Jiayi Li

Real-time Controller and Interface for a PID Control Demonstrator

Faculty of Engineering and Applied Science

Aerospace Vehicle Design

MSc

Academic Year:2024-2025

Supervisor: Dr. James Whidborne

August 2025

Faculty of Engineering and Applied Science
Aerospace Vehicle Design

MSc

Academic Year: 2024-2025

Jiayi Li

Real-time Controller and Interface for a PID Control Demonstrator

Supervisor: Dr. James Whidborne
August 2025

This thesis is submitted in partial fulfilment of the requirements for the degree of MSc.
© Cranfield University 2025. All rights reserved. No part of this publication may be reproduced without permission.

Academic Integrity Declaration

I declare that:

- The thesis submitted has been written by me alone.
- The thesis submitted has not been previously submitted to this university or any other.
- All content, including primary and/or secondary data, is true to the best of my knowledge.
- All quotations and references have been duly acknowledged according to the requirements of academic research.

I understand that to knowingly submit work in violation of the above statement will be considered by examiners as academic misconduct.

Real-time Controller and Interface for a PID Control Demonstrator

Jiayi Li

Abstract

This paper reports on the redesign of a flight desk control demonstrator for teaching feedback control. The original system, limited by a basic Java interface and minimal controller functionality, was upgraded to improve usability and pedagogical value. The backend was re-implemented as modular firmware on an Arduino Nano, featuring a deterministic 2 ms sampling loop and support for both PID and PIV controllers. Robust host–device communication was achieved using a framed UART protocol. A new MATLAB App Designer GUI was developed with modular panels for controller tuning, real-time step response visualisation, tutorials, and data export. Results show that the system met technical and educational requirements, providing an interactive, reproducible, and extensible platform for control systems education.

NOMENCLATURE

PID	Proportional–Integral–Derivative controller.
PIV	Proportional–Integral with velocity feedback term $K_v\dot{y}$.
GUI	Graphical User Interface (MATLAB App Designer).
UART	Framed serial link between host and firmware.
PWM	Pulse-Width Modulation command to the servo (µs).
r	Reference / setpoint.
y	Measured output (flap angle).
$e = r - y$	Control error.
u	Control effort (pre-clamp actuator command).
$v = \dot{y}$	Output velocity used in PIV feedback.
K_p, K_i, K_d	PID gains.
K_v	Velocity feedback gain (PIV).
T_s	Sampling period of the control loop (here 2 ms).
PM	Phase margin at crossover.
GM	Gain margin.
t_s	Settling time (to a specified band).
OS	Percentage overshoot.

Greek Symbol

α	Angle of attack (AoA).
ω_c	Crossover frequency where $\lvert L(j\omega_c)\rvert = 1$ (0 dB).

1.0 Introduction

Motivation

Control theory forms a cornerstone of modern engineering education, yet its abstract mathematical concepts can be challenging for students to connect with practical, real-world behaviour. Laboratory demonstrators provide a tangible means to bridge this gap, allowing learners to visualise system dynamics and relate theoretical analysis to observable responses. An earlier Cranfield flight desk demonstrator was developed for this purpose, but limitations in its usability, robustness, and pedagogical value reduced its effectiveness. In particular, the reliance on a basic Java-based GUI, limited controller functionality, and fragile hardware assembly constrained its broader adoption. A modernised platform was therefore required, one capable of supporting interactive learning with richer functionality, clearer usability, and greater extensibility.

Objectives

The overarching objective of this project was to redesign the demonstrator as a more effective teaching tool for control systems education. This required achieving the following specific goals:

- Implement modular embedded firmware on the Arduino Nano, supporting both classical PID and PIV controllers.
- Develop a new MATLAB App Designer GUI to provide real-time visualisation, interactive controller tuning, and embedded tutorials.
- Establish a robust communication protocol between host and hardware, ensuring deterministic control at fixed sampling rates.
- Validate that the system meets both technical requirements (robust control, reliable communication, modular design) and educational requirements (clarity, usability, reproducibility).

Contributions

The contributions of this work are threefold. First, a fully modular backend firmware was implemented, incorporating a deterministic sampling loop, reusable controller modules, and a communication façade for reliable host interaction. Second, a modern, event-driven GUI was created in MATLAB, with modular panels for controller tuning, step response plotting, tutorials, and data export. Third, the system was validated as an integrated demonstrator, showing that it supports real-time controller tuning, provides

clear visual feedback, and facilitates reproducible experiments. In addition, the project produced a template firmware “library” to support future integration of alternative control algorithms, laying a foundation for continued development.

Structure of Paper

The remainder of this paper is organised as follows. Chapter 2 reviews the background and related work on educational control demonstrators. Chapter 4 presents the methodology, including requirements capture, the system design approach, and the layered software architecture. Chapter 5 describes the backend firmware implementation, while Chapter 6 details the frontend GUI development. Chapter 7 presents the implementation results, validation, and critical assessment. Finally, Chapter 8 concludes with a summary of findings and outlines directions for future work.

2.0 Literature review

2.1 User Interface Design for Learning Efficiency

Human–Computer Interaction (HCI) research shows that an intuitive, well-designed interface can significantly enhance learning by reducing unnecessary mental effort. In educational software and intelligent tutoring systems (ITS), the GUI acts as the communication medium between learners and the content, so its design directly influences engagement and comprehension[5]. A clear, easy-to-use interface motivates students to continue using the tool, whereas a poorly designed interface can frustrate and demotivate them, reducing the likelihood that they will learn effectively[5]. Modern HCI studies emphasize minimizing extraneous cognitive load – the mental effort imposed by the interface itself – so that learners can focus on the intrinsic task of understanding the material[4]. According to cognitive load theory, while intrinsic load is tied to the complexity of the subject matter, extraneous load stems from how information is presented[4].
By streamlining navigation, using familiar visual cues, and providing timely feedback, interface designers can reduce extraneous load and thereby improve overall learning outcomes[1,2,3]. For example, an ITS for logic tutoring was recently redesigned to replace free-response input with a multiple-choice interface. This change cut students' problem-solving time roughly in half without harming learning gains[6]. Students could select auto-generated answers instead of typing everything, which saved time and cognitive effort while achieving the same results – effectively increasing learning efficiency. This illustrates a broader trend in modern HCI: interfaces that intelligently adapt to learners can maintain or even improve learning performance while reducing the time and effort required.

2.2 Physical Control System Demonstrators for Teaching

In engineering education – especially in control systems courses – hands-on demonstrator platforms are widely used to reinforce theoretical concepts with real-world experience. These range from classical laboratory experiments to modern, open-source setups and remote laboratories.

2.2.1 Classic Unstable System Experiments

One of the most well-known is the **inverted pendulum** on a cart. This inherently unstable system is frequently used to teach students how a Proportional–Integral–Derivative (PID) controller can stabilize an otherwise unstable process[7]. Because the inverted pendulum has dynamics analogous to real-world problems (such as rocket stabilization), it provides an engaging

context for learning control theory. Recent efforts have produced low-cost, open-source versions using 3D-printed parts and Arduinos, with costs under $100[8]. These designs allow wide adoption in classrooms. Student feedback confirms that hands-on work with the pendulum improved understanding of PID control and increased confidence[8].
Another widely adopted experiment is the **ball-and-beam** system, another open-loop unstable plant. In this setup, a ball rolls on a beam, and the controller must tilt the beam to balance the ball at a setpoint. This illustrates setpoint regulation, steady-state error, and damping[9]. Recent implementations using Arduino microcontrollers and IR sensors demonstrate that low-cost setups can replicate expensive commercial rigs and give students valuable PID tuning practice[9].

2.2.2 Magnetic Levitation and Advanced Projects

More advanced demonstrators such as **magnetic levitation** systems suspend a steel ball using an electromagnet with feedback control. These systems are nonlinear and unstable, offering rich learning opportunities[10]. MIT introduced low-cost maglev project kits for undergraduate courses, enabling each student to build and test their own desktop levitator[11]. Students could implement proportional control for marginal stability, then design and test improvements such as derivative compensation to achieve smooth levitation.

2.2.3 Process Control and Industrial-Style Demonstrators

Beyond mechanical rigs, **coupled water tanks** are widely used in process control teaching[12]. Students regulate the liquid level by controlling pump flow, mirroring industrial chemical or thermal plants. Labs built with LabVIEW or MATLAB/Simulink provide practice with system identification, sensor calibration, and PID tuning[12]. Such projects expose students to industrial practices like dealing with noise and actuator nonlinearities.
Similarly, programmable logic controller (PLC) platforms are essential in automation training. Traditionally costly, they are now supported by open-source PLC frameworks such as **OpenPLC**, which can run on Arduino or Raspberry Pi hardware[13]. This democratizes access, allowing students to practice ladder logic and automation sequencing without expensive vendor-specific kits.

2.2.4 Remote and Open-Source Laboratories

Recent trends include open-source **remote labs** that allow students to access experiments via the internet. For example, RaspyControl Lab offers a

low-cost Raspberry Pi–based remote control laboratory with web access, enabling students to run tank control experiments from anywhere[14]. Its open-source nature makes it adaptable to different plants (thermal chambers, inverted pendulums, etc.), supporting scalable and flexible learning.

3.0 Summary

The literature highlights a rich ecosystem of control demonstrators, from time-honored pendulum and ball-and-beam systems to modern open-source kits and scalable remote labs. All share the goal of making control theory tangible and interactive. The trend towards openness and embedding HCI principles (simplicity, feedback, cognitive load management) strengthens their educational value. Well-designed demonstrators, combined with intuitive GUIs, provide powerful experiential learning opportunities that bridge theory and practice.

4.0 System Design & Architecture

4.1 Overview of Demonstrator

The present project builds upon an existing flight desk control demonstrator originally developed by Duran[15]. The demonstrator consists of a small-scale wind tunnel housing a NACA 0012 aerofoil with a trailing-edge flap actuated by a servomotor. Sensor feedback is provided by a potentiometer, which measures flap deflection, and a Pitot-static system, which measures air velocity within the tunnel. Data acquisition and actuation are handled by an Arduino Nano, enabling the platform to function as a closed-loop control system for demonstration purposes. The overall hardware arrangement is illustrated in Figure 1, while a more detailed description of the hardware assembly is provided in Appendix I.

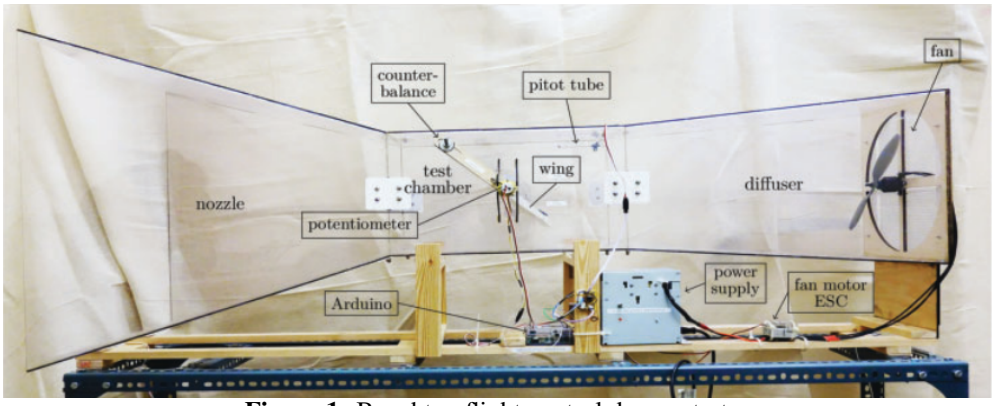

Figure 1: Benchtop flight control demonstrator

The purpose of the demonstrator is to provide a physical environment in which students can investigate fundamental concepts of feedback control. By observing the dynamic response of the aerofoil system to step inputs or disturbances, learners are able to relate theoretical control-system behaviour to a tangible physical process. In its original implementation, the system included a basic embedded controller and a Java-based graphical user interface for PID tuning and data visualisation. While effective as a proof of concept, limitations in usability, extensibility, and analysis capability restricted its broader pedagogical value.

