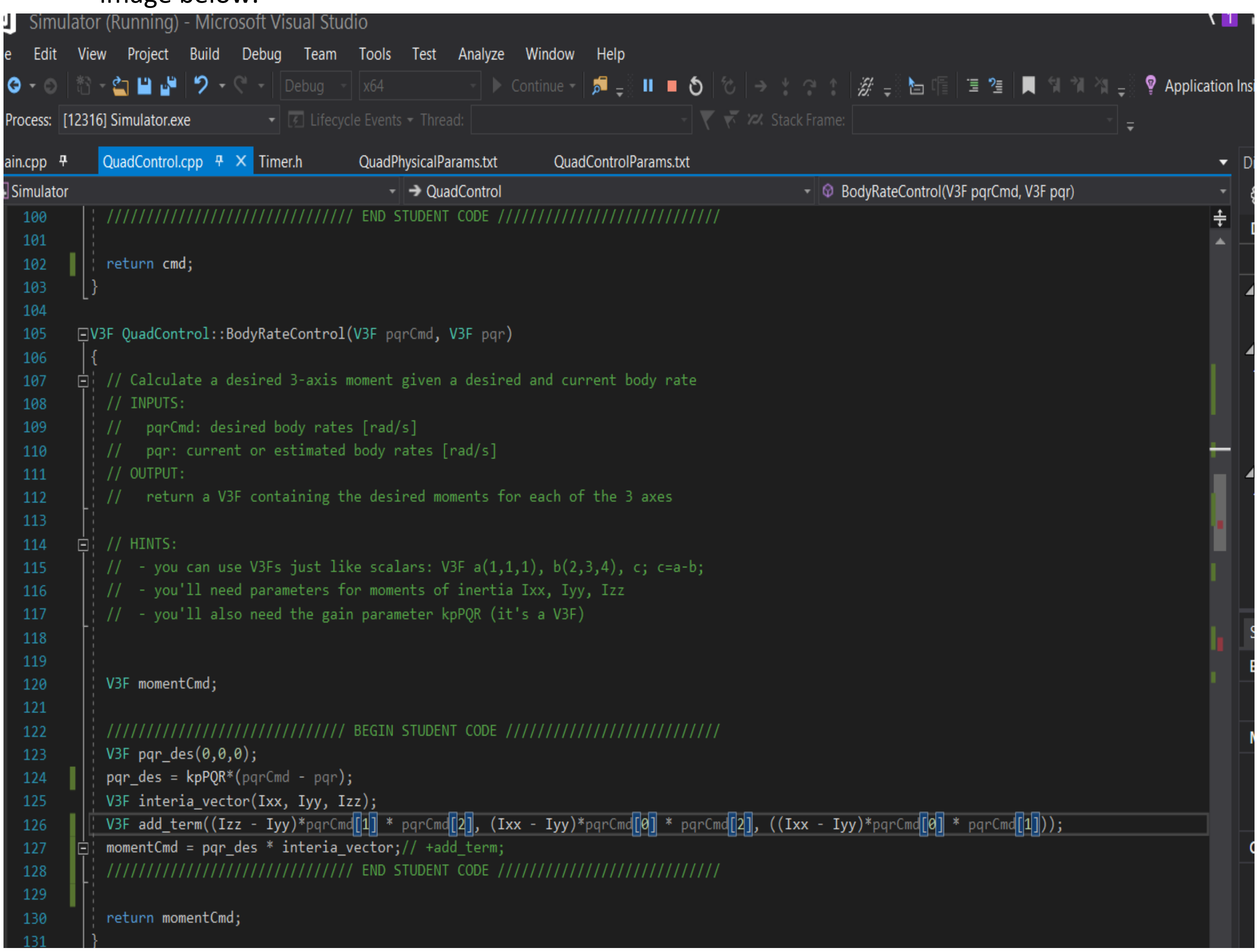
Provide a Writeup / README that includes all the rubric points and how you addressed each one. You can submit your writeup as markdown or pdf.

1. You are reading the write up ☺

Implemented body rate control in C++.

