The University of Jordan

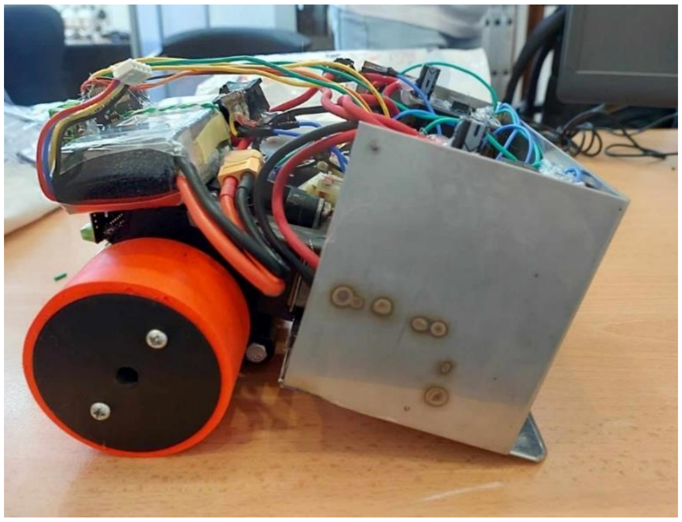
School of Engineering

Mechatronics Engineering Department

Artificial Intelligence

0908485

Implementing PID controller-based ANN and Fuzzy Logic Controller to a DC Encoder Motor – "A case of a Sumo Robot" –



Wael Jad Allah 0195016

Nada Zeineddin 0195841

Esra'a Tanashat 2180118

January 2024

Table of Contents

Table of Figures

List of Tables

Literature Review and System Understanding

Direct current (DC) motors remain a workhorse in the realm of control systems, offering versatile outputs like angular speed and displacement. Within the sumo robot domain, their integration with wheels, drums, or cables unlocks efficient translational motion [1].

PID controllers, renowned for their error-minimizing prowess through dynamic process control input adjustments, have earned their place as a mainstay in industrial settings. Appreciated for their high-performance characteristics, PID controllers stand as fundamental continuous feedback mechanisms [2].

This report delves into the utilization of both conventional PID controllers and Artificial Neural Networks (ANNs) in conjunction with DC motors to achieve optimal performance. Employing MATLAB simulation software for motor and controller modeling, we conduct a comparative analysis of PID and ANN controllers. The results showcase significant strides in ANN development, highlighting their effectiveness in simulating motor operations and delivering superior real-time control outcomes [3].

Beyond the general confines of mobile robots, sumo robots operate within the specialized domain of sumo wrestling competitions. These unique robots necessitate precise control and robust navigation capabilities to excel within the tightly constrained arenas they compete in. Integrating PID and fuzzy controllers becomes crucial in achieving the agility and responsiveness demanded for successful sumo robot maneuvers. [4]

The application of PID and fuzzy controllers in DC motor control for sumo robots emerges as a vital area of research and development. This review emphasizes the established effectiveness of PID controllers in industrial settings and explores the burgeoning potential of ANNs for enhanced motor operation simulation. Sumo robots, with their specialized competition contexts, underscore the pressing need for precise control mechanisms, highlighting the critical role advanced control algorithms play in securing optimal navigation and success within confined arenas. This interdisciplinary exploration sheds light on the synergistic interplay between DC motor control, specialized robotics, and advanced control systems, particularly in the captivating realm of sumo robot applications.

Mathematical Representation

DC motor modeling

For a dc motor open-loop dynamic equation is [5]:

$$L\frac{di}{dt} + iR + k_b\varpi = V \tag{1}$$

$$J\frac{d\omega}{dt} = k_t i + T_d \tag{2}$$

where L is armature inductance (h), R is armature resistance (ohm), V is input voltage (volt), I for armature current (ampere), kb for back emf constants, and Kt reveals as torque constants.

DC motor model used in this study

In this study, DC motor with encoder is used. The model parameters are shown in Table 1

Table 1: DC motor model parameter being studied.

Parameters	Nominal
R_a (Armature Resistance)	22 ohms
L_a (Armature Inductance)	100 mH
Nominal Voltage	5V

In order to obtain the characteristics of the system, the system feed with several step voltage have magnitude at 14V and 15V. These step voltage response graphs shown in Figure 1.

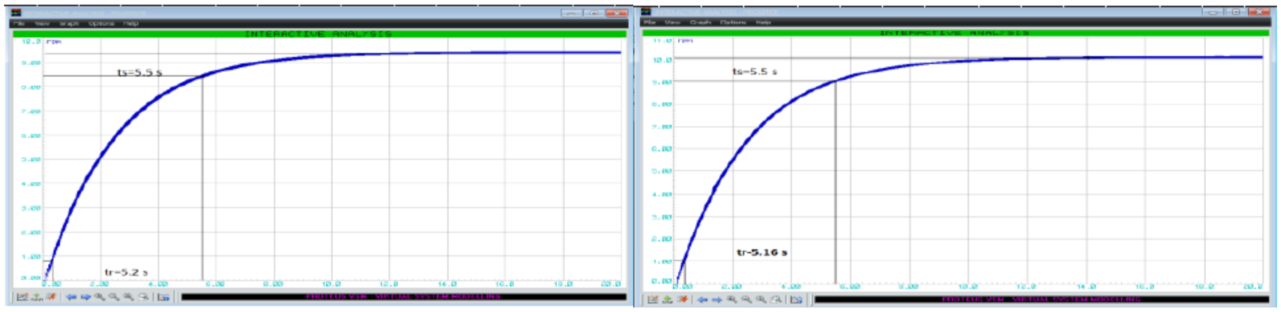


Figure 1:Graph Response for 14 & 15 V step

Figure 1 visual examination, shows the system is an overdamped system, its means that system has a dumping ratio greater than 1. Also based on graphics examination, proteus simulation results give consistent results, the system has a rise time value and setting a relatively constant as shown in the following Table 2.

Table 2: Settling time, t_r and t_s of DC motor model for various step input.

V_{step}, V	$N_{setling}$, rpm	t_r, s	t_s, s
14	940	5.2	5.5
15	1100	5.2	5.5
16	1080	5.2	5.6

Sampled data from the DC motor model step response.

In order to get step response data sample, motor dc model is feed with 15 V step input and then data is sampled in 0.25 second. Table 3 shown this sampling result.

Table 3: Data sampled at 0.25 s interval of dc motor model speed when feeding with 15 v step signal.

Time, s	N, rpm	No.	N, rpm	No.	N, rpm	No.	N, rpm	No.	N, rpm
0	0	1	6.4900	2	8.8200	3	9.7100	4	9.99
0.25	0.6900	1.25	6.8200	2.25	8.9600	3.25	9.7600	4.25	9.99
0.5	1.5900	1.5	7.1100	2.5	9.0800	3.5	9.7900	4.5	9.99
0.75	2.3400	1.75	7.3600	2.75	9.2000	3.75	9.8100	4.75	10

Data in Table 3 is used for transfer function estimation. Transfer function estimation obtained using MATLAB, through the identification tool application. The model used is a process model for a second order system, without delay and overdamped. The transfer function estimation results are:

$$Estimated\ transfer\ function = \frac{0.676}{0.5833s^2 + 2.629s + 1} \tag{3}$$

PID Controller Implementation

Open-loop System

In the SUMO robot, the robot moves according to the motor actions and the motor moves according to the implemented sensors as shown in Figure 22.

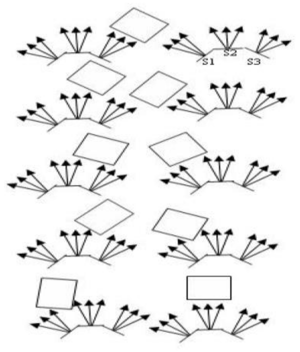


Figure 2: Robot sensors.

According to the sensor readings there are many cases for the speed and direction of the robot, in the following Table 4 we considered all the possible cases.

Table 4: Direction Cases

Case	Left sensor (L)	Center sensor (C)	Right sensor (R)	Output of left motor	Speed (RPM)	Output of Right motor	Speed (RPM)
1	0	0	0	0	0	0	0
2	0	0	1	1	50	0	0
3	0	1	0	1	100	1	100
4	0	1	1	1	75	0	0
5	1	0	0	0	0	1	50
6	1	0	1	1	50	1	50
7	1	1	0	0	0	1	75
8	1	1	1	1	75	1	75

These are some examples who the robot will move:

- If the output of the three sensors is (001) the robot must rotate to the right by make the left motor move forward with speed 50RPM and the left motor stop.
- If the output of the three sensors is (010) the robot must move forward by make the left and right motors move forward with speed 100RPM (full speed).

- Cases 6 and 8 are not logical so if one of them occurs, the robot will move forward with 50RPM or 75RPM.

We implement all cases using logical expressions and represent it in MATLAB.

Table 5: Logical expressions and representations.

Speed (RPM)	Logical expression	Logical expression
50	C'.R	L.C'
75	C.R	L.C
100	L'.C.R'	L'.C.R'

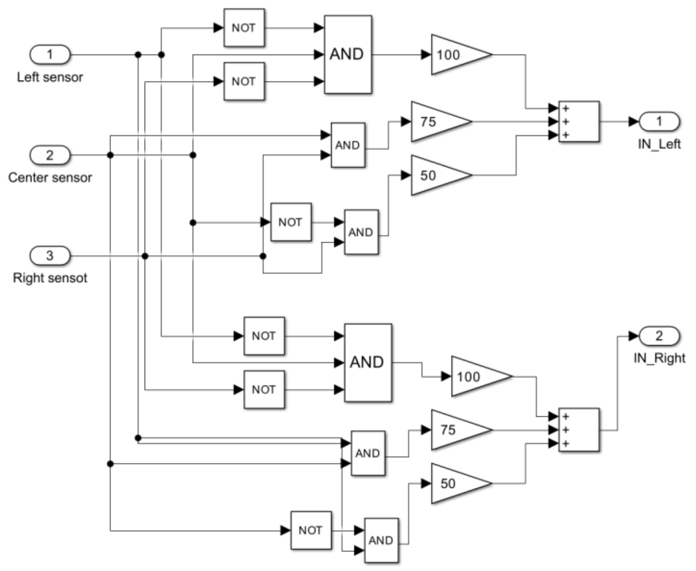


Figure 3:Sensors representation as logical gates using MATLAB.

We use pulse generators in MATLAB to get all possible cases:
We use 10 sec between every two changes to monitor the stability of the controlled signal as shown in Figure 4.

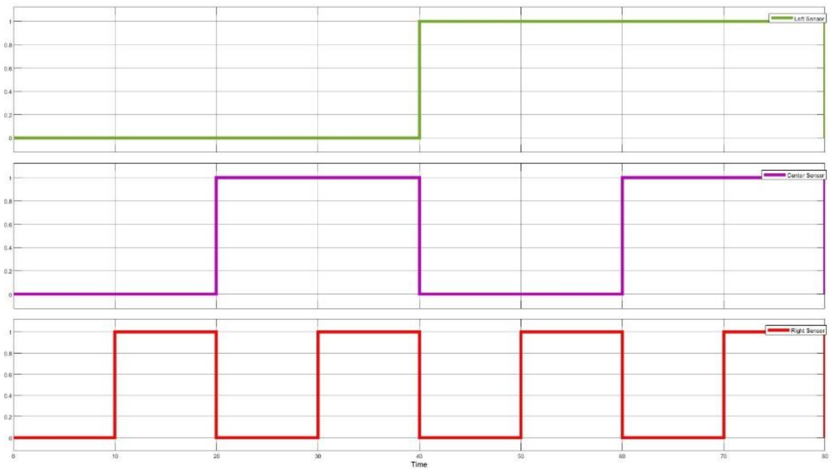


Figure 4: Pulse signal.

