Extended Kalman Filter (EKF) for Roll and Pitch Angle Estimation Using MPU6050

RAVI YADAV

December 31, 2024

Abstract

The Extended Kalman Filter (EKF) is a powerful algorithm used in signal processing and control systems to estimate the state of a nonlinear dynamic system. The EKF extends the capabilities of the Kalman Filter to handle nonlinear systems, making it widely used in applications such as navigation, robotics, and time-series analysis. In this article, MPU6050 sensor is used for estimating roll and pitch angles estimation because this sensor serves as accelerometer and rate gyro sensor.

1 Introduction

The original Kalman Filter, developed by Rudolf E. Kálmán in the 1960s, revolutionized the field of linear filtering and estimation. However, real-world systems often exhibit nonlinear behavior, prompting the development of the EKF. This extension enables the application of Kalman's principles to nonlinear systems by linearizing them around the current estimate.The Extended Kalman Filter (EKF) is an extension of the classical Kalman Filter, which is designed for linear systems.

In this article, Euler angle formulation is used for Roll and pitch angle dynamics. The dynamics are highly non linear and Kalman Filter can't be as straight forward. There, EKF approach is formulated for MPU6050 and implemented for roll and pitch angles. EKF algorithm linearize the system dynamics using Taylor series expansion about the current state of the system. After linearizing, Jacobian matrices are computed for state transition matrix and observation matrix.

2 Roll and Pitch Angle Dynamics Using Euler Angle's Formulation

MPU 6050 measures outputs acceleration along all three axes (a_x, a_y and a_z) and angular rates about all three axes (p, q and r). According to the Euler angle formulation:

$$\begin{bmatrix} \dot{\phi} \\ \dot{\theta} \\ \dot{\psi} \end{bmatrix} = \begin{bmatrix} 1 & S_\phi T_\theta & C_\phi T_\theta \\ 0 & C_\phi & -S_\phi \\ 0 & \frac{S_\phi}{C_\theta} & \frac{C_\phi}{C_\theta} \end{bmatrix} \begin{bmatrix} p \\ q \\ r \end{bmatrix} \tag{1}$$

In this article, only roll and pitch angles are estimated. Therefore -

$$\begin{aligned} \dot{\phi} &= p + (S_\phi T_\theta)q + (C_\phi T_\theta)r \\ \dot{\theta} &= 0 + (C_\phi)q - (S_\phi)r \end{aligned} \tag{2}$$

Where,
$S(.)$: $sin(.)$, $C(.)$: $cos(.)$, $T(.)$: $tan(.)$
ϕ: Roll angle
θ: Pitch angle
$\dot{\phi}$: Rate of change of roll angle
$\dot{\theta}$: Rate of change of pitch angle
$\dot{\psi}$: Rate of change of yaw angle
p: Roll rate
q: pitch rate
r: Yaw rate

As it can be seen in equation 2 that the dynamic is non linear and simple Kalman filter can't be used.

3 Kalman and Extended Kalman Filter

Algorithm for Kalman filter for linear dynamics is given below -

	Equation	**Name**	**Alternative names in the literature**
Predict	$\hat{\boldsymbol{x}}_{n+1,n} = \boldsymbol{F}\hat{\boldsymbol{x}}_{n,n} + \boldsymbol{G}\boldsymbol{u}_n$	State Extrapolation	Predictor Equation Transition Equation Prediction Equation Dynamic Model State Space Model
	$\boldsymbol{P}_{n+1,n} = \boldsymbol{F}\boldsymbol{P}_{n,n}\boldsymbol{F}^T + \boldsymbol{Q}$	Covariance Extrapolation	Predictor Covariance Equation
Update	$\hat{\boldsymbol{x}}_{n,n} = \hat{\boldsymbol{x}}_{n,n-1} + \boldsymbol{K}_n$ $\times(\boldsymbol{z}_n - \boldsymbol{H}\hat{\boldsymbol{x}}_{n,n-1})$	State Update	Filtering Equation
	$\boldsymbol{P}_{n,n} = (\boldsymbol{I} - \boldsymbol{K}_n\boldsymbol{H})\,\boldsymbol{P}_{n,n-1}$ $\times\,(\boldsymbol{I} - \boldsymbol{K}_n\boldsymbol{H})^T + \boldsymbol{K}_n\boldsymbol{R}_n\boldsymbol{K}_n^T$	Covariance Update	Corrector Equation
	$\boldsymbol{K}_n = \boldsymbol{P}_{n,n-1}\boldsymbol{H}^T$ $\times\left(\boldsymbol{H}\boldsymbol{P}_{n,n-1}\boldsymbol{H}^T + \boldsymbol{R}_n\right)^{-1}$	Kalman Gain	Weight Equation

