Udacity Flying Car Nanodegree 4th Project
Build an Estimator in C++

Xinjie Qiu
qiuxinjie@gmail.com

May 27, 2018

1 Introduction

The purpose of this project is to develope and implement Extended Kalman Filter used by a controller to estimate the state of a flight vehicle from realistic sensors in the C++ simulator.

The math that is useful to implement EKF is provided by Udacity in the Estimation for Quadrotors document.

2 Project Structure

- The EKF to implement is in QuadEstimatorEKF.cpp in the src directory.
- Parameters for tuning the EKF are in the parameter file QuadEstimatorEKF.txt in the config directory.
- Re-tuned controller QuadController.cpp and QuadControlParams.txt to work successfully with the implemented estimator.

3 Implement Estimator

3.1 Sensor Noise

In real drone, sensor measurement is never perfect. There must be some noise associated with the sensor readings. In this step, some simulated noisy sensor data were collected from a static quad to estimate the standard deviation of the quad's sensor.

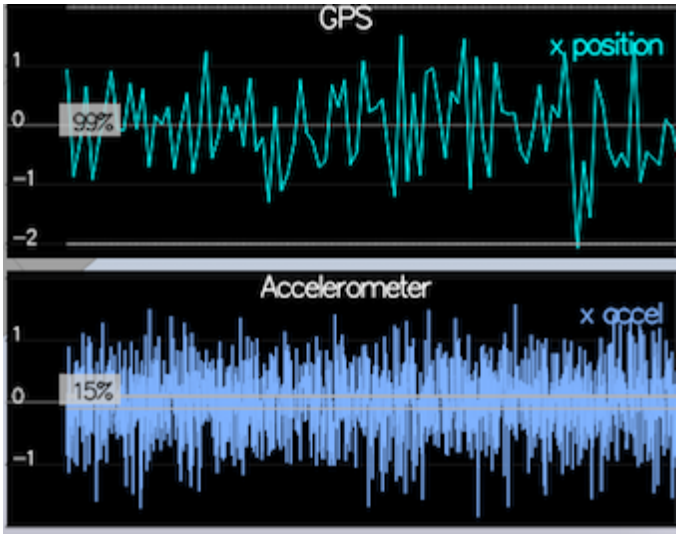

Figure 1: Noise in the GPS posiiton and IMU Accelerometer position measurement with some arbitrary bound.

In scenario 6 NoisySensors simulation, displayed in Figure 1 are two plots, one for GPS X position and one for The accelerometer's x acceleration measurement.

The dashed lines are a visualization of an arbitrary chosen standard deviation from 0 for each signal. The standard deviation chosen for GPS X position measurement is so large that it encloses 99% of the measurement. The standard deviation chosen for IMU Accelerometer X acceleration measurement is so small that it only encloses 15% of the measurement. Our task is to calculate the correct standard deviation for each sensor that should enclose 68% of the sensor measurements.

the collected data are recorded to the following csv files with headers:

- config/log/Graph1.txt (GPS X data),
- config/log/Graph2.txt (IMU Accelerometer X data).

Calculate standard deviation of the the GPS X signal and the IMU Accelerometer X signal from these two logged files with the following Python code:

```
import pandas as pd
import numpy as np

gps_x  = pd.read_csv('./config/log/Graph1.txt')
imu_ax = pd.read_csv('./config/log/Graph2.txt')

gps_x_std  = np.std(gps_x['␣Quad.GPS.X'])
imu_ax_std = np.std(imu_ax['␣Quad.IMU.AX'])

print('GPS␣position␣std␣is␣{0:.2f}'.format(gps_x_std))
print('IMU␣accelerometer␣position␣std␣is␣{0:.2f}'.format(imu_ax_std))
```

Calculation Result Output

```
GPS x position std is 0.72
IMU x accelerometer std is 0.51
```

Plug in the result into the top of config/6_Sensornoise.txt to set MeasuredStdDev_GPSPosXY and MeasuredStdDev_AccelXY to be the values just calculated.

```
MeasuredStdDev_GPSPosXY = 0.72
MeasuredStdDev_AccelXY = 0.51
```

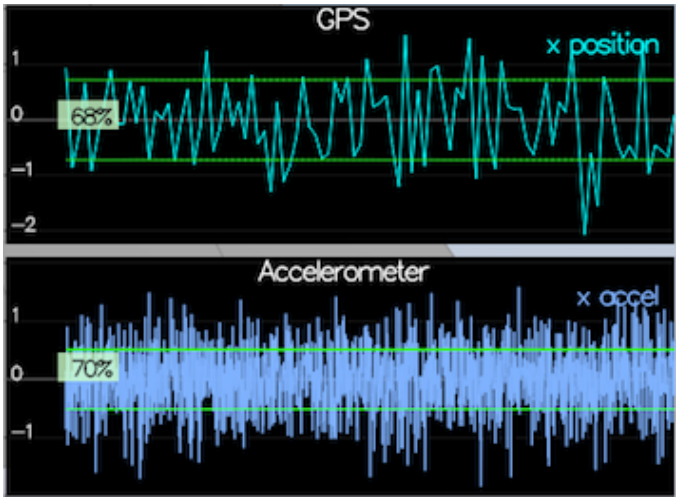

Figure 2: Noise in the GPS posiiton and IMU Accelerometer position measurement with $\pm 1\sigma$ bound.

