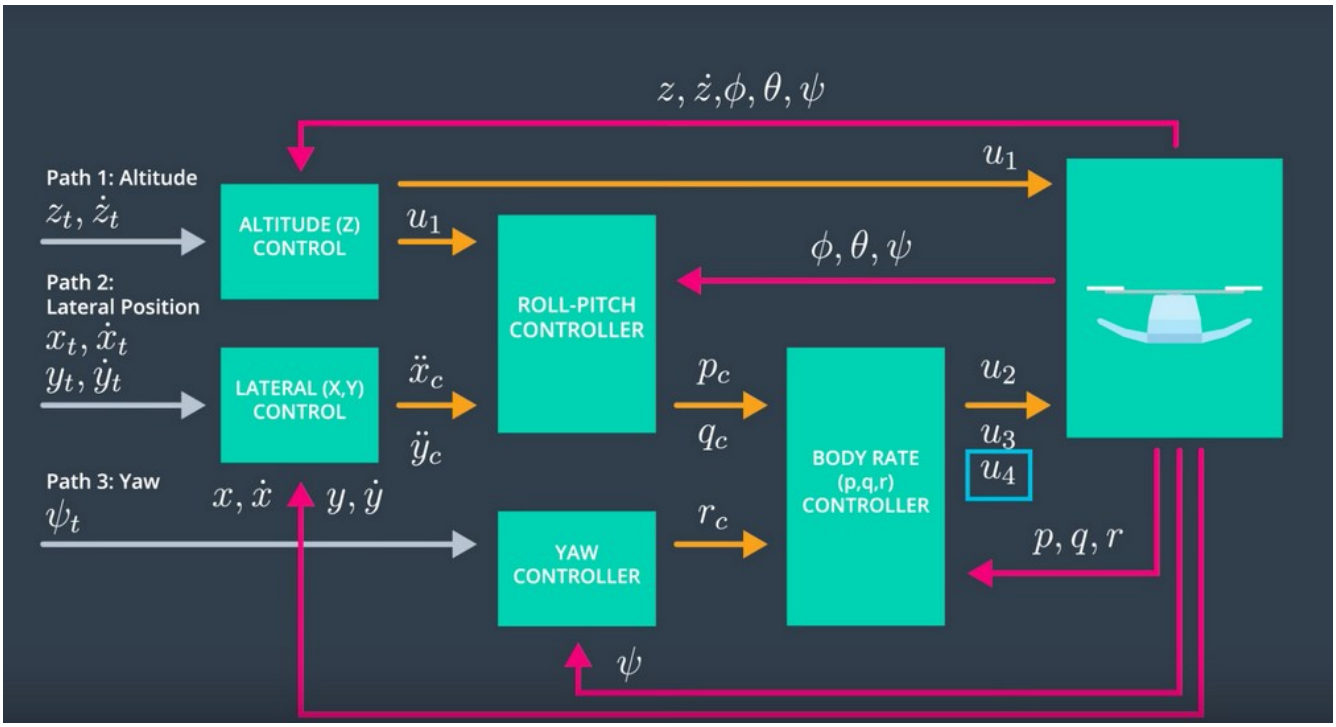
Report on Controller Setup and Tuning to Control Drone in a simulator on C++



→ Implemented body rate control in C++.

Since the Body Rate controller controls the ppr control, the entire system acts like Cascaded controller of Proportional Derivative and Proportional Controller.

```
V3F QuadControl::BodyRateControl(V3F pqrCmd, V3F pqr)
{
   V3F momentCmd;
   const V3F pqr_dot = kpPQR * (pqrCmd - pqr);
   momentCmd = V3F(Ixx, Iyy, Izz) * pqr_dot;
   return momentCmd;
}
```

→ Implement roll pitch control in C++.

The roll-pitch controller takes as input the commanded acc, the current attitude and the thrust command, and outputs the desired pitch and roll rates in body frame. The current tilt b_a_y and b_a_y from the rotation matrix R. Computation of desired tilt b_c_x and b_c_y by **normalizing** the desired acceleration by the thrust. Constraining tilt value is needed to prevent the drone from fliping.

Next, a P controller determines the desired roll and pitch rate in the world frame (b_c_x_dot and b_c_y_dot). In order to output the desired roll and pitch rate in the body frame, we apply a non-linear transformation as seen in the lectures, taking into account the rotation matrix as seen in the lectures.

```
V3F QuadControl::RollPitchControl(V3F accelCmd, Quaternion<float> attitude, float collThrustCmd)
{

   V3F pqrCmd;
   Mat3x3F R = attitude.RotationMatrix_IwrtB();
   const float b_x_a = R(0,2);
   const float b_y_a = R(1,2);

   // Target attitude
   const float thrust_acceleration = -collThrustCmd / mass;
   const float b_x_c = accelCmd.x / (thrust_acceleration);
   const float b_y_c = accelCmd.y / (thrust_acceleration);

   // Commanded rates in world frame
   const float b_x_c_dot = kpBank * (b_x_c - b_x_a);
   const float b_y_c_dot = kpBank * (b_y_c - b_y_a);

   // Roll and pitch rates
   const float r_33_inv = 1.0F / R(2,2);
   pqrCmd.x =  r_33_inv * (R(1,0)*b_x_c_dot - R(0,0)*b_y_c_dot);
   pqrCmd.y =  r_33_inv * (R(1,1)*b_x_c_dot - R(0,1)*b_y_c_dot);
   pqrCmd.z = 0.0F;  // yaw controller set in YawControl
   return pqrCmd;
}
```

→ Implement altitude controller in C++.

