SETUP

Setup is slightly cumbersome, please refer to the Official README that has detailed setup instructions here: https://github.com/udacity/FCND-Controls-CPP

1. Implemented body rate control in C++

Values tuned to: kpPQR = 90, 90, 6

```
V3F kpPos;
kpPos.x = kpPosXY;
kpPos.y = kpPosXY;
kpPos.z = 0.f;

V3F kpVel;
kpVel.x = kpVelXY;
kpVel.y = kpVelXY;
kpVel.z = 0.f;

V3F capVelCmd;
if ( velCmd.mag() > maxSpeedXY ) {
  capVelCmd = velCmd.norm() * maxSpeedXY;
} else {
  capVelCmd = velCmd;
}

  accelCmd = kpPosZ * ( posCmd - pos ) + kpVelZ * ( capVelCmd - vel ) + accelCmd;

if ( accelCmd.mag() > maxAccelXY ) {
  accelCmd = accelCmd.norm() * maxAccelXY;
}
```

2. Implement roll pitch control in C++

Tuned Parameter kpBank = 11

```cpp
// returns a desired roll and pitch rate
V3F QuadControl::RollPitchControl(V3F accelCmd, Quaternion<float> attitude, float collThrustCmd)
{
  // Calculate a desired pitch and roll angle rates based on a desired global
  //   lateral acceleration, the current attitude of the quad, and desired
  //   collective thrust command
  // 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)

  // 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 ///////////////////////////

    if ( collThrustCmd > 0 ) {
    float c =  - collThrustCmd / mass;
    float b_x, b_y;
    float b_x_d, b_y_d;
    b_x = CONSTRAIN((accelCmd.x / c), -sin(maxTiltAngle), sin(maxTiltAngle));
    b_y = CONSTRAIN((accelCmd.y / c), -sin(maxTiltAngle), sin(maxTiltAngle));
    b_x_d = kpBank * (b_x - R(0,2));
    b_y_d = kpBank * (b_y - R(1,2));

    pqrCmd.x = (R(1,0) * b_x_d - R(0,0) * b_y_d) / R(2,2);
    pqrCmd.y = (R(1,1) * b_x_d - R(0,1) * b_y_d) / R(2,2);
  } else {
    pqrCmd.x = 0.0;
    pqrCmd.y = 0.0;
  }

  pqrCmd.z = 0;



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

  return pqrCmd;
}
```

3. Implement Altitude Controller in C++

```cpp
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 z_e = posZCmd - posZ;
  float p_t = kpPosZ * z_e;

  float z_d_e = velZCmd - velZ;
  integratedAltitudeError += z_e * dt;


  float d_t = kpVelZ * z_d_e + velZ;
  float i_t = KiPosZ * integratedAltitudeError;
  float b_z = R(2,2);

  float u_1_b = p_t + d_t + i_t + accelZCmd;

  float a = ( u_1_b - CONST_GRAVITY ) / b_z;

  thrust = - mass * CONSTRAIN(a, - maxAscentRate / dt, maxAscentRate / dt);




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

  return thrust;
}
```

Tuned Parameter
# Position control gains
kpPosXY = 30
kpPosZ = 30
KiPosZ = 30

# Velocity control gains
kpVelXY = 10.5
kpVelZ = 10.5

4. Implement Lateral Position Control in C++

```
V3F kpPos;
kpPos.x = kpPosXY;
kpPos.y = kpPosXY;
kpPos.z = 0.f;

V3F kpVel;
kpVel.x = kpVelXY;
kpVel.y = kpVelXY;
kpVel.z = 0.f;

V3F capVelCmd;
if ( velCmd.mag() > maxSpeedXY ) {
  capVelCmd = velCmd.norm() * maxSpeedXY;
} else {
  capVelCmd = velCmd;
}

  accelCmd = kpPosZ * ( posCmd - pos ) + kpVelZ * ( capVelCmd - vel ) + accelCmd;

if ( accelCmd.mag() > maxAccelXY ) {
  accelCmd = accelCmd.norm() * maxAccelXY;
}
```