Run the simulator again with corrected values, the dashed lines in Fig 2 in the simulation eventually turn green, indicating capturing approx 68% of the respective measurements, which is what we should expect within $\pm 1\sigma$ for a Gaussian noise model.

These calculated values also match the settings in SimulatedSensors.txt, where you can also grab the simulated noise parameters for all the other sensors.

```
[SimIMU]
AccelStd = .5, .5, 1.5
GyroStd = .5, .5, .5
dt = .002

[SimGPS]
PosStd = .7, .7, 2
#PosRandomWalkStd = .1, .1, .1
VelStd = .1, .1, .3
dt = .1
```

3.2 Attitude Estimation

In this step, we will include information from IMU and use complementary filters in stead of Kalman filter for our attitude state estimation.

The vehicle's attitude state, x_t consists pitch θ and roll ϕ.

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

The observation, z_t consists angles pitch θ and roll ϕ as estimated from the accelerometer in the global frame, and the angular velocity p and q from the gyro in the body frame.

$$z_t = \begin{bmatrix} \theta \\ \phi \\ p \\ q \end{bmatrix}$$

Since the pitch and roll angles obtained by an accelerometer have large noises but a small drift, while pitch and roll angles estimated by integrating angular velocity from gyro sensor have a little noise but a large drift, a linear complementary filter expressed below can obtain a more accurate attitude estimation, due to high-frequency noise attenuated after passing the low-pass filter, and the low-frequency signal be eliminated after passing the high-pass filter.

For pitch:

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

Similarly for roll:

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

τ is a time constant and T_s is the filter sampling period:

3.2.1 Linear Complementary Filter

Assume that θ and ϕ are small, so that the turn rates measured by the gyro in the body frame approximate the global turn rates,

$$\begin{bmatrix} \dot{\phi} \\ \dot{\theta} \\ \dot{\psi} \end{bmatrix} \approx \begin{bmatrix} p \\ q \\ r \end{bmatrix}$$

This allows us to use a linear complementary filter-type attitude filter, which is already implemented in the function UpdateFromIMU() in QuadEstimatorEKF.cpp, quoted below:

```
void QuadEstimatorEKF::UpdateFromIMU(V3F accel, V3F gyro)
{
    // SMALL ANGLE GYRO INTEGRATION:

    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;

    // CALCULATE UPDATE
    accelRoll = atan2f(accel.y, accel.z);
    accelPitch = atan2f(-accel.x, 9.81f);

    // FUSE INTEGRATION AND UPDATE
    rollEst = attitudeTau / (attitudeTau + dtIMU) * predictedRoll +
              dtIMU / (attitudeTau + dtIMU) * accelRoll;
    pitchEst = attitudeTau / (attitudeTau + dtIMU) * predictedPitch +
               dtIMU / (attitudeTau + dtIMU) * accelPitch;

    lastGyro = gyro;
}
```

In scenario 7 AttitudeEstimation simulation, IMU is the only sensor used and noise levels are set to 0. There are two plots visible in this simulation in Fig 3. The top graph is showing errors in each of the estimated Euler angles. The bottom shows the true Euler angles and the estimates. There's quite a bit of error in attitude estimation, 0.2 rad for yaw, 0.05 rad for pitch and roll.

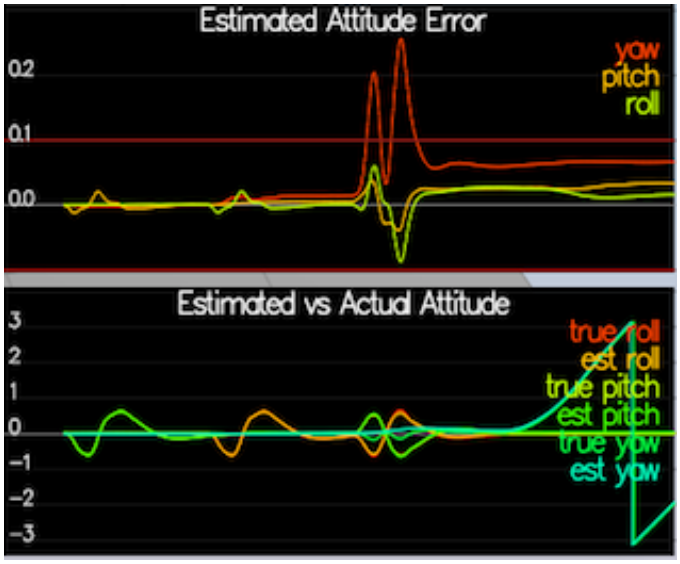

Figure 3: Errors in each of the estimated Euler angles from a linear complementary altitude filter.

