Building an Estimator

Minh Gia Hoang

Step 1: Sensor Noise

The standard deviation (SD) of a random variable X taking values $x_1, x_2, \ldots, x_N$ is

$$\sigma_X = \sqrt{\frac{1}{N}\sum_{i=1}^{N}(x_i - \mu_X)^2}, \tag{1}$$

where

$$\mu_X = \frac{1}{N}\sum_{i=1}^{N} x_i.$$

Applying (1), $MeasuredStdDev\_GPSPosXY = 0.711$ and $MeasuredStdDev\_AccelXY = 0.488$. This task is done in Jupyter notebook `06_SensorNoise.ipynb`. These results are matched the settings in `SimulatedSensors.txt` i.e., $PosStd = 0.7$ and $AccelStd = 0.5$.

Figure 1 and Figure 2 show that the SDs capture approx. 68% of the respective measurements.

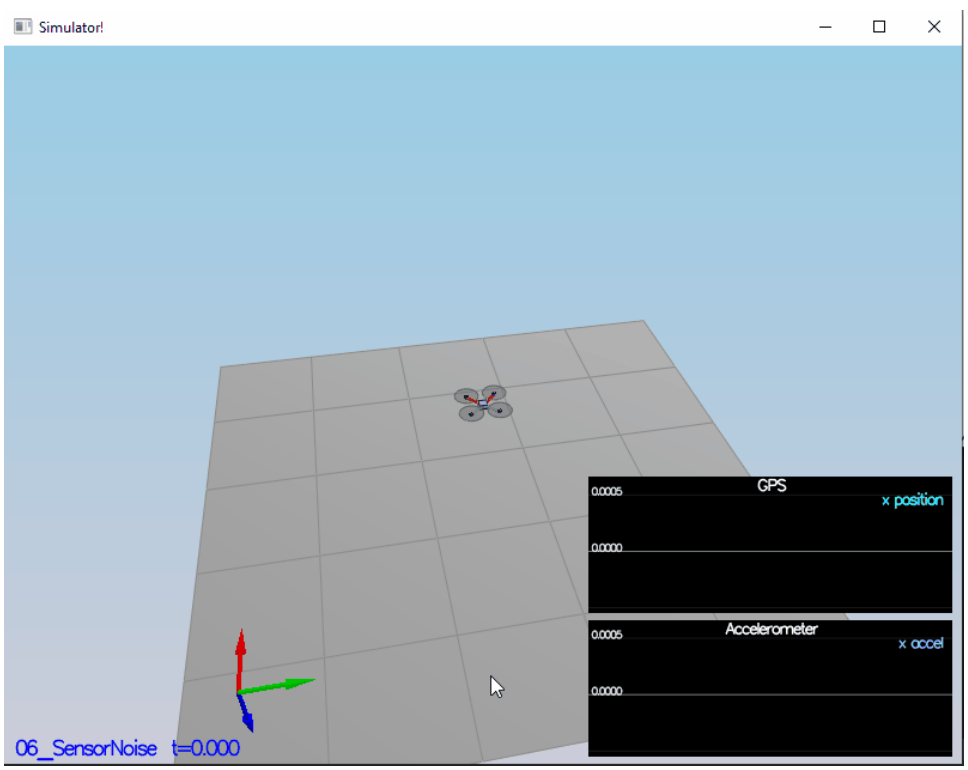

Figure 1: Result of estimated sensor noise standard deviation.

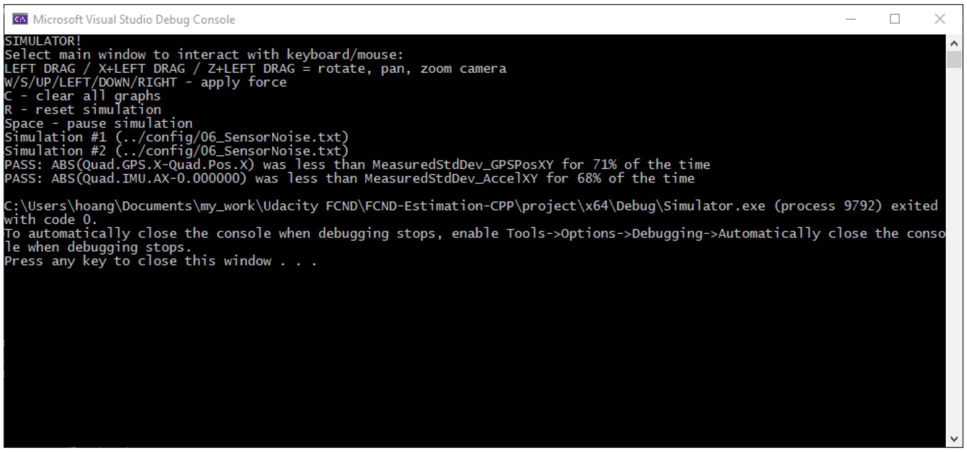

Figure 2: PASS message for estimated sensor noise standard deviation.

