# FCND Estimation – Udacity Flying Car

Tiansi Dong

University of Bonn

## 1 Sensor Noise

For this task, I need to calculate `MeasuredStdDev_GPSPosXY` and `MeasuredStdDev_AccelXY` using the sensor data in the file `SimulatedSensors.txt`. I used `numpy` function `numpy.std` to compute. The code is push to `https://github.com/HimmelStein/FCND-EXe/blob/master/estimation.py`. The result is correct, as shown in Fig 1.

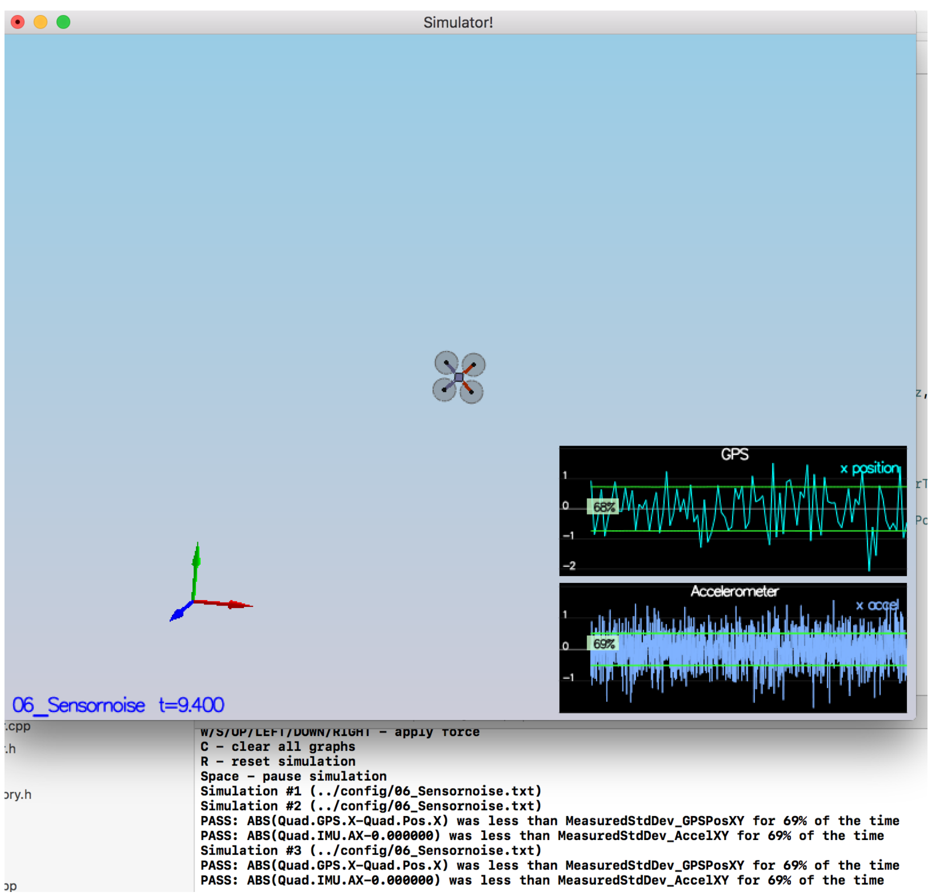


**Fig. 1.** Sensor noises

## 2 Attitude Estimation

In this task, I need to implement a better rate gyro attitude integration scheme in the `UpdateFromIMU()` function.

I first compute the rotation rate of roll, pitch, yaw (body rates in body frame $[rad/s]$) from the gyroscope.

```
float p = gyro.x, q = gyro.y, r= gyro.z
```

Then, I used the `FromEuler123_RPY` function from the `Quaternion<float>` class to transform the rotation during the interval `dtIMU` into the inertia reference frame.

```
Quaternion<float> d_q;
d_q = Quaternion<float>::FromEuler123_RPY(p*dtIMU,
                                          q*dtIMU,
                                          r*dtIMU);
```

It is also necessary to transform current attitude into the inertia reference frame.

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

The predicted quaternion can be computed as follows.

```
Quaternion<float>  q_bar = d_q*q_t;
```

The predicted Roll, Pitch, and Yaw can be computed using the `Roll()`, `Pitch`, and `Yaw()` in the `Quaternion<float>` class.

```
predictedRoll = q_bar.Roll();
predictedPitch = q_bar.Pitch();
ekfState(6) = q_bar.Yaw();
```

The result is illustrated in Fig 2.

