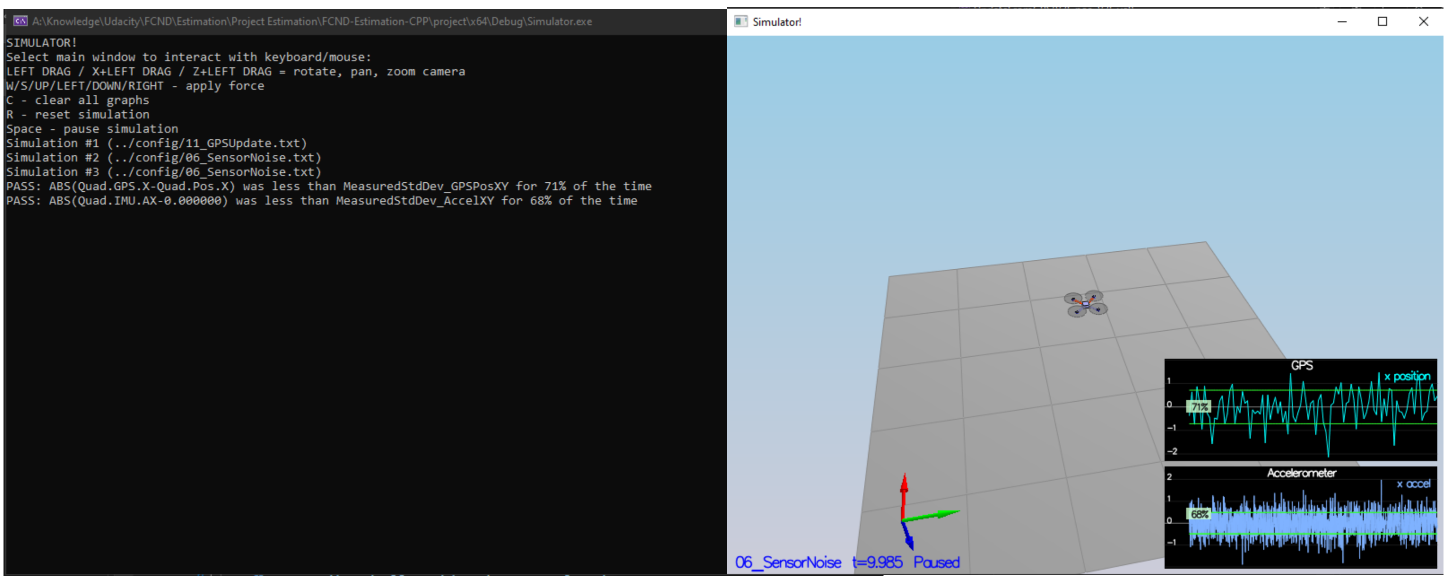
Estimation Project Write-up

In this document I'll explain step-by-step how I've met this project rubric points.

