FCND Estimation Project Report

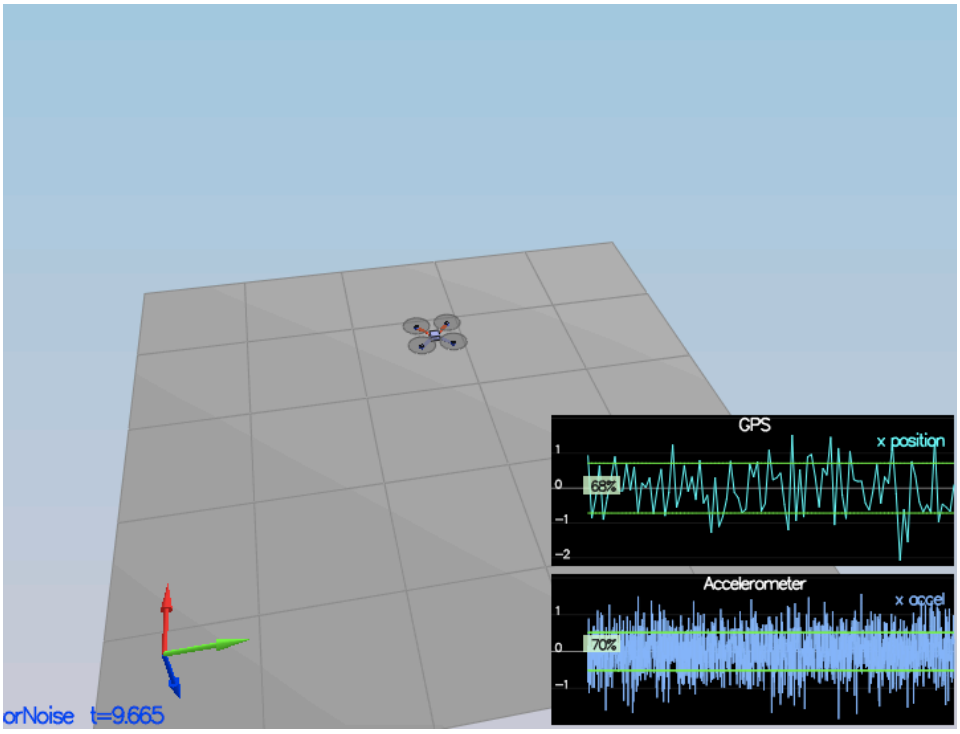
The goal of this project is to complete C++ code and also tune the parameters to implement the Estimator for a Quadrotor. The project is defined in steps and success criteria for each step. The implementation for each step will be explained in this report.

1-Sensor noise

The purpose of this step is adding additional realism to the problem by adding noise to the quad's sensors. In order to do that I calculated the Standard deviation of the recorded GPS X data and Accelerometer X data and calculate the Standard deviation for those data. The result was 0.7155998315 for GPS and 0.5121631606 for Accelerometer which I plugged into **config/6_Sensornoise.txt** and ran the scenario **06_NoisySensors**.

S**uccess criteria:** Your standard deviations should accurately capture the value of approximately 68% of the respective measurements.

The following picture show the result from the simulator:



The following is the console output:

Simulation #8 (../config/06_SensorNoise.txt)
PASS: ABS(Quad.GPS.X-Quad.Pos.X) was less than MeasuredStdDev_GPSPosXY for 68% of the time
PASS: ABS(Quad.IMU.AX-0.000000) was less than MeasuredStdDev_AccelXY for 70% of the time

