Implementation and Tuning of a Raspberry Pi-Based PID Controller for Precision DC Motor Speed Regulation

Dr.V.V.kulkarni
AISSMS COE, Pune
vvkulkarni@aissmscoe.com

Mayuri D. Shinde
AISSMS COE, Pune
123mdshinde@gmail.com

Asim A. Mujawar
AISSMS COE, Pune
Officialasim27@gmail.com

Sarvesh Madhukar Yadav
AISSMS COE, Pune
sarveshy.1718@gmail.com

***Abstract*— Here's what we did: we designed, modeled, simulated, and actually built a Proportional-Integral-Derivative (PID) controller to keep a Permanent Magnet DC (PMDC) motor spinning at just the right speed, all with a Raspberry Pi 4B running Python. We figured out the plant's transfer function using the motor's own equations, checked it against real-world measurements, and dialed in our PID gains ($K_p = 0.0914$, $K_i = 12.527$, $K_d = 9.5 \times 10^{-5}$) using Python's control library. That gave us sharp numbers: 11.5 ms rise time, 39.2 ms to settle, only 6.54% overshoot, absolutely no steady-state error, and a phase margin of 74.1°. Every target? Nailed. When we threw sudden load changes at the motor, the system cut speed swings from 36.3% (open-loop) to just 6.5%, snapping back to speed in under 40 ms. All six specs, met.**

Index Terms*— *PID controller, DC motor speed control, Raspberry Pi, Python, closed-loop control, transfer function, disturbance rejection, PWM, incremental encoder.

I. INTRODUCTION

If you care about robots, CNCs, or even simple conveyor drives, you know PMDC motor speed needs to be spot-on—any drift hurts the whole system. Open-loop just doesn't cut it when the motor gets hit with load changes; that's why we need closed-loop control.

We used the classic PID algorithm, which still rules industry for a reason. Proportional action means speed, integral wipes out steady-state errors, and derivative keeps the overshoot in check. Our project runs the whole works—encoder feedback, PID math, motor commands—all in Python on a Raspberry Pi 4B, no PC required. The Pi reads speed from an incremental encoder using GPIO interrupts, sends PID corrections straight to the motor driver (BTS7960) via hardware PWM, and knows exactly where the motor stands, every 10 ms.

What's special here? It's a full end-to-end demo: we developed the math, tuned everything in Python, and ran the final system on affordable hardware you can buy anywhere.

II. SYSTEM ARCHITECTURE

Everything's on the Raspberry Pi 4B. We wrote a multi-threaded Python script that sorts interrupts from the encoder (pigpio library), reads the potentiometer setpoint through an ADS1115 ADC over I²C, calculates PID output, sends hardware PWM to spin the BTS7960, and updates a live plot over HDMI. No need for any external computer.

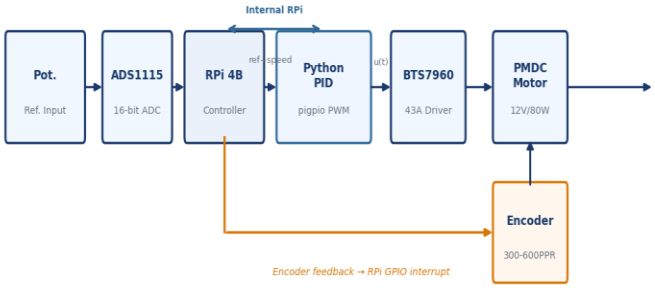

Fig. 1. Closed-loop system architecture. Raspberry Pi 4B runs the Python PID script; encoder provides feedback via GPIO interrupt.

Here's how the signals flow: you twist the potentiometer (0–3.3 V), ADS1115 digitizes it, Raspberry Pi gets the new target speed, PID does its thing, pigpio spits out PWM, BTS7960 drives the motor, encoder senses actual speed and signals the Pi, looping every 10 ms.