1. Sensor Noise:

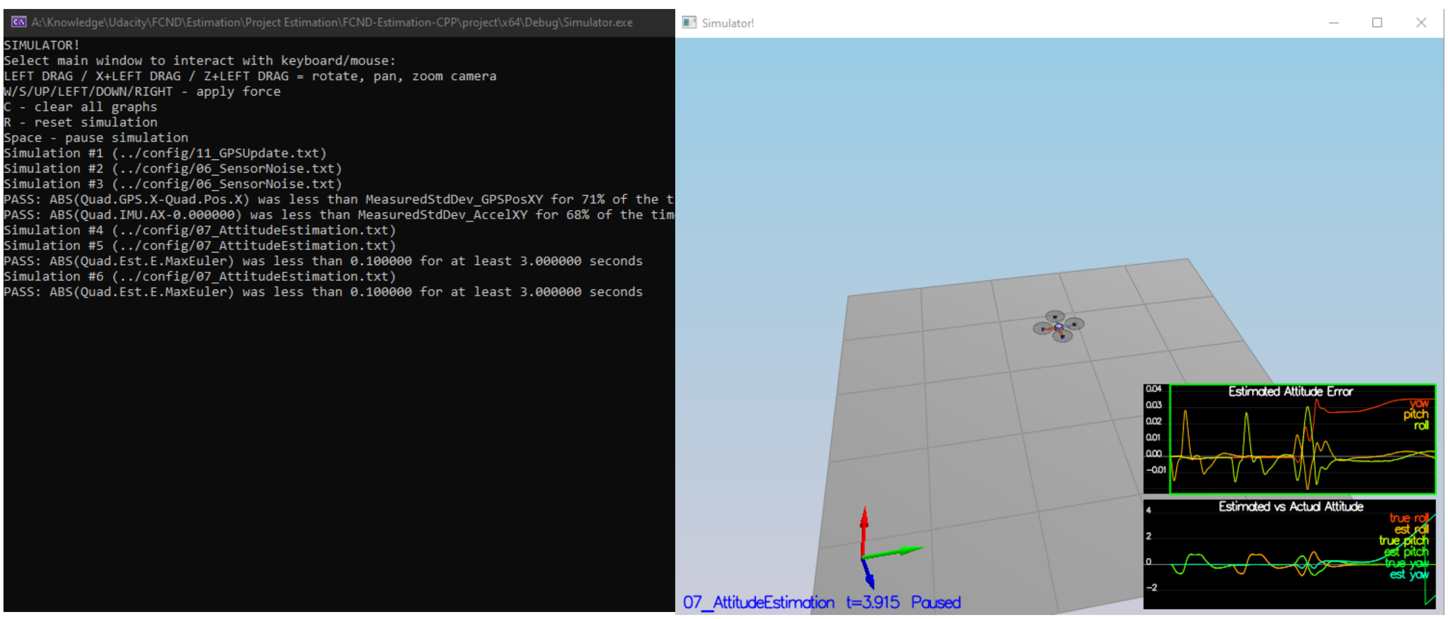
- I used the data in **config/log/Graph1.txt** & **config/log/Graph2.txt** files to calculate the **MeasuredStdDev_GPSPosXY** & **MeasuredStdDev_AccelXY** and they equal **0.714950933** & **0.488826997** respectively.



2. Attitude Estimation:

- In the UpdateFromIMU function I’ve coded a rotation matrix to convert the gyro rates from the body frame to the Euler Angels, and then calculated the predicted Roll & Pitch angels.

```
float phi = rollEst;
float theta = pitchEst;

Mat3x3F R;
R(0, 0) = 1;
R(0, 1) = sin(phi) * tan(theta);
R(0, 2) = cos(phi) * tan(theta);
R(1, 0) = 0;
R(1, 1) = cos(phi);
R(1, 2) = -sin(phi);
R(2, 0) = 0;
R(2, 1) = sin(phi) / cos(theta);
R(2, 2) = cos(phi) / cos(theta);

V3F euler_dot = R * gyro;

float predictedRoll = phi + dtIMU * euler_dot.x;
float predictedPitch = theta + dtIMU * euler_dot.y;

ekfState(6) = ekfState(6) + dtIMU * euler_dot.z;
```



