Building a PID Controller in C++

By Abdulrahman Alemadi

1 IMPLEMENT BODY RATE CONTROL IN C++.

This is the first part of the project. When we run the simulator the drone lands quickly That means that the drone is not trying to do anything, a blank slate in other words.

For the BodyRate Controller be able to work, we use these inputs:

```
//  pqrCmd: desired body rates [rad/s]
//  pqr: current or estimated body rates [rad/s]
```

The inputs above are in V3F format. V3F is class that takes in 3 variables as a vector; {x, y, z}

Along with inertia variables in BaseController's class (Ixx, Iyy, Izz)

First, we calculate how far away the drone is from our desired state

```
V3F rate_error = pqrCmd - pqr;
```

We declare inertia variables as a vector:

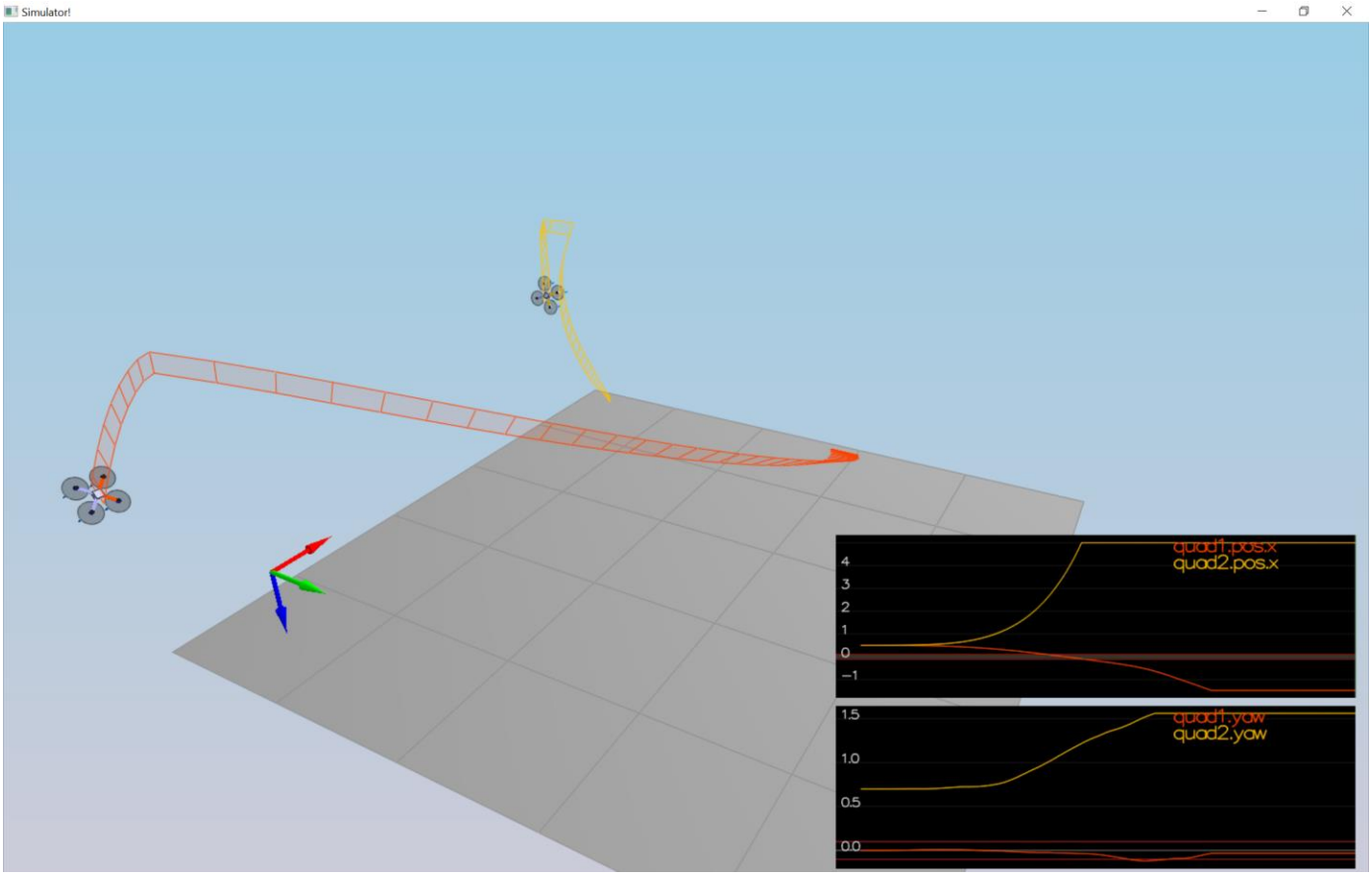
```
V3F inertia = V3F(Ixx, Iyy, Izz);
```