Figure 1: Algorithm of Kalman Filter

For non linear dynamics (example shown in 2), instead of using State transition matrix and observation matrices, jacobian matrices are used which is obtained from linearization using Taylor series approximation. Algorithm for EKF is given below

	Equation	LKF Equation	EKF Equation
Predict	State Extrapolation	$\hat{\boldsymbol{x}}_{n+1,n} = \boldsymbol{F}\hat{\boldsymbol{x}}_{n,n} + \boldsymbol{G}\boldsymbol{u}_n$	$\hat{\boldsymbol{x}}_{n+1,n} = \boldsymbol{f}(\hat{\boldsymbol{x}}_{n,n}) + \boldsymbol{G}\boldsymbol{u}_n$
	Covariance Extrapolation	$\boldsymbol{P}_{n+1,n} = \boldsymbol{F}\boldsymbol{P}_{n,n}\boldsymbol{F}^T + \boldsymbol{Q}$	$\boldsymbol{P}_{n+1,n} = \frac{\partial \boldsymbol{f}}{\partial \boldsymbol{x}}\boldsymbol{P}_{n,n}\left(\frac{\partial \boldsymbol{f}}{\partial \boldsymbol{x}}\right)^T + \boldsymbol{Q}$
Update	State Update	$\hat{\boldsymbol{x}}_{n,n} = \hat{\boldsymbol{x}}_{n,n-1} + \boldsymbol{K}_n\left(\boldsymbol{z}_n - \boldsymbol{H}\hat{\boldsymbol{x}}_{n,n-1}\right)$	$\hat{\boldsymbol{x}}_{n,n} = \hat{\boldsymbol{x}}_{n,n-1} + \boldsymbol{K}_n\left(\boldsymbol{z}_n - \boldsymbol{h}(\hat{\boldsymbol{x}}_{n,n-1})\right)$
	Covariance Update	$\boldsymbol{P}_{n,n} = \left(\boldsymbol{I} - \boldsymbol{K}_n\boldsymbol{H}\right)\boldsymbol{P}_{n,n-1} \times \left(\boldsymbol{I} - \boldsymbol{K}_n\boldsymbol{H}\right)^T + \boldsymbol{K}_n\boldsymbol{R}_n\boldsymbol{K}_n^T$	$\boldsymbol{P}_{n,n} = \left(\boldsymbol{I} - \boldsymbol{K}_n\frac{\partial \boldsymbol{h}}{\partial \boldsymbol{x}}\right)\boldsymbol{P}_{n,n-1} \times \left(\boldsymbol{I} - \boldsymbol{K}_n\frac{\partial \boldsymbol{h}}{\partial \boldsymbol{x}}\right)^T + \boldsymbol{K}_n\boldsymbol{R}_n\boldsymbol{K}_n^T$
	Kalman Gain	$\boldsymbol{K}_n = \boldsymbol{P}_{n,n-1}\boldsymbol{H}^T \times \left(\boldsymbol{H}\boldsymbol{P}_{n,n-1}\boldsymbol{H}^T + \boldsymbol{R}_n\right)^{-1}$	$\boldsymbol{K}_n = \boldsymbol{P}_{n,n-1}\left(\frac{\partial \boldsymbol{h}}{\partial \boldsymbol{x}}\right)^T \times \left(\frac{\partial \boldsymbol{h}}{\partial \boldsymbol{x}}\boldsymbol{P}_{n,n-1}\left(\frac{\partial \boldsymbol{h}}{\partial \boldsymbol{x}}\right)^T + \boldsymbol{R}_n\right)^{-1}$

