



Republic of Tunisia

Ministry of Higher Education and Scientific Research

University of Monastir

Higher Institute of Computer Science and Mathematics of Monastir



Digital Control of Processes : Application on Microcontroller

Specialty:

2nd year of Engineering in Microelectronics

Realized by:

Kayoum Djedidi, Malak Guedouar

Mariem Hizem, Wijden Elmetamri

ROS based Digital PID Controller for DC Motors

Supervised by: Mrs. Imen BEN AMEUR

Academic Year: 2023 / 2024

Contents

General introduction	3
1 IMPLEMENTATION AND TESTING	4
1.1 Introduction	5
1.2 Objectives	5
1.3 materials used	5
1.3.1 Raspberry Pi 4 Model B	5
1.3.2 L298N motor driver	6
1.3.3 Motor Reducer JGA25-371 12V 18RPM	6
1.4 IMPLEMENTATION AND TESTING	7
1.4.1 Project Preparation	7
1.4.2 Control Settings	13
1.4.3 User Interface Design	18
1.4.4 ROS Package and Node Development	20
1.4.5 Conclusion	22
General conclusion	22

General introduction

The development of digital control systems for direct current (DC) motors is crucial in many industrial and research fields. These systems enable precise regulation of motor speed and position, essential for applications such as robotics, industrial automation, and embedded systems.

DC motors are widely used due to their simplicity, efficiency, and ease of control. They provide high torque at low speeds and are found in various applications, from household appliances to industrial machines.

The digital control of DC motors presents several challenges, including understanding electronics principles, dynamic system modeling, and control theory. Integrating sensors, acquiring real-time data, and interacting with user interfaces add further complexity.

To address these challenges, we created a modular DC Motor controller with an object-oriented design integrated with the Robot Operating System (ROS), with PID correction capabilities. Coupled with an interactive user interface and the ROS tools and libraries to build robust robot applications, this project facilitates real-time data acquisition and visualization and simplifies system component integration.

In summary, this project aims to design and implement a digital control system for DC motors, ensuring a robust, flexible, and efficient control system for various applications.

[This is the link to this projects' Github repo.](#)

1 IMPLEMENTATION AND TESTING

1.1 Introduction

In this chapter, we will detail the overall layout of the project and outline the steps needed to implement the various sections and elements of the system. The UI is designed to ensure intuitive and efficient interaction, providing users with clear access to data monitoring, control settings, and system configuration.

1.2 Objectives

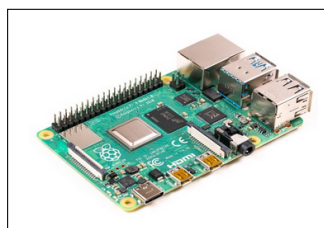
The aim of our project is to implement a digital speed control system using a Raspberry Pi. The objectives include:

- **Modeling of DC Motors:** This section covers the modeling of a direct current (DC) motor, including theoretical modeling from first principles and experimental open-loop modeling.
- **Speed Control:** This part starts with basic proportional control and then progresses to PI (Proportional-Integral) and PID (Proportional-Integral-Derivative) control. It also includes the design of a digital control system according to specified requirements.
- **Position Control:** This section involves the design and implementation of a position controller for a DC motor.
- **ROS Integration:** The `ros_integration` script integrates a motor control system with ROS, providing basic control and real-time monitoring features, note that it doesn't currently work together with the control interface.

1.3 materials used

1.3.1 Raspberry Pi 4 Model B

The Raspberry Pi 4 Model B is a versatile single-board computer developed by the Raspberry Pi Foundation. It features a powerful quad-core ARM Cortex-A72 processor running at up to 1.5 GHz, along with options for 2GB, 4GB, or 8GB of RAM, providing significant improvements in performance compared to its predecessors.

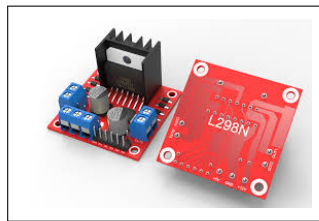


1. IMPLEMENTATION AND TESTING

1.3.2 L298N motor driver

The L298N motor driver is widely used in robotics and automation projects for its high current handling capacity and ease of use. Key specifications include:

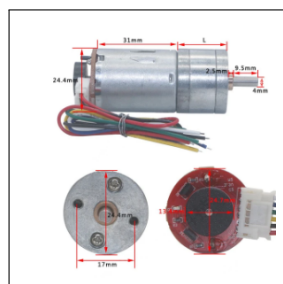
- Operating Voltage: 5 V to 35 V
- Maximum Output Current: 2 A per channel
- Number of Channels: 2 (can control two DC motors independently)
- Peak Current: 3 A per channel
- Control Logic Voltage: 5 V
- Built-in diodes: For back EMF protection



1.3.3 Motor Reducer JGA25-371 12V 18RPM

This JGA25-371 DC gear motor has a built-in encoder with 12 counts per turn, ensuring precise control of engine speed.