Step 2: Attitude Estimation

Estimated roll $\hat{\phi}_t$ and pitch $\hat{\theta}_t$ are estimated by complementary filters as

$$\hat{\phi}_t = \frac{\tau}{\tau + T_s}\left(\hat{\phi}_{t-1} + T_s z_{t,\dot{\phi}}\right) + \frac{T_s}{\tau + T_s} z_{t,\phi},$$
$$\hat{\theta}_t = \frac{\tau}{\tau + T_s}\left(\hat{\theta}_{t-1} + T_s z_{t,\dot{\theta}}\right) + \frac{T_s}{\tau + T_s} z_{t,\theta},$$

where τ is a time constant, T_s the filter sampling period, $z_{t,\dot{\phi}}$ and $z_{t,\dot{\theta}}$ the gyro measurement in x and y, respectively, $z_{t,\phi}$ and $z_{t,\theta}$ the estimated roll and pitch from accelerometer leveling.

The above linear complementary filter simply integrates the gyro measurements to get the angle estimates. This can be improved by nonlinear complement filter when the angles are updated using quaternion as follows

$$q_t = dq * q_{t-1},$$

where q_t is the predicted quaternion of q_{t-1}, dq the quaternion consisting of the measurement of the angular rates from the IMU in the body frame.

Thus the nonlinear complementary filter is

$$\hat{\phi}_t = \frac{\tau}{\tau + T_s}\bar{\phi}_t + \frac{T_s}{\tau + T_s} z_{t,\phi},$$
$$\hat{\theta}_t = \frac{\tau}{\tau + T_s}\bar{\theta}_t + \frac{T_s}{\tau + T_s} z_{t,\theta},$$

where $\bar{\phi}_t = Roll(q_t)$ and $\bar{\theta}_t = Pitch(q_t)$.

The nonlinear filter is implemented in lines 101 – 110 in `QuadEstimatorEKF.cpp` as follows.

```
Quaternion<float> attitude = Quaternion<float>::FromEuler123_RPY(rollEst, pitchEst,
ekfState(6));
Quaternion<float> predictedAttitude = attitude.IntegrateBodyRate(V3D(gyro.x, gyro.y,
gyro.z), dtIMU);
```

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

Figure 3 and Figure 4 show the result of implemented nonlinear complementary filter which can estimate within 0.1 rad each of the Euler angles.

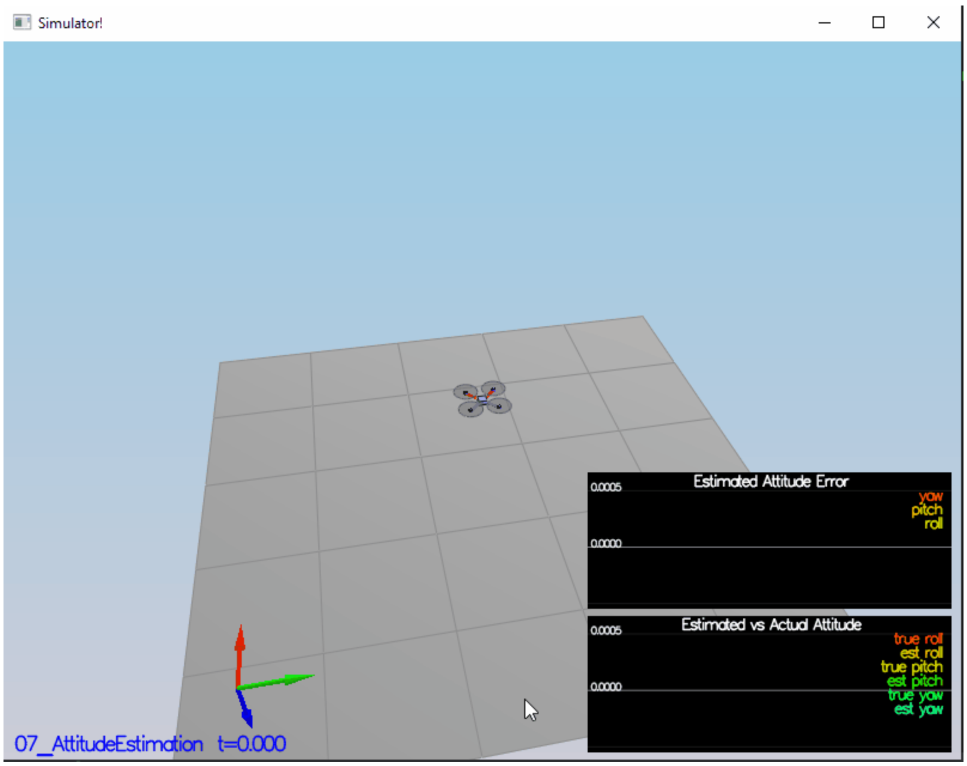

Figure 3: Result of nonlinear complementary filters.

