Self-Balancing and Traction Vehicle

Aim:

This project is about building a self-balancing and traction vehicle. The main goal is to create a vehicle that can balance itself without falling and move smoothly over different surfaces.

Introduction:

This kind of vehicles offers remarkable stability and safety for urban environments. These vehicles use advanced sensors and real-time data to maintain balance and grip, even on slippery or uneven roads. Traction control systems further enhance safety by preventing wheel spin and maintaining grip on slippery or uneven surfaces. By automatically applying brakes or adjusting engine power when a loss of traction is detected, these systems help drivers maintain control, reduce the likelihood of accidents.

Materials to be used:

Component	Use in Project
Microcontroller (Arduino/ESP32)	Brain of the vehicle; reads data from sensors and controls motors
Gyroscope + Accelerometer (MPU6050)	Detects tilt, angle, and motion for balancing.
Motor Driver (L298D)	Controls the speed and direction of motors.
DC Geared Motors	Drives the wheels forward and backward.
Wheels	Allow movement of the vehicle.
Battery	Powers all electronic parts.
Frame	Base structure to hold everything together.
Jumper Wires	Connect all components together.
Switch	Turns the system ON/OFF.

Benefits and Applications:

Benefits:

Improved Stability and Safety: Helps prevent tipping over, useful in uneven terrain or during sudden movements.

Ease of Use: Simplifies operation for users with less driving or riding experience.

Increased Safety: Prevents wheel slip during acceleration on slippery surfaces like rain, ice, or gravel.

Improved Vehicle Performance: Ensures optimal power distribution to the wheels for efficient acceleration.

Enhanced Towing and Load-Carrying: Traction control helps prevent traction loss due to shifting weight.

Applications:

Agriculture: TCS in tractors and ATVs to improve grip in muddy or uneven fields.

Healthcare & Accessibility: For mobility aids like self-balancing wheelchairs.

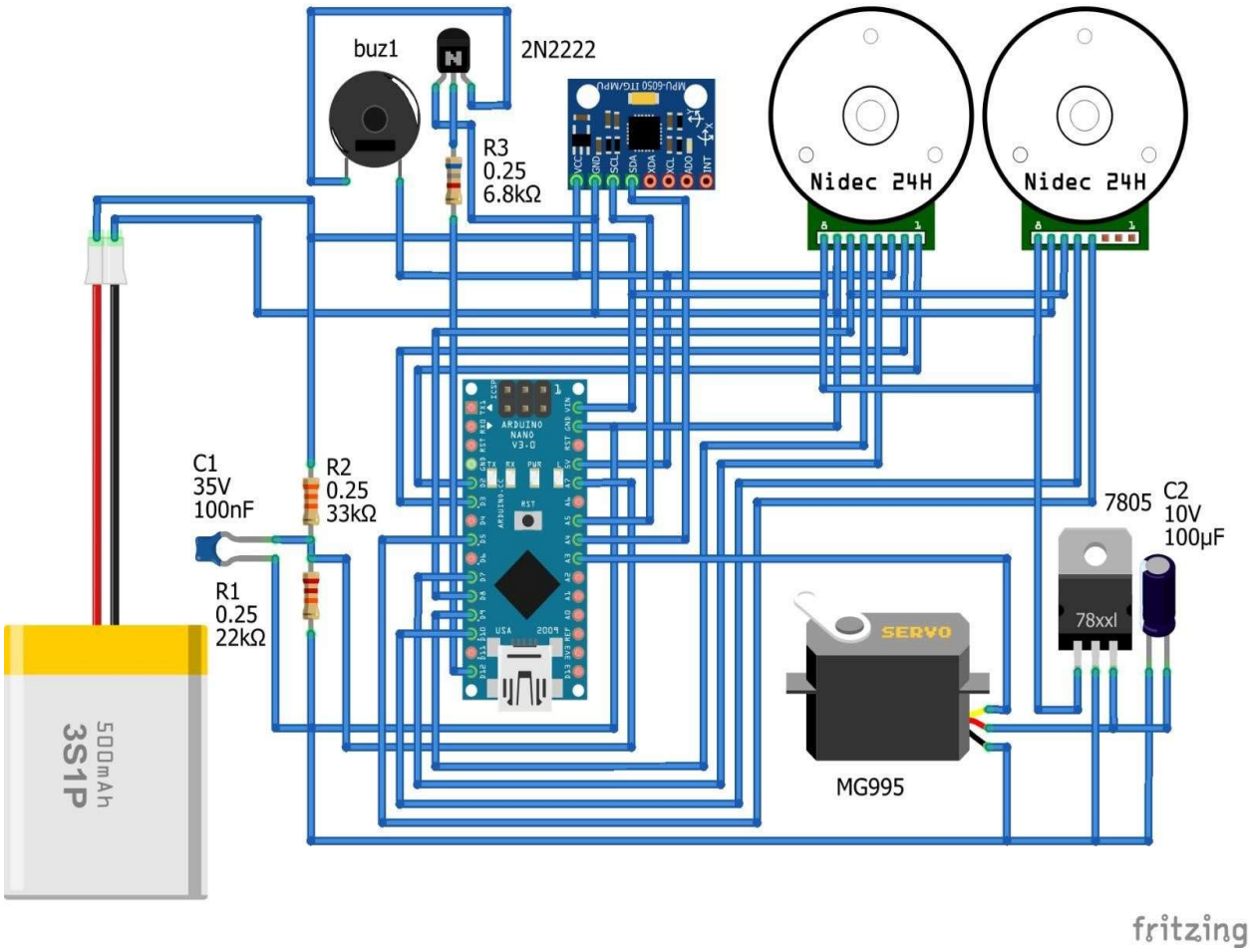
Logistics & Warehousing: Self-balancing robots and trolleys for automated material handling.

