Flying Car and Autonomous Flight Engineer Nanodegree

Project 4 – Estimation

OBJECTIVE : Build a state estimator for the quadrotor to satisfy the conditions of the following scenarios. These are a continuation of the scenarios from the control project. The project is completed using an ideal controller and in the last step, the ideal controller is replaced with the controller from the previous project and the parameters are re-tuned.

1. Scenario 6 – Estimate sensor noise
2. Scenario 7 – Estimate attitude
3. Scenario 8,9 – Implement EKF prediction step
4. Scenario 10 – Implement magnetometer update
5. Scenario 11a – Implement GPS update
6. Scenario 11b – Replace controller and re-tune

APPROACH : The section 7 of the Estimation for Quadrotors PDF accompanying the project was used as reference to implement all the above scenarios. An EKF is used to estimate 7 states.

$$x_t = \begin{bmatrix} x \\ y \\ z \\ \dot{x} \\ \dot{y} \\ \dot{z} \\ \psi \end{bmatrix}$$

A non-linear complementary filter is used to estimate roll and pitch.

$$x_t = \begin{bmatrix} \theta \\ \phi \end{bmatrix}$$

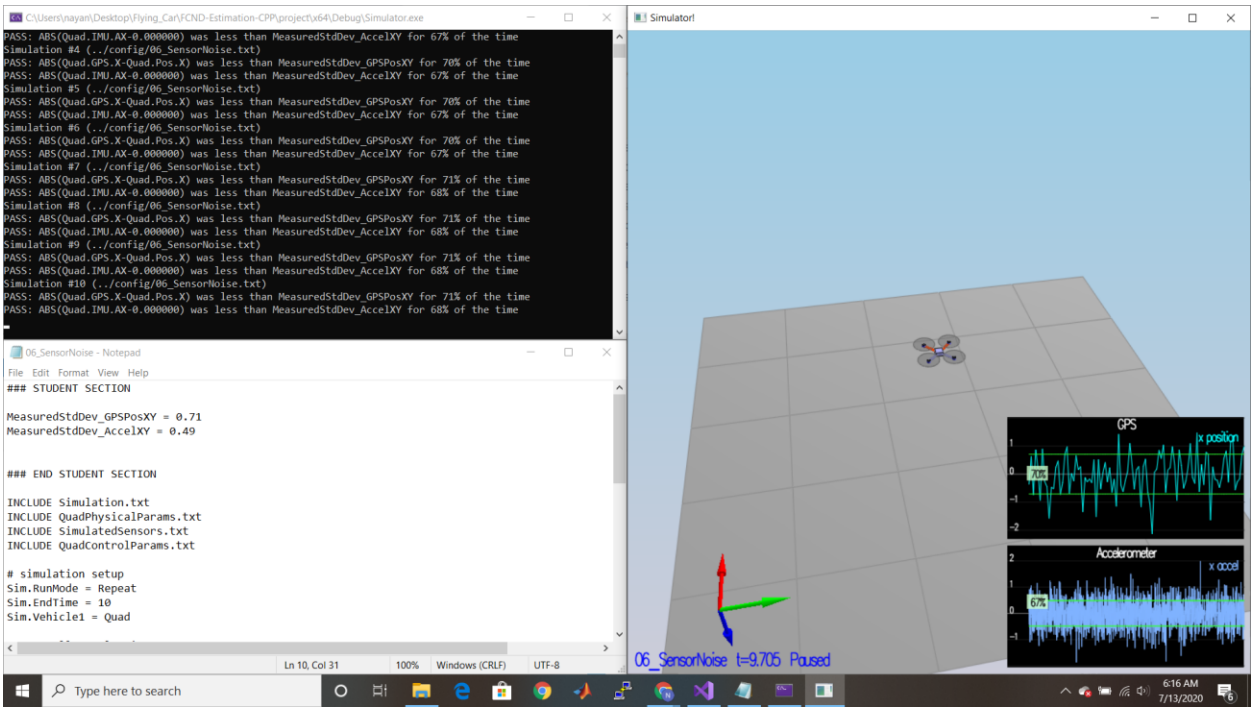
IMPLEMENTATION :

Scenario 6 – Estimate sensor noise

After running scenario 6 in the simulator to collect accelerometer and GPS data, the standard deviations of the measurements were calculated. The standard deviations obtained were

MeasuredStdDev_GPSPosXY = 0.709187828
MeasuredStdDev_AccelXY = 0.487435033

These are close enough the corresponding values in SimulatedSensors.txt. These values were updated in the config/6_Sensornoise.txt file.