3. Prediction Step:

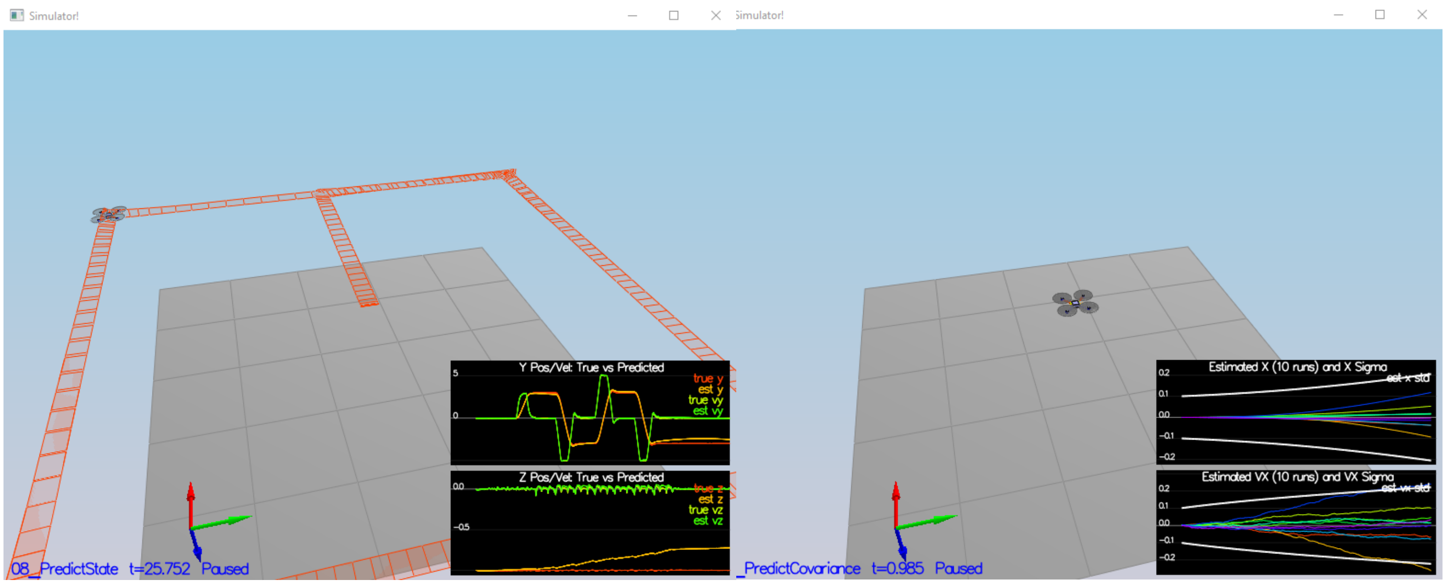
- In the PredictState function I've coded the calculations of the state prediction, in the GetRbgPrime, and in the Predict function as follows:

```
VectorXf QuadEstimatorEKF::PredictState(VectorXf curState, float dt, V3F accel, V3F gyro)
{
    assert(curState.size() == QUAD_EKF_NUM_STATES);
    VectorXf predictedState = curState;
    // Predict the current state forward by time dt using current accelerations and body rates as input
    // INPUTS:
    //   curState: starting state
    //   dt: time step to predict forward by [s]
    //   accel: acceleration of the vehicle, in body frame, *not including gravity* [m/s2]
    //   gyro: body rates of the vehicle, in body frame [rad/s]
    //
    // OUTPUT:
    //   return the predicted state as a vector

    // HINTS
    // - dt is the time duration for which you should predict. It will be very short (on the order of 1ms)
    //   so simplistic integration methods are fine here
    // - we've created an Attitude Quaternion for you from the current state. Use
    //   attitude.Rotate_BtoI(<V3F>) to rotate a vector from body frame to inertial frame
    // - the yaw integral is already done in the IMU update. Be sure not to integrate it again here

    Quaternion<float> attitude = Quaternion<float>::FromEuler123_RPY(rollEst, pitchEst, curState(6));

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

    V3F accI = attitude.Rotate_BtoI(accel) + V3F(0.f, 0.f, -9.81);

    predictedState(0) += predictedState(3) * dt;
    predictedState(1) += predictedState(4) * dt;
    predictedState(2) += predictedState(5) * dt;
    predictedState(3) += accI.x * dt;
    predictedState(4) += accI.y * dt;
    predictedState(5) += accI.z * dt;

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

```
MatrixXf QuadEstimatorEKF::GetRbgPrime(float roll, float pitch, float yaw)
{
    // first, figure out the Rbg_prime
    MatrixXf RbgPrime(3, 3);
    RbgPrime.setZero();

    // Return the partial derivative of the Rbg rotation matrix with respect to yaw. We call this RbgPrime.
    // INPUTS:
    //   roll, pitch, yaw: Euler angles at which to calculate RbgPrime
    //
    // OUTPUT:
    //   return the 3x3 matrix representing the partial derivative at the given point

    // HINTS
    // - this is just a matter of putting the right sin() and cos() functions in the right place.
    //   make sure you write clear code and triple-check your math
    // - You can also do some numerical partial derivatives in a unit test scheme to check
    //   that your calculations are reasonable

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

    RbgPrime(0, 0) = -cos(pitch) * sin(yaw);
    RbgPrime(0, 1) = -sin(roll) * sin(pitch) * sin(yaw) - cos(roll) * cos(yaw);
    RbgPrime(0, 2) = -cos(roll) * sin(pitch) * sin(yaw) + sin(roll) * cos(yaw);
    RbgPrime(1, 0) = cos(pitch) * cos(yaw);
    RbgPrime(1, 1) = sin(roll) * sin(pitch) * cos(yaw) - cos(roll) * sin(yaw);
    RbgPrime(1, 2) = cos(roll) * sin(pitch) * cos(yaw) + sin(roll) * sin(yaw);

