German University in Cairo

Mechatronics Engineering (MCTR601)

Self-Tuning PID Controller for

Autonomous Car Tracking in Urban Traffic

Name	ID	Lab Number
Yasmina Tamer Ramadan	52-1449	P-30
Mohamed Habbak	56-29298	P-29
Mohmed Senna	52-22657	P-34

Table of Contents

1.Project Description:

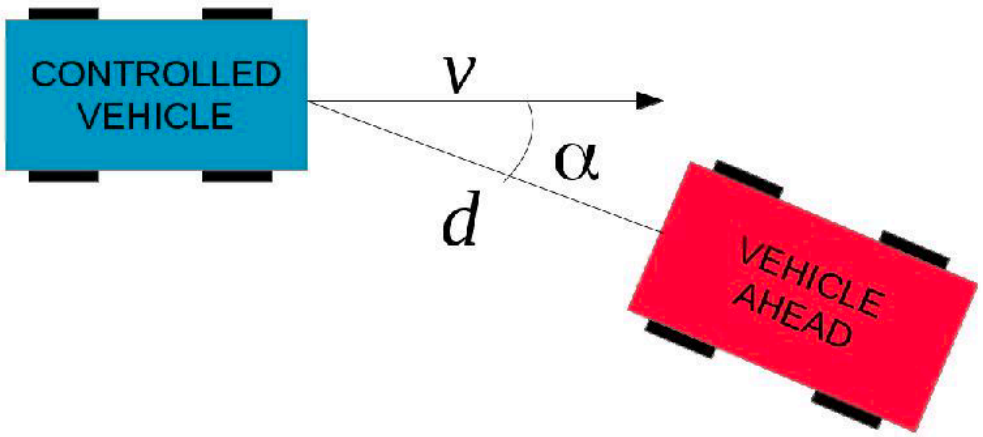
-Our project focuses on controlling the vehicle's steering direction using a self-tuned PID controller, based on data received wirelessly via Wi-Fi signals from the car ahead. We adjust our car's direction by utilizing IMU gyroscope data to obtain rad/sec readings, which are then integrated to determine the angle. Consequently, the vehicle's steering is adjusted based on the wireless signals received from the preceding car.

-Additionally, we have implemented a separate PID-based control system for obstacle avoidance using “Ultrasonic sensor “to maintain a safe distance from the car ahead. This system ensures that if the leading car approaches, our vehicle moves backward to preserve the safety distance.

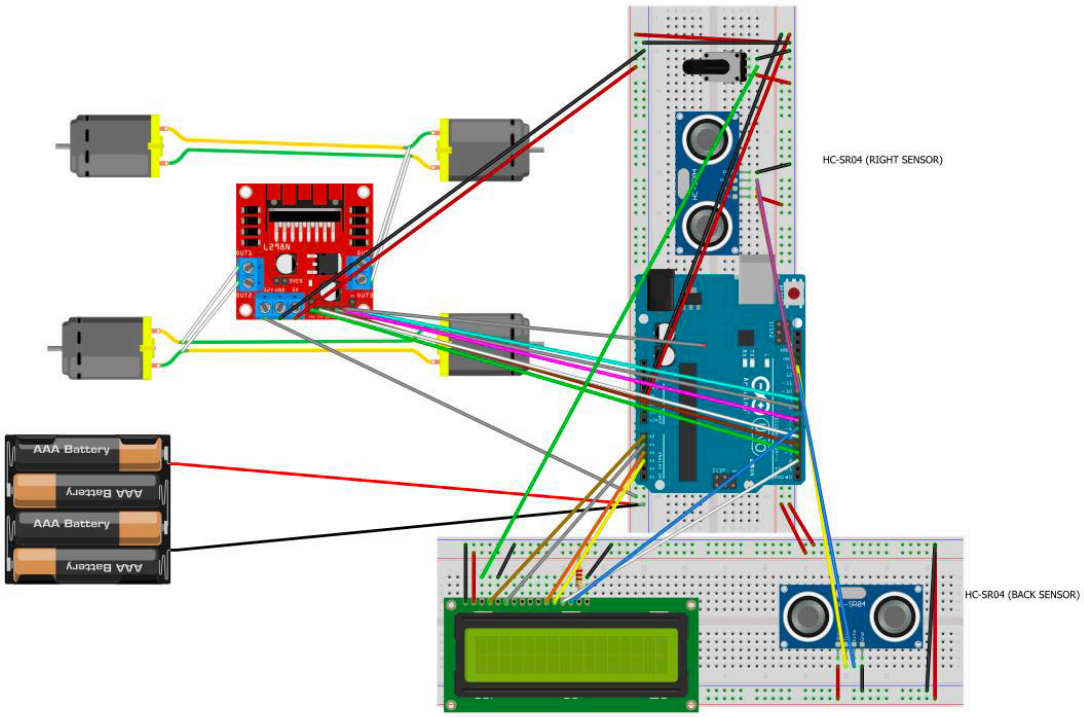
-Initially, we attempted to integrate both control systems into a single codebase. However, due to the complexity, we decided to separate the controls. We now manage each system independently, switching between them with a dedicated button.

