Udacity Project 2 Write Up: C++ Controller

Derek Lukacs
July 23, 2018

Contents

1 Overview

Figure 1 shows the flow of information in the controller. This diagram is similar to the control diagram but focuses on the actual implementation. The implementation inside of the blue blocks is what was done for this project. By defining the behavior inside these blocks the control behavior is defined.

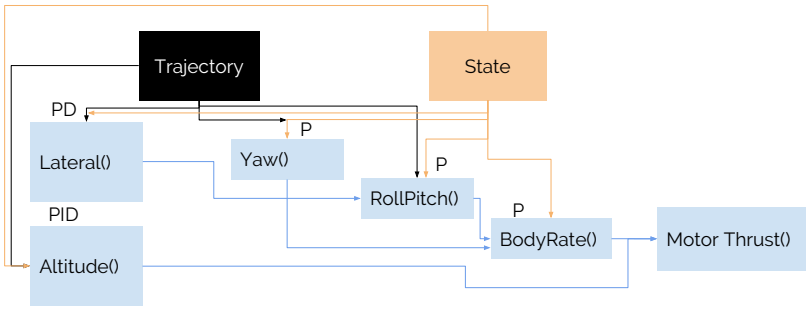

Figure 1: Information flow for the quad controller

2 Implementations

After receiving the Udacity quadrotor simulation package I was tasked with implementing several functions to achieve the desired performance. This section outlines the specifics of how these functions were implemented.

2.1 Body Rate Control

The body rate controller in this project takes three desired angular rates as well as the current angular rates to calculate a moment to induce. This is done with a proportional term on the error between PQR_{actual} and $PQR_{desired}$. Additionally, in order to take into consideration of the dynamics of the vehicle, the inertial vector is factored in.

```
V3F QuadControl::BodyRateControl(V3F pqrCmd, V3F pqr)
{
  // Calculate a desired 3-axis moment given a desired and current body rate
  // INPUTS:
  //   pqrCmd: desired body rates [rad/s]
  //   pqr: current or estimated body rates [rad/s]
  // OUTPUT:
  //   return a V3F containing the desired moments for each of the 3 axes

  V3F momentCmd;
  ////////////////////////////// BEGIN STUDENT CODE ////////////////////////////
    V3F inertia_vec(Ixx, Iyy, Izz);
    momentCmd = inertia_vec*kpPQR*(pqrCmd - pqr);
  ////////////////////////////// END STUDENT CODE //////////////////////////////
  return momentCmd;
}
```

2.2 Roll Pitch Control

In order to control the roll and pitch we must control the body rates. To calculate *pqrCmd* it is necessary to calculate the roll and pitch rates and then do a coordinate transformation into the body frame. Roll and pitch rates come from a proportional term on the roll pitch error.

```
V3F QuadControl::RollPitchControl(V3F accelCmd, Quaternion<float> attitude, float
    collThrustCmd)
{
  // 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
  // INPUTS:
  //   accelCmd: desired acceleration in global XY coordinates [m/s2]
  //   attitude: current or estimated attitude of the vehicle
  //   collThrustCmd: desired collective thrust of the quad [N]
  // OUTPUT:
  //   return a V3F containing the desired pitch and roll rates. The Z
  //     element of the V3F should be left at its default value (0)

  V3F pqrCmd;
  Mat3x3F R = attitude.RotationMatrix_IwrtB();

  ////////////////////////////// BEGIN STUDENT CODE ////////////////////////////

    // parse roll pitch angles
    float b_a_x = R(0,2);
    float b_a_y = R(1,2);
    //turn collThrust into m/s^2
    float collAccelCmd = -collThrustCmd / mass;
    //calculate target r,p angles
    float b_a_x_target = atan ( accelCmd.x / collAccelCmd);
    float b_a_y_target = atan ( accelCmd.y / collAccelCmd);
    //constrain angles
    b_a_x_target = CONSTRAIN(b_a_x_target, -maxTiltAngle, maxTiltAngle);
    b_a_y_target = CONSTRAIN(b_a_y_target, -maxTiltAngle, maxTiltAngle);
    // use propertional term to calculate desired rate
    float b_c_x_dot = kpBank* (b_a_x_target - b_a_x);
    float b_c_y_dot = kpBank* (b_a_y_target - b_a_y);
    // convert to body frame rates
    pqrCmd.x = 1.0/R(2,2) * (  R(1,0)*b_c_x_dot - R(0,0)* b_c_y_dot );
    pqrCmd.y = 1.0/R(2,2) * (  R(1,1)*b_c_x_dot - R(0,1)* b_c_y_dot );

  ////////////////////////////// END STUDENT CODE //////////////////////////////
  return pqrCmd;
}
```

