

Modernisation of Water Tank Control Laboratory System



Mohamad Ajan

Division of Industrial Electrical Engineering and Automation
Faculty of Engineering, Lund University

Modernisation of a Water Tank Control Laboratory System



LUNDS UNIVERSITET
Lunds Tekniska Högskola

LTH at Campus Helsingborg

Department of Electrical Engineering and Automation

Bachelor thesis:

Mohamad Ajan

© Copyright Mohamad Ajan

LTH at Campus Helsingborg
Lund University
Box 882
SE-251 08 Helsingborg
Sweden

LTH at Campus Helsingborg
Lund University
Box 882
251 08 Helsingborg

Printed in Sweden
Lund University
Lund 2026

Abstract

This thesis presents an open and documented replacement for the two-tank water level laboratory used to teach PID control at LTH Helsingborg. The existing system depends on a proprietary, closed-source Windows application and a legacy data acquisition card, tied to an ageing Windows XP-era PC that cannot be modified, documented, or used remotely. The new platform pairs a Raspberry Pi Pico W microcontroller, running MicroPython as a data-acquisition bridge, with a Python web application built on Flask and Flask-SocketIO, which students reach through a browser on any device. A complete PID controller reproduces the functionality of the original software, and the platform offers two operating modes: a hardware mode that drives the physical tanks, and a simulation mode that replaces them with a mathematical model so the laboratory exercises can be completed entirely remotely. It is delivered in two editions a lab edition for the workstations and a student edition with a self-contained executable for students' own computers.

Testing confirmed reliable communication with the microcontroller at 10 Hz, correct operation of a three-layer pump-safety architecture, and a simulation that matches its analytical steady-state behaviour, while six unit tests of the controller and model all pass. Full closed-loop control of the physical tank was not completed, as it requires a signal conditioning circuit that remains as future work. The result modernises the laboratory onto current Windows 11 hardware and, through its simulation mode, makes the PID control exercises accessible to students remotely for the first time.

Keywords: *PID control, Anti-windup, Water tank system, Simulation, Raspberry Pi Pico, Remote laboratory, Flask, Open-source*

Sammanfattning

Detta examensarbete presenterar en öppen och dokumenterad ersättning för det tvåtankssystem som används i undervisningen av PID-reglering vid LTH Helsingborg. Det befintliga systemet bygger på en Windows-applikation med sluten källkod och ett äldre datainsamlingskort som är kopplade till en dator från Windows XP-eran. Eftersom systemet är gammalt är det svårt att vidareutveckla, dokumentera och använda på distans.

Den nya lösningen består av en Raspberry Pi Pico W som samlar in och skickar mätdata samt en webbapplikation utvecklad i Python med Flask och Flask-SocketIO. Studenterna använder systemet direkt i webbläsaren på valfri enhet. Plattformen innehåller en PID-regulator som motsvarar funktionerna i den ursprungliga programvaran och erbjuder två driftlägen. I hårdvaruläget styrs de fysiska tankarna, medan simuleringsläget använder en matematisk modell i stället för den verkliga anläggningen. Detta gör att laborationerna kan genomföras både i laboratoriet och på distans. Plattformen finns både som en version för laborations datorerna och som en fristående version för studenternas egna datorer.

Tester visade att kommunikationen med mikrokontrollern fungerar stabilt vid 10Hz, att pumpens säkerhetssystem fungerar som avsett och att simuleringen överensstämmer med modellens analytiska jämviktsläge. Samtliga sex enhetstester för regulatören och modellen godkändes.

Arbetet moderniserar laborationen så att den kan användas på dagens Windows 11-datorer och förbättrar studenternas interaktion med systemet genom ett mer lättillgängligt gränssnitt. Genom simulering läget kan PID-laborationerna genomföras på distans för första gången, vilket ger studenterna större tillgång till laboration miljön och fler möjligheter att utforska och förstå reglertekniska koncept även utanför laborationerna.

Nyckelord: *PID-reglering, Anti-windup, Vattentank, Simulering, Raspberry Pi Pico, Distance Laboratorium, Flask, Open-source*

Abbreviations

ADC	Analogue-to-digital converter
API	Application Programming Interface
DAQ	Data Acquisition
GIL	Global Interpreter Lock
HTTP	Hypertext Transfer Protocol
LPF	Low-pass filter
NDJSON	Newline-Delimited JSON
ODE	Ordinary Differential Equation
PID	Proportional–Integral–Derivative controller
PWM	Pulse Width Modulation
RLock	Re-entrant lock
RMSE	Root Mean Square Error
ss	Steady state

Definitions

Anti-windup: A mechanism that suspends integral action while the actuator is saturated, preventing the integral term from accumulating to an unrecoverable value.

Dead time: (transport delay) A pure delay between a control action and its first observable effect on the process.

Derivative kick: A large, brief spike in the control output caused by the derivative term reacting to a step change in the setpoint.

Soft-idle ramp: A firmware safety routine that linearly reduces the pump output to zero when no valid command has been received within a set interval.

Stale-link fault: A fault the control loop declares when the most recent telemetry sample exceeds a defined age threshold.

Watchdog timer: A hardware timer that resets the microcontroller if it is not periodically refreshed, recovering the system from a hang or crash.

Table of Contents

List of Figures.....	7
List of Tables.....	8
Chapter 1 — Introduction.....	9
1.1 Background.....	9
1.2 Purpose.....	11
1.3 Goals.....	12
1.4 Research Questions.....	12
1.5 Motivation.....	13
1.6 Delimitations.....	14
1.7 Summary of Contributions.....	14
Chapter 2 — Theoretical Background.....	16
2.1 PID Control.....	16
2.2 Two-Tank Liquid Level System.....	22
2.3 Real-Time Web Communication.....	25
Chapter 3 — Method.....	27
3.1 Development Approach.....	27
3.2 Literature Study.....	28
3.3 Testing Strategy.....	29
3.4 Source Evaluation.....	30
Chapter 4 — System Design and Implementation.....	33
4.1 Architecture Overview.....	33
4.2 Raspberry Pi Pico Firmware.....	35
4.3 Communication Protocol.....	39
4.4 PID Controller Module.....	41
4.5 Tank Simulation Model.....	42
4.6 Flask Backend and Control Loop.....	44
4.7 Browser Interface.....	47
4.8 Safety Architecture.....	49
4.9 Security Considerations.....	50
4.10 Deployment and Packaging.....	51
Chapter 5 — Results and Testing.....	54
5.1 Unit Tests: PID Controller and Simulation Model.....	54
5.2 Hardware Path Testing.....	57
5.3 Simulation Validation.....	58
5.4 Deployment Testing.....	60
5.5 Summary of Completed and Open Work.....	63
Chapter 6 — Discussion and Conclusions.....	64
6.1 Design Decisions and Trade-offs.....	64
6.2 Deviations from the Initial Specification.....	66
6.3 Limitations.....	66
6.4 Answers to Research Questions.....	67
6.5 Research Questions Revisited.....	68
6.6 Ethical Considerations.....	71
6.7 Future Work.....	72
6.8 Conclusion.....	73
Bibliography.....	74

List of Figures

Figure 1.1. The physical two-tank water system at LTH Helsingborg.....	9
Figure 1.2. The Velleman VM167 Interface Board used in the original laboratory setup.....	10
Figure 1.3. The pid_velleman application interface running on the legacy Windows XP system.....	10
Figure 2.1. Block diagram of PID controller.....	16
Figure 2.2. Schematic of the two-tank system.....	22
Figure 2.3. The Raspberry Pi Pico W microcontroller board used as the DAQ bridge in this thesis.....	26
Figure 4.1. High-level data flow in hardware mode and in simulation mode.....	34
Figure 4.2. Example telemetry sample frame.....	39
Figure 4.3. Example startup hello frame.....	40
Figure 4.4. Example command frames sent from the PC to the Raspberry Pi Pico.....	40
Figure 4.5. The browser interface of the new laboratory system. Numbered components are described in Section 4.7.....	48
Figure 5.1. Simulated closed-loop step response, $SP=0.5$, $K=0.8$, $T_i=15s$, $T_d=2.0s$. Y-axis: normalised level [0,1]. X-axis: time [s].....	55
Figure 5.2. Simulated closed-loop step response, $SP=0.75$, $K=10$, $T_i=0s$, $T_d=0s$. Y-axis: normalised level [0,1]. X-axis: time [s].....	56
Figure 5.3. Real-time simulation UI capturing the visual tank overflow during peak overshoot.....	56
Figure 5.4. The updated laboratory setup running the new software environment on a university Windows 11 workstation.....	61

List of Tables

Table 4.1. Tank model parameters (engineering estimates).....	40
Table 4.2. Python modules comprising the Flask application.....	42
Table 4.3. Three-layer pump cutoff safety architecture.....	46
Table 5.1. Unit test results.....	51
Table 5.2. Summary of completed and open work.....	60

Chapter 1 — Introduction

This chapter introduces the laboratory system. It describes the background and the limitations of the existing equipment, states the purpose and goals of the work, formulates the research questions, motivates the choice of subject, and defines the delimitations of the thesis.

1.1 Background

This thesis is carried out in collaboration with the Department of Electrical Engineering and Automation at LTH Helsingborg, which both hosts the physical laboratory equipment and supervises the work. The department uses a physical two-tank water system for laboratory exercises in control engineering, offering students practical experience with PID controllers and how proportional, integral, and derivative terms affect a real physical process [1].

The water tank system consists of two coupled tanks, a 15V DC pump (Airpax), and capacitance-based level sensors, as shown in Figure 1.1.

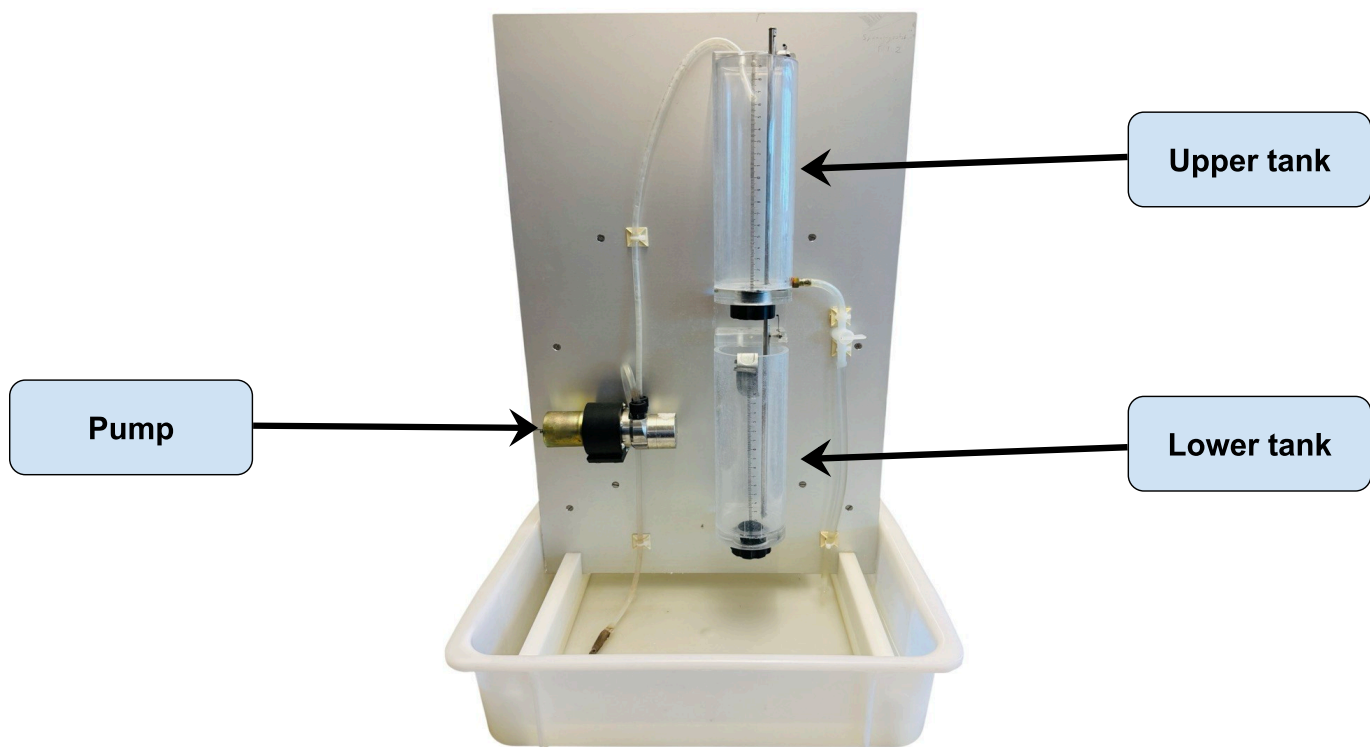


Figure 1.1. The physical two-tank water system at LTH Helsingborg

A custom interface board conditions the electrical signals between the tank hardware and a data acquisition (DAQ) card. The Velleman VM167 Interface Board, shown in Figure 1.2 which connects to a PC via a proprietary interface cable.

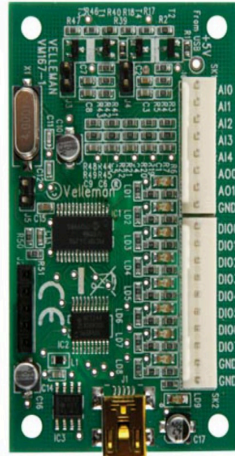


Figure 1.2. The Velleman VM167 Interface Board used in the original laboratory setup.

On the PC, a closed-source Windows executable called `pid_velleman` provides the user interface, implements the PID controller, and communicates with the VM167 board, as shown in Figure 1.3.

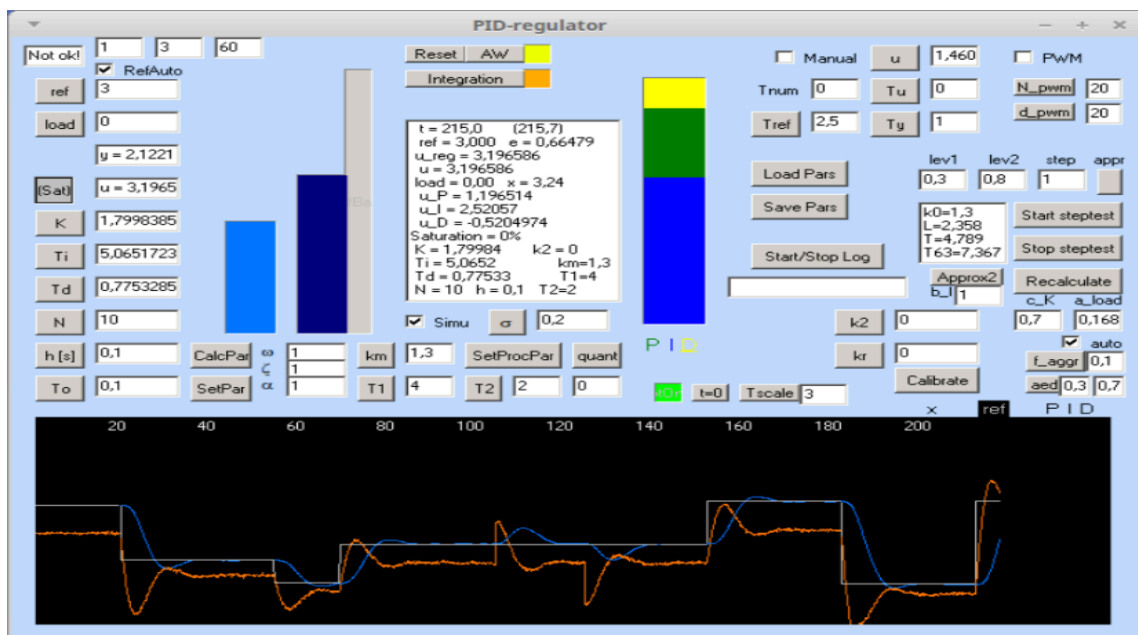


Figure 1.3. The `pid_velleman` application interface running on the legacy Windows XP system.