Figure 2: Algorithm of EKF

Here, $f(.)$ is an state transition function and $h(.)$ is an observation function. $\frac{\partial f}{\partial x}$ and $\frac{\partial h}{\partial x}$ are Jacobian of state transition and observation functions, respectively.
$\mathbf{x}^T$ is $[\phi \quad \theta]$.

4 Implementation of EKF in MPU 6050

For IMU sensor, MPU6050 sensor is used along with Teensy4.0. Arduino IDE is used for programming. For roll and pitch angle estimation, state transition function is -

$$f(\mathbf{x}) = \begin{bmatrix} \phi^{t+\Delta t} \\ \theta^{t+\Delta t} \end{bmatrix} = \begin{bmatrix} \phi^t + (\dot{\phi})^t \Delta t \\ \theta^t + (\dot{\theta})^t \Delta t \end{bmatrix} = \begin{bmatrix} \phi^t + \{p + (S_\phi T_\theta)q + (C_\phi T_\theta)r\}^t \Delta t \\ \theta^t + \{(C_\phi)q - (S_\phi)r\}^t \Delta t \end{bmatrix} = \begin{bmatrix} f_1 \\ f_2 \end{bmatrix} \tag{3}$$

Observation model $h(\mathbf{x})$, for MPU6050 is the acceleration obtained in inertial frame using estimated roll and pitch angles.

$$h(\mathbf{x}) = g \begin{bmatrix} -sin(\theta) \\ cos(\theta)sin(\phi) \\ cos(\theta)cos(\phi) \end{bmatrix} = \begin{bmatrix} h_1 \\ h_2 \\ h_3 \end{bmatrix} \tag{4}$$

Jacobian of $f(\mathbf{x})$ and $h(\mathbf{x})$ are given below -

$$\frac{\partial f}{\partial \mathbf{x}} = \begin{bmatrix} \frac{\partial f_1}{\partial \phi} & \frac{\partial f_1}{\partial \theta} \\ \\ \frac{\partial f_2}{\partial \phi} & \frac{\partial f_2}{\partial \theta} \end{bmatrix} = \begin{bmatrix} 1 + (C_\phi q - S_\phi r) T_\theta \Delta t & (S_\phi q + C_\phi r) \frac{\Delta t}{C_\theta^2} \\ \\ -(S_\phi q + C_\phi r) \Delta t & 1 \end{bmatrix} \tag{5}$$

$$\frac{\partial h}{\partial \mathbf{x}} = \begin{bmatrix} \frac{\partial h_1}{\partial \phi} & \frac{\partial h_1}{\partial \theta} \\ \\ \frac{\partial h_2}{\partial \phi} & \frac{\partial h_2}{\partial \theta} \\ \\ \frac{\partial h_3}{\partial \phi} & \frac{\partial h_3}{\partial \theta} \end{bmatrix} = g \begin{bmatrix} 0 & -cos\theta \\ \\ cos\theta\, cos\phi & -sin\theta\, sin\phi \\ \\ -cos\theta\, sin\phi & -sin\theta\, cos\phi \end{bmatrix} \tag{6}$$