3.2.2 Nonlinear Complementary Filter

We will improve the altitude estimation with implementation of a better non-linear complementary attitude filter using quaternions.

First, we define a quaternion, q_t, from the euler angles for θ , ϕ and ψ. The included Quaternion class has a handy Quaternion::FromEuler123_RPY() function for creating a quaternion from Euler Roll/Pitch/Yaw angles:

```
// use euler angles to define a quaternion
Quaternion<float> q_t = Quaternion<float>::FromEuler123_RPY(rollEst, pitchEst, ekfState(6));
```

Then the predicted quaternion, $\bar{q}_t$, can be obtained by integrating the measurement of the angular rates from the IMU in the body frame with the included Quaternion class Quaternion::IntegrateBodyRate() function:

```
// integrate the body rates into a new predicted quaternion
Quaternion<float>  q_t_bar = q_t.IntegrateBodyRate(V3D(gyro.x, gyro.y, gyro.z), dtIMU);
```

Finally we can obtain $\bar{\theta}_t$ and $\bar{\phi}_t$ as follows:

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

```
// extract roll, pitch, yaw from new quaternion
float predictedRoll  = q_t_bar.Roll();
float predictedPitch = q_t_bar.Pitch();
ekfState(6)     = q_t_bar.Yaw();
```

Using these predicated estimates, we can compute the non-linear complementary filter as the same way as above. For pitch:

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

Similarly for roll:

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

Put the above pieces together, we have an improved altitude state non-linear complementary filter implementation.

```
void QuadEstimatorEKF::UpdateFromIMU(V3F accel, V3F gyro)
{
    // Nonlinear Complementary Filter
    // use euler angles to define a quaternion
    Quaternion<float> q_t = Quaternion<float>::FromEuler123_RPY(rollEst, pitchEst, ekfState(6));

    // integrate the body rates into a new predicted quaternion
    Quaternion<float>  q_t_bar = q_t.IntegrateBodyRate(V3D(gyro.x, gyro.y, gyro.z), dtIMU);

    // extract roll, pitch, yaw from new quaternion
    float predictedRoll  = q_t_bar.Roll();
    float predictedPitch = q_t_bar.Pitch();
    ekfState(6)     = q_t_bar.Yaw();

    // CALCULATE UPDATE
    accelRoll = atan2f(accel.y, accel.z);
    accelPitch = atan2f(-accel.x, 9.81f);

    // FUSE INTEGRATION AND UPDATE
    rollEst = attitudeTau / (attitudeTau + dtIMU) * predictedRoll +
              dtIMU / (attitudeTau + dtIMU) * accelRoll;
    pitchEst = attitudeTau / (attitudeTau + dtIMU) * predictedPitch +
               dtIMU / (attitudeTau + dtIMU) * accelPitch;

    lastGyro = gyro;
}
```

The new non-linear complementary filter reduces the attitude errors from 0.2 rad rad to 0.01 rad for yaw, from 0.05 rad to 0.02 rad for pitch and roll, as shown in the screenshot below in Fig. 4, well within the requirement of 0.1 rad for each of the Euler angles for at least 3 seconds.

3.3 Implement Filter Prediction

3.3.1 Perfect IMU

Scenario 08 PredictState is configured to use a perfect IMU (only an IMU).

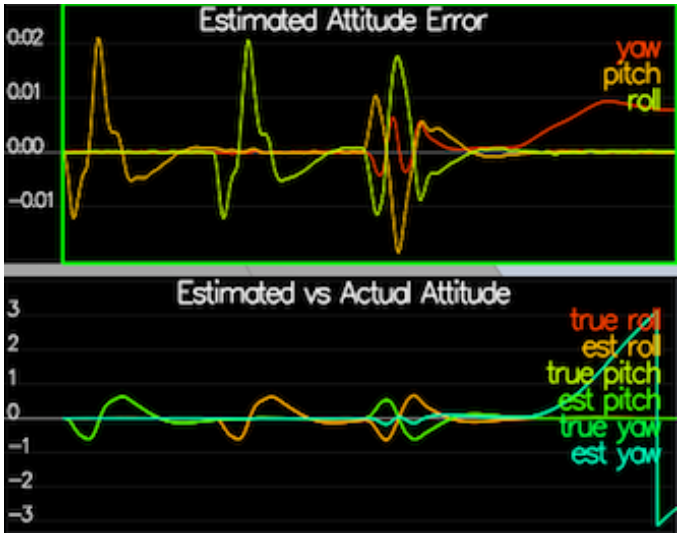

Figure 4: Errors in each of the estimated Euler angles from a non-linear complementary altitude filter.

