ROBOFEST 4.0

TWO WHEEL SELF BALANCING ROBOT

Submitted By

INDIAN INSTITUTE OF TECHNOLOGY, BHUBANESWAR

2024

Contents:

1. Steps in the working of Robot

A. Description of Robot

A two-wheeled self-balancing robot operates on the principle of an inverted pendulum, using precise control to maintain an upright position. Its center of mass can shift away from the vertical due to uneven weight distribution, making it dynamically stable yet statically unstable. Built with a lightweight frame 3D-printed from durable PLA material, our robot features two NEMA 17 stepper motors that deliver smooth and stable movement. The control system, powered by an Arduino Nano, employs a PID control mechanism to maintain balance.
The robot is engineered to maintain its balance using a combination of elements involving sensors and a control algorithm. The detailed logical steps involved in the working of the robot have been mentioned below:

Initialization and Setup
To initialize the robot, start it from a standstill position, either lying down or standing upright with minimal vibrations. Upon powering up, the onboard MPU 6050 sensor begins calibration, indicated by a blinking LED on the Arduino Nano. If the robot is lying down, gently lift it to a vertical position once the calibration is complete. The robot will then automatically engage its balancing mechanism, using the calibrated sensor data to maintain an upright stance. After initialization, users can place additional loads or weights on top of the robot as needed, and it will adjust to maintain balance accordingly.

Data Acquisition

The microcontroller reads real-time data from the MPU 6050, which includes measurements from its built-in gyroscope and accelerometer. Using the Wire library, the microcontroller communicates with the MPU 6050 over the I2C protocol, ensuring quick data transmission while minimizing the number of required connections. The raw data obtained from the MPU 6050 is processed by the microcontroller, which performs calculations to derive the angular velocity and tilt angle. To achieve a more accurate and stable measurement of the tilt angle, a complementary filter is employed. This filter combines data from both the gyroscope and the accelerometer, leveraging the strengths of each sensor. The accelerometer helps correct long-term drift in the gyroscope data, while the gyroscope smooths out the noisy, short-term variations in the accelerometer readings. Together, they provide a more precise and stable tilt angle estimation, essential for effective balancing.

Error Calculation & Control Algorithm

In the control system, the error is determined by comparing the desired upright position, calibrated as 0 degrees, with the actual tilt angle measured by the sensors. This difference, known as the positional error, indicates how much the robot deviates from the vertical alignment. Additionally, the system calculates the rate of change of this error, which helps predict the robot's future movement trajectory and assess how quickly the tilt is changing.

The PID (Proportional-Integral-Derivative) controller uses this error to dynamically adjust the motor outputs and maintain balance. The Proportional component of the PID responds to the magnitude of the error, applying a corrective force that is directly proportional to the deviation from the upright position. The Integral component addresses accumulated errors over time, correcting small, persistent biases that could cause the robot to drift. More formally, it takes care of the steady state error. Lastly, the Derivative component considers the rate of change of the error, allowing the system to anticipate future deviations and make smoother, more stable adjustments.