Students use this software to set PID parameters, observe the tank level in real time, and record the system's response.

The system has several weaknesses. The `pid_velleman` software is a compiled executable with no available source code, meaning it cannot be modified, extended, or documented in detail. It was originally developed for Windows XP and, while the VM167 driver is still available, the overall setup is fragile and dependent on a legacy PC running an outdated operating system. Most importantly, students cannot perform the laboratory exercises remotely, without physical access to the lab, they cannot complete the experiment.

The proposed solution replaces the existing hardware interface with a modern, open, and programmable alternative communicating with a new PC application. The new software should replicate all original functionality PID control with adjustable parameters, real-time visualisation of sensor data, and manual control mode while adding a simulation mode in which the physical hardware is replaced by a mathematical model of the tank system, allowing students to complete the same experiments remotely from anywhere.

What is new in this thesis is the combination of an open, fully documented replacement for the proprietary system with a software simulation mode that, for the first time, allows the laboratory exercises to be completed remotely. Additionally, the lab PCs can now run on Windows 11 instead of the old Windows XP, which significantly improves the students' interaction with the system.

1.2 Purpose

The purpose of this thesis is to modernise a control engineering laboratory system at LTH Helsingborg by replacing its proprietary, closed-source hardware and software with an open, documented, and accessible solution. The expected outcome is a system that preserves the full educational value of the original laboratory exercise while enabling remote experimentation, making the lab more flexible, future-proof, and available to all students regardless of their physical location.

1.3 Goals

The goals of this thesis are:

1. Build a DAQ-bridge firmware (10 Hz telemetry, safety).
2. Implement a complete PID controller with the same parameters and features as the original application (gain K , integral time T_i , derivative time T_d , anti-windup, manual mode). PC application replicating pid_velleman PID features.
3. Support two operating modes a hardware mode using the real physical system and a simulation mode using a mathematical software, modes are switchable from the user interface without code changes.
4. Detect whether a compatible microcontroller is connected via serial communication and indicate this status clearly in the user interface.
5. Base the simulation on a mathematical model of the two-tank system and validate that it reproduces the qualitative dynamic behaviour observed in the physical laboratory system.
6. Validate the new system by reproducing the laboratory exercises from the original lab manual and comparing the results to expected behaviour.

1.4 Research Questions

The following questions will be answered through this thesis:

1. What signal conditioning is required between the existing water tank interface board and a Raspberry Pi Pico microcontroller ?
2. How should the communication protocol between the microcontroller and the PC application be designed to reliably support real-time closed-loop control at 10 Hz?
3. How can the PID algorithm and user interface of the original pid_velleman application be faithfully reimplemented in a new open-source application?
4. What mathematical model best describes the two coupled tank system, and how can its parameters be identified experimentally?
5. How well does the software simulation reproduce the behaviour of the real hardware, and what are its limitations?
6. Can the new system successfully replicate all laboratory exercises described in the original lab manual?

1.5 Motivation

This thesis was chosen because it addresses a concrete, practical problem in an active teaching laboratory, while covering several core areas of the Computer Science and Engineering programme: embedded systems, signal processing, control theory, software development, and system integration. The work produces a tangible deliverable of a working laboratory system rather than a purely theoretical study. The thesis covers the full development cycle from hardware design and firmware development to software architecture and experimental validation, providing broad and applied engineering experience.

For the department, the benefit is a modernised laboratory that is no longer dependent on a single legacy PC or proprietary software. The new system will be open source, fully documented, and maintainable by future students or staff. The addition of a simulation mode directly increases the accessibility of the laboratory, allowing students to prepare for or complete experiments from anywhere without needing to be on campus.

For society more broadly, open and accessible remote laboratory systems support students who cannot physically attend campus due to distance, health, or other personal circumstances, contributing to more inclusive engineering education [2],[3]. The documented approach may also serve as a reference for similar modernisation efforts at other institutions [4],[5].

1.6 Delimitations

The following are explicitly outside the scope of this thesis:

- The physical tank hardware (tanks, pipes, pump, valves) will not be modified or replaced.
- The Raspberry Pi Pico is the preferred hardware platform as it can be programmed both in C and Python, and was provided as the preferred microcontroller for this thesis.
- Python is the preferred programming language on the PC side, as it is the language students learn in the programme.
- The simulation mode of the application can optionally be deployed to a public server, but cloud deployment and production hosting configuration are outside the scope of this thesis.
- Advanced control methods beyond PID (e.g. model predictive control, state-space control) are not included.
- The system will be developed and tested for Windows 11 as the primary platform.
- The physical assembly of a signal conditioning circuit between the microcontroller and the existing tank hardware is outside the scope of this thesis and is identified as future work.

1.7 Summary of Contributions

This thesis has presented the design and implementation of a modernised control engineering laboratory platform for the two-tank water level system at LTH Helsingborg [1]. The work replaces a proprietary closed-source Windows executable (`pid_velleman.exe`) and a legacy Velleman VM167 DAQ card with an open, documented, and accessible alternative based on a Raspberry Pi Pico microcontroller [15] and a Python web application.

The primary contributions of the work are:

1. Identification of signal conditioning requirements. The components and circuit topology required to bridge the existing tank interface board to the Raspberry Pi Pico were identified, including a passive voltage divider and unity-gain buffer for each sensor path, and a non-inverting amplifier for the pump control path, all using a TL084CN quad op-amp. The formal circuit design and physical assembly are left as future work.
2. Raspberry Pi Pico firmware [16] written in MicroPython that implements the DAQ bridge function: ADC sampling at 10 Hz with 8-sample averaging, PWM pump control, NDJSON communication over USB CDC, and a three-layer safety

architecture soft-idle ramp, hardware watchdog, and boot-time safe-state initialisation that cuts off the pump independently of the PC application.

3. A PC-side Python web application (Flask [21] and Flask-SocketIO [14]) that replicates all original laboratory features PID control with adjustable $K/T_i/T_d$, integral anti-windup, manual control mode, and real-time sensor visualisation and adds capabilities the original system lacked: a simulation mode, live parameter adjustment, and a browser-based interface accessible from any device.
4. A two-tank simulation model based on Torricelli's law [10] and mass balance equations [11], with actuator dead time, a first-order pump lag, and overflow detection. The model allows students to conduct the full set of laboratory exercises without access to the physical equipment.
5. A defence-in-depth safety architecture with three independent pump cutoff layers and explicit handling of all identified fault modes, designed so that no single component failure can leave the pump running uncontrolled.
6. A two-tier software distribution strategy that addresses the distinct deployment contexts of the lab and student editions: the lab edition is distributed as open Python source code, readable and verifiable by IT staff and instructors, started with a single python start.py command; the student edition is distributed as a Windows standalone executable [23] that requires no Python installation and provides a zero-friction double-click entry point for home use.

Chapter 2 — Theoretical Background

This chapter explores some of the technologies, tools, and concepts that were used to complete this thesis.

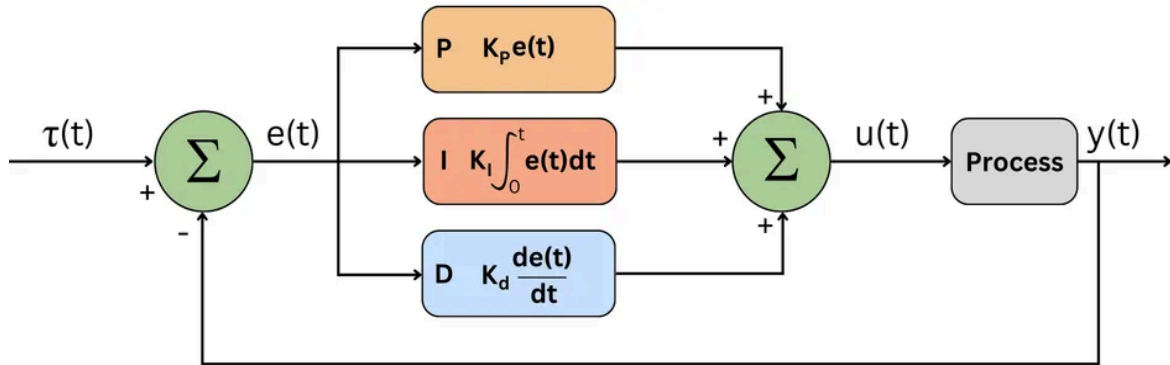


Figure 2.1. Block diagram of PID controller.

2.1 PID Control

The proportional–integral–derivative (PID) controller, is one of the most common control algorithms in industrial practice. Most feedback loops rely on this algorithm or close variations of it, and it appears across a wide range of implementations from simple stand-alone controllers to large distributed process control systems [6]. Its prominence comes from a three-parameter structure that maps directly onto observable closed loop behaviour and performs reliably across a wide range of processes without requiring a detailed mathematical model of the plant. This section describes the controller structure, its discrete-time implementation, and the practical considerations relevant to this thesis.

2.1.1 The Ideal PID Form

The continuous time PID controller in the ideal parallel form is given by equation (2.1). Figure 2.1 shows the corresponding parallel structure, in which the error $e(t)$ is processed by the three terms and summed to form the control signal $u(t)$.

$$u(t) = K \left[e(t) + \frac{1}{T_i} \int_0^t e(\tau) d\tau + T_d \frac{de(t)}{dt} \right] \quad (2.1)$$

Where $e(t) = r(t) - y(t)$ is the control error, defined as the difference between the setpoint r and the measured process output y . The controller parameters are the proportional gain K , the integral time T_i in seconds, and the derivative time T_d in seconds. The output $u(t)$ is the control signal sent to the actuator in this laboratory, the normalised pump duty cycle [6]. The three terms contribute to the control action in distinct ways.

The proportional term, given by equation (2.2):

$$P = K \cdot e(t) \quad (2.2)$$

produces an output proportional to the current error, but alone it cannot eliminate steady-state offset unless the process itself contains an integrator.

The integral term, given by equation (2.3):

$$I = \frac{K}{T_i} \int_0^t e(\tau) d\tau \quad (2.3)$$

accumulates the error over time and eliminates any steady-state offset by continuously adjusting the control signal until the error reaches zero.

The derivative term, given by equation (2.4):

$$D = K \cdot T_d \cdot \frac{de}{dt} \quad (2.4)$$

acts on the rate of change of the error, providing corrective action before the error has fully developed, which improves transient response and can reduce overshoot.

2.1.2 Derivative on Measurement and Derivative Filtering

A well known limitation of the ideal PID formula is the derivative kick, when the setpoint is changed as a step, the instantaneous derivative of the error is theoretically infinite, which produces a large spike in the control output that most actuators cannot follow. The standard remedy in industrial implementations is to compute the derivative term from the measured process output rather than from the error as given by equation (2.5) [6]:

$$D = -K \cdot T_d \cdot \frac{dy(t)}{dt} \quad (2.5)$$

The negative sign reflects the fact that $\frac{dy}{dt}$ and $\frac{de}{dt}$ differ in sign when the setpoint is constant, since $e = r - y$ and when r is constant, $de/dt = -dy/dt$. With derivative on measurement, a step change in setpoint produces no contribution from the D term, the derivative acts only in response to changes in the process output. This is the convention adopted in the implementation described in Chapter 4.

Differentiating a measured signal amplifies high frequency measurement noise. A practical derivative term therefore includes a first-order low-pass filter with time constant as equation (2.6) [6]:

$$\tau_f = \frac{T_d}{N} \quad (2.6)$$

where N is the derivative filter coefficient. The filter approximates ideal derivative action for low frequency signal components while limiting the gain at high frequencies to N , meaning measurement noise is amplified by at most a factor N . Typical values of N are 8 to 20 [6]. The transfer function of the filtered derivative term is given by equation (2.7):

$$D(s) = -\frac{sK T_d}{1+s(T_d/N)} \quad (2.7)$$

This filtered derivative form is used in the discrete implementation described in Chapter 4.

2.1.3 Integral Anti-Windup

When the control output saturates (for example, when the pump reaches full power $u = 1$ or zero $u = 0$) the integral term continues to accumulate even though the actuator cannot follow the controller command. Over time this causes the integrator to build up a large value, which delays recovery to normal operation once the saturation condition ends and typically produces a large overshoot. This effect is known as integrator windup and is a common consequence of actuator saturation in industrial systems [7].

The standard remedy is to suspend integral action as soon as the control signal enters the saturation region and resume it as soon as the controller reenters the linear region of operation [7]. In the implementation described in Chapter 4, this is realised through conditional integration: the integrator is not updated at a given time step if the controller output is saturated and the current error would push it deeper into saturation. The integrator resumes as soon as the output leaves the saturation region, preserving the integrator state accumulated during normal operation.

The laboratory platform supports toggling anti-windup on and off from the user interface, allowing students to directly compare system behaviour with and without windup protection by observing the integrator value and the recovery response.

2.1.4 Discrete-Time Implementation

Process control software executes in sampled-data discrete time rather than continuous time. The derivative and integral expressions above must be discretised at the sample period T . To implement the PID controller using a computer, the continuous-time system is approximated by a discrete-time system that executes periodically: the sensor input is read, the control output is computed, the output is sent to the actuator, and the controller state is updated [8].

The integral term is discretised using backward Euler integration, as given by equation (2.8):

$$I_k = I_{k-1} + K_i \cdot e_k \cdot T \quad (2.8)$$

The derivative term uses a backward difference applied to the filtered measurement, as given by equation (2.9):

$$D_k = -K \cdot T_d \cdot \frac{y_k^f - y_{k-1}^f}{T} \quad (2.9)$$

where y_k^f is the filtered measurement at time step k and $K_i = \frac{K}{T_i}$ is the integral gain. The filtered measurement is updated at each time step according to equation (2.10) [9]:

$$y_k^f = y_{k-1}^f + \alpha \cdot (y_k - y_{k-1}^f), \quad \alpha = \frac{T}{T + \tau_f} \quad (2.10)$$

The three terms are summed to give the unclamped output, which is then clamped to the range $[u_{min}, u_{max}] = [0.0, 1.0]$. The proportional and integral terms use the filtered measurement rather than the raw sensor reading, which avoids noise amplification in both terms.

For the two-tank system, whose time constants are in the range of ~60 seconds, a sample period of $T = 0.1$ s (10 Hz) gives several hundred samples per time constant, which is more than adequate for accurate discretisation [8].