Angular rates obtained from MPU6050 contain bias. For estimation roll and pitch, it is essential to compute bias($\mathbf{b}$) and reduce the bias value from the obtained angular rates. Is is assumed that bias is not varying with respect to time i.e. $\dot{\mathbf{b}} = 0$. Modified roll, pitch and yaw rates are given as $\bar{p} = p - b_x$, $\bar{q} = q - b_y$ and $\bar{r} = r - b_z$. The state of the system is modified ans given as $\mathbf{x}^T = [\phi\ \theta\ b_x\ b_y\ b_z]$ and state transition function is given as -

$$f(\mathbf{x}) = \begin{bmatrix} \phi^{t+\Delta t} \\ \theta^{t+\Delta t} \\ b_x^{t+\Delta t} \\ b_y^{t+\Delta t} \\ b_z^{t+\Delta t} \end{bmatrix} = \begin{bmatrix} \phi^t + (\dot{\phi})^t \Delta t \\ \theta^t + (\dot{\theta})^t \Delta t \\ b_x^t \\ b_y^t \\ b_z^t \end{bmatrix} = \begin{bmatrix} \phi^t + \{\bar{p} + (S_\phi T_\theta)\bar{q} + (C_\phi T_\theta)\bar{r}\}^t \Delta t \\ \theta^t + \{(C_\phi)\bar{q} - (S_\phi)\bar{r}\}^t \Delta t \\ b_x^t \\ b_y^t \\ b_z^t \end{bmatrix} = \begin{bmatrix} f_1 \\ f_2 \\ f_3 \\ f_4 \\ f_5 \end{bmatrix} \tag{7}$$

$$\frac{\partial f}{\partial \mathbf{x}} = \begin{bmatrix} \frac{\partial f_1}{\partial \phi} & \frac{\partial f_1}{\partial \theta} & \frac{\partial f_1}{\partial b_x} & \frac{\partial f_1}{\partial b_y} & \frac{\partial f_1}{\partial b_z} \\ \\ \frac{\partial f_2}{\partial \phi} & \frac{\partial f_2}{\partial \theta} & \frac{\partial f_2}{\partial b_x} & \frac{\partial f_2}{\partial b_y} & \frac{\partial f_2}{\partial b_z} \\ \\ \frac{\partial f_3}{\partial \phi} & \frac{\partial f_3}{\partial \theta} & \frac{\partial f_3}{\partial b_x} & \frac{\partial f_3}{\partial b_y} & \frac{\partial f_3}{\partial b_z} \\ \\ \frac{\partial f_4}{\partial \phi} & \frac{\partial f_4}{\partial \theta} & \frac{\partial f_4}{\partial b_x} & \frac{\partial f_4}{\partial b_y} & \frac{\partial f_4}{\partial b_z} \\ \\ \frac{\partial f_5}{\partial \phi} & \frac{\partial f_5}{\partial \theta} & \frac{\partial f_5}{\partial b_x} & \frac{\partial f_5}{\partial b_y} & \frac{\partial f_5}{\partial b_z} \end{bmatrix} \tag{8}$$

$$\frac{\partial f}{\partial \mathbf{x}} = \begin{bmatrix} 1+(C_\phi \bar{q} - S_\phi \bar{r})T_\theta \Delta t & (S_\phi \bar{q} + C_\phi \bar{r})\frac{\Delta t}{C_\theta^2} & -\Delta t & (-S_\phi T_\theta)\Delta t & (-C_\phi T_\theta)\Delta t \\ -(S_\phi \bar{q} + C_\phi \bar{r})\Delta t & 1 & 0 & -C_\phi \Delta t & S_\phi \Delta t \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 \end{bmatrix} \tag{9}$$

Observation matrix and its Jacobian remains same as equation-4 & 6. Process noise matrix in computes as -

$$\begin{aligned} Q &= W \Sigma W^T \\ W &= \frac{\partial f(\mathbf{x})}{\partial \omega} \end{aligned} \tag{10}$$

Here, $\omega^T = [p \ q \ r]$.

$$W = \begin{bmatrix} \Delta t & S_\phi T_\theta \Delta t & C_\phi T_\theta \Delta t \\ 0 & C_\phi \Delta t & -S_\phi \Delta t \\ 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix} \tag{11}$$

and Σ is angular rates error covariance matrix.

$$\Sigma = \begin{bmatrix} \sigma_p^2 & 0 & 0 \\ 0 & \sigma_q^2 & 0 \\ 0 & 0 & \sigma_r^2 \end{bmatrix} = \begin{bmatrix} 0.1 & 0 & 0 \\ 0 & 0.4 & 0 \\ 0 & 0 & 0.3 \end{bmatrix} \tag{12}$$

