

The Preserve: Lehigh Library Digital Collections

Resistance Tracking for Predictive Maintenance in Brushed DC Motors: An Adaptive Model-Based Approach

Citation

Mickley, Seth. *Resistance Tracking for Predictive Maintenance in Brushed DC Motors: An Adaptive Model-Based Approach.* 2025, https://preserve.lehigh.edu/lehigh-scholarship/graduate-publications-theses-dissertations/theses-dissertations/resistance-0.

Find more at https://preserve.lehigh.edu/

This document is brought to you for free and open access by Lehigh Preserve. It has been accepted for inclusion by an authorized administrator of Lehigh Preserve. For more information, please contact preserve@lehigh.edu.

Resistance Tracking for Predictive Maintenance in Brushed DC Motors: An Adaptive Model-Based Approach

by

Seth Mickley

Presented to

the Graduate and Research Committee of

Lehigh University

in Candidacy for the Degree of

Master of Science

in

Electrical Engineering

Lehigh University

January 2026

© 2025 Copyright

Seth Mickley

All Rights Reserved

THESIS SIGNATURE SHEET

This thesis is accepted and approved in partial fulfillment of the requirements for the Master of Science in Electrical Engineering.

Resistance Tracking for Predictive Maintenance in Brushed DC Motors:
An Adaptive Model-Based Approach
Seth Mickley

_____

Date

_____

Professor Javad Khazaei

Thesis Advisor

_____

Professor Svetlana Tatic-Lucic

Chair of Electrical and Computer Engineering

Acknowledgements

I would like to express my sincere gratitude to Professor Javad Khazaei for his guidance, encouragement, and support throughout this project. His willingness to allow me the freedom to experiment and explore my own ideas made this research both challenging and deeply rewarding. I have greatly appreciated the opportunity to learn under his mentorship and value the patience, insight, and rigor he brought to every discussion. His emphasis on independent thinking and critical analysis has profoundly shaped my approach to engineering research.

I am also deeply grateful to retired Professor Harold Scholz of Penn State Lehigh Valley for inspiring me early in my academic journey to become an independent learner. His influence and belief in my potential have been foundational to my development and perseverance.

I would like to thank the faculty and staff of the Electrical and Computer Engineering Department at Lehigh University for fostering an environment that encourages creativity, practical exploration, and collaboration. Finally, I owe special thanks to my family and friends for their unwavering support, understanding, and motivation throughout this process.

Soli Deo Gloria.

Table of Contents

Acknowledgements **iv**

Abstract **1**

1 Background **2**

1.1 Introduction to Brushed DC Motors 2

1.2 Existing Methods for Brushed Motor Diagnosis 6

1.3 Limitations of Existing Research 12

1.4 Machine-Learning–Based Diagnostics in Other Motor Types 13

1.5 Proposed Approach: The Resistance Spectrum Method 16

1.5.1 Brushed Motor Design 17

2 Controller Design **21**

2.1 Dynamic Model of Brushed DC Motors 21

2.2 Model Predictive Control for Brushed DC Motors 24

2.3 Proposed MPC Controller 32

2.3.1 Prediction Model, Cost, and Actuation Limits 33

2.3.2 Two-Channel Backsolver Observer 33

2.4 Phase-Locked Loop 41

2.4.1 Ripple PLL and Phase-Synchronous Scheduling 41

2.4.2 Windowing and Minima Confirmation. 42

2.5 Inner Control Loop (Fitted Bus + Adaptive Phase) 48
2.5.1 Duty Computation and Quantization 49
2.5.2 Bias Removal and Adaptive Ripple Compensation 51

3 Adaptive Control and Online Model Identification 56

3.1 Overview 56
3.2 Memory and Buffering Strategy 58
3.3 Signal Preprocessing and Filtering Pipeline 59
3.4 Model Fitting Procedure 62
3.4.1 Acquisition protocol: acceleration/launch-cycle fitting 62
3.4.2 Electrical Parameter Estimation 64
3.4.3 Mechanical parameter estimation 65
3.5 Resistance Spectrum Reconstruction 66
3.5.1 Bus-voltage Reference (optional online fit) 68

4 Results 70

Conclusion 85
Vita 194

List of Tables

1.1 Comparison of diagnostic methods for brushed DC motor wear. . . . 11

1.2 ML methods for motor health diagnostics (discussed methods). 15

2.1 Comparison of MPC based control methods for brushed DC motors. . 39

2.2 Key configuration parameters for the windowed-minima detector. . . 43

2.3 Principal PLL signals and their functional roles. 46

2.4 Compensator variables, units, and roles. 52

4.1 True theoretical parameters for Motors 1–4. 81

4.2 Percent error of solver-identified parameters for Motors 1–4 using the **850 Hz** Butterworth filter. 81

4.3 Percent error of solver-identified parameters for Motors 1–4 using the **550 Hz** Butterworth filter. 81

List of Figures

1.1 Internals of a brushed DC motor (modified with labels by the author). Source: [1]. 3
1.2 Comparison of SCR and PWM DC motor drives. 5
1.3 Experimental setup for sparking assessment in brushed DC motors. Source: Fig. 1 from [2]. 8
1.4 Experimental setup for brush contact assessment using a rotating disc. Source: Fig. 3 from [3]. 9
1.5 Internal resistance arrangement for a brushed DC motor with a 12-segment commutator. The ring topology emphasizes cyclic structure. 18
1.6 Photograph of a 24-segment armature showing winding arrangement and commutator construction. 19

2.1 Electrical and mechanical model of a brushed DC motor. The diagram shows the coupling between the voltage, current, and torque domains. 22
2.2 General MPC architecture. 24
2.3 Overall control architecture and layout of the proposed drive. 32
2.4 MPC layer implementing receding-horizon speed control, using disturbance and resistance observers that supply d each control step and R once per phase-aligned window to the predictive model. 40

2.5 Simplified control flow of the rectified-bus phase-locked loop. The PLL tracks the minima of V_{bus}, estimates the 120 Hz period, and synthesizes a continuous phase $\phi(t)$ used for all ripple-synchronous events. 47
2.6 Compensator control flow. The phase ϕ comes from the PLL; the adaptive offset ϕ_{raw} is advanced by a commanded rate that is nudged at sub-phase ticks using bias-removed, ripple-synchronous error. . . . 55

4.1 Uncompensated armature current showing pronounced 120 Hz modulation synchronized with bus recharge. 72
4.2 Armature current with ripple compensation applied. The 120 Hz component is strongly attenuated. 73
4.3 Uncompensated load voltage showing strong recharge-edge peaking caused by bulk-capacitor charging. 74
4.4 Load voltage under compensation, showing suppression of recharge-edge distortion and alignment with the reference waveform. 75
4.5 Duty-cycle trajectory showing phase-synchronized modulation during bus ripple cycles. 76
4.6 Tracking-error signal used for adaptive phase correction. The mean error collapses once compensation converges. 77
4.7 Adaptive phase reference $\phi(t)$ used for ripple synchronization. Small, bounded corrections maintain phase lock. 78
4.8 Steady-state speed with compensator active, showing reduced 120 Hz modulation. 79
4.9 Speed response at 50 rad/s. Ripple correction is minimal at low power. 80
4.10 Illustrative MPC-style preview control result at $\omega^\star = 200$ rad/s showing zero overshoot within the steady-state error band. 83
4.11 Filtered resistance spectrum after compensation, showing uniform contact resistance and negligible distortion. 84

Abbreviations

Abbreviation	Definition
AC	Alternating Current: Periodically reversing electric current.
ADC	Analog-to-Digital Converter: Converts analog voltages or currents to digital values.
AI	Artificial Intelligence: Algorithms and systems that simulate human intelligence.
ANN	Artificial Neural Network: Computational model inspired by biological neural networks, used for pattern recognition and ML.
ANFIS	Adaptive Neuro-Fuzzy Inference System: Hybrid model combining neural networks and fuzzy logic for nonlinear modeling and control.
BEMF	Back Electromotive Force: Voltage induced by rotor motion opposing applied voltage.
BLDC	Brushless DC (Motor): Electric motor using electronic commutation instead of mechanical brushes and commutator.
BPNN	Backpropagation Neural Network: ANN trained using the backpropagation algorithm for regression or classification.
DC	Direct Current: Electric current that flows in one direction only.
ECR	Electrical Contact Resistance: Resistance at the interface where electrical contact is made, such as between brush and commutator.
EMF	Electromotive Force: Voltage developed by a source of electrical energy.
FEM	Finite Element Method: Numerical technique for solving complex physical problems via discretization.
FPGA	Field-Programmable Gate Array: Reconfigurable logic device for high-speed, parallel signal processing.
HMM	Hidden Markov Model: Statistical model for systems with unobservable (hidden) states, used in sequence modeling and fault detection.
HP	Horsepower: Unit of power (1 HP $\approx$ 746 W), often used to rate motors.
HVAC	Heating, Ventilation, and Air Conditioning: Systems for indoor climate control.
IGBT	Insulated-Gate Bipolar Transistor: High-power switch combining MOSFET gate drive with bipolar conduction.
MCU	Microcontroller Unit: Integrated controller for embedded applications (e.g., TI C2000, STM32).
MFCC	Mel Frequency Cepstral Coefficients: Features used in signal processing, especially for audio and vibration analysis.
ML	Machine Learning: Subset of AI focused on algorithms that improve automatically through experience.
MLP-NN	Multilayer Perceptron Neural Network: ANN with multiple layers for complex pattern recognition.
MOSFET	Metal–Oxide–Semiconductor Field-Effect Transistor: Fast, voltage-driven power switch used in PWM drives.
MPC	Model Predictive Control: Optimization-based control using a rolling horizon.
MSE	Mean Squared Error: Average squared difference between estimated and true values.

Abbreviation	Definition
PC	Personal Computer: General-purpose computer.
PI	Proportional–Integral (Controller): Feedback controller used in automation and motor drives.
PLL	Phase-Locked Loop: Control system that locks a phase/oscillator to a reference signal.
MRAS	Model Reference Adaptive System: Adaptive control or estimation method comparing a reference model to the plant response.
PM	Permanent Magnet: Motors or systems utilizing permanent magnets instead of wound fields.
PMDC	Permanent Magnet DC (Motor): Brushed DC motor using permanent magnets for the stator field.
PWM	Pulse Width Modulation: Control of average voltage or current by rapid switching with variable duty cycle.
RBF-NN	Radial Basis Function Neural Network: ANN using radial basis functions as activations.
RF	Random Forest: Ensemble ML algorithm building multiple decision trees and merging their results.
RNN	Recurrent Neural Network: ANN suited for processing sequences (time series).
RUL	Remaining Useful Life: Estimated time before a system or component can no longer perform its function.
SBC	Single-Board Computer: Compact computer on a single PCB (e.g., Raspberry Pi, BeagleBone).
SCR	Silicon Controlled Rectifier: Thyristor device used in phase-angle-controlled drives.
SISO	Single-Input Single-Output: Control system with one input variable and one output variable.
SVM	Support Vector Machine: Supervised learning model for classification (and, with extension, regression).
SVR	Support Vector Regression: SVM applied to regression (predicting continuous values).
TI	Texas Instruments: Semiconductor manufacturer (e.g., C2000 MCUs, fuel-gauging ICs).
WMM	Wavelet Modulus Maxima: Time–frequency technique for detecting abrupt changes such as arcing events.

Abstract

Brushed Permanent-Magnet Direct-Current (PMDC) motors remain widely used in industrial and commercial systems, yet condition-monitoring methods for these drives continue to lag behind those of modern motor types. Common approaches such as thermal trending, vibration analysis, and arcing detection have largely remained confined to laboratory studies and have not translated into practical field solutions for simple, low-cost drives. These techniques would benefit from reliable access to winding and commutator resistance, a fundamental indicator of brush and segment wear, but in practice this parameter is difficult to extract during normal operation.

This work presents a cascaded control architecture that enables reliable, resistance-based health monitoring. The inner loop uses a custom Phase-Locked Loop (PLL) bus-aware compensator to provide a clean, ripple-synchronous reference and to stabilize the drive under supply ripple. Building on this foundation, the outer loop implements Model Predictive Control (MPC) for speed regulation and adaptive model fitting to estimate winding and commutator resistance in real time. The estimation becomes practical because the inner-loop compensator suppresses supply ripple that would otherwise corrupt the measurement. Although the primary focus is extracting a clean signal for resistance estimation, the inner loop also provides broader control advantages and may serve as the foundation for a future sensorless approach for Alternating-Current (AC) fed brushed DC motors, a capability not available in current drive designs.

Chapter 1

Background

1.1 Introduction to Brushed DC Motors

Despite the rise of brushless technology, traditional brushed DC motors remain prevalent, especially in legacy systems and cost-sensitive embedded environments. Their mechanical simplicity, low cost, and straightforward control keep them relevant across industrial domains. Historically, brushed motors have excelled in automotive subsystems, Heating, Ventilation, and Air Conditioning (HVAC) actuators, industrial valve controls, conveyor systems, and specialized high-torque applications such as mining equipment, railway actuators, and starter motors [4, 5].

The key operational difference between brushed and brushless motors lies in how each manages current direction to sustain torque. Brushless motors rely on external electronic controllers to synthesize sinusoidal or trapezoidal stator fields. Brushed DC motors use a mechanical commutator and carbon brushes to reverse current through rotor windings at the proper intervals. The armature contains multiple windings embedded in a laminated iron core, with each winding tied to a commutator segment. As the rotor turns, stationary brushes maintain sliding contact with the commutator bars and direct current through the appropriate windings to sustain

torque. This inherent mechanical commutation lets brushed motors operate from basic DC sources and simplifies control electronics [5].

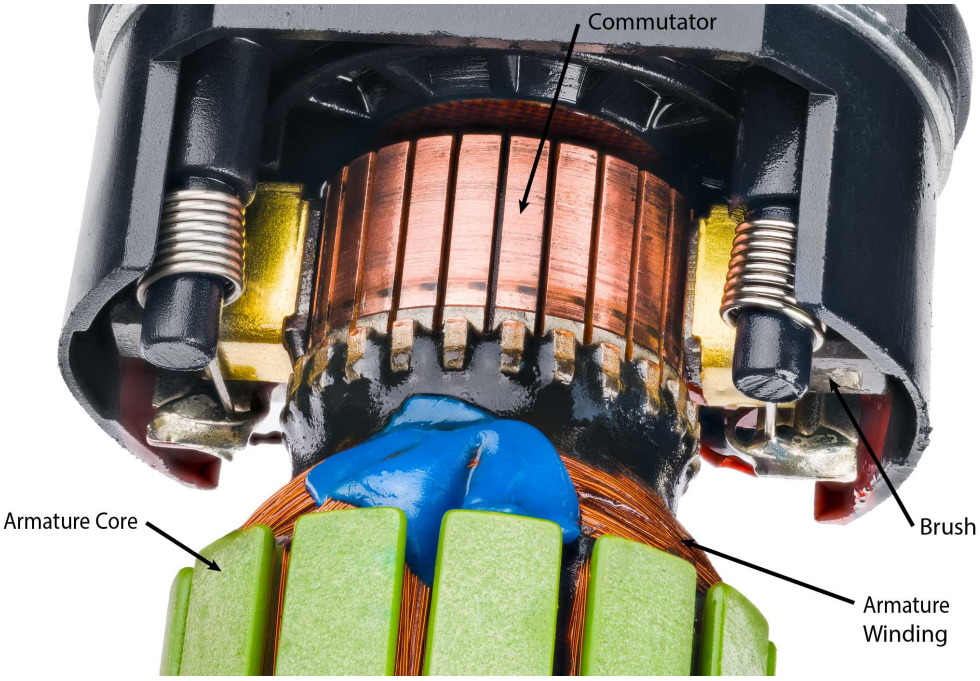


Figure 1.1: *Internals of a brushed DC motor (modified with labels by the author). Source: [1].*

Drive Topologies

Pure DC supplies are scarce in industrial settings, so brushed DC motors are commonly controlled using either Silicon-Controlled Rectifier (SCR) drives or Pulse-Width Modulation (PWM) circuits.

SCR drives. SCR drives use phase-angle control. One or more SCRs in the rectifier stage regulate the average motor voltage by delaying conduction in each AC half-cycle. This approach is simple, rugged, and highly cost-effective at high power, making it well suited for heavy industrial environments such as steel mills, mining hoists, rolling mills, and large material-handling systems where high torque and low cost are primary requirements [6].

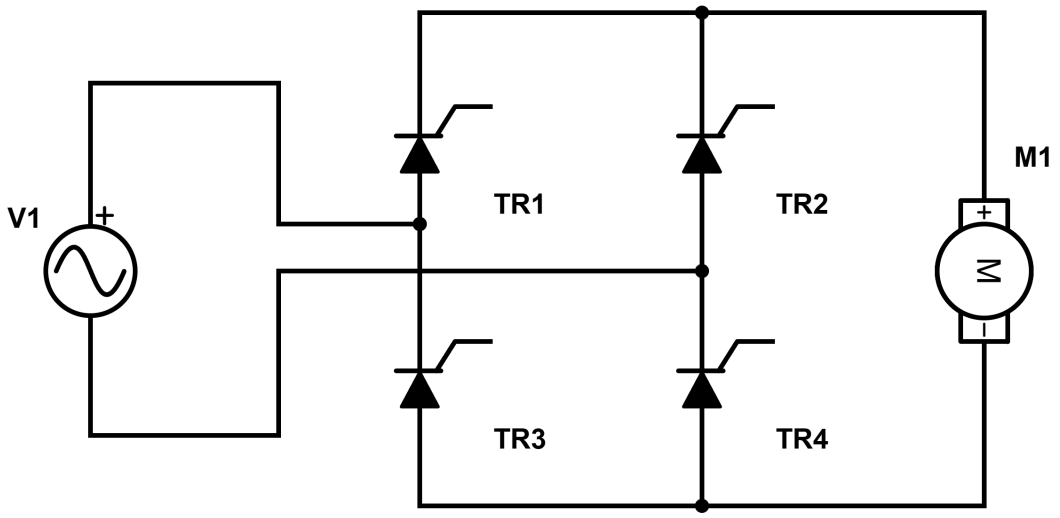
PWM drives. PWM controllers use high-frequency switches, such as Insulated-

Gate Bipolar Transistors (IGBTs) or Metal–Oxide–Semiconductor Field-Effect Transistors (MOSFETs), to chop the DC bus into a pulsed waveform. Varying the pulse width adjusts the average voltage delivered to the motor, providing smooth and efficient speed regulation. High switching frequency improves control precision, reduces torque ripple, and raises efficiency compared with SCR methods. PWM has become the standard in automotive subsystems, compact industrial actuators, robotics, and embedded applications where current demands are moderate and responsive control is essential [7].

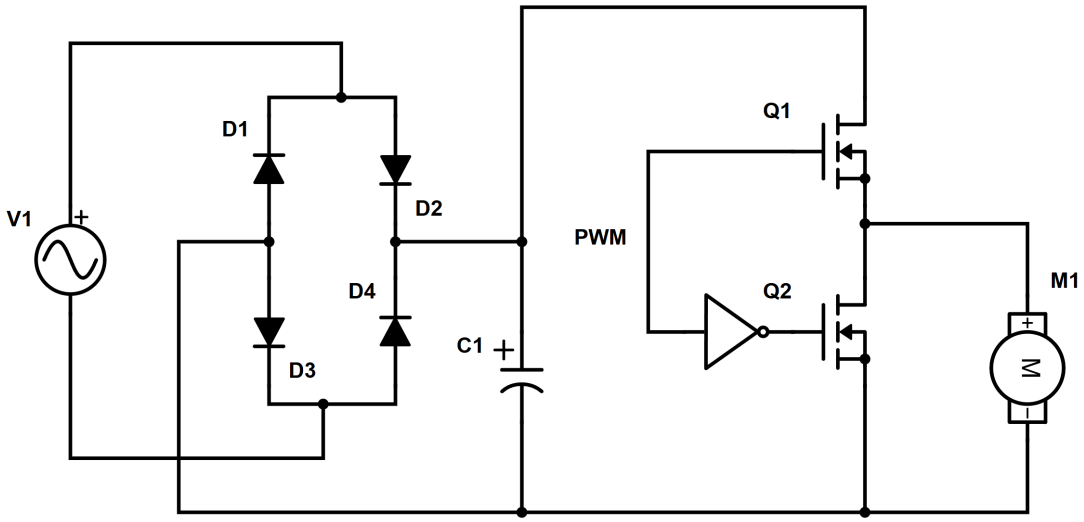
Motor Classes and Applications

Brushed DC motors span several topologies. Shunt-wound designs offer stable speed regulation for steady-state applications such as industrial machinery and HVAC blowers. Series-wound designs provide high starting torque and are used in traction systems, automotive starters, and electric tools. Compound-wound designs combine shunt and series characteristics for dynamic loads, including cranes and elevators [4–6].

Power classes also vary. Permanent-Magnet (PM) motors are common in embedded applications that require compact size and simple control, including automotive climate systems, medical actuators, and small industrial devices. Mid-range motors (1–10 HP) appear in conveyor systems, large HVAC installations, pumps, and automation equipment. High-power brushed motors (exceeding 10 HP), typically compound or separately excited shunt machines, remain essential in legacy installations such as steel rolling mills, marine winches, mining hoists, and heavy railway actuators where modernization is difficult [4, 6].



(a) Single-phase SCR bridge DC motor drive.



(b) Single-phase PWM chopper DC motor drive with diode bridge rectifier.

Figure 1.2: *Comparison of SCR and PWM DC motor drives.*

Wear at the Brush–Commutator Interface

All brushed DC motors share a critical vulnerability: the brush–commutator–armature interface. This zone facilitates current transfer to the rotating windings but introduces wear mechanisms over time, including resistance drift, carbon buildup, and arcing.

Analyses across industrial sectors identify brush and commutator wear as a leading failure cause [8, 9]. Focusing on PMDC motors, whose fixed field and simple series path simplify modeling, lays the groundwork for a scalable predictive-maintenance framework. The methodology extends to shunt and compound-wound designs common in high-power environments.

Over decades, many diagnostic strategies have targeted brush wear. The brush, commutator, and armature are interdependent. Excessive brush wear increases arcing, which accelerates commutator pitting, raises contact resistance, and elevates local temperature. If unchecked, this damage path can lead to failure, which makes brush wear a baseline metric for predictive maintenance.

Brush design balances electrical conductivity, mechanical durability, and thermal performance. Copper lowers resistivity and supports current but erodes faster. Higher graphite content extends life and reduces abrasion, yet increases resistive losses and heat [3, 10]. Mechanical alignment and spring pressure remain essential but offer little diagnostic visibility during normal operation. Subtle degradation often goes unnoticed until performance declines. The next section reviews practical methods that aim to surface wear trends before faults arise.

1.2 Existing Methods for Brushed Motor Diagnosis

Thermal Monitoring

Thermal monitoring is practical and effective for identifying early wear. Heat generation tracks electrical and mechanical losses, including copper losses and brush–commutator friction. Studies show that abnormal heat near brushes or bearings signals commutator wear or carbon accumulation; faulty motors exhibit higher steady-state tempera-

tures than healthy ones [11,12]. Differences are measurable with basic thermocouples, and calibrated thermal models can be embedded for real-time monitoring.

Work in [13] reinforced thermal modeling by examining operation in vacuum, where conduction and convection are limited. Wear-related heating remained observable, especially when resistance-based temperature estimation was combined with Thermal-Network modeling and the Finite Element Method (FEM). The study used advanced electrothermal techniques but confirmed a key insight: even under constrained conditions, internal degradation can be inferred from modeled external signals.

Limitations. Detailed thermal property identification, Finite-Element calibration, and controlled boundary conditions are difficult to obtain and maintain in the field. Ambient airflow, load variation, enclosure geometry, and installation variability complicate modeling. Not all degradation manifests as gradual heat; arcing events are abrupt and localized and may evade thermal networks.

Arcing as Degradation Mechanism and Signal

Arcing is both a wear driver and a diagnostic indicator. Using a custom switching apparatus, [14] isolated arcing effects with a Cu-graphite–to–copper contact pair and showed that arcing accelerates erosion and roughness, raising resistance and reducing service life. Quenching reduced damage, underscoring arcing's dominant role.

Mild arcing can burn off films, but sustained arcing raises temperature, damages surfaces, and drives chemical decomposition, which degrades the interface [15–17]. These processes rank among primary failure modes [18, 19]. In laboratories and in industry, arcing behavior is a standard health indicator.

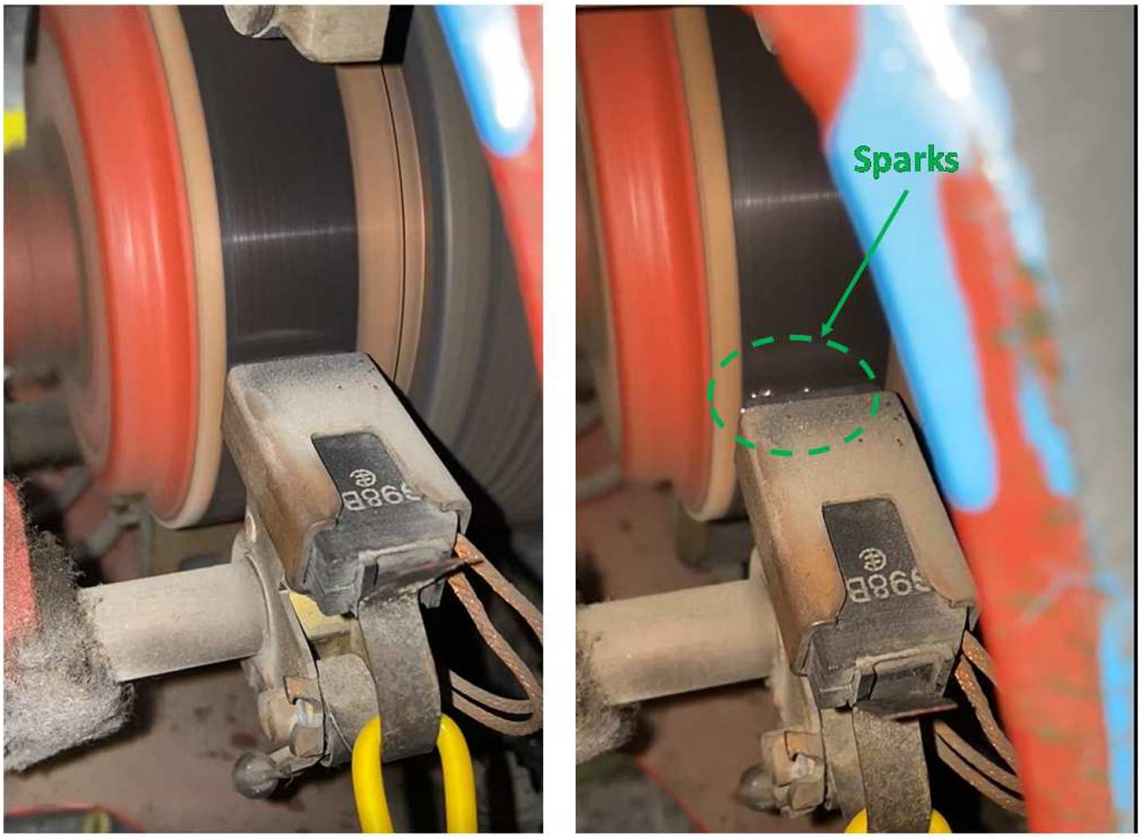


Figure 1.3: *Experimental setup for sparking assessment in brushed DC motors. Source: Fig. 1 from [2].*

Beyond inspection, arcing is detectable through sensorless electrical diagnostics. Techniques focus on sharp, microsecond-scale transients in voltage or current. Wavelet Modulus Maxima (WMM) analysis is prominent because it offers time–frequency localization that isolates transients [20–22]. The modulus-maxima step targets abrupt multi-scale energy changes that correlate with arcs. Two studies extended this approach to startup transients, when arcing is prevalent, and demonstrated nonintrusive diagnostics under controlled conditions [2, 23].

Vibration and Stray-Flux Sensing

Vibration analysis [24, 25] uses accelerometers or vibrometers to detect oscillations. Signal changes correlate with mechanical wear or imbalance, such as commutator sur-

face degradation, brush chatter, or bearing faults. These signals give indirect evidence of electrical contact issues but primarily reflect broader mechanical disturbances.

Stray-flux analysis [26] monitors magnetic disturbances from sparking, brush lift-off, and uneven commutation. Although [26] used a wound-field DC motor, the concept applies to PM machines. The signature is transient flux leakage caused by irregular current or arc-induced imbalance. Placement is sensitive, and smaller motors can reduce sensitivity due to weaker fields and tight packaging.

Electrical Contact Resistance (ECR)

Electrical Contact Resistance (ECR) studies the interface directly. High-precision Kelvin four-wire measurements isolate the resistive component [27] but are performed on rigs, not complete motors. Simpler two-wire rigs, such as a graphite brush on a rotating copper disc, measure voltage and current at the same terminals [28]. Because windings are excluded, the voltage drop reflects interface resistance alone. This configuration is ideal for chemistry and contamination studies but sacrifices realism, since current ripple, inductive coupling, and commutation dynamics are absent.

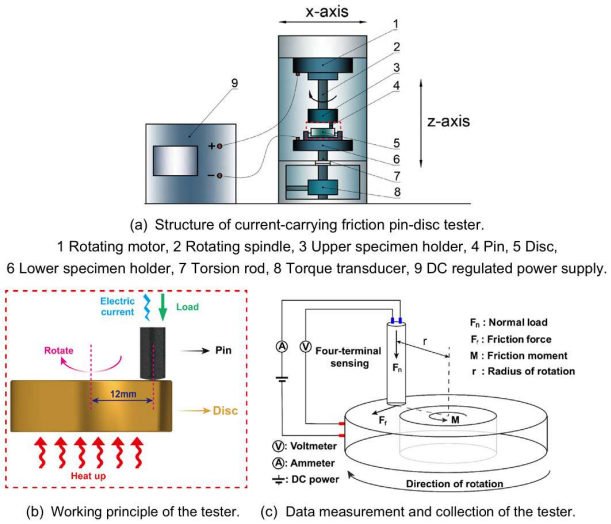


Figure 1.4: *Experimental setup for brush contact assessment using a rotating disc. Source: Fig. 3 from [3].*

A related study [13] applied the method more directly to a motor configuration, though the test is impractical to perform routinely in typical motor settings. The authors measured resistance across a complete motor at very low speeds near stall to minimize the influence of back electromotive force (BEMF). The work confirmed that resistance trends correlate with heating and wear, but the experimental conditions were tightly controlled.

Degradation Trends and Statistical Models

Electrical response can capture degradation trends. In [29], an on–off hydraulic valve system driven by brushed DC motors was monitored using only voltage and current. As degradation progressed, the actuation current response became delayed and distorted, indicating increased resistance and reduced mechanical efficiency. Although resistance was not explicitly computed, the electrical response served as a practical degradation indicator.

Benedik *et al.* analyzed more than 3,900 vacuum-cleaner motors and used Multiple Linear Regression (MLR) to predict brush wear from voltage, current density, brush pressure, and sliding speed [30]. The model showed high predictive power but relied on design parameters that embedded diagnostics seldom know. Still, the study validated regression-based wear modeling and emphasized temperature, current, and voltage as key predictors.

Table 1.1 summarizes established metrics and methods, highlighting principles, sensors, and deployability.

Among these, WMM analysis is practical and informative for predictive diagnostics. It provides a field-accessible metric for visual or recorded evaluation of arcing behavior. Yet WMM has not been deployed widely in embedded hardware due to implementation and conditioning challenges discussed next.

Table 1.1: *Comparison of diagnostic methods for brushed DC motor wear.*

Method	Principle	Sensors	Pros	Cons	Deployability
Thermal Monitoring	Temperature rises with losses and wear.	Thermocouple, NTC, or RTD.	Non-invasive; industry standard.	Requires embedded sensors; large motors need many; retrofits are difficult.	Medium (easy for new builds, limited retrofit).
Thermal Network / FEM Modeling	Model-based internal temperature estimation.	External sensors; computational model.	No added hardware; infers internal state.	Requires calibration and accurate placement.	Medium (high-value machines).
Arcing Detection (Electrical, WMM)	Detects high-frequency transients in voltage/current.	Existing V/I sensors, fast ADC.	No extra hardware; direct arc detection.	Needs high-rate sampling and signal processing.	Medium–High (research-proven).
Vibration Analysis	Wear changes vibration signatures.	Accelerometer or vibrometer.	Mature; effective for bearings and chatter.	Mounting required; environmental noise; not brush-specific.	Medium (general motor health).
Stray-Flux Sensing	Detects commutation/arc-related magnetic fields.	Hall sensor, coil, or magnetometer.	Non-contact; electrical-fault sensitive.	Placement critical; EMI-prone.	Medium (setup-sensitive).
Electrical Contact Resistance (ECR)	Interface resistance tracks degradation.	2- or 4-wire rigs; Kelvin probes.	Highly accurate; isolates interface.	Not fieldable; isolated test or low speed.	Very Low (laboratory).
ECR Motor (Low-Speed)	Resistance from slow ramp at low BEMF.	Motor terminals; slow ramp.	Uses actual machine.	Time-consuming; multiple samples.	Low (diagnostics or lab).
Electrical Response (Trends)	Degradation shifts V–I behavior.	Standard V/I sensors.	No extra hardware; field-tested; suitable for trending.	Less direct; may miss local faults.	High (easy to embed).
Statistical Regression Models	Predict wear from multiple inputs.	V/I data plus design parameters.	Flexible; accurate with data.	Needs brush/-force data; often impractical.	Medium (if inputs are digital).

1.3 Limitations of Existing Research

Several promising signal-based approaches face barriers that limit integration into embedded predictive-maintenance hardware.

Sampling and PWM Constraints

Arc detection in laboratories typically uses steady DC or low-frequency supplies. Researchers then isolate arcs in armature current or stray-flux signals at kilohertz sampling rates [2,23,26]. Under these conditions, WMM or entropy-based transforms focus on bands up to roughly 2.5 kHz and capture millisecond-scale arc events.

In embedded drives, PWM control is ubiquitous between 10–20 kHz. Switching transients and ripple overlap with arc spectra. Frequent duty-cycle updates shift transition timing, which blurs discrimination between genuine arcing and modulation artifacts. To date, a robust, validated solution for arc detection in PWM-driven embedded systems is not widely documented.

Sensor and Compute Constraints

The spectral overlap increases sampling-rate and bandwidth requirements, often exceeding the capabilities of standard motor amplifiers and Analog-to-Digital Converters (ADCs) once galvanic isolation is introduced. Isolation adds delay and limits bandwidth. Many microcontrollers can nominally sample near 1 MSPS with 12–16 bit resolution, but practical integration reduces this margin (for example, in the C2000 line from Texas Instruments). Field-Programmable Gate Arrays (FPGAs) can alleviate these constraints through parallel processing, but they add cost and design complexity.

Implications for This Work

Our strategy minimizes external sensors, draws from arc-based insights, and targets robustness to real-world noise. Thermal and vibration methods remain complementary. We also take cues from data-driven and Machine-Learning (ML) diagnostics common in other motor types, while coupling the health metric directly into control, a concept that remains underexplored for brushed DC drives.

To our knowledge, Hu *et al.* present the only ML-based diagnostic study focused specifically on brushed DC motors. Their work applied a Hidden Markov Model (HMM) to simulated current signals representing four modeled fault conditions: normal, inter-turn short, brush wear, and resistance increase, generated from the motor's dynamic equations with added Gaussian noise [31]. The model was trained offline and achieved high accuracy in distinguishing these simulated states, demonstrating the potential of supervised probabilistic learning for sequence-based fault recognition. However, the method remains limited to categorical classification of simulated faults rather than continuous health estimation. In this work, we take a control-oriented approach instead, estimating evolving electrical parameters in real time with a focus on resistance as a quantitative health metric directly integrated into the control loop rather than inferred from raw waveform classification.

1.4 Machine-Learning–Based Diagnostics in Other Motor Types

ML methods are more mature in induction and Brushless DC (BLDC) machines. For example, [32] used a piezoelectric contact microphone and a low-power microcontroller to extract Mel-Frequency Cepstral Coefficients (MFCCs) from induction-motor acoustics and classify bearing faults in real time. Other studies used Multilayer Perceptron Neural Networks (MLP-NNs) and Radial-Basis-Function Neural Networks

(RBF-NNs) to detect rotor-bar and stator faults [33], and Support Vector Machines (SVMs) to identify eccentricities and winding anomalies [34]. These approaches compress time-series data into statistical features such as skewness, kurtosis, and entropy for classifier training.

BLDC motors have likewise benefited from Recurrent Neural Networks (RNNs) that analyze voltage and current sequences for early-stage faults [35]. While effective, sequence models demand more compution and often act as black boxes focused on fault labels rather than predictive health.

Remaining Useful Life (RUL) estimation is central to predictive maintenance. As summarized in [36] (Table V), most RUL studies for electric motors rely on vibration features, especially for bearings. Comparable approaches for armature, commutator, and brush health in brushed DC motors are largely unexplored. This work addresses that gap by identifying an electrical metric sensitive to these failure modes.

Support Vector Regression (SVR) enables direct RUL estimation by mapping compressed trends, such as mean and slope, to remaining life [37]. The method is computationally light and well suited to embedded deployment. Conceptually, it aligns with our resistance-trend framework, in which progressive resistance change serves as the principal degradation indicator from which RUL could be inferred. While forecasting and long-term trending remain outside the scope of this work, SVR represents a natural companion method once the resistance metric is extracted, offering a lightweight and easily interpretable approach to remaining-life prediction.

Among the many methods surveyed in [36], only one focuses directly on resistance tracking. In induction drives, online stator and rotor resistance estimation using a Model Reference Adaptive System (MRAS) is well established [38]. Challenges include temperature drift and speed dependence, which necessitate observing key model parameters as the motor ages. Methods used to track such changes often employ Extended Kalman Filters (EKFs), Sliding-Mode Observers (SMOs), or fuzzy-

logic adaptation. Our brushed drives face similar temperature and speed effects, but the dominant barrier differs: the uncontrolled AC-rectified front end introduces deep periodic sags and ripple. These distortions complicate stable, interpretable measurements as the motor rotates. Overcoming bus-induced distortion is therefore the central control challenge addressed in this work.

In MRAS, a reference model is compared with an adaptive model driven by real-time voltage and current, and a Proportional–Integral (PI) law refines the resistance estimate until the models agree. Recent studies replace the PI law with small Artificial Neural Networks (ANNs) and variable learning rates [38]. We retain the principle of fitting a model to measured behavior but diverge in implementation by using lightweight, windowed optimization and introducing a custom inner loop to manage the severe supply ripple characteristic of AC-fed brushed drives.

Table 1.2 compares representative ML strategies from research and field deployments.