The state, x_t we are estimating (excluding yaw, which is updated in the IMU update) position and velocity,

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

u_t is the IMU accelerometer measurement in the body frame, indicated by the superscripts b.

$$u_t = \begin{bmatrix} \ddot{x}^b \\ \ddot{y}^b \\ \ddot{z}^b \end{bmatrix}$$

To convert acceleration vector from body frame to inertial frame, an attitude Quaternion is created from the current state.

```
Quaternion<float> attitude = Quaternion<float>::FromEuler123_RPY(rollEst, pitchEst, curState(6));
```

Then Use Quaternion::Rotate_BtoI() function to rotate the vector from body frame to inertial frame.

```
V3F accel_i = attitude.Rotate_BtoI(accel);
```

Finally, in QuadEstimatorEKF.cpp, a simple state prediction is implemented in the PredictState() functon to predict the current state forward by time dt using current velocities and accelerations (already convered in the inertial frame, as indicated by the i superscripts). Since dt will be very short (on the order of 1ms) so simplistic integration methods are fine here.

The prediction function is expressed as,

$$\begin{bmatrix} x_{t+1,x} \\ x_{t+1,y} \\ x_{t+1,z} \\ x_{t+1,\dot{x}} \\ x_{t+1,\dot{y}} \\ x_{t+1,\dot{z}} \end{bmatrix} = \begin{bmatrix} x_{t,x} + x_{t,\dot{x}}\Delta t \\ x_{t,y} + x_{t,\dot{y}}\Delta t \\ x_{t,z} + x_{t,\dot{z}}\Delta t \\ x_{t,\dot{x}} + \ddot{x}^i\Delta t \\ x_{t,\dot{y}} + \ddot{y}^i\Delta t \\ x_{t,\dot{z}} + \ddot{z}^i\Delta t - g\Delta t \end{bmatrix}$$

```
VectorXf QuadEstimatorEKF::PredictState(VectorXf curState, float dt, V3F accel, V3F gyro)
{
    // Predict the current state forward by time dt using current accelerations
    // and body rates as input
    //
    // INPUTS:
    //   curState: starting state
    //   dt: time step to predict forward by [s]
    //   accel: acceleration of the vehicle, in body frame, *not including gravity* [m/s2]
    //   gyro: body rates of the vehicle, in body frame [rad/s]
    //
    // OUTPUT:
    //   return the predicted state as a vector

    assert(curState.size() == QUAD_EKF_NUM_STATES);
    VectorXf predictedState = curState;

    Quaternion<float> attitude = Quaternion<float>::FromEuler123_RPY(rollEst, pitchEst, curState(6));

    // rotates acceleration vector in body coordinate to global coords
    V3F accel_i = attitude.Rotate_BtoI(accel);

    // predict the current state using current accelerations and body rates
    predictedState(0) += curState(3)*dt;
    predictedState(1) += curState(4)*dt;
    predictedState(2) += curState(5)*dt;
    predictedState(3) += accel_i.x*dt;
    predictedState(4) += accel_i.y*dt;
    predictedState(5) += (accel_i.z - CONST_GRAVITY)*dt;

    return predictedState;
}
```

When run scenario 08 PredictState in the simulator, the estimator state track the actual state, with only reasonably slow drift, as shown in the Fig 5 below:

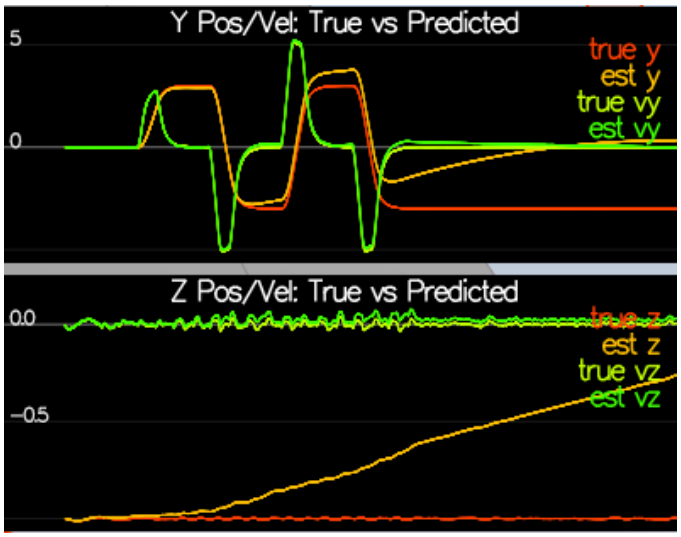

Figure 5: state prediction with perfect IMU.