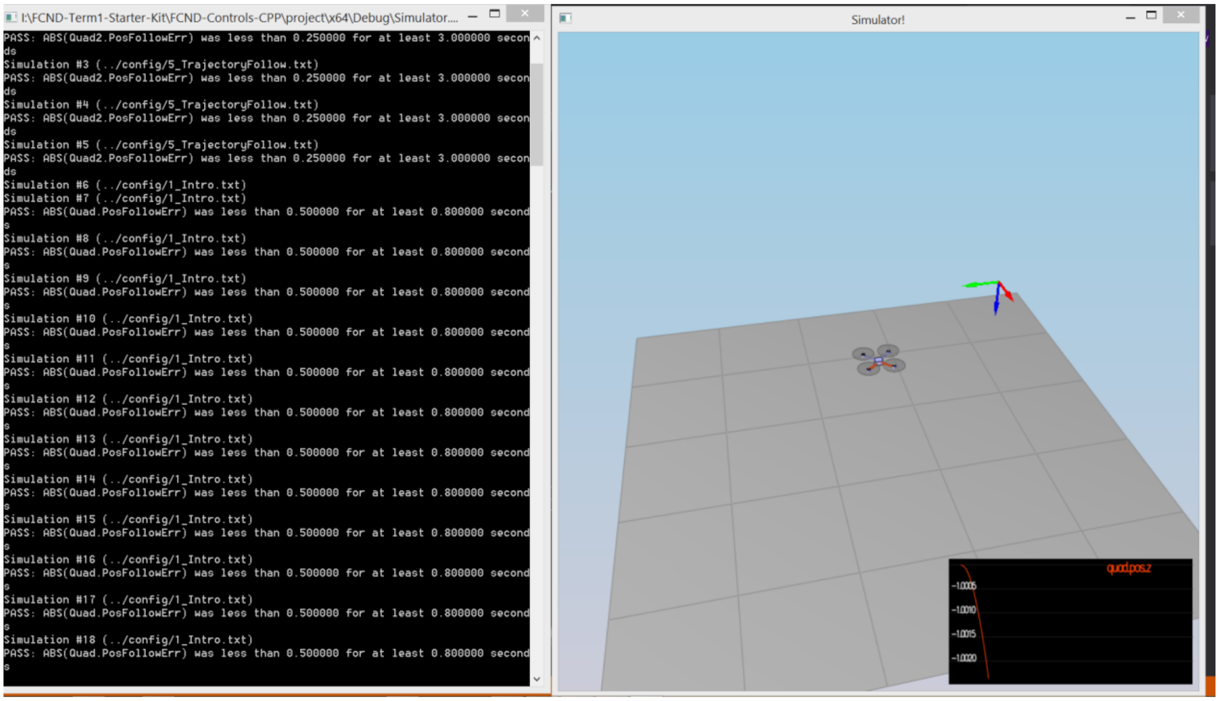
1. It was implemented using a P controller and the known dynamics of a quadrotor. **M** = **Ixomega_dot + omega(crossproduct)Ixomega**
2. Where **M** = Net Moment vector, **I** = inertia matrix, **omega_dot** = vector with the angular acceleration, **omega** = vector with the body rates.
3. Note, that the second term (**omega(cross)Ixomega)** in the equation in most cases can be ignored as it results in a very small number (in part due to the symmetry in the Ixx and Iyy components of the inertia matrix). The body rate controller was implemented as seen in the image below.

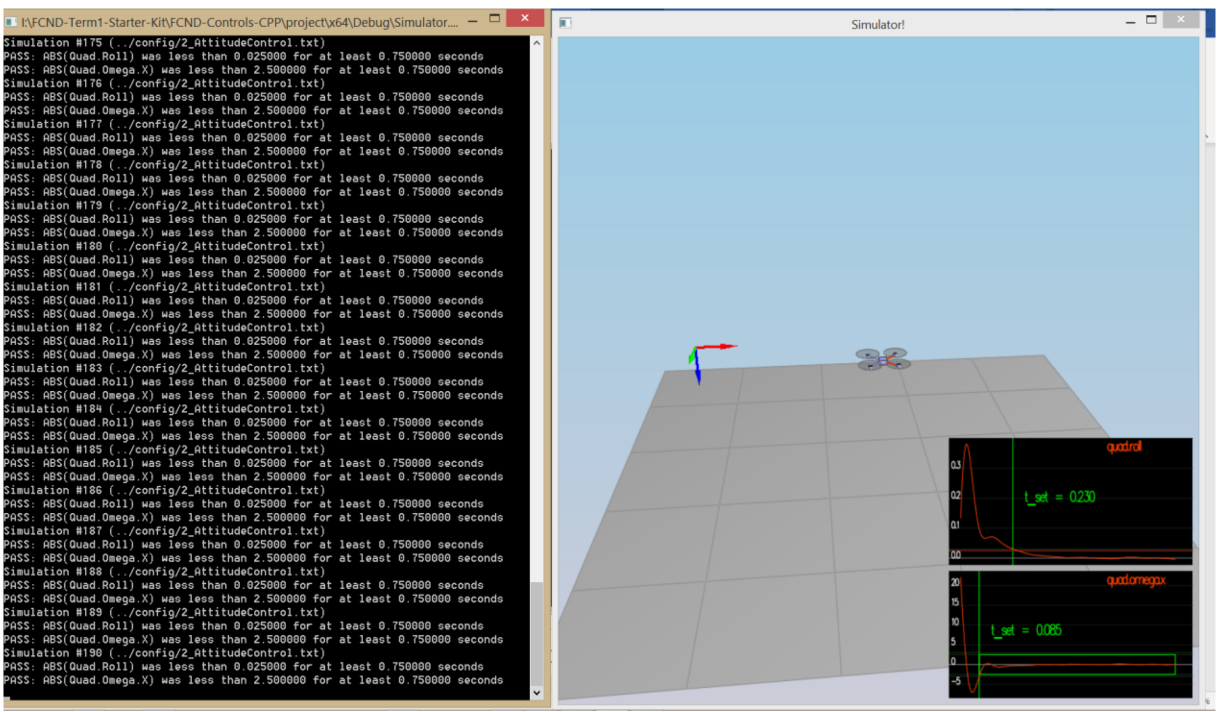


Implement roll pitch control in C++.