Table 1.2: *ML methods for motor health diagnostics (discussed methods).*

Method	Domain	Inputs	Contribution	Deployability
HMM Sequence Modeling	Brushed DC (sim)	Current, vibration sequence	Captures sequential and degradation states; interpretable; uses standard sensors; embedded-ready	Likely High (simulation to date)
MFCC + Classifier (SVM, RF, NN)	Induction (field)	Acoustic (MFCC)	Edge-capable; high bearing-fault accuracy; low cost hardware	High (deployed)
MLP-NN / RBF-NN	Induction (lab/-field)	Current, vibration, temperature, voltage	Handles multiple fault types; moderate compute	Medium (needs data/compute)
SVM / Random Forest	Induction, BLDC, general	Current, vibration, statistical features	Robust with simple features; widely used	Medium–High (often fielded)
Decision Tree / Ensemble	Induction, BLDC, general	Current, vibration, temperature	Interpretable; low compute	High (field-proven)
Fuzzy Logic / ANFIS	Induction, BLDC, general	Current, vibration, temperature	Handles nonlinearity and uncertainty	Medium–High (embedded use)
Deep NN / RNN	Induction, BLDC	Raw waveforms, trends	High sensitivity to time series; heavier inference	Medium (fielded; compute-heavy)
SVR Direct RUL Estimation	Bearings, general	Compressed trends (e.g., mean, slope)	Lightweight, direct RUL prediction; embedded-friendly	High (embedded-ready)
MRAS / ANN Resistance Estimation	Induction (drive)	Voltage, current, speed	Online resistance tracking for control; reference/adaptive modeling	Medium (drive-specific)

1.5 Proposed Approach: The Resistance Spectrum Method

The resistance-based metric introduced here is, to the author's knowledge, novel in the literature. It pairs naturally with the ML approaches above, but the spectrum itself is designed to embed substantial diagnostic information for rapid health checks without long-term trend accumulation. It can also be combined with other methods for robustness and automation.

Extraction Challenges

The dominant source of variability is contact resistance at the commutator and brushes, which fluctuates with rotation and age. BEMF complicates direct extraction from terminal measurements during operation. Standstill measurements are sometimes possible in laboratories but rarely feasible in the field unless periodic calibration is acceptable. In addition, a continuously oscillating bus emphasizes the inductive term $\mathrm{d}i/\mathrm{d}t$ in the terminal voltage, which further obscures the resistive component. A practical method must therefore separate BEMF and inductance from the resistance measurement. That requires an accurate motor model and a control strategy that minimizes supply ripple.

Modeling and Identification

We explicitly identify the plant so that electrical dynamics can be separated from rectifier-induced disturbances. A straightforward Mean Squared Error (MSE) minimization leverages the large volume of real-time data available on modern embedded controllers. The model runs in parallel with the motor, and its internal resistance parameter R is updated to minimize the difference between measured and model-predicted signals. Unlike continuous gradient adaptation, the method operates within

a moving window, collecting data over short intervals to update R efficiently while avoiding the computational overhead and difficult tuning associated with full extended filtering. A custom inner loop that combines a PLL with bus-aware voltage correction suppresses periodic sags from the AC-rectified supply and provides the clean environment required for accurate fitting.

Building the Resistance Spectrum

With a stable, ripple-compensated estimate of R, the controller generates a position-resolved resistance spectrum $R(\theta)$ during controlled startup or other known-load conditions. Aggregating resistance estimates across rotation yields a spatial map of commutator-segment condition. This profile is physically meaningful and directly interpretable, serving either as a standalone diagnostic for maintenance assessment or as an input to predictive models such as lightweight ML frameworks for RUL estimation. Once the plant is established and disturbance terms are back-solved, the controller can identify intervals of negligible load torque and use them for opportunistic recalibration without requiring a dedicated offline test.

1.5.1 Brushed Motor Design

Within a brushed DC motor, the electrical circuit emerges from the interaction of brushes, commutator bars, and armature windings. The commutator is a ring of copper bars, each tied to an armature winding end and insulated by mica or high-temperature polymer. Armature windings use thin enamel for inter-turn isolation. Coils sit in armature slots and terminate at specific bars.

As the armature rotates, carbon brushes maintain sliding contact with two (or more) bars. Current enters via one brush, flows through the contacted bar and the corresponding winding(s), and exits through the opposite brush. Because each brush may bridge adjacent bars, current shares multiple parallel paths; the configuration

depends on the winding scheme [39].

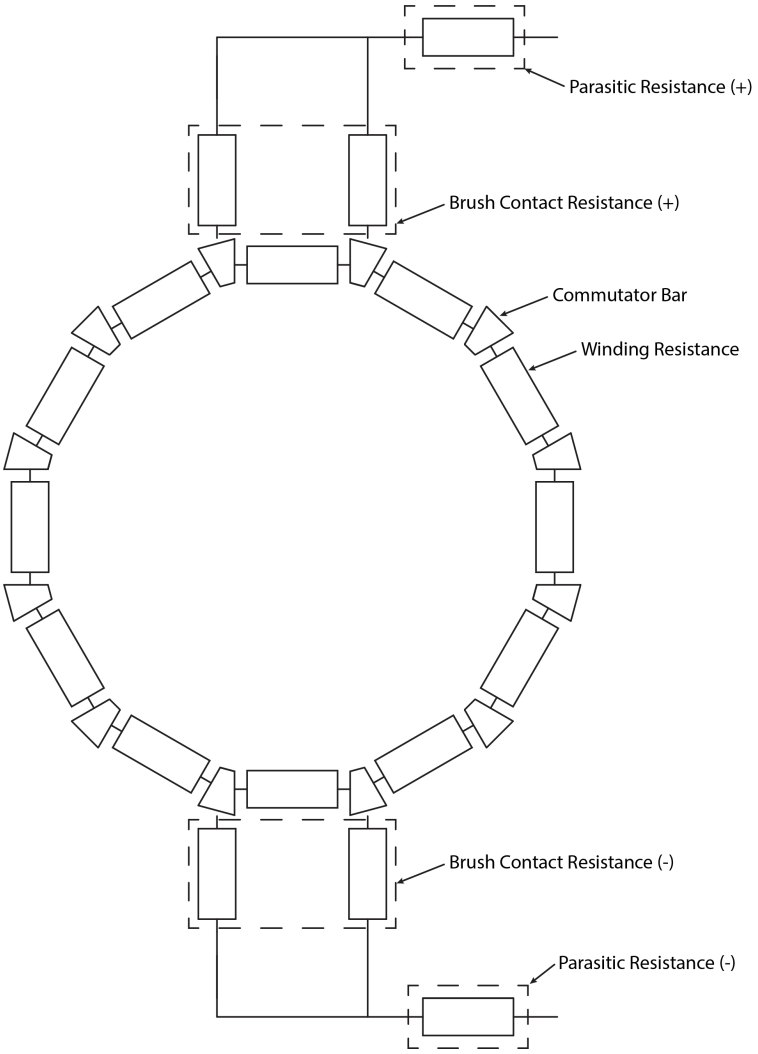


Figure 1.5: *Internal resistance arrangement for a brushed DC motor with a 12-segment commutator. The ring topology emphasizes cyclic structure.*

In the present case, the motor uses a lap winding. Each commutator segment connects to a single coil, forming loops that create several parallel current paths. This improves efficiency and helps uniform torque production. Figure 1.5 illustrates a simplified 12-segment topology. Figure 1.6 shows a 24-segment armature common in industrial designs.



Figure 1.6: *Photograph of a 24-segment armature showing winding arrangement and commutator construction.*

The resistance pattern across bars follows a cyclic relationship. Measuring between adjacent bars samples a single coil, while measuring across more distant bars includes multiple coils in series, peaking at the bar diametrically opposite the reference (for example, 12 bars apart in a 24-bar armature). A healthy lap winding yields a symmetric spectrum with a flat maximum at $N/2$ bars and repeats cyclically.

Different winding setups, such as wave winding, produce distinct profiles in their simplest forms. Wave winding can appear multi-peaked, while classic lap winding is more sawtooth-like and symmetric [40]. With many evenly spaced brushes, parallel paths flatten the spectrum and can mask underlying geometry. Diagnostic interpretation should therefore account for winding scheme and brush arrangement.

Over time, the resistance spectrum $R(\theta)$ changes with wear. Asymmetry often reflects debris accumulation, pitting, and uneven mechanical wear. These indicators

inform cleaning, brush replacement, or commutator resurfacing. Persistent arcing occurs where local contact resistance is elevated and conduction is briefly lost; armature inductance forces current to continue, producing a voltage spike and discharge. Discharges damage the surface and increase average resistance at affected segments. Tracking $R(\theta)$ captures these electrical and mechanical processes and supports preemptive maintenance before failure or performance loss.

Fully quantifying the rate of spectral change may require trending or data-driven models, but simple metrics such as the rate of resistance increase and the maximum local resistance already support maintenance planning. The Resistance Spectrum Method establishes a physics-based wear metric suitable for real-time control and predictive diagnostics and provides a natural foundation for future ML approaches.

Chapter 2

Controller Design

2.1 Dynamic Model of Brushed DC Motors

It is well established that, among the electrical parameters of a brushed DC motor, the resistance (R), particularly the brush and commutator contact resistance, drifts significantly with age and wear and can increase more substantially in specific sections of the motor [4, 6]. Before isolating and monitoring this parameter, it is essential to account for the other influential variables. The two primary parameters that define a complete electrical model are the back-EMF constant (k_v) and the armature inductance (L). These values are determined by the motor's construction and materials and, under normal operating conditions, remain effectively steady. This motivates the use of R as the primary wear-sensitive variable and the focus of diagnostic tracking.

The dynamic equations of a brushed DC motor follow directly from the classical electrical and mechanical models given in [6,39, Ch. 2; Ch. 9]. Applying Kirchhoff's Voltage Law (KVL) to the equivalent circuit and Newton's second law for rotation

yields the coupled system

$$V = L\,\frac{dI}{dt} + R\,I + k_v\,\omega, \tag{2.1}$$

$$T_{\mathrm{e}} = k_t I, \tag{2.2}$$

$$T_{\mathrm{e}} = J\frac{d\omega}{dt} + \lambda\omega + T_{\mathrm{l}}. \tag{2.3}$$

Here V is the terminal voltage, I the armature current, L the armature inductance, R the armature resistance (represented in Figure 2.1 as R_T to indicate the lumped brush and winding resistance), k_v and k_t the back-EMF and torque constants, ω the angular speed, λ the viscous friction coefficient, J the total inertia, and T_{l} the external load torque.

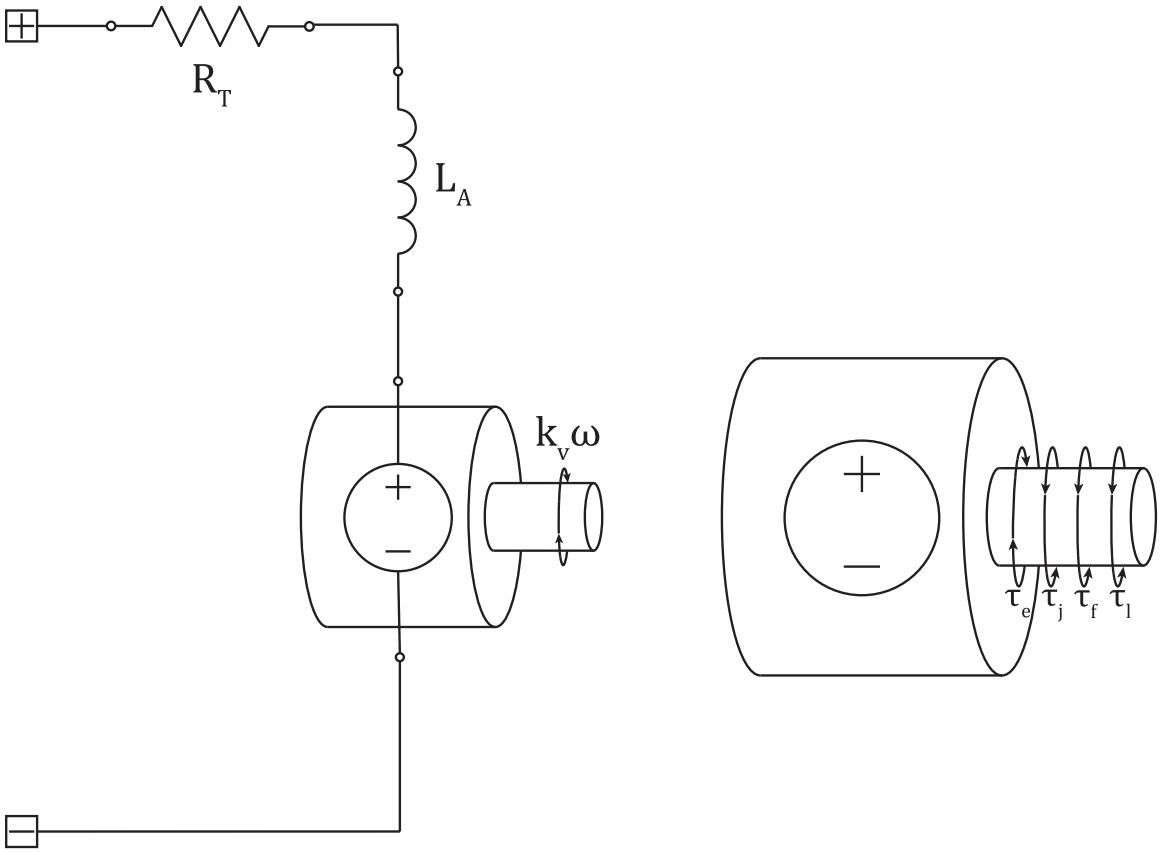


Figure 2.1: *Electrical and mechanical model of a brushed DC motor. The diagram shows the coupling between the voltage, current, and torque domains.*

Equation (2.1) represents the electrical dynamics obtained from KVL, while (2.2) defines the electromagnetic torque proportional to armature current. Equation (2.3) expresses the mechanical torque balance derived from Newton's law of rotation, where the developed electromagnetic torque T_{e} must overcome the inertial, frictional, and load torques, as illustrated in Figure 2.1. Combining these relations yields the system's time-domain form

$$\frac{dI}{dt} = \frac{1}{L}\big(V - RI - k_v\omega\big), \tag{2.4}$$

$$\frac{d\omega}{dt} = \frac{k_t}{J}I - \frac{\lambda}{J}\omega - \frac{T_{\mathrm{l}}}{J}. \tag{2.5}$$

Under nominal operating conditions, parameters such as k_t, k_v, L, λ, and J can be regarded as constant, since the air-gap flux and mechanical characteristics remain steady for a fixed excitation or permanent-magnet field. However, both theory and experiment indicate that elevated temperature and long-term aging can alter flux remanence and increase the effective armature resistance [6, Ch. 9]. Such effects are small within the intended control bandwidth but will be examined in future hardware testing to establish the limits of this assumption.

This fully parameterized model keeps each term physically interpretable. R captures wear-sensitive resistance, L the armature inductance, k_v and k_t the electromechanical conversion factors, λ the viscous friction, J the inertia, and T_{l} the external load torque. Such a formulation enables both predictive control and parameter estimation without resorting to abstract coefficients, while identifying R as the most sensitive and diagnostically valuable observable during normal operation.

2.2 Model Predictive Control for Brushed DC Motors

MPC is an advanced control strategy in which, at each control step, the controller forecasts the plant's response to a set of possible future input sequences and selects the one that minimizes a prescribed cost function. Unlike classical feedback controllers (e.g., PI or PID), MPC uses an internal model of the plant to explicitly plan control actions over a finite prediction horizon, allowing anticipation of disturbances and direct incorporation of system constraints such as voltage, current, or rate limits.

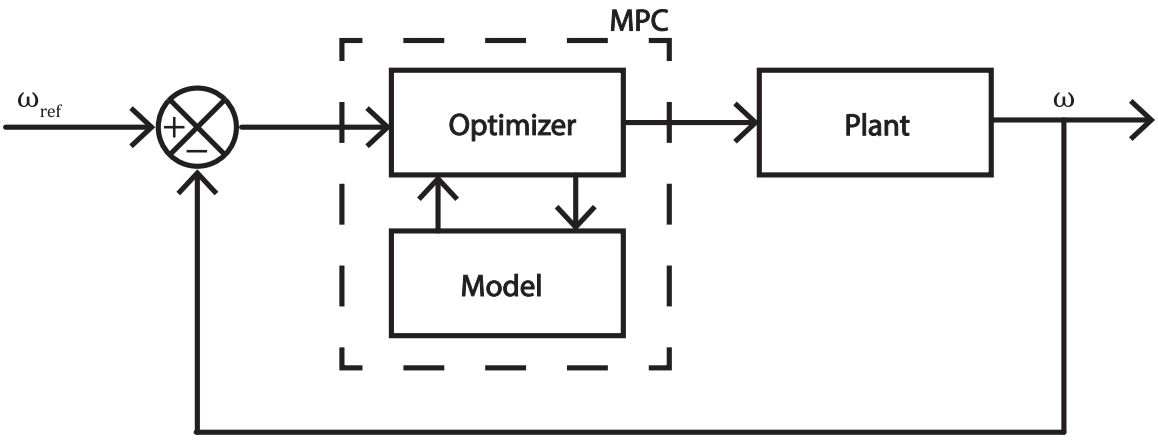


Figure 2.2: *General MPC architecture.*

A canonical discrete-time MPC formulation proceeds as follows [41]:

1. Measure or estimate the current system state $x[k]$.

2. For each candidate future input sequence

$$U = \{u[k],\, u[k+1],\, \ldots,\, u[k+N-1]\},$$

use the plant model to predict the corresponding sequence of future outputs

$$Y = \{y[k+1],\, y[k+2],\, \ldots,\, y[k+N]\}$$

over a prediction horizon of N steps.

3. Evaluate a cost function of the form

$$J = \sum_{i=1}^{N} \left\| y[k+i] - y_{\text{ref}}[k+i] \right\|_Q^2 + \sum_{i=0}^{N-1} \left\| u[k+i] - u[k+i-1] \right\|_R^2, \tag{2.6}$$

where

- $y[k+i]$ is the predicted output at future step $k+i$,
- $y_{\text{ref}}[k+i]$ is the reference trajectory to be tracked,
- $u[k+i]$ is the control input applied at step $k+i$,
- Q is a positive semidefinite weighting matrix penalizing tracking error,
- R is a positive definite weighting matrix penalizing rapid or excessive control movement, and
- $\|\cdot\|_Q^2 = (\cdot)^\mathsf{T} Q(\cdot)$ denotes a quadratic norm weighted by Q.

The first term enforces accurate reference tracking, while the second discourages abrupt control changes that could stress actuators or excite unmodeled dynamics.

4. Apply only the first control action $u[k]$ from the optimal sequence to the plant, discard the rest, and repeat this optimization at the next sampling instant (the *receding horizon* principle).

This framework provides a systematic means of handling multivariable interactions and actuator limits while maintaining predictive performance. Although it originated in process control, MPC has become increasingly common in motor-drive applications, where it offers improved transient shaping and constraint handling compared to conventional feedback designs.

Our Approach.

The controller developed here follows this same receding-horizon MPC structure: it repeatedly predicts the motor's response under candidate control sequences and applies the first action of the lowest–cost sequence. The key difference lies in the *cost function*. Because the drive already enforces a strict ramp-rate and quantization filter on the commanded voltage, explicit move penalties are unnecessary. At each actuation step the allowable voltage change is clipped to

$$|V[k] - V[k-1]| \leq S_{\max}\Delta t,$$

where $S_{\max}$ is the configured slew limit (e.g. 200 V/s) and Δt is the control update period; any requested change beyond this bound is saturated and then quantized to the hardware's minimum step. With this ramp limiter in place, overshoot and actuator stress are already mitigated.

A further practical motivation is that we want the controller's operating conditions during calibration to match those in normal use. Calibration is performed at the system's true operating maximum ramp rate so that identified parameters remain meaningful under real operating dynamics. Using artificially fast, "snappier" steps can slightly improve MSE in the model fit but would not reflect normal operation and is not strictly necessary for accurate parameter estimation.

Consequently, our implementation adopts a deliberately simple optimization cost that minimizes only the accumulated absolute speed-tracking error

$$J = \sum_{i=1}^{N} \Big| \, \omega[k+i] - \omega_{\text{ref}}[k+i] \, \Big| \tag{2.7}$$

over the prediction horizon. With a sufficiently long look-ahead, this objective yields smooth and stable control, although if the horizon is chosen too short, the controller

can exhibit chatter due to limited foresight. A more conventional MPC would augment this cost with quadratic move suppression and terminal penalties, but for this application these terms add complexity with little practical benefit, so the simplified form is retained.

Only a few studies have investigated MPC strategies for DC motor drives [42–44]. In [42], Nowak *et al.* propose a discrete-matrix formulation to represent plant dynamics, avoiding explicit model identification by operating directly on experimentally measured system responses. Luo *et al.* [43] and Iqbal *et al.* [44] extend this concept to permanent-magnet DC motors using finite control sets and observer-based feedback, illustrating the adaptability of MPC to various DC-fed topologies.

Dynamic Matrix Control (DMC) Approach

The first notable example [42] employs Dynamic Matrix Control (DMC), a classical variant of Model Predictive Control, for a brushed DC motor. The DMC algorithm is based on a step–response model of the plant, typically derived experimentally or from simulation, that describes how the system output responds to a unit step in the control input. Future outputs are predicted by convolving the planned sequence of input changes with these stored step–response coefficients.

More formally, the predicted output over a finite horizon can be written as

$$y(t) = g_1 \Delta u(t-1) + g_2 \Delta u(t-2) + g_3 \Delta u(t-3) + \cdots , \tag{2.8}$$

where g_i are the plant's step–response coefficients and $\Delta u(t-n)$ is the change in input at time $t-n$. For a prediction horizon N, these terms are often stacked in

matrix form [41]:

$$\begin{bmatrix} y[k+1|k] \\ y[k+2|k] \\ \vdots \\ y[k+N|k] \end{bmatrix} = \mathbf{S} \begin{bmatrix} \Delta u[k] \\ \Delta u[k+1] \\ \vdots \\ \Delta u[k+N-1] \end{bmatrix} + y_0, \tag{2.9}$$

where $\mathbf{S}$ is the *dynamic matrix* assembled from the step–response coefficients, and y_0 is the predicted free response if no new control actions are applied. In the standard formulation, DMC minimizes a cost function that penalizes future tracking errors, often expressed as the sum of squared deviations

$$J = \sum_{i=1}^{N} \Big(y[k+i|k] - y_{\text{ref}}[k+i] \Big)^2$$

Minimizing J with respect to the future input changes Δu yields the optimal control sequence. At each sampling instant, only the first element of this sequence is applied, and the optimization is repeated at the next step according to the *receding–horizon* principle. While other formulations may include additional weighting terms on control effort or rate of change, this pure tracking form is retained in the present work for simplicity and to maintain transparency in controller behavior.

Limitations of the Matrix-Based Approach.

DMC's strength is its simple matrix formulation for tracking arbitrary references and enforcing input/output constraints. However, because the model is reduced to a non-parametric step response, all physical parameters and disturbances are lumped together. This loss of structure makes it effectively impossible to monitor or adapt quantities such as the armature resistance R; any change in R merely appears as unexplained output deviation. In our application we require explicit knowledge of plant parameters, motivating a more physically structured modeling approach.

Another MPC formulation for brushed DC motors is given in [43]. Rather than using a matrix or state-space realization, they fit a Single-Input, Single-Output (SISO) transfer function using the MATLAB System Identification Toolbox. Their identified model couples the electrical and mechanical dynamics into a second-order transfer function of the form

$$\omega(s) = \frac{KV_{\mathrm{dc}}(s) - T_{\mathrm{L}}(s)(sL_a + R_a)}{s^2 L_a J + s(R_a J + L_a D) + (R_a D + K^2)}, \tag{2.10}$$

which, for the no-load case ($T_{\mathrm{L}}(s) = 0$), simplifies to

$$\frac{\omega(s)}{V_{\mathrm{dc}}(s)} = \frac{a}{s^2 + sb + c} \tag{2.11}$$

with lumped coefficients

$$\begin{aligned} a &= \frac{K}{L_a J}, \\ b &= \frac{R_a J + L_a D}{L_a J}, \\ c &= \frac{R_a D + K^2}{L_a J}. \end{aligned}$$

Although this retains some physical interpretability, the parameters are still aggregated and are not updated online.

Handling Disturbances

Another important consideration is how each approach treats unmodeled disturbances. In the SISO transfer-function formulation discussed above, any unexpected effects such as load-torque variation or parameter drift are absorbed into the $T_L(s)$ term (which is often not estimated online) or appear as output tracking error. If the DC bus voltage is not well regulated, this limitation becomes especially problematic,

since supply variations are not represented in the identified transfer function. During system identification, $T_L(s)$ is typically controlled or set to zero to ensure a proper fit. In the case of [42], the DC motor is modeled as a clean DC-fed system, and any mismatch between the real plant and the fitted model is treated as generic modeling error. Once deployed for control, the algorithm predicts future speed under the assumption of zero load torque and compensates observed deviation at the next step by adjusting the applied voltage or switch on-time. Although this behavior is often described as predictive control, it more closely resembles feed-forward modeling with rapid feedback correction than true disturbance-aware MPC, since T_{L} in practical systems is rarely known.

Another predictive control strategy was proposed in [43, 44], both introducing full-order state observers implemented as discrete-time Kalman filters to estimate unmeasured load-torque disturbances in brushed DC motors. Their motor models are expressed in discrete state-space form:

$$\mathbf{x}(k+1) = \mathbf{A}_d\,\mathbf{x}(k) + \mathbf{B}_d\,u(k) + \mathbf{v}(k), \tag{2.12}$$

$$\mathbf{y}(k) = \mathbf{C}_d\,\mathbf{x}(k) + \mathbf{w}(k), \tag{2.13}$$

where the state vector $\mathbf{x} = [\,\omega,\, i_a,\, T_L\,]^T$ includes angular velocity, armature current, and load torque (modeled as a constant or slowly varying state). $\mathbf{v}(k)$ and $\mathbf{w}(k)$ represent process and measurement noise, respectively. The Kalman filter recursively updates the state estimate as

$$\hat{\mathbf{x}}(k|k) = \hat{\mathbf{x}}(k|k-1) + \mathbf{K}\Big[\mathbf{y}(k) - \hat{\mathbf{y}}(k|k-1)\Big], \tag{2.14}$$

where $\hat{\mathbf{x}}(k|k-1)$ is the predicted state, $\hat{\mathbf{y}}(k|k-1)$ the predicted output, and $\mathbf{K}$ the observer gain determined from the assumed noise statistics.

These observer structures successfully track load-torque disturbances and im-

prove speed regulation under varying loads. However, they require full state-space representations that include both the electrical and mechanical equations, and they assume that the applied input voltage $u(k)$ is accurately known and relatively noise-free, meaning a well-behaved DC bus.

Our Design Goals

In our application we seek to address two practical issues that remain challenging in the prior work:

1. **Time-varying resistance.** We aim to treat R as a slowly varying parameter that can be identified and tracked online, enabling condition monitoring without requiring extensive external instrumentation or offline identification.

2. **AC-fed dynamic correction.** To the best of the authors' knowledge, no prior research has demonstrated MPC of brushed DC motors under AC-fed supply conditions. Our approach explicitly targets this use case while remaining practical to implement with modest hardware.

Our method remains physics-based and model-predictive, but is designed to be *lightweight and interpretable*. It draws inspiration from Kalman filtering in its use of online state correction, yet avoids the full tuning complexity of a high-order observer. In particular, if a Kalman filter (or other advanced observer) were layered on top of our framework, the armature resistance R could be treated as an additional state; however, this would introduce tuning challenges because R and the disturbance term d evolve on significantly different time scales. By contrast, the proposed scheme offers straightforward parameter adaptation and disturbance estimation with minimal tuning effort.

Note. The architectural changes introduced here also establish the foundation for the *Resistance Spectrum Method* developed in the following chapter. By enabling reliable,

ripple-compensated tracking of R, a similar framework can be extended to extract a position-resolved resistance profile $R(\theta)$, providing both diagnostic and predictive value under controlled startup or recalibration conditions.

2.3 Proposed MPC Controller

The drive employs a receding-horizon MPC with a simple absolute-error cost on speed and hard actuation constraints imposed by ramp and quantization limits. To preserve physical interpretability and enable online parameter visibility, the controller incorporates a *two-channel backsolver observer*: a fast mechanical disturbance estimator for d and a phase-gated electrical resistance estimator for R.

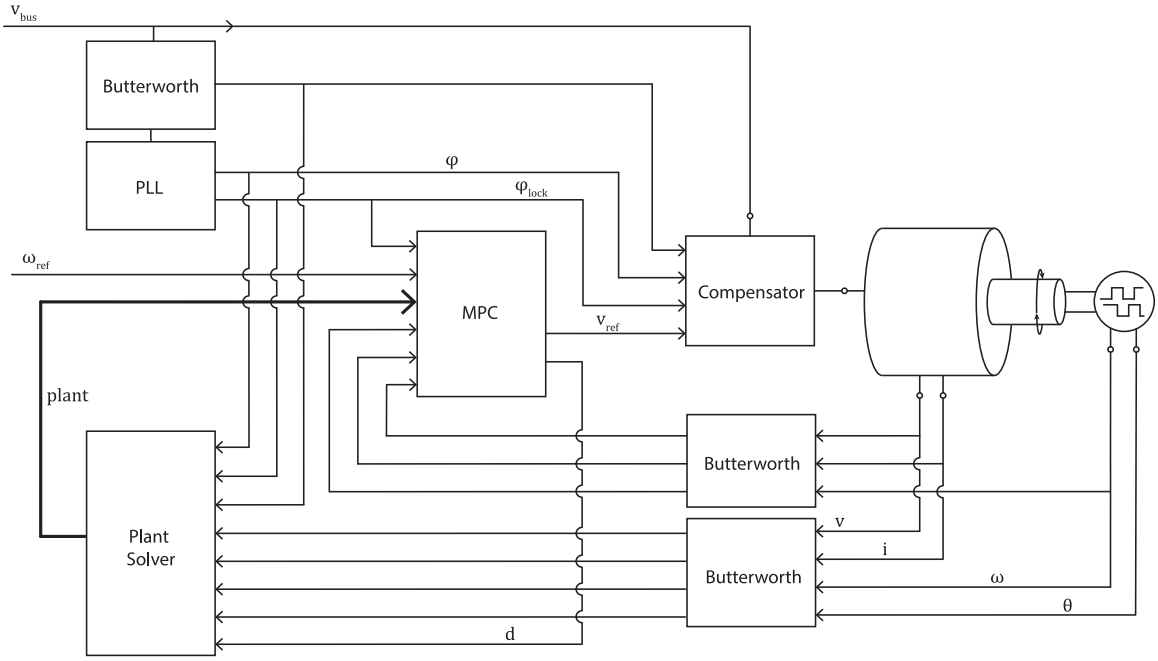


Figure 2.3: *Overall control architecture and layout of the proposed drive.*

The block diagram in Fig. 2.3 illustrates the overall control structure of the proposed drive. This section focuses on the MPC and PLL layers, while the following chapter details the adaptive plant solvers. Here, φ denotes the instantaneous electrical phase reference generated by the ripple-synchronous PLL, while φ_{lock} rep-

resents the lock-valid signal that enables the controller to transition from open-loop to synchronized operation.

2.3.1 Prediction Model, Cost, and Actuation Limits

The predictive model is derived from a discretized form of the DC motor dynamics using a forward–Euler step of period Δt. The instantaneous rate of current change is given by

$$\dot{I}_k = \frac{1}{L}\big(V_k - R\, I_k - k_v\, \omega_k\big),$$

which, when discretized, yields the state-update equations used within the prediction horizon:

$$I_{k+1} = I_k + \Delta t\, \dot{I}_k, \tag{2.15}$$

$$\omega_{k+1} = \omega_k + \Delta t\left(\frac{k_t}{J}\, I_k - \frac{\lambda}{J}\, \omega_k + d\right). \tag{2.16}$$

Here I_k and ω_k denote the armature current and angular speed at the k-th prediction step. Candidate voltage sequences are evaluated using the absolute speed-tracking cost introduced in (2.7). No additional move penalty is applied, since the commanded voltage is already constrained by ramp and quantization limits:

$$|V[k] - V[k-1]| \le S_{\max}\, \Delta t, \qquad V[k] \in \{V_{\min},\, V_{\min}{+}V_{\text{step}},\, \ldots,\, V_{\max}\}.$$

2.3.2 Two-Channel Backsolver Observer

Fast Disturbance Backsolver d (Mechanical Channel)

At each planning instant, defined as one *control interval* composed of a fixed number of base control edges at 12,000 Hz, the controller estimates a *constant* disturbance d acting over the most recent window of buffered samples. For example, if the control

interval spans 20 base edges, the planning rate is 12,000/20 = 600 Hz. All buffers used for fitting advance at this same planning rate.

Using the discrete mechanical update from (2.16), the one-step prediction with a candidate d is

$$\omega_k^{\text{pred}}(d) = \omega_{k-1} + \Delta t\left(\frac{k_t}{J}\, I_k - \frac{\lambda}{J}\,\omega_{k-1} + d\right),$$

where ω_k is the measured rotor speed, I_k is the armature current, k_t is the torque constant, J is the total rotor–load inertia, and λ represents viscous damping. The unknown term d acts as an equivalent bias torque that captures external load, static friction, or other unmodeled forces.

From the rolling buffers $\{\omega_k, I_k\}$, the observer minimizes the absolute-error objective

$$J_d(d) = \sum_{k=2}^{B} \Big|\, \omega_k^{\text{pred}}(d) - \omega_k \,\Big|,$$

over a discrete grid of candidate values to find

$$d^\star = \arg\min_d J_d(d).$$

Purpose within the planning cycle. This backsolve step is executed before each new MPC planning horizon begins. Its purpose is to determine the disturbance torque d_{obs} that best explains the measured evolution of rotor speed ω_k given the recent history of applied current and voltage. In other words, it finds the steady mechanical bias that aligns the model's predicted ω_k^{pred} with the measured ω_k. By solving for this bias, the controller refreshes its internal model to reflect actual load conditions before generating the next voltage trajectory. Without first updating d_{obs}, the optimizer would assume a perfectly matched model, causing systematic tracking offsets between predicted and observed speed. Real-time estimation of d_{obs} recenters the prediction model so that the MPC plans around the true instantaneous operating point of the drive.

Anti-chatter guards. Two criteria prevent overreaction to noise and small numerical fluctuations:

$$\text{(i) } J_d(0) - J_d(d^\star) \geq \epsilon_{\text{imp}},$$

$$\text{(ii) } \left| J_d(d^\star) - J_d^{\text{last}} \right| \geq \epsilon_{\text{cons}}.$$

If either condition fails, the estimate reverts to $d_{\text{obs}} = 0$; otherwise $d_{\text{obs}} \leftarrow d^\star$ and the stored cost J_d^{last} is updated. This yields a lightweight, windowed estimate of the disturbance torque that explains the observed ω_k trajectory while maintaining stability between planning cycles.

Phase-Gated Resistance Backsolver R (Electrical Channel)

While the disturbance estimate d_{obs} accounts for mechanical bias, the armature resistance R must be identified under ripple-rich electrical conditions. Because the rectified DC bus voltage rises and falls each half-cycle, instantaneous voltage–current (V–I) samples can be distorted if taken at arbitrary times. To reduce this distortion, the controller samples R near the *minima* of the rectified ripple where the bus slope is small and the armature current changes most slowly. A PLL continuously tracks the ripple phase $\phi(t)$, so sampling can be synchronized to these minima.

The R backsolver operates at a slower rate than the disturbance backsolver. While d_{obs} updates every control interval, R_{obs} is refreshed only after a full set of phase-aligned samples has been accumulated. This staggered timing allows the resistance estimate to represent long-term electrical drift rather than short-term ripple or switching transients. During steady-state operation, the same mechanism performs a slow bias correction. When the current and speed remain nearly constant, the controller verifies Ohmic consistency according to

$$V_{\text{bus,used}} \approx L\,\dot{I} + R_{\text{obs}}\,I + k_v\,\omega,$$

and adjusts the effective command voltage whenever a small residual bias is detected. In this way, R_{obs} functions both as a diagnostic estimate and as a steady-state correction term that accounts for gradual electrical aging or temperature-related drift.

The sampler operates on a decimated phase basis. It enables data collection once every fixed number of control intervals and records a sample when the instantaneous phase $\phi(t)$ crosses a predefined target $\phi_\star$ near the ripple minimum, within a dynamic acceptance window,

$$|\phi - \phi_\star| \le \max\Big(\phi_{\min},\ \tfrac{1}{2}|\Delta\phi_{\mathrm{tick}}| + \phi_{\mathrm{margin}}\Big),$$

where $\Delta\phi_{\mathrm{tick}}$ is the phase increment per control edge. The parameters $\phi_{\min}$ and ϕ_{margin} set the minimum and adaptive angular tolerances so that each electrical sample is collected in a stable segment of the ripple cycle.

When a valid crossing occurs, the controller records the tuple

$$\big(V_k,\ I_{k-1},\ I_k,\ \omega_k,\ t_k\big), \qquad \dot{I}_k \approx \frac{I_k - I_{k-1}}{\Delta t},$$

and appends it to a circular buffer of N_R phase-aligned samples. Here V_k is the measured terminal voltage, I_k is the armature current, ω_k is the measured speed, and L and k_v are the known inductance and back-EMF constants. After N_R samples have been collected, the effective resistance is identified from

$$V_k \approx L\,\dot{I}_k + R\,I_k + k_v\,\omega_k,$$

by minimizing the absolute residual

$$J_R(R) = \sum_{k=1}^{N_R} \Big|\, V_k - (L\,\dot{I}_k + R\,I_k + k_v\,\omega_k)\,\Big|.$$

The minimizing value

$$R^\star = \arg\min_R J_R(R)$$

represents the resistance that best explains the instantaneous electrical behavior of the motor over the sampled window. An update is accepted only if $J_R(R_{\text{prev}}) - J_R(R^\star) \geq \epsilon_{\text{imp}}$; otherwise the previous estimate R_{obs} is retained to prevent noise-induced oscillation.

Purpose within the identification loop. This backsolver determines the electrical parameter that makes the observed voltage–current relationship consistent with the motor's dynamic model. Just as d_{obs} explains the measured evolution of the rotor speed ω_k, R_{obs} explains the voltage residual that cannot be attributed to inductance or back-EMF. Because it operates more slowly and with phase-synchronous gating at ripple minima, it captures gradual resistance trends due to brush wear, commutator aging, or temperature variation without being affected by fast switching dynamics.

Interpreting R_{obs} and d_{obs}. The two observers operate independently within each planning cycle. The mechanical channel provides a fast bias correction d_{obs} (units of rad/s^2), while the electrical channel produces a slower, phase-synchronous R_{obs} (units of Ω). Because R_{obs} samples at ripple minima and d_{obs} fits over short sliding windows, they serve complementary purposes: one maintains electrical consistency and the other compensates mechanical disturbance.

Illustrative Snapshot and Note on $\dot{I}_{\text{fast}}$.

The following runtime excerpt shows a typical controller interval used for backsolver estimation. It is not part of the planner itself but records how each observer's buffer appears during normal operation. The symbol $\dot{I}_{\text{fast}}$ is a per-tick finite-difference estimate of current slope at the base control rate of 12,000 Hz. Its numeric values

can appear large because they are expressed per second; however, the actual change in current per tick, $\Delta I \approx \dot{I}_{\text{fast}} \Delta t$, is small since $\Delta t \approx 83\,\mu$s. Sampling near ripple minima further reduces $|\dot{I}_{\text{fast}}|$, which improves the conditioning of the R fit.

Mechanical buffer (fast disturbance backsolver d): Columns list the sample index (`Idx`), timestamp (`Time`), measured current (`Current`), measured speed (Ω), predicted speed (Ω_{pred}), instantaneous prediction error (`PredErr`), numerical speed derivative ($\dot{\Omega}$), and the most recent disturbance estimate (d_{obs}).

Electrical buffer (phase-gated resistance backsolver R): Columns show the ordered sample index (`Ord`), timestamp (`Time`), previous and current armature currents ($I_{\text{prev}}, I_{\text{now}}$), numerically estimated current derivative ($\dot{I}_{\text{fast}}$), measured and modeled voltages ($V_{\text{now}}, V_{\text{model}}$), and the most recent resistance estimate (R_{obs}).