In the present project, the hardware platform itself was retained without modification. However, substantial enhancements were made at the software level. Specifically, the embedded C firmware running on the Arduino was redesigned to incorporate both PID and PIV (Proportional–Integral–Velocity)

control algorithms, providing a richer set of comparative control experiments. In addition, a new MATLAB-based graphical user interface was developed to replace the legacy Java implementation. This GUI introduces modular panels for controller tuning, real-time response plotting, automated extraction of control performance metrics (rise time, overshoot, settling time, steady-state error), and tutorial elements designed to support classroom use.

These developments significantly expand the functionality of the demonstrator while maintaining compatibility with the existing hardware, positioning the platform as a more diversified and effective teaching tool.

Although the demonstrator hardware remained unchanged, the redesign required a systematic definition of objectives to ensure that the new software components addressed both the pedagogical and technical shortcomings identified in the Introduction. Accordingly, the next step in the methodology was to formalise these requirements and structure them within an engineering design framework, as outlined in Section **??**.

4.2 Requirements Capture and Design Approach

Building on the overview of the demonstrator and the enhancements planned, the next stage of the methodology was to translate the identified shortcomings into a structured set of requirements. This process ensured that the redesign was not only technically functional but also aligned with the intended pedagogical outcomes.

The project requirements were captured through three inputs:

(i) A review of the prior Cranfield demonstrator[(?)] to identify limitations affecting usability and pedagogy;
(ii) Discussions with supervisors to align the work with intended learning outcomes for control teaching; and
(iii) Analysis of technical constraints imposed by the inherited hardware (Arduino timing, sensor fidelity, serial bandwidth).

From this process, the requirements were structured into two sets:

Educational requirements

E1. Enable interactive tuning of classical controllers (PID and PIV) on a real plant.

E2. Provide real-time visualisation of responses with automatically computed metrics (rise time, % overshoot, settling time, steady-state error).

E3. Improve clarity and usability for learners (modular GUI, clear plots, tutorial/scaffolded explanations).

E4. Support reproducible classroom/lab activities (consistent procedures, logging, screenshots/exports).

Technical requirements

T1. Deterministic embedded control loop on Arduino with configurable sampling.

T2. Robust host–device communication (framed serial protocol; parameter updates; streamed telemetry).

T3. Modular MATLAB GUI (connection "master" panel; PID/PIV panels; step-response plotter; tutorial panel).

T4. Data handling for analysis and export (buffers, logs, error handling).

To ensure traceability from requirements to verification, a V-model was adopted: the left branch decomposes needs into architecture and modules (frontend GUI, communication layer, backend firmware); the right branch mirrors these with unit, integration, and system testing, plus an educational validation against E1–E4. The model used here is shown in Figure 2 and underpins the structure of Sections 3–5: design (left), implementation, then integration/validation (right).

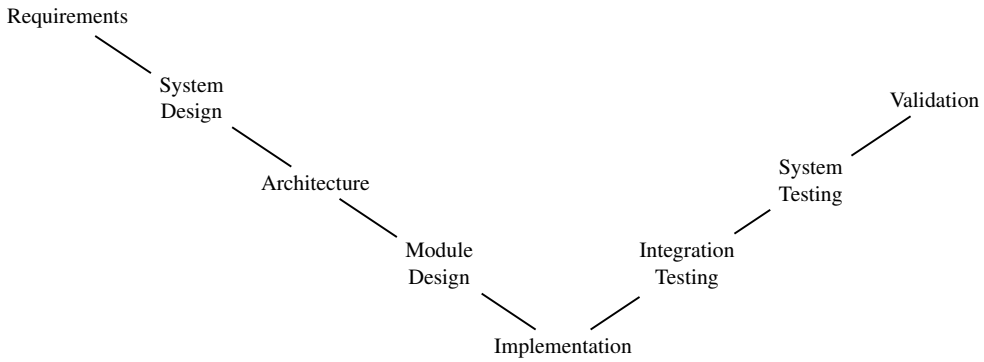

Figure 2: V-model applied to the demonstrator redesign

4.3 System Design

The redesigned demonstrator follows a layered architecture[21,23], consisting of three tiers: the *Frontend (GUI)*, the *Communication Layer*, and the *Backend (Embedded Controller)*. The rationale for this structure is to achieve separation of concerns, improve maintainability, and support incremental extension of functionality. Each layer exposes a narrow, stable interface to adjacent layers, which reduces coupling and simplifies debugging. The high-level structure is shown in Figure 3.

- **Frontend (GUI):** MATLAB App Designer application composed of custom, reusable components (*PID Tuner*, *PIV Tuner*, *Step Response Plotter*, *Tutorial Panel*, *Export to CSV*). One component, the *Arduino Connection* panel, acts as a master façade for connection lifecycle and messaging.
- **Communication Layer:** Framed serial protocol (UART) responsible for encoding/decoding command and telemetry frames, versioning, and basic error handling. The GUI subscribes to new-frame events rather than polling.
- **Backend (Embedded Controller):** Modular C firmware implementing a deterministic sampling loop, PID and PIV controllers, sensor I/O (potentiometer, Pitot), actuator drivers (servo), and utility functions (e.g., `buzzerTone()` for user feedback).

Figure 3 shows the overall layering. The layered model reduces coupling across tiers, easing future redesigns (e.g., alternative controllers, different plants) while preserving the external interfaces[21,22].

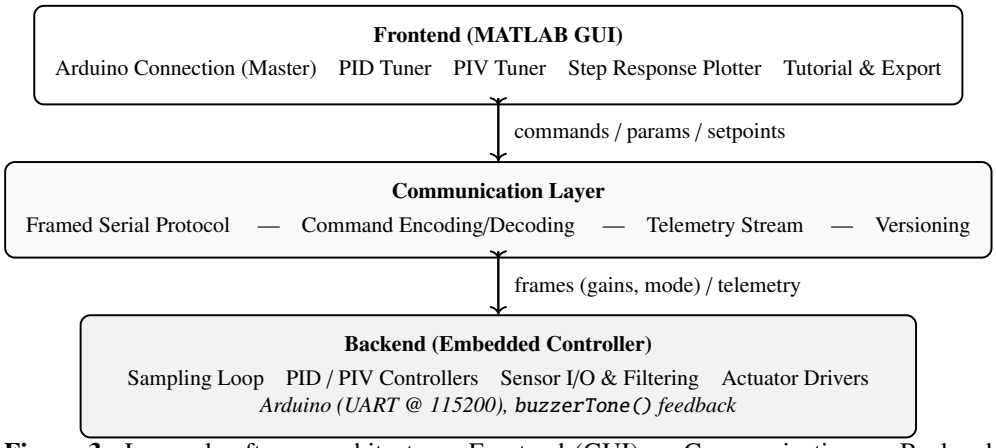

Figure 3: Layered software architecture: Frontend (GUI) → Communication → Backend (Arduino). Each layer exposes a narrow interface to reduce coupling and simplify evolution[21,23].

4.4 Frontend: Modular, Event-Driven GUI

The frontend was implemented in MATLAB App Designer following a modular, event-driven architecture. The application is organised into a set of custom `ComponentContainer` classes, each responsible for a single functional concern. This modularisation ensures that the overall GUI can be assembled as a composition of independent panels, each with a well-defined API. The principal panels include the *Arduino Connection* panel, the *PID/PIV Tuner*, the *Step Response Plotter*, the *Tutorial Panel*, and an *Export Utility*. By encapsulating functionality in discrete components, the architecture supports reusability, independent testing, and straightforward replacement or extension of individual modules.

At the core of the design is the *Arduino Connection* panel, which acts as a façade to the communication layer. It establishes and maintains the serial session, encodes outgoing command frames, and decodes incoming telemetry. Importantly, the connection panel is also responsible for raising *events* whenever new telemetry is available. This makes it the central point of integration: all other components interact with hardware data indirectly via the connection panel rather than accessing the serial interface directly.

Inter-component communication is handled using MATLAB's event/listener mechanism. Events are declared in the connection panel and raised when a new frame is processed. Other panels, such as the *Step Response Plotter* or the *Telemetry Logger*, attach listeners to these events. When triggered, the

associated callback methods execute within the subscribing panel, updating state or rendering visualisations. This event-driven model ensures *loose coupling*: panels do not need to know about each other's internal implementation, only about the existence of published events. Figure 4 illustrates this pattern.

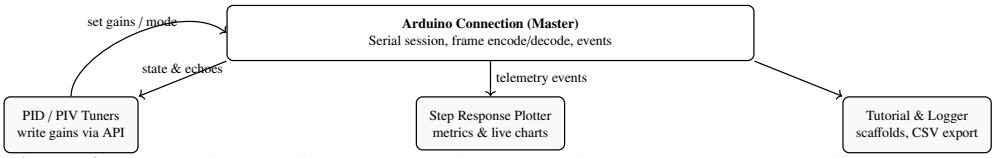

Figure 4: Frontend event-driven pattern: the Connection component emits telemetry events; downstream panels subscribe and act, remaining decoupled from serial details[22].

The PID/PIV tuner panel communicates with the connection panel via a well-defined public interface. When the user adjusts a gain value, the tuner invokes a setter method on the connection panel, which constructs the appropriate serial frame and transmits it to the backend. The step response plotter, by contrast, operates passively: it listens to telemetry events and refreshes its plots whenever new data are available. The tutorial panel and export utility follow the same subscription pattern, reacting to event notifications rather than polling for data.

This architecture provides three clear advantages. First, it enforces a single point of control for hardware communication, reducing complexity and potential race conditions. Second, it allows new components to be added incrementally: any new panel need only subscribe to existing events or expose new listeners of its own. Third, it supports independent testing: individual components can be validated by simulating events, without requiring the full system or physical hardware. Collectively, these properties align with best practice in layered and modular software architecture[21,23,22] and provide a scalable foundation for future extensions of the frontend.

4.5 Communication Layer: Framed Serial Protocol

The communication layer bridges the frontend and backend using a framed UART protocol at 115200 baud. Each frame encodes controller parameters (gains, setpoints, mode) or telemetry values (sensor readings, actuator commands, state echoes). Framing and checksums were included to improve robustness against transmission errors. This design choice ensures a *clear separation of concerns*: GUI components interact only with high-level events, while the connection panel handles all protocol-level details.

Centralising communication avoids race conditions and simplifies debugging, since only one component is responsible for maintaining connection state. It also aligns with architectural best practices that recommend minimising interdependencies between modules[21,23].

4.5.1 Backend Architecture

The backend consists of modular embedded C++ firmware running on an Arduino Nano. The design follows a layered architecture in which each module is responsible for a single concern and interacts with others through narrow, stable interfaces. This prevents cross-coupling between unrelated functions, simplifies debugging, and supports future extensions of the control system.

Module Structure.

