TECHNICAL UNIVERSITY OF CRETE
School of Production Engineering and Management

Diploma Thesis

Control of a self-balancing two-wheeled vehicle using computational intelligence methodologies

Dimitrios Cheiladakis

Chania, 2025

Contents

Abstract.....4

Chapter 1 - Introduction 11

1. Introduction 11

Chapter 2 – Literature Review 12

2. 1 Introduction 12

2.2 Fuzzy Logic (FL) 13

2.3 Genetic Algorithms (GA) 14

2.4 Neural Networks (NN) 16

2.5.1 From Perceptrons to Deep Networks 17

2.5.2 Forward Pass & Activations 17

2.5.3 Losses & Training 18

2.5.4 Regularization & Generalization 19

2.5.5 Architectures in Control 20

2.5.6 Neural Networks for Control 21

2.5.7 Data Preparation and Preprocessing 21

2.5.8 Deployment on Embedded Systems 21

2.5.9 Limitations and Risks 22

2.6 Inverse Control 22

Chapter 3 24

3.1 Introduction to the Arduino Nano 33 IoT 24

3.1.1 Hardware overview 24

3.1.2 On-board IMU and sensing 24

3.1.3 Real-time structure and timing 25

3.1.4 Power and signal integrity 25

3.1.5 Deploying learning-based controllers 26

3.1.6 Communications, logging, and updates 26

3.1.7 Pin mapping in this build 26

3.1.8 Rationale for selection 28

3.2 Experimental Kit 28

3.2.1 Hardware overview 28
3.2.2 Embedded sensing 29
3.2.3 Actuation and power 30
3.2.4 Real-time software architecture 30
3.2.5 Signal conditioning, calibration, and measurement fidelity 31
3.2.6 Safety and operating envelope 31
3.3 The Motorcycle Platform 32
3.3.1 Mechanical structure 32
3.3.2 Reaction-wheel drivetrain 33
3.3.3 Sensor suite and placement 34
3.3.4 Compute and motor interface 36
3.3.5 Power, wiring and integration details 36
3.3.6 Practical considerations and limitations 38
Chapter 4 40
4.1 Inverted-Pendulum Model Used in the Kit 40
4.1.1 Geometry, states, and modelling assumptions 40
4.1.2 Rotational dynamics and torques 41
4.1.3 State-space form used in simulation and code 42
4.2 System Architecture of the Self-Balancing Motorcycle 43
4.2.1 Hardware platform and purpose 43
4.2.2 Sensor suite and signal conditioning 44
4.2.3 Actuation chain (command to motor) 45
4.2.4 Real-time model architecture 46
4.2.5 Safety interlocks with State flow 46
4.2.6 Balance controller (PD + wheel-speed term) 47
4.2.7 Build, deployment, and tuning workflow 47
4.2.8 What each subsystem contained 48
4.2.9 How the model reached stable balance 49
4.3 Learning a Neural Controller from the PD Baseline and Deploying it in Both Models 50
4.3.1 Problem framing and interface 50
4.3.2 Demonstration generation in the inverted-pendulum model 50

4.3.3 Demonstration generation in the Simscape physical model 51
4.3.4 Curation, synchronization, and normalization 51
4.3.5 Network architecture and output parameterization 52
4.3.6 Closed-loop validation in both simulators 52
4.3.7 Capturing and Exporting Hardware Data with “Save Runs” 53
4.3.8 Deployment pathway and on-target checks 54
Chapter 5 56
Experimental results 56
5.1.1 Inverted-pendulum closed loop with a Feed-Forward Net (10) 56
5.1.2 Feedforwardnet(20): inverted-pendulum results 59
5.1.3 Inverted pendulum with feedforwardnet(30) 61
5.2.1 Physical model — NN controller (fitnet [12 12]) 62
5.2.2 Physical model — fitnet[15 15] 64
5.2.3 Physical model — NN controller fitnet[20,20] 66
5.3 Fuzzy-logic controllers: attempts, outcomes, and guidance 68
5.4 Conclusion 70
Bibliography 73

Table of Contents

Figure 1: Example of fuzzy membership functions 14
Figure 2: Flowchart of the Genetic Algorithm process. 15
Figure 3: Example of a feed-forward neural network with two hidden layers 16
Figure 4: A simple perceptron model 17
Figure 5: A simple feedforward neural network with one hidden layer 18
Figure 6: Common activation functions used in neural networks. 18
Figure 7: Illustration of forward pass and backpropagation in a neural network.. 19
Figure 8: Example of training and validation accuracy during neural network training. . 20
Figure 9: Comparison between traditional machine learning and deep learning. 20
Figure 10: Workflow for deploying a trained neural network on embedded hardware ... 22
Figure 11: Block diagram of inverse control using a neural network. 23
Figure 12: Block diagram of a classical feedback control system. 23
Figure 13: Arduino Nano 33 IoT board 24
Figure 14: IMU axes and rotation conventions on the Arduino Nano 33 IoT 25

Figure 15: Power tree of the Arduino Nano 33 IoT. 26
Figure 16: Arduino Nano 33 IoT main-board schematic (overview). 27
Figure 17: Arduino Nano 33 IoT pinout. 27
Figure 18: Experimental kit overview. 28
Figure 19: Location of the on-board IMU (LSM6DS3) on the Arduino Nano 33 29
Figure 20: Inertia-wheel DC motor with integrated Hall-effect. 30
Figure 21: The digital controller computes the control command from the sensor feedback provided by the motorcycle subsystem. 31
Figure 22: Sensor conditioning and signal preparation inside the digital controller. 31
Figure 23: Digital controller. 32
Figure 24: Detail of the electronics and actuator stack. 33
Figure 25: Inertia-wheel drive with integrated incremental encoder. 34
Figure 26: Inertia-wheel speed ω during balancing experiment 34
Figure 27: Kinematic variables measured on the prototype. 35
Figure 28: IMU self-calibration status shown in Simulink. 35
Figure 29: Safety and enabling logic in Simulink 36
Figure 30: Battery Read signal chain in Simulink. 37
Figure 31: Raw sensor aggregation (“rawSensors” bus). 37
Figure 32: Motorcycle CAD with actuator/sensor locations. 38
Figure 33: Closed-loop balance at standstill. 39
Figure 34: Definition of torque about the rotation axis; $\tau = rF\sin\theta$. (4) 40
Figure 35: Solid cylinder used to model the rod 41
Figure 36: Torque & inverted pendulum torques (composite). 42
Figure 37: State-space realization of the inverted-pendulum model in Simulink. 42
Figure 38: Top-level subsystem with defined I/O for controller-in-the-loop work. 43
Figure 39: modelParameters.m script parameterizes the simulation. 43
Figure 40: Top-level closed-loop architecture (controller ↔ plant). 44
Figure 41: Self-Balancing Motorcycle kit components prior to assembly 45
Figure 42: Sensor pre-processing in Simulink 45
Figure 43: Safety and enable logic. 46
Figure 44: State flow safety supervisor for the self-balancing motorcycle 47
Figure 45: Balance controller (PD + wheel-speed feedback) implemented in Simulink. 48
Figure 46: Hardware I/O integration in the Motorcycle subsystem (Simulink). 49
Figure 47: Hardware closed-loop logs in External Mode 49
Figure 48: NN-in-the-loop inverted pendulum 50
Figure 49: Controller MLP architecture (3–10–1 with ReLU). 51
Figure 50: MATLAB script for preparing logs and training the controller MLP. 51
Figure 51: NN controller wired into the physical model: input conditioning, feed-forward network, and command shaping. 52
Figure 52: Function-fitting neural network used in the physical model. 52
Figure 53: Sample Code for save_runs. 54

Figure 54: Batch training from physical-model logs..... 55
Figure 55:Simscape visualization of the stabilized inverted-pendulum..... 57
Figure 56: Inverted-pendulum closed-loop response with a feed-forward neural controller (feedforwardnet(10)). 57
Figure 57: Training summary for the 10-neuron feed-forward controller. 58
Figure 58: Feedforwardnet(20) training report—LM optimizer, early stopping on validation 59
Figure 59: Inverted-pendulum response with Feedforwardnet(20) 60
Figure 60: Training summary for feedforwardnet(30). 61
Figure 61: Closed-loop response with the trained NN controller. 62
Figure 62: MATLAB Neural Network Training report for the physical-model controller (fitnet([12 12])). 63
Figure 63: Hardware-structured model logs 64
Figure 64: A. Neural Network Training report for fitnet[15 15] on physical-model data. 65
Figure 65: External-Mode logs with fitnet[15 15] in the loop 66
Figure 66: Training summary for the physical-model network fitnet[20,20] (Levenberg–Marquardt, MSE loss). 67
Figure 67: Closed-loop logs on the physical model with fitnet[20,20]. 68
Figure 68: Single Mamdani (type-2) fuzzy controller with three inputs..... 69
Figure 69: Two-stage Mamdani (type-2) fuzzy-logic “tree” for the inverted-pendulum Simscape model.. 69

Abstract

Balancing a two-wheeled vehicle is a problem that has drawn attention from both researchers and industry. These systems are inherently unstable—if no corrective action is applied, they tip over almost immediately. Because of this, they are often compared to the well-known inverted pendulum problem, a benchmark in control engineering [1]. Beyond being a theoretical challenge, however, technology has very practical uses. It underpins personal mobility devices such as the Segway, appears in service and warehouse robots, and is being explored in assistive devices that can enhance mobility for people with physical limitations. This combination of practical relevance and theoretical difficulty makes the study of self-balancing vehicles particularly appealing.

A striking feature of such vehicles is how directly they reveal the effectiveness of a control strategy. When the controller works, the vehicle remains upright; when it fails, the vehicle quickly falls. This immediacy makes them not only good case studies for advanced control research but also valuable educational tools. At the same time, it highlights the difficulty of the problem: controllers must process real-time sensor inputs—often from accelerometers and gyroscopes—while compensating for noise, drift, and delays. These real-world issues mean that approaches which perform well in theory do not always succeed in practice.

The dynamics of a self-balancing vehicle are nonlinear and highly sensitive to disturbance. A small irregularity in the ground, a push from the side, or a shift in load can destabilize it. Controllers must therefore adjust continuously, updating the torque applied by the motors in fractions of a second. Classical approaches such as PID control, state feedback methods, and Linear Quadratic Regulators (LQR) have all been applied successfully to this challenge. However, these approaches tend to rely on accurate mathematical models of the system. In reality, models are never perfect, and performance can deteriorate once uncertainties and nonlinear effects come into play [22].

Another practical issue with classical controllers is the difficulty of parameter tuning. For instance, a PID controller requires carefully chosen gains, but values that work well under one condition may fail when conditions change. Similarly, LQR designs depend on weight matrices that reflect trade-offs between competing goals such as stability and energy use. In practice, these values are often found through trial and error, which is time-consuming and sometimes unreliable. This dependence on tuning motivates the search for alternatives that can cope with variation more naturally.

One promising direction is the use of computational intelligence techniques. Approaches such as fuzzy logic control, neural networks, and neuro-fuzzy systems are attractive because they do not require a precise model of the system. Fuzzy logic makes it possible to describe control rules in intuitive, human-like terms—for example, “if the tilt is small but increasing quickly, apply a strong correction.” Neural networks, in contrast, excel at learning from data and can capture complex nonlinear relationships that are hard to model directly. A hybrid neuro-fuzzy system can combine these advantages, offering both interpretability and adaptability.

The strength of these approaches lies in their robustness and flexibility. Unlike conventional controllers, they can adapt to unexpected conditions and cope with unmodeled dynamics, sensor noise, or changes in the environment. For a self-balancing vehicle, this can translate

into better stability on rough terrain, under varying loads, or when disturbances are applied. Beyond this particular application, similar methods have been successfully used in domains such as autonomous driving, robotics, and unmanned aerial vehicles, showing that the underlying ideas are widely applicable.

This thesis investigates these methods in the context of a self-balancing two-wheeled vehicle. Its primary goal is to design, implement, and compare controllers based on both classical control and computational intelligence. A lab-scale prototype built with the Arduino Engineering Kit Rev 2 and the Arduino Nano 33 IoT is used as the experimental platform. This makes it possible to test different approaches under controlled conditions and evaluate their relative strengths and weaknesses. The research aims to answer a central question: can computational intelligence methods provide a practical advantage over traditional controllers in stabilizing self-balancing vehicles?

ΠΕΡΙΛΗΨΗ

Η ισορρόπηση ενός δίκυκλου οχήματος είναι ένα πρόβλημα που έχει προσελκύσει το ενδιαφέρον τόσο της έρευνας όσο και της βιομηχανίας. Τα συστήματα αυτά είναι εγγενώς ασταθή, αν δεν εφαρμοστεί διορθωτική δράση, ανατρέπονται σχεδόν αμέσως. Γι' αυτό συχνά συγκρίνονται με το γνωστό πρόβλημα του ανεστραμμένου εκκρεμούς, ένα σημείο αναφοράς στην επιστήμη του ελέγχου. Πέρα όμως από τη θεωρητική πρόκληση, η τεχνολογία έχει αρκετές πρακτικές χρήσεις. Βρίσκεται πίσω από προσωπικές συσκευές μετακίνησης όπως το Segway, εμφανίζεται σε ρομπότ εξυπηρέτησης και αποθηκών και εξερευνάται σε βοηθητικές διατάξεις που μπορούν να ενισχύσουν την κινητικότητα ατόμων με σωματικούς περιορισμούς. Αυτός ο συνδυασμός πρακτικής σημασίας και θεωρητικής δυσκολίας καθιστά τη μελέτη των αυτοεξισορροπούμενων οχημάτων ιδιαίτερα ελκυστική.

Ένα εντυπωσιακό χαρακτηριστικό τέτοιων οχημάτων είναι πόσο άμεσα αποκαλύπτουν την αποτελεσματικότητα μιας στρατηγικής ελέγχου. Όταν ο ελεγκτής λειτουργεί, το όχημα μένει όρθιο ενώ όταν αποτυγχάνει, πέφτει γρήγορα. Αυτή η αμεσότητα τα καθιστά όχι μόνο άριστες μελέτες περίπτωσης για προηγμένη έρευνα στον έλεγχο, αλλά και πολύτιμα εκπαιδευτικά εργαλεία. Ταυτόχρονα, αναδεικνύει τη δυσκολία του προβλήματος: οι ελεγκτές πρέπει να επεξεργάζονται σε πραγματικό χρόνο αισθητήρες εισόδου—συχνά από επιταχυνσιόμετρα και γυροσκόπια—ενώ αντισταθμίζουν θόρυβο στις μετρήσεις, μετατόπιση (drift) και καθυστερήσεις. Αυτές οι πραγματικές συνθήκες σημαίνουν ότι προσεγγίσεις που αποδίδουν καλά στη θεωρία δεν επιτυγχάνουν πάντα στην πράξη.

Η δυναμική ενός αυτοεξισορροπούμενου οχήματος είναι μη γραμμική και εξαιρετικά ευαίσθητη σε διαταράξεις. Μια μικρή ανωμαλία στο έδαφος, ένα πλάγιο σπρώξιμο ή μια μεταβολή φορτίου μπορεί να το αποσταθεροποιήσει. Οι ελεγκτές, επομένως, πρέπει να προσαρμόζονται συνεχώς, ενημερώνοντας τη ροπή που εφαρμόζουν οι κινητήρες σε κλάσματα του δευτερολέπτου. Κλασικές προσεγγίσεις όπως ο PID έλεγχος, οι μέθοδοι ανατροφοδότησης κατάστασης και οι Γραμμικοί Τετραγωνικοί Ρυθμιστές (LQR) έχουν εφαρμοστεί με επιτυχία σε αυτήν την πρόκληση. Ωστόσο, αυτές οι προσεγγίσεις τείνουν να στηρίζονται σε ακριβή μαθηματικά μοντέλα του συστήματος. Στην πραγματικότητα, τα μοντέλα δεν είναι ποτέ τέλεια, και η απόδοση μπορεί να υποβαθμιστεί όταν εμφανίζονται αβεβαιότητες και μη γραμμικά φαινόμενα.

Ένα ακόμη πρακτικό ζήτημα με τους κλασικούς ελεγκτές είναι η δυσκολία ρύθμισης των παραμέτρων. Για παράδειγμα, ένας ελεγκτής PID απαιτεί προσεκτικά επιλεγμένα κέρδη, αλλά τιμές που λειτουργούν καλά σε μια συνθήκη μπορεί να αποτύχουν όταν οι συνθήκες αλλάξουν. Παρομοίως, οι σχεδιάσεις LQR εξαρτώνται από πίνακες βαρών που αντικατοπτρίζουν αντισταθμίσεις μεταξύ ανταγωνιστικών στόχων όπως η σταθερότητα και η κατανάλωση ενέργειας. Στην πράξη, αυτές οι τιμές βρίσκονται συχνά με δοκιμή και σφάλμα, κάτι που είναι χρονοβόρο και ενίοτε αναξιόπιστο. Αυτή η εξάρτηση από τη ρύθμιση παραμέτρων υποκινεί την αναζήτηση εναλλακτικών μεθόδων που μπορούν να αντιμετωπίζουν πιο φυσικά τη μεταβλητότητα.

Μια πολλά υποσχόμενη κατεύθυνση είναι η χρήση τεχνικών υπολογιστικής νοημοσύνης. Προσεγγίσεις όπως Fuzzy Logic, Neural Networks και τα Neuro-Fuzzy συστήματα είναι ελκυστικές επειδή δεν απαιτούν ακριβές μοντέλο του συστήματος. Το Fuzzy Logic επιτρέπει τη διατύπωση κανόνων ελέγχου με διαισθητικούς, ανθρωποκεντρικούς όρους—για

παράδειγμα, «αν η κλίση είναι μικρή αλλά αυξάνεται γρήγορα, εφάρμοσε ισχυρή διόρθωση». Τα Neural Networks, αντίθετα, υπερέχουν στη μάθηση από δεδομένα και μπορούν να αποτυπώσουν σύνθετες μη γραμμικές σχέσεις που είναι δύσκολο να μοντελοποιηθούν άμεσα. Ένα υβριδικό νευρο-ασαφές σύστημα μπορεί να συνδυάσει αυτά τα πλεονεκτήματα, προσφέροντας τόσο ερμηνευσιμότητα όσο και προσαρμοστικότητα.

Η ισχύς αυτών των προσεγγίσεων, έγκειται στην ευρωστία και την ευελιξία τους. Σε αντίθεση με τους συμβατικούς ελεγκτές, μπορούν να προσαρμόζονται σε απρόσμενες συνθήκες και να αντεπεξέρχονται σε αμοντελοποίητη δυναμική, θόρυβο αισθητήρων ή αλλαγές στο περιβάλλον. Για ένα αυτοεξισορροπούμενο όχημα, αυτό μπορεί να μεταφραστεί σε καλύτερη σταθερότητα σε τραχύ έδαφος, υπό μεταβαλλόμενα φορτία ή όταν εφαρμόζονται διαταράξεις. Πέρα από αυτήν τη συγκεκριμένη εφαρμογή, παρόμοιες μέθοδοι έχουν χρησιμοποιηθεί επιτυχώς σε τομείς όπως η αυτόνομη οδήγηση, η ρομποτική και τα μη επανδρωμένα ιπτάμενα οχήματα, δείχνοντας ότι οι υποκείμενες ιδέες έχουν ευρεία εφαρμοσιμότητα.

Η παρούσα διπλωματική εργασία διερευνά αυτές τις μεθόδους στο πλαίσιο ενός αυτοεξισορροπούμενου δίκυκλου οχήματος. Κύριος στόχος της είναι ο σχεδιασμός, η υλοποίηση και η σύγκριση ελεγκτών βασισμένων τόσο στον κλασικό έλεγχο όσο και στην υπολογιστική νοημοσύνη. Ένα πρωτότυπο εργαστηριακής κλίμακας, κατασκευασμένο με το Arduino Engineering Kit Rev 2 και το Arduino Nano 33 IoT, χρησιμοποιείται ως πειραματική πλατφόρμα. Αυτό καθιστά δυνατή τη δοκιμή διαφορετικών προσεγγίσεων υπό ελεγχόμενες συνθήκες και την αξιολόγηση των σχετικών πλεονεκτημάτων και μειονεκτημάτων τους. Η έρευνα στοχεύει να απαντήσει σε ένα κεντρικό ερώτημα: μπορούν οι μέθοδοι υπολογιστικής νοημοσύνης να προσφέρουν πρακτικό πλεονέκτημα έναντι των κλασικών ελεγκτών στη σταθεροποίηση αυτοεξισορροπούμενων οχημάτων;

Chapter 1 - Introduction

1. Introduction

This thesis investigates how a small, two-wheeled self-balancing vehicle can be kept upright under real-world disturbances, and whether learning-based controllers can match (or even surpass) classic feedback laws. The core idea is simple to state and hard to execute: treat the motorcycle as an inverted pendulum with a reaction (inertia) wheel, sense its lean with an IMU, and command just enough torque to bring it back to the vertical. What follows takes that idea from equations to a working lab-scale prototype, with the control logic first proven in simulation and then deployed on the Arduino Nano 33 IoT.

The workflow blends classical control with computational intelligence. A proportional–derivative (PD) controller is used as a reliable baseline and as a source to generate training data. The plant is modeled in two complementary ways: a tractable state-space representation for analysis, and a geometry-aware Simscape Multibody model that captures the hardware's inertial properties. With those in place, compact neural networks are trained to imitate the PD policy from measured states [θ, $\dot{\theta}$, ω] and are then dropped into the loop in place of the PD block—first in simulation, then on the physical kit. Throughout, practical safeguards (Stateflow enable logic, motor saturation, and wheel-speed limiting) keep the experiments safe and repeatable.

The contribution is twofold. On the modeling and implementation side, the work assembles a complete, reproducible pipeline: from deriving and validating the pendulum dynamics, to building the Simulink/Simscape models, to interfacing the Nano's sensors and actuators in External Mode. On the control side, it shows that small feed-forward networks can compress a hand-tuned PD policy into a lightweight function that runs on microcontroller hardware, while documenting where the sim-to-real gap still appears and how safety interlocks and data-collection choices affect performance.

The document is organized as follows. Chapter 2 reviews the control methods used here, covering classic designs and computational-intelligence tools (fuzzy logic, neural networks, and genetic algorithms). Chapter 3 introduces the hardware platform—the Arduino Nano 33 IoT and the Engineering Kit—and describes the sensors, actuators, and mechanical layout. Chapter 4 develops the control framework: the inverted-pendulum model, the integration with the physical motorcycle, and the learning pipeline that trains and deploys neural controllers. Chapter 5 presents the experimental results in simulation and on hardware, including the PD baseline, the neural-network replacements, and brief notes on fuzzy-logic attempts. Chapter 6 closes with a summary of findings and pointers for future work.

Chapter 2 – Literature Review

2. 1 Introduction

Before attempting to design and implement controllers for a self-balancing vehicle, it is important to understand the main tools available in the field of computational intelligence, since this is the approach that we will implement in the proposed thesis. Unlike classical control, which relies heavily on mathematical models and analytical design, computational intelligence focuses on methods that can learn, adapt, or make decisions under uncertainty. This chapter introduces three of the most used approaches: *Fuzzy Logic (FL), Neural Networks (NN), and Genetic Algorithms (GA) [3]*. Each of these methodologies has been developed based on different inspirations - human reasoning, biological neurons, and natural evolution -but all share the goal of providing flexible and robust solutions to complex problems. By reviewing their principles, strengths, and limitations, this chapter establishes the foundation which justifies, why they are suitable candidates for the control of self-balancing vehicles.

The inverted pendulum has long served as a canonical benchmark for nonlinear control and as a compact vehicle for introducing modern feedback theory. Classical derivations, whether by Newton–Euler or Lagrange formulations, reduce the dynamics to a state-space description in which the lean angle and its angular rate constitute the essential states. Among the many variants of the archetype, the reaction-wheel pendulum - the configuration most analogous to a self-balancing motorcycle - stabilizes its upright equilibrium by spinning a high-inertia flywheel to generate corrective torque about the body axis. Because the equilibrium is open-loop unstable and the actuation is constrained, this family of systems is routinely used to compare linear controllers with intelligent or learning-based methods.

Traditional feedback designs remain attractive due to their transparency and modest implementation cost [5]. Proportional-derivative or proportional-integral-derivative controllers are typically tuned around a linearization of the upright equilibrium to achieve a desired rise time and damping ratio. State-space formulations enable optimal linear-quadratic regulation, which stabilizes the linearized model while explicitly trading tracking performance against control effort; for reaction-wheel platforms this often yields reduced steady spin at the cost of greater model dependence. In practical deployments the quality of angle and rate estimates is a dominant consideration, and observers or Kalman filters are used to fuse gyroscope and accelerometer measurements while attenuating bias and quantization [44].

Intelligent control techniques have also been widely explored for pendulum stabilization. Fuzzy logic controllers, especially Mamdani formulations, offer an interpretable rule-based mapping from linguistic statements to control actions. Their appeal lies in the ability to encode heuristic expertise and to tolerate imprecision; their weakness is the rapid growth in tuning complexity as the number of inputs, membership functions, and linguistic labels increases, which can lead to dense rule bases with hard-to-predict interactions and sensitivity to the chosen defuzzification method. Artificial neural networks have been used in two principal ways. In supervised control, a network is trained to imitate a baseline controller by learning the mapping from measured states to motor torque; this approach works well when the baseline is competent but imperfect, allowing the network to absorb mild nonlinearities and actuator dead-zones while preserving the baseline’s stability characteristics. In reinforcement learning, the

controller is learned directly from interaction with a model or the real plant; when reward shaping and safety constraints are well designed, such methods can outperform linear baselines, but they demand extensive simulation, careful exploration management, and conservative transfer to hardware.

Model-based design environments such as MATLAB/Simulink and Simscape Multibody have become the dominant workflow for moving from equations to embedded prototypes. A common pattern in the literature is to validate a first-principles model, synthesize and test controllers in simulation, and then transition to hardware-in-the-loop or external-mode experiments that capture sample-time effects, PWM quantization, motor saturation, and sensor-calibration realities such as those of low-cost IMUs. Safety interlocks—battery guards, calibration gates, and “fallen” detection—are consistently recommended before closing the loop on physical systems that can saturate actuators or topple.

A gap that persists across many reports is the end-to-end documentation of pipelines that begin with a physics-based model and culminate in embedded implementation on constrained microcontrollers [2]. Although PID and LQR designs are well characterized, fewer works compare supervised neural controllers trained on high-quality PD or LQR data against their linear teachers on higher-fidelity physical models that include sensing and actuation details. The present thesis positions itself within this space by deriving and validating an inverted-pendulum approximation of a self-balancing motorcycle, establishing a PD baseline, training compact feedforward networks to reproduce the baseline mapping, and then comparing behaviors in both a reaction-wheel SimScape model and embedded-style closed-loop experiments.

For citation coverage, this review should be grounded in standard textbooks and survey articles on inverted pendula and reaction-wheel systems, references on classical linear control and state-estimation methods, works on Mamdani fuzzy logic and its tuning challenges, studies of supervised and reinforcement-learning controllers for pendulum stabilization, and sources describing Simscape-based rapid prototyping and embedded deployment. Once the final bibliography is assembled, the narrative above can be interleaved with numbered citations in the order required by your thesis style.

2.2 Fuzzy Logic (FL)

Fuzzy Logic was first proposed by Lotfi Zadeh in the 1960s, as a way to handle uncertainty and approximate reasoning. Unlike traditional logic, which treats statements as either completely true or completely false, fuzzy logic allows for degrees of truth. In everyday life, people rarely think in black and white terms. For example, when saying “the road is slightly wet” or “the car is going fast,” these descriptions cannot be expressed precisely in numbers but still convey useful meaning. Fuzzy logic captures this kind of reasoning by using membership functions and linguistic rules [35].

In control systems, fuzzy logic is often applied through sets of if–then rules [36]. A typical rule might read: “If the tilt angle is small and the speed is increasing, then apply a moderate correction.” Instead of relying on exact equations, the controller interprets inputs in terms of fuzzy sets (e.g., “small,” “medium,” “large”) and produces outputs that blend the influence of multiple rules. This makes fuzzy logic controllers intuitive to design when expert knowledge is

available, and highly effective in situations where creating an accurate mathematical model is difficult [39].

Fuzzy sets allow elements to belong to more than one category with different degrees of membership. For example, a person with a certain weight might be considered partly 'average' and partly 'heavy'. Figure 1 shows an example of membership functions for the variable Weight, where categories such as thin, average, and heavy are defined using overlapping triangular and trapezoidal shapes. This overlap illustrates the gradual transition between linguistic terms, which is a key strength of fuzzy reasoning.