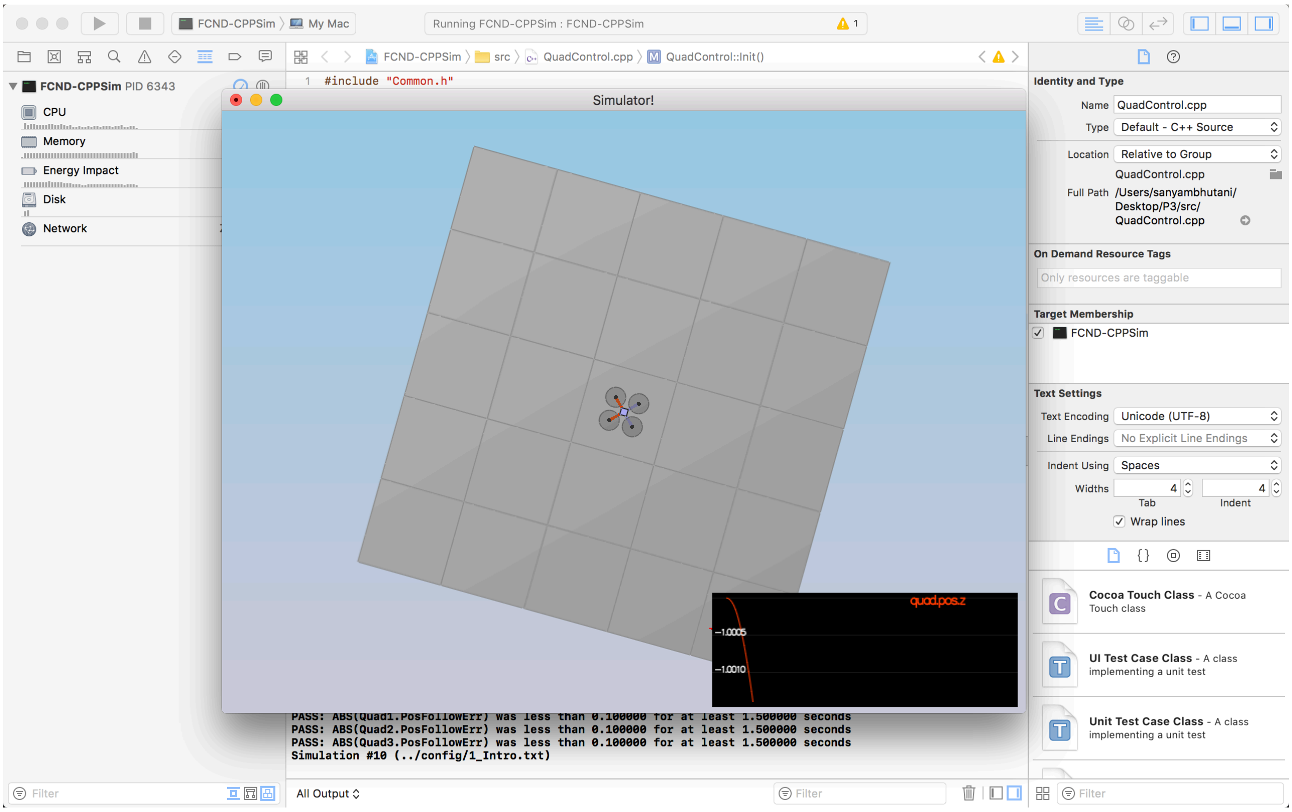
5. Implement Yaw Control in C++

```
// returns desired yaw rate
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_c_2_pi = 0;
  if ( yawCmd > 0 ) {
    yaw_c_2_pi = fmodf(yawCmd, 2 * F_PI);
  } else {
    yaw_c_2_pi = -fmodf(-yawCmd, 2 * F_PI);
  }
  float err = yaw_c_2_pi - yaw;
  if ( err > F_PI ) {
    err -= 2 * F_PI;
  } if ( err < -F_PI ) {
    err += 2 * F_PI;
  }
  yawRateCmd = kpYaw * err;


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

  return yawRateCmd;

}
```