-In conclusion, the vehicle-to-vehicle tracking system ensures maintaining an accurate and safe distance between our car and the one ahead. It also controls the steering direction using gyroscope sensor and PID closed-loop control laws.

Part number	Name
1-	H-bride (L298N)
2-	4 DC MOTORS
3- 4- 5- 6- 7-	ULTRASONIC SENSOR IMU(GYROSCOPE AND ACCELEMTER) BUTTONS LEDS BATTERIES 2 ESP32 WIRLESS MODULE



Done with fritting software :



v

2. Methodology

2.1 Mechanical design :

The vehicle to vehicle self tunning system control system utilizes a compact and robust 4-wheeled car chassis measuring 255mm in length, 115mm in width, and 66mm in height. It is equipped with four DC gearbox motors, one for each wheel, enabling precise movement control . The system's mechanical design also considers weight distribution and a low center of gravity. As no servo motors were used, the steering functionality was achieved in a skid

steering like manner. The turns took place my adjusting motor driver inputs and to achieve the desired direction of motion.With the use of IMU readings , we managed to get angle in order to adjust the steering control . Overall, the mechanical design provides a reliable structure for efficient and automated steering and safety distance control.

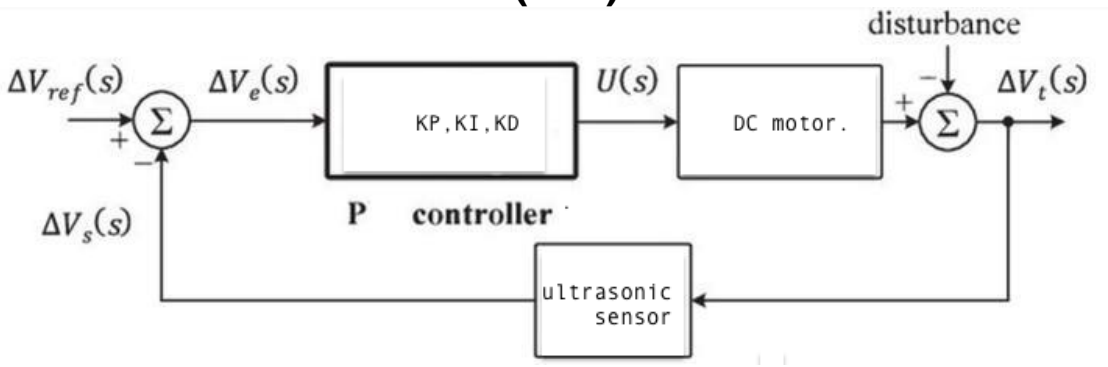
2.2 Electrical design :

The electrical design of the car revolves around the 2 ESP32- microcontrollers wirelessly, and their integration with the motor driver and sensors. The ESP32 serves as the central control unit, processing sensor data and sending signals to the H-bridge and accordingly the motors. Ultrasonic sensors are strategically placed on the car to provide distance measurements to detect obstacles as well as determine the desired distance needed to perform control to increase the system's reliability. These sensors are connected to the esp32 allowing it to receive and process the sensor readings. The ESP32 board interfaces with the H-bridge circuit, which in turn controls the DC motors responsible for propelling and steering the vehicle. The electrical connections between the ESP32, H-bridge, and motors are carefully established to ensure proper communication and efficient power distribution. Power management considerations, such as the use of appropriate voltage regulators and battery systems, are also incorporated into the design to ensure reliable and consistent operation of vehicle to vehicle tracking system.

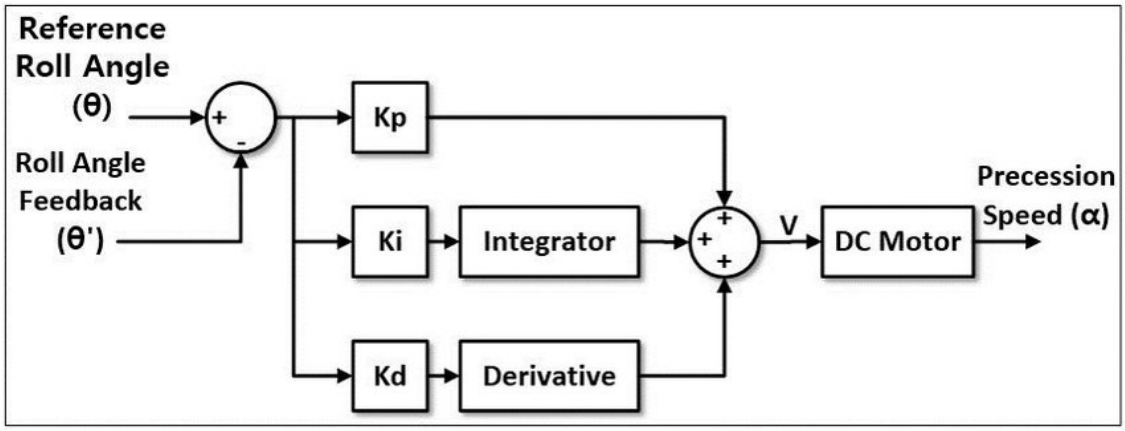
2.3.1. Modeling

Block diagram:

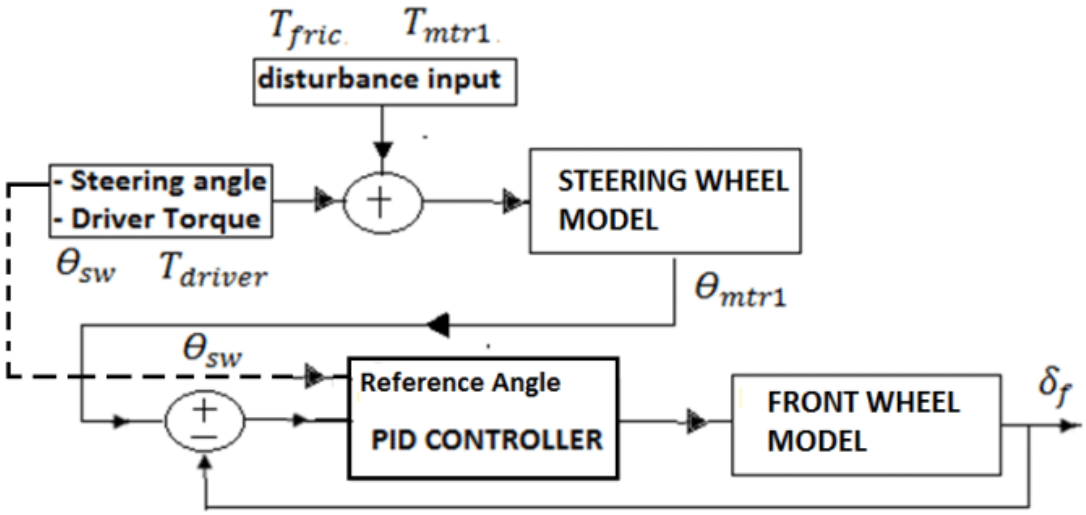
CONTROLLING DC MOTORS BASED ON ULTRASONIC SENSOR (PID)



IMU(gyroscope) sensor based DC MOTOR control law (PID):

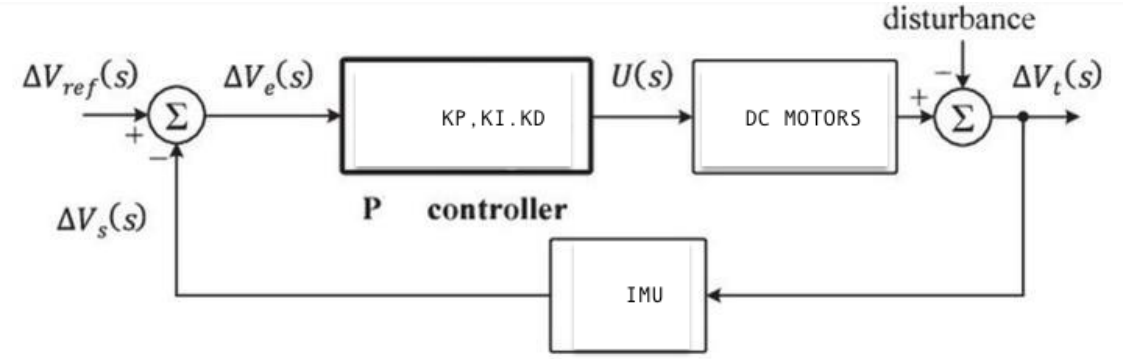


Another steering model :