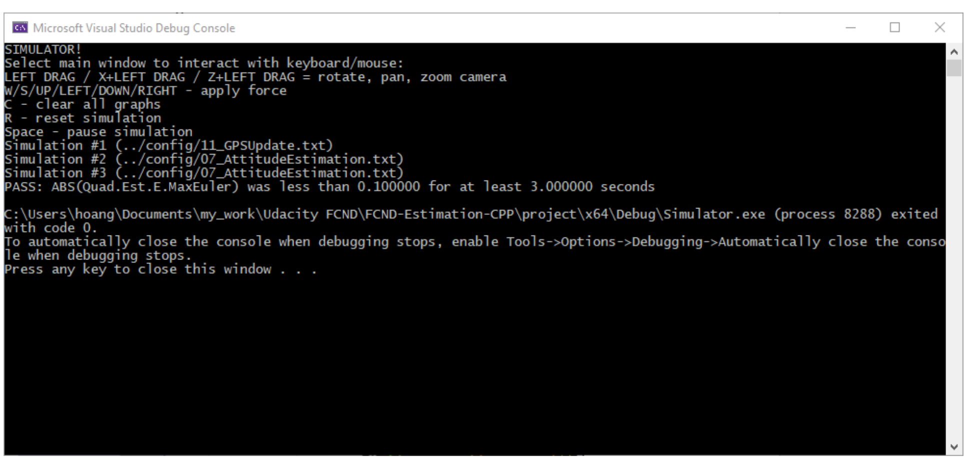

Figure 4: PASS message of nonlinear complementary filters.

Step 3: Prediction Step

In this part, we implement the prediction step following instruction in (Tellex, Brown, & Lupashin). This step comprises calculation of a transition model and covariance update.

The transition model is

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

where $x_t = (x, y, z, \dot{x}, \dot{y}, \dot{z}, \psi)^T$ is the state vector, ψ the vehicle's yaw, u_t the control input or the acceleration (not including gravity g) in the body frame, $\boldsymbol{R}_{bg}$ the rotation matrix which rotates from the body frame to the global frame, and ΔT the sampling period. The transition function is implemented in lines 173 – 181 in `QuadEstimatorEKF.cpp` as follows.

```
// compute acceleration in inertial frame
V3F accelInertial = attitude.Rotate_BtoI(accel) + V3F(0.0f, 0.0f, -9.81f);

predictedState[0] = curState[0] + curState[3] * dt;
predictedState[1] = curState[1] + curState[4] * dt;
predictedState[2] = curState[2] + curState[5] * dt;
predictedState[3] = curState[3] + accelInertial[0] * dt;
predictedState[4] = curState[4] + accelInertial[1] * dt;
predictedState[5] = curState[5] + accelInertial[2] * dt;
predictedState[6] = curState[6];
```

Figure 5 shows that the estimator state can track the actual state in `08_PredictState` scenario, with only reasonably slow drift. Note that this scenario has ideal IMU thus the slow drift is caused by signal integration in transition function.