Personal Transport: Self-balancing motorcycles.

Urban Mobility: Self-balancing scooters, hoverboards, and bicycles.

Image of Prototype:

Circuit Design for SSTV:



Our Approach:

Step 1: Design Mechanical Structure

- Use CAD software to design a sturdy chassis with a low center of gravity.
- Allow space for battery, controller, motors, and sensors.
- Typically two large wheels for balance.

Step 2: Selecting and Integrating Sensors

- For Balance: We used MPU6050 which has gyroscope and accelerometer
- For Traction: We ued motor encoders to monitor wheel speed and detect slippage.

Step 3: Sensor Integration and Data Processing

- The IMU provides raw acceleration and angular velocity data.

Step 4: Control System Development

- A closed-loop Proportional–Integral–Derivative (PID) controller was implemented to regulate the balance of the vehicle.

Step 5: Software Implementation

- Continuously read IMU sensor data.
- Calculate the PID control output.

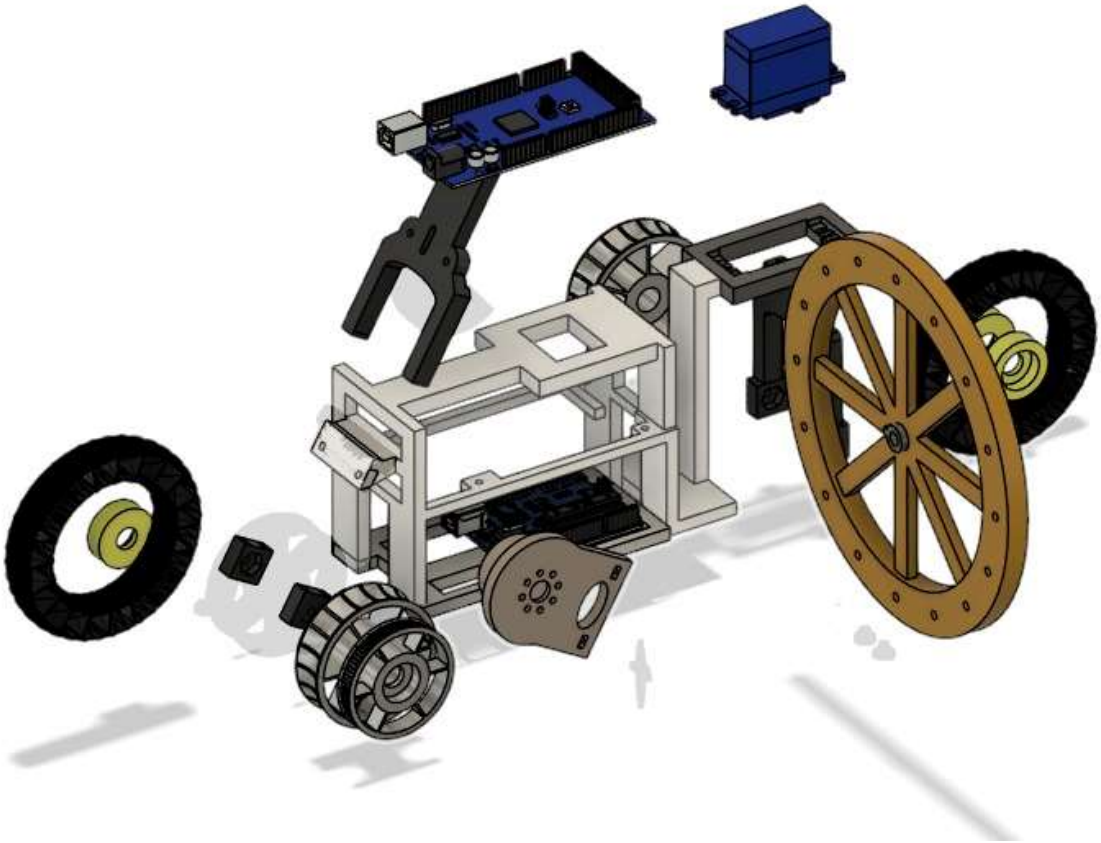
Step 6: Testing

- Checking responses on Serial Monitor.
- Balancing tests

Conclusion:

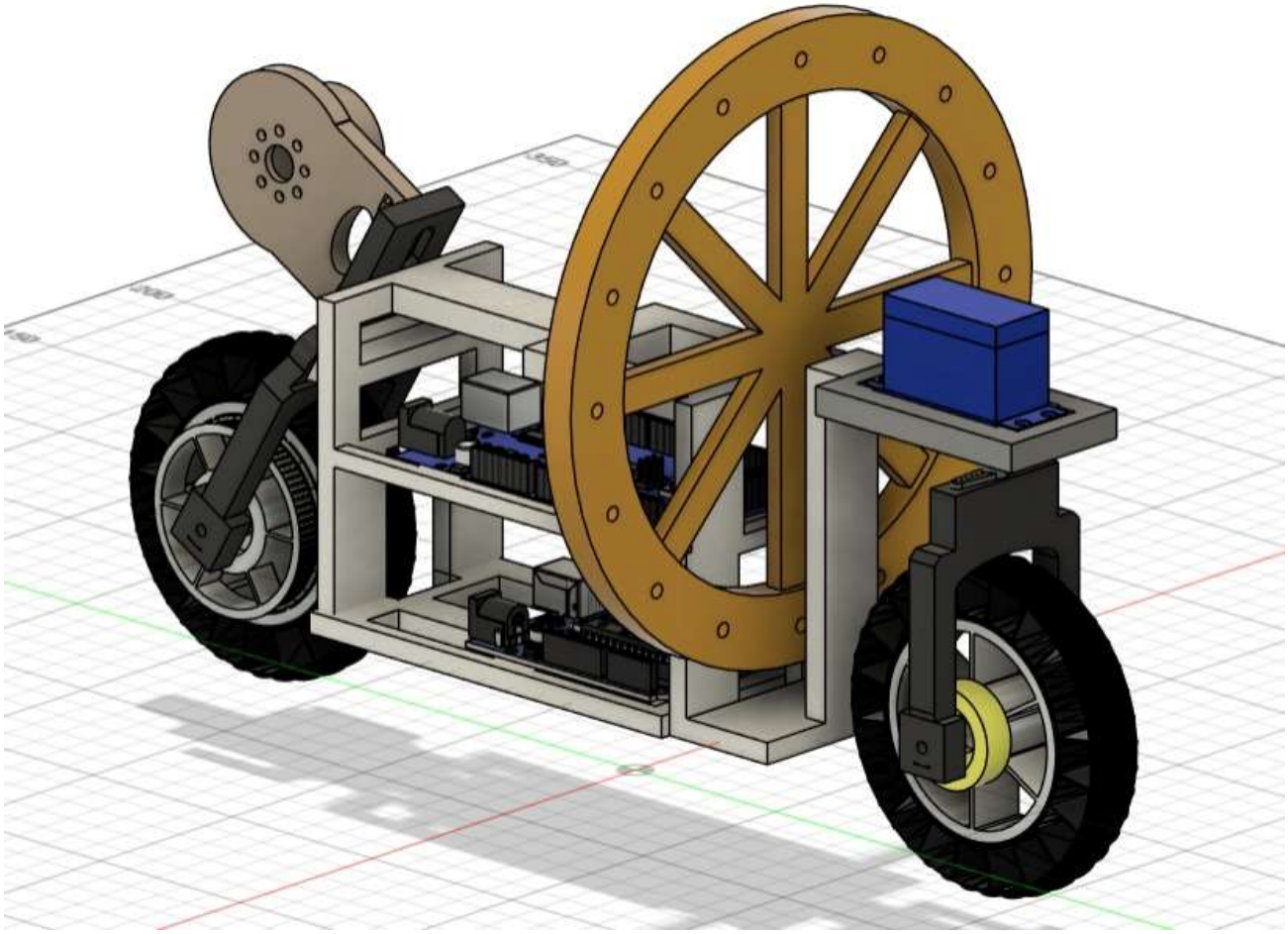
The self-balancing bike project aimed to integrate inertial sensing, feedback control, and mechanical stabilization using a flywheel-based system. Although the final

