ACKNOWLEDGEMENT

“In the name of ALLAH the most Beneficial and the most Merciful.”

First and foremost, I thank Almighty Allah for giving me this wonderful chance to write my report. Glory to the Almighty, who created and proportioned, who destined and then guided, and who taught man what he did not know. Then there are my parents' efforts, who taught me to persevere in the face of adversity and to never let my guard down. And it is only through hard work and consistent efforts that I have arrived at this stage, which is nothing short of a blessing for me. I'd want to express my gratitude to all of my instructors who have worked tirelessly to bring out the best in me, and it is because of them that I am able to finish my report without trouble. In addition, I would also like to thank some of my seniors in helping me throughout the difficult times. This report on “Complex Engineering Problem” is submitted to ENGINEER QURRATULAIN of IIEE by ALI HYDER SHAH, NUSAIBA KHALID, RUMMAN AHMED KHAN and ZAIN UL ABDIN.

ABSTRACT

The development of self-balancing robots provides a hands-on application of advanced feedback control systems, enabling students to apply theoretical knowledge to real-world problems. This project aims to design and implement a self-balancing robot using Proportional-Derivative (PD) and Proportional-Integral-Derivative (PID) control techniques to achieve stable upright positioning. By incorporating sensors such as gyroscopes and accelerometers, the system continuously monitors its tilt and adjusts the motor speeds to correct deviations from the vertical. Throughout the course, students analyze system dynamics, model the robot's behavior, and tune the controller to ensure quick and precise responses. This project not only reinforces key concepts in control theory but also introduces practical challenges in sensor integration, real-time feedback, and system stability. The outcomes of this project highlight the balance between theoretical understanding and practical implementation in feedback control systems. Recommendations for further refinement and enhancements are also discussed, ensuring that future iterations of the course continue to foster deep learning and innovation.

SELF BALANCING ROBOT:

The self-balancing robot is similar to an upside down pendulum. Unlike a normal pendulum which keeps on swinging once given a nudge, this inverted pendulum cannot stay balanced on its own. It will simply fall over. Then how do we balance it? Consider balancing a broomstick on our index finger which is a classic example of balancing an inverted pendulum. We move our finger in the direction in which the stick is falling. Similar is the case with a self-balancing robot, only that the robot will fall either forward or backward. Just like how we balance a stick on our finger, we balance the robot by driving its wheels in the direction in which it is falling. What we are trying to do here is to keep the center of gravity of the robot exactly above the pivot point.

To drive the motors, we need some information on the state of the robot. We need to know the direction in which the robot is falling, how much the robot has tilted and the speed with which it is falling. All this information can be deduced from the readings obtained from MPU6050. We combine all these inputs and generate a signal which drives the motors and keeps the robot balanced.

Introduction

This project focuses on the development of a two-wheeled, self-balancing robot that autonomously maintains its stability using an ESP32 microcontroller, L298N motor driver, and MPU6050 sensor. By continuously monitoring the robot's tilt through sensor data and dynamically adjusting the motor speeds, the system ensures real-time balance. The control algorithm processes feedback from the MPU6050 to execute precise corrective actions, demonstrating core principles of feedback control systems. This project highlights the practical application of control theory, embedded systems, and sensor integration, offering valuable insights into the challenges and solutions of designing responsive, real-time balancing mechanisms.

COMPONENTS:

ESP32 Microcontroller

The ESP32 serves as the brain of the robot. It processes data from the MPU6050 sensor and executes the control algorithm to maintain balance. The ESP32 receives real-time feedback on the robot's orientation and calculates the necessary adjustments to the motor speeds. It also sends control signals to the L298N motor driver to achieve the desired motor actions. Its high processing power and wireless capabilities make it suitable for handling the real-time calculations required for balancing.

L298N Motor Driver

The L298N is responsible for controlling the direction and speed of the two DC motors based on signals from the ESP32. The motor driver acts as an interface between the microcontroller and the motors by amplifying the low-power control signals from the ESP32 into higherpower

signals that can drive the motors. The driver allows for precise control of the motors, enabling forward, backward, and speed adjustments to keep the robot balanced.

MPU6050 Sensor

The MPU6050 is a combined gyroscope and accelerometer sensor that measures the robot's tilt and angular velocity. The sensor provides critical real-time data on the robot's orientation (pitch and roll) to the ESP32. This information is essential for the control algorithm to determine if the robot is tilting and how much correction is needed. The sensor enables the robot to detect and respond to imbalances by feeding continuous feedback to the microcontroller.