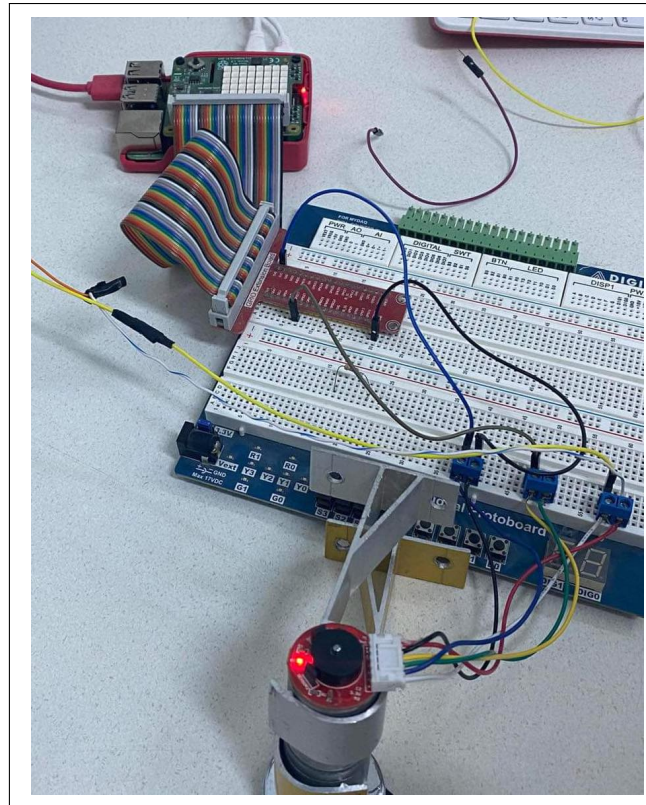
- Operating voltage: between 6 V and 24 V
- 12 V vacuum current: 46 mA
- 12 V stall current: 1 A
- Locking torque at 12 V: 28 kg cm
- Reducer size: 25 mm
- Weight: 99 g



1.4 IMPLEMENTATION AND TESTING

1.4.1 Project Preparation

1.4.1.1 Measure motor specifications and verify with datasheet

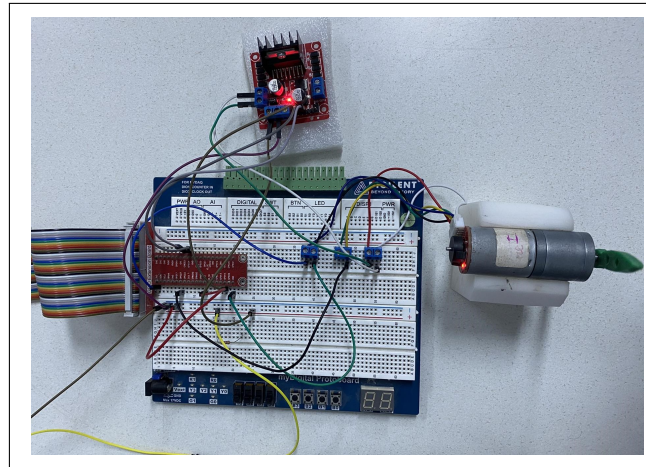


These tests were essential for ensuring the reliability and precision of our system in real-world operational scenarios. By conducting these assessments, we verified that our system could perform accurately and dependably when faced with actual usage conditions.

The metrics of PPR (Pulses Per Revolution) and CPR (Counts Per Revolution) played a central role in evaluating our system's performance. PPR quantifies the number of electrical pulses generated by the motor per revolution, providing crucial insights into its rotational behavior. Meanwhile, CPR represents the counts registered by the system for each revolution, offering a precise measure of its positional accuracy.

1. IMPLEMENTATION AND TESTING

1.4.1.2 Measure motor with L298N motor driver specifications



This section is dedicated to measurement, where we started by varying the PWM value, which allowed us to obtain numerous values for the gear and motor. Therefore, we averaged some of the speeds and recorded them in the table below. Simultaneously, we measured the voltage at the terminals of the two pins on the power board.

```
1 import sys
2 import time
3 import RPi.GPIO as GPIO
4
5 GPIO.setmode(GPIO.BOARD)
6
7 StepPinForward = 18 # M1 #GPIO24
8 StepPinBackward = 16 # M2 #GPIO23
9 PWMPin = 35 # ENA # GPIO19
10 Enc_A = 36 # GPIO16
11 test_duration = 10
12 ppr = 24 # d apres cdc
13 reduction_ratio = 226
14
15 counter = 0
16
17 GPIO.setup(StepPinForward, GPIO.OUT)
18 GPIO.setup(StepPinBackward, GPIO.OUT)
19 GPIO.setup(PWMPin, GPIO.OUT)
20 motor_pwm = GPIO.PWM(PWMPin, 5000)
21 motor_pwm.start(0)
22
23 1 usage KayoumDjedidi
24 def init():
25     print("Rotary Encoder Test Program")
26     GPIO.setup(Enc_A, GPIO.IN, pull_up_down=GPIO.PUD_UP)
27     GPIO.add_event_detect(Enc_A, GPIO.RISING, callback=rotation_decode)
28     return
29
30 1 usage KayoumDjedidi
31 def rotation_decode(channel):
32     global counter
33     counter += 1
34
35 1 usage KayoumDjedidi
36 def calculate_speeds():
37     motor_speed = ((counter / ppr) / test_duration) * 60
38     gear_speed = motor_speed / reduction_ratio
```


1. IMPLEMENTATION AND TESTING

```
print("Motor RPM:", motor_speed)
print("Gear RPM:", gear_speed)
37
38
39 try:
40     init()
41     while True:
42         GPIO.output(StepPinForward, GPIO.LOW)
43         GPIO.output(StepPinBackward, GPIO.HIGH)
44         motor_pwm.ChangeDutyCycle(100)
45         time.sleep(test_duration)
46         print("Counter:", counter)
47         calculate_speeds()
48         counter = 0
49
50 except KeyboardInterrupt:
51     GPIO.cleanup()
52
```

This setup allows for accurate motor control and real-time speed monitoring, crucial for applications requiring precise motor regulation.

```
"""
input 11.7v
counter 17062 tours equivalent a: 4265.5 rpm
RPM du reducteur:18.873893805309734 """
```

We collected measurements of PWM, voltage, motor RPM, and gear RPM. These measurements were organized into a table to analyze the relationship between PWM values and motor performance.

