# SELF BALANCING ROBOT AND ARTIFICIAL HORIZON USING KALMAN FILTER

Submitted By

ANIRUDH EDPUGANTI (CB.EN.U4AIE20005)
ONTEDDU CHAITANYA REDDY (CB.EN.U4AIE20045)
PILLALAMARRI AKSHAYA(CB.EN.U4AIE20049)

……

For the Completion of

19AIE114 - PRINCIPLE OF MEASUREMENTS AND SENSORS

CSE - AI

$5^{th}$ Aug 2021

## Introduction

We wonder how a hoverboard balances itself because it uses the principle of a self-balancing robot that is two-wheeled and balances itself.

What happens if the pilot doesn't know the information in terms of the aircraft's attitude both in pitch and roll.

To resolve this problem artificial horizon is used

It is referred to as gyros or artificial horizons, operated with a gyroscope that indicates at which level aircraft are oriented to the earth.

## Objective

- To design a self-balancing robot using an MPU 6050 and PID controller.
- To reduce noise in MPU 6050 using Kalman filter in Artificial horizons.

## Hardware and Software Requirements

- Arduino Uno R3
- MPU6050
- L298N Motor Driven
- Arduino
- Processing
- Python

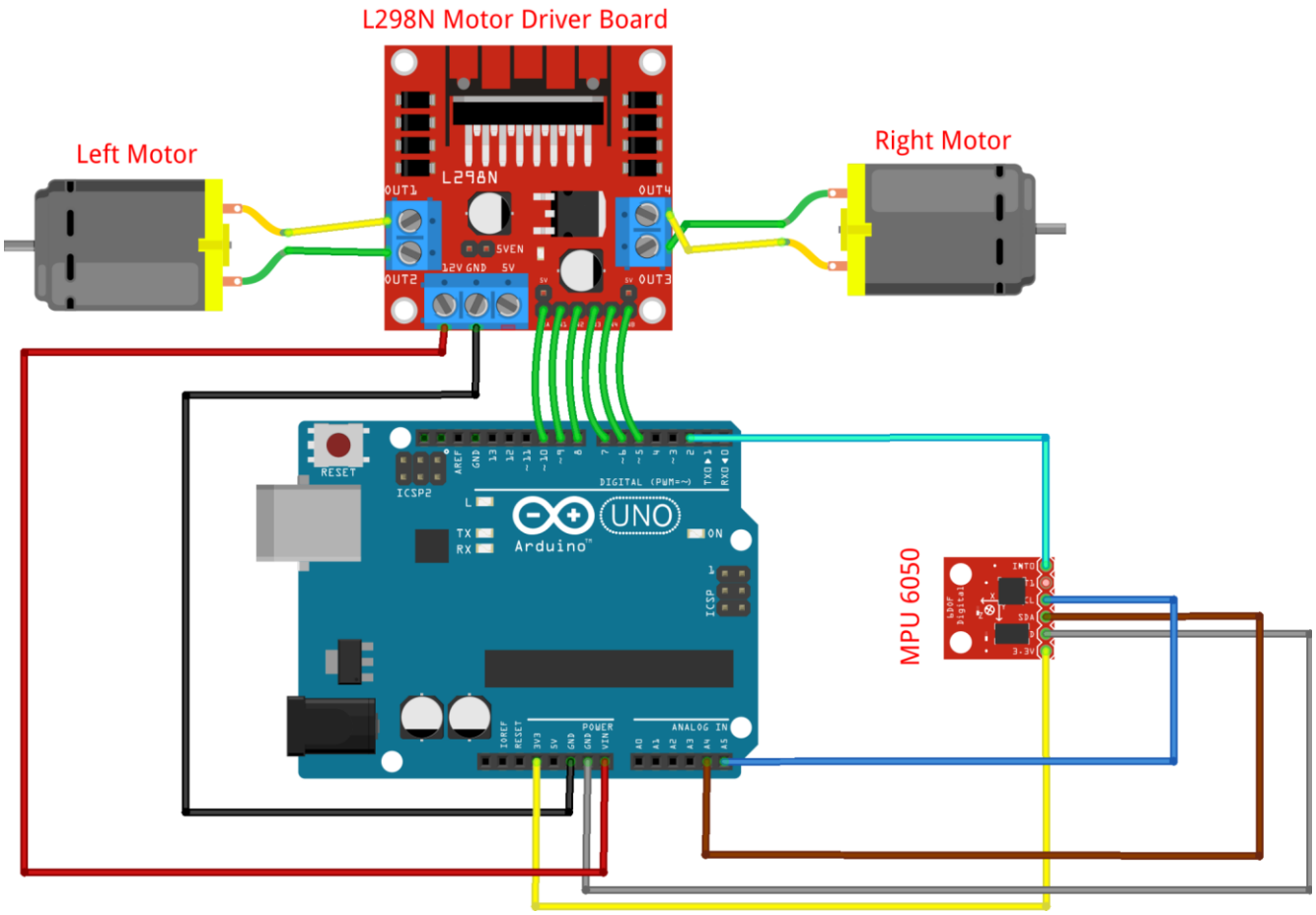
## Tools

- Fritzing (for circuit designing)

## Implementation and Outputs

### SELF BALANCING ROBOT

**CIRCUIT DESIGN:**




**CONNECTIONS:**

| MPU6050 | Arduino Uno R3 |
|---|---|
| VCC | 3.3V |

| | |
|---|---|
| **GND** | **GND** |
| **SCL** | **A5** |
| **SDA** | **A4** |
| **INT** | **2** |

| **L298N Motor Driver** | **Arduino Uno R3** |
|---|---|
| **ENA** | **5** |
| **IN1** | **6** |
| **IN2** | **7** |
| **IN3** | **8** |
| **IN4** | **9** |
| **ENB** | **10** |
| **+12V** | **Vin** |
| **GND** | **GND** |

## PID Controller

Proportional (Kp): Proportional constant is proportional to the error of the state variable. Though Kp tries to minimize the steady-state error, it fails to make the error term zero. Due to steady-state error, Kp alone is not sufficient to nullify the error.

Integrator (Ki): Since the steady-state error is always a constant, integrating a constant always gets added up as the value increases. Sole dependence on this constant can make the error zero but this shoots up rapidly and only gets controlled only after reaching the other side of the target.

Derivative (Kd): Using this constant alone cannot make the error zero, but it aims to flatten the error trajectory, damping the external force which in turn nullifies the overshoot made by the integrator (Ki).