We multiply the inertia by the rate error and multiply by our body rate gain controller kpPQR; the gain controller is a sensitivity knob for our formula. The higher the value, the more reactive the resulting command will be.

```
momentCmd = (kpPQR * rate_error) * inertia;
```

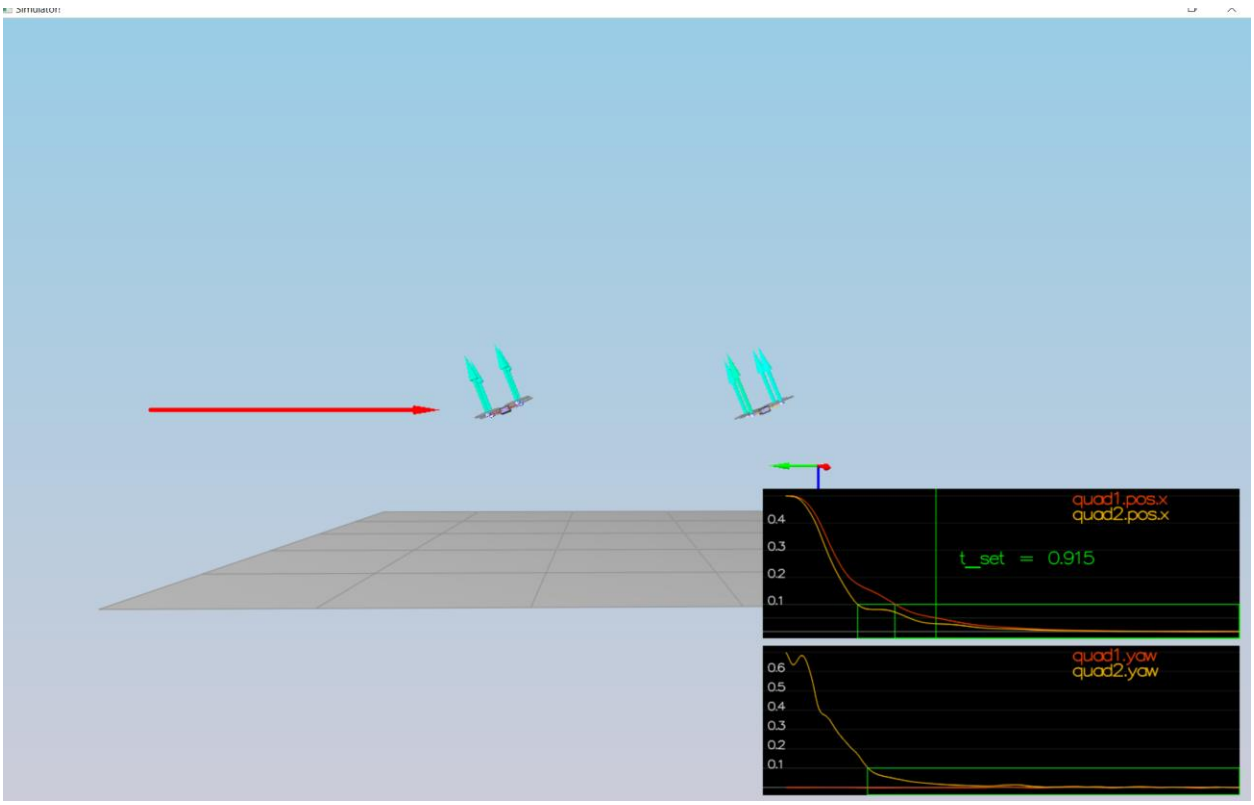
As a result, momentCmd will calculate the moment needed to feed it into the VehicleCommand QuadControl::GenerateMotorCommands function.