The main modules of the backend are:

- **Communicator** (`Communicator.h/.cpp`): manages all serial communication with the host GUI. It decodes framed UART packets (12 bytes: proportional gain, integral gain, derivative gain or K_v, motor speed, setpoint, and control mode) and dispatches them to the runtime. It also encodes telemetry packets (10 bytes) including measured input, output, air velocity, setpoint echo, and loop period, transmitting every 25 ms.
- **Simulation / Runtime** (`Simulation.h/.cpp`): provides the lifecycle and scheduling of the real-time control loop. It initialises hardware, enforces sampling, calls the active controller, and schedules periodic housekeeping tasks such as telemetry, fan motor updates, zero-angle recalibration, and buzzer signalling.
- **Controllers**:
  - **PID_v3** (`PID_v3.*`): a full PID controller supporting proportional-on-error/measurement, a filtered derivative term, setpoint and input scaling, and optional back-calculation anti-windup.
  - **PIVController** (`PIVController.*`): a proportional–integral controller with velocity feedback, where the derivative channel of the serial packet is re-purposed to carry K_v. This allows comparisons between classical PID and PIV strategies.
- **Sensor I/O**: reads raw values from the potentiometer (flap angle) and

Pitot tube (air velocity). Simple exponential smoothing filters reduce noise. Inputs are mapped into the servo microsecond domain so that the controller, setpoint, and actuator share a common unit space.

- **Actuator Driver**: generates PWM signals to command the flap servo (in microseconds). The motor fan speed is updated gradually to avoid step transients.
- **Utilities**: auxiliary functions such as `buzzerTone()` (auditory and LED feedback) and `resetZeroAngle()` (periodic recalibration of the neutral angle).

Firmware Lifecycle.

The runtime follows the standard Arduino pattern of `setup()` followed by repeated calls to `loop()`:

- **`setup()`**: initialises hardware, attaches actuators, configures controller limits, establishes common domains for setpoint, input, and output, and sets controller parameters (sample time, derivative filter, anti-windup). Depending on the selected mode, either PID or PIV initialisation is performed.
- **`serialEvent()`**: invoked whenever 12 bytes are available on the serial interface. This function reconstructs the frame and updates the controller tunings, setpoint, motor command, and mode by calling either `updateSimulationPID()` or `updateSimulationPIV()`.
- **`loop()`**: executes the deterministic control loop at a fixed sampling time of 2 ms. Within each cycle, the firmware:
  1. Acquires and filters sensor inputs.
  2. Computes the control law:
     - In PID mode: calls `PID_v3.Compute()` and writes the servo output directly.
     - In PIV mode: executes the PI core, then subtracts a velocity feedback term $K_v\dot{y}$ computed from the filtered change in measured angle.
  3. Clamps the output to physical limits and writes the actuator command.

Additionally, periodic housekeeping tasks are executed on slower schedules: telemetry transmission every 25 ms, fan motor speed adjustment every 100 ms, buzzer toggling, and zero-angle recalibration every 2 s. Figure 5

summarises the data and control flow between sensors, conditioning, the controller (PID/PIV), actuators, and the serial command path.

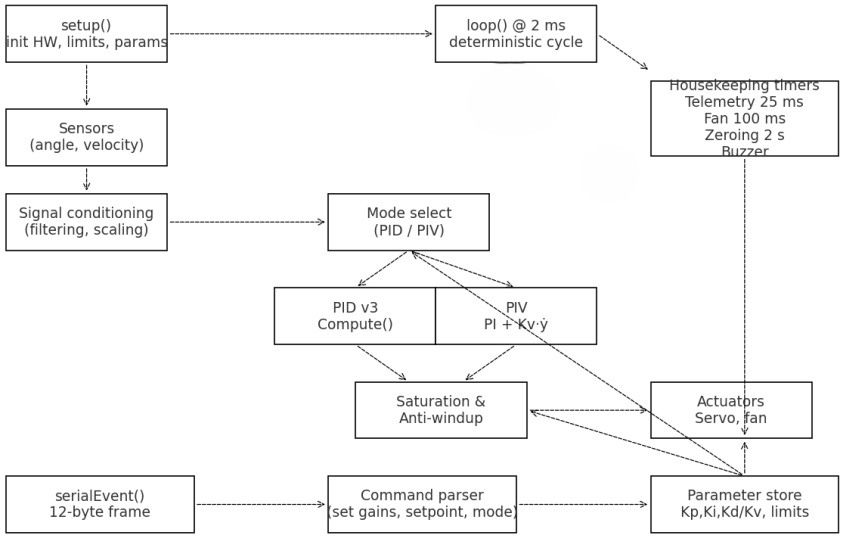

Figure 5: Lifecycle and data/control flow

To make the runtime behaviour explicit, the following pseudocode summarizes the firmware architecture at a level suitable for readers who do not have the source open. It shows (i) one-time hardware and controller initialisation in setup(), (ii) a hard real-time loop in loop() scheduled at 2 ms, and (iii) an asynchronous serial pathway that updates controller parameters via compact 12-byte frames. The loop itself is partitioned into acquisition/conditioning, control-law evaluation (PID or PIV), actuation with saturation and anti-windup, and non-blocking housekeeping tasks. This listing is intended as an explanatory map of the execution order, data dependencies, and timing guarantees; the complete firmware is provided in Appendix **??**.

```
1 /* ============================
2    One-time initialisation
3    ============================ */
4 void setup() {
5   initHardwarePinsAndADC();              // Configure GPIO, ...
       ADC, timers/clock.
6   attachActuators();                     // Bind servo PWM, ...
```

```
      fan driver, buzzer.
  setDomainsAndLimits();                  // Define ...
      units/ranges, output clamps.
  configureControllerParams();            // Set Ts=2 ms, ...
      D-filter, AWU hooks.
  if (mode == PID) initPID(); else initPIV(); // Preload ...
      gains, internal state.
  initSerial(115200);                     // Start 12-byte ...
      frame protocol (UART).
  startHousekeepingTimers();              // Arm 25 ms, 100 ms, ...
      2 s software timers.
}

/* ========================================
   Deterministic control loop @ 2 ms period
   ======================================== */
void loop() {
  const uint32_t t0 = micros();           // Timestamp loop ...
      start for deadline.

  // (1) Acquire & condition signals
  Raw   r = readSensors();                // Read angle (and ...
      velocity if available).
  State x = condition(r);                 // Apply filtering, ...
      scaling, unit mapping.
                                          // Also compute error ...
                                              = r_set - y_meas.

  // (2) Compute control law (selectable mode)
  float u;                                // Actuator command ...
      before clamping.
  if (mode == PID) {
    u = PID_Compute(x.error);             // Standard PID on ...
        error (with D filter).
  } else {                                // PIV = PI on error ...
      + velocity feedback
    float v   = estimateVelocity(x.angle); // Differentiate or ...
        use velocity sensor.
    float uPI = PI_Compute(x.error);      // Integral removes ...
        steady-state error.
    u = uPI - Kv * v;                     // Velocity feedback ...
        adds damping.
  }

  // (3) Enforce limits and actuate
  u = clamp(u, u_min, u_max);             // Physical/driver ...
      safety limits.
  antiWindupUpdate(u);                    // Feed back clamp to ...
      limit integrator wind-up.
  writeActuators(u);                      // Update servo PWM ...
      and fan setpoint.
```

```
39
40    // (4) Non-blocking housekeeping (coarse schedulers)
41    if (timerElapsed(Telemetry25ms)) sendTelemetry();  // Stream ...
          selected signals.
42    if (timerElapsed(Fan100ms))      updateFan();      // ...
          Thermal/airflow management.
43    if (timerElapsed(Zero2s))        maybeRezero();    // ...
          Periodic sensor bias check.
44    if (timerElapsed(Buzzer))        toggleBuzzer();   // ...
          UI/alert feedback.
45
46    // (5) Deadline hold (if not ISR-driven)
47    busyWaitUntil(t0 + 2000);          // Guarantee 2 ms period ...
          in microseconds.
48  }
49
50  /* ========================================
51     Asynchronous serial control (parameter path)
52     ======================================== */
53  void serialEvent() {
54    // Frames are fixed-length (12 B) to keep parsing ...
          deterministic.
55    while (Serial.available() ≥ 12) {
56      Frame f = readFrame12B();            // Read and unpack ...
            command frame.
57      if (!valid(f)) continue;             // CRC / bounds check.
58      applyCommand(f);                     // Update gains, ...
            setpoint, mode atomically.
59      // Keep internal simulation/preview in sync with the ...
            active mode:
60      if (f.mode == PID) updateSimulationPID(f);
61      else               updateSimulationPIV(f);
62    }
63  }
```

Interfaces and Data Flow.

The communication interface ensures consistency across layers. On the receive side, PID mode interprets the third field of the packet as the derivative gain K_d, whereas PIV mode interprets it as the velocity gain K_v. This design choice maintains backward compatibility with the 12 byte packet format. On the transmit side, telemetry values are mapped back into degrees or physical units so that the GUI can plot results without further conversion. Internally, all control computations are performed in the servo microsecond domain, simplifying saturation and anti-windup logic.

System Architecture Diagram.

The overall backend software architecture is illustrated in Figure 6, showing the major modules and their communication.

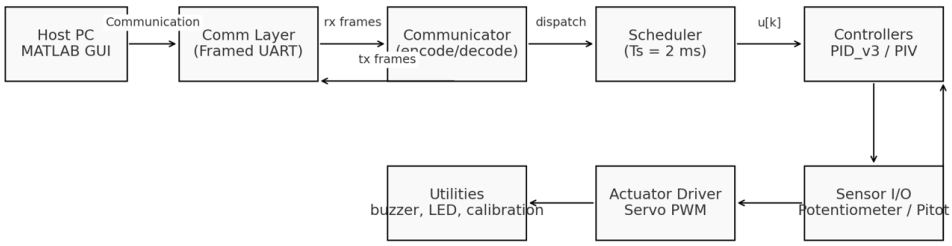

Figure 6: Backend software architecture: high-level modules and communication between host, communication layer, and embedded firmware.

Runtime Flow.

The runtime execution flow is illustrated in Figure 7. At startup, the `setup()` function configures the hardware and loads the correct controller parameters before handing control over to the main `loop()`. This loop runs every 2 ms and is responsible for reading sensors, applying either a PID or PIV control law, and sending commands to the actuators. In parallel, slower background tasks handle telemetry, fan management, and user feedback such as the buzzer or calibration routines. The third pathway, `serialEvent()`, reacts to incoming serial data, allowing parameters such as controller gains or the setpoint to be adjusted on the fly. In this way, the system combines a fast, deterministic control loop with flexible communication and background services, providing both reliability and adaptability.

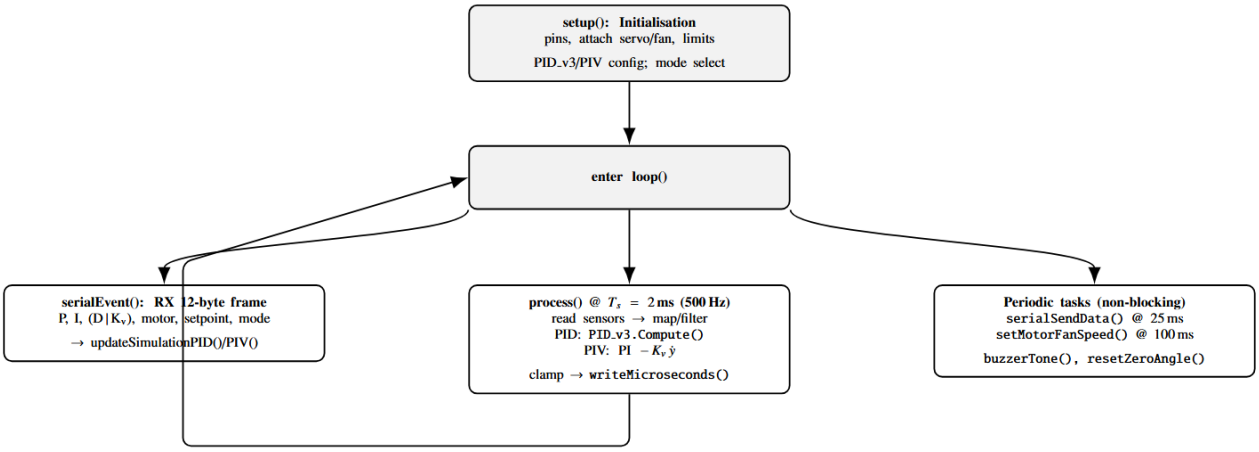

Figure 7: Firmware runtime flow: initialisation in setup(), deterministic control in loop() via process() at fixed Ts, with asynchronous updates through serialEvent() and non- blocking periodic tasks.

