UNIVERSITY OF PADOVA

Embedded Real–Time Control

Laboratory Report

Brescia Davide (2182457)
Favaro Manuel (2199436)
Filannino Daniele (2179425)

Academic Year 2025-2026

Contents

1 Laboratory 1

The objective of this laboratory was to introduce the fundamental concepts required for embedded real-time programming on the STM32 platform. In particular, the laboratory focused on peripheral configuration using STM32CubeIDE and on communication with external devices through digital interfaces. The tasks performed during this laboratory were:

- Configure and handle external interrupts;
- Acquire data from the SX1509 keypad and the line sensor via I^{2}C;
- Implement periodic line sensor reading using both polling and timer interrupts

1.1 Relevant theoretical notions

This laboratory introduced interrupt-driven programming and real-time data acquisition on an STM32 microcontroller.
The focus was placed on understanding how asynchronous events, periodic tasks, and peripheral communications can be efficiently managed in embedded systems. The main notions covered were:

- External interrupt handling (EXTI)
- I^{2}C communication with peripheral devices
- Polling versus interrupt-based execution
- Hardware timer configuration
- Real-time debug output using SWV ITM

1.1.1 External Interrupts and EXTI

Hardware interrupts allow the microcontroller to respond immediately to external events without continuously polling inputs. The SX1509 keypad triggers an interrupt on pin `PF4` when a key is pressed. The callback `HAL_GPIO_EXTI_Callback` reads the internal registers of the SX1509 to identify the pressed key, and clears the interrupt flag, allowing the microcontroller to be ready for the next interrupts.

1.1.2 I^{2}C communication

The SX1509 I/O expander communicates with the STM32 through the I^{2}C serial bus, enabling data transfer between two devices. In this laboratory, we use I^{2}C to read data from registers like `REG_KEY_DATA_1`, `REG_KEY_DATA_2`, and `REG_DATA_B` on the SX1509. To access these registers, we use `HAL_I2C_Mem_Read` function. This function requires the following parameters:

- device I^{2}C address of the slave device (e.g., `0x3E` for the line sensor).
- register address of the master device (e.g., `REG_KEY_DATA_1` for keypad data).
- addressing mode, used for memory address (8-bit usually)
- buffer pointer, where the data read from the register will be stored
- number of bytes to read
- timeout

In this laboratory, two SX1509 devices were connected to the same I^{2}C bus. Each device has a unique address, allowing the microcontroller to differentiate between them and communicate with them separately.

1.1.3 Polling versus timer interrupts

Sensor acquisition can be performed using two different approaches: software polling and hardware timer interrupts. These two techniques differ in terms of CPU usage, timing precision, and system responsiveness.

Polling using HAL_GetTick()

This method involves periodically checking the sensor value by using time-based functions such as `HAL_GetTick()`, which track the system time in milliseconds. The sensor is checked at regular intervals by comparing the current system time with a pre-defined timeout value. This approach is simple to implement and does not require dedicated hardware, but it still requires the CPU to repeatedly check the time and sensor status, potentially reducing system efficiency if the polling period is very short.

Periodic acquisition using Timer Interrupts

This method involves configuring a hardware timer (e.g. TIM6) to generate periodic interrupts at specific intervals. When the timer reaches its set period, it triggers an interrupt, which allows the microcontroller to perform other tasks concurrently while waiting for the interrupt. This approach is non-blocking, meaning the CPU can execute other operations without being tied up in a busy-wait loop, offering greater efficiency, especially in real-time or multitasking systems.

1.1.4 Hardware Timer Configuration

Timers in STM32 can be used to generate interrupts at regular intervals. We used the TIM6 timer to trigger interrupts every 100 ms, allowing periodic readings of the line sensor without blocking the CPU and ensuring very high precision in polling. The timer configuration involves setting the prescaler (PSC) and the auto-reload register (ARR) to determine the interrupt frequency.

1.1.5 SWV ITM Debugging

The Serial Wire Viewer (SWV) allows real-time transmission of debug messages. SWV was used to redirect the output of the `printf()` function to the debugger console. This is especially useful for observing variables and events during code execution without the need for a physical display.

1.2 Line Sensor Implementation

The main purpose of this section is to implement a routine that periodically reads the data transmitted by the line sensor via I^{2}C, given a predetermined time interval. Before proceeding with the implementation, let us take a look at how the line sensor works.

1.2.1 Polulu QTR-MD-08RC

The TurtleBot is equipped with a Polulu QTR-MD-08RC[1] infrared line sensor, designed to detect the contrast between dark and light surfaces. The sensing mechanism relies on an infrared LED, which illuminates the surface below the TurtleBot, and a phototransistor responsible for measuring the intensity of the reflected radiation. This detection principle is governed by the optical properties of the surface: light-colored materials exhibit a higher reflectance, whereas dark surfaces tend to absorb most of the incident infrared light.
As illustrated in the schematic diagram (Figure 1), the sensor operates on an RC timing principle. It is important to note that this schematic depicts the circuit of a single sensing channel; the QTR-MD-08RC module integrates an array of eight identical circuits arranged linearly. Due to the interfacing with the `SX1509` I/O expander, the readout strategy is adapted to a fixed-time threshold method rather than continuous time measurement.
The measurement cycle begins by driving the sensor lines `HIGH` to charge the internal 2.2 nF capacitors. After a brief charging period, the lines are switched to input mode. The system waits for a predetermined delay (calibration time) and then performs a parallel read of the SX1509's `Register B`. Since each of the eight sensors is physically mapped to a specific bit position (0 through 7) within this register, the operation captures a simultaneous digital snapshot of the entire array.
This results in an 8-bit integer value ranging from 0 to 255, where the state of each bit is determined as follows:

- A logic `HIGH` (1) indicates that the capacitor has not yet discharged, implying a low phototransistor current due to a dark surface (low reflectance).
- A logic `LOW` (0) indicates that the capacitor has already discharged, implying a high phototransistor current due to a light surface (high reflectance).

[1]For more info visit:
`https://www.pololu.com/product/4148`

Consequently, the integer value read from `Register B` acts as a direct bitmask, encoding the presence and position of the line under the eight sensors.

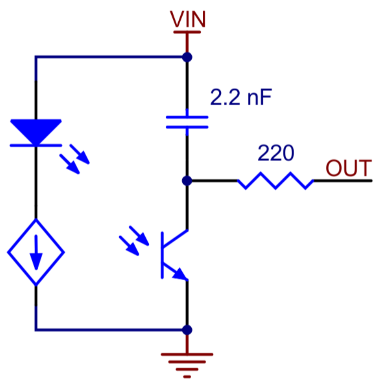


Figure 1: Line Sensor Functional Diagram

1.2.2 Busy-Wait Polling - Initial Approach

The initial approach considered for reading the line sensor was based on a busy-wait loop. In this scheme, the firmware would continuously poll the system tick counter and perform the I^2C read operation inside a tight loop until the specified timeout was reached. While conceptually simple, this approach has a significant drawback: during the execution of the busy-wait loop, the CPU remains fully occupied, continuously checking the tick counter and performing I^2C reads. In practice, this keeps the microcontroller in a quasi-halt state for the duration of the wait, preventing it from executing any other tasks or responding to interrupts efficiently. The first implementation is shown in Listing 1.

Listing 1: Busy-Wait Polling routine

```c
void PollingLineSensor(uint8_t polling_period){
    uint8_t data;
    HAL_StatusTypeDef status;
    while (HAL_getTick() < timeout){
        status = HAL_I2C_Mem_Read(&hi2c1, SX1509_I2C_ADDR1 << 1, REG_DATA_B, 1,
            &data, 1, I2C_TIMEOUT);
        if (status != HAL_OK) {
            printf("I2C communication error (%X).\n", status);}
        printf("Line sensor data: (%d).\n", data);
    }
```

1.2.3 Periodic Polling Implementation

To overcome the previous limitation, a non-blocking periodic polling mechanism using `HAL_GetTick()` was implemented instead. In this approach, the firmware calculates a timeout but allows the main loop to continue executing other code until the scheduled polling instant arrives, maintaining CPU availability and system responsiveness. To implement non-blocking periodic sampling of the line sensor, the STM32 firmware relies on the `HAL_GetTick()` function, which returns the number of milliseconds elapsed since system startup. This monotonic counter allows the creation of software timers without suspending program execution. Instead of halting the CPU for a fixed interval (as would occur with the `HAL_Delay()` function) the code continuously checks whether the current system time has exceeded a predefined timeout value. At the beginning of the polling routine, a timeout is initialized as `HAL_GetTick() + polling_period` (in milliseconds). This computation schedules the next sampling event exactly `polling_period` milliseconds after the current time. For instance, if `HAL_GetTick()` returns 1000 ms and the polling period is set to 100 ms, the next sensor acquisition is scheduled for timestamp 1100 ms. The main loop then proceeds without blocking, repeatedly evaluating whether `HAL_GetTick() > timeout`. When this condition becomes true, it indicates that the scheduled polling period has elapsed. The line sensor is then read, and a new timeout is computed using the same method.
This approach establishes a new deadline for the following cycle, ensuring that each polling event maintains a consistent temporal spacing.
Finally, the sensor is accessed through the `SX1509` I/O expander, from which the routine reads the `REG_DATA_B`

register located at I^2C address 1. This register contains the digital states corresponding to the line sensor outputs, and the acquisition data is performed using `HAL_I2C_Mem_Read(...)`. This function retrieves one byte of data from the device. In case the communication fails, the routine prints a diagnostic message indicating the HAL error code. The retrieved value is then made available and printed on the debug console[2].

Listing 2: Periodic Polling routine

```c
uint16_t polling_period = 100;
uint16_t timeout = HAL_GetTick() + polling_period;
uint8_t data;
HAL_StatusTypeDef status;

// this is the main loop of the CPU
while (1){
    if (HAL_GetTick() > timeout){
        status = HAL_I2C_Mem_Read(&hi2c1, SX1509_I2C_ADDR1 << 1, REG_DATA_B, 1,
           &data, 1, I2C_TIMEOUT);
        if (status != HAL_OK)
            printf("I2C communication error (%X).\n", status);
        printf("Line sensor data: (%d).\n", data);
        timeout = HAL_GetTick() + polling_period;
        }
}
```

1.2.4 Periodic Polling Using the TIM6 Interrupt

To perform periodic sampling of the line sensor, we use the TIM6 peripheral configured to generate an interrupt every 100 ms. This approach provides a reliable and precise sampling rate, which is particularly important in applications where consistent timing is required for accurate line detection and control.
Using a hardware timer interrupt is one of the most effective methods for periodic polling because it avoids blocking the CPU. Instead of relying on delay functions or continuous polling loops (which would waste processing time and reduce system responsiveness) the microcontroller executes the Interrupt Service Routine (ISR) only when the timer event occurs. In other words, the CPU continues executing the main program without interruption and is diverted to the ISR only upon the rising edge of the TIM6 interrupt signal.
This mechanism ensures both high timing accuracy and efficient use of computational resources, allowing the system to maintain precise sensor sampling while performing other tasks concurrently.

[2]A video of the line sensor acquisition test is available at:
`https://github.com/DanieleFila/Embedded_Lab/blob/main/Videos/LAB1_LineSensorTest.mp4`

1.3 Dynamic LED blinking via keypad

The second objective of this lab was to create a system where the user can input a desired blinking frequency for a LED by typing it on the keypad (e.g., typing *125#* to set the LED to blink at `125 Hz`). To achieve this goal, we must understand how the keypad works, detect which button is pressed, convert the input into numerical data, process it to extract the desired frequency, and finally apply this frequency to the LED control logic.

1.3.1 Decoding keypad inputs

The 4x4 matrix keypad used in this laboratory is arranged in rows and columns, with each line connected to a GPIO pin of the SX1509_2 through an internal pull-up resistor. Consequently, when no key is pressed all lines remain at logic level '1', and the registers `REG_KEY_DATA_1` and `REG_KEY_DATA_2` contain only bits set to '1'. When a key is pressed, the corresponding row and column lines are pulled to ground, forcing the associated bits in the two registers to '0'. The microcontroller, therefore, detects the pressed key by identifying which bits transition from '1' to '0', uniquely determining the involved row and column. Once these values are read, they must be decoded to obtain the exact key location within the matrix, as shown in Listing 3.

Listing 3: GetKeyPadData

```
    uint8_t column = 0;
    uint8_t row = 0;
    HAL_I2C_Mem_Read(&hi2c1, SX1509_I2C_ADDR2 << 1, REG_KEY_DATA_1, 1, &column,
        1, I2C_TIMEOUT);
    HAL_I2C_Mem_Read(&hi2c1, SX1509_I2C_ADDR2 << 1, REG_KEY_DATA_2, 1, &row, 1,
        I2C_TIMEOUT);
```

Once the bytes are obtained, they must be translated into the corresponding keypad character. The key detection logic relies on finding which bit is pulled to zero. We can make the following observations:

- The encoding is the same for both rows and columns
- Only one of the last four bits of the byte can be '0', because the first four bits are always fixed to '1'.

Therefore, we concluded that the decoding can be implemented with a simple `switch-case`, as shown in Listing 4 .

Listing 4: GetPosZero

```
int GetPosZero(uint8_t val) {
    switch (val) {
        case 0b11111110: return 0;
        case 0b11111101: return 1;
        case 0b11111011: return 2;
        case 0b11110111: return 3;
         default: return -1;
    }
}
```

1.3.2 Processing Data

After obtaining the row and column corresponding to the pressed button, the key can be directly identified since we know the keypad layout. Once the character has been determined, it is then necessary to process it in order to compute the desired LED blinking frequency. From the keypad we can obtain four possible classes of input:

- **Numeric key (`'0'–'9'`):** Each time a digit is pressed, the current frequency value must be updated by appending the new digit to the right of the existing number. Since the number is written in base 10, this operation is performed by shifting the previous value to the left (multiplying it by 10) and adding the newly entered digit. The frequency is initialized to `0` before the first insertion.
- **Clear key (`'*'`):** Pressing this key resets the frequency to zero, allowing the user to start a new input sequence.
- **Confirmation key (`'#'`):** When this key is pressed, the current frequency value is confirmed and passed to the function responsible for blinking the LED (`Blink(freq)`).

- **Other keys:** All remaining keys (e.g., `A`, `B`, `C`, `D`) have no associated functionality and are therefore ignored.