2.2 Two-Tank Liquid Level System

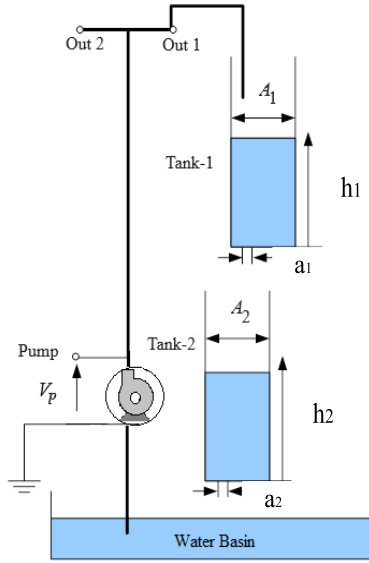


Figure 2.2. Schematic of the two-tank system

2.2.1 Torricelli's Law and Outflow Dynamics

The hydraulic outflow from an open tank through an orifice at the base can be modelled using Torricelli's theorem, which states that the velocity of a fluid jet escaping through an orifice is equal to the velocity of free fall from the surface of the liquid to the orifice as equation (2.11) :

$$v = \sqrt{2gh} \quad (2.11)$$

where $g = 9.81 \text{ m/s}^2$ is the acceleration due to gravity and h is the height of liquid above the orifice. The volumetric flow rate through an orifice of effective cross section a is given by equation (2.12):

$$Q = a \cdot \sqrt{2gh} \quad (2.12)$$

This is the classical form of Torricelli's law [10]. The effective area a accounts for the contraction of the fluid jet (vena contracta effect) and subsumes the discharge coefficient into the area parameter a . The result is a nonlinear function of the liquid height, which means the plant gains change with the operating point.

2.2.2 Mass Balance Equations

The two-tank system used in this laboratory consists of an upper tank (tank 1) that receives inflow from the pump and drains through an orifice into a lower tank (tank 2), which in turn drains through a second orifice back to the reservoir, as shown in Figure 2.2. Applying the conservation of mass to each tank, and modelling the pump as delivering a volumetric flow $k_{pump} \cdot u$ proportional to the normalised control signal $u \in [0, 1]$, the governing ordinary differential equations (ODEs) are given by equations (2.13) and (2.14) :

$$A_1 \frac{dh_1}{dt} = k_{pump} \cdot u - a_1 \sqrt{2gh_1} \quad (2.13)$$

$$A_2 \frac{dh_2}{dt} = a_1 \sqrt{2gh_1} - a_2 \sqrt{2gh_2} \quad (2.14)$$

where h_1 and h_2 are the liquid levels in the upper and lower tanks in metres, A_1 and A_2 are the horizontal cross-sectional areas of the tanks, a_1 is the effective orifice area between the upper and lower tanks, and a_2 is the effective drain orifice area of the lower tank. This model structure follows the same mass balance approach used by Johansson to model interconnected water tank systems for control education [11]. Johansson's process describes four interconnected tanks; the present system is the two-tank cascade obtained by retaining a single pump inflow and a single drain path per tank, without cross-coupling or pump-split terms.

The sensor output is modelled as the normalised level as equation (2.15):

$$y = \frac{h}{h_{max}} \quad (2.15)$$

where h_{max} is the physical height of the tank rim. This normalisation keeps all variables in the range $[0, 1]$ and ensures the PID setpoint and sensor output share a common dimensionless scale.

2.2.3 System Characteristics

The two-tank system has several properties that make it pedagogically valuable. Nonlinearity. The outflow is proportional to $\sqrt{h}$ rather than to h , so the effective plant gain decreases as the tank level rises. A PID controller tuned at a low setpoint will exhibit different performance at a higher setpoint, which motivates students to explore the operating-point dependence of linear controller tuning.

Two-tank cascade. The lower tank receives its inflow from the upper tank rather than directly from the pump. This cascaded arrangement means that pump commands affect the lower tank level only after passing through the dynamics of the upper tank. The result is a plant with two dominant time constants and a phase response that falls more steeply with frequency than a single-tank system. Under proportional-only control, the lower tank level will oscillate and overshoot for gains that would stabilize a single tank, the integral term is essential to eliminate steady-state offset in the lower tank, and the derivative term helps manage the increased phase lag.

Inherent open-loop stability. With no pump input, both tanks drain to their minimum level, so the plant is open-loop stable. This property makes open-loop and manual mode experiments safe: the tanks will not overflow unless the pump is actively driven at high output. Actuator dynamics. The pump introduces additional dynamics not captured by the basic two-tank ODE. Motor inertia and pipe compliance mean that the flow through the pump does not respond instantaneously to a change in commanded duty. These effects are modelled in the simulation as a first-order lag with time constant $\tau = 0.5$ s combined with a pure transport delay of 0.4 s, both of which increase the phase lag in the closed loop and are classical sources of degraded stability margin under aggressive gain tuning.

2.3 Real-Time Web Communication

2.3.1 WebSocket and Socket.IO

HTTP, the underlying transport of the web, follows a request response pattern: the client initiates every exchange, and the server can only send data in reply. For a control laboratory that must push sensor readings to the browser ten times per second without the browser repeatedly polling, the request response model is unsuitable because it introduces latency proportional to the polling interval and generates unnecessary overhead on the server.

The WebSocket protocol [12] solves this problem by defining a persistent, full duplex communication channel between a browser and a server over a single TCP connection. After an initial HTTP handshake that upgrades the connection to the WebSocket protocol, either party can send frames to the other at any time without waiting to be asked. The RFC 6455 specification [12] standardises the framing format, control frames (ping, pong, close), and the handshake procedure. The result is a channel with latency on the order of a round-trip time on the local network well below the 100 ms control period.

Socket.IO [13] is a higher level library built on top of WebSocket that adds features commonly needed in real time applications: automatic reconnection after a dropped connection, namespace and room multiplexing, acknowledgement callbacks, and an event-based API. Socket.IO uses WebSocket as its primary transport and falls back to HTTP long-polling if WebSocket is not available. The server-side implementation used in this project is Flask-SocketIO [14], which integrates the Socket.IO server with the Flask web framework and runs it within the Flask application process.

The control loop emits one `data` event per 100 ms tick over the Socket.IO channel to all connected browsers. Each event carries the current timestamp, both tank levels, the pump output, the setpoint, and the three PID component values. The browser accumulates these events into a scrolling Chart.js plot without ever polling the server, achieving true push-based real-time visualisation.

2.3.2 Embedded Microcontrollers as DAQ Devices

A data acquisition (DAQ) device bridges the analogue physical world sensor voltages and actuator signals and the digital computing environment. Traditional DAQ cards such as the Velleman VM167 connect via an interface cable and are driven by proprietary drivers that expose a vendor specific API on the PC side.

The Raspberry Pi Pico W [15], shown in Figure 2.1, is a microcontroller board built around the RP2040 dual-core processor. It includes a 12-bit analogue-to-digital converter with three user-accessible input channels, 16 PWM channels, and native USB support. When running MicroPython [16], the Pico appears to the PC as a USB Communications Device Class (CDC) device, a virtual serial port with no proprietary driver required. This makes the Pico a capable, low-cost replacement for proprietary DAQ cards in low-speed laboratory applications. It reads sensor voltages, drives PWM outputs, and communicates with the PC over a standard USB serial interface that any operating system can access natively [15].

The system described in this thesis positions the Pico as a pure DAQ bridge: it samples the ADC channels, normalises the readings against stored calibration constants, drives the PWM output to the pump, and relays data and commands over USB serial. All PID control logic, timing, and mode management execute on the PC, which has substantially more processing resources and is easier to program, debug, and modify than a microcontroller.

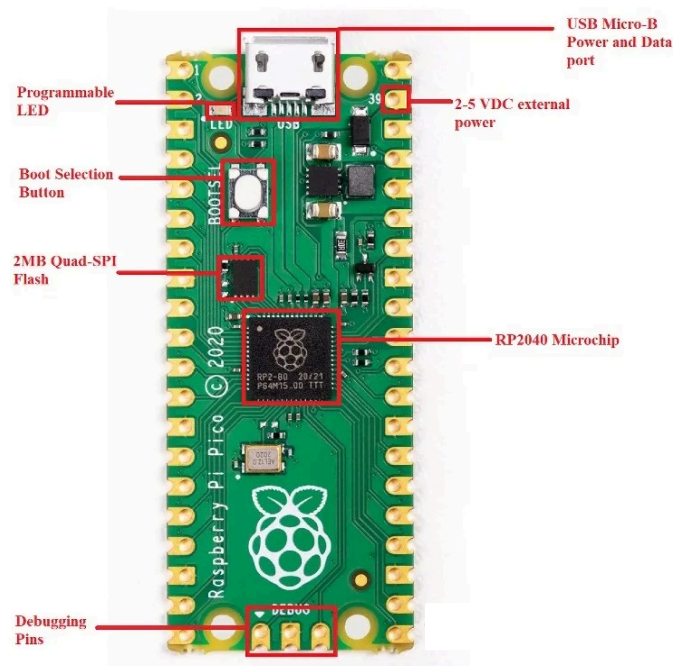


Figure 2.3. The Raspberry Pi Pico W microcontroller board used as the DAQ bridge in this thesis.

Chapter 3 — Method

This chapter presents the methodology and different phases of the thesis. It describes the process used to test different technologies, and outlines the criteria used to evaluate different sources.

3.1 Development Approach

The system was developed using an iterative, component-by-component approach in which each major subsystem was designed, implemented, and tested independently before being integrated with the rest of the platform. This incremental development approach follows established software-engineering practice [24, ch. 2]. It was chosen because the platform consists of several loosely coupled subsystems the Pico firmware, the serial communication driver, the PID controller module, the tank simulation model, the Flask backend, and the browser interface each of which has a well-defined interface and can be verified in isolation.

The work was organised into three phases, with the output of each phase feeding the next:

1. **Subsystem development:** each subsystem was developed and tested in isolation.
2. **Integration:** the verified subsystems were combined into the complete platform.
3. **Validation:** the full system was validated end-to-end.

Phase 1) Subsystem development. Each subsystem was developed and tested in isolation.

Signal conditioning requirements. The components and circuit topology were identified from the interface board documentation. The formal circuit design and physical assembly are left as future work.

Pico firmware. The firmware was written in MicroPython and tested against the expected serial protocol using a PC terminal before integration with the Flask backend. The key design criterion was that it must enforce pump safety independently of the PC application.

Flask backend and PID controller. The backend was developed incrementally; the simulation mode, which requires no hardware, was implemented and tested first, followed by the hardware communication path. The PID controller module was developed and unit-tested independently of the control loop.

Browser interface. The frontend was developed and tested against a running simulation backend, which allowed the interface logic to be verified without hardware present.

Phase 2) Integration. The verified subsystems were combined into the complete platform. The simulation path was assembled and tested first, followed by the hardware communication path.

Phase 3) Validation. The full system was validated end-to-end in simulation mode, and the hardware path Pico detection, telemetry, pump-command transmission, and the safety responses was verified directly (Chapter 5).

3.2 Literature Study

The literature study was carried out in two phases. In the first phase, foundational textbooks on PID control and feedback systems were reviewed to establish the theoretical basis for the implementation [6]-[8]. These sources were selected based on their standing in the control engineering literature and their direct relevance to the algorithm implemented in this project.

In the second phase, peer-reviewed articles were identified through searches in the IEEE Xplore and Google Scholar databases to establish the context of remote and virtual laboratories in engineering education. Search terms included combinations of "remote laboratory", "virtual laboratory", "engineering education", "PID control", "water tank", "open-source hardware", and "Raspberry Pi". Articles were selected if they described systems similar in function to the one developed here web-accessible control laboratory platforms or if they provided empirical evidence about the effectiveness of virtual laboratory learning. The review focused on work published since 2005, with the exception of the foundational Åström and Östberg 1986 paper [17] on the original LTH teaching laboratory, which is directly referenced as the historical starting point for the physical equipment used in this project.

Technical documentation for the specific hardware and software components used in the implementation the Raspberry Pi Pico W datasheet [15], the WebSocket protocol specification [12], and the Flask-SocketIO library documentation [14] was consulted during implementation and referenced where claims are made about the specific behaviour of these components. Security considerations were informed by the OWASP mobile application risk framework [18].

3.3 Testing Strategy

The testing strategy followed from the architecture of the system, each component was tested at the unit level before integration, and each component was tested at the unit level before integration, and the integrated system was validated end-to-end in simulation mode, while in hardware mode the communication, telemetry, and safety paths were verified directly (Section 5.2). This unit-then-integration strategy follows standard software testing practice [24, ch. 8].

Unit testing

The PID controller module (`pid_controller.py`) and the tank simulation model (`tank_model.py`) were tested. The test suite verifies the following properties:

- The closed-loop system converges to the setpoint within the expected time under well-tuned gains.
- Live PID re-tuning does not reset the integrator state (integral preservation test).
- The tank model remains bounded and non-negative under zero pump input (stability at rest).
- The tank model clamps out-of-range pump commands without producing numerical errors (input clamping test).
- The simulation overshoots the setpoint under high-gain proportional-only control (overshoot realism test).
- The PID output is always within the actuator limits $[0, 1]$ regardless of input (output limits test).

Integration and validation testing

Simulation validation. The simulation was validated by observing the real-time software charts to ensure the simulated system responded correctly and realistically to the PID controller. The model's behavior was further checked for internal consistency ensuring non-negative levels, steady-state equilibrium, correct directional response, and by comparing the simulated step response against published descriptions of similar two-tank systems [17] to assess whether the model produces physically plausible behaviour. The quantitative RMSE comparison against measured hardware step responses specified in Section 1.3 is reported in Section 5.3.

Hardware integration testing. End-to-end hardware testing was conducted by connecting the Pico to a PC running Windows 11, switching the application to hardware mode, and verifying that sensor readings appeared in the browser interface and that pump commands were transmitted correctly to the Pico. Full closed-loop step tests on the real hardware were not completed during the project period; the limitations of the hardware testing conducted are stated explicitly in Section 5.2.

Deployment testing. The lab edition was installed and run on the labs Windows 11 PC to verify that both hardware mode and simulation mode start correctly. The student edition was installed in a separate clean Python environment to verify that simulation mode can be started without any hardware present.

3.4 Source Evaluation

The sources used in this thesis span four categories, each with different characteristics regarding reliability and relevance.

Internal course documentation. The original laboratory manual [1] is a teaching document produced by the Division of Industrial Electrical Engineering and Automation at LTH Helsingborg, the same division that owns and operates the physical equipment. As the authoritative description of the original laboratory exercises and of the `pid_velleman` application's intended behaviour, it is reliable for the purpose for which it is cited: establishing the functional requirements that the new system must replicate.

Peer-reviewed textbooks. The primary theoretical foundation draws on two textbooks by Åström and Hägglund [6] and one by Åström and Murray [8], along with Johnson and Moradi [7]. Åström is a widely acknowledged authority in the field of control engineering, with these works regarded as standard references for PID control theory by the research community. The content of these books is stable and has been extensively cited in subsequent literature over several decades, their reliability for the theoretical claims made in Chapter 2 is high. The discrete-time filter implementation additionally draws on Proakis and Manolakis [9], a standard digital signal processing textbook, used to support the first-order recursive low-pass filter form in Section 2.1.4. The hydraulic modelling of the tank system draws on White [10], a standard fluid mechanics textbook, used to support Torricelli's law and the discharge-coefficient form of the orifice outflow equation in Section 2.2.1. These textbooks

are likewise long-established standard references, and their reliability for the theoretical claims they support is high.

Peer-reviewed journal articles and conference papers. The articles on remote and virtual laboratories [3]-[5][19][20] were published in established IEEE, Elsevier, and PLOS journals and conferences with standard peer-review processes. The meta-analysis by Ma and Wang [3] is based on 45 independent studies and follows established systematic review methodology, making its conclusions particularly robust. The Álvarez Ariza and Nomesqui Galvis article [4] was published in HardwareX, an open-access journal specifically focused on research hardware. The Åström and Östberg 1986 paper [17] and the Johansson 2000 paper [11] are historical primary sources that describe the specific laboratory equipment used in this thesis, their reliability for this purpose is well established.