1. We divide the commanded acceleration by normalized commanded collective thrust (normalized by mass of the drone) to get the desired rotation angles in the x and y directions.
2. Constrained the desired rotation angles to be within the maximum tilt angle of the drone.
3. Used a P controller to get the desired rotation rates in the x and y directions. (roll and pitch)
4. Used provided rotation matrix to convert desired rotation rates in the inertial (world) frame to desired rotation rates in the body frame.
5. Image below shows code:

```cpp
//   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)

// HINTS:
//  - we already provide rotation matrix R: to get element R[1,2] (python) use R(1,2) (C++)
//  - you'll need the roll/pitch gain kpBank
//  - collThrustCmd is a force in Newtons! You'll likely want to convert it to acceleration first

V3F pqrCmd;
Mat3x3F R = attitude.RotationMatrix_IwrtB();

////////////////////////////// BEGIN STUDENT CODE ///////////////////////////
V3F b_cmd;
b_cmd = accelCmd / (collThrustCmd/mass);
float max_radians = maxTiltAngle;
b_cmd.x = CONSTRAIN(b_cmd.x, -max_radians, max_radians);
b_cmd.y = CONSTRAIN(b_cmd.y, -max_radians, max_radians);
V3F b_dot_cmd;
b_dot_cmd[0] = kpBank * (b_cmd[0] - R(0, 2));
b_dot_cmd[1] = kpBank * (b_cmd[1] - R(1, 2));

pqrCmd[0] = R(1,0)* b_dot_cmd[0] - R(0,0)*b_dot_cmd[1];
pqrCmd[1] = R(1, 1)*b_dot_cmd[0] - R(0, 1)*b_dot_cmd[1];

pqrCmd[0] = pqrCmd[0]  / R(2, 2);
pqrCmd[1] = pqrCmd[1]  / R(2, 2);

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


return pqrCmd;
}
```





Implement altitude controller in C++.

1. Constrained commanded velocity to be between the max ascent and descent rate of the quad.
2. Used a PID controller to calculate the desired acceleration in the z direction. `integratedAltitudeError` is the integrated term that allows for the

system to compensate for systematic bias or errors in z direction such as underestimated mass.

3. Used Newtonian equations and rotation values, F= m*(g – c)/b_{33} , of vehicle to derive the required thrust (Force) necessary in the body frame to achieve the desired acceleration in world frame. F = desired thrust, m = mass, g is acceleration due to gravity, c is commanded acceleration, and b_{33} is a rotation matrix element. Image below shows code:

```
QuadControl                                  YawControl(float yawCmd, float yaw)
}

float QuadControl::AltitudeControl(float posZCmd, float velZCmd, float posZ, float velZ, Quaternion<float> attitude, float accelZCmd, float
{
  // 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 z_cmd;
  velZCmd = CONSTRAIN(velZCmd, -maxAscentRate, maxDescentRate);
  z_cmd = accelZCmd + kpPosZ * ( posZCmd - posZ) + kpVelZ*( velZCmd - velZ) + KiPosZ*integratedAltitudeError;
  integratedAltitudeError += (posZCmd - posZ)*dt;
  thrust = mass * (CONST_GRAVITY - z_cmd) / (R(2, 2));
  ///////////////////////////// END STUDENT CODE ////////////////////////////

  return thrust;
}
```

Implement lateral position control in C++.

1. Limited commanded velocity to vehicle constraints
2. Used a PD controller to derive commanded/desired acceleration.
3. Image below shows code:

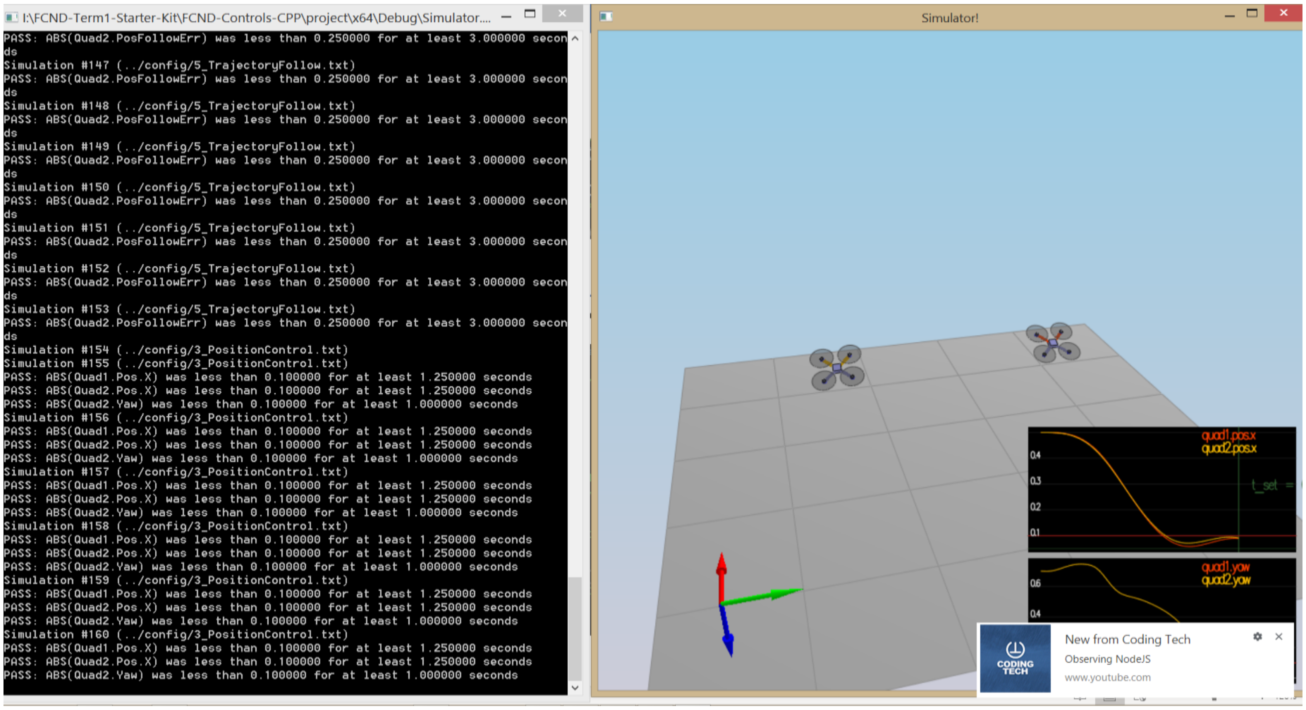
```
QuadControl    LateralPositionControl(V3F posCmd, V3F velCmd, V3F pos, V3F v
  // Calculate a desired horizontal acceleration based on
  //  desired lateral position/velocity/acceleration and current pose
  // INPUTS:
  //   posCmd: desired position, in NED [m]
  //   velCmd: desired velocity, in 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 ///////////////////////////
  velCmd.x = CONSTRAIN(velCmd.x, -maxSpeedXY, maxSpeedXY);
  velCmd.y = CONSTRAIN(velCmd.y, -maxSpeedXY, maxSpeedXY);
  accelCmd.x = accelCmdFF.x - kpPosXY * (posCmd.x - pos.x) - kpVelXY * (velCmd.x - vel.x);
  accelCmd.y = accelCmdFF.y - kpPosXY * (posCmd.y - pos.y) - kpVelXY * (velCmd.y  - vel.y);
  accelCmd.x = CONSTRAIN(accelCmd.x, -maxAccelXY, maxAccelXY);
  accelCmd.y = CONSTRAIN(accelCmd.y, -maxAccelXY, maxAccelXY);
  ///////////////////////////// END STUDENT CODE ////////////////////////////
  return accelCmd;
}
```