Listing 5: Keypad input processing

```
int colPos = GetPosZero(column);  // int since we need to use -1
int rowPos = GetPosZero(row);

if (colPos < 0 || rowPos < 0) {
    printf("Invalid value, unable to determine the key.\n");
} else {
    led_on = false;
    char keyPressed = keypadLayout[rowPos][colPos];
    printf("Key Pressed: %c\n", keyPressed );

    if (keyPressed >= '0' && keyPressed <= '9') { // numeric input
        freq = freq*10 + (keyPressed - '0');      // left shift + new digit
    }
    else if (keyPressed == '#') {                 // confirm and blink
        led_on = true;
        Blink(freq);
    }
    else if (keyPressed == '*') {                 // clear frequency
        printf("The frequency has been resetted.\n");
        freq = 0;
    }
    else {
        printf("The button pressed has no function.\n");
    }

    printf("Frequency: %d\n", freq);
}
```

1.3.3 Blink() function

The goal of the `Blink()` function is to toggle the LED with a period of 1/`freq` seconds, producing a blinking effect at the selected frequency. A critical case to manage is the condition in which `freq = 0`. In this situation, a division by zero would occur when computing the delay, causing a fault in the STM32. To avoid this, the function explicitly checks for this condition and bypasses the toggling logic, directly setting the LED to the ON state. This behavior is not only safe but also physically meaningful, as a zero frequency corresponds to a constant signal.
In order to generate the desired blinking frequency, a straightforward approach consists of using the `HAL_Delay()` function, which introduces a delay specified in milliseconds and takes an unsigned 32-bit integer as input. Consequently, the time interval between two LED toggles is computed as 1000/`freq` milliseconds. Since the delay value must be an integer, precision is lost at higher frequencies: for `freq > 1000`, the integer division returns zero, resulting in no effective delay. However, this limitation does not affect the application because the human eye is unable to detect LED flickering above approximately 100 Hz, as confirmed during our testing. Although in video recordings flickering may become visible due to camera frame rates, for real-time visual perception this constraint is irrelevant for our use case[3]. The implementation of this function is shown in Listing 6

[3] A video showing this phenomenon is available at:
`https://github.com/DanieleFila/Embedded_Lab/blob/main/Videos/LAB1_BlinkFunction.mp4`

Listing 6: Blink Function

```c
void Blink(uint16_t freq) {
    printf("Started Blinking!!!\n");
    if (freq == 0){
        printf("Freq 0 Hz, always on\n");
        HAL_GPIO_WritePin(GPIOE, GPIO_PIN_5, 1);
        return;
    }
   while (1){
        uint32_t delay = 1000/freq; //HAL_Delay come input uint32
        HAL_GPIO_TogglePin(GPIOE, GPIO_PIN_5);
        HAL_Delay(delay);
    }
}
```

This approach presents a critical limitation: once the program enters the loop, it remains blocked and cannot respond to interrupts, making it impossible to stop or adjust the LED blinking dynamically. This blocking behavior prevents the microcontroller from handling other tasks while the LED is blinking.
To overcome the limitation of blocking loops, a non-blocking LED blinking strategy can be implemented using a software-based counter within a timer callback function. In this approach, the function `TimerCallback()` is invoked at fixed intervals (e.g., every millisecond) by a hardware timer. A static counter is incremented on each call, and when the counter reaches the threshold determined by `1000 / freq`, the LED is toggled and the counter is reset. The LED is toggled only if the boolean variable `led_on` is set to `true`, allowing blinking to be started or stopped dynamically. This method is non-blocking, so the microcontroller can continue executing other tasks or responding to interrupts while the LED is blinking. The implementation of this function is shown in Listing 7.

Listing 7: Non-blocking blinking via timer callback

```c
void TimerCallback(void) { //this hw timer has to be set with a period of 1 ms
    static uint16_t counter = 0;

    if (led_on) {
        counter++;  // increment counter every millisecond (timer period)
        if (counter >= 1000 / freq) {  // check if threshold reached
            HAL_GPIO_TogglePin(GPIOE, GPIO_PIN_5); // toggle LED
            counter = 0;  // reset counter
        }
    }
}
```

2 Laboratory 2

The objective of this laboratory was to identify the dynamic parameters of the DC motors mounted on the TurtleBot and to prepare the software infrastructure required for closed-loop speed control. In particular, the laboratory focused on open-loop motor actuation, angular speed measurement through incremental encoders, and data acquisition for model identification.
The experimental activity included the analysis of the motor actuation hardware, the implementation of voltage-based motor control using PWM signals, the measurement of wheel angular speed and the acquisition of open-loop step responses used to estimate the motor dynamics.

2.1 Hardware

2.1.1 Motors

The TurtleBot is equipped with two brushed DC motors coupled with a gearbox and incremental encoders. A brushed DC motor converts electrical energy into mechanical rotation through the interaction between the magnetic field generated by the stator and the current flowing in the rotor windings. The generated torque is proportional to the armature current, while the angular speed is influenced by the applied voltage and by the back electromotive force. From a control perspective, the motor can be modeled as a first-order dynamical system that relates the applied voltage to the resulting angular speed. This simplified representation is sufficiently accurate for speed regulation tasks and is widely adopted in embedded motor control applications. The motors are driven through a DRV8871 motor driver which integrates an H-bridge power stage. The H-bridge allows bidirectional control of the motor by appropriately switching the polarity of the voltage applied to the motor terminals. Depending on the logic levels applied to the driver inputs, the motor can operate in different modes:

- Forward: the motor rotates in the forward direction;
- Reverse: the motor rotates in the opposite direction;
- Brake: the motor terminals are shorted, providing active braking;
- Coast: the motor terminals are left floating, allowing free deceleration.

By modulating these modes using PWM signals, it is possible to regulate the average voltage applied to the motor and therefore its angular speed. Motor speed control is achieved by applying a Pulse Width Modulated (PWM) signal to the driver inputs. The duty cycle of the PWM signal determines the average voltage applied to the motor according to

$$V_{\text{avg}} = D \cdot V_{\text{supply}},$$

where D is the duty cycle. Two PWM-based actuation strategies are supported:

- Forward and Coast: the PWM alternates between forward drive and coast mode;
- Forward and Brake: the PWM alternates between forward drive and active braking.

In this laboratory, the motors are driven using the *Forward and Coast* strategy, which provides smoother behavior and reduced current stress. The *Forward and Brake* mode can be enabled by modifying the corresponding section of the control code. The maximum rated voltage of the motors is 6 V. Since the TurtleBot is powered by a 12 V supply, the maximum allowable duty cycle must be limited to approximately 50% to avoid over-voltage conditions; for this reason the motor control function explicitly saturates the commanded voltage within the interval $[-6 \text{ V}, +6 \text{ V}]$ before converting it into a PWM duty cycle. This constraint prevents damage to the motors. An additional aspect to consider is the voltage-to-duty-cycle conversion factor, denoted as `V2DUTY`. This parameter depends directly on the motor supply voltage. Consequently, when switching from a bench power supply to battery operation, the value of `V2DUTY` must be updated accordingly. Failing to do so would result in an incorrect mapping between the commanded voltage and the effective motor input, leading to inaccurate speed control. Initially, two separate functions were implemented to control the left and right motors independently. These routines were later unified into a single global function to improve code modularity and maintainability. The function verifies that the commanded voltage respects the motor limitations, converts the voltage into a corresponding PWM duty cycle, and applies the signal to the appropriate timer channels. The motors are driven in *Forward and Coast* mode by default, while switching to *Forward and Brake* is possible by uncommenting the corresponding code section. Listing 8 reports the final implementation of the motor control routine.

Listing 8: Motor Function

```c
void Motor_SetVoltage(uint8_t motor, float voltage) {
    static const uint32_t first_channel[2]  = {TIM_CHANNEL_1, TIM_CHANNEL_3};
    static const uint32_t second_channel[2] = {TIM_CHANNEL_2, TIM_CHANNEL_4};
    int32_t duty;
    // Max Volt of motor is 6, dont go above!
    if (voltage > MAX_MOTOR_VOLTAGE)
        voltage = MAX_MOTOR_VOLTAGE;
    if (voltage < -MAX_MOTOR_VOLTAGE)
        voltage = -MAX_MOTOR_VOLTAGE;
    // Convert voltage to duty cycle
    duty = (int32_t)(voltage * V2DUTY);
    // Set PWM
    if(duty >= 0) {
        /* alternate between forward and coast */
        __HAL_TIM_SET_COMPARE(&htim8, first_channel[motor], (uint32_t)duty);
        __HAL_TIM_SET_COMPARE(&htim8, second_channel[motor], 0);
        /* alternate between forward and brake , TIM8_ARR_VALUE is a define
        __HAL_TIM_SET_COMPARE (&htim8, first_channel[motor] , ( uint32_t )
            TIM8_ARR_VALUE );
        __HAL_TIM_SET_COMPARE (&htim8, second_channel[motor] , ( uint32_t )(
            TIM8_ARR_VALUE - duty ));*/
    } else {
        __HAL_TIM_SET_COMPARE(&htim8, first_channel[motor], 0);
        __HAL_TIM_SET_COMPARE(&htim8, second_channel[motor], (uint32_t)(-duty));
    }
}
```

2.1.2 Encoders

The angular velocity of the motors is measured using incremental quadrature encoders mounted on the motor shafts. An incremental encoder generates a sequence of digital pulses as the shaft rotates, allowing the angular position and speed to be estimated by counting these pulses over time. Each encoder provides two square-wave signals, referred to as channels A and B, which are phase-shifted by 90 degrees. This quadrature configuration allows not only the measurement of rotational speed but also the determination of the rotation direction by analyzing the phase relationship between the two signals. In this system the encoder signals are directly connected to hardware timers configured in encoder mode. Specifically, timers TIM3 and TIM4 are used to read the left and right motor encoders. When operating in encoder mode the timer automatically increments or decrements its internal counter based on the detected encoder pulses and rotation direction. The number of pulses counted over a fixed sampling interval is proportional to the angular displacement of the motor shaft. By measuring the difference between two consecutive counter values, it is therefore possible to estimate the angular velocity of the motor without additional software-level pulse decoding.
The implemented encoder-reading routine computes the angular velocity by evaluating the increment of the timer counter between two consecutive sampling instants. Initially, separate functions were developed for each encoder; these were later merged into a single global function that selects the appropriate timer based on the requested encoder index. The code performs the following steps:

- Select the timer associated with the requested encoder;
- Read the current counter value;
- Compute the difference with respect to the previous counter value accounting for counter overflow and underflow;
- Determine the rotation direction using the timer counting mode;
- Convert the pulse increment into angular velocity expressed in rad/s.

The conversion from pulse count to angular velocity is performed according to:

$$\omega = \frac{\Delta N \cdot 2\pi}{N_{\text{ppr}} \cdot T_s},$$

where ΔN is the number of pulses counted during the sampling interval T_s, and N_{ppr} is the number of pulses per revolution of the encoder, including the effects of the gearbox and the selected counting mode. Listing 9 reports the final implementation of the encoder reading routine. The function maintains a static memory of the previous counter values, correctly handles counter wrap-around, and returns the estimated angular velocity in rad/s.

Listing 9: encoder Function

```c
float ReadEncoder(uint8_t enc) {
    static uint32_t prevCount[2] = {0, 0};  // 0 -> TIM3, 1 -> TIM4
    uint32_t ARR_Value;
    uint32_t currCount;
    int32_t diffCount;
    TIM_HandleTypeDef *htim;
    // Timer Selection
    if(enc == 0){
        htim = &htim3;
        ARR_Value = TIM3_ARR_VALUE;
    }else if(enc == 1) {
        htim = &htim4;
        ARR_Value = TIM4_ARR_VALUE;
    }else {return 0.0f; } //error
    currCount = __HAL_TIM_GET_COUNTER(htim);
    /* evaluate increment of Timer counter from previous count */
    if(__HAL_TIM_IS_TIM_COUNTING_DOWN(htim)) {
        if(currCount <= prevCount[enc])
            diffCount = currCount - prevCount[enc];
        else
            diffCount = -((ARR_Value + 1) - currCount) - prevCount[enc];
    }else{
        if(currCount >= prevCount[enc])
            diffCount = currCount - prevCount[enc];
        else
            diffCount = ((ARR_Value + 1) - prevCount[enc]) + currCount;
    }
    prevCount[enc] = currCount;
    // Convertion from pulses in (rad/s)
    float w = (diffCount * 2.0f * 3.1416f) / (ARR_Value * TS);
    return w;
}
```

2.2 Exporting data to Matlab

Having established control over the motors and enabled data acquisition from the encoders, the focus shifts to transmitting this telemetry to Matlab, which is essential for data logging and subsequent signal processing.

2.2.1 Organizing the signals

Before sending the signals to Matlab, they need to be organized into a structured data type. In this laboratory, four signals are sent to Matlab: the two motor angular velocities (in rad/s) and the respective step input (the reference voltage), which are activated only when the variable `startedTransmission` is set. This occurs only when the voltage is applied to the motors, approximately 5 seconds after startup, in order to ensure that the step input is synchronized with the motor response.

Listing 10: Telemetry Transmission to Matlab Structure

```c
data.w1 =  ReadEncoder(MOTOR1);
data.w2 =  ReadEncoder(MOTOR2);
if (startedTrasmission==0){
    data.u1 = 0.0f;
    data.u2 = 0.0f;
}else{
    data.u1 = reference[MOTOR1];
    data.u2 = reference[MOTOR2];}
ertc_dlog_send(&logger, &data, sizeof(data));
```

Once this is done, from Matlab it is possible to access this struct and plot the signals in real-time using the provided `DataLogTest` file. This file stores the four signals into a `data` struct, which can then be saved for later use in a Matlab script for our further analysis of the step response.

2.3 Motor Parameter Identification

To verify that everything was functioning correctly (encoders, counters, etc.), the first test we performed consisted of manually moving both motors together. We observed that the measured angular velocity curves roughly overlapped, confirming that the setup was working properly. We adopted the $Forward\&Coast$ mode to conduct the following tests.
Subsequent experimental testing revealed a significant asymmetry in performance between the two actuators. The left-side motor (*Motor 1*) exhibited higher internal resistance compared to the right-side motor (*Motor 2*). Consequently, *Motor 1* achieved a lower angular velocity under the same applied voltage. Furthermore, the steady-state response displayed a distinct sinusoidal fluctuation. This oscillatory behavior is likely attributable to mechanical misalignment; the wheel does not appear to be perfectly coaxial with the motor shaft. This eccentricity introduces a rotating unbalance, resulting in a periodic load variation that translates into the observed velocity oscillations.
Following these observations, we started the system's parameter identification process by defining the valid operating regions for each motor to extract their respective transfer functions. For the slower actuator (*Motor 1*), the operational range was defined between the minimum start-up voltage and a conservative upper limit of 5.5 V. Beyond this threshold, the motor exhibited severe sinusoidal oscillations and saturation, with no significant increase in average speed. For the more efficient actuator (*Motor 2*), the lower limit was set to its minimum start-up voltage. The upper limit, however, was constrained to match the maximum velocity achievable by *Motor 1* ($\approx$ 12 rad/s). Operating *Motor 2* beyond this velocity would be unnecessary, since the robot's straight-line motion is constrained by the maximum speed of the slower motor; exceeding this limit would result in trajectory drift rather than increased forward speed.
With the valid operating regions established, the focus shifted to system identification.
The objective was to characterize the dynamic behavior of the motors by approximating their response with a First Order Plus Time Delay (FOPTD) model, with reference to the transfer function in (1), where μ represents the steady-state gain of the system, Ts refers to the time constant, and τ is the pure time delay.
A comparison of the frequency responses, that we have seen in class, revealed that the second-order model provides negligible improvements over the first-order approximation within our bandwidth of interest. Specifically, the Bode plots show that the slower pole has a marginal impact on the system dynamics in this frequency range. Consequently, the FOPTD model was derived by retaining the fast pole (associated with the motor's inductance) and disregarding the negligible contribution of the slower dynamics.

$$G(s) = \frac{\mu}{1+T}e^{-\tau s}\ [\frac{rad/s}{V}] \tag{1}$$

2.3.1 First approach - Graphical analysis

To have a first estimate of the parameters μ, T, and τ, we used the graphical method on the system's open-loop response. Experiments were performed using `5 V` (for *Motor 1*) and `3 V` (for *Motor 2*). This asymmetry compensates for the higher mechanical resistance of *Motor 1* (discussed in the previous section), allowing both motors to reach comparable steady-state angular speeds.
The numerical values for the model parameters were derived from the experimental data using the following criteria:

- **Time Delay (τ):** The dead time was approximated as 2% of the rise time calculated by the Matlab `step` function. This heuristic approach was adopted to account for the minor initial lag observed in the physical system's response.