The Takkouch et al. 2023 paper [2] is published in the Journal of Perspectives in Applied Academic Practice and is used to support the accessibility claims in Section 6.6.

Technical standards and specifications. The WebSocket protocol RFC 6455 [12] is an IETF standards document, the definitive specification for the protocol implemented in this thesis . Such standards documents are authoritative by definition for claims about protocol behaviour.

Software documentation. The Flask-SocketIO documentation [14], the Socket.IO documentation [13], the MicroPython documentation [16], the Flask documentation [21], the Chart.js documentation [22], the PyInstaller manual [23], and the Raspberry Pi Pico W datasheet [15] are primary technical documentation produced and maintained by the respective developers and manufacturers. These sources are considered reliable for claims about the behaviour of the specific software and hardware components used.

Web resources. The OWASP foundation publishes its security risk frameworks as publicly accessible web documents [18]. OWASP is a widely referenced non-profit organisation in the information security field; its publications are used here specifically for characterising the security risk category relevant to the discussion in Section 4.9. No Wikipedia articles have been cited.

All sources have been selected for their direct relevance to the claims they are used to support. Where multiple sources could have been cited for a given claim, preference was given to primary sources over secondary ones, and to peer-reviewed publications over informal web content.

Chapter 4 — System Design and Implementation

This chapter outlines the design decisions and implementation hurdles that shaped the final system architecture. It details the technical choices made throughout the thesis and explains how various deployment constraints were systematically managed.

4.1 Architecture Overview

The platform replaces the original `pid_velleman.exe` and VM167 DAQ card with a two-tier architecture, a Python backend [21] running on the laboratory PC and a browser-based frontend that any student device can load. The two tiers communicate over WebSocket via Socket.IO [13], which delivers sensor data to the browser at 10 Hz without polling. The system supports two operating modes, selectable from the browser interface without restarting the application:

Hardware mode. The Raspberry Pi Pico, connected to the laboratory PC via USB, reads the tank level sensors and drives the pump. The PC application reads sensor data from the Pico over a USB serial connection and sends pump commands back at 10 Hz.

Simulation mode. The tank model runs entirely in software on the PC. No hardware is required. This mode is the default and is the mode available in the student edition.

The system is also distributed in two editions:

Lab edition. Installed on each of the laboratory workstations. Contains both the hardware mode and the simulation mode. Each workstation runs an independent Flask instance; there is no shared server.

Student edition. A reduced distribution that contains only the simulation mode. It can be installed on any PC with Python 3.10 or later and used from home without any hardware.

The data flow in each mode is illustrated in Figure 4.1 (hardware mode) and Figure 4.2 (simulation mode).

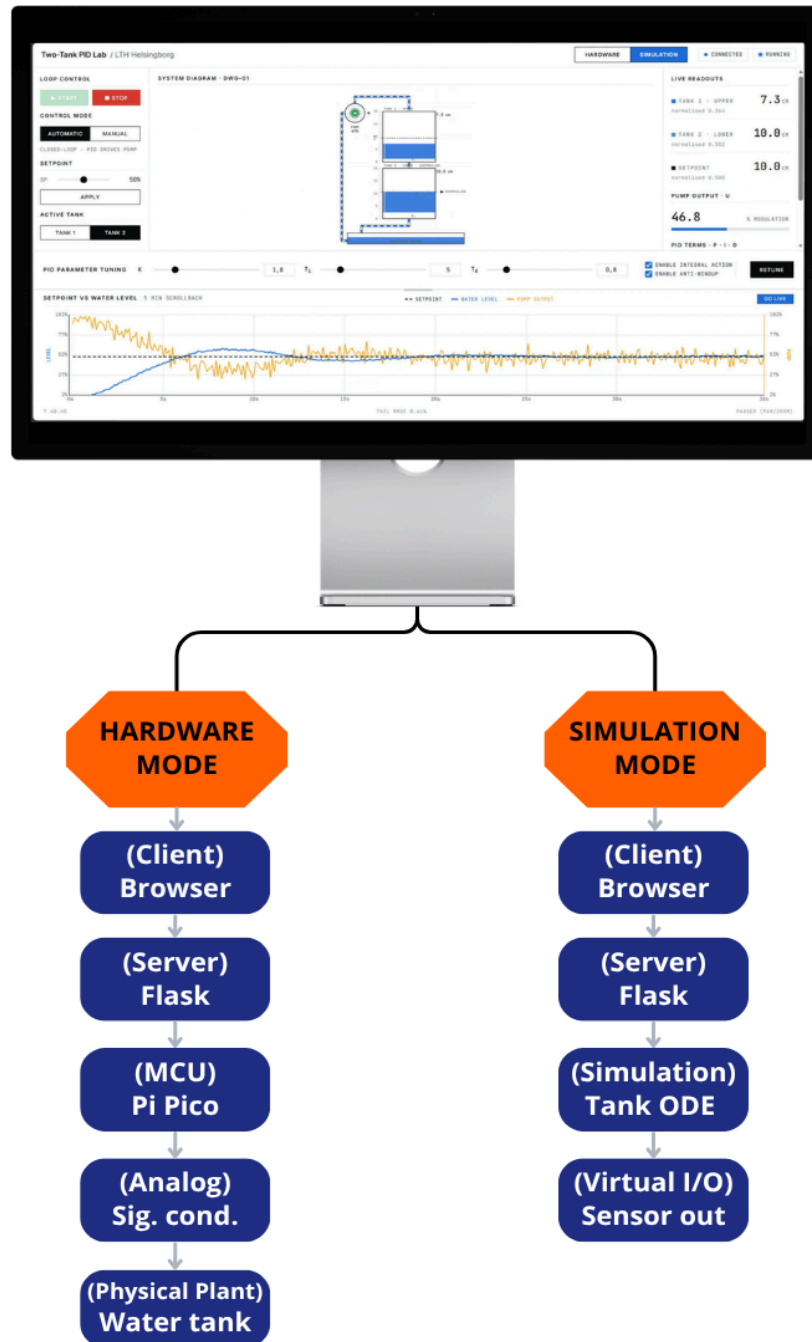


Figure 4.1. High-level data flow in hardware mode and in simulation mode.

4.2 Raspberry Pi Pico Firmware

The Pico firmware is written in MicroPython [16] and implements the DAQ bridge role introduced in Section 2.3.2. Before describing the implementation in detail, a brief overview of the key concepts used in the firmware is provided.

read_u16() is a MicroPython function that reads the raw value from an ADC channel and returns it as a 16-bit integer in the range 0–65535, where 0 corresponds to 0 V and 65535 corresponds to the ADC reference voltage of 3.3 V.

asyncio is a Python concurrency framework that allows multiple tasks to run cooperatively within a single thread. Instead of running in parallel, each task voluntarily yields control to the others at defined points such as when waiting for input or sleeping. This allows the firmware to read sensors, receive commands, and monitor safety conditions simultaneously without requiring a multi-threaded operating system.

Coroutines are the individual tasks that run within the asyncio event loop. The firmware uses three coroutines `tx_telemetry`, `rx_commands`, and `safety_idle` each responsible for one aspect of the firmware's behaviour.

NDJSON (Newline-Delimited JSON) is the message format used for communication between the Pico and the PC. Each message is a single line of JSON text terminated by a newline character, making it easy to parse and human-readable for debugging.

The firmware is responsible for reading the ADC channels, driving the PWM output, communicating with the PC over USB serial, and enforcing hardware-side pump safety independently of the PC. The communication protocol and firmware structure are described in detail in Section 4.3.

4.2.1 Boot Sequence and Safe State

The boot sequence follows a strict safety-first order. The first action at import time is to drive GPIO pin GP2 to logic-low through a plain digital output before any PWM peripheral is configured. This ensures that even if subsequent initialisation steps fail, the pump driver remains at zero.

The boot order is:

1. Set GP2 to output low (hardware safe state).
2. Initialise PWM on GP2 with duty cycle = 0.
3. Initialise the hardware watchdog timer (5000 ms timeout).
4. Initialise the two ADC channels (ADC0 on GP26, ADC1 on GP27).
5. Emit the “hello” message over USB serial.
6. Enter the asyncio event loop.

If any step in 2–4 raises an exception, the chip remains in step 1 state: the pump pin is low and the PWM is not active, so no pump activation can occur.

4.2.2 ADC Sampling and Normalisation

Each sensor input is sampled as 8 consecutive `read_u16()` readings, which are averaged to reduce quantisation noise. The averaged 16-bit value is then converted to a normalised level using two calibration constants stored in the firmware, as given by equation (4.1):

$$y_{norm} = \frac{raw - raw_{empty}}{raw_{full} - raw_{empty}}, \quad \text{clamped to } [0, 1] \quad (4.1)$$

where `raw_empty` and `raw_full` correspond to the ADC readings at zero and full tank level respectively. For the current configuration, $V_{empty} = 0.0$ V and $V_{full} = 2.0$ V, mapped from the 3.3 V reference of the Pico's ADC, as given by equation (4.2):

$$raw_{full} = \frac{V_{full}}{V_{ref}} \cdot 65535 = \frac{2.0}{3.3} \cdot 65535 \approx 39718 \quad (4.2)$$

A 0–2 V mapping was chosen to leave headroom below the 3.3 V ADC maximum and to protect the input, at the cost of leaving roughly 40% of the ADC range unused. The calibration constants `raw_empty` and `raw_full` are configurable, so scaling the sensor signal to the full ADC range which recovers this resolution is a change to the firmware constants alone, and is part of the signal-conditioning design discussed in Section 6.7.

Performing the normalisation on the Pico means the PC receives dimensionless values in $[0, 1]$ regardless of the sensor voltage range. If the physical sensors are recalibrated or replaced, only the firmware constants need to change not the PC software.

4.2.3 Asyncio Coroutines

The firmware runs three concurrent coroutines within a MicroPython [16] asyncio event loop:

`tx_telemetry`. Executes every 100 ms (10 Hz). Reads both ADC channels, increments a sequence counter, and emits one NDJSON sample frame over USB serial.

`rx_commands`. Uses `uselect.poll()` to check for available input without blocking. Characters are accumulated into a line buffer until a newline character arrives, at which point the line is parsed as JSON and dispatched to the command handler. A runaway buffer is capped at 512 characters to prevent memory exhaustion.

`safety_idle`. Checks every 100 ms whether a valid command has been received within the last 1000 ms. If the PC has been silent for 1 second and the pump is still running, this coroutine begins a linear ramp of the PWM duty cycle from its current value to zero over 500 ms, emitting a `soft_idle` error event. This ramp completes regardless of the PC's state and runs independently of the hardware watchdog.

4.2.4 Hardware Watchdog

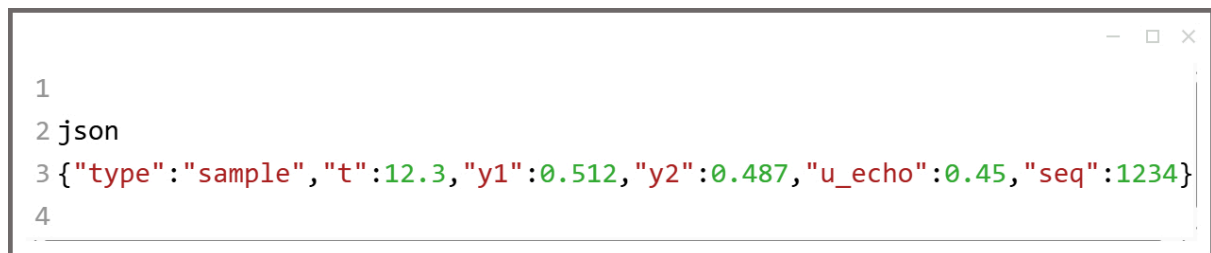
A hardware watchdog timer (`machine.WDT`) is initialised at boot with a 5000 ms timeout. The watchdog is fed only when a valid command, `set_pwm`, `ping`, `stop`, or `set_rate`, is successfully parsed. Malformed input does not feed the watchdog. If the PC application stops sending commands for any reason (crash, cable disconnect, mode switch that closes the serial port), the watchdog will expire after 5 seconds and reset the Pico [15]. The boot sequence then sets PWM duty to zero before any other action. .

4.3 Communication Protocol

The communication protocol between the Pico and the PC application uses newline-delimited JSON (NDJSON) over a USB CDC virtual serial port at 115200 baud. NDJSON was chosen because it is human-readable for debugging, requires no framing overhead beyond the newline delimiter, and is trivially parsable in both MicroPython [16] and Python. USB CDC provides a reliable byte stream; a malformed line is simply discarded rather than corrupting subsequent frames.

4.3.1 Pico → PC: Telemetry Frames

The Pico emits one sample frame per 100 ms as shown in *Figure 4.2* :



```
1
2 json
3 {"type":"sample","t":12.3,"y1":0.512,"y2":0.487,"u_echo":0.45,"seq":1234}
4
```

Figure 4.2: Example telemetry sample frame.

The `t` field is seconds since boot (diagnostic only; the PC uses its own monotonic clock for PID timing). The `y1` and `y2` fields are normalised tank levels in $[0, 1]$. The `u_echo` field is the PWM duty currently being applied by the Pico, which allows the PC to detect if the watchdog or soft-idle ramp has zeroed the pump independently of a PC command. The `seq` field is a monotonically increasing counter that allows the PC to detect dropped lines.

At boot, the Pico emits a hello frame before any sample frames:



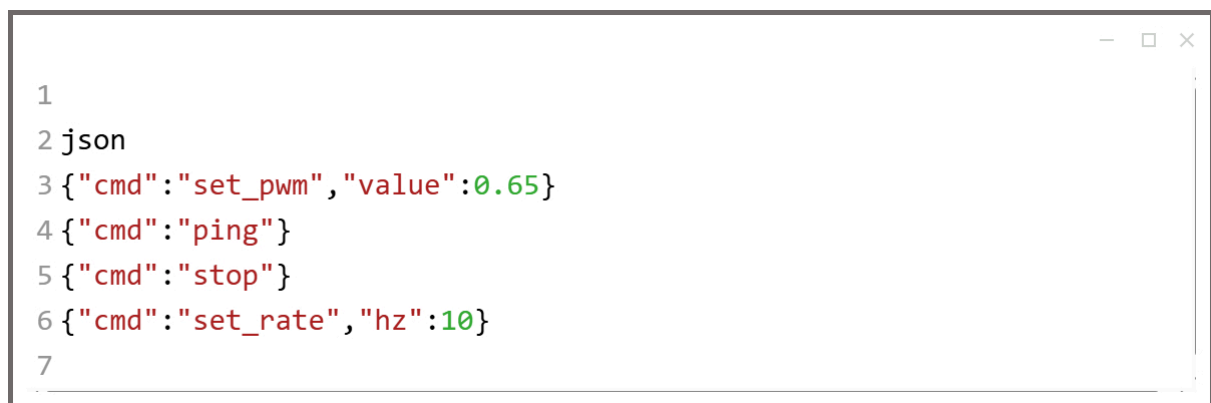
```
1
2 json
3 {"type":"hello","fw":"watertank-pico-0.1","seq":0}
4
```

Figure 4.3. Example startup hello frame.

The PC driver verifies the receipt of this frame during connection establishment.

4.3.2 PC → Pico: Command Frames

The PC sends command frames to the Pico:



```
1
2 json
3 {"cmd":"set_pwm","value":0.65}
4 {"cmd":"ping"}
5 {"cmd":"stop"}
6 {"cmd":"set_rate","hz":10}
7
```

Figure 4.4. Example command frames sent from the PC to the Raspberry Pi Pico.

