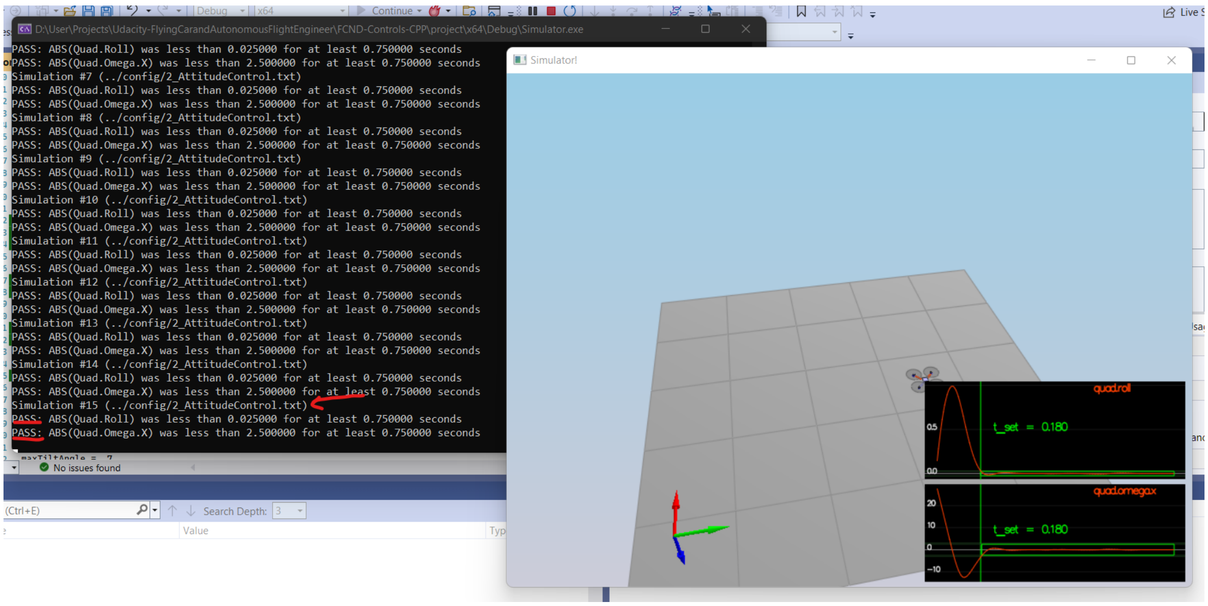
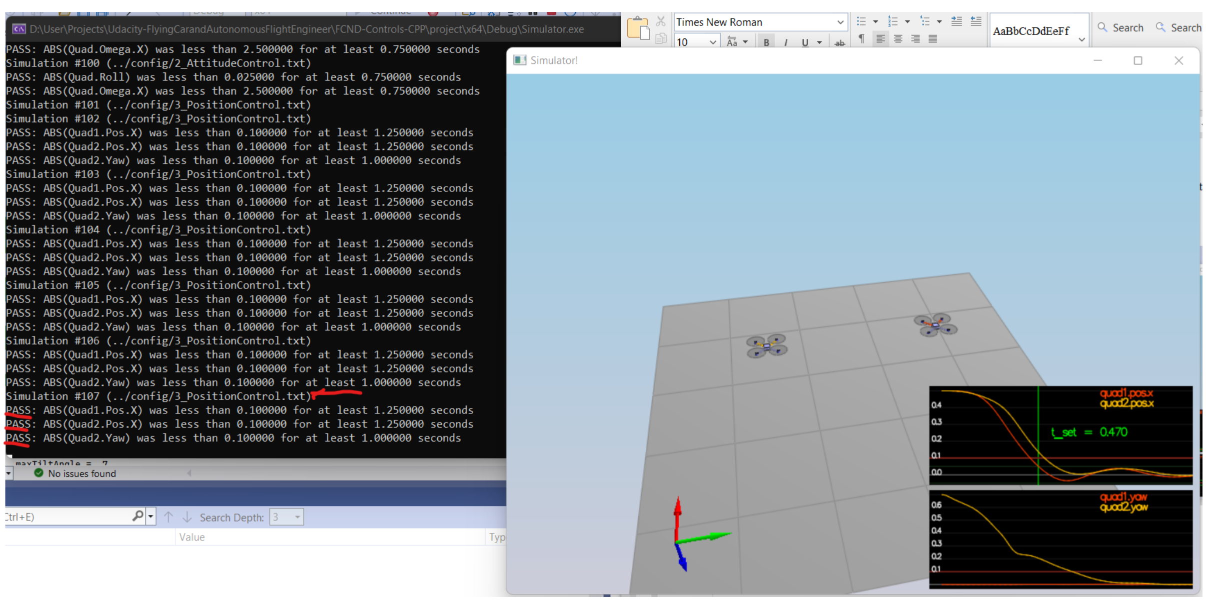
Udacity - Flying Cars and Autonomous Flight Engineer Nanodegree
Controls Project Write Up
Yu Hin Hau
12/13/2021

