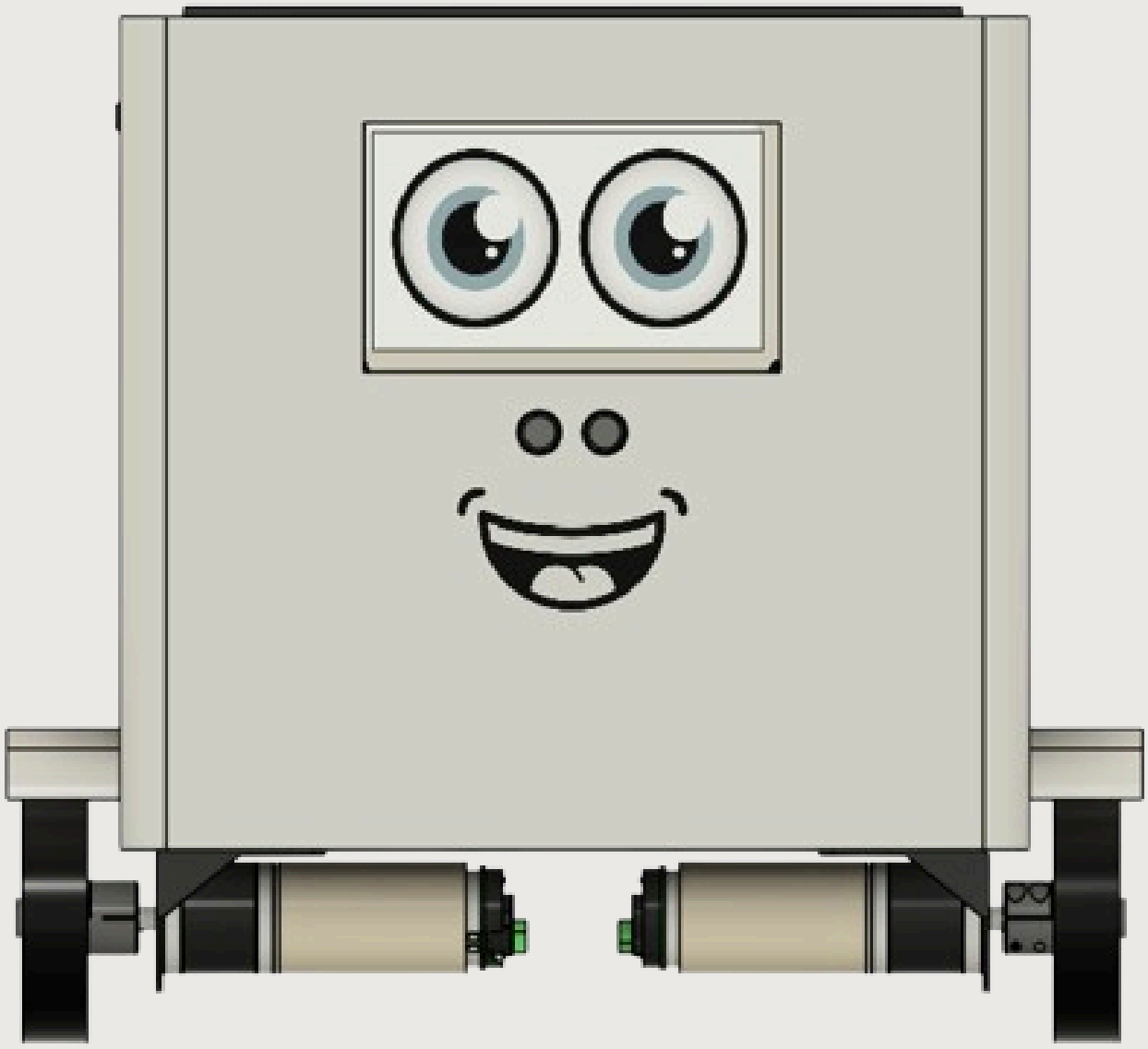
ROBOFEST 4.0



IIT Madras

Gyro Guide

INDEX

Robot Description

Description of Robot:

A two-wheeled self-balancing robot designed to revolutionize personal assistance in homes, hospitals, and beyond while maintaining balance autonomously. The robot uses an imu and feedback mechanism to constantly monitor and adjust its angular position relative to the desired orientation.

Engineered with a robust three-layer structure combining laser-cut steel, 3D-printed components, and carbon fiber reinforcement, the robot is both lightweight and robust, capable of withstanding the demands of real-world usage while remaining adaptable for future enhancements. Equipped with powerful electronics, high-torque motors, and sophisticated control algorithms, it balances and adapts in real time to its environment.

Key Features:

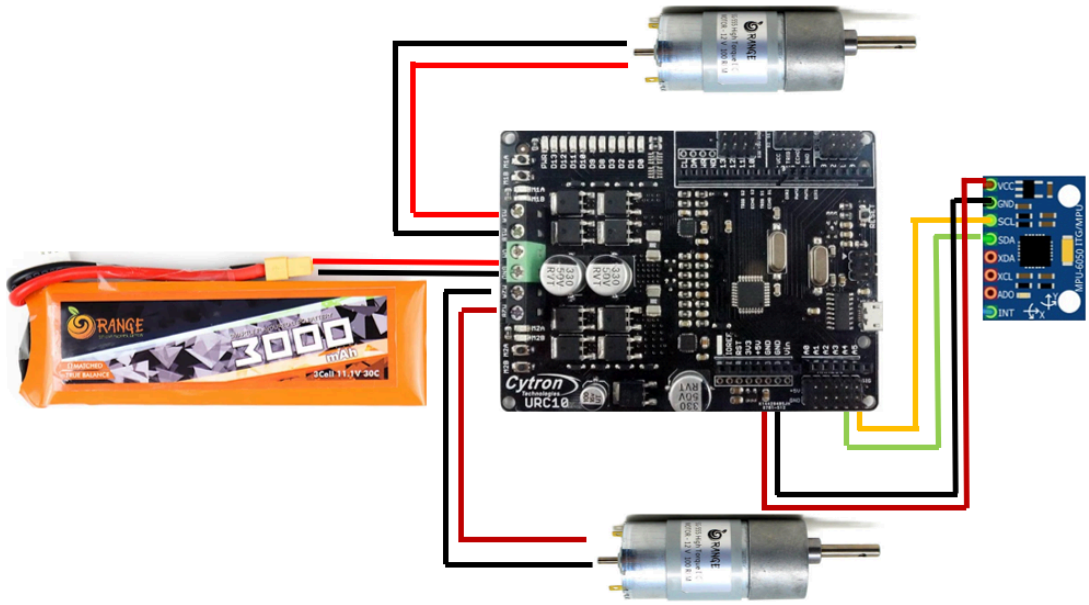
1. The selection of a High torque DC Motor 12V 100RPM 234N-cm Encoder compatible with 100mm diameter wheels provides sufficient torque, traction, and stability to counteract external disturbances and maintain balance, even on uneven terrain.
2. The robot employs an MPU-6050 sensor for precise angular position and acceleration measurement, enabling real-time feedback for maintaining balance.
3. PID control loop: Incorporates a PID control loop algorithm to continuously adjust motor torque based on feedback from the gyroscopic sensor, ensuring stable balance on two wheels.
4. Obstacle Avoidance: The robot integrates sensors and algorithms to detect obstacles in its path and autonomously navigate around them, enhancing safety and efficiency.
5. Joystick Control: Users can intuitively control the robot's movement and direction using a joystick interface, allowing for precise and user-friendly interaction.
6. Dynamic Balance: The robot maintains balance dynamically through sophisticated gyroscopic mechanisms and feedback control systems, ensuring stability even on uneven surfaces or during sudden movements.
7. User interface