3.3.2 Realistic IMU

Now let's introduce a realistic IMU with noise. In scenario 09 PredictionCov, a small fleet of quadcopter all using your prediction code to integrate forward. There are two plots in Fig 6: The top graph shows 10 (prediction-only) position X estimates, the bottom graph shows 10 (prediction-only) velocity estimates. However the estimated covariance (white bounds) currently do not capture the growing errors. This is because the covariance matrix has never been updated. It is always the value set in the QuadEstimatorEKF.txt

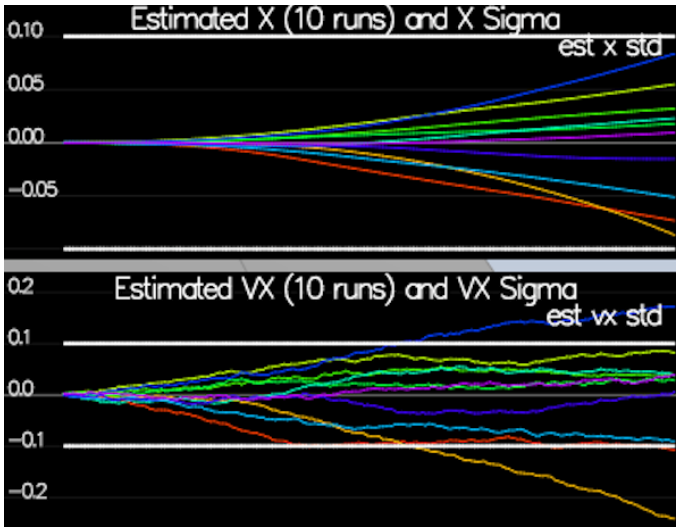

Figure 6: State covariance constant prediction for realistic IMU.

```
[QuadEstimatorEKF]
InitState = 0, 0, -1, 0, 0, 0, 0
InitStdDevs = .1, .1, .3, .1, .1, .3, .05
```

To update the covariance matrix to a predicted covariance matrix according to the EKF equation, first a partial derivative of the body-to-global rotation matrix in the function GetRbgPrime() need to be calculated. R'_{bg} is defined as:

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

```
MatrixXf QuadEstimatorEKF::GetRbgPrime(float roll, float pitch, float yaw)
{
    // 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

    // first, figure out the Rbg_prime
    MatrixXf RbgPrime(3, 3);
    RbgPrime.setZero();

    float cosTheta = cos(pitch);
    float sinTheta = sin(pitch);
    float cosPhi   = cos(roll);
    float sinPhi   = sin(roll);
    float cosPsi   = cos(yaw);
    float sinPsi   = sin(yaw);

    // Fill in matrix elements
    RbgPrime(0,0)  = -cosTheta*sinPsi;
    RbgPrime(0,1)  = -sinPhi*sinTheta*sinPsi - cosTheta*cosPsi;
    RbgPrime(0,2)  = -cosPhi*sinTheta*sinPsi + sinPhi*cosPsi;
    RbgPrime(1,0)  = -cosTheta*cosPsi;
    RbgPrime(1,1)  = sinPhi*sinTheta*cosPsi - cosTheta*sinPsi;
    RbgPrime(1,2)  = cosPhi*sinTheta*cosPsi + sinPhi*sinPsi;
    // the last row of the matrix are all zeros

    return RbgPrime;
}
```

Once R'_{bg} function is implement, the rest of the prediction step to predict the state covariance forward in Predict() is also implemented.

First we take the Jacobian:

$$g'(x_t, u_t, \Delta t) = \begin{bmatrix} 1 & 0 & 0 & \Delta t & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & \Delta t & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & \Delta t & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & R'_{bg}[0:]u_t[0:3]\Delta t \\ 0 & 0 & 0 & 0 & 1 & 0 & R'_{bg}[1:]u_t[0:3]\Delta t \\ 0 & 0 & 0 & 0 & 0 & 1 & R'_{bg}[2:]u_t[0:3]\Delta t \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 \end{bmatrix} \tag{1}$$

Then update covariance according to the classic EKF algorithm update equations to calculate the predictied state covariance in each update step:

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

where Σ_t is the state covariance matrix, and Q_t is the transition covariance.

```
void QuadEstimatorEKF::Predict(float dt, V3F accel, V3F gyro)
{
    // Predict the current covariance forward by dt using the current accelerations
    // and body rates as input.
    //
    // INPUTS:
    //   dt: time step to predict forward by [s]
    //   accel: acceleration of the vehicle, in body frame, *not including gravity* [m/s2]
    //   gyro: body rates of the vehicle, in body frame [rad/s]
    //   state (member variable): current state (state at the beginning of this prediction)
    //
    // OUTPUT:
    //   update the member variable cov to the predicted covariance

    // 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));

    // an empty Jacobian currently simply set to identity
    MatrixXf gPrime(QUAD_EKF_NUM_STATES, QUAD_EKF_NUM_STATES);
    gPrime.setIdentity();

    gPrime(0,3) = dt;
    gPrime(1,4) = dt;
    gPrime(2,5) = dt;
    gPrime(3,6) = (RbgPrime(0)*accel).sum()*dt;
    gPrime(4,6) = (RbgPrime(1)*accel).sum()*dt;
    gPrime(5,6) = (RbgPrime(2)*accel).sum()*dt;

    ekfCov = gPrime * ekfCov * gPrime.transpose() + Q;

    ekfState = newState;
}
```