We represent it in MATLAB using subsystem and the output of the subsystem is the required speed that will set the moving direction of the robot.

Then we multiply it with step input to make the final value of the step equal to the required speed.

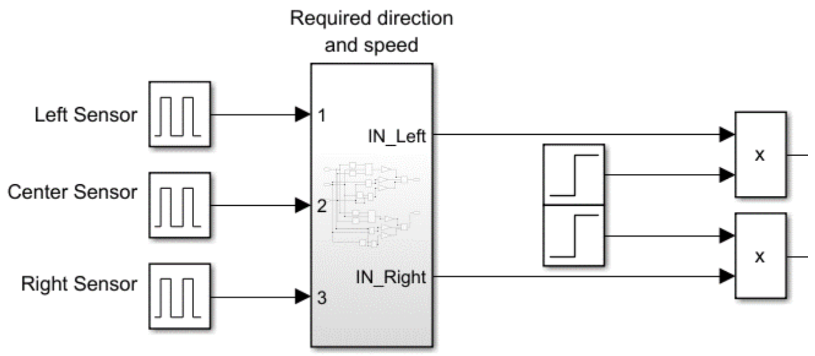


Figure 5: Simulink representation for the subsystem

Shown in Figure 6, the system simulation without implementing the PID controllers.

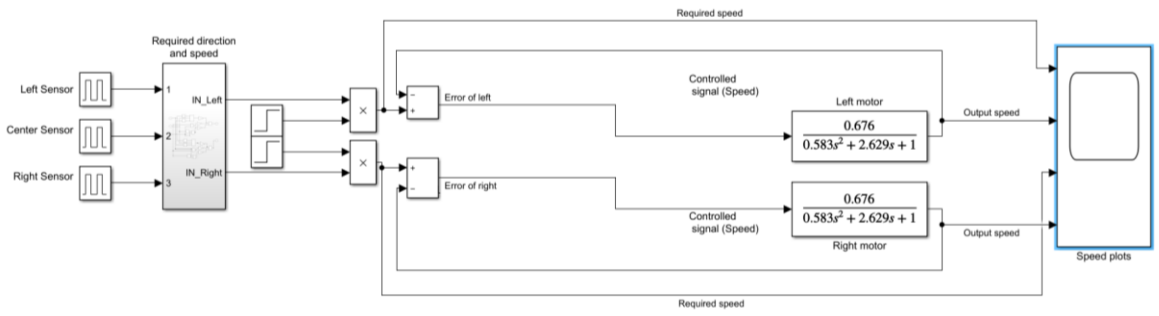


Figure 6: Schematic diagram for the system

Shown in Figure 7 and Figure 8 the system response for both the left and right motor.

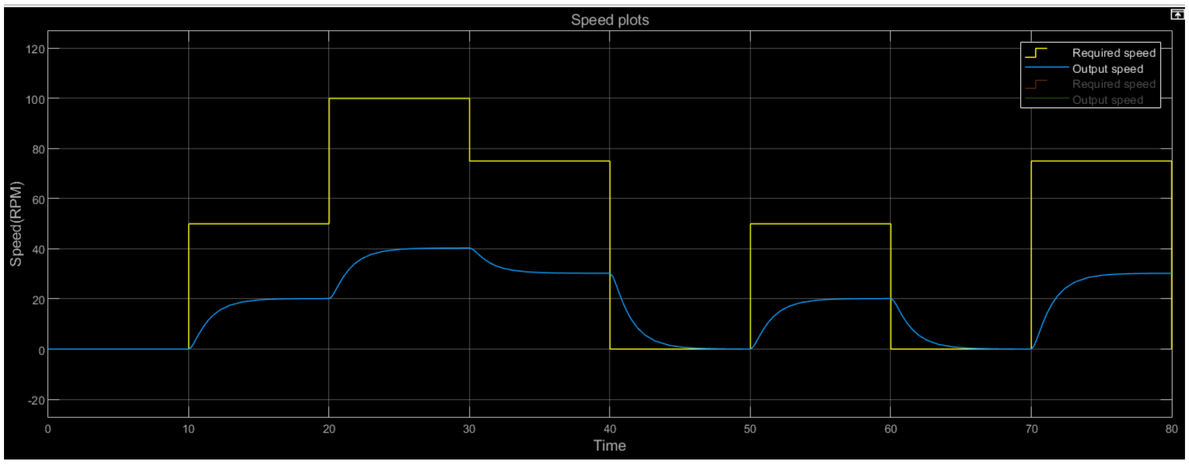


Figure 7: The system response for left motor

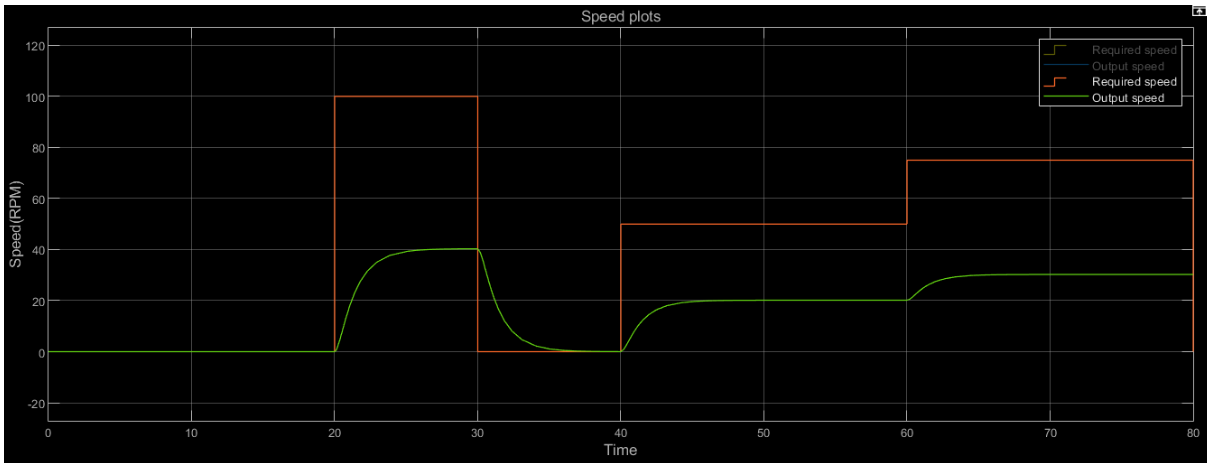


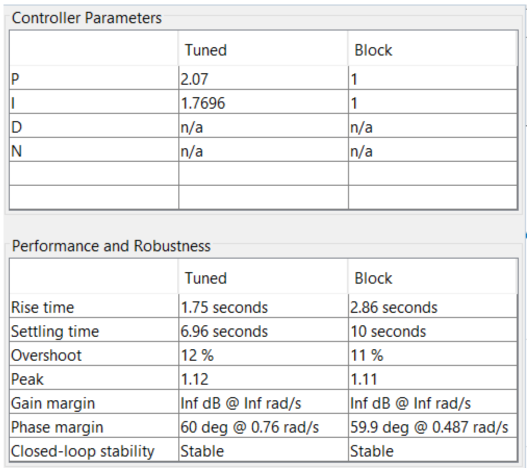
Figure 8: The system response for right motor

Shown in Figure 7 & Figure 8, that the response does not match the desire signal; in fact, it is far away from it so to solve that problem we try to implemented a PID controllers.:

Implementing PI Controller

PI controllers are an excellent choice for applications where:

- Steady-state accuracy is paramount: for systems requiring precise control at the final value, PI's ability to eliminate residual error shines.
- Smooth operation is desired: if minimizing overshoot and ensuring a stable transition are crucial, PI's controlled approach excels.



Controller Parameters

	Tuned	Block
P	2.07	1
I	1.7696	1
D	n/a	n/a
N	n/a	n/a

Performance and Robustness

	Tuned	Block
Rise time	1.75 seconds	2.86 seconds
Settling time	6.96 seconds	10 seconds
Overshoot	12 %	11 %
Peak	1.12	1.11
Gain margin	Inf dB @ Inf rad/s	Inf dB @ Inf rad/s
Phase margin	60 deg @ 0.76 rad/s	59.9 deg @ 0.487 rad/s
Closed-loop stability	Stable	Stable

Figure 9: PI parameters.

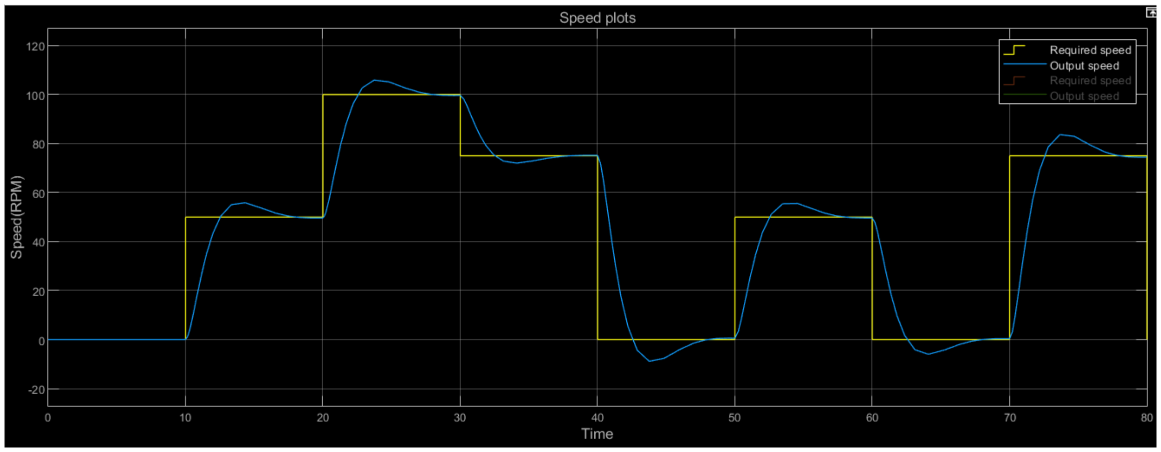


Figure 10: PI controller (left motor)

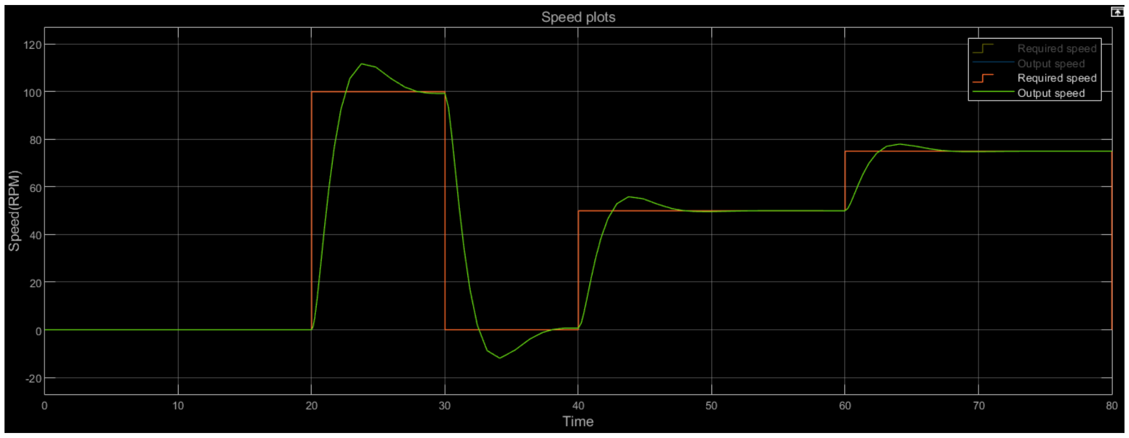


Figure 11: PI controller (right motor)