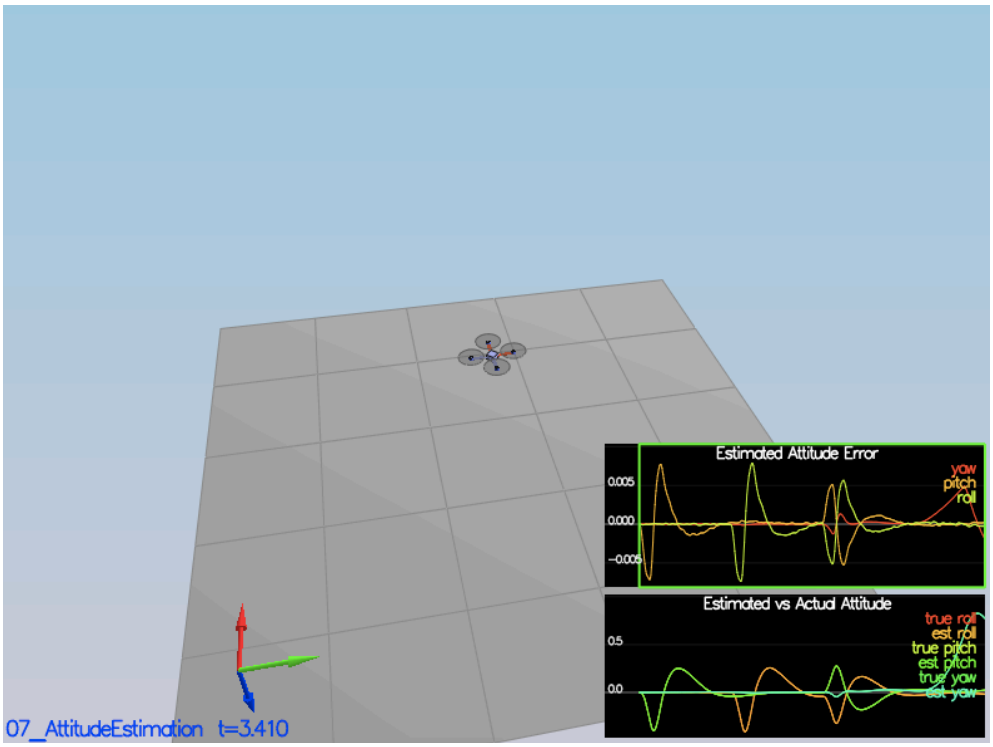
2-Attitude Estimation

The purpose of this step is to improve the complementary attitude filter with a better rate gyro attitude integration scheme. In order to do that we need to implement a better rate gyro nonlinear attitude integration scheme in function **UpdateFromIMU()** in **QuadEstimatorEKF.cpp**. I used Quaternion class to get Quaternion from Euler angles and then integrate the body rate. Here's the code:

```
Quaternion<float> quaternion = Quaternion<float>::FromEuler123_RPY(rollEst, pitchEst, ekfState(6));

Quaternion<float> attitude =  quaternion.IntegrateBodyRate(gyro, dtIMU);

float predictedRoll = attitude.Roll();
float predictedPitch = attitude.Pitch();
ekfState(6) = 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;
```

Success criteria: Your attitude estimator needs to get within 0.1 rad for each of the Euler angles for at least 3 seconds.

After implementing this function I ran the **07_AttitudeEstimation** simulation. The following picture shows the result:



The following is the console output:

Simulation #5 (../config/07_AttitudeEstimation.txt)
PASS: ABS(Quad.Est.E.MaxEuler) was less than 0.100000 for at least 3.000000 seconds

Step 3: Prediction Step

The purpose of this step is to implement the prediction step of the filter. This includes implementation of 3 functions **PredictState()**, **GetRbgPrime()** and **Predict()**. Transition model equations should be used to implement these functions. Transition model equations from the paper introduced in classroom are as the following picture:

$$R_{bg} = \begin{bmatrix} \cos\theta\cos\psi & \sin\phi\sin\theta\cos\psi - \cos\phi\sin\psi & \cos\phi\sin\theta\cos\psi + \sin\phi\sin\psi \\ \cos\theta\sin\psi & \sin\phi\sin\theta\sin\psi + \cos\phi\cos\psi & \cos\phi\sin\theta\sin\psi - \sin\phi\cos\psi \\ -\sin\theta & \cos\theta\sin\phi & \cos\theta\cos\phi \end{bmatrix} \tag{48}$$