The `set_pwm` command is sent every 100 ms tick. It carries the pump duty in $[0, 1]$ and feeds the Pico watchdog. The `ping` command is sent by a background heartbeat thread at 2 Hz whenever the control loop is paused but the serial port is open, keeping the watchdog alive during idle periods. The `stop` command zeros the pump immediately and is sent before a controlled disconnect. The `set_rate` command changes the telemetry frequency; it is reserved for future use.

4.3.3 PC Serial Driver

The PC-side serial driver (`pico_serial.py`) manages the USB serial connection to the Pico. On connection, the driver auto-detects the Pico's USB CDC port by matching the USB Vendor ID 0x2E8A (assigned to Raspberry Pi Ltd [15]) using `pyserial`'s port enumeration. This makes the system robust to COM port reassignment across sessions without manual configuration. A background reader thread continuously calls `readline()` on the serial port, parses each JSON line, and writes the resulting sample into a single-slot mutex-protected cache. The control loop calls `read_latest()` to retrieve the most recent sample and its age in milliseconds. If the age exceeds the stale-link threshold (250 ms = 2.5 control ticks), the control loop declares a link fault and stops. A second background heartbeat thread pings the Pico at 2 Hz to keep the watchdog fed when the control loop is paused.

4.4 PID Controller Module

The PID controller is implemented as the `PIDController` class in `pid_controller.py`. It uses the ideal parallel form with derivative-on-measurement and a first-order derivative filter, as described in Section 2.1. The class is instantiated once at application startup and retains its internal state (integrator accumulator, filter state) across ticks.

Parameters are exposed using the $K/T_i/T_d$ parameterisation from the lab manual [1] proportional gain, integral time in seconds, and derivative time in seconds. The internal computation converts these to gains before each compute call: $K_i = K/T_i$ and $K_d = K \cdot T_d$. The default values are $K = 1.8$, $T_i = 5.0$ s, and $T_d = 0.8$ s, which serve as initial values intended as a starting point for the laboratory exercise.

The derivative filter coefficient is $N = 10$, a value within the typical industrial range of 8 to 20 (Section 2.1.2), giving a filter time constant $\tau_f = T_d/N = 0.08$ s at the default derivative time. The filtered measurement update and derivative equations are as described in Section 2.1.4.

Two features can be toggled from the browser interface without restarting the control loop, as shown in Figure 4.5, component 8. When integral action is disabled, the integrator is cleared and K_i is set to zero, allowing students to observe the steady-state error that results from proportional-only or PD control. When anti-windup is disabled, the integrator continues to accumulate during output saturation, allowing students to observe integrator windup and the

resulting overshoot on recovery. When enabled, the conditional integration scheme described in Section 2.1.3 is applied.

PID parameters can be updated live via the `/api/pid` route while the control loop is running. A live retune does not reset the integrator state, which preserves the accumulated integral action and avoids a transient in the control output.

4.5 Tank Simulation Model

The tank simulation model is implemented in `tank_model.py` using the mass balance equations derived in Section 2.2.2. The model is integrated using the Euler method at a fixed step size of 0.1 s, matching the hardware sample rate so that the simulation and hardware modes share the same discretisation. The model parameters were chosen by engineering judgment to produce pedagogically representative behaviour rather than fitted from experimental measurements; the implications of this are discussed in Section 5.3.

4.5.1 Model Parameters

The model parameters are engineering estimates derived from the physical dimensions of the tank system. They have not been identified experimentally from measured step responses. The parameter values are engineering estimates derived from the physical dimensions of the system (see Section 6.3). Table 4.1 lists the parameter values used in the simulation.

Table 4.1. Tank model parameters (engineering estimates).

Parameter	Symbol	Value	Unit	Description
Tank cross-section	$A_1 = A_2$	0.008	m ²	Horizontal area of each tank
Upper orifice area	a_1	3.0×10^{-4}	m ²	Effective area between tank 1 and tank 2
Lower orifice area	a_2	2.6×10^{-4}	m ²	Effective drain area of tank 2
Pump gain	k_{pump}	7.5×10^{-4}	m ³ /s	Volumetric flow per unit control signal
Maximum level	h_{max}	0.20	m	Physical rim height
Pump time constant	τ_{pump}	0.5	s	First-order actuator lag
Transport delay	τ_d	0.4	s	Pure dead time from pump to tank 1

The parameter values were selected so that the closed-loop time constants at the default PID gains are in the range of 5–10 seconds consistent with typical behaviour of small educational water tank systems and so that high-gain proportional-only control produces visible overshoot and oscillation.

4.5.2 Actuator Dynamics

Two actuator dynamics are modelled beyond the basic ODE:

Transport delay (dead time). A FIFO buffer of length 4 samples represents the pure time delay from pump command to the arrival of flow at tank 1. At each step, the oldest sample from the buffer is used as the effective pump command, and the new command is appended. This dead time is a classical source of phase lag in closed loops; under aggressive proportional tuning, it is sufficient to cause sustained oscillation.

First-order pump lag. A first-order filter with time constant τ_{pump} smooths the delayed command before it is used in the ODE, as given by equation (4.3):

$$u_{\text{eff}}(k) = u_{\text{eff}}(k - 1) + \frac{T}{T + \tau_{\text{pump}}} \cdot (u_{\text{delayed}}(k) - u_{\text{eff}}(k - 1)) \quad (4.3)$$

This models the inertia of the pump motor and the compliance of the connecting pipes.

4.5.3 Overflow and Noise

When the Euler step would push a tank level above h_{max} , the level is clamped at h_{max} and an overflow flag is set. The browser interface displays a spilling animation when the overflow flag is active, which gives students direct visual feedback if their setpoint or gains cause the tank to overflow.

Gaussian measurement noise with standard deviation $\sigma = 0.006$ (normalised) is added to the output of each tank, simulating the sensor noise present in real hardware.. This represents about 0.6 % of full scale, a representative estimate for this class of level sensing, chosen to be visible on the telemetry plots without obscuring the step response. It can be disabled for reproducible comparisons.

4.6 Flask Backend and Control Loop

This section describes the tank simulation model, its parameters, the actuator dynamics, and the handling of overflow.

4.6.1 Application Structure

The Flask application [21] (app.py) is the orchestration layer. It owns the control loop thread, the API routes, and the Socket.IO [13] event emission. Each module has a single well-defined responsibility, which allows the PID controller and tank model to be unit-tested independently of the Flask application. The application comprises five Python modules, listed in Table 4.2.

Table 4.2. Python modules comprising the Flask application.

Module	Responsibility
app.py	Flask app, Socket.IO server, API routes, control loop
pid_controller.py	PID algorithm (K/Ti/Td form, anti-windup, derivative filter)
tank_model.py	Two-tank ODE model (Torricelli, Euler integration)
pico_serial.py	USB serial driver, auto-detect, reader thread, heartbeat
state.py	Thread-safe shared state between the control loop and HTTP routes

The distribution also includes a Windows launcher script (start.py) that automatically creates a virtual environment, installs dependencies, starts the Flask server on port 5055, and opens the browser. This script is not part of the control system and is not discussed further here.

4.6.2 Control Loop Design

The control loop runs in a background thread started by Flask-SocketIO [14] using `socketio.start_background_task()`. It executes at a target rate of 10 Hz using a fixed-phase scheduling scheme: the target time for the next tick is computed as `next_tick += 0.1`, and the loop sleeps for the remaining time. This ensures the loop runs at a stable 10 Hz regardless of minor variations in computation time. If a tick overruns, the phase reference is reset to the

current time rather than accumulating the overrun into subsequent ticks, preventing the loop from attempting to catch up on missed ticks and bounding timing jitter to at most one period.

Each tick executes the following sequence:

1. **Acquire sensor reading.** In simulation mode, call `tank.step(u_prev)` to advance the ODE by one step. In hardware mode, call `pico.read_latest()` and check the sample age against the 250 ms stale-link threshold.
2. **Select control channel.** Use the level of the active tank (`y1` or `y2`) as the process variable for the PID.
3. **Compute PID output.** In automatic mode, compute `u, P, I, D = pid.compute(y, dt=0.1)`. In manual mode, use the manually commanded pump value and reset the PID integrator.
4. **Send actuator command.** In hardware mode, send the computed pump duty to the Pico. In simulation mode, the output is consumed on the next call to `tank.step()`.
5. **Publish state.** Append the current readings to the history deque (maximum 3000 points = 5 minutes at 10 Hz) and emit a data event over Socket.IO to all connected browsers.

4.6.3 Shared State and Thread Safety

The `SharedState` class in `state.py` holds all mutable runtime state shared between the control loop thread and the HTTP request threads. All multi-field reads and all writes are protected by a single `threading.RLock`. Single boolean reads such as `state.running` in the loop's while-condition do not acquire the lock, relying on CPython's Global Interpreter Lock which ensures that simple attribute reads are atomic at the bytecode level.

HTTP POST routes update the shared state for example changing the setpoint or PID gains and return immediately. The control loop picks up the new values on its next tick. PID parameter changes applied via `/api/pid` take effect without restarting the loop and without resetting the integrator.

4.7 Browser Interface

The browser interface is a single-page application served by Flask [21] from templates/index.html. It uses vanilla JavaScript with no framework dependencies, Socket.IO [13] for the real-time data stream, and Chart.js [22] for the strip chart. The full interface is shown in Figure 4.5.

The layout is divided into three columns. The left column contains the loop start and stop buttons (1), the automatic/manual control mode toggle (2), the setpoint slider (3), the active tank selector (4), the PID parameter inputs with a Retune button (7), and the integral action and anti-windup enable checkboxes (8). The centre column displays an inline SVG schematic of the two-tank system (6) that updates dynamically: the fill level of each tank scales with the current normalised level reading, and an overflow animation is displayed when a tank overflows. The active tank being controlled is highlighted. The right column contains numeric readout boxes for both tank levels, the setpoint, the pump output, and the three P/I/D contribution bars (9), and a scrolling strip chart (5) displaying the normalised level, the setpoint, and the pump output.

Mode selection between hardware and simulation is provided as a segmented control in the header (10), alongside a connection status indicator (11). Switching mode stops the running control loop, sends the mode-change request to the backend, and displays any resulting error such as "Pico not found" as an error notification banner. The mode toggle reverts to simulation if the hardware connection attempt fails.

The strip chart (5) uses Chart.js [22] with animation disabled. The visible window shows the last 60 seconds of data, while the client-side buffer retains a full 5-minute scrollbar, allowing students to review earlier behaviour without reloading the page. Animations are disabled to maintain smooth rendering at 10 Hz.

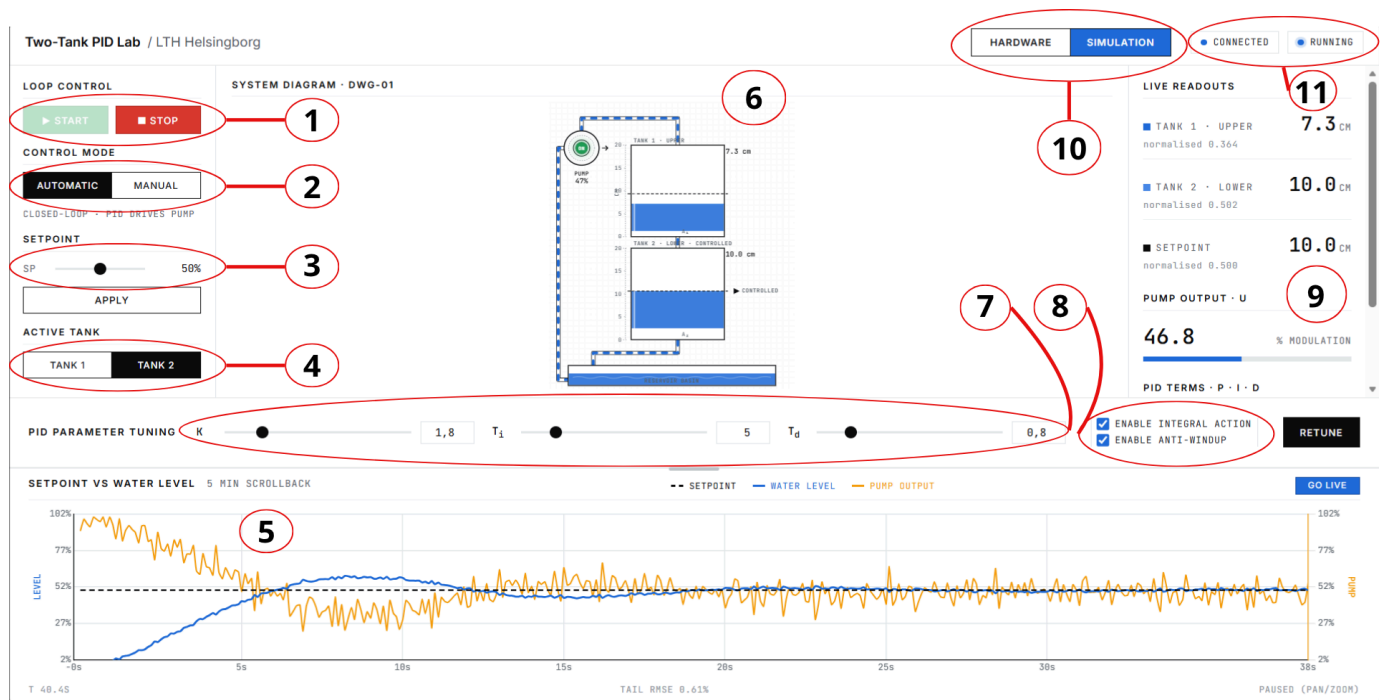


Figure 4.5. The browser interface of the new laboratory system. Numbered components are described in Section 4.7.

4.8 Safety Architecture

The safety architecture is based on the principle of defence-in-depth: three independent mechanisms can each cut off pump power without requiring coordination with the other two.

Table 4.3 summarises the three layers.

Table 4.3. Three-layer pump cutoff safety architecture.

Layer	Location	Trigger condition	Action
1) Stale-link check	PC (control loop)	Hardware telemetry age > 250 ms or absent	Stop control loop, zero pump command, revert mode to simulation, notify browser
2) Soft-idle ramp	Pico firmware	No valid command received for > 1000 ms	Ramp PWM duty linearly to zero over 500 ms
3) Hardware watchdog	Pico hardware WDT [15]	No valid command received for > 5000 ms	Pico reset; boot sequence sets PWM = 0 before any other action

The stale-link check (Layer 1) is implemented in the control loop in `app.py`. When a link fault is detected, the loop stops, attempts a best-effort zero pump command, disconnects the serial port, reverts the operating mode to simulation, and emits a `link_error` event over `Socket.IO` [13] so the browser can display a user-visible notification.

The soft-idle ramp (Layer 2) is the `safety_idle()` coroutine in the Pico firmware [16]. It runs independently of the PC application. If the PC stops sending commands for any reason including an application crash, a cable disconnect, or a deliberate pause of the control loop the Pico begins ramping the pump down after 1 second without requiring any further instruction from the PC.

The hardware watchdog (Layer 3) is independent of the MicroPython runtime. It is a hardware peripheral in the RP2040 processor [15]. Even if the MicroPython interpreter itself hangs or crashes, the watchdog will expire after 5 seconds and reset the processor. The reset execution begins from the top of `main.py`, where the first action is to set the pump GPIO to low. In addition to these three cutoff layers, the PC always clamps the pump duty to `[0,1]`

before sending it to the Pico, and the Pico clamps it again on receipt, providing an additional guard against any software error that could generate an out-of-range value.