Scenario 7 – Estimate attitude

The rate gyro attitude integration scheme was updated in the UpdateFromIMU() function as shown below. After implementing the update, the attitude errors went down to about 0.02.

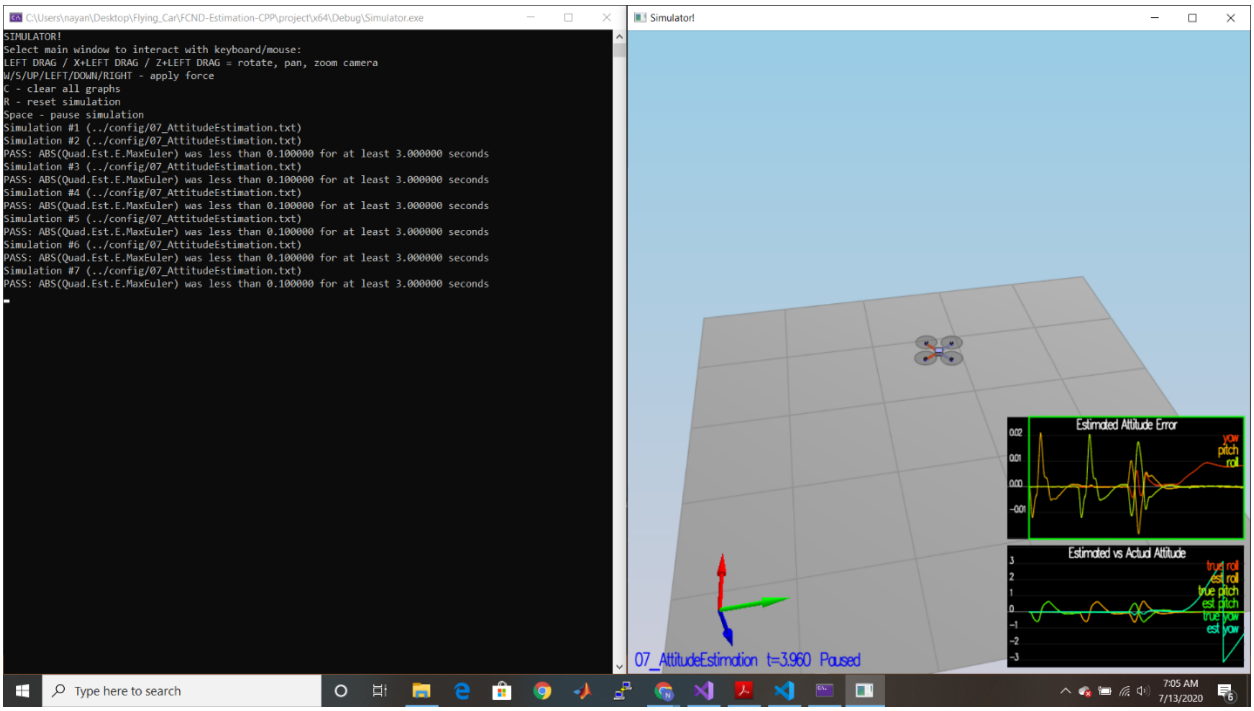
From section 7.1.2

$$\bar{q}_t = dq * q_t$$

$$\bar{\theta}_t = Pitch(\bar{q}_t)$$
$$\bar{\phi}_t = Roll(\bar{q}_t)$$

The updated yaw value is also stored in the state vector.

```
  Quaternion<float> qt = Quaternion<float>::FromEuler123_RPY(rollEst, pitchEst,
ekfState(6));
  qt = qt.IntegrateBodyRate_fast(lastGyro, dtIMU/2);
  float predictedPitch = qt.Pitch();
  float predictedRoll = qt.Roll();
  ekfState(6) = qt.Yaw();
```