III. MATHEMATICAL MODELLING

A. Electrical Sub-System

Applying Kirchhoff's Voltage Law to the armature circuit (Fig. 2):

$$V(t) = L\cdot(di/dt) + R\cdot i(t) + K_e\cdot\omega(t) \quad (1)$$

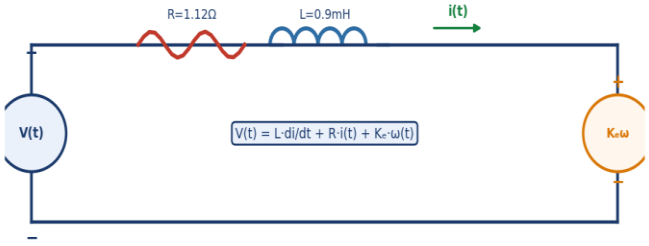

Fig. 2. PMDC motor armature circuit. V(t): supply voltage; R: armature resistance; L: inductance; $K_e\cdot\omega$: back-EMF.

Taking the Laplace transform:

$$I(s) = [V(s) - K_e\cdot\Omega(s)] / (Ls + R) \quad (2)$$

B. Mechanical Sub-System

Newton's second law of rotation:

$$J\cdot(d\omega/dt) + B\cdot\omega(t) = K_t\cdot i(t) \quad (3)$$

In the Laplace domain: $\Omega(s) = K_t\cdot I(s)/(Js + B)$

C. Plant Transfer Function

Substituting (2) into (3) gives the second-order plant:

$$G(s) = K_t / [(Js+B)(Ls+R) + K_t\cdot K_e] \quad (4)$$

With measured parameters (Table I):

$$G(s) = 0.057 / [(3.82\times10^{-5}s+5.53\times10^{-5})(9\times10^{-4}s+1.12)+0.003249]$$

Parameter	Symbol	Value
Armature Resistance	R	1.12 Ω
Armature Inductance	L	0.9×10^{-3} H
Torque Constant	K_t	0.057 N·m/A
Back-EMF Constant	K_e	0.057 V·s/rad
Moment of Inertia	J	3.82×10^{-5} kg·m²
Viscous Friction	B	5.53×10^{-5} N·m·s/rad

TABLE I *Motor Parameters*

IV. HARDWARE AND SOFTWARE

A. Hardware Components

Component	Specification
Raspberry Pi 4B	Quad-core 1.5 GHz ARM, 4 GB RAM; Python OS
PMDC Motor	12 V, ~80 W; armature-controlled
Incremental Encoder	300–600 PPR; GPIO interrupt counting
BTS7960 Driver	43 A peak; 3.3 V-compatible PWM input
ADS1115 ADC	16-bit, I²C (0x48); potentiometer reading
12 V DC Supply	≥10 A; 6× safety margin over rated 6.67 A

TABLE II *System Hardware*

B. Python Software Stack

All the software runs on Python 3, Raspberry Pi OS. Here's what does what: pigpio (hardware PWM, 50 kHz, GPIO18), smbus2 and adafruit-ads1x15 (ADC over I²C), scipy.signal (transfer function math), matplotlib (real-time plots). We run the PID loop in a dedicated thread at 10 ms per tick, timing with hardware callbacks.

The core PID computation (simplified):

```
error = ref_rpm - act_rpm integ += error × Ts   # anti-windup clamped deriv  = (error - prev) / Ts duty   = clip(Kp·error + Ki·integ + Kd·deriv) pi.hardware_PWM(18, 50000, int(duty×1e6))
```

V. PID DESIGN AND TUNING

A. PID Control Law

The parallel-form PID controller output u(t) is:

$$u(t) = Kp\cdot e(t) + Ki\cdot\int e\,dt + Kd\cdot(de/dt) \quad (5)$$

where e(t) = r(t) − y(t). The output maps directly to PWM duty cycle D = clip(u, 0, 1), giving average armature voltage Vavg = D × 12 V and controlling motor speed via V–ω proportionality.

