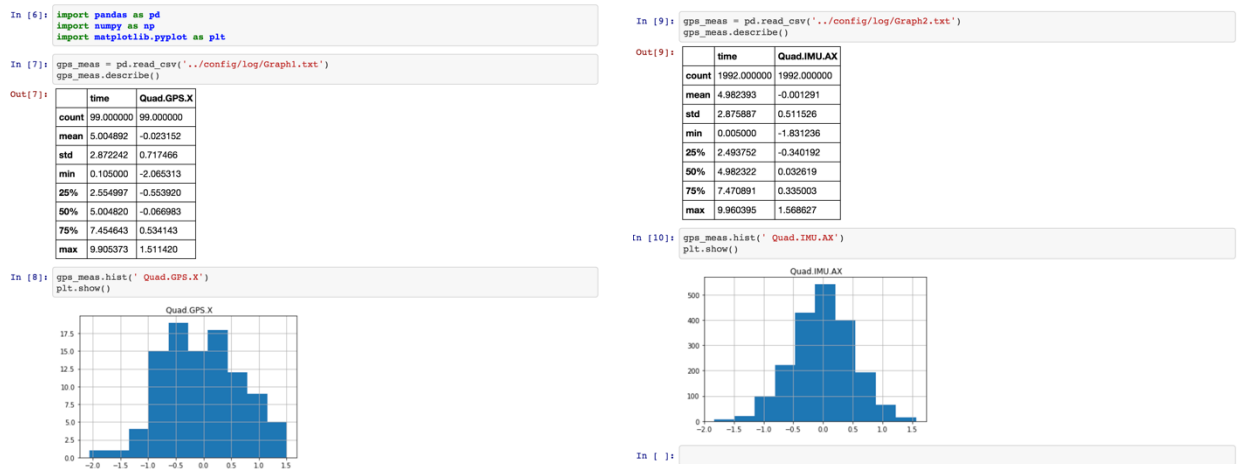
FCND PROJECT 4: BUILDING AN ESTIMATOR

SENSOR NOISE

To calculate the GPS and Accelerometer measurement standard deviations in the X-direction, I used the logs recorded by the simulator after running the simulator for approximately 10 seconds. Next, I used the *Pandas* python library to read those logs into *DataFrames* and proceeded to compute the standard deviation of the "Quad.GPS.X" and "Quad.IMU.AX" columns. These steps are illustrated below and the notebook can be found "notebooks" folder of the my project's github repository (submitted with project).



In [6]:
```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
```

In [7]:
```
gps_meas = pd.read_csv('../config/log/Graph1.txt')
gps_meas.describe()
```

Out[7]:

	time	Quad.GPS.X
count	99.000000	99.000000
mean	5.004892	-0.023152
std	2.872242	0.717466
min	0.105000	-2.065313
25%	2.554997	-0.553920
50%	5.004820	-0.066983
75%	7.454643	0.534143
max	9.905373	1.511420

In [8]:
```
gps_meas.hist(' Quad.GPS.X')
plt.show()
```


In [9]:
```
gps_meas = pd.read_csv('../config/log/Graph2.txt')
gps_meas.describe()
```

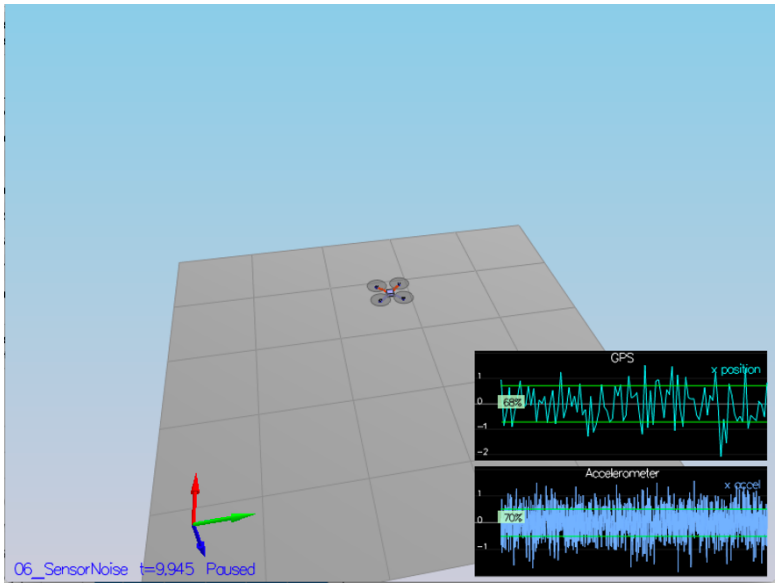
Out[9]:

	time	Quad.IMU.AX
count	1992.000000	1992.000000
mean	4.982393	-0.001291
std	2.875887	0.511526
min	0.005000	-1.831236
25%	2.493752	-0.340192
50%	4.982322	0.032619
75%	7.470891	0.335003
max	9.960395	1.568627

In [10]:
```
gps_meas.hist(' Quad.IMU.AX')
plt.show()
```


In [ ]:

The second cell in the left panel above shows that the GPS.X standard deviation is 0.717 while the second cell in right panel above shows the Quad.IMU.AX standard deviation is 0.511. Furthermore, we can see from the above histograms that that the accelerometer measurement distribution looks more gaussian than those of the GPS. Finally, using the above measured standard deviations in scenario "06_SensorNoist", we can see from the image below that 68-70% of the measurements fall within the lines denoting +/- 1 standard deviation.



ATTITUDE ESTIMATION

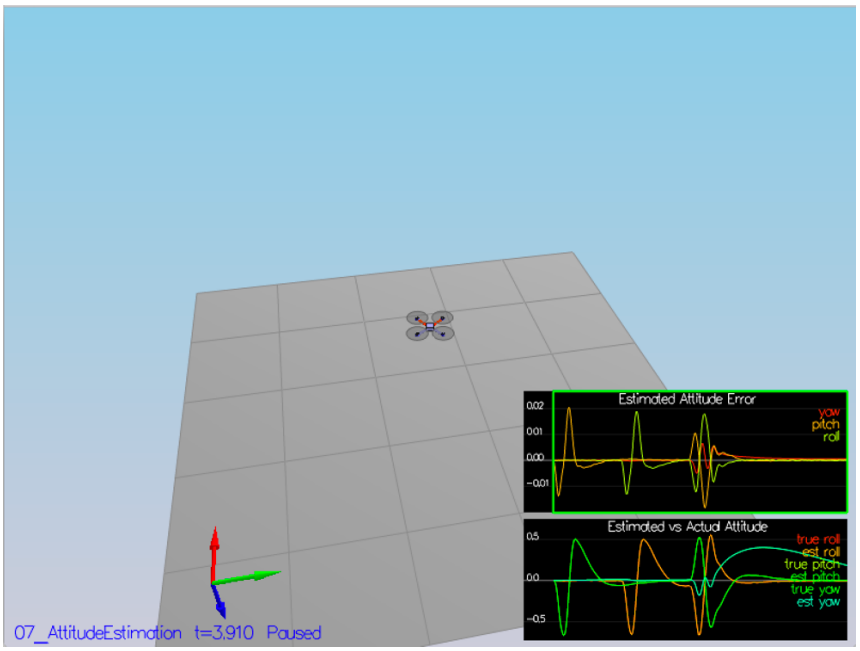
The code snippet below illustrates how attitude estimation was improved above the supplied small-angle integration method. More specifically, the method *FromEuler123_RPY* was used to obtain a Quaternion representation of the vehicle's attitude using the current estimate of the roll, pitch, and yaw. Next, this attitude was integrated forward in time using the method *IntegrateBodyRate* and measured angular velocities (p,q,r). Finally, the resulting roll, pitch, and yaw were extracted and supplied to the complementary filter.

```
Quaternion < float > attitude = Quaternion < float >::FromEuler123_RPY(rollEst, pitchEst, ekfState(6));
Quaternion < float > predicted_attitude = attitude.IntegrateBodyRate(gyro, dtIMU);

float predictedPitch = predicted_attitude.Pitch();
float predictedRoll = predicted_attitude.Roll();
ekfState(6) = predicted_attitude.Yaw(); // normalize yaw to -pi .. pi

if (ekfState(6) > F_PI) ekfState(6) -= 2. f * F_PI;
if (ekfState(6) < -F_PI) ekfState(6) += 2. f * F_PI;
```

