FCND Project 4: Estimation of Quadrotor Report

Submitted by: Abuhanif Bhuiyan, Jul 27, 2018

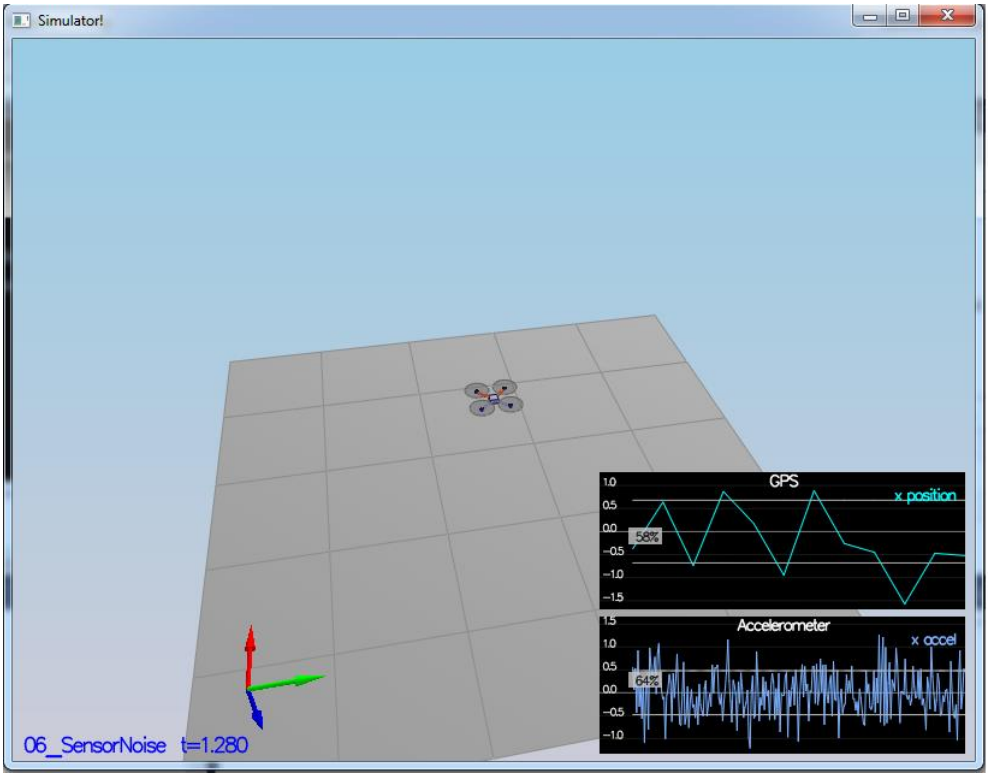
In this project I have implemented a state estimation for a quadrotor with an extended Kalman filter (EKF). The following requirements were implemented below.

1: Sensor Noise

Standard deviations were calculated by importing the data and using the standard deviation formula.

