# Building An Estimator Project

Dhar Rawal [dharrawal@gmail.com](mailto:dharrawal@gmail.com) 281-995-6423 12/15/2020

## Udacity – Flying Car and Autonomous Flight Engineer Program

## Introduction

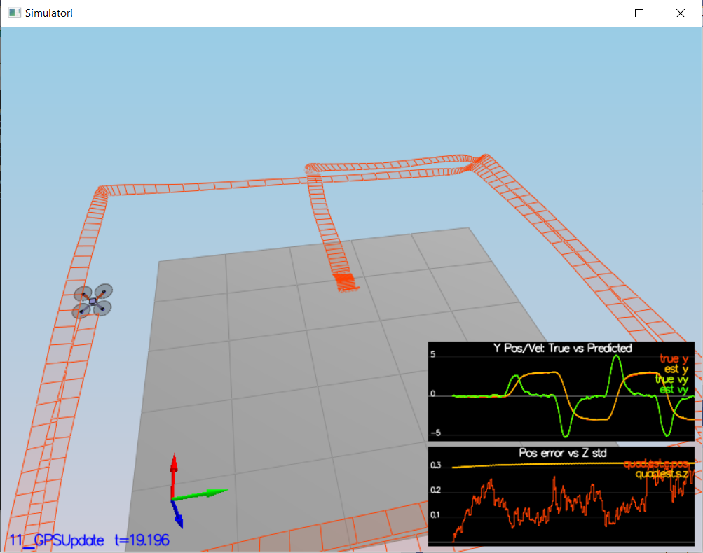
The challenge here is to build an estimator for a simulated drone such that it flies a given path with the pre-built controller in the presence of system noise and model inaccuracies as well as sensor noise and drift. This estimator is implemented in C++ for the same reasons as the controller was built in C++, namely speed and repeatable timing.

**Sensor integration will be done using the extended Kalman filter.**

**We will use the accelerometer and gyro measurements to help predict the position of the drone. These two sensors return measurements with relatively high frequency, which is what we need to predict. Note how the sensors are balanced in terms of noise and drift.**

**We will use the magnetometer and GPS measurements to tweak/update our confidence in the predictions. These two sensors return measurements relatively slowly. Again, note how the sensors are balanced in terms of noise and drift.**

Image of the drone flying in the simulator using our estimator



## Starter Code

The starter code consists of a esimator C++ project. The important files are – QuadEstimatorEKF.cpp/h and QuadEstimatorEKF.txt. For the last scenario challenge in this project, we will use the QuadControl.cpp and QuadControlParams.txt that we wrote for the controls project.

### QuadEstimatorEKF.cpp/h

This is where I worked to implement the estimation methods:

|  |  |
| --- | --- |
| **Methods** | **Description** |
| UpdateFromIMU | Implements a non-linear complementary filter to predict attitude using the accelerometer and gyro sensors |
| PredictState | Predicts the next state given the current state and time dt using the accelerometer sensor readings. This is the “g” function of the EKF |
| GetRbgPrime | Helper function that returns the jacobian of the rotation matrix, used for the predict part of the EKF |
| Predict | Predicts the covariance. The new state is already calculated by the PredictState function |
| UpdateFromGPS | Uses GPS position and velocity sensor data to update the state and covariance as part of EKF |
| UpdateFromMag | Uses Magnetometer yaw measurement to update the state and covariance as part of EKF. |
| Update | This function implements the update part of the EKF. |

### SensorNoise.txt

This is where we adjust the standard deviations of the GPS and accelerometer after analyzing logs of the sensor noise measurements

### 11\_GPSUpdate.txt

This is where we can turn off the ideal estimator and ideal sensors while tuning the control parameters to work with our estimator

### QuadEstimatorEKF.txt

This is where we adjust the standard deviations of the prediction process parameters QPosXYStd and QVelXYStd to capture the magnitude of the system/model dynamics errors

### QuadControlParams.txt

This is where you can retune the control gains to improve drone handling with our estimator in the face of system/model inaccuracies and sensor noise and drift.