Electronics Description:

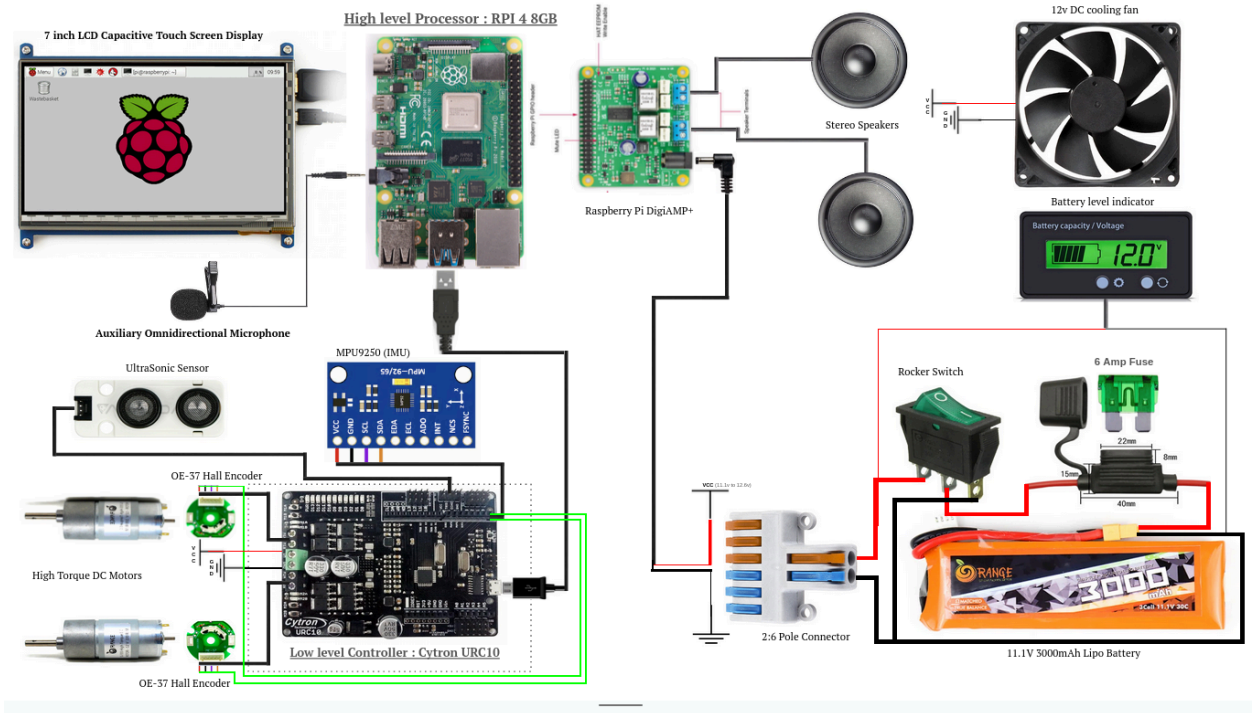
Electronic Schematic:

After extensive ideation and brainstorming, we finalized our robot's schematic. The schematic is structured into three main subunits: Power, Low-level controller, and High-level processing units. Each subunit plays a crucial role in ensuring the functionality and efficiency of our system. Below is a detailed overview of our schematic, highlighting the integration and functionality of each component.

Robot iteration 1:



Robot iteration 2:



Additions in Final prototype: The final prototype we aim for would be capable of much more than any self-balancing robot out there. It's not just a playful robot anymore. We're aiming to

deploy it as a personal assistant robot in hospitals and other places where there's a need for this kind of robot. More details about the final prototype are mentioned in the later sections.
We're currently designing a **custom PCB** which integrates the following:

- A 40-pin header for RPI and low-level controller seamless connections.
- MPU 6050 QFN-24 IC on the board.
- ESP32 IC with a PCB antenna model.
- Boot and Reset Buttons for ESP32
- A 5v, 3.3v power circuit for powering ESP32 and other components on the board.
- Motor driver circuit (H-bridge, voltage regulation using MOSFETs).
- Sensors I/O section for seamless connection.
- Power regulators convert the 11.1v to lower.
- Servo connectors for the arms.
- Voltage Monitoring Circuit.
- Fuse sockets for a safe and reliable connection.
- Decoupling Capacitors.
- Reverse Polarity Protection.
- Thermistor for regulating the DC cooling fan.

Features:

1. Balance:
The robot uses an MPU9250 IMU and closed-loop control to maintain real-time balance by adjusting wheel speed and direction, even when carrying loads or on uneven surfaces.
2. Speech and Audio Enabled:
With speakers and an amplifier, the robot provides audio feedback and can respond with spoken commands. An auxiliary microphone allows for future voice control.
3. Interactive User Interface:
A 7-inch touchscreen provides an intuitive display for control and monitoring, enhancing user interaction with real-time system data.

List of Components

A. Electronics components

a. Electronic modules:

The electronic system of the robot is divided into three primary modules: Power, Low-level controller, and High-level processing. Each module is critical in ensuring the robot's performance and stability.

Power unit:
The power unit supplies energy to all the robot components, ensuring efficient operation and balance. It consists of

- **11.1V 10000mAh LiPo Battery:** Provides sufficient power for the robot's motors and electronics, allowing for about 45 minutes of operation.
- **Battery Level Indicator:** Monitors the battery's power levels to avoid over-discharge during operation.
- **Rocker Switch:** Manages power on/off for the entire system.
- **10 Amp Fuse & Chip Fuse Holder:** Protects the circuit from overcurrent, ensuring safety during operation.
- **XT60 Male Connector:** Ensures a secure and reliable power connection between the battery and the other components.

Low-Level Controller:

The low-level controller handles real-time control tasks such as motor control and sensor data processing, ensuring the robot maintains balance and quickly reacts to environmental changes. It consists of

- **Cytron URC10**: This motor controller is combined with an Arduino, which controls the robot's DC motors using sensor feedback.
- **MPU6050 (IMU)**: Provides essential data on the robot's orientation and angular velocity, enabling real-time balance control.
- **OE-37 Hall Encoder**: Measures the motor's rotation, providing feedback for speed and position control of the robot.
- **12V DC Motor (100 RPM, 173.6 N-cm**): Drives the robot's wheels with enough torque to maintain balance and stability driven by the motor driver (cytron)

High-level processing:

The high-level processing unit manages the robot's operations, advanced computations, and user interactions.

- **Raspberry Pi 4 (8GB RAM)**: Handles high-level processing tasks like decision-making, user interaction, and managing sensors. It also communicates with the low-level controller.
- **7-inch LCD Touch Screen Display**: Provides an interface for user interaction, displaying vital information and controls.
- **Auxiliary Microphone & JBL A140 140W Coaxial Dual Speaker**: Used for voice commands and audio output, enabling user-robot interaction.
- **Raspberry Pi IQAudio DGAmp+**: Amplifies audio signals for the speaker, ensuring clear and loud output.
- **Ultrasonic Sensor**: Detects obstacles in the robot's path and provides distance feedback for autonomous navigation.