Run the implemented EKF covariance prediction and tune the process parameters QPosXYStd to 0.01 and the QVelXYStd to 0.1 in QuadEstimatorEKF.txt to try to capture the magnitude of the error.

```
# Process noise model
# note that the process covariance matrix is diag(pow(QStd,2))*dtIMU

QPosXYStd = .01
QVelXYStd = .1
```

In the first part of the plot in Fig 7, our covariance (the white line) grows very much like the data.

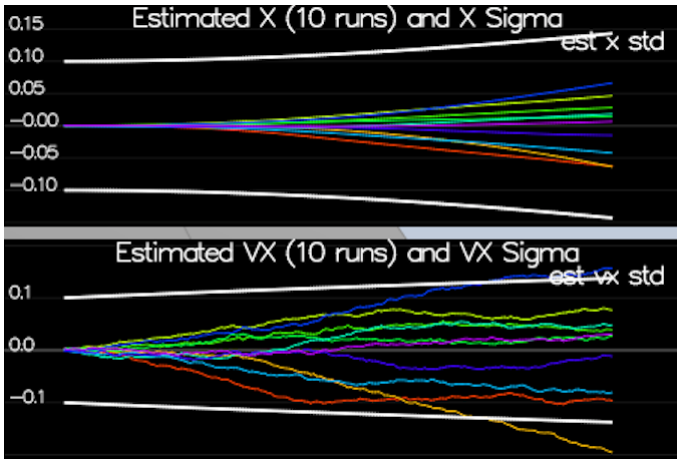

Figure 7: State covariance with EKF prediction for realistic IMU.

3.4 Magnetometer Update

In the prevous steps, only the accelerometer and gyro sensors were used for the state estimation. In this step, magnetometer sensor measurement is used to estimate the vehicle's heading.

Scenario 10 MagUpdate uses a realistic IMU, but the magnetometer update hasn't been implemented yet, resulting in the estimate yaw drifting away from the real value. The estimated standard deviation of that state (white boundary) is also increasing.

Tune the parameter QYawStd to 0.02 in QuadEstimatorEKF.txt for the QuadEstimatorEKF so that it approximately captures the magnitude of the drift, as demonstrated in Fig 8:

```
# Process noise model
# note that the process covariance matrix is diag(pow(QStd,2))*dtIMU

QYawStd = .02
```

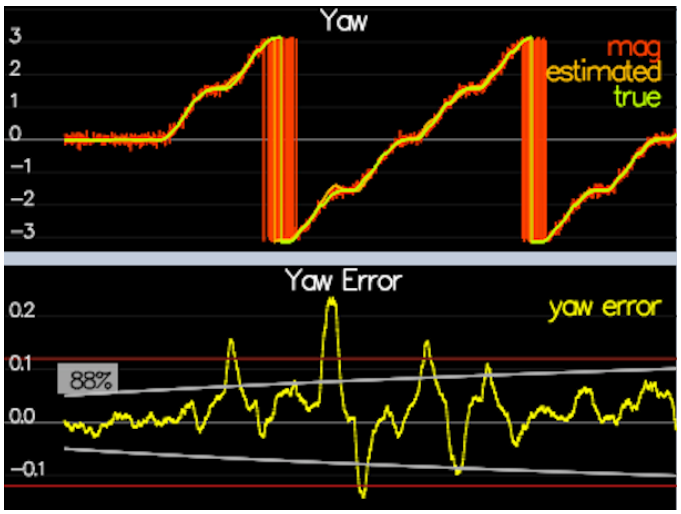

Figure 8: Yaw estimation with no magnetometer update.

Now implement magnetometer update in the function UpdateFromMag(). The estimated yaw is the last element in the state vector: ekfState(6).

The reading from the magnetometer z_t reporting yaw in the global frame.

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

a measurement function $h(x_t)$ is defined as,

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

$h'(x_t)$ is the Jacobian of h with respect to x_t. This Jacobian is a 1×7 matrix.

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

The angle error between the current estimated state and the magnetometer measured value is normalized so the update to the yaw would be the short way around the circle, not the long way.

EKF also updates magnetomer measurement covariance R_Mag, which is a member variable.