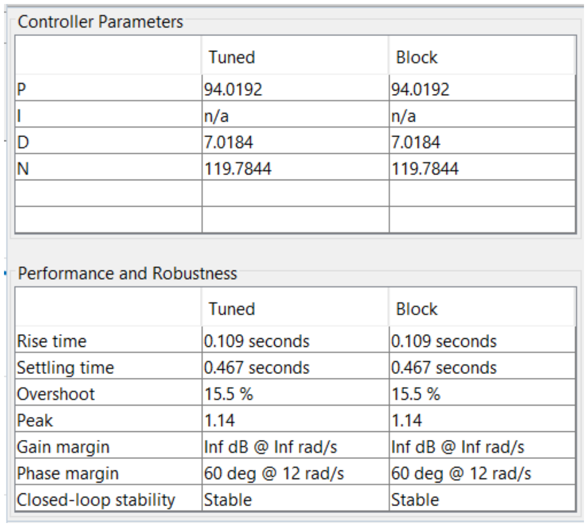
It's important to remember that no controller is perfect, and PI comes with its own set of trade-offs:

- Slower Initial Response: Compared to more aggressive controllers like PD, PI might take slightly longer to react to errors, especially during the initial transient phase.
- Sensitivity to Tuning: The integral term's tuning parameter can be sensitive, and improper adjustments can lead to sluggish or even unstable behavior.

Implementing PD Controller

PD controllers are an optimal choice for applications where:

- Transient response is critical: the focus on damping out oscillations makes them well-suited for applications that prioritize a swift transition to desired values.
- Overshoot mitigation is a priority: PD controllers are adept at minimizing overshoot and ensuring a controlled response during dynamic processes.
- Improved stability during dynamic changes: the proportional and derivative components work together to provide a balanced response.



Controller Parameters

	Tuned	Block
P	94.0192	94.0192
I	n/a	n/a
D	7.0184	7.0184
N	119.7844	119.7844

Performance and Robustness

	Tuned	Block
Rise time	0.109 seconds	0.109 seconds
Settling time	0.467 seconds	0.467 seconds
Overshoot	15.5 %	15.5 %
Peak	1.14	1.14
Gain margin	Inf dB @ Inf rad/s	Inf dB @ Inf rad/s
Phase margin	60 deg @ 12 rad/s	60 deg @ 12 rad/s
Closed-loop stability	Stable	Stable

Figure 12: PD parameters.

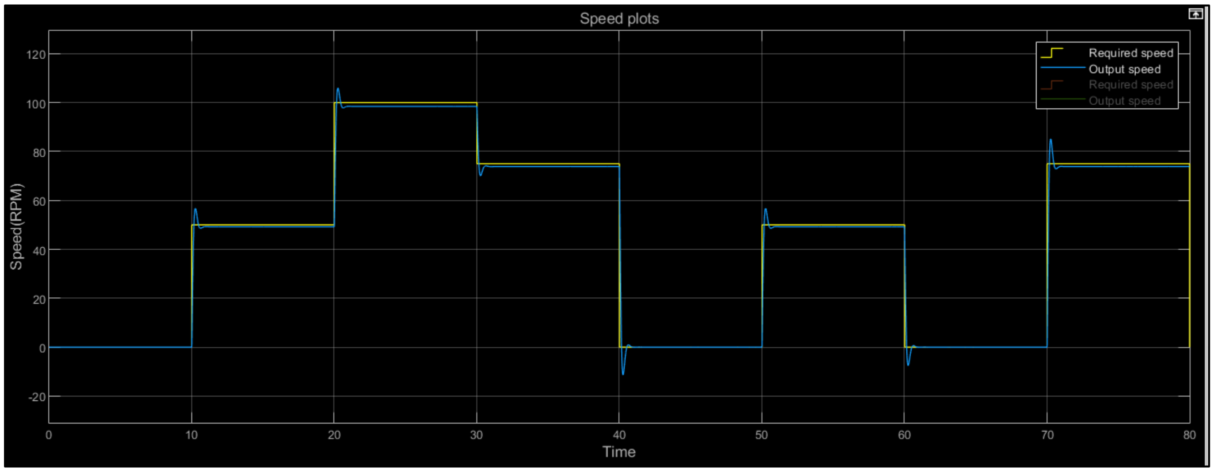


Figure 13: PD controller (left motor)

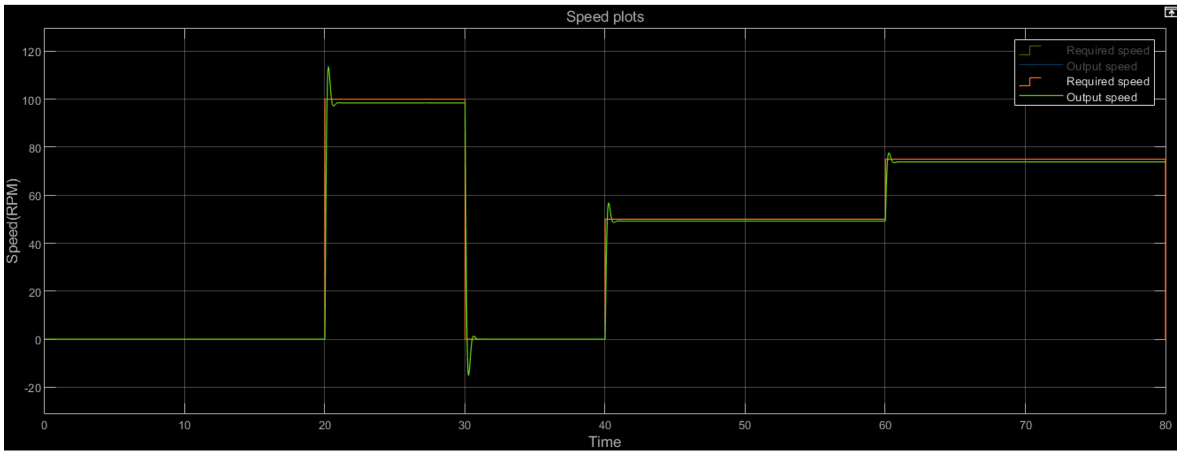


Figure 14: PD controller (right motor)

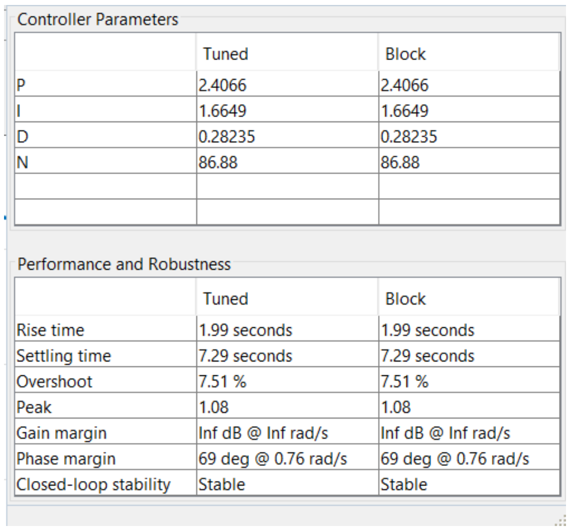
PD controllers, while effective in various applications, come with their own set of trade-offs:

- Potential for Increased Noise Amplification: PD controllers can be more sensitive to high-frequency noise in the system.
- Limited Steady-State Accuracy: Unlike PI controllers, PD controllers may exhibit limitations in achieving precise steady-state accuracy.
- Challenging Tuning: Tuning a PD controller can be challenging, especially in systems with complex dynamics.

Implementing PID Controller

PID controllers are well-suited for applications where:

- Steady-state accuracy is paramount: PID controllers excel in systems that demand precise control at the final value.
- Smooth operation is desired: PID controllers are effective in minimizing overshoot and ensuring a stable transition.
- Rapid response is essential: PID controllers achieve a faster response to errors during the transient phase is crucial.



Controller Parameters

	Tuned	Block
P	2.4066	2.4066
I	1.6649	1.6649
D	0.28235	0.28235
N	86.88	86.88

Performance and Robustness

	Tuned	Block
Rise time	1.99 seconds	1.99 seconds
Settling time	7.29 seconds	7.29 seconds
Overshoot	7.51 %	7.51 %
Peak	1.08	1.08
Gain margin	Inf dB @ Inf rad/s	Inf dB @ Inf rad/s
Phase margin	69 deg @ 0.76 rad/s	69 deg @ 0.76 rad/s
Closed-loop stability	Stable	Stable

Figure 15: PID parameters.

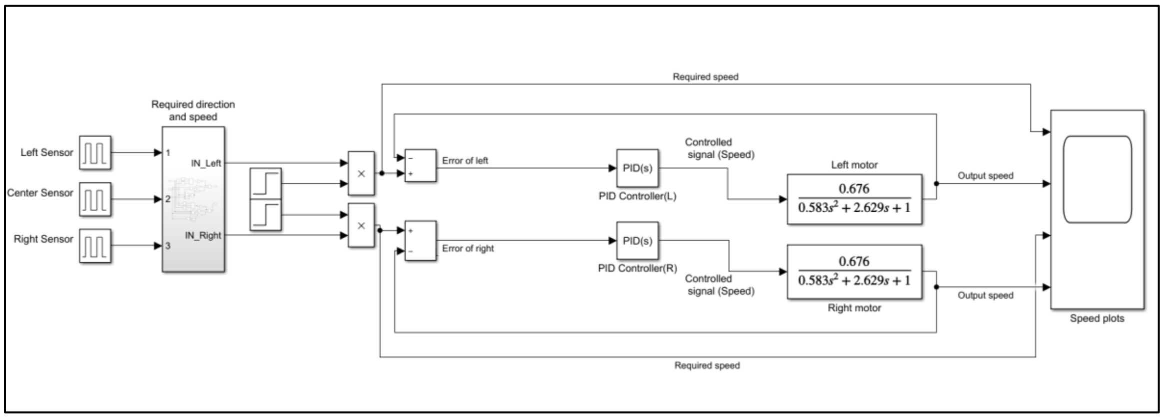


Figure 16: Schematic diagram for the system with PID controllers

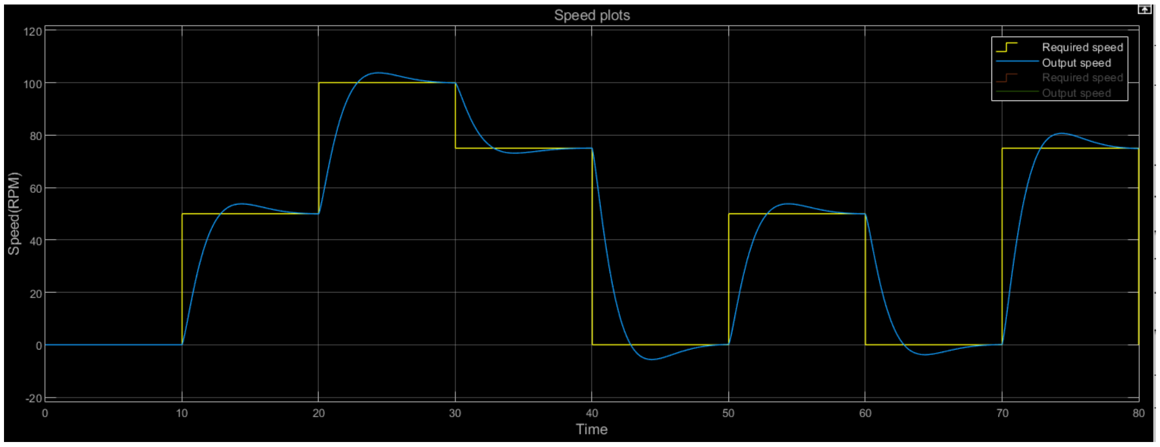


Figure 17: PID controller (left motor)