Idx	Time [s]	Current	Ω	Ω_{pred}	PredErr	$\dot{\Omega}$	d_{obs}
1	0.76913	3.86869	199.99022	–	–	-6.99243	–
2	0.76922	3.75346	199.98959	199.98971	0.00013	-7.57993	-1.544
… (rows 3–40 omitted)							

Total abs prediction error, $d = 0$: 0.00279

Total abs prediction error, best d: 0.00279

No d found with sufficient improvement (0.00000 < min 0.00400), using $d = 0$

Ord	Time [s]	I_{prev}	I_{now}	$\dot{I}_{\text{fast}}$	V_{now}	V_{model}	R_{obs}
1	0.12682	276.63838	276.55350	-1014.07606	20.37192	20.39345	0.019922
2	0.16030	280.14410	280.42748	3385.58614	27.36077	27.33368	0.020097
… (rows 3–20 omitted)							

Total abs residual, $R = 0.02000$: 0.332273

Total abs residual, best R: 0.332273 ($R^\star = 0.02000$)

[0.77240] R update: no sufficient improvement (0.000000 < 0.004000), retaining prior R

Table 2.1: *Comparison of MPC based control methods for brushed DC motors.*

Method	Parameter Estimation	Control Strategy	Contribution / Limitations	Adaptivity
DMC [42]	Fixed parameters from datasheet or offline step response; no explicit physical plant or online R estimation.	Dynamic Matrix Control using precomputed input–output step-response data.	Early MPC variant without a known physical plant; relies on empirical response mapping. Provides disturbance rejection via feedback but lacks adaptive parameter updates. Developed for nominal DC-fed operation.	Low: disturbance rejection via feedback only; no adaptive parameter update.
FCS MPC [43]	Offline regression for L, R, k_v, J, and λ; no recursive update.	Finite control set MPC (chopper state selection) based on cost minimization.	Hardware demonstration showing strong performance under steady DC supply; requires recalibration for drift and does not explicitly address ripple or supply variation effects.	Medium: adaptation requires offline recalibration.
FCS MPC + Observer [44]	Offline plant identification; R not updated online.	FCS MPC with a discrete-time Kalman filter observer for speed, current, and load torque.	Improves noise immunity and load-torque estimation, but plant parameters remain fixed after calibration; does not explicitly consider operation under ripple-fed or AC-supplied conditions.	Medium: observer adapts to noise and disturbances but not to plant drift.
Proposed (This Work)	Plant model established during any short calibration window with known torque; rolling-buffer regression then continuously tracks disturbance and resistance drift (R).	Adaptive model-based MPC with a two-channel backsolver observer (fast mechanical disturbance d and phase-gated electrical resistance R).	Continuously tracks electrical and mechanical parameters for health monitoring and adaptive control. PLL-synchronized sampling and bus-aware correction stabilize operation under DC and ripple-fed AC supplies; additional calibration windows can be entered as needed.	High: adaptive after calibration; disturbance and R drift compensated online during operation.

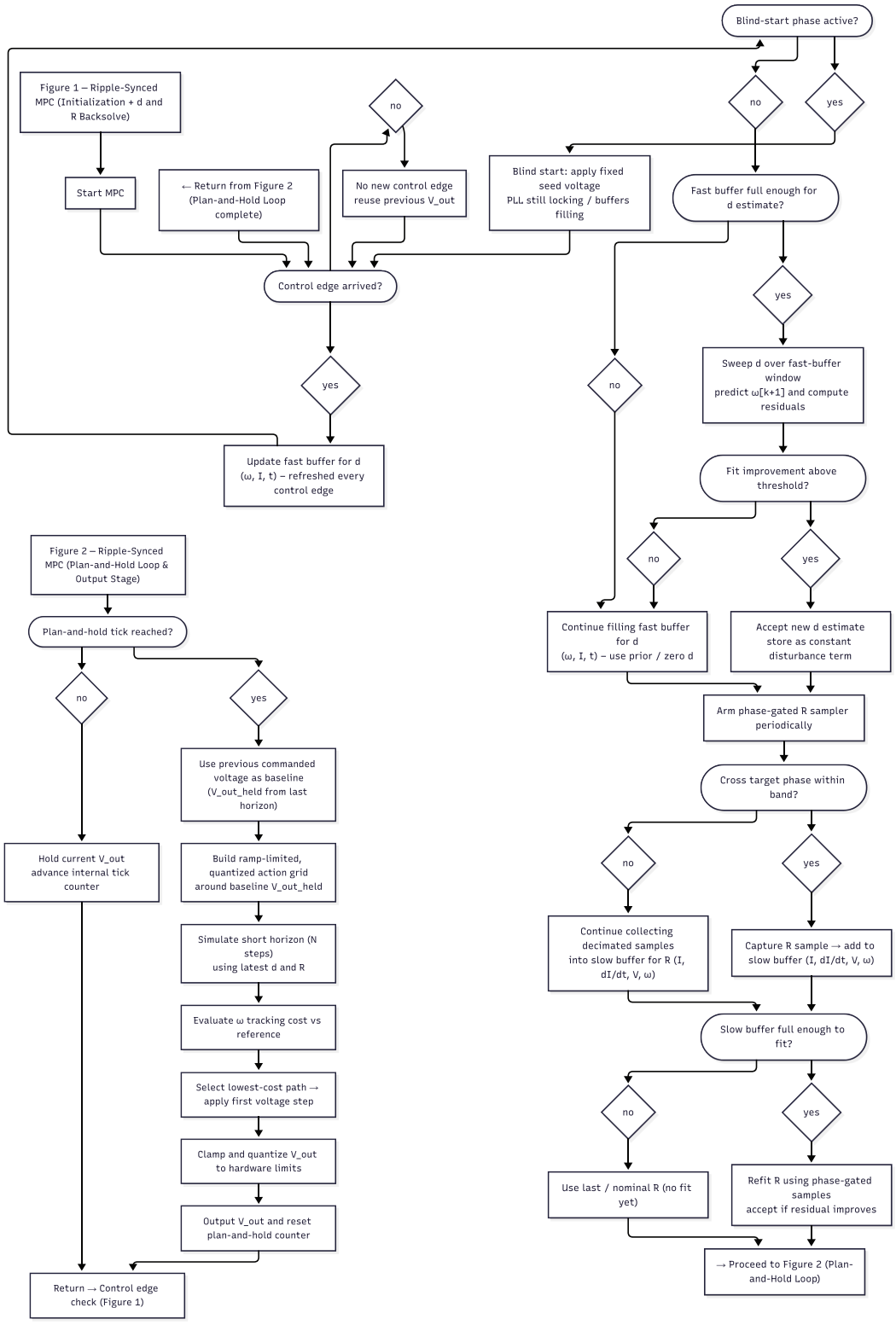


Figure 2.4: *MPC layer implementing receding-horizon speed control, using disturbance and resistance observers that supply d each control step and R once per phase-aligned window to the predictive model.*

2.4 Phase-Locked Loop

2.4.1 Ripple PLL and Phase-Synchronous Scheduling

The rectified DC bus exhibits a dominant 120 Hz ripple originating from the AC mains supply. To synchronize both the inner voltage compensator and the outer MPC layer with this periodic sag, the controller implements a custom PLL that detects the ripple's minima and maintains a continuous electrical phase reference. This reference enables the controller to align duty updates, sampling events, and model-fitting windows with the true bus condition in real time.

Motivation and observed limitations. During early testing, several conventional PI-type PLLs were evaluated for tracking the 120 Hz ripple on the rectified DC bus. These classical structures, as described by Gardner [45], employ continuous feedback using a phase detector, loop filter, and controlled oscillator. Although effective for sinusoidal or line-synchronous signals, these loops became unstable under the present conditions. In particular, when the drive entered steady state and the current momentarily dipped during commutation, the rectified waveform exhibited small rebounds that appeared as false zero crossings to a traditional PLL. This led to erratic phase corrections, jitter in the 120 Hz reference, and transient loss of lock even though the underlying ripple frequency remained constant.

To eliminate this sensitivity, the present design replaces continuous feedback with a discrete, geometry-based phase extraction method. The PLL defines phase by directly identifying the *local minima* of the rectified waveform rather than integrating continuous phase error. This event-driven structure removes the need for a proportional–integral loop filter altogether, making the phase reference immune to amplitude drift, load-dependent asymmetry, and the largely different steady-state current characteristic of DC motor operation. Each confirmed minimum therefore

represents a clean physical event that corresponds to the instant when the AC source transitions from discharge back into recharge.

2.4.2 Windowing and Minima Confirmation.

The PLL operates at the system's base simulation step of $\Delta t = 5 \times 10^{-7}$ s (equivalent to 2 MHz), which provides sufficient temporal resolution to capture the fine detail of the rectified waveform during modeling. In practical hardware, this level of granularity is unnecessary for control or observer updates, since the relevant dynamic content occurs at much lower frequencies. To reduce computational load within simulation, the more intensive minima check executes only once every `DS_SKIP` samples. The corresponding downsampled frame interval is

$$\Delta t_{\text{ds}} = \texttt{DS_SKIP}\,\Delta t,$$

yielding $\Delta t_{\text{ds}} = 20$ μs for `DS_SKIP`$= 40$, or a 50 kHz window update rate. This rate is still fast enough to resolve the 120 Hz ripple with high temporal precision.

At runtime, the loop observes the measured bus voltage $V_{\text{bus}}(t)$ and computes its one-step difference

$$\Delta V_{\text{bus}}(t) = V_{\text{bus}}(t) - V_{\text{bus}}(t - \Delta t),$$

to identify slope reversals. A negative-to-positive transition of ΔV_{bus} indicates that a local minimum may be approaching, at which point the PLL "arms" a short inspection window centered on the candidate sample time t_c. A circular buffer stores the most recent V_{bus} values:

$$\mathcal{W}(t_c) = \Big\{ V_{\text{bus}}(t_c + m\Delta t_{\text{ds}}) \;:\; m = -M, \ldots, M \Big\},$$

where $2M+1$ = `WIN_SIZE` defines the total window length. The midpoint of this buffer corresponds to the current sample, allowing the loop to test whether the true minimum lies at the window's center. The parameters governing this process are summarized in Table 2.2.

Table 2.2: *Key configuration parameters for the windowed-minima detector.*

Parameter	**Symbol / Variable**	**Description**
`DS_SKIP`	Δt_{ds}	Downsample interval; number of base samples (Δt) skipped before each window update.
`WIN_SIZE`	$2M+1$	Total samples in the sliding buffer used for minima confirmation.
`MID`	M	Index of the buffer midpoint tested as the candidate minimum.
`N_MINIMA`	N_{hist}	Length of the historical minima buffer used for frequency estimation.
`FREQ_TICK_MIN/MAX`	$f_{\min}$, $f_{\max}$	Accepted ripple-frequency band for valid minima pairs (119–121 Hz).

A point t_c is accepted as a valid minimum only if

$$V_{\text{bus}}(t_c) = \min_{m\in\{-M,\dots,M\}} V_{\text{bus}}(t_c + m\Delta t_{\text{ds}}) \quad \text{and} \quad \Delta^- < 0,\ \Delta^+ \geq 0,$$

where the backward and forward voltage slopes are defined as

$$\Delta^- = V_{\text{bus}}(t_c) - V_{\text{bus}}(t_c - \Delta t_{\text{ds}}), \qquad \Delta^+ = V_{\text{bus}}(t_c + \Delta t_{\text{ds}}) - V_{\text{bus}}(t_c), \tag{2.17}$$

and the sign reversal $\Delta^- < 0,\ \Delta^+ \geq 0$ indicates a local minimum at t_c. Once confirmed, the timestamp $t_{\min}^{(k)}$ of the minimum is stored in a short history buffer.

Window delay and timing precision. Although the window spans a finite number of samples, the resulting delay is negligible. Since the decision is evaluated at the

window center, the effective confirmation latency equals half the window duration:

$$\tau_{\text{win}} \approx \tfrac{1}{2}\texttt{WIN_SIZE}\,\Delta t_{\text{ds}}.$$

With $\texttt{WIN_SIZE} = 100$ and $\Delta t_{\text{ds}} = 20\ \mu$s, this amounts to $\tau_{\text{win}} \approx 1.0$ ms, about 12% of the 8.33 ms half-cycle at 120 Hz. Because the PLL timestamps the event at the *center* of the window ($t_{\min}^{(k)} = t_c$), subsequent timing is extrapolated forward with continuous phase synthesis, eliminating any observable lag. The outer 12 kHz controller therefore operates in real time, phase-locked to the actual bus minima detected at the 50 kHz window rate.

Period and frequency estimation. From the list of recent minima $\{t_{\min}^{(k)}\}$, the PLL estimates the ripple period as the mean of all pairwise time differences within a narrow acceptance band:

$$f_{\min} \le \frac{1}{\Delta T_{ij}} \le f_{\max}, \qquad \Delta T_{ij} = t_{\min}^{(i)} - t_{\min}^{(j)}.$$

The bounds used in this study are $f_{\min} = 119$ Hz and $f_{\max} = 121$ Hz, chosen as representative limits for a nominal 120 Hz supply; in practical hardware, tighter limits would typically be applied. The accepted intervals yield the current period and frequency estimates,

$$T_{\text{est}} = \text{mean}\{\Delta T_{ij}\}_{\text{valid}}, \qquad f_{\text{est}} = 1/T_{\text{est}}.$$

Once several consecutive estimates converge within the tolerance band, the loop declares lock and records the most recent minimum t_{lock} and corresponding period T_{lock}.

Lock and phase synthesis. When locked, the loop synthesizes a continuously advancing electrical phase

$$\phi(t) = 2\pi \frac{t - t_{\text{lock}}}{T_{\text{lock}}} \bmod 2\pi, \tag{2.18}$$

with $\phi = 0$ at each detected bus minimum. The phase is broadcast to two consumers: (1) the **inner bus-voltage loop**, which modulates duty cycles to cancel the 120 Hz sag in real time, and (2) the **outer MPC**, which uses $\phi(t)$ to align its sampling of the electrical channel (the R backsolver) and ensure that all phase-gated measurements occur at consistent ripple positions.

Dual tick generation. Two synchronized timing streams are produced from the locked phase:

- The *period tick*, emitted every T_{lock}, serves as the 120 Hz heartbeat used by higher-level controllers.
- The faster *in-cycle tick*, generated at intervals of $T_N = \texttt{delay_frac}\, T_{\text{lock}}$, defines multiple sub-phase events within each ripple cycle. This spacing allows modulation or sampling several times per half-cycle, always at repeatable phase offsets.

Both tick streams are generated at the raw control rate, and the 12 kHz controller cadence is derived from these phase ticks rather than from fixed timing, ensuring phase-aligned updates without quantization error.

Phase continuity and controller enable. A controller-on flag is asserted only after a valid lock is achieved. During lock, both the inner and outer loops advance synchronously on the same phase grid. If the ripple signal becomes unreliable, the PLL maintains the previous cadence until a new stable frequency is confirmed, preventing

transient phase jumps that could destabilize the drive.

Relock and handoff. If the detected ripple frequency drifts outside the established frequency band, the PLL continues operating at the previous cadence while collecting new minima. When enough valid intervals have accumulated, it schedules a handoff at the next confirmed minimum. At that instant, the loop updates both the locked period and the in-cycle tick spacing together, ensuring that the new 120 Hz tick and the first sub-phase tick begin on the same boundary. This preserves phase continuity and prevents duplicate or missing ticks during frequency adjustments.
Table 2.3 summarizes the principal PLL signals and their meanings.

Table 2.3: *Principal PLL signals and their functional roles.*

Symbol / Variable	**Description**
V_{bus}	Measured rectified DC bus voltage containing the 120 Hz ripple.
ΔV_{bus}	One-step voltage difference used to detect slope reversals and identify minima.
$t_{\text{min}}^{(k)}$	Timestamp of the k-th confirmed bus-voltage minimum.
T_{lock}, f_{lock}	Locked period and frequency estimates (nominally 120 Hz).
$\phi(t)$	Continuous electrical phase, wrapped to $[0, 2\pi)$, defining position within each ripple cycle.
`delay_frac`	Fraction of the ripple period defining sub-phase tick spacing (T_N).
Period tick	120 Hz event providing the primary synchronization heartbeat.
In-cycle tick	Faster sub-phase event used for duty-cycle modulation and phase-gated sampling.
`controller_on`	Logical flag indicating that a valid lock has been achieved.

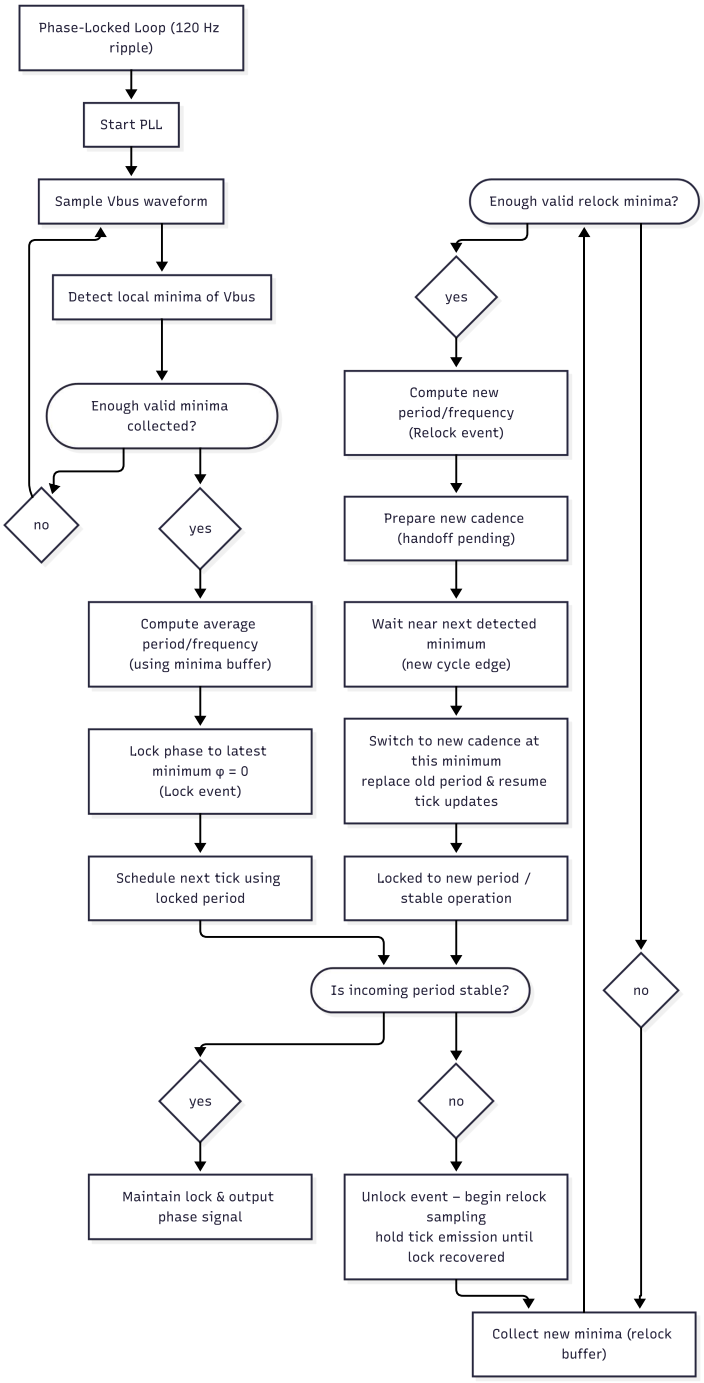


Figure 2.5: *Simplified control flow of the rectified-bus phase-locked loop. The PLL tracks the minima of* V_{bus}*, estimates the* 120 Hz *period, and synthesizes a continuous phase* $\phi(t)$ *used for all ripple-synchronous events.*

2.5 Inner Control Loop (Fitted Bus + Adaptive Phase)

Using the phase from the PLL we propose a ripple–synchronous inner voltage regulator that (i) models the rectified bus with a finite Fourier series, (ii) aligns that model to the measured ripple via an *adaptive phase offset* $\phi_{\text{raw}}(t)$, and (iii) computes a duty command so that the *motor terminal voltage* tracks the outer MPC request V_{cmd}.

Bus model and phase alignment. The rectified DC bus is modeled as a finite Fourier series

$$\hat{V}_{\text{bus}}(\theta) = V_0 + \sum_{k=1}^{K}\Big(a_k \cos k\theta + b_k \sin k\theta\Big), \tag{2.19}$$

with coefficients identified either online when the adaptive controller runs (discussed in the next section) or offline and stored in `BUS_COEFFS`. During operation, the loop evaluates this model at a *shifted* electrical angle

$$\theta_{\text{use}}(t) = \phi(t) + \phi_{\text{raw}}(t), \qquad V_{\text{bus,used}}(t) = \hat{V}_{\text{bus}}\Big(\theta_{\text{use}}(t)\Big), \tag{2.20}$$

where $\phi(t)$ is the PLL phase defined in (2.18) and $\phi_{\text{raw}}(t)$ is the adaptive offset estimated by the inner loop.

Evaluating the bus model at this shifted phase allows the duty calculator to anticipate the instantaneous bus ceiling as it approaches, rather than relying on a static fit or a noisy direct sample.

The adaptation of ϕ_{raw} occurs only on PLL N-tick edges; that is, it executes once every subdivided ripple interval when the PLL emits its internal N-*tick* timing signal (in our implementation, N is chosen so that the tick rate is about 600 Hz, providing several control updates per 120 Hz bus cycle). Between these ticks, the phase offset is integrated continuously from the current commanded rate.

2.5.1 Duty Computation and Quantization

Given the outer command $V_{\text{cmd}} \geq 0$, the ideal continuous PWM duty fraction would be

$$d_{\text{tgt}}(t) \;=\; \text{sat}_{[0,\,1-\varepsilon]}\left(\frac{V_{\text{cmd}}}{V_{\text{bus,used}}(t)}\right),$$

where a small headroom ε prevents the commanded duty from ever reaching 100%, ensuring the PWM remains realizable.

In practice, the duty cycle cannot vary continuously; it changes in discrete increments determined by the PWM carrier and the controller's update rate. As discussed by Buso and Mattavelli [46], digital PWM implementations inherently quantize the duty ratio according to the timer resolution and update frequency. If the PWM frequency is f_{pwm} (period $T_{\text{pwm}} = 1/f_{\text{pwm}}$) and the inner loop runs every dt, then during one PWM period the controller can update the compare register approximately

$$N_{\text{steps}} \;=\; \left\lfloor \frac{T_{\text{pwm}}}{dt_{\text{eff}}} \right\rfloor, \qquad dt_{\text{eff}} = \max(dt,\, 1\ \mu\text{s}),$$

where the code enforces a **1** μs minimum step size for numerical stability and to prevent an unrealistically large number of substeps if an excessively small dt is passed in. Each update corresponds to one discrete step of duty fraction

$$\Delta d \;=\; \frac{1}{N_{\text{steps}}} \;=\; \frac{1}{\left\lfloor \frac{1/f_{\text{pwm}}}{dt_{\text{eff}}} \right\rfloor}.$$

Thus Δd is the smallest raw change in duty cycle the controller can make in a single update. For example, with $f_{\text{pwm}} = 5$ kHz ($T_{\text{pwm}} = 200\ \mu$s) and $dt = 0.5\ \mu$s, the code clamps to $dt_{\text{eff}} = 1\ \mu$s, giving $N_{\text{steps}} = 200$ and $\Delta d = 0.005$ (0.5% duty steps).

The controller also reports a minimum resolvable output-voltage increment

$$V_{\text{min_step}} = V_{\text{bus,max}} \cdot \Delta d_{\text{eff}},$$

where $V_{\text{bus,max}} \approx 83.2$ V and Δd_{eff} is the effective duty step after dithering. With the present parameters this yields $V_{\min_step} \approx 0.020$ V.

Dithering for Finer Resolution.

The smallest usable duty increment is limited not only by the digital timer but also by the **power device and gate driver**: each MOSFET or IGBT requires a finite turn-on and turn-off time, set by its total gate charge and the driver's peak current. Pulses shorter than this minimum on-time will not fully enhance the device and lead to excessive switching losses.

To achieve an effective duty finer than the quantization step Δd, the implementation applies *accumulator-based dithering*. This technique is a well-established resolution enhancement method in digitally controlled converters [47]. In this approach, the controller alternates between two adjacent quantized duty levels over successive PWM cycles so that the time-averaged output equals the desired target value. This time-averaging refines the *effective* duty resolution beyond the raw step size Δd while maintaining compliance with the device's minimum safe pulse width. Because the motor's electrical and mechanical dynamics are much slower than the PWM carrier, the dithering sequence is naturally filtered by the plant, producing an accurate average voltage without introducing measurable ripple.

For example, if $\Delta d = 0.05$ but the target is $d_{\text{tgt}} = 0.225$, the two available levels are

$$d_{\text{low}} = 0.20, \qquad d_{\text{high}} = 0.25.$$

Let n_ℓ and n_h be the number of cycles at d_{low} and d_{high}, respectively. The average duty is

$$\bar{d} = \frac{n_\ell d_{\text{low}} + n_h d_{\text{high}}}{n_\ell + n_h}.$$

To match d_{tgt},

$$\frac{0.20\,n_\ell + 0.25\,n_h}{n_\ell + n_h} = 0.225 \iff n_h = n_\ell.$$

Thus any equal number of low and high cycles satisfies the condition; the minimal repeating pattern is one cycle each

$$[20\%,\ 25\%] \ \Rightarrow\ \frac{0.20 + 0.25}{2} = 0.225.$$

By alternating between these quantized levels, the controller produces an effective duty of 0.225 even though the hardware resolution is limited to 0.05.

2.5.2 Bias Removal and Adaptive Ripple Compensation

The rectified AC bus exhibits a steep rise from valley to crest each half cycle as the mains crosses zero and the bulk capacitor begins to recharge. This produces a brief inrush tendency at the motor terminals. A hardware choke can attenuate this behavior, but it adds size, cost, and loss. Instead, the controller exploits the instrumented *prestage* bus measurement $V_{\text{bus}}(t)$ and a ripple-synchronous PLL phase $\phi(t)$ to align compensation with the instantaneous bus condition. Using a phase-registered bus model $\hat{V}_{\text{bus}}(\cdot)$ and a small adaptive phase offset $\phi_{\text{raw}}(t)$, the compensator pre-shapes duty around the recharge interval, suppressing peaks without a series choke.

Variable definitions. The compensator works with a small set of signals that are shared with the PLL and the MPC. Table 2.4 lists the principal symbols and implementation variables. In equations, ϕ and ϕ_{raw} are expressed in radians unless stated otherwise; the implementation updates the phase *rate* in degrees per second for convenient gain tuning.

Table 2.4: *Compensator variables, units, and roles.*

Symbol / Variable	**Units**	**Meaning**
$\phi(t)$	rad	Ripple phase from the PLL, wrapped to $[0, 2\pi)$.
$\phi_{\text{raw}}(t)$	rad	Adaptive *phase offset* applied to the bus model; evolves only by integrating the commanded phase rate.
$\dot{\phi}_{\text{cmd}}$	deg/s	Commanded phase *rate* adjusted at sub-phase ticks; internal implementation quantity used to update ϕ_{raw}.
T_N	s	Sub-phase tick spacing within each 120 Hz cycle ("N-tick" period).
$\hat{V}_{\text{bus}}(\cdot)$	V	Fourier bus model evaluated at $\phi + \phi_{\text{raw}}$.
$V_{\text{bus,used}}(t)$	V	Bus estimate used for duty ceiling: $V_{\text{bus,used}} = \hat{V}_{\text{bus}}(\phi + \phi_{\text{raw}})$.
$V_{\text{cmd}}(t)$	V	Requested load voltage from the outer regulation objective.
$V_{\text{load}}(t)$	V	Estimated or measured motor-terminal voltage.
$e_{\text{raw}}(t)$	V	Raw error: $V_{\text{load}} - V_{\text{cmd}}$.
$\overline{e}$	V	Cycle mean of e_{raw} over one ripple rotation (used for bias removal).
$e_{\text{dev}}(t)$	V	Bias-removed error: $e_{\text{raw}} - \overline{e}$.
$e_{\text{use}}(t)$	V	Gated error used for updates after dead-zone removal.
V_{engage}	V	Engage threshold: compensation is active only when $V_{\text{load}} \geq V_{\text{engage}}$.
e_{dz}	V	Dead-zone half-width for ripple error.
K_P, K_D	deg/(s · V), deg/V	Proportional and derivative gains in the phase-rate update law.

Load-voltage estimate and why a bias appears. With a bus model $\hat{V}_{\text{bus}}(\cdot)$ and duty $d(t)$, the terminal voltage is estimated as

$$V_{\text{load}}(t) \;=\; d(t)\, V_{\text{bus,used}}(t), \qquad V_{\text{bus,used}}(t) \;=\; \hat{V}_{\text{bus}}\Big(\phi(t) + \phi_{\text{raw}}(t)\Big).$$

If duty does not anticipate the upcoming recharge, $V_{\text{load}}(t)$ shows a brief overshoot near the crest and sits slightly low for the rest of the cycle. The cycle average of $e_{\text{raw}} = V_{\text{load}} - V_{\text{cmd}}$ therefore drifts negative even when the flat portion closely tracks the command. Removing this mean bias reveals the true ripple excursion that the compensator should target.

Bias removal and gating. Define the bias-removed deviation and the gated error:

$$\begin{aligned} e_{\text{dev}}(t) &= e_{\text{raw}}(t) - \overline{e}, \\ e_{\text{use}}(t) &= \Big(|e_{\text{dev}}(t)| - e_{\text{dz}}\Big)_+ \, \text{sgn}\Big(e_{\text{dev}}(t)\Big) \, \mathbf{1}\{\, V_{\text{load}}(t) \geq V_{\text{engage}} \,\}\,, \end{aligned} \tag{2.21}$$

where $(x)_+ = \max(x, 0)$ and $\mathbf{1}\{\cdot\}$ is the indicator function. The engage threshold restricts action to the high-voltage segment where the recharge peak occurs, and the dead zone prevents small, benign deviations from triggering corrections.

Core compensator rule: phase-rate nudging. The compensator adjusts the *rate* of the model phase, not the phase itself. Between N-ticks, ϕ_{raw} evolves by integration:

$$\phi_{\text{raw}}(t{+}dt) \;=\; \text{wrap}_{[-\pi,\pi)}\Big[\phi_{\text{raw}}(t) + \dot{\phi}_{\text{cmd}} \frac{\pi}{180}\, dt\Big]\,. \tag{2.22}$$

At each sub-phase tick t_k with spacing T_N, the commanded rate is updated by a gated PD law:

$$\dot{e}(t_k) \approx \frac{e_{\text{use}}(t_k) - e_{\text{use}}(t_{k-1})}{T_N}, \tag{2.23}$$

$$\dot{\phi}_{\text{cmd}} \leftarrow \dot{\phi}_{\text{cmd}} \;+\; \underbrace{K_P\, e_{\text{use}}(t_k)\, \mathbf{1}_{\{\dot{e}(t_k)\geq 0\}}}_{\text{P acts only when error is not improving}} \;+\; \underbrace{K_D\, \dot{e}(t_k)}_{\text{D provides signed damping}}\,. \tag{2.24}$$

The proportional term is disabled when $\dot{e}(t_k) < 0$ since the deviation is already decaying. The derivative term remains continuous and signed, providing damping across zero-crossings of the deviation. Only the rate $\dot{\phi}_{\text{cmd}}$ is nudged; ϕ_{raw} remains

smooth because it is obtained solely by integrating (2.22).

Per-cycle sequence. Each ripple cycle proceeds as follows:

1. **Bus prediction:** evaluate $V_{\text{bus,used}} = \hat{V}_{\text{bus}}(\phi + \phi_{\text{raw}})$ and compute a duty target with headroom $d_{\text{tgt}} = V_{\text{cmd}}/V_{\text{bus,used}}$.

2. **Quantization:** apply PWM quantization and optional dithering to d_{tgt}.

3. **Error accumulation:** update e_{raw}, refresh $\bar{e}$ once per 2π of ϕ, and form e_{use} with the engage and dead-zone gates.

4. **Phase-rate update:** at each N-tick, compute $\dot{e}$ and update $\dot{\phi}_{\text{cmd}}$ by (2.24); between ticks, integrate ϕ_{raw} using (2.22).

Why it works. The bus model defines a predictive ceiling for the duty ratio near the recharge interval, while the adaptive offset ϕ_{raw} maintains alignment between the model and the actual ripple observed in the plant. By updating the *rate* rather than the instantaneous phase, the controller avoids abrupt jumps and prevents the re-imprinting of ripple onto the PWM stream. Through bias removal and phase-gated action, the compensator selectively targets the relevant peak region, rejecting the 120 Hz sag while allowing the outer controller to maintain its commanded voltage.

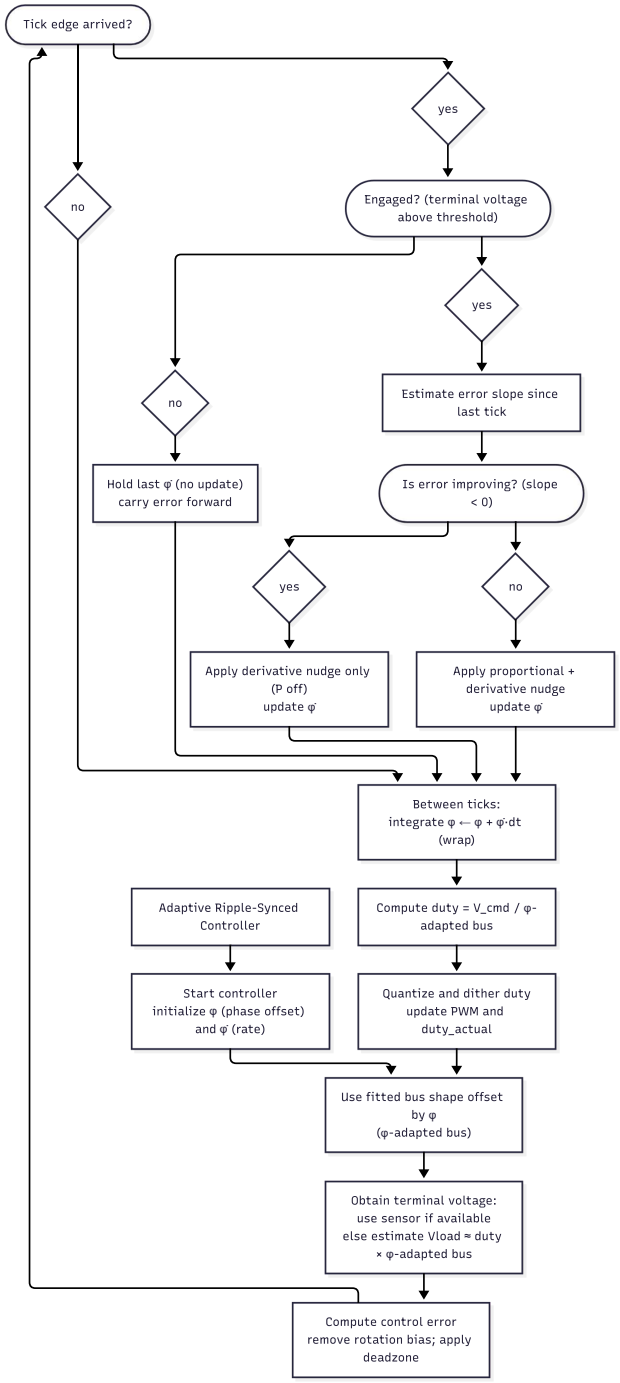


Figure 2.6: *Compensator control flow. The phase ϕ comes from the PLL; the adaptive offset ϕ_{raw} is advanced by a commanded rate that is nudged at sub-phase ticks using bias-removed, ripple-synchronous error.*

Chapter 3

Adaptive Control and Online Model Identification

3.1 Overview

The term *adaptive*, in the context of MPC, refers to the controller's ability to modify its internal model or control matrix to account for variations in plant dynamics or load conditions. In the present work, the focus is on accurately identifying the key plant dynamics; therefore, the adaptive framework is designed to periodically recalibrate the system and re-estimate the plant parameters. We assume that these recalibration events occur at defined intervals, such as during startup, cooldown, or under known load conditions, when the operating state is sufficiently stable for reliable identification. During each calibration window, the motor is driven under controlled or unloaded conditions to ensure that the updated model accurately reflects the system's intrinsic behavior.

Unlike purely offline identification, adaptive control requires that model fitting be performed *online* using data captured during normal operation. Among the electrical and mechanical parameters, the armature resistance R changes the most

over the motor's lifetime due to commutator wear and thermal aging [6]. Other variables generally remain steady except under extreme operating conditions, where high temperature or heavy loading can cause measurable variation in the back-EMF and torque constants (k_v and k_t), primarily through magnetic flux weakening. Accordingly, R is treated as a time-varying parameter that can be adapted continuously, whereas full recalibration cycles primarily serve to refine the overall fit and ensure long-term consistency.

To identify the full coupled plant, samples of multiple quantities must be gathered. For the plant dynamics we log

$$\{\, I,\, V_{\text{load}},\, \omega,\, \theta \,\}$$

(four channels: current, terminal voltage, speed, and rotor angle), and for the bus reference we optionally collect

$$\{\, V_{bus},\, \phi \,\}$$

(two additional channels: rectified bus voltage and mains phase).

Multiple acquisition rates are advantageous in hardware because the electrical subsystem responds at higher bandwidths, whereas the mechanical dynamics evolve much more slowly. In our simulation, however, a single, properly filtered sampling stream proved sufficient and simplified synchronization. An interesting side effect of operating on a rectified bus is that the 120 Hz ripple introduces a small, predictable sinusoid into the drive, which acts as built-in *persistent excitation*. This natural ripple improves identifiability by exciting multiple modes of the coupled model, particularly aiding estimation of L and k_v. As emphasized in [48], mechanical terms such as the viscous damping coefficient λ are typically less observable in short electrical transients. Our results confirm this limitation: although the ripple improves electrical parameter fitting, the periodic AC rectification also introduces additional complex-

ity that degrades λ estimation. This discrepancy is examined further in the Results section.

Why a Model-Based Approach Is Required

In principle, a controller could adaptively tune its gains using only the applied voltage u_k and measured speed $y_k = \omega_k$. However, with an unknown or time-varying load torque T_L, such a minimalist approach cannot separate electrical and mechanical effects, particularly the resistance R, which becomes indistinguishable from unmeasured load disturbances. Accurate estimation of R and related parameters therefore requires access to the internal electrical states. By using a model-based identification scheme that captures all key channels simultaneously (I, V, ω, θ), the controller preserves observability of both the electrical and mechanical subsystems and can update R online while still regulating speed even when T_L is unknown.

3.2 Memory and Buffering Strategy

Up to 1 MB of RAM is assumed for high-resolution data capture. A full electrical rotation at fine angular resolution is required to reconstruct $R(\theta)$ accurately; while segmented runs could reduce memory use, a single full-rotation capture avoids discontinuities and simplifies fitting.

It should be noted that a sensorless implementation would not eliminate the need for ω and θ themselves, as these quantities remain essential for the internal observer and estimator dynamics. Rather, the mechanical states would be reconstructed from electrical feedback, allowing physical sensors to be omitted while retaining the same state dimensionality. In this case, the buffering and computational load remain largely unchanged, since the internal model must still track the same variables with equivalent temporal resolution.

Buffer operation (edge-symmetric capture). Data collection is edge-triggered to ensure symmetry about the electrical waveform. Sampling begins on a phase reference and terminates only when the same phase offset reappears, guaranteeing an integer number of ripple or commutation cycles. This symmetry ensures phase-aligned residuals and well-conditioned regression matrices.