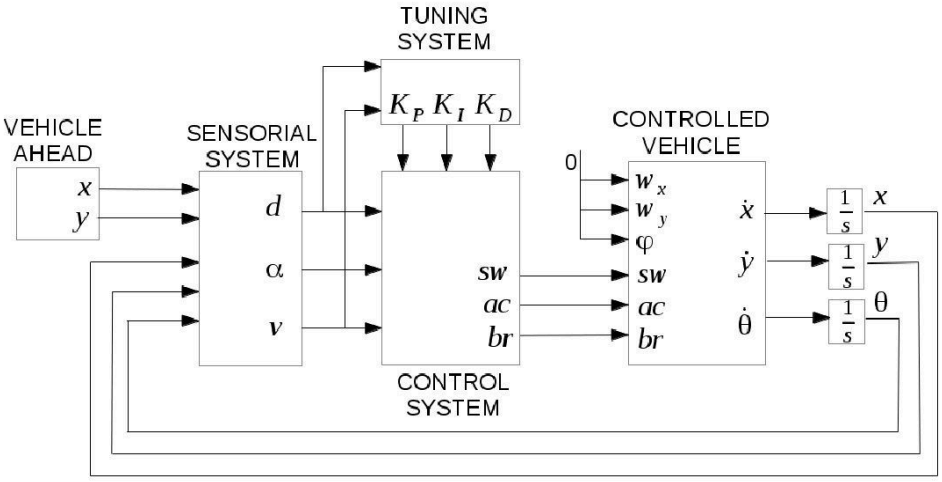


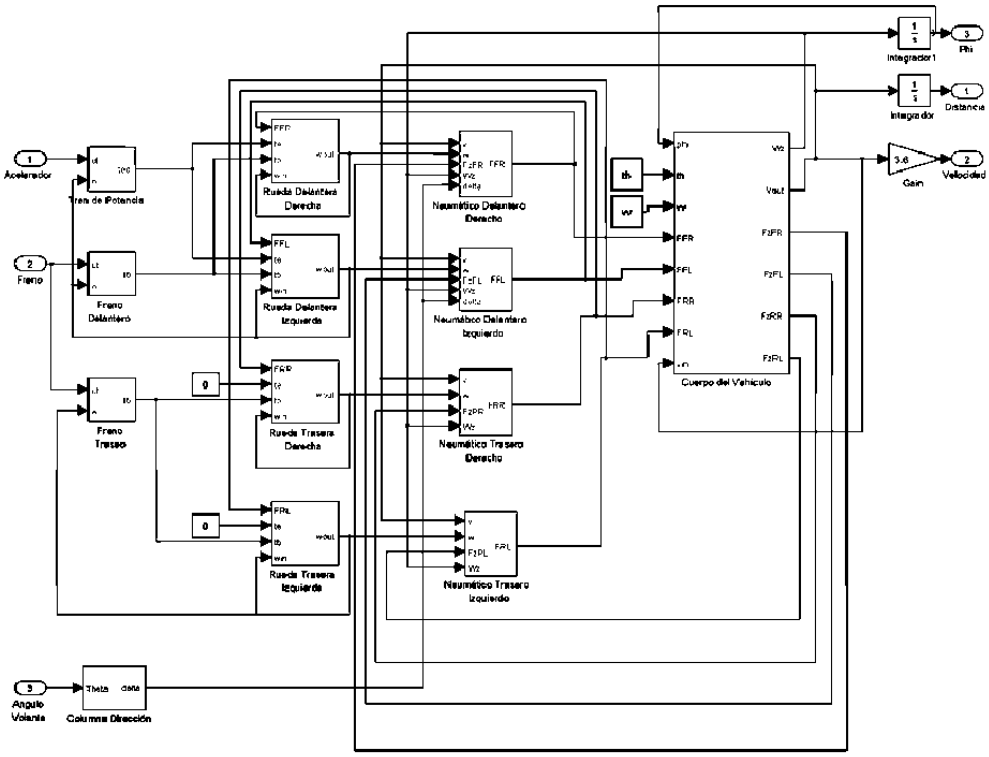
Maltlab/simulink Weblink :file:///Users/macbookpro/Downloads/PID%20MATLAB%20TO%20BE%20TESTED.html

Basic steering block diagram:



Initial Architecture of the system (mind map):





2.3.2. Controller Design :

PID (discretized) = Kp* ek + Kd* (ek-(ek-1))/ Δt + Ki* Σ ek (k=0 → k=n)

Control Law for Distance:

The PID controller computes the control output based on the error between the setpoint distance (setpointUS) and the measured distance (inputUS). The basic PID control law can be written as:

outputUS(t) = Kp * e(t) + Ki * ∫0^t e(τ) dτ + Kd * d/dt e(t)

Where:

e(t) = setpointUS - inputUS(t)

is the error at time t.

$e(t)$=setpointUS−inputUS(t) is the error at time

- Kp is the proportional gain.
- Ki is the integral gain.
- Kd is the derivative gain.

2. Steering Angle Control:

The steering control is handled by the **run_pid** function. The input to this PID controller is the angular displacement from the gyroscope (**input**), and the output (**output**) is used to control the car's steering.

The PID controller computes the control output based on the error between the setpoint angle (**setpoint**) and the measured angle (**input**). Assuming the setpoint is zero (desired angle is straight) .

output(t) = Kp * θ(t) + Ki * ∫0^t θ(τ) dτ + Kd * d/dt θ(t)

Where:

$\theta(t)$ = setpoint - input(t) = 0 - input(t) (since the desired angle is zero).

- Kp is the proportional gain.
- Ki is the integral gain.
- Kd is the derivative gain.

Initial Setup and Manual Tuning:

- Begin by tuning the controller gains manually through trial and error. Start with only proportional (P) control to keep the system simple and manageable.
- If necessary, proceed to proportional-derivative (PD) control to address any observed oscillations or transient responses.
- Use proportional-integral-derivative (PID) control only if essential, as the integral (I) term might introduce unwanted noise into the system.
-

Proportional Control Implementation:

The error is calculated as:
Error = Desired Value – Current Value

where the current readings are always sensor signals.

