Project 3

FCND Controls CPP

by

Neema Rustin Badihian

Implement body rate control in C++.

First, I found the error of the body rotation rates (**rate_error**) by subtracting the actual pqr from the commanded pqr rates. Then I found the error of the kp constant for each pqr (**kpErr**) by multiplying kpPQR by the error of the body rotation rates I previously found. Finally, I found the moments about the x, y, and z axes by multiplying the moments of inertia for each axis by the **kpErr** for each axis and then placed these values in the **momentCmd** variable.

```
V3F rate_error = pqrCmd - pqr;
V3F kpErr = kpPQR * rate_error;

float xMoment = Ixx * kpErr[0];
float yMoment = Iyy * kpErr[1];
float zMoment = Izz * kpErr[2];

momentCmd = V3F(xMoment, yMoment, zMoment);
```

Implement roll pitch control in C++.

For the roll pitch controller, I used both Lesson 14 and the python solution for the project as references to find a solution that worked. First I found acceleration (**c_d**) by dividing the collective thrust by the mass of the vehicle. Then I went about find the the **b_x_p_term** and **b_y_p_term** by using the same process for both. I initialized variables for both **b_y_actual** and **b_x_actual** by getting their value from the rotation matrix **R**, this way the code would be easier to understand. The commanded b_x and b_y were found by taking the commanded acceleration for x and y individually and dividing them by **c_d** I found earlier. The error for both the b_x and b_y was found by simply subtracting the actual from the commanded. Finally, the p_terms for both b_x and b_y was found by multiplying them by the **kpBank** variable. With the p terms calculated, the commanded p and q were found by substituting the p terms into the formula we used before:

$$\begin{matrix} p_{cmd} \\ q_{cmd} \end{matrix} = \frac{1}{R_{33}} * \begin{matrix} R_{21} \\ R_{22} \end{matrix} * b_p^x - \begin{matrix} R_{11} \\ R_{12} \end{matrix} * b_p^y$$

```
if (collThrustCmd > 0) {

    float c_d = -collThrustCmd / mass;

    float b_x_actual = R(0, 2);
    float b_x_cmd = accelCmd.x / c_d;
    float b_x_err = b_x_cmd - b_x_actual;
    float b_x_p_term = kpBank * b_x_err;

    float b_y_actual = R(1, 2);
    float b_y_cmd = accelCmd.y / c_d;
    float b_y_err = b_y_cmd - b_y_actual;
    float b_y_p_term = kpBank * b_y_err;

    float p_cmd = (1 / R(2, 2)) * (R(1, 0) * b_x_p_term - R(0, 0) * b_y_p_term);
    float q_cmd = (1 / R(2, 2)) * (R(1, 1) * b_x_p_term - R(0, 1) * b_y_p_term);

    pqrCmd.x = p_cmd;
    pqrCmd.y = q_cmd;
    pqrCmd.z = 0.0;
}

else {
    pqrCmd.x = 0.0;
    pqrCmd.y = 0.0;
    pqrCmd.z = 0.0;
}
```

Implement altitude controller in C++.

The altitude controller depended heavily on the z axis properties. First, I found the error in the vertical position of the drone (**z_err**) by subtracting the actual position (**posZ**) from the commanded position (**posZCmd**). Next I found the error in the vertical velocity (**z_dot_err**) by subtracting the actual velocity (**velZ**) from the commanded velocity (**velZCmd**). The b_z term is the R_{33} value of the given rotation matrix so I created a new variable named **b_z** to make it easier to understand the code. To find $\bar{u}_1$ I need the p term, d term, i term, and commanded vertical acceleration. We are given the vertical acceleration as **accelZCmd** but the others needed to be calculated. The p term was found by multiplying **kpPosZ** by **z_err** (which was calculated earlier) and the d term was found by multiplying **kpVelZ** by **z_dot_err** (also calculated earlier) and adding the actual vertical velocity. Before finding the i term, it was necessary to adjust the integrated altitude error by adding to it the product of **z_err** and the time (**dt**). The i term was then found by multiplying **KiPosZ** by the (now adjusted) integrated altitude error. With the p, d, and i terms now calculated, I was able to find $\bar{u}_1$ by adding them together along with the commanded vertical acceleration. The c term I would need to find the desired thrust was calculated by taking the difference of $\bar{u}_1$ and gravity and dividing it by **b_z** (which remember was taken from the R_{33} value of the given rotation matrix). Finally, I used the **CONSTRAIN** function to clip **c** with the lower bound being **-maxAscentRate / dt** and the upper bound being **maxAscentRate / dt** and multiplied this calculation by **-mass**.