- **Time Constant (T):** Consistent with first-order system theory, the time constant was determined by identifying the time instant where the response reaches 63.2% of its steady-state angular velocity ($t_{63.2\%}$). The value was then calculated by subtracting the delay found in the previous step:

$$T = t_{63.2\%} - \tau$$

- **Static Gain (μ):** This parameter was computed as the ratio between the measured steady-state angular velocity and the amplitude of the step input voltage.

The resulting values are reported in Table 1.

Table 1: Estimated FOPTD model parameters for both motors.

Actuator	**Input Step** [V]	$\boldsymbol{\mu}$ $[\frac{rad/s}{V}]$	**T** $[s]$	$\boldsymbol{\tau}$ $[s]$
Motor 1	5.0000	2.6680	0.4700	0.0500
Motor 2	3.0000	3.9845	0.5800	0.0600

As shown in the step responses in Figure 2 and Figure 3, *Motor 1* clearly exhibits the oscillatory behavior discussed previously. Each plot includes both the experimental response and the simulated response obtained in Matlab using the parameters reported in Table 1, and the two curves are nearly superimposed, confirming the accuracy of the identified model.

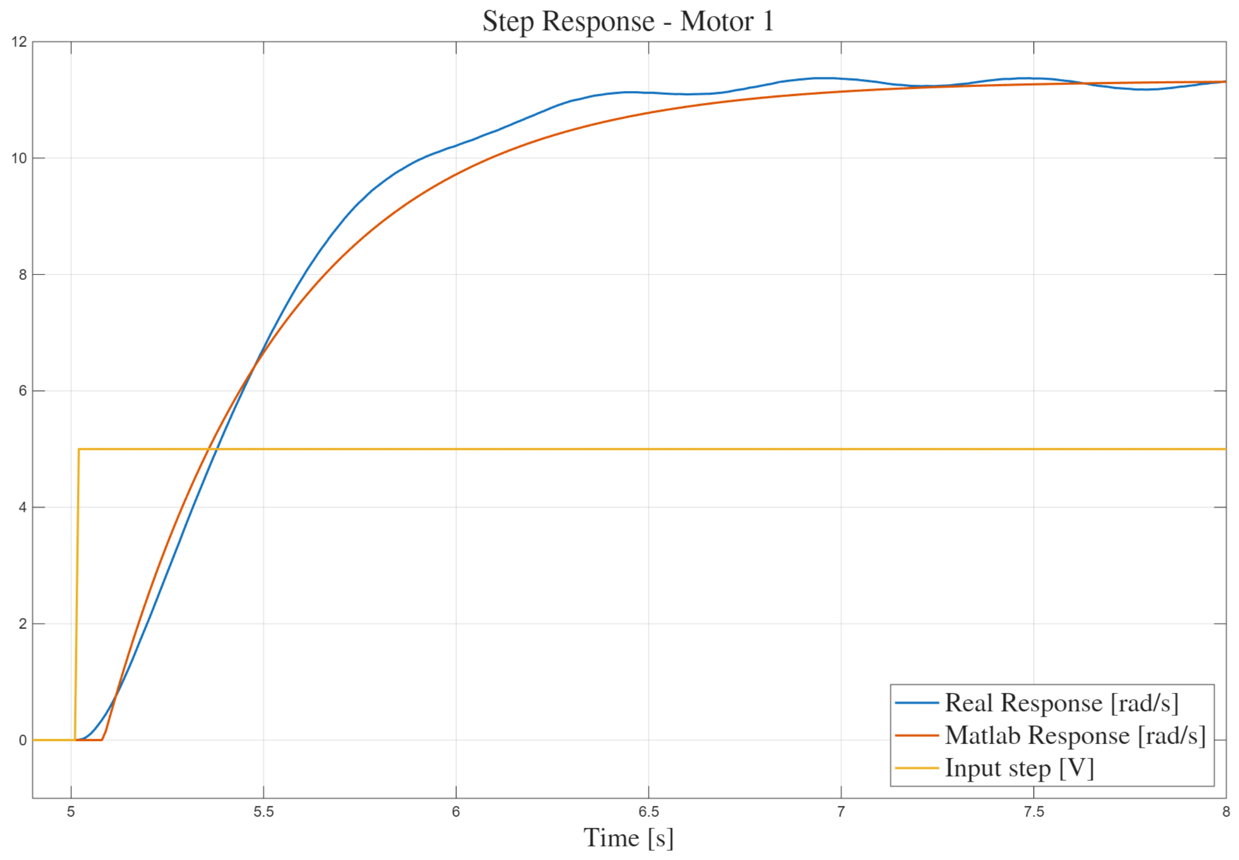


Figure 2: Open-Loop Response of Motor 1

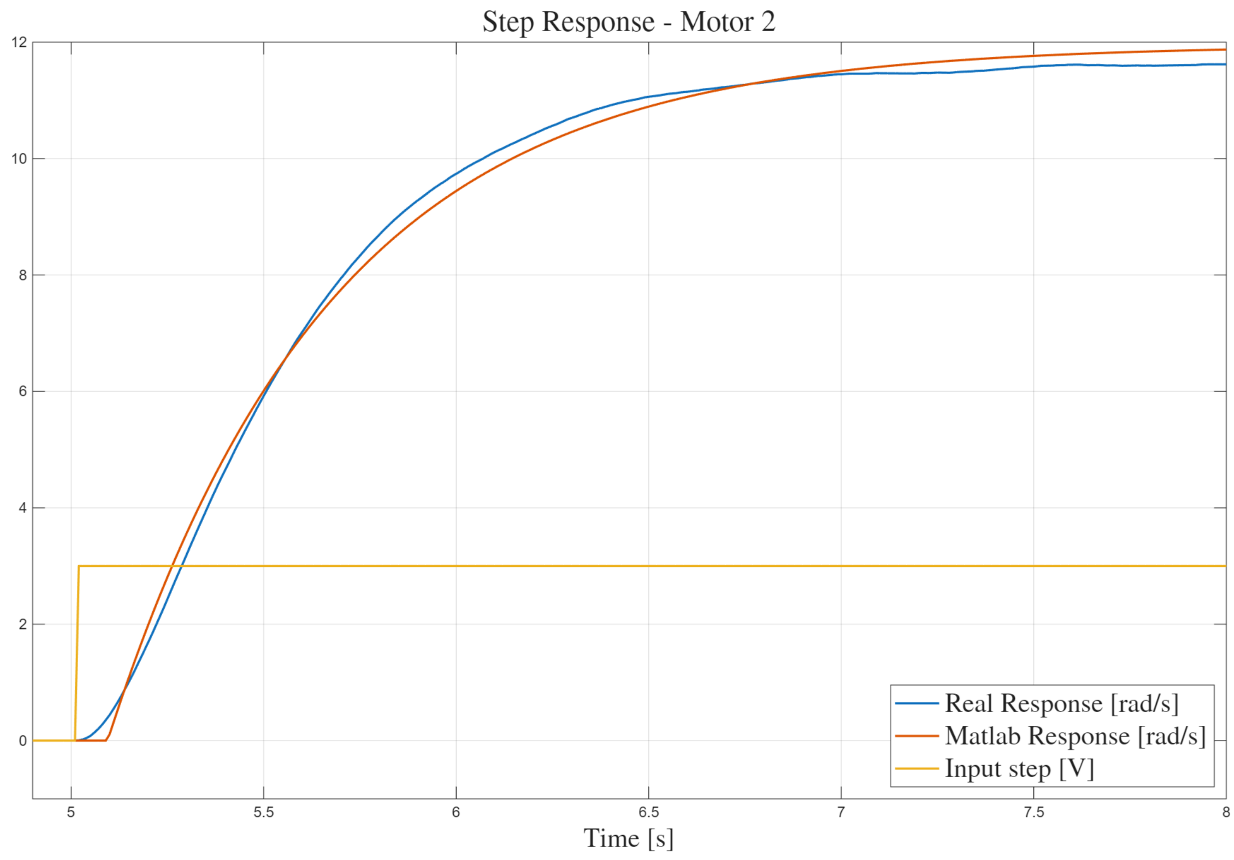


Figure 3: Open-Loop Response of Motor 2

2.3.2 Second approach - Tangent Method

Although the initial graphical inspection provided a reasonable first approximation, it suffers from subjectivity, particularly in estimating the time delay. To improve the accuracy and robustness of the identification, we adopted the Tangent Method.
This technique is based on analyzing the point of maximum slope (inflection point) of the step response curve. A tangent line is drawn at this inflection point, allowing for a geometric determination of the parameters:

- **Time Delay (τ):** identified as the time interval between the step application and the intersection of the tangent line with the time axis ($t_0 - t_{step}$).
- **Time Constant (T):** determined by the time interval between the intersection with the time axis and the intersection with the steady-state asymptote ($t_1 - t_0$).
- **Static Gain (μ):** This parameter was computed as the ratio between the measured steady-state angular velocity and the amplitude of the step input voltage.

This geometric approach minimizes the estimation error inherent in the visual approximation method used previously. We conducted a series of tests using different input voltages; the resulting parameters are collected in Table 2 and Table 3.
Combined tests were also conducted to investigate potential cross-coupling effects or limitations due to power supply loading. As shown in Table 4, the estimated parameters do not exhibit significant deviations compared to the single-motor configuration. Additionally, a representative graphical example of the Tangent Method applied to *Motor 2* (driven by a 3V input) is depicted in Figure 4.

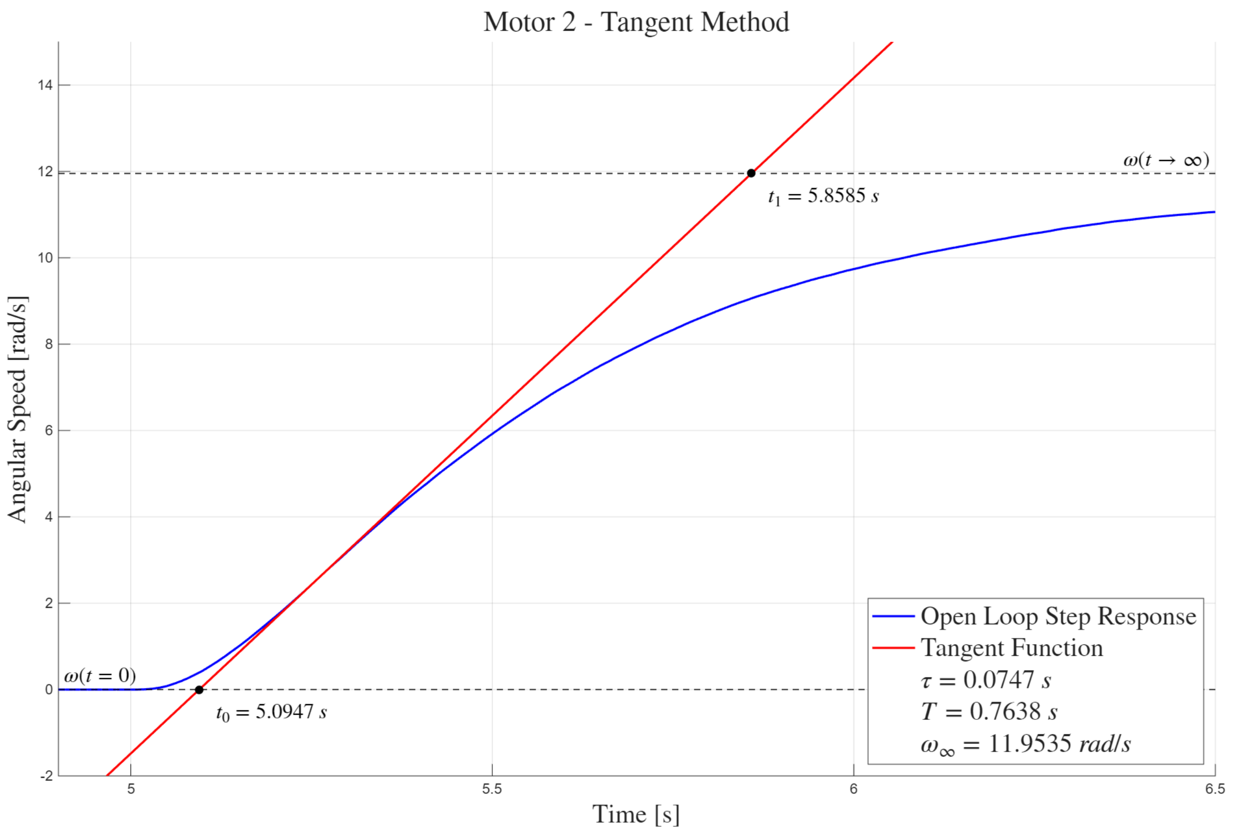


Figure 4: Example of Tangent Method applied to Motor 2

Table 2: Parameters found for Motor 1 using the Tangent Method.

Input Step [V]	$\boldsymbol{\omega}$ $[rad/s]$	$\boldsymbol{\mu}$ $[\frac{rad/s}{V}]$	**T** $[s]$	$\boldsymbol{\tau}$ $[s]$
2.5	2.1830	0.8732	0.3977	0.1219
3.0	4.6496	1.5499	0.6420	0.0765
3.5	6.9251	1.9786	0.6206	0.0628
4.0	8.9717	2.2429	0.6452	0.0640
4.5	10.3051	2.2900	0.5898	0.0625
5.0	11.9553	2.3911	0.5842	0.0562
5.5	12.3170	2.2395	0.6206	0.0479

Table 3: Parameters found for Motor 2 using the Tangent Method.

Input Step [V]	$\boldsymbol{\omega}$ $[rad/s]$	$\boldsymbol{\mu}$ $[\frac{rad/s}{V}]$	**T** $[s]$	$\boldsymbol{\tau}$ $[s]$
1.5	3.0657	2.0438	0.6905	0.1124
2.0	6.0482	3.0241	0.7103	0.0655
2.5	9.2360	3.6944	0.7979	0.0736
3.0	11.8092	3.9364	0.7382	0.0772
3.5	11.9369	3.4105	0.6918	0.0606

The lower bound of the voltage range was determined by the system's dead zone. Voltages below the reported values were insufficient to overcome static friction, even without an external load; therefore, ensuring a reliable start-up under load conditions would be impossible.
Finally, the definitive model parameters for μ, T, and τ were computed as the arithmetic mean of the values identified within the target operating range, referring to the Tangent Method as more reliable. Specifically, we

Table 4: Experimental parameters obtained from Combined Tests (both motors active).