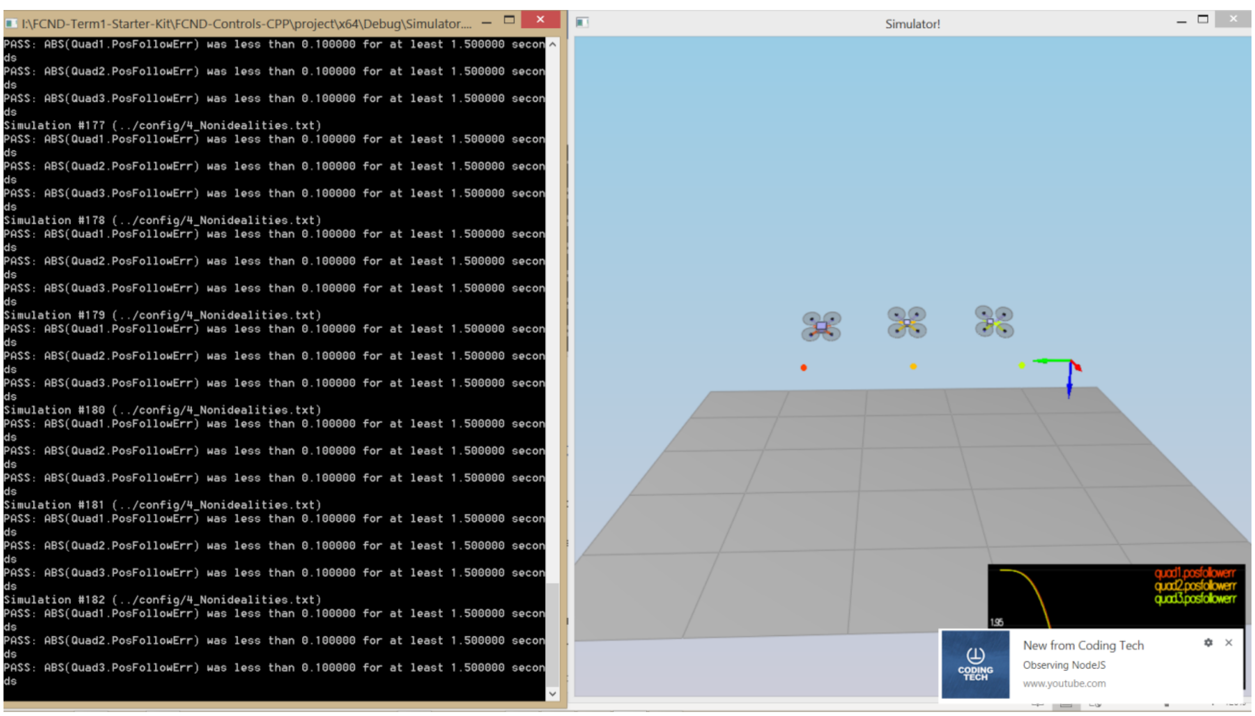
Implement yaw control in C++.

