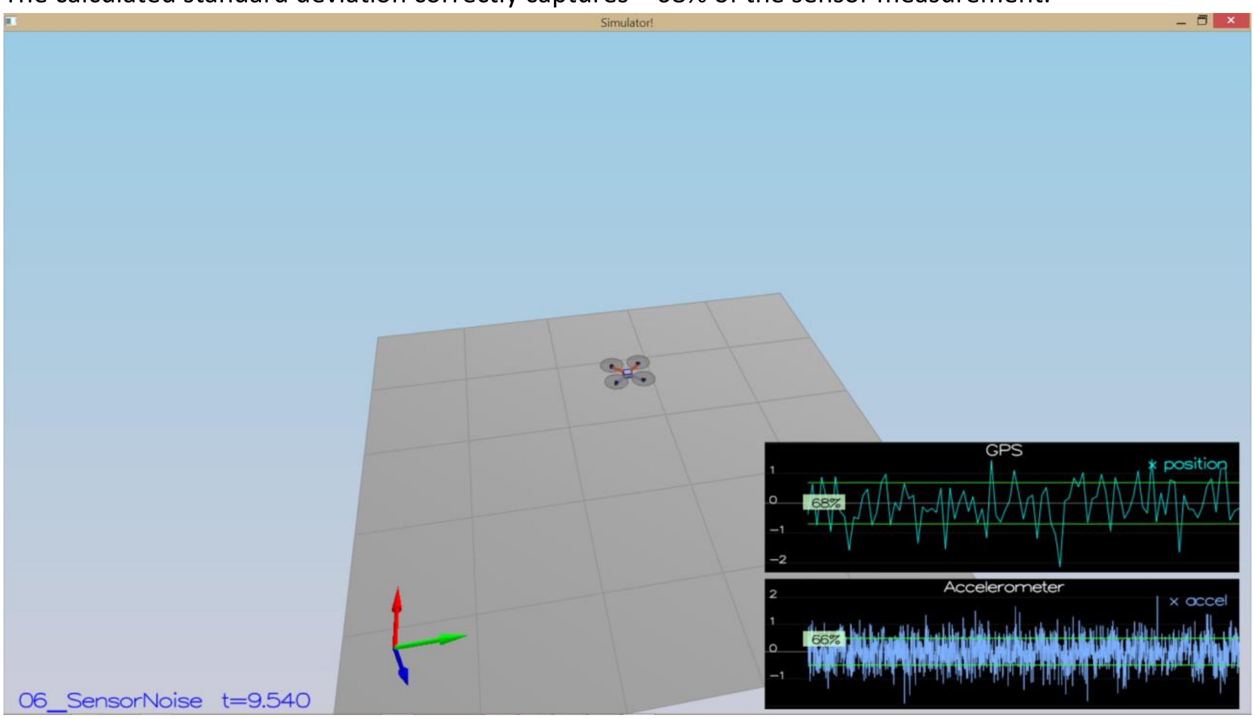
Building an Estimator

Provide a Writeup / README that includes all the rubric points and how you addressed each one. You can submit your writeup as markdown or pdf.

a) You are reading the writeup ☺

Determine the standard deviation of the measurement noise of both GPS X data and Accelerometer X data.

a) The calculated standard deviation correctly captures ~ 68% of the sensor measurement.

b) 

c) Took data from log file and exported it to Microsoft Excel. I then used Excel's standard deviation formula (sample) to calculate standard deviation given simulated sensor measurements. Microsoft Excel uses the following Standard deviation formula :

d) $$s_x = \sqrt{\frac{\sum_{i=1}^{n}(x_i - \bar{x})^2}{n-1}}$$

e) Where n = number data observations, X_i is the i[th] data observation, X_bar is the mean of all data in sample.

Implement a better rate gyro attitude integration scheme in the `UpdateFromIMU()` function.

a) The improved integration scheme resulted in an attitude estimator of less than 0.1 rad for 3 seconds. (The max absolute error seems to be 0.015 rad based on a visual inspection of the graphs).

b) I created a Euler Rate Matrix based on the current Euler angles then converted the body rates to Euler rates by multiplying the matrix with the body rates. I then integrated that to update the current Euler angles. Euler Rate Matrix formula and code shown below:

$$[E'_{123}(\phi,\theta,\psi)]^{-1} = \frac{1}{c_\theta}\begin{bmatrix} c_\theta & s_\phi s_\theta & c_\phi s_\theta \\ 0 & c_\phi c_\theta & -s_\phi c_\theta \\ 0 & s_\phi & c_\phi \end{bmatrix}.$$

```
Simulator    QuadEstimatorEKF    UpdateFromIMU(V3F accel, V3F gyro)

// 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;
// predictedRoll = rollEst + dtIMU * gyro.x;
//ekfState(6) = ekfState(6) + dtIMU * gyro.z; // yaw

float phi, theta, psi;
phi = rollEst;
theta = pitchEst;
psi = ekfState(6);

V3F body_rates(gyro.x, gyro.y, gyro.z);
//printf("phi and sin(phi) \n");
//printf("%s %s \n", to_string(phi).c_str(), to_string(sin(phi)).c_str());

float v[9] = {1.0f, tan(theta)*sin(phi), tan(theta)*cos(phi), 0.0f, cos(phi), -1*sin(phi), 0.0f, sin(phi) / cos(theta), cos(phi) / cos(theta) };
Mat3x3F R_dot = v;
V3F euler_rates = R_dot * body_rates;
float predictedPitch = pitchEst + dtIMU * euler_rates[1];
float predictedRoll = rollEst + dtIMU * euler_rates[0];
ekfState(6) = ekfState(6) + dtIMU * euler_rates[2];   // 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;

```

Implement all of the elements of the prediction step for the estimator.

a) The prediction step includes the state update element (`PredictState()` function) as seen in code below:

```
VectorXf QuadEstimatorEKF::PredictState(VectorXf curState, float dt, V3F accel, V3F gyro)
{
  assert(curState.size() == QUAD_EKF_NUM_STATES);
  VectorXf predictedState = curState;
  // ...

  // ...

  Quaternion<float> attitude = Quaternion<float>::FromEuler123_RPY(rollEst, pitchEst, curState(6));
  ////////////////////////////// BEGIN STUDENT CODE ///////////////////////////
  Matrix<float, 7, 4> B;
  Matrix<float, 7, 1> A;
  V3F accel_worldf;
  V3F gravity(0, 0, CONST_GRAVITY);
  V3F true_accel = accel;
  gravity = attitude.Rotate_ItoB(gravity);
  true_accel = true_accel - gravity;
  accel_worldf = attitude.Rotate_BtoI(true_accel);
  predictedState(0) = predictedState(0) + predictedState(3) * dt;
  predictedState(1) = predictedState(1) + predictedState(4) * dt;
  predictedState(2) = predictedState(2) + predictedState(5) * dt;
  predictedState(3) = predictedState(3) + accel_worldf[0] * dt;
  predictedState(4) = predictedState(4) + accel_worldf[1] * dt;
  predictedState(5) = predictedState(5) + accel_worldf[2]* dt;
  /////////////////////////////// END STUDENT CODE ////////////////////////////

  return predictedState;
}
```

b)

c) As can be seen implemented in the code below:

a. Calculated the Rgb prime matrix by taking the partial derivatives wrt to yaw angle.

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

b. The acceleration is accounted for as a command in the calculation of gPrime; ***VectorXf RbgPrime_u = RbgPrime * accel_vec;***

c. and the covariance update follows the classic EKF update equation: ***ekfCov_p = gPrime * ekfCov * gPrime_T + Q (for prediction)*** ***and*** ***ekfCov = (eye - K*H)*ekfCov***

d)

```cpp
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 ///////////////////////////
  float phi = roll;
  float theta = pitch;
  float psi = yaw;

  RbgPrime(0, 0) = -1 * cos(theta)*sin(psi);
  RbgPrime(0, 1) = -1 * sin(phi)*sin(theta)*sin(psi) - 1 * cos(phi)*cos(psi);
  RbgPrime(0, 2) = -1 * cos(phi)*sin(theta)*sin(psi) + sin(phi)*cos(psi);
  RbgPrime(1, 0) = cos(theta)*cos(phi);
  RbgPrime(1, 1) = sin(phi)*sin(theta)*cos(psi) - 1 * cos(phi)*sin(psi);
  RbgPrime(1, 2) = cos(phi)*sin(theta)*cos(psi) + sin(phi)*sin(psi);
  RbgPrime(2, 0) = 0;
  RbgPrime(2, 1) = 0;
  RbgPrime(2, 2) = 0;
```

e)

```cpp
void QuadEstimatorEKF::Predict(float dt, V3F accel, V3F gyro)
{
  // predict the state forward
  VectorXf newState = PredictState(ekfState, dt, accel, gyro);

  // ...

  // ...

  // we'll want the partial derivative of the Rbg matrix
  MatrixXf RbgPrime = GetRbgPrime(rollEst, pitchEst, ekfState(6));

  // we've created an empty Jacobian for you, currently simply set to identity
  MatrixXf gPrime(QUAD_EKF_NUM_STATES, QUAD_EKF_NUM_STATES);
  gPrime.setIdentity();

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

  MatrixXf ekfCov_p(QUAD_EKF_NUM_STATES, QUAD_EKF_NUM_STATES);
  VectorXf accel_vec(3);
  accel_vec(0) = accel[0];
  accel_vec(1) = accel[1];
  accel_vec(2) = accel[2];
  VectorXf RbgPrime_u = RbgPrime * accel_vec;
  gPrime(0, 3) = dt;
  gPrime(1, 4) = dt;
  gPrime(2, 5) = dt;
  gPrime(3, 6) = RbgPrime_u(0) * dt;
  gPrime(4, 6) = RbgPrime_u(1) * dt;
  gPrime(5, 6) = RbgPrime_u(2) * dt;
  MatrixXf gPrime_T = gPrime.transpose();
  ekfCov_p = gPrime * ekfCov * gPrime_T  + Q;
  ekfCov = ekfCov_p;
  ///////////////////////////// END STUDENT CODE ////////////////////////////

  ekfState = newState;
```