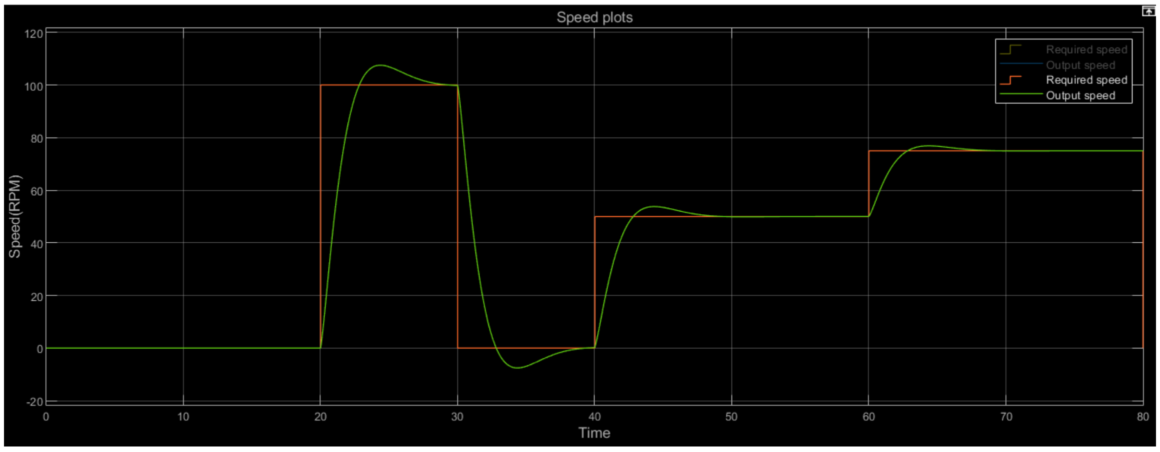


Figure 18: PID controller (right motor)

Implementing PID Controller using C++

Implementing a Proportional-Integral-Derivative (PID) controller using C++ involves creating a program to autonomously regulate a system's behavior. The PID controller, a cornerstone in control systems, utilizes three components—Proportional (P), Integral (I), and Derivative (D)—to continuously adjust the control input based on the error signal, representing the deviation between the desired setpoint and the current system output.

In C++, the process begins by defining variables to store critical parameters such as error, integral, derivative terms, and the PID output. The gains (proportional, integral, and derivative) are initialized, and the program continuously reads the system's output, calculates the error, and computes the PID output using a predetermined formula.

The controller's output is then applied to the system, and the process iterates in a loop. Tuning gains, implementing anti-windup mechanisms, and ensuring safety checks are essential considerations. By combining these elements, a well-implemented PID controller in C++ provides a robust method for maintaining desired setpoints in various dynamic systems, from simple motor control to complex industrial processes.

Analyze the PID controller's performance

In this study, we explored how different types of controllers—PI, PD, and PID—affect the performance of a simulated system. As expected, each controller (like different dance moves) had its own impact on important measures such as how quickly the system settles, how fast it rises, and the maximum amount it overshoots. Let's take a closer look at how each controller performed:

PI Controller: known for its steady-state accuracy and smooth error reduction, exhibited a respectable settling time of 0.467 seconds. However, its initial response was slightly sluggish compared to the nimble PD controller.

PD Controller: The PD controller, known for its agility in responding quickly, demonstrated exceptional speed by achieving a settling time of 0.109 seconds. Its proportional action promptly corrected errors, and the derivative term functioned as a predictive mechanism, anticipating and addressing future changes preemptively. Nevertheless, this rapid response had a drawback—a modest but discernible overshoot of 15.5%, suggesting a degree of initial overeagerness in the controller's output.

PID Controller: The PID controller, recognized for its versatility in combining speed and stability, amalgamated the attributes of PI and PD to attain a harmonized performance. The PID controller tuned in C++ exhibited outstanding stability, reducing overshoot to a minimal 1.214%, showcasing its precise control capabilities. However, this precision was accompanied by a trade-off in terms of speed, as evidenced by a settling time of 10.009 seconds, marking the slowest among the considered contenders.

Table 6: System analysis.

Sr No	Parameters	Without controller simulator	Simulated value PI	Simulated value PD	Simulated value PID	Tuning with C++
1	Settling Time	0.978	6.96 sec	0.467	7.29 sec	10.009
2	Rise Time	0.4401	1.75 sec	0.109	1.99 sec	10.009
3	MP	0	12%	15.5%	7.51%	1.214%
4	Ess (AT 100 RPM)	59.2	0.765	1.6	0.14	4.9

Steady-State Error (ess):

- Significant Reduction with PI and PID: As evident in the table, both PI and PID controllers effectively reduced ess compared to the uncontrolled system. PI achieved an ess of 0.765, while PID tuning with C++ resulted in an even lower ess of 0.14. This demonstrates their ability to drive the system to its desired steady-state value accurately.
- PD Less Effective in Reducing ess: In contrast, the PD controller, despite its swift response, exhibited a higher ess of 1.6, suggesting that it might not be the optimal choice for applications where minimizing steady-state error is critical.

Neural Network Controller Design and Implementation

ANN Implementation

Using our previous design in MATLAB we replace the PID controller with ANN as shown in Figure 19.

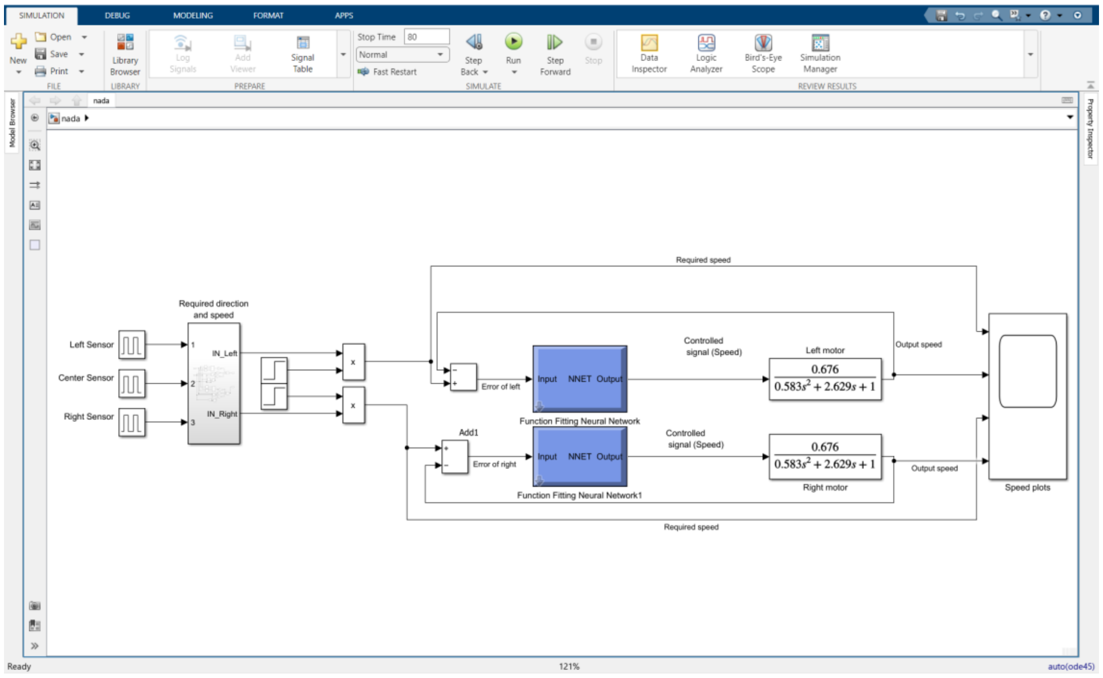


Figure 19: Schematic diagram for the system with ANN

By directly learning the relationship between control error and system output, ANNs offer the potential for data-driven, self-adaptive control with improved accuracy and robustness.

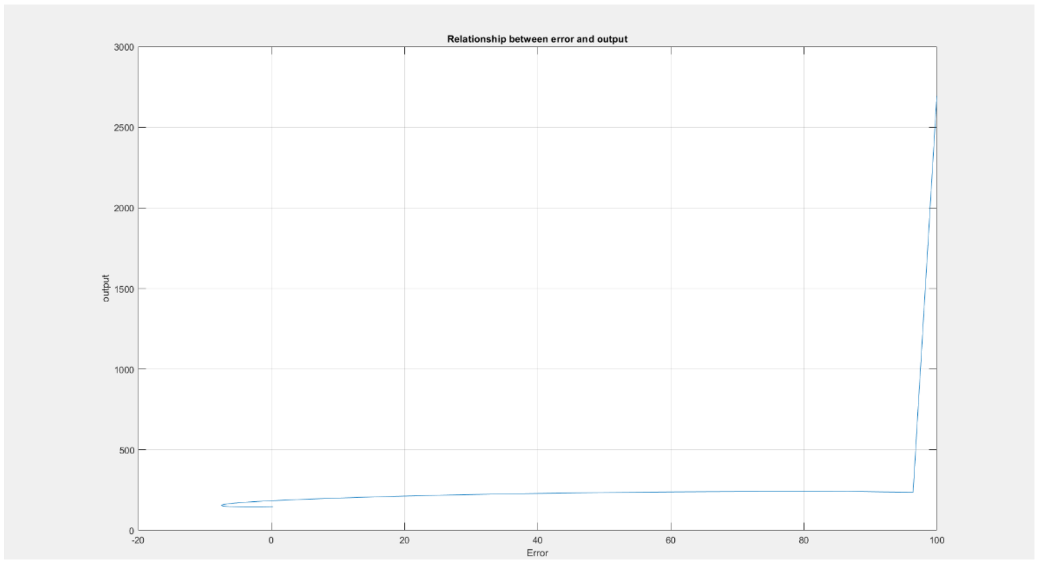


Figure 20: Relation between error and output

Model Performance and Training Convergence

The trained ANN demonstrated efficient learning of the input-output relationship, achieving a best validation performance of 9.3907 Mean Squared Error (MSE) at epoch 84, as depicted in the graph below. Each epoch represents a complete pass through the training dataset. As evident from the graph, the model progressively improves its performance with increasing epochs, as shown by the downward trend of the blue line representing the training MSE. Notably, the validation MSE (red line) closely follows the training MSE, indicating minimal overfitting and good generalization to unseen data. This convergence of training and validation performance at epoch 84 suggests that further training might not yield significant benefits, implying an optimal stopping point for the training process.