Sampling footprint. Front-end sampling runs at $f_s = 200$ kHz ($\Delta t = 5$ μs). For the identification buffer, one sample is logged every 0.75×10^{-5} s (7.5 μs, effective ~ 133 kHz) for $N = 20{,}000$ samples, corresponding to approximately 0.15–0.20 s of data per calibration ramp. With four plant channels, the required memory is

$$M_{\text{plant}} = N \times 4 \times 4 = 320{,}000 \text{ bytes} \approx 313 \text{ kB},$$

and including the two optional bus channels adds roughly 160 kB more, remaining well within a nominal 1 MB budget.

This total exceeds the on-chip RAM capacity of the TI C2000 family devices used in the current prototype. Accordingly, external memory would be required for full-resolution buffering. While the C2000 can manage data acquisition and filtering in real time, an external SRAM or SDRAM module is necessary to support complete high-resolution captures for spectrum fitting and model identification.

3.3 Signal Preprocessing and Filtering Pipeline

All measured signals were preprocessed prior to model fitting to suppress switching artifacts and produce clean derivatives suitable for parameter extraction. The processing chain consisted of three main stages: (i) Butterworth IIR front-end filtering, (ii) Savitzky–Golay local smoothing, and (iii) derivative estimation using a fourth-order finite-difference kernel.

Each of these stages contributes differently to the conditioning of the signal. The Butterworth filter provides smooth attenuation of PWM carrier harmonics without distorting the 120 Hz bus ripple that carries diagnostic information. The Savitzky–Golay filter restores local curvature lost during low-pass filtering and preserves the true amplitude and phase of slow transients, a property essential for reliable derivative estimation. Finally, the finite-difference derivative stage benefits directly from this preprocessing, producing stable, phase-aligned estimates of $\dot{I}$ and $\dot{\omega}$ that are consistent with the physical model.

In simulation, these filters are not dramatically impactful because the signals are largely noise-free and free from sampling jitter. However, their inclusion establishes a realistic baseline for later hardware validation, where sensor noise, quantization, and switching artifacts will be significant. In practice, additional filtering or adaptive schemes may be required to handle such effects, but the present configuration provides a representative and computationally efficient foundation for control and diagnostic testing.

Butterworth IIR filtering

Each electrical and mechanical signal was first passed through a digital Butterworth low-pass filter designed for a $\sim$ 850 Hz cutoff at the 200 kHz sampling rate. Butterworth filters provide maximally flat magnitude response in the passband, making them a standard choice for general-purpose noise suppression in control and signal-conditioning applications [49]. A fourth-order realization was applied to voltage and current channels, and a second-order realization to speed and angle channels. The continuous-time prototype

$$H(s) = \frac{1}{\sqrt{1 + (s/\omega_c)^{2N}}},$$

was discretized via the bilinear transform $s = \frac{2}{T}\frac{1-z^{-1}}{1+z^{-1}}$, producing the recursive form

$$y[n] = \sum_{i=0}^{M} b_i\, x[n-i] - \sum_{j=1}^{N} a_j\, y[n-j].$$

Coefficients (a_j, b_i) were precomputed in MATLAB to ensure numerical stability and unity DC gain at the 200 kHz front-end rate. This retains all low-frequency content (including 120 Hz ripple) while attenuating PWM harmonics well beyond the control bandwidth.

Savitzky–Golay Smoothing

After Butterworth filtering, residual quantization noise was locally smoothed using a five-point quadratic Savitzky–Golay (SG) filter. This method fits a polynomial of degree two to each local window via least squares, preserving local curvature and phase alignment. The specific convolution coefficients used were

$$y_k = \frac{-3x_{k-2} + 12x_{k-1} + 17x_k + 12x_{k+1} - 3x_{k+2}}{35},$$

which corresponds to the "Quadratic A_{20}, 5-point" filter tabulated in Appendix II, Table I of Savitzky and Golay's original paper [50]. This configuration provides modest high-frequency attenuation (about 15–20 dB at the Nyquist limit of the downsampled stream) while maintaining zero phase distortion and preserving the waveform's curvature.

Derivative estimation

Derivatives of current and speed were computed using the five-point, fourth-order central-difference method [51], which is exact for cubic polynomials and exhibits trun-

cation error $\mathcal{O}(\Delta t^4)$:

$$\left.\frac{dx}{dt}\right|_k \approx \frac{x_{k-2} - 8x_{k-1} + 8x_{k+1} - x_{k+2}}{12\,\Delta t}.$$

For short buffers (e.g., near data boundaries), a three-point fallback $\Big[-1, 0, 1\Big]/(2\Delta t)$ was applied. This approach produces high-fidelity, low-noise derivative estimates that are both phase-correct and compatible with the SG-preprocessed data, providing stable dynamic terms for the predictive controller.

3.4 Model Fitting Procedure

Relation to the predictive controller. The parameter fits obtained from the filtered signals supply the coefficients used in the MPC forward model, which computes future current and speed trajectories for voltage scheduling and phase compensation. Accurate identification of $(L, R, k_v, k_t/J, \lambda/J)$ ensures the controller remains synchronized with the true drive dynamics as the motor ages or as conditions vary.

The coupled discrete model used within the MPC predictor is

$$\begin{aligned}
\Delta I_k &= \frac{1}{L}\Big(V_k - R\,I_k - k_v\,\omega_k\Big),\\
I_{k+1} &= I_k + \Delta t\,\Delta I_k,\\
\omega_{k+1} &= \omega_k + \Delta t\left(\frac{k_t}{J}\,I_k - \frac{\lambda}{J}\,\omega_k + d\right),
\end{aligned}$$

where d represents the disturbance torque estimated by the observer.

3.4.1 Acquisition protocol: acceleration/launch-cycle fitting

In the reference study [48], parameter extraction was performed in the *steady-state region*, where the current, voltage, and speed remain approximately constant. Such a

regime is mathematically advantageous because derivative terms vanish, simplifying the regression and improving numerical conditioning when the excitation waveform is DC-fed and stable. However, in our rectified AC-fed configuration, the DC bus itself contains residual ripple and harmonic content, preventing a truly steady operating condition. Consequently, we observed experimentally that steady-state fitting offers no measurable advantage, and its reduced excitation spectrum can in fact degrade conditioning, whereas acceleration-based datasets provide a richer and more informative signal basis.

Extensive testing therefore showed that parameter fitting is most reliable when performed on the *acceleration cycle*. Even the mechanical coefficients benefit from this regime because the simultaneous variation in I, $\dot{I}$, and ω increases excitation diversity, although mechanical parameters still require a longer collection interval for tight confidence bounds.

To ensure a known, repeatable operating condition available during normal use, we standardize on the *launch cycle* for all datasets. Operationally, this means the motor spends much longer accelerating to speed than the per-sample interval, allowing a large number of samples to be collected during launch. This approach offers two advantages: (i) faster tuning and (ii) a quick pre-use calibration whenever the motor can be left unloaded for less than one second. As noted in *Parameters Identification of a Permanent Magnet DC Motor: A Review* [48], mechanical parameters are typically less observable in short electrical transients. The extended launch window used here addresses this limitation by providing the longer excitation horizon needed to identify k_t/J and λ/J reliably.

Single-pass identification and buffer sizing. In our implementation we deliberately perform a *single-pass, all-at-once* identification (electrical and mechanical together) to avoid two separate tuning steps. Practically, this implies provisioning a

larger launch buffer so that the mechanical timing requirements are met while simultaneously oversampling the electrical signals; i.e., we collect *more electrical-buffer samples* than strictly necessary so the unified window also suffices for the slower mechanical fit. This assumes additional memory is available, which may not always be the case on minimal microcontrollers; however, given the operational advantages of one-shot tuning, we treat external RAM as a reasonable system assumption for this calibration mode. When memory is constrained, the same protocol can be executed with decimation or staggered multi-rate capture, but the single-pass buffer with external RAM remains the preferred configuration for repeatable, operator-friendly calibration.

3.4.2 Electrical Parameter Estimation

The electrical subsystem obeys

$$V = L\,\dot{I} + R\,I + k_v\,\omega.$$

Let there be M samples indexed by $i = 1, \ldots, M$, with filtered sequences $\{V_i, I_i, \dot{I}_i, \omega_i\}$. The parameters (L, R, k_v) are obtained by minimizing the mean-squared residual between the measured and modeled voltages,

$$\min_{L,R,k_v} \; \frac{1}{M} \sum_{i=1}^{M} \Big(V_i - L\,\dot{I}_i - R\,I_i - k_v\,\omega_i \Big)^2, \tag{3.1}$$

consistent with the standard least-squares framework where each observation satisfies $V_i = \varphi_i^\top \theta + e_i$ and the goal is to minimize $\sum e_i^2$ as described by Ljung [52].

Matrix form. Define the regressor matrix $X \in R^{M\times 3}$ and parameter vector $\theta \in R^3$:

$$X = \begin{bmatrix} \dot{I}_1 & I_1 & \omega_1 \\ \vdots & \vdots & \vdots \\ \dot{I}_M & I_M & \omega_M \end{bmatrix}, \qquad \theta = \begin{bmatrix} L \\ R \\ k_v \end{bmatrix}, \qquad \mathbf{v} = \begin{bmatrix} V_1 \\ \vdots \\ V_M \end{bmatrix}.$$

The least-squares estimate is

$$\hat{\theta} = (X^\top X)^{-1} X^\top \mathbf{v},$$

following the canonical regression relation $\mathbf{v} = X\theta + \mathbf{e}$, equivalent to Ljung's formulation $y(t) = \varphi^\top(t)\theta + e(t)$ in system identification theory [52]. Robust trimming removes outliers from commutation spikes and edge transients, yielding refined estimates of L, R, and k_v.

3.4.3 Mechanical parameter estimation

Using the same synchronized dataset, the mechanical subsystem

$$\dot{\omega} = \frac{k_t}{J}\, I - \frac{\lambda}{J}\, \omega + d$$

is regressed for coefficients $\frac{k_t}{J}$ and $\frac{\lambda}{J}$. The least-squares form is

$$\min_{a,b} \sum_{n=1}^{N} \Big[\dot{\omega}_n - (a\, I_n + b\, \omega_n)\Big]^2,$$

where $a = \frac{k_t}{J}$ and $b = -\frac{\lambda}{J}$. Because the data come from a single synchronized buffer captured during the launch cycle, the electrical–mechanical coupling is preserved and the regressions share identical timestamps. Empirically, launch-cycle datasets improve observability of a; b benefits from a longer window due to its low-frequency dominance.

3.5 Resistance Spectrum Reconstruction

Once the electrical parameters (L, R, k_v) are identified, the rotation-dependent internal resistance can be reconstructed directly from the measured signals. Starting from the instantaneous electrical balance,

$$V = L\,\dot{I} + R\,I + k_v\,\omega,$$

the effective resistance at each angular sample is computed by isolating R:

$$R(\theta) = \frac{V - (L\,\dot{I} + k_v\,\omega)}{\max(|I|,\, I_{\min})\ \mathrm{sgn}(I)}.$$

The denominator incorporates two safeguards for numerical stability. First, $\mathrm{sgn}(I)$ preserves the correct conduction polarity, preventing $R(\theta)$ from flipping sign when current reverses between commutator segments. Second, the term $\max(|I|,\, I_{\min})$ applies a lower bound on the divisor to avoid exaggerated values near current zero-crossings. Here $I_{\min}$ is defined adaptively as the 5th percentile of the absolute current magnitude,

$$I_{\min} = P_5\Big(|I|\Big),$$

representing the level below which only 5% of all $|I|$ samples occur. This percentile-based floor suppresses noise-dominated samples while preserving the dominant portion of the waveform where current remains well above sensor noise, ensuring a stable and physically meaningful reconstruction of $R(\theta)$.

Interpretation and significance. The resulting $R(\theta)$ profile represents the *instantaneous effective contact resistance* of the commutator–brush interface as a function of electrical angle. Unlike the bulk winding resistance R, which is relatively constant, this spectral resistance captures the small, periodic perturbations that occur

as successive commutator segments pass under the brushes. These variations reflect the combined effects of segment geometry, brush seating, contact film condition, and localized wear. As the motor ages, these modulations typically increase in amplitude and become asymmetric, providing a quantitative indicator of wear and eccentricity.

Filtering and angular registration. Because $R(\theta)$ is derived from time-domain signals that have already been low-pass filtered, it inherits their smoothness while preserving the essential commutation harmonics. Angular registration between $R(\theta)$ and the physical commutator is maintained through the θ channel, allowing successive calibrations to be phase-aligned and compared directly. In post-processing, $R(\theta)$ can be analyzed using Fourier or harmonic decomposition to extract diagnostic metrics such as:

- Mean resistance $\bar{R}$ — representing the effective armature heating load.
- Peak-to-peak variation $\Delta R(\theta)$ — a proxy for brush-contact nonuniformity.
- Harmonic distortion content — useful for identifying repeating wear patterns.

Practical implications. The $R(\theta)$ reconstruction forms the *core diagnostic signal* of the entire framework. It enables condition monitoring without additional sensors and can be computed in real time from a single buffered rotation. Because it relies only on measurable electrical quantities, it integrates naturally with the predictive control layer; the same data used for control tuning also provide continuous health feedback. In this sense, $R(\theta)$ bridges the boundary between adaptive control and predictive maintenance by converting electrical observables into a direct, physically interpretable wear signature.

3.5.1 Bus-voltage Reference (optional online fit)

The rectified bus voltage is optionally modeled as a truncated Fourier series:

$$V_{bus}(\phi) \approx a_0 + \sum_{k=1}^{K} \Big[a_k \cos(k\phi) + b_k \sin(k\phi) \Big].$$

Coefficients $\{a_k, b_k\}$ are determined by least squares with median-absolute-deviation outlier rejection to suppress transient spikes. Although the bus profile can be precharacterized, the solver may also refit it online during calibration to verify the ripple phase and amplitude.

A similar harmonic representation has been used for ripple analysis in power-conversion research. For example, Huang *et al.* [53] express the input current of a feedforward voltage–controlled AC–DC converter as a truncated Fourier series to analyze and suppress the double–line–frequency ripple component. Their method preshapes the input current by modulating the duty cycle inversely with the rectified line voltage, thereby maintaining near-constant instantaneous power and reducing bus-voltage ripple without a large electrolytic capacitor.

Although their approach is more feedforward-oriented, the underlying principle of ripple-synchronous modeling is closely related to the present work. A similar strategy may be worth exploring for high-power brushed PWM drives, where the large bulk-capacitor bank represents a significant portion of both cost and mass. In such systems, a phase-adaptive bus model could potentially be coupled with predictive or feedforward compensation to reduce the required capacitance while preserving voltage stability.

In practice, the fitting routine collects a minimum of

$$N_{\mathrm{VB,fit}} = 10{,}000$$

samples at a spacing of

$$\Delta t_{\mathrm{VB,fit}} = 5 \times 10^{-5}\,\mathrm{s},$$

resulting in an effective capture window of about

$$T_{\mathrm{fit}} = N_{\mathrm{VB,fit}} \times \Delta t_{\mathrm{VB,fit}} \approx 0.5\ \mathrm{s}.$$

Thus, the bus model fit typically requires about half a second, longer than the main plant identification ramp but still short enough for inline calibration or startup routines. This buffer configuration was not tested as extensively as the main identification buffer, yet it is likely that a smaller number of samples would still produce a stable and accurate fit in practice.

Chapter 4

Results

The MATLAB/Simulink simulation models a brushed permanent-magnet DC motor comparable to the Motenergy ME1003, rated at approximately 11 kW continuous and 20 kW peak output. The nominal parameters used are

$$R = 20\ \mathrm{m\Omega}, \quad L = 100\ \mu\mathrm{H}, \quad k_v = 0.186\ \mathrm{V/(rad/s)}, \quad J = 0.05\ \mathrm{kg \cdot m^2}, \quad \lambda = 0.005.$$

These values correspond to **Motor 1** in Table 4.1 and are representative of a large, high-torque traction-class motor operating near the upper limit typically driven by PWM converters. The larger motor size provides a favorable signal-to-noise ratio, allowing ripple behavior and compensator performance to be examined without excessive quantization or numerical artifacts. In practice, the same control and identification framework applies to smaller drives provided adequate filtering, sampling synchronization, and quantization control are maintained.

During startup, the simulated current exhibits an initial inrush of roughly 300 A as the armature inductance charges and the back-EMF ramps. This magnitude, approximately 2.3× the nominal operating current, is consistent with observed behavior in ME1003-class machines driven from low-impedance sources. Such inrush underscores the need for robust voltage tracking and ripple management, since even

brief 120 Hz bus sags can couple strongly into the torque channel when the current slope $\dot{I}$ is large.

Bus capacitance and expected ripple. The DC-bus capacitance strongly influences ripple magnitude. For this study, the bus capacitor was set to $C_{\text{bus}} = 0.068$ F, typical of commercially available electrolytic or hybrid film capacitors rated around 80–100 V. For an 11 kW drive at a nominal bus of 83 V, the average steady-state current is approximately 130 A. Over one half-cycle of the rectified 120 Hz bus ($\Delta t \approx 8.33$ ms), a purely capacitive discharge yields the approximate voltage sag

$$\Delta V \approx \frac{I\,\Delta t}{C} \approx \frac{130 \times 8.33 \times 10^{-3}}{0.068} \approx 16 \text{ V},$$

or roughly 19% of nominal. While this capacitance lies toward the low end for passive smoothing at this power level, it was intentionally selected to evaluate achievable ripple reduction with the proposed compensator. A slightly oversized bulk value stabilizes the electrical subsystem and isolates the compensator's influence, ensuring that observed ripple behavior arises primarily from control action rather than exaggerated supply droop.

In hardware, a 0.068 F bulk capacitor is representative of the type used in this study and is sufficient for the intended experimental evaluation. If a precise sizing were required for a non-compensated system, it would follow the same principle of determining the allowable speed-regulation error band from a known reference specification.

To size the capacitor bank against a speed-regulation target, we derive an allowable ω error band from such a reference. As a concrete benchmark, the IronHorse GSD8 series (10 A model) specifies approximately $\pm 0.1\%$ steady-state speed accuracy when encoder feedback is employed. Although high-current traction drives will not typically achieve the same absolute precision as low-current units, our compensator re-

sults indicate that comparable or better percentage regulation is attainable for larger motors. For lower-power applications near the 10 A class, the simulations further suggest that acceptable regulation may be obtained even without a compensator.

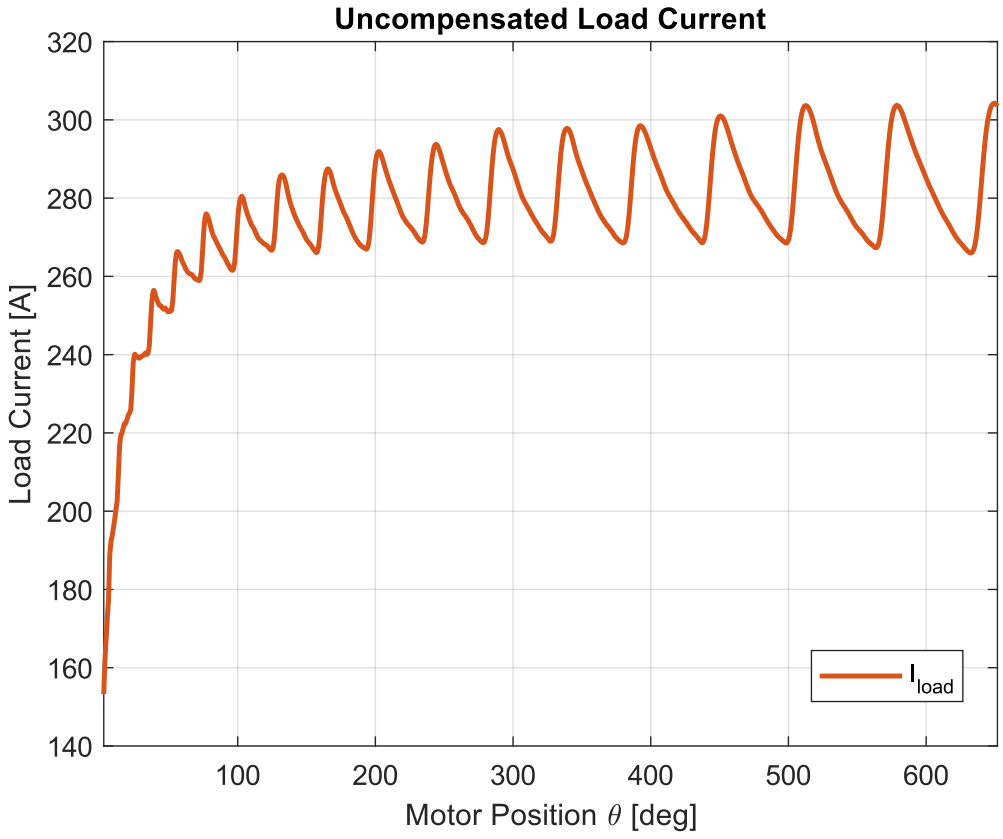


Figure 4.1: *Uncompensated armature current showing pronounced* 120 Hz *modulation synchronized with bus recharge.*

Without the proposed compensator, the raw plant response exhibits strong bus-coupled oscillations. The load-voltage ripple follows the 120 Hz recharge pattern, producing a distinct modulation in mechanical speed ω. If the instantaneous bus voltage is not measured and compensated in real time, this disturbance propagates directly into the torque loop, causing visible periodic speed wobble.

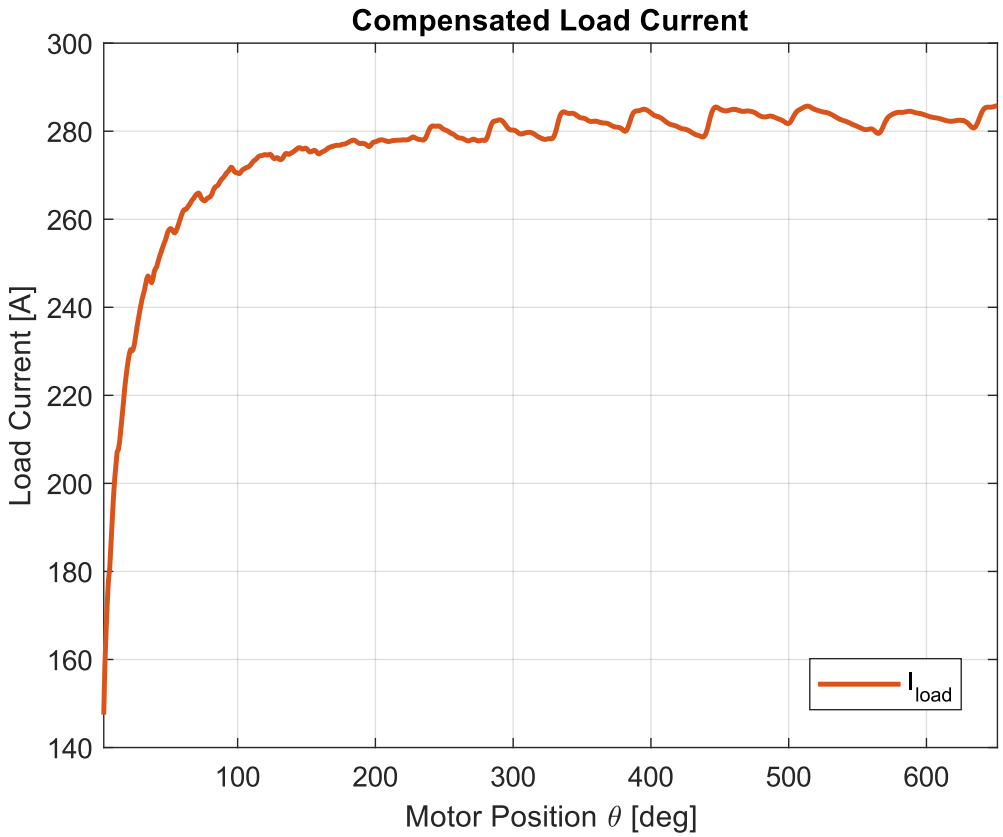


Figure 4.2: *Armature current with ripple compensation applied. The* 120 Hz *component is strongly attenuated.*

With the ripple-compensated controller enabled, the armature current becomes substantially smoother. The compensator tracks the ripple phase and adjusts duty to maintain a nearly constant effective load voltage. In steady state, the current ripple amplitude is reduced by more than an order of magnitude relative to the uncompensated case.

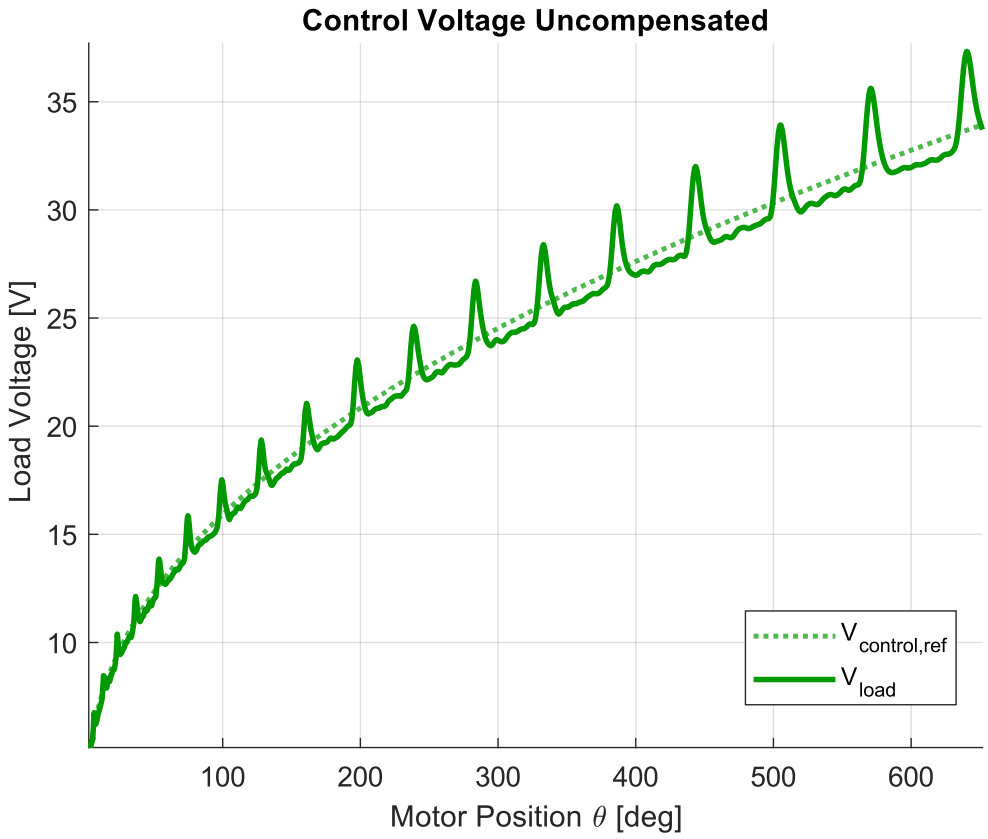


Figure 4.3: *Uncompensated load voltage showing strong recharge-edge peaking caused by bulk-capacitor charging.*

In the uncompensated case, the terminal voltage sharply overshoots the reference at each recharge edge and dips slightly below it afterward, forming a pronounced oscillatory envelope. This distortion introduces a steady bias in the average applied voltage, which becomes more severe as the commanded load voltage increases. Such bias prevents consistent ripple control and motivates the use of phase-synchronized compensation to maintain a uniform effective voltage across each bus cycle.

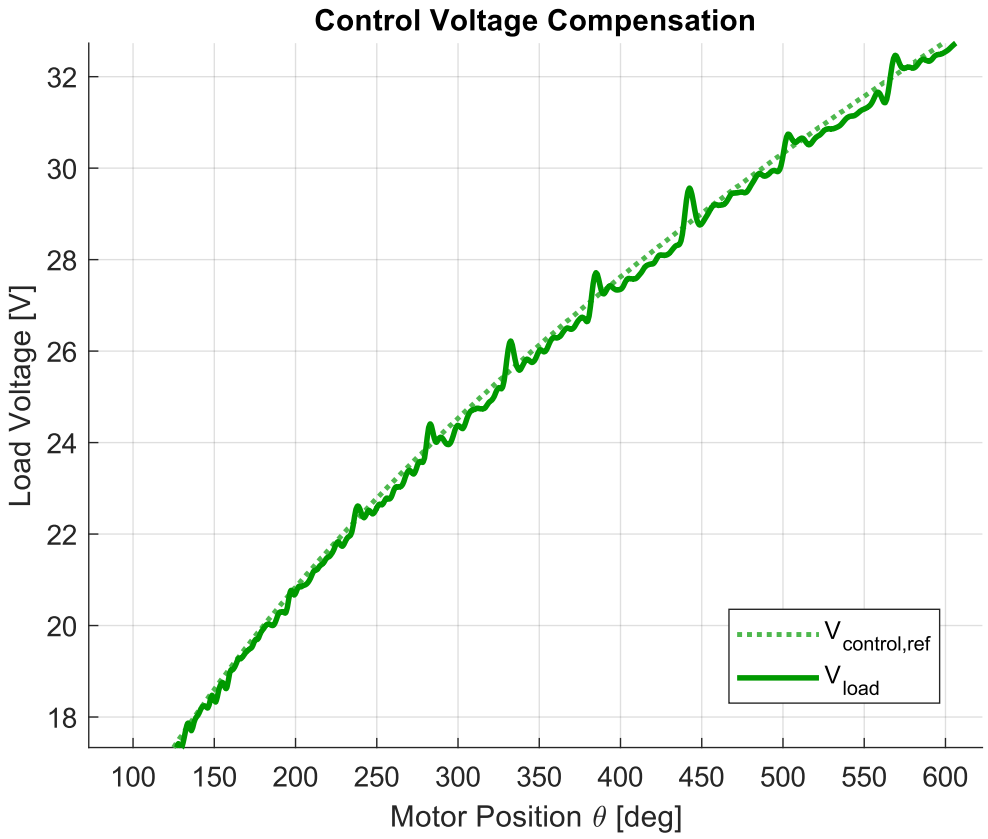


Figure 4.4: *Load voltage under compensation, showing suppression of recharge-edge distortion and alignment with the reference waveform.*

With compensation enabled, the recharge-edge overshoot and subsequent undershoot are strongly suppressed, producing a smooth voltage waveform that closely follows the reference profile. The effective bias observed in the uncompensated case is eliminated, and the terminal voltage remains consistent throughout each bus cycle. During the initial settling phase, the controller requires a short period of a few tenths of a second to refine the phase correction ϕ_{raw}, after which the alignment remains stable during extended operation.

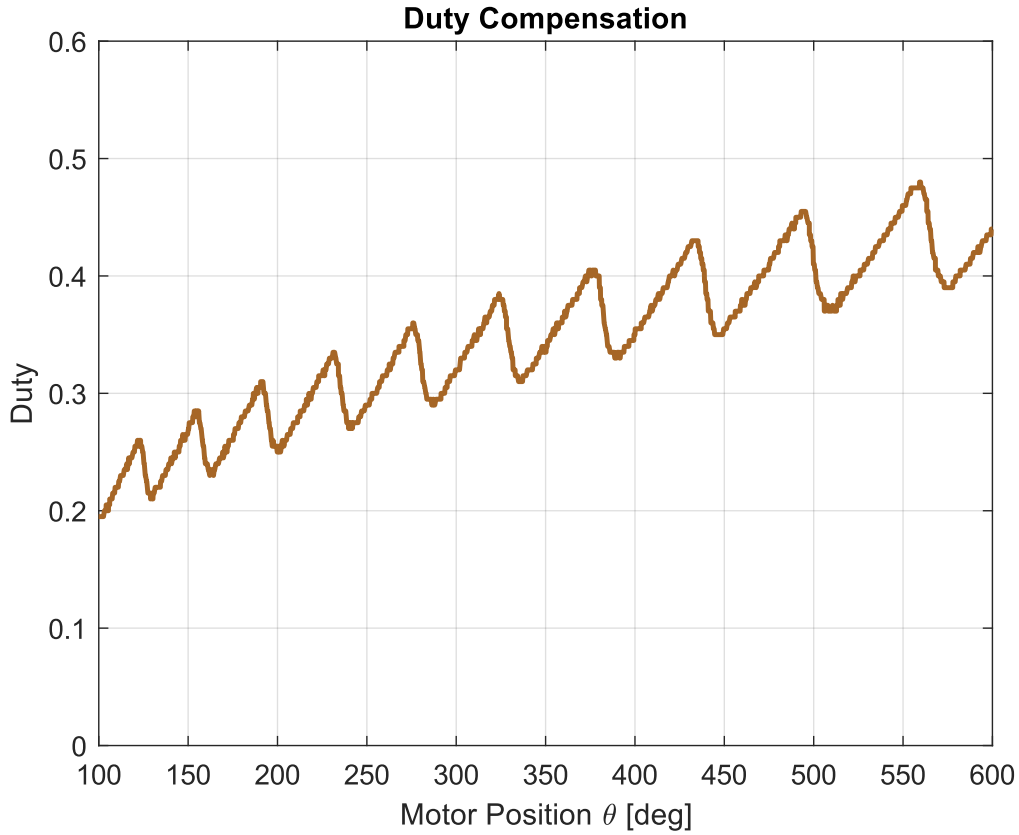


Figure 4.5: *Duty-cycle trajectory showing phase-synchronized modulation during bus ripple cycles.*

The duty-cycle trajectory illustrates the control effort. During each bus recharge, the controller momentarily reduces duty to counteract the rising bus voltage, maintaining a stable effective terminal voltage. This modulation remains phase-locked to the detected ripple.

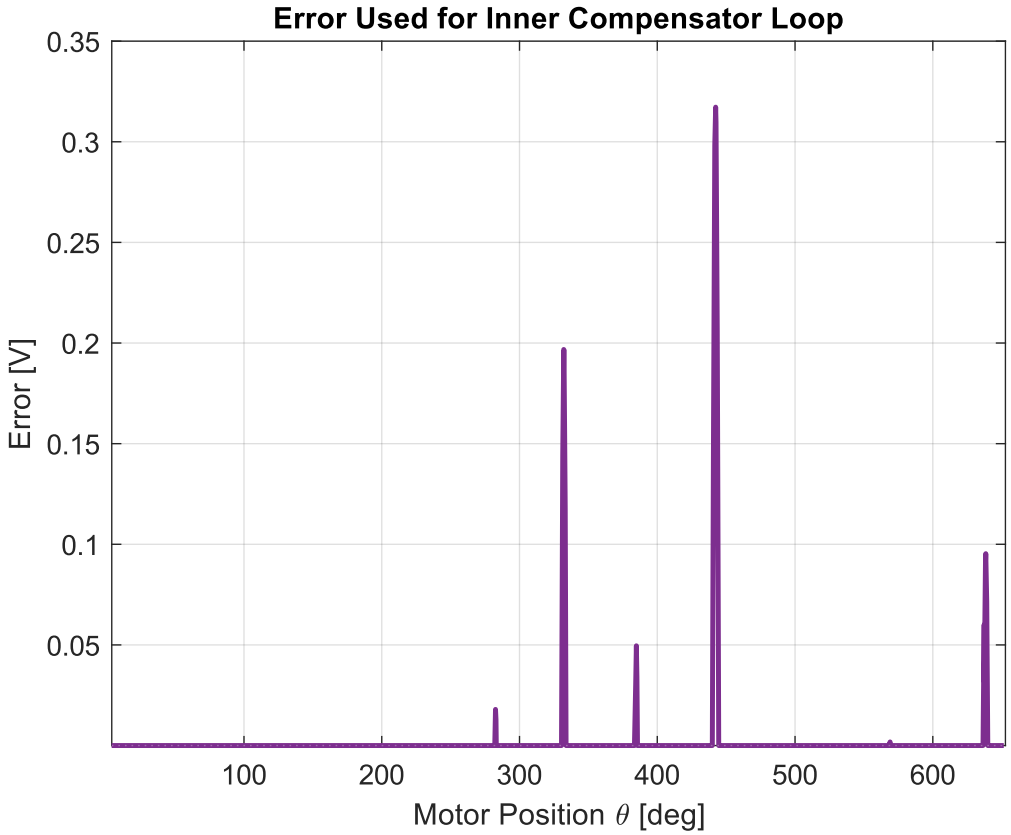


Figure 4.6: *Tracking-error signal used for adaptive phase correction. The mean error collapses once compensation converges.*

Error convergence is achieved by adaptively updating the ripple phase rate $\dot{\phi}$ based on the instantaneous deviation between commanded and measured voltage. The error signal drives small phase corrections so the controller remains aligned with the bus ripple despite slow parameter drift.

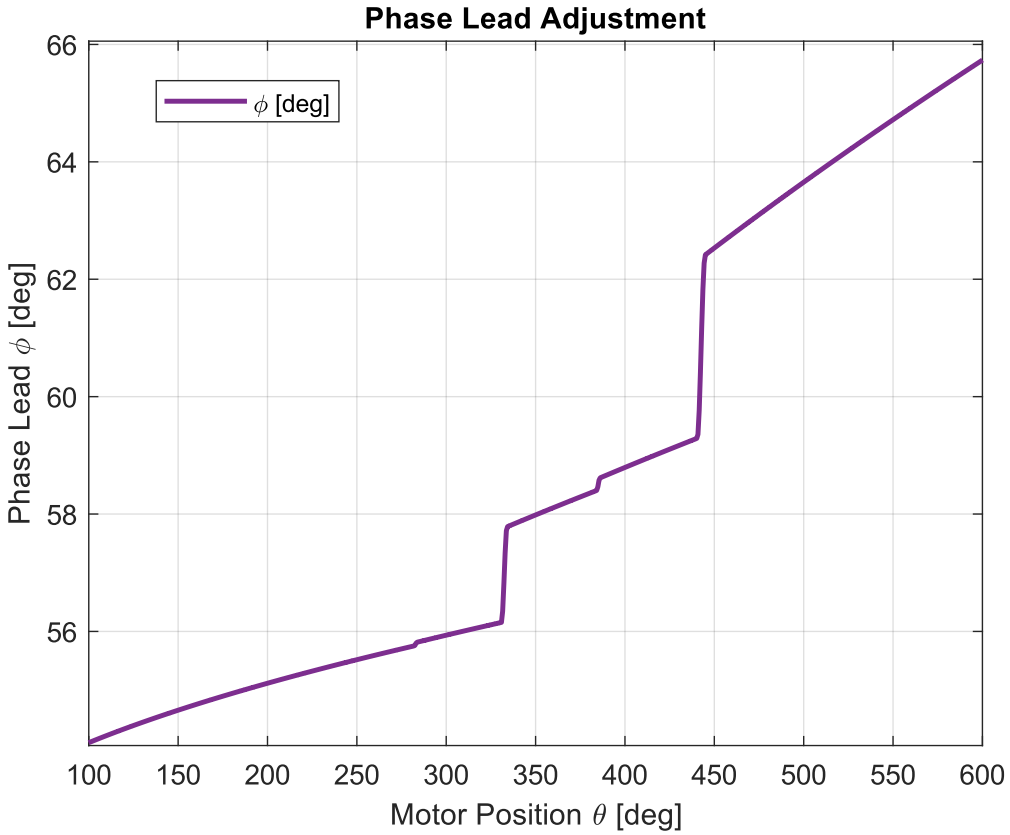


Figure 4.7: *Adaptive phase reference $\phi(t)$ used for ripple synchronization. Small, bounded corrections maintain phase lock.*

The adaptive phase reference $\phi(t)$ remains smooth and bounded throughout operation. The controller continuously adjusts ϕ_{raw} based on the ripple observed in V_{load}, maintaining alignment between the predicted and actual response while preserving overall phase stability, rather than attempting to follow instantaneous bus-phase fluctuations directly.