Sr.No	Name of the Component	Quantity	Make/Company
1	Raspberry Pi 4 8GB RAM	1	Raspberry Pi Foundation
2	7-inch LCD Touch Screen Display	1	Constflick Technologies
3	Auxiliary Microphone	1	MAONO
4	Raspberry PI IQAudio DGAmp+	1	Texas Instruments
5	JBL A140 140W, Coaxial Dual Speaker	1	JBL
6	12VDC 1.66W Cooling Fan	1	Sunon
7	Battery level indicator	1	Generic
8	Rocker Switch	1	Generic
9	10 Amp Fuse	1	Generic
10	Chip fuse holder	1	Generic
11	2:6 Pole Connector	1	Generic
12	XT60 Male Connector	1	Generic
13	Cytron URC10	1	Cytron Technologies
14	OE-37 Hall Encoder	2	Orange
17	Barrel Jack connector cable	1	Generic
18	USB to Micro USB cable	1	Generic
19	11.1V 10000mAh Lipo Battery	1	Orange Brand
19	IMAX B6 Lipo Battery Charger	1	SkyRC
20	Silicon 12-20 AWG wires	Many	Generic

List of Tools/Equipment Used:

Sr.No	Tools

1	Soldering kit
2	Multimeter
3	Lipo Battery Charger
4	Wire stripper

B. Mechanical Components

a. Mechanical Module:

The robot's mechanical structure consists of

Chassis and Frame Design:
The robot's frame is designed to support all electronic components

- **Steel Bottom Plate (Laser-Cut):** The bottom-most layer is a laser-cut steel plate, providing a sturdy base for the robot. The weight of this plate helps lower the center of gravity, enhancing the stability of the balancing mechanism.
- **3D-Printed Middle and Top Plates:** The upper layers are made from 3D-printed parts designed to reduce overall weight while providing structural support for the mounted components. These plates house critical electronics like the battery, motor controller, and sensors.
- **3D-Printed Extenders and Carbon Fiber Rods:** The layers of the robot are connected using 3D-printed extenders, and additional carbon fiber rods are incorporated to reinforce the structure. Carbon fiber rods are lightweight yet strong, ensuring the overall rigidity of the structure without adding excess weight.

Wheels and Motor Mounts: The robot’s movement is powered by two high-torque DC motors, and the choice of wheels plays a crucial role in ensuring smooth and stable motion.The motors are securely mounted to the frame with 10mm M4 bolts to ensure stability during high-speed movements

Outer Frame: The 3d printed outer frame houses various components such as display, fan, indicator, switch, etc.

Sr.No	Name of the Component	Quantity	Make/Company
1	3d printed outer body parts	6	3D additive printing

2	3D Printed plates	4	3D additive printing
4	152 mm Rubber wheel	2	Plastic & rubber
5	Motor mounts	2	Mild Steel / EasyMech Brand
6	15mm M3 bolts & nuts for outer body	20	Mild Steel / EasyMech Brand
7	10mm M4 for motors	12	Mild Steel / EasyMech Brand
8	Anti Slip Motor Coupling	2	Aluminium/ EasyMech Brand
9	hex bolts and nuts (20 *M4)	28	Nylon
10	M4 15mm Bolts Nuts for connecting links	25	Mild steel

b. Fabrication Components:

- 3D-printed plates for structure.
- Laser-cut steel bottom plate.
- Carbon fiber rods for extra strength and stability
- 3D-printed extenders to connect the plates.

c. List of Tools/Equipment Used:

Sr.No	Tools
1	3D Printer
2	Laser cutting Machine
3	Power Drill
4	Screwdrivers, Plyers

C.Electromechanical Components:

Sr.No	Name of the Component	Quantity	Make/Company

1	12v DC motor 100 RPM, 173.6 N-cm	2	Orange Brand
2	Ultrasonic Sensor	1	M5 STACK
3	MPU6050 (IMU)	1	InvenSense's

Software

Software Used:

1. Arduino
2. MATLAB (initial simulation)
3. ROS (Robot Operating System)
4. Python
5. Some well-known Libraries and packages

Software Developed:

1. IMU RAW
2. IMU Calibration (Code for calibration: link)
3. Gyro angle calculations
4. Acceleration angle calculations
5. Complementary angle calculations
6. Basic robot self-balancing robot code (Including Motor control)
7. Adding closed-loop control using the motor feedback
8. Updated Libraries using custom functions for our own controllers and using interrupts for a better loop call
9. RPI Codes, Speakers, MIC codes
10. TTS and STT codes
11. Emotion animation in the display using the pygame or any other way (link)
12. Joystick control code (RPI)

IMU raw data after calibration	**Angle calculation using acc data**
#include "Wire.h" #include "I2Cdev.h" #include "MPU6050.h" #include "math.h"	#include "Wire.h" #include "I2Cdev.h" #include "MPU6050.h" #include "math.h"

```
MPU6050 mpu;

int16_t accY, accZ, accX, gyroX, gyroY, gyroZ;

void setup() {
  mpu.initialize();
  Serial.begin(9600);
  mpu.setXAccelOffset(824.00000);
  mpu.setYAccelOffset(-4404.00000);
  mpu.setZAccelOffset(1589.00000);
  mpu.setXGyroOffset(87.00000);
  mpu.setYGyroOffset(-3.00000);
  mpu.setZGyroOffset(13.00000);
}

void loop() {
  accX = mpu.getAccelerationX();
  accY = mpu.getAccelerationY();
  accZ = mpu.getAccelerationZ();
  gyroX = mpu.getRotationX();
  gyroY = mpu.getRotationY();
  gyroZ = mpu.getRotationZ();

  Serial.print(accX);
  Serial.print(" ");
  Serial.print(accY);
  Serial.print(" ");
  Serial.print(accZ);
  Serial.print(" ");
  Serial.print(gyroX);
  Serial.print(" ");
  Serial.print(gyroY);
  Serial.print(" ");
  Serial.println(gyroZ);
}
```

Angle calculation using gyro data

```
#include "Wire.h"
#include "I2Cdev.h"
#include "MPU6050.h"
```

```
MPU6050 mpu;

int16_t accY, accZ;
float accAngle;

void setup() {
  mpu.initialize();
  Serial.begin(9600);
  mpu.setXAccelOffset(824.00000);
  mpu.setYAccelOffset(-4404.00000);
  mpu.setZAccelOffset(1589.00000);
  mpu.setXGyroOffset(87.00000);
  mpu.setYGyroOffset(-3.00000);
  mpu.setZGyroOffset(13.00000);
}

void loop() {
  accZ = mpu.getAccelerationZ();
  accY = mpu.getAccelerationY();

  accAngle = atan2(accY, accZ)*RAD_TO_DEG;

  if(isnan(accAngle));
  else
    Serial.print(accAngle);
    Serial.print("  ");
    Serial.println(accZ/16384.0);
}
```