OUTPUTS

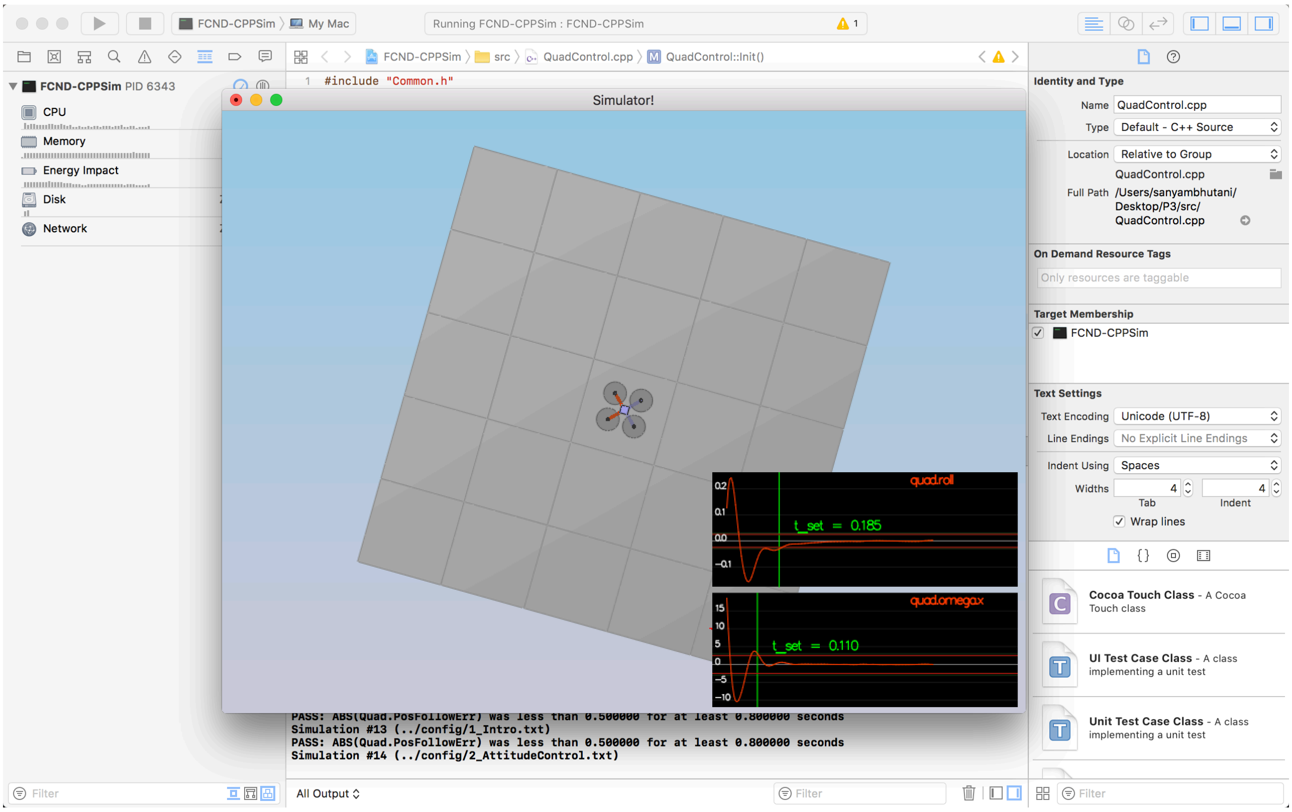
Scenario 1:

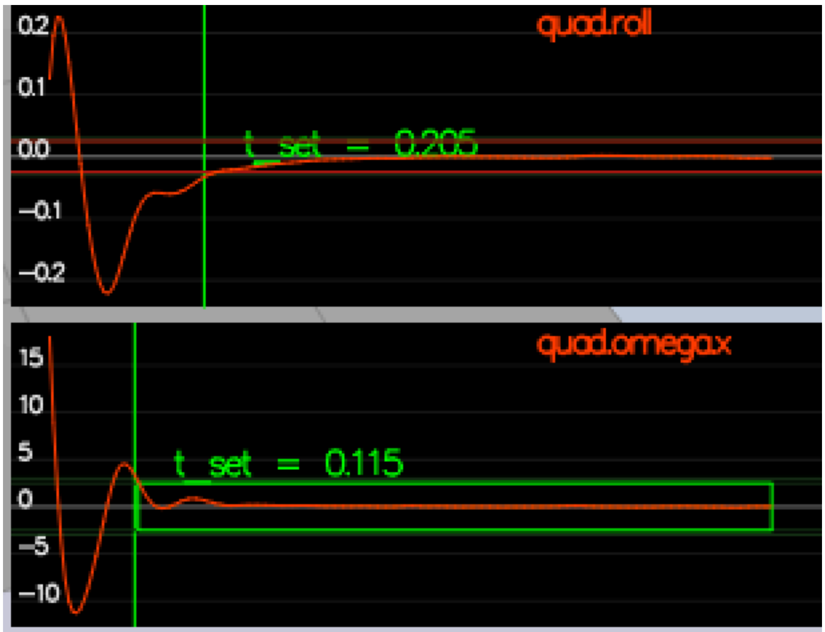
Tuning the Weight of the Quad Allows it to float.




```
Simulation #13 (../config/1_Intro.txt)
PASS: ABS(Quad.PosFollowErr) was less than 0.500000 for at least 0.800000 seconds
Simulation #14 (../config/1_Intro.txt)
PASS: ABS(Quad.PosFollowErr) was less than 0.500000 for at least 0.800000 seconds
Simulation #15 (../config/1_Intro.txt)
PASS: ABS(Quad.PosFollowErr) was less than 0.500000 for at least 0.800000 seconds
Simulation #16 (../config/1_Intro.txt)
PASS: ABS(Quad.PosFollowErr) was less than 0.500000 for at least 0.800000 seconds
```

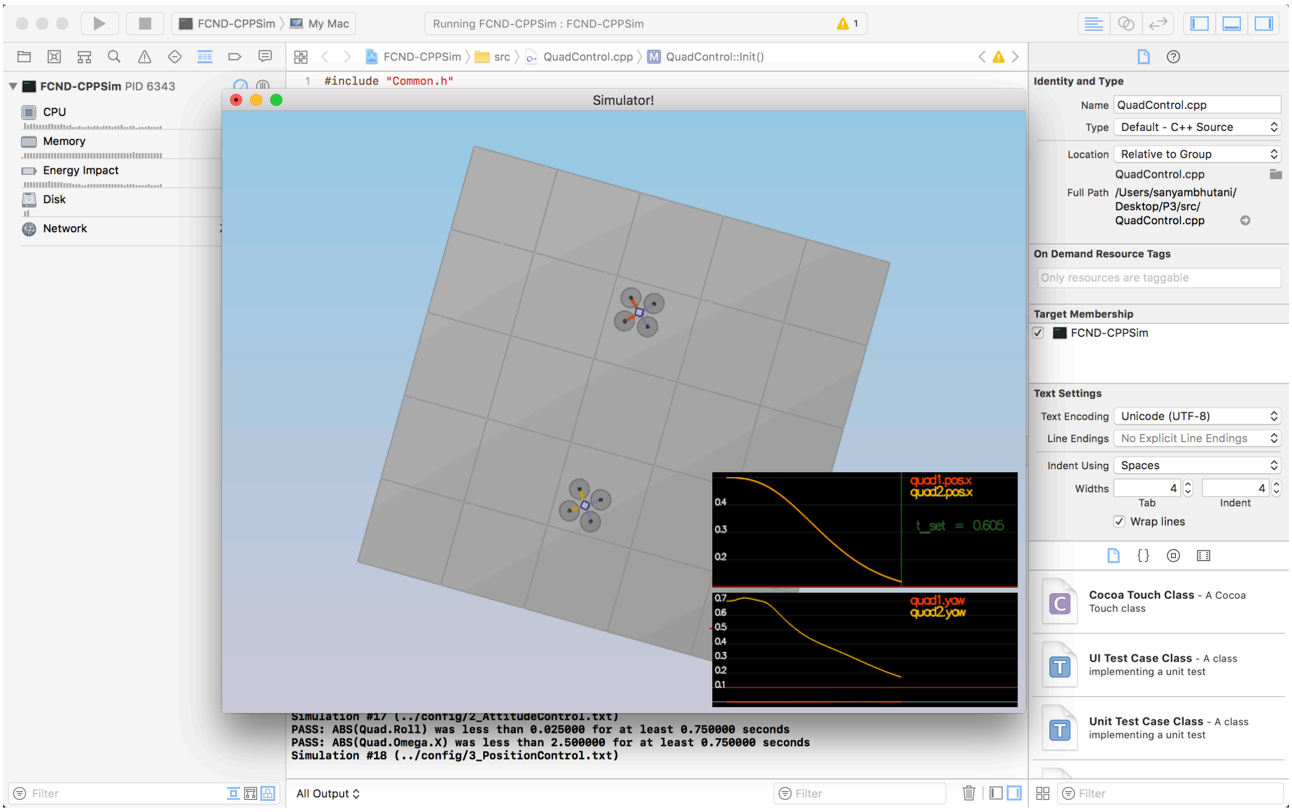
Scenario 2:

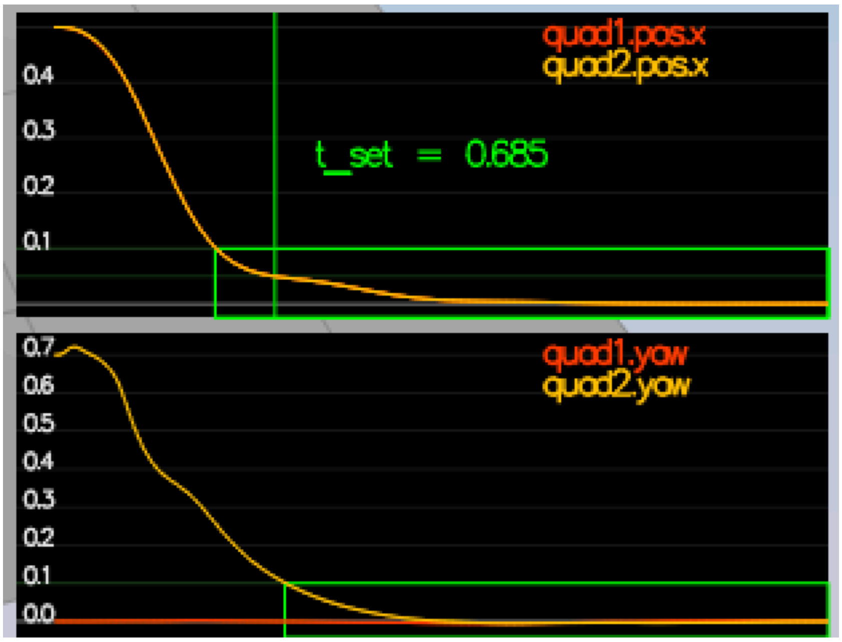





```
PASS: ABS(Quad.Roll) was less than 0.025000 for at least 0.750000 seconds
PASS: ABS(Quad.Omega.X) was less than 2.500000 for at least 0.750000 seconds
Simulation #34 (../config/2_AttitudeControl.txt)
PASS: ABS(Quad.Roll) was less than 0.025000 for at least 0.750000 seconds
PASS: ABS(Quad.Omega.X) was less than 2.500000 for at least 0.750000 seconds
Simulation #35 (../config/2_AttitudeControl.txt)
PASS: ABS(Quad.Roll) was less than 0.025000 for at least 0.750000 seconds
PASS: ABS(Quad.Omega.X) was less than 2.500000 for at least 0.750000 seconds
```