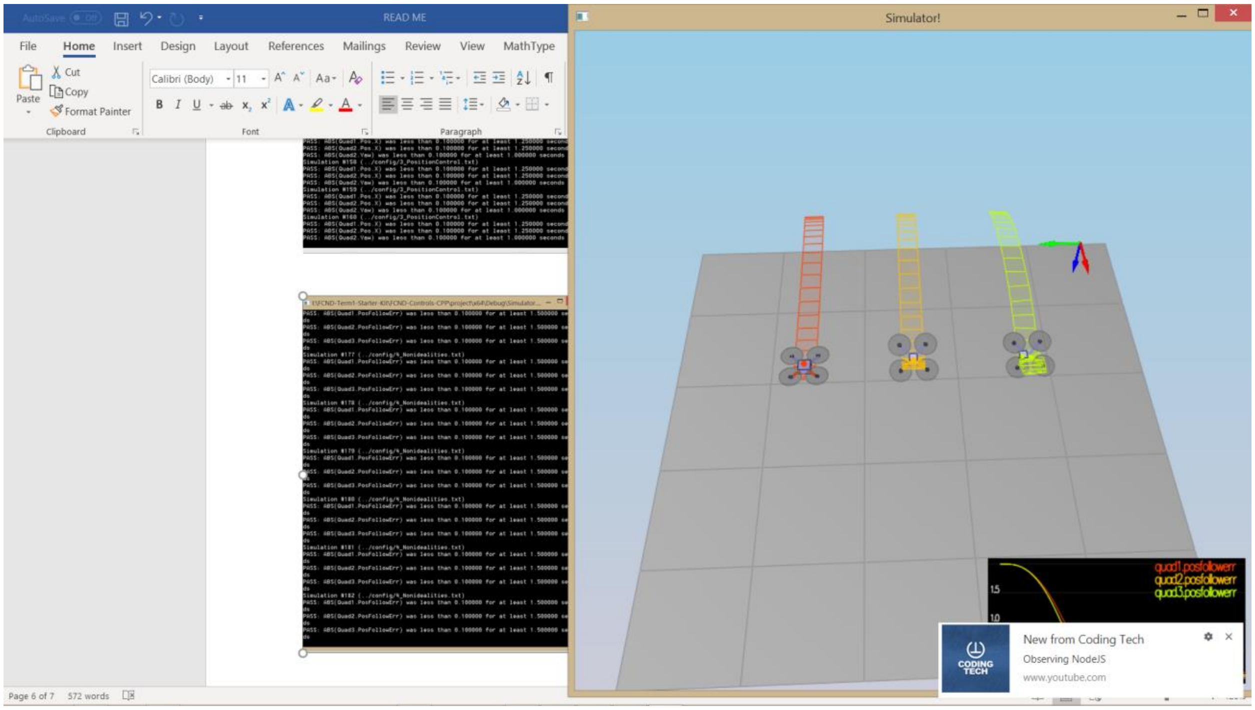
1. Used a P controller to determine the desired yaw rates. I also wrapped the yaw rates to in fmod to output yaw rates between 0 and 180 degrees (pi/2).

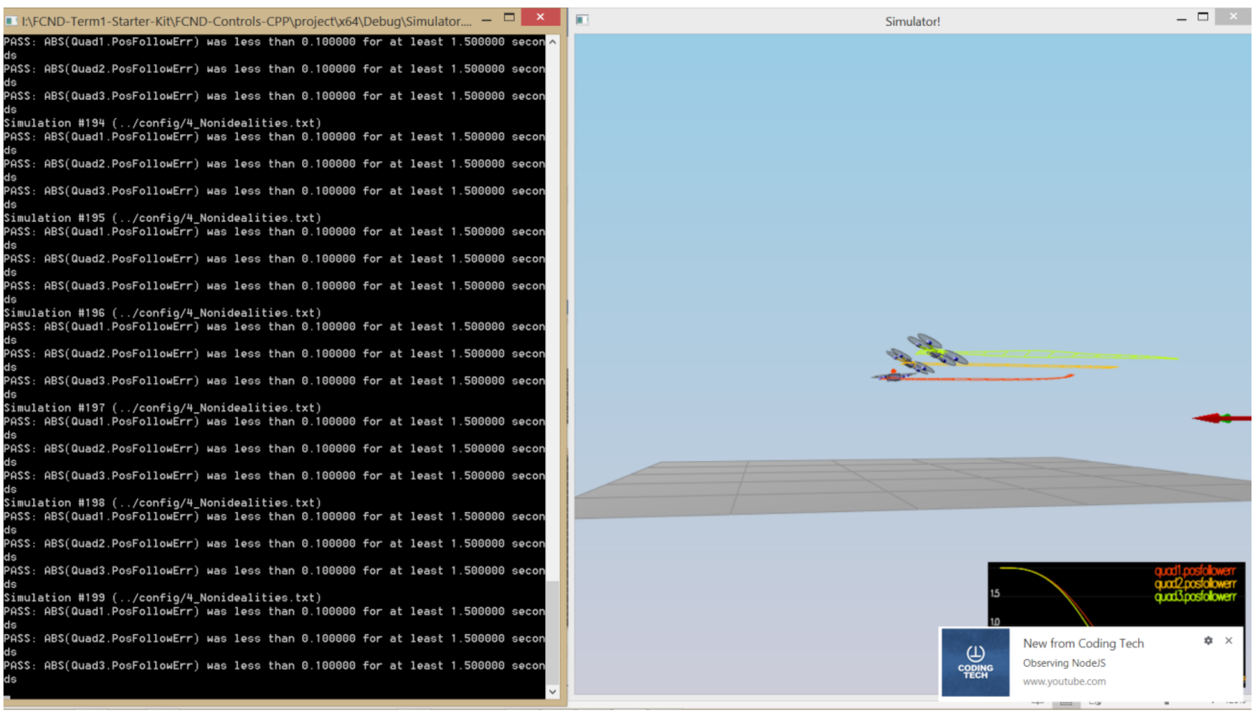
```
float QuadControl::YawControl(float yawCmd, float yaw)
{
  // Calculate a desired yaw rate to control yaw to yawCmd
  // INPUTS:
  //   yawCmd: commanded yaw [rad]
  //   yaw: current yaw [rad]
  // OUTPUT:
  //   return a desired yaw rate [rad/s]
  // HINTS:
  //  - use fmodf(foo,b) to unwrap a radian angle measure float foo to range [0,b].
  //  - use the yaw control gain parameter kpYaw

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


  yawRateCmd = kpYaw * ( fmodf(yawCmd, 3.142/2) - fmodf(yaw, 3.142/2) );
  ///////////////////////////// END STUDENT CODE ////////////////////////////

  return yawRateCmd;
```