The control signal is computed as: Control Signal = Error × KP

The error is calculated as: Error = Desired Value – Current Value, where the curent readings are always sensor signals.

Error Handling and PWM Adjustment:

Adjust the PWM signal based on the control signal.

Assume a maximum error for which KP has been confirmed to work effectively.

Implement defensive coding to handle hardware noise by adding a condition:

- If the current state is within ±0.5 of the error, set the error to 0 (stop the motors).

Distance Control Strategy:

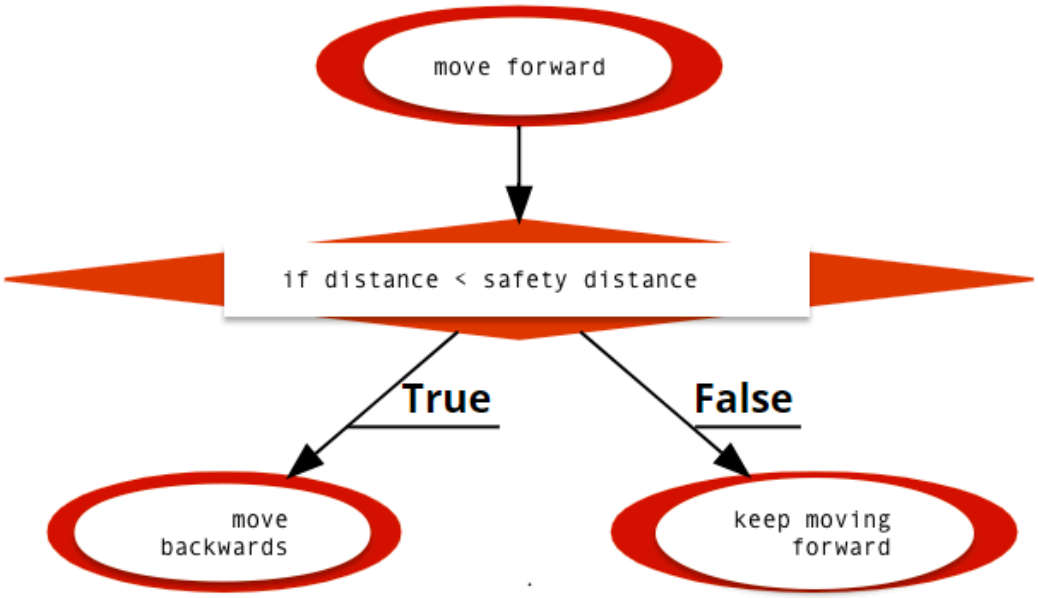
- **Move Forward**: If the distance to the car ahead is greater than the safety distance, move the car forward.
- **Move Backward**: If the distance to the car ahead is less than the safety distance, move the car backward to maintain the safety distance.

Steering Angle Control Strategy:

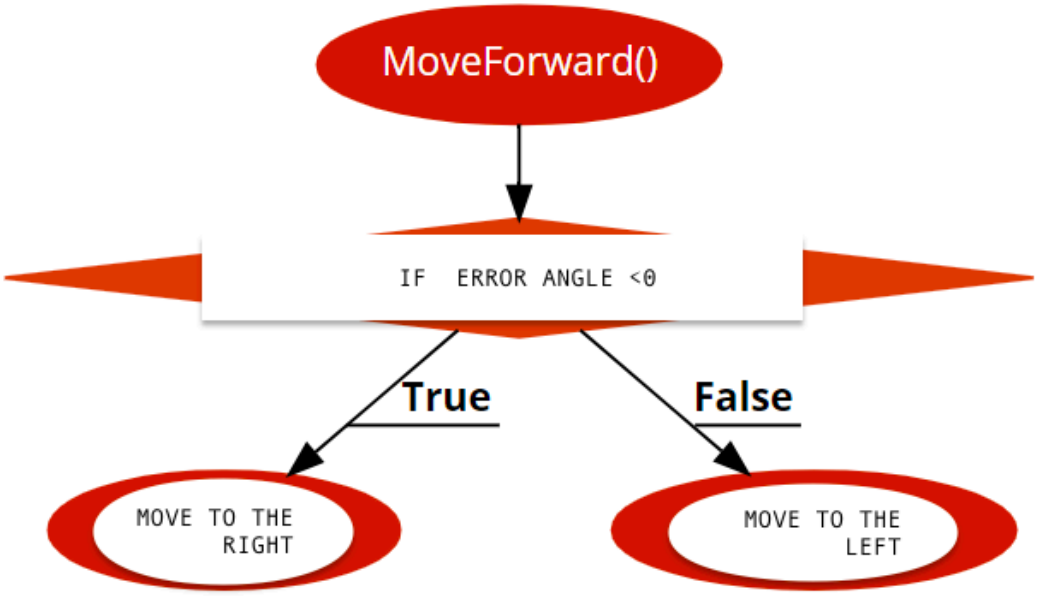
- **Move Right**: If the angle deviation is positive (greater than a specified threshold), turn the car to the right.
- **Move Left**: If the angle deviation is negative (less than a specified threshold), turn the car to the left.