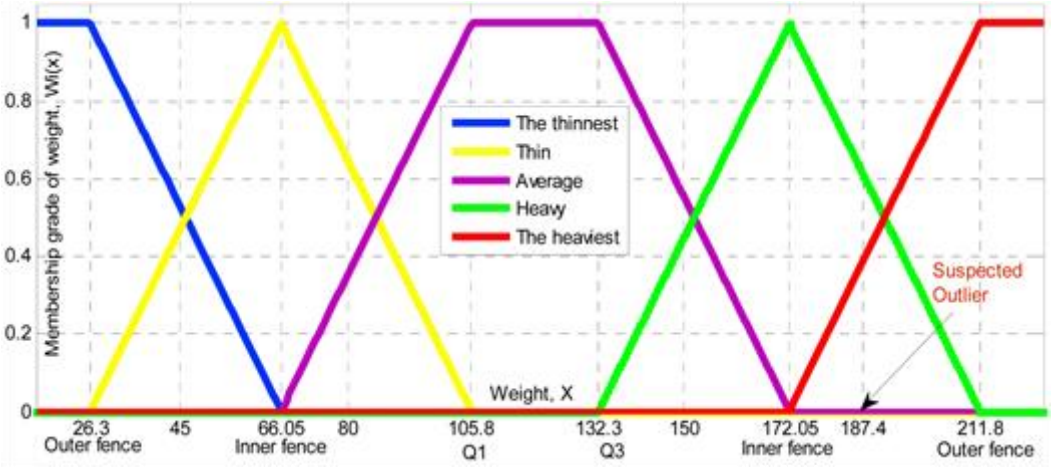


Figure 1: Example of fuzzy membership functions for the variable "Weight," showing overlapping categories (thin, average, heavy, etc.). [45]

The main advantage of fuzzy logic is its ability to handle uncertainty and nonlinearity without requiring a precise model of the system. It has been widely applied in real-world products—washing machines that adjust cycles based on load, cameras that automatically tune focus, and cars with adaptive braking or steering assistance. However, fuzzy systems are not without drawbacks. As the number of inputs and conditions increases, the number of rules can grow rapidly, creating what is known as the "rule explosion" problem. Furthermore, designing membership functions and tuning rule weights often depends on subjective judgment, which may limit reproducibility. Despite these challenges, fuzzy logic remains popular because it provides a balance between interpretability, simplicity, and practical effectiveness.

2.3 Genetic Algorithms (GA)

Genetic Algorithms (GAs) are a type of optimization method inspired by Darwin's principle of natural selection. Instead of trying to find the best solution in a single step, a GA begins with a population of possible solutions. Each solution is evaluated according to a fitness function, which measures how good it is relative to the problem at hand. The best solutions are then selected and combined using operations such as crossover (recombination of solutions) and mutation (small random changes). Over many generations, the population evolves toward better solutions. The overall process of a genetic algorithm is illustrated in Figure 2. The algorithm begins with an initial population, which is repeatedly evaluated and improved through selection, crossover, and mutation until a termination condition is met. The final output is a set of solutions that represent near-optimal candidates for the problem at hand.

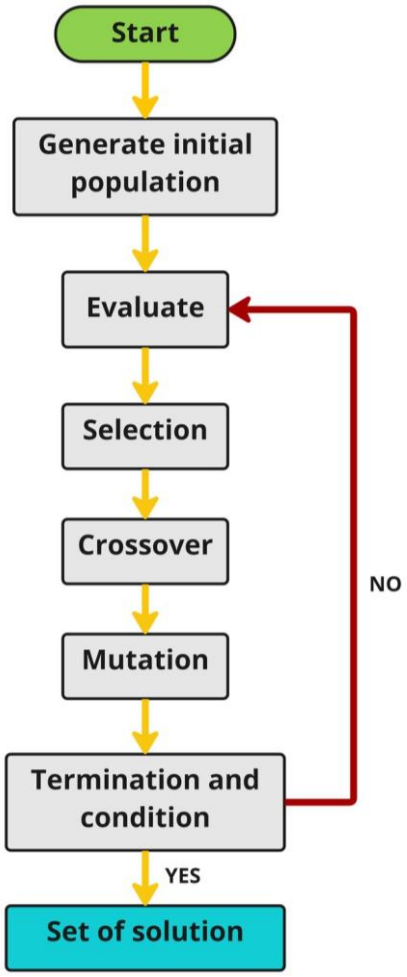


Figure 2: Flowchart of the Genetic Algorithm process, showing the iterative loop of evaluation, selection, crossover, and mutation until a termination condition is satisfied. [46]

In engineering, GAs are widely used for problems where the solution space is too large or too complex for traditional optimization methods [6]. For example, they can tune the parameters of a PID controller, search for the best membership functions in a fuzzy system or even design the architecture of a neural network. Their greatest strength is their ability to perform global search—they are less likely to get stuck in local optima compared to gradient-based methods. This makes them highly valuable when the problem landscape is rugged or nonlinear.

Nevertheless, GAs also comes with trade-offs. Because they rely on evaluating many potential solutions, they can be computationally expensive, particularly if each evaluation requires a long simulation or experiment. They also do not guarantee finding the exact global optimum; instead, they tend to converge on “good enough” solutions. Furthermore, the choice of parameters such as population size, crossover rate, and mutation rate can strongly affect performance, requiring careful experimentation. Even with these limitations, GAs remain a flexible and robust optimization tool, especially when combined with other methods in so-called “hybrid” intelligent systems.

2.4 Neural Networks (NN)

Artificial Neural Networks (ANNs) take their inspiration from the way biological neurons process information. A network consists of layers of artificial “neurons” connected by weighted links. Each neuron performs a simple operation, but when many of them are combined, the system can capture highly complex relationships between inputs and outputs. In control applications, this means that neural networks can learn how a system behaves directly from data, even if the underlying equations are unknown or too complicated to model.

In Figure 3 the structure of a typical feed-forward neural network is presented, where inputs are processed by multiple hidden layers before producing an output. By adjusting the weights of the connections, the network learns to approximate nonlinear input–output relationships.

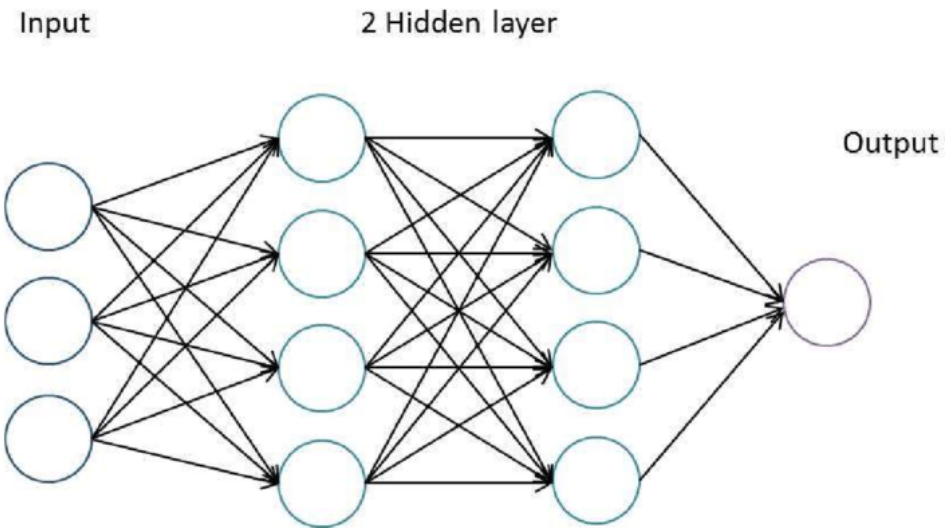


Figure 3: Example of a feed-forward neural network with two hidden layers. [47]

One of the defining features of neural networks is their ability to learn and adapt. A neural controller can be trained offline using experimental data, or it can be designed to update its parameters online as the system operates. For a self-balancing vehicle, this adaptability is appealing. Instead of relying on fixed equations, the controller can improve its performance as it gathers more experience. Neural networks have been successfully used in robotics, speech recognition, medical diagnosis, and many other areas where patterns must be extracted from noisy or nonlinear data.

Despite these strengths, neural networks are often criticized for being difficult to interpret. Unlike fuzzy logic, where the rules are transparent and human-readable, the knowledge in a neural network is stored in a large set of numerical weights. This makes it challenging to explain why a particular decision was made, which can be a concern in safety-critical systems. Training also requires a sufficient amount of data and computing power, and poor training can lead to overfitting, where the network performs well on training data but fails in real-world conditions. Even so, neural networks remain one of the most widely used computational intelligence tools, and their flexibility makes them especially attractive for complex control problems.

2.5.1 From Perceptrons to Deep Networks

The history of neural networks really begins with the perceptron, introduced by Frank Rosenblatt in the late 1950s. The model itself was not complicated. Inputs were multiplied by adjustable weights, a bias was added, and the result was passed through a threshold. If the value passed the threshold the output was one thing, and if not, it was another. On its own the perceptron could only solve linearly separable problems, which seems very restrictive by today's standards. But at the time it felt groundbreaking because, for the first time, a machine could "learn" by adjusting its parameters based on examples rather than being programmed step by step.

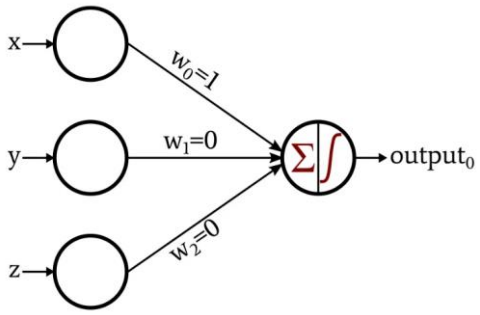


Figure 4: A simple perceptron model with three inputs (x, y, z), associated weights, and a summation/activation function producing a single output. [48]

The perceptron's limits became clear fairly quickly, but researchers also realized that linking several perceptrons together could make the system far more capable. This gave rise to the multilayer perceptron (MLP), which is still widely used today. Adding hidden layers allowed the network to model nonlinear functions that the original perceptron could never capture. In fact, later research showed that MLPs could approximate almost any continuous function, given enough neurons and training. That property gave them a solid theoretical foundation and made them attractive for handling the nonlinear dynamics often found in engineering.

Progress in the early years was slow, partly because the hardware available at the time wasn't up to the task of training big models. Things changed in the 2000s with faster processors, more data, and new algorithms. This period saw the rise of deep learning—networks with many hidden layers that could finally be trained effectively. These deeper models proved able to extract features at multiple levels of complexity. While they are often highlighted in image or speech recognition, the same ideas are valuable for control problems such as keeping a vehicle upright.

2.5.2 Forward Pass & Activations

When a neural network makes a prediction, it does so through what is called the forward pass. Information flows from the inputs, through the hidden layers, and finally to the outputs (see Figure 5). Each neuron applies a weighted sum to its inputs, adds a bias, and then pushes the result through an activation function [29]. That activation step is what introduces nonlinearity. Without it, no matter how many layers you had, the whole network would behave like a single linear mapping, which is rarely enough for real-world problems.

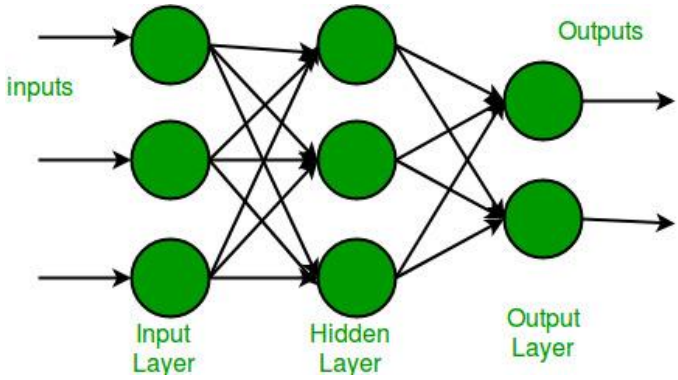


Figure 5: A simple feedforward neural network with one hidden layer. The forward pass consists of propagating the inputs through successive layers until the final output is generated. [49]

Different activation functions have been used over time. In the early days, sigmoid and tanh functions were popular because they produced smooth outputs between fixed ranges [30]. The drawback was that they often made training slow, since gradients tended to vanish. A major improvement came with the rectified linear unit (ReLU), which is extremely simple but highly effective: it outputs zero for negative inputs and grows linearly for positive ones. Variations of ReLU are now the default choice in most networks (see Figure 6). For control tasks, the activation used in the output layer depends on the problem. A linear output is typical when predicting a continuous control signal such as torque, while sigmoid or softmax is better suited for classification. In practice, hidden layers usually rely on ReLU, and the output is chosen to fit the task.

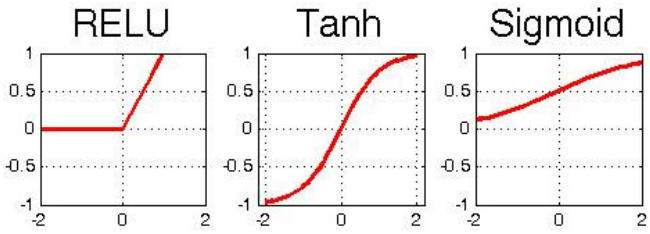


Figure 6: Common activation functions used in neural networks. Sigmoid and Tanh provide smooth nonlinear mappings but can suffer from vanishing gradients, while ReLU offers simplicity and faster convergence. [50]

2.5.3 Losses & Training

For a network to learn, it needs a way to measure how far off its guesses are from what they should be. That is exactly what a loss function does. It compares the prediction with the target and gives back a single number that represents the error. In control problems, the most common choice is mean squared error (MSE) because it makes large mistakes stand out more. In other words, the bigger the error, the harsher the penalty. For classification tasks, another option is cross-entropy, which works better since it looks at the differences between probability distributions.

Once the loss is defined, the challenge is to reduce it. This is where training comes in. One popular approach is the backpropagation algorithm, which calculates how much each weight in the network contributed to the error. Those values - the gradients - tell the optimizer how to adjust the weights in the next step. The most basic optimizer is stochastic gradient descent (SGD). Although it usually performs efficiently, it can be slow and sometimes unstable, especially in complex problems. Because of that, optimizers like Adam are widely used today. They adapt the learning rate automatically, so the process tends to be faster and more reliable.

In short, picking the right loss function and optimizer has a huge impact on whether a network learns well or just struggles.

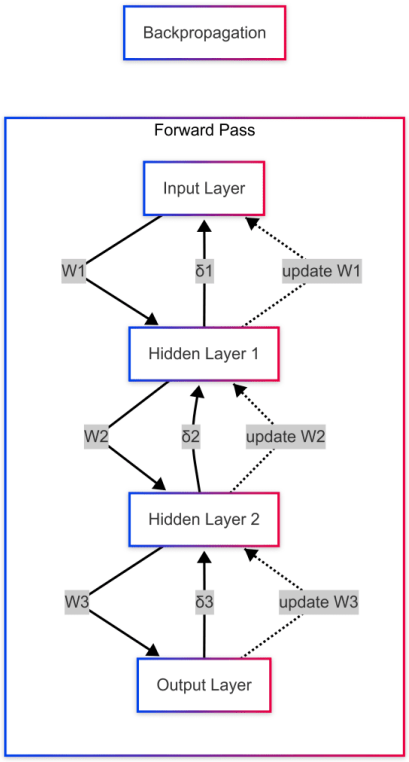


Figure 7: Illustration of forward pass and backpropagation in a neural network. The forward pass propagates inputs through successive layers, while backpropagation computes errors (δ) and updates the weights (W) accordingly. [50]

2.5.4 Regularization & Generalization

One of the biggest risks with neural networks is overfitting. This happens when the model memorizes the training data instead of learning the underlying patterns. It looks great during training but performs badly when faced with new inputs. For control systems that need to work in changing conditions, this is a serious problem. A network that only works in one carefully controlled situation is not much use in practice. To avoid this, different regularization techniques are applied. Weight decay prevents the weights from becoming too large, dropout randomly removes neurons during training to force the network to share responsibility and early stopping halts training as soon as the model begins to worsen on validation data.

Other methods improve the learning process right from the start. The way the weights are initialized can have a big impact. Poor initialization often leads to gradients vanishing or exploding, which slows down or even prevents learning. Techniques like Xavier initialization and He initialization were designed to give networks a better starting point. Normalization, such as batch normalization, helps too by keeping the values flowing through the network stable. These tricks don't make overfitting disappear completely, but together they greatly improve the odds that the model will generalize to new situations instead of just memorizing the old ones. In Figure 8 the training and validation accuracy is described. The widening gap between the two curves on the figure indicates overfitting, as the model performs well on training data but struggles to generalize to unseen data.

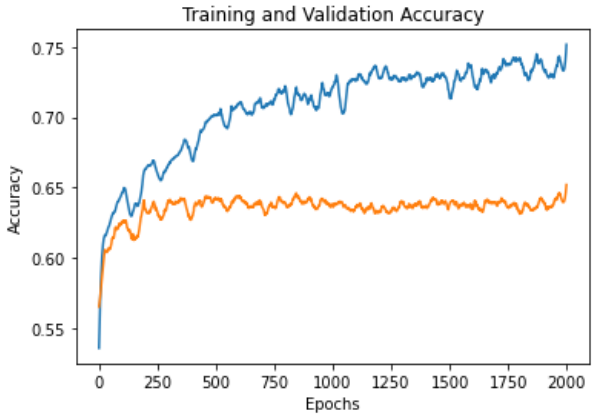


Figure 8: Example of training and validation accuracy during neural network training. [52]

2.5.5 Architectures in Control

Neural networks are not all built the same way. The multilayer perceptron (MLP) is the most straightforward and is often enough for many control problems. It works well when the data is mostly static and relationships between inputs and outputs are what matter. But when time plays a role—when the system's current state depends on what happened before—other designs are more useful. Recurrent neural networks (RNNs) were created for this reason. They keep information from previous steps, which lets them capture patterns across time. More advanced versions, such as LSTMs and GRUs, handle longer-term dependencies without running into the vanishing gradient problem. While classical approaches separate feature extraction from classification, deep learning architectures combine both steps within the network, enabling more complex representations (see Figure 9).

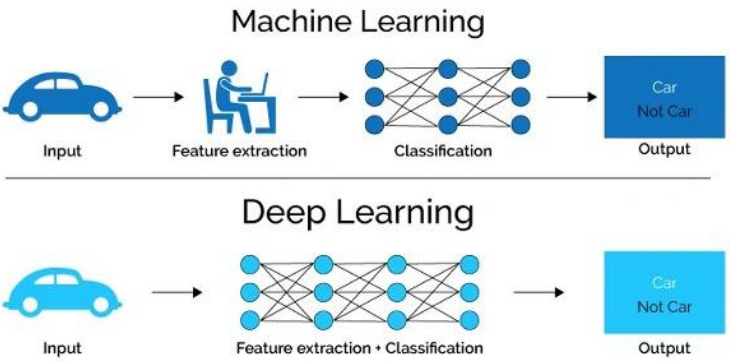


Figure 9: Comparison between traditional machine learning and deep learning. [53]

Other architectures have also proven useful in specific cases. Convolutional neural networks (CNNs), for example, are usually connected with images. However, they can also be applied in control when spatial or visual data is part of the system. For a self-balancing vehicle, which mainly uses sensor readings like angles and speeds, CNNs are less relevant. In that case, smaller MLPs or compact recurrent networks are often more practical. At the end of the day, choosing the right architecture is about matching it to the type of data available and the limits of the hardware.

2.5.6 Neural Networks for Control

In the context of this project, there are a many ways neural networks can be applied to control. The most direct one is to feed the sensor values into the network and let it decide the motor command straight away. For a balancing vehicle this could mean giving it the tilt angle from the IMU, the angular velocity, and the wheel speed, and asking it to predict the torque needed to stay upright. This approach is simple in design, but it requires a large amount of training data to perform efficiently. A second option is to use the network not as the controller itself, but as a model of the vehicle. Here the network predicts how the system will respond to a given input, and that prediction is then used by a model-based controller to plan the best action.

Another practical approach is to let the neural network work alongside a more traditional controller. For example, a PID controller might handle the basic balancing task, while the network helps correct for disturbances or nonlinear effects that the PID alone cannot manage. This type of hybrid setup is often more reliable, because the classical controller provides stability while the neural network adds adaptability [28]. There are also mixed methods such as neuro-fuzzy systems, which combine fuzzy rules with the learning ability of neural networks. In the end, there isn't a single "best" way to use neural networks in control. Their role depends on the system, the data available, and the balance between reliability and flexibility that the application demands.

2.5.7 Data Preparation and Preprocessing

Before the actual training of a neural network for a task similar to the one proposed to this thesis we had to preprocess the data and convert them in a format which could be useful. The IMU readings, especially the tilt angle, were noisy. Angular velocity had a bit of drift, and the wheel speed signals sometimes didn't match up with the other sensors in time. If the data were used as was, the network would have ended up learning errors instead of real patterns. To overcome this problem, we filtered the signals to remove noise and normalized the values so that everything stayed on a similar scale [4]. We also made sure the data streams were aligned in time, because even a small delay between signals could confuse the model.

Another important step was deciding how to split the data. The standard approach is to divide it into three sets: training, validation, and testing. The training set is what the model learns from. The validation set helps to tune the network and spot overfitting. The test set is saved until the end to see how the model performs on completely new data. For this project, we also needed to make sure the dataset included different scenarios. We collected runs on smooth and rough ground, with different payloads, and under small disturbances. Without this variety the network might have worked well in the lab but failed in real-world conditions.

2.5.8 Deployment on Embedded Systems

Designing and training a network in simulation significantly differs from deploying the controller to a real device. The Arduino Nano 33 IoT, which used here, has very limited memory and processing power. That meant the model couldn't be large or overly complex. A deep network with millions of parameters might run fine on a computer, but it simply wouldn't fit or run fast enough on microcontroller with limited resources. The only option was to use a compact design with just a few layers.

Even with smaller networks, optimization was needed. One common method is quantization, which basically shrinks the model by reducing weight precision—for example, from 32-bit

floating-point numbers to 8-bit integers. This makes the model lighter and faster, while keeping accuracy fairly close to the original. Tools like TensorFlow Lite Micro make this process easier and let the model run on hardware like the Arduino. For this project, speed was critical. The controller needed to respond within milliseconds to keep the vehicle upright, so keeping the model small wasn't just a nice improvement—it was absolutely necessary. The overall workflow for deploying a trained neural network on embedded hardware is presented in Figure 10.

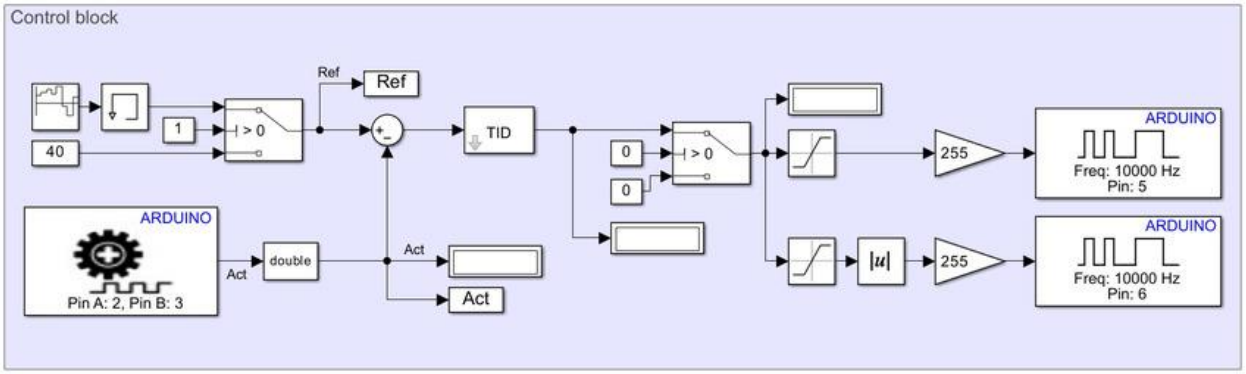


Figure 10: Workflow for deploying a trained neural network on embedded hardware. After training on a computer, the model is optimized and transferred to a microcontroller (e.g., Arduino Nano 33 IoT) for real-time control. [54]

2.5.9 Limitations and Risks

Even though neural networks can be powerful, they come with some clear downsides. The biggest one is their dependency on data. If the training data doesn't cover enough situations, the network can behave unpredictably when something new happens. For example, if the model is trained using only smooth lab conditions, it might fail completely on rough ground or when the vehicle carries extra weight. That makes them risky for real-world control unless there is a backup plan, like a simpler classical controller, to step in when things go wrong.

Another issue is that neural networks are often seen as "black boxes." Unlike fuzzy logic, where you can read the rules and understand the reasoning, neural networks hide their knowledge inside thousands of weights. This makes it very difficult to explain why the controller produced a certain output. In a safety-critical system, that lack of transparency can be a real problem. Because of these limits, they are often used alongside traditional methods. In practice, the classical controller provides a solid foundation, while the neural network adds adaptability where it's needed most.

2.6 Inverse Control

Inverse control, often called model inversion control, is based on a very straightforward idea. If we know how a system reacts to inputs, then in theory we should also be able to work backwards: figure out the input that produces a specific output. [32] In other words, if the system maps inputs to outputs, the inverse controller maps the desired outputs back to the right inputs. At first glance, this makes the control problem look very direct. Instead of tuning a feedback controller step by step, we try to "cancel out" the system's dynamics by applying the opposite. An example of inverse control is presented in Figure 11, showing a block diagram of it, The inverse model computes the required control input u(k) from the desired output yd(k+1), which is then applied to the process to achieve the actual output y(k+1).

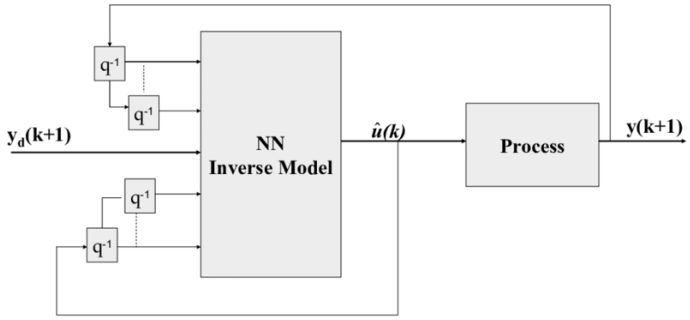


Figure 11: Block diagram of inverse control using a neural network. [55]

If a system naturally lags or overshoots, an inverse controller tries to pre-compensate for this behavior so that the response becomes faster and cleaner. When the model of the system is known and can be inverted analytically, the method can be very effective. In practice, however, many real systems are too complicated to be inverted exactly. Nonlinearities, time delays, and disturbances often get in the way, which means that the inverse must be approximated. One way to do this is by training a neural network to learn the inverse directly from data instead of writing it out mathematically [43].

The main drawback of inverse control is that it depends heavily on the accuracy of the model. If the model is even slightly wrong, the inputs calculated by the inverse may not give the expected outputs, which can cause instability. For this reason, inverse control is rarely used on its own. More often it is combined with classical feedback controllers, which can correct for the errors that the inverse model doesn't capture. This hybrid approach offers a good balance: the inverse controller provides fast responses, while the feedback loop keeps the system stable when conditions change.

In the case of the self-balancing vehicle, inverse control can be imagined as calculating the exact motor torque that will bring the tilt angle back to zero. If the dynamics between torque, tilt, and wheel speed are well captured, the controller could directly compute the correction needed to stay upright. Of course, in real experiments the model is never perfect, and external disturbances - like bumps or uneven surfaces - make things even harder. That is why inverse control is often treated as a useful foundation, but one that works best when paired with adaptive methods such as neural networks or fuzzy logic. This way, the benefits of model-based control are preserved, while the shortcomings are handled by more flexible learning techniques. Figure 12 shows a block diagram of a classical feedback control system which unlike inverse control that directly computes the input from the desired output, the classical feedback relies on the control error to adjust the system's behavior.

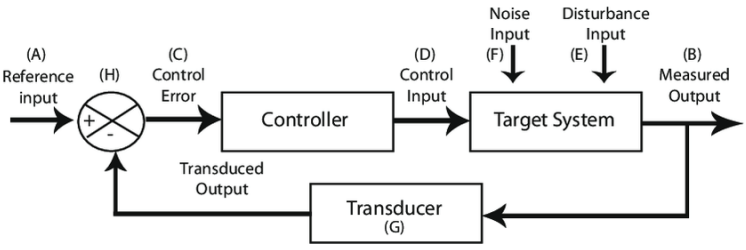


Figure 12: Block diagram of a classical feedback control system. [56]

Chapter 3

3.1 Introduction to the Arduino Nano 33 IoT

The Arduino Nano 33 IoT is the microcontroller used throughout this project. It keeps the familiar Nano footprint but adds a modern 32-bit core, integrated wireless, and a six-axis IMU, which makes it a compact platform for mobile control. In the self-balancing vehicle, it runs the high-rate stabilizing loop, acquires and fuses sensor data, and, when needed, streams telemetry over USB, Wi-Fi or BLE without additional shields.