Inputs [V]		**Motor 1 Parameters**			**Motor 2 Parameters**		
M1	**M2**	$\boldsymbol{\mu}$ $[\frac{rad/s}{V}]$	**T** $[s]$	$\boldsymbol{\tau}$ $[s]$	$\boldsymbol{\mu}$ $[\frac{rad/s}{V}]$	**T** $[s]$	$\boldsymbol{\tau}$ $[s]$
5.0	3.0	2.2668	0.6570	0.0636	3.9845	0.7638	0.0747
5.5	3.5	2.1534	0.5531	0.0599	3.3103	0.7053	0.0655

selected the voltage intervals yielding steady-state velocities between 9 rad/s and 12 rad/s. This range was chosen as it represents the optimal trade-off between high speed and system stability for our control objectives. Final averaged parameters adopted for the control design are grouped in Table 5.

Table 5: Final FOPTD model parameters

Actuator	$\bar{\boldsymbol{\mu}}$ $[\frac{rad/s}{V}]$	$\bar{\mathbf{T}}$ $[s]$	$\bar{\boldsymbol{\tau}}$ $[s]$
Motor 1	2.6680	0.4700	0.0636
Motor 2	3.9845	0.5800	0.0747

2.4 Controller Design and Tuning

With the FOPTD model parameters identified, the focus shifts to the design of the control strategy. We selected a control architecture employing two independent Proportional-Integral (PI) controllers, one for each motor.
The choice of a PI regulator is motivated by its ability to eliminate steady-state error through the integral action, while avoiding the amplification of high-frequency measurement noise.
The tuning process was conducted in two stages.

2.4.1 First Approach - Ziegler-Nichols Open-Loop Tuning

The Ziegler-Nichols Open-Loop method is an empirical tuning technique that relies on the system's open-loop step response. It makes use of the FOPTD model parameters (μ, T, τ) identified in the previous section to calculate the controller gains.
For a PI controller, the method uses the following formulas to determine the Proportional Gain (K_p) and the Integral Time (T_i):

$$K_p = \frac{0.9}{\mu} \cdot \frac{T}{\tau} \tag{2}$$

$$T_i = 3\,\tau \tag{3}$$

These formulas are designed to achieve a decay ratio of roughly one-quarter, resulting in a fast response that typically exhibits significant overshoot.
This behavior was confirmed by our tests: when applying the ZN-derived gains in both Matlab simulations and direct experiments on the TurtleBot, the system exhibited excessive overshoot. Nevertheless, these values provided a valuable starting point for the subsequent fine-tuning process.
The control law is implemented according to the canonical PI structure defined by the transfer function:

$$C(s) = K_p + \frac{1}{s}\,\frac{K_p}{T_i} \tag{4}$$

2.4.2 Second Approach - Matlab Simulations and Experimental Refinement

Given the aggressive nature of the Ziegler-Nichols method, a refinement phase was necessary. We adopted a model-based approach, using a Matlab script to simulate closed-loop responses in order to define an ideal reference dynamic, making use of the 'pidtune' built-in function. The real controller gains were then manually adjusted to match this simulated behavior, effectively compensating for physical non-linearities. This process is detailed in the *Laboratory 3* section.

3 Laboratory 3

The objective of this laboratory was to implement and validate a closed-loop speed control system for the TurtleBot motors using a Proportional-Integral (PI) controller. Built upon the motor model identified in *Laboratory 2*, the controller was designed to regulate the angular velocity of each motor and ensure accurate reference tracking under real operating conditions. Remember that even for tuning we adopted the $Forward\&Coast$ mode.

3.1 Tuning the PI Controller

After establishing an initial baseline using the Ziegler-Nichols open-loop tuning method, a fine-tuning phase was carried out to optimize the controller performance under real operating conditions. The proportional and integral gains (K_p and K_i) were manually adjusted directly on the TurtleBot in order to achieve a stable and responsive behavior, while reducing the excessive overshoot observed with the initial parameters.
The resulting experimental closed-loop responses were then compared with theoretical simulations performed in Matlab. The following analysis highlights the discrepancies between simulated and real behaviors, reports the final controller gains in Table 6, and presents a comparison of the corresponding step responses.

3.1.1 Controller Gain Comparison

The PI controller parameters obtained from the three different tuning approaches are summarized in Table 6, where $K_i = \frac{K_p}{T_i}$.

Table 6: Comparison of PI Gains (K_p, K_i)

	Motor 1		**Motor 2**	
Tuning Method	K_p	K_i	K_p	K_i
Ziegler-Nichols (Open Loop)	2.49	13.07	1.75	7.83
Matlab Simulation (pidtune)	0.54	2.07	0.35	1.14
Manual Fine-Tuning (Real System)	4.00	10.00	2.60	5.90

The data highlight a substantial discrepancy between the simulation-based optimal values and the gains required by the physical system.
This difference confirms the limitations of the FOPTD model. The Matlab script computes the gains assuming an ideal linear system, optimizing for stability margins. However, real TurtleBot actuators are subject to significant static friction and mechanical resistance introduced by the gearbox. A low proportional gain (like 0.54) fails to generate enough voltage to overcome this initial mechanical resistance, resulting in a slow start. Conversely, the higher gains identified manually (closer to or even exceeding the ZN baseline) are necessary to provide the required "breakaway torque" and ensure responsive tracking of the velocity setpoint.
Regarding the integral gain, the tuning focus shifted towards achieving a smooth response profile and minimizing the overshoot.
Furthermore, a "joint" optimization strategy was adopted. The parameters were not tuned solely for individual motor performance but were iteratively adjusted to align the settling times of both actuators. Ensuring that Motor 1 and Motor 2 reach the steady-state simultaneously is crucial for differential drive kinematics, as it prevents unwanted curvature in the robot's trajectory during the acceleration transient.

3.1.2 Response Comparison: Simulation vs. Real Hardware

To visually assess the performance, we compared the simulated closed-loop response (using the conservative pidtune parameters) against the real experimental response obtained with the manually optimized gains.
As illustrated in Figure 5 and Figure 6, the simulated response represents the ideal behavior of the linear model. In this specific test case, the plots depict the closed-loop response to a step reference velocity of 10 rad/s. The real response demonstrates that, thanks to the increased gains, the physical system is able to track this reference effectively, despite the mechanical non-linearities that the simulation does not account for.
Indeed, as detailed in the legends within the plots, it is worth noting that the dynamic behavior of the manually tuned real system closely mirrors the ideal simulation. The key performance metrics, specifically Rise Time (RT), Settling Time (ST), and Overshoot (OS) are remarkably similar in both scenarios.

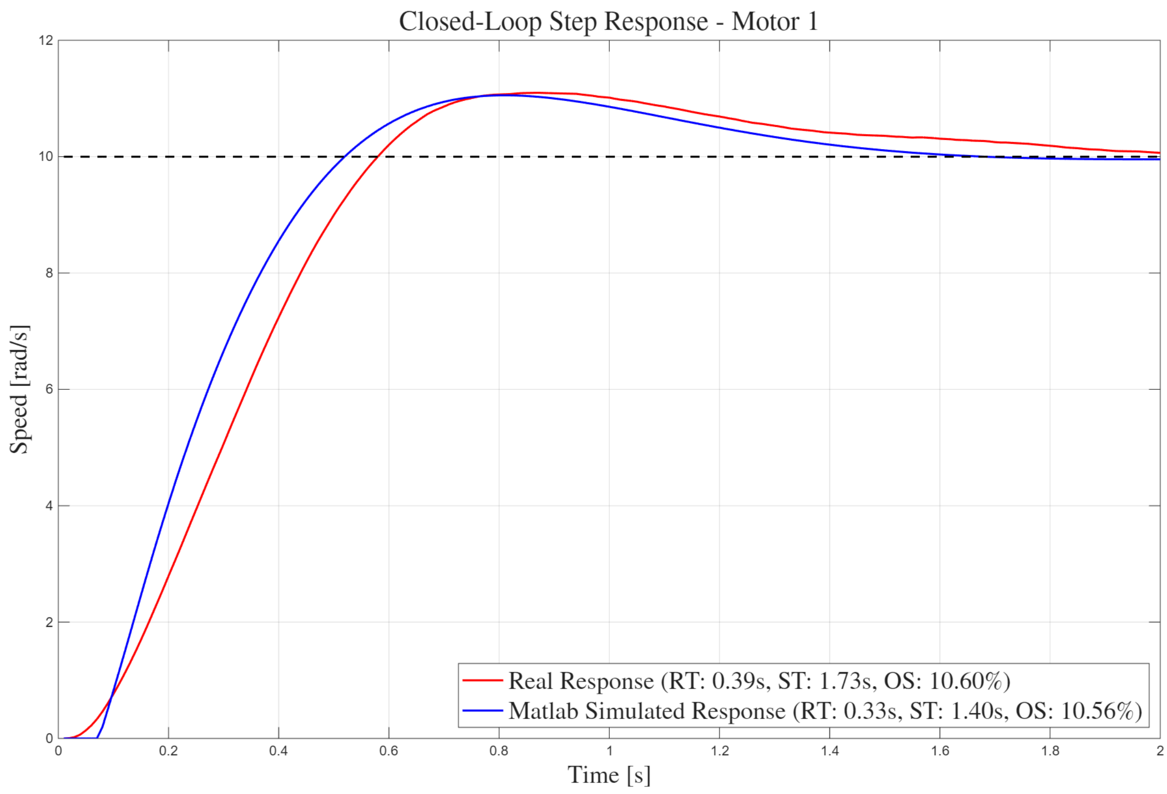


Figure 5: Closed-loop Step Response for Motor 1 - Manually Tune Vs Matlab Pidtune

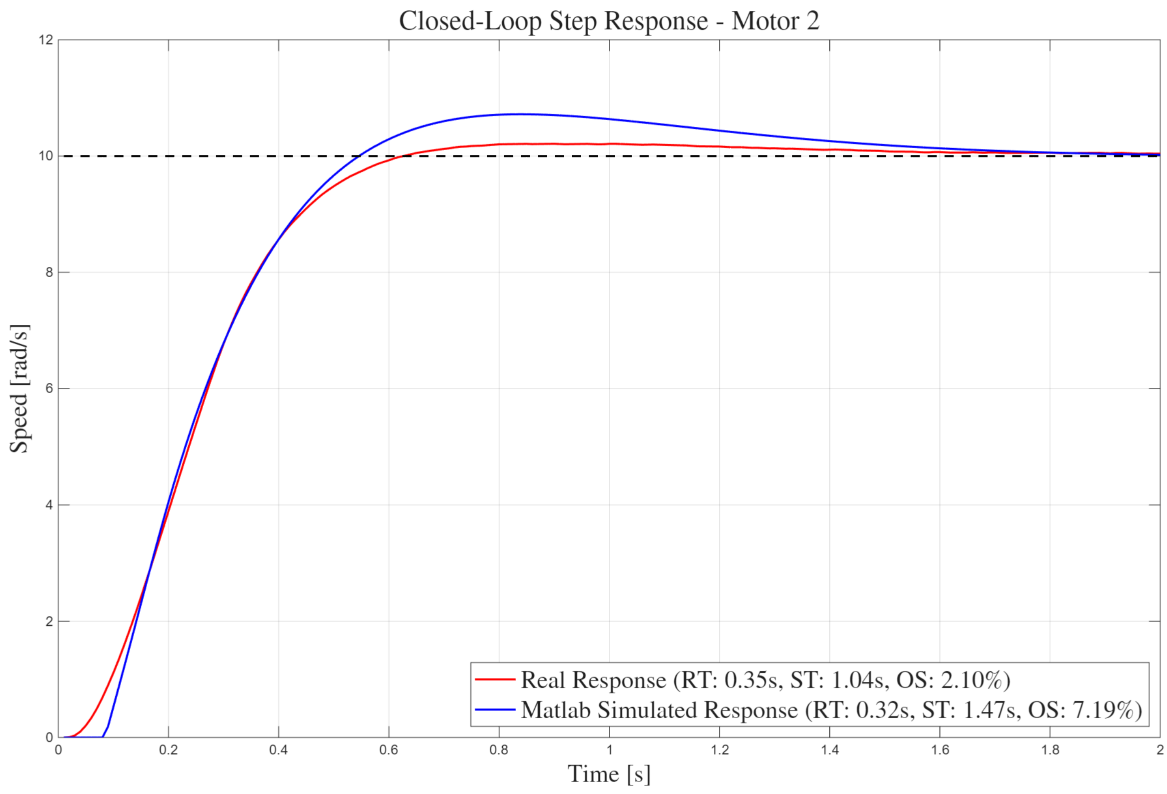


Figure 6: Closed-loop Step Response for Motor 2 - Manually Tune Vs Matlab Pidtune

3.1.3 Implementation of the PI Controller

The code implementation of the control loop is presented in Listing 11. The function `HAL_TIM_PeriodElapsedCallback` is triggered periodically by the hardware timer. Inside this routine, the angular velocities of both motors (w_1, w_2) are measured via the encoders and compared with the reference values to compute the tracking errors. These errors are then fed into the PI function, which computes the required control effort (voltage) to be applied to the motors.

Listing 11: Routine that reads encoders and applies the control law

```c
void HAL_TIM_PeriodElapsedCallback(TIM_HandleTypeDef *htim){
    if(htim->Instance == TIM6 ) {
        float w1 = ReadEncoder(MOTOR1);
        float w2 = ReadEncoder(MOTOR2);

        float error1 = reference[MOTOR1] - w1;
        float error2 = reference[MOTOR2] - w2;

        float control1 = PI(error1, MOTOR1);
        float control2 = PI(error2, MOTOR2);

        Motor_SetVoltage(MOTOR1, control1);
        Motor_SetVoltage(MOTOR2, control2);
        /* sending data to matlab functions....*/
        .
        .
    }
}
```

The control law is implemented in the `PI` function shown in Listing 12. To transition from the continuous time domain to the discrete domain of the microcontroller, the integral action is approximated using the Forward Euler integration.
The integral term is computed by accumulating the error at each time step, scaled by the integral gain (K_i) and the sampling time (TS). Crucially, the accumulator variable `I` is declared as `static`. This ensures that the integral value is preserved in memory between function calls, allowing the controller to maintain its state independently for both Motor 1 and Motor 2.

Listing 12: Implementation of PI Controller

```c
float PI(float error, int motor){
    float P = kp[motor] * error;
    static float I[2] = {0.0f,0.0f};
    I[motor] = I[motor] + ki[motor] * error * TS;
    return P + I[motor];}
```

3.1.4 AntiWindup Back Calculation

Although the tuned PI controller performs well under normal conditions, a critical issue emerges during actuator saturation or power interruptions. A specific scenario was observed: if the motor power supply is disconnected, the microcontroller will continue to run the control loop with a non-zero reference, since the error remains constant. Consequently, the integral term accumulates indefinitely, reaching a huge value. When power is restored, this accumulated “charge” causes the motors to accelerate uncontrollably (maximum voltage) for a prolonged period before the integrator can “unwind”, leading to dangerous behavior.