The image below shows the drone successfully passing the simulation requirement of maintaining an attitude estimation error within 0.1 rad for at least 3 seconds.



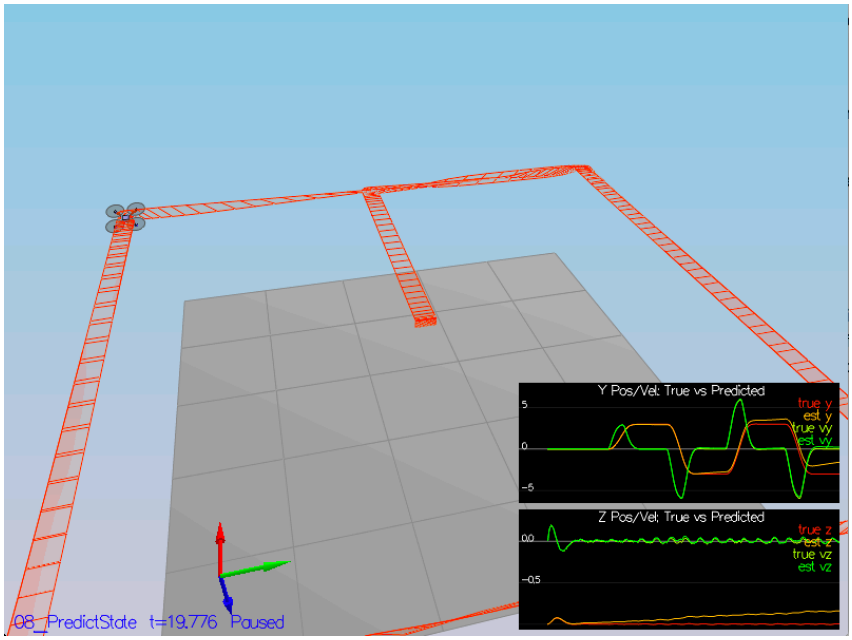
PREDICTION STEP

The first step in the prediction process is to advance the vehicle state. This is accomplished by the *PredictState* method. The code snippet below advances each of the vehicle state variables. Position variables (x, y, and z) are advanced by their respective velocities multiplied by the timestep duration *dt*. Velocity variables (xdot, ydot, zdot) are advanced by their respective accelerations multiplied by the *dt*.

Pay close attention to how the vehicles attitude is used to transform accelerations in the body frame to accelerations in the inertial. Furthermore, gravity is incorporated to compute the total acceleration in the inertial frame.

```
// Update X
predictedState(0) += predictedState(3) * dt;
// Update Y
predictedState(1) += predictedState(4) * dt;
// Update Z
predictedState(2) += predictedState(5) * dt;
// Total acceleration in inertial frame;
V3F total_accel = attitude.Rotate_BtoI(accel) - V3F(0.0, 0.0, CONST_GRAVITY);
// Update Xdot
predictedState(3) += total_accel.x * dt;
// Update Ydot
predictedState(4) += total_accel.y * dt;
// Update Zdot
predictedState(5) += total_accel.z * dt;
```

With the *PredictState* method completed, the estimated state tacks the actual state with with reasonably slow drift as desired.