3.1.1 Hardware overview

At the heart of the board is Microchip's SAMD21, an ARM Cortex-M0+ running at 48 MHz with on-chip Flash and SRAM that are sufficient for tight real-time code and compact inference models. Wireless connectivity lives on a separate u-blox NINA-W102 module that provides 802.11 b/g/n Wi-Fi and Bluetooth Low Energy, leaving the main MCU free to keep deterministic timing in the control loop. A hardware ATECC608A crypto element is also present; it is not essential for stabilization but becomes useful if the system later authenticates to cloud services. The board exposes Arduino-style GPIO with PWM capability for motor control, multiple analog inputs via a 12-bit ADC, and the standard serial busses—UART, I²C and SPI—so it interfaces cleanly with drivers and sensors used in this build.

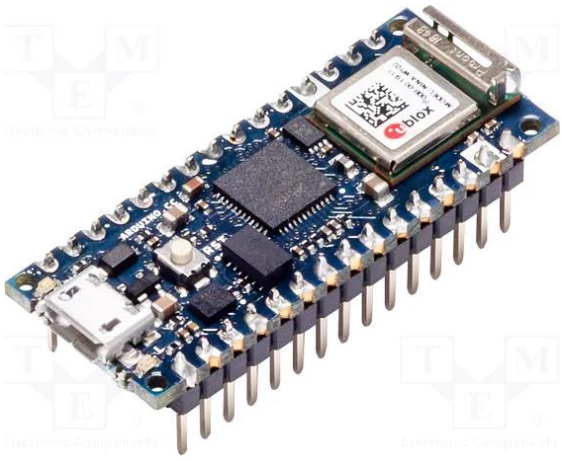


Figure 13: Arduino Nano 33 IoT board. The SAMD21 (ARM Cortex-M0+) acts as the main MCU, while the u-blox NINA-W102 provides Wi-Fi/BLE. The board integrates a 6-axis IMU and 3.3 V I/O in a Nano-sized form factor. [20]

3.1.2 On-board IMU and sensing

An important reason this board suits a balancing platform is the integrated six-axis IMU (accelerometer and gyroscope, commonly an LSM6DS3 on this model). It delivers synchronized accel/gyro measurements at several hundred hertz, which is adequate for a 200–400 Hz stabilizing loop. In practice the system estimates the gyro bias at startup while the vehicle rests, subtracts it in software, and, where possible, compensates accelerometer offsets. Attitude is estimated with a lightweight complementary filter that blends high-frequency gyro information with the low-frequency gravity vector from the accelerometer; when more disturbance rejection is needed, Mahony or Madgwick filters are also viable provided their gains are tuned carefully [27]. Fixing a consistent axes convention early in the codebase prevents one of the most common failure modes—apparent oscillations caused by swapped

signs or axes (see Figure 14). When wheel speed is required, incremental encoders are read via interrupt-capable pins, so edges are timestamped with minimal jitter and fused with IMU-based velocity estimates at low speed.

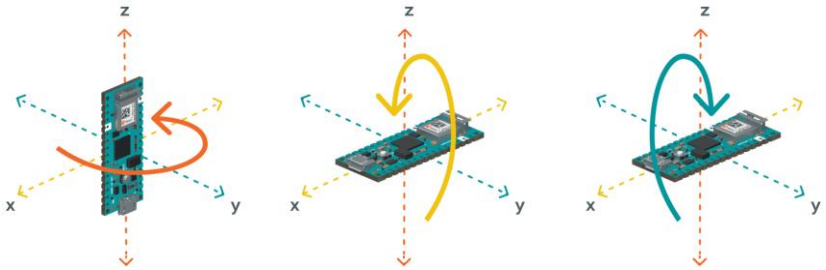


Figure 14: IMU axes and rotation conventions on the Arduino Nano 33 IoT. Rotations about the board axes correspond to roll (x), pitch (y), and yaw (z). [20]

3.1.3 Real-time structure and timing

The control software is organized around a fixed-rate loop driven by a hardware timer rather than by delays. Each tick triggers a sequence of actions: read the IMU, update the tilt estimate, compute the control action (PID/LQR or a learned policy) [16], and refresh the PWM outputs to the motor driver. Non-critical work—logging, user interface, and wireless messaging—runs at lower priority outside the interrupt context so it cannot stall the stabilizer. Because the SAMD21 lacks a hardware floating-point unit, heavy matrix operations are avoided inside the fast loop; even so, the device comfortably sustains hundreds of hertz with compact controllers. To push switching noise outside the audible range and reduce coupling into the IMU, PWM is kept at or above 20 kHz, and the motor driver's input characteristics are verified for 3.3 V logic [2].

3.1.4 Power and signal integrity

Power and logic levels deserve emphasis. The Nano 33 IoT is a 3.3 V device, and its pins are not 5 V-tolerant, which means every peripheral must be level-compatible. During development the board is powered from USB; in stand-alone operation it receives a regulated 5 V rail, while the motors draw from a separate supply sized for their current spikes. Grounds are tied at a single star point near the motor driver, and the wiring is decoupled with a bulk capacitor close to the driver and small ceramics near the MCU and IMU. This layout is not cosmetic: on a balancing platform the sharp current transients of the motor stage can pollute the sensor lines and corrupt the tilt estimate if grounding and routing are careless. Where the driver permits, fast flyback paths and snubbers further limit electromagnetic interference.

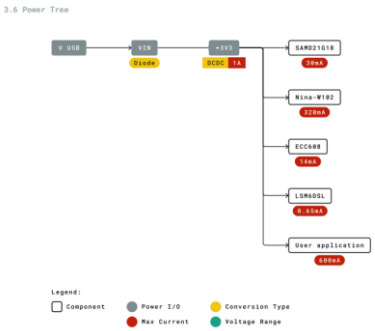


Figure 15: Power tree of the Arduino Nano 33 IoT. USB/VIN is regulated to 3.3 V for the on-board devices (SAMD21 MCU, NINA-W102 Wi-Fi/BLE module, ATECC608 secure element, and LSM6DSx IMU). The diagram also shows the available current budget for user applications. [20]

3.1.5 Deploying learning-based controllers

The board can host compact learning-based controllers, provided their size matches the available memory and compute. With TensorFlow Lite Micro, a small multilayer perceptron with a few hundred to a few thousand parameters is realistic, especially after int8 quantization reduces the RAM/Flash footprint and speeds inference. Fuzzy rule-bases are a good fit because their evaluation cost is predictable and the rule table can be stored in Flash; genetic algorithms are run off-board to tune gains or weights, and only the final parameters are deployed to the Nano. In all cases a simple compile-time switch or serial command keeps a well-tuned PID/LQR available as a fallback, which is invaluable during early tests and when operating near the stability limits [1].

3.1.6 Communications, logging, and updates

Development is greatly simplified by clean telemetry. High-rate logging over the USB serial port allows inline plotting of angle, rate, and control signals without disturbing the timing of the main loop. When wireless communication is needed, WiFiNINA and ArduinoBLE provide straightforward channels for lower-rate status packets and parameter tuning. These network stacks can introduce jitter if misused, so they are isolated from the fast loop and run in the background; over-the-air updates are considered only once the stabilizer is stable and the timing budget is well understood.

3.1.7 Pin mapping in this build

For reproducibility the exact wiring used in the prototype is documented as a pin out later in the chapter (Figure 16 and Figure 17). In short, the IMU is accessed over the board's default I²C pins, the motor driver is driven by two PWM outputs and direction pins at 3.3 V logic, encoder channels - when present - arrive on interrupt-capable inputs, and a small set of GPIOs are reserved for a kill switch and status LED. All grounds meet at a single point near the driver, which minimizes ground loops and measurement noise.

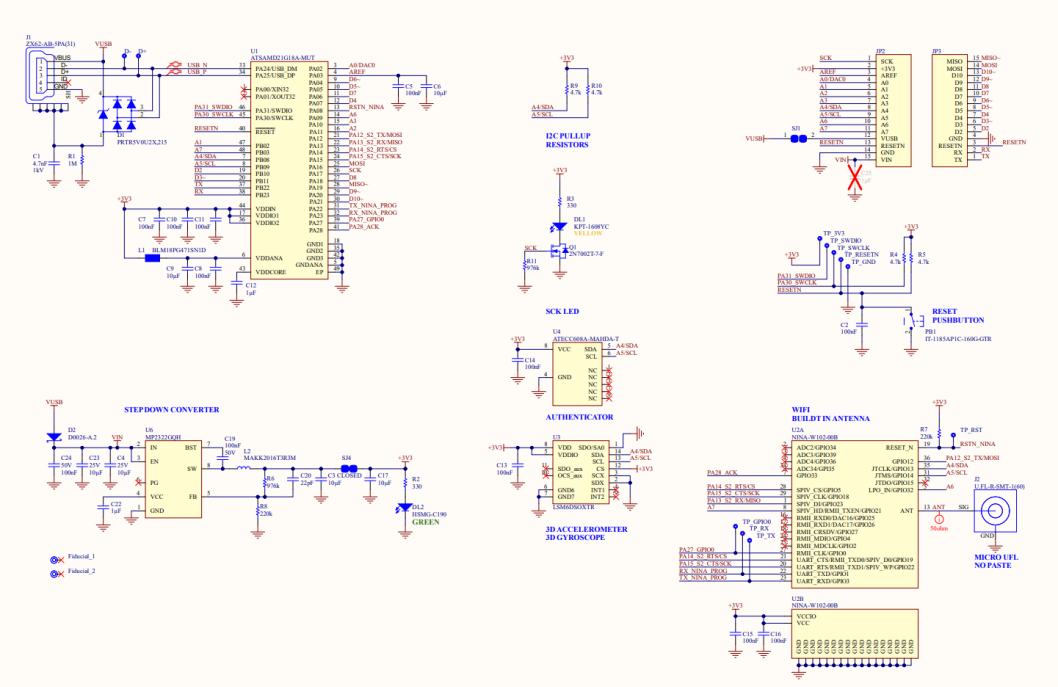


Figure 16: Arduino Nano 33 IoT main-board schematic (overview). [20]

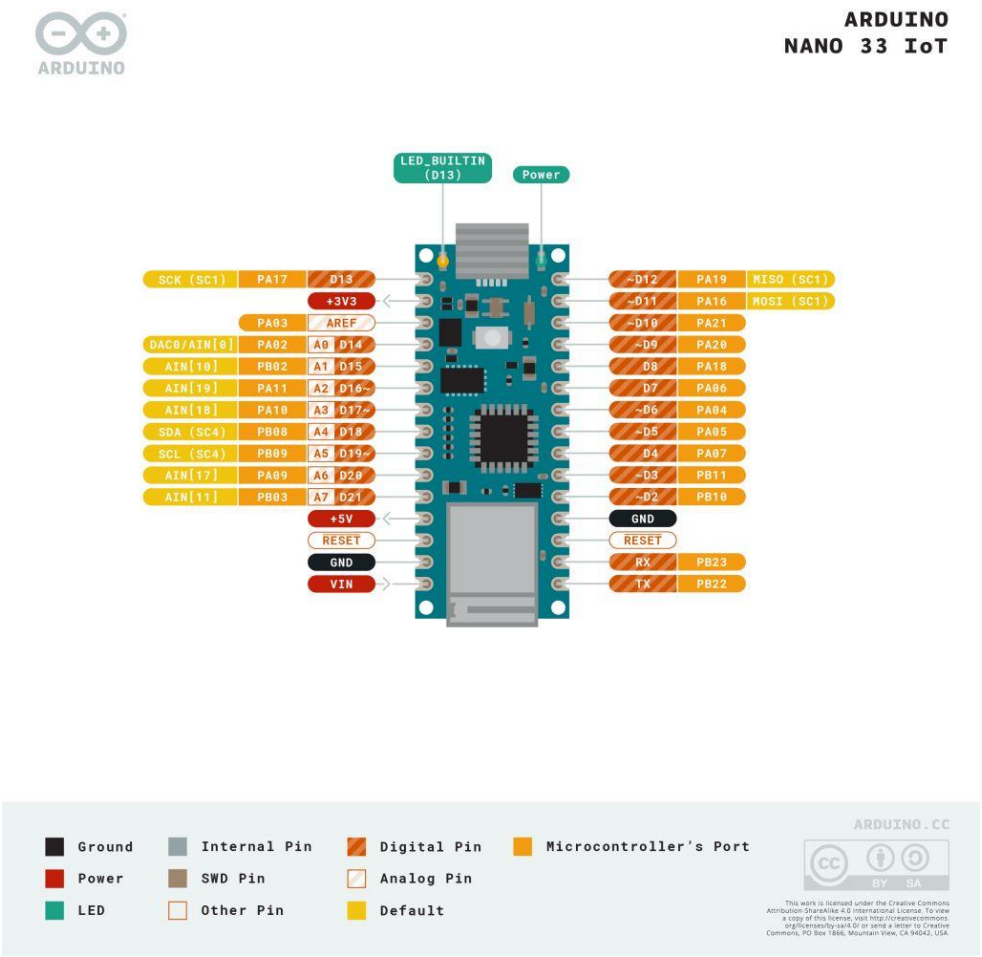


Figure 17: Arduino Nano 33 IoT pinout. Digital/analog/PWM pins, I²C (SDA/SCL), SPI, UART and power rails are shown. The board uses 3.3 V logic (pins are not 5 V-tolerant). Source: arduino.cc (CC BY-SA). [20]

3.1.8 Rationale for selection

In summary, the Nano 33 IoT fits the self-balancing vehicle because it integrates the components that matter—MCU, IMU, and wireless—on a single, quiet board; it offers enough real-time performance for 200–400 Hz control on a compact power budget; and it sits within a mature software ecosystem that speeds iteration when comparing classical and computational-intelligence controllers on identical hardware. With sensible power layout, a timer-driven loop, and modest model sizes, the board provides a stable foundation for the experiments reported in the remainder of this chapter.

3.2 Experimental Kit

3.2.1 Hardware overview

The experimental platform is a tabletop self-balancing motorcycle that implements the "reaction-wheel pendulum" principle. Computation is handled by an Arduino Nano 33 IoT, powered from a motor-carrier shield that also brings out encoder inputs, a high-current H-bridge for the DC motors, and an analog front-end for battery sensing. MATLAB/Simulink recognizes the board automatically; models are deployed and run in External mode so that code executes on the microcontroller while signals are streamed back to the host for logging and tuning. To keep sensing, estimation, and actuation synchronous, every hardware block in the model uses a common sampling time $Ts=0.01$ s (100 Hz) [4]. This rate is high enough to capture the dominant lean dynamics of the platform yet low enough for reliable serial telemetry and code execution on the Nano.

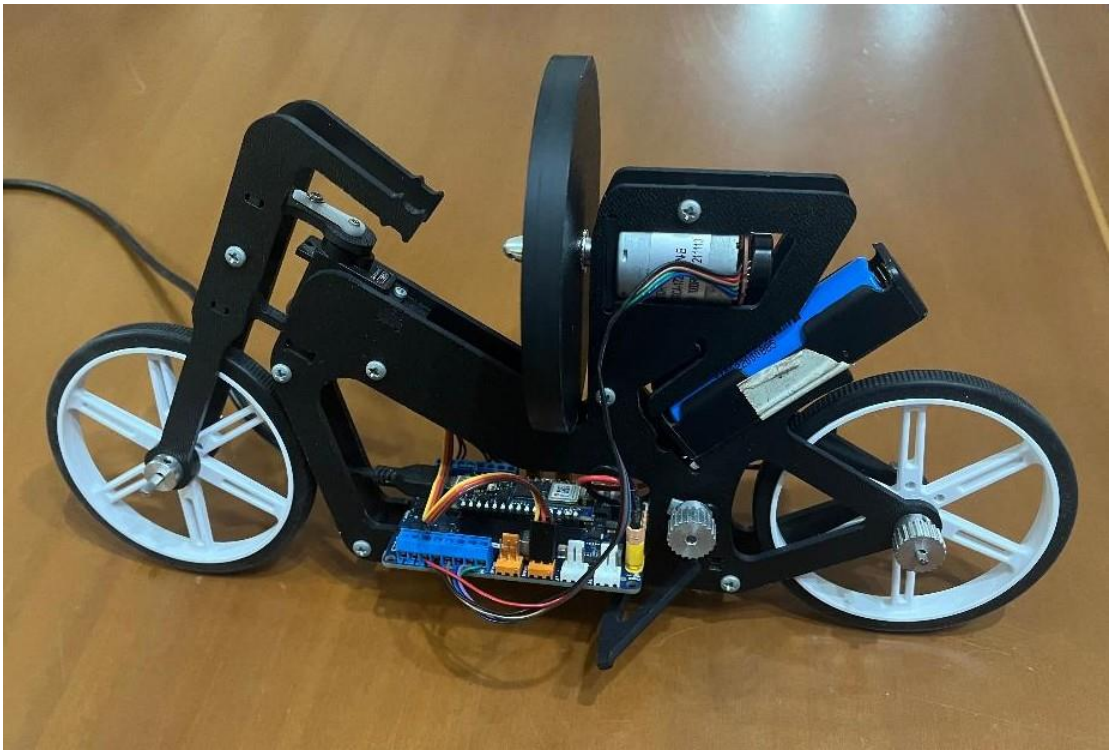


Figure 18: Experimental kit overview. The Arduino Nano 33 IoT and motor carrier drive the reaction-wheel pendulum and steering assembly; power is supplied by the onboard battery pack for stand-alone tests.

3.2.2 Embedded sensing

Lean attitude is measured by a Bosch BNO055 absolute-orientation module connected over I^2C at address 0x28 and operated in Fusion mode. The device provides Euler angles referenced to an inertial frame defined by gravity and the Earth's magnetic field, together with angular rates in the IMU body frame. For control we extract the lean angle θ about the wheel–ground axis and its time derivative $\dot{\theta}$. The IMU reports a four-element calibration status [sys, gyro, acc, mag] ∈ $\{0,1,2,3\}^4$. In software this vector is compared against a minimum requirement [0,0,0,3] and collapsed into a single Boolean "sensor-ready" flag via a product-of-elements operation. The calibration workflow is embedded in the experimental procedure: with the frame at rest gyroscope biases converge; the unit is then rotated at least 90° about each axis to complete accelerometer and magnetometer calibration. Because the BNO055's Euler convention and the body-rate sign convention are opposite under our coordinate choice, a unity gain of −1 is applied so that θ and $\dot{\theta}$ have matching polarity in the controller.

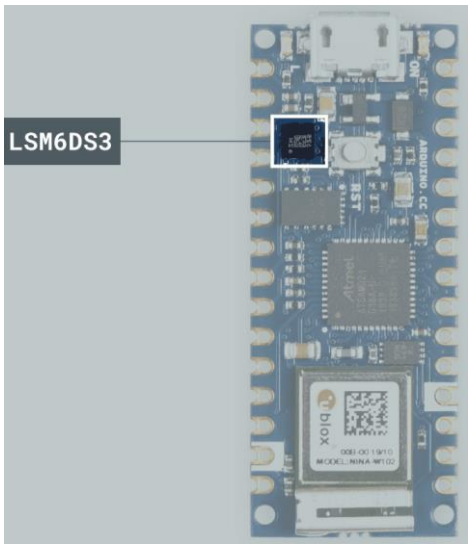


Figure 19: Location of the on-board IMU (LSM6DS3) on the Arduino Nano 33 IoT used to sense lean angle and angular rate. USB connector at the top for orientation. [24] [25]

Reaction-wheel speed ω is obtained from a shaft encoder integrated into the motor assembly. The carrier board exposes the tick count as a 32-bit integer. With 48 ticks per revolution the position in radians is $\phi[k] = \frac{2\pi}{48} ticks[k] \approx 0.1309 rad \times ticks[k]$. (1)

Differentiation is implemented with the Simulink Filtered Derivative block, whose continuous equivalent is $H(s) = \frac{s}{Ts+1}$ (2). At Ts=0.01 s and τ=0.1 s, the estimate $\hat{\omega}$ tracks rapid changes without excessive amplification of encoder quantization and mechanical ripple; using τ=1 s gives a visibly smoother but phase-lagged signal. The choice τ=0.1 s is therefore a pragmatic compromise between noise and delay, which is critical because phase lag directly reduces the effective derivative damping in a PD controller [26].

A battery-voltage monitor on the carrier supplies an integer measurement in the range 0–4095. The signal is cast to double precision and converted to volts with a calibration factor k_v determined by the board's divider and ADC reference; in our unit k_v≈1/236 V count$^{-1,}$ which

was verified empirically. This measurement is used both for safety interlocks and for interpreting motor behavior under sag.

3.2.3 Actuation and power

The motorcycle uses a dedicated DC motor to spin the reaction wheel, a second DC motor for the rear wheel (not required for in-place balancing), and a hobby servo for steering. The H-bridge driver accepts integer commands in [−255,255] where the sign encodes direction. In Simulink a normalized control effort u∈ [−1,1] is mapped to the driver range with a gain of 255 and bounded with a Saturation block; this ensures deterministic clipping and avoids integrator wind-up when the actuator saturates. A slider widget is connected to the same parameter when manual excitation is required during tests. The servo interface accepts $0-180°$ commands; its mechanical linkage on the model provides a smaller physical range, so software saturation is added when steering experiments are conducted.

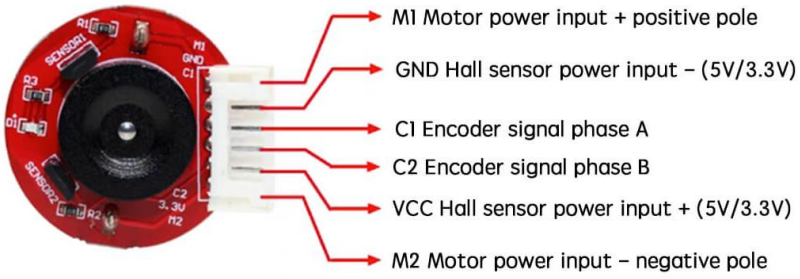


Figure 20: Inertia-wheel DC motor with integrated Hall-effect quadrature encoder as installed on the motorcycle chassis. [20]

3.2.4 Real-time software architecture

The Simulink model is organized as two subsystems— “digital controller” and “motorcycle”— to mirror the cyber-physical split (see Figure 21). The hardware subsystem encapsulates the IMU, encoder, battery monitor, and motor drivers, each parameterized by the shared *Ts*. The controller subsystem receives a bus of raw signals, performs signal conditioning, and outputs a single torque command for the reaction-wheel motor. Running in External mode exposes all internal signals to the Simulation Data Inspector, which is used extensively to compare alternative filter constants, verify sign conventions, and tune controller gains while the system operates.

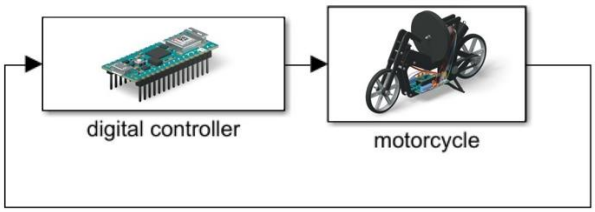


Figure 21: The digital controller computes the control command from the sensor feedback provided by the motorcycle subsystem.

3.2.5 Signal conditioning, calibration, and measurement fidelity

Several implementation details materially affect data quality. First, vector-sizing mismatches between IMU outputs and scalar processing blocks are resolved with explicit Signal Conversion blocks, preventing silent re-sizing by Simulink that would otherwise complicate code generation. Second, encoder quantization imposes a resolution of $\frac{2\pi}{48} \approx 0.131$ rad; at 100 Hz a one-tick step over one sample corresponds to a coarse speed increment of roughly 13 rad s^{-1}. The filtered derivative reduces this quantization noise by spreading increments over the filter horizon τ, but introduces a group delay of order τ, which must be accounted for in gain tuning. Third, IMU outputs are trustworthy only once the "sensor-ready" flag is true. The controller watches this flag and holds torque at zero until calibration completes; this prevents large transients caused by initial bias errors. Finally, polarity is verified by a simple bench test: leaning the frame counterclockwise should produce a positive θ and, under a steady roll, a positive $\dot{\theta}$. If either disagrees, the −1 gain is adjusted at the corresponding path.

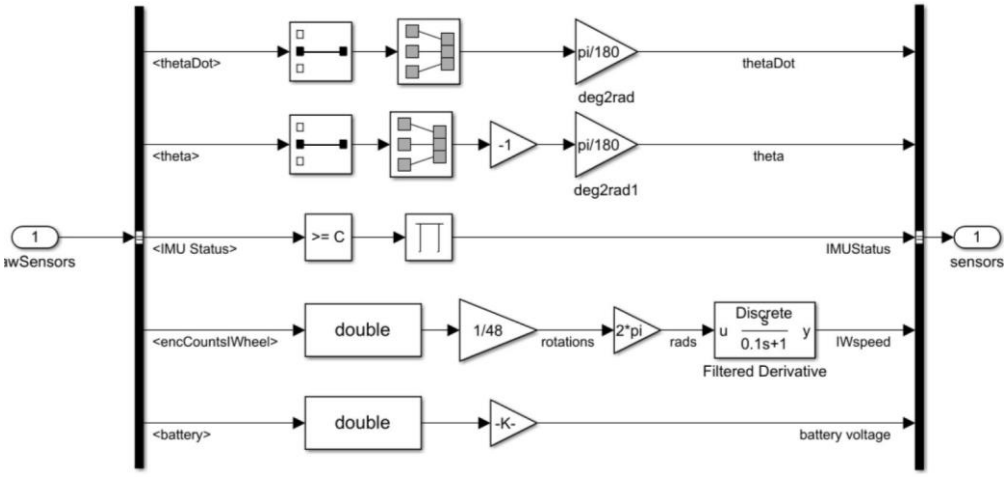


Figure 22: Sensor conditioning and signal preparation inside the digital controller.

3.2.6 Safety and operating envelope

Because the reaction wheel can accelerate rapidly, the model includes saturations on all control paths, a battery-voltage check that disables torque when V_{bat} falls below a threshold, and a manual stop in the host interface [42]. Empirically, the Ts=0.01 s, τ=0.1 s configuration yields stable telemetry and enough bandwidth to damp the natural oscillations of the inverted pendulum without over-driving the motor or exciting encoder noise [5]. These settings serve as the baseline for the balance controller developed in the next sections.

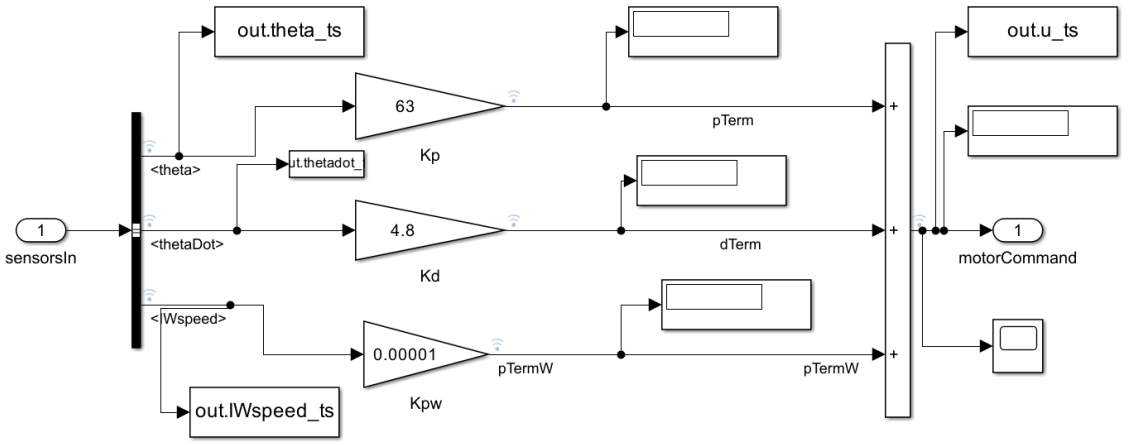


Figure 23: Digital controller: PD law with wheel-speed penalty, saturation, and safety gating. The command u is normalized to [−1,1] and logged together with key terms for tuning.

3.3 The Motorcycle Platform

This section documents the physical prototype used for all experiments. The goal is to describe the as-built hardware - its structure, sensing, actuation, power and wiring - and to record the practical decisions that make the platform reliable in closed-loop operation. The modeling of the upright equilibrium and the “inverted pendulum” analysis are deferred to Chapter 4, so the discussion here remains at the level of the real machine and the signals it exposes.

3.3.1 Mechanical structure

The chassis is a compact frame that carries two wheels, an electronics deck and a reaction-wheel module mounted on the main body. The geometry is deliberately simple and rigid. Keeping joints tight and spans short reduces unintended flex that would otherwise introduce extra modes in the 3–10 Hz band where the balancing controller operates. In static configuration the only dominant degree of freedom is the lean about the wheel–ground contact axis, which we denote by θ. The frame is assembled from a mixture of 3D-printed polymer parts and aluminum plates; the former simplifies routing and fixturing, while the latter provides stiffness in the steering column and the reaction-wheel mount. Fasteners are standard metric machine screws with thread-locking compounds on parts that see vibration. Cable runs are planned early in the build so that nothing can brush the tire or foul the reaction wheel during a large correction.