2.


I:\FCND-Term1-Starter-Kit\FCND-Controls-CPP\project\x64\Debug\Simulator....
PASS: ABS(Quad2.PosFollowErr) was less than 0.250000 for at least 3.000000 seconds
Simulation #147 (../config/5_TrajectoryFollow.txt)
PASS: ABS(Quad2.PosFollowErr) was less than 0.250000 for at least 3.000000 seconds
Simulation #148 (../config/5_TrajectoryFollow.txt)
PASS: ABS(Quad2.PosFollowErr) was less than 0.250000 for at least 3.000000 seconds
Simulation #149 (../config/5_TrajectoryFollow.txt)
PASS: ABS(Quad2.PosFollowErr) was less than 0.250000 for at least 3.000000 seconds
Simulation #150 (../config/5_TrajectoryFollow.txt)
PASS: ABS(Quad2.PosFollowErr) was less than 0.250000 for at least 3.000000 seconds
Simulation #151 (../config/5_TrajectoryFollow.txt)
PASS: ABS(Quad2.PosFollowErr) was less than 0.250000 for at least 3.000000 seconds
Simulation #152 (../config/5_TrajectoryFollow.txt)
PASS: ABS(Quad2.PosFollowErr) was less than 0.250000 for at least 3.000000 seconds
Simulation #153 (../config/5_TrajectoryFollow.txt)
PASS: ABS(Quad2.PosFollowErr) was less than 0.250000 for at least 3.000000 seconds
Simulation #154 (../config/3_PositionControl.txt)
Simulation #155 (../config/3_PositionControl.txt)
PASS: ABS(Quad1.Pos.X) was less than 0.100000 for at least 1.250000 seconds
PASS: ABS(Quad2.Pos.X) was less than 0.100000 for at least 1.250000 seconds
PASS: ABS(Quad2.Yaw) was less than 0.100000 for at least 1.000000 seconds
Simulation #156 (../config/3_PositionControl.txt)
PASS: ABS(Quad1.Pos.X) was less than 0.100000 for at least 1.250000 seconds
PASS: ABS(Quad2.Pos.X) was less than 0.100000 for at least 1.250000 seconds
PASS: ABS(Quad2.Yaw) was less than 0.100000 for at least 1.000000 seconds
Simulation #157 (../config/3_PositionControl.txt)
PASS: ABS(Quad1.Pos.X) was less than 0.100000 for at least 1.250000 seconds
PASS: ABS(Quad2.Pos.X) was less than 0.100000 for at least 1.250000 seconds
PASS: ABS(Quad2.Yaw) was less than 0.100000 for at least 1.000000 seconds
Simulation #158 (../config/3_PositionControl.txt)
PASS: ABS(Quad1.Pos.X) was less than 0.100000 for at least 1.250000 seconds
PASS: ABS(Quad2.Pos.X) was less than 0.100000 for at least 1.250000 seconds
PASS: ABS(Quad2.Yaw) was less than 0.100000 for at least 1.000000 seconds
Simulation #159 (../config/3_PositionControl.txt)
PASS: ABS(Quad1.Pos.X) was less than 0.100000 for at least 1.250000 seconds
PASS: ABS(Quad2.Pos.X) was less than 0.100000 for at least 1.250000 seconds
PASS: ABS(Quad2.Yaw) was less than 0.100000 for at least 1.000000 seconds
Simulation #160 (../config/3_PositionControl.txt)
PASS: ABS(Quad1.Pos.X) was less than 0.100000 for at least 1.250000 seconds
PASS: ABS(Quad2.Pos.X) was less than 0.100000 for at least 1.250000 seconds
PASS: ABS(Quad2.Yaw) was less than 0.100000 for at least 1.000000 seconds
Simulator!
quad1.pos.x
quad2.pos.x
quad1.yaw
quad2.yaw
New from Coding Tech
Observing NodeJS
www.youtube.com


