Case Study Embedded Control Solutions Motor Speed Control

Felipe Rojas, Oben Sustam, Ibrahim Kemal Yasar, Basem Aboudeshish, Mohamed Mady
Technische Hochschule Deggendorf
Artificial Intelligence for Smart Sensors & Actuators

***Abstract*—This project aims to create and execute a mechanism for regulating the velocity of a DC motor utilizing a ESP32 microcontroller. The initiative will commence with an overview of the subject of velocity control for DC motor. The requirements, risks and limitations of the project will be recognized and examined. The design of the system will encompass a design diagram, electronic hardware selection, and circuit diagram. The development stage will consist of programming and coding the microcontroller to regulate the motor's speed. The system will be examined and demonstrated to ensure proper performance with the testing and demonstration stages. Ultimately, the project will be documented and a future outlook for advancements will be outlined. Additional materials such as project timeline and full code will be provided in the appendix.**

I. Introduction

The project task assigned was to design and develop a control system for a DC motor using an ESP32 microcontroller, with the goal of implementing a control system that can maintain a constant speed of the motor regardless of the load. As a project group, we have successfully completed this task and would like to present our findings in this final report.

The first step in the project was to define and detail the project requirements. We conducted an analysis to specify the objectives that needed to be achieved in order for the project to be successful. We took into consideration the project description provided and further expanded and detailed it to ensure that all the goals of the project were clearly outlined.

Once the requirements were defined, we proceeded to the design phase, where we planned all the hardware and software components that would be used in the project. This included selecting the appropriate motor and other electronic components that were required to build the control system.

In the development phase, we implemented the project by building the hardware and programming the software. We ensured that a detailed description of the work done during this phase was documented, so that the project could be replicated.

Testing was an essential part of the project, as it allowed us to review the work done so far and check if the objectives had been achieved. We were able to successfully test the control system and ensure that it met all the requirements outlined in the project.

Finally, we demonstrated the result of the project to the whole course. The demonstration presented the functioning of the control system and also highlighted the unique features and solutions that we had found during the project.

II. Theoretical Work and Implementation Steps

The process of examining and evaluating existing research, known as the theoretical work, will be utilized in this study to illustrate the method of utilizing an ESP32 microcontroller and the L298N Motor Driver to regulate the rotation and velocity of a direct current motor. To begin, the inner workings and functionality of the L298N motor driver will be examined and explained. Following this, instructions and guidance will be provided on how to operate the ESP32 in conjunction with the Arduino IDE and the L298N motor driver to manage and manipulate the speed of the DC motor.

In theoretical work stage, we present a theoretical examination of utilizing an ESP32 microcontroller in combination with the L298N Motor Driver to control the speed of a DC motor. The ESP32 DOIT DEVKIT V1 Board is used as the development board to evaluate the ESP-WROOM-32 module and is based on the ESP32 microcontroller which supports features such as WiFi, Bluetooth, Ethernet, and low power consumption. The DC motor converts electrical energy into mechanical energy by taking electrical power through direct current and converting it into mechanical rotation. The L298N motor driver, a dual H-Bridge motor driver, is used to control the speed of the DC motor, and can drive DC motors that have voltages between 5 and 35V and peak current up to 2A. The power source used in this study is 4x 1.5 AA batteries or Bench power supply. Jumper wires are also required to connect the various components in the circuit.

[1] We will be using the L298N motor driver to control a DC motor. The L298N is a suitable choice for motors that require 6V or 12V to operate, and can handle up to 3A at 35V. The motor driver has a terminal block for a motor (OUT1 and OUT2 for motor 1) and a three terminal block for power supply (+12V, GND, and +5V). The +12V terminal is where the power supply, in this case 4 AA 1.5V batteries or a bench power supply, should be connected. The GND terminal is connected to the power

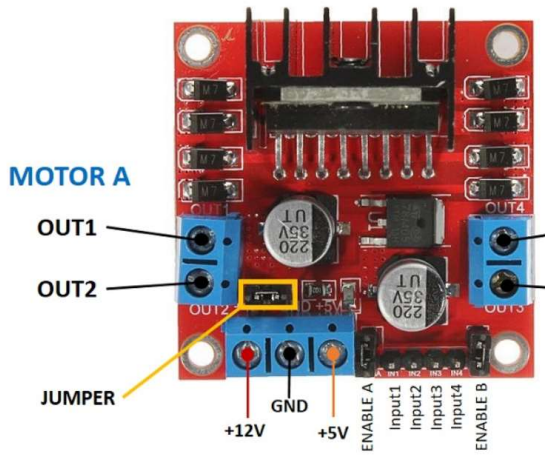

Fig. 1. L298N motor driver https://www.electrorules.com/esp32-with-dc-motor-and-l298n-motor-driver-control-speed-and-direction/ (Nair,2022)

supply ground. The +5V terminal is used to power the L298N chip, but if the jumper is in place, the chip will be powered by the motor's power supply and the +5V terminal is not needed. It's important to note that if you supply more than 12V, you should remove the jumper and supply 5V to the +5V terminal. At the bottom right of the L298N, there are one enable terminal that can be used to control the speed of one motor. The enable pins have jumper caps by default, but these should be removed to control the speed of the motors.

In summary, the L298N motor driver is a suitable choice for controlling a DC motor with a power supply between 6V and 12V. The power supply should be connected to the +12V terminal, the ground to the GND terminal and if needed +5V to the +5V terminal. The enable pins can be used to control the speed of the motor by removing the jumper caps.

SIGNAL ON THE ENABLE PIN	MOTOR STATE
HIGH	Motor enabled
LOW	Motor not enabled
PWM	Motor enabled: speed proportional to duty cycle

Fig. 2. Enable pins https://www.electrorules.com/esp32-with-dc-motor-and-l298n-motor-driver-control-speed-and-direction/ (Nair,2022)

[1] We will be using the enable pins of the L298N motor driver to control the DC motor. The enable pins function as an ON and OFF switch for the motors, allowing for precise control over the motor's speed and operation. By sending a HIGH signal to the enable 1 pin, motor A will be ready to be controlled and will run at maximum speed. Conversely, sending a LOW signal to the enable 1 pin will turn off the motor.

Additionally, by sending a PWM signal to the enable pins, the speed of the motor can be controlled in a more fine-grained manner. The motor speed is proportional to the duty cycle of the PWM signal, with a higher duty cycle resulting in a faster motor speed. However, it's important to note that for small duty cycles, the motors might not spin, and instead make a continuous buzz sound. This is something that should be considered when controlling the motor speed using PWM signals.

In summary, the enable pins of the L298N motor driver can be used to control the speed of the DC motor by sending a HIGH, LOW or PWM signals. However, when using PWM signals, it's important to consider that small duty cycles might not spin the motor and make a continuous buzz sound.

The first step is to upload the code to the ESP32 and make sure the correct board and COM port are selected.

Next, the motor pins are defined and set as outputs in the setup() function using the pinMode() function.

To control the speed of the DC motor, a PWM signal is applied to the enable pin of the L298N motor driver. To use PWM with the ESP32, the PWM signal properties must be set first using the ledcSetup() function, which accepts the PWM channel, frequency, and resolution as arguments. The ledcAttachPin() function is then used to choose the GPIO where the signal will be received, in this case the enable1Pin GPIO, and the channel that is generating the signal, the pwmChannel.

To change the DC motor speed, the ledcWrite() function is used to change the duty cycle of the PWM signal, which is passed as an argument, along with the PWM channel. In the example, a while loop is used to increase the duty cycle by 5 in each iteration, resulting in an increase in motor speed. Additionally, the digitalWrite() function is used to set the motor1Pin1 and motor1Pin2 as HIGH and LOW respectively to move the motor forward.