## 3 Prediction Step

Using the values of predicted roll, pitch, and yaw, I need to predict the state of the drone in this task. Concretely, I need implement `PredictState()` function, a partial derivative of rotation matrix from body frame into the global frame `Rbg` prime matrix, and update the covariance. The result is illustrated in Fig 3.

### 3.1 Predict state

The current state is a six-tuple $(x, y, z, \dot{x}, \dot{y}, \dot{z})$. I need to predict the state `predictedState` after `dt` temporal duration. `predictedState` is initialised by the current state. After `dt` temporal duration, new positions can be directly computed as follows.

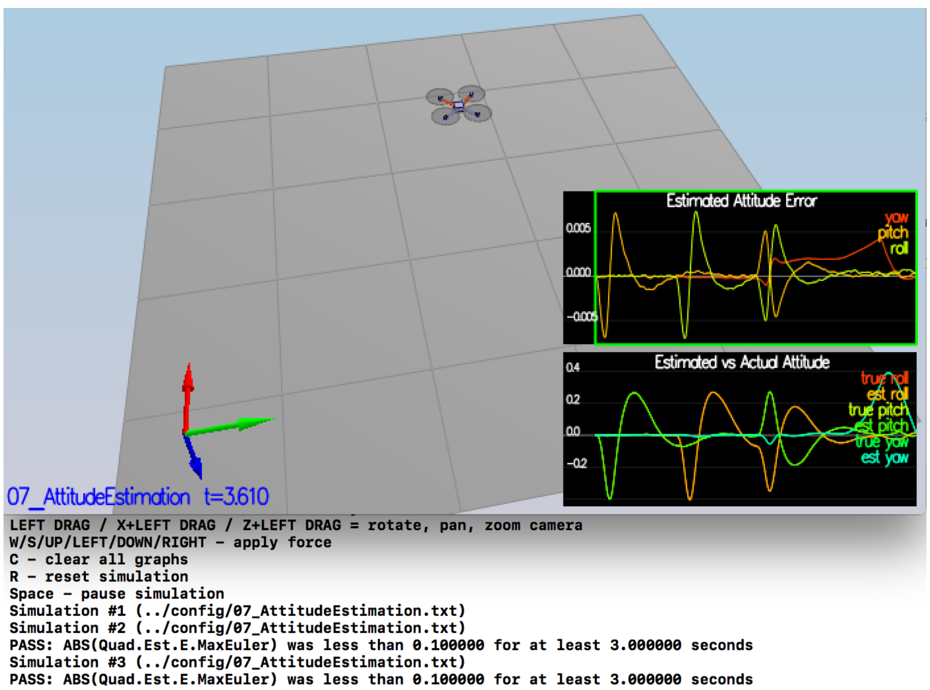


**Fig. 2.** Attitude Estimation

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

New velocities can be predicted, if accelerations in the $x, y, z$ directions in the global framework are known. However, the accelerations provided by the accelerator is in the body framework. So, I need to transform it into the global framework. First, I transform current attitude into the quaternion representation.

```
Quaternion<float> atn = attitude.Normalise();
```

then, use `attitude.Rotate_BtoI(<V3F>)` to rotate the acceleration vector from body frame to the global frame as follows.

```
V3F in = V3F(accel.x, accel.y, accel.z);
V3F vxyz = atn.Rotate_BtoI(in);
```

The velocity can be predicted as follows.

```
predictedState[3] += vxyz[0]*dt;
predictedState[4] += vxyz[1]*dt;
predictedState[5] += vxyz[2]*dt;
```

As the acceleration in the $z$ direction does not consider the gravity of the earth, it shall be further updated by subtracting `CONST_GRAVITY*dt`. That is,

```
predictedState[5] += vxyz[2]*dt - CONST_GRAVITY*dt;
```