MeasuredStdDev_GPSPosXY = 0.68

MeasuredStdDev_accelXY = 0.48



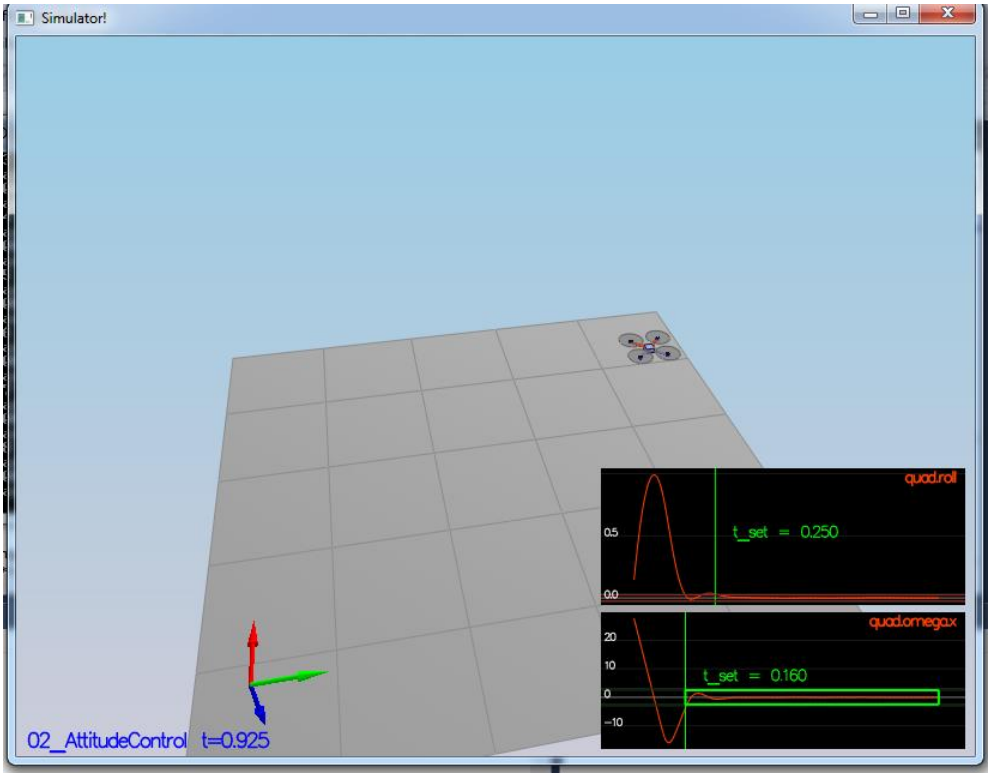
Success criteria: *Your standard deviations should accurately capture the value of approximately 68% of the respective measurements.*

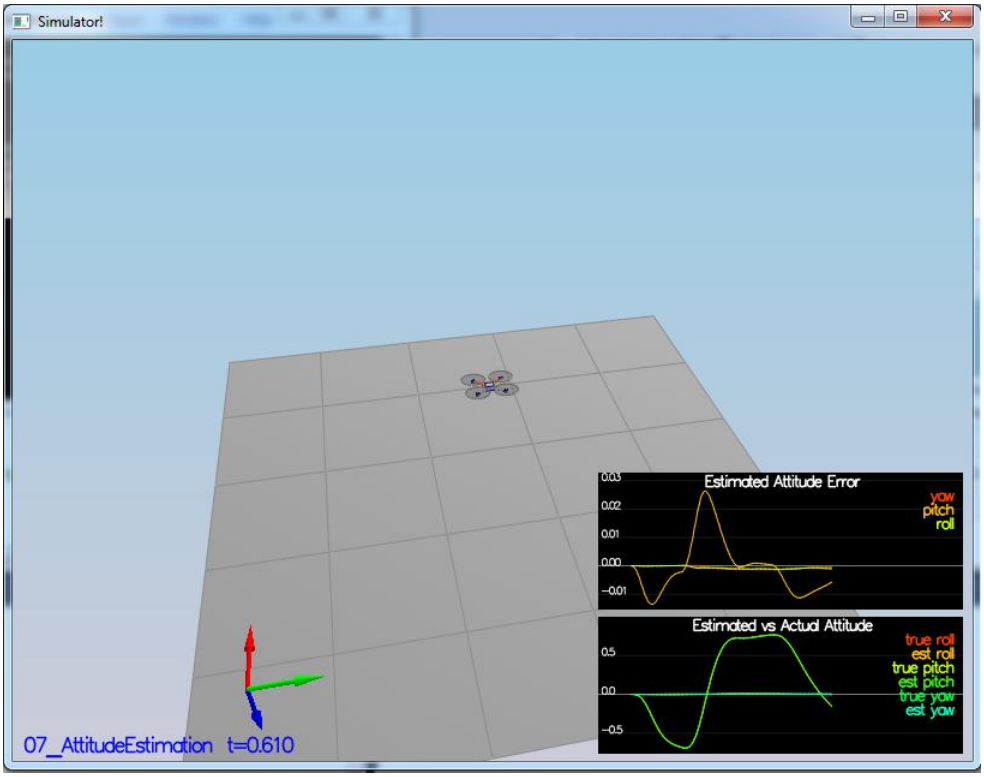
2: Attitude Estimation

I have computed the next attitude using the quaternion integration provided in the quaternion class. Then I updated the predicted state. The code used:

```
// Created a quaternion class object from the Roll, Pitch and Yaw
```

