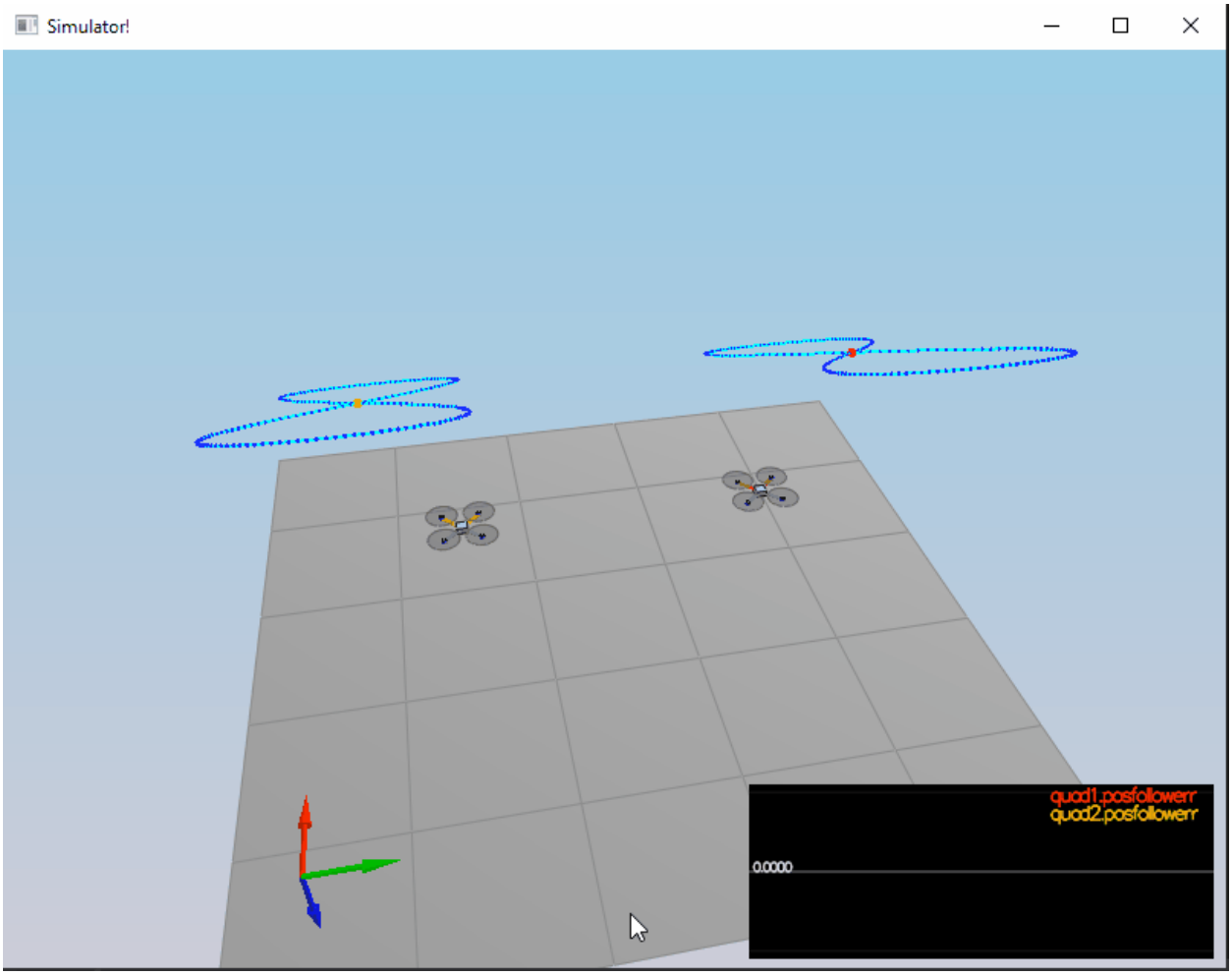

Figure 12: Result for extra challenge 1 (scenario 5).

Extra Challenge 2

Q: For flying a trajectory, is there a way to provide even more information for even better tracking?

A: Acceleration information can be used to activate acceleration feedforward control.

Q: How about trying to fly this trajectory as quickly as possible (but within following threshold)!

A: It may be done by planning an acceleration profile as high as possible but within the vehicle constraints and tunning properly the damping ratio.

References

Lupashin, S. (n.d.). *Double Integrator Control: Cascaded P Controller Gains vs Damping Ratio.* Fotokite.

Schoellig, A. P., Wiltsche, C., & D'Andrea, R. (2012). Feed-Forward Parameter Identification for Precise Periodic Quadrocopter Motions. *American Control Conference.* Montréal.