FCND Control – Udacity Flying Car

Tiansi Dong

1 Body rate and roll/pitch control (scenario 2)

To control the body rates, we need to decide the total force generated by the four motors, desired moments to adjust roll and pitch, and related parameters.

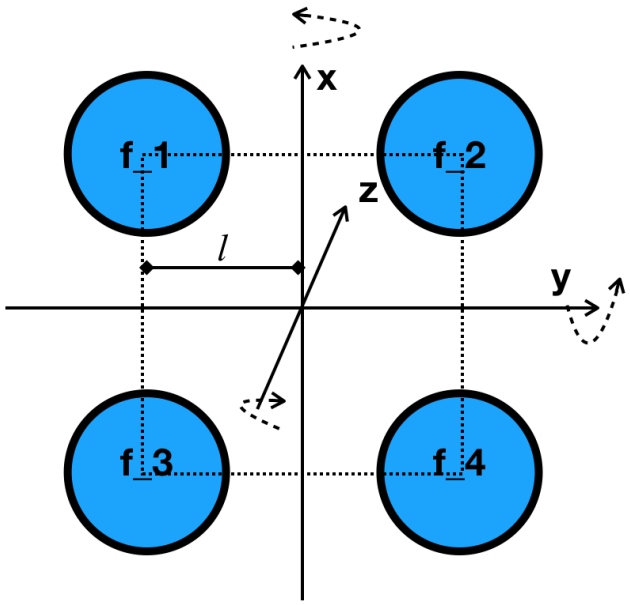

Fig. 1. Four motors

1.1 Generate motor command

Based on the motor configuration in Figure 1, we have four formula as follow.

$$f_1 + f_2 + f_3 + f_4 = F_{total} \tag{1}$$
$$f_1 * l + f_3 * l - f_2 * l - f_4 * l = M_x \tag{2}$$
$$f_1 * l + f_2 * l - f_3 * l - f_4 * l = M_y \tag{3}$$
$$f_1 * \kappa - f_2 * \kappa - f_3 * \kappa + f_4 * \kappa = -M_z \tag{4}$$

we have,

$$f_1 + f_2 + f_3 + f_4 = F_{total} \tag{5}$$
$$f_1 + f_3 - f_2 - f_4 = M_x/l \tag{6}$$
$$f_1 + f_2 - f_3 - f_4 = M_y/l \tag{7}$$
$$f_1 - f_2 - f_3 + f_4 = -M_z/\kappa \tag{8}$$

(5) + (6) + (7) + (8)

$$4f_1 = F_{total} + M_x/l + M_y/l + M_z/\kappa \tag{9}$$

(5) - (6) + (7) - (8)

$$4f_2 = F_{total} - M_x/l + M_y/l - M_z/\kappa \tag{10}$$

(5) + (6) - (7) - (8)

$$4f_3 = F_{total} + M_x/l - M_y/l + M_z/\kappa \tag{11}$$

(5) - (6) - (7) + (8)

$$4f_4 = F_{total} - M_x/l - M_y/l - M_z/\kappa \tag{12}$$

The `GenerateMotorCommands()` function is implemented as follows.

```
float Mx = momentCmd.x;
float My = momentCmd.y;
float Mz = -momentCmd.z;
float Ftotal = collThrustCmd ;

float l = L/sqrt(2);

float f1 = (Mx/l + My/l - Mz/kappa + Ftotal)/4;
float f2 = (-Mx/l + My/l + Mz/kappa + Ftotal)/4;
float f3 = (Mx/l - My/l + Mz/kappa + Ftotal)/4;
float f4 = (-Mx/l - My/l - Mz/kappa + Ftotal)/4;

cmd.desiredThrustsN[0] = f1; // front left
cmd.desiredThrustsN[1] = f2; // front right
cmd.desiredThrustsN[2] = f3; // rear left
cmd.desiredThrustsN[3] = f4; // rear right
```

1.2 Body rate control

We need to compute the desired three-dimensional moment `momentCmd` to fulfil the transform from the current body rates `pqr` to the desired body rates `pqrCmd`. The mathematical formula is

$$\Delta_{pqr} = pqrCmd - pqr$$
$$\boldsymbol{M} = I \cdot kp\Delta_{pqr}$$

We implement the code in the function `BodyRateControl()`

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

1.3 Tuning parameters: `kpPQR`

The first two values of `kpPQR` in `QuadControlParams.txt` is tuned to $(100, 73)$ to meet the criteria, as illustrated in Fig 2.