In summary, the ESP32 microcontroller is used to control the speed of a DC motor by applying a PWM signal to the enable pin of the L298N motor driver, the properties of which are set using the ledcSetup() function, and the signal is received on the enable1Pin GPIO, the duty cycle of the PWM signal is changed using ledcWrite() function, which results in the change of the motor speed, and the digitalWrite() function is used to set the direction of the motor.

The PID controller is a widely used control algorithm that is composed of three separate control terms: the proportional, integral, and derivative control. The proportional control term provides a correction based on the current error, the integral control term provides a correction based on the accumulated error over time, and the derivative control term provides a correction based on the rate of change of the error.

The proportional gain (Kp) determines the ratio of output to the error, the integral gain (Ki) determines how much the past errors affect the current control action and the derivative gain (Kd) determines how much the rate of change of the error affects the control action.

Tuning the PID controller is a process of adjusting the three gains, Kp, Ki, and Kd, to optimize the performance of the control system. There are several methods to tune

the controller such as Ziegler-Nichols method, Tyreus Luyben method and Cohen-Coon method.

In our project, we will be using the ESP32 microcontroller to implement the PID controller for controlling the speed of a DC motor. The ESP32 will be used to read the speed feedback from the motor, calculate the error, and generate the control output to drive the motor. The built-in connectivity options of the ESP32 will also be used to remotely monitor and control the system.

The performance of the PID controller will be evaluated by measuring the response time, steady-state error, and overshoot of the system. The PID controller will be tuned using Ziegler-Nichols tuning method to achieve the optimal performance of the control system.

We will also explain the use of the ESP32 microcontroller as the platform for implementing the PID controller, and how it will be used to control the speed of the DC motor.

III. Requirements

The main goal of this project was to design and develop a control system for a DC motor using an ESP32 microcontroller, with the aim of implementing a control system that can maintain a constant speed of the motor regardless of the load. In order to achieve this goal, the following requirements were identified and fulfilled during the course of the project:

The system must be able to choose a desired speed and receive encoder feedback, and send this data to the motor driver through the ESP32 microcontroller. The code for the system must be able to be uploaded to the ESP32 after the software for the motor pins, PWM properties, and DC motor speed control have been completed.

The system must use a DC motor with an integrated encoder that converts electrical energy into mechanical rotation. The system must be able to receive speed feedback from the motor through the encoder.

The system must use a motor driver to control the speed of the DC motor, and the controller type and power source must be selected accordingly. The DC motor must be compatible with a 6V power supply and be capable of providing a maximum speed of 6200 rpm. The motor driver must be rated to handle a maximum voltage of 35V and be able to drive the desired DC motor.

The motor driver must be able to provide a maximum current of 2A per channel for the DC motor.

The power supply must be capable of providing a maximum power of 18W and a maximum voltage of 12V to all the microcontroller and other components of the system, including the 6V DC motor.

The controller must be able to achieve a settling time of less than 10 seconds, meaning that the output of the controlled system should reach and remain within a specified error tolerance of its final value for at least less than 10 seconds.

Selection of appropriate microcontroller: The ESP32 microcontroller was selected as it is capable of handling the high-speed pulse-width modulation (PWM) signals required to control the speed of the DC motor.

Motor speed measurement: A speed measurement system, such as an encoder, was implemented to measure the speed of the motor. This was used to compare the desired speed to the actual speed and make adjustments as needed.

Control algorithm: A control algorithm was implemented to control the speed of the motor. The algorithm was designed to maintain a constant speed regardless of the load on the motor.

Power supply: A power supply was implemented to provide the necessary voltage and current to the motor and microcontroller.

Safety features: Safety features were implemented to protect the components from damage in case of unexpected events such as overcurrent or overheating.

Testing and demonstration: The control system was thoroughly tested and demonstrated to ensure that it met all the requirements and performed as expected.

Overall, the project was successful in meeting all the requirements and fulfilling the main goal of designing and developing a control system for a DC motor using an ESP32 microcontroller that can maintain a constant speed regardless of the load.

IV. Risks and Limitations

During the course of this project, there were several risks that we had to consider and take measures to mitigate. The following are the major risks that we identified and the steps we took to address them:

Damaging electronic component by burning their pins due to wrong wiring or pin configuration: One of the major risks that we identified was the possibility of damaging electronic components by burning their pins due to incorrect wiring or pin configuration. To mitigate this risk, we made sure to carefully follow the wiring diagrams and pin configurations provided in the datasheets of the components. We also double-checked all the connections before applying power to the circuit.

Burning motor driver by mistake through swapping the connections from the electrical power source: Another risk that we identified was the possibility of burning the motor driver by accidentally swapping the connections from the electrical power source. To mitigate this risk, we made sure to clearly label the connections on the circuit board and double-checked the connections before applying power.

Wrong conversion of encoder data to speed that causes inaccurate speed control: We also identified the risk of incorrect conversion of encoder data to speed, which could lead to inaccurate speed control. To mitigate this risk, we carefully reviewed the mathematical formulas used to convert encoder data to speed, and also tested the control system with different loads to ensure accurate speed control.

Wrong selection of controller parameters that makes it difficult to achieve certain speed: Another risk was the possibility of wrong selection of controller parameters

that could make it difficult to achieve certain speeds. To mitigate this risk, we carefully reviewed the controller parameters and performed multiple testing to find the optimal settings.

In summary, we were able to identify and mitigate the major risks that were present during the project, ensuring that the control system was functioning as intended.

V. Design

A. System Diagram

A system block diagram shows a visual representation of system's components and subsystems, and how they interact with each other. It usually includes blocks which are input, control, actuators, and sensors.

- Input blocks represent the sources data that the system receives. Desired speed can be counted as an input for our system.
- Control blocks represent the decision-making components that the system uses to control its operation. microcontroller and PID control can be counted as an control block for our system.
- Actuator blocks represent the actuators in the system. They receive control signals from the control system and convert it to physical actions. DC motor can be counted as a actuator block for our system.
- Sensor blocks represent the sensors in the system. They detect changes in environments and convert the detected changes into electrical signals. Their duty is to provide input to the control system. Encoder can be counted as an control block for our system.

The implementation of motor speed control is shown in the system diagram in Figure 3. microcontroller, PID control, motor driver, DC motor, and encoder blocks are represented there. The microcontroller receives the user's requested speed and subtracts encoder data to find error. The PID controller receives the error and uses it for producing a PWM signal. The motor driver receives this PWM signal and uses it to drive the DC motor. The DC motor receives a control signal from the PID controller and uses it's speed. The encoder is sensing the motor's position and rotation, it is used to control the movement and location of the motor. The microcontroller is receiving encoder data, it uses to modify the PWM signal and keep the motor in desired speed.

Overall, the system diagram in Figure 3 demonstrates how a PID control for a DC motor with an encoder is implemented. Which allows for precise and efficient control of the movement and speed of the motor in our system.

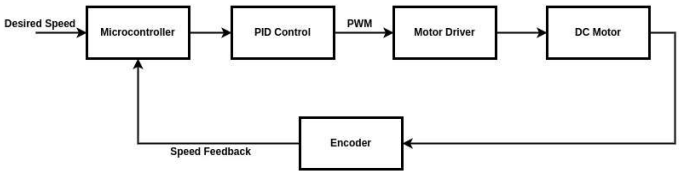

Fig. 3. System diagram

B. Electronic Hardware

The electronic hardware selection of our system is critical for system performance. The electronic hardware consists of several components, including microcontroller, motor, motor driver, power supply, breadboard and jumper wires. These components work together to perform the desired functions of the system.

1) Microcontroller: The microcontroller, which is the brain of the system, is responsible for processing the input data and controlling the output devices. In our system, we have used a high-performance ESP32-S2 that is efficient for motor speed control. [2]

Fig. 4. ESP32-S2 microcontroller

Specification	Data
Maximum speed	240 MHz
ROM	128 KB
SRAM	320KB

TABLE I
Microcontroller Specification

2) Motor: The 6V DC motor with an encoder is the actuator of the system with sensor on it. It can provide maximum 14mN*m torque and 6200 rpm speed, which is enough for the operation of the system. The use of a 6V DC motor also allows for a more efficient operation, as it requires less power compared to higher voltage motors.