In this project, I implemented several PID controllers for a simulated drone and ran it through several test scenario.

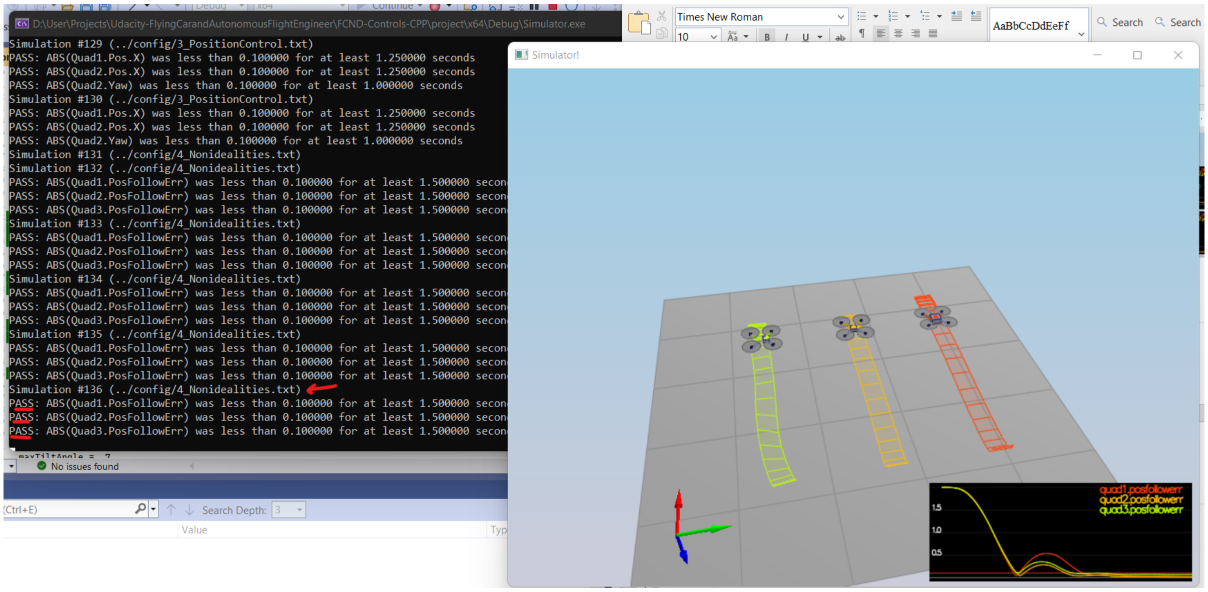
Scenario 2 - Attitude Control



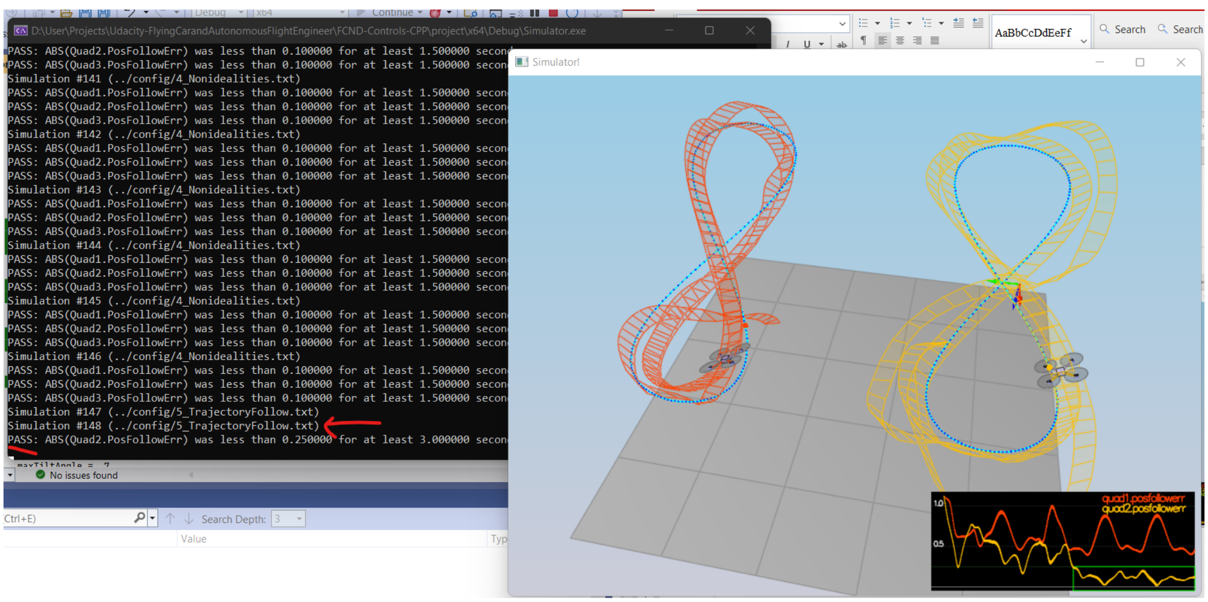
Scenario 3 - Position Control



Scenario 4 - Nonidealities



Scenario 5 - Tracking



Criteria 1 - Implement Body Rate Control

I implemented a linearlized version of Euler’s rotation dynamics equation here. From the target and actual body rate, I cauclated moment command.

```
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 error = pqrCmd - pqr;

    V3F MOI(this->Ixx, this->Iyy, this->Izz);

    momentCmd = MOI * kpPQR * error; //+ pqr.cross(MOI * pqr);

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

    return momentCmd;
}
```

Criteria 2 - Implement Roll Pitch Control

Here I implemented a P controller for angle and constrained the tilt angle to prevent the drone from flipping.

```
    ////////////////////////////// BEGIN STUDENT CODE ///////////////////////////
    // Exercise 4.2

    float b_x_c = accelCmd.x;
    float b_y_c = accelCmd.y;

    b_x_c = CONSTRAIN(b_x_c, -this->maxTiltAngle, this->maxTiltAngle);
    b_y_c = CONSTRAIN(b_y_c, -this->maxTiltAngle, this->maxTiltAngle);


    float b_x_a = R(0, 2);
    float b_y_a = R(1, 2);

    float b_dot_x_c = this->kpBank * (b_x_c - b_x_a);
    float b_dot_y_c = this->kpBank * (b_y_c - b_y_a);


    float p_c = (1.0 / R(2, 2)) * (R(1, 0) * b_dot_x_c - R(0, 0) * b_dot_y_c);
    float q_c = (1.0 / R(2, 2)) * (R(1, 1) * b_dot_x_c - R(0, 1) * b_dot_y_c);

    pqrCmd.x = p_c;
    pqrCmd.y = q_c;



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

    return pqrCmd;
}
```

Criteria 3 - Implement Altitude Control

I implemented a cascade PI-P controller for the altitdue and altitude rate and constrained the ascending/descending rate.