## WORKING LOGIC:

### Calibration of MPU 6050

We place the MPU 6050 sensor in a position parallel to the ground in a balanced position. Then we calibrate the sensor to gain the offset values of the gyroscope and accelerometer. These we take as the reference values of the axes.

### Tuning of PID controller

Although we have many methods to tune, we do manual tuning in this project. We start with fixing a value to Kp which drives the quick response of the robot. Since when we apply a greater external force, if the response is not quick enough, it fails to get back to the balanced position. Hence, we use a relatively high value for Kp. Since the integrator takes care of the oscillation of the robot, i.e quick response, so this should also take value nearer to that of value Kp. But, it overshoots and goes off the target.
To take care of this, we give a relatively small value for Kd and experiment with the combination of values in order to fit the optimal set of PID values.

**Working of Self Balancing Robot**

First, we start with the calibration of MPU 6050. We set the offset values. Then we tune the PID constants in order to optimize the output. Now, we set the speeds of the motors which depend on the amount of weight we put on the robot and also on the symmetrical arrangement of it. We should also set a reference point of the robot at which it is balanced. Although, the ideal reference point is considered to be *180°*. In the real-world scenario, we don't maintain a pin-pointed symmetry hence we experiment by taking a lot of values when the robot is still (oscillates on its own) and take the mean of the values to decide that set or reference point.

Once all the constants and connections are set, we now see the logic of movement. When there is tilt detected by the MPU sensor, it, in turn, gives the reading of the angle by which it is tilted to the Arduino. Now, a signal from Arduino is sent to the Motor driver inorder to change the direction of the motor movement and also controlling its speed.

Even after choosing an optimal set of PID values, setpoint, and offset values we still fail to overcome abrupt external forces which are beyond the scope of normal noise. We can control this kind of situation by using Kalman filters instead of normal complementary filters.