The encoder which is mounted on motor allows us to monitor the position and speed of the motor. Our encoder provides a resolution of 48 pulses per revolution. The combination of a 6V DC motor with an encoder provides a precise and efficient control of the speed of the wheel in our system. [3]

Fig. 5. Pololu 6V DC motor with encoder

Specification	Data
Max. operating current	2.4A
Operating current	50mA
Nominal rotational speed	6200rpm
Resolution	48imp/revol.
Torque	max. 14mNm

TABLE II
Motor Specification

3) Motor Driver: The motor driver controls the speed and direction of the motor. We have used L298N motor driver that can handle maximum 35V, which we only need 6V. The motor drive can control of the speed and direction of the motor.We have used Pulse Width Modulation (PWM) to control the speed of the motor, and digital inputs to control the direction of the motor. [4]

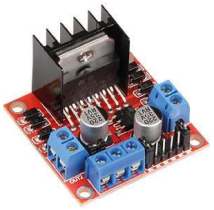

Fig. 6. L298N motor driver

4) Power Supply: The power supply is providing power to encoder and motor. We have used a adjustable power supply that can provide a maximum power of 18W with a maximum current of 1.5A. Which satisfy the power need of 6V DC motor. [5]

Motor power consumption was calculated and compared with the power supply maximum power and then it is decided that power supply can satisfy system's needs

Power Supply = 6V * 1.5A = 9 W

Motor Power = 6V * 50 mA = 0.3 W

Fig. 7. Adjustable power supply

5) Breadboard: The breadboard is used to wire system's component in a safer and easier way. In details, it is a plastic board with a grid of holes in it, that allows you to connect electronic component wires such as motor encoder and power cables.

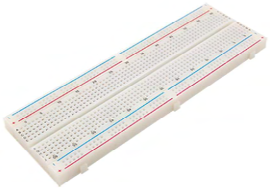

Fig. 8. Breadboard

6) Jumper Wires: The jumper wire is a short piece of insulated wire with connectors at both ends, we used it to make connections between microcontroller, motor and motor driver.

Fig. 9. Jumper Wires

C. Circuit Diagram

Figure 10 shows the circuit diagram for our proposed motor speed control system. The system consists of an ESP32-S2 controller, a 6V DC motor with encoder, a L298N motor driver, and an adjustable DC power supply. Fritzing software was used to draw this diagram. [6]

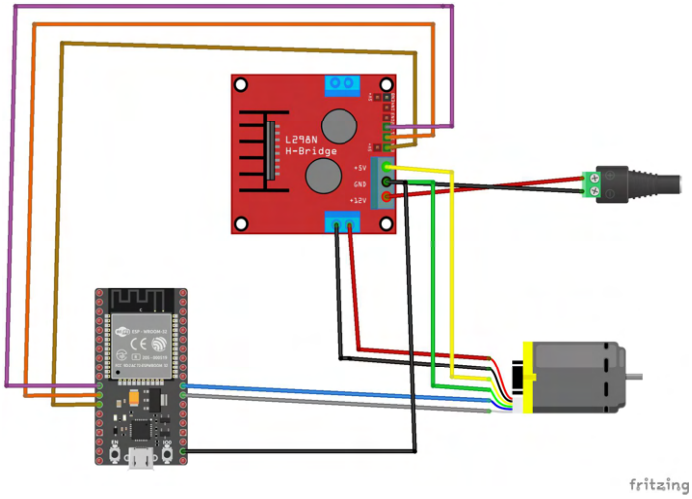

Fig. 10. Circuit diagram

The pin connections of our system is decided respect to power and data transformation needs of the components. General input-output pins are used in ESP32. Only one channel of L298N is used because we had only one motor to control.

The ESP32-S2 is receiving power from computer. This connection is also helpful to visualize motor speed via Arduino IDE. It is connected to the L298N using three pins: one for controlling the speed of the motor (PWM pin), two for controlling the direction of the motor (direction pins). ESP32-S2 is also connected to DC motor using two pins: each of them are receiving different encoder data. For the signal and power security: ESP32-S2 is grounded with L988N.

The L298N is connected to the DC motor using four pins: two for driving motor by giving voltage to it respect to the PWM data, two for supplying 5V to encoder which it can operate.

The DC power supply is connected to the L298N using two pins: one for providing power (VCC pin), and one for grounding(GND pin).

Overall, the pin connections of our system are designed to provide reliable and stable power and communication between the components.

D. Bill of Materials

One of the critical design criteria of the project was to stay in the limitation of given budget. It was achieved by ordering all of the electronics components under 50€.

Component	Price
6V DC Motor with Encoder	23€
Universal Power Supply 3-12V	11€
L298N Motor Driver	7€
Breadboard	5€
Jumper wires	3€

TABLE III
BOM List

E. Software

General overview of the used software are given in this section. They are Fritzing, SolidWorks, and Arduino.

1) Fritzing: Fritzing was used to design the circuit diagram before wiring on the real hardware. We have used it as a simulation and visualization tool. We were able to find some components on its library. However some of the components such as motor driver, needed to be added manually to the library. The software has a good community. You can find your problem's solution by asking to the forum. Usually questions were asked before you ask it in the forum.

Fig. 11. Fritzing Logo

2) SolidWorks: SolidWorks was used to design wheel of the system. It is one of the most common CAD programs and it is free with our student account. Therefore we have decided to use SolidWorks for mechanical design.

Fig. 12. SolidWorks logo

3) Arduino IDE: For the software development of ESP32, Arduino IDE was chosen. Because it can be downloaded in all operations system such as Windows, Mac, or Linux. The language which is used for programming is closed to the C++. Our main motivation to choose it was most of the students in our group had prior experience with this IDE.

Before starting the software, hardware library of ESP32 should be installed on IDE. It is explained in following steps: [7]

Fig. 13. Arduino IDE Logo

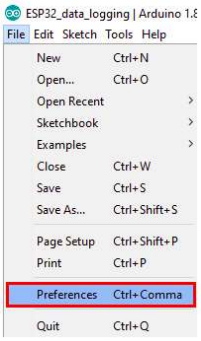

Fig. 14. Preferences

1) Preferences button should be clicked under File tab.

2) Following link should be added to the 'Additional Board Manager URLs' field:

```
https://raw.githubusercontent.com/espressif/arduino-esp32/gh-pages/package_esp32_index.json
```

Fig. 15. URL

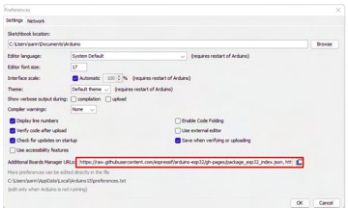

Fig. 16. Additional board URL

3) After adding hardware library, it can be installed from Board Manager

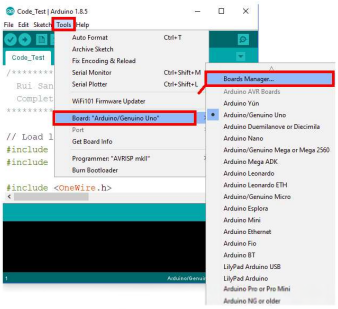

Fig. 17. Board Manager

Arduino IDE can be used for software development of ESP32 now. Board will be available in IDE and there won't be any problem during the compiling and uploading the code.

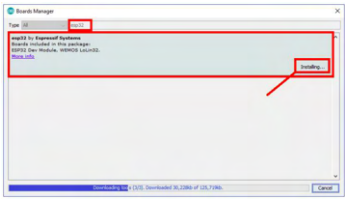

Fig. 18. Package installation

F. Mechanical Design

To make better demonstration, 3D printed wheel was designed with SolidWorks and manufactured by using 3D printer. Motor shaft teeth was taken from motor data-sheet. Diameter and thickness of the wheel was decided intuitively.