Implement the magnetometer update.

a) Incorporated magnetometer data into state and normalized the difference between current state and magnetometer value to ensure it takes the shorter way around.

b)

```
    zFromX(4) = ekfState(4);
    zFromX(5) = ekfState(5);
    ////////////////////////////// END STUDENT CODE ///////////////////////////

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

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 ///////////////////////////

    zFromX(0) = ekfState(6);
    hPrime(6) = 1;
    if (z(0) - zFromX(0) > F_PI) z(0) += 2.f*F_PI;
    if (z(0) -zFromX(0) < -F_PI) z(0) -= 2.f*F_PI;

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

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

Implement the GPS update.

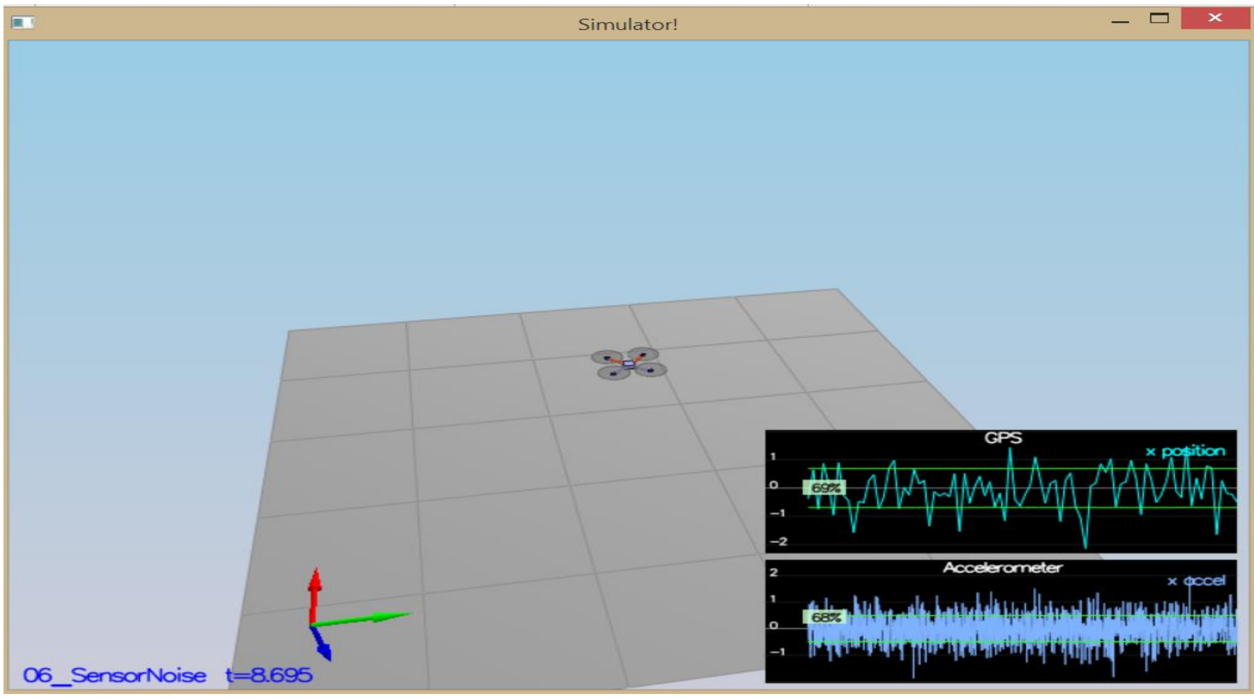
The estimator should correctly incorporate the GPS information to update the current state estimate as seen in image below:

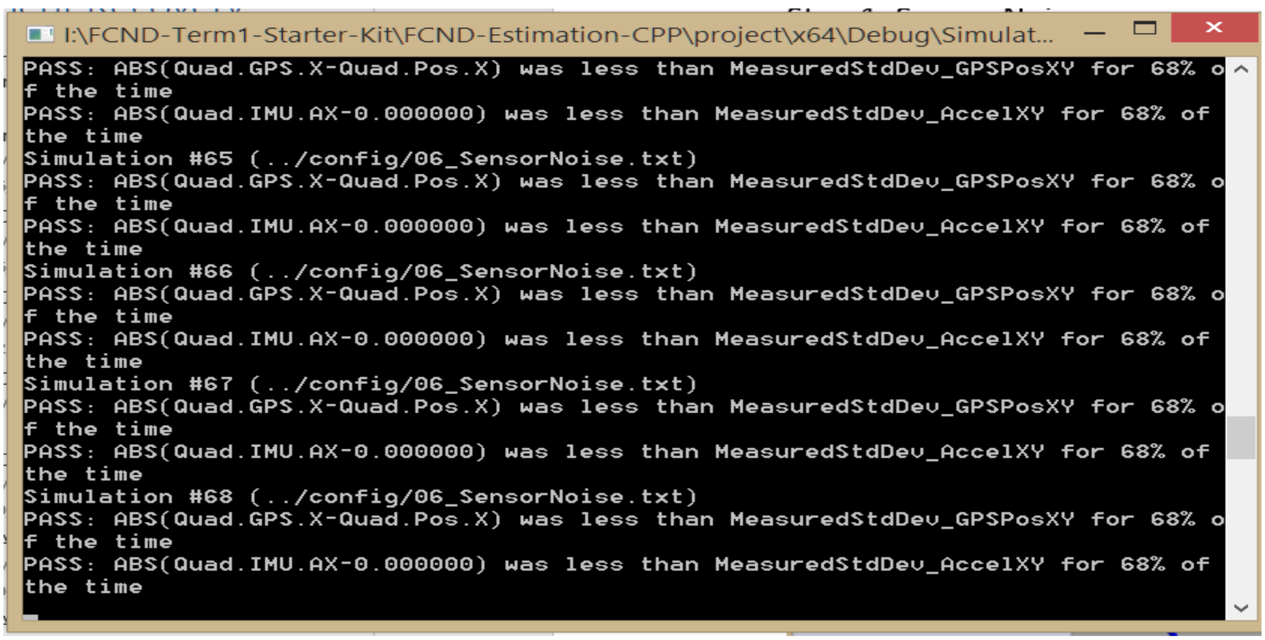
Simulator → QuadEstimatorEKF → UpdateFromGPS(V3F pos, V3F vel)