Scenario 8,9 – Implement EKF prediction step

The PredictState() function was implemented using the following equations :

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

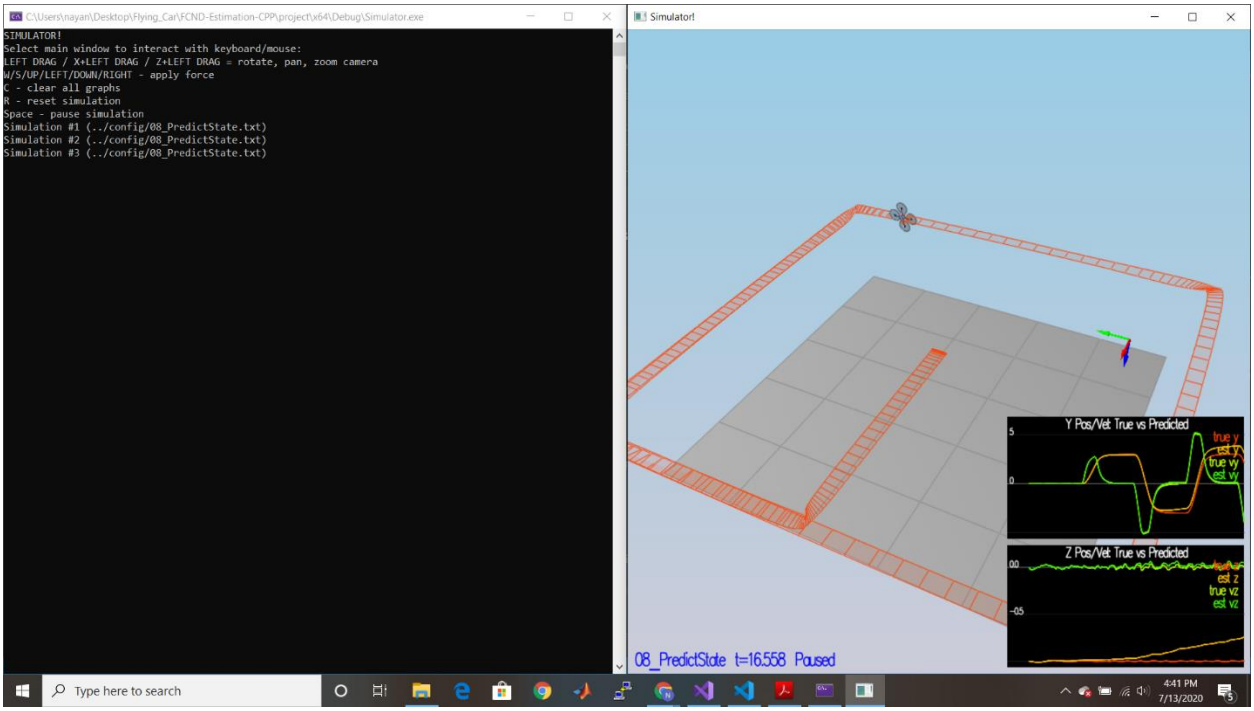
$$g(x_t, u_t, \Delta t) = \begin{bmatrix} x_{t,x} + x_{t,\dot{x}}\Delta t \\ x_{t,y} + x_{t,\dot{y}}\Delta t \\ x_{t,z} + x_{t,\dot{z}}\Delta t \\ x_{t,\dot{x}} \\ x_{t,\dot{y}} \\ x_{t,\dot{z}} - g\Delta t \\ x_{t,\psi} \end{bmatrix} + \begin{bmatrix} 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0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

The code is shown below

```
// EKF Algorithm Line 2

const auto accel_IF = attitude.Rotate_BtoI(accel);

predictedState(0) += predictedState(3) * dt;
predictedState(1) += predictedState(4) * dt;
```

```
predictedState(2) += predictedState(5) * dt;

predictedState(3) += accel_IF.x * dt;
predictedState(4) += accel_IF.y * dt;
predictedState(5) += (accel_IF.z - 9.81) * dt;
```

After implementing the PredictState() function, the estimated state tracks the actual state with only an acceptable level of drift with an ideal IMU.