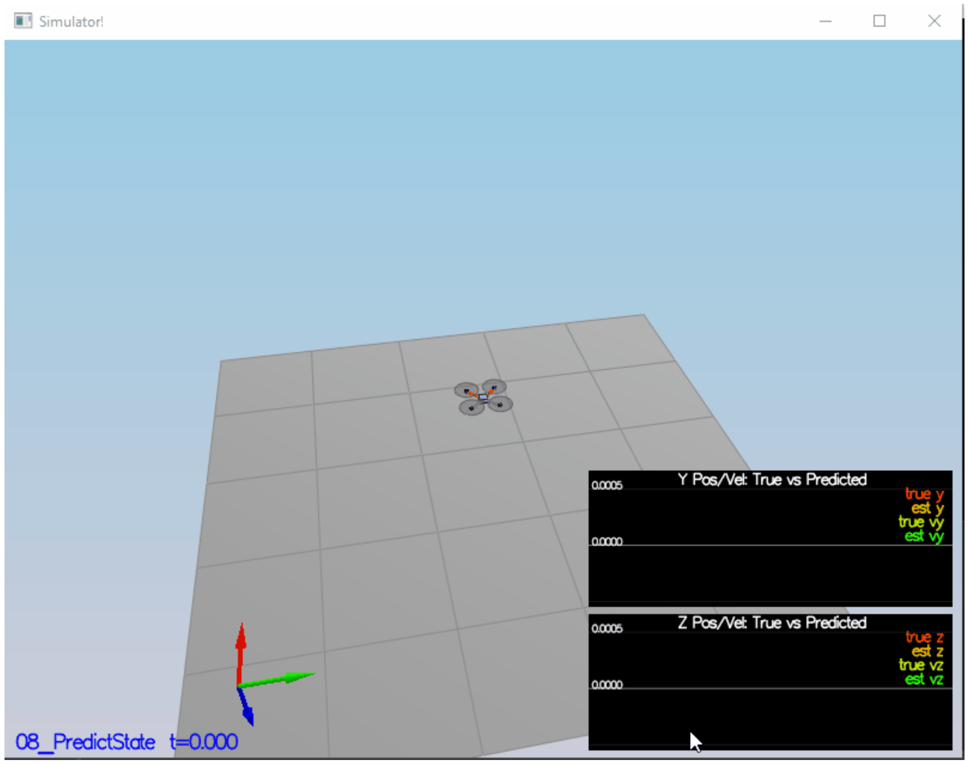

Figure 5: Result of state transition model.

On the other hand, the covariance update follows the classic EKF update equation

$$\bar{\Sigma}_{\mathrm{t}} = g'(x_t, u_t, \Delta T)\Sigma_{t-1} g'(x_t, u_t, \Delta T)^T + Q_t,$$

where Σ_{t-1} is the covariance at $t-1$, $\bar{\Sigma}_{\mathrm{t}}$ the updated covariance at t (before correction step), $g'(x_t, u_t, \Delta T)$ the Jacobian matrix, and Q_t the process noise covariance. The Jacobian of the transition model is

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

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

where $\boldsymbol{R}'_{bg}$ is the partial derivative of the $\boldsymbol{R}_{bg}$ with respect to yaw ψ as follows

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

The $\boldsymbol{R'}_{bg}$ is implemented in lines 208 – 221 in `QuadEstimatorEKF.cpp` as follows.

```
Float cPhi = cos(roll);
float sPhi = sin(roll);
float cTheta = cos(pitch);
float sTheta = sin(pitch);
float cPsi = cos(yaw);
float sPsi = sin(yaw);

RbgPrime(0, 0) = -cTheta * sPsi;
RbgPrime(0, 1) = -sPhi * sTheta * sPsi - cPhi * cPsi;
RbgPrime(0, 2) = -cPhi * sTheta * sPsi + sPhi * cPsi;

RbgPrime(1, 0) = cTheta * cPsi;
RbgPrime(1, 1) = sPhi * sTheta * cPsi - cPhi * sPsi;
RbgPrime(1, 2) = cPhi * sTheta * cPsi + sPhi * sPsi;
```

Then the Jacobian is implemented in lines 268 – 276 in `QuadEstimatorEKF.cpp` as follows. The covariance update is also included.

```
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) *
accel.z) * dt;
gPrime(4, 6) = (RbgPrime(1, 0) * accel.x + RbgPrime(1, 1) * accel.y + RbgPrime(1, 2) *
accel.z) * dt;
gPrime(5, 6) = (RbgPrime(2, 0) * accel.x + RbgPrime(2, 1) * accel.y + RbgPrime(2, 2) *
accel.z) * dt;

// update covariance
ekfCov = gPrime * ekfCov * gPrime.transpose() + Q;
```

Tuning process noise covariance matrix Q is critical in EKF.

$$Q = diag(QPosXYStd^2, QPosXYStd^2, QPosZStd^2, QVelXYStd^2, QVelZStd^2, QYawStd^2)\Delta T,$$

where $QPosXYStd$, $QPosZStd$, $QVelXYStd$, $QVelZStd$, and $QYawStd$ are the power spectrum densities (PSDs) of, respectively, the position random noise in X and Y, the position random noise in Z, the velocity random noise in X and Y, the velocity random noise in Z, and the yaw random noise.

We tune $QPosXYStd = 0$ and $QVelXYStd = 0.2$. Generally, position random noise is very small. We integrate velocity to get the position thus there is no other noise source i.e., $QPosXYStd \approx 0$ and $QPosZStd \approx 0$. One the other hand, the imperfect control input (i.e., acceleration) is a noise source when calculate the predicted velocity in the transition function, thus $QVelXYStd$ and $QVelZStd$ must be fine-tuned to account for this effect.