Motor Control

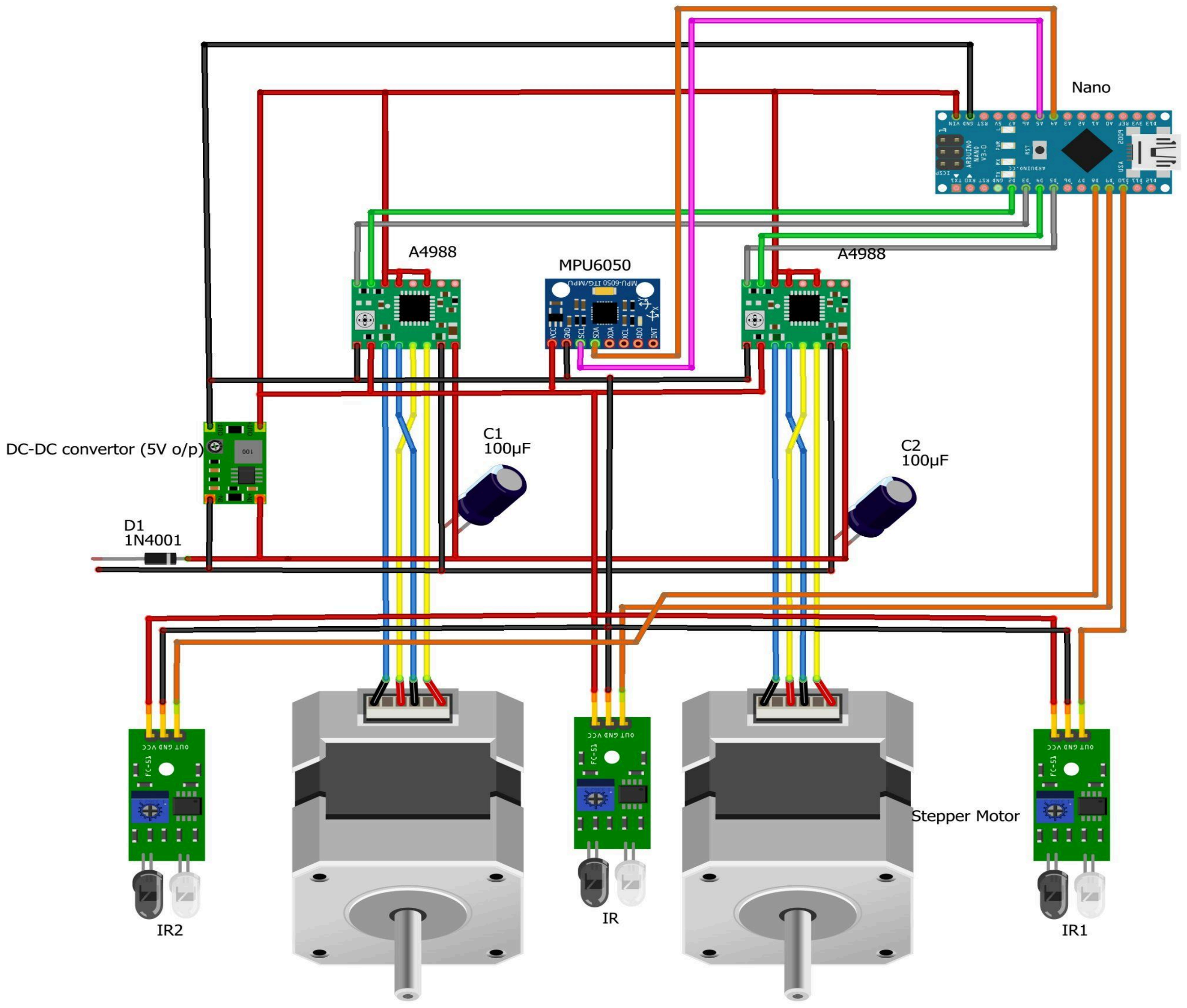
The PID controller's output signals are sent to the motor driver, which adjusts the speed and direction of the stepper motors to maintain balance. Pulse Width Modulation (PWM) signals control the effective power supplied to the motors, allowing precise acceleration or deceleration. The system utilizes an Interrupt Service Routine (ISR) that executes every 20 microseconds, generating pulses to control motor speed and direction. This high-frequency, real-time pulse generation ensures smooth and responsive motor control, enabling the robot to quickly react to changes in tilt and maintain an upright position. The ISR handles pulse timing and motor direction switching, ensuring that corrective actions are applied accurately and consistently.

Feedback Loop

Errors are recalculated continuously and the updated values are sent to the control algorithm in real time to keep adjusting the motor movements accordingly. By continuously calculating and feeding the error into the PID controller, the robot can dynamically balance itself, ensuring real-time corrections to maintain its upright stance.

B. Electronics Description

The MPU6050 detects the tilt of the robot and sends this data to the Arduino Nano, which processes it to determine how to adjust the motors to keep the robot balanced. The Arduino Nano sends direction and step signals to the A4988 drivers, which then control the movement of the stepper motors. The motors adjust their movement to compensate for any imbalance detected by the MPU6050. The IR sensors provide additional functionality for line following. The DC-DC converter ensures that all components are powered correctly, and capacitors are used to stabilize the voltage supply for the motors.



The above figure shows the circuitry employed to get the bot running.

C. Features

Payload Capacity:

Quite conveniently and without spilling anything, the robot is expected to be able to carry a payload effectively owing to the good design and sturdy construction of the frame. This battery case almost weighs 1200 grams with a well-balanced design that enables it to carry different types of objects comfortably. Such a feature comes in handy in the delivery and material handling applications where moving about with a load and still managing to keep in an upright position is a requirement.

Line Following with IR Sensors:

Infrared (IR) sensors are particularly important in line-following robots, as they allow the robot to detect and follow marked paths or lines. These sensors enable the robot to automatically adjust its movement based on the detected lines. This capability is especially useful in applications where the robot needs to operate in confined areas, such as storage facilities or assembly lines, where robots perform tasks as part of a larger process.

Autonomous Self-Balancing:

Based on the inverted pendulum structure of the robot, self-balancing can be achieved automatically with the assistance of the user being limited. The MPU 6050, which is one of the sensors installed in the robot system, detects the activities such as angle tilt and the speed at which the tilt changes or angular velocity and feeds those to the PID controller.

2. List of Components used:

All our components except the wheels were 3D printed to strike the balance between the payload carrying capacity and the weight of the robot itself.