Here are the drones without the BodyRate Controller


Simulator!
quad1.pos.x
quad2.pos.x
4
3
2
1
0
−1
quad1.yaw
quad2.yaw
1.5
1.0
0.5
0.0

2 IMPLEMENT ROLL PITCH CONTROL IN C++.

This controller stabilizes the drone in roll-pitch attitude. As we can see below, when the drone is exposed to lateral forces, it stabilized itself by tilting in the opposite direction.



The inputs of the functions are below:

```
// 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)
```

First, we need to convert the collective force command to acceleration by this formula: a = f/m.

```
float collective_accel = collThrustCmd / mass;
```

Now we will declare the target acceleration for x and y.

target_x is the desired roll rate.

target_y is the desired pitch rate. Both are in m/s^2 as they are accelerations.

We will declare the target variables with a constraint of the tilt angle. As after a certain amount of roll or pitch would be unproductive, as it will flip the drone. The formula for the targets is accelCmd / collective_accel which is the acceleration of a roll/pitch direction divided by the aforementioned collective acceleration.

```
float target_x = -CONSTRAIN(accelCmd.x / collective_accel, -maxTiltAngle, maxTiltAngle);

float target_y = -CONSTRAIN(accelCmd.y / collective_accel, -maxTiltAngle, maxTiltAngle);
```

If the commanded collective thrust is less than 0, it means that the drone is on its correct desired timestep position. Therefore the drone should do nothing or start braking, in other words, the target acceleration should be assigned to 0.

```
if (collThrustCmd < 0)
{
        target_x = 0;
        target_y = 0;
};
```

Otherwise, it should continue with the preassigned target velocity. And we will declare the commanded pitch/roll velocity, which is the kpBank parameter 'sensitivity knob' multiplied by the error of the acceleration.

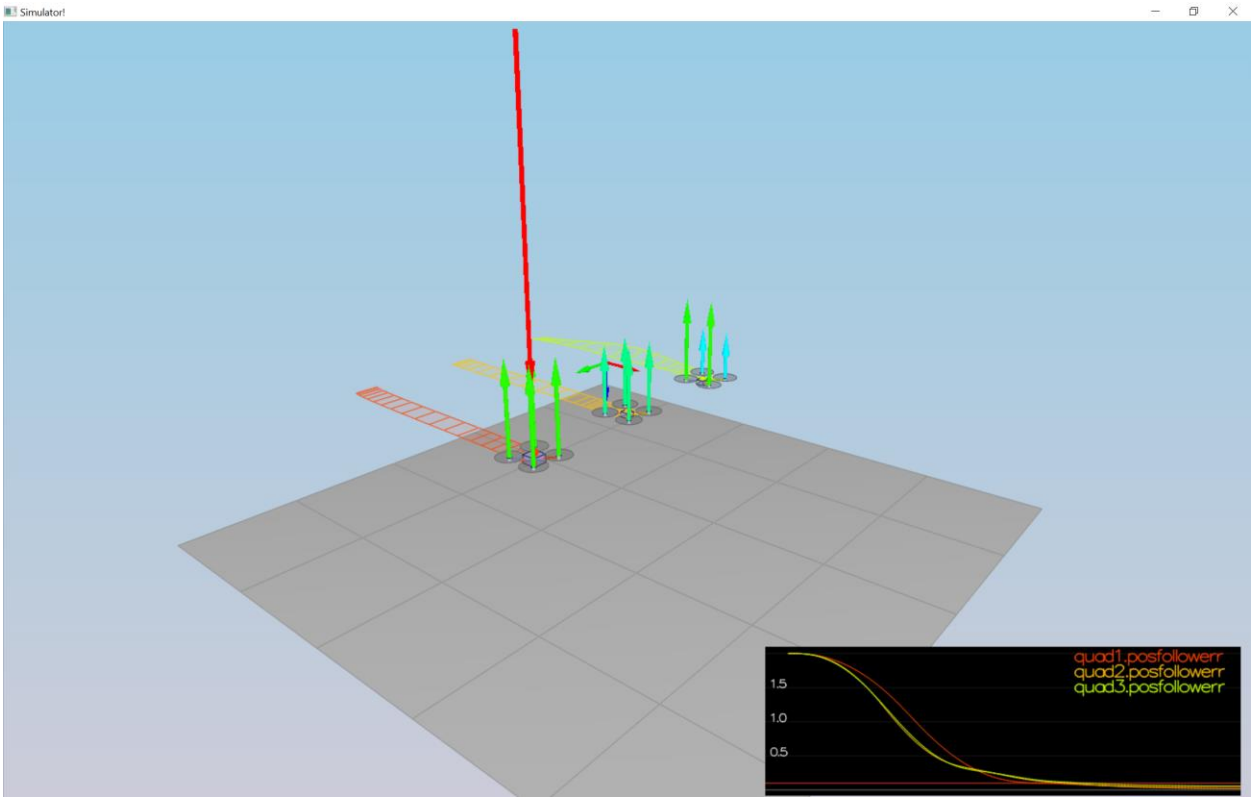
```
float b_x_c_dot = kpBank * (target_x - actual_x);
float b_y_c_dot = kpBank * (target_y - actual_y);
```