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

        // Used the built in function of the quaternon class to integrate using giro and
DT of the IMU
        attitude.IntegrateBodyRate(gyro, dtIMU);

        // Get the integrated Roll, Pitch and Yaw to be used on the integraton and update
function
        float predictedPitch = attitude.Pitch();  //Pitch
        float predictedRoll = attitude.Roll();  //Roll
        ekfState(6) = attitude.Yaw();      // Yaw

        // Normalized 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;
```





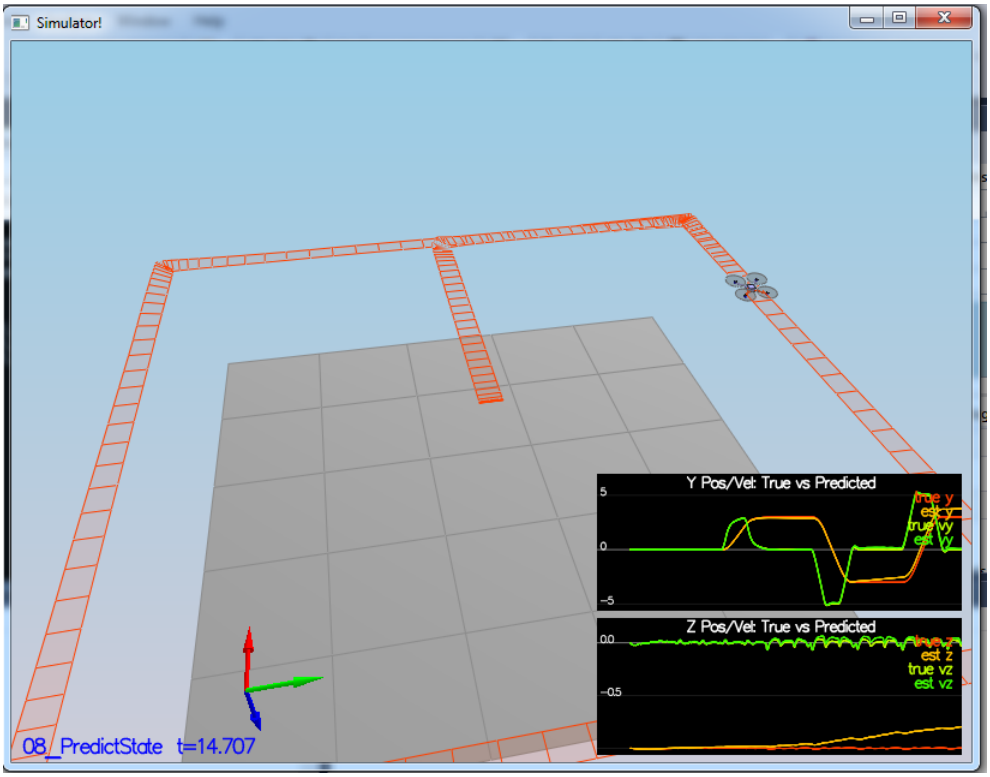
Success criteria: A*ttitude estimator needs to get within 0.1 rad for each of the Euler angles for at least 3 seconds. Met the requirement.*

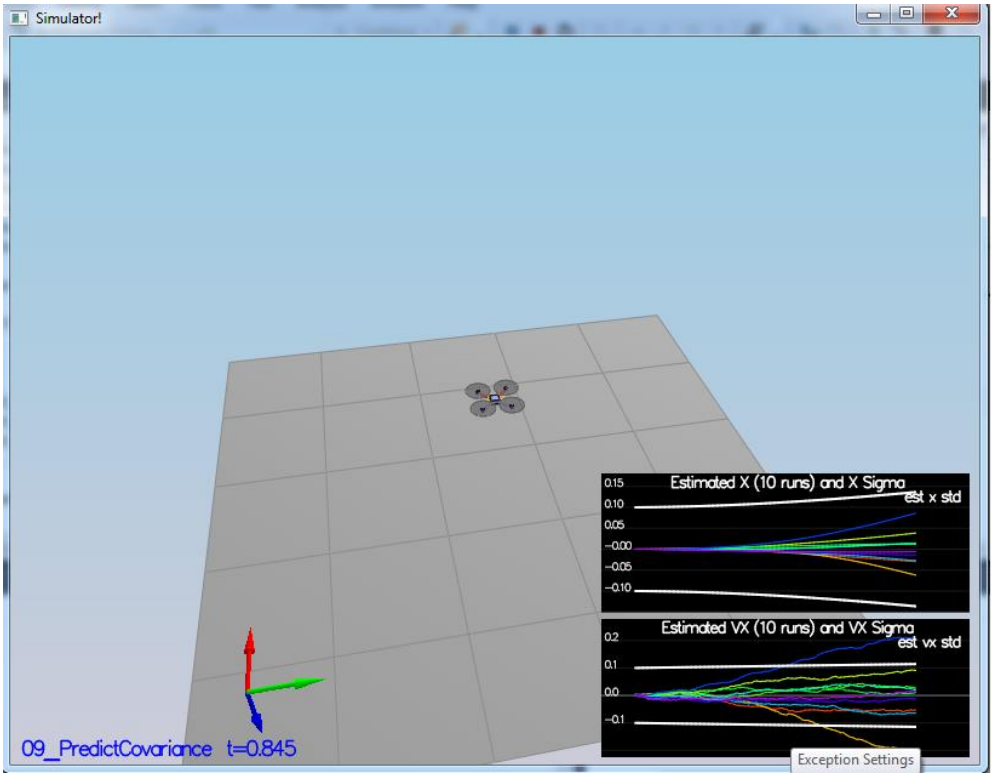
3: Prediction Step

Prediction step was implemented using the transition function to compute the next step. I have used tow extra variables A & B. A for the linear part and B for the nonlinear advance of the state. The code written for this:

```
VectorXf A(7);
A[0] = curState[0] + curState[3] * dt;
A[1] = curState[1] + curState[4] * dt;
A[2] = curState[2] + curState[5] * dt ;
A[3] = curState[3];
A[4] = curState[4];
A[5] = curState[5] - CONST_GRAVITY * dt; // Deducting the gravity in z-direction
A[6] = curState[6];

V3F s_dot_dot = attitude.Rotate_BtoI(accel);

VectorXf B(7);
B[0] = 0.0;
B[1] = 0.0;
B[2] = 0.0;
B[3] = s_dot_dot.x * dt;
B[4] = s_dot_dot.y * dt;
B[5] = s_dot_dot.z * dt;
B[6] = 0.0; // To avoids double integration
```

```
predictedState = A + B;


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

// filled  out the Rbg prime matrix
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));

// These are zero
RbgPrime(2, 0) = 0;
RbgPrime(2, 1) = 0;
RbgPrime(2, 2) = 0;

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

return RbgPrime;
```





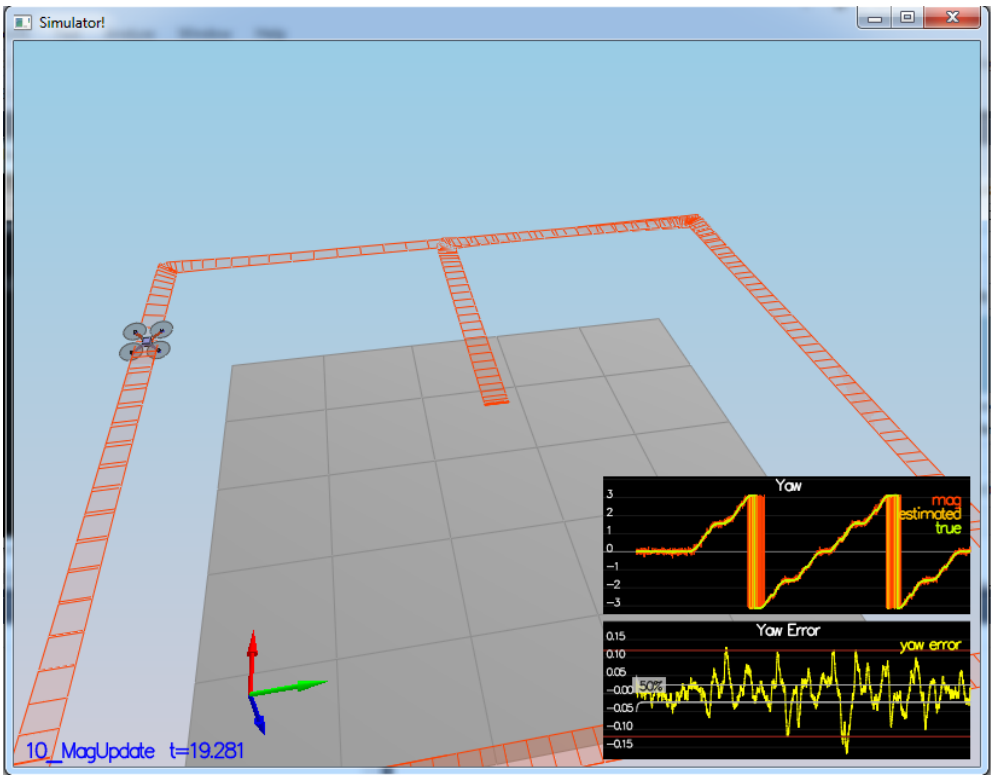
Success criteria: *This step doesn't have any specific measurable criteria being checked.*

4: Magnetometer Update

Used the following codes to update the Magnetometer update:

```
////////////////////////////// BEGIN STUDENT CODE ////////////////////////////
  hPrime(0,6) = 1;
  zFromX[0] = ekfState(6);

  float reading = magYaw;
  float current = ekfState(6);

  if (reading < 0){
    if (current > 0)
    {
      if (reading<-M_PI/2 && current> M_PI/2)
       reading = 2*M_PI + reading;
    }
  }
  else
  {
    if (current < 0)
      if (current < -M_PI/2 && reading> M_PI/2)
      {
         reading =  reading -2*M_PI;
      }
  }
```

```
  z(0) = reading;
  ////////////////////////////// END STUDENT CODE ////////////////////////////