A. Mechanical Components

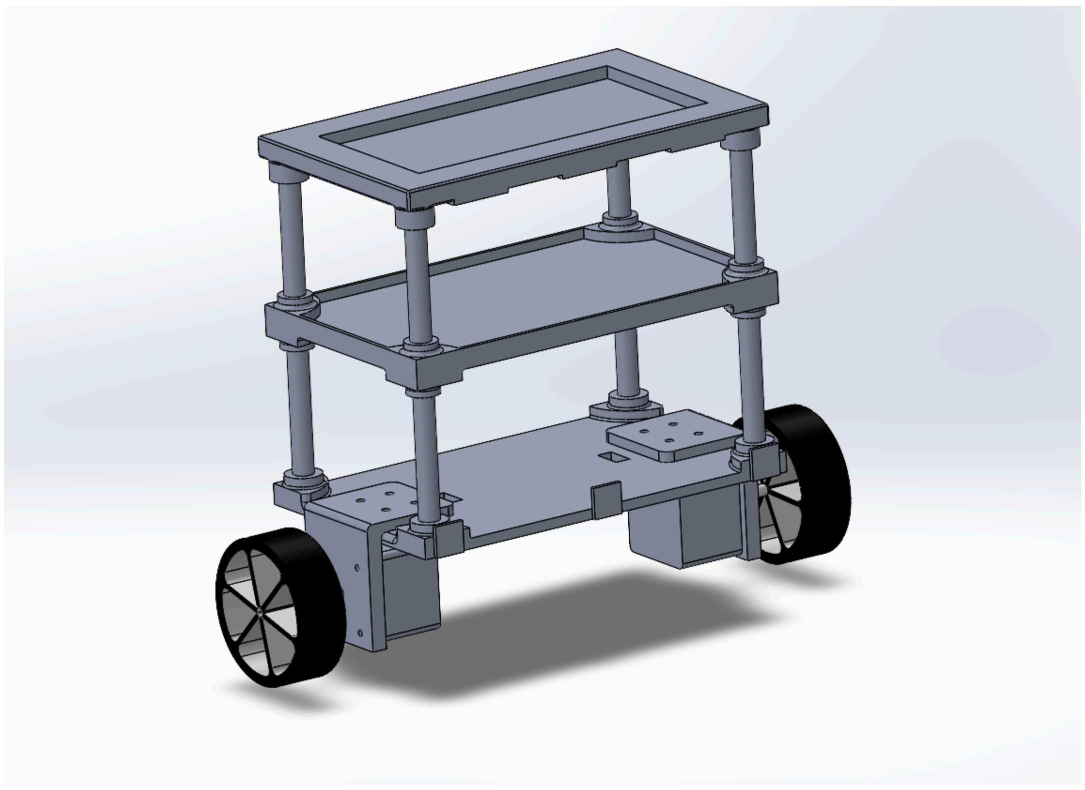


Fig 1 Complete Assembly

The above figure shows the complete detailed assembly of our robot.

Plates(x3)

Our model consists of three plates.