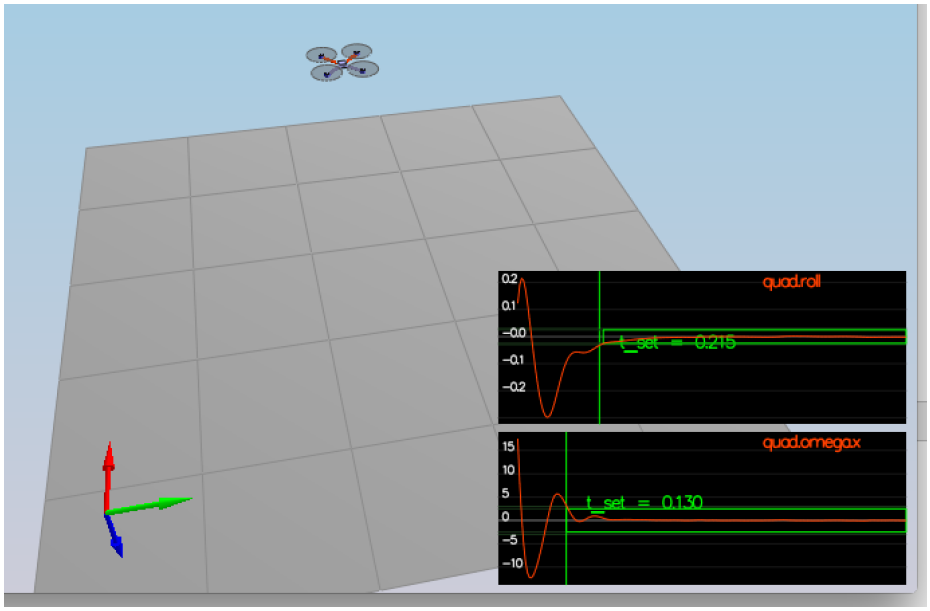

Fig. 2. Body rate and roll/pitch control

2 Position/velocity and yaw angle control (scenario 3)

In this task, we need to control the position, the altitude, and the yaw of the quad. Two identical quads are created in the testing environment: one offset from its target point (but initialised with yaw = 0) and the other with a yaw offset of 45 degrees.

We need to implement: (1) a lateral position control `LateralPositionControl()` with corresponding parameters `kpPosXY` and `kpVelXY`; (2) a altitude control `AltitudeControl()` with parameters `kpPosZ` and `kpVelZ`; (3) a yaw control `YawControl()` with `kpYaw` and the third paramter of `kpPQR`.

2.1 Lateral position control

Given current information about current position, velocity, and desired position, velocity, and acceleration, we need to compute the current acceleration. All these information is in 3-dimensional space.

The current acceleration courses spatial changes, and results in the desired position, velocity, and acceleration after the given temporal duration. In lateral control, we only consider changes in the x- and y-axises, which can be written as follows.

$$\Delta_X = [\Delta_{pos_x}, \Delta_{vel_x}, accelCmd_x]$$
$$\Delta_Y = [\Delta_{pos_y}, \Delta_{vel_y}, accelCmd_y]$$

The desired acceleration in the x-direction is computed by the sum of these changes in the x-direction with a multiplied gain as follow.

$$accelCmd_x = kp_{Pos_x} * \Delta_{pos_x} + kp_{Vel_x} * \Delta_{vel_x} + accelFF_x$$

The desired acceleration in the y-direction is computed in the similar way.

$$accelCmd_y = kp_{Pos_y} * \Delta_{pos_y} + kp_{Vel_y} * \Delta_{vel_y} + accelFF_y$$