PWM	Voltage(V)	Motor RPM	Gear RPM
100	9.61	3372	14.4
95	8.9	2346	10.46
90	8.6	2262	10.08
85	8.4	2180	9.67
80	8.1	2091	9.21
75	5.9	2015	8.9
70	5.7	1877	8.25
65	5.4	1741	7.75
60	5	1626	7.19
55	4.6	1450	6.55
50	4.3	1310	5.7
45	3.8	1188	4.91
40	3.2	930	4.12
35	2.8	750	3.31
30	2.2	548	2.54
25	1.7	410	1.77
20	1.3	281	1.31
15	0	0	0
10	0	0	0
5	0	0	0
0	0	0	0

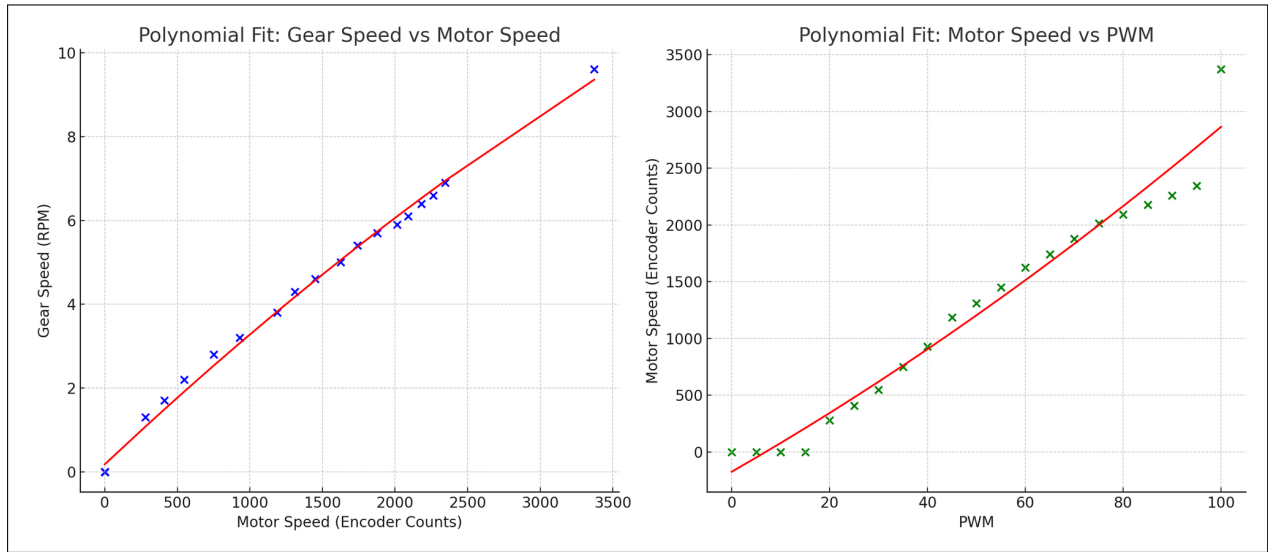
1. IMPLEMENTATION AND TESTING

To visualize the relationship between PWM values, voltage, motor RPM, and gear RPM, we can plot the data using Python. Below is the code to generate the plots :

```
1  import csv
2  import matplotlib.pyplot as plt
3
4  # Read data from CSV file
5  data = []
6  with open('table.csv', 'r') as file:
7      reader = csv.reader(file)
8      for row in reader:
9          data.append(row)
10
11
12  # Plot first column as x and second column as y
13  x_data = [float(row[0]) for row in data]
14  y_data = [float(row[1]) for row in data]
15  plt.figure()
16  plt.subplot(*args: 2,2,1)
17  plt.plot(*args: x_data, y_data, marker='o', linestyle='-')
18  plt.xlabel('PWM')
19  plt.ylabel('Motor Voltage')
20  plt.title('Motor Voltage en fonction du PWM')
21  plt.grid(True)
22
23
24  # Plot first column as x and third column as y
25  x_data = [float(row[0]) for row in data]
26  y_data = [float(row[2]) for row in data]
27  plt.subplot(*args: 2,2,2)
28  plt.plot(*args: x_data, y_data, marker='o', linestyle='-')
29  plt.xlabel('PWM')
30  plt.ylabel('Motor RPM')
31  plt.title('Motor RPM en fonction du PWM')
32  plt.grid(True)
33
34
35  # Plot first column as x and fourth column as y
36  x_data = [float(row[0]) for row in data]
37  y_data = [float(row[3]) for row in data]
38  plt.subplot(*args: 2,2,3)
39  plt.plot(*args: x_data, y_data, marker='o', linestyle='-')
40  plt.xlabel('PWM')
41  plt.ylabel('Gear RPM')
42  plt.title('Gear RPM en fonction du PWM')
43  plt.grid(True)
44  plt.show()
45
```

The code will generate two plots:

1. IMPLEMENTATION AND TESTING



- Figure 1: Motor speed as a function of PWM

The first graph shows a positive linear relationship between motor speed and PWM. This means that when the PWM value increases, the motor speed also increases. The equation of the regression line is as follows:

$$\text{Motor speed (encoder counts)} = 28,511 \text{ PWM} - 204,043$$

- Figure 2: Gear Speed as a Function of PWM

The second graph shows a positive linear relationship between the gear speed and the PWM. This means that when the PWM value increases, the speed of the reducer also increases. The equation of the regression line is as follows:

$$\text{Reducer speed (tr/min)} = 0.1264 \text{ PWM} - 0.9186$$

These equations allow us to estimate the motor speed and gear speed for a given PWM input. To use these equations to calculate the required PWM to achieve a desired gear speed, we can rearrange the second equation to solve for PWM:

$$\text{PWM} = \frac{\text{Gear Speed (RPM)} + 0.9186}{0.1264}$$

=> Due to measurement errors the values are not exact, so we continued testing to obtain improved and adaptable values that align with our objectives.

Ultimately, we collected the data to develop a suitable model of the motor, utilizing an object-oriented approach to ensure modularity and facilitate the interpretation of results.

1. IMPLEMENTATION AND TESTING

The code defines a 'MotorModel' class for a motor control system. The class includes methods to calculate PWM (Pulse Width Modulation) from gear speed and plot the relationships between gear speed, motor speed, and PWM. It initializes model parameters, estimates motor speed from gear speed, constrains PWM values between 0 and 100, and generates plots for visualization.

```
import matplotlib.pyplot as plt

3 usages new *
class MotorModel:
    new *
    def __init__(self, gear_motor_slope=0.00320, gear_motor_intercept=0.4000, motor_pwm_slope=36.394, motor_pwm_intercept=-300.667):
        self.gear_motor_slope = gear_motor_slope
        self.gear_motor_intercept = gear_motor_intercept
        self.motor_pwm_slope = motor_pwm_slope
        self.motor_pwm_intercept = motor_pwm_intercept

1 usage new *
    def calculate_pwm_from_gear_speed(self, desired_gear_speed):
        estimated_motor_speed = (desired_gear_speed - self.gear_motor_intercept) / self.gear_motor_slope
        print(f"Estimated Motor Speed: {estimated_motor_speed}")
        estimated_pwm = (estimated_motor_speed - self.motor_pwm_intercept) / self.motor_pwm_slope
        print(f"Raw PWM: {estimated_pwm}")
        estimated_pwm = max(0, min(estimated_pwm, 100))
        return estimated_motor_speed, estimated_pwm

2 usages new *
    def plot_relationships(self, gear_speeds):
        motor_speeds = []
        pwms = []
        for gear_speed in gear_speeds:
            motor_speed, pwm = self.calculate_pwm_from_gear_speed(gear_speed)
            motor_speeds.append(motor_speed)
            pwms.append(pwm)

        plt.figure(figsize=(10, 5))
        plt.subplot(*args: 1, 2, 1)
        plt.plot(*args: gear_speeds, motor_speeds, '-o', label='Motor Speed')
        plt.title('Motor Speed vs. Gear Speed')
        plt.xlabel('Gear Speed (RPM)')
        plt.ylabel('Motor Speed (Encoder Counts)')
        plt.grid(True)
        plt.legend()

        plt.subplot(*args: 1, 2, 2)
        plt.plot(*args: gear_speeds, pwms, '-o', color='red', label='PWM')
        plt.title('PWM vs. Gear Speed')
        plt.xlabel('Gear Speed (RPM)')
        plt.ylabel('PWM (%)')
        plt.grid(True)
        plt.legend()

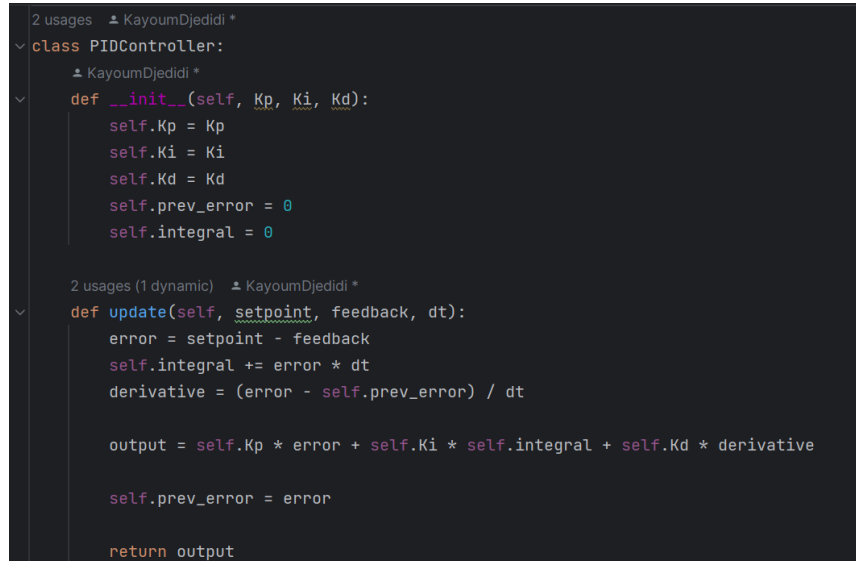
        plt.tight_layout()
        plt.show()

if __name__ == "__main__":
    motor_model = MotorModel()
    gear_speeds = range(5, 16)
    motor_model.plot_relationships(gear_speeds)
```