GetRgbPrime() function is implemented using the equation

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

the code is shown below

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

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

RbgPrime(0, 2) = -cos(roll)*sin(pitch)*sin(yaw) + sin(roll)*cos(yaw);
RbgPrime(1, 2) =  cos(roll)*sin(pitch)*cos(yaw) + sin(roll)*sin(yaw);
```

The Predict() function is implemented using the equation

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

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

and the step 4 from the EKF algorithm is completed

Algorithm 2 E(KF) algorithm.

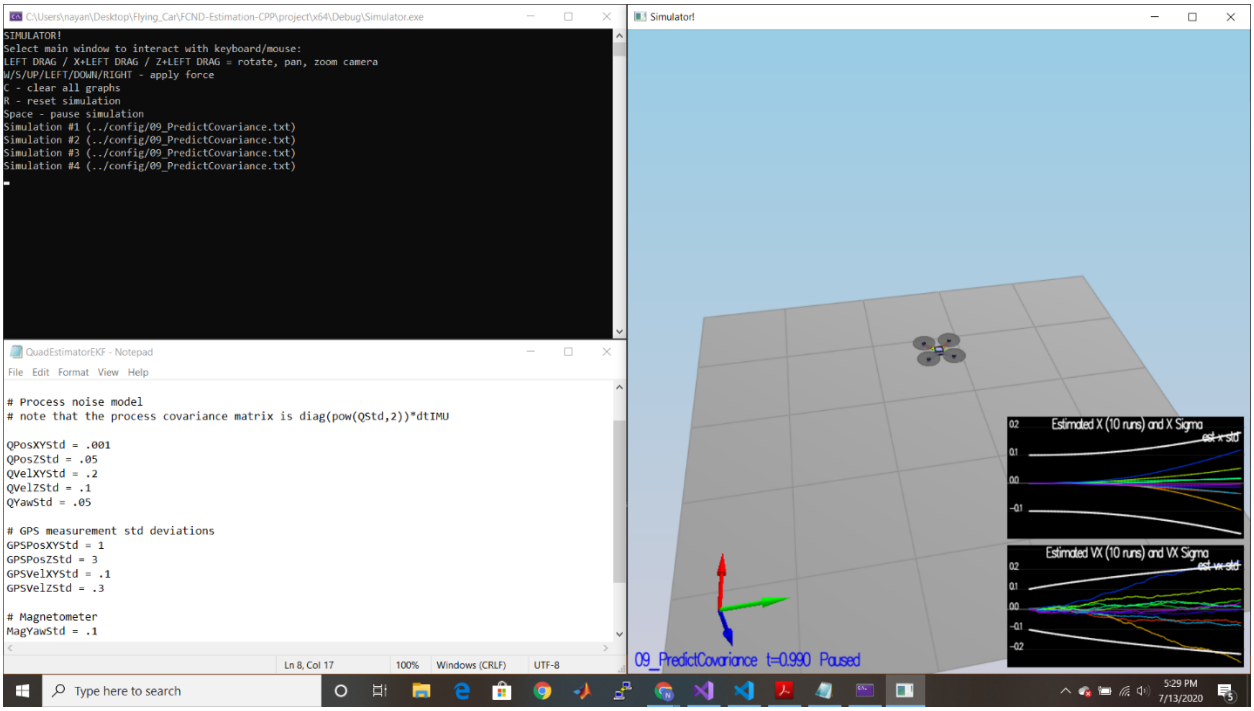
1: **function** PREDICT($\mu_{t-1}, \Sigma_{t-1}, u_t, \Delta t$)
2: $\bar{\mu}_t = g(u_t, \mu_{t-1})$
3: $G_t = g'(u_t, x_t, \Delta t)$
4: $\bar{\Sigma}_t = G_t \Sigma_{t-1} G_t^T + Q_t$
5: **return** $\bar{\mu}_t, \bar{\Sigma}_t$
6: **function** UPDATE($\bar{\mu}_t, \bar{\Sigma}_t, z_t$)
7: $H_t = h'(\bar{\mu}_t)$
8: $K_t = \bar{\Sigma}_t H_t^T (H_t \bar{\Sigma}_t H_t^T + R_t)^{-1}$
9: $\mu_t = \bar{\mu}_t + K_t(z_t - h(\bar{\mu}_t))$
10: $\Sigma_t = (I - K_t H_t)\bar{\Sigma}_t$
11: **return** μ_t, Σ_t
12: **function** EXTENDEDKALMANFILTER
13: $u_t = \text{COMPUTECONTROL}(\mu_{t-1}, \Sigma_{t-1})$
14: $\bar{\mu}_t, \bar{\Sigma}_t = \text{PREDICT}(\mu_{t-1}, \Sigma_{t-1}, u_t, \Delta t)$
15: $z_t = \text{READSENSOR}()$
16: $\mu_t, \Sigma_t = \text{UPDATE}(\bar{\mu}_t, \Sigma_t, z_t)$

The code is shown below

