Project: Estimation

Setup

This project will continue to use the C++ development environment you set up in the Controls C++ project.
Clone the repository
git clone https://github.com/udacity/FCND-Estimation-CPP.git
Import the code into your IDE like done in the Controls C++ project
You should now be able to compile and run the estimation simulator just as you did in the controls project

Project Structure

For this project, you will be interacting with a few more files than before.
The EKF is already partially implemented for you in **QuadEstimatorEKF.cpp**
Parameters for tuning the EKF are in the parameter file **QuadEstimatorEKF.txt**
When you turn on various sensors (the scenarios configure them, e.g. Quad.Sensors += SimIMU, SimMag, SimGPS), additional sensor plots will become available to see what the simulated sensors measure.
The EKF implementation exposes both the estimated state and a number of additional variables. In particular:
Quad.Est.E.X is the error in estimated X position from true value. More generally, the variables in <vehicle>.Est.E.* are relative errors, though some are combined errors (e.g. MaxEuler).
Quad.Est.S.X is the estimated standard deviation of the X state (that is, the square root of the appropriate diagonal variable in the covariance matrix). More generally, the variables in <vehicle>.Est.S.* are standard deviations calculated from the estimator state covariance matrix.
Quad.Est.D contains miscellaneous additional debug variables useful in diagnosing the filter. You may or might not find these useful but they were helpful to us in verifying the filter and may give you some ideas if you hit a block.

config Directory

In the config directory, in addition to finding the configuration files for your controller and your estimator, you will also see configuration files for each of the simulations. For this project, you will be working with simulations 06 through 11 and you may find it insightful to take a look at the configuration for the simulation.

As an example, if we look through the configuration file for scenario 07, we see the following parameters controlling the sensor:

# Sensors

```
Quad.Sensors = SimIMU
# use a perfect IMU
SimIMU.AccelStd = 0,0,0
SimIMU.GyroStd = 0,0,0
```

This configuration tells us that the simulator is only using an IMU and the sensor data will have no noise. You will notice that for each simulator these parameters will change slightly as additional sensors are being used and the noise behavior of the sensors change.

The Tasks

Once again, you will be building up your estimator in pieces. At each step, there will be a set of success criteria that will be displayed both in the plots and in the terminal output to help you along the way.

Project outline:

1- **Sensor Noise:**

For the controls project, the simulator was working with a perfect set of sensors, meaning none of the sensors had any noise. The first step to adding additional realism to the problem, and developing an estimator, is adding noise to the quad's sensors. For the first step, you will collect some simulated noisy sensor data and estimate the standard deviation of the quad's sensor.

Run the simulator in the same way as you have before

Choose scenario 06_NoisySensors. In this simulation, the interest is to record some sensor data on a static quad, so you will not see the quad move. You will see two plots at the bottom, one for GPS X position and one for The accelerometer's x measurement. The dashed lines are a visualization of a single standard deviation from 0 for each signal. The standard deviations are initially set to arbitrary values (after processing the data in the next step, you will be adjusting these values). If they were set correctly, we should see ~68% of the measurement points fall into the +/- 1 sigma bound. When you run this scenario, the graphs you see will be recorded to the following csv files with headers: config/log/Graph1.txt (GPS X data) and config/log/Graph2.txt (Accelerometer X data).

Process the logged files to figure out the standard deviation of the the GPS X signal and the IMU Accelerometer X signal.

Plug in your result into the top of config/6_Sensornoise.txt. Specially, set the values for MeasuredStdDev_GPSPosXY and MeasuredStdDev_AccelXY to be the values you have calculated.

Run the simulator. If your values are correct, the dashed lines in the simulation will eventually turn green, indicating you're capturing approx 68% of the respective measurements (which is what we expect within +/- 1 sigma bound for a Gaussian noise model)

Success criteria: Your standard deviations should accurately capture the value of approximately 68% of the respective measurements.