implementation faced unresolved issues in sensor response, communication, or control execution, the project provided substantial hands-on exposure to real-time systems, PID control tuning, sensor interfacing (MPU6050), and motor driver configurations (L298N and L293D). Working with both hardware (servo motors, encoders, buzzers) and software (Arduino IDE, ESP8266 communication) deepened understanding of embedded control challenges. Despite the technical setbacks, the project served as a strong foundation in practical mechatronics, control theory, and system integration offering valuable insights for future iterations and related robotics projects.



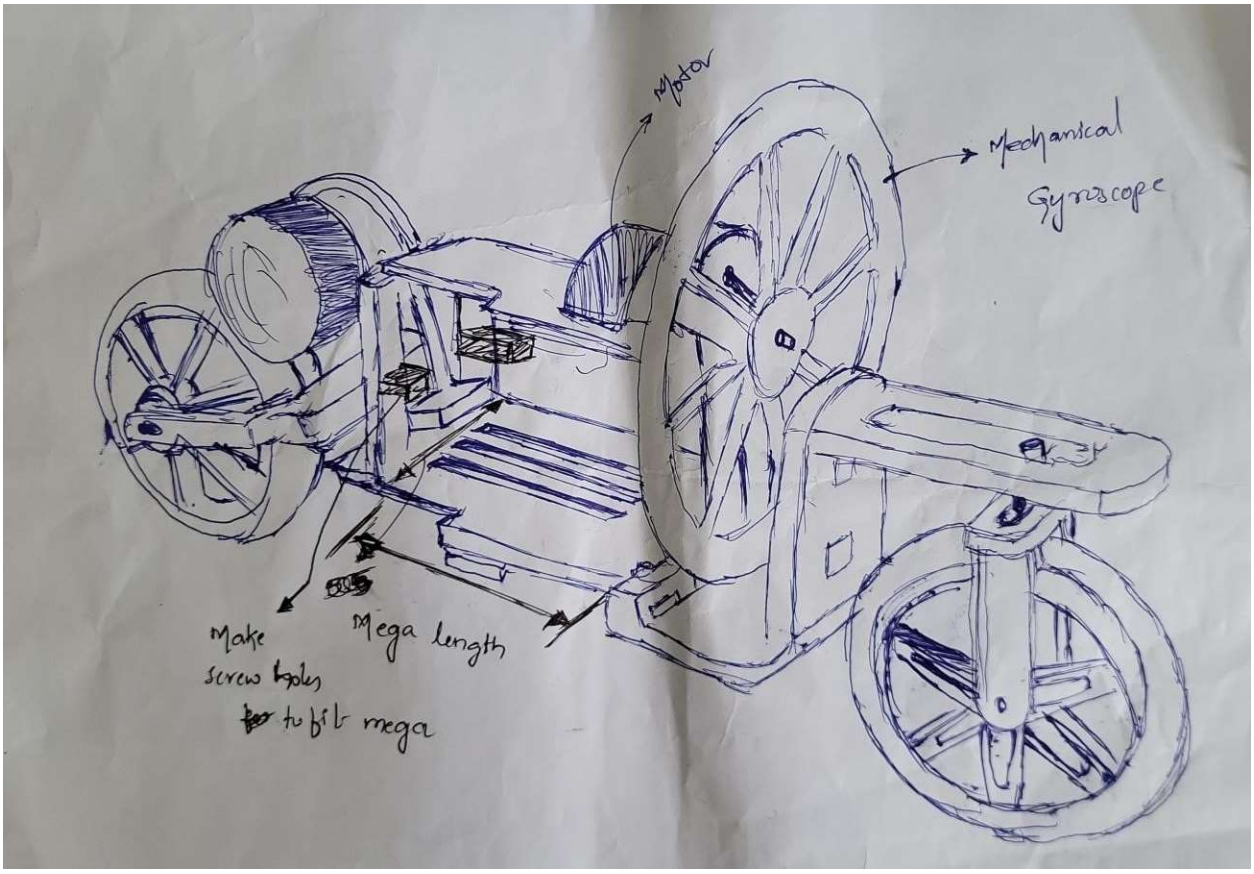
Annexure-1

The CAD model of t



Annexure-2

The rough drawing of the chassis is given in this annexure.



Annexure-3 (The code we used to run the mod)

Arduino IDE code for Arduino Mega.

```
#include <Wire.h>

#include <Servo.h>

#include <PID_v1.h>
```

```
#include <MPU6050.h>

// MPU6050
MPU6050 mpu;
double angle, gyroRate;

// PID double setpoint = 0;       // Desired angle (upright)
double input, output;      // Measured tilt and PID output
double Kp = 20, Ki = 0.5, Kd = 0.8;
PID pid(&input, &output, &setpoint, Kp, Ki, Kd, DIRECT);

// Flywheel Motor (L298N)
#define FLY_ENA 5
#define FLY_IN1 6
#define FLY_IN2 7

// Rear Wheel Motor (L293D)
#define REAR_EN 10
#define REAR_IN1 8
#define REAR_IN2 9

// Servo Motor (MG995)
Servo steeringServo;
#define SERVO_PIN 11

// Buzzers
#define BUZZER1 3
#define BUZZER2 4

// ESP8266 Serial
```

```
#define ESP Serial3  // Use Serial3 on Mega (pins 14 RX3, 15 TX3)

// Encoder (Optional if implemented)
volatile long encoderCount = 0;

// Timer unsigned long
lastTime = 0; void setup() {
  Serial.begin(115200);
  ESP.begin(9600);

  // Motor Pins
  pinMode(FLY_ENA, OUTPUT);
pinMode(FLY_IN1, OUTPUT);
pinMode(FLY_IN2, OUTPUT);
pinMode(REAR_EN, OUTPUT);
pinMode(REAR_IN1, OUTPUT);
pinMode(REAR_IN2, OUTPUT);

  // Buzzer Pins
  pinMode(BUZZER1, OUTPUT);
pinMode(BUZZER2, OUTPUT);

  // Servo Init