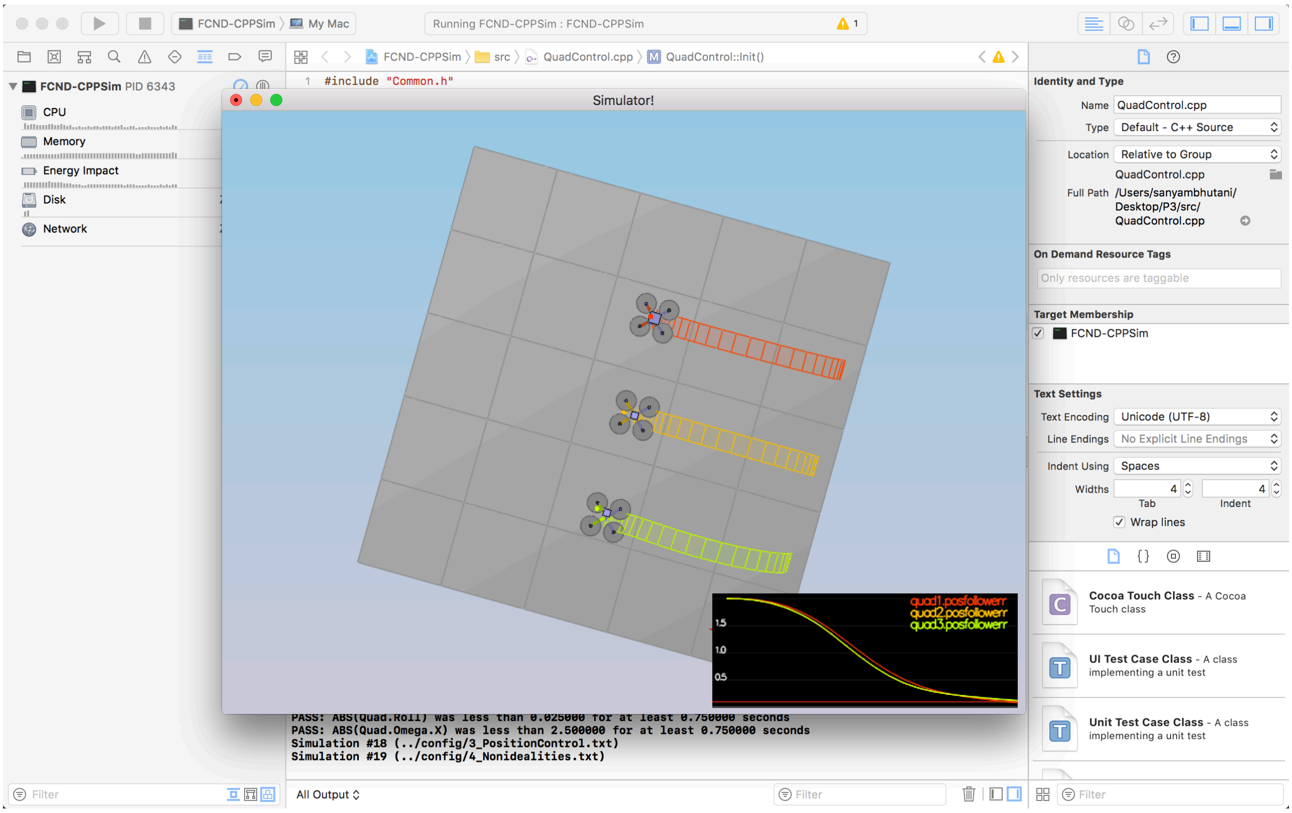
Scenario 3:

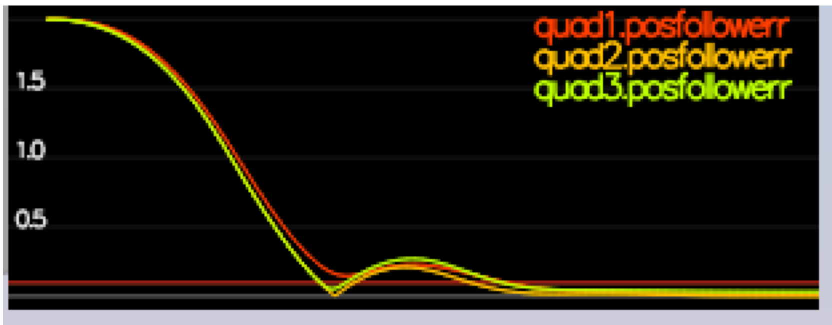





```
Simulation #43 (../config/3_PositionControl.txt)
PASS: ABS(Quad1.Pos.X) was less than 0.100000 for at least 1.250000 seconds
PASS: ABS(Quad2.Pos.X) was less than 0.100000 for at least 1.250000 seconds
PASS: ABS(Quad2.Yaw) was less than 0.100000 for at least 1.000000 seconds
```

Scenario 4:






```
Simulation #47 (../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
```

Optional Scenarios:

I haven’t tried the Optional Scenarios since the original exercises required very exhausting tuning.