Estimation Project

Welcome to the estimation project. In this project, you will be developing the estimation portion of the controller used in the CPP simulator. By the end of the project, your simulated quad will be flying with your estimator and your custom controller (from the previous project)!

This README is broken down into the following sections:

- Setup - the environment and code setup required to get started and a brief overview of the project structure
- The Tasks - the tasks you will need to complete for the project
- Tips and Tricks - some additional tips and tricks you may find useful along the way
- Submission - overview of the requirements for your project submission

Setup

This project will continue to use the C++ development environment you set up in the Controls C++ project.

1. Clone the repository

```
git clone https://github.com/udacity/FCND-Estimation-CPP.git
```

2. Import the code into your IDE like done in the Controls C++ project

3. You should now be able to compile and run the estimation simulator just as you did in the controls project

Project Structure

For this project, you will be interacting with a few more files than before.

- The EKF is already partially implemented for you in `QuadEstimatorEKF.cpp`
- Parameters for tuning the EKF are in the parameter file `QuadEstimatorEKF.txt`
- When you turn on various sensors (the scenarios configure them, e.g. `Quad.Sensors += SimIMU, SimMag, SimGPS`), additional sensor plots will become available to see what the simulated sensors measure.

- The EKF implementation exposes both the estimated state and a number of additional variables. In particular:
    - `Quad.Est.E.X` is the error in estimated X position from true value. More generally, the variables in `<vehicle>.Est.E.*`are relative errors, though some are combined errors (e.g. MaxEuler).
    - `Quad.Est.S.X` is the estimated standard deviation of the X state (that is, the square root of the appropriate diagonal variable in the covariance matrix). More generally, the variables in `<vehicle>.Est.S.*` are standard deviations calculated from the estimator state covariance matrix.
    - `Quad.Est.D` contains miscellaneous additional debug variables useful in diagnosing the filter. You may or might not find these useful but they were helpful to us in verifying the filter and may give you some ideas if you hit a block.

`config` Directory

In the `config` directory, in addition to finding the configuration files for your controller and your estimator, you will also see configuration files for each of the simulations. For this project, you will be working with simulations 06 through 11 and you may find it insightful to take a look at the configuration for the simulation.
As an example, if we look through the configuration file for scenario 07, we see the following parameters controlling the sensor:

```
# Sensors
Quad.Sensors = SimIMU
# use a perfect IMU
SimIMU.AccelStd = 0,0,0
SimIMU.GyroStd = 0,0,0
```

This configuration tells us that the simulator is only using an IMU and the sensor data will have no noise. You will notice that for each simulator these parameters will change slightly as additional sensors are being used and the noise behavior of the sensors change.