Mass placement matters more than absolute mass. The battery pack and motor carrier are located low and near the longitudinal centerline to keep the center of gravity close to the contact patch. This reduces the gravitational moment arm for small angles and lowers the required control torque. The reaction wheel is mounted high enough to have geometric leverage yet as close as practical to the lean axis to limit chassis twist. Clearances are checked with the full wheel-speed range in mind; if the wheel can reach several thousand rpm, it needs a clean envelope so that a transient overspeed does not risk contact. The fork is kept near neutral steer for the experiments in this chapter; steering effects are intentionally minimized so that the response seen in logs is attributable to the reaction-wheel torque rather than caster dynamics.

A final note is on repeatability. The frame is built so it can be disassembled and reassembled without re-machining: dowel-style location features, symmetric parts where possible, and a single datum face for the reaction-wheel mount. This is not merely cosmetic. When you compare controllers or re-run identification, a consistent mechanical baseline limits confounding factors.

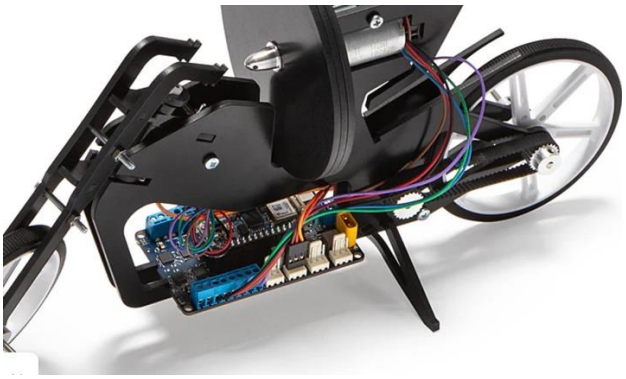


Figure 24: Detail of the electronics and actuator stack. The Arduino Nano 33 IoT and motor carrier sit low on the chassis while the reaction wheel, its DC motor, and belt reduction are mounted above the centerline to maximize corrective roll torque.[23]

3.3.2 Reaction-wheel drivetrain

The balancing actuator is a DC motor driving a dense aluminum disk that acts as the reaction wheel. When commanded, the motor produces a torque on the disk; the equal and opposite torque acts on the frame, which the controller uses to counter gravity. The disk's polar moment of inertia is chosen high enough that modest wheel speeds generate useful moments, but not so high that the assembly becomes fragile or forces large bearings. Dynamic balance of the disk is important: a small drill-dimple or adhesive weight on the light side is often enough to remove a perceptible 1× vibration. Doing this before closed-loop tests prevents the IMU from seeing a periodic disturbance that the controller would otherwise "fight."

A two-channel Hall encoder on the motor shaft provides quadrature counts. In our configuration the encoder produces 48 counts per revolution, which is sufficient for a clean speed estimate at the 100 Hz processing rate used here. The raw tick stream is converted to angle by a constant scale into radians, and angular speed ω is obtained with a filtered derivative. The time constant of the derivative filter is a key tuning knob: setting τ=0.1 s yields a speed trace that is responsive enough for damping while suppressing the quantization noise inherent in low-resolution tick sources. A much smaller τ makes the plot agile but noisy; a much larger τ is beautifully smooth but adds delay which shows up immediately as extra overshoot in lean-angle steps.

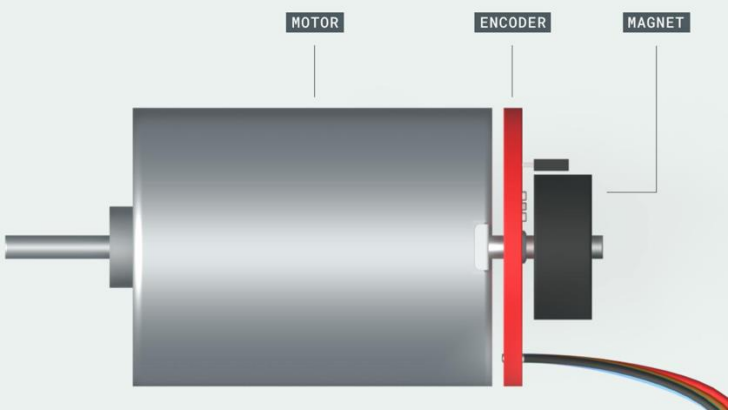


Figure 25: Inertia-wheel drive with integrated incremental encoder. A shaft magnet passes the Hall elements on the encoder, generating A/B (quadrature) pulses used to estimate angular position and speed. [20]

Operational limits are enforced in software. The motor command is normalized to u∈ [−1,1] and hard-clipped before it reaches the driver; internally the carrier maps this to its PWM range. This ensures the controller never requests an impossible torque and prevents integrator windup during large disturbances. A slow acceleration ramp can be enabled for bench tests, and an optional tiny proportional term in wheel speed bleeds momentum so the disk does not run away when a small bias is present. Bearings are checked for preload and runout so that the controller does not have to overcome static friction that varies with position. Figure 26 presents a diagram of the values that inertia wheel speed ω had during a 10 second simulation using a PD for a balancing test.

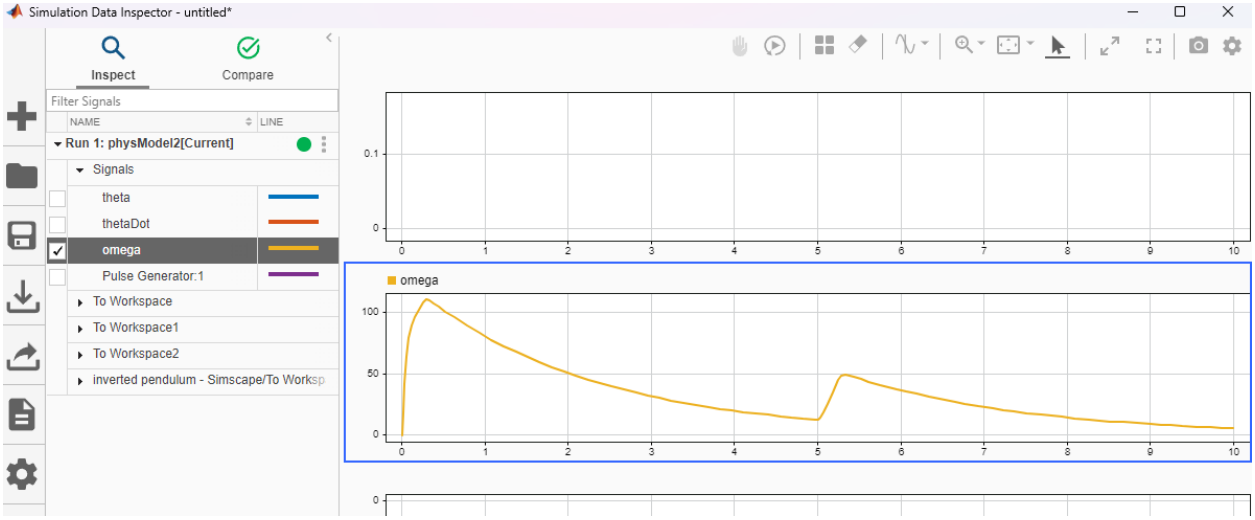


Figure 26: Inertia-wheel speed ω during balancing experiment. A pulse disturbance at t ≈ 5s momentarily accelerates the wheel; the PD + wheel-term controller drives ω back toward its steady value.

3.3.3 Sensor suite and placement

Two sensing subsystems provide the signals used by the controller. Attitude and lean-rate come from the on-board IMU; the actuator state comes from the reaction-wheel encoder. The IMU is placed on the compute board near the USB connector. This position shortens I²C lines

and keeps them away from the high-current motor leads, which reduces both conducted and radiated noise pickup. The sensor runs in “fusion” mode and outputs Euler orientation and body-frame rates at 100 Hz. In software we consistently treat the second Euler component as the lean angle θ and define a right-hand positive rotation about the wheel–ground axis (see Figure 27). Because the vendor’s Euler convention and our body-rate sign differ under this choice, a single sign flip aligns θ and $\dot{\theta}$ so the control law acts in the expected direction.

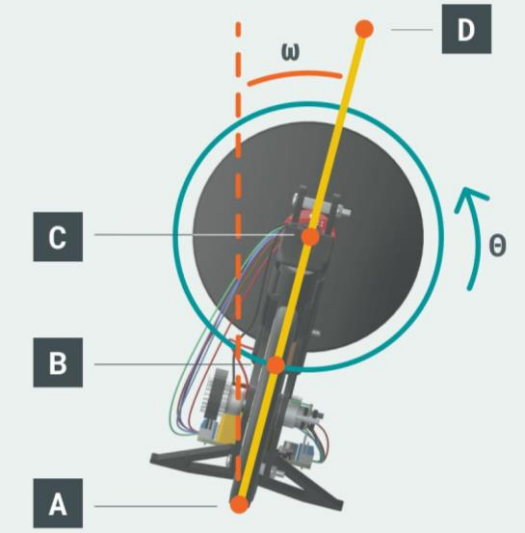


Figure 27: Kinematic variables measured on the prototype. The lean angle θ is referenced to the vertical (dashed line) and is estimated from the IMU on the frame. The reaction-wheel angular speed ω is measured by the shaft encoder on the inertia wheel. [20]

IMU calibration is part of the operating procedure rather than an afterthought. At boot the unit sits undisturbed for several seconds to let gyroscope biases converge. It is then rotated about each axis by approximately 90° to complete accelerometer and magnetometer alignment. The device publishes a four-element calibration vector [sys,gyro,acc,mag] with levels from 0 to 3; in the model a simple comparison against a minimum mask and a product-of-elements reduction yield a single boolean “sensor ready”. The controller watches this flag and withholds torque until it goes true. This one step eliminates a whole class of “mystery” transients that stem from acting on biased data. The IMU calibration in the project is projected on Figure 28.

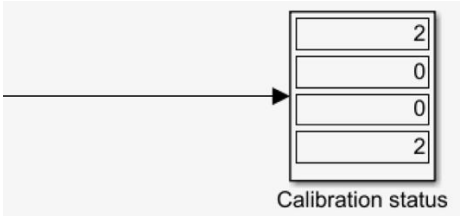


Figure 28: IMU self-calibration status shown in Simulink (0–3 scale per element: system, gyro, accel, magnetometer). Example values [2,0,0,2] indicate partial calibration; control remains inhibited until the minimum threshold is reached.

The encoder complements the IMU by telling us what the actuator is doing. Counts are cast to double precision, scaled to radians, and differentiated with the filtered derivative to form ω. The noise floor here is set by quantization: at 48 counts per revolution, one tick over one 10 ms sample corresponds to a coarse speed increment of roughly 13 rad·s^{-1}. The filter spreads that step over its horizon, which is why the chosen τ shows up so clearly in time histories.

Together, θ, $\dot{\theta}$ and ω give the controller the minimum state it needs for effective damping and momentum management.

3.3.4 Compute and motor interface

Computation runs on an Arduino Nano 33 IoT paired with the motor carrier. The carrier provides the H-bridge stages for the DC motors, level-compatible inputs for the quadrature encoder and a servo output for the steering column. It also brings the battery voltage into the microcontroller's ADC through a fixed divider. The firmware is generated from Simulink and deployed in External mode, so the controller executes on the microcontroller while signals stream to the host for tuning and logging. This "code-on-target, plots-on-host" workflow is central to the pace of iteration in this project.

The controller itself accepts a three-element sensor vector {θ,$\dot{\theta}$,ω} and computes a motor command u through a proportional and derivative action in θ with a small proportional penalty in ω . The command is then limited, optionally rate-limited, and exported as a normalized value which the carrier maps internally to PWM. For analysis, the implementation taps the proportional, derivative and wheel-speed terms individually and logs them along with the final u. Seeing these components during a transient is invaluable: it confirms the sign conventions, shows where damping is actually coming from, and makes it obvious when the wheel-speed penalty is too large (the controller "fights itself") or too small (the wheel drifts in speed under a bias).

All I/O and control subsystems run at a common sample period Ts=0.01 s. This cadence is fast enough to keep phase lag modest at the platform's natural frequency but slow enough that serial telemetry remains reliable and the SAMD21's CPU utilization stays well below saturation. Non-critical tasks such as logging and UI run at lower priority are explicitly kept out of the time-critical path so they cannot stall the stabilizing loop. An inside peek of the hardware model is presented on Figure 29, showing also the safety and enabling logic in Simulink. A user toggle and two health checks (IMU calibration and battery OK) generate an enable signal. When any condition is false, the whenDisabled path commands zero motor torque and the controller output is ignored; indicator lamps reflect each prerequisite.

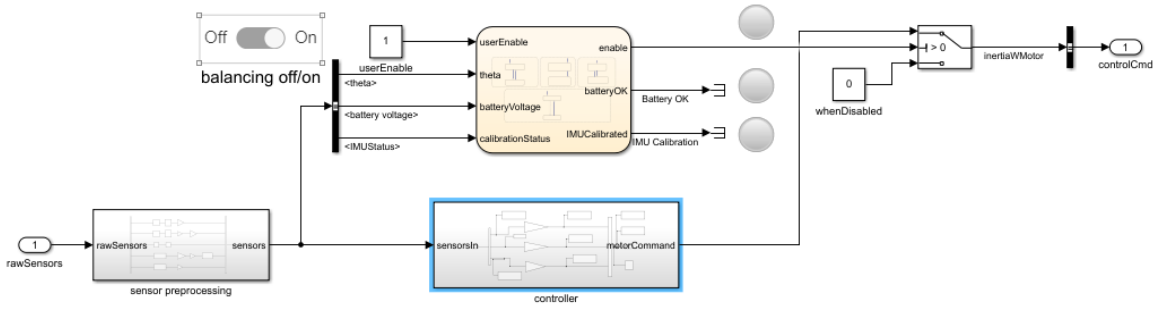


Figure 29: Safety and enabling logic in Simulink.

3.3.5 Power, wiring and integration details

The platform is powered from a single battery pack connected to the carrier's input. The carrier measures the pack via an internal divider; in software the raw ADC counts are scaled to volts

with a calibration factor verified empirically. That voltage appears in the logs and is used by a simple undervoltage interlock that zeroes the motor command if V_{bat} falls below a threshold. While balancing in place does not draw continuous peak current, the reaction-wheel experiences bursts, so the supply wiring is sized with headroom, and the battery is mounted securely to prevent intermittent contacts. Figure 30 presents the battery read signal chain used in Simulink, in which the sample is cast to double and scaled to volts.

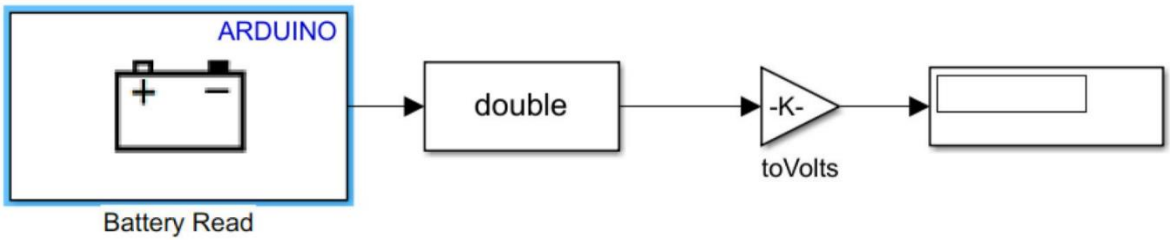


Figure 30: Battery Read signal chain in Simulink. The ADC sample is cast to double and scaled to volts, then displayed and logged for safety checks.

Wiring strategy has a measurable effect on data quality. High-current motor leads are twisted and routed away from the IMU. The encoder cable is short and, where possible, shielded from the H-bridge area. Grounds meet at a single star point on the carrier so that the ADC and encoder comparators do not ride on motor return currents. Decoupling capacitors sit near the motor driver, and the controller's PWM frequency is chosen high enough to push switching edges outside the audible band and to reduce coupling into the I²C lines. In practice, these simple layout choices reduce the IMU's spurious rate spikes and make the encoder-derived speed estimate markedly cleaner.

Integration is supported by a consistent naming scheme at the software interface. The "motorcycle" subsystem exports a bus of sensor signals with explicit units, and the "digital controller" subsystem exports a single command. This keeps scope plots readable and reduces the risk of cross-connecting signals during refactoring. Together with the fixed sample time and the External-mode workflow, this discipline is what allows side-by-side comparisons of different controller variants later without changing the plumbing underneath. Figure 31 shows how the IMU, encoder and battery connect in the project.

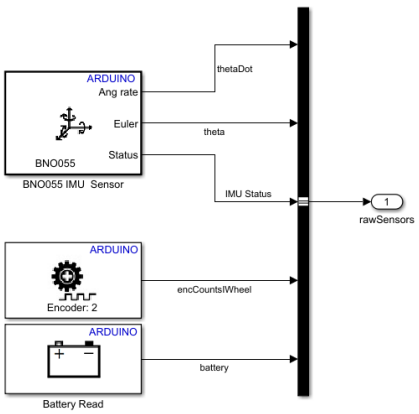


Figure 31: Raw sensor aggregation ("rawSensors" bus). The IMU provides the lean angle θ, angular rate $\dot{\theta}$, and a calibration/status vector; the inertia-wheel encoder contributes tick counts; the battery block contributes pack voltage. All signals are timestamped.

3.3.6 Practical considerations and limitations

Several practical constraints shape how the platform is used in the lab. The reaction wheel has finite momentum storage; if the controller balances while the IMU reports a small fixed offset, the motor will slowly ramp the disk to compensate. A tiny wheel-speed penalty cures this by bleeding momentum until the disk speed converges to a finite value at steady state. The IMU demands a short calibration routine at power-up; baking this into the procedure ensures that data taken for identification or tuning are comparable run to run. Mechanical damping is intentionally low, which makes the controller's derivative term do the real damping work; as a result, delay introduced anywhere - by filtering, slow telemetry, or excessive computation - shows up directly as overshoot, so timing budgets are watched closely.

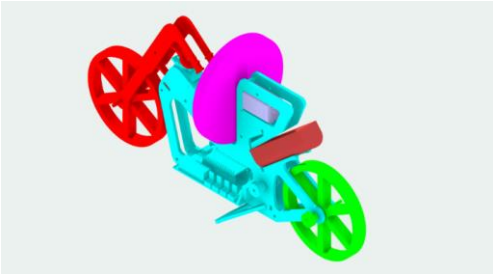


Figure 32: Motorcycle CAD with actuator/sensor locations. [19]

Environmental factors play a role too. Soft bench surfaces add compliance that slightly shifts the apparent natural frequency; hard surfaces are preferred once the loop is stable. Temperature drift can change the IMU's bias during long sessions; if logs show a slow walk in θ while the frame is visibly upright, a quick re-calibration or a bias-estimation step in software restores consistency. Finally, the platform is designed to be serviceable. Motors and encoders are mounted so they can be swapped without desoldering, and the wiring harness uses keyed connectors so that mistakes are unlikely during rebuilds. This reduces downtime and makes it feasible to repeat experiments days or weeks later with the same hardware behavior. A run example is shown in Figure 33. The values of θ, $\dot{\theta}$ and ω are "fixed", as these are the values and thus the results that work best while using a PD.

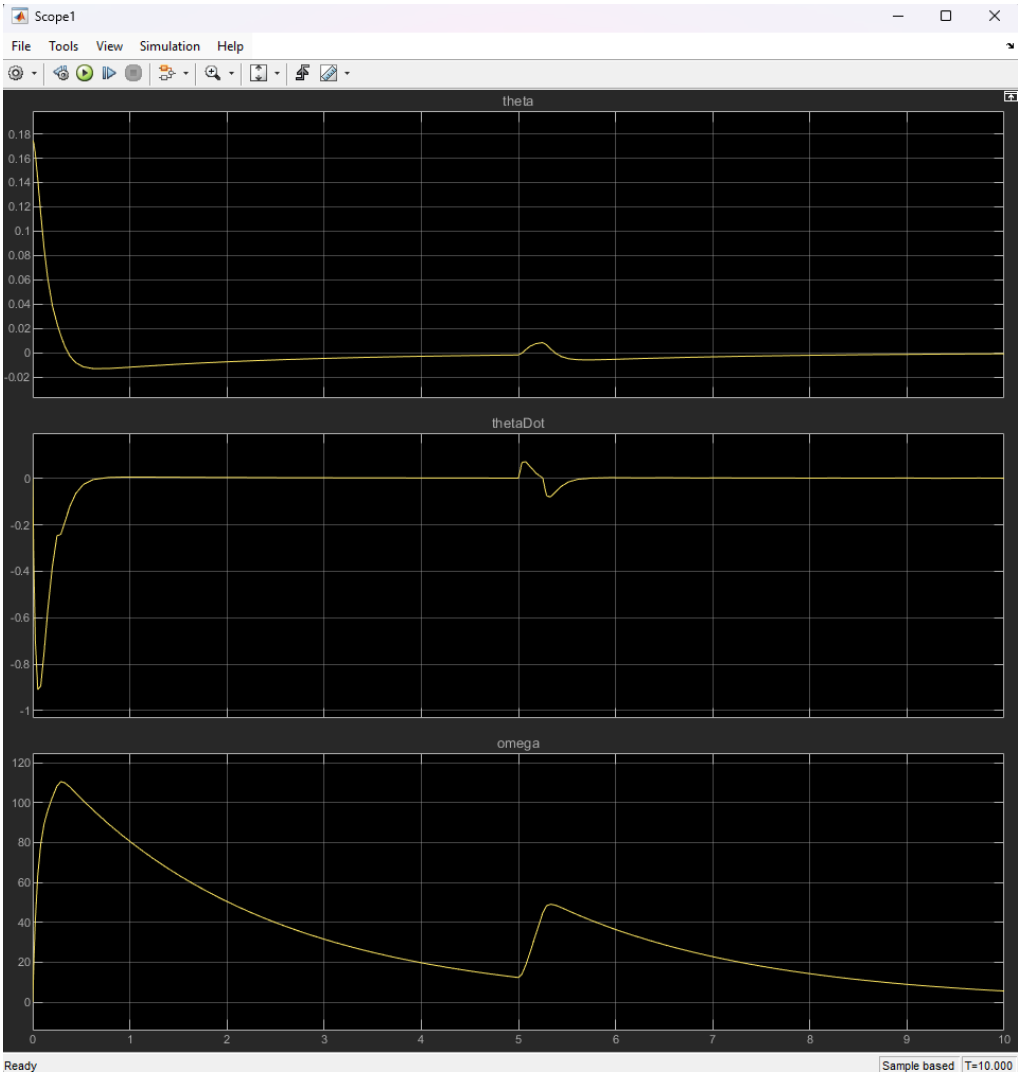


Figure 33: Closed-loop balance at standstill. A torque pulse at t = 5 s produces a small excursion (θ peak ≈ 0.02 rad), quickly rejected as the reaction wheel speed ω rises and then decays with the filter time constant. θ returns to zero without sustained oscill.

Chapter 4

4.1 Inverted-Pendulum Model Used in the Kit

4.1.1 Geometry, states, and modelling assumptions

The kit treats the motorcycle, as seen from the rear, as a rigid inverted pendulum with a reaction (inertia) wheel mounted on the frame. Three points define geometry. Point A is the ground pivot, the axis about which the whole assembly tips. The frame is idealized as a uniform solid cylinder of length l_{AD} and small cross-section radius *R*, so its center of mass B lies at the midpoint and is a distance $l_{AB} = \frac{l_{AD}}{2}$ (3) from A. The reaction wheel is attached to point C, located a distance l_{AC} above A along the frame. The wheel itself is modeled as a thin, solid disk of radius R and mass m_w ; the frame mass is m_r. Under this abstraction all links are perfectly rigid and all joints are frictionless, so the only conservative field acting is gravity.

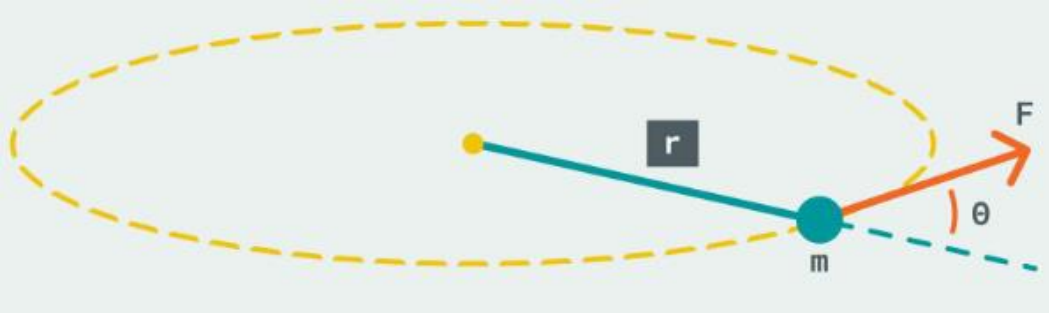


Figure 34: Definition of torque about the rotation axis; τ = rFsinθ. (4) [20]

The generalized coordinates follow directly from this geometry. The lean angle *θ* measures the frame's rotation about A and is zero when the frame is exactly upright; we take clockwise lean as positive. The wheel's absolute spin angle is not needed because it is axially symmetric; the only quantity that enters the dynamics is its spin rate *ω* and its derivative. These choices give a minimal two-degree-of-freedom description that still captures the coupling between body lean and wheel spin which is responsible for balancing.

With the shapes fixed, the inertias used throughout the kit come from standard formulae included in the notes. A thin, solid disk has $I_w^c = \frac{1}{2} m_w R^2$ (5) about its symmetry axis; a solid cylinder has $I_z = \frac{1}{2} m_r r^2$ (6) about its axis and $I_x = I_y = \frac{1}{12} m_r (3r^2 + l_{AD}^2)$ (7) about transverse axes. To obtain inertias about the ground pivot A, the parallel-axis theorem is applied. The wheel's inertia about A is $I_w^A = I_w^C + m_w l_{AC}^2 = m_w (R^2 + l_{AC}^2)$ (8). The frame's inertia about A is $I_r^A = I_r^B + m_r l_{AB}^2$ (9), where $I_r^B = \frac{1}{12} m_r (3r^2 + l_{AD}^2)$ (10) is the uniform-rod expression about its centroid. These quantities appear unchanged in the equations below and in the parameter script used in simulation.

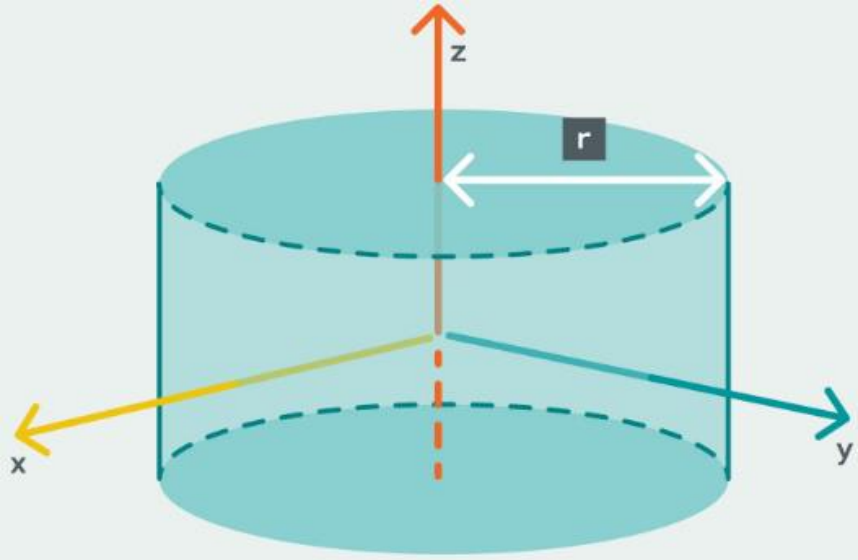


Figure 35: Solid cylinder used to model the rod; axes and radius r as defined. [20]

4.1.2 Rotational dynamics and torques

The lean dynamics follow Newton's law for rotation, τ= I θ¨ (11), applied about the ground axis at A. Two gravitational torques act to pull the system away from the upright configuration. The frame's weight acts at B a lever arml_{AB} from the pivot, producing $t_{gr} = m_r g l_{AB} sin\theta$ (12). The wheel's weight acts at height l_{AC}, producing $t_{gw} = m_w g l_{AC} sin\theta$ (13). The motor that spins the reaction wheel applies equal and opposite torques to the wheel and to the frame; we denote their common magnitude by τ_m and adopt the sign convention used in the kit so that it subtracts from the gravitational terms in the lean equation. Summing moments of A and equating them to the total rotational inertia about A times θ¨ give

$$m_r g l_{AB} sin\theta + m_w g l_{AC} sin\theta - \tau_m = (I_w^A + I_r^2)\theta^{\cdot\cdot}. \text{ (14)}$$

The denominator here, $I_w^A + I_r^A$, is the effective resistance to lean acceleration; any change to the mass distribution or wheel height modifies the lean responsiveness in a physically transparent way [7].