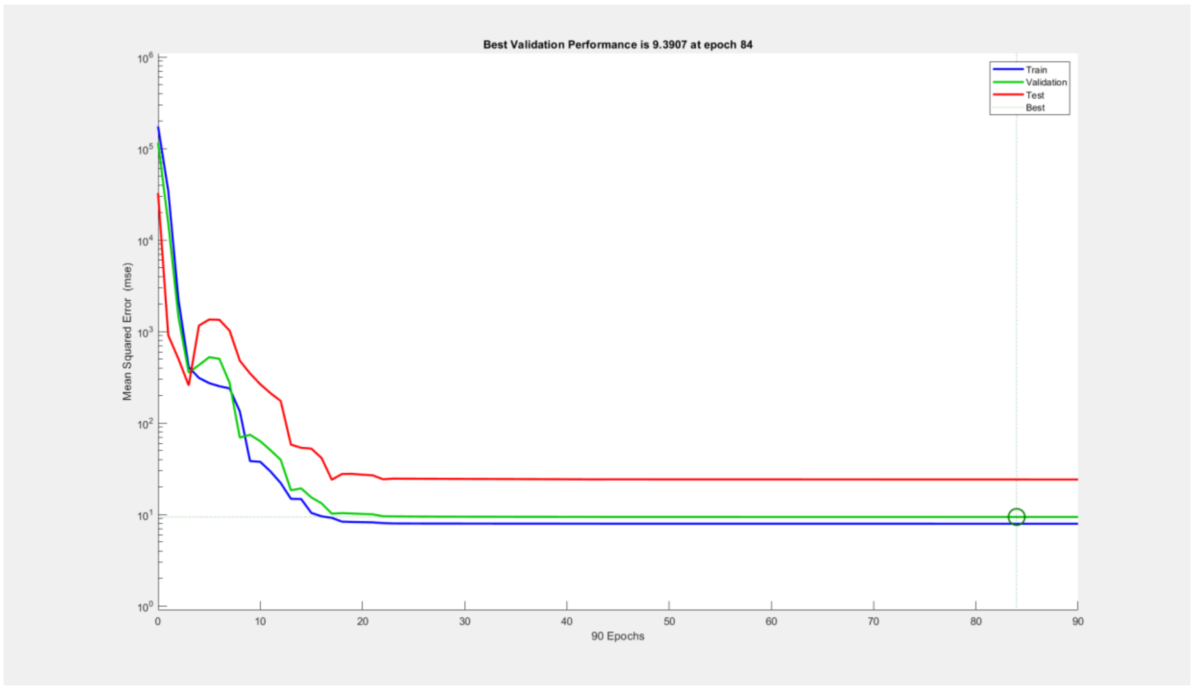


Figure 21: Performance of ANN

Analysis of ANN Training Progress

The four presented graphs provide valuable insights into the learning process of the trained ANN model. Each graph depicts the relationship between the number of training epochs and various performance metrics, offering a multifaceted understanding of the model's optimization and generalization capabilities.

Target-Specific Learning

First image focuses on a single target feature, showcasing a notable decrease in Mean Squared Error (MSE) as epochs increase. This demonstrates the progressive refinement of the model's ability to predict this specific feature with increasing exposure to training data. The eventual plateauing of the MSE curve suggests convergence towards an optimal learning state for this particular target.

Comparative Feature Learning

Second image expands the analysis to multiple target features, revealing their differential learning rates. While all features exhibit decreasing MSE with further training, some (represented by distinct color lines) demonstrate faster convergence than others. This implies inherent disparities in their complexity or relative ease of extraction from the training data.

Overall Model Performance

Third image offers a holistic perspective by plotting the average MSE across all target features. Like individual target analyses, we observe a general downward trend, confirming the model's overall improvement in performance with more training. However, the curve might not be as smooth as individual MSEs, reflecting the differences in feature complexity and their impact on learning rates.

Prediction Accuracy and Generalization

Fourth image adopts a different metric, analyzing the correlation between the ANN's predicted values and the actual values for each target feature. Ideally, we seek a correlation approaching 1 as training progresses, indicating enhanced accuracy and generalization across all features.

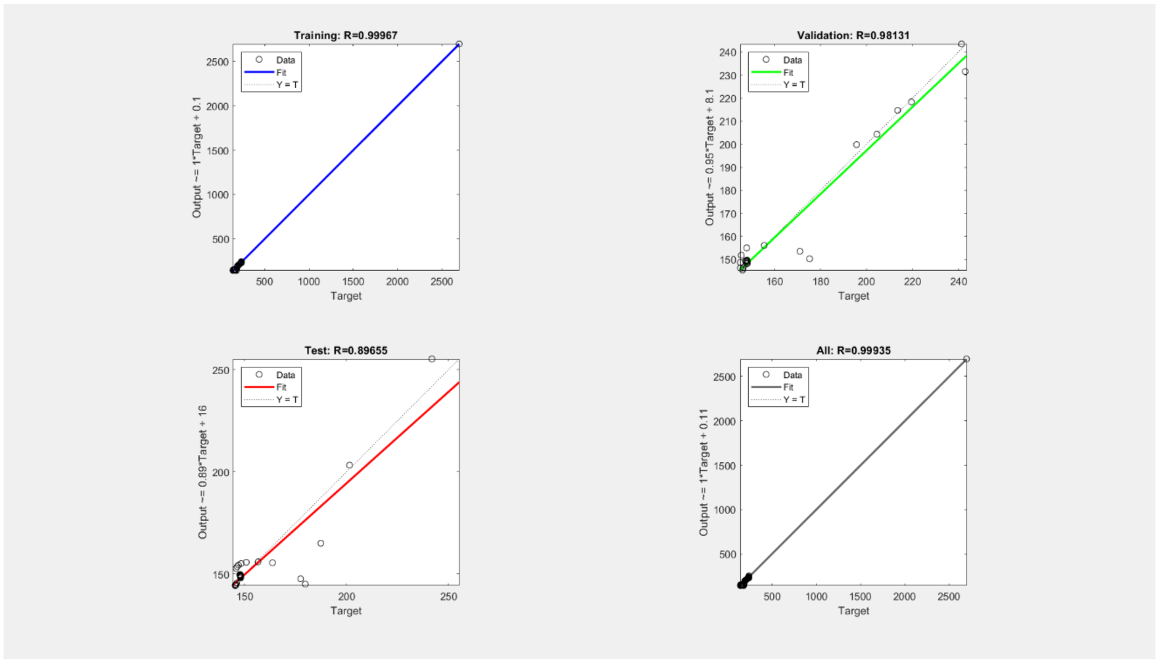


Figure 22: Regression of ANN

In & show the response of the system after implementing the ANN.

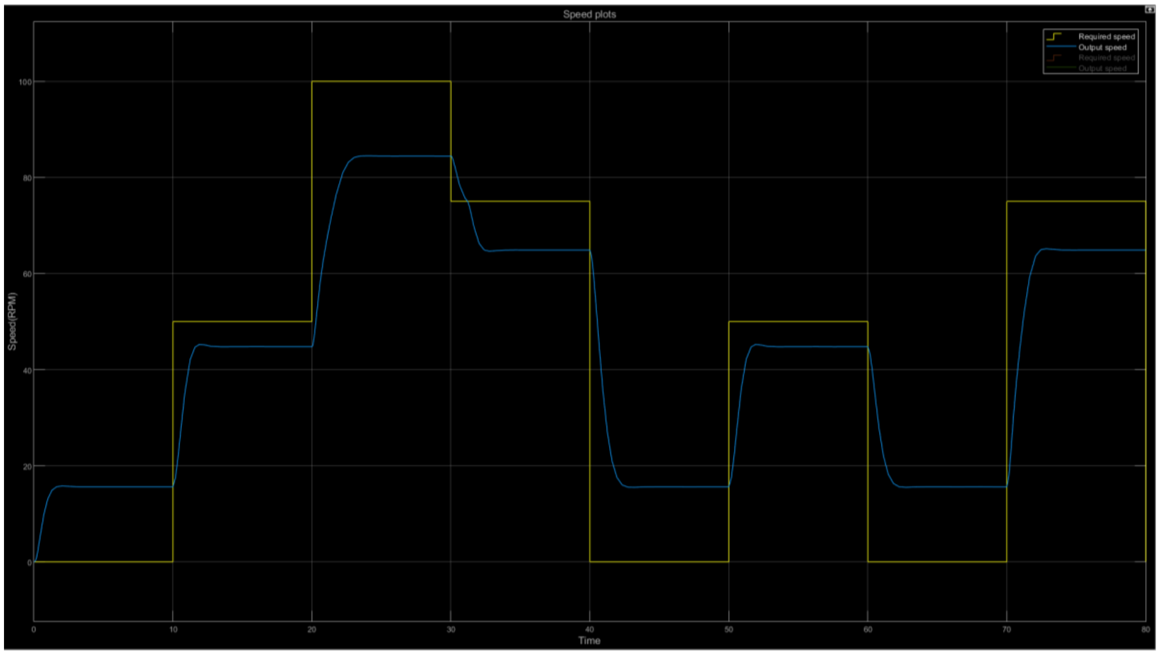


Figure 23: ANN controller (right motor)

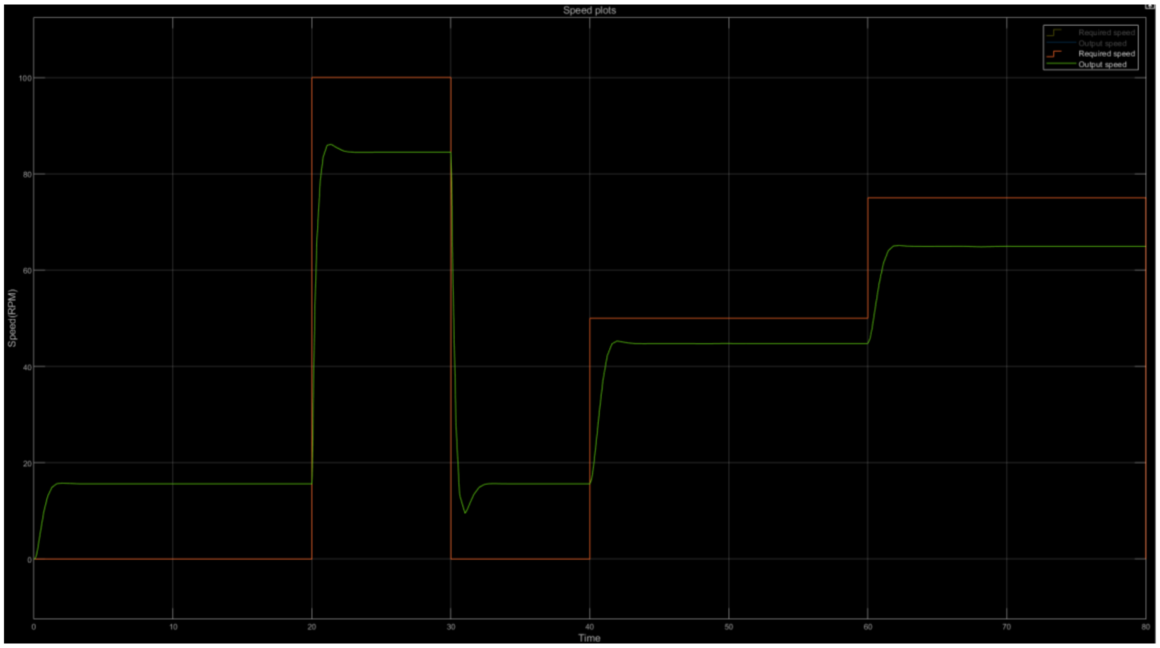


Figure 24: ANN controller (right motor)

Performance Evaluation and Comparison

Selecting the optimal control strategy for our sumo robot requires a careful assessment of system characteristics and performance objectives. In this report, we explore the merits of two dominant approaches: Proportional-Integral-Derivative (PID) control and Artificial Neural Networks (ANNs).