1.4.2 Control Settings

then we need to create a PID model for our corrector, which is a basic implementation of a PID controller where the values can be edited by the user, we created a basic class for this which we will implement later in the full system.

here's the code for the PID controller :



```
2 usages  KayoumDjedidi *
class PIDController:
    KayoumDjedidi *
    def __init__(self, Kp, Ki, Kd):
        self.Kp = Kp
        self.Ki = Ki
        self.Kd = Kd
        self.prev_error = 0
        self.integral = 0

2 usages (1 dynamic)  KayoumDjedidi *
def update(self, setpoint, feedback, dt):
    error = setpoint - feedback
    self.integral += error * dt
    derivative = (error - self.prev_error) / dt

    output = self.Kp * error + self.Ki * self.integral + self.Kd * derivative

    self.prev_error = error

    return output
```

the motor Model and PID models were then used as starting ground to create our full system, which includes a modular implementation of this motor model.

the full system must also include the option to drive the motor in an open loop or a closed loop. for the closed loop, we implemented the PID controller, where the parameters of the PID are configurable by the user directly both models are used interchangeably and are tied together heres the code for the full system:

1. IMPLEMENTATION AND TESTING

```
import matplotlib.pyplot as plt
import numpy as np
from Controller_PID import PIDController

7 usages new *
class MotorSystem:
    new *
    def __init__(self, Kp, Ki, Kd, gear_motor_slope=0.00320, gear_motor_intercept=0.4000, motor_pwm_slope=36.394,
        motor_pwm_intercept=-300.667):
        self.pid = PIDController(Kp, Ki, Kd)
        self.gear_motor_slope = gear_motor_slope
        self.gear_motor_intercept = gear_motor_intercept
        self.motor_pwm_slope = motor_pwm_slope
        self.motor_pwm_intercept = motor_pwm_intercept

1 usage new *
    def calculate_pwm_from_gear_speed(self, desired_gear_speed):
        estimated_motor_speed = (desired_gear_speed - self.gear_motor_intercept) / self.gear_motor_slope
        print(f"Estimated Motor Speed: {estimated_motor_speed}")
        estimated_pwm = (estimated_motor_speed - self.motor_pwm_intercept) / self.motor_pwm_slope
        print(f"Raw PWM: {estimated_pwm}")
        estimated_pwm = max(0, min(estimated_pwm, 100))
        return estimated_motor_speed, estimated_pwm

2 usages (1 dynamic) new *
    def control_motor(self, setpoint, dt, time_end):
        times = np.arange(start=0, *args: time_end, dt)
        motor_speeds = []
        gear_speeds = []
        pwms = []

        for t in times:
            motor_speed, pwm = self.calculate_pwm_from_gear_speed(setpoint)
            motor_speeds.append(motor_speed)
            control_signal = self.pid.update(setpoint, motor_speed, dt)
            control_pwm = max(0, min(control_signal, 100))
            pwms.append(control_pwm)
            _, gear_speed = self.calculate_speeds(dt)
            gear_speeds.append(gear_speed)

        return times, motor_speeds, gear_speeds, pwms
```

1. IMPLEMENTATION AND TESTING

```
2 usages new *
def plot_motor_control(self, setpoint, dt, time_end):
    times, motor_speeds, pwms = self.control_motor(setpoint, dt, time_end)
    plt.figure(figsize=(10, 5))
    plt.subplot(*args 1, 2, 1)
    plt.plot(*args times, motor_speeds, '-o', label='Motor Speed')
    plt.title('Motor Speed vs. Time')
    plt.xlabel('Time (s)')
    plt.ylabel('Motor Speed (Encoder Counts)')
    plt.grid(True)
    plt.legend()

    plt.subplot(*args 1, 2, 2)
    plt.plot(*args times, pwms, '-o', color='red', label='PWM')
    plt.title('PWM vs. Time')
    plt.xlabel('Time (s)')
    plt.ylabel('PWM (%)')
    plt.grid(True)
    plt.legend()

    plt.tight_layout()
    plt.show()

if __name__ == "__main__":
    motor_system = MotorSystem(Kp=1.0, Ki=0.1, Kd=0.05)
    motor_system.plot_motor_control(setpoint=5.0, dt=0.01, time_end=2.0)
```

Now to implement this full system on the Raspberry Pi, our control board of choice, we prepared the following code: this code will import our full system and use it to drive the motor according to the user's input, and then plot the results.

The code defines a 'MotorSystem' class that includes a PID controller for motor control. It estimates motor speed and PWM from gear speed and controls the motor using the PID controller to maintain a setpoint. The class methods calculate and constrain PWM values, simulate the motor control system over time, and generate plots to visualize motor speed and PWM versus time.