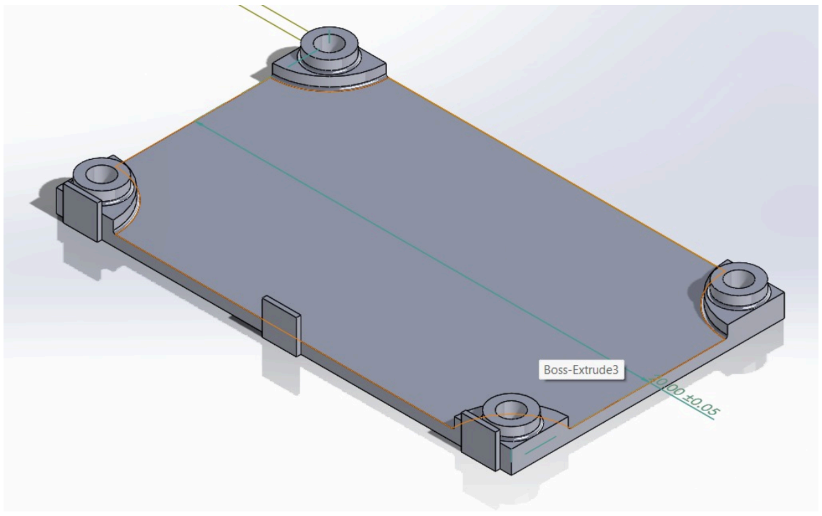


Fig 2 Lower Plate

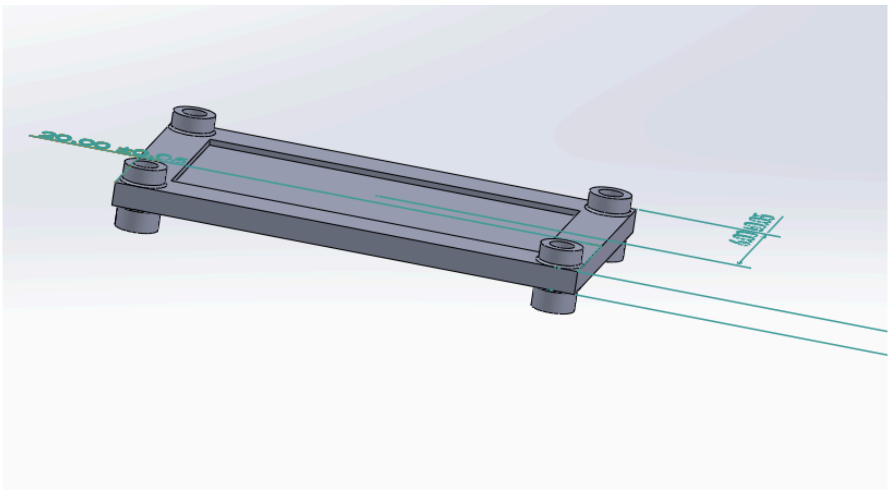


Fig 3 Middle Plate

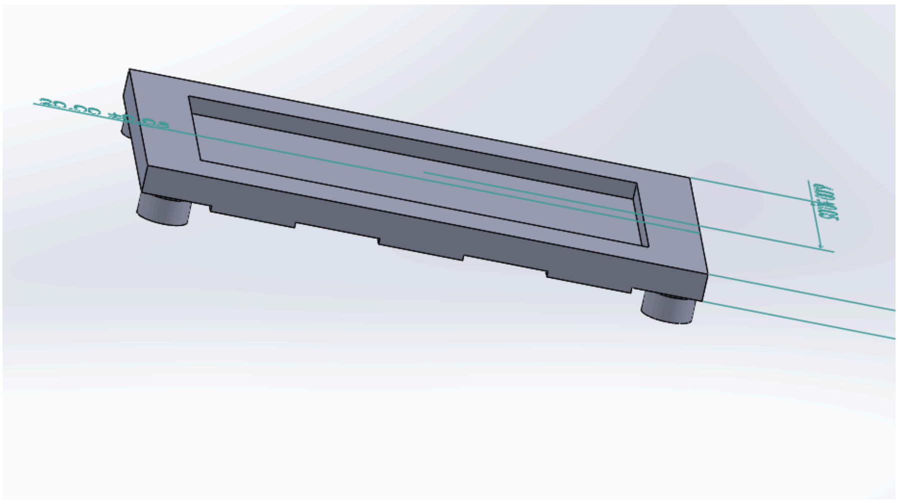


Fig 4 Top plate

Rods (x8)

Four rods would connect the upper and middle plates while the rest would connect the middle and lowermost one.

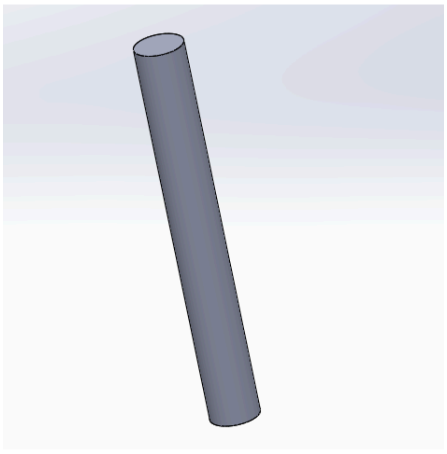


Fig 5 Rod

Wheels

Plastic wheels were deemed suitable for the bot as they were lightweight and capable of handling the torque from the motor. Wheels of appropriate dimensions were chosen for this purpose to give the required ground clearance.

Nuts and Bolts

Nuts and bolts were used to attach the motors and the cubical box to the legs securely.

B. Electronics component

Arduino Nano(Microcontroller)

The microcontroller that we have used in our robot is the arduino nano. It features 22 digital I/O pins, 8 analog inputs, and operates at 5V with a clock speed of 16MHz. It can be programmed via the USB interface using the Arduino IDE. Its small size and versatility make it suitable for embedded systems and space-constrained applications.

Motor Driver

A4988 stepper motor driver has been used to ensure it’s compatibility with the stepper motor that we have decided to use for our robot.

Motor

The NEMA17 stepper motor has been chosen for its precision and high torque. This motor allows for accurate positioning and speed control, making it ideal for applications that require fine control over movement.

IR sensor module

In order to enable the robot to follow a path, IR sensors have been used in the circuitry which detects the line and accordingly moves it forward such that it maintains the route that is intended to be followed.

Voltage Regulators

Voltage regulators (LM2596 DC-DC Step down converter) ensure that constant voltage is supplied to the electronic components even in the case of slight fluctuations in the voltage supply and prevent any kind of damage.

Resistors and Capacitors

Resistors help in limiting the amount of current that flows into various electronic components. Capacitors smooth voltage fluctuations and filter noise in power supplies, especially near stepper motor drivers, ensuring stable operation. They also reduce electromagnetic interference (EMI) generated by switching currents, improving motor performance and accuracy.

Diodes

Diodes protect circuits by blocking reverse current flow and ensuring unidirectional current, safeguarding sensitive components from damage.

C. Tools and Equipment

3D printer

One of the major equipment that we have used to manufacture the robot at the proof of concept stage is the 3D printer. We first designed our model in solidworks and then printed the components. Care was taken to ensure that the design consists of parts which can be assembled post printing. This, though might seem insignificant, was essential to ensure that the printing process happens seamlessly and without any cut-offs due to overloading.

Pliers, Screwdrivers etc.

These tools were used to tighten the nuts and bolts securely.

Drill machine and Drill Bits

The machine was used to slightly widen the holes to fit the bolt exactly and securely.

Soldering Iron

Soldering iron fit the naked wires to their correct spots and ensured proper conductivity.

Multimeter

Multimeter is used to adjust the motor driver current, ensuring accurate power delivery to the motors, and to check wiring continuity for proper circuit functionality.

Electro - Mechanical Component

Fig 5 1.8° 42mm Hybrid Stepper Motor-NEMA17

Stepper motor was used to convert the electrical signals to mechanical output. Stepper motors offer a variety of advantages over DC motors and servo motors. They do not need an encoder (which is required in DC motors) and overall precision is easier to achieve using stepper motors. The step pulse frequency makes it easier to control the motor which is essential to adjust not just the speed of the robot but the acceleration/deceleration depending on the tilt error.

