Project Write Up: Controls

Talha Khalid

September 7, 2018

1. Implementing Motor Commands

Motor commands were implemented by first determining individual moment commands based off the overall moment and collective thrust command. The overall moments were converted to a force value by dividing it by the arm length (except moment in the Z direction).

```
78    float l = L/sqrtf(2.f);
79    float mom_1=collThrustCmd;
80    float mom_2 = momentCmd.x/l;
81    float mom_3=momentCmd.y/l;
82    float mom_4=-momentCmd.z/kappa;
```

After determining individual moments, the respective thrusts on each motor was set based off the formulas shown in the lessons.

```
85      cmd.desiredThrustsN[0] = (mom_1+mom_2+mom_3+mom_4)/4; //front Left
86      cmd.desiredThrustsN[1] = (mom_1-mom_2+mom_3-mom_4)/4; //front right
87      cmd.desiredThrustsN[2] = (mom_1+mom_2-mom_3-mom_4)/4;// rear Left
88      cmd.desiredThrustsN[3] = (mom_1-mom_2-mom_3+mom_4)/4;//rear Right
```

2. Implementing Body Rate Control

The body rate controller was set using a P controller. The body rates in p,q,r were determined based off the error between the current and target body rates, multiplied by the proportional gain kpPQR. The moment of inertias was accounted for in the derivation of the model by multiplying moment of inertias in each respective direction by the appropriate body rate.

```
    V3F moi = V3F(Ixx, Iyy, Izz);
    V3F error_pqr = pqrCmd - pqr ;
114 V3F m_c = kpPQR * error_pqr ;
115
116  momentCmd = moi * m_c ;
```

3. Implementing Roll Pitch Control

The roll pitch controller implementation begins by first checking if the collective thrust is a positive value. If not, the pitch and roll rates in the x and y direction were set to 0. But if so, the

target bx and by terms were constrained between the opposite ends of the max tilt angle. Next, the error in bx and by was computed respectively and used in junction with a P controller to determine overall value. Finally, the roll and pitch were determined by applying the derived linear algebraic expression presented in the lessons.

```
146     float curr_accel, b_x, b_y,b_x_cmd,b_y_cmd, b_x_error, b_y_error, b_x_term,
147     if(collThrustCmd > 0){
148       curr_accel = -collThrustCmd/mass;
149
150       b_x_cmd = CONSTRAIN(accelCmd.x/curr_accel,-maxTiltAngle,maxTiltAngle);
151       b_y_cmd = CONSTRAIN(accelCmd.y/curr_accel,-maxTiltAngle,maxTiltAngle);
152
153       b_x = R(0,2); //current b_x
154       b_x_error = b_x_cmd - b_x;
155       b_x_term = kpBank*b_x_error;
156
157       b_y= R(1,2); //current b_y
158       b_y_error = b_y_cmd - b_y;
159       b_y_term = kpBank*b_y_error;
160
161       pqrCmd.x = (R(1,0)*b_x_term - R(0,0)*b_y_term)/R(2,2);
162       pqrCmd.y = (R(1,1)*b_x_term - R(0,1)*b_y_term)/R(2,2);
163     }
164     else{
165       pqrCmd.x = 0;
166       pqrCmd.y = 0;
167     }
```

4. Implementing Altitude Control

The altitude controller was implemented as a PID controller. The P and D terms were determined based off the error between current and target position/velocity values, respectively. The I term was determined based off the position error over the incremental time dt. In order to express it as a acceleration term, the output of the PID controller – gravity divided by b, based off the derivition provided in the lessons. The acceleration was contrained between opposite ends of the maxAscentRate divided by time. The output thrust was computed as the accceleration time mass.

```
199     float z_pos_error_term, z_vel_error_term, Z_accel_PID,acceler,int_error,ki_ter
200     z_pos_error_term = kpPosZ*(posZCmd-posZ);
201     z_vel_error_term = kpVelZ*(velZCmd - velZ);
202     int_error = 0;
203     int_error += z_pos_error_term * dt;
204     ki_term = int_error * KiPosZ;
205
206     Z_accel_PID = z_pos_error_term +  z_vel_error_term + ki_term+ accelZCmd;
207
208     acceler = (Z_accel_PID-CONST_GRAVITY)/R(2,2);
209     acceler = CONSTRAIN(acceler,-maxAscentRate/dt,maxAscentRate/dt);
210     thrust = -acceler* mass;
```

5. Implementing Lateral Position Control

The lateral position controller was implemented as a PD controller with feed forward acceleration by determining the error between target and current position/velocity, and then multiplied by the respective gains. Checks were performed to make sure that the magnitude of the target velocity and acceleration were not more than the maxSpeedXY parameter. If it was, the respective vector was normalized and capped to the maxSpeedXY.

```
247     V3F pos_err = posCmd - pos ;
248
249        if (velCmd.mag() > maxSpeedXY) {
250           velCmd = velCmd.norm() * maxSpeedXY;
251         }
252
253       V3F vel_err = velCmd - vel ;
254
255       accelCmd = kpPosXY * pos_err + kpVelXY * vel_err + accelCmd ;
256
257         if (accelCmd.mag() > maxAccelXY) {
258           accelCmd = accelCmd.norm() * maxAccelXY;
259         }
260
261        accelCmd.z = 0;
262
```