Summary.

By isolating the communication façade, enforcing deterministic sampling, and encapsulating control laws in swappable modules, the backend architecture achieves three properties: (i) *debuggability*, as faults can be traced to a specific module; (ii) *upgradeability*, since new controllers can be integrated through the same update hooks and packet format; and (iii) *determinism*, with a guaranteed 2 ms control period. This structure aligns with software engineering best practices [21,23] and provides a scalable and reliable foundation for control education.

5.0 Embedded Control Software Redesign (Backend)

5.1 C Code Rewrite

The backend firmware was completely refactored from a monolithic sketch into a set of modular C++ components. Core functionality was separated into dedicated modules, including the `Communicator` for framed serial I/O, the `Simulation` module for plant interfacing, and distinct controller classes for PID and PIV. This modularisation promotes clarity, simplifies debugging, and supports incremental evolution of the code base. Each module exposes a narrow, well-defined API, consistent with principles of information hiding and software reusability[21].

In order to facilitate long-term maintainability, a lightweight *library structure* was created to consolidate reusable functionality. This library encapsulates controller code, communication utilities, and simulation routines, enabling future students to embed additional algorithms with minimal effort. A template interface was defined so that any new control algorithm can be integrated by adhering to the same update and initialisation patterns used by the existing PID and PIV controllers. This design ensures forward compatibility and reduces the barrier for future extensions.

5.2 Implementation of Controllers

Two control strategies were implemented: a classical PID controller and a Proportional–Integral–Velocity (PIV) controller. The PID implementation is based on the widely adopted `PID_v2` library, extended to incorporate features such as input, output, and setpoint limits, as well as a derivative weight filter. The PIV controller was implemented as a custom C++ class (`PIVController`), structurally mirroring the PID class but introducing a velocity feedforward term, $Kv \cdot \frac{dr(t)}{dt}$, to improve tracking of setpoint changes. Both controllers support configuration of gains, limits, and sample times through the GUI, and can be switched at runtime via the control mode field in the communication frame.

The modularity of the implementation allows each controller to be developed and tested in isolation. The PID path follows the classical $u(t) = K_p e(t) + K_i \int e(t)\,dt + K_d \dot{e}(t)$ formulation, whereas the PIV path augments the PI action with a velocity-dependent term to provide faster response to setpoint transitions. The results of both approaches are presented in Section 7.5.

5.3 Design Decisions

Key design choices were made to ensure deterministic and reproducible behaviour. The sampling period was fixed at $T_s = 2$ ms, corresponding to a 500 Hz control loop, balancing real-time responsiveness with the computational constraints of the Arduino Nano. All timing was implemented using `millis()` counters to avoid blocking delays, and the process loop was structured to guarantee that sensor acquisition, controller computation, and actuator updates complete within each sample interval.
Telemetry was framed into 12-byte packets transmitted at 115200 baud, providing a compromise between communication throughput and robustness. The decision to reuse the "D" field in the packet for K_v when operating in PIV mode enabled backward compatibility with the PID packet structure, simplifying host integration. A buzzer/LED utility (`buzzerTone()`) was integrated to provide auditory and visual confirmation of key events, consistent with human-computer interaction principles of immediate and multimodal feedback[(2)].
Collectively, these design choices yielded a backend that is modular, extensible, and pedagogically effective. The control loop structure is shown in Figure 8.

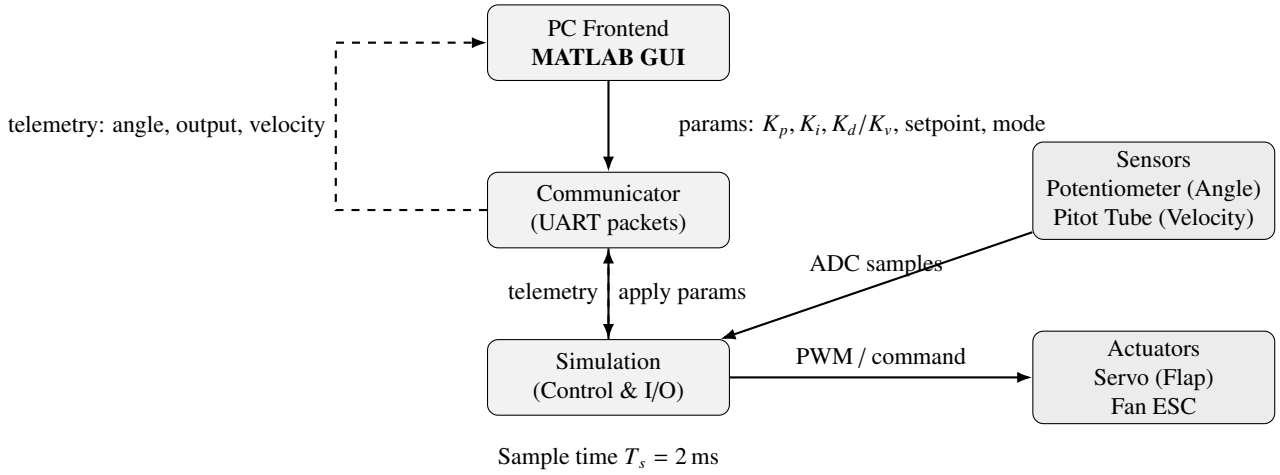

Figure 8: High-level backend architecture: clearer layout with separate telemetry and parameter flows.

6.0 Frontend GUI Design

The front-end graphical user interface (GUI) for the control system has been developed in MATLAB App Designer as a human-centred teaching tool. Its design is driven by pedagogical goals of minimizing cognitive load on learners, providing immediate visual feedback, and reducing user frustration during experimentation. To achieve these goals, the interface leverages established human-computer interaction (HCI) principles throughout.
In particular, the GUI aligns with Nielsen's usability heuristics[1]—for example, emphasizing "recognition rather than recall" to minimize the user's memory burden, and incorporating clear status indicators to prevent errors. It also follows Norman's design philosophies on feedback and visibility[2], ensuring that key system states and the results of user actions are always apparent. For instance, whenever a parameter is adjusted or a command is executed, the interface provides instant feedback (through updated plots, status messages, or color cues) to keep the student informed of the system's response. Additionally, the development adhered to the ISO 9241-210 human-centred design framework[3], involving iterative prototyping and user feedback, so that the final GUI meets the needs and expectations of students.

6.1 Layout and Organization

The overall GUI layout is illustrated in Figure 9. The interface is organized into two main columns: a left-side column for connection and instructional panels, and a right-side column for the visualization of system responses and controller tuning controls. This division reflects the typical workflow a student follows: first establishing a connection and understanding the task (left side), then conducting experiments and adjusting parameters (right side). The most critical output—the real-time step response plot—is placed prominently at the top-right to capture attention, while supportive controls and information are arranged around it in a logical manner.

Crucially, related interface elements are grouped to reduce eye travel distance and streamline interactions. For example, the controller sliders are positioned directly below the step response graph they influence, so that students can manipulate a parameter and immediately observe the effect above with minimal visual scanning. Such spatial coupling of controls and results creates a natural mapping between action and outcome, in line with Norman's principles of mapping and feedback[2]. The layout also exploits a "Z-pattern" viewing flow common in user interfaces: the user's gaze initially goes to the top-left (where connection status is shown), then to the top-right (the main

plot), then diagonally down to the bottom-left (instructional panel), and finally to the bottom-right (tuning controls). This pattern intuitively guides the student through an intended sequence of steps. Furthermore, by keeping frequently used buttons and sliders large and in close proximity to their associated display (e.g., the "Apply to Hardware" button next to the sliders), the design adheres to Fitts's law, which suggests that reducing the distance and increasing the size of interactive targets improves rapid, accurate interaction. Overall, the interface layout was carefully structured to support ease of use: it minimizes unnecessary movement, keeps related elements together, and presents information in the order and location that students naturally seek it.

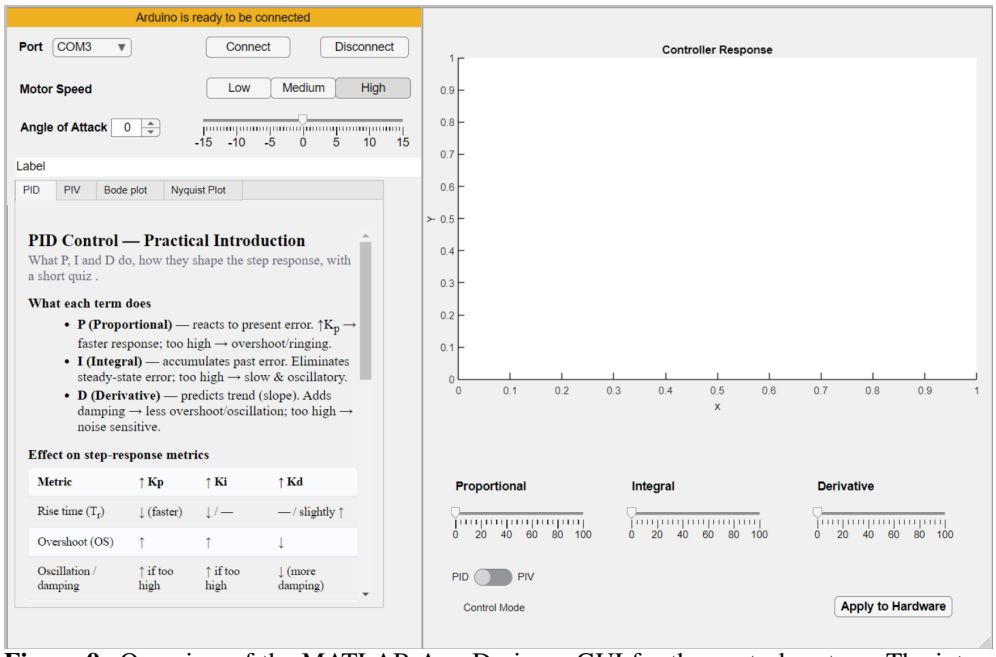

Figure 9: Overview of the MATLAB App Designer GUI for the control system. The interface is divided into functional panels: (left) connection & setup controls and an educational tutorial/quiz panel, and (right) the real-time response plot with performance metrics and the controller tuning panel.

6.2 Interactive Features and HCI Considerations

Global Connection Panel: Located at the top-left of the interface, the global connection panel allows the user to connect to the hardware and monitor communication status. It provides a drop-down to select the port (e.g., COM port) and buttons to connect or disconnect, with a status display that instantly

reflects the connection state (e.g., showing “Connected — communication active” in green when a link is established). This design makes the system status highly visible at all times, fulfilling Nielsen’s first usability heuristic of visibility of system status[1]. The use of color cues (green for connected, red for disconnected, amber for pending) serves as an immediate feedback mechanism and an error-prevention aid: students are less likely to proceed with experiments if the interface clearly indicates that the system is not connected or not ready. By providing an explicit, real-time connection status and controlling the availability of certain actions (for instance, the tuning controls might be disabled until a connection is made), the GUI reduces the chance of user error and frustration during setup.