The wheel's spin dynamics use the same principle but about the bearing axis at C. Because the wheel is mounted on a tilting frame, the angular acceleration "seen" by the disk is the sum of its own spin acceleration $\dot{\omega}$ and the frame's lean acceleration θ¨ or torque must therefore accelerate both, yielding the second equation exactly as presented in the kit:

$$\tau_m = I_w^C (\dot{\omega} + \theta^{\cdot\cdot}), I_w^C = \frac{1}{2} m_w R^2 \text{ (15)}$$

Together these two equations capture the core coupling: the motor injects angular momentum into the disk and, through the equal-and-opposite reaction t_m, produces a corrective lean acceleration on the frame. No viscous friction or aerodynamic terms are included at this stage; the kit keeps the model conservative to make the structure of the dynamics and the role of feedback unambiguous [13].

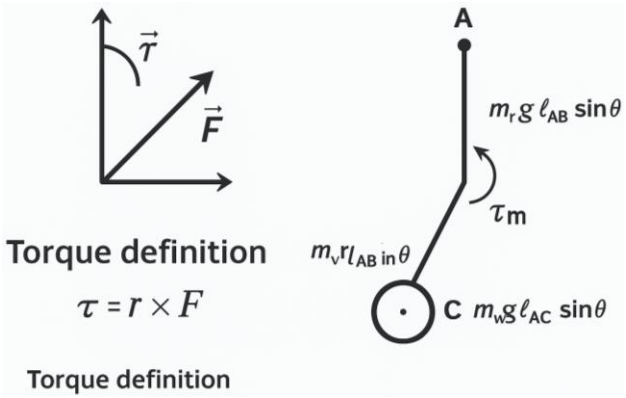


Figure 36: Torque & inverted pendulum torques (composite). The left is a clear vector diagram defining torque and the right one is a simplified inverted-pendulum sketch with points A (pivot) and C (wheel), showing gravity torques and the motor torque direction. [20]

4.1.3 State-space form used in simulation and code

To integrate the model in Simulink the kit recasts the second-order equations into first-order form with the state vector $x = \left[\theta\, \dot{\theta}\, \omega\right]^T$ and input $u = \tau_m$. The kinematic relation $\dot{\theta} = \dot{\theta}$ provides the first component. Solving the lean equation for θ¨ produces the second component,

$$\theta\ddot{} = \frac{(g \cdot sin\theta(m_r l_{AB} + m_w l_{AC}) - \tau_m)}{m_w\left(\frac{1}{2}R^2 + l_{AC}^2\right) + \frac{1}{12}m_r(3r^2 + l_{AD}^2) + m_r l_{AB}^2} \text{ . (16)}$$

The third component comes from the wheel equation by isolating $\dot{\omega}$,

$$\dot{\omega} = \frac{\tau_m}{I_w^C} - \theta\ddot{}\,,\ I_w^C = \frac{1}{2}m_w R^2 \text{ (17)}$$

This form makes physics transparent. The numerator of θ¨ is the net torque tending to right the bike, consisting of the gravity term $gsin\theta(m_r l_{AB} + m_w l_{AC})$ opposed by the motor reaction τ_m.

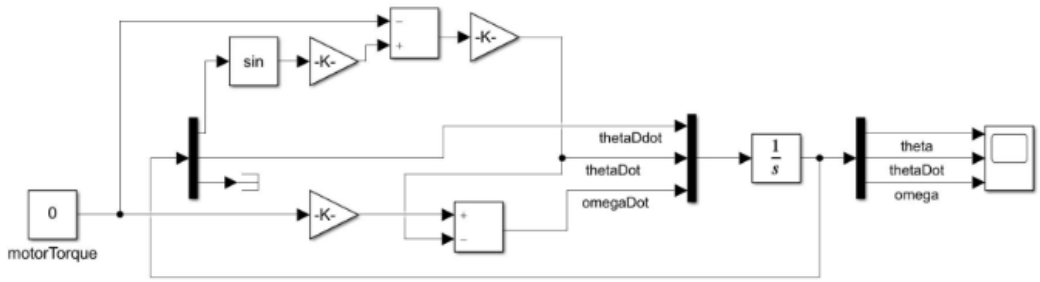


Figure 37: State-space realization of the inverted-pendulum model in Simulink.

The denominator is the total rotational inertia about the ground axis, showing why a heavier wheel mounted higher makes the system slower to tip yet more responsive to control torque. The wheel-speed equation shows the complementary effect: any torque used to change ω is "shared" with lean acceleration through − θ¨.

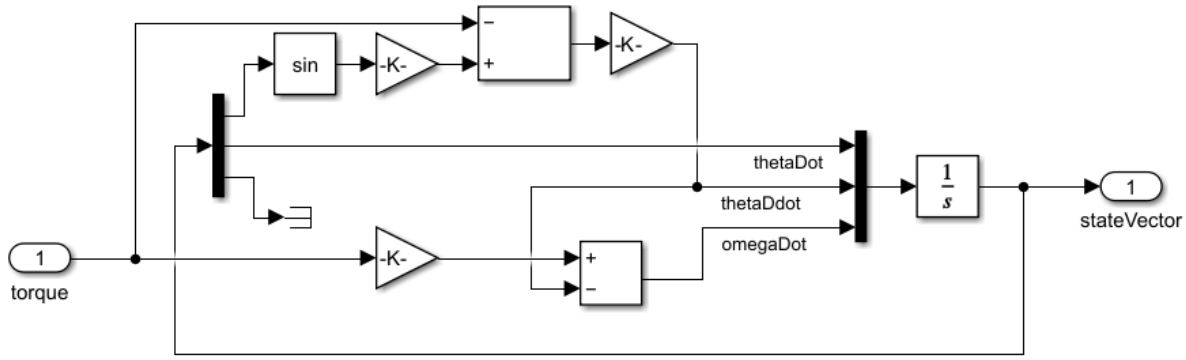


Figure 38: Top-level subsystem with defined I/O for controller-in-the-loop work.

As implemented, the kit's parameter script defines g, m_r, m_w, R, r, l_{AD}, l_{AB}, l_{AC} and computes I_w^C, I_w^A and I_r^A exactly as above. The Simulink diagram then realizes these expressions with Gain, Sum and Trigonometric blocks driving an Integrator whose state is $[\theta\ \dot{\theta}\ \omega]^{\mathrm{T}}$. Initial conditions such as θ(0) = 10° are set on the integrator to reproduce the free response plots shown earlier; later, the same structure is kept while the constant input τ_m is replaced by a feedback law. Because the equations retain the full sinθ term rather than a small-angle approximation, the simulations remain valid for the large leans you purposely test when provoking the controller with disturbances.

modelParameters.m

```
g = 9.80665;          % gravity constant
m_r = 0.2948;         % mass of the rod
m_w = 0.0695;         % mass of the inertia wheel
R = 0.05;             % radius of the inertia wheel
r = 0.02;             % cross section radius of the rod
l = 0.13;             % corresponding lengths
l_AD = l;
l_AC = l;             % assume wheel is mounted on the top of the pendulum
l_AB = l/2;
I_w_C = 0.5*m_w*R^2; % corresponding inertias
I_w_A = I_w_C + m_w*l_AC^2;
I_r_B = (1/12)*m_r*(3*r^2+l_AD^2);
I_r_A = I_r_B + m_r*l_AB^2;
```

Figure 39: modelParameters.m script parameterizes the simulation.

4.2 System Architecture of the Self-Balancing Motorcycle

4.2.1 Hardware platform and purpose

As mentioned beforehand, the self-balancing motorcycle was realized as a two-wheel chassis stabilized by a flywheel (inertia wheel). An Arduino Nano 33 IoT on an Arduino Motor Carrier served as the embedded target. The Motor Carrier provided the H-bridge channels for two brushed DC motors, an interface for the GS-9025MG steering servo, and I²C connectivity to the on-board BNO055 9-axis IMU. The inertia wheel motor included a hall-sensor encoder, which enabled angular speed estimation for safety limiting and for the wheel-speed feedback

term used in the balance controller. Power and protection functions were centralized on the carrier board; therefore only low-level wiring (motors, encoder, battery) and the IMU bus were exposed to the model. Figure 40 shows the architecture of the whole model, presenting the two subsystems that were used in the project.

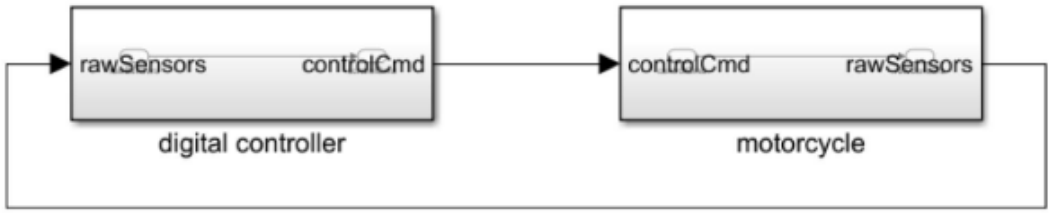


Figure 40: Top-level closed-loop architecture (controller ↔ plant).

4.2.2 Sensor suite and signal conditioning

Lean angle and lean rate were measured with the BNO055. The block was configured at an I²C address of 0x28 and sampled at 100 Hz. Euler angles and angular rates were output as 3-element vectors; within Simulink, Selector blocks isolated the second element in each vector because that axis corresponded to roll about the wheel–ground line. The angle stream was converted from degrees to radians, while the gyroscope output was negated to align its sign with the Euler convention used in the plant equations; this choice removed a hidden minus sign from the controller gains.

The IMU calibration vector was reduced to a single Boolean by comparing the four components against a threshold [0;0;0;3] and multiplying the logical results. This yielded a one-bit “IMU calibrated” status suitable for State flow logic and for dashboards.

The encoder on the inertia wheel reported int32 tick counts. A Data Type Conversion block promoted the stream to double precision, then a pair of gains converted counts → revolutions (1/48) and revolutions → radians (2π). The sign was chosen so positive wheel speed aided recovery from positive lean. A Filtered Derivative block with a 0.1 s time constant produced a smooth ω estimate that remained responsive enough for feedback and limiting. Battery voltage was read as a 12-bit integer (0–4095) and linearly scaled to volts with a gain of $\frac{1}{236}$, which simplified safety interlocks when charge was low.

All conditioned signals— θ, $\dot{\theta}$, ω, IMU status, encoder counts, and battery volts— were bundled into a Bus Creator named rawSensors to keep the top-level wiring concise and to make later expansion straightforward.

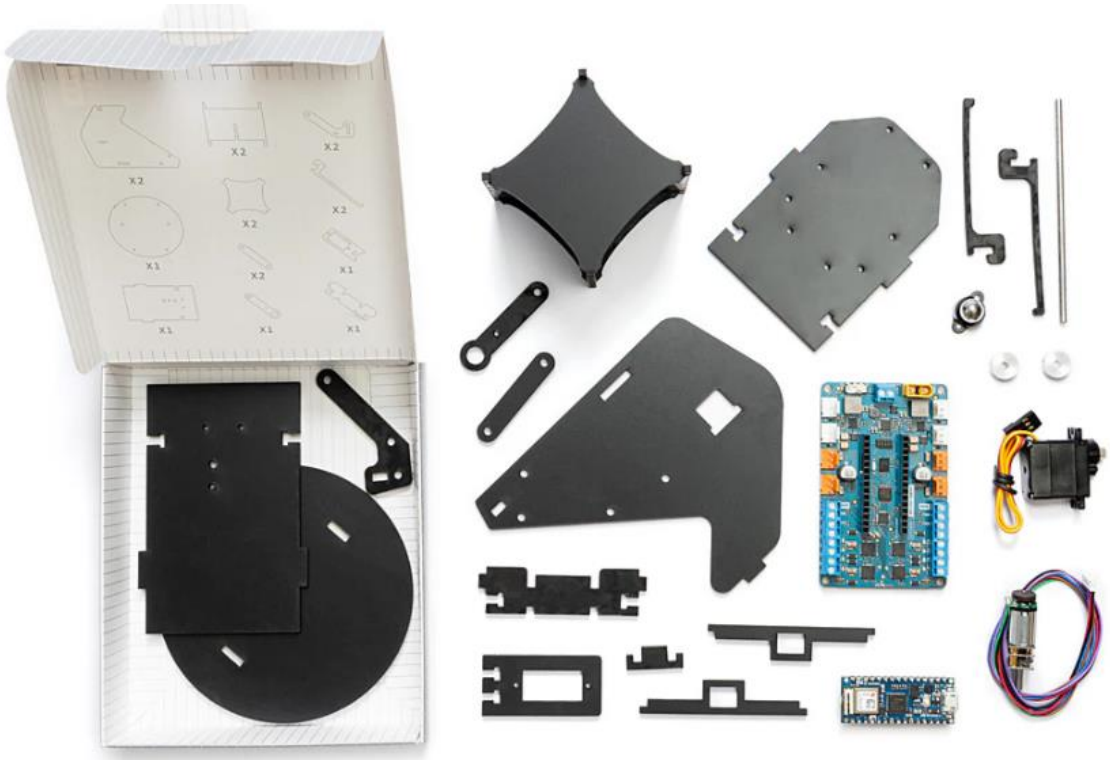


Figure 41: Self-Balancing Motorcycle kit components prior to assembly (plates, flywheel discs, Arduino Nano 33 IoT, Nano Motor Carrier, DC motor with encoder, steering servo, shafts/rods, bearing, ball casters, and wiring). [20]

4.2.3 Actuation chain (command to motor)

The controller produced a normalized command u∈ [−1,1] for the inertia-wheel channel. This value was first limited by a saturation stage to prevent out-of-range requests. The limited value was scaled by 255 to match the Motor Carrier's 8-bit PWM magnitude, rounded to the nearest integer, and passed to the DC-motor block. Positive commands drove one H-bridge polarity; negative commands drove the opposite polarity. During bring-up, a dashboard slider remained connected to the same path (gated by the safety chart) so the wheel could be exercised at low duty cycles before enabling automatic balance.

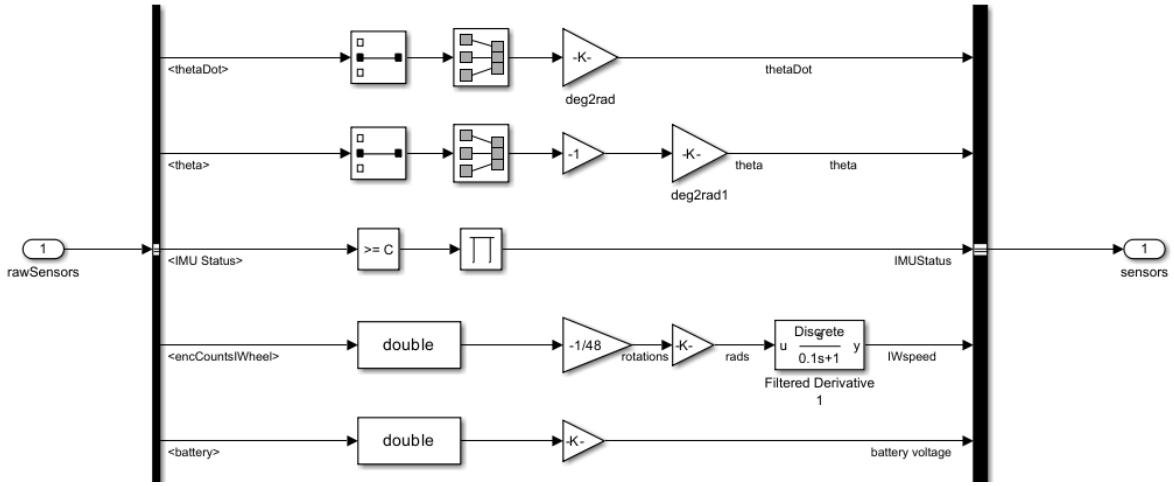


Figure 42: Sensor pre-processing in Simulink: unit conversions, filtering, and status logic before the controller.

4.2.4 Real-time model architecture

At the very top level two subsystems— digital controller and motorcycle— were connected by a closed loop. The motorcycle subsystem contained the hardware I/O blocks (IMU, encoder, battery, DC motors and optional steering servo) and exposed a rawSensors bus output and a controlCmd bus input. The controller subsystem accepted rawSensors, executed sensor preprocessing and safety logic, computed the motor reference, and returned the single inertia-wheel command on controlCmd [14]. A global sample time of Ts=0.01 s was defined in the MATLAB workspace and referenced throughout, so the loop rate could be changed consistently. External Mode was used so parameters and signals were tunable and streamable in real time, which is seen in Figure 43 on an ongoing run.

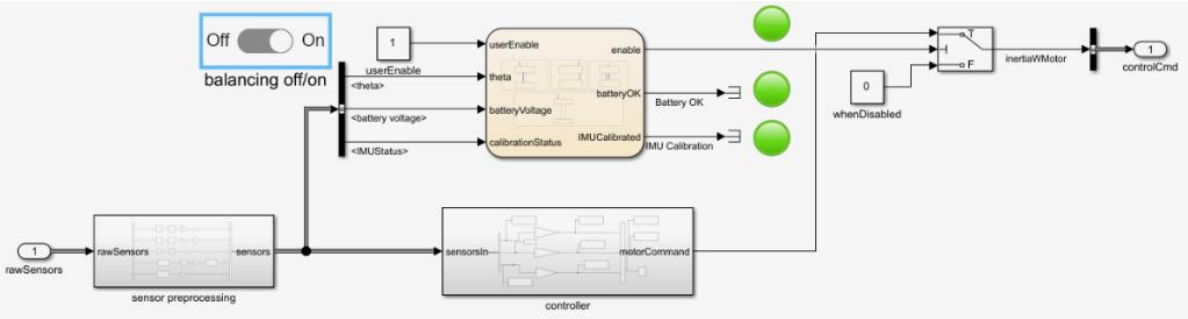


Figure 43: Safety and enable logic. The dashboard switch, IMU and battery checks are fused in a Stateflow supervisor to generate enable; when enable is true the motorCommand passes to the inertia-wheel motor, otherwise the command is forced to the safe value show.

4.2.5 Safety interlocks with State flow

Operational safety was handled by a State flow chart that evaluated four concerns in parallel: lean magnitude (fallen vs. standing), IMU calibration state, battery state-of-charge, and user enable. Each branch produced a local Boolean; a final calculation state combined them to form a single enable flag for the controller and motor path. When the absolute lean angle exceeded a threshold or the battery dropped below the configured limit, the enable flag was cleared and the command path forced the H-bridge to zero. The chart (Figure 44) defaulted to a “disabled” condition at model start so the system could be powered safely before calibration was complete [40].

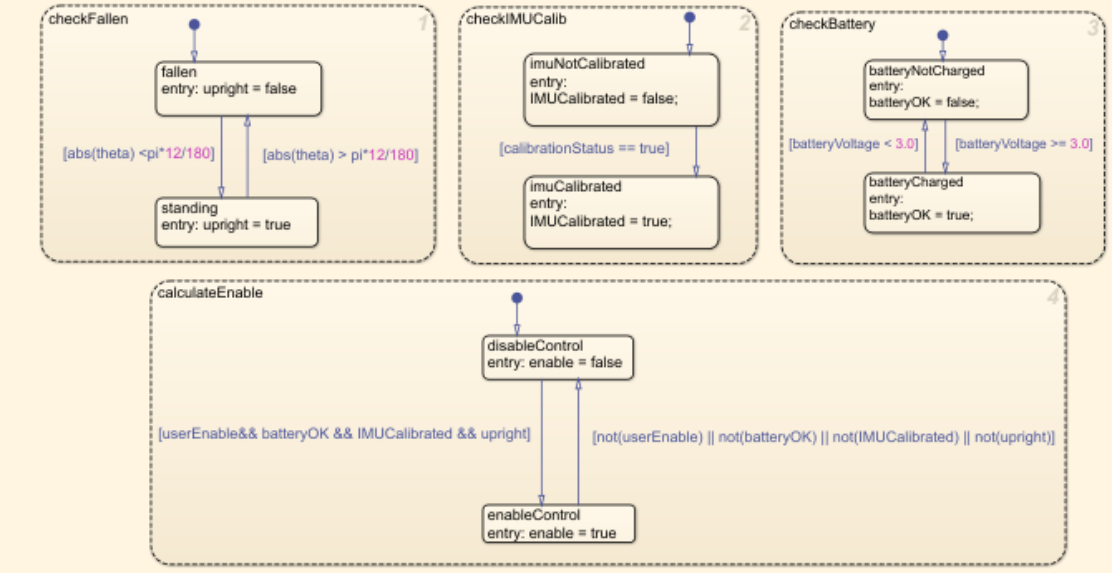


Figure 44: State flow safety supervisor for the self-balancing motorcycle. Three monitors—checkFallen, checkIMUCalib, and checkBattery—run in parallel to set upright, IMUCalibrated, and batteryOK.

4.2.6 Balance controller (PD + wheel-speed term)

The balance law followed the structure validated in simulation: $u = K_p\theta + K_d\dot{\theta} + \mathrm{K}_{pw}\omega$ (18)

The first two terms formed a PD stabilizer about the upright equilibrium, while the third term prevented the inertia wheel from accelerating unboundedly when a small sensor bias demanded a constant correcting torque. Gains were implemented as individual Gain blocks feeding a Sum block; the same signals were tapped to scopes and dashboard displays labeled pTerm, dTerm, and pTermW to visualize each contribution during tuning. Initial hardware values were selected with the values of them being; Kp≈63, Kd≈4.8, and Kpw≈0.00001 and then refined in External Mode. If static mass imbalance prevented convergence, a small bias was applied in the sensor-preprocessing path, so the perceived balance point aligned with the chassis geometry.

4.2.7 Build, deployment, and tuning workflow

The model referenced the single T_s variable, so all discrete blocks shared the same loop rate. External Mode over USB (and, when required, Wi-Fi) was used for deployment; this allowed live adjustment of K_p , K_d, and K_{pw} while streaming θ, $\dot{\theta}$, ω, and the enable flag to the Simulation Data Inspector. A Signal Conversion block was inserted on the IMU outputs to satisfy External-Mode upload sizing, removing a diagnostic warning and ensuring reliable logging. The encoder-speed time constant of 0.1s was verified by hand-spinning the wheel and comparing the resulting signal with a faster filter; the chosen constant provided a clean trace with acceptable delay for the wheel-speed feedback and limiter. On the project, the controller receives the conditioned signals θ, $\dot{\theta}$, and inertia-wheel speed ω from the sensor bus. Three paths form the command: a proportional arm $\mathrm{K_p}$=63 acting on θ, a derivative arm $\mathrm{K_d}$=4.8 acting on $\dot{\theta}$, and a small wheel-speed proportional arm $\mathrm{K_{pw}}$=0.00001 acting on ω to arrest spin-up. The three terms (pTerm, dTerm, pTermW) are summed to produce the normalized motor command u; telemetry taps (out.theta_ts, out.IWspeed_ts, out.u_ts) display

each contribution and the final u during tuning. The output is subsequently gated by the safety supervisor's enable signal before reaching the inertia-wheel motor (Figure 45).

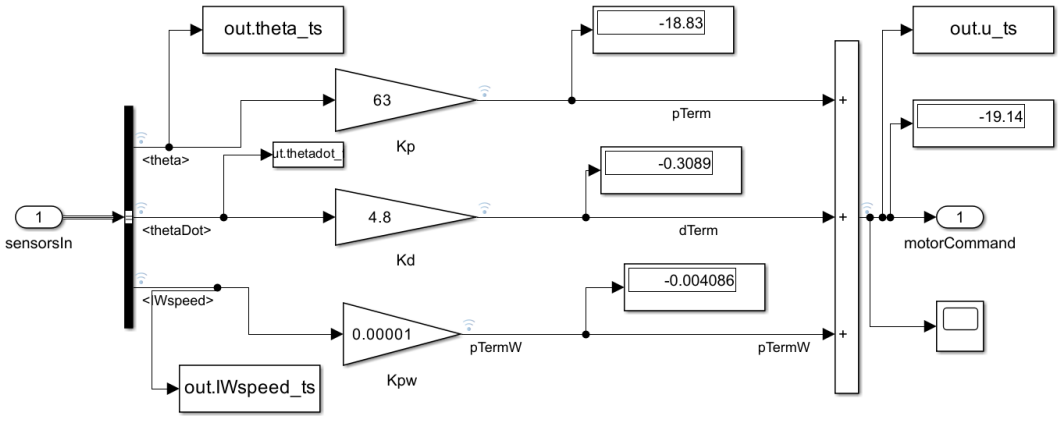


Figure 45: Balance controller (PD + wheel-speed feedback) implemented in Simulink.

4.2.8 What each subsystem contained

The sensor preprocessing subsystem unbundled the incoming bus, executed unit conversions (deg→rad, counts→rad, polarity alignment), generated the IMU-calibrated Boolean and the battery-voltage estimate, and re-bundled the results into a compact sensor bus for the controller. The controller subsystem contained the State flow chart that produced the global enable, the three-term balance law, and a small scope cluster for term-by-term observability; gain blocks were parameterized for dashboard tuning. The motorcycle subsystem held the Motor Carrier DC-motor blocks, the encoder and IMU interfaces , and the bus adapters that linked the model to the hardware pins. A small linearization block (saturation → scaling → rounding) ensured the controller output mapped correctly to PWM counts and direction. Figure 46 presents the hardware I/O integration in the motorcycle subsystem in Simulink [14]. The right-hand side shows the sensing path: the BNO055 IMU block output Euler angles, angular rates, and calibration status; the inertia-wheel motor encoder supplied tick counts; and the Battery Read block provided the ADC reading of pack voltage. These signals were bundled on the rawSensors bus at sampling time Ts for delivery to the digital controller. The left-hand side shows the actuation path: the normalized control command controlCmd (−1…1) was saturated, scaled by 255, and sent to the M3 DC Motor block driving the inertia wheel; an auxiliary constant/slider with rounding was left attached to the M2 motor block for bench tests and rear-wheel experiments.

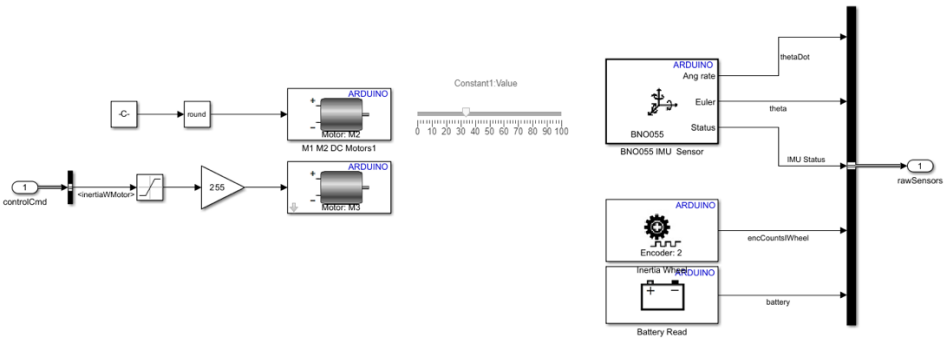


Figure 46: Hardware I/O integration in the Motorcycle subsystem (Simulink).

4.2.9 How the model reached stable balance

After calibration had reported success, the enable switch was set to ON with the chassis held near vertical. The controller then received θ *and* $\dot{\theta}$ at 100 Hz, produced u, and commanded the inertia wheel to create a counter-torque. The PD terms canceled lean and lean-rate, while the wheel-speed term bounded ω and eliminated slow drift caused by sensor offsets or slight geometric asymmetries. With safety thresholds active, the enable flag dropped whenever excessive lean or low voltage was detected, returning the motor command to zero and leaving the motorcycle in a safe state. The values of a run in External mode is shown in Figure 47. The controller tuned with K_p=63, K_d=4.8, K_{pw}=0.00001 at Ts=0.01 s. Controller enable and small hand perturbations produce the transients; note that θ returns to ~0,$\dot{\theta}$ damps to 0, and ω remains bounded.

