Lesson 3 - Full Control of a quadrotor

Udacity - Flying Car Nanodegree - Project 03

Pedro Fernando Flores Palmeros
April 2020

1 Introduction

This is the final report fo the Project 3 which is realted to the full control of a quadrotor type-x, the controllers are PID, PD and P, depending of each subsystem nature.

There are many sections as functions developed by the student in the `QuadControl.cpp`.

2 GenerateMotorCommands

In this function the main idea is to compute the forces that each motor has to generate, as arguments the `collThrustCmd` and `momentCmd` are given, in the function these variables are renamed as `u1 = collTrhustCmd` and `tau_x = momentCmd.x/l` where `l` is the distance between any motor and the longitudinal axis of the quadrotor as can be seen in Fig(1).

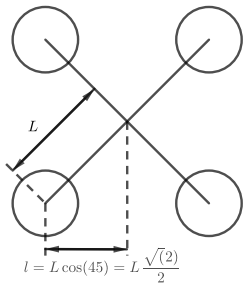

Figure 1: Superior view of a the quadrotor

The mathematical expression of the control inputs are

$$
\begin{aligned}
u1 &= f_1 + f_2 + f_3 + f_4 \\
\tau_x &= f_1 - f_2 - f_3 + f_4 \\
\tau_y &= f_1 + f_2 - f_3 - f_4 \\
\tau_z &= -f_1 + f_2 - f_3 + f_4
\end{aligned}
$$

observe that in τ_x, τ_y, τ_z are included the distance l and κ, hence the equations to compute the forces are given by

$$f_1 = (u1 + \tau_x + \tau_y + \tau_z)/4$$
$$f_2 = (u1 - \tau_x + \tau_y + \tau_z)/4$$
$$f_3 = (u1 + \tau_x - \tau_y - \tau_z)/4$$
$$f_4 = (u1 - \tau_x - \tau_y + \tau_z)/4$$

The code of this method is shown in the next cell.

```
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 l = L*0.707106781;

   float u1 = collThrustCmd;
   float tau_x = momentCmd.x/l;
   float tau_y = momentCmd.y/l;
   float tau_z = momentCmd.z/kappa;

   cmd.desiredThrustsN[0] = (u1 + tau_x + tau_y + tau_z)/4;
   cmd.desiredThrustsN[1] = (u1 - tau_x + tau_y + tau_z)/4;
   cmd.desiredThrustsN[2] = (u1 + tau_x - tau_y - tau_z)/4;
   cmd.desiredThrustsN[3] = (u1 - tau_x - tau_y + tau_z)/4;


   cmd.desiredThrustsN[0] =
       CONSTRAIN(cmd.desiredThrustsN[0],minMotorThrust,maxMotorThrust);
   cmd.desiredThrustsN[1] =
       CONSTRAIN(cmd.desiredThrustsN[1],minMotorThrust,maxMotorThrust);
   cmd.desiredThrustsN[2] =
       CONSTRAIN(cmd.desiredThrustsN[2],minMotorThrust,maxMotorThrust);
   cmd.desiredThrustsN[3] =
       CONSTRAIN(cmd.desiredThrustsN[3],minMotorThrust,maxMotorThrust);
```

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

  return cmd;
}
```

3 BodyRateControl

In this section a proportional controller for the body rate is computed, as inputs the desired or commanded body rate and the quadrotor boy rate are given as a parameters to this function. Hence the error between reference and state has to be computed and multiplied by a gain and also the Inertia Tensor as can be seen in the method body.