NOTE: Your answer should match the settings in SimulatedSensors.txt, where you can also grab the simulated noise parameters for all the other sensors.

2- Attitude Estimation:

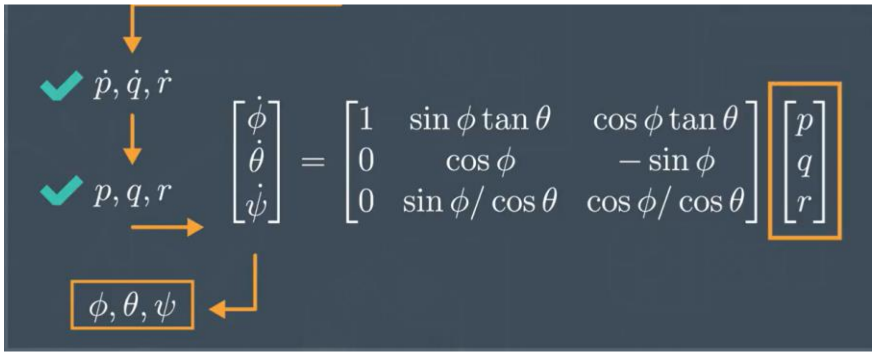
Now let's look at the first step to our state estimation: including information from our IMU. In this step, you will be improving the complementary filter-type attitude filter with a better rate gyro attitude integration scheme.

1- Run scenario 07_AttitudeEstimation. For this simulation, the only sensor used is the IMU and noise levels are set to 0 (see config/07_AttitudeEstimation.txt for all the settings for this simulation). There are two plots visible in this simulation.

- The top graph is showing errors in each of the estimated Euler angles.
- The bottom shows the true Euler angles and the estimates. Observe that there's quite a bit of error in attitude estimation.

2- In QuadEstimatorEKF.cpp, you will see the function UpdateFromIMU() contains a complementary filter-type attitude filter. To reduce the errors in the estimated attitude (Euler Angles), implement a better rate gyro attitude integration scheme. You should be able to reduce the attitude errors to get within 0.1 rad for each of the Euler angles, as shown in the screenshot below.

- UpdateFromIMU:
  - `create a rotation matrix based on your current Euler angles using this equation from the control lesson :`



$$\begin{bmatrix} \dot{\phi} \\ \dot{\theta} \\ \dot{\psi} \end{bmatrix} = \begin{bmatrix} 1 & \sin\phi\tan\theta & \cos\phi\tan\theta \\ 0 & \cos\phi & -\sin\phi \\ 0 & \sin\phi/\cos\theta & \cos\phi/\cos\theta \end{bmatrix} \boxed{\begin{bmatrix} p \\ q \\ r \end{bmatrix}}$$

The rotation matrix is :

$$\begin{bmatrix} 1 & \sin\phi\tan\theta & \cos\phi\tan\theta \\ 0 & \cos\phi & -\sin\phi \\ 0 & \sin\phi/\cos\theta & \cos\phi/\cos\theta \end{bmatrix}$$

And we have the p,q,r from the gyros readings ,by multiplying rotation matrix by the gyros [p.q.r] then we can get the eular angles rates theta_dot , phi_dot, psi_dot
and to get the predicted roll and pitch (phi , theta) we use those equations :

```
predicted_Pitch = Estimated_pitch + theta_dot * dt
predicted_Roll = Estimated_Roll + phi_dot * dt
```

```
3 - Prediction Step:
    In this next step you will be implementing the prediction step of your filter.

    Run scenario 08_PredictState. This scenario is configured to use a perfect IMU (only an IMU).
           Due to the sensitivity of double-integration to attitude errors, we've made the
           accelerometer update very insignificant (QuadEstimatorEKF.attitudeTau = 100). The plots
           on this simulation show element of your estimated state and that of the true state. At
           the moment you should see that your estimated state does not follow the true state.

    In QuadEstimatorEKF.cpp, implement the state prediction step in the PredictState() functon. If
           you do it correctly, when you run scenario 08_PredictState you should see the estimator
           state track the actual state, with only reasonably slow drift,

    The current state  is x_t [ x , y , z , x_dot , y_dot , z_dot ]
    To get to the predicted state forward x_t+1 we use those functions for X_t+1 , Y_t+1 ,Z_t+1
```