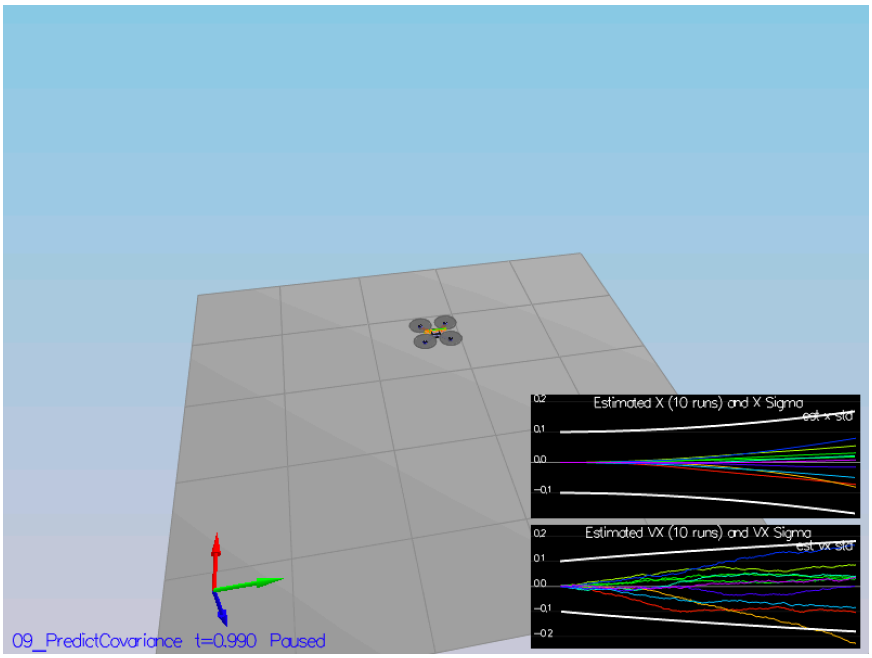
Next, we update the vehicle's covariance around its estimated state in the *Predict* method. This accomplished by the code snippet below where we construct the Jacobian of the nonlinear state transition matrix (gPrime) and fill its entries appropriately. Note the vehicles acceleration being multiplied by the derivative of the vehicle's rotation matrix (RbgPrime) and the EKF update equation on the last line.

```
// Acceleration in inertial frame;
VectorXf accelVec(3);
accelVec << accel.x, accel.y, accel.z;
accelVec = RbgPrime * accelVec;

gPrime.row(0) << 1, 0, 0, dt, 0, 0, 0;
gPrime.row(1) << 0, 1, 0, 0, dt, 0, 0;
gPrime.row(2) << 0, 0, 1, 0, 0, dt, 0;
gPrime.row(3) << 0, 0, 0, 1, 0, 0, accelVec(0) * dt;
gPrime.row(4) << 0, 0, 0, 0, 1, 0, accelVec(1) * dt;
gPrime.row(5) << 0, 0, 0, 0, 0, 1, accelVec(2) * dt;
gPrime.row(6) << 0, 0, 0, 0, 0, 0, 1;

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

Finally, the state process parameters QPosXYStd and QVelXYStd were tuned in order to capture the evolution of the state error estimates. This ultimately lead to appropriately growing covariance bounds as illustrated in the figure below.



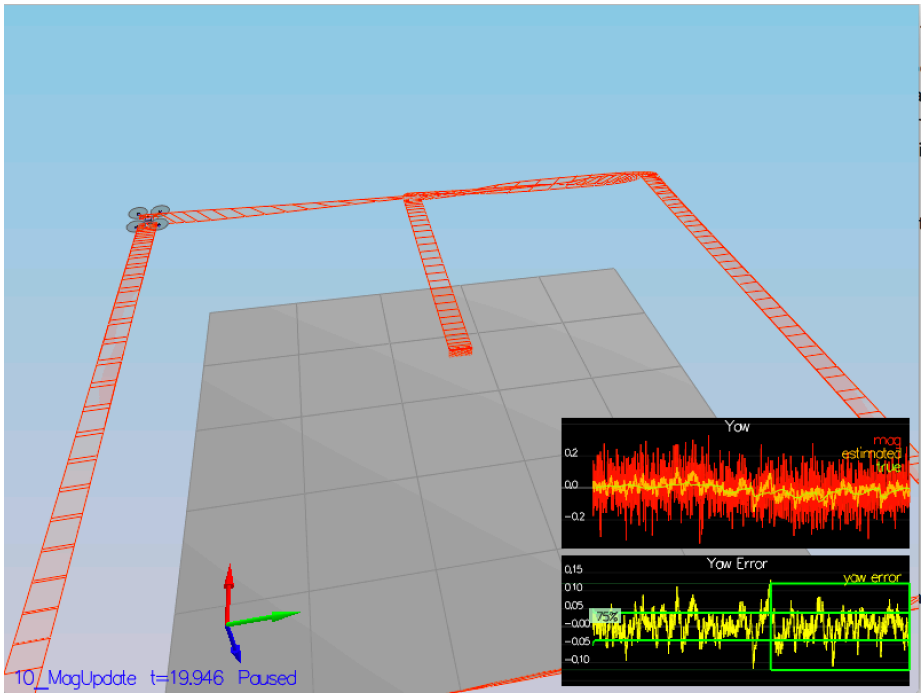
MAGNETOMETER UPDATE

Since our magnetometer directly observes the vehicle yaw in the inertial frame, the measurement matrix is a constant and contains a single entry (so no Jacobian is necessary). However, care must be taken to ensure that the difference between the measured and predicted yaw (z and zFromX) falls within [-Pi Pi] which is accomplished by adding or subtracting 2Pi to the measured yaw when needed. This is illustrated in the code snippet below.