2.3 Altitude Controller

The altitude controller calculates the necessary total thrust to control the vertical position of the vehicle. Error for both proportional corrections and derivative corrections are applied. Any acceleration commands can be passed into the controller as a feed forward term. The addition of an integral term assists with errors in the mass model of the vehicle.

```
float QuadControl::AltitudeControl(float posZCmd, float velZCmd, float posZ, float velZ,
    Quaternion<float> attitude, float accelZCmd, float dt)
{
  // Calculate desired quad thrust based on altitude setpoint, actual altitude,
  //   vertical velocity setpoint, actual vertical velocity, and a vertical
  //   acceleration feed-forward command
  // INPUTS:
  //   posZCmd, velZCmd: desired vertical position and velocity in NED [m]
  //   posZ, velZ: current vertical position and velocity in NED [m]
  //   accelZCmd: feed-forward vertical acceleration in NED [m/s2]
  //   dt: the time step of the measurements [seconds]
  // OUTPUT:
  //   return a collective thrust command in [N]

  Mat3x3F R = attitude.RotationMatrix_IwrtB();
  float thrust = 0;

  ////////////////////////////// BEGIN STUDENT CODE ////////////////////////////
```

```
    float pos_err = posZCmd - posZ;
    float vel_err = velZCmd - velZ;

    integratedAltitudeError += dt*pos_err * KiPosZ;
    float z_accel = pos_err * kpPosZ + vel_err*kpVelZ + accelZCmd + integratedAltitudeError;
    thrust = (9.81 - z_accel) *mass / R(2,2);

  ////////////////////////////// END STUDENT CODE ////////////////////////////

  return thrust;
}
```

2.4 Lateral Position Controller

The lateral position controller calculates the necessary horizontal acceleration to control the XY position. This then gets fed into the roll pitch controller which seeks to achieve this acceleration. The lateral acceleration is calculated with proportional and derivative terms. The acceleration command is then constrained by ± maxAccelXY. This is a second order control loop.

```
// returns a desired acceleration in global frame
V3F QuadControl::LateralPositionControl(V3F posCmd, V3F velCmd, V3F pos, V3F vel, V3F
    accelCmdFF)
{
  // Calculate a desired horizontal acceleration based on
  //  desired lateral position/velocity/acceleration and current pose
  // INPUTS:
  //   posCmd: desired position, in NED [m]
  //   velCmd: desired velocity, in NED [m/s]
  //   pos: current position, NED [m]
  //   vel: current velocity, NED [m/s]
  //   accelCmdFF: feed-forward acceleration, NED [m/s2]
  // OUTPUT:
  //   return a V3F with desired horizontal accelerations.
  //     the Z component should be 0
  // HINTS:
  //  - use the gain parameters kpPosXY and kpVelXY
  //  - make sure you limit the maximum horizontal velocity and acceleration
  //    to maxSpeedXY and maxAccelXY

  // we initialize the returned desired acceleration to the feed-forward value.
  // Make sure to _add_, not simply replace, the result of your controller
  // to this variable
  V3F accelCmd = accelCmdFF;

  ////////////////////////////// BEGIN STUDENT CODE ////////////////////////////
    V3F pos_err =  posCmd - pos;
    V3F vel_err =  velCmd - vel;

    accelCmd.x = accelCmd.x + pos_err.x * kpPosXY + vel_err.x * kpVelXY;
    accelCmd.y = accelCmd.y + pos_err.y * kpPosXY + vel_err.y * kpVelXY;
    accelCmd.x = CONSTRAIN(accelCmd.x, -maxAccelXY, maxAccelXY );
    accelCmd.y = CONSTRAIN(accelCmd.y, -maxAccelXY, maxAccelXY );
  ////////////////////////////// END STUDENT CODE ////////////////////////////

  return accelCmd;
}
```