  steeringServo.attach(SERVO_PIN);
steeringServo.write(90); // Center position

  //     MPU6050     Init                Wire.begin();
mpu.initialize();     if  (!mpu.testConnection()) {
digitalWrite(BUZZER1, HIGH);       while (1); //
Stop here if MPU6050 not found
```

```
  }

  // PID Init
  pid.SetMode(AUTOMATIC);   pid.SetOutputLimits(-
255, 255);  // Output for PWM   pid.SetSampleTime(10);
// 10 ms loop

  // Initial buzzer beep
beepBuzzers();
}
void loop() {
  // Read angle from MPU6050
angle = getTiltAngle();   input
= angle;

  // Compute PID
pid.Compute();

  // Apply flywheel motor control
controlFlywheel(output);

  // Rear wheel constant forward (can be PID or manual later)
driveRearWheel(150);  // Adjust as needed

  // Read from ESP8266 for steering   if (ESP.available()) {
char cmd = ESP.read();     if (cmd == 'L')
steeringServo.write(60);        // Turn Left     else if (cmd ==
'R') steeringServo.write(120);  // Turn Right     else
steeringServo.write(90);                  // Center
  }
```

```
  // Optional alert if tilt > threshold
if (abs(angle) > 30) {
beepBuzzers();
  }
  delay(10);
}

// --- FUNCTIONS --- void
controlFlywheel(double val) {
  if (val > 0) {
digitalWrite(FLY_IN1, HIGH);
digitalWrite(FLY_IN2, LOW);
  } else {    digitalWrite(FLY_IN1,
LOW);      digitalWrite(FLY_IN2,
HIGH);
    val = -val;
  }
  analogWrite(FLY_ENA, constrain(val, 0, 255));
}

void driveRearWheel(int speed) {
digitalWrite(REAR_IN1, HIGH);   digitalWrite(REAR_IN2,
LOW);   analogWrite(REAR_EN, constrain(speed, 0,
255));
}
void beepBuzzers() {
digitalWrite(BUZZER1, HIGH);
digitalWrite(BUZZER2, HIGH);
delay(100);
```

```
digitalWrite(BUZZER1, LOW);
digitalWrite(BUZZER2, LOW); }

// Function to get tilt angle using MPU6050 double
getTiltAngle() {     int16_t ax, ay, az, gx, gy, gz;
mpu.getMotion6(&ax, &ay, &az, &gx, &gy, &gz);

  // Basic calculation of pitch angle from accelerometer
double accX = ax / 16384.0;   double accZ = az /
16384.0;   double pitch = atan2(accX, accZ) * 180.0 /
PI;   return pitch;
}
```