Covariance matrix of measurement error (in accelerometer) R is given below -

$$R = \begin{bmatrix} \sigma_{a_x}^2 & 0 & 0 \\ 0 & \sigma_{a_y}^2 & 0 \\ 0 & 0 & \sigma_{a_z}^2 \end{bmatrix} = \begin{bmatrix} 0.02 & 0 & 0 \\ 0 & 0.02 & 0 \\ 0 & 0 & 0.02 \end{bmatrix} \tag{13}$$

Measurement vector (z):

$$z = \begin{bmatrix} a_x \\ a_y \\ a_z \end{bmatrix} \tag{14}$$

For initialization, at t = 0, $\mathbf{x^T} = [0, 0, 0, 0, 0]$ and process covariance matrix (P) is -

$$P = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 \end{bmatrix} \tag{15}$$

5 Results

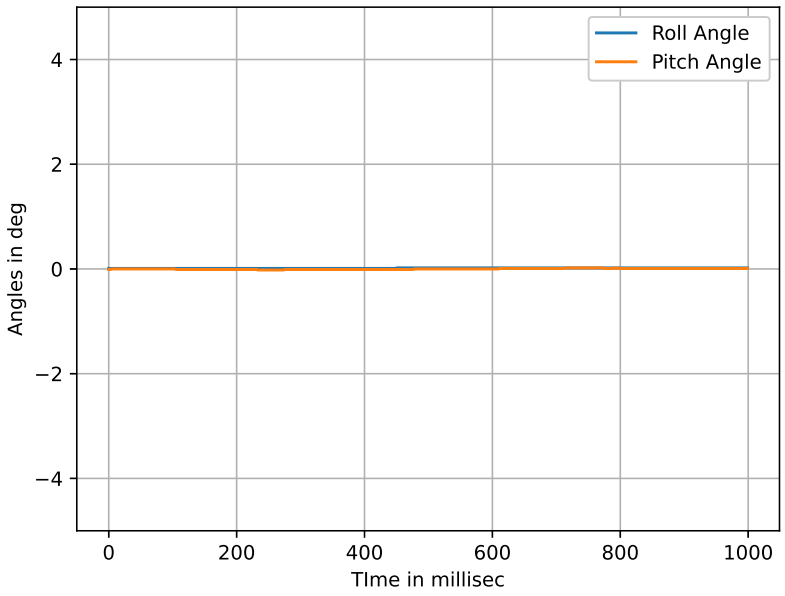

Figure 3: Roll and Pitch Angles at stationary Conditions

6 Arduino Code for EKF

```
#include <Wire.h>
#include <Adafruit_MPU6050.h>
#include <Adafruit_Sensor.h>
#include <BasicLinearAlgebra.h>

Adafruit_MPU6050 mpusensor;

//Gyro and Accelerometer Variables
const int MPU = 0x68;
float offsetAccX, offsetAccY, offsetAccZ;
float AccX, AccY, AccZ;
float rollRate; float pitchRate; float yawRate;
float offfsetRollRate = 0.0; float offfsetPitchRate = 0.0;
float offfsetYawRate = 0.0;
float meaRoll, meaPitch;
float rollAngle = 0.0; float pitchAngle = 0.0;
float accX[6] = {0, 0, 0, 0, 0, 0};
float accY[6] = {0, 0, 0, 0, 0, 0};
float accZ[6] = {0, 0, 0, 0, 0, 0};
float gyroX[6] = {0, 0, 0, 0, 0, 0};
float gyroY[6] = {0, 0, 0, 0, 0, 0};
float gyroZ[6] = {0, 0, 0, 0, 0, 0};
float a[2] = {1.64927209, -0.70219636};
float b[3] = {0.01323107, 0.02646213, 0.01323107};
float Ts = 4; float dt = Ts/1000;
unsigned long int start = 0;
float g = 9.81;
// 2D Kalman Filter
BLA::Matrix<5, 5>F; BLA::Matrix<5, 5>P;
BLA::Matrix<3, 3>Q; BLA::Matrix<5, 1>S;
BLA::Matrix<5, 5>I; BLA::Matrix<5, 3>K;
BLA::Matrix<3, 3>R; BLA::Matrix<3, 3>L;
BLA::Matrix<3, 1>M; BLA::Matrix<3, 1>hx;
BLA::Matrix<3, 5>H; BLA::Matrix<5, 3>W;

float cp, sp, ct, st, tt;
float p, q, r;
float bx = 0; float by = 0; float bz = 0;


float* butterworth(float value, float arr[]) {
  arr[3] = value;
  arr[0] = a[0] * arr[1] + a[1] * arr[2] + b[0] * arr[3] + b[
1] * arr[4] + b[2] * arr[5];
  arr[2] = arr[1];
  arr[1] = arr[0];
  arr[5] = arr[4];
  arr[4] = arr[3];
  return arr;
}


void gyro_signals(void){
  Wire.beginTransmission(MPU);
  Wire.write(0x1A);
```

```
        Wire.endTransmission();
55
        // Range of MPU Accelerometer
57      Wire.beginTransmission(MPU);
        Wire.write(0x1C);
59      Wire.write(0x10);
        Wire.endTransmission();