```
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.setIdentity();
    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);
}
```

Meet the performance criteria of each step.

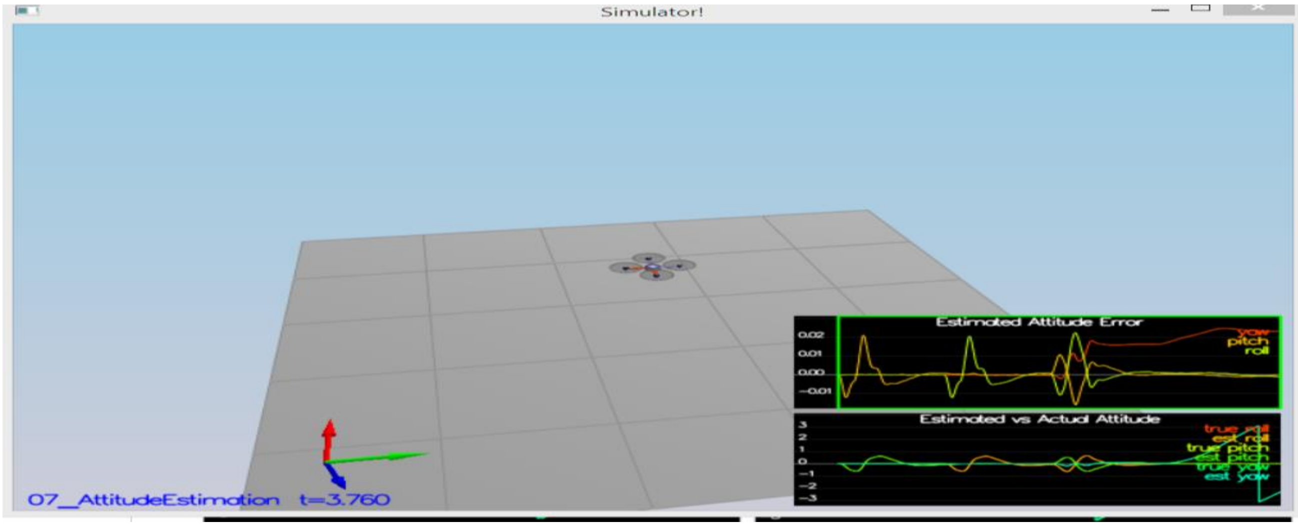
Step 1: Sensor Noise : *standard deviations should accurately capture the value of approximately 68% of the respective measurements. This criterion was met as seen in images below:*





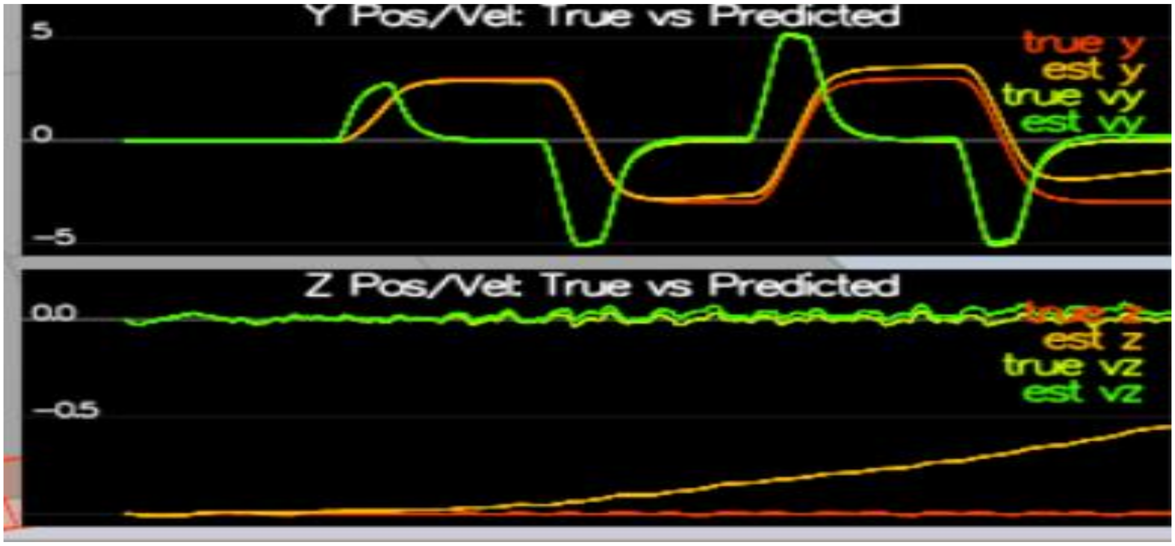
```
I:\FCND-Term1-Starter-Kit\FCND-Estimation-CPP\project\x64\Debug\Simulat...
PASS: ABS(Quad.GPS.X-Quad.Pos.X) was less than MeasuredStdDev_GPSPosXY for 68% o
f the time
PASS: ABS(Quad.IMU.AX-0.000000) was less than MeasuredStdDev_AccelXY for 68% of
the time
Simulation #65 (../config/06_SensorNoise.txt)
PASS: ABS(Quad.GPS.X-Quad.Pos.X) was less than MeasuredStdDev_GPSPosXY for 68% o
f the time
PASS: ABS(Quad.IMU.AX-0.000000) was less than MeasuredStdDev_AccelXY for 68% of
the time