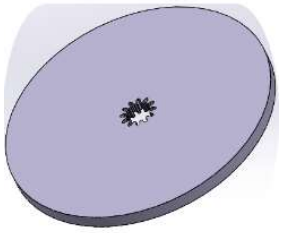

Fig. 19. CAD design of wheel

3D wheel was printed by using Ender3-Pro [8] printer. It was used during the Demonstration presentation in the class.

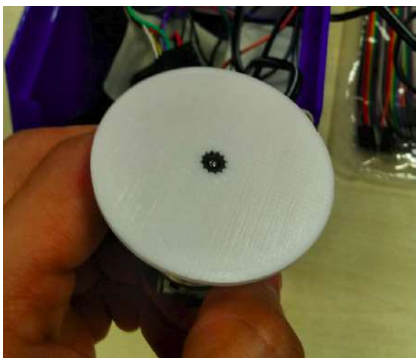

Fig. 20. Printed wheel

VI. Development

A. Setting the encoder

The encoder that was used in the project is a two-channel Hall effect encoder [9] which can sense the rotation of a magnetic disk attached to the rear part of the motor shaft. This encoder has a resolution of 48 CPR (counts per revolution), meaning that it can count up to 48 signals in one rotation of the disk. There are two output signals in the encoder, output A (in blue) and output B (in yellow) which can be seen in the figure 21

The 48 counts per revolution of the encoder comes from counting both edges of both channels (A and B). Both

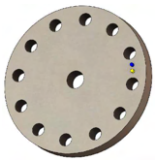

Fig. 21. 48 CPR encoder disk

edges means that the Rising and Falling states on both channels are used are signals to calculate the speed.

So, signal A will have 12 rising signals and 12 falling signals and signal B will have 12 rising signals and 12 falling signal, which count up to 48 CPR.

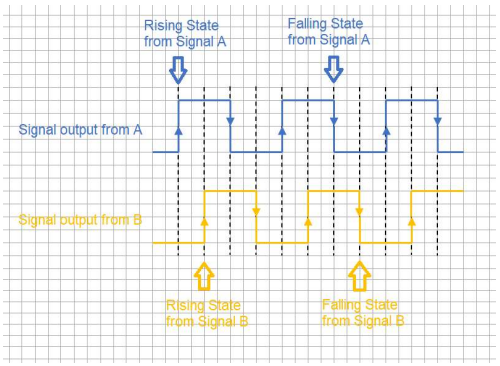

Fig. 22. encoder signals

In the figure 22, an representation of how the output signals of the encoder work is shown. There are four different signals than can be used for the speed and rotation control. There is a phase delay between the two signal of approximately 90°.

- Rising state from Signal A
- Falling state from Signal A
- Rising state from Signal B
- Falling state from Signal B

Using this logic, the speed and the direction of rotation of the motor can be calculated. In the project's code, we let the motor rotate for a small period of time while counting all the RISING signals from the output A. This means that for each revolution, the program will add 12 counts. After the interval of time (for example 500 ms), the program will take all the counts make in that time and use equation 1 to get RPMs:

$$RPMs = \frac{Counts * 1000[ms/s] * 60[s/m]}{12[counts/rev] * timeinternal[ms]} \quad (1)$$

To know in which direction the motor is rotating, we just need to compare the state of the other signal. For example, in the figure 22 when the output signal of A is RISING, the state of the output signal B is LOW. This indicates one direction of rotation. If the state of the output signal B is HIGH, that would indicate a rotation in the other direction.

B. Using Arduino IDE to get the speed

To get the ESP32 to read the signals from A and B, the initial approach was to compare the signals on every step to their previous value. Lets say the code check signal A, and reads 0 logic, on the next loop the code will check again and this time it reads a 1. For the program, this means a RISING state, so the count variable can be added by one.

However, there is a problem with this approach. The Arduino IDE loop function can last longer than the time signal A takes to go from HIGH to LOW. This means that there will be times when the ESP32 reads a 0 logic two times in a row, skipping one of the counts.

This results in inaccurate calculation of the motor. This problem was found out because the nominal speed of the motor at max voltage is 6200 RPMs according to the manufacturer, and the speed according to the code was around 4500 RPMs.

To solve this problem, there is a special function events on microcontrollers called "Interrupts".

An interrupt is an special event that triggers with a special signal and stops the main loop of the program, executes a small instruction called **Interrupt Service Routine**, and returns to the main loop again. This interrupt function is extremely useful when a special event needs to happen regardless of the code in the main loop. ESP32 uses a special RAM memory location called the Instruction RAM (IRAM [10]) for the Interrupts subroutines. This is to prevent time loss from reading the instruction from the memory space where all the other subroutines are.

In the project, an Interrupt subroutine was created (figure 23) so whenever the ESP32 detected a Rising state change on the A channel, it would stop the main loop, add a count to the counter variable, and return to the main loop. The Subroutine also reads the state of the B signal, to check if the motor is rotating clockwise or counter clockwise.

```
void IRAM_ATTR count_funct()
{
  int b = digitalRead(encoderPinB);
  if(b > 0){
    count++;
  }
  else{
    count--;
  }
}
```

Fig. 23. Interrupt subroutine code

With the use of interrupts, the program can now get a more accurate speed reading of the motor, and we can begin with the controller design. For example, a PWM of 1/3 was sent to the motor driver and the code results is shown in the figure 24

C. PID controller development

One of the most used controllers in the education and industry fields is the PID controller. It can be used for

```
float measureSpeed(float timeInterval){
  float speed = ((count*1000*60)/(12*timeInterval));
  Serial.print("RPMs = ");
  Serial.println(speed);
  return speed;
}
```

```
18:47:20.012 -> 290.00
18:47:20.012 -> RPMS:2068.97
18:47:20.264 -> 290.00
18:47:20.309 -> RPMS:2068.97
18:47:20.560 -> 290.00
18:47:20.560 -> RPMS:2068.97
18:47:20.884 -> 290.00
18:47:20.884 -> RPMS:2068.97
18:47:21.153 -> 290.00
18:47:21.153 -> RPMS:2068.97
```

Fig. 24. RPM reading with a 1/3 PWM signal

simple systems to robust control. It is a closed loop control, meaning that a feedback is needed for the control system, in this case we will be using the encoder to measure the motor's speed and use that information to reduce and try to get rid of the error. PID can be seen as a controller that uses the present, the past and the future states of the error of the system to reach a desirable state.

Because of the popularity and flexibility of this controller, we decided to try and test different PID controller for our project. The PID diagram of the controller can be seen in figure 25

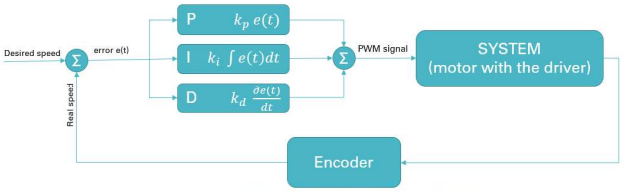

Fig. 25. PID diagram

Basically, the PID has three components. The Proportional part, the Integral part and the Derivative part. Each part interacts in a different way with the system, and helps to adjust the output signal. The control depends on the error, which is calculated by subtracting the actual reading of the output with a desired value called Set point

$$error = Desired\_speed - actual\_speed \tag{2}$$

1) Proportional component: The proportional component can be seen as the "present state" of the error, and is pretty straightforward. It works directly on the error of the system. The bigger the error, the bigger the correction. Proportional control uses a variable called Kp, which is multiplied with the error of the system in time, and gives a control signal for the system.

$$Proportional = K_p * e(t) \tag{3}$$

However, the closer the output signal is to the set point, the control signal will get smaller and smaller until

it doesn't affect the system at all. This leaves us with a Steady state error that the proportional control can't eliminate from the system.