2.3 Programming :

A preliminary generic flow chart: ULTRASONIC BASED PID CONTROL



IMU (GYROSCOPE) BASED PID CONTROL LAW



Code (no RTOS):

```
 #include <esp_now.h>
#include <WiFi.h>
#include <Wire.h>
#include <ArduPID.h>
#include <MPU6050_light.h>

MPU6050 mpu(Wire);


 // Motor pin definitions
const int right_motor_a_pin = 17;     // IN1
const int right_motor_b_pin = 16;     // IN2
const int right_motor_pwm_pin = 25;   // ENA. changeddddddd
```

```
const int left_motor_c_pin = 19;      // IN3
const int left_motor_d_pin = 18;      // IN4
const int left_motor_pwm_pin = 23;    // ENB

// PID controller
ArduPID myController;

// Control variables
double input = 0;
double output = 0;
double setpoint = 0; // Desired angle setpoint
double p = 15;
double i = 1;
double d = 0.5;
double minOutput = 50; // The minimum output to send (plus or minus)
unsigned long timer = 0;
unsigned long timerInterval = 50; // Control loop interval
int smallAngleThreshold=10;

// PWM configuration
const int pwm_freq = 5000;
const int pwm_resolution = 8; // 8-bit resolution: values between 0 and
255
const int right_motor_channel = 0;
const int left_motor_channel = 1;

// ESP-NOW configuration
uint8_t broadcastAddress[] = {0x08, 0xB6, 0x1F, 0xBD, 0x37, 0x18};
float incomingGyro;
typedef struct struct_message {
    float angle;
} struct_message;
struct_message incomingReadings;
esp_now_peer_info_t peerInfo;

// Function prototypes
void setup_serial();
void setup_motors();
void setup_pid();
void run_pid();
void OnDataRecv(const uint8_t *mac, const uint8_t *incomingData, int
```

```
len);
void drive_forward(int right_motor_pwm, int left_motor_pwm);
void drive_backwards(int right_motor_pwm, int left_motor_pwm);
void drive_break();
void drive_left(int left_motor_pwm);
void drive_right(int right_motor_pwm);

// ESP-NOW receive callback
void OnDataRecv(const uint8_t *mac, const uint8_t *incomingData, int len)
{
   memcpy(&incomingReadings, incomingData,
sizeof(incomingReadings));
   incomingGyro = incomingReadings.angle;  //get angle
}

// Setup functions
void setup_serial() {
   Serial.begin(115200);
}

void setup_motors() {
   pinMode(right_motor_a_pin, OUTPUT);
   digitalWrite(right_motor_a_pin, LOW);
   pinMode(right_motor_b_pin, OUTPUT);
   digitalWrite(right_motor_b_pin, LOW);
   pinMode(right_motor_pwm_pin, OUTPUT);
   ledcSetup(right_motor_channel, pwm_freq, pwm_resolution);
   ledcAttachPin(right_motor_pwm_pin, right_motor_channel);
   ledcWrite(right_motor_channel, 0);

   pinMode(left_motor_c_pin, OUTPUT);
   digitalWrite(left_motor_c_pin, LOW);
   pinMode(left_motor_d_pin, OUTPUT);
   digitalWrite(left_motor_d_pin, LOW);
   pinMode(left_motor_pwm_pin, OUTPUT);
   ledcSetup(left_motor_channel, pwm_freq, pwm_resolution);
   ledcAttachPin(left_motor_pwm_pin, left_motor_channel);
   ledcWrite(left_motor_channel, 0);
}

void setup_pid() {
```

```
    myController.begin(&input, &output, &setpoint, p, i, d);
    myController.setSampleTime(100); // Ensure at least 100ms between
successful compute() calls
    myController.setOutputLimits(-255, 255);
    myController.start();
}

void setup() {
    setup_serial();
    setup_motors();
    setup_pid();
    mpu.begin();
    // Init ESP-NOW
    WiFi.mode(WIFI_STA);
    if (esp_now_init() != ESP_OK) {
        Serial.println("Error initializing ESP-NOW");
        return;
    }
    esp_now_register_recv_cb(OnDataRecv);
}

// Motor driving functions
void drive_forward(int right_motor_pwm, int left_motor_pwm) {
    digitalWrite(right_motor_a_pin, HIGH);
    digitalWrite(left_motor_c_pin, HIGH);
    digitalWrite(right_motor_b_pin, LOW);
    digitalWrite(left_motor_d_pin, LOW);
    ledcWrite(right_motor_channel, right_motor_pwm);
    ledcWrite(left_motor_channel, left_motor_pwm);
}

void drive_backwards(int right_motor_pwm, int left_motor_pwm) {
    digitalWrite(right_motor_b_pin, HIGH);
    digitalWrite(left_motor_d_pin, HIGH);
    digitalWrite(right_motor_a_pin, LOW);
    digitalWrite(left_motor_c_pin, LOW);
    ledcWrite(right_motor_channel, right_motor_pwm);
    ledcWrite(left_motor_channel, left_motor_pwm);
}

void drive_break() {
```

```
    digitalWrite(right_motor_b_pin, HIGH);
    digitalWrite(left_motor_d_pin, HIGH);
    digitalWrite(right_motor_a_pin, HIGH);
    digitalWrite(left_motor_c_pin, HIGH);
    ledcWrite(right_motor_channel, 0);
    ledcWrite(left_motor_channel, 0);
}

void drive_left(int left_motor_pwm) {
    digitalWrite(right_motor_a_pin, LOW);
    digitalWrite(right_motor_b_pin, HIGH);
    digitalWrite(left_motor_c_pin, HIGH);
    digitalWrite(left_motor_d_pin, LOW);
    ledcWrite(right_motor_channel, 0);
    ledcWrite(left_motor_channel, left_motor_pwm);
}

void drive_right(int right_motor_pwm) {
    digitalWrite(right_motor_a_pin, HIGH);
    digitalWrite(right_motor_b_pin, LOW);
    ledcWrite(right_motor_channel, right_motor_pwm);
}

// Main loop
 void run_pid() {
  mpu.update();
    if (millis() - timer >= timerInterval) {
        input = incomingGyro+mpu.getAngleZ();

        // Check for small angles
        if (abs(input) < smallAngleThreshold) {
            drive_forward(100,100);  // Do nothing if the angle is very small
        } else {
            int mappedOutput = map(input, -90, 90, -255, 255);

            if (incomingGyro > 0) {
                drive_right(mappedOutput); //right
            } else {
                drive_left(-mappedOutput); //lefts
            }
        }
```

```
        myController.compute();
        myController.debug(&Serial, "myController", PRINT_INPUT |
PRINT_OUTPUT | PRINT_SETPOINT | PRINT_BIAS | PRINT_P | PRINT_I |
PRINT_D);
        timer = millis();
    }
}


void loop() {
    run_pid();
}
```