The C++ implementation of the magnetometer update:

```
void QuadEstimatorEKF::UpdateFromMag(float magYaw)
{
    // MAGNETOMETER UPDATE
    VectorXf z(1), zFromX(1);
    z(0) = magYaw;

    MatrixXf hPrime(1, QUAD_EKF_NUM_STATES);
    hPrime.setZero();
     hPrime << 0, 0, 0, 0, 0, 0, 1;

    zFromX(0) = ekfState(6);
    float angle_diff = fmod(zFromX(0) - z(0), (2.0f*(float)M_PI));
    if (angle_diff <= -(float)M_PI)
        zFromX(0) += (2.0f*(float)M_PI);
    else if (angle_diff > (float)M_PI)
        zFromX(0) -= (2.0f*(float)M_PI);

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

The resulting aaw error plot is in Fig 9, with the parameter QYawStd re-tuned to 0.06 to better balance between the long term drift and short-time noise from the magnetometer.

```
# Process noise model
# note that the process covariance matrix is diag(pow(QStd,2))*dtIMU

QYawStd = .06
```

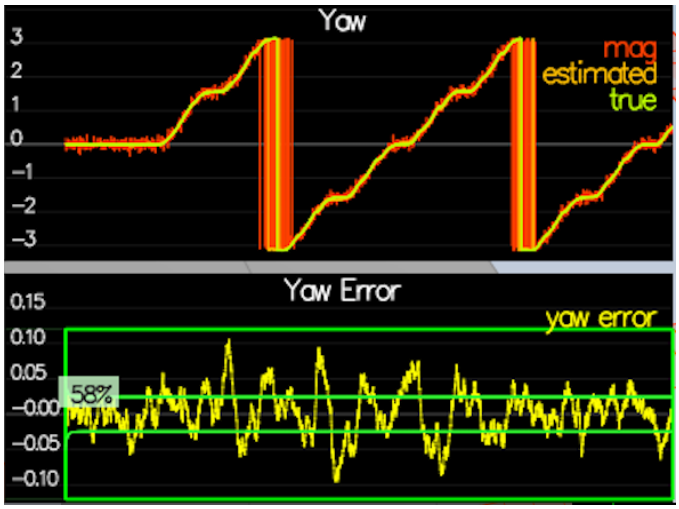

Figure 9: Yaw estimation with magnetometer update.

3.5 Closed Loop + GPS Update

Position and velocity observation are measured from the GPS.

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

The measurement model is:

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

Jacobian $h'(x_t)$ is the partial derivative of $h(x_t)$, which is a identity matrix, augmented with a vector of zeros for $\frac{\partial}{\partial x_{t,\phi}} h(x_t)$:

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

Implement the EKF GPS Update in the function UpdateFromGPS():

```
void QuadEstimatorEKF::UpdateFromGPS(V3F pos, V3F vel)
{
    // GPS UPDATE
    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();
    hPrime.topLeftCorner(6, 6) = MatrixXf::Identity(6, 6);

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

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

Tune the process noise model in QuadEstimatorEKF.txt to try to approximately capture the error of the filter.

Without having to change values of QPosZStd and QVelZStd, the entire simulation cycle can be completed with estimated position error of < 0.4m as shown in Fig 10, less than the $<$ 1m requirement.

Now we have a fulling working estimator!

```
# Process noise model
# note that the process covariance matrix is diag(pow(QStd,2))*dtIMU

QPosZStd = .05
QVelZStd = .1
```