```
  // EKF Algorithm Line 3
  gPrime(0,3) = dt;
  gPrime(1,4) = dt;
  gPrime(2,5) = dt;
  gPrime(3,6) = (RbgPrime(0,0)*accel.x + RbgPrime(0,1)*accel.y + RbgPrime(0,2)*ac
cel.z) * dt;
  gPrime(4,6) = (RbgPrime(1,0)*accel.x + RbgPrime(1,1)*accel.y + RbgPrime(1,2)*ac
cel.z) * dt;
  gPrime(5,6) = (RbgPrime(2,0)*accel.x + RbgPrime(2,1)*accel.y + RbgPrime(2,2)*ac
cel.z) * dt;
```

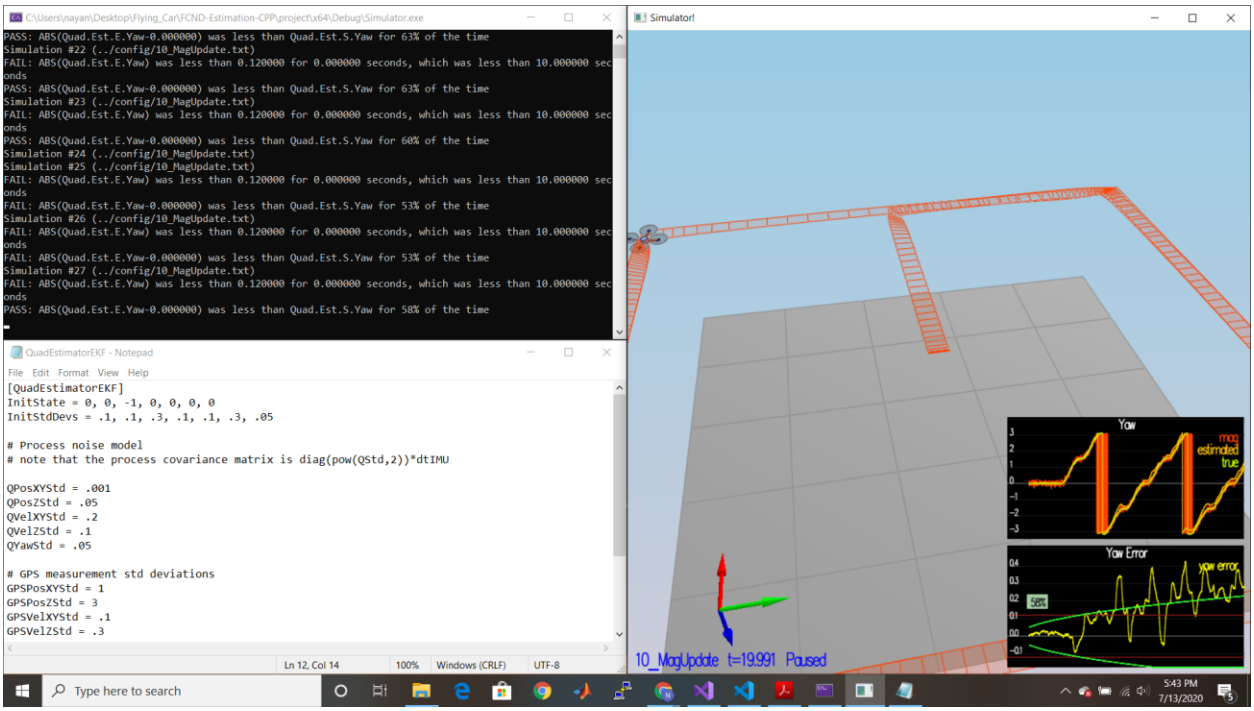
```
// EKF Algorithm Line 4
ekfCov = gPrime * ekfCov * gPrime.transpose() + Q;
```

Reasonable values for QPosXYStd and QVelXYStd are found by trial and error such that it models the error characteristics well in a short time interval. The values obtained were 0.001 and 0.2 respectively.