Using a larger Kp value can increase the speed of the controller, but the system will overshoot a lot, and can start to oscillate becoming unstable.

2) Integral component: The Integral component affects the "future state" of the error. This component is proportional to the magnitud of the error and also the duration of the error. To get rid of the Steady State error, this component will keep adding previous errors in the system in time (integration of the errors) and multiply them with a Integral variable called Ki and give us a new control signal for the system

$$Integral = K_i * \int e(t)dt \tag{4}$$

Even a small value in the error of the system will make the Integral control signal to change, so this is a good way to get rid of the steady state error. But larger values of Ki can slow the speed response of the controller and affect the stability or transient response of the system.

PI controllers are used then a high-speed response of the system is not required.

3) Derivative component: The last part of the PID controller is the derivative component, which will try to control the using using the "past state" of the error. The derivative component is calculated using the slope of the error over time, this is the difference of two error in a give period of time. High values on the Kp and Ki constants can make the system response faster, but less stable. The derivative component is used to help the system reduce this problem.

$$Derivative = K_d * \frac{\delta e(t)}{\delta t} \tag{5}$$

If the rate of change in the system is large, the derivative component will slow down the ramp of the control signal, meaning less overshoot and more stability of the system.

The table IV shows a small summarize of how each component of the PID controller affects the control of the system.

Constant	Rise t.	Overshoot	Settling t	Steady-State e.
K_p	↓	↑	Small ↑	↓
K_i	↓	↑	↑	Big ↓
K_d	Small ↑	Small ↓	Small ↓	No change

TABLE IV
EFFECT OF PID PARAMETERS ON A SYSTEM [11]

In this table, an ↑ means an increase of that parameter and a ↓ means the decrease. For example, increasing the K_p will decrease the Rise time, increase the Overshoot, slightly increase the Settling time and decrease the Steady-state error.

D. Implementing PID in the ESP32

To implement the PID control in the Arduino IDE, the following variables are needed:

- Actual speed of the motor (from the encoder)
- Desired speed of the system (Set point)
- Time interval between control loops
- Kp, Ki and Kd variables
- Variable for the accumulative errors
- Variable for the error from the previous control loop
- Variable for the control signal

```
float setpoint = 3000.0;          // Desired speed -> 3000 RPMs
float power = 0;                  // Variable for control signal
float kp = 0;                     // Proportional Constant
float ki = 0;                     // Integral Constant
float kd = 0;                     // Derivative Constant
float error = 0;                  // Variable for the error of the system
float previousError = 0;          // Variable for the previous error
float previousInte = 0;           // Variable for the accumulative errors in time
```

Fig. 26. PID variables for control code

The interval time for the control loop and the speed calculation starts with 500 ms. This time interval will get lower when the code is totally functional.

In every control loop the system will:

1) Get the speed value from the encoder and reset the counts to 0.
2) Calculate the error
3) Calculate the Proportional component of the control system
4) Calculate the Integral component of the control system
5) Calculate the Derivative component of the control system
6) Get the control signal. (Proportional+ Integral + Derivative components)
7) Adapt the control signal as PWM signal for the motor driver.
8) Update the previous error variable
9) Update the accumulative error variable
10) Repeat the control loop

The code in the Arduino IDE is in the figure 27

```
// time difference
deltaTime = (float)(millis()-previousTime);
//Serial.println(deltaTime);
if(deltaTime >= 500)
{
  previousTime = previousTime + deltaTime;

  rpms = measureSpeed(deltaTime);
  count = 0;
  error = errorCalculation(setpoint, rpms);

  // proportional
  float proportional = kp*error;

  // derivative
  float derivative = kd*(error-previousError)/(deltaTime);

  // integral
  float integral = previousInte + deltaTime*(error+previousError)/2;
  // control signal
  float u = proportional+derivative+integral;

  // motor power
  power =fabs(power + u);
  if( power > 255 ){
    power = 255;
  }
setMotor(power, pwmChannel,motorPin1, motorPin2);

previousError = error;
previousInte = integral;
}
```

Fig. 27. PID code

To test if the code is working, the Integral and Derivative values were set to 0 and a really small Proportional constant was set.

The result is shown in the figure 28

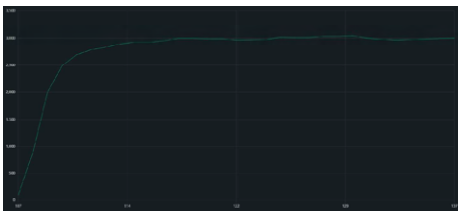

Fig. 28. PID code test

The figure 28 is only a test to see how the system reacts to the controller. This controller does not have any tuning yet, but with all the components working correctly, the PID control system can be tuned for optimal speed control.

VII. Testing

A. Tuning methods classifications and types

The controller tuning methods are classified into two main categories, closed-loop, and open-loop. The closed-loop tuning refers to tuning a controller for a system that has the ability to self-adjustment its state, while open-loop tuning refers to tuning a controller for a system that is manually controlled without self-adjustment, as shown in Fig. 29.

In this project, we implemented closed-loop tuning techniques which are considered more accurate and efficient than open-loop tuning techniques.

There are several closed-loop methods to tune the controller:

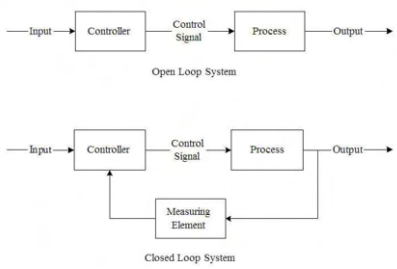

Fig. 29. Open vs closed loop

- Heuristic tuning method.
- Closed loop Ziegler-Nichols method.
- Modified Ziegler–Nichols method.

1) Heuristic tuning method: This method steps through the controller's gains from proportional to integral to derivative in sequence. Usually, for the new PID loops, the method involves starting with a rough and safe initial guess without implementing any mathematical equations and adjusting the gains incrementally or detrimentally according to the process response till getting the desired performance is achieved for each process gain. This depends on the following[12]:

- The proportional gain (P-action) is introduced to increase the speed of the response by controlling the rising time, which is the amount of time the system takes to go from 10% to 90% of the steady-state, or the final value. High proportional values result in oscillation.
- The integral gain (I-action) is introduced to obtain a desired steady-state response that is within 5% of the final value by controlling the settling time. This will result in a higher oscillating response over a longer period.
- The derivative gain (D-action) is introduced for damping overshoots. This may result in high-frequency oscillation, plus the noise sensitivity, as shown in Fig. 30.

Another important issue is the different types of system responses which are discussed in detail in the demonstration section.

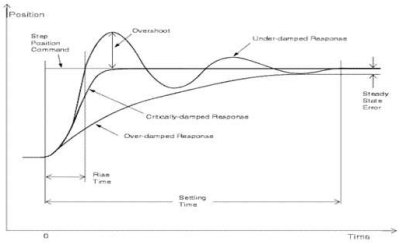

Fig. 30. critically damped, overdamped, and underdamped [13]

Pros and cons:

- Pro: Intuitive and simple approach, i.e., initial assumptions are made according to personal experience about the process.
- Con: Takes a long time to achieve good performance.
- Con: Doesn't guarantee to reach a robust solution.