finally, we derive the commanded roll/pitch position by the below formula.

```
// desired roll-pitch
float p_c = (1 / R33)*(R21*b_x_c_dot - R11 * b_y_c_dot);
float q_c = (1 / R33)*(R22*b_x_c_dot - R12 * b_y_c_dot);
```

3 IMPLEMENT ALTITUDE CONTROLLER IN C++:

The altitude controller makes sure the altitude is not changed because of gusts of wind or other controllers' input. As we can see below when the drone is imposed to downward forces, it comprimises by raising the upward thrust.



This part of the code is similar in all controllers aforementioned above. But this time we have one more tuning parameter along with the vertical positional velocity, we have the tuning for the vertical velocity. The other differences are only regarding the differences in physical forces. And the full code for the altitude controller is below.

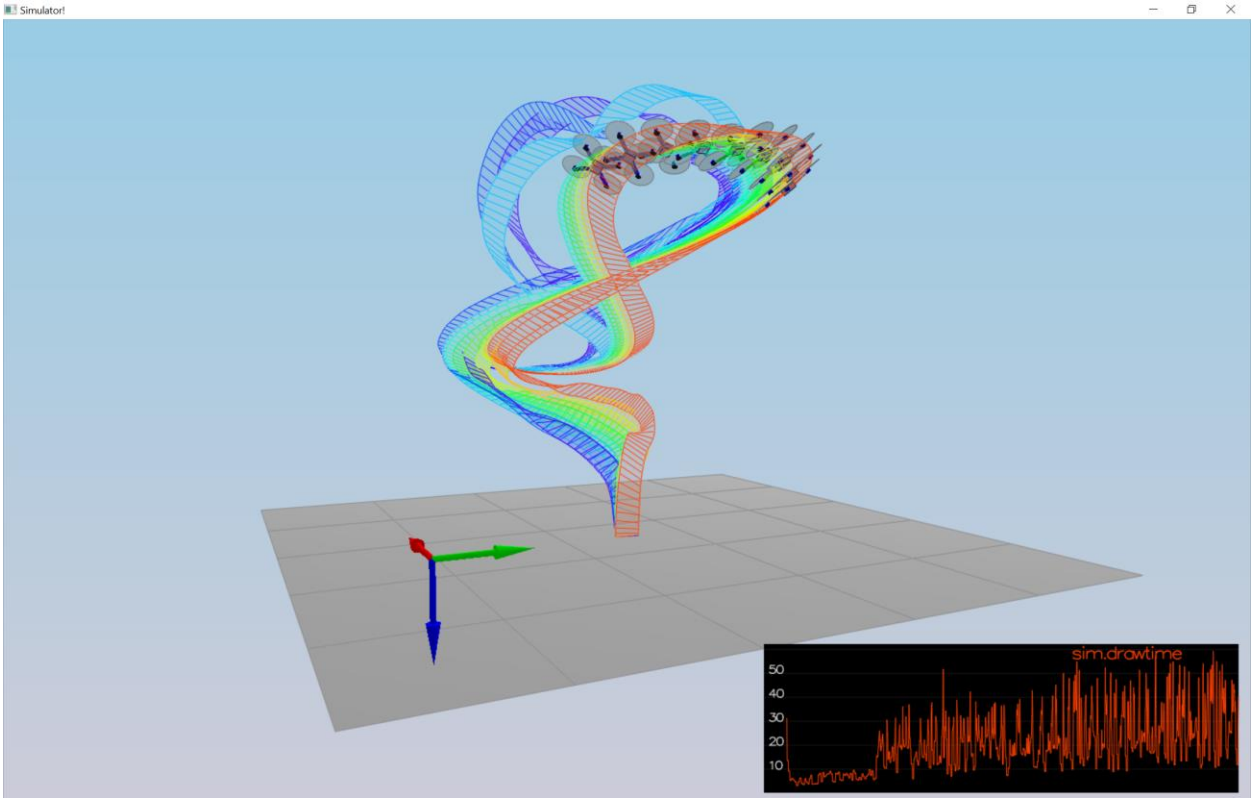
```
float QuadControl::AltitudeControl(float posZCmd, float velZCmd, float posZ, float velZ,
Quaternion<float> attitude, float accelZCmd, float dt)
{
        // Calculate desired quad thrust based on altitude setpoint, actual altitude,
        //   vertical velocity setpoint, actual vertical velocity, 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();
        float thrust = 0;

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


        float p_term = kpPosZ * (posZCmd - posZ);
        float d_term = kpVelZ * (velZCmd - velZ);
        float u1_bar = p_term + d_term + accelZCmd - 9.81f;

        //velZCmd = CONSTRAIN(velZCmd, -maxAscentRate, maxDescentRate);


        // acceleration to force f = ma
        thrust = -(u1_bar * mass / R(2, 2));
        ////////////////////////////// END STUDENT CODE //////////////////////////////

        return thrust;
}
```

