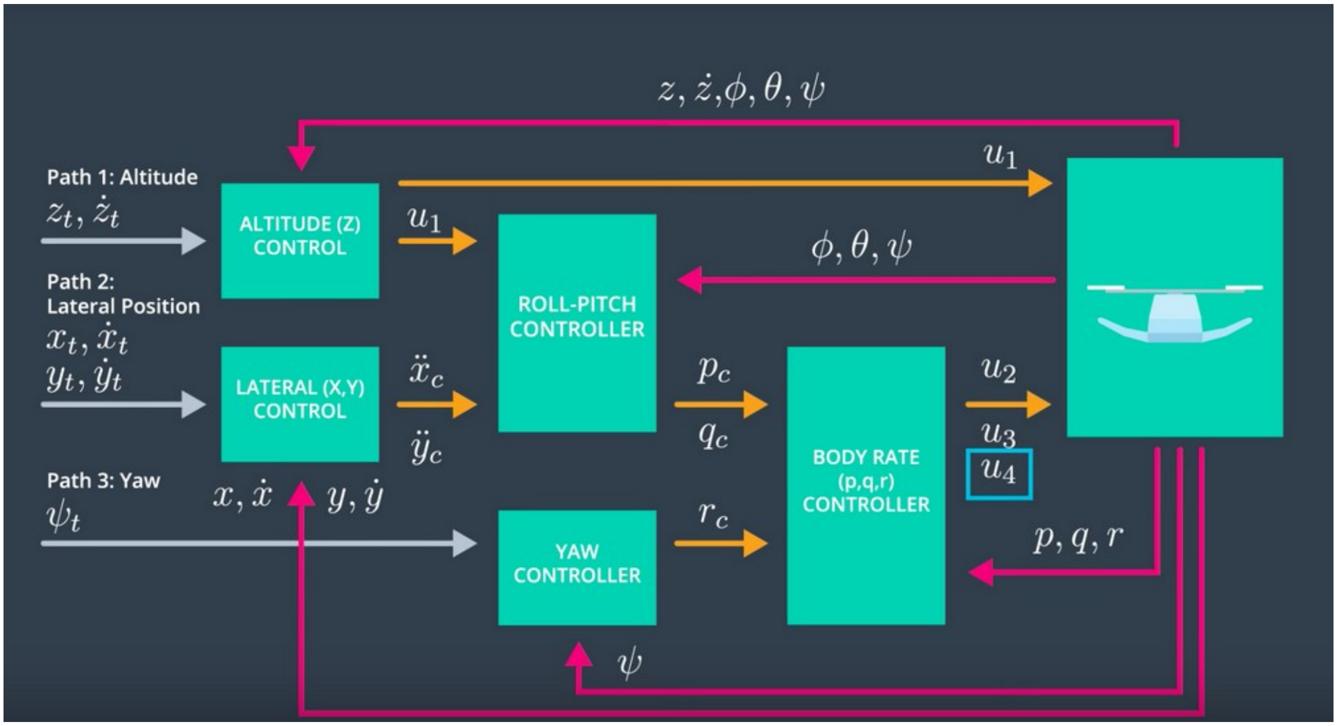
Report on Controller Setup and Tuning to Control Drone in a simulator on C++



→ 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 diff_pqr = kpPQR * (pqrCmd - pqr);
momentCmd = V3F(Ixx, Iyy, Izz) * diff_pqr;
return momentCmd;
}
```

→ Implement roll pitch control in C++.

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

Next, a P controller determines the desired roll and pitch rate in the world frame (`diff_bXC` and `diff_bYC`). 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 bXA = R(0,2), bYA = R(1,2);
const float acc_thrust = -collThrustCmd / mass;
const float bXC = accelCmd.x / (acc_thrust), bYC = accelCmd.y / (acc_thrust);
const float diff_bXC = kpBank * (bXC - bXA);
const float diff_bYC = kpBank * (bYC - bYA);
const float inv_rot = 1.0F / R(2,2);
pqrCmd.x = inv_rot * (R(1,0)*diff_bXC - R(0,0)*diff_bYC);
pqrCmd.y = inv_rot * (R(1,1)*diff_bXC - R(0,1)*diff_bYC);
pqrCmd.z = 0.0F;
return pqrCmd;
}
```

→ Implement altitude controller in C++.

The altitude controller is a PD controller. The desired accel is `U1_alter` Then we account for the non-linear effects of the attitude by including `BZ` .

```
ffloat 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;
// Thrust
const float BZ = R(2,2);
velZCmd = CONSTRAIN(velZCmd, -maxAscentRate, maxDescentRate);
//err
const float err = posZCmd - posZ;
const float diff_err = velZCmd - velZ;
integratedAltitudeError += err * dt;
const float U1_alter = kpPosZ * err + kpVelZ * diff_err + KiPosZ * integratedAltitudeError +
accelZCmd;
float acc_z_desired = (U1_alter - CONST_GRAVITY) / BZ;
thrust = -acc_z_desired * mass;
return thrust;
}
```

→ Implement lateral position control in C++.

The lateral control is a 2^{nd} order system, and thus a PD controller is used. The code takes 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 accelCmdFF)
{ accelCmdFF.z = 0;
velCmd.z = 0;
posCmd.z = pos.z;
V3F accelCmd = accelCmdFF;
velCmd.x = CONSTRAIN(velCmd.x, -maxSpeedXY, maxSpeedXY);
velCmd.y = CONSTRAIN(velCmd.y, -maxSpeedXY, maxSpeedXY);
// Control Loops Params
const V3F err = posCmd - pos;
const V3F diff_err = velCmd - vel;
accelCmd = kpPosXY*err + kpVelXY*diff_err + accelCmd;
// Desired accel
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 for angle wrap.

```
float QuadControl::YawControl(float yawCmd, float yaw)
{
float diff_yaw_CMd=0;
const float err = normalizeAngle(yawCmd - yaw);
diff_yaw_CMd = kpYaw * err;
return diff_yaw_CMd;
}
```

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

Following steps were considered during generating motor analysis

- Adjacent motors spin in opposite direction. And Opposite in same same direction.
- `k_m / k_f` is considered.
- `L` is the distance from the center of the quad to one of the rotors.

With these considerations, we solve the linear equation are solved

```
VehicleCommand QuadControl::GenerateMotorCommands(float collThrustCmd, V3F momentCmd)
{ const float len = L*0.5F*sqrt(2.0F);
```

```
const float len_inv = 1.0F / len;
const float forc_thrust = collThrustCmd;
const float TX = momentCmd.x, TY = momentCmd.y, TZ = momentCmd.z;
const float kVariable_inverse = 1.0F / kappa;
cmd.desiredThrustsN[0] = (1/(4.0)) * (forc_thrust + len_inv*TX + len_inv*TY -
kVariable_inverse*TZ);
cmd.desiredThrustsN[1] = (1/(4.0)) * (forc_thrust - len_inv*TX + len_inv*TY +
kVariable_inverse*TZ);
cmd.desiredThrustsN[2] = (1/(4.0)) * (forc_thrust + len_inv*TX - len_inv*TY +
kVariable_inverse*TZ);
cmd.desiredThrustsN[3] = (1/(4.0)) * (forc_thrust - len_inv*TX - len_inv*TY -
kVariable_inverse*TZ);
return cmd;
}
```