Scenario 10 – Implement magnetometer update

Running scenario 10_MagUpdate the yaw parameter QYawStd is set such that it models the error characteristics well. The value was 0.5. The second test case succeeds but the first one fails because magnetometer reading has not been incorporated yet.



The magnetometer reading is easily incorporated using the equations below with additional steps to ensure that the error is wrapped to stay between -pi and pi.

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

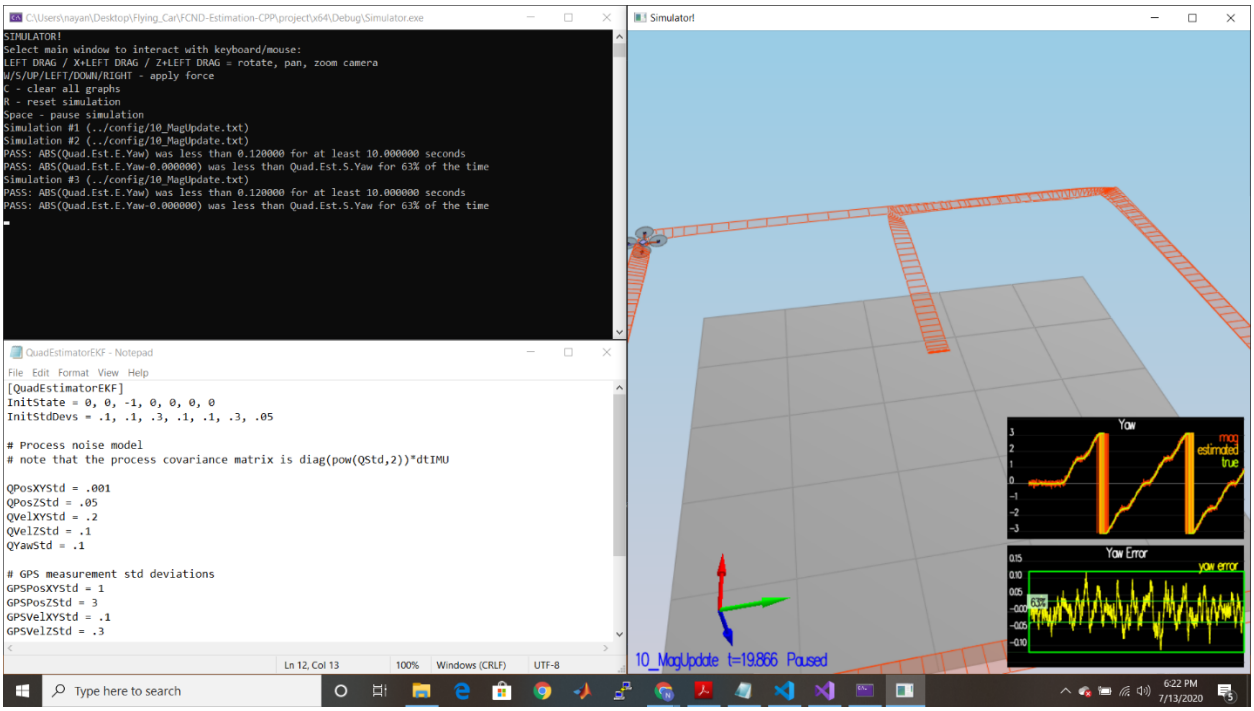
$$h(x_t) = \begin{bmatrix} x_{t,\psi} \end{bmatrix}$$

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

The code is given below

