Cuadrirotor Control Implementation

Irving Vasquez

May 22, 2018

1 Control

In this project I have completed the implementation of a cascade controller for a cuadrotor. I was thinking that a good application for the proyect is building inspection. So, I modified the trajectory generator and I added the yaw command for ech point. See figure 1 . Feel free to watch my video: `https://youtu.be/Qpp3OU2DmAo`. My github is `https://github.com/irvingvasquez/FCND-Controls-CPP`.

The controller diagram is drawn in figure 2. Next, I will describe each part.

1.1 Body Rate

The body rate controller computes the moments on the three axis accordingly to the following:

$$\alpha = \left[\begin{array}{c} k_p^p(p_c - p_a) \\ k_p^q(q_c - q_a) \\ k_p^r(r_c - r_a) \end{array} \right] \tag{1}$$

$$M_c = I\alpha \tag{2}$$

See the implemented code:

```
float u_bar_p = kpPQR.x * (pqrCmd.x - pqr.x);
float u_bar_q = kpPQR.y * (pqrCmd.y - pqr.y);
float u_bar_r = kpPQR.z * (pqrCmd.z - pqr.z);

momentCmd.x = Ixx * u_bar_p;
momentCmd.y = Iyy * u_bar_q;
momentCmd.z = Izz * u_bar_r;
```

1.2 Roll/pitch control

This control calculates the needed roll and pitch rates to reach a commanded pitch and roll values.

First, the rotation matrix angles are computed:

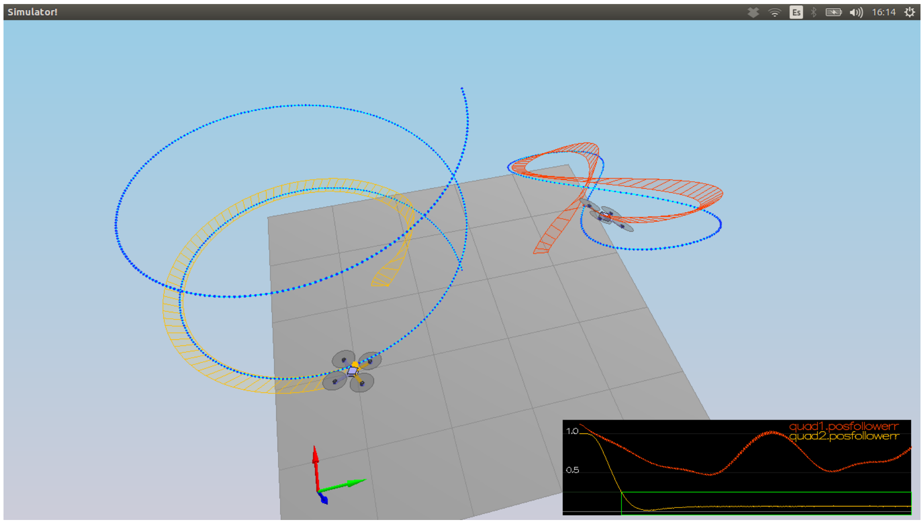

Figure 1: Helix trajectory for building inspection.

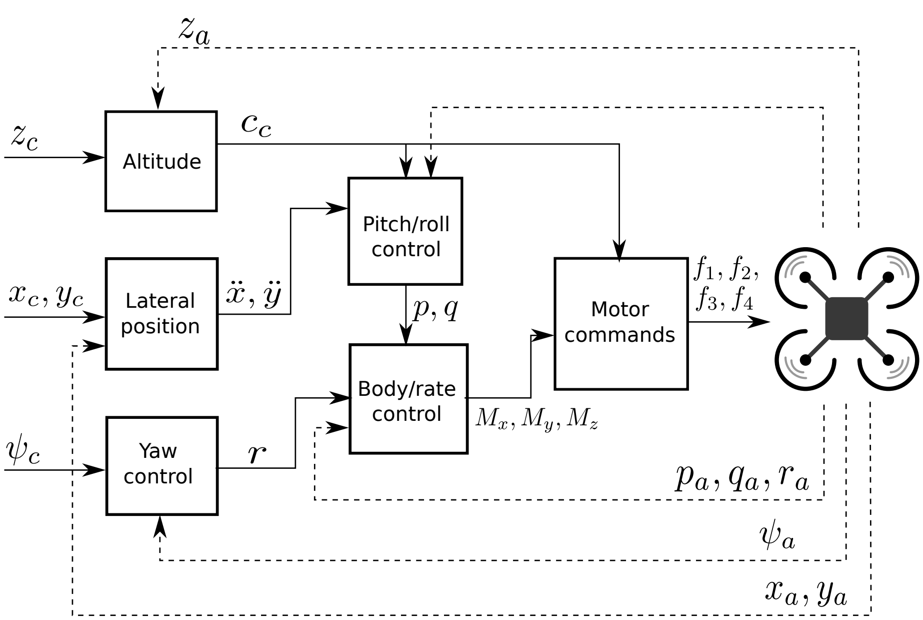

Figure 2: Control diagram.

