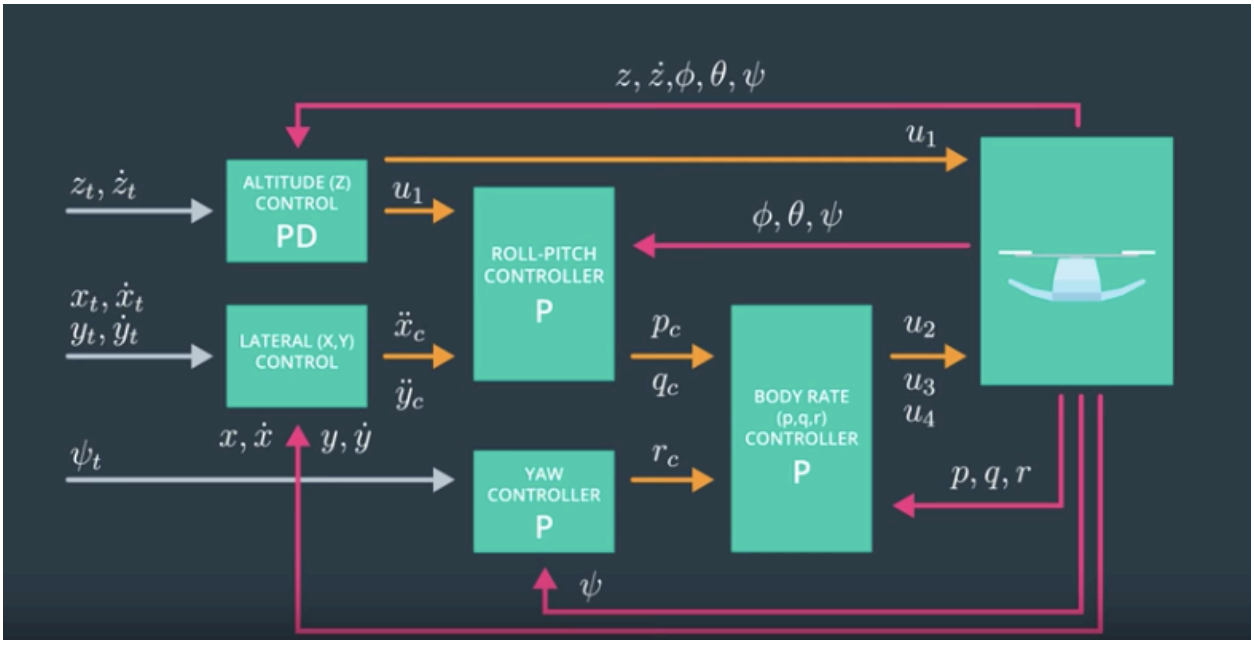
FCND Controls Project Report

The goal of this project is to complete C++ code and also tune the parameters to implement the 3D Controller for a Quadrotor. The Controller is composed of 5 smaller controller components:

1-Altitude controller which is a PD controller
2-Lateral position controller which is a PD controller
3-Roll-pitch controller which is a P controller
4-Body rate controller which is a P controller
5-Yaw controller which is a P controller

Roll-pitch controller, body rate controller and yaw controller together can be considered as Attitude controller which is a PD controller. The following picture from class shows the overall cascaded controller architecture:



There are 6 function in QuadControll.cpp which should be implemented. The equations and calculations are based the course material and papers introduced in the classroom.

1-GenerateMotorCommands

Purpose: Convert a desired 3-axis moment and collective thrust command to individual motor thrust commands

$T1$, $T2$, $T3$ and $T4$ are thrusts for front left, front right, rear left and rear right propellers. U1, U2, U3 and U4 are control inputs. T is the collective thrust, MX is moment around x axis, MY is the moment around y axis and MZ is the moment around z axis. l is the perpendicular distance to axes. kappa is drag/thrust ratio. Then we will have the following equations:

U1 = T
U2 = MX / l
U3 = MY / l
U4 = MZ / kappa

T1 + T2 + T3 + T4 = U1
T1 - T2 + T3 - T4 = U2
T1 + T2 - T3 - T4 = U3
-T1 + T2 + T3 - T4 = U4

To calculate the thrust values based on control inputs, we need to find the inverse of the following matrix:

1 1 1 1
1 1 1 -1
1 1 -1 -1
-1 1 1 -1

which will be:

1/4 1/4 1/4 -1/4
1/4 -1/4 1/4 1/4
1/4 1/4 -1/4 1/4
1/4 -1/4 -1/4 -1/4

which means:

T1 = (UF + UX + UY - UZ) / 4
T2 = (UF - UX + UY + UZ) / 4
T3 = (UF + UX - UY + UZ) / 4
T4 = (UF - UX - UY - UZ) / 4

Which is implemented in GenerateMotorCommands function:

```
float l = L / sqrtf(2.0f);
float UF = collThrustCmd ;
float UX = momentCmd.x / l ;
float UY = momentCmd.y / l ;
float UZ = momentCmd.z / kappa;

float thrust0 = (UF + UX + UY - UZ) / 4.f;
float thrust1 = (UF - UX + UY + UZ) / 4.f;
float thrust2 = (UF + UX - UY + UZ) / 4.f;
float thrust3 = (UF - UX - UY - UZ) / 4.f;

cmd.desiredThrustsN[0] = thrust0;
cmd.desiredThrustsN[1] = thrust1;
cmd.desiredThrustsN[2] = thrust2;
cmd.desiredThrustsN[3] = thrust3;
```

2-BodyRateControl

Purpose: Calculate a desired 3-axis moment given a desired and current body rate

As the control diagram shows, body rate controller is a P controller and gets body rates p, q and r as input and its outputs are the 3 controls u2,u3 and u4 which are in fact the 3 moments. The moment is the product of moment of inertia by rotational acceleration which is proportional to body rate error. The implementation is simple:

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

After this function was implemented, by tuning kpPQR I could pass the attitude control simulation but other simulations still failed.