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

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

    gPrime(0, 3) = dt;
    gPrime(1, 4) = dt;
    gPrime(2, 5) = dt;
    gPrime(3, 6) = (RbgPrime(0, 0) * accel.x + RbgPrime(0, 1) * accel.y + RbgPrime(0, 2) * accel.z) * dt;
    gPrime(4, 6) = (RbgPrime(1, 0) * accel.x + RbgPrime(1, 1) * accel.y + RbgPrime(1, 2) * accel.z) * dt;
    gPrime(5, 6) = (RbgPrime(2, 0) * accel.x + RbgPrime(2, 1) * accel.y + RbgPrime(2, 2) * accel.z) * dt;

    ekfCov = gPrime * ekfCov * gPrime.transpose() + Q;

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

And here are the results in the simulation after tuning the **QPosXYStd** & **QVelXYStd** in the **QuadEstimatorEKF.txt** file to **<u>0.1</u>** & **<u>0.2</u>** respectively:

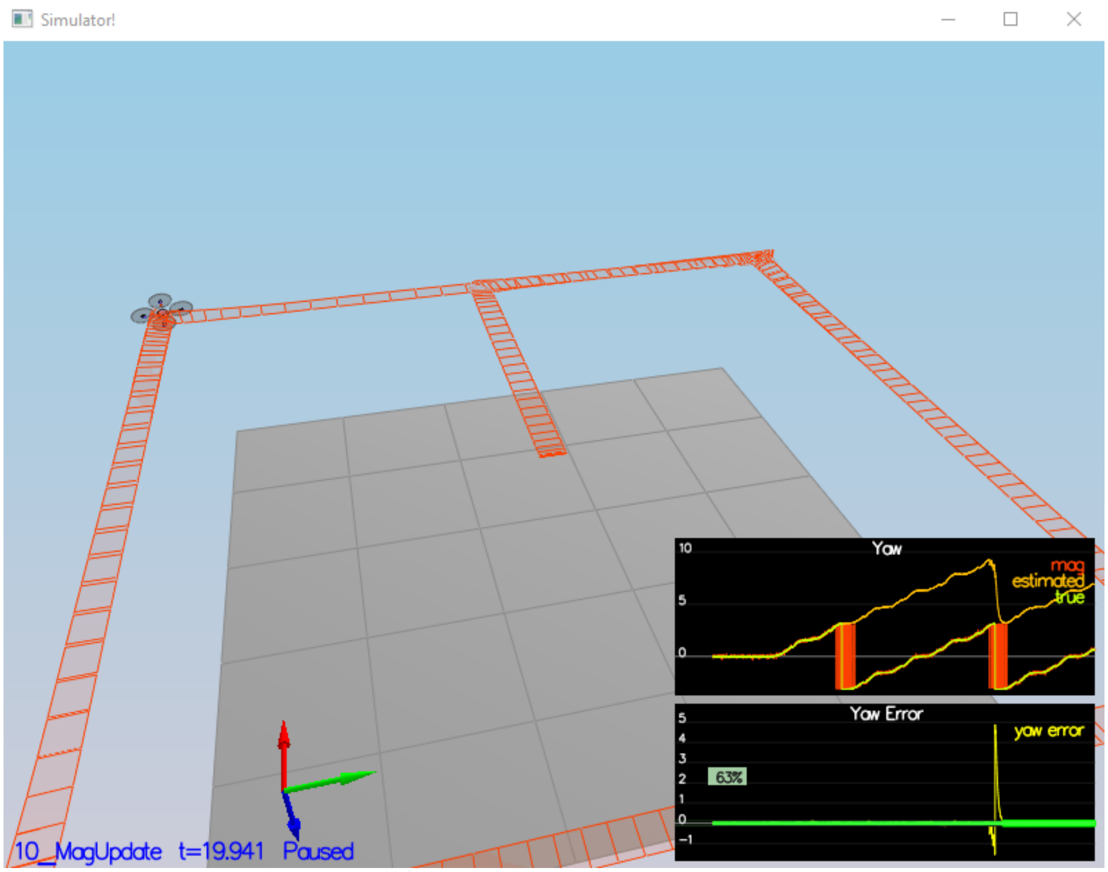


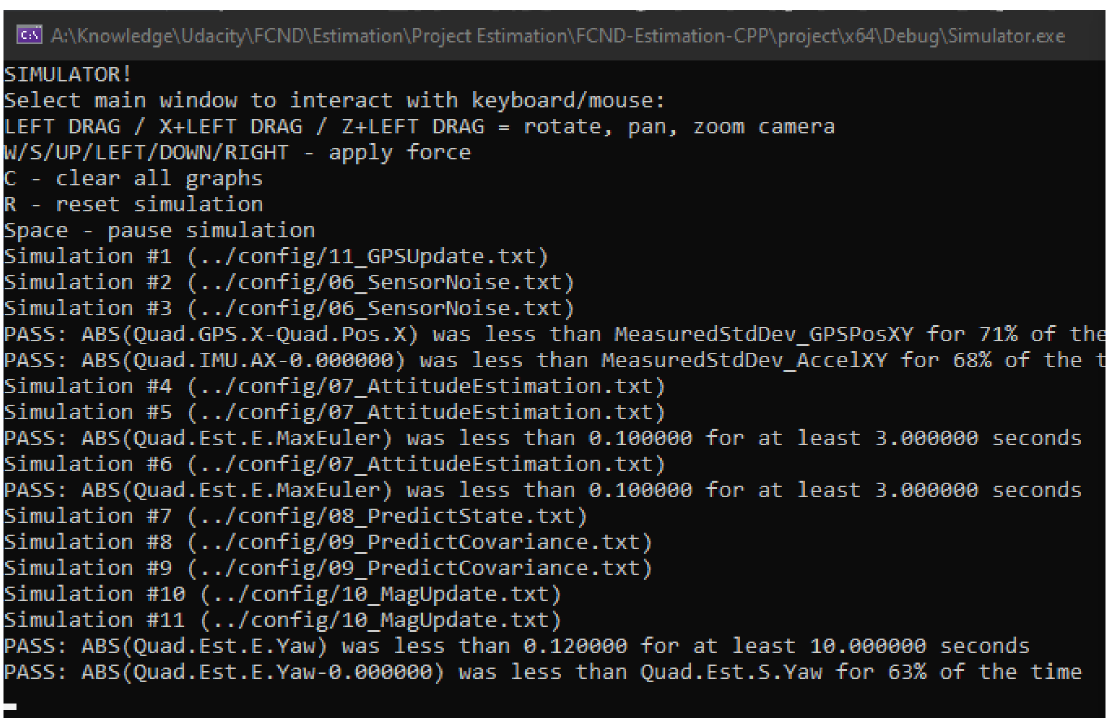
4. <u>Magnetometer Update:</u>

- Here I've assigned the Hprime matrix values, also the estimated yaw angel to the zFromX vector.
- And normalized the yaw angel so that it would be within –pi ~ +pi.

```
void QuadEstimatorEKF::UpdateFromMag(float magYaw)
{
    VectorXf z(1), zFromX(1);
    z(0) = magYaw;

    MatrixXf hPrime(1, QUAD_EKF_NUM_STATES);
    hPrime.setZero();

    // MAGNETOMETER UPDATE
    // Hints:
    //  - Your current estimated yaw can be found in the state vector: ekfState(6)
    //  - Make sure to normalize the difference between your measured and estimated yaw
    //    (you don't want to update your yaw the long way around the circle)
    //  - The magnetomer measurement covariance is available in member variable R_Mag
    ////////////////////////////// BEGIN STUDENT CODE ///////////////////////////

    hPrime(0, 6) = 1;

    zFromX(0) = ekfState(6);

    if (z(0) - zFromX(0) > M_PI) z(0) -= 2.f * M_PI;
    else if (z(0) - zFromX(0) < -M_PI) z(0) += 2.f * M_PI;

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

    Update(z, hPrime, R_Mag, zFromX);
}
```

Here is the result in the simulation after tuning the parameter **QYawStd** to **<u>0.1</u>** in the **QuadEstimatorEKF.txt** file:




A:\Knowledge\Udacity\FCND\Estimation\Project Estimation\FCND-Estimation-CPP\project\x64\Debug\Simulator.exe

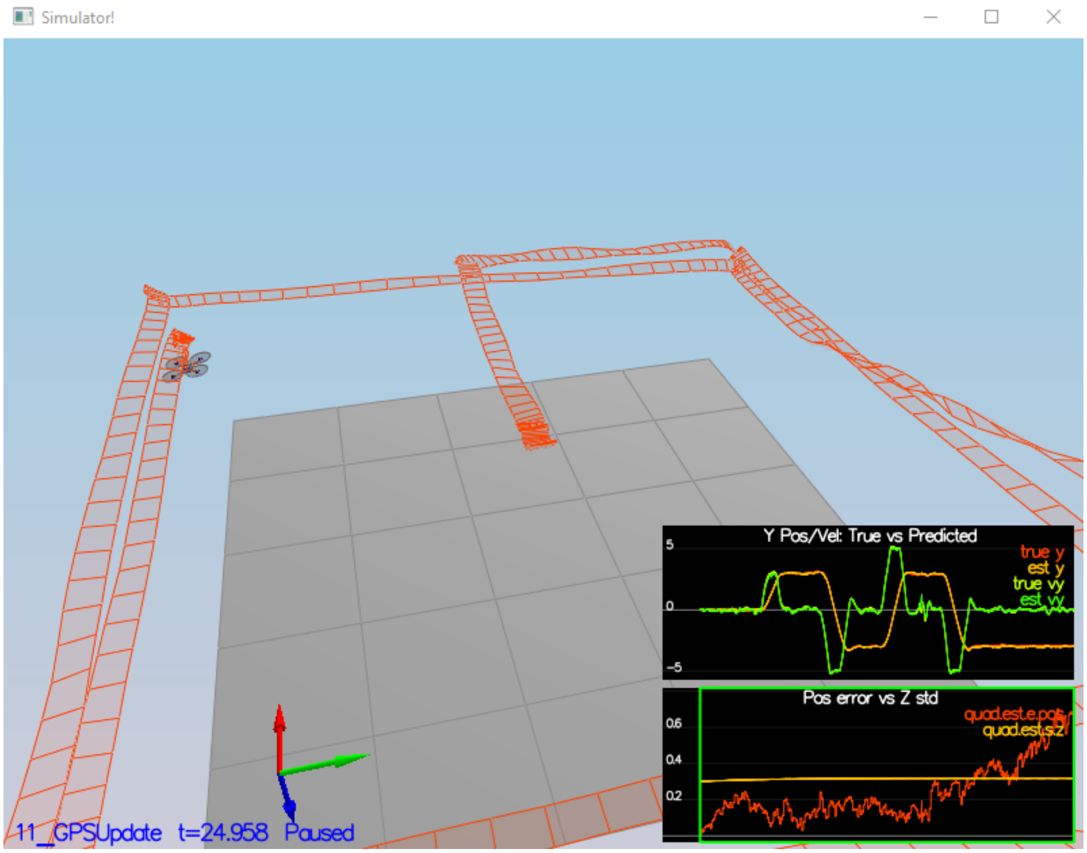
```
SIMULATOR!
Select main window to interact with keyboard/mouse:
LEFT DRAG / X+LEFT DRAG / Z+LEFT DRAG = rotate, pan, zoom camera
W/S/UP/LEFT/DOWN/RIGHT - apply force
C - clear all graphs
R - reset simulation
Space - pause simulation
Simulation #1 (../config/11_GPSUpdate.txt)
Simulation #2 (../config/06_SensorNoise.txt)
Simulation #3 (../config/06_SensorNoise.txt)
PASS: ABS(Quad.GPS.X-Quad.Pos.X) was less than MeasuredStdDev_GPSPosXY for 71% of the
PASS: ABS(Quad.IMU.AX-0.000000) was less than MeasuredStdDev_AccelXY for 68% of the t
Simulation #4 (../config/07_AttitudeEstimation.txt)
Simulation #5 (../config/07_AttitudeEstimation.txt)
PASS: ABS(Quad.Est.E.MaxEuler) was less than 0.100000 for at least 3.000000 seconds
Simulation #6 (../config/07_AttitudeEstimation.txt)
PASS: ABS(Quad.Est.E.MaxEuler) was less than 0.100000 for at least 3.000000 seconds
Simulation #7 (../config/08_PredictState.txt)
Simulation #8 (../config/09_PredictCovariance.txt)
Simulation #9 (../config/09_PredictCovariance.txt)
Simulation #10 (../config/10_MagUpdate.txt)
Simulation #11 (../config/10_MagUpdate.txt)
PASS: ABS(Quad.Est.E.Yaw) was less than 0.120000 for at least 10.000000 seconds
PASS: ABS(Quad.Est.E.Yaw-0.000000) was less than Quad.Est.S.Yaw for 63% of the time
```