We need to constrain the range of the velocity. The implementation is listed as below.

```
V3F pd = V3F(kpPosXY, kpVelXY, 1.f);
V3F deltaX = V3F((posCmd-pos)[0],
                 CONSTRAIN((velCmd-vel)[0],
                           -maxSpeedXY,
                           maxSpeedXY),
                 accelCmd[0]);
V3F deltaY = V3F((posCmd-pos)[1],
                  CONSTRAIN((velCmd-vel)[1],
                           -maxSpeedXY,
                           maxSpeedXY),
                  accelCmd[1]);
accelCmd = V3F(pd.dot(deltaX), pd.dot(deltaY), 0.f);
```

2.2 Altitude control

The altitude control is computed similarly. The desired acceleration is the sum of changes of position and velocity in the z-direction weighted by their gains.

$$accelCmd_z = kp_{Pos_z} * \Delta_{pos_z} + kp_{Vel_z} * \Delta_{vel_z} + accelFF_z$$

which can be implemented as follow.

```
V3F zk = V3F(kpPosZ, kpVelZ, 1);
V3F delta = V3F(posZCmd-posZ, (velZCmd-velZ), accelZCmd);
float u_bar_1 = zk.dot(delta);
```

Different from the lateral control, we must consider the gravity in the z-direction. The direction of the thrust in the body frame, it must be transformed in the global frame. The thrust projected to the global z-direction minus the force of gravity shall generate the acceleration `accelCmd_z`.

$$f_{z_\perp} - mg = ma$$

The final thrust is implemented as follow.

```
thrust = -mass*(u_bar_1-CONST_GRAVITY)/R(2,2);
```

2.3 Yaw control

The control of yaw is quite straight forward. It is the change of the yaw multiplied by its gain. After tuning all the parameters, we got the desired result as illustrated in Fig 3.

```
yawRateCmd = kpYaw * (yawCmd - yaw);
```

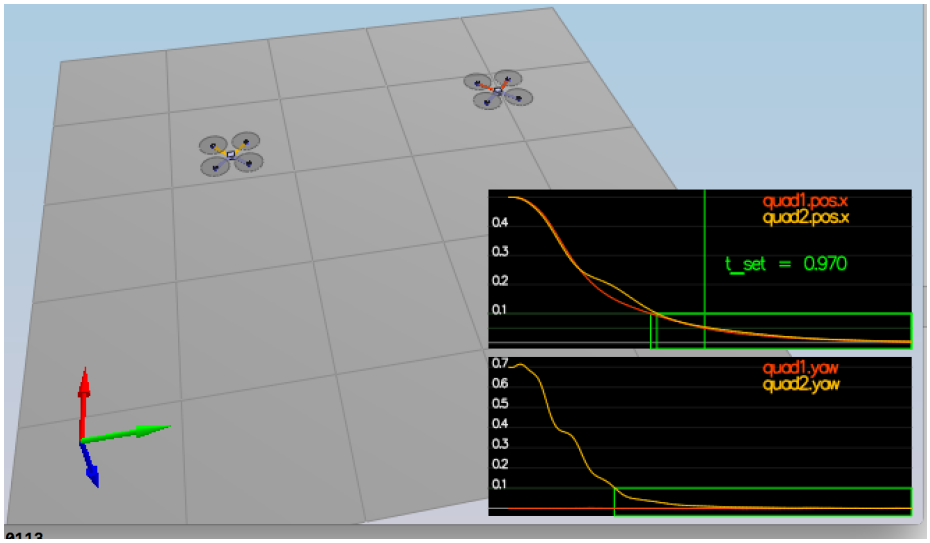

Fig. 3. Position/velocity and yaw angle control

3 Non-idealities and robustness (scenario 4)

In this task, there are 3 quads which are expected to move one meter forward. These quads are designed differently: The green one has its center of mass shifted back; the orange one is ideal; the red one is much heavier than usual. We need to update the parameters, and edit `AltitudeControl()` to add basic integral control to help with the different-mass vehicle.

The position different during a unit temporal duration if $\Delta_{pos_z} * \delta_t$, which is accumulated, and adds an additional acceleration. An implementation is as follow.

```
this->integratedAltitudeError += (posZCmd-posZ)*dt;
u_bar_1 += this->integratedAltitudeError * KiPosZ;
```

After tuning, we got satisfying performances of three drones as shown in Fig 4.