Angle using a complementary filter

```
#include "Wire.h"
#include "I2Cdev.h"
#include "MPU6050.h"
```

```
MPU6050 mpu;

int16_t gyroX, gyroRate;
float gyroAngle=0;
unsigned long currTime, prevTime=0, loopTime;

void setup() {
  mpu.initialize();
  Serial.begin(9600);
  mpu.setXAccelOffset(824.00000);
  mpu.setYAccelOffset(-4404.00000);
  mpu.setZAccelOffset(1589.00000);
  mpu.setXGyroOffset(87.00000);
  mpu.setYGyroOffset(-3.00000);
  mpu.setZGyroOffset(13.00000);
}

void loop() {
  currTime = millis();
  loopTime = currTime - prevTime;
  prevTime = currTime;

  gyroX = mpu.getRotationX();
  gyroRate = gyroX/131.0; //map(gyroX, -32768,
32767, -250, 250)
  gyroAngle = gyroAngle +
(float)gyroRate*loopTime/1000;

  Serial.println(gyroAngle);
}
```

```
#include "math.h"

MPU6050 mpu;

int16_t accY, accZ;
float accAngle;
float currentAngle;
float previousAngle = 0.0;


int16_t gyroX, gyroRate;
float gyroAngle=0;
unsigned long currTime, prevTime=0, loopTime;

void setup() {
  mpu.initialize();
  Serial.begin(9600);
  mpu.setXAccelOffset(824.00000);
  mpu.setYAccelOffset(-4404.00000);
  mpu.setZAccelOffset(1589.00000);
  mpu.setXGyroOffset(87.00000);
  mpu.setYGyroOffset(-3.00000);
  mpu.setZGyroOffset(13.00000);
}

void loop() {
  currTime = millis();
  loopTime = currTime - prevTime;
  prevTime = currTime;

  accZ = mpu.getAccelerationZ();
  accY = mpu.getAccelerationY();
  gyroX = mpu.getRotationX();
  gyroRate = gyroX/131.0;
  gyroAngle = (float)gyroRate*loopTime/1000;
  accAngle = atan2(accY, accZ)*RAD_TO_DEG;

  currentAngle = 0.9934 * (previousAngle +
gyroAngle) + 0.0066 * (accAngle);

  previousAngle = currentAngle;
```

```
  Serial.println(currentAngle);
}
```

Open Loop self-balancing code with a complementary filter

```
#include "Wire.h"
#include "I2Cdev.h"
#include "MPU6050.h"
#include "math.h"

#define leftMotorPWMPin   6
#define leftMotorDirPin   7
#define rightMotorPWMPin  5
#define rightMotorDirPin  4

#define Kp  30
#define Kd  0.01
#define Ki  0
#define sampleTime  0.005  // 5 milliseconds
#define targetAngle -1.7

MPU6050 mpu;

int16_t accY, accZ, gyroX;
volatile int motorPower, gyroRate;
volatile float accAngle, gyroAngle, currentAngle, prevAngle=0, error, prevError=0, errorSum=0;
unsigned long currTime, prevTime=0, loopTime;

void setMotors(int leftMotorSpeed, int rightMotorSpeed) {
 if (leftMotorSpeed >= 0) {
  analogWrite(leftMotorPWMPin, leftMotorSpeed);
  digitalWrite(leftMotorDirPin, LOW);
 }
 else {
  analogWrite(leftMotorPWMPin, abs(leftMotorSpeed));   digitalWrite(leftMotorDirPin, HIGH);
 }

 if (rightMotorSpeed >= 0) {
  analogWrite(rightMotorPWMPin, rightMotorSpeed);
  digitalWrite(rightMotorDirPin, LOW);
```

```
 }
 else {
  analogWrite(rightMotorPWMPin, abs(rightMotorSpeed));    digitalWrite(rightMotorDirPin, HIGH);
 }
 Serial.print(leftMotorSpeed);
 Serial.print(" ");
 Serial.print(rightMotorSpeed);
 Serial.print(" ");
 Serial.println(currentAngle);
}

void setup() {
 pinMode(leftMotorPWMPin, OUTPUT);
 pinMode(leftMotorDirPin, OUTPUT);
 pinMode(rightMotorPWMPin, OUTPUT);
 pinMode(rightMotorDirPin, OUTPUT);

 Serial.begin(9600);

 mpu.initialize();
 mpu.setXAccelOffset(824.00000);
 mpu.setYAccelOffset(-4404.00000);
 mpu.setZAccelOffset(1589.00000);
 mpu.setXGyroOffset(87.00000);
 mpu.setYGyroOffset(-3.00000);
 mpu.setZGyroOffset(13.00000);
}

void loop() {
 currTime = micros();
 float loopTime = (currTime - prevTime) / 1000000.0;  if (loopTime == 0) return;  // Avoid divide by zero
 prevTime = currTime;

 accZ = mpu.getAccelerationZ();
 accY = mpu.getAccelerationY();
 gyroX = mpu.getRotationX();
 gyroRate = gyroX/131.0; //map(gyroX, -32768, 32767, -250, 250)
 gyroAngle = (float)gyroRate*loopTime;
 accAngle = atan2(accY, accZ)*RAD_TO_DEG;

 currentAngle = 0.98 * (prevAngle + gyroAngle) + 0.02 * accAngle;
```

```
  error = currentAngle - targetAngle;

  errorSum += error;
  errorSum = constrain(errorSum, -100, 100);
  motorPower = Kp * error + Ki * errorSum * (loopTime) - Kd * (currentAngle - prevAngle) / (loopTime);
  motorPower = constrain(motorPower, -255, 255);


  setMotors(motorPower, motorPower);
  prevAngle = currentAngle;

}
```

Closed loop DC motor control with a hall effect quadrature encoder

```
#define LH_PWM 5
#define LH_DIR 4
#define RH_PWM 6
#define RH_DIR 7
#define LH_A 2
#define LH_B 8
#define RH_A 3
#define RH_B 9

volatile long LH_ticks;
volatile long RH_ticks;

bool LH_Direction;
bool RH_Direction;

unsigned long Prev_time = 0;

int Desired_Speed_LH;
int Current_Speed_LH;
float Prev_duration_LH;

int Desired_Speed_RH;
int Current_Speed_RH;
float Prev_duration_RH;

int T = 1000;
```

```
bool readingEnabled = false;
String inputString = "";
int incomingByte = 0;

#define Kp 1
#define Kd 0.01
#define Ki 0

int Prev_error_LH;
int cum_error_LH;
int Output_LH;

int Prev_error_RH;
int cum_error_RH;
int Output_RH;

void setup() {
  pinMode(LH_PWM, OUTPUT);
  pinMode(LH_DIR, OUTPUT);
  pinMode(RH_PWM, OUTPUT);
  pinMode(RH_DIR, OUTPUT);
  pinMode(LH_A, INPUT);
  pinMode(LH_B, INPUT);
  pinMode(RH_A, INPUT);
  pinMode(RH_B, INPUT);
  Serial.begin(9600);
  EncoderINIT();
  Serial.println("<Arduino is ready>");
}

void loop() {
  ReadData();
  if (Desired_Speed_LH >= 0) digitalWrite(LH_DIR, HIGH);
  else digitalWrite(LH_DIR, LOW);
  if (Desired_Speed_RH >= 0) digitalWrite(RH_DIR, HIGH);
  else digitalWrite(RH_DIR, LOW);
  Speed();
  analogWrite(LH_PWM, abs(Output_LH));
  analogWrite(RH_PWM, abs(Output_RH));
  Serial.print(LH_ticks);
```

```
  Serial.print(",");
  Serial.print(LH_Direction);
  Serial.print(",");
  Serial.print(RH_ticks);
  Serial.print(",");
  Serial.print(RH_Direction);
  Serial.print("Output_LH : ");
  Serial.print(Output_LH);
  Serial.print("Output_RH : ");
  Serial.println(Output_RH);
}

void ReadData() {
  while (Serial.available()) {
    char inputChar = Serial.read();
    if (inputChar == '<') {
      inputString = "";
      readingEnabled = true;
    } else if (inputChar == '>') {
      readingEnabled = false;
      processInput();
    } else if (readingEnabled) {
      inputString += inputChar;
    }
  }
}

void processInput() {
  int commaIndex = inputString.indexOf(',');
  if (commaIndex != -1) {
    String aString = inputString.substring(0, commaIndex);
    String bString = inputString.substring(commaIndex + 1);
    Desired_Speed_LH = aString.toInt();
    Desired_Speed_RH = bString.toInt();
  }
  inputString = "";
}

void Speed() {
  unsigned long Current_time = millis();
  float delta_time = (Current_time - Prev_time) / 1000;
```

```
 if (delta_time >= T / 1000) {
  int Current_duration_LH = LH_ticks;
  int Current_duration_RH = RH_ticks;
  float rotations_LH = (Current_duration_LH - Prev_duration_LH) / (7 * 50.1);
  float rotations_RH = (Current_duration_RH - Prev_duration_RH) / (7 * 50.1);
  Current_Speed_LH = rotations_LH * 60 / delta_time;
  Current_Speed_RH = rotations_RH * 60 / delta_time;
  if (Desired_Speed_LH >= 0) Current_Speed_LH = 245 / 92 * Current_Speed_LH;
  else Current_Speed_LH = -245 / 92 * Current_Speed_LH;
  if (Desired_Speed_RH >= 0) Current_Speed_RH = 245 / 92 * Current_Speed_RH;
  else Current_Speed_RH = -245 / 92 * Current_Speed_RH;
  int Current_error_LH = Desired_Speed_LH - Current_Speed_LH;
  int delta_error_LH = Current_error_LH - Prev_error_LH;
  cum_error_LH += Current_error_LH;
  Output_LH = Kp * Current_error_LH + Kd * delta_error_LH / delta_time + Ki * cum_error_LH * delta_time;
  Prev_error_LH = Current_error_LH;
  int Current_error_RH = Desired_Speed_RH - Current_Speed_RH;
  int delta_error_RH = Current_error_RH - Prev_error_RH;
  cum_error_RH += Current_error_RH;
  Output_RH = Kp * Current_error_RH + Kd * delta_error_RH / delta_time + Ki * cum_error_RH * delta_time;
  Prev_error_RH = Current_error_RH;
  Prev_duration_LH = Current_duration_LH;
  Prev_duration_RH = Current_duration_RH;
  Prev_time = Current_time;
 }
}

void EncoderINIT() {
 attachInterrupt(digitalPinToInterrupt(LH_A), Encoder_callback_left, RISING);
 attachInterrupt(digitalPinToInterrupt(RH_A), Encoder_callback_right, RISING);
}

void Encoder_callback_left() {
 if (digitalRead(LH_B) == 0) {
  LH_Direction = true;
  LH_ticks++;
 } else {
  LH_Direction = false;
  LH_ticks--;
 }
}
```

```
void Encoder_callback_right() {
  if (digitalRead(RH_B) == 0) {
    RH_Direction = true;
    RH_ticks++;
  } else {
    RH_Direction = false;
    RH_ticks--;
  }
}
```