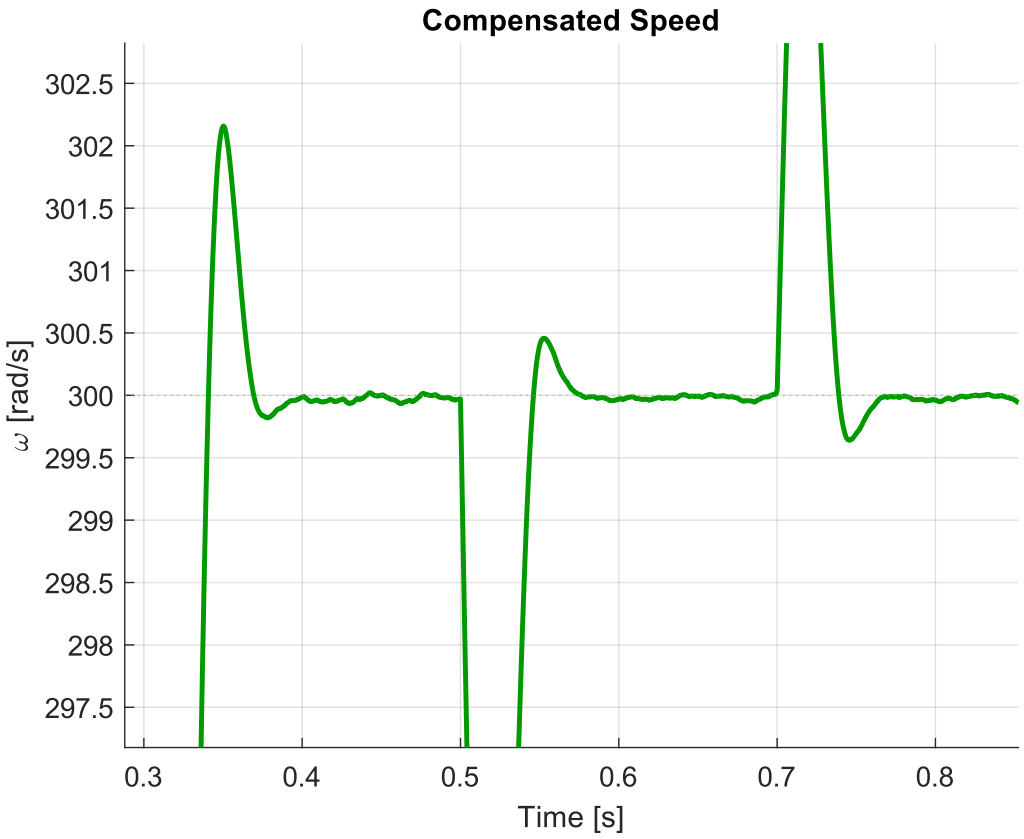


Figure 4.8: *Steady-state speed with compensator active, showing reduced* 120 Hz *modulation.*

With the compensator active, speed ripple is significantly reduced. In steady state, the variation in ω falls below 0.075 rad/s, representing an improvement of approximately 0.375 rad/s (about 83.3%) relative to the uncompensated condition. The larger 0.45 rad/s variation observed in earlier tests can be reproduced by employing a dual-voltage-monitor configuration that measures both the bus and load voltages directly, providing higher measurement resolution even without active compensation. Although modest in absolute terms, this reduction indicates a measurable increase in mechanical smoothness, which is particularly beneficial in precision or low-inertia systems.

The following results are obtained using the Motor 1 parameters listed in Table 4.1 at a 300 rad/s setpoint. Under comparable conditions, Motor 4 exhibits a

reduction from 1.4 rad/s to below 0.18 rad/s, corresponding to an 87.1% improvement in steady-state ripple. At higher operating speeds (e.g., 500 rad/s), the absolute ripple amplitude increases slightly, but the relative percentage improvement remains similar across configurations.

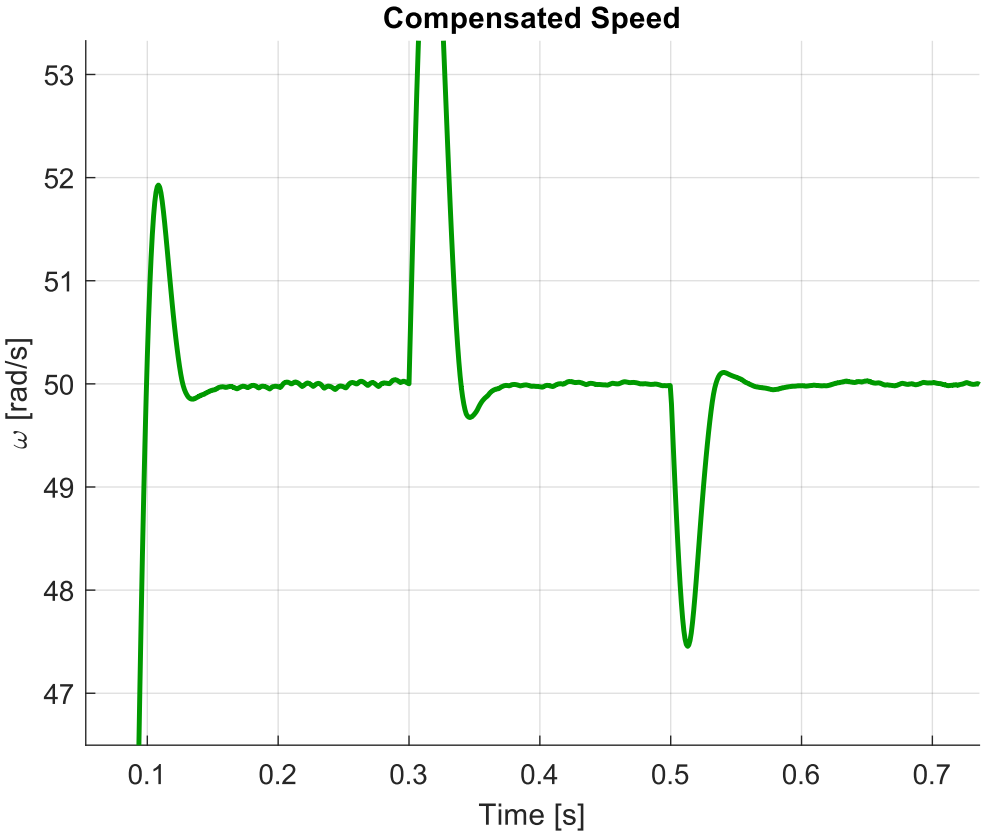


Figure 4.9: *Speed response at* 50 rad/s. *Ripple correction is minimal at low power.*

At lower speeds (e.g., 50 rad/s), the compensator continues to function correctly, but voltage scaling becomes increasingly important to preserve effective control authority. Because DC-bus sag is minimal at these low power levels, the compensator has little disturbance to correct, resulting in only minor improvement in measured ripple amplitude (approximately 0.03 rad/s using the Motor 1 parameters from Table 4.1). The present tuning was designed for moderate and higher-speed operation; further optimization of the allowable voltage error band could enhance responsiveness in the low-speed regime.

Table 4.1: *True theoretical parameters for Motors 1–4.*

Motor (Theoretical Values)	$a = k_t/J$	$b = \lambda/J$	k_v [V·s/rad]	R [mΩ]	L [μH]
Motor 1	3.7200	0.1000	0.1860	20.0	100.0
Motor 2	2.0000	0.0700	0.2000	15.0	120.0
Motor 3	5.6667	0.1333	0.1700	25.0	90.0
Motor 4	15.0000	0.2500	0.1200	60.0	60.0

Table 4.2: *Percent error of solver-identified parameters for Motors 1–4 using the* ***850 Hz*** *Butterworth filter.*

Motor (850 Hz Butterworth)	$a = k_t/J$	$b = \lambda/J$	k_v [V·s/rad]	R [mΩ]	L [μH]
Motor 1	−0.009%	+0.329%	+0.001%	−0.138%	+0.557%
Motor 2	+0.033%	+6.444%	−0.009%	−0.085%	+0.555%
Motor 3	−0.066%	−3.761%	+0.018%	−0.219%	+0.559%
Motor 4	−0.719%	−27.921%	+0.133%	−0.527%	+0.582%

Table 4.3: *Percent error of solver-identified parameters for Motors 1–4 using the* ***550 Hz*** *Butterworth filter.*

Motor (550 Hz Butterworth)	$a = k_t/J$	$b = \lambda/J$	k_v [V·s/rad]	R [mΩ]	L [μH]
Motor 1	+0.040%	+5.074%	+0.016%	−0.163%	+0.414%
Motor 2	+0.058%	+9.992%	+0.024%	−0.148%	+0.407%
Motor 3	+0.028%	+2.911%	+0.014%	−0.176%	+0.420%
Motor 4	−0.054%	−1.938%	+0.013%	−0.135%	+0.501%

Tables 4.1–4.3 summarize the percent error between solver-identified parameters and their theoretical values across four motor configurations and two Butterworth bandwidths. Among all parameters, $b = \lambda/J$ consistently exhibits the highest sensitivity to filtering and noise, with deviations up to −27.9% in the Motor 4 case at 850 Hz. This error has only a minor practical impact on closed-loop behavior, since

λ acts primarily as a viscous damping term whose variation is largely absorbed into the overall torque disturbance. In simulation, even substantial deviations in b produced no differences in steady-state performance, as overshoot, settling time, and phase alignment remained effectively unchanged while the steady-state ripple amplitude increased slightly in proportion to the damping-term error. In the worst case, the steady-state error band expanded by roughly 20–35% (increasing with speed), yet with the compensator active the resulting variation remained extremely small, on the order of a few hundredths of the setpoint value, rendering the practical influence negligible.

From our investigation, the comparatively poor fit of b appears to arise from two main factors. First, the effective window for low-frequency excitation during the launch cycle is limited. Second, there is a large scaling disparity between the damping torque term $\lambda\omega$ and the dominant electromagnetic torque term $k_t I$. This combination weakens the observability of λ/J and leads to greater estimation variance, particularly at wider filter bandwidths. While the review in [48] emphasized the benefits of steady-state fitting, our results suggest that under rectified (AC-fed) excitation such a regime offers no measurable advantage. Because bus ripple and commutation inherently disrupt steady-state conditions, launch-cycle fitting provides richer excitation and more robust overall identification. These findings indicate that isolating viscous damping in rectified drives remains an open problem that future hardware studies will examine further to determine whether λ can be more accurately identified through refined excitation or extended data-collection intervals.

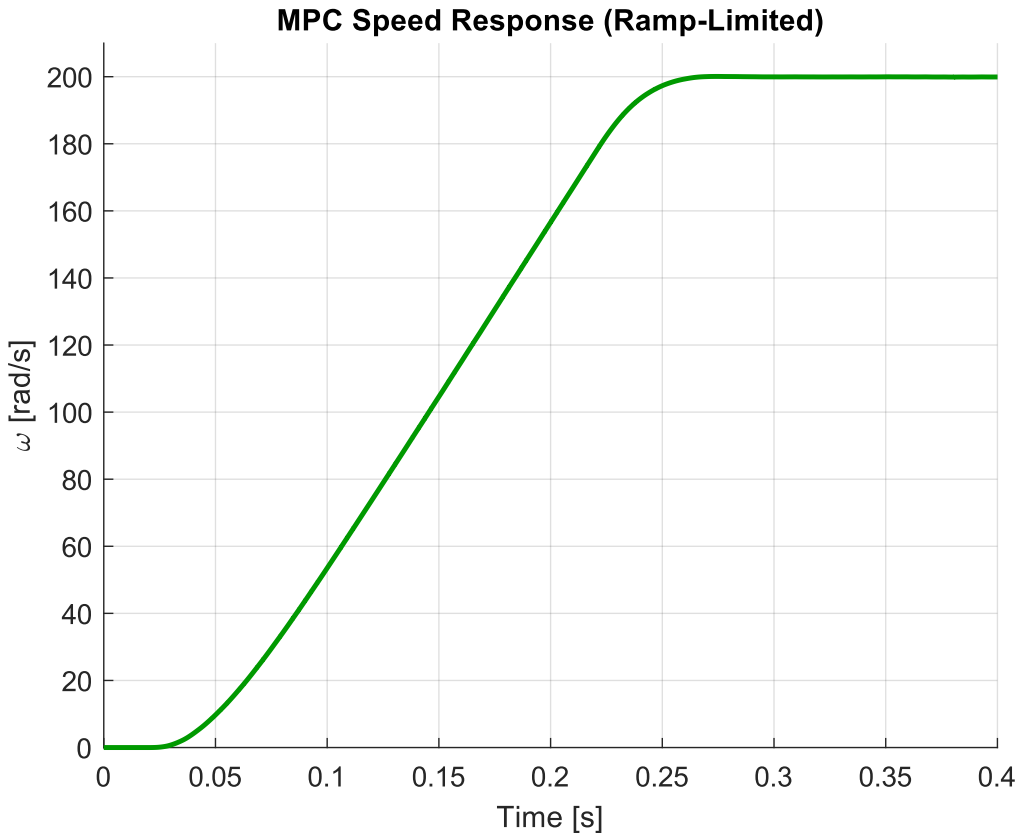


Figure 4.10: *Illustrative MPC-style preview control result at* $\omega^\star = 200$ rad/s *showing zero overshoot within the steady-state error band.*

A short demonstration with a finite preview horizon illustrates the benefit of model-based look-ahead. The controller employs a linear 810-move MPC structure, where each move represents a forward progression without change of direction. The total preview window corresponds to approximately 0.05 s (about 30 steps ahead at 600 Hz). Within this horizon, the response exhibits zero overshoot within the steady-state error band: ± 0.14 rad/s with compensation and ± 0.37 rad/s without. Although current limiting was not included, it could be incorporated with minimal complexity to provide a more complete and optimized control solution. This demonstration is not intended as a full model predictive control design study, but rather as an example of the practical capabilities enabled by a model-based framework for brushed DC motor diagnostics and control.

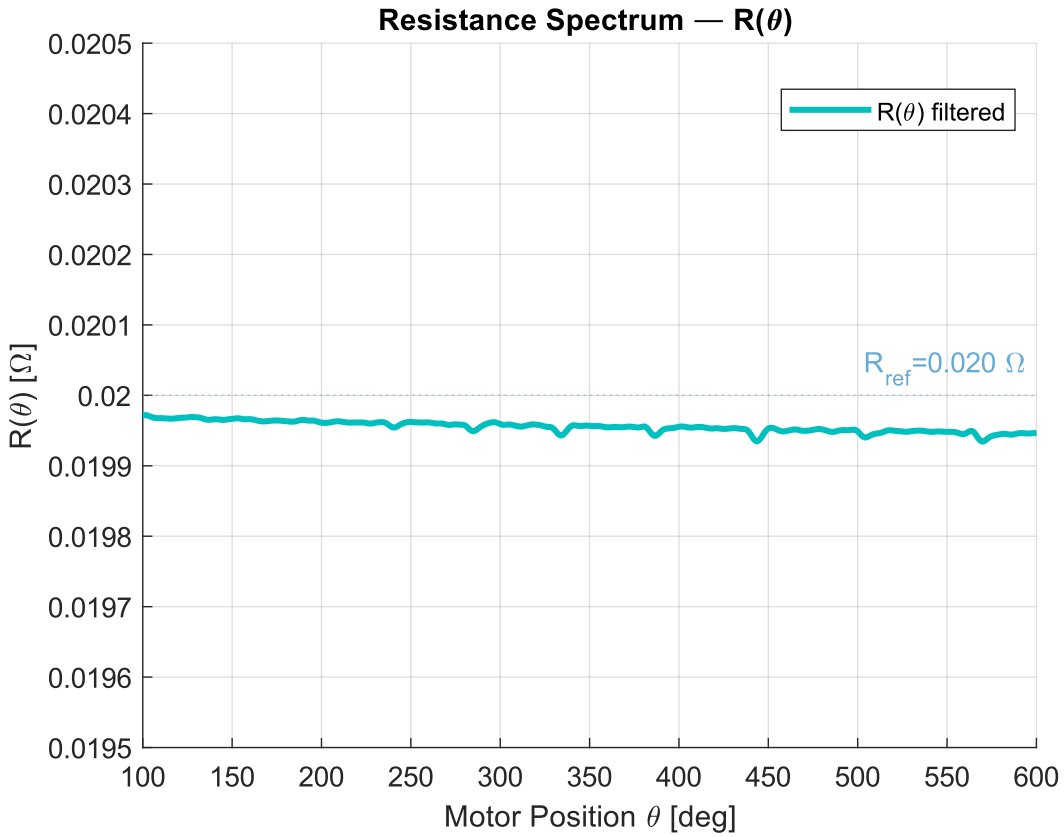


Figure 4.11: *Filtered resistance spectrum after compensation, showing uniform contact resistance and negligible distortion.*

Finally, the resistance-spectrum reconstruction confirms that the controller preserves the diagnostic signal. The raw $R(\theta)$ data are filtered using a fourth-order 120 Hz Butterworth filter to yield a clearly defined trace and suppress high-frequency numerical noise. Because the simulated $R(\theta)$ is constant, the resulting spectrum remains essentially flat apart from minor numerical noise and small residual ripple near the capacitor recharge edges. In a real motor, variations of this magnitude would be overshadowed by the true resistance spectrum, which is expected to vary substantially across commutator segments even under initial calibration.

Methodological limitation. One important limitation should be reemphasized. Both forms of resistance information produced by the framework depend on a stable

and well-characterized plant, so they remain accurate only while parameter drift is small. The first, $R_{\text{obs}}(t)$, is a continuously updated resistance estimate that supports real-time control and stability. The second, $R(\theta)$, is reconstructed during calibration to capture spatial variation in resistance for predictive maintenance. The compensator indirectly improves both by reducing bus sag and ripple distortion, creating a cleaner electrical environment for estimation, although it is less critical for the spectrum reconstruction itself. Accurate extraction of $R(\theta)$ still relies on the assumption of a nearly known plant, although recalibration is possible through the adaptive identification framework developed in this study. The spectrum shown above was therefore obtained under controlled calibration conditions that assume a known applied load torque, a condition that may not always be practical to reproduce in every motor application. In practice, small variations in the plant are expected to have only a minor influence on the reconstructed spectrum compared with the much larger localized resistive fluctuations produced by brush and commutator wear, which dominate the overall signal behavior.

Conclusion

The resistance-spectrum analysis demonstrates that the proposed control architecture preserves both stability and diagnostic integrity. By actively suppressing bus sag and ripple distortion, the compensator creates the clean electrical conditions required for accurate resistance estimation while simultaneously improving control performance. This stabilized environment allows the real-time resistance estimate $R_{\text{obs}}(t)$ and the position-resolved spectrum $R(\theta)$ to be obtained with minimal distortion, enabling both adaptive control and predictive maintenance functions within the same framework.

A natural next step will be to determine whether the resistance spectrum can be reconstructed under load once the plant has been fully established and calibrated. If achievable, this would substantially enhance the diagnostic value of the method by enabling resistance-based condition monitoring during normal operation rather than only during dedicated calibration cycles. Although periodic recalibration will remain necessary as parameters drift over time, the required frequency and extent of such recalibration must be determined experimentally through hardware testing.

The broader control strategy itself exhibits strong performance under idealized conditions. In a controlled simulation that included PWM modulation and 120 Hz bus ripple but excluded sensor and process noise, the adaptive controller demonstrated stable operation, effective phase-synchronized compensation, and accurate model identification. While these results represent best-case performance, they confirm that the proposed structure is theoretically sound and numerically stable. By suppressing bus ripple, the compensator reduces the need for passive filtering and lowers the required bulk capacitance, offering a path toward simpler and more efficient hardware implementations once practical noise and quantization effects are addressed.

During parameter identification, the primary limitation was the accuracy of the viscous damping term λ/J. Compared with other parameters, $b = \lambda/J$ showed higher sensitivity to noise and filter bandwidth, likely due to its small signal contribution relative to electromagnetic torque and the limited low-frequency excitation available in the launch-cycle dataset. While this discrepancy had no measurable effect on the simulated control response, it may become more evident in hardware, where frictional nonlinearities and environmental variation are present. Future studies will investigate whether λ can be more accurately isolated through targeted low-speed data collection or extended excitation, determining whether this limitation is intrinsic to rectified operation or primarily measurement-related.

Future work will therefore focus on hardware validation, operation under noisy and dynamic environments, and eventual extension to a fully sensorless AC-fed brushed-motor controller.

Bibliography

[1] KPixMining. Commutator inside electric dc motor. Dreamstime. Image ID 293593496. [Online]. Available: https://www.dreamstime.com/commutator-inside-electric-dc-motor-part-isolated-white-background-closeup-power-machine-rotor-carbon-brushes-coil-springs-image293593496

[2] J. A. Antonino-Daviu and P. M. V. Pla, "A non-intrusive method for sparking assessment in brush dc motors based on wavelet analysis," in *Proc. IEEE 47th Annu. Conf. of the Industrial Electronics Society (IECON)*, 2021, pp. 958–961.

[3] X. Liu, M. Hu, Z. Li, C. Zhou, Q. Xiao, W. Yang, and D. Chen, "Effect of copper contents on the current-carrying wear properties of carbon brush under different temperatures conditions," *Journal of Materials Research and Technology*, vol. 15, pp. 3110–3121, 2021.

[4] A. Hughes and B. Drury, *Electric Motors and Drives: Fundamentals, Types and Applications*, 5th ed. Elsevier, 2019.

[5] P. C. Sen, *Principles of Electric Machines and Power Electronics*, 3rd ed. Wiley, 2013.

[6] R. Krishnan, *Electric Motor Drives: Modeling, Analysis, and Control*. Upper Saddle River, NJ, USA: Prentice Hall, 2001.

[7] B. K. Bose, *Modern Power Electronics and AC Drives*. Prentice Hall, 2002.

[8] M. Petushkov, D. Shcherbina, and A. Sarvarov, "Diagnostics of dc motors in the mining and processing production of ojsc 'mmk'," in *Proc. IEEE Conf. Russian Young Researchers in Electrical and Electronic Engineering (EIConRus)*, 2019, pp. 1589–1593.

[9] V. V. Kharlamov, D. I. Popov, Y. V. Moskaleva, and M. F. Baysadykova, "Model of electric-brush wear including brush–commutator dynamics," *Russian Engineering Research*, vol. 42, no. 4, pp. 330–335, 2022.

[10] R. Liu, K. Cheng, J. Chen, X. Xiong, and X. Lin, "Friction and wear properties of high-temperature and low-temperature sintered copper-graphite brushes at different ambient temperatures," *Journal of Materials Research and Technology*, vol. 9, no. 4, pp. 7288–7296, 2020.

[11] A. A. Wahab, N. F. Abdullah, and M. A. H. Rasid, "Commutator fault detection of brushed dc motor using thermal assessment," *IOP Conference Series: Materials Science and Engineering*, vol. 469, p. 012057, 2019.

[12] N. F. Abdullah, A. A. Wahab, and M. A. H. Rasid, "Study of temperature rise of small brushed dc motor under different load," *MATEC Web of Conferences*, vol. 225, p. 03010, 2018.

[13] T. Li, T. Feng, Y. Wang, and X. Xu, "Experimental study and numerical analysis of vacuum thermal characteristics of brush dc motor," *Energy Storage and Saving*, vol. 2, pp. 459–466, 2023.

[14] K. Sawa, T. Yamamoto, and T. Ueno, "Evaluation of arc erosion of cu-graphite brush used in small dc motors with and without quenching device," in *Proc. IEEE Holm Conf. on Electrical Contacts (HOLM)*, 2023, pp. 1–6.

[15] Y. Niwa and Y. Akiyama, "The relation of a brush life to the commutator width and brush width ratio of a universal motor," in *Proc. IEEE Power Electronics and Drive Systems*, Singapore, 2009, pp. 1386–1390.

[16] E. Spunei, I. Piroi, and F. Piroi, "Influence of brush positioning on separately excited dc motor operation," in *Proc. Int. Conf. and Exposition on Electrical and Power Engineering (EPE)*, Iaşi, Romania, 2022, pp. 51–56.

[17] J. K. Lancaster, "The influence of arcing on the wear of carbon brushes on copper," *Wear*, vol. 6, no. 5, pp. 341–352, 1963.

[18] K. Sawa, K. Nakano, T. Ueno, and K. Toya, "Effects of brush spring force on brush wear and arc behaviors of small dc motor," in *Proc. IEEE 68th Holm Conf. on Electrical Contacts (HOLM)*, 2023, pp. 1–6.

[19] K. Sawa, Y. Watanabe, and T. Ueno, "Mechanical wear and arc erosion of brush and commutator in dc motor for automotive fuel pump," in *Proc. IEEE Holm Conf. on Electrical Contacts*, Albuquerque, NM, USA, 2018, pp. 412–418.

[20] W.-G. Shin, Y.-S. Song, and Y.-K. Seo, "Correlation analysis of brush temperature in brush-type dc motor for predicting motor life," *Journal of Mechanical Science and Technology*, vol. 26, no. 7, pp. 2151–2154, 2012.

[21] A. A. Wahab and M. A. H. Rasid, "Characterization of brushed dc motor with brush fault using thermal assessment," *IOP Conference Series: Materials Science and Engineering*, vol. 788, no. 1, p. 012095, 2020.

[22] G. C. R. Sincero, J. Cros, and P. Viarouge, "Arc models for simulation of brush motor commutations," *IEEE Transactions on Magnetics*, vol. 44, no. 6, pp. 1519–1522, Jun. 2008.

[23] P. M. Velasco-Pla and J. A. Antonino-Daviu, "Sparking detection in brushed dc motors through the analysis of the armature current under the starting," in *Proc. Int. Conf. on Electrical Machines (ICEM)*, 2022, pp. 1537–1541.

[24] A. A. Sinyavskiy and A. P. Maltsev, "Increasing the efficiency of diagnostics in the brush-commutator assembly of a direct current electric motor," in *Proc. Int. Conf. on Industrial Engineering, Applications and Manufacturing (ICIEAM)*, Moscow, Russia, 2018.

[25] V. G. Titov, M. N. Kolchev, and E. A. Melikhov, "Vibration diagnostics of the brush-collector assembly as means of assessing motor wear," in *Proc. Int. Conf. on Diagnostics in Electrical Engineering (Diagnostika)*, Prague, Czech Republic, 2021.

[26] F. Zhang, Y. Yang, and Z. Cai, "Multifractal spectrum and complex cepstrum analysis of armature currents and stray flux signals for sparking detection in dc motors," in *Proc. Int. Conf. on Electrical Machines and Systems (ICEMS)*, 2021.

[27] W. Liu, X. Zheng, and H. Yang, "Study the influence of power-current impact with different conditions on the current-carrying wear property of carbon brush/friction disc contact," *Tribology International*, vol. 186, 2023.

[28] A. Laszlo and M. Kalacska, "Electrical contact resistance and wear of a dynamic graphite-copper contact under sliding conditions," *Tribology International*, vol. 153, p. 106610, 2021.

[29] T. Yang, S. Li, S. Duan, Y. Tao, and X. Han, "Performance degradation model and reliability evaluation of brush dc motor for the intelligent on–off valve," *Journal of Electrical Engineering & Technology*, vol. 18, no. 3, pp. 1909–1918, 2023.

[30] B. Benedik, K. Taškova, J. Tavčar, and J. Duhovnik, "Prediction of vacuum cleaner motor brush life: A regression approach," *IET Electric Power Applications*, vol. 9, no. 9, pp. 569–577, 2015.

[31] J. Hu *et al.*, "Application of hmm in fault diagnosis of permanent magnet dc motors," in *Proc. IEEE PHM-Beijing*, 2024.

[32] J. D. Wang, Y. X. Zhou, and J. Y. Li, "A new entropy bi-cepstrum based method for dc motor brush abnormality recognition," *IEEE Transactions on Industry Applications*, vol. 57, no. 4, pp. 3392–3401, Jul./Aug. 2021.

[33] Bundasak and Wittayasirikul, "Predictive maintenance using ai for motor health prediction system," in *Proc. iEECON*, 2022.

[34] Dongare *et al.*, "Incipient faults detection in induction motor using mlp-nn and rbf-nn-based fault classifier," in *Proc. IEEE STPEC*, 2021.

[35] Chitra *et al.*, "Brushless dc motor with intelligent fault detection method based on rnn," in *Proc. IEEE RAEEUCCI*, 2024.

[36] Qiu *et al.*, "Application of artificial intelligence-based technique in electric motors: A review," *IEEE Transactions on Power Electronics*, vol. 39, no. 10, 2024.

[37] R. Khelif, B. Chebel-Morello, S. Malinowski, E. Laajili, F. Fnaiech, and N. Zerhouni, "Direct remaining useful life estimation based on support vector regression," *IEEE Transactions on Industrial Electronics*, vol. 64, no. 3, pp. 2276–2285, Mar. 2017.

[38] Y. Ren, C. H. Liu, and X. Zhang, "Online rotor and stator resistance estimation based on model reference adaptive system," in *Proc. IEEE 4th Conf. on Energy Internet and Energy System Integration (EI2)*, Wuhan, China, 2020, pp. 2573–2577.

[39] A. E. Fitzgerald, C. K. Jr., and S. D. Umans, *Electric Machinery*, 7th ed. McGraw-Hill, 2013.

[40] F. A. Hasan, *Electrical Machines I*, 2016, module I: DC Machines — Lap and Wave Windings. [Online]. Available: https://www.uotechnology.edu.iq/eng/dep$_e$*lectrical*/

[41] F. Borrelli, A. Bemporad, and M. Morari, *Predictive Control for Linear and Hybrid Systems*. Cambridge University Press, 2017.

[42] T. Nowak, D. Rutkowski, and Z. Stawarski, "Model predictive control for dc motors," *Przeglad Elektrotechniczny (Electrical Review)*, vol. 88, no. 12b, pp. 118–121, 2012.

[43] C. Luo, Z. Q. Zhu, and D. Howe, "Finite control set model predictive speed control of a chopper-fed pmdc motor drive," *AEÜ - International Journal of Electronics and Communications*, vol. 98, pp. 53–60, 2019.

[44] A. Iqbal, S. Mekhilef, H. Hizam, and N. A. Rahim, "Finite control set model predictive speed control of a permanent magnet dc motor drive incorporating kalman filter observer," *International Journal of Power Electronics and Drive Systems*, vol. 9, no. 4, pp. 1683–1691, Dec. 2018.

[45] F. M. Gardner, *Phaselock Techniques*, 3rd ed. Wiley, 2005.

[46] S. Buso and P. Mattavelli, *Digital Control in Power Electronics*. Springer, 2023.

[47] A. V. Peterchev and S. R. Sanders, "Quantization resolution and limit cycling in digitally controlled pwm converters," *IEEE Transactions on Power Electronics*, vol. 18, no. 1, pp. 301–308, Jan. 2003.

[48] M. F. Fazdi and P.-W. Hsueh, "Parameters identification of a permanent magnet dc motor: A review," *Electronics*, vol. 12, p. 2559, 2023.

[49] A. V. Oppenheim and R. W. Schafer, *Discrete-Time Signal Processing*, 2nd ed. Upper Saddle River, NJ, USA: Prentice Hall, 1999.

[50] A. Savitzky and M. J. E. Golay, "Smoothing and differentiation of data by simplified least squares procedures," *Analytical Chemistry*, vol. 36, no. 8, pp. 1627–1639, 1964.

[51] S. C. Chapra and R. P. Canale, *Numerical Methods for Engineers*, 7th ed. New York: McGraw–Hill Education, 2015.

[52] L. Ljung, *System Identification: Theory for the User*, 2nd ed. Prentice Hall, 1999.

[53] Y.-T. Huang, C.-C. Yang, T.-S. Li, and Y.-M. Chen, "A feedforward voltage control strategy for reducing the output voltage double-line-frequency ripple in single-phase ac–dc converters," *IEEE Journal of Emerging and Selected Topics in Power Electronics*, vol. 9, no. 6, pp. 6605–6612, Dec. 2021.

```
function [a, b, c, g, L_out, ...
          ready_low, ready_high, ...
          buffer_count, LOW_buf_count, HIGH_buf_count,
              VBUS_buf_count, ...
          R_theta_sample, theta_sample, i_theta_sample,
              v_theta_sample, ...
          D_coeffs, D_coeffs_buffer_ready, D_fit_K_used, ...
          D_k_gain, res_phase_sample, res_duty_sample,
              res_fit_sample] = ...
          solver_block(I_low, V_low, omega_low, ...              %
              LOW stream (mechanical)
                       I_high, V_high, omega_high, theta_high, ...%
                           HIGH stream (electrical)
                       V_bus, phase, ...                         %
                           D_fit sampler (Vbus fit)
                       start_sampling, stop_mech, stop_elec,
                           stop_vbus, ...
                       clock)
%#codegen
% LOW  -> mechanical (a,b)
% HIGH -> electrical (L,R, c k_v ) +   -overlay stream
% D_fit-> Vbus(  ) harmonic fit + overlay

    %==================== FIXED CONFIG ====================
    V_FIT_K = 8;
    M_RES   = 1 + 2*V_FIT_K;
    VB_fit_min_samples     = 10000;
    VB_fit_sample_interval = 5e-5;    % 50 us
    k_sigma = 3;

    % LOW (mechanical)
    LOW_min_samples       = 20000;
```

```
    LOW_sample_interval   = 0.75e-5;
    LOW_downsample_ticks  = round(LOW_sample_interval / 5e-7);

    % HIGH (electrical)
    HIGH_min_samples       = 20000;
    HIGH_sample_interval   = 0.75e-5;
    HIGH_downsample_ticks  = round(HIGH_sample_interval / 5e-7);

    %   -overlay (uniform in   )
    THETA_fit_dtheta       = 1e-3;      % rad/sample (~0.057   )
    THETA_fit_min_points   = 256;
    THETA_FIT_max_capacity = 100000;    % long streams

    % Other constants
    SIM_MODE              = true;
    I_MIN_SIM             = 1e-12;   I_MIN_HW = 0.5;
    I_MIN_FOR_R_BASE      = (SIM_MODE) * I_MIN_SIM + (~SIM_MODE) *
       I_MIN_HW;

    %================== Outputs (fixed-size) ==================
    a=0; b=0; c=0; g=0; L_out=0;
    ready_low=0; ready_high=0;
    buffer_count=0; LOW_buf_count=0; HIGH_buf_count=0;
       VBUS_buf_count=0;

    R_theta_sample=0; theta_sample=0; i_theta_sample=0;
       v_theta_sample=0;

    D_coeffs       = zeros(1, M_RES);
    D_coeffs_buffer_ready = 0;
    D_fit_K_used   = V_FIT_K;
    D_k_gain       = 0.0;
```

```
    res_phase_sample = 0.0;
    res_duty_sample  = 0.0;
    res_fit_sample   = 0.0;

    %================ Buffers & State ================
    % LOW (mechanical)
    persistent LOW_I_buf LOW_V_buf LOW_omega_buf LOW_buf_idx
       LOW_downsample_counter low_sampling_active
    % HIGH (electrical)
    persistent HIGH_I_buf HIGH_V_buf HIGH_omega_buf HIGH_theta_buf
       HIGH_buf_idx HIGH_downsample_counter high_sampling_active
    % Solves / flags / outputs
    persistent low_finalized high_finalized a_ b_ R_ c_ L_
       ready_flag_low ready_flag_high

    %   -overlay storage / drip
    persistent THETA_vec_persist I_theta_vec_persist
       V_theta_vec_persist R_theta_vec_persist
    persistent theta_readout_started theta_readout_idx
       theta_final_count theta_solved_once

    % D_fit (VBUS fit)
    persistent VB_Vbus_buf VB_phase_buf VB_buf_idx
       vb_sampling_active
    persistent VB_coeffs_persist VB_ready_persist VB_final_count
       VB_k_gain_persist VB_solved_once

    % D_fit resampler state
    persistent VB_seeded VB_prev_t VB_prev_v VB_prev_phase
       VB_next_emit_t

    % D_fit overlay stream
    persistent RES_phase_vec_persist RES_y_vec_persist
```

```
    RES_yhat_vec_persist
persistent res_readout_started res_readout_idx final_res_count

% Edge detect memories
persistent prev_start_sampling prev_stop_mech prev_stop_elec
    prev_stop_vbus

if isempty(LOW_I_buf)
    % Capacities
    LOW_max_capacity     = LOW_min_samples * 2;
    HIGH_max_capacity    = HIGH_min_samples * 2;
    VB_FIT_max_capacity  = VB_fit_min_samples * 2;

    % Alloc LOW
    LOW_I_buf      = zeros(1, LOW_max_capacity);
    LOW_V_buf      = zeros(1, LOW_max_capacity);
    LOW_omega_buf  = zeros(1, LOW_max_capacity);
    LOW_buf_idx    = 0; LOW_downsample_counter = 0;
        low_sampling_active=false;

    % Alloc HIGH
    HIGH_I_buf      = zeros(1, HIGH_max_capacity);
    HIGH_V_buf      = zeros(1, HIGH_max_capacity);
    HIGH_omega_buf  = zeros(1, HIGH_max_capacity);
    HIGH_theta_buf  = zeros(1, HIGH_max_capacity);
    HIGH_buf_idx    = 0; HIGH_downsample_counter = 0;
        high_sampling_active=false;

    % Flags/outputs
    low_finalized=false; high_finalized=false;
    a_=0; b_=0; R_=0; c_=0; L_=0;
    ready_flag_low=false; ready_flag_high=false;