$$b_c^x = \frac{m\ddot{x}}{-c_c} \tag{3}$$

$$b_c^y = \frac{m\ddot{y}}{-c_c} \tag{4}$$

Then, the commanded euler rates are computed:

$$\dot{b}_u^x = k_p(b_c^x - b_a^x) \tag{5}$$

$$\dot{b}_u^y = k_p(b_c^y - b_a^y) \tag{6}$$

Finally, the rotations in the body-frame are computed

$$\begin{bmatrix} p_c \\ q_c \end{bmatrix} = \frac{1}{R_{33}} \begin{bmatrix} R_{21} & -R_{11} \\ R_{22} & -R_{12} \end{bmatrix} \begin{bmatrix} \dot{b}_u^x \\ \dot{b}_u^y \end{bmatrix} \tag{7}$$

See the implementation code

```
float c_c = -collThrustCmd / mass;

float b_c_x = accelCmd.x / c_c;
float b_c_y = accelCmd.y / c_c;

// check max bank angle
if (b_c_x < - maxTiltAngle)
b_c_x = -maxTiltAngle;
else if (b_c_x > maxTiltAngle)
b_c_x = maxTiltAngle;

if (b_c_y < - maxTiltAngle)
b_c_y = -maxTiltAngle;
else if (b_c_y > maxTiltAngle)
b_c_y = maxTiltAngle;

float b_a_x = R(0,2);
float b_a_y = R(1,2);

float b_c_x_dot = kpBank * (b_c_x - b_a_x);
float b_c_y_dot = kpBank * (b_c_y - b_a_y);

//p_c
pqrCmd.x =  1/R(2,2) * (R(1,0)*b_c_x_dot - R(0,0)*b_c_y_dot);
//q_c
pqrCmd.y =  1/R(2,2) * (R(1,1)*b_c_x_dot - R(0,1)*b_c_y_dot);
```

1.3 Altitude controller

The altitude controller adjust the collective thrust in order to reach a commanded altitude (NED).

First a proportional controller calculates the velocity:

$$\dot{z}_u = K_p^z \cdot (z_c - z_a) + \dot{z}_c \tag{8}$$

Then a proportional integral control is used to calculate the desired thrust:

$$\ddot{z}_u = K_p^{\dot{z}} \cdot (\dot{z}_u - \dot{z}_a) + \int e_z dt \tag{9}$$

Finally, the vertical acceleration is converted into force:

$$C_c = \frac{1}{R_{22}}(g - \ddot{z}_u)m \tag{10}$$

Observe the implemented code:

```
float z_error = posZCmd - posZ;
float z_dot_c = kpPosZ * (z_error) + velZCmd;

if(-z_dot_c > maxAscentRate)
z_dot_c = -maxAscentRate;
else
if(z_dot_c > maxDescentRate)
z_dot_c = maxDescentRate;

integratedAltitudeError += z_error * dt;
float z_dot_dot = kpVelZ * (z_dot_c - velZ) + KiPosZ * (integratedAltitudeError) +  accelZCr

// converting acceleration to force
thrust = (CONST_GRAVITY - z_dot_dot) * mass / R(2,2);

// check thrust
if (thrust > maxMotorThrust*4.0)
thrust = maxMotorThrust*4.0;
else if (thrust < minMotorThrust*4.0)
thrust = minMotorThrust*4.0;
```

1.4 Lateral position control

First I calculate the velocity input with a proportional controller.

$$\begin{bmatrix} \dot{x}_u \\ \dot{y}_u \end{bmatrix} = k_p^{xy} \left(\begin{bmatrix} x_c \\ y_c \end{bmatrix} - \begin{bmatrix} x_a \\ y_a \end{bmatrix} \right) + \begin{bmatrix} \dot{x}_a \\ \dot{y}_a \end{bmatrix} \tag{11}$$

Next, the acceleration input is computed

$$\begin{bmatrix} \ddot{x}_u \\ \ddot{y}_u \end{bmatrix} = k_p^{accel} \left(\begin{bmatrix} \dot{x}_u \\ \dot{y}_u \end{bmatrix} - \begin{bmatrix} \dot{x}_a \\ \dot{y}_a \end{bmatrix} \right) + \begin{bmatrix} \ddot{x}_c \\ \ddot{y}_c \end{bmatrix} \tag{12}$$