Figure 6 and **Error! Reference source not found.**shows the result of the covariance prediction after tunning. The predicted covariance can capture the magnitude of the error for the short period of time.

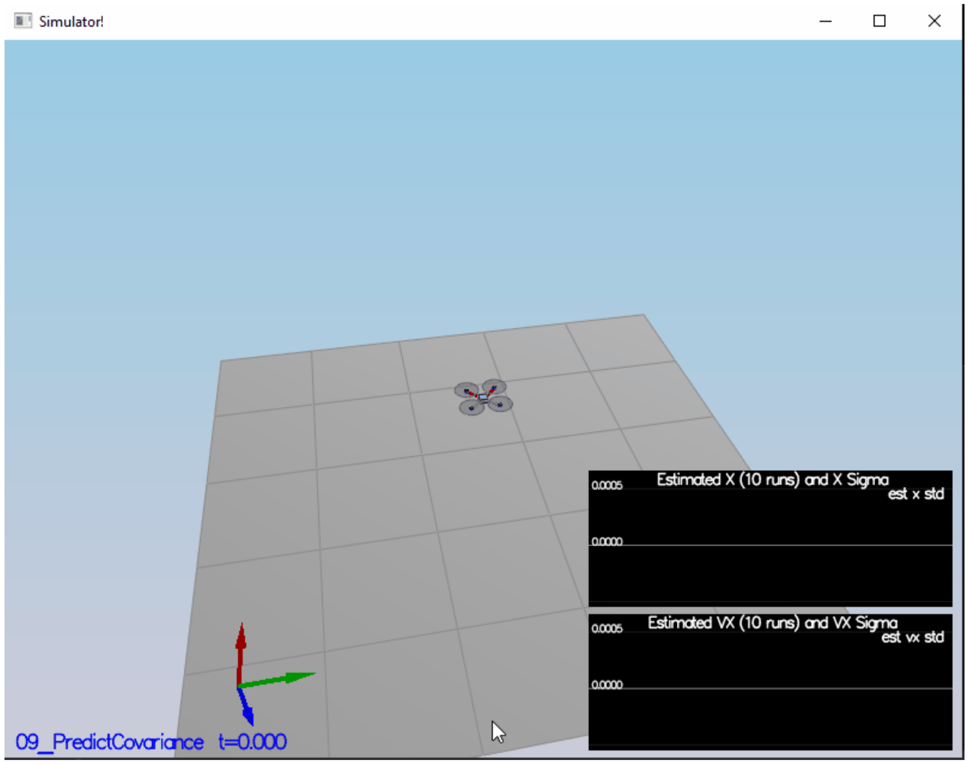

Figure 6: Result of predicted covariance.

Step 4: Magnetometer Update

The magnetometer gives the yaw measurement in the global frame i.e.,

$$z_t = [\Psi],$$

and the corresponding measurement model is

$$h(x_t) = [x_{t,\Psi}].$$

As the measurement model is linear, its Jacobian is a matrix of zeros and ones as follows.

$$h'(x_t) = [0 \quad 0 \quad 0 \quad 0 \quad 0 \quad 0 \quad 1].$$

The magnetometer update is implemented in lines 331 – 337 in `QuadEstimatorEKF.cpp`.

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

zFromX(0) = ekfState(6);