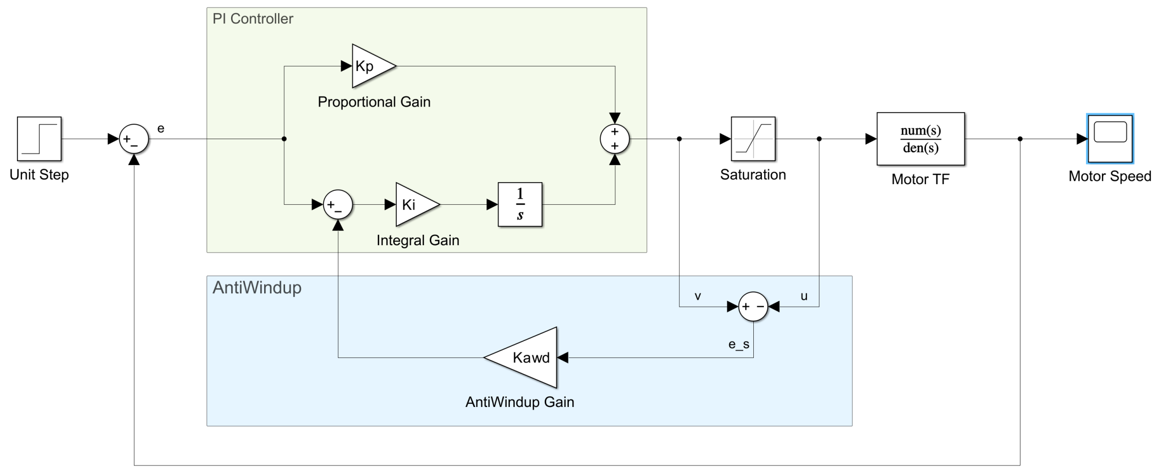


Figure 7: Block Diagram of Anti-Windup Mechanism

To address this, we implemented an Anti-Windup mechanism based on the back-calculation technique, whose block diagram is reported in Figure 7.
The algorithm calculates the theoretical control output (v). If this value exceeds the maximum available voltage (`MAX_MOTOR_VOLTAGE`), the output is clamped (u), and the difference between the theoretical and clamped value ($e_s = v - u$) is computed. This difference is then multiplied by an anti-windup gain (K_{awd}) and subtracted from the integral update.
Regarding the tuning of the anti-windup loop, the gain was selected according to the following relationship:

$$K_{awd} = \sqrt{\frac{K_p}{K_i}} \tag{5}$$

This ensures that the integrator state never grows significantly beyond the physical saturation limit. As a secondary benefit, this mechanism also handles cases where the reference speed is physically unreachable, keeping the voltage at the maximum feasible limit without losing system responsiveness.

Listing 13: Implementation of PI-AntiWindup

```c
float PI(float error, int motor){
    float P = kp[motor] * error;
    static float I[2] = {0.0f,0.0f};

    /* antiwind up*/
    float v = I[motor] + P;
    float u = v;
    if (v > MAX_MOTOR_VOLTAGE){
        u = MAX_MOTOR_VOLTAGE; //saturation}
    else if (v < -MAX_MOTOR_VOLTAGE){
        u = -MAX_MOTOR_VOLTAGE;}
    float es = v - u;
    float kawd = sqrt (kp[motor]/ki[motor]);
    I[motor] = I[motor] + (ki[motor] * (error - kawd* es) )* TS;
    return u;}
```

3.2 Experimental Validation: The Case of Unreachable Reference

To experimentally validate the necessity and effectiveness of the anti-windup mechanism, a specific stress test was conducted on Motor 2. It should be noted that, while all other motor tests were performed in *Forward&Coast* (FC) mode, the anti-windup algorithm validation was conducted also in *Forward&Brake* (FB) mode. This configuration was retained from a preliminary robustness check designed to verify that the PI controllers, originally tuned for FC, remained stable in FB mode. Since in or case the anti-windup logic operates independently of the motor braking dynamic, the results obtained in FB are fully applicable to the FC configuration as well.
The motor was given a reference velocity step of 20 rad/s. Since the maximum physical speed of the motor under these conditions is approximately 15 rad/s, this reference is theoretically unreachable, forcing persistent system saturation.

The results, comparing the system response without and with the anti-windup algorithm, are presented in Figure 8 and Figure 9, respectively.

3.2.1 Analysis of Windup Phenomenon

Figure 8 illustrates the scenario without anti-windup. Because the motor cannot reach the target 20 rad/s, a positive error persists indefinitely. Consequently, the integral action of the PI controller continuously accumulates this error, causing the computed input signal (red line) to "explode", linearly increasing far beyond reasonable values (reaching over $80V$ in the plot).
It must be emphasized that a final safety saturation check implemented in the firmware guaranties that the actual physical voltage applied to the motor terminals never exceeds $6V$ regardless of the PI's output. Therefore, the motor speed (blue line) saturates at its physical maximum. However, the internal state of the integrator grows excessively.
The critical consequence is that if the reference were suddenly lowered to a reachable value (e.g., 5 rad/s), the motors would continue running at full speed ($6V$) for a significant period until this huge accumulated integral error unwinds, resulting in extremely sluggish and potentially dangerous control behavior.

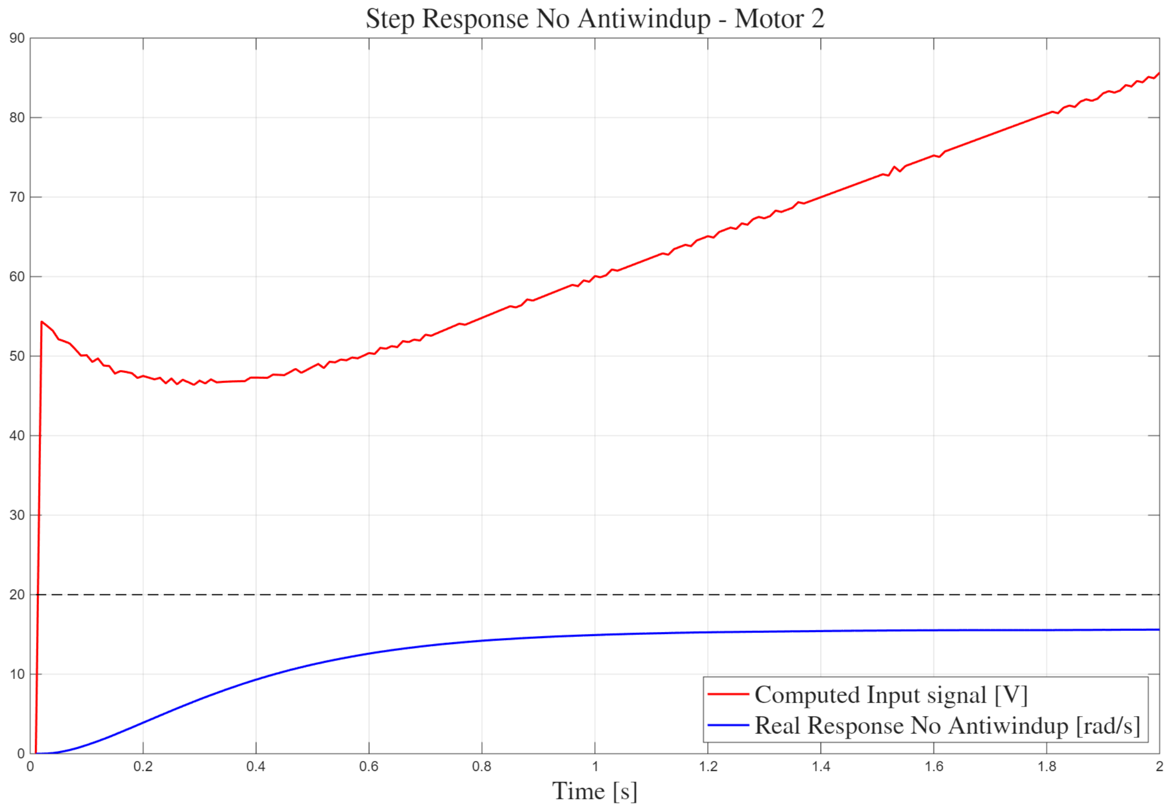


Figure 8: Motor 2 Closed-Loop response without AntiWindup and unreachable reference (20 rad/s)

3.2.2 Effectiveness of Anti-Windup Solution

Conversely, Figure 9 shows the behavior with the back-calculation anti-windup active. The computed control signal is immediately and correctly clamped at the $6V$ limit.
In this case, the voltage remains constant at $6V$ because the controller is exerting maximum allowable effort to reach the reference of 20 rad/s. However, thanks to the back-calculation, the integrator state does not wind up beyond the value corresponding to this limit. As a result, if the reference signal were to drop, the controller would be able to respond immediately, reducing the voltage and tracking the new reference without delay.

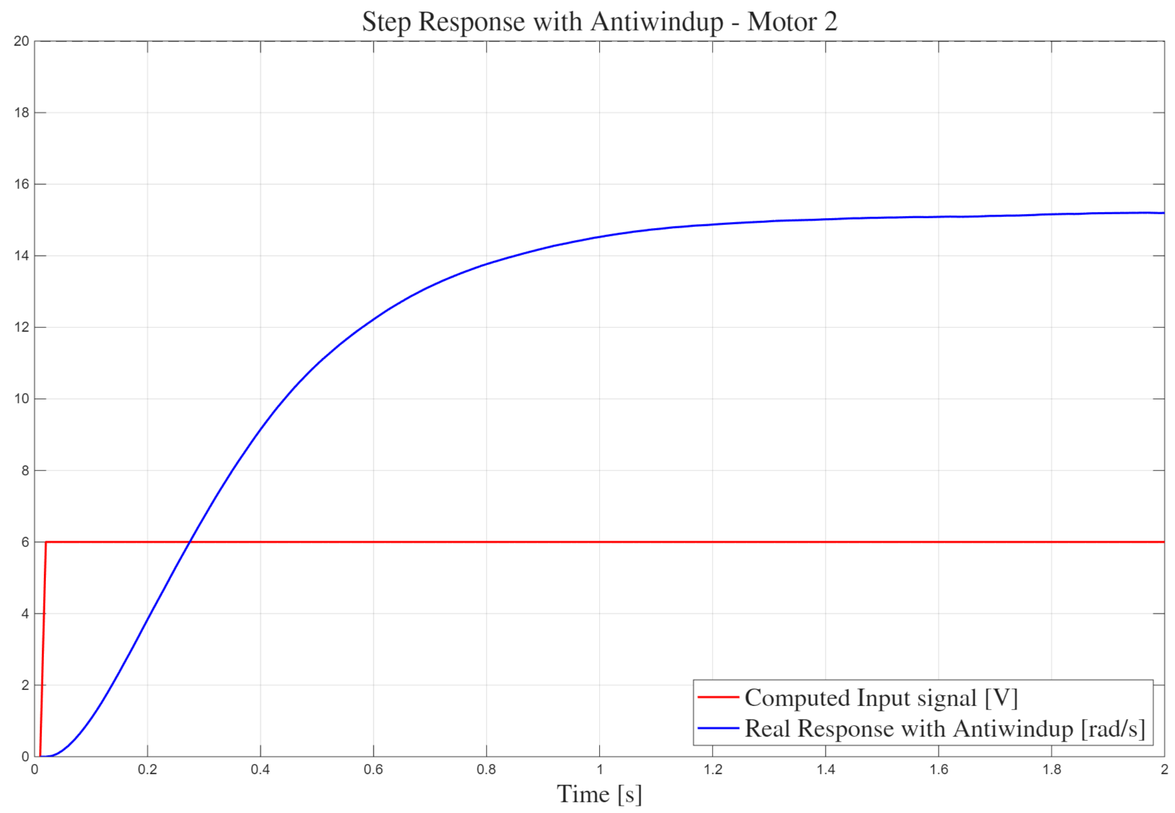


Figure 9: Motor 2 Closed-Loop response with AntiWindup and unreachable reference (20 rad/s)

3.3 Dynamic keypad speed

As the final feature of this laboratory, we implemented the control of motors via a keypad, allowing us to observe their behavior while changing the reference. The function used to set each motor's reference is a variant of the one adopted for dynamic blinking in Lab 1: instead of calling the blink function after entering a digit, we update the reference speed achieved by the PI controller. Since the system manages two motors and also allows for negative references, additional keypad functionalities were introduced by exploiting previously unused keys:

- `#`: sets the reference for both motors;
- `A`: inverts the reference sign and then sets it for both motors;
- `B`: sets the reference only for motor 1;
- `C`: sets the reference only for motor 2;
- `D`: inverts the reference sign and is intended to be used only before pressing a key that sets the reference, since the frequency update method is not compatible with negative values. For example, it is useful for assigning a negative speed to Motor 1 only.

The implementation of this function allowed us (see Figure 10) to compare the response of the motors to arbitrary reference changes. The responses to reference changes indeed overlap during the transient phase, even in the presence of large speed variations. This behavior highlight the effectiveness of the motor tuning process, ensuring motor synchronization that will allow for the straight-line motion of the TurtleBot[4].

[4]The corresponding video demonstration is available at:
`https://github.com/DanieleFila/Embedded_Lab/blob/main/Videos/LAB3_DynamicSpeedviaKeypad.mp4`

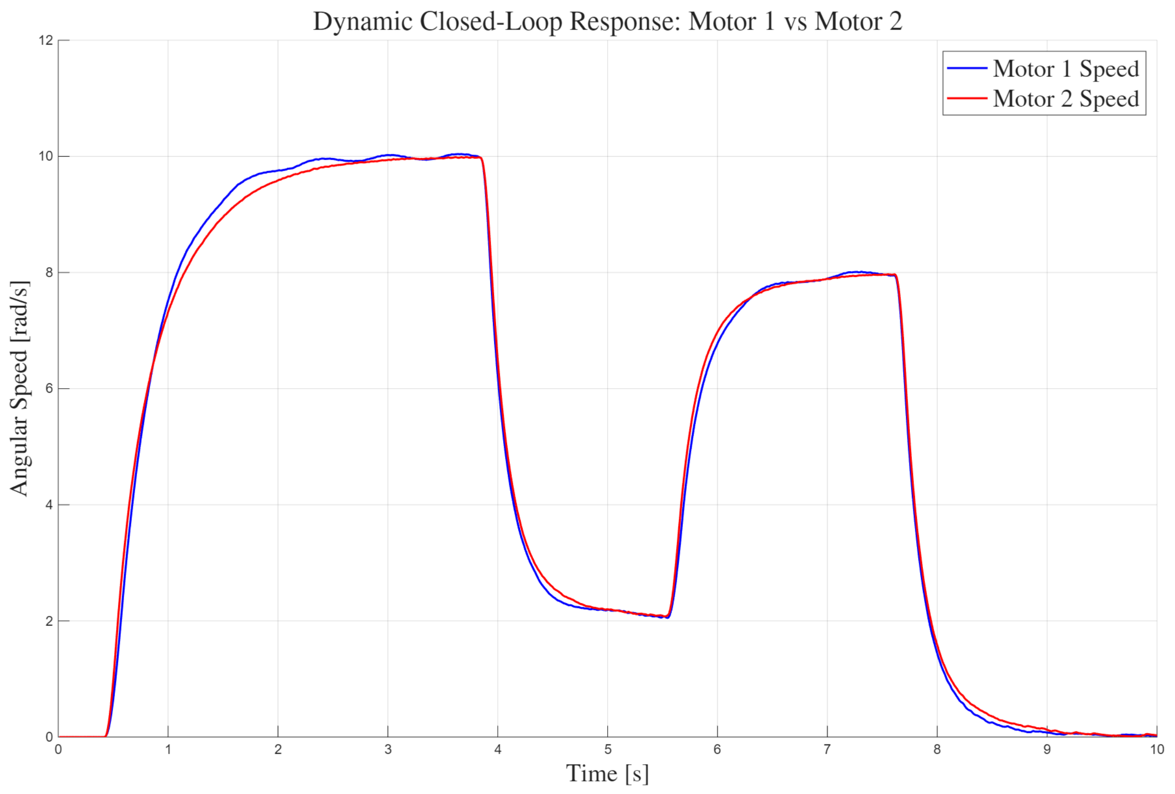


Figure 10: Motors response to changes in reference

4 Laboratory 4

The objective of this laboratory is to develop a line follower capable of completing the two proposed circuits (Figure 11 and Figure 12) with the best lap time. The first circuit has a bullet shape and represents a relatively simple track, while the second, hourglass-shaped, features tighter curves and frequent changes of direction, requiring more precise control. The experiment focuses not only on the calibration of the main system parameters (speed, controller gain) but also on ideating strategies to optimize performance on both tracks.