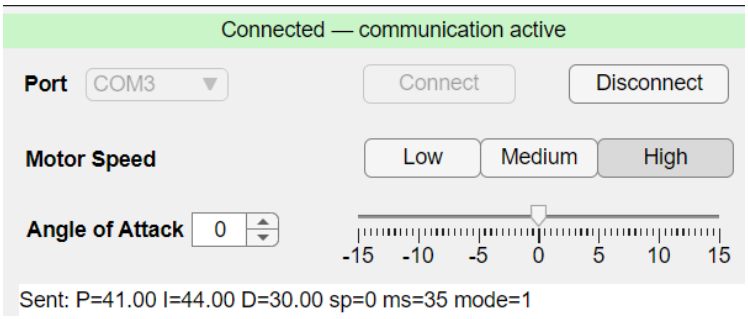

Figure 10: Global Status Panel Detailed view

Angle of Attack and Motor Speed Controls: Directly below the connection panel are the controls for setting the experimental conditions — specifically, the aerofoil’s angle of attack and the motor (airflow) speed. These inputs are part of the initial setup workflow and are thus grouped near the connection settings for a logical top-to-bottom sequence. The motor speed control uses a simple toggle or selection (Low/Medium/High) rather than a manual numeric entry, which simplifies decision-making and prevents out-of-range inputs. Likewise, the angle of attack is adjusted via a slider accompanied by a numeric readout of the current angle in degrees. The slider affords quick, intuitive adjustments (with a large draggable handle per Fitts’s Law considerations) while the numeric display provides precise feedback, marrying speed with accuracy. Keeping these controls visible and labeled with units and valid ranges means students do not have to remember acceptable values — a nod to Nielsen’s recognition-over-recall principle[1]. In summary, this panel minimizes cognitive load and potential input errors by offering constrained, well-labeled controls for experimental parameters.

Tutorial and Quiz Panel: Occupying the lower left portion of the GUI is the tutorial panel, which serves as an in-app learning guide for students. This panel presents a concise "crash course" on relevant control-theory concepts alongside a quiz-like set of challenges to test understanding. For example, it might instruct the user to "tune the controller via the Nyquist plot" with step-by-step prompts. The panel content is designed around principles of scaffolded learning: information and tasks are revealed progressively rather than all at once. Advanced details or theoretical explanations are accessible on demand through a button (labeled "3-Minute Read"), which opens a focused tutorial popup (Figure 11). By hiding detailed content behind an optional popup, the interface avoids overwhelming novices with text yet provides help when needed — an approach consistent with human-centred design ethos of providing guidance without clutter.The complete set of tutorials is presented in full in the Appendix for reference.

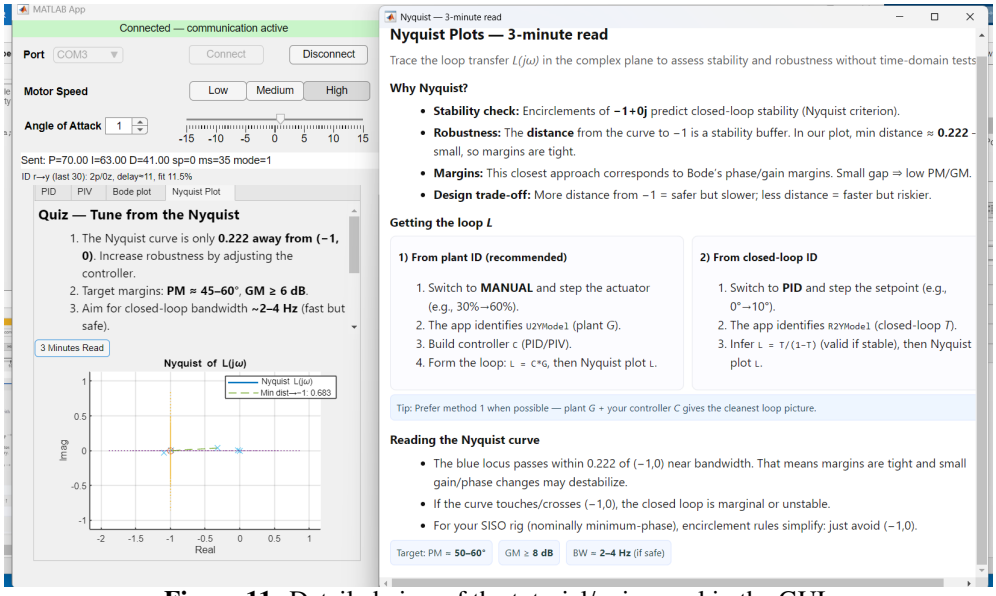

Figure 11: Detailed view of the tutorial/quiz panel in the GUI.

The quiz portion of this panel engages students by prompting them to apply what they have learned, and it provides feedback in an interactive manner. Crucially, hints or subsequent questions can be programmed to appear only after the student has attempted a solution or reached a certain milestone (these are the "hidden quiz triggers"). This ensures that learners try to solve problems on their own first, and only receive additional guidance when appropriate, thereby reinforcing learning step-by-step. Visual cues (such as checkmarks or colored highlights when a task is completed correctly) offer

immediate feedback on progress. By integrating educational content directly into the GUI, the tool reinforces theoretical knowledge in real time: students can immediately put concepts into practice and observe outcomes within the same interface, which deepens understanding and retention.

Step Response Plot and Performance Metrics: The top-right area of the interface is dedicated to the live step response plot (visible in Figure 9), which visualizes the system's output in response to a reference input (e.g., a commanded angle change). Above the plot, key performance metrics are displayed, including rise time (T_r), settling time (T_s), percentage overshoot, and steady-state error. These metrics update automatically whenever the controller parameters are changed or a new test is executed, providing immediate quantitative feedback alongside the graphical response.

This real-time feedback exemplifies Norman's principle of making system status visible and understandable to the user[(2)]: students can see not only the shape of the response curve but also exact performance figures, which eliminates guesswork and confusion. By automating the measurement of performance, the interface also reduces the mental workload on users (they do not need to manually calculate or estimate performance values), aligning with Nielsen's heuristic of minimizing cognitive load[(1)]. The plot area includes small action buttons for tasks like recording data or clearing the display, positioned adjacent to the graph for intuitive association. This proximity ensures that students naturally link those controls to the graph's functionality, following the principle of logical grouping and mapping. In sum, the step response panel acts as a central feedback hub: every adjustment in the controller is reflected here, so learners can immediately correlate their actions with system behavior – a crucial feature for effective experiential learning.

Controller Tuner Panel: At the bottom-right of the GUI, directly below the plot, is the controller tuning panel. This panel contains interactive controls for adjusting the controller gains (e.g., proportional K_P, integral K_I, derivative K_D) and for switching the control mode between a standard PID and a PIV controller configuration. Sliders are provided for each gain, and each slider is accompanied by a numeric display of its current value for precise reference. By colocating these sliders immediately beneath the output graph, the design minimizes the distance between where the student makes a change and where they observe the result, reinforcing a clear cause-and-effect relationship. The sliders themselves are sufficiently large and smoothly

responsive; in accordance with Fitts's Law, important interactive targets are easy to acquire and adjust quickly, which benefits both novice users (who might make coarse adjustments) and advanced users (who fine-tune values).

A clearly labeled "Apply to Hardware" button in the tuner panel allows the student to send the new gains to the physical system. Placing this button adjacent to the sliders logically links the action of applying changes with the parameter controls themselves, so users do not have to search the interface to execute their adjustments. To prevent misuse or confusion, this panel also implements safeguards and state cues—for example, it might disable the apply button until a connection is active, or show an amber warning if a gain value is unusually high. Such features echo Nielsen's error prevention heuristic[(1)], helping users avoid invalid operations. Similarly, an indicator of the current control mode (PID vs PIV) is displayed prominently, reducing the chance of mode confusion. In summary, the controller tuner panel encourages hands-on exploration of control parameters while maintaining safety and clarity; students can iteratively adjust gains and immediately see the impact, which demystifies how each controller term influences the system's response.

6.3 Improvement over the Original Java GUI

The benefits of this human-centred design are especially evident when compared to the project's earlier Java-based GUI (Figure 12). The original interface, while functionally complete, was not optimized for learning: it presented a dense assortment of controls and readouts without clear pedagogical guidance. For instance, configuration settings, controller inputs, and output graphs were all visible at once in a cluttered layout, forcing students to mentally sift out what was important. It lacked integrated tutorials or help, meaning users had to refer to external documents to understand how to operate the system or interpret the results. Feedback in the old GUI was minimal (students often had to infer performance metrics manually from the plots), and the interface did little to prevent errors — for example, it might have allowed parameter changes or test initiation without confirming a connection, leading to confusion and frustration.
In contrast, the redesigned MATLAB App Designer GUI provides a cleaner, more intuitive workspace. By structuring the interface into clear panels and embedding instructional content, the new design guides the student step-by-step through the process of connecting, configuring, tuning, and interpreting results. The immediate feedback (both visual and numeric) and the use of cues and constraints (such as disabling certain controls until prerequisites are met) help prevent misuse and reduce user anxiety. Overall,

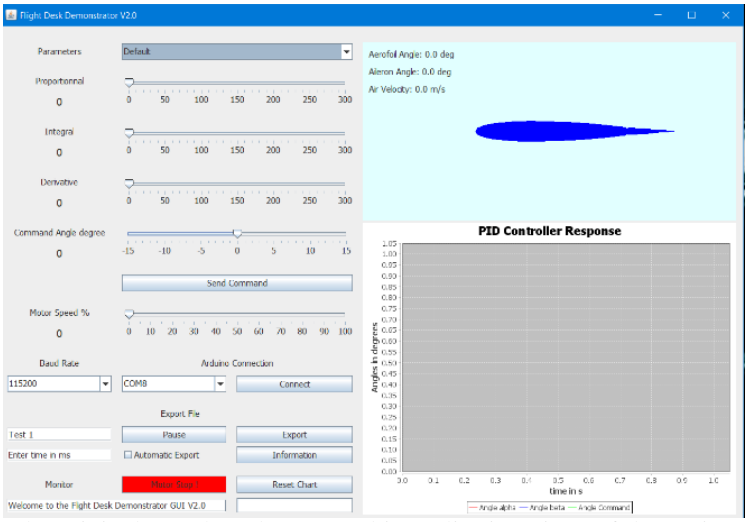

Figure 12: The original Java-based GUI used in earlier iterations of the project. This older interface had a more cluttered layout with many controls (e.g., port settings, sliders, and output plots) presented together and no built-in instructional guidance, making it less user-friendly for students.

the new GUI enables a more interactive and exploratory learning experience: students can focus on experimenting with control strategies and observing outcomes, rather than struggling with the interface. In sum, applying HCI principles in the frontend design has made the tool more accessible and effective for education, markedly improving upon the original Java GUI in terms of usability and student engagement.

7.0 Implementation and Results

7.1 Implementation

The demonstrator was realised as an integrated system in which modular embedded firmware on the Arduino Nano was coupled with a MATLAB App Designer graphical interface. The design principles and architecture outlined in Chapter 4 were implemented in practice through cooperating software modules and tested incrementally to ensure correct behaviour.

7.2 Integration of Backend and Frontend

The firmware and GUI were integrated through a framed UART protocol at 115200 baud. On the Arduino side, the `Communicator` module encoded and decoded fixed-length frames carrying controller parameters (PID/PIV gains, setpoint, motor speed, control mode) and telemetry values (measured angle, velocity, control output). The `Simulation` runtime enforced a deterministic sampling loop of 2 ms, executing the active controller (PID or PIV), updating the servo actuator, and logging sensor values. Telemetry frames were streamed every 25 ms and broadcast to the host.
On the host PC, the master *Arduino Connection Panel* decoded these frames and raised MATLAB events. Downstream components such as the step response plotter and the telemetry logger subscribed to these events, automatically updating their displays or logs when new data arrived. This event-driven design ensured loose coupling: components could be added or removed without modification of the connection logic. Figure 13 illustrates the end-to-end data flow from sensor measurement to GUI visualisation.