61
        //Getting Accelerations
63      Wire.beginTransmission(MPU);
        Wire.write(0x3B);
65      Wire.endTransmission();
        Wire.requestFrom(MPU, 6);
67
        int16_t AccXLSB = Wire.read()<<8|Wire.read();
69      int16_t AccYLSB = Wire.read()<<8|Wire.read();
        int16_t AccZLSB = Wire.read()<<8|Wire.read();
71
        butterworth((float)AccXLSB/4096 - offsetAccX, accX);
73      butterworth((float)AccYLSB/4096 - offsetAccY, accY);
        butterworth((float)AccZLSB/4096 + offsetAccZ, accZ);
75      AccX = accX[0]; AccY = accY[0]; AccZ = accZ[0];

77      //Range of GyroRates
        Wire.beginTransmission(MPU);
79      Wire.write(0x1B);
        Wire.write(0x8);
81      Wire.endTransmission();

83      //Getting GyroRates
        Wire.beginTransmission(MPU);
85      Wire.write(0x43);
        Wire.endTransmission();
87      Wire.requestFrom(MPU, 6);

89      int16_t GyroX = Wire.read()<<8|Wire.read();
        int16_t GyroY = Wire.read()<<8|Wire.read();
91      int16_t GyroZ = Wire.read()<<8|Wire.read();

93      butterworth((float)GyroX/65.5 - offfsetRollRate, gyroX);
        butterworth((float)GyroY/65.5 - offfsetPitchRate, gyroY);
95      butterworth((float)GyroZ/65.5 - offfsetYawRate, gyroZ);
        rollRate = gyroX[0]; pitchRate = gyroY[0]; yawRate = gyroZ[
    0];
97      p = rollRate - bx; q = pitchRate - by; r = yawRate - bz;
        cp = cos(rollAngle/57.2957); sp = sin(rollAngle/57.2957);
99      ct = cos(pitchAngle/57.2957); st = sin(pitchAngle/57.2957);
        tt = tan(pitchAngle/57.2957);
101     S(0,0) += dt*(p + q*sp*tt + r*cp*tt);
        S(1,0) += dt*(q*cp - r*sp);
103     rollAngle = S(0,0);
        pitchAngle = S(1,0);
105    W = {dt, sp*tt*dt, cp*tt*dt, 0, cp*dt, -sp*dt, 0, 0, 0, 0,
    0, 0, 0, 0, 0};
        F = {1+(q*cp-r*sp)*tt*dt, dt*(q*sp+r*cp)/(ct*ct), -dt, -sp*
    tt*dt, -cp*tt*dt,
107       -(q*sp+r*cp)*dt, 1, 0, -cp*dt, sp*dt,
```

```
          0, 0, 1, 0, 0,
          0, 0, 0, 1, 0,
          0, 0, 0, 0, 1};
      H = {0, -ct, 0, 0, 0, ct*cp, -st*sp, 0, 0, 0, -ct*sp, -st*cp, 0, 0, 0};
      P = F*P*~F + W*Q*~W;
      L = H*P*~H + R;
      K = P*~H*Inverse(L);
      M = {AccX*g, AccY*g, AccZ*g};
      hx = {-g*st, g*ct*sp, g*ct*cp};
      S = S + K*(M - hx);
      rollAngle = S(0,0);
      pitchAngle = S(1,0);
      P = (I - K*H)*P;
    }


    void setup() {
      // put your setup code here, to run once:
      pinMode(13, OUTPUT);