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



Success criteria: *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.*

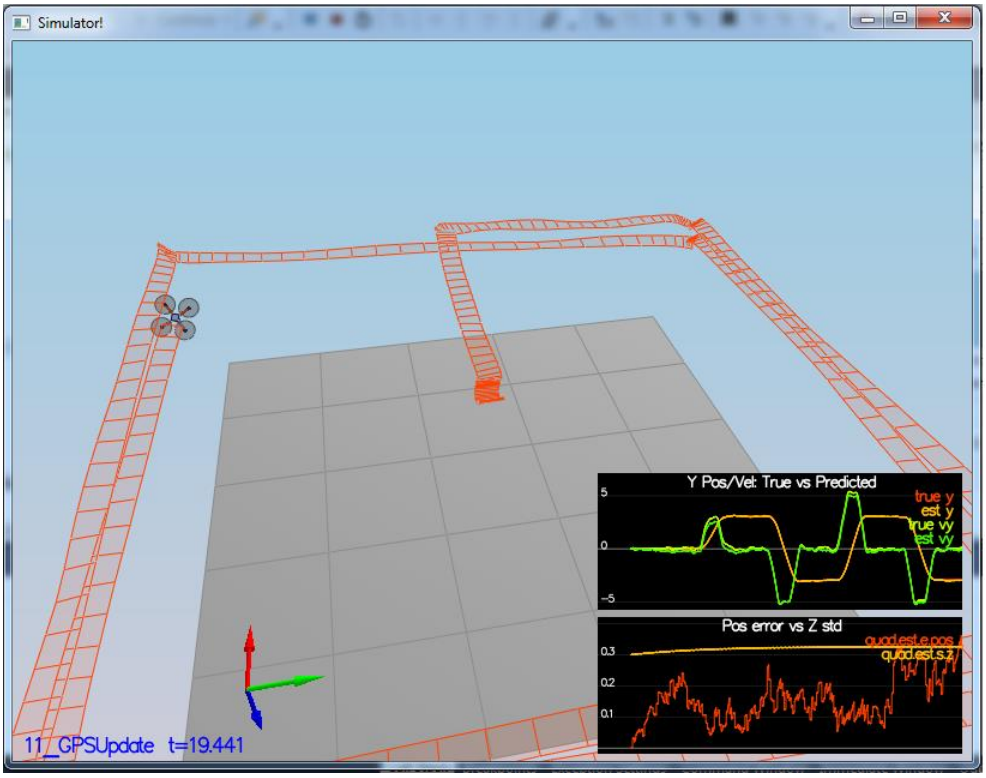
5: Closed Loop + GPS Update

The GPS update was done by coping the corresponding values and checked the correct range.

Code:

```
  ////////////////////////////// BEGIN STUDENT CODE ////////////////////////////
  hPrime(0,6) = 1;
  zFromX[0] = ekfState(6);

  float reading = magYaw;
  float current = ekfState(6);

  if (reading < 0){
```