3. Software Used and Developed

The software used for writing and developing the code is the Arduino IDE. Additionally, SolidWorks was utilized for designing the mechanical components of the robot, enabling the creation of detailed 3D models and simulations. The following sections present the code developed specifically for our robot, with comments included to ensure smooth readability and understanding.

Code:

```
// Include Wire.h library for communication
#include <Wire.h>

// pins
const int dirPin1 = 5;
const int stepPin1 = 4;
const int dirPin2 = 3;
const int stepPin2 = 2;
const int irSensorPin = 9;
const int irSensorPin1 = 8;
const int irSensorPin2 = 10;

float Kp = 14;
float Ki = 2;
float Kd = 2;
float turnRate = 30;
float maxSpeed = 15;

float angleGyro, angleAcc, angle, balancePidSetpoint;
float pidErrorTemp, pidIMem, pidSetpoint, pidOutput, pidLastDError;
float pidOutputLeft, pidOutputRight;
```

```
int leftMotor, throttleLeftMotor, throttleCounterLeftMotor, throttleLeftMotorMemory;
int rightMotor, throttleRightMotor, throttleCounterRightMotor,
throttleRightMotorMemory;
int receiveCounter;

// MPU-6050 I2C address (0x68)
int gyroAdd = 0x68;
int accCalibration = -410;
long gyroYawCalibration, gyroPitchCalibration;
int gyroPitchData, gyroPitchData, accelerometerData;

byte start, inputByte;
unsigned long loopTimer;

// Initial setup runs only at the start
void setup()
{
    Serial.begin(9600);
    Wire.begin();
    // Setting I2C clock speed to 400kHz
    TWBR = 12;

    // To create a timer of 20us
    TCCR2A = 0;
    TCCR2B = 0;
    TIMSK2 |= (1 << OCIE2A);
    TCCR2B |= (1 << CS21);
    OCR2A = 39;
    TCCR2A |= (1 << WGM21);

    // Start MPU-6050
    Wire.beginTransmission(gyroAdd);
    Wire.write(0x6B);
    Wire.write(0x00);
    Wire.endTransmission();
    // Setting full scale of the gyro to +/- 250 degrees per second
    Wire.beginTransmission(gyroAdd);
    Wire.write(0x1B);
    Wire.write(0x00);
    Wire.endTransmission();
    // Setting full scale of the accelerometer to +/- 4g.
    Wire.beginTransmission(gyroAdd);
    Wire.write(0x1C);
    Wire.write(0x08);
    Wire.endTransmission();
    // Setting filtering to improve the raw data.
    Wire.beginTransmission(gyroAdd);
    Wire.write(0x1A);
    Wire.write(0x03);
    Wire.endTransmission();

    pinMode(stepPin1, OUTPUT);
    pinMode(dirPin1, OUTPUT);
    pinMode(stepPin2, OUTPUT);
    pinMode(dirPin2, OUTPUT);
    pinMode(13, OUTPUT);

    pinMode(irSensorPin, INPUT);
    pinMode(irSensorPin1, INPUT);
    pinMode(irSensorPin2, INPUT);

    calibrateGyro();
    loopTimer = micros() + 4000;
}

// Main program loop
void loop()
{
    int IRS = digitalRead(irSensorPin);
    int IRSL = digitalRead(irSensorPin1);
    int IRSR = digitalRead(irSensorPin2);

    if (IRS)
    {
        inputByte = 0x04;
        receiveCounter = 0;
    }
    if (receiveCounter <= 25)
        receiveCounter++;
    else
        inputByte = 0x00;

    // Angle calculations
    Wire.beginTransmission(gyroAdd);
    Wire.write(0x3F);
    Wire.endTransmission();
    Wire.requestFrom(gyroAdd, 2);
    accelerometerData = Wire.read() << 8 | Wire.read();
    accelerometerData += accCalibration;
    if (accelerometerData > 8200) accelerometerData = 8200;
    if (accelerometerData < -8200) accelerometerData = -8200;

    angleAcc = asin((float)accelerometerData / 8200.0) * 57.296;

    if (start == 0 && angleAcc > -0.5 && angleAcc < 0.5)
    {
        angleGyro = angleAcc;
        start = 1;
    }

    Wire.beginTransmission(gyroAdd);
    Wire.write(0x43);
    Wire.endTransmission();
    Wire.requestFrom(gyroAdd, 4);
    gyroPitchData = Wire.read() << 8 | Wire.read();
    gyroPitchData = Wire.read() << 8 | Wire.read();

    gyroPitchData -= gyroPitchCalibration;
    angleGyro += gyroPitchData * 0.000031;

    gyroPitchData -= gyroYawCalibration;
    // angleGyro -= gyroPitchData * 0.0000003;

    // Complementary Filter to correct the drift of the gyro angle
    angleGyro = angleGyro * 0.9996 + angleAcc * 0.0004;

    pidCalculation();
    movementCalculation();
    controlMotors();

    // to ensure every loop is exactly 4ms
    while (loopTimer > micros());
    loopTimer += 4000;
}

// to calibrate gyro settings
void calibrateGyro()
{
    for (receiveCounter = 0; receiveCounter < 500; receiveCounter++)
    {
        if (receiveCounter % 15 == 0)
```

```
            digitalWrite(13, !digitalRead(13));
        Wire.beginTransmission(gyroAdd);
        Wire.write(0x43);
        Wire.endTransmission();
        Wire.requestFrom(gyroAdd, 4);
        gyroYawCalibration += Wire.read() << 8 | Wire.read();
        gyroPitchCalibration += Wire.read() << 8 | Wire.read();
        delayMicroseconds(3700);