The Tasks

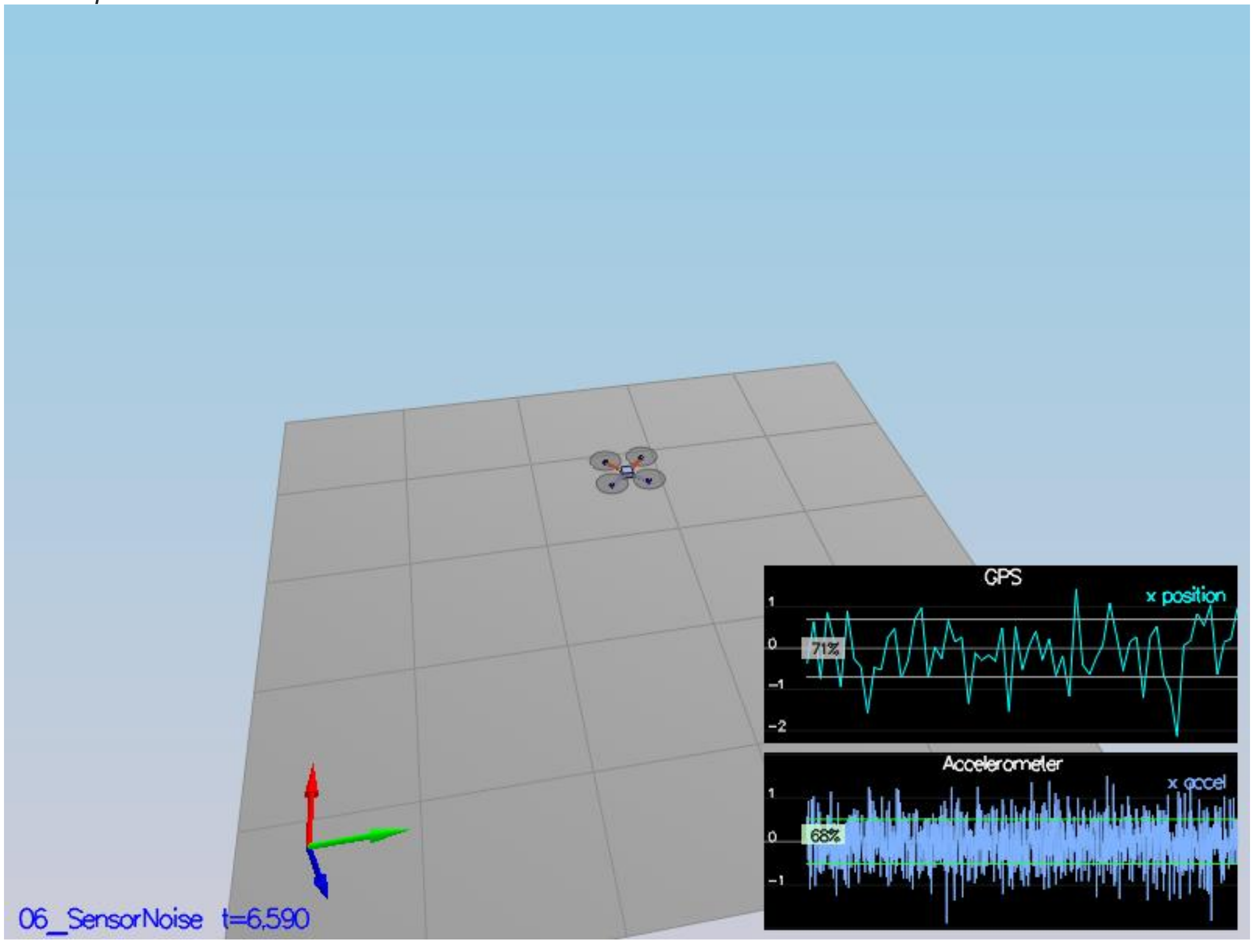
Once again, you will be building up your estimator in pieces. At each step, there will be a set of success criteria that will be displayed both in the plots and in the terminal output to help you along the way.

Project outline:

- Step 1: Sensor Noise

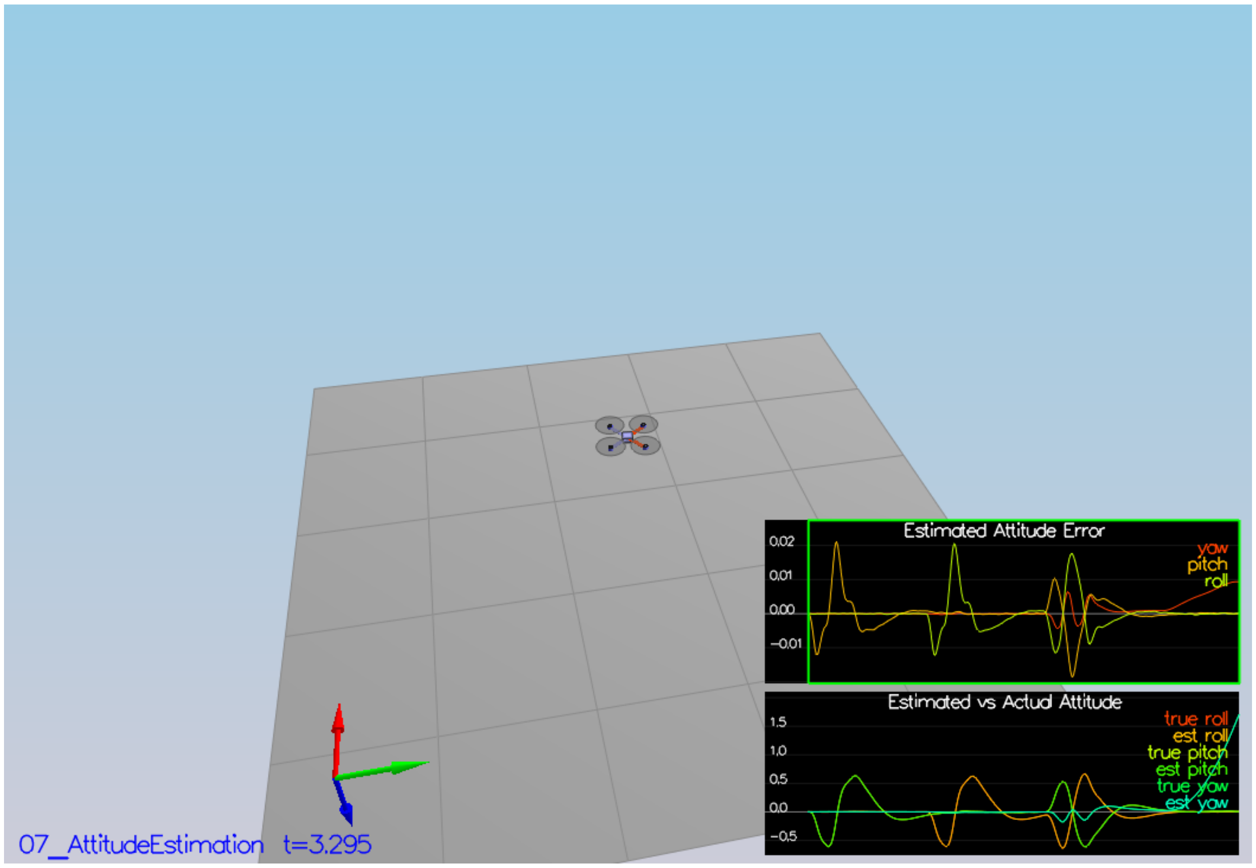
Step 1: Sensor Noise

Result: *The standard deviations was accurately capture the value of approximately 68% of the respective measurements.*



Step 2: Attitude Estimation

Result:



Success criteria: *Estimator was within 0.1 rad for each of the Euler angles for at least 3 seconds.*