CODE+FREERTOS+TOGGLING LED :

```
#include <esp_now.h>
#include <WiFi.h>
#include <Wire.h>
#include <ArduPID.h>
#include <MPU6050_light.h>
#include <Arduino.h>
#include <freertos/FreeRTOS.h>
#include <freertos/task.h>

MPU6050 mpu(Wire);

// Motor pin definitions
const int right_motor_a_pin = 17;
const int right_motor_b_pin = 16;
```

```
const int right_motor_pwm_pin = 25;
const int left_motor_c_pin = 19;
const int left_motor_d_pin = 18;
const int left_motor_pwm_pin = 23;

// PID controller
ArduPID myController;

// Control variables
double input = 0;
double output = 0;
double setpoint = 0;
double p = 15;
double i = 1;
double d = 0.5;
double minOutput = 50;
unsigned long timer = 0;
unsigned long timerInterval = 50;
int smallAngleThreshold = 10;

// PWM configuration
const int pwm_freq = 5000;
const int pwm_resolution = 8;
const int right_motor_channel = 0;
const int left_motor_channel = 1;

// ESP-NOW configuration
uint8_t broadcastAddress[] = {0x08, 0xB6, 0x1F, 0xBD,
0x37, 0x18};
float incomingGyro;
typedef struct struct_message {
    float angle;
} struct_message;
struct_message incomingReadings;
```

```
esp_now_peer_info_t peerInfo;

// Task handles
TaskHandle_t pidTaskHandle = NULL;
TaskHandle_t ledTaskHandle = NULL;

// LED pin
const int ledPin = 2;

// Function prototypes
void setup_serial();
void setup_motors();
void setup_pid();
void run_pid(void *parameter);
void OnDataRecv(const uint8_t *mac, const uint8_t
*incomingData, int len);
void drive_forward(int right_motor_pwm, int
left_motor_pwm);
void drive_backwards(int right_motor_pwm, int
left_motor_pwm);
void drive_break();
void drive_left(int left_motor_pwm);
void drive_right(int right_motor_pwm);
void toggleLED(void *parameter);

// ESP-NOW receive callback
void OnDataRecv(const uint8_t *mac, const uint8_t
*incomingData, int len) {
    memcpy(&incomingReadings, incomingData,
sizeof(incomingReadings));
    incomingGyro = incomingReadings.angle;
}

// Setup functions
```

```
void setup_serial() {
    Serial.begin(115200);
}

void setup_motors() {
    pinMode(right_motor_a_pin, OUTPUT);
    digitalWrite(right_motor_a_pin, LOW);
    pinMode(right_motor_b_pin, OUTPUT);
    digitalWrite(right_motor_b_pin, LOW);
    pinMode(right_motor_pwm_pin, OUTPUT);
    ledcSetup(right_motor_channel, pwm_freq,
pwm_resolution);
    ledcAttachPin(right_motor_pwm_pin,
right_motor_channel);
    ledcWrite(right_motor_channel, 0);

    pinMode(left_motor_c_pin, OUTPUT);
    digitalWrite(left_motor_c_pin, LOW);
    pinMode(left_motor_d_pin, OUTPUT);
    digitalWrite(left_motor_d_pin, LOW);
    pinMode(left_motor_pwm_pin, OUTPUT);
    ledcSetup(left_motor_channel, pwm_freq,
pwm_resolution);
    ledcAttachPin(left_motor_pwm_pin, left_motor_channel);
    ledcWrite(left_motor_channel, 0);
}

void setup_pid() {
    myController.begin(&input, &output, &setpoint, p, i, d);
    myController.setSampleTime(100);
    myController.setOutputLimits(-255, 255);
    myController.start();
}
```

```
void setup() {
    setup_serial();
    setup_motors();
    setup_pid();