2) Closed loop Ziegler-Nichols tuning method: The most widely used method for tuning P, PI, and PID controllers is based on sustained oscillations. It was introduced for the first time by Ziegler and Nichols in 1942. This method depends on measuring 2 important parameters which are ultimate gain and its oscillating frequency (Kmax and f0). The ultimate gain Kmax can be measured by setting both the integral and derivative parameters into zeros, then incrementing the proportional factor till the controller starts to become unstable and oscillates in a sustainable form without decay. The value of the proportional gain at this moment will be the ultimate gain Kmax, oscillating every T0. The frequency of oscillation can be calculated by using the following equation.

$$f0 = \frac{1}{T0} \tag{6}$$

After that, the gain can be calculated for each part of the controller according to the table V:

Controller Type	Kp	Ki	Kd
P	0.5 Kmax	-	-
PI	0.45 Kmax	1.2 f0	-
PID	0.6 Kmax	2.0 f0	0.125 f0

TABLE V
Closed loop Ziegler-Nichols method relations

A brief explanation of the closed-loop Ziegler-Nichols tuning method is shown in Fig. 31

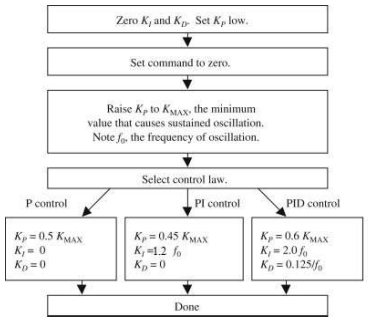

Fig. 31. Closed-loop Ziegler and Nichols tuning method [14]

Pros and cons:

- Pro: The process is simple, consuming less time while getting a reasonable performance for simple loops.
- Con: The gain values obtained from this method still require some fine-tuning to achieve the desired performance.
- Con: The method is based on the assumption of a first-order process with a relatively long time constant T with respect to the dead time.
- Con: High proportional gains due to the 25% decay ratio design specification every cycle,
- Con: Low integral action with too low damping of the closed-loop system, which would result in aggressive control performance, providing a smaller stability margin against changes in the process dynamics and non-linearities.

3) Modified Ziegler–Nichols method: This method was introduced to overcome the large overshoots issue and 25% decay ratio issues in the classical Ziegler-Nichols method by introducing 2 different settings, which are some overshoots, and no overshoots settings, according to the application requirements. Also, it depends on the ultimate gain and time constant measurements.[15] The following table no. VI shows these modified settings calculations.

Controller Type	Kp	Ki	Kd
Some overshoots	0.33 Kmax	2.0 f0	3.0 f0
No overshoots	0.2 Kmax	2.0 f0	3.0 f0

TABLE VI
Modified Ziegler–Nichols method relations

B. Heuristic tuning method results

In this project, we have implemented both the heuristic, and closed-loop Ziegler-Nichols tuning methods for P, PI, PD, and PID controllers. Better serial plotter software was used to plot the system response which has several features like autofocus, names, and colors of variables that can be changed. Also, saving the output into a CSV file.

1) Proportional gain heuristic tuning method results for P, PI, and PID controllers: Three examples will be demonstrated for the system response as a result of proportional gain Kp variation.

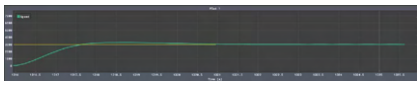

Fig. 32. Kp = 0.005

Kp = 0.005, rising time = 2s, overshoot = 3310 rpm = 10.34%, and settling time = 4.5s. According to fig. 32, the response is slow which is nearly critically damped with a good settling time.

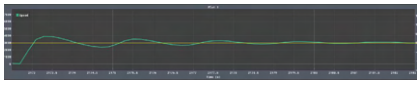

Fig. 33. Kp = 0.05

Kp = 0.05, rising time = 0.45s, overshoot = 3905 rpm = 30.167%, and settling time = 8.5s. According to fig. 33, the response is very fast with high overshoot and a very large settling time.

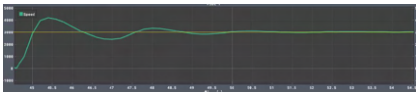

Fig. 34. Kp = 0.03

Kp = 0.03, rising time = 0.45s, overshoot = 4372 rpm = 45.73%, and settling time = 4.75s. According to fig. 34, the response is very fast with a very high overshoot and a good settling time.

The best proportional gain value is Kp = 0.03 with high a rising time and acceptable settling time when the heuristic tuning method is implemented.

2) Integral gain heuristic tuning method results for PI, and PID controllers: Three examples will be demonstrated for the system response as a result of integral gain Ki variation.

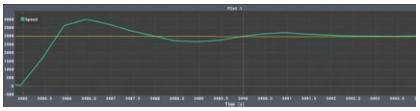

Fig. 35. Kp = 0.03, Ki = 0.2

Kp = 0.03, Ki = 0.2, rising time = 0.85s, overshoot = 4034 rpm = 34.47%, and settling time = 6.35s. According to fig. 35, the response is slow with a high overshoot and a relatively large settling time.

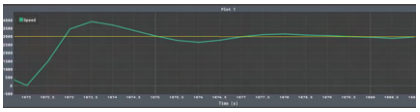

Fig. 36. Kp = 0.03, Ki = 1.7

Kp = 0.03, Ki = 1.7, rising time = 0.85s, overshoot = 3973 rpm = 32.5%, and settling time = 7.25s. According to fig. 36, the response is slower with a high overshoot and a very large settling time.

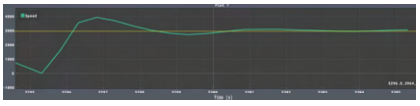

Fig. 37. Kp = 0.03, Ki = 0.35

Kp = 0.03, Ki = 0.35, rising time = 0.85s, overshoot = 3964 rpm = 32.1%, and settling time = 4.5s. According to fig. 37, the response is still slow with the same overshoot but the settling time has greatly improved. **The best integral gain value is Ki = 0.35 with the lowest settling time and acceptable rising time when the heuristic tuning method is implemented.**

3) Derivative gain heuristic tuning method results for PD controllers: Three examples will be demonstrated for the system response as a result of derivative gain Kd variation in the PD controller.

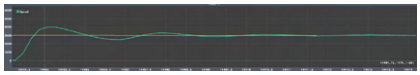

Fig. 38. Kp = 0.03, Kd = 0.025

Kp = 0.03, Kd = 0.025, rising time = 0.5s, overshoot = 4014 rpm = 33.8%, and settling time = 4.25s.

According to fig. 38, the response is fast with a high overshoot but the settling time is not the best.

Fig. 39. Kp = 0.03, Kd = 15

Kp = 0.03, Kd = 15, rising time = 2.7s, no overshoot = 2970 rpm, and settling time = 1.6s. According to fig. 39, the response time is greater than the settling time without overshooting.

Fig. 40. Kp = 0.03, Kd = 12.5

Kp = 0.03, Kd = 12.5, rising time = 0.4s, overshoot = 3043 rpm = 1.43%, and settling time = 1.7s. According to fig. 40, the response time has improved with a negligible overshoot and a very short settling time. **The best derivative gain value for PD controller is Kd = 12.5 with negligible overshoot, good rising time, and a very short settling time when the heuristic tuning method is implemented.**

4) Derivative gain heuristic tuning method results for PID controllers: Three examples will be demonstrated for the system response as a result of derivative gain Kd variation in the PID controller.

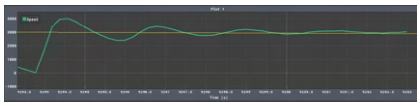

Fig. 41. Kp = 0.04, Ki = 0.35, Kd = 0.001

Kp = 0.03, Ki = 0.35, Kd = 0.001, rising time = 0.4s, overshoot = 4034 rpm = 34.46%, and settling time = 5.5s. According to fig. **??**, the response time is good with a high overshoot and a long settling time.

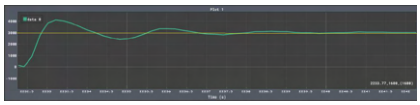

Fig. 42. Kp = 0.03, Ki = 0.35, Kd = 0.0735

Kp = 0.03, Ki = 0,35, Kd = 0.0735, rising time = 0.35s, overshoot = 4080 rpm = 36.0%, and settling time = 5.5s.

According to fig. 42, the response time is very fast without a significant improvement in overshoot and the same long settling time.