Self-balancing code with a Closed-loop motor control with encoder feedback

```
#include "Wire.h"
#include "I2Cdev.h"
#include "MPU6050.h"
#include "math.h"

#define leftMotorPWMPin   6
#define leftMotorDirPin   7
#define rightMotorPWMPin  5
#define rightMotorDirPin  4

#define rightEncoderPinA  9
#define rightEncoderPinB  10
#define leftEncoderPinA   2
#define leftEncoderPinB   3

#define Kp  30
#define Kd  0.01
#define Ki  0
#define sampleTime  0.005
#define targetAngle -1.7

#define speedKp  1.0
#define speedKi  0.01
#define speedKd  0.1

MPU6050 mpu;

volatile long rightEncoderCount = 0;
volatile long leftEncoderCount = 0;
```

```
volatile int motorPower, gyroRate;
volatile float accAngle, gyroAngle, currentAngle, prevAngle = 0, error, prevError = 0, errorSum = 0;
unsigned long currTime, prevTime = 0, loopTime;
volatile float rightMotorSpeed = 0, leftMotorSpeed = 0, rightSpeedError = 0, leftSpeedError = 0,
rightSpeedErrorSum = 0, leftSpeedErrorSum = 0;
volatile float prevRightSpeedError = 0, prevLeftSpeedError = 0;
volatile long prevRightEncoderCount = 0, prevLeftEncoderCount = 0;

void rightEncoderISR() {
 if (digitalRead(rightEncoderPinB) == HIGH) {
  rightEncoderCount++;
 } else {
  rightEncoderCount--;
 }
}

void leftEncoderISR() {
 if (digitalRead(leftEncoderPinB) == HIGH) {
  leftEncoderCount++;
 } else {
  leftEncoderCount--;
 }
}

void setMotors(int leftMotorSpeed, int rightMotorSpeed) {
 if (leftMotorSpeed >= 0) {
  analogWrite(leftMotorPWMPin, leftMotorSpeed);
  digitalWrite(leftMotorDirPin, LOW);
 } else {
  analogWrite(leftMotorPWMPin, abs(leftMotorSpeed));
  digitalWrite(leftMotorDirPin, HIGH);
 }

 if (rightMotorSpeed >= 0) {
  analogWrite(rightMotorPWMPin, rightMotorSpeed);
  digitalWrite(rightMotorDirPin, LOW);
 } else {
  analogWrite(rightMotorPWMPin, abs(rightMotorSpeed));
  digitalWrite(rightMotorDirPin, HIGH);
 }
}
```

```
void setup() {
  pinMode(leftMotorPWMPin, OUTPUT);
  pinMode(leftMotorDirPin, OUTPUT);
  pinMode(rightMotorPWMPin, OUTPUT);
  pinMode(rightMotorDirPin, OUTPUT);

  pinMode(rightEncoderPinA, INPUT);
  pinMode(rightEncoderPinB, INPUT);
  pinMode(leftEncoderPinA, INPUT);
  pinMode(leftEncoderPinB, INPUT);

  attachInterrupt(digitalPinToInterrupt(rightEncoderPinA), rightEncoderISR, CHANGE);
  attachInterrupt(digitalPinToInterrupt(leftEncoderPinA), leftEncoderISR, CHANGE);

  Serial.begin(9600);

  mpu.initialize();
  mpu.setXAccelOffset(824.00000);
  mpu.setYAccelOffset(-4404.00000);
  mpu.setZAccelOffset(1589.00000);
  mpu.setXGyroOffset(87.00000);
  mpu.setYGyroOffset(-3.00000);
  mpu.setZGyroOffset(13.00000);
}

void loop() {
  currTime = micros();
  float loopTime = (currTime - prevTime) / 1000000.0;
  if (loopTime == 0) return;
  prevTime = currTime;

  accZ = mpu.getAccelerationZ();
  accY = mpu.getAccelerationY();
  gyroX = mpu.getRotationX();
  gyroRate = gyroX / 131.0;
  gyroAngle = (float)gyroRate * loopTime;
  accAngle = atan2(accY, accZ) * RAD_TO_DEG;

  currentAngle = 0.98 * (prevAngle + gyroAngle) + 0.02 * accAngle;
```

```
  error = currentAngle - targetAngle;
  errorSum += error;
  errorSum = constrain(errorSum, -100, 100);
  motorPower = Kp * error + Ki * errorSum * loopTime - Kd * (currentAngle - prevAngle) / loopTime;
  motorPower = constrain(motorPower, -255, 255);

  long rightEncoderTicks = rightEncoderCount - prevRightEncoderCount;
  long leftEncoderTicks = leftEncoderCount - prevLeftEncoderCount;

  rightMotorSpeed = (rightEncoderTicks / loopTime) * 0.01;
  leftMotorSpeed = (leftEncoderTicks / loopTime) * 0.01;

  prevRightEncoderCount = rightEncoderCount;
  prevLeftEncoderCount = leftEncoderCount;

  rightSpeedError = motorPower - rightMotorSpeed;
  leftSpeedError = motorPower - leftMotorSpeed;

  rightSpeedErrorSum += rightSpeedError;
  leftSpeedErrorSum += leftSpeedError;

  int rightMotorCorrection = speedKp * rightSpeedError + speedKi * rightSpeedErrorSum * loopTime + speedKd
* (rightSpeedError - prevRightSpeedError) / loopTime;
  int leftMotorCorrection = speedKp * leftSpeedError + speedKi * leftSpeedErrorSum * loopTime + speedKd *
(leftSpeedError - prevLeftSpeedError) / loopTime;

  rightMotorCorrection = constrain(rightMotorCorrection, -255, 255);
  leftMotorCorrection = constrain(leftMotorCorrection, -255, 255);

  setMotors(leftMotorCorrection, rightMotorCorrection);

  prevRightSpeedError = rightSpeedError;
  prevLeftSpeedError = leftSpeedError;
  prevAngle = currentAngle;
}
```