PID: A Legacy of Stability and Simplicity

PID control boasts a longstanding reputation for reliability and ease of implementation. Its transparent operation and well-established tuning techniques make it ideal for linear systems like our sumo robot, where predictable relationships between inputs and outputs prevail.

PID Strengths:

Simplicity: Intuitive framework readily implemented and adjusted.

Fast Response: Reacts promptly to sensor data changes, ensuring immediate adjustments.

Robustness: Handles anticipated disturbances effectively within stable environments.

PID Weaknesses:

Limited Adaptability: Struggles with complex non-linearities or dynamic situations.

Tuning Challenges: Finding optimal PID parameters can be time-consuming and iterative.

Disturbance Sensitivity: Unforeseen factors like friction variations or opponent maneuvers can affect performance.

ANN: Rising to the Challenge of Complexity

ANNs offer a data-driven approach, capable of learning from experiences and adapting to unforeseen scenarios. Their flexibility makes them attractive for handling non-linearities and dynamic environments, potentially surpassing PID's capabilities in challenging situations.

ANN Strengths:

Adaptability: Learns from data, adjusting control strategies to handle diverse situations.

Robustness: Demonstrates superior resilience to disturbances and unexpected changes.

Optimization Potential: Offers extensive customization options for fine-tuning performance.

ANN Weaknesses:

Complexity: Requires substantial data and computational resources for training and operation.

Training Time: Learning process can be time-consuming, affecting implementation timelines.

Interpretability: Understanding the intricate decision-making processes within an ANN can be challenging.

Matching Your Champion to the Ring:

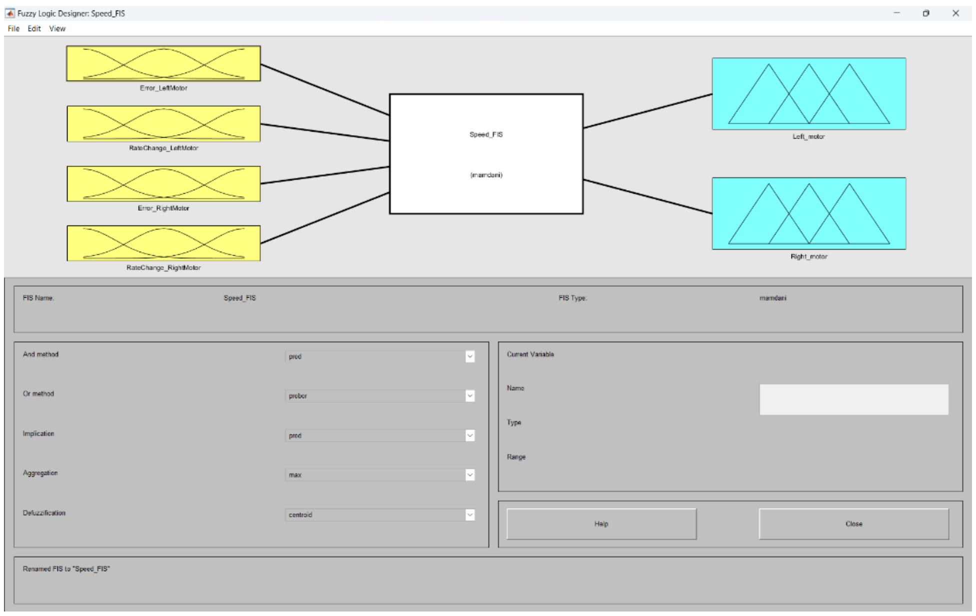
Considering our current sumo robot, utilizing a PID controller initially appears advantageous. Its simplicity, fast response, and robustness align well with the predictable environment and linear system characteristics. However, as we introduce increased complexity with varying terrain, unpredictable obstacles, or evolving opponent strategies, the adaptability and robustness of ANNs might become invaluable

Fuzzy Controller Design and Implementation

Simulink model and responses for the robot with fuzzy controlling

We use rate change of the output, because we want to control the system better when the fuzzy deal with the error as zero and control the fast spikes that happen. Also, to monitor the changing of the output at any spike of error when the required speed changing suddenly.

The tunning of the fuzzy for the left motor and the right motor are the same.



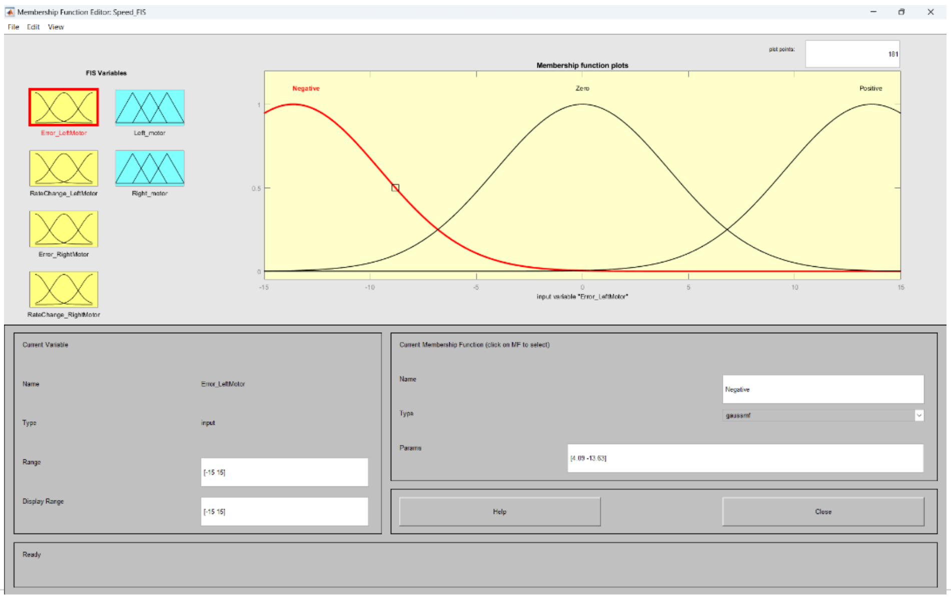
FIS Name: Speed_FIS FIS Type: mamdani

And method: prod
Or method: probor
Implication: prod
Aggregation: max
Defuzzification: centroid

Current Variable
Name
Type
Range

Renamed FIS to "Speed_FIS"

Figure 25: FUZZY inputs and outputs



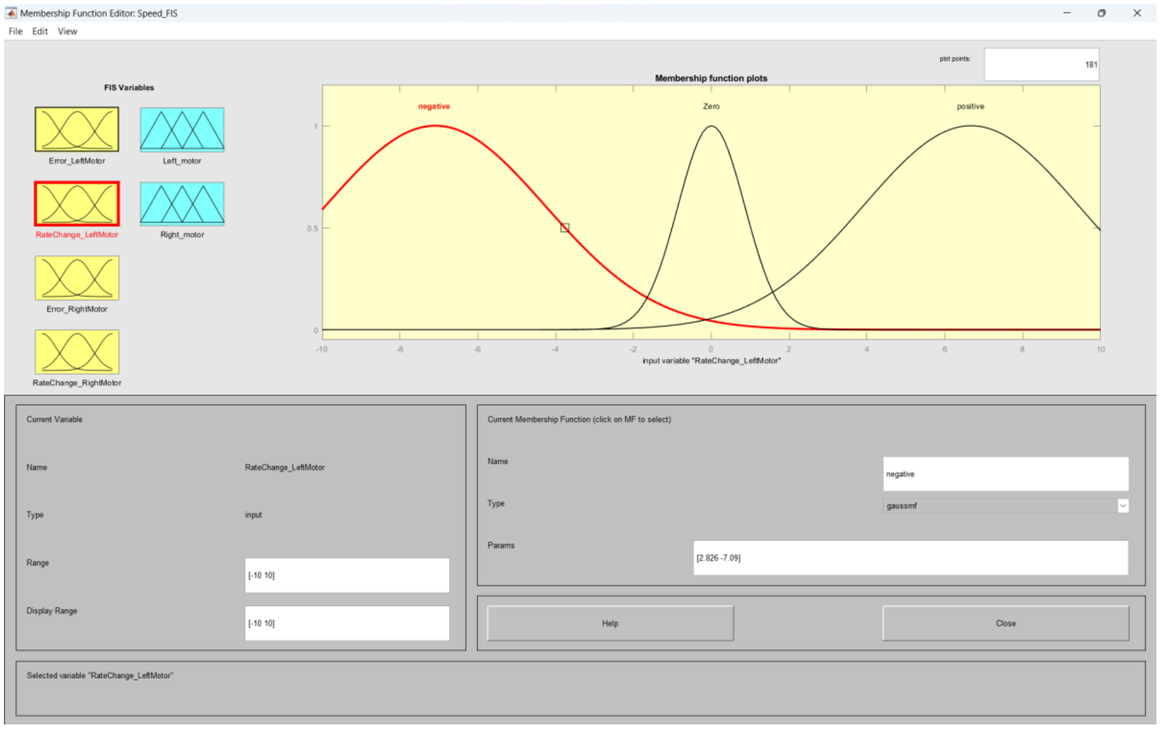
plot points: 181

Current Variable
Name: Error_LeftMotor
Type: input
Range: [-15 15]
Display Range: [-15 15]

Current Membership Function (click on MF to select)
Name: Negative
Type: gaussmf
Params: [4.09 -13.63]

Ready

Figure 26: Membership function of error of left motor



Current Variable

Name RateChange_LeftMotor

Type input

Range [-10 10]

Display Range [-10 10]

Current Membership Function (click on MF to select)