    }
    gyroPitchCalibration /= 500;
    gyroYawCalibration /= 500;
}

// PID calculations
void pidCalculation()
{
    pidErrorTemp = angleGyro - balancePidSetpoint - pidSetpoint;
    if (pidOutput > 10 || pidOutput < -10)
        pidErrorTemp += pidOutput * 0.015;

    pidIMem += Ki * pidErrorTemp;
    if (pidIMem > 400)
        pidIMem = 400;
    else if (pidIMem < -400)
        pidIMem = -400;
    // Calculate the PID output value
    pidOutput = Kp * pidErrorTemp + pidIMem + Kd * (pidErrorTemp - pidLastDError);
    if (pidOutput > 400)
        pidOutput = 400;
    else if (pidOutput < -400)
        pidOutput = -400;

    pidLastDError = pidErrorTemp;

    if (pidOutput < 5 && pidOutput > -5)
        pidOutput = 0;

    // if angle is greater than 30 stop the bot
    if (angleGyro > 30 || angleGyro < -30 || start == 0)
    {
        pidOutput = 0;
        pidIMem = 0;
        start = 0;
        balancePidSetpoint = 0;
    }
}

// Movement control calculations for LRFB-(Left,Right,Front,Back)
void movementCalculation()
{
    pidOutputLeft = pidOutput;
    pidOutputRight = pidOutput;

    // Turn robot leftward
    if (inputByte & B00000001)
    {
        pidOutputLeft += turnRate;
        pidOutputRight -= turnRate;
    }
    // Turn robot rightward
    if (inputByte & B00000010)
    {
        pidOutputLeft -= turnRate;
        pidOutputRight += turnRate;
    }

    // Move forward
    if (inputByte & B00000100)
    {
        if (pidSetpoint > -2.5)
            pidSetpoint -= 0.05;
        if (pidOutput > maxSpeed * -1)
            pidSetpoint -= 0.005;
    }
    // Move backward
    if (inputByte & B00001000)
    {
        if (pidSetpoint < 2.5)
            pidSetpoint += 0.05;
        if (pidOutput < maxSpeed)
            pidSetpoint += 0.005;
    }

    // restore settings
    if (!(inputByte & B00001100))
    {
        if (pidSetpoint > 0.5)
            pidSetpoint -= 0.05;
        else if (pidSetpoint < -0.5)
            pidSetpoint += 0.05;
        else
            pidSetpoint = 0;
    }

    if (pidSetpoint == 0)
    {
        if (pidOutput < 0)
            balancePidSetpoint += 0.0010;
        if (pidOutput > 0)
            balancePidSetpoint -= 0.0010;
    }
}

// Motor control pulse calculations
void controlMotors()
{
    if (pidOutputLeft > 0)
        pidOutputLeft = 405 - (1 / (pidOutputLeft + 9)) * 5500;
    else if (pidOutputLeft < 0)
        pidOutputLeft = -405 - (1 / (pidOutputLeft - 9)) * 5500;

    if (pidOutputRight > 0)
        pidOutputRight = 405 - (1 / (pidOutputRight + 9)) * 5500;
    else if (pidOutputRight < 0)
        pidOutputRight = -405 - (1 / (pidOutputRight - 9)) * 5500;

    // pulse time calculation for the left and right stepper motor controllers
    if (pidOutputLeft > 0)
        leftMotor = 400 - pidOutputLeft;
    else if (pidOutputLeft < 0)
        leftMotor = -400 - pidOutputLeft;
    else
        leftMotor = 0;

    if (pidOutputRight > 0)
        rightMotor = 400 - pidOutputRight;
    else if (pidOutputRight < 0)
        rightMotor = -400 - pidOutputRight;
    else
        rightMotor = 0;

    throttleLeftMotor = leftMotor;
    throttleRightMotor = rightMotor;
}

ISR(TIMER2_COMPA_vect)
{
    // Left motor pulse calculations
    throttleCounterLeftMotor++;
    if (throttleCounterLeftMotor > throttleLeftMotorMemory)
    {
        throttleCounterLeftMotor = 0;
        throttleLeftMotorMemory = throttleLeftMotor;
        if (throttleLeftMotorMemory < 0)
        {
            PORTD &= 0b11110111;
            throttleLeftMotorMemory *= -1;
        }
        else
            PORTD |= 0b00001000;
    }
    else if (throttleCounterLeftMotor == 1)
        PORTD |= 0b00000100;
    else if (throttleCounterLeftMotor == 2)
        PORTD &= 0b11111011;

    // right motor pulse calculations
    throttleCounterRightMotor++;
    if (throttleCounterRightMotor > throttleRightMotorMemory)
    {
        throttleCounterRightMotor = 0;
        throttleRightMotorMemory = throttleRightMotor;
        if (throttleRightMotorMemory < 0)
        {
            PORTD |= 0b00100000;
            throttleRightMotorMemory *= -1;
        }
        else
            PORTD &= 0b11011111;
    }
    else if (throttleCounterRightMotor == 1)
        PORTD |= 0b00010000;
    else if (throttleCounterRightMotor == 2)
        PORTD &= 0b11101111;
}
```

4. Additional Features in Actual Robot-Making

An additional middle plate has been added to the model, keeping future enhancements in mind. The connections have been designed to allow easy integration of a Bluetooth module with minimal wiring, enabling remote control of the robot. This flexible design ensures that more functionalities can be added without significant changes.