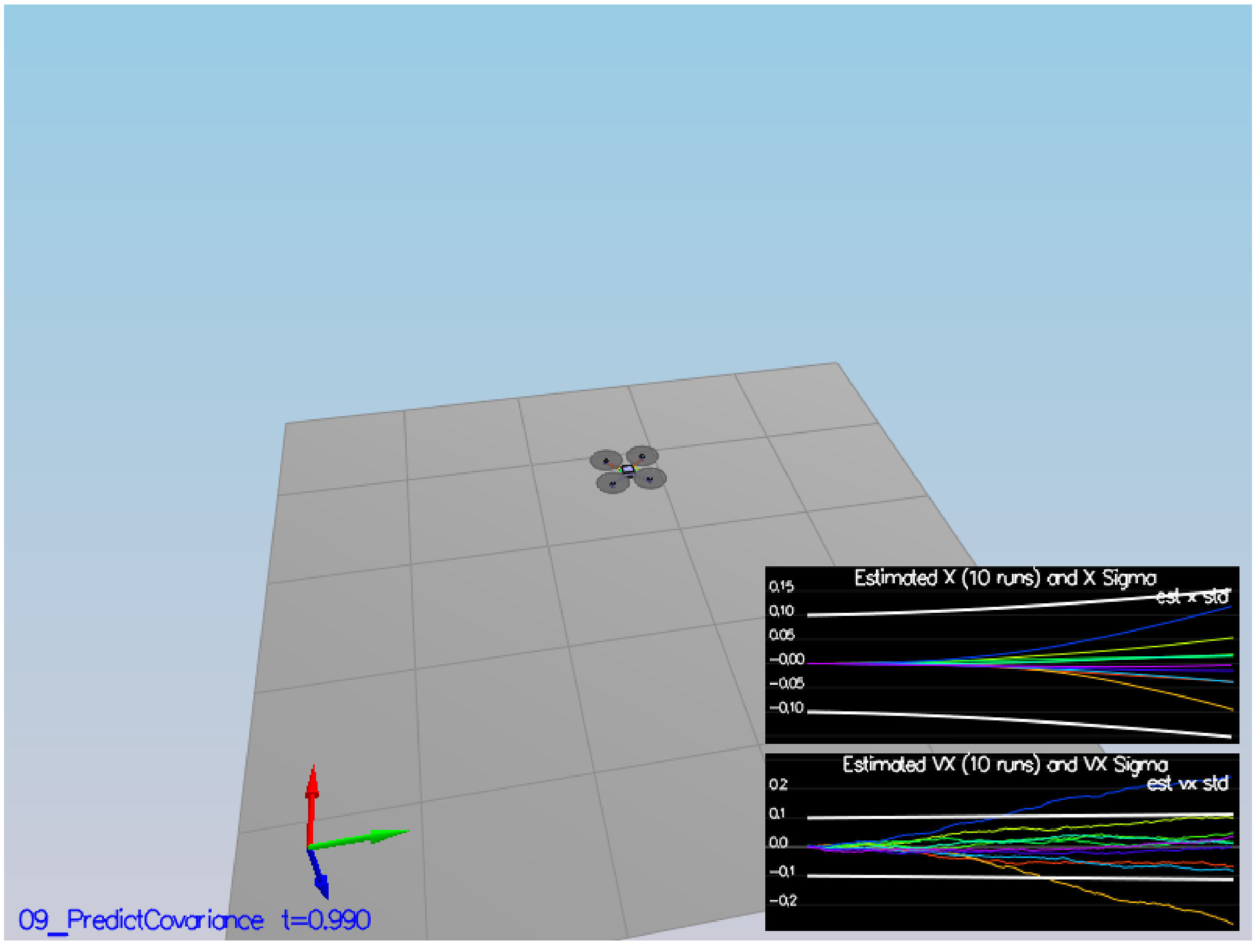
Code Snippet:

```
  ////////////////////////////// BEGIN STUDENT CODE ///////////////////////////
       Quaternion<float> quat = Quaternion<float>::FromEuler123_RPY(rollEst, pitchEst, 
ekfState(6));
       quat.IntegrateBodyRate(gyro, dtIMU);

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

  // SMALL ANGLE GYRO INTEGRATION:
  // (replace the code below)
  // make sure you comment it out when you add your own code -- otherwise e.g. you might 
integrate yaw twice

  //float predictedPitch = pitchEst + dtIMU * gyro.y;
  //float predictedRoll = rollEst + dtIMU * gyro.x;