Simulation #66 (../config/06_SensorNoise.txt)
PASS: ABS(Quad.GPS.X-Quad.Pos.X) was less than MeasuredStdDev_GPSPosXY for 68% o
f the time
PASS: ABS(Quad.IMU.AX-0.000000) was less than MeasuredStdDev_AccelXY for 68% of
the time
Simulation #67 (../config/06_SensorNoise.txt)
PASS: ABS(Quad.GPS.X-Quad.Pos.X) was less than MeasuredStdDev_GPSPosXY for 68% o
f the time
PASS: ABS(Quad.IMU.AX-0.000000) was less than MeasuredStdDev_AccelXY for 68% of
the time
Simulation #68 (../config/06_SensorNoise.txt)
PASS: ABS(Quad.GPS.X-Quad.Pos.X) was less than MeasuredStdDev_GPSPosXY for 68% o
f the time
PASS: ABS(Quad.IMU.AX-0.000000) was less than MeasuredStdDev_AccelXY for 68% of
the time
```

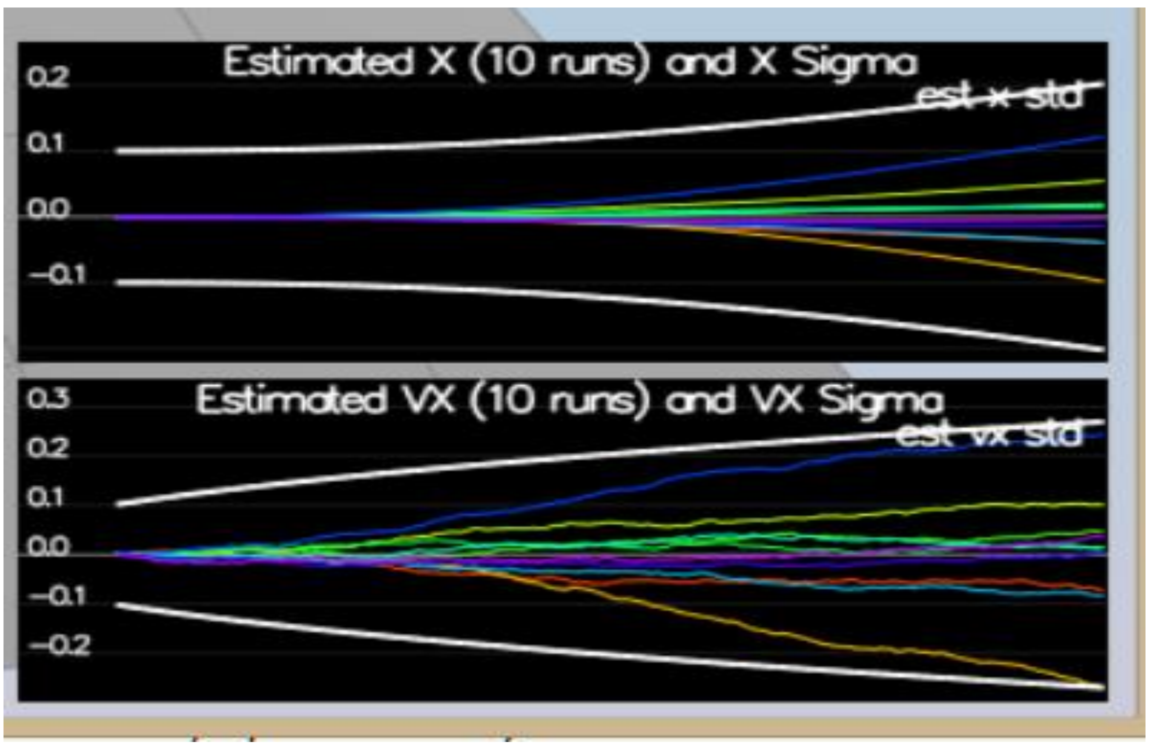
Step 2: Attitude Estimation : *our attitude estimator needs to get within 0.1 rad for each of the Euler angles for at least 3 seconds. The criterion was met as can be seen in images below:*



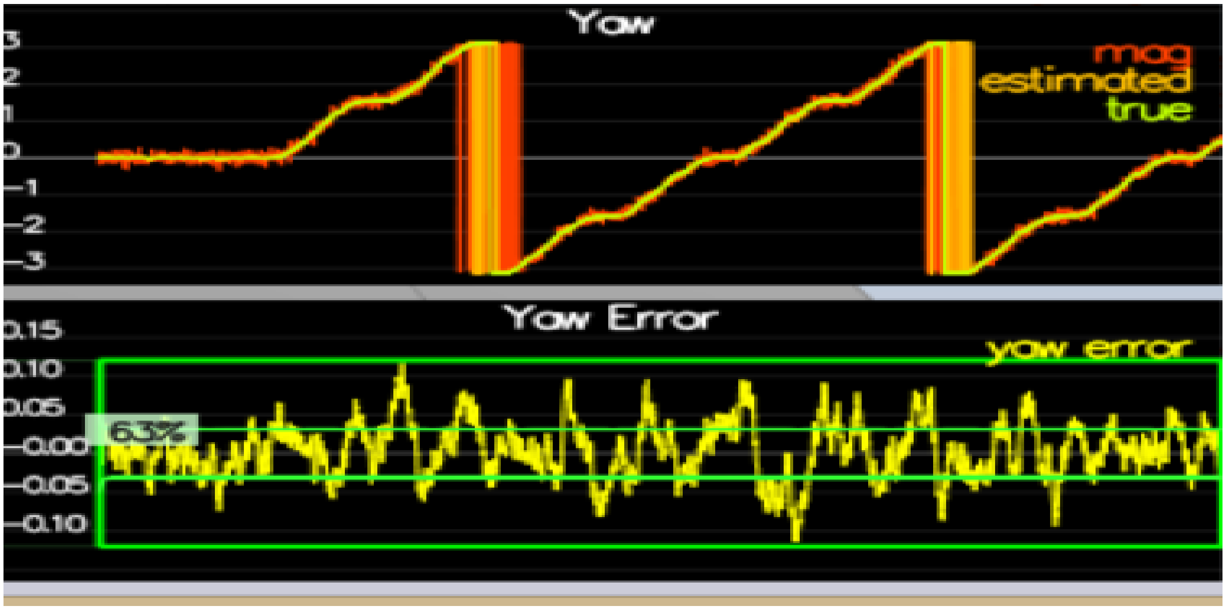