3-RollPitchControl

Purpose: Calculate a desired pitch and roll angle rates based on a desired global lateral acceleration, the current attitude of the quad, and desired collective thrust command

The roll-pitch controller is a P controller responsible for commanding the roll and pitch rates in the body frame. The equations for roll-pitch calculation based on class notebook are:

$$b_a^x = R_{13} \text{ and } b_a^y = R_{23}$$

$$\dot{b}_c^x = k_p(b_c^x - b_a^x)$$

$$\dot{b}_c^y = k_p(b_c^y - b_a^y)$$

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

Which is implemented as the following:

```
float c = collThrustCmd / mass;
V3F b = V3F(R(0, 2), R(1, 2), 0.f);
V3F b_c = -V3F(accelCmd.x , accelCmd.y , 0.f)/c;
b_c.constrain(-maxTiltAngle, maxTiltAngle);

V3F b_c_dot = kpBank * (b_c - b);
```

```
pqrCmd.x = (R(1, 0) * b_c_dot.x - R(0, 0) * b_c_dot.y) / R(2, 2);
pqrCmd.y = (R(1, 1) * b_c_dot.x - R(0, 1) * b_c_dot.y) / R(2, 2);
pqrCmd.z = 0.f;
```

4-AltitudeControl

Purpose: Calculate desired quad thrust based on altitude setpoint, actual altitude, vertical velocity setpoint, actual vertical velocity, and a vertical acceleration feed-forward command

Altitude controller is a PD controller. The equations used for this function based on the class notebook are:

$$\bar{u}_1 = k_{p\text{-}z}(z_t - z_a) + k_{d\text{-}z}(\dot{z}_t - \dot{z}_a) + \ddot{z}_t$$
$$b_z = R_{33}$$
$$c = (\bar{u}_1 - g)/b_z$$

The acceleration is calculated based on the errors in position and velocity and then the thrust is calculated considering the mass:

```
float z_err = posZCmd - posZ;
float b_z = R(2, 2);

velZCmd += kpPosZ * z_err;
velZCmd = CONSTRAIN(velZCmd, -maxAscentRate, maxDescentRate);

integratedAltitudeError += z_err * dt;
float z_err_dot = velZCmd - velZ;
accelZCmd += KiPosZ * integratedAltitudeError + kpVelZ * z_err_dot;

thrust = mass * (9.81 - accelZCmd) / b_z;
```

After this function implemented I tried to tune kpBank, kpPosZ, KiPosZ and kpVelZ and changed them to larger values. Although the simulations look better but still only attitude simulation passes.

5-LateralPositionControl

Purpose:Calculate a desired horizontal acceleration based on desired lateral position/velocity/ acceleration and current pose

Lateral controller is a PD controller which takes lateral position and velocity as input and outputs lateral accelerations. This function implements this by finding position error and calculating velocity command followed by calculating velocity error and acceleration command:

```
velCmd += kpPosXY * (posCmd - pos);
velCmd.x = CONSTRAIN(velCmd.x, -maxSpeedXY, maxSpeedXY);
```

```
velCmd.y = CONSTRAIN(velCmd.y, -maxSpeedXY, maxSpeedXY);

accelCmd += kpVelXY * (velCmd - vel);
accelCmd.x = CONSTRAIN(accelCmd.x, -maxAccelXY, maxAccelXY);
accelCmd.y = CONSTRAIN(accelCmd.y, -maxAccelXY, maxAccelXY);
accelCmd.z = 0.0F;
```

After implementing this function I tried to tune kpPosXY and kpVelXY along with fine tuning previous parameters. Attitude control simulation passed. Position control simulation partially pass(2 out of 3 pass). Non idealities simulation passes. Trajectory follow passes.

6-YawControl

Purpose: Calculate a desired yaw rate to control yaw to yawCmd

Control over yaw is decoupled from the other directions. A P controller is used to control the drone's yaw which calculates the yaw error based on desired and actual yaw and calculates the body rate r based on the error:

```
float psi_error = yawCmd - yaw;
psi_error = fmodf(psi_error, 2.0 * 3.1415);
yawRateCmd = kpYaw * psi_error;
```

After implementing this function, I tried to tune kpYaw along with fine tuning previous parameters and could pass all the simulations.

Results

Here's the simulation results for 5 scenarios:

1-Intro

PASS: ABS(Quad.PosFollowErr) was less than 0.500000 for at least 0.800000 seconds

2-Attitude control

PASS: ABS(Quad.Roll) was less than 0.025000 for at least 0.750000 seconds
PASS: ABS(Quad.Omega.X) was less than 2.500000 for at least 0.750000 seconds

3-Position control

PASS: ABS(Quad1.Pos.X) was less than 0.100000 for at least 1.250000 seconds
PASS: ABS(Quad2.Pos.X) was less than 0.100000 for at least 1.250000 seconds
PASS: ABS(Quad2.Yaw) was less than 0.100000 for at least 1.000000 seconds

4-Non idealities

PASS: ABS(Quad1.PosFollowErr) was less than 0.100000 for at least 1.500000 seconds

PASS: ABS(Quad2.PosFollowErr) was less than 0.100000 for at least 1.500000 seconds
PASS: ABS(Quad3.PosFollowErr) was less than 0.100000 for at least 1.500000 seconds

5-Trajectory follow

PASS: ABS(Quad2.PosFollowErr) was less than 0.250000 for at least 3.000000 seconds