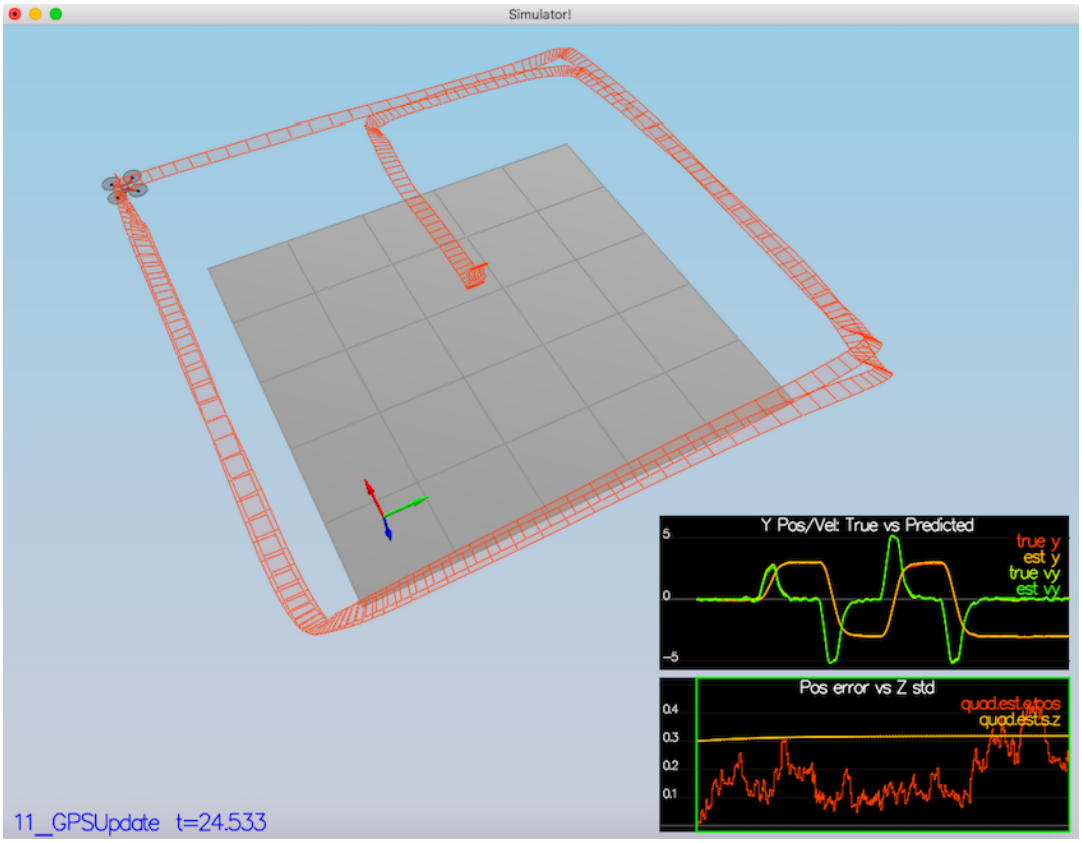

Figure 10: Position and velocity estimation with GPS update.

4 Adding Previously Implemented Controller

Replace QuadController.cpp with the controller and uadControlParams.txt with the control parameters from in the last project. Run scenario 11 GPSUpdate, the entire simulation cycle is completed once again with an estimated position error of $<$ 1m, shown in Fig 11, without de-tuning any control gains.

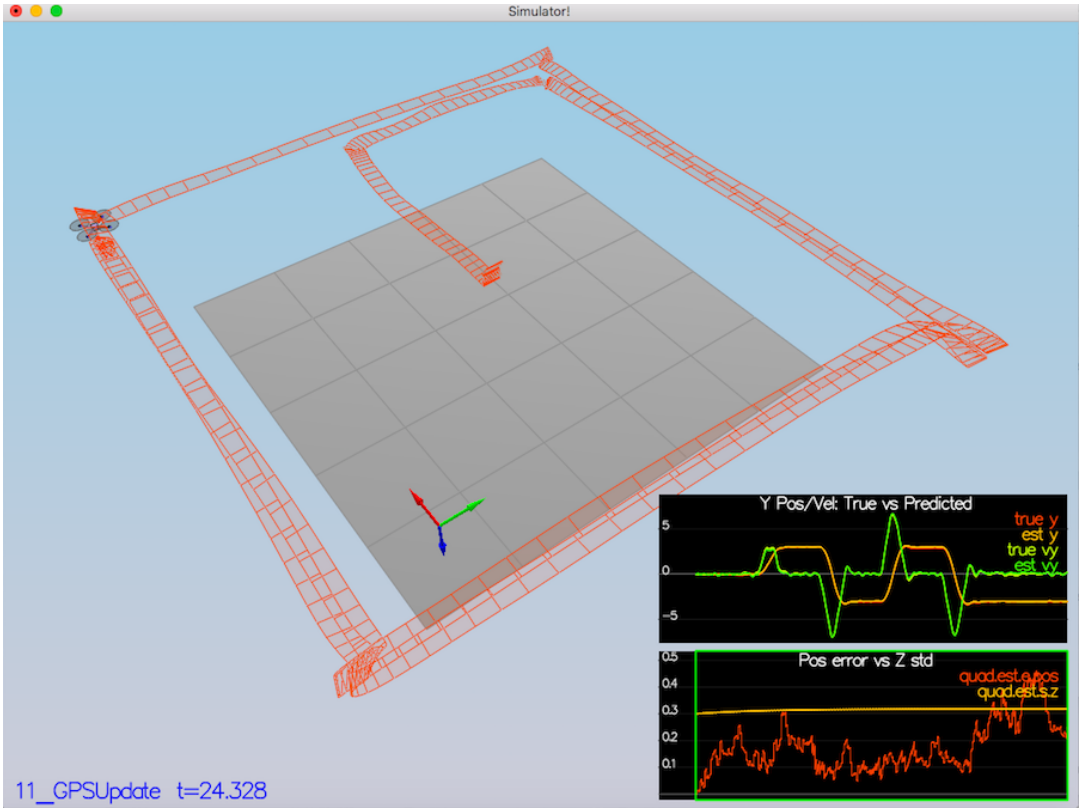

Figure 11: Position and velocity estimation with GPS update from my previous controller.

Appendix

This Appendix section is removed, in honor of the appendix being removed from our dear student friend @Luis95014 at 5:30 am on May 27, 2018. Hope he recover well and finish his FCND estimation project soon.