Interface Module:

ROS (Robot Operating system) will be used for seamless audio and speech data through a developed ROS framework. The SST and TTS modules will further use this data. We'll use an Amazon Alexa API as a Language processing module.

We will utilize **Google** Cloud's Speech-to-Text (STT) and Text-to-Speech (TTS) services. These powerful tools will enable the robot to understand spoken commands and respond verbally, enhancing user interaction and creating a more intuitive experience. By leveraging these APIs, we can effectively convert spoken language into text and synthesize natural-sounding speech from text, making our assistant both responsive and engaging. This integration will streamline communication and allow for a range of functionalities that can adapt to user needs.

In later stages, we'll be using the open-source indic language models from **ai4barath**, which is an initiative launched by IIT Madras with the goal of advancing artificial intelligence (AI) technologies for Indian languages. It has support for the top 15 Indian languages spoken by far. We're more interested in this because we want to deploy these robots in Indian hospitals where the patients, practitioners, etc, are locals or wish to speak in their mother tongue.

Emotion animation in the display:
We'll be using articubot_one_ui (GitHub) and modifying it a bit for our application. It provides a graphical user interface (GUI) for interacting with the robot.
Main functions:

- Provides a user interface for controlling and monitoring a ROS robot
- Allows users to send commands and receive feedback from the robot
- Supports various ROS functionalities and features

Joystick control code: We'll be using joy_node pkg in ROS.

```
#!/usr/bin/env python3

import rospy
from geometry_msgs.msg import Twist
import serial
import time

# Open the serial connection to the Arduino
ser = serial.Serial("/dev/ttyUSB0", 9600)
```

```
# Wheel velocities
front_left_w = front_right_w = rear_left_w = rear_right_w = 0.0

# Marker values for communication
START_MARKER = 60  # Unicode code for <
END_MARKER = 62    # Unicode code for >

# Robot parameters
max_speed = 10000
wheel_radius = 37.0  # mm
robot_width = 475.0   # mm
separation_between_wheels = 50  # mm

def recv_from_arduino():
    received_data = ""
    received_char = "!"

    while ord(received_char) != START_MARKER:
        received_char = ser.read()

    while ord(received_char) != END_MARKER:
        if ord(received_char) != START_MARKER:
            received_data += received_char.decode('utf-8')
        received_char = ser.read()

    return received_data

def wait_for_arduino():
    print("Waiting for Arduino to be ready")
    msg = ""
    while "Arduino is ready" not in msg:
        while ser.in_waiting == 0:
            if rospy.is_shutdown():
                return False
        msg = recv_from_arduino()
    return True

def turn_off():
    print('Turning off hardware interface node')
    ser.write("<0,0>".encode('utf-8'))
    time.sleep(3)
```

```
    ser.close()

def convert_vel_cmd(msg):
    global front_left_w, front_right_w, rear_left_w, rear_right_w
    x_vel = msg.linear.x
    y_vel = msg.linear.y
    w_vel = msg.angular.z

    front_left_w = (1/wheel_radius) * (x_vel - y_vel - (robot_width + separation_between_wheels) * w_vel)
    front_right_w = (1/wheel_radius) * (x_vel + y_vel + (robot_width + separation_between_wheels) *
w_vel)
    rear_left_w = (1/wheel_radius) * (x_vel + y_vel - (robot_width + separation_between_wheels) * w_vel)
    rear_right_w = (1/wheel_radius) * (x_vel - y_vel + (robot_width + separation_between_wheels) * w_vel)

def main():
    global ser, front_left_w, front_right_w, rear_left_w, rear_right_w
    rospy.init_node('hw_interface_node_front_wheels')
    rospy.on_shutdown(turn_off)

    if not wait_for_arduino():
        return

    print('Hardware interface node running')
    rospy.Subscriber("/cmd_vel", Twist, convert_vel_cmd)

    rate = rospy.Rate(4)
    while not rospy.is_shutdown():
        output_string = f"<{int(front_left_w * max_speed)},{int(front_right_w * max_speed)}>"
        ser.reset_output_buffer()
        ser.write(output_string.encode('utf-8'))
        ser.reset_input_buffer()

        while ser.in_waiting == 0:
            if rospy.is_shutdown():
                return

        rate.sleep()

if __name__ == '__main__':
    try:
        main()
```

```
except rospy.ROSInterruptException:
  pass
```

Additional Features in Actual Robo-Making :

Final Prototype

Mechanical Features:

- **Servo-driven arms:** We will integrate servos to control the robot's arms, allowing for precise movements. These arms will play a key role in helping the robot recover from a fall. When an external force impacts the robot, the arms adjust to stabilize it, leveraging the moment of inertia for balance recovery. The arms will also be designed to handle more than 1 kg of external payload, adding functionality for lifting and manipulation.
- **Force Sensors on Shoulders:** Force sensors will be installed on the shoulder region to enable interactive movements. These sensors will detect contact with humans, especially when the patient is trying to get assistance while walking for a better recovery, enhancing the robot's ability to assist in tasks, especially in healthcare or interactive settings.

Electronics Features:

- **Camera and Detection Sensors:** A camera will be integrated to provide visual data for autonomous navigation, enabling real-time object detection, tracking, and interaction. Additionally, the robot will feature ultrasonic sensors that can trigger alarms when a person or obstacle comes into proximity, enhancing safety and interaction.
- **Dock Charging:** We will implement an automatic docking system for charging. The robot will be equipped with sensors and algorithms to autonomously navigate to a charging dock when battery levels are low, ensuring continuous operation without human intervention.
- **Advanced IMU Integration:** To ensure precise localization and balance, we will use an IMU (Inertial Measurement Unit) in conjunction with the servos and other sensors. The IMU will provide real-time data on the robot's orientation and movement, which will be processed by advanced control algorithms (non-linear control methods like MPC).

Software Features:

- **Nonlinear Control Model:** To address various disturbances, such as backlash or dynamic errors, we will implement a sophisticated nonlinear control algorithm. This

will ensure precise balance and movement control, even in the presence of unexpected external forces.

- **Autonomous Navigation and Line Following:** The robot will have autonomous navigation capabilities using a combination of odometry, IMU, and camera data. It will also feature line-following capabilities for guided navigation within predefined paths.
- **Teleoperation:** Remote control and teleoperation functionalities will be added, enabling users to operate the robot from a distance, ensuring flexibility in use cases such as remote assistance or surveillance.
- **Extended Kalman Filter (EKF):** An EKF will be used to fuse data from odometry, the IMU, and the camera to improve the robot's localization and navigation accuracy. This will allow robust performance in dynamic environments, enhancing balance and movement.
- **Stability Algorithm with Moment of Inertia:** A novel algorithm will be incorporated to adjust the robot's arms based on balance error. By dynamically altering the position of the arms, the robot will improve its stability during movement and when impacted by external forces.