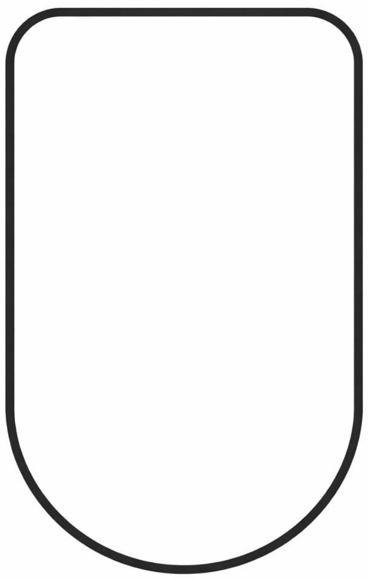


Figure 11: Circuit A

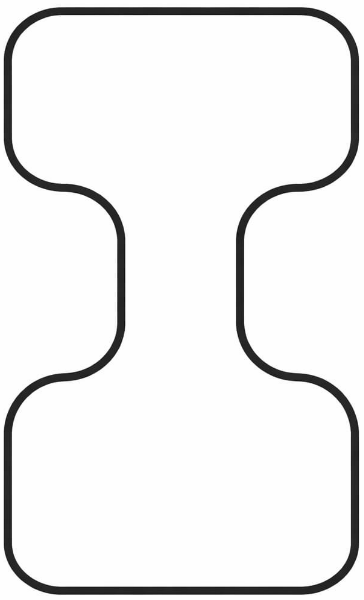


Figure 12: Circuit B

4.1 First Approach

The first simple approach consisted of a line following method based directly on the line sensor input. The idea was straightforward: if the line sensors indicated that the line was on the left, the TurtleBot would turn left (for example, by multiplying the wheel speeds by a factor of 1.5 for the right wheel and dividing by 1.5 for the left wheel), and vice versa if the line was on the right.
While this strategy allowed the robot to generally follow the line, it effectively implemented only two discrete states: turning left or turning right, without any continuous relationship to the actual position of the line. The control actions were not proportional to the line displacement, resulting in discontinuous and oscillating behavior rather than smooth tracking[5].

[5]The corresponding video demonstration is available at:
`https://github.com/DanieleFila/Embedded_Lab/blob/main/Videos/LAB4_FirstApproach_NoController.mp4`

4.2 Creating the P Controller

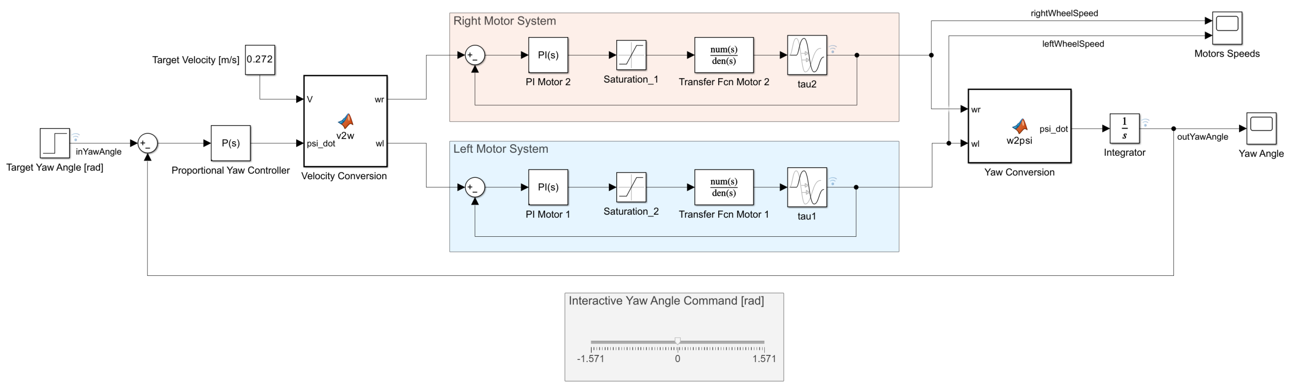


Figure 13: Complete System with Yaw Controller

The control system is implemented using a proportional (P) controller, as shown in the block diagram in Figure 13, which illustrates the entire control chain. The main objective of the controller is to maintain the yaw error at zero, ensuring that the TurtleBot remains centered on the line. The P controller receives the measured yaw error as input and calculates the corrective yaw command, which is then passed to the kinematic conversion block. This block translates the command into appropriate wheel velocities, which are finally applied to the motors, with the PI anti-windup controller implemented in *Laboratory 3*.
A proportional controller is sufficient for this application because the task primarily requires simple error correction. While a PD controller could have been used to improve transient response, it would introduce additional complexity, and the derivative term would amplify sensor noise. The P-only solution represents a simple and robust approach that achieves the desired performance with minimal tuning.

4.2.1 Calculating the error

Since the TurtleBot's goal is to remain centered on the line, we define the error as the distance between the ideal position (the center of the sensor array) and the detected line position. To estimate this position, a weighted average of the readings from the eight photodiodes is computed, where the weight assigned to each sensor is proportional to its distance from the center of the sensing module.
The function that implements the logic to calculate this error, called ESL, is shown in Listing 14

Listing 14: ESL Function

```c
float esl(uint8_t data){
    float numerator = 0.0f;
    uint8_t active_sensors = 0;
    float lambda;
    for (uint8_t n = 0; n < N_SENSORS; n++) {
        uint8_t bn = (data >> n) & 0x01; //equivalent to bn = data[n]
        if (bn == 1) {
            lambda = (((float)(N_SENSORS - 1) / 2.0f) - n) * PITCH;
            numerator += lambda;
            active_sensors++;
        }
    }
    // Avoid division by zero, happens when no sensor is active
    if (active_sensors == 0)
        return 0.0f;
    return numerator / active_sensors;
}
```

This estimation method is simple and effective, but it can be further refined. One potential improvement is the removal of outliers, which can help reduce noise caused by line imperfections, such as salt and pepper noise, as we will later see on *Circuit C*, or anomalous reflections due to dirt on the surface. In our tests, however, the introduction of such a filter was not necessary.

Another option involves assigning higher weights to the outer photodiodes, producing a more aggressive (non-linear) response when the robot deviates significantly from the line. This idea was implemented, not by modifying the error calculation itself, but by adjusting the controller. This approach achieves a more reactive behavior without altering the fundamental definition of the error.

4.2.2 Kinematic conversion

Based on the differential drive kinematic model described in class, and assuming a pure rolling condition (no slipping) for the wheels, we implemented the specific conversions required to interface the high-level path planning with the low-level motor drivers.
The robot geometry is defined by the wheel radius r and the track width D (distance between the two wheels).

1. From Control Inputs to Wheel Velocities

The control algorithm sets a desired constant linear velocity V and a target yaw rate $\dot{\psi}$. These global commands must be converted into individual angular velocities for the right (ω_r) and left (ω_l) wheels to drive the motors. Starting from the linear velocity equations derived from the geometry:

$$V_r = V + \dot{\psi}\frac{D}{2} \quad , \quad V_l = V - \dot{\psi}\frac{D}{2} \tag{6}$$

Since the relationship between linear and angular velocity is $V = \omega \cdot r$, the final reference commands sent to the PI controllers are:

$$\omega_r = \frac{V + \dot{\psi}\frac{D}{2}}{r} \quad , \quad \omega_l = \frac{V - \dot{\psi}\frac{D}{2}}{r} \tag{7}$$

2. From Wheel Velocities to Yaw Rate

Conversely, to estimate the actual turning rate of the robot (used for the feedback loop), we convert the measured angular velocities of the wheels back into the global yaw rate. The yaw rate is proportional to the difference in linear tangential velocities of the wheels:

$$\dot{\psi} = \frac{V_r - V_l}{D} \tag{8}$$

By substituting the angular velocities measured by the encoders, we finally obtain:

$$\dot{\psi} = \frac{r \cdot \omega_r - r \cdot \omega_l}{D} = \frac{r}{D}(\omega_r - \omega_l) \tag{9}$$

4.3 Simulation of the Yaw Control Strategy

Before proceeding with the firmware implementation on the microcontroller, we adopted a Model-Based Design approach to validate the steering logic using Simulink. This preliminary step was crucial to verify the kinematic relationships and the expected system behavior in a safe, virtual environment.

4.3.1 System Architecture and Workflow

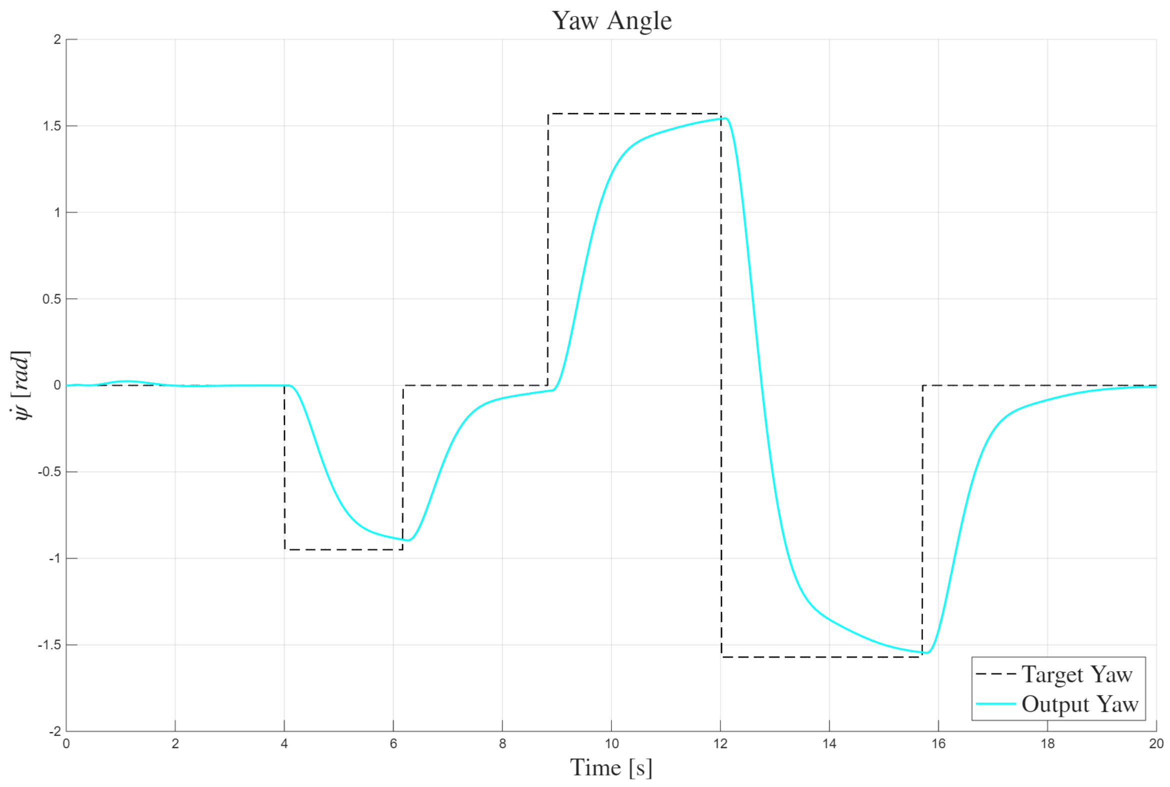
The simulation model, shown in Figure 13, integrates the components characterized in the previous laboratory sessions. Specifically, the "Left Motor System" and "Right Motor System" blocks contain the FOPTD transfer functions of the real actuators, regulated by the PI controllers tuned in the previous section.
The control architecture is designed as follows:

- **Inputs:** The system receives a constant linear velocity reference ($v = 0.272$ m/s, corresponding to ≈ 8 rad/s for the wheels) and a variable Target Yaw Angle (ψ). While in the real application this angle is computed by the line-following algorithm, in this simulation it is modeled as a step input variable ranging from $-\pi/2$ to $+\pi/2$.
- **Kinematic Conversion:** The error between the target and actual yaw is processed via a proportional controller. A kinematic block then converts the global velocity pair $(v, \dot{\psi})$ into individual angular velocity references for the left and right wheels (w_L, w_R).
- **Feedback:** The outputs of the motor models are converted back into the robot's global heading (Yaw Angle) via integration, closing the loop.

To streamline the testing process, the simulation environment is automated by specific Matlab scripts. A setup script initializes all variables (motor parameters, PI gains) before execution. During runtime, an interactive slider allows the user to dynamically adjust the Target Yaw input. Finally, a post-processing script is automatically triggered at the end of the simulation to extract data and generate clean plots for analysis.

4.3.2 Simulation Results

The dynamic behavior of the system under a varied sequence of commands is illustrated in the following plots. The top plot of Figure 14 displays the Yaw Angle response. Unlike a simple static test, here the target (dashed line) is varied dynamically to cover the full operating range, requesting turns to various angles including sharp turns up to $+\pi/2$ and $-\pi/2$. The system output (cyan line) effectively tracks these changing reference steps.
Correspondingly, the bottom plot of Figure 14 details the wheels speed modulation. It can be observed that the system maintains the base linear velocity ($\approx 8\ rad/s$) as an equilibrium point. To execute the steering commands, the differential drive logic forces the wheel speeds to diverge significantly. For instance, at $t \approx 12$s, to perform the sharp negative turn (transitioning from $+\pi/2$ to $-\pi/2$), the left wheel accelerates to nearly 16 rad/s, while the right wheel drops almost to 0 rad/s, generating the maximum possible rotational moment.
Finally, a specific detail can be observed in the initial startup phase ($0 < t < 2$ s). A slight deviation from the zero-yaw target occurs even before any steering command is issued. This behavior is expected and is due to the acceleration transient: as the motors ramp up to the cruising velocity ($0.272 m/s \approx 8\ rad/s$), the inherent physical asymmetry between the left and right motor models (different gains and time constants) generates a temporary imbalance. However, the simulation confirms the robustness of the control logic: the feedback loop immediately detects this uncommanded rotation and compensates for it, stabilizing the robot's heading before the first steering step is applied.