1. IMPLEMENTATION AND TESTING

```
import sys
import time
import RPi.GPIO as GPIO
import matplotlib.pyplot as plt
import numpy as np
from full_system import MotorSystem

1 usage: new *
class RaspberryPiMotorController:
    new *
    def __init__(self, step_pin_forward, step_pin_backward, pwm_pin, encoder_pin, ppr, reduction_ratio):
        self.step_pin_forward = step_pin_forward
        self.step_pin_backward = step_pin_backward
        self.pwm_pin = pwm_pin
        self.encoder_pin = encoder_pin
        self.ppr = ppr
        self.reduction_ratio = reduction_ratio
        self.counter = 0
        self.motor_pwm = None
        self.init_gpio()

1 usage: new *
    def init_gpio(self):
        GPIO.setmode(GPIO.BOARD)
        GPIO.setup(self.step_pin_forward, GPIO.OUT)
        GPIO.setup(self.step_pin_backward, GPIO.OUT)
        GPIO.setup(self.pwm_pin, GPIO.OUT)
        self.motor_pwm = GPIO.PWM(self.pwm_pin, 5000)
        self.motor_pwm.start(0)
        GPIO.setup(self.encoder_pin, GPIO.IN, pull_up_down=GPIO.PUD_UP)
        GPIO.add_event_detect(self.encoder_pin, GPIO.RISING, callback=self.rotation_decode)

1 usage: new *
    def rotation_decode(self, channel):
        self.counter += 1

2 usages: new *
    def calculate_speeds(self, duration):
        motor_speed = ((self.counter / self.ppr) / duration) * 60 # Motor speed in RPM
        gear_speed = motor_speed / self.reduction_ratio # Gear speed in RPM
        return motor_speed, gear_speed

1 usage: new *
    def get_speeds(self, motor_system, setpoint, dt, time_end):
        times = np.arange(start=0, stop=time_end, dt)
        motor_speeds = []
        gear_speeds = []
        pwms = []
```


1. IMPLEMENTATION AND TESTING

```
45         for t in times:
46             time.sleep(dt)
47             motor_speed, _ = self.calculate_speeds(dt)
48             motor_speeds.append(motor_speed)
49
50             control_signal = motor_system.pid.update(setpoint, motor_speed, dt)
51             control_pwm = max(0, min(control_signal, 100))
52             pwms.append(control_pwm)
53
54             self.motor_pwm.ChangeDutyCycle(control_pwm)
55
56             _, gear_speed = self.calculate_speeds(dt)
57             gear_speeds.append(gear_speed)
58
59         return times, motor_speeds, gear_speeds, pwms
60
61     1 usage new *
62     def plot_speeds(self, times, motor_speeds, gear_speeds, pwms):
63         fig, (ax1, ax2, ax3) = plt.subplots( rows=3, ncols=1, figsize=(10, 12))
64
65         ax1.plot(times, motor_speeds, label='Motor Speed', color='blue')
66         ax1.set_xlabel('Time (s)')
67         ax1.set_ylabel('Motor Speed (RPM)')
68         ax1.set_title('Motor Speed over Time')
69         ax1.grid(True)
70
71         ax2.plot(times, gear_speeds, label='Gear Speed', color='green')
72         ax2.set_xlabel('Time (s)')
73         ax2.set_ylabel('Gear Speed (RPM)')
74         ax2.set_title('Gear Speed over Time')
75         ax2.grid(True)
76
77         ax3.plot(times, pwms, label='PWM', color='red')
78         ax3.set_xlabel('Time (s)')
79         ax3.set_ylabel('PWM (%)')
80         ax3.set_title('PWM over Time')
81         ax3.grid(True)
82
83         plt.tight_layout()
84         plt.show()
85
86     7 usages (5 dynamic) new *
87     def cleanup(self):
88         GPIO.cleanup()
89
90     1 usage new *
91     def main():
92         try:
93             controller = RaspberryPiMotorController(
94                 step_pin_forward=18,
95                 step_pin_backward=16,
96                 pwm_pin=35,
97                 encoder_pin=36,
98                 ppr=24,
99                 reduction_ratio=226
```

```
100         )
101         motor_system = MotorSystem(Kp=1.0, Ki=0.1, Kd=0.05)
102         setpoint = 5.0 # Desired gear speed in RPM
103         dt = 0.01 # Time step for control loop
104         time_end = 10 # Total time for simulation in seconds
105
106         times, motor_speeds, gear_speeds, pwms = controller.get_speeds(motor_system, setpoint, dt, time_end)
107         controller.plot_speeds(times, motor_speeds, gear_speeds, pwms)
108
109     except KeyboardInterrupt:
110         controller.cleanup()
111     finally:
112         controller.cleanup()
113
114     if __name__ == "__main__":
115         main()
```

To keep our project more organized, we have created a top-level file to manage the structure

1. IMPLEMENTATION AND TESTING

and order of executing the different parts of the project and to facilitate debugging, here is the top-level code:

```
from motor_model import MotorModel
from full_system import MotorSystem
from Integration_in_RaspberryPi import main as raspberry_pi_main

usage new *
def main():
    # Plotting motor model relationships
    motor_model = MotorModel()
    gear_speeds = range(5, 16)
    motor_model.plot_relationships(gear_speeds)

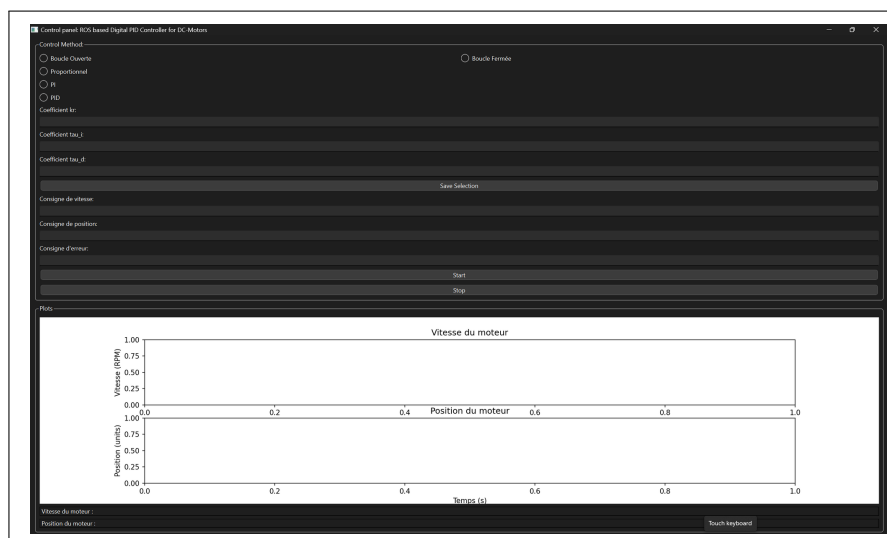
    # Controlling the motor using PID
    motor_system = MotorSystem(Kp=1.0, Ki=0.1, Kd=0.05)
    motor_system.plot_motor_control(setpoint=5.0, dt=0.01, time_end=2.0)

    # Running the integration on Raspberry Pi
    raspberry_pi_main()

if __name__ == "__main__":
    main()
```