6. Implementing Yaw Controller

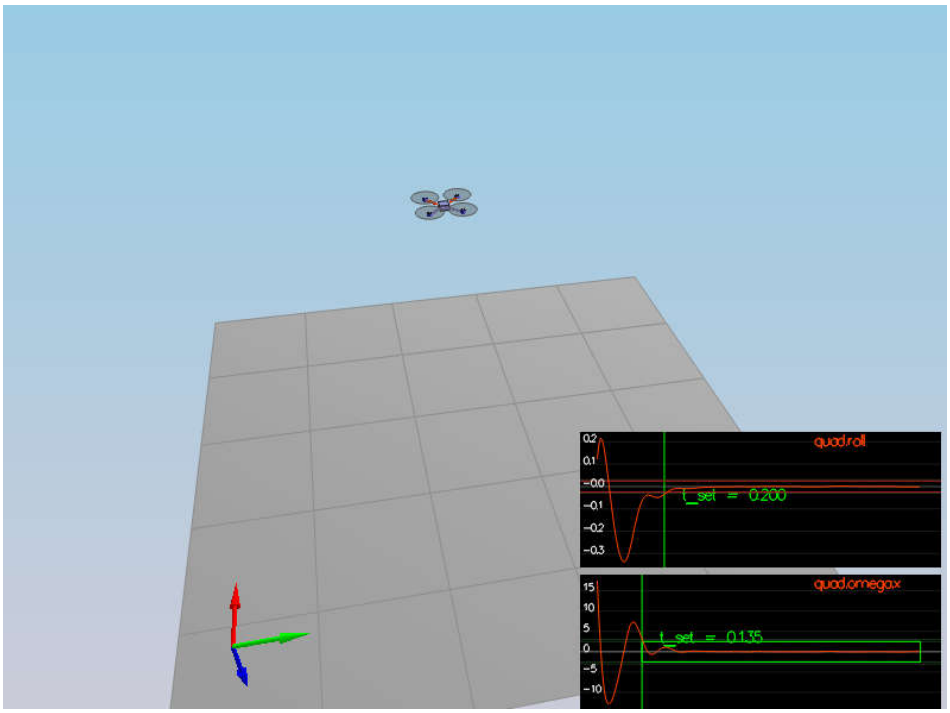
The yaw controller was implemented as a P controller. First, a check was done to see whether the target yaw was greater than 0 or less than. If 0 (i.e positive), the commanded yaw was set to the modulus between commanded yaw and 2*PI. Opposite is true if commanded yaw was negative. The error in yaw was calculated between the commanded yaw and current yaw. The next check that was done to see if the error in yaw was greater than PI or less than -PI (180^{o} to -

180^{o}). If error in yaw greater than PI, 2*PI is subtracted from the error in yaw. If error in yaw is less than -PI, 2*PI was added to error in yaw. Finally, the commanded yaw rate is calculated as the error in yaw times the yaw P gain.

```
284     float yaw_2p = 0;
285     if (yawCmd > 0){
286       yawCmd = fmodf(yawCmd, 2.f * F_PI);
287     } else {
288       yawCmd = fmodf(yawCmd, -2.f * F_PI);
289     }
290     float err_yaw = yawCmd - yaw ;
291
292      if ( err_yaw > F_PI ) {
293        err_yaw = err_yaw - 2.f * F_PI;
294      }
295      if ( err_yaw < -F_PI ) {
296        err_yaw = err_yaw + 2.f * F_PI;
297      }
298     yawRateCmd = kpYaw * err_yaw;
```