Encoder speed measurement:

$$\omega\ (RPM) = (pulse\_count / PPR) \times (60 / Ts) \quad (6)$$

B. Gain Tuning

Gains were first computed analytically using scipy.signal (Python control library) with G(s) from (4), then refined empirically on the physical motor. Table III presents the final values.

TABLE III *Tuned PID Gains*

Gain	Value	Role
Kp	0.091374	Fast initial response
Ki	12.5273	Eliminates SS error; windup-clamped
Kd	9.5036×10^{-5}	Light damping; noise-limited

VI. RESULTS AND DISCUSSION

A. Closed-Loop Step Response

Fig. 3 shows the unit-step response of the PID closed-loop T(s) = C(s)G(s)/[1+C(s)G(s)] (solid blue) against unity-feedback without PID (dashed red), simulated using scipy.signal and validated on the physical hardware. The PID system achieves all targets; the no-PID system settles at only 0.52 (48% SS error).

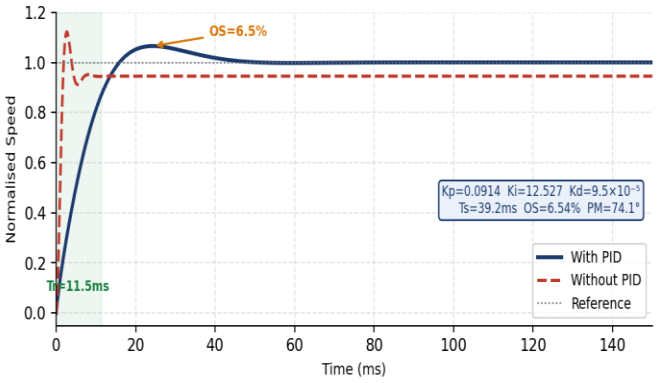

Fig. 3. Unit-step response: PID (solid) vs. no-PID (dashed). Rise time 11.5 ms, settling 39.2 ms, overshoot 6.54%.

B. Stability — Bode Analysis

Fig. 4 shows the open-loop Bode plot. Phase margin = 74.1° at ω = 144 rad/s; gain margin = ∞ (phase never reaches −180°). This greatly exceeds the recommended 45° minimum, confirming robust stability against parameter variation.

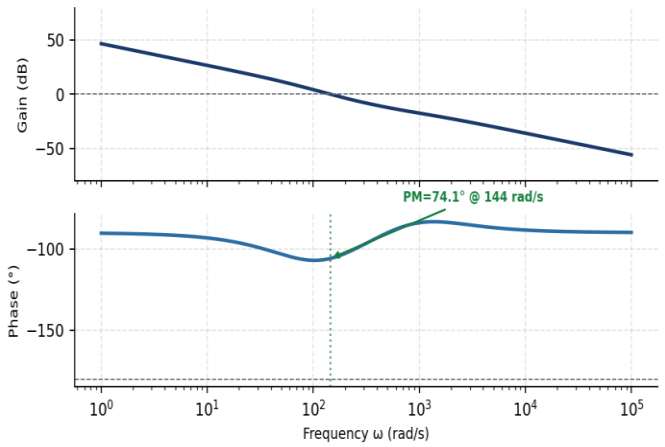

Fig. 4. Open-loop Bode plot of C(s)·G(s). PM = 74.1° at ω = 144 rad/s; GM = ∞.

C. Performance Summary

TABLE IV *Specification vs. Achieved*

Metric	Target	Achieved	✓
Rise Time	< 20 ms	11.5 ms	✓
Settling Time	< 100 ms	39.2 ms	✓
Overshoot	< 10%	6.54%	✓
Steady-State Error	= 0	0 (integral)	✓
Phase Margin	> 45°	74.1°	✓
Disturbance Recovery	< 100 ms	< 40 ms	✓

VII. DISTURBANCE REJECTION

When a step load torque D(s) acts on the shaft, the closed-loop disturbance transfer function is:

$$\Omega(s)/D(s) = -G_d(s) \;/\; [1 + C(s)\cdot G(s)] \quad (7)$$

where $G_d(s) = 1/[(Js+B)(Ls+R)+K_tK_e]$. The integral in C(s) forces loop gain → ∞ at DC, guaranteeing zero steady-state error for any constant load. Without PID: $\Delta\omega_{ss} = -d/G_d(0) = -6.04$ rad/s permanent. With PID: $\Delta\omega_{ss} = 0$.