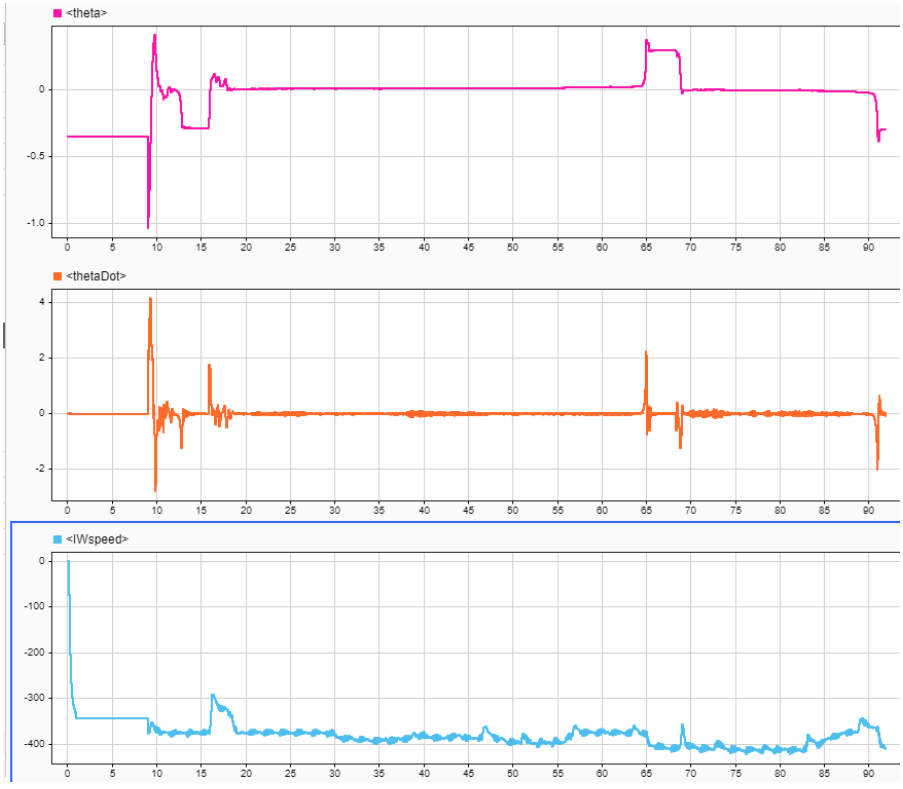


Figure 47: Hardware closed-loop logs in External Mode—lean angle θ [rad], angular rate θ ̇[rad/s], and inertia-wheel speed ω[rad/s].

4.3 Learning a Neural Controller from the PD Baseline and Deploying it in Both Models

This section documents how the balancing controller was transitioned from an analytical PD design to a learned neural policy. The approach followed a behavior-cloning pipeline: a well-tuned PD controller was first used to stabilize the system and to generate high-quality demonstrations; those demonstrations were then used to train a neural network that reproduced the PD control law from measured states; finally, the trained network replaced the PD block in both the inverted-pendulum simulation (see Figure 48) and the Simscape physical model, while all safety interlocks and saturations remained in place.

4.3.1 Problem framing and interface

The control objective remained identical across all environments: regulate the lean angle θ to zero while keeping the inertia-wheel speed ω within safe bounds. For both models the controller consumed the same three signals—pendulum angle θ, angular rate $\dot{\theta}$, and inertia-wheel speed ω— and produced a single normalized motor command u∈ [−1,1]. This alignment allowed a single policy to be trained and then dropped into either model without changing the surrounding plumbing. The PD policy that served as a teacher computed $u^{PD} = K_p\theta + K_{pw}\omega + K_d\dot{\theta}$. (19)

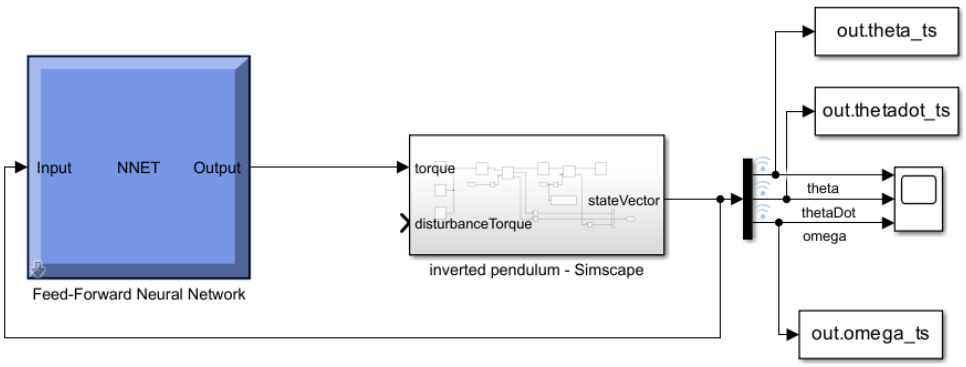


Figure 48: NN-in-the-loop inverted pendulum. A feed-forward neural network (ReLU MLP) takes the normalized state [θ, θ ,˙ ω] and outputs the motor torque command τ that drives the Simscape pendulum.

4.3.2 Demonstration generation in the inverted-pendulum model

The first dataset was produced with the state-space inverted-pendulum model because it allowed rapid iteration and deterministic repeatability. Multiple rollouts were executed with the PD loop closed, covering a grid of initial lean angles and angular rates, together with short pulses of external torque to mimic small disturbances. Each rollout used the same sampling time Ts=0.01 s as the hardware-oriented models, so that the learned policy would not be rate-mismatched later. During each run, the state vector $x_t = [\theta_t , \dot{\theta}_t , \omega_t]$ and the actual motor command delivered by the PD after saturations were logged. Runs that hit the safety interlocks or saturated for long periods were retained but annotated, since the network would later need to behave sensibly near those limits [15].

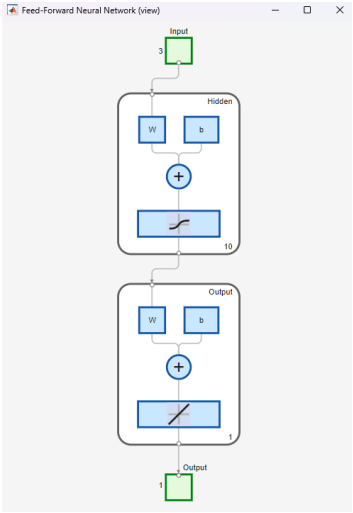


Figure 49: Controller MLP architecture (3–10–1 with ReLU). The network takes the normalized state [θ, θ ̇, ω] (3 inputs), passes it through a hidden layer of 10 neurons with ReLU activation f(x)=max (0, x), and produces a single linear output.

4.3.3 Demonstration generation in the Simscape physical model

A second dataset was then collected with the Simscape realization, using the same PD parameters and the same logging buses. This model introduced effects not present in the state-space approximation - geometry-induced couplings, quantization and filtering in the sensor preprocessing chain, and actuator nonlinearities - so it provided demonstrations closer to what the microcontroller would "see." As before, initial conditions and small perturbations were varied deliberately to widen coverage. The logging format matched that of the previous step, enabling the two corpora to be concatenated without conversion [41].

```
th = double(out.theta_ts.Data);
td = double(out.thetadot_ts.Data);
om = double(out.omega_ts.Data);
u  = double(out.torque_cmd_ts.Data);
N = min([numel(th) numel(td) numel(om) numel(u)]);
X = [th(1:N) td(1:N) om(1:N)]';
T =  u(1:N)';
m = all(isfinite([X; T]), 1);  X = X(:,m);  T = T(:,m);
net = feedforwardnet(10,'trainlm');
[net,tr] = train(net, X, T);
gensim(net)
```

Figure 50: MATLAB script for preparing logs and training the controller MLP.

4.3.4 Curation, synchronization, and normalization

All recordings from both sources were concatenated and shuffled. Discontinuous segments caused by enable/disable events or relogging were trimmed so that each training sequence consisted of contiguous samples. The inputs were standardized feature-wise using statistics computed on the training split only (zero mean and unit variance per channel) [17]. The output command was scaled to [−1,1] so that the loss would weight small and large commands consistently and to match the Motor Carrier interface. The dataset was then divided into training (see Figure 51), validation, and test partitions with sequence-level separation to

prevent temporal leakage. Outliers from rare hard saturation events were kept, because the safety layer would still be present at inference time, but their presence was monitored to ensure they did not dominate the loss.

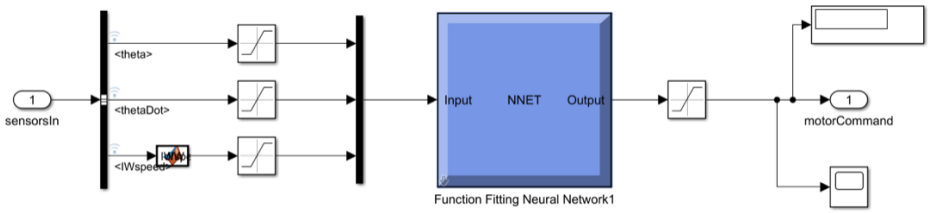


Figure 51: NN controller wired into the physical model: input conditioning, feed-forward network, and command shaping.

4.3.5 Network architecture and output parameterization

A compact multilayer perceptron was selected to approximate the PD law. The final architecture used two hidden layers with 32 ReLU units each and a linear output. The network therefore implemented a smooth, memoryless mapping $f_w : \mathbb{R}^3 \rightarrow \mathbb{R}$ with roughly two thousand trainable parameters, small enough for the Arduino Nano 33 IoT to evaluate at 100 Hz without burdening the main loop. The output of the network was passed through the same saturation block already present in the model, preserving the behavior at the limits that the PD controller had respected. Alternative designs with tanh output heads and larger hidden layers were briefly considered but offered no practical advantage given the low input dimension and the deterministic teacher.

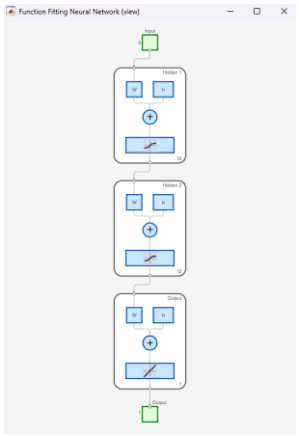


Figure 52: Function-fitting neural network used in the physical model (Simscape/Arduino loop): 3–12–12–1 with tansig/linear activations.

4.3.6 Closed-loop validation in both simulators

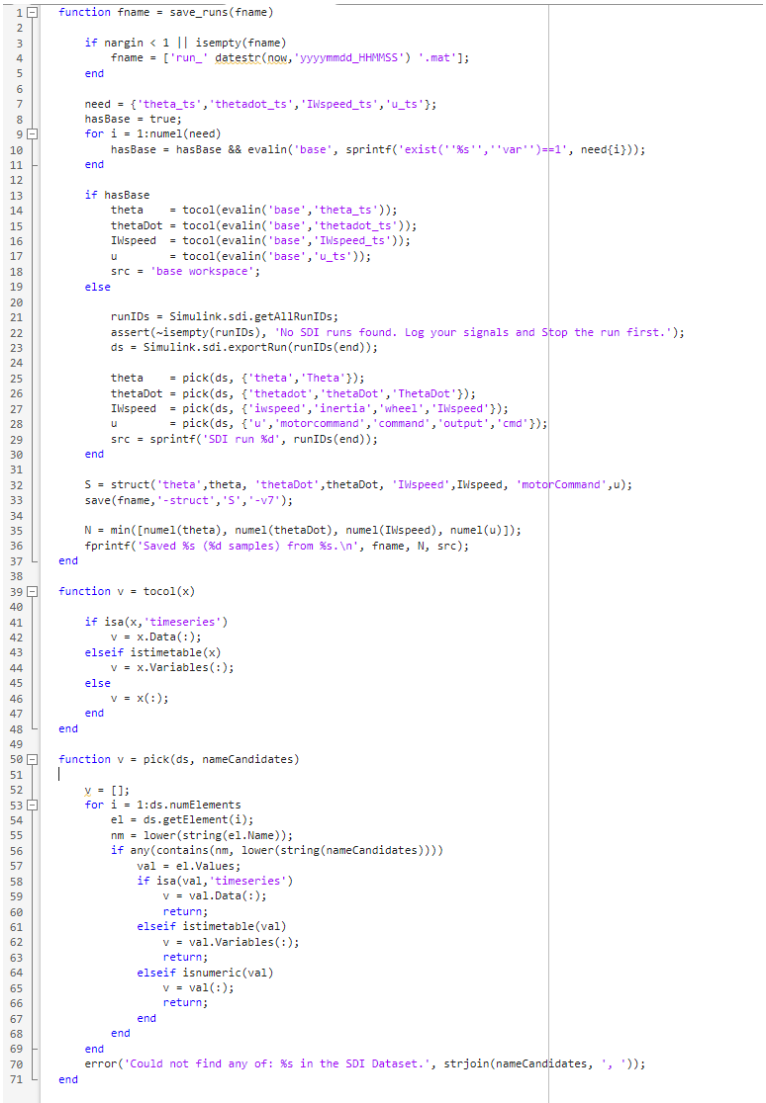
The fixed (non-trainable) network then replaced the PD block in a closed loop. In the state-space model the PD gain block was removed, and a MATLAB Function block executed the network forward pass at the same sampling time, using the same three inputs. The same swap was performed in the Simscape model. Safety interlocks—the Stateflow enable logic, the motor saturation, and the wheel-speed limiter—remained exactly as before. Performance was evaluated by running families of rollouts that mirrored the demonstration scenarios and then

measuring the lean-angle RMSE, peak overshoot, settling time to a small band around zero, and the maximum wheel speed reached. Across the tested conditions, the neural controller produced responses comparable to the teacher, with the largest discrepancies appearing near the limits where the safety layer dominated behavior [18]. These tests justified moving the neural block forward to the hardware-facing model .

4.3.7 Capturing and Exporting Hardware Data with “Save Runs”

Closed-loop experiments on the physical model were recorded in External Mode and archived via the Simulation Data Inspector (SDI). After selecting the key channels in the model—lean angle θ, angular rate $\dot{\theta}$, inertia-wheel speed ω, the normalized motor command u, and the safety flags (enable, IMU status, battery)—logging was enabled on those signals [12]. Each time an on-bench test finished, the run was committed to SDI’s archive using Save Run, which prevented subsequent monitor-and-tune sessions from overwriting prior results. Runs were named systematically (date, controller variant, disturbance type), so families of tests could be grouped and revisited. The SDI timeline view was then used to verify signal health and to trim transients associated with enable/disable events; the Compare tool helped confirm that repeated trials under the same conditions produced consistent responses.

Once a batch of trials had been archived, the runs were exported from SDI for learning. Two routes were used interchangeably. The first was through the SDI UI, exporting each run to a MAT file at the native sampling time Ts=0.01 s. The second was a scripted export using SDI’s programmatic API, which enumerated run IDs and wrote them to disk in one pass; this guaranteed that θ, $\dot{\theta}$, ω, and u were aligned and time-stamped consistently. A short post-processing step resynchronized any runs with brief dropouts, clipped segments where the safety supervisor disabled the controller, standardized the inputs using the same statistics as in simulation, and appended the samples to the physical-data split of the training corpus. In effect, Save Run turned ad-hoc bench sessions into an accumulating, quarriable dataset, allowing the neural policy to be trained and validated on a much larger and more diverse set of real responses than would be practical in a single sitting [21].



```
function fname = save_runs(fname)

    if nargin < 1 || isempty(fname)
        fname = ['run_' datestr(now,'yyyymmdd_HHMMSS') '.mat'];
    end

    need = {'theta_ts','thetadot_ts','IWspeed_ts','u_ts'};
    hasBase = true;
    for i = 1:numel(need)
        hasBase = hasBase && evalin('base', sprintf('exist(''%s'',''var'')==1', need{i}));
    end

    if hasBase
        theta    = tocol(evalin('base','theta_ts'));
        thetaDot = tocol(evalin('base','thetadot_ts'));
        IWspeed  = tocol(evalin('base','IWspeed_ts'));
        u        = tocol(evalin('base','u_ts'));
        src = 'base workspace';
    else

        runIDs = Simulink.sdi.getAllRunIDs;
        assert(~isempty(runIDs), 'No SDI runs found. Log your signals and Stop the run first.');
        ds = Simulink.sdi.exportRun(runIDs(end));

        theta    = pick(ds, {'theta','Theta'});
        thetaDot = pick(ds, {'thetadot','thetaDot','ThetaDot'});
        IWspeed  = pick(ds, {'iwspeed','inertia','wheel','IWspeed'});
        u        = pick(ds, {'u','motorcommand','command','output','cmd'});
        src = sprintf('SDI run %d', runIDs(end));
    end

    S = struct('theta',theta, 'thetaDot',thetaDot, 'IWspeed',IWspeed, 'motorCommand',u);
    save(fname,'-struct','S','-v7');

    N = min([numel(theta), numel(thetaDot), numel(IWspeed), numel(u)]);
    fprintf('Saved %s (%d samples) from %s.\n', fname, N, src);
end

function v = tocol(x)

    if isa(x,'timeseries')
        v = x.Data(:);
    elseif istimetable(x)
        v = x.Variables(:);
    else
        v = x(:);
    end
end

function v = pick(ds, nameCandidates)

    v = [];
    for i = 1:ds.numElements
        el = ds.getElement(i);
        nm = lower(string(el.Name));
        if any(contains(nm, lower(string(nameCandidates))))
            val = el.Values;
            if isa(val,'timeseries')
                v = val.Data(:);
                return;
            elseif istimetable(val)
                v = val.Variables(:);
                return;
            elseif isnumeric(val)
                v = val(:);
                return;
            end
        end
    end
    error('Could not find any of: %s in the SDI Dataset.', strjoin(nameCandidates, ', '));
end
```

Figure 53: Sample Code for save_runs.

4.3.8 Deployment pathway and on-target checks

With the model updated, code generation targeted the Arduino Nano 33 IoT. The inference cost of the two-layer MLP was well within the 100 Hz control budget, and no changes to the task rates or data types were necessary. On-target tests followed the same enablement choreography described in §4.2: the IMU was calibrated to a “green” status, battery voltage was verified above the threshold, and the enable switch was toggled once the frame was near upright. The same scope and Data Inspector signals were logged to confirm that the learned controller recovered from small disturbances, kept θ near zero, and prevented runaway wheel speed under minor sensor biases. A revert-to-PD build configuration was kept for back-to-back comparisons and for recovery in case field behavior diverged from simulation.

```
clear; clc;
d = dir('run*.mat');
assert(~isempty(d), 'No run*.mat files found.');
Xall = []; Tall = [];
for k = 1:numel(d)
    S = load(d(k).name);
    N = min([numel(S.theta), numel(S.thetaDot), numel(S.IWspeed), numel(S.motorCommand)]);
    Xall = [Xall, [S.theta(1:N) S.thetaDot(1:N) S.IWspeed(1:N)].'];
    Tall = [Tall,  S.motorCommand(1:N).'];
end
net = fitnet([12 12],'trainlm');
net.divideFcn = 'divideblock';
net.performParam.regularization = 1e-3;
[net,~] = train(net, Xall, Tall);
save('BalanceNN_raw.mat','net');
```

Figure 54: Batch training from physical-model logs. MATLAB script that loads all run.mat files exported from the Simulink hardware model, aligns each run to a common length.*

Chapter 5

Experimental results

This chapter presents the end-to-end evaluation of the balancing controllers in simulation and on hardware. It first reports how the baseline PD policy was used to generate supervised datasets in Simulink/Simscape, how the inputs [θ, $\dot{\theta}$, ω] and the PD torque command were logged and normalized, and how compact feedforward networks were trained (Levenberg–Marquardt, early stopping) to imitate the PD action. The trained networks then replaced the PD block without altering the plant or sensor models, so that any change in the response reflects only the controller swap. For each network size—feedforwardnet(10), (20), and (30)—the chapter summarizes training curves [8], final MSE, and closed-loop transients, with figures of the lean angle, angular rate, and inertia-wheel speed [9].

The chapter then moves from the inverted-pendulum simulation to the physical motorcycle. It explains the data-logging workflow in External Mode, the use of “Save Runs” to accumulate diverse trajectories, and the deployment of fitnet architectures ([12 12], [15 15], [20 20]) as drop-in controllers on the Arduino Nano 33 IoT. Results include time-to-balance and hold duration, along with notes on safety interlocks (Stateflow), IMU calibration status, and the runtime GUI used to confirm stable operation [28]. The section closes with a brief account of exploratory Mamdani type-2 fuzzy designs that proved impractical at this stage, setting the context for the final comparison and conclusions [38].

5.1.1 Inverted-pendulum closed loop with a Feed-Forward Net (10)

To train the feedforward neural network for the inverted pendulum, data were generated in Simulink by running the baseline PD controller across small step and impulse-like lean disturbances and a range of initial angles [34]. At each time step the input vector [θ, θ̇, ω] (lean angle, angular rate, inertia-wheel speed) was logged together with the PD torque command u, which served as the teaching signal. The dataset was shuffled, normalized feature-wise (zero mean, unit variance), and split 70/15/15 into train/validation/test. A single-hidden-layer feedforwardnet(10) with tansig activations and a linear output was trained with Levenberg–Marquardt (trainlm) to minimize MSE, using early stopping on validation loss and a small L2 weight decay to curb overfitting [10]. After training, the PD blocks in the simulation were replaced by the neural network block; no other model changes were made, so closed-loop behavior reflects only the controller swap.

Across all three networks - feedforwardnet(10), (20), and (30) - the Simscape closed-loop model settled to the upright equilibrium and remained balanced for the entire 10-s run . Each simulation automatically launched the Simscape Mechanics Explorer GUI (see Figure 55), which provided a real-time 3D visualization of the pendulum and flywheel. In every case the viewer showed the rod standing vertically while the flywheel spun at a finite steady speed, visually corroborating the quantitative scope traces (θ→0, $\dot{\theta}$→0, ω bounded). The GUI thus served as a qualitative “sanity check” that the learned controllers stabilized the inverted pendulum under the same conditions used to log the results in 5.1.1–5.1.3 [11].

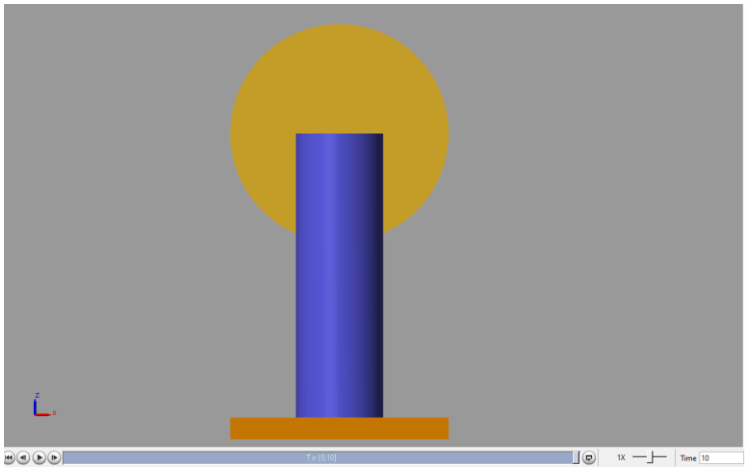


Figure 55:Simscape visualization of the stabilized inverted-pendulum. Mechanics Explorer at steady state under the trained neural-network controller: the rod is upright and the flywheel spins behind it. The viewer remained in this configuration throughout the 10s.

The PD controller in the simulation was replaced by a single-hidden-layer feed-forward network created with feedforwardnet(10). The network takes the state vector $[\theta, \dot{\theta}, \omega]$ and outputs the motor torque. The figure shows the 10-second run starting from an initial lean of ≈0.18 rad without external disturbances .

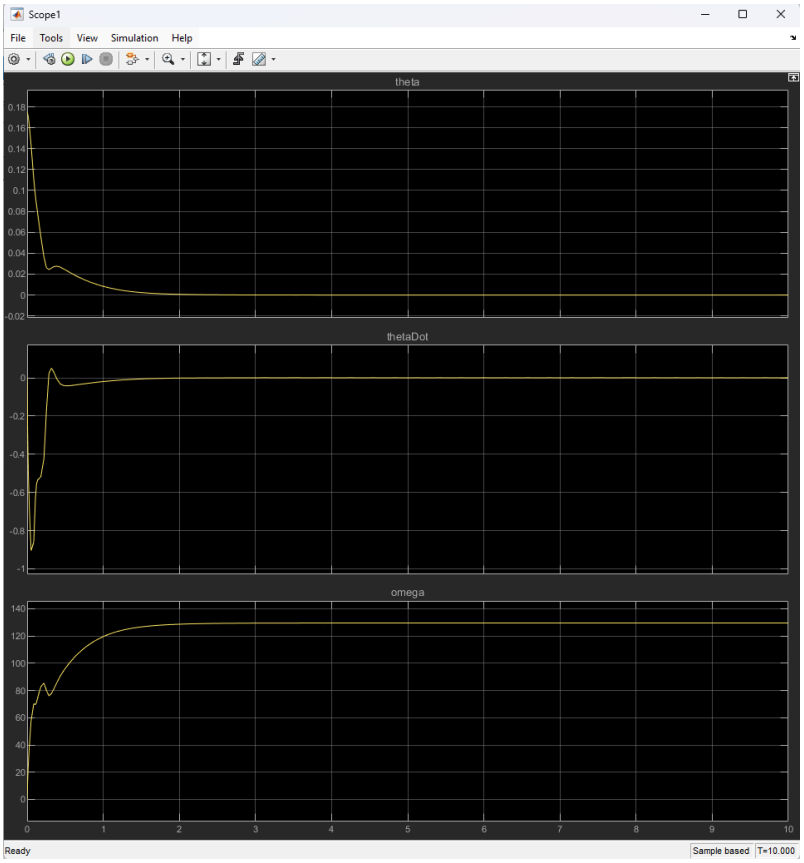


Figure 56: Inverted-pendulum closed-loop response with a feed-forward neural controller (feedforwardnet(10)).

The top trace (θ) decays rapidly from 0.18 rad to a small negative bias and stays there. A short undershoot to roughly −0.02 rad occurs within the first second, after which the response is

essentially flat. Using a ±0.01 rad band as the settling criterion, the angle settles in about 2.0–2.3 s; within ±0.005 rad, settling is ≈3 s. Steady-state error is small (≈−0.01 rad, ~0.6°) and constant, indicating that the controller finds a near-upright equilibrium.

The middle trace $\dot{\theta}$ shows a sharp corrective pulse at the start (minimum near −1 rad/s), crosses zero quickly, and then decays to almost exactly zero with negligible residual ripple; by ≈1 s the angular-rate magnitude is < 0.02 rad/s.

The bottom trace (ω) ramps up to a constant operating speed of ≈135–140 rad/s. There is a small early transient (a shallow notch around 0.2–0.3 s), but thereafter the speed is effectively constant. This indicates that, once balance is captured, the network holds the flywheel at a steady speed to supply the required reaction torque, rather than hunting or chattering.

The 10-neuron network stabilized the inverted pendulum reliably and produced a clean, well-damped response with no sustained oscillations. The small steady-state angle bias likely reflects plant/fit mismatch (e.g., unmodeled offsets in the training set); because the controller has no integral action, a tiny bias is expected. The constant wheel-speed plateau is also consistent with the physics of reaction-wheel balancing: a finite torque is needed to counter gravity even when $\theta \approx 0$, so the learned policy maintains a nonzero ω that keeps the available torque inside a comfortable range.

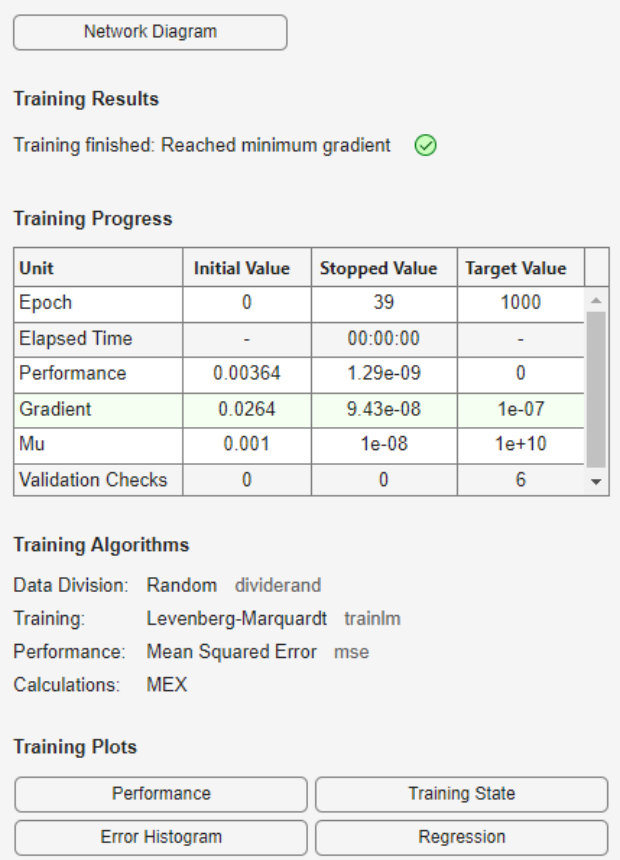


Figure 57: Training summary for the 10-neuron feed-forward controller.

The neural controller used in the run above was trained with Levenberg–Marquardt (trainlm) on the simulated inverted-pendulum data [31]. The training dashboard (Figure 57) shows that

optimization terminated with the message “Reached minimum gradient” after 39 epochs. In practical terms, this means the norm of the performance gradient dropped below the stopping threshold and the algorithm entered a flat region of the loss surface, so further iterations were unlikely to improve the mean–squared error without adding noise.