5. Concurrence / Deviation from stage 1 Report

During the design and development of the robot, certain adjustments were deemed essential to the initial component selection to better meet the project's requirements. These deviations were driven *by the need for enhanced precision, stability, and adherence to the primary objectives. The major changes and their justification can be enumerated as follows:*

1. **Using Stepper Motor Instead of DC Motor**:

   Precise Position Control:Since DC motors work with intermittent rotation, while stepper motors are capable of discrete singular steps this affords precise control over their position, making them ideal for applications where accuracy is crucial, such as balancing a robot. A stepper corresponds to a precise rotation therefore, adjustments to the robot position would be carried out to correct an imbalance.

   Better Torque at Low Speeds:Stepper motors have no appreciably negative effects on a minor voltage fluctuation that can potentially lead to speed drift in DC motors. Since the rotation of a stepper motor depends on the number of input pulses rather than the voltage, the result is constant speed, an important factor in the stable and reliable performance of a self-balancing robot.

No Speed Drift Due to Voltage Fluctuation: Stepper motors are not affected by minor voltage fluctuations that can cause speed drift in DC motors. Since the rotation of a stepper motor is controlled by the number of input pulses rather than the voltage, the speed remains consistent, which is essential for reliable and stable performance in a self-balancing robot.

2. **Using Arduino Nano Instead of ESP32**:

   No External Controls Allowed: Since it was required for the project to have no need for any external control, Arduino Nano was chosen instead of ESP32. Even though ESP32 provides WiFi and Bluetooth connectivity, it was not necessary for the execution of the desired autonomous function without oversight. The Arduino Nano is, in this case, the simpler and compact Arduino platform, while still efficient in the handling of controlling tasks involved in this project.

3. **Inclusion of IR Sensors for Autonomous Movement**:

   Enhanced Navigation Capability: To support autonomous movement, IR sensors were added to the design to enable line-following functionality. These sensors detect contrasts on the surface and provide data to the control system, allowing the robot to track a predefined path. This addition makes the robot capable of navigating complex environments independently, improving its versatility for tasks like material transport, assembly line operations, and competitive robotics scenarios.

6. Other Description

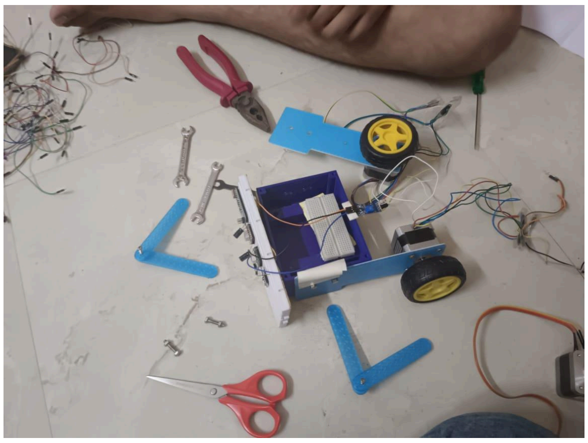
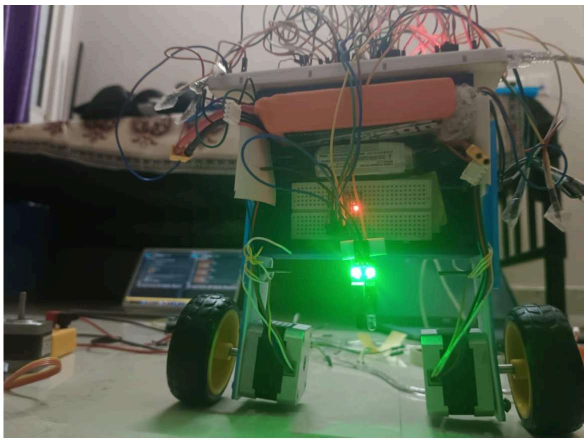
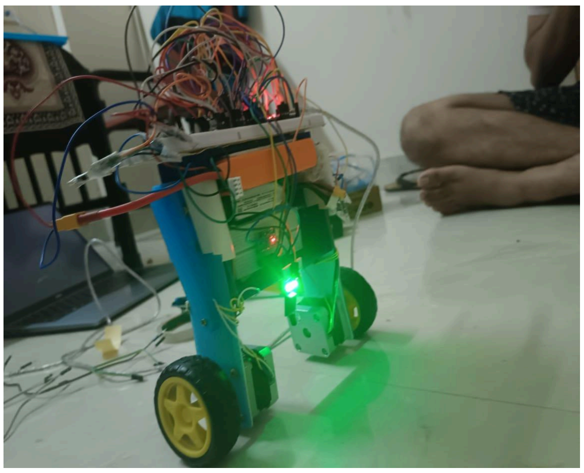
One of the key techniques used in building the robot was opting for an **Interrupt Service Routine (ISR)** over the standard **AccelStepper library**. This choice was driven by several critical factors:

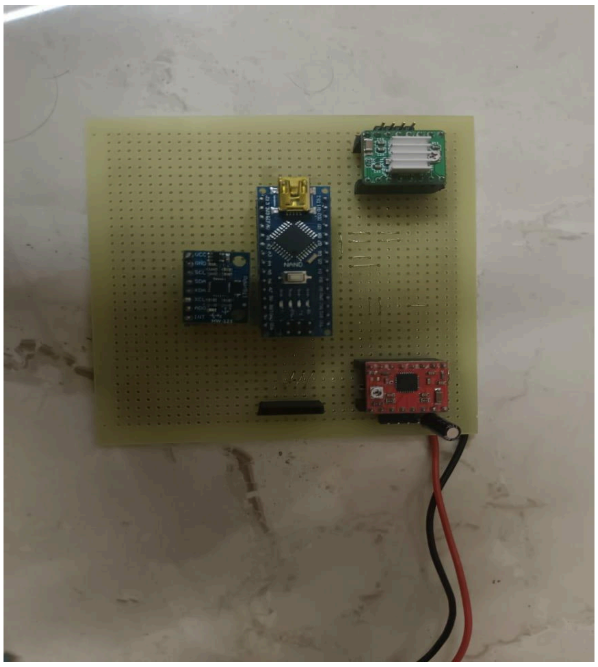
1. **Precision and Control**: The ISR is triggered every 20 microseconds, allowing for precise and consistent pulse generation for the stepper motors. The AccelStepper library, while convenient, can introduce overhead that may lead to slight delays or inconsistencies, especially when precise timing is critical. By using an ISR, we ensured that motor control tasks were executed at specific, tightly controlled intervals, maintaining the robot's stability and accuracy.
2. **Real-Time Response**: With the ISR operating at a high frequency, the robot can make rapid adjustments in response to changes in tilt or angular velocity detected by the MPU 6050. This real-time response is essential for maintaining balance, as it allows the system to react quickly and predictively without lag. By contrast, the standard library might introduce latency, reducing the system's ability to respond immediately.
3. **Efficient Use of Processing Resources**: Since the ISR handles motor control directly, it bypasses the higher-level abstractions of the library, resulting in more efficient code execution. The Arduino Nano loop function, which runs every 4 milliseconds, can then focus on other tasks such as reading sensor data, performing PID calculations, and handling communication, without being burdened by motor

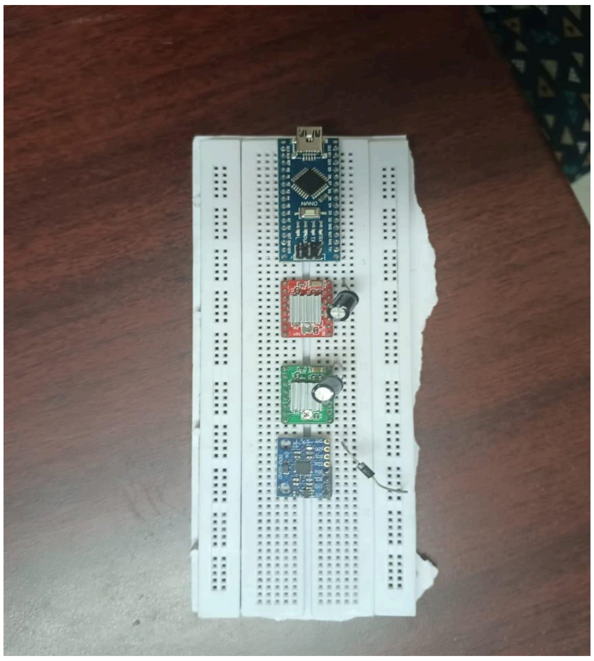
control logic. This division of tasks ensures smooth and uninterrupted operation of the system.

Moreover, we used the **register values of the Arduino Nano** to configure specific parameters, including the SCL (Serial Clock Line) frequency and conditions under which the ISR is called. By directly setting these registers, we could fine-tune the communication speed between the microcontroller and sensors, ensuring fast and reliable data acquisition. This low-level control also allowed for precise timing adjustments for the ISR, which was essential for maintaining consistent motor performance and stability.

7. Photos:





Video Link:

https://drive.google.com/file/d/1Eq4Ps5q3Ih3qkz_DgO-prheO2mjEV1y3/view?usp=sharing

8. ABOUT THE TEAM

Mentor:

Dr. Satyanarayan Panigrahi
Associate Professor
School of Mechanical Sciences

Ph.D. Mechanical (Technical Acoustics) - 2007
Indian Institute of Science, Bangalore

Research Interests

Technical Acoustics, Industrial Noise Control, Automotive noise control, Acoustic based Condition Monitoring of Machines, and Musical Acoustics.

Team Members:

1. Ayush Gupta
2. Divya Kumar
3. Jiya Chakraborty
4. Rachit Jain
5. Vivek Singh