**CODE:**

```
#include <PID_v1.h>
#include <LMotorController.h>
#include "I2Cdev.h"
#include "MPU6050_6Axis_MotionApps20.h"

#if I2CDEV_IMPLEMENTATION == I2CDEV_ARDUINO_WIRE
    #include "Wire.h"
#endif

#define MIN_ABS_SPEED 20

MPU6050 mpu;

// MPU control/status vars
bool dmpReady = false;  // set true if DMP init was successful
uint8_t mpuIntStatus;   // holds actual interrupt status byte from MPU
uint8_t devStatus;      // return status after each device operation (0 =
success, !0 = error)
uint16_t packetSize;    // expected DMP packet size (default is 42 bytes)
uint16_t fifoCount;     // count of all bytes currently in FIFO
uint8_t fifoBuffer[64]; // FIFO storage buffer
```

```
// orientation/motion vars
Quaternion q;           // [w, x, y, z]         quaternion container
VectorFloat gravity;    // [x, y, z]            gravity vector
float ypr[3];           // [yaw, pitch, roll]   yaw/pitch/roll container and
gravity vector

//PID
double originalSetpoint = 180;
double setpoint = originalSetpoint;
double input, output;
double Kp = 50;
double Kd = 1.4;
double Ki = 60;
PID pid(&input, &output, &setpoint, Kp, Ki, Kd, DIRECT);

double motorSpeedFactorLeft = 0.5;
double motorSpeedFactorRight = 0.5;
//MOTOR CONTROLLER
int ENA = 5;
int IN1 = 6;
int IN2 = 7;
int IN3 = 8;
int IN4 = 9;
int ENB = 10;
LMotorController motorController(ENA, IN1, IN2, ENB, IN3, IN4,
motorSpeedFactorLeft, motorSpeedFactorRight);

//timers
long time1Hz = 0;
long time5Hz = 0;

volatile bool mpuInterrupt = false;     // indicates whether MPU interrupt pin
has gone high
void dmpDataReady()
{
    mpuInterrupt = true;
}


void setup()
{
```

```
    // join I2C bus (I2Cdev library doesn't do this automatically)
    #if I2CDEV_IMPLEMENTATION == I2CDEV_ARDUINO_WIRE
        Wire.begin();
        TWBR = 24; // 400kHz I2C clock (200kHz if CPU is 8MHz)
    #elif I2CDEV_IMPLEMENTATION == I2CDEV_BUILTIN_FASTWIRE
        Fastwire::setup(400, true);
    #endif

    // initialize serial communication
    // (115200 chosen because it is required for Teapot Demo output, but it's
    // really up to you depending on your project)
    Serial.begin(115200);
    while (!Serial); // wait for Leonardo enumeration, others continue
immediately

    // initialize device
    Serial.println(F("Initializing I2C devices..."));
    mpu.initialize();

    // verify connection
    Serial.println(F("Testing device connections..."));
    Serial.println(mpu.testConnection() ? F("MPU6050 connection successful") :
F("MPU6050 connection failed"));

    // load and configure the DMP
    Serial.println(F("Initializing DMP..."));
    devStatus = mpu.dmpInitialize();

    // supply your own gyro offsets here, scaled for min sensitivity
    mpu.setXGyroOffset(220);
    mpu.setYGyroOffset(76);
    mpu.setZGyroOffset(-85);
    mpu.setZAccelOffset(1788); // 1688 factory default for the test chip

    // make sure it worked (returns 0 if so)
    if (devStatus == 0)
    {
        // turn on the DMP, now that it's ready
        Serial.println(F("Enabling DMP..."));
        mpu.setDMPEnabled(true);

        // enable Arduino interrupt detection
```