float innovation = z(0) - zFromX(0);
if (innovation > F_PI) z(0) -= 2.f*F_PI;
if (innovation < -F_PI) z(0) += 2.f*F_PI;
```

We tune $QYawStd = 0.1$. Figure 7 shows the estimated standard deviation accurately captures the error and maintain an error of less than 0.1 rad in heading.

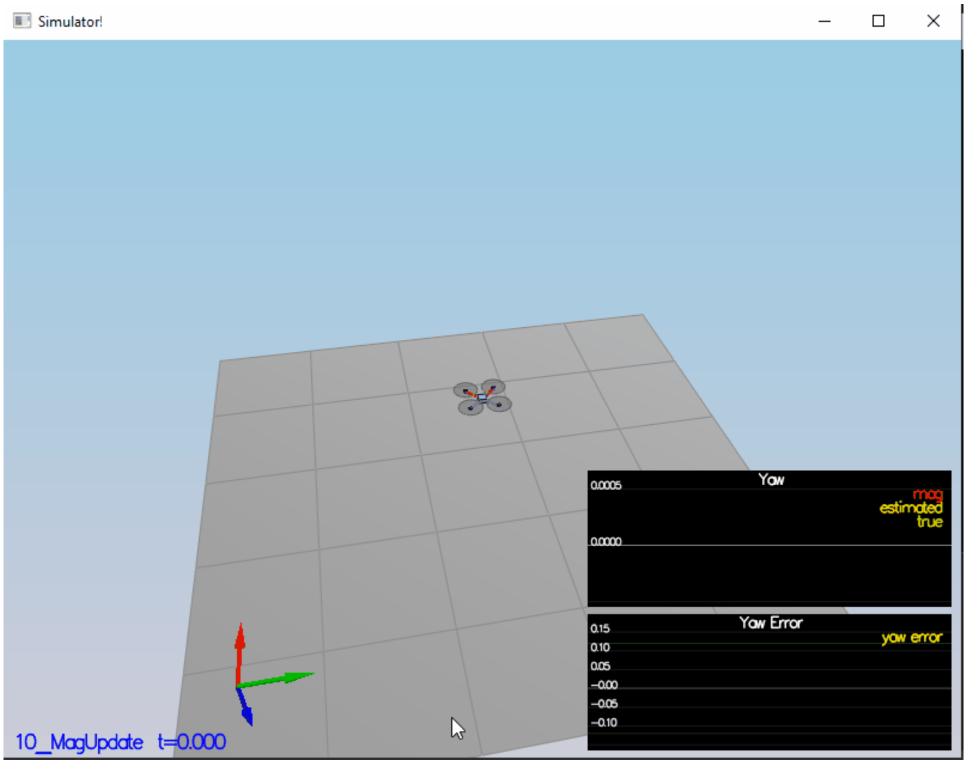

Figure 7: Result of magnetometer update.

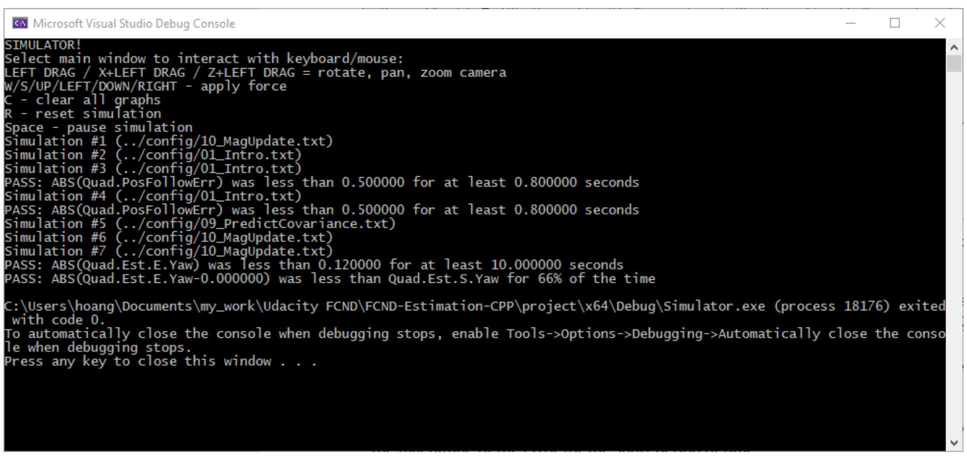

Figure 8: PASS message of magnetometer update.

Step 5: Closed Loop + GPS Update

In this session, we continue updating the predicted state using position and velocity from the GPS receiver. The GPS measurement is given as

$$z_t = [x \quad y \quad z \quad \dot{x} \quad \dot{y} \quad \dot{z}]^T,$$

and the measurement model is

$$h(x_t) = [x_{t,x} \quad x_{t,y} \quad x_{t,z} \quad x_{t,\dot{x}} \quad x_{t,\dot{y}} \quad x_{t,\dot{z}}]^T.$$

Then the Jacobian is

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

The GPS update is implemented in lines 302 – 308 in `QuadEstimatorEKF.cpp`.

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