      digitalWrite(13, LOW);
      Serial.begin(9600);

      // Try to initialize!
      if (!mpusensor.begin()) {
        Serial.println("Failed to find MPU6050 chip");
        while (1) {
          delay(10);
        }
      }
      Serial.println("MPU6050 Found!");
      Wire.setClock(400000);
      Wire.begin();
      delay(250);
      Wire.beginTransmission(MPU);
      Wire.write(0x6B);
      Wire.write(0x00);
      Wire.endTransmission();

      digitalWrite(13, HIGH);
      Serial.println("Calibration is going on.....");
      int N = 2000;
      float sumRollRate = 0; float sumPitchRate = 0; float sumYawRate = 0;
      float sumAccX = 0; float sumAccY = 0; float sumAccZ = 0;
      for (int i = 1; i < N; i++){
        gyro_signals();
        sumRollRate += rollRate; sumPitchRate += pitchRate; sumYawRate += yawRate;
        sumAccX += AccX; sumAccY += AccY; sumAccZ += AccZ;
      }
      offfsetRollRate = sumRollRate/N; offfsetPitchRate = sumPitchRate/N; offfsetYawRate = sumYawRate/N;
      offsetAccX = sumAccX/N;
      offsetAccY = sumAccY/N;
      offsetAccZ = 1 - sumAccZ/N;
      Serial.println("Calibration done!!!");
```

```
161
        digitalWrite(13, LOW);
163     rollAngle = 0; pitchAngle = 0;
        delay(1000);
165     digitalWrite(13, HIGH);

167     cp = cos(rollAngle/57.2957); sp = sin(rollAngle/57.2957);
        ct = cos(pitchAngle/57.2957); st = sin(pitchAngle/57.2957);
169     tt = tan(pitchAngle/57.2957);
        p = rollRate - bx; q = pitchRate - by; r = yawRate - bz;
171     F = {1+(q*cp*tt-r*sp*tt)*dt, (q*sp+r*cp)*dt/(cp*cp), -(q*sp
    +r*cp)*dt, 1};
        H = {0, -g*ct, g*ct*cp, -g*st*sp, -g*ct*sp, -g*st*cp};
173
        Q = {0.1, 0, 0,
175       0, 0.4, 0,
          0, 0, 0.3};
177
        I = {1, 0, 0, 0, 0,
179       0, 1, 0, 0, 0,
          0, 0, 1, 0, 0,
181       0, 0, 0, 1, 0,
          0, 0, 0, 0, 1};
183
        R = {0.02, 0, 0,
185       0, 0.02, 0,
          0, 0 , 0.02};
187
        P = {1, 0, 0, 0, 0,
189       0, 1, 0, 0, 0,
          0, 0, 1, 0, 0,
191       0, 0, 0, 1, 0,
          0, 0, 0, 0, 1};
193
        S = {0, 0, 0, 0, 0};
195   }

197   void loop() {
        // put your main code here, to run repeatedly:
199     start = millis();
        gyro_signals();
201
        Serial.print(rollAngle);
203     Serial.print("\t");
        Serial.println(pitchAngle);
205
        while ((millis() - start) < Ts);
207   }

209
```