```

```
110         %   -overlay
111         THETA_vec_persist      = zeros(1, THETA_FIT_max_capacity);
112         I_theta_vec_persist    = zeros(1, THETA_FIT_max_capacity);
113         V_theta_vec_persist    = zeros(1, THETA_FIT_max_capacity);
114         R_theta_vec_persist    = zeros(1, THETA_FIT_max_capacity);
115         theta_readout_started  = false;
116         theta_readout_idx      = 1;
117         theta_final_count      = 0;
118         theta_solved_once      = false;
119
120         % D_fit
121         VB_Vbus_buf   = zeros(1, VB_FIT_max_capacity);
122         VB_phase_buf  = zeros(1, VB_FIT_max_capacity);
123         VB_buf_idx    = 0; vb_sampling_active=false;
124
125         VB_coeffs_persist  = zeros(1, M_RES);
126         VB_ready_persist   = 0;
127         VB_final_count     = 0;
128         VB_k_gain_persist  = 0.0;
129         VB_solved_once     = false;
130
131         VB_seeded      = false;
132         VB_prev_t      = 0; VB_prev_v = 0; VB_prev_phase = 0;
               VB_next_emit_t = 0;
133
134         RES_phase_vec_persist = zeros(1, VB_FIT_max_capacity);
135         RES_y_vec_persist     = zeros(1, VB_FIT_max_capacity);
136         RES_yhat_vec_persist  = zeros(1, VB_FIT_max_capacity);
137         res_readout_started   = false;
138         res_readout_idx       = 1;
139         final_res_count       = 0;
140
141         prev_start_sampling=0; prev_stop_mech=0; prev_stop_elec=0;
```

```
        prev_stop_vbus=0;
142     end
143
144     %================= Edge detect =================
145     rising_start = (start_sampling ~= 0) && (prev_start_sampling ==
            0);
146     rising_low   = (stop_mech      ~= 0) && (prev_stop_mech      ==
            0);
147     rising_high  = (stop_elec      ~= 0) && (prev_stop_elec      ==
            0);
148     rising_vbus  = (stop_vbus      ~= 0) && (prev_stop_vbus      ==
            0);
149     prev_start_sampling = start_sampling;
150     prev_stop_mech      = stop_mech;
151     prev_stop_elec      = stop_elec;
152     prev_stop_vbus      = stop_vbus;
153
154     %============== Start edge: reset ==============
155     if rising_start
156         % LOW
157         LOW_buf_idx=0; LOW_downsample_counter=0; low_sampling_active
                =true;
158         % HIGH
159         HIGH_buf_idx=0; HIGH_downsample_counter=0;
                high_sampling_active=true;
160
161         % D_fit
162         VB_buf_idx=0; vb_sampling_active=true; VB_solved_once=false;
163         VB_coeffs_persist(:)=0; VB_ready_persist=0; VB_final_count
                =0; VB_k_gain_persist=0.0;
164
165         % Flags/outputs
166         low_finalized=false; high_finalized=false;
```

```
167         a_=0; b_=0; R_=0; c_=0; L_=0;
168         ready_flag_low=false; ready_flag_high=false;
169
170         %   -overlay
171         THETA_vec_persist(:)=0; I_theta_vec_persist(:)=0;
               V_theta_vec_persist(:)=0; R_theta_vec_persist(:)=0;
172         theta_readout_started=false; theta_readout_idx=1;
               theta_final_count=0; theta_solved_once=false;
173
174         % VBUS overlay
175         RES_phase_vec_persist(:)=0; RES_y_vec_persist(:)=0;
               RES_yhat_vec_persist(:)=0;
176         res_readout_started=false; res_readout_idx=1;
               final_res_count=0;
177
178         % Resampler seed
179         VB_seeded=false; VB_prev_t=0; VB_prev_v=0; VB_prev_phase=0;
               VB_next_emit_t=0;
180     end
181
182     %================ Sampling =================
183     if low_sampling_active
184         LOW_downsample_counter = LOW_downsample_counter + 1;
185         if LOW_downsample_counter >= LOW_downsample_ticks
186             LOW_downsample_counter = 0;
187             LOW_buf_idx = LOW_buf_idx + 1;
188             LOW_I_buf(LOW_buf_idx)      = I_low;
189             LOW_V_buf(LOW_buf_idx)      = V_low;
190             LOW_omega_buf(LOW_buf_idx)  = omega_low;
191         end
192         if ~ready_flag_low && (LOW_buf_idx >= LOW_min_samples)
193             ready_flag_low = true;
194         end
```

```
195     end
196
197     if high_sampling_active
198         HIGH_downsample_counter = HIGH_downsample_counter + 1;
199         if HIGH_downsample_counter >= HIGH_downsample_ticks
200             HIGH_downsample_counter = 0;
201             HIGH_buf_idx = HIGH_buf_idx + 1;
202             HIGH_I_buf(HIGH_buf_idx)      = I_high;
203             HIGH_V_buf(HIGH_buf_idx)      = V_high;
204             HIGH_omega_buf(HIGH_buf_idx)  = omega_high;
205             HIGH_theta_buf(HIGH_buf_idx)  = theta_high;
206         end
207         if ~ready_flag_high && (HIGH_buf_idx >= HIGH_min_samples)
208             ready_flag_high = true;
209         end
210     end
211
212     %================= VBUS resampler (uniform time)
        ==================
213     if vb_sampling_active && ~VB_solved_once
214         if ~VB_seeded
215             VB_prev_t     = clock;
216             VB_prev_v     = V_bus;
217             VB_prev_phase = phase;
218             VB_next_emit_t= VB_prev_t + VB_fit_sample_interval;
219             VB_seeded     = true;
220         end
221         if clock > VB_prev_t
222             while (VB_next_emit_t < clock) && (VB_buf_idx < numel(
                    VB_Vbus_buf))
223                 tau = (VB_next_emit_t - VB_prev_t) / (clock -
                        VB_prev_t);
224                 v_i  = VB_prev_v     + tau * (V_bus  - VB_prev_v);
```

```
225                 ph_i = VB_prev_phase + tau * (phase  - VB_prev_phase
                        );
226                 VB_buf_idx = VB_buf_idx + 1;
227                 VB_Vbus_buf(VB_buf_idx)  = v_i;
228                 VB_phase_buf(VB_buf_idx) = ph_i;
229                 VB_next_emit_t = VB_next_emit_t +
                        VB_fit_sample_interval;
230             end
231             VB_prev_t     = clock;
232             VB_prev_v     = V_bus;
233             VB_prev_phase = phase;
234         end
235     end
236
237     %================ Finalize LOW ================
238     if low_sampling_active && rising_low
239         low_sampling_active = false;
240         [~, ~, ~, ~, ~] = legacy_solver_band( ...
241             LOW_V_buf(1:LOW_buf_idx), LOW_I_buf(1:LOW_buf_idx),
                    LOW_omega_buf(1:LOW_buf_idx), ...
242             LOW_sample_interval, 'LOW', k_sigma);
243         low_finalized = true;
244     end
245
246     %================ Finalize HIGH ================
247     if high_sampling_active && rising_high
248         high_sampling_active = false;
249
250         % Solve L,R,c and a,b from HIGH band
251         [L_high, R_high, c_high, a_high, b_high] =
                legacy_solver_band( ...
252             HIGH_V_buf(1:HIGH_buf_idx), HIGH_I_buf(1:HIGH_buf_idx),
                    HIGH_omega_buf(1:HIGH_buf_idx), ...
```

```
253             HIGH_sample_interval, 'HIGH', k_sigma);
254
255         L_ = L_high; R_ = R_high; c_ = c_high;
256         a_ = a_high;  b_ = b_high;
257
258         % ====   -overlay (unchanged) ====
259         N = HIGH_buf_idx;
260         if N >= 5
261             Ih_raw = HIGH_I_buf(1:N);
262             Vh_raw = HIGH_V_buf(1:N);
263             Oh_raw = HIGH_omega_buf(1:N);
264             Th_raw = HIGH_theta_buf(1:N);
265
266             Ih_d = Ih_raw;  Vh_d = Vh_raw;  Oh_d = Oh_raw;
267
268             dI = zeros(1,N);
269             if N >= 2
270                 dI(1) = (Ih_d(2) - Ih_d(1)) / HIGH_sample_interval;
271                 dI(N) = (Ih_d(N) - Ih_d(N-1)) / HIGH_sample_interval
                        ;
272             end
273             if N >= 3
274                 dI(2:N-1) = (Ih_d(3:N) - Ih_d(1:N-2)) / (2*
                        HIGH_sample_interval);
275             end
276
277             Th_unwrap = unwrap_theta_row(Th_raw);
278             Th_int = Th_unwrap(1) + cumsum(Oh_d)*
                    HIGH_sample_interval;
279
280             Afit = [Th_int(:), ones(numel(Th_int),1)];
281             xfit = Afit \ Th_unwrap(:);
282             alpha = xfit(1); beta = xfit(2);
```

```
283
284             Th_dense = alpha .* Th_int + beta;
285
286             Lc = min([numel(Th_dense), numel(Ih_d), numel(Vh_d),
                    numel(Oh_d), numel(dI)]);
287             Th_dense = Th_dense(1:Lc);
288             Irow     = Ih_d(1:Lc);
289             Vrow     = Vh_d(1:Lc);
290             Wrow     = Oh_d(1:Lc);
291             dIrow    = dI(1:Lc);
292
293             mfinite = isfinite(Th_dense) & isfinite(Irow) & isfinite
                    (Vrow) & isfinite(Wrow) & isfinite(dIrow);
294             Th_dense = Th_dense(mfinite);  Irow = Irow(mfinite);
295             Vrow     = Vrow(mfinite);      Wrow = Wrow(mfinite);
296             dIrow    = dIrow(mfinite);
297
298             if numel(Th_dense) >= 3
299                 th_min = Th_dense(1);
300                 th_max = Th_dense(end);
301                 span   = max(th_max - th_min, eps);
302
303                 dth   = THETA_fit_dtheta;
304                 M_est = max(THETA_fit_min_points, floor(span/dth));
305                 M_out = min(M_est, THETA_FIT_max_capacity);
306
307                 if M_out >= 2
308                     theta_grid = linspace(th_min, th_max, M_out);
309                     I_th  = interp1(Th_dense, Irow,  theta_grid, '
                            linear');
310                     V_th  = interp1(Th_dense, Vrow,  theta_grid, '
                            linear');
311                     w_th  = interp1(Th_dense, Wrow,  theta_grid, '
```

```
                        linear');
312                     dI_th = interp1(Th_dense, dIrow, theta_grid, '
                        linear');
313
314                     % R(  ) with current floor
315                     absI = abs(I_th);
316                     p05  = pct1(absI, 5);
317                     I_MIN_FOR_R = max(p05, I_MIN_FOR_R_BASE);
318                     y_fit = V_th - (L_.*dI_th + c_.*w_th);
319                     denom = max(absI, I_MIN_FOR_R) .* sign(I_th);
320                     R_th  = y_fit ./ denom;
321
322                     THETA_vec_persist(:)=0; I_theta_vec_persist(:)
                        =0; V_theta_vec_persist(:)=0;
                        R_theta_vec_persist(:)=0;
323                     THETA_vec_persist(1:M_out)   = theta_grid;
324                     I_theta_vec_persist(1:M_out) = I_th;
325                     V_theta_vec_persist(1:M_out) = V_th;
326                     R_theta_vec_persist(1:M_out) = R_th;
327
328                     theta_final_count     = M_out;
329                     theta_readout_started = false;
330                     theta_readout_idx     = 1;
331                     theta_solved_once     = true;
332                 else
333                     zero_theta_overlay();
334                 end
335             else
336                 zero_theta_overlay();
337             end
338         else
339             zero_theta_overlay();
340         end
```

```
341
342         high_finalized = true;
343     end
344
345     %================ Finalize D_fit (VBUS) ================
346     if vb_sampling_active && rising_vbus
347         vb_sampling_active = false;
348
349         vbn = VB_buf_idx;
350         if vbn >= (2*V_FIT_K + 1)
351             vbus_full = VB_Vbus_buf(1:vbn);
352             ph_full   = VB_phase_buf(1:vbn);
353
354             % Harmonic design matrix
355             K   = vbn;
356             Yk  = vbus_full(:);
357             Xk  = zeros(K, M_RES); Xk(:,1) = 1.0;
358
359             c1 = cos(ph_full(:)); s1 = sin(ph_full(:));
360             ck = c1; sk = s1; col = 2;
361             for kk = 1:V_FIT_K
362                 Xk(:,col) = ck; col = col+1;
363                 Xk(:,col) = sk; col = col+1;
364                 ck_next = c1.*ck - s1.*sk;
365                 sk_next = s1.*ck + c1.*sk;
366                 ck = ck_next; sk = sk_next;
367             end
368
369             theta_coef = (Xk.'*Xk) \ (Xk.'*Yk);
370
371             resid = Xk*theta_coef - Yk;
372             sMAD  = 1.4826*median(abs(resid - median(resid))) + eps;
373             keep  = abs(resid) < k_sigma*sMAD;
```

```
374             if nnz(keep) >= (M_RES + 8)
375                 X_used = Xk(keep,:); Y_used = Yk(keep);
376                 theta_coef  = (X_used.'*X_used) \ (X_used.'*Y_used);
377             end
378
379             h_vec = theta_coef(2:end);
380             VB_k_gain_persist     = sqrt(sum(h_vec.^2));
381             VB_coeffs_persist(:)  = 0;
382             VB_coeffs_persist(1:M_RES) = theta_coef.';  % row
383
384             % Overlay vectors
385             yhat_full = Xk * theta_coef;
386             RES_phase_vec_persist(:)=0; RES_y_vec_persist(:)=0;
                   RES_yhat_vec_persist(:)=0;
387             limitN = min(vbn, numel(RES_phase_vec_persist));
388             if limitN > 0
389                 RES_phase_vec_persist(1:limitN) = ph_full(1:limitN);
390                 RES_y_vec_persist(1:limitN)     = vbus_full(1:limitN
                       );
391                 RES_yhat_vec_persist(1:limitN)  = yhat_full(1:limitN
                       );
392             end
393             final_res_count      = limitN;
394             res_readout_started  = false; res_readout_idx = 1;
395
396             VB_ready_persist = 1;
397         else
398             VB_coeffs_persist(:)=0; VB_ready_persist=0;
399             final_res_count=0; VB_k_gain_persist=0.0;
400         end
401
402         VB_solved_once = true;
403     end
```

```
404
405     %================ Drip readouts =================
406     if theta_solved_once && (theta_final_count > 0)
407         if ~theta_readout_started
408             theta_readout_started = true; theta_readout_idx = 1;
409         end
410         limt = min(theta_final_count, numel(THETA_vec_persist));
411         if theta_readout_idx <= limt
412             theta_sample   = THETA_vec_persist(theta_readout_idx);
413             i_theta_sample = I_theta_vec_persist(theta_readout_idx);
414             v_theta_sample = V_theta_vec_persist(theta_readout_idx);
415             R_theta_sample = R_theta_vec_persist(theta_readout_idx);
416             theta_readout_idx  = theta_readout_idx + 1;
417         else
418             R_theta_sample = 0; theta_sample = 0; i_theta_sample =
                    0; v_theta_sample = 0;
419         end
420     else
421         R_theta_sample = 0; theta_sample = 0; i_theta_sample = 0;
                v_theta_sample = 0;
422     end
423
424     if VB_solved_once && (final_res_count > 0)
425         if ~res_readout_started
426             res_readout_started = true; res_readout_idx = 1;
427         end
428         limv = min(final_res_count, numel(RES_phase_vec_persist));
429         if res_readout_idx <= limv
430             res_phase_sample = RES_phase_vec_persist(res_readout_idx
                    );
431             res_duty_sample  = RES_y_vec_persist(res_readout_idx);
432             res_fit_sample   = RES_yhat_vec_persist(res_readout_idx)
                    ;
```

```
            res_readout_idx  = res_readout_idx + 1;
        else
            res_phase_sample = 0.0; res_duty_sample = 0.0;
               res_fit_sample = 0.0;
        end
    else
        res_phase_sample = 0.0; res_duty_sample = 0.0;
           res_fit_sample = 0.0;
    end

    %================= Consolidated outputs =================
    a = a_; b = b_; c = c_; g = R_; L_out = L_;
    ready_low  = double(ready_flag_low);
    ready_high = double(ready_flag_high);

    LOW_buf_count   = double(LOW_buf_idx);
    HIGH_buf_count  = double(HIGH_buf_idx);
    VBUS_buf_count  = double(VB_buf_idx);
    buffer_count    = double(max([LOW_buf_idx, HIGH_buf_idx,
       VB_buf_idx]));

    D_coeffs = VB_coeffs_persist;
    D_coeffs_buffer_ready = double(VB_buf_idx >= VB_fit_min_samples)
       ;
    D_fit_K_used   = V_FIT_K;
    D_k_gain       = VB_k_gain_persist;

    %================= Utilities =================
    function [L_, R_, c_kv_, a_, b_] = legacy_solver_band(V, I,
       omega, dt, ~, ~) %#ok<INUSD>
        % --- smoothing and derivatives ---
        I_s     = sgolay_smooth5(I);
        omega_s = sgolay_smooth5(omega);
```

```
461         dI      = sgolay_diff5(I_s, dt);
462         domega  = sgolay_diff5(omega_s, dt);
463
464         % ===== Electrical OLS =====
465         mask_e = isfinite(V) & isfinite(I_s) & isfinite(omega_s) &
                isfinite(dI) & (abs(I_s) > 1e-4);
466         Ve = V(mask_e)';  dIe = dI(mask_e)';  Ie = I_s(mask_e)';  We
                = omega_s(mask_e)';
467         Xe = [dIe, Ie, We];
468         theta_e = (Xe.'*Xe) \ (Xe.'*Ve);
469         L_      = theta_e(1);
470         R_      = theta_e(2);
471         c_kv_   = theta_e(3);
472
473         % ===== Mechanical OLS =====
474         mask_m = isfinite(domega) & isfinite(I_s) & isfinite(omega_s
                );
475         Im = I_s(mask_m)';  Wm = omega_s(mask_m)';  dWm = domega(
                mask_m)';
476         Xm = [Im, -Wm];   % [a, b] with b = -  /J
477         theta_m = (Xm.'*Xm) \ (Xm.'*dWm);
478         a_ = theta_m(1);
479         b_ = theta_m(2);
480     end
481
482     function xs = sgolay_smooth5(x)
483         N = numel(x); xs = x;
484         if N < 5, return; end
485         c5 = [-3 12 17 12 -3]/35;
486         for k=1:N
487             if k<=2, idx=1:5;
488             elseif k>=N-1, idx=N-4:N;
489             else, idx=k-2:k+2;
```

```
            end
            xs(k) = sum(c5 .* x(idx));
        end
    end

    function dydt = sgolay_diff5(x, dt)
        N = length(x); dydt = zeros(size(x));
        coeff = [1 -8 0 8 -1]/12;
        for k = 1:N
            if N >= 5
                if k <= 2, idx = 1:5;
                elseif k >= N-1, idx = N-4:N;
                else, idx = k-2:k+2;
                end
                xi = x(idx);
                dydt(k) = sum(coeff .* xi(:)') / dt;
            else
                i0 = max(1,k-1); i1 = min(N,k+1);
                c2 = [-1 0 1]/2;
                xi = zeros(1,3);
                ii = (i0:i1) - k + 2;
                xi(ii) = x(i0:i1);
                dydt(k) = sum(c2 .* xi) / dt;
            end
        end
    end

    function tu = unwrap_theta_row(t)
        tu = t(:)'; % row
        if isempty(tu), return; end
        for k = 2:numel(tu)
            d = tu(k) - tu(k-1);
            if d < -pi
```

```
                tu(k:end) = tu(k:end) + 2*pi;
            elseif d > pi
                tu(k:end) = tu(k:end) - 2*pi;
            end
        end
    end

    function zero_theta_overlay()
        THETA_vec_persist(:)   = 0;
        I_theta_vec_persist(:) = 0;
        V_theta_vec_persist(:) = 0;
        R_theta_vec_persist(:) = 0;

        theta_final_count      = 0;
        theta_readout_idx      = 1;
        theta_readout_started  = false;
        theta_solved_once      = false;
    end

    function v = pct1(x, p)
        x = sort(x(:)); n = numel(x);
        if n==0, v=0; return; end
        i = max(1, min(n, round((p/100)*n)));
        v = x(i);
    end
end
```

Listing 4.1: *Adaptive Plant Solver*

```
function [begin_flag, mech_flag, elec_flag, vbus_flag] =
   system_started3( ...
    start, is_min, clock, start_delay, ...
    ready_mech, ready_elec, ready_vbus, ...
    end_delay_mech, end_delay_elec, end_delay_vbus, ...
```

```
    dc_mode)
%#codegen
% begin_flag : Latches high after start, (next is_min edge if AC),
    then start_delay
% mech_flag  : Latches after ready_mech , (next is_min if AC),
    then end_delay_mech
% elec_flag  : Latches after ready_elec , (next is_min if AC),
    then end_delay_elec
% vbus_flag  : Latches after ready_vbus , (next is_min if AC),
    then end_delay_vbus
% dc_mode    : If true, ignores is_min edges for begin/end paths

% ---- Normalize boolean-like inputs to logical ----
start_b   = (start      ~= 0);
is_min_b  = (is_min     ~= 0);
dc_mode_b = (dc_mode    ~= 0);
ready_vec = [ (ready_mech ~= 0), (ready_elec ~= 0), (ready_vbus ~=
    0) ];

% ---- Persistent state ----
persistent begin_phase begin_time begin_latched is_min_prev
persistent end_phase end_time end_latched ready_prev
if isempty(begin_phase)
    begin_phase    = int32(0);
    begin_time     = 0;
    begin_latched  = false;
    is_min_prev    = false;       % logical

    end_phase      = int32([0 0 0]);
    end_time       = [0 0 0];
    end_latched    = false(1,3);  % logical(1x3)
    ready_prev     = false(1,3);  % logical(1x3)
end
```

```
33
34 % Per-channel delays (double)
35 end_delay = [end_delay_mech, end_delay_elec, end_delay_vbus];
36
37 % ================ Begin flag logic ================
38 if ~begin_latched
39     switch begin_phase
40         case 0  % wait for start
41             if start_b
42                 if dc_mode_b
43                     begin_time  = clock;
44                     begin_phase = int32(2);
45                 else
46                     begin_phase = int32(1);
47                 end
48             end
49
50         case 1  % wait for rising edge on is_min (AC mode)
51             if (~is_min_prev) && is_min_b
52                 begin_time  = clock;
53                 begin_phase = int32(2);
54             end
55
56         case 2  % wait for start_delay to elapse
57             if (clock - begin_time) >= start_delay
58                 begin_latched = true;
59                 begin_phase   = int32(3);
60             end
61
62         case 3
63             % latched
64     end
65 end
```

```
66
67 % ================ End flags logic (3 channels) ================
68 if begin_latched
69     for k = 1:3
70         if ~end_latched(k)
71             switch end_phase(k)
72                 case 0  % wait for ready(k) rising edge
73                     if ready_vec(k) && ~ready_prev(k)
74                         if dc_mode_b
75                             end_time(k)  = clock;
76                             end_phase(k) = int32(2);
77                         else
78                             end_phase(k) = int32(1);
79                         end
80                     end
81
82                 case 1  % wait for NEXT rising is_min (AC mode)
83                     if (~is_min_prev) && is_min_b
84                         end_time(k)  = clock;
85                         end_phase(k) = int32(2);
86                     end
87
88                 case 2  % wait for channel-specific end_delay
89                     if (clock - end_time(k)) >= end_delay(k)
90                         end_latched(k) = true;
91                         end_phase(k)   = int32(3);
92                     end
93
94                 case 3
95                     % latched
96             end
97         end
98     end
```

```
end

% ===== Update edge memories (types preserved as logical) =====
is_min_prev   = is_min_b;       % logical <- logical
ready_prev(:) = ready_vec;      % logical(1x3) <- logical(1x3)

% ===== Outputs =====
begin_flag = double(begin_latched);
mech_flag  = double(end_latched(1));  % mechanical coefficient
    fitting
elec_flag  = double(end_latched(2));  % electrical coefficient
    fitting
vbus_flag  = double(end_latched(3));  % voltage bus coefficient
    fitting
end
```

Listing 4.2: *Adaptive Solver Trigger*

```
function [V_out, d] = mpc_ripple_sync(V_applied, I_meas, omega_meas,
    omega_ref, ...
    trigger_control, artificial_trigger_control, lock, pll_phase,
        ...
    clock, V_min_step, N_tick_period)
% True receding-horizon MPC with phase-gated R sampling.
% - Replans every actuation tick (burst/hold grid, quantized steps &
    ramp limit)
% - d backsolver at fast rate
% - R backsolver uses decimated, phase-gated samples only
%
% Inputs:
%   V_applied, I_meas, omega_meas, omega_ref : plant & reference
    signals
%   trigger_control, artificial_trigger_control, lock : edge gating
%   pll_phase : current PLL phase [rad], wrap to [0, 2*pi) upstream
```

```
    if desired
%   clock : current time (s)
%   V_min_step : DAC/ladder min step
%   N_tick_period : control-edge period (s). If <= ~0, defaults to
    1/12000
%
% Outputs:
%   V_out : next commanded voltage (quantized & ramp-limited)
%   d     : best-fit constant disturbance on omega dot over recent
    fast window

%#codegen

% === Core MPC / Observer Parameters ===
horizon_len_ctrl = 10;        % integer >= 1
period_ticks     = 20;        % plan every N fast edges
default_tick_dt  = 0.0000833333333333; % 1/12000
min_improve      = 4e-3;
d_err_tol        = 0.001;
max_ramp         = 200;       % V/s
V_min_step_floor = 0.025;     % min hardware step used by controller

if N_tick_period > 1e-6
    tick_dt = N_tick_period;
else
    tick_dt = default_tick_dt;
end
tick_actuation = period_ticks * tick_dt;
mini_dt        = tick_actuation;

buffer_len = 2 * period_ticks;   % for d observer (fast buffer)
V_max     = 84;
V_min     = 0;
```

```
43
44 % AC Parameters (your labels: R=c, kv=g)
45 a = 3.72;  b = 0.1002;  g = 0.186;  L = 0.0001006;
46 c = 0.01998;            % planner's constant R (ohms)
47 V_scale = 1;            % used by controller ONLY (NOT by R
       backsolve)
48
49 lambda_final = 0.7;     % omega error cost
50
51 % === Quantization & ramp (action grid) ===
52 V_step          = max(V_min_step, V_min_step_floor);
53 V_ramp_per_tick = max_ramp * tick_actuation;
54 n_bins          = max(1, floor(V_ramp_per_tick / V_step));
55 action_grid     = -n_bins:n_bins;
56
57 % === R-Observer cadence (phase-gated) ===
58 R_period_ticks = 400;   % arm once every 400 control edges
59 R_buffer_len   = 20;    % decimated samples stored for R
60
61 % ---- Phase-gated R sample config (internal) ----
62 PHASE_TARGET      = 0.0;   % radians; set your grab point here (e.g
       ., 0, pi/2, pi)
63 PHASE_BAND_MIN    = 0.035; % ~2.0 degrees minimum band to avoid
       misses at low dphi
64 PHASE_BAND_MARGIN = 0.005; % cushion added to 0.5*dphi_tick
65
66 % === Persistent State ===
67 persistent V_out_held initialized
68 persistent V_hist I_hist omega_hist last_trigger
       last_artificial_trigger
69 persistent omega_buf I_buf V_fast_buf domega_buf t_buf tick_count
70 persistent action_tick_count
71 persistent first_step_hold first_step_count
```

```
72  persistent buffer_filled
73  persistent omega_pred_buf
74  persistent omega_pred_err_buf
75  persistent last_d last_err_best_d
76
77  % Dedicated R buffers (phase-gated decimation)
78  persistent R_I_prev_buf R_I_now_buf R_dI_buf R_V_buf R_omega_buf
        R_t_buf R_buffer_filled
79  persistent last_R last_err_best_R
80
81  % Phase sampling helpers
82  persistent last_phase phase_waiting
83
84  if isempty(V_out_held)
85      V_out_held = 0;
86      initialized = false;
87
88      V_hist     = V_applied * ones(1,3);
89      I_hist     = I_meas    * ones(1,3);
90      omega_hist = omega_meas* ones(1,3);
91      last_trigger = 0;
92      last_artificial_trigger = 0;
93
94      omega_buf   = omega_meas*ones(1,buffer_len);
95      I_buf       = I_meas   *ones(1,buffer_len);
96      V_fast_buf  = V_applied*ones(1,buffer_len);
97      domega_buf  = zeros(1,buffer_len);
98      t_buf       = zeros(1,buffer_len);
99
100     tick_count         = 0;
101     action_tick_count  = 0;
102     first_step_hold    = false;
103     first_step_count   = 0;
```

```
104     buffer_filled      = 0;
105     omega_pred_buf     = omega_meas*ones(1,buffer_len);
106     omega_pred_err_buf = zeros(1,buffer_len);
107 
108     last_d         = 0;
109     last_err_best_d= Inf;
110 
111     R_I_prev_buf   = I_meas     * ones(1, R_buffer_len);
112     R_I_now_buf    = I_meas     * ones(1, R_buffer_len);
113     R_dI_buf       = zeros(1, R_buffer_len);
114     R_V_buf        = V_applied  * ones(1, R_buffer_len);
115     R_omega_buf    = omega_meas * ones(1, R_buffer_len);
116     R_t_buf        = zeros(1, R_buffer_len);
117     R_buffer_filled = 0;
118 
119     last_R         = c;   % start at planner's constant R
120     last_err_best_R= Inf;
121 
122     last_phase     = NaN;
123     phase_waiting  = false;
124 end
125 
126 d = last_d;
127 
128 % === 3-point moving average on inputs ===
129 V_hist         = [V_hist(2:3), V_applied];
130 I_hist         = [I_hist(2:3), I_meas];
131 omega_hist     = [omega_hist(2:3), omega_meas];
132 V_applied_filt = (V_hist(1)+V_hist(2)+V_hist(3))/3;
133 I_meas_filt    = (I_hist(1)+I_hist(2)+I_hist(3))/3;
134 omega_meas_filt= (omega_hist(1)+omega_hist(2)+omega_hist(3))/3;
135 
136 % === Trigger edge detect ===
```

```
137 if lock == 0
138     control_edge = (artificial_trigger_control ~= 0) && (
        last_artificial_trigger == 0);
139 else
140     control_edge = (trigger_control ~= 0) && (last_trigger == 0);
141 end
142
143 if control_edge
144     tick_count = tick_count + 1;
145
146     % --- fast buffers (update first) ---
147     if buffer_filled < buffer_len
148         buffer_filled = buffer_filled + 1;
149     end
150     omega_buf  = [omega_buf(2:end), omega_meas];
151     I_buf      = [I_buf(2:end), I_meas];
152     V_fast_buf = [V_fast_buf(2:end), V_applied];
153     t_buf      = [t_buf(2:end), clock];
154
155     if buffer_filled > 1
156         domega = (omega_buf(end) - omega_buf(end-1)) / tick_dt;
157     else
158         domega = 0;
159     end
160     domega_buf = [domega_buf(2:end), domega];
161
162     % ---------- ARM phase-gated R sampler every R_period_ticks
        ----------
163     if mod(tick_count, R_period_ticks) == 0
164         phase_waiting = true;
165     end
166
167     % ---------- R PHASE-GATED SAMPLE ----------
```

```
168     if ~isnan(pll_phase)
169         prev_phase = last_phase;
170         if isnan(prev_phase)
171             prev_phase = pll_phase;
172         end
173 
174         if phase_waiting && (buffer_filled >= 2)
175             dphi_tick  = abs(wrapToPi_local(pll_phase - prev_phase))
                ;
176             phase_band = max(PHASE_BAND_MIN, 0.5*dphi_tick +
                PHASE_BAND_MARGIN);
177 
178             phi_prev_rel = wrapToPi_local(prev_phase  - PHASE_TARGET
                );
179             phi_now_rel  = wrapToPi_local(pll_phase   - PHASE_TARGET
                );
180 
181             crossed = (phi_prev_rel < 0) && (phi_now_rel >= 0); %
                rising through target
182             in_band = abs(phi_now_rel) <= phase_band;
183 
184             if crossed && in_band
185                 I_prev_fast    = I_buf(end-1);
186                 I_now_fast     = I_buf(end);
187                 dI_fast        = (I_now_fast - I_prev_fast)/tick_dt;
188                 V_now_fast     = V_fast_buf(end);
189                 omega_now_fast = omega_buf(end);
190                 t_now_fast     = t_buf(end);
191 
192                 if R_buffer_filled < R_buffer_len
193                     R_buffer_filled = R_buffer_filled + 1;
194                 end
195                 R_I_prev_buf  = [R_I_prev_buf(2:end),  I_prev_fast];
```

```
196                 R_I_now_buf    = [R_I_now_buf(2:end),    I_now_fast];
197                 R_dI_buf       = [R_dI_buf(2:end),       dI_fast];
198                 R_V_buf        = [R_V_buf(2:end),        V_now_fast];
199                 R_omega_buf    = [R_omega_buf(2:end),
                        omega_now_fast];
200                 R_t_buf        = [R_t_buf(2:end),        t_now_fast];
201
202                 phase_waiting = false; % disarm until next arm
203             end
204         end
205
206         % update last_phase each control edge
207         last_phase = pll_phase;
208     end
209     % -------- END R PHASE-GATED SAMPLE --------
210
211     % === Build omega prediction error (for d) ===
212     for k = 1:buffer_filled
213         buf_idx = buffer_len - buffer_filled + k;
214         if k == 1
215             omega_pred_buf(buf_idx)     = omega_buf(buf_idx);
216             omega_pred_err_buf(buf_idx) = NaN;
217         else
218             idx_prev    = buf_idx - 1;
219             omega_prev  = omega_buf(idx_prev);
220             I_now_fast  = I_buf(buf_idx);
221             omega_pred  = omega_prev + (a*I_now_fast - b*omega_prev)
                    *tick_dt;
222             omega_pred_buf(buf_idx)     = omega_pred;
223             omega_pred_err_buf(buf_idx) = omega_pred - omega_buf(
                    buf_idx);
224         end
225     end
```

```
226
227     if ~initialized
228         initialized        = true;
229         V_out_held         = V_step;
230         first_step_hold    = true;
231         first_step_count   = 1;
232         action_tick_count  = 0;
233         V_out = V_out_held;
234         last_trigger = trigger_control;
235         last_artificial_trigger = artificial_trigger_control;
236         return;
237     end
238
239     if first_step_hold
240         V_out = V_out_held;
241         first_step_count = first_step_count + 1;
242         if first_step_count >= 4
243             first_step_hold = false;
244         end
245         last_trigger = trigger_control;
246         last_artificial_trigger = artificial_trigger_control;
247         return;
248     end
249
250     % === Only plan on actuation ticks ===
251     if action_tick_count == 0
252
253         % ---- Disturbance backsolver (d) ----
254         best_d = 0; best_err = Inf; err_zero = 0;
255         if buffer_filled == buffer_len
256             for k = 2:buffer_filled
257                 buf_idx    = buffer_len - buffer_filled + k;
258                 omega_prev = omega_buf(buf_idx-1);
```

```
259                 I_now_fast = I_buf(buf_idx);
260                 omega_pred0= omega_prev + (a*I_now_fast - b*
                        omega_prev)*tick_dt;
261                 err_zero   = err_zero + abs(omega_pred0 - omega_buf(
                        buf_idx));
262             end
263             for d_test = -10000:1:10000
264                 abs_err_sum = 0;
265                 for k = 2:buffer_filled
266                     buf_idx    = buffer_len - buffer_filled + k;
267                     omega_prev = omega_buf(buf_idx-1);
268                     I_now_fast = I_buf(buf_idx);
269                     omega_pred = omega_prev + (a*I_now_fast - b*
                            omega_prev + d_test)*tick_dt;
270                     abs_err_sum = abs_err_sum + abs(omega_pred -
                            omega_buf(buf_idx));
271                 end
272                 if abs_err_sum < best_err
273                     best_err = abs_err_sum;
274                     best_d   = d_test;
275                 end
276             end
277             improve = err_zero - best_err;
278
279             if improve >= min_improve
280                 if abs(best_err - last_err_best_d) < d_err_tol
281                     d = last_d;
282                 else
283                     last_d = best_d; d = last_d; last_err_best_d =
                            best_err;
284                 end
285             else
286                 d = 0; last_d = 0; last_err_best_d = Inf;
```

```
287             end
288         end
289
290         % ---- R Backsolve update (if buffer filled) ----
291         if R_buffer_filled >= 1
292             R_model  = c;
293
294             if R_buffer_filled == R_buffer_len
295                 err_R_at_model = 0; best_R_err = Inf; best_R =
                        last_R;
296                 for kR = 1:R_buffer_len
297                     Inow  = R_I_now_buf(kR);
298                     dIf   = R_dI_buf(kR);
299                     Vnow  = R_V_buf(kR);
300                     omg   = R_omega_buf(kR);
301                     V_res = Vnow - (L*dIf + R_model*Inow + g*omg);
302                     err_R_at_model = err_R_at_model + abs(V_res);
303                 end
304                 for R_test = 0.0000:0.0005:0.2000
305                     abs_err_sum_R = 0;
306                     for kR = 1:R_buffer_len
307                         Inow  = R_I_now_buf(kR);
308                         dIf   = R_dI_buf(kR);
309                         Vnow  = R_V_buf(kR);
310                         omg   = R_omega_buf(kR);
311                         V_res = Vnow - (L*dIf + R_test*Inow + g*omg)
                                ;
312                         abs_err_sum_R = abs_err_sum_R + abs(V_res);
313                     end
314                     if abs_err_sum_R < best_R_err
315                         best_R_err = abs_err_sum_R; best_R = R_test;
316                     end
317                 end
```

```
318                 improve_R = err_R_at_model - best_R_err;
319                 if improve_R >= min_improve
320                     last_R = best_R;
321                     last_err_best_R = best_R_err;
322                 else
323                     last_R = c;
324                     last_err_best_R = Inf;
325                 end
326             end
327         end
328
329         % ---- Burst/Hold Path Planning ----
330         n_plans = numel(action_grid) * horizon_len_ctrl;
331         plans   = zeros(n_plans, horizon_len_ctrl);
332         row     = 1;
333         for idx_a = 1:numel(action_grid)
334             a_ = action_grid(idx_a);
335             for drop = 0:(horizon_len_ctrl-1)
336                 plan = zeros(1, horizon_len_ctrl);
337                 for kk = 1:(horizon_len_ctrl-drop)
338                     plan(kk) = a_;
339                 end
340                 plans(row, :) = plan;
341                 row = row + 1;
342             end
343         end
344
345         best_cost       = Inf;
346         best_path_local = zeros(1, horizon_len_ctrl);
347
348         I_sim0     = I_meas_filt;
349         omega_sim0 = omega_meas_filt;
350
```

```
351         for i = 1:n_plans
352             actions = plans(i,:);
353             V_path  = zeros(1, horizon_len_ctrl);
354             for k = 1:horizon_len_ctrl
355                 V_path(k) = V_out_held + sum(actions(1:k))*V_step;
356                 if V_path(k) < V_min, V_path(k) = V_min; end
357                 if V_path(k) > V_max, V_path(k) = V_max; end
358             end
359
360             I_tmp  = I_sim0;
361             om_tmp = omega_sim0;
362             cum_err = 0.0;
363             for k = 1:horizon_len_ctrl
364                 V_sim  = V_path(k);
365                 dI_sim = (V_scale * V_sim - c*I_tmp - g*om_tmp) / L;
                        % planner uses constant c (R=c)
366                 I_tmp  = I_tmp + dI_sim * mini_dt;
367                 dome   =  a*I_tmp - b*om_tmp + d;
368                 om_tmp = om_tmp + dome * mini_dt;
369                 cum_err = cum_err + abs(om_tmp - omega_ref);
370             end
371
372             cost = lambda_final * cum_err;
373
374             if cost < best_cost
375                 best_cost       = cost;
376                 best_path_local = V_path;
377             end
378         end
379
380         V_out_held = best_path_local(1);
381         if V_out_held < V_min, V_out_held = V_min; end
382         if V_out_held > V_max, V_out_held = V_max; end
```

```
    end

    V_out = V_out_held;

    action_tick_count = action_tick_count + 1;
    if action_tick_count >= period_ticks
        action_tick_count = 0;
    end

else
    V_out = V_out_held;
end

% Latch triggers
last_trigger = trigger_control;
last_artificial_trigger = artificial_trigger_control;

end % ===== end main function =====


function y = wrapToPi_local(x)
% Wrap angle in radians to [-pi, pi)
y = mod(x + pi, 2*pi) - pi;
end
```