2.5 Yaw Controller

The yaw controller calculates the desired yaw rate based on the current yaw error and a proportional term. This is a first order system.

```
float QuadControl::YawControl(float yawCmd, float yaw)
{
  // Calculate a desired yaw rate to control yaw to yawCmd
```

```
  // INPUTS:
  //   yawCmd: commanded yaw [rad]
  //   yaw: current yaw [rad]
  // OUTPUT:
  //   return a desired yaw rate [rad/s]
  // HINTS:
  //  - use fmodf(foo,b) to unwrap a radian angle measure float foo to range [0,b].
  //  - use the yaw control gain parameter kpYaw

  float yawRateCmd=0;
  ////////////////////////////// BEGIN STUDENT CODE ////////////////////////////
    yaw = fmodf(yaw,2*3.14159);
    float yaw_err = yawCmd - yaw;
    yawRateCmd = yaw_err * kpYaw;
  ////////////////////////////// END STUDENT CODE //////////////////////////////
  return yawRateCmd;
}
```

2.6 Motor Thrusts

Calculating the motor thrusts is done by satisfying several conditions by solving a system of equations for the individual motor thrusts.

$$
\begin{aligned}
F &= T_0 + T_1 + T_2 + T_3 \\
M_x &= L\cos(\pi/4)\,(T_0 - T_1 + T_2 - T_3) \\
M_y &= L\cos(\pi/4)\,(-T_0 - T_1 + T_2 + T_3) \\
M_z &= \kappa\,(T_0 - T_1 - T_2 + T_3)
\end{aligned}
$$

The solution to this system of equations is given by:

$$
\begin{aligned}
T_0 &= \frac{1}{4}\left(F + \sqrt{2}\frac{M_x}{L} + \sqrt{2}\frac{M_y}{L} - \frac{M_z}{\kappa}\right) \\
T_1 &= \frac{1}{4}\left(F - \sqrt{2}\frac{M_x}{L} + \sqrt{2}\frac{M_y}{L} + \frac{M_z}{\kappa}\right) \\
T_2 &= \frac{1}{4}\left(F + \sqrt{2}\frac{M_x}{L} - \sqrt{2}\frac{M_y}{L} + \frac{M_z}{\kappa}\right) \\
T_3 &= \frac{1}{4}\left(F - \sqrt{2}\frac{M_x}{L} - \sqrt{2}\frac{M_y}{L} - \frac{M_z}{\kappa}\right)
\end{aligned}
$$

These equations are implemented below to mix the thrust between the motors and then each thrust is constrained within the operational limits of the motors.

```
VehicleCommand QuadControl::GenerateMotorCommands(float collThrustCmd, V3F momentCmd)
{
  // Convert a desired 3-axis moment and collective thrust command to
  //   individual motor thrust commands
  // INPUTS:
  //   collThrustCmd: desired collective thrust [N]
  //   momentCmd: desired rotation moment about each axis [N m]
  // OUTPUT:
  //   set class member variable cmd (class variable for graphing) where
  //   cmd.desiredThrustsN[0..3]: motor commands, in [N]

  // HINTS:
  // - you can access parts of momentCmd via e.g. momentCmd.x
  // You'll need the arm length parameter L, and the drag/thrust ratio kappa

  ////////////////////////////// BEGIN STUDENT CODE //////////////////////////////

```

```
    float t = collThrustCmd;
    float r = momentCmd.x;
    float p = momentCmd.y;
    float y = momentCmd.z;

    cmd.desiredThrustsN[0] = (1.0/4.0)*t + (1.0/4.0)*sqrt(2.0)*r/L+(1.0/4.0)*sqrt(2.0)*p/L
    -(1.0/4.0)*y/kappa; // front left
    cmd.desiredThrustsN[1] = (1.0/4.0)*t - (1.0/4.0)*sqrt(2.0)*r/L+(1.0/4.0)*sqrt(2.0)*p/L
    +(1.0/4.0)*y/kappa; // front right
    cmd.desiredThrustsN[2] = (1.0/4.0)*t + (1.0/4.0)*sqrt(2.0)*r/L-(1.0/4.0)*sqrt(2.0)*p/L
    +(1.0/4.0)*y/kappa; // rear left
    cmd.desiredThrustsN[3] = (1.0/4.0)*t - (1.0/4.0)*sqrt(2.0)*r/L-(1.0/4.0)*sqrt(2.0)*p/L
    -(1.0/4.0)*y/kappa; // rear right

    for(int i =0; i<4 ; ++i){
        cmd.desiredThrustsN[i] = CONSTRAIN(cmd.desiredThrustsN[i], minMotorThrust,
    maxMotorThrust);
    }
  ////////////////////////////// END STUDENT CODE ////////////////////////////
  return cmd;
}
```

3 Performance

4 Path Planning Expansion

Appendices

A QuadControl.cpp

```
CODE HERE
```