## Determine standard deviation of GPS and Accelerometer Measurement Noise

Just run the 1st scenario, import the data into Matlab and calculate the standard deviations using the Matlab code below:

clc; clear;

gpsXData = importdata("Graph1.txt", ',', 1);

gpsX = gpsXData.data(:, 2);

std(gpsX)

accXData = importdata("Graph2.txt", ',', 1);

accX = accXData.data(:, 2);

std(accX)

## Non-linear complementary filter for a better Attitude estimate

This is where we integrate the accelerometer and gyro measurements to improve our estimate of the attitude. Code from UpdateFromIMU() is below:

## // gyro x, y, z are angular rates (velocities p, q, r) in body frame

## // pitchEst, rollEst and ekfState(6) are the Euler angles (angles that the body frame makes w.r.t. inertial frame)

## // accel x, y, z are accelerations in the body frame

## // convert the estimates into a quaternion

## Quaternion<float> attitudeEstimate = Quaternion<float>::FromEuler123\_RPY(rollEst, pitchEst, ekfState(6));

## // call IntegrateBodyRate and pass in the gyro rates

## V3D pqr(gyro.x, gyro.y, gyro.z);

## attitudeEstimate.IntegrateBodyRate(pqr, dtIMU);

## // get predicted pitch and roll values as Euler angles (angles that the body frame makes w.r.t. inertial frame)

## float predictedPitch = attitudeEstimate.Pitch();

## float predictedRoll = attitudeEstimate.Roll();

## ekfState(6) = attitudeEstimate.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;

## EKF Prediction Implementation

Below is the relevant code to implement the “g” function. The predicted state is basically the “mu\_bar” in the EKF algorithm

## // This is not the full EKF.

## // It's just the g function of EKF

## // Run with just this code to see effect of model noise (inaccuracies). Since we assume perfect IMU

## V3F accelerationsInInertialFrame = attitude.Rotate\_BtoI(accel);

## accelerationsInInertialFrame.z -= GRAVITY; // take out gravitational acceleration from the z component of acceleration

## // gyro rotation rates are NOT USED HERE

## predictedState[0] += predictedState[3]\*dt; // x

## predictedState[1] += predictedState[4]\*dt; // y

## predictedState[2] += predictedState[5]\*dt; // z

## predictedState[3] += accelerationsInInertialFrame.x\*dt; // x dot

## predictedState[4] += accelerationsInInertialFrame.y\*dt; // y dot

## predictedState[5] += accelerationsInInertialFrame.z\*dt; // z dot

## predictedState[6] = predictedState[6]; // psi does not change

Below is the relevant code to implement the “g\_prime” function. This is used to predict the covariance of the new state:

**From GetRbgPrime():**

float theta = pitch;

float phi = roll;

float psi = yaw;

RbgPrime(0, 0) = -cosf(theta) \* sinf(psi);

RbgPrime(0, 1) = -sinf(phi) \* sinf(theta) \* sinf(psi) - cosf(phi)\*cosf(psi);

RbgPrime(0, 2) = -cosf(phi)\*sinf(theta) \* sinf(psi) + sinf(phi)\*cosf(psi);

RbgPrime(1, 0) = cosf(theta) \* cosf(psi);

RbgPrime(1, 1) = sinf(phi)\*sinf(theta)\*cosf(psi) - cosf(phi)\*sinf(psi);

RbgPrime(1, 2) = cosf(phi)\*sinf(theta)\*cosf(psi) + sinf(phi)\*sinf(psi);

RbgPrime(2, 0) = 0;

RbgPrime(2, 1) = 0;

RbgPrime(2, 2) = 0;

**From Predict()**

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;

MatrixXf Gt = gPrime;

MatrixXf GtTranspose = Gt.transpose();

ekfCov = Gt \* ekfCov \* GtTranspose + Q;

## EKF Update of yaw using the Magnetometer measurement estimates

hPrime(0, 6) = 1.0f;

zFromX(0) = ekfState(6);

// normalize difference between yaws

if (zFromX(0) - z(0) > F\_PI) zFromX(0) -= 2.f \* F\_PI;

if (zFromX(0) - z(0) < -F\_PI) zFromX(0) += 2.f \* F\_PI;

## EKF Update of position and velocities using GPS measurement estimates

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(0, 0) = hPrime(1, 1) = hPrime(2, 2) = hPrime(3, 3) = hPrime(4, 4) = hPrime(5, 5) = 1.0f;

## Actual EKF Update Implementation

This has already been implemented for us.

## Control Integration and Tuning

Once the estimation code was in place and tested, the next step is to integrate our PID controller and retune it to fly the drone successfully.

TIPS:

* Reduce the gains.
* Tune control gains using ideal estimator and sensors first before testing/tweaking with our approximate estimator and noisy/drifting sensors.

### FINAL TUNED CONTROL PARAMETERS

# Position control gains

kpPosXY = 1.6 #2.4

kpPosZ = 2.0 #3

# Integral control gains

KiPosZ = 24

# Velocity control gains

kpVelXY = 7 #10.0

kpVelZ = 9 #14.1

# Angle control gains

kpBank = 8

kpYaw = 1.7

# Angle rate gains

kpPQR = 80, 80, 5.1

## References

* Estimation for Quadrotors by Stefanie Tellex, Andy Brown and Sergei Lupashin
* Udacity lectures and exercises