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

float p_term = kpPosZ * z_err;
float d_term = kpVelZ * z_dot_err + velZ;

integratedAltitudeError += z_err * dt;
float i_term = KiPosZ * integratedAltitudeError;

float u_bar = p_term + d_term + i_term + accelZCmd;

float c = (u_bar - CONST_GRAVITY) / b_z;

thrust = -mass * CONSTRAIN(c, -maxAscentRate / dt, maxAscentRate / dt);
```

Implement lateral position control in C++.

The lateral position control gave me the most trouble. I had to search online and through Slack a lot until I understood how to do it properly. First off, I used **kpPosXY** and **kpVelXY** to create two V3F variables (**kp_pos** and **kp_vel**) with 0.0 for their z values. I then declared another V3F variable called **vel_cmd** so that I would alter the value of **velCmd** as

I did my calculations. I then checked if **velCmd**'s **mag()** method returned a value greater than the maximum xy speed. If it did, I set **vel_cmd** to equal the return value of **velCmd**'s **norm()** method times the maximum xy speed. If **mag()** did not return a value greater than the maximum xy speed, I simply set **vel_cmd** to equal **velCmd**. Now that I had set **vel_cmd** appropriately, I could find the error in velocity by subtracting the actual velocity from the commanded velocity. I also found the error in position with this same method, subtracting the actual position from the commanded position. With these errors on hand and the V3F **kp_pos** and **kp_vel**, I calculated the commanded acceleration by adding to it the sum of **kp_pos** times **pos_err** and **kp_vel** times **vel_err**. I leaned on the **norm()** method again if **accelCmd**'s **mag()** method returned a value greater than the maximum xy acceleration by setting **accelCmd** to **accelCmd.norm()** times the maximum xy acceleration.

```
V3F kp_pos = V3F(kpPosXY, kpPosXY, 0.0);
V3F kp_vel = V3F(kpVelXY, kpVelXY, 0.0);
V3F vel_cmd;

if (velCmd.mag() > maxSpeedXY) {
    vel_cmd = velCmd.norm() * maxSpeedXY;
} else {
    vel_cmd = velCmd;
}

V3F pos_err = posCmd - pos;
V3F vel_err = vel_cmd - vel;

accelCmd += kp_pos * pos_err + kp_vel * vel_err;

if (accelCmd.mag() > maxAccelXY) {
    accelCmd = accelCmd.norm() * maxAccelXY;
}
```

Implement yaw control in C++.

Yaw control was definitely the easiest of all tasks. Calculating the commanded yaw rate requires use of the psi error so first I calculated that by subtracting the actual yaw from the commanded yaw. I then used the **fmodf** function as suggested in the instructions. Finally, I calculated the commanded yaw rate by multiplying **kpYaw** by **psi_err**.

```
float psi_err = yawCmd - yaw;
psi_err = fmodf(psi_err, 2.0 * M_PI);
yawRateCmd = kpYaw * psi_err;
```

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

For this I used the **t1, t2, t3, t4** equations for finding the individual motor thrust commands. **T1** and **t2** required the use of the variable **l** which I calculated by dividing **L** by the square root of 2. I then calculated **t1** and **t2** by dividing the desired rotation moment about the x and y axes by **l**. I calculated **t3** by dividing the desired rotation moment about the z axis by **kappa**. And I set **t4** to equal the desired collective thrust. Finally, I plugged these **t** values into the four equations for calculating each of the individual motor thrust commands.

```
float l = L / sqrtf(2.f);
float t1 = momentCmd.x / l;
float t2 = momentCmd.y / l;
float t3 = -momentCmd.z / kappa;
float t4 = collThrustCmd;

cmd.desiredThrustsN[0] = (t1 + t2 + t3 + t4) / 4.f; // front left
cmd.desiredThrustsN[1] = (-t1 + t2 - t3 + t4) / 4.f; // front right
cmd.desiredThrustsN[2] = (t1 - t2 - t3 + t4) / 4.f; // rear left
cmd.desiredThrustsN[3] = (-t1 - t2 + t3 + t4) / 4.f; // rear right
```