Then the transition function is:

$$g(x_t, u_t, \Delta t) = \begin{bmatrix} x_{t,x} + x_{t,\dot{x}}\Delta t \\ x_{t,y} + x_{t,\dot{y}}\Delta t \\ x_{t,z} + x_{t,\dot{z}}\Delta t \\ x_{t,\dot{x}} \\ x_{t,\dot{y}} \\ x_{t,\dot{z}} - g\Delta t \\ x_{t,\psi} \end{bmatrix} + \begin{bmatrix} 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ R_{bg}[0:] & & & 0 \\ R_{bg}[1:] & & & 0 \\ R_{bg}[2:] & & & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} u_t \Delta t \tag{49}$$

Then we take the Jacobian:

$$g'(x_t, u_t, \Delta t) = \begin{bmatrix} 1 & 0 & 0 & \Delta t & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & \Delta t & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & \Delta t & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & \frac{\partial}{\partial x_{t,\psi}}(x_{t,\dot{x}} + R_{bg}[0:]u_t[0:3]\Delta t) \\ 0 & 0 & 0 & 0 & 1 & 0 & \frac{\partial}{\partial x_{t,\psi}}(x_{t,\dot{y}} + R_{bg}[1:]u_t[0:3]\Delta t) \\ 0 & 0 & 0 & 0 & 0 & 1 & \frac{\partial}{\partial x_{t,\psi}}(x_{t,\dot{z}} + R_{bg}[2:]u_t[0:3]\Delta t) \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 \end{bmatrix} \tag{50}$$

$$= \begin{bmatrix} 1 & 0 & 0 & \Delta t & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & \Delta t & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & \Delta t & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & R'_{bg}[0:]u_t[0:3]\Delta t \\ 0 & 0 & 0 & 0 & 1 & 0 & R'_{bg}[1:]u_t[0:3]\Delta t \\ 0 & 0 & 0 & 0 & 0 & 1 & R'_{bg}[2:]u_t[0:3]\Delta t \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 \end{bmatrix} \tag{51}$$

We define R'_{bg} as $\frac{\partial}{\partial x_{t,\psi}}$, defined as Equation 71 from Diebel [5]:

$$R'_{bg} = \begin{bmatrix} -\cos\theta\sin\psi & -\sin\phi\sin\theta\sin\psi - \cos\phi\cos\psi & -cos\phi\sin\theta\sin\psi + \sin\phi\cos\psi \\ \cos\theta\cos\psi & \sin\phi\sin\theta\cos\psi - \cos\phi\sin\psi & \cos\phi\sin\theta\cos\psi + \sin\phi\sin\psi \\ 0 & 0 & 0 \end{bmatrix} \tag{52}$$