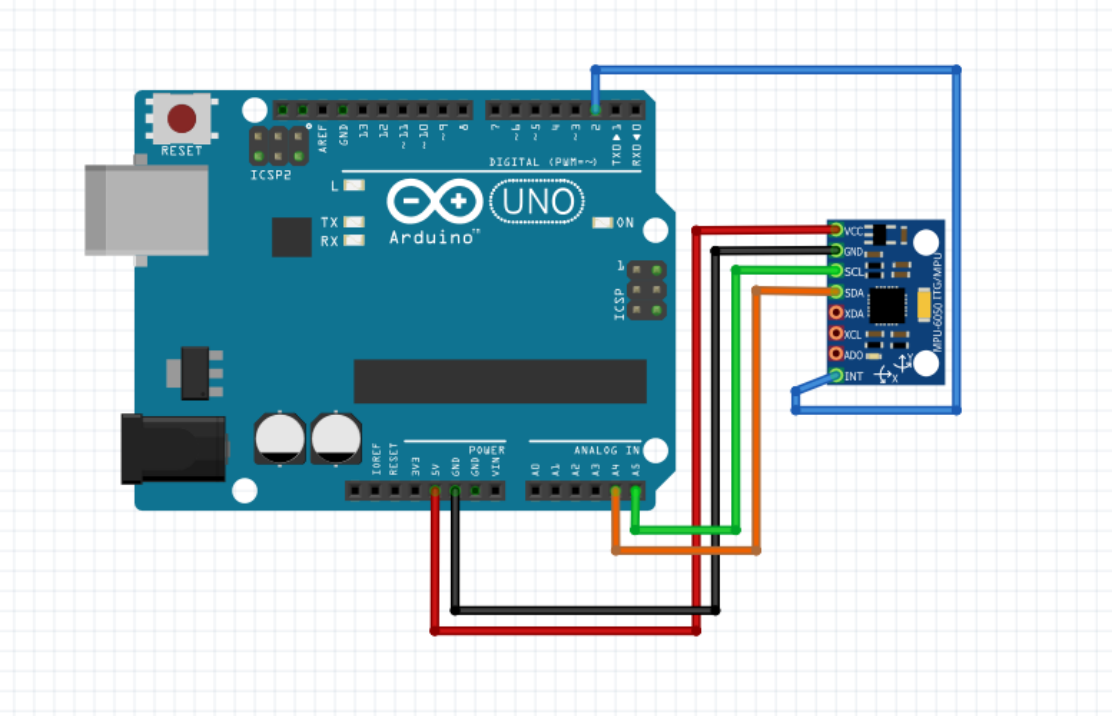
```
        Serial.println(F("Enabling interrupt detection (Arduino external
interrupt 0)..."));
        attachInterrupt(0, dmpDataReady, RISING);
        mpuIntStatus = mpu.getIntStatus();

        // set our DMP Ready flag so the main loop() function knows it's okay to
use it
        Serial.println(F("DMP ready! Waiting for first interrupt..."));
        dmpReady = true;

        // get expected DMP packet size for later comparison
        packetSize = mpu.dmpGetFIFOPacketSize();

        //setup PID

        pid.SetMode(AUTOMATIC);
        pid.SetSampleTime(10);
        pid.SetOutputLimits(-255, 255);
    }
    else
    {
        // ERROR!
        // 1 = initial memory load failed
        // 2 = DMP configuration updates failed
        // (if it's going to break, usually the code will be 1)
        Serial.print(F("DMP Initialization failed (code "));
        Serial.print(devStatus);
        Serial.println(F(")"));
    }
}


void loop()
{
    // if programming failed, don't try to do anything
    if (!dmpReady) return;

    // wait for MPU interrupt or extra packet(s) available
    while (!mpuInterrupt && fifoCount < packetSize)
    {
        //no mpu data - performing PID calculations and output to motors
```

```
        pid.Compute();
        motorController.move(output, MIN_ABS_SPEED);

    }

    // reset interrupt flag and get INT_STATUS byte
    mpuInterrupt = false;
    mpuIntStatus = mpu.getIntStatus();

    // get current FIFO count
    fifoCount = mpu.getFIFOCount();

    // check for overflow (this should never happen unless our code is too
inefficient)
    if ((mpuIntStatus & 0x10) || fifoCount == 1024)
    {
        // reset so we can continue cleanly
        mpu.resetFIFO();
        Serial.println(F("FIFO overflow!"));

    // otherwise, check for DMP data ready interrupt (this should happen
frequently)
    }
    else if (mpuIntStatus & 0x02)
    {
        // wait for correct available data length, should be a VERY short wait
        while (fifoCount < packetSize) fifoCount = mpu.getFIFOCount();

        // read a packet from FIFO
        mpu.getFIFOBytes(fifoBuffer, packetSize);

        // track FIFO count here in case there is > 1 packet available
        // (this lets us immediately read more without waiting for an interrupt)
        fifoCount -= packetSize;

        mpu.dmpGetQuaternion(&q, fifoBuffer);
        mpu.dmpGetGravity(&gravity, &q);
        mpu.dmpGetYawPitchRoll(ypr, &q, &gravity);
        #if LOG_INPUT
            Serial.print("ypr\t");
            Serial.print(ypr[0] * 180/M_PI);
            Serial.print("\t");
```

```
            Serial.print(ypr[1] * 180/M_PI);
            Serial.print("\t");
            Serial.println(ypr[2] * 180/M_PI);
        #endif
        input = ypr[1] * 180/M_PI + 180;
    }

        // track to get the values of mpu
        Serial.println(input);
}
```