Arduino IDE code for ESP32

```
#include <ESP8266WiFi.h>
#include <ESP8266WebServer.h>

ESP8266WebServer server(80);

// Wi-Fi settings for ESP8266 Access Point
const char* ssid = "BikeControl"; const
char* password = "12345678";

// Web server handler functions void
handleRoot() {   server.send(200, "text/html",
R"rawliteral(
   <!DOCTYPE html>
```

```
<html>
<head><title>Self-Balancing Bike Control</title></head>
<body style='text-align:center;font-family:sans-serif'>
  <h2>Control Your Self-Balancing Bike</h2>
  <button onclick="sendCmd('F')">Forward</button>
  <button onclick="sendCmd('B')">Backward</button>
  <button onclick="sendCmd('S')">Stop</button><br><br>
  <button onclick="sendCmd('L')">Turn Left</button>
  <button onclick="sendCmd('R')">Turn Right</button><br><br>
  <button onclick="sendCmd('C')">Center</button><br><br>

  <label for="speed">Speed Control:</label>
  <input type="range" min="0" max="255" value="150" id="speed"
onchange="setSpeed()">
  <p>Speed: <span id="spdval">150</span></p>

  <script>
    // Function to send commands to Arduino
function sendCmd(cmd) {      fetch("/cmd?val="
+ cmd);
    }
    // Function to adjust speed        function setSpeed() {
let    val    =    document.getElementById('speed').value;
document.getElementById('spdval').innerText      =      val;
fetch("/speed?val=" + val);
    }
```

```
    </script>
  </body>
  </html>
 )rawliteral");
}
// Handle the received command (F/B/L/R/S) void
handleCmd() {
 String cmd = server.arg("val");
 Serial.print(cmd);  // Send the command to Arduino through Serial
Serial.print("#");  // Add the end marker   server.send(200,
"text/plain", "OK");
}
// Handle the speed adjustment from the slider void
handleSpeed() {
 String speed = server.arg("val");
 Serial.print("S");
 Serial.print(speed);   // Send the speed command to Arduino
Serial.print("#");   server.send(200, "text/plain", "OK");
}
void setup() {
 // Initialize Serial communication
 Serial.begin(9600); // For communication with Arduino

 // Set up the ESP8266 as an access point
 WiFi.softAP(ssid, password);
```

```
  // Set up the routes and handlers   server.on("/", handleRoot);          //
Homepage   server.on("/cmd", handleCmd);        // Command handler (move
actions)   server.on("/speed", handleSpeed);     // Speed handler (slider)
  // Start the web server   server.begin();
  Serial.println("Web server started.");
}
 void loop() {
  // Handle incoming client requests   server.handleClient();
}
```