4 IMPLEMENT LATERAL POSITION CONTROL IN C++.

This part of the controller is not concerned with the stability of the drone like the other controllers, but it is responsible for commanding the drone's position through the x and y axes.

Below is the controller in work by doing a figure 8.



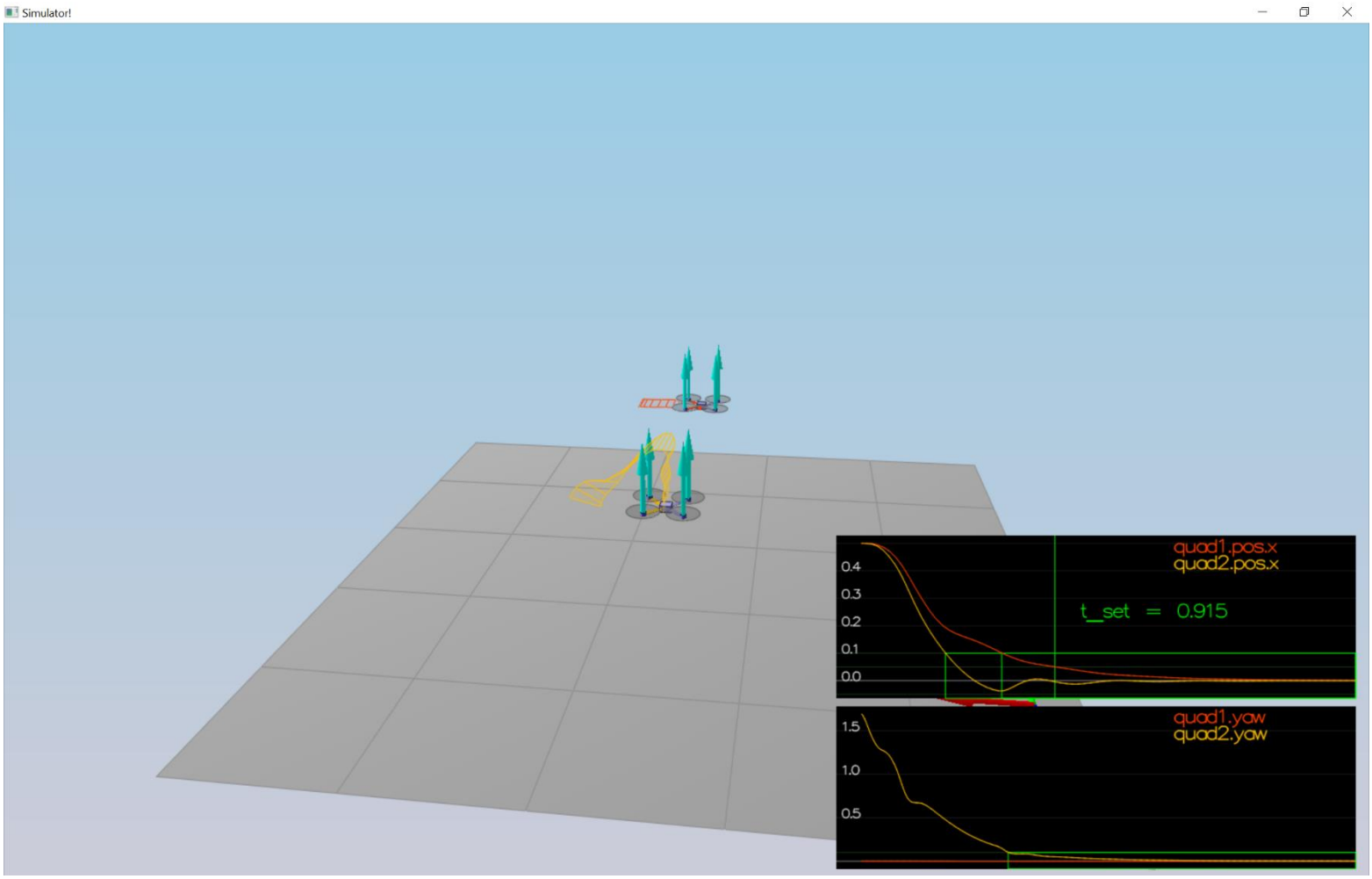
The challenging part of this part is transforming body attitude to the global frame.

```
// 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

// 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 ////////////////////////////
float vel_cmd = sqrtf(velCmd.x*velCmd.x + velCmd.y*velCmd.y);  // calculate the magnitude of the acceleration being commanded

if (vel_cmd > maxAccelXY) // if the commanded velocity is too high then reduce
{
        velCmd = velCmd * maxAccelXY / vel_cmd;
}

float x_p_term = kpPosXY * (posCmd.x - pos.x);
float x_d_term = kpVelXY * (velCmd.x - vel.x);
accelCmd.x = x_p_term + x_d_term + accelCmdFF.x;

float y_p_term = kpPosXY * (posCmd.y - pos.y);
float y_d_term = kpVelXY * (velCmd.y - vel.y);
accelCmd.y = y_p_term + y_d_term + accelCmdFF.y;

float ac_cmd = sqrtf(accelCmd.x*accelCmd.x + accelCmd.y*accelCmd.y);  // calculate the magnitude of the acceleration being commanded

if (ac_cmd > maxAccelXY) // if the commanded acceleration is too high then reduce
{
        accelCmd = accelCmd * maxAccelXY / ac_cmd;
}
return accelCmd;
```

5 IMPLEMENT YAW CONTROL IN C++.

As we can see below, drone2 (yellow) loses control for a bit, because of rapid yaw change, then it quickly stabilizes