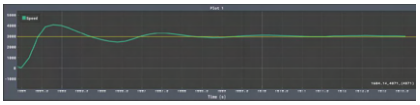

Fig. 43. Kp = 0.03, Ki = 0.35, Kd = 12.5

Kp = 0.03, Ki = 0.35, Kd = 12.5, rising time = 0.4s, overshoot = 4057 rpm = 35.23%, and settling time = 4s. According to fig. **??**, the response time is fast without a significant improvement in overshoot and the settling time has improved significantly. **The best derivative gain value for PD controller is Kd = 12.5 with a good rising time and a very short settling time. Unfortunately, there is no significant effect for the derivative gain in reducing the overshoot in the PID controller when the heuristic tuning method is implemented.**

C. Closed loop Ziegler-Nichols tuning method results

1) Ultimate gain testing results: After conducting several tests, the system begins to oscillate in a sustained frequency with an ultimate gain of Kmax 0.082 and a period of 1.7 seconds.

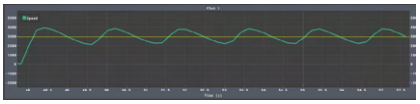

Fig. 44. Kmax = 0.082

Kmax = 0.082, T = 1.7s, and f0 = 1/T = 0.588Hz.

2) Closed loop Ziegler-Nichols tuning for P controller testing results: Applying the equations for P controller gain factors from table V Kp = 0.041. The system response will be as follows:

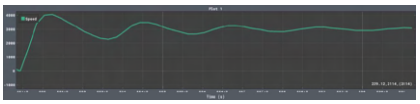

Fig. 45. Kp = 0.041

Rising time = 0.35s, overshoot = 4071 rpm = 35.57%, and settling time = 7.75s.

According to fig. 45, the response time is very fast with high overshoot and a very long settling time.

3) Closed loop Ziegler-Nichols tuning for PI controller testing results: Applying the equations for PI controller gain factors from table V Kp = 0.0369, and Ki = 0.7056. The system response will be as follows:

Rising time = 0.75s, overshoot = 4051 rpm = 35.03%, and settling time = 6.1s.

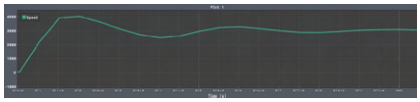

Fig. 46. Kp = 0.0369, Ki = 0.7056

According to fig. 46, the response time is slow with high overshoot and long settling time.

4) Closed loop Ziegler-Nichols tuning for PID controller testing results: Applying the equations for PID controller gain factors from table V Kp = 0.0492, Ki = 1.176, and Kd = 0.0735. The system response will be as follows:

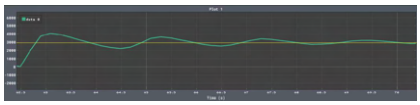

Fig. 47. Kp = 0.0492, Ki = 1.176, Kd = 0.0735

Rising time = 0.34s, overshoot = 4046 rpm = 34.867%, and settling time = 7.25s.

According to fig. 47, the response time is very fast with high overshoot and a very long settling time.

VIII. Demonstration

Finally, we need to demonstrate the differences between these controllers and which one can be considered better, and to go through this we need first to understand some definitions that will be used to evaluate the system.

Overshooting: It's the maximum positive deviation output with respect to the desired value.

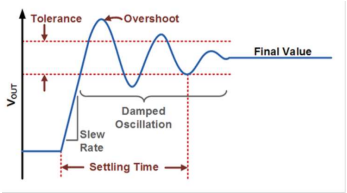

Fig. 48. Overshooting

Settling Time: the time from the input transition to the time when the output enters the specified error zone and does not leave again.

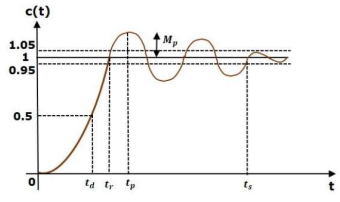

Fig. 49. Settling Time

Rise time: It's the time separating two points on the rising edge of a signal. The points are often located 10% and 90% up the rising edge of the curve, but sometimes other points on the curve are chosen.

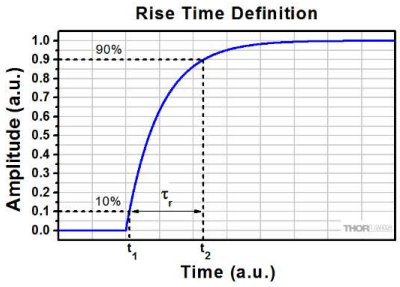

Fig. 50. Rise Time

For heuristic proportional gain tuning for the P controller, and based on the results of many trials, we've noticed that the best evaluation happened at KP = 0.03, and we found it has a Rising time = 0.45s, Overshoot = 4372 rpm = 45.73%, and Settling time = 4.75s. The overshoot seems to be high but both settling and rising times are very good here.

For Closed loop Ziegler-Nichols tuning for P controller, and based on the results of many trials, we've noticed that the best evaluation happened at KP = 0.041 and we found that its Rising time = 0.35s, Overshoot = 4071 rpm = 35.57%, and Settling time = 7.75s. Although the response time is very fast and the overshoot is better, the settling time is more than the heuristic.

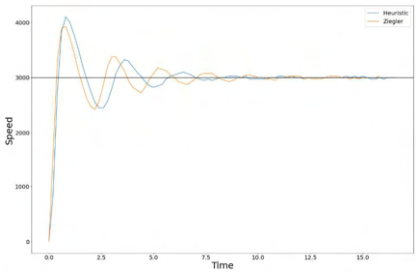

Fig. 51. P Controller Results

For heuristic proportional gain tuning for the PI controller, and based on the results of many trials, we've noticed that the best evaluation happened at KP = 0.03, and KI = 0.35 and we found its Rising time = 0.85s, Overshoot = 3964 rpm = 32.1%, and Settling time = 4.5s. And although we can clearly see that the rising time is more than the case in the P controller, we still got better overshoot and settling time.

For Closed loop Ziegler-Nichols tuning for PI controller, and based on the results of many trials, we've noticed that the best evaluation happened at KP = 0.0369, and KI = 0.7056, and its Rising time = 0.75s, Overshoot = 4051 rpm = 35.03%, and Settling time = 6.1s. So rising time has improved, but overshoot and settling time were better with the heuristic.

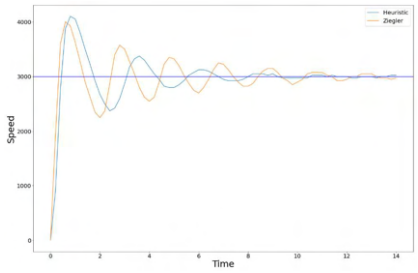

Fig. 52. PI Controller Results

For heuristic proportional gain tuning for the PD controller, and based on the results of many trials, we've noticed that the best evaluation happened at KP = 0.03, KD = 12.5 and we found its Rising time = 0.4s, Overshoot = 3043 rpm = 1.43%, and Settling time = 1.7s. And here we can see that rising time is the best so far, there's almost no overshooting, and settling time became the best.

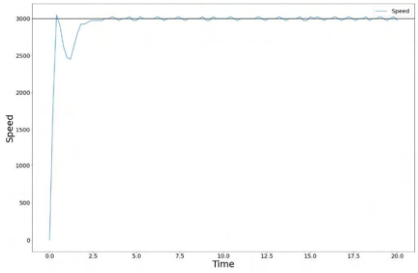

Fig. 53. PD Controller Result

For heuristic proportional gain tuning for the PID controller, and based on the results of many trials, we've noticed that the best evaluation happened at KP = 0.03, KI = 0.35, KD = 12.5 and we found its Rising time = 0.4s, Overshoot = 4057 rpm = 35.23%, and Settling time = 4s. The response time is fast without a significant improvement in overshoot and the settling time has improved significantly.