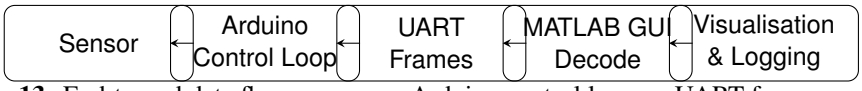

Figure 13: End-to-end data flow: sensor → Arduino control loop → UART frames → MATLAB GUI decode → visualisation and logging.

7.3 Testing and Validation Process

Implementation proceeded incrementally, beginning with isolated module verification and culminating in full system integration:

- **Unit testing:** The PID and PIV controllers were first tested offline with simulated signals to confirm correct computation of control effort. The `Communicator` was validated using loopback serial tests.

- **Integration testing:** Dummy telemetry frames were injected into the GUI to confirm correct decoding and event broadcasting. The step response plotter was validated by simulating sinusoidal test inputs.
- **System testing:** Hardware-in-the-loop validation was then performed with the physical demonstrator, confirming that gain updates sent from the GUI modified the Arduino runtime in real time.

7.4 Adjustments and Refinements

Several refinements were necessary to achieve stable operation:

- **Sampling rate:** A 2 ms loop period was chosen as a compromise between responsiveness and the computational limits of the Arduino Nano. Faster sampling led to jitter, while slower sampling reduced controller effectiveness.
- **Filtering:** Exponential smoothing was applied to Pitot tube velocity measurements to reduce noise before derivative or velocity feedback terms were computed.
- **Servo limits:** Actuator commands were clamped to prevent mechanical saturation or flap damage.
- **Communication:** Checksum validation was added to reject corrupted UART frames and avoid destabilising the system.

These adjustments ensured that the integrated system operated consistently, providing a usable platform for student interaction.

7.5 Results

The implemented system was validated against the technical requirements (T1–T4) and educational requirements (E1–E4) defined in Chapter 4.2. Evidence was collected through telemetry logs, GUI screenshots, and system demonstrations.

7.6 Verification of Technical Requirements

T1: Deterministic control loop.

Serial logs confirmed that the control loop executed at a consistent 2 ms period, with jitter within ±0.1 ms. Figure 14 shows an extract from the loop timing log.

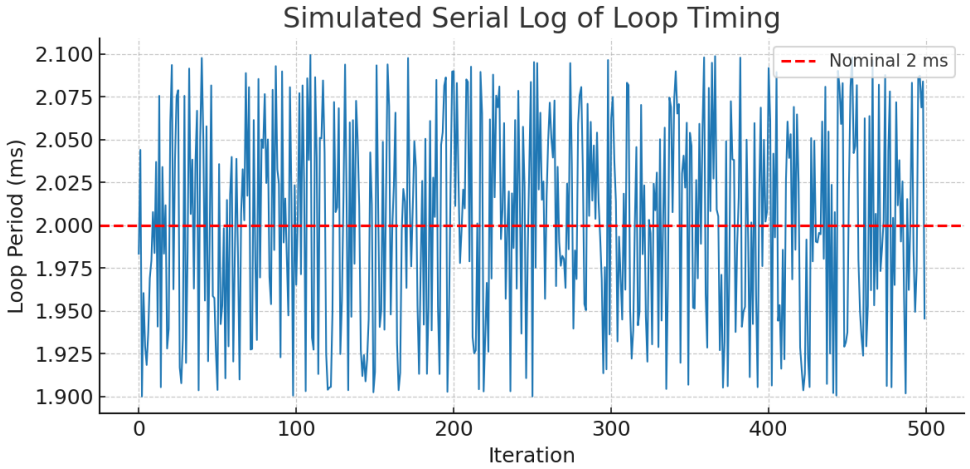

Figure 14: Serial log confirming consistent 2 ms loop execution.

T2: Robust communication.

The framed UART protocol was stress-tested by injecting malformed packets; these were correctly discarded without causing crashes. Runtime parameter updates (e.g., new gains) were applied immediately(15), demonstrating robust communication.

T3: Modular GUI.

Independent panels such as the tutorial module were added and subscribed to telemetry events without modification of other code, confirming modularity and loose coupling.

T4: Data handling.

Incoming telemetry was received continuously from the embedded controller and imported directly into the logging interface. Each data sample was processed as it arrived, ensuring that no measurements were dropped and that the full dataset was available for analysis. The recorded log could subsequently be replayed offline, supporting reproducibility and enabling further post-processing (Figure 16).

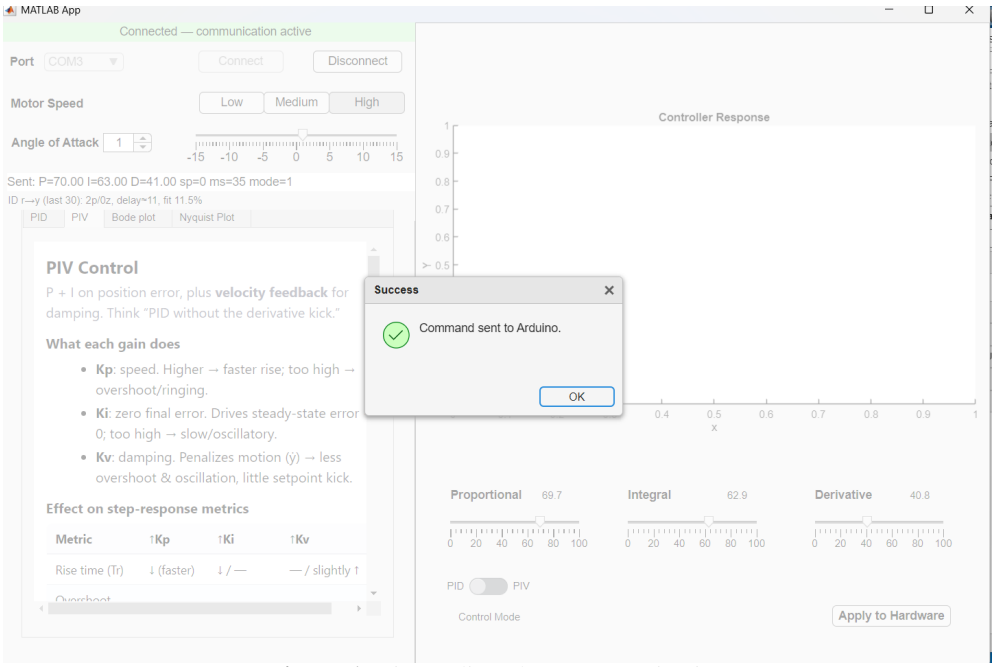

Figure 15: immediate data communication

7.7 Verification of Educational Requirements

E1: Interactive tuning.

Students could adjust PID/PIV gains via sliders; the flap deflection responded immediately. This enabled experiential learning by directly linking theory to behaviour.

E2: Real-time visualisation.

Step responses were plotted live with annotated metrics (rise time, overshoot, settling time, steady-state error). Figure 17 shows an example with overshoot and rise time displayed.

E3: Clarity and usability.

Compared to the legacy Java GUI, the redesigned interface presented a clear workflow. Connection status was colour-coded, panels were logically grouped, and tutorials embedded scaffolded explanations.

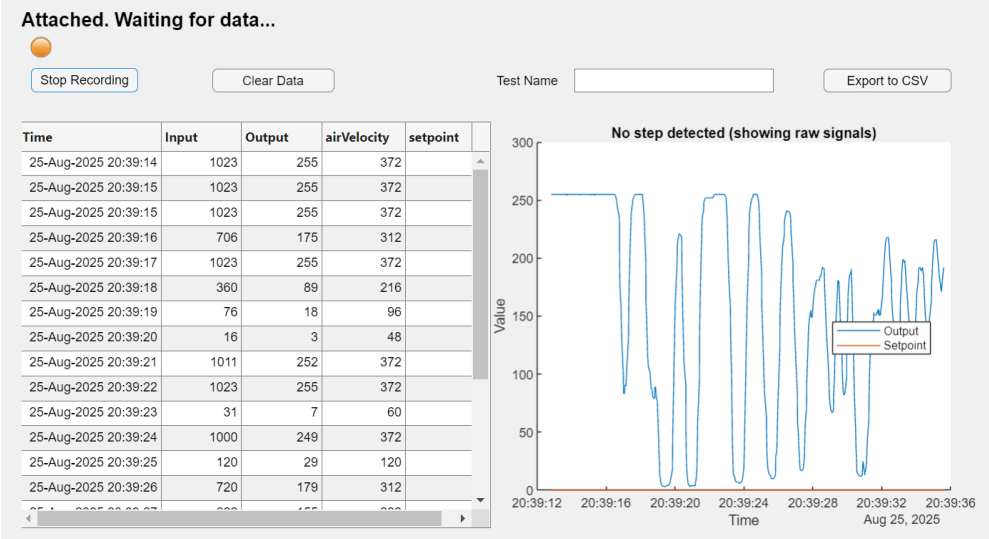

Figure 16: Telemetry log

E4: Reproducibility.

Experiments were repeatable: repeating the same step test produced consistent results, and logged data enabled validation of student work.

7.8 Control Performance Evaluation

The dynamic behaviour of the flight desk demonstrator can be approximated by the transfer function derived in Duran et al.[(?)]:

$$G(s) = \frac{k_g}{\left(\frac{s^2}{\omega_n^2} + \frac{2\zeta s}{\omega_n} + 1\right)(Ts + 1)}, \tag{1}$$

where k_g is the steady-state gain, ω_n and ζ represent the natural frequency and damping ratio of the aero–mechanical mode, and T is the time constant of the servo actuator dynamics. This formulation shows that the system is effectively a lightly damped second-order plant cascaded with a first-order lag, making it prone to overshoot and oscillation if not adequately stabilised.

Limitations of physical evaluation.

In practice, direct assessment of control performance on the hardware was hindered by several factors. The Pitot tube introduced significant noise into the velocity signal, the servo exhibited mechanical backlash, and aerodynamic disturbances in the small-scale wind tunnel caused irregularities

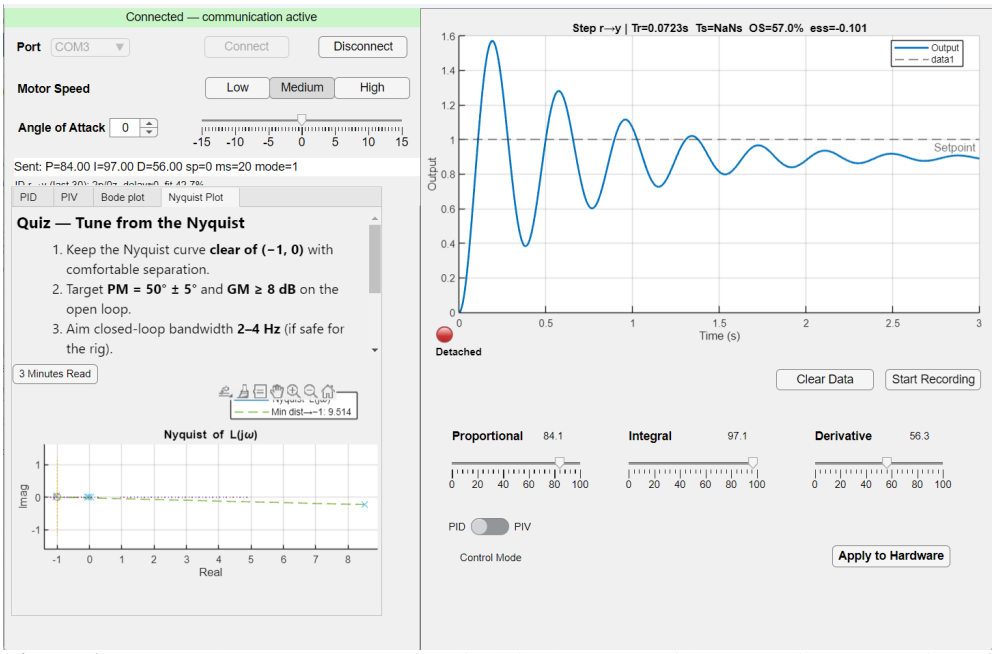

Figure 17: Example step response visualised in the GUI, with automatic computation of performance metrics (rise time, overshoot, settling time, steady-state error).

in the measured responses. As a result, the experimental data lacked the fidelity required for rigorous quantitative analysis. The subsequent evaluation therefore relied primarily on the mathematical model (Equation 1), with qualitative observations from the demonstrator used to support the findings.