Code for 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]
```

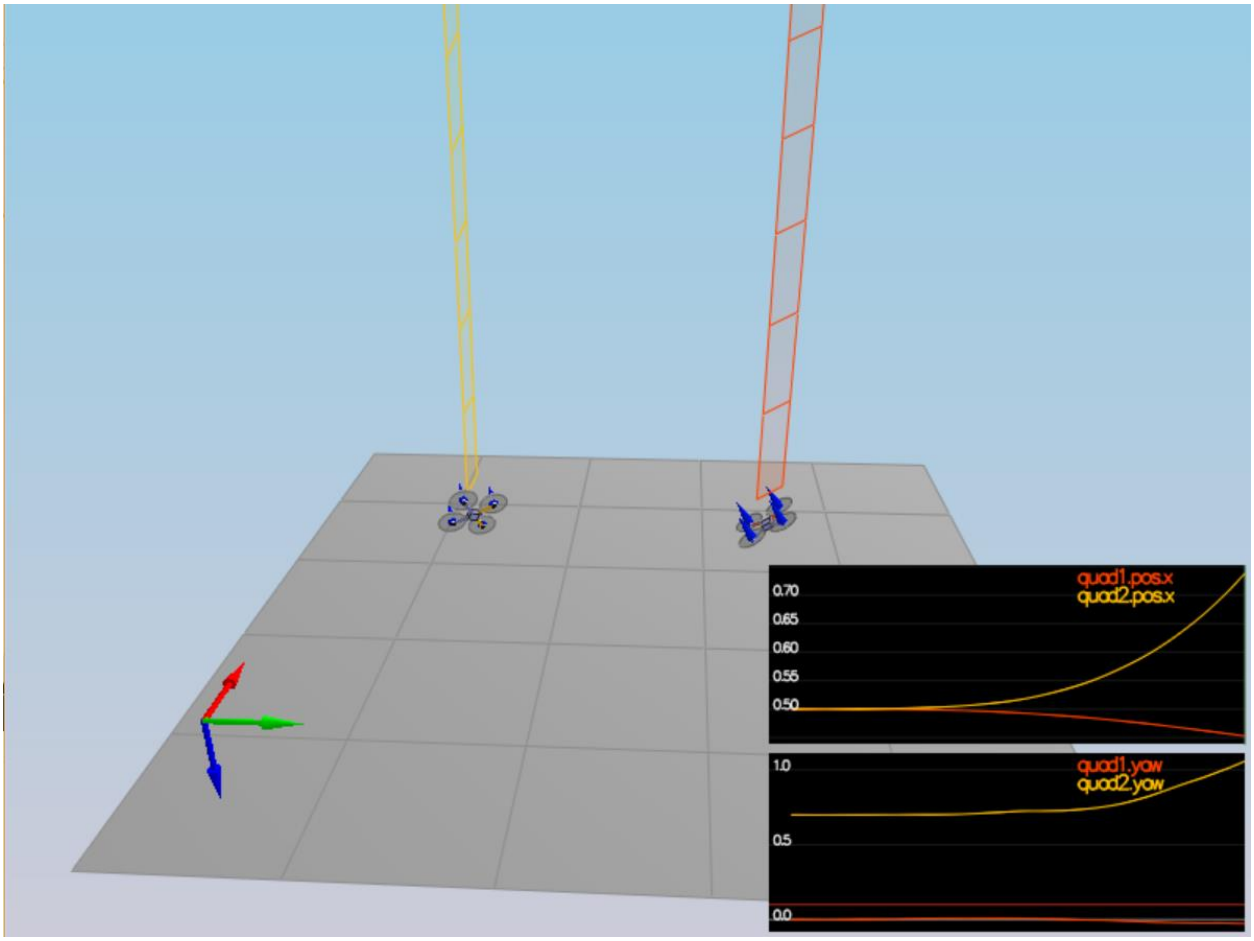
```
    // HINTS:
    //  - use fmodf(foo,b) to unwrap a radian angle measure float foo to range [0,b].
    //  - use the yaw control gain parameter kpYaw

    float yawRateCmd = 0;
    ////////////////////////////// BEGIN STUDENT CODE ////////////////////////////

    float yaw_error = yawCmd - yaw;
    yawRateCmd = yaw_error * kpYaw;

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

    return yawRateCmd;

}
```

6 IMPLEMENT CALCULATING THE MOTOR COMMANDS GIVEN COMMANDED THRUST AND MOMENTS IN C++.

Finally, we have the 'thrust regulator.' The code below assigns the desired thrusts to each motor, to achieve the commands given from the controllers above based on the actual physical variables of the drone. For example, it solves the problem of how much thurst need for rotor #2 to make a 90-degree yaw while gaining 3 feet in a 1 kg drone.

Without this part of the code, the rest of the code implemented above is obsolete. When disabling the GenerateMotorCommands function, we get the drones falling to the ground

without doing anything as shown below.




```
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 / sqrt(2); // perpendicular length to propeller


        float A = collThrustCmd;
        float B = momentCmd.x / l;
        float C = momentCmd.y / l;
        float D = -momentCmd.z / kappa;

        float F0 = (A + B + C + D) / 4.0;// front left
        float F1 = (A - B + C - D) / 4.0; // front right
        float F2 = (A + B - C - D) / 4.0; // rear left
        float F3 = (A - B - C + D) / 4.0; // rear right

        cmd.desiredThrustsN[0] = F0; // front left
        cmd.desiredThrustsN[1] = F1; // front right
        cmd.desiredThrustsN[2] = F2; // rear left
        cmd.desiredThrustsN[3] = F3; // rear right

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

        return cmd;
}
```