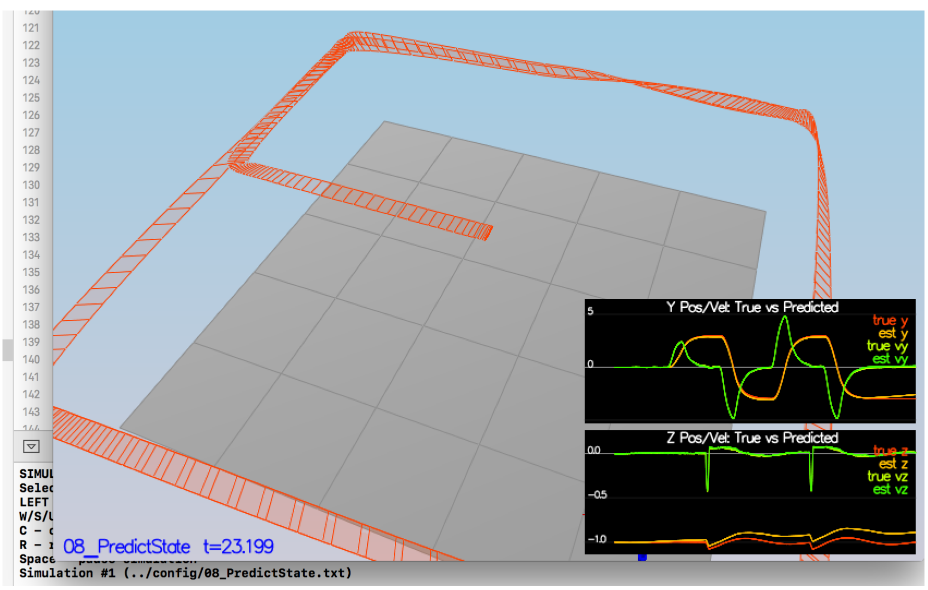


**Fig. 3.** Prediction Step

### 3.2 Partial derivative of the rotation matrix

The partial derivative of the rotation matrix is calculated by strictly following the formula in the document *Estimation for Quadrotors* by Stefanie Tellex, Andy Brown, and Sergei Lupashin.

```
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(pitch)*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(pitch)*sin(yaw);
```

### 3.3 Update covariance

The covariance is updated by following the formula.

$$\bar{\Sigma}_t = G_t \Sigma_{t-1} G_t^\intercal + Q_t \tag{1}$$

$G_t$ (`gPrime`) can be computed using `RbgPrime`, $Q$ is given. The `ekfCov` is predicted as follows.

```
gPrime(0,3) = dt;
gPrime(1,4) = dt;
gPrime(2,5) = dt;
gPrime(3,6) = RbgPrime(0,0)*accel.x*dt
              + RbgPrime(0,1)*accel.y*dt
              + RbgPrime(0,2)*accel.z*dt;
gPrime(4,6) = RbgPrime(1,0)*accel.x*dt
              + RbgPrime(1,1)*accel.y*dt
              + RbgPrime(1,2)*accel.z*dt;
gPrime(5,6) = RbgPrime(2,0)*accel.x*dt
              + RbgPrime(2,1)*accel.y*dt
              + RbgPrime(2,2)*accel.z*dt;

MatrixXf newCov = gPrime * ekfCov;
gPrime.transposeInPlace();
newCov *= gPrime;
ekfCov = newCov + Q ;
```

## 4 Magnetometer Update

Magnetomer is a sensor of yaw in the global frame, which only has one parameter, and can be easily updated as follows.

$$
\begin{aligned}
z_t &= [\psi] \\
h(x_t) &= [x_{t,\phi}] \\
h'(x_t) &= [0, 0, 0, 0, 0, 0, 1]
\end{aligned}
$$