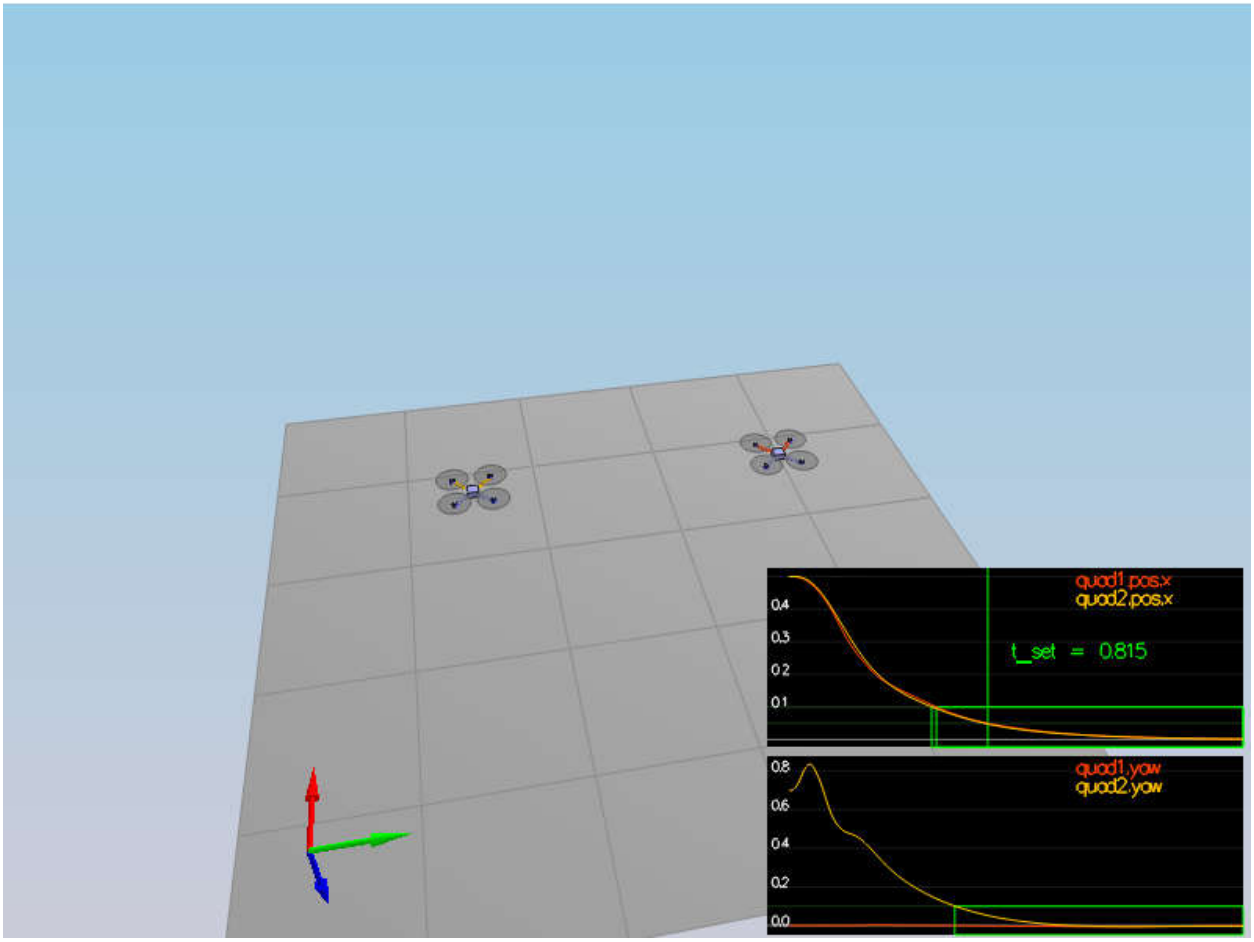
7. Scenario #2 Result

The drone was able to successfully roll while other parameters were held to 0 after tuning of the kpPQR and kpBank parameters. Roll was less than 0.025 radians for 0.75 seconds and roll rate was less than 0.25 rad/s.



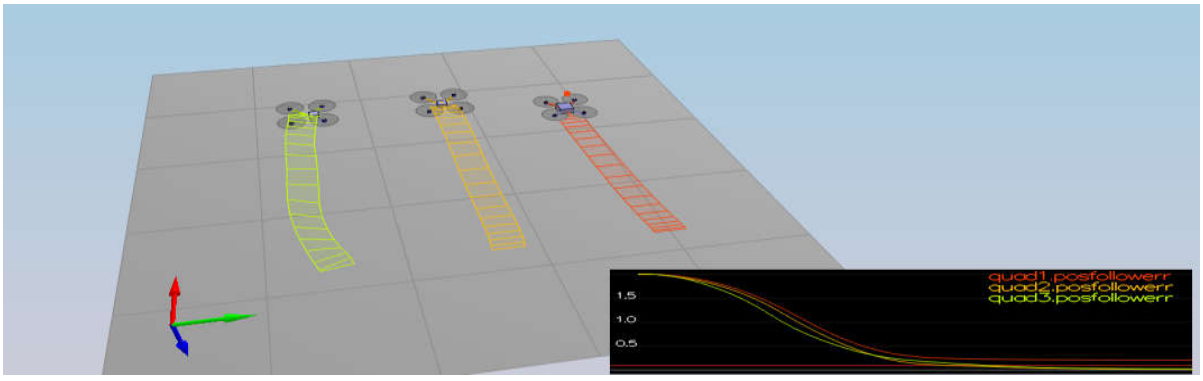
8. Scenario #3 Result

Both drones had minimal tracking errors and the yaw for both drones was controlled to 0 after tuning of the kpPosXY, kpPosZ, kpVelXY, kPVelZ, kpYaw and the 3rd component of kpPQR. X positions of both drones were within 0.1 m and Drone #2's yaw was within 0.1 of target.



9. Scenario #4 Result

After some more fine tuning of the various parameters as well as implementing the I component in the altitude controller, drones 2 and 3 were able to reach the target position with minimal error. Drone 1 was not able to reach the target position but got very close. Drones #2 and #3 had position errors less than 0.25 m. However, Drone #1 did not have less than 0.25m error.



10. Scenario #5 Result

Drone #2 was able to follow the path successfully with position error less than 0.25 m. However, Drone #1 was not able to follow within less than 0.5m. The average error was between 1m and 0.5 m.