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

We tune

- $GPSPosXYStd = 0.7$ (based on the sensor noise standard deviation in Step 1: Sensor Noise),
- $GPSPosZStd = 2.0$ (GPS vertical accuracy is usually worse than horizontal one),
- $GPSVelXYStd = 0.7$,
- $GPSVelZStd = 0.3$.

We also retune the process noise

- $QPosZStd = 0$,
- $QVelZStd = 0.05$.

Figure 9 and Figure 10 show the result of GPS update to complete the closed-loop EKF. The quadrotor is able to complete the entire simulation cycle with estimated position error of < 1 m.

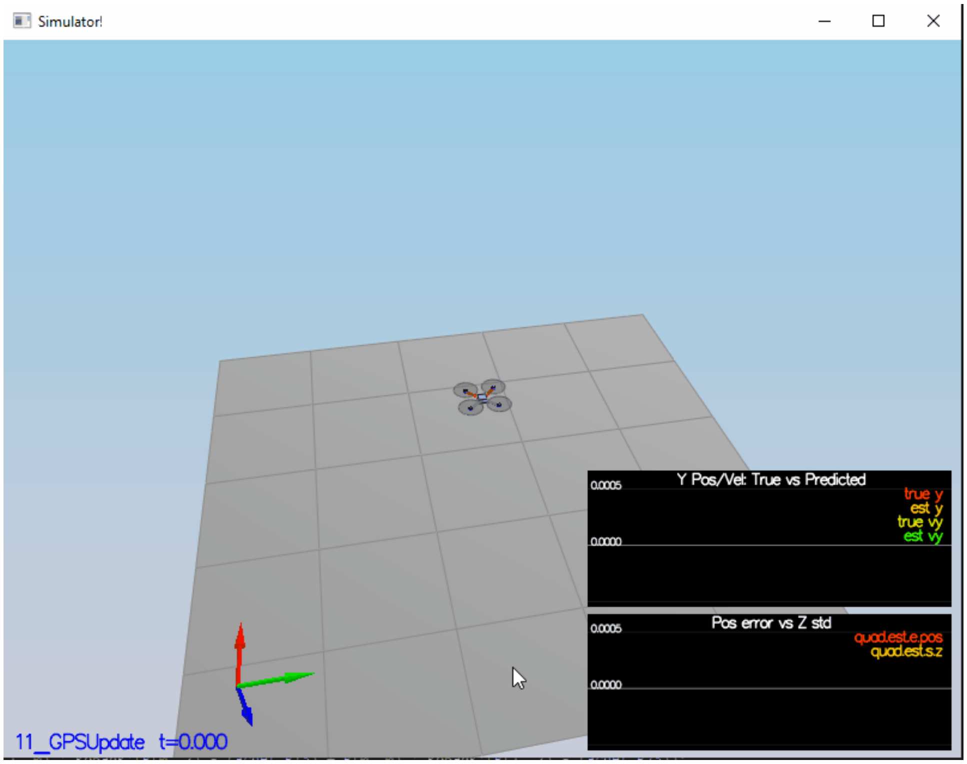

Figure 9: Result of GPS update.

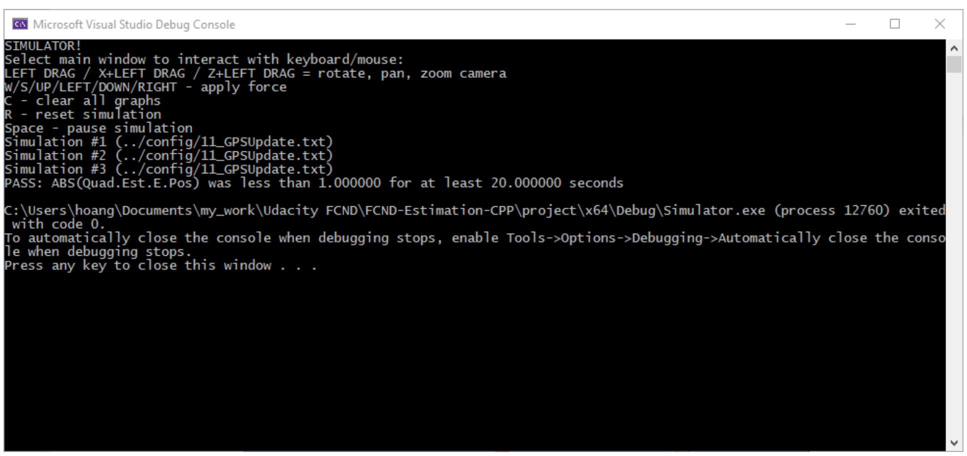

Figure 10: PASS message of GPS update.

Step 6: Adding Your Controller

Now, we replace the provided controller with our controller we built in the previous project (Hoang, 2021) in `QuadController.cpp`. In order to make it worked, we de-tune the controller to stabilize it as in Table 1.

Table 1: Result of de-tuned controller.

(Hoang, 2021)	**Detuned**