Listing 4.3: *MPC with Backsolver*

```
function [is_min, min_time, min_value, freq_hz, ...
          delayed_tick, delayed_tick_time, ...
          delayed_tick_N, delayed_tick_N_time, freq_N_tick_hz, ...
          locked, controller_on, N_tick_period, ...
          is_min_locked_now, phase] = ...
    find_center_is_min(V_bus, clk_in, delay_frac, start_frac)
%#codegen
```

```
% PLL on rectified bus ripple (  120   Hz), with:
%  - clean min detection (center-aligned window)
%  - lock / relock with atomic handoff to new cadence
%  - fast N-ticks (no DS quantization)
%  - phase   : 120 Hz phase (0..2  ) anchored at locked minima

% ---------------- Debug switch ----------------
DEBUG = false;
% ------------------------------------------------

% ---------------- Config ----------------
DS_SKIP         = 40;
WIN_SIZE        = 100;
MID             = WIN_SIZE / 2;
N_MINIMA        = 8;
FREQ_TICK_MIN   = 119;
FREQ_TICK_MAX   = 121;
PERIOD_MIN      = 1 / FREQ_TICK_MAX;
PERIOD_MAX      = 1 / FREQ_TICK_MIN;

REFRAC          = 0.25 * (1/FREQ_TICK_MAX);  % ~2 ms refractory for
   is_min
EPS_T           = 1e-9;
SAFE_FRAC       = max(delay_frac, eps);

% ------------- Persistent state -------------
persistent buf buf_time buf_full buf_ptr ds_counter sample_number
persistent locked_state locked_min_time locked_period locked_freq
persistent minima_times minima_count started
persistent period_next_time N_next_time
persistent freq_N_tick_hz_persist
persistent researching relock_times relock_count
persistent controller_on_latched
```

```
persistent last_min_time_gridlocked

persistent n_tick_active last_delayed_tick_N_time
persistent simultaneous_first_tick_pending first_tick_time
persistent N_tick_period_current
persistent handoff_pending old_N_tick_period
persistent pending_relock relock_locked_period relock_locked_freq

% Atomic handoff support
persistent new_N_tick_period_next
persistent N_tick_period_report
persistent freeze_period_report_until_handoff

% is_min handling (aligned-at-center version)
persistent raw_prev_v raw_prev_diff_v
persistent min_arm min_arm_time
persistent last_is_min_now_time

% Quiescence before handoff edge
persistent quiescent_until_edge

% ------------- One-time init -------------
if isempty(buf)
    buf = nan(1, WIN_SIZE);
    buf_time = zeros(1, WIN_SIZE);
    buf_ptr = 0;
    buf_full = false;
    ds_counter = 0;
    sample_number = 0;

    locked_state = false;
    locked_period = 0;
    locked_freq = 0;
```

```
    locked_min_time = 0;

    minima_times = nan(1, N_MINIMA);
    minima_count = 0;
    started = false;

    period_next_time = 0;
    N_next_time = 0;
    freq_N_tick_hz_persist = 0;

    researching = false;
    relock_times = nan(1, N_MINIMA);
    relock_count = 0;

    controller_on_latched = false;
    last_min_time_gridlocked = 0;

    n_tick_active = false;
    last_delayed_tick_N_time = 0;
    simultaneous_first_tick_pending = false;
    first_tick_time = 0;
    N_tick_period_current = 0;
    handoff_pending = false;
    old_N_tick_period = 0;
    pending_relock = false;
    relock_locked_period = 0;
    relock_locked_freq = 0;

    new_N_tick_period_next = 0;
    N_tick_period_report = 0;
    freeze_period_report_until_handoff = false;

    raw_prev_v = V_bus;
```

```
106     raw_prev_diff_v = 0;
107     min_arm = false;
108     min_arm_time = -Inf;
109     last_is_min_now_time = -Inf;
110 
111     quiescent_until_edge = false;
112 end
113 
114 % ------------- Defaults -------------
115 is_min = 0;
116 min_time = 0;
117 min_value = 0;
118 freq_hz = locked_freq;
119 delayed_tick = 0;
120 delayed_tick_time = 0;
121 delayed_tick_N = 0;
122 delayed_tick_N_time = 0;
123 freq_N_tick_hz = 0;
124 locked = double(locked_state);
125 controller_on = double(controller_on_latched);
126 N_tick_period = N_tick_period_report;
127 is_min_locked_now = 0;
128 
129 % Phase (0..2  ) anchored to locked min
130 if locked_state && (locked_period > 0)
131     theta = 2*pi * ((clk_in - locked_min_time) / locked_period);
132     phase = theta - 2*pi * floor(theta / (2*pi));
133 else
134     phase = 0;
135 end
136 
137 % =========================
138 % PRE-DS: slope flip -> arm
```

```
% =========================
sample_time = clk_in;
raw_diff_v  = V_bus - raw_prev_v;
if locked_state && (raw_prev_diff_v < 0) && (raw_diff_v >= 0)
    min_arm = true;
    min_arm_time = sample_time;
end
raw_prev_diff_v = raw_diff_v;
raw_prev_v      = V_bus;

% ============================================================
% FAST N-TICK (pre-DS): run every call to avoid DS quantization
% ============================================================
if n_tick_active && ~handoff_pending && (N_tick_period_current > 0)
    if N_next_time <= 0
        N_next_time = sample_time + N_tick_period_current;
    end
    if sample_time >= N_next_time
        delayed_tick_N = 1;
        delayed_tick_N_time = N_next_time;

        if last_delayed_tick_N_time > 0
            freq_N_tick_hz = 1 / (delayed_tick_N_time -
               last_delayed_tick_N_time);
        else
            freq_N_tick_hz = 0;
        end
        last_delayed_tick_N_time = delayed_tick_N_time;

        N_next_time = N_next_time + N_tick_period_current;

        if ~freeze_period_report_until_handoff
            N_tick_period_report = N_tick_period_current;
```

```
171         end
172         N_tick_period = N_tick_period_report;
173     end
174 end
175 
176 % ================================================================
177 % Downsample gate     only guards the heavy min detection work
178 % ================================================================
179 ds_counter = ds_counter + 1;
180 do_ds = false;
181 if ds_counter >= DS_SKIP
182     ds_counter = 0;
183     sample_number = sample_number + 1;
184     do_ds = true;
185 end
186 
187 % =========================
188 % Handoff edge commit block
189 % =========================
190 if simultaneous_first_tick_pending && locked_state && sample_time >=
        first_tick_time
191     % Period tick
192     delayed_tick = 1;
193     delayed_tick_time = first_tick_time;
194     period_next_time = first_tick_time + locked_period;
195 
196     % Single N-tick at the edge
197     delayed_tick_N = 1;
198     delayed_tick_N_time = first_tick_time;
199     if last_delayed_tick_N_time > 0
200         freq_N_tick_hz = 1 / (delayed_tick_N_time -
              last_delayed_tick_N_time);
201     else
```

```
202         freq_N_tick_hz = 0;
203     end
204     last_delayed_tick_N_time = delayed_tick_N_time;
205
206     % Commit new cadence
207     if new_N_tick_period_next > 0
208         N_tick_period_current = new_N_tick_period_next;
209         new_N_tick_period_next = 0;
210     end
211     N_next_time = first_tick_time + N_tick_period_current;
212
213     N_tick_period_report = N_tick_period_current;
214     freeze_period_report_until_handoff = false;
215
216     simultaneous_first_tick_pending = false;
217     handoff_pending = false;
218     quiescent_until_edge = false;
219     N_tick_period = N_tick_period_report;
220     if DEBUG
221         fprintf('[PLL HANDOFF EDGE] Handoff tick at first_tick_time:
                %.5f\n', first_tick_time);
222     end
223 end
224
225 % ========================================
226 % Old-cadence N-ticks while handoff is pending
227 % ========================================
228 if handoff_pending
229     if ~quiescent_until_edge
230         if (N_next_time >= first_tick_time - EPS_T)
231             quiescent_until_edge = true;
232             N_next_time = first_tick_time;  % align; edge block will
                emit
```

```
233             if DEBUG
234                 fprintf('[PLL N-TICK-OLD] quiescent until handoff (
                        next old tick %.5f >= edge %.5f)\n', ...
235                     N_next_time, first_tick_time);
236             end
237         elseif sample_time >= N_next_time
238             delayed_tick_N = 1;
239             delayed_tick_N_time = N_next_time;
240             if DEBUG
241                 fprintf('[PLL N-TICK-OLD] N-tick at: %.5f (handoff
                        pending)\n', N_next_time);
242             end
243
244             if last_delayed_tick_N_time > 0
245                 freq_N_tick_hz = 1 / (delayed_tick_N_time -
                        last_delayed_tick_N_time);
246             else
247                 freq_N_tick_hz = 0;
248             end
249             last_delayed_tick_N_time = delayed_tick_N_time;
250
251             N_next_time = N_next_time + old_N_tick_period;
252             N_tick_period = N_tick_period_report;   % still OLD
253         end
254     end
255     % Allow DS work below (to finalize handoff timings)
256 end
257
258 % ========================
259 % Regular 120 Hz delayed tick
260 % ========================
261 if locked_state && ~handoff_pending && (period_next_time > 0) &&
    sample_time >= period_next_time
```

```
262     delayed_tick = 1;
263     delayed_tick_time = period_next_time;
264     period_next_time = period_next_time + locked_period;
265 end
266 
267 % =========================
268 % Heavy work (min detection) on DS frames only
269 % =========================
270 if do_ds
271     % Buffer update (store V_bus)
272     buf_ptr = buf_ptr + 1;
273     if buf_ptr > WIN_SIZE
274         buf_ptr = 1;
275         buf_full = true;
276     end
277     buf(buf_ptr) = V_bus;
278     buf_time(buf_ptr) = sample_time;
279 
280     if buf_full
281         if ~started && buf(end) > 0
282             % proceed
283         elseif ~started
284             % wait for first positive window
285         end
286 
287         center = mod(buf_ptr - MID - 1, WIN_SIZE) + 1;
288         [~, min_idx] = min(buf);
289 
290         if min_idx == center
291             started   = true;
292             is_min    = 1;
293             min_time  = buf_time(center);
294             min_value = buf(center);
```

```
295
296             % Emit aligned instantaneous tick exactly at min_time
297             if locked_state && min_arm && (min_time -
                    last_is_min_now_time > REFRAC)
298                 is_min_locked_now   = 1;
299                 last_is_min_now_time= min_time;
300             end
301             min_arm = false;
302
303             % ---------- Initial Lock ----------
304             if ~locked_state && ~researching && ~pending_relock
305                 minima_count = minima_count + 1;
306                 idx = mod(minima_count-1, N_MINIMA) + 1;
307                 minima_times(idx) = min_time;
308
309                 n_valid_pairs = 0;
310                 valid_periods = nan(1, N_MINIMA*(N_MINIMA-1)/2);
311                 match_idx = 1;
312                 for j = 1:N_MINIMA
313                     for k = j+1:N_MINIMA
314                         if isnan(minima_times(j)) || isnan(
                                minima_times(k)), continue; end
315                         dtp = abs(minima_times(j) - minima_times(k))
                                ;
316                         if dtp >= PERIOD_MIN && dtp <= PERIOD_MAX
317                             valid_periods(match_idx) = dtp;
318                             match_idx = match_idx + 1;
319                             n_valid_pairs = n_valid_pairs + 1;
320                         end
321                     end
322                 end
323                 if n_valid_pairs >= 2
324                     locked_period  = mean(valid_periods(~isnan(
```

```
                    valid_periods)));
325                 locked_freq    = 1/locked_period;
326                 locked_min_time= min_time;
327                 locked_state   = true;
328                 freq_hz        = locked_freq;
329
330                 % Set cadence from delay_frac
331                 N_tick_period_current = SAFE_FRAC *
                        locked_period;
332                 N_tick_period_report  = N_tick_period_current;
333                 freq_N_tick_hz_persist= locked_freq / SAFE_FRAC;
334                 freeze_period_report_until_handoff = false;
335
336                 % Synchronized schedule
337                 first_tick_time = locked_min_time + start_frac *
                        locked_period;
338                 period_next_time= first_tick_time;
339                 N_next_time     = first_tick_time;
340                 n_tick_active   = true;
341                 simultaneous_first_tick_pending = true;
342
343                 researching         = false;
344                 relock_times(:)     = nan;
345                 relock_count        = 0;
346
347                 controller_on_latched = true;
348                 controller_on = double(controller_on_latched);
349
350                 last_min_time_gridlocked = min_time;
351                 locked   = double(locked_state);
352                 handoff_pending = false;
353                 old_N_tick_period = 0;
354                 pending_relock = false;
```

```
355                     relock_locked_period = 0;
356                     relock_locked_freq   = 0;
357
358                     quiescent_until_edge = false;
359                 end
360
361             % ---------- Check for Unlock ----------
362             elseif locked_state && ~researching && ~pending_relock
363                 if last_min_time_gridlocked > 0
364                     observed_period = min_time -
                        last_min_time_gridlocked;
365                     observed_freq   = 1 / observed_period;
366                     if observed_freq < FREQ_TICK_MIN ||
                        observed_freq > FREQ_TICK_MAX
367                         researching = true;
368                         relock_times(:) = nan;
369                         relock_count = 0;
370
371                         old_N_tick_period = N_tick_period_current;
372
373                         handoff_pending = false;
374                         locked_state = false;
375                         quiescent_until_edge = false;
376                     end
377                 end
378                 last_min_time_gridlocked = min_time;
379             end
380
381             % ---------- Collect for Relock ----------
382             if researching && ~pending_relock
383                 relock_count = relock_count + 1;
384                 idx2 = mod(relock_count-1, N_MINIMA) + 1;
385                 relock_times(idx2) = min_time;
```

```
386
387                 n_valid_pairs2 = 0;
388                 valid_periods2 = nan(1, N_MINIMA*(N_MINIMA-1)/2);
389                 match_idx2 = 1;
390                 for j = 1:N_MINIMA
391                     for k = j+1:N_MINIMA
392                         if isnan(relock_times(j)) || isnan(
                               relock_times(k)), continue; end
393                         dtp2 = abs(relock_times(j) - relock_times(k)
                               );
394                         if dtp2 >= PERIOD_MIN && dtp2 <= PERIOD_MAX
395                             valid_periods2(match_idx2) = dtp2;
396                             match_idx2 = match_idx2 + 1;
397                             n_valid_pairs2 = n_valid_pairs2 + 1;
398                         end
399                     end
400                 end
401                 if n_valid_pairs2 >= 3
402                     relock_locked_period = mean(valid_periods2(~
                           isnan(valid_periods2)));
403                     relock_locked_freq   = 1/relock_locked_period;
404                     pending_relock       = true;
405                     researching          = false;
406                     relock_times(:)      = nan;
407                     relock_count         = 0;
408                     last_min_time_gridlocked = min_time;
409                     if DEBUG
410                         fprintf('[PLL HANDOFF] RELock scheduled. New
                               period=%.5f\n', relock_locked_period);
411                     end
412                 end
413             end
414
```

```
415             % ---------- Relock min: begin handoff (stage only)
                    ----------
416             if pending_relock
417                 locked_period  = relock_locked_period;
418                 locked_freq    = relock_locked_freq;
419                 locked_min_time= min_time;
420                 locked_state   = true;
421
422                 new_N_tick_period_next = SAFE_FRAC * locked_period;
423                 first_tick_time = locked_min_time + start_frac *
                        locked_period;
424
425                 n_tick_active = true;
426                 if N_next_time <= 0
427                     N_next_time = sample_time + old_N_tick_period;
428                 end
429                 period_next_time = first_tick_time;
430                 simultaneous_first_tick_pending = true;
431
432                 controller_on_latched = true;
433                 controller_on = double(controller_on_latched);
434
435                 last_min_time_gridlocked = min_time;
436                 locked = double(locked_state);
437
438                 N_tick_period_report = old_N_tick_period;
439                 freeze_period_report_until_handoff = true;
440
441                 handoff_pending = true;
442                 pending_relock  = false;
443
444                 quiescent_until_edge = false;
445
```

```
                if DEBUG
                    fprintf('[PLL HANDOFF EDGE] Relock at min %.5f,
                        handoff at %.5f\n', ...
                        locked_min_time, first_tick_time);
                end
            end
        end
    end
end

% ---------- Default reporting ----------
if freeze_period_report_until_handoff
    N_tick_period = N_tick_period_report;   % old
else
    N_tick_period_report = N_tick_period_current;
    N_tick_period = N_tick_period_report;
end

locked = double(locked_state);
controller_on = double(controller_on_latched);
freq_hz = locked_freq;

end
```

Listing 4.4: *Phase Lock Loop*

```
function V_out = V_des_ramp_filter(V_des_raw, dt)
%#codegen
% Simple linear voltage ramp limiter.
%  - Always ramps toward V_des_raw at |dV/dt| <= max_slope

    % ===== Tunables =====
    max_slope   = 200;     % V/s, slew limit
    eps_zero    = 1e-6;    % zero detect tolerance
```

```
    snap_eps    = 1e-4;    % when |delta| < snap_eps, snap to target

    % ===== Persistent state =====
    persistent V_out_prev
    persistent passthrough_enabled

    if isempty(V_out_prev)
        % Start from the first commanded value (avoids a fake ramp
            at t=0)
        V_out_prev = V_des_raw;
    end
    if isempty(passthrough_enabled)
        passthrough_enabled = false; % set true via a separate
            setter if desired
    end

    % ===== Optional passthrough =====
    if passthrough_enabled
        V_out = V_des_raw;
        V_out_prev = V_out;
        return;
    end

    % ===== Slew limiting =====
    if dt <= 0
        % Defensive: no time elapsed => hold
        V_out = V_out_prev;
        return;
    end

    % Treat near-zero as zero (makes start/stop clean and avoids
        jitter)
    if abs(V_des_raw) < eps_zero
```

```
        V_des = 0.0;
    else
        V_des = V_des_raw;
    end

    delta    = V_des - V_out_prev;
    max_step = max_slope * dt;

    if abs(delta) <= snap_eps
        V_out = V_des;                    % snap when very close
    else
        step  = max(min(delta, max_step), -max_step);
        V_out = V_out_prev + step;        % ramp toward target
    end

    V_out_prev = V_out;
end
```

Listing 4.5: *Ramp Filter*

```
function [pwm, duty_actual, V_min_step, phi_raw_deg_out, ...
          err_mean_prev_rot, err_dev_raw, err_dev_used,
             err_dev_signage] = pwm_driver_vff_select( ...
    V_cmd, phase, dt, Vbus_inst, V_load_meas, I_meas, omega_meas,
       ...
    trigger_control, artificial_trigger_control, N_tick_period, lock
       , clock)
%#codegen
%
% Modes:
% 0 -> FLAT (use Vbus ceiling)
% 1 -> DIRECT (measured Vbus)
% 2 -> FITTED +   _static
% 4 -> FITTED +   _adapt    (NUDGE      only;    =         dt; hold-
```

```
    last when gate closes)
%
% Key idea for MODE 4:
%   - Keep a commanded      (deg/s): phi_rate_cmd_degps
%   - On tick edges: phi_rate_cmd_degps += nudge(error, d/dt error)
%   - Between ticks:    <-    + phi_rate_cmd_degps * dt  (no direct
    error->  )

% ===== User knobs =====
MODE                 = 4;     % set 0/1/2/4
PHI_STATIC_DEG       = 90;
PHI_START_DEG        = 50;

% Engage + deadzone
VLOAD_PI_MIN_V       = 6.0;   % controller engages only when
    Vload_est >= this
ERR_PI_GATE_MIN_V    = 0.6;   % deadzone (|err| below => 0)
% ERR_PI_GATE_MIN_V    = 0.4;   % deadzone (|err| below => 0)

% Baseline      if no history
PHI_RATE_BASE_DEFAULT_DEGPS = 20.0;

% === Gains to NUDGE      (deg/s contributions) ===
% P applies ONLY when error slope >= 0 (not improving)
KRATE_P_degps_per_V        = 40;    % P
KRATE_D_degps_per_Vps      = 1;   % unified D gain: +derr_dt ->
    speed up     , -derr_dt -> slow

SENSE_DIR = 1;                       % flip if polarity inverted

% ===== PWM / duty quantization =====
f_pwm         = 5000;
headroom      = 0.01;
```

```
vmin_sane     = 5.0;
enable_dither = true;
f_export      = 120;

% ===== Bus-shape coefficients (fitted) =====
BUS_COEFFS = [ ...
    74.4137, ...
    -5.4340,  3.9080, ...
    -2.4635, -0.9743, ...
    -0.3123, -1.1455, ...
     0.2623, -0.4518, ...
     0.2323, -0.1082, ...
     0.1527,  0.0415, ...
     0.0352,  0.0872, ...
    -0.0254,  0.0412 ...
];

% ===== Derived =====
if nargin < 5  || isempty(V_load_meas), V_load_meas = []; end
if nargin < 6  || isempty(I_meas),      I_meas      = 0;  end %#ok<
    NASGU>
if nargin < 7  || isempty(omega_meas),  omega_meas  = 0;  end %#ok<
    NASGU>
if nargin < 12 || isempty(clock),       clock       = 0;  end

PI_OVER_180 = pi/180;
dt_eff      = max(dt, 1.0e-6);

Ts_pwm    = 1/f_pwm;
n_steps   = max(1, round(Ts_pwm / dt_eff));
duty_step = 1 / n_steps;

if enable_dither
```

```
72      n_cycles_per_update = max(1, round(f_pwm / f_export));
73      dither_factor       = max(1, floor(n_cycles_per_update/2));
74      effective_step      = duty_step / dither_factor;
75  else
76      effective_step      = duty_step;
77  end
78
79  % ===== Persistent =====
80  persistent t_in_period high_count sample_count duty_accum
        duty_current acc
81  persistent phi_raw_rad
82  persistent last_phase_wrapped theta_unwrapped rot_start_theta
83  persistent last_trigger last_artificial_trigger last_tick_time
84  persistent Vbus_max V_MIN_STEP_CONST last_effective_step
85  persistent err_buf cur_len mean_prev_rot
86
87  % MODE 4 controller (     nudging)
88  persistent mode4_active phi_rate_cmd_degps err_pi_prev engaged_prev
89
90  MAX_ROT_SAMPLES = 16384;
91
92  if isempty(t_in_period)
93      t_in_period  = 0; high_count = 0; sample_count = 0; duty_accum =
            0; duty_current = 0; acc = 0;
94
95      phi_raw_rad  = PHI_START_DEG * PI_OVER_180;
96
97      last_phase_wrapped = wrapToPi(phase);
98      theta_unwrapped    = 0.0;
99      rot_start_theta    = 0.0;
100
101     last_trigger            = false;
102     last_artificial_trigger = false;
```

```
103     last_tick_time          = clock;
104
105     vhat_ceiling     = bus_max_from_coeffs(BUS_COEFFS);
106     Vbus_max         = max(vhat_ceiling, vmin_sane);
107     V_MIN_STEP_CONST = Vbus_max * effective_step;
108     last_effective_step = effective_step;
109
110     err_buf       = zeros(1, MAX_ROT_SAMPLES);
111     cur_len       = 0;
112     mean_prev_rot = 0.0;
113
114     % MODE 4 init
115     mode4_active       = false;
116     phi_rate_cmd_degps = PHI_RATE_BASE_DEFAULT_DEGPS * SENSE_DIR; %
        start at baseline
117     err_pi_prev        = 0.0;
118     engaged_prev       = false;
119 end
120
121 % Recompute constant V_min_step if quantization changed
122 if abs(effective_step - last_effective_step) > eps
123     vhat_ceiling     = bus_max_from_coeffs(BUS_COEFFS);
124     Vbus_max         = max(vhat_ceiling, vmin_sane);
125     V_MIN_STEP_CONST = Vbus_max * effective_step;
126     last_effective_step = effective_step;
127 end
128
129 % ===== Outputs (init) =====
130 phi_raw_deg_out   = phi_raw_rad / PI_OVER_180;
131 err_mean_prev_rot = mean_prev_rot;
132 err_dev_used      = 0.0;   % default
133 err_dev_raw       = 0.0;
134 err_dev_signage   = 0.0;
```

```
135
136 % ===== Tick edge detection =====
137 use_artificial = (lock == 0);
138 if use_artificial
139     trig_now = (artificial_trigger_control ~= 0);
140     edge     = trig_now && ~last_artificial_trigger;
141     last_artificial_trigger = trig_now;
142 else
143     trig_now = (trigger_control ~= 0);
144     edge     = trig_now && ~last_trigger;
145     last_trigger = trig_now;
146 end
147
148 % ===== MODE 4 entry: start      from baseline =====
149 if MODE == 4 && ~mode4_active
150     phi_rate_cmd_degps = PHI_RATE_BASE_DEFAULT_DEGPS * SENSE_DIR;
151     err_pi_prev   = 0.0;
152     engaged_prev  = false;
153     mode4_active  = true;
154 elseif MODE ~= 4
155     mode4_active  = false;
156 end
157
158 % ===== Select bus (uses current   ) =====
159 phi_use_rad = phi_raw_rad;
160 switch MODE
161     case 0
162         Vbus_used = Vbus_max;
163     case 1
164         if ~isempty(Vbus_inst), Vbus_used = max(Vbus_inst, vmin_sane
            ); else, Vbus_used = Vbus_max; end
165     otherwise % 2/4
166         Vbus_used = max(bus_hat_from_phase(phase + phi_use_rad,
```

```
            BUS_COEFFS), vmin_sane);
end

% ===== Estimate Vload =====
if ~isempty(V_load_meas)
    Vload_est = V_load_meas;
elseif ~isempty(Vbus_inst)
    Vload_est = duty_current * max(Vbus_inst, vmin_sane);
else
    Vload_est = duty_current * Vbus_used;
end

% ===== Errors =====
e_raw           = Vload_est - V_cmd;
err_dev_raw     = e_raw - mean_prev_rot;
err_dev_signage = err_dev_raw;   % simple passthrough for
   compatibility

% Deadzone after engage
if Vload_est >= VLOAD_PI_MIN_V
    mag_over = abs(err_dev_raw) - ERR_PI_GATE_MIN_V;
    if mag_over > 0
        err_dev_used = sign(err_dev_raw) * mag_over;  % Volts after
           deadzone
    else
        err_dev_used = 0.0;
    end
else
    err_dev_used = 0.0;
end

% ===== Buffer error for per-rotation mean =====
if cur_len < MAX_ROT_SAMPLES
```

```
197     cur_len = cur_len + 1; err_buf(cur_len) = e_raw;
198 else
199     err_buf(1:end-1) = err_buf(2:end); err_buf(end) = e_raw;
200 end
201
202 % ===== TICK: NUDGE      ONLY (hold-last when not engaged) =====
203 if edge && MODE == 4
204     % tick_dt calc
205     if ~isempty(N_tick_period) && (N_tick_period > 0)
206         tick_dt = N_tick_period;
207     else
208         tick_dt = clock - last_tick_time;
209         if isempty(tick_dt) || ~isfinite(tick_dt) || tick_dt <= 0,
              tick_dt = 0; end
210     end
211     last_tick_time = clock;
212
213     engaged = (Vload_est >= VLOAD_PI_MIN_V);
214
215     if engaged && (tick_dt > 0)
216         if ~engaged_prev, err_pi_prev = err_dev_used; end
217         derr_dt = (err_dev_used - err_pi_prev) / tick_dt;   % V/s
218
219         % P term is OFF when converging (slope < 0)
220         if derr_dt >= 0
221             p_term = KRATE_P_degps_per_V * err_dev_used;
222         else
223             p_term = 0.0;
224         end
225
226         % Unified D term (signed)
227         d_term = KRATE_D_degps_per_Vps * derr_dt;
228
```

```
229         % *** The nudge is applied to      (rate)     not to    ***
230         nudge = (p_term + d_term) * SENSE_DIR;
231         phi_rate_cmd_degps = phi_rate_cmd_degps + nudge;   % HOLD-
                LAST policy
232
233         err_pi_prev = err_dev_used;
234     else
235         % HOLD-LAST: do NOT reset phi_rate_cmd_degps to baseline
                here.
236         err_pi_prev = err_dev_used;  % keep continuity for next
                derivative
237     end
238
239     engaged_prev = engaged;
240
241     % % fprintf('[%.6f] M4: engaged=%d err=%.4f derr=%.4f cmdRate
            =%.3f\n', ...
242     % %     clock, engaged, err_dev_used, derr_dt,
            phi_rate_cmd_degps);
243 end
244
245 % ===== CONTINUOUS STEP:    =          dt (only place    changes)
        =====
246 phi_raw_rad     = wrapToPi(phi_raw_rad + (phi_rate_cmd_degps *
        dt_eff) * PI_OVER_180);
247 phi_raw_deg_out = phi_raw_rad / PI_OVER_180;
248
249 % ===== Rotation handling: update bias once per 2   of input phase
        =====
250 theta_now = wrapToPi(phase);
251 dtheta    = theta_now - last_phase_wrapped;
252 if dtheta < -pi, dtheta = dtheta + 2*pi; elseif dtheta > pi, dtheta
        = 0.0; end
```

```
253 last_phase_wrapped = theta_now;
254 theta_unwrapped    = theta_unwrapped + dtheta;
255
256 if (theta_unwrapped - rot_start_theta) >= 2*pi - 1e-6
257     if cur_len > 0, mean_prev_rot = sum(err_buf(1:cur_len)) /
            cur_len; else, mean_prev_rot = 0.0; end
258     cur_len = 0;
259
260     rot_start_theta   = rot_start_theta + 2*pi;
261     err_mean_prev_rot = mean_prev_rot;
262
263     % refresh outputs for continuity
264     e_raw           = Vload_est - V_cmd;
265     err_dev_raw     = e_raw - mean_prev_rot;
266     err_dev_signage = err_dev_raw;
267
268     if Vload_est >= VLOAD_PI_MIN_V
269         mag_over = abs(err_dev_raw) - ERR_PI_GATE_MIN_V;
270         if mag_over > 0, err_dev_used = sign(err_dev_raw) * mag_over
                ; else, err_dev_used = 0.0; end
271     else
272         err_dev_used = 0.0;
273     end
274
275     phi_raw_deg_out = phi_raw_rad / PI_OVER_180;
276 end
277
278 % ===== Duty target / PWM =====
279 V_cmd    = max(V_cmd, 0);
280 d_target = min(max(V_cmd / Vbus_used, 0), 1 - headroom);
281
282 base_bin = floor(d_target / duty_step);
283 frac     = d_target - base_bin * duty_step;
```

```
284 if enable_dither
285     acc = acc + frac;
286     if acc >= duty_step
287         acc = acc - duty_step; duty_current = (base_bin + 1) *
                duty_step;
288     else
289         duty_current = base_bin * duty_step;
290     end
291 else
292     duty_current = base_bin * duty_step; acc = 0;
293 end
294
295 t_in_period = t_in_period + dt_eff;
296 new_period  = false;
297 if t_in_period >= Ts_pwm, t_in_period = t_in_period - Ts_pwm;
       new_period = true; end
298 pwm = double((t_in_period * f_pwm) < duty_current);
299
300 sample_count = sample_count + 1;
301 if pwm == 1, high_count = high_count + 1; end
302 if new_period
303     if sample_count > 0, duty_accum = high_count / sample_count; end
304     high_count = 0; sample_count = 0;
305 end
306 duty_actual = duty_accum;
307
308 % ===== V_min_step =====
309 V_min_step = V_MIN_STEP_CONST;
310
311 % final freshness
312 phi_raw_deg_out   = phi_raw_rad / PI_OVER_180;
313 err_mean_prev_rot = mean_prev_rot;
314
```

```
end

% ---------- Helpers ----------
function Vhat = bus_hat_from_phase(theta, TH)
V0 = TH(1); K  = max(0, floor((numel(TH)-1)/2));
c1 = cos(theta); s1 = sin(theta); ck = c1; sk = s1; H = 0; col = 2;
for k = 1:K
    ak = TH(col); bk = TH(col+1);
    H  = H + ak*ck + bk*sk;
    ck_next = c1*ck - s1*sk; sk_next = s1*ck + c1*sk;
    ck = ck_next; sk = sk_next; col = col + 2;
end
Vhat = V0 + H;
end

function vmax = bus_max_from_coeffs(TH)
V0 = TH(1); K  = max(0, floor((numel(TH)-1)/2));
amp_sum = 0; col = 2;
for k = 1:K
    ak = TH(col); bk = TH(col+1);
    amp_sum = amp_sum + hypot(ak, bk);
    col = col + 2;
end
vmax = V0 + amp_sum;
end

function a = wrapToPi(a), a = mod(a + pi, 2*pi) - pi; end
```