  //ekfState(6) = ekfState(6) + dtIMU * gyro.z; // 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;

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

Step 3: Prediction Step

Result:


Estimated X (10 runs) and X Sigma
est x std
0.15
0.10
0.05
-0.00
-0.05
-0.10
Estimated VX (10 runs) and VX Sigma
est vx std
0.2
0.1
0.0
-0.1
-0.2
09_PredictCovariance t=0.990

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

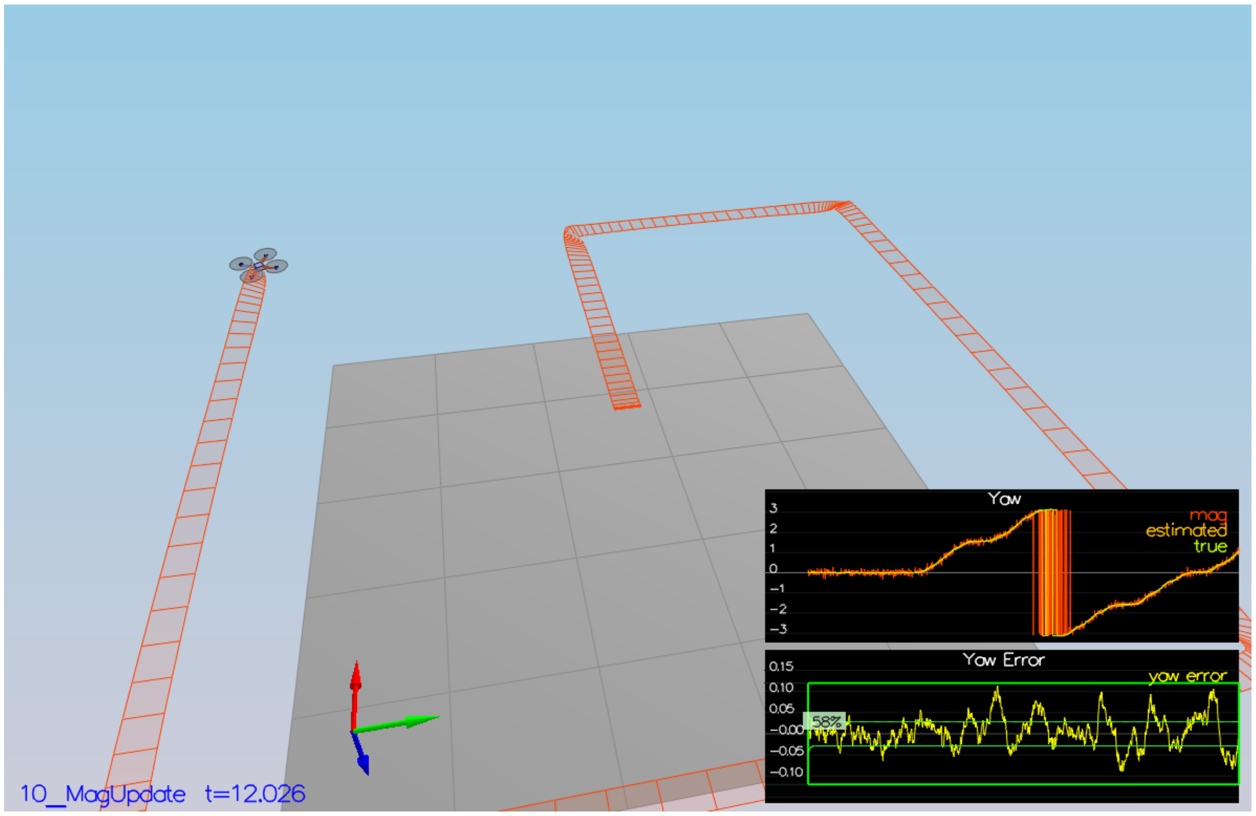
Code Snippet:

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

RbgPrime(0, 0) = -cos(pitch) * sin(yaw);
RbgPrime(0, 1) = -sin(roll)*sin(pitch)*sin(yaw) - cos(pitch)*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 ////////////////////////////
```