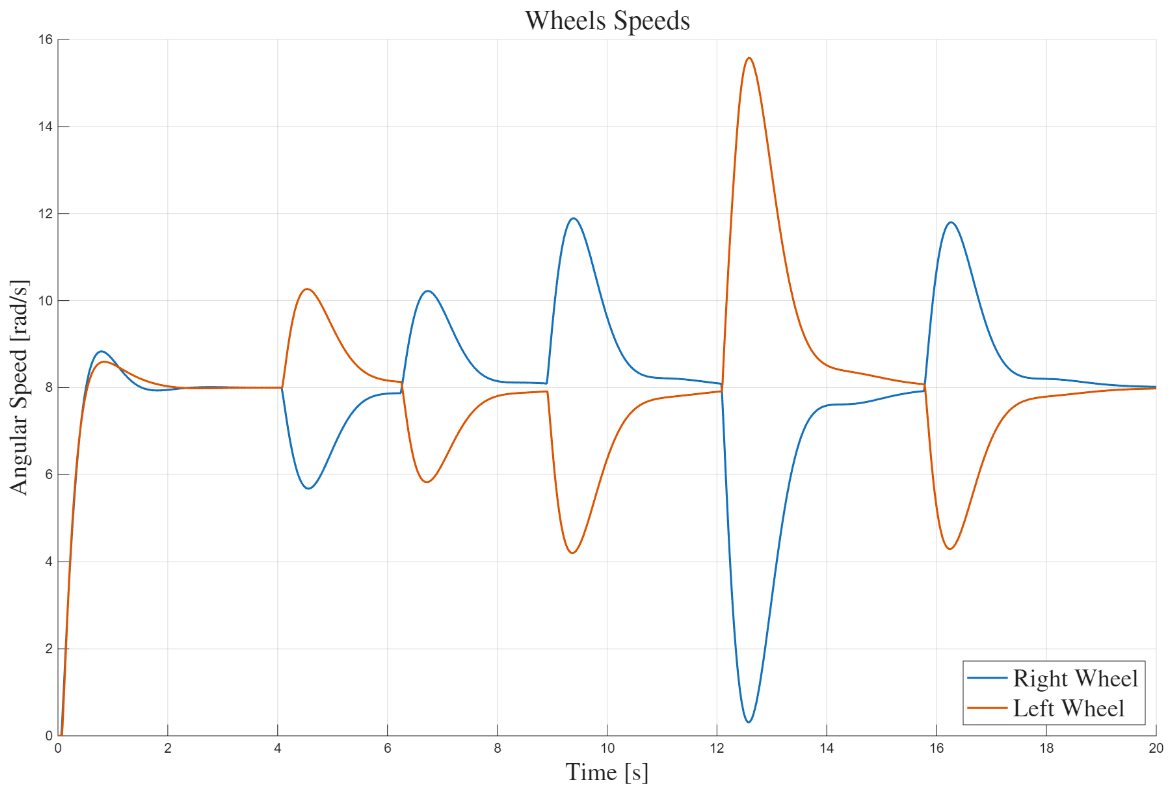


Figure 14: Simulation Results of the Yaw Control Strategy. Top: Yaw Angle Tracking, showing how the system follows the dynamic steps. Bottom: Wheel Velocities, showing the differential speed modulation required to execute those turns.

4.4 Tuning the Control Parameters

Once the controller logic was implemented, it was necessary to tune the key parameters to achieve the best lap time. The main focus was on:

- the reference speed;
- the proportional gain K;
- the motor driving mode (FC / FB).

4.4.1 Choice of Driving Mode: FC vs FB

Initially, the FB (Forward and Brake) mode was considered the most suitable, as it allows for rapid braking and theoretically higher responsiveness in curves. However, experimental tests revealed the opposite behavior: braking was excessively abrupt, causing the TurtleBot to tilt and generate significant vibrations.
This effect occurred because the inner wheel in a turn experienced a very rapid reduction in speed, while the outer wheel, which needed to accelerate simultaneously, could not respond as quickly, resulting in an unbalanced impulsive torque.
To achieve smoother and therefore faster behavior, the FC (Forward and Coast) mode was adopted. This mode ensures gentler transitions and more symmetrical motion between the two wheels, improving overall stability and lap performance.

4.4.2 Choice of reference Speed

In theory, the reference speed (in the code named `MaxSpeed`) should be set to the maximum physically achievable by the robot, since the velocity of the TurtleBot is the mean of the two wheels

$$v = \frac{\omega_L + \omega_R}{2} \approx \text{reference Speed.}$$

In practice, however, limitations arise due to the maximum speed of the motors. Therefore, a value of `MaxSpeed = 12 rad/s` was selected, slightly below the maximum speed of the right motor and lowered later if needed (see Laboratory 5). This choice allows smoother navigation through curves: the inner wheel naturally slows down while the outer wheel accelerates, resulting in a higher average speed over the track.

4.4.3 Choice of the Proportional Gain K

We observed that the proportional gain K is closely related to the reference speed. As the robot's speed increases, it becomes easier to deviate from the line, making a sufficiently aggressive controller necessary to quickly correct errors. For the selected reference speed, we experimentally determined the minimum value of K that allowed the TurtleBot to reliably complete the entire circuit without leaving the track.
While this tuning approach worked well for Circuit B, which features many more curves and where the TurtleBot generally moves at a lower speed when entering corners, the resulting gain for Circuit A was relatively high. This was necessary to handle the 90-degree curve after the straight section, where the robot reaches its maximum speed. Such a high value of K introduces oscillations along straight segments due to aggressive corrections, which, in turn, reduces the effective speed.

To mitigate this behavior, we slightly modified the P-controller by adopting a variable gain approach:

- the default value of K is reduced to avoid aggressive corrections when the robot is nearly centered on the line, such as on straight sections;
- K is increased and multiplied by an experimentally determined factor when a significant deviation is detected, typically when approaching a curve

This effectively changes the controller's response from purely linear (the error is linear with respect to the misalignment, and the P is clearly linear) without modifying the error computation itself. We preferred this approach over altering the error (for example, we would have obtained similar results by squaring it) because it is more customizable and easier to tune.
This strategy provides smoother behavior on straight paths, avoiding unnecessary aggressive steering while maintaining high responsiveness in curves, resulting in a slight improvement in overall lap time (on the order of half a second). The final implementation of the P controller is shown in Listing 15.

Listing 15: Controller P

```
float controller = MAXSPEED*1.3f;

float esl_calculated = esl(lineSensorData);
float yaw_error = esl_calculated / HDISTANCE;

if (yaw_error > 3.0f / HDISTANCE){
    yaw_error = yaw_error*1.5f;
}else if (yaw_error< -(3.0f / HDISTANCE)){
    yaw_error = yaw_error*1.5f;
}

float yaw = controller * yaw_error;
```

4.4.4 Parameters Calibration via Keypad

The calibration of the reference speed and K was performed at runtime using the keypad, eliminating the need to manually modify the values in the code for each test. To facilitate tuning, the input handling logic was modified so that:

- pressing the `A` key assigns the entered value to the reference speed;
- pressing the `#` key assigns the value of K according to the relation:

$$K = \text{Reference} \cdot \frac{\text{entered\_value}}{10}$$

Here, K is not an arbitrary value, but depends on the reference speed, reflecting the proportional relationship between controller gain and reference speed.
This procedure allowed for the rapid adjustment of the parameters during field tests, drastically reducing optimization time and enabling an empirical search for the best compromise between stability, speed, and responsiveness.

4.5 On Track Performance Analysis

In this section, we present the analysis of the telemetry data acquired during the navigation of specific test circuits. The focus is on evaluating the dynamic response of the actuators and the effectiveness of the differential drive strategy in handling real-world track topologies, characterized by varying curvature radii and straights.

4.5.1 Circuit A (Bullet Track)

The first experimental validation was conducted on *Circuit A* [6], a track characterized by a bullet-like shape consisting of two straight sections connected by two sharp 90-degree corners and one wide semi-circular turn (see Figure 11). The robot traversed the track in a clockwise direction.
The recorded angular velocities of the motors are presented in Figure 15. By analyzing the time-domain data, we can accurately reconstruct the robot's path:

- **Startup and Steady State** $(0 < t < 4\text{s})$**:** The initial phase shows the motors ramping up to the cruising speed (≈ 6 rad/s). Once the setpoint is reached, the robot proceeds along the first straight section.
- **First 90° Turn** $(t \approx 5\text{s})$**:** A sharp divergence in velocities is observed. Since the track is clockwise (right turns), the left wheel accelerates significantly while the right wheel decelerates. This differential action corresponds to the first sharp corner.
- **Short Straight** $(6s < t < 8s)$**:** Following the first corner, the velocities reconverge to the average value for approximately 2 seconds, corresponding to the short straight segment connecting the two corners.
- **Second 90° Turn** $(t \approx 9\text{s})$**:** A second sharp right turn occurs, exhibiting a velocity profile almost identical to the first one.
- **Wide Arc** $(t > 13\text{s})$**:** Starting from $t \approx 13$s, the behavior changes. Instead of a sharp spike, we observe a sustained separation between the two signals, with ω_l remaining consistently higher than ω_r. This indicates that the robot is negotiating the long, constant-curvature semi-circle to complete the lap.

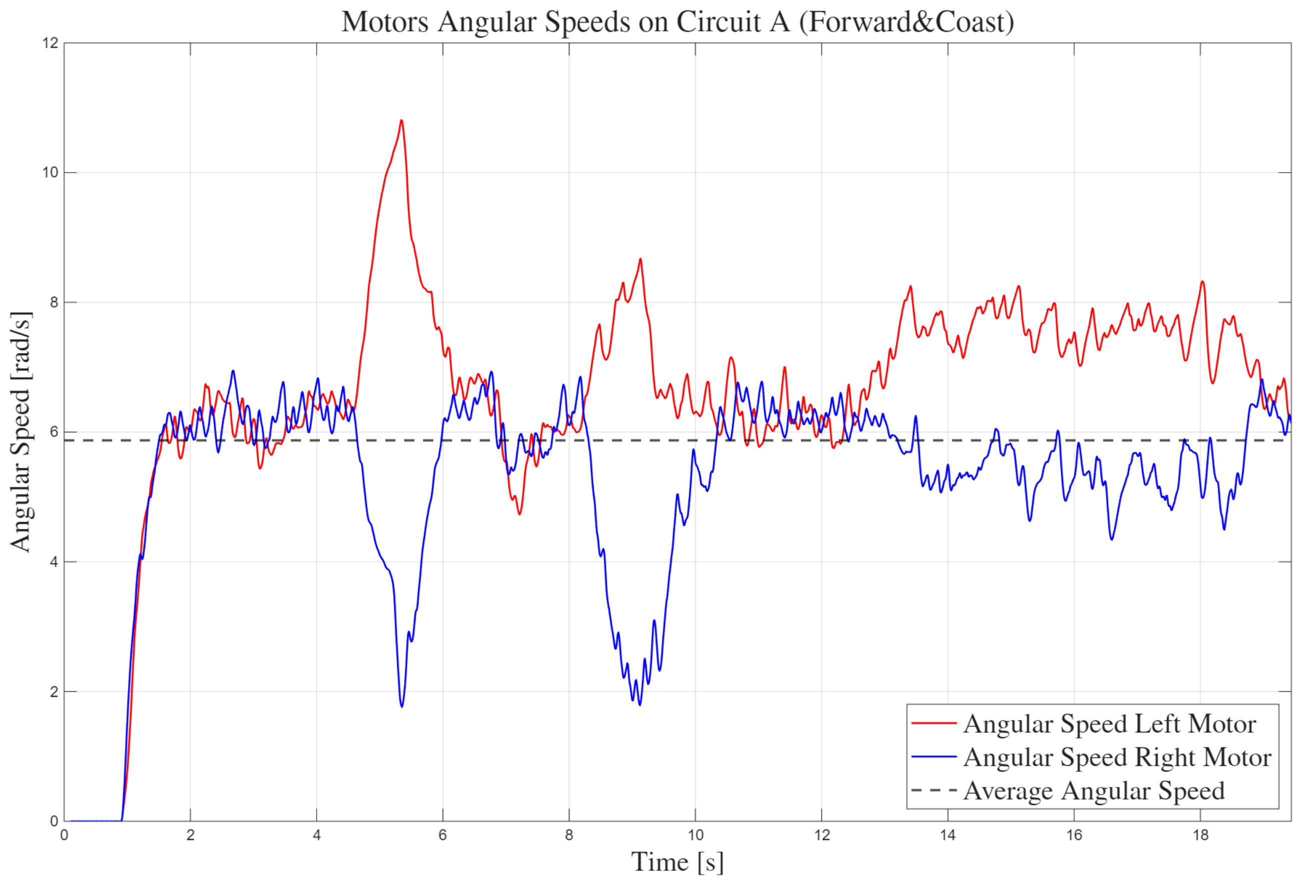


Figure 15: Experimental Angular Velocities on Circuit A. The plot clearly shows the sequence: straight → turn → straight → turn → straight → wide arc.

[6]A video of the TurtleBot completing this circuit is available at:
`https://github.com/DanieleFila/Embedded_Lab/blob/main/Videos/LAB4_BulletTrack.mp4`

4.5.2 Circuit B (Clepsydra Track)

The second test was performed on *Circuit B*[7], which features a more complex hourglass geometry (Figure 12). Unlike the previous track, this circuit introduces alternating curvatures, requiring the robot to manage quick transitions between right and left turns. The track was navigated in a clockwise direction.

Dynamic Behavior Analysis

The angular velocities recorded in Forward & Coast mode are shown in Figure 16. The complexity of the track is reflected in the highly dynamic nature of the control signals.
By analyzing the plot, we can observe that the wheels rarely have time to stabilize at a constant cruising speed. The sequence of turns is so tight (e.g., at $t \approx 11$s and $t \approx 13$s) that the motors are in a constant state of transient response.
An interesting event occurs in the central section of the hourglass ($10s < t < 15s$). Specifically, around $t \approx 12$s, we observe a sharp velocity inversion: the robot executes a hard right turn (left motor peaks, right motor drops), followed immediately by a hard left turn. This rapid switch demonstrates the system's ability to handle rapid turn changes.
The straight sections are identifiable only by brief intervals where the signals converge, specifically at the start/end of the lap and in the middle section ($17s < t < 19s$).