    mpu.begin();

    // Init ESP-NOW
    WiFi.mode(WIFI_STA);
    if (esp_now_init() != ESP_OK) {
        Serial.println("Error initializing ESP-NOW");
        return;
    }
    esp_now_register_recv_cb(OnDataRecv);

    // LED setup
    pinMode(ledPin, OUTPUT);

    // Create FreeRTOS tasks
    xTaskCreate(run_pid, "PID Task", 4096, NULL, 1,
&pidTaskHandle);
    xTaskCreate(toggleLED, "LED Task", 1024, NULL, 1,
&ledTaskHandle);
}

void drive_forward(int right_motor_pwm, int left_motor_pwm)
{
    digitalWrite(right_motor_a_pin, HIGH);
    digitalWrite(left_motor_c_pin, HIGH);
    digitalWrite(right_motor_b_pin, LOW);
    digitalWrite(left_motor_d_pin, LOW);
    ledcWrite(right_motor_channel, right_motor_pwm);
    ledcWrite(left_motor_channel, left_motor_pwm);
}
```

```
void drive_backwards(int right_motor_pwm, int
left_motor_pwm) {
    digitalWrite(right_motor_b_pin, HIGH);
    digitalWrite(left_motor_d_pin, HIGH);
    digitalWrite(right_motor_a_pin, LOW);
    digitalWrite(left_motor_c_pin, LOW);
    ledcWrite(right_motor_channel, right_motor_pwm);
    ledcWrite(left_motor_channel, left_motor_pwm);
}

void drive_break() {
    digitalWrite(right_motor_b_pin, HIGH);
    digitalWrite(left_motor_d_pin, HIGH);
    digitalWrite(right_motor_a_pin, HIGH);
    digitalWrite(left_motor_c_pin, HIGH);
    ledcWrite(right_motor_channel, 0);
    ledcWrite(left_motor_channel, 0);
}

void drive_left(int left_motor_pwm) {
    digitalWrite(right_motor_a_pin, LOW);
    digitalWrite(right_motor_b_pin, HIGH);
    digitalWrite(left_motor_c_pin, HIGH);
    digitalWrite(left_motor_d_pin, LOW);
    ledcWrite(right_motor_channel, 0);
    ledcWrite(left_motor_channel, left_motor_pwm);
}

void drive_right(int right_motor_pwm) {
    digitalWrite(right_motor_a_pin, HIGH);
    digitalWrite(right_motor_b_pin, LOW);
    ledcWrite(right_motor_channel, right_motor_pwm);
}
```

```
// PID task function
void run_pid(void *parameter) {
    TickType_t xLastWakeTime;
    const TickType_t xFrequency =
pdMS_TO_TICKS(timerInterval);

    xLastWakeTime = xTaskGetTickCount();

    while (true) {
        mpu.update();
        input = incomingGyro + mpu.getAngleZ();

        if (abs(input) < smallAngleThreshold) {
            drive_forward(100, 100);
        } else {
            int mappedOutput = map(input, -90, 90, -255, 255);

            if (incomingGyro > 0) {
                drive_right(mappedOutput);
            } else {
                drive_left(-mappedOutput);
            }
        }

        myController.compute();
        myController.debug(&Serial, "myController",
PRINT_INPUT | PRINT_OUTPUT | PRINT_SETPOINT |
PRINT_BIAS | PRINT_P | PRINT_I | PRINT_D);

        vTaskDelayUntil(&xLastWakeTime, xFrequency);
    }
}

// LED toggle task function
```

```
void toggleLED(void *parameter) {
    const TickType_t xDelay = pdMS_TO_TICKS(1000);

    while (true) {
        digitalWrite(ledPin, !digitalRead(ledPin));
        vTaskDelay(xDelay);
    }
}

void loop() {
    // Empty loop since FreeRTOS tasks are running
}
```