Motors:

The two DC motors provide the physical movement necessary for balancing. Controlled by the L298N motor driver, they adjust the robot's speed and direction based on signals from the ESP32. The motors work together to move forward or backward, depending on the tilt, to restore balance. The smooth and quick response of the motors is critical for achieving effective balance in the system.

Additional Components

Other components used in the project include a power supply for the motors, resistors, capacitors, and possibly a voltage regulator to ensure stable operation of the electronics. These components were selected to support the main components and ensure reliable performance of the robot.

Together, these components form a feedback loop where the MPU6050 detects tilt, the ESP32 processes the data and calculates adjustments, and the L298N driver controls the motors to restore balance.

Electronics Connections

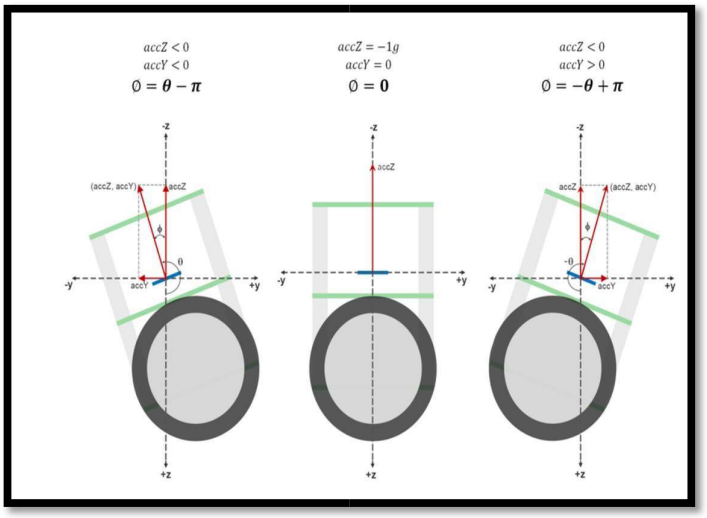
The electronics connections in this project were carefully designed to ensure proper functionality and safety. The ESP32 microcontroller was connected to the MPU6050 sensor via the I2C interface, allowing for real-time data exchange. The L298N motor driver was connected to the motors and the ESP32, with control signals being sent from the ESP32 to adjust the motor speed and direction. Power supply connections were also made to provide the necessary voltage and current to the motors and control circuitry.

Control Algorithm

The core of the balancing robot's functionality is the control algorithm, which reads data from the MPU6050 sensor and adjusts the motor outputs to maintain balance. A PID (ProportionalIntegral-Derivative) controller was implemented to process the sensor data and determine the appropriate motor commands. The algorithm continuously monitors the robot's tilt and corrects any deviations from the upright position, ensuring that the robot remains balanced.

The robot is built on three layers. The bottom layer contains the two motors and the motor driver. The middle layer has the has the battery and an on/off switch. The top most layer has the controller ESP32, and the MPU 6050 sensor.

1. Measuring Angle of Inclination



The MPU6050 sensor integrates a 3axis accelerometer and a 3-axis gyroscope, allowing it to measure both linear acceleration and angular velocity along all three axes. In the context of the self-balancing robot, the accelerometer is used to measure the acceleration along the y and z axes, which is essential for determining the robot's tilt. The angle of inclination is calculated using the atan2(y, z) function, which computes the angle in radians between the positive z-axis and the point defined by the coordinates (z, y). This function returns a positive value for counter-clockwise angles (y > 0) and a negative value for clockwise angles (y < 0). By using the DMP (Digital Motion Processor) of the MPU6050 to process and fuse the accelerometer and gyroscope data, the system ensures accurate and smooth angle measurements, which are crucial for maintaining the robot's balance. The code uses Jeff Rowberg's library to simplify reading and processing data from the MPU6050, making it easier to implement the control algorithm necessary for the robot's balancing mechanism.