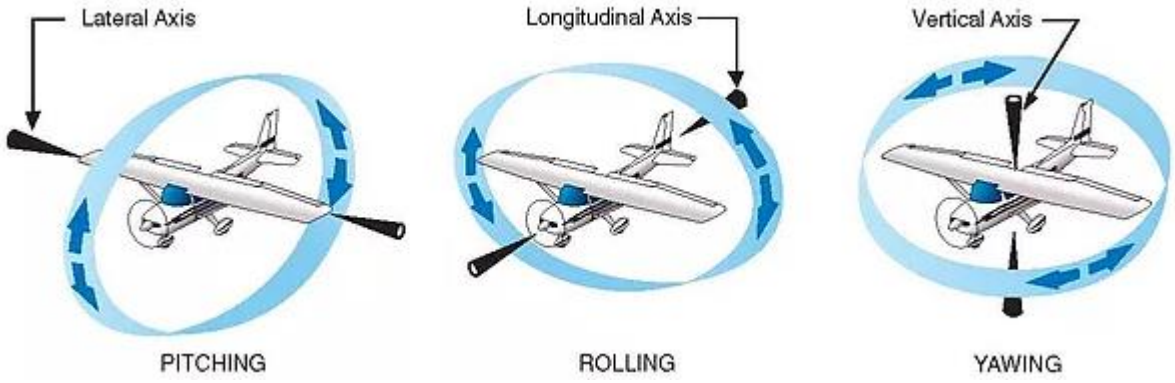
# ARTIFICIAL HORIZON

## CIRCUIT DESIGN




## WORKING LOGIC

Rotating the gyroscope around the longitudinal axis indicates the degree of bank or roll, whereas the lateral axis indicates pitch — nose up, nose down or level. The gyroscope is gimbaled at a lateral axis for pitch attitude and a longitudinal axis for roll. These values are given by MPU, which is extended to Processing in this project to show an Animated movement.



Axis of aeroplane

**Kalman Filter**

Basically, the Kalman filter is used to predict the unknown variables with a minimal amount of information in hand and producing the most appropriate and optimal solution. However, this assumes the information to be a Gaussian distribution and hence is known as the Linear Quadratic Estimator.

How does the Kalman Filter work?

The Kalman Filter is usually used to help predict the readings using its previous predictions and current measured value. And then it checks the error between the prediction and the measured value and makes itself better for the next prediction.

Why do we use a Kalman filter?

- Easy to formulate and implement with basic understanding.
- It will give the best results due to optimality and structure.
- Uses a minimal piece of information.

**Intuitive Understanding**

Let's consider the measured values at time t to be represented by $X_t$and the error due to the sensor is $E_{mea}$. Predicted values at time t are represented by $\widehat{X_t}$ and error in the predictions be represented by $E_{est}$. Based on the previous state estimate and current measured value we will predict the next state estimate. Thus in view of Linear Algebra, the current state estimate can be represented as a linear combination of the previous state estimate and measured value.

The equation can be represented as

$$\hat{X}_{t+1} = \alpha \times \hat{X}_t + \beta \times X_t$$

When error in prediction increases we should rely more on the measured values and similarly when an error in measured value is more then we will be more on the predictions. For that, we use the constants alpha and beta. We need to configure the alpha and beta based upon the error in the measurements and error in the estimates such that when alpha increases beta decreases and when beta decreases alpha increases. For that, we use the basic weighting theorem.