Step 4: Magnetometer Update

Result:



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. The result meet the target.*

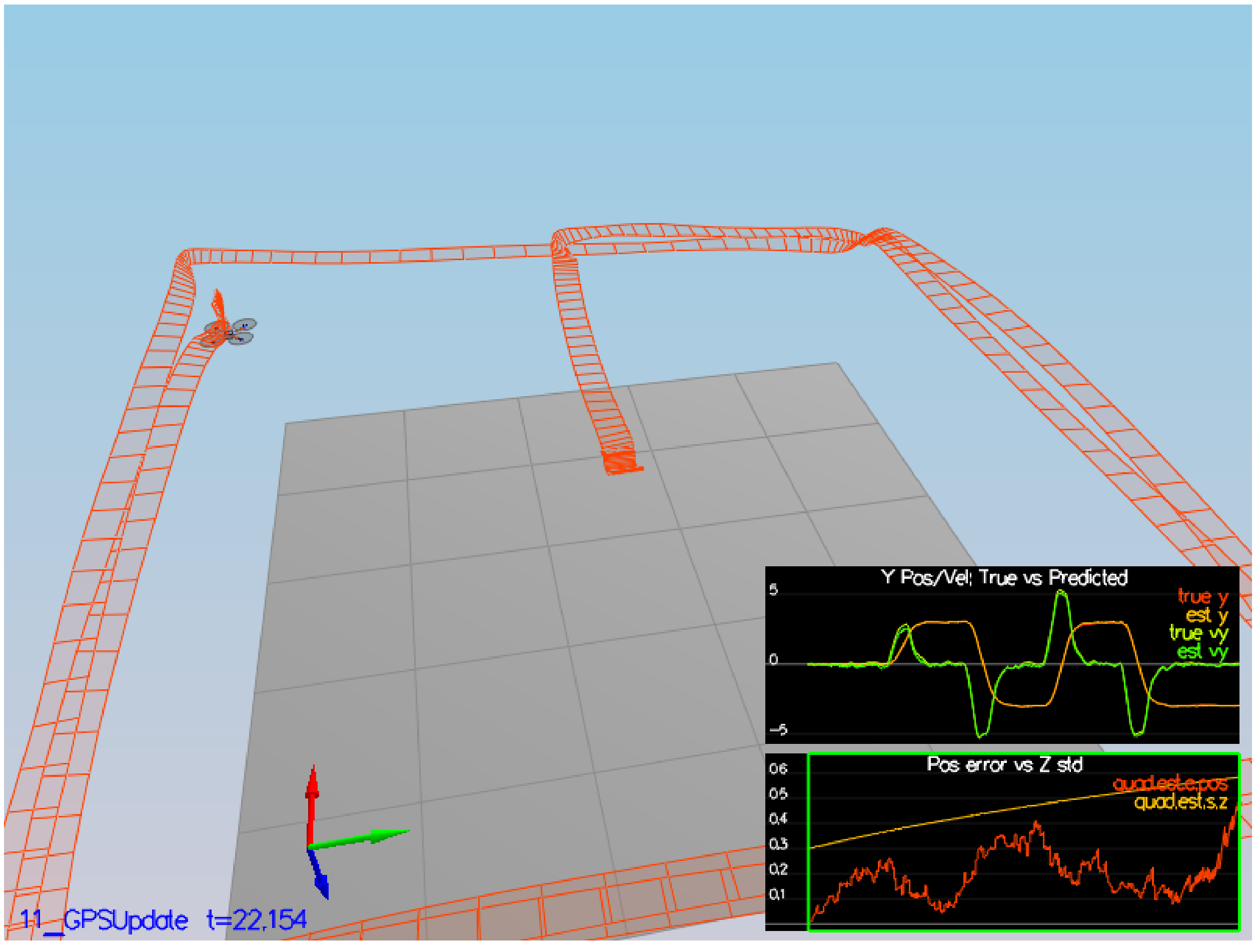
Code Snippet:

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

hPrime(6) = 1;