```
zFromX(0) = ekfState(6);
if ((zFromX(0) - z(0)) > M_PI) {
    zFromX(0) -= 2.f*M_PI;
}
else if ((zFromX(0) - z(0)) < -M_PI) {
    zFromX(0) += 2.f*M_PI;
}
hPrime(0,6) = 1;
```

The result of scenario 10_MagUpdate will now pass both test cases.



Scenario 11a – Implement GPS update

The scenario 11_GPSUpdate starts off with an ideal estimator and an ideal IMU. The scenario is investigated by disabling the ideal estimator first and then the ideal IMU. It can be observed that the error magnitude changes. After disabling both ideal estimator and ideal IMU, the GPS measurement standard deviations are updated such that the error characteristics are modeled well.

The UpdateFromGPS() function is easily implemented using the following equations

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

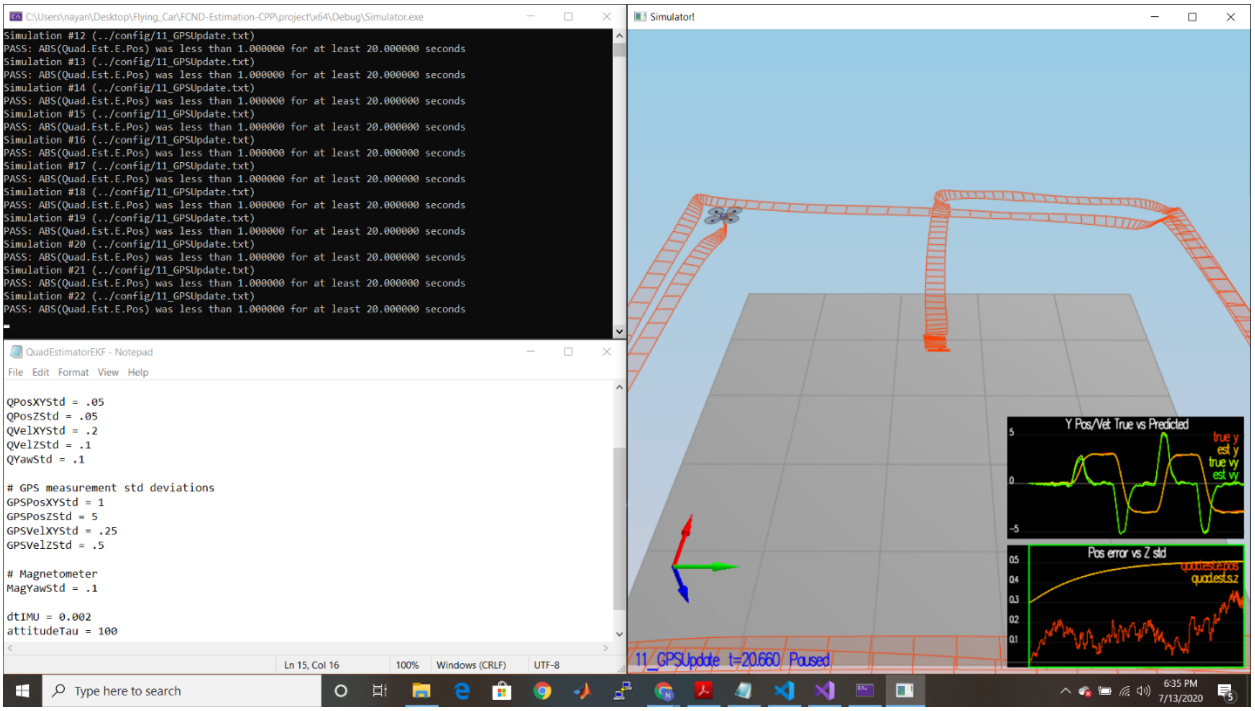
$$h(x_t) = \begin{bmatrix} x_{t,x} \\ x_{t,y} \\ x_{t,z} \\ x_{t,\dot{x}} \\ x_{t,\dot{y}} \\ x_{t,\dot{z}} \end{bmatrix}$$

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

The code is shown below

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

hPrime.setIdentity();
```