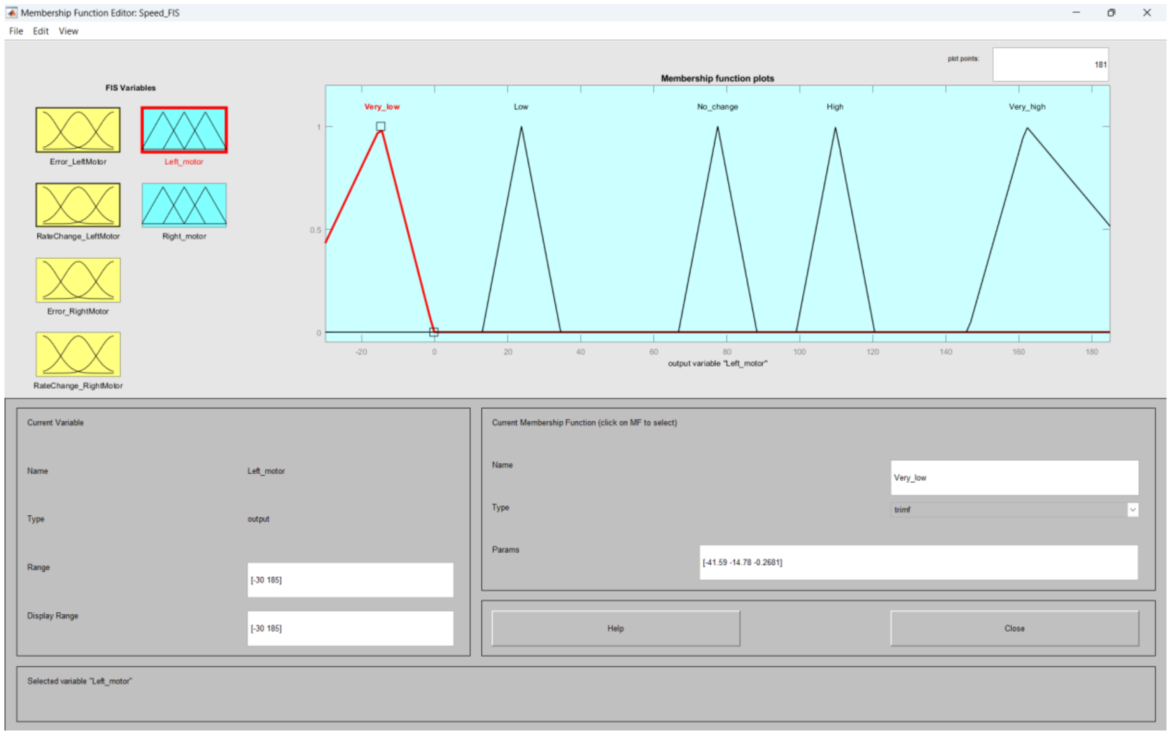
Name negative

Type gaussmf

Params [2.826 -7.09]

Selected variable "RateChange_LeftMotor"

Figure 27: Membership function of rate change of left motor



Current Variable

Name Left_motor

Type output

Range [-30 185]

Display Range [-30 185]

Current Membership Function (click on MF to select)

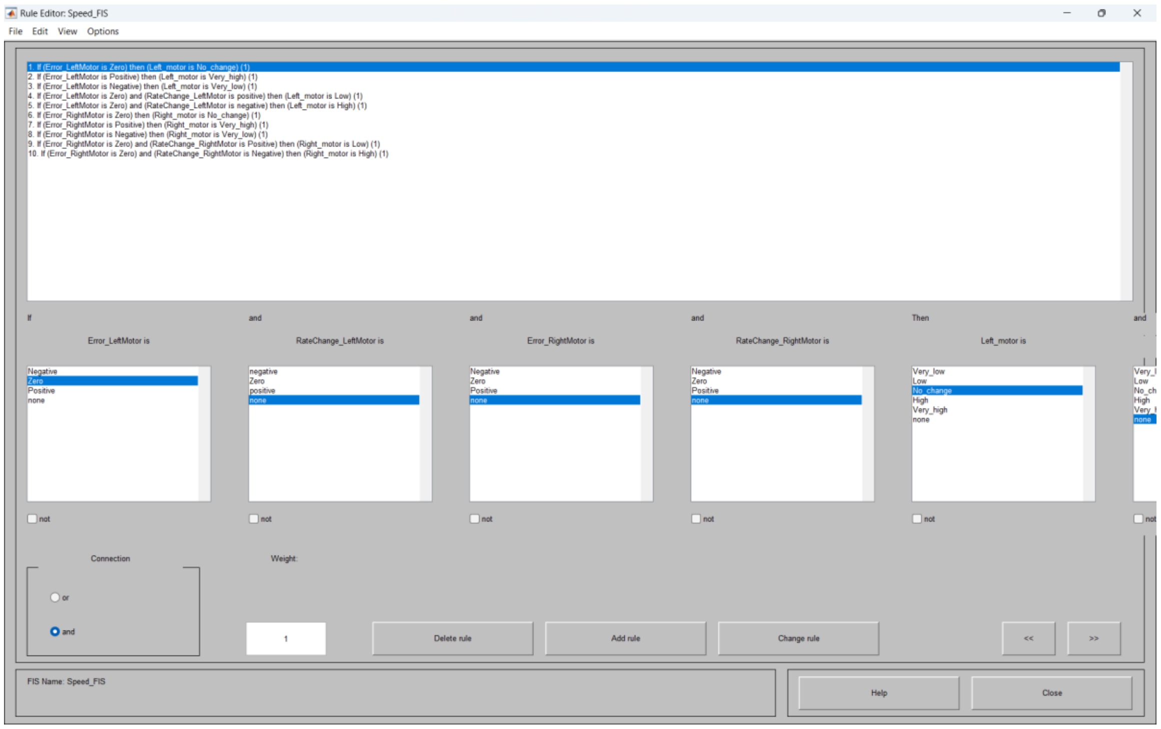
Name Very_low

Type trimf

Params [-41.59 -14.78 -0.2681]

Selected variable "Left_motor"

Figure 28: Membership function of speed of left motor



```
1. If (Error_LeftMotor is Zero) then (Left_motor is No_change) (1)
2. If (Error_LeftMotor is Positive) then (Left_motor is Very_high) (1)
3. If (Error_LeftMotor is Negative) then (Left_motor is Very_low) (1)
4. If (Error_LeftMotor is Zero) and (RateChange_LeftMotor is positive) then (Left_motor is Low) (1)
5. If (Error_LeftMotor is Zero) and (RateChange_LeftMotor is negative) then (Left_motor is High) (1)
6. If (Error_RightMotor is Zero) then (Right_motor is No_change) (1)
7. If (Error_RightMotor is Positive) then (Right_motor is Very_high) (1)
8. If (Error_RightMotor is Negative) then (Right_motor is Very_low) (1)
9. If (Error_RightMotor is Zero) and (RateChange_RightMotor is Positive) then (Right_motor is Low) (1)
10. If (Error_RightMotor is Zero) and (RateChange_RightMotor is Negative) then (Right_motor is High) (1)
```

Figure 29: FUZZY rules

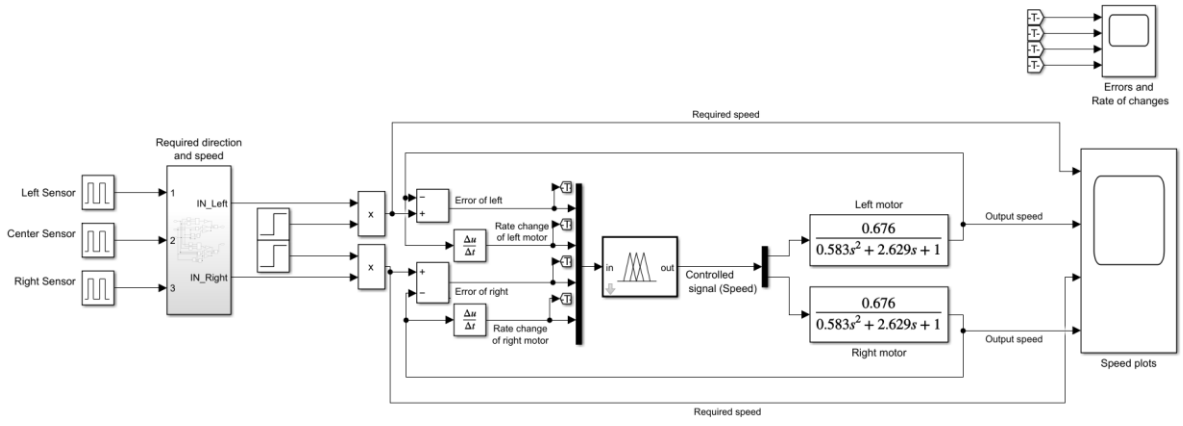

Errors and Rate of changes
Required speed
Required direction and speed
Left Sensor
Center Sensor
Right Sensor
IN_Left
IN_Right
Error of left
Rate change of left motor
Error of right
Rate change of right motor
in
out
Controlled signal (Speed)
Left motor
$\frac{0.676}{0.583s^2+2.629s+1}$
$\frac{0.676}{0.583s^2+2.629s+1}$
Right motor
Output speed
Output speed
Speed plots
Required speed

Figure 30: Schematic diagram for the system with Fuzzy.