$$\begin{gathered} x_{t,x} + x_{t,\dot{x}}\Delta t \\ x_{t,y} + x_{t,\dot{y}}\Delta t \\ x_{t,z} + x_{t,\dot{z}}\Delta t \end{gathered}$$

```
    The accelerometer readings are in the body frame so we need to transform them into the inertial
           ( global) frame first using a built in function Rotate_BtoI()
    Then we can get the X_dot_t+1 , Y_dot_t+1 , Z_dot_t+1 from those functions

    X_dot_t+1 = X_dot_t + Accel_X_dot_dot * dt
    Y_dot_t+1 = Y_dot_t + Accel_Y_dot_dot * dt
    Z_dot_t+1 = Z_dot_t + Accel_Z_dot_dot * dt - gravity *dt
```

- In QuadEstimatorEKF.cpp, calculate the partial derivative of the body-to-global rotation matrix in the function GetRbgPrime(). Once you have that function implement, implement the rest of the prediction step (predict the state covariance forward) in Predict().
  First we need to create the R_bg_prime matrix which rotates from the body frame to
  The global frame and take the derivative with respect to psi and it's as follow :

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

Secondly we make the predict step by creating the Jacobian g_prime as follows :

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

Then we then we update the covariance according to this function: $\bar{\Sigma}_t = G_t \Sigma_{t-1} G_t^T + Q_t$

Then we tune the QPosXYStd and the QVelXYStd process parameters in QuadEstimatorEKF.txt to try to capture the magnitude of the error

4- Magnetometer Update:

Up until now we've only used the accelerometer and gyro for our state estimation. In this step, you will be adding the information from the magnetometer to improve your filter's performance in estimating the vehicle's heading.

Run scenario 10_MagUpdate. This scenario uses a realistic IMU, but the magnetometer update hasn't been implemented yet. As a result, you will notice that the estimate yaw is drifting away from the real value (and the estimated standard deviation is also increasing). Note that in this case the plot is showing you the estimated yaw error (quad.est.e.yaw), which is drifting away from zero as the simulation runs. You should also see the estimated standard deviation of that state (white boundary) is also increasing.

Tune the parameter QYawStd (QuadEstimatorEKF.txt) for the QuadEstimatorEKF so that it approximately captures the magnitude of the drift

Implement magnetometer update in the function **UpdateFromMag**():

In this step we update the derivative of the measurement model h_prime to look like this :

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

Set the sensor reading matrix to the psi from the state vector `[x ,y ,z , x_dot , y_dot , z_dot, psi]`

And then normalize the difference between the measured and estimated yaw

5- Closed Loop + GPS Update:

Run scenario 11_GPSUpdate. At the moment this scenario is using both an ideal estimator and and ideal IMU. Even with these ideal elements, watch the position and velocity errors (bottom right). As you see they are drifting away, since GPS update is not yet implemented.

Let's change to using your estimator by setting Quad.UseIdealEstimator to 0 in config/11_GPSUpdate.txt. Rerun the scenario to get an idea of how well your estimator work with an ideal IMU.

Now repeat with realistic IMU by commenting out these lines in config/11_GPSUpdate.txt:

```
#SimIMU.AccelStd = 0,0,0
#SimIMU.GyroStd = 0,0,0
```

Tune the process noise model in QuadEstimatorEKF.txt to try to approximately capture the error you see with the estimated uncertainty (standard deviation) of the filter.

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

We update the h_prime matrix to take this form :

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

And we update the readings matrix to take this form :

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

6- Adding Your Controller :

Adding the controller we made at the control lesson and retune it to work with our estimator