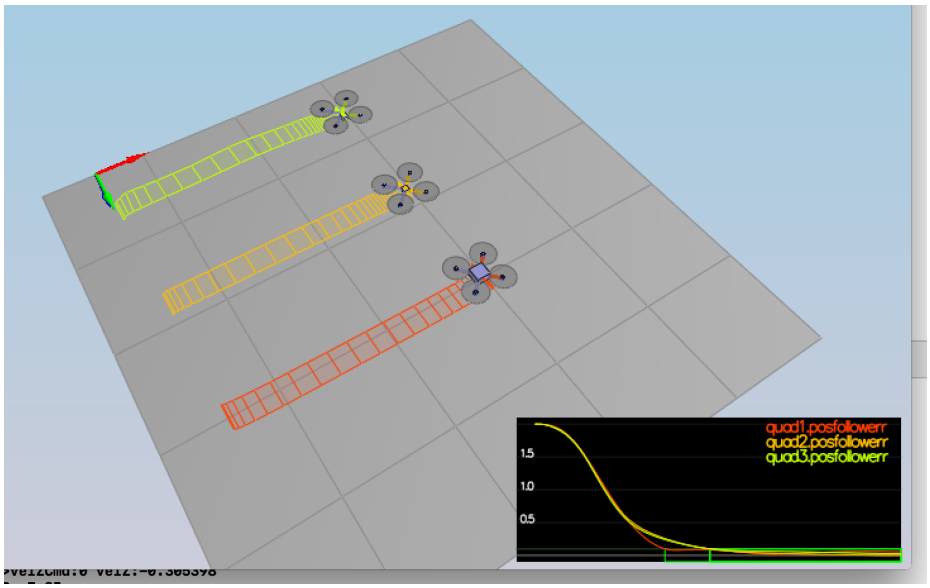

Fig. 4. Non-idealities

4 Tracking trajectories: scenario 5

We also tried the task of scenario 5. Two drones with different physical features are trying to follow trajectories, one is heavier than the other. After the tuning of some parameters, the light drone followed the trajectory better than the heavy one, and met the required criteria, as shown in Fig 5.

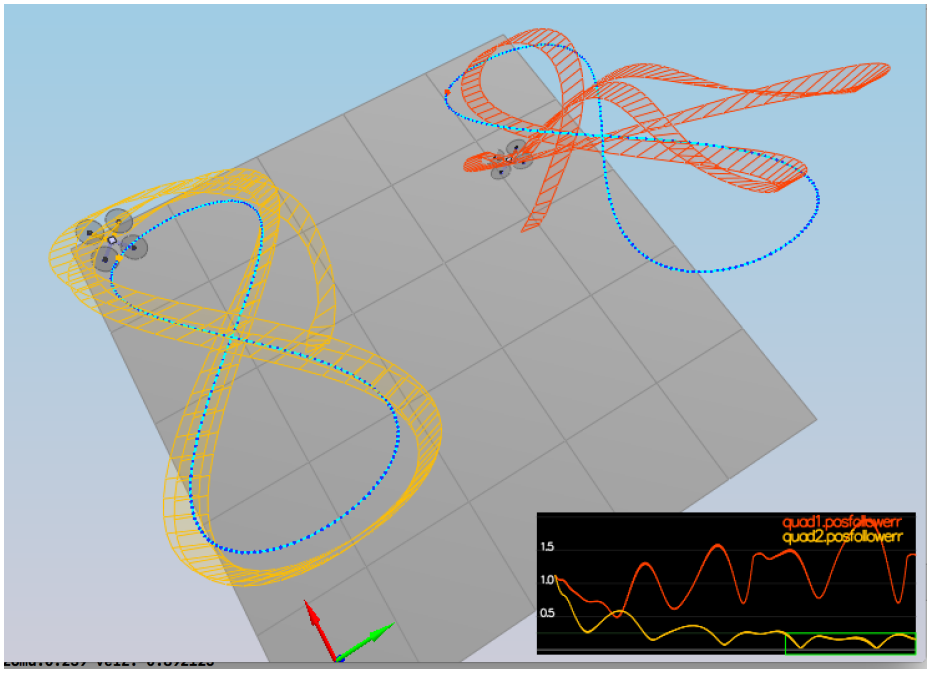

Fig. 5. Tracking trajectories