See the following C++ code:

```
// P control for position
float u_vel_x = kpPosXY * (posCmd.x - pos.x) + velCmd.x;
float u_vel_y = kpPosXY * (posCmd.y - pos.y) + velCmd.y;

if(u_vel_x > maxSpeedXY)
u_vel_x = maxSpeedXY;
if(u_vel_y > maxSpeedXY)
u_vel_y = maxSpeedXY;
if(u_vel_x < -maxSpeedXY)
u_vel_x = -maxSpeedXY;
if(u_vel_y < -maxSpeedXY)
u_vel_y = -maxSpeedXY;

// P with Feedforward for velocity
// + Feedforward was not added since it is already in the accelCmd
float u_acc_x = kpVelXY * (u_vel_x - vel.x);
float u_acc_y = kpVelXY * (u_vel_y - vel.y);

accelCmd.x += u_acc_x;
accelCmd.y += u_acc_y;

if(accelCmd.x > maxAccelXY)
accelCmd.x = maxAccelXY;
else if (accelCmd.x < -maxAccelXY)
accelCmd.x = -maxAccelXY;

if(accelCmd.y > maxAccelXY)
accelCmd.y = maxAccelXY;
else if (accelCmd.y < -maxAccelXY)
accelCmd.y = -maxAccelXY;
```

1.5 Yaw Control

I have implemented a p control where the control input is computed with:

$$\dot{\psi} = k_p^{\psi} \cdot (\psi_c - \psi_a) \tag{13}$$

See the following code where I validate the angle ranges:

```
  // Convert to [-PI,PI]
  float yawCmd_range = fmod(yawCmd, 6.28318); // 2PI
  if (yawCmd_range>M_PI)
```

```
yawCmd_range = yawCmd_range - 2*M_PI;

float yaw_range = fmod(yaw, 6.28318);
if(yaw_range > M_PI)
yaw_range = yaw_range - 2*M_PI;

// If we are working in quadrants 3 and 4 we will move to positive representation
if(yawCmd_range > M_PI/2 && yaw_range < -M_PI/2)
yaw_range = yaw_range + 2*M_PI;
else if(yaw_range > M_PI/2 && yawCmd_range < -M_PI/2)
yawCmd_range = yawCmd_range + 2*M_PI;


yawRateCmd = kpYaw * (yawCmd_range - yaw_range);
```

1.6 Motor Commands

I compute the motor commands solving the following equation:

$$\begin{bmatrix} 1 & 1 & 1 & 1 \\ 1 & -1 & 1 & -1 \\ 1 & 1 & -1 & -1 \\ 1 & -1 & -1 & 1 \end{bmatrix} \begin{bmatrix} f_1 \\ f_2 \\ f_3 \\ f_4 \end{bmatrix} = \begin{bmatrix} \bar{a} \\ \bar{b} \\ \bar{c} \\ \bar{d} \end{bmatrix} \tag{14}$$

where

$$\bar{a} = c_c \tag{15}$$

$$\bar{b} = \frac{M_x\sqrt{2}}{L} \tag{16}$$

$$\bar{c} = \frac{M_y\sqrt{2}}{L} \tag{17}$$

$$\bar{d} = \frac{M_x}{\kappa} \tag{18}$$

Remeber that $\kappa = \tau/F$

2 Trajectory generation

I am thinking in a tower inpection application so I have extended the Helix generator. I have included the velocity for each point and the yaw orientation.

See my code:

```
with open('HelixVelYaw.txt', 'w') as the_file:
t=0;
while t <= maxtime:
x = math.sin(t * 2 * math.pi / period) * radius;
y = math.cos(t * 2 * math.pi / period) * radius;

t_mas_1 = t+timestep;
x_next = math.sin(t_mas_1 * 2 * math.pi / period) * radius;
y_next = math.cos(t_mas_1 * 2 * math.pi / period) * radius;
z_next = z - 0.005
vx = (x_next - x)/timestep;
vy = (y_next - y)/timestep;
vz = (z_next - z)/timestep;

roll = 0.0;
pitch = 0.0;
yaw = -t * 2 * math.pi / period;

the_file.write(fmt(t) + "," + fmt(x) + "," + fmt(y) + "," + fmt(z));
the_file.write("," + fmt(vx) + "," + fmt(vy) + "," + fmt(vz));
the_file.write("," + fmt(yaw) + "," + fmt(pitch) + "," + fmt(roll) + "\n");
t += timestep;
z -= 0.005
```