```
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

  // HINTS:
  //  - you can use V3Fs just like scalars: V3F a(1,1,1), b(2,3,4), c; c=a-b;
  //  - you'll need parameters for moments of inertia Ixx, Iyy, Izz
  //  - you'll also need the gain parameter kpPQR (it's a V3F)

  V3F momentCmd;

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

  V3F I;

  I.x = Ixx;
  I.y = Iyy;
  I.z = Izz;

  momentCmd = I*kpPQR*(pqrCmd - pqr);


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

  return momentCmd;
}
```

After the implementation of this controller the *scenario 2* was tested to very the well

behavior of the the platform

4 AltitudeControl

In this section a *Forwarded PID* Controller is design, to do this, the error between the desired altitude and the drone altitude has to be computed if multiplied by a constant `kpPosZ` the proportional term of the controller is obtained.

For the derivative term is given by the product of the gain `kpVelZ` by the difference of the desired altitude speed and the the drones speed in the $z - axis$.

The integral term hast to be computed by the summatory of the error by the δt or `dt` in the code.

The Feedforward term is just the addition of the desired acceleration of the drone in the z-axis to the the PID controller.

The code of this part shown below.

```
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]

   // HINTS:
   //  - we already provide rotation matrix R: to get element R[1,2] (python) use
      R(1,2) (C++)
   //  - you'll need the gain parameters kpPosZ and kpVelZ
   //  - maxAscentRate and maxDescentRate are maximum vertical speeds. Note
      they're both >=0!
   //  - make sure to return a force, not an acceleration
   //  - remember that for an upright quad in NED, thrust should be HIGHER if the
      desired Z acceleration is LOWER

   Mat3x3F R = attitude.RotationMatrix_IwrtB();

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

```
  float e_z = posZCmd - posZ;
  float e_z_dot = velZCmd - velZ;
  integratedAltitudeError += e_z *dt;

  /*  Proportional term of the PID */
  float P_z = kpPosZ*e_z;

  /* Derivative term of the PD) */
  float D_z = kpVelZ*e_z_dot;

  /* Itegral Tem of the PID */
  float I_z = KiPosZ*integratedAltitudeError;

  float PID_CTRL = P_z + D_z + I_z + accelZCmd;

  float thrust = -mass * CONSTRAIN(((PID_CTRL - 9.81)/R(2,2)),
      -maxAscentRate/dt, maxAscentRate/dt);


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

  return thrust;
}
```

5 LateralPositionControl

As in the altitude controller, in the xy plane a *Feedforward PID* is developed, the main idea is the same as in the altitude case, the main difference is that in this part the position is given a vector three floats, so all controllers are computed at the same time, but at the end the controller in the z-axis is set to 0 because the altitude controller have been developed.

```
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

  // make sure we don't have any incoming z-component
  accelCmdFF.z = 0;
  velCmd.z = 0;
  posCmd.z = pos.z;

  // 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 ////////////////////////////
  accelCmd.x = CONSTRAIN(accelCmd.x,-maxAccelXY, maxAccelXY);
  accelCmd.y = CONSTRAIN(accelCmd.y,-maxAccelXY, maxAccelXY);
  accelCmd.z = 0;

  velCmd.x = CONSTRAIN(velCmd.x, -maxSpeedXY, maxSpeedXY);
  velCmd.y = CONSTRAIN(velCmd.y, -maxSpeedXY, maxSpeedXY);
  velCmd.z = 0;



  V3F e_xi = posCmd - pos;

  V3F dot_e_xi = velCmd - vel;

  V3F u_xyz = kpPosXY*e_xi + kpVelXY*dot_e_xi + accelCmdFF;

  accelCmd = u_xyz;
  accelCmd.z = 0.0;



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

  return accelCmd;
}
```

6 YawControl

For the Yaw control a simple P controller is taken into account this is because in the Body Rate controller is the derivative part of the yaw controller.

```
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 ////////////////////////////

   float yaw_error = yawRateCmd - yaw;
   yaw_error = fmodf(yaw_error, F_PI*2.f);

   if (yaw_error >F_PI){
      yaw_error = yaw_error - 2.0f*F_PI;
   } else if (yaw_error < -M_PI){
      yaw_error = yaw_error + 2.0f*F_PI;
   }

   //cout << "yaw error: " << yaw_error << endl;
   yawRateCmd = kpYaw*yaw_error;


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

   return yawRateCmd;

}
```