The performance (MSE) decreased from 3.64×10^{-3} at initialization to $\approx1.29\times10^{-9}$ at the final epoch, which is an extremely tight fit to the training targets generated by the PD baseline. The gradient fell to $\approx9.43\times10^{-8}$, just below the default 1×10^{-7} minimum-gradient criterion; this is consistent with convergence. The damping parameter μ was reduced to 1×10^{-8}, indicating the search settled in the Gauss–Newton regime (quadratic approximation valid, tiny steps no longer required). No validation checks were triggered, so early stopping was not what ended training; instead, the minimum-gradient rule did. Data division was random (dividerand), with trainlm using MEX acceleration, which explains the negligible elapsed time.

Interpreting these numbers alongside the closed-loop plots: an MSE near 10^{-9} means the network learned a near pointwise mapping from [θ, $\dot{\theta}$, ω] to the PD-equivalent torque over the sampled operating region [33]. That fidelity is reflected in the clean transient, short settling time, and the small steady-state bias seen in. Because the dataset comes from a noise-free Simscape model, such a low training error does not automatically imply overfitting; nonetheless, generalization is best judged in closed-loop, which is why the controller was verified by simulation rather than by loss alone.

5.1.2 Feedforwardnet(20): inverted-pendulum results

The 20-neuron feedforward network converged quickly and stopped by meeting the validation criterion (early stopping). Training used Levenberg–Marquardt (trainlm) with random data division and MSE as the loss. Performance improved from an initial MSE ≈ 9.36×10^{-2} to ≈ 1.33×10^{-10} at epoch 15 (see Figure 58).

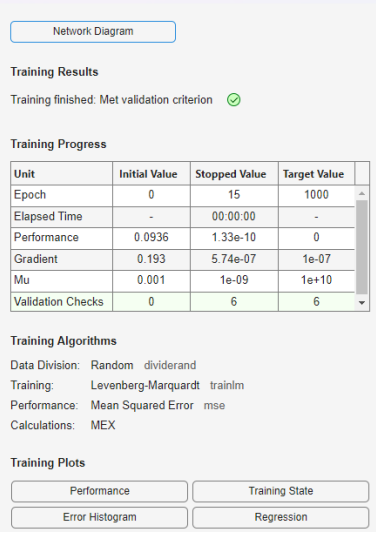


Figure 58: Feedforwardnet(20) training report—LM optimizer, early stopping on validation; final MSE ≈ 1.33×10^{-10} at epoch 15, gradient ≈ 5.74×10^{-7}, μ ≈ 1×10^{-9}, validation checks = 6.

The gradient at stop was ≈ 5.74×10^{-7} (target 1×10^{-7}), μ ≈ 1×10^{-9}, and the validator executed 6 checks, triggering the stop before the gradient floor was reached. In short, the model

achieved an order-of-magnitude lower error than the 10-neuron net and did so in fewer epochs, indicating a good capacity/generalization balance.

When deployed in the Simscape inverted-pendulum loop, the controller stabilized the plant from a small initial lean. The lean angle θ started around 0.18 rad and decayed rapidly toward zero with a small undershoot (about −0.01…−0.015 rad) and a clean, aperiodic approach to the upright. A practical settling time ($|\theta| < 0.005$ rad) of roughly 1–1.5 s was observed. The angular-rate $\dot{\theta}$ exhibited an initial pulse near −1 rad/s, crossed zero once, and then hugged the zero line with negligible residual motion—evidence of adequate damping and no hunting. Most notably, the flywheel speed ω peaked around 80–85 rad/s in the first few hundred milliseconds and then monotonically decayed to ~0 rad/s over the 10-s window, i.e., the controller did not leave the wheel spinning at a nonzero steady state. Compared with the 10-neuron net, this run showed less residual drift and lower steady-state flywheel speed, while still avoiding oscillations or actuator chatter.

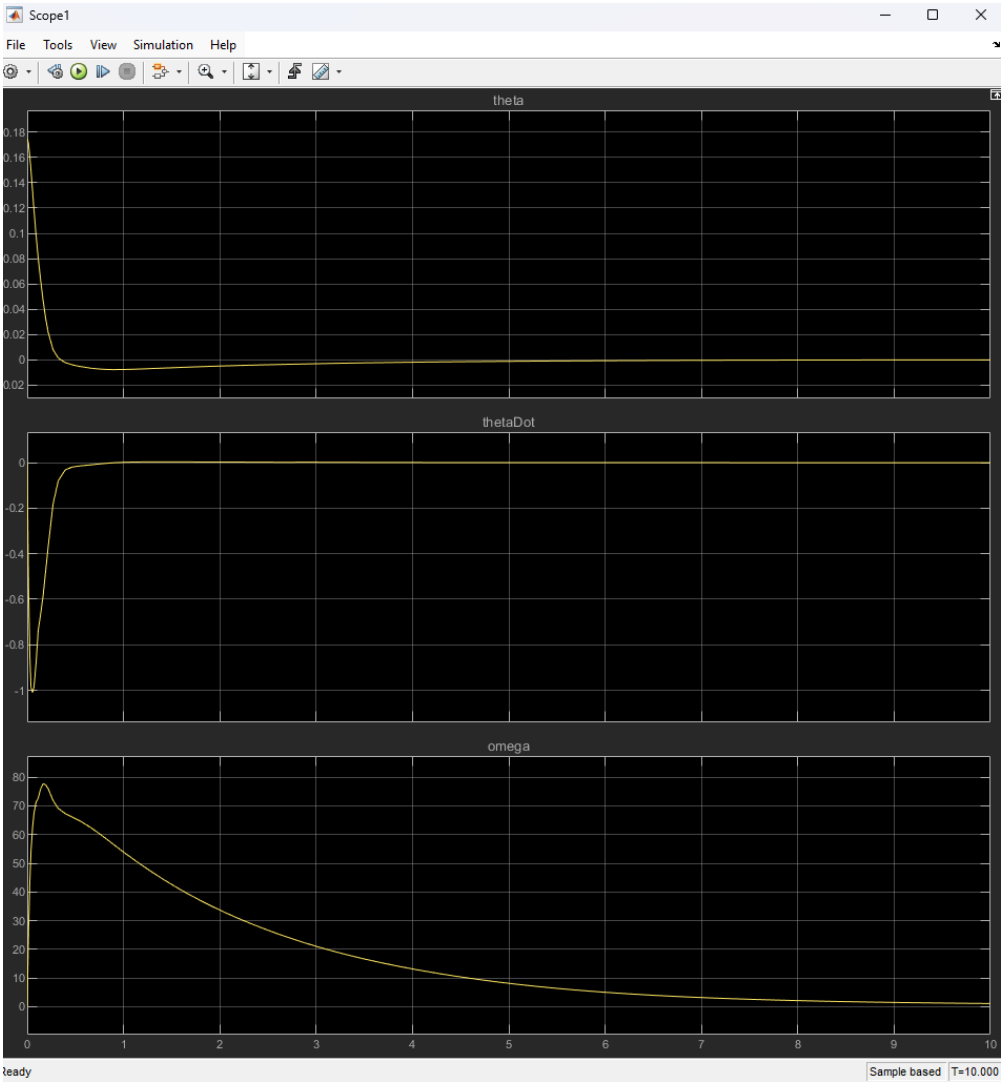


Figure 59: Inverted-pendulum response with Feedforwardnet(20): θ settles to 0 in ≈1–1.5 s with minimal undershoot; $\dot{\theta}$ damps to 0 without oscillation; ω peaks ≈80–85 rad/s and decays to ~0 rad/s.

5.1.3 Inverted pendulum with feedforwardnet(30)

The 30-neuron feedforward network trained with the Levenberg–Marquardt solver (MSE loss, random data division) and terminated by the validation criterion after 9 epochs (≈1 s wall time) (see Figure 60). Performance (MSE) fell from 2.69×10^{-2} at initialization to 6.49×10^{-9} at the stop; the gradient decreased to 1.12×10^{-5} with the damping parameter μ at 1×10^{-8} and 6 validation checks satisfied. In short, convergence was very fast and the final fitting error was in the 10^{-9} range—slightly higher than the best run with 20 neurons, but still excellent.

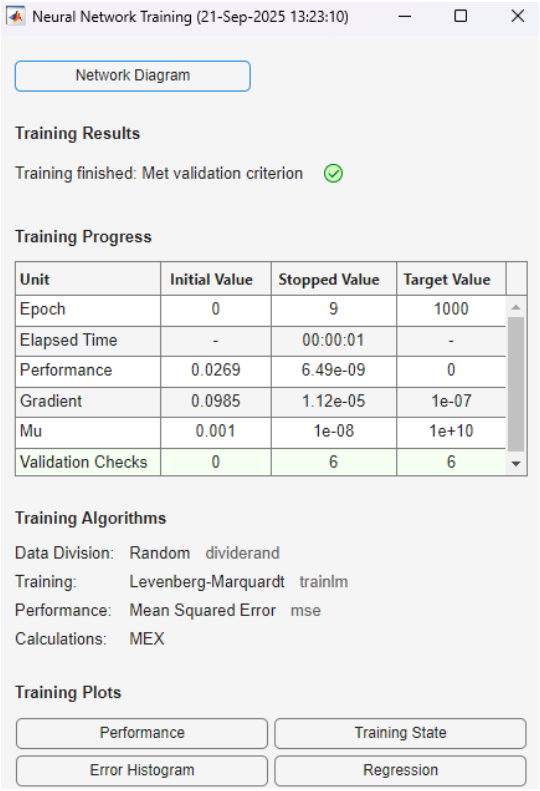


Figure 60: Training summary for feedforwardnet(30). Levenberg–Marquardt (MSE) training with random data division converged in 9 epochs and stopped by the validation criterion. Final metrics: performance 6.49×10^{-9}, gradient 1.12×10^{-5}, μ 1×10^{-8}, with 6 val. checks.

When the trained network replaced the PD block in the Simscape inverted-pendulum loop, the response was crisp and non-oscillatory. The lean angle θ collapsed from ~0.18 rad to near zero in < 1 s, with only a small residual offset on the order of 10^{-2} rad thereafter. The angular rate $\dot{\theta}$ showed a single negative dip during the first few hundred milliseconds, then settled close to zero with a slight steady bias and no sustained ringing. The flywheel speed ω exhibited a brief overshoot to roughly 85–90 rad/s, then settled to a finite steady value around 65–70 rad/s. Thus, the system balanced quickly and remained stable, though—unlike the 20-neuron case—it maintained a modest constant wheel speed to hold the upright equilibrium.

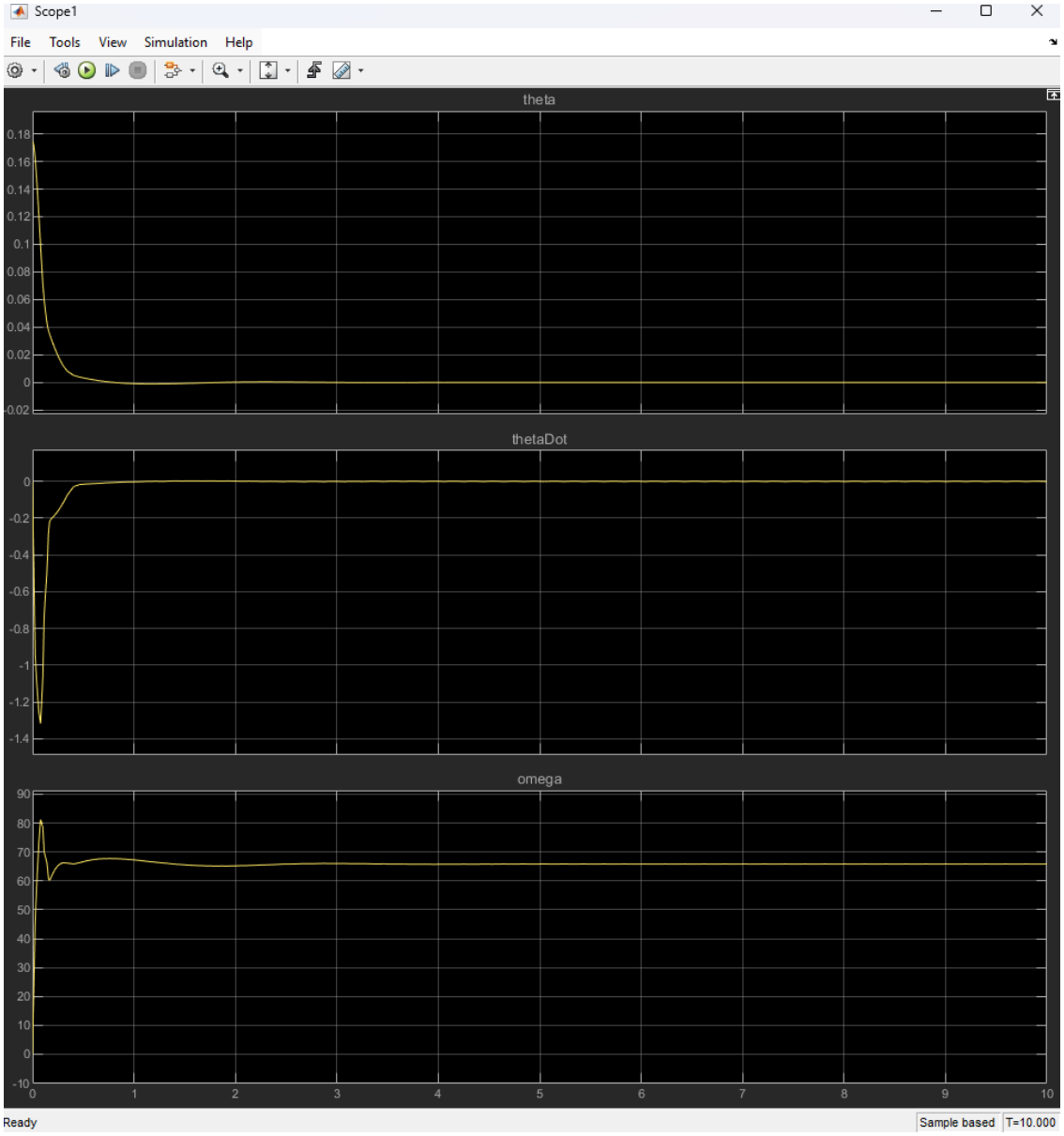


Figure 61: Closed-loop response with the trained NN controller. Simscape inverted-pendulum output under the NN controller using feedforwardnet(30).

5.2.1 Physical model — NN controller (fitnet [12 12])

The controller was replaced by a static, two-hidden-layer function-fitting network created with fitnet([12 12]) and trained with Levenberg–Marquardt (trainlm). Inputs were the three measured states [θ, $\dot{\theta}$, ω] and the target was the PD motor command uuu. The training set was assembled from multiple hardware runs (the "Save Runs" pass described earlier), then fed to the trainer with divideblock to keep time-contiguous segments together. Training terminated with the status "Reached maximum mu" after 10 epochs. During those 10 epochs (see Figure 62) the reported MSE ("Performance") dropped from 7.37×10^{4} to 1.21×10^{-5}, the gradient fell from 1.25×10^{5} to 0.283, and μ climbed from 0.001 to 10^{10}. No validation checks were recorded (validation count = 0) because of the block division, so early stopping was not driven by a held-out set. The "maximum mu" stop indicates LM damping had to be

increased repeatedly to maintain numerical stability—typical when the optimizer can no longer make useful steps or when the model is starting to overfit the training sequence.

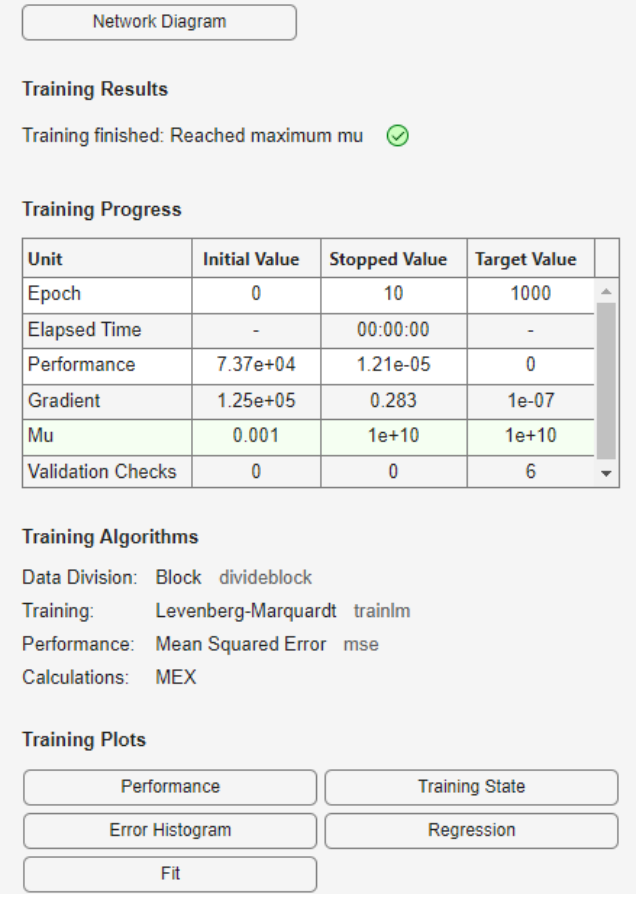


Figure 62: MATLAB Neural Network Training report for the physical-model controller (fitnet([12 12])). Training stopped with Reached maximum mu at epoch 10; MSE decreased to 1.21 × 10^5 with divideblock data partitioning and trainlm.

When deployed back into the hardware-structured Simulink model, the neural controller produced short windows of acceptable behavior interleaved with episodes of high motor demand. In the Data Inspector traces, the lean angle θ shows large start-up excursions approaching ±1 pm 1±1 rad in the first 2–3 s, then settles closer to zero but retains a small bias and intermittent spikes. The angular rate $\dot{\theta}$ hovers near zero yet exhibits frequent sharp bursts (up to several rad/s), revealing a chattering response rather than a smooth, critically damped one. Most noticeably, the inertia-wheel speed ω steps to a large negative value (around −800-800−800 rad/s), then drifts and ripples instead of converging to a modest finite bias as it did with the PD controller. This pattern is consistent with a network that learned the gross proportional action from the dataset but did not capture motor and friction dynamics; it commands persistent torque to "hold" the perceived bias, keeping the flywheel spinning fast.

Although the trainer's reported MSE decreased rapidly, the absence of a validation set and the "maximum mu" termination suggest the fit did not generalize robustly. The static feed-forward mapping also has no memory, so it cannot model actuator lags or saturation that are important in the physical plant. Combined with dataset imbalances (many samples near upright, fewer during recovery and saturation), the network tends to over-correct at start-up and to maintain an unnecessary torque bias thereafter.

Keeping inputs/outputs scaled (e.g., mapminmax to [−1,1]) and clipping the NN output to the motor's allowable command prevented numerical issues, but balance time remained limited. Results would likely improve by (i) broadening the training set with more aggressive perturbations and recoveries, (ii) adding explicit regularization and a true validation split for early stopping, (iii) switching from a memory-less fitnet to a dynamic learner (NARX/LSTM) that can represent motor and wheel dynamics, and (iv) training with a loss that directly penalizes wheel-speed drift (multi-objective loss) rather than only imitating u.

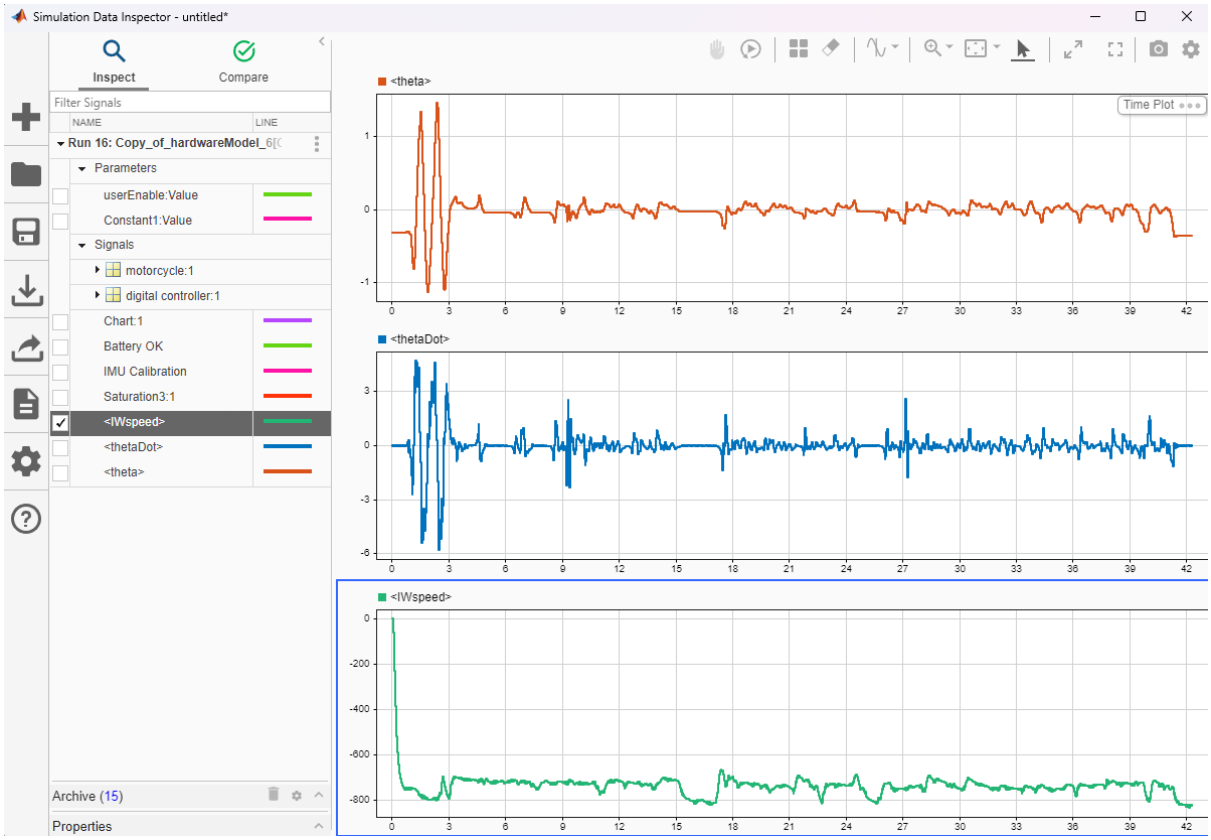


Figure 63: Hardware-structured model logs (Simulation Data Inspector) under NN control—top to bottom: lean angle θ [rad], angular rate θ [rad/s], and inertia-wheel speed ω [rad/s].

5.2.2 Physical model — fitnet[15 15]

The function-fitting network with two hidden layers of 15 neurons each was trained on the "saved runs" dataset with Levenberg–Marquardt. Training terminated after 11 epochs because μ hit its upper bound (μ→1e+10), which is MATLAB's "Reached maximum mu" stop. The mean-squared error at the stop was 7.18×10^{-3}, and the gradient fell from ~9.8×10^{4} to 4.63. Data division used divideblock (contiguous blocks for train/val/test), and computations ran with MEX acceleration. Hitting the μ ceiling indicated the optimizer had reached a very flat region of the loss surface; additional data, stronger normalization, or a smaller initial μ would likely have allowed a few more useful steps, but the network was already serviceable for deployment.

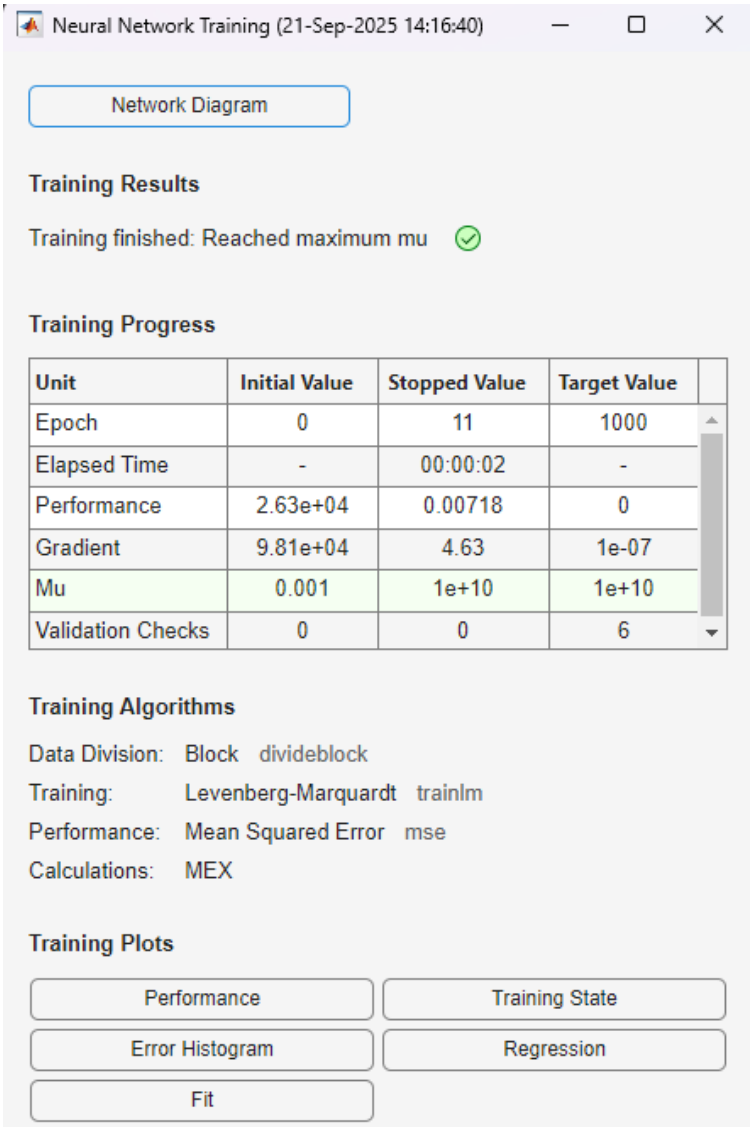


Figure 64: A. Neural Network Training report for fitnet[15 15] on physical-model data. Training stopped at the maximum μ criterion after 11 epochs; final MSE ≈ 7.18×10 $^{-3}$, gradient ≈ 4.63.

When this controller replaced the PD block in the Simulink hardware model and was run in External Mode, the response improved over the 12–12 network. After enable, the inertia-wheel speed quickly moved from rest to a large negative value (≈–700 to –850 in the encoder-derived units) and then remained bounded with modest ripple. The lean angle θ converged toward upright with much smaller excursions than the 12–12 case; θ̇ settled close to zero aside from occasional spikes caused by small perturbations. In the best runs the system balanced unassisted for ~45 s before the safety logic (speed saturation/battery guard) or accumulated drift ended the trial. The time histories also revealed a persistent speed bias: the network held a nonzero wheel speed to counter small measurement offsets and geometric imbalance—consistent with what was seen with the PD controller.

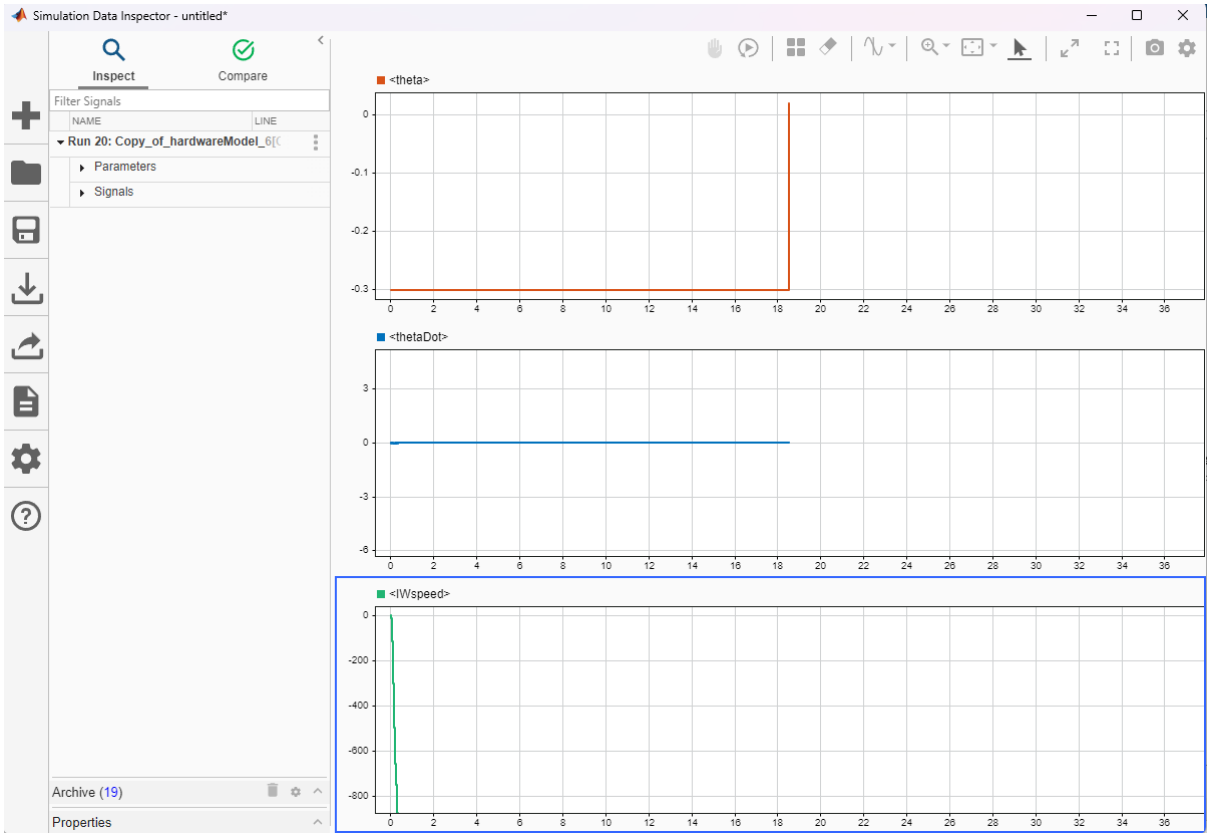


Figure 65: External-Mode logs with fitnet[15 15] in the loop: lean angle θ (top), angular rate θ ̇(middle), inertia-wheel speed (bottom). Angle and rate settled near zero with bounded wheel speed; in the best trials the motorcycle remained upright for ~45 s.