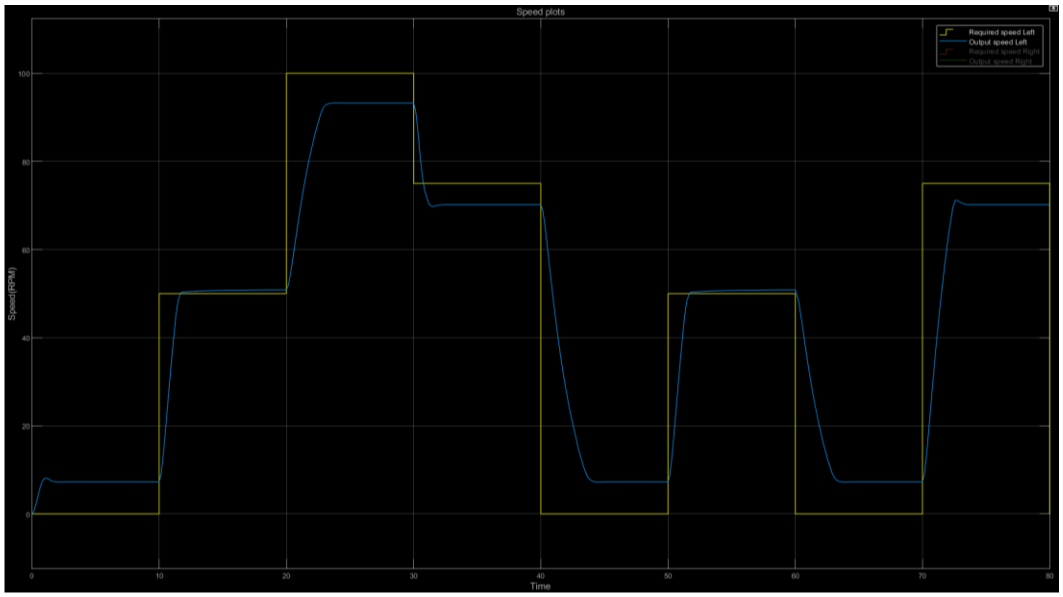


Figure 31:Response of using FUZZY on the left motor

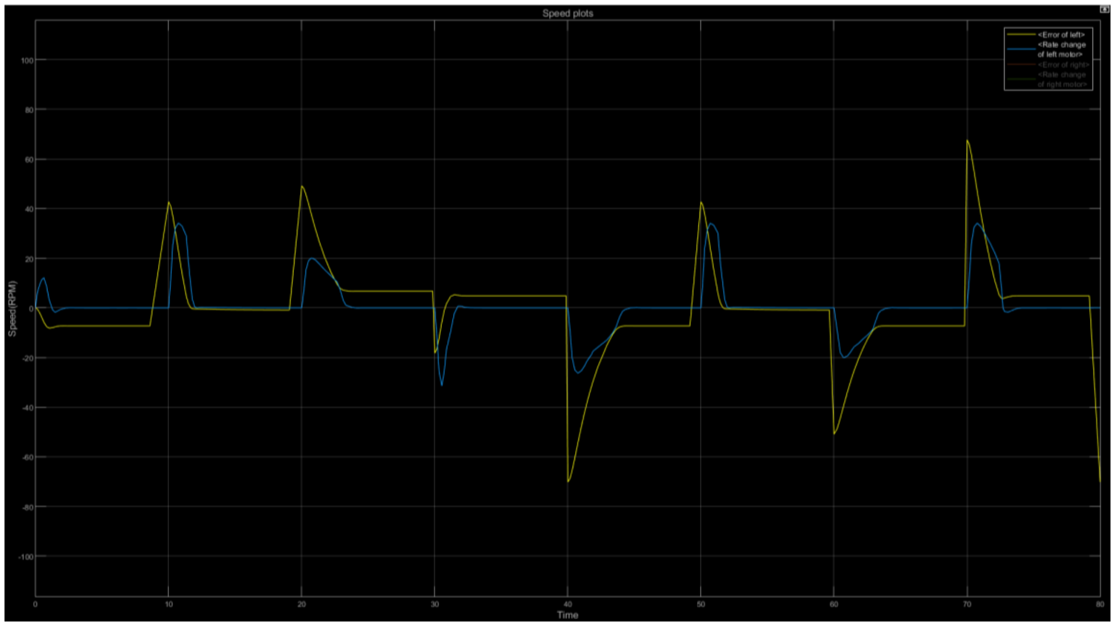


Figure 32: The error and rate change of the left motor

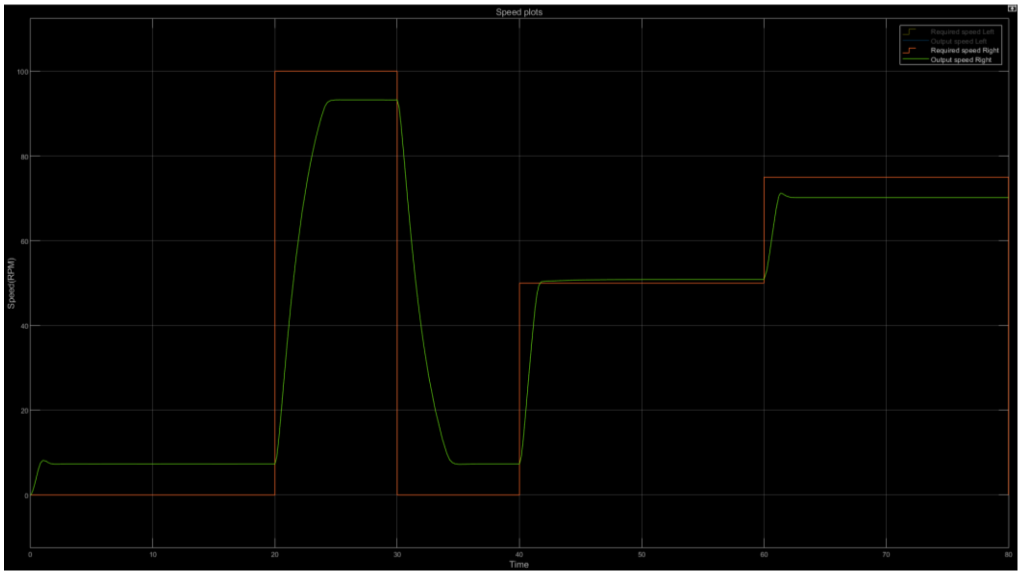


Figure 33: Response of using FUZZY on the right motor

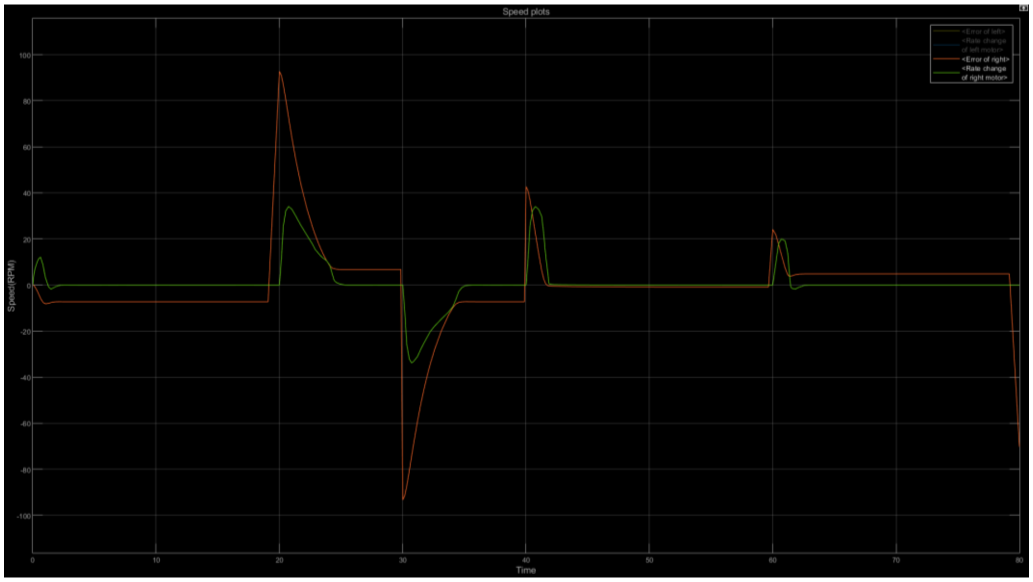


Figure 34: The error and rate change of the right motor

References

[1] P. Parikh, S. Sheth, R. Vasani, and J. K. Gohil, "Implementing fuzzy logic controller and PID controller to a DC encoder motor – 'A case of an automated guided vehicle,'" *Procedia Manufacturing*, vol. 20, pp. 219–226, 2018. doi:10.1016/j.promfg.2018.02.032

[2] R. Rudra and R. Banerjee, "Modeling and simulation of DC Motor Speed Regulation by field current control using MATLAB," *International Journal of Computer and Electrical Engineering*, vol. 9, no. 2, pp. 502–512, 2017. doi:10.17706/ijcee.2017.9.2.502-512

[3] S. Parasuraman, B. Shirinzadeh, and V. Ganapathy, "Sensors fusion technique for mobile robot navigation using Fuzzy Logic Control System," *Mobile Robots Navigation,* 2010. doi:10.5772/8986

[4] H. S. Dakheel, Z. B. Abdullah, N. S. Jasim, and S. W. Shneen, "Simulation model of ann and PID controller for direct current servo motor by using MATLAB/Simulink," *TELKOMNIKA (Telecommunication Computing Electronics and Control)*, vol. 20, no. 4, p. 922, 2022. doi:10.12928/telkomnika.v20i4.23248

[5] S. A. Wahyu and D. P. Riky, "DC Motor Simulation Transfer Function Estimation: Case Study Proteus Ver. 7," IOP Conference Series: Materials Science and Engineering, vol. 674, no. 1, p. 012040, 2019. doi:10.1088/1757-899x/674/1/012040

[6] H. Erdem, "A practical fuzzy logic controller for Sumo Robot Competition," *Lecture Notes in Computer Science*, pp. 217–225. doi:10.1007/978-3-540-77046-6_27

Appendix

CODE C++ (PID)

```
#include <iostream>

#include <chrono>

#include <thread>


class PIDController {

public:

    PIDController(double kp, double ki, double kd)

        : kp_(kp), ki_(ki), kd_(kd), integral_(0.0), prevError_(0.0),
startTime_(std::chrono::steady_clock::now()) {}


    double compute(double error) {

        integral_ += error;

        double derivative = error - prevError_;

        prevError_ = error;


        double output = kp_ * error + ki_ * integral_ + kd_ * derivative;

        return output;

    }

    // Function to get the elapsed time since the PID controller was created

    double elapsedTime() const {

        auto currentTime = std::chrono::steady_clock::now();

        return std::chrono::duration_cast<std::chrono::milliseconds>(currentTime -
startTime_).count() / 1000.0;

    }

    // Getters for PID gains

    double getKp() const { return kp_; }
    double getKi() const { return ki_; }
    double getKd() const { return kd_; }
private:
    double kp_;
    double ki_;
    double kd_;
    double integral_;
    double prevError_;
    std::chrono::steady_clock::time_point startTime_;
};
double calculateInput(int time) {
    return 45.0 + 0.1 * time;
}
int main() {
    double kp = 0.5;
    double ki = 0.01;
    double kd = 0.1;
    PIDController pid(kp, ki, kd);
    double setpoint = 50.0;
    double maxOvershoot = 0.0;
    double steadyStateError = 0.0;
    for (int time = 0; time < 100; ++time) {
        double input = calculateInput(time);
        double error = setpoint - input;
        double output = pid.compute(error);
        // Simulate applying the output (replace this with your actual process control)
        // For this example, just print the output
        std::cout << "Time: " << time << " Input: " << input << " Output: " << output
<< std::endl;
```

```
        // Check for overshoot
        if (error < 0 && output > maxOvershoot) {
            maxOvershoot = output;
        }
        // Check for steady-state error after 50 time steps
        if (time >= 50) {
            steadyStateError = error;
        }
        // Add a delay to simulate the passage of time
        std::this_thread::sleep_for(std::chrono::milliseconds(100));
    }
    // Calculate and print performance metrics
    double riseTime = pid.elapsedTime(); // Elapsed time to reach the setpoint
    double settlingTime = pid.elapsedTime(); // Elapsed time to settle within a certain
range of the setpoint
    std::cout << "Rise Time: " << riseTime << " seconds" << std::endl;
    std::cout << "Settling Time: " << settlingTime << " seconds" << std::endl;
    std::cout << "Overshoot: " << maxOvershoot << std::endl;
    std::cout << "Steady-State Error: " << steadyStateError << std::endl;
    // Print the final tuned values of kp, ki, and kd
    std::cout << "Final Tuned Values:" << std::endl;
    std::cout << "Kp: " << pid.getKp() << std::endl;
    std::cout << "Ki: " << pid.getKi() << std::endl;
    std::cout << "Kd: " << pid.getKd() << std::endl;
    return 0;
}
```

MATLAB code

```
% Numerator Coefficients

numerator_coeff = [0.676];

% Denominator Coefficients

denominator_coeff = [0.583, 2.629, 1.0];

% Create the transfer function

sys = tf(numerator_coeff, denominator_coeff);

% Find the poles

poles = roots(denominator_coeff);

% Extract real and imaginary parts of the poles

real_parts = real(poles);

% Calculate time constants

time_constants = 1 ./ real_parts;

% Calculate settling time, rise time, and overshoot

Ts = 4 * max(time_constants);

Tr = 1.8 * max(time_constants);

% Calculate overshoot

zeta = -real_parts(1) / abs(real_parts(1));

% For underdamped systems (0 < zeta < 1)

if zeta > 0 && zeta < 1

    % Calculate percent overshoot

    % Formula: %OS = exp((-zeta * pi) / sqrt(1 -
zeta^2)) * 100

    percent_overshoot = exp((-zeta * pi) / sqrt(1 -
zeta^2)) * 100;

else
```

```
    percent_overshoot = 0;
end
% Display the results
disp('Settling Time (Ts):');
disp(Ts)
disp('Rise Time (Tr):');
disp(Tr)
disp('Percent Overshoot (%OS):');
disp(percent_overshoot)
```