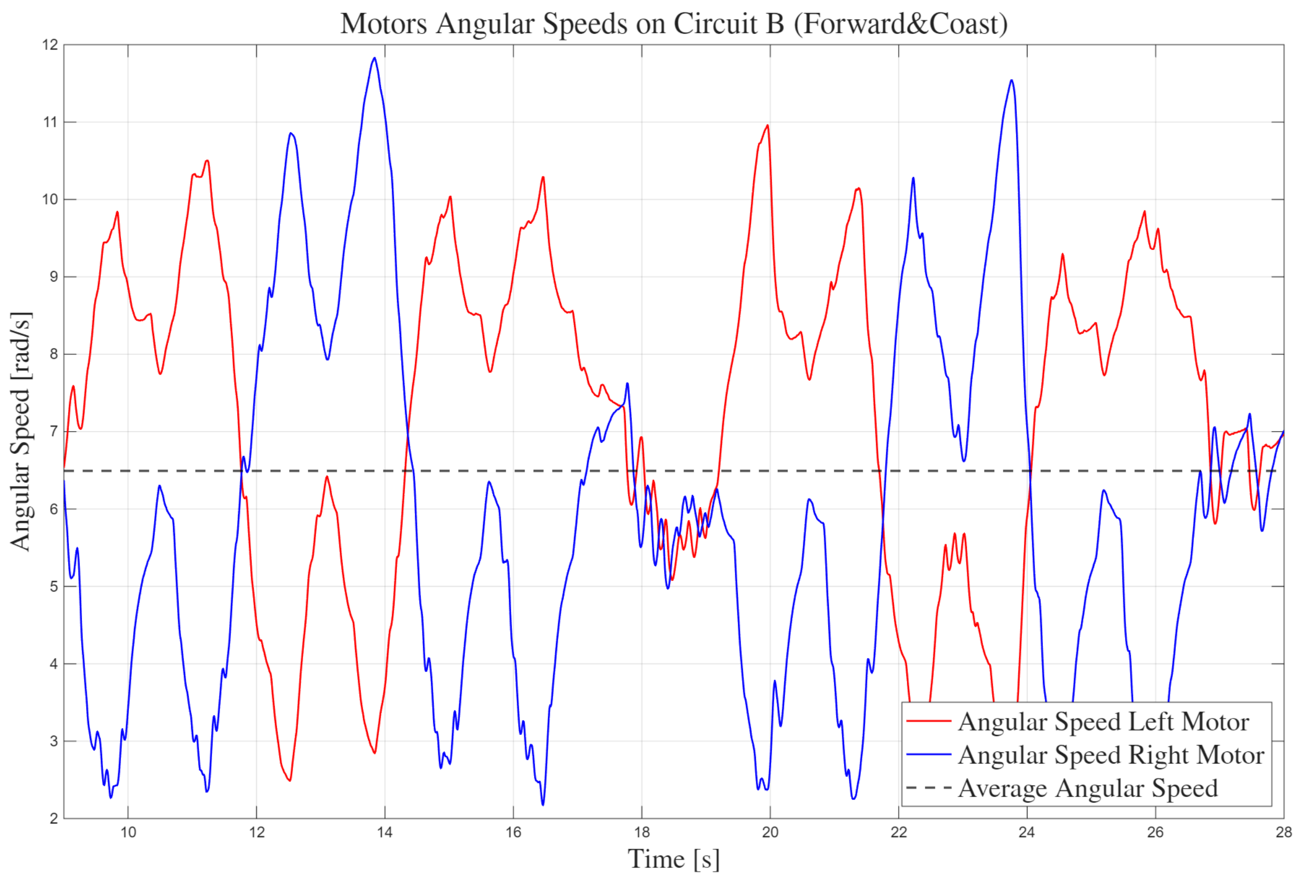


Figure 16: Experimental Angular Velocities on Circuit B.

4.5.3 Comparison with Simulation Dynamics

It is worth noting that the velocity profiles in the real experiment are not as smooth as those obtained in the Simulink environment. The real data exhibits high-frequency oscillations (noise) around the reference values.
This behavior is expected and is intrinsic to the nature of the Proportional line-following controller. Unlike the simulation where the exact yaw error is known continuously, the real sensor provides discrete position feedback. Consequently, the controller continuously applies small corrective actions (micro-steering) to keep the line centered, resulting in the observed oscillatory texture of the plot.

[7]A video of the TurtleBot completing this circuit is available at:
`https://github.com/DanieleFila/Embedded_Lab/blob/main/Videos/LAB4_HourglassTrack.mp4`

4.5.4 Quantitative Performance Summary

To provide a concrete assessment of the controller's efficiency, a post-processing analysis was performed on the telemetry data using a dedicated Matlab script. The methodology consisted of three calculation steps:

1. **Average Angular Velocity:** The recorded samples of both left and right motors were averaged to find the global mean angular speed of the robot: $\bar{\omega} = \text{mean}([\bar{\omega}_l, \bar{\omega}_r])$.
2. **Average Linear Velocity:** This value was converted into the robot's linear speed using the known wheel radius ($r = 0.034$ m).
3. **Estimated Distance:** Finally, the total track length was estimated by multiplying the average linear velocity by the measured lap time.

Table 7 summarizes the results of all conducted tests. It is important to note that the recorded lap times are intended for demonstrative purposes only and do not represent the absolute best performance achievable. For instance, while Circuit A features fewer curves and would theoretically allow for a faster completion time, the primary objective of these tests was to validate the navigation stability rather than to optimize for speed.

Table 7: Experimental Performance Metrics Summary

Test Case	**Lap Time** [s]	**Avg. Speed** $\bar{\omega}$ [rad/s]	**Avg. Linear Vel.** $\bar{v}$ [m/s]	**Est. Distance** L [m]
Circuit A (Bullet) - FC	19.38	5.87	0.20	3.87
Circuit B (Clepsydra) - FC	22.07	6.49	0.22	4.87

5 Appendix: Laboratory 5

The objective of this laboratory is to re-implement the code developed in the previous labs, organizing it into tasks that are then scheduled by the FreeRTOS operating system. Additionally, this lab addresses the navigation of a new track, characterized by more challenging conditions compared to the previous ones.

5.1 FreeRtos

We can easily notice that the original algorithm is essentially sequential. The introduction of multiple tasks is therefore not a functional necessity but an opportunity to structure and modularize the code.
In the previous implementation, a hardware timer was used to trigger the control loop every 10 ms. In this lab, we instead use the function `osDelayUntil(10)`, which allows the operating system to resume the task at precise, regular intervals, independent of the task's execution time. This approach eliminates the jitter that would occur if we had used `osDelay(10)`, which only pauses the task for a fixed duration without compensating for the time already spent executing the task.

The system is structured into three main tasks:

- `lineTask`: responsible for acquiring data from the line sensor.
- `controlTask`: computes the control actions, from the P controller, which calculates wheel speed, to setting the voltage to the motors calculated by the PI anti-windup controller, and also acquires telemetry data.
- `communicationTask`: handles the transmission of telemetry data to the PC.

Since communication is not as time-critical as the other functions, the `communicationTask` has been assigned a lower priority to ensure that control and sensor tasks can execute without interruption.

5.1.1 Queues

Since the system has been divided into separate tasks, variables no longer have a global scope. Therefore, data must either be stored in global variables or shared using FreeRTOS mechanisms.
In our implementation, two types of communication are required:

1. The output of the line sensor, acquired by `lineTask`, must be sent to the `controlTask`;
2. Telemetry data, acquired by the `controlTask`, must be sent to the `communicationTask`.

The telemetry data produced by the `controlTask` is structured and more complex; for simplicity, it is shared using a global variable. In contrast, the line sensor data is a simple integer and can be efficiently transmitted using a FreeRTOS queue. Specifically, a Message Queue is sufficient for this purpose, whereas a Mail Queue is intended for larger or more complex memory blocks and would be unnecessary overhead. The functions used for respectively sending and receiving line sensor data are shown in Listing 16:

Listing 16: Queue Function

```
osMessageQueuePut(SensorDataQueueHandle, &lineSensorData, 0, 0);
osMessageQueueGet(SensorDataQueueHandle, &lineSensorData, 0, 0);
```

The two parameters set to zero represent, respectively, the message priority (not supported by FreeRTOS and therefore always `0`) and the maximum time to wait for incoming data if the queue is empty. A value of `0` means "never wait, return immediately." Configuring the queue with a zero timeout avoids unnecessary delays.

5.1.2 Semaphores

To ensure proper synchronization and preserve the sequential order between line sensor data acquisition and its subsequent processing, a binary semaphore was introduced. This mechanism allows the activation of the `controlTask` to be conditioned on the completion of a line sensor acquisition cycle, maintaining the causal order present in the original code.
The implementation is straightforward: The functions used for respectively acquiring and releasing a binary semaphore are shown in Listing 17:

Listing 17: Semaphore Function

```
osSemaphoreAcquire(binarySemSyncHandle, osWaitForever);
osSemaphoreRelease(binarySemSyncHandle);
```

With this setup, the `controlTask` remains suspended until the `lineTask` releases the semaphore. Since the parameter `osWaitForever` is used, the task waits indefinitely, guaranteeing that each controller iteration always uses the most recent measurements.

5.1.3 Mutex

In preemptive real-time systems, concurrent access to shared resources may lead to race conditions and inconsistent system states. In our application, for instance, while the `communicationTask` (low priority) is transmitting telemetry data, a preemption by the higher-priority `controlTask` could modify the same data structure, resulting in corrupted or incoherent information being sent.
To prevent such scenarios, a mutex is employed. A mutex is a binary semaphore that implements priority inheritance, a mechanism designed to avoid priority inversion. Specifically, when a low-priority task holds a mutex required by a higher-priority task, the scheduler temporarily raises the priority of the holding task, allowing it to complete its critical section as soon as possible. Once the mutex is released, the task's original priority is restored.
The implementation in FreeRTOS is simple, as can be shown in Listing 18

Listing 18: Mutex Function

```
osMutexAcquire(DataMutexHandle, osWaitForever);
//execution of critical task
osMutexRelease(DataMutexHandle);
```

In order to guarantee reliable and consistent execution of both the control loop and the communication subsystem, it may reveal necessary to protect also other critical operations in which concurrent access could lead to data corruption. For this reason, mutexes should be used to ensure the atomic execution of sensitive sections of code, including the computation of the control laws, the actuation of the motors, and the assignment of data to telemetry structures

5.2 A noisy Track

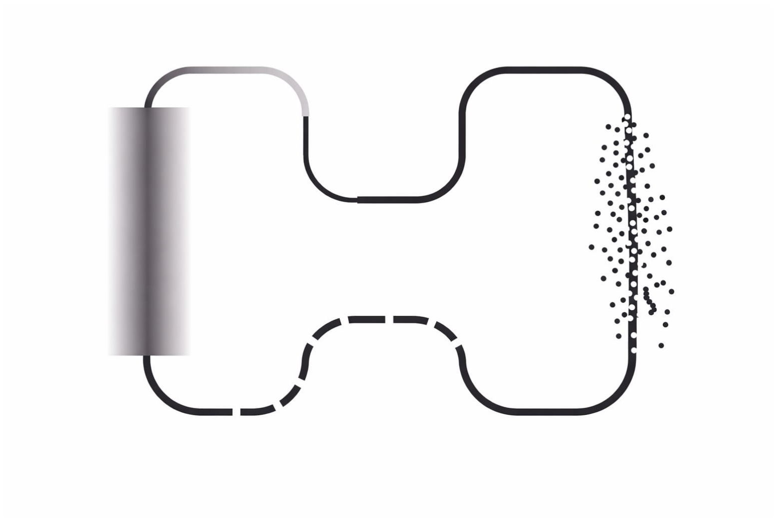


Figure 17: Noisy Track

The last challenge of this laboratory experience was to tackle a noisy track, shown in Figure 17, which presents several irregularities that were not present in previous scenarios. Yet, the implemented control strategy is able to handle them without requiring modifications, just a recalibration of the control loop. The following analysis describes the robot behavior while traversing the track clockwise:

- **Thin and wide lines**: The ESL computation does not assume any specific line width model; therefore, variations in line thickness do not affect the controller performance.
- **Salt–and–pepper noise**: Although the line remains perceivable, the robot exhibits noticeable lateral oscillations due to rapid variations in the ESL. Nevertheless, the TurtleBot remains centered on the line. Outlier filtering could have been applied, but it proved unnecessary.;
- **Dashed line**: In the absence of visible line segments, the ESL becomes zero, and the robot proceeds straight ahead. As long as the visible segments are aligned and the TurtleBot is able to recenter itself before each segment ends, it successfully reaches the next one;

- **Blurred line**: Since the sensor operates through thresholding and does not detect grayscale values, the blurred line is effectively interpreted as a wider line and therefore does not pose any issues. Moreover, the main central line remains clearly visible, so this does not represent a significant problem for the application;
- **Gray line** (critical section): This represents the main challenge, as the robot does not detect the line at all. Unlike the dashed case, there are no visible segments that allow reacquisition. Moreover, the gray line contains a 90 degree curve that cannot be inferred from the line sensor measurements, representing the major challenge of this track, since straight segments can be overcome simply by continuing straight.

As a result, this scenario cannot be handled solely through the previously implemented feedback loop. The only viable solution consists of introducing a logic based on prior knowledge of the track layout.
When traversing the track counterclockwise, the thin line preceding the gray curve could potentially be used as a marker, since it activates only one or two photodiodes of the line sensor, whereas normal lines typically activate two to three. However, the robot is not always perfectly centered, and similar sensor patterns may occur in other regions, making this approach unreliable without additional handling.
Conversely, when traversing the track clockwise, the gray curve is preceded by a long straight section in which the line is completely undetected. This feature allows discrimination between the dashed-line section (characterized by short losses of detection) and the gray line (characterized by prolonged loss of detection). The implemented logic, therefore, consists of monitoring the duration of line loss: if the line is not detected for a number of consecutive samples exceeding a predefined threshold, the robot initiates a right rotation (negative yaw).
This solution remains an open-loop approach and is therefore sensitive to variations in speed, battery charge level, and the exact position of the TurtleBot relative to the line. As a result, tuning the four main parameters (maximum speed, proportional controller gain, delay before initiating the right turn, and steering intensity during the maneuver) was particularly challenging since they are not independent and strongly influence each other. For this reason, after some trial and error, we devised a calibration procedure to achieve stable and reliable behavior.
First, the maximum speed and the aggressiveness of the P controller were tuned to reliably traverse the dashed-line section. The maximum speed was reduced compared to previous tracks, since maintaining track adherence is more critical than completing the circuit quickly. Conversely, the P controller gain was increased to ensure that the TurtleBot remains tightly centered on the line, which is essential in the dashed sections, where slight curvature combined with off-center positioning could cause the robot to leave the track.
Once the TurtleBot was able to complete the entire track, except for the gray line section, a strategy was implemented to detect prolonged line loss and trigger a corrective maneuver. This was achieved using a counter within the control loop, incremented at each cycle (10 ms per iteration) when no line was detected and reset whenever the line reappeared. The threshold for activating the turn has been chosen to be longer than the duration required to cross dashed segments and approximately coincident with the beginning of the gray curve.
Subsequently, the artificial yaw error used to induce the right turn was calibrated so that the robot appeared to continue tracking the line, despite executing an open-loop maneuver, as shown in the uploaded videos. With the resulting tuning, the robot is able to overcome the critical section in a sufficiently repeatable manner, completing the circuit correctly an average of three consecutive times, with a peak performance of seven consecutive successful laps.
Moreover, when failures occurred, the robot typically recovered by continuing to rotate to the right, eventually reaching the dashed-line section and rejoining the track[8]. The code fragment implementing this logic is reported in Listing 8, where it can be observed that after a certain duration of rightward rotation, the maneuver is intentionally stopped, allowing the TurtleBot to resume straight motion toward the blurred line section and preventing excessive turning while enabling successful reacquisition of the track. This logic has been implemented by simply adding the code shown in Listing 19, artificially manipulating the yaw error before it is applied by the P controller, hence before the last line of code shown in Listing 15.

Listing 19: Manipultaion of Yaw Error

```
    if (yaw_error == 0) {
        counter++;
    }else {counter = 0;}
    if (counter>40){
        yaw_error = -0.01f / HDISTANCE;
    }else if (counter > 65){
        counter = 0;
    }
```

[8]A video of the TurtleBot completing this circuit is available at:
https://github.com/DanieleFila/Embedded_Lab/blob/main/Videos/LAB5_NoisyTrack.mp4