1.4.3 User Interface Design

Now what remains is our interface to control the whole project. since its code is very lengthy, it is not practical to include it in this report. here's the finalized interface and its different interactions.



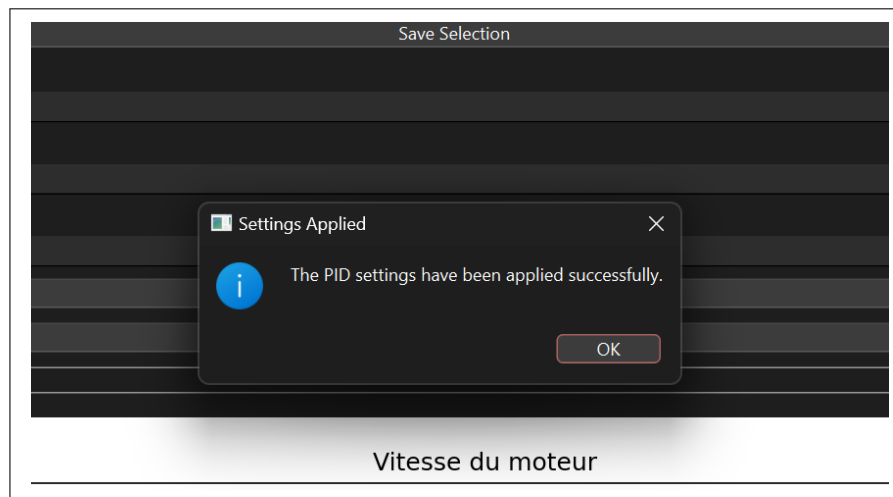
The control settings empower users to configure the control methods and parameters according to their specific requirements. It encompasses the following elements:

1. IMPLEMENTATION AND TESTING

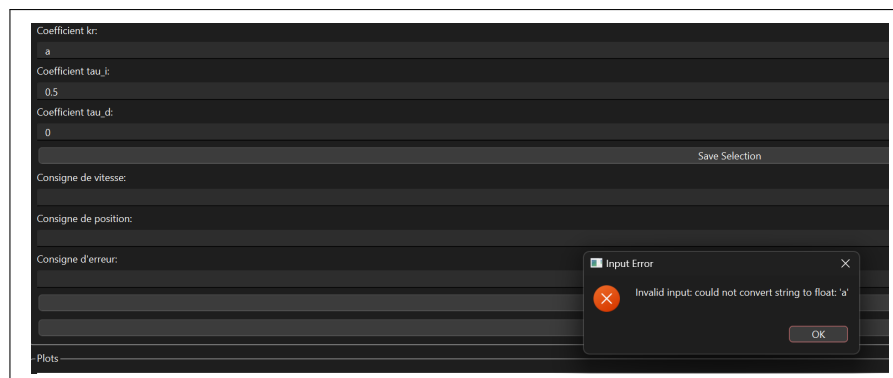
- Options to select control methods including open-loop and closed-loop control, for which the user can pick from Proportional (P), Proportional-Integral (PI), and Proportional-Integral-Derivative (PID).
- Input fields for choosing the controller type (P, PI, PID).
- Dropdown menus or sliders for adjusting PID parameters (P, I, D) for both speed and position control.
- Input fields for setting speed and position setpoints.

The user gets to pick their control method of choice, then must input the parameters of choice correctly, in case settings are missing or out of bounds, a warning will be shown, improving the user experience and providing security features, assuring the parameters are not beyond the physical system's capabilities.

Here's an example of handling errors, warnings, and communicating with the user:



Notice the confirmation message indicating that the settings have been applied successfully. The interface provides clear feedback to the user, enhancing user experience and ensuring confidence in the system's operation



The warning alerts the user about the incorrect parameters entered, preventing potential errors or system malfunctions.

1.4.4 ROS Package and Node Development

When it comes to the ROS integration, with the time we had available for this project, The `ros_integration.py` script integrates a motor control system with ROS, providing basic control and real-time monitoring features, note that it doesn't currently communicate with the control interface.

Key Features:

- ROS Node Initialization:

Initializes a ROS node named motor-control node.

- Motor System Integration:

Uses the MotorSystem class with a PID controller to manage motor speed and position. Dynamic Reconfiguration:

Supports real-time adjustment of PID parameters (K_p , K_i , K_d) via ROS dynamic reconfiguring.