For Closed loop Ziegler-Nichols tuning for PID controller, and based on the results of many trials, we've noticed that the best evaluation happened at KP = 0.0492, KI = 1.176, and KD = 0.0735, and we found its Rising time = 0.34s, Overshoot = 4046 rpm = 34.86%, and Settling time = 7.25s.

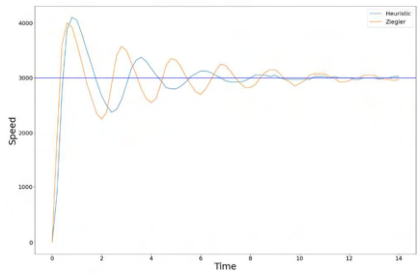

Fig. 54. PID Controller Results

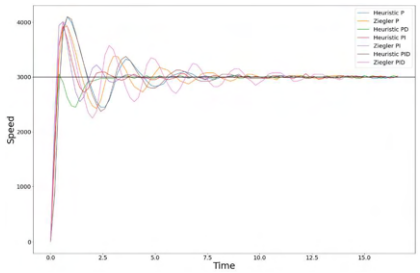

Fig. 55. All Results

PID (Proportional-Integral-Derivative) controllers are widely used in industrial control systems and other applications that require continuously modulated control. This is because PID controllers have several advantages over other types of controllers. These include:

Simplicity: PID controllers are relatively simple to design and implement, making them a popular choice for many control applications.

Robustness: PID controllers are relatively insensitive to parameter variations and unmodeled dynamics, making them suitable for controlling a wide range of systems.

Good performance: PID controllers can provide good performance in terms of reference tracking and disturbance rejection, which are important for many control applications.

Widely available: PID controllers are widely available in a variety of forms including software libraries and commercial off-the-shelf hardware.

Cost Effective: PID controllers are relatively inexpensive and easy to implement, which makes them a cost-effective solution for many control applications.

Easy to tune: PID controllers are easy to tune, which means that the controller can be adjusted to achieve the desired performance.

And therefore we have made a lot of trials on PID controller by changing the parameters KP, KI, and KD, and below is the results:

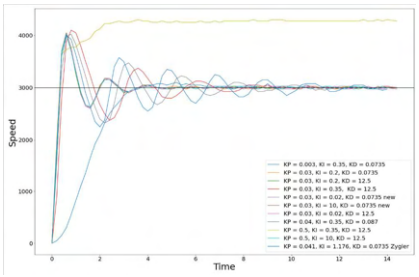

Fig. 56. PID Controller Trials

Based on these results and explainations, we decided to go with the PID controller with KP = 0.003, KI = 0.35, and KP = 0.0735, and the overshoot for this system was almost perfect with just 3065 (2.19%), and both settling and rising times were 3.25 seconds.

IX. Compilation and outlook

There are several ways to improve a motor speed control project, including:

Choosing the appropriate control method: Depending on the application, different control methods may be more suitable. For example, a variable frequency drive may be more appropriate for applications that require precise speed control, while a simple voltage control circuit may be sufficient for less demanding applications.

Adding feedback control: By incorporating a sensor to measure the motor's speed, and using that information to adjust the control inputs, the system can achieve more precise and stable speed control.

Improving power supply: A motor's speed is closely related to the voltage and frequency supplied to it. By providing a clean and stable power supply, the motor can run at its optimal speed.

Choosing the right motor: Depending on the application, different types of motors may be more appropriate. For example, a DC motor may be more suitable for applications that require precise speed control, while an AC motor may be more suitable for applications that require high torque at low speeds.

Using advanced control algorithms: Many advanced control algorithms have been developed to improve the performance of motor speed control systems. These include PID controllers, fuzzy logic controllers, and neural network-based controllers.

Considering the load and environment conditions: The load and environment conditions will affect the performance of the motor speed control system. For example, a system that is operating in a high temperature environment may require additional cooling,

while a system that is subject to large loads may require a more powerful motor.

Regularly monitoring and tuning: Regularly monitoring the performance of the motor speed control system and tuning the control parameters can help to improve the system's performance over time.

One way to make motor speed control smart is to use a closed-loop control system, which uses feedback from sensors to adjust the motor's speed. This can include using sensors to measure the motor's current speed, and then using that information to adjust the motor's input power to achieve the desired speed. Another way to make motor speed control smart is to use a programmable controller, such as a PLC, to control the motor. This allows for the use of more advanced control algorithms and can also allow for remote monitoring and control of the motor. Additionally, using machine learning algorithms for prediction and optimization of the motor speed control can also be considered as a smart approach.

There are several ways to apply artificial intelligence (AI) to control the speed of a motor:

Predictive control: Using historical data and machine learning algorithms, an AI model can predict the desired speed of the motor based on various inputs such as load, temperature, and voltage. This can help to optimize the speed control of the motor and improve its efficiency.

Adaptive control: An AI-based controller can adapt to changing conditions in real-time and adjust the speed of the motor accordingly. This can help to improve the stability and accuracy of the motor control system.

Reinforcement learning: An AI-based controller can be trained using reinforcement learning techniques to optimize the speed control of the motor. This can help to improve the overall performance of the motor control system.

Self-optimizing control: An AI-based controller can learn from the data it receives from the motor and its environment, and adjust its control parameters to optimize the motor speed.

Anomaly detection: AI can be used to monitor the motor and detect any abnormal behavior, such as increased vibration or increased current consumption, which could indicate a problem with the motor. This can help to prevent damage to the motor and improve its overall reliability.

It's worth noting that applying AI to control the speed of a motor requires a good understanding of the motor dynamics and control principles, as well as a significant amount of data to train the AI model with.

References

[1] A. G. Nair, "Esp32 with dc motor and l298n motor driver," accessed: 26-10-2022. [Online]. Available: https://www.electrorules.com/esp32-with-dc-motor-and-l298n-motor-driver-control-speed-and-direction/

[2] ESPRESSIF, "Esp32-s2-saola-1," https://docs.espressif.com/projects/esp-idf/en/latest/esp32s2/hw-reference/esp32s2/user-guide-saola-1-v1.2.html, 2023.

[3] T. E. Component, "Motor," https://www.tme.eu/de/en/details/pololu-4820/dc-motors/pololu/lp-6v-motor-48-cpr-encoder-25d-metal-gea/, 2023.

[4] Reichelt, "Motor driver," , 2016.

[5] ——, "Power supply," https://www.reichelt.de/universal-schaltnetzteil-18-w-3-12-v-1500-ma-son-x-ps025-p288294.html, 2023.

[6] Fritzing, "Fritzing," https://fritzing.org/, 2023.

[7] R. Santos, "Installing the esp32 board in arduino ide (windows, mac os x, linux)," https://randomnerdtutorials.com/installing-the-esp32-board-in-arduino-ide-windows-instructions/, 2016.

[8] Creality, "Ender-3 pro 3d printer," https://www.creality.com/products/ender-3-pro-3d-printer, 2016.

[9] P. Corporation, "25d metal gearmotors," accessed: 15-12-2022. [Online]. Available: https://www.pololu.com/file/0J1829/pololu-25d-metal-gearmotors.pdf

[10] Espressiff. Memory types. [Online]. Available: https://docs.espressif.com/projects/esp-idf/en/latest/esp32/api-guides/memory-types.html

[11] K. Tehrani and A. Mpanda, PID Control Theory, 02 2012.

[12] INCAtools. Explore the 3 pid tuning methods. [Online]. Available: https://www.incatools.com/pid-tuning/pid-tuning-methods

[13] D. Collins, "How to address overshoot in servo control," https://www.motioncontroltips.com/how-to-address-overshoot-in-servo-control/.

[14] G. Ellis. Ziegler nichols method. Accessed: 13-01-2023. [Online]. Available: https://www.sciencedirect.com/topics/computer-science/ziegler-nichols-method

[15] M. Shahrokhi and A. Zomorrodi, Comparison of PID Controller Tuning Methods.