4.9 Security Considerations

The replacement system improves on the original closed-source distribution by shipping the entire backend as readable Python source code. Any user can inspect every line before running it. Several further measures reduce the attack surface:

Local-only server binding. In the student edition, the Flask server [21] binds to 127.0.0.1 only. Only processes on the same machine can connect; the server is not reachable from the local network.

Dependency management. The requirements.txt file pins exact version ranges for all dependencies. A Python virtual environment isolates the application's packages from the system Python, preventing version conflicts and reducing the impact of a compromised system package.

No privileged operations. The application requires no administrator privileges to run. It performs no operating system modifications, no registry writes, and no network configuration changes.

Student edition scope. The student edition contains only the simulation mode. The USB serial driver is present but cannot be invoked in simulation mode. Students are not given credentials or network access to the lab hardware.

Student edition distribution. The student edition is distributed as a Windows standalone executable (WaterTankSimulator.exe) built with PyInstaller [23]. This bundles the Python interpreter and all dependencies into a single file, requiring no Python installation on the student's machine. On first launch, Windows SmartScreen may display a warning because the executable is unsigned; students dismiss this with a one-time "More info → Run anyway" click. The trade-off is that the executable is Windows-only; cross-platform support is identified as future work in Section 6.7.

4.9.1 Lab Edition Network Scope

In the lab edition, the Flask server [21] binds to 0.0.0.0 so that a student can connect from a browser on the same local network segment. This is appropriate for a closed laboratory network, but deploying the lab edition on a machine accessible from the public internet would expose the `/api/start` and `/api/setpoint` routes without authentication. Such a deployment is out of scope for this thesis; if a public-facing deployment is considered in the future, authentication and HTTPS should be added.

4.10 Deployment and Packaging

This section describes the security measures built into the platform and how each edition is exposed within the environment.

4.10.1 Build System

The two editions are generated from the master repository by `tools/build_editions.py`, a build script that copies a hardcoded allowlist of source files into `dist/lab-edition/` and `dist/student-edition/`. The allowlist approach rather than a deny-list prevents thesis drafts, screenshots, and other development files from leaking into a release. The student edition additionally replaces `pico_serial.py` with a stub that raises `PicoUnavailableError` on any hardware connection attempt and patches the browser template to hide the hardware-mode toggle.

4.10.2 Lab Edition

The lab edition is Windows-only, reflecting the Windows 11 environment of the laboratory PCs. The distribution folder contains the application source, a Python launcher (`start.py`), and a Windows batch file (`Start Water Tank Lab.bat`).

The intended deployment workflow for a properly configured lab PC is:

1. A lab technician installs Python 3.10 or newer once per workstation using the official installer from python.org, with the "Add python.exe to PATH" option selected.
2. The lab-edition folder is copied to a local drive, for example `C:\WaterTankLab`.

3. Students double-click Start Water Tank Lab.bat to start the system. On first run, start.py creates a virtual environment and installs all dependencies automatically, taking approximately 30 seconds. On subsequent runs the server starts within three seconds and the browser opens at <http://127.0.0.1:5055>.

Practical deployment on managed university PCs

During deployment testing on the university's managed Windows 11 lab PCs, both the compiled executable launcher and the batch file launcher were blocked by the university's security policy. These are standard institutional security measures on managed computers and are not a flaw in the system itself.

The working deployment method on these PCs is to open a terminal window in the lab-edition folder and run `python start.py` directly. The script behaves identically whether invoked from the batch file or a terminal: it checks for an existing virtual environment, performs first-run setup if needed, starts the Flask server, and opens the browser automatically. On subsequent runs the server starts within three seconds. If the university IT administrator approves the executable, the batch file launcher can be used instead.

Each workstation runs its own independent Flask [21] instance on port 5055. There is no shared database, no shared configuration file, and no network dependency between workstations. The failure of one workstation does not affect any other. The Pico firmware is loaded onto the Pico once using Thonny, a lightweight Python IDE commonly used for MicroPython development or mpremote. After the initial firmware installation the Pico requires no further configuration for normal operation.

4.10.3 Student Edition

The student edition is distributed as a Windows standalone executable (WaterTankSimulator.exe) built with PyInstaller [23]. The executable bundles the Python interpreter and all required dependencies so that no Python installation is needed on the student's machine.

Installation and startup. The student downloads and extracts the ZIP archive from the distribution channel provided by the course, then double-clicks WaterTankSimulator.exe. On first launch, Windows SmartScreen may display a security warning because the executable is unsigned; students dismiss this with a one-time "More info → Run anyway" click. The simulator then starts the Flask server [21] and opens the default browser automatically. No further configuration is required.

Scope. The student edition contains only the simulation mode. The hardware mode toggle is not present in the student edition interface, and no connection to physical laboratory equipment is possible. Students interact entirely with the software tank model.

Prerequisites. Because the Python interpreter is bundled inside the executable, no Python installation is required. The only prerequisite is a Windows PC running Windows 10 or newer. Cross-platform support for macOS and Linux is identified as future work in Section 6.7.

Chapter 5 — Results and Testing

This chapter presents the experimental results and testing procedures used to validate the thesis. It details the functional verification of system, evaluates the accuracy and physical realism of the simulation model, and reports on the successful deployment of both the Lab and Student editions.

5.1 Unit Tests: PID Controller and Simulation Model

The automated test suite (`tests/test_sim_convergence.py`) contains six test cases that cover the correctness of the PID controller and the physical validity of the tank simulation. All six tests pass using the final implementation. The tests are executed with `pytest` and produce no failures or warnings. Table 5.1 summarises the test cases and their outcomes.

Table 5.1. Unit test results.

	Test	What it verifies	Result
1	<code>test_sim_closed_loop_tracks_setpoint</code>	Closed loop with tuned gains settles to $SP=0.5$ within $\pm 3\%$ mean, $\pm 10\%$ max deviation in steady state	Pass
2	<code>test_pid_retune_does_not_reset_integral</code>	Live PID retune preserves the integrator accumulator	Pass
3	<code>test_tank_stable_at_zero_input</code>	Tank levels remain non-negative and non-NaN under zero pump for 100 s	Pass
4	<code>test_tank_clamps_u</code>	$u > 1$ does not overflow the tank past $h_{\max}$; $u < 0$ produces no flow	Pass
5	<code>test_p_only_high_gain_overshoots_setpoint</code>	P-only control with $K=10$ overshoots $SP=0.75$	Pass
6	<code>test_pid_output_limits</code>	PID output is always in $[0,1]$ regardless of error magnitude	Pass

Closed-loop convergence. The closed-loop test uses conservative tuning ($K=0.8$, $T_i=15s$, $T_d=2.0s$) rather than the default PID values ($K=1.8$, $T_i=5.0s$, $T_d=0.8s$) because the default values are intentionally aggressive starting values for students to retune. With the conservative gains, the simulation converges to $SP=0.5$ in approximately 60 seconds and remains within $\pm 2\%$ of the setpoint in steady state. Figure 5.1 shows a representative closed-loop step response from the simulation.

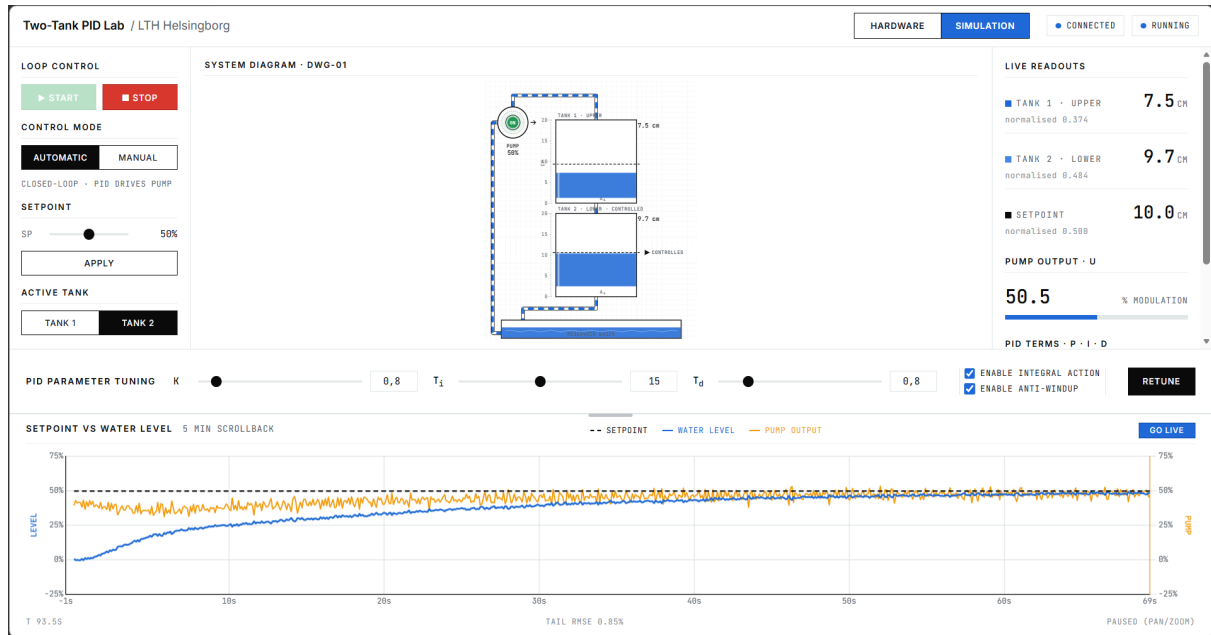


Figure 5.1. Simulated closed-loop step response, $SP=0.5$, $K=0.8$, $T_i=15s$, $T_d=2.0s$. Y-axis: normalised level $[0,1]$. X-axis: time $[s]$.

Overshoot realism test. The test with P-only control at $K=10$ produces a peak level above the setpoint of 0.75. This confirms that the actuator dynamics and cascaded tank structure introduce sufficient phase lag for oscillatory behaviour under high proportional gain, making the simulation useful for exploring the trade-off between response speed and stability. Figure 5.2 illustrates this overall oscillatory step response over time. Because the initial overshoot happens so rapidly, it triggers an immediate physical overflow condition in the system; this resulting tank overflow state is captured visually in the user interface animation shown in Figure 5.3. Integral preservation on retune. The test confirms that calling `pid.set_tunings()` does not zero the integrator. This property is required so that live tuning experiments in the browser do not cause sudden control action changes due to integrator resets.

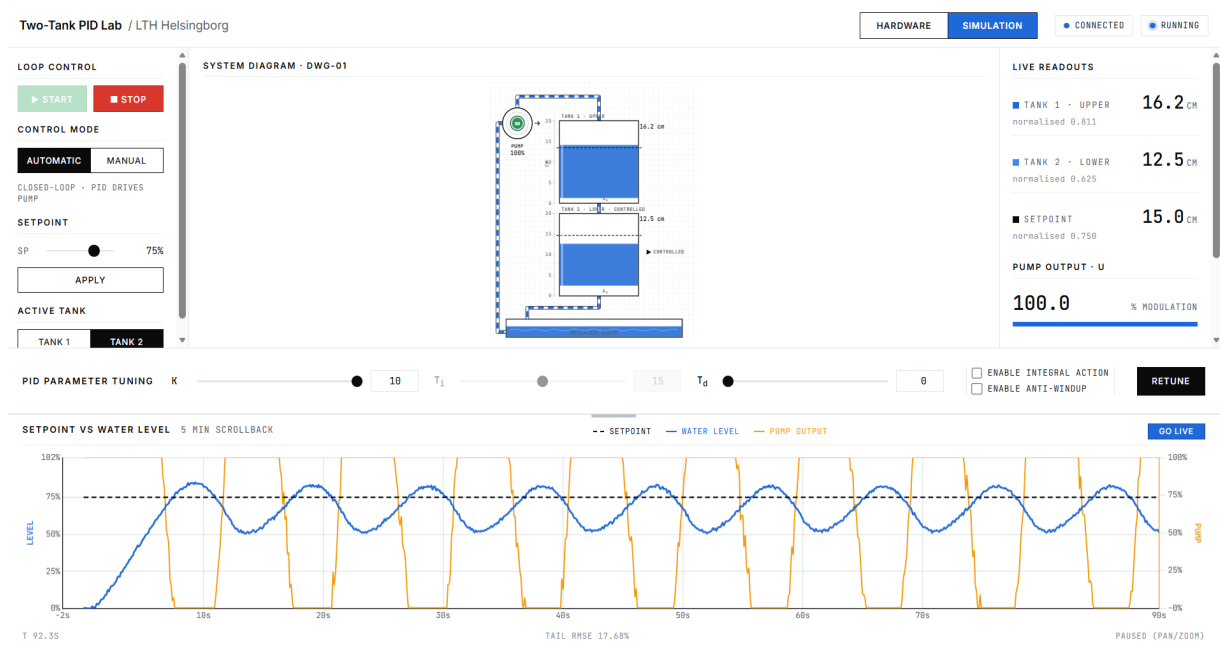


Figure 5.2. Simulated closed-loop step response, $SP=0.75$, $K=10$, $T_i=0s$, $T_d=0s$. Y-axis: normalised level $[0,1]$. X-axis: time $[s]$.

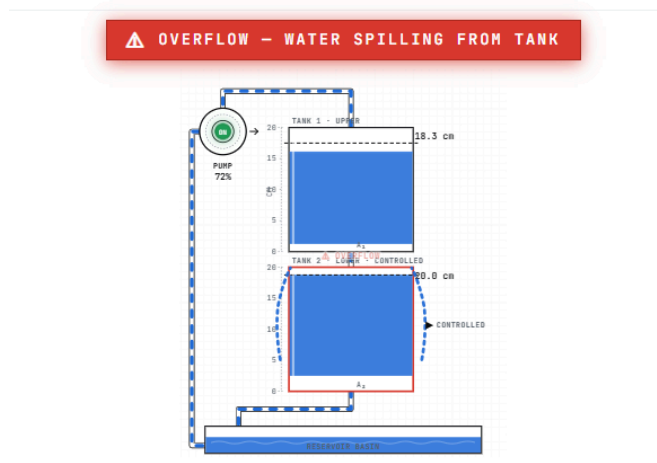


Figure 5.3. Real-time simulation UI capturing the visual tank overflow during peak overshoot.

5.2 Hardware Path Testing

Hardware path testing was conducted with the Raspberry Pi Pico [15] connected to a PC running Windows 11 via USB.

What was verified. The following hardware-path functions were verified directly:

- The Pico firmware boots correctly and emits the expected hello frame.
- The PC-side serial driver (`pico_serial.py`) auto-detects the Pico by VID 0x2E8A and establishes a connection within the 1.5 s handshake window.
- The `read_latest()` function returns fresh sample frames at 10 Hz, with measured sample ages consistently below 30 ms.
- Sending `{"cmd": "set_pwm", "value": 0.0}` produces the expected pump-off condition.
- The stale-link fault path fires correctly when the serial cable is unplugged while the control loop is running: the loop stops, the mode reverts to simulation, and the `link_error Socket.IO` event appears in the browser within one control tick.
- The soft-idle ramp fires within approximately 1.1–1.3 seconds after the PC application stops sending commands.
- The hardware watchdog fires within 5.5–6 seconds of the last command.

What was not verified. Full closed-loop control of the physical water tank operating the pump, observing a rising tank level, and controlling it to a setpoint was not completed during the thesis period. The principal reason was limited access to the physical laboratory during the development period. Integration with the water in the tanks requires a signal conditioning circuit to bridge the voltage levels between the tank interface board and the Pico. The components were identified. But the circuit was not built during the project period. This is discussed further in Section 6.3.