Listing 4.6: *Compensator*

```
function [V_out, d] = mpc_ripple_sync(V_applied, I_meas, omega_meas,
    omega_ref, ...
    trigger_control, artificial_trigger_control, lock, pll_phase,
       ...
```

```
    clock, V_min_step, N_tick_period)
% True receding-horizon MPC with phase-gated R sampling.
% - Replans every actuation tick (burst/hold grid, quantized steps & ramp limit)
% - d backsolver at fast rate
% - R backsolver uses decimated, phase-gated samples only
%
% Inputs:
%   V_applied, I_meas, omega_meas, omega_ref : plant & reference signals
%   trigger_control, artificial_trigger_control, lock : edge gating
%   pll_phase : current PLL phase [rad], wrap to [0, 2*pi) upstream if desired
%   clock : current time (s)
%   V_min_step : DAC/ladder min step
%   N_tick_period : control-edge period (s). If <= ~0, defaults to 1/12000
%
% Outputs:
%   V_out : next commanded voltage (quantized & ramp-limited)
%   d     : best-fit constant disturbance on omega dot over recent fast window

%#codegen
coder.extrinsic('fprintf')

% === Core MPC / Observer Parameters ===
horizon_len_ctrl = 10;        % integer >= 1
period_ticks     = 20;        % plan every N fast edges
default_tick_dt  = 0.0000833333333333; % 1/12000
min_improve      = 4e-3;
d_err_tol        = 0.001;
max_ramp         = 200;       % V/s
```

```
V_min_step_floor = 0.025;     % min hardware step used by controller

if N_tick_period > 1e-6
    tick_dt = N_tick_period;
else
    tick_dt = default_tick_dt;
end
tick_actuation = period_ticks * tick_dt;
mini_dt        = tick_actuation;

buffer_len = 2 * period_ticks;   % for d observer (fast buffer)
V_max      = 84;
V_min      = 0;

% AC Parameters (R=c, kv=g)
a = 3.72;  b = 0.1002;  g = 0.186;  L = 0.0001006;
c = 0.01998;            % planner's constant R (ohms)
V_scale = 1;            % used by controller ONLY (NOT by R
   backsolve)

lambda_final = 0.7;     % omega error cost

% === Quantization & ramp (action grid) ===
V_step          = max(V_min_step, V_min_step_floor);
V_ramp_per_tick = max_ramp * tick_actuation;
n_bins          = max(1, floor(V_ramp_per_tick / V_step));
action_grid     = -n_bins:n_bins;

% === R-Observer cadence (phase-gated) ===
R_period_ticks = 400;   % arm once every 400 control edges
R_buffer_len   = 20;    % decimated samples stored for R

% ---- Phase-gated R sample config (internal) ----
```

```
63 PHASE_TARGET      = 0.0;   % radians; set your grab point here (e.g
       ., 0, pi/2, pi)
64 PHASE_BAND_MIN    = 0.035; % ~2.0 degrees minimum band to avoid
       misses at low dphi
65 PHASE_BAND_MARGIN = 0.005; % cushion added to 0.5*dphi_tick
66
67 % === Persistent State ===
68 persistent V_out_held initialized
69 persistent V_hist I_hist omega_hist last_trigger
       last_artificial_trigger
70 persistent omega_buf I_buf V_fast_buf domega_buf t_buf tick_count
71 persistent action_tick_count
72 persistent first_step_hold first_step_count
73 persistent buffer_filled
74 persistent omega_pred_buf
75 persistent omega_pred_err_buf
76 persistent last_d last_err_best_d
77
78 % Dedicated R buffers (phase-gated decimation)
79 persistent R_I_prev_buf R_I_now_buf R_dI_buf R_V_buf R_omega_buf
       R_t_buf R_buffer_filled
80 persistent last_R last_err_best_R
81
82 % Phase sampling helpers
83 persistent last_phase phase_waiting
84
85 if isempty(V_out_held)
86     V_out_held = 0;
87     initialized = false;
88
89     V_hist     = V_applied * ones(1,3);
90     I_hist     = I_meas    * ones(1,3);
91     omega_hist = omega_meas* ones(1,3);
```

```
92      last_trigger = 0;
93      last_artificial_trigger = 0;
94
95      omega_buf    = omega_meas*ones(1,buffer_len);
96      I_buf        = I_meas   *ones(1,buffer_len);
97      V_fast_buf   = V_applied*ones(1,buffer_len);
98      domega_buf   = zeros(1,buffer_len);
99      t_buf        = zeros(1,buffer_len);
100
101     tick_count         = 0;
102     action_tick_count  = 0;
103     first_step_hold    = false;
104     first_step_count   = 0;
105     buffer_filled      = 0;
106     omega_pred_buf     = omega_meas*ones(1,buffer_len);
107     omega_pred_err_buf = zeros(1,buffer_len);
108
109     last_d         = 0;
110     last_err_best_d= Inf;
111
112     R_I_prev_buf   = I_meas     * ones(1, R_buffer_len);
113     R_I_now_buf    = I_meas     * ones(1, R_buffer_len);
114     R_dI_buf       = zeros(1, R_buffer_len);
115     R_V_buf        = V_applied  * ones(1, R_buffer_len);
116     R_omega_buf    = omega_meas * ones(1, R_buffer_len);
117     R_t_buf        = zeros(1, R_buffer_len);
118     R_buffer_filled = 0;
119
120     last_R         = c;   % start at planner's constant R
121     last_err_best_R= Inf;
122
123     last_phase     = NaN;
124     phase_waiting  = false;
```

```
125 end
126
127 d = last_d;
128
129 % === 3-point moving average on inputs ===
130 V_hist         = [V_hist(2:3), V_applied];
131 I_hist         = [I_hist(2:3), I_meas];
132 omega_hist     = [omega_hist(2:3), omega_meas];
133 V_applied_filt = (V_hist(1)+V_hist(2)+V_hist(3))/3;
134 I_meas_filt    = (I_hist(1)+I_hist(2)+I_hist(3))/3;
135 omega_meas_filt= (omega_hist(1)+omega_hist(2)+omega_hist(3))/3;
136
137 % === Trigger edge detect ===
138 if lock == 0
139     control_edge = (artificial_trigger_control ~= 0) && (
        last_artificial_trigger == 0);
140 else
141     control_edge = (trigger_control ~= 0) && (last_trigger == 0);
142 end
143
144 if control_edge
145     tick_count = tick_count + 1;
146
147     % --- fast buffers (update first) ---
148     if buffer_filled < buffer_len
149         buffer_filled = buffer_filled + 1;
150     end
151     omega_buf  = [omega_buf(2:end), omega_meas];
152     I_buf      = [I_buf(2:end), I_meas];
153     V_fast_buf = [V_fast_buf(2:end), V_applied];
154     t_buf      = [t_buf(2:end), clock];
155
156     if buffer_filled > 1
```

```
157         domega = (omega_buf(end) - omega_buf(end-1)) / tick_dt;
158     else
159         domega = 0;
160     end
161     domega_buf = [domega_buf(2:end), domega];
162 
163     % ---------- ARM phase-gated R sampler every R_period_ticks
        ----------
164     if mod(tick_count, R_period_ticks) == 0
165         phase_waiting = true;
166     end
167 
168     % ---------- R PHASE-GATED SAMPLE ----------
169     % Grab one R sample at the next rising crossing of PHASE_TARGET
170     % after the 'phase_waiting' flag is armed. Uses latest fast-
        buffer pair.
171     if ~isnan(pll_phase)
172         prev_phase = last_phase;
173         if isnan(prev_phase)
174             prev_phase = pll_phase;
175         end
176 
177         if phase_waiting && (buffer_filled >= 2)
178             % Estimate per-edge phase motion and dynamic band
179             dphi_tick  = abs(wrapToPi_local(pll_phase - prev_phase))
                ;
180             phase_band = max(PHASE_BAND_MIN, 0.5*dphi_tick +
                PHASE_BAND_MARGIN);
181 
182             % Crossing detection relative to the internally-set
                target
183             phi_prev_rel = wrapToPi_local(prev_phase  - PHASE_TARGET
                );
```

```
184             phi_now_rel  = wrapToPi_local(pll_phase   - PHASE_TARGET
                    );
185
186             crossed = (phi_prev_rel < 0) && (phi_now_rel >= 0); %
                    rising through target
187             in_band = abs(phi_now_rel) <= phase_band;
188
189             if crossed && in_band
190                 I_prev_fast    = I_buf(end-1);
191                 I_now_fast     = I_buf(end);
192                 dI_fast        = (I_now_fast - I_prev_fast)/tick_dt;
193                 V_now_fast     = V_fast_buf(end);
194                 omega_now_fast = omega_buf(end);
195                 t_now_fast     = t_buf(end);
196
197                 if R_buffer_filled < R_buffer_len
198                     R_buffer_filled = R_buffer_filled + 1;
199                 end
200                 R_I_prev_buf  = [R_I_prev_buf(2:end),  I_prev_fast];
201                 R_I_now_buf   = [R_I_now_buf(2:end),   I_now_fast];
202                 R_dI_buf      = [R_dI_buf(2:end),      dI_fast];
203                 R_V_buf       = [R_V_buf(2:end),       V_now_fast];
204                 R_omega_buf   = [R_omega_buf(2:end),
                        omega_now_fast];
205                 R_t_buf       = [R_t_buf(2:end),       t_now_fast];
206
207                 fprintf('[%.5f] R phase-sample @phase~=%.3f rad (
                        band=%.4f): I_prev=%.5f I_now=%.5f dI=%.5f V=%.5f
                         omega=%.5f\n', ...
208                     clock, pll_phase, phase_band, I_prev_fast,
                            I_now_fast, dI_fast, V_now_fast,
                            omega_now_fast);
209
```

```
210                 phase_waiting = false; % disarm until next arm
211             end
212         end
213
214         % update last_phase each control edge
215         last_phase = pll_phase;
216     end
217     % -------- END R PHASE-GATED SAMPLE --------
218
219     % === Build omega prediction error (for d) ===
220     for k = 1:buffer_filled
221         buf_idx = buffer_len - buffer_filled + k;
222         if k == 1
223             omega_pred_buf(buf_idx)     = omega_buf(buf_idx);
224             omega_pred_err_buf(buf_idx) = NaN;
225         else
226             idx_prev    = buf_idx - 1;
227             omega_prev  = omega_buf(idx_prev);
228             I_now_fast  = I_buf(buf_idx);
229             omega_pred  = omega_prev + (a*I_now_fast - b*omega_prev)
                    *tick_dt;
230             omega_pred_buf(buf_idx)     = omega_pred;
231             omega_pred_err_buf(buf_idx) = omega_pred - omega_buf(
                    buf_idx);
232         end
233     end
234
235     if ~initialized
236         initialized        = true;
237         V_out_held         = V_step;
238         first_step_hold    = true;
239         first_step_count   = 1;
240         action_tick_count  = 0;
```

```
241         V_out = V_out_held;
242         fprintf('[%.5f] First trigger: V_out set to hardware step
                %.3f V\n', clock, V_step);
243         last_trigger = trigger_control;
244         last_artificial_trigger = artificial_trigger_control;
245         return;
246     end
247
248     if first_step_hold
249         V_out = V_out_held;
250         first_step_count = first_step_count + 1;
251         if first_step_count >= 4
252             first_step_hold = false;
253         end
254         last_trigger = trigger_control;
255         last_artificial_trigger = artificial_trigger_control;
256         return;
257     end
258
259     % === Only plan / print on actuation ticks ===
260     if action_tick_count == 0
261         fprintf('[%.5f] ---- Most Recent %.0f Buffer Ticks (tick_dt
                =%.6f) ----\n', ...
262             clock, buffer_filled, tick_dt);
263         fprintf(['Idx  Time         ',...
264                  'Current      ',...
265                  'Omega        ',...
266                  'Omega_pred   ',...
267                  'PredErr      ',...
268                  'Domega       ',...
269                  'd_obs\n']);
270         for k = 1:buffer_filled
271             buf_idx = buffer_len - buffer_filled + k;
```

```
272             if k == 1
273                 fprintf('%3.0f  %10.5f  %12.5f  %12.5f  %12s  %12s
                        %12.5f  %10s\n', ...
274                     k, t_buf(buf_idx), I_buf(buf_idx), omega_buf(
                            buf_idx), ...
275                     '--', '--', domega_buf(buf_idx), '--');
276             else
277                 omega_now  = omega_buf(buf_idx);
278                 omega_prev = omega_buf(buf_idx-1);
279                 I_now_fast = I_buf(buf_idx);
280                 d_obs_row  = (omega_now - omega_prev)/tick_dt - a*
                        I_now_fast + b*omega_now;
281                 fprintf('%3.0f  %10.5f  %12.5f  %12.5f  %12.5f
                        %12.5f  %12.5f  %10.3f\n', ...
282                     k, t_buf(buf_idx), I_buf(buf_idx), omega_buf(
                            buf_idx), ...
283                     omega_pred_buf(buf_idx), omega_pred_err_buf(
                            buf_idx), ...
284                     domega_buf(buf_idx), d_obs_row);
285             end
286         end
287         fprintf('-----------------------------------------\n');
288
289         % ---- Disturbance backsolver (d) ----
290         best_d = 0; best_err = Inf; err_zero = 0;
291         if buffer_filled == buffer_len
292             for k = 2:buffer_filled
293                 buf_idx    = buffer_len - buffer_filled + k;
294                 omega_prev = omega_buf(buf_idx-1);
295                 I_now_fast = I_buf(buf_idx);
296                 omega_pred0= omega_prev + (a*I_now_fast - b*
                        omega_prev)*tick_dt;
297                 err_zero   = err_zero + abs(omega_pred0 - omega_buf(
```

```
                    buf_idx));
298             end
299             for d_test = -10000:1:10000
300                 abs_err_sum = 0;
301                 for k = 2:buffer_filled
302                     buf_idx    = buffer_len - buffer_filled + k;
303                     omega_prev = omega_buf(buf_idx-1);
304                     I_now_fast = I_buf(buf_idx);
305                     omega_pred = omega_prev + (a*I_now_fast - b*
                            omega_prev + d_test)*tick_dt;
306                     abs_err_sum = abs_err_sum + abs(omega_pred -
                            omega_buf(buf_idx));
307                 end
308                 if abs_err_sum < best_err
309                     best_err = abs_err_sum;
310                     best_d   = d_test;
311                 end
312             end
313             improve = err_zero - best_err;
314             fprintf('Total abs prediction error, d=0:    %.5f\n',
                    err_zero);
315             fprintf('Total abs prediction error, best d: %.5f\n',
                    best_err);
316
317             if improve >= min_improve
318                 if abs(best_err - last_err_best_d) < d_err_tol
319                     d = last_d;
320                     fprintf('Holding d steady at %.3f (best_err=%.5f
                            ~ last_err=%.5f)\n', d, best_err,
                            last_err_best_d);
321                 else
322                     last_d = best_d; d = last_d; last_err_best_d =
                            best_err;
```

```
323                     fprintf('Best-fit d to minimize total abs error:
                            %.3f (improvement = %.5f)\n', d, improve);
324                 end
325             else
326                 d = 0; last_d = 0; last_err_best_d = Inf;
327                 fprintf('No d found with sufficient improvement (%.5
                        f < min %.5f), using d = 0\n', ...
328                     improve, min_improve);
329             end
330             fprintf('Current R estimate (from R backsolver): %.5f\n'
                    , last_R);
331         else
332             fprintf('Current R estimate (from R backsolver): %.5f\n'
                    , last_R);
333         end
334
335         % ---- R Backsolve TABLE (tail print) ----
336         if R_buffer_filled >= 1
337             R_model  = c;
338             fprintf('[%.5f] ---- R Backsolve (phase-gated decimation
                    ) | R_model=%.5f, g=%.5f ----\n', ...
339                     clock, R_model, g);
340             fprintf(['Ord  Time        ',...
341                      'I_prev       ',...
342                      'I_now        ',...
343                      'dI(fast)     ',...
344                      'V_now        ',...
345                      'V_model      ',...
346                      'Resid        ',...
347                      'R_obs\n']);
348             start_idx = R_buffer_len - R_buffer_filled + 1;
349             ord = 0;
350             for kR = start_idx:R_buffer_len
```

```
                ord = ord + 1;
                tR    = R_t_buf(kR);
                Iprev = R_I_prev_buf(kR);
                Inow  = R_I_now_buf(kR);
                dIf   = R_dI_buf(kR);
                Vnow  = R_V_buf(kR);
                omg   = R_omega_buf(kR);

                V_model = (L*dIf + R_model*Inow + g*omg);
                V_resid = Vnow - V_model;

                if abs(Inow) > 1e-9
                    R_obs = (Vnow - L*dIf - g*omg) / Inow;
                else
                    R_obs = NaN;
                end

                fprintf('%3.0f  %10.5f  %12.5f  %12.5f  %12.5f
                   %12.5f  %12.5f  %12.5f  %10.6f\n', ...
                        ord, tR, Iprev, Inow, dIf, Vnow, V_model,
                           V_resid, R_obs);
            end

            if R_buffer_filled == R_buffer_len
                err_R_at_model = 0; best_R_err = Inf; best_R =
                   last_R;
                for kR = 1:R_buffer_len
                    Inow  = R_I_now_buf(kR);
                    dIf   = R_dI_buf(kR);
                    Vnow  = R_V_buf(kR);
                    omg   = R_omega_buf(kR);
                    V_res = Vnow - (L*dIf + R_model*Inow + g*omg);
                    err_R_at_model = err_R_at_model + abs(V_res);
```

```
381                 end
382                 for R_test = 0.0000:0.0005:0.2000
383                     abs_err_sum_R = 0;
384                     for kR = 1:R_buffer_len
385                         Inow  = R_I_now_buf(kR);
386                         dIf   = R_dI_buf(kR);
387                         Vnow  = R_V_buf(kR);
388                         omg   = R_omega_buf(kR);
389                         V_res = Vnow - (L*dIf + R_test*Inow + g*omg)
                                ;
390                         abs_err_sum_R = abs_err_sum_R + abs(V_res);
391                     end
392                     if abs_err_sum_R < best_R_err
393                         best_R_err = abs_err_sum_R; best_R = R_test;
394                     end
395                 end
396                 fprintf('Total abs residual, R=R_model (%.5f): %.6f\
                    n', R_model, err_R_at_model);
397                 fprintf('Total abs residual, best R:          %.6f
                    (R*=%.5f)\n', best_R_err, best_R);
398
399                 improve_R = err_R_at_model - best_R_err;
400                 if improve_R >= min_improve
401                     last_R = best_R;
402                     last_err_best_R = best_R_err;
403                     fprintf('[%.5f] R update: Best-fit R=%.5f (
                        improvement=%.6f)\n', clock, last_R,
                        improve_R);
404                 else
405                     last_R = c;
406                     last_err_best_R = Inf;
407                     fprintf('[%.5f] R update: no sufficient
                        improvement (%.6f < %.6f), keeping nominal R
```

```
                    =%.5f\n', ...
408                         clock, improve_R, min_improve, c);
409             end
410         else
411             fprintf('R buffer filling: %.0f/%.0f samples\n',
                    R_buffer_filled, R_buffer_len);
412         end
413         fprintf('
                ------------------------------------------------\n'
                );
414     else
415         fprintf('[%.5f] ---- R Backsolve: waiting for phase-
                gated samples (%.0f/%.0f) ----\n', ...
416             clock, R_buffer_filled, R_buffer_len);
417     end
418
419     % ---- Burst/Hold Path Planning ----
420     n_plans = numel(action_grid) * horizon_len_ctrl;
421     plans   = zeros(n_plans, horizon_len_ctrl);
422     row     = 1;
423     for idx_a = 1:numel(action_grid)
424         a_ = action_grid(idx_a);
425         for drop = 0:(horizon_len_ctrl-1)
426             plan = zeros(1, horizon_len_ctrl);
427             for kk = 1:(horizon_len_ctrl-drop)
428                 plan(kk) = a_;
429             end
430             plans(row, :) = plan;
431             row = row + 1;
432         end
433     end
434
435     fprintf('\n[%.5f] MPC Burst/Hold Path Planning (%.0f plans,
```

```
            V_step=%.3f, n_bins=%.0f)\n', ...
436             clock, n_plans, V_step, n_bins);
437         fprintf('Plan          FinalOmega   FinalErr    Cost    [
                V_path]\n');
438
439         best_cost       = Inf;
440         best_path_local = zeros(1, horizon_len_ctrl);
441
442         I_sim0     = I_meas_filt;
443         omega_sim0 = omega_meas_filt;
444
445         for i = 1:n_plans
446             actions = plans(i,:);
447             V_path  = zeros(1, horizon_len_ctrl);
448             for k = 1:horizon_len_ctrl
449                 V_path(k) = V_out_held + sum(actions(1:k))*V_step;
450                 if V_path(k) < V_min, V_path(k) = V_min; end
451                 if V_path(k) > V_max, V_path(k) = V_max; end
452             end
453
454             I_tmp  = I_sim0;
455             om_tmp = omega_sim0;
456             cum_err = 0.0;
457             for k = 1:horizon_len_ctrl
458                 V_sim  = V_path(k);
459                 dI_sim = (V_scale * V_sim - c*I_tmp - g*om_tmp) / L;
                        % planner uses constant c (R=c)
460                 I_tmp  = I_tmp + dI_sim * mini_dt;
461                 dome   =  a*I_tmp - b*om_tmp + d;
462                 om_tmp = om_tmp + dome * mini_dt;
463                 cum_err = cum_err + abs(om_tmp - omega_ref);
464             end
465
```

```
466             cost = lambda_final * cum_err;
467
468             % Print plan line
469             action_str = '';
470             for j = 1:numel(actions)
471                 action_str = [action_str, sprintf('%3.0f', actions(j
                        ))];
472             end
473             fprintf('%s  %10.4f  %9.4f  %12.6f   [', action_str,
                    om_tmp, (om_tmp - omega_ref), cost);
474             for m = 1:numel(V_path)
475                 fprintf('%5.2f ', V_path(m));
476             end
477             fprintf(']\n');
478
479             if cost < best_cost
480                 best_cost       = cost;
481                 best_path_local = V_path;
482             end
483         end
484
485         V_out_held = best_path_local(1);
486         if V_out_held < V_min, V_out_held = V_min; end
487         if V_out_held > V_max, V_out_held = V_max; end
488
489         fprintf('\n>>> Selected Best plan, V_out=%.2f at t=%.5f\n',
                V_out_held, clock);
490         fprintf('Chosen V_path: [');
491         for m = 1:numel(best_path_local)
492             fprintf('%5.2f ', best_path_local(m));
493         end
494         fprintf(']\n');
495     end
```

```
496
497     V_out = V_out_held;
498
499     action_tick_count = action_tick_count + 1;
500     if action_tick_count >= period_ticks
501         action_tick_count = 0;
502     end
503
504 else
505     V_out = V_out_held;
506 end
507
508 % Latch triggers
509 last_trigger = trigger_control;
510 last_artificial_trigger = artificial_trigger_control;
511
512 end % ===== end main function =====
513
514
515 function y = wrapToPi_local(x)
516 % Wrap angle in radians to [-pi, pi)
517 y = mod(x + pi, 2*pi) - pi;
518 end
```

Listing 4.7: *MPC with Debug*

```
1 clearvars; close all; clc;
2
3 model_path = 'C:\Users\sethm\OneDrive\Desktop\Lehigh\ECE 450 Grid\
    Thesis Update\Week 3';
4 model_name = 'LCLC_PI';
5
6 cd(model_path);
7 if ~isfile([model_name, '.slx'])
```

```
8       error('Model file %s.slx not found in path: %s', model_name,
            model_path);
9   end
10
11  load_system(model_name);
12
13  disp('Running simulation...');
14  simOut = sim(model_name);
15
16  theta_out            = extractSignal(simOut,'theta_out');
                        % deg (angle-domain grid)
17  theta_time_out       = extractSignal(simOut,'theta_time_out');
                    % deg (same length as time-domain streams)
18  R_theta_out          = extractSignal(simOut,'R_theta_out');
                      % ohm on theta_out grid
19  R_theta_filt_out     = extractSignal(simOut,'R_theta_filtered_out');
                % ohm on theta_out grid
20
21  V_control_out        = extractSignal(simOut,'V_control_out');
                    % time-domain (V)
22  V_control_ref_out    = extractSignal(simOut,'V_control_ref_out');
                % time-domain (V, ref)
23  I_control_out        = extractSignal(simOut,'I_control_out');
                    % time-domain (A)
24  err_used_out         = extractSignal(simOut,'err_used_out');
                     % time-domain (V)
25
26  % Duty (time-domain     slice by theta bins)
27  duty_ts = firstNonEmpty( ...
28      extractSignal(simOut,'duty_actual_out'), ...
29      extractSignal(simOut,'duty_actual'), ...
30      extractSignal(simOut,'duty_out'), ...
31      extractSignal(simOut,'duty'));
```

```
32
33 % Phase raw (time-domain; will coerce to degrees)
34 phi_candidates = { ...
35     'phi_raw_deg_out','phi_raw_deg','phi_deg_out','phi_deg', ...
36     'phi_out_deg','phi_out', 'phi_raw_out_deg','phi_raw_out', ...
37     'phi_inner_deg','phi_inner','phi_raw','phi' ...
38 };
39 phi_raw_any = [];
40 for k = 1:numel(phi_candidates)
41     s = extractSignal(simOut, phi_candidates{k});
42     if ~isempty(s), phi_raw_any = s; break; end
43 end
44 phi_raw_deg_ts = coerceDegrees(phi_raw_any);
45
46 % Speed    (time-domain, plotted vs time)
47 omega_ts = firstNonEmpty( ...
48     extractSignal(simOut,'omega_out'), ...
49     extractSignal(simOut,'omega_meas'), ...
50     extractSignal(simOut,'omega'));
51
52 % time vector
53 t = [];
54 try, t = double(simOut.tout(:)); catch, t = []; end
55
56 % Row-shape
57 toRow = @(x) double(x(:)).';
58 theta_out         = toRow(theta_out);
59 theta_time_out    = toRow(theta_time_out);
60 R_theta_out       = toRow(R_theta_out);
61 R_theta_filt_out  = toRow(R_theta_filt_out);
62 V_control_out     = toRow(V_control_out);
63 V_control_ref_out = toRow(V_control_ref_out);
64 I_control_out     = toRow(I_control_out);
```

```
err_used_out      = toRow(err_used_out);
duty_ts           = toRow(duty_ts);
phi_raw_deg_ts    = toRow(phi_raw_deg_ts);
omega_ts          = toRow(omega_ts);

% ============================================================
% Window by first/last nonzero in theta_out
% ============================================================
if isempty(theta_out), error('theta_out missing'); end
tol_theta_nz = 1e-12;
nz_mask = abs(theta_out) > tol_theta_nz;
if ~any(nz_mask), error('theta_out is all zero (under tolerance).');
    end

idx_theta_first = find(nz_mask,1,'first');
idx_theta_last  = find(nz_mask,1,'last');

theta_first = theta_out(idx_theta_first);
theta_last  = theta_out(idx_theta_last);

% Trim R* on theta_out grid
idx_theta_grid = idx_theta_first:idx_theta_last;
trimThetaGrid = @(sig) local_trim_theta_grid(sig, theta_out,
   idx_theta_grid);
R_trimmed      = trimThetaGrid(R_theta_out);
Rfilt_trimmed  = trimThetaGrid(R_theta_filt_out);

% ============================================================
% Map these angles to bin indices in theta_time_out
% ============================================================
if isempty(theta_time_out)
    error('theta_time_out missing (required to slice time-domain
       signals).');
```

```
end

idx_start_time = nearest_index(theta_time_out, theta_first);
idx_end_time   = nearest_index(theta_time_out, theta_last);

% Ensure increasing order, guard bounds
n_time = numel(theta_time_out);
i0 = max(1, min(n_time, min(idx_start_time, idx_end_time)));
i1 = max(1, min(n_time, max(idx_start_time, idx_end_time)));

idx_time_window = i0:i1;

% Slice time-domain arrays strictly by these bins (theta indexing)
theta_time_win = theta_time_out(idx_time_window);
Vctrl_win      = slice_safe(V_control_out,     idx_time_window);
Vref_win       = slice_safe(V_control_ref_out, idx_time_window);
Ictrl_win      = slice_safe(I_control_out,     idx_time_window);
ERR_win        = slice_safe(err_used_out,      idx_time_window);

% Duty & Phi (same slicing method)
DUTY_win       = slice_safe(duty_ts,        idx_time_window);
PHI_win_deg    = slice_safe(phi_raw_deg_ts, idx_time_window);

% times (for reference/annotation)
t_win = [];
if ~isempty(t)
    n_t   = min(numel(t), numel(theta_time_out));
    idx_ok = idx_time_window(idx_time_window <= n_t);
    if ~isempty(idx_ok), t_win = t(idx_ok); end
end

% Warnings if phi/duty are empty post-slice
if isempty(DUTY_win)
```

```
128     warning('DUTY_win is empty: check duty signal name or alignment.
            ');
129 end
130 if isempty(PHI_win_deg)
131     warning(['PHI_win_deg is empty: could not find a usable phi
            signal.\n' ...
132             'Tried names: %s'], strjoin(phi_candidates, ', '));
133 end
134
135 % ============================================================
136 % Visual style     consistent category colors
137 % ============================================================
138 % Base colors (RGB 0-1)
139 col.R.raw   = [0.00, 0.45, 0.74]; % BLUE
140 col.R.filt  = [0.00, 0.75, 0.75]; % TEAL
141 col.V.base  = [0.00, 0.60, 0.00]; % GREEN  (omega)
142 col.I.base  = [0.85, 0.33, 0.10]; % ORANGE/RED
143 col.E.base  = [0.49, 0.18, 0.56]; % PURPLE (phi)
144 col.D.base  = [0.65, 0.40, 0.15]; % BROWN  (duty)
145 col.TH.base = [0.20, 0.20, 0.20]; % DARK GRAY
146
147 % Tone helper
148 tone  = @(c, f) max(min(c .* f + (1-f)*1.0, 1.0), 0.0);
149
150 % Assign tones
151 cR_raw  = col.R.raw;
152 cR_filt = col.R.filt;
153 cV      = col.V.base;   % GREEN (omega)
154 cI      = col.I.base;
155 cE      = col.E.base;   % PURPLE (phi)
156 cD      = col.D.base;   % BROWN (duty)
157 cTH     = col.TH.base;
158
```

```
% ================================================================
% Plot knobs
% ================================================================
R_center1 = 0.02;
R_pad1    = 0.0005;

% R vs time
R_center_t = 0.02;
R_pad_t    = 0.0001;
t_zoom     = [0.19 0.25];

% Axis windows
theta_xlim    = [100 600];   % deg for theta-binned figs
V_ylim_fig2   = [18 32];     % V for Figure 2
omega_center  = 300;         % center for omega
omega_pad     = 5;           % +/- window

% ================================================================
% Figures
% ================================================================

% --- Figure 1: R(  ) raw + filtered ---
if ~isempty(R_trimmed) || ~isempty(Rfilt_trimmed)
    figure; hold on;
    if ~isempty(R_trimmed)
        plot(R_trimmed(:,1), R_trimmed(:,2), '-', 'LineWidth', 1.8, ...
            'Color', cR_raw,  'DisplayName', 'R(\theta) raw');
    end
    if ~isempty(Rfilt_trimmed)
        plot(Rfilt_trimmed(:,1), Rfilt_trimmed(:,2), '-', 'LineWidth', 2.2, ...
            'Color', cR_filt, 'DisplayName', 'R(\theta) filtered');
```

```
    end
    yline(R_center1, ':', sprintf('R_{ref}=%.3f \\Omega',R_center1),
        ...
        'HandleVisibility','off', 'Color', tone(cR_raw,0.6));
    xlabel('\theta (deg)'); ylabel('R(\theta) [\Omega]');
    title('Figure 1     R(\theta)'); grid on; legend('Location','
       best');
    if all(isfinite(theta_xlim)), xlim(theta_xlim); else,
       safe_xlim_from_vec(theta_out(idx_theta_grid)); end
    safe_ylim_centerpad(R_center1, R_pad1);
end

% --- Figure 2: V_control vs    ---
if ~isempty(Vctrl_win)
    figure; hold on;
    if ~isempty(Vref_win)
        plot(theta_time_win, Vref_win, ':', 'LineWidth', 1.8, ...
            'Color', tone(cV,0.7), 'DisplayName', 'V_{control,ref}(\
               theta)');
    end
    plot(theta_time_win, Vctrl_win, '-', 'LineWidth', 2.0, ...
        'Color', cV, 'DisplayName', 'V_{control}(\theta)');
    xlabel('\theta (deg)'); ylabel('V_{control} [V]');
    title('Figure 2     V_{control} vs \theta');
    grid on; legend('Location','best');
    if all(isfinite(theta_xlim)), xlim(theta_xlim); else,
       safe_xlim_from_vec(theta_time_win); end
    if all(isfinite(V_ylim_fig2)), ylim(V_ylim_fig2); end
end

% --- Figure 3: I_control vs    ---
if ~isempty(Ictrl_win)
    figure;
```

```
218     plot(theta_time_win, Ictrl_win, '-', 'LineWidth', 1.8, ...
219         'Color', cI, 'DisplayName', 'I_{control}(\theta)');
220     xlabel('\theta (deg)'); ylabel('I_{control} [A]');
221     title('Figure 3     I_{control} vs \theta');
222     grid on; legend('Location','best');
223     if all(isfinite(theta_xlim)), xlim(theta_xlim); else,
            safe_xlim_from_vec(theta_time_win); end
224 end
225
226 % --- Figure 4:    vs TIME ---
227 if ~isempty(t_win)
228     figure;
229     plot(t_win, theta_time_win, '-', 'LineWidth', 1.8, ...
230         'Color', cTH, 'DisplayName', '\theta(t)');
231     xlabel('Time [s]'); ylabel('\theta [deg]');
232     title('Figure 4     \theta vs Time'); grid on; legend('Location'
            ,'best');
233     safe_xlim_from_vec(t_win);
234 end
235
236 % --- Figure 5: err_used vs    ---
237 if ~isempty(ERR_win)
238     figure;
239     plot(theta_time_win, ERR_win, '-', 'LineWidth', 1.8, ...
240         'Color', cE, 'DisplayName', 'err_{used}(\theta)');
241     hold on; yline(0, ':', '0 V', 'HandleVisibility','off', 'Color',
            tone(cE,0.6));
242     xlabel('\theta (deg)'); ylabel('err_{used} [V]');
243     title('Figure 5     err\_{used} vs \theta'); grid on; legend('
            Location','best');
244     if all(isfinite(theta_xlim)), xlim(theta_xlim); else,
            safe_xlim_from_vec(theta_time_win); end
245 end
```

```
246
247 % --- Figure 6: R(  ) (single curve) ---
248 if ~isempty(Rfilt_trimmed) || ~isempty(R_trimmed)
249     figure; hold on;
250     if ~isempty(Rfilt_trimmed)
251         plot(Rfilt_trimmed(:,1), Rfilt_trimmed(:,2), '-', 'LineWidth
            ', 2.2, ...
252             'Color', cR_filt, 'DisplayName', 'R(\theta) filtered');
253     else
254         plot(R_trimmed(:,1), R_trimmed(:,2), '-', 'LineWidth', 1.8,
            ...
255             'Color', cR_raw, 'DisplayName', 'R(\theta) raw');
256     end
257     yline(R_center1, ':', sprintf('R_{ref}=%.3f \\Omega',R_center1),
          ...
258         'HandleVisibility','off', 'Color', tone(cR_raw,0.6));
259     xlabel('\theta (deg)'); ylabel('R(\theta) [\Omega]');
260     title('Figure 6     R(\theta) (single curve)'); grid on; legend(
        'Location','best');
261     if all(isfinite(theta_xlim)), xlim(theta_xlim); else,
        safe_xlim_from_vec(theta_out(idx_theta_grid)); end
262     safe_ylim_centerpad(R_center1, R_pad1);
263 end
264
265 % --- Figure 7: R vs Time (zoomed only) ---
266 if ~isempty(t) && (~isempty(R_theta_out) || ~isempty(
    R_theta_filt_out))
267     R_time = []; R_label = '';
268     if ~isempty(R_theta_filt_out), R_time = R_theta_filt_out(:);
        R_label = 'R(t) filtered';
269     elseif ~isempty(R_theta_out),  R_time = R_theta_out(:);
        R_label = 'R(t) raw';
270     end
```

```
271     if ~isempty(R_time)
272         nR = min(numel(t), numel(R_time));
273         if nR > 1
274             tR = t(1:nR); rY = R_time(1:nR);
275             figure; plot(tR, rY, '-', 'LineWidth', 1.8, ...
276                 'Color', cR_raw, 'DisplayName', R_label); hold on;
277             yline(R_center_t, ':', sprintf('R_{ref}=%.3f \\Omega',
                    R_center_t), ...
278                 'HandleVisibility','off', 'Color', tone(cR_raw,0.6))
                        ;
279             xlabel('Time [s]'); ylabel('R [\Omega]');
280             title(sprintf('Figure 7     R vs Time (zoom %.4 f  %.4f
                    s)', t_zoom(1), t_zoom(2)));
281             grid on; legend('Location','best');
282             safe_ylim_centerpad(R_center_t, R_pad_t);
283             if numel(t_zoom)==2 && all(isfinite(t_zoom)) && t_zoom
                    (1) < t_zoom(2)
284                 xlim(t_zoom);
285             else
286                 safe_xlim_from_vec(tR);
287             end
288         end
289     end
290 end
291
292 % --- Figure 8: duty vs    (BROWN, ylim [0,0.6]) ---
293 if ~isempty(DUTY_win)
294     figure;
295     plot(theta_time_win, DUTY_win, '-', 'LineWidth', 1.8, ...
296         'Color', cD, 'DisplayName', 'duty(\theta)');
297     xlabel('\theta (deg)'); ylabel('Duty [-]');
298     title('Figure 8     duty vs \theta'); grid on; legend('Location'
        ,'best');
```

```
    if all(isfinite(theta_xlim)), xlim(theta_xlim); else,
        safe_xlim_from_vec(theta_time_win); end
    ylim([0 0.6]);
end

% --- Figure 9: \phi_raw (deg) vs    (PURPLE) ---
if ~isempty(PHI_win_deg)
    figure;
    plot(theta_time_win, PHI_win_deg, '-', 'LineWidth', 1.8, ...
        'Color', cE, 'DisplayName', '\phi_{\rm raw}(\theta) [deg]');
    xlabel('\theta (deg)'); ylabel('\phi_{\rm raw} [deg]');
    title('Figure 9     \phi_{\rm raw} (deg) vs \theta'); grid on;
        legend('Location','best');
    if all(isfinite(theta_xlim)), xlim(theta_xlim); else,
        safe_xlim_from_vec(theta_time_win); end
end

% --- Figure 10: \omega vs Time (GREEN, zoom 300 5 , ref label on
    LEFT) ---
if ~isempty(t) && ~isempty(omega_ts)
    nW = min(numel(t), numel(omega_ts));
    if nW > 1
        tW = t(1:nW); oY = omega_ts(1:nW);
        figure; hold on;
        plot(tW, oY, '-', 'LineWidth', 1.8, 'Color', cV, '
            DisplayName', '\omega(t)');
        xlabel('Time [s]'); ylabel('\omega [rad/s]');
        title('Figure 10     \omega vs Time'); grid on; legend('
            Location','best');
        safe_xlim_from_vec(tW);
        ylim([omega_center - omega_pad, omega_center + omega_pad]);
            % 300    5

```

```
325         % y reference line and left-side label
326         yline(omega_center, ':', 'HandleVisibility','off', 'Color',
                tone(cV,0.6));
327         xl = xlim;
328         text(xl(1) + 0.02*(xl(2)-xl(1)), omega_center + 0.3, ...
329              sprintf('\\omega_{ref}=%d', omega_center), ...
330              'Color', tone(cV,0.8), 'FontSize', 10, '
                    HorizontalAlignment', 'left');
331     end
332 end
333
334 % --- Extract scalars from simOut (last finite sample) ---
335 a_sim = scalar_lastfinite(extractSignal(simOut,'a_out'));
336 b_sim = scalar_lastfinite(extractSignal(simOut,'b_out'));
337 c_sim = scalar_lastfinite(extractSignal(simOut,'c_out'));
338 g_sim = scalar_lastfinite(extractSignal(simOut,'g_out'));
339 L_sim = scalar_lastfinite(extractSignal(simOut,'L_out'));
340
341 % --- Assigned base parameters ---
342 R_assign   = 0.06;          % ohm
343 L_assign   = 0.6e-4;         % H
344 kv_assign  = 0.12;          % V/(rad/s)  (SI => k_t = k_v)
345 J_assign   = 0.008;           % kg*m^2
346 lam_assign = 0.002;          % N*m*s/rad
347
348 % --- Theory with corrected mapping ---
349 kt = kv_assign;                 % SI: k_t = k_v
350 a_theory = kt   / J_assign;     % 3.72
351 b_theory = lam_assign / J_assign; % 0.10
352 c_theory = kv_assign;           % 0.186
353 g_theory = R_assign;            % 0.020
354 L_theory = L_assign;            % 1.0e-4
355
```

```
356 names   = {'a (=k_t/J)','b (=  /J)','c (=k_v)','g (=R)','L'}';
357 theory  = [a_theory; b_theory; c_theory; g_theory; L_theory];
358 simvals = [a_sim;    b_sim;    c_sim;    g_sim;    L_sim];
359
360 abs_err = simvals - theory;
361 pct_err = 100 * (simvals - theory) ./ max(abs(theory), eps);
362
363 fprintf('\n=== Coefficient Comparison (sim vs theory; corrected
       mapping) ===\n');
364 fprintf('%10s | %14s | %14s | %14s | %9s\n', 'var','theory','sim','
       abs error','% error');
365 fprintf('%s\n', repmat('-',1,78));
366 for k = 1:numel(names)
367     t  = theory(k);
368     s  = simvals(k);
369     ae = abs_err(k);
370     pe = pct_err(k);
371     if isnan(s)
372         fprintf('%10s | %14.6g | %14s | %14s | %9s\n', names{k}, t,
              'NaN', 'NaN', 'NaN');
373     else
374         fprintf('%10s | %14.6g | %14.6g | %14.6g | %8.3f%%\n', names
              {k}, t, s, ae, pe);
375     end
376 end
377 fprintf('%s\n\n', repmat('-',1,78));
378
379 % ================================================================
380 % Helper functions
381 % ================================================================
382
383 function v = scalar_lastfinite(x)
384     v = NaN;
```

```
    if isempty(x), return; end
    x = double(x(:));
    x = x(isfinite(x));
    if ~isempty(x), v = x(end); end
end

function A = extractSignal(simOut, fieldName)
    A = [];
    try
        hasProp = (isobject(simOut) && isprop(simOut, fieldName));
        hasFld  = (~isobject(simOut) && isstruct(simOut) && isfield(
            simOut, fieldName));
        if ~(hasProp || hasFld), return; end
        x = simOut.(fieldName);
        A = toArraySafe(x);
    catch ME
        warning('extractSignal: %s (%s)', fieldName, ME.message);
        A = [];
    end
end

function A = firstNonEmpty(varargin)
    A = [];
    for k = 1:nargin
        x = varargin{k};
        if ~isempty(x), A = x; return; end
    end
end

function deg = coerceDegrees(vec)
    deg = [];
    if isempty(vec), return; end
    v = double(vec(:));
```

```
417     vmax = max(abs(v));
418     if ~isfinite(vmax), deg = []; return; end
419     if vmax <= 2*pi*1.25
420         deg = v * (180/pi);   % looks like radians
421     else
422         deg = v;              % already in degrees
423     end
424 end
425 
426 function A = toArraySafe(x)
427     if isempty(x), A = []; return; end
428     if isnumeric(x), A = double(x); return; end
429     if isa(x,'timeseries'), A = double(x.Data); return; end
430     if istimetable(x)
431         try, A = double(x.Variables); catch, A = double(table2array(
              timetable2table(x))); end
432         return;
433     end
434     if isa(x,'Simulink.SimulationData.Dataset')
435         try
436             if x.numElements==1
437                 el = x{1};
438                 if isprop(el,'Values')
439                     vals = el.Values;
440                     if isa(vals,'timeseries'), A = double(vals.Data)
                          ; else, A = double(vals); end
441                     return;
442                 end
443             end
444         catch, end
445     end
446     if isstruct(x)
447         if isfield(x,'signals') && isfield(x.signals,'values'), A =
```

```
            double(x.signals.values); return; end
        if isfield(x,'values'), A = double(x.values); return; end
    end
    try, A = double(x); catch, A = []; end
end

function TR = local_trim_theta_grid(sig, theta_grid, idx_rng)
    if isempty(sig) || isempty(theta_grid) || isempty(idx_rng)
        TR = []; return;
    end
    nSig = numel(sig); nTh = numel(theta_grid);
    idx_ok = idx_rng(idx_rng <= nSig & idx_rng <= nTh);
    if isempty(idx_ok), TR = []; return; end
    TR = [theta_grid(idx_ok).', sig(idx_ok).'];
end

function idx = nearest_index(vec, val)
    if isempty(vec), idx = []; return; end
    tol = 1e-9;
    match = find(abs(vec - val) <= tol, 1, 'first');
    if ~isempty(match), idx = match; return; end
    [~, idx] = min(abs(vec - val));
end

function v = slice_safe(sig, idxs)
    if isempty(sig) || isempty(idxs), v = []; return; end
    n = numel(sig);
    idxs = idxs(idxs >= 1 & idxs <= n);
    if isempty(idxs), v = []; else, v = sig(idxs); end
end

function safe_xlim_from_vec(xvec)
    if isempty(xvec), return; end
```

```
    x = xvec(:); x = x(isfinite(x));
    if numel(x) >= 2
        x1 = x(1); x2 = x(end);
        if x1 < x2, xlim([x1 x2]); end
    end
end

function safe_ylim_centerpad(center, pad)
    if isfinite(center) && isfinite(pad) && pad > 0
        ylim([center - pad, center + pad]);
    end
end
```

Listing 4.8: *MATLAB Plots*

Vita

Seth Mickley was born on September 29, 1998, to Beth and Kerry Mickley in Bethlehem, Pennsylvania, and has lived in the Lehigh Valley region near Lehigh University for most of his life.

He attended a local high school that went bankrupt during his enrollment, leading to a delayed start to his college career. Seth began his undergraduate studies in electrical engineering at Penn State Lehigh Valley, where he spent two years before transferring to the main campus at University Park in 2017. He completed his Bachelor of Science in Electrical Engineering in 2021 and began his Master of Science in Electrical Engineering at Lehigh University in 2023.

His fascination with hands-on electronics began in 2017 at Penn State Lehigh Valley, where open laboratory access encouraged him to experiment beyond the classroom. These experiences sparked his passion for practical engineering and problem-solving. During his undergraduate capstone project, he designed and built a battery spot welder, which further deepened his interest in circuit design and control systems.

After earning his bachelor's degree, Seth worked for two years as a control engineer before returning to graduate school to pursue deeper study in the areas he found most engaging during his undergraduate work. His interest in experimentation led him to investigate challenges he observed in industry, ultimately inspiring his research on adaptive control and predictive maintenance for brushed DC motors. At Lehigh, he met Professor Javad Khazaei, who encouraged him to pursue independent research and helped refine his thesis topic on motor control design.

Seth looks forward to a professional career in motor-drive and control system design and hopes to continue exploring the intersection of practical experimentation and theoretical development. He also looks forward to starting a family and continuing to grow both personally and professionally in the years ahead.