CODE:

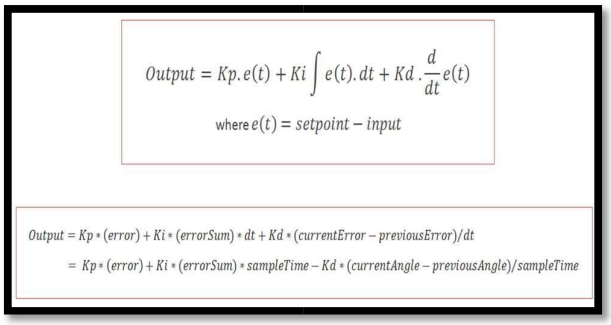
```
#include "I2Cdev.h"
#include "MPU6050_6Axis_MotionApps20.h"
#include "Wire.h"

MPU6050 mpu;
uint8_t devStatus; // return status after each device operation (0 = success,
!0 = error)

void setup() {
  // join I2C bus
  Wire.begin(21, 22); // ESP32 default SDA pin 21, SCL pin 22
  Wire.setClock(400000); // 400kHz I2C clock

  // initialize serial communication     Serial.begin(115200);
  while (!Serial); // wait for Serial to be ready
```

```
    // 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"));

    // wait for ready
    Serial.println(F("\nSend any character to begin DMP programming and demo:
"));     while (Serial.available() && Serial.read()); // empty buffer     while (!Serial.available());
// wait for data     while (Serial.available() && Serial.read()); // empty buffer again
    //         load         and         configure         the         DMP
Serial.println(F("Initializing  DMP..."));          devStatus  =
mpu.dmpInitialize();

    // make sure it worked (returns 0 if so)     if (devStatus == 0)
{
        // Calibration Time: generate offsets and calibrate our MPU6050
mpu.CalibrateAccel(6);        mpu.CalibrateGyro(6);        mpu.PrintActiveOffsets();
    } else {
        Serial.print(F("DMP Initialization failed (code "));
        Serial.print(devStatus);
        Serial.println(F(")"));
    }
}
void loop() {
    // Add your loop code here
}
```

The position of the MPU6050 when the program starts running is the zero inclination point. The angle of inclination will be measured with respect to this point. Keep the robot steady at a fixed angle and you will observe that the angle will gradually increase or decrease. It won't stay steady. This is due to the drift which is inherent to the gyroscope.

2. PID Control for Generating Output

PID stands for Proportional, Integral, and Derivative. Each of these terms provides a unique response to our self-balancing robot.



The proportional term, as its name suggests, generates a response that is proportional to the error. For our system, the error is the angle of inclination of the robot.

The integral term generates a response based on the accumulated error. This is essentially the sum of all the errors multiplied by the sampling period. This is a response based on the behavior of the system in past.

The derivative term is proportional to the derivative of the error. This is the difference between the current error and the previous error divided by the sampling period. This acts as a predictive term that responds to how the robot might behave in the next sampling loop.

Multiplying each of these terms by their corresponding constants (i.e, Kp, Ki and Kd) and summing the result, we generate the output which is then sent as command to drive the motor.

3. Tuning the PID Constants

- Set Ki and Kd to zero and gradually increase Kp so that the robot starts to oscillate about the zero position.
- Increase Ki so that the response of the robot is faster when it is out of balance. Ki should be large enough so that the angle of inclination does not increase. The robot should come back to zero position if it is inclined.
- Increase Kd so as to reduce the oscillations. The overshoots should also be reduced by now.
- Repeat the above steps by fine tuning each parameter to achieve the best result.

CODE:

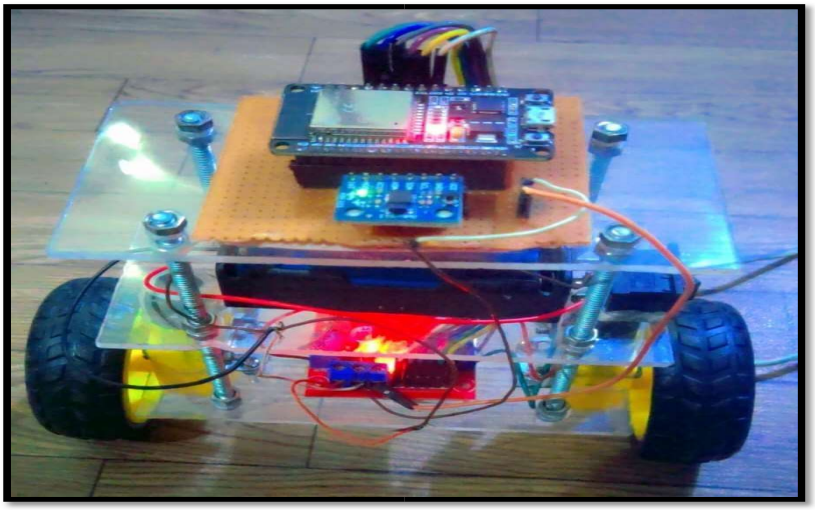
```
#include <Arduino.h>
#include <PID_v1.h>
#include "I2Cdev.h"
#include "MPU6050_6Axis_MotionApps20.h"
#include <Wire.h>
#include <driver/ledc.h>  // Ensure ESP32 LEDC functions are included

// Function prototypes void rotateMotor(int speed1, int
speed2);
// Define MPU6050 and necessary variables
MPU6050 mpu;
#define INTERRUPT_PIN 15 bool
dmpReady = false; uint8_t
mpuIntStatus; uint8_t devStatus;
uint16_t packetSize; uint16_t
fifoCount; uint8_t fifoBuffer[64];
Quaternion q;
VectorFloat gravity;
float ypr[3];
VectorInt16 gy = {-3940,  -1193,  600 }; // Updated MPU values volatile bool
mpuInterrupt = false; void IRAM_ATTR dmpDataReady() { mpuInterrupt = true; }
```

```
// PID variables double setpointPitchAngle
= -2.2; double pitchGyroAngle = 0; double
pitchPIDOutput = 0; double
setpointYawRate = 0; double yawGyroRate
= 0; double yawPIDOutput = 0;
PID pitchPID(&pitchGyroAngle, &pitchPIDOutput, &setpointPitchAngle, 30, 105,
0.8, DIRECT);
PID yawPID(&yawGyroRate, &yawPIDOutput, &setpointYawRate, 0.5, 0.5, 0,
DIRECT);

// MPU6050 offset values void
setupMPUOffsets() {
mpu.setXAccelOffset(-3940);
mpu.setYAccelOffset(-1193);
mpu.setZAccelOffset(600);
mpu.setXGyroOffset(100);
mpu.setYGyroOffset(-6);
mpu.setZGyroOffset(-24);
}
// Motor control pins int
enableMotor1 = 16; int
motor1Pin1 = 5; int
motor1Pin2 = 18; int
motor2Pin1 = 19; int
motor2Pin2 = 23; int
enableMotor2 = 17;
 void setupPID() {   pitchPID.SetOutputLimits(-
255, 255);   pitchPID.SetMode(AUTOMATIC);
pitchPID.SetSampleTime(10);
yawPID.SetOutputLimits(-255, 255);
yawPID.SetMode(AUTOMATIC);
yawPID.SetSampleTime(10);
} void setupMotors() {   pinMode(motor1Pin1, OUTPUT);
pinMode(motor1Pin2, OUTPUT);   pinMode(motor2Pin1, OUTPUT);
pinMode(motor2Pin2, OUTPUT);   ledcAttachPin(enableMotor1, 0);
ledcSetup(0, 5000, 8);  // 5kHz PWM, 8-bit resolution
ledcAttachPin(enableMotor2, 1);   ledcSetup(1, 5000, 8);  // 5kHz PWM,
8-bit resolution   rotateMotor(0, 0);  // Ensure rotateMotor function is
correctly declared
}
void setupMPU() {
  Wire.begin(21, 22);  // SDA, SCL
Wire.setClock(400000); // 400kHz I2C clock
mpu.initialize();   setupMPUOffsets(); // Set the MPU
offsets   pinMode(INTERRUPT_PIN, INPUT);   devStatus
= mpu.dmpInitialize();   if (devStatus == 0) {
mpu.setDMPEnabled(true);     mpuIntStatus =
mpu.getIntStatus();     dmpReady = true;     packetSize =
mpu.dmpGetFIFOPacketSize();
```

```
  } else {
    Serial.println("DMP Initialization failed");
  } }
void setup() {   Serial.begin(115200);   setupMotors();      setupMPU();   setupPID();
attachInterrupt(digitalPinToInterrupt(INTERRUPT_PIN), dmpDataReady, RISING);
} void loop() {   if (!dmpReady) return;   if (mpu.dmpGetCurrentFIFOPacket(fifoBuffer)) {
mpu.dmpGetQuaternion(&q, fifoBuffer);     mpu.dmpGetGravity(&gravity, &q);
mpu.dmpGetYawPitchRoll(ypr, &q, &gravity);     mpu.dmpGetGyro(&gy, fifoBuffer);     yawGyroRate
= gy.z;     pitchGyroAngle = ypr[1] * 180/M_PI;     pitchPID.Compute();     yawPID.Compute();
rotateMotor(pitchPIDOutput + yawPIDOutput, pitchPIDOutput - yawPIDOutput);
  } } void rotateMotor(int speed1, int speed2) {   if
(speed1 < 0) {     digitalWrite(motor1Pin1, LOW);
digitalWrite(motor1Pin2, HIGH);
  } else {     digitalWrite(motor1Pin1, HIGH);
digitalWrite(motor1Pin2, LOW);
  }   if (speed2 < 0) {
digitalWrite(motor2Pin1, LOW);
digitalWrite(motor2Pin2, HIGH);
} else {     digitalWrite(motor2Pin1, HIGH);
digitalWrite(motor2Pin2, LOW);
  }   speed1 = abs(speed1);   speed2 =
abs(speed2);   speed1 = constrain(speed1, 0, 255);
speed2 = constrain(speed2, 0, 255);   ledcWrite(0,
speed1);   ledcWrite(1, speed2);
}
```