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


    float z_error = posZCmd - posZ;

    this->integratedAltitudeError += z_error * dt;
    //cout << this->integratedAltitudeError << endl;

    float h_dot_cmd = this->kpPosZ * z_error + this->KiPosZ * this->integratedAltitudeError
    h_dot_cmd = CONSTRAIN(h_dot_cmd, -this->maxDescentRate, this->maxAscentRate);

    float acceleration_cmd = this->kpVelZ * (h_dot_cmd - velZ) + accelZCmd;
    thrust = this->mass * -acceleration_cmd / R(2, 2);


    thrust = CONSTRAIN(thrust, this->minMotorThrust * 4.0, this->maxMotorThrust * 4.0);


    //cout << posZCmd << "\t" << posZ << "\t" << thrust << "\t" << this->kpPosZ << "\t" << t


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

    return thrust;
```

Criteria 4 - Implement Lateral Position Control

Here I implemented a P-P controller that control position and positon rate.

```
    V3F accelCmd = accelCmdFF;

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

    //posCmd.x = 3;
    //posCmd.y = -1;


    V3F u_vel = this->kpPosXY * (posCmd - pos) + velCmd;


    u_vel.x = CONSTRAIN(u_vel.x, -this->maxSpeedXY, this->maxSpeedXY);
    u_vel.y = CONSTRAIN(u_vel.y, -this->maxSpeedXY, this->maxSpeedXY);

    accelCmd = this->kpVelXY * -(u_vel - vel) + accelCmdFF;

    accelCmd.x = CONSTRAIN(accelCmd.x, -this->maxAccelXY, this->maxAccelXY);
    accelCmd.y = CONSTRAIN(accelCmd.y, -this->maxAccelXY, this->maxAccelXY);



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

    return accelCmd;
}
```

Criteria 5 - Implement Yaw Control

Here, I calculated the yaw error and implement a simple P controller.

```
    float yawRateCmd = 0;
    ////////////////////////////// BEGIN STUDENT CODE ///////////////////////////
    float error = fmodf(yawCmd, 100) - fmodf(yaw, 100);


    yawRateCmd = this->kpYaw * error;



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

    return yawRateCmd;
```

Criteria 6 - Motor Commands from Thrust and Moment

Motor commands are calculated from the 4 vehicle dynamics equation.

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




    float Fnet = collThrustCmd;
    float Mx = momentCmd.x;
    float My = momentCmd.y;
    float Mz = momentCmd.z;



    float F1 = (1 / 4.0) * ((1) * Fnet + (1 / this->L) * Mx + (1 / this->L) * My - (1 / this->kappa) * Mz);
    float F2 = (1 / 4.0) * ((1) * Fnet - (1 / this->L) * Mx + (1 / this->L) * My + (1 / this->kappa) * Mz);
    float F3 = (1 / 4.0) * ((1) * Fnet - (1 / this->L) * Mx - (1 / this->L) * My - (1 / this->kappa) * Mz);
    float F4 = (1 / 4.0) * ((1) * Fnet + (1 / this->L) * Mx - (1 / this->L) * My + (1 / this->kappa) * Mz);

    // Motor Ordering is Different in the C++ Simulator... remap to Udemy ordering
    cmd.desiredThrustsN[0] = CONSTRAIN(F1, this->minMotorThrust, this->maxMotorThrust);// front left
    cmd.desiredThrustsN[1] = CONSTRAIN(F2, this->minMotorThrust, this->maxMotorThrust); // front right
    cmd.desiredThrustsN[2] = CONSTRAIN(F4, this->minMotorThrust, this->maxMotorThrust); // rear left
    cmd.desiredThrustsN[3] = CONSTRAIN(F3, this->minMotorThrust, this->maxMotorThrust); // rear right


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

    return cmd;
}
```