```
hPrime(0, 6) = 1;
zFromX = hPrime * ekfState;

VectorXf yaw_error = z - zFromX;
if (yaw_error(0) > F_PI) z(0) -= 2. f * F_PI;
if (yaw_error(0) < -F_PI) z(0) += 2. f * F_PI;
```

When the magnetometer update was completed, the vehicle is able to successfully estimate its heading with 0.1 rad error for at least 10 seconds as shown below.

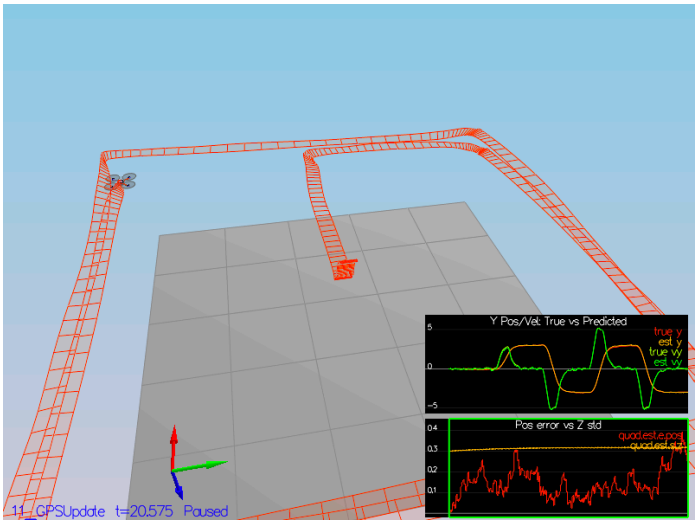


GPS UPDATE

The GPS directly observes the vehicle position and velocity in the inertial frame. Like the magnetometer, the measurement matrix in this case is a constant (so no Jacobian is necessary). This is illustrated in the code snippet below.

```
hPrime(0, 0) = 1;
hPrime(1, 1) = 1;
hPrime(2, 2) = 1;
hPrime(3, 3) = 1;
hPrime(4, 4) = 1;
hPrime(5, 5) = 1;
zFromX = hPrime * ekfState;
```

With the GPS update completed, the vehicle is able to successfully estimate its position with less than 1m error for as shown below.



ADDING MY CONTROLLER

After adding my controller (QuadController.cpp and QuadControlParams.txt) and decreasing gains kpPosXY, kpPosZ, KiPosZ, kpVelXY, kpVelZ, kpBank, kpPQR by 30% the vehicle successfully completes the simulation while maintaining a position error estimate of > 1 m as illustrated in the figure to the right.