# Position control gains 	
kpPosXY = 3
	# Position control gains
	kpPosXY = 2.2

kpPosZ = 25
KiPosZ = 42
 	
# Velocity control gains 	
kpVelXY = 13
kpVelZ = 14
 	
# Angle control gains 	
kpBank = 15
kpYaw = 3.5
 	
# Angle rate gains 	
kpPQR = 90, 92, 20
	kpPosZ = 25
	KiPosZ = 30
	
	# Velocity control gains
	kpVelXY = 12
	kpVelZ = 5
	
	# Angle control gains
	kpBank = 11
	kpYaw = 3.5
	
	# Angle rate gains
	kpPQR = 85, 82, 15

Figure 11 and Figure 12 show the results of combining the estimator with the de-tuned controller. The vehicle is to once again complete the entire simulation cycle with an estimated position error of < 1 m.

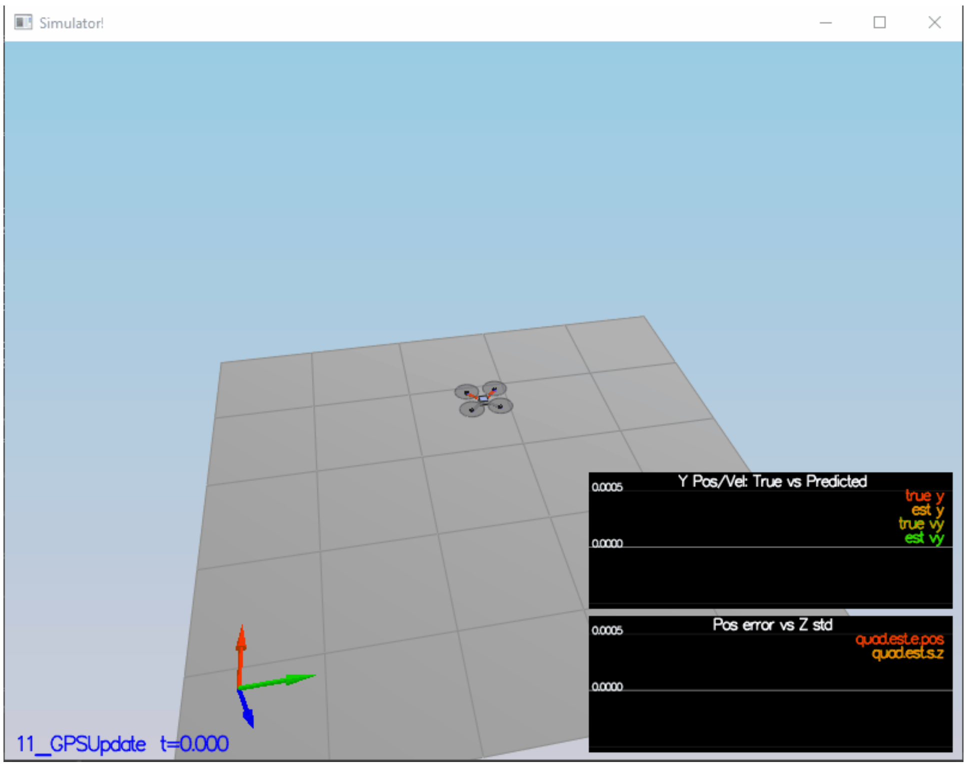

Figure 11: Result of combining estimator with de-tuned controller.

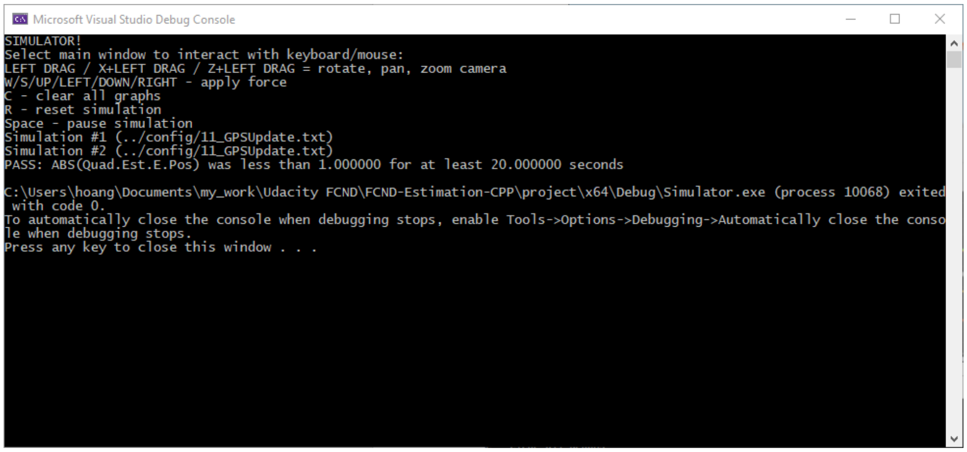

Figure 12: PASS message of combining estimator with de-tuned controller.

References

Hoang, G. M. (2021). *Building a Controller.* Retrieved from https://github.com/giaminhhoang/FCND-Controls-CPP

Tellex, S., Brown, A., & Lupashin, S. (n.d.). *Estimation for Quadrotors.* Udacity.