I:\FCND-Term1-Starter-Kit\FCND-Controls-CPP\project\x64\Debug\Simulator....
PASS: ABS(Quad1.PosFollowErr) was less than 0.100000 for at least 1.500000 seconds
PASS: ABS(Quad2.PosFollowErr) was less than 0.100000 for at least 1.500000 seconds
PASS: ABS(Quad3.PosFollowErr) was less than 0.100000 for at least 1.500000 seconds
Simulation #177 (../config/4_Nonidealities.txt)
PASS: ABS(Quad1.PosFollowErr) was less than 0.100000 for at least 1.500000 seconds
PASS: ABS(Quad2.PosFollowErr) was less than 0.100000 for at least 1.500000 seconds
PASS: ABS(Quad3.PosFollowErr) was less than 0.100000 for at least 1.500000 seconds
Simulation #178 (../config/4_Nonidealities.txt)
PASS: ABS(Quad1.PosFollowErr) was less than 0.100000 for at least 1.500000 seconds
PASS: ABS(Quad2.PosFollowErr) was less than 0.100000 for at least 1.500000 seconds
PASS: ABS(Quad3.PosFollowErr) was less than 0.100000 for at least 1.500000 seconds
Simulation #179 (../config/4_Nonidealities.txt)
PASS: ABS(Quad1.PosFollowErr) was less than 0.100000 for at least 1.500000 seconds
PASS: ABS(Quad2.PosFollowErr) was less than 0.100000 for at least 1.500000 seconds
PASS: ABS(Quad3.PosFollowErr) was less than 0.100000 for at least 1.500000 seconds
Simulation #180 (../config/4_Nonidealities.txt)
PASS: ABS(Quad1.PosFollowErr) was less than 0.100000 for at least 1.500000 seconds
PASS: ABS(Quad2.PosFollowErr) was less than 0.100000 for at least 1.500000 seconds
PASS: ABS(Quad3.PosFollowErr) was less than 0.100000 for at least 1.500000 seconds
Simulation #181 (../config/4_Nonidealities.txt)
PASS: ABS(Quad1.PosFollowErr) was less than 0.100000 for at least 1.500000 seconds
PASS: ABS(Quad2.PosFollowErr) was less than 0.100000 for at least 1.500000 seconds
PASS: ABS(Quad3.PosFollowErr) was less than 0.100000 for at least 1.500000 seconds
Simulation #182 (../config/4_Nonidealities.txt)
PASS: ABS(Quad1.PosFollowErr) was less than 0.100000 for at least 1.500000 seconds
PASS: ABS(Quad2.PosFollowErr) was less than 0.100000 for at least 1.500000 seconds
PASS: ABS(Quad3.PosFollowErr) was less than 0.100000 for at least 1.500000 seconds
Simulator!
quad1.posfollowerr
quad2.posfollowerr
quad3.posfollowerr
New from Coding Tech
Observing NodeJS
www.youtube.com





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

1. l = Perpendicular distance of the rotor to the X axis = L /sqrt(2). Where L = length of rotor arm from center of mass.
2. Solved the following system of equations to determine the necessary thrust force for each rotor.
3. F1 + F2 + F3 + F4 = C

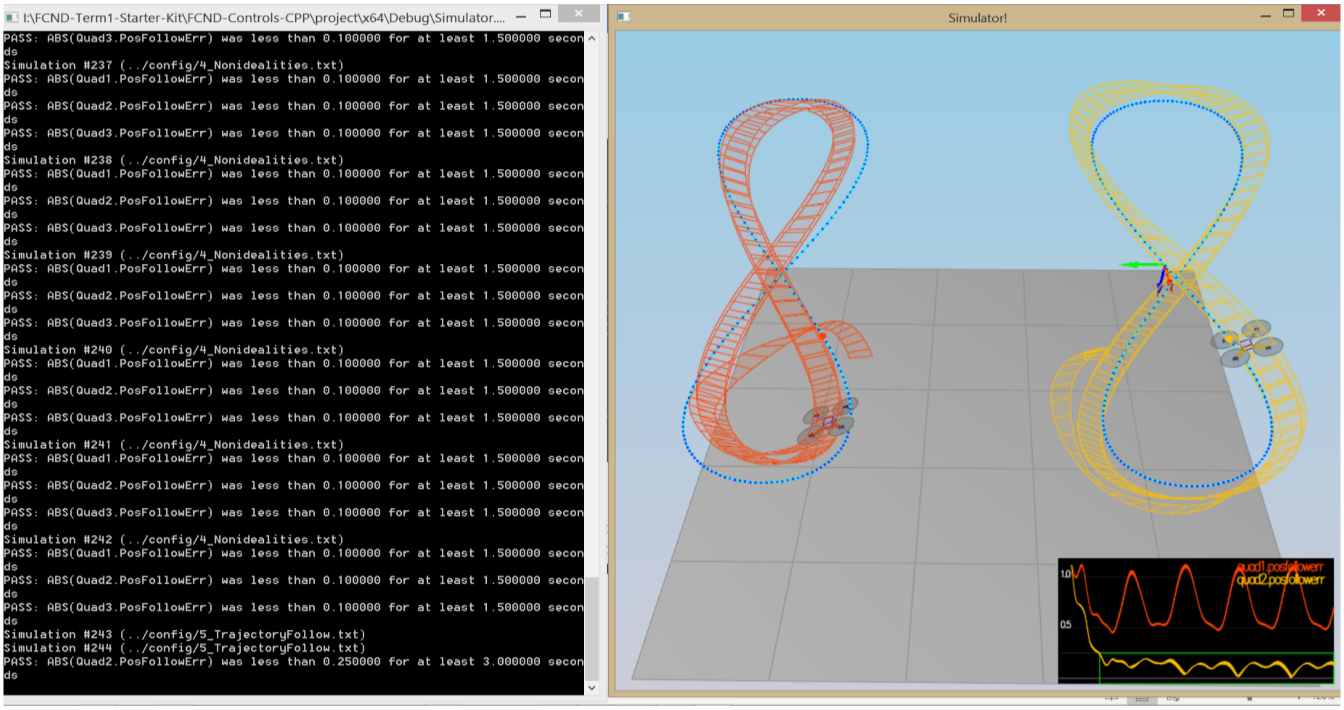
4. l(F1 - F2 - F3 + F4) = Mx
5. l(F1 + F2 - F3 - F4) = My
6. -kappa*F1 + kappa*F2 – kappa*F3 + kappa*F4 = Mz
7. Where Fi = Thrust force for rotor i where i is an element of of set {1, 2, 3, 4}, C = collective thrust, Mi is the moment around axis I where I an element of the set {x,y,z}, kappa = r `ratio between thrust [N] and torque due to drag [N m].`
8. Code can be seen below:

QuadControl | YawControl(float yawCmd, float yaw)

```
    ////////////////////////////// BEGIN STUDENT CODE ///////////////////////////
    float l;
    l = L / sqrt(2);
    float F_4;
    float F_3;
    float F_2;
    float F_1;

    F_4 = 0.25*(collThrustCmd + (momentCmd.x / l) - (momentCmd.y / l) + (momentCmd.z / (kappa)));
    //F_4 = CONSTRAIN(F_4, 0, 4.5 );
    F_3 = 0.5*(collThrustCmd - momentCmd.y / l) - F_4;
    //F_3 = CONSTRAIN(F_3, 0, 4.5);
    F_2 = 0.5*(collThrustCmd - momentCmd.x / l) - F_3;
    //F_2 = CONSTRAIN(F_2, 0, 4.5);
    F_1 = collThrustCmd - F_2 - F_3 - F_4;
   // F_1 = CONSTRAIN(F_1, 0, 4.5);


        //cmd.desiredThrustsN[0] = mass * 9.81f / 4.f; // front left
        //cmd.desiredThrustsN[1] = mass * 9.81f / 4.f; // front right
        //cmd.desiredThrustsN[2] = mass * 9.81f / 4.f; // rear left
        //cmd.desiredThrustsN[3] = mass * 9.81f / 4.f; // rear right


    cmd.desiredThrustsN[0] = F_1; // front left
    cmd.desiredThrustsN[1] = F_2; // front right
    cmd.desiredThrustsN[2] = F_4; // rear left
    cmd.desiredThrustsN[3] = F_3; // rear right

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

    return cmd;
}
```

Your C++ controller is successfully able to fly the provided test trajectory and visually passes inspection of the scenarios leading up to the test trajectory.

1. Controller was able to fly the provided test trajectory and visually passed scenarios leading up to the test trajectory.

2. 
I:\FCND-Term1-Starter-Kit\FCND-Controls-CPP\project\x64\Debug\Simulator....
PASS: ABS(Quad3.PosFollowErr) was less than 0.100000 for at least 1.500000 seconds
Simulation #237 (../config/4_Nonidealities.txt)
PASS: ABS(Quad1.PosFollowErr) was less than 0.100000 for at least 1.500000 seconds
PASS: ABS(Quad2.PosFollowErr) was less than 0.100000 for at least 1.500000 seconds
PASS: ABS(Quad3.PosFollowErr) was less than 0.100000 for at least 1.500000 seconds
Simulation #238 (../config/4_Nonidealities.txt)
PASS: ABS(Quad1.PosFollowErr) was less than 0.100000 for at least 1.500000 seconds
PASS: ABS(Quad2.PosFollowErr) was less than 0.100000 for at least 1.500000 seconds
PASS: ABS(Quad3.PosFollowErr) was less than 0.100000 for at least 1.500000 seconds
Simulation #239 (../config/4_Nonidealities.txt)
PASS: ABS(Quad1.PosFollowErr) was less than 0.100000 for at least 1.500000 seconds
PASS: ABS(Quad2.PosFollowErr) was less than 0.100000 for at least 1.500000 seconds
PASS: ABS(Quad3.PosFollowErr) was less than 0.100000 for at least 1.500000 seconds
Simulation #240 (../config/4_Nonidealities.txt)
PASS: ABS(Quad1.PosFollowErr) was less than 0.100000 for at least 1.500000 seconds
PASS: ABS(Quad2.PosFollowErr) was less than 0.100000 for at least 1.500000 seconds
PASS: ABS(Quad3.PosFollowErr) was less than 0.100000 for at least 1.500000 seconds
Simulation #241 (../config/4_Nonidealities.txt)
PASS: ABS(Quad1.PosFollowErr) was less than 0.100000 for at least 1.500000 seconds
PASS: ABS(Quad2.PosFollowErr) was less than 0.100000 for at least 1.500000 seconds
PASS: ABS(Quad3.PosFollowErr) was less than 0.100000 for at least 1.500000 seconds
Simulation #242 (../config/4_Nonidealities.txt)
PASS: ABS(Quad1.PosFollowErr) was less than 0.100000 for at least 1.500000 seconds
PASS: ABS(Quad2.PosFollowErr) was less than 0.100000 for at least 1.500000 seconds
PASS: ABS(Quad3.PosFollowErr) was less than 0.100000 for at least 1.500000 seconds
Simulation #243 (../config/5_TrajectoryFollow.txt)
Simulation #244 (../config/5_TrajectoryFollow.txt)
PASS: ABS(Quad2.PosFollowErr) was less than 0.250000 for at least 3.000000 seconds
Simulator!
quad1.posfollowerr
quad2.posfollowerr
1.0
0.5