We know that

$$\frac{\mathrm{E_{mea}}}{\mathrm{E_{mea}} + \mathrm{E_{est}}} + \frac{\mathrm{E_{est}}}{\mathrm{E_{mea}} + \mathrm{E_{est}}} = 1$$

and also in this equation, as one part increases the other part decreases to keep their sum constant. This is somewhat similar to that of the requirement in the above equation. So now we can say that

$$\alpha = \frac{E_{mea}}{E_{mea} + E_{est}}$$

$$\beta = \frac{E_{est}}{E_{mea} + E_{est}}$$

***β*** is defined as Kalman Gain.

Rearranging the above equation we get

$$\hat{X}_{t+1} = \left(1 - \frac{E_{est}}{E_{mea} + E_{est}}\right) \times \hat{X}_t + \frac{E_{est}}{E_{mea} + E_{est}} \times X_t$$

$$\hat{X}_{t+1} = \hat{X}_t + \frac{E_{est}}{E_{mea} + E_{est}} \times \left(X_t - \hat{X}_t\right)$$

$$\hat{X}_{t+1} = \hat{X}_t + \beta \times \left(X_t - \hat{X}_t\right)$$

This obtained equation is the KALMAN FILTER MODEL.

**CODE**

```
#include<Wire.h> // wire library
#include<SimpleKalmanFilter.h>

const int MPU_addr=0x68; // MPU address

int16_t AcX,AcY,AcZ,Tmp,GyX,GyY,GyZ; // 16 bit data array

int minVal=265;
int maxVal=402;

SimpleKalmanFilter kfx = SimpleKalmanFilter(0.16, 0.4, 0.01);
SimpleKalmanFilter kfy = SimpleKalmanFilter(0.16, 0.4, 0.01);
SimpleKalmanFilter kfz = SimpleKalmanFilter(0.16, 0.4, 0.01);
```

```
float X_est, Y_est, Z_est;
double x; double y; double z;

void setup() {
Serial.begin(9600);
Wire.begin();
Wire.beginTransmission(MPU_addr);
Wire.write(0x6B);
Wire.write(0);
Wire.endTransmission(true);

}

void loop() {
Wire.beginTransmission(MPU_addr);
Wire.write(0x3B);
Wire.endTransmission(false);
Wire.requestFrom(MPU_addr,14,true);
AcX=Wire.read()<<8|Wire.read();
AcY=Wire.read()<<8|Wire.read();
AcZ=Wire.read()<<8|Wire.read();

 int xAng = map(AcX,minVal,maxVal,-90,90);
int yAng = map(AcY,minVal,maxVal,-90,90);
int zAng = map(AcZ,minVal,maxVal,-90,90);



x= RAD_TO_DEG * (atan2(-yAng, -zAng)+PI);
y= RAD_TO_DEG * (atan2(-xAng, -zAng)+PI);
z= RAD_TO_DEG * (atan2(-yAng, -xAng)+PI);


X_est = kfx.updateEstimate(x);
Y_est = kfy.updateEstimate(y);
Z_est = kfz.updateEstimate(z);
```

```
// Uncomment this part for results with Kalman Filter

//Serial.print(X_est);
//Serial.print(" ");
//Serial.print(Y_est);
//Serial.print(" ");
//Serial.print(Z_est);
//Serial.print("\n");
//

// Uncomment this part for results without Kalman filter
//Serial.print(x);
//Serial.print(" ");
//Serial.print(y);
//Serial.print(" ");
//Serial.print(z);
//Serial.print("\n");

 }
```

## CONCLUSION

We choose these projects to explore the ability and accuracy of sensors and also to show the use of these sensors in the real world.

This self-balancing robot is much essential in these times like Covid which could serve all the patients without any threat to life. This project can be advanced further with the addition of extra sensors and make it more reliable working robots which are very essential in today's world.

The main idea of Artificial Horizon was to be able to show the working of the MPU 6050 sensor and its accuracy which was further taken up into a self-balancing robot. Apart from this, when natural horizons don't work, this is very essential for pilots to know the rate of tilt of the aeroplane.

After embedding the Kalman Filter Model to this, its results were more promising showing minimal errors in the sensor output.

On a final note, this project was very useful not only to explore new concepts but also to show the real-world applications of what we actually learn.