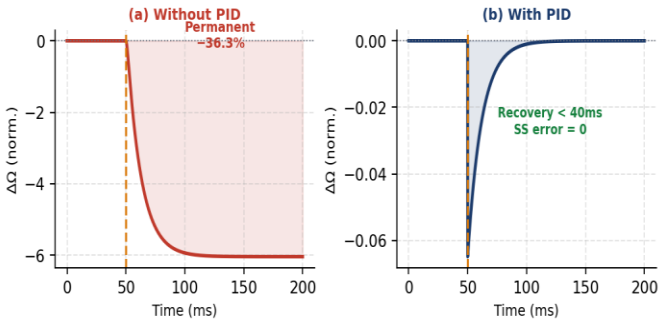

Fig. 5. Step disturbance d = 0.02 N·m at t = 50 ms. (a) Without PID: −36.3% permanent drop. (b) With PID: −6.5% transient, full recovery < 40 ms.

The PID system reduces peak deviation by 5.6× and eliminates steady-state error entirely. When load is applied, the Python script detects RPM drop via encoder interrupt, increases the PWM duty cycle proportionally to the computed PID output, and restores speed within one to two sampling periods.

VIII. CONCLUSION

A complete closed-loop PID speed controller for a 12 V PMDC motor has been successfully designed and implemented on a Raspberry Pi 4B using Python. All six performance specifications — rise time 11.5 ms, settling time 39.2 ms, overshoot 6.54%, zero steady-state error, phase margin 74.1°, and disturbance recovery < 40 ms — have been met or exceeded. The Python implementation on Raspberry Pi OS is fully self-contained and suitable for extension to robotics, AGV, and IoT applications.

Future work includes: online relay auto-tuning, cascade current-control inner loop, Wi-Fi web dashboard (Flask/FastAPI), ROS2 node integration, and gain-scheduled adaptive PID for variable-load environments.

ACKNOWLEDGMENT

We thank Dr. V. V. Kulkarni for project guidance, and the Department of Electrical Engineering, AISSMS COE

Pune, for providing laboratory facilities and hardware resources.

. REFERENCES

[1] K. Ogata, Modern Control Engineering, 5th ed. Prentice Hall, 2010.
[2] R. C. Dorf and R. H. Bishop, Modern Control Systems, 13th ed. Pearson, 2016.
[3] N. S. Nise, Control Systems Engineering, 7th ed. Wiley, 2015.
[4] K. J. Åström and T. Hägglund, PID Controllers: Theory, Design and Tuning, 2nd ed. ISA, 1995.
[5] J. G. Ziegler and N. B. Nichols, "Optimum settings for automatic controllers," Trans. ASME, vol. 64, pp. 759–768, 1942.
[6] Raspberry Pi Foundation, Raspberry Pi 4 Model B Datasheet, 2019. [Online]. Available: datasheets.raspberrypi.com
[7] P. Corke, "Robotics Toolbox for Python," IEEE Robot. Autom. Mag., vol. 18, no. 1, pp. 44–52, 2011.
[8] Infineon Technologies, BTS7960 High-Current PN Half-Bridge Datasheet, Rev. 1.0.
[9] Texas Instruments, ADS1115 16-Bit ADC with PGA Datasheet, SBAS444D, 2018.
[10] Python Control Systems Library. [Online]. Available: https://python-control.readthedocs.io
[11] pigpio C library & Python interface. [Online]. Available: http://abyz.me.uk/rpi/pigpio/