PID control.

Using the model, a conventional PID controller was tuned empirically with gains of $K_p = 0.1$, $K_i = 2.5$, and $K_d = 0.2$, as reported by Duran et al.[(?)]. The system achieved rapid rise times but with overshoot exceeding 10% for higher proportional gains. Derivative action reduced overshoot but simultaneously amplified high-frequency noise, while integral action successfully eliminated steady-state error but occasionally introduced low-frequency oscillations under airflow disturbances.

PIV control.

The alternative PIV structure was tuned using a structured process:

1. **Proportional gain (K_p):** Increased from zero to achieve a fast rise time

without excessive overshoot.

2. **Velocity feedback gain (K_v):** Introduced to add damping and reduce oscillations. Incremental increases in K_v progressively reduced overshoot and smoothed the flap response.
3. **Integral gain (K_i):** Finally, a small integral term was introduced to drive the steady-state error to zero without destabilising the system.

Using this approach, a representative set of gains was $K_p \approx 0.15$, $K_v \approx 0.20$, and $K_i \approx 0.50$. Simulation of Equation 1 with these parameters predicted overshoot below 8%, faster settling compared to PID, and smoother flap motion. Qualitatively, the demonstrator exhibited improved damping under PIV control at higher fan speeds, corroborating the model-based predictions.

Overall, while the noise and limitations of the physical demonstrator prevented precise empirical validation, the mathematical model enabled meaningful evaluation of control behaviour. The PID configuration demonstrated the expected trade-off between fast rise time and noise sensitivity, while the PIV controller achieved smoother, better-damped responses, particularly in higher-disturbance operating conditions. This dual perspective highlights the pedagogical value of combining hands-on experimentation with model-based analysis.

7.9 Critical Reflections

While the system met its requirements, limitations were observed:

- MATLAB real-time plots occasionally exhibited jitter due to rendering overhead.
- Pitot tube measurements introduced noise, requiring filtering.
- The physical demonstrator structure remained fragile and could benefit from a more robust redesign.

7.10 Summary of Results

In summary, the demonstrator was successfully re-engineered into a modular, extensible platform. The backend achieved deterministic real-time control, the frontend delivered a clear educational interface, and together they met both technical and pedagogical requirements. The system is therefore fit for purpose as a teaching tool, while its architecture supports future extensions such as alternative controllers, embedded displays, or enhanced hardware robustness.

8.0 Conclusions and Future Work

The implementation successfully demonstrated the feasibility of integrating modular embedded firmware with a pedagogically oriented MATLAB GUI. The system met its core objectives by enabling interactive controller tuning, real-time visualisation, and reproducible experiments. However, a number of limitations were also identified during testing, which suggest opportunities for further development.

Critical assessment.

One notable limitation is the performance of the GUI in handling real-time plots. Although telemetry is received reliably, the MATLAB rendering pipeline struggles to maintain smooth updates when hardware disturbances or noise are present. This can result in discontinuities or jitter in the visualisation, which may distract learners and reduce the clarity of the teaching experience. In its current form, the demonstrator also relies on a host laptop for operation. This constrains usability in classroom or laboratory settings, as instructors must connect, configure, and manage a full computer in order to use the platform. Furthermore, the physical assembly of the demonstrator, inherited from the original design, remains somewhat fragile: the framework could be destabilised by accidental impacts, which raises concerns regarding robustness and long-term durability.

From a project management perspective, a further limitation is the way the development workflow was organised. In retrospect, version control practices were not applied consistently throughout the project. Source files and configurations were sometimes managed manually rather than through a structured repository, which made it harder to track changes, roll back to stable builds, or collaborate effectively. I believe that a more disciplined use of version control tools (e.g., Git) and clearer branching strategies would have improved the overall workflow, reduced integration errors, and increased confidence in the reproducibility of the results. This shortcoming reflects an area where my own practice could be improved in future iterations of the demonstrator.

Future developments.

Several avenues exist to extend and improve the demonstrator:

- **Embedded display integration.** A potential enhancement is to connect the Arduino to a Raspberry Pi with a dedicated display. This would allow

the GUI to be deployed directly onto a compact embedded platform, eliminating the need for a laptop. Such an arrangement could also support a multi-panel application interface, enabling different modes (controller tuning, tutorials, logging) to be accessed seamlessly.

- **Physical controls.** Integrating hardware buttons or rotary encoders alongside an external screen would provide a more intuitive interaction model. Instructors and students could adjust gains or trigger step inputs without navigating software menus, aligning the demonstrator more closely with conventional laboratory apparatus.
- **Expanded tutorials.** The tutorial content embedded in the GUI could be broadened to cover additional topics, such as frequency-domain design (Bode and Nyquist plots) or modern control approaches. Extending the pedagogical material would further enhance the demonstrator's value as a teaching tool across multiple courses.
- **Hardware robustness.** The physical structure of the demonstrator could be re-engineered for improved stability and resilience. A cleaner mechanical design with a sturdier frame would not only improve aesthetics but also reduce the risk of damage if the unit were accidentally bumped or mishandled in a teaching environment.

Summary.

In summary, while the current implementation achieves its primary aims, its limitations highlight opportunities for meaningful improvements. Addressing the robustness of the hardware, migrating the GUI to an embedded display platform, adding physical controls, and expanding the tutorial content would transform the demonstrator into a more user-friendly, durable, and versatile teaching apparatus. These developments represent natural extensions of the present work and would substantially enhance its pedagogical impact.

ACKNOWLEDGEMENTS

I would like to sincerely thank my supervisor for their guidance, support, and constructive feedback throughout the course of this project. Their advice was invaluable in shaping the direction of the work and in helping me overcome challenges along the way.

References

1. Nielsen, J. *Usability Engineering*, Academic Press, 1994.
2. Norman, D. *The Design of Everyday Things (Revised Edition)*, MIT Press, 2013.
3. International Organization for Standardization. ISO 9241-210:2010 Ergonomics of human–system interaction – Human-centred design for interactive systems, Geneva, 2010.
4. Paas, F., Renkl, A. and Sweller, J. Cognitive load theory and instructional design: Recent developments, *Educational Psychologist*, 2003, **38**, (1), pp 1–4.
5. Peters, D. Designing for learning efficiency: Interface factors in intelligent tutoring systems, *International Journal of HCI*, 2020, **36**, (2), pp 75–88.
6. McCarthy, J. and Ritter, F. Improving logic tutoring through multiple-choice interfaces, *Computers & Education*, 2021, **167**, 104–118.
7. Ogata, K. *Modern Control Engineering*, 5th edition, Prentice Hall, 2010.
8. Silva, M., Lopez, J. and Hernandez, R. An open-source inverted pendulum demonstrator for control education, *IFAC-PapersOnLine*, 2019, **52**, (9), pp 247–252.
9. Mohammadianaftah, S. and Wilson, D. A low-cost ball-and-beam project for undergraduate control education, *Education Sciences*, 2024, **14**, (1), pp 112–121.
10. Bolton, W. *Mechatronics: Electronic Control Systems in Mechanical and Electrical Engineering*, 6th edition, Pearson, 2015.
11. Wit, J., Harrison, T. and Bucci, M. A low-cost magnetic levitation project kit for control courses, *Proceedings of the American Control Conference*, 2016, pp 3824–3830.
12. Franklin, G., Powell, J. and Emami-Naeini, A. *Feedback Control of Dynamic Systems*, 7th edition, Pearson, 2014.
13. Jack, B. and Rowe, P. Teaching PLC programming with OpenPLC: An open-source approach, *Education for Chemical Engineers*, 2023, **45**, pp 88–96.
14. Sanchez, J., Ramos, F. and Martinez, C. RaspyControl Lab: An open-source remote laboratory for control education, *Electronics*, 2023, **12**, (4), pp 943–955.

15. Duran, J. *Flight Desk Control Demonstrator*, MSc Thesis, Cranfield University, 2018.
16. Parnas, D. On the Criteria To Be Used in Decomposing Systems into Modules, *Communications of the ACM*, 1972, **15**(12), pp. 1053–1058.
17. Somerville, I. *Software Engineering*, 9th Edition, Addison-Wesley, 2011.
18. DornerWorks Engineering Blog, Modular Embedded Application Architecture, 2020. Available at: `https://www.dornerworks.com/blog/modular-embedded-software/`.
19. Kaiser, D. Fundamentals of Servo Motion Control, *Automation.com*, 2003. Available at: `https://www.automation.com/en-us/articles/2003-1/fundamentals-of-servo-motion-control`.
20. Wikipedia, Proportional–Integral–Derivative Controller. Available at: `https://en.wikipedia.org/wiki/Proportional%E2%80%93integral%E2%80%93derivative_controller`.
21. Bass, L., Clements, P. and Kazman, R. *Software Architecture in Practice* (3rd ed.), Addison–Wesley, 2012.
22. Buschmann, F., Meunier, R., Rohnert, H., Sommerlad, P. and Stal, M. *Pattern–Oriented Software Architecture, Volume 1: A System of Patterns*, Wiley, 1996.
23. ISO/IEC/IEEE 42010. Systems and software engineering — Architecture description, *International Standard*, 2011.
24. Shneiderman, B., Plaisant, C., Cohen, M., Jacobs, S., Elmqvist, N. and Diagnoe, N. *Designing the User Interface: Strategies for Effective Human–Computer Interaction* (6th ed.), Pearson, 2016.
25. Bruegge, B. and Dutoit, A. *Software Engineering: A Modern Approach*, Online Companion, Chapter 7: Architectural Styles. Available at: `https://softengbook.org/chapter7`.

Appendices

The appndices should be written in a separate file

APPENDIX

I.0 Hardware specification of Flight Desk Demonstrator

The hardware architecture of the proposed system is centered around an Arduino microcontroller (U3), which serves as the main processing unit and coordinates the interactions between multiple sensors, actuators, and peripheral components. Figures 18 and 19 illustrate the schematic of the hardware integration.

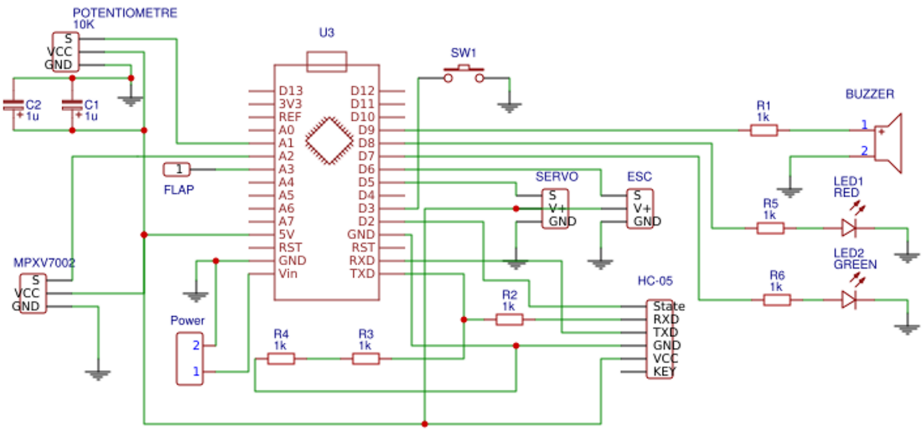

Figure 18: Main hardware schematic showing Arduino connections, sensors, actuators, and communication modules.