The altitude controller is a PD controller. The desired acceleration u_1_bar with the PD controller. Then we account for the non-linear effects of the attitude by incldуing b_z in its calculation .

```
float QuadControl::AltitudeControl(float posZCmd, float velZCmd, float posZ, float velZ,
                                   Quaternion<float> attitude, float accelZCmd,
float dt)
{
   Mat3x3F R = attitude.RotationMatrix_IwrtB();
    float thrust = 0;
   // Get z component of the thrust
    const float b_z = R(2,2);

    // Constrain commanded velocity (NED, descending means higher Z)
    velZCmd = CONSTRAIN(velZCmd, -maxAscentRate, maxDescentRate);

    // Compute error
    const float error = posZCmd - posZ;
    const float error_dot = velZCmd - velZ;
    integratedAltitudeError += error * dt;

    // Compute desired acceleration
    const float u1_bar = kpPosZ * error + \
                         kpVelZ * error_dot + \
                         KiPosZ * integratedAltitudeError + \
                         accelZCmd;
    float acc_z_desired = (u1_bar - CONST_GRAVITY) / b_z;

    // Compute thrust (positive upwards)
    thrust = -acc_z_desired * mass;
    return thrust;
}
```

→ Implement lateral position control in C++.

The lateral control is a second-order system problem, and thus we need to use a PD controller. The code is rather simple since we simply take as input position and velocities and output desired accelerations, all in NED coordinates which is also under constrain.

V3F QuadControl::LateralPositionControl(V3F posCmd, V3F velCmd, V3F pos, V3F vel, V3F accelCmd)

```
{
    accelCmd.z = 0;
    velCmd.z = 0;
    posCmd.z = pos.z;

   // Contrain desired velocity
    velCmd.x = CONSTRAIN(velCmd.x, -maxSpeedXY, maxSpeedXY);
    velCmd.y = CONSTRAIN(velCmd.y, -maxSpeedXY, maxSpeedXY);

    // Compute PD controller + feedforward
    const V3F error = posCmd - pos;
    const V3F error_dot = velCmd - vel;

    accelCmd = kpPosXY*error + kpVelXY*error_dot + accelCmd;

    // Constrain desired acceleration
    accelCmd.x = CONSTRAIN(accelCmd.x, -maxAccelXY, maxAccelXY);
    accelCmd.y = CONSTRAIN(accelCmd.y, -maxAccelXY, maxAccelXY);
    accelCmd.z = 0.0F;
     return accelCmd;
}
```

→ Implement yaw control in C++

Yaw controller is a P controller that takes as input the current and commanded yaw, and outputs the desired yaw rate in rad/s. We additionally, need to **normalize** the error to account for angle wrap.

float QuadControl::YawControl(float yawCmd, float yaw)

```
{
    float yawRateCmd=0;
    const float error = normalizeAngle(yawCmd - yaw);
    yawRateCmd = kpYaw * error;
    return yawRateCmd;
}
```

→ Implement calculating the motor commands given commanded thrust and moments in C++.

Following steps were considered during coding this part of the code

- Adjacent motors spin in opposite direction than shown in the lecture.
- The constants `k_m` and `k_f` are not given. Instead, the ration between them, is given.
- The distance `L` is the distance from the center of the quad to one of the rotors.

With these considerations, we solve the linear equation symbolically using Matlab and write the operations directly in C++ to improve computational performance. For example we expand all the operations instead of performing matrix multiplication.

`VehicleCommand QuadControl::GenerateMotorCommands(float collThrustCmd, V3F momentCmd)`

```
{    const float l = L * 0.5F * sqrt(2.0F);
    const float l_inv = 1.0F / l;
    const float k_inv = 1.0F / kappa;

    const float f = collThrustCmd;
    const float t_x = momentCmd.x;
    const float t_y = momentCmd.y;
    const float t_z = momentCmd.z;
    cmd.desiredThrustsN[0] = 0.25 * (f + l_inv*t_x + l_inv*t_y - k_inv*t_z); //
front left
    cmd.desiredThrustsN[1] = 0.25 * (f - l_inv*t_x + l_inv*t_y + k_inv*t_z); //
front right
    cmd.desiredThrustsN[2] = 0.25 * (f + l_inv*t_x - l_inv*t_y + k_inv*t_z); //
rear left
    cmd.desiredThrustsN[3] = 0.25 * (f - l_inv*t_x - l_inv*t_y - k_inv*t_z); //
rear right

    return cmd;
}
```