The result is illustrated in Fig 4

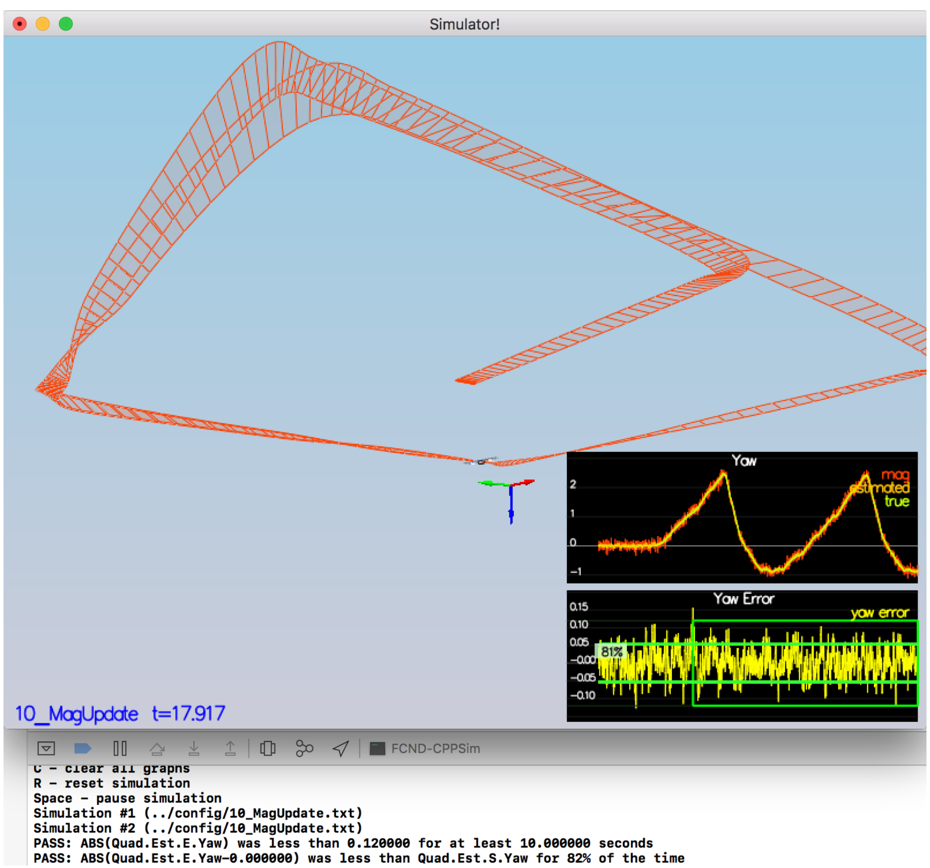


**Fig. 4.** Magnetometer update

## 5 GPS Update and Adding Controller

GPS is simply updated by setting $z_t, h(x_t)$, and $h'(x_t)$ as follows.

$$
\begin{aligned}
z_t &= [x, y, z, \dot{x}, \dot{y}, \dot{z}]^\intercal \\
h(x_t) &= [x_{t,x}, x_{t,y}, x_{t,z}, x_{t,\dot{x}}, y_{t,\dot{y}}, z_{t,\dot{z}}]^\intercal \\
h'(x_t) &= I_6, [0, 0, 0, 0, 0, 0, 0]^\intercal
\end{aligned}
$$

Parameters are set as follows. The results is shown in Fig 5.

```
QPosXYStd = .22
QPosZStd = .08
QVelXYStd = .11
QVelZStd = 0.15
QYawStd = 0.3
GPSPosXYStd = 0.2
GPSPosZStd = 3.5
GPSVelXYStd = 0.2
GPSVelZStd = 0.3
```

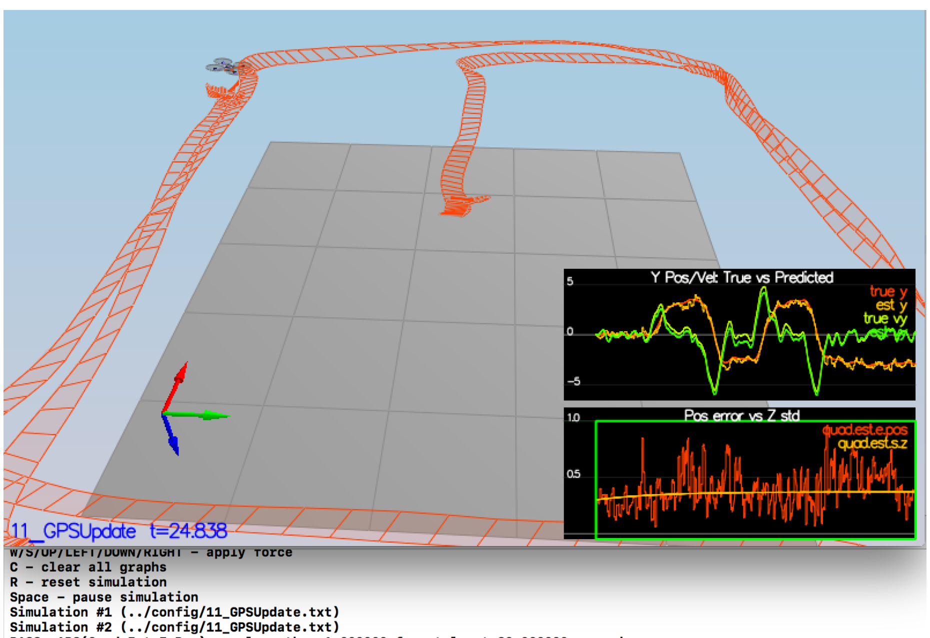


**Fig. 5.** GPS update