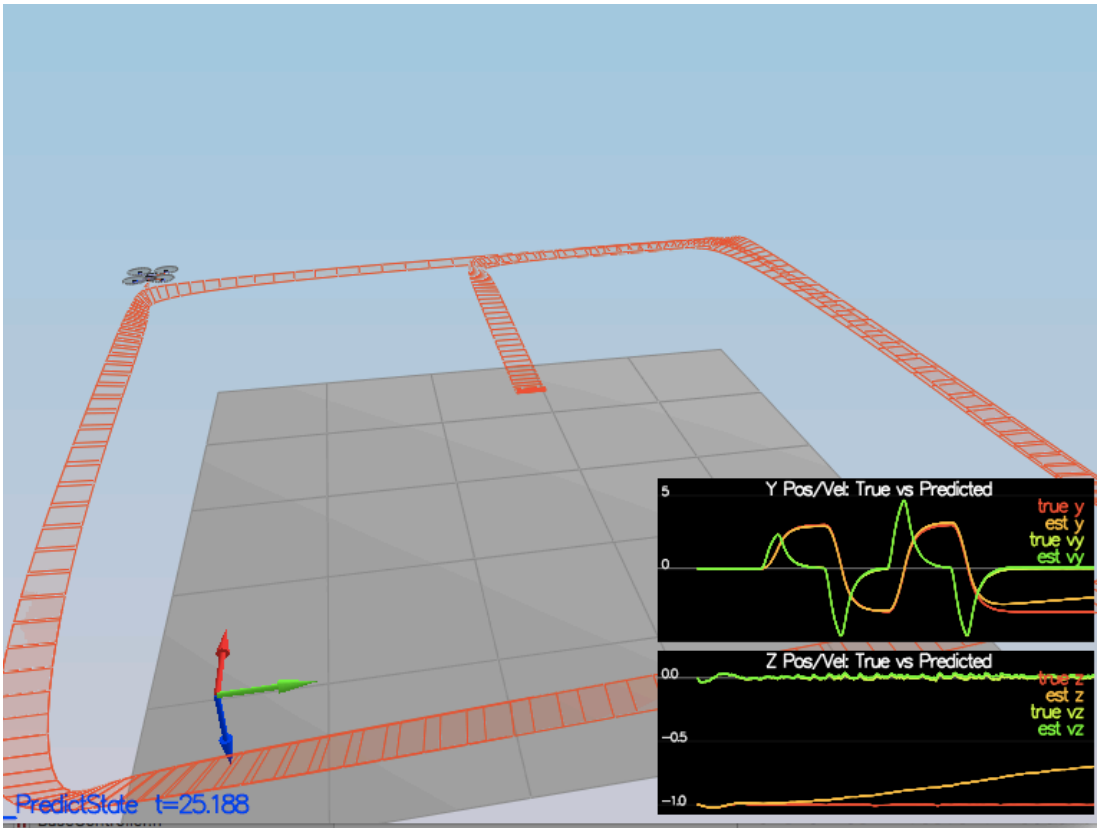
-**PredictState() f**unction should predict the current state forward by time dt using current accelerations and body rates as input. Since dt is very short, simplistic integration methods will be implemented. The code is as following:

*predictedState(0) = curState(0) + curState(3) * dt;*
*predictedState(1) = curState(1) + curState(4) * dt;*
*predictedState(2) = curState(2) + curState(5) * dt;*

V3F acc_world = attitude.Rotate_BtoI(accel);
acc_world.z -= 9.81;

*predictedState(3) = curState(3) + acc_world.x * dt;*
*predictedState(4) = curState(4) + acc_world.y * dt;*
*predictedState(5) = curState(5) + acc_world.z * dt;*

in which the **predictedState** with indices 0 to 5 is represent the state vector [x,y,z,x',y',z']. x, y, x are calculated by integrating x', y' , z' and x' , y' , z' are calculated by integrating from associated accelerations. Also **attitude.Rotate_BtoI(<V3F>)** function is used to rotate a vector from body frame to inertial frame. The yaw integral is already done in the IMU update and we don't do it here. After implementing this function I ran scenario **08_PredictState** and the simulation result is like the following picture:



-**GetRbgPrime()** function should be implemented based on equation 52 of the above equations picture and will be used in **Predict()** function. The code is as the following:

*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));*

Row 3 is has all zero values because the matrix is initialized to zero.

Predict() function should populate the matrix **g'** elements based on the equation 51 in the equations picture. Once we have **g'** the covariance can be calculated using this equation:

$$\bar{\Sigma}_t = G_t \Sigma_{t-1} G^T_t + Q_t$$

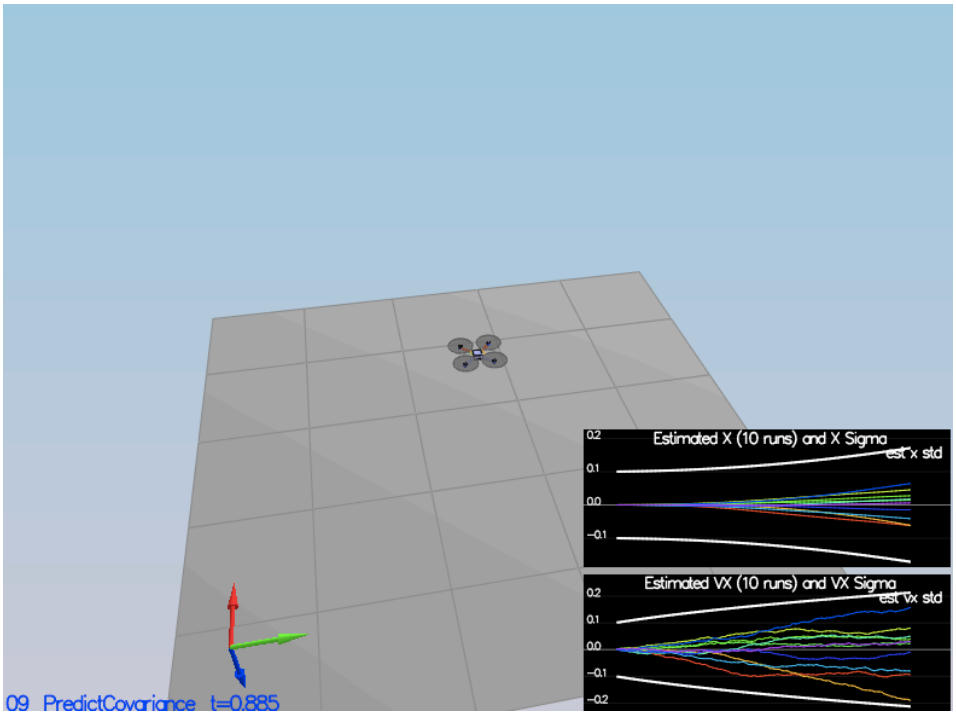
The code is as following:

gPrime(0,3) = dt;
gPrime(1,4) = dt;
gPrime(2,5) = dt;

*gPrime(3, 6) = (RbgPrime(0) * accel).sum() * dt;*
*gPrime(4, 6) = (RbgPrime(1) * accel).sum() * dt;*
*gPrime(5, 6) = (RbgPrime(2) * accel).sum() * dt;*

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

After implementing the **predict** function and tuning QPosXYStd and QVelXYStd, I ran the scenario **09_PredictionCov** and the simulation result is shown in the following picture:



Step 4: Magnetometer Update

The purpose of this step is to add the information from the magnetometer to improve the filter's performance in estimating the vehicle's heading. I first tuned **QYawStd** to .1 to capture the magnitude of the drift, as demonstrated in the projects steps. Then I implemented the **UpdateFromMag** function based on the following equations from the paper:

$$z_t = \begin{bmatrix} \psi \end{bmatrix} \tag{56}$$

$$h(x_t) = \begin{bmatrix} x_{t,\psi} \end{bmatrix} \tag{57}$$

Again since this is linear, the derivative is a matrix of zeros and ones.

$$h'(x_t) = \begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 0 & 1 \end{bmatrix} \tag{58}$$

The current estimated Yaw is in **ekfState(6)** and then we should normalize the difference to make sure we don't update the yaw around the circle. Here's the code:

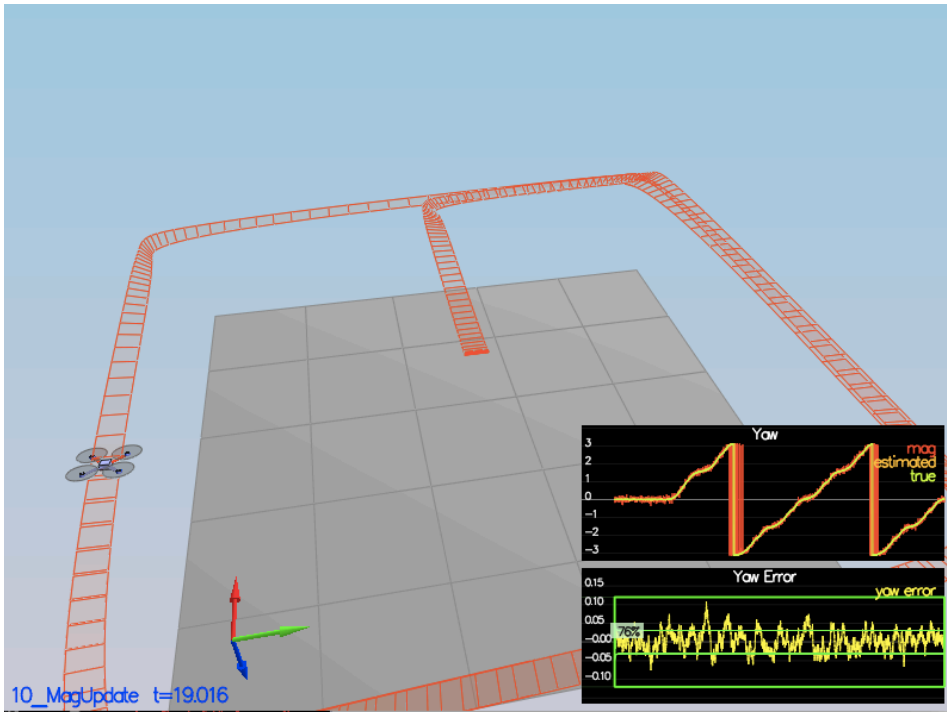
```
zFromX(0) = ekfState(6);
float yawDiff = z(0) - zFromX(0);

if (yawDiff > 3.1415){
   zFromX(0) += 2.f * 3.1415;
}
else if (yawDiff < -3.1415){
   zFromX(0) -= 2.f * 3.1415;
}

hPrime(0, 6) = 1;
```

Success criteria: Your goal is to both have an estimated standard deviation that accurately captures the error and maintain an error of less than 0.1rad in heading for at least 10 seconds of the simulation.

If we run the 10_MagUpdate scenario now the output from console is:

Simulation #4 (../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 76% of the time

and simulation result will be like the following picture:



Step 5: Closed Loop + GPS Update

The purpose of this step is to use our estimator instead of ideal estimator. We should first change **config/11_GPSUpdate.txt** file to stop using ideal estimator. Then I implemented **UpdateFromGPS()** function based on the following from the paper:

$$z_t = \begin{bmatrix} x \\ y \\ z \\ \dot{x} \\ \dot{y} \\ \dot{z} \end{bmatrix} \tag{53}$$

Then the measurement model is:

$$h(x_t) = \begin{bmatrix} x_{t,x} \\ x_{t,y} \\ x_{t,z} \\ x_{t,\dot{x}} \\ x_{t,\dot{y}} \\ x_{t,\dot{z}} \end{bmatrix} \tag{54}$$

Then the partial derivative is the identity matrix, augmented with a vector of zeros for $\frac{\partial}{\partial x_{t,\psi}} h(x_t)$:

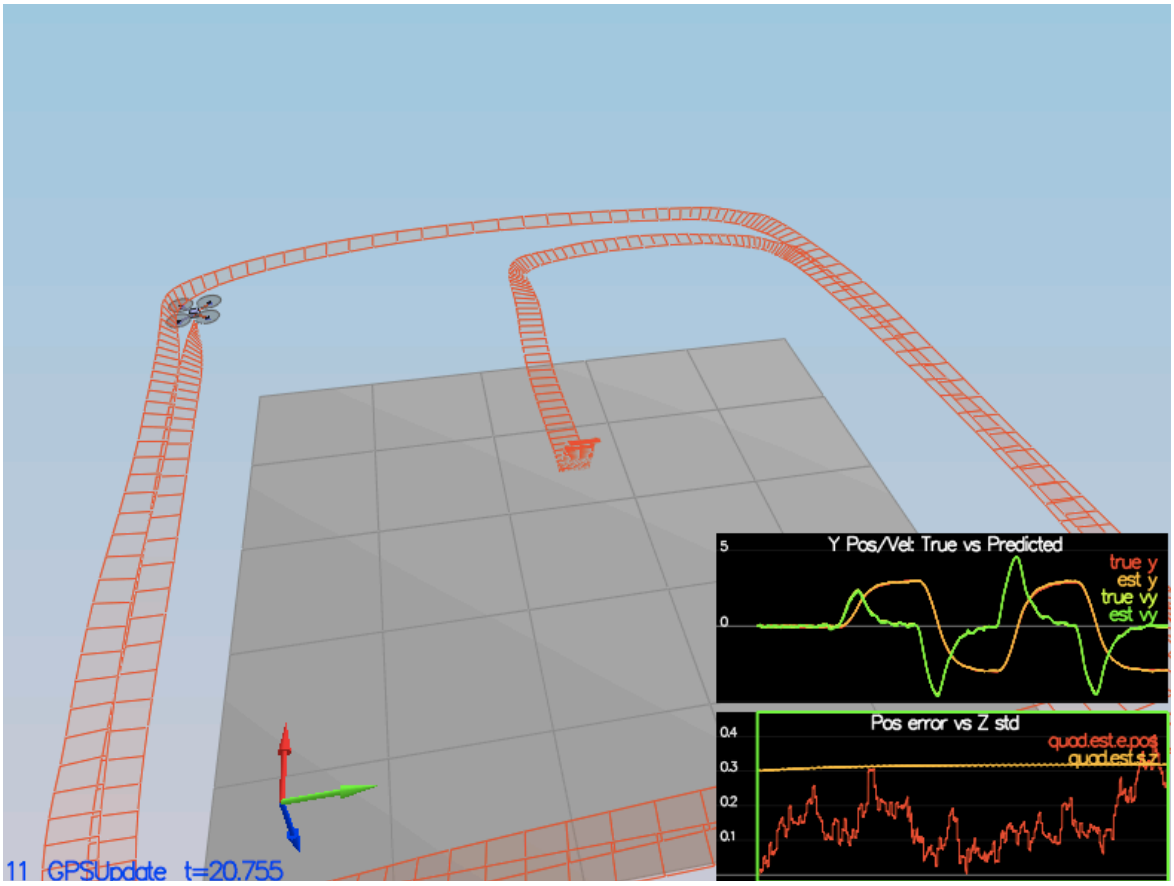
$$h'(x_t) = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 \end{bmatrix} \tag{55}$$

The code is like this:

```
for (int i = 0; i < 6; i++){
   zFromX(i) = ekfState(i);
   hPrime(i, i) = 1;
}
```

Success criteria: Your objective is to complete the entire simulation cycle with estimated position error of < 1m.

If we run simulation **11_GPSUpdate** now, the result will be like the following picture:



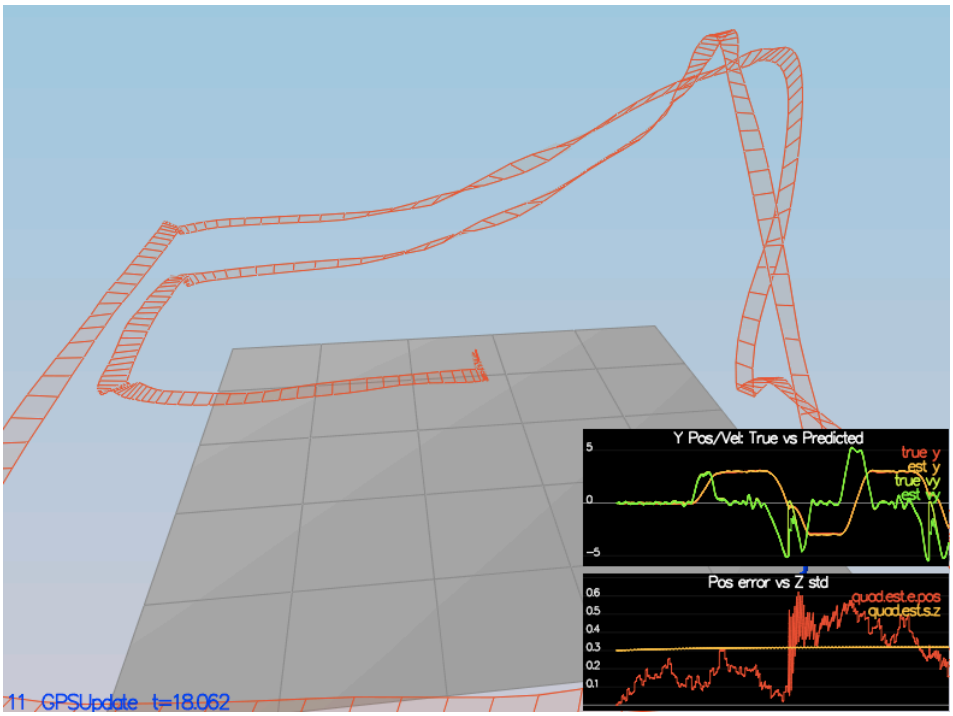
The console output is as follows:

Simulation #4 (../config/11_GPSUpdate.txt)
PASS: ABS(Quad.Est.E.Pos) was less than 1.000000 for at least 20.000000 seconds

Step 6: Adding Your Controller

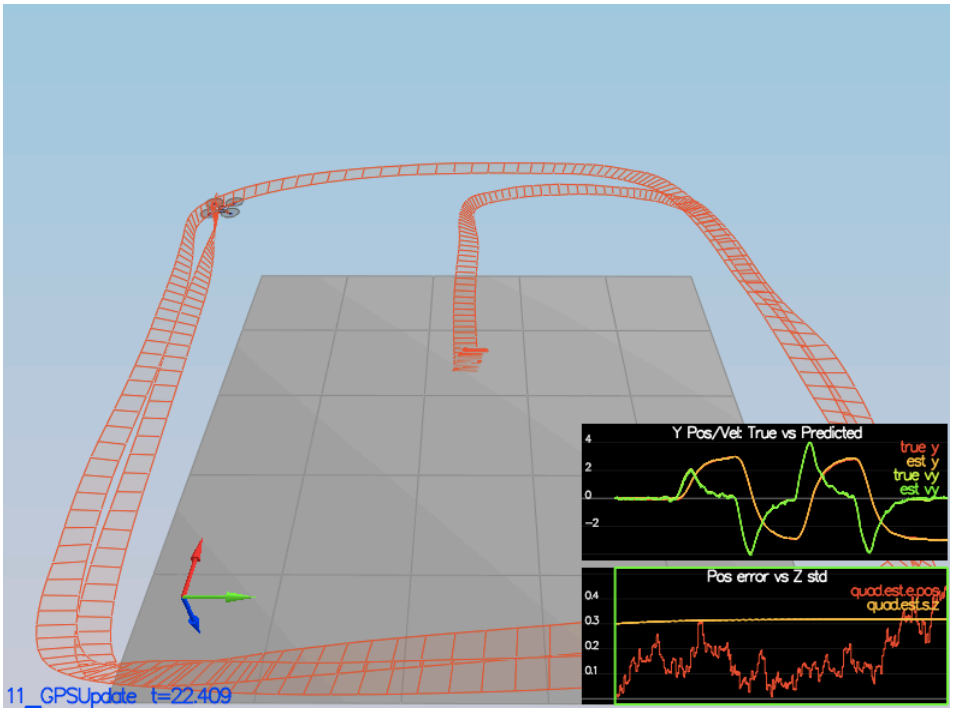
The purpose of this step is to use the controller from Controls Project instead of a controller that has been relaxed to work with an estimated state. I replaced the **QuadController.cpp** and

QuadControlParams.txt files with the files from my Controls project and ran simulation **11_GPSUpdate**. The result in simulator was like the following picture:



Success criteria: Your objective is to complete the entire simulation cycle with estimated position error of < 1m.

After tuning the parameters in **QuadControlParams.txt**, specifically decreasing position and velocity gains, the result gets better and passes the success criteria. Here's the simulator picture:



Here's the console output:

Simulation #2 (../config/11_GPSUpdate.txt)
PASS: ABS(Quad.Est.E.Pos) was less than 1.000000 for at least 20.000000 seconds

After the changes to control parameters, I also tried the scenarios 6 to 10 again and they all pass the success criteria.