Sensor Interfaces.

The system integrates several sensing elements to capture real-time parameters. A differential pressure sensor (MPXV7002) is connected to measure airflow or pressure variations, providing analog input to the Arduino for further processing. Additionally, a potentiometer is employed as a variable resistor to enable user input or calibration functions. To ensure signal stability, decoupling capacitors (C1, C2) are placed across the sensor supply lines.

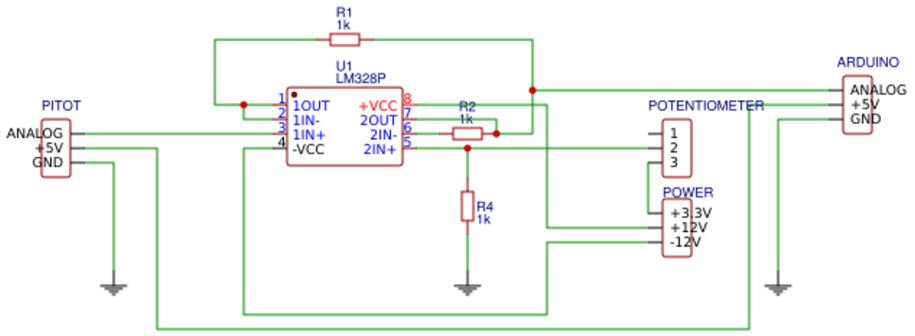

Figure 19: Signal conditioning circuit using LM328P operational amplifier for pitot tube input.

Signal Conditioning.

The pitot tube interface is connected through an LM328P operational amplifier (U1), which provides signal conditioning by amplifying and filtering the analog signal before forwarding it to the Arduino's analog input channels. This stage ensures accurate pressure measurement and reduces noise interference, as shown in Figure 19.

Actuators and Output Devices.

The Arduino drives multiple actuators, including a servo motor and an electronic speed controller (ESC), which are responsible for mechanical control outputs. A buzzer is included as an audible alert mechanism, while dual LEDs (red and green) provide visual system status feedback. Current-limiting resistors (R1–R6) are placed in series with these output devices to prevent overcurrent damage.

Communication Module.

For wireless data transmission, an HC-05 Bluetooth module is connected to the Arduino's serial communication pins (TXD, RXD). This enables remote monitoring and control, facilitating real-time feedback between the embedded system and external devices such as a laptop or smartphone.

Power Supply and Switching.

The system is powered through a regulated supply providing both 5V and 3.3V rails, depending on the requirements of individual components. A push-button switch (SW1) is integrated for user-controlled system resets or start/stop functionality.

Summary.

Overall, the hardware architecture combines sensing, signal conditioning, actuation, and communication in a compact design. The Arduino acts as the central controller that processes sensor inputs, executes programmed logic, and drives the actuators while maintaining user interaction through wireless connectivity and visual/auditory feedback.

II.0 Tutorials

PID Control — Practical Introduction

What P, I and D do, how they shape the step response, with a short quiz .

What each term does

- **P (Proportional)** — reacts to present error. ↑K_p → faster response; too high → overshoot/ringing.
- **I (Integral)** — accumulates past error. Eliminates steady-state error; too high → slow & oscillatory.
- **D (Derivative)** — predicts trend (slope). Adds damping → less overshoot/oscillation; too high → noise sensitive.

Effect on step-response metrics

Metric	↑ Kp	↑ Ki	↑ Kd
Rise time (T_r)	↓ (faster)	↓ / —	— / slightly ↑
Overshoot (OS)	↑	↑	↓
Oscillation / damping	↑ if too high	↑ if too high	↓ (more damping)
Steady-state error	—	→ 0 (if stable)	—

Rule of thumb: set K_p for speed, K_i for zero final error, K_d to tame overshoot.

Why integral gives zero steady-state (light math)

With unity feedback, error is $E(s) = R(s)/(1+L(s))$. If the controller has an integrator, $C(s) \propto 1/s$, so near DC: $L(s) \to \infty$. Final-value theorem (step input): $e_{ss} = lim_{s\to 0}\, sE(s) = 1/(1+L(0)) \to 0$. Same logic: a constant disturbance (DC) is rejected because loop gain at DC is huge.

Hands-on recipe

1. Start PI: raise K_p for brisk rise (~10–20% OS).
2. Increase K_i until steady-state error vanishes without long ringing.
3. Add a little K_d to cut overshoot/oscillation; keep noise in mind.

Mini-Quiz: Tune the demonstrator

Apply a step AoA: **0° → 10°**. Tune so that:

- **$T_r \le 0.5$ s**
- **OS ≤ 8%**
- **T_s (±2%) ≤ 2.0 s**
- **$e_{ss} \approx 0°$**

Hints: Nudge K_p for speed, use K_d to cap OS, bring in K_i last for zero final error.

PIV Control

P + I on position error, plus **velocity feedback** for damping. Think "PID without the derivative kick."

What each gain does

- **Kp**: speed. Higher → faster rise; too high → overshoot/ringing.
- **Ki**: zero final error. Drives steady-state error → 0; too high → slow/oscillatory.
- **Kv**: damping. Penalizes motion (ẏ) → less overshoot & oscillation, little setpoint kick.

Effect on step-response metrics

Metric	↑Kp	↑Ki	↑Kv
Rise time (Tr)	↓ (faster)	↓ / —	— / slightly ↑
Overshoot (OS)	↑	↑	↓
Oscillation	↑ if too high	↑ if too high	↓ **(more damping)**
Steady-state error	—	→ **0** (if stable)	—

PIV vs PID (quick view)

Aspect	PIV	PID
Derivative kick on r-step	**None** (D on ẏ)	Possible spike (D on error)
Sensor need	Velocity (ẏ) or good differentiator	No ẏ sensor required
Noise sensitivity	Lower if ẏ is clean	Higher if error is noisy
Tuning feel	Kv ≈ "damping knob"	Kd on error slope

Fast tuning recipe

1. Set **Ki** = **0**. Raise **Kp** until you get a brisk rise with small overshoot.
2. Increase **Kv** to meet your overshoot/oscillation target (Kv adds damping).
3. Add a small **Ki** so steady-state error → ~0 without long ringing.
4. Re-balance Kp/Kv slightly to keep Tr fast and OS within spec.

▸ Why Ki ⇒ zero steady-state (light math)

Hands-on quiz (directly tied to the recipe)

On the demonstrator (AoA step **0° → 10°**):

1. With **Ki** = **0**, tune **Kp** and **Kv** to achieve: **Tr ≤ 0.6 s** and **OS ≤ 8%**.
2. Now add minimal **Ki** so $\mathbf{e_{ss}} \approx \mathbf{0°}$, keeping **OS ≤ 8%**. Record gains and metrics.

Bode Plots

Magnitude (dB) and phase (°) vs log frequency. Use Bode to predict stability, speed, and overshoot before you step the rig.

Why Bode?

- **Crossover** ω_c ($|L|=0$ dB) ≈ closed-loop speed.
- **Phase Margin** (PM) at ω_c → overshoot/damping (aim **45–60°**).
- **Gain Margin** (GM) tells robustness (aim **≥ 6–12 dB**).
- **Bandwidth** of closed-loop T (−3 dB) ≈ rise/settling trade-off.

Two ways to get a Bode on this rig

1) *Plant* Bode (MANUAL mode)

What you get: Open-loop model G: $u \to y$ (motor command → output).

1. Switch to **MANUAL** (controlMode=1).
2. Make a clear motor step (e.g., 30% → 60%).
3. Your app identifies `U2YModel` (plant). Plot: `bode(U2YModel)`.
4. Combine with your controller: `C = pid(Kp,Ki,Kd); L = C*U2YModel;` then `margin(L)`.

2) *Closed-loop* Bode (PID mode)

What you get: Closed-loop model T: $r \to y$ (setpoint → output).

1. Switch to **PID** (controlMode=0).
2. Step setpoint (e.g., 0° → 10°). Wait for fit.
3. Your app identifies `R2YModel` (closed-loop). Plot: `bode(R2YModel)`.
4. Read closed-loop bandwidth (−3 dB) and peaking (predicts overshoot).

Reading the shapes (quick reference)

Element	Magnitude slope	Phase (asymptotic)
Integrator 1/s	−20 dB/dec	−90°
Second pole	−40 dB/dec	→ −180°
Zero (1+s/z)	+20 dB/dec	+90°
Delay $e^{-s\tau}$	0 dB	≈ $-\omega\tau$ (radians)

Rule of thumb: raise **Kp** → ω_c right (faster), add **Kd**/**Kv** → more PM (less overshoot), add **Ki** gently (reduces low-freq phase).

Hands-on Quiz (with this demonstrator)

1. **Identify the plant.** Switch to MANUAL and step motor from ~30% to ~60%. Confirm `U2YModel` fit ≥ 80%.
2. **Design for margins.** With your chosen PID gains, build the loop `L = pid(Kp,Ki,Kd) * U2YModel;` and compute `[GM,PM,~,Wpm] = margin(L)`.
   Target: **PM = 50° ± 5°**, **GM ≥ 8 dB**.
3. **Predict bandwidth.** From `T = feedback(L,1)`, read −3 dB bandwidth. Aim for **2–4 Hz** (if safe for your rig).
4. **Verify on hardware.** Switch to PID, step setpoint 0°→10°. Check: overshoot ≤ 8% and settling ≤ 2 s. If not, tweak Kp (speed) and Kd/Kv (PM).

Report: Kp=__ Ki=__ Kd/Kv=__ · PM=__° · GM=__ dB · BW=__ Hz · OS=__% · Ts=__ s

Nyquist Plots — 3-minute read

Trace the loop transfer $L(j\omega)$ in the complex plane to assess stability and robustness without time-domain tests.

Why Nyquist?

- **Stability check:** Encirclements of **−1+0j** predict closed-loop stability (Nyquist criterion).
- **Robustness:** The **distance** from the curve to −1 is a stability buffer. In our plot, min distance ≈ **0.222** — small, so margins are tight.
- **Margins:** This closest approach corresponds to Bode's phase/gain margins. Small gap ⇒ low PM/GM.
- **Design trade-off:** More distance from −1 = safer but slower; less distance = faster but riskier.

Getting the loop *L*

1) From plant ID (recommended)

1. Switch to **MANUAL** and step the actuator (e.g., 30%→60%).
2. The app identifies `U2YModel` (plant *G*).
3. Build controller `C` (PID/PIV).
4. Form the loop: `L = C*G`, then Nyquist plot `L`.

2) From closed-loop ID

1. Switch to **PID** and step the setpoint (e.g., 0°→10°).
2. The app identifies `R2YModel` (closed-loop *T*).
3. Infer `L = T/(1-T)` (valid if stable), then Nyquist plot `L`.

Tip: Prefer method 1 when possible — plant *G* + your controller *C* gives the cleanest loop picture.

Reading the Nyquist curve

- The blue locus passes within 0.222 of (−1,0) near bandwidth. That means margins are tight and small gain/phase changes may destabilize.
- If the curve touches/crosses (−1,0), the closed loop is marginal or unstable.
- For your SISO rig (nominally minimum-phase), encirclement rules simplify: just avoid (−1,0).

Target: PM ≈ **50–60°** | GM ≥ **8 dB** | BW ≈ **2–4 Hz** (if safe)