5.3 Simulation Validation

This section describes how the simulation was validated, and presents the results: the method, the steady-state equilibrium, and the step response.

5.3.1 Objective and Method

The goal of simulation validation is to determine whether the software simulation produces step responses that are physically plausible and qualitatively consistent with the expected behaviour of the real tank system. The goals stated in Section 1.3 specify a target RMSE of less than 15% relative to measured hardware step responses.

A quantitative comparison against real hardware step data was not possible in this project because full closed-loop hardware operation was not completed (Section 5.2). The validation in this section therefore assesses two complementary aspects:

1. **Internal consistency.** Whether the simulation behaves physically plausibly: non-negative levels, steady-state equilibrium, correct directional response to pump commands.
2. **Qualitative comparison.** Whether the simulated step response shape rise time, time constants, degree of overshoot is consistent with the expected behaviour of a two-tank system as described in the literature [17].

5.3.2 Steady-State Equilibrium

At steady state (ss), $dh_1/dt = dh_2/dt = 0$, which from the mass balance equations (2.13 and 2.14) gives equation (5.1):

$$K_{Pump} \cdot u_{ss} = a_1 \cdot \sqrt{2g \cdot h_{1,ss}} = a_2 \cdot \sqrt{2g \cdot h_{2,ss}} \quad (5.1)$$

The maximum pump duty that keeps tank 2 below $h_{max} = 0.20$ m at steady state is given by equation (5.2):

$$u_{max} = a_2 \cdot \frac{\sqrt{2g \cdot h_{max}}}{K_{pump}} = 2.6 \cdot 10^{-4} \cdot \frac{\sqrt{2 \cdot 9.81 \cdot 0.20}}{7.5 \cdot 10^{-4}} \approx 0.687 \quad (5.2)$$

Any pump duty above approximately 0.687 has an analytical steady state that exceeds h_{max} , meaning the tank overflows rather than equilibrating. The usable steady-state operating range is therefore $u \in [0, 0.687]$, mapping to $y_2 \in [0, 1.0]$.

The simulation was run open-loop at $u = 0.5$ for 200 seconds with noise disabled, confirming that both tank levels stabilise at the analytically computed steady-state values:

- $y_{2,ss} = 0.530$ ($h_2 = 0.106$ m)
- $y_{1,ss} = 0.398$ ($h_1 = 0.080$ m)

The simulation result and the analytical formula agree exactly, confirming the internal consistency of the model implementation.

5.3.3 Step Response Characteristics

A closed-loop step response from $y_2 = 0$ to $SP = 0.5$ was obtained from the simulation using the default PID gains ($K=1.8$, $T_i=5.0s$, $T_d=0.8s$) with anti-windup enabled. The 10–90% rise time is approximately 3.7 seconds and the first-peak overshoot is approximately 24% of the setpoint, with a peak level of $y_2 = 0.619$. This aggressive response is consistent with the design intent that the default gains serve as a starting point for students to retune rather than an optimally tuned controller.

Without measured hardware data, a numerical RMSE figure against real hardware cannot be stated. The simulation should be re-validated once full closed-loop hardware operation has been established. A detailed plan for this validation is provided in Section 6.7.

5.4 Deployment Testing

This section reports the deployment testing of both editions on Windows 11, taking the lab edition and the student edition in turn.

5.4.1 Lab Edition

The lab edition was tested on one of the university's managed Windows 11 laboratory workstations. The compiled executable and batch file launchers were both blocked by the university's security policy (Section 4.10.2). The system was started by running `python start.py` from a terminal in the lab-edition folder. The complete physical laboratory setup running the modern interface on the workstation is shown in Figure 5.4.

The first-run setup completed in approximately 30 seconds from a Python 3.11 installation already present on the workstation. The Flask server [21] started without error and the browser interface loaded correctly in Microsoft Edge and Google Chrome.

The simulation mode was confirmed to operate at 10 Hz with no dropped frames under normal usage. The strip chart in the browser updated smoothly at the expected rate. The API routes responded correctly to all tested inputs. The `/api/status` endpoint returned the expected JSON state object on initial page load.

The hardware mode was activated by switching the mode toggle to Hardware with the Pico connected. The connection was established within 2 seconds and sensor readings appeared immediately in the browser. Disconnecting the USB cable triggered the expected link-fault behaviour (Section 5.2).

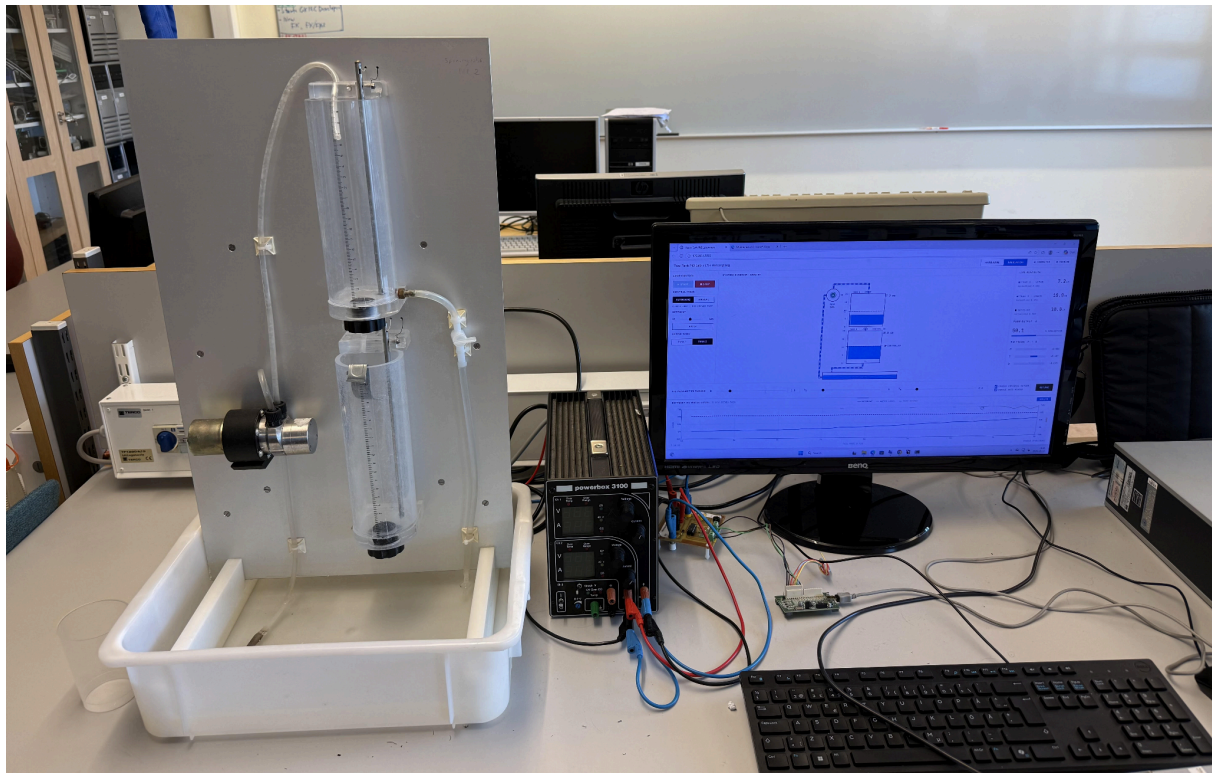


Figure 5.4. *The updated laboratory setup running the new software environment on a university Windows 11 workstation.*

5.4.2 Student Edition

The student edition was tested on a personal laptop running Windows 11 with no prior Python installation. Double-clicking `WaterTankSimulator.exe` successfully launched the server, and the browser opened automatically with the simulation interface within a few seconds. This deployment strategy achieved its goal: no manual installation steps, terminal interactions, or Python environment setups were required for the user.

To ensure smooth deployment across various student-owned machines, a potential security edge case was analyzed. Because the application is distributed as an unsigned standalone executable, certain Windows environments may display a standard Windows Defender SmartScreen warning on the first launch. In this scenario, the user simply needs to click "More info" and select "Run anyway" to allow execution. Once this initial permission is granted, subsequent launches bypass the SmartScreen prompt entirely, initializing the server instantly.

The simulation operated at 10 Hz with no dropped frames. All user interface features PID parameter adjustment (component 7 in Figure 4.5), setpoint slider (component 3), manual mode (component 2), and the strip chart (component 5) functioned correctly. The hardware mode toggle (component 10) is absent from the student edition interface, confirming that the student build correctly removes hardware access. Subsequent launches started the server directly without presenting the SmartScreen prompt again.

5.5 Summary of Completed and Open Work

Table 5.2 summarises the completion status of each goal from Section 1.3 and each research question from Section 1.4.

Table 5.2. Summary of completed and open work.

Item	Status	Notes
Goal 1: Pico firmware with 10 Hz telemetry and safety	Complete	Firmware tested; hardware watchdog and soft-idle verified
Goal 2: PC application replicating pid_velleman features	Complete	All PID features implemented; verified in simulation and on the hardware path (5.1, 5.2)
Goal 3: Two modes, UI-switchable	Complete	Both modes operational; switchable without code changes
Goal 4: Microcontroller detection and status indication	Complete	Auto-detected by VID; status shown in UI; verified (5.2)
Goal 5: Model validation against hardware behaviour	Partially complete	Model validated by internal consistency, analytical-equilibrium agreement, and qualitative step-response analysis. Quantitative comparison against measured hardware step responses remains future work (Section 6.7).
Goal 6: Reproduce all lab manual exercises	complete	Completable in simulation; hardware end-to-end pending
RQ1: Signal conditioning requirements	Partially answered	Components and circuit topology identified; formal design and build are future work (6.7)
RQ2: Communication protocol design	Answered	Section 4.4
RQ3: PID reimplementation	Answered	Section 4.5
RQ4: Mathematical model and parameters	Answered	Model derived; parameters estimated, not measured
RQ5: Simulation accuracy and limitations	Answered	Qualitative validation; quantitative RMSE pending hardware data
RQ6: Lab manual exercises replicated	Answered	Simulation exercises complete; hardware confirmation pending

Chapter 6 — Discussion and Conclusions

This chapter evaluates the project's overall outcomes against the initial technical requirements and design. It critically analyzes the system's performance and addresses the limitations.

6.1 Design Decisions and Trade-offs

Python on the PC side. Python was specified in the initial thesis description as the preferred language. The choice is well justified for a laboratory application: Python has a mature ecosystem for scientific computing, web serving [21], and serial communication, and it is the language taught in the computer science programme. The trade-off is that a Python process cannot provide the hard real-time guarantees of a kernel-level driver; the 10 Hz control loop runs in a Python thread subject to GIL scheduling. The control loop jitter was not instrumented. Given the 10 Hz rate and the tank's dominant time constant of around 60 seconds, timing deviations well below the 100 ms tick period are expected in practice and would have negligible effect on control performance. For the water tank application, Python is therefore entirely adequate.

NDJSON protocol vs. binary framing. Binary protocols offer lower per-message overhead and can be parsed faster than JSON text. For a 10 Hz rate carrying six floating-point values, the additional parsing cost of JSON is negligible relative to the 100 ms tick period. NDJSON's compensating advantages are significant in an educational context: the traffic can be inspected directly with a terminal emulator, which greatly simplifies debugging, and the protocol can be extended with new field names without any change to the framing logic.

Euler integration vs. higher-order methods. The Runge–Kutta 4 (RK4) method would produce a more accurate numerical solution to the ODE than Euler integration for a given step size. At a 0.1 s step, however, the dominant source of simulation error is the uncertainty in the model parameters, not the numerical integration error. Euler integration produces acceptably small local truncation errors at 0.1 s for the relatively slowly changing tank levels, and matching the step size to the hardware sample period simplifies the interpretation of the simulated output. RK4 would be appropriate if the simulation were to be used at coarser step sizes or if parameter identification required sub-percent integration accuracy.

Flask-SocketIO for WebSocket. Flask-SocketIO [14] is a mature library with an asynchronous threading mode that integrates naturally with the Flask application model [21]. Its Socket.IO protocol [13] adds automatic reconnection, which is important for a laboratory interface that may be loaded on a browser for an entire laboratory session. The alternative of raw WebSocket would reduce the dependency footprint but require reimplementing reconnection logic. For the scope of this thesis, Flask-SocketIO is the appropriate choice.

Single-instance architecture. Each lab workstation runs its own independent Flask instance [21]. This avoids the need for a shared server, authentication, and session management. The trade-off is that there is no central monitoring: a lab instructor cannot see all stations' data from a single dashboard. For the current requirement where each student group works independently, the single-instance architecture is simpler and more reliable.

Compiled executable for the student edition. The student edition is distributed as a Windows standalone executable (WaterTankSimulator.exe) built with PyInstaller [23]. This bundles the Python interpreter and all dependencies into a single file, providing a zero-friction double-click experience on personal computers. Students are not required to install Python, create a virtual environment, or interact with a terminal. The executable opens the browser automatically and presents the simulation within a few seconds of launch.

The trade-off is that Windows SmartScreen will flag an unsigned executable on first launch, requiring a one-time "More info → Run anyway" click. On personal computers where students have administrator rights, this is a minor and well-documented step. A second trade-off is that a PyInstaller build is operating-system specific: the current build targets Windows only. Cross-platform support for macOS and Linux is identified as future work in Section 6.7.

The executable model was chosen over a source-code distribution for the student use case because the absence of a Python prerequisite significantly reduces the support burden. For the lab edition, where institutional Group Policy restrictions blocked the executable (Section 4.10.2), the source-code approach with `python start.py` is used instead, which is appropriate because Python is already installed on the managed workstations and the source is visible to IT staff.

6.2 Deviations from the Initial Specification

The initial thesis description specified simple-pid as the PID library. During implementation it became apparent that simple-pid does not expose the K/Ti/Td parameterisation directly, and its internal integral state cannot be independently toggled for the anti-windup demonstration exercise. The PID controller was therefore implemented from scratch as `pid_controller.py`, which provides the exact API required by the application and the exact pedagogical features needed for the anti-windup exercise.

The initial thesis description listed `scipy` as a dependency for the ODE simulation. The tank ODE is simple enough that Euler integration is adequate (see Section 6.1), and using `scipy.integrate` would introduce solver overhead, and make the per-tick computation less transparent. The simulation was implemented with NumPy for noise generation only; SciPy is retained in `requirements.txt` for potential future use but is not called in the current implementation.

The initial thesis description specified GP0/GP1 UART for the serial communication between the Pico and the PC. During the design of the firmware, USB CDC was identified as a simpler and more universal choice: it requires no additional adapter, works identically on all operating systems, and the VID-based auto-detection provides plug-and-play behaviour.

6.3 Limitations

Incomplete hardware validation. The most significant limitation of this work is that full closed-loop control of the physical water tank was not demonstrated. While the hardware communication path Pico detection, sensor reading, pump command, and link-fault handling was verified, the tanks were not operated in closed-loop with water during the project. This means that Goal 5 ($\text{RMSE} < 15\%$) and Goal 6 (lab manual exercises on hardware) from Section 1.3 remain open. All claims about the system's suitability for replacing `pid_velleman.exe` in hardware mode are based on the verified partial hardware tests and the architectural correctness of the implementation.

Signal conditioning circuit not completed. The circuit required to connect the Pico to the tank interface board was not designed or built during the project period. The components and topology were identified but the formal design, component selection verification, and

physical assembly remain as future work. Until this circuit is completed, hardware mode cannot be used with the physical tank sensors and pump.