5. Closed loop + GPS Update:

- In the UpdateFromGPS function I’ve assigned the Hprime matrix values, and also zFromX matrix values as follows:

```
void QuadEstimatorEKF::UpdateFromGPS(V3F pos, V3F vel)
{
    VectorXf z(6), zFromX(6);
    z(0) = pos.x;
    z(1) = pos.y;
    z(2) = pos.z;
    z(3) = vel.x;
    z(4) = vel.y;
    z(5) = vel.z;

    MatrixXf hPrime(6, QUAD_EKF_NUM_STATES);
    hPrime.setZero();

    // GPS UPDATE
    // Hints:
    //  - The GPS measurement covariance is available in member variable R_GPS
    //  - this is a very simple update
    ////////////////////////////// BEGIN STUDENT CODE ///////////////////////////

    hPrime(0, 0) = hPrime(1, 1) = hPrime(2, 2) = hPrime(3, 3) = hPrime(4, 4) = hPrime(5, 5) = 1;

    zFromX(0) = ekfState(0);
    zFromX(1) = ekfState(1);
    zFromX(2) = ekfState(2);
    zFromX(3) = ekfState(3);
    zFromX(4) = ekfState(4);
    zFromX(5) = ekfState(5);

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

    Update(z, hPrime, R_GPS, zFromX);
}
```

- And here is the final result in the simulator after turning off the Ideal IMU and using my own Controller and control parameters from the last Project:




```
A:\Knowledge\Udacity\FCND\Estimation\Project Estimation\FCND-Estimation-CPP\project\x64\Debug\Simulator.exe
SIMULATOR!
Select main window to interact with keyboard/mouse:
LEFT DRAG / X+LEFT DRAG / Z+LEFT DRAG = rotate, pan, zoom camera
W/S/UP/LEFT/DOWN/RIGHT - apply force
C - clear all graphs
R - reset simulation
Space - pause simulation
Simulation #1 (../config/11_GPSUpdate.txt)
Simulation #2 (../config/06_SensorNoise.txt)
Simulation #3 (../config/06_SensorNoise.txt)
PASS: ABS(Quad.GPS.X-Quad.Pos.X) was less than MeasuredStdDev_GPSPosXY for 71% of the
PASS: ABS(Quad.IMU.AX-0.000000) was less than MeasuredStdDev_AccelXY for 68% of the t
Simulation #4 (../config/07_AttitudeEstimation.txt)
Simulation #5 (../config/07_AttitudeEstimation.txt)
PASS: ABS(Quad.Est.E.MaxEuler) was less than 0.100000 for at least 3.000000 seconds
Simulation #6 (../config/07_AttitudeEstimation.txt)
PASS: ABS(Quad.Est.E.MaxEuler) was less than 0.100000 for at least 3.000000 seconds
Simulation #7 (../config/08_PredictState.txt)
Simulation #8 (../config/09_PredictCovariance.txt)
Simulation #9 (../config/09_PredictCovariance.txt)
Simulation #10 (../config/10_MagUpdate.txt)
Simulation #11 (../config/10_MagUpdate.txt)
PASS: ABS(Quad.Est.E.Yaw) was less than 0.120000 for at least 10.000000 seconds
PASS: ABS(Quad.Est.E.Yaw-0.000000) was less than Quad.Est.S.Yaw for 63% of the time
Simulation #12 (../config/11_GPSUpdate.txt)
Simulation #13 (../config/11_GPSUpdate.txt)
PASS: ABS(Quad.Est.E.Pos) was less than 1.000000 for at least 20.000000 seconds
```