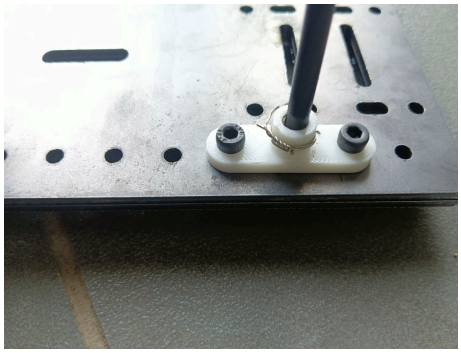
Concurrence / Deviation from Level1 document with reasoning

We have an upgrade and also a deviation from the ideation stage idea due to the rules of the grand finale that we got recently; we can't use the novel idea that we came up with in the ideation stage, the primary usage of the inertia adjustment mechanism is for the robot to restore its balance after falling, without any support (for more details refer to the paper) since fast motion cannot be achieved while tracking the line. We're making our final robot relatively larger, which is the most challenging part of self-balancing because Weight is a primary and crucial factor in designing a self-balancing robot.
The 3-layer structure was introduced for better integration of components and enhanced stability, rather than the initial simpler design.

- **Shifting the weight to the top of the robot:**
  The weight was focused on the bottom of the ideation. However, we shifted the weight to the top after concluding that concentrating the weight on the top of the robot enhanced the performance through iterations of experiments since the rate of fall was relatively reduced. This is because $\ddot{\theta}$ it is inversely proportional to l.
- **Addition of carbon fiber rods:**
  Adding carbon fiber rods was not part of the initial idea but was necessary to strengthen the frame and improve balance. Using hollow carbon fiber rods and 3d printed extenders instead of solid connecting rods further **reduced the overall weight** of the robot while

maintaining the same level of structural integrity, allowing for faster speeds and longer battery life without compromising balance or strength.

- **Connection Mechanisms for Carbon Fiber Rods:**
  Carbon rods are attached to the frame using a 3d printed connecting mount. The rods are slid through the mount, and a strip passes through the through-hole of the rod and the 3D-printed part to secure them together.



Weight optimization of the metal platform plates: Optimized design for the base and top plates of the robot, which led to weight reduction.

Any Other description not mentioned above

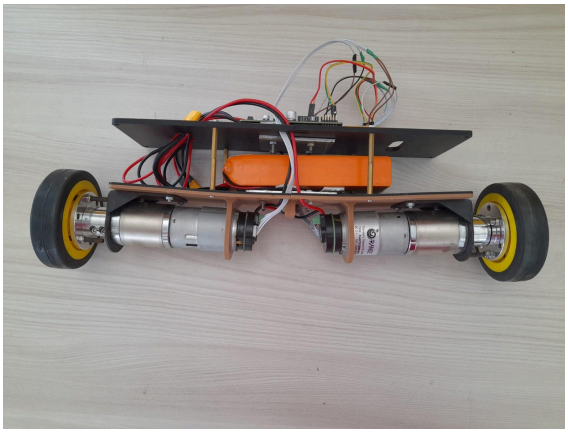
Issues faced:

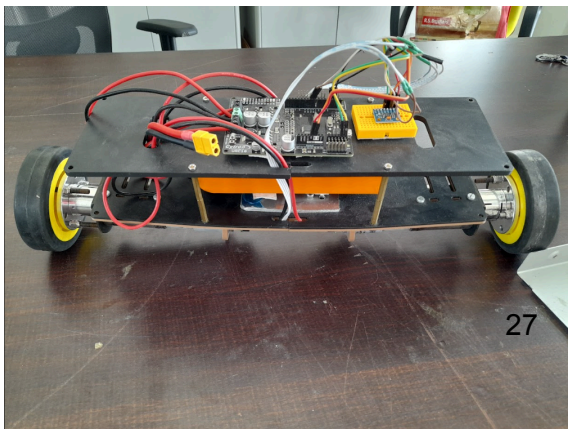
Backlash in motors:

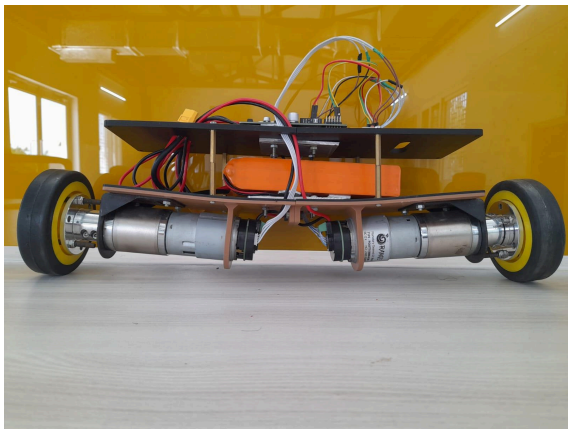
- **Delayed Wheel Response:** Due to the backlash, the wheels do not immediately respond to the motor's instructions, leading to an imbalance as the robot may lean too far in one direction before the wheels begin correcting it.
- **Erratic Movements:** As the robot tries to adjust its position, the delay can cause it to overcompensate, leading to a cycle of unstable and jerky movements.
- **Reduced Accuracy in Control:** The imbalance caused by backlash affects the robot's ability to make fine movements, especially at low speeds or when performing sensitive tasks like following a line while balancing.

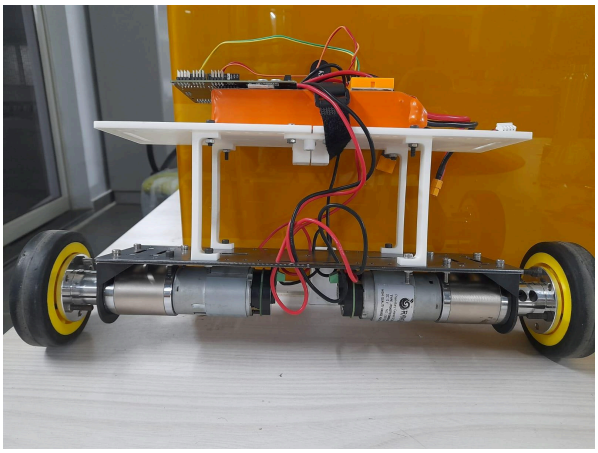
Photos of Robo-Making

Robot iteration 1:

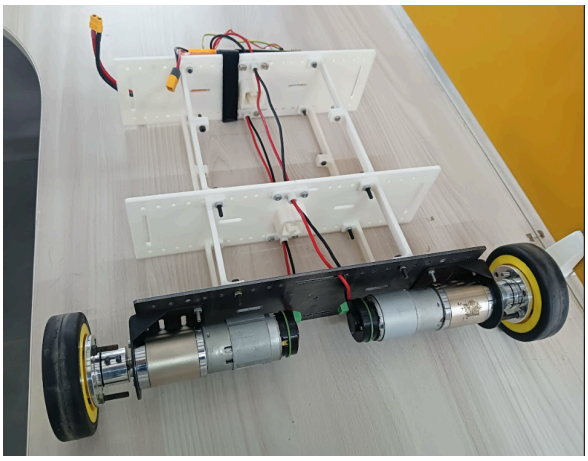


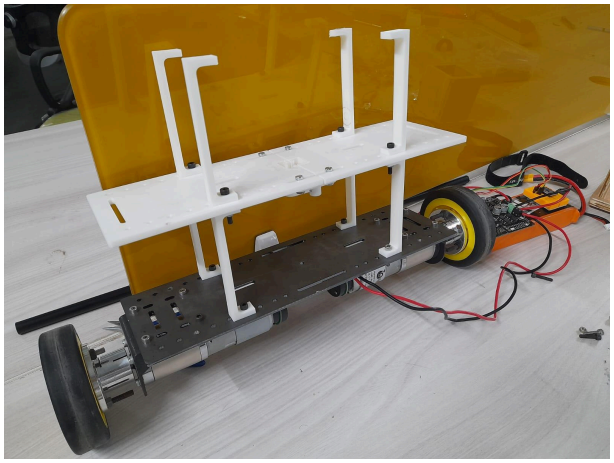


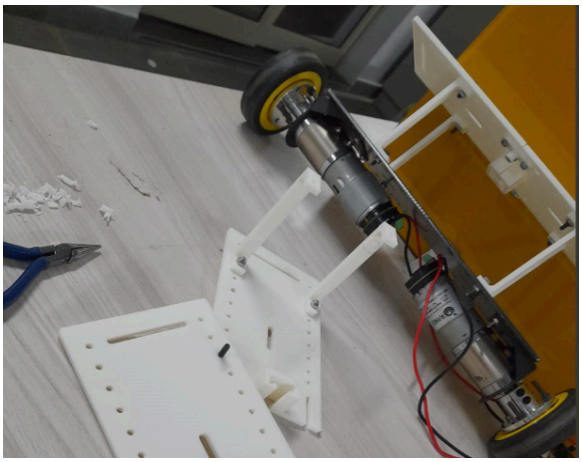


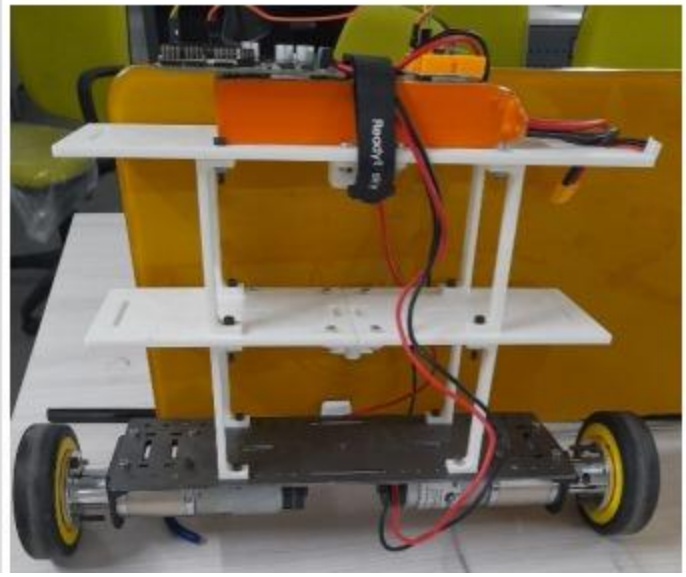


Robot iteration 2:

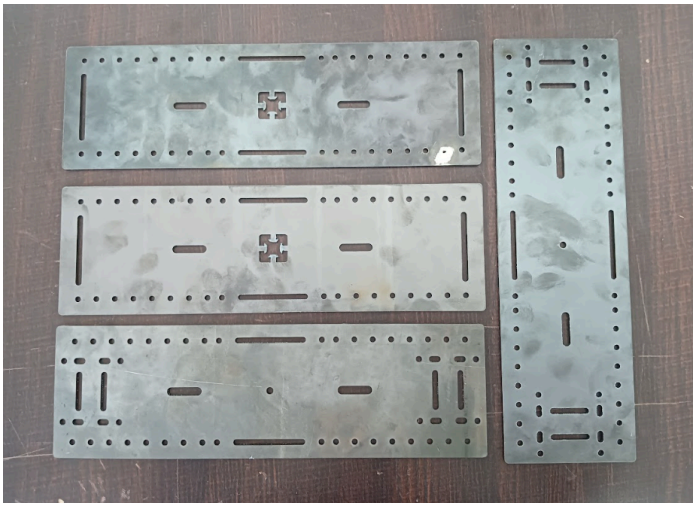


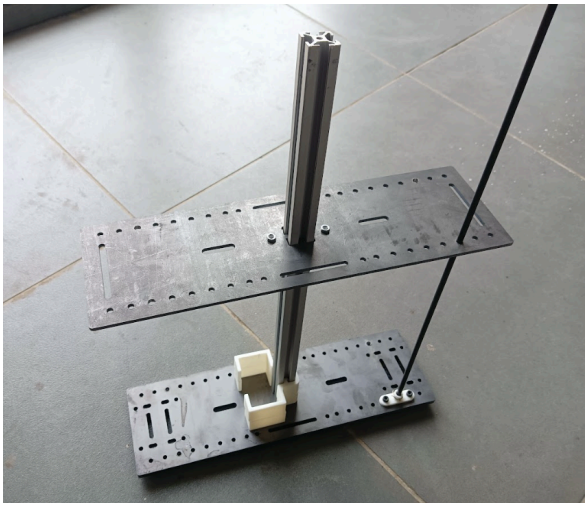


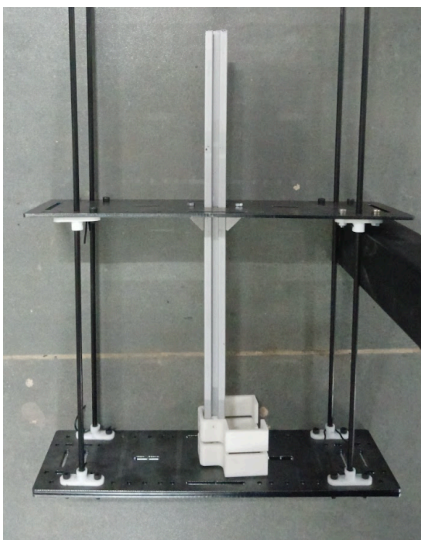


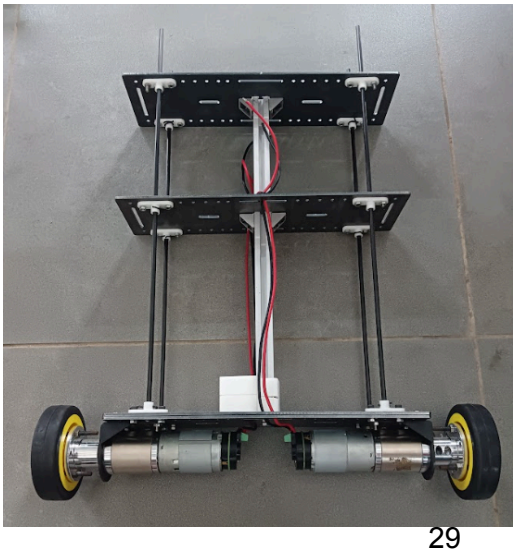


Towards the Final Robot:









Videos

https://drive.google.com/file/d/1Bvk-XDr4VcHhfs8b6nHjHAHrGA7pVVKE/view?usp=sharing
https://drive.google.com/file/d/1C9eW48nWljdvD4dgAfJb3Rck-8BpSBZL/view?usp=sharing
https://drive.google.com/file/d/13eLFOpOb4_bq0W5i6vQRqEZMvrCCU8RV/view?usp=sharing
https://drive.google.com/file/d/1w1ItHoNcsVdgyb87j3-47-N5Ur8aOHcJ/view?usp=sharing

Thank You.