Testing and Calibration

After assembling the robot and implementing the control algorithm, extensive testing and calibration were performed. This involved tuning the PID controller to achieve the desired balance performance. Various test scenarios were used to evaluate the robot's ability to maintain balance under different conditions. Adjustments were made to the motor control parameters and sensor calibration to optimize the robot's performance.



Applications:

Self-balancing robots have several practical applications across various fields due to their stability, maneuverability, and efficiency. Here are some key applications:

1. **Personal Transportation**: Self-balancing robots form the basis for devices like Segways and hoverboards, which are used for personal transportation. These devices offer efficient, compact, and easy-to-use mobility solutions for short-distance travel.
2. **Robotics Education**: Self-balancing robots are widely used in education and research to teach principles of control systems, robotics, and embedded electronics. They provide hands-on experience in sensor integration, PID control, and system stability.
3. **Healthcare Assistance**: In hospitals, self-balancing robots can be used as mobility aids or autonomous assistants to transport medical supplies, equipment, or even patients across short distances.
4. **Surveillance and Security**: Self-balancing robots can be equipped with cameras and sensors to perform surveillance and monitoring tasks. Their ability to maneuver in tight spaces makes them ideal for patrolling large facilities, warehouses, or restricted areas.
5. **Agriculture**: These robots can be adapted for agricultural use, where they assist in monitoring crops, inspecting fields, and even performing tasks like planting and fertilizing with minimal human intervention.
6. **Industrial Automation**: In industrial settings, self-balancing robots can assist with material handling, quality inspection, and automated transport of goods in factories and warehouses.
7. **Telepresence**: Self-balancing robots can serve as mobile telepresence devices, allowing people to interact remotely by maneuvering the robot and communicating through video and audio in real-time.
8. **Autonomous Delivery Systems**: Companies are exploring self-balancing robots as part of autonomous delivery systems for last-mile delivery. Their ability to balance and navigate efficiently makes them suitable for delivering small packages or groceries in urban environments.
9. **Rehabilitation Devices**: In physical therapy, self-balancing robots can be used to support rehabilitation exercises, particularly for individuals with mobility issues, helping them regain balance and coordination.
10. **Entertainment and Toys**: Self-balancing robots are popular as interactive toys and entertainment robots, offering engaging experiences through their ability to perform complex maneuvers and maintain balance while moving.

These applications showcase the versatility of self-balancing robots in improving efficiency, mobility, and automation across various domains.

CONCLUSION:

This project successfully demonstrated the principles of control systems and embedded electronics through the development of a wheel-balancing robot. By integrating the ESP32 microcontroller, L298N motor driver, MPU6050 sensor, and other components, a functional and responsive robot was created. The project provided valuable insights into the challenges of real-time control and the importance of precise sensor data and algorithm tuning.

While the project was successful in demonstrating the principles of control systems and embedded electronics, there are areas where further improvement could enhance the robot's performance. Spending more time fine-tuning the PID constants would likely yield even smoother and more precise balancing. Additionally, the compact size of our robot posed challenges in achieving the level of stability that might be more easily attained in larger designs. Despite these limitations, I believe our robot performed admirably, successfully balancing. Though there's always room for refinement, we are proud of the results achieved through the hard work and dedication put into this project.