Building an Estimator

1. Step1: Sensor noise:

In this step, an excel sheet is used to import sensors data and then calculate the standard deviation. So, the solution is:

```
MeasuredStdDev_GPSPosXY = 0.7
```

```
MeasuredStdDev_AccelXY = 0.5
```

2. Step2: Attitude estimation:

In this step a simplistic rotation matrix is implemented

```
// defining the rotation matrix...
    Mat3x3F rot = Mat3x3F::Zeros();
    float phi = rollEst;
    float theta = pitchEst;
    rot(0, 0) = 1;
    rot(0, 1) = sin(phi) * tan(theta);
    rot(0, 2) = cos(phi) * tan(theta);
    rot(1, 0) = 0;
    rot(1, 1) = cos(phi);
    rot(1, 2) = -sin(phi);
    rot(2, 0) = 0;
    rot(2, 1) = sin(phi) / cos(theta);
    rot(2, 2) = cos(phi) / cos(theta);
    V3F angle_dot = rot * gyro;
    float predictedRoll = rollEst + dtIMU * angle_dot.x;
    float predictedPitch = pitchEst + dtIMU * angle_dot.y;
    ekfState(6) = ekfState(6) + dtIMU * angle_dot.z;
    // 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;
                                         }
```

3. Step3: Prediction :

In this step a simple integration is implemented in the PridictState function

```
V3F Inertial_frame_acc = attitude.Rotate_BtoI(accel) - V3F(0.0, 0.0, CONST_GRAVITY);

  predictedState(0) = curState(0) + curState(3) * dt; // X-pos
  predictedState(1) = curState(1) + curState(4) * dt; // Y-pos
  predictedState(2) = curState(2) + curState(5) * dt; // Z-pos
  predictedState(3) = curState(3) + Inertial_frame_acc.x * dt; // X-pos
  predictedState(4) = curState(4) + Inertial_frame_acc.y * dt; // Y-pos
  predictedState(5) = curState(5) + Inertial_frame_acc.z * dt; // Z-pos
```

Partial derivative of the body-to-global rotation matrix in the function GetRbgPrime() function

```
 float theta = pitch;
  float phi = roll;
  float psi = yaw;
  float R11, R12, R13, R21, R22, R23;

  R11 = -cos(theta) * sin(psi);
  R21 = cos(theta) * cos(psi);

  R12 = -sin(phi) * sin(theta) * sin(psi) - cos(phi) * cos(psi);
  R22 = sin(phi) * sin(theta) * cos(psi) - cos(phi) * sin(psi);

  R13 = -cos(phi) * sin(theta) * sin(psi) + sin(phi) * cos(psi);
  R23 = cos(phi) * sin(theta) * cos(psi) + sin(phi) * sin(psi);

  RbgPrime(0, 0) = R11;
  RbgPrime(0, 1) = R12;
  RbgPrime(0, 2) = R13;
  RbgPrime(1, 0) = R21;
  RbgPrime(1, 1) = R22;
  RbgPrime(1, 2) = R23;
```

Implementing the rest of the prediction step (predict the state covariance forward) in Predict() function (tuned values `QPosXYStd= 0.05` `QVelXYStd= 0.1`)

```
// g' update:
  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;

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

4. Step4: magnetometer update:

After tuning (QYawStd = 0.31) , then Implementing magnetometer update in the function UpdateFromMag()

```
hPrime(QUAD_EKF_NUM_STATES - 1) = 1;
  zFromX(0) = ekfState(6);

  if ((z(0) - zFromX(0)) > F_PI) {
      zFromX(0) += 2.f * F_PI;
  }
  else if ((z(0) - zFromX(0)) < -F_PI) {
      zFromX(0) -= 2.f * F_PI;
                                            }
```

5. Step5: Closed loop + GPS update:

Implementing the EKF GPS Update in the function UpdateFromGPS()

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

6. Step6: Adding Your Controller:

The files `QuadController.cpp` and `QuadControlParams.txt` is replaced with a retuned parameters from the previous control project