- ROS Topics:

Subscribes to `/motor-speed` and `/motor-position` for control commands. Publishes to `/motor-speed-feedback` and `/motor-position-feedback` for real-time feedback. ROS Service:

Provides a `/reset-pid` service to reset the PID controller. Real-time Publishing:

Simulates real-time data feedback for motor speed and position. How It Was Made: Initialization:

The ROS node is initialized, and the MotorSystem class is instantiated with default PID parameters. ROS Communication:

Sets up subscribers and publishers for motor speed and position control and feedback. Implements a dynamic reconfigure server for real-time PID parameter adjustments.

- Service and Callbacks:

Defines a service to reset the PID controller. Implements callbacks to handle incoming messages and publish feedback. Running the Node:

1. IMPLEMENTATION AND TESTING

Uses `rospy.spin()` to keep the node running and responsive.

- Package Setup: Dependencies:

Includes necessary dependencies in `package.xml` and `CMakeLists.txt`.

- Building and Running:

Builds the package using `catkin-make`. Runs the node with `roslaunch`.

- Tools:

Uses `rqt-reconfigure` for dynamic reconfiguration and `rqt-plot` for real-time visualization. This setup provides a starting point for a flexible and responsive motor control system integrated with ROS.

```
1  #!/usr/bin/env python
2
3  import rospy
4  from std_msgs.msg import Float32
5  from dynamic_reconfigure.server import Server
6  from motor_control.cfg import PIDConfig
7  from Controller_PID import PIDController
8  from motor_model import MotorSystem
9  import numpy as np
10
11
12  class MotorControlNode:
13      new *
14      def __init__(self):
15          rospy.init_node('motor_control_node', anonymous=True)
16
17          # Initialize MotorSystem with default PID parameters
18          self.motor_system = MotorSystem(Kp=1.0, Ki=0.1, Kd=0.05)
19
20          # ROS Subscribers
21          self.speed_subscriber = rospy.Subscriber('/motor_speed', Float32, self.speed_callback)
22          self.position_subscriber = rospy.Subscriber('/motor_position', Float32, self.position_callback)
23
24          # ROS Publishers
25          self.speed_publisher = rospy.Publisher('/motor_speed_feedback', Float32, queue_size=10)
26          self.position_publisher = rospy.Publisher('/motor_position_feedback', Float32, queue_size=10)
27
28          # ROS Service to reset the PID controller
29          self.reset_service = rospy.Service('/reset_pid', Empty, self.reset_pid)
30
31          # Dynamic reconfigure server
32          self.srv = Server(PIDConfig, self.reconfigure_callback)
33
34      new *
35      def speed_callback(self, msg):
36          setpoint = msg.data
37          rospy.loginfo(f"Setting motor speed to: {setpoint}")
38          times, motor_speeds, gear_speeds, pwms = self.motor_system.control_motor(setpoint, dt=0.01, time_end=2.0)
39          for speed in motor_speeds:
40              self.speed_publisher.publish(speed)
41          rospy.sleep(0.01) # simulate real-time publishing
42
43      new *
44      def position_callback(self, msg):
45          setpoint = msg.data
46          rospy.loginfo(f"Setting motor position to: {setpoint}")
47          times, motor_speeds, gear_speeds, pwms = self.motor_system.control_motor(setpoint, dt=0.01, time_end=2.0)
48          for position in gear_speeds:
49              self.position_publisher.publish(position)
50          rospy.sleep(0.01) # simulate real-time publishing
```

1. IMPLEMENTATION AND TESTING

```
48
    1 usage new *
49     def reconfigure_callback(self, config, level):
50         rospy.loginfo(f"Reconfigure request: {config}")
51         self.motor_system.pid.Kp = config['Kp']
52         self.motor_system.pid.Ki = config['Ki']
53         self.motor_system.pid.Kd = config['Kd']
54         return config
55
    1 usage new *
56     def reset_pid(self, req):
57         rospy.loginfo("Resetting PID controller")
58         self.motor_system.pid.reset()
59         return EmptyResponse()
60
    1 usage new *
61     def run(self):
62         rospy.spin()
63
64
65 ▶ if __name__ == '__main__':
66     try:
67         node = MotorControlNode()
68         node.run()
69     except rospy.ROSInterruptException:
70         pass
71
```

In retrospect, the ROS integration aspect was addressed after the course concluded, and unfortunately, we did not have the opportunity to test it in the laboratory setting.

1.4.5 Conclusion

In conclusion, the systematic approach detailed in this chapter has resulted in a reliable and user-friendly motor control system. The use of ROS has allowed for modular development, easy integration of various components, and real-time control and monitoring. This robust foundation sets the stage for further enhancements and customizations to meet specific user needs and operational scenarios.

General Conclusions

In this project, we have outlined the comprehensive implementation and testing process for our motor control system. We began by preparing a template and establishing motor specifications through detailed measurements, ensuring the accuracy and reliability of our system. By analyzing the relationship between PWM values and motor performance, we derived key equations that enable precise control over motor speed and gear speed.

The Control Settings section was designed to provide users with the flexibility to adjust setpoints, select control strategies, and fine-tune PID parameters dynamically. This ensures that the system can be tailored to meet various operational requirements efficiently.

The User Interface (UI) was meticulously crafted to facilitate intuitive interaction and effective control. It includes sections for data acquisition, control settings, system disturbances, configuration, and essential utilities. Each section was implemented with user-friendliness and functionality in mind, ensuring that users can monitor and control the motor system with ease.

Throughout the implementation, we emphasized real-time data acquisition and visualization, robust control settings, and thorough system testing to guarantee seamless integration and performance. The final UI design encapsulates these elements, providing a cohesive and efficient platform for motor control.