```
nds
Simulation #86 (../config/07_AttitudeEstimation.txt)
PASS: ABS(Quad.Est.E.MaxEuler) was less than 0.100000 for at least 3.000000 seco
nds
Simulation #87 (../config/07_AttitudeEstimation.txt)
PASS: ABS(Quad.Est.E.MaxEuler) was less than 0.100000 for at least 3.000000 seco
nds
Simulation #88 (../config/07_AttitudeEstimation.txt)
PASS: ABS(Quad.Est.E.MaxEuler) was less than 0.100000 for at least 3.000000 seco
nds
Simulation #89 (../config/07_AttitudeEstimation.txt)
PASS: ABS(Quad.Est.E.MaxEuler) was less than 0.100000 for at least 3.000000 seco
nds
```

Step 3: Prediction Step : *This step doesn't have any specific measurable criteria being checked. However moderate/slow drift can be seen between true state and estimated state and covariance appears to grow appropriately like the data. Please see images below:*



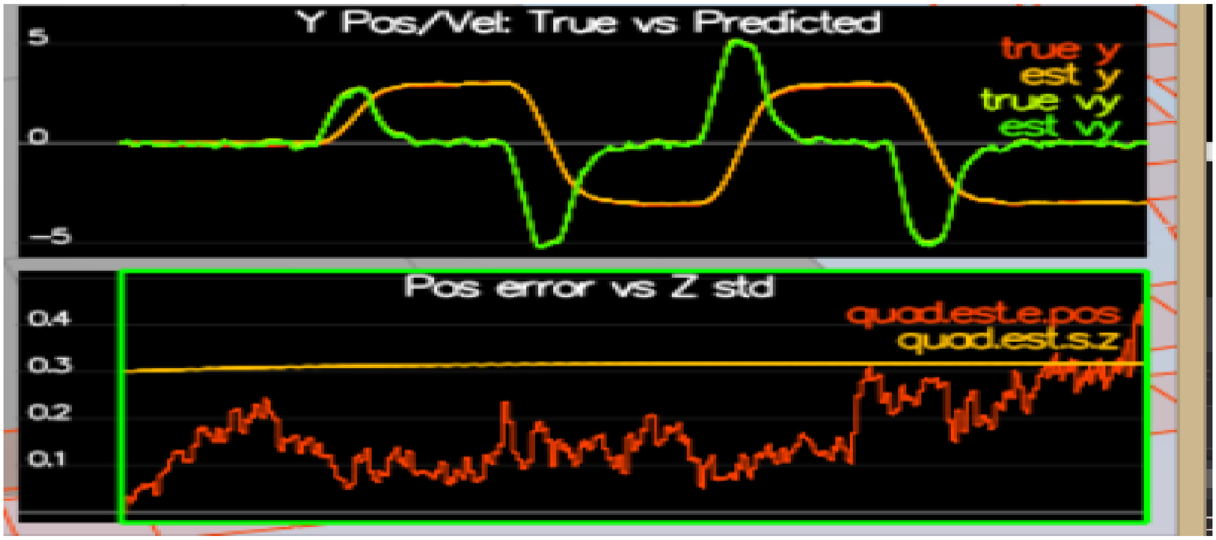


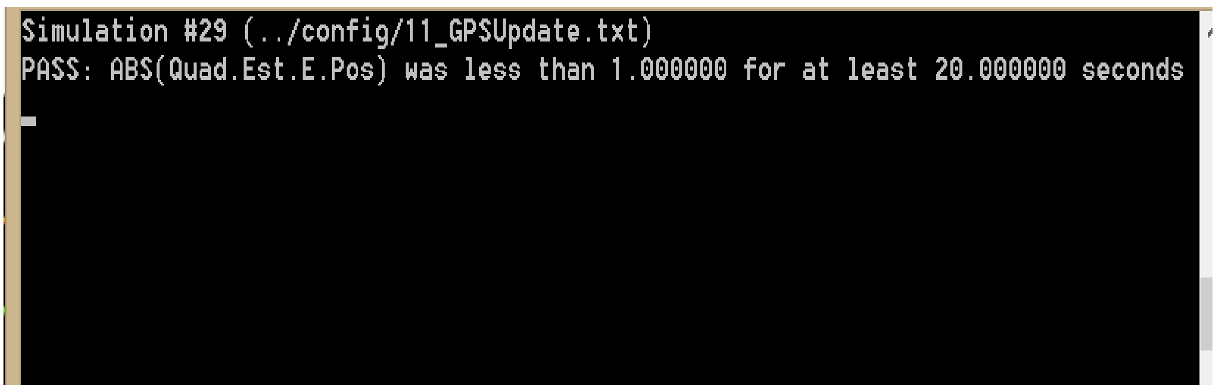
Step 4: Magnetometer Update: *Your goal is to both have an estimated standard deviation that accurately captures the error and maintain an error of less than 0.1rad in heading for at least 10 seconds of the simulation. This goal was met as can be seen in the images below:*




