Control of a 3D Quadrotor

Frederick Chyan

April 29, 2018

1 Implemented Controller

The controller architecture is shown in Fig 1. Implementation of different control blocks are discussed.

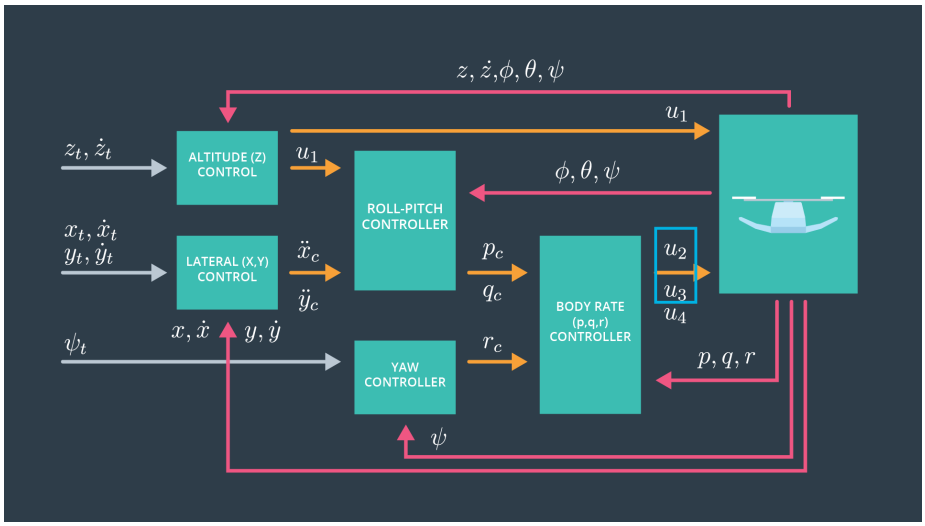

Figure 1: Cascade control loops of decoupled linear systems.

1.1 Body Rate Control

Body rate is the rotational rates (p, q, r) of the three axes in the **body frame**. The body rate controller is a simple proportional controller that takes in the desired body rate $pqrCmd$ and current body rate pqr. The desired rotational acceleration is obtained by multiplying the error $pqrCmd - pqr$ by the proportional gain $kpPQR$. Then multiplying them with the moments of inertia (Ixx, Iyy, Izz) to get the desired moments for each of the 3 axes.

1.2 Roll Pitch Control

The roll pitch controller takes collective thrust command, desired acceleration and attitude as input and output desired roll and pitch rate. This is the most interesting controller. Since the thrust (which controls/maintains z) has already

been decided, the job of this controller is to adjust the orientation/attitude of the drone to make sure it is moving in the correct x, y directions. The notation b^x correspond to rotation matrix R13 which can be described as amount of z in body frame that gets map to x in the world frame. Essentially, it means by tilting the drone in certain angle, a portion of the force/thrust that was used to move upwards(z) will now be used to move sideways (x, y).

1. calculated the commanded rotation matrix element b_c^x and b_c^y, this is accelCmd.x(y) / c, where c is -thrust/mass. 2. constrain the tilt angle. 3. calculate the proportional error by subtracting current attitude b^x and b^y. 4. Apply the bank angle proportional constant for the P controller. 5. Calculate the desired roll and pitch rate in the body frame using the formula $p_c = 1/R33 * (R21 * \dot{b}_c^x - R11 * \dot{b}_c^y)$ and $q_c = 1/R33 * (R22 * \dot{b}_c^x - R12 * \dot{b}_c^y)$

1.3 Altitude Control (Python)

The complete altitude controller takes the desired and current altitude along with its derivatives (velocity and acceleration) and attitude as input and output the commanded thrust u1 that is the collective thrust for all four rotors.

1. calculate the error in altitude, which is simply commanded altitude minus current altitude, multiply that by the proportional gain for altitude. Add commanded vertical velocity, then restrict the speed. This is the desired commanded velocity, call it altitude_dot_cmd. 2. Calculate the difference between altitude_dot_cmd and current velocity and multiply that by the derivative gain k_d_z. Add feed forward acceleration on top of it. Call this term u1bar, it's the desired acceleration in the z direction. 3. However, this is not enough, since the drone might be tilted. Calculate the real body acceleration needed in the z direction by dividing u1bar by b_z, where b_z is the portion of z in the body frame that actually remains in z in the world frame. Multiply the acceleration by the drone mass to get the collective thrust required.

1.4 Altitude Control (C++)

The C++ version of the altitude controller is the same except it also handles gravity and accumulated error. The gravity term is added to the acceleration command in the altitude controller, so that the generated collective thrust includes the gravity. To handle non-idealities such as different body mass, we introduce the integral controller. This is done by accumulating the position error in every controller loop (accumulated error += current_error * delta_time). Then add this term to the generated command acceleration by multiplying the accumulated error with the integral gain.

1.5 Lateral Position Control

The lateral position controller is relatively simple PD controller since it does not involve coordinate transformation and the calculations are carried out in world frame. It take the commanded and measured current position, velocity and feed forward acceleration as input and output commanded acceleration. It does so by calculating the controller commanded velocity (velCmd = kpPosXY * (posCmd - pos) + velCmd) first and constraining it (velCmd = velCmd * maxSpeedXY / velCmdNorm), and then calculated the commanded acceleration (accelCmd

= accelCmd + kpVelXY * (velCmd - vel)) and then constraining it (accelCmd = accelCmd * maxAccelXY / accelCmdNorm). The accelCmd is the output. This is broken into multiple step so we can constrain the horizontal velocity and acceleration.

1.6 Yaw Control

This is the simplest controller of all, it's a proportional controller for yaw. The only tricky part is recalculating/normalizing the yaw angle. First, we can mod the input by 2*pi because there is no point in making 360 degrees rotation. Next if the error term is greater than pi (180 degrees), subtract 2*pi (360 degrees) from it, because it's better to rotate from the other side. i.e. rotating 30 degrees clockwise is easier than rotating 330 degrees counter clockwise. Similarly if it's less than -pi, add 2*pi to it. Then the output is basically k_p_yaw * psi_err.

1.7 Motor Commands

Given commanded thrust u1 and desired moments in the three body axes u2, u3, u4, the controller needs to be able to convert it to real motor commands for each of the four rotors. This is done by solving a system of linear equations.

$$\begin{cases} F_1 + F_2 + F_3 + F_4 = term_t \\ F_1 - F_2 + F_3 - F_4 = term_x \\ F_1 + F_2 - F_3 - F_4 = term_y \\ F_1 - F_2 - F_3 + F_4 = term_z \end{cases}$$

where

$$\begin{cases} term_t = collThrustCmd \\ term_x = momentCmd.x/l \\ term_y = momentCmd.y/l \\ term_z = -momentCmd.z/kappa \\ l = \frac{L}{2\sqrt{2}} \end{cases}$$

Note that term_z is negated with a negative sign as the rotors are rotating in the opposite direction. Also lower case l is the perpendicular distance to axes.

Solving this 4 equations and 4 unknowns gives us F1, F2, F3, F4, the commanded individual thrust for the four rotors.

$$\begin{cases} cmd.desiredThrustsN[0] = (term_t + term_x + term_y + term_z)/4; \\ cmd.desiredThrustsN[1] = (term_t - term_x + term_y - term_z)/4; \\ cmd.desiredThrustsN[2] = (term_t + term_x - term_y - term_z)/4; \\ cmd.desiredThrustsN[3] = (term_t - term_x - term_y + term_z)/4; \end{cases}$$