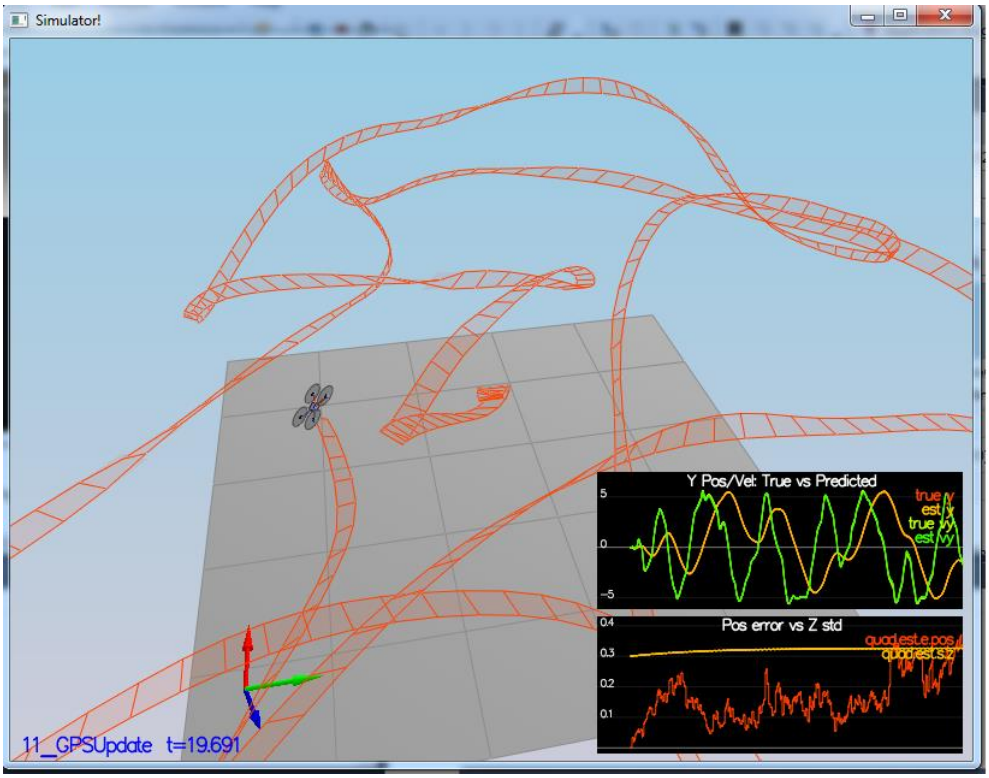
```
  if (current > 0)
  {
    if (reading<-M_PI/2 && current> M_PI/2)
     reading = 2*M_PI + reading;
  }
}
else
{
  if (current < 0)
    if (current < -M_PI/2 && reading> M_PI/2)
    {
       reading =  reading -2*M_PI;
    }
}

z(0) = reading;

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



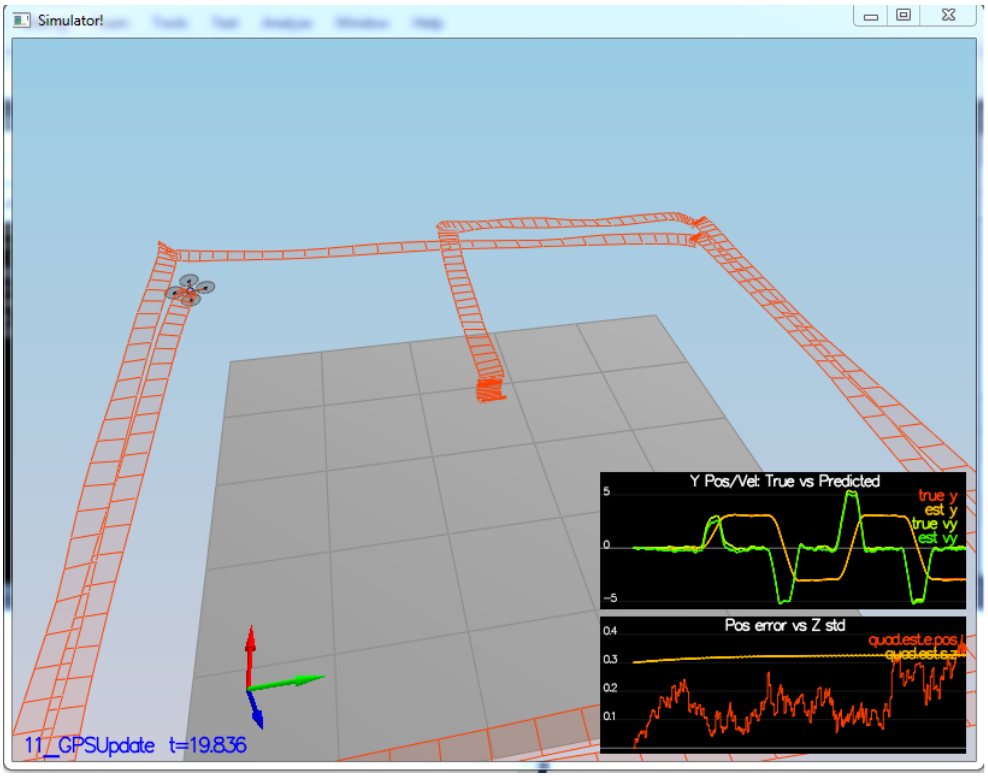
Success criteria: *Your objective is to complete the entire simulation cycle with estimated position error of < 1m.*

6: Adding Your Controller

Previous control project code was used for the controller for the EKF estimation project. The results were not very good as the flying looked like a drunk driver flying. But with the tuned controller the flying meets the success criteria. Will keep working on the tuning futher at my own time.



Previous controller with old parameter.



With tuned up Controller with tuned up parameter.

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

Submissions:

Submitted items for the project are:

a completed estimator that meets the performance criteria for each of the steps by submitting:

1. QuadEstimatorEKF.cpp
2. config/QuadEstimatorEKF.txt

a re-tuned controller that, in conjunction with your tuned estimator, is capable of meeting the criteria laid out in Step 6 by submitting:

3. QuadController.cpp
4. config/QuadControlParams.txt

5. A write up addressing all the points of the rubric