```
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 64% of the t
ime
Simulation #5 (../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 64% of the t
ime
```

Step 5: Closed Loop + GPS Update: *Your objective is to complete the entire simulation cycle with estimated position error of < 1m. The criterion was met as seen in images below:*

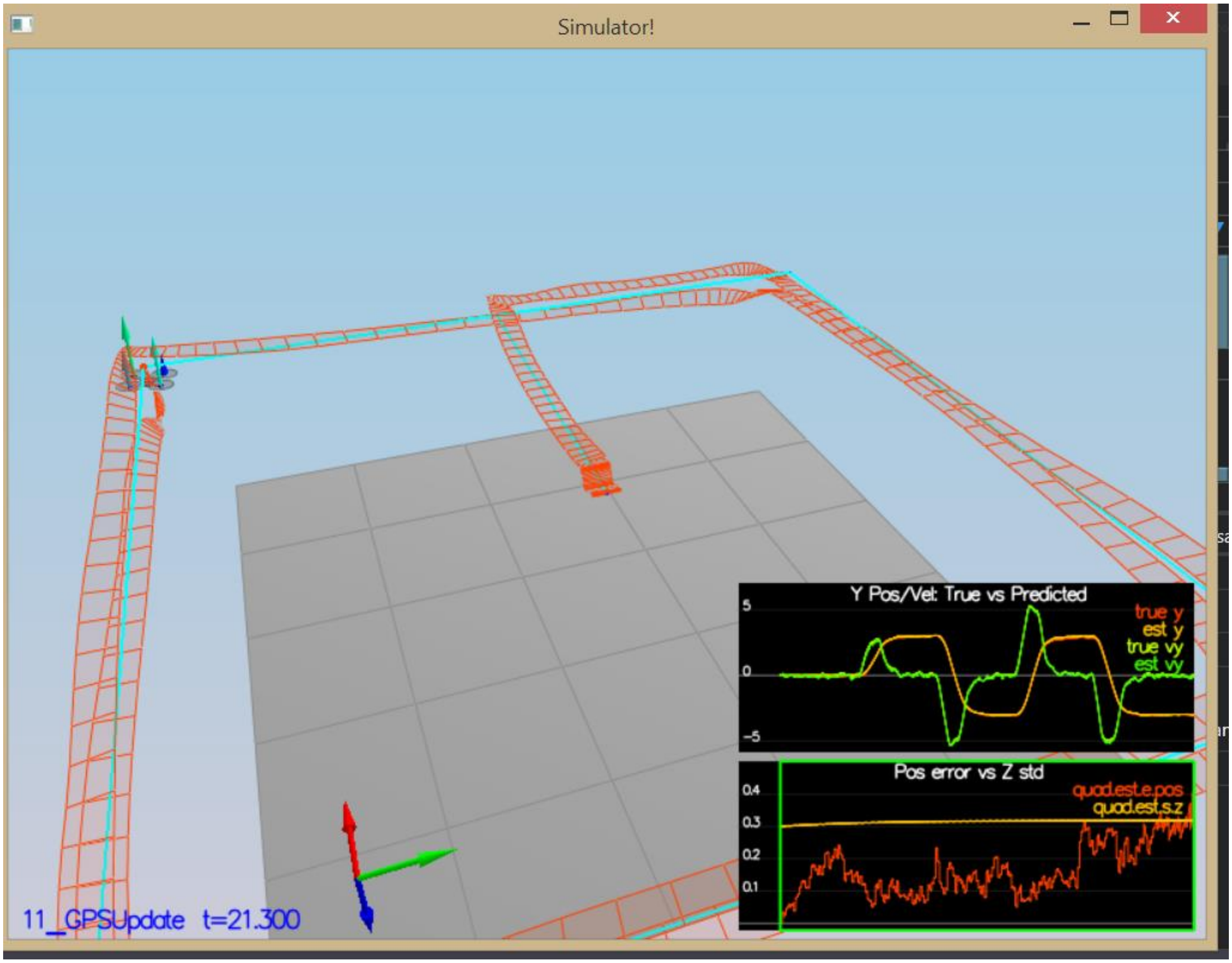



Simulation #29 (../config/11_GPSUpdate.txt)
PASS: ABS(Quad.Est.E.Pos) was less than 1.000000 for at least 20.000000 seconds

De-tune your controller to successfully fly the final desired box trajectory with your estimator and realistic sensors.

a) De-tuned controller gains and was able to successfully fly the entire box with <1 m error for box flight b)