zFromX(0) = ekfState(6);
float diff = magYaw - ekfState(6);
if (diff > F_PI) {
       zFromX(0) += 2.f*F_PI;
}
else if (diff < -F_PI) {
       zFromX(0) -= 2.f*F_PI;
}
```

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

Step 5: Closed Loop + GPS Update

Result



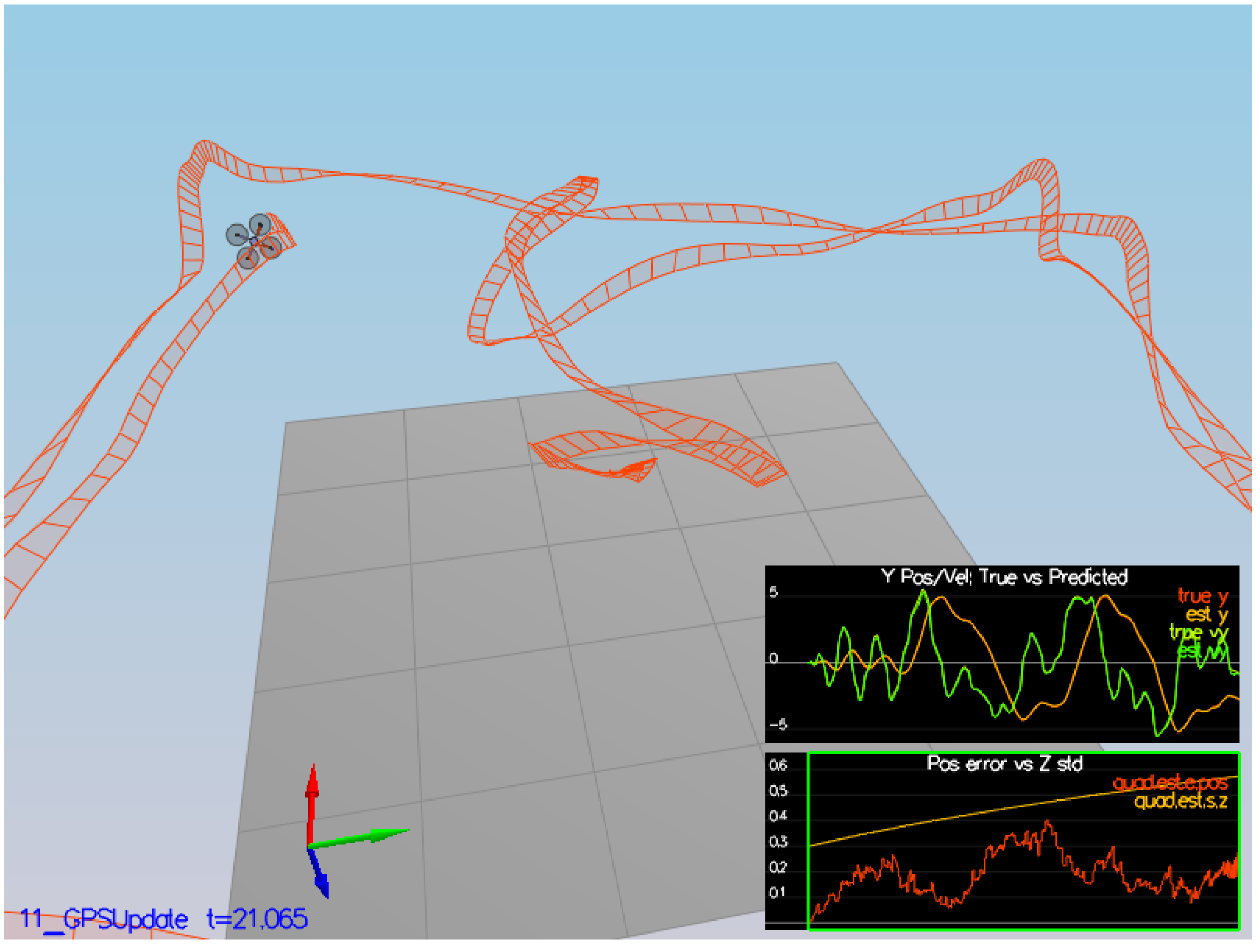
Success criteria: *Your objective is to complete the entire simulation cycle with estimated position error of < 1m. The result shows that target has been met.*

Code Snippet:

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

  hPrime.topLeftCorner(QUAD_EKF_NUM_STATES - 1, QUAD_EKF_NUM_STATES - 1) =
MatrixXf::Identity(QUAD_EKF_NUM_STATES - 1, QUAD_EKF_NUM_STATES - 1);
  zFromX = hPrime * ekfState;

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

Step 6: Adding Your Controller

Result:



Success criteria: *Objective is to complete the entire simulation cycle with estimated position error of < 1m. The result has met the success criteria.*

Submission

Following has been submitted.

For this project, you will need to submit:

- a completed estimator that meets the performance criteria for each of the steps by submitting:
  - `QuadEstimatorEKF.cpp`
  - `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:
    - `QuadController.cpp`
    - `config/QuadControlParams.txt`
- a write up addressing all the points of the rubric