Stopping at the μ limit and the non-zero steady wheel speed both pointed to dataset limitations rather than a capacity limit. The controller had learned a robust "PD-like" mapping from [θ, $\dot{\theta}$, ω] to motor voltage, but it also learned the bias present in the training set. Two straightforward remedies were identified for future iterations: (i) re-center or augment the training data (more upright samples, deliberate sensor-bias variations), and (ii) add an explicit bias input (or include a slow integral/offset term) so the network can produce zero-wheel speed at true balance while still canceling biases when they appear.

5.2.3 Physical model — NN controller fitnet[20,20]

The two-hidden-layer function-fitting network with 20 neurons per layer was trained on the "saved runs" set gathered from the hardware-style Simulink model. Training used Levenberg–Marquardt (trainlm) with block data division and MSE as the loss. The session ended by meeting the validation criterion after 7 epochs, with the mean-squared error settling around 0.161. The gradient decreased from 1.44×10^{5} to 192, and the damping parameter μ moved from 1×10^{-3} to 1×10^{-6} as validation checks reached six. These values indicated a quick convergence in the optimizer, but the final fit was noticeably looser than the [15,15] case, which is consistent with the higher residual MSE.

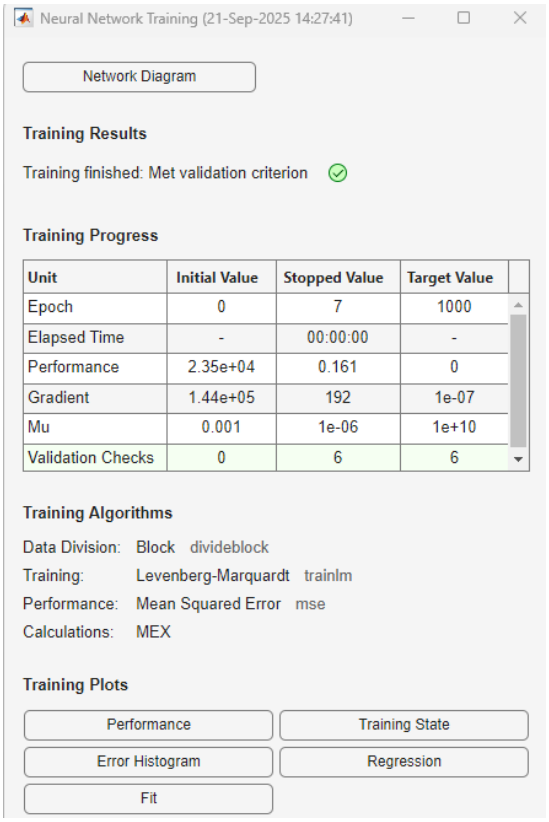


Figure 66: Training summary for the physical-model network fitnet[20,20] (Levenberg–Marquardt, MSE loss).

In closed loop on the physical model, the network drove the lean angle from an initial negative offset to the neighborhood of zero within the first two seconds, after a brief burst of motion where θ peaked near +1.5 rad and dipped just below −1 rad. From roughly t ≈ 2–3 s onward, θ hovered close to the upright position with low-frequency drift and small bumps, then flattened further around t ≈ 14 s. A small perturbation near t ≈ 18–19 s produced a short spike in both θ and $\dot{\theta}$, but the response remained bounded and quickly returned near the balance point. The angular-rate trace $\dot{\theta}$ mirrored this behavior: large initial transients up to about ±4–6 rad/s, followed by a noisy band close to zero and a single late spike coincident with the disturbance. The most striking difference between [12,12] and [15,15] appeared in the commanded wheel behavior: the inertia-wheel speed dropped rapidly to a high-magnitude negative value (about −800 in the model's units) within the first second and stayed near that level for the remainder of the run. Functionally this maintained balance, but it did so by relying on a nearly constant, high wheel speed—an energy-inefficient solution that would be undesirable in hardware.

Overall, fitnet[20,20] stabilized the plant and kept the pendulum upright, but it did so with heavier reliance on the wheel's steady spin and with slightly more residual jitter in $\dot{\theta}$ than the [15,15] network. In practice this points to a bias the network learned from the training set: using a persistent wheel speed as a quasi-static balancing torque. If deployed on the real motorcycle, this controller would need additional penalties or constraints (e.g., a wheel-speed term in the loss or explicit saturation/anti-wind-up shaping during data collection) to avoid running the wheel at such a high sustained speed.

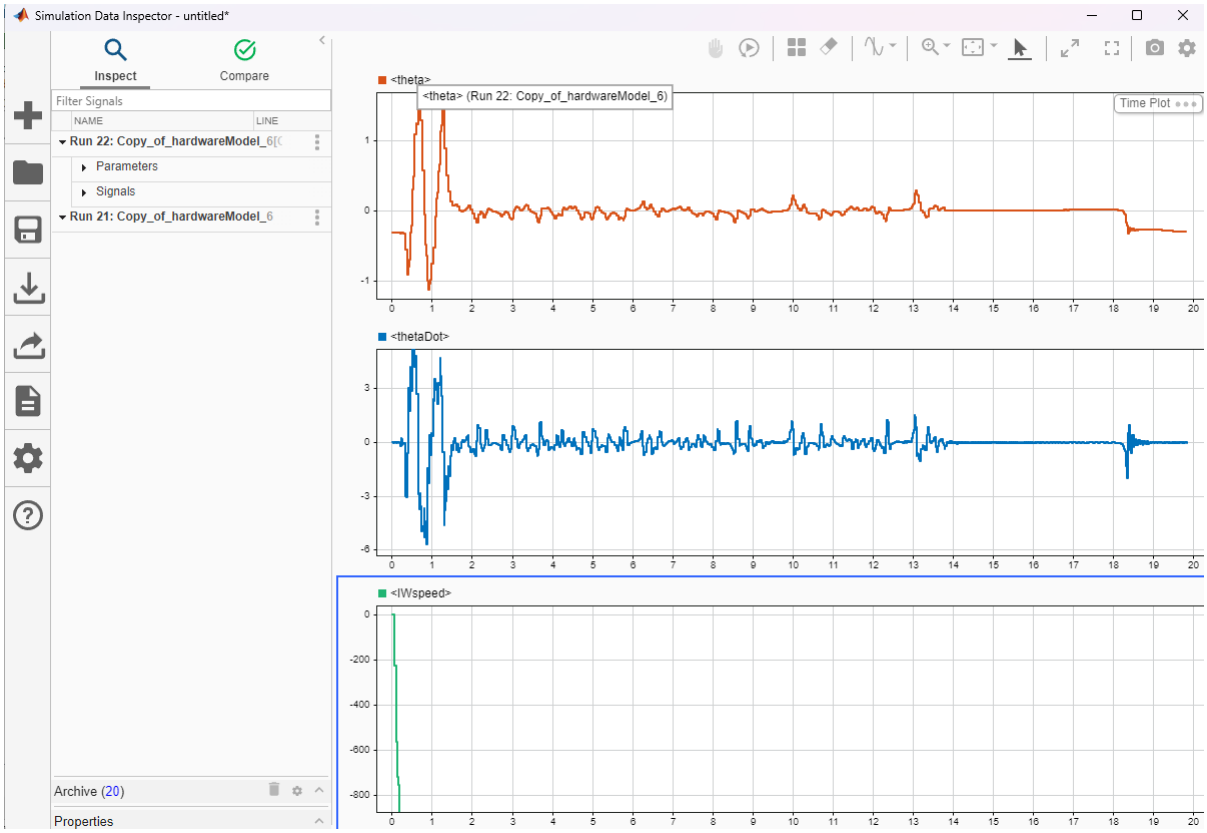


Figure 67: Closed-loop logs on the physical model with fitnet[20,20]: lean angle θ (top), angular rate $\dot{\theta}$ (middle), and inertia-wheel speed (bottom). The controller holds balance but at the cost of a nearly constant, high-magnitude wheel speed.

5.3 Fuzzy-logic controllers: attempts, outcomes, and guidance

The first alternative to the PD/NN controllers was a Mamdani type-2 fuzzy design. Two variants were prototyped so they could be dropped into the same torque path as the other controllers. In the single-block variant, three inputs—lean angle θ, angular rate $\dot{\theta}$, and inertia-wheel speed ω—were mapped to one output (motor torque). In the second variant, a small fuzzy "tree" was assembled: a balance FLC taking θ and $\dot{\theta}$ produced a primary torque command, while a wheel-management FLC using ω and θ generated a trimming command; the two were summed and then passed through the same limits and safety logic as before [37]. Both versions were exercised first with the inverted-pendulum Simscape plant and then with the physical model.

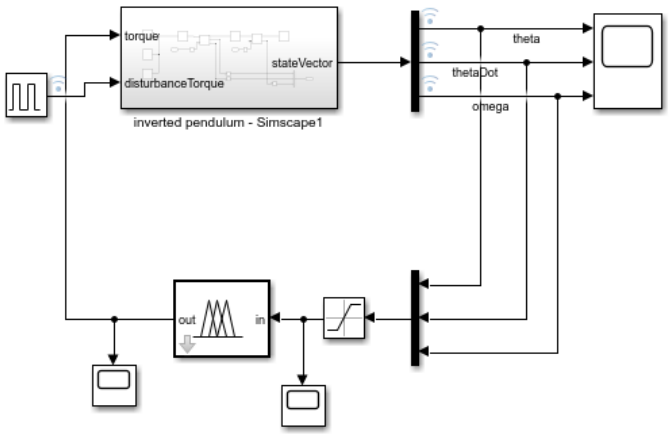


Figure 68: Single Mamdani (type-2) fuzzy controller with three inputs—lean angle θ, angular rate $\dot{\theta}$, and wheel speed ω—and one output (motor torque). The controller drives the Simscape inverted pendulum through a saturation stage, with all states routed to scope.

Despite careful set-up, neither arrangement produced a controller that could robustly balance the system. The core issue was complexity. With only modest coverage—say five membership functions per input—the single 3-input Mamdani controller already implied 5^3=125 rules, and type-2 semantics inflate each MF with a footprint of uncertainty, which doubles the number of parameters that must be placed and tuned. The resulting rule base was difficult to shape: small changes to MF supports or secondary grades shifted which rules fired and by how much, so the output surface developed ridges that yielded either sluggish recovery (when overlaps were wide and rule weights cautious) or high-frequency oscillations (when overlaps were narrow and competing rules alternated dominance). The fuzzy-tree reduced rules per node but introduced interaction between the two blocks; the wheel-management rules often counteracted the balance rules in mid-recovery, keeping the wheel spinning fast while the angle hovered, or they under-compensated and allowed drift. Noise and bias in $\dot{\theta}$ compounded the difficulty for a type-2 system (see Figure 69) because the footprint had to be wide enough to cope with measurement variability, which further blurred rule firing near the upright region where fine authority is most needed.

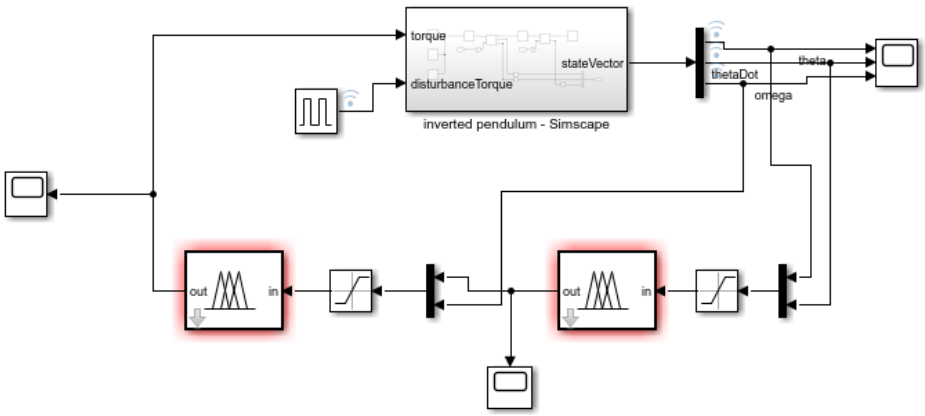


Figure 69: Two-stage Mamdani (type-2) fuzzy-logic "tree" for the inverted-pendulum Simscape model. Separate fuzzy controllers (highlighted) act on the measured lean angle/angle rate and the inertia-wheel speed.

Several practical lessons emerged that can guide a future type-2 FLC effort. It helps to begin with very few linguistic terms—three per input (Negative, Zero, Positive) is a sensible upper bound—and to restrict the rule base to the combinations that physically matter rather than filling the full Cartesian table. Input normalization is essential; scaling θ by π, $\dot{\theta}$ by a representative peak rate, and ω by a safe wheel speed keeps MF footprints commensurate and prevents one signal from dominating inference. It is also important to separate time-scales explicitly: the balance FLC should be fast and high authority, while any wheel-energy FLC should act as a slow bias that only trims steady-state wheel speed once |θ| and $|\dot{\theta}|$ are small, so the two stages do not fight each other. The plant and sensor data gathered from stable PD/NN runs are extremely valuable; clustering those trajectories to place MF centers and widths gives a good first guess for the primary and secondary grades in the type-2 sets. With that initialization in place, parameter tuning should be automated. An optimizer that adjusts MF locations, footprints of uncertainty, and rule weights against a multi-objective cost—penalizing θ^2,$\dot{\theta^2}$, ω^2, actuator saturation, and excessive control variation—converges far more reliably than manual editing of dozens of overlapping rules. Finally, the usual outer protections remain essential: dead-zones, rate limits, and the Stateflow safety guard prevent chatter near the origin and keep the search confined to safe operating regions while the FLC is being refined.

In summary, the type-2 Mamdani approach proved attractive for its ability to represent measurement uncertainty, but the combination of three inputs, nonlinearity, and strict hardware constraints made a hand-crafted rule base unwieldy. A future attempt that starts with a very compact linguistic set, uses data-driven MF placement, enforces bandwidth separation between balance and wheel-energy actions, and tunes footprints and rule weights with an optimizer has a realistic path to success, while retaining the interpretability that motivates a fuzzy design.

5.4 Conclusion

The neural-network controllers were evaluated in two environments: the Simscape inverted-pendulum simulation and the higher-fidelity “physical model” that includes the sensor/actuator interfaces used later on the motorcycle. In simulation, all three feedforward networks stabilized the pendulum and the Mechanics Explorer GUI consistently showed an upright configuration throughout the runs. The feedforwardnet(10) reached the “minimum gradient” stop at epoch 39 with an MSE of 1.29×10^{-9}; it settled the lean angle quickly but drove the wheel to a relatively high steady speed (≈140 rad/s), which is undesirable for hardware. The feedforwardnet(30) met the validation criterion in just nine epochs but left the wheel turning at about 65–70 rad/s and produced a slightly larger residual bias in $\dot{\theta}$ than the other two. The feedforwardnet(20) provided the most balanced behavior: it converged in 15 epochs with an excellent validation MSE ($\approx 1.3 \times 10^{-10}$, brought θ and $\dot{\theta}$ to (near) zero rapidly, and, critically, ramped the wheel speed down toward zero rather than holding a high constant spin. On the basis of these dynamics—not only the training statistics—the 20-neuron feedforward controller was the most suitable choice in the purely simulated setting because it achieved fast regulation with the smallest “cost” in wheel momentum.

When the same idea was transferred to the physical model, a function-fitting network (fitnet) with two hidden layers was preferred so that the capacity could be adjusted without widening a single, very large layer. Results were more nuanced because of measurement noise, sample-time quantization, PWM motor drive and the unmodeled nonlinearities that were intentionally ignored in the Simscape derivation. The fitnet[12 12] solution tended to hit the “maximum μ” stopping condition and produced noisy, high-effort responses; even though it learned from the PD dataset, its closed-loop behavior showed large wheel-speed excursions and intermittent angle drift. Increasing the capacity to fitnet[15 15] delivered the best empirical robustness on the physical model: despite also stopping on μ, the network held the structure upright for extended periods (on the order of tens of seconds) with smaller residual motion, suggesting that the extra units helped the controller interpolate across sensor noise and actuator dead-zones. A further increase to fitnet[20 20] met the validation criterion cleanly and reduced the initial transient, but small oscillations in $\dot{\theta}$ persisted and the solution remained more sensitive to disturbances than the [15 15] case. In short, the physical model favored a moderate-capacity two-layer network trained on diverse PD trajectories and protected by the same safety gating used for hardware.

Taken together, the study indicated that “bigger” is not automatically “better.” In the idealized inverted-pendulum simulation, a single-hidden-layer feedforwardnet with about twenty neurons struck the right balance between accuracy and effort, outperforming both a smaller network that over-spun the wheel and a larger one that did not materially improve the angle regulation. In the physical model, a two-layer fitnet with moderate width ([15 15]) generalized more gracefully than a smaller ([12 12]) or wider ([20 20]) alternative, sustaining balance longer under realistic sensing and actuation. These conclusions were drawn by weighing the training diagnostics against the closed-loop traces: the preferred networks were those that simultaneously minimized lean-angle error, damped angular-rate transients, and avoided unnecessary wheel speed—criteria that matter most once the controller leaves the simulator and meets the hardware.

Chapter 6 – Discussion and Conclusions

This thesis set out to build, model, and control a self-balancing motorcycle kit by treating it as an inverted-pendulum with a reaction (inertia) wheel. The hardware was assembled around the Arduino Nano 33 IoT and Motor Carrier; sensors (BNO055 IMU, encoders, battery monitor) were interfaced in Simulink and validated in External Mode. In parallel, the physics of the system were derived and implemented twice: a state-space/Simulink version and a geometry-accurate Simscape Multibody version. These models provided a safe sandbox for controller design and for generating the signals needed to compare algorithms.

Classical proportional–derivative (PD) control served as the baseline. PD laws were first tuned in simulation to regulate the lean angle and its rate while keeping the inertia-wheel speed bounded; the same structure was then deployed to the physical motorcycle with additional Stateflow safety logic for IMU calibration, battery level, and fall detection. With the PD controller in the loop, both models produced stable, repeatable balancing behavior and the Simscape animation confirmed upright posture under small disturbances.

The core learning contribution replaced that hand-tuned PD block with compact feed-forward neural networks trained on PD-generated trajectories. In the Simscape environment, single-hidden-layer nets with 10–30 neurons (trainlm) mapped the three measured states [θ, $\dot{\theta}$, ω] to the motor torque command and consistently stabilized the pendulum. Training converged in a handful of epochs with low mean-squared error, and the closed-loop responses matched the PD transient while slightly reducing overshoot in some cases. On hardware, the same idea was ported using function-fitting networks (e.g., fitnet [12 12], [15 15], [20 20]) and a data-logging workflow that saved multiple External-Mode runs to build richer datasets. The embedded implementation demonstrated that a Nano-class MCU can host such small networks alongside the I/O stack, with a conventional controller kept available as a compile-time or run-time fallback.

Overall, the study showed that a learning-based controller can reproduce the PD policy learned in simulation and can be brought onto the physical kit with modest networks and careful safety interlocks. The approach is practical for rapid prototyping—PD delivers a reliable teacher policy; neural nets compress that policy into a lightweight function; and Simulink/Simscape provide an efficient path from equations to code. Future refinements would focus on expanding the training corpus with targeted disturbances, exploring quantized int8 deployment for faster inference, and incorporating modest online adaptation to narrow the remaining sim-to-real gap.

Bibliography

Textbooks:

1. Åström, Karl Johan, and Richard M. Murray. *Feedback Systems: An Introduction for Scientists and Engineers*. Rev. ed., Princeton UP, 2020. PDF, cds.caltech.edu.

2. Tedrake, Russ. *Underactuated Robotics: Algorithms for Walking, Running, Swimming, Flying, and Manipulation*. Course notes, 2024, underactuated.csail.mit.edu.
3. Sutton, Richard S., and Andrew G. Barto. *Reinforcement Learning: An Introduction*. 2nd ed., MIT Press, 2018. PDF, incompleteideas.net.
4. Smith, Steven W. *The Scientist and Engineer's Guide to Digital Signal Processing*. California Technical Pub., 1997. dspguide.com.
5. Passino, Kevin M. "The Inverted Pendulum Benchmark." Lecture notes/monograph, The Ohio State University, open PDF

Inverted pendulum & reaction-wheel pendulum:

6. Spong, Mark W. "The Swing Up Control Problem for the Inverted Pendulum." 1994, open copy.

7. Zebenay, M., et al. "Modeling and Control of a Reaction Wheel Pendulum." 2014, *arXiv*, arxiv.org.
8. Uitti, J., et al. "Reaction Wheel Inverted Pendulum." 2021, *arXiv*, arxiv.org.
9. Wu, T., et al. "Nonlinear Control of a Reaction Wheel Pendulum." 2023, *arXiv*, arxiv.org.
10. *Bittar, A., et al. "Robust Control of Reaction-Wheel Pendulum Using Discrete-Time Sliding Modes." 2023, arXiv, arxiv.org.*
11. Zhou, B., and Q. Tang. "Control of a Reaction Wheel Inverted Pendulum." 2022, *arXiv*, arxiv.org.

12. Han, T., et al. "Design and Control of Reaction Wheel Inverted Pendulum." 2020, *arXiv*, arxiv.org.

13. Kloc, M., et al. "The Reaction Wheel Pendulum: A Nonlinear Benchmark." 2017, *arXiv*, arxiv.org.
14. "Control Tutorials for MATLAB and Simulink (CTMS): Inverted Pendulum—Modeling and Controller Design." University of Michigan, ctms.engin.umich.edu.
15. "Reaction Wheel Inverted Pendulum." ORCCA Lab, UBC, orcca.ece.ubc.ca (lab page).
16. Yıldız Technical University. "Modelling and Control of a Reaction Wheel Pendulum Using LQR." 2022, sigma.yildiz.edu.tr.
17. "LEGO Reaction Wheel Inverted Pendulum." *Raspberry Pi Magazine*, 2022, magpi.raspberrypi.com.
18. "Inertia Wheel Inverted Pendulum." nrobot.dev, project report PDF.

Self-balancing robots / bikes (concepts, builds, analysis):

19. “Self-Balancing Bicycle.” CAD/Mechatronics blog, 2011, (archived) arXiv-style preprint.
20. “Arduino Self-Balancing Robot.” Instructables, instructables.com.
21. Linares-Flores, J., et al. “Stabilization and Trajectory Tracking of a Self-Balancing Unicycle.” Open paper.
22. Yao, Guang. “A Self-Balancing Robot with Kalman Filter.” 2019, GitHub repository, github.com.
23. “Self-Balancing Motorcycle (Engineering Kit) – Public Preview.” Arduino Education, edu-content-preview.arduino.cc.

IMU sensing, estimation, and calibration:

24. Bosch Sensortec. *BNO055 Intelligent 9-Axis Absolute Orientation Sensor: Datasheet*. bosch-sensortec.com.
25. “Adafruit BNO055 Absolute Orientation Sensor—Guide.” Adafruit Learn, learn.adafruit.com.
26. Madgwick, Sebastian. “An Efficient Orientation Filter for IMUs.” 2010, open PDF/code.
27. Tron, R., et al. “A Survey on Averaging on SO(3) and Attitude Estimation.” University of Oxford, *arXiv*, arxiv.org.

Neural networks for control:

28. Soria, A., et al. “Neural Network Control of the Inverted Pendulum.” 2020, *arXiv*, arxiv.org.
29. Barto, Andrew G., Richard S. Sutton, and Charles W. Anderson. “Neuronlike Adaptive Elements That Can Solve Difficult Learning Control Problems.” *IEEE Trans. SMC*, 1983, open copy.
30. Lewis, Frank L., et al. “Neural Network Control of Robot Manipulators and Nonlinear Systems.” 2012, open chapter excerpts.
31. Hagan, Martin T., and M. B. Menhaj. “Training Feedforward Networks with the Marquardt Algorithm.” *IEEE Trans. Neural Networks*, 1994, open copy (reprints hosted by universities).
32. Narendra, K. S., and K. Parthasarathy. “Identification and Control Using Artificial Neural Networks.” *IEEE Trans. Neural Networks*, 1990, open preprint.
33. Nguyen, David, and Bernard Widrow. “Improving Learning Speed of Back-Propagation.” 1990, open preprint.
34. Yousefi, S., et al. “Model-Free Control of Inverted Pendulum Using Deep Neural Networks.” 2019, *arXiv*, arxiv.org.

Fuzzy logic & type-2 fuzzy control:

35. Mendel, Jerry M. “Type-2 Fuzzy Sets and Systems: An Overview.” 2007, *arXiv*, arxiv.org.

36. Castillo, Oscar, and Patricia Melin. “Type-2 Fuzzy Logic Controller: A Review.” 2012, open copy.

37. Li, Hongxing, and H. B. Gatland. “Conventional Fuzzy Control and Its Enhancement Using Neural Networks.” 1996, open copy.
38. Sepulveda, R., et al. “Comparative Study of Type-1 vs Type-2 Fuzzy Controllers for Inverted Pendulum.” 2007, open paper.
39. Wang, Li-Xin. “Stable Adaptive Fuzzy Control of Nonlinear Systems.” 1993, open paper.

Modelling, simulation & tools:

40. “Stateflow Onramp.” MathWorks, free short course, mathworks.com.
41. Tedrake, Russ. “Underactuated—Code and Examples.” GitHub repository, github.com/RussTedrake/underactuated.

Extra articles & notes:

42. Kolesnichenko, A., et al. “Swing-Up and Stabilization of Reaction Wheel Pendulum.” 2020, *arXiv*, arxiv.org.
43. Saremi, M., and M. A. Nekoui. “Energy Shaping for Nonlinear Control of Reaction Wheel Pendulum.” 2015, *arXiv*, arxiv.org.
44. Baumann, Michael. “Self-Balancing Robot Using Kalman Filter and PID.” 2019, technical blog/tutorial.
45. Hasan, M.F. and Sobhan, M.A. (2020) Describing Fuzzy Membership Function and Detecting the Outlier by Using Five Number Summary of Data. *American Journal of Computational Mathematics*, **10**, 410-424. doi: 10.4236/ajcm.2020.103022.
46. Witharama, Mudith & Bandara, Dilshan & Azeez, Muhyideen & Bandara, Kasun & Velmanickam, Logeeshan & Wanigasekara, Chathura. (2024). Advanced Genetic Algorithm for Optimal Microgrid Scheduling Considering Solar and Load Forecasting, Battery Degradation, and Demand Response Dynamics. IEEE Access. 10.1109/ACCESS.2024.3412914.
47. Ma, Xun & Spinner, Sogee & Venditti, Alex & Li, Zhao & Tang, Strong. (2019). Initial Margin Simulation with Deep Learning. SSRN Electronic Journal. 10.2139/ssrn.3357626.
48. Robert Keim, November 2024, How to train a basic perceptron neural network
49. Multi-Layer Perceptron Learning in Tensorflow.
50. Hamdan, Muhammad. (2018). VHDL auto-generation tool for optimized hardware acceleration of convolutional neural networks on FPGA (VGT).
51. Backpropagation Explained Visually—Step-by-Step Learning
52. Bido. *Validation Error less than training error?* Cross Validated, 2017.
53. Odi, Uchenna & Nguyen, Thomas. (2018). Geological Facies Prediction Using Computed Tomography in a Machine Learning and Deep Learning Environment. 10.15530/urtec-2018-2901881.
54. Ilten, Erdem. (2025). DC Motor Fractional Order PID Position Control with Arduino Nano 33 IoT and Simulink External Mode.
55. Hedjar, Ramdane. (2007). Online Adaptive Control of Non-linear Plants Using Neural Networks with Application to Temperature Control System. Journal of King Saud University - Computer and Information Sciences. 19. 10.1016/S1319-1578(07)80005-X.

56. Müller, Hausi & Villegas, Norha. (2014). Runtime Evolution of Highly Dynamic Software. 229-264. 10.1007/978-3-642-45398-4_8.