```
# Physical properties
Mass = 0.5
L = 0.17
Ixx = 0.0023
Iyy = 0.0023
Izz = 0.0046
kappa = 0.016
minMotorThrust = .1
maxMotorThrust = 4.5

# Position control gains  36 , 60 , and 25
kpPosXY = 2
kpPosZ =  10
KiPosZ =  1

# Velocity control gains 15 and 25
kpVelXY =  10
kpVelZ =  1

# Angle control gains 8.3 and 1.8
kpBank = 10.3
kpYaw =  20

# Angle rate gains 61.5, 25.6, 5
kpPQR = 60, 60, 2
```




```
Simulation #7394 (../config/11_GPSUpdate.txt)
PASS: ABS(Quad.Est.E.Pos) was less than 1.000000 for at least 20.000000 seconds
Simulation #7395 (../config/11_GPSUpdate.txt)
PASS: ABS(Quad.Est.E.Pos) was less than 1.000000 for at least 20.000000 seconds
Simulation #7396 (../config/11_GPSUpdate.txt)
PASS: ABS(Quad.Est.E.Pos) was less than 1.000000 for at least 20.000000 seconds
Simulation #7397 (../config/11_GPSUpdate.txt)
PASS: ABS(Quad.Est.E.Pos) was less than 1.000000 for at least 20.000000 seconds
```