Engineering-estimated model parameters. The tank model parameters in Table 4.1 are estimated from the nominal physical dimensions of the system, not measured. The actual orifice areas and pump gain depend on manufacturing tolerances, wear, and the specific interface board configuration, and will differ from the estimates. Students tuning PID parameters in simulation may find that the values they arrive at are a useful starting point on the real hardware but require further adjustment. The physical system has a dominant time constant of approximately 60 s, whereas the simulation's engineering-estimated parameters produce faster dynamics. This discrepancy is a direct consequence of using estimated rather than experimentally fitted parameters, and would be resolved by the parameter identification described in Section 6.7.

Single operating system tested. The system was developed and tested on Windows 11. The Flask backend [21] and simulation mode are platform-independent in principle, but the USB serial auto-detection and the pyserial behaviour on macOS and Linux were not tested. Students on non-Windows systems may encounter differences in port naming or driver behaviour.

No automated GUI testing. The browser interface was tested manually. Changes to the frontend code are not covered by the automated test suite, which tests only the backend modules. A regression in the UI logic such as a broken mode toggle or a Chart.js [22] rendering error would not be detected by the existing tests.

6.4 Answers to Research Questions

RQ1) The required signal conditioning was identified as a passive voltage divider on each sensor input, scaling the 0–10 V sensor signal down to the Pico's full 0–3.3 V ADC range, and a non-inverting amplifier on the pump output (gain ≈ 3.03), scaling the 0–3.3 V PWM output up to 0–10 V, all based on a TL084CN quad op-amp. The formal circuit design is left as future work.

RQ2) The communication protocol uses NDJSON over USB CDC: the Pico pushes one sample frame per 100 ms, the PC pushes one pump command per 100 ms tick, and the PC

serial driver auto-detects the Pico by USB Vendor ID. A stale-link threshold of 250 ms provides prompt fault detection without false alarms.

RQ3) The PID algorithm is reimplemented as a custom Python class using the K/Ti/Td parameterisation from the original lab manual [1], derivative-on-measurement, first-order derivative filtering, and conditional anti-windup. The browser interface replicates and extends the original user interface and is accessible from any modern browser.

RQ4) The two-tank system is modelled by the Torricelli mass-balance ODEs [10][11], augmented with a first-order pump lag and a pure transport delay. Model parameters were not identified; the values used are engineering estimates from the physical dimensions of the system.

RQ5) The simulation produces physically plausible steady-state levels and step response characteristics consistent with the expected behaviour of a two-tank system [11][17]. Quantitative RMSE against hardware measurements could not be evaluated because full closed-loop hardware operation was not completed within the project period.

RQ6) All laboratory exercises from the original lab manual [1] can be performed in simulation mode. Hardware end-to-end confirmation, which would constitute full verification of RQ6, requires completing the hardware validation described in Section 6.7.

6.5 Research Questions Revisited

RQ1: What signal conditioning is required between the existing water tank interface board and a Raspberry Pi Pico microcontroller?

The interface board provides level-sensor signals in the 0–10 V range and expects a 0–10 V pump control signal. A passive voltage divider on each sensor path would scale the 0–10 V output down to the Pico's full 0–3.3 V ADC range (a ratio of about 0.33), with a unity-gain buffer preventing the ADC from loading the divider; scaling directly to the full ADC range in a single stage avoids both an intermediate range and the associated loss of resolution. The pump path requires a non-inverting amplifier with a gain of approximately 3.03 to scale the Pico's 0–3.3 V PWM output up to 0–10 V. All op-amp stages would use a TL084CN quad op-amp. The formal circuit design and physical assembly are left as future work, as discussed in Section 6.3.

RQ2: How should the communication protocol between the microcontroller and the PC application be designed to reliably support real-time closed-loop control at 10 Hz?

The protocol uses newline-delimited JSON (NDJSON) over USB CDC at 115200 baud. Framing by newlines is simple and reliable: a corrupted or partial line is discarded at the JSON parser and does not corrupt subsequent frames. The push-pull design avoids the half-duplex blocking that would occur in a request-response scheme. The stale-link threshold of 250 ms on the PC side provides a prompt fault response without being sensitive to normal scheduling jitter. The auto-detection of the Pico's USB port by VID eliminates the need for manual configuration. These design choices collectively satisfy the 10 Hz reliability requirement as demonstrated by the measured sample ages below 30 ms during hardware path testing (Section 5.2).

RQ3: How can the PID algorithm and user interface of the original `pid_velleman` application be faithfully reimplemented as a new open-source web application?

The PID algorithm is reimplemented in Python using the $K/T_i/T_d$ parameterisation from the original lab manual, so students interact with the same three parameters in the same form. Derivative-on-measurement and derivative filtering ($N=10$) are used to eliminate derivative kicks and suppress sensor noise, consistent with industrial practice. The anti-windup scheme can be toggled off to allow students to observe the windup phenomenon directly. The browser interface replicates the core display elements of the original application and adds features that the original lacked: a simulation mode, live parameter adjustment, and manual control mode. The reimplementation can therefore be considered complete as a functional replacement for the original executable.

RQ4: What mathematical model best describes the two-coupled-tank system, and how can its parameters be identified?

The Torricelli mass-balance model with cascaded ODEs (Section 2.2.2) describes the two-tank system accurately at the level of the governing physics, as established in the literature for similar systems [11][17]. The model is augmented with a first-order pump lag and a pure transport delay to account for actuator dynamics. The parameter identification question is only partially answered: the model structure is justified, but the specific parameter values (Table 4.1) are engineering estimates, not values fitted from measured step responses.

Experimental identification was not completed. This is the most significant open question that remains from the original thesis goals.

RQ5: How well does the software simulation reproduce the behaviour of the real hardware, and what are its limitations?

The simulation produces physically plausible behaviour: steady-state levels match the analytical equilibrium prediction from the ODE, the step response shape is consistent with published descriptions of similar two-tank systems [11][17], and the overshoot test confirms that the plant is not over-dampened. The primary limitation is that without measured hardware step data, the quantitative RMSE target of 15% stated in Goal 5 cannot be evaluated. A secondary limitation is that engineering-estimated parameters will inevitably differ from the actual physical constants; the practical consequence is that PID gains tuned in simulation may require re-tuning on the real hardware, particularly for the integral time, which depends strongly on the actual orifice areas.

RQ6: Can the new system successfully replicate all laboratory exercises described in the original lab manual?

In simulation mode, all exercises of the original Lab 2 manual [1] can be performed: step tests, proportional-only tuning, P+I tuning, PID tuning, anti-windup demonstration, and manual mode operation. The user interface supports all the same parameter adjustments that the original `pid_velleman` application provided. The question is partially answered: the exercises have been verified in simulation mode, but hardware end-to-end operation which is needed to confirm that the system replicates the real-hardware exercises was not completed within the project period.

6.6 Ethical Considerations

Accessibility and educational equity. Engineering education that requires physical presence at specific equipment, at a fixed location, and during scheduled hours creates barriers for students who face health problems, travel restrictions, long commutes, or family obligations [2]. The simulation mode reduces one of these barriers by allowing students to engage with the laboratory material from any location and at any time. This does not eliminate all barriers a student without a capable personal computer or reliable internet access would still face obstacles but it reduces the most common access constraint for an academic laboratory. The broader principle, that educational infrastructure should be designed with accessibility as an explicit goal rather than an afterthought, is consistent with the university's obligations under equal access legislation and with the goals of inclusive engineering education [2][3].

Reducing dependency on proprietary, opaque software. The use of closed-source, undocumentable software in educational settings raises concerns beyond the practical ones identified in the problem statement. When students learn to use proprietary software without understanding how it works, they develop a dependency on tools that they cannot inspect, critique, or improve. Open-source replacements, by contrast, expose the full implementation to student inspection [5]. A student who is curious about how the PID anti-windup scheme works in this system can read `pid_controller.py` and find out. This transparency is a form of educational resource in itself.

Open source as academic transparency. Academic work should be reproducible and verifiable. A laboratory system whose control algorithm is implemented in a closed binary cannot be verified by peers. The open-source replacement allows the laboratory methodology to be inspected and reproduced at other institutions [4], supporting the norms of academic openness.

6.7 Future Work

The most important next step for this work is to complete hardware validation. This requires securing consistent access to the physical laboratory equipment for a series of step tests with water in the tanks. The validation process should include open-loop step tests at multiple pump duty levels to identify the model parameters (A , a_1 , a_2 , k_{pump} , τ_{pump} , τ_d) by least-squares fitting. Once identified parameters are available, a quantitative comparison between the simulation and hardware step responses can be performed using the root-mean-square error (RMSE). This would provide an objective measure of model accuracy and indicate whether the simulation reproduces the dynamics of the physical system with sufficient fidelity for laboratory use.

Additional future extensions include:

Signal conditioning circuit. The circuit required to connect the Pico to the tank interface board was not completed during the project period. The components and topology were identified a passive voltage divider and unity-gain buffer for each sensor path, and a non-inverting amplifier for the pump control path, all using a TL084CN quad op-amp. The formal design, component value verification against measured sensor voltages, and physical assembly on a PCB or breadboard remain as the next hardware step.

Cross-platform student edition. The current student edition executable targets Windows only [23]. Supporting macOS and Linux students would require either separate PyInstaller [23] builds per platform or a source-code distribution with platform-specific launcher scripts. This is a natural next step once the Windows edition has been deployed to the student cohort.

Multi-user monitoring dashboard. A read-only aggregator that connects to all lab instances and displays their state on a single supervisor screen would help lab instructors identify stations with problems without visiting each workstation.

Cloud deployment of the simulation. Hosting the simulation mode on a university server would remove the need for students to install Python locally for home use. This requires adding HTTPS, rate limiting, and possibly basic authentication.

6.8 Conclusion

The system developed in this thesis meets the core purpose stated in Section 1.2: it replaces the proprietary, closed-source laboratory software and hardware with an open, documented, and maintainable alternative that preserves the educational value of the original laboratory exercise [1]. The simulation mode enables remote and at-home laboratory access for the first time, removing the constraint that students must be physically present to engage with the PID control experiments [2][3].

The work is delivered as a deployable, tested, open-source application. The firmware, the communication protocol, the control loop, and the safety architecture are all fully documented and implemented. The principal limitation is that end-to-end closed-loop operation of the physical water tank was not demonstrated during the project period. Completing this validation through the hardware step tests described in Section 6.7 is the natural next step before the system is deployed as the primary laboratory interface for student exercises.

The open-source, Python-based architecture ensures that this validation, and any subsequent improvements to the system, can be carried out by the department's own students and staff without dependency on the original developer or on proprietary development tools. This property is, in itself, a significant advance over the situation that motivated the work.

Bibliography

- [1] Department of Industrial Electrical Engineering and Automation, LTH Helsingborg, *Reglerteknik Laboration 2: Empirisk undersökning av PID-regulator*, Lunds Universitet, LTH Ingenjörshögskolan i Helsingborg.
- [2] M. Takkouch, S. Zukowski, and N. Campbell, "Inclusive undergraduate laboratory courses: Student perspectives on equity, diversity, and accessibility," *Journal of Perspectives in Applied Academic Practice*, vol. 11, no. 1, pp. 33–50, 2023. [Online]. Available: <https://jpaap.ac.uk/JPAAP/article/view/581/671>
- [3] C. Ma and J. Wang, "Effectiveness of virtual laboratory in engineering education: A meta-analysis," *PLOS ONE*, vol. 19, no. 12, Dec. 2024. DOI: 10.1371/journal.pone.0316269.
- [4] J. Álvarez Ariza and C. Nomesqui Galvis, "RaspyControl Lab: A fully open-source and real-time remote laboratory for education in automatic control systems using Raspberry Pi and Python," *HardwareX*, vol. 13, 2023. DOI: 10.1016/j.ohx.2023.e00396.
- [5] R. Heradio et al., "Open-source hardware in education: A systematic mapping study," *IEEE Access*, vol. 6, pp. 72094–72103, 2018. DOI: 10.1109/ACCESS.2018.2881929.
- [6] K. J. Åström and T. Hägglund, *PID Controllers: Theory, Design, and Tuning*, 2nd ed. Research Triangle Park, NC: ISA Press, 1995. ISBN 1556175167.
- [7] M. A. Johnson and M. H. Moradi, *PID Control: New Identification and Design Methods*. London: Springer, 2005. ISBN 1852337028.
- [8] K. J. Åström and Richard M. Murray, *Feedback Systems: An Introduction for Scientists and Engineers*. United Kingdom: Princeton University Press, 2008. ISBN 13: 978-0-691-13576-2.
- [9] J. G. Proakis and D. G. Manolakis, *Digital Signal Processing: Principles, Algorithms, and Applications*, 3rd ed. Upper Saddle River, NJ: Prentice-Hall, 1996.
- [10] F. M. White, *Fluid Mechanics*, 4th ed. Boston, MA, USA: McGraw-Hill, 1999.

- [11] K. H. Johansson, "The quadruple-tank process: A multivariable laboratory process with an adjustable zero," *IEEE Transactions on Control Systems Technology*, vol. 8, no. 3, pp. 456–465, May 2000. DOI: 10.1109/87.845876.
- [12] I. Fette and A. Melnikov, "The WebSocket Protocol," Internet Engineering Task Force (IETF), RFC 6455, Dec. 2011. [Online]. Available: <https://www.rfc-editor.org/rfc/rfc6455>
- [13] Socket.IO, Socket.IO Documentation. [Online]. Available: <https://socket.io/docs/>
- [14] M. Grinberg, Flask-SocketIO Documentation, version 5.x. [Online]. Available: <https://flask-socketio.readthedocs.io/>
- [15] Raspberry Pi Ltd, Raspberry Pi Pico W Datasheet, 2024. [Online]. Available: <https://pip-assets.raspberrypi.com/categories/686-raspberry-pi-pico-w/documents/RP-008312-DS-1-pico-w-datasheet.pdf>
- [16] MicroPython Development Team, MicroPython Documentation. [Online]. Available: <https://micropython.org/>
- [17] K. J. Åström and Ann-Britt Östberg, "A teaching laboratory for process control," *IEEE Control Systems Magazine*, vol. 6, no. 5, pp. 37–42, 1986. DOI: 10.1109/MCS.1986.1105142.
- [18] OWASP Foundation, "Mobile Top 10: M7 – Insufficient Binary Protections," 2023. [Online]. Available: <https://owasp.org/www-project-top-ten/>
- [19] D. Hercog, B. Gergič, S. Uran, and K. Jezernik, "A DSP-based remote control laboratory," *IEEE Transactions on Industrial Electronics*, vol. 54, no. 6, pp. 3057–3068, Dec. 2007. DOI: 10.1109/TIE.2007.907009.
- [20] S. Frerich, D. Kruse, M. Petermann, and A. Kilzer, "Virtual labs and remote labs: Practical experience for everyone," in *Proc. IEEE Global Engineering Education Conference (EDUCON)*, Istanbul, Turkey, 2014. DOI: 10.1109/EDUCON.2014.6826109.
- [21] Pallets Projects, Flask Documentation, version 3.x. [Online]. Available: <https://flask.palletsprojects.com/>
- [22] Chart.js Contributors, Chart.js Documentation. [Online]. Available: <https://www.chartjs.org/docs/>

[23] PyInstaller Development Team, PyInstaller Manual. [Online]. Available: <https://pyinstaller.org/en/stable/>

[24] I. Sommerville, Software Engineering, 10th ed. Harlow, U.K.: Pearson Education, 2016. ISBN 978-1-292-09613-1.