The scenario 11_GPSUpdate will now pass the test case

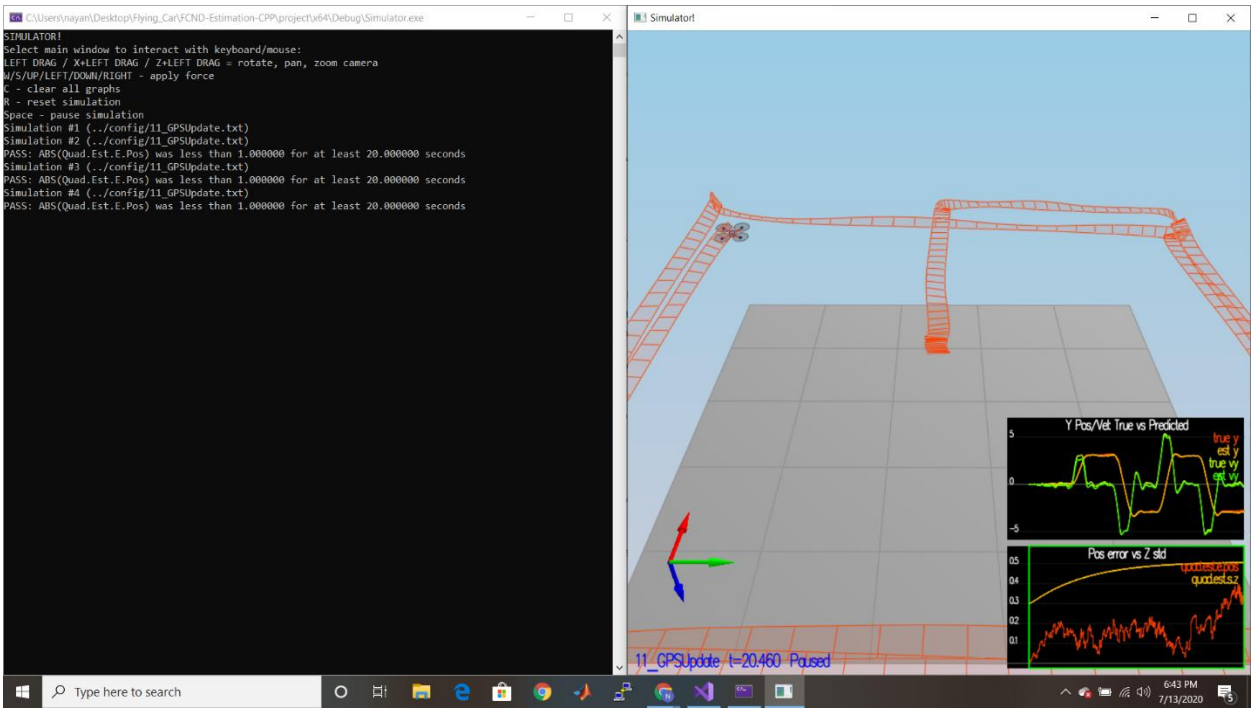


Scenario 11b – Replace controller and re-tune

The files QuadController.cpp and QuadControlParams.txt were replaced with those from the previous project. Fortunately, the controller parameters did not need a lot of fine tuning. However,

different parameters were tested to understand the effect on the performance of the quadcopter. It was observed that the quadcopter is now significantly more sensitive to changes in the controller gains. It was also observed that the performance of the altitude controller had deteriorated.

The error does stay below the